IMPROVING PROCESSOR UTILIZATION IN
MULTIPLE CONTEXT PROCESSOR ARCHITECTURES

A Dissertation Presented to
The Faculty of the
Fritz J. And Dolores H. Russ
College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirement for the Degree
Doctor of Philosophy

by
Timothy F. Killeen
August, 1997
Acknowledgment

I would like to thank the members of my committee, Dr. Jeffrey Dill, Dr. Jeffrey Giesey, Dr. Kendree Sampson, and Dr. Constantinos Vassiliadis for their insightful comments and recommendations. I am especially grateful to my advisor, Dr. Mehmet Celenk for his persistence and tireless attention to detail.
## TABLE OF CONTENTS

### Chapter 1
- Introduction ......................................................... 3

### Chapter 2
- Background and Related Research........................................ 8
- Instruction Stream Interleaving ........................................ 8
- Multiple Context Processor Architectures .............................. 34
- Finely Threaded Processors ............................................. 37
- Coarsely Threaded Processors ........................................... 40
- Interprocess Communication ............................................ 41

### Chapter 3
- Proposed Multiple Context Architecture .............................. 46
- MCA Architecture ...................................................... 46
- Instruction Stream Execution Control .................................. 48
- Register Access and Interprocess Communication ...................... 58
- Fine-grain Scheduling .................................................. 68

### Chapter 4
- MCA Instruction Set Extensions ........................................ 74
- Context Management Instructions ....................................... 74
- Scheduling Instructions .................................................. 76
- Interprocess Communication Instructions ............................... 80

### Chapter 5
- MC/OS Operating System Design ......................................... 83
- MC/OS System Organization ............................................. 83
- Nano-Kernel Structure .................................................... 86
- Kernel Tasks ............................................................... 92
- Coarse-grain Scheduling (CGS) Task .................................... 93
- System Task .............................................................. 104
- Clock Task ............................................................... 104
- Server Tasks ............................................................. 106

### Chapter 6
- Simulation Platform ....................................................... 107
- MCA Processor Simulation .............................................. 107
- MC/OS Simulation ....................................................... 112

### Chapter 7
Simulation Results ........................................... 114  
Distributed Task Performance ............................... 117  
Computation Intensive Task Performance ................ 127  
I/O Intensive Application Performance ................... 131  
Comparison of Scheduling Algorithms ..................... 135  
Summary ................................................................ 139  

Chapter 8  
Conclusions and Future Work .................................. 143  

References ........................................................ 146  

Appendix I  
MC/OS Source Code ................................................. 149  
Process Control Block ........................................... 150  
Task Initialization Table ......................................... 153  
Hardware Initialization Routine .............................. 157  
MC/OS System Initialization and Loading .................. 159  
Clock and System Tasks .......................................... 163  
Process Scheduling Functions ................................. 184  
Coarse-grain Scheduler Task ................................. 193  
Round Robin Scheduling Variations ......................... 198  
Unmodified Minix Scheduling and IPC ..................... 203  

Appendix II  
Simulation Source Code ........................................... 217  
MA Simulation with “bochs” ................................. 217  
Bochs Module Modifications ..................................... 219  

Appendix III  
Experimental Data and Simulation Results ................. 252  
Minix Instruction Counts ......................................... 254  
MC/OS IPC Directed Scheduling Results ................... 261  
Round Robin Scheduling Results ............................. 285
List of Tables

Table 1 Stream priority levels ........................................... 51
Table 2 Stream state conditions ............................................ 52
Table 3 Register dependency (Rd) setting ................................. 53
Table 4 Instruction dependency (Id) setting ............................. 53
Table 5 Context management instructions ................................. 75
Table 6 Scheduling instructions ............................................. 77
Table 7 IPC instructions ..................................................... 80
Table 8 Context management functions .................................... 88
Table 9 Critical kernel tasks/drivers ....................................... 93
Table 10 Distributed application performance results ..................... 118
Table 11 Computation intensive performance results
separate execution .............................................................. 128
Table 12 Computation intensive performance results
concurrent execution ........................................................... 129
Table 13 I/O intensive performance results ............................... 132
Table 14 Comparison of round robin and IPC driven
scheduling algorithms ....................................................... 136
List of Illustrations

Figure 2.1 Multitasking operating system model .......................... 11
Figure 2.2 Multithreaded multitasking system model ..................... 12
Figure 2.3 Timing diagram for two communicating tasks .................. 14
Figure 2.4 Utilization (a) and speedup (b) vs process switch cost ........ 22
Figure 2.5 Task execution timeline showing short latency delays ........ 25
Figure 2.6 Non-interleaved execution of threads $P_1^1$ and $P_1^2$ .......... 28
Figure 2.7 Software multithreaded execution timeline for $P_1^2$ and $P_1^3$ of Figure 2.2 ....................................................... 30
Figure 2.8 Hardware multithreaded execution timeline .................. 33
Figure 2.9 Hardware multithreaded processor model ....................... 35
Figure 2.10 Steps required to copy a message between tasks ............. 43
Figure 3.1 MCA architectural model ........................................ 47
Figure 3.2 Block diagram of MCA processor architecture ................. 49
Figure 3.3 Stream state register (SSR) ...................................... 51
Figure 3.4 Illustration of register and instruction dependencies .......... 54
Figure 3.5 Register windows and two level register access mechanism ...... 56
Figure 3.6 Window tag register (WTR) ...................................... 57
Figure 3.7 Sample interprocess communication between processes .......... 60
Figure 3.8 Example IPC between two tasks using register window mechanism .................................................. 61
Figure 3.9 Task name register table (TNR) ........................................ 64
Figure 3.10 Stream selection flow diagram ................................. 70
Figure 3.11 Task activity register (TAR) set ................................ 72
Figure 4.1 Sample interprocess communication between processes .......................................................... 81
Figure 5.1 MC/OS organizational structure ................................ 84
Figure 5.2 Scheduling of four communicating contexts .................. 97
Figure 5.3 Round robin scheduling activity on 2 hardware context MCA processor ......................................... 98
Figure 5.4 IPC driven scheduling activity on 2 hardware context MCA processor ............................................ 100
Figure 6.1 MCA simulation platform ........................................... 109
Figure 6.2 MCA extensions to bochs simulation software .............. 110
Figure 6.3 MCA instruction extraction and simulation within bochs ................................................................. 111
Figure 6.4 Syntax of MCA instruction simulation call .................... 112
Figure 7.1 Kernel overhead for multiple contexts executing distributed application ........................................... 123
Figure 7.2 Processor Utilization for multiple contexts running distributed application ...................................... 124
Figure 7.3 Execution speedup with multiple hardware contexts for distributed application ................................. 127
Figure 7.4 Kernel overhead for I/O bound application with IPC driven scheduling ............................................ 133
Figure 7.5 Processor Utilization for multiple contexts running I/O bound application ....................................... 134
Figure 7.6 Speedup of I/O bound application for IPC driven scheduling........ 134
Chapter 1

Introduction

General purpose computing systems, such as personal computers and workstations, often have available far more CPU (central processing unit) capacity than a single large program can use effectively because of latencies introduced by much slower peripheral devices. Technological advances in processor manufacturing are broadening the performance gap between the CPU and peripheral devices to the point where main memory access can considerably affect performance. Event-driven, or distributed, programming is an effective software design paradigm to overcome this discrepancy. This involves using several smaller cooperating tasks to perform the same work of a single large program. For this strategy to work correctly, tasks must have a reliable method of data exchange and synchronization. Message passing is a commonly used communication mechanism in many event-driven systems such as micro-kernel, client/server, and distributed multitasking operating systems, which require an efficient means of interprocess communication (IPC) for data transfer and synchronization across process boundaries [1].

Coarse-grained multitasking of independent processes has long been used to overlap computations with long latency operations such as input/output (I/O) and event synchronization. This improves processor utilization and overall system throughput. This
feature has usually been implemented in software resulting in substantial process switching and communication overhead. Event-driven programming exacerbates this problem by simultaneously increasing the level of multitasking. That is, large programs are split into many smaller, cooperating tasks, and all of these tasks compete for the CPU. As a result, there is an increase in interprocess communication and synchronization required among these cooperating tasks. In distributed systems, IPC and process switching overhead can represent a significant percentage of total execution time, erasing much, or all, of the gains offered by multitasking. Methods for reducing both IPC and process switch overhead are therefore needed.

Multithreading is a hardware and/or software oriented technique used for improving processor utilization through support for multiple execution contexts and rapid context switching [2-3]. This involves overlapping several independent program contexts in an effort to mitigate the effects of short-term latencies such as main memory access and pipeline stalls. While this paradigm reduces the overhead associated with short term latencies, and in some cases context switching, it does not directly address reducing the cost of IPC between tasks.

While a substantial amount of effort has been invested in improving intra-process (or procedure) communication [4-14], most efforts at improving IPC efficiency have met with limited success. This is partly due to the fact that most designs attempt to adapt intra-process communication techniques to IPC, with the constraints imposed by a processor
designed to optimize single task execution. For example, recent Intel processors, such as the 80486 and Pentium, have added capability for segmented memory management. Segments can be used to share and/or exchange data buffers in memory. However, segmentation was designed to assist in flexible memory management within individual programs, not as a way to support communication between processes. Not surprisingly, most IPC mechanisms perform poorly. For example, independent of processor speed, transferring a small amount of data in a typical system consumes between 2500 and 5000 cycles, with even the most optimistic designs to date requiring at least 250 cycles for an 8 byte message [15]. This is between one and two orders of magnitude slower than similar main memory access. The result is that the efficacy of distributed systems in general, and micro-kernel systems in particular, is contested due to the observed lack of efficiency. Not surprisingly, a further result is that end-users tend to circumvent IPC [16].

An additional factor affecting the performance of multitasking and multithreaded systems involves task scheduling. There are two main issues which must be addressed. First, it is important to minimize the frequency of context switching to reduce overhead. In addition, it is equally important, especially in multithreaded systems, to have the correct mix of tasks scheduled to hide as much short/long latency delay as possible. Identifying which tasks to run, and when, is normally performed in software and introduces additional overhead. This can be substantial as the number of tasks increases. Most multithreaded processors usually employ two levels of processor scheduling [17]. A fine grained scheduler allocates CPU resources and a coarse grained scheduler determines which tasks should be made available
to the fine grained scheduler. Since there is generally little interaction between these two schedulers, sophisticated run-time determination of good process mixes is difficult.

The main objective of this research is to develop an architecture that reduces the overhead associated with IPC and task scheduling by integrating support for these into both the hardware and operating system kernel in order to improve processor utilization and overall throughput in a general purpose multitasking environment. We propose a coarse-grained operating system scheduling mechanism which performs space-domain multiplexing of program tasks into a multiple context processor architecture. Further, we propose a hardware implemented fine-grained instruction scheduler, which maps instructions into the processor’s execution unit(s) using time-domain multiplexing. Interaction between the hardware based fine-grained scheduler and a software coarse-grained scheduler makes it possible to probabilistically schedule tasks to improve on conventional scheduling techniques in IPC driven environments. We propose to reduce IPC overhead with the use of a simple, effective and flexible hardware oriented message protocol for short message transfer through a logically relocatable (i.e., logically addressed) register file. Relocation is accomplished through indirect register addressing at the instruction level.

In Chapter 2, we discuss IPC and scheduling issues related to multitasking and multithreaded processors. We also present related research performed by others which influences our proposed architecture. Chapter 3 describes the architecture we developed to support efficient IPC and scheduling. In Chapter 4, we describe the instruction set
requirements necessary to implement the complete Multiple Context Architecture (MCA) processor. In Chapter 5, we discuss the Multiple Context Operating System (MC/OS) design and how it interacts with MCA to minimize run-time overhead in a general multitasking environment. Chapters 6 and 7 describe the simulation platform and experiments conducted to evaluate the effectiveness of the architecture developed in this study.
Chapter 2

Background and Related Research

This Chapter contains background information regarding multitasking computing systems. We specifically address the overhead introduced by interprocess communication and task scheduling in order to provide the necessary motivation and required characteristics of the proposed architecture. The interested reader can find additional information on basic knowledge of instruction execution on conventional uniprocessor systems in [19], multitasking operating systems in [18], and interprocess communication in [3]. We first discuss performance issues, specifically processor utilization and execution overhead, using a theoretical multitasking processor model. Existing multiple context processor architectures are presented next. Multiple context architectures [20] provide a mechanism for reducing overhead in multitasking systems.

2.1 Instruction Stream Interleaving

Stream interleaving, or multithreading, can be accomplished at both a fine and coarse level. Coarse grain multithreading usually involves switching among several compiler generated contexts, or threads, within a single program. This primarily software approach is used to hide short latency delays, such as cache misses or branch delays, without incurring the full cost of a hardware context switch between threads. This method of multithreading is considered coarse in that instruction interleaving is performed by the compiler and is thus transparent to the hardware. That is, interleaving of instructions is performed within a
single program, not between programs.

As mentioned above, fine grain multithreading, or multistreaming, hides the same short latency delays by interleaving instructions for separate programs. To accomplish this, the processor must maintain several hardware contexts, one for each independent program. Multistreaming is fine grained since instruction interleaving is performed by the hardware, and is therefore transparent to the compiler and user software. That is, interleaving is performed between programs, resulting in a finer granularity of instruction interleaving.

Both techniques can be very effective at increasing processor utilization and overall throughput without greatly increasing overhead. However, the two approaches are vastly different in the way this is accomplished. The first is hardware independent, but requires extensive compiler support to achieve success. More important, it is application dependent. That is, the level of interleaving is a one-to-one relation to the inherent parallelism which can be found within a given program. This is not true of multistreaming, which derives success from having a large number of independent programs to choose from. One important advantage of multistreaming over multithreading is that a software threaded program can be mapped easily onto a hardware interleaved processor (as one context), without any loss in performance. However, the reverse is not true without some performance degradation.

To illustrate how multithreading and multistreaming systems operate and the inherent
differences between the two approaches, we will consider a typical multitasking environment as shown in Figure 2.1. This model is representative of many operating systems where independent user tasks request services from O/S (operating system) servers via system calls to the O/S kernel. In this model, we have three distinct layers. Within each layer, we can identify potential parallelism. At the user layer, individual users can submit programs that are unrelated. Within the middle (services) layer, many of the services provided, such as file and memory management, must interact but can be performed either sequentially or in parallel. Within the kernel layer, interrupt handlers, scheduling and service request/reply operations are largely independent and can be performed in parallel. Lastly, within specific subtasks, for example the memory manager service, local parallelism may be uncovered by the compiler, allowing the generation of threads within separate programs.
Figure 2.2 illustrates how this might be implemented in a hardware multithreaded environment. The three layer model appears to the hardware (and operating system) as a collection of (smaller) tasks which must be scheduled and executed. In this example we have shown two user programs, each consisting of four distinct threads, and two server programs, each containing m threads. The directed graph presentation also shows dependencies among the threads. User process 1, for example, shows that threads $P_{1}^{2}$ and $P_{1}^{3}$ are both dependent on thread $P_{1}^{1}$, and thread $P_{1}^{4}$ is dependent on both $P_{1}^{2}$ and $P_{1}^{3}$. This
means that $P_{11}$ must complete before starting either $P_{12}$ or $P_{13}$. However, since $P_{12}$ and $P_{13}$ are independent, their instructions can be interleaved. Finally, $P_{14}$ must wait for $P_{12}$ and $P_{13}$ to complete before executing. This is illustrated by the following example.

![Diagram of multithreaded multitasking system model]

*Figure 2.2 Multithreaded multitasking system model.*
Suppose the two user processes shown in Figure 2.2 are to be executed and have the following characteristics: First, assume each user thread issues one request for some operating system service during its execution. Further, a service request involves a request to the kernel which processes any IPC and calls the appropriate server to process the request. The server replies, requiring scheduling of the user thread by the kernel, as well as processing of any reply IPC, at which point the user thread resumes execution. This sequence of steps is labeled from 1 to 4 in Figure 2.2.

This process is also illustrated in Figure 2.3, below. This diagram shows the effects of process switch/scheduling when two processes, \( P_1 \) and \( S^i_2 \), execute concurrently on a processor. Assume that \( P_1 \) and \( S^i_2 \) are two threads from Figure 2.2. It can be concluded from the figures that total overhead and total execution times are given by

\[
T_{ov} = (t_3-t_2) + (t_5-t_4) + (t_7-t_6) + (t_9-t_8) \tag{2.1a}
\]

and

\[
T_{ex}(P_1) = (t_2-t_4) + (t_6-t_3) + (t_9-t_8) \tag{2.1b}
\]

\[
T_{ex}(S^i_2) = (t_4-t_3) + (t_9-t_7) \tag{2.1c}
\]
Figure 2.3 Timing diagram for two concurrent tasks.

Note that without overhead considerations, $T_{th} = T_{seq} = T_{ex}(P_1) + T_{ex}(S_2)$. When overhead is factored in, in an effort to hide long latency delays resulting from communication between the two programs, $T_{seq}$ remains the same (i.e., if the two separate portions were compiled together as a single program) while $T'_{th} = T_{ex}(P_1) + T_{ex}(S_2) + T_{ov}$. $T_{ov}$ consists of at least two kinds of overhead. First, time is spent scheduling processes and performing context
switches when it is necessary to stop executing one process and begin executing another. Second, if there is any interprocess communication between the processes, data must be moved between address spaces. It is important to reduce these overheads to improve the system performance.

The main objective in the design of high performance computing systems is to improve the overall performance of a collection of processes in a general-purpose multitasking environment by minimizing overhead not essential to a program’s execution. In order to accurately predict real performance in multitasking systems, we use the following definition of utilization in the ensuing discussion:

\[
\text{Given a sequence (program) of } N \text{ instructions necessary to execute a given task } T, \text{ on a processor } P \text{ with } F \text{ function units, in } C_{\text{cycles}} \text{ clock cycles, then processor utilization for } T \text{ is given by } U = \frac{N}{(C_{\text{cycles}}F)}.
\]

Similar definitions can be found in [3]. Here, the sequence of N instructions consists of necessary instructions from both application and operating system space. In other words, two processors can be considered equivalent if their utilization values are identical, regardless of the underlying architecture of the processor. This definition is general enough to allow meaningful comparison between processors having multiple functional units with multiple instructions issued each clock cycle.

Unfortunately, in real systems executing arbitrary programs the delays, both short and long,
are inconsistent and unpredictable. We are forced to examine probabilistic events within the context of a deterministic environment. Moreover, there are additional sources of overhead in software and hardware multithreaded processors. In the remainder of this section we explore the algebraic representation of the various components of overhead in multitasking systems.

For performance analysis, we consider a task consisting of a sequence of instructions labeled 1 through N. The sequential execution time of T, denoted as \( T_{\text{seq}} \), and the processor utilization, called \( U_{\text{seq}} \), are given by

\[
T_{\text{seq}} = \sum_{k=1}^{N} C_{\text{pi}} \leq N(C_{\text{pi max}}) \tag{2.2a}
\]

or

\[
T_{\text{seq}} \approx N(C_{\text{pi avg}}) \approx N(C_{\text{pi seq}}) \tag{2.2b}
\]

and,

\[
U_{\text{seq}} = 1/C_{\text{pi avg}} \tag{2.2c}
\]

where \( C_{\text{pi}} \) represents the number of cycles required per completed instruction, and is replaced by \( C_{\text{pi avg}} \). In subsequent analysis, we refer to \( C_{\text{pi avg}} \) as \( C_{\text{pi seq}} \) to denote the average instruction execution time in a sequential processor. This number, for large tasks, can be interpreted as a practical upper bound. Sequential uniprocessor utilization, \( U_{\text{seq}} \), is easily seen to be the inverse of \( C_{\text{pi avg}} \) since \( C_{\text{cycles}} \approx N(C_{\text{pi avg}}) \) and \( F=1 \).

Exploitation of parallelism, through the development of parallel threads of execution, naturally leads to a reduction in total execution time by reducing \( C_{\text{pi avg}} \). This is measured...
by the system speedup $S$ defined as $S = \frac{T_{\text{seq}}}{T_{\text{th}}}$, where $T_{\text{th}}$ is the threaded execution time of the task $T$.

The equations 2.2a-2.2c are derived assuming ideal execution environments. Several factors including inter-process communication (IPC) costs, data and control dependencies, scheduling, and process switch overhead contribute to longer execution times and reduced processor utilization in practice. We now consider these four performance degradation factors within the context of a multitasking processor environment using the model of Figures 2.1 and 2.2. Figure 2.1 shows the system structure for several separate tasks in a typical multitasking operating system. In Figure 2.2 we depict a conceptual organization of the process model of Figure 2.1 as a collection of distinct threads, or tasks, within three processes.

Essentially, communication and process switch overheads are a function of the number of separate subtasks of the larger task $T$, while data and control dependencies are related to the instruction sequence itself, affecting processor (i.e., pipeline) efficiency and the way in which the task $T$ is partitioned into subtasks. Reflecting the factors noted above, the actual multithreaded execution time, $T'_{\text{th}}$, is given as

$$T'_{\text{th}} = T_{\text{th}} + T_d + T_c + T_s + T_{\text{sch}}$$

where $T_d$, $T_c$, $T_s$ and $T_{\text{sch}}$ are the overheads of control and data dependencies, interprocess
communication, process switches, and process scheduling. We will be considering the effect of process scheduling, as an operating system activity, on system performance. In particular, as the cumulative number of tasks increases, the complexity of scheduling can be expected to increase because of the increased number of possible scheduling combinations. In the ensuing discussion, however, we combine process switch and scheduling cost since they are closely related and in sequential processing are inseparable.

**Control dependencies.** Control dependencies arise when the execution of one group of instructions in a task depends on the results of another group of instructions. This is illustrated by the conditional statement

```plaintext
if (A)  
    B;  
else  
    C;
```

In this example the execution of B or C is dependent on the evaluation of A. Dependencies between adjacent instructions limit processor utilization by requiring delays or gaps. Instruction interleaving, by filling gaps with (independent) instructions from separate streams, can greatly improve processor utilization if process (stream) switch time is small [3]. The contribution to total execution time of control and data dependencies is determined by the total number of process switches and delays, which effectively increases $C_{pi}$, since more cycles are required to complete the task T. Thus, the cost of data dependencies is
\[ T_d = S_i(N) \] (2.4a)

where \( S_i \) is the delay rate, in cycles per instruction, encountered in executing the \( N \) instructions of task \( T \). Recognizing that \( S_i \) represents a linear increase in \( C_{pi_p} \), \( T_d \) can be included in \( T_{th} \) by using a \( C_{pi_p} \) value that corresponds to the average instruction issue and delay rate \( (S_i) \).

**Interprocess Communication Overhead.** Communication costs are a function of the number of subtasks and the resulting communication required to synchronize these tasks and exchange data among them. The total cost of interprocess communication is given by

\[ T_c = (t-1+M)C_m \] (2.4b)

Here it is assumed that the task \( T \) is subdivided into \( t \) subtasks which results in \( t-1 \) messages to synchronize initial execution and \( M \) number of additional messages during execution. The cost of intra-processor communication \( (C_m) \) is defined as the additional number of clock cycles required to manage the transfer of a message from one task to another within a single processor.

**Scheduling and context switch overhead.** Scheduling and context overhead arises when a single processor is called upon to execute several tasks concurrently. The number of
process switches encountered during execution of the task T is a function of the number of subtasks per processor, the frequency and method of IPC, as well as other less predictable factors such as interrupts, total number of processes in the system, and so on. Process switch cost $T_s$, therefore, is given as

$$T_s = (t-1+S_w)C_s$$ \hfill (2.4c)

Here, it is assumed that the task T is subdivided into t subtasks, requiring t-1 process switches to initiate all t subtasks, as well as $S_w$ additional switches during execution, and $C_s$ is the cost of a process switch. Then a better estimate of $T_{ih}$ is given by

$$T_{ih} = C_{pi_{ih}}(k+N-1)+(t-1+M)C_m+(t-1+S_w)C_s$$ \hfill (2.5a)

where $M$ and $S$ are application and implementation dependent communication and process switch counts, and $C_{pi_{ih}}$ is the average instruction issue rate when control dependencies are considered. This results in a speedup of

$$S' = N \cdot C_{pi_{seq}} / [(C_{pi_{ih}}(k+N-1)+(t-1+M)C_m+(t-1+S_w)C_s)]$$ \hfill (2.5b)

and,

$$U'_{ih} = N / [(C_{pi_{ih}}(k+N-1)+(t-1+M)C_m+(t-1+S_w)C_s)]$$ \hfill (2.5c)

These equations may be simplified in the following way. First, the terms $k-1$ and $t-1$ may be dropped based on the observation that, for large N, $N \gg k$ and $N \gg t$. This results in
\[ T'_{th} = C_{pi_{th}}(N)+(M)C_m+(S_w)C_s \]  \hspace{1cm} (2.6a)

\[ S' = N\cdot C_{pi_{seq}} / [(C_{pi_{th}}(N)+(M)C_m+(S_w)C_s) \]  \hspace{1cm} (2.6b)

and,

\[ U'_{th} = N / [C_{pi_{th}}(N)+(M)C_m+(S_w)C_s] \]  \hspace{1cm} (2.6c)

Finally, terms can be normalized by factoring out \( N \). With this factoring we express all terms as a cost per instruction rather than as a total cost for an \( N \) instruction task. Through a change of variables, with \( M_r=M/N \) and \( S_r=S_w/N \), we get

\[ T'_{th} = N(C_{pi_{th}}+M_rC_m+S_rC_s) \]  \hspace{1cm} (2.7a)

with

\[ S' = C_{pi_{seq}} / (C_{pi_{th}}+M_rC_m+S_rC_s) \]  \hspace{1cm} (2.7b)

and

\[ U'_{th} = 1 / (C_{pi_{th}}+M_rC_m+S_rC_s) \]  \hspace{1cm} (2.7c)

These equations describe the effects of multithreading (\( C_{pi_{th}} \)) and resulting system overhead (parameters \( M_rC_m \) and \( S_rC_s \)) on the processor utilization of multitasking systems. From a system design perspective, we wish to minimize the cost and/or frequency of scheduling and IPC without affecting \( C_{pi_{th}} \).

A parameterized plot showing utilization and speedup, with \( C_{pi_{seq}}=3.0 \), as a function of process switch cost per instruction is given in Figure 2.4 based on equations 2.7b and 2.7c with the remaining terms kept constant.
Processor Utilization

![Graph showing processor utilization](image)

Parallel Execution Speedup vs sequential processor with Cpi=3.0

![Graph showing parallel execution speedup](image)

Figure 2.4 Utilization (a) and speedup (b) vs process switch cost.
Plots for three different constant values of $C_{pi_{th}}$ and $M_rC_m$ are shown. The formulas yield a decreasing plot, asymptotically approaching 0, as a function of $S_rC_s$. For example, using the utilization plot of Figure 2.4a, there are essentially three important cases worth considering:

$C_{pi_{th}} + M_rC_m \gg 1 > S_rC_s$. If the constant term substantially exceeds unity (i.e., $C_{pi_{th}}$ and $M_rC_m$ are greater than unity), the utilization is nearly constant and changes in the variable term have little noticeable effect on performance. If $S_rC_s$ increases enough, it begins to contribute to degradation; however, performance is already poor. This case is illustrated by the curves with $C_{pi_{th}} + M_rC_m = 3$ in Figure 2.4.

$C_{pi_{th}} + M_rC_m \approx 1 > S_rC_s$. When the constant terms is close to ideal (i.e., when $C_{pi_{th}}$ is close to unity, and $M_rC_m$ is close to zero), the impact of the variable term(s) on performance can be appreciated. This is illustrated by the top curves in Figure 2.4.

$C_{pi_{seq}} = C_{pi_{th}}$. Finally, the equations 2.7(a)-(c) indicate that performance in a concurrent computing system can be worse than sequential processing. If the ratio of $C_{pi_{seq}}$ and $C_{pi_{th}}$ is small, as it effectively is on many modern processors, then increased process switch and IPC costs, in the denominator of equation 2.7b, can result in a speedup of less than 1. This can result when a single (large) program is split into several sub-programs (or threads). In other words, it might be a disadvantage to implement distributed software on a processor optimized for single thread performance because the added overhead of IPC and context
switching between tasks can add more latency than the multitasking hides. This last
observation clearly illustrates the need to consider all three terms together when designing
a multitasking system.

Using the model and equations described above, we can determine processor utilization for
the execution of the sample programs of Figure 2.2. This is accomplished by calculating
the amount of time, in clock cycles, spent executing each thread. The processor is fully
utilized if it executes one necessary instruction in each clock cycle. Instructions which
contribute to $T_{ov}$ are considered overhead. In this example, these are kernel instructions.
That is, if the user and server programs were written as a single program, then no kernel
instructions would be required. In this example, utilization is given by

$$U = \frac{\text{(# user and server instructions)}}{\text{( # clock cycles)}}$$

(2.8)

To simplify calculations, we specify that all user and server level threads are exactly ten
instructions long, while kernel threads are five instructions in length. Note, however, that
for each system call/return the kernel thread is invoked twice. In this way, the execution
of each thread involves ten instructions from user, server, and kernel space, for a total of
thirty instructions to fully execute one user thread. Finally, all threads exhibit a short
latency pattern such that a delay of $L$ instructions occurs for every $K$ instructions executed.
This is not realistic of programs in general, but is useful for illustrating the theoretical value
of multithreading. Older, non-pipelined processors, such as the Intel 8086 [38], do exhibit
fairly large and regular latencies because the processor requires several clock periods to execute any given instruction. However, the latency is not the same for all instructions. Modern processors, such as the Intel Pentium or MIPS R4000 [39-40], have been engineered through the use of pipelining so that some instructions can be issued without any latency between instructions. However, other instructions, such as branches, still require a delay before another instruction can be issued. Hence, the values for K and L represent average values. Several values of K and L are illustrated in Figure 2.5, where the unit of time measurement, or quantum, is in clock cycles.

![Legend](Image)

**Figure 2.5** Task execution timeline showing short latency delays.

In the three cases shown above, the processor is idle 50%, 33%, and 25% of the time, respectively. Interleaving instructions from other threads in these idle slots will increase overall throughput and increase processor utilization. Where the idle slots occur in time
is a function of the processor architecture and sequence of instructions being executed. Filling the idle slots can be accomplished by a compiler (software multithreading), or by the processor itself (hardware multithreading). Now, for any value of K and L we can compute the total execution time with and without instruction interleaving. In the following discussion we use values of K=1 and L=1. As before, the choice for K and L is mainly illustrative. However, studies have shown that programs exhibit small instruction to latency ratios (i.e., limited instruction level parallelism) [41,42]. We describe three execution environments: Sequential (no interleaving of any kind), software multithreading (compiler generated interleaving), and multistreaming (hardware supported stream interleaving).

**Sequential** - Without any instruction interleaving, the total execution time, $T_s$, consists of the execution time of all instructions executed, including any short latency delays, and the time required to switch between programs. Within a particular program, threads are executed sequentially in thread order, and the transition from one thread to the next incurs no delay. Thus, for the user program 1 of Figure 2.2, the order is $P_1^1, P_1^2, P_1^3, and P_1^4$. Switching between programs, however, involves scheduling, performed by the kernel, and hardware copying of state, information, which we assign the constant value of $S=2$ clock cycles. This is tracked separately from the kernel thread, because it is an action separate from scheduling, and is necessary whether scheduling is performed or not. As with the latency $L$, $S$ is mainly illustrative. The interaction between programs using the model of Figure 2.2 is given by
where $T_s$ is the total execution time, in clock cycles, of a collection of program threads. During execution threads will issue system calls for operating system services (steps 1-4 in Figure 2.2). This is stated in equation (2.9) as an average number of service calls per thread. The number of context switches to service each call is 4 in Figure 2.2 (other models might vary, but the minimum is 2). Combining all of this for the two programs of Figure 2.2 we see that there are 8 user threads, service calls/thread is 1, switches/call is 4, and the combined thread length is 30 as noted previously. The multiplier, $(K+L)/K$, accounts for the delay $L$ for every $K$ instructions, and is 2 in this example (i.e., $K=L=1$). The final term, $S$, represents the transition from program 1 to program 2. Hence $T_s$ becomes

$$T_s = 8*[1]*(4)*2+(30)*2]+2 = 8*[8+60]+2 = 546$$

The values inside the brackets in equation (2.9), 8 and 60, represent the total execution time of an individual thread, which we call $T_{ss}$ and $T_{st}$, respectively. $T_{ss}$ is the time spent switching contexts between sub-threads, and $T_{st}$ is the instruction (and delay) execution time excluding context switching. This is illustrated in the partial sequential timeline shown in Figure 2.6 for the execution of $P_1^1$ and $P_1^2$ of Figure 2.2.
Note that in addition to the inter-instruction delay due to short term latencies, there is also the overhead penalty introduced by the kernel and switching between programs (threads). Without these two kinds of overhead, the ideal execution time, $T_{\text{ideal}}$, for the above example is simply the sum of all instructions, excluding those from the kernel, or 160. Hence, for the sequential execution model processor utilization, $U_s$ is

$$U_s = \frac{T_{\text{ideal}}}{T_s} = \frac{160}{546} = 29\%$$ (2.11)

Because of short latency delays, and the need to copy contexts' in and out of the processor
when switching between programs, the hardware is utilized at less than 30% of capacity. In addition, we note again that the user program is requesting service from a server program, not from the kernel. The kernel thread is run in order to pass data between the user and server threads, schedule threads, and so on. While this is necessary for the model of Figure 2.2, it represents additional overhead resulting from the fact that two independent processes (user and server) are cooperating to accomplish a single task (i.e., whatever calculation the user program is designed for).

**Software Multithreading** - In this case, instruction interleaving is software driven in the sense that the compiler orders instructions for separate threads comprising a single program in order to maximize performance. However, no overlapping between programs is possible. Hence there is no fine grain scheduling, only long term scheduling performed by the operating system. A context switch between programs occurs only when no thread within a given program can execute. Furthermore, when a context switch is required, all threads for one program are moved out of the processor and stored in main memory, while all active threads for the next program are copied into the CPU. Thus, intuitively, a context switch in this environment can be more expensive. We keep the cost of context switching, \( S=2 \), the same for simplicity. Lastly, in this example we assume all server threads within each server can execute in parallel, and that the kernel itself is multithreaded (although this is not shown explicitly in Figure 2.2) and that kernel threads can run in parallel.

In a software threaded environment, processor utilization is directly related to the number
of intra-program threads a compiled application program exhibits. In this example, as noted above, each program has three distinct sections: Startup, execution, and conclusion. The first and last sections are single threaded (e.g., \( P_1^1 \) and \( P_1^4 \) of Figure 2.2) while the middle is multi-threaded (e.g., \( P_1^2 \) and \( P_1^3 \) of Figure 2.2). Hence, only the middle section allows for instruction interleaving. The timing diagram of Figure 2.7 illustrates this.

![Figure 2.7 Software multithreaded execution timeline for \( P_1^2 \) and \( P_1^3 \) of Figure 2.2.](image)

For software multithreading, the startup and conclusion sections of each program execute in the same amount of time as in the sequential case, on a per thread basis. The middle section, due to overlapping, executes faster. The total execution time for a software multithreaded system, \( T_{sm} \), for both programs is
\[ T_{sm} = (T_{ss} + T_{st}) \times (S_s) + \frac{(I_s)}{(I)} \times (T_{st}^* (M) + (\text{sw/thread}) \times (S + 1 - 1) + S \]
\[ = 4 \times 68 + \frac{4}{2} \times [60 + 4 \times (2 + 1)] + 2 \]
\[ = 418 \]

The first term consists of \( T_{ss} \) and \( T_{st} \), from equation (2.9), and the number of user threads which must be run sequentially (i.e., cannot be interleaved due to control dependencies, etc.), \( S_s \), and represents the sequentially executed portion, exactly as in equation (2.9). The second term is the threaded execution time. The first factor in this term, \( (I_s)/(I) \), accounts for the number of threads interleaved and the number simultaneously interleaved. For example, in Figure 2.2, \( P_1^2 \) and \( P_1^3 \) from \( P_1 \) are interleaved, and \( P_2^2 \) and \( P_2^3 \) from \( P_2 \) are interleaved. Thus the number interleaved is 4, and the interleave is 2.

The value inside the brackets is the execution time of a threaded section. The interleaved execution time is longer than that of a single thread for two reasons. First, \( I \) threads are interleaved. Each instructions uses \( K \) of \( K+L \) quantums. If the ratio \( (K+L)/K \) is greater than \( I \), then there will still be unused quantums when \( I \) streams are interleaved. For example, if \( K=1 \) and \( L=3 \), then if fewer than \( I=4 \) streams are interleaved then some unused quantums remain. For \( I>=4 \) all quantums can be utilized. Thus the interleaved execution time of \( I \) inerleaved threads is related to the execution time of a single thread by the multiplier, \( M \), given as

\[ M = I \times (K/(K+L)) \quad \text{if} \quad (K+L)/K \geq I \]
\[ M = 1 \quad \text{if} \quad (K+L)/K < I \]

Second, the last interleaved thread to complete finishes after the first completes by the
factor I-1. In Figure 2.7, \( P_1 \) uses delay slots resulting from the execution of \( P_1 \). Hence, this adds one extra delay (e.g., for I=2) following each program switch. Since there are four switches required per thread, the total execution time increases by four from 68 to 72.

With software threading alone total execution time decreases from 546 to 418, but this is still far above the ideal value of 160. We have managed to utilize some of the short latency delays, but have not addressed any of the longer latency delays (e.g., time in the kernel). For this example software multithreading yields a utilization, \( U_{sm} \), of

\[
U_{sm} = \frac{T_{ideal}}{T_{sm}} = \frac{160}{418} = 38\%
\]  

(2.14)

With software multithreading we get higher processor utilization and, in this case at least, a speedup, \( S_{sm} \), over sequential execution of

\[
S_{sm} = \frac{T_s}{T_{sm}} = \frac{546}{418} = 1.31
\]  

(2.15)

**Hardware Multithreading** - A hardware threaded processor allows instruction overlapping of threads from independent programs. Hence, more parallelism can be exploited than is possible with software threading alone. We gain an additional benefit which results from maintaining several thread states resident simultaneously. Specifically, the switch cost, \( S \), reduces to 0 (provided the desired thread is resident). Therefore, we get
the execution timeline of Figure 2.8.

With hardware multithreading, execution time is shorter than that of software multithreading. Since interleaving can be accomplished during all phases of execution, total execution time, \( T_{hm} \), for hardware multithreading is given by

\[
T_{hm} = \frac{I}{I} \times [T_{st} \times (I \times K/(K+L))] + S
\]

\[
= \frac{8}{2} \times [60] + 2
\]

\[
= 242
\]

Observe that equation (2.16) is very similar to equation (2.12) except that the delay penalty is removed from the threaded execution time because the switch time, \( S \), between threads is removed because multiple threads are resident. Thus, it is possible to allow interleaving across context switch boundaries. This also implies that no threads must be executed
sequentially. Hence, the first term of equation (2.12) is also removed. Utilization, $U_{hm}$, and speedup, $S_{hm}$, become

$$U_{hm} = \frac{T_{\text{ideal}}}{T_{hm}}$$

$$= \frac{160}{242}$$

$$= 67\%$$

and

$$S_{hm} = \frac{T_s}{T_{hm}}$$

$$= \frac{546}{242}$$

$$= 2.28$$

In order to approach 100% utilization it is necessary to reduce the need for kernel intervention during ordinary system calls. For this to be possible, IPC processing must be removed from the kernel and scheduling overhead must be minimized. A method to address this is discussed in Chapter 3.

### 2.2 Multiple Context Processor Architectures

Hardware multithreading is a means of improving processor utilization for a wide range of latency durations. Figure 2.9 shows a multithreaded processor model employing a two level scheduling mechanism.
Figure 2.9 Hardware multithreaded processor model.
The course-grain scheduler allows overlapping computations with longer latency operations, such as system calls, disk, and input/output operations. This is accomplished by multiplexing N software threads onto S hardware streams. The fine-grain scheduler (labeled SSU, Stream Selection Unit), which is described in Chapter 3, introduces overlapping computations with shorter latency operations such as main memory access, cache misses, interprocess communication, and event synchronization. This is implemented by time multiplexing instructions from different streams among a processor's F functional units. If the processor has more than one functional unit, as depicted in Figure 2.9, then there is spacial multiplexing as well.

Figure 2.9 illustrates a single processor configuration. A multiple processor organization would employ one fine-grain scheduler for each processor. For a P processor system, the coarse scheduling software would be responsible for multiplexing N software threads onto P*S hardware streams distributed among the P processors.

A number of processor architectures with multiple hardware contexts have been proposed. These can be divided into two broad categories: Finely or coarsely threaded systems. This classification is based not only on the way in which the processor functions, but also on which aspects of performance the designers wish to emphasize. In the following two sections we describe finely and coarsely threaded processors proposed in the literature.
2.3.1 Finely Threaded Processors

Finely threaded processors, such as Denelcor HEP [21], MASA [22], Horizon [23], DEMUS [24], and Tera MTA [17,25-26] architectures switch among instruction streams on every cycle. The advantage of this approach is in simplicity of implementation. The most obvious drawbacks are the poor performance of individual threads and the need for a large supply of ready threads to keep utilization high.

**HEP.** The Denelcor HEP system was an early commercial multithreaded processor. A HEP system consists of as many as 16 processors connected by a high-speed switch. Each processor has support for up to 128 instruction streams spread across 16 distinct address spaces, called protection domains. Each instruction stream consists of a program counter and a portion of the processor register file. Since the register set is shared among all streams of the processor, register reservation bits are needed to ensure atomicity of operations on shared registers.

**MASA.** MASA differs from HEP in its support for lightweight procedure calls and traps through the use of parallel streams. This is supported through a register window style bank of register sets. In MASA, newly created streams receive their own private register set, and in addition share a portion of the parent’s register set. This allows new streams to be created for each procedure call, and parameters can be passed through shared registers.

**Horizon.** The Horizon is a successor of the HEP, but unlike the HEP each stream of
the Horizon has its own private register set. This eliminates the need for synchronization and reservation bits on registers, allowing faster clock rates and smaller instruction field sizes for a given total register file size. Hence Horizon scales better to a larger number of processors.

DEMUS. The Delay Enforced Multistreaming processor is typical of older finely threaded processors in that it maintains a fixed, inflexible division of high-speed register resources for each resident context. The significant feature is its ability to interleave independent instruction streams while supporting a limited degree of coarsely threaded execution in a low cost general purpose computer.

DEMUS supports software independent instruction streams through the use of completely separate state space for each context, including general purpose registers. The advantages of this approach are simple hardware and no need for software (compiler) support. However, separate state space for all resident contexts is inflexible, resulting in wasted register resources, and interprocess communication (IPC) is not efficiently supported between streams. The result is that DEMUS performs well only in situations where there are large numbers of unrelated tasks. However, with the trend toward distributed systems this is not realistic.

Tera MTA. The Tera MTA inherits much of its architecture from Horizon and HEP. The Tera architecture is a shared memory multiprocessor with up to 16 multi-stream processors
interconnected by a packet-switched interconnection network. Like the HEP, Tera supports up to 16 separate processes and 128 streams per processor. Thus, using the model of Figure 2.9, Tera employs up to 16 fine-grain scheduling units. Unlike the HEP, each process has an independent register set with each stream managing its own register set. Tera has a hardware scheduler which selects instructions from streams using a modified round-robin mechanism. On every clock tick, processor logic selects a stream that is ready and issues its next instruction. Compiler generated lookahead is added to each instruction allowing the same stream to issue several instructions consecutively provided no conflicts are indicated. Thus, the processor can be kept fully utilized with fewer than the maximum number of streams loaded.

