Unit- 4
Process Control

Process Creation

The only way to create a new process in UNIX is to use the *fork* system call. The process which calls *fork* is called the *parent* process and the newly created process is called the *child* process.

```
pid = fork();
```

On return of the *fork* system call, the two processes have identical user-level context except for the value of *pid*. *pid* for the parent process is the process ID of the *child* process. And *pid* for child process is 0. The process 0 is the only process which is not created via *fork*.

The steps followed by the kernel for *fork* are:

1. It creates a new entry in the process table.
2. It assigns a unique ID to the newly created process.
3. It makes a logical copy of the regions of the parent process. If a regions can be shared, only its reference count is incremented instead of making a physical copy of the region.
4. The reference counts of file table entries and inodes of the process are increased.
5. It turned the child process ID to the parent and 0 to the child.

The kernel first checks if it has enough resources to create a new process. In 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.

The kernel assigns a unique ID to a process. It is one greater than the previously assigned ID. If another process has the proposed ID number, the kernel attempts to assign the next higher ID number.

When ID numbers reach the maximum value, assignment starts from 0 again.

Ordinary users cannot create a process that would occupy the last remaining slot in the process table. On the other hand, a super user can create as many processes as it wants (limited by the size of the process table.)

The kernel assigns parent process ID in the child slot, putting the child in process tree structure, and initialize various scheduling parameters, such as the initial priority value, initial CPU usage, and other timing fields.

The initial state of the process is "being created".

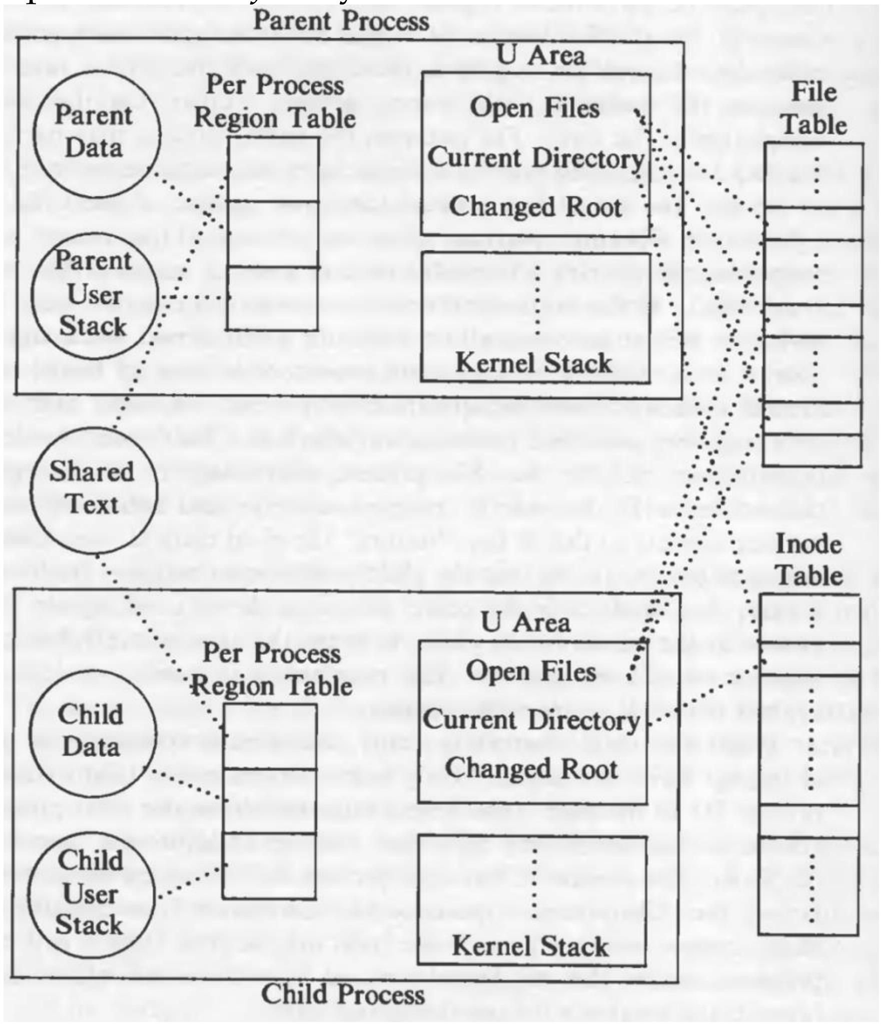
The kernel duplicates every region in the parent process using algorithm *dupreg*, and attaches every region to the child process using algorithm *attachreg*. In a swapping system, it copies the contents of regions that are not shared into a new area of main memory.

The kernel copies the parent context layer 1, containing user saved register context and the kernel stack frame of the *fork* system call. If the implementation is one where the kernel stack is part of the u-area, the kernel automatically creates the child kernel stack when it creates the child u-area. Otherwise, the parent process must copy its kernel stack to a private area of memory associated with the child process. The kernel then creates a dummy context layer 2 for the child process, containing the saved register context for context layer 1.

It sets the program counter and other registers in the saved register context so that it can "restore" the child context, even though it had never executed before, and so that the child process can recognize itself as the child when it runs. For instance, if the kernel code tests the value of register 0 to decide if the process is the parent or the child, it writes the appropriate value in the child saved register context in layer 1.

When the child context is ready, the parent completes its part of *fork* by changing the child state to "ready to run (in memory)" and by returning the child process ID to the user. The kernel later schedules the child process for execution via the normal scheduling algorithm, and the child process "completes" its part of the *fork*. The context of the child process was set up by the parent process; to the kernel, the child process appears to have awakened after awaiting a resource. The child process executes part of the code for the *fork* system call, according to the program counter that the kernel restored from the saved register context in context layer 2, and returns a 0 from the system call.

The figure give a logical view of the parent and child processes and their relationship with the kernel data structures immediately after completion of the *fork* system call:



Consider a program where a process has some global variables and has opened some files. After opening the files, the process *fork*s a child process. In this scenario, as the data region was copied, both the processes have their own copies of the global variables, and changing a variable in one process' context will not affect the variable in other process' context. But as the user file descriptor entries of the two processes point to the same file table entry, if one

process *read*s/*write*s a file, the offset in the file table will change and the other process will get affected, because when it tries to *read*/*write*, it will do it with respect to the changed offset.

