agement begins after printing a message on the console listing total memory, memory in use by MINIX 3, and available memory:

```
Physical memory: total 63996 KB, system 12834 KB, free 51162 KB.
```

The next function is *get_nice_value* (line 18263). It is called to determine the "nice level" of each process in the boot image. The *image* table provides a *queue* value for each boot image process which defines on which priority queue the process will be scheduled. These range from 0 for high priority processes like *CLOCK* to 15 for *IDLE*. But the traditional meaning of "nice level" in UNIX systems is a value that can be either positive or negative. Thus *get_nice_value* scales the kernel priority values on a scale centered on zero for user processes. This is done using constants defined as macros in *include/sys/resource.h* (not listed), *PRIO_MIN* and *PRIO_MAX*, with values of -20 and +20. These are scaled between *MIN_USER_Q* and *MAX_USER_Q*, defined in *kernel/proc.h*, so if a decision is made to provide fewer or more scheduling queues the *nice* command will still work. *Init*, the root process in the user process tree, is scheduled in priority queue 7 and receives a "nice" value of 0, which is inherited by a child after a fork.

The last two functions contained in *main.c* have already been mentioned in passing. *Get_mem_chunks* (line 18280) is called only once. It takes the memory information returned by the boot monitor as an ASCII string of hexadecimal base:size pairs, converts the information into units of clicks, and stores the information in the *mem_chunks* array. *Patch_mem_chunks* (line 18333) continues building the free memory list, and is called several times, once for the kernel itself and once for *init* and each of the system processes initialized during the main loop of *pm_init*. It corrects the raw boot monitor information. Its job is easier because it gets its data in click units. For each process, *pm_init* is passed the base and size of the text and data allocations for that process. For each process, the base of the last element in the array of free blocks is increased by the sum of the lengths of the text and data segments. Then the size of that block is decreased by the same amount to mark the memory for that process as in use.

4.8.3 Implementation of FORK, EXIT, and WAIT

The fork, exit, and wait system calls are implemented by the procedures *do_fork*, *do_pm_exit*, and *do_waitpid* in the file *forkexit.c*. The procedure *do_fork* (line 18430) follows the steps shown in Fig. 4-36. Notice that the second call to *procs_in_use* (line 18445) reserves the last few process table slots for the superuser. In computing how much memory the child needs, the gap between the data and stack segments is included, but the text segment is not. Either the parent's text is shared, or, if the process has common I and D space, its text segment is of zero length. After doing the computation, a call is made to *alloc_mem* to get the memory. If this is successful, the base addresses of child and parent are

converted from clicks into absolute bytes, and sys_copy is called to send a message to the system task to get the copying done.

Now a slot is found in the process table. The test involving *procs_in_use* earlier guarantees that one will exist. After the slot has been found, it is filled in, first by copying the parent's slot there, and then updating the fields *mp_parent*, *mp_flags*, *mp_child_utime*, *mp_child_stime*, *mp_seg*, *mp_exitstatus*, and *mp_sigstatus*. Some of these fields need special handling. Only certain bits in the *mp_flags* field are inherited. The *mp_seg* field is an array containing elements for the text, data, and stack segments, and the text portion is left pointing to the parent's text segment if the flags indicate this is a separate I and D program that can share text.

The next step is assigning a PID to the child. The call to *get_free_pid* does what its name indicates. This is not as simple as one might think, and we will describe the function further on.

Sys_fork and *tell_fs* inform the kernel and file system, respectively, that a new process has been created, so they can update their process tables. (All the procedures beginning with *sys_* are library routines that send a message to the system task in the kernel to request one of the services of Fig. 2-45.) Process creation and destruction are always initiated by the PM and then propagated to the kernel and file system when completed.

The reply message to the child is sent explicitly at the end of *do_fork*. The reply to the parent, containing the child's PID, is sent by the loop in *main*, as the normal reply to a request.

The next system call handled by the PM is exit. The procedure *do_pm_exit* (line 18509) accepts the call, but most of the work is done by the call to *pm_exit*, a few lines further down. The reason for this division of labor is that *pm_exit* is also called to take care of processes terminated by a signal. The work is the same, but the parameters are different, so it is convenient to split things up this way.

The first thing *pm_exit* does is to stop the timer, if the process has one running. Then the time used by the child is added to the parent's account. Next, the kernel and file system are notified that the process is no longer runnable (lines 18550 and 18551). The sys_exit kernel call sends a message to the system task telling it to clear the slot used by this process in the kernel's process table. Next the memory is released. A call to *find_share* determines whether the text segment is being shared by another process, and if not the text segment is released by a call to *free_mem*. This is followed by another call to the same procedure to release the data and stack. It is not worth the trouble to decide whether all the memory could be released in one call to *free_mem*. If the parent is waiting, *cleanup* is called to release the process table slot. If the parent is not waiting, the process becomes a zombie, indicated by the *ZOMBIE* bit in the *mp_flags* word, and the parent is sent a *SIGCHILD* signal.

Whether the process is completely eliminated or made into a zombie, the final action of *pm_exit* is to loop through the process table and look for children of the

process it has just terminated (lines 18582 to 18589). If any are found, they are disinherited and become children of *init*. If *init* is waiting and a child is hanging, *cleanup* is then called for that child. This deals with situations such as the one shown in Fig. 4-45(a). In this figure we see that process 12 is about to exit, and that its parent, 7, is waiting for it. *Cleanup* will be called to get rid of 12, so 52 and 53 are turned into children of *init*, as shown in Fig. 4-45(b). Now we have the situation that 53, which has already exited, is the child of a process doing a wait. Consequently, it can also be cleaned up.