As with the DEMUS processor, the major drawback of the Tera MTA is the lack of support for cooperating (i.e., communicating) processes within the processor itself. In particular, since each hardware thread has a separate register file, sharing resources is more difficult. Software support has been considered within the Tera operating system. The Tera long term scheduling mechanism allows scheduling groups of threads simultaneously, called gang scheduling. This can reduce the frequency the long term scheduler is called. However, since the gang scheduling option is compiler controlled, dynamic (run-time) group scheduling of threads is not possible.
2.3.2 Coarsely Threaded Processors

Coarsely threaded processors execute a single stream for an extended period and switch streams only when a high-latency operation occurs. Examples include Register Connection (RC) [28], Register Relocation (RR) [29], and an algorithmic approach by Hidaka et al [4]. This approach improves single thread performance and requires fewer resident threads, but generally requires greater hardware and software support. Further, these systems usually suffer from high process switch overhead and lack of specific support for interprocess communication. As a result, distributed processing applications that rely on IPC for communication and synchronization do not derive much advantage from coarsely threaded processors.

RC and RR. Register Connection (RC) and Register Relocation (RR) are two separate architectures which propose to use resources more efficiently by allowing separate threads to share the same resources and registers. This results in improved process switch performance and IPC times among tasks. However, these processors rely on compile time thread and, therefore, resource management. Compilation dependent resource management makes these architectures incapable of efficient dynamic stream interleaving of independent tasks or IPC among separately compiled processes, which are important features in a multitasking or distributed system.

Hidaka et al. presents a mechanism for flexible dynamic register allocation on Sun SPARC workstations using the hardware supported window allocation scheme. This handles
window sharing between consecutive procedure instantiations, underflow, and overflow conditions more efficiently. This approach performs best for nested procedure calls and multithreaded environments that do not require extensive communication among independent tasks. This is due to the restriction that register windows must be allocated contiguously, precluding arbitrary register sharing among tasks. However, this is too restrictive in the more general case where independent (threaded) tasks utilize IPC, which causes poor processor utilization in many situations.

These architectures increase processor utilization through support for overlapping tasks during short latency operations. However they address only one aspect of performance, processor activity, choosing to ignore the overhead this incurs. This results in increased performance based on one metric, but no improvement, and possibly degradation, in other areas. In a general purpose multitasking system, it is important to consider both hardware and software performance issues to predict the overall performance more accurately.

2.4 Interprocess Communication

Numerous message passing based operating systems have been developed. These include Minix [18], Amoeba [30], Chorus [31], L3 [15], and Mach [32]. These operating systems are similar in that the underlying IPC mechanism is implemented in software. Liedtke [15] has developed an efficient software only implementation in the L3 micro-kernel. L3 supports a wide range of message types, including very long messages, without sacrificing efficiency. Short message (8-byte) transfers require as few as 250 cycles on commercial
uniprocessor architectures. However, this is still significantly slower than main memory access on the same processors.

A primarily hardware oriented approach to improve IPC was proposed by Cheriton [43], using registers for short messages. This study presents the basic mechanism for sharing general purpose registers between (partially) independent tasks, but fails to clearly identify the potential benefits of register IPC. In particular, IPC through the logical relocation (addressing) of registers is not considered. Thus, IPC between independent tasks is not feasible.

A major reason for the inefficiency of IPC in conventional architectures stems from the method of IPC implementation. Figure 2.10 illustrates the steps that must be followed to copy a message between two tasks, from task A to task B. First, A creates a message using its local data. This involves two accesses to memory. The first to copy data into the CPU, and the second from the CPU into a message buffer. Note that whether there is a cache or not only affects the time this takes, not the fact that a single datum must be copied twice. Next, a call to the operating system results in the message being transferred into a message buffer belonging to task B. Lastly, for task B to make any use of the message it must be copied back into the CPU.
The above IPC process involves four memory data transfers, labeled 1-4 in Figure 2.10, and two context switches. If CPU registers are utilized, then a single memory transfer and a context switch are needed. Even in a shared memory system, this kind of communication
still requires three memory transfers (transfers 1, 2 and 4 omitting 3 because of the shared address space) and one context switch (bypassing the O/S kernel). Moreover, it involves greater software synchronization and mutual exclusion overhead. In addition, it becomes important from a scheduling standpoint that both processes be available at the same time; otherwise, the effectiveness of shared memory is lost. In order to use CPU registers as a general IPC and synchronization mechanism, registers must be mapped out of task A's register set and mapped into Task B's. This requires new hardware primitives and different, albeit simpler, communication software.

Multitasking operating systems such as UNIX support multistreaming/multithreaded operations by executing operating system services (servers) and user applications (clients) as independent processes and relying on interprocess communication for synchronization and data exchanges. Processes may be further divided into collections of sub-processes, or threads, for faster execution.

In general purpose multitasking systems, it is necessary to consider several performance issues to obtain an accurate reflection of potential performance. In this thesis, we assume the execution of tasks in a UNIX-like multitasking operating system on a general multistreamed processor. We propose to address performance inhibiting issues within the context of a multitasking environment.

While a large number of general purpose registers is desirable to support multiple contexts,
it may not be feasible to provide direct addressing of all registers due to limitations on field-width in instruction formats. Partitioning a large register file into smaller, sharable, contexts can lead to a more efficient use of registers, and smaller instruction/program sizes. This also introduces a convenient mechanism for faster interprocess communication. Multiple contexts and direct support for IPC can increase processor utilization and decrease task execution time. We present a simple mechanism that supports sharable, relocatable processor contexts with minimal additional hardware support in multiple resident context architectures.

The proposed system addresses three areas critical to improving system performance: Coarse-grain scheduling, fine-grain scheduling, and hardware supported IPC. This requires both architectural as well as software enhancements to previously proposed systems. Architectural changes are required to support IPC and fine-grain scheduling, or time domain multiplexing of tasks. Software enhancements are needed to perform efficient long term spacial multiplexing of tasks using hardware generated run-time statistics to assist coarse-grain scheduling. These enhancements are discussed in more detail in the following three chapters.

In the next chapter, we discuss the proposed architecture and necessary hardware and software support required for shared register multiple stream processing.
Chapter 3

Proposed Multiple Context Architecture

This Chapter describes processor architecture issues specific to the proposed MCA (Multiple Context Architecture) processor and presents hardware related solutions to the overhead problems discussed in previous sections. Chapter 4 describes the instructions related to this architecture.

3.1 MCA Architecture

The MCA is a multitasking system in which many parallel programs of various sizes, in terms of both size and time, execute concurrently. MCA is a hardware multithreaded (e.g., multiple context) processor, with a hardware determined number of threads competing simultaneously for the processor's instruction execution stream using a model similar to that shown in Figure 2.9. A modified model is given in Figure 3.1. The newer model reflects architectural changes for hardware support for IPC and scheduling. Each of these is described in more detail in the following sections.

The MCA processor supports two levels of processor scheduling. The processor manages fine-grain (hardware) scheduling at the instruction level without intervention from the operating system. The operating system kernel is the coarse-grain scheduler and manages
the processor at a higher level with support from MCA status registers, determining which threads are hardware resident at any moment.

Figure 3.1 MCA architectural model.
The MCA processor also supports hardware based IPC. This is accomplished with the addition of indirect register access and instruction primitives that support mapping register windows between hardware streams.

3.2 Instruction Stream Execution Control

The hardware required to support the MCA processor is similar to other multiple context architectures discussed previously. In particular, the DEMUS [24] or Tera MTA [20,25,26] architectures, with modifications we describe below, serve as suitable models for our platform. A block diagram of the MCA processor is given in Figure 3.2. In the ensuing discussion, we refer to several aspects of this architecture and define the purpose of each of MCA's registers.

The most important distinction between MCA and conventional single context processors is in the Task State File block in Figure 3.2. Instead of a single PC (Program Counter), PSW (Program Status Word), and Instruction Register (or Instruction Queue), the MCA architecture contains multiple sets of task state registers.
The processor is designed to allow dispatching to each execution unit (EU) one instruction from one stream every clock cycle. Figure 3.2 shows a single EU, but the processor can have additional EUs. Since problems may be encountered when the same stream is chosen in consecutive cycles, the fine-grain scheduler or stream selection unit (SSU), also shown in Figure 3.1, must avoid execution-related conflicts due to data and control dependencies [3,33,34]. Execution-related conflicts result when an operand required by one instruction is also being modified by another instruction currently being executed. A simple avoidance mechanism would involve selection of a new stream in each clock cycle, without repeating...
any stream currently in the execution unit. While this is relatively easy to implement, there are several problems which make this infeasible. First, this unnecessarily limits performance of individual streams. Second, this mechanism only avoids intra-stream conflicts. Inter-stream conflicts, which might occur if two streams are communicating or sharing data, will not be detected.

To prevent any conflicts, a stream should not be selected for execution if there are any data dependencies among instructions belonging to any stream, which is already in the execution unit. Further, in order to avoid inter-stream conflicts, information from instructions of other ready streams is needed. A stream is deemed ready for dispatch if the processor’s register-available logic does not detect any register conflicts with instructions previously dispatched, and the stream has not been disabled for any other reason. The SSU accomplishes this by accessing the processor’s stream state register (SSR), shown in Figure 3.3, to determine which streams, if any, are ready to issue.
The SSR consists of four fields: Priority (Pr), Status (St), Register dependency (Rd), and Instruction dependency (Id). The Pr field contains two bits which determine the protection level of the stream, with 0 being highest and 3 lowest, as shown in Table I.

**Figure 3.3 Stream State Register (SSR).**

**Table I. Stream Priority Levels**

<table>
<thead>
<tr>
<th>Priority Levels</th>
<th>Priority Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pr&lt;sub&gt;0&lt;/sub&gt;</td>
<td>0</td>
</tr>
<tr>
<td>Pr&lt;sub&gt;1&lt;/sub&gt;</td>
<td>0</td>
</tr>
<tr>
<td>Pr&lt;sub&gt;0&lt;/sub&gt;</td>
<td>1</td>
</tr>
<tr>
<td>Pr&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
</tr>
</tbody>
</table>

| 0 0  | Highest - Reserved for kernel, interrupt routines |
| 0 1  | High - Privileged tasks and device drivers       |
| 1 0  | Low - Server processes (memory manager, file system, etc.) |
| 1 1  | Lowest - User programs                           |
Higher protection level streams are selected by the SSU before lower protection tasks. For example, the kernel and interrupt routines run with protection level 0 while user tasks run in level 3. Protection levels are static and determined by the operating system when the task is initially loaded.

The St field uses eight bits, and records the high level run-time status of a stream. Exactly one of the eight bits is 1 at all times, the rest are 0. The effect each St setting has on the state of a hardware slot is shown in Table II.

<table>
<thead>
<tr>
<th>SSR St field settings</th>
<th>Resulting hardware slot stream state</th>
</tr>
</thead>
<tbody>
<tr>
<td>St₀ St₁ St₂ St₃ St₄ St₅ St₆ St₇</td>
<td>Empty (available)</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>Loaded and stream is ready</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>Waiting on IPC receive operation</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>Waiting on IPC send operation</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>Waiting on IPC send/receive pair operation</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>Blocked by coarse-grain scheduler (e.g., time quantum expired)</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>Blocked by hardware interrupt</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>Unused (by MCA), available for future use</td>
</tr>
</tbody>
</table>

Clearly, a stream can be issued only if the St₁ bit of its Status field is one.

The Rd and Id fields are each one bit fields and indicate pending register or instruction dependencies with instructions already issued in the execution pipeline. Rd and Id field
values are listed in Tables III and IV, respectively.

Table III. Register dependency (Rd) setting

<table>
<thead>
<tr>
<th>Rd value</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No dependencies among instructions in EU.</td>
</tr>
<tr>
<td></td>
<td>(e.g., hardware slot can be selected by SSU)</td>
</tr>
<tr>
<td>1</td>
<td>One or more dependencies exist</td>
</tr>
<tr>
<td></td>
<td>(e.g., SSU will not select instruction from this slot)</td>
</tr>
</tbody>
</table>

Table IV. Instruction dependency (Id) Setting

<table>
<thead>
<tr>
<th>Id value</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Next instruction will follow current, so IFU may fetch when ready.</td>
</tr>
<tr>
<td>1</td>
<td>Next instruction may not execute due to change in program counter by current or previous instruction, so next instruction must not be fetched.</td>
</tr>
</tbody>
</table>

The Rd field is used (along with St and Pr) to select which stream (hardware slot) to issue an instruction from. The Id field is used by the instruction fetch unit (IFU) to determine when to fetch the next instruction for a given stream. The setting and evaluation of Rd and Id fields is illustrated by the code fragment shown in Figure 3.4.
In this example, the first instruction modifies register R1. The next instruction, add, uses and modifies the same register. If the second instruction is executed before the first completes, there is a read after write (RAW) hazard. Hence the add must not be selected until the load has completed. Similarly, the conditional branch cannot be evaluated until the add is complete. The instruction dependency flag is set in the branch instruction because it will not be necessary to execute the move instruction if x evaluates to 0.

The preceding example also illustrates why the SSU must access *fully decoded* instructions residing in the decoded instruction file in order to set the Rd and Id flags. Instructions must be fully decoded so that indirect register references are already resolved, thus allowing the SSU to detect conflicts in the more general register access mechanism of the MCA processor. This is discussed in more detail later.

By disabling a stream's ready flag for a specified number of cycles, a simple mechanism can be established for a stream to stall, allowing for both hardware and compiler support for data dependency control. This limits the need for hardware pipeline interlocks. It also

<table>
<thead>
<tr>
<th>C code</th>
<th>MCA instructions</th>
<th>Rd</th>
<th>Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=w+y;</td>
<td>Load R1,w</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Add R1,y</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>if (x)</td>
<td>Jzero label</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a=0;</td>
<td>Move 0,a</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>label:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 3.4 Illustration of register and instruction dependencies.*
reduces the amount of state information which must be saved when a full context switch is required. Further, information contained in an instruction can be used to control subsequent instruction fetching in order to reduce instruction memory traffic. A similar approach to stream control has been adopted in DEMUS and the Tera processors, using "explicit-dependence lookahead" [26]. In Tera, each instruction contains a three bit lookahead field specifying how many subsequent instructions may be issued before a dependency would be encountered. The size of this field is dependent in part on the depth of the execution pipeline. We propose a single bit lookahead mechanism in MCA. This is also a flexible approach with simpler implementation and smaller instruction space requirement. In this case, a reservation table approach [35] is applied to instructions during compilation. Any conflicting instruction that is in the execution unit sets the Rd bit to 1 in the present instruction. In this case, the lookahead is done at compile time only, and is depth dependent. The only drawback is that compiler support is needed to fully utilize the dependency feature since hardware conflict resolution is not used.

Unlike many of the processors discussed previously, a shared-register multistream processor cannot issue an instruction to the execution unit until register availability is verified. The status of a process's registers cannot be verified until immediately prior to execution for the following reasons: Registers may be deallocated from one stream and allocated to another stream between successive dispatch of instructions. Also, registers may be shared by two or more streams or temporarily locked by another executing instruction. In the case of interprocess communication, registers are transferred between
the address space of one process and the address space of another. Flexible register access is facilitated by a two-level register window access mechanism suggested by Killeen et al. [5,35], shown in Figure 3.5.

The window access scheme of Figure 3.5 works as follows: The register address field of an instruction consists of two fields: window number and window offset, respectively. The window number field accesses a process register, which we call the window access register (WAR), containing the physical register number of the first register in a window. Window offset is appended to this address as a means of selecting a particular register within the window. Window number is W bits wide, and window offset is O bits. This allows an instruction to uniquely address $2^W$ WAR entries and $2^O$ registers within each window.

![Figure 3.5 Register windows and two level register access mechanism.](image-url)
Generation of the effective register address is performed during instruction decoding. This is necessary for two reasons. First, since window decoding requires a register access, it cannot be done at the same time as operand fetch without extending the clock period. Second, dependency checking by the SSU can only be accomplished after effective addresses are determined. Dependency checking is necessary across all active streams in order to guarantee that a desired register is available, as well as enforcing mutual exclusion.

The SSU will not select a stream with effective register address conflicts with any instruction already committed to the Execution Unit (EU). Dependency checking is accomplished with the addition of a Window Tag Register (WTR), shown in Figure 3.6.

\[
\begin{array}{c|c|c|c|c}
Pw_0 & Pw_1 & & & Pw_{N-1} \\
\end{array}
\]

Figure 3.6 Window Tag Register (WTR).

This register contains a single bit for each physical window access register in the MCA processor. The bit for a window is set when an instruction is issued which requires access to a register within the particular window. When the instruction exits the EU, the bit is cleared again. Dependency checking, therefore, for a stream is a simple exclusive-or operation of all WAR entries against the WTR. A non-zero result indicates a conflict and the Rd flag in that stream’s SSR register is set. This can be done in parallel for all WAR
entries. Thus, dependency checking can be accomplished in a single clock cycle.

3.3 Register Access and Interprocess Communication

There are important differences between the MCA method of windowed register allocation and access mechanism found in typical processors utilizing register windows [36, 6-11]. In these processors, windows are allocated and returned on procedure call, in the form a single dynamic window. This operates as a stack of registers, and does not allow a general allocation mechanism or exchange of registers between two processes. As a result, each process has to have its own contiguous register file, disjoint from all others. Only a limited form of exchange of registers between consecutive procedure calls is possible through overlapping two windows. We propose a number of visible windows for each process, with a dynamically relocatable register allocation scheme rather than the register stack approach.

The multiple window register allocation scheme proposed supports random access to a large register file using a limited number of instruction bits, while the SSU must guarantee mutual exclusion. The WAR mechanism of Figure 3.5 allows indirect register access, which in turn makes it possible for the hardware to perform interprocess communication. This is accomplished by exchanging WAR entries between two resident contexts. IPC at the application level is illustrated by the example high level language (C) code fragment shown in Figure 3.7.

In this example, sendrec, receive, and reply are C language macros used by the compiler
to format the corresponding MCA assembly language instructions, which are described in more detail in Chapter 4. The user task must first make two system calls before initiating communication with the server process. The function \texttt{register} allows a process to establish a custom public key, \texttt{as}, using a user chosen identification name, \texttt{me}. The value, \texttt{as}, uniquely identifies an MC/OS task for IPC purposes. This identifier serves as a public key which other tasks can use to address IPC messages. The second call, \texttt{get\_task\_id}, requests the public identifier of the server task. The variables, \texttt{m} and \texttt{msg}, are the locations, respectively, of the source message and destination message buffers. In MCA, these correspond to a window of registers specified by an entry in a WAR register of each task.

An IPC operation consists of transferring the contents of the source WAR register (m) into the destination WAR register (msg).

For this example the IPC proceeds as follows. The client tasks issues a \texttt{sendrec} instruction. If the server is not already waiting, the client task blocks. The block is performed by the SSU which set the St$_4$ bit of the SSR for the task to 1. Eventually, when the server attempts the receive operation, the task will be unblocked, the SSU will set the St$_1$ bit to 1, allowing the IPC to proceed. If the server is already waiting, the client task does not need to block, the IPC operation is performed immediately.
Client Task Code Sample

... as=register (me); /* register my OwnId with system */
who=get_task_id(server_name); /* get OwnId of server */
... /* prepare message */
sendrec (&m, who);
... /* examine & process result */

Server Task Code Sample

... receive (&msg, who); /* places OwnId of sender in who */
... /* process request */
... /* prepare reply */
... reply (&msg, who);
...

Figure 3.7 Sample interprocess communication between processes.

IPC is accomplished by copying the contents of an WAR entry from the sending task into a WAR entry of the receiving task. This process is illustrated in Figure 3.8.
The operation which copies one register window into the window access register of another process must be implemented as an atomic instruction for efficiency and protection. Obviously, register sharing is possible only if more than one process has access to the same physical window of registers. This means that two different forms of window copy are necessary, one which leaves the source WAR entry intact (for shared access), and another...
which invalidates the source WAR entry (exclusive access). Mutual exclusion and race conditions are controlled by SSU by blocking tasks whose instructions require registers currently in use by another task. A further extension would allow separate read, write, and read/write access rights on a per process basis with shared windows. In the simulation of the MCA architecture, we have restricted window access to single tasks without any register sharing feature. This simplifies the operating system and architecture and still yields good experimental results.

As described in the preceding example, independent tasks communicate in MCA by exchanging access to register windows by transferring the contents of one WAR (window access register) entry in the sender's state into a WAR register in the receiver's state, as depicted in Figure 3.8. For this operation to succeed, however, it is necessary for both sender and receiver to know how to address one another. If we wish to allow a task to receive messages from any other task, which is essential in client/server and operating system task implementation, then at least the sending task must know where to find the receiving task.

A number of solutions to this problem have been explored within the context of operating and distributed system design [18]. The solution we chose in MCA involves maintaining a table in memory identifying each task as it is created. In essence, tasks register themselves with some publicly available label, and in return are assigned a global identification or key. A task wishing to communicate with another task can simply request
the other's identification from the kernel. Once the identification is known, a message can be sent by supplying the receiver's key and the message itself. To make it possible for a server to receive request messages from a number of other tasks, the sender should include its own key as part of the message. The message exchange itself is normally carried out in software by the OS kernel. It has been pointed out previously that this is secure, but inefficient, due to system (kernel) overhead needed to copy the message and manage context switching between tasks. We chose to incorporate support for IPC into the MCA design. This improves efficiency as well as security since IPC are atomic.

Hardware managed IPC is accomplished in MCA with the addition of a Task Name Register Table (TNR), as shown in Figure 3.9. MCA has a single global TNR file, organized as a fully associative memory. The TNR file has one entry for each hardware context, and consists of three fields: OwnId, WholId, and WARnum. OwnId is an I bit field, containing the OS assigned key of the task currently resident in the corresponding MCA hardware context. WholId, also I bits wide, contains the assigned key of the task to be sent to or received from. WARnum contains the WAR register whose contents are to be exchanged. OwnId is loaded as a task is made resident, and is only addressable by the kernel. WholId and WARnum are set for each send/receive request, and are indirectly modifiable by all tasks.

The number of TNR slots, \( S \), shown in Figure 3.9 is a variable under investigation in this research. Obviously, \( S \) should be large to allow as many context entries as possible,
perhaps even more than the number of hardware contexts. This could be accomplished with the addition of a resident/swapped field in the TNR. However, the size and complexity of the needed associative memory should be kept to a minimum.

Refer again to the example of Figure 3.7. The sequence of steps for a task attempting to send a message works as follows: First, each task should issue the system call register to obtain its public IPC key. This is the OwnId field of the TNR table. Next, a call to `get_task_id` is made to obtain the key of the destination task. In the example, `server_name` is the name of the destination task. Assume the client task of Figure 3.7 is resident in hardware context slot 0, while the server task occupies slot 1. In this case, the client is OwnId=123, while the server is OwnId=456. The call to `get_task_id` would return the value 456 to the client task (e.g., who=456). Since the client knows the OwnId of the server, it is now possible for the client task to initiate communication with the server.
Next, the client task uses a send instruction specifying its source WAR in m (where the message is stored, m=11) and destination key (e.g., who=456).

During instruction decode these values are placed in the WARnum and WhoId fields for the task, respectively. An associative read request is then performed on the TNR, looking for two entries: An OwnId matching the WhoId field of the sending task, and a WhoId field matching the sender’s OwnId. In this example, the sending tasks’s OwnId of 123 would be matched against a receiving task’s WhoId field, while the WhoId value of 456 is matched against a receiving task’s OwnId. A match occurs only if both fields match for a given hardware context slot entry in TNR. In this example, hardware context slot 1 matches.

If a match is discovered and the matching hardware context slot’s Rd field of the SSR is not set to 1, and the St\textsubscript{2} field bit for that slot is also 1 (e.g., blocked on a receive operation), then the desired destination task is waiting the IPC so it can proceed. In this case, the send instruction issues (executes) resulting in:

A. WARnum entries are extracted from the WARnum fields of both the sender and receiver, and the corresponding WAR register contents of the sending task are transferred into that of the receiving task, as shown in Figure 3.7;

B. The WAR entry (11) of the sending task is invalidated (because we do not allow shared
resources at present);

C. The St_t bit of the SSR of the receiving task is set to 1 (ready and running);

D. The receive instruction for the receiving task issues as a NOP (no operation). In this example, the contents of WAR entry 11 of the client task are copied into WAR entry 5 of the server task, and WAR entry 11 of the client is invalidated. This prevents the client from accessing the message after it is sent.

If no match is discovered, the St_3 or St_4 field of the SSR of the sending task is set to 1, marking the context as blocked on a send or send/receive, and no instruction issues. This blocks the task until a receive instruction is issued by the desired task in the future.

A receive operation works similarly to the send operation described above with one exception. A task can specify that a receive from any task is acceptable. This is denoted by a special WhoId value, 0. In this case all tasks, if any, blocked attempting to send will cause a match and MCA will arbitrarily select the highest priority (ready) task to perform the IPC operation with. An arbitrary receive is necessary to allow client/server communication similar to the one depicted in Figure 3.7. In this case, the server has no way of knowing in advance who to expect a message from.

Note that copying WAR entries in this manner works automatically in the case where either
sender or receiver has blocked and been swapped out of the processor by the coarse-grain scheduler. When the task is reloaded, the blocked instruction is evaluated normally, and IPC proceeds. It does, however, require coordination between the hardware and coarse scheduler. The scheduling software must have the ability to query the TNR table to look for matching pairs. This naturally increases the complexity of the coarse scheduling routine, and requires additional hardware instructions dedicated for this purpose. These are described in the next chapter.

In MCA, the method of IPC has two important characteristics. First, both tasks must be resident for IPC to proceed. This creates a practical lower bound of three for the number of hardware contexts in an MCA processor: Two for IPC operations between tasks, and a third for the OS kernel, which must always be resident to be able to process hardware interrupts. Both tasks must be resident because WAR entries are exchanged as an atomic operation, as illustrated in Figure 3.8. While this is safer and simpler to implement, it requires that one processor slot be blocked for an IPC operation to complete. As a result, since some hardware slots are expected to be blocked awaiting IPC, the processor will have fewer slots to choose from when selecting instructions to issue. The number of blocked slots is application dependent and is part of the simulation investigation presented later. The intuitive result, however, is that more hardware slots are required to keep the processor fully utilized.

The second characteristic is that IPC is fully synchronous. As with the first case, this is
simple to implement and is the minimum required to support all forms of IPC. Asynchronous IPC, if desired, could be implemented in software as a server task, but would naturally incur higher overhead.

3.4 Fine-grain Scheduling

The MCA processor incorporates two levels of scheduling. A fine-grain scheduler, implemented in hardware, is responsible for issuing instructions from ready streams for execution. The coarse-grain scheduler is part of the operating system kernel and is responsible for loading and unloading hardware streams so that the processor has an adequate supply of ready streams to schedule. The goals of the MCA processor schedulers (combined) are: (1) efficient processor utilization; (2) fairness and starvation avoidance; (3) high single thread performance; and (4) good interactive response time.

Achieving high processor utilization over a short period is the responsibility of the hardware scheduler (SSU). Methods to achieve this have been studied extensively [28,17,26,36,37] and will not be repeated here. Similar to other finely threaded processors, the MCA processor is designed to be able to switch between ready streams on alternative clock cycles. In order to achieve good single thread performance, however, it is necessary to execute instructions within a single stream for extended periods, only switching streams on fairly long-latency events.

The MCA processor incorporates a dual mode hardware scheduling mechanism. In default
or block mode, MCA will continue to execute the same stream until it blocks for some reason, then a new stream is selected. In cycle mode, MCA switches between streams on every cycle. As noted by Laudon et al. [36] cycle mode involves greater hardware complexity. This effectively reduces the number of hardware contexts the processor can support on a single chip. Thus, it is unclear at this point whether the added hardware complexity is worth the cost. This is especially true given advances in the ability of compilers to extract parallelism and detect dependencies within conventional sequential code. Throughout the remainder of this study we focus on block mode scheduling and determine its effect on the MC/OS coarse-grain scheduler. One exception to dual mode operation involves the zero, or kernel, slot. If slot zero is selected, the SSU disables cycle mode. This guarantees that the kernel will be able to execute uninterrupted until it chooses to give up the processor to other tasks.

Detection of the ready state of a stream has been discussed previously. In block mode, the MCA processor continues to execute the current stream until the stream blocks. When a block occurs, the SSU switches to another ready stream, selecting the highest priority resident stream. In the event that no stream is ready, the system is considered deadlocked, and the SSU switches to stream zero, which always contains the MC/OS kernel task. This process is illustrated by the flow diagram in Figure 3.10.
Figure 3.10 Stream selection flow diagram.

This flowchart depicts the close interaction between the SSU hardware and coarse-grain
scheduler software of the MC/OS kernel task. The left hand side of the flowchart corresponds to SSU actions, performed in hardware. The right side represents operations done by the kernel.

Hardware interrupts also cause an immediate switch to the kernel stream. Several status registers within MCA record the current state of the processor. The currently executing stream is stored in CSTREAM. This is moved into ISTREAM on an interrupt so that the SSU and/or kernel can switch back to this task after the interrupt is serviced. When a stream blocks, CSTREAM is copied into BSTREAM. The contents of this register are used by the coarse-grain scheduler when a system deadlock occurs. Under ideal circumstances, the kernel task is rarely run except for normal interrupt handling, and the SSU switches between resident, non-kernel, streams. In the event that a single stream monopolizes the processor, the interrupt handler that handles the real time clock can force a hardware context slot to block by setting the SSR status bit St₈ to 1. Eventually, the coarse-grain scheduler must decide to unblock the stream, or swap it out in favor of another task. This helps ensure the fairness criteria of the MCA scheduling policy.

The MCA processor contains a set of status registers, shown in Figure 3.11, collectively referred to as Task Activity Registers (TAR) to support both fine and coarse scheduling as well as system accounting. MCA maintains one TAR entry for every hardware context. Each TAR entry consists of four distinctly maintained registers or fields, called Sints, Ssw, Sipc, and Sact registers. The contents of these registers are as follows. Sints (Stream
Instructions) maintains a count of the number of instructions issued, and is incremented for each instruction issued for a stream from the hardware context slot in which it resides. **Sints** reflects the amount of CPU time a stream has received. This can be used by accounting to assign usage costs, or by the coarse-grain scheduler as a time quantum measurement.

<table>
<thead>
<tr>
<th>Sints</th>
<th>Ssw</th>
<th>Sipc</th>
<th>Sact</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>10</td>
<td>200</td>
<td>75</td>
</tr>
<tr>
<td>2500</td>
<td>20</td>
<td>150</td>
<td>100</td>
</tr>
</tbody>
</table>

**Figure 3.11 Task Activity Register (TAR) Set.**

The **Ssw** (Stream Switches) register records the number of times a stream is selected, and is incremented on each SSU switch to the context. In cycle mode, **Sints** is equal to **Ssw** since a new stream is chosen each clock cycle. In block mode however **Ssw** will be smaller than **Sints**. The **Sipc** (Stream IPC) register is similar to **Ssw**, recording the number of IPC operations performed by a stream, and is incremented for each successful send or receive operation. Either of these values can be used to provide statistical histogram information.
to the coarse-grain scheduler to assist in scheduling tasks more effectively. The use of the **Sipc** register is discussed in Chapter 5, Section 5.3, where IPC driven scheduling is presented.

The last register, **Sact** (Slot Activity), is manipulated directly by the coarse-grain scheduler for use in long term scheduling. In essence, depending on the scheduling algorithm in use, one (or possibly more) of the **Sints**, **Ssw** or **Sipc** registers contents are merged with the **Sact** contents. The resulting value is then used to which contexts should be switched in and out when a full context switch operation is required. This is referred to as a scheduling cycle, and is covered in more detail in Chapter 5. The TAR registers are set/cleared whenever a stream is loaded, and as part of a scheduling cycle.
Chapter 4

MCA Instruction Set Extensions

This Chapter describes the instruction set architecture of the MCA processor. We only describe assembly language extensions which can be added to an existing instruction set such as the 80x86. The added instructions are those required to support multiple resident contexts, hardware supported interprocess communication, and support for fine and coarse grain scheduling (CGS). In our simulation study, for example, these instructions were added to the 80x86 instruction set to simulate an 80x86 based MCA processor. These instructions are grouped and presented according to the class of action supported.

4.1. Context Management Instructions

The primary function of this group of instructions is selection, swapping and state control of hardware contexts, or slots. We use the term slot to avoid the confusing term context which is often used to describe both hardware (hardware contexts) and software state information (software contexts). A hardware slot is, therefore, unambiguous. The group of context management instructions is listed in Table V, and explained below.
### Table V. Context Management Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld_St</td>
<td>Slot#</td>
<td>St_val</td>
<td>Set St register of slot# to St_val</td>
</tr>
<tr>
<td>Ld_Who</td>
<td>Slot#</td>
<td>Who_val</td>
<td>Set WhoId of slot# to Who_val</td>
</tr>
<tr>
<td>Ld_Own</td>
<td>Slot#</td>
<td>Own_val</td>
<td>Set OwnId of slot# to Own_val</td>
</tr>
<tr>
<td>Ld_WARN</td>
<td>Slot#</td>
<td>WAR_val</td>
<td>Set WARnum of slot# to WAR_val</td>
</tr>
<tr>
<td>Get_St</td>
<td>Slot#</td>
<td>St_val</td>
<td>Read St register of slot# into St_val</td>
</tr>
<tr>
<td>Get_Who</td>
<td>Slot#</td>
<td>Who_val</td>
<td>Read WhoId of slot# into Who_val</td>
</tr>
<tr>
<td>Ld_WAR</td>
<td>Slot#</td>
<td>WAR_val</td>
<td>Load WAR register of slot#</td>
</tr>
<tr>
<td>St_WAR</td>
<td>Slot#</td>
<td>WAR_addr</td>
<td>Read WAR reg. to WAR_addr</td>
</tr>
<tr>
<td>ClrSlt</td>
<td>Slot#</td>
<td></td>
<td>Mark slot# as unused (St=0)</td>
</tr>
<tr>
<td>SwSlt</td>
<td>Slot#</td>
<td></td>
<td>Switch to Slot# explicitly</td>
</tr>
</tbody>
</table>

**Ld_St, Ld_Who, Ld_Own, and Ld_WARN.** These are the first four instructions used by the kernel to load context state information in the SSR (Figure 3.3), and TNR (Figure 3.9) when performing a full context switch. When loading a new context for the first time, only Ld_St and Ld_Own are required since the process cannot yet be blocked on IPC send or receive. The addressing mode is immediate or direct, so the complete contents of either SSR or TNR is either found in the instruction operand, memory or in a register specified by St_val.

**Get_St and Get_Who.** The next two instructions are used for context save and duplication (e.g., fork system call) and modify the contents of the SSR register (Figure 3.3).
As above addressing modes supported are immediate and direct for St_val and Who_val.

**Ld_WAR and St_WAR.** These instructions are used for context switch save, allowing the kernel to save and restore WAR entries (see Figure 3.5). In addition, these instructions are used for register file window allocation and deallocation.

**ClrSlt.** ClrSlt is issued when a context slot becomes available, and simply sets the St₀ field of the SSR for the slot to 1 (unused, see Figure 3.3). This prevents the SSU from selecting or manipulating that slot’s registers. ClrSlt is issued by the kernel system task when a process terminates.

**SwSlt.** SwSlt can be used to directly circumvent the SSU hardware selection mechanism to immediately switch between slots.

Since each of these operations directly affects the state of processor slots, they are reserved and can only be executed by streams executing at protection level 0 and 1. A higher protection level results in an invalid instruction trap.

### 4.2 Scheduling Instructions

MCA instructions that support fine and coarse scheduling are listed in Table VI. This group of instructions is designed to allow the operating system’s scheduler to determine which context(s) should be copied out to memory, and which non-resident software
context(s) should be made resident when a full context switch is needed.

**Release.** This instruction is used to give up the processor to the SSU using the normal SSU scheduling mechanism. The instruction is not reserved, and can be used by tasks at any protection level. However, Release is normally used by the coarse scheduler, slot zero, to return scheduling control to the SSU for normal execution of user tasks. If slot zero issues the instruction, the mode flag is unmasked (see Figure 3.2), and processing will revert to cycle mode if the mode flag is set accordingly. Release has no argument because it is used to trigger the SSU to take hardware scheduling action rather than manipulating processor register values.

**Table VI. Scheduling Instructions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release</td>
<td></td>
<td></td>
<td>Give up processor to another slot</td>
</tr>
<tr>
<td>SetMode</td>
<td>Mode</td>
<td></td>
<td>Normal SSU selection</td>
</tr>
<tr>
<td>ChkBk</td>
<td></td>
<td></td>
<td>Set SSU scheduling mode to Mode</td>
</tr>
<tr>
<td>RetSlt</td>
<td></td>
<td></td>
<td>Return last slot# to block</td>
</tr>
<tr>
<td>UpdAct</td>
<td>Scale</td>
<td></td>
<td>Return last slot# interrupted</td>
</tr>
<tr>
<td>SetAct</td>
<td>Tar_type</td>
<td>Scale</td>
<td>Shift L/R all TAR(Sact) registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Copy TAR reg. into TAR(Sact),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Shift L/R Scale bits 1st - all slots</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type: 0=Sints, 1=Ssw, 2=Sipc</td>
</tr>
<tr>
<td>ClrAct</td>
<td>Slot#</td>
<td></td>
<td>Clear TAR registers for slot#</td>
</tr>
<tr>
<td>GetAct</td>
<td>Slot#</td>
<td></td>
<td>Return TAR(Sact) of Slot#</td>
</tr>
</tbody>
</table>

**SetMode.** SetMode is used to place the processor into block or cycle mode, respectively. It is reserved for protection level zero only.
**ChkBlk and RetSlt.** These two instructions allow the coarse-grain scheduler to determine what might have caused a context switch to the kernel by reading the contents of the BSTREAM and ISTREAM registers, respectively. This can happen for several reasons. First, a hardware interrupt results in a switch to the kernel. Further, the system can "deadlock" (refer to Figure 3.10). That is, no hardware slot is ready to execute. This can happen for three reasons: First, all slots are blocked awaiting IPC operations. Second, all slots are blocked awaiting register availability. Third, all slots are blocked with at least one expired time quantum.

If a slot was interrupted by a hardware interrupt, it can be reset to ready and the scheduler can issue a Release immediately. ChkBlk, on the other hand, can be used to see whether the last running slot blocked on an IPC operation. If the last slot to issue an instruction blocked on an IPC operation, then all other contexts must also be blocked (or empty). Hence, a full scheduling cycle, and possibly full context switch, is needed. If the result of ChkBlk is false, the switch to slot zero was due to some other cause, in which case the scheduling decision(s) may be different.

An important distinction between our processor, and consequently the operating system, and other processors is that it is no longer possible to issue a software interrupt or trap to the operating system. All system calls must be implemented through explicit user level instructions. This completely isolates the operating system kernel from other software running on the system. This makes the kernel smaller and easier to write and evaluate since
there is a single interface class (hardware) into the kernel.

The remaining four instructions in this group access and/or modify the various TAR registers. Their function, starting with the simplest first, are as follows.

**ClrAct.** ClrAct simply clears all four TAR entries for the specified slot. This is issued when contexts are loaded so that scheduling data is reset for the slot just loaded.

**GetAct.** GetAct returns the TAR entry $S_{act}$ for the specified slot. This field is used by the coarse grain scheduling algorithm in MC/OS. The value returned represents an approximate least recently used value for whichever scheduling mechanism is employed in the scheduler. This is discussed in more detail in the next Chapter.

**UpdAct.** UpdAct is used to clear or modify the contents of the $S_{act}$ register for all processor slots. Note that ClrAct acts on a single slot, or horizontally, while UpdAct acts on all slots. This instruction can be used by the coarse scheduler to reset scheduling fields simultaneously for all resident tasks.

**SetAct.** SetAct is used to add the contents of one of the three other TAR fields to the $S_{act}$ register. As discussed in the next Chapter, this assists in coarse-grain scheduling decisions.
4.3 Interprocess Communication Instructions

The final group of extended instructions implements the IPC portion of the MCA processor. These instructions are listed in Table VII.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send</td>
<td>WhoId</td>
<td>WARnum</td>
<td>send WARnum window to WhoID</td>
</tr>
<tr>
<td>SendRec</td>
<td>WhoId</td>
<td>WARnum</td>
<td>send/rec WARnum to WhoId</td>
</tr>
<tr>
<td>Rcv</td>
<td>WhoId</td>
<td>WARnum</td>
<td>receive from WhoId into WARnum</td>
</tr>
<tr>
<td>ChkRcv</td>
<td>WhoId</td>
<td>OwnId</td>
<td>Is WhoId blocked sending to OwnId</td>
</tr>
<tr>
<td>ChkSnd</td>
<td>WhoId</td>
<td>OwnId</td>
<td>Is WhoId blocked Rcv from OwnId</td>
</tr>
</tbody>
</table>

Send, SendRec, and Rcv. The first three instructions are the hardware equivalents of conventional software implemented system call IPC operations. They cause the contents of the source (e.g., task issuing Send or SendRec) task’s WAR register to be transferred into the WAR of the destination task (e.g., task issuing second half of SendRec or Rcv), depicted in Figures 3.8 and 3.9. These instructions are guaranteed indivisible for the task issuing the instruction. That is, while the operation is performed, no other task can access the WAR register’s affected by the instruction. System call implementations of these instructions are also atomic. However, since this is accomplished in hardware as a single instruction, they are considerably more efficient because software, or indirect, managed mutual exclusion is not required. A typical IPC code sequence was given in Chapter 3, Figure 3.7, and is repeated here as Figure 4.1.
Client Task Code Sample

... 
as=register (me); /* register my OwnId with system */
who=get_task_id(server_name);/* get OwnId of server */
...
/* prepare message */
sendrec (&m, who);
...
/* examine & process result */

Server Task Code Sample

... 
receive (&msg, who); /* places OwnId of sender in who */
...
/* process request */
...
/* prepare reply */
...
reply (&msg, who);
...

Figure 4.1 Sample interprocess communication between processes.

In this example, sendrec, receive, and reply are C language macros used by the compiler to format the corresponding MCA instructions in Table VII. The user task, client, must first make two system calls before initiating communication with the server process. The function register allows a process to establish a custom public key, as, using a chosen identification name, me. The value, as, uniquely identifies an MC/OS task for IPC purposes. This identifier serves as a public key which other tasks can use to address IPC messages. The second call, get_task_id, requests the public identifier of the server task.

At present, MC/OS does not facilitate selective public keys. That is, once a task registers its identification, anyone knowing that registration may request the public key. While this
permits possibly unwanted tasks from communicating with servers, it is not difficult for a server to filter this out. The OwnId value is copied into the first physical register of the window transferred during all three hardware IPC operations. Hence, the server automatically receives the sender's OwnId as part of the Rev instruction. The server task is then free to reject unwanted requests. This feature of Rev has the added advantage of preventing one task masquerading as another.

**ChkSnd and ChkRev.** The last two IPC instructions are actually used by the kernel's coarse scheduling routines. These instructions allow a task to query other resident tasks to see if one is attempting a specific IPC operation. This is performed by the scheduler for IPC driven scheduling, and will be discussed further in the next section. In addition, critical MC/OS tasks can use this command to see if an IPC operation can be immediately satisfied. If the operation cannot be satisfied, the task can delay the IPC rather than block. One limitation of the instruction is that it only queries resident tasks, not those swapped to memory. A complete query of all tasks requires a (more expensive) system call to the MC/OS system task.

The next Chapter describes the MC/OS operating system, its organization and how it uses the instructions of the MCA processor to support IPC and perform effective long term scheduling.
Chapter 5

MC/OS Operating System Design

This Chapter describes the prototype operating system we have developed to execute on the MCA processor, named the Multiple Context Operating System (MC/OS). Chapters 3 and 4 present architectural features that can be used to efficiently support task scheduling and interprocess communication. In order to take full advantage of these features, which are normally implemented in software, it is necessary to develop software support as well. This provides a level of abstraction from the hardware, allowing programs to be written independent from the hardware architecture.

MC/OS is a multi-user distributed version of the UNIX operating system, derived from Minix 1.7.2 [18], whose kernel has been re-written for the MCA. We first present an overview of the MC/OS organization. Next the important pieces of the nano-kernel and kernel tasks, which support IPC, context management and long term scheduling, are described in detail. Lastly, we discuss several possible alternative scheduling algorithms and their effect in the MCA environment.

