Chapter 1

MULTI-TASKING SYSTEMS - I: RYMOS

In the previous sections, we have examined how high-level languages are compiled down to machine language. And, since practically all applications for 32-bit microprocessors are written in HLLs, it is important to know how well the processor architecture supports the data types and kind of stack-based addressing required by HLLs in evaluating different microprocessors.

We have also seen how a language like C can give us some of the advantages of a HLL and also offer us access to machine features usually only accessible with assembly language.

However, before looking at additional architectural and hardware features found in 32-bit machines (and not found in 8-bit machines), we have to address another "high-level" question: *the operating system environment* advanced processors work under.

In a very broad sense, computers can be used in two types of applications:

1. Computationally intensive applications requiring vast number-crunching capability;

2. Performing various tasks in response to real-time events such as interrupts from a keyboard, printer buffer, disk controller or various sensors in a control environment.

In the first case—which includes problems like weather forecasting on a CRAY supercomputer— the processor is completely occupied for considerable lengths of time in solving the problem. (Note "considerable time" may

mean anything from 10 seconds to several hours.) These applications can be solved in *batched* mode, running each job until completion and then loading and running another job. This straightforward approach maximizes the utilization of the processor.

In the second broad range of computer applications, however, the situation is quite different. Suppose, for example, that a computer is being used as a text editor. Each keystroke will generate an interrupt and the text editor will process the next character. Assuming that processing a character requires an average of 100 machine instructions, that the operator types at 50 words per minute (or about 5 characters per second) and that the computer operates at 100 MIPS, then the character processing program will be invoked every 20 ms and run for 1 μsec. In other words, the computer will be only be used 0.005% of the time! This is not a cost-effective use of such a powerful processor.

With so much idle time available, the processor's resources should be exploited to perform other tasks. You have already seen a very rudimentary way of accomplishing a kind of *multi-tasking* in the third year course using interrupt-driven code. This is an extremely limited and inflexible way of achieving "multi-tasking", however, and we shall examine a much better way—using a multi-taking operating system—in this chapter.

We shall see later that multi-tasking, real-time operating systems offer more than mere switching between tasks. They can substantially reduce software complexity in, for example, industrial or process control applications with their inter-process communication features. These features also make it easier to divide a large programming project among independent groups who need only agree on the protocol of "inter-task communication" rather than on the details of of implementation.

We shall develop our own tiny multi-tasking system, *RYMOS* (Ryerson Multi-tasking Operating System), to be run on the 68000 SBC. This will bring out the principles involved in such systems and the architectural support needed to implement them. And although our system will work, we shall encounter a "brick wall" when it comes to making the system absolutely secure. In particular, we will see that the 68000 (and other machines of the same generation such as the 8086 as well as most small microcontrollers such as the 6809 or 68hc11) lack essential memory management hardware to make a secure system. Furthermore, we shall see that when tasks may be invoked unpredictably, hardware for virtual memory support is required for efficient,

cost-effective operation[1].

Operating system theory, like another subset to this course— compilers, easily merits one or two courses all by itself. The objective here, however, is not to look at *all* the aspects of operating system theory; rather, we give an overview of a the various components of an operating system and then concentrate on those features that are closely related to architectural and hardware features of advanced microprocessors. For example, when we look at memory management, our final concern is how hardware features to implement segmented or virtual-paged memory can vastly enhance an operating system's performance and capabilities; but we leave aside such issues as the Deutsch-Schorr algorithm for efficient garbage collection and compaction which is purely a software performance issue[2].

We approach multi-tasking operating systems as follows:

- First, we develop a very simple operating system, *RYMOS*, with a minimum of fancy features. Our primary focus here is on the *context switch*, the routine that actually performs the change from one process to another in the the processor. Later we shall examine how some advanced microprocessors simplify this task.

- Next we extend our overview of *RYMOS* to include some generic interprocess communication facilities common in commercial operating systems. At the same time, we introduce the specific features of *real-time* operating systems so important in industrial and process control applications.

- We then examine some popular commercial real-time multi-tasking operating systems.

[1]Note that, with additional memory-management hardware, the 68000-based multitasking system can be made secure while the 8086 is inherently insecure. For true virtual memory support, the 68000 processor hardware in insufficient.

[2]Nor do we consider such traditional subjects in operating system theory as file structures. At the hardware level, the performance of file systems depends on the maximum data-transfer rate on the bus, the amount of hardware caching of the disk, etc. These considerations are essentially independent of the software used to access disk files insofar as a bad software file organization will will be simple "less bad" with good hardware whereas good hardware performance will enhance better software techniques. Of course, if and when it becomes common (and it is already done on some LISP machines) to implement such algorithms in hardware, you will have to learn about them.

1.1 Multi-tasking: An overview

We first consider the limitations of interrupt-driven code. If we have several tasks to run and each is invoked by an interrupt, we can simply make the Interrupt Service Routine be the task for the corresponding interrupt. By assigning priorities to the interrupts, we can control the relative importance of tasks.

This approach lacks flexibility, however. For example, it is extremely difficult to change priorities on the fly. The source of this problem is the way in which a tasks state is saved when an interrupt occurs. As tasks are suspended by higher-priority interrupts, their state is saved on the stack. Normally, when a higher-level task is completed, control is returned to the next higher task with a Return from Interrupt instruction which returns control to the task whose state has been saved on top of the stack. If priorities change, however, the pending task with the highest priority may no longer be on top of the stack. We would have to determine where it was on the stack, and swap its state with the state of the task (of now lower priority) currently on top of the stack. This is unacceptably complex and inefficient.

Additional problems arise when different tasks use the same interrupt. Consider the problems involved in the following very simple multi-tasking situation. Suppose that we have two equal priority tasks that never have to wait for I/O and we wish to switch between the two every 10 msec. We can use a timer chip to generate an interrupt every 10 msec and then swap the tasks in the interrupt service routine. If task 1 is running, then the task 1 state will be placed on the stack when the interrupt occurs (the contents of the registers, the value of the PC where it was interrupted). We now want to continue with Task 2 (which presumably was running 10 msec earlier). Somehow or other we reset the registers to their values when Task 2 was last executing and then transfer control to it. However, before doing so, we must pop the Task-1 state information off the stack and save it somewhere. Otherwise, when Task-2 resumes the stack will not be the same as it was when it was executing previously.

This is much more complex than elementary interrupt-driven code and the proper method for solving it is not to hack away at various tricks; rather a multi-tasking operating system is required.

In a multi-tasking operating system, we want to be able to write each task as if it had a microprocessor all to itself (if it did it would be called a

multi-processing environment). In essence, a multi-tasking operating system simulates a multi-processor environment (one processor per task) in software.

To simulate a multi-processor environment, the single processor must make a model of all of the other processors. We call the single processor the *real machine* and all of the modelled processors the *virtual machines*. Each of the tasks to be run is assigned to a separate processor. One task will be loaded into the real processor while all the other ones are loaded into virtual processors. Since the real processor is the only one that can actually do any work, only the task loaded on it will run. The job of the operating system is to swap tasks between the real processor and one of the virtual processors in response to real-time events.

To see how this is done, consider a processor as a gigantic state machine. At any point in time, this "gigantic state machine" is in a unique state determined by the contents of all registers and all memory. If we duplicated this state in distinct, but identical, hardware and started it running the task would continue as before, albeit on a different machine.

The trick to obtaining multi-tasking is to switch the real processor from a state corresponding to one task to that corresponding to another on a virtual machine. In theory, switching the state involves changing both memory and registers. Changing registers should be easy; the difficult (or at least time-consuming) part seems to be changing memory. We shall see later that memory mangement hardware makes this quite simple and secure. Lacking such hardware, we make some assumptions.

First, we assume that each task has program code in non-overlapping regions of memory. Hence no swapping of program code memory is required, switching the program counter will suffice. Next, we assume that each process can dynamically obtain parcels of memory (that will not be used by any other processes) by using the operating system's memory allocation routines. The operating system ensures that such memory blocks do not overlap (through software checking) and hence no switching of this memory is required. Each task may also require some memory for global variables. Again, such memory may be allocated at load time by the operating system such that it does not overlap with any other task's memory space. Finally, each task requires its own stack space. Again non-overlapping stack space can be allocated at load time and switching from one task's stack to another is done by simply changing the stack pointer.

In short, we divide real memory amongst the various virtual machines

so that the memory allocated to one machine does not overlap with that of another. Hence the only things we need save to define the state of a machine are the contents of registers. By enforcing the above rules on memory usage, we can switch the processor state from one process to another merely by saving the current value of the registers (corresponding to the state of the process that is about to be suspended) and setting each of the registers to the values saved the last time the new process was running.

The routine that actually switches the computer from one task to another is called the *context switch*. We will examine how it works in detail later on and you will have to write a context switch routine as one of your projects.

A multi-tasking operating system needs much more than a context switch, however. It needs routines to determine what task to switch to, to allocate memory, to bootstrap the system, to allow intertask communication, etc.

Many of these routines will need some global information about the various tasks on the system. One of the important pieces of information is a process's *state*.

At any given time exactly one task will be running and we say that the state of this task is *current*. Other tasks may be in one of several states: they may be *ready* in which case there is nothing to prevent them from executing other than the fact that some other task is current; they may be *waiting* for some external event before they can become "ready" (let alone current); they may be *suspended* in which case they will not become "ready" again until some other process explicitly "unsuspends" them; or they may be *sleeping* until some predetermined time elapses. There may be several other kinds of states depending on the particular multi-tasking operating system. Often, for example, there are various special states associated with inter-process communication.

The state of a process and other vital information about each task is maintained in a *process table*. The operating system *kernel*[3] routines will use the information in the process table in deciding what to do and ensure that the information is kept up-to-date and accurate.

One of the central routines in a multi-tasking operating system is the task *scheduler* or *dispatcher*. It may be invoked in a variety of ways including interrupts or explicit calls and its job is to select a task from among all those

[3]The kernel is the set of primitive routines and data structures that implement the operating system. The kernel does not include utility programs, compilers, loaders, etc.

that are ready to become current. The scheduler does not actually make that task current; the context switch does that but the scheduler is the context switch's "boss".

As an absolute minimum, a multi-tasking operaing system needs the following routines and data structures:

- A *process table* containing global information about the tasks in the system;
- A *scheduler* to determine which ready task to make current;
- A *context switch* to perform the actual switching from one task to another;
- *Memory management* routines to allocate and release memory.
- A *bootstrap* routine to bring the system to life;
- Various miscellaneous routines such as interrupt handlers, peripheral device initialization routines, etc.

With these routines we can put together a rudimentary kernel. Note that several useful features are missing from this list. For example, there are no routines to create or destroy tasks. In our initial, primitive *RYMOS* we shall dispense with such apparent necessities by creating immortal processes directly within the bootstrap routine. (We make them immortal to avoid the need of a routine to kill a task). You will, however, have to add create and kill routines as part of your first project.

RYMOS is strongly patterned after the XINU operating system described in Douglas Comer's book *Operating Systems: The XINU Approach* published by Prentice-Hall. This is an excellent book if you want to explore operating systems in greater depth than offered in this course. (Note that XINU is a [recursive] acronym for "XINU Is Not Unix".)

We now consider each of these items for primitive *RYMOS*.

1.2 Primitive RYMOS

1.2.1 The Process Table

Any multi-tasking operating system must have a process table where it maintains the information about a task required to schedule and re-start it. Some of the information in the process table can often be examined by the user with the *process status* command. In UNIX, for example, the ps command gives this information. Running ps from within this editing session, for example, produces the following:

```
      F UID   PID  PPID CP PRI NI  SZ  RSS WCHAN    STAT TT  TIME COMMAND
   80003   0     0     0  0 -25  0   0    0 runout   D    ?   6:28 swapper
20088000   0     1     0  0   5  0  52    0 child    IW   ?   0:05 /sbin/init -
   80003   0     2     0  0 -24  0   0    0 child    D    ?   2:15 pagedaemon
   88000   0    44     1  0   1  0  68    0 select   IW   ?   0:19 portmap
20088000   0   126     1  1   3  0  40    0 Sysbase  IW   co  0:00 - std.9600 c
a0488201 142 17071 17061  1  15  0  80  240 kernelma S    p0  0:00 -sh (csh)
20000001 142 17100 17071 27  31  0 152  384          R    p0  0:00 ps -agxl
a0488200 142 17070 17062  0  15  0  96    0 kernelma IW   p1  0:00 -sh (csh)
20008000 142 17094 17070  0   3  0 192    0 Sysbase  IW   p1  0:03 vi hll.tex r
```

This information indicates for example that the *ps* program is currently running (obviously!) and that all the other tasks are waiting. We illustrate this not to examine the type of process table maintained by UNIX (we will do that later in the course) but rather to show what kind of information about tasks is kept in process tables by a typical operating system.

The simplest form of process table is an array of records where each record contains global information about a process. More complex forms use the per process stack(s) and per process global memory in addition to the basic array of records. There are definite advantages to these more complex organizations, especially when the processor supports virtual memory and has separate user and kernel stacks; however, RYMOS uses the simple "array of records" process table format.

