PROCESS CONTROL:

This will describe the use and implementation of the system calls that control the process context. The fork system call creates a new process, the exit call terminates process execution, and the wait call allows a parent process to synchronize its execution with the exit of a child process.

Signals inform processes of asynchronous events. Because the kernel synchronizes execution of exit and wait via signals, the chapter presents signals before exit and wait. The exec system call allows a process to invoke a "new" program, overlaying its address space with the executable image of a file.

The brk system call allows a process to allocate more memory dynamically; similarly, the system allows the user stack to grow dynamically by allocating more space when necessary, using the same mechanisms as for brk.

Figure 6.1 shows the relationship between the system calls and the memory management algorithms. Almost all calls use sleep and wakeup, not shown in the figure. Furthermore, exec interacts with the file system algorithms.

System Calls Dealing with Memory Management			System Calls Dealing with Synchronization				Miscellaneous	
fork	exec	brk	exit	wait	signal	kill	setpgrp	setuid
dupreg attachreg	detachreg allocreg attachreg growreg loadreg mapreg	growreg	detachreg					

Figure 6.1: Process System Calls and Relation to Other Algorithms

PROCESS CREATION:

The only way for a user to create a new process in the UNIX operating system is to invoke the fork system call. The process that invokes fork is called the parent process, and the newly created process is called the child process. The syntax for the fork system call is

pid = fork();

On return from the fork system call, the two processes have identical copies of their user-level context except for the return value pid. In the parent process, pid is the child process ID; in the child process, pid is 0. Process 0, created internally by the kernel when the system is booted, is the only process not created via fork. The kernel does the following sequence of operations for fork.

1. It allocates a slot in the process table for the new process.
2. It assigns a unique ID number to the child process.
3. It makes a logical copy of the context of the parent process. Since certain portions of a process, such as the text region, may be shared between processes, the kernel can sometimes increment a region reference count instead of copying the region to a new physical location in memory.
4. It increments file and inode table counters for files associated with the process.
5. It returns the ID number of the child to the parent process, and a 0 value to the child process.

The implementation of the fork system call is not trivial, because the child process appears to start its execution sequence out of thin air. The algorithm for fork (figure 6.2) varies slightly for demand paging and swapping systems; the ensuing discussion is based on traditional swapping systems but will point out the places that change for demand paging systems.

It also assumes that the system has enough main memory available to store the child process.

Figure 6.2: Algorithm fork

Input: none

Output: to parent process, child PID number to child process. 0

{

check for available kernel resources;

get free proc table slot, unique PID number;

check that user not running too many processes;

mark child state "being created;"

copy data from parent proc table slot to new child slot;

increment counts on current directory inode and changed root (if applicable);

```
        increment open file counts in file table;
        make copy of parent context (u area, text, data, stack) in memory;
        push dummy system level context layer onto child system level context;
                dummy context contains data allowing child process
                to recognize itself and start running from here when scheduled;
        if (executing process is parent process)
        {
                change child state to "ready to run;"
                return(child ID); /* from system to user */
        }
        else /* executing process is the child process */
        {
                initialize u area timing fields;
                return(0): /* to user */
        }
}
```


The kernel first ascertains that it has available resources to complete the fork successfully. On a swapping system, it needs space either in memory or on disk to hold the child process; on a paging system, it has to allocate memory for auxiliary tables such as page tables. If the resources are unavailable, the fork call fails.

The kernel finds a slot in the process table to start constructing the context of the child process and makes sure that the user doing the fork does not have too many processes already running. It also picks a unique ID number for the new process, one greater than the most recently assigned ID number. If another process already has the proposed ID number, the kernel attempts to assign the next higher ID number.

When the ID numbers reach a maximum value, assignment starts from 0 again. Since most processes execute for a short time, most ID numbers are not in use when ID assignment wraps around.

The system imposes a (configurable) limit on the number of processes a user can simultaneously execute so that no user can steal many process table slots, thereby preventing other users from creating new processes.

