UNIX AND THE MC68000

BY ANDREW L. ROOD, ROBERT C. CLINE, AND JON A. BREWSTER

A *software perspective on the* MC68000 CPU *architecture and* UNIX *compatibility*

THE MOTOROLA MC68000 CPU architecture is well suited to UNIX implementation; its linear, 32-bit addressing simplifies the system programmer's task by allowing direct access to the entire memory address space at all times. The 68000's dual-state architecture also maps conveniently into the UNIX user and kernel modes. The CPU TRAP instruction provides a disciplined way to move from user to supervisor state during a kernel call, and the RTE (return from exception) instruction simplifies the return from supervisor to user state.

The MC68000 architecture provides a powerful yet simple interrupt organization that includes seven levels of interrupt priority. The multiple priority levels are used in UNIX implementations to help organize the kernel device-driver code in an environment comprised of high-speed mass storage devices and low-speed user I/O devices (terminals).

I/O devices are used in a memory-mapped manner rather than having a set of I/O-specific instructions. This allows the UNIX implementer to write device drivers in a high-level language, manipulating I/O devices as though they were memory-resident data structures.

MC68000 Architecture Overview

In this article, we use the programmer's model rather than the hardware implementation when discussing the MC68000 CPU architecture. The MC68000 is a general-register processor; the CPU incorporates a number of internal registers that can be loaded from main memory, manipulated, and stored in main memory. This is different from a single-accumulator CPU such as the Intel 80286, where most operations happen in one register (the accumulator), or a stack-oriented machine such as the Hewlett-Packard 3000, where all operations occur on the stack. (See figure 1 for a description of the registers.)

Actually, the program counter, stack pointers, and status registers are not considered general registers. The remaining registers are broken into two types, address registers and data registers. Each type has a set of dedicated instructions that enrich its general capability. For example, data registers can handle byte (8-bit), word (16-bit), and long word (32-bit) data. The address registers can be used as software stack pointers and are intended for address calculations.

There are two distinct CPU stack registers (see figure 1), one for each state, and a small set of privileged instructions whose operation is state-dependent. If a user program attempts to execute a privileged instruction, a trap will occur so that the

(continued)

Andrew L. Rood received a B.S. *in mathematics from Stanford University and an* M.S. *in computer science from Oregon State University. He is a Software Research and Development Project Manager at Hewlett-Packard's Corvallis Workstation Operation.*

Robert C. Cline earned a B.S. *in mathematics from the University of Massachusetts and an* M.S. *in computer science from Indiana University. He is a Software Research and Development Engineer at Hewlett-Packard's Corvallis Workstation Operation.*

Jon A. Brewster, who holds a B.S.E.E. *degree from Oregon State University, is a software Research and Development Manager at Hewlett-Packard's Corvallis Workstation Operation.*

The authors can be contacted at Hewlett-Packard, Corvallis Workstation Operation, 1000 Northeast Circle Blvd., Corvallis, OR 97330.

kernel can arbitrate the violation. The CPU architecture provides vectored interrupts and seven levels of interrupt priority. (Note that access to the interrupt-level mask is a privileged facility.) The MC68000 architecture provides no I/O instructions. I/O is presumed to be performed in a memory-mapped manner using the normal LOAD and STORE instructions.

A Quick UNIX Summary

The UNIX system is a multitasking operating system designed for software development. It has become popular because of its simplicity of design and the ease with which it can be ported to a variety of machine architectures. The simplicity and portability are due mostly to the fact that 90 percent of the system's code is written in the high-level language C.

UNIX is a process-oriented system. The management of processes in the UNIX system is fairly simple. Each runnable process (program) is placed in a list. These processes are ordered by a priority system. Each process shares the CPU via a time-slicing, round-robin, scheduling algorithm where the process with the highest priority gets to use the CPU first. The time slicing is normally governed by a periodic interrupt that occurs each time a system clock ticks.

The UNIX system is oriented around two states of operation: operating system, or kernel, state, and user state. All driver activity, process management activity, and low-level file management activity occur in the kernel state. Processes normally run in the user state. Each time a kernel intrinsic (operating system request) is called by a process, a state transition changes the user state to the kernel state. When the intrinsic has finished, the state is changed back to the user state. This provides some insulation of user processes from the system internals.

The memory management primitives in the UNIX system make no assumptions about the sophistication of memory management hardware available to them. UNIX systems have been implemented with no memory management and with sophisticated, paged, segmented systems. Even a memory protection scheme can be dispensed with if the system being designed is to be just an applications engine.

Due to the multiprocessing nature of a UNIX operating system, some sort of memory protection is recommended if you intend to do software development on the machine. This is to prevent experimental software from writing over memory occupied by its neighboring processes and destroying them. It is entirely up to the implementer to say how much memory management power is required. Such a decision boils down to the trade-off between price, performance, and design schedule.

The I/O system of UNIX is designed around two generic device models: the block device and the character device. Block devices are those, such as disk and tape devices, that treat data in blocks. Character devices are those, such as data terminals, that handle data one character at a time. Strict conventions exist in the system as to how devices of each type are to be accessed. Such a well-defined interface makes driver writing easier. Because driver writing makes up the bulk of the work for most operating system ports, the device model of the UNIX operating system makes porting less trying than with many other less organized systems.