The process table contains the following information about each process:

Registers: The contents of all the registers the last time the task was running (or the initial values if it has not yet run). The context switch is

responsible for updating these values when it switches to a new task as well as making the machine registers equal to the stored values for the new task it is switching to.

State: The state of the task. In primitive *RYMOS* we consider only three states: *current* when the task is running; *ready* if it is ready to run; and *free* if the entry in the process table is not in use (i.e. contains no information related to any active task). The scheduler is responsible for changing the state information from current to ready when it stops a task and for setting the state of the new task to current. When you add a routine to create new tasks dynamically, it is useful to add a new state, SUSPENDED, as well. When a process is created, its state is set to suspended. It can then be "unsuspended" when the programmer decides it is time to make it ready for execution.

Priority: This number indicates the relative importance of the task. When the scheduler selects a new task to make current, it will select the ready one with the highest priority. Note that the priority can be changed during the task's lifetime. Initially, we will run all processes at the same priority.

Stack area: These fields give the size of the stack and its starting address. When a task is created this area is allocated by the system memory allocator. When the task dies, this information is used to release the stack area back to the pool of free memory.

ID Number (PID): The PID uniquely identifies tasks. It is assigned at task creation time and the number is not re-used until the global "PID counter" turns over (i.e. > 65,000 processes later with a 16-bit counter or > 4,000,000,000 processes later with a 32-bit counter).

Name: This "user-friendly" task identifier need not be unique. It is used only as a convenience to the user.

Starting Address: This records the starting address of the routine.

Note that not all of the information is absolutely necessary. For example, the starting address is not really required since the "old PC" must be set to this address when the process is created and it is only when the process starts

running the first time that this address is of any importance. Nonetheless keeping this kind of information along with the task's name can be helpful in debugging and in supplying snapshot information about processor activity to a user.

The precise format of a process table entry is shown below.

```
/*
*    The proc.h include file defines the structure of process entries in the
*    RYMOS process table.
*/
typedef struct process { /* type definition for entry in process table */
    long regs[17];    /* prev. register values d0-d7, a0-a6, SP, PC */
    int cond_reg;    /* condition code/status register */
    int *initstk;    /* initial value of stack pointer */
    int stksize;    /* size of stack area */
    int *text;        /* initial value of program counter */
    short priority; /* process priority */
    short state;    /* state of process (FREE, ReaDY, or CURrent */
    struct process *pnext;    /* pointer to next process in ready list */
    short pid;        /* process id number */
    char name[8];    /* process name */
} PROCESS, *PROCPTR;

#define SP 15
#define PC 16
```

1.2.2 The Task Scheduler

The scheduler must select the highest priority task among all the those that are ready and make it current. To make it current, the state field in the process table entry has to be updated; then the context switch is called to switch the tasks.

The scheduler must also behave correctly when the current task has a higher priority than any ready task. In this case, the current task should be allowed to continue.

In primitive *RYMOS*, the scheduler is usually called when the time-slice allocated for the current process has elapsed. However, the scheduler can

also be called in other situations. In our initial, bare-bones *RYMOS* the scheduler is also called from the bootstrap routine to start up the very first task. In this case, there is no current task (i.e. the "task" that called the scheduler is non-existent) and the scheduler should not set the state of this "non-existent task" to ready! When a process is killed, the last thing it will do is to call the scheduler to select a new task. This is another example of the calling task not really being the current task. In later versions of *RYMOS*, we shall encounter other situations where the calling task is not still current and should not be made ready.

The scheduler, of course, should operate quickly and, since selecting the highest priority ready task is its main function, the way the selection is done should be looked at closely.

The most simple-minded way of selecting the highest priority ready task is to step though each entry in the process table looking for it. If the process table had 100 entries, however, this simplistic approach would take a lot of time.

A better method is to link together all the process table entries for ready tasks in a sorted linked list with the highest priority task at the beginning of the list. If such a list is maintained, the scheduler simply deletes the first task on the list and makes it current.[4].

In order to maintain such a list we need link fields in the process table entry structure and primitive routines to insert and delete elements from the list. The required link field is the:

struct*process *pnext;*

The utility routines *rdy_del* and *rdy_ins* are used to delete and insert routines respectively in the ready list. Note that the *rdy_del* routine returns a pointer to the process table entry that was deleted—i.e. a pointer to the task that is about to be made current. Hence the selection function of the scheduler can be done with one line of C code:

[4]This "better method" is in fact the most "simple-minded" approach to organizing a *priority queue*. While a linked-list implementation of a priority queue is appropriate for a small number of items (e.g. number of tasks in our system), other approaches (such as tree structures) perform better for insertions. We do not consider such issues in this course; they are the proper domain of a computer science course on operating systems or data structures.)

curproc = rdy_del();

The following pseudo-code gives a formal description of the scheduler.

if the last running process is still in CURRENT state **then**
  if the current process's priority > highest priority ready task **then**
    return *Just keep it current*
  else
    Set state of current process to READY
    Insert it in READY LIST
  endif
endif
old_current_process := current_process
new_current_process := deleted highest priority task on "ready list"
set the state of that task to CURRENT
call context_switch(old_current_process, new_current_process)
return

The C code for the scheduler is shown below. below. (Note that certain features of the pseudo-code have not been implemented; you will have to do this as part of an assignment):

```
/*    YOU HAVE TO MODIFY THIS ROUTINE SO THAT IT
*    RETURNS IMMEDIATELY IF THE CALLING PROCESS IS CURRENT
*    AND HAS A HIGHER PRIOIRTY THAN ANYTHING ON THE READY LIST
*/

#include "magic.h"
#include "proc.h"

extern PROCPTR curproc;
extern PROCPTR rdyhead;    /* Pointer to ready list of processes */
extern PROCESS proctab[NPROCS];
extern int dummy(), ctxsw();
resched()
{
    PROCPTR oldproc;

    if (curproc->state == CURRENT) { /* Is "current process REALLY current */
```

```
        curproc->state = READY;
        rdy_ins(curproc);
    }
    /* Force context switch to first on ready list */
    oldproc = curproc;
    curproc = rdy_del();
    curproc->state = CURRENT;
    ctxsw(oldproc->regs, curproc->regs);
}
```

1.2.3 The Context Switch

The context switch (*ctxsw*) simply saves the current values of registers in the save area for the process that is being suspended and re-initializes the registers with the values from the save area of the process being switched to.

Thus, *D0* would, for example, first be saved in the area reserved for it in the old task's process table entry. Then it would be loaded with the value it used to have the last time the new process was executing; this value is obtained of course from the save area reserved for that task.

This all looks extremely trivial! It may, perhaps, be "trivial", but it is also quite subtle.[5] Although the problem is "trivial" for most of the registers, things get tricky when it comes to the Program Counter and Stack Pointer.

The Stack Pointer is the simpler one. We just have to make sure that we don't switch to the new stack until we're finished with the old one.

But what value should be saved for the program counter? After all, the current value of the program counter changes with every instruction. Before we entered the context switch, we were in the environment of the task that is being switched out. As each instruction within the context switch is executed, we get ever closer to the environment of the task we're switching to. In this ambiguous never-never land between two tasks with a continually changing PC , what is the current value of the program counter of the task

[5]One has to be careful about how the register contents are saved. The addresses of the register save areas are passed to the context switch on the stack; i.e. these addresses are in memory. In order to move the contents of any register to the proper area, the save area pointer has to be loaded into an address register. Doing this, of course, destroys the previous contents of the address register before we we have the means of storing it in its save area. Proper use of the stack for temporary storage can avoid this problem.

being suspended?

To answer this question, we step back from the inner workings of the context switch and recall that the scheduler called the context switch. The scheduler was running in the pristine environment of the task now being suspended. The call to the context switch within the scheduler, however, is written in the same way as any other call to a function in C. In particular, the scheduler expects that the context switch function call will return. We know of that it doesn't return at all (at least not in the normal way); rather it starts up a new task! But eventually there will be some future context switch back to the task that was suspended. If we restart the task *as if it had just returned from ctxsw* the original task that was suspended and is now being restarted will have no hint of the shenanigans that went on within the context switch. As far as it is concerned, the context switch will have behaved itself and returned!

So the saved value of the suspended task will simply be the return address to *resched* and this can be taken off the stack (before switching stacks, of course).

A final consideration is how to change the program counter to that of the new current task. We can get the right value from the save area but we can't simply move it into the PC since there is no instruction for that. The only instructions that can change the PC to arbitrary values are the (conditional) jump and branches, jump to subroutine, and return from subroutine or interrupt instructions. Since changing the PC must be the last register that is set to the new environment and since we want *ctxsw* to look like a "normal" subroutine that ends with an tt rts, we use the return from subroutine instruction to change the PC. Hence we must get the old PC for the task that is about to become current and put it on the new stack prior to executing the tt rts.

In primitive *RYMOS*, tasks are switched at interrupt time. Since all tasks have the same priority, this means that the processor time slices between all the tasks with each one receiving identical treatment.

Consider the following switching between 3 tasks. Assume that Task-1 is running when the first interrupt occurs. This causes the PC and status register to be saved on the Task-1's stack and the timer interrupt service routine is entered in Task-1's environment. Then the ISR re-enables interrupts and calls *resched* which in turn calls *ctxsw*. Task-2 is then initiated. It runs with its own stack and Task-1's stack is frozen. Note that there has been

no Return from Interrupt for the first interrupt that resulted in switching to Task-2 yet. Eventually Task-2 is interrupted and a similar sequence results in a switch to Task-3. Now two interrupts have occurred without any Return from interrupt instructions. We know that interrupts cause the stack to grow and that we must eventually return from interrupt to avoid stack overflow. There is no danger of that happening here, however, since each interrupt has happened with a *different stack*.

A third interrupt results in a context switch from Task-3 back to Task-1. Task-1 picks up in the kernel's *resched* routine as if it had just returned from *ctxsw*. *Reshed* returns to its caller, the timer ISR in this case. Finally, the timer ISR does execute a Return from Interrupt and the processor is back in Task-1's main routine in precisely the same condition it left when it was interrupted three timer tics earlier.

The astute reader may wonder why the PC has to be stored in the process table save area since it is already on that task's stack. Indeed, why not save all the registers on the task's stack and simply keep the SP in the process table? This makes the context switch easier to write and makes the process table smaller. This approach does create problems, however, when we move to a virtual memory system. In particular, if task overflows its stack area or generates a *page fault* when pushing something onto the stack, it will have to be suspended while the operating system fixes the problem. But where do we store the registers in such a case!

1.2.4 Memory Management

Requests for memory are honoured by the kernel's memory management routines by searching for some unused memory in the globally allocated *heap*. The management routines maintain a linked list of unused memory blocks within the heap. The list is kept in sorted order by increasing starting addresses of the blocks. Each block contains a link to the next block and the length of the block.

When the memory allocator receives a request for some number of bytes of memory, it marches through the list of free blocks until it finds one that is large enough to honour the request. It chops off the number of bytes requested from this block, rearranges the linked list of free blocks accordingly and returns the starting address of the block it found to the caller.

The structure of the linked list is defined in *memory.h* as shown in below:

```
/*
*    Memory.h is an include file defining the characteristics of the memory
*    heap used by the memory management routines in RYMOS.
*/

#define HEAPSIZE 1500    /* Size of heap in units of MBLCK (i.e. = 8 bytes) */

typedef struct mblock {    /* Structure of the free memory list */
    struct mblock *mnext;
    unsigned int mlen;
} MBLCK, *MBLKPTR;
```

The heap is initialized as a single block in the meminit routine shown below:

```
/* Meminit initializes the memory heap.
*
*    Arguments:
*        none
*
*    Returns:
*        nothing
*/

#include "magic.h"
#include "memory.h"

MBLCK memory[HEAPSIZE];    /* Allocate actual heap */
MBLCK memlist;    /* head of free memory list */

meminit()
{
    MBLKPTR p;
    printf("Initializing memory ...\n");
    memlist.mnext =  memory;
    memlist.mnext->mlen = HEAPSIZE*sizeof(MBLCK);
    memlist.mnext->mnext = 0;
}
```

The pseudo-code for the memory allocator is shown below.

getmem(n_bytes)
Routine to find n_bytes of memory and return the starting address
for each block on the free list
  if block is an exact fit for n_bytes **then**
    delete it from the free block list
    return(its starting address)
  else if block size > n_bytes **then**
      chop off extra bytes at end of block
      insert this chopped off block in free list
      return(starting address of block found)
    endif
  endif
endfor
if we don't return from within the for loop,
there are no free blocks large enough
to honour the request, so crash system
abort

The actual C code follows. Note the use of the comma operator in the *for* loop to obtain two loop variables.

```
/*
*    Gets a block of memory of a specified number of bytes and returns
*    its starting address.
*
*    Arguments:
*        nbytes - number of bytes to allocate
*
*    Returns:
*        pointer to start of block
*
*    BUGS:
*    The number of bytes requested must be a multiple of the size of
*    MBLCK (8 bytes at this time).
*    Just issues a message and stops if no more memory.
*/
```

```
#include "magic.h"
#include "memory.h"

extern MBLCK memlist;

int
*getmem(nbytes)
int nbytes;
{
    /* Adapted from Comer  */
    MBLKPTR q, p, leftover;

    for ( q = &memlist, p = memlist.mnext; p != NULL; q = p, p = p->mnext)
        if (p->mlen == nbytes)  { /* Exact fit? */
            q->mnext = p->mnext; /* Unlink it from free list */
            return( (int *) p);    /* Return its starting address */
        } else if (p->mlen > nbytes) { /* More than enough ? */
            /* Get address of leftover block */
            leftover = (MBLKPTR) ( (unsigned) p + nbytes);
            q->mnext = leftover;
            leftover->mnext = p->mnext;
            leftover->mlen = p->mlen - nbytes;
            return ( (int*) p);
        }
    printf("Out of memory\nABORTING\n");
    stop();
}
```

The pseudo-code for the memory release routine is given below:
freemem(starting_address, size)
Routine to free "size" bytes of memory
starting at "starting_address"
while bf not *at end of list* bf and *block_start* < *starting_address*
step to next block on list
endwhile
determine ending address of previous block
if new block abuts with end of previous block **then**

coalesce both blocks into a single bigger block
else
create a new block and insert after previous block
endif
if end of new block abuts with beginning of next block **then**
join them both into a single bigger block
endif
return
The actual C code for `freemem` is:
../kernel/src/freemem.c

1.2.5 Bootstrapping the System

The bootstrap routine is the first program to begin execution when *RYMOS* is loaded into the ECB. At this point there is no process table, no interrupt service routine vectors, no memory heap, etc. There is nothing but the raw 68000 machine and all the structures required for the successful operation of *RYMOS* have to be set up.