Similarly, ordinary users cannot create a process that would occupy the last remaining slot in the process table, or else the system could effectively deadlock. That is, the kernel cannot guarantee that existing processes will exit naturally and, therefore, no new processes could be created, because all the process table slots are in use.

SIGNALS

Signals inform processes of the occurrence of asynchronous events. Processes may send each other signal with the kill system call, or the kernel may send signals internally. There are 19 signals in the System V (Release 2) UNIX system that can be classified as follows:

- Signals having to do with the termination of a process, sent when a process exits or when a process invokes the signal system call with the death of child parameter;
- Signals having to do with process induced exceptions such as when a process accesses an address outside its virtual address space, when it attempts to write memory that is read-only (such as program text), or when it executes a privileged instruction or for various hardware errors;
- Signals having to do with the unrecoverable conditions during a system call, such as running out of system resources during exec after the original address space have been released;
- Signals caused by an unexpected error condition during a system call, such as making a nonexistent system call (the process passed a system call number that does not correspond to a legal system call), writing a pipe that has no reader processes, or using an illegal "reference" value for the /seek system call.
- Signals originating from a process in user mode, such as when a process wishes to receive an alarm signal after a period of time, or when processes send arbitrary signals to each other with the kill system call;
- Signals related to terminal interaction such as when a user hangs up a terminal (or the “carrier" signal drops on such a line for any reason), or when a user presses the “break" or “delete" keys on a terminal keyboard;
- Signals for tracing execution of a process

The treatment of signals has several facets, namely how the kernel sends a signal to a process, how the process handles a signal, and how a process controls its reaction to signals. To send a signal to a process, the kernel sets a bit in the signal field of the process table entry, corresponding to the type of signal received.

If the process is asleep at an interruptible priority, the kernel awakens it. The job of the sender (process or kernel) is complete. A process can remember different types of signals, but it has no memory of how many signals it receives of a particular type.

The kernel checks for receipt of a signal when a process is about to return from kernel mode to user mode and when it enters or leaves the sleep state at a suitably low scheduling priority (see Figure 6.3). The kernel handles signals only when a process returns from kernel mode to user mode. Thus, a signal does not have an instant effect on a process running in kernel mode. If a process is running in user mode, and the kernel handles an interrupt that causes a signal to be sent to the process, the kernel will recognize and handle the signal when it returns from the interrupt. Thus, a process never executes in user mode before handling outstanding signals.