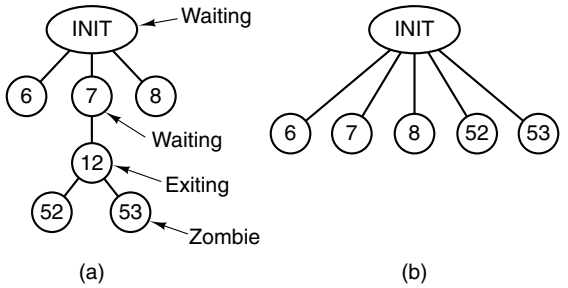

Figure 4-45. (a) The situation as process 12 is about to exit. (b) The situation after it has exited.

When the parent process does a wait or a waitpid, control comes to procedure *do_waitpid* on line 18598. The parameters supplied by the two calls are different, and the actions expected are also different, but the setup done in lines 18613 to 18615 prepares internal variables so *do_waitpid* can perform the actions of either call. The loop on lines 18623 to 18642 scans the entire process table to see if the process has any children at all, and if so, checks to see if any are zombies that can now be cleaned up. If a zombie is found (line 18630), it is cleaned up and *do_waitpid* returns the *SUSPEND* return code. If a traced child is found, the reply message being constructed is modified to indicate the process is stopped, and *do_waitpid* returns.

If the process doing the wait has no children, it simply receives an error return (line 18653). If it has children, but none are zombies or are being traced, a test is made to see if *do_waitpid* was called with a bit set indicating the parent did not want to wait. If not (the usual case), then a bit is set on line 18648 to indicate that it is waiting, and the parent is suspended until a child terminates.

When a process has exited and its parent is waiting for it, in whichever order these events occur, the procedure *cleanup* (line 18660) is called to perform the last rites. Not much remains to be done by this point. The parent is awakened from its wait or waitpid call and is given the PID of the terminated child, as well as its exit and signal status. The file system has already released the child's memory, and the kernel has already suspended scheduling and freed up the child's slot in the process table. At this point, the child process is gone forever.

4.8.4 Implementation of EXEC

The code for exec follows the outline of Fig. 4-40. It is contained in the procedure *do_exec* (line 18747) in *exec.c*. After making a few validity checks, the PM fetches the name of the file to be executed from user space (lines 18773 to 18776). Recall that the library procedures which implement exec build a stack within the old core image, as we saw in Fig. 4-38. This stack is fetched into the PM's memory space next (line 18782).

The next few steps are written as a loop (lines 18789 to 18801). However, for ordinary binary executables only one pass through the loop takes place. We will first describe this case. On line 18791 a message to the file system switches to the user's directory so the path to the file will be interpreted relative to the user's, rather than to PM's, working directory. Then *allowed* is called—if execution is allowed it opens the file. If the test fails a negative number is returned instead of a valid file descriptor, and *do_exit* terminates indicating failure. If the file is present and executable, the PM calls *read_header* and gets the segment sizes. For an ordinary binary the return code from *read_header* will cause an exit from the loop at line 18800.

Now we will look at what happens if the executable is a script. MINIX 3, like most UNIX-like operating systems, supports executable scripts. *Read_header* tests the first two bytes of the file for the magic **shebang** (#!) sequence and returns a special code if this is found, indicating a script. The first line of a script marked this way specifies the interpreter for the script, and possibly also specifies flags and options for the interpreter. For instance, a script can be written with a first line like

```
#! /bin/sh
```

to show it is to be interpreted by the Bourne shell, or

```
#! /usr/local/bin/perl –wT
```

to be interpreted with Perl with flags set to warn of possible problems. This complicates the job of exec, however. When a script is to be run, the file that *do_exec* must load into memory is not the script itself. Instead the binary for the interpreter must be loaded. When a script is identified *patch_stack* is called on line 18801 at the bottom of the loop.

What *patch_stack* does can be illustrated by an example. Suppose that a Perl script is called with a few arguments on the command line, like this:

```
perl_prog.pl file1 file2
```

If the perl script was written with a shebang line similar to the one we saw above *patch_stack* creates a stack to execute the perl binary as if the command line were:

```
/usr/local/bin/perl -wT perl_prog.pl file1 file2
```

If it is successful in this, the first part of this line, that is, the path to the binary executable of the interpreter, is returned. Then the body of the loop is executed once more, this time reading the file header and getting the segment sizes of the file to be executed. It is not permitted for the first line of a script to point to another script as its interpreter. That is why the variable *r* is used. It can only be incremented once, allowing only one chance to call *patch_stack*. If on the second time through the loop the code indicating a script is encountered, the test on line 18800 will break the loop. The code for a script, represented symbolically as *ESCRIPT*, is a negative number (defined on line 18741). In this case the test on line 18803 will cause *do_exit* to return with an error code telling whether the problem is a file that canot be executed or a command line that is too long.

Some work remains to be done to complete the exec operation. *Find_share* checks to see if the new process can share text with a process that is already running (line 18809), and *new_mem* allocates memory for the new image and releases the old memory. Both the image in memory and the process table need to be made ready before the exec-ed program can run. On lines 18819 to 18821 the executable file's i-node, filesystem, and modification time are saved in the process table. Then the stack is fixed up as in Fig. 4-38(c) and copied to the new image in memory. Next the text (if not already sharing text) and data segments are copied from the disk to the memory image by calling *rw_seg* (lines 18834 to 18841). If the *setuid* or *setgid* bits are set the file system needs to be notified to put the effective id information into the FS part of process table entry (lines 18845 to 18852). In the PM's part of the file table a pointer to the arguments to the new program is saved so the *ps* command will be able to show the command line, signal bitmasks are initialized, the FS is notified to close any file descriptors that should be closed after an exec, and the name of the command is saved for display by *ps* or during debugging (lines 18856 to 18877). Usually, the last step is to tell the kernel, but if tracing is enabled a signal must be sent (lines 18878 to 18881).