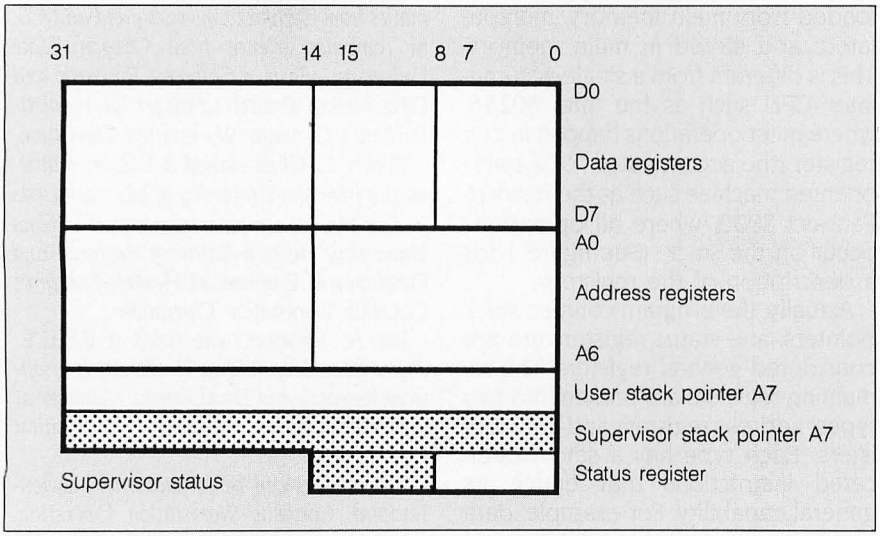

Figure 1: *The* MC68000 CPU *register model.*

Linear Address Space

The MC68000 family of processors has a large linear address space in which each memory address is 32 bits long. This allows up to 4 gigabytes of memory to be directly addressed. Since only 24 address lines are brought out of the MC68000 and MC68010 processors, only 16 megabytes are accessible in those members of the family. (The MC68020 presents all 32 bits of address.) This linear addressing scheme sharply contrasts with segmented addressing schemes, which have a small local address range and a way to relocate the local address range within a larger global addressing range.

Linear addressing may be the single most important feature of the MC68000 processor family. It greatly simplifies the task of implementing an operating system and makes large, complex working sets of segment pointers for managing process images unnecessary. (It is also the single biggest difference between the MC68000 and the Intel 80286 CPUs.)

Theoretically, a system with the address reach of a segmented architecture is as powerful as the MC68000 with its large linear reach. However, it is easier for programmers to use the linear addressing scheme (a programmer in this context is an assembly language programmer or a compiler writer).

The reasons to write in assembly

(continued)

language are varied, but it often comes down to speed, space optimization, and minimal complexity. Keeping the compiler writer's job easy has paid off in two ways: first, high-quality compilers were available early in the MC68000 life cycle; second, the compilers have been of high enough quality to reduce the amount of assembly language programming.

One drawback of linear addressing is that the code density of the actual machine instructions may suffer from the presence of many long addresses (the top byte of which is nearly always 0). This problem is alleviated somewhat by short (16-bit) relative addressing modes. However, UNIX implementations prefer separate instruction and data areas, and, consequently, the relative addressing modes are used infrequently.

Address space management becomes easy with a large, linear address space. The compiler needs to be concerned with only three spaces: code, data, and stack. Also, the operating system can quickly allocate and deallocate any number of easily accessible memory fragments. Very large processes such as those for expert systems can be designed easily because you don't have to be concerned about the overhead involved in passing control between routines in different code segments.

The same can be said for the use of large data structures. UNIX includes many application programs whose memory image exceeds 1 megabyte and that use individual data structures several hundred kilobytes in size. There is also a class of recursive programs that are heavy stack users. A machine such as the 80286 with a 64K-byte stack limit can be a real disadvantage in this type of application.

Another nice feature of a large address space is that parts of it can be easily reserved for other functions. An example is mapping I/O device interfaces into memory. Many megabytes of address space may be allocated for I/O without impacting other needs. (Segmented-architecture machines typically provide a separate I/O address space to keep the intersegment communication to a minimum.) Figure 2 shows an example of address allocation.

Dual Processor States

The MC68000 is a dual-state processor. The processor has two stack registers, one used while in the supervisor state and the other used while in the user state. When an instruction is executed that changes the CPU state, the stack is automatically changed as well. The CPU provides status lines coming off-chip to allow support chips (such as an external memory management unit) to determine the current CPU state.

The state of the CPU can be determined by reading the supervisor state bit in the CPU status register. There is a small set of privileged instructions, including STOP, RESET, RTE, MOVE to SR, AND, EOR, OR, and MOVE (word) IMMEDIATE to SR, that operate differently in user state than in supervisor state. In user state they cause a privilege violation and associated CPU fault. In supervisor state they modify the CPU as indicated. Note that any instruction that explicitly sets the supervisor portion of the status register (SR) is a privileged instruction.

Interrupts are dispatched in supervisor state even if the CPU was processing in user state when the interrupt occurred.

UNIX Use of Dual States

In UNIX a process is a user task that has resources allocated to it, including main memory, CPU time, I/O processing, and space in system tables. It is by time-slicing processes that UNIX performs multitasking.

Figure 3 shows a memory-oriented picture of a process. Normal processes have a kernel and user side, each of which contains code, data, and stack. Process 0, the original process, acts as the scheduler in the system. Process 0 is unique in that it has no user side, only a kernel side.

A zombie process is a process that has no user side and no kernel side; in fact, a zombie process has no memory allocated to it at all. It is simply an entry in the process table waiting to be deallocated.