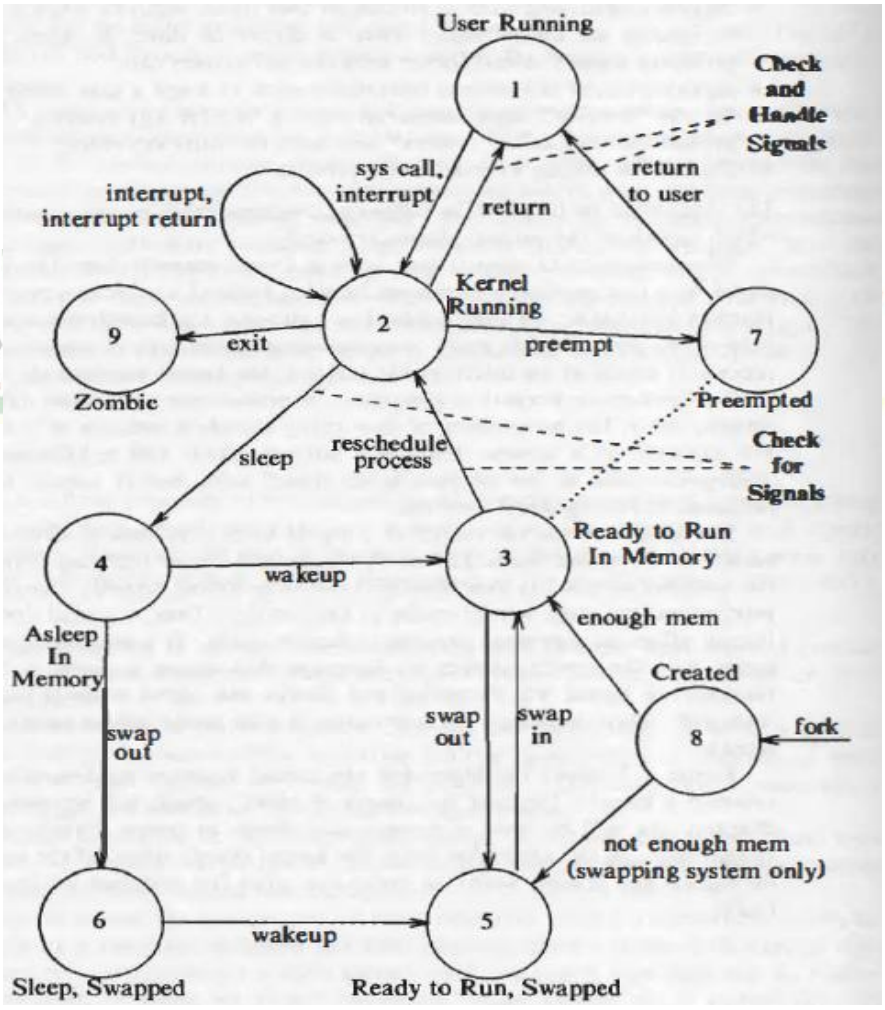

Figure 6.3: Checking and Handling Signals in the Process State Diagram

Figure 6.4 shows the algorithm the kernel executes to determine if a process received a signal.

```
algorithm issig                    /* test for receipt of signals */
input:  none
output: true, if process received signals that it does not ignore
        false otherwise
{
    while (received signal field in process table entry not 0)
    {
        find a signal number sent to the process;
        if (signal is death of child)
        {
            if (ignoring death of child signals)
                free process table entries of zombie children;
            else if (catching death of child signals)
                return(true);
        }
        else if (not ignoring signal)
            return(true);
        turn off signal bit in received signal field in process table;
    }
    return(false);
}
```

Figure 6.4: Algorithm for Recognizing Signals

Handle Signals:

The kernel handles signals in the context of the process that receives them so a process must run to handle signals. There are three cases for handling signals: the process exits on receipt of the signal, it ignores the signal, or it executes a particular (user) function on receipt of the signal. The default action is to call exit in kernel mode, but a process can specify special action to take on receipt of certain signals with the signal system call.

The syntax for the signal system call is

oldfunction = signal (signum, function);

Where signum is the signal number the process is specifying the action for, function is the address of the (user) function the process wants to invoke on receipt of the signal, and the return value oldfunction was the value of function in the most recently specified call to signal for signum.

The process can pass the values 1 or 0 instead of a function address: The process will ignore future occurrences of the signal if the parameter value is 1 and exit in the kernel on receipt of the signal if its value is 0 (the default value).

The u area contains an array of signal-handler fields, one for each signal defined in the system. The kernel stores the address of the user function in the field that corresponds to the signal number.

When handling a signal (Figure 6.5) the kernel determines the signal type and turns off the appropriate signal bit in the process table entry, set when the process received the signal. If the signal handling function is set to its default value, the kernel will dump a "core" image of the process for certain types of signals before exiting.