In describing the work of *do_exec* we mentioned a number of supporting functions provided in *exec.c*. *Read_header* (line 18889) not only reads the header and returns the segment sizes, it also verifies that the file is a valid MINIX 3 executable for the same CPU type as the operating system is compiled for. The constant value *A_I80386* on line 18944 is determined by a *#ifdef ... #endif* sequence at compile time. Binary executable programs for 32-bit MINIX 3 on Intel platforms must have this constant in their headers to be acceptable. If MINIX 3 were to be compiled to run in 16-bit mode the value here would be *A_I8086*. If you are curious, you can see values defined for other CPUs in *include/a.out.h*.

Procedure *new_mem* (line 18980) checks to see if sufficient memory is available for the new memory image. It searches for a hole big enough for just the data and stack if the text is being shared; otherwise it searches for a single hole big enough for the combined text, data, and stack. A possible improvement here would be to search for two separate holes. In earlier versions of MINIX it was required that the text and data/stack segments be contiguous, but this is not

necessary in MINIX 3. If sufficient memory is found, the old memory is released and the new memory is acquired. If insufficient memory is available, the exec call fails. After the new memory is allocated, *new_mem* updates the memory map (in *mp_seg*) and reports it to the kernel with the sys_newmap kernel call.

The final job of *new_mem* is to zero the bss segment, gap, and stack segment. (The bss segment is that part of the data segment that contains all the uninitialized global variables.) The work is done by the system task, called by sys_memset at line 19064. Many compilers generate explicit code to zero the bss segment, but doing it here allows MINIX 3 to work even with compilers that do not. The gap between data and stack segments is also zeroed, so that when the data segment is extended by brk, the newly acquired memory will contain zeroes. This is not only a convenience for the programmer, who can count on new variables having an initial value of zero, it is also a security feature on a multiuser operating system, where a process previously using this memory may have been using data that should not be seen by other processes.

The next procedure, *patch_ptr* (line 19074), relocates pointers like those of Fig. 4-38(b) to the form of Fig. 4-38(c). The work is simple: examine the stack to find all the pointers and add the base address to each one.

The next two functions work together. We described their purpose earlier. When a script is exec-ed the binary for the interpreter of the script is the executable that must be run. *Insert_arg* (line 19106) inserts strings into the PM copy of the stack. This is directed by *patch_stack* (line 19162), which finds all of the strings on the shebang line of the script, and calls *insert_arg*. The pointers have to be corrected, too, of course. *Insert_arg*'s job is straightforward, but there are a number of things that can go wrong and must be tested. This is a good place to mention that checking for problems when dealing with scripts is particularly important. Scripts, after all, can be written by users, and all computer professionals recognize that users are often the major cause of problems. But, seriously, a major difference between a script and a compiled binary is that you can generally trust the compiler to have refused to produce output for a wide range of errors in the source code. A script is not validated this way.

Fig. 4-46 shows how this would work for a call to a shell script, *s.sh*, which operates on a file *f1*. The command line looks like this:

```
s.sh f1
```

and the shebang line of the script indicates it is to be interpreted by the Bourne shell:

```
#! /bin/sh
```

In part (a) of the figure is the stack copied from the caller's space. Part (b) shows how this is transformed by *patch_stack* and *insert_arg*. Both of these diagrams correspond to Fig.4-38(b).

The next function defined in *exec.c* is *rw_seg* (line 19208). Is called once or twice per exec, possibly to load the text segment and always to load the data

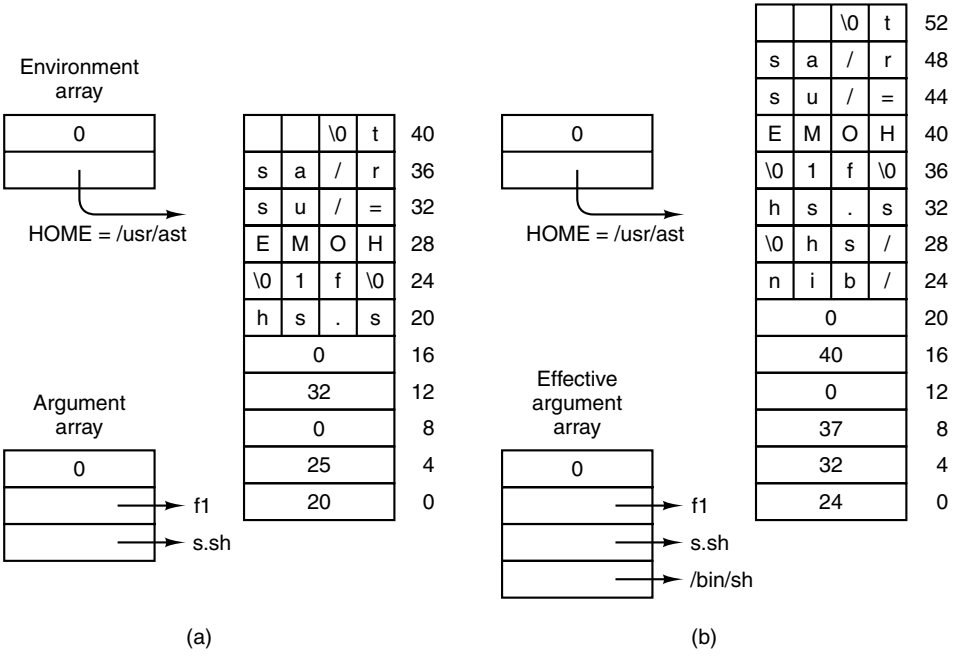

Figure 4-46. a. Arrays passed to *execve* and the stack created when a script is executed. b. After processing by *patch_stack*, the arrays and the stack look like this. The script name is passed to the program which interprets the script.

segment. Rather than just reading the file block by block and then copying the blocks to the user, a trick is used to allow the file system to load the entire segment directly to the user space. In effect, the call is decoded by the file system in a slightly special way so that it appears to be a read of the entire segment by the user process itself. Only a few lines at the beginning of the file system's read routine know that some monkey business is going on here. Loading is appreciably speeded up by this maneuver.

