INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700  800/521-0600
CONTENTION FREE SYNCHRONIZATION
IN
SHARED MEMORY MULTIPROCESSORS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By

Muhammad Arshad, M.S.

*****

The Ohio State University
1997

Dissertation Committee:
Prof. J.E. DeGroat, Co-advisor
Prof. F. Ozguner, Co-advisor
Prof. M. Singhal

Approved by
Co-advisor

Department of Electrical Engineering
ABSTRACT

This research focuses on contention free synchronization techniques in shared memory multiprocessors. We provide new contention free algorithms for the mutual exclusion and readers-writers locks, and concurrent queue implementations for the producer-consumer problem. In shared memory multiprocessors, processors may synchronize by busy waiting or blocking themselves. Busy waiting on global flags is known to cause contention. It is essential to eliminate contention due to synchronization to realize the performance and scalability of shared memory multiprocessors. Recognizing contention as a problem, contention free solutions have been developed in the past. However, most of these solutions rely on cache coherency or use powerful atomic primitives, e.g., compare&swap. There are many industry standard buses that provide local shared memory but do not support cache coherency. Likewise, there are many processors that support swap or fetch&add but do not support the more powerful atomic primitives.

New or improved solutions using local shared memory and the more widely supported swap or fetch&add atomic primitives have practical significance in that they have wider applicability. We have developed a number of solutions that meet this criteria. Our algorithms use the swap (or fetch&add) atomic primitive, guarantee FIFO behavior, and eliminate contention due to busy waiting by spinning in the local shared memory. They work equally well in architectures with or without coherent caches.

We observe that any two processors in a system of N processors may synchronize in a contention free manner using swap or fetch&add and a single synchronization variable. We further observe that more than one synchronization variable may be combined for contention free synchronization of more than two processors. We exploit this fact to build data structures which we then use to develop new algorithms.
We have developed three mutual exclusion locks: an array based lock using \texttt{fetch\&add}, another array based solution using \texttt{swap}, and a linked list based solution using \texttt{swap} as the only atomic primitive. The solutions are efficient, scalable, and provide guaranteed FIFO behavior. The first two extend the previous array-based locks to architectures \textit{without} coherent caches. The original solutions relied on cache coherency to eliminate contention. One of these array based solutions has also been extended by another author providing similar capabilities. The third solution extends a linked list based lock. It uses local spinning to eliminate contention, and ensures FIFO behavior using \texttt{swap} as the only atomic primitive. The original linked list based solution required \texttt{compare\&swap} to guarantee the FIFO behavior.

To our knowledge no contention free solution has been attempted to the readers-writers problem using \texttt{swap} or \texttt{fetch\&add} as the only primitives. We have developed two array based solutions using \texttt{fetch\&add}. We have also developed a singly linked list and three doubly linked list based solutions using \texttt{swap} as the only atomic primitive. All of our solutions guarantee FIFO behavior. One of our array based solutions relies on coherent cache. The rest of the solutions use local shared memory to eliminate contention.

We have also developed lock free solutions to the producer-consumer problem. Using \texttt{swap} as the only atomic primitive we have developed a linked list based solution for multiple producers, single consumer. Previous solutions have required \texttt{compare\&swap} to provide the same capability that we provide using \texttt{swap}. We have also developed array based producer-consumer solutions. These solutions offer a bounded buffer and are characterized by the fact that no producer or consumer is required to wait.

We have measured the performance of our algorithms on a shared bus shared memory multiprocessor system that does not support a coherent cache. The results verify that our algorithms do eliminate contention due to synchronization in architectures \textit{without} coherent caches.
To Saaid and Raaid
ACKNOWLEDGMENTS

I would like to thank my advisor Prof. J.E. DeGroat for her guidance, support, encouragement, and understanding throughout this work. I also wish to thank the members of the dissertation committee and Prof. G.J. Aubrecht, the graduate school representative, for reviewing the dissertation and for providing valuable feedback.

I would like to thank the management at Lucent Technologies Bell Laboratories for supporting me in my pursuit of the doctoral degree here at OSU. In particular, I wish to extend my thanks to G.J. Watkins, Jr., S. Srinivas, and P.V. Lessek for formally approving the study program and M. Jameel, D.M. Rouse, and P. Shah for ensuring its completion. I would also like to express my sincere thanks to M. Jameel and D.M. Rouse for their constant support, encouragement, and involvement at the personal level throughout the stretched period of my studies.

Thanks are due to friends and colleagues who encouraged me and supported me at times my spirit was down. In particular, I would like to express my gratitude to M.A. Tamny for his support and review of a number of the algorithms and A. Khurshid for his support and valuable suggestions.

I am indebted to many of my other friends and colleagues here at Lucent Technologies for their help in many different ways. I would like to thank S.W. Steenburgh who set up the system, J. J. Howe who built an instance of the OS tailored to my needs, and M.A. Kelly and J. Akula who provided support for networking. I could not have possibly completed the performance measurements work without their help. I would also like to express my thanks to R.A. Widlicka, R.C. Lian, and L.F. Chang for their help and patience as I frequently bothered them with many questions related to the system I used for performance measurements. Thanks are also due to E.W. Krause for his help and effort in trying to locate and M. Bohus for providing the circuit packs to setup a system that supported
atomic swap in the hardware. I am also very grateful to D.L. Perry for helping me with a number of Framemaker questions during the writing of the dissertation.

Lastly, I want to express my thanks to my wife who stood by me in this prolonged pursuit of higher education. It was a struggle to juggle priorities between work, family, and the studies and without her support I may never have succeeded in this endeavor. My deepest thanks to her for doing all she did over the years including taking our two little boys on a visit to Pakistan all by herself to give me a few months in which I could concentrate and finish the work. To acknowledge the sacrifice that little Saaid and Raaid had to make, I have dedicated this work to them.

I would also like to thank my father-in-law A.H. Khan who, any time we need help, comes all the way from Pakistan and stays with us. His extended stays during last couple of years have been very helpful in enabling me to spend the necessary time to complete the dissertation work.
VITA

Apr. 10, 1954.......................... Born, Arifwala, District Sahiwal, Pakistan

Dec. 1981............................. M.S. Electrical Engineering
                               The Ohio State University, Columbus, Ohio

Feb. 1982 - Sep. 1983.......... Project Engineer,
                               Intek Inc. Columbus, Ohio

Sep. 1983 - Nov. 1983......... Design Engineer,
                               NCR, Cambridge, Ohio

Nov. 1983 - Aug. 1995.......... Member of Technical Staff
                               AT&T Bell Laboratories, Columbus, Ohio

Sep. 1995 - Present ............ Distinguished Member of Technical Staff
                               Lucent Technologies Bell Laboratories,
                               Columbus, Ohio

PUBLICATIONS

1. Arshad, M., and DeGroat, J.E., "Concurrent Updates of Events List for Parallel
   VHDL Simulations", NOTE BOOK OF SESSIONS, VIUF Spring 1996 Conference.

FIELDS OF STUDY

Major Field: Electrical Engineering
Minor Field: Computer and Information Science
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ii</td>
</tr>
<tr>
<td>DEDICATION</td>
<td>iv</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>v</td>
</tr>
<tr>
<td>VITA</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xli</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xlii</td>
</tr>
<tr>
<td>CHAPTERS:</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Shared Memory Multiprocessors</td>
<td></td>
</tr>
<tr>
<td>1.1.1 A Classification of Computer Architectures</td>
<td>3</td>
</tr>
<tr>
<td>1.1.2 Shared Memory Multiprocessor Architectures</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Attributes of Shared Information</td>
<td>7</td>
</tr>
<tr>
<td>1.2.1 Atomicity of Data</td>
<td>7</td>
</tr>
<tr>
<td>1.2.2 Atomicity of Update</td>
<td>8</td>
</tr>
<tr>
<td>1.3 Synchronization Problems in Shared Memory Multiprocessors</td>
<td>9</td>
</tr>
<tr>
<td>1.3.1 The Mutual Exclusion Problem</td>
<td>11</td>
</tr>
<tr>
<td>1.3.2 The Readers-Writers Problem</td>
<td>13</td>
</tr>
<tr>
<td>1.3.3 The Producer-Consumer Problem</td>
<td>14</td>
</tr>
<tr>
<td>1.3.4 The Barrier Synchronization Problem</td>
<td>16</td>
</tr>
<tr>
<td>1.4 Synchronization Mechanisms</td>
<td>17</td>
</tr>
<tr>
<td>1.4.1 Hardware Atomic Updates</td>
<td>17</td>
</tr>
<tr>
<td>1.4.2 Software Solutions to Synchronization Problems</td>
<td>18</td>
</tr>
<tr>
<td>1.5 The Problem Statement</td>
<td>21</td>
</tr>
<tr>
<td>1.6 Organization of the Rest of the Dissertation</td>
<td>21</td>
</tr>
</tbody>
</table>

viii
4.2 Related Work .................................................................................................... 113
4.3 Array-Based Solutions to the Producer-Consumer Problem .............. 115
  4.3.1 Observations about Arrays ............................................................... 115
  4.3.2 Single Producer, Single Consumer .................................................. 118
  4.3.3 Multiple Producers, Single Consumer ............................................ 122
  4.3.4 Single Producer, Multiple Consumers ............................................ 128
  4.3.5 Multiple Producers, Multiple Consumers ..................................... 130
4.4 Linked List Based Solutions to the Producer-Consumer Problem .... 135
  4.4.1 Multiple Producers, Single Consumer ............................................ 135
  4.4.2 Multiple Producers, Multiple Consumers ....................................... 142
4.5 Summary ......................................................................................................... 142

5. PERFORMANCE MEASUREMENTS ................................................................. 144
  5.1 Introduction ..................................................................................................... 144
  5.2 System Setup .................................................................................................. 144
    5.2.1 A VME Based Shared Memory System ........................................... 145
    5.2.2 Hardware Limitations ........................................................................ 146
    5.2.3 Emulation of Atomic Swap and Fetch&Add ................................ 147
    5.2.4 Effect of the VME Arbitration ............................................................ 151
  5.3 Performance Measurements .......................................................................... 152
    5.3.1 Factors Affecting the Performance .................................................. 152
    5.3.2 Details of How the Performance is Measured ................................154
  5.4 Measurement Results .................................................................................... 156
    5.4.1 Mutual Exclusion Locks .................................................................... 156
    5.4.2 Readers-Writers Locks................................................................. 161
  5.5 Summary ......................................................................................................... 174

6. SUMMARY AND CONCLUSIONS ............................................................................. 175
  6.1 Summary ......................................................................................................... 175
  6.2 Performance ..................................................................................................... 176
  6.3 Recommendations .......................................................................................... 177
  6.4 Suggested Future Work .................................................................................. 179

BIBLIOGRAPHY ...................................................................................................................... 180
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Characteristics of SPARC CPU-2CE</td>
<td>145</td>
</tr>
<tr>
<td>2</td>
<td>Normalized Per Lock Time (µs) with 16 µs Loop in CS</td>
<td>158</td>
</tr>
<tr>
<td>3</td>
<td>Normalized Per Lock Time (µs) with Mutual Exclusion Check in CS</td>
<td>158</td>
</tr>
<tr>
<td>4</td>
<td>Per Lock Times (µs) for 16 µs Loop in CS for RW.AR-1</td>
<td>163</td>
</tr>
<tr>
<td>5</td>
<td>Per Lock Times (µs) for Mutual Exclusion Check in CS for RW.AR-1</td>
<td>163</td>
</tr>
<tr>
<td>6</td>
<td>Per Lock Times (µs) for 16 µs Loop in CS for RW.AR-2</td>
<td>164</td>
</tr>
<tr>
<td>7</td>
<td>Per Lock Times (µs) for Mutual Exclusion Check in CS for RW.AR-2</td>
<td>164</td>
</tr>
<tr>
<td>8</td>
<td>Per Lock Times (µs) for 16 µs Loop in CS for RW.SL-1</td>
<td>165</td>
</tr>
<tr>
<td>9</td>
<td>Per Lock Times (µs) for Mutual Exclusion Check in CS for RW.SL-1</td>
<td>165</td>
</tr>
<tr>
<td>10</td>
<td>Per Lock Times (µs) for 16 µs Loop in CS for RW.DL-1</td>
<td>166</td>
</tr>
<tr>
<td>11</td>
<td>Per Lock Times (µs) for Mutual Exclusion Check in CS for RW.DL-1</td>
<td>166</td>
</tr>
<tr>
<td>12</td>
<td>Per Lock Times (µs) for 16 µs Loop in CS for RW.DL-2</td>
<td>167</td>
</tr>
<tr>
<td>13</td>
<td>Per Lock Times (µs) for Mutual Exclusion Check in CS for RW.DL-2</td>
<td>167</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A Bus Based Shared Memory Multiprocessor System</td>
</tr>
<tr>
<td>2</td>
<td>An Interconnection Network Based Shared Memory Architecture</td>
</tr>
<tr>
<td>3</td>
<td>An Abstraction of the Mutual Exclusion Problem</td>
</tr>
<tr>
<td>4</td>
<td>An Abstract View of the Producer-Consumer Problem</td>
</tr>
<tr>
<td>5</td>
<td>A Hierarchy of Synchronization Mechanisms</td>
</tr>
<tr>
<td>6</td>
<td>A Software Combining Tree</td>
</tr>
<tr>
<td>7</td>
<td>Data Structure for Anderson’s Array Based Queuing Lock</td>
</tr>
<tr>
<td>8</td>
<td>A Variant of Anderson’s Array Based Queuing Lock</td>
</tr>
<tr>
<td>9</td>
<td>Data Structure for Graunke and Thakkar’s Queuing Lock</td>
</tr>
<tr>
<td>10</td>
<td>A Variant of Graunke and Thakkar’s Queuing Lock</td>
</tr>
<tr>
<td>11</td>
<td>Data Structures for Mellor-Crummey and Scott’s Queuing Lock</td>
</tr>
<tr>
<td>12</td>
<td>Mellor-Crummey and Scott’s Queuing Lock</td>
</tr>
<tr>
<td>13</td>
<td>Release in Mellor-Crummey and Scott’s Lock Using <em>Swap</em></td>
</tr>
<tr>
<td>14</td>
<td>A Scenario Leading to Loss of FIFO Behavior</td>
</tr>
<tr>
<td>15</td>
<td>Resolving a Resource Request Between Two Processors</td>
</tr>
<tr>
<td>16</td>
<td>Data Structure for the Array Based Lock Using <em>Fetch&amp;Add</em></td>
</tr>
<tr>
<td>17</td>
<td>Code for the Array Based Lock (μE-1)</td>
</tr>
<tr>
<td>18</td>
<td>Data Structure for the Array/Linked List Based Lock Using <em>Swap</em></td>
</tr>
<tr>
<td>19</td>
<td>Code for the Array/Linked List Based Lock (μE-2)</td>
</tr>
<tr>
<td>20</td>
<td>Data Structure for the List Based Lock Using <em>Swap</em></td>
</tr>
<tr>
<td>21</td>
<td>Code for the Linked List Based Lock (μE-3)</td>
</tr>
<tr>
<td>22</td>
<td>Preserving the FIFO Behavior Using <em>Swap</em></td>
</tr>
<tr>
<td>23</td>
<td>Data Structure for the Lock by Krieger, et. al.</td>
</tr>
<tr>
<td>24</td>
<td>A Reader Acquiring the Lock in the Lock by Krieger, et. al.</td>
</tr>
<tr>
<td>25</td>
<td>Data Structure for the Array Based Lock (RW.AR-1)</td>
</tr>
<tr>
<td>26</td>
<td>Initialization of the Array Based Lock (RW.AR-1)</td>
</tr>
<tr>
<td>27</td>
<td>Code for the Array Based Readers-Writers Lock (RW.AR-1)</td>
</tr>
</tbody>
</table>
Code for A Writer in the Array Based Lock (RW.AR-2) .............................................. 77
Code for A Reader in the Array Based Lock (RW.AR-2) .............................................. 78
An Instance of Data Structure Showing Readers Only .............................................. 80
Data Structure for the Singly Linked List Based Lock............................................ 83
Data Structure - Using Singly Linked Lists & Count ............................................... 84
Code to Initialize the Singly Linked List Based Lock (RW.SL-1) ............................. 85
Code to Acquire the Singly Linked List Based Lock (RW.SL-1) ............................. 85
Code to Release the Singly Linked List Based Lock (RW.SL-1) ............................ 86
Data Structures for RW.DL-1 ................................................................................... 90
An Instance of Data Structure for RW.DL-1 ............................................................ 91
Code to Initialize the Lock in RW.DL-1 ................................................................ 92
Algorithms to Acquire the Lock in RW.DL-1 ......................................................... 93
Algorithms to Release the Lock in RW.DL-1 ........................................................ 94
Data Structure for the Global Shared Memory for RW.DL-2 ................................. 95
Code to Initialize the Lock in RW.DL-2 ................................................................ 96
Algorithm for a Reader to Acquire the Lock in RW.DL-2 ...................................... 98
Algorithm for a Writer to Acquire the Lock in RW.DL-2 ........................................ 99
Algorithm for a Reader to Release the Lock in RW.DL-2 ..................................... 100
Algorithm for a Writer to Release the Lock in RW.DL-2 ...................................... 101
Unlinking from a Doubly Linked List .................................................................... 103
Data Structures for RW.DL-3 .................................................................................. 105
An Instance of the Data Structure for RW.DL-3 .................................................... 105
Algorithm for a Reader to Release the Lock in RW.DL-3 ..................................... 107
An Abstract View of the Producer-Consumer Problem ........................................ 112
Mapping an Unsigned Integer Onto a Limited Range of Indexes .......................... 117
Array Based Single Producer, Single Consumer Data Structure .......................... 118
Array-Based Single Producer, Single Consumer .................................................. 119
Array Based Single Producer, Single Consumer - Alternative Solution .............. 121
Conditional Atomic Add Using Compare&Swap ..................................................... 124
Conditional Atomic Add Using Fetch&Add .............................................................. 125
Array Based Multiple Producers, Single Consumer Data Structure .................... 126
Array Based Multiple Producers, Single Consumer ............................................. 127
Array Based Single Producer, Multiple Consumers Data Structure .................... 128
Array Based Single Producer, Multiple Consumers ............................................. 129
Array Based Multiple Producers, Multiple Consumers Data Structure ............. 132
CHAPTER 1

INTRODUCTION

This research focuses on contention free synchronization techniques in shared memory multiprocessors. We provide new contention free algorithms for the mutual exclusion and readers-writers locks, and concurrent queue implementations for the producer-consumer problem.

Busy waiting or blocking are used for processor synchronization in shared memory multiprocessors. Concurrent objects offer another alternative. Busy waiting on global flags is known to cause contention [2][7][8][32][63][73][86][108]. Likewise, a concurrent implementation of an object, in which a processor may have to retry an operation, may also cause contention. It is necessary to eliminate contention due to synchronization to realize the performance and scalability of shared memory multiprocessors. Recognizing contention as a problem, contention free solutions have been developed in the past. However, most of these solutions rely on cache coherency or use powerful atomic primitives, e.g., compare&swap. There are many industry standard buses that provide local shared memory but do not support cache coherency. Likewise, there are many processors that support swap or fetch&add but do not support the more powerful atomic primitives.

New or improved solutions using local shared memory and the more widely supported swap or fetch&add atomic primitives have practical significance in that they have wider applicability. We have developed a number of solutions that meet this criteria.

We observe that any two processors in a system of N processors can synchronize in a contention free manner using swap or fetch&add and a single synchronization variable. Further, more than one variable may be combined to synchronize more than two processors. We exploit this fact to build data structures that we then use to develop new algorithms.
We have developed three mutual exclusion locks: an array based lock using \textit{fetch\&add}, another array based lock using \textit{swap}, and a linked list based lock using \textit{swap} as the only atomic primitive. The solutions are efficient, scalable, and provide guaranteed FIFO behavior. The first two extend the previous array-based locks \cite{6}\cite{7}\cite{32} to architectures without coherent caches. The original solutions relied on cache coherency to eliminate contention. One of these array based solutions \cite{32} has also been extended by Craig \cite{17}. The third solution extends a linked list based lock \cite{73}. It uses local spinning to eliminate contention, and ensures FIFO behavior using \textit{swap} as the only atomic primitive. A solution presented in \cite{73} required \textit{compare\&swap} to guarantee the FIFO behavior.

To our knowledge no contention free solution has been attempted to the readers-writers problem using \textit{swap} or \textit{fetch\&add} as the only atomic primitives. We have developed two array based solutions using \textit{fetch\&add}. We have also developed a singly linked list and three doubly linked list based solutions using \textit{swap} as the only atomic primitive. All of our solutions guarantee FIFO behavior. One of our array based solutions relies on a coherent cache. The rest of the solutions use local shared memory to eliminate contention and do not rely on a coherent cache.

We have also developed lock free solutions to the producer-consumer problem. Using \textit{swap} as the only atomic primitive we have developed a linked list based solution for multiple producers, single consumer. Previous solutions \cite{70} have required \textit{compare\&swap} to provide the same capability that we provide using \textit{swap}. We have also developed array based producer-consumer solutions. These solutions offer a bounded buffer and are characterized by the fact that no producer or consumer is required to wait.

We have measured the performance of our algorithms on a shared bus shared memory multiprocessor system that does not support a coherent cache. The results verify that our algorithms do eliminate contention due to synchronization in architectures without coherent caches.

We review the classification of computer architectures and two shared memory models in Section 1.1. The attributes of shared information that independent entities accessing and updating the information must preserve are discussed in Section 1.2. Well known
synchronization problems are reviewed in Section 1.3 and synchronization mechanisms are discussed in Section 1.4. The problem addressed in this research is stated in Section 1.5 and the organization of the rest of the dissertation is described in Section 1.6.

1.1 Shared Memory Multiprocessors

In this section we review a classification of computer architectures. Next, we review two shared memory multiprocessor architectures in the multiple instruction multiple data (MIMD) category of the classification. These serve as the architectural models in this study.

1.1.1 A Classification of Computer Architectures

Computers may be classified in a number of different ways [24][25][33]. The classification scheme proposed by Flynn is used more widely [25]. According to this scheme an architecture is characterized by the multiplicity of hardware provided to service the instruction and the data streams. Since there could be one or more units to service the instructions and one or more units to service the data, all architectures may be classified in one of the following four categories:

- Single Instruction stream and Single Data stream (SISD),
- Single Instruction stream and Multiple Data streams (SIMD),
- Multiple Instruction streams and Single Data stream (MISD), or
- Multiple Instruction streams and Multiple Data streams (MIMD).

The first category in this classification is essentially the single processor architecture as we know it today. The remaining three form the basis of parallel computers. MISD is not very popular and no machine of this type is known to have been realized [45]. Arrays of processors are an example of SIMD while most multiprocessor systems and multiple computer systems fall under the MIMD category.

The MIMD category may further be classified in more than one way according to some criteria. MIMD systems are categorized as loosely coupled, moderately coupled, or tightly coupled according to the level of interaction between different processors [14][45]. In
loosely coupled architectures, processors generally communicate over a communication facility or network using messages. In tightly coupled architectures, processors communicate through a shared memory using shared variables. Architectures using shared memory are called shared memory multiprocessors. Shared memory offers a simpler programming model. Therefore, it is attractive from programming point of view. Improvements in processor performance and advances in communication networks have reached a level where the shared memory model is being extended as a programming model to locally networked processors [5]. Using messages to communicate among the networked processors, the lower level software makes the memory associated with each networked processor appear to the user as the virtual shared memory. The virtual shared memory and the associated processors, therefore, appear to the user as a shared memory multiprocessor which the user can program using the shared memory programming model.

1.1.2 Shared Memory Multiprocessor Architectures

Shared memory multiprocessors either use a shared bus or a multistage interconnection network to communicate between processors and memory. In a bus based architecture, a single bus is shared by all processors to communicate with the global shared memory as shown in Figure 1. In addition to the global shared memory, each processor may have some local shared memory that it can access as its local memory while the rest of the processors can access it as the shared memory over the shared bus. Also, each processor may have cache memory which is not explicitly shown in Figure 1. The bus and the global shared memory in this architecture are shared resources. The bus is primarily used to communicate between the processors and the memory. Since the bus is a shared resource, it limits the maximum number of processors this type of architecture can support. Approximately twenty processors represent a reasonable upper bound [3].

An architecture may use local memory as well as cache to reduce the number of accesses over the shared bus. The local memory may be private or shared. The local private memory is accessible to the processor with which it is associated but not accessible to the rest of the processors. The processor with which it is associated can access it locally without using the shared bus. It may be used for text or private data to reduce accesses over the shared bus. The local shared memory, on the other hand, is accessible to all processors.
The processor with which it is associated can access it locally without using the shared bus while the rest of the processors can access it over the shared bus. It may be successfully exploited to reduce the number of accesses over the shared bus if the processor with which it is associated accesses the data more frequently than the other processors. The local shared memory can be viewed as an extension to the global shared memory. It may also be viewed as distributed shared memory.

![Figure 1. A Bus Based Shared Memory Multiprocessor System](image)

A cache is normally used as a fast memory to increase the performance of the processor. It keeps a copy of the text or data from the main memory. The subsequent accesses to the same text or data may be satisfied from the cached copy. A cache may be used to cache the data shared among a number of processors or cache the text or data private to a processor. When the shared data is cached, other processors that share the data may also cache it in their cache memories. The multiple copies of the same data in different caches must always be kept identical, i.e., coherent. This requires the cache used for shared data to be coherent as opposed to the cache used for text or data private to a processor. When a cache is used for text or data private to a processor, there is only one cached copy and the issue of coherency does not arise. The bus based architectures generally use snoopy cache coherency protocols which use the bus as the broadcast medium [3]. These can be
grouped in to invalidation based and distributed write or update based protocols [10]. Controllers for coherent caches are comparatively complex and, as a result, expensive. As an alternative, the shared data either may not be cached or cached by only one processor at a time. The later requires some assistance from the software.

The other shared memory architecture uses a multistage interconnection network to connect $N$ processors to $M$ memories as shown in Figure 2. The memory in this model is distributed. The interconnection network allows each processor to access any memory module in the system. However, at any given time only one processor can access a given module. The combining networks [29][85][92] allow multiple processors to access the same memory module provided they are accessing the same location in the module. However, if they access different locations within a module, the accesses must be serviced one after the other.

![Figure 2. An Interconnection Network Based Shared Memory Architecture](image)

In addition to the distributed global shared memory, each processor may have local memory as well as cache. The local memory may be private or it may be shared [15]. The cache
may or may not support coherency. If it does not support coherency, the shared data can
not be cached. If it does support coherency, it uses directory based coherency protocol [3].
The snoopy cache protocols generally used by bus based architectures use the bus as a
broadcast medium and are not suitable for the interconnection network based architec-
tures. The bus based architectures, however, may use the directory based protocols at the
expense of some loss in performance [3].

Using this model, architectures with hundreds or even thousands of processors have
been demonstrated [29][85][92]. The software in these architectures, however, need to
partition the shared data to distribute memory accesses over memory modules as evenly
as possible [93]. If memory accesses are not evenly distributed, the communication band-
width of the interconnection network will degrade and so will the performance of the ma-
chine. Any focused traffic could create hot spots [63][86]. In a buffered multistage
interconnection network, the hot spots not only affect the access to hot memories but also
to the rest of the memories [63][86][108]. In a non-blocking network, i.e., the one that does
not buffer, the hot spots do affect the access to the hot memories even though they do not
significantly affect the access to the rest of the memories [105].

1.2 Attributes of Shared Information

There are two attributes of shared information that need to be preserved during an up-
date. We refer to these attributes as atomicity of data and atomicity of update. Atomicity of
data refers to preserving a unit of shared information as an atomic unit during an update.
Atomicity of update refers to preserving the collective effect of all the updates on a unit of
shared information during concurrent updates.

Both of these attributes need to be ensured in any system that uses shared information.
We discuss these in the context of shared memory multiprocessors where shared vari-
ables represent shared information and multiple processors or processes represent inde-
pendent entities that may concurrently access a shared variable.

1.2.1 Atomicity of Data

By its very nature digital information is discrete and is represented in discrete units. The
smallest unit of information is a bit, an octet of which forms a byte. Four bytes constitute
a word. Higher level data structures, i.e., units, are composed of multiple ordered words and may themselves be further combined in a hierarchical fashion to compose yet other higher level structures, i.e., larger units. At any level a unit as a whole is meaningful and conveys some information. To preserve the information content, the unit must be preserved as a whole in its entirety as an atomic unit during an update. We refer to this attribute of shared information as atomicity of data.

We may use a shared variable $v$ as an example. Let there be one writer that updates $v$ and multiple readers that read $v$. The information represented by $v$ is consistent before the writer starts the update or after it completes the update. During the update, the information is likely to be inconsistent since the writer may have updated only a part of $v$. Since the readers do not change the information, any number of readers may concurrently read $v$. For information to be consistent, a reader must read $v$ either before any part of it is updated or after all parts of it are updated.

The atomicity of data for a partial or full word is preserved by the hardware. Therefore, the software may allow any number of readers to read a partial or full word concurrently with a single writer updating it. A reader that reads prior to the update reads the old value. A reader that reads after the update reads the atomically updated new value.

Maintaining the atomicity of data consisting of multiple words is the responsibility of the software. The software must ensure that a reader reads an entire multiword variable either before any part of the variable is updated or after all parts of the variable are updated.

1.2.2 Atomicity of Update

If we allow multiple writers to concurrently update a shared variable, we need to preserve not only the atomicity of data, but also the collective effect of all the updates. We refer to preserving the collective effect of all the updates of a shared variable as the atomicity of update.

If we let two writers $P_i$ and $P_j$ concurrently update a variable $v$ and the update by $P_j$ follows the update by $P_i$, it would appear as if $P_i$ never updated $v$ unless its update is observed

---

1. Throughout this dissertation we will take a word to be equal to the native size of a machine. Most of the present day machines being 32 bits, we will consider the word to be a four byte quantity.
by some element in the system. We may assume \( v \) is a single word shared variable with an initial value of \( k \). Let \( P_i \) and \( P_j \) each add 1 to \( v \). Assuming \( P_i \) precedes \( P_j \), it reads the value \( k \) for variable \( v \) from the memory into a register, adds 1, and writes the result, \( k+1 \), back to \( v \). \( P_j \) which follows \( P_i \), may read the variable \( v \) either before or after \( P_i \) writes \( k+1 \) back. If \( P_j \) reads \( v \) before \( P_i \) writes \( k+1 \) back, it also reads the value \( k \) for \( v \) into a register, adds 1, and writes \( k+1 \) back. Assuming \( P_j \) writes \( k+1 \) after \( P_i \) does, the update by \( P_i \) is lost. The value of variable \( v \), which should have been \( k+2 \), is instead \( k+1 \). Even though the atomicity of data is preserved, the collective effect of multiple updates is lost. To preserve the collective effect of multiple updates a writer must update, i.e., read, modify, and write, a variable without an intervening write by another writer.

Atomicity of update for a partial or full\(^2\) word may be ensured either by the hardware or by the software. An update generally involves reading a shared variable, modifying the value to a new value, and writing back the new value. Hardware allows a limited set of modify functions. Any number of readers and writers may concurrently access a single word shared variable for reading and updating, as long as the update is limited to the hardware supported modify functions. Ensuring the atomicity of update implicitly ensures the atomicity of data as well.

Ensuring the atomicity of update of a partial or full word for modify functions not supported by the hardware is the responsibility of the software. Likewise, ensuring the atomicity of update of a data structure consisting of multiple words is also the responsibility of the software. The software must ensure that a reader reads an entire multiword variable either before any part of the variable is updated, or after all parts of the variable are updated. Additionally, the software must ensure that a writer atomically updates a variable without an intervening write by another writer between the time the writer reads the variable and writes the modified value back.

1.3 Synchronization Problems in Shared Memory Multiprocessors

Two processes are concurrent if their executions overlap in time. Concurrent processes may execute on the same processor or on different processors in a multiprocessor system.

---

2. In some instances the hardware may ensure the atomic update of two full words, e.g., Motorola 680x0 or Intel Pentium processors provide instructions for atomically swapping two words.
Unless otherwise noted, we assume they execute on different processors in the context of this discussion as well as in the rest of the dissertation.

Concurrent processes may operate on disjoint sets of data and may not share resources. Such processes are disjoint or non-interacting and are not of interest in this study. Alternatively, the concurrent processes may operate on shared sets of data and may also share resources. Such processes are cooperating processes. They need to coordinate their activities to ensure the atomicity of data and the atomicity of update, while accessing shared data or other shared objects. Some well known problems requiring such coordination are listed below.

1. Each process may need to update the same shared data structure. To do this, each process requires exclusive access to the data it updates. This is the well-known mutual exclusion problem [20].

2. Some processes may update while others may only read a shared data structure. Only one process may update the data at a time. However, multiple processes may concurrently read the data structure. This is the essence of the classic readers-writers problem [16].

3. A task may be divided such that a process works on or produces an item and then forwards it to the next process, which uses or consumes the item. This is known as the producer-consumer problem.

4. A task may be split into phases and each process may execute a part of the task during each phase. Each process needs to know when it can start a phase. Next, it needs to inform others when it has finished its part and wait for the rest of the processes to finish before it can start the next phase. This kind of coordination is known as barrier synchronization.

To coordinate their activities, processes need to communicate with each other. In a shared memory system processes communicate using synchronization variables. Since the synchronization variables themselves are shared variables, the atomicity of data and atomicity of update for the synchronization variables needs to be ensured. The means used to ensure the atomicity of data and the atomicity of update for synchronization variables as
well as the protocols used to coordinate process activities using the synchronization variables are broadly categorized as *synchronization*.

The problems listed above represent well known synchronization problems. A solution to any of the problems must satisfy *safety* and *liveness* properties in order to be a correct solution. The two are considered basic properties. A solution may satisfy additional properties, *e.g.*, *fairness*. Safety properties specify what a solution *may or, alternatively, may not* do while the liveness properties specify what a solution *must* do [12][54][56]. As an example, in a solution to the mutual exclusion problem, the mutual exclusion is the safety property and deadlock freedom is the liveness property. In what follows we discuss each problem and describe its safety, liveness, and fairness properties.

### 1.3.1 The Mutual Exclusion Problem

Dijkstra calls mutual exclusion "a key problem" [20]. It is used to provide exclusive access to shared objects. Further, it is used to solve other synchronization problems. It is the latter use that qualifies it as "a key problem". Once a process has exclusive access to a shared variable, it can manipulate the variable without interference from any other process. This enables mutual exclusion to solve the rest of the synchronization problems.

Two entities, \( P_i \) and \( P_j \), must mutually exclude each other if an operation, \( O_i \) by \( P_i \), may not overlap with an operation, \( O_j \) by \( P_j \). This is the essence of the mutual exclusion problem. For each entity the problem may be abstracted as follows.

```plaintext
initial declarations
repeat forever
    non-critical section // does not involve shared objects
    entrance
    critical section // involves shared objects
    exit
```

*Figure 3. An Abstraction of the Mutual Exclusion Problem*
A process may execute its non-critical section followed by the entrance code to acquire exclusive access to a shared object. It can then execute its critical section at the end of which it executes the exit code to give up the exclusive access to the shared object. A process may execute this sequence any number of times.

A solution consists of the specification of the initial declarations, the entrance code, and the exit code. A solution must satisfy the safety and the liveness properties for it to be a correct solution [12][58]. These are the basic properties and were satisfied in the solution originally developed by Dekker and generalized by Dijkstra [19][20]. The safety properties for mutual exclusion can be stated as follows.

» Mutual Exclusion Property: A process $P_1$ and a process $P_j$ may not execute their critical sections concurrently.

While mutual exclusion specifies that only one process may enter its critical section, it does not require that any process ever does so. If one or more processes try to enter their critical sections but no process ever does, mutual exclusion is not violated. However, the processes are in a deadlock. This situation is addressed by the liveness property which specifies what must occur.

» Deadlock Freedom Property: When no process is executing in its critical section then some process trying to enter its critical section must be able to do so in a finite time.

We must not have a situation in which there is no process executing in its critical section and all processes executing in their entrance sections continue to wait forever. Such a situation represents a deadlock condition in which all cooperating processes will eventually cease performing any useful work.

The deadlock freedom guarantees that the system as a whole makes progress. However, it promises nothing about an individual process which may wait forever in its entrance section while trying to enter its critical section. This is called starvation, which is a permissible condition in most systems. It is addressed by the fairness property.

» Starvation Freedom Property: Every process trying to enter its critical section must be able to do so in a finite time.
A solution that satisfies this property guarantees against lockout or starvation of an individual process; this property was first satisfied by Knuth in his solution [49]. Since a starvation free solution ensures no process is locked out, starvation freedom may also be viewed as the liveness property for an individual process.

Starvation freedom ensures that each process trying to enter its critical section eventually does so. However, it says nothing about the number of times other processes may enter and exit their critical sections before a given process gets its turn. This is addressed by the stronger fairness property, first come, first served or first in, first out (FIFO).

First In, First Out Property: If a process $P_i$ tries to enter its critical section prior to when another process $P_j$ tries to enter its critical section, then $P_i$ must enter its critical section before $P_j$ enters its critical section.

A process tries to enter its critical section by executing its entrance code. We associate an imaginary barrier with a statement prior to the statements used for waiting in the entrance code. A process that crosses the barrier by executing the statement with which it is associated is considered trying to enter its critical section. A formal definition is given by Lamport [58]. The FIFO property was first satisfied by Lamport in his solution [53].

The first come, first serve or first in, first out (FIFO) property does not imply deadlock freedom. First in, first out and deadlock freedom properties together, however, do imply starvation freedom [58]. Lamport describes some additional properties such as premature termination or failure of a process while executing the critical section [58]. These properties are not relevant to the work pursued in this study and therefore not stated here.

1.3.2 The Readers-Writers Problem

The general or classic readers-writers problem introduced by Courtois, Heyman, and Parnas may have multiple readers and multiple writers as opposed to the concurrent reading while writing problem which may have multiple readers but one writer [16][54][83]. We are interested in the general readers-writers problem in this study. It requires that writers have exclusive access to a shared object while readers may have concurrent access. This may also be viewed as a generalization of the mutual exclusion problem. The abstraction presented in the mutual exclusion problem is equally applicable to the readers-writers
problem. The entrance and the exit code, however, differs to allow readers concurrent access to a shared object.

A solution consists of the specification of the initial declarations, the entrance code, and the exit code. The safety properties for the readers-writers may be stated as follows.

- **Readers-Writers Property:** A process $P_i$ and a process $P_j$ *may* execute their critical sections concurrently if $P_i$ and $P_j$ are both readers but *may not* execute their critical sections concurrently if either $P_i$ or $P_j$ is a writer.

The liveness properties for the readers-writers problem may be stated as follows.

- **Deadlock Freedom Property:** When no process is executing in its critical section then some process trying to enter its critical section *must* be able to do so in a finite time.

- **Concurrent Reading Property:** If a reader process $P_i$ is executing in its critical section and no writer is *trying to enter* its critical section, then another reader process $P_j$ *trying to enter* its critical section *must* be able to do so without waiting for $P_i$ to exit its critical section.

As explained in the previous section, a process *trying to enter* its critical section implies the execution of a statement associated with an imaginary barrier in its entrance code.

Since the safety property allows, but does not require, readers to read concurrently, a solution to the mutual exclusion problem also satisfies the safety property for the readers-writers problem. Further, a solution to the mutual exclusion problem also satisfies the deadlock freedom property. Therefore, a solution that does not satisfy the concurrent reading property does not qualify as a solution to the readers-writers problem. Instead, it is a solution to the mutual exclusion problem.

The fairness properties as stated in the mutual exclusion problem are applicable to the readers-writers problem as well and are not repeated here.

### 1.3.3 The Producer-Consumer Problem

This problem is first discussed by Dijkstra and later further discussed by P. B. Hansen and others [20][35][89]. The mutual exclusion and the producer-consumer are considered
basic communication problems in shared memory multiprocessor systems or in multipro­
cessing systems in general. A producer communicates with a consumer through a buffer. 
The producer produces an item and queues it at the end of the buffer. The consumer re­
moves an item from the front of the buffer and consumes it. On the average, the consumer 
removes as many items from the front of the buffer as the producer queues at the end of 
the buffer. There may be a single or multiple producers. Likewise, there may be a single 
or multiple consumers. The aggregate rate at which the producers can queue can not be 
greater than the aggregate rate at which the consumer can remove items from the buffer. 
The instantaneous rate, however, may differ. The buffer acts as an elastic store to absorb 
the difference in the instantaneous rates.