The dump is a convenience to programmers, allowing them to ascertain its causes and, thereby, to debug their programs. The kernel dumps core for signals that imply something is wrong with a process, such as when a process executes an illegal instruction or when it accesses an address outside its virtual address space. But the kernel does not dump core for signals that do not imply a program error.

```
algorithm psig    /* handle signals after recognizing their existence */
input:  none
output: none
{
        get signal number set in process table entry;
        clear signal number in process table entry;
        if (user had called signal sys call to ignore this signal)
                return;                 /* done */
        if (user specified function to handle the signal)
        {
                get user virtual address of signal catcher stored in u area;
                /* the next statement has undesirable side-effects */
                clear u area entry that stored address of signal catcher;
                modify user level context:
                        artificially create user stack frame to mimic
                        call to signal catcher function;
                modify system level context:
                        write address of signal catcher into program
                        counter field of user saved register context;
                return;
        }
        if (signal is type that system should dump core image of process)
        {
                create file named "core" in current directory;
                write contents of user level context to file "core";
        }
        invoke exit algorithm immediately;
}
```

Figure 6.5: Algorithm for handling signals

When a process receives a signal that it had previously decided to ignore, it continues as if the signal had never occurred. Because the kernel does not reset the field in the u area that shows the signal is ignored, the process will ignore the signal if it happens again, too. If a process receives a signal that it had previously decided to catch, it executes the user specified signal handling function immediately when it returns to user mode, after the kernel does the following steps:

1. The kernel accesses the user saved register context, finding the program counter and stack pointer that it had saved for return to the user process.
2. It clears the signal handler field in the u area, setting it to the default state.
3. The kernel creates a new stack frame on the user stack, writing in the values of the program counter and stack pointer it had retrieved from the user saved register context and allocating new space, if necessary.
    a. The user stack looks as if the process had called a user-level function (the signal catcher) at the point where it had made the system call or where the kernel had interrupted it (before recognition of the signal).
4. The kernel changes the user saved register context: It resets the value for the program counter to the address of the signal catcher function and sets the value for the stack pointer to account for the growth of the user stack.

Sending Signals from Processes:

Processes use the kill system call to send signals. The syntax for the system call is

kill (pid, signum)

Where pid identifies the set of processes to receive the signal, and signum is the signal number being sent. The following list shows the correspondence between values of pid and sets of processes.

- If pid is a positive integer, the kernel sends the signal to the process with process ID pid.
- If pid is 0, the kernel sends the signal to all processes in the sender's process group.
- If pid is -1, the kernel sends the signal to all processes whose real user ID equals the effective user ID of the sender. If the sending process has effective user ID of super user, the kernel sends the signal to all processes except processes 0 and 1.
- If pid is a negative integer but not - 1 , the kernel sends the signal to all processes in the process group equal to the absolute value of pid.

In all cases, if the sending process does not have effective user ID of super user, or its real or effective user ID do not match the real or effective user ID of the receiving process, kill fails.

PROCESS TERMINATION:

Processes on a UNIX system terminate by executing the exit system call. An exiting process enters the zombie state, relinquishes its resources, and dismantles its context except for its slot in the process table. The syntax for the call is

exit (status);

Where the value of status is returned to the parent process for its examination; Processes may call exit explicitly or implicitly at the end of a program: the startup routine linked with all C programs calls exit when the program returns from the main function, the entry point of all programs. Alternatively, the kernel may invoke exit internally for a process on receipt of uncaught signals as discussed above. If so, the value of status is the signal number.

The system imposes no time limit on the execution of a process, and processes frequently exist for a long time.


```
algorithm exit
input:  return code for parent process
output: none
{
        ignore all signals;
        if (process group leader with associated control terminal)
        {
                send hangup signal to all members of process group;
                reset process group for all members to 0;
        }
        close all open files (internal version of algorithm close);
        release current directory (algorithm iput);
        release current (changed) root, if exists (algorithm iput);
        free regions, memory associated with process (algorithm freereg);
        write accounting record;
        make process state zombie
        assign parent process ID of all child processes to be init process (1);
                if any children were zombie, send death of child signal to init;
        send death of child signal to parent process;
        context switch;
}
```

Figure 6.6: Algorithm for exit

Figure 6.6 shows the algorithm for exit. The kernel first disables signal handling for the process, because it no longer makes any sense to handle signals. If the exiting process is a process group leader associated with a control terminal, the kernel assumes the user is not doing any useful work and sends a "hangup" signal to all processes in the process group. Thus, if a user types "end of file" (control-d character) in the login shell while some processes associated with the terminal are still alive, the exiting process will send them a hangup signal.