The final procedure in *exec.c* is *find_share* (line 19256). It searches for a process that can share text by comparing the i-node, device, and modification times of the file to be executed with those of existing processes. This is just a straightforward search of the appropriate fields in *mproc*. Of course, it must ignore the process on behalf of which the search is being made.

4.8.5 Implementation of BRK

As we have just seen, the basic memory model used by MINIX 3 is quite simple: each process is given a single contiguous allocation for its data and stack when it is created. It is never moved around in memory, it never grows, and it never shrinks. All that can happen is that the data segment can eat away at the gap from the low end, and the stack can eat away at it from the high end. Under

these circumstances, the implementation of the brk call in *break.c* is especially easy. It consists of verifying that the new sizes are feasible and then updating the tables to reflect them.

The top-level procedure is *do_brk* (line 19328), but most of the work is done in *adjust* (line 19361). The latter checks to see if the stack and data segments have collided. If they have, the brk call cannot be carried out, but the process is not killed immediately. A safety factor, *SAFETY_BYTES*, is added to the top of the data segment before making the test, so (hopefully) the decision that the stack has grown too far can be made while there is still enough room on the stack for the process to continue for a short while. It gets control back (with an error message), so it can print appropriate messages and shut down gracefully.

Note that *SAFETY_BYTES* and *SAFETY_CLICKS* are defined using #define statements in the middle of the procedure (line 19393). This use is rather unusual; normally such definitions appear at the beginning of files, or in separate header files. The associated comment reveals that the programmer found deciding upon the size of the safety factor to be difficult. No doubt this definition was done in this way to attract attention and, perhaps, to stimulate additional experimentation.

The base of the data segment is constant, so if *adjust* has to adjust the data segment, all it does is update the length field. The stack grows downward from a fixed end point, so if *adjust* also notices that the stack pointer, which is given to it as a parameter, has grown beyond the stack segment (to a lower address), both the origin and length are updated.

4.8.6 Implementation of Signal Handling

Eight POSIX system calls are related to signals. These calls are summarized in Fig. 4-47. These system calls, as well as the signals themselves, are processed in the file *signal.c*.

System call	Purpose
sigaction	Modify response to future signal
sigprocmask	Change set of blocked signals
kill	Send signal to another process
alarm	Send ALRM signal to self after delay
pause	Suspend self until future signal
sigsuspend	Change set of blocked signals, then PAUSE
sigpending	Examine set of pending (blocked) signals
sigreturn	Clean up after signal handler

Figure 4-47. System calls relating to signals.

The sigaction system call supports the *sigaction* and *signal* functions, which allow a process to alter how it will respond to signals. *Sigaction* is required by

POSIX and is the preferred call for most purposes, but the *signal* library function is required by Standard C, and programs that must be portable to non-POSIX systems should be written using it. The code for *do_sigaction* (line 19544) begins with checks for a valid signal number and verification that the call is not an attempt to change the response to a sigkill signal (lines 19550 and 19551). (It is not permitted to ignore, catch, or block sigkill. Sigkill is the ultimate means by which a user can control his processes and a system manager can control his users.) *Sigaction* is called with pointers to a *sigaction* structure, *sig_osa*, which receives the old signal attributes that were in effect before the call, and another such structure, *sig_nsa*, containing a new set of attributes.

The first step is to call the system task to copy the current attributes into the structure pointed to by *sig_osa*. *Sigaction* can be called with a *NULL* pointer in *sig_nsa* to examine the old signal attributes without changing them. In this case *do_sigaction* returns immediately (line 19560). If *sig_nsa* is not *NULL*, the structure defining the new signal action is copied to the PM's space.

The code in lines 19567 to 19585 modifies the *mp_catch*, *mp_ignore*, and *mp_sigpending* bitmaps according to whether the new action is to be to ignore the signal, to use the default handler, or to catch the signal. The *sa_handler* field of the *sigaction* structure is used to pass a pointer to the procedure to the function to be executed if a signal is to be caught, or one of the special codes *SIG_IGN* or *SIG_DFL*, whose meanings should be clear if you understand the POSIX standards for signal handling discussed earlier. A special MINIX 3-specific code, *SIG_MESS* is also possible; this will be explained below.

The library functions *sigaddset* and *sigdelset* are used, to modify the signal bitmaps, although the actions are straightforward bit manipulation operations that could have been implemented with simple macros. However, these functions are required by the POSIX standard in order to make programs that use them easily portable, even to systems in which the number of signals exceeds the number of bits available in an integer. Using the library functions helps to make MINIX 3 itself easily portable to different architectures.

We mentioned a special case above. The *SIG_MESS* code detected on line 19576 is available only for privileged (system) processes. Such processes are normally blocked, waiting for request messages. Thus the ordinary method of receiving a signal, in which the PM asks the kernel to put a signal frame on the recipients stack, will be delayed until a message wakes up the recipient. A *SIG_MESS* code tells the PM to deliver a notification message, which has higher priority than normal messages. A notification message contains the set of pending signals as an argument, allowing multiple signals to be passed in one message.

Finally, the other signal-related fields in the PM's part of the process table are filled in. For each potential signal there is a bitmap, the *sa_mask*, which defines which signals are to be blocked while a handler for that signal is executing. For each signal there is also a pointer, *sa_handler*. It can contain a pointer to the handler function, or special values to indicate the signal is to be ignored, handled

in the default way, or used to generate a message. The address of the library routine that invokes sigreturn when the handler terminates is stored in *mp_sigreturn*. This address is one of the fields in the message received by the PM.