UNIX kernel code normally executes in the supervisor state of the CPU, and the UNIX user code normally executes in the user state of the CPU. In the UNIX framework there is a distinguished user called the superuser. The concept of a superuser should not be confused with the supervisor or kernel portion of a process. A superuser process is a normal process from the CPU perspective,

(continued)

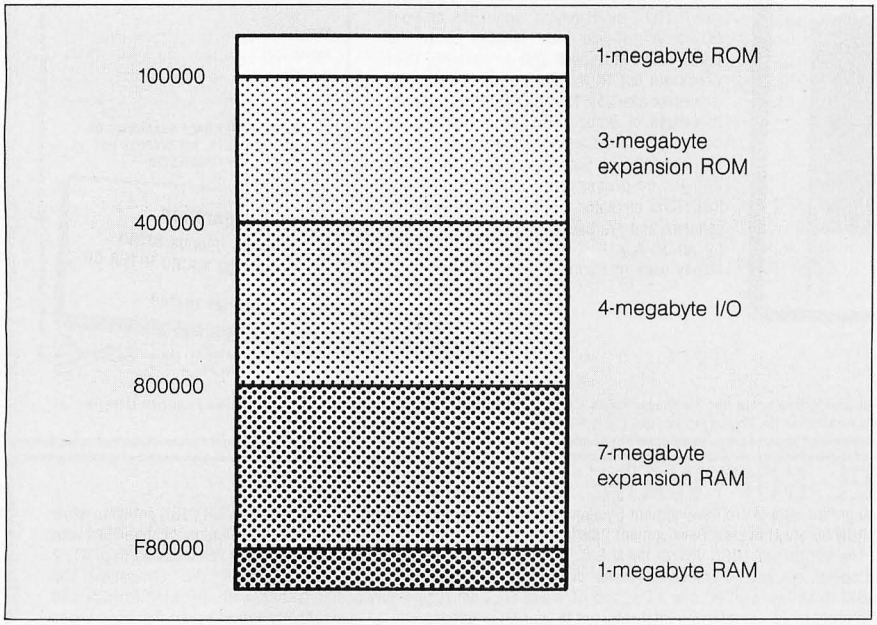

Figure 2: A *typical* MC68000 *address allocation scheme.*

but one that is distinguished by the superuser process identification number (pid). The pid is checked by the operating system during certain critical operations, for example, mounting new file-system devices.

Only superuser processes are allowed to perform these operations. Any process (even those belonging to the superuser) has both a kernel side, which executes in the supervisor state of the CPU, and a user side, which executes in the CPU's user state.

In contrast to the dual privilege levels associated with UNIX, the 80286 processor provides four levels of privilege, each protected from the levels above it. For more sophisticated operating systems than UNIX this four-state CPU can provide a good implementation vehicle. In the UNIX environment it is not immediately clear that four CPU states are required or even useful.

TRAP DISCIPLINE

The TRAP instruction can be used to implement kernel calls and the associated state change from user to supervisor. A matching RTE instruction is used to return to the user state. The register environment can be used to pass information to the kernel, such as the type of kernel call being made. (The registers must be saved explicitly by the kernel as desired.)

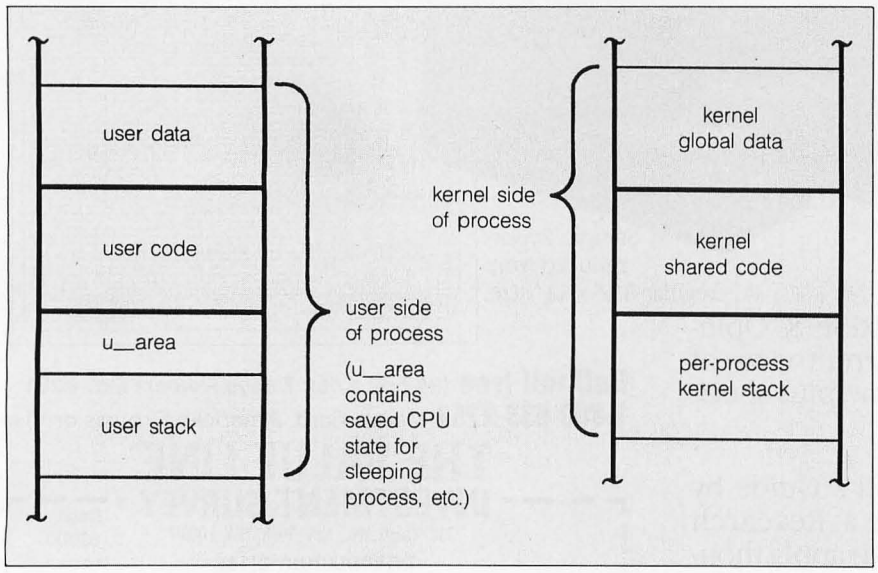

Figure 3: A *graphic description of a process, showing user and kernel elements.*

THE USER VIEW OF A KERNEL CALL USING TRAP

```
;
; Syscall(callno,arg1,arg2)
;
;       Perform the kernel call indicated by callno.
;       Send arg1 and arg2 (typically pointers) to the
;       kernel as part of the call.
;
Syscall move.l  4(sp),d0      ; Get the kernel call number.
        move.l  8(sp),d1      ; Get the first argument (arg1).
        move.l  12(sp),d2     ; Get the second argument from
                              ; the stack.

        trap    #1            ; Trap to supervisor state.

        rts
```

The kernel call is a frequently used operation, but typically a costly one. Overhead involved in trapping to the kernel includes TRAP and RTE instructions, register state save and restore, kernel call decode and dispatch, and kernel call implementation. See the text boxes "The User View of a Kernel Call Using TRAP," below, and "The Kernel View of a Kernel Call" on page 186.