The kernel also resets the process group number to 0 for processes in the process group, because it is possible that another process will later get the process ID of the process that just exited and that it too will be a process group leader. Processes that belonged to the old process group will not belong to the later process group.

The kernel then goes through the open file descriptors, closing each one internally with algorithm close, and releases the inodes it bad accessed for the current directory and changed root (if it exists) via algorithm iput.

AWAITING PROCESS TERMINATION:

A process can synchronize its execution with the termination of a child process by executing the wait system call. The syntax for the system call is

pid = wait(stat_addr);

Where pid is the process ID of the zombie child, and stat_addr is the address in user space of an integer that will contain the exit status code of the child.

Figure 6.7 shows the algorithm for wait. The kernel searches for a zombie child of the process and, if there are no children, returns an error. If it finds a zombie child, it extracts the PID number and the parameter supplied to the child's exit call and returns those values from the system call.

An exiting process can thus specify various return codes to give the reason it exited, but many programs do not consistently set it in practice. The kernel adds the accumulated time the child process executed in user and in kernel mode to the appropriate fields in the parent process u area and, finally, releases the process table slot formerly occupied by the zombie process. The slot is now available for a new process.

If the process executing wait has child processes but none are zombie, it sleeps at an interruptible priority until the arrival of a signal. The kernel does not contain an explicit wake up call for a process sleeping in wait: such processes only wake up on receipt of signals.

```
algorithm wait
input:   address of variable to store status of exiting process
output: child ID, child exit code
{
      if (waiting process has no child processes)
            return(error);

      for (;;)       /* loop until return from inside loop */
      {
            if (waiting process has zombie child)
            {
                  pick arbitrary zombie child;
                  add child CPU usage to parent;
                  free child process table entry;
                  return(child ID, child exit code);
            }
            if (process has no children)
                  return error;
            sleep at interruptible priority (event child process exits);
      }
}
```

Figure 6.7: Algorithm for wait

INVOKING OTHER PROGRAMS:

The exec system call invokes another program, overlaying the memory space of a process with a copy of an executable file. The contents of the user-level context that existed before the exec call are no longer accessible afterward except for exec's parameters, which the kernel copies from the old address space to the new address space. The syntax for the system call is

execve(filename, argv, envp)

Where filename is the name of the executable file being invoked, argv is a pointer to an array of character pointers that are parameters to the executable program. And envp is a pointer to an array of character pointers that are the environment of the executed program. There are several library functions that call the exec system call such as execl, execv, execle, and so on. All call execve eventually; hence it is used here to specify the exec system call. When a program uses command line parameters, as in

main (argc, argv)

The array argv is a copy of the argv parameter to exec. The character strings in the environment are of the form "name-value" and may contain useful information for programs, such as the user's home directory and a path of directories to search for executable programs.

Processes can access their environment via the global variable environ, initialized by the C startup routine.

Figure 6.8 shows the algorithm for the exec system call. Exec first accesses the file via algorithm namei to determine if it is an executable, regular (nondirectory) file and to determine if the user has permission to execute the program. The kernel then reads the file header to determine the layout of the executable file.

```
algorithm exec
input:  (1) file name
        (2) parameter list
        (3) environment variables list
output: none
{
      get file inode (algorithm namei);
      verify file executable, user has permission to execute;
      read file headers, check that it is a load module;
      copy exec parameters from old address space to system space;
      for (every region attached to process)
            detach all old regions (algorithm detach);
      for (every region specified in load module)
      {
            allocate new regions (algorithm allocreg);
            attach the regions (algorithm attachreg);
            load region into memory if appropriate (algorithm loadreg);
      }
      copy exec parameters into new user stack region;
      special processing for setuid programs, tracing;
      initialize user register save area for return to user mode;
      release inode of file (algorithm iput);
}
```

Figure 6.8: Algorithm for Exec

THE USER ID OF A PROCESS:

The kernel associates two user IDs with a process, independent of the process lD: the real user ID and the effective user ID or setuid (set user ID). The real user lD identifies the user who is responsible for the running process. The effective user ID is used to assign ownership of newly created files, to check file access permissions, and to check permission to send signals to processes via the kill system call.