When the buffer is full, the producer can not queue any more items, i.e., it must wait for 
space to become available. Likewise, when the buffer is empty, a consumer can not re­
move an item, i.e., it must wait for an item to become available. An abstract view of the 
producer-consumer problem is presented below.

<table>
<thead>
<tr>
<th>Producer</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial declarations</td>
<td>initial declarations</td>
</tr>
<tr>
<td>repeat forever</td>
<td>repeat forever</td>
</tr>
<tr>
<td>produce an item</td>
<td>remove an item</td>
</tr>
<tr>
<td>acquire space</td>
<td>free the space</td>
</tr>
<tr>
<td>queue the item</td>
<td>consume the item</td>
</tr>
</tbody>
</table>

Figure 4. An Abstract View of the Producer-Consumer Problem

A solution to a producer consists of the initial declarations to specify the initial conditions, 
acquiring space and queuing the item. Likewise, a solution to the consumer consists of 
initial declarations to specify the initial conditions, removing an item and freeing the 
space.

The safety properties for a solution to the producer-consumer problem are listed below.

- Messages may be removed only in the order in which they are queued.
A producer **may not** queue a message if there is no space available in the buffer.

A consumer **may not** remove a message if there is no message queued in the buffer.

The liveness properties for the producer-consumer problem may be stated as follows.

- When space is available, some producer trying to queue an item (message) **must** be able to do so in a finite time.
- When messages are available, some consumer trying to remove an item (message) **must** be able to do so in a finite time.

When a solution is intended for multiple producers and multiple consumers the fairness properties may be relevant. The fairness properties are similar to the fairness properties for the mutual exclusion problem, but applicable only within the set of multiple producers or within the set of multiple consumers.

### 1.3.4 The Barrier Synchronization Problem

This problem is specific to multiprocessors whereas the rest of the problems arise in multiprocessor environments irrespective of the multiplicity of the hardware, i.e., multiprocessor on single processors or multiprocessors.

Consider a multiprocessor with two or more processors. An application may be divided into multiple phases and each processor may be assigned a part of the work in each phase. Before the next phase may start, the previous phase must completely finish. This requires each processor to inform the others that it has finished its part of a given phase. When all processors have finished, the next phase may start.

Barrier synchronization may be partitioned into two steps. In the first step, each processor informs the others that it has finished its part of the work for a given phase. It then waits for the rest of the processors to finish their parts, unless it itself is the last processor; this constitutes the second step.

Mellor-Crummey and Scott present an overview of barrier algorithms [74]. We do not focus on the barrier synchronization in this study. Therefore, we do not discuss it further.
1.4 Synchronization Mechanisms.

Solutions to the synchronization problems entail updating the synchronization variables. As an example, a process in its entrance code in the mutual exclusion problem updates a synchronization variable to indicate it has entered its critical section. The update of a synchronization variable itself must ensure the atomicity of update.

Early solutions to synchronization problems were based on mutual exclusion which itself was solved using simple reads and writes [18][19][20][23][49][53][84]. The atomicity of data for simple reads and writes is guaranteed by the hardware. However, the atomicity of update had to be ensured by the software. These solutions were quite complex and not very practical.

As Dijkstra observed, the solutions were complex because simple reads and writes are incapable of providing uninterrupted observation and modification of a variable [20]. In other words, the atomicity of update is not ensured by the hardware. If the hardware could ensure uninterrupted observation and modification, i.e., atomicity of update, of a variable, it could greatly simplify the solutions. In what follows we discuss the hardware and software mechanisms to update synchronization variables atomically and provide solutions to the synchronization problems.

1.4.1 Hardware Atomic Updates

In a single processor system, the currently running process may acquire mutual exclusion by momentarily disabling the interrupts. Once it acquires mutual exclusion, it can atomically update a shared variable without interference from another process. This simple but effective mechanism can be used to solve synchronization problems in single processor systems. However, it does not work in shared memory multiprocessors.

In a shared memory multiprocessor system, a process may use atomic read-modify-write capability to atomically update a synchronization variable. An atomic read-modify-write implemented in the hardware ensures the atomicity of update for a partial or full word, or in some cases for two full words. This, in turn, can be used by the software to build solutions to the synchronization problems. There are two mechanisms used by the hardware to implement atomic read-modify-write capability.
The first mechanism uses mutual exclusion at the hardware level. Using a hardware sig-
nal known as lock, it allows one processor to read, modify, and write back a variable in
one uninterruptible sequence. The lock is asserted for the duration of one read-modify-
write instruction. It is used by the hardware to exclude other processors from accessing
the shared memory. The processor executing the atomic instruction has exclusive access
to the variable during the execution of the instruction. Early implementations allowed
atomic update of a bit. Later, came the more powerful mechanisms such as swap; this is
also known as exchange or fetch&store. The swap allows atomic update of a word in the
memory. Next, the processors added fetch&add and compare&swap to their repertoire of
atomic instructions. Atomic fetch&add reads a variable from the memory, adds a value to
it, and writes it back. Atomic compare&swap reads a variable from the memory, compares
it with a compare value in a register, and, only if the comparison succeeds, writes a new
value from another register to the memory. Some processors, e.g., Motorola 680x0 and
Intel Pentium, support an atomic compare&swap of two words.

The second mechanism implemented by later processors, e.g., MIPS and PowerPC, does
not use explicit lock signal. In this mechanism, when a processor reads, i.e., loads, a vari-
able for an atomic update, it internally saves the address of the variable; such a load is
called load-linked. After modifying the variable the processor attempts to write, i.e., store,
the variable back; such a store is referred to as store-conditional. The write succeeds only
if no other processor has written to that memory address since this processor read the
variable. Conceptually it is similar to compare&swap. However, it allows the use of an ar-
bitrary number of instructions to modify the value after a load-linked and before a store-
conditional. That makes it a very flexible mechanism. Also, this mechanism does not per-
mit a spurious successful write to a word that has been changed and then changed back
to its old value, a major weakness of compare&swap.

1.4.2 Software Solutions to Synchronization Problems

There are two approaches to solving synchronization problems. The first approach is
based on the concept of critical sections or mutual exclusion [20]. In this approach only
one process is allowed to enter its critical section or access a shared object at a time. Oth-
er processes must wait. A process may wait by spinning on a variable, the value of which
indicates whether the object is free. This form of waiting is called *busy waiting*. As an alternative, a process may wait by blocking itself in a queue associated with the shared object and giving up control. This form of waiting is known as *blocking*. When the object becomes available, the process is unblocked.

Busy waiting is used when the expected waiting time is short. It may also be used when blocking may not be suitable, e.g., inside an operating system kernel. Solutions in which processes busy-wait are also referred to as *spin locks*. A busy waiting process polls one or more synchronization variables to check whether it can acquire the lock. Solutions requiring no atomic read and write, based on atomic read and write, and based on atomic read-modify-write have been implemented [2][8][18][19][20][23][32][49][53][57][73][84]. Solutions that do not use hardware provided read-modify-write are complex and not very practical. Solutions based on read-modify-write are quite simple. However, polling of synchronization variables tend to produce large amounts of memory and interconnection network contention which causes performance degradation [2][7][8][32][63][73][86][108].

Recognizing contention as a serious performance problem, a number of algorithms have been invented that do not produce memory or interconnection network contention due to busy waiting [8][17][32][50][73].

If busy waiting processes reside on the same processor, the waiting processes are simply taking cycles away from the process that has the lock and delaying it in finishing its work. Also, in priority based preemptive systems, a deadlock may occur if a process with a priority higher than the one holding the lock starts busy waiting. Therefore, busy waiting is not suitable for single processor systems.

Blocking is preferred when the time to wait may be comparable or more than the time required for context switching. When a process blocks itself, it queues itself in a queue associated with the locked object. When the object becomes available, the process is signaled to resume. The implementation of blocking on single processors is simple. A process momentarily disables the interrupts to gain mutual exclusion, queues itself, or signals a waiting process. Also, since all processes are on the same processor, it does not require additional hardware to signal a process. However in a multiprocessor system, it requires hardware support to signal a process that may reside on a different processor.
A process that waits by blocking itself do not cause memory or network contention. However, it requires a context switch to give up control when it fails to get a lock and another context switch when it resumes.

Figure 5. A Hierarchy of Synchronization Mechanisms

The second approach to solving synchronization problems is based on concurrent objects. A concurrent object is a shared data structure that can be concurrently accessed by multiple processes without requiring exclusive access [36][38][39]. A concurrent object implementation uses atomic primitives provided by the hardware and may be lock free or wait free at the software level. An implementation is lock free (also sometimes known as non-blocking) if some process must complete an operation in a finite number of steps [39][40]. It implies the system as a whole makes progress but it does not guarantee starvation freedom for an individual process. An implementation is wait free if each process must complete an operation in a finite number of steps [39][40]. This implies starvation freedom.

A hierarchy of different synchronization mechanisms is presented in Figure 5. The higher level mechanisms are built using the lower level mechanisms.
1.5 The Problem Statement

We have discussed shared memory multiprocessors, reviewed the attributes of shared information that need to be preserved, and described well-known synchronization problems and their safety and liveness properties. Now we state the problem we have addressed in this study.

- Given the existing hardware mechanisms that comprise shared memory, processors that communicate using shared variables, and given the existing atomic read-modify-write primitives, what new or improved solutions can be developed for different synchronization problems? In particular we are interested in contention free synchronization for the mutual exclusion and readers-writers problems. Further, we are interested in wait free synchronization for the producer-consumer problem. We require the following of our solutions.

1. Use swap (or in some cases fetch&add) as the read-modify-write primitive.
2. Use local shared memory but do not rely on cache coherency.
3. Use local only spinning to eliminate contention due to busy waiting.

The results would help existing architectures in improving their performance. Any architecture in which processors, at a minimum, support the swap read-modify-write primitive can benefit from these solutions. Since, our solutions do not depend on cache coherency they equally benefit architectures with or without coherent caches.

1.6 Organization of the Rest of the Dissertation

In this chapter we have reviewed the shared memory architectures, discussed the attributes of shared information that needs to be preserved during updates, introduced and reviewed the well known synchronization problems, and provided an overview of the synchronization mechanisms. We then stated the problem, followed by this section which describes the organization of the rest of the dissertation.

We make a simple and somewhat obvious observation regarding swap in CHAPTER 2. We show that swap is an adequate primitive to synchronize any two processors out of N processors. The result holds true for fetch&add as well. Our contribution is in observing this fact, that it is possible to use more than one synchronization variable to build interesting
and useful mechanisms previously considered not possible using swap or fetch&add as the only atomic primitives. Next, using such data structures, we develop a number of new algorithms in CHAPTER 2 and the subsequent chapters.

We offer new solutions for contention free exclusive locks in CHAPTER 2. The first solution uses fetch&add and provides a simple extension to Anderson's lock [8]. The next two solutions are based on swap as the only read-modify-write primitive. These solutions guarantee FIFO behavior and eliminate contention in architectures with or without coherent caches using local shared memory.

We extend our solutions to readers-writers locks in CHAPTER 3. These are new contention free solutions. Each solution guarantees the FIFO behavior. The first two solutions extend the fetch&add based exclusive lock to readers-writers locks. The first is a simple algorithm that relies on a coherent cache to eliminate contention while the second eliminates contention by busy waiting in its local shared memory. To our knowledge, no solution similar to these has previously been attempted. Next, we offer a number of solutions using swap as the only read-modify-write primitive. Each of our solutions guarantees the FIFO behavior and eliminates contention by busy waiting in local shared memory. Contention free solutions to the readers-writers problem using swap as the only atomic primitive have previously been considered infeasible [50].

CHAPTER 4 presents lock-free concurrent queue solutions to the producer-consumer problem. A solution to the producer-consumer may be implemented as a bounded buffer, i.e., as an array, or as a linked list. We provide solutions using both approaches. The array based solution uses fetch&add primitive to implement a solution for multiple producers, single consumer. The solution is extended to multiple producers, multiple consumers. However, the extension requires compare&swap in addition to fetch&add. The linked list based implementation uses swap as the only read-modify-write primitive to implement a solution for multiple producers and a single consumer. The solution may be extended to multiple producers, multiple consumers. However, the extension either requires a two word compare&swap or load-linked, store-conditional capability. Further, even though such a solution is lock free, it is neither wait free nor contention free. Therefore, we
suggest a solution in which multiple consumers use a contention free lock based on *swap* as the only atomic primitive. The producers in this solution are still lock free and wait free.

We have measured the performance of our contention free mutual exclusion and readers-writers algorithms. The results of the performance measurements are presented in CHAPTER 5. The results verify that the algorithms are efficient and scalable. The performance does not degrade as we add more processors to the system. The lock overhead stays fairly constant irrespective of the number of processors in the system.

We summarize the results in CHAPTER 6. Based on the performance results, we also give recommendations on the use of various algorithms in different architectures. Lastly, we suggest future work in the area of synchronization in shared memory multiprocessors.
CHAPTER 2

CONTENTION FREE MUTUAL EXCLUSION LOCKS

E.W. Dijkstra [19] introduced the mutual exclusion problem in 1965 and solved it using simple reads and writes for a shared memory multiprocessor model. A number of researchers have studied the problem since and offered improvements to the existing solutions or provided new solutions.

Based on a simple observation regarding swap and fetch&add atomic primitives, we show how two processes can synchronize in a contention free manner. We further show how a linked list can maintain the FIFO behavior using swap as the only primitive. We then use these capabilities to build new contention-free fair mutual exclusion and readers-writers locks and lock-free producer consumer. In this chapter, we extend two of the existing exclusive locks to architectures that may not support coherent caches. The original solutions eliminate contention on architectures with coherent caches but not on architectures without coherent caches [7][8][32]. Our third solution also eliminates contention on architectures with or without coherent caches. The previous solution also eliminates contention on architectures with or without coherent caches but does not guarantee the FIFO behavior using swap [71][72][73]. Our solution guarantees the FIFO behavior using swap as the only atomic primitive. It provides the same characteristics using swap as the previous solution provides using compare&swap.

2.1 Introduction

Mutual exclusion is one of the fundamental synchronization problems. Only one process may enter its critical section or access a shared object at a time. The rest of the processes must wait. A waiting process may busy wait or block itself.
In a single processor multiprocessing environment, a process must block itself while waiting. Busy waiting amounts to wasting machine cycles. Unless a waiting process gives up control, the process that owns the lock cannot execute, and unless the process that owns the lock executes, it cannot release the lock. Therefore, blocking is the only reasonable mechanism in single processor systems. Also, a process may acquire mutual exclusion by simple means, e.g., by disabling interrupts, enforcing run to completion scheduling, etc., in a single processor environment.

In a multiprocessor environment, a process may busy wait or block itself while waiting. Busy waiting is preferable when the critical section is small. Also, there may be situations, e.g., while executing the operating system code, in which blocking may be inappropriate. In general, when the expected waiting time is less than the context switch time, busy waiting is preferable [47]. However, when the expected waiting time is more than the context switch time, blocking is preferred. A process in a multiprocessor environment must use a synchronization variable to acquire mutual exclusion. It cannot use simple means such as disabling interrupts to enforce mutual exclusion. These means have no effect on processes running on other processors.

In this chapter we deal with busy waiting. Traditional implementations of busy-waiting, may result in excessive transactions over the bus or the interconnection network due to polling causing performance degradation [2][7][8][32][63][73][86][108]. When many waiting processors poll the same synchronization variable, they cause what Pfister and Notron termed as hot spots in multistage interconnection networks [86]. The presence of hot spots may degrade the performance of the interconnection network for all traffic and not only the traffic due to busy waiting [63][86][108]. Likewise, in bus based shared memory architectures, the disproportionate amount of traffic due to busy waiting will impede the progress of the processor holding the lock by repeatedly contending for the bus. Degraded performance is detrimental in high performance systems and an obstacle to scalability in large shared memory multiprocessor systems.

Contention free locks have been developed in the past [8][32][73]. We provide a detailed overview of these locks in Section 2.2.5 while the rest of the Section 2.2 provides an overview of the rest of the mutual exclusion locks. We present a simple observation regarding
the swap and fetch&add primitives as to how they can be used to synchronize any 2 processors out a system of N processors in Section 2.3. We present the contention free locks that we have developed in Section 2.4 and conclude the chapter with a summary in Section 2.5. We have measured the performance of these algorithms. The performance data is presented and discussed in CHAPTER 5.

2.2 Related Work

We review the related work in this section. We have grouped the previous implementations into five groups. The locks in each group are discussed. The order of presentation is such that each subsequent group offers some advantage over the locks in a previous group.

There is another whole class of mutual exclusion algorithms that we do not discuss here [67][68][90][94][99][100][102]. These algorithms are designed for message based distributed parallel computers. They are generally categorized as consensus based or token based [68]. The working of these algorithms differ from the algorithms for the shared memory model to the extent that they warrant a separate discussion. Our work is focused on shared memory algorithms. Therefore, we do not discuss algorithms designed for message based distributed parallel computers. Likewise, we do not discuss the solutions in which the waiting processes block themselves, e.g., semaphores, monitors, etc., [20][21][22][43][91].

2.2.1 Solutions Using Simple Reads and Writes

Dijkstra reported the first correct solution to the mutual exclusion problem in his seminal paper on co-operating sequential processes [20]. He credits the Dutch mathematician Th. J. Dekker for the first correct solution for two processes that he himself generalized to N processes [19][20]. He presented the solution for two processes as well as the generalized solution for N processes along with the proof of correctness [20]. These solutions are based on simple reads and writes and depend only on the atomicity of data of simple read and write operations. The solutions are quite complicated and inefficient. As Dijkstra noted, the complexity results from the fact that using simple reads and writes, a process may either inspect a variable or update a variable at a time. A process updates a common variable based on the inspected values of some other common variables. The update is successful if the values of the common variables inspected by the process do not change until
after the update. However, other processes may change the values of those common variables between the time a process inspects them and updates another common variable. As a result, the process may have to try an update a number of times before it succeeds.

A number of solutions followed that tried to improve Dijkstra' original solution in one way or the other [18][23][49][53][84]. Notable among these is Lamport's Bakery algorithm [53] which guarantees the FIFO behavior as well as provides fault tolerance; it is further improved by Peterson[84]. Lamport later studied the mutual exclusion problem further and provided solutions that did not rely on atomicity of data for simple read and write operations [57][58]. These solutions have theoretical importance but like earlier solutions are not of much practical value. Later, he invented another solution, which is based on simple reads and writes and relies on the atomicity of these operations. This solution is faster as compared to the previous solutions [59].

Even the improved solutions in this group are not efficient enough to be of practical use. Therefore, we do not provide much detail about these solutions. Yet they have historical as well as theoretical significance and their inclusion, brief as it may be, is imperative.

2.2.2 Bit Test&Set Locks

This is the simplest mutual exclusion spin lock. It is built using hardware provided atomic \textit{bit test\&set} or its equivalent, e.g., \textit{bit test\&clear}, read-modify-write primitives. To acquire the lock, a process atomically sets a bit reading back its prior value. If the prior value is clear, the process has atomically set the bit. Therefore, it can acquire the lock. If the prior value was already set, some other process has the lock. Setting a bit which is already set, has no effect. To release the lock, the process holding the lock simply clears the bit.

A process that finds the lock busy must wait. In the simplest form of the lock, it may continuously execute the \textit{bit test\&set} instruction. This, however, may cause contention due to multiple processes atomically modifying the bit in the global memory. As a result, the performance may degrade. It should be noted that execution of \textit{bit test\&set} atomic read-modify-write instruction will cause cache invalidations on all processors waiting for the lock. In an interconnection network based multiprocessor, the focused traffic to one memory module may result in \textit{hot spots} in the interconnection network [85].
Rather than executing *bit test&set* repeatedly, a process may use a simple read to test the bit. Only when a process finds that the bit is clear does it try to set it using *bit test&set* in an attempt to acquire the lock. This is referred to as *bit test, test&set* [97]. Using simple reads to test the bit eliminates cache invalidations due to atomic *bit test&set* instruction.

When the bit is cleared, however, a number of processes may attempt to lock it, causing a surge of atomic *bit test&set* instructions.

In practice, some kind of delay is introduced between successive attempts by a process. One may introduce a constant delay or a variable delay. By introducing the delay, the total traffic due to busy waiting is reduced. However, depending on the delay, the latency may increase. Exponential back-off is reported to provide the best results for reducing contention [2][8][73]. Mellor-Crummey and Scott have presented the code for this type of lock with exponential back-off [73].

### 2.2.3 Number or Ticket Locks

These locks are based on the atomic *fetch&add* (more precisely *fetch&increment*) primitive. The lock consists of two counters. One of the counters is a *request counter*. It is used to request the lock. It keeps a count of how many times the lock has been requested. The other counter is the *release counter*. This keeps a count of the times the lock has been released. The counters are initialized to the same value, preferably to 0. When the lock is free, the two are equal.

To request the lock, a process atomically adds one to the request counter reading back its pre-addition value. Next, it checks if the release counter is equal to the pre-addition value of the request counter. If it is, the lock is free and the process can acquire the lock. If not, the process waits until the release counter equals the pre-addition value of the request counter. At which point it can acquire the lock. To release the lock, a process simply adds one to the release counter. Since the process holding the lock is the only one that will add one to the release counter, it does not need to use atomic *fetch&add*. Instead, a simple add will suffice.

This lock guarantees the FIFO behavior. Further, it does not generate repeated atomic instructions. After obtaining its request number, a process simply waits for the release
number to match its request number. To check the release number it repeatedly reads the release number using a simple read. In a cache coherent architecture, it will continue to spin on a cached copy of the release counter until it is updated. When the release counter is updated, the cached value will be updated in an update (also called distributed write) based coherent cache. Therefore, this lock does not generate contention in an architecture with update based cache coherency [10][69][104]. However, in an invalidation based coherent cache, the cached copy of the release counter of each waiting process is invalidated. This is followed by a read by each process. Depending on the number of waiting processes this could still create some contention. On systems without coherent caches, repeated reads would cause contention.

Mellor-Crummey and Scott propose a proportional back off [73]. Since the difference between the request number and the current release number indicates the number of waiting processes, the process waits for a time proportional to the difference before reading the release number again. The multiplier (the constant of proportionality) is an estimate of the time a process will hold the lock. It is difficult to estimate the time accurately. An overestimate will result in increased latency since a waiting process may delay in checking the release number while the lock may have been released. On the other hand an underestimate will result in more retries and contribute towards some contention. Mellor-Crummey and Scott suggest the use of minimum time a process may hold the lock as the multiplier so as to minimize the latency [73]. They also provide the code for the ticket lock with proportional back-off [73].

2.2.4 Software Combining and Tournament Locks

Large shared memory multiprocessors use a multistage interconnection network to communicate between the processors and the memory modules [29][51][85][92]. One may use buffered (or blocking) multistage interconnection networks such as Omega or its variants or non blocking networks such as the one used in Monarch [61][92][107]. When many processors repeatedly read the same memory location, they may cause contention, i.e., hot spots [86]. Spin locks, barriers, and other synchronization mechanisms may require a processor to repeatedly read a shared variable and are a primary cause of contention. The contention may be for a path through the interconnect, for a memory module, or for a
memory location [93]. In buffered networks the hot spot traffic may cause tree saturation even when the hot spot traffic is a small fraction of the total memory accesses [63][86]. The tree saturation affects all traffic and not just the hot spot traffic [86].

One may take different approaches [93] to minimize contention for a path through the interconnect or a memory module. However, to address the contention for a memory location hardware combining is proposed as the primary mechanism. Hardware combining is implemented in the interconnection network. It recognizes requests to the same memory location from multiple processors, combines such requests, and forwards it as one request to the memory module. When the memory responds, the network at each stage remembers the multiple requesters and forwards the data to multiple processors. The combining proposed in the NYU Ultracomputer also combines atomic fetch&add operations within the network [29]. This allows truly simultaneous or concurrent execution of these operations. The network in a sense becomes a part of the distributed computing.

Hardware combining, however, may be quite expensive [86]. Also, its effectiveness depends on the extent to which the combining can be done. If the combining is restricted, tree saturation can still occur even when a combining network is used [62]. Yew, Tzeng, and Lawrie present software combining as an economical alternative to hardware combining [108]. Software combining takes advantage of the distributed memory. It represents a single hot spot address with a tree and distributes the multiple hot spot addresses of the tree over different memory modules. We will explain it further shortly. This is a general technique that can be used for different synchronization mechanisms including the spin locks and barriers. Conceptually, it is orthogonal to the rest of the techniques presented in this section. Any technique discussed in this section can be used in conjunction with the software combining. Further, it should be noted that this technique is particularly suited for large scale distributed shared memory multiprocessors where the nodes of the tree can be assigned to different memory modules.

In software combining, a single shared variable such as a spin lock, is replaced with a tree of variables as shown in Figure 6. As pointed out before, the technique can be used for any shared variables. We will explain it with reference to a spin lock.
Each node of the tree is assigned a number of processors. The lock is set to free by clearing each node of the tree including the leaf nodes. To acquire the lock, a processor starts at its designated leaf node and attempts to acquire the lock at that node. It may use any of the locking techniques described in this section. The processor that succeeds in locking the node at level \( n \), attempts to acquire the lock at the next level, i.e., \( n+1 \). In other words, there is a contest at each level. The winners of the contest at one level proceed to the next level and the winners at the next level proceed to next to the next level in a tournament style. The processor that wins the contest at the root acquires the lock. When a processor wins the contest at level \( n+1 \), it releases the lock at level \( n \). To release the lock a processor release the lock at the root.

![A Software Combining Tree](image)

**Figure 6. A Software Combining Tree**

The height and the number of processors that contest at each level may be adjusted to maximize the performance. It should be noted that a processor must acquire lock at each level along the path from a leaf node to the root. This increases the latency.

Graunke and Thakkar refer to the approach described above as *pessimistic variation* as opposed to an *optimistic variation* they describe [32]. In optimistic variation, a processor starts at the root. If it succeeds in locking the root, it acquires the lock. If it does not succeed in locking the root, it descends to the next lower level along its designated path and attempts to lock the node at that level. As long as it continues to fail, it will continue to
descend downward along its designated path. When it succeeds in acquiring the lock at any level, it will start back up the path as it would in the pessimistic variation. When it succeeds in locking the root, it can acquire the lock.

The optimistic variation assumes that the lock is lightly contended. If that assumption holds there is high probability that a processor may acquire the lock with fewer attempts than in the pessimistic variation. On the other hand if the assumption does not hold, the processor will waste extra attempts in trying to lock the nodes from root down to its leaf node. With each failed attempt it will descend and may end up waiting at its leaf node. It should be noted that when a processor fails in locking the root and subsequently the nodes on its path from the root to the leaf node, it does not wait at any level until it reaches the leaf node. On the other hand, if it succeeds at any level, it starts back up as it would if it has acquired the lock at that level in the pessimistic variation. In the pessimistic variation, a processor waits at a given level until it gets the lock at that level.

By assigning the nodes of the tree to different memory modules, the number of processors that busy wait for the same variable can be controlled. In the limiting case, the tree could be a binary tree, in which only two processors contend for a given node. In this case one processor gets the lock and there is only one processor that has to wait at any given node. For a binary tree the number of nodes is \( N_p/2 + N_p/4 + N_p/8 + \ldots + 1 \); where \( N_p \) is the number of processors in the system. The sum of the above nodes is simply \( N_p - 1 \). It is interesting to compare it with the queuing locks which, in general, require \( N_p \) elements.

### 2.2.5 Queuing Locks

In general a queuing lock works as follows. A queue is associated with the lock. A processor requesting the lock queues itself at the end of the queue. Next, it checks if it is at the head of the queue. If it is, it owns the lock. If it is not, it waits in the queue for its turn. When the processor at the head of the queue releases the lock, it removes itself from the head. The lock passes on to the next processor in the queue. Each waiting processor spins on a different variable making it possible to eliminate contention by properly locating the variables in the memory. As an example, the variables on which different processors spin may be located in different cache lines in a cache coherent bus based architecture or in different memory modules in a multistage interconnection network based architecture.
processor would spin on the cached copy in the first case or spin on a variable in its associated memory module in the second case.

A queue may take the form of an array or a linked list. In an array based queue, a processor uses the next available element in the array to queue itself. Since more than one processor may concurrently request a lock and attempt to acquire the next element to queue itself, a processor must atomically read and advance the index to secure a slot for itself. In a linked list based queue, a processor uses a data structure that is, a priori, associated with it. However, it must atomically link its data structure at the tail of the linked list to properly queue itself.

We review three queuing locks in the following. The first one uses an array to implement the queue [7][8]. The second uses an array structure but functionally builds an implicit linked list of the waiting processors [32]. The third uses a singly linked list of data structures associated with waiting processors [72][73]. These data structures reside in their local shared memories. The second lock, which is due to Graunke & Thakkar, has been also extended by Craig to provide similar capabilities as our locks provide [17]. We will briefly review the extension in Section 2.2.5.2 with Graunke & Thakkar’s original lock.

2.2.5.1 Array Based Queuing Lock

This lock is due to Anderson[7][8]. We present a variant of this lock in this subsection. The lock consists of an unsigned integer variable and a circular array both of which are declared in the global shared memory. The unsigned global variable which we call Tail is used to assign the consecutive numbers to processors in the order in which they request the lock. Further, it is used to derive an index into the array. The array is used by the waiting processors to ‘stand in line’ and busy wait for their turn to get the lock. If the lock is free, the indexed element holds a token indicating the lock is free. Otherwise, the processor requesting the lock waits spinning on the indexed element until the element indicates the lock is free, at which point the processor acquires the lock. The variable Tail can be viewed as the tail of the queue of the waiting processors. The processor at the head of the queue always owns the lock.

---

1. We would like to thank John Mellor-Crummey for pointing out this reference in an email correspondence. We learned about it at a comparatively later stage in our work but tried to briefly include it in the review.
A processor acquires the lock, when its corresponding indexed element in the array holds a token `lock`. Assuming $N_p$ as the total number of processors in the system, the remaining $N_p-1$ processors may all request the lock, each queue itself, and busy wait. Each waiting processor occupies one element in the array. As a result, the size of the array must be equal to or greater than the number of processors in the system, i.e., $N_a \geq N_p$, where $N_a$ is the size of the array and $N_p$ is the number of processors in the system. We impose an additional constraint on the size by requiring it to be a power of 2, i.e., $N_a = 2^k$, where ‘$k$’ is an integer\(^2\). The additional constraint provides a safeguard against a potential failure situation that may occur when the variable `Tail` rolls over from its maximum value to 0. Further, it provides an efficient form of modulo division since $X \& (N_a-1)$ is equal to $X \mod N_a$ for $N_a = 2^k$. The logical bit-wise AND operation is considerably faster than the division operation on any machine.

```c
struct lock {
    unsigned int Tail;
    unsigned int Que[N_a];
}Lock;
```

**Figure 7. Data Structure for Anderson's Array Based Queuing Lock**

Each element of the array is used as a binary flag, one value indicating the lock is free and the other indicating the lock is busy. From functional point of view the array may be an array of bits. However, to avoid contention, each element of the array must be assigned a separate cache line in a cache coherent architecture and a separate memory module in a multistage interconnection network based architecture. For our presentation we use an

---

\(^2\) In his original algorithm, Anderson does not impose this constraint. As a result the algorithm may fail when `Tail` rolls over from its maximum value to 0. In their variant, Mellor-Crummey and Scott correct the problem using a different approach [73].
array of unsigned integers. The data structure is presented in Figure 7. We also graphically show an instance of the data structure in which four processors are busy waiting each on a cached copy of their indexed element.

The lock is initialized to free by initializing Tail to index the element that is initialized to LOCK. The rest of the elements are initialized to WAIT to indicate they do not have the lock. When a processor finds LOCK in its indexed element, it owns the lock. On the other hand, when a processor finds WAIT in its indexed element, it must wait until its indexed element is passed the token LOCK.

**Initialization**

initialize(struct lock *L)

L->Tail := i; 0≤i≤Na // choice of the element is arbitrary
L->Que[n] := LOCK; n=i // the element pointed to by Tail
 := WAIT; 0≤n≤Na; n≠i // must be set to LOCK

**Acquiring the Lock**

acquire(struct lock *L, unsigned int *my_index)

// my_index points to a processor local variable.
// X & (Na-1) is equal to X mod Na for Na=2^k
*my_index := fetch&add(&L->Tail,1) & (Na-1)
repeat while (L->Que[*my_index] ≠ LOCK)

**Releasing the Lock**

release(struct lock *L, unsigned int *my_index)

L->Que[*my_index] := WAIT // initialize for the next time
L->Que[(*my_index+1)&(Na-1)] := LOCK // pass on the lock to the next
 // element

**Figure 8. A Variant of Anderson's Array Based Queuing Lock**

Additionally, each processor uses a local variable of the same type as Tail. It passes a pointer to this variable as a parameter to the function that acquires the lock. To acquire the lock, a processor atomically adds one to Tail reading back its pre-addition value. The
pre-addition value gives the processor its request number the lower 'k' bits of which provide the index of the corresponding element in the array. The processor spins on the value of its indexed element. If the lock is free the loop immediately terminates. If the lock is not free the requesting processor waits spinning on the indexed element.

To release the lock, a processor sets its own element to WAIT and sets the next element in the array to LOCK. If a processor is waiting for the lock it will get the lock. Otherwise, the lock becomes free.

As Anderson notes, the performance of the algorithm depends on the underlying architecture. It eliminates contention in cache coherent shared bus architectures as long as each element of the array is assigned a separate cache line. Likewise, it evenly distributes traffic in multistage interconnection network based architectures as long as each element of the array resides in a separate memory module. As a result, it eliminates hot spots. However, it does not help eliminate contention in architectures that do not have coherent caches.

2.2.5.2 Array/Linked List Based Queuing Lock

The next lock is due to Graunke and Thakkar [32]. It uses an array, the elements of which are statically associated with the processors in the system. There is a one to one correspondence between the array elements and the processors. Functionally, however, it builds an implicit linked list of waiting processors. We present a variant of this lock.

The lock consists of a data structure which is almost identical to the data structure for Anderson's lock, a pointer (rather than an integer) and an array. The pointer which we will call Tail, except for its least significant bit, points to the array element associated with the last processor that requested the lock, i.e., it serves as tail. The least significant bit of Tail specifies the value of token LOCK used by the last processor. When the lock is free, the value of the element pointed to by Tail is LOCK. When the lock is busy, the value of the element is WAIT. The processor requesting the lock waits spinning on the element associated with its predecessor, i.e., the last processor that requested the lock prior to it, until the element changes from WAIT to LOCK. Conceptually, Tail serves a similar role as it did in Anderson's algorithm.
Each element of the array is used as a binary flag. However, to eliminate contention each element is assigned a different cache line in a cache coherent architecture or a separate memory module in an interconnection network based architecture. Further, each element must reside on an even address. This allows the use of the least significant bit of the address as an independent quantity; the need for this is explained in the next paragraph.

For reasons to be explained later, LOCK in this algorithm is not a statically assigned quantity. Instead it needs to change between at least two values. In addition to pointing to the array element associated with the last processor that requested the lock, Tail also contains the information about what value is specified as LOCK by the last processor requesting the lock. Since LOCK and WAIT can be represented by a single bit, it is possible to pack the identity of LOCK and the address of an array element in a single long word provided the array elements are aligned on the even boundaries. Each element of the array holds either LOCK or WAIT. The sense of LOCK and WAIT is such that WAIT = \( \neg \text{LOCK} \), i.e., one is the complement of the other. When we change one the other is implicitly changed.

```c
struct lock {
    unsigned int *Tail;
    unsigned int Que[N_a];
} Lock;
```

**Figure 9. Data Structure for Graunke and Thakkar's Queuing Lock**

The data structure for each lock, therefore, consists of an array and a variable Tail, both declared in the global shared memory as shown in Figure 9. The size of the array is \( N_a \) such that \( N_a = N_p \), where \( N_p \) is the number of processors. Each element of the array is declared as an unsigned integer even though it is used only as a binary flag. We also show
an instance of the data structure where two processors are busy waiting each spinning on the array element associated with its predecessor.

The lock is set to free by initializing Tail to point to an array element which itself is initialized to LOCK; the least significant bit of Tail is initialized to LOCK. The absolute value of LOCK is not important. However for the LOCK to be free, the value in the least significant bit of Tail must be equal to the value of the element to which Tail otherwise points. This choice is opposite to the choice made by Graunke and Thakkar [32] and later by Mellor-Crummey and Scott [73]. We made this choice to be consistent in our presentation so that the value LOCK always means the lock is free. The initialization of the rest of the elements is irrelevant.

**Initialization**

initialize(lock struct *L)

```
L->Tail := &Q[i] | (L->Que[i] & 0x1); 0<=i<N_a
// The choice of the element is arbitrary; the least significant bit of Tail,
// however, must be set equal to the value in the element pointed to by Tail.
// The rest of the elements do not need to be initialized.
```

**Acquiring the Lock**

acquire(struct lock *L)

```
// tail, pred, and LOCK are local variables.
// Every processor knows its associated element, my_index.
tail := swap(&L->Tail, (&L->Que[my_index] | (L->Que[my_index] & 0x1)))
LOCK := tail & 0x1 // get the value of token LOCK from tail
pred := tail & ~0x1 // get the pointer to predecessor's array element
repeat while(*pred != LOCK)
```

**Releasing the Lock**

release(struct lock *L)

```
L->Que[my_index] := L->Que[my_index] & 0x1
```

Figure 10. A Variant of Graunke and Thakkar’s Queuing Lock
To acquire the lock, a processor concatenates the address of its element with the new value for lock, i.e., the complement of the value in the first bit of its element, and swaps it with the value in Tail. The processor derives a pointer, pred, and the associated value of lock, LOCK, from tail. Next, the processor spins until the value of the element pointed to by pred becomes equal to LOCK. When the lock is free, the value of the element is already equal to LOCK, and the loop immediately terminates. When the lock is busy, the processor waits spinning on the element. The waiting processors thus build an implicit linked list.

To release the lock, a processor complements the least significant bit of its associated element. The alternating value of LOCK avoids the race condition that would exist if a constant is used as LOCK. The performance of this lock is similar to Anderson's lock.

Craig has extended this lock to use fixed memory, provide FIFO behavior with local only spinning in systems without coherent caches using swap, and provide priority locks [17]. In his extension the lock consists of two elements in the global memory. One of them serves the same role as Tail does in the original lock. The other, we will refer to as sLock, holds the state of the lock. When the lock is free, Tail points to sLock and sLock indicates the lock is free. To acquire the lock, a processor initializes its own sLock structure in the global memory to indicate the lock is not free and swaps it with the pointer in Tail. Next, it examines the value of the sLock returned from Tail. If the value indicates the lock is free, it acquires the lock. Otherwise, it waits spinning on the sLock returned from Tail. This eliminates contention on a system with coherent cache. Once a processor acquires the lock, it also owns the sLock data structure returned from Tail. The processor may release it or use it for the next time. To release the lock, a processor changes the value of the sLock that it put in Tail when it acquired the lock. As a result, the lock is passed to the next waiting processor or becomes free.

In the extension in which processors wait in their local shared memories, a processor first swaps the address of its local spin variable with the value in sLock returned from Tail. If the returned value from sLock indicates the lock is free, it acquires the lock. Otherwise, it busy waits spinning on its spin variable. To release the lock, it swaps the value indicating the lock is free with sLock it put in Tail. If it finds a processor is waiting, it updates the spin variable of the waiting processor passing it the lock.

39
2.2.5.3 Linked List Based Queuing Lock

This lock was invented by Mellor-Crummey and Scott [72][73]. The lock consists of a pointer which is NULL when the lock is free and points to a data structure associated with the last processor in the linked list of waiting processors when the lock is busy. The pointer, therefore, acts as tail of the queue of the waiting processors.

The waiting processors build an explicit linked list. Therefore, the structure associated with each processor may reside anywhere in memory as long as it is accessible to the rest of the processors. This allows each processor to use a structure in its local shared memory. A processor that needs to busy wait may spin on a variable in its local shared memory. This eliminates contention on architectures with or without coherent caches.

```c
struct lock {
    struct proc *Tail;
} Lock;

struct proc {
    struct proc *succ;
    unsigned int spin;
} Proc;
```

**Figure 11. Data Structures for Mellor-Crummey and Scott's Queuing Lock**

The data structure for this lock is shown in Figure 11. Per lock data consists of a variable Tail that resides in the global shared memory. The data structure associated with each processor used to link behind the predecessor and busy wait is a per processor, rather than per lock, data structure. Each processor may either statically define or dynamically create its data structure in its local shared memory. The data structure has two elements succ and spin. Succ is a pointer for a successor to link behind it and spin, even though declared as an unsigned integer, is a binary flag to busy wait when the lock is busy.
The lock is set to free by initializing `Tail` to `NULL`. A processor `Pj` requesting the lock swaps the contents of `Tail` with a pointer to its data structure in its local shared memory and tests the returned value. If the returned value is `NULL`, the lock is free, and `Pj` acquires the lock. If the returned value is not `NULL`, the lock is busy, and the returned value points to a predecessor `Pₚ`, the last processor that requested the lock prior to `Pj`. `Pj` prepares to spin, links itself behind `Pₚ`, and waits spinning on the `spin` element of its data structure in its local shared memory. When `Pₚ` releases the lock, it changes `Pj`'s `spin` variable from `WAIT` to `LOCK`, passing it the lock.

**Initialization**

```c
initialize(struct lock *L)
L->Tail := NULL   // NULL is defined to be 0.
```

**Acquiring the Lock**

```c
acquire(struct lock *L, struct proc *I)
I->succ := NULL
pred := swap(&L->Tail, I)   // pred is a local variable of type proc
if (pred != NULL)   // if there is a predecessor, lock is busy
  I->spin := WAIT   // prepare to spin
  pred->succ := I   // link behind the predecessor and wait
repeat while (I->spin != LOCK)
```

**Releasing the Lock using Compare&swap**

```c
release(struct lock *L, struct proc *I)
if (I->succ == NULL)
  if (compare&swap(L, I, NULL) == I)
    return   // lock is successfully released
  repeat while (I->succ == NULL)
  I->succ->spin := LOCK
```

**Figure 12. Mellor-Crummey and Scott's Queuing Lock**

To release the lock, `Pj` checks if it has a successor by checking its `succ` element. If a successor is waiting, `Pj` changes its `spin` element, passing it the lock. If it does not have a
successor, it attempts to free the lock by changing Tail to NULL as shown in Figure 14. Another processor, however, may request the lock during time \( \Delta T_u \), the time between when \( P_j \) checks to see it has a successor and when it actually updates Tail to NULL. Using compare&swap \( P_j \) atomically updates Tail only if it still points to \( P_j \) at the time of the update. If another processor requesting the lock has changed Tail, compare&swap does not modify it. The processor waits for the next processor requesting the lock, to link behind it. After which it passes the lock to the next processor. This ensures the FIFO behavior.

When compare&swap is not available, a processor releasing the lock attempts to free the lock using swap. The code for this is given in Figure 13. When swap is used instead of compare&swap, the FIFO behavior is not guaranteed. Further, the possibility of starvation is also introduced.

\[\text{Releasing the Lock using swap}\]

```c
release(struct lock *L, struct proc *I)
if (I->succ == NULL)
    tail := swap(&L->Tail, NULL) // L->Tail is changed to NULL
    if (tail == I)
        return // the lock is successfully released
    tail := swap(&L->Tail, tail) // attempt to restore L->Tail
repeat while(I->succ == NULL) // wait for the successor to link
    if (tail == NULL) // L->Tail is restored
        I->succ->spin := LOCK // pass lock on to the successor
        else // could not restore L->Tail
            tail->succ := I->succ // link the successor behind tail
        else
            I->succ->spin := LOCK // pass lock on to the successor
```

Figure 13. Release in Mellor-Crummey and Scott's Lock Using Swap

To release the lock, a processor, e.g., \( P_j \), checks for a successor and changes Tail to NULL using swap when there is no successor waiting for the lock. If no processor requests the lock between when \( P_j \) checks for a successor and when it changes Tail to NULL, i.e.,
during time $\Delta T_u$, the return value from \texttt{swap} points to $P_j$. This indicates $P_j$ has successfully released the lock.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{scenario.png}
\caption{A Scenario Leading to Loss of FIFO Behavior}
\end{figure}

However, if the value returned from \texttt{swap} does not point to the releasing processor, one or more processors requesting the lock have modified \texttt{Tail} during time $\Delta T_u$, as shown in Figure 14. The \texttt{swap} returns a pointer to the last processor $P_i$ requesting the lock before...
P_j changed Tail to NULL. P_j attempts to point Tail back to P_i. The time from when P_j changed Tail to NULL and when it attempts to change it back to P_i represents the window of vulnerability \( \Delta T_v \). If no other processor requests the lock during this time, P_j succeeds in pointing Tail back to P_i and preserving the FIFO behavior. However, another processor P_m may request the lock during \( \Delta T_v \), as shown in Figure 14. If that happens, P_m acquires the lock since Tail is NULL at the time. As a result, the FIFO behavior is lost. P_j waits until the next processor, P_k, requesting the lock links behind it and then links processors P_k...P_i behind P_n, the processor Tail was pointing to when Tail was switched to P_i.

The original version of this lock that uses \textit{compare} & \textit{swap} guarantees the FIFO behavior, spins only on local shared memory, requires a single pointer variable for the lock, and works equally well in architectures with or without coherent caches. The version that uses \textit{swap} as the only atomic primitive, does not guarantee the FIFO behavior and introduces the possibility of starvation.

### 2.3 Synchronization Using \textit{Swap} or \textit{Fetch} & \textit{Add}

In this section, we present a somewhat obvious result showing that the \textit{swap} read-modify-write primitive is adequate to synchronize any two processors in a system of N processors using a single synchronization variable. Further, if synchronization causes one of the processors to wait, it may always wait in its local shared memory. This result is not significant in itself. What is significant, however, is the observation that more than one synchronization variable may be combined to provide capabilities previously considered infeasible using \textit{swap} or \textit{fetch} & \textit{add}. We exploit this fact to build a singly linked list with guaranteed FIFO behavior. We then use this linked list in this and the following chapters to develop fair mutual exclusion and readers-writers locks and a lock free multiple producer single consumer. The discussion in the following two sub-sections is equally valid for \textit{fetch} & \textit{add}.

#### 2.3.1 Synchronizing 2 Processors in a System of N Processors

Let \( v \) be a synchronization variable and P_i and P_j two processors that may concurrently update it. Further, let the initial state of \( v \) be \( s_0 \) and known to both processors. The first update changes the state to \( s_1 \) and the second update changes the state to \( s_2 \).
Since there are only two processors that may update \( v \), there are only two possible update sequences: \( P_i \) precedes \( P_j \) or \( P_j \) precedes \( P_i \). If \( P_i \) precedes \( P_j \), the return value from \textit{swap} indicates \( v \) was in state \( S_0 \) prior to its update by \( P_i \). On the other hand, if \( P_j \) precedes, the return value from \textit{swap} indicates \( v \) was \textit{not} in state \( S_0 \) prior to its update by \( P_i \); it implies it must be in state \( S_1 \). While updating the variable \( v \), processors may use their identity as a part of the update. As a result, states \( S_1 \) and \( S_2 \) may provide the identity of the processor that caused the corresponding state change. Knowing the order in which the synchronization variable is updated and the identity of the updating processor, the two processors may synchronize and coordinate their activities. Since the processor identities are known through \( S_1 \) and \( S_2 \), \( P_i \) and \( P_j \) may be any two processors out of the \( N \) processors in the system. The result equally applies to \textit{fetch&add} as well.

2.3.2 Contention Free Busy Waiting

As an example, two processors may use a synchronization variable \( v \) to acquire a lock. The processor that may have to wait may spin on a variable in its local shared memory. The lock may be granted in the order in which it is requested. We may assign the following meanings to states associated with the synchronization variable \( v \).

- \( S_0 \): The contents of \( v \) are a unique value indicating the lock is free.
- \( S_1 \): The contents of \( v \) are a unique value indicating the lock is busy. However, there is no other processor waiting for the lock. The unique value may be a pointer to a data structure associated with the processor holding the lock.
- \( S_2 \): The contents of \( v \) point to a data structure associated with the processor waiting for the lock. The lock is acquired by the other processor.

To acquire the lock, a processor \( P_i \) prepares to wait by writing \textit{wait} in its variable \textit{spin} and then swaps the contents of \( v \) with the address of \textit{spin}. If the \textit{swap} returns \( S_0 \), \( P_i \) acquires the lock since the lock was free. Otherwise, the variable \( v \) must be in state \( S_1 \), the processor waits spinning on its local variable \textit{spin}. Since we limited the number of processors to two, a requesting processor will find the lock either in state \( S_0 \) or \( S_1 \).
A processor, \( P_i \), releasing the lock will find \( V \) in either state \( S_1 \) or \( S_2 \). If \( V \) is in state \( S_2 \), the second processor, \( P_j \), has already requested the lock; see Scenario I in Figure 15. The releasing processor updates the spin variable of \( P_j \) passing it the lock. This effectively changes the state from \( S_2 \) to \( S_1 \). If \( P_j \) however, finds \( V \) in state \( S_1 \), it attempts to change \( V \) to state \( S_0 \) using \texttt{swap}. The time between when \( P_i \) checks \( V \) and when it actually updates it, is \( \Delta T_u \). The other processor, \( P_j \), may request the lock during \( \Delta T_u \), in which case \( P_i \) has actually changed the state from \( S_2 \) to \( S_0 \). The previous state is indicated by the value returned from \texttt{swap}. If the return value indicates \( V \) was in state \( S_1 \), \( P_i \) has freed the lock which later may be acquired by \( P_j \). This is depicted in Scenario II in Figure 15. On the other hand, if the return value from \texttt{swap} is not \( S_1 \), \( P_j \) has requested the lock during time \( \Delta T_u \). The value returned from \texttt{swap} points to \( P_j \). Since \( P_j \) has already requested the lock, no processor other than \( P_i \) will modify \( V \). Therefore, \( P_i \) may use a simple write to write the value returned from \texttt{swap} back to \( V \). This is followed by modifying the spin variable of \( P_j \) passing it the lock. This sequence is shown in Scenario III in Figure 15.

![Figure 15. Resolving a Resource Request Between Two Processors](image)

It is possible to combine more than one synchronization variable to build interesting and useful data structures. As we will show in Section 2.4.3, we have built a linked list with guaranteed FIFO behavior using two variables. This is a useful data structure that we use to build fair mutual exclusion locks with local only spinning.
2.4 New Scalable Mutual Exclusion Algorithms

In addition to building a lock using a singly linked list with guaranteed FIFO behavior using swap as the only atomic primitive, we also provide two simple extensions to the original locks of Anderson and Graunke & Thakkar. Each of the new algorithms allows a processor to acquire the lock immediately if the lock is available or queue itself at the tail of the waiting processors and wait if the lock is busy. Each of these algorithms guarantees the FIFO behavior, spins on its local shared memory for busy waiting, and requires $O(1)$ transactions for each lock acquisition with or without coherent caches.

The first algorithm uses fetch&add, requires per lock memory linear to the number of processes, and extends Anderson's lock to architectures with or without coherent caches [8]. Anderson's lock benefits by spinning on a cached copy of a variable. We extend this by allowing a waiting processor to put a pointer to its spin variable in the queue. The waiting processor can now spin on its spin variable which may reside in its local shared memory. The processor releasing the lock will recognize a processor is waiting and change the spin variable of the waiting processor, passing it the lock.

Using an approach similar to the one above, i.e., putting a pointer in the queue, the second algorithm extends Graunke & Thakkar's lock to architectures that do not support coherent caches [32]. This algorithm uses atomic swap and requires per lock memory linear to the number of processes. Graunke & Thakkar's lock has characteristics similar to Anderson's lock in that it spins on a cached copy of the flag indicating whether the lock is free. When the lock is freed, the flag is changed, the cached copy is updated, and the processor waiting for the lock gets the lock. Instead of waiting on a cached copy of the flag, we put a pointer to the spin variable in the queue. This allows the waiting processor to busy wait in its local shared memory eliminating contention on architectures with or without coherent caches.

The third lock builds on the mutual exclusion lock of Mellor-Crummey and Scott [73]. Their lock does not guarantee the FIFO behavior when it uses swap as the only atomic primitive. We have developed a lock that does guarantee the FIFO behavior using swap as the only read-modify-write primitive at the expense of one extra global variable. It is similar to their lock otherwise.
2.4.1 Array Based Queuing Lock Using Fetch&Add

This algorithms is an extension of Anderson's queuing lock [7][8]. It uses an unsigned integer variable and a circular array, both of which are declared in the global shared memory. The unsigned global variable Tail, is used to assign consecutive numbers to processors in the order in which they request the lock. Further, it is used to derive the index of the corresponding element in the array. The array is used by the waiting processors to 'stand in line' and busy wait for their turn to get the lock. If the lock is free, the indexed element holds a token indicating the lock is free. Otherwise, the processor requesting the lock uses the element to queue itself and busy wait.

The manner in which a waiting processor busy waits differentiates this algorithm from Anderson's lock. In Anderson's lock, a processor waits spinning on the indexed element itself until the indexed element indicates the lock is free. In a cache coherent architecture the processor spins on a cached copy of the element. However, in an architecture without a coherent cache the processor continues to spin on the indexed element itself providing no relief from excessive accesses to the global memory over the bus.

Rather than relying on a coherent cache, a processor may leave the address of its spin variable in the indexed element before it busy waits. Its spin variable may reside in its local shared memory allowing it to spin locally. Our algorithm follows this approach to eliminate contention on architectures with or without coherent caches. A processor trying to acquire the lock and a processor releasing the lock are the only two processors that may concurrently update a given element in the array. The two may synchronize using swap or fetch&add as shown in Section 2.3.

2.4.1.1 Data Structure

The per lock data consists of an array and an unsigned integer variable Tail, both declared in the global shared memory known to all processors. Further, an unsigned integer spin used as a binary flag is declared in the local shared memory of each processor to busy wait when necessary; this is per processor rather than per lock data structure. The size of the array is Na such that Na ≥ Np and also Na = 2^k; where 'k' is an integer. The array is circular with element 0 following the element Na-1. Each element of the array is declared as a pointer and may hold one of the following three distinct values.
1. A NULL interpreted by a processor requesting the lock as an indication that the token is not available, i.e., the lock is busy, and interpreted by a processor releasing the lock that there is no waiting processor.

2. A token LOCK interpreted by a processor requesting the lock as an indication that the lock is free. A processor releasing the lock may never encounter this value.

3. A pointer to the spin variable of the next processor waiting for the lock. This is used by the processor releasing the lock to pass the lock on to the next waiting processor. A processor releasing the lock encounters this value only when a processor is waiting for the lock. On the other hand, a processor requesting the lock may never encounter this value.

```c
struct lock {
    unsigned int Tail;
    unsigned int *Que[Np];
} Lock;
unsigned int spin;
```

**Figure 16. Data Structure for the Array Based Lock Using Fetch&Add**

Since a processor must release a lock before it requests the same lock again, the number of processors using the lock and requesting the lock could, at most, be Np, where Np is the number of processors in the system. If the processor using the lock occupies the array element indexed 'i' then, at the most, there could be Np-1 waiting processors occupying Np-1 consecutive elements in the array after the element indexed 'i'. The releasing processor only modifies element 'i+1' when it releases the lock. Therefore, an array of size Na such that Na ≥ Np, ensures the next element in the array is available when a processor requests the lock and no element is incorrectly over-written when a processor requests or
releases the lock. The global data structure is shown in Figure 16. An instance of the data structure with four waiting processors is also shown in Figure 16.

**LOCK** must be a unique bit pattern distinguishable from the valid pointers that the waiting processors use to point to their spin variables. The address of Tail or the address of any of the array elements may provide such a unique bit pattern.

### 2.4.1.2 Algorithm \( \mu E-1 \)

The algorithm is presented in Figure 17. The lock is initialized to free by setting all the elements of array \( L->Q[] \) to **NULL** except one which is set to **LOCK**. Also, Tail is initialized such that the lower 'k' bits of its value provide index to the element which is set to **LOCK**.

To acquire the lock, a processor first prepares its spin variable for busy waiting and then atomically adds '1' to variable Tail reading back its pre-addition value. The pre-addition value gives the processor its request number, the lower 'k' bits of which provide the index of the corresponding element in the array: \( k=\log_2(N_a) \) where \( N_a \) is the size of the array. If the lock is free, the indexed element holds the token **LOCK** and the processor may acquire the lock. If the lock is not free, the indexed element is **NULL** and the requesting processor must wait. It updates the element to point it to its spin variable and then waits spinning on its spin variable in its local shared memory. The processor holding the lock, however, may release the lock any time and update its next element. The two processors thus may update the same element and their updates must be synchronized. One may use swap or fetch&add to synchronize the two. We use fetch&add in this algorithm. The processor requesting the lock atomically adds the address of its spin variable to the indexed element and checks the return value from fetch&add. If the return value is not **LOCK**, the processor waits spinning on its spin variable until it is passed the token **LOCK**. If the return value is **LOCK**, the processor can immediately acquire the lock.

To release the lock, a processor sets its own element to **NULL** and checks the next element in the array. If the next element is **NULL** there is no successor waiting for the lock. The processor attempts to release the lock by atomically updating the next element to **LOCK** using fetch&add. If the next element is **NULL** and the return value from fetch&add is also **NULL**, the next element is updated to **LOCK**. The lock is free, and no further action is
needed. If the next element is not NULL or the return value from \texttt{fetch\&add} is not NULL, there is a successor waiting for the lock. The local variable \texttt{succ} points to the spin variable of the successor. We pass the lock on to the successor.

**Initialization**

\begin{verbatim}
initialize(struct lock *L)
L->Tail := i; 0\leq i < N_a  // i is arbitrary.
L->Que[n] := LOCK; n = i \& (N_a-1)  // LOCK may be \&L->Tail.
    := NULL; 0\leq n < N_a; n \neq i \& (N_a-1)
\end{verbatim}

**Acquiring the Lock**

\begin{verbatim}
acquire(struct lock *L, unsigned int *my_index)
    // my_index points to a processor local variable.
    // X \& (N_a-1) is equal to X \mod N_a for N_a=2^k
1. spin := WAIT  // WAIT\&LOCK; may be NULL
2. *my_index := fetch\&add(&L->Tail,1) \& (N_a-1)
3. if (L->Que[*my_index] \neq LOCK)
4.   if (fetch\&add(&Que[*my_index],&spin) \neq LOCK)
5.     repeat while (spin \neq LOCK)
\end{verbatim}

**Releasing the Lock**

\begin{verbatim}
release(struct lock *L, unsigned int *my_index)
    // next_index and succ (short for successor) are local variables.
1. L->Que[*my_index] := NULL  // initialize for the next time
2. next_index := (*my_index+1) \& (N_a-1)  // next index in a circular manner
3. succ := L->Que[next_index]
4. if (succ == NULL)  // no successor, try to free the lock
5.   succ := fetch\&add(L->Que[next_index],LOCK)
6. if (succ \neq NULL)  // there is a successor.
7.   *succ := LOCK  // pass the lock
\end{verbatim}

Figure 17. Code for the Array Based Lock (μE-1)
2.4.1.3 Discussion

The ticket lock presented by Mellor-Crummey and Scott [73], Anderson's lock [8], and the new lock presented in this section may all be viewed as enhanced versions of a bakery lock using atomic fetch&add. In these locks a process requesting the lock takes a number by atomically incrementing a global shared variable. The difference lies in how a process waits for its turn to be served. In the ticket lock, a process continues to check whether the next number to be served is the number that the process holds. In the ticket lock with proportional back off the frequency at which a process checks is adjusted based on the difference between the number being served and the number the process holds. In Anderson's lock, a process itself checks if it is at the head of the queue. In our lock a process is told by the process ahead of it when it is at the head of the queue. Our lock achieves for architectures with or without coherent caches what Anderson's lock achieves for architectures with coherent caches. Anderson's lock, in turn, achieves for architectures with coherent caches based on any protocol [27][28][48][82] what the ticket lock achieves for architectures with coherent caches based on distributed write or update protocol [69][104]. One may see these algorithms on a continuum where functions, assisted by the hardware, are gradually relegated to software. Accordingly, the hardware may be simplified to realize savings in cost.

Most modern processors support the atomic swap primitive while a number of them support the atomic fetch&add primitive as well. In this algorithm, fetch&add is required. If a machine does not directly support it, it itself will have to be built using a lock. That may not be very attractive given contention free locks requiring swap as the only atomic primitive are available. There may be machines that support fetch&add but do not support swap. This algorithm is a good candidate for such machines.

The restriction \( N_a = 2^k \) ensures that the effective size of the queue will not reduce to less than \( N_a \) when the variable \( \text{Tail} \) rolls over from \( 2^n - 1 \) to 0. If \( N_a \neq 2^k \), \( 2^n \) is not an integer multiple of \( N_a \). As a result \( 2^n - k \cdot N_a < N_a \) for some \( k \), effectively reducing the size of the queue to \( 2^n - k \cdot N_a \) which we will call \( n_a \). If \( n_a \geq N_p \), still there is no problem. However, if \( n_a < N_p \), a processor may incorrectly acquire the lock out of order and before the processor originally holding the lock has released the lock. Other waiting processors may over-write

52
the pointers of the previously waiting processors, blocking them forever. To avoid these
disastrous consequences, $2^\alpha \cdot 2^k N_p$ must be equal or greater than $N_p$. An easy way to en-
sure this is to make $2^n$ an integer multiple of $N_p$, i.e., $N_p = 2^k$, where $k < n$. An additional
advantage of this restriction is that one may use a mask operation to compute modulo
division. A mask operation is much faster than division and will improve the performance.
In their version of Anderson's lock, Mellor-Crummey and Scott use a different approach
to take care of this boundary problem [73].

2.4.2 Array/Linked List Based Queuing Lock Using Swap

This algorithm is a simple extension to Graunke and Thakkar's lock [32]. Using an ap-
proach similar to the one used in Section 2.4.1, Graunke and Thakkar's lock can also be
extended to eliminate contention on architectures with or without coherent caches. The
data structure is similar to the data structure in Graunke and Thakkar's lock except that
the array is an array of pointers and Tail is a pointer that points to the array element
associated with the last processor that acquired the lock. Tail does not include the value
of lock as it does in Graunke and Thakkar's lock. Graunke and Thakkar used alternating
values of lock to avoid a race condition. Since they needed to know the value of the lock,
they stored it in Tail along with the pointer.

Tail always points to the last processor that acquired the lock. The last processor must
provide some means to indicate whether it is still holding the lock or it has released the
lock. It provides this indication using two distinct values in its associated array element
to which Tail points. The next processor requesting the lock refers to the array element
pointed to by Tail to determine whether the lock is free. Since a processor releasing the
lock has no knowledge of when the next processor may request the lock, it can not release
or re-use this memory except for again acquiring the lock it just released.

Let us assume the lock is free. Tail points to Que[i] and Que[i] holds LOCK. Let us fur-
ther assume processors $P_j$, $P_j$, and $P_k$ concurrently attempt to acquire the lock.

As a first step each processor checks its associated element to see if it holds LOCK. $P_j$ will
find its associated element does hold LOCK while $P_j$ and $P_k$ will find otherwise. $P_j$ will at-
tempt to acquire the lock by atomically changing its associated element to NULL. $P_j$ and
P_k will also set their associated elements to NULL and then attempt to acquire the lock by first queuing themselves at the tail and then checking whether their predecessor, the processor to which Tail points, has released the lock. Let us assume P_j queues itself ahead of P_k. P_j will find Tail points to Que[i] while P_k will find Tail points to Que[j]. P_i and P_j will synchronize using Que[i] as the synchronization variable to determine who may acquire the lock; likewise, P_j and P_k will synchronize using Que[j]. At any given time, only two processors attempt to synchronize using an array element as a synchronization variable. As we know, two processors can always synchronize using a synchronization variable and swap as we discussed in Section 2.3.1. The processor that may have to wait, will busy wait spinning on its spin variable in its local shared memory. If P_i succeeds, P_j waits behind P_i and will get the lock when P_i releases the lock. On the other hand, if P_j succeeds, P_i will queue itself at the tail and wait for the lock.

2.4.2.1 Data Structures

The per lock data structure consists of an array and a variable Tail, both declared in the global shared memory known to all processors. Further, an integer variable spin, used as a binary variable, is declared in the local shared memory of each processor to busy wait when necessary. The size of the array is N_a such that N_a = N_p, where N_p is the number of processors. Each element of the array is declared as a pointer and may hold one of the following three distinct values.

1. A NULL interpreted by a processor requesting the lock that the token is not available, i.e., the lock is busy, and interpreted by a processor releasing the lock that there is no waiting processor.

2. A token LOCK interpreted by a processor requesting the lock that the lock is free. A processor releasing the lock may never encounter this value.

3. A pointer to the spin variable of another processor. A processor releasing the lock interprets it as the address of the spin variable of the next waiting processor and passes the lock onto it. A processor requesting the lock may encounter this value in its associated array element. It is the address of the spin variable of the processor that acquired the lock right after this processor released the lock last time. The processor must check further to find out whether the lock is free.
struct lock {
    unsigned int *Tail;
    unsigned int *Que[N];
}Lock;

unsigned int spin;

Figure 18. Data Structure for the Array/Linked List Based Lock Using Swap

Tail always points to the last processor that requested the lock. Whether that processor has released the lock is indicated by the element associated with that processor. If it has released the lock, the element holds a token `LOCK` indicating the lock is free. If the processor is waiting for the lock or is holding the lock, the element holds a `NULL` value. Tail may be declared as an integer such that its value is an index into the array or a pointer such that it directly points to an element in the array. We somewhat arbitrarily chose it to be a pointer. The per lock global data structure, per processor local data structure, and an instance of the data structure with two waiting processors is shown in Figure 18.

2.4.2.2 Algorithm μE-2

The code for this algorithm is shown in Figure 19. The lock is initialized to free by initializing `Tail` to point to an element `Que[i]` and initializing `Que[i]` to `LOCK`. The choice of the element is arbitrary. The rest of the elements are initialized to `NULL`. Whenever the lock is free, `Tail` points to an element which holds `LOCK`.

To acquire the lock, a processor prepares its spin variable for busy waiting. Next, it checks if its associated element holds `LOCK` and, if it does, the processor atomically swaps `NULL` with the value in its associated element in an attempt to acquire the lock. If the return value is `LOCK`, the lock was previously free. No other processor has acquired the lock.
since this processor released it the last time. This processor may acquire the lock without any additional steps.

**Initialization**

initialize(struct lock *L)

L->Tail := &L->Que[i]  // Choice of the element is arbitrary.
L->Que[n] := LOCK; n=i  // LOCK may be &L->Tail.
:= NULL; 0≤n<Nₐ; n≠i

**Acquiring the Lock**

acquire(struct lock *L, unsigned int my_index)

// my_index is the index of the array element associated with a processor.
1. spin := WAIT  // WAIT≠LOCK; may be NULL
2. succ := L->Que[my_index]
3. if (succ == LOCK) // lock is free, attempt to lock
4. succ := swap(&L->Que[my_index], NULL)
5. if (succ ≠ LOCK) // could not get the lock
6. L->Que[my_index] := NULL  // change the element to NULL
7. pred := swap(&L->Tail, &L->Que[my_index])  // queue at the tail
8. if (swap(pred, &spin) ≠ LOCK) // link behind the predecessor, and
9. repeat while (spin ≠ LOCK) // wait for the lock if lock is not free

**Releasing the Lock**

release(struct lock *L, unsigned int my_index)

1. succ := L->Que[my_index]  // read to see if there is a successor
2. if (succ == NULL) // no successor, attempt to free
3. succ := swap(L->Que[my_index], LOCK)  // the lock
4. if (succ ≠ NULL) // there is a successor,
5. *succ := LOCK  // pass the lock, and

**Figure 19. Code for the Array/Linked List Based Lock (µE-2)**

If either its associated element did not hold LOCK or the return value from swap is not LOCK, some other processor has acquired the lock since this processor released it the last
time. This processor sets its associated element to NULL, queues itself at the tail, and attempts to acquire the lock. The processor atomically swaps a pointer to its associated element with the value in Tail. The return value points to the element associated with the last processor that requested the lock. The processor swaps the address of its spin variable with the contents of the element pointed to by the return value from Tail. If the returned value is LOCK, the lock is free and the requesting processor can acquire the lock. If the returned value is not LOCK, the last processor that requested the lock is either waiting for the lock or is holding the lock. In either case, this processor must wait. It waits spinning in its local shared memory. When the processor ahead of it releases the lock, the releasing processor will pass it the lock.

To release the lock a processor checks its associated element to see if a successor is waiting for the lock. If no successor is waiting, it atomically changes its own element to LOCK, reading back its current content. If the swap returns NULL, no processor is waiting for the lock. The lock has been set to free. If the swap returns a value other than NULL, a processor is waiting for the lock. The value returned by swap points to the spin variable of the waiting processor. The releasing processor restores the address of the spin variable of the waiting processor in its associated element and changes the spin variable of the waiting processor to LOCK passing it the lock.

2.4.2.3 Discussion

The manner in which a processor requesting the lock synchronizes with its predecessor eliminates the race condition for which Graunke and Thakkar use alternating lock values. In their original lock a processor requesting the lock only monitors the value in the array element associated with its predecessor. The predecessor has no way of knowing whether a processor requesting the lock has read the value of its associated element. This caused Graunke and Thakkar to use alternating lock values to avoid the race condition that otherwise would exist.

While we present this algorithm as a simple extension to Graunke & Thakkar's work, we do not recommend it because the next algorithm offers a better alternative for architectures that support swap as the only atomic primitive.
2.4.3 Linked List Based Queuing Lock Using Swap

The version of Mellor-Crummey and Scott's lock, that uses swap as the only read-modify-write primitive does not guarantee the FIFO behavior. The problem arises when a processor, \( P_j \), attempts to free the lock by changing \( \text{Tail} \) to NULL; refer to Figure 14. In what follows we show that a linked list can maintain the FIFO behavior using \( \text{swap} \) as the only read-modify-write primitive at the expense of one extra global variable. The lock that ensures the FIFO behavior, therefore, consists of two long words. One of these is \( \text{Tail} \) and serves the same purpose as it did in Mellor-Crummey and Scott's lock. The other, we call \( \text{Head} \), serves as the head of the new segment of the fragmented linked list starting with processor \( P_m \) of Figure 14. We use \( \text{Head} \) as an additional synchronization variable and impose the following discipline; refer to Figure 14.

1. The lock is free only when both \( \text{Tail} \) and \( \text{Head} \) together indicate the lock is free. Referring to Section 2.3.2, \( \text{Tail} \) must be in state \( S_0 \) and likewise \( \text{Head} \) must be in state \( S_2 \) for the lock to be free.

2. When a processor releasing the lock finds itself at the end of the list, it initializes \( \text{Head} \) to state \( S_1 \) as described in Section 2.3.2. Next, it swaps the contents of \( \text{Tail} \) with NULL in an attempt to free the lock. The returned value from \( \text{swap} \) points to the last processor, \( P_l \), that requested the lock immediately prior to the swap. This may be the processor releasing the lock itself or a different processor. In either case the processor releasing the lock links \( \text{Head} \) behind \( P_i \). Next, it waits for its \( \text{succ} \) element to be updated. It passes the lock on to the next processor if there is one. Otherwise it recognizes \( \text{Head} \) is linked behind it, in which case it attempts to change \( \text{Head} \) to state \( S_0 \) to release the lock.

3. When a processor, \( P_m \), requesting the lock swaps the contents of \( \text{Tail} \) with a pointer to itself and finds \( \text{Tail} \) is NULL, it is required to further check \( \text{Head} \) to determine whether it can acquire the lock. If \( \text{Head} \) is in state \( S_0 \), the lock is free. On the other hand if \( \text{Head} \) is in state \( S_1 \), the lock is not free. \( P_m \) will update \( \text{Head} \) to state \( S_2 \) and busy wait in its local shared memory.

\footnote{Since a processor releasing the lock links \( \text{Head} \) behind \( P_i \), the \( \text{succ} \) element of the releasing processor is already updated at this point if \( P_j \) is the releasing processor itself.}
The problem of maintaining the FIFO behavior is reduced to two processors, \(P_i\) and \(P_m\) and is the problem discussed and solved in Section 2.3.2: the equivalents of \(P_i\), \(P_m\) and \(s\text{head}\) are \(P_i\), \(P_j\) and \(v\) in Section 2.3.2. We can ensure the FIFO behavior as long as \(P_i\) and \(P_m\) can synchronize and ensure the FIFO behavior between the two of them. That we already know is possible according to our discussion in Section 2.3.2. Therefore, it is possible to ensure the FIFO behavior in a linked list using \textit{swap} as the only atomic primitive.

Conceptually, one may think of it as follows. \texttt{Tail} may segment the linked list of \(N\) processors into two groups when we attempt to change \texttt{Tail} to \texttt{NULL}. One of these groups is \(P_j \ldots P_i\) and the other is \(P_m \ldots P_n\) in Figure 14. These two segmented groups can be synchronized using one more variable since only the last processor in the first group and the first processor in the second group participate in this synchronization. Therefore, two synchronization variables let us build a behavior using \textit{swap} that otherwise requires more powerful \textit{compare\&swap} primitive.

2.4.3.1 Data Structures

A linked list consists of a head and tail and accordingly requires two global variables, \texttt{Head} and \texttt{Tail}. To maintain the FIFO behavior using \textit{swap} requires an additional variable \texttt{sHead}. The head of the list is implicitly known in a fair lock because the processor that owns the lock is always at the head. Therefore, the variable \texttt{Head} is eliminated and \texttt{Lock} consists of a structure with two members, \texttt{Tail} and \texttt{sHead} as shown in Figure 20.

```c
struct lock {
    struct proc *Tail;
    struct proc *sHead;
} Lock;

struct proc {
    struct proc *succ;
    unsigned int spin;
} Proc;
```

![Figure 20. Data Structure for the List Based Lock Using Swap](image-url)
The per processor data structure, consisting of two elements succ and spin, is the same as in Mellor-Crummey and Scott's lock. The modified data structure and an instance of the data structure with two waiting processors are shown in Figure 20.

2.4.3.2 Algorithm μE-3

The lock is initialized to free by initializing L->sHead and L->Tail both to NULL. We take $S_0$ to be NULL, $S_1$ to be L->sHead, and $S_2$ the address of the data structure of the waiting processor as the state values for synchronization variable sHead. While for state $S_2$ this is the only choice, for states $S_0$ and $S_1$ other choices are also possible.

To acquire the lock a processor first initializes its succ element to NULL and its spin variable to WAIT. Next, it swaps the contents of Tail with a pointer to its own data structure and examines the return value. If the return value which we assign to a local variable pred (short for predecessor) is not NULL, the lock is busy and pred points to a predecessor. The processor requesting the lock links behind the predecessor and waits spinning on its spin variable in its local shared memory. Steps so far are equivalent to the steps in Mellor-Crummey and Scott's lock.

If pred is NULL, indicating the Tail was NULL at the time this processor requested the lock, we require the processor requesting the lock take an extra step and examine the variable sHead. The processor requesting the lock swaps the value in sHead with a pointer to its data structure. If the return value is not NULL, the lock is busy and the requesting processor waits spinning on its spin variable in its local shared memory. On the other hand, if the return value from swap is NULL, the lock is free, and the processor can acquire the lock.

A processor releasing the lock checks Tail to determine if a successor is waiting for the lock. When Tail points to the releasing processor, no other processor is waiting for the lock, and sHead is not in use. Also, no other processor will attempt to modify sHead until Tail is changed to NULL. Therefore, the releasing processor can change sHead using a simple write. It changes sHead to point to sHead itself and swaps the contents of Tail with NULL. The return value from the swap points to the last processor, $P_j$, that requested the lock before the swap. The releasing processor points the succ element of $P_j$ to sHead.
If no other processor requested the lock between when the releasing processor checked Tail and when it actually swapped the contents of Tail, then \( P_1 \) is the releasing processor itself. In which case the releasing processor updated its own succ element to point to sHead.

**Initialization**

initialize(struct lock *L)

\[
\begin{align*}
L->\text{Tail} & := \text{NULL} & // S_{0,T} \text{ is defined NULL, i.e., 0.} \\
L->\text{sHead} & := \text{NULL} & // S_{0,H} \text{ is also defined as NULL, i.e., 0.}
\end{align*}
\]

**Acquiring the Lock**

acquire(struct lock *L, struct proc *I)

\[
\begin{align*}
I->\text{succ} & := \text{NULL} \\
I->\text{spin} & := \text{WAIT} & // prepare to spin \\
pred & := \text{swap}(\&L->\text{Tail}, I) & // pred is a local variable of type proc \\
\text{if} (\text{pred} \neq \text{NULL}) & & // if there is a predecessor, lock is busy \\
& \text{pred->succ} := I & // link behind the predecessor and wait \\
& \text{repeat while} \ (I->\text{spin} \neq \text{LOCK}) \\
\text{else if} (\text{swap}(\&L->\text{sHead}, I) \neq \text{NULL}) & & //If there is a successor, pass the \\
& \text{repeat while} \ (I->\text{spin} \neq \text{LOCK})
\end{align*}
\]

**Releasing the Lock**

release(struct lock *L, struct proc *I)

\[
\begin{align*}
\text{if} \ (L->\text{Tail} = I) & & // If success is sHead, swap NULL \text{ with the value in sHead} \\
\Delta T & \quad L->\text{sHead} := & \&L->\text{sHead} & // point sHead to itself \\
& \quad P_1 := \text{swap}(\&L->\text{Tail}, \text{NULL}) & // swap NULL with value in Tail, i.e., P_1 \\
& \quad P_1->\text{succ} := & \&L->\text{sHead} & // point \ P_1->\text{succ} \text{ to sHead} \\
\Delta T & \quad \text{repeat while} \ (I->\text{succ} = \text{NULL}) & // ensure I->\text{succ} is updated \\
& \quad \text{if} \ ((\text{succ}:=I->\text{succ}) = & \&L->\text{sHead}) & // if successor is sHead, swap NULL \\
& \quad \quad \text{succ} := \text{swap}(\&L->\text{sHead}, \text{NULL}) & // with the value in sHead \\
& \quad \text{if} \ (\text{succ} \neq & \&L->\text{sHead}) & // if there is a successor, pass the \\
& \quad \quad \text{succ->spin} := \text{LOCK} & // lock on to the successor
\end{align*}
\]

Figure 21. Code for the Linked List Based Lock (µE-3)
The releasing processor waits until its succ indicates a valid link. Even though the releasing processor does not differentiate, there are two cases that should be pointed out. The first case is when \( P_i = P_j \) assuming \( P_j \) is the releasing processor. In this case succ element of \( P_i \) (which is \( P_j \) itself) points to sHead and the wait loop immediately terminates. The other case is when \( P_i \neq P_j \). In this case \( P_j \) waits for the next processor requesting the lock to complete the link. This is the processor that requested the lock after \( P_j \) and may not be the same as \( P_i \); see Figure 14.

If Tail did not point to the releasing processor, the processor skips the steps that change Tail to NULL and link sHead behind the last processor that requested the lock before Tail is switched to NULL. However, it is still obliged to ensure that its succ element is updated.

At this stage, the succ element of the releasing processor points at either a waiting successor or sHead. If it points at sHead, the releasing processor swaps the contents of sHead with NULL and checks the return value. If sHead points to itself at the time of the swap as indicated by the return value, no processor is waiting and the lock is free as a result of the swap. If sHead does not point to itself, it points to the next waiting successor. The releasing processor passes the lock on to the waiting processor. After this point, sHead is irrelevant until it is used again in an attempt to free the lock. If the succ element of the releasing processor points to a waiting successor rather than sHead, the releasing processor simply passes the lock on to the waiting processor.

A possible sequence in which a processor \( P_j \) attempts to release the lock is shown in Figure 22. Processors \( P_k \) and \( P_i \) request the lock and change the contents of Tail in between when \( P_j \) checks and finds it is at the tail and updates Tail to NULL. \( P_j \) updates the succ element of \( P_i \) to point to sHead. We also show processors \( P_m \) and \( P_n \) requesting the lock in that order after Tail is switched to NULL. Since sHead is not changed to NULL, \( P_m \) points sHead to itself and busy waits spinning on its local variable spin. \( P_n \) follows \( P_m \), links behind it, and busy waits spinning on its variable spin. This sequence may be compared with the sequence in Figure 14.
2.4.3.3 Discussion

We can preserve the FIFO behavior of a linked list using *swap* as the only atomic primitive. This allows us to build a fair lock which guarantees the FIFO behavior and always spins on the local memory. Craig has also extended Graunke & Thakkar's lock to provide these characteristics [17]. However, his lock always requires three *swap* instructions to acquire the lock and one *swap* to release the lock and is somewhat more complex.
When the lock is heavily contended, i.e., a processor requests the lock before its predecessor frees the lock, the processor requesting the lock requires one swap and one simple write to queue itself while a processor releasing the lock requires one simple write to release the lock. As verified by the performance measurements, the lock is very efficient under contention.

When the lock is lightly contended the lock is either set to free before another processor requests the lock or a processor releasing the lock and another processor requesting the lock collide. Under these conditions, a processor uses one extra swap to acquire the lock and may use up to two extra swap instructions and a write to release the lock. The worst case occurs when a processor acquires a free lock and sets it back to free.

2.5 Summary

We have presented three new locks in this chapter. The first two extend the previous locks to architectures that do not support coherent caches. The first extends Anderson’s lock to eliminate contention on architectures with or without coherent caches. The second is a simple extension of Graunke & Thakkar’s lock. It uses swap and eliminates contention in architectures with or without coherent caches. The third solution uses a linked list and preserves the FIFO behavior using swap as the only atomic primitive.

In architectures in which atomic fetch&add is directly supported by the hardware but swap is not supported, Anderson’s lock is recommended for architectures with coherent caches. On the other hand, if an architecture does not support coherent cache, our array based lock is a better choice.

The second is interesting in that it extends Graunke and Thakkar’s lock to architectures that do not support coherent caches. However, we do not recommend it since the linked list based lock offers a better alternative. If an architecture supports swap as well as compare&swap, Mellor-Crummey and Scott’s linked list based lock is a better choice. On the other hand if an architecture does not support compare&swap but supports swap, our linked list based lock is recommended since it preserves the FIFO behavior and is comparable to Mellor-Crummey and Scott’s lock in all other respects.
It should be noted that all the algorithms that eliminate contention from architectures with or without coherent caches rely on the local shared memory. Therefore, the existence of the local shared memory is crucial to these algorithms. Mellor-Crummey and Scott discuss the importance of local shared memory [73]. Our results support their recommendations in that the local shared memory can be successfully exploited to eliminate contention due to synchronization.

The locks presented here are fair and do not take into consideration other characteristics, e.g., priority, that may be desired under some circumstances. The linked list based lock can be adapted for priority locks with minimal effort.
CHAPTER 3

CONTENTION FREE READERS-WRITERS LOCKS

The readers-writers problem was introduced by Courtois, Heymans, and Parnas [16]. There are different variants of this problem. We present solutions for a fair readers-writers lock. A fair readers-writers guarantees FIFO behavior for all processors requesting the lock. Two of our solutions use `fetch&add` while the rest of them use `swap` as the only atomic primitive. They all eliminate contention. One relies on coherent cache to eliminate contention. The rest of them spin in their local memory to busy wait. To the best of our knowledge no contention free fair readers-writers solutions have been previously reported using `fetch&add` or `swap` as the only atomic primitives.

3.1 Introduction

Since the readers only read the data, they can be granted concurrent access to a shared resource. The writers, on the other hand, update the data and require exclusive access. In a single processor multiprocessing environment, granting concurrent access to multiple readers has no major advantage over granting access to a single reader. Since there is only one processor, one reader can execute at a time anyway. In a multiprocessor environment, however, the readers on different processors can execute concurrently if they are all granted concurrent access to the shared data; this increases parallelism.

We present new contention free solutions for a fair readers-writers lock in this chapter. We construct two solutions using atomic `fetch&add` and a number of solutions using atomic `swap`. In all of our solutions we ensure the FIFO behavior and, except for one algorithm, busy wait by spinning on local variables. The one that spins on global variables is a simple algorithm but it relies on coherent cache to eliminate contention.
A fair readers-writers algorithm grants the lock in the order in which the processors request the lock. A writer may acquire the lock when all processors, readers as well as writers, that requested the lock prior to that writer have released the lock. A reader may acquire the lock when all writers that requested the lock prior to that reader have released the lock. This allows multiple readers to concurrently acquire the lock.

The chapter is organized as follows. A brief review of relevant literature is presented in Section 3.2. The next three sections present three approaches to construct readers-writers locks. Two array based locks are presented in Section 3.3. They use atomic `fetch&add` as the only read-modify-write primitive. In the following two sections, we extend our linked list based exclusive lock to readers-writers lock. First, we take an approach which is conceptually similar to the approach taken by Mellor-Crummey and Scott in implementing their readers-writers lock [72][74]. Their lock, however, uses `compare&swap`, atomic increment, atomic decrement, and `swap` read-modify-write primitives. Our lock which is based on a singly linked list uses `swap` as the only atomic primitive and provides the same characteristics. Next, we present three variations of our doubly linked list based lock in Section 3.5. All three guarantee the FIFO behavior, use `swap` as the only atomic primitive, and spin in the local shared memory. They have a conceptual similarity to the readers-writers lock implemented by Krieger, et. al. [50]. However, the lock implemented by Krieger, et. al. requires `compare&swap` for the same capabilities that we have implemented using `swap`. A brief overview of their lock is presented in Section 3.2.2. The chapter is concluded with a summary in Section 3.6.

### 3.2 Related Work

Courtois, Heymans, and Parnas address two variants of the readers-writers problem [16]. In their first variant, the readers have priority over the writers while in the second the writers have priority over the readers. Their solutions are based on semaphores. Likewise, a number of solutions that followed are also based on semaphores or other similar mechanisms such as conditional critical regions or monitors [12][34][43][89]. These solutions require mutual exclusion to manipulate the shared data associated with the solution to the readers-writers problem itself. Also, the readers and writers block themselves when
they need to wait. In these solutions, readers exclude writers which is the usual notion of the readers-writers problem.

Later Lamport [54] took a different approach allowing multiple readers to concurrently read while a writer is writing. This approach, however, requires that a reader verifies it has a consistent copy of the data after it has read the data. This is because a writer is allowed to update the data when a reader might be reading it. As a result, a reader may have to make multiple attempts before it can get a consistent copy of the data. This approach is further followed by other researchers [84]. We are not interested in this kind of approach in our work and, therefore, do not discuss it further.

Busy waiting readers-writers locks may be considered an extension of the busy waiting mutual exclusion locks. However, the fact that readers may release the lock in any order which may be different than the order in which they acquired the lock, makes the problem non-trivial. We present an overview of two solutions from the literature [50], [72], [74].

3.2.1 A Singly Linked List Based Solution

The first solution is due to Mellor-Crummey and Scott [72], [74]. It uses a singly linked list. However, multiple readers releasing the lock in an order different than the one in which they acquired it renders the use of a singly linked list by itself inadequate. Some mechanism is needed to keep a count of the active readers, i.e., the readers that concurrently hold the lock. Mellor-Crummey and Scott use a global variable to explicitly keep the count of the active readers. When an active reader which is not at the head of the list, releases the lock the list becomes discontinuous. Also, if an active reader finds that it is not the last active reader and a writer is waiting behind it, it must put a pointer to the waiting writer in a known global place so that the last reader may unblock the writer. Since the list becomes discontinuous, the last reader can not access the writer though the linked list. So, in addition to keeping the count of active readers, a pointer to the first waiting writer is needed as a part of the global data structure.

Their solution uses swap, atomic increment, atomic decrement, and compare&swap primitives. It offers the FIFO behavior and eliminates contention by using local spinning.
3.2.2 A Doubly Linked List Based Solution

Krieger, et al., use a doubly linked list to implement their solution to the readers-writers problem [50]. Using a doubly linked list, an active reader that is not at the head of the list can unlink itself without fragmenting the list. An explicit count of active readers, therefore, is not needed. Since the list is maintained and does not become fragmented, the last reader will find a waiting writer as its successor and can unblock it. Therefore, a separate pointer to point to the first waiting writer is not needed either. This eliminates the need for atomic increment and decrement primitives and is the primary motivation behind the development of their lock.

The data structure for their lock is graphically shown in Figure 23. To be consistent with our presentation in this document, we use succ and pred in the data structure in place of next and prev which are used in their original algorithm.

![Figure 23. Data Structure for the Lock by Krieger, et. al.](image)

The functions to acquire and release the lock for a writer are conceptually equivalent to the functions to acquire and release the lock in Mellor-Crummey and Scott's mutual exclusion lock [72][74]. A writer additionally sets the pred of its successor to NULL. We do not repeat these functions here.
The function to acquire the lock for readers is shown in Figure 24. The function works in general but fails to satisfy concurrent reading property in a specific timing window; we explain it with an example. Let there be two readers R₀ and R₁ trying to acquire the lock. Let us further assume that R₀ precedes R₁ in linking itself in the doubly linked list and finds its predecessor is a reader. Therefore, it acquires the lock and reaches statement 10 in the algorithm. If R₀ executes statement 10 prior to R₁ executing statement 7 then R₀ does not see R₁ as its successor. Now, if R₀ executes statement 12 before R₁ executes statement 8, R₁ will see its predecessor as an active reader and acquire the lock. However, if R₀ executes statement 12 after R₁ executes statement 8, R₁ does not see its predecessor as the active reader and will busy wait. The fact that a reader continues to busy wait while its predecessor reader acquires the lock violates concurrent reading property. Further a reader unblocks its successor in the release function (not shown here) only if it is the last reader. Therefore, R₁ busy waits until all the active readers have released the lock.

**A Reader Acquiring the Lock**

```c
rd_acquire(struct rw_lock *L struct rw_proc *I)  
1. I->state := READER // indicate this a reader  
2. I->spin := WAIT // prepare to wait  
3. I->succ := I->pred := NULL // initialize succ and pred to NULL  
4. tail := swap(&L->Tail, I) // link at the tail  
5. if (tail != NULL) // there is a predecessor  
6. I->pred := tail // link behind it  
7. tail->succ := I  
8. if (tail->state != ACTIVE_READER) // if predecessor is not an active reader  
9. repeat while (I->spin = WAIT) // reader, busy wait  
    // At this point, the reader has acquired the lock  
10. if (I->succ != NULL && I->succ->state == READER) // if the successor  
11. I->succ->spin := NULL // is a reader, pass it the lock  
12.I->state := ACTIVE_READER
```

*Figure 24. A Reader Acquiring the Lock in the Lock by Krieger, et. al.*
One may try to correct the situation by changing state to ACTIVE_RREADER prior to checking the successor. However, that creates a different problem. The successor may already have acquired and released the lock before a predecessor unblocks it.

This problem already exists in their function to release the lock. A reader releasing the lock attempts to acquire the exclusive lock using predecessor's lock element. However, the predecessor may have already unlinked itself and de-allocated or reused the memory. Since the reader atomically writes to predecessor's lock element to acquire the lock, it may over-write the element that it thinks is the lock element of the predecessor but in reality may have been reused for some other purpose.

It is not safe to modify or examine a processor's data structure if the processor is not waiting for the update or the examination. A processor that is not waiting may de-allocate the memory for the data structure or use it for some other purpose. If the memory is reused, the update may over-write some new data causing irreparable damage to the system.

3.3 Array Based Readers-Writers Locks Using Fetch&Add

We extend the contention free array based mutual exclusion lock to fair readers-writers lock. The extension is fairly simple. A reader may acquire the lock if its immediate predecessor is a reader that has acquired the lock. This implies that the knowledge about a reader that has acquired the lock needs to be conveyed to its successor. We use a mechanism similar to the one used in the mutual exclusion lock to convey that the lock is free. A reader that has acquired the lock updates the next element in the array to signal the successor that it can acquire the lock if it is a reader. To be specific we use a token READ to indicate this condition. The successor will recognize that the lock is available for a reader but not for a writer. It will acquire the lock if it is a reader.

The readers can release the lock concurrently and in an order different than the order in which they acquire it. This implies the active readers must act and behave as a group letting any reader release the lock. To this end, we require a reader removes its specific information from its corresponding element in the array before it joins the group of active readers. We also use an additional variable Head that indexes the first element in the array that belongs to the group of active readers. The value of this element is LOCK, which
indicates it is the first element of a group of consecutive elements belonging to the active readers. Any reader that wants to release the lock, adds one to Head and then checks if the value of the element indexed by the pre-addition value of Head is LOCK. If it is, the reader is at the head of the queue. It makes the indexed element NULL. If the indexed element is not LOCK, it indicates that a reader that started the release prior to this reader has not yet completed its release. This reader waits for the previous reader to finish, at which point it can change its corresponding element to NULL. It then updates the next element to LOCK passing on the token LOCK. It does not need to know whether the waiting processor is a reader trying to acquire or release the lock or a writer waiting to acquire the lock.

We present two variants of this lock. The first is a simpler version but relies on coherent cache to eliminate contention. The second is somewhat more complex but uses local only spinning to eliminate contention.

3.3.1 Array Based Readers-Writers Lock (RW.AR-1)

It may be viewed as an extension of Anderson's mutual exclusion lock. It uses fetch&add as the only atomic primitive and ensures the FIFO behavior.

3.3.1.1 Data Structure

The data structure is similar to the data structure for the mutual exclusion lock except that the size of the array needs to be, at least, one greater than the number of the processors in the system, i.e., \( N_a > N_p \); \( N_a \) is the size of the array and \( N_p \) is the number of processors in the system. The reason for this is explained in Section 3.3.3. Also, another global variable is needed in addition to variable Tail. Further, like in the mutual exclusion lock, we require \( N_a=2^k \) for the same reasons as stated in Section 2.4.1 in CHAPTER 2. The additional variable which we call Head, is needed by the readers to release the lock. The array elements may assume three different values. An additional value is needed to signal the next processor that it can acquire the lock if it is a reader.

Like Tail, Head is also an unsigned integer that varies from 0 to \( 2^n-1 \), where 'n' is the native size of the machine. Readers use Head to release the lock in a manner similar to all processors use Tail to acquire the lock. When no processor is in the process of releasing
the lock, Head points to the first element in the array at which a processor will release the lock. Conceptually one may think of it as the head of the queue from where a processor will remove itself.

The data structure is shown in Figure 25. An instance of the data structure with three readers concurrently holding the lock followed by a writer, which itself is followed by a reader, is also shown in Figure 25. Since the writer requires exclusive access, it can get the lock only after all the readers that requested the lock prior to it have acquired and released the lock. Therefore, the writer waits spinning on the cached copy of the array element until it is updated to LOCK. The reader that follows the writer must also wait since it can acquire the lock only after all the writers that requested the lock prior to it have acquired and released the lock.

```
struct rw_lock {
    unsigned int Head;
    unsigned int Tail;
    unsigned int *Que[N];
} rwLock;
```

![Diagram of rw_lock structure](image)

**Figure 25. Data Structure for the Array Based Lock (RW.AR-1)**

### 3.3.1.2 Algorithm RW.AR-1

The lock is initialized to free by setting Tail and Head both to index the same element of the array and setting the indexed element itself to LOCK. The rest of the elements are initialized to NULL. This sets the lock to free. The tokens LOCK and READ must be unique.

Since a writer in a readers-writers lock requires mutual exclusion, the algorithms to acquire and release the readers-writers lock for a writer are conceptually the same as the
algorithms to acquire the mutual exclusion lock in Anderson's original array based lock. The minor difference is due to the use of variable Head.

**Initialization**

```c
wr_initialize(struct lock *L)
L->Head := i; 0≤i<N_a  // The choice of the element is arbitrary
L->Tail := L->Head;  // Tail and Head index the same element
L->Que[n]:= LOCK; n=i
:= NULL; 0≤n<N_a; n≠i  // NULL is defined to be 0
```

**Figure 26. Initialization of the Array Based Lock (RW.AR-1)**

A reader can acquire the lock when the lock is either free or the processor ahead of it is also a reader and has acquired the lock. If the processor ahead of this reader is a reader that has acquired the lock, it has placed a unique token READ in the next element which is the indexed element of this reader. This reader acquires the lock as soon as the value of its indexed element is either LOCK or READ, i.e., non NULL. Once it acquires the lock, it updates the next array element to READ. This signals the successor that it can acquire the lock if it is a reader.

To release the lock a reader atomically adds one to Head and checks the value of the element indexed by the pre-addition value of Head. If the value of the indexed element is not LOCK, the reader waits until it becomes LOCK. This allows a reader, that started the release before this reader, to complete the release. When the value of its indexed element is LOCK, the reader can update its indexed element to NULL preparing for the next time. Next, it updates the next element in the array to LOCK.

This is a simple algorithm that could eliminate contention in architectures with coherent caches or with distributed memory but not in architectures without coherent caches. It requires each array element to be in a different cache line or in a different memory module for it to eliminate contention due to busy waiting.
A Writer Acquiring the Lock

wr_acquire(struct lock *L)
    // X & (Na-1) is equal to X mod Na for Na=2^k, my_index is a local variable
    1. my_index := fetch&add(&L->Tail, 1) & (Na-1)
    2. repeat while (L->Que[my_index] ≠ LOCK)

A Writer Releasing the Lock

wr_release(struct lock *L)
    1. L->Que[L->Head & (Na-1)] := NULL // initialize for the next time
    2. L->Head := L->Head + 1 // advance Head
    3. L->Que[L->Head & (Na-1)] := LOCK // advance LOCK

A Reader Acquiring the Lock

rd_acquire(struct lock *L)
    // X & (Na-1) is equal to X mod Na for Na=2^k, my_index is a local variable
    1. my_index := fetch&add(&L->Tail, 1) & (Na-1)
    2. repeat while (L->Que[my_index] = NULL)
    3. L->Que[(my_index+1) & (Na-1)] := READ

A Reader Releasing the Lock

rd_release(struct lock *L)
    // my_index is a local variable
    1. my_index := fetch&add(L->Head, 1) & (Na-1)
    2. repeat while (L->Que[my_index] ≠ LOCK)
    3. L->Que[my_index] := NULL // initialize for next time
    4. L->Que[(my_index+1) & (Na-1)] := LOCK // pass the lock

Figure 27. Code for the Array Based Readers-Writers Lock (RW.AR-1)

3.3.2 Array Based Readers-Writers Lock (RW.AR-2)

The previous algorithm may be modified to eliminate contention in architectures that do not support coherent caches. We present the modified data structure and the algorithm in this section.
3.3.2.1 Data Structure

Except for the fact that each processor needs a spin variable in its local shared memory and the array must be an array of pointers, the data structure is the same as for the previous algorithm. In addition to the three values of NULL, LOCK, and a pointer described in Section 2.4.1.1 in CHAPTER 2 for the mutual exclusion lock, each element of the array may have a fourth value for the readers-writers lock as follows.

4. A token READ interpreted by a processor requesting the lock that the lock is available for a read but not available for a write. This token is written at index \(i+1\) by a reader at index \(i\) after acquiring the lock. The token at index \(i+1\) is later updated to LOCK by a reader releasing the lock at index \(i\). During this update a reader releasing the lock interprets the value READ to mean that there is no processor waiting to acquire or release the lock at index \(i+1\).

The tokens LOCK and READ must be unique and distinguishable from valid pointers to spin variables of various processors. One may choose the address of Tail as LOCK and the address of Head as READ. Addresses of the array elements are other possible values that could be used for LOCK and READ.

3.3.2.2 Algorithm RW.AR-2

The initialization is exactly the same as in the previous lock and not repeated here.

The algorithms to acquire and release the readers-writers lock for a writer are conceptually the same as to acquire the mutual exclusion lock in the array based lock; see Section 2.4.1. The minor differences are due to variable Head. The variable Head always indexes the element that a processor releasing the lock can use to remove itself from the queue.

A reader can acquire the lock when the lock is either free or the processor ahead of it is also a reader and has acquired the lock. When a reader acquires the lock, it places a unique token READ in the element immediately following the element that it used to get the lock. This unique token must differ from NULL and the token LOCK.

A reader gets its so called request number by atomically adding one to variable Tail and reading back its pre-addition value. Using the pre-addition value, it finds the index of the
corresponding element in the array. If the array element holds the token \texttt{LOCK} or the token \texttt{READ}, i.e., it is not \texttt{NULL}, the reader acquires the lock. A processor requesting the lock encounters only one of these three values in the element.

**A Writer Acquiring the Lock**

\begin{verbatim}
wr_acquire(struct lock *L)
    // X (N_a-1) is equal to X mod N_a for N_a=2^k
    // my_index is a local variable
1. spin := WAIT       // WAIT\#LOCK\#READ: may be NULL
2. my_index := fetch&add(L->Tail,1) & (N_a-1)
3. if (L->Que[my_index] != LOCK)
4.   if (fetch&add(L->Que[my_index],&spin) # LOCK)
5.       repeat while (spin # LOCK)
\end{verbatim}

**A Writer Releasing the Lock**

\begin{verbatim}
wr_release(struct lock *L)
    // next_index and succ (short for successor) are local variables.
1. L->Que[L->Head & (N_a-1)] := NULL // initialize for the next time
2. L->Head := L->Head + 1          // advance Head
3. next_index := L->Head & (N_a-1)  // circularly next index
4. if ((succ:=L->Que[next_index]) == NULL)
5.   succ := fetch&add(L->Que[next_index],LOCK)
6. if (succ # NULL)                  // there is a successor.
7.   *succ := LOCK                   // pass the lock
\end{verbatim}

**Figure 28. Code for A Writer in the Array Based Lock (RWAR-2)**

If the array element is \texttt{NULL}, the reader must wait. It atomically adds the address of its \texttt{spin} variable to the element, reading back its pre-addition value. If the prior value is still \texttt{NULL}, the reader waits spinning in its local shared memory waiting for the lock. If the pre-addition value is not \texttt{NULL}, it must be either \texttt{LOCK} or \texttt{READ}, in which case the reader can proceed to acquire the lock.

77
**A Reader Acquiring the Lock**

```c
rd_acquire(struct lock *L)
    // X & (N_a-1) is equal to X mod N_a for N_a=2^k
    // my_index, token, and succ are local variables
1. spin := NULL // NULL is defined to be 0
2. my_index := fetch&add(&L->Tail, 1) & (N_a-1)
3. if (L->Que[my_index] == NULL)
4.    if ((token:=fetch&add(&L->Que[my_index], &spin)) == NULL)
5.        repeat while ((token:=spin) == NULL)
6.            if (fetch&add(&L->Que[my_index], -(&spin)) != (token+(&spin)))
7.                repeat while (spin != LOCK)
8.                next_index := (my_index+1) & (N_a-1)
9.                succ := fetch&add(&L->Que[next_index], READ)
10. if (succ != NULL)
11.    *succ := READ
```

**A Reader Releasing the Lock**

```c
rd_release(struct lock *L)
    // next_index and succ (short for successor) are local variables.
1. my_index := fetch&add(L->Head, 1) & (N_a-1)
2. if (L->Que[my_index] != LOCK)
3.    spin := NULL // prepare to wait
4.    if (fetch&add(L->Que[my_index], &spin) != LOCK)
5.        repeat while (spin != LOCK)
6.    L->Que[my_index] := NULL // initialize for the next time
7.    next_index := (my_index+1) & (N_a-1) // circularly next index
8.    succ := L->Que[next_index]
9. if (succ == READ) // no waiting successor
10.    succ := fetch&add(L->Que[next_index], (LOCK-READ))
11. if (succ != READ) // there is a successor,
12.    *(succ-READ) := LOCK // pass the lock
```

Figure 29. Code for A Reader in the Array Based Lock (RW.AR-2)
Since the reader added the address of its spin variable, the element holds the sum of the
token and the address of its spin variable at the moment the reader acquires the lock.
Before the reader proceeds, it atomically subtracts the address of its spin variable from
the value in the element. This removes its specific information from the element it used
to acquire the lock. The expected pre-subtraction value is the sum of the address of its
spin variable and the variable token at the time the reader acquired the lock. If the pre-
subtraction value differs, it is an indication that the last of the active readers that
acquired the lock prior to this reader is in the process of releasing the lock. The reader
releasing the lock does not differentiate between a writer waiting for the lock, another
reader releasing the lock, or a reader that just acquired the lock but did not have a chance
to subtract the address of its spin variable from the element. Therefore, the reader
acquiring the lock must wait, letting the reader releasing the lock to pass it the token
LOCK, at which point it can continue.

Next, the reader atomically adds token READ to the next element. If the pre-addition value
of the next element is not NULL, it indicates a processor is already waiting for the lock.
The processor passes the token READ to the waiting processor through its spin variable.
If the waiting processor is a reader it also acquires the lock.

More than one reader may acquire the lock at the same time and perform reads concurrently.
Further, the readers may release the lock in a different order than they acquired it. To facilitate this, all readers that hold the lock simultaneously must act as a group letting any reader release the lock irrespective of the order in which it acquired the lock. Variable Head provides the index of the element that can be used to release the lock. When no reader is in the process of releasing the lock, Head points to the first element in a group of contiguous elements, one corresponding to each reader that has acquired the lock. The first element holds the token LOCK; the rest of the elements each hold the token READ as shown in Figure 25. Further, if there is no processor waiting for the lock, the element indexed by Tail also holds the token READ as shown in Figure 30.

To release the lock, a reader first atomically adds one to variable Head reading back its
pre-addition value. The pre-addition value gives the reader its sequence number or the request number that it can use to index the array. If all other readers that started
releasing the lock have completed the release, the indexed element holds the token LOCK. If the indexed element does not hold the token LOCK, another reader is in the process of releasing the lock. To effect an orderly release of the lock, this reader waits for the other reader to complete the release. It prepares its spin variable to busy wait and adds its address to the indexed element, reading back the pre-addition value. If the pre-addition value is still not equal to LOCK, the reader waits for the token LOCK, spinning in its local shared memory. When the token LOCK arrives, the reader is at the head of the queue. Now it is at the point where it would have been if the indexed element contained LOCK. It changes the indexed element to NULL.

```c
struct lock {
    unsigned int Head;
    unsigned int Tail;
    unsigned int *Que[Ns];
} Lock;
unsigned int spin;
```

**Figure 30. An Instance of Data Structure Showing Readers Only**

The reader had changed its next element to READ after it acquired the lock. If no processor has requested the lock since then, or a reader has completed acquiring the lock, the next element must hold token READ. On the other hand, if a writer has requested the lock or a reader is either in the process of acquiring or releasing the lock, the value in the next element is the sum of the token READ and the address of the spin variable of the processor waiting at the next element. The reader assumes the next element holds READ and atomically adds the difference of LOCK and READ, i.e., LOCK-READ, to the next element reading back the pre-addition value. If the pre-addition value is READ, it is changed to LOCK and no further action is needed on the part of this reader. If the pre-addition value is not READ.
it must be the sum of \texttt{READ} and the address of the \texttt{spin} variable of the processor waiting for the token \texttt{LOCK}. The reader recovers the address of the \texttt{spin} variable of the waiting processor by subtracting the value \texttt{READ} from the pre-addition value and passes the token \texttt{LOCK} to the waiting processor.

3.3.3 Comments

The size of the array for a readers-writers lock in both of these algorithms is greater than the number of processors in the system, i.e., \( N_a > N_p \). It is possible that all processors in the system may acquire the lock for reading. Each reader adds a token \texttt{READ} to the next element after it acquires the lock in both the algorithms. Further, if the pre-addition value of the next element is not \texttt{NULL} in case of the second algorithm, it assumes the value points to the \texttt{spin} variable of the next waiting processor. It passes the token \texttt{READ} to the waiting processor through its \texttt{spin} variable.

If \( N_a=N_p \) and \texttt{Head} corresponds to index '1' in the array, then the value in the element '1' is \texttt{LOCK}. If we let all processors acquire the lock for reading without any processor yet releasing the lock, the last processor will be at index '1-1'. After acquiring the lock, it will add the token \texttt{READ} to the next element at index '1' reading back its pre-addition value \texttt{LOCK} in case of the second algorithm. The value at index '1' is changed from \texttt{LOCK} to \texttt{LOCK+READ}. A processor that wants to release the lock expects the token \texttt{LOCK} at index '1'. Since the value is changed, it waits forever the token \texttt{LOCK} to arrive. As a result the processor that tried to release the lock when \texttt{Head} corresponded to index 1 is blocked forever and so is every other reader in the system. Further, in case of the second algorithm, the reader will assume \texttt{LOCK} to be the address of the \texttt{spin} variable of a waiting processor and try to pass the lock on to it incorrectly over-writing some memory.

While solutions for the case where \( N_a=N_p \) can be formulated, the easiest solution is to increase the size of \( N_a \) so that \( N_a>N_p \). This results in an array that is, at least, one greater than the number of processors and eliminates the problem.

There is considerable manipulation of pointers in these algorithms that involves addition and subtraction. All additions and subtractions involving pointers must be carried out in the unsigned integer domain to ensure proper results. Higher level languages generally
do not allow such operations on the pointers. Even when they do, the results are not the same as they would be if these pointers were unsigned integer quantities.

When the tokens READ or LOCK are passed in the spin variable, they should be passed as unsigned integers to be consistent with the type declaration of the spin variable.

3.4 Singly Linked List Based Readers-Writers Lock Using Swap

In this algorithm writers as well readers use a singly linked list to acquire the lock. After acquiring the lock a writer stays at the head of the singly linked list during the time it is holding the lock. A reader, on the other hand, adds one to the count of the active readers and takes itself off of the linked list. To release the lock a reader subtracts one from the count of the active readers and if there are no more active readers and a writer is waiting for the lock, it passes the lock on to the waiting writer. If no writer is waiting, the lock is free without further action.

We note that multiple processors cannot concurrently add to or subtract from a single variable using swap as the only read-modify-write primitive. Therefore, we need to find a way to maintain a count of the active readers for this approach.

We can maintain a count of the active readers by either using one variable or two variables. If we use one variable, each reader that acquires the lock must add one to the variable and each reader that releases the lock must subtract one from the same variable. The access to the variable must be exclusive allowing one reader to modify it at a time. This implies we associate a mutual exclusion lock with the variable used to keep the count. Each reader acquiring or releasing the readers-writers lock will lock the count to access and update it.

An alternative is to use two unsigned integer variables to keep a count of the active readers. A reader adds one to one of the variables when it acquires the lock and adds one to the other variable when it releases the lock. Using unsigned arithmetic, the difference of the two always gives the number of active readers correctly. Since multiple readers may concurrently release the lock, they must have exclusive access to the second variable to update it. The first variable, however, can be accessed by readers acquiring the lock
without requiring exclusive access; as we will see shortly they already have exclusive
access to it. The advantage of this approach is that readers acquiring the lock do not have
to contend with readers releasing the lock to atomically update the count. As a result, the
readers can acquire the lock concurrently with other readers releasing the lock. We use
the second approach in our algorithm.

3.4.1 Data Structure

This algorithm uses two singly linked lists. One of these lists is used to acquire the lock
by all processors, the other is used to release the lock by the active readers. There is a
mutual exclusion lock consisting of Tail and sHead, associated with each of these linked
lists. The structure aLock is associated with the list used to acquire the lock and rLock
is associated with the list used to release the lock. In addition to these data structures,
we use two unsigned integer elements Acqs and Reis to keep a count of the active read­
ers. The element wHead is a pointer used by a writer when it is at the head of the singly
linked list but it has to further wait since there are one or more active readers holding the
lock. The data structure is shown in Figure 31. A graphical view of the data structure is
presented in Figure 32.

```
struct rw_lock {
    struct lock aLock;       // mutual exclusion lock to acquire the lock
    struct lock rLock;       // mutual exclusion lock to release the lock
    struct proc *wHead;      // pointer for a waiting writer
    unsigned int Acqs;       // number of readers that acquired the lock
    unsigned int Reis;       // number of readers that released the lock
} rwLock;
```

Figure 31. Data Structure for the Singly Linked List Based Lock

In addition to the above data structure to be defined in the global shared memory, each
processor requires a local data structure for the mutual exclusion lock. This data struc­
ture is exactly the same as in the mutual exclusion lock described in Section 2.4.3.1 in
CHAPTER 2, and serves the same purpose.
To acquire the lock all processors first acquire $aLock$. When a processor has acquired $aLock$, it has reached the head of the singly linked list used to acquire the lock. This implies any writer that requested the lock prior to this processor has released the lock. Therefore, a reader can acquire the readers-writers lock soon as it acquires $aLock$. A writer, however, needs to further check the number of active readers and can acquire the lock only if no reader is holding the lock. After acquiring $aLock$ a reader has exclusive access to $acqs$ until it releases $aLock$. It adds one to $acqs$ and releases $aLock$ taking itself off of the list used to acquire the readers-writers lock. After which an active reader is on neither list until it attempts to release the lock by acquiring $rLock$. A writer, on the other hand, stays at the head of the linked list used to acquire the lock during the time it is holding the lock.

To release the readers-writers lock, a writer simply releases $aLock$. A reader on the other hand first, acquires $rLock$, adds one to $rels$, checks if it is the last active reader, passes
on the lock to a waiting writer if it is the last active reader and a writer is waiting, and
then releases the rLock. The code to initialize, acquire and release the lock follows.

**Initialization**

```c
rw_initialize(struct rw_lock *L)
initialize(&L->aLock) // set aLock to free
initialize(&L->rLock) // set rLock to free
L->wHead := NULL // NULL is defined to be 0
L->Acqs := 0
L->Rels := 0
```

**Figure 33. Code to Initialize the Singly Linked List Based Lock (RW.SL-1)**

**A Reader Acquiring the Lock**

```c
rd_acquire(struct rw_lock *L, struct proc *P)
1. acquire(&L->aLock, I) // acquire aLock
2. L->Acqs := L->Acqs+1
3. release(&L->aLock, I) // release aLock
```

**A Writer Acquiring the Lock**

```c
wr_acquire(struct rw_lock *L, struct proc *P)
1. acquire(&L->aLock, I) // acquire aLock
2. I->spin := WAIT // prepare to wait
3. L->wHead := I // point wHead to yourself
4. if ((L->Acqs != L->Rels) || // if count of active readers is not
     (swap(&L->wHead, NULL) == NULL)) // zero or wHead is already consumed
5.       repeat while(I->spin == WAIT) // wait until the lock is granted
```

**Figure 34. Code to Acquire the Singly Linked List Based Lock (RW.SL-1)**

A reader acquiring the readers-writers lock is quite straightforward. It acquires aLock,
adds one to Acqs, and releases aLock. A writer acquiring the readers-writers lock
acquires aLock, prepares to spin, points wHead to itself and checks whether the number of active readers is zero. If the number of active readers is zero it can acquire the readers-writers lock. However, it takes extra steps to ensure that it acquires the lock by itself iff a reader is not going to pass it the lock. A reader releasing the lock may see the writer on its line 4 in which case the writer must wait for the reader since the reader updates writer's spin variable. The writer and the last active reader must \textit{atomically see and consume} the pointer in wHead before checking whether to wait and pass on the lock respectively. As a result if a releasing reader sees it, the writer will find wHead NULL and wait for the reader to give it the lock. If the writer sees it, it will itself acquire the lock since the reader will not see it. If the count of active readers is not zero the writer waits spinning on variable spin in its local shared memory.

\textbf{A Reader Releasing the Lock}

\begin{verbatim}
rd_release(struct rw_lock *L, struct proc *I)
1. acquire(&L->rLock,I) // acquire rLock
2. L->Rels := L->Rels + 1
3. if ((L->Acqs == L->Rels) && // if count of active readers is 0
4.     (writer := swap (&L->wHead, NULL) ) // consume wHead
5.   if (L->Acqs == L->Rels) // if count of active readers is still 0
6.     writer->spin := FREE // grant the lock to the writer
7.   else // else
8.     L->wHead := writer // link the writer back in wHead
9. release(&L->rLock,I) // and release rLock
\end{verbatim}

\textbf{A Writer Releasing the Lock}

\begin{verbatim}
wr_release(struct rw_lock *L, struct proc *I)
1. release(&L->aLock, I) // release aLock; this releases the
   // readers-writers lock
\end{verbatim}

\textbf{Figure 35. Code to Release the Singly Linked List Based Lock (RW.SL-1)}
To release the readers-writers lock a writer simply releases rLock. A reader on the other hand, acquires rLock, updates Reis, and checks to see if it is the last reader. If it is the last reader, it consumes the contents of wHead. If no writer was waiting, wHead was already NULL and line 4 has no effect. The readers-writers lock is now free. If a writer was waiting, the reader needs to check again if the count of the active readers is still zero before it passes the lock on to the waiting writer. This check is necessary since a reader ahead of the waiting writer may have acquired the lock **after** this reader executes line 3 but **before** it executes line 4. Since Acqs is updated before a reader passes the lock to its successor, the number of active readers is guaranteed to be nonzero on line 5 if a reader did acquire the lock ahead of the waiting writer. If the number of active readers is still zero, no reader preceded the waiting writer and the lock is passed on to the waiting writer. Otherwise, the reader puts the pointer to the waiting writer back in wHead and releases rLock. If this reader was not the last active reader as determined on line 3 of the algorithm it simply releases rLock.

### 3.5 Doubly Linked List Based Readers-Writers Locks Using Swap

There are three possible approaches to implementing the readers-writers lock using a doubly linked list. We will present the algorithms for all three. A brief description of each approach is provided below. One may view these approaches as three levels of the same concept allowing more concurrency at the expense of additional complexity.

1. **The doubly linked list itself may be considered a resource and accessed using a mutual exclusion lock.** All processors acquiring or releasing the readers-writers lock use exclusive access to the doubly linked list to manipulate it. However, a processor that must wait to acquire the readers-writers lock first releases the mutual exclusion lock to the doubly linked list and then busy waits spinning on a variable in its local shared memory.

2. **Since the processors acquiring the lock link at the tail of the doubly linked list and it is possible to do so without exclusive access, the processors need not lock the doubly linked list for acquiring the lock.** Since only one writer acquires the readers-writers lock at a time, it can release the lock without exclusive access to the doubly linked list. This leaves the active readers as the only entity that need
to manage the doubly linked list using exclusive access. This approach which is more complex than the first approach is based on this observation. It lets the readers acquire the lock concurrently while other readers are releasing the lock.

3. A processor does not need exclusive access to the entire linked list when it wants to unlink itself from within a doubly linked list. If it can acquire exclusive access to its predecessor and to itself, it can unlink itself. This is the minimum exclusive access that it requires. In this approach, which is more complex compared to the previous two approaches, no processor requires exclusive access to the doubly linked list. This approach not only lets the readers acquire the lock concurrently with other readers releasing the lock, but also lets the readers release the lock concurrently as long as such readers are not adjacent to each other.

A writer can acquire the lock only when the lock is free. A reader, on the other hand, can acquire the lock when either the lock is free or the processor ahead of the reader is an active reader. This implies that in addition to knowing whether the lock is free, a reader also needs to know whether its predecessor is an active reader. Each processor needs to convey this knowledge to its successor. In a "Singly Linked List Based Readers-Writers Lock Using Swap" on page 82, when a processor reached the head of the linked list used to acquire the lock, it implicitly knew that either the lock is free or the processor ahead of it is an active reader. When using a doubly linked list, a reader explicitly needs to know whether its predecessor is an active reader when the lock is not free.

When a reader acquires the lock it needs to pass the lock on to its successor if the successor is a reader. For this purpose either a waiting processor needs to identify itself to its predecessor as to whether it is a reader or a writer, or each processor waiting to acquire the lock needs to have the intelligence to determine whether it can acquire the lock. This, to some degree, is similar to the situation where a writer has to further wait after reaching the head of the list in the readers-writers lock using a singly linked list. We put the onus on the waiting processor to determine whether it can acquire the lock.

3.5.1 All Processors Lock the Doubly Linked List (RW.DL-1)

In this approach the linked list itself is considered a shared resource and accessed using a mutual exclusion lock. A processor, reader or writer, first acquires exclusive access to
the doubly linked list, links itself at the tail, updates its spin variable, gives up the mutual exclusion lock, and then spins on its local variable if it needs to wait. Likewise, a processor that wants to release a readers-writers lock, first acquires the exclusive lock to the doubly linked list, releases the readers-writers lock, and then releases the exclusive lock. It should be noted that a processor never busy waits for the readers-writers lock while it is holding the exclusive lock to the doubly linked list. The advantage of this approach is that it is simple. The disadvantage is that all processors that acquire or release the readers-writers lock have to contend and acquire the exclusive lock to the doubly linked list.

3.5.1.1 Data Structures

In addition to the data structures in the global and the local shared memory needed for the exclusive lock to the doubly linked list, one variable in the global shared memory and two variables in the local shared memory of each processor are needed to implement the doubly linked list itself. The readers-writers lock, therefore, consists of two elements dLock which is an instance of the structure for the mutual exclusion lock and Tail, a pointer to the rw_proc structure, used to point to the doubly linked list. The data structure declarations are shown in Figure 36. Each processor needs one proc structure per processor for the mutual exclusion lock and another two elements, succ and pred for the doubly linked list; the structure declaration for proc is shown in Figure 20. The element pred is also used to busy wait for the readers-writers lock in addition to providing a back link to its predecessor.

Two readers P_i and P_j that own the lock are shown at the head of the doubly linked list in Figure 37. These are followed by a writer, P_k, that is linked in the doubly linked list but waiting for the readers-writers lock. Another reader, P_h, is linked behind the writer and must wait for the lock. Then we show another processor that is at the head of the singly linked list and, therefore, has just acquired the mutual exclusion lock to the doubly linked list itself. This processor, P_m, may be a reader or a writer, will link behind P_i, release the mutual exclusion lock, and then wait for the readers-writers lock spinning on its element pred in its local shared memory. Next, we show P_j, one of the active readers, waiting for the mutual exclusion lock. As soon as P_m releases dLock, P_j will acquire it, unlink itself from the doubly linked list, and release the mutual exclusion lock. At some point later, P_i
will also release the readers-writers lock updating the pred element of \( P_k \). This will pass the readers-writers lock on to \( P_k \).

```c
struct rw_lock {         // data structure in the global shared memory
    struct lock dLock;    // exclusive lock to access the doubly linked list
    struct rw_proc *Tail;  // Tail of the doubly linked list
} rwLock;

struct rw_proc {         // data structure in the local shared memory
    struct proc Proc;     // proc structure for the exclusive lock
    struct rw_proc *succ; // pointer to a successor in the doubly linked list
    struct rw_proc *pred; // pointer to a predecessor in the doubly linked list
} rwProc;
```

**Figure 36. Data Structures for RW.DL-1**

3.5.1.2 Algorithm RW.DL-1

We make use of the succ element of the local structure \( \text{rw}_\text{proc} \) of each processor to convey whether a processor is an active reader. The succ element of the structure is not used until a successor processor needs to link behind it. Therefore, a processor can use it to convey to its successor whether it is an active reader. The successor can link behind its predecessor and, at the same time, determine whether its predecessor is an active reader by swapping the contents of the succ element of its predecessor with a pointer to itself and later examining the contents returned by the swap. It is important to use atomic swap since the predecessor may update its succ element asynchronously after it has released the mutual exclusion lock; the predecessor must also use swap to update its succ element. A processor initializes its succ element to indicate it is not an active reader. A reader, after acquiring the lock, will change succ to indicate it is an active reader provided it does not have a successor to which it can pass on the lock. If it has a successor, it will pass the lock on to the successor to which its succ element points.

When an active reader passes the lock on to its successor it does not inquire whether the successor is a reader. We put the onus on the waiting readers and writers to ensure that readers acquire the lock but writers do not acquire the lock in this scenario. To accomplish
this, a reader initializes its \texttt{pred} element for waiting. While linking itself at the end of the doubly linked list, if it finds the lock is free or its predecessor is an active reader, it updates \texttt{pred} to indicate the lock is free or to point to the predecessor respectively. On the other hand if neither the lock is free nor the predecessor is an active reader, it does not update its \texttt{pred} element. After releasing the mutual exclusion lock, the reader waits until \texttt{pred} has a value different than the initialized value. If the reader itself updated \texttt{pred}, it will immediately acquire the readers-writers lock. Otherwise it will wait until its predecessor passes the lock to it by pointing its \texttt{pred} to either \texttt{NULL} or to its predecessor.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure37.png}
\caption{An Instance of Data Structure for RW.DL-1}
\end{figure}

A writer sets its \texttt{pred} element to its predecessor, pointed to by \texttt{Tail} of the doubly linked list. If there is a predecessor, the writer links itself behind the predecessor. If the \texttt{Tail} indicates the lock is free, the \texttt{pred} is set to indicate the lock is free. Then the writer releases the mutual exclusion lock and waits until its \texttt{pred} element indicates the lock is free.
We set the mutual exclusion lock associated with the doubly linked list to free and Tail of the doubly linked list to NULL to initialize the readers-writers lock to free. When the tail of the doubly linked list is free, it indicates the doubly linked list is empty.

**Initialization**

```c
rw_initialize(struct rw_lock *L)
initialize(&L->dLock)  // set dLock to free
L->Tail := NULL      // NULL is defined to be 0
```

**Figure 38. Code to Initialize the Lock in RW.DL-1**

To acquire the readers-writers lock, a reader initializes its succ element to indicate it is not an active reader and sets its pred element for busy waiting before acquiring the exclusive access to the doubly linked list. Next, it acquires exclusive access to the doubly linked list and checks if the lock is free. If the lock is not free it links behind the predecessor using swap and checks the return value from the swap. If either the lock is free or the predecessor is an active reader as indicated by the return value from the swap, the reader updates its pred to NULL or to point to the predecessor accordingly. Next, it updates Tail to point to itself and releases the exclusive lock. Next, it waits until its pred element has a value different than the initialized value. When it updates pred, it will immediately acquire the lock. Otherwise, it will wait until its predecessor updates its pred.

When a reader acquires the lock, it needs to pass the lock on if a reader is waiting behind it or otherwise indicate it is an active reader. To accomplish this, the reader swaps NULL with the contents of its succ element. If the return value from the swap indicates a waiting successor, the reader updates successor’s pred. If the successor is a reader, it acquires the lock. If the return value from the swap equals the initialization value, there is no successor. The element succ is updated to indicate the reader is an active reader.

A writer initializes its succ element to indicate it is not an active reader. Next, it acquires exclusive access to the doubly linked list and sets its pred element equal to the value in Tail setting it to indicate the lock is free if the lock is free or to its predecessor otherwise.
Next, it points Tail to itself, releases the mutual exclusion lock, and waits until its pred indicates the lock is free.

**A Reader Acquiring the Lock**

```c
rd_acquire(struct rw_lock *L struct rw_proc *I)
   // succ is a local variable.
1. I->succ := I                // indicates not an active reader
2. I->pred := I                // prepare to wait
3. acquire(&L->dLock,&I->Proc) // exclusive lock to doubly linked list
4. if ((L->Tail == NULL) || (swap(&L->Tail->succ, I) == NULL))
5.   I->pred := L->Tail        // point I->pred to the predecessor
6.   L->Tail := I              // point Tail to yourself and
7.   release(&L->dLock,&I->Proc) // release lock to doubly linked list
8.   repeat while (I->pred == I) // busy wait if necessary
9.   succ := swap(&I->succ,NULL) // update yourself to an active reader
10. if (succ != I)              // a successor is waiting
11.   I->succ := succ          // link the successor behind yourself
12.   succ->pred := I          // and pass the successor behind yourself
```

**A Writer Acquiring the Lock**

```c
wr_acquire(struct rw_lock *L struct rw_proc *I)
1. I->succ := I                // indicates not an active reader
2. acquire(&L->dLock,&I->Proc) // exclusive lock to doubly linked list
3. I->pred := L->Tail          // if the lock is not free
4. if (L->Tail != NULL)        // if the lock is not free
5.   L->Tail->succ := I        // link behind the predecessor
6.   L->Tail := I              // point Tail to yourself, and
7.   release(&L->dLock,&I->Proc) // release lock to doubly linked list
8.   repeat while (I->pred != NULL) // spin if lock is not free
```

**Figure 39. Algorithms to Acquire the Lock in RW.DL-1**

To release the lock, a reader acquires exclusive access to the doubly linked list. If it has a predecessor, predecessor's succ element gets the value of its succ element. As a result,
the predecessor links to its successor if it has a successor. Otherwise, the predecessor’s succ element gets set to indicate it is an active reader. If the reader does not have a successor, it sets Tail equal to the value in its pred element. As a result, the Tail points to its predecessor if it has a predecessor or gets set to NULL otherwise. If the reader does have a successor, it sets the pred element of its successor to the value in its pred element either linking it to the predecessor or setting it to NULL.

**A Reader Releasing the Lock**

```c
rd_release(struct rw_lock *L, struct rw_proc *I) {
    // succ is a local variable.
    1. acquire(L->dLock, &I->Proc) // exclusive lock to doubly linked list
    2. if (I->pred != NULL) // if there is a predecessor,
       3. I->pred->succ := I->succ // link it to the successor
    4. if (L->Tail == I) // if this processor is at the tail,
       5. L->Tail := I->pred // point Tail to the predecessor
       6. else // otherwise,
          7. I->succ->pred := I->pred // link successor to the predecessor
       8. release(&L->dLock, &I->Proc) // release exclusive lock
```

**A Writer Releasing the Lock**

```c
wr_release(struct rw_lock *L, struct rw_proc *I) {
    1. acquire(&L->dLock, &I->Proc) // exclusive lock doubly linked list
    2. if (L->Tail == I) // if no successor,
       3. L->Tail := NULL // free the readers-writers lock
       4. else // if there is successor
          5. I->succ->pred := NULL // pass the lock on to the successor
       6. release(&L->dLock, &I->Proc) // release doubly linked list lock
```

**Figure 40. Algorithms to Release the Lock in RW.DL-1**

Since a writer releasing the lock can not have a predecessor, it simply checks for a successor after acquiring the exclusive lock to the doubly linked list. If it has no successor it frees the lock. If it does have a successor, it passes the lock on to the successor by changing its pred element to NULL and releases the exclusive lock to the doubly linked list.
3.5.2 Only Active Readers Lock the Doubly Linked List (RW.DL-2)

This approach allows more concurrency than the previous approach. Since the processors acquiring the lock always link at the tail of the doubly linked list, writers as well as readers can acquire the lock without exclusive access to the doubly linked list. Also a writer may release the readers-writers lock without locking the doubly linked list as it is the only processor that will release the lock at a time. This leaves the active readers as the only entity that requires exclusive access to the doubly linked list to release the readers-writers lock. This observation leads to the second approach presented in this section.

3.5.2.1 Data Structure

The local data structure for this approach is exactly the same as for the previous approach. The local data structure for the mutual exclusion lock is used only by the active readers as they are the only ones requiring exclusive access to the doubly linked list.

```
struct rw_lock {
    struct lock dLock;          // exclusive lock to access the doubly linked list
    struct rw_proc *Head;       // Head to maintain the FIFO behavior
    struct rw_proc *Tail;       // Tail of the doubly linked list
} rwLock;
```

**Figure 41. Data Structure for the Global Shared Memory for RW.DL-2**

We need two global variables, Head and Tail, for the doubly linked list to allow processors to directly link at the tail of the doubly linked list and acquire the lock while maintaining the FIFO behavior. Therefore, the global data structure has one additional global variable as shown Figure 41.

3.5.2.2 Algorithm RW.DL-2

In this approach a processor links itself at the tail of the doubly linked list and determines whether the lock is free. If the lock is free, the processor, reader or writer, acquires the lock. If the lock is not free, the processor checks if the predecessor is an active reader. If the predecessor is not an active reader, the processor simply links behind its predecessor. If the predecessor is an active reader, the processor first updates its pred to point to the
predecessor and then links behind it. At this point, a writer simply waits for the lock spinning on its pred element until its pred element is updated to NULL by its predecessor. A reader, however, acquires the lock if the predecessor is an active reader and waits spinning on its pred element if the predecessor is not an active reader. The pred element of a reader is either updated by a writer predecessor that releases the lock or by a reader that acquires the lock and passes the lock on to its successor. In the first case the pred is updated to NULL. In the second case it points to its predecessor completing the back link.

After acquiring the lock, a writer stays at the head the doubly linked list until it releases the lock. At that point it checks if it has a successor. If it does, it passes the lock on to the successor. Otherwise, it frees the lock. A reader on the other hand, checks if it has a successor right after acquiring the lock. If it has a successor, it links the successor's pred to itself. This signals the successor that it can acquire the lock if it is a reader. On the other hand, if the reader finds no successor, it takes the same steps a processor takes to release the linked list based mutual exclusion lock except for the last step. Instead of setting Head to NULL to indicate the lock is free, it points Head to itself. This is a convenient way to indicate to a successor that its predecessor is an active reader.

To release the lock, a reader acquires exclusive lock to the doubly linked list, unlinks itself, and releases the lock. The lock passes on to the next waiting successor. A writer simply takes itself off of the linked list freeing the lock or passing it to a waiting successor.

**Initialization**

```c
rw_initialize(struct rw_lock *L)
    initialize (&L->dLock) // set exclusive lock to free
    L->Head := NULL // set readers-writers lock to free; NULL
    L->Tail := NULL // is defined to be 0
```

**Figure 42. Code to Initialize the Lock in RW.DL-2**

To initialize the readers-writers lock, we initialize the exclusive lock to free and set the Head and Tail for the doubly linked list to NULL.
To acquire the lock a reader sets its pred for busy waiting and its succ to a known value. Next, it swaps the contents of Tail with a pointer to itself. If the prior contents of Tail indicate it was not NULL, there is a predecessor waiting ahead of this reader. The reader links behind the predecessor and waits. If the prior contents of Tail indicate it was NULL, the reader needs to check Head to determine whether it can acquire the lock. The reader swaps the contents of Head with a pointer to itself and examines the prior contents of Head. The prior contents of Head may indicate it is in one of the following three states.

1. Head is NULL indicating the lock is free. The reader sets its pred element to NULL and acquires the lock. It is important for the reader to set its pred element properly since it is pred that it later uses to know whether it has a predecessor.

2. Head points to itself indicating the lock is not free. The reader simply waits. A pointer to it is available in Head and can be used by its predecessor to pass the lock on to it.

3. Head is neither NULL nor points to itself. This means the Head points to a predecessor which is an active reader. This reader first completes its back link, then links behind the predecessor using atomic swap. If the value returned by swap is a pointer to Head, the reader has successfully linked itself and can acquire the lock. If the return value is not the address of Head, the predecessor is in the process of unlinking itself. This reader waits for its pred to be updated, at which point it can acquire the lock. The reader acquiring the lock must set its pred before it links behind the predecessor. The predecessor is an active reader and may unlink itself anytime. When it unlinks, it allows a successor to complete linking behind it. After which it links the successor to its predecessor. If this reader updates its pred after linking behind the predecessor, it may update the pred after its predecessor, pointing its pred to the processor that just unlinked.

After acquiring the lock the reader executes steps that are similar to releasing the mutual exclusion lock but with the following differences. Lines 16 through 21 are the same as for releasing a mutual exclusion lock. However, it swaps the contents of Head with a pointer to itself on line 22 rather than to NULL. This points Head to this reader. The reader checks the return value from the swap atomic operation. If a processor was waiting prior to the swap, the reader links it behind itself. Next, it points the waiting processor's pred element...
to itself. As a result, the waiting processor will acquire the lock if it is a reader. If no pro-
cessor was waiting, Head has been changed to point to this reader. This, in turn, conveys
to a subsequent successor requesting the lock that its predecessor is an active reader.

**A Reader Acquiring the Lock**

```c
rd_acquire(struct rw_lock *L struct rw_proc *I)
1. I->succ := I
2. I->pred := I // prepare to wait
3. tail := swap(&L->Tail, I) // link at the tail
4. if (tail ≠ NULL)
5. tail->succ := I // tail points to a waiting predecessor
6. repeat while (I->pred == I) // link behind it and wait
7. else // tail is NULL
8. if ((head:swap(&L->Head, I)) == NULL)
9. I->pred := NULL // lock is free, acquire the lock
10. else if (head == &L->Head) // Head points to itself
11. repeat while (I->pred == I) // wait
12. else // predecessor is an active reader
13. I->pred := head // link behind it, and
14. if(swap(&head->succ, I) ≠ &L->Head)
15. repeat while (I->pred == head)
   // At this point, the reader has acquired the lock
16. if (L->Tail == I) // if there is no successor
17. L->Head := &L->Head // point Head to itself, Tail to NULL,
18. Plast := swap(&L->Tail, NULL) // and last processor prior to
19. Plast->succ := &L->Head // swap, to Head
20. repeat while ((succ:=I->succ) == I) // wait until succ is updated
21. if (I->succ == &L->Head) // if successor points to Head,
22. succ := swap(&L->Head, I) // point Head to yourself
23. if (succ ≠ &L->Head) // if there is a successor,
24. I->succ := succ // link it behind yourself
25. succ->pred := I
```

Figure 43. Algorithm for a Reader to Acquire the Lock in RW.DL-2
A Writer Acquiring the Lock

wr_acquire(struct rw_lock *L struct rw_proc *I)
1. I->succ := I
2. I->pred := I // prepare to wait
3. tail := swap(&L->Tail, I) // link at the tail
4. if (tail # NULL)
5.   tail->succ := I // tail points to a waiting predecessor
6.   repeat while (I->pred # NULL) // link behind it and wait for the lock
7. else if (head := swap(&L->Head, I) # NULL) // otherwise (tail is NULL)
8.   if (head # &L->Head) // Head points to an active reader
9.     I->pred := head // update pred and
10.    head->succ := I // link behind the active reader
11.   repeat while (I->pred # NULL) // now wait for the lock

Figure 44. Algorithm for a Writer to Acquire the Lock in RW.DL-2

The algorithm to acquire the readers-writers lock for a writer is similar to the one for a reader up to the point the reader acquires the lock. The differences are listed below.

1. A writer waits until its pred element becomes NULL. This is because the writer requires exclusive access. When its pred is updated to NULL, it indicates the writer is at the head of the doubly linked list and all processors that requested the lock prior to the writer have released the lock.

2. Unlike a reader, a writer can not acquire the lock if its predecessor is an active reader. It can acquire the lock only when the lock is free. In all other cases it must wait. However, if it finds the predecessor is an active reader, it must link behind the active reader before it waits, spinning on its pred element for it to become NULL.

3. Since a writer implicitly knows it has no predecessor after it acquires the lock, it is unimportant for it to set its pred element properly. Therefore, a writer may or may not set its pred to NULL when the lock is free.
With these differences in mind, we can explain how a writer acquires the readers-writers lock. It initializes pred for busy waiting, links at the tail of the doubly linked list, and checks if Tail is NULL. If Tail is not NULL, it simply links behind the predecessor and waits for its pred to become NULL, at which point it can acquire the lock.

When Tail is NULL, like the reader, the writer also checks Head to find out if the lock is free. It swaps a pointer to itself with the contents of Head and checks the return value from the swap. If the prior contents of Head are NULL, the lock is free and the writer can acquire the lock. If the lock is not free the writer must wait. However, before it waits, it updates its pred element and links behind the predecessor if it knows the predecessor.

Since a writer is not supposed to pass the lock on to a successor after acquiring the lock, it takes no further steps until it releases the lock.

**A Reader Releasing the Lock**

```c
rd_release(struct rw_lock *L struct rw_proc *I)
   // succ is a local variable.
1. acquire(L->dLock,&I->Proc)       // exclusive lock to doubly linked list
2. succ := swap(&I->succ,I)        // signal a reader to wait
3. if (I->pred != NULL)            // if there is a predecessor,
   4. I->pred->succ := succ         // link it to the successor
5. if (I->succ == &L->Head)        // if this processor is at the tail,
   6. if (((head:=swap(&L->Head,I->pred)) != I)   // if I have a predecessor,
   7. if (I->pred != NULL)           // link predecessor to the successor
   8. I->pred->succ := head          // wait for successor to update
9. repeat while (I->succ != head)  // succ and link it to the predecessor
10 I->succ->pred := I->pred         // otherwise,
11.else                            // link successor to the predecessor
12. I->succ->pred := I->pred       // release exclusive lock
13.release(&L->dLock,&I->Proc)
```

*Figure 45. Algorithm for a Reader to Release the Lock in RW.DL-2*
To release the lock, a reader first acquires exclusive access to the doubly linked list. Next, it consumes its succ element to signal a successor it is in the process of unlinking itself. It then links its predecessor to its successor or to Head; assuming it has a predecessor. Then it checks if it points to Head and if it does, it points Head to its predecessor. If no processor was waiting, then Head was pointing to this reader prior to the swap on line 6 and points to the predecessor after the swap. The reader has unlinked itself and released the readers-writers lock. If the reader did not have a predecessor, its pred is NULL. As a result, Head is set to NULL and the readers-writers lock is free. If a processor was waiting, the Head was pointing to the waiting processor prior to the swap as indicated by the contents returned from the swap. The reader links the predecessor to the waiting processor. Next, it waits to ensure that the waiting processor has linked behind it and then updates the back link of the waiting processor. It is important for the reader releasing the lock to wait until the successor links behind it before updating the successor's pred element. This ensures that the successor has updated its pred and the value written in the pred by this reader will not be incorrectly written over by the successor.

If the reader did not point to Head, it has a successor waiting for the lock. It links the successor to its predecessor if it has one, or sets its successor's back link to NULL otherwise.

**A Writer Releasing the Lock**

```c
wr_release(struct rw_lock *L struct rw_proc *I)
1. if (L->Tail = I) // if there is no successor
2. L->Head := &L->Head // point Head to itself, Tail to NULL,
3. Plast := swap(&L->Tail,NULL) // and last processor prior to
4. Plast->succ := &L->Head // swap, to Head
5. repeat while (I->succ = I) // wait until succ is updated
6. if (I->succ = &L->Head) // if successor points to Head,
7. I->succ := swap(&L->Head,NULL) // point Head to NULL
8. if (I->succ != &L->Head) // if there is a successor, pass it the
9. I->succ->pred := NULL // lock by updating its pred to NULL
```

Figure 46. Algorithm for a Writer to Release the Lock in RW.DL-2

101
A writer takes exactly the same steps to release the lock as a processor does to release the mutual exclusion lock. The only difference is the use of element pred instead of spin and the value used to update pred to pass the lock.

3.5.3 No Processor Locks the Doubly Linked List (RW.DL-3)

In this approach not only can readers acquire the lock concurrently with other readers releasing the lock, but also multiple readers can release the lock concurrently except when they are adjacent to each other. This approach allows maximum concurrency at the expense of additional complexity.

If we refer back to the algorithms presented in the last approach in Section 3.5.2, we note that only readers used exclusive access to the doubly linked list. While acquiring the lock readers as well as writers directly linked at the tail of the doubly linked list and acquired the lock if they could, or waited otherwise. Similarly, the writers did not use exclusive access to the doubly linked list to release the lock. Therefore, it seems reasonable to expect that the algorithms for a reader to acquire the lock and for a writer to acquire as well as release the lock should be the same as presented in the previous approach in Section 3.5.2. The fact is that they are the same and will not be repeated here. So this approach differs from the previous approach only in how the readers release the lock. Accordingly, this section will only focus on a reader releasing the lock.

To unlink from a doubly linked list, a processor must link its predecessor to its successor and its successor to its predecessor. Since a processor knows its successor and predecessor through its fore and back links respectively, these links must not change during the time the processor is unlinking. A processor P_k can preserve its back link L_{kj} by locking its predecessor P_j because its predecessor cannot change L_{kj} unless the predecessor can lock itself; refer to Figure 47. Likewise, P_k can preserve its fore link by locking itself because its successor cannot change L_{ik} unless the successor can lock P_k. Therefore, a processor needs to lock its predecessor and itself before it unlinks itself; as an alternative a processor can lock itself and its successor. Whether a processor locks itself first or its predecessor first is not important. As long as all processors follow the same order, a deadlock will not occur.
It is important to understand different scenarios when a processor is unlinking. Assume processor \( P_k \) has locked its predecessor and itself and is unlinking from the doubly linked list as shown in Figure 47. If neither \( P_j \) nor \( P_i \) request the lock when \( P_k \) is unlinking, \( P_k \) unlinks itself and unlocks \( P_j \). If \( P_j \) attempts to lock itself, it will find it is already locked. However, it needs to distinguish whether it is locked by \( P_k \) or additionally locked by \( P_i \) while \( P_k \) has not released the lock yet.

**Figure 47. Unlinking from a Doubly Linked List**

If \( P_i \) attempts to lock its predecessor, it may see \( P_k \) as its predecessor if it attempts *before* its back link is updated to \( L_{ij} \) or see \( P_j \) as its predecessor if it attempts *after* its back link is updated. In the first case it must recognize that its predecessor is unlinking. In the second case it must recognize whether \( P_j \) is locked by \( P_j \) itself or still locked by \( P_k \). If it is locked by \( P_j \), it becomes the first case. If it is locked by \( P_k \) then \( P_i \) can lock \( P_j \) except it must wait until \( P_k \) unlocks \( P_j \). When a processor finds its predecessor is locked, *either* it can wait for its pred to be updated and attempt to lock the new predecessor or its predecessor can attempt to lock the new predecessor on its behalf. In the first case a processor may make \( N \) attempts to lock a predecessor. In the second case, \( N \) of its predecessors may each make one attempt on its behalf to lock a predecessor. The overall waiting time for a processor to lock its predecessor is finite and roughly the same. At the most it may make \( N \) attempts where \( N \) is the number of its predecessors. The algorithm, however, is somewhat simpler when a processor itself retries. Also, it reduces the number of instructions by a few that its unlinking predecessor otherwise would have to execute on its behalf. Since it
is waiting it can execute those instruction for itself by using its waiting time instead of 
having its predecessors spend their useful time on its behalf.

Given the above discussion it should be obvious a processor can not spin on its fore or 
back link when its fore or back link is already updated but the unlinking processor has 
not unlocked its predecessor yet. The discussion also shows that there could be two pro­
cessors requesting the lock and a third processor releasing the same lock. Were the lock 
associated with each processor always contested by only two processors at a time, a single 
variable would suffice to represent the lock as shown in Section 2.3.2 in CHAPTER 2. 
Given three processors access the same lock, we instead need two variables for the lock.

3.5.3.1 Data Structure

The lock consists of a data structure that consists of two variables Tail and Head in the 
global shared memory as shown in Figure 48. The variables are used to access the doubly 
linked list to acquire the readers-writers lock and ensure the FIFO behavior in a manner 
similar to in the mutual exclusion lock or in the previous approach presented in Section 
3.5.2. The variables succ and pred in the data structure in the local shared memory are 
used to link a processor in the doubly linked list. Except for the readers releasing the lock, 
these are the only two variables used by the rest of the algorithms. The element pred is 
also used to spin while waiting for the readers-writers lock. The third and fourth elements 
are used by an active reader itself or its successor to lock the reader to release the read­
ers-writers lock. The elements tail and head serve a role conceptually similar to the role 
served by Tail and sHead in the mutual exclusion lock but for locking a single processor. 
While locking itself or its predecessor a processor uses the element spin to busy wait in 
its own local shared memory when it needs to wait.

We show the data structure with three active readers followed by a waiting writer followed 
by two waiting readers in Figure 49. This situation may arise when Pk finds itself at the 
end of the list after acquiring the lock. As a result, it attempts to point Head to itself after 
first setting it to point to Head, and then setting Tail to NULL. During the time it checks 
if it is the last processor in the list and it changes Tail to NULL, P1 followed by Pm request 
the lock. Pk waits for its successor to update its succ element and points its successor's 
pred to itself. Since the successor is a writer, it continues to wait for its pred to become
NULL before it acquires the lock. \texttt{Head} is linked behind \texttt{P_m} which links behind \texttt{P_i} and then waits for its own \texttt{pred} element to change from its initialized value. Likewise, \texttt{P_n} links behind \texttt{Head} and since \texttt{Head} is pointing to itself, \texttt{P_n} knows its predecessor is not an active reader. Further, it does not have a pointer to its predecessor at this point either. It simply waits spinning on its \texttt{pred} element until its \texttt{pred} element changes from its initialized value.

```c
struct rw_lock { // data structure in the \textbf{global shared memory}
    struct rw_proc *Head; // head of the doubly linked list
    struct rw_proc *Tail; // tail of the doubly linked list
} rwLock;

struct rw_proc { // data structure in the \textbf{local shared memory}
    struct rw_proc *succ; // pointer to a successor in the doubly linked list
    struct rw_proc *pred; // pointer to a predecessor in the doubly linked list
    struct rw_proc *head; // head of the list to lock a single processor
    struct rw_proc *tail; // tail of the list to lock a single processor
    unsigned int spin; // element to busy wait when locking a processor
} rwProc;
```

\textbf{Figure 48. Data Structures for RW.DL-3}

\textbf{Figure 49. An Instance of the Data Structure for RW.DL-3}
3.5.3.2 Algorithm RW.DL-3

Except for initializing tail and head in its local data structure, the algorithm for a reader to acquire the lock is exactly the same as in the last approach presented in Section 3.5.2. To set the local lock associated with each processor to free, each processor should initialize its tail and head to NULL before it acquires the lock for reading. The algorithms for a writer to acquire as well as release the lock are exactly the same as in Section 3.5.2. These algorithms will not be repeated here. We will only present the algorithm for a reader to release the lock without locking the doubly linked list. To initialize the lock, we set Head and Tail to NULL as shown in Figure 42. However, there is no exclusive lock associated with the doubly linked list that needs to be initialized.

To unlink itself, a reader first locks its predecessor and then locks itself. Next, it links its predecessor to its successor and the successor to its predecessor. To lock its predecessor, a processor first points its pred to itself. This signals the predecessor that it must wait for its successor to attempt the lock. Next, it attempts to lock tail of its predecessor if it has a predecessor. If the tail is already locked, it is locked by the predecessor itself and the predecessor is in the process of unlinking itself from the doubly linked list. The processor waits until its predecessor updates its pred element. At this point it attempts to lock tail of the new predecessor assuming the new predecessor exists. If it can lock tail of the predecessor it can acquire the lock without further retries. Lines 1 through 4 of the algorithm allow a processor either to lock tail of its predecessor or exhaust the list so that it does not have a predecessor, i.e., its pred element is NULL.

After locking tail, a processor attempts to lock the head element of its predecessor. If the predecessor is not locked, the head is NULL and this processor can lock the predecessor. If head is not NULL, the predecessor is still locked by a previous predecessor. This processor must wait until the previous predecessor unlocks. It must use its spin element to busy wait as its pred is already updated. Lines 5 through 8 implement this part of the algorithm.

Having locked head of its predecessor, it unlocks tail of its predecessor before it links its successor to its predecessor. This allows the predecessor and the successor to contend for locking the predecessor after this processor has updated the back link of its successor.
A Reader Releasing the Lock

```c
rd_release(struct rw_lock *L, struct rw_proc *I)

// succ and pred are local variables.
1. pred := swap(&I->pred, I)  // consume the back link
2. repeat while (pred != NULL && swap(&pred->tail, I) != NULL)
3.   repeat while (I->pred == I)  // wait until pred is updated
4.   pred := swap(&I->pred, I)  // consume the back link, try again
5.   I->spin := WAIT  // prepare to wait
6. if (pred != NULL)
7.   if (swap(&pred->head, I) != NULL)
8.       repeat while (I->spin == WAIT)  // wait until pred->head is released
9.   if (swap(&pred->tail, NULL) != I)
10.  pred->tail := pred  // pred must have requested the lock,
11.  pred->head := pred  // advance it to head
12.  I->spin := WAIT  // prepare to wait
13. if (swap(&I->tail, I) != NULL || swap(&I->head, I) != NULL)
14.  repeat while (I->spin == WAIT)  // wait until lock is passed to you
15. succ := swap(&I->succ, I)  // signal the successor
16. if (pred != NULL)  // if there is a predecessor,
17.  pred->succ := succ  // link it to the successor
18. if (succ == &L->Head)  // if the successor is Head
19.   if ((head:=swap(&L->Head, pred)) != I)  // point it to the predecessor
20.   if (pred != NULL)  // re-link the predecessor to the
21.     pred->succ := head  // successor
22.   repeat while (I->succ != head)  // wait for the successor to link
23.     I->succ->pred := pred  // and update successor's back link
24. else  // it is a proper successor
25.   if (swap(&succ->pred, pred) != I)  // update its back link and if it
26.     repeat while (I->tail == I)  // is unlinking, let it request the lock
27.   if (pred != NULL)  // if there is a predecessor, unlock it
28. if ((lock:=swap(&pred->head, NULL)) != I)
29.   pre->head := lock  // and pass the lock to the waiting
30.   lock->spin := FREE  // processor
```

Figure 50. Algorithm for a Reader to Release the Lock in RW.DL-3

107
The one that proceeds will find the head is locked and wait spinning on its spin variable. When the reader unlocks its predecessor, it will grant the lock to the processor that reached head. If no processor reached head, it will free the predecessor. Lines 9 through 11 free tail of the predecessor taking into account that the predecessor may have already requested the lock. Since the back link of the successor is not updated to point to the predecessor yet, the successor could not have attempted to lock the predecessor.

Next, the processor attempts to lock itself. If it finds either its tail is not NULL, or its head is not NULL, it waits spinning on its spin variable. As soon as its successor unlocks it, it will be passed the lock. Lines 12 through 14 implement this part.

Lines 15 through 26 link the predecessor to the successor, and the successor to the predecessor, taking into consideration the special cases such as when the processor is at the head or tail of the doubly linked list. When it is at the head of the doubly linked list, it has no predecessor that it needs to link to the successor. On the other hand, when it is at the tail of the doubly linked list, it may not have a successor but another reader or writer may be trying to link at the tail. In particular, when a reader attempts to link at the tail, we want the joining reader to acquire the readers-writers lock without any undue delay. This is the same approach that is used in Section 3.5.2. Except for substituting pred in place of i->pred, the part of the algorithm that deals with letting a successor to link at the tail is exactly the same as in Section 3.5.2. The part that links an already linked successor to its predecessor, lines 24 through 26, somewhat differs. The fundamental difference is that in addition to linking the successor to its predecessor, it checks if the successor has started a request for locking its predecessor. If so, it waits for the successor letting it attempt to lock its predecessor. Since we do not want a processor accessing the memory that may be released, the processor unlinking itself must wait and allow its successor to access its memory before it completes its release of the readers-writers lock.

All a reader unlinking itself needs to do after that is to unlock its predecessor. It unlocks the predecessor if the predecessor exists. If a processor, its successor or predecessor, is waiting to lock the predecessor, it updates spin of the waiting processor. As a result the waiting processor gets the lock for the predecessor.
3.6 Summary

We have presented a number of solutions to the readers-writers problem. The solutions that use the fetch&add read-modify-write capability may be used by architectures that support the fetch&add atomic primitive. We have provided two solutions. Both of our solutions guarantee the FIFO behavior. The first solution relies on coherent cache to eliminate contention. The second uses local shared memory for busy waiting and eliminates contention from architectures with or without coherent caches. Architectures that support coherent caches may use the first solution. The first solution is simpler not only compared to the second solution but also compared to the rest of the solutions that are based on linked lists. Both of these solutions, however, require an array the size of which must be at least one greater than the number of processors that may simultaneously request the lock. The array, along with global variables Tail and Head, is required for each lock. The first solution may also be used in distributed shared memory architectures. As long as each element of the array is declared in a different memory module, the busy waiting will not cause hot spots in the interconnection network. To the best of our knowledge, no solution similar to these have been implemented before.

We have also implemented a number of solutions that use linked lists as the data structure and swap as the only read-modify-write primitive. All of these solutions guarantee the FIFO behavior. Further they all use the local shared memory for busy waiting and, therefore, eliminate contention on architectures with or without coherent caches. Each of these solutions require a small constant amount of per lock global memory as compared to the per lock global memory linear to the number of processors in the fetch&add based solutions. However, these solutions are comparatively more complex.

The first of these solutions uses a singly linked list and is conceptually similar to Mellor-Crummey and Scott's solution. Their solution requires atomic decrement, atomic increment, compare&swap, and swap. Our solution provides the same characteristics but requires swap as the only atomic primitive.

Next, we offered three solutions using a doubly linked list. These solutions are variants of the same concept and offer a trade-off between complexity and concurrency. Each of these solutions guarantees the FIFO behavior using swap as the only atomic primitive.
These solutions negate the assertion that it is not possible to detect and deque an element at the tail of a list without using \textit{compare\&swap} [50].

All these locks are extensively tested on an actual shared memory multiprocessor. Versions that contain bugs can not continue execution beyond a minute or so. Correct versions execute for many hours. We have quantitatively studied and compared the performance of different algorithms. The results are presented in CHAPTER 5.

We have not considered other variants of readers-writers in our work. We believe the techniques presented here can be extended to other variants, e.g., reader preference, or writer preference, as well. All of the linked list based readers-writers algorithms can be simplified to varying degrees using \textit{compare\&swap}. 
CHAPTER 4

LOCK FREE PRODUCER-CONSUMER

The producer-consumer problem was first introduced by Dijkstra in his seminal paper [20] and later discussed by Hansen [35] and others [12][89] in detail. It is one of the basic communication problems in multiprocessing. A producer process produces new items and appends them at the end of a queue. A consumer process removes these items from the front of the queue and consumes them. The queue is a shared object which may be manipulated using exclusive access. As an alternative it may be implemented as a concurrent object. A concurrent implementation allows concurrent access to the object without using critical sections. A queue may be implemented using a bounded buffer, i.e., an array, or using a linked list. We have developed new algorithms that provide concurrent implementation of a FIFO queue for both of these data structures. We present these algorithms in this chapter. The algorithms use fetch&add or swap atomic primitives for multiple producers, single consumer access.

4.1 Introduction

An abstract view of the producer-consumer problem is presented in Figure 4. We elaborate on the part that deals with accessing the queue in Figure 51.

When a producer makes an attempt to acquire a buffer, an empty buffer may not be available. Likewise, when a consumer attempts to remove a buffer, a full buffer may not be available. In the event a buffer is not available, a process, producer or consumer, may:

1. return and retry later after some time,

2. busy wait for a buffer to become available, or

3. block itself, in which case it must be signaled when a buffer becomes available.
When a buffer is available, a processor does not need to wait. It can acquire a buffer, fill in the data, queue the buffer, and inform the consumer if it is a producer. Likewise, it can remove a buffer, empty the data, free the buffer, and inform the producer if it is a consumer. There is no inherent wait involved except when a buffer is not available. However, to acquire, queue, remove, or free a buffer, a process needs to update global queue information. This information is shared by all producers and consumers which may all attempt to concurrently update it. In a traditional implementation, a producer locks the entire queue structure, acquires a buffer, writes its data to the buffer, queues the buffer, and unlocks the queue structure. During the time a processor has exclusive access to the queue, other processors are blocked.

### Figure 5.1. An Abstract View of the Producer-Consumer Problem

An alternative to using critical sections is wait free synchronization [39]. In this chapter we show that using currently available atomic read-modify-write primitives, it is possible to implement highly concurrent data structures. As long as empty buffers are available to a producer and full buffers are available to a consumer, producers and consumers can all concurrently acquire, queue, remove, and free the buffers. Such a solution is lock free as it does not use critical sections to manipulate the queue. A lock free solution is also wait free if it guarantees that each process will complete a concurrent operation in a finite number of steps. Except for one, the algorithms presented in this chapter are wait free.

Dijkstra alludes to two possible implementations of the producer-consumer problem: a bounded circular array based solution which he calls cyclic buffering and a linked list based solution which he refers to as chaining [21]. An array based solution is simple but
the size of the array is fixed a priori and, therefore, can not be changed during execution. A linked list offers more flexibility but relatively may be more complex. We implement lock free producer-consumer solutions using both of these data structures. All of our circular array based solutions are wait free. The linked list based multiple producers, single consumer solution is also wait free. However, in the linked list based multiple producers, multiple consumers solution, the consumers use a contention free lock to exclude each other; the producers can queue concurrently with other producers as well as consumers.

When a buffer is not available, a process may return from its queue (or remove) function and later retry, it may busy wait, or it may block itself. In this chapter we present solutions that let a process return from its queue (or remove) function. These solutions can be extended to busy wait or block when a buffer is not available.

We present a brief overview of the related work from the literature in Section 4.2. We present two new array based algorithms in Section 4.3. The first is simpler, works for multiple producers, single consumer, and uses atomic \texttt{fetch\&add} as the only atomic primitive. The second works for multiple producers, multiple consumers but requires \texttt{compare\&swap} in addition to \texttt{fetch\&add}. Next, we present two new linked list based algorithms in Section 4.4. The linked list based algorithms implement a linked list based FIFO queue using \texttt{swap} as the only atomic primitive. The first works for multiple producers, single consumer and is lock free as well as wait free. This we believe is a novel contribution. The second extends it to multiple producers, multiple consumers. However it is a hybrid, in that the producers are lock free as well as wait free, but consumers block each other. The consumers, however, do not block the producers.

4.2 Related Work

The producer-consumer communication is one of the basic forms of communication and is widely used in operating systems, communications systems, and multiprocessor systems. It is one of the basic synchronization problems. The problem was introduced by Dijkstra and later discussed by Hansen and others in detail [12][20][35][89]. Traditional solutions to the problem rely on critical sections. The queue is protected by a lock. A process acquires exclusive access to the queue, queues or removes an item and unlocks the queue. Traditional solutions are based on semaphores or other similar mechanisms such
as conditional critical regions or monitors [12][34][43][45][89]. These do not allow concurrent access to the queue.

A concurrent implementation of a FIFO queue is more suitable in highly parallel systems as it allows higher parallelism. The concurrent implementation should not only be lock free but also wait free for it to be efficient. If the implementation is not wait free, the queue is heavily accessed, and the processes have differing speeds, starvation is possible.

A number of researchers have proposed or implemented concurrent queues. Hwang and Briggs, and others have proposed lock free algorithms based on compare&swap [45]. These algorithms are incompletely specified as they omit a number of details. Lamport presents a wait free algorithm that allows one process to concurrently queue with another process that removes items from the queue [56]. This is a simple but elegant solution, uses an array to implement the queue, and does not require any atomic primitive. Gottlieb et al., present a highly concurrent queue implementation using arrays and the fetch&add atomic primitive [30]. This implementation does allow processes to concurrently queue and remove, but uses critical sections to synchronize access to individual queue elements. Rudolph presents a design for an unbounded concurrent queue which represents a single queue as an array of linked lists [96]. Different linked lists of the array may be accessed concurrently but a given linked list is protected using a lock. Herlihy and Wing present a concurrent queue using fetch&add and swap primitives [36]. The queue operation is wait free while the remove is lock free but not wait free. Also, they use an infinite array in their implementation. The practical arrays are bounded and it is nontrivial to change their algorithm for bounded arrays.

Mellor-Crummey presents a linked list based queue using swap and compare&swap [70]. The queue operation is wait free but the remove operation uses a lock. Mellor-Crummey uses compare&swap to maintain the list. Our linked list based solution is close to his solution. However, we maintain the list using swap as the only atomic primitive.

There are a number of other algorithms [75][87][88][101]. Most of these use compare&swap or its equivalent atomic read-modify-write capabilities. There is no linked list based wait free solution that has been reported in the literature.
4.3 Array-Based Solutions to the Producer-Consumer Problem

The array as a bounded buffer offers a convenient and simple data structure to implement a solution to the producer-consumer problem. Each element of the array may be either a structure that itself carries the data or a pointer to a structure that carries the data. In all the algorithms presented here, we will take each element of the array to be a pointer to a structure that carries the data. The array, therefore, will always be an array of pointers. Our choice is deliberate and has certain advantages over the alternative; we will explain some of the advantages shortly.

Array-based concurrent queues are a highly useful data structure and have been addressed in the literature. Gottlieb et. al. present a queue implementation that is highly concurrent [30]. It does not lock the entire queue for queuing or removing an item. However, it does lock the individual queue elements for access. Herlihy and Wing give a non-blocking queue implementation [36]. They, however, implemented an infinite queue which avoids some of the problems associated with finite queues. Practical queues are finite and a lock free solution offers a concurrent queue object of practical significance. We discuss such a solution in the context of a solution to the producer-consumer problem.

We will present solutions for a single producer and single consumer, multiple producers and a single consumer, a single producer and multiple consumers, and multiple producers and multiple consumers. As we will see, the single producer, single consumer solution does not require atomic read-modify-write primitives and can be implemented using simple reads and writes. Before we present these solutions, however, we make some observations about arrays and their characteristics.

4.3.1 Observations about Arrays

An array of $N_a$ elements consists of $N_a$ contiguous memory elements. The size of each element is declared prior to declaring the array itself. Each element of the array is indexed ranging from 0 to $N_a - 1$ or 1 to $N_a$ depending on the language. We will use the indexes from 0 to $N_a - 1$ according to C language convention. The array is considered circular if we consider the array element indexed 0 to follow conceptually the array element indexed $N_a - 1$. 

115
Elements of an array \(A[N_a]\) may be accessed using an index or a pointer; we will refer to the first as the index form and the latter as the pointer form of accessing the array elements. A producer always queues an item at the next available element of the circular array. Likewise, a consumer always removes an item from the next element when an item is available. The first element may be accessed using \(A[i]\) or using a pointer \(P\), where \(i\) is initialized to 0 and \(P\) is initialized to \(A[0]\). The subsequent elements may be accessed using \(A[i]\) or \(P\), after incrementing \(i\) or \(P\).

Adding 1 to index \(i\) results in a simple integer addition. However, adding 1 to \(P\) results in adding \(s\), the size of the structure of the element, to pointer \(P\). When we access the next element using \(P\), however, it simply provides the address of the next element. On the other hand, when we access the element using \(A[i]\), the address is computed as \&A[0]+i\times s\), where \(s\) is the size of the element. If the size of the element is not a power of 2, multiplication results in an expensive operation as a compiler has no choice but to use the multiplication operation itself. Use of a pointer avoids the multiplication. This is a potential performance advantage that the pointer form has over the index form. The index form, however, can match it if the size of the element is a power of 2. Better yet, processors provide a scaling factor that can be used as a multiplier in address calculations of indexed elements. The scaling factor is normally one, two, four, or eight. As long as the size \(s\) of the element is equal to one of the scaling factors, there is no performance penalty in address calculations using the index form. If we chose the array to be an array of pointers we ensure the size of an element is equal to a scaling factor. The index form of addressing is then as efficient as the pointer form of addressing.

Otherwise, the index form of accessing array elements has some advantages over the pointer form. It is simple, intuitive, and provides information as to the beginning of the array and which element of the array is being accessed. If we consider an unsigned integer \(i\) as an indexing variable and let \(i\) be any positive integer, we can derive a corresponding index by taking \(i \mod N_a\). We can access the next element in the circular queue by adding one to \(i\) and taking \(i \mod N_a\) except when \(i\) rolls over from its maximum value to 0. With this exception, the value \(i \mod N_a\) maps \(i\) onto \(0 \rightarrow N_a - 1\) as \(i\) varies \(n \times N_a \rightarrow (n+1) \times N_a - 1\). This is graphically shown in Figure 52. For an arbitrary array size \(N_a\), \(i \mod N_a\) will map...
nxNa→nxNa+k to 0→k and will start mapping nxNa+k+1 starting from 0 again for some k such that nxNa+k is the largest integer a computer can represent. The effect is as if the size of the queue has been reduced. This is not a major problem as long as Na is chosen such that the size is not drastically reduced.

As an alternative, we may fix the size of the array to be a power of 2. This ensures that i mod Na will always map i onto 0→Na-1 including instances when i rolls over from its maximum value to 0. This is because the maximum number is an integer multiple of Na when Na is a power of 2, i.e., Na=2^k. As a result the size of the queue is not reduced when the indexing variable rolls over.

![Diagram](image)

**Figure 5.2. Mapping an Unsigned Integer Onto a Limited Range of Indexes**

There are two other advantages that result from the size of the array being a power of 2. The first is that the expensive mod operation can be replaced with an inexpensive mask operation; i mod Na = i and Na-1 when Na=2^k. The second is that the checking of an empty or a full queue is considerably simplified. There are different ways to check a full or an empty queue. A producer may use an indexing variable $I_p$ to access the next free element in the array. Likewise, a consumer may use an indexing variable $I_c$ to access the next full element in the array. If $I_p$ and $I_c$ are declared to be the unsigned integers, the expression $I_p-I_c\geq Na$ may be used to check whether the queue is full. The expression will work correctly irrespective of the rollover as long as $I_p$ and $I_c$ are declared as unsigned integers.
As an example if $i_p=2$ and $i_c=2^n-2$, $i_p-i_c=4$ where $n$ is the native size of the machine. Further, checking in this manner allows the producers to use all the array elements for queuing their items as opposed to some other implementations that need to reserve an element to distinguish between a full and an empty queue [12].

### 4.3.2 Single Producer, Single Consumer

The first algorithm deals with a single producer and a single consumer. It does not require atomic read-modify-write primitives. Even though somewhat trivial, the solution is remarkably simple. We present two alternatives. The first alternative is conceptually the same as the solution by Lamport [56]. The discussion is useful and may help understand the following sections.

#### 4.3.2.1 Data Structure

The data structure consists of two unsigned integer variables and an array. The array has $N_a$ elements. We restrict $N_a$ such that $N_a=2^k$; where $k$ is an integer. Each element in the array consists of a pointer; as an alternative each element may consist of a data structure. Unsigned integer variables $i_p$ and $i_c$ are producer and consumer indexing variables respectively. Each varies from 0 to $2^n-1$ and then wraps around to 0. The $k$ lower bits of each provide the index of the corresponding element in the array. The unsigned variables $i_p$ and $i_c$ also provide the total count of the items queued by the producer and removed by the consumer respectively until these counts roll over from maximum value of $2^n-1$ to 0. The data structure is shown in Figure 53. We also show an instance of the data structure with two items queued in Figure 53.

```c
struct array {
    unsigned int i_p;
    unsigned int i_c;
    void *A[N_a];
} Array;
```

![Figure 53. Array Based Single Producer, Single Consumer Data Structure](image)

118
4.3.2.2 Algorithm PC-A.1→1

At any point in time, \( I_p - I_c \) represent the difference in the total number of messages queued by a producer and the number of messages removed by a consumer. Stated differently, \( I_p - I_c \) represent the number of messages queued but not yet removed. If this number is 0, there is no message in the queue and there is nothing a consumer can remove. On the other hand, if this number reaches \( N_a \), i.e., the total number of elements in the array, there is no space left in the queue and a producer can not queue another item.

**Initialization**

```c
initialize(struct array *Q)
Q->lp := 0  // initialize the producer index to 0
Q->ic := 0  // initialize the consumer index to 0
```

**Single Producer**

```c
queue(struct array *Q, void *p)
if (Q->lp - Q->ic \( \geq N_a \))  // if the queue is full
    return(Q_FULL)  // return Q_FULL, otherwise
Q->A[lp & (Na-1)] := p  // queue the item
Q->lp := Q->lp+1  // increment lp, and
return(SUCCESS)  // return SUCCESS
```

**Single Consumer**

```c
remove(struct array *Q, void **p)
if (Q->lp - Q->ic \( = 0 \))  // if the queue is empty
    return(Q_EMPTY)  // return Q_EMPTY, otherwise
*p := Q->A[I_c & (Na-1)]  // remove the item
Q->ic := Q->ic+1  // increment ic, and
return(SUCCESS)  // return SUCCESS
```

**Figure 54. Array-Based Single Producer, Single Consumer**

The code to initialize the producer-consumer data structure and queue and remove an item is presented in Figure 54. \( I_p \) and \( I_c \) are both initialized to zero. The choice is arbitrary if \( I_p \) and \( I_c \) are not used to count the total number of items queued and removed.
They can be initialized to any value as long as they are both initialized to the same value. However, if \( I_p \) and \( I_c \) are used to count the total number of items queued by a producer and removed by a consumer, they must be initialized to 0.

A producer checks the space in the array by checking the difference of \( I_p \) and \( I_c \). If \( I_p \) leads \( I_c \) by \( N_a \), the producer has queued \( N_a \) items. In other words it has used all the available elements of the queue. There are no available elements in the array and the producer simply returns with a code to indicate the queue is full. If there is space, \( I_p \) points to the element where the producer can queue the next item. It queues the item, adds one to \( I_p \) and returns a code indicating success. A consumer performs complementary steps to remove an item from the queue. One may appreciate the simplicity of the algorithm and the fact that it does not require atomic read-modify-write primitives.

It is important to note that the producer updates \( I_p \) after it queues the item. Likewise, a consumer updates \( I_c \) after it removes an item from the queue. If a producer updates \( I_p \) prior to queuing the element, the consumer could incorrectly remove an item that is not yet queued. Similarly, if a consumer updates \( I_c \) prior to removing the item from the queue, a producer could incorrectly over-write an element before the consumer removes the previous item. Either of these situations would violate the safety properties.

An alternative but equally simple solution is shown in Figure 55. Here the producer and consumer communicate with each other using the elements of the array rather than communicating through the indexing variables \( I_p \) and \( I_c \). In a circular buffer, an element queued by the producer must be removed by the consumer before the same element could be used by the producer again. Instead of checking the difference, \( I_p - I_c \), to determine if there is space in the queue, a producer may check the element indexed by \( I_p \). If the element is NULL, i.e., 0, the previous item has been removed by the consumer. Otherwise, the previous item queued at the element indexed by \( I_p \) or any element after it is not yet removed. This implies there is no space available in the queue. Since a producer expects the empty array elements to be NULL, the consumer must change the element to NULL after removing the item from an element. In a similar manner, the consumer checks the next element indexed by \( I_c \). If the element is NULL, no item is queued at this element or any element beyond it. This implies there is no message that needs to be removed.
Initialization

initialize(struct array *Q)

Q->ip := 0  // initialize the producer index to 0
Q->ic := 0  // initialize the consumer index to 0
Q->A[i] := NULL; 0 \leq i < N_a  // set all elements to NULL, NULL = 0

Single Producer

queue(struct array *Q, void *p)

i_p := Q->ip & (N_a-1)  // derive the index, i_p is local variable
if (Q->A[i_p] # NULL) // if the item is not removed, the
    return (Q_FULL)  // queue is full, return Q_FULL
Q->A[i_p] := p  // queue the item
Q->ip := Q->ip+1 // increment the indexing variable
return (SUCCESS)  // and return

Single Consumer

remove(struct array *Q, void **p)

i_c := Q->ic & (N_a-1)  // derive the index, i_c is local variable
if (Q->A[i_c] = NULL) // if the item is not yet queued, the
    return (Q_EMPTY)  // queue is empty, return Q_EMPTY
*p := Q->A[i_c]; Q->A[i_c] := NULL  // remove the item and queue a NULL
Q->ic := Q->ic+1 // increment the indexing variable
return (SUCCESS)  // and return

Figure 55. Array Based Single Producer, Single Consumer - Alternative Solution

It should be noted that the original solution for a producer works equally well with the
alternative solution for the consumer. Similarly, the alternative solution for the producer
works with the original solution for the consumer. Further, when we use the alternative
consumer solution, a producer may update Ip before or after updating the contents of the
element itself. The fact that a consumer does not use Ip to determine the presence of an
item in the queue, gives a producer the flexibility as to when it updates Ip. In a similar
fashion, when we use the alternative producer solution the consumer has the flexibility
to update Ic before or after updating the element itself.
4.3.3 Multiple Producers, Single Consumer

In a solution to multiple producers, single consumer, the producers need to concurrently determine whether the queue is full. When the queue is not full, they concurrently queue their items. We discuss concurrent checking of a full queue and concurrent queuing before presenting the data structure and the algorithm.

4.3.3.1 Concurrent Queuing

Assuming the queue is not full, a producer may queue an item concurrently with other producers. Using atomic *fetch&add* it can add one to an indexing variable reading back the pre-addition. The pre-addition value gives producer the index of the element of the array where it can queue its item.

If the consumer uses the difference $I_p - I_c$, to determine the number of messages in the queue, the producer could not add one to $I_p$ to acquire an element for queuing its item. Doing so will violate the safety properties. This problem can be solved if producers use another unsigned integer counter just like $I_p$ to reserve an element to queue their items.

We refer to this variable as $R_p$ for producers and $R_c$ for consumers. A producer may atomically read and update $R_p$, queue the item, and then update $I_p$. Likewise, a consumer may atomically read and update $R_c$, remove the item from the queue, and then update $I_c$.

To update $I_p$ or $I_c$ after queuing or removing the item without using critical sections, however, requires the more powerful *compare&swap* capability. When we deal with multiple producers, multiple consumers, we have no choice but to use this approach.

In a multiple producers but single consumer, the consumer may use the alternative approach, i.e., it may use the contents of the array element indexed by $I_c$ to determine whether a message is available. The consumer, therefore, does not use $I_p$. Since multiple producers use $R_p$ to acquire the elements for queuing their items, they do not need $I_p$ either. As a result we do not use $I_p$ in this approach. This implies we do not require the *compare&swap* primitive. We prefer a solution which uses less powerful read-modify-write primitives to provide the same capabilities.
4.3.3.2 Concurrent Checking of a Full Queue

In a single producer, single consumer, available space on the queue may be determined by either taking the difference $I_p - I_c$, or by examining the contents of the array element indexed by $I_p$. Multiple producers may not examine the contents of an element without first reserving the element. A producer, after reserving the element, may find the element is not available, i.e., the queue is full. This is not a problem if the producer intends to busy wait until the element becomes available. However, if the producer wants to return, it must decrement the indexing variable before it returns. It is not possible to decrement the indexing variable and guarantee the safety property using the `fetch&add` primitive. Therefore, multiple producers cannot use the contents of the array elements to determine whether the queue is full. The following sequence illustrates the difficulty.

The difference $I_p - I_c$ gives the number of messages that have been queued but not yet removed from the queue in the single producer, single consumer. When this number reaches $N_a$, it implies the queue is full. In other words, space, i.e., the number of free elements in the array, is given by $N_a - (I_p - I_c)$. In multiple producers, single consumer the available space is given by $N_a - (R_p - I_c)$ since the elements that are reserved by the producers but not yet updated must also be subtracted from the available space. $R_p$, however, is an indexing variable and, as explained in the previous paragraph, cannot be safely used to check whether the queue is full.

A possible solution is to define a new signed integer variable $s$ to keep track of available space in the array. Mathematically, $s = N_a - (R_p - I_c)$. We use $s$ as an up down counter which is decremented by a producer when an element is reserved and incremented by the consumer when an element is freed. We may rewrite the above mathematical expression as, $s = N_a - R_p + I_c$. At initialization $R_p = I_c$. Therefore $s$ is initialized to $N_a$. Each producer atomically subtracts one from $s$ and reads its pre-addition value. If the pre-addition value is equal to or less than 0, there is no space left in the queue. However, since $s$ is not an indexing variable, a producer can add one back to restore its value before it returns. If the pre-addition value is greater than 0, an element is available. Since the producer atomically subtracted one from the number of available elements, it has reserved an element. When it atomically increments the reservation index, i.e., $R_p$, the element indexed by the
pre-addition value of \( R_p \) is guaranteed to be free. \( S \) holds redundant information. It is the same information that is provided by \( N_a - R_p + I_c \) after \( R_p \) is incremented. An element may be reserved by adding one to \( R_p \) if we knew a priori that the element indexed by the return value of \texttt{fetch\&add} is free. It may equally well be reserved by subtracting one from \( S \). Since we use \( S \) to check whether the queue is full, we implicitly reserve an element in the queue when the queue is not full. We use this approach to check whether the queue is full.

### 4.3.3.3 Conditional Atomic Addition

Ideally, we would like to atomically check and conditionally decrement or increment a variable. In other words we would like to have a primitive that reads a variable from the memory and only if the variable meets a given condition, it adds a specified value to the variable, and writes it back. If the variable does not meet the condition, its value in memory is not changed. Such a primitive, however, is not provided by present day processors.

```c
int read_igt_add(int *var, int cmp_val, int addend)

// old, new, and temp are local integer variables.
old := *var

repeat while (old > cmp_val)

new := old + addend;

temp := compare_and_swap(var, old, new) // conditionally update

if (temp == old) // if update was successful

return(old) // return the old value, otherwise

old := temp // update old to the latest var value

return(old) // did not succeed, return var
```

**Figure 56. Conditional Atomic Add Using Compare\&\&Swap**

It is possible to use \texttt{compare\&\&swap} to build such a conditional atomic add capability which we call \texttt{read_igt_add}, \texttt{read if greater then add}. This is shown in Figure 56. It meets the criteria stated above in that the variable in the memory is updated only if it satisfies the condition. However, \texttt{compare\&\&swap} is inherently a non-blocking primitive and can not guarantee a wait free implementation. In other words, a producer may make repeated
attempts to add a value to a variable in the memory conditionally. However, it has no guarantee it will succeed in a finite number of attempts. The function takes three arguments: address of the variable to be updated, the compare value to check for the condition, and the addend to be added to the variable when the condition is satisfied.

An alternate implementation of read_igt_add() using atomic fetch&add is provided in Figure 57. The first if statement in the following code may seem redundant but is important to avoid the race condition that may otherwise exist in case a number of producers repeatedly try to execute the function. The function is conceptually similar to TIR and TDR functions in [30] where the race condition is explained in detail.

```c
int
read_igt_add(int *var, int cmp_val, int addend)
    // old is a local integer variables.
old := *var
if (old > cmp_val) // if the condition is satisfied
    old := fetch&add(var, addend) // add the addend to the variable
if (old <= cmp_val) // if pre-addition value does not satisfy
    fetch&add(var, -addend) // the condition, subtract the addend
return(old) // return the pre-addition value of var
```

**Figure 57. Conditional Atomic Add Using Fetch&Add**

It is interesting to note that if we use read_igt_add() constructed using compare&swap, we can safely increment the indexing variable. As a result, we can use $N_a - (R_p - I_o)$ to check whether there is space in the queue. Since the variable in the memory is updated only if it satisfies the condition, we do not have a situation where the indexing variable is incremented and then decremented. Since we chose to implement a solution using fetch&add we could not use $N_a - (R_p - I_o)$ to check whether queue is full and had to use a redundant variable $s$. 

125
4.3.3.4 Data Structure

The data structure for a multiple producer, single consumer consists of an array, an unsigned integer variable \( I_c \), an unsigned integer variable \( R_p \) and a signed integer \( S \). The array and the variable \( I_c \) are exactly the same as in the single producer, single consumer case. The signed integer \( S \) indicates the available space in the queue. It is atomically checked and conditionally decremented by producers before queuing an item and incremented by the consumer after removing an item. \( R_p \) is the indexing variable a producer uses to acquire the element where it queues its item. It is atomically read and incremented by the producers. The data structure is shown in Figure 58.

```
struct array {
    signed int S;
    unsigned int Rp;
    unsigned int Ic;
    void *A[Na];
} Array;
```

Figure 58. Array Based Multiple Producers, Single Consumer Data Structure

4.3.3.5 Algorithm PCA-M->1

The variables \( R_p \) and \( I_c \) are initialized to 0 and the variable \( S \) is initialized to \( Na \), i.e., the size of the array. Since the consumer uses the contents of an element itself to determine whether a message is queued, it is imperative to initialize all the elements of the array properly. We initialize the elements of the array to NULL to indicate that no messages are queued, i.e., the queue is empty.

As a first step a producer checks whether there is space in the queue. The producer atomically checks and conditionally decrements \( S \) by 1. If there is space, the producer has decremented \( S \) by 1 reserving an element in the queue. When the producer atomically increments \( R_p \), the element indexed using the pre-addition value of \( R_p \) is guaranteed to be
free. If there were no space in the queue, the return value from function read_igt_add() would be equal to or less than 0 and the function returns a code indicating the queue is full. If the queue is not full, a producer atomically adds one to R_p reading back its pre-addition value. Next, it derives the index corresponding to variable R_p. The local variable r_p is the index of the element where the producer can queue its item. It queues the item and returns a code indicating success.

**Initialization**

\[\text{initialize} (\text{struct array } *Q)\]
\[Q->R_p := 0 \quad \text{// initialize the producer reservation index to 0}\]
\[Q->I_c := 0 \quad \text{// initialize the consumer index to 0}\]
\[Q->S := Na \quad \text{// initialize space to } Na\]
\[Q->A[i] := \text{NULL}; 0 \leq i < Na \quad \text{// set all elements to NULL, NULL = 0}\]

**Multiple Producers**

\[\text{queue} (\text{struct array } *Q, \text{ void } *p)\]
\[\text{if } (\text{read_igt_add}(&Q->S, 0, -1) \leq 0) \quad \text{// if there is no space}\]
\[\quad \text{return (Q_FULL)} \quad \text{// return queue is full, otherwise}\]
\[r_p := \text{fetch} & \text{add} (Q->R_p, 1) & (Na-1) \quad \text{// increment } R_p \text{ and derive the index}\]
\[Q->A[r_p] := p \quad \text{// queue the item and}\]
\[\quad \text{return (SUCCESS)} \quad \text{// return}\]

**Single Consumer**

\[\text{remove} (\text{struct array } *Q, \text{ void } **p)\]
\[i_c := Q->I_c & (Na-1) \quad \text{// derive the index from } I_c\]
\[\text{if } (Q->A[i_c] = \text{NULL}) \quad \text{// if the indexed element is NULL}\]
\[\quad \text{return (Q_EMPTY)} \quad \text{// the queue is empty, otherwise}\]
\[*p := Q->A[i_c]; Q->A[i_c] := \text{NULL} \quad \text{// remove the item, free the element,}\]
\[\quad \text{fetch} & \text{add} (Q->S, +1) \quad \text{// increment space,}\]
\[Q->I_c := Q->I_c+1 \quad \text{// increment indexing variable } I_c\]
\[\quad \text{return (SUCCESS)} \quad \text{// and return}\]

Figure 59. Array Based Multiple Producers, Single Consumer
The consumer works in a manner similar to the consumer in the alternative solution for a single producer, single consumer except that it additionally adds one atomically to variable S.

4.3.4 Single Producer, Multiple Consumers

Before we discuss multiple producers, multiple consumers, it is instructive to discuss single producer, multiple consumers. Single producer, multiple consumers is symmetrical to a multiple producers, single consumer. One may think of it as a multiple producer, single consumer where the producers are queuing empty buffers and the consumer is removing empty buffers. We may convert a multiple producers, single consumer to a single producer, multiple consumers by substituting the corresponding complementary variables. The complementary variable for space is messages. Likewise, we may use \( I_p \) and \( R_c \) as complementary variables to \( R_p \) and \( I_c \). Further, the code for a producer and consumer should also be complemented, i.e., the code for a producer will resemble the code for a consumer and visa versa.

4.3.4.1 Data Structure

The data structure, therefore, looks like as shown in Figure 60. This data structure is complementary to the data structure for multiple producers, single consumer. We obtain it by replacing the variable \( S \) for space with the variable \( M \) for messages or number of items queued. Likewise, \( I_c \) is replaced by \( R_c \) and \( R_p \) is replaced by \( I_p \).

```c
struct array {
    signed int M;
    unsigned int I_p,
    unsigned int R_c;
    void *A[Na];
} Array;
```

**Figure 60. Array Based Single Producer, Multiple Consumers Data Structure**
4.3.4.2 Algorithm PC-A.1→M

The algorithm is also a complement of multiple producers, single consumer algorithm. The variables $R_c$ and $I_p$ are initialized to 0 and the variable $M$, i.e., the number of messages, is also initialized to 0. Since the producer uses the contents of an element to determine whether the next element is free, the array must be initialized properly. All the elements of the array are initialized to NULL indicating the queue is empty.

**Initialization**

```c
initialize(struct array *Q)
Rp := 0          // initialize the producer reservation index to 0
Ic := 0          // initialize the consumer index to 0
M := 0           // initialize the number of queued items 0
Q->A[i] := NULL; 0≤ i<N  // set all elements to NULL, NULL = 0
```

**Single Producer**

```c
queue(struct array *Q, void *p)
i_p := Q->I_p & (N_a-1)  // derive the index
if (Q->A[i_p] ≠ NULL)    // if the indexed element is not NULL
    return(Q_FULL)  // the queue is full, otherwise
Q->A[i_p] := p           // queue the item
fetch&add(&Q->M, +1)    // increment the queued items by 1
Q->I_p := Q->I_p+1      // increment $I_p$ by 1, and
return(SUCCESS)         // return
```

**Multiple Consumers**

```c
remove(struct array *Q, void **p)
if (read_igt_add(&Q->M, 0,-1)) ≤0)    // if there is no item, the queue
    return(Q_EMPTY)  // is empty, otherwise
rc := fetch&add(&Q->Rc, 1) & (N_a-1)  // increment $R_c$ and derive index
*p := Q->A[rc]; Q->A[rc] := NULL    // remove the item, free the element
return(SUCCESS)                      // and return
```

Figure 61. Array Based Single Producer, Multiple Consumers
As a first step, a consumer checks whether there is a message in the queue. The consumer atomically checks and conditionally decrements \( M \) by 1. If the queue is not empty, the consumer has decremented the item count by 1, reserving an item it intends to remove. When it atomically increments \( R_c \), it will have an item that it can remove from the queue. If the queue is empty, the return value from function \( \text{read_igt_add}() \) is equal to or less than 0. We return the code indicating the queue is empty.

When the queue is not empty, a consumer atomically adds one to \( R_c \) reading back its pre-addition value. Next, it derives the index \( r_c \) corresponding to variable \( R_c \). The local variable \( r_c \) is the index of the element from where the consumer can remove an item. The consumer removes the item, frees the element by placing a NULL in the element, and returns the code indicating success.

The producer works in a manner similar to the producer in the alternative solution for a single producer, single consumer except that it additionally adds one atomically to variable \( M \) to reflect that it has queued another item.

4.3.5 Multiple Producers, Multiple Consumers

In multiple producers, single consumer, the producers used \( S \) to check and reserve space in the queue and the single consumer updated \( S \) after freeing an element. Likewise, in a single producer, multiple consumers, the consumers used \( M \) to check and reserve items for removal from the queue and the single producer updated \( M \) after queuing an item. In multiple producers, multiple consumers, the multiple producers need to use \( S \) for checking and reserving the space at the tail of the queue and the multiple consumers need to use \( M \) for checking and reserving the items for removal from the head of the queue. As explained in Section 4.3.3.2 if producers or consumers want to return from \( \text{queue}() \) or \( \text{remove}() \) functions, they need to use up down counters to check and reserve an empty or full element at the tail or the head of the queue. Were the producers or consumers busy waiting for an element not yet available, they could use the indexing variables \( R_p \) or \( R_c \) to reserve the element. When processors busy wait they do not need to undo the atomic addition to the indexing variables. They could atomically increment the indexing variable and use the pre-addition value to index and check an empty or full element in the queue. If the element is not available they could busy wait. For algorithms in which a producer
or consumer wants to return from its queue() or remove() functions, we need to either conditionally add or undo the addition to the variable used for checking whether the queue is full or empty.

Each of the multiple consumers must add one to $s$ after freeing an element. Likewise, each of the multiple producers must add one to $M$ after queuing an element. However, a consumer can not add one to $s$ after freeing an element unless each element reserved prior to it has been freed. This is an important consideration. If it is violated, the safety properties are violated. To illustrate this, consider two consumers $C_0$ and $C_1$. Both atomically subtract one from $M$ to conditionally check and reserve a message for removal from the head of the queue. Next, $C_0$ and $C_1$ each atomically add one to $R_p$ reading back $i$ and $i+1$ respectively. Now, $C_1$ gets ahead of $C_0$ and removes the contents of the element indexed by $i+1$ while $C_0$ has not yet removed the contents of the element indexed by $i$. Let us further assume at this point, that the queue is full. This implies $s=0$ and $R_p=i$. If $C_1$ adds one to $s$ and a producer checks $s$ to reserve an empty element, it will conclude an empty element is available. The producer will atomically add one to $R_p$ reading back its pre-addition value which is $i$. It will queue its item at element indexed by $i$ over-writing its previous contents which has not yet been read by $C_0$.

Therefore, a consumer can add one to $s$ after freeing an element only if all elements reserved for freeing prior to it have also been freed. Likewise, a producer can add one to $M$ after queuing an element only if all elements reserved for queuing prior to it have also been queued. This further implies that if $C_1$ can not add one to $s$ after freeing its element at $i+1$ because $C_0$ has not yet freed its element at $i$, then $C_0$ must add one to $s$ on behalf of $C_1$. As we will see, this requires compare&swap capability. The algorithm presented in this section, therefore, requires the compare&swap read-modify-writer primitive.

4.3.5.1 Data Structure

The algorithm uses signed integer $s$ to keep a count of free elements and a signed integer $M$ to keep a count of items or messages in the queue. Producers use $s$ to atomically check and conditionally reserve an empty element. Likewise, consumers use $M$ to atomically check and conditionally reserve an item for removal. Further, a producer uses $R_p$ to acquire an element to queue an item. After queuing the item, it adds one to $I_p$ provided all
the producers who acquired elements prior to it have also queued their items. Similar comments apply to the consumers that use \( R_c \) to reserve an element and \( I_c \) to indicate the element up to which all elements are freed. The data structure is shown in Figure 62.

At any time, the difference between \( R_p \) and \( I_p \), i.e., \( R_p - I_p \), represents the number of producers concurrently queuing the items. All elements up to but excluding \( I_p \) have been queued. Also, when all producers that decremented \( S \) to reserve space have incremented \( R_p \) to acquire an element, \( S = R_p - I_c \). Likewise, the difference between \( R_c \) and \( I_c \), i.e., \( R_c - I_c \), represents the number of consumers concurrently removing the items. All elements up to but excluding \( I_c \) have been freed. When all consumers that decremented \( M \) to reserve items for removal, have incremented \( R_c \) to remove items, \( M = I_p - R_c \).

```
struct array {
    signed int S;
    signed int M;
    unsigned int Rp;
    unsigned int Ip;
    unsigned int Rc;
    unsigned int Ic;
    void *A[Na];
} Array;
```

![Figure 62. Array Based Multiple Producers, Multiple Consumers Data Structure](image)

4.3.5.2 Algorithm PC-A.M→M

The first five lines of the algorithm for multiple producers are the same as in multiple producers, single consumer algorithm. Likewise, the first five lines of the algorithm for multiple consumers are the same as in the single producer, multiple consumers algorithm. The working of this part of the algorithm is explained in the corresponding algorithms and is not repeated here. We explain the rest of the algorithm for multiple producers. Due to symmetry the explanation is equally applicable to multiple consumers.
After queuing an item, a producer has the important task of updating the number of items or messages, i.e., the variable \( M \). As previously discussed, a producer may update \( M \) only if all items prior to the item it queued have also been queued. We use \( I_p \) to indicate the index of the element up to which the queueing have been completed. In other words there is no element before the element indicated by \( I_p \) for which the element is reserved but not yet written. If all the prior reserved elements have been queued and \( I_p \) and \( M \) updated then \( I_p \) must equal the pre-addition value of \( R_p \) read back by a producer, i.e., \( r_p = I_p \).

If \( I_p \neq r_p \), the producer does not update \( I_p \), breaks out of the loop, and adds \( m \) which is 0, to \( M \). A prior producer that has not yet updated \( I_p \) and \( M \) will update the two for this producer as well. If \( I_p = r_p \) the producer increments \( I_p \). Next, it increments \( m \), a local variable to count the number of items it needs to add to \( M \). Next, it increments \( r_p \) and checks if an other producer has queued an item at the next element. If so, it iterates the loop and attempts to update \( I_p \). The conditions specified in the repeat while loop allow a processor to successfully update \( I_p \) and increment \( m \) for itself and all subsequent producers that have already queued items in the queue after this processor.

Let us assume \( I_p = r_p \) for a producer \( P_q \). Then \( P_0 \) must update \( I_p \) and add one to \( M \). \( P_0 \) atomically updates \( I_p \) and adds one to \( m \) first time through the loop. Next, it increments \( r_p \) and checks if the next element is reserved and queued. If the next element has not been reserved or reserved but not queued, the loop terminates. If the next element has been written, i.e, an item has been queued, there are two interesting possibilities.

- If the producer that queued the next element attempted the update \textit{before} \( P_0 \) advanced \( I_p \), the attempt by that producer would fail. \( P_0 \) will do the update for the next element and continue further updates for elements after that.

- If the producer that queued the next element attempted the update \textit{after} \( P_0 \) advanced \( I_p \), the producer may \textit{collide} with \( P_0 \). Only one of the producers will win in this collision. The producer that loses will quit without doing any update. The one that wins will do the update and continue further updates until it either stops due to one of the two conditions in the loop, or collides, loses, and breaks out of the loop when it attempts to update \( I_p \) within the loop.
**Initialization**

initialize(struct array *Q)

\[
R_p := R_c := 0 \quad \text{// initialize reservation indexes to 0}
\]

\[
I_p := I_c := 0 \quad \text{// initialize update indexes to 0}
\]

\[
M := 0; \quad S := N_a \quad \text{// initialize messages and space}
\]

**Multiple Producers**

```
queue(struct array *Q, void *p)
    if (read_igt_add(&Q->S,0,-1)) \leq 0)
        return(Q_FULL)
    r_p := fetch&add(&Q->R_p,1)
    Q->A[r_p&(N_a-1)] := p; m := 0
    repeat
        if (compare&swap(Q->I_p, r_p, r_p+l) \neq r_p)
            break
        m++; r_p++
    while ((Q->R_p-r_p > 0) && (Q->A[r_p&(N_a-1)] \neq NULL))
    fetch&add(&Q->M, m)
    return(SUCCESS)
```

**Multiple Consumers**

```
remove(struct array *Q, void **p)
    if (read_igt_add(&Q->M,0,-1)) \leq 0)
        return(Q_EMPTY)
    r_c := fetch&add(&Q->R_c,1); i := r_c & (N_a-1)
    *p := Q->A[i]; Q->A[i] := NULL; s := 0
    repeat
        if (compare&swap(Q->I_c, r_c, r_c+l) \neq r_c)
            break
        s++; r_c++
    while ((Q->R_c-r_c > 0) && Q->A[r_c&(N_a-1)] == NULL)
    fetch&add(&Q->S, s)
    return(SUCCESS)
```

**Figure 63. Array Based Multiple Producers, Multiple Consumers**
The order of evaluation of the conditions in the repeat while loop is important. The second condition is evaluated only if the first condition is true. Programming languages such as C guarantee this order of evaluation. The first condition in the repeat while loop may seem redundant. The fact is that it is essential to ensure that \( I_p \) is not updated beyond \( R_p \). In the absence of this condition, \( I_p \) may be incorrectly updated beyond \( R_p \) when the queue is full.

4.4 Linked List Based Solutions to the Producer-Consumer Problem

One may use a linked list to construct a solution to the producer-consumer problem rather than using an array. While using a linked list one has to maintain a head and a tail to the linked list. The tail is updated by a producer. The tail is also updated by a consumer to reflect that the queue is empty when the consumer removes the last item from the queue. As a result, even when we deal with a single producer, single consumer the tail is updated by two processors. The two processors must use atomic read-modify-write primitives to update the tail. However, using an atomic read-modify-write primitive any number of producers can update the tail. Therefore, the solution for the single producer, single consumer is close enough to the solution for multiple producers, single consumer that we only present the later. The same holds true about solutions to the single producer, multiple consumers and multiple producers, multiple consumers. Again, we only present the solution to the latter.

4.4.1 Multiple Producers, Single Consumer

The first algorithm presented in this section allows multiple producers to queue items to a single consumer concurrently with the consumer removing the items. It uses swap as the only atomic primitive and maintains the FIFO behavior. The solution which we consider a novel contribution is lock free as well wait free.

If we refer back to the abstract view of the producer, it acquires and fills a buffer prior to queuing the buffer in a queue. We assume empty buffers are linked together in a queue of empty buffers. To acquire a buffer is equivalent to removing a buffer from the linked list of empty buffers. Likewise, to free a buffer is equivalent to queuing an empty buffer in the queue of empty buffers.
Empty buffers may be maintained either in one global linked list for all processors or in a linked list on per process or per processor basis. If we maintain the buffers in one global linked list, multiple processors may attempt to acquire a buffer at the same time. This is equivalent to multiple consumers which requires lock based solution. To construct a lock free solution using swap as the only atomic primitive, we restrict the empty buffers to be queued on a per process or per processor basis. If we follow this restriction, we can construct a wait free solution to the producer-consumer problem that requires no locks and uses swap as the only atomic primitive.

We associate empty buffers with the producers. Therefore, each producer has its own linked list of empty buffers. When it needs to queue an item, it acquires a buffer from its own list, writes its data in the buffer, and queues the buffer to a consumer. The consumer after removing the buffer and consuming the item, frees the buffer, i.e., queues it back in the linked list of empty buffers of the producer that owns the buffer. Since there are multiple producers, each producer must mark its buffers so that the consumer could identify which producer the buffer came from and queue it accordingly. As an example the producers may use the address of the tail of their linked list of empty buffers as the identification. The consumer could then simply queue the empty buffer in the queue pointed to by the identification pointer which is the tail of the free list for that buffer.

Even though there is no requirement to maintain the FIFO ordering for the empty buffers, there is no requirement stating otherwise either. This implies we could use the queue() function given below to free the buffers. Likewise, we could use the remove() function to acquire empty buffers. Therefore, we only discuss the details of how multiple producers can concurrently queue buffers at the tail and a single consumer remove buffers from the head of a linked list.

In a list based solution, we need a head pointer from where a consumer can remove and a tail pointer where a producer can queue buffers. Further, when a consumer removes the last buffer, it must update the tail to reflect the queue is empty. We chose tail to be a pointer to the buffer element used to link the next buffer; we refer to this element as the succ element. Therefore tail, in our implementation, is a pointer to a pointer. When the queue is empty, the tail points to the head. When the queue is not empty, it points to the
success element of the last buffer. In either case the tail always points to the element where a producer can link its buffer. When a consumer removes the last buffer, it must point the tail to the head. Using `compare&swap`, a consumer may set the head to NULL and conditionally switch the tail to point to the head. A producer may queue a buffer in between when a consumer checks the buffer it is removing is the last buffer and attempts to switch the tail to point to the head. `Compare&swap` will succeed only if the tail still points to the buffer being removed. Otherwise, it fails which implies a producer has or is in the process of queuing a buffer. The consumer waits for the producer to link the buffer and points the head to the next buffer.

When `compare&swap` is not available, the consumer cannot switch the tail conditionally. Using `swap`, it can switch the tail to point to the head unconditionally. However, if a producer queues a buffer in between when the consumer checks and switches the tail, the consumer needs the head to point to the next buffer. This necessitates the use of a secondary head. The consumer switches the tail to the secondary head and links it behind the last buffer that is queued prior to when the tail is switched. This simple but effective mechanism maintains the FIFO ordering and is our contribution.

Our first implementation uses `swap` as the only atomic primitive and guarantees the FIFO behavior.

4.4.1.1 A Solution Using `Swap` as the Only Atomic Primitive

The data structure consists of global variables `Tail`, `Head`, and `sHead` and buffers each of which consists of a `succ` and `own` members in addition to other members not shown in Figure 64.

```c
struct list {
    struct buffer **Tail;
    struct buffer *Head;
    struct buffer *sHead;
} List;

struct buffer {
    // other members;
    struct buffer *succ;
    struct buffer **own;
} Buffer;
```

**Figure 64. List Based Multiple Producers, Single Consumer Data Structure**
Tail is initialized to point to sHead which is initialized to NULL. Head is also initialized to point to sHead. This condition indicates the queue is empty. The data structure is restored to this condition whenever the queue is empty. An instance of the data structure when the queue is empty, is shown in Figure 65.

From a producer's point of view, Tail at all times points to an element where the producer may write the address of its buffer to link the buffer in the linked list. A producer atomically points Tail to succ element of its buffer and points the location returned by Tail to its buffer effectively linking it in the list. At the time of initialization or when queue is empty, Tail points to the secondary head, sHead. At all other times Tail points to the succ element of the previous buffer in the list. A producer does not need to distinguish between the two cases. It simply points the element returned by the swap to its buffer linking the buffer in the queue. We will comment on why the interrupts should be disabled while executing these two instructions shortly.

Figure 65. Queuing Buffers in an Empty List
The consumer checks if `Head` points to `sHead`. If it does, the consumer further checks `sHead`. If `sHead` is `NULL`, there is no buffer in the queue, the producer returns a code indicating the queue is empty. If `sHead` is not `NULL`, the consumer copies its contents to `Head` after which `sHead` is irrelevant until it is used again when a consumer removes the last buffer from the queue. Next, the consumer copies the pointer from `Head` to `p`. If `Head` did not point to `sHead`, this is the first step a consumer executes. `Head` in that case points to the next available buffer in the queue.

Next, the consumer checks if the buffer is the last in the queue. If it is, the consumer sets `sHead` to `NULL`. Next, it atomically points `Tail` to `sHead` reading back the address of the `succ` element of the last buffer `b_1` queued before `Tail` is switched to point to `sHead`. Next, it links `sHead` after `b_1`. If a producer queues a buffer in between when the consumer finds the buffer at the head of the queue as the last buffer and switches `Tail` to point to `sHead`, `b_1` is not the buffer being removed. Otherwise, `b_1` is the buffer being removed from the head of the queue itself. If the buffer being removed is not the last buffer in the queue, these steps are skipped over and the consumer directly reaches the following statement.

The consumer waits to ensure that if a producer is in the process of linking its buffer behind the buffer being removed, it has a chance to do so before the buffer at the head is removed. When `b_1` is the buffer being removed, its `succ` element points to `sHead` and the loop immediately terminates. Otherwise, the wait covers the window between the second and the third statements in the producer. If a producer is interrupted after it switches `Tail` but before it links its buffer behind its predecessor, the consumer has no choice but to wait which is not desirable. This is a part of the reason a producer must disable its interrupts while linking its buffer in the queue.

Next, the producer copies the contents of the `succ` element of the buffer being removed to `Head`. As a result, `Head` either points to the next buffer in the queue or to `sHead`. On every entry to the `remove()` function, the consumer checks if `Head` points to `sHead`. When it does, all buffers queued prior to where `sHead` was linked have been removed. Any buffers queued after where `sHead` was linked start a new list with `sHead` pointing to the first buffer in this new list. The consumer copies the pointer from `sHead` to `Head` and continues to remove the buffers in the order in which they are queued. On the other hand, if no more
buffers are queued after Tail was switched to point to sHead, sHead is NULL. The consumer recognizes this fact and returns a code indicating the queue is empty.

**Initialization**

initialize(struct queue *Q)

```c
Q->Head := &Q->sHead  // point Head to sHead
Q->sHead := NULL      // initialize sHead to NULL
Q->Tail := &Q->sHead  // point Tail to sHead
```

**Multiple Producers**

queue(struct list *Q, struct buffer *p)

```c
// pred is a local variable of type struct buffer **pred
p->succ := NULL       // initialize to indicate no successor

// disable interrupts here
pred := swap(&Q->Tail, &p->succ)  // pred points where the producer
*pred := p                // can link itself

// enable interrupts here
```

**Single Consumer Using Swap**

remove(struct list *Q, struct buffer **p)

```c
if (Q->Head == &Q->sHead)  // if Head points to sHead and
    if (Q->sHead == NULL)   // if sHead is NULL then
        return(Q_EMPTY)    // queue is empty
    Q->Head := sHead       // otherwise copy sHead to Head
*P := Q->Head           // copy the buffer pointer to p

if (Q->Tail == (*p)->succ)  // if this is the last buffer in the queue
    Q->sHead := NULL      // set sHead to NULL
    b1 := swap(&Q->Tail, &Q->sHead) // switch Tail to point to sHead
    *b1 := &Q->sHead      // link sHead behind b1
    repeat while ((*p)->succ == NULL) // ensure the successor is linked
        Q->Head := (*p)->succ   // point Head to the successor
    return(SUCCESS)
```

**Figure 66. List Based Multiple Producers, Single Consumer Using Swap**

140
4.4.1.2 Solution Using Compare&Swap

The data structure consists of Head and Tail; sHead is not needed. The initialization consists of initializing Head to NULL and pointing Tail to Head. The change is transparent to producers; we do not repeat the code for the producers. The consumer code is somewhat simplified as shown in Figure 67. The consumer checks Head and returns a code indicating the queue is empty when Head is NULL. Otherwise, there is a buffer in the queue. The consumer copies Head to p. Next, it checks if the buffer is the last buffer in the queue. If it is, the consumer sets Head to NULL and attempts to conditionally switch Tail to point to Head provided Tail still points to the buffer being removed. If the consumer succeeds in pointing Tail to Head, the queue is now empty and the consumer returns with the buffer. If the attempt to point Tail to Head fails, a producer has queued or is in the process of queueing another buffer at the tail. The consumer makes sure a buffer has been linked behind the one being removed, points Head to the next buffer, and returns.

**Initialization**

```c
initialize(struct queue *Q)
Q->Head := NULL
Q->Tail := &Q->Head
```

**Single Consumer Using Compare & Swap**

```c
remove(struct list *Q, struct buffer **p)
if (Q->Head =  NULL)
    return(Q_EMPTY)
*p := Q->Head
Q->Head := (*p)->succ
if (Q->Head =  NULL)
    if (compare&swap(&Q->Tail, &(*p)->succ, &Q->Head) =  SUCCESS)
        return(SUCCESS)
    repeat while ((*p)->succ =  NULL)
Q->Head := (*p)->succ
return(SUCCESS)
```

**Figure 67. List Based Single Consumer Using Compare&Swap**
4.4.2 Multiple Producers, Multiple Consumers

The multiple producer, single consumer solution presented in the previous section may be extended to multiple producers, multiple consumers. Multiple consumers may concurrently remove items from the head of the queue using either compare&swap, i.e., compare and swap of two long words or load-linked/store-conditional. However, the solution does not guarantee a starvation free behavior for consumers, and may cause contention due to consumer retries. We prefer an alternative solution that uses a contention free lock, requires swap as the only atomic primitive, guarantees FIFO ordering for the consumers, and does not cause contention. We present this solution in the following.

We associate a lock with Head. Each consumer acquires the lock, removes a buffer using one of the algorithms given in Section 4.4.1, and releases the lock. If an architecture supports compare&swap one may use the lock by Mellor-Crummey and Scott and use the consumer described in Section 4.4.1.2. On the other hand if an architecture does not support compare&swap but supports swap, one may use the mutual exclusion lock that we have developed and use the consumer described in Section 4.4.1.1. The mutual exclusion lock that we have developed guarantees the FIFO behavior and thus eliminates the possibility of starvation. Also, the lock is contention free since every waiting processor waits in its local shared memory. The producers are still lock free and wait free.

4.5 Summary

We have presented highly concurrent queues in this chapter. We show that it is possible to implement array based concurrent queues using atomic fetch&add primitive except when we allow multiple processors to concurrently queue and remove items from the queue. This particular case requires compare&swap primitive in addition to fetch&add. All of our array based implementations are lock free and wait free.

Likewise, we present a linked list based queue to let multiple processors concurrently queue and a single processor remove items from the queue using atomic swap as the only read-modify-write primitive. This implementation is also lock free as well as wait free. To allow multiple processors to concurrently remove items from the queue, however, requires either a lock or compare&swap.
Producers and consumers may use the concurrent queues to queue and remove items from a given queue. The queue() function returns an indication when the queue is full and queues the item otherwise. Similarly, the remove() function returns with an indication when the queue is empty and removes an item otherwise. When the queue is full a producer may do some other useful work and retry. Likewise, when the queue is empty, a consumer may later retry.

The queue implementations may be modified to allow a producer or a consumer wait when an empty or full buffer is not available. Contention free busy waiting techniques may be combined with the queue implementation. This would ensure system performance is not affected when some processors wait for other processors to finish their work. The implementations may further be extended where processes could block themselves rather than busy wait. Using minimum hardware support, i.e., the capability to interrupt a processor, a process that queues an item may wake up a blocked process on any processor.

In the array based solutions, we considered arrays to be the arrays of pointers. This is not an inherent limitation of these algorithms. The algorithms can be easily modified to use arrays of data structures other than the pointers. A member within these data structures may be used as a flag to indicate when the write to, or the read from, the data structure is complete.
CHAPTER 5
PERFORMANCE MEASUREMENTS

We present the performance of our mutual exclusion and readers-writers algorithms in this chapter. The performance is measured using a VME based system with 16 processors and a memory module.

5.1 Introduction
The contention free mutual exclusion and readers-writers algorithms that we have developed rely on local shared memory to eliminate contention due to synchronization. These algorithms do not rely on coherent caches. When the lock is busy, a processor requesting the lock queues itself on a queue of waiting processors. This requires a constant number of transactions over the bus. It then busy waits spinning on a variable in its local shared memory and does not generate traffic to off-board memory. Therefore, it incurs only a constant overhead due to synchronization. The performance measurements verify that the synchronization overhead remains fairly constant and does not increase as a function of the number of processors in the system.

We use a VME based shared memory multiprocessor system to measure the performance. The system configuration follows the model presented in Figure 1. The system setup and limitations are described in the next section. The factors affecting the performance and the details of how the performance is measured are discussed in Section 5.3. The results are presented in Section 5.4. The chapter is concluded with a summary in Section 5.5.

5.2 System Setup
In this section we describe the system setup, its hardware limitations, emulation of atomic swap and other instructions using atomic load/store unsigned byte, the effect of VME arbitration, and a few miscellaneous aspects of the system.
5.2.1 A VME Based Shared Memory System

We used a VME based system with 16 processors and a memory board for performance measurements. Each processor board is a commercial SPARC CPU-2CE\textsuperscript{1} single board computer manufactured by Force Computers Inc [26]. Some of its relevant characteristics are listed in Table 1. One mega bytes of memory on each board may be shared to provide the local shared memory. The separate memory board provides the global shared memory. The system provides the configuration depicted in Figure 1. Each processor board is equipped with 64 Kilobytes of cache which is used to cache instructions and data from the local private memory. VME does not support cache coherency [46]. Therefore, there is no coherent cache for the shared memory in this system. To keep the shared data consistent, it is simply not cached. This is achieved through the MMU by specifying that the global, as well as the local, shared memory pages are not to be cached.

<table>
<thead>
<tr>
<th>Central Processing Unit</th>
<th>SPARC RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Frequency</td>
<td>40 MHz</td>
</tr>
<tr>
<td>Integer Performance</td>
<td>28.5 MIPS</td>
</tr>
<tr>
<td>Cache: Data &amp; Instruction</td>
<td>64 kilobytes (used for local private memory)</td>
</tr>
<tr>
<td>MMU</td>
<td>Sun-4 MMU ASIC</td>
</tr>
<tr>
<td>VME Bus</td>
<td>32-bit IEEE/ANSI Standard</td>
</tr>
<tr>
<td>VME Bus Arbiter</td>
<td>4 Level SGL</td>
</tr>
<tr>
<td>Ethernet</td>
<td>IEEE 802.3 (7990 Lance, AMD 7992B SIA)</td>
</tr>
<tr>
<td>Programmable Timers</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 1. Characteristics of SPARC CPU-2CE

Each processor board provides an Ethernet connection. The Ethernet connections of all processor boards are connected to a host system over the network through a hub. This allows access to all processor boards from any workstation connected to the network for remotely downloading and executing programs.

Each CPU board runs the VxWorks\textsuperscript{2} operating system [106]. Downloading, executing a downloaded program, and outputting the results to either the standard output or to a file

\textsuperscript{1} SPARC and SPARC CPU-2CE are registered trademarks of SPARC International Inc.

\textsuperscript{2} VxWorks is a registered trademark of Wind River Systems Inc., Alameda, California.

145
on the host system are the main operating system capabilities that we used. The dynamic linking and loading capability provided by the operating system was particularly useful. Enabling and disabling of interrupts was the only other operating system capability that, at times, was used. In particular, the timing services provided by the operating system were not used due to their overhead. Instead, a one microsecond hardware timer is directly used for all the timing needs.

5.2.2 Hardware Limitations

The SPARC CPU-2CE boards support read-modify-write cycles across the VME bus. This allows any processor to atomically access the global shared memory as well as the local shared memory of each processor. The SPARC processor used on these boards supports two atomic instructions: load/store unsigned byte and swap. Atomic load/store unsigned byte, ldstub, moves an unsigned byte from memory into a destination register and re-writes the same byte in the memory with all ones, i.e., 0xFF. Atomic swap exchanges the contents of a long register with the contents of a memory location.

The swap instruction on these boards, however, intermittently fails. As a result, these boards could not be used to measure performance using hardware supported atomic swap. A considerable effort was expended on locating another setup that supports atomic swap and fetch&add in the hardware. A setup must also provide local as well as global shared memory and the ability to disable the coherent cache if one is supported by the system. Another two setups were tried. One of them was based on SPARC CPU-3CE boards from Force Computers Inc. After setting up the system, we learned that 3CE boards did not implement read-modify-write in the hardware. Therefore, this effort had to be abandoned. Next, we tried an in-house system at Lucent Technologies Bell Laboratories. It works and has been used to test the algorithms. However, it has only four processors and does not have global shared memory. The local shared memory of one of the processors has to be used as the global shared memory. Arbitration to this memory is not designed to be fair. Also, the processors are not of the same speed and type. Because of these reasons, we could not collect any meaningful data and abandoned this approach as well. Some other avenues were also explored but some characteristic was always missing on any potential system. Having failed to find another satisfactory setup, we decided to
use SPARC CPU-2CE boards and emulate other atomic instructions using \textit{ldstub}. The code to emulate \textit{swap}, \textit{fetch\&add}, and \textit{compare\&swap} is shown in Section 5.2.3.

The behavior of read-modify-write and shared memory access are inconsistent between off-board and on-board memory on SPARC CPU-2CE. The atomic \textit{ldstub} to an off-board memory invokes a read-modify-write cycle that works without additional support. However, the \textit{ldstub} to an on-board memory requires additional steps. The steps involve disabling the interrupts, requesting the VME bus through a bus locker register, monitoring the bus locker register for owning the VME bus, performing the read-modify-write cycle, removing the request for VME from the bus locker register, and finally enabling the interrupts. The steps used to access the on-board memory can not be used to access the off-board memory. Therefore, a user is forced to recognize whether the address of the \textit{ldstub} is local in order to perform the operation correctly. Further, the \textit{ldstub} also intermittently fails if the target memory resides on one of the other processor boards.

The local shared memory on each processor circuit pack can be accessed by the rest of the processors using its corresponding global address. However, the processor to which this memory is local can not access it using the same global address. It must access it using the local address. This also forces a user to always recognize whether the access is to its own local shared memory or to an off-board shared memory.

\textbf{5.2.3 Emulation of Atomic Swap and Fetch\&Add}\footnote{Fetch\&Add is shown as \texttt{aadd()}, i.e., atomic add, in the code.}

We emulate atomic \textit{swap} and other atomic instructions using \textit{ldstub}. One may use \textit{ldstub} to implement a lock to protect a synchronization variable \(v\). The lock, in turn, may be used to perform atomic operations on \(v\). This, at a minimum, requires four memory accesses. As an alternative, one may use a byte variable \(v_b\) as a synchronization variable. A successful \textit{ldstub} followed by a simple write to \(v_b\) then may be used to emulate \textit{swap}, \textit{fetch\&add}, or \textit{compare\&swap}. We follow this approach. The code is given in Figure 68.

For programming convenience \textit{ldstub} instruction is encapsulated in a separate \textit{ldstub()} function which is implemented in C language. It uses \texttt{asm()} \textit{escape} to make use of the assembly language instruction; see Figure 69. The second argument is not needed by the
ldstub instruction. It is just a convenient way to ensure a register is available where the value from the memory could be read.

**Code for Emulating Atomic swap**

```c
unsigned char
swap(unsigned char *p, unsigned char v)
    // ret is local variable of type unsigned char.
do
    ret := ldstub(p,v);
while (ret == 0xFF);
*p := v;
return(ret);
```

**Code for Emulating Atomic fetch&add**

```c
unsigned char
aadd(unsigned char *p, unsigned char v)
    // ret is local variable of type unsigned char.
do
    ret := ldstub(p,v);
while (ret == 0xFF);
*p := (ret+v) & 0x7F;
return(ret);
```

**Code for Emulating Atomic compar&swap**

```c
unsigned char
cas(unsigned char *p, unsigned char ov, unsigned char nv)
    // ret is local variable of type unsigned char.
ret := ldstub(p,nv);
if (ret == ov)
    *p := nv;
return(ret);
```

**Figure 68. Emulation of Other Atomic Instructions Using ldstub**
The advantage of the approach we follow to emulate atomic instructions is that it requires two memory accesses rather than four. An atomic load/store followed by a simple write will suffice in the optimistic case when there are no retries. Protecting the synchronization variable with a lock would require four accesses in a similar scenario.

However, there are disadvantages to this approach as well. First, it limits the synchronization variable to an unsigned byte. Further, the value $0xFF$ of the unsigned byte can not be used as it is reserved by the processor. The processor writes $0xFF$ to the byte in the memory from where it reads using $ldstub$. The other disadvantage is that a simple read of a synchronization variable may return the intermediate value, i.e., $0xFF$. This problem can be avoided if the synchronization variable is updated using a lock.

**Function $ldstub$ in C with $asm$ Escape**

```c
unsigned char
ldstub(unsigned char *p, unsigned char v)
{
    asm("ldstub [%0o],%01");
    return(v);
}
```

**Figure 69. Function $ldstub$ for SPARC processor**

A synchronization variable may not be written directly when emulated instructions are used. This applies equally to the approach we follow or any other approach. When the atomic instructions are directly supported by the hardware, simple reads and writes may be mixed with atomic read-modify-writes when it is otherwise appropriate to do so.

A loop is embedded in the emulation of $swap$ and $fetch$&$add$. Conceptually, the synchronization variable is momentarily locked by atomically swapping $0xFF$ with the value in the variable. The processor that succeeds in reading a value other than $0xFF$ can write a new value. A processor that reads $0xFF$ must retry. Excessive retries in $swap$ or $fetch$&$add$ will affect the performance of an algorithm. We added code to keep a count of the retries.
The code to implement swap with this additional code is shown in Figure 70. Rather than keeping a simple count of retries, we build a histogram of retries. Using an array of 16 elements, we update the array element corresponding to the number of retries. This gives additional information regarding the distribution of retries. To limit the size of the array, we add one to the last element if the number of retries is equal to or greater than 16. The function fetch&add follows the same approach.

**C Code for Emulating Atomic Swap while Keeping a Count of Retries**

```c
unsigned char
swap(unsigned char *p, unsigned char v)
{
    unsigned char ret;
    int tries;
    tries=0;
    do {
        ret = ldstub(p,v);
        tries++;
    } while (ret == 0xFF);
    *p = v;
    if (tries > 16)
        tries = 16;
    Attempts[tries-1]++;
    return(ret);
}
```

**Figure 70. Swap with Additional Code for a Histogram of Retries**

We measured the cost of building the histogram as well as writing the swap in C language and the ldstub as a separate C function. According to our measurements the cost of building the histogram is approximately 0.2 μs per call. The cost of writing the swap in C and ldstub as a separate function is approximately 0.4 μs compared with a swap written in assembly language encapsulating the ldstub instruction within the swap function. Since these costs are nominal, we thought it was appropriate to keep the code in C with ldstub as a separate function and also the code for recording the retry statistics.
5.2.4 Effect of the VME Arbitration

Before collecting the performance measurements we experimented with the system to fine tune it. The VME arbiter is enabled on one circuit pack known as slot-1 device. This resides in the first slot of the VME. To provide fair access to all requesting processors, VME round robin arbitration may be used in a system with up to four processors. However, single level arbitration along with the fair requester is the only option in a system with 16 processors. When the fair requester option is enabled, each processor monitors the arbitration bus and requests the bus a second time only after there is no pending request on the bus since it acquired the bus last time. This provides a fair access to all the masters requesting the bus. The time it takes to acquire the bus depends on the physical distance of the requester from the slot-1 device. This is due to the fact that the Bus Grant is daisy chained and passes through all the requesters between the slot-1 device and the one to which the bus is granted. The slot-1 device has a further advantage over the others in addition to what can be attributed to the physical distance of a board from the slot-1 device.

As we discuss in Section 5.3.1, we will measure the performance with a 16 μs wait loop in the critical section as well as with a check that makes 20 accesses to the global shared memory. The test with 20 accesses to the global memory in the critical section revealed that the affect of the physical distance of a board from the slot-1 device distorts the measurements considerably. Some measure to neutralize this effect was necessary.

![VME Bus Diagram](image)

**Figure 71. VME with 17 Slots Equipped**

Slot-1 through slot-8 are equipped with the processor boards, slot-9 is equipped with the memory board, and slot-10 through slot-17 are also equipped with the processor boards.
in the system that we used for testing. The processors are numbered from 0 through 15 as depicted in Figure 71. All of our tests start with one processor and add one processor in each subsequent phase. The times measured in each phase are divided by the number of the processors participating in that phase; for more details see Section 3.1. The effect due to the physical distance of the board may be neutralized in a number of ways. We chose the following simple scheme for this purpose.

We start with a processor roughly at an average distance from slot-1. To be specific we chose \( P_8 \) as the first processor. The subsequent processors are added alternately from each side of \( P_8 \). To be specific, \( P_7 \) will be added next, \( P_9 \) will be added after that and so on and so forth. This has an averaging effect. Some zigzag is still expected but an overall shift due to arbitration is not expected. Also, due to the slot-1 device having some advantage over the others, its contribution would be somewhat less than any other processor. We verified these assertions in preliminary testing prior to starting the actual tests.

5.3 Performance Measurements

Before measuring the performance, we would like to describe what we would like to measure and how we are going to measure it.

5.3.1 Factors Affecting the Performance

A process is generally executing in its non critical section. When it needs to access a shared resource, it first needs to acquire a lock associated with the resource. To acquire the lock, it executes the entrance code; see the abstraction in Figure 3. As a result, it acquires the lock if the lock is free, or busy waits otherwise. When it acquires the lock, it enters the critical section, performs the operation on the shared resource, and exits. The activities of a process as it acquires and releases a lock may be modeled as shown in Figure 72. The entrance code may be divided conceptually into three parts. The first part checks whether the lock is free and may also indicate the presence of the process waiting for the lock. The second part consists of the code that busy waits by spinning on a variable. The third part consists of the code that executes just prior to entering the critical section. The exit code may also have a short loop to synchronize with a processor that may have started but not yet completed the first part of its entrance code.
The first and the third parts of the entrance code are expected to generate a fixed number of transactions. It is the second part of the entrance code, the code that busy waits, that can generate large amounts of traffic. If this traffic uses the same bus or the network over which a process in its critical section is performing its useful work, it may severely impede the progress of the later. This effect is exactly what we would like to measure.

![Figure 72. A Model for Performance Measurements](image)

The extent to which the performance of the process executing in its critical section in a given system is affected depends on the following factors:

1. the manner in which processes busy wait,

2. the number of processes that concurrently busy wait in their entrance code, and

3. the extent to which the process in its critical section utilizes the bus or the network.

The manner in which a process busy waits depends on the algorithm. Algorithms in which processes busy wait spinning on their local shared memories should have no effect on the performance of the process in its critical section. On the other hand, processes that spin on the global memory would have an adverse effect the magnitude of which will depend on the frequency of access. We have measured the performance for a number of algorithms; for details see Section 5.4. For each algorithm we vary the number of processors that concurrently busy wait in their entrance code. The extent to which the process executing in its critical section utilizes the bus is an important factor. One could think of two
extremes of a spectrum. On one extreme, the process makes no access to the global shared memory. On the other extreme it utilizes 100% capacity of the bus or the network. In the first case, one would expect the performance is not affected. This, however, is neither realistic nor completely true. It is not realistic because the reason a process acquires the lock is to access a shared resource. It is not completely true because even when a process makes no access to off-board memory while executing in its critical section, it must make one or more accesses to the global memory to release the lock. These accesses will be affected when there is contention on the bus. Therefore, there will be some affect on the performance. This would represent the minimum possible affect. The other extreme, where the process utilizes 100% capacity of the bus, may be unlikely but possible. Most of the real situations are likely to lie between these two extremes. We measure and present the performance for both of these cases.

5.3.2 Details of How the Performance is Measured

The code to measure the performance consists of a main loop and a sub loop within the main loop. The hardware timer is programmed to free run. When it free runs, it rolls over every 2 seconds. The main loop includes the sub loop to ensure that the timer does not roll over. The timer is started and its value $t_0$ noted before entering the sub loop. Within the sub loop, a process acquires the lock, executes its critical section, and releases the lock. When the sub loop terminates, the timer is read and its value $t_1$ noted. Subtracting $t_0$ from $t_1$ gives the gross time for one sub loop. The C code for the main loop and the sub loop from our test program is shown in Figure 73. The main loop continues for $10^5$ iterations through the sub loop for the mutual exclusion locks and $10^4$ iterations for the readers-writers locks. The gross times for all sub loops within a main loop are added and give the gross time for the main loop.

The gross time includes the sub loop overhead as well as few instruction that execute concurrently on all processors prior to when they start busy waiting in the entrance code. To improve the accuracy of measurements the loop overhead and the time for instructions that execute concurrently should be subtracted from the gross time. To do this, we execute the main loop again with a null function in the sub loop. The null function approximates the instructions that execute concurrently on all processors. The accumulated time

154
for all the sub loops within the main loop is subtracted from the gross time for the main
loop. This represent the actual time for all iterations in the main loop.

\[
\text{time} = 0; \\
\text{for (i=0; i<(LOOP\_COUNT/K); i++) { \\
\quad \text{*SUN2CE\_LIMIT0 = 0; /* start the timer */} \\
\quad \text{time0 = *SUN2CE\_TIMER0; /* t0 = current time */} \\
\quad \text{for (j=0; j<K; j++) { \\
\quad \quad \text{acquire (&Lock, (unsigned char)prcr);} \\
\quad \quad /* execute the code for the critical section */} \\
\quad \quad \text{release (&Lock, (unsigned char)prcr);} \\
\quad \} \\
\quad \text{time += *SUN2CE\_TIMER0-time0; /* t1 = current time */} \\
\} \\
\text{......}
\]

**Figure 7.3. C Code for the Main Loop to Measure the Performance**

The main loop is executed $N_p$ times where $N_p$ is the number of processors in the system. Each execution of main loop is referred to as a phase. We start with one processor during the first phase and add one processor during each subsequent phase. Therefore, $n_p$ processors participate during phase $n_p$. Each processor acquires the lock, executes its critical section, and releases the lock $K$ times in a phase. The system collectively acquires and releases the lock $n_p\times K$ times during the same period. The time reported by each processor to complete its $K$ iterations is also the time for the system to complete $n_p\times K$ iterations. Therefore, the system time to acquire and release the lock is the time reported by a processor for its $K$ iterations divided by $n_p\times K$. We refer to this as normalized per lock time.

When all processors that participate in a phase execute concurrently, the time reported by each must be approximately the same. We collect the times reported by all processors that participate in a phase. These times are found to be identical. Next, we calculate the normalized time for each processor. Even though the normalized times of all processors
are almost identical, we average them. We repeat each test three times to verify the data is repeatable; the data is found to be very repeatable. We present the averaged normalized per lock data from one out of the three tests for each algorithm in the next section.

During each phase, processors synchronize using a simple but efficient barrier. Next, the measurement code executing as a process determines if it is supposed to participate in the current phase. The processors that are not supposed to participate skip over the main loop and directly go to the barrier at the end of the phase. Others execute the main loop and then go to the barrier at the end of the phase. All processors synchronize at the barrier. Next, they output the data for the just completed phase and go back to the barrier at the beginning of the phase.

The measurement code executes as a VxWorks process. Disabling the interrupts is not necessary since no other higher priority processes are executing. However, as a precaution we still disabled the interrupts during the execution of the main loop.

5.4 Measurement Results

In this section we present the measurement results for the mutual exclusion and the readers-writers locks.

5.4.1 Mutual Exclusion Locks

We have measured the performance of Bit Test&Set lock, Anderson's lock, and Graunke & Thakkar's lock, and three of our own mutual exclusion locks presented in CHAPTER 2. We refer to our lock as pE-1, pE-2 and pE-3. Anderson and Graunke & Thakkar's locks rely on coherent caches to eliminate contention [8][32]. Our purpose to include them here is to show that they do not eliminate contention in a system that does not support coherent cache. Algorithms μE-1 and μE-2 extend Anderson and Graunke & Thakkar's locks to eliminate contention on machines that may not support coherent caches. Algorithm μE-3 is based on a linked list and is similar to MCS lock of Mellor-Crummey and Scott [73]. However, it maintains the FIFO behavior using swap as the only atomic primitive while MCS lock requires compare&swap to guarantee the FIFO behavior.
We use a 16 µs loop in the critical section for the case where a processor executing in its
critical section makes no accesses to the off-board memory. Some authors have used an
empty critical section [73]. We use a 16 µs timing loop because we emulate atomic swap
and fetch&add. If a processor releases the lock right after it acquires the lock, it does not
give a chance to other processors to complete their swaps. After releasing the lock, it goes
back and attempts to acquire the lock. The result is a number of processors trying to con­
currently execute swap. This in turn causes many to retry ldstub before they succeed in
executing swap. We experimented with an empty critical section. There were almost no
retries up to three processors, and approximately 2.2% retries when the fourth processor
was added. From that point on, the number of retries steadily increased as more proces­
sors are added. When the number of retries is significant, it affects the performance in a
manner similar to when a processor spins on a global memory. If a processor briefly holds
the lock rather than releasing it immediately, the number of retries is minimized. We use
a 16 µs loop in the critical section. As a result, there are almost no retries while executing
swap or fetch&add in our mutual exclusion algorithms.

To generate traffic to the off-board memory, each processor adds one to its private count
and then adds one to a shared count. The private count for each processor and the shared
count are kept in the global shared memory. Next, the processor checks the shared count
against the sum of the private counts of all the processors. We refer to it as the mutual
exclusion check. The check results in 20 accesses to the global memory. These accesses
are close enough that we expect them to consume close to 100% bandwidth of the bus.

The performance of each algorithm is measured first with 16 µs loop and then with the
mutual exclusion check in the critical section. Averaged normalized per lock times in mi­
cro seconds for 16 µs loop in the critical section are presented in Table 2. While the times
with the mutual exclusion check in the critical section are presented in Table 3.

The data for various algorithms is also plotted as a function of number of processors. The
plot with the 16 µs loop in the critical section is shown in Figure 74. The plot with the
mutual exclusion check in the critical section is shown in Figure 75. The 16 µs loop time
is included in the data presented in the tables as well as the plots. It may be subtracted
from the corresponding data to get the time it takes just to acquire and release the lock.
<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Bit Test&amp;Set</th>
<th>Anderson</th>
<th>Graunke&amp;Thakkar</th>
<th>µE-1</th>
<th>µE-2</th>
<th>µE-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>35.243</td>
<td>32.986</td>
<td>32.959</td>
<td>23.980</td>
<td>23.318</td>
<td>21.673</td>
</tr>
<tr>
<td>4</td>
<td>42.005</td>
<td>36.798</td>
<td>34.990</td>
<td>23.900</td>
<td>23.265</td>
<td>21.601</td>
</tr>
<tr>
<td>5</td>
<td>45.621</td>
<td>40.584</td>
<td>42.920</td>
<td>24.042</td>
<td>23.374</td>
<td>21.771</td>
</tr>
<tr>
<td>6</td>
<td>52.732</td>
<td>44.680</td>
<td>47.203</td>
<td>24.097</td>
<td>23.413</td>
<td>21.764</td>
</tr>
<tr>
<td>7</td>
<td>61.055</td>
<td>53.661</td>
<td>57.436</td>
<td>24.185</td>
<td>23.522</td>
<td>21.886</td>
</tr>
<tr>
<td>8</td>
<td>61.932</td>
<td>60.013</td>
<td>60.246</td>
<td>24.162</td>
<td>23.488</td>
<td>21.834</td>
</tr>
<tr>
<td>9</td>
<td>68.666</td>
<td>62.887</td>
<td>62.402</td>
<td>24.269</td>
<td>23.569</td>
<td>21.937</td>
</tr>
<tr>
<td>10</td>
<td>72.241</td>
<td>68.257</td>
<td>67.042</td>
<td>24.151</td>
<td>23.472</td>
<td>21.825</td>
</tr>
<tr>
<td>11</td>
<td>78.417</td>
<td>69.708</td>
<td>77.494</td>
<td>24.314</td>
<td>23.631</td>
<td>21.982</td>
</tr>
<tr>
<td>12</td>
<td>81.419</td>
<td>74.267</td>
<td>80.022</td>
<td>24.196</td>
<td>23.517</td>
<td>21.878</td>
</tr>
<tr>
<td>13</td>
<td>87.831</td>
<td>83.833</td>
<td>86.876</td>
<td>24.288</td>
<td>23.605</td>
<td>21.969</td>
</tr>
<tr>
<td>14</td>
<td>90.313</td>
<td>84.025</td>
<td>89.343</td>
<td>24.191</td>
<td>23.508</td>
<td>21.883</td>
</tr>
<tr>
<td>15</td>
<td>100.278</td>
<td>94.604</td>
<td>99.019</td>
<td>24.280</td>
<td>23.609</td>
<td>21.978</td>
</tr>
<tr>
<td>16</td>
<td>99.098</td>
<td>95.002</td>
<td>102.379</td>
<td>24.178</td>
<td>23.496</td>
<td>21.869</td>
</tr>
</tbody>
</table>

Table 2. Normalized Per Lock Time (µs) with 16 µs Loop in CS

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Bit Test&amp;Set</th>
<th>Anderson</th>
<th>Graunke&amp;Thakkar</th>
<th>µE-1</th>
<th>µE-2</th>
<th>µE-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>127.352</td>
<td>111.430</td>
<td>110.628</td>
<td>41.072</td>
<td>49.328</td>
<td>34.684</td>
</tr>
<tr>
<td>3</td>
<td>200.820</td>
<td>173.511</td>
<td>172.340</td>
<td>41.978</td>
<td>50.525</td>
<td>34.978</td>
</tr>
<tr>
<td>4</td>
<td>265.005</td>
<td>222.989</td>
<td>221.577</td>
<td>40.986</td>
<td>49.067</td>
<td>34.165</td>
</tr>
<tr>
<td>5</td>
<td>345.600</td>
<td>290.918</td>
<td>289.628</td>
<td>41.929</td>
<td>50.449</td>
<td>34.962</td>
</tr>
<tr>
<td>6</td>
<td>401.930</td>
<td>334.579</td>
<td>333.381</td>
<td>41.379</td>
<td>49.499</td>
<td>34.629</td>
</tr>
<tr>
<td>7</td>
<td>490.801</td>
<td>406.964</td>
<td>405.701</td>
<td>42.266</td>
<td>50.746</td>
<td>35.293</td>
</tr>
<tr>
<td>8</td>
<td>542.368</td>
<td>446.799</td>
<td>445.393</td>
<td>41.602</td>
<td>49.589</td>
<td>34.821</td>
</tr>
<tr>
<td>9</td>
<td>628.177</td>
<td>523.924</td>
<td>522.560</td>
<td>42.466</td>
<td>50.906</td>
<td>35.549</td>
</tr>
<tr>
<td>10</td>
<td>674.961</td>
<td>559.401</td>
<td>557.630</td>
<td>41.579</td>
<td>49.628</td>
<td>34.826</td>
</tr>
<tr>
<td>11</td>
<td>759.502</td>
<td>645.662</td>
<td>641.016</td>
<td>42.651</td>
<td>51.069</td>
<td>35.670</td>
</tr>
<tr>
<td>12</td>
<td>801.410</td>
<td>671.733</td>
<td>672.416</td>
<td>41.628</td>
<td>49.635</td>
<td>34.926</td>
</tr>
<tr>
<td>13</td>
<td>884.000</td>
<td>757.680</td>
<td>754.339</td>
<td>42.333</td>
<td>50.947</td>
<td>35.579</td>
</tr>
<tr>
<td>14</td>
<td>916.721</td>
<td>783.559</td>
<td>781.781</td>
<td>41.606</td>
<td>49.630</td>
<td>34.847</td>
</tr>
<tr>
<td>15</td>
<td>1029.568</td>
<td>867.555</td>
<td>866.003</td>
<td>42.487</td>
<td>50.915</td>
<td>35.541</td>
</tr>
<tr>
<td>16</td>
<td>1096.417</td>
<td>892.287</td>
<td>895.900</td>
<td>41.563</td>
<td>49.574</td>
<td>34.785</td>
</tr>
</tbody>
</table>

Table 3. Normalized Per Lock Time (µs) with Mutual Exclusion Check in CS

158
Figure 74. Performance with 16 μs Loop in the Critical Section

Figure 75. Performance with Mutual Exclusion Check in the Critical Section
The algorithms perform as expected. The algorithms that spin on variables in the global memory perform poorly as the number of processors that wait for the lock increases. In the absence of a coherent cache Anderson and Graunke & Thakkar's lock perform almost as poorly as the simple Bit Test&Set algorithm. The small difference is due to the fact that in Bit Test&Set, processors execute read-modify-write cycles when they are busy waiting whereas in Anderson and Graunke & Thakkar's locks they execute simple reads. In the absence of any contention, the Bit Test&Set performs the best as it is the simplest of all. The other two take slightly more time as they execute a few more instructions to acquire/release the lock. As the contention increases, the performance of all three quickly degrades. The degradation is linear with the number of waiting processors for a given activity in the critical section.

The extent to which a processor utilizes the bus after acquiring the lock is an important factor in determining how its performance is affected by processors busy waiting on the global memory. For a fixed number of processors waiting in the entrance code, the performance is expected to degrade linearly with the number of access to the global memory by the processor holding the lock.

The algorithms in which processors busy wait by spinning on their local memories do not affect the performance of the processor holding the lock. Since they generate a constant number of transactions over the bus for checking the lock and queuing themselves but do not generate bus traffic while waiting, they are not expected to affect the performance. The measurements verify it. The number of accesses a processor makes to the global memory for acquiring and releasing the lock may differ based on the contention for the lock. For µE-1 and µE-3, the processor takes extra steps to acquire and release the lock when the lock is free compared to when the lock is busy. This is reflected in the measurements as they take more time to acquire/release the lock when the lock is free. The algorithms µE-2 is at an advantage as it takes fewer steps when the lock is free compared to when the lock is busy. This is likewise reflected in the measurements. If the lock is heavily contended, i.e., there are always one or more processors waiting for the lock before it is released, the linked list based lock µE-3 provides the best performance. Except for the slight zigzag due to VME arbitration as explained in Section 5.2.4, the normalized time to
acquire and release the lock stays fairly constant. This is expected in our measurements since during the time a processor is holding the lock, there is only one processor that checks and queues itself. This is the processor that just released the lock. If the number of processors that request the lock during the time a processor is holding the lock increases with the increase in the number of processors, the performance may be slightly impacted. However, that is not the case in our setup. Therefore, as verified by the measurements, the performance is not affected when more processors are added.

5.4.2 Readers-Writers Locks

We have developed a number of fair readers-writers locks. We have measured the performance of all of them except one. The performance of RW.DL-3 could not be measured because of the intermittent hardware failures. The swap from one processor board to the local shared memory of another processor board fails intermittently. In particular, it fails if the processor on the board whose local shared memory is being accessed is also doing a local swap. For an algorithm that may tolerate such failures without breaking down, the performance could be measured. However, for an algorithms, e.g., RW.DL-3, that could not tolerate such failures, measuring the performance on this setup became impossible.

All the algorithms that we have developed and presented in CHAPTER 3 are fair. The first of these is the simplest but relies on a coherent cache to eliminate contention. Since our setup does not have a coherent cache, the performance of this algorithm will be close to the performance of any algorithm that busy waits spinning on the global memory. We will use this in our measurements as a reference.

We measure the performance for the same two cases as we did for the mutual exclusion locks. In one case we use 16 μs wait loop in the critical section. In the other case we use the mutual exclusion check. The check, however, is modified. The writers execute the same function as is executed in the critical section of the mutual exclusion locks. The readers use a modified function. In the modified function, a reader checks for mutual exclusion by checking the sum of the counts for all the processors against the shared sum. However, it does not update its own count or the shared count. Since multiple readers may concurrently execute in the critical section, readers can only read but not update any shared data. However, to make the two tests as equal as possible in terms of the number
of accesses to the global memory, the modified check reads the private counts of four additional processors.

The readers-writer algorithms have an additional parameter, the number of times the lock is acquired for reading versus writing. This needs to be taken into account during the measurements. We run both tests for each algorithm for 9 different combinations of reader/writer locks. The combinations consist of \( n \) readers/\( 16-n \) writers where \( n \) takes on value 0, 1, 2, 4, 8, 12, 14, 15, and 16.

Each test executes \( 10^4 \) iterations. In each iteration a processor acquires the lock, executes its critical section, and releases the lock. Each test is run three times to ensure the repeatability. Averaged normalized per lock times for each algorithm for all readers/writers combinations stated above are presented in the following tables.

Next we plot the per lock times for all algorithms on the same graph. The graphs are plotted for three combinations of readers and writers. We chose 4 readers and 12 writers, 8 readers and 8 writers, and 12 readers and 4 writers as the three combinations. For each combination a plot is given for 16 µs loop in the critical section and another plot is shown for the mutual exclusion check in the critical section.

The algorithms perform as expected. The one that relies on coherent cache is the simplest among this group. In a system that supports coherent cache, we expect it to outperform the others. However, in a system that does not support a coherent cache it will perform poorly as it spins on the global variable. Since the setup that we have used for the performance measurements does not support coherent cache, the performance of this algorithm quickly degrades as we increase the number of processors in the test. We use the performance of this algorithm as a reference. The performance of the rest of the algorithms does not degrade as a function of the number of processors in the system. Among these, the RW.DL-1 performs relatively poorly compared to others. This is expected since it protects the doubly linked list using a mutual exclusion lock and therefore provides the least concurrency. The other three provide the same level of concurrency. Their performance is expected to be close to each other. The difference in their performance is attributable to the difference in their overhead.
<table>
<thead>
<tr>
<th># of Processors</th>
<th>Number of Read Locks vs. Write Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0:16</td>
</tr>
<tr>
<td>2</td>
<td>43.340</td>
</tr>
<tr>
<td>4</td>
<td>72.473</td>
</tr>
<tr>
<td>5</td>
<td>88.747</td>
</tr>
<tr>
<td>6</td>
<td>100.251</td>
</tr>
<tr>
<td>7</td>
<td>121.740</td>
</tr>
<tr>
<td>8</td>
<td>135.950</td>
</tr>
<tr>
<td>9</td>
<td>151.692</td>
</tr>
<tr>
<td>10</td>
<td>187.263</td>
</tr>
<tr>
<td>11</td>
<td>206.411</td>
</tr>
<tr>
<td>13</td>
<td>273.656</td>
</tr>
</tbody>
</table>

Table 4. Per Lock Times (μs) for 16 μs Loop in CS for RW.AR-1

<table>
<thead>
<tr>
<th># of Processors</th>
<th>Number of Read Locks vs. Write Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0:16</td>
</tr>
<tr>
<td>2</td>
<td>127.563</td>
</tr>
<tr>
<td>3</td>
<td>201.742</td>
</tr>
<tr>
<td>4</td>
<td>261.339</td>
</tr>
<tr>
<td>5</td>
<td>338.550</td>
</tr>
<tr>
<td>6</td>
<td>390.195</td>
</tr>
<tr>
<td>7</td>
<td>477.132</td>
</tr>
<tr>
<td>8</td>
<td>521.481</td>
</tr>
<tr>
<td>9</td>
<td>611.847</td>
</tr>
<tr>
<td>10</td>
<td>653.188</td>
</tr>
<tr>
<td>11</td>
<td>753.912</td>
</tr>
<tr>
<td>12</td>
<td>790.246</td>
</tr>
<tr>
<td>13</td>
<td>882.948</td>
</tr>
<tr>
<td>14</td>
<td>915.824</td>
</tr>
<tr>
<td>15</td>
<td>1014.846</td>
</tr>
<tr>
<td>16</td>
<td>1045.787</td>
</tr>
</tbody>
</table>

Table 5. Per Lock Times (μs) for Mutual Exclusion Check in CS for RW.AR-1
<table>
<thead>
<tr>
<th># of Processors</th>
<th>Number of Read Locks vs. Write Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0:16</td>
</tr>
<tr>
<td>1</td>
<td>28.810</td>
</tr>
<tr>
<td>3</td>
<td>26.814</td>
</tr>
<tr>
<td>5</td>
<td>26.841</td>
</tr>
<tr>
<td>7</td>
<td>27.057</td>
</tr>
<tr>
<td>9</td>
<td>27.153</td>
</tr>
<tr>
<td>10</td>
<td>27.042</td>
</tr>
<tr>
<td>11</td>
<td>27.227</td>
</tr>
</tbody>
</table>

Table 6. Per Lock Times (μs) for 16 μs Loop in CS for RW AR-2

<table>
<thead>
<tr>
<th># of Processors</th>
<th>Number of Read Locks vs. Write Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0:16</td>
</tr>
<tr>
<td>1</td>
<td>28.229</td>
</tr>
<tr>
<td>2</td>
<td>43.805</td>
</tr>
<tr>
<td>3</td>
<td>44.732</td>
</tr>
<tr>
<td>4</td>
<td>43.718</td>
</tr>
<tr>
<td>5</td>
<td>44.667</td>
</tr>
<tr>
<td>6</td>
<td>44.173</td>
</tr>
<tr>
<td>7</td>
<td>45.068</td>
</tr>
<tr>
<td>8</td>
<td>44.423</td>
</tr>
<tr>
<td>9</td>
<td>45.264</td>
</tr>
<tr>
<td>10</td>
<td>44.406</td>
</tr>
<tr>
<td>11</td>
<td>45.482</td>
</tr>
<tr>
<td>12</td>
<td>44.457</td>
</tr>
<tr>
<td>13</td>
<td>45.335</td>
</tr>
<tr>
<td>14</td>
<td>44.392</td>
</tr>
<tr>
<td>15</td>
<td>45.290</td>
</tr>
<tr>
<td>16</td>
<td>44.358</td>
</tr>
</tbody>
</table>

Table 7. Per Lock Times (μs) for Mutual Exclusion Check in CS for RW AR-2
<table>
<thead>
<tr>
<th># of Processors</th>
<th>Number of Read Locks vs. Write Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:16</td>
<td>37.856 38.734 39.564 41.217 44.516 47.838 49.463 50.285 51.023</td>
</tr>
<tr>
<td>1:15</td>
<td>41.961 42.479 43.270 44.172 45.821 47.468 48.264 48.662 48.566</td>
</tr>
<tr>
<td>2:14</td>
<td>42.509 42.459 43.226 45.018 48.804 52.609 54.490 55.402 57.761</td>
</tr>
<tr>
<td>4:12</td>
<td>41.786 41.528 41.505 41.682 42.022 42.328 42.479 42.579 44.091</td>
</tr>
<tr>
<td>8:8</td>
<td>42.556 42.269 41.940 41.290 40.016 38.733 38.076 37.769 37.445</td>
</tr>
<tr>
<td>12:4</td>
<td>41.938 41.410 40.961 40.073 38.303 36.548 35.661 35.199 34.842</td>
</tr>
<tr>
<td>14:2</td>
<td>42.681 42.133 41.716 40.888 39.217 37.549 36.713 36.293 35.997</td>
</tr>
<tr>
<td>15:1</td>
<td>42.039 41.441 40.957 40.014 38.147 36.288 35.344 34.889 34.547</td>
</tr>
<tr>
<td>16:0</td>
<td>42.792 42.256 41.731 40.856 39.121 37.335 36.567 36.009 35.744</td>
</tr>
<tr>
<td>2</td>
<td>41.156 45.440 50.630 60.433 79.349 98.278 107.726 112.453 116.587</td>
</tr>
<tr>
<td>3</td>
<td>41.866 45.802 49.952 58.996 76.436 94.865 103.546 108.185 115.349</td>
</tr>
<tr>
<td>4</td>
<td>41.156 44.705 48.643 56.940 74.114 90.431 97.817 102.423 109.487</td>
</tr>
<tr>
<td>5</td>
<td>41.921 45.530 49.349 58.090 73.881 88.219 97.787 102.122 110.291</td>
</tr>
<tr>
<td>6</td>
<td>41.626 44.911 48.402 55.572 70.191 82.971 91.500 96.044 102.682</td>
</tr>
<tr>
<td>7</td>
<td>42.328 45.621 49.327 56.050 69.501 83.943 91.175 95.753 101.795</td>
</tr>
<tr>
<td>8</td>
<td>41.894 44.967 48.272 54.475 67.236 79.845 86.050 89.609 93.769</td>
</tr>
<tr>
<td>9</td>
<td>42.578 45.815 49.029 55.501 68.853 82.123 87.900 91.117 94.042</td>
</tr>
<tr>
<td>10</td>
<td>41.820 44.729 47.931 54.282 66.160 79.090 83.680 86.026 88.010</td>
</tr>
<tr>
<td>11</td>
<td>42.741 45.788 49.115 55.557 68.972 82.331 86.944 89.194 90.980</td>
</tr>
<tr>
<td>12</td>
<td>41.953 44.793 47.949 53.857 66.268 79.183 83.562 85.336 87.299</td>
</tr>
<tr>
<td>13</td>
<td>42.663 45.604 49.043 55.305 67.869 82.012 86.536 88.428 90.642</td>
</tr>
<tr>
<td>14</td>
<td>41.892 44.690 47.861 53.754 65.768 79.231 83.307 84.874 87.105</td>
</tr>
<tr>
<td>15</td>
<td>42.583 45.527 48.883 54.997 67.975 81.859 86.415 88.095 90.568</td>
</tr>
<tr>
<td>16</td>
<td>41.823 44.621 47.930 53.440 64.950 78.696 82.873 84.527 86.779</td>
</tr>
</tbody>
</table>

Table 8. Per Lock Times (μs) for 16 μ Loop in CS for RW.SL-1

<table>
<thead>
<tr>
<th># of Processors</th>
<th>Number of Read Locks vs. Write Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:16</td>
<td>37.354 38.251 39.150 40.841 44.323 47.822 49.558 50.403 51.255</td>
</tr>
<tr>
<td>1:15</td>
<td>41.314 45.440 50.630 60.433 79.349 98.278 107.726 112.453 116.587</td>
</tr>
<tr>
<td>2:14</td>
<td>41.866 45.802 49.952 58.996 76.436 94.865 103.546 108.185 115.349</td>
</tr>
<tr>
<td>4:12</td>
<td>41.156 44.705 48.643 56.940 74.114 90.431 97.817 102.423 109.487</td>
</tr>
<tr>
<td>8:8</td>
<td>41.921 45.530 49.349 58.090 73.881 88.219 97.787 102.122 110.291</td>
</tr>
<tr>
<td>12:4</td>
<td>41.626 44.911 48.402 55.572 70.191 82.971 91.500 96.044 102.682</td>
</tr>
<tr>
<td>14:2</td>
<td>42.328 45.621 49.327 56.050 69.501 83.943 91.175 95.753 101.795</td>
</tr>
<tr>
<td>15:1</td>
<td>41.894 44.967 48.272 54.475 67.236 79.845 86.050 89.609 93.769</td>
</tr>
<tr>
<td>16:0</td>
<td>42.578 45.815 49.029 55.501 68.853 82.123 87.900 91.117 94.042</td>
</tr>
</tbody>
</table>

Table 9. Per Lock Times (μs) for Mutual Exclusion Check in CS for RW.SL-1
<table>
<thead>
<tr>
<th># of Processors</th>
<th>0:16</th>
<th>1:15</th>
<th>2:14</th>
<th>4:12</th>
<th>8:8</th>
<th>12:4</th>
<th>14:2</th>
<th>15:1</th>
<th>16:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>48.669</td>
<td>49.208</td>
<td>49.746</td>
<td>50.798</td>
<td>52.920</td>
<td>55.033</td>
<td>56.084</td>
<td>56.623</td>
<td>57.157</td>
</tr>
<tr>
<td>2</td>
<td>41.010</td>
<td>41.374</td>
<td>41.700</td>
<td>42.340</td>
<td>43.640</td>
<td>44.952</td>
<td>45.600</td>
<td>45.918</td>
<td>46.192</td>
</tr>
<tr>
<td>3</td>
<td>41.813</td>
<td>42.166</td>
<td>42.388</td>
<td>42.757</td>
<td>43.495</td>
<td>44.218</td>
<td>44.586</td>
<td>44.772</td>
<td>44.604</td>
</tr>
<tr>
<td>4</td>
<td>41.165</td>
<td>41.453</td>
<td>41.603</td>
<td>41.832</td>
<td>42.432</td>
<td>42.980</td>
<td>43.203</td>
<td>43.441</td>
<td>43.023</td>
</tr>
<tr>
<td>5</td>
<td>41.899</td>
<td>42.324</td>
<td>42.478</td>
<td>42.728</td>
<td>43.176</td>
<td>44.041</td>
<td>44.036</td>
<td>44.207</td>
<td>43.745</td>
</tr>
<tr>
<td>6</td>
<td>41.507</td>
<td>42.034</td>
<td>42.383</td>
<td>42.632</td>
<td>42.594</td>
<td>43.746</td>
<td>43.396</td>
<td>43.803</td>
<td>42.789</td>
</tr>
<tr>
<td>7</td>
<td>42.238</td>
<td>42.782</td>
<td>43.128</td>
<td>43.299</td>
<td>44.361</td>
<td>44.913</td>
<td>45.353</td>
<td>45.534</td>
<td>43.741</td>
</tr>
<tr>
<td>8</td>
<td>41.775</td>
<td>42.315</td>
<td>42.788</td>
<td>43.551</td>
<td>43.304</td>
<td>43.781</td>
<td>43.979</td>
<td>44.089</td>
<td>42.426</td>
</tr>
<tr>
<td>9</td>
<td>42.478</td>
<td>42.952</td>
<td>43.390</td>
<td>43.658</td>
<td>43.981</td>
<td>43.956</td>
<td>45.081</td>
<td>44.735</td>
<td>43.827</td>
</tr>
<tr>
<td>10</td>
<td>41.687</td>
<td>42.333</td>
<td>42.560</td>
<td>42.786</td>
<td>43.105</td>
<td>43.132</td>
<td>43.612</td>
<td>43.438</td>
<td>42.400</td>
</tr>
<tr>
<td>11</td>
<td>42.596</td>
<td>43.152</td>
<td>43.563</td>
<td>43.977</td>
<td>44.343</td>
<td>44.986</td>
<td>44.688</td>
<td>45.209</td>
<td>44.540</td>
</tr>
<tr>
<td>12</td>
<td>41.807</td>
<td>42.328</td>
<td>42.300</td>
<td>42.368</td>
<td>42.850</td>
<td>43.492</td>
<td>43.424</td>
<td>43.255</td>
<td>43.092</td>
</tr>
<tr>
<td>13</td>
<td>42.477</td>
<td>43.123</td>
<td>43.134</td>
<td>43.850</td>
<td>43.933</td>
<td>44.646</td>
<td>44.988</td>
<td>45.110</td>
<td>43.972</td>
</tr>
<tr>
<td>14</td>
<td>41.749</td>
<td>42.238</td>
<td>42.380</td>
<td>42.679</td>
<td>43.377</td>
<td>43.191</td>
<td>43.491</td>
<td>43.522</td>
<td>42.957</td>
</tr>
<tr>
<td>15</td>
<td>42.457</td>
<td>42.968</td>
<td>43.546</td>
<td>43.695</td>
<td>44.714</td>
<td>44.712</td>
<td>45.762</td>
<td>46.790</td>
<td>44.642</td>
</tr>
<tr>
<td>16</td>
<td>41.709</td>
<td>42.204</td>
<td>42.507</td>
<td>43.317</td>
<td>44.041</td>
<td>44.612</td>
<td>44.399</td>
<td>46.414</td>
<td>43.541</td>
</tr>
</tbody>
</table>

Table 10. Per Lock Times (us) for 16 us Loop in CS for RW.DL-1

<table>
<thead>
<tr>
<th># of Processors</th>
<th>0:16</th>
<th>1:15</th>
<th>2:14</th>
<th>4:12</th>
<th>8:8</th>
<th>12:4</th>
<th>14:2</th>
<th>15:1</th>
<th>16:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>48.098</td>
<td>48.683</td>
<td>49.256</td>
<td>50.402</td>
<td>52.698</td>
<td>54.987</td>
<td>56.137</td>
<td>56.720</td>
<td>57.284</td>
</tr>
<tr>
<td>2</td>
<td>114.208</td>
<td>114.400</td>
<td>115.047</td>
<td>116.547</td>
<td>119.608</td>
<td>122.567</td>
<td>124.123</td>
<td>124.828</td>
<td>120.907</td>
</tr>
<tr>
<td>3</td>
<td>116.302</td>
<td>115.989</td>
<td>115.896</td>
<td>115.664</td>
<td>115.194</td>
<td>114.663</td>
<td>114.460</td>
<td>114.288</td>
<td>113.095</td>
</tr>
<tr>
<td>4</td>
<td>111.982</td>
<td>111.627</td>
<td>111.109</td>
<td>109.861</td>
<td>107.538</td>
<td>104.906</td>
<td>103.498</td>
<td>102.557</td>
<td>101.966</td>
</tr>
<tr>
<td>5</td>
<td>116.076</td>
<td>115.487</td>
<td>114.746</td>
<td>113.074</td>
<td>110.187</td>
<td>106.703</td>
<td>104.088</td>
<td>103.188</td>
<td>101.344</td>
</tr>
<tr>
<td>6</td>
<td>112.305</td>
<td>111.660</td>
<td>110.666</td>
<td>108.651</td>
<td>104.057</td>
<td>100.227</td>
<td>97.930</td>
<td>97.474</td>
<td>95.430</td>
</tr>
<tr>
<td>8</td>
<td>112.248</td>
<td>111.627</td>
<td>110.681</td>
<td>108.615</td>
<td>104.164</td>
<td>99.974</td>
<td>97.572</td>
<td>97.615</td>
<td>95.565</td>
</tr>
<tr>
<td>10</td>
<td>112.364</td>
<td>111.817</td>
<td>110.773</td>
<td>108.608</td>
<td>104.582</td>
<td>102.334</td>
<td>100.149</td>
<td>97.679</td>
<td>95.361</td>
</tr>
<tr>
<td>12</td>
<td>112.499</td>
<td>111.728</td>
<td>110.729</td>
<td>109.112</td>
<td>105.668</td>
<td>102.923</td>
<td>100.707</td>
<td>99.302</td>
<td>95.320</td>
</tr>
<tr>
<td>14</td>
<td>112.317</td>
<td>111.878</td>
<td>110.839</td>
<td>109.966</td>
<td>105.890</td>
<td>102.993</td>
<td>101.336</td>
<td>100.126</td>
<td>95.687</td>
</tr>
<tr>
<td>16</td>
<td>112.398</td>
<td>111.867</td>
<td>111.310</td>
<td>109.963</td>
<td>107.263</td>
<td>104.245</td>
<td>101.994</td>
<td>100.463</td>
<td>95.495</td>
</tr>
</tbody>
</table>

Table 11. Per Lock Times (us) for Mutual Exclusion Check in CS for RW.DL-1
<table>
<thead>
<tr>
<th># of Processors</th>
<th>0:16</th>
<th>1:15</th>
<th>2:14</th>
<th>4:12</th>
<th>8:8</th>
<th>12:4</th>
<th>14:2</th>
<th>15:1</th>
<th>16:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30.574</td>
<td>32.137</td>
<td>33.684</td>
<td>36.768</td>
<td>42.964</td>
<td>49.150</td>
<td>52.258</td>
<td>53.813</td>
<td>55.357</td>
</tr>
<tr>
<td>2</td>
<td>21.253</td>
<td>22.932</td>
<td>25.107</td>
<td>29.417</td>
<td>38.717</td>
<td>47.979</td>
<td>52.893</td>
<td>55.251</td>
<td>59.140</td>
</tr>
<tr>
<td>3</td>
<td>21.587</td>
<td>22.554</td>
<td>24.895</td>
<td>28.887</td>
<td>37.400</td>
<td>45.930</td>
<td>50.283</td>
<td>52.437</td>
<td>56.783</td>
</tr>
<tr>
<td>4</td>
<td>21.519</td>
<td>22.350</td>
<td>23.913</td>
<td>26.838</td>
<td>32.988</td>
<td>39.083</td>
<td>42.148</td>
<td>43.683</td>
<td>45.920</td>
</tr>
<tr>
<td>6</td>
<td>21.716</td>
<td>22.377</td>
<td>23.223</td>
<td>24.555</td>
<td>27.893</td>
<td>30.515</td>
<td>32.813</td>
<td>33.541</td>
<td>33.598</td>
</tr>
<tr>
<td>15</td>
<td>21.924</td>
<td>22.742</td>
<td>23.310</td>
<td>24.991</td>
<td>27.819</td>
<td>30.761</td>
<td>33.107</td>
<td>34.196</td>
<td>32.748</td>
</tr>
<tr>
<td>16</td>
<td>21.812</td>
<td>22.544</td>
<td>23.316</td>
<td>24.687</td>
<td>27.549</td>
<td>30.017</td>
<td>32.536</td>
<td>33.470</td>
<td>31.598</td>
</tr>
</tbody>
</table>

Table 12. Per Lock Times (µs) for 16 µs Loop in CS for RW.DL-2

<table>
<thead>
<tr>
<th># of Processors</th>
<th>0:16</th>
<th>1:15</th>
<th>2:14</th>
<th>4:12</th>
<th>8:8</th>
<th>12:4</th>
<th>14:2</th>
<th>15:1</th>
<th>16:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29.892</td>
<td>31.501</td>
<td>33.103</td>
<td>36.288</td>
<td>42.690</td>
<td>49.091</td>
<td>52.282</td>
<td>53.877</td>
<td>55.488</td>
</tr>
<tr>
<td>2</td>
<td>34.565</td>
<td>39.812</td>
<td>45.096</td>
<td>55.848</td>
<td>77.126</td>
<td>98.342</td>
<td>108.983</td>
<td>114.281</td>
<td>119.595</td>
</tr>
<tr>
<td>3</td>
<td>34.951</td>
<td>38.881</td>
<td>43.813</td>
<td>53.666</td>
<td>72.009</td>
<td>87.873</td>
<td>99.717</td>
<td>105.504</td>
<td>113.531</td>
</tr>
<tr>
<td>4</td>
<td>34.129</td>
<td>37.677</td>
<td>42.034</td>
<td>51.114</td>
<td>66.346</td>
<td>80.560</td>
<td>92.642</td>
<td>98.864</td>
<td>107.871</td>
</tr>
<tr>
<td>5</td>
<td>34.946</td>
<td>38.600</td>
<td>42.934</td>
<td>52.270</td>
<td>65.075</td>
<td>82.171</td>
<td>91.796</td>
<td>99.595</td>
<td>110.086</td>
</tr>
<tr>
<td>6</td>
<td>34.567</td>
<td>37.982</td>
<td>41.863</td>
<td>49.330</td>
<td>62.267</td>
<td>75.453</td>
<td>85.858</td>
<td>92.428</td>
<td>103.288</td>
</tr>
<tr>
<td>7</td>
<td>35.278</td>
<td>38.779</td>
<td>42.671</td>
<td>50.022</td>
<td>63.329</td>
<td>76.616</td>
<td>86.106</td>
<td>92.657</td>
<td>103.829</td>
</tr>
<tr>
<td>8</td>
<td>34.800</td>
<td>38.089</td>
<td>41.757</td>
<td>48.164</td>
<td>60.253</td>
<td>73.492</td>
<td>81.754</td>
<td>87.094</td>
<td>96.467</td>
</tr>
<tr>
<td>9</td>
<td>35.479</td>
<td>38.904</td>
<td>42.671</td>
<td>49.257</td>
<td>62.181</td>
<td>75.711</td>
<td>84.038</td>
<td>89.326</td>
<td>97.271</td>
</tr>
<tr>
<td>10</td>
<td>34.761</td>
<td>37.988</td>
<td>41.461</td>
<td>47.704</td>
<td>60.305</td>
<td>72.797</td>
<td>80.140</td>
<td>85.011</td>
<td>91.106</td>
</tr>
<tr>
<td>11</td>
<td>35.614</td>
<td>39.010</td>
<td>42.224</td>
<td>49.218</td>
<td>62.107</td>
<td>75.471</td>
<td>82.875</td>
<td>87.487</td>
<td>92.869</td>
</tr>
<tr>
<td>12</td>
<td>34.839</td>
<td>38.042</td>
<td>41.283</td>
<td>47.460</td>
<td>59.902</td>
<td>72.694</td>
<td>79.416</td>
<td>83.429</td>
<td>87.806</td>
</tr>
<tr>
<td>13</td>
<td>35.527</td>
<td>38.895</td>
<td>42.176</td>
<td>48.721</td>
<td>61.723</td>
<td>75.369</td>
<td>82.243</td>
<td>86.168</td>
<td>90.054</td>
</tr>
<tr>
<td>14</td>
<td>34.808</td>
<td>37.979</td>
<td>41.096</td>
<td>47.300</td>
<td>59.713</td>
<td>72.680</td>
<td>79.143</td>
<td>82.560</td>
<td>85.606</td>
</tr>
<tr>
<td>15</td>
<td>35.505</td>
<td>38.817</td>
<td>42.035</td>
<td>48.672</td>
<td>61.637</td>
<td>75.242</td>
<td>81.765</td>
<td>85.221</td>
<td>87.869</td>
</tr>
<tr>
<td>16</td>
<td>34.759</td>
<td>37.975</td>
<td>41.194</td>
<td>47.444</td>
<td>59.833</td>
<td>72.744</td>
<td>79.086</td>
<td>82.185</td>
<td>83.812</td>
</tr>
</tbody>
</table>

Table 13. Per Lock Times (µs) for Mutual Exclusion Check in CS for RW.DL-2

167
Since the readers can concurrently enter their critical sections, as the number of reader locks increases, one would expect the normalized per lock time to decrease. As we will see shortly this is true. However, there is another factor that needs to be considered. As the readers may acquire and release the lock concurrently, they are more likely to execute swap or fetch&add concurrently as compared to the mutual exclusion lock. As a result, swap or fetch&add may execute ldstub multiple times before it succeeds. This will have the effect of increasing the per lock time. The retries data\textsuperscript{4} shows that this phenomenon does occur. The number of retries increases as the ratio of reader to writer locks increases. The effect of retries is more when a processor acquires more reader locks compared to the writer locks. In case of some algorithms the effect is more pronounced when there are 2 to 5 processors participating in the test. When the number of processors increases beyond 5, the number of retries start decreasing. In other algorithms, the effect continues to increase as the number of processors increases. According to the data, the number of retries may vary anywhere from 5-30%. The retries tend to increase the per lock time. All the data presented in the tables as well as the graphs include this effect as there is no simple way to subtract it out.

To see the effect of the readers executing concurrently in the critical section, we have plotted a family of curves for two algorithms for various readers/writers combinations. We chose the array based simple algorithm RW.AR-1 and the doubly linked list based algorithm RW.DL-2. The first has the worst performance while the later has the best performance in the group. We plot per lock time for 0/16, 4/12, 8/8, 12/4, and 16/0 reader/writer lock combinations. The graph for the array based RW.AR-1 lock with 16 μs loop in the critical section is shown in Figure 82. The graph for the same lock with the mutual exclusion check in the critical section is shown in Figure 83. Both of these plots show that the slope of the curve decreases as the number of reader locks increases. Since in this lock the processors spin on the global memory, their wasteful cycles become useful cycles as the number of readers increases. The gain in performance is obvious from the two graphs. The per lock time for this algorithm is minimum when all the locks acquired are reader locks. It should be noted that the effect of readers concurrently executing their critical sections is slightly offset by the effect of retries; this is not obvious from the plots.

\textsuperscript{4} This data is not presented in the dissertation here.
Figure 76. Plots for 4 Readers/12 Writers with 16 $\mu$s Loop in CS

Figure 77. Plots for 4 Readers/12 Writers with Mutual Exclusion Check in CS

169
Figure 78. Plots for 8 Readers/8 Writers with 16 μs Loop in CS

Figure 79. Plots for 8 Readers/8 Writers with Mutual Exclusion Check in CS
Figure 80. Plots for 12 Readers/4 Writers with 16 μs Loop in CS

Figure 81. Plots for 12 Readers/4 Writers with Mutual Exclusion Check in CS
Figure 82. Algorithm RW.AR-1 with 16 μs Loop in CS

Figure 83. Algorithm RW.AR-1 with Mutual Exclusion Check in CS
Figure 84. Algorithm RW.DL-2 with 16 µs Loop in CS

Figure 85. Algorithm RW.DL-2 with Mutual Exclusion Check in CS
Similar graphs for the doubly linked list based algorithm, RW.DL-2 are shown in the next two plots. The graph with 16 µs loop in the critical section is shown in Figure 84. While the graph with mutual exclusion check in the critical section is shown in Figure 85. The performance of this lock decreases as the ratio of the reader to writer locks increases. There are two reasons for this behavior. First, the overhead of the reader lock is more compared to the writer lock. Second, the retries increase as the ratio of the reader to writer locks increases.

In case of the 16 µs loop in the critical section, the bus is initially very lightly used. Therefore, as we increase the number of processors, the per lock time for any reader/writer combination initially decreases. However, beyond 7-8 processors, the concurrent acquiring and releasing of the lock coupled with the effect of retries seem to saturate the bus. Consequently, the per lock time does not decrease any more. In fact it starts slightly increasing. In case of the mutual exclusion check in the critical section, the bus is already almost saturated. Therefore, we see some of the same effect as we see with the 16 µs loop in the critical section but the effect is not as much.

The same phenomenon takes place in the array based simple lock as well. However, every writer lock that is changed to a reader lock eliminates global spinning, the effect of which dominates the overall performance. Therefore, as the ratio of readers to writers is increased, accordingly the performance always improves.

5.5 Summary

We have measured and presented the performance of our algorithms in this chapter. All of the algorithms perform as expected and eliminate contention due to busy waiting by spinning on the local share memory. The absolute performance somewhat varies depending on the overhead to acquire and release the lock. In systems where the lock is heavily contended, the singly linked list based mutual exclusion algorithm µE-3 and the doubly linked list based readers-writers algorithm RW.DL-2 give the best absolute and relative performance. They also guarantee the FIFO behavior. Since they do not rely on coherent caches, they may be used in any shared memory system as long as the system provides some local shared memory.
CHAPTER 6

SUMMARY AND CONCLUSIONS

This chapter summarizes the work, presents some recommendations, and lists the future work that may be carried out in this area.

6.1 Summary

We show that swap and likewise fetch&add may be used to synchronize any two processors in a system of N processors. Further, either may be used to synchronize two processors in a contention free manner. This simple and somewhat obvious observation in itself is not of much significance. However, what is significant is the fact that one may combine more than one synchronization variable to build useful data structures. We exploit this fact to build a singly linked list using swap as the only atomic primitive. Later, we use the list to develop a number of algorithms.

We have developed three new mutual exclusion algorithms. The first two extend the original locks of Anderson and Graunke & Thakkar to eliminate contention in systems with or without coherent caches [8][33]. Their original locks are designed to eliminate contention in systems with coherent caches or systems with distributed memory. The third lock extends Mellor-Crummey and Scott's lock to preserve the FIFO behavior using swap as the only primitive. FIFO behavior is not preserved in their lock when swap is the only atomic primitive used. All of these locks eliminate contention by spinning on local shared memory. Each requires a constant number of transactions over the bus or the network to acquire and release the lock. The first two require per lock global memory linear to the number of processors. The third requires a small fixed amount of global memory.

Next, we present six readers-writers algorithms. Two of them use fetch&add and use memory linear to the number of processors. The simpler of the two relies on coherent
cache to eliminate contention. The other spins in local shared memory and eliminates contention in architectures with or without coherent caches.

The rest of the four readers-writers all use swap as the only atomic primitive, use a fixed amount of global shared memory per lock, and eliminate contention by busy waiting in the local shared memory. One of these locks is based on a singly linked list. The other three use a doubly linked data structure and provide a varying degree of concurrency at the expense of additional complexity.

To the best of our knowledge no fair readers-writers locks using swap or fetch&add as the only atomic primitive that eliminates contention in architectures without coherent caches have been previously reported.

Next, we present concurrent queue implementations in the context of the producer consumer problem. Our array based implementation for multiple producer single consumer uses fetch&add as the only primitive. It is lock free as well as wait free. We extend the array based implementation to allow multiple consumers. The extension which is also lock free and wait free, however, requires compare&swap capability.

Our linked list based implementation of a concurrent queue uses swap as the only atomic primitive to maintain the list. It allows concurrent queuing by multiple processors along with dequeuing by one processor. To the best of our knowledge no wait free concurrent queue using swap as the only primitive has been previously reported. We believe this is a novel contribution. In fact it is the ability to maintain a FIFO linked list using swap as the only atomic primitive that we exploit to develop list based mutual exclusion and readers-writers locks.

6.2 Performance

We have measured the performance of our mutual exclusion and readers-writers algorithms using a VME based shared memory system with 16 processors and a memory board. The performance is measured with a 16 μs loop in the critical section as well as with a test that makes 20 accesses to the global shared memory. The performance measurements indicate the per lock cost of our mutual exclusion locks stays constant and is...
not sensitive to the number of processors in the system. Likewise, the per lock cost of our readers-writers locks that use local only spinning stays constant for a given ratio of reader locks versus writer locks.

The performance of our linked list base mutual exclusion lock, μE-3 is better than the performance of the other two when the lock is relatively highly contended. The estimated cost of acquiring and releasing the lock under these conditions in our VME based system is 5.5 µs. The cost of the other two under the same conditions is slightly higher. The per lock cost of the linked list based algorithm is comparatively higher when the lock is lightly contended. The worst case occurs when a processor acquires a free lock and sets the lock back to free. It takes approximately 14 µs to acquire and release the lock in the worst case scenario on our test system. The other two locks perform relatively better when the lock is lightly contended.

The impact of global spinning is very detrimental to the performance of the system. The performance degradation depends on the number of waiting processors and the extent to which the bus is utilized by the processor holding the lock. With 20 accesses to the global memory in the critical section, the per lock time of a simple Bit Test&Set algorithm increases from 20 µs to 1.1 ms when the number of processors increases from 1 to 16.

The performance of the readers writers lock likewise stays constant and is not sensitive to the number of processors in the system. The doubly linked list based lock RW.DL-2 performs better compared to the others in the group of the algorithms whose performance is measured when the lock is highly contended. Like its mutual exclusion counterpart, it also takes more time to acquire a free lock and set the lock free as it executes extra steps under this condition.

6.3 Recommendations

It is fair to assume that contention free locks, in general, would be preferred over the other types of locks. There are many situations in which FIFO behavior may also be very desirable. As an example in message processing systems with high traffic rates FIFO behavior is very desirable as it eliminates the possibility of starvation. When FIFO behavior is required or desired, the system does not support a coherent cache but does support local
shared memory, array based mutual exclusion algorithm μE-1 or linked list based mutual
exclusion algorithm μE-3 may be used. If the system supports swap but does not support
compare&swap, μE-3 is the clear choice. Our linked list based algorithm μE-3 is simpler
compared to Craig's algorithm which extends Graunke & Thakkar' lock to provide similar
capabilities [17]. His algorithm, however, provides another choice when swap is the only
atomic primitive supported by the system. If the system supports compare&swap in ad-
dition to swap, Mellor-Crummey and Scott's MCS lock is a better choice [73]. In systems
that do not support swap or compare&swap but do support atomic fetch&add, μE-1 may
be used to eliminate contention and maintain the FIFO behavior. In a cache coherent sys-
tem or a system with distributed memory in a network based architecture, Anderson' lock
may be a better choice as it is quite simple [8].

Among the readers-writers locks we recommend the use of the array based RW.AR-1 in
system that support coherent cache or the network based systems with distributed mem-
ory that support fetch&add. Systems that do not support coherent cache, but provide lo-
cal shared memory and support fetch&add, but not other atomic primitives, may use the
array based RW.AR-2.

Systems that support swap may use singly linked list based RW.SL-1 or the doubly linked
list based RW.DL-2. Both spin on the local shared memory and eliminate contention from
architectures with or without coherent caches. RW.SL-1 seems to perform somewhat bet-
ter when the lock is lightly contended and the ratio of reader locks versus writer locks is
higher than one. When the lock is heavily contended or when a process acquires more
writer locks compared to the reader locks, RW.DL-2 seems to perform somewhat better.
If a system supports compare&swap in addition to swap, the list based readers-writers
algorithms may be modified to take advantage of compare&swap.

The concurrent queue implementations are useful for any processor to processor commu-
nication. There are a number of systems that support swap but not other more powerful
atomic primitives. Our linked list based concurrent queue allows such systems to build
wait free inter-processor communication. In many systems the multiple producers, single
consumer model may suffice. We recommend the use of the list based wait free multiple
producers, single consumer concurrent queue in such systems. If a situation requires
concurrent consumer access, consumers may use a contention free lock to lock the queue.

6.4 Suggested Future Work

We did not deal with priority locks in our mutual exclusion locks. We believe our linked list based mutual exclusion lock may be easily extended to provide priority locks. It will require the head of the list to be explicitly maintained. This is one area where further research work may be pursued.

We only considered fair readers-writers in our work. There are other variants of readers-writers locks, e.g., reader preference, writer preference, etc. It would be interesting to see how our readers-writers algorithms may be modified for these variants.

Another area of related research is to extend this work to implement blocking techniques using minimal hardware support. The suggested hardware support is the ability to interrupt a processor from any other processor in the system. Such capabilities are usually already provided, e.g., the mail box technique in VME. Using the hardware support and linked list mechanisms, processors could block themselves rather than busy wait. We believe it is also possible to combine blocking and busy waiting in a manner that processors could dynamically change from one to the other. This we think is an interesting area of work.


---

1. SPARC and SPARC CPU-2CE are trademarks of SPARC International Inc.


185


2. Wind River Systems and VxWorks are registered trademarks of Wind River Systems, Inc., Alameda, California.