The 80286 processor implements a more complex state-transition mechanism using call gates to allow procedures to transfer control to the same or more privileged levels. This scheme is more complex than the MC68000 implementation of CPU state transition, and it is exactly this added complexity that makes UNIX implementation with the 80286 more difficult.

UNIX CONTEXT SHIFTING

Context shifting in the UNIX system is the operation of putting the currently executing process to sleep and resuming the execution of some other process. Because of the process orientation of UNIX, and its time-slicing algorithms for scheduling, context shifting is frequently performed.

Context shifting occurs either at the end of a kernel call or in response to some interrupt during user state processing (e.g., the heartbeat interrupt from the system clock). Context-shift overhead includes saving the currently executing process state, selecting a new process to run, and restoring the state of the new process. (See the text boxes "CPU State Save in a Context Shift" on page 188 and "CPU State Resume in a Context Shift" on page 190.)

The 80286 provides a complex task environment that allows high-performance context shifting between tasks. As with the privilege architecture of the 80286, the task separation mechanism is more powerful than is required in a UNIX implementation.

KERNEL VIEW OF CONTEXT SHIFTING

The kernel view of context shifting involves two functions: sleep and switch (see figure 4). Sleep is a function that,

(continued)

The Kernel View of a Kernel Call

```
;
; trap
;
;       The address of this routine is placed in memory at the
;       address of vector 33 (the TRAP 1 vector); that is, at
;       memory location 132 decimal. When the TRAP #1 instruction
;       is executed, the CPU generates an internal exception and
;       automatically transfers to this routine in supervisor
;       state. The user return address and status register are
;       saved on the supervisor stack.
;
;       On entry it is presumed that the registers contain:
;               d0 . . . the kernel call number
;               d1 . . . first argument to the kernel call
;               d2 . . . second argument to the kernel call
;       Typically the arguments are pointers into the user's data
;       space.
;
trap    movem.l #0xFFFE,-(sp)     ; Save registers d0-d7 and a0-a6.
        move.l  usp,a0
        move.l  a0,-(sp)          ; Save the user stack pointer.

        move.l  d0,-(sp)
        move.l  d1,-(sp)
        move.l  d2,-(sp)          ; Push the incoming parameters so
                                  ; a high-level language routine
                                  ; can access them using standard
                                  ; compiler parameter passing
                                  ; techniques.

        jsr     C-trap            ; Call a C handler to perform
                                  ; the desired kernel call.
;
; After the kernel call, this process may be context-shifted
; out. We have saved the user stack pointer with the register
; set so that we can do a complete restore later after some
; other user process has run.
;
        add.l   #12,sp            ; Strip the parameters off
                                  ; the stack.

        move.l  (sp)+,a0          ; Get the saved user stack
                                  ; pointer.

        move.l  a0,usp            ; Restore user stack pointer.

        movem.l +(sp),#0x7FFF     ; Restore registers a6-a0 and
                                  ; d7-d0.

        rte                       ; Return from exception back
                                  ; to the user, restoring the
                                  ; CPU status and returning
                                  ; to user state of the CPU.
```

(continued)

when called, returns after some interval, during which other processes may have executed. Switch is the function that actually performs the selection of a new process and transfers control to that process.

The kernel also provides a wakeup function. Wakeup is used to make a process (which has been sleeping, waiting for some external event) ready to run. This is typically after the event has occurred.

Figure 4 shows sleep in a context switch. Sleep calls switch, which in turn performs two operations; it saves the current process's state and then selects the next process to run and runs it. Switch uses two critical functions to perform this: save and resume. Save is a function that, when called once, returns twice. Resume is a function that never returns. Resume actually transfers to some other save, acting as the second return for that save call.

Figure 5 shows an example of control flow from process A to process B and back again to process A. The execution of a sleep call in process A eventually calls switch, which calls save. The save call first returns a 0 value. This call also saves the state of process A for later resumption. Control after the first return from the save call continues into a resume call. This transfers control to process B. Later, process B will itself execute a resume call that transfers control back to process A. This resume (from B) acts as the second return of the save call in process A. This time the save returns the value 1, which causes switch to return to the sleep call, which returns to the user process.

Save's primary responsibility is to save the CPU state so that it can be included as part of the saved process image. The image can later be resumed.

UNIX Fork Paradigm

In UNIX all processes are created by making copies of existing processes, except, of course, the first process, which is handcrafted by the kernel during system boot-up. This process then splits in two using a fork( ) operation. Each child of this first process in turn can divide as many times as needed to create processes to perform all tasks desired.

To perform a new task, a child process may overlay itself with a new program image using an exec kernel call. Figure 6 shows the fork operation pictorially and demonstrates why the term fork was chosen for this operation.