The kernel allows a process to change its effective user ID when it execs a setuid program or when it invokes the setuid system call explicitly. A setuid program is an executable file that has the setuid bit set in its permission mode field.

When a process execs a setuid program, the kernel sets the effective user ID fields in the process table and u area to the owner ID of the file. To distinguish the two fields, let us call the field in the process table the saved user ID. An example illustrates the difference between the two fields.

The syntax for the setuid system call is ***setuid(uid)***; where uid is the new user ID, and its result depends on the current value of the effective user ID. If the effective user ID of the calling process is super user, the kernel resets the real and effective user ID fields in the process table and u area to uid.

If the effective user ID of the calling process is not super user, the kernel resets the effective user ID in the u area to uid if uid has the value of the real user ID or if it has the value of the saved user ID. Otherwise, the system call returns an error. Generally, a process inherits its real and effective user IDs from its parent during the fork system call and maintains their values across exec system calls. The mkdir command is a typical setuid program.

CHANGING THE SIZE OF A PROCESS:

A process may increase or decrease the size of its data region by using the brk system call. The syntax for the brk system call is ***brk(endds);*** where endds becomes the value of the highest virtual address of the data region of the process (called its break value).

Alternatively, a user can call ***oldendds = sbrk(increment);*** where increment changes the current break value by the specified number of bytes, and oldendds is the break value before the call. Sbrk is a C library routine that calls brk. If the data space of the process increases as a result of the call, the newly allocated data space is virtually contiguous to the old data space; that is, the virtual address space of the process extends continuously into the newly allocated data space.

The kernel checks that the new process size is less than the system maximum and that the new data region does not overlap previously assigned virtual address space (figure 6.9).

If all checks pass, the kernel invokes growreg to allocate auxiliary memory (e.g., page tables) for the data region and increments the process size field. On a swapping system, it also attempts to allocate memory for the new space and clear its contents to zero; if there is no room in memory, it swaps the process out to get the new space.

If the process is calling brk to free previously allocated space, the kernel releases the memory; if the process accesses virtual addresses in pages that it had released, it incurs a memory fault.

```
algorithm brk
input:   new break value
output: old break value
{
        lock process data region;
        if (region size increasing)
                if (new region size is illegal)
                {
                        unlock data region;
                        return(error);
                }
        change region size (algorithm growreg);
        zero out addresses in new data space;
        unlock process data region;
}
```

Figure 6.9: Algorithm for brk

THE SHELL:

The shell reads a command line from its standard input and interprets it according to a fixed set of rules. The standard input and standard output file descriptors for the login shell are usually the terminal on which the user logged in.

If the shell recognizes the input string as a built-in command (for example, commands cd, for, while and others), it executes the command internally without creating new processes; otherwise, it assumes the command is the name of an executable file.

The simplest command lines contain a program name and some parameters, such as

who

grep -n include *.c

Is -I

The shell forks and creates a child process, which ***execs*** the program that the user specified on the command line. The parent process, the shell that the user is using, waits until the child process exits from the command and then loops back to read the next command.

To run a process asynchronously (in the background), as in

nroff -mm bigdocument &

the shell sets an internal variable ***amper*** when it parses the ampersand character. If it finds the variable set at the end of the loop, it does not execute wait but immediately restarts the loop and reads the next command line.