5.1. MC/OS System Organization

MC/OS is a highly distributed operating system designed to take advantage of the IPC and context management features of the MCA processor. MC/OS utilizes a nano-kernel
organization. That is, there is very small portion of the operating system, which is responsible for interrupt handling and process management runs as the lowest level, or nano-kernel, task. Other functions, which are normally considered part of the kernel, execute separately while sharing a global data area in memory. A general organizational model of the MC/OS structure is shown in Figure 5.1. MC/OS is similar to other UNIX based operating systems in that it supports the concept of a process. In addition, MC/OS also extends a process to include the notion of multiple context spaces for processes, or multiple threads of control. Threads can exist both within a common address space or they can maintain separate address spaces.

![Figure 5.1 MC/OS organizational structure.](image)
A context in MC/OS is essentially the same as a process in UNIX in that it has an address space, a set of registers, a program counter, and a stack. In effect, a collection of contexts in MC/OS is similar to a collection of independent processes in UNIX with the exception that contexts may share common address spaces, as depicted in Figure 2.2, and more than one context is normally resident and running within the processor. An example of a set of contexts sharing a common address space is the MC/OS kernel itself, as shown in Figure 5.1. The kernel consists of several parts: a nano-kernel and a collection of independent contexts sharing a global process table structure. To the user, however, this appears as a single large program providing system resource management.

MC/OS is roughly divided into four protection layers. The lowest protection layer comprises the nano-kernel which consists of process/context management routines and interrupt routines. These run with hardware protection level 0, allowing full access to all privileged MCA instructions and internal registers. The next higher contains the kernel driver tasks, which are responsible for processing hardware device commands, such as disk access or real-time clock control. These run with hardware protection level 1. This also allows use of privileged instructions, but is a lower (hardware) scheduling priority than level 0. The third protection level contains important user-level servers. The three most important servers are the memory and file system managers and IPC name registry server. Since these run as user-level processes, servers can be added and removed according to requests placed on the system. Moreover, because they are not part of the kernel, servers can be added and removed without restarting the entire system. This is important for fault-
tolerance and reliability. Finally, the highest protection level is populated by user programs (e.g., shell processes, user commands, etc.). The four layers correspond to the four protection levels supported by the MCA processor as described earlier. Server processes and user processes can be designed to run with hardware protection level 2 or 3. The choice is made depending on scheduling requirements. An interactive application might be written to run as a level 2 task in order gain higher scheduling priority, and hence better response time.

The nano-kernel and one kernel level task, the Coarse Grain Scheduler (CGS), are unique in that they share the same context space, executing in the privileged context slot (slot zero). However, they run as separate tasks in different protection levels. We describe this in more detail in the following sections.

All other contexts, both kernel and user, run in the remaining unrestricted MCA slots, and are allowed to freely migrate between slots. The CGS task is responsible for slot and context management.

5.2. Nano-Kernel Structure

The MC/OS nano-kernel consists of two basic sections: a collection of slot management routines and interrupt handling routines. The nano-kernel runs only in slot zero (privileged), and has three primary functions: system initialization, hardware interrupt processing, and hardware context slot management. A primary function of context slot
management is long term task scheduling.

System initialization is the only time the nano-kernel is executed (as a task). It is responsible for creating the global process table used by kernel tasks, loading and initializing important kernel tasks into MCA hardware slots, and switching to the first kernel task. After this initialization, nano-kernel functions are called only from certain interrupt routines and kernel tasks (the real-time clock task and the coarse grain scheduler which calls shared nano-kernel routines). Since the MCA processor does not support software interrupts, no other entry into the nano-kernel is supported.

Hardware interrupts are handled in the MC/OS in much the same manner as in other operating systems. In MCA, control switches from the current context slot to the privileged slot (slot zero), and then to a normal interrupt procedure. This consists of saving the current slot state, loading the nano-kernel context, branching to the appropriate interrupt routine, processing the interrupt, and switching back to the task executing in slot zero. Since this is always the CGS task, all interrupts lead back to the scheduler temporarily. Normally, the CGS task resumes the interrupted task. However, if the situation warrants, a full scheduling cycle is initiated.

Slot management routines in MC/OS are shared by the nano-kernel and CGS task. To prevent the two separate tasks from simultaneously manipulating hardware slots, a simple semaphore system is employed in critical regions. Context management consists of the five
program procedures in Table VIII, which does not include the scheduling task.

Table VIII. Context Management Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>pick_slot</td>
<td>Find a hardware context slot &amp; load with a context</td>
</tr>
<tr>
<td>pick_proc</td>
<td>Search list of active contexts for one to make resident</td>
</tr>
<tr>
<td>ready</td>
<td>Add a context to active (ready) list</td>
</tr>
<tr>
<td>unready</td>
<td>Remove a context from active list</td>
</tr>
<tr>
<td>sched</td>
<td>Remove a context from hardware slot, find &amp; load another</td>
</tr>
</tbody>
</table>

**pick_slot.** This is called whenever a context is to be loaded into a hardware slot. The syntax for the pick_slot function is

```
pick_slot (pid);
```

where pid is a pointer to the entry in the process control block (PCB) table containing the software context which is to be made resident. The hardware slot the task is loaded into is stored in the slot number field of the task’s PCB entry if the call is successful. If the call fails, the slot number field is invalidated. Accordingly, MC/OS maintains a data structure in the kernel’s global memory area, called mcpi, containing the current status of each hardware slot. This indicates whether or not a slot is empty. If occupied, the process id of the task is stored in the entry corresponding to that hardware slot. In addition, the hardware slot a context resides in (if any) is also stored in the PCB entry for the task. Thus, the current state of the system can be determined in two ways. Pick_slot examines the mcpi table to find the first available (unused) slot to load the context. The context is loaded, and
system tables updated to reflect this. If no slot is available, the function fails. At the time the context is loaded, the processor checks for any pending IPC operation and sets the St field of the newly loaded context's SSR register.

**pick_proc.** Pick_proc searches a list of active software contexts (i.e., non-resident entries in the PCB) to determine if one should be made resident. pick_proc takes no arguments, but returns a value indicating which software context PCB entry was made resident. The syntax for pick_proc is

```
entry=pick_proc();
```

Selecting a task to make resident is more difficult on the MCA processor than in a typical single context processor. The approach used in many operating systems, called round-robin scheduling, is to maintain several queues of tasks, one or more for runnable tasks, the ready queue(s), and one or more for blocked tasks. Choosing the next task to make resident involves selecting the first task in the highest priority ready queue. Since there is a single resident context and IPC is implemented in software, the status of all tasks in the system is easily maintained by examining the one resident context each time a full context switch is required. In the multiple context environment of the MCA processor, however, the full status of the system can only be determined by examining all resident contexts, which is a potentially time consuming task. In addition, the best context may not be a runnable context, but rather one which is IPC blocked. That is, a task which is blocked on an IPC operation with some resident task which is also blocked attempting an IPC operation with the non-resident task. Hence, during a full scheduling cycle the queue of blocked tasks may
need to be examined. For this reason MC/OS maintains a single list of all active, non-resident, contexts called the *active* list. The list is maintained by using a field in each PCB entry to link tasks together in a simple linear list.

A context is a candidate for loading if it meets one of the following criteria:

1. It is runnable (i.e., *St* flag entry, stored in PCB entry, is 1).
2. It is blocked on an IPC operation and the task it wishes to communicate with is also blocked on a *matching* IPC operation (e.g., trying to reciprocate the IPC).

The first criterion is obvious and requires no additional explanation. The second criterion can be met in two distinct ways. Either the reciprocating context is resident in the processor or it is non-resident and resides in the active list. If one is resident and the other not, then loading one context will allow the IPC operation to proceed, making both tasks runnable. If neither is resident, then both must be made resident before the IPC operation can complete. This makes pick_proc more complicated because a search in two different places (e.g., the processor or the active list) is needed to determine whether a matching context exists, and if so, whether it is resident or not. It requires fewer instructions to check for a matching IPC blocked context that is resident than one in the active list. A single instruction determines the former, while a software loop to scan the active list is needed to determine the latter.

Pick_proc uses a two level task priority in addition to the *St* flag to pick a context to load.
The highest priority tasks are kernel tasks and all other tasks are user tasks with lower priority. The best candidate is defined to be the highest priority candidate based on the two criteria given above. This allows the routine to exit immediately once a candidate kernel task is found, making pick_proc execute as quickly as possible.

**ready.** When a task is first created, an entry is made into the process control block, the entry is marked as runnable by setting the St flag in the task’s PCB entry to 1, and the task as added to the active list. No value is returned by ready. The syntax of ready is

```c
ready(pid);
```

where pid is a pointer to the PCB entry of the task. When a context is made resident, or terminates, it is removed from the active list. Contexts swapped out of the processor are added back onto the active list. Runnable contexts, regardless of priority, are added to the front of the active list, blocked contexts at the rear. This approach is simple, yet inactive contexts migrate to the rear of the active list, and are only examined if no high priority task is available. We discuss the results of this approach in more detail later.

**unready.** Unready is called whenever a context is made resident, or when it exits. The syntax of unready is

```c
unready(pid);
```

where pid is a pointer to the PCB entry of the task. The context does not need to be put into any queue when it is made resident since that is recorded in mepi and the task’s PCB entry.
sched. Sched does not perform any scheduling within the system. It is called by CGS during a full scheduling cycle to perform the following operations:

1. Remove a (CGS) designated resident context, freeing a hardware slot.
2. Call pick_proc to load a different context into the newly available slot.
3. If no context selected and the removed context is runnable, reload the same context, else, add the removed context to the active list.

Note that due to the order of operations, the removed context will never be re-loaded if another context is available. This prevents a single task from monopolizing the processor, even if it has a high priority.

5.3 Kernel Tasks

Layered above the Me/OS nano-kernel are critical system tasks and device drivers, shown in Figure 5.1. All drivers and tasks are implemented as separate contexts sharing a common data space with the nano-kernel, similar to the organization of threads within processes shown in Figure 2.2. All tasks interact with each other through the exchange of messages using the MCA IPC mechanism. The global data space allows tasks to examine and update the PCB table information. The clock task, for example, uses PCB data for accounting and signal generation. Table IX lists the important tasks and drivers in this layer. In addition to these, there are driver tasks for each hardware device, including hard disk drive, floppy drive, Ethernet network card, sound card, etc. These are not discussed here because they function similarly to device drivers in other operating systems. In the following sections we describe the function of the three tasks shown in Table IX.
Table IX. Critical Kernel Tasks/Drivers

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGS system</td>
<td>Coarse grain scheduling</td>
</tr>
<tr>
<td>clock</td>
<td>Real-time clock functions</td>
</tr>
<tr>
<td>system</td>
<td>Support device independent system calls</td>
</tr>
</tbody>
</table>

5.3.1 Coarse Grain Scheduling (CGS) Task

The coarse grain scheduler, CGS, runs in the same address space and hardware slot as the MC/OS nano-kernel. The two tasks share the same data and context management routines, but perform different functions. Under normal operation, the CGS task does not start until system initialization is complete and the nano-kernel performs a conventional context switch (in hardware slot 0) to the CGS task. The scheduler promptly release the processor for normal hardware scheduling and, in either block or cycle mode, the MCA processor chooses a stream from one of the un-privileged (non-zero) hardware slots. Control returns to the scheduler if no other hardware slot is ready or a hardware interrupt occurs.

When a hardware interrupt happens the MCA processor sets the St<sub>6</sub> bit of the SSR register for the currently executing hardware slot is set to 1, indicating it is blocked by a hardware interrupt. The processor then changes to block mode, switches to slot 0, and processes the interrupt. On interrupt return, the processor still has slot zero selected, so control returns
to the CGS task as if it had been running when the interrupt occurred. In this situation a scheduling cycle is not always necessary since the interrupted task could be resumed, but scheduling may be reinitiated. This will happen, for example, when the time quantum expires for a user level task and the task should be swapped out in favor of another waiting context. If a full scheduling cycle is not needed, the status flag of the interrupted slot is cleared, and the processor is released for normal hardware scheduling.

The other entry into the CGS task happens if no other hardware slot is ready. In this case the processor switches to slot zero by default, but the mode is not changed. This will happen if all resident slots are either blocked or empty (e.g., the St1 bit of the SSR register is 0 for all hardware slots). In either case a scheduling cycle is needed to make one or more regular slots runnable again.

**Scheduling Cycle.** The CGS task performs a scheduling cycle when control returns to the task in one of the two ways described above. If there is no need for a scheduling cycle, the processor is immediately released again. In the case of a full schedule cycle the CGS tasks follows the following sequence of steps:

- Issue UpdAct instruction to establish new slot activity values.
- Examine hardware slots to determine least/most active slot.
- Swap the context in the selected slot to memory, freeing a slot.
- Call the routine sched to load a new context.
- Release processor to the hardware scheduler (SSU).

The UpdAct instruction, along with the contents of the Task Activity Register Table (see
A number of scheduling algorithms have been proposed. These include static priority queues, several of which use dynamic priority queues, and a mechanism called handoff scheduling employed in Mach [27]. Early versions of UNIX (e.g., SVR3 and earlier) used a multi-level round robin scheduling mechanism using static task priorities. More recent UNIX releases (e.g., SVR4) employ a more complex dynamic priority mechanism [27]. This decreases the probability of starvation (i.e., when a task is never executed due to the scheduling policy), and allows more flexible scheduling for critical tasks such as real-time applications. The Mach operating system refined scheduling further by allowing threads to explicitly release the processor to another thread. This is used to assist the scheduling software, reducing the time required to schedule threads. This is important because the time complexity of the scheduling portion of the kernel increases as the number of tasks grows, and rapid context switching is important in a distributed system [27]. We implemented and compared two scheduling mechanisms in the MC/OS CGS task. The first utilizes a multi-level round-robin algorithm. The second is a variation of the handoff scheduling, called IPC directed scheduling. These are described below.

**Round Robin Scheduling.** Most operating systems employ a round robin type scheduling mechanism [18, 45]. A task that has executed for a while, or blocked on a long latency operation, is placed at the end of a queue and the task at the front is loaded and
executed. This ensures fairness and lack of starvation since all tasks eventually get some processor time. The Sints field of the TAR register can be used to update time quantums for tasks, allowing preemption and rescheduling in situations where one task monopolizes the CPU.

Enhancements to increase response time for interactive tasks include the use of multiple queues, and dynamic priority assignment. Round-robin scheduling does not guarantee optimum response (to the user) or optimum processor utilization and overall throughput in a multiple context processor. To illustrate this, consider the following example shown in Figure 5.2. The figure illustrates the interactions among four tasks executing in a multiple context processor. In this example we specify a system with two hardware slots which may be used to schedule user tasks. Hence, this system can have simultaneously resident any two of the four tasks shown, while the other two must be swapped to memory.
In this example, we assume that context A is resident and has blocked while attempting an IPC operation with context B. The arrows indicate IPC activity between contexts, with the solid arrow indicating pending activity and the dashed arrow future activity. Hence, at some point context C will attempt IPC with context D. Assume further that the contexts in a ready queue are ordered: C, B, and D, respectively, and as noted that the processor has two slots available for scheduling, including the one occupied by context A. A Round-robin scheduling algorithm will select the task at the front of the (highest priority) ready queue and make it resident in the second hardware slot. Thus, context C is scheduled ahead of context B, leading to the following possible scheduling sequence shown in Figure 5.3.
<table>
<thead>
<tr>
<th>Action</th>
<th>Status</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A IPCs B</td>
<td>A resident</td>
<td>A blocked</td>
</tr>
<tr>
<td>Load C</td>
<td>A &amp; C resident</td>
<td>A blocked, C running</td>
</tr>
<tr>
<td>C IPCs D</td>
<td>A &amp; C resident</td>
<td>A &amp; C blocked</td>
</tr>
<tr>
<td>Swap A</td>
<td>C resident</td>
<td>C blocked</td>
</tr>
<tr>
<td>Load B</td>
<td>C &amp; B resident</td>
<td>C blocked, B running</td>
</tr>
<tr>
<td>B IPCs A</td>
<td>C &amp; B resident</td>
<td>B &amp; C blocked</td>
</tr>
<tr>
<td>Swap C</td>
<td>B resident</td>
<td>B blocked</td>
</tr>
<tr>
<td>Load A</td>
<td>A &amp; B resident</td>
<td>A &amp; B running</td>
</tr>
<tr>
<td>B exits</td>
<td>A resident</td>
<td>A blocked, exited</td>
</tr>
<tr>
<td>Load D</td>
<td>D resident</td>
<td>D running</td>
</tr>
<tr>
<td>D IPCs C</td>
<td>D resident</td>
<td>D blocked</td>
</tr>
<tr>
<td>Load C</td>
<td>C &amp; D resident</td>
<td>C &amp; D running</td>
</tr>
</tbody>
</table>

Figure 5.3 Round robin scheduling activity on 2 hardware context MCA processor.

This round-robin scheduling of the four contexts requires five full scheduling actions (since only loading and swapping are considered scheduling actions, and the swap-load pairs on lines 4-5 and 7-8 are parts of a single scheduling cycle), and contexts A and C are each swapped and re-loaded. Fewer scheduling actions would be required if, in line four, context C were swapped rather than context A. However even fewer scheduling actions are required if scheduling actions are based on the IPC requirements of tasks. We call this IPC directed scheduling.

**IPC Directed Scheduling.** This form of scheduling is based on the handoff scheduling policy used in the Mach micro-kernel [27], but modified to use run-time IPC patterns gathered by the MCA hardware to assist the CGS task. This allows the CGS task to
schedule tasks efficiently without the need for explicit direction by a programmer.

In Mach, a thread can explicitly release the processor to another thread. This avoids the need to search ready queues, which in turn reduces the overall cost of scheduling in Mach. The technique is normally used to improve IPC performance by switching explicitly between communicating threads. The drawback to this approach is that it is software controlled. That is, it must be programmed explicitly when the program is written. The MCA processor is designed to gather IPC statistics, stored in the TAR register, during task execution. In addition, since IPC is performed in hardware, the TNR register of each resident task includes the task id of destination tasks during IPC operations. This information facilitates making scheduling decisions based on the run-time accumulation of IPC interaction among tasks. This is desirable in the MCA processor because tasks which communicate with each other can be kept resident. This allows IPC operations to execute immediately, without blocking resident tasks.

IPC directed scheduling involves two distinct steps. The first uses a high speed deterministic attempt to select a new task to make resident. This task, if it exists, is chosen to satisfy an immediate pending IPC request for a resident task. If this fails, a probabilistic mechanism is employed in an effort to find another task, or tasks, to load. In this case, the scheduler simply tries to select a task that is likely, based on a histogram of past IPC activity, to execute without blocking.
For deterministic scheduling, when a scheduling operation is needed, information stored in the TNR for each hardware slot can be used by the CGS task to schedule tasks without searching the active task list. For probabilistic scheduling, information in the TAR and process table (PCB), as the active task list is scanned, is used to schedule tasks.

IPC directed scheduling can be used to schedule the tasks of Figure 5.2 more efficiently than round robin scheduling can. In the example given above, if context B was favored over context C during the initial scheduling cycle, then the scheduling pattern of Figure 5.4 would result. In this example IPC directed scheduling requires just three scheduling actions, and no contexts must be swapped and re-loaded. Note, however, that this method is not necessarily fair, only potentially more efficient in a multiple context processor. However, we have found in our simulations that IPC directed scheduling outperforms round-robin scheduling and provides good response without starvation.

<table>
<thead>
<tr>
<th>Action</th>
<th>Status</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A IPCs B Load B</td>
<td>A resident</td>
<td>A blocked</td>
</tr>
<tr>
<td>B IPCs A Load C</td>
<td>A &amp; B resident</td>
<td>A blocked, B running</td>
</tr>
<tr>
<td>C IPCs D Load D</td>
<td>C resident</td>
<td>A &amp; B running, then exit</td>
</tr>
<tr>
<td>D IPCs C</td>
<td>C &amp; D resident</td>
<td>C running</td>
</tr>
</tbody>
</table>

*Figure 5.4* IPC driven scheduling activity on 2 hardware context MCA processor.
Response time and starvation avoidance can be ensured in IPC directed scheduling by also incorporating time quantums in the scheduling algorithm. In MC/OS, the IPC mechanism is controlled by also monitoring an instruction quantum for each context. Context's exceeding their instruction quantum are blocked by the real-time clock task. The SSR field St is set to five. Since the context is not blocked on an IPC operation, it will be swapped out ahead of any context blocked on IPC. The algorithm for selecting which context to swap out, and which to make resident, in IPC directed scheduling is as follows:

1. Choose an empty hardware slot, if it exists; else,
2. Update TAR Sact field with Sipc value for all slots. Then
3. Check last blocked hardware slot for IPC blockage, and select the destination task to make resident if possible; else, select the highest priority runnable task in active list. Next,
4. Choose a hardware slot with SSR field St equal to 1, if it exists; else,
5. Choose hardware slot with lowest Sact (lowest activity) field

The first step simply attempts to use free slots before swapping active slots to memory. If there is an available slot or no waiting tasks (e.g., nothing to load anyway) then no swapping is required. If there is an available slot and waiting contexts, then a call to pick Proc is made, a task is selected from the active list and made resident, and the processor is released after this is completed. If no slots are available and there are contexts on the active list, the remaining scheduling steps are taken.

The next step updates the Sact (see Figure 3.11) field in the TAR for each hardware context slot with all IPC activity since the last scheduling cycle. This action approximates least
recently used (LRU) information for each resident task. The update mechanism is controlled by the CGS task and is normally a weighted average of task activity since the last scheduling cycle. That is, the current context activity is added to the prior activity after first reducing the value of prior activity (e.g., divide prior activity by two, then add current). For example, if Sact is being used to gather IPC activity, then this operation will add the contents of the Sipc field to the (modified) Sact field in the TAR register of each hardware slot. Using the values shown in Figure 3.11, for instance, the final Sact value for slot 1 would be 237 (200+37/2), and for slot S-2 would be 200 (150 + 100/2). Thus, recently active contexts will have a high Sact activity value, while less active (with respect to IPC operations, not instruction execution) contexts will have a lower Sact value. Due to the weighting of prior activity, the affects of high activity will carry over, albeit at a diminishing rate, in subsequent scheduling cycles. This way, a context which is inactive temporarily, but otherwise active, will not be immediately swapped out in favor of a recently active, but historically less active context.

The third step allows the coarse scheduler to check for the last blocked task. If it has blocked on an IPC operation then all other slots must be blocked (IPC or otherwise); otherwise, the hardware scheduling mechanism would have selected another slot for execution. Using the most recently blocked slot, the scheduler can check in a single step if the destination task exists in the active list and is also blocked on a complementary IPC operation. This avoids the need to search the entire active list. If a matching context is found, it is loaded after one of the remaining hardware slots is swapped to memory (the
remaining steps in the algorithm).

If a matching context is not found, then the active list is examined and the highest priority task which is either ready (St1 field is 1), or blocked on an IPC operation that can be satisfied, is chosen. This process is essentially the same as round-robin scheduling. A task blocked on an IPC operation is a candidate if a currently resident context is blocked attempting IPC with that task (i.e., a resident task is blocked trying to send to a task on the active list which is attempting a receive with the resident task).

Once a task is selected for loading a hardware slot must be made available by swapping a resident context to memory. If a slot has exceeded its time (instruction) quantum, it will be swapped before any others. Otherwise, the least active slot, based in Sact values, is chosen.

Once a hardware slot is selected, its software state is saved and a call to sched is made by CGS. Upon return, the processor is released for hardware instruction scheduling by the SSU. The difference between the coarse scheduler and the scheduling activities of the nano-kernel at system startup is that the nano-kernel does not immediately release the processor as the coarse scheduler does. The nano-kernel, when finished with initialization, performs a conventional context switch to the CGS task which then releases the processor.
5.3.2 System Task

The system task, system, supports device independent system calls through message requests sent directly to the task. As with all other kernel level tasks, there are no other user level entry points into system. Interrupt service routines can, however, make calls to routines within system, bypassing the IPC mechanism for simplicity and speed.

The primary responsibility of the system task is for support of user level process creation and control. This includes thread management, fork, signal, and kill operations. From the user's perspective, these functions are identical to those found in other UNIX based systems. In MC/OS, the internal implementation is only a little different. The main difference is that system does not directly control the scheduling of newly created contexts. This is handled solely by the coarse grain scheduler (CGS) task. This is complicated by the fact that CGS does not accept any IPC messages. Hence, system can only add new contexts to the active list, but cannot cause a new context to execute immediately.

5.3.3 Clock Task

The last kernel task listed in Table IX is the real-time clock task. Clock is responsible for monitoring the allocation of processor time in cooperation with the IPC scheduling mechanisms described above. In addition the clock task supports such system calls as alarm, time, and various accounting calls. These are functionally equivalent to other UNIX-like systems.
The clock task itself consists of two parts, a user interface task and an interrupt routine that is called on each real-time clock interrupt. The user interface task accepts message requests for alarm signals and accounting information. Requests for accounting information (i.e., system resource usage) are processed immediately and a reply message is sent. Alarm requests, which allow a task to request an “interrupt” signal at some point in the future, are handled by adding the request to an alarm queue, along with a “request being serviced” reply. The alarm signal, when the alarm expires, is handled by sending a message to system which then processes the signal.

The interrupt routine is responsible for time quantum management for resident contexts, and checking for expired alarm signals. Tracking the exact time actually allotted to a given context in a multiple context processor is difficult to accomplish since the hardware can switch arbitrarily often between hardware slots. This could be accomplished with the addition of an instruction count register to record all instructions executed for each slot over a period. The clock interrupt routine could then check and clear these registers periodically. However, if the number of hardware contexts is large the complexity of the clock routine could become excessive for coarse scheduling IPC. Consequently, the MC/OS clock interrupt routine simply tracks which context was running each time an interrupt occurs. If it discovers the same context in consecutive interrupts, then that context might be compute bound, and action can be taken if it is a user task (system tasks are assumed safe and allowed to execute freely). This greatly simplifies the accounting work required of the clock task. As a result, the amount of time spent in the clock task is greatly reduced. This
Improves overall throughput without significantly affecting response time for individual programs.

5.4 Server Tasks

Server tasks run at a higher protection level, see Figure 5.1, than kernel tasks, and do not share any portion of the operating system address space. Functionally, servers are separate from the kernel and are treated as higher priority user programs. The three primary servers in MC/OS are the memory manager (mm), file system (fs), and IPC registry or post office (po). In the current version of MC/OS, mm and fs are unchanged from the original system used for development, Minix 1.7.2 [18], and are not discussed further. The IPC registry is responsible for managing the public and private mapping of processes to OwnId values to support the hardware based MCA IPC mechanism. The po has not been implemented in the simulation version of MC/OS because we were able to conduct experiments without using the registry to obtain public keys. This was accomplished by using fixed key values for each task.
This Chapter describes the simulation environment developed to test and compare the MCA architecture and MC/OS operating system with an unmodified multitasking operating system and architecture. The added registers and hardware instructions needed to implement the MCA architecture are general, and therefore can either be developed as a completely new architecture or added as an extension of an existing architecture. This provides a general purpose platform to develop new architectures and system software and be able to study the effect on general purpose commercial applications. For this reason, we chose to implement MCA as an extension to the 80x86 architecture. Choosing to extend an existing architecture also reduced development time, allowing us to develop the MC/OS operating system from the existing system, Minix 1.7.2. Further, performance analysis is enhanced because we can run nearly identical software on the original 80x86 and proposed MCA systems.

6.1. MCA Processor Simulation

For simulation experiments, the MCA architecture is implemented as an extension of the 80x86 architecture. The MCA organization simulated consists of a single processor (e.g., SSU) and single execution unit, with support for a variable number of hardware context
slots and register windows for IPC. This was accomplished by simulating an 80x86 processor in software, using available simulation software. In addition to simulating the processor, the software also simulates peripheral devices, allowing us to effectively develop a working system, albeit all simulated in software. All of the instructions and hardware registers discussed in the previous chapters were implemented in the simulation environment. This allowed us to study the effect of changing the number of hardware contexts, as well as other system parameters, on system performance.

We started the simulation platform using a shareware simulator called "bochs"[46] for the 80x86 family processors. Extensions for the MCA processor were added as additional modules to the original bochs simulator. The basic program simulates the 80x86 instruction set as well as memory, floppy, hard drive, and video peripheral devices. The relationship between the components of the simulation environment are shown in Figure 6.1. The third level in Figure 6.1 presents a virtual MCA processor, corresponding to Figure 3.2, to programs running in the first two levels. The bochs simulator runs in an X-window environment, using one X-window for the bochs simulator, and another for the PC video display. A program using the modified bochs simulator runs on a virtual processor. The virtual processor can be configured when started by bochs to be either a virtual 80x86, or a configuration of the MCA processor. This enables to simulation of an MCA processor with an arbitrary number of hardware contexts. The X-window interface makes it possible to monitor the simulation while interacting in the simulated environment.
<table>
<thead>
<tr>
<th>User Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Experiment Software)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Minix 1.7.2</td>
</tr>
<tr>
<td>MC/OS 1.0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>MCA Virtual Processor System</td>
</tr>
<tr>
<td>(80x86 simulation system modified to execute MCA instructions)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Bochs Simulation Software</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>X Window Driver</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Unix Operating System</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>80x86 Physical System Hardware</td>
</tr>
<tr>
<td>(100 MHz Pentium, 16 Mb Ram, 1Gb SCSI hard disk)</td>
</tr>
</tbody>
</table>

**Figure 6.1** MCA simulation platform.

MCA hardware instructions were implemented as software interrupts in the bochs simulator. This made it possible to add MCA functionality with few modifications to the original simulation package. Two distinct modules were added to bochs, as shown in Figure 6.2. The first module, soft_int.c, performs MCA instruction simulation. When a hardware context must be loaded or saved, a separate routine is called in mcp_load.c. The routines in mcp_load.c are Intel 80x86 dependent, and so are isolated from the hardware independent features of MCA instruction simulation.
MCA instructions are embedded within the Intel 80x86 software interrupt instruction. The Intel INT instruction allows a second byte operand field to identify the type of software interrupt. This is used to store the MCA opcode. This allows MCA instructions to be simulated easily with minimal modifications to the bochs software. This process is illustrated in Figure 6.3.
Additional arguments are needed for some MCA instructions, described in Chapter 4. These arguments were loaded into general purpose registers before issuing the INT instruction so that they would be available as the INT is processed. This was accomplished by adding a special library routine, sendmcp, for issuing all the software interrupt calls. The syntax for the sendmcp call is shown in Figure 6.4.
sendmcp( mnemonic, arg1, arg2);

- mnemonic - MCA opcode
- arg1 - First instruction argument
- arg2 - Second instruction argument

Figure 6.4 Syntax of MCA instruction simulation call.

The purpose of the sendmcp function is to place arguments in the registers of the virtual processor and issue the 80x86 software interrupt instruction. Although this could be done in assembly language, it is simpler to incorporate this into the high level language. This made it possible to add instructions without affecting the basic function of the bochs simulator or compiler support within MC/OS and Minix. In addition, by using software interrupts to simulate new instructions, instructions can be added and modified without affecting the compiler and language (C) the MC/OS operating system is written in. One advantage of this approach is that it speeds development. A significant drawback is that all new instructions are really executed as system calls. This increases the simulated instruction count over what would be required in a hardware implementation of the processor. This has been accounted for in the simulation results presented in the next Chapter.

6.2. MC/OS Simulation

The simulation version of the MC/OS operating system is nearly identical to the system described in Chapter 5. The test system was kept as close to original Minix 1.7.2 as possible so that comparisons with the unmodified system would highlight the architectural
improvements of the MCA processor and scheduling mechanisms rather than differences between the two operating systems. As a result, some parts of the MC/OS were not needed, such as the IPC registry task, and they were not implemented in simulation. Since this is not an original part of Minix, it was left out. IPC in the unmodified Minix is handled by using fixed identification values for all tasks. Hence, we used the same technique in the experimental version of MS/OS.
In this chapter we outline the experiments we have used to evaluate the performance of the architectural features proposed and described in the previous Chapters. A number of experiments were conducted to determine how well the MCA architecture, combined with the MC/OS operating system, performs relative to a conventional multitasking operating system and hardware. As described in the previous Chapter, the experiments were designed to limit the differences between MCA and MC/OS and our control system, a UNIX-like operating system running on single context processor. This makes it possible to accurately compare the architectural enhancements proposed for MCA with existing systems, running typical program workloads.

The experiments described in this section were designed to compare the performance of MCA and MC/OS with a typical multitasking system under a variety of workloads. The workloads were chosen to be similar to those which might be encountered under normal use. These include computation intensive tasks, I/O intensive tasks, distributed tasks, and a combination of each of these run concurrently. Each of these is described and execution results for the control system and the MCA processor are compared. Results are compared for a number of configurations of the MCA architecture.
Experimental data was gathered by accumulating instruction counts during the course of the experiment. The hardware simulator is capable of running both the control system as well as arbitrary configurations of the MCA architecture. As mentioned in Chapter 6, simulation is performed at the instruction level. Hence, it was possible to modify to the simulator to also count the frequency of each instruction simulated during a test run. Modifications were made to accumulate instruction counts for the following categories of instruction: total instructions executed, MCA instructions simulated, instructions by software context, instructions by MCA instruction opcode, and by 80x86 instruction opcode.

Reducing kernel overhead is an important motivating factor for this research. Hence, it is important to be able to accurately determine the ratio of kernel time versus the time spent in all other tasks during an experiment. Tracking instructions executed by software context makes it possible to determine the overall contribution of each task for a particular experiment. This was used to track the total kernel (scheduling) overhead.

The original Minix system, since it consists of a single context which does not utilize any MCA instructions, does not allow direct measurement of the number of kernel instruction executed. For these values, we estimate instruction counts by hand calculation of instructions for scheduling, IPC, and context switching, and observed switches based on 80x86 opcode analysis, also performed in [44].
An additional MCA instruction, dump, was added to support the gathering of test data during experiments. This instruction simply causes the hardware simulator to dump all instruction count fields to standard output. Each time these values are displayed the counts are reset to zero. Thus, for any given experiment this instruction was issued just before and immediately after the test run. The first set of values was thrown out, with the second set being the results for that experiment.

In addition to evaluation of the MCA architecture, the two long term scheduling mechanisms described in Chapter 5 were also tested. These are conventional priority based round robin and the new approach proposed in this research, IPC driven scheduling.

Experimental results are presented and evaluated for various configurations of the MCA processor and scheduling algorithms. For each experimental workload, the MCA processor configuration was modified for a range of hardware context slots to determine the effect the number of resident tasks has on overall performance and kernel overhead. A configuration using a small number of hardware slots corresponds to a more heavily loaded system. That is, the ratio of active tasks to available slots is high. Thus, more scheduling activity is necessary. A small ratio (e.g., many available slots and few tasks) corresponds to a lightly loaded system, and does not require frequent scheduling activity. The MCA processor was configured to use between 4 and 24 hardware slots. Fewer than 4 slots result in run-time failures of the MC/OS, and experiments never used more than 24 active contexts so that increasing the number of contexts beyond 24 did not provide
additional information.

In the following sections we present and interpret the results of each test performed on the proposed system.

7.1. Distributed Task Performance

We examined the performance of the MCA processor using a large distributed task, which is typical of many distributed applications. In our study, the distributed task is the MC/OS operating system itself. As discussed previously, MC/OS consists of four independent (e.g., distributed) programs: the kernel, memory manager, file system, and process/user manager (e.g., init in UNIX). In addition, the kernel is partitioned into a variable number of software threads, based on a few core task threads, and with a separate thread for each physical device driver. Each thread shares a common instruction and data address space but separate stack space. The number of active software contexts in the MC/OS system depends on the number of peripheral devices supported by the system. The MC/OS system (see MC/OS structure, Figure 5.1) contains at least 10 software contexts: CGS, system, clock, disk (hard disk driver), floppy (floppy disk driver), memory (kernel memory map driver), tty (keyboard and monitor driver), mm (user memory server), fs (file system server), and init (user task server). The MC/OS operating system is therefore a good example of a commercial distributed and/or multithreaded system.

The MC/OS operating system was evaluated for the range of hardware context
configurations described in the previous section, from 1 to 24, to measure the performance of a distributed system on the MCA processor. The numerical results that reflect the performance characteristics of the MCA processor are presented in Table X. These figures represent counts for an entire experimental run, not values per context. Individual context counts are included in Appendix C. The values presented here are for IPC driven long term scheduling. Round robin scheduling is discussed later.

Table X. Distributed Application Performance Results

<table>
<thead>
<tr>
<th>Hardware Contexts</th>
<th>Total Ins. (M)</th>
<th>MCA Ins. (k)</th>
<th>Kernel Ins. (M)</th>
<th>Context loads</th>
<th>Message Exchanges</th>
<th>Processor Releases</th>
<th>MCA queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10.13</td>
<td></td>
<td>3.71</td>
<td>8,990</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8.51</td>
<td>44.30</td>
<td>1.399</td>
<td>1,204</td>
<td>4,600</td>
<td>3,424</td>
<td>8,528</td>
</tr>
<tr>
<td>5</td>
<td>7.39</td>
<td>32.50</td>
<td>0.573</td>
<td>462</td>
<td>4,578</td>
<td>2,667</td>
<td>2,296</td>
</tr>
<tr>
<td>6</td>
<td>7.26</td>
<td>30.20</td>
<td>0.507</td>
<td>315</td>
<td>4,575</td>
<td>2,562</td>
<td>1,535</td>
</tr>
<tr>
<td>8</td>
<td>7.13</td>
<td>27.32</td>
<td>0.451</td>
<td>141</td>
<td>4,575</td>
<td>2,223</td>
<td>1,335</td>
</tr>
<tr>
<td>10</td>
<td>7.27</td>
<td>26.80</td>
<td>0.600</td>
<td>67</td>
<td>4,571</td>
<td>2,096</td>
<td>6,446</td>
</tr>
<tr>
<td>16</td>
<td>6.94</td>
<td>23.60</td>
<td>0.287</td>
<td>50</td>
<td>4,576</td>
<td>7,547</td>
<td>120</td>
</tr>
<tr>
<td>24</td>
<td>6.93</td>
<td>22.35</td>
<td>0.275</td>
<td>50</td>
<td>4,576</td>
<td>6,286</td>
<td>1,113</td>
</tr>
</tbody>
</table>

The data shown in each column is as follows:

**Hardware Contexts.** This column shows the number of MCA hardware slots simulated in the experiment, including slot zero which is reserved for the kernel/CGS task. This was accomplished by providing a value to the MC/OS nano-kernel during initialization specifying the number of available hardware contexts. The row containing a single slot
represents the original system without multiple contexts or register windows used for IPC.
This is the control system in each experiment.

**Total Instructions.** This represents the number of instructions, in millions \((10^6)\), counted by the simulator during an experiment. This includes 80x86 and the extended MCA instructions described in Chapter 4. This total has been adjusted for each MCA instruction executed to reflect the way in which MCA instructions are simulated by the modified bochs simulator. This is because these instructions are simulated using a library function consisting of a number of 80x86 instructions. Hence the simulation of each MCA instruction increments the total instruction count by more than one. This is discussed in Appendix B where the simulation source code is presented. The total is also adjusted for each full context switch, because of the method of simulating hardware context save and restore operations. In simulation, only a few instructions result in a full context save/restore for contexts to hardware slots other than slot zero. In practice, the cost would be the same for all slots. Actual instruction counts and calculations used to derive the values in Table X are given in Appendix C. In general, the difference between adjusted and unadjusted values is small because the number of MCA instructions simulated during an experiment is a small fraction, less than 5 percent, of the total instructions simulated.

**MCA Instructions.** This column includes the total number of MCA instructions simulated, measured in thousands \((10^3)\). Note that the vast majority of instructions are 80x86 not MCA instructions. reduction of overall system overhead in this study was
achieved by reducing the total number of instructions executed during an experiment. This suggests that a few MCA instructions replaced a large number of 80x86 instructions to accomplish the same task.

**Kernel Instructions.** The number of kernel/CGS instructions, in millions, during an experiment is counted to determine the scheduling overhead.

**Context Loads.** This is the number of full context loads during an execution run. In the unmodified system, shown in row 1 in Table X, a context load involves a context save and restore. In the MCA architecture, because there are multiple slots, a load may or may not involve a prior context save. The number of context saves was tracked as well, but not displayed here (see Appendix C). The number of loads, however, is a direct indication of the number of times the system switches between threads during an experiment.

**Message Exchanges.** This indicates the number of IPC operation send/receive pairs. The number of IPC operations is a good indication of the level of interaction among threads during execution. Observe that we count 1 IPC operation for each send/receive pair.

**Processor Releases.** This gives the number of times during a test that the CGS scheduling task releases the processor to the hardware scheduler (SSU). The CGS task releases the processor whenever possible, and is only invoked following interrupts or if no other slot is ready. Following interrupts the CGS normally releases the processor immediately. The
number of releases is an indicator for the number of times the CGS task executes during execution.

**MCA Queries.** This includes a group of commonly used MCA instructions used to read the contents of various MCA register values. These instructions are used primarily by the CGS task and certain system calls (e.g., fork, exec, etc.). This figure is used to estimate how much work the CGS task is performing. A large number of queries relative to the number of context loads implies the CGS task is spending a lot of time looking for a suitable task to load/swap.

The combined figures of the last four columns give a more reliable measure of processor activity. For example, if the number of context loads is small but the number queries is high, the processor can either be deadlocked and spending a great deal of time finding a task to load, or it could simply be idle. The number of releases indicates which of the two. If the system is idle, the number of releases is high as well. To illustrate this, observe the rows in Table X above for 4 and 24 contexts. Both have relatively high query values, but with 24 contexts, the number of releases is extremely high. It is easy to see that with that many hardware contexts there are not likely to be many non-resident software contexts. Hence, there is nothing for the CGS task to do if all resident tasks are blocked. In this case, each time the CGS task is executed, it releases the processor immediately. This is the equivalent of an idle task in other operating systems.
The configuration with 1 context represents the control, or original, system. It consists of the unmodified Minix operating system running on a single hardware context 80x86 processor. For this system, only the total instruction and context load counts for the experiment are recorded. The number of kernel instructions are estimated as described in the previous section. Very little additional information was available without further modifying the original system to use MCA instructions. Multiple context values ranging from 4 to 24 correspond to various MCA processor configurations. For configurations with fewer than four hardware contexts, performance was extremely poor (worse, in fact, than the original system). This suggests that multiple context management adds additional overhead. This overhead must be offset by reducing overhead in other areas, namely by reducing the number of full context switches and reducing IPC costs. As illustrated by the four context configuration, however, just a few hardware contexts can reduce system overhead substantially. Operating system overhead rates are shown in Figure 7.1. For each experiment the same application was executed and statistics gathered. The variable in each experiment was the number of hardware contexts, all other processor characteristics (e.g., memory, etc.) were kept constant.
In these experiments we considered all kernel instructions as overhead. Hence, overhead is given by

\[ \text{Overhead} = 100 \times \left( \frac{\text{kernel instructions}}{\text{total instructions} - \text{kernel instructions}} \right) \% \tag{7.1} \]

This is equivalent to the utilization calculations developed in Chapter 2. In Figure 7.1 we show overhead rather than utilization. Figure 7.2 shows processor utilization.
Distributed Application - Processor Utilization for IPC Driven Scheduling

![Bar Chart](image)

**Figure 7.2** Processor utilization for multiple contexts running distributed application.

The relationship between utilization and overhead is given by

\[
\text{Utilization} = 100 - \text{Overhead} \quad (7.2)
\]

Hence, for the values shown in Table X for the control system (1 hardware context), the overhead is found to be approximately 58%. Thus, utilization is only 42%. Utilization is discussed in more detail in section 7.5.