POSIX allows a process to manipulate its own signal handling, even while within a signal handler. This can be used to change signal response to subsequent signals while a signal is being processed, and then to restore the normal set of responses. The next group of system calls support these signal-manipulation features. Sigpending is handled by *do_sigpending* (line 19597), which returns the *mp_sigpending* bitmap, so a process can determine if it has pending signals. Sigprocmask, handled by *do_sigprocmask*, returns the set of signals that are currently blocked, and can also be used to change the state of a single signal in the set, or to replace the entire set with a new one. The moment that a signal is unblocked is an appropriate time to check for pending signals, and this is done by calls to *check_pending* on line 19635 and line 19641. *Do_sigsuspend* (line 19657) carries out the sigsuspend system call. This call suspends a process until a signal is received. Like the other functions we have discussed here, it manipulates bitmaps. It also sets the sigsuspended bit in *mp_flags*, which is all it takes to prevent execution of the process. Again, this is a good time to make a call to *check_pending*. Finally, *do_sigreturn* handles sigreturn, which is used to return from a custom handler. It restores the signal context that existed when the handler was entered, and it also calls *check_pending* on line 19682.

When a user process, such as the *kill* command, invokes the kill system call, the PM's *do_kill* function (line 19689) is invoked. A single call to kill may require delivery of signals to a group of several processes, and *do_kill* just calls *check_sig*, which checks the entire process table for eligible recipients.

Some signals, such as sigint, originate in the kernel itself. *Ksig_pending* (line 19699) is activated when a message from the kernel about pending signals is sent to the PM. There may be more than one process with pending signals, so the loop on lines 19714 to 19722 repeatedly asks the system task for a pending signal, passes it on to *handle_sig*, and then tells the system task it is done, until there are no more processes with signals pending. The messages come with a bitmap, allowing the kernel to generate multiple signals with one message. The next function, *handle_sig*, processes the bitmap one bit at a time on lines 19750 to 19763. Some kernel signals need special attention: the process ID is changed in some cases to cause the signal to be delivered to a group of processes (lines 19753 to 19757). Otherwise, each set bit results in a call to *check_sig*, just as in *do_kill*.

Alarms and Timers

The alarm system call is handled by *do_alarm* (line 19769). It calls the next function, *set_alarm*, which is a separate function because it is also used to turn off a timer when a process exits with a timer still on. This is done by calling *set_alarm* with an alarm time of zero. *Set_alarm* does its work with timers

maintained within the process manager. It first determines if a timer is already set on behalf of the requesting process, and if so, whether it has expired, so the system call can return the time in seconds remaining on a previous alarm, or zero if no timer was set. A comment within the code explains some problems with dealing with long times. Some rather ugly code on line 19918 multiplies the argument to the call, a time in seconds, by the constant *HZ*, the number of clock ticks per second, to get a time in tick units. Three casts are needed to make the result the correct *clock_t* data type. Then on the next line the calculation is reversed with *ticks* cast from *clock_t* to *unsigned long*. The result is compared with a cast of the original alarm time argument cast to *unsigned long*. If they are not equal it means the requested time resulted in a number that was out of range of one of the data types used, and a value which means "never" is substituted. Finally, either *pm_set_timer* or *pm_cancel_timer* is called to add or remove a timer from the process manager's timer queue. The key argument to the former call is *cause_sigalarm*, the watchdog function to be executed when the timer expires.

Any interaction with the timer maintained in kernel space is hidden in the calls to the *pm_XXX_timer* routines. Every request for an alarm that eventually culminates in an alarm will normally result in a request to set a timer in kernel space. The only exception would be if more than one request for a timeout at the exact same time were to occur. However, processes may cancel their alarms or terminate before their alarms expire. A kernel call to request setting a timer in kernel space only needs to be made when there is a change to the timer at the head of the process manager's timer queue.

Upon expiration of a timer in the kernel-space timer queue that was set on behalf of the PM, the system task announces the fact by sending the PM a notification message, detected as type *SYN_ALARM* by the main loop of the PM. This results in a call to *pm_expire_timers*, which ultimately results in execution of the next function, *cause_sigalrm.*

Cause_sigalarm (line 19935) is the watchdog, mentioned above. It gets the process number of the process to be signaled, checks some flags, resets the *ALARM_ON* flag, and calls *check_sig* to send the *SIGALRM* signal.

The default action of the *SIGALRM* signal is to kill the process if it is not caught. If the *SIGALRM* is to be caught, a handler must be installed by sigaction. Fig. 4-48 shows the complete sequence of events for a *SIGALRM* signal with a custom handler. The figure shows that three sequences of messages occur. First, in message (1) the user does an alarm call via a message to the PM. At this point the process manager sets up a timer in the queue of timers it maintains for user processes, and acknowledges with message (2). Nothing more may happen for a while. When the timer for this request reaches the head of the PM's timer queue, because timers ahead of it have expired or have been cancelled, message (3) is sent to the system task to have it set up a new kernel-space timer for the process manager, and is acknowledged by message (4). Again, some time will pass before anything more happens. But after this timer reaches the head of the kernel-

space timer queue the clock interrupt handler will find it has expired. The remaining messages in the sequence will follow quickly. The clock interupt handler sends a *HARD_INT* message (5) to the clock task, which causes it to run and update its timers. The timer watchdog function, *cause_alarm*, initiates message (6), a notification to the PM. The PM now updates its timers, and after determining from its part of the process table that a handler is installed for *SIGALRM* in the target process, sends message (7) to the system task to have it do the stack manipulations needed to send the signal to the user process. This is acknowledged by message (8). The user process will be scheduled and will execute the handler, and then will make a sigreturn call (9) to the process manager. The process manager then sends message (10) to the system task to complete the cleanup, and this is acknowledged by message (11). Not shown in this diagram is another pair of messages from the PM to the system task to get the uptime, made before message (3).