The figure shows that the child process has access to a copy of the shell command line after the fork. To redirect standard output to a file, as in

nroff -mm bigdocument > output

the child creats the output file specified on the command line; if the creat fails (for creating a file in a directory with wrong permissions, for example), the child would exit immediately. But if the creat succeeds, the child closes its previous standard output file and ***dups*** the file descriptor of the new output file. The standard output file descriptor now refers to the redirected output file. The child process closes the file descriptor obtained from creat to conserve file descriptors for the ***exceed*** program. The shell redirects standard input and standard error files in a similar way.

SYSTEM BOOT AND THE INIT PROCESS:

To initialize a system from an inactive state, an administrator goes through a "bootstrap" sequence: The administrator "boots" the system. Boot procedures vary according to machine type, but the goal is common to all machines: to get a copy of the operating system into machine memory and to start executing it. This is usually done in a series of stages; hence the name bootstrap.

The administrator may set switches on the computer console to specify the address of a special hardcoded bootstrap program or just push a single button that instructs the machine to load a bootstrap program from its microcode. This program may consist of only a few instructions that instruct the machine to execute another program.

On UNIX systems, the bootstrap procedure eventually reads the boot block (block 0) of a disk, and loads it into memory. The program contained in the boot block loads the kernel from the file system (from the file "/unix", for example, or another name specified by an administrator). After the kernel is loaded in memory, the boot program transfers control to the start address of the kernel, and the kernel starts running (***algorithm start Figure 6.10***).

The kernel initializes its internal data structures. For instance, it constructs the linked lists of free buffers and inodes, constructs hash queues for buffers and inodes, initializes region structures, page table entries, and so on. After completing the initialization phase, it mounts the root file system onto root ("/") and fashions the environment for process 0, creating a u area, initializing slot 0 in the process table and making root the current directory of process 0, among other things.

When the environment of process 0 is set up, the system is running as process 0. Process 0 forks, invoking the fork algorithm directly from the kernel, because it is executing in kernel mode. The new process, process 1, running in kernel mode, creates its user-level context by allocating a data region and attaching it to its address space.

It grows the region to its proper size and copies code from the kernel address space to the new region: This code now forms the user-level context of process 1. Process 1 then sets up the saved user register context, "returns" from kernel to user mode, and executes the code it had just copied from the kernel.

Process 1 is a user-level process as opposed to process 0, which is a kernel-level process that executes in kernel mode. The text for process 1, copied from the kernel, consists of a call to the exec system call to execute the program "/etc/init". Process 1 calls exec and execute the program in the normal fashion. Process 1 is commonly called init because it is responsible for initialization of new processes.

```
algorithm start                /* system startup procedure */
input:  none
output: none
{
        initialize all kernel data structures;
        pseudo-mount of root;
        hand-craft environment of process 0;
        fork process 1:
        {
                /* process 1 in here */
                allocate region;
                attach region to init address space;
                grow region to accommodate code about to copy in;
                copy code from kernel space to init user space to exec init;
                change mode: return from kernel to user mode;
                /* init never gets here---as result of above change mode,
                 * init exec's /etc/init and becomes a "normal" user process
                 * with respect to invocation of system calls
                 */
        }
        /* proc 0 continues here */
        fork kernel processes;
        /* process 0 invokes the swapper to manage the allocation of
         * process address space to main memory and the swap devices.
         * This is an infinite loop;  process 0 usually sleeps in the
         * loop unless there is work for it to do.
         */
        execute code for swapper algorithm;
}
```

Figure 6.10 Algorithm for Booting the system

The init process is a process dispatcher, spawning processes that allow users to log in to the system, among others. lnit reads the file “/etc/inittab”, for instructions about which processes to spawn. The file “/etc/inittab” contains lines that contain an "id," a state identifier (single user, multi-user, etc.), an “action”, and a program specification.

lnit reads the file and, if the state in which it was invoked matches the state identifier of a line, creates a process that executes the given program specification.