(continued)

CPU State Save in a Context Shift

```
;
; save(save-area)
;
;       The function save( ) saves the CPU register state
;       in preparation for a later resume call. Registers
;       and user stack pointer are saved in save-area[17].
;       This save call returns 0, indicating it is the first
;       return from save. When resume simulates the second
;       return from save, it will return 1.
;
;       The save-area will look like:
;
;
;                       user stack ptr                  [16] = usp
;
;                                                       [15] = a7
;                       saved registers                 [14] = a6
;                       a7,a6,a5, . . . ,a1,a0          . . .  . . .
;                       d7,d6,d5, . . . ,d1,d0          [1]  = d1
;                                       <---save-area   [0]  = d0
;
;

save    move.w  #0x2700,sr      ; Go to high-priority supervisor
                                ; state to prevent interrupts during
                                ; this function call. This also clears
                                ; the rest of the status bits.

        move.l  4(sp),a0        ; Get address of register save-area.

        movem.l #0xFFFF,(a0)+   ; Save all registers.

        move.l  usp,a1          ; Get user stack pointer.

        move.l  a1,(a0)         ; Save the user stack pointer.
;
; At this point other processing may occur, for example, the saving
; of other types of process description information, especially
; the stack image that contains the return address for this call.
;

;
; Note that a1, a0, and d0 are used as working registers here although
; they are also saved with the rest of the registers.
;
        move.l  #0x0,d0         ; The return value for this save
                                ; call is 0. For this example the
                                ; return value is presumed to be in
                                ; d0 upon return from function call.

        move.w  #0x2000,sr      ; Go to low-priority supervisor
                                ; state and enable interrupts.
        rts
```

CPU State Resume in a Context Shift

```
;
; resume(save-area, . . . . )
;
;       Resume the process whose saved CPU state is at save-area.
;
resume  move.w  #0x2700,sr      ; Go to high-priority supervisor
                                ; state so you don't get interrupted
                                ; during this state transition.

        move.l  4(sp),a0        ; Get the pointer to the CPU saved
                                ; state.
;
; Here you may access other parameters, such as memory management
; information about how to map the new process.
;
; Before resuming the new process, its image must be appropriately
; restored (and possibly mapped using an MMU).
;
        movem.l (a0)+,#0xFFFF   ; Restore all CPU registers.
                                ; This includes a7, the
                                ; supervisor stack pointer that
                                ; still points to the return
                                ; address of the original call.

        move.l  (a0),a1         ; Get the user stack pointer.

        move.l  a1,usp          ; Restore the user stack pointer.

        move.l  #0x1,d0         ; Get a 1 as the return value for
                                ; the upcoming simulated return
                                ; to save( ).

        move.w  #0x2000,sr      ; Go to low-priority supervisor
                                ; state, that is, reenable
                                ; interrupts.

        rts                     ; Return using the return address
                                ; copy from the original call
                                ; to save.
```

The programmer's view of the fork call is that of a function, fork( ), which, when called once, returns twice. (See the text box "A Programmer's View of Fork( )" on page 196). It returns once in the calling process with the pid of the new process created. It returns a second time in a new process that is an exact copy of the original except that the new process has a different pid number and the fork call returns the value 0.

During the context shift operation the kernel function save is used to save the process state and prepare a second flow of control to be used in the new process. Rather than terminating in a resume call, the first return from the save continues as the control flow in the parent process. (See the text box "A Kernel View of Fork( )" on page 198.)

Interrupt Handling and UNIX

The MC68000 processor provides a great deal of flexibility for the system

(continued)

designer in handling interrupts. As with most processors, all interrupts can be handled in a vectored fashion. This is where each interrupt, internal and external, is associated with a location in memory (the vector address) that holds the address of the handling routine for that particular interrupt.

When an interrupt occurs, the appropriate location in memory or vector is used to fetch the starting address of the appropriate interrupt handler, and control is transferred to it (this is also how things work on the Intel 80286 processor). Figure 7 shows how vectored interrupts access interrupt handlers.

```
sleep( )
{
  . . .
  switch( )
  . . .
}

switch( )
{
  . . .
  if (save(proc) == 1)
    return
  . . .
  <select new process>
  . . .
  resume(new process)
}
```

Figure 4: Sleep *in a context switch.* Sleep *calls* switch, *which saves the current process state and then selects the next process to run and runs it.*

But this is all the 80286 really offers; the MC68000 offers a seven-level processor priority scheme for selectively masking interrupts. Each interrupting device is assigned a priority level. When the processor is set to a particular priority level, all interrupts at that level and below are held off. This allows higher-level, more critical interrupts to supersede the handling of lower-level, less critical interrupts. The Intel 80286 allows only for the turning on and off of all interrupts simultaneously, which is much less flexible.

This selective masking of interrupts is very useful to the UNIX system. Consider a low-priority event, the scheduling event triggered by a periodic interrupt from the system clock. While the scheduling of a process must occur in the multiprocessing UNIX system, it is not crucial that it be done at a precise moment. Any time the scheduling process is taking place, another interrupt can be handled without harm to the overall operation of the system.

Consider also a high-priority event, the disk interrupt indicating that a sector on the disk has been located. If this interrupt is not serviced immediately, the disk will rotate beyond the desired sector and time will be lost waiting for it to come around again. Such delays could impact system performance greatly. This is especially true with UNIX because it is traditionally a disk-based system. It is easy to see that you would want the disk interrupt to supersede the handling of process-scheduling chores.

On the MC68000 this is easily accomplished by making the disk interrupt a higher priority than the clock interrupt. This allows the disk interrupt to supersede the scheduling process and ensure that it is serviced immediately to prevent any disk performance loss.

On processors where interrupts cannot be selectively masked by priority, the disk interrupt would have to be held off while the scheduling interrupt is handled. Unless external interrupt priority hardware was provided, the disk sector would be missed and system performance would suffer. From the preceding example you can see the advantages of the MC68000 interrupt system for UNIX implementations.