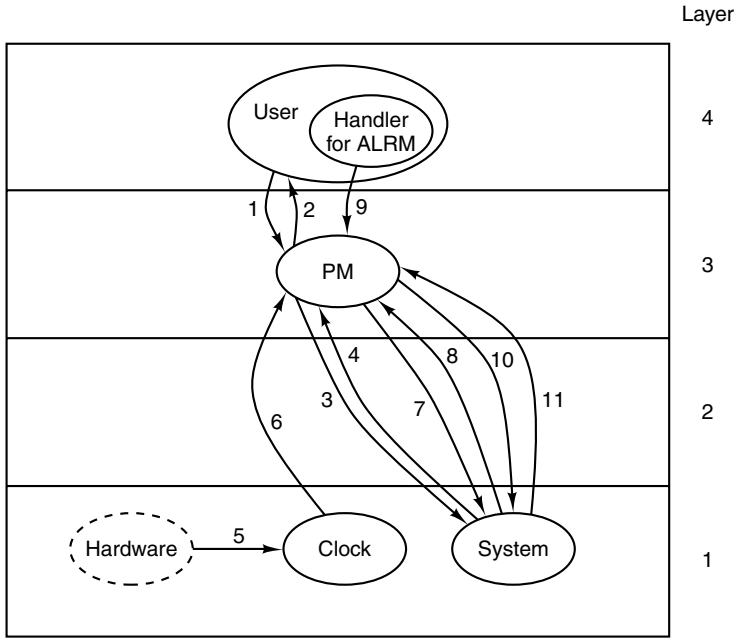

Figure 4-48. Messages for an alarm. The most important are: (1) User does **alarm**. (3) PM asks system task to set timer. (6) Clock tells PM time has expired. (7) PM requests signal to user. (9) Handler terminates with call to sigreturn. See text for details.

The next function, *do_pause*, takes care of the pause system call (line 19853). It isn't really related to alarms and timers, although it can be used in a program to suspend execution until an alarm (or some other signal) is received.

All that is necessary is to set a bit and return the *SUSPEND* code, which causes the main loop of the PM to refrain from replying, thus keeping the caller blocked. The kernel need not even be informed, since it knows that the caller is blocked.

Support Functions for Signals

Several support functions in *signal.c* have been mentioned in passing. We will now look at them in more detail. By far the most important is *sig_proc* (line 19864), which actually sends a signal. First a number of tests are made. Attempts to send to dead or zombie processes are serious problems that cause a system panic (lines 19889 to 19893). A process that is currently being traced is stopped when signaled (lines 19894 to 19899). If the signal is to be ignored, *sig_proc*'s work is complete on line 19902. This is the default action for some signals, for instance, those signals that are required to be there by POSIX but do not have to (and are not) supported by MINIX 3. If the signal is blocked, the only action that needs to be taken is to set a bit in that process' *mp_sigpending* bitmap. The key test (line 19910) is to distinguish processes that have been enabled to catch signals from those that have not. With the exception of signals that are converted into messages to be sent to system services all other special considerations have been eliminated by this point and a process that cannot catch the signal must be terminated.

First we will look at the processing of signals that are eligible to be caught (lines 19911 to 19950). A message is constructed to be sent to the kernel, some parts of which are copies of information in the PM's part of the process table. If the process to be signaled was previously suspended by sigsuspend, the signal mask that was saved at the time of suspension is included in the message; otherwise the current signal mask is included (line 19914). Other items included in the message are several addresses in the space of the signaled process space: the signal handler, the address of the *sigreturn* library routine to be called on completion of the handler, and the current stack pointer.

Next, space is allocated on the process' stack. Figure 4-49 shows the structure that is put on the stack. The *sigcontext* portion is put on the stack to preserve it for later restoration, since the corresponding structure in the process table itself is altered in preparation for execution of the signal handler. The *sigframe* part provides a return address for the signal handler and data needed by sigreturn to complete restoration of the process' state when the handler is done. The return address and frame pointer are not actually used by any part of MINIX 3. They are there to fool a debugger if anyone should ever try to trace execution of a signal handler.

The structure to be put on the signaled process' stack is fairly large. The code in lines 19923 and 19924 reserves space for it, following which a call to *adjust* tests to see whether there is enough room on the process' stack. If there is not

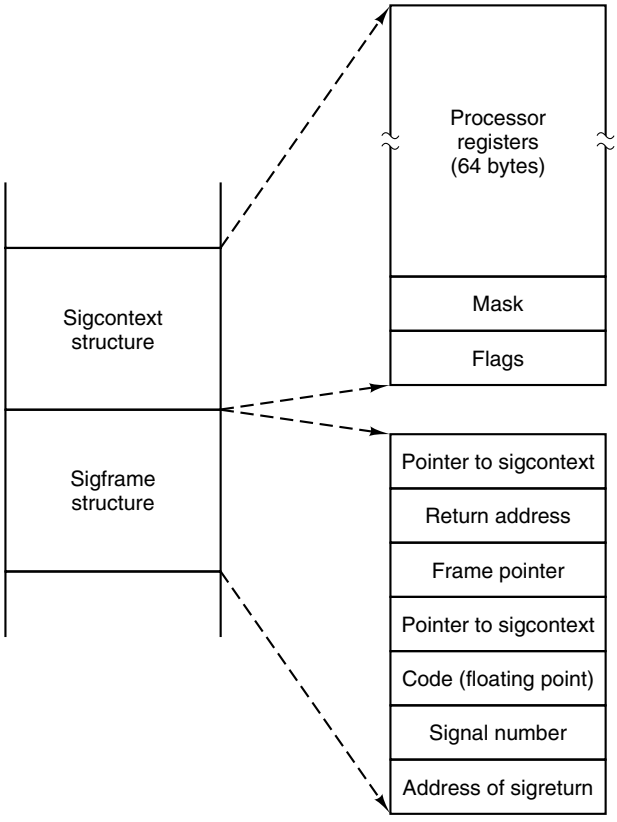

Figure 4-49. The sigcontext and sigframe structures pushed on the stack to prepare for a signal handler. The processor registers are a copy of the stack-frame used during a context switch.

enough stack space, the process is killed by jumping to the label *doterminate* using the seldom-used C goto (lines 19926 and 19927).