The bootstrap calls various routines to perform the necessary initialization and then puts some entries into the process table. It fills in the process table to run two tasks called "hello" and "bye". It also fills in the table for a third process, the null process. This task is an infinite loop that does absolutely nothing. We need it because the rest of the system is easier to write if we assume that there is always at least one process that can run and the null process is there to ensure that this assumption is always true. Hence the null process can never die, has the lowest possible priority so that it will only run if there is nothing else to run and has a smaller stack area than regular tasks so that it consumes the minimum of computer resources. Finally it calls *resched* and the real *RYMOS* takes over.

There is a problem in calling *resched* the first time. *Resched* assumes that *RYMOS* is already running but for the first call to *resched RYMOS* is in the final stages of being born. In particular, this call to *resched* must never return. We achieve this by telling *resched* that the "last running process" (which obviously cannot exist yet) does in fact exist as a FREE entry on the process table with minimum priority. This ensures that *resched* will switch to a task on the ready list and that it will not put this non-existent current

task back on the ready list. Hence it will never return to it which is just what we want.

The bootstrap routine also allocates storage for some global data. In particular, it allocates space for the process table and for pointers to the ready list and to the current process and a pointer to the first block of free memory. It also allocates and initializes the PID counter to 0. Pseudo-code for the bootstrap routine is given below.

bootstrap
    Tell the operator the bootstrap routine has started
    Initialize the memory heap
    Initialize the timer chip and interrupt vector
    for each process table entry
        table_entry.state := FREE
    endfor
    Fill in the 0th entry in the process table for the "nullproc" process
    Set the priority of the nullproc to lowest possible value
    Fill in the next entry in the process table for the ("hello(" process
    Fill in the next entry in the process table for the ("bye(" process
    Link these two entries into the ready list
    Set the current process to point to a non-existent process
    Enable interrupts
    call resched

The C code for the bootstrap routine is:

```
/*
*    Boot.c brings RYMOS to life.  It initializes the memory heap, the
*    timer chip and interrupt vectors for the 68000, puts the null process
*    as well as two real processes on the ready list
*    and invokes resched to start RYMOS.
*
*    Arguments:
*        none
*
*    Returns:
*        nothing
*/

#include "magic.h"
```

```
#include "proc.h"
#include "memory.h"
extern int ctxsw();

/*   Global variable allocation */

MBLCK memhead; /* Pointer to head of free memory list */
PROCESS proctab[NPROCS];    /* Actual process table */
PROCPTR rdyhead;    /* Pointer to ready list of processes */
PROCPTR curproc;    /* Pointer to current process */
short next_pid = 0; /* Process id number to be given to next process created */
extern int meminit();
extern int hello();
extern int bye();
extern int nullproc();
extern int *getmem();
extern int resched();
extern int stop();

main()
{
    int i;

    printf("Booting RYMOS...\n");


    meminit(); /* Initialize heap for memory allocation */

    timeinit();    /* Initialize the PIT and interrupt vector */

    /* Make all entries in process table FREE */
    for(i = 0; i < NPROCS; i++)
        proctab[i].state = FREE;


    /* Insert the null process in the process table as RDY */

    strcpy(proctab[0].name, "null");
    proctab[0].pid = next_pid++;
```

```
proctab[0].pnext = &proctab[1];
proctab[0].state = READY;
proctab[0].text = nullproc;
proctab[0].stksize = 32;
proctab[0].initstk = getmem(32);
proctab[0].cond_reg = STDCCSR;
proctab[0].regs[SP] = (long) proctab[0].initstk + proctab[0].stksize ;
proctab[0].regs[PC] = null;
proctab[0].priority = 0;


strcpy(proctab[1].name, "hello");
proctab[1].pid = next_pid++;
proctab[1].pnext = &proctab[2];
proctab[1].state = READY;
proctab[1].text = hello;
proctab[1].stksize = STDSTKSZ;
proctab[1].initstk = getmem(STDSTKSZ);
proctab[1].cond_reg = STDCCSR;
proctab[1].regs[SP] = (long) proctab[1].initstk + proctab[1].stksize ;
proctab[1].regs[PC] = hello;
proctab[1].priority = STDPRI;

strcpy(proctab[2].name, "bye");
proctab[2].pid = next_pid++;
proctab[2].pnext = NULL;
proctab[2].state = READY;
proctab[2].text = bye;
proctab[2].stksize = STDSTKSZ;
proctab[2].initstk = getmem(STDSTKSZ);
proctab[2].cond_reg = STDCCSR;
proctab[2].regs[SP] = (long) proctab[2].initstk + proctab[2].stksize ;
proctab[2].regs[PC] = bye;
proctab[2].priority = STDPRI;

rdyhead = &proctab[0];
curproc = &proctab[3];    /* "Non-existent" process to fool resched */
strcpy(proctab[3].name, "Start");
proctab[3].priority = 0;    /* Ensure it will be swapped out */
```

```
    enable();    /* Enable processor interrupts */
    resched();
    stop();
}
```

1.2.6 Utility Routines

Ready List Insertion and Deletion

Recall that the list of ready tasks is maintained in order of decreasing priority so that the first one on the list is the next one that should become current. Hence only the first element on the list need ever be deleted which makes the deletion routine particularly simple. Its listing is given below.

```
/*
*    Deletes the first process on the ready list and returns a pointer
*    to the table entry of the deleted process.  (The calling program then
*    either makes the process current or kills it.)
*
*    Arguments:
*        none
*
*    Returns:
*        pointer to deleted entry
*
*    BUGS:
*    Does not detect if ready list is empty (although this should "never"
*    happen).
*/

#include "proc.h"

extern PROCPTR rdyhead;

PROCPTR
rdy_del()
```

```
{
    PROCPTR gone;
    gone = rdyhead;
    rdyhead = rdyhead->pnext;
    return gone;
}
```

Unlike the delete routine, the insert routine must be able to follow the list and its code is slightly more complex (although not nearly as complex as the memory management routines). The list must be maintained in decreasing priority order; but where should a process be inserted if it is equal to another one on the list? Should it go just before or just after the existing entry or does it matter? It does indeed matter. If it went before another entry on the list with the same priority, that equal-priority task would take second place to the one that has just been inserted. This hardly seems fair as the newly inserted task may just have finished executing and will get to execute again before the other task which has been patiently waiting its turn. The solution to this unfairness, of course, is to insert an equal-priority task after any other tasks of the same priority.

Pseudo-code and the C routine for ready-list insertion are given below. Note that the C routine does not incorporate all of the pseudo-code. This is easy to fix and will be part of one of your assignments.

rdy_ins(p)
Inserts the process table entry pointed to by p just before
the a lower priority task.
We assume that global variable ready_head points to the
first entry on the ready list
insert_priority := priority of task p points to
next_ready_task := ready_head
Step though list until point where task should be inserted
while not at end of list **and** insert_priority > priority of task on list **do**
next_ready_task := next task on list
endwhile
insert task pointed to by p in ready_list here
return

```
/*
*    !!!! YOU HAVE TO ADD TO THIS ROUTINE !!!!
*
*    Rdy_ins inserts the specified process in the ready list just before
*    the first process having lower priority or just before the tail.
*
*    Arguments:
*        p - pointer to entry to insert
*
*    Returns:
*        nothing
*/

#include "magic.h"
#include "proc.h"

extern PROCPTR rdyhead;

rdy_ins(p)
PROCPTR p;
{
    PROCPTR q;
    short ins_pri;
    ins_pri = p->priority;
    q = rdyhead;

    /* Mindlessly go through list till end ignoring priority */
    while ((q->pnext != (PROCPTR) NULL))
        q = q->pnext;

    p->pnext = q->pnext;
    q->pnext = p;
    return;
}
```

Timer Chip Initialization and Interrupt Routine

In primitive *RYMOS*, the only event that will result is a task switch is a periodic interrupt generated by a programmable chip on the ECB. So nothing will work unless this chip is properly programmed and the interrupt service routine interfaces correctly with *RYMOS*.

The chip used to generate the interrupts is the Motorola 68221 Parallel Interface and Timer (PIT) chip. This is a very complex peripheral chip and since you probably had your fill of peripheral chips last year we will keep the discussion of this chip down to an absolute minimum. You've probably also had your fill of assembly language programming as well, so we program the chip, set up the interrupt vector and write the interrupt service routine in C.

The only thing we may want to change from time to time is the frequency of timer interrupts. The time between interrupts is specified in milliseconds in the constant symbol INT_TIME defined in the *timer.h* header file reproduced below. So if you want tasks to switch 10 times a second instead of 5 times simply redefine INT_TIME in *timer.h* as 100 msecs and recompile the cooperating system.

```
/*
*    The timer.h include file contains information of the Motorola
*    Parallel Interface/Timer (MC68230 PIT) which is used to generate
*    the clock interrupts for RYMOS.
*/

#define ECB YES
#define CLOCK 4000/32    /*Clock freq (kHz) for ECB divided by PIT  prescaler*/
#define INT_TIME 1000    /* in units of milliseconds */
#define COUNT    (INT_TIME*CLOCK)    /* Preload timer counter with this */
/*    Split 24 bit COUNT into 8 bit pieces (high, middle, low) */
#define COUNTHI (COUNT/(256*256))
#define COUNTMID (COUNT/256 - COUNTHI*256)
#define COUNTLO (COUNT%256)

#ifdef ECB
#define PIT 0x10001    /* Base address of PIT on Motorola ECB */
#define PITINC 2    /* Registers are 2 locations apart */
```

```
#endif

#define TCR *((char *)(PIT + PITINC*16))    /* Timer control register */
#define TIVR *((char *)(PIT + PITINC*17)) /* Timer Interrupt C_Vector Register
#define CPRH *((char *)(PIT + PITINC*19))    /* Counter Preload Register High *
#define CPRM *((char *)(PIT + PITINC*20))    /* Counter Preload Register Mid */
#define CPRL *((char *)(PIT + PITINC*21))    /* Counter Preload Register Low */
#define TSR *((char *)(PIT + PITINC*26))    /* Timer Status Register */

#define TIME_IVECT    (70)    /* Interrupt vector number used by timer */
#define T_CNTRL 0xa1    /* Internal clock, use preload regs */
```

The chip is programmed to generating periodic interrupts by setting the on-chip Timer Control Register to 0xA1. The chip then uses an internal 24-bit counter which is decremented to zero at which time an interrupt is generated. The counter is then reloaded with its preload-register and the sequence repeats. So the preload register has to be initialized with with a count that will decrement to 0 every INT_TIME msecs.

The counter divides the system clock by 32 and decrements at that rate. Since the ECB system frequency is 4 MHz, the counter decrements every 8 μs. Hence the value of the preload register can be calculated as:

$$\frac{preload = Int\_time(inmsecs) \times System\_clock(inkHz)}{32}$$

These calculations are done by the C pre-processor and defined in the *timer.h* file. The PIT chip only has an 8-bit data bus; hence the 24-bit preload register is actually loaded as 3 8-bit pieces. Again the splitting or the 24-bits into 3 bytes is done by calculations defined in *timer.h*.

The PIT is memory-mapped in the ECB starting at location 0x10001. This address information is included in *timer.h*. We also see statements like:

```
#define TCR *((char *)(PIT + PITINC*16))
```

This statement indicates that the PIT control register (register 16) is memory mapped to location 0x10021. Specifically, the define statement tells the C compiler to treat the symbol TCR as "the contents of the byte at location 0x10021", i.e. as the contents of the control register mapped to this address. Hence we can write a statement like:

```
TCR = 0xA1;
```

to initialize the control register.

The actual code for the PIT initialization is reproduced below.

```
/*     Timeinit initializes the hardware timer to generate
*    periodic interrupts.
*
*    Arguments:
*        none
*
*    Returns:
*        nothing
*/

#include "timer.h"
#include "magic.h"
extern int timintrpt();
timeinit()
{
    long *j;

    /* Initialize counter preload register */
    Printf("Initializing timer for %d msec interrupt time\n", INT_TIME);
    CPRH = COUNTHI;
    CPRM = COUNTMID;
    CPRL = COUNTLO;
    TIVR = TIME_IVECT;  / *Timer interrupt vector register */
    TCR = T_CNTRL;    /* Timer control register */
    j =  TIME_IVECT*4;    /* Address of interrupt vector    */
    *j =  timintrpt;
}
```

The 68000 uses automatic vectoring of interrupts. When a device generates an interrupt, the 68000 generates an interrupt acknowledge bus cycle during which the interrupting device must place a vector number on the data bus. Here vector 70 is used for timer interrupts and the timer vector number register of the PIT is initialized with:

```
TIVR = TIME_IVECT;
```

After reading the interrupt vector number, the 68000 then obtains the starting address of the routine for that kind of interrupt from its vector table. The table is located at the low end of memory with each entry containing an

address. Since addresses are 32-bits long, each entry is 4 bytes long. Hence the vector entry for interrupt #70 will be at address 4×70. This address is calculated in the timer initialization routine and vector entry is set to the address of the proper interrupt service routine: *timintrpt*.

The timer interrupt service routine is very simple. It acknowledges the interrupt by writing to the status register (TSR = 1;), re-enables processor interrupts, and calls *resched*. It is reproduced below:

```
/*    Timintrpt is the interrupt service routine for timer interrupts.
*    It invokes the scheduler.
*
*    Arguments:
*        none
*
*    Returns:
*        nothing
*/

extern int enable();
#include "timer.h"
timintrpt()
{
    TSR = 1;
    enable();
    resched();
}
```

You may wonder how a C routine can possibly generate a Return from Exception instruction. After all, the compiler assumes that functions are called as subroutines not as interrupt service routines. The truth is the C compiler *cannot* put in RTE instructions. To get around this, we stop the compilation process after the symbolic assembly language program has been produced. This file is then edited and the tt rts is replaced with an tt rte instruction. The compilation process then continues with the edited version of the assembly language program. All of this can be done automatically using using some features of UNIX as we shall see in the next section.

The Hello, Bye and Null Tasks

These processes simply print out "hello" or "bye" endlessly; the null process does absolutely nothing.

A dummy delay loop is included in both the *hello* and *bye* tasks to slow them down so that context switching can be observed.

The source code is given in Figures **??** and **??** below:

```
/*    Hello prints "hello " endlessly.
 *    Arguments:    none
 *    Returns:  nothing
*/
#include "magic.h"

hello()
{
    int i;
    while(1) {
        printf("hello ");
        for (i = 0; i < DUMDEL ; i++)    /* Empty loop to slow down output */
            ;
    }
}
```

Figure 1.1: The Hello Task (kernel/src/hello.c)

1.3 A Digression on UNIX Features to Help Maintain RYMOS

NOTE: This section is intended to give you a brief glimpse at a useful UNIX facility—*make*. You are not responsible for knowing this material although the more you understand the UNIX system, the more shortcuts you will find to doing your work at the terminal. The section gives a very brief overview

```
#include "magic.h"

bye()
{
    int i;
    while(1) {
        printf("BYE ");
        for (i = 0; i < DUMDEL ; i++)    /* Empty loop to slow down output */
            ;
    }
}
```

Figure 1.2: The Bye Task (kernel/src/bye.c)

```
/*
 *    Nullproc is the null process that runs if nothing else runs.
 *    It does nothing.
 *    Arguments:   none
 *    Returns:    nothing
*/
#include "magic.h"

nullproc()
{
    while(1)
        ;
}
```

Figure 1.3: The Null Task (kernel/src/null.c)

of how a *Makefile* is written, but you will be supplied with a Makefile for your projects.

The source code for *RYMOS* has been organized in several different files. Most of these are C code, some are assembler (*ctxsw*, *stop*, *enable* and *disable*) and some are header files. This modular organization is the proper way to approach large software projects. It raises questions, however, as to precisely how all these files are to be compiled into a single executable program.

Other questions arise with the *cross-development* approach we are using. *RYMOS* runs on the ECB, not on the Sun workstation. machine. But the ECB is not a convenient machine to develop complex programs on. For example, it has no C compiler!

Let's first see how this can be done manually.

Suppose we have a single C program , prog.c, that we want to run on the ECB. We first compile it and stop the compilation process after the symbolic assembly language file has been produced as follows:

```
gcc68k -S prog.c
```

We now assemble *prog.s* (the assembler language translation produced by the C compiler) into object code with

```
gas68k -m68010 -o prog.o prog.s
```

Object code contains a position independent machine-language translation of the assembler. Thus no absolute addresses have been assigned yet and the code may contain calls to subroutines that were not in the original code (these are called *external* routines). These external references may be contained in other object code files on in a library of such files. Putting the pieces of object code together, resolving external references and producing an executable file is done by the *linker* as follows:

```
ld68k -T 1000 -e _main -s -d -N prog.o
```

The options here tell the linker to produce a file that starts execution at address 0x1000 (where we run the program on the ECB) and to use the standard library for unresolved externals. We also don't want the linker to produce a real executable file since the only execution environment it knows about is UNIX and the ECB does not run under UNIX. The -ns option tells the linker to stop just before this final step and produce a file called a.out. The *-e _main* option tells the linker to use the first instruction of the routine called *main* as the entry point. The *-d* option indicates that unintialized

data should be allocated space in the object code. The *-S* option strips the symbol table from the output. The *-N* option allows the text portion to be writable.

To load this program into the ECB we first have to convert its format to *S-records* which is the only file format the ECB understands. This is done with the *srec* command as follows:

```
srec a.out > prog.ecb
```

Finally, the file *prog.ecb* can be downloaded and run on the ECB in the normal way.

The sequence of commands, then, to create a downloadable file is:

```
gcc68k -S  prog.c
gas68k -m68010 -o prog.o prog.s
ld68k -T 1000 -e _main -s -d -N prog.o

srec a.out > prog.ecb
```

This looks like a royal pain. A straightforward solution is to write a shell script command to this all for you.

Now suppose that the program was split into three parts with two parts in C and one section in assembler in the following files: sect1.c, sect2.c and sect3.s. To make a downloadable file, each of the C modules would be separately compiled to assembler; each assembler file assembled; all three object files linked and the S-record file created as follows:

```
gcc68k -S sect1.c
gcc68k -S sect2.c
gas68k -m68010 -o sect1.o sect1.s
gas68k -m68010 -o sect2.o sect2.s
gas68k -m68010 -o sect3.o sect3.s
ld68k -T 1000 -e _main -s -d -N sect1.o sect2.o sect3.o
srec a.out > prog.ecb
```

We again avoid typing such a ghastly sequence of commands by putting them in a shell script command file and executing it instead.

This approach however can be wasteful. Suppose an initial version of *prog* has already been made and a bug is found in *sect2.c*. This is edited and a new *prog* must now be generated. If the above command sequence were executed, there would be needless re-compilation and assembly of sections 1 and 3. All that is really necessary is to re-compile sect2, re-link everything and re-generate the S-record file.

Another problem with the above command sequence if stored in a shell script is that it will mindlessly continue even if compiler syntax errors are detected early in the process. All of these problems and many others can be solved with the UNIX *make* utility.

1.3.1 Make: Painless Program Maintenance

The *make* utility allows the programmer to specify all the actions that are required to generate some target in a file usually called *Makefile*. As well as describing the actions to generate a target, the Makefile also indicates which files depend on which other files. In addition to telling the *make* utility what order to perform the actions in, this information is used to determine which actions are *not* required to generate the target.

To make these ideas clearer, let's look at the Makefile for the *prog* target discussed above:

The first line, *prog.ecb: a.out*, tells *make* that the file *prog.ecb depends on* (that's how to read the semi-colon) *a.out*. In order to generate the target file *prog.ecb* the command line(s) that follow (up to the first blank line) should be executed.

The other entries in the Makefile state the other dependencies. Thus, *a.out* depends on all the object files and the *link* command should be used to regenerate *a.out* from them.

Make does not just blindly follow the generation rules, however. It looks at the modification dates of the files to determine which tasks should be done. Consider again the example posed earlier where one version of *prog.ecb* has already been created and then *sect2.c* is edited to fix a bug. The make utility is then invoked by typing the command *make*. Make sees that *prog.ecb* depends on *a.out* which in turn depends on the object files that are themselves dependent on the source code files. Make starts out at the bottom layer of dependencies. It notices that the object file *sect1.o* was modified more recently than the source code *sect1.c* that it depends on. Hence it knows

```
prog.ecb : a.out
    srec a.out > prog.ecb

a.out: sect1.o sect2.o sect3.o
    ld68k -T 1000 -e _main -s -d -N sect1.o sect2.o sect3.o

sect1.o: sect1.c
    gcc68k -S sect1.c
    gas68k -m68010 -o sect1.o sect1.s

sect2.o: sect2.c
    gcc68k -S sect2.c
    gas68k -m68010 -o sect2.o sect2.s

sect3.o: sect3.s
    gas68k -m68010  -o sect3.o sect3.s
```

Figure 1.4: Makefile for Prog.ecb as Target

there is no need to re-compile sect1.c. Similarly it figures out that there is no need to re-assemble *sect3*. It sees, however, that the source code *sect2.c* is newer than its target *sect2.o*. Hence it recompiles *sect2.c* to bring the object code up to date. Should the compiler detect any errors, *make* will not waste any time by going any further. But if the compilation is successful, *make* will recognize that *a.out* should now be re-generated since one of the things it depends on, *sect2.o* is now newer than it. After re-linking (assuming it is successful), *make* will generate an updated S-record file *prog.ecb*.

Dependencies can be more complex and more deeply nested than indicated here. For example, several files may include the same header file. If the header file is changed, then all the files that include it should be re-compiled. By stating such dependencies in the *Makefile*, this will all be done automatically when *make* is invoked. There is no limit to the number of rules or dependencies in a Makefile.

The *Makefile* that maintains *RYMOS* (not the incomplete one in /u/public/elt048/kernel that you have access to but the real, complete one), for example, contains all the dependencies to generate different versions of *RYMOS*. Simply typing *make* does it all. Not only is *RYMOS* regenerated but all the course notes that refer to source code files are also updated automatically.

Chapter 2

EXTENDED RYMOS PSEUDO-CODE

2.1 Pseudo-code for Basic Routines

2.1.1 Create

```
function create(name, stack_size, priority, text, nargs, args)


        disable interrupts

        find a free entry in process table

        if no free entries then
                enable interrupts
                return ERROR
        endif

        Allocate stack space

        for i = nargs downto 1
                push argument i onto new stack
        endfor
```

```
        Fill in each field in process table record

        enable interrupts
        Return &process_table_entry
```

2.1.2 Kill

```
procedure kill(address of proc_table entry)

        disable_interrupts

        Free stack area


        if process is CURRENT then
                proc_table entry state := FREE
                call resched
        endif

        if process is WAITing on a semaphore then
                signal the semaphore
        endif

        if process is SLEEPing then
                delete it from delta list
        endif

        if process is READY then
                delete it from ready list
        endif

        proc_table entry state := FREE

        enable_interrupts

        return
```

2.1.3 Suspend

```
suspend(address of proc_table entry)

        disable interrupts

        if process is READY then
                delete it from ready list
                state := SUSPENDED
        else
                state := SUSPENDED
                call resched
        endif
        enable interrupts
        return
```

2.1.4 Unsuspend

```
unsuspend(address of proc_tab entry)

        disable interrupts

        insert process in ready list

        call resched

        enable interrupts

        return
```

2.2 Semaphore routines

Semaphores are a common method of process synchronization. They can be used, for example, to mediate requests for limited resources that cannot be shared by different tasks.

Semaphores have an integer value (positive, zero, or negative). A simple way to interpret semaphores is to consider their value as the number of resources available minus the number of requesters.

2.2.1 Semaphore Use Examples

Mutual Exclusion

Suppose we have a semaphore called "Printer_available". The following code would be used to to reliably gain access to the printer:

```
wait(Printer_available);
/* Use printer */
signal(Printer_available);
```

Process synchronization

Suppose we have two tasks, **prod** and **cons** , where **prod** produces data for **cons**. We use two semaphores, **consumed** and **produced**, to coordinate the two tasks as follows:

```
prod(consumed, produced)
{
        int i;
        for(i = 1; i <=2000; i++) {
                wait(consumed);
                n++;    /* n is a global variable shared by both tasks */
                signal(produced);
        }
}

cons(consumed, produced)
{
        int i;
        for(i = 1; i <= 2000; i++) {
                wait(produced);
                printf("n is %d\n", n);
                signal(consumed);
```

```
        }
}
```

The signal and wait routines are described here in pseudo-code. Note that additional routines to create and delete semaphores and data structures to support them (e.g. the semaphore array) are also required.

Each record in the semaphore array contains 3 fields: a flag indicating if the record is being used or not; the value of the semaphore; a pointer to the head of the list of tasks waiting for the semaphore. Semaphores are identified as an index into this array of semaphore records.

The process table is alos modified to include a field identifying which (if any) semaphore the task is waiting for.

2.2.2 Semaphore pseudo-code

Semaphore signal

```
signal(semaphore_id)


        disable interupts
        increment semtab[semaphore_id].count

        if count < 0 then
                delete first task on semaphore wait list
                insert it into ready list
                call resched
        endif

        enable interrupts
        return
```

Semaphore Wait

```
wait(semaphore_id)
        disable interrupts
        decrement semtab[semaphore_id].count
```

```
if count < 0 then
        current_process.state := WAIT
        current_process.semaphore := semaphore_id
        insert it into semaphore list
        call resched
endif

enable interrupts

return
```

2.3 Message Passing Routines (Synchronous)

These routines assume that the process table has been modified to include two additional fields: a flag to indicate if a message is available; a 4-byte message field.

2.3.1 Send

```
send(address of destination proc_tab entry, message)

        disable interrupts
        if message already waiting at destination then
                enable interrupts
                return NOT_SENT
        else
                destination msg_avail flag := TRUE
                destination message = message
                if destination state = RECEIVING then
                        insert destination task in ready list
                        call resched
                endif
        endif
        enable interrupts
        return OK
```

2.3.2 Receive

```
receive
        disable interrupts
        if msg_avail = FALSE then
                state = RECEIVING
                call resched
        endif

        msg = message in process table entry
        msg_avail = FALSE
        enable interrupts
        return msg
```

Chapter 3

Cache Memory

3.1 Improved Performance with Cache Memory

Since the first computers were built over 40 years ago, CPU speed has always been faster than the speed of inexpensive memory. For example, in the days of vacuum tube CPUs, economical delay-line memory was slower than the CPU; similarly, magnetic core was slower than transistorized CPUs; today, 20 MHz (and faster) VLSI CPUs are faster than economical dynamic RAM.

A solution to this problem is to make some of the memory system with very high speed, but expensive, components (eg. static RAM). This high speed portion of the memory system is called the *cache*. But how big should the cache be and how should it be organized?

Suppose we may have 16 Mbytes of 200 ns RAM and a 64 K cache of 50 ns RAM. If we naively assume that all memory locations are equally likely to be accessed, then the chances of hitting the cache are the same as the fraction of memory that is cached. Hence, we calculate that the average access time is:

$$\frac{16000 \times 200 + 64 \times 50}{16064} = 199.4ns$$

This hardly seems an improvement!

The trick is to organize the cache so that it is accessed much more frequently than random chance would indicate. The percentage of memory accesses that involve the cache is called the cache *hit ratio*. Assuming, for example, that we have a hit ratio of 90% with the above sizes, the average

access time becomes:

$$avg.accesstime = main\_access\_timetimes(1-hit\_ratio) + cache\_access\_timetimes\ hit\_ratio$$
$$avg.accesstime = 200 \times .1 + 50 \times .9 = 65ns.$$

3.2 Locality of Reference

Hit ratios approaching 100% are achieved by exploiting the fact that, within a short time period, most memory references are clustered around the same part of memory. This is obviously true in the case of fetching instructions from a loop 100 bytes long that is executed 100,000 times. Clearly the next 10 million instruction bytes fetched from memory will comes from a small addressing space only 100 bytes long. If these 100 bytes are not in the cache initially, we put them in the cache the first time through the loop and hence hit the cache the other 99,999 times through the loop achieving a very respectable hit ratio of 99.999%.

Since all programs contain loops (or, equivalently for our purposes, recursion), which is where most of the execution time is spent, we will achieve locality of reference very often.

Locality of reference also applies to data fetches; within a loop the same data locations will the accessed over and over again. Furthermore, in compiled code for block structured languages, data references are often made to local variables and passed parameters which are all clustered together on the stack.

Hence high hit ratios are achieved by mapping entries in the cache to the most recently used addresses.

When the CPU initiates a memory access, hardware determines if that address is currently in the cache; if so, the contents are returned immediately and no wait states need be inserted in the CPU. Otherwise, main memory is accessed and wait states are inserted until the data has been fetched. This data is now placed in the cache so that it will be found quickly the next time it is accessed.[1] When a hit is not achieved, the cache hardware usually anticipates that the CPU will soon read the next location in memory and tries to transfer a few of the following locations in main memory to the cache

[1]Note that the access to main memory is initiated concurrently with searching the cache. The external bus cycle to memory is then aborted if a hit is found. Hence, if a hit is not found, the system is not slowed down.

before the processor actually needs them. (This is especially useful when the access involved program memory.)

The two basic problems in cache design are:

1. How to determine *very quickly* if the address is currently in the cache;
2. When new data is brought into a full cache, which old data should be removed.

3.3 Determining a hit

3.3.1 Fully associative cache

We first consider a *fully associative cache*.

The cache is usually organized in 2^n blocks with each cache block containing a copy of 2^m contiguous bytes of main memory. In other words, the bottom m bits of the address from the CPU correspond to an index into a byte of a block.

The upper bits of the address (i.e. the 32-m bits of a 32-bit address machine) are stored in the *tag* field of the cache entry. A cache entry, then, looks like (m = 4):

=.5in figures/cache.fass.eps

Figure 3.1: Fully associative cache memory entry format

An address produced by the CPU is considered to be in the format:

=.5in figures/cache.fassaddr.eps

Figure 3.2: Fully associative cache memory entry format

Hence, to determine if an address is in the cache, the upper bits of the address are compared with the tag bits of all cache entries.

Obviously, this comparison must be done in parallel. Detecting equality of two bits requires an exclusive-NOR gate; hence, the circuitry to detect a cache hit in a 1 K cache organized as 64 blocks of 16 bits each would require $28 \times 64 = 1792$ exclusive-NOR gates and 64 28-input AND gates. This would require around 20,000 transistors or about the same as the 32,000 transistors required to implement the cache itself.

The organization of this a fully-associative cache is shown in Fig. **??** below.

=2.5in figures/cache.fass.org.eps

Figure 3.3: Fully associative cache memory organization

3.3.2 Direct Mapping Cache

The amount of hardware required to detect a match can be reduced by using the direct-mapping cache approach. Unlike the fully-associative scheme where a cache block can be mapped to any block in main memory, the direct mapping scheme allows cache blocks only to be mapped to certain memory blocks and no memory block can be mapped to more than one cache block.

This is achieved by numbering the cache blocks form 0 to N-1 (where N is the number of blocks) and only allowing a cache block i to be mapped to memory block j where $j \equiv i \pmod{N}$. For example, with N = 64, cache block 2 could only map one of memory blocks 2, 66, 130, etc. Similarly, those memory blocks could only be mapped to cache block 2. Since cache block #2 can hold main memory blocks 2, or 66, or 130, or 194, etc., we still need a tag field in the cache block to indicate which, if any, of these memory blocks are actually cached. The tag field need only be 22 bits now, however; the lower 4 bits of the address field are used to index into the particular byte of the block we want to access and the 6 bits before these are used to determine which cache block is eligible to hold the particular block we are looking for. (In general, of course, the number of bits in the tag field of a direct mapped cache is $32 - m - n$ when the block size is 2^n bytes and there are 2^m blocks in the cache. In this case m and n are 6 and 4, respectively.)

The disadvantage of this method is obvious; it is now impossible to cache blocks 2 and 66 simultaneously even if the rest of the cache is empty. The advantage is that far less hardware is required to detect a hit. Only 22 exclusive-NOR gates (instead of 1792) would be required and a single 22-input AND gate. This would require only about 200 transistors (instead of 20,000).

The overall organization of a direct-mapped cache with 64 16-byte blocks is shown in Fig. **??**.

=3.5in figures/cache.dir.org.eps

Figure 3.4: Direct mapped cache memory address format

Using the same cache parameters as previously, we view a memory address in the format shown in Fig. **??**.

=.5in figures/cache.diraddr.eps

Figure 3.5: Direct mapped cache memory address format

The middle six bits of the address (the *block* bits) are used to select the only possible cache-block that can be used for this address. To determine if the block is cached, we need only compare the tag field *of this cache block only* with the tag field of the CPU address. Hence no parallel comparison of tag fields is required

3.3.3 Block set-associative cache

The fully-associative cache is clearly the most flexible while the direct-mapping cache is the easiest and cheapest to implement. The desirable combination of flexibility and low cost can be achieved by combining the direct mapping and fully associative methods into the *block set-associative* cache.

In this case, groups of cache blocks are grouped into *sets*; like direct mapping, a memory block can only appear in specific sets and, like associative blocking, all blocks in the set must be compared in parallel.

Suppose, for example, that the 64 blocks are organized as 32 sets of two blocks each. Each block in set i can map into any main memory block j where $j \equiv i \pmod{S}$ (where S is the number of sets).

The organization of a 2-way set associative cache is shown in Figure **??** below.

=3in figures/cache.set.org.eps

Figure 3.6: 4-way set-associative cache organization

Note that the other two forms of cache organization—fully associative and direct mapped—are simply special (extreme) cases of set-associative. Fully associative caching is set associative with a set size equal to the number of cache blocks while direct mapping is set-associative with a set size of 1.

3.4 Block replacement policy

The most commonly used *block replacement policy* (i.e. which block should be overwritten when a new block is required) is the *least recently used* algorithm. The hardware selects for replacement the block least recently used. This is implemented by adding several bits to each block indicating the length of time that has elapsed since a hit on it occurred. This is easily done by clearing this field whenever a hit occurs and incrementing it periodically.

When the old block is overwritten, it may be necessary to first write it back to main memory if a hit on it has occurred during a write cycle. If hits only occurred during read cycles, then, of course, there is no need to write it back to main memory before overwriting it. This is clearly faster than writing it back. In order for the hardware to determine if memory write-back is required when discarding a block, a bit each cache entry is used to indicate if a hit occurred on a write cycle. (This bit is cleared by hardware when the cache block is discarded.)

3.5 Instruction, Data and General caches

We have considered that a cache is used for any memory access in our discussion so far. However, there are often advantages to having separate caches for *data* and *instruction* accesses. The reason is simply that locality of reference is separate for instructions and data and, more importantly, that an instruction-only cache does not have to deal with *writes* to the cache contents (since most operating systems do not allow self-modifying machine language instructions). If the cache contents cannot be modified, the complexities involved in updating the main memory when the cache has been changed are eliminated from the cache controller design. This also simplifies the cache replacement policy.

Indeed, some machines only cache instructions for this reason. Others, like the Motorola 68030, have separate 4K caches for data and instructions. Note that if separate instruction and data caches are built external to the chip, the CPU chip must give some hardware indication of whether a program or data reference is in progress so that the correct cache is enabled.

3.6 Cache controller programming

We have considered that cache memory is a hardware only improvement to system performance and requires no intervention from the programmer. This is not entirely true, however. For example, suppose that I/O devices are memory mapped and a program waits for a status register mapped to address 0xff001003 to become negative with the following code:

```
loop:     tstb 0xff001003
          bpl loop
```

If the status register is initially not in the cache and is positive, it will be brought into the cache and, of course, remain positive. We will then have an infinite loop even though the actual memory location 0xff001003 becomes negative because the cache copy will remain positive.

Consequently, the cache controller must be programmable at least to the extent that we can configure certain addresses or address ranges to be ineligible for caching.

More complex issues arise when we consider multi-processing systems. Typically these systems will have a large main memory system shared by all processors and each processor will also have its own cache. (Note that local cache memories are essential because all processors use the same single bus to communicate with main memory. To avoid making the memory bus a huge bottleneck and have all the processors continually waiting their turn to use it, it is essential that each processor achieve a very high hit-ratio on their local caches.)

A problem arises when a processor modifies it cache copy and the controller writes this value back to main memory. If another processor has the same address cached, it has to recognize that it is no longer valid. Cache-controller hardware to do this is called *bus-snooping* circuitry and is essential in multi-processor systems (or even in systems where DMA controllers can write to addresses eligible for caching).

Chapter 4

Memory Management Hardware/Software

Cache memory is purely a hardware technique used to enhance system performance. It does not make the memory subsystem more reliable, allow *virtual* memory (typically used to allow a program to use more memory than is actually available), or require any software support[1]. We will now consider memory management hardware/software for a virtual memory environment. Some of the hardware techniques used in cache memory are used here too; in particular, as we shall see, fast hardware is required to determine if a CPU address matches a known address. In addition, *translation tables* which are nominally maintained in main memory (from the programmer's point of view) often have a special cache called the *translation look aside buffer* associated with them.

In general, memory management hardware translates the address generated by the processor (called the *logical address*) into a physical address or indicates a fault situation if the translation is unsuccessful. Operating system software controls the method of translation and handles unsuccessful translations. (It is often possible to "repair" such unsuccessful translations but this is done by software and may require disk access, context switches, etc. involving a delay of hundreds of milliseconds rather than the 100 ns required to "fix" an unsuccessful translation in the case of a cache miss.)

[1]Some software support may be necessary to disable caching of memory-mapped I/O locations. This is a small amount of software support, however and would not be required for chips like the Intel 80x86 family which have separate I/O instructions

Before looking at advanced virtual memory management hardware/software like that found in the 80486, 68040, or Sparc chips, we first consider the general principles and examine a simple MMU whose underlying hardware is relatively easy to understand—the Motorola 68451 memory management unit (MMU) in a 68000-based system[2].

We will use this MMU to make a multi-tasking operating system *reliable*. Without such hardware support, it is absolutely impossible to make RYMOS secure in the face of a malicious or careless assembly language programmer. In particular, the simple context switch of RYMOS will only work if all tasks use separate memory areas. But there is nothing to prevent a task from writing to memory used by another task (perhaps overwriting the machine code of the other task or even of the kernel). When such events occur, they will cause the operating system to crash.

4.1 General principles of memory management

In this section, we explore the general principles of memory management and examine what special features on the CPU are required to take advantage of memory management.

The memory management unit sits between the CPU and main memory. as shown in Figure **??**. Its function is to translate the *logical address* furnished by the CPU into a *physical address* passed onto the memory subsystem or to indicate that it is unable to translate the given logical address.

=1.5in figures/mmu-gen.eps

Figure 4.1: Memory management unit generic connections

The MMU has access to a *translation table* that it uses to perform the translation (or decide that the given logical address is illegal). Conceptually,

[2]This was used in the department's first UNIX machine—the Charles River system to reliably implement UNIX. It is now obsolete but is easier to understand than the more sophisticated memory management units used in today's machines.

the translation table consists of a number entries giving logical and corresponding physical addresses as well as status information such as whether the translation is valid. When the CPU initiates a memory access, the logical address it places on the bus is compared by the MMU to all the logical addresses it knows how to translate. If it succeeds in finding one, the corresponding physical address is placed on the memory system address bus; otherwise, a signal is asserted indicating unsuccessful translation.

4.1.1 CPU requirements

The CPU is responsible for recognizing the "address translation error" signal (usually called a *bus error*). This is done in software with a "bus error service routine" that is invoked when such an error occurs, much like an interrupt service routine. There is, however, a very important difference between a bus error and an interrupt. Interrupts are only recognized by the CPU after an instruction has completed execution. Bus errors, however, must be recognized as soon as they occur because the instruction *cannot* continue if a bus error occurs. Note, for example, that the execution of a memory-to-memory move instruction involves at least three memory accesses: one (or more) to read the instruction; one to read the source operand; and one to write to the destination.

Consequently, the CPU in a system with memory management needs to be able to abort an instruction before it has completed. Simple 8-bit processors, for example, do not have this capability. Nor does the Intel 8088, used in the IBM PC and XT. Consequently, computers based on these chips cannot use memory management systems.)

The situation is actually more complicated than this once we consider what we may want the "bus error service routine" to do. The simplest way to use memory management hardware is to *protect* the address space used by one task from being corrupted by another task. In this situation, an address translation fault means that the instruction causing it was doing something "illegal"; the operating system then summarily kills the offending task (and prints an error message like "Bus error").

However, more sophisticated use of memory management allows the operating system to provide each task with a *virtual address space* that may be larger than the actual physical RAM on the system. Only some portions of a task's logical memory spaced is mapped to physical memory at any time.

If a translation error occurs, it may be due to an "illegal" access (as in the protected system) *or* it may be that the logical address is legal but is not mapped to physical memory (in which case, the actual memory contents will be stored on a disk drive). In the latter case, the operating system will have to "repair" the condition that led to the bus error by reading in the memory location from disk to physical memory and mapping the logical address to this location. Once this is done, the instruction that originally caused the bus error will be *re-executed* from the beginning or *continued* from the point where it was aborted.

To re-execute or continue an instruction is a complicated operation, however. Re-executing requires that any partial effects of the instruction be undone. (For example, if the 680x0 instruction `move -(a0),(a3)` results in a bus error when `(a3)` is accessed and `a0` has already been decremented, the CPU would have to increment `a0` before re-executing the instruction.) Instead of re-executing an instruction that caused a bus error, some processors save enough information on the stack to allow it to be continued from whatever point in its execution it had reached when the fault occurred.

In short, a more sophisticated processor is required to implement either instruction re-execute or instruction continuation. For example, the 68000 does recognize bus errors and aborts the current instruction. However, it is incapable of re-execute or continuing the instruction. Consequently, while it can be used in a protected memory management system, it cannot be used in a virtual memory system.

Which is better, instruction re-execute or continuation? Instruction continuation may seem superior because we will not waste time repeating that portion of the instruction that has already been executed. However, a lot of information has to be pushed onto the stack to implement instruction re-execute which may consume more time than the time saved. Which method is better really depends on overall CPU architecture and the interplay of complex tradeoffs in chip design. The "best method" is ultimately a subject of debate. The Motorola 680x0 family uses instruction continuation while the Intel 80x86 family uses instruction re-execute.

There is one additional feature that the CPU must have when memory management is used. The MMU must, of course, be programmable and the operating system programs it to perform the mapping of logical to physical space that is desired. Obviously, the benefits of memory protection are completely lost if any task can re-program the MMU; nothing but the operating

system must do this. Consequently, the CPU must have at least two different modes of operation: one mode that the Operating system alone can use and that allows the MMU to be reprogrammed and another mode for ordinary tasks in which any attempts to re-program the MMU are prevented by hardware. For example, the 680x0 family has two operating modes: User and Supervisor. The 80x86 implements a more sophisticated system with four modes of operation.

The protection possible when the MMU knows the operational mode of the CPU is greater than this, however. It was mentioned earlier that each translation table entry has status information associated with it. Besides indicating that a particular translation table entry is valid, the status often indicates if the translation is only legal in supervisor mode. In this way, the kernel can program the MMU so that user-level tasks cannot access mapped areas that correspond to private kernel memory. To summarize, a CPU must have the following features in order to take advantage of memory management:

- Ability to run in at least two operational modes (e.g. user and supervisor).
- Ability to abort an instruction and execute a "bus error service routine" whenever an address cannot be translated.
- If virtual memory is to be implemented, the CPU must also be able to continue or re-execute a previously aborted instruction.

4.1.2 Types of MMUs

There are many different types of MMUs. Let us first look at a hypothetical, simple MMU that has been programmed to translate a 16 Megabyte logical addressing range into a 8 Megabyte physical memory system as shown in Fig. **??**.

=5.5in figures/genmmu-map.eps

Figure 4.2: Simple memory mapping example

Let us first be clear on what the diagram means. The logical address are produced by the CPU and range in value form 0x000000 to 0xffffff. (i.e. The logical address space uses 24 bits and is 16 MB in size.) These addresses are then translated by the MMU into physical addresses in an 8 MB range (0x000000–0x7fffff).

Looking at the physical address map first, we see that the kernel is loaded into the lowest 2.5 MB of memory (code in the first 1.5 MB and data and stack in the next two .5 MB slots). Various portions of memory are then allocated to sections of Task-A. The code section occupies 1 MB from 0x300000 to 0x3fffff; Its stack and data sections, which are each 512 Kbytes, start at 0x500000 and 0x600000 respectively. The kernel and Task-A account for 4.5 Mbytes of the 8 MB of physical memory. The other 3.5 MB are not accounted for (yet!).

Advantages of memory management

But what is gained by all this translation stuff? The most important benefit of address translation is that our multi-tasking operating system can now be made *secure*. The operating system programs the MMU at the time it switches contexts to Task-A, setting up appropriate translation tables. Note that, although kernel resources are memory mapped so that the kernel can access them, any attempt by Task-A to access these locations would be denied by the MMU because the CPU would have to be in supervisor state to enable those mappings.

Besides making security possible, address translation also makes things much more *convenient*. Without memory management, tasks must be loaded at *specific* memory locations or be compiled for position independent code (which usually results in less efficient execution). For example, Task-A has been compiled so that its program starts at location 0x000000 and its stack is initialized at the high end of memory (0xffffff). Without memory management, Task-A would have had to be loaded at those locations. This would have been quite a trick since the stack address 0xffffff doesn't even exist in physical memory and the low end of memory is already occupied by the kernel!

Indeed, with memory management hardware, we can compile all tasks so that they begin at 0x0 have a stack at the high end of memory. We can also obtain other advantages.

Suppose, for example, that Task-A requires more stack space. As currently mapped, the last logical stack address available is 0xff8000 which is mapped to 0x480000. We could add another 0.5 MB of logical stack addresses between 0xf00000 and 0xf7ffff. These would definitely not be mapped to the physical memory contiguous with the current portion of physical memory allocated to the stack since this would overwrite the code portion of Task-A. However, there are spare 0.5 MB sections elsewhere in physical memory that we could map the increases stack area to. In short, we have the capability of mapping large contiguous logical address spaces into smaller, non-contiguous physical sections. This greatly reduces the problem of *memory fragmentation* that would otherwise arise.

A final thing to note about the hypothetical MMU of Fig. **??** is that entire ranges of addresses are mapped, not individual addresses. There are, in general, two ways to do this depending on whether the size of all the ranges is fixed and how many different mappings there are. These are called the *paged* and *segmented* approaches to memory management. We examine them now.

Segmented memory

In segmented memory systems, the MMU translates a relatively small number of variably sized *segments* from logical to physical space. A segment is generally chosen to correspond to a logical division in the program's address space such as stack, initialized data, un-initialized data, code, etc.

Segmentation is the primary method used in such processors as the Intel 80286 where the segment size can be as small as 16 bytes or as large as 64 Kb.

The main advantage of segmentation is the relatively small number of address translation entries that are necessary (hence simplifying the hardware design of the MMU). Another important advantage is that memory is used quite efficiently as the size of the segment can be tailor made for the section of the program that it holds.

There are some serious disadvantages, however, especially in a *virtual memory* system. The problem is related to treating an entire section of the program (e.g. the code segment) as one indivisible unit that is entirely mapped or entirely un-mapped. In virtual memory systems, logically valid address regions may be un-mapped and stored on disk until needed. In the

case of the code segment, for example, the operating system may want to un-map part of the section and keep the rest mapped. But partial mapping of a section can only be done on a segmented system by splitting the section into two or more different segments. As soon as we begin to do this in earnest, however, we dramatically increase the number of segments and lose the advantages associated with segmented systems.

In short, segmented memory management systems are most appropriate to small systems where virtual memory is not used extensively, the number of tasks is small and fits into physical memory, and the size of physical memory is small.

Paged memory

In *paged* systems, all translatable sections of memory are the same size, called the *page size*. Page sizes are small compared to the size of memory, but usually larger than the smallest segment size. (Typical page sizes are 256 bytes for the VAX, 4K for the 80386, and 8K for the Sun Sparcstations.)

The main advantage of paged systems is their greater flexibility compared to segmented systems, particularly in virtual memory situations. Suppose, for example, that a task has a 1 Mb code section. With a segmented system, it would occupy one segment and would either be entirely mapped or entirely un-mapped. With paging, however, it would require 256 4K pages any combination of which could be mapped at any given time. In an extreme example, the program may spend almost all of its time in an inner loop that occupies less than 1 page of memory. With paging, only this single page would need to be mapped once the program was up and running. In other words, the "real" memory resources consumed by the program once it was running would be only 4K for a paging system versus 1 Mb for a segmented system, an improvement of more than 99%.

There are disadvantages to paging, however. Most important is the sheer size of the mapping tables. In principle, every *addressable* page must be mapped. For 32-bit machines with an 4K page size, the 4 Gbytes of addressable space can be divided into 1024 K pages. Each page descriptor would require at least 4 bytes. Hence out page tables would require 4 MB of memory !

This is a considerable overhead and various techniques are available for reducing it. Nevertheless, even with reduction techniques, there are almost

always far more address translation table entries in a paged system than in a segmented system. While a segmented system may have a small enough number of entries in its translation table that they can all be accommodated in specialized hardware registers, paging systems usually have such large tables that they must be stored in memory itself. This, in turn, requires that caching techniques be used so that the most frequently used page accesses do not require additional memory cycles to retrieve the translation information from memory. (Note, however, that memory table lookup, when required, is done by hardware in microcode and is transparent to the programmer.)

Virtual memory

Virtual memory systems add kernel-level software to a memory management system that allow users to access more memory than physically exists on a system. This is done in an entirely transparent way so that the programmer need have no knowledge of the actual amount of memory present. The programmer simply assumes that, say, 4 GBytes of memory exist and use as much of this memory space as required for his application. If the program requires 64 MByte of memory and there is only 8 MB of physical memory available, the program will still work. The only difference will be that it will run somewhat slower when there is less physical memory than required due to *page faults*.

The basic idea in paged virtual memory systems is to retain in physical memory only those pages that are currently being used. As with cache memory, "locality of reference" tells us that at any given time only a small proportion of a program's total address space is in use. When a program running in virtual memory system accesses a page that is not mapped, the kernel software (in the "bus error service routine" examines the status information in the translation table entry that caused the fault. The kernel can use bits in the status field to indicate that the page requested is "cached" on disk. (The kernel could use the physical address field of the translation entry for other purposes in this case; for example, it could hold the block number on disk where the page was saved.)