In the overhead calculation of equation 7.1 we see that the overhead of the original (single context) system is substantially higher than a system with even a few hardware contexts. Hence, a four context MCA processor reduces overhead by more than 60 percent (e.g., from...
58% to 20%). Additional contexts reduce this further, until there are more hardware contexts than threads in the application. This occurs between 10 and 24 contexts for this experiment. Thus, the reduction in overhead levels off, and can even begin to increase, as the number of contexts is increased beyond the number of active contexts. One reason observed for the increase is that the long term scheduler can spend "excessive" time looking for contexts to load. This is observed by the number of processor release instructions issued. Note that the number of processor release instructions (column 7 in Table X) declines steadily until the number of hardware contexts reaches ten. This is the number of fairly active threads in the experimental version of the MC/OS operating system. After ten, the number of releases increases sharply.

Processor releases are performed only by the long term scheduler task within the kernel. These occur whenever no other hardware context is ready and control is passed to the scheduler. Upon completion, the processor is released. Thus, if the processor is lightly loaded, the scheduler task is selected frequently, and when no ready contexts are found, the processor is released again. This happens often when an interactive program is the only user task executing. Human responses to program prompts are very slow relative to the speed of the CPU, so for extended periods all tasks are blocked awaiting the user's response.

Eliminating system overhead results in a reduction in the total number of instructions necessary to perform a given task. This also reduces memory traffic. By executing fewer
instructions and reducing memory traffic, we can expect applications to execute more rapidly. The speedup resulting from this reduced instruction count is shown in Figure 7.3. Speedup is given by

\[
\text{Speedup} = \frac{N_{\text{original}}}{N_{\text{MCA}}}
\]  

(7.3)

In equation 7.3 \(N_{\text{original}}\) is the total number of instructions simulated for the unmodified system (row 1, Table X), and \(N_{\text{MCA}}\) is the number of instructions for a particular MCA configuration. For an MCA using 4 hardware contexts, the speedup value is 1.19. Hence, even with a minimal number of contexts (4), a speedup of almost 20% can be achieved. With additional contexts, a speedup of more than 45% was observed. As the number of contexts increases further, the apparent overhead can begin to increase, as noted above. If idle scheduling time is excluded, speedup appears to asymptotically approach fifty percent. This makes sense since the estimated overhead of the unmodified kernel is a little more than 55%. Even with unlimited contexts some kernel overhead is needed to initially load tasks into hardware slots when created, and clear slots as tasks terminate.
7.2 Computation Intensive Application Performance

The MCA processor was designed to perform well in a distributed computing environment. It is also important that performance not degrade when executing compute bound tasks. Therefore, a number of computationally intensive applications were tested to measure the performance compared with that of the unmodified system. Three typical programs were included: Roff, sort, and compress. Roff is a document formatting filter (similar to nroff and troff) [47]. It reads text from a file containing embedded formatting commands, formats the text, and writes output to another file. Sort reads input from a file, sorts the
entries, and writes the sorted result to another file. Compress [47] uses a Lempel-Ziv compression algorithm to compress an input file. Of the three, roff is the most computationally intensive. These applications were run separately and concurrently. Results for separate execution are shown in Table XI. Concurrent results are given in Table XII.

Table XI. Computation Intensive Performance Results

(Separate Execution)

**Application: roff**

<table>
<thead>
<tr>
<th>Hardware Contexts</th>
<th>Total Ins. (M)</th>
<th>MCA Ins. (k)</th>
<th>Kernel Context Ins. (M)</th>
<th>Context Loads</th>
<th>Message Exchanges</th>
<th>Processor Releases</th>
<th>Processor Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20.57</td>
<td></td>
<td>0.730</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>19.89</td>
<td>9.11</td>
<td>0.327</td>
<td>21</td>
<td>534</td>
<td>2,381</td>
<td>101</td>
</tr>
<tr>
<td>16</td>
<td>19.89</td>
<td>9.14</td>
<td>0.314</td>
<td>5</td>
<td>562</td>
<td>2,468</td>
<td>21</td>
</tr>
<tr>
<td>24</td>
<td>19.85</td>
<td>6.75</td>
<td>0.264</td>
<td>5</td>
<td>558</td>
<td>2,481</td>
<td>65</td>
</tr>
</tbody>
</table>

**Application: sort**

<table>
<thead>
<tr>
<th>Hardware Contexts</th>
<th>Total Ins. (M)</th>
<th>MCA Ins. (k)</th>
<th>Kernel Context Ins. (M)</th>
<th>Context Loads</th>
<th>Message Exchanges</th>
<th>Processor Releases</th>
<th>Processor Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10.02</td>
<td></td>
<td>0.580</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9.23</td>
<td>7.31</td>
<td>0.183</td>
<td>60</td>
<td>818</td>
<td>1,286</td>
<td>292</td>
</tr>
<tr>
<td>16</td>
<td>9.18</td>
<td>6.25</td>
<td>0.159</td>
<td>5</td>
<td>816</td>
<td>1,250</td>
<td>25</td>
</tr>
<tr>
<td>24</td>
<td>9.14</td>
<td>4.91</td>
<td>0.133</td>
<td>4</td>
<td>789</td>
<td>1,255</td>
<td>8</td>
</tr>
</tbody>
</table>

**Application: compress**

<table>
<thead>
<tr>
<th>Hardware Contexts</th>
<th>Total Ins. (M)</th>
<th>MCA Ins. (k)</th>
<th>Kernel Context Ins. (M)</th>
<th>Context Loads</th>
<th>Message Exchanges</th>
<th>Processor Releases</th>
<th>Processor Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12.50</td>
<td></td>
<td>0.530</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>11.96</td>
<td>6.80</td>
<td>0.218</td>
<td>38</td>
<td>536</td>
<td>1,515</td>
<td>184</td>
</tr>
<tr>
<td>16</td>
<td>11.90</td>
<td>6.11</td>
<td>0.193</td>
<td>4</td>
<td>509</td>
<td>1,518</td>
<td>8</td>
</tr>
<tr>
<td>24</td>
<td>11.89</td>
<td>4.73</td>
<td>0.161</td>
<td>5</td>
<td>532</td>
<td>1,528</td>
<td>49</td>
</tr>
</tbody>
</table>
Table XII. Computation Intensive Performance Results

(Concurrent Execution)

<table>
<thead>
<tr>
<th>Hardware Contexts</th>
<th>Total Ins. (M)</th>
<th>MCA Ins. (K)</th>
<th>Kernel Ins. (M)</th>
<th>Context Loads</th>
<th>Message Exchanges</th>
<th>Processor Releases</th>
<th>Processor Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>44.45</td>
<td>2.760</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>40.65</td>
<td>26.76</td>
<td>0.781</td>
<td>167</td>
<td>2,571</td>
<td>5,423</td>
<td>799</td>
</tr>
<tr>
<td>16</td>
<td>40.67</td>
<td>16.43</td>
<td>0.700</td>
<td>26</td>
<td>2,588</td>
<td>6,118</td>
<td>116</td>
</tr>
<tr>
<td>24</td>
<td>40.88</td>
<td>16.13</td>
<td>0.564</td>
<td>14</td>
<td>1,880</td>
<td>5,157</td>
<td>86</td>
</tr>
</tbody>
</table>

As expected all processor configurations perform fairly well on compute bound applications. This includes the unmodified system. Context swaps (loads in the table above) and system calls are minimal. Thus system overhead is extremely small, less than 6% for the unmodified system. The MCA processor, however, still outperforms the original system, with overhead of approximately 2% for a configuration of 6 hardware contexts. Using equation 7.3 we calculate a speedup of approximately 9%. This suggests that the additional overhead required to manage additional hardware contexts is small and is more than offset by the savings in context switch and IPC realized by the architecture.

Of even greater interest is the scalability of the system [3]. That is, as concurrence is increased, how does system performance change? If a system scales poorly, overhead will increase more rapidly than the increase in concurrency; if it scales well, overhead should increase at the same rate as concurrency, or even more slowly. In this case we examined how the multiple context processor performs executing several independent applications, comparing this with the original architecture. These results are summarized in Table XII, above. This test was conducted by executing the same roff, sort, and compress
experiments, but concurrently rather than separately. This made it possible to compare concurrent results, running the same MCA configurations as in individual tests, and against the sum of the three individual results (i.e., with less concurrency). Thus, we were able to evaluate the scalability of the MCA processor as additional concurrence is introduced.

It can be concluded from the results of the concurrent tests that the multiple context processor outperforms the original single context processor. Moreover, a single context system is not particularly scalable. The concurrent execution on the original system required an additional 1.36 million instructions, almost all of which is within the scheduling and IPC service routines of the kernel. The remainder is in task routines called by the kernel. Thus, the entire increase in instruction count can be attributed to added overhead. Further, the MCA architecture scales extremely well, particularly with few hardware slots. For instance, the combined total and kernel instruction counts for a 6 slot configuration, running each application separately, were 41.08 and 0.728 million, respectively.

Comparing with the concurrent results for 6 slots, 40.65 and 0.781 million instructions, we can conclude that the total instruction count dropped while kernel overhead has risen slightly. The reason for the different directions (increase in kernel instructions, yet a decrease in total instructions) is found in what functions the kernel is performing in each case. In the concurrent run, more context switches were necessary, requiring kernel scheduling overhead. In the individual runs, fewer full context switches were needed, but
there is a slightly higher rate of process releases. That is, the scheduler is simply releasing the processor to the hardware scheduler (i.e., attempting to run resident tasks). Thus, during individual runs there were no ready non-resident tasks, but the list of active contexts was searched regardless, requiring more instructions. This suggests that some changes could be made in the scheduling algorithm to improve performance further. This is being explored in continued research.

As the number of hardware contexts is increased, fewer initial context switches are required, so kernel overhead is reduced. However, when there are more hardware contexts than tasks, an even greater percentage of processor releases are issued, leading to a total instruction count that almost exactly matches that of separate execution. This is illustrated by the 24 context data results. For separate execution, the total number of instructions is 40.88 million, with 0.558 million kernel instructions. For concurrent execution, the total number of instructions is also 40.88 million, with slightly more kernel instructions, 0.564 million.

7.3 I/O Intensive Application Performance

In addition to distributed and compute bound applications, certain programs also exhibit high levels of input and output operations. Performance of the MCA processor was measured for this class of application as well. Testing was limited to non-interactive I/O intensive applications simply because it is easier to control uniformity of results.
Table X shows experimental results for the UNIX file system check utility, fsck [47]. This program examines and repairs the hard disk file system structure. This involves reading and checking the system super block and all I-node (UNIX file directory) entries for inconsistencies. This involves a large number of disk I/O system calls, with some computation after each call.

Table XIII. I/O Intensive Performance Results

<table>
<thead>
<tr>
<th>Hardware Contexts</th>
<th>Total Ins. (M)</th>
<th>MCA Ins. (K)</th>
<th>Kernel Context Ins. (M)</th>
<th>Context Loads</th>
<th>Message Exchanges</th>
<th>Processor Releases</th>
<th>MCA Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26.77</td>
<td>6.549</td>
<td>19,950</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>21.71</td>
<td>96.02</td>
<td>1.467</td>
<td>2,325</td>
<td>10,050</td>
<td>7,603</td>
<td>8,042</td>
</tr>
<tr>
<td>5</td>
<td>21.62</td>
<td>72.97</td>
<td>1.542</td>
<td>581</td>
<td>10,112</td>
<td>5,723</td>
<td>26,210</td>
</tr>
<tr>
<td>6</td>
<td>20.55</td>
<td>48.50</td>
<td>0.814</td>
<td>66</td>
<td>10,070</td>
<td>5,205</td>
<td>273</td>
</tr>
<tr>
<td>8</td>
<td>20.61</td>
<td>49.98</td>
<td>0.821</td>
<td>43</td>
<td>10,066</td>
<td>5,434</td>
<td>442</td>
</tr>
<tr>
<td>10</td>
<td>20.42</td>
<td>42.70</td>
<td>0.597</td>
<td>13</td>
<td>10,069</td>
<td>4,475</td>
<td>334</td>
</tr>
<tr>
<td>16</td>
<td>20.41</td>
<td>39.70</td>
<td>0.586</td>
<td>12</td>
<td>10,067</td>
<td>4,633</td>
<td>24</td>
</tr>
<tr>
<td>24</td>
<td>20.46</td>
<td>39.95</td>
<td>0.646</td>
<td>12</td>
<td>10,068</td>
<td>4,100</td>
<td>28</td>
</tr>
</tbody>
</table>

It can be observed from Table XIII that the characteristics of an I/O bound application are between the two extreme cases, distributed and compute bound. Since fsck interacts extensively with MC/OS, this test yields results that are similar in some respects to those of the distributed experiment. However, there are differences. There are few context switches in the I/O task, thus a few threads are responsible for performing most of the work. However, the rate of message passing is about the same for distributed and I/O tasks. Hence, the few tasks are communicating often. These factors indicate that performance can be improved with multiple hardware contexts, and that this performance gain can be
realized with just a few hardware slots. Overhead and speedup over a range of values is shown in Figures 7.4 through 7.6. Calculations for overhead and speedup are the same as given previously in equations 7.1 through 7.3, respectively.

Figure 7.4 Kernel overhead for I/O bound application with IPC driven scheduling.
Figure 7.5 Processor utilization for multiple contexts running I/O bound application.

Figure 7.6 Speedup of I/O bound application for IPC driven scheduling.
7.4 Comparison of Scheduling Algorithms

The most common method of long-term scheduling of application programs in multitasking operating systems involves some form of round robin scheduling [17-18]. By using multiple queues, a round robin algorithm can provide adequate priority, response time, and fairness with little overhead. As noted in Chapter 5, however, this is more difficult in the MCA architecture since IPC is not controlled by the kernel. As a result, a blocked task can become ready without the kernel’s interaction or knowledge. For this reason, separate ready and blocked queues of tasks are not possible to maintain without excessive overhead. Thus, the task of finding the next ready task to load during a scheduling cycle is more involved, and the simplicity of round robin scheduling is compromised, resulting in higher kernel scheduling overhead.

IPC driven scheduling was designed to keep kernel overhead to a minimum as explained in Section 5.4. We compared round robin and IPC driven scheduling algorithms over the complete set of applications discussed above. A comparison for two configurations of the MCA processor are given in Table XIV. Figures for round robin scheduling are labeled RR, while IPC driven scheduling results are labeled IPC.
Table XIV. Comparison of Round Robin and IPC Driven Scheduling Algorithms

<table>
<thead>
<tr>
<th>Method Program</th>
<th>Hardware</th>
<th>Total Ins. (M)</th>
<th>MCA Ins. (K)</th>
<th>Kernel Ins.(M)</th>
<th>Context Loads</th>
<th>Message Exchanges</th>
<th>Processor Releases</th>
<th>MCA Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Contexts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>roff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPC</td>
<td>6</td>
<td>19.89</td>
<td>9.11</td>
<td>0.327</td>
<td>21</td>
<td>534</td>
<td>2,381</td>
<td>101</td>
</tr>
<tr>
<td>RR</td>
<td>6</td>
<td>20.16</td>
<td>18.70</td>
<td>0.553</td>
<td>22</td>
<td>532</td>
<td>2,787</td>
<td>11,163</td>
</tr>
<tr>
<td>IPC</td>
<td>24</td>
<td>19.85</td>
<td>6.75</td>
<td>0.264</td>
<td>5</td>
<td>558</td>
<td>2,481</td>
<td>65</td>
</tr>
<tr>
<td>RR</td>
<td>24</td>
<td>19.87</td>
<td>7.30</td>
<td>0.281</td>
<td>5</td>
<td>564</td>
<td>2,764</td>
<td>47</td>
</tr>
<tr>
<td>sort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPC</td>
<td>6</td>
<td>9.23</td>
<td>7.31</td>
<td>0.183</td>
<td>60</td>
<td>818</td>
<td>1,286</td>
<td>292</td>
</tr>
<tr>
<td>RR</td>
<td>6</td>
<td>9.34</td>
<td>12.65</td>
<td>0.292</td>
<td>43</td>
<td>791</td>
<td>1,470</td>
<td>6,708</td>
</tr>
<tr>
<td>IPC</td>
<td>24</td>
<td>9.14</td>
<td>4.91</td>
<td>0.133</td>
<td>4</td>
<td>789</td>
<td>1,255</td>
<td>8</td>
</tr>
<tr>
<td>RR</td>
<td>24</td>
<td>9.16</td>
<td>5.25</td>
<td>0.138</td>
<td>5</td>
<td>814</td>
<td>1,366</td>
<td>47</td>
</tr>
<tr>
<td>compress</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPC</td>
<td>6</td>
<td>11.96</td>
<td>6.80</td>
<td>0.218</td>
<td>38</td>
<td>536</td>
<td>1,515</td>
<td>184</td>
</tr>
<tr>
<td>RR</td>
<td>6</td>
<td>12.03</td>
<td>14.01</td>
<td>0.348</td>
<td>26</td>
<td>509</td>
<td>1,769</td>
<td>8,547</td>
</tr>
<tr>
<td>IPC</td>
<td>24</td>
<td>11.89</td>
<td>4.73</td>
<td>0.161</td>
<td>5</td>
<td>532</td>
<td>1,528</td>
<td>49</td>
</tr>
<tr>
<td>RR</td>
<td>24</td>
<td>11.90</td>
<td>5.14</td>
<td>0.174</td>
<td>5</td>
<td>536</td>
<td>1,726</td>
<td>47</td>
</tr>
<tr>
<td>concurrent</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPC</td>
<td>6</td>
<td>40.65</td>
<td>26.76</td>
<td>0.781</td>
<td>167</td>
<td>2,571</td>
<td>5,423</td>
<td>799</td>
</tr>
<tr>
<td>RR</td>
<td>6</td>
<td>42.30</td>
<td>44.13</td>
<td>1.182</td>
<td>162</td>
<td>2,347</td>
<td>5,376</td>
<td>23,833</td>
</tr>
<tr>
<td>IPC</td>
<td>24</td>
<td>40.88</td>
<td>16.13</td>
<td>0.564</td>
<td>14</td>
<td>1,880</td>
<td>5,157</td>
<td>86</td>
</tr>
<tr>
<td>RR</td>
<td>24</td>
<td>41.37</td>
<td>20.05</td>
<td>0.537</td>
<td>24</td>
<td>2,450</td>
<td>7,666</td>
<td>252</td>
</tr>
<tr>
<td>distributed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPC</td>
<td>6</td>
<td>7.26</td>
<td>30.20</td>
<td>0.507</td>
<td>315</td>
<td>4,575</td>
<td>2,562</td>
<td>1,535</td>
</tr>
<tr>
<td>RR</td>
<td>6</td>
<td>7.53</td>
<td>34.20</td>
<td>0.945</td>
<td>331</td>
<td>4,571</td>
<td>2,341</td>
<td>8,970</td>
</tr>
<tr>
<td>IPC</td>
<td>24</td>
<td>6.93</td>
<td>22.35</td>
<td>0.275</td>
<td>50</td>
<td>4,576</td>
<td>6,286</td>
<td>1,113</td>
</tr>
<tr>
<td>RR</td>
<td>24</td>
<td>6.95</td>
<td>24.75</td>
<td>0.321</td>
<td>62</td>
<td>4,573</td>
<td>7,944</td>
<td>498</td>
</tr>
<tr>
<td>I/O bound</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPC</td>
<td>6</td>
<td>20.55</td>
<td>48.50</td>
<td>0.814</td>
<td>66</td>
<td>10,070</td>
<td>5,205</td>
<td>273</td>
</tr>
<tr>
<td>RR</td>
<td>6</td>
<td>21.46</td>
<td>64.70</td>
<td>1.476</td>
<td>28</td>
<td>10,115</td>
<td>6,677</td>
<td>19,117</td>
</tr>
<tr>
<td>IPC</td>
<td>24</td>
<td>20.46</td>
<td>39.95</td>
<td>0.646</td>
<td>12</td>
<td>10,068</td>
<td>4,100</td>
<td>28</td>
</tr>
<tr>
<td>RR</td>
<td>24</td>
<td>20.48</td>
<td>39.65</td>
<td>0.537</td>
<td>12</td>
<td>10,116</td>
<td>4,119</td>
<td>994</td>
</tr>
</tbody>
</table>
The kernel overhead observed in these experiments is computed and shown in Figures 7.7 and 7.8.

Figure 7.7 Kernel overhead for 6 hardware context configuration.
Scheduling Overhead for IPC Directed Scheduling vs Round Robin

![Bar chart showing overhead for different applications](chart.png)

Figure 7.8 Kernel Overhead for 24 hardware context configuration.

In all tests and for all configurations, IPC driven scheduling outperforms round robin unless there are more hardware slots available than ready tasks. In this case, the IPC scheduling algorithm spends more time looking for a task to load. The difference is more pronounced when the number of active tasks is high relative to the number of hardware slots. IPC scheduling results in fewer context switches (involving save and restore in this case), and lower kernel overhead. IPC scheduling works more efficiently than round robin because when a deadlock occurs (i.e., all hardware slots are blocked), the CGS task checks the last running task first to see if its block can be removed. This often avoids the need to search the list of non-resident tasks, saving significant time in the CGS.
The only disadvantage of the IPC scheduling mechanism is in ensuring fairness among tasks. IPC scheduling attempts to execute the same set of resident tasks for as long as possible, only invoking a full scheduling cycle when an immediate check fails. However, the CGS task still gives precedence to privileged system tasks when choosing which task(s) to make resident. The result was that fairness (and starvation) was not observed to be a problem in the tests we ran. Unfortunately, the simulation environment (mainly the slow speed of instruction simulation) restricted the complexity of our experiments. In the future we hope to enhance the capabilities of the simulation platform in order to evaluate fairness in IPC directed scheduling.

7.5 Summary

The results from the previous sections indicate that a multiple context processor with support for IPC can substantially reduce program execution overhead. In Chapter 2 we developed several equations to describe the contribution of three forms of overhead, computed as a cost per instruction, to the total execution time of a task in a multitasking system. These include context scheduling and switch cost, IPC cost, and number of cycles per instruction.

Through experimentation we have determined the overhead contribution of two of these sources of overhead, IPC and scheduling, under various workloads. The average number of cycles to execute an instruction was not evaluated in this research. This was not considered for two reasons. First, the hardware simulation platform does not support such
fine-grain simulation. Simulation is performed at the instruction level only. Second, since
the MCA processor is implemented as an extension of an existing architecture, the rate of
instruction execution is assumed to be the same for both the original system (e.g., 80x86)
and the MCA processor. This assumption makes it possible to eliminate the CPI term from
equations 2.6 a-c. This leaves the two measured sources of overhead as the only variable
factors in these equations. A combined cost per instruction was computed using the
formula

\[ \text{Cost}_{\text{Instruction}} = \frac{\text{kernel instructions}}{\text{total instructions} - \text{kernel instructions}} \]  \hspace{1cm} (7.4)

Cost per instruction was computed for each MCA configuration, as well as for the original
system, for the distributed application results of section 7.1. Using equation 2.7b, plots
showing system overhead and processor utilization are shown in Figures 7.9 and 7.10.
Figure 7.9 Overhead of MCA processor by hardware context for distributed application.
Figure 7.10 MCA processor utilization by hardware context for distributed application.
Chapter 8

Conclusions and Future Work

In this research we have developed a multiple hardware context processor architecture, MCA. This architecture has hardware support for fine and long term scheduling, and supports interprocess communication through the use of register windows. The register windows can be arbitrarily mapped between task address spaces. In addition, a distributed operating system, MC/OS, based on Minix 1.7.2 was developed and ported to the MCA architecture. MC/OS was designed to take full advantage of the extensions added to the MCA architecture in order to efficiently support scheduling and IPC.

Various configurations of the MCA architecture and MC/OS were evaluated through simulation of the hardware platform. Experiments were conducted using a number of different software applications exhibiting distinct execution profiles. Experimental results show that process switch and IPC overhead are both substantial on a conventional multitasking system. This is true even for computation intensive applications, and is especially apparent in distributed systems. Results clearly indicate the benefits of multiple hardware contexts, hardware scheduling support, and sharable register windows for supporting interprocess communication. The MCA processor architecture yields speedups of more than 45% for typical distributed applications when many contexts are available. Speedup is significant even with a limited number of available contexts. For
conventional applications, such as I/O and computationally intensive tasks, performance gains are also substantial. In addition, as the number of independent applications increases, the MCA processor exhibits little degradation in processor utilization whereas the single context multitasking system used as our benchmark degrades much more rapidly.

Task scheduling was also found to contribute overhead in multitasking systems. In a multiple context architecture such as MCA, experiments also reveal that the traditional scheduling mechanism can be improved. IPC driven scheduling reduced scheduling overhead by almost 50% in distributed applications. In addition, the performance of IPC driven versus round robin scheduling is better under higher scheduling loads.

Indirect windowed register access through window access registers provides significant flexibility in the use of registers. Indirection coupled with appropriate primitive instruction supports shared resource multithreading. This flexibility enables better utilization of scarce register resources and reduces memory access traffic by allowing more tasks and data to remain resident than is possible with existing multithreaded architectures.

Additional analysis and simulation is needed using a variety of distributed computing applications in order to evaluate the optimum size and number of a register windows and number of resident contexts required. The limitations of simulating an architecture platform have made more elaborate experiments of this nature impractical and very time consuming.
In addition, further work is necessary to determine the time and space complexity of the register dependency detection and scheduling mechanism of the stream selection unit (SSU). This will lead to the development of the SSU portion of the processor. In particular support for other methods of processor allocation to ensure fairness and response time are essential for execution of highly interactive applications.

Further extensions to the MCA processor and additional features in MC/OS include extension of the register window concept to general purpose registers, shared window access, and hardware support for asynchronous interprocess communication. Evaluation of these features will require a more sophisticated simulation environment, and development of a compiler for the MCA instruction set.

An additional extension includes support for rapid process/thread creation and destruction. This extension is of particular interest since the cost of creating and destroying threads can introduce excessive overhead if thread lifetimes are short. In addition, as the cost of thread creation is reduced, the prospects for further distributing applications is enhanced. It may be possible to greatly reduce the number of procedure call/return operations, which are also high overhead operations, by threading large procedures and using messages/register windows for argument passing.
References


46. Lawton, K., BOCHS software, copyright 1996. bochs@world.std.com.

47. UNIX on-line manual pages.
Appendix I

MC/OS Source Code

The MC/OS operating system, described in Chapter 5, is derived from the Minix 1.7.2 operating system. Appendix I describes the portions of Minix which were rewritten in order to implement the scheduling and IPC operations supported by MC/OS on the MCA processor. Further, both IPC directed and round-robin coarse grain scheduling algorithms are presented. In addition, pertinent routines from the unmodified Minix operating system are given at the end of this appendix. These routines are responsible for scheduling and IPC on a single context processor such as an 80x86 based system.

At startup, MC/OS is loaded by a standard boot loader which places the system initialization code in a fixed location in memory and then jumps to the first instruction in the MC/OS kernel. This represents the first phase, hardware initialization, of system initialization. The initialization process is illustrated in Figure I.1. The three source code files modified for the first phase are proc.h, table.c, and start.c. Other modules are also involved during initialization, but are not affected when ported to the MCA architecture.
Hardware initialization

extract hardware configuration (including # hardware contexts) from boot loader.

(proc.h, table.c, start.c)

-------------------

Initialize and start O/S

load process control block (proc), load hardware contexts, schedule tasks (clock, system, etc.), load and perform conventional context switch to CGS task.

(main.c, calls to proc.c)

-------------------

Normal execution state

enter on hardware interrupts, or direct calls from CGS task for scheduling and task management.

(proc.c, CGS.c, system.c)

Figure I.1 MC/OS kernel initialization process.

The second initialization step involves loading all system data structures, scheduling system tasks, and switching to the first system task. This is performed by functions within the file main.c and proc.c. At this point the hardware and system data areas are initialized and the system enters its normal execution state. Subsequent scheduling and task management (i.e., fork(), exec(), etc.) involve functions in proc.c, CGS.c, and system.c.

Each portion of MC/OS which differs from Minix 1.7.2 is presented in the following sections, accompanied by a description of the purpose of the code and differences between MC/OS and Minix 1.7.2.

I.1 Process Control Block
Task state information is stored in a process control block (proc) in Minix. This is also true in MC/OS. The structure, proc, stores both hardware state (e.g., register values) as well as system and accounting information. Additional fields were added at lines 39, 40, and 46. The field contents are as follows: p_getfrom stores the WhoID of a process for a pending IPC operation, the field p_state contains the SSR entry for the process, and p_slot contains the hardware slot number the task resides in, or -1 if the task is not currently resident.

```c
#ifndef PROC_H
#define PROC_H

/* Here is the declaration of the process table. It contains the process' */
/* registers, memory map, accounting, and message send/receive information. */
/* Many assembly code routines reference fields in it. The offsets to these */
/* fields are defined in the assembler include file sconst.h. When changing */
/* 'proc', be sure to change sconst.h to match. */
*/

struct proc {
    struct stackframe_s p_reg;   /* process' registers saved in stack frame */

    #if (CHIP == INTEL)
        reg_t p_ldt_sel;         /* selector in gdt giving ldt base and limit*/
        struct segdesc_s p_ldt[2];  /* local descriptors for code and data */
        /* 2 is LDT_SIZE - avoid include protect.h */
    #endif /* (CHIP == INTEL) */

    reg_t *p_stguard;           /* stack guard word */
    int p_nr;                  /* number of this process (for fast access) */
    int p_int_blocked;         /* nonzero if int msg blocked by busy task */
    int p_int_held;            /* nonzero if int msg held by busy syscall */
    struct proc *p_nextheld;   /* next in chain of held-up int processes */
    int p_flags;               /* P_SLOT_FREE, SENDING, RECEIVING, etc. */
    struct mem_map p_map[NR_SEGS];/* memory map */
    pid_t p_pid;               /* process id passed in from MM */
    clock_t user_time;        /* user time in ticks */
};
```
/* Bits for p_flags in proc[]. A process is runnable if p_flags == 0. */
#define P_SLOT_FREE 001 /* set when slot is not in use */
#define NO_MAP 002 /* keeps unmapped forked child from running */
#define SENDING 004 /* set when process blocked trying to send */
#define RECEIVING 010 /* set when process blocked trying to recv */
#define PENDING 020 /* set when inform() of signal pending */
#define SIG_PENDING 040 /* keeps to-be-signalled proc from running */
#define P_STOP 0100 /* set when process is being traced */
#define MAX_SLOTS 32 /* part of architecture, obviously */
#define MAX_SLOTS 32 /* part of architecture, obviously */
#define BEG_PROC_ADDR (&proc[0])
#define END_PROC_ADDR (&proc[NR_TASKS + NR_PROCS])
#define END_TASK_ADDR (&proc[NR_TASKS])
#define BEG_SERV_ADDR (&proc[NR_TASKS])
#define BEG_USER_ADDR (&proc[NR_TASKS + LOW_USER])
#define NIL_PROC (struct proc *) 0)
#define isidlehardware(n) ((n) == IDLE || (n) == HARDWARE)
#define isokprocn(n) ((unsigned) ((n) + NR_TASKS) < NR_PROCS + NR_TASKS)
#define isoksrc_dest(n) (isokprocn(n) || (n) == ANY)
#define isoksusern(n) ((unsigned) (n) < NR_PROCS)
#define isrshardware(n) ((unsigned) ((n) - LOW_USER) < NR_PROCS - LOW_USER)
#define issysentn(n) (n == FS_PROC_NR || (n) == MM_PROC_NR)
#define istaskp(p) ((p) < END_TASK_ADDR && (p) != proc_addr(IDLE))

/* Magic process table addresses. */
#define BEG_PROC_ADDR (&proc[0])
#define END_PROC_ADDR (&proc[NR_TASKS + NR_PROCS])
#define END_TASK_ADDR (&proc[NR_TASKS])
#define BEG_SERV_ADDR (&proc[NR_TASKS])
#define BEG_USER_ADDR (&proc[NR_TASKS + LOW_USER])
#define NIL_PROC (struct proc *) 0)
#define isidlehardware(n) ((n) == IDLE || (n) == HARDWARE)
#define isokprocn(n) ((unsigned) ((n) + NR_TASKS) < NR_PROCS + NR_TASKS)
#define isoksrc_dest(n) (isokprocn(n) || (n) == ANY)
#define isoksusern(n) ((unsigned) (n) < NR_PROCS)
#define isrshardware(n) ((unsigned) ((n) - LOW_USER) < NR_PROCS - LOW_USER)
#define issysentn(n) (n == FS_PROC_NR || (n) == MM_PROC_NR)
#define istaskp(p) ((p) < END_TASK_ADDR && (p) != proc_addr(IDLE))

/* Bits for p_flags in proc[]. A process is runnable if p_flags == 0. */
#define P_SLOT_FREE 001 /* set when slot is not in use */
#define NO_MAP 002 /* keeps unmapped forked child from running */
#define SENDING 004 /* set when process blocked trying to send */
#define RECEIVING 010 /* set when process blocked trying to recv */
#define PENDING 020 /* set when inform() of signal pending */
#define SIG_PENDING 040 /* keeps to-be-signalled proc from running */
#define P_STOP 0100 /* set when process is being traced */
#define MAX_SLOTS 32 /* part of architecture, obviously */
#define MAX_SLOTS 32 /* part of architecture, obviously */
#define BEG_PROC_ADDR (&proc[0])
#define END_PROC_ADDR (&proc[NR_TASKS + NR_PROCS])
#define END_TASK_ADDR (&proc[NR_TASKS])
#define BEG_SERV_ADDR (&proc[NR_TASKS])
#define BEG_USER_ADDR (&proc[NR_TASKS + LOW_USER])
#define NIL_PROC (struct proc *) 0)
#define isidlehardware(n) ((n) == IDLE || (n) == HARDWARE)
#define isokprocn(n) ((unsigned) ((n) + NR_TASKS) < NR_PROCS + NR_TASKS)
#define isoksrc_dest(n) (isokprocn(n) || (n) == ANY)
#define isoksusern(n) ((unsigned) (n) < NR_PROCS)
#define isrshardware(n) ((unsigned) ((n) - LOW_USER) < NR_PROCS - LOW_USER)
#define issysentn(n) (n == FS_PROC_NR || (n) == MM_PROC_NR)
#define istaskp(p) ((p) < END_TASK_ADDR && (p) != proc_addr(IDLE))

struct proc *p_nextready; /* pointer to next ready process */
unsigned p_pendcount; /* count of pending and unfinished signals */

int slot_num; /* current CPU context slot, -1=non-resident */
char p_name[16]; /* name of the process */

};

/* Guard word for task stacks. */
#define STACK_GUARD ((reg_t) (sizeof(reg_t) == 2 ? 0xBEEF : 0xDEADBEEF))

/* define for # resident contexts, should parameter passed by boot?? */
#define MAX_SLOTS 32 /* part of architecture, obviously */

/* Magic process table addresses. */
#define BEG_PROC_ADDR (&proc[0])
#define END_PROC_ADDR (&proc[NR_TASKS + NR_PROCS])
#define END_TASK_ADDR (&proc[NR_TASKS])
#define BEG_SERV_ADDR (&proc[NR_TASKS])
#define BEG_USER_ADDR (&proc[NR_TASKS + LOW_USER])
#define NIL_PROC (struct proc *) 0)
#define isidlehardware(n) ((n) == IDLE || (n) == HARDWARE)
#define isokprocn(n) ((unsigned) ((n) + NR_TASKS) < NR_PROCS + NR_TASKS)
#define isoksrc_dest(n) (isokprocn(n) || (n) == ANY)
#define isoksusern(n) ((unsigned) (n) < NR_PROCS)
#define isrshardware(n) ((unsigned) ((n) - LOW_USER) < NR_PROCS - LOW_USER)
#define issysentn(n) (n == FS_PROC_NR || (n) == MM_PROC_NR)
#define istaskp(p) ((p) < END_TASK_ADDR && (p) != proc_addr(IDLE))

/* Bits for p_flags in proc[]. A process is runnable if p_flags == 0. */
#define P_SLOT_FREE 001 /* set when slot is not in use */
#define NO_MAP 002 /* keeps unmapped forked child from running */
#define SENDING 004 /* set when process blocked trying to send */
#define RECEIVING 010 /* set when process blocked trying to recv */
#define PENDING 020 /* set when inform() of signal pending */
#define SIG_PENDING 040 /* keeps to-be-signalled proc from running */
#define P_STOP 0100 /* set when process is being traced */
#define MAX_SLOTS 32 /* part of architecture, obviously */
#define MAX_SLOTS 32 /* part of architecture, obviously */
#define BEG_PROC_ADDR (&proc[0])
#define END_PROC_ADDR (&proc[NR_TASKS + NR_PROCS])
#define END_TASK_ADDR (&proc[NR_TASKS])
#define BEG_SERV_ADDR (&proc[NR_TASKS])
#define BEG_USER_ADDR (&proc[NR_TASKS + LOW_USER])
#define NIL_PROC (struct proc *) 0)
#define isidlehardware(n) ((n) == IDLE || (n) == HARDWARE)
#define isokprocn(n) ((unsigned) ((n) + NR_TASKS) < NR_PROCS + NR_TASKS)
#define isoksrc_dest(n) (isokprocn(n) || (n) == ANY)
#define isoksusern(n) ((unsigned) (n) < NR_PROCS)
#define isrshardware(n) ((unsigned) ((n) - LOW_USER) < NR_PROCS - LOW_USER)
#define issysentn(n) (n == FS_PROC_NR || (n) == MM_PROC_NR)
#define istaskp(p) ((p) < END_TASK_ADDR && (p) != proc_addr(IDLE))

struct proc *p_nextready; /* pointer to next ready process */
unsigned p_pendcount; /* count of pending and unfinished signals */

int slot_num; /* current CPU context slot, -1=non-resident */
char p_name[16]; /* name of the process */

};

/* Guard word for task stacks. */
#define STACK_GUARD ((reg_t) (sizeof(reg_t) == 2 ? 0xBEEF : 0xDEADBEEF))

/* define for # resident contexts, should parameter passed by boot?? */
#define MAX_SLOTS 32 /* part of architecture, obviously */

/* Magic process table addresses. */
#define BEG_PROC_ADDR (&proc[0])
#define END_PROC_ADDR (&proc[NR_TASKS + NR_PROCS])
#define END_TASK_ADDR (&proc[NR_TASKS])
#define BEG_SERV_ADDR (&proc[NR_TASKS])
#define BEG_USER_ADDR (&proc[NR_TASKS + LOW_USER])
#define NIL_PROC (struct proc *) 0)
#define isidlehardware(n) ((n) == IDLE || (n) == HARDWARE)
#define isokprocn(n) ((unsigned) ((n) + NR_TASKS) < NR_PROCS + NR_TASKS)
#define isoksrc_dest(n) (isokprocn(n) || (n) == ANY)
#define isoksusern(n) ((unsigned) (n) < NR_PROCS)
#define isrshardware(n) ((unsigned) ((n) - LOW_USER) < NR_PROCS - LOW_USER)
#define issysentn(n) (n == FS_PROC_NR || (n) == MM_PROC_NR)
#define istaskp(p) ((p) < END_TASK_ADDR && (p) != proc_addr(IDLE))
Additional changes in proc.h are marked with an ‘*’, and occur in lines 63-105. The constant NR_SLOTS defines the maximum number of hardware contexts MC/OS supports.

The remaining fields are used to maintain the active task list, and to manage hardware contexts. The array mcpi used to keep track of which task is resident in each hardware context slot, while slots is a count of the number of empty (available) slots.

### 1.2 Task Initialization Table

Tasks and user level server processes such as the memory manager (mm), file system (fs), and user interface manager (init) are registered in this module, table.c. There was a very minor name change, from idle to CGS, made in this file. It is included mainly to make it
easier to see the number of separate tasks and processes comprising the MC/OS (and Minix) operating system.

```c
/* The purpose of "table.c" contains all the data. In the *.h files,
 * declared variables appear with EXTERN in front of them, as in
 *
 * EXTERN int x;
 *
 * Normally EXTERN is defined as extern, so when they are included in another
 * file, no storage is allocated. If the EXTERN were not present, but just
 * say,
 *
 * int x;
 *
 * then including this file in several source files would cause 'x' to be
 * declared several times. While some linkers accept this, others do not,
 * so they are declared extern when included normally. However, it must
 * be declared for real somewhere. That is done here, by redefining
 * EXTERN as the null string, so the inclusion of all the *.h files in
 * table.c actually generates storage for them. All the initialized
 * variables are also declared here, since
 *
 * extern int x = 4;
 *
 * is not allowed. If such variables are shared, they must also be declared
 * in one of the *.h files without the initialization.
 */

#define _TABLE

#include "kernel.h"
#include <termios.h>
#include <minix/com.h>
#include "proc.h"
#include "tty.h"

/* The startup routine of each task is given below, from -NR_TASKS upwards.
 * The order of the names here MUST agree with the numerical values assigned to
 * the tasks in <minix/com.h>.
 */
#define SMALL_STACK (128 * sizeof(char *))
#define TTY_STACK (3 * SMALL_STACK)
#define SYN_ALRM_STACK SMALL_STACK
```
#define DP8390_STACK (SMALL_STACK * ENABLE_NETWORKING)
#define IDLE_STACK SMALL_STACK
#define PRINTER_STACK SMALL_STACK
#if (CHIP == INTEL)
#define WINCH_STACK (2 * SMALL_STACK * ENABLE_WINI)
#else
#define WINCH_STACK (3 * SMALL_STACK * ENABLE_WINI)
#endif
#if (MACHINE == ATARI)
#define SCSI_STACK (3 * SMALL_STACK)
#endif
#if (MACHINE == IBM_PC)
#define SCSI_STACK (2 * SMALL_STACK * ENABLE_SCSI)
#endif
/* XXX can be smaller? */
#define CDROM_STACK (4 * SMALL_STACK * ENABLE_CDROM)
#define AUDIO_STACK (4 * SMALL_STACK * ENABLE_AUDIO)
#define MIXER_STACK (4 * SMALL_STACK * ENABLE_AUDIO)
#define FLOP_STACK (3 * SMALL_STACK)
#define MEM_STACK SMALL_STACK
#define CLOCK_STACK SMALL_STACK
#define SYS_STACK SMALL_STACK
#define HARDWARE_STACK 0 /* dummy task, uses kernel stack */
#define TOT_STACK_SPACE (TTY_STACK + DP8390_STACK + SCSI_STACK + SYN_ALRM_STACK + IDLE_STACK + HARDWARE_STACK + PRINTER_STACK + WINCH_STACK + FLOP_STACK + MEM_STACK + CLOCK_STACK + SYS_STACK + CDROM_STACK + AUDIO_STACK + MIXER_STACK)
/* SCSI, CDROM and AUDIO may in the future have different choices like WINCHESTER, but for now the choice is fixed. */
#define scsi_task aha_scsi_task
#define cdrom_task mcd_task
#define audio_task dsp_task
Some notes about the following table:

1) The tty_task should always be first so that other tasks can use printf if their initialisation has problems.
2) If you add a new kernel task, add it before the printer task.
3) The task name is used for the process name (p_name).

PUBLIC struct tasktab tasktab[] = {
    { tty_task, TTY_STACK, "TTY" },
#if ENABLE_NETWORKING
    { dp8390_task, DP8390_STACK, "DP8390" },
#endif
#if ENABLE_CDROM
    { cdrom_task, CDROM_STACK, "CDROM" },
#endif
#if ENABLE_AUDIO
    { audio_task, AUDIO_STACK, "AUDIO" },
    { mixer_task, MIXER_STACK, "MIXER" },
#endif
#if ENABLE_SCSI
    { scsi_task, SCSI_STACK, "SCSI" },
#endif
#if ENABLE_WINI
    { winchester_task, WINCH_STACK, "WINCH" },
#endif
    { syn_alrm_task, SYN_ALRM_STACK, "SYN_AL" },
    { cgs_task, IDLE_STACK, "CGS" },
    { printer_task, PRINTER_STACK, "PRINTER" },
    { floppy_task, FLOP_STACK, "FLOPPY" },
    { mem_task, MEM_STACK, "MEMORY" },
    { clock_task, CLOCK_STACK, "CLOCK" },
    { sys_task, SYS_STACK, "SYS" },
    { 0, HARDWARE_STACK, "HARDWARE" },
    { 0, 0, "MM" },
    { 0, 0, "FS" },
#if ENABLE_NETWORKING
    { 0, 0, "INET" },
#endif
    { 0, 0, "INIT" },
};

/* Stack space for all the task stacks. (Declared as (char *) to align it.) */
PUBLIC char *t_stack[TOT_STACK_SPACE / sizeof(char *)];

* The number of kernel tasks must be the same as NR_TASKS.
* If NR_TASKS is not correct then you will get the compile error:
* "array size is negative"
*/

#define NKT (sizeof tasktab / sizeof (struct tasktab) - (INIT_PROC_NR + 1))

extern int dummy_tasktab_check[NR_TASKS == NKT ? 1 : -1];

I.3 Hardware Initialization Routine

Hardware initialization is performed within the module start.c, shown below. Hardware
dependent initialization is performed. Where necessary, some sections must be written in
assembly code in order to directly access processor registers. This code is found in
mpx386.s, and was not modified in this research. A single line, line 56, was added to enable
MC/OS to read the number of hardware contexts environment variable passed from the boot
loader. This makes it possible to run MC/OS on systems with different hardware
configurations without needing to reconfigure and recompile the system.