The call to *adjust* has a potential problem. Recall from our discussion of the implementation of brk that *adjust* returns an error if the stack is within *SAFETY_BYTES* of running into the data segment. The extra margin of error is provided because the validity of the stack can only be checked occasionally by software. This margin of error is probably excessive in the present instance, since it is known exactly how much space is needed on the stack for the signal, and additional space is needed only for the signal handler, presumably a relatively simple function. It is possible that some processes may be terminated unnecessarily because the call to *adjust* fails. This is certainly better than having programs fail mysteriously at other times, but finer tuning of these tests may be possible at some time in the future.

If there is enough room on the stack for the struct, two more flags are checked. The *SA_NODEFER* flag indicates if the signaled process is to block further signals of the same type while handling a signal. The *SA_RESETHAND* flag tells if the signal handler is to be reset upon receiving this signal. (This provides faithful emulation of the old *signal* call. Although this "feature" is often considered a fault in the old call, support of old features requires supporting their faults as well.) The kernel is then notified, using the sys_sigsend kernel call (line 19940) to put the sigframe on the stack. Finally, the bit indicating that a signal is pending is cleared, and *unpause* is called to terminate any system call on which the process may be hanging. When the signaled process next executes, the signal handler will run. If for some reason all of the tests above failed, the PM panics (line 19949).

The exception mentioned above—signals converted into messages for system services—is tested for on line 19951, and carried out by the sys_kill kernel call that follows. This causes the system task to send a notification message to the signaled process. Recall that, unlike most other notifications, a notification from the system task carries a payload in addition to the basic information about its origin and a timestamp. It also transmits a bitmap of signals, so the signaled system process learns of all pending signals. If the sys_kill call fails, the PM panics. If it succeeds *sig_proc* returns (line 19954). If the test on line 19951 failed, execution falls through to the *doterminate* label.

Now let us look at the termination code marked by the label *doterminate* (line 19957). The label and a goto are the easiest way to handle the possible failure of the call to *adjust*. Here signals are processed that for one reason or another cannot or should not be caught. It is possible that the signal was one to be ignored, in which case *sig_proc* just returns. Otherwise the process must be terminated. The only question is whether a core dump is also needed. Finally, the process is terminated as if it had exited, through a call to *pm_exit* (line 19967).

Check_sig (line 19973) is where the PM checks to see if a signal can be sent. The call

```
kill(0, sig);
```

causes the indicated signal to be sent to all the processes in the caller's group (i.e., all the processes started from the same terminal). Signals originating in the kernel and the reboot system call also may affect multiple processes. For this reason, *check_sig* loops on lines 19996 to 20026 to scan through the process table to find all the processes to which a signal should be sent. The loop contains a large number of tests. Only if all of them are passed is the signal sent, by calling *sig_proc* on line 20023.

Check_pending (line 20036) is another important function called several times in the code we have just reviewed. It loops through all the bits in the *mp_sigpending* bitmap for the process referred to by *do_sigmask*, *do_sigreturn*, or *do_sigsuspend*, to see if any blocked signal has become unblocked. It calls

sig_proc to send the first unblocked pending signal it finds. Since all signal handlers eventually cause execution of *do_sigreturn*, this code suffices eventually to deliver all pending unmasked signals.

The procedure *unpause* (line 20065) has to do with signals that are sent to processes suspended on pause, wait, read, write, or sigsuspend calls. Pause, wait, and sigsuspend can be checked by consulting the PM's part of the process table, but if none of these are found, the file system must be asked to use its own *do_unpause* function to check for a possible hangup on read or write. In every case the action is the same: an error reply is sent to the waiting call and the flag bit that corresponds to the cause of the wait is reset so the process may resume execution and process the signal.

The final procedure in this file is *dump_core* (line 20093), which writes core dumps to the disk. A core dump consists of a header with information about the size of the segments occupied by a process, a copy of all the process' state information, obtained by copying the kernel process table information for the process, and the memory image of each of the segments. A debugger can interpret this information to help the programmer determine what went wrong during execution of the process.

The code to write the file is straightforward. The potential problem mentioned in the previous section again raises its head, but in a somewhat different form. To be sure the stack segment to be recorded in the core dump is up to date, *adjust* is called on line 20120. This call may fail because of the safety margin built into it. The success of the call is not checked by *dump_core*, so the core dump will be written in any case, but within the file the information about the stack may be incorrect.

Support Functions for Timers

The MINIX 3 process manager handles requests for alarms from user processes, which are not allowed to contact the kernel or the system task directly themselves. All details of scheduling an alarm at the clock task are hidden behind this interface. Only system processes are allowed to set an alarm timer at the kernel. Support for this is provided in the file *timers.c* (line 20200).

The process manager maintains a list of requests for alarms, and asks the system task to notify it when it is time for an alarm. When an alarm comes from the kernel the process manager passes it on to the process that should receive it.

Three functions are provided here to support timers. *Pm_set_timer* sets a timer and adds it to the PM's list of timers, *pm_expire_timer* checks for expired timers and *pm_cancel_timer* removes a timer from the PM's list. All three of these take advantage of functions in the timers library, declared in *include/timers.h*. The function *Pm_set_timer* calls *tmrs_settimer*, *pm_expire_timer* calls *tmrs_exptimers*, and *pm_cancel_timer* calls *tmrs_clrtimers*. These all manage

the business of traversing a linked list and inserting or removing an item, as required. Only when an item is inserted at or removed from the head of the queue does it become necessary to involve the system task in order to adjust the kernel-space timer queue. In such cases each of the *pm_XXX_timer* functions uses a sys_setalarm kernel call to request help at the kernel level.

4.8.7 Implementation of Other System Calls

The process manager handles three system calls that involve time in *time.c*: time, stime, and times. They are summarized in Fig. 4-50.