The kernel would then look for a page in physical memory that had not been accessed recently and write it to disk. The newly freed physical memory page could then be used to read in the page that had caused the *page fault*. Of course, since a multi-tasking operating system is assumed, the kernel could

simply schedule these disk transfers and do a context switch to another task whose current pages were in memory while it waited for the disk controller to complete the page transfers. Thus the CPU would not necessarily be sitting idly by while the "page repair" took place.

4.2 The 68451 MMU

The Motorola 68451 Memory Management Unit implements a segmented[3] system. The MMU sits between the processor and main memory as shown in Fig. **??**.

=1.5in figures/mmu451-more.eps

Figure 4.3: Basic configuration of 68551 MMU in 68000 system

4.2.1 Some 68000 details

To understand the operation of the address translation mechanism some preliminary information about how the 68000 works is necessary. Specifically:

- Although the 68000 has 32-bit address and data registers internally, its external connections only provide a 16-bit data bus and a 24-bit address bus.
- Address line A_0 is missing; rather the address lines A_{23} to A_1 give the *word* address. The processor also asserts strobe lines ($\overline{UDS}$ and $\overline{LDS}$) to indicate if it wants to access the lower or upper byte of the word or the entire word.
- The 68000 operates in either "user" mode or "supervisor" mode. When in *user* mode, certain instructions (such as enabling or disabling interrupts) are disallowed. The processor automatically switches to supervisor mode when an interrupt, bus error, or software interrupt occurs.

[3]If all the segments are made the same size, it could be argued that the 68451 implements paging. There are only 32 translation entries on the chip, however, and most would consider this too small a number to be considered a paging system.

- The 68000 gives an external hardware indication of the type of memory access it is performing with the *function code* lines (the 3-bit FC_2 to FC_0 code). These indicate if the processor is accessing instructions or data and whether it is in user or supervisor state as shown in Table **??** below.

Function code	Meaning
000	Undefined (reserved)
001	**User data**
010	**User program**
011	User defined
100	Undefined (reserved)
101	**Supervisor data**
110	**Supervisor program**
111	**CPU space**

Table 4.1: Function codes for 68x00 processors

- (Note that, in a sense, there are 35 address bits in the 68x00 family since the Function Code bits can be used to split the address space into 8 different "segments". Although not all of the segments are predefined, the `MOVES` instruction (which is available only in supervisor mode) can coerce any value onto the FC lines.)

Basic 68451 operation

Inside the chip are 32 *translation descriptors* that are used to translate a logical address into a physical address. The physical address is then presented to the memory system in the normal fashion. Note that only the upper 16-bits of the 24-bit address are translated. The lower 8 bits are identical in both the logical and physical addresses.

If the translation of the logical address to physical address does not succeed, the *Bus error* signal is asserted and the instruction in progress is aborted.

In simplified terms, the attempt to translate from logical to physical addresses proceeds as follows.

1. Each descriptor contains a logical address to match with the logical address from the CPU, a physical translated address, and an *address mask* indicating which bits of the logical address have to match.

2. If a match is found between the CPU's logical address and the unmasked bits of a descriptor's logical address, the physical address is formed from the unmasked bits of the descriptor's physical address and the masked bits of the logical address.

3. If no match is found, a BERR signal is asserted.

Suppose, for example, that the logical, physical and mask fields of Descriptors 0, 1, 2, 3, 4, and 5 are as shown in Table **??**. (Note that since the the 68000 only has a 24-bit address bus and the 8 low order bits of the logical address are not translated, only the most significant 16 bits remain to be translated. Consequently, the logical address, physical address and logical address mask entries in a descriptor are 16 bits.)

The logical mask field effectively indicates the size of the segment being translated. In the example in Table **??**, the 0xF800 logical masks indicate that only the most significant 5 bits of the logical address need match. Consequently, the least significant 19 bits do not matter; thus, these entries describe 512 Kb segments. Similarly, the entry with a mask 0f 0xF000 describes a 1 Mb segment. (Note also that the segments described correspond to the Task-A code, data and stack segments of Fig.**??**.)

	Simple 68451 descriptors					
	Descript. 0	**Descript. 1**	**Descript. 2**	**Descript. 3**	**Descript. 4**	**Desc**
Logical	0000	0800	8000	F800	4000	5000
Physical	3800	4000	6000	4800	0000	1000
Mask	F800	F800	F800	F800	F000	F800

Table 4.2: Simple example of 68451 descriptors

The mappings defined by Table **??** are described pictorially in Figure **??**. Let us now look at the descriptors in more detail.

Descriptors 2 and 3 map the 512 Kb segments corresponding to Task A's data and stack sections. The other descriptors are not as "clean"; they

=4.5in figures/map-451-1.eps

Figure 4.4: How Descriptors 0–5 are mapped in Table **??**

illustrate some "weaknesses" in the 68451's design and how these problems are solved.

Descriptors 0 and 1 are used to map the 1 Mb code segment for Task A from the logical address range 0x000000–0x0fffff to the physical range 0x380000–0x47ffff. Since 1 Mb descriptors are possible, why not just use 1 descriptor to map the entire range? The reason is that the 68451 chip simply ignores the bits in the physical address that are masked. Only the unmasked bit positions are changed. We want to map addresses beginning with 0x00 to addresses beginning with 0x38 which means we want to change the most significant **5** bits. Hence the mask must begin with 5 1's, meaning that the segment size can only be 512 Kb. To do the mapping we want, we are forced to split what is "logically" a single 1 Mb segment into 2 physical 512 Kb segments.

In more sophisticated MMUs, it would be possible to do what we want if the physical address were obtained by *adding* the descriptor's physical address to the logical address instead of just substituting the most significant bits. Such an MMU, however, would require more circuitry (i.e. an adder) and would be slower. (As we shall see later, however, this method is used in the Intel 80x86 family of processors).

The kernel code section occupies the 1.5 Mb logical address space from 0x400000–0x57ffff. It is also split into 2 physical segments: one mapping the first Mb (Descriptor-4) and the other mapping the remaining 512 Kb (Descriptor-5). The reason for splitting the "logical" segment in two parts here is different than it was in the previous case. A segment size of 1.5 Mb is simply not possible under any circumstances with the 68451. Although segment sizes can be as small as 256 bytes or as big as 8 Mb, the segment size must be an exact power of 2; since $1.5Mb = 2^{20} + 2^{19}$ is not an exact power of 2, it cannot be mapping with a single segment. Note, however, that in this case we are at least able to map the lower 1 Mb with a single descriptor because we are mapping to a physical address space where only the 4 most significant bits differ from the logical address.

As described so far, there is a major problem with the 68451 chip. Specifically, there is no distinction between user space and supervisor space. Figure **??** shows that the 68000's Function Code lines are connected to the MMU chip. Let us look at how these are used.

Besides 32 descriptors, the 68451 chip has 16 8-bit registers collectively called the *Address Space Table*. For our purposes, we will only consider the *Address Space Table*(*AST* to have 8 instead of 16 entries, however[4]. The AST is programmed by the kernel. When an address access is made, the MMU uses the 3 Function Code lines as index number (in the range 0–7) into the AST where it retrieves an *Address Space Number ASN*. This ASN is then used to further identify the Descriptors that are eligible to be used in translating the logical address.

Each descriptor also as an Address Space Number associated with it. In order for a descriptor to be eligible to translate a logical address, its ASN must match the ASN retrieved from the Address Space Table from the Function Code number. The match need not be exact, however, as each descriptor also has an 8-bit *Address Space Mask* which specifies which bits of the global AST must match the descriptor AST in the same way that the Address Mask indicates which bits of the logical address have to match the bits of the descriptor's logical address.

In short, for a descriptor to successfully translate a logical address, two conditions must be met:

1. The CPU's address must match the descriptor's logical address (after masking by the descriptor's address mask); and,

2. The ASN derived from the Address Space Table must match the descriptor's masked ASN.

Finally, if no descriptor matches the CPU's address as described above, the Bus Error line is asserted.

The above description of how an address is translated explains how the ASN is used, but does not provide any details on what the ASN really means and how its value is set. The MMU hardware simply follows the rules stated

[4]It is possible to use all 16 entries in more complex situations where more than one *bus master* is used on the system. Examining such complex systems is beyond the scope of the present discussion.

above; it is up the operating system software to set up the various fields of the descriptors and the global Address Space Table.

The Address Space Numbers in the AST are usually chosen by the operating system to prevent tasks from accessing areas that are mapped but which the operating system a particular task or tasks to gain access to. To see how this can be done, consider the expanded set of descriptors given in Table **??** and the Address Space Table layout when each task is running as shown in Table **??**.

	Descript. 0	**Descript. 1**	**Descript. 2**	**Descript. 3**	**Descript. 4**	**D**
Logical	0000	0800	8000	F800	4000	50
Physical	3800	4000	6000	4800	0000	10
Mask	F800	F800	F800	F800	F000	F8
ASN	02	02	01	01	82	82
ASN Mask	7F	7F	7F	7F	80	80
	Descript. 6	**Descript. 7**	**Descript. 8**	**Descript. 9**	**Descript. 10**	**I**
Logical	0000	F800	F800	A800	6800	8
Physical	2800	7000	5800	1800	2000	3
Mask	F800	F800	F800	F800	F800	F
ASN	08	04	10	81	81	1
ASN Mask	7F	7F	7F	80	80	7

Table 4.3: Complete 68451 descriptors

=4.2in figures/map-451-2.eps

Figure 4.5: How Descriptors 0–11 are mapped

When examining Table **??**, note that there several descriptors for the same logical address range. For example, Descriptors 3, 7 and 8 all map the logical address space 0xf80000–0xffffff. But they are map to different physical addresses. They are, of course, the stack space descriptors for the tasks A, B and C.

AST Entry	Task A	Task B	Task C
User data (1)	0x01	0x04	0x10
User program (2)	0x02	0x08	0x02
Supervisor data (5)	0x81	0x84	0x90
Supervisor program (6)	0x82	0x86	0x82

Table 4.4: Sample Address Space Table entries for 68451

In short, the descriptors describe the mappings for several different tasks. It is the Address Space Table that determines which task has active descriptors at any given time. Consequently, the AST is usually re-programmed at context switch time by the kernel. This is much more efficient than re-programming the entire MMU every time a context switch occurs.

Since the AST is indexed by the Function Code bits and, in normal operation, only 4 for the possible FC bit patterns are used (i.e. 1 for user data, 2 for user program, 5 for supervisor data and 6 for supervisor program), only those 4 entries in the AST need be reprogrammed at context switch time. This makes the overhead of implementing memory management in a multi-tasking environment very low as far as the additional time required to perform a context switch.

Referring now to the Space Mask entries in the the descriptors, note that many are 0x7f (i.e. the MSB is 0) while the others have the MSB set. We see that all segments belonging to the kernel have an ASN and mask with the most significant bit set. Consequently, only AST entries with this bit set can possibly match. Examining Table **??**, we see that only the entries accessed in supervisor mode meet this requirement. Consequently, all kernel segments are protected from being accessed by user tasks. Since the masks in user descriptors never have the most significant bit set, the kernel can still access user space as well as its own space.

Figure **??** shows how the descriptors perform the required mappings for all of the tasks. (The reader should confirm his or her understanding of the descriptors and address space tables by seeing how they relate to the figure.)

The following sections until ?? in *small print* **are for reference only.**

4.2.2 Other programmable registers

Segment status register

There is a segment status register associated with each of the 32 descriptors as shown in Figure **??**.

=0.4in figures/451-status.eps

Figure 4.6: 68451 Descriptor Status Register configuration

The bits are used as follows:

U: The Used bit is set when a descriptor is used successfully in translation. The Operating System can use this information to implement a least-recently used segment replacement policy. The bit can be cleared by writing to it.

I: The Interrupt bit is used to enable interrupts when the descriptor is used in a translation. The Operating System could exploit this to set data breakpoints or to emulate virtual memory-mapped devices that have not yet been installed.

IP: The Interrupt Pending bit indicates that this descriptor has generated an interrupt.

M: The Modified bit is set when data is written to a segment. The Operating system can use this information to determine if an old segment has to be re-written to disk when it is replaced.

WP: The Write Protect bit prevents write access to the segment being described. An attempt to write will generate a *FAULT*.

E: The Enable bit indicates that the descriptor is valid and can be used in translation.

Descriptor Pointer

Each descriptor is 9 bytes long; hence all 32 descriptors are 288 bytes long. Rather than have each descriptor memory mapped, only one descriptor at a time is memory mapped to the MMU's *accumulator*. The 5 low bits of the Descriptor Pointer indicate which descriptor is currently accessible.

Read Descriptor Pointer

If a FAULT occurs, this register is set to the descriptor number that caused the fault (e.g. write violation).

Local Status Register

Figure **??** shows the layout of the "local" status register. (Note that this "local" register applies to the status of the entire 68451 chip, *not* to any "local" condition in a specific descriptor. It is called "local" because it is possible to design a memory management unit using several 68451 chips to allow for more than 32 descriptors. The *GAL* and *GAT* fields in the local status register are used for "global" purposes in a multi-68451 system. Such systems are beyond the scope of these notes.)

=0.3in figures/451-lsr.eps

Figure 4.7: 68451 Local Status Register

The bits indicate:

L7-4: describe the cause of the last event (e.g. 1100 = write violation, 1010 = undefined segment (no successful translation), etc.)

RW: indicates whether a read or write operation was in progress when event occurred;

LIP: Local Interrupt Pending is set if an interrupt has been generated by any of the descriptors.

4.3 Programming the 68451

The MMU resides in physical address space and its internal registers are memory mapped to 64-bytes (32 words) of memory. These are programmed like any internal registers of a memory-mapped peripheral device. We can program the chip in C by first setting up define statements setting up the memory-mapped addresses of the registers:

```
#define MMU             0x???????? /* Whatever the base address of MMU is */
#define Ast0_altfc3_0   (*((char * ) MMU))
```

```
#define Ast1_user_data  (*((char * ) MMU + 2))
#define Ast2_user_prog  (*((char * ) MMU + 4))
   . . .
#define Ast5_super_data  (*((char * ) MMU + 10))
#define Ast6_super_prog  (*((char * ) MMU + 12))
#define Ast7_intrpt_ack  (*((char * ) MMU + 14))
   . . .
#define Log_base_addr    (*((int *) MMU + 32))
#define Log_addr_mask    (*((int *) MMU + 34))
#define Phys_base_addr   (*((int *) MMU + 36))
#define Addr_space_num   (*((char *) MMU + 38))
#define MMU_status       (*((char *) MMU + 39))
#define Addr_space_mask  (*((char *) MMU + 40))
#define Descriptor_ptr   (*((char *) MMU + 41))
#define Intrpt_vector    (*((char *) MMU + 43))
#define MMU_global_status    (*((char *) MMU + 45))
#define MMU_local_status (*((char *) MMU + 47))
#define MMU_segment_status  (*((char *) MMU + 49))
#define Intrpt_descriptor   (*((char *) MMU + 57))
#define Result_descriptor(*((char *) MMU + 59))
#define Direct_translate (*((char *) MMU + 61))
#define Load_descriptor  (*((char *) MMU + 63))
```

The outline of some utility routines follow:

4.3.1 Setup_descriptor

```
int setup_descriptor(unsigned int descriptor_number,
    unsigned int physical_base,
    unsigned int logical_base,
    unsigned int address_mask,
    unsigned char address_space_num,
    unsigned char address_space_mask,
    unsigned char status)
{
    /* Make MMU accumulator be proper descriptor */
    Descriptor_ptr = descriptor_number;
    /* Check if this descriptor is already in use */
    if ((MMU_status & 1) == 1) { /* In use */
```

```
        ERROR
    } else {
        disable_interrupts();  /* Disallow interrupts here */
        Log_base_addr = logical_base;
        Log_addr_mask = address_mask;
        Phys_base_addr = physical_base;
        Addr_space_num = address_space_num;
        MMU_status = status;
        Addr_space_mask = address_space_mask;
    }
}
```

4.4 Paged Memory In Motorola Systems

In this section we examine how paged memory management systems are implemented in the Motorola 68000 series[5].

First, let us look at the requirements for paged systems as compared to segmented systems. The most striking feature is that the translation tables are *much* larger in a paging system simply because the page size is much smaller than the segment sizes used in segmented systems.

In a paged system, all addresses have the format shown in Fig. **??**. Basically, the most significant bits indicate the page number while the lower bits indicate the offset within a page.

=0.4in figures/page-addr.eps

Figure 4.8: Address format in a paged system

In the 680x0 family, the page size can be any power of 2 between 256 bytes and 16 Kb. However, once the bootstrap routine sets the page size, it does not change and all pages have the same size. We will assume in most examples that the page size is 4 Kb (i.e. the 20 upper bits of the address are the page number and the lower 12 bits are the offset within the page).

[5]This applies to the 68020 using the separate 68881 MMU chip as well as the 68030 and 68040 which both incorporate the MMU into the same chip as the CPU.

We have already seen that the number of page descriptors will be very large in a paged system. Indeed, it will be so large that the "page tables" will have to be stored in memory. The Memory Management Unit will contain a pointer to these page tables as well as a specialized cache memory to hold the translation information for the most recently used pages. In order to map all of virtual memory in a single "page table", we would need 2^{20} or about 1 million entries! Furthermore, each task should have its own page table, so a separate million entry page table would be needed for each task. Assuming each page table entry is 8 bytes and that we have 50 tasks, we would require 400,000,000 bytes of storage just for the page tables! This kind of arrangement is shown in Fig. **??**.

=2.4in figures/pagetab-huge.eps

Figure 4.9: A hypothetical "huge" page table

These calculations should be enough to convince you that this way of implementing paged memory is totally impractical. The basic problem is due to the requirement of mapping all of the logical address space. Very few applications really need 4 Gigabytes of addressing space! We can dramatically reduce the size of the tables by limiting the number of pages that are actually mapped in at least 3 ways:

1. Impose a limit on the number of page descriptors.
2. Use a *multi-level* tree structure in our page tables.
3. Use a form of segmentation to allow several pages that occupy contiguous logical and physical space to be mapped with a single page table entry.

Let us examine how each of these methods could work. As we shall see, they are particularly powerful when combined.

In the example of Fig. **??**, the "root pointer" simply indicated the starting address of an array of page descriptors with the assumption that the array was big enough to translate *any* logical address. It is a simple matter to add additional information to the *root pointer* giving the number of entries in the

table it is pointing to. For example, if only the first 80 K of logical address space is used, then only 20 page descriptors are required since any logical address outside of this range would be assumed to be illegal. Thus we add a *limit field* to the root pointer. In this case, the limit would be 20 as shown in Fig. **??**.

=2.4in figures/pagetab-small.eps

Figure 4.10: A hypothetical "small" page table

This method has the severe disadvantage that it can only map a contiguous region of logical addresses. As in a segmented system, it is often necessary to map several regions which are far apart in memory. The most common example is mapping code regions to low memory and stack regions to high memory.

One way to achieve this is to use a *two-level* page table. In stead of having the root pointer give the starting address of the array of all page descriptors, it points to an array of *table descriptors*. Each table descriptor in the array in turn points to another array of page descriptors.

The first few bits of the logical address are used to index into the table of descriptors. The selected descriptor points to a page table and the reaming bits in the logical page number are used to index into this table to retrieve the physical address of the page, or the *frame*[6].

This type of arrangement is shown in Fig. **??**. There are 16 entries in the descriptor tables and one is selected by the most significant 4 bits of the logical address. The remaining 16 bits of the logical page number ($A_{27} - -A_{12}$) are used to index into a page table whose starting address is determined by the descriptor table pointer found in the first table.

Note that each descriptor in the first level table is like a "mini root pointer" with its own limit field. We have further specified the limit to be an *upper* or *lower* limit by appending `-U` or `-L` to the limit number. The actual index must be less than or equal to upper limits or higher than or equal to lower limits. This is particularly useful when only the high addresses of a section of memory are accessed as is the case for stack accesses.

[6]The word *frame* is often used for the portion of physical memory corresponding to a logical address page.

Finally, note that only 12 of the 16 descriptors in the first level table actually point to page tables. The others have been set up as "invalid" entries. Similarly, the first entry in the page table pointed to by descriptor 'A' has a "bad" status. Only the second entry is mapped.

=3.4in figures/pagetab-2level.eps

Figure 4.11: A hypothetical "2-level" page table

The reader should confirm the tables as set up result in the following translations shown in Table **??**. Any address not in the logical address range given in the table would result in a Bus Error.

Logical address	**Physical address**
00000000--00000FFF	00AB1000--00AB1FFF
00001000--00001FFF	008BD000--008BDFFF
00002000--00002FFF	00345000--00345FFF
00003000--00003FFF	00451000--00451FFF
A0001000--A0001FFF	C0678000--C0678FFF
FFFFE000--FFFFEFFF	00123000--00123FFF
FFFFF000--FFFFFFFF	00067000--00067FFF

Table 4.5: Sample translations for Fig. **??**

Note that we have been able to translate a very wide addressing range with a relatively small table. The total number of entries is 16 in level-1 and 8 in the level-2 tables. Only 24 entries occupying 196 bytes is required for the entire mapping. This is a very considerable improvement over a single-level table!

The idea of multi-level tables can be extended to more than 2 levels. Fig. **??** shows a 3-level configuration. The advantages that we obtain with the 2-level mapping are obtained at a lower level here. Tables with more than 2 levels are particularly efficient when the application requires many, many small portions of memory at widely different logical addresses. This is often

the situation in artificial intelligence applications or in data base applications. It would be quite rare in scientific "number-crunching" applications.

Fig. **??** also uses a new "trick" to reduce size of page tables—the *early termination* page descriptor. The second level of tables contain pointers to the third level page tables except for the second entry in the descriptor table pointed to by entry "A" of the first table. The *type* entry here indicates that this is a page descriptor, not a pointer to a third level page table. Such an "early-termination" page descriptor is used to map the entire logical addressing range that accesses to a contiguous portion of physical memory starting at the address indicated by the value field of the descriptor (i.e. the field that would normally point to a third level page table). In this case, the entire 1 Mb logical address space `A0100000--A01FFFFF` is mapped to the physical memory range `B0400000--B04FFFFF`. This dispenses with an entire 256-entry third level page table. Of course, it requires that we have 1 Mb of *contiguous* physical memory. This kind of mapping would be particularly useful for mapping I/O space. Note that early termination descriptors could also be used in the Level-1 tables. In this case, one descriptor would map a 256 Mb contiguous range.

=4.4in figures/pagetab-3level.eps

Figure 4.12: A hypothetical "3-level" page table

4.4.1 Register, Table and Descriptor Formats

We now examine the details of the 680x0 memory management registers and the format of descriptors. We shall also examine such details as how user and supervisor spaces are protected from each other.

The *Translation Control Register* (*TTC*) is used to set up the main parameters in the translation process such as page size and number of levels in the table. The format of the TTC is shown in Fig. **??**.

The fields in the TTC have the following meaning:

E This bit *enables* address translation; if E = 0, the logical addresses are not translated. The E bit is set to zero on reset.

=0.3in figures/ttc.eps

Figure 4.13: The 680x0 TTC Register

S In the preceding discussion, we have assumed a single "root pointer" register. In fact, it is possible to use two separate root pointers: one for user mode accesses; another for supervisor mode. If the S bit is set, both pointers are used. In our examples, we will assume that only a single root pointer is used for both modes of operation.

F The F bit is used to achieve an extra level of table translation. If set, the first level of the address translation tree is indexed by the Function Code bits, not by address bits. In our examples, we assume this bit is not set.

PS These 4 bits determine the page size: 1000 for 256 byte pages to 1111 for 32 Kbyte pages.

IS This 4 bit field indicates the number of most significant bits to *ignore*. For example, if it is set to 8, only the lower 24 bits of the logical address is used. This field is used in small systems. We will always assume it is zero.

TIA–TID These four fields give the number of logical address bits used to index the 4 levels of tables A, B, C, and D. Note that PS+IS+TIA+TIB+TIC+TID must be 32. Any attempt to load the TTC with values inconsistent with this result in an exception. In our two-level table example, we had PS = 12, TIA = 4, TIB = 16, and all the other levels zero. In the 3-level example, we had, PS = 12, TIA = 4, TIB = 8, TIC = 8, and TIC = 0.

The *root pointer* register format is shown in Fig. **??**.
The meaning of the various fields is:

L/U Indicates if the *limit* is an upper limit (L/U = 1) or a lower limit (L/U = 0).

=0.3in figures/root-ptr.eps

Figure 4.14: The 680x0 Root Pointer Register

Limit This 15-bit field gives the upper or lower limit on the size of the table pointed to by the root pointer.

DT The *Descriptor Type* field indicates what the root pointer is pointing to. The possible values are:

OO: Invalid.

O1: Page descriptor. If a page descriptor is found earlier than expected, it is called an "early termination" descriptor and maps a contiguous space of memory.

1O: Points to a 4-byte descriptor. (Descriptor's come in 4- and 8-byte varieties. We consider only 8-byte descriptors here.)

11: Points to an 8-byte descriptor.

Physical Address This 24-bit field gives the starting address of the table the root pointer is pointing to.

Table descriptors are much like root pointers. Their format is given in Fig. **??**. The primary difference between root and table descriptor pointers is the meaning of the bits in the *status field*. These bits are described as follows:

S: When the *Supervisor* bit is set, the CPU must be in supervisor mode for the translation to continue. Otherwise a bus error is generated.

CI: The *Cache Inhibit* bit is used to inform external hardware that any addresses matching this descriptor should not be cached. This would be used, for example. to prevent I/O space from being cached or to disable caching of shared memory in a multi-processing environment from being cached if it controlled system-level resources (e.g. semaphores).

Note that table descriptors

=0.8in figures/table-descriptor.eps

Figure 4.15: The 680x0 Table Descriptor format

=0.8in figures/page-descriptor.eps

Figure 4.16: The 680x0 Page Descriptor format

4.4.2 Physical operation

4.4.3 Kernel software considerations