/* This file contains the C startup code for Minix on Intel processors.
* It cooperates with mpx.s to set up a good environment for main().
* This code runs in real mode for a 16 bit kernel and may have to switch
* to protected mode for a 286.
* For a 32 bit kernel this already runs in protected mode, but the selectors
* are still those given by the BIOS with interrupts disabled, so the
* descriptors need to be reloaded and interrupt descriptors made.
*/

#include "kernel.h"
#include <string.h>
#include <stdlib.h>
#include <minix/boot.h>
#include "protect.h"

PRIVATE char k_environ[256]; /* environment strings passed by loader */
int NR_SLOTS; /* number of cpu slots in MCP mode */
FORWARD_PROTOTYPE(int k_atoi, (char *s));

PUBLIC void cstart(cs, ds, mcs, mds, parmoff, parmsgsz)
U16_t cs, ds; /* Kernel code and data segment */
U16_t mcs, mds; /* Monitor code and data segment */
U16_t parmoff, parmsgsz; /* boot parameters offset and length */
{
/* Perform system initializations prior to calling main(). */

register char *envp;
phys_bytes mcode_base, mdata_base;
unsigned mon_start;

/* Record where the kernel and the monitor are. */
code_base = seg2phys(cs);
data_base = seg2phys(ds);
mcode_base = seg2phys(mcs);
mdata_base = seg2phys(mds);

/* Initialize protected mode descriptors. */
prot_init();

/* Copy the boot parameters to kernel memory. */
if (parmsgsz > sizeof k_environ - 2) parmsgsz = sizeof k_environ - 2;
phys_copy(mdata_base + parmoff, vir2phys(k_environ), (phys_bytes) parmsgsz);

/* Convert important boot environment variables. */
boot_parameters.bp_rootdev = k_atoi(k_getenv("rootdev"));
boot_parameters.bp_ramimagedev = k_atoi(k_getenv("ramimagedev"));
boot_parameters.bp_ramsize = k_atoi(k_getenv("ramsize"));
boot_parameters.bp_processor = k_atoi(k_getenv("processor"));
NR_SLOTS = k_atoi(k_getenv("slots"));

/* Type of VDU: */
envp = k_getenv("video");
if (strcmp(envp, "ega") == 0) ega = TRUE;
if (strcmp(envp, "vga") == 0) vga = ega = TRUE;

/* Memory sizes: */
low_memsize = k_atoi(k_getenv("memsize"));
ext_memsize = k_atoi(k_getenv("emssize"));

/* Processor? */
processor = boot_parameters.bp_processor; /* 86, 186, 286, 386, ... */
/* XT, AT or MCA bus? */
extp = k_getenv("bus");
if (envp == NIL_PTR || strcmp(envp, "at") == 0) {
    pc_at = TRUE;
} else
if (strcmp(envp, "mca") == 0) {
    pc_at = ps_mca = TRUE;
}

/* Decide if mode is protected. */
#if _WORD_SIZE == 2
protected_mode = processor >= 286;
#endif

/* Is there a monitor to return to? If so then keep it safe. */
if (!protected_mode) mon_return = 0;
mon_start = mcode_base / 1024;
if (mon_return && low_memsiz > mon_start) low_memsiz = mon_start;

/* Return to assembler code to switch to protected mode (if 286), reload
* selectors and call main(). */

I.4 MC/OS System Initialization and Loading

The second step in loading MC/OS, described above, is performed by the function main() in main.c below. Changes were made in this function to initialize MCA data structures and to mark tasks as runnable, p_state=1, and not resident, p_slot=-1. Each task is then added to the active list, through a call to ready(), which also makes tasks resident provided the number of available hardware contexts, slots, is positive. In addition, a minor optimization was added to predefine the hardware interrupt message. Hardware interrupts are always the same, no information other than that one has occurred is known. Finally, a call to restart() results in a conventional context switch, in hardware slot 0, to the CGS task.
/* This file contains the main program of MINIX. The routine main() initializes the system and starts the ball rolling by setting up the proc table, interrupt vectors, and scheduling each task to run to initialize itself. */

/* The entries into this file are: */
main: MINIX main program
panic: abort MINIX due to a fatal error

#include "kernel.h"
#include <signal.h>
#include <unistd.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"

* 17 extern message hw_int;

/*============================================================*
main
*============================================================*/
PUBLIC void main()
{
 /* Start the ball rolling. */

 register struct proc *rp;
 register int t;
 int sizeindex;
 phys_clicks text_base;
 vir_clicks text_clicks;
 vir_clicks data_clicks;
 phys_bytes phys_b;
 reg_t ktsb; /* kernel task stack base */
 struct memory *memp;
 struct tasktab *ttp;

 /* Initialize the interrupt controller. */
intr_init(1);

 /* Interpret memory sizes. */
mem_init();

* 43 hw_int.m_source=HARDWARE; /* hard ints always the same */
* 44 hw_int.m_type=HARD_INT;

 /* Clear the process table.
 * Set up mappings for proc_addr() and proc_number() macros.
 */
for (rp = BEG_PROC_ADDR, t = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++t)
{
    rp->p_flags = P_SLOT_FREE;
    *rp->slot_num = -1; /* not resident yet */
    *rp->p_state = 1; /* runnable state */
    rp->p_int_blocked = FALSE;
    rp->p_nr = t; /* proc number from ptr */
    (pproc_addr + NR_TASKS)[t] = rp; /* proc ptr from number */
}

slots = NR_SLOTS - 1; /* how many cpu slots available */
for (t = 0; t < NR_SLOTS; t++) /* all slots empty at moment */
mcpi[t] = -99; /* 0 ALWAYS is kernel slot */
/* but we never look at it anyway */

/* Set up proc table entries for tasks and servers. The stacks of the */
/* kernel tasks are initialized to an array in data space. The stacks */
/* of the servers have been added to the data segment by the monitor, so */
/* the stack pointer is set to the end of the data segment. All the */
/* processes are in low memory on the 8086. On the 386 only the kernel */
/* is in low memory, the rest is loaded in extended memory. */

/* Task stacks. */
ktsb = (reg_t) t_stack;

/* Now going multiple context, so clear system counters */
/* Used for testing only */
sendmcp(238, 0, 0);
/* */

for (t = -NR_TASKS; t <= LOW_USER; ++t)
{
    rp = proc_addr(t); /* t's process slot */
    ttp = &tasktab[t + NR_TASKS]; /* t's task attributes */
    strcpy(rp->p_name, ttp->name);
    if (t < 0)
    {
        if (ttp->stksize > 0)
        {
            rp->p_stguard = (reg_t *) ktsb;
            *rp->p_stguard = STACK_GUARD;
        }
        ktsb += ttp->stksize;
        rp->p_reg.sp = ktsb;
        text_base = code_base >> CLICK_SHIFT;
        /* tasks are all in the kernel */
        /* and use the full kernel sizes */
    }
}
memp = &mem[0]; /* remove from this memory chunk */
} 

else
sizeindex = 2 * t + 2; /* MM, FS, INIT have their own sizes */

rp->p_reg.pc = (reg_t) ttp->initial_pc;
rp->p_reg.psw = istaskp(rp) ? INIT_TASK_PSW : INIT_PSW;

if (t >= 0) { 
    memp = &mem[O];
text_base = Oxl00000» CLICK_SHIFT;
    /* remove from this memory chunk */memp = &mem[O];
rp->p_flags = 0;
alloc_segments(rp);
if (!isidlehardware(t))
    ready(rp); /* IDLE, HARDWARE neveready */

    /* Initialize the server stack pointer. Take it down one word */
    * to give crtsos something to use as "argc".
    */
    rp->p_reg.sp = (reg_t) (rp->p_map[S].mem_vir +
                   rp->p_map[S].mem_len) << CLICK_SHIFT;
    rp->p_reg.sp -= sizeof(reg_t);
}

else
sizeindex = 2 * t + 2; /* MM, FS, INIT have their own sizes */

rp->p_reg.pc = (reg_t) ttp->initial_pc;
rp->p_reg.psw = istaskp(rp) ? INIT_TASK_PSW : INIT_PSW;

text_clicks = sizes[sizeindex];
data_clicks = sizes[sizeindex + 1];
rp->p_map[T].mem_phys = text_base;
rp->p_map[T].mem_len = text_clicks;
rp->p_map[D].mem_phys = text_base + text_clicks;
rp->p_map[D].mem_len = data_clicks;
rp->p_map[S].mem_phys = text_base + text_clicks + data_clicks;
rp->p_map[S].mem_vir = data_clicks; /* empty - stack is in data */
text_base += text_clicks + data_clicks; /* ready for next, if server */
memp->size -= (text_base - memp->base);
memp->base = text_base; /* memory no longer free */

if (t >= 0) 
   
   /* Initialize the server stack pointer. Take it down one word */
   * to give crtsos something to use as "argc".
   */
   rp->p_reg.sp = (reg_t) (rp->p_map[S].mem_vir +
                     rp->p_map[S].mem_len) << CLICK_SHIFT;
   rp->p_reg.sp -= sizeof(reg_t);
}

#if WORD_SIZE == 4
/* Servers are loaded in extended memory if in 386 mode. */
if (t < 0) 
{
    memp = &mem[1];
text_base = 0x100000 >> CLICK_SHIFT;
}
#endif

rp->p_flags = 0;
alloc_segments(rp);
if (!isidlehardware(t))
    ready(rp); /* IDLE, HARDWARE neveready */

proc[NR_TASKS+INIT_PROC_NR].p_pid = 1; /* INIT of course has pid 1 */
idle_ptr = bill_ptr = proc_addr(IDLE); /* has to point somewhere */
proc_ptr = idle_ptr;
/* Now go to the assembly code to start running the current process. */
* 146   restart();  /* conventional switch to CGS task */
* 147   */ /* had also called pick_proc() before restart */

1.5 Clock and SystemTasks

The kernel clock task is included here because they are loaded and initialized at system startup. After startup, the clock task processes real time clock interrupts and manages program timers (alarms) as well as providing coarse and fine-grain scheduling assistance by checking task time quantums.

/* clock.c
* This file contains the code and data for the clock task. The clock task
* accepts six message types:
* *
* HARD_INT: a clock interrupt has occurred
* GET_UPTIME: get the time since boot in ticks
* GET_TIME: a process wants the real time in seconds
* SET_TIME: a process wants to set the real time in seconds
* SET_ALARM: a process wants to be alerted after a specified interval
* SET_SYN_AL: set the sync alarm
* *
* The input message is format m6. The parameters are as follows:
* *
* m_type   CLOCK_PROC   FUNC   NEW_TIME
* ---------------------------------------------
* | HARD_INT |       |       |
* |-----------+-------+-------|
* | GET_UPTIME|       |       |
* |-----------+-------+-------|
* | GET_TIME  |       |       |
* |-----------+-------+-------|
* | SET_TIME  |       |       |
* |-----------+-------+-------|
* | SET_ALARM | proc_nr |       |
* |-----------+-------+-------|
* | SET_SYN_AL| proc_nr |       |
* |-----------+-------+-------|
* | NEW_TIME  | DELTA_CLICKS, and SECONDS_LEFT all refer to the same field in
* | the message, depending upon the message type. 
* *
* Reply messages are of type OK, except in the case of a HARD_INT, to
* which no reply is generated. For the GET_* messages the time is returned
* in the NEW_TIME field, and for the SET_ALARM and SET_SYN_ALARM the time
* in seconds remaining until the alarm is returned is returned in the same
* field.

* When an alarm goes off, if the caller is a user process, a SIGALRM signal
* is sent to it. If it is a task, a function specified by the caller will
* be invoked. This function may, for example, send a message, but only if
* it is certain that the task will be blocked when the timer goes off. A
* synchronous alarm sends a message to the synchronous alarm task, which
* in turn can dispatch a message to another server. This is the only way
* to send an alarm to a server, since servers cannot use the function-call
* mechanism available to tasks and servers cannot receive signals.

#include "kernel.h"
#include <signal.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"

/* Constant definitions. */
#define MILLISEC 100 /* how often to call the scheduler (msec) */
#define SCHED_RATE (MILLISEC*HZ/1000) /* number of ticks per schedule */

/* Clock parameters. */
#if (CHIP == INTEL)
#define COUNTER_FREQ (2*TIMER_FREQ) /* counter frequency using square wave*/
#define LATCH_COUNT Ox00 /* ccOOxxxx, c = channel, x = any */
#define SQUARE_WAVE Ox36 /* ccaammmb, a = access, m = mode, b = BCD */
/* llxl1, 11 = LSB then MSB, xl l = sq wave */
#define TIMER_COUNT (unsigned) (TIMER_FREQ/HZ)) /* initial value for counter*/
#define TIMER_FREQ 1193182L /* clock frequency for timer in PC and AT */
#endif

#define CLOCK_ACK_BIT 0x80 /* PS/2 clock interrupt acknowledge bit */

#if (CHIP == M68000)
#define TIMER_FREQ 2457600L
#endif

/* Clock task variables. */
PUBLIC clock_t realtime; /* real time clock, used by idle ONLY */
PRIVATE time_t boot_time; /* time in seconds of system boot */
PUBLIC clock_t next_alarm; /* probable time of next alarm */
PRIVATE message mc; /* message buffer for both input and output */
PRIVATE int watchdogproc; /* contains proc_nr at call of *watch_dog[]*/
PRIVATE watchdog_t watch_dog[NR_TASKS+NR_PROCS];
/* Variables used by both clock task and synchronous alarm task * /
PRIVATE int syn_al_alive= TRUE; /* don't wake syn_alrm_task before inited */
PRIVATE int syn_table[NR_TASKS+NR_PROCS]; /* which tasks get CLOCK_INT */
/* Variables changed by interrupt handler */
PUBLIC clock_t pending_ticks; /* ticks seen by low level only */
PRIVATE int sched_ticks = SCHED_RATE; /* counter: when 0, call scheduler */
PRIVATE struct proc *prev_ptr; /* last user process run by clock task */
FORWARD _PROTOTYPE( void common_setalarm, (int proc_nr,
  long delta_ticks, watchdog_t function));
FORWARD _PROTOTYPE( void do_clocktick, (void));
FORWARD _PROTOTYPE( void do_get_time, (void));
FORWARD _PROTOTYPE( void do_getuptime, (void));
FORWARD _PROTOTYPE( void do_set_time, (message *m_ptr));
FORWARD _PROTOTYPE( void do_setalarm, (message *m_ptr));
FORWARD _PROTOTYPE( void init_clock, (void));
FORWARD _PROTOTYPE( void cause_alarm, (void));
FORWARD _PROTOTYPE( void do_setsyn_alrm, (message *m_ptr));
FORWARD _PROTOTYPE( int clock_handler, (int irq));
/* ==============================================================*/
/* clock task */
/* ==============================================================*/
PUBLIC void clock_task()
{
/* Main program of clock task. It corrects realtime by adding pending
  * ticks seen only by the interrupt service, then it determines which
  * of the 6 possible calls this is by looking at 'mc.m_type'. Then
  * it dispatches.
  */
  int opcode;
  init_clock(); /* initialize clock task */
/* Main loop of the clock task. Get work, process it, sometimes reply. */
while (TRUE) {
  receive(ANY, &mc); /* go get a message */
  opcode = mc.m_type; /* extract the function code */
  lock();
  realtime += pending_ticks; /* transfer ticks from low level handler */
  pending_ticks = 0; /* so we don't have to worry about them */
  unlock();
}
switch (opcode) {
    case HARD_INT:  do_clocktick();  break;
    case GET_UPTIME: do_getuptime();  break;
    case GET_TIME:   do_get_time();   break;
    case SET_TIME:   do_set_time(&mc); break;
    case SET_ALARM:  do_setalarm(&mc); break;
    case SET_SYNC_AL:do_setsyn_alrm(&mc); break;
    default:
        *sendmcp(247,49,mc.m_source);  /* system diagnostic */
        panic("clock task got bad message", mc.m_type);
    }
    /* Send reply, except for clock tick. */
    mc.m_type = OK;
    if (opcode != HARD_INT) send(mc.m_source, &mc);
}

/*==========================================================*
 * do clocktick *
 *==========================================================*/
PRIVATE void do_clocktick()
{
    /* Despite its name, this routine is not called on every clock tick. It
     * is called on those clock ticks when a lot of work needs to be done.
     */

    register struct proc *rp;
    register int proc_nr;
    if (next_alarm <= realtime)
        { /* An alarm may have gone off, but proc may have exited, so check. */
            next_alarm = LONG_MAX;  /* start computing next alarm */
            for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++)
            {
                if (rp->p_alarm != 0)
                    { /* See if this alarm time has been reached. */
                        if (rp->p_alarm <= realtime)
                            { /* A timer has gone off. If it is a user proc,
                                * send it a signal. If it is a task, call the
                                * function previously specified by the task.
                                */
                                proc_nr = proc_number(rp);
                                if (watch_dog[proc_nr+NR_TASKS])
                                    {
                                        watchdog_proc = proc_nr;
                                        (*watch_dog[proc_nr+NR_TASKS])();
                                    }
                    }
            }
        }

        /* Send reply, except for clock tick. */
        mc.m_type = OK;
        if (opcode != HARD_INT) send(mc.m_source, &mc);
}
The system task was significantly modified to support task management in a multiple context architecture. For example, the fork() system call had to be changed to retrieve state information from either the process control block, proc, if the forking task is not resident, or by accessing hardware state information in the processor if the task is still resident. The possibility that tasks may still be resident, which is not possible in a single context processor, adds considerable complexity to the system calls relating to task management. Changes made to support this are highlighted and shown in italics.
* MM and FS, a library is provided with routines whose names are of the
form sys_xxx, e.g. sys_xit sends the SYS_XIT message to sys_task. The
message types and parameters are:

* SYS_FORK informs kernel that a process has forked
* SYS_NEWMAP allows MM to set up a process memory map
* SYS_GETMAP allows MM to get a process' memory map
* SYS_EXEC sets program counter and stack pointer after EXEC
* SYS_XIT informs kernel that a process has exited
* SYS_GETSP caller wants to read out some process' stack pointer
* SYS_TIMES caller wants to get accounting times for a process
* SYS_ABORT MM or FS cannot go on; abort MINIX
* SYS_FRESH start with a fresh process image during EXEC (68000 only)

# if ENABLE_COMPAT
* SYS_OLDSIG send a signal to a process (old style)
# endif

* SYS_SENDSIG send a signal to a process (POSIX style)
* SYS_SIGRETURN complete POSIX-style signalling
* SYS_KILL cause a signal to be sent via MM
* SYS_ENDSIG finish up after SYS_KILL-type signal
* SYS_COPY request a block of data to be copied between processes
* SYS_VCOPY request a series of data blocks to be copied between procs
* SYS_GBOOT copies the boot parameters to a process
* SYS_MEM returns the next free chunk of physical memory
* SYS_UMAP compute the physical address for a given virtual address
* SYS_TRACE request a trace operation

* In addition to the main sys_task() entry point, there are 5 other minor
entry points:
* cause_sig: take action to cause a signal to occur, sooner or later
* inform: tell MM about pending signals
* numap: umap D segment starting from process number instead of pointer
* umap: compute the physical address for a given virtual address
* alloc_segments: allocate segments for 8088 or higher processor
* build_sig: put old-style signal info on stack (68000, Intel ver. in klib)
*/

#include "kernel.h"
#include <signal.h>
#include <unistd.h>
#include <sys/sigcontext.h>
#include <sys/ptrace.h>
#include <minix/boot.h>
#include <minix/com.h>
#include "proc.h"
#if (CHIP == INTEL)
#include "protect.h"
#endif

/* set when modifying ready Q */

/* PSW masks. */
#define IF_MASK 0x0000200
#define IOPL_MASK 0x003000

message sys_m;
PRIVATE char sig_stuff[SIG_PUSH_BYTES]; /* used to send signals to processes */

FORWARD _PROTOTYPE( int do_abort, (message *m_ptr));
FORWARD _PROTOTYPE( int do_copy, (message *m_ptr));
FORWARD _PROTOTYPE( int do_exec, (message *m_ptr));
FORWARD _PROTOTYPE( int do_fork, (message *m_ptr));
FORWARD _PROTOTYPE( int do_gboot, (message *m_ptr));
FORWARD _PROTOTYPE( int do_getsp, (message *m_ptr));
FORWARD _PROTOTYPE( int do_kill, (message *m_ptr));
FORWARD _PROTOTYPE( int do_mem, (message *m_ptr));
FORWARD _PROTOTYPE( int do_newmap, (message *m_ptr));
FORWARD _PROTOTYPE( int do_sendsig, (message *m_ptr));
#if ENABLE_COMPAT
FORWARD _PROTOTYPE( int do_oldsig, (message *m_ptr));
#endif
#ifndef
FORWARD _PROTOTYPE( int do_sigreturn, (message *m_ptr));
FORWARD _PROTOTYPE( int do_endsig, (message *m_ptr));
FORWARD _PROTOTYPE( int do_times, (message *m_ptr));
FORWARD _PROTOTYPE( int do_trace, (message *m_ptr));
FORWARD _PROTOTYPE( int do_umap, (message *m_ptr));
FORWARD _PROTOTYPE( int do_xit, (message *m_ptr));
FORWARD _PROTOTYPE( int do_vcopy, (message *m_ptr));
FORWARD _PROTOTYPE( int do_getmap, (message *m_ptr));
#endif

/* PSW masks. */

#ifndef
    #if (CHIP == INTEL)
        #include "protect.h"
    #endif
    extern unsigned char switching;
#endif

PUBLIC void sys_task()
{
    /* Main entry point of sys_task. Get the message and dispatch on type. */
/* pointer to request message */

r = do_kill(&sys_m); break;

r = do_endsig(&sys_m);

r = do_copy(&sys_m); break;

r = do_vcopy(&sys_m); break;

r = do_gboot(&sys_m); break;

r = do_mem(&sys_m);

r = do_umap(&sys_m); break;

r = do_trace(&sys_m); break;

r = E_BAD_FCN;

r = do_sendsig(&sys_m);

r = do_sigretum(&sys_m); break;

sys_m.m_type = r; /* 'r' reports status of call */

send(sys_m.m_source, &sys_m); /* send reply to caller */

/*=============================================================*

| do fork |

*=============================================================*/

PRIVATE int do_fork(m_ptr)

register message *m_ptr; /* pointer to request message */

{ /* Handle sys_fork(). m_ptr->PROC1 has forked. The child is m_ptr->PROC2. */

#if (CHIP == INTEL)

reg_t old_ldt_set;

#endif
register struct proc *rpc;
struct proc *rpp;

if (!isoksuser(m_ptr->PROC1) || !isoksuser(m_ptr->PROC2))
    return(E_BAD_PROC);

switching=1;  /* this is critical now */
rpp = proc_addr(m_ptr->PROC1);
rpc = proc_addr(m_ptr->PROC2);

/* Copy parent 'proc' struct to child. */
#if (CHIP == INTEL)
old_ldt_sel = rpc->p_ldt_sel;  /* stop this being obliterated by copy */
#endif

if (rpp->slot_num>0)
{
    sendmcp(252,rpp->slot_num,rpp);  /*"save" context of resident task */
rpp->p_state=sendmcp(248,rpp->slot_num,0);
rpp->p_getfrom=sendmcp(248,rpp->slot_num,1);
rpp->p_messbuf=sendmcp(248,rpp->slot_num,2);
}

switching=0;  /* from here it's ok if interrupted */
rpc = *rpp;  /* copy 'proc' struct */

#if (CHIP == INTEL)
    rpc->p_ldt_sel = old_ldt_sel;
#endif

rpc->p_nr = m_ptr->PROC2;  /* this was obliterated by copy */

#if (SHADOWING == 0)
rpc->p_flags |= NO_MAP;  /* inhibit the process from running */
#endif

rpc->p_flags &=~(PENDING | SIG_PENDING | P_STOP);

/* Only 1 in group should have PENDING, child does not inherit trace status*/
sigemptyset(&rpc->p_pending);
rpc->p_pendcount = 0;
rpc->p_pid = m_ptr->PID;  /* install child's pid */
rpc->p_reg.retreg = 0;  /* child sees pid = 0 to know it is child */
rpc->user_time = 0;  /* set all the accounting times to 0 */
rpc->sys_time = 0;
rpc->child_utime = 0;
rpc->child_stime = 0;

#if (SHADOWING == 1)
rpc->p_nflips = 0;
}
mkshadow(rpp, (phys_clicks)m_ptr->ml_p1); /* run child first */
#endif

return(OK);
}

/* Copy the map from MM. */
src_phys = umap(proc_addr(caller), D, (vir_bytes)mapptr, sizeof(rp->p_map);
if (src_phys == 0)
panic("bad call to sys_newmap", NO_NUM);
phys_copy(src_phys, vir2phys(rp->p_map), (phys_bytes)sizeof(rp->p_map));

#if (SHADOWING == 0)
# if (CHIP != M68000)
alloc_segments(rp);
#else
pmmu_init_proc(rp);
#endif

old_flags = rp->p_flags; /* save the previous value of the flags */
rp->p_flags &= ~NO_MAP;
if (old_flags != 0 && rp->p_flags == 0)
{ rp->slot_num=-1; /* this one not resident yet */
* 304  switching=1;
* 305  ready(rp);
* 306  switching=0;
* 307}
308  #endif
309
310  return(OK);
311  }
312
313
/*======================================================================*
* do_exec                                                              *
*======================================================================*/

PRIVATE int do_exec(m_ptr)
register message *m_ptr;  /* pointer to request message */
{
  /* Handle sys_exec(). Perform code loading. */
  register struct proc *rp;
  phys_bytes dst_phys;
  int caller;  /* where the map has to be stored */
  int k;  /* process whose map is to be loaded */
  struct mem_map *map_ptr;  /* virtual address of map inside caller (MM) */

  /* Extract message parameters and copy new memory map to MM. */
  caller = m_ptr->m_source;
  k = m_ptr->PROC1;
  map_ptr = (struct mem_map *) m_ptr->MEM_PTR;
  if (!isokproc(k))
    panic("do_exec got bad proc: ", m_ptr->PROC1);
  rp = proc_addr(k);  /* ptr to entry of the map */

  /* Copy the map to MM. */
  dst_phys = umap(proc_addr(caller), D, (vir_bytes) map_ptr, sizeof(rp->p_map));
  if (dst_phys == 0) panic("bad call to sys_exec", NO_NUM);
  phys_copy(vir2phys(rp->p_map), dst_phys, sizeof(rp->p_map));

  return(OK);
}

/*======================================================================*
* do_getmap                                                             *
*======================================================================*/

PRIVATE int do_getmap(m_ptr)
message *m_ptr;  /* pointer to request message */
{
  /* Handle sys_getmap(). Report the memory map to MM. */

  /* Extract message parameters and copy new memory map to MM. */
  caller = m_ptr->m_source;
  k = m_ptr->PROC1;
  map_ptr = (struct mem_map *) m_ptr->MEM_PTR;
  if (!isokproc(k))
    panic("do_getmap got bad proc: ", m_ptr->PROC1);
  rp = proc_addr(k);  /* ptr to entry of the map */

  /* Copy the map to MM. */
  dst_phys = umap(proc_addr(caller), D, (vir_bytes) map_ptr, sizeof(rp->p_map));
  if (dst_phys == 0) panic("bad call to sys_getmap", NO_NUM);
  phys_copy(vir2phys(rp->p_map), dst_phys, sizeof(rp->p_map));

  return(OK);
}
/* Handle sys_exec(). A process has done a successful EXEC. Patch it up. */

register struct proc *rp;
reg_t sp;       /* new sp */
phys_bytes phys_name;
char *np;
#define NLEN (sizeof(rp->p_name)-1)

if (!isoksusern(m_ptr->PROC1))
    return E_BAD_PROC;
/* PROC2 field is used as flag to indicate process is being traced */
if (m_ptr->PROC2)
    cause_sig(m_ptr->PROC1, SIGTRAP);
sp = (reg_t) m_ptr->STACK_PTR;
rp = proc_addr(m_ptr->PROC1);
if (CHIP == M68000)
    rp->p_splow = sp; /* set the stack pointer low water */
#define FPP
/* Initialize fpp for this process */
fpp_new_state(rp);
#endif

rp->p_reg.pc = (reg_t) m_ptr->IP_PTR; /* set pc */
rp->p_reg.sp = sp; /* set the stack pointer */
rp->p_state=1; /* reset state as well */
rp->p_state=l; /* reset state as well */
rp->p_flags &=-RECEIVING; /* MM does not reply to EXEC call */
if (rp->slot_num>=0) /* slot already allocated, just reload it */
    
    {  
        sendmcp(253,rp->slot_num,rp); /* load state info */
        sendmcp(244,rp->slot_num,rp->p_nr); /* set SSR state */
    }

switching=0;

/* Save command name for debugging, ps(1) output, etc. */
phys_name = numap(m_ptr->m_source, (vir_bytes) m_ptr->NAME_PTR,
(vir_bytes) NLEN);
if (phys_name != 0) {
    phys_copy(phys_name, vir2phys(rp->p_name), (phys_bytes) NLEN);
    for (np = rp->p_name; (*np & BYTE) >=' '; np++) {}
    *np = 0;
}

return(OK);
175

/*======================================*
 * do_xit
 *======================================*/

PRIVATE int do_xit(m_ptr)

message *m_ptr;     /* pointer to request message */

{
    /* Handle sys_xit(). A process has exited. */

    register struct proc *rp, *rc;
    struct proc *np, *xp;

    int parent;     /* number of exiting proc's parent */
    int proc_nr;    /* number of process doing the exit */

    phys_clicks base, size;

    parent = m_ptr->PROC1;    /* slot number of parent process */
    proc_nr = m_ptr->PROC2;   /* slot number of exiting process */

    if (! isoksusern(parent) || !isoksusern(proc_nr))
        return(E_BAD_PROC);

    rp = proc_addr(parent);
    rc = proc_addr(proc_nr);

    switching=1;     /* is messing with ready Q, so denote */

    rp->child_utime += rc->user_time + rc->child_utime;  /* accum child times */
    rp->child_stime += rc->sys_time + rc->child_stime;   /* turn off alarm timer */

    unready(rc);     /* was only on flags==0, not so in MCP mode */

    #if (SHADOWING == 1)
    rmshadow(rc, &base, &size);
    m_ptr->ml_i1 = (int)base;
    m_ptr->ml_i2 = (int)size;
    #endif

    strcpy(rc->p_name, "<noname>");     /* process no longer has a name */

    if (rc->p_flags & PENDING)
    { --sig_procs;
        sigemptyset(&rc->p_pending);
        rc->p_pendcount = 0;
        rc->p_flags = P_SLOT_FREE;

        mcpi[rc->slot_num]=99;
        sendmcp(254,rc->slot_num,rc->slot_num);  /* release slot in cpu */
        slots++;
        rc->slot_num=-1;     /* no longer resident */
switching=0;
return(OK);
}

/*=====================================================*
 * do_getsp
 *=========================================================*
 PRIVATE int do_getsp(m_ptr)
 register message *m_ptr;    /* pointer to request message */
 {
 /* Handle sys_getsp(). MM wants to know what sp is. */
 register struct proc *rp;

 if (!isoksuser(m_ptr->PROC)) return(E_BAD_PROC);
 rp = proc_addr(m_ptr->PROC1);
 m_ptr->STACK_PTR = (char *) rp->p_reg.sp; /* return sp here (bad type) */
 return(OK);
 }

/*=====================================================*
 * do_times
 *=========================================================
 PRIVATE int do_times(m_ptr)
 register message *m_ptr;    /* pointer to request message */
 {
 /* Handle sys_times(). Retrieve the accounting information. */
 register struct proc *rp;

 if (!isoksuser(m_ptr->PROC)) return E_BAD_PROC;
 rp = proc_addr(m_ptr->PROC1);

 /* Insert the times needed by the TIMES system call in the message. */
 /* halt the volatile time counters in rp */
 m_ptr->USER_TIME  = rp->user_time;
 m_ptr->SYSTEM_TIME = rp->sys_time;
 unlock();
 m_ptr->CHILD_UTIME = rp->child_utime;
 m_ptr->CHILD_STIME = rp->child_stime;
 m_ptr->BOOT_TICKS = get_uptime();
 return(OK);
 }
/*============================================================* 
/* do_abort */
/*=====================================================*/

PRIVATE int do_abort(m_ptr)
message *m_ptr; /* pointer to request message */
{
/* Handle sys_abort. MINIX is unable to continue. Terminate operation. */
char monitor_code[64];
phys_bytes src_phys;

if (m_ptr->m1_i1 == RBT_MONITOR) {
    /* The monitor is to run user specified instructions. */
    src_phys = numap(m_ptr->m_source, (vir_bytes) m_ptr->m1_p1,
        (vir_bytes) sizeof(monitor_code));
    if (src_phys == 0) panic("bad monitor code from", m_ptr->m_source);
    phys_copy(src_phys, vir2phys(monitor_code),
        (phys_bytes) sizeof(monitor_code));
    reboot_code = vir2phys(monitor_code);
}
wreboot(m_ptr->m1_i1);
return(OK); /* pro-forma (really EDISASTER) */
}

/* pro-forma (really EDISASTER) */

/*==================================*/
/* do_sendsig */
/*==================================*/

PRIVATE int do_sendsig(m_ptr)
message *m_ptr; /* pointer to request message */
{
/* Handle sys_sendsig, POSIX-style signal */

struct sigframe {

_PROTOTYPE( void (*sf_retadr), (void) );
int sf_signo;
int sf_code;
struct sigcontext *sf_scp;
#if ENABLE_COMPAT
struct sigcontext *sf_oldscp;
#else
reg_t sf_fp;
#endif
_PROTOTYPE( void (*sf_retadr2), (void) );
struct sigcontext *sf_scpcopy;
#endif

};

/* Handle sys_sendsig, POSIX-style signal */
struct sigmsg smsg;

register struct proc *rp;
phys_bytes src_phys, dst_phys;

struct sigcontext sc, *scp;
struct sigframe fr, *frp;

if (!isokusern(m_ptr->PROC1))
    return(E_BAD_PROC);

rp = proc_addr(m_ptr->PROC1);
/* atomic operation */
sendmcp(242,11,rp->p_nr);
if (rp->slot_num>=0)
  {
    sendmcp(252,rp->slot_num,rp);
    rp->p_state=sendmcp(248,rp->slot_num,0);
    rp->p_getfrom=sendmcp(248,rp->slot_num,1);
    rp->p_messbuf=sendmcp(248,rp->slot_num,2);
  }
switching=0;

/* Get the sigmsg structure into our address space. */
src_phys = umap(proc_addr(MM_PROC_NR), D, (vir_bytes) m_ptr->SIG_CTX_PTR,
(vir_bytes) sizeof(struct sigmsg));
if (src_phys == 0)
    panic("do_sendsig can't signal: bad sigmsg address from MM", NO_NUM);

phys_copy(src_phys, vir2phys(&smsg), (phys_bytes) sizeof(struct sigmsg));

/* Compute theusr stack pointer value where sigcontext will be stored. */
scp = (struct sigcontext *) smsg.sm_stkptr - 1;

/* Copy the registers to the sigcontext structure. */
memcpy(&sc.sc_regs, &rp->p_reg, sizeof(struct sigregs));

/* Finish the sigcontext initialization. */
sc.sc_flags = SC_SIGCONTEXT;
sc.sc_mask = smsg.sm_mask;

dst_phys = umap(rp, D, (vir_bytes) scp,
(vir_bytes) sizeof(struct sigcontext));
if (dst_phys == 0) return(EFAULT);
phys_copy(vir2phys(&sc), dst_phys, (phys_bytes) sizeof(struct sigcontext));

/* Initialize the sigframe structure. */
frp = (struct sigframe *) scp - 1;
fr.sf_signo = smsg.sm_signo;
fr.sf_code = 0;  /* XXX - should be used for type of FP exception */
fr.sf scp = scp;
#if ENABLE_COMPAT
fr.sf oldscp copy = scp;
#else
fr.sf fp = rp->p_reg.fp;
#endif
fr.sf scp copy = scp;
fr.sf retadr = (void (*)(())) smsg.sm sigreturn;

/* Copy the sigframe structure to the user's stack. */
dst phys = umap(rp, D, (vir_bytes) frp, (vir_bytes) sizeof(struct sigframe));
if (dst phys == 0) return(EFAULT);
phys_copy(vir2phys(&fr), dst phys, (phys_bytes) sizeof(struct sigframe));

/* Reset user registers to execute the signal handler. */
rp->p_reg.sp = (reg_t) frp;
rp->p_reg.pc = (reg_t) smsg.sm_sighandler;
switching = 1;
if (rp->slot_num >= 0)
{
    sendmcp(253, rp->slot_num, rp);
    sendmcp(244, rp->slot_num, rp->p_nr);
    if (rp->p_state)
    {
        sendmcp(231, rp->slot_num, rp->p_getfrom);
        sendmcp(232, rp->slot_num, rp->p_messbuf);
        sendmcp(230, rp->slot_num, rp->p_state);
    }
    switching = 0;
}
return(OK);

do_sigreturn

PRIVATE int do_sigreturn(m_ptr)
{  
    /* POSIX style signals require sys_sigreturn to put things in order before the
    signalled process can resume execution */
    struct sigcontext sc;
    register struct proc *rp;
    phys_bytes src_phys;
if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);

rp = proc_addr(m_ptr->PROC1);

*switching=1;  /* must be atomic */
*if (rp->slot_num>=0)
*
* if (rp->slot_num)
*} {  sendmcp(252,rp->slot_num,rp);
* rp->p_state=sendmcp(248,rp->slot_num,0);
* rp->p_getfrom=sendmcp(248,rp->slot_num,1);
* rp->p_messbuf=sendmcp(248,rp->slot_num,2);
*
*switching=0;
*/ Copy in the sigcontext structure. */
src_phys = umap(rp, D, (vir_bytes) m_ptr->SIG_CTX_PTR,
 (vir_bytes) sizeof(struct sigcontext));
if (src_phys == 0) return(EFAULT);
phys_copy(src_phys, vir2phys(&sc), (phys_bytes) sizeof(struct sigcontext));
/* Make sure that this is not just a jmp_buf. */
if ((sc.sc_flags & SC_SIGCONTEXT) == 0) return(EINVAL);
/* Fix up only certain key registers if the compiler doesn't use
* register variables within functions containing setjmp.
*/
if (sc.sc_flags & SC_NOREGLOCALS) {
  rp->p_reg.retreg = sc.sc_retreg;
  rp->p_reg.fp = sc.sc_fp;
  rp->p_reg.pc = sc.sc_pc;
  rp->p_reg.sp = sc.sc_sp;
  return (OK);
}
sc.sc_psw = rp->p_reg.psw;

#if (CHIP == INTEL)
/* Don't panic kernel if user gave bad selectors. */
sc.sc_cs = rp->p_reg.cs;
sc.sc_ds = rp->p_reg.ds;
sc.sc_es = rp->p_reg.es;
#if _WORD_SIZE == 4
sc.sc_fs = rp->p_reg.fs;
sc.sc_gs = rp->p_reg.gs;
#endif
#endif
/* Restore the registers. */
memcpy(&rp->p_reg, (char *)&sc.sc_regs, sizeof(struct sigregs));

*switching=1;
if (rp->slot_num >= 0)
{
    sendmcp(253, rp->slot_num, rp);
    sendmcp(244, rp->slot_num, rp->p_nr);
    if (rp->p_state)
    {
        sendmcp(231, rp->slot_num, rp->p_getfrom);
        sendmcp(232, rp->slot_num, rp->p_messbuf);
        sendmcp(230, rp->slot_num, rp->p_state);
    }
    switching = 0;
}
return(OK);

/*==========================================================*/
/* do_kill                                                */
/*==========================================================*/
PRIVATE int do_kill(m_ptr)
    register message *m_ptr;    /* pointer to request message */
{
    if (!isokusern(m_ptr->PR))
        return(E_BAD_PROC);
    cause_sig(m_ptr->PR, m_ptr->SIGNUM);
    return(OK);
}

/*==========================================================*/
/* do_endsig                                               */
/*==========================================================*/
PRIVATE int do_endsig(m_ptr)
    register message *m_ptr;    /* pointer to request message */
{
    /* Finish up after a KSIG-type signal, caused by a SYS_KILL message or a call
       to cause_sig by a task */
    if (!isokusern(m_ptr->PROC))
        return(E_BAD_PROC);
    register struct proc *rp;
    if (!isokusern(m_ptr->PROC))
182

return(E_BAD_PROC);
* 806 /*
* 807  rp = proc_addr(m_ptr->PROC1);
* 808 */
* 809 /* MM has finished one KSIG. */
* 810 /* if (rp->p_pendcount != 0 && --rp->p_pendcount == 0
* 811   && (rp->p_flags & ~SIG_PENDING) == 0)
* 812    lock_ready(rp);
* 813 */
* 814 return(OK);
* 815 }
* 816

... /* more uninteresting routines */

1101
1102 /*  *---------------------------------------------------------------------*/
1103 *
1104  cause_sig
1105 /*  *---------------------------------------------------------------------*/
1106 PUBLIC void cause_sig(proc_nr, sig_nr)
1107 int proc_nr;    /* process to be signalled */
1108 int sig_nr;     /* signal to be sent, 1 to _NSIG */
1109 {
1110 /* A task wants to send a signal to a process. Examples of such tasks are:
1111 * TTY wanting to cause SIGINT upon getting a DEL
1112 * CLOCK wanting to cause SIGALRM when timer expires
1113 * FS also uses this to send a signal, via the SYS_KILL message.
1114 * Signals are handled by sending a message to MM. The tasks don't dare do
1115 * that directly, for fear of what would happen if MM were busy. Instead they
1116 * call cause_sig, which sets bits in p_pending, and then carefully checks to
1117 * see if MM is free. If so, a message is sent to it. If not, when it becomes
1118 * free, a message is sent. The process being signaled is blocked while MM
1119 * has not seen or finished with all signals for it. These signals are
1120 * counted in p_pendcount, and the SIG_PENDING flag is kept nonzero while
1121 * there are some. It is not sufficient to ready the process when MM is
1122 * informed, because MM can block waiting for FS to do a core dump.
1123 */
1124
1125  register struct proc *rp, *mmp;
1126  int f;
1127  rp = proc_addr(proc_nr);
1128  if (sigismember(&rp->p_pend, sig_nr))
1129     return;   /* this signal already pending */
1130  sigaddset(&rp->p_pend, sig_nr);
1131  ++rp->p_pendcount;  /* count new signal pending */
1132  if (rp->p_flags & PENDING)
1133     return;  /* another signal already pending */
if (rp->p_flags == 0) /* is this necessary? in MCP mode */
lock_unready(rp);

if (rp->p_flags |= PENDING | SIG_PENDIND;
++sig_procs; /* count new process pending */

mmp = proc_addr(MM_PROC_NR);
if (mmp->slot_num> 0)
{
    f = sendmcp(248, mmp->slot_num, 0);
mmp->p_getfrom = sendmcp(248, mmp->slot_num, 1);
}
else
    mmp->p_state;

mmp->p_flags &= ~(RECEIVING | SENDING);
if (f == 2)
mmp->p_flags |= RECEIVING;
else if (f == 3 || f == 5)
mmp->p_flags |= SENDING;

if ((mmp->p_flags & RECEIVING) == 0) || mmp->p_getfrom != ANY)
return;

inform();

PUBLIC void inform()
{
/* When a signal is detected by the kernel (e.g., DEL), or generated by a task
(e.g. clock task for SIGALRM), cause_sig() is called to set a bit in the
p_pending field of the process to signal. Then inform() is called to see
if MM is idle and can be told about it. Whenever MM blocks, a check is
made to see if 'sig_procs' is nonzero; if so, inform() is called.
*/

register struct proc *rp;

/* MM is waiting for new input. Find a process with pending signals. */
for (rp = BEG_SERV_ADDR; rp < END_PROC_ADDR; rp++)
if (rp->p_flags & PENDING)
{
sys_m.m_type = KSIG;
sys_m.SIG_PROC = proc_number(rp);
sys_m.SIG_MAP = rp->p_pending;
sig_procs--;  
sys_m.m_source=HARDWARE;
if (send(MM_PROC_NR, &sys_m) != OK)
    panic("can't inform MM", NO_NUM);
sigemptyset(&rp->p_pending); /* the ball is now in MM's court */
rp->p_flags &= ~PENDING;/* remains inhibited by SIG_PENDING */
return;
}

1.6 Process Scheduling Functions

Task scheduling routines that manipulate the process control block, proc, are grouped together in proc.c. For the MCA architecture, these were all completely rewritten so marking and using italics is pointless. Each of the routines, with the exception of interrupt() is discussed in detail in Chapter 5.

The various routines, which handle hardware interrupts, sometimes need to inform other kernel tasks of important events. For example, the clock interrupt routine informs the clock task when actions must be taken (e.g., alarms have expired, etc.). The function interrupt() is called in these situations so that a message can be sent to the appropriate task. Ordinarily, the kernel does not communicate with other tasks. When it does, it is also crucial that it is not blocked. Hence, the interrupt function must check first to see whether the message can be sent safely. It is only safe if the receiving task is resident and blocked awaiting IPC. If it is not safe, a flag is set in the proc entry for the task indicating a pending hardware interrupt. In the future, when the task is made resident, the pending interrupt message is sent with the unhold() function. Unhold is called by the CGS task when scheduling is performed.
First, the proc.c used for IPC directed scheduling is listed. The main difference between
IPC and round-robin scheduling is in the routine pick_proc(). The remaining functions are
unchanged.

```c
/* proc.c - IPC directed scheduling
 * This file contains essentially all of the process handling.
 * It has just one entry point from the outside:
 * interrupt: called by interrupt routines to send a message to task
 * It also has several minor entry points used by CGS and main:
 * ready: put process on the ready queue so it can be run
 * unready: remove a process from the ready queue
 * sched: a process has run too long; schedule another one
 * pick_proc: pick a process to run (used by system initialization)
 * unhold: process all held-up interrupts
 */

#include "kernel.h"
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"

PUBLIC unsigned char switching;     /* nonzero to inhibit interrupt() */
message hw_int;

PUBLIC void pick_slot(rp)
/* find location in cpu resident contexts to place this task.
 * if there is room, rp->slot_num is set to that, else it is
 * set to -1.
 */
struct proc *rp;                /* number of task to be run/made resident */
{                               
  if (slot_nr=rp->slot_num>0 || slots<1) /* already added, none avail. */
    return;
  for (rp->slot_num=NR_SLOTS; rp->slot_num; rp->slot_num--)
  { if (mpci[mpci->slot_num]=-99) /* available, so grab it */
  { mcpi[mpci->slot_num]=rp->p_nr; /* should use ptr for speed */
    slot_nr=rp->slot_num;        /* one to "run" next */
    sendmcp(253,rp->slot_num,rp); /* load context into cpu slot */
    /* register, and make active INT244 */
    sendmcp(244,rp->slot_num,rp->p_nr);
```
if (rp->p_state>=2) /* is this a re-load */
{
    sendmcp(231,rp->slot_num,rp->p_getfrom);
    if (rp->p_state>1)
        sendmcp(232,rp->slot_num,rp->p_messbuf);
    sendmcp(230,rp->slot_num,rp->p_state);
    rp->p_state=sendmcp(248,rp->slot_num,0);
}
slots--;
if (rp->p_int_blocked && rp->p_state==2) /* pending hdw int */
{
    send(rp->p_nr, &hw_int);
    rp->p_int_blocked=FALSE;
}
rp->p_state=1;
return; /* and allow normal load */
}

/*========================================================*/
/* interrupt */
/*========================================================*/
PUBLIC void interrupt(task)
int task; /* number of task to be started */
{
/* An interrupt has occurred. Schedule the task that handles it. */
register struct proc *rp,*pp; /* pointer to task's proc entry */

rp = proc_addr(task);

/* If this call would compete with other process-switching functions, put
* it on the 'held' queue to be flushed at the next non-competing restart().
* The competing conditions are:
* (1) k_reenter == (typeof k_reenter) -1:
* Call from the task level, typically from an output interrupt
* routine. An interrupt handler might reenter interrupt(). Rare,
* so not worth special treatment.
* (2) k_reenter > 0:
* Call from a nested interrupt handler. A previous interrupt handler
* might be inside interrupt() or sys_call().
* (3) switching != 0:
* Some process-switching function other than interrupt() is being
* called from the task level, typically sched() from CLOCK. An
* interrupt handler might call interrupt and pass the k_reenter test.
if (k_reenter != 0 || switching)
{
    lock();
    if (!rp->p_int_held)
    {
        rp->p_int_held = TRUE;
        if (held_head != NIL_PROC)
            held_tail->p_nextheld = rp;
        else
            held_head = rp;
        held_tail = rp;
        rp->p_nextheld = NIL PROC;
    }
    unlock();
    return;
}

if (rp->slot_num>=0)
{
    rp->p_state=sendmcp(248,rp->slot_num,0);
    rp->p_getfrom=sendmcp(248,rp->slot_num,1);
}

if (rp->slot_num<=0 || rp->p_state != 2)
{
    rp->p_int_blocked = TRUE;
    return;
}

/* Destination should be waiting for an interrupt.
 * Send it a message with source HARDWARE and type HARD_INT.
 * No more information can be reliably provided since interrupt messages
 * are not queued.
 */
send(rp->p_nr, &hw_int);
rp->p_int_blocked = FALSE;

/* sys_call - not used any longer */
PUBLIC int sys_call(function, src_dest, m_ptr)
{
    return 1;
}
struct proc *rp, *pp=NIL_PROC, *lp=NIL_PROC, *tp;
/* lp is proc to load; pp indicates TWO processes must be loaded */

PUBLIC int pick_proc()
{
/* Decide who to run now. A new process is selected by setting 'res_ptr'.
* When a fresh user (or idle) process is selected, record it in 'bill_ptr',
* so the clock task can tell who to bill for system time. Performs IPC directed
* scheduling by looking to satisfy pending IPC for resident contexts first. */

struct proc *rp, *pp=NIL_PROC, *lp=NIL_PROC, *tp;
/* lp is proc to load; pp indicates TWO processes must be loaded */

if (!slots)
    return 0; /* no room to load anything */

rp=rdy_head;
while ( rp != NIL_PROC)
{
    /* diagnostics, not needed any longer, I hope
    sendmcp(242,80,rp->p_nr);
    sendmcp(242,80,rp->p_state);
    sendmcp(242,84,rp->p_getfrom);
    */

    if(rp->p_state==1 && rp->slot_num<0)
    { /* a ready task. If all else fails, load this one */
        if (lp==NIL_PROC)
            lp=rp;
        else if (rp->p_nr<0) /* favor tasks over users */
            lp=rp;
        pp=NIL_PROC;
    }
    else if (rp->p_state==2 && !rp->p_int_blocked)
    { /* has blocked int, maybe other msg too, choose over ready task */
        lp=rp; /* only can happen to tasks, not users */
        pp=NIL_PROC;
    }
    else if (rp->p_state==2) /* now look for IPC blockages */
    { /* also IPC blocked, but on send, more hassle to do, so less favorable */
        tp=proc_addr(rp->p_getfrom);
    }
    else if (rp->p_state == 3 || rp->p_state == 5)
    { /* most important, so exit loop RIGHT NOW */
        pp=NIL_PROC;
        lp=rp;
        break; /* do resident ASAP */
    }
}

if(Islets)
    return 0; /* no room to load anything */
185 if (tp->slot_num>0) /* is destination already resident, good if so */
186 {
187 tp->p_state=sendmcp(248,tp->slot_num,0);
188 tp->p_getfrom=sendmcp(248,tp->slot_num,1);
189 }
190 if (tp->p_state==2 &&
191       (tp->p_getfrom==ANY || tp->p_getfrom==rp->p_nr))
192 {
193 if (lp==NIL_PROC)
194 {
195       lp=rp;
196       pp=tp;
197     }
198 else if (rp->p_nr<0) /* favor tasks over users */
199 {
200       lp=rp;
201       pp=tp;
202   }
203 if (tp->slot_num>0) /* always favor resident tasks */
204 {
205      /* if a task, stop RIGHT NOW */
206       lp=rp;
207       pp=NIL_PROC;
208       break; /* do right away */
209   }
210 }
211 }
212 rp=rp->p_nextready;
213 }
214 if (lp != NIL_PROC)
215 {
216   pick_slot(lp); /* make resident (should NOT be already) */
217 if (lp->slot_num>0) /* was able to add */
218 {
219     unready(lp);
220     if (pp==NIL_PROC || pp->slot_num>0)
221       return 1;
222 else
223       return 100+pp->p_nr; /* must load this next */
224 }
225 return 0;
226 }
227
228 /* No one is "ready". Run the idle task. The idle task really is
229  * responsible for finding some task(s) to make non-resident if
230  * there are waiting ready tasks (i.e., all resident tasks are
231  * blocked, awaiting something).
232  * I think proc_ptr should ALWAYS point to idle. I don't want
the system to accidentally restart a task in slot 0. That
would be disastrous.
*/
return 0;
}

/*===============================================================================================*/
/* ready*/
/*===============================================================================================*/
PUBLIC void ready(rp)
register struct proc *rp; /* this process is now runnable */
{
/* Add 'rp' to the end of queue of runnable processes.
* adds to front if a task, rear otherwise.
*/
if (rp->slot_num > 0) /* process resident already */
    return;
if (rdy_head == NIL_PROC)
    { 
        rdy_head = rdy_tail = rp; /* add to empty queue */
        rp->p_nextready=NIL_PROC;
    }
else if (istaskp(rp) || rp->p_state>1)
    { /* add to front of Q */
        rp->p_nextready = rdy_head;
        rdy_head=rp;
    }
else
    { 
        rdy_tail->p_nextready = rp;
        rdy_tail = rp;
        rp->p_nextready = NIL_PROC;
    }
if (slots & rp->p_flags==0)
pick_proc();
}

/*===============================================================================================*/
/* unready*/
/*===============================================================================================*/
PUBLIC void unready(rp)
register struct proc *rp; /* this process is no longer runnable */
{
/* A process has blocked. */

register struct proc *xp;

if (istaskp(rp)) /* task stack still ok? */
    if (*rp->p_stguard != STACK_GUARD)
        panic("stack overrun by task", proc_number(rp));

if ( (xp = rdy_head) == NIL_PROC)
    return;
if (xp == rp)
    { /* Remove head of queue */
        rdy_head = xp->p_nextready;
        return;
    }

/* Search body of queue. */
while (xp->p_nextready != rp)
    if ( (xp = xp->p_nextready) == NIL_PROC) return;
    xp->p_nextready = xp->p_nextready->p_nextready;
    if (rdy_tail == rp)
        rdy_tail = xp;

/*=========================================================*
* sched *
*==========================================================*/
PUBLIC int sched(rp)
struct proc *rp;
{
    /* the current process has run too long, or blocked (more likely).
    * Select a new task to run, then place this one on front (task or
    * blocked) of ready queue, or rear if user task & NOT blocked. Hence,
    * a compute bound user task is round-robined, while all others
    * are slotted to front of Q. This works since systems tasks are
    * higher priority, and blocked tasks of any kind won't be loaded
    * until their IPC can be completed anyway.
    */

    struct proc *xp;
    int r;

    r = pick_proc();
    if (rdy_head == NIL_PROC) /* ready done in-line for speed */
        { 
            rdy_head = rdy_tail = rp;
            rp->p_nextready = NIL_PROC;
        }
    else if (rp->p_state > 1 || istaskp(rp))
        { 
            
        }


if ( (held_head = rp->p_nexthead) == NIL_PROC)
    held_tail = NIL_PROC;
    rp->p_int_held = FALSE;
    unlock(); /* reduce latency; held queue may change! */
    interrupt(proc_number(r));
    lock(); /* protect the held queue again */
} while ( (rp = held_head) != NIL_PROC);

/*=========================================================*
* unhold *
*======================================================*/
PUBLIC void unhold()
{
    /* Flush any held-up interrupts. k_reenter must be 0. held_head must not
     * be NIL_PROC. Interrupts must be disabled. They will be enabled but will
     * be disabled when this returns.
     */
    register struct proc *rp;    /* current head of held queue */
    if (switching) return;
    rp = held_head;
    do {
        if ( (held_head = rp->p_nexthead) == NIL_PROC)
            held_tail = NIL_PROC;
            rp->p_int_held = FALSE;
            unlock(); /* reduce latency; held queue may change! */
            interrupt(proc_number(rp));
            lock(); /* protect the held queue again */
    } while ( (rp = held_head) != NIL_PROC);
I.7 Coarse-grain Scheduler Task

The CGS task is responsible for long term scheduling of tasks between memory and the processor itself. This is described in detail in Chapter 5. The CGS task decides which tasks should be kept resident in the MCA hardware slots available. This section contains the IPC directed scheduling version of the CGS. In the following section, round-robin scheduling is presented.

Activities of the CGS under IPC directed scheduling are broken into four parts. The first part consists of deciding whether or not a full scheduling cycle is required. This is contained in lines 43-44 and 148-149. If there are available hardware slots, or an entry into CGS was the result of a hardware interrupt (i.e., not all other contexts are blocked), then no full scheduling is required. If hardware slots are available, a call to pick_proc() is made to try to fill them with ready tasks; otherwise, the processor is simply released, lines 33-35, to the SSU as described in Chapter 4.

If no hardware slots are available and all resident tasks are blocked, then a full scheduling cycle is initiated, in lines 45-147. The second step attempts to find a non-resident task in the active list with a complementary IPC operation for the last running context, done in lines 45-78. If the scheduler reaches line 45, the last running task is blocked on an IPC operation. If there is a task in the active list which would allow the IPC operation to complete, that task is selected and made resident.

The third and fourth steps involve selecting a task to swap out of the processor, and
selecting a task from the active list, assuming one was not found in step 2, to make resident. Occasionally, it may be necessary to load two tasks from the active list. This happens if the task selected is blocked on an IPC operation with another task also on the active list.

First, a task is selected for removal. This is done in lines 79-103. The selected task is copied to the proc structure and the hardware slot is marked as empty. The mechanism for deciding which task to remove depends on the scheduling algorithm chosen. In this version, the task with the least IPC activity since the last scheduling run is chosen. Recently loaded contexts are weighted to avoid thrashing.

The last step is to load a new task. If a task has already been found in step 2, it is loaded immediately and the active list is not searched. This happens in lines 104-112. If a task is not already chosen, a call to sched (which calls pick_proc) is made and a task is loaded in lines 113-147. This section is complicated by the possibility of needing to load two tasks. Pick_proc returns the proc entry of the second task, if this is necessary, so that a second full scheduling operation is not required. However, the routine to select a task to swap out is duplicated.

/*CGS.c - IPC directed scheduling.
 * Minix "idle_task" In the MCA processor simulation this is, in fact,
 * the only task the kernel should select to run. It is the job of the
 * idle to task to monitor the MCA state: Either run other tasks, or
 * find a task to remove so another can be loaded in its place. I'm
 * not sure whether this quite fits as "idle" or not, but you'd rather
 * it didn't run much, and when it does, just long enough to find
 * another task to run in its place.
 */
#include "kernel.h"
#include <signal.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"

extern clock_t pending_ticks, realtime, next_alarm;
extern unsigned char switching;
void pick_slot (register struct proc *rp);

/*======================================================*
 * CGS task *
 *========================================================*/
PUBLIC void cgs_task(void)
{
    int slot, state, s1=0, m1, other, s2;
    struct proc *xp, *rp;

    sendmcp(238, t_lost_ticks, pending_ticks + realtime);
    /* Main loop of the idle_task. */
    while (TRUE)
    {
        do {
            /* give up processor to another slot */
            sendmcp(243, t_lost_ticks, pending_ticks + realtime);
        } while (rdy_head == NIL_PROC);

        /* Now, if it gets back here, then all other slots are */
        /* either empty or blocked. If empty, then look for another. */
        /* if all full AND blocked, then must decide to remove one. */
        /* Fortunately, if this is the case, this task should run */
        /* undisturbed until its finished. */
        switching = 1;
        state = sendmcp(239, NR_SLOTS - 1, 0); /* retrieve state of system */
        if (state && !slots) /* all blocked */
        {
            if (state > 0)
            {
                m1 = mcpi[state]; /* get proc # of... */
                xp = proc_addr(m1); /* last one to block */
                xp->p_state = sendmcp(248, state, 0);
                xp->p_getfrom = sendmcp(248, state, 1);
                if (xp->p_state == 2 && xp->p_getfrom == ANY)
                {
                    rp = rdy_head;
                    while (rp != NIL_PROC)
                    {
                        if (rp->p_state > 2 && rp->p_getfrom == xp->p_nr)
                            break;
                        rp = rp->next;
                    }
                }
            }
        }
    }
}
else
    rp=rp->p_nextready;
}
else if (xp->p_state==2)
{
    rp=proc_addr(xp->p_getfrom);
    if (rp->p_state <= 2 || rp->p_getfrom != xp->p_nr)
        rp=NIL_PROC;
}
else if (xp->p_state==3 || xp->p_state==5)
{
    rp=proc_addr(xp->p_getfrom);
    if (rp->p_state != 2 ||
        (rp->p_getfrom != ANY && rp->p_getfrom != xp->p_nr))
        rp=NIL_PROC;
}
else if (xp->p_state==4 || xp->p_state==5)
{
    rp=proc_addr(xp->p_getfrom);
    if (rp->p_state != 2 ||
        (rp->p_getfrom != ANY && rp->p_getfrom != xp->p_nr))
        rp=NIL_PROC;
}
if (rp != NIL_PROC) /* This is one to load */
    s2=state;
else
    s2=0;

s1=slot=1; /* start at user end */
if (s2==1)
    s1=slot=2; /* DON'T swap s2 */
sendmcp(236,0,0); /* update use counts */
m1=sendmcp(235,s1,0); /* assume this is least used slot */
do {
    slot++;
    state=sendmcp(235,slot,0);
    if (state < m1 && slot != s2)
    {
        s1=slot;
        m1=state;
    }
} while (slot < NR_SLOTS-1);
/* save state here */
exl=proc_addr(mcp[sn]);
sendmcp(252,s1,ex);
ex->p_state=sendmcp(254,s1,sl); /* mark empty & get state */
ex->p_getfrom=sendmcp(248,s1,1);
ex->p_messbuf=sendmcp(248,s1,2);
mcpi[s1]=-99; /* in O/S table as well */
ex->slot_num=-1;
slots=1;
if (s2) /* swap specific task in */
{

pick_slot(rp);
if (rp->slot_num > 0)
unready(rp);
other=0;
ready(xp); /* re-queue this one */
}
else
other=sched(xp); /* & select new one */
if (other>1) /* must load TWO tasks */
{
s2=s1;
if (s2 != 1)
s1=slot=1; /* start at user end */
else
  s1=slot=2;
m1=sendmcp(235,s1,0); /* assume this is least used slot */
do {
  slot++;
  state=sendmcp(235,slot,0);
  if (state<m1 && slot != s2)
  {
    s1=slot;
    m1=state;
  }
} while (slot < NR_SLOTS-1);
/* save state here */
xp=proc_addr(mcp[s1]);
sendmcp(252,s1,xp);
xml->p_state=sendmcp(254,s1,1); /* mark empty & get state */
xp->p_getfrom=sendmcp(248,s1,1);
xp->p_messbuf=sendmcp(248,s1,2);
mcp[s1]=-99; /* in O/S table as well */
xml->slot_num=-1;
slots=1;
lp=proc_addr(other-100); /* one to load */
pick_slot(rp);
if (lp->slot_num>0)
unready(rp);
ready(xp);
}
}
else if (slots && state)
pick_proc(); /* now look for a new task */
switching=0;
} /* now go back & release processor again */
}
I.8 Round Robin Scheduling Variations

Implementation of different coarse-grain scheduling mechanisms involves changes in two places within the MC/OS system software. First, the CGS task must be modified to change the criteria for initiating a scheduling cycle, and the method for determining which task must be swapped out when a new task must be made resident. In addition, the function pick_proc() must be changed to match the selection policy when choosing which task(s) on the active list to make resident.

For round-robin scheduling, both the CGS task and pick_proc functions are simple and straightforward when compared to IPC directed scheduling. This is because task selection is simplified.

The CGS task requires the greatest number (in terms of lines) of changes to convert from IPC directed to round-robin scheduling. For round-robin scheduling, the second step from the IPC directed CGS task is eliminated. Thus, if the CGS task is entered and all tasks are blocked with no empty slots, a full scheduling cycle is always initiated. Specifically, no attempt is made to identify a task to complement the last running resident task. Instead, the CGS task proceeds as follows: 1) Determine whether a full scheduling cycle is required, lines 39-43; if not, proceed as in the IPC directed case, lines 72-83; if so, 2) find a task to swap out, lines 44-66; 3) call sched (which calls pick_proc) to find a task to load, load the selected task, and release the processor to the SSU (hardware scheduler), in lines 67-72.
Minix "idle_task" In the MC processor simulation this is, in fact,
the only task the kernel should select to run. It is the job of the
idle to task to monitor the MC state: Either run other tasks, or
find a task to remove so another can be loaded in its place. I'm
not sure whether this quite fits as "idle" or not, but you'd rather
it didn't run much, and when it does, just long enough to find
another task to run in its place.

#include "kernel.h"
#include <signal.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"

extern clock_t pending_ticks, realtime, next_alarm;
extern unsigned char switching;

PUBLIC void idle_task(void)
{
    int slot, state, sl=0, ml, times=0, fails=0;
    struct proc *xp;
    /* Main loop of the idle_task. */
    while (TRUE)
    {
        do {
            sendmcp(243, 0, 0); /* give up the processor to another slot */
        } while (rdy_head==NIL_PROC);

        /* Now, if it gets back here, then all other slots are */
        /* either empty or blocked. If empty, then look for another. */
        /* if all full AND blocked, then must decide to remove one. */
        /* Fortunately, if this is the case, this task should run */
        /* undisturbed until its finished. */
        switching=1;
        state=2; /* assume all blocked, if empty break loop */
        for(slot=NR_SLOTS-1; slot && state>1; slot--)
            state=sendmcp(248, slot, 0); /* retrieve state of SLOT */
        if (state != 1) /* all blocked, or atleast 1 empty */
        {
            if (state >= 2) /* blocked, remove one */
            {
                sl=slot=1; /* start at user end */
                sendmcp(236, 0, 0); /* update use counts */
            } else
m1=sendmcp(235,1,0); /* assume this is least used slot */
do {
    slot++;
    state=sendmcp(235,slot,0);
    if (state<m1)
    {
        s1=slot;
        m1=state;
    }
} while (slot < NR_SLOTS-1);
/* save state here */
xp=proc_addr(mcpi[s1]);
sendmcp(252,s1,xp);
xp->p_state=sendmcp(254,s1,1); /* mark empty & get state */
xp->p_getfrom=sendmcp(248,s1,1);
xp->p_messbuf=sendmcp(248,s1,2);
mcri[s1]=-99; /* in O/S table as well */
slots=1;
sched(xp); /* & select new one */
/* if (times && mcpi[s1]>3)
    show_state(76);
*/
}
else
    pick_proc(); /* now look for a new task */
}

switching=0;
/* diagnostic; no longer used
if (fails>6000)
{
    fails=0;
    show_state(77);
}
/*
*/
/* now go back & release processor again */

Pick_proc searches the active list looking for the first task which is ready or for which an IPC operation can be completed. Note that the active list is always searched unlike the case for IPC directed scheduling. The search through the list is essentially the same. The highest priority task is found and returned. In the listing below, only the pick_proc routine is
shown. The remaining functions are identical to the IPC directed functions in proc.c

/* proc.c - round-robin scheduling policy
 * This file contains essentially all of the process and message handling.
 * It has ust one entry point from the outside:
 */

* interrupt: called by interrupt routines to send a message to task
* ready: put process on the ready queue so it can be run
* unready: remove a process from the ready queue
* sched: a process has run too long; schedule another one
* pick_proc: pick a process to run (used by system initialization)
* unhold: process all held-up interrupts
* */

... /* this section is identical to proc.c for IPC-directed scheduling */

/* pick Proc *
* =================:==================================*/
PUBLIC int pick_proc()
{
    /* Decide who to run now. A new process is selected by setting 'res_ptr'.
     * When a fresh user (or idle) process is selected, record it in 'bill_ptr',
     * so the clock task can tell who to bill for system time.
     */

    register struct proc *rp, *pp, *sp;    /* process to run, prev. in line */
    int Ok=0;

    if (!slots)
        return 0;       /* no room to load anything */
    rp=pp=rdy_head;
    while ( rp != NIL_PROC)
    {
        if (rp->p_state)    /* if blocked */
            {            /* check to see if can unblock */
                if (rp->p_state == 2)    /* blocked on rcv, check senders */
                    {    /* blocked on rev, check senders */
                        Ok=sendmcp(233,rp->p_getfrom,rp->p_nr);
                        if (!Ok && rp->p_int_blocked)
                            Ok=1;    /* has blocked int, but no other msg */
                    }
                else
                    Ok=sendmcp(234,rp->p_nr,rp->p_getfrom);
            }
    }
}
 could do better, but this is a complex operation
    anyway. Simply ensure it works now, and talk about
    in dissertation.
 */

 }  

 if (!rp->p_state || Ok)
  
  pick_slot(rp);  /* make resident (should NOT be already) */
  if (rp->slot_num>0)  /* was able to add */
    if (rp == pp)
      rdy_head=rp->p_nextready;
    else
      { 
        pp->p_nextready=rp->p_nextready;
        if (rp == rdy_tail)
          rdy_tail=pp;
      }
  return 1;
 }

 else
  
  

e else
  
  pp=rp;
  rp=rp->p_nextready;
 }

 }  

 /* nothing directly loadable, check for a send/recv pair both
    of whom are blocked. If found, load the receiver first.
 */

 rp=pp=rdy_head;
 while ( rp != NIL_PROC)
  
  if (rp->p_state == 2)  /* if blocked */
    /* check to see if can unblock */
    sp=rdy_head;
    while (sp != NIL_PROC && !Ok)
      if (sp->p_state>2 && sp->p_getfrom==rp->p_nr)
        Ok=1;
    else
      sp=sp->p_nextready;
  }

 if (!rp->p_state || Ok)
  
  pick_slot(rp);  /* make resident (should NOT be already) */
  if (rp->slot_num>0)  /* was able to add */
    if (rp == pp)
      rdy_head=rp->p_nextready;
    else
{ pp->p_nextready=rp->p_nextready; 
if (rp == rdy_tail) 
    rdy_tail=pp; 
} 
return 1;

} 
else 
{ 
  pp=rp;
  rp=rp->p_nextready;
}

} /* No one is "ready". Run the idle task. The idle task really is 
* responsible for finding some task(s) to make non-resident if 
* there are waiting ready tasks (i.e., all resident tasks are 
* blocked, awaiting something). 
* I think proc_ptr should ALWAYS point to idle. I don't want 
* the system to accidentally restart a task in slot 0. That 
* would be disastrous. 
*/

bill_ptr = proc_ptr = idle_ptr;
return 0;
}

... /* remainder of proc.c identical to proc.c for IPC directed scheduling. */

I.9 Unmodified Minix Scheduling and IPC

The original Minix 1.7.2 process scheduling and IPC routines, all contained in proc.c, are 
given below. This represents the base system, or single context system, used in our 
experiments to determine the increase in processor utilization and speedup of the MCA 
processor versus a conventional single context processor without hardware support for 
scheduling and IPC. A description of these routines is presented in [45], and a detailed 
timing analysis is given in [44].
sys_call: called when a process or task does SEND, RECEIVE or SENDREC
interrupt: called by interrupt routines to send a message to task

It also has several minor entry points:

lock_ready: put a process on one of the ready queues so it can be run
lock_unready: remove a process from the ready queues
lock_sched: a process has run too long; schedule another one
lock_mini_send: send a message (used by interrupt signals, etc.)
lock_pick_proc: pick a process to run (used by system initialization)
unhold: repeat all held-up interrupts

#include "kernel.h"
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"

PRIVATE unsigned char switching; /* nonzero to inhibit interrupt() */

FORWARD _PROTOTYPE( int mini_send, (struct proc *caller_ptr, int dest,
message *m_ptr) );
FORWARD _PROTOTYPE( int mini_rec, (struct proc *caller_ptr, int src,
message *m_ptr) );
FORWARD _PROTOTYPE( void ready, (struct proc *rp) );
FORWARD _PROTOTYPE( void sched, (void) );
FORWARD _PROTOTYPE( void unready, (struct proc *rp);)
FORWARD _PROTOTYPE( void pick_proc, (void) );

#if (CHIP == M68000)
FORWARD _PROTOTYPE( void cp_mess, (int src, struct proc *src_p, message *src_m,
struct proc *dst_p, message *dst_m) );
#endif

#if (CHIP == INTEL)
#define CopyMess(s,sp,sm,dp,dm) 
        cp_mess(s,(sp)->p_map[0].mem_phys,(vir_bytes)sm,(dp)->p_map[0].mem_phys, 
(vir_bytes)dm)
#endif

#if (CHIP == M68000)
#define CopyMess(s,sp,sm,dp,dm) 
        cp_mess(s,sp,sm,dp,dm)
#endif

/*========================================================*
 * interrupt *
 *======================================================*/
PUBLIC void interrupt(task)  
int task;            /* number of task to be started */  
{  
/* An interrupt has occurred. Schedule the task that handles it. */  

register struct proc *rp;    /* pointer to task's proc entry */  

rp = proc_addr(task);  

/* If this call would compete with other process-switching functions, put  
* it on the 'held' queue to be flushed at the next non-competing restart().  
* The competing conditions are:  
* (1) k_reenter == (typeof k_reenter) -1:  
*     Call from the task level, typically from an output interrupt  
*     routine. An interrupt handler might reenter interrupt(). Rare,  
*     so not worth special treatment.  
* (2) k_reenter > 0:  
*     Call from a nested interrupt handler. A previous interrupt handler  
*     might be inside interrupt() or sys_call().  
* (3) switching != 0:  
*     Some process-switching function other than interrupt() is being  
*     called from the task level, typically sched() from CLOCK. An  
*     interrupt handler might call interrupt and pass the k_reenter test.  
*/  
if (k_reenter != 0 || switching) {  
    lock();  
    if (!rp->p_int_held) {  
        rp->p_int_held = TRUE;  
        if (held_head != NIL_PROC)  
            held_tail->p_nextheld = rp;  
        else  
            held_head = rp;  
        held_tail = rp;  
        rp->p_nextheld = NIL_PROC;  
    }  
    unlock();  
    return;  
}  

/* If task is not waiting for an interrupt, record the blockage. */  
if ((rp->p_flags & (RECEIVING | SENDING)) != RECEIVING ||  
    !isrxhardware(rp->p_getfrom)) {  
    rp->p_int_blocked = TRUE;  
    return;  
}  

/* Destination is waiting for an interrupt.  
* Send it a message with source HARDWARE and type HARD_INT.  

No more information can be reliably provided since interrupt messages are not queued.

```c
rp->p_messbuf->m_source = HARDWARE;
rp->p_messbuf->m_type = HARD_INT;
rp->p_flags &~ RECEIVING;
rp->p_int_blocked = FALSE;
```

/* Make rp ready and run it unless a task is already running. This is ready(rp) in-line for speed. */

```c
if (rdy_head[TASK_Q] != NIL_PROC)
    rdy_tail[TASK_Q]->p_nextready = rp;
else
    proc_ptr = rdy_head[TASK_Q] = rp;
```

```c
PUBLIC int sys_call(function, src_dest, m_ptr)
```

/* The only system calls that exist in MINIX are sending and receiving messages. These are done by trapping to the kernel with an INT instruction. */

```c
register struct proc *rp;
```

/* Check for bad system call parameters. */

```c
if (!isoksrc_dest(src_dest)) return(E_BAD_SRC);
rp = proc_ptr;
```

/* if (isuserp(rp) && function != BOTH) return(E_NO_PERM); */

```c
PUBLIC int sys_call(function, src_dest, m_ptr)
```

```c
int function; /* SEND, RECEIVE, or BOTH */
int src_dest; /* source to receive from or dest to send to */
message *m_ptr; /* pointer to message */
```

```c
{ /* The parameters are ok. Do the call. */
```

```c
if (function & SEND) {
    /* Function = SEND or BOTH. */
    n = mini_send(rp, src_dest, m_ptr);
    if (function == SEND || n != OK)
```

```c
    /* Function = SEND or BOTH. */
    n = mini_send(rp, src_dest, m_ptr);
    if (function == SEND || n != OK)
```

```c
```

```c
/* The only system calls that exist in MINIX are sending and receiving messages. These are done by trapping to the kernel with an INT instruction. */
```

```c
/* The trap is caught and sys_call() is called to send or receive a message (or both). The caller is always given by proc_ptr. */
```

```c
PUBLIC int sys_call(function, src_dest, m_ptr)
```
return(n);  /* done, or SEND failed */
}
/* Function = RECEIVE or BOTH.
 * We have checked user calls are BOTH, and trust 'function' otherwise.
 */
return(mini_rec(rp, src_dest, m_ptr));

/* mini_send

PRIVATE int mini_send(caller_ptr, dest, m_ptr)

register struct proc *caller_ptr; /* who is trying to send a message? */
int dest;  /* to whom is message being sent? */
message *m_ptr;  /* pointer to message buffer */
{
    /* Send a message from 'caller_ptr' to 'dest'. If 'dest' is blocked waiting
    * for this message, copy the message to it and unblock 'dest'. If 'dest' is
    * not waiting at all, or is waiting for another source, queue 'caller_ptr'.
    */
    register struct proc *dest_ptr, *next_ptr;
    vir_bytes vb;  /* message buffer pointer as vir_bytes */
    vir_clicks vlo, vhi;  /* virtual clicks containing message to send */
    /* User processes are only allowed to send to FS and MM. Check for this. */
    /* if (isuser(caller_ptr) & & !issysent(dest)) return(E_BAD_DEST); */
    dest_ptr = proc_addr(dest);  /* pointer to destination's proc entry */
    if (dest_ptr->p_flags & P_SLOT_FREE) return(E_BAD_DEST);  /* dead dest */
    #if ALLOW_GAP_MESSAGES
    /* This check allows a message to be anywhere in data or stack or gap.
    * It will have to be made more elaborate later for machines which
    * don't have the gap mapped.
    */
    vb = (vir_bytes) m_ptr;
    vlo = vb >> CLICK_SHIFT;  /* vir click for bottom of message */
    vhi = (vb + MESS_SIZE - 1) >> CLICK_SHIFT;  /* vir click for top of msg */
    if (vlo < caller_ptr->p_map[D].mem_vir || vlo > vhi ||
        vhi >= caller_ptr->p_map[S].mem_vir + caller_ptr->p_map[S].mem_len)
        return(EFAULT);
    else
        /* Check for messages wrapping around top of memory or outside data seg. */
        vb = (vir_bytes) m_ptr;
        vlo = vb >> CLICK_SHIFT;  /* vir click for bottom of message */
        vhi = (vb + MESS_SIZE - 1) >> CLICK_SHIFT;  /* vir click for top of msg */
if (vhi < vlo ||
    vhi - caller_ptr->p_map[D].mem_vir >= caller_ptr->p_map[D].mem_len)
return(EFAULT);
#endif
/* Check for deadlock by 'caller_ptr' and 'dest' sending to each other. */
if (dest_ptr->p_flags & SENDING) {
    next_ptr = proc_add(dest_ptr->p_sendto);
    while (TRUE) {
        if (next_ptr == caller_ptr) return(ELOCKED);
        if (next_ptr->p_flags & SENDING)
            next_ptr = proc_addr(next_ptr->p_sendto);
        else
            break;
    }
}
/* Check to see if 'dest' is blocked waiting for this message. */
if ( (dest_ptr->p_flags & (RECEIVING | SENDING)) == RECEIVING &&
    (dest_ptr->p_getfrom == ANY ||
    dest_ptr->p_getfrom == proc_number(caller_ptr))) {
    /* Destination is indeed waiting for this message. */
    CopyMess(proc_number(caller_ptr), caller_ptr, m_ptr, dest_ptr,
    dest_ptr->p_msgbuf);
    dest_ptr->p_flags = ~RECEIVING; /* deblock destination */
    if (dest_ptr->p_flags == 0) ready(dest_ptr);
} else {
    /* Destination is not waiting. Block and queue caller. */
    caller_ptr->p_msgbuf = m_ptr;
    if (caller_ptr->p_flags == 0) unready(caller_ptr);
    caller_ptr->p_flags = SENDING;
    caller_ptr->p_sendto = dest;
    /* Process is now blocked. Put in on the destination's queue. */
    if ( (next_ptr = dest_ptr->p_callerq) == NIL_PROC)
        dest_ptr->p_callerq = caller_ptr;
    else {
        while (next_ptr->p_sendlink != NIL_PROC)
            next_ptr = next_ptr->p_sendlink;
        next_ptr->p_sendlink = caller_ptr;
    }
    caller_ptr->p_sendlink = NIL_PROC;
}
return(OK);
PRIVATE int mini_rec(caller_ptr, src, m_ptr)

register struct proc *caller_ptr; /* process trying to get message */

int src; /* which message source is wanted (or ANY) */

message *m_ptr; /* pointer to message buffer */

{
    /* A process or task wants to get a message. If one is already queued,
     * acquire it and deblock the sender. If no message from the desired source
     * is available, block the caller. No need to check parameters for validity.
     * Users calls are always sendrec(), and mini_send() has checked already.
     * Calls from the tasks, MM, and FS are trusted.
     */

    register struct proc *sender_ptr;
    register struct proc *previous_ptr;

    /* Check to see if a message from desired source is already available. */
    if (!((caller_ptr->p_flags & SENDING)) {
        /* Check caller queue. */
        for (sender_ptr = caller_ptr->p_callerq; sender_ptr != NIL_PROC;
            previous_ptr = sender_ptr, sender_ptr = sender_ptr->p_sendlink) {
            if (src == ANY || src == proc_number(sender_ptr)) {
                /* An acceptable message has been found. */
                CopyMess(proc_number(sender_ptr), sender_ptr,
                    sender_ptr->p_messbuf, caller_ptr, m_ptr);
                if (sender_ptr == caller_ptr->p_callerq)
                    previous_ptr->p_sendlink = sender_ptr->p_sendlink;
                else
                    previous_ptr->p_sendlink = sender_ptr->p_sendlink;
                if ((sender_ptr->p_flags & ~SENDING) == 0)
                    ready(sender_ptr); /* deblock sender */
            }
            return(OK);
        }
        /* Check for blocked interrupt. */
        if (caller_ptr->p_int_blocked && isrxhardware(src)) {
            m_ptr->m_source = HARDWARE;
            m_ptr->m_type = HARD_INT;
            caller_ptr->p_int_blocked = FALSE;
            return(OK);
        }
        /* No suitable message is available. Block the process trying to receive. */
        caller_ptr->p_getfrom = src;
        caller_ptr->p_messbuf = m_ptr;
    }
if (caller_ptr->p_flags == 0) unready(caller_ptr);
caller_ptr->p_flags |= RECEIVING;

/* If MM has just blocked and there are kernel signals pending, now is the
 * time to tell MM about them, since it will be able to accept the message.
 */
if (sig_procs > 0 && proc_number(caller_ptr) == MM_PROC_NR && src == ANY)
inform();
return(OK);

PRIVATE void ready(rp)
/* process to run */

if (rp = rdy_head[TASK_Q]) != NIL_PROC) {
    proc_ptr = rp;
    return;
}
if (rp = rdy_head[SERVER_Q]) != NIL_PROC) {
    proc_ptr = rp;
    return;
}
if (rp = rdy_head[USER_Q]) != NIL_PROC) {
    proc_ptr = rp;
    bill_ptr = proc_ptr;
    return;
}
/* No one is ready. Run the idle task. The idle task might be made an
 * always-ready user task to avoid this special case.
 */
bill_ptr = proc_ptr = proc_addr(IDLE);

PRIVATE void pick_proc()
/* Decide who to run now. A new process is selected by setting 'proc_ptr'.
 * When a fresh user (or idle) process is selected, record it in 'bill_ptr',
 * so the clock task can tell who to bill for system time.
 */

register struct proc *rp; /* process to run */

if (rp = rdy_head[TASK_Q]) != NIL_PROC) {
    proc_ptr = rp;
    return;
}
if (rp = rdy_head[SERVER_Q]) != NIL_PROC) {
    proc_ptr = rp;
    return;
}
if (rp = rdy_head[USER_Q]) != NIL_PROC) {
    proc_ptr = rp;
    bill_ptr = proc_ptr;
    return;
}
/* No one is ready. Run the idle task. The idle task might be made an
 * always-ready user task to avoid this special case.
 */
return(OK);
register struct proc *rp; /* this process is now runnable */
{
/* Add 'rp' to the end of one of the queues of runnable processes. Three
* queues are maintained:
* TASK_Q - (highest priority) for runnable tasks
* SERVER_Q - (middle priority) for MM and FS only
* USER_Q - (lowest priority) for user processes
*/

if (istaskp(rp)) {
    if (rdy_head[TASK_Q] != NIL_PROC)
        /* Add to tail of nonempty queue. */
        rdy_tail[TASK_Q]->p_nextready = rp;
    else {
        proc_ptr = /* run fresh task next */
        rdy_head[TASK_Q] = rp; /* add to empty queue */
    }
    rdy_tail[TASK_Q] = rp;
    rp->p_nextready = NIL_PROC; /* new entry has no successor */
    return;
}
if (!isuserptrp) { /* others are similar */
    if (rdy_head[SERVER_Q] != NIL_PROC)
        rdy_tail[SERVER_Q]->p_nextready = rp;
    else
        rdy_head[SERVER_Q] = rp;
    rdy_tail[SERVER_Q] = rp;
    rp->p_nextready = NIL_PROC;
    return;
}
#endif
/* put at front of Q
if (rdy_head[USER_Q] == NIL_PROC)
    rdy_tail[USER_Q] = rp;
    rp->p_nextready = rdy_head[USER_Q];
rdy_head[USER_Q] = rp;
*/
/* put at end of Q
if (rdy_head[USER_Q] != NIL_PROC)
    rdy_tail[USER_Q]->p_nextready = rp;
else
    rdy_head[USER_Q] = rp;
rdy_tail[USER_Q] = rp;
rp->p_nextready = NIL_PROC;
}

/*==================================*/
/* unready */
/*==================================*/
PRIVATE void unready(rp)
{
/* A process has blocked. */
    register struct proc *xp;
    register struct proc **qtail; /* TASK_Q, SERVER_Q, or USER_Q rdy_tail */
    if (istaskp(rp)) {
        /* task stack still ok? */
        if (*rp->p_stguard != STACK_GUARD)
            panic("stack overrun by task", proc_number(rp));
        if (xp = rdy_head[TASK_Q]) == NIL_PROC) return;
        if (xp == rp) {
            /* Remove head of queue */
            rdy_head[TASK_Q] = xp->p_nextready;
            if (rp == proc_ptr) pick_proc();
            return;
        }
        qtail = &rdy_tail[TASK_Q];
    } else if (!isuserp(rp)) {
        if ( (xp = rdy_head[SERVER_Q]) == NIL_PROC) return;
        if (xp == rp) {
            rdy_head[SERVER_Q] = xp->p_nextready;
            #if (CHIP == M68000)
            if (rp == proc_ptr)
                pick_proc();
            return;
            #endif
        }
        qtail = &rdy_tail[SERVER_Q];
    } else
    #if (SHADOWING == 1)
if (isshapow(rp)) {
    if ((xp = rdy_head[SHADOW_Q]) == NIL_PROC) return;
    if (xp == rp) {
        rdy_head[SHADOW_Q] = xp->p_nextready;
        if (rp == proc_ptr)
            pick_proc();
        return;
    }
    qtail = &rdy_tail[SHADOW_Q];
} else
#endif
{
    if ((xp = rdy_head[USER_Q]) == NIL_PROC) return;
    if (xp == rp) {
        rdy_head[USER_Q] = xp->p_nextready;
        #if (CHIP == M68000)
            if (rp == proc_ptr)
                pick_proc();
        #endif
        return;
    }
    qtail = &rdy_tail[USER_Q];
}

/* Search body of queue. A process can be made unready even if it is
not running by being sent a signal that kills it. */
while (xp->p_nextready != rp)
    if ((xp = xp->p_nextready) == NIL_PROC) return;
xp->p_nextready = xp->p_nextready->p_nextready;
if (*qtail == rp) *qtail = xp;
}

/*=======================================================*
 * sched *
 *========================================================*/
PRIVATE void sched()
{
    /* The current process has run too long. If another low priority (user)
    * process is runnable, put the current process on the end of the user queue,
    * possibly promoting another user to head of the queue.
    */
    if (rdy_head[USER_Q] == NIL_PROC) return;
    rdy_head[USER_Q]->slices++;
    /* One or more user processes queued. */
    rdy_tail[USER_Q]->p_nextready = rdy_head[USER_Q];
rdy_tail[USER_Q] = rdy_head[USER_Q];
rdy_head[USER_Q] = rdy_head[USER_Q]->p_nextready;
rdy_tail[USER_Q]->p_nextready = NIL_PROC;
pick_proc();
}

/*==============================================*
  *       lock_mini_send                       *
  *============================================*/
PUBLIC int lock_mini_send(caller_ptr, dest, m_ptr)
struct proc *caller_ptr; /* who is trying to send a message? */
int dest; /* to whom is message being sent? */
message *m_ptr; /* pointer to message buffer */
{
/* Safe gateway to mini_send() for tasks. */
int result;
switching = TRUE;
result = mini_send(caller_ptr, dest, m_ptr);
switching = FALSE;
return(result);
}

/*==============================================*
  *       lock_pick_proc                      *
  *============================================*/
PUBLIC void lock_pick_proc()
{
/* Safe gateway to pick_proc() for tasks. */
switching = TRUE;
pick_proc();
switching = FALSE;
}

/*==============================================*
  *       lock_ready                        *
  *============================================*/
PUBLIC void lock_ready(rp)
struct proc *rp; /* this process is now runnable */
{
/* Safe gateway to ready() for tasks. */
switching = TRUE;
ready(rp);
switching = FALSE;
}

switching = TRUE;
sched();
switching = FALSE;
}

switching = TRUE;
unready(rp);
switching = FALSE;
}

/* current head of held queue */
if (switching) return;

rp = held_head;
do {
    register struct proc *rp; /* current head of held queue */
    if (switching) return;
    rp = held_head;
    do {
        if ((held_head = rp->p_nextheld) == NIL_PROC) held_tail = NIL_PROC;
579     rp->p_int_held = FALSE;
580   unlock();  // * reduce latency; held queue may change! */
581   interrupt(proc_number(rp));
582   lock();   // * protect the held queue again */
583 }
584 while ( (rp = held_head) != NIL_PROC);
585 }
586
587 #if (CHIP == M68000)
588 /*==========================================================*/
589  *
590 * cp_mess
591 /*==========================================================*/
592 PRIVATE void cp_mess(src, src_p, src_m, dst_p, dst_m)
593 int src; /* sender process */
594 register struct proc *src_p; /* source proc entry */
595 message *src_m; /* source message */
596 register struct proc *dst_p; /* destination proc entry */
597 message *dst_m; /* destination buffer */
598 {
599 #if (SHADOWING == 0)
600 /* convert virtual address to physical address */
601 /* The caller has already checked if all addresses are within bounds */
602 src_m = (message *)((char *)src_m + ((phys_bytes)src_p->p_map[D].mem_phys
603 - src_p->p_map[D].mem_vir) << CLICK_SHIFT));
604 dst_m = (message *)((char *)dst_m + ((phys_bytes)dst_p->p_map[D].mem_phys
605 - dst_p->p_map[D].mem_vir) << CLICK_SHIFT));
606 #else
607 register phys_bytes correction;
608 if (correction = src_p->p_shadow) {
609     correction = (correction - src_p->p_map[D].mem_phys) << CLICK_SHIFT;
610     src_m = (message *)((char *)src_m + correction);
611 }
612 if (correction = dst_p->p_shadow) {
613     correction = (correction - dst_p->p_map[D].mem_phys) << CLICK_SHIFT;
614     dst_m = (message *)((char *)dst_m + correction);
615 }
616 #endif
617 #ifdefNEEDFSTRUCOPY
618 phys_copy(src_m,dst_m,(phys_bytes) sizeof(message));
619 #else
620  *dst_m = *src_m;
621 #endif
622 dst_m->m_source = src;
623 }
624 #endif
625
626 #endif
Simulation Source Code

The MCA processor simulation, described in Chapter 6, was performed using an 80x86 simulation package called "bochs". This software runs as an X window application under the UNIX operating system and provides a virtual 80x86 system environment. The package was modified to provide a virtual MCA system environment. This Appendix presents and discusses the modifications to the bochs package needed to implement the MCA processor simulation used in this study.

II.1 MCA Simulation with "bochs"

The MCA processor architecture was simulated within the bochs package by the addition of two additional modules and minor changes to the existing package in order to call the added modules when appropriate. This made it possible to isolate the changes needed to simulate the MCA processor which reduced development time.

The two additional modules and their relationship with the bochs simulator are shown in Figure II.1, below.
MCA instructions are implemented in the module `soft_int.c`. This file handles 80x86 software generated interrupts, and was extracted from the original bochs file, `exception.c`, which handles all forms of traps and interrupts in an 80x86 processor. The second module, `mcp_load.c`, consists of a single function, `mcp_load`, which is used to load a hardware context in the simulated MCA processor. It is isolated in a separate file because it contains the hardware dependent characteristics (e.g., on the peculiarities of the 80x86 architecture) of the base system. By having this kept separate, it allows for easy porting to other architectures by simply replacing the `mcp_load` function for the 80x86 architecture with one for a different processor.
Instruction simulation is illustrated in Figure II.2, below. This shows which functions are used to fetch and decode instructions, and how MCA instructions are simulated within the overall system. The modified portions of the original bochs software are included in the following sections, as well as the complete source code of the two additional modules which implement MCA instructions.

Figure II.2 MCA instruction simulation with bochs environment.

II.2 Bochs Module Modifications

Config.h contains system configuration constants which define the characteristics of the hardware being simulated. This includes the type of processor, word size, word order,
etc. This file was modified to contain the number of hardware contexts the MCA system would support. This was fixed at 32, an arbitrary but sufficiently high value. Recall from Appendix I that at system boot time the number of hardware contexts in use was specified by the boot loader. The number of contexts in each experiment was controlled in this way, allowing for experiments to be run for different configurations without recompiling the simulator.

```c
/* config.h. Generated automatically by configure. */
#define BX_CPU 2
#define MCP_MODE 32 /* added by tim k for multiple context sim. */
#define WORDS_BIGENDIAN 0
#define SIZEOFUnsigned_CHAR 1
#define SIZEOFUnsigned_SHORT 2
#define SIZEOFUnsigned_INT 4
#define SIZEOFUnsigned_LONG 4
#define SIZEOFUnsigned_LONG_LONG 8
#if SIZEOFUnsigned_CHAR != 1
#error "sizeof (unsigned char) != 1"
#else
typedef unsigned char Bit8u;
typedef signed char Bit8s;
#endif
#if SIZEOFUnsigned_SHORT != 2
#error "sizeof (unsigned short) != 2"
#else
typedef unsigned short Bit16u;
typedef signed short Bit16s;
#endif
#if SIZEOFUnsigned_INT == 4
typedef unsigned int Bit32u;
typedef signed int Bit32s;
#elif SIZEOFUnsigned_LONG == 4
typedef unsigned long Bit32u;
typedef signed long Bit32s;
#else
#error "can't find sizeof(type) of 4 bytes!"
#endif
```
Bochs.h contains declarations and prototypes for simulation variables and functions accessed among the various modules in the system. Bochs.h was modified to contain declarations for all global MCA simulation variables and values.

```c
/* bochs.h
 * system typedefs and extern declarations of globals.
 *
 * externs added for multiple context simulation by Tim Killeen.
 * See MCP_MODE.
 *
Copyright Notice
================
BOCHS is shareware for PERSONAL USE only.
For more information, read the file 'LICENSE' included in the bochs distribution. If you don't have access to this file, or have questions regarding the licensing policy, the author may be contacted via:

  US Mail: Kevin Lawton
          528 Lexington St.
          Waltham, MA 02154

  EMail: bochs@world.std.com
*/
#ifndef BX_BOCHS_H
#define BX_BOCHS_H
#include <stdarg.h>
#include <stdio.h>
#include "config.h"
#include "memory.h"

/* --- TYPEDEFS --- */
typedef struct {
  Bit8u (* read_funct)(Bit32u addr);
  void (* write_funct)(Bit32u addr, Bit8u value);
  char *handler_name;
  Bit32u start_addr;
  Bit32u end_addr;
  int irq;
}...*/endif /* _BX_CONFIG_H */
```
typedef struct {
    Bit8u (* read_funct) (Bit32u addr);  /* read handler */
    void (* write_funct) (Bit32u addr, Bit8u value); /* write handler */
    char * handler_name;  /* name of device */
    int irq;  /* associated irq if any */
} bx_io_handler_t;

typedef struct {
    Bit32u period;
    Bit32u remaining;
    Boolean active;
    Boolean continuous;
    void (* funct) (void);
} bx_timer_t;

void bx_register_io_handlers(void);
int bx_register_io_handler( bx_iodev_t);
void bx_register_int_vector(Bit8u interrupt, Bit8u* code, int code_size,
    void (* funct)(int vector));
int bx_register_timer(void (*funct)(void), Bit32u useconds,
    Boolean continuous, Boolean active);
void bx_activate_timer(int timer_index, Boolean activate, Bit32u useconds);
void bx_iodev_setup(void);
void bx_add_input(int fd, int event_type, void (* funct)(void));

/* see main.c for declaration and comments regarding next group */

#if MCP_MODE > 0
extern bx_cpu_t mcp[MCP_MODE];  // to store each context
extern int mcpi[MCP_MODE];      // to store whether context empty/used
extern int mcpw[MCP_MODE];      // contain WAR entries for tasks
extern int mcpp[MCP_MODE];      // contain WhoID of TNR
extern char mcpm[MCP_MODE][24]; // Stores messages (i.e., reg. windows)
extern Bit16u mcpl[MCP_MODE];   // store memory locations of messages
extern char * mcpp[MCP_MODE];   // where in memory to copy messages
extern int mcp_slot, mcp_prev;  // present, prior slot running
extern int mcpD;                // DEBUGGING flag
extern int mcpit[256],mcpin[56],mcpic[2],mcps[MCP_MODE],mcpr[MCP_MODE];
extern int mcpiw[64];           // above to here hold experiment results
#endif

Global variables are declared and initialized in main.c. This module was modified to perform this function on MCA simulation variables as well. The copyright and additional
/* main.c
 * Contains variable declarations and calls for system initialization.
 * actual cpu simulation happens in decode.c.
 * See copyright below. This file hacked a bit by Tim Killeen.
 * Modified for multiple context simulation. Look for MCP_MODE
 * bracketed code.

Copyright Notice
================

BOCHS is shareware for PERSONAL USE only.

For more information, read the file 'LICENSE' included in the bochs
distribution. If you don't have access to this file, or have questions
regarding the licensing policy, the author may be contacted via:

US Mail: Kevin Lawton
528 Lexington St.
Waltham, MA 02154

EMail: bochs@world.std.com
*/
#define BX_MAIN_C 1
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <ctype.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include "bochs.h"
#include "iodev/iodev.h"

void bx_sighup_handler (int sig);
void show_timers ();
void hga_update ();

/* prototypes */
static void bx_usage (char *progname);
void bx_init_cpu (void);
void bx_init_debug (void);
void bx_cpu_loop (void);

/* typedefs */

/* the device id and stepping id are loaded into DH & DL upon processor startup. for device id: 3 = 80386, 4 = 80486. just make up a number for the stepping (revision) id. */
#define BX_DEVICE_ID 2
#define BX_STEPPING_ID 0

bx_cpu_t bx_cpu; /* CPU state stored here */

#if MCP_MODE > 0
bx_cpu_t mcp[MCP_MODE]; /* resident copies of CPU state(s) */
int mcp[i[MCP_MODE]; /* store status of each slot */
char mcpp[MCP_MODE][24];/* message buffer for each resident task */
char *mcpp[MCP_MODE]; /* ptr to each slots msg buffer */
Bit16u mcp[i[MCP_MODE]; /* where on receive to put message (memory) */
int mcpw[MCP_MODE]; /* task id (not slot) to s/r msg */
int mcp[D[MCP_MODE]; /* O/S task id (not slot) */
int mcp_slot; /* which state is currently active */
int mcp_prev; /* last context active */
int mcpD; /* mcp debugging flag */
int mcpit[256], mcpin[56], mcpic[2]; /* for instruction profiling */
int mcps[MCP_MODE], mcpr[MCP_MODE]; /* for scheduling (TAR values) */
int mcpw[64];
#endif

... #if MCP_MODE > 0
void bx_init_mcp ()
/* initialize things for multiple resident context simulation */
{
  int i;
  for (i = 0; i < MCP_MODE; i++)
    mcpp[i] = mcpp[i];
  mcp_slot = 0; /* kernel slot */
  mcp[0] = 1; /* kernel ALWAYS must be ready */
}
#endif

main (int argc, char *argv[])
{
  int n;
  #if 0

void manual_tests();
#endif

#if 0
manual_tests();
#endif

/* read the .bochsrc file */
parse_bochsrc();

n = 2;
while (n <= argc)
{
    if (!strcmp (argv[n - 1], "-log"))
    {
        strcpy (logfilename, argv[n]);
        n += 2;
    }
    else if (!strcmp (argv[n - 1], "-sanity-check"))
    {
        bx_sanity_checks();
        exit (0);
        n += 1;
    }
    else if (!strcmp (argv[n - 1], "-bootA"))
    {
        strcpy (bx_options.bootdrive, "a");
        n += 1;
    }
    else if (!strcmp (argv[n - 1], "-bootC"))
    {
        strcpy (bx_options.bootdrive, "c");
        n += 1;
    }
    else
    {
        bx_usage (argv[0]);
    }
    if (!strcmp (logfilename, "."))
    {
        bx_logfd = stderr;
    }
    else
    {
        bx_logfd = fopen (logfilename, "w");
        if (!bx_logfd)
        {
            }
printf (stderr, "could not open log file '", logfilename);
exit (1);
}

bx_init_cpu ();
bx_init_memory (BX_PHY_MEM_SIZE);
bx_init_mcp ();
bx_init_debug ();
bx_iodev_setup ();

/* give bios POST to do minimal setup of the post routine */
bx_post_setup ();

/* start off by jumping to bios POST routine */
bx_imp_post ();

/* bx_start_timers() will be called by the POST code after
 * initializing iodev devices */
bx_cpu_loop ();  /* OK, start simulating instructions */
return (0);
}

Exception.c processes most interrupts. When simulating the MCA processor, however, software generated interrupts are processed in soft_int.c since these are really MCA instructions.

The exception handling code must save the currently executing hardware state information, mark the slot as INT blocked, reset the system to hardware slot 0, then process the interrupt. In simulation this requires copying the CPU structure bx_cpu into the MCA context array, mcp, store the running slot number in mcp_prev, mark the current slot as INT blocked in mcpi, and set the active slot, mcp_slot, to 0. Following this, hardware interrupts are processed normally by the bochs simulator and MC/OS interrupt handling software.

/* exception.c
 * The major function of this file consists of bx_interrupt which handles
the soft INT instructions. As is the Intel architecture, this is a
very nasty little routine.

Changes made in soft_int.c, by Tim Killeen, CAN effect these routines.
Basically, I've added code in bx_Int_Ib to process in "hardware" a
range of interrupts from FF *down*. These are special instructions for
multiple context simulation and *shouldn't* effect anyone.

in bx_interrupt I've added code to cause an automatic context switch
to task 0, the "kernel" by definition.

Copyright Notice

BOCHS is shareware for PERSONAL USE only.

For more information, read the file 'LICENSE' included in the bochs
distribution. If you don't have access to this file, or have questions
regarding the licensing policy, the author may be contacted via:

US Mail: Kevin Lawton
528 Lexington St.
Waltham, MA 02154

EMail: bochs@world.std.com

#include "bochs.h"

extern volatile Boolean bx_async_event;
extern Bit32u bx_prev_eip;

void bx_shutdown_cpu (void);

void bx_interrupt (Bit8u vector, Boolean is_INT, Boolean is_error_code)
{
  Bit16u cs, ip;

  #if MCP_MODE > 0
  mcp[mcp_slot] = bx_cpu;    /* save current context */
  if (mcp_slot)
    
    mcp_prev = mcp_slot;    /* and who was running */
    mcpif[mcp_slot] = 99;    /* this task was INT'd */
  
  printf("exc: %i\n",mcp_slot);
}
Mcp_load.c is used to load a hardware context with a task’s state information stored in the MC/OS process table, proc. This code is very dependent on the fact that simulation was performed using the Intel 80x86 architecture as the base system. As a result, the code is difficult to read and follow. The interested reader should refer to the Intel processor manual for clarification [38,39]. The mcp_load operation is essentially the same as an IRET 80x86 instruction, except that a software context is being placed into another hardware context slot without actually replacing the currently running context. Hence, the IRET instruction
cannot be used directly, and some parts must be disabled; namely, checking for and changing I/O and CPU protection levels.

In order to simplify the simulation and coding process, state information is first loaded into the bochs CPU structure, bx_cpu, then copied to the MCA state structure array, mcp[]. Thus, the code contained in mcp_load.c is very similar to that found elsewhere within bochs.

```c
/* mcp_load.c
* derived from ctrl_xfer.c for MCA simulation.
* handles transfer between tasks, segments. Things like IRET, etc.
* does what IRET does, but only for mcp context load, so location
* of descriptors is handled via direct address rather than offsets
* from ss (which is not valid anyway).
*
Copyright Notice
===============

BOCHS is shareware for PERSONAL USE only.

For more information, read the file 'LICENSE' included in the bochs distribution. If you don't have access to this file, or have questions regarding the licensing policy, the author may be contacted via:

US Mail: Kevin Lawton
528 Lexington St.
Waltham, MA 02154

EMail: bochs@world.std.com
*/

#include "bochs.h"

INLINE void mcp_IRET(Bit32u addr, Bit16u flags16)
/* almost identical to bx_IRET, except that this is for a context load,
* so there is no iret instruction. Hence, things will be found at
* different locations than the current cs & ss references. Data is
* found using pointer into PCB (which is where it is during normal
* iret as well, the difference is in the method of access.
*/
{
```
Bit16u ip, cs, flags;

#if BX_CPU >= 2
if (bx_protected_mode())
{
  Bit16u raw_cs_selector, raw_ss_selector;
  Bit16u new_ip, new_sp;
  bx_selector_t cs_selector, ss_selector;
  Bit32u dword1, dword2;
  bx_descriptor_t cs_descriptor, ss_descriptor;
  Bit8u prev_cpl;

  if (bx_dbg.protected) bx_printf(0, "iret()\n");

  if (bx_cpu.eflags.nt)
  { /* NT = 1: RETURN FROM NESTED TASK */
    bx_printf(1, "IRET: nested task return not implemented\n");
  }
  else /* NT = 0: INTERRUPT RETURN ON STACK */
  { /* FLAGS SP+4 */
    /* CS SP+2 */
    /* IP SP+0 */
    /* return CS selector RPL must be >= CPL, else #GP(return selector) */
    /* This was popped off already in soft_int.c, INT253 */
    raw_cs_selector=bx_cpu.cs.cache.u.segment.base
    /* selector must be non-null else #GP(0) */
    if ((raw_cs_selector & 0xffff) == 0 )
    { /* INTERRUPT RETURN TO SAME LEVEL */
      bx_printf(1,"mcp_iret: return CS selector null\n");
      bx_segment_exception(BX_GP_EXCEPTION, 0);
      return;
    }
    bx_parse_selector(raw_cs_selector, &cs_selector);
    if (cs_selector.rpl < CPL)
    { /* INTERRUPT RETURN TO SAME LEVEL */
      bx_printf(1,"mcp_iret: return selector RPL, < CPL\n");
      bx_segment_exception(BX_GP_EXCEPTION, raw_cs_selector);
      return;
    }
    /* printf("mcp_load: rpl %i, CPL %i\n",cs_selector.rpl, CPL); */
    if (cs_selector.rpl < /*==*/ CPL)
    { /* INTERRUPT RETURN TO SAME LEVEL */
      if (bx_dbg.protected)
bxprintf(0, "mcp_iret: return on stack, same level\n");

/* selector index must be within descriptor table limits, */
/* else #GP(return selector) */
bx_fetch_raw_descriptor(&cs_selector, &dword1, &dword2, BX_GP_EXCEPTION);

BX_HANDLE_EXCEPTION()

bx_parse_descriptor(dword1, dword2, &cs_descriptor);

/* AR byte must indicate code segment else #GP(return selector) */
if ( cs_descriptor.segment==0 ||
    cs_descriptor.u.segment.executable==0 )
{
    bxprintf(1, "mcp_iret0: AR byte indicated non code segment\n");
    bx_segment_exception(BX_GP_EXCEPTION, raw_cs_selector);
    return;
}

/* if non-conforming then code segment DPL must = CPL, */
/* else #GP(return selector) */
if ( cs_descriptor.u.segment.c_ed==0 &&
    cs_descriptor.dpl!=CPL)
{
    bxprintf(1, "mcp_iret: non-conforming, DPL != CPL\n");
    bx_segment_exception(BX_GP_EXCEPTION, raw_cs_selector);
    return;
}

/* if conforming, then code segment DPL must be <= CPL */
/* else #GP(return selector) */
if ( cs_descriptor.u.segment.c_ed &&
    cs_descriptor.dpl>CPL)
{
    bxprintf(1, "mcp_iret: conforming, DPL > CPL\n");
    bx_segment_exception(BX_GP_EXCEPTION, raw_cs_selector);
    return;
}

/* segment must be present else #NP(return selector) */
if ( cs_descriptor.p==0 )
{
    bxprintf(1, "mcp_iret: not present\n");
    bx_segment_exception(BX_NP_EXCEPTION, raw_cs_selector);
    return;
}

if ( bx_cpu.eip > cs_descriptor.u.segment.limit )
{
    bxprintf(1, "mcp_iret: IP > descriptor limit\n");
    bx_segment_exception(BX_GP_EXCEPTION, 0);
}
return;
}

/* load CS-cache with new code segment descriptor */
bx_load_cs(&cs_selector, &cs_descriptor, CPL);
bx_write_flags(flags16, CPL==0, CPL<=IOPL);
return; /* all done with return to same level */

else /* INTERRUPT RETURN TO OUTER PRIVILEGE LEVEL */
{
/* selector index must be within its descriptor table limits,
* else #GP(return selector)
*/

bx_fetch_raw_descriptor(&cs_selector, &dword1, &dword2,
BX_GP_EXCEPTION);
BX_HANDLE_EXCEPTION()
bx_parse_descriptor(dword1, dword2, &cs_descriptor);
/* AR byte must indicate code segment else #GP(return selector) */
if (cs_descriptor.segment==0 ||
    cs_descriptor.u.segment.executable==0 )
{
    bx_printf(1, "mcp_iretl: AR byte indicated non code segment\n");
    bx_segment_exception(BX_GP_EXCEPTION, raw_cs_selector);
    return;
}
/* if non-conforming then code segment DPL must = CS selector RPL,
* else #GP(return selector)
*/
if (cs_descriptor.u.segment.c_ed==0 &&
    cs_descriptor.dpl!=cs_selector.rpl)
{
    bx_printf(1, "mcp_iret: descriptor DPL != selector RPL\n");
    bx_segment_exception(BX_GP_EXCEPTION, raw_cs_selector);
    return;
}
/* if conforming then code segment DPL must be > CPL,
* else #GP(return selector)
*/
if (cs_descriptor.u.segment.c_ed &&
    cs_descriptor.dpl <= CPL )
{
    bx_printf(1, "mcp_iret: descriptor DPL <= CPL\n");
    bx_segment_exception(BX_GP_EXCEPTION, raw_cs_selector);
    return;
}
/* segment must be present, else #NP(return selector) */
if (cs_descriptor.p==0 )
{

bx_printf(1, "mcp_iret: segment not present\n");
bx_segment_exception(BX_NP_EXCEPTION, raw_cs_selector);
return;
}

/* examine return SS selector (at SP+8) and associated descriptor */
/* already popped off in soft_int.c, INT253 */
raw_ss_selector=bx_cpu.ss.cache.u.segment.base;

/* selector must be non-null, else #GP(0) */
if ( (raw_ss_selector & 0xfffc) == 0 )
{
    bx_printf(1, "mcp_iret: SS selector null\n");
bx_segment_exception(BX_GP_EXCEPTION, 0);
    return;
}

bx_parse_selector(raw_ss_selector, &ss_selector);

/* selector RPL must = RPL of return CS selector,
* else #GP(SS selector) */
if ( ss_selector.rpl != cs_selector.rpl)
{
    bx_printf(1, "mcp_iret: SS.rpl != CS.rpl\n");
bx_segment_exception(BX_GP_EXCEPTION, raw_ss_selector);
    return;
}

/* selector index must be within its descriptor table limits,
* else #GP(SS selector) */

bx_fetch_raw_descriptor(&ss_selector, &dword1, &dword2, BX_GP_EXCEPTION);
BX_HANDLE_EXCEPTION()

bx_parse_descriptor(dword1, dword2, &ss_descriptor);
/* AR byte must indicate a writable data segment, */
/* else #GP(SS selector) */

if ( ss_selector.segment==0 ||
    ss_descriptor.u.segment.executable ||
    ss_descriptor.u.segment.r_w==0 )
{
    bx_printf(1, "mcp_iret: SS AR byte not writable code segment\n");
bx_segment_exception(BX_GP_EXCEPTION, raw_ss_selector);
    return;
}
/* stack segment DPL must equal the RPL of the return CS selector, */
/* else #GP(SS selector) */

if ( ss_descriptor.dpl != cs_selector.rpl )

bx_printf(1, "mcp_iret: SS.dpl != CS selector RPL\n");
bx_segment_exception(BX_GP_EXCEPTION, raw_ss_selector);
return;
}
/* SS must be present, else #NS(SS selector) */
if ( ss_descriptor.p==0 )
{
    bx_printf(1, "mcp_iret: SS not present!\n");
bx_segment_exception(BX_NP_EXCEPTION, raw_ss_selector);
return;
}
/* IP must be in code segment limit, else #GP(0) */
if ( bx_cpu.eip > cs_descriptor.u.segment.limit )
{
    bx_printf(1, "mcp_iret: IP > descriptor limit\n");
bx_segment_exception(BX_GP_EXCEPTION, 0);
return;
}
prev_cpl = CPL; /* previous CPL */
/* load CS:IP from stack */
/* load the CS-cache with CS descriptor */
/* set CPL to the RPL of the return CS selector */
bx_load_cs(&cs_selector, &cs_descriptor, cs_selector.rpl);
bx_write_flags(flagsI6, prev_cpl==0, prev_cpl<=IOPL);
/* load the SS-cache with SS descriptor */
bx_load_ss(&ss_selector, &ss_descriptor, cs_selector.rpl);
bx_validate_es_and_ds();
if (bx_dbg.protected)
    bx_printf(0, "mcp_iret: WARNING ES & DS validity checks incomplete!\n");
return;
} /* done with return to OUTER level */
bx_printf(1, "IRET: shouldn't get here!\n");
return;
} /* else */
#endif
bx_pop_16(&ip);
bx_pop_16(&cs);
bx_pop_16(&flags);
/* ?? * /
if (flags & 0x0100) {
    bx_printf(0, "mcp_iret: found TF set\n");
bx_debug(&bx_cpu.cs, bx_prev_eip);
}
bx_load_seg_reg(&bx_cpu.cs, cs);
bx_cpu.eip = (Bit32u) ip;
The primary function which simulates MCA instructions in this research is contained in soft_int.c. The simulation of instructions is discussed in detail in Chapter 6. The main simulation code is contained in lines 226-740. Several routines to manipulate hardware context slot information and process IPC messages are in lines 95-225. Their function is as follows:

**mcs_st():** This function is called to display processor status, and was used during development of the MC/OS operating system primitives for detection of problems in implementation.

**cpmess_in():** This function copies a message from a task's memory space into the processor's physical register file, pointed to by the WAR entry in mcpp. This form of IPC simulation was chosen to simplify the development of test software for experimentation. It is a compromise between efficiency (e.g., building messages directly in MCA registers), and ease of implementation. This method keeps experimental programs nearly identical to those run on the unmodified Minix system.
cpmess_out() : This function copies a message from an MCA physical register back into the main memory location specified by the receiving task.

cpmess_over(): This function exchanges WAR entries between tasks in preparation for copying a message back to the receiving task’s address space. This is, in effect, the actual IPC operation that would occur in a hardware prototype version (or simulation) of the MCA architecture. Cpmess_in and cpmess_out exist in order to simplify simulation.

new_slot(): This function performs the simulation of the SSU, described in Chapter 3.

MCA instruction opcodes cannot occupy the byte immediately following the 80x86 soft interrupt opcode since regular vectored interrupts are specified here. As a result, this byte is used to indicate whether the interrupt is actually an MCA instruction or a standard interrupt. The actual MCA opcode resides in the second byte following the INT instruction opcode. For efficiency there are two types of MCA instructions, IPC and other. This allows IPC instructions to be decoded more quickly and easily since the type of IPC is stored in the third byte. IPC instructions use interrupt vector value 240, while all other MCA instructions use vector number 250. IPC instructions are evaluated in lines 249-345, while other instructions are decoded in lines 346-740.

1 /* soft_int.c
2 * This module processes, briefly, INT instructions. Most of the work,
3 * however, is done in exception.c. System might be optimized a bit
4 * in future to call bx_interrupt directly from bx_cpu_loop in decode.
5 * BUT, we'll put that off for awhile.
This file hacked, very much, by Tim Killeen. Basically, I've added code in bx_int_1b to handle "in hardware" a sequence of interrupts to support multiple context processor simulation. Programs outside this line of research should be totally unaffected. The exception, of course, would be if your interrupt vector approaches 256 software interrupts.

Copyright Notice
===============


BOCHS is shareware for PERSONAL USE only.

For more information, read the file 'LICENSE' included in the bochs distribution. If you don't have access to this file, or have questions regarding the licensing policy, the author may be contacted via:

US Mail: Kevin Lawton
528 Lexington St.
Waltham, MA 02154

EMail: bochs@world.std.com

/*
#include "bochs.h"
#include "instr_proto.h"

/* All changes for MCA processor simulation are enclosed in * MCP_MODE compiler directives. */
#endif
#define ANY 132 /* minix "any" slot */
int mcp_show = 0, mcp_msg = -1;
int lost_tiks, ttl_tiks;
#endif

INLINE void
mcp_st ()
/* show state of cpu */
{
int val;
for (val = 1; val < MCP_MODE; val++)
if (mcp[va] > 0)
printf ("%0i%0iw%0i ", (Bit16s) mcp[val], mcp[val], (Bit16s) mcpw[val]);
putchar ("n");
}
INLINE void mcp_state ()
/* show state of cpu */
{
    int val;
    for (val = 0; val < MCP_MODE; val++)
        if (mcpi[val] == mcpi[val])
            printf ("slot %i, as %i, status %i, t/f %i loc %i eip %i, sch %i\n",
                val, (Bits) mcpt[val], mcpi[val], (Bits) mcpw[val], mcpl[val], mcp[val].eip, mcprj[val]);
}
INLINE void cpmess_in (int slot, Bits bx)
/* routine extracts a message from 286's memory, and stores in
 * mcpm buffer for the slot currently executing.
 */
{
    Bits proc_num = mcpt[slot]; /* who the process is in O/S */
    Bits p1, *p2;
    p1 = mcpp[slot];
    p2 = p1 + 1;
    if (slot == 0)
        proc_num = -1; /* kernel IS hardware */
    /* do I care about the above??? */
    if (slot != mcp_slot)
        bxaccessphysical (mcp[slot].ds.cache.u.segment.base + bx, /*where from*/
            24 /* how much*/, BX_READ, mcpp[slot] /* where to*/);
    else
        bxaccessphysical (bx_cpu.ds.cache.u.segment.base + bx, /*where from*/
            24 /* how much*/, BX_READ, mcpp[slot] /* where to*/);
    if (proc_num >= 0 || (Bits) p1 != -1) /* kernel can send on behalf */
        *p1 = proc_num; /* of hardware, but others mustn't */
}
INLINE void cpmess_out (int to, Bits addr)
/* copies a message from MCP message buffer in simulated system into
 * needed for simulation, and would vanish in a real MCP processor
 */
{
    bxaccessphysical (mcp[to].ds.cache.u.segment.base + addr, /*where to*/
        24, BX_WRITE, mcpp[to] /* where from*/);
}
INLINE void cpmess_ovr (int to, int fr)
/* routine copies message buffer between two resident processes.
 * This IS a window exchange in the simulated system.
 */
* Accomplished by simply swapping pointers to buffers.
* /
{
    char *t = mcpp[to];
    mcpp[to] = mcpp[fr];
    mcpp[fr] = t;
}

INLINE void new_slot()
/* Select a new MCP context for execution, the last one has probably
* blocked. New contexts can be selected every clock, but in simulation
* this would be painfully slow. In actual practice this should be
* done in parallel (in hdw) with normal execution, so would be free
* As a result, might consider two "modes" of execution: new_slot
* after EACH instruction, or new_slot only on context blocks.
* 
* Slot states (stored in mcpi):
* 0 - Empty/available slot
* 1 - ready/slot occupied
* 2 - blocked on receive
* 3 - blocked on receive
* 5 - blocked on send in send/receive pair.
* 99 - was running when interrupt to kernel occurred.
* This is reset when kernel is finished
* processing interrupt.
* 
* Context 0, by definition, MUST always be active. This is essentially
* the "idle" task in the kernel. If this task is ever run it should
* take action to select a ready, but non-resident, task to run.
* This then IS the long-term scheduler.
* /
{
    int slot = 0;
    mcp[mcp_slot] = bx_cpu; /* save current */
    if (mcp[prev] > 0)
    {
        if (mcpi[mcp_prev] == 99) /* was INT'd ?? */
        {
            mcpi[mcp_prev] = 1; /* so restart this one */
            mcp_slot = mcp_prev;
            bx_cpu = mcp[mcp_slot];
            mcp_prev = 0;
            return;
        }
    }
    /* possibly schedule this task? */
}

mcp_slot = 1;
while (mep_slot < MCP_MODE) /* active iff 1 */
{
  if (mcp[mcp_slot] == 99)
    mcp[mcp_slot] = 1;
  if (mcp[mcp_slot] == 1 && slot == 0)
    slot = mcp_slot; /* select first available */
  else if (mcp[mcp_slot] == 1 && (Bit16s) mcpt[mcp_slot] < 0)
    slot = mcp_slot; /* select task over user */
  mcp_slot++;
}
/* if (mcp_prev==-1 && slot==0) */
  *
    * mcp_st();
    * scanf("%c", &mcp_slot);
    * if (mcp_slot=='.')
    *  mcp_show=3;
    * else
    *  mcp_show=0;
  */
  */
  mcp_slot = slot;
  bx_cpu = mcp[mcp_slot]; /* restore new */
if (mcp_slot)
  mcp_prev = 0; /* found a slot */
else
  mcp_prev = -1; /* no slot loaded */
}
#endif

INLINE void bx_INT_1b ()
/* This routine handles the full range of supported software interrupts. */
/* Most are routed off to bx_interrupt in exception.c, where normal */
/* vectored interrupts get processed. */
/* In addition, several hardware processed, but software initiated, */
/* interrupts are managed here. These have been added to test new */
/* architectural extensions to support multiple resident contexts and */
/* efficient IPC (interprocess communication). */
{
  Bit8u imm8;
  #if MCP_MODE > 0
  Bit8u msg_s;
  Bit16u val, psw, es_val, ds_val;
  Bit32u addr, dw1, dw2;
  Bit16s pr;
  bx_selector_t selector;
  bx_descriptor_t descriptor;
#endif

im8 = bx_fetch_next_byte();
BX_HANDLE_EXCEPTION()

#if MCP_MODE > 0
    if (im8 >= 240)
        /* note: NONE of these instructions are currently doing any
        * validation. That can be addressed *later*
        */

    mepic[1]++;
    if (im8 == 240) /* send or receive msgs */
        
            im8 = CX; /* 1=send, 2=rcv, 3=both */
            if (mcp_show > 2)
                printf("240: %i %i %i\n", im8, mcp_slot, (Bit16s) AX);
            mcpin[im8]++;
            /* record # of each */

    switch (im8)
        
            case 1: /* send */
            case 3: /* both, do send first */
                cpmess_in (mcp_slot, BX); /* move msg into buf */
                for (msg_s = 1; msg_s < MCP_MODE; msg_s++)
                    if (mcp[msg_s] == AX &&& mcp[i][msg_s] == 2)
                        if (mcpw[msg_s] == ANY || mcpw[msg_s] == mcpt[mcp_slot])
                            
                                cpmess_ovr (msg_s, mcp_slot);
                                cpmess_out (msg_s, mcp[[msg_s]]);
                                if (mcp_show)
                                    
                                        printf("s%i -> %i ", (Bit16s) mcpt[mcp_slot], (Bit16s) mcpt[msg_s]);
                                        mep_st();
                                    
                                mcpi[msg_s] = 1; /* unblock task */
                                mcp[mcp_slot] = mcps[msg_s] = 5;/* very active this period */
                                break; /* jump out of loop */

                            
                    if (msg_s == MCP_MODE) /* no rcvr, block task */
                        
                            if (mcp_slot == 0)
                                
                                    printf("kernel send is blocking %i\n", (Bit16s) AX);
                                    /*
                                        scanf(" %e",&val);
                                        if (val>='1')
                                        {
                                            if (val=='s')
                                                * mcp_show=0;
                                        * for (val=0; val<MCP_MODE; val++)
if (mcpi[val] == mcpi[val])
printf(\"slot %i, as %i, status %i, t/f %i loc %i eip %i, sch %i\n\", val, (Bit16s)mcpt[val], mcpi[val], mcpw[val], mcpl[val].eip, mcpr[val]);
*/
}
mcpi[mcp_slot] = imm8 + 2; /* blocked on send */
mcpw[mcp_slot] = AX; /* who sending to */
mcpl[mcp_slot] = BX; /* where to put on rcv later */
AX = 0;
mcp_msg = mcp_slot;
new_slot (); /* run a new context */
break;
else if (imm8 != 3) /* only send, quit */
  
  
  AX = 0; /* success */
  break;
  */
/* otherwise, just fall right into receive (& block) */
case 2: /* receive */
  for (msg_s = 1; msg_s < MCP_MODE; msg_s++) /* look for a sendr */
    
    if (mcpw[msg_s] == mcpt[mcp_slot] && mcpi[msg_s] >= 3)
      if (AX == ANY || AX == mcpt[msg_s]) /* got one! */
        
        
        cpmsg_ovr (mcp_slot, msg_s);
        cpmsg_out (mcp_slot, BX);
        mcps[mcp_slot] = mcps[msg_s] = 5;
        if (mcp_show) /* diagnostic stuff */
          
          
          
          printf (\"%i \rightarrow %i\", (Bit16s) mcpt[mcp_slot], (Bit16s) mcpt[msg_s]);
          mcp_st ();
        
        if (mcpi[msg_s] == 3) /* just a send, unblock */
          mcpi[msg_s] = 1;
        else
          
          mcpi[msg_s] = 2; /* sendrec, block on rcv */
          AX = 0; /* success */
          break;
      
      if (msg_s == MCP_MODE) /* no sendr, block */
        
        
        mcpi[mcp_slot] = 2; /* block on rcv */
        mcpw[mcp_slot] = AX; /* want to rcv from */
        mcpl[mcp_slot] = BX; /* where in heck to put it */
        AX = 0; /* will work someday */
mcp_msg = mcp_slot;
new_slot(); /* run new context */
#endif
break;
default: /* not sure what it is */
    break; /* so skip it */
}
#endif
#else if (imm8 == 250)
    imm8 = CX; /* actual call number passed as arg */
#endif
if (mcp_show > 2)
    printf("250: %i %i\n", imm8, mcp_slot);
    mcpin[imm8 - 200]++; /* record # of each */
switch (imm8)
{
    case 230: /* set state */
        mcpi[AX] = BX;
        /* now must check if this will result in messages */
        /* begin exchanged */
        /* */
        if (mcpi[AX] == 2) /* waiting, see if someone sending */
            { /*
                for (msg_s = 0; msg_s < MCP_MODE; msg_s++)
                    if (mcpw[msg_s] == mcpt[AX] && mcpi[msg_s] >= 3)
                        if (mcpw[AX] == ANY || mcpw[AX] == mcpt[msg_s])
                            { /*
                                cpmess_ovr (AX, msg_s);
                                cpmess_out (AX, mcpi[AX]);
                                /*printf("230:mr: %i/%i", AX, mcpt[AX], mcpi[AX]); */
                                if (mcpi[msg_s] == 3)
                                    mcpi[msg_s] = 1;
                                else
                                    mcpi[msg_s] = 2;
                                    mcpi[AX] = 1;
                                    mcps[AX] = mcps[msg_s] = 4;
                                    break; /* only 1 allowed */
                            } /*
                } */
            else if (mcpi[AX] == 3 || mcpi[AX] == 5) /* sending */
                { /*
                    * printf("230:ld %i as %i, mode %i\n", AX, (Bit16s)mcpt[AX], mcpi[AX]); */
                    * mcp_st();
                    */
            cpmess_in (AX, mcpl[AX]);
            for (msg_s = 1; msg_s < MCP_MODE; msg_s++)
            for (msg_s = 0; msg_s < MCP_MODE; msg_s++)
if (mcpt[msg_s] == mcpw[AX] && mcpi[msg_s] == 2)
    if (mcpw[msg_s] == ANY || mcpw[msg_s] == mcpt[AX])
        { /* printf("230: %i %i %i\n", (Bit16s)mcpt[msg_s], mcpi[AX], (Bit16s)mcpt[AX]); */
            mcp_show = 1;
        }
break;
case 231: /* set who waiting/sending from/to */
    mcpw[AX] = BX;
    break;
case 232: /* set message buffer location */
    mcpl[AX] = BX;
    break;
case 233: /* check rcve in mem, sender loaded */
    val = 0;
    for (msg_s = 1; val == 0 && msg_s < MCP_MODE; msg_s++)
        val = ((AX == ANY || AX == mcpt[msg_s]) && mcpw[msg_s] == BX && mcpi[msg_s] > 2);
    AX = val;
    break;
case 234: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 235: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 236: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 237: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 238: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 239: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 240: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 241: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 242: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 243: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 244: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 245: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 246: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 247: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 248: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 249: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 250: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 251: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 252: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 253: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 254: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
case 255: /* return activity */
    val = AX;
    AX = mcpr[val] + 2 * mcps[val]; /* HOW active or not since last swap? */
    break;
}
/*
break;

case 236:
mcp_msg = -1;
for (val = 1; val < MCP_MODE; val++)
{
    if (mcpr[val])
        mcpr[val]--;
    if (mcpr[val] <= 99)
        mcpr[val] += 2 * mcps[val];
    if (mcps[val])
        mcps[val] /= 2;
else
    mcpr[val] /= 2;
}

break;

case 238:
printf ("total ticks %i, lost %i instr %i, ints %i\n", ttl_tiks,
        lost_tiks, mcpic[0], mcpic[1]);
printf ("by proc_nr:\n);
for (val = 0; val < 64; val++)
{
    if (mcpiw[val])
        printf ("%i %i\n", val, mcpiw[val]);
    mcpiw[val] = 0;
}
printf ("by int:\n);
for (val = 0; val < 56; val++)
{
    if (mcpin[val])
        printf ("%i %i\n", val, mcpin[val]);
    mcpin[val] = 0;
}
printf ("by opcode:\n);
for (val = 0; val < 256; val++)
{
    /* if (mcpit[val])
        * printf("%i %i\n",val,mcpit[val]);
        */
    mcpit[val] = 0;
}
mcpic[0] = mcpic[1] = 0;

break;

case 239: /* return getfrom if blocked, else 0 */
    /* mcp_st();
    */
    scanf(" %c",&val);
    /*
    val = AX;
while (val)
    if (mcpi[val] == 1 || mcpi[val] == 99)
        val = AX = 0;
    else
        val--;
if (AX)
    AX = mcp_msg;
break;
case 240: /* return last running slot (used on INT) */
    AX = mcp_prev;
break;
case 241: /* reset an INTd job */
    mcpi[AX] = 1; /* now runnable */
    break;
case 242: /* another debug call */
    if ((Bit16s) AX < 80)
        printf("242: slot %i %i %i\n", mcp_slot, (Bit16s) AX, (Bit16s) BX);
    else
    {
        printf("%4i \n", (Bit16s) BX);
        if (AX == 81)
        {
            if ((Bit16s) BX >= 0)
                printf("%4i %4i %4i %4i \n", (Bit16s) mcpt[BX], mcpi[BX],
(Bit16s) mcpw[BX],
                        mcpr[BX]);
            else
                printf(" \n");
        }
        else if (AX == 84)
            putchar (\n');
    }
break;
case 243: /* give up processor (s/b task 0 only) */
    lost_tiks = AX;
    ttl_tiks = BX;
    new_slot ();
    break;
case 244: /* Register slot for messaging */
    /* done by kernel, this is when a task */
    /* is ready to execute! */
mcp[BX] = BX;
    if ((Bit16s) BX >= 9)
    {
        mcp_show = 3;
    }
    /*printf("244:register %i as %i\n",AX,(Bit16s)BX); */
meps[AX] = 3; /* mark as ACTIVE this period */
mcpi[AX] = 1; /* and runnable unless changed */
break;
case 245: /* register MY slot for messaging */
printf("245:register %i as %i\n", mcp_slot, AX);
mcpt[mcp_slot] = AX; /* O/S "task" number */
break;
case 246:
printf("246:setting slot %i to %is status\n", AX, BX);
mcpi[AX] = mcpi[BX];
mcpw[AX] = mcpw[BX];
mcpl[AX] = mcpl[BX];
break;
case 247:
printf("247: prev %i, slot %i CPL %i, eip %i, #=%i/%i enter a # ",
mcp_prev, mcp_slot, CPL,
  bx_cpu.eip, (Bit16s) AX, (Bit16s) BX);
scanf(" %c", &val);
AX = val;
if(val >= '1')
{
  if(val == 's')
    mcp_show = 1;
  for(val = 0; val < MCP_MODE; val++)
    if (mcpi[val] == mcpt[val])
      printf("slot %i, as %i, status %i, t/f %i loc %i eip %i, sch %i\n",
          val, (Bit16s) mcpt[val], mcpi[val], mcpw[val], mcpl[val],
mcp[val].eip, mcpr[val]);
  if (AX == '5')
    mcpD = 1;
  break;
}
case 248:
if (BX == 0)
  AX = mcpi[AX]; /* return status of slot AX */
else if (BX == 1)
  AX = mcpw[AX]; /* return who waiting on */
else if (BX == 2)
  AX = mcpl[AX]; /* return where to/from */
break;
case 249: /* switch W/O save first */
mcp_slot = AX; /* one to go to */
printf("249: switching to slot %i %i\n", mcp_slot, bx_cpu.eip);
mcp[mcp_slot] = bx_cpu; /* save state */
mcp[ AX] = 1; /* make active/runnable */
break;
case 250: /* copy & switch contexts */
mcp[mcp_slot] = bx_cpu; /* save current task state */
mcp_slot = AX; /* contains slot # to go to */
printf("250 switching to slot %i %u\n", mcp_slot, bx_cpu.eip);
mcp[mcp_slot] = bx_cpu;            /* and copy & go */
mcp[mcp_slot] = 1;            /* active & runnable */
break;

case 251: /* just switch contexts */
mcp[mcp_slot] = bx_cpu;            /* always gotta save old */
printf("251 switching from %i %u -- It, mcp_slot, bx_cpu.eip);
mcp_slot = AX;                /* get new slot # */
bx_cpu = mcp[mcp_slot];                /* & go to new */
printf("to %i %u\n", mcp_slot, bx_cpu.eip);
break;

case 252: /* store context in PCB */
if (mcp_show)
{
    printf("252 %i->%i ", mcp_slot, (Bit16s) mcpt[AX]);
mcp_st ();
}
mcp[mcp_slot] = bx_cpu;            /* save current, so can use it */
addr = bx_cpu.ds.cache.u.segment.base + BX + 4;            /* PCB area */

val = AX;
if (mcp_show > 2)
printf("252: save %i\n", val);
bx_cpu = mcp[AX];            /* store this one */
/* note we skip segment reg. descriptor stores */
/* they won't change on save/restore */
/* printf("252: slot %i, limit %i, eip/cs %i %i, ss/SP %i %i iopl %i\n", val,
* bx_cpu.cs.cache.u.segment.limit,bx_cpu.eip,
* bx_cpu.cs.cache.u.segment.base,
* bx_cpu.ss.cache.u.segment.base,SP,bx_cpu.eflags.iopl);
*/
bx_access_physical (addr, 2, BX_WRITE, &DI);
addr += 2;
bx_access_physical (addr, 2, BX_WRITE, &SI);
addr += 2;
bx_access_physical (addr, 2, BX_WRITE, &BP);
addr += 4;
bx_access_physical (addr, 2, BX_WRITE, &BX);
addr += 2;
bx_access_physical (addr, 2, BX_WRITE, &DX);
addr += 2;
bx_access_physical (addr, 2, BX_WRITE, &CX);
addr += 2;
bx_access_physical (addr, 2, BX_WRITE, &AX);
addr += 4;
bx_access_physical (addr, 2, BX_WRITE, &IP);
addr += 2;
addr += 2;            /* skip cs descriptor write */
psw = bx_read_flags();
bx_access_physical(addr, 2, BX_WRITE, &psw);
addr += 2;
bx_access_physical(addr, 2, BX_WRITE, &SP);
addr += 2;
/* that's all for now folks */
bx_cpu = mcp[mcp_slot];
break;

case 253: /* Set up new context */
    mcp[mcp_slot] = bx_cpu; /* save this so can do easily */
    imm8 = AX; /* new slot # */
    addr = bx_cpu.ds.cache.u.segment.base + BX; /* PCB area */
    bx_access_physical(addr, 2, BX_READ, &es_val);
    addr += 2;
    bx_access_physical(addr, 2, BX_READ, &ds_val);
    addr += 2;
    bx_access_physical(addr, 2, BX_READ, &DI);
    addr += 2;
    bx_access_physical(addr, 2, BX_READ, &SI);
    addr += 2;
    bx_access_physical(addr, 2, BX_READ, &BP);
    addr += 2;
    bx_access_physical(addr, 2, BX_READ, &BX);
    addr += 2;
    bx_access_physical(addr, 2, BX_READ, &DX);
    addr += 2;
    bx_access_physical(addr, 2, BX_READ, &CX);
    addr += 2;
    bx_access_physical(addr, 2, BX_READ, &AX);
    addr += 4;
    bx_access_physical(addr, 2, BX_READ, &IP);
    addr += 2;
    bx_access_physical(addr, 2, BX_READ, &val);
    addr += 2;
    bx_cpu.cs.cache.u.segment.base = val;
    /* above is WRONG, but OK, since set in mcp_load */
    bx_access_physical(addr, 2, BX_READ, &psw);
    addr += 2;
    bx_access_physical(addr, 2, BX_READ, &SP);
    addr += 2;
    bx_access_physical(addr, 2, BX_READ, &val);
    addr += 2;
    bx_cpu.ss.cache.u.segment.base = val;
    /* see comment above for cs */
    /* set up local descriptor table entries */
    bx_access_physical(addr, 2, BX_READ, &val);
    if ((val & 0xfffc) == 0) /*selector null? */
    {
bx_printf(1, "mcp:LLDT: selector null\n");
bx_load_lldr(&selector, NULL);
break;
}
bx_parse_selector(val, &selector); /* parse fields in it */
if(selector.ti != 0)
{
    bx_printf(1, "mcp:LLDT: selector.ti !=0\n");
bx_segment_exception(BX_GP_EXCEPTION, val);
break;
}
if(selector.index * 8 + 7 > bx_cpu.gdtr.limit)
{
    bx_printf(1, "mcp:lldt: GDT: index>limit\n");
bx_segment_exception(BX_GP_EXCEPTION, val);
break;
}
bx_access_physical(bx_cpu.gdtr.base + selector.index * 8, 4,
BX_READ, &dw1);
bx_access_physical(bx_cpu.gdtr.base + selector.index * 8 + 4, 4,
BX_READ, &dw2);
bx_parse_descriptor(dw1, dw2, &descriptor);
if(descriptor.segment || descriptor.type != 2)
{
    bx_printf(1, "mcp:lldt: doesn't point to an LDT descriptor\n");
bx_segment_exception(BX_GP_EXCEPTION, val);
break;
}
if(descriptor.p == 0)
{
    bx_printf(1, "mcp:lldt: LDT descriptor not present\n");
bx_segment_exception(BX_GP_EXCEPTION, val);
break;
}
bx_load_lldr(&selector, &descriptor);
descriptor.valid = 1;

/* establish ES & DS from selector values */
bx_load_seg_reg(&bx_cpu.es, es_val);
bx_load_seg_reg(&bx_cpu.ds, ds_val);

/* now perform a pseudo IRET */
mcp_IRET(addr, psw);

mcp[imm8] = bx_cpu; /* put entry into table */
bx_cpu = mcp[mcp_slot]; /* restore current state */
break;

case 254: /* mark a slot as empty */
AX = mcpi[BX];
mcpi[BX] = mcpr[BX] = mcps[BX] = 0; /* now it's empty */
break;
case 255: /* display slot info (debug) */
  if (mcp_slot && mcp_slot < 16)
    {
      for (val = 1; val < 16; val++)
        if (mcpi[val] > 0)
          printf("slot %i, status %i reg as %i, to/fr who %i\n", val,
                 mcpi[val], mcpt[val], mepw[val]);
      printf("%i %i Press a key ", mcp_slot, AX);
      scanf(" %c", &val);
    }
  break;
  default:
    bx_printf (0, "unimplemented MPC call: %u\n", imm8);
    break;
  }
else
  {
    if (mcp_slot && imm8 != 34)
      {
        printf("soft int: %i %i\n", mcp_slot, imm8);
        if (imm8 == 32)
          {
            printf(" press a key ");
            scanf(" %c", &val);
          }
      }
    #endif
    bx_interrupt(imm8, 1, 0);
  #if MCP_MODE>0
    }
  #endif
  }
  INLINE void
bx_INTO()
  {
    if (bx_cpu.eflags.of)
      bx_interrupt(4, 1, 0);
  }
Appendix III

Experimental Data and Simulation Results

The experimental data produced by the bochs simulator for each set of experiments is included and described in this Appendix. The method used to gather data was described in Chapter 7 of the main body of the dissertation. Different programs were selected to fit into several execution profiles such as distributed, I/O intensive, and computation intensive routines. Each set of programs was then executed in the simulation environment on both the unmodified Minix system as well as several configurations of the MCA processor.

All aspects of the system hardware were kept fixed except the number of hardware context slots. The number of slots was varied from 4-24. The upper limit of 24 was selected to greater than the maximum number of active tasks observed at any point in any of the experiments. This represents a lightly loaded, in terms of hardware context utilization, system while a 4 context configuration corresponds to a more heavily loaded system.

Before each program was executed, a special MCA instruction was issued which causes the bochs simulator to print resultant data to a file, and clear counters used for accumulating results. Following the completion of each program, the same MCA instruction was issued a second time, displaying the desired results to the file again. Since values were reset after each call, only the final set of values was used.
Each experiment gathered statistics in three distinct areas: instruction counts by task, MCA instructions by MCA opcode, and 80x86 instructions by opcode.

Instruction counts by software task were used to identify the number of instructions executed by the MC/OS task scheduler and kernel. This makes it possible to determine the amount of overhead and processor utilization.

MCA instructions by MCA opcode made it possible to determine the amount of IPC and scheduling activity during an experiment. This was useful in determining how representative the program was for the type experiment being conducted. A compute intensive program, for example, should not exhibit a high degree of IPC activity.

In addition, the MCA opcode count also made it possible to revise total instruction counts to compensate for the simulation methods used. There were two compensations applied. First, since each MCA instruction was actually simulated by invoking a C library routine, sendmcp, the measured 80x86 instruction count was higher than it would have been if MCA instructions were simulated directly. The sendmcp library routine adds to the total instruction count a procedure call, procedure return, and several memory move operations to place arguments into 80x86 registers. This was determined to add 8 instructions to the total instruction count for each MCA instruction simulated.

The second adjustment was made for hardware context load and save operations, which
were simulated using a single sendmcp call apiece. This would clearly require several instructions, resulting in under reported total instruction count. The number of instructions to load or save a task state is dependent on the hardware architecture as well as how much state is saved, specialized instructions used for state saving, etc. We estimated that a full state load or save operation in an MCA processor would add at least 50 instructions to the total instruction count. This is roughly the number required to save/load the state on an Intel processor if special instructions are not used. As a conservative measure, therefore, we added 500 to the total count for each state load/save. In most cases, the actual value added is small compared with the total instruction count because the number of state save/load operations was extremely small.

The following sections contain the original experimental results used to develop the tables and figures contained in Chapter 7. The instruction counts by 80x86 opcode have been omitted from most results in order to save space. The specific values by opcode were not a part of this study.

III.1 Minix Instruction Counts

This section contains experimental results for the system boot loader, unmodified Minix 1.7.2, and test programs. Each set is described below.

**Boot Loader:** The loader is used to initialize the hardware and initially load the operating system into memory. Since it is independent from this study, the instruction counts for the
load process are removed from the instruction counts for loading the remainder of the Minix system. The values are included here to illustrate the unedited output from the bochs system. In addition, the total instruction count is deducted from the total instruction count for Minix experiments. This is necessary because results for Minix experiments could only be displayed once (since MCA instructions are not a part of Minix). Hence, a good deal of editing is required to get accurate experimental values for the unmodified Minix results.

The number shown below were generated by displaying results immediately after control is transferred from the loader to the Minix start() function.

Boot loader total instructions 378054, interrupts 1 (MCA instructions)
by proc_nr:
0 378054
by int:
38 1
by opcode:
opcode followed by instruction count:
0 98
1 3998
3 2627
5 287
6 62
7 50
8 1574
9 5468
11 1361
12 1
14 2
15 699
16 9
17 1000
19 1449
21 224
25 461
27 518
29 16
30 9
31 20
<table>
<thead>
<tr>
<th>Number</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>904</td>
</tr>
<tr>
<td>35</td>
<td>244</td>
</tr>
<tr>
<td>36</td>
<td>4</td>
</tr>
<tr>
<td>37</td>
<td>1069</td>
</tr>
<tr>
<td>38</td>
<td>4</td>
</tr>
<tr>
<td>40</td>
<td>99</td>
</tr>
<tr>
<td>41</td>
<td>2619</td>
</tr>
<tr>
<td>43</td>
<td>1116</td>
</tr>
<tr>
<td>45</td>
<td>441</td>
</tr>
<tr>
<td>46</td>
<td>18</td>
</tr>
<tr>
<td>48</td>
<td>1319</td>
</tr>
<tr>
<td>49</td>
<td>7463</td>
</tr>
<tr>
<td>51</td>
<td>1</td>
</tr>
<tr>
<td>57</td>
<td>7698</td>
</tr>
<tr>
<td>58</td>
<td>53</td>
</tr>
<tr>
<td>59</td>
<td>430</td>
</tr>
<tr>
<td>60</td>
<td>513</td>
</tr>
<tr>
<td>61</td>
<td>297</td>
</tr>
<tr>
<td>64</td>
<td>1121</td>
</tr>
<tr>
<td>65</td>
<td>151</td>
</tr>
<tr>
<td>66</td>
<td>518</td>
</tr>
<tr>
<td>67</td>
<td>37</td>
</tr>
<tr>
<td>70</td>
<td>1661</td>
</tr>
<tr>
<td>71</td>
<td>150</td>
</tr>
<tr>
<td>73</td>
<td>487</td>
</tr>
<tr>
<td>74</td>
<td>27</td>
</tr>
<tr>
<td>78</td>
<td>28</td>
</tr>
<tr>
<td>79</td>
<td>35</td>
</tr>
<tr>
<td>80</td>
<td>10082</td>
</tr>
<tr>
<td>81</td>
<td>4011</td>
</tr>
<tr>
<td>82</td>
<td>3198</td>
</tr>
<tr>
<td>83</td>
<td>1704</td>
</tr>
<tr>
<td>84</td>
<td>4</td>
</tr>
<tr>
<td>85</td>
<td>5685</td>
</tr>
<tr>
<td>86</td>
<td>7218</td>
</tr>
<tr>
<td>87</td>
<td>4113</td>
</tr>
<tr>
<td>88</td>
<td>1005</td>
</tr>
<tr>
<td>89</td>
<td>3269</td>
</tr>
<tr>
<td>90</td>
<td>2652</td>
</tr>
<tr>
<td>91</td>
<td>8538</td>
</tr>
<tr>
<td>93</td>
<td>5673</td>
</tr>
<tr>
<td>94</td>
<td>4980</td>
</tr>
<tr>
<td>95</td>
<td>4045</td>
</tr>
<tr>
<td>106</td>
<td>1</td>
</tr>
<tr>
<td>114</td>
<td>2321</td>
</tr>
<tr>
<td>115</td>
<td>3883</td>
</tr>
<tr>
<td>116</td>
<td>10396</td>
</tr>
<tr>
<td>117</td>
<td>16060</td>
</tr>
</tbody>
</table>
Minix Distributed Application: The distributed application observed in this study was the Minix (and MC/OS) operating system during initialization.

total instructions 10511907, ints 1
by proc_nr:
0 10511907
by int:
38 1
The total instruction count shown above is prior to deducting the instruction count for the boot loader. When this is accounted for, the instruction count corresponds to that given in Table X, for a single hardware context.

**Minix roff Application:** The UNIX command roff was used to format a large, 65k UNIX man page for output. As above, the number shown below include instructions unrelated to the experiment. The values shown also reflect the Minix distributed application values. These results were excluded from the roff experiment output to get the values specific to the roff command used in Table XI.

Unmodified Minix: roff  
total ins 30791436, ints 1  
by proc_nr:  
0 30791436  
by int:  
38 1

Minix sort Application: The UNIX command sort was used to sort the contents of a 65k byte ASCII text file into ascending alphabetical order by line. There were approximately 1000 lines of text in the file. The values shown also reflect the Minix distributed application values. These results were subtracted from the sort experiment output to get the values specific to the sort command used in Table XI.

Unmodified Minix: sort  
total instructions 20218745, ints 1 (MCA instructions)  
by proc_nr:  
0 20218745  
by int:  
38 1

Minix compress Application: The UNIX command sort was used to compress the contents
of a 65k byte ASCII text file. The values shown also reflect the Minix distributed application values. These results were deleted from the compress experiment output to get the values specific to the compress command used in Table XI.

Unmodified Minix: compress
total ins 22728663, ints 1
by proc_nr:
  0 22728663
by int:
  38 1

Minix Concurrent Execution: The three commands listed above were run separately and concurrently. Concurrent results were given in Table XII, for a single hardware context. The values shown also reflect the Minix distributed application values. These results were deducted from the concurrent experiment output to get the values specific to those used in Table XII.

Unmodified Minix: concurrent
total instructions 54963911, ints 1 (MCA)
by proc_nr:
  0 54963911
by int:
  38 1

Minix I/O Application: The UNIX command fsck was used to check file system integrity of the main user file system. The values shown also reflect the Minix distributed application values. These results were subtracted from the sort experiment output to get the values specific to the fsck command used in Table XIII.

Unmodified Minix: I/O
total ins 37292597, ints 1
by proc_nr:
  0 37292597
by int:
  38 1
### III.3 MC/OS IPC Directed Scheduling Results

Results for MC/OS and MCA simulation experiments are presented in this section. Since MCA instructions were simulated, the instruction to display instruction count values could be issued at any time. As a result, the values presented did not require any special processing before being placed in tables apart from the adjustments discussed previously to account for MCA instruction simulation.

The values used to build Table X for hardware contexts 4-24 are shown below.

<table>
<thead>
<tr>
<th>Context</th>
<th>Total Instructions</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8266932</td>
<td>53812</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>379408</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>15092</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>467684</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>588</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>2509</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>400962</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>9700</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>490623</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>295028</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>2966898</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>10703</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>1289512</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>105937</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>140464</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>9770</td>
<td></td>
</tr>
</tbody>
</table>

by proc_nr:
0 1681919
9 379408
10 15092
11 467684
12 588
14 2509
15 135
16 400962
17 9700
18 490623
20 295028
21 2966898
22 10703
23 1289512
24 105937
25 140464
26 9770

by int:
1 4453
2 4488
3 4469
30 1243
31 1243
MC/OS Distributed: 5 Contexts
total ins 7997505, ints 47693

by proc_nr:
0 1302363
9 366023
10 15092
11 470063
12 588
14 554
15 135
16 409503
17 111115
18 508132
20 304503
21 3042194
22 10703
23 1286960
24 119343
25 140464
26 9770

by int:
1 4536
2 4572
3 4556
30 889
31 889
32 889
34 4748
35 3616
36 904
38 1
43 3351
44 926
48 13013
52 929
53 926
54 921
55 2027
MC/OS Distributed: 6 Contexts
total ins 7716265, ints 37560
by proc_nr:
0 1022215
9 366023
10 15092
11 470063
12 588
14 821
15 135
16 409503
17 11038
18 507086
20 304251
21 3042210
22 10703
23 1286960
24 119343
25 140464
26 9770
by int:
1 4533
2 4568
3 4554
30 335
31 335
32 335
34 2927
35 1700
36 340
38 1
43 2883
44 372
48 11611
52 365
53 372
54 361
55 1968
MC/OS Distributed: 8 Contexts
total ins 7480437, ints 29658
by proc_nr:
0 774009
9 366143
10 15092
11 470063
12 588
14 1266
15 135
16 409503
17 11038
18 519436
20 303484
21 3042440
22 10703
23 1286960
24 119343
25 140464
26 9770
by int:
1 4534
2 4570
3 4554
30 97
31 97
32 97
34 1949
35 595
36 85
38 1
43 2763
44 134
48 7746
52 110
53 134
54 104
55 2088
MC/OS Distributed: 10 Contexts

total ins 7469080, ints 31071

by proc_nr:

0 759335
9 366577
10 15172
11 470063
12 588
14 910
15 135
16 408845
17 11143
18 522662
20 303850
21 3042560
22 10703
23 1286960
24 119343
25 140464
26 9770

by int:

1 4536
2 4571
3 4554
30 36
31 36
32 36
34 1780
35 180
36 20
38 1
43 2798
44 73
48 10354
52 45
53 73
54 41
55 1937
MC/OS Distributed: 16 Contexts
total ins 7150169, ints 26528
by proc_nr:
0 452882
9 366817
10 15172
11 470063
12 588
14 732
15 135
16 409503
17 11038
18 509704
20 303903
21 3042476
22 10703
23 1286960
24 119343
25 140380
26 9770
by int:
1 4538
2 4572
3 4553
30 25
31 25
32 25
34 49
35 15
36 1
38 1
43 9115
44 62
48 1392
52 26
53 62
54 22
55 2045
MC/OS Distributed: 20 Contexts

total ins 7111287, ints 24903

<table>
<thead>
<tr>
<th>proc_nr</th>
<th>byproc_nr</th>
<th>ints</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>410948</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>366732</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>15172</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>470063</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>588</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>3310</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>409503</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>11038</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>509793</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>304289</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>3042476</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>10703</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>1286960</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>119343</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>140464</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>9770</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>byint</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4538</td>
</tr>
<tr>
<td>2</td>
<td>4574</td>
</tr>
<tr>
<td>3</td>
<td>4554</td>
</tr>
<tr>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td>31</td>
<td>25</td>
</tr>
<tr>
<td>32</td>
<td>25</td>
</tr>
<tr>
<td>33</td>
<td>50</td>
</tr>
<tr>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td>43</td>
<td>8532</td>
</tr>
<tr>
<td>44</td>
<td>62</td>
</tr>
<tr>
<td>48</td>
<td>480</td>
</tr>
<tr>
<td>52</td>
<td>25</td>
</tr>
<tr>
<td>53</td>
<td>62</td>
</tr>
<tr>
<td>54</td>
<td>21</td>
</tr>
<tr>
<td>55</td>
<td>1929</td>
</tr>
</tbody>
</table>
MC/OS Distributed: 24 Contexts
total ins 7146096, ints 26188
by proc_nr:
0 448462
9 366694
10 15172
11 470063
12 588
14 643
15 135
16 409503
17 11038
18 509793
20 304289
21 3042476
22 10703
23 1286960
24 119343
25 140464
26 97770
by int:
1 4537
2 4573
3 4554
30 25
31 25
32 25
34 50
38 1
43 8927
44 62
48 1220
52 25
53 62
54 21
55 2081
MC/OS roff, sort and compress Results: The same set of UNIX commands that were run under Minix were also run under several MCA configurations. These results are shown below.

MC/OS roff: 6 contexts
total ticks 3242, lost 47   instr 19951539, ints 9106
by proc_nr:
  0 370803
  11 100604
  16 1935
  17 1264
  18 92749
  20 28436
  21 401135
  24 70
  25 38972
  26 18915571
by int:
  1 534
  2 534
  3 534
  30 19
  31 19
  32 19
  33 26
  35 90
  36 18
  38 1
  39 2381
  42 1
  43 2381
  44 21
  48 101
  52 20
  53 21
  54 19
  55 2367

MC/OS roff: 16 contexts
total ticks 3361, lost 54   instr 19961650, ints 9142
by proc_nr:
  0 359448
  11 103525
  16 4900
  17 2886
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>92890</td>
<td>20</td>
<td>30686</td>
<td>21</td>
<td>412155</td>
</tr>
<tr>
<td>24</td>
<td>617</td>
<td>25</td>
<td>38972</td>
<td>26</td>
<td>18915571</td>
</tr>
<tr>
<td></td>
<td>by int:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>560</td>
<td>2</td>
<td>562</td>
<td>3</td>
<td>560</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>31</td>
<td>3</td>
<td>32</td>
<td>3</td>
</tr>
<tr>
<td>33</td>
<td>6</td>
<td>38</td>
<td>1</td>
<td>39</td>
<td>2468</td>
</tr>
<tr>
<td>42</td>
<td>1</td>
<td>43</td>
<td>2468</td>
<td>44</td>
<td>5</td>
</tr>
<tr>
<td>48</td>
<td>21</td>
<td>52</td>
<td>3</td>
<td>53</td>
<td>5</td>
</tr>
<tr>
<td>54</td>
<td>2</td>
<td>55</td>
<td>2471</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MC/OS roff: 24 contexts

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>286938</td>
<td>11</td>
<td>105612</td>
<td>16</td>
<td>4900</td>
</tr>
<tr>
<td>17</td>
<td>2676</td>
<td>18</td>
<td>92806</td>
<td>20</td>
<td>29647</td>
</tr>
<tr>
<td>21</td>
<td>410602</td>
<td>24</td>
<td>617</td>
<td>25</td>
<td>38972</td>
</tr>
<tr>
<td>26</td>
<td>18915571</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

by int:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>557</td>
<td>2</td>
<td>558</td>
<td>3</td>
<td>557</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>31</td>
<td>3</td>
<td>32</td>
<td>3</td>
</tr>
<tr>
<td>33</td>
<td>4</td>
<td>38</td>
<td>1</td>
<td>39</td>
<td>2</td>
</tr>
</tbody>
</table>
42 1
43 2410
44 5
48 27
52 3
53 5
54 2
55 2414

MC/OS sort: 6 contexts
total ticks 4291, lost 52 instr 9263135, ints 7312
by proc_nr:
  0  217492
 11 104882
 16  77579
 17  5000
 18 164142
 20  42993
 21  686101
 24  502
 25 22164
 26  7942280
by int:
  1  814
  2  818
  3  816
 30  58
 31  58
 32  58
 33  17
 35 275
 36  56
 38  1
 39 1286
43 1286
 44  60
 48 292
 52  59
 53  60
 54  58
 55 1240

MC/OS sort: 16 contexts
total ticks 4427, lost 64 instr 9233916, ints 6252
by proc_nr:
  0  189642
 11 104395
 16  75662
17 5301
18 164343
20 42017
21 687495
24 617
25 22164
26 7942280
by int:
  1  814
  2  816
  3  814
  30 3
  31 3
  32 3
  33 6
  38 1
  39 1250
  42 1
  43 1250
  44 5
  48 25
  52 3
  53 5
  54 2
  55 1251

MC/OS sort: 24 contexts
total ticks 4423, lost 41 instr 9177294, ints 4876
by proc_nr:
  0 155171
  9  693
 11 100361
 16  74614
 17  3045
 18 162573
 20  39012
 21  677381
 25 22164
 26 7942280
by int:
  1  787
  2  790
  3  788
  30 2
  31 2
  32 2
  33 4
  38 1
39 2
43 1238
44 4
48 10
52 2
53 4
54 2
55 1238

MC/OS compress: 6 contexts
total ticks 5634, lost 57 instr 11998584, ints 6804
by proc_nr:
0 250317
9 3057
11 76821
16 3479
17 3083
18 95591
20 40031
21 296839
24 572
25 21262
26 11207532
by int:
1 534
2 536
3 534
30 36
31 36
32 36
33 40
35 169
36 34
38 1
39 1515
42 1
43 1515
44 38
48 184
52 38
53 38
54 37
55 1482

MC/OS compress: 16 contexts
total ticks 5810, lost 73 instr 11953379, ints 6111
by proc_nr:
0 222708
9 3057
11 75709
16 2431
17 840
18 93336
20 33086
21 293418
25 21262
26 11207532
by int:
1 507
2 509
3 509
30 2
31 2
32 2
33 6
38 1
39 1518
43 1518
44 4
48 8
52 2
53 4
54 2
55 1517

MC/OS compress: 24 contexts
total ticks 5795, lost 47 instr 11925963, ints 4694
by proc_nr:
0 181876
9 3057
11 76821
16 3479
17 2679
18 94938
20 36091
21 297611
24 617
25 21262
26 11207532
by int:
1 531
2 532
3 531
30 3
31 3
MC/OS concurrent: 6 contexts
total ticks 5634, lost 36 instr 40653184, ints 26756
by proc_nr:
0 905949
9 3750
11 323985
16 82993
17 8400
18 92806
20 104750
21 1385594
24 1234
25 82395
26 18915571
27 7942280
28 11207532
by int:
1 2570
2 2573
3 2571
30 128
31 128
32 128
33 72
35 453
36 83
38 3
39 5423
42 2
43 5422
44 114
48 799
52 167
53 155
54 96
MC/OS concurrent: 16 contexts
total ticks 16211, lost 49 instr 40759464, ints 26436
by proc_nr:
0 901810
9 7142
11 291643
16 105637
17 9972
18 435850
20 172380
21 1657795
22 850
23 2079
24 1234
25 18956627
26 70657
27 7948820
28 10226988
by int:
1 2577
2 2588
3 2584
30 14
31 14
32 14
33 50
35 30
36 2
38 3
39 6188
42 2
43 6188
44 26
48 116
52 16
53 26
54 14
55 6124

MC/OS concurrent: 24 contexts
total ticks 13588, lost 36 instr 40991598, ints 16125
by proc_nr:
0 623985
9 3750
11 623985
16 82993
MC/OS I/O Intensive Results: This experiment, as in the Minix test, involved running the file system consistency check, fsck, on the main file system.
21 6218745
22 10703
23 1362899
24 14206481
25 140464
26 9770
by int:
1 14565
2 14606
3 14585
30 3624
31 3624
32 3624
34 22266
35 10989
36 3663
38 1
43 11161
44 3666
48 32338
52 3669
53 3666
54 3663
55 7176
MC/OS I/O Intensive: 5 contexts
total ins 29301783, ints 105469
by proc_nr:
  0  2447869
  9  428645
 10  15092
 11  1883147
 12   588
 14  1355
 15  135
 16  447982
 17  13033
 18  1655276
 20  372667
 21  6294823
 22  10703
 23  1360347
 24  14219887
 25  140464
 26   9770
by int:
  1  14647
  2  14689
  3  14673
 30  1000
 31  1000
 32  1000
 33  5317
 35  4060
 36  1015
 38  1
 43  8389
 44  1043
 48  28506
 52  1046
 53  1043
 54  1039
 55  7001
MC/OS I/O Intensive: 6 contexts
total ins 29586996, ints 98882
by proc_nr:
<table>
<thead>
<tr>
<th>proc_nr</th>
<th>ins</th>
<th>ints</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2734284</td>
<td>2734284</td>
</tr>
<tr>
<td>9</td>
<td>428645</td>
<td>428645</td>
</tr>
<tr>
<td>10</td>
<td>15092</td>
<td>15092</td>
</tr>
<tr>
<td>11</td>
<td>1883147</td>
<td>1883147</td>
</tr>
<tr>
<td>12</td>
<td>588</td>
<td>588</td>
</tr>
<tr>
<td>14</td>
<td>643</td>
<td>643</td>
</tr>
<tr>
<td>15</td>
<td>135</td>
<td>135</td>
</tr>
<tr>
<td>16</td>
<td>447982</td>
<td>447982</td>
</tr>
<tr>
<td>17</td>
<td>13033</td>
<td>13033</td>
</tr>
<tr>
<td>18</td>
<td>1654419</td>
<td>1654419</td>
</tr>
<tr>
<td>20</td>
<td>372484</td>
<td>372484</td>
</tr>
<tr>
<td>21</td>
<td>6295373</td>
<td>6295373</td>
</tr>
<tr>
<td>22</td>
<td>10703</td>
<td>10703</td>
</tr>
<tr>
<td>23</td>
<td>1360347</td>
<td>1360347</td>
</tr>
<tr>
<td>24</td>
<td>14219887</td>
<td>14219887</td>
</tr>
<tr>
<td>25</td>
<td>140464</td>
<td>140464</td>
</tr>
<tr>
<td>26</td>
<td>9770</td>
<td>9770</td>
</tr>
</tbody>
</table>

by int:
<table>
<thead>
<tr>
<th>int</th>
<th>ins</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14645</td>
</tr>
<tr>
<td>2</td>
<td>14686</td>
</tr>
<tr>
<td>3</td>
<td>14671</td>
</tr>
<tr>
<td>30</td>
<td>316</td>
</tr>
<tr>
<td>31</td>
<td>316</td>
</tr>
<tr>
<td>32</td>
<td>316</td>
</tr>
<tr>
<td>34</td>
<td>6704</td>
</tr>
<tr>
<td>35</td>
<td>1585</td>
</tr>
<tr>
<td>36</td>
<td>317</td>
</tr>
<tr>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td>43</td>
<td>9018</td>
</tr>
<tr>
<td>44</td>
<td>359</td>
</tr>
<tr>
<td>48</td>
<td>28087</td>
</tr>
<tr>
<td>52</td>
<td>348</td>
</tr>
<tr>
<td>53</td>
<td>359</td>
</tr>
<tr>
<td>54</td>
<td>344</td>
</tr>
<tr>
<td>55</td>
<td>6810</td>
</tr>
</tbody>
</table>
MC/OS I/O Intensive: 8 contexts
total ins 28650417, ints 86672
by proc_nr:
0 1769858
9 438893
10 15092
11 1883147
12 588
14 3755
15 135
16 447982
17 13033
18 1669194
20 372495
21 6295074
22 10703
23 1360347
24 14219887
25 140464
26 9770
by int:
1 14650
2 14692
3 14671
30 128
31 128
32 128
34 1479
35 777
36 111
38 1
43 6946
44 171
48 25803
52 142
53 171
54 137
55 6537
MC/OS I/O Intensive: 10 contexts
total ins 28678483, ints 88871
by proc_nr:
0 1805928
9 428830
10 15172
11 1883147
12 588
14 643
15 135
16 448230
17 13138
18 1673163
20 372267
21 6296071
22 10703
23 1360347
24 14219887
25 140464
26 9770
by int:
1 14646
2 14687
3 14672
30 42
31 42
32 42
34 1250
35 162
36 18
38 1
43 7175
44 85
48 29212
52 49
53 85
54 45
55 6658
MC/OS I/O Intensive: 16 contexts
total ins 27967165, ints 66830
by proc_nr:

<table>
<thead>
<tr>
<th>proc_nr</th>
<th>ins</th>
<th>ints</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1110621</td>
<td>9429193</td>
</tr>
<tr>
<td>9</td>
<td>15172</td>
<td>588</td>
</tr>
<tr>
<td>10</td>
<td>1883147</td>
<td>821</td>
</tr>
<tr>
<td>11</td>
<td>588</td>
<td>135</td>
</tr>
<tr>
<td>12</td>
<td>447982</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>13033</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1657615</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>372340</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>6295431</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>10703</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>1360347</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>14219887</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>140380</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>9770</td>
<td></td>
</tr>
</tbody>
</table>

by int:

<table>
<thead>
<tr>
<th>int</th>
<th>ins</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14648</td>
</tr>
<tr>
<td>2</td>
<td>14688</td>
</tr>
<tr>
<td>3</td>
<td>14670</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>31</td>
</tr>
<tr>
<td>34</td>
<td>61</td>
</tr>
<tr>
<td>35</td>
<td>15</td>
</tr>
<tr>
<td>36</td>
<td>1</td>
</tr>
<tr>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td>43</td>
<td>14007</td>
</tr>
<tr>
<td>44</td>
<td>74</td>
</tr>
<tr>
<td>48</td>
<td>1660</td>
</tr>
<tr>
<td>52</td>
<td>32</td>
</tr>
<tr>
<td>53</td>
<td>74</td>
</tr>
<tr>
<td>54</td>
<td>28</td>
</tr>
<tr>
<td>55</td>
<td>6778</td>
</tr>
</tbody>
</table>
III.4 Round Robin Scheduling Results

Round robin (RR) scheduling was compared against IPC directed scheduling for a number of applications. Results were discussed in Chapter 7, Section 4. Numerical values were also given in Table XIV. The Round robin experimental results used to construct Table XIV are given below.
MC/OS (RR) Distributed: 6 contexts
total ins 7716265, ints 37560
by proc_nr:
0 1022215
9 366023
10 15092
11 470063
12 588
14 821
15 135
16 409503
17 11038
18 507086
20 304251
21 3042210
22 10703
23 1286960
24 119343
25 140464
26 9770
by int:
1 4533
2 4568
3 4554
30 335
31 335
32 335
34 2927
35 1700
36 340
38 1
43 2883
44 372
48 11611
52 365
53 372
54 361
55 1968

MC/OS (RR) Distributed: 24 contexts
total ins 7146096, ints 24753
by proc_nr:
0 448462
9 366694
10 15172
11 470063
12 588
14 643
15 135
16 409503
17 11038
18 509793
20 304289
21 3042476
22 10703
23 1286960
24 119343
25 140464
26 9770
by int:
1 4537
2 4573
3 4554
30 25
31 25
32 25
34 50
38 1
43 8927
44 62
48 1220
52 25
53 62
54 21
55 646

MC/OS (RR) roff: 6 contexts
total ins 20299196, ints 18737
by proc_nr:
0 645281
9 693
11 100780
16 1935
17 1182
MC/OS (RR) roff: 24 contexts
total ins 19931032, ints 7295
by proc_nr:
  0 317272
  11 111748
  16 4900
  17 2991
  18 92870
  20 30686
  21 415405
  24 617
  25 38972
  26 18915571
by int:
  1 562
  2 564
  3 562
  30 3
MC/OS (RR) sort: 6 contexts
total ins 9428926, ints 12646
by proc_nr:
0 353154
11 109011
16 75396
17 3045
18 162463
20 37973
21 723440
25 22164
26 7942280
by int:
1 789
2 791
3 791
30 41
31 41
32 41
33 132
35 195
36 39
38 1
43 1470
44 43
48 6706
52 41
53 43
54 41
55 1441
MC/OS (RR) sort: 24 contexts
total ins 9201154, ints 5249
by proc_nr:
0 164194
11 102004
16 75662
17 5196
18 164323
20 42017
21 682697
24 617
25 22164
26 7942280
by int:
1 812
2 814
3 812
30 3
31 3
32 3
33 4
38 1
42 1
43 1366
44 5
48 47
52 3
53 5
54 2
55 1368

MC/OS (RR) compress: 6 contexts
total ins 12132750, ints 14010
by proc_nr:
0 416473
9 3057
11 67478
16 2431
17 840
18 93142
20 33086
21 287449
25 21262
26 11207532
by int:
1 507
2 509
3 509
30 24
31 24
32 24
33 112
34 1
35 110
36 22
38 1
43 1769
44 26
48 8547
52 24
53 26
54 24
55 1751

MC/OS (RR) compress: 24 contexts
total ins 11940546, ints 5135
by proc_nr:
0 199108
9 3057
11 72937
16 3479
17 2886
18 95502
20 37206
21 296960
24 617
25 21262
26 11207532
by int:
1 534
2 536
3 534
30 3
31 3
32 3
33 4
38 1
42 1
43 1726
44 5
48 47
52 3
53 5
54 2
55 1728

MC/OS (RR) I/O Intensive: 6 contexts
total ins 29547557, ints 44090
by proc_nr:
0 863408
9 391707
11 234443
16 4918
17 11140
18 1641756
20 54858
21 23743518
24 572
25 7917
26 98344
27 2494976
by int:
1 7856
2 7859
3 7856
30 168
31 168
32 168
33 524
34 23
35 835
36 167
38 1
42 1
43 4541
44 171
48 8842
52 171
53 171
MC/OS (RR) I/O Intensive: 24 contexts
This set of results is reduced by figures for distributed application, round robin, when added to Table XIV because the fsck operation was run immediately after system initialization, and the counters were not cleared until following the fsck.
total ins 36071642, ints 58886
by proc_nr:
<table>
<thead>
<tr>
<th>proc_nr</th>
<th>ins</th>
<th>ints</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>962093</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>763226</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>15172</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>690747</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>588</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>2954</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>412342</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>23635</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>2096733</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>321506</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>26707692</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>10703</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>1289421</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>120577</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>159066</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>2495052</td>
<td></td>
</tr>
</tbody>
</table>
by int:
<table>
<thead>
<tr>
<th>int</th>
<th>ins</th>
<th>ints</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12057</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12094</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12065</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>15416</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>533</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>6400</td>
<td></td>
</tr>
</tbody>
</table>