You can also see that it is not necessarily just an advantage for a UNIX system. The situation described in the example could arise in many systems today. The MC68000 interrupt architecture provides the flexibility necessary to build an efficient interrupt control system.

```
Process A executing
  . . .
  sleep( )
    --> switch( )
          . . .
            --> save(A) . . .[saves the CPU state of process A]
            <-- returns . . .[on first return save( ) returns 0]
          . . .
          <select process B>
          . . .
          resume B ---------> Process B executing
                                . . .
                                sleep( )
                                  --> switch( )
                                        . . .
                                          --> save(B)
                                          <-- returns 0
                                        . . .
                                        <select process A>
                                        . . .
                    <---------------- resume(A)
                  <-- returns . . . [on second return (from a resume) ]
                                    [save( ) returns 1                ]
      <-- returns
Process A continues execution
```

Figure 5: *Context shift control flow.*

C Language Compatibility

UNIX, the MC68000, and C are all well suited for each other. C itself is a good systems programming language. In fact, 90 percent of the UNIX operating system and most of the user programs for UNIX are written in C. Since the MC68000 allows for a very efficient C compiler, this in turn provides an efficient UNIX implementation.

The compiler writer's task is made easy by the MC68000. The portable

(continued)

C compiler was designed for a register machine with two types of general-purpose registers. Setting up the data and address registers as these two types works very well. The MC68000's nearly orthogonal instruction set helps keep code generation straightforward. (An orthogonal instruction set is one in which any registers can be used with any instructions and with any addressing modes.)

The result of having a CPU that supports an efficient compiler is a code density (number of assembly instructions per C statement) that is good enough for libraries, system work, and even I/O drivers. The C programmer does see the CPU peeking through now and then even though C is a high-level language. For example, proper use of the register statement requires CPU knowledge.

C on the MC68000 also allows for efficient pointer/integer arithmetic. This is due to the 32-bit linear address space. An integer and a pointer are both 32-bit quantities.

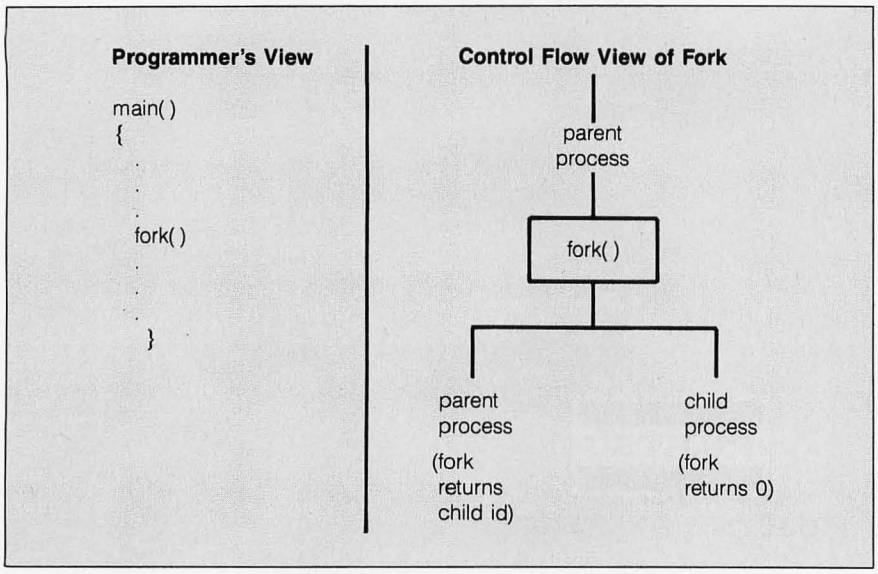

Figure 6: *The programmer's view of a* fork( ) *call.*

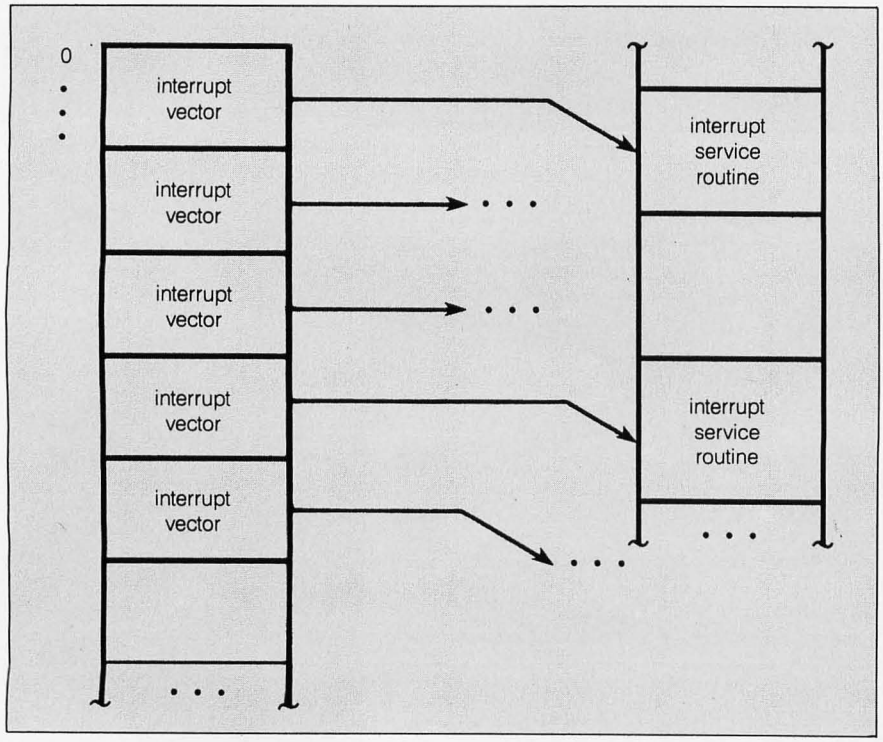

Figure 7: *Vectored interrupts.*

Missing Features

The MC68000 has some weaknesses with regard to UNIX implementation, primarily the omission of features that are typically implemented separately from the CPU using other VLSI support or discrete implementations. One point in favor of the MC68000 is that its architecture generally does not prohibit or impede external implementation of missing features.

The most glaring omission in the MC68000 family is memory management. The MC68000 philosophy is to do nothing for memory management rather than do a partial job. The MC68000 CPU generates 32-bit *logical* addresses. If these are to be translated into physical addresses, off-processor hardware to perform the translation must be provided. (Memory management typically consists of address translation and address bounds checking.)

First, consider address translation. Address translation involves a distinction between logical addresses (name-space addresses used in a program) and physical addresses, those that are presented to the main memory. Figure 8 demonstrates this logical/physical distinction using a system block diagram. Address translation allows several processes to use an identical logical address space while using completely separate portions of physical address space. This permits several copies of the same process to coexist in one physical memory.

Because UNIX uses the fork paradigm to duplicate user processes and implement multitasking, some form of memory management is highly desirable, and here there is an advantage to the MC68000 philosophy. Since there is no memory management on board the MC68000, UNIX implementers are forced to choose

(continued)

their own style of memory management hardware and can select the trade-off between performance, price, and design schedule that is best for their target market.

MC68000 UNIX implementations on the market range in memory management unit complexity from no extra hardware to an MMU providing segmentation and paging. Memory management also involves memory protection, often implemented by bounds registers that are compared to user logical addresses. Memory protection allows several processes to coexist in the same physical memory without overwriting each other or reading each other's data.

A Programmer's View of Fork( )

```
main( )
{
        /*
         * Simple fork example program.
         */
        int child-pid;

        /* . . . */

        child-pid = fork( );

        if (child-pid == 0)
          {
            /*
             * This is executing in the child process.
             */
            printf(" Hi from the child process. \ n");
            /*
             * At this point the child process may
             * overlay itself with another command
             * using an exec call; the fork-exec sequence
             * is the "normal" way to start a new program.
             *
             * Here the new program "factorize" is called
             * to overlay the child process image.
             */
            execv("factorize",argv);
          }
        if (child-id > 0)
          {
            /*
             * This is executing in the parent process.
             */
            printf(" Hello from the parent process. \ n");
            printf(" My child's id is %d \ n",child-pid);
          }

        if (child-pid == -1)
          {
            /*
             * This is executing in the parent process,
             * but the fork( ) failed, so there is no
             * child process.
             */
            printf(" FORK FAILED! Awk! \ n");
          }
}
```

Memory protection is highly desirable in a multiuser environment to protect users from each other. Although the MC68000 itself does not provide any form of memory protection, it does incorporate VLSI MMUs, the MC68451 and the MC68851. The MC68451 MMU provides memory translation in a segmented fashion and memory protection associated with each segment. The MC68851 provides address translation and bounds checking in a paged environment.

The 80286 provides two modes of addressing: Real Address Mode and Protected Virtual Address Mode. When operating in Real Address Mode, the CPU issues logical addresses, as does the MC68000. When operating in Protected Virtual Address Mode the 80286 performs memory translation and bounds and privilege checking on board the CPU. It is possible to use this virtual address mode to implement process separation in UNIX. This area is one in which the 80286 architecture makes implementation of UNIX easy, in that the implementer is not required to add external memory management hardware.

Interruptibility

The 68000 CPU is not restartable for all instructions. If the execution of a given instruction causes a fault (e.g., because of an MMU bounds violation), that instruction cannot necessarily be restarted transparently. Transparently here implies that the restart occurs in such a manner that the interrupted process cannot detect that the interrupt and restart have occurred.

Some MC68000 instructions can be aborted in such a manner that the internal CPU status cannot be recovered and restored. Such instructions cannot be restarted transparently. In a paged memory environment a CPU exception is typically generated by an MMU when a desired page is not in main memory. If the CPU cannot be restarted following the abortion of any instruction, the CPU cannot be used reliably in a paged memory environment.

(continued)

A KERNEL VIEW OF FORK( )

```
/*
 * Newproc is the internal, kernel form of the fork( )
 * system call. Newproc creates a duplicate of the
 * calling process and introduces it to the system.
 */
newproc( )
{
 /* . . . */

 /*
  * The new process memory image is duplicated. A new
  * entry in the procedure table is allocated and it is
  * filled with information about this new process.
  */
 /* . . . */

 if (save(save-area) == 1)
   {
     /*
      * When save returns with the value 1 (after a
      * corresponding call to resume) this copy of the
      * process continues execution. The two processes
      * are duplicates of each other, but traditionally
      * this process is the parent process. This
      * process is the one that originally called
      * fork( ) and is the one to which the kernel
      * will return the process number of the child.
      */
     /* . . . */
     return(1);
   }
 /*
  * This flow of control will become the child process.
  * At this point any additional processing that is
  * necessary to make the child ready to run is performed.
  * The kernel will return a 0 value to the fork( )
  * call for this flow of control.
  */
 /* . . . */
 return(0);
}
```

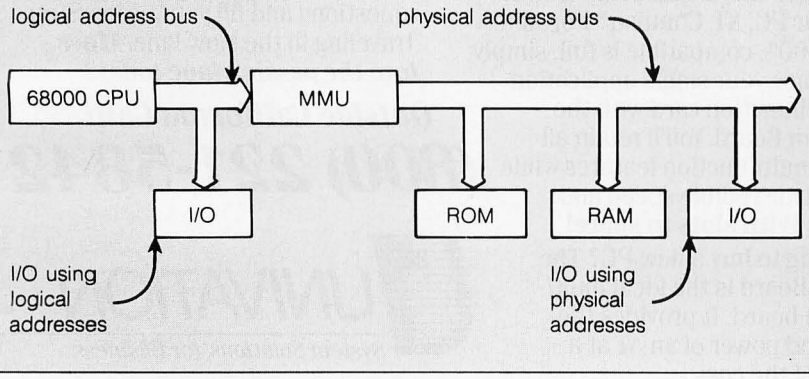

Figure 8: *Address translation.*

Bounds checking can also be used to detect user stack underflow and allow stack growth and process restart. The user memory bounds are set to the end of the user's stack. When an underflow occurs, the MMU generates a bounds violation. The exception handler for the bounds violation can allocate more memory for stack, remap the process, and restart the process. If the CPU cannot be restarted, the process cannot be guaranteed to recover properly after the stack region is expanded by the kernel.

The MC68010 and MC68020 do allow restart of any instruction. On MC68000 UNIX implementations it is possible to implement automatic stack growth using software conventions. A simple test is performed in the preamble to every procedure call to see if sufficient stack is available. This can be accomplished by touching the deepest possible region of use on the stack, using a dummy TST.B instruction. If the test fails, causing a CPU exception and kernel allocation of more stack, the process can be restarted at the instruction after the dummy TST.B instruction. Since this is a software convention, all programs wishing automatic stack expansion must conform to the convention to operate properly.

The text box "Recovery and Restart After a Bus Exception" on page 200 shows a typical bus error exception handler that calls a high-level language routine to implement user stack expansion and process remapping.

COPROCESSORS

The MC68000 family allows for multiple CPU environments. In the UNIX environment there is the possibility for much parallel processing if multiple processors are available. The MC68000 bus operates asynchronously. This aids in configuring buses with a multimaster capability that support multiple CPUs using the same memory and peripherals. Many types of coprocessors are common to UNIX implementations, including floating-point coprocessors, graphics coprocessors, and DMA (direct memory ac-

(continued)

Recovery and Restart After a Bus Exception

```
;
; The address of this routine is placed in ROM at the bus address
; vector location: vector number 2, memory address 8. Control
; transfers to this location at bus error time.
;
; Typically an MMU will cause a bus error when a bounds violation
; occurs (e.g., during stack underflow or page fault).
;
; At entry the stack will look like this:
;
; sp + 14
;
; sp + 12       Program
;               Counter          . . . Restore during process restart
; sp + 10
;    .          Status Register  . . . Restore during process restart
;    .
;    .          Instruction      . . . Examine; verify stack underflow
;               Register
;                                . . . Examine; verify stack underflow
;               Access Address         and determine how much memory
;                                      to allocate
; sp + 2
;                                . . . Function Code; I/N bit; R/W bit
; sp—>
;
bus-fault     movem.l  #0xFFFE,-(sp)    ; Save d0-d7 and a0-a6.
                                        ; These will also be
                                        ; available to C-bus-handler.

              jsr      C-bus-handler
;
; A high-level language routine can examine the access address
; and instruction register to determine the reason for the
; fault. If the instruction being executed was a TST.B
; instruction and the access address is "slightly" below
; the stack, this may represent a stack underflow that
; requires stack expansion and process remapping.
; If this was not an implicit stack expansion request,
; the user process should be sent a signal indicating that
; the bus error occurred.

; Here the PC should be decremented and the aborted instruction
; restarted. Since the proper status cannot be assured on a
; 68000 CPU, the restart is not attempted. On the presumption
; that the bad instruction was a dummy TST.B, bus-fault simply
; restores what status it can and returns to the user.
;
              movem.l  (sp)+,#0x7FFF    ; Restore the registers.

              addq.l   #8,sp            ; Strip fcode, access address,
                                        ; and instruction register
                                        ; off the stack.

              rte
```

cess) processors. Because UNIX is a multitasking system with multiple processes executing at any time, the operating system easily manages multiprocessor environments.

For example, the operating system may initiate a disk transfer on behalf of a process, start the operation on a DMA coprocessor, put that process to sleep, and continue executing another process. Similarly, processes may be put to sleep while graphics coprocessors operate on a user's terminal. The MC68000 family includes the MC68881 floating-point coprocessor, which can be used effectively in a UNIX environment.

Summary

The MC68000 architecture provides a powerful yet simple programmer's model that makes UNIX implementation easy. The difficult programmer's model is a primary weakness of the typical segmented architectures in use in today's UNIX microcomputers, and in overcoming that, the MC68000 became a good choice for UNIX implementation. There are also other good choices, such as those UNIX products on the market that use segmented architectures and others that use stack architectures. Still, a majority of implementations use the MC68000 family.

The MC68000 has many architectural features that help UNIX implementation, including a large linear address space, dual-state processor, and a seven-level interrupt priority scheme. The processor architecture is conducive to using the C programming language, which is the primary development language for UNIX. Further, the MC68000 is a general-register processor that makes UNIX implementation easy because UNIX was originally developed on machines with similar architectures.

The weak side of the MC68000 architecture involves the omission of helpful features, in particular on-board memory management. This desirable feature represents the main advantage of the 80286, but when all factors are taken into account, it is apparent that the architecture of the Motorola MC68000 CPU processor is very well suited to UNIX implementation. ■