Call	Function
time	Get current real time and uptime in seconds
stime	Set the real time clock
times	Get the process accounting times

Figure 4-50. Three system calls involving time.

The real time is maintained by the clock task within the kernel, but the clock task itself does not exchange messages with any process except the system task. As a consequence, the only way to get or set the real time is to send a message to the system task. This is, in fact, what *do_time* (line 20320) and *do_stime* (line 20341) both do. The real time is measured in seconds since Jan 1, 1970.

Accounting information is also maintained by the kernel for each process. At each clock tick it charges one tick to some process. The kernel doesn't know about parent-child relationships, so it falls to the process manager to accumulate time information for the children of a process. When a child exits, its times are accumulated in the parent's slot in the PM's part of the process table. *Do_times* (line 20366) retrieves the time usage of a parent process from the system task with a sys_times kernel call, then fills in a reply message with user and system time charged to children.

The file *getset.c* contains one procedure, *do_getset* (line 20415), which carries out seven POSIX-required PM system calls. They are shown in Fig. 4-51. They are all so simple that they are not worth an entire procedure each. The getuid and getgid calls both return the real and effective UID or GID.

Setting the uid or gid is slightly more complex than just reading it. A check has to be made to see if the caller is authorized to set the uid or gid. If the caller passes the test, the file system must be informed of the new uid or gid, since file protection depends on it. The setsid call creates a new session, and a process which is already a process group leader is not allowed to do this. The test on line 20463 checks this. The file system completes the job of making a process into a session leader with no controlling terminal.

In contrast to the system calls considered so far in this chapter, the calls in *misc.c* are not required by POSIX. These calls are necessary because the user-

System Call	Description
getuid	Return real and effective UID
getgid	Return real and effective GID
getpid	Return PIDs of process and its parent
setuid	Set caller's real and effective UID
setgid	Set caller's real and effective GID
setsid	Create new session, return PID
getpgrp	Return ID of process group

Figure 4-51. The system calls supported in *servers/pm/getset.c*.

space device drivers and servers of MINIX 3 need support for communication with the kernel that is not necessary in monolithic operating systems. Fig. 4-52 shows these calls and their purposes.

System Call	Description
do_allocmem	Allocate a chunk of memory
do_freemem	Deallocate a chunk of memory
do_getsysinfo	Get info about PM from kernel
do_getprocnr	Get index to proc table from PID or name
do_reboot	Kill all processes, tell FS and kernel
do_getsetpriority	Get or set system priority
do_svrctrl	Make a process into a server

Figure 4-52. Special-purpose MINIX 3 system calls in *servers/pm/misc.c*.

The first two are handled entirely by the PM. *do_allocmem* reads the request from a received message, converts it into click units, and calls *alloc_mem*. This is used, for example, by the memory driver to allocate memory for the RAM disk. *Do_freemem* is similar, but calls *free_mem*.

The next calls usually need help from other parts of the system. They may be thought of as interfaces to the system task. *Do_getsysinfo* (line 20554) can do several things, depending on the request in the message received. It can call the system task to get information about the kernel contained in the *kinfo* structure (defined in the file *include/minix/type.h*). It can also be used to provide the address of the PM's own part of the process table or a copy of the entire process table to another process upon request. The final action is carried out by a call to *sys_datacopy* (line 20582). *Do_getprocnr* can find an index into the process table in its own section if given PID, and calls the system task for help if all it has to work with is the name of the target process.

The next two calls, although not required by POSIX, will probably be found in some form in most UNIX-like systems. *Do_reboot* sends a *KILL* signal to all processes, and tells the file system to get ready for a reboot. Only after the file system has been synched is the kernel notified with a *sys_abort* call (line 20667). A reboot may be the result of a panic, or a request from the superuser to halt or restart, and the kernel needs to know which case applies. *Do_getsetpriority*, supports the famous UNIX *nice* utility, which allows a user to reduce the priority of a process in order to be a good neighbor to other processes (possibly his own). More importantly, this call is used by the MINIX 3 system to provide fine-grained control of relative priorities of system components. A network or disk device that must handle a rapid stream of data can be given priority over one that receives data more slowly, such as a keyboard. Also, a high-priority process that is stuck in a loop and preventing other processes from running may have its priority lowered temporarily. Changing priority is done by scheduling the process on a lower (or higher) priority queue, as described in the discussion of implementation of scheduling in Chap. 2. When this is initiated by the scheduler in the kernel there is no need to involve the PM, of course, but an ordinary process must use a system call. At the level of the PM it is just a matter of reading the current value returned in a message or generating a message with a new value. A kernel call, *sys_nice* sends the new value to the system task.

The last function in *misc.c* is *do_svrctl*. It is currently used to enable and disable swapping. Other functions once served by this call are expected to be implemented in the reincarnation server.

The last system call we will consider in this chapter is ptrace, handled by *trace.c*. This file is not listed in Appendix B, but may be found on the CD-ROM and the MINIX 3 Web site. Ptrace is used by debugging programs. The parameter to this call can be one of eleven commands. These are shown in Fig. 4-53. In the PM *do_trace* processes four of them: *T_OK*, *T_RESUME*, I T_EXIT , *T_STEP*. Requests to enable and exit tracing are completed here. All other commands are passed on to the system task, which has access to the kernel's part of the process table. This is done by calling the *sys_trace* library function. Two support functions for tracing are provided. *Find_proc* searches the process table for the process to be traced, and *stop_proc* stops a traced process when it is signaled.

4.8.8 Memory Management Utilities

We will end this chapter by describing briefly two more files which provide support functions for the process manager. These are *alloc.c* and *utility.c*. Because internal details of these files are not discussed here, they are not printed in Appendix B (to keep this already fat book from becoming even fatter). However, they are available on the CD-ROM and the MINIX 3 Web site.

Alloc.c is where the system keeps track of which parts of memory are in use and which are free. It has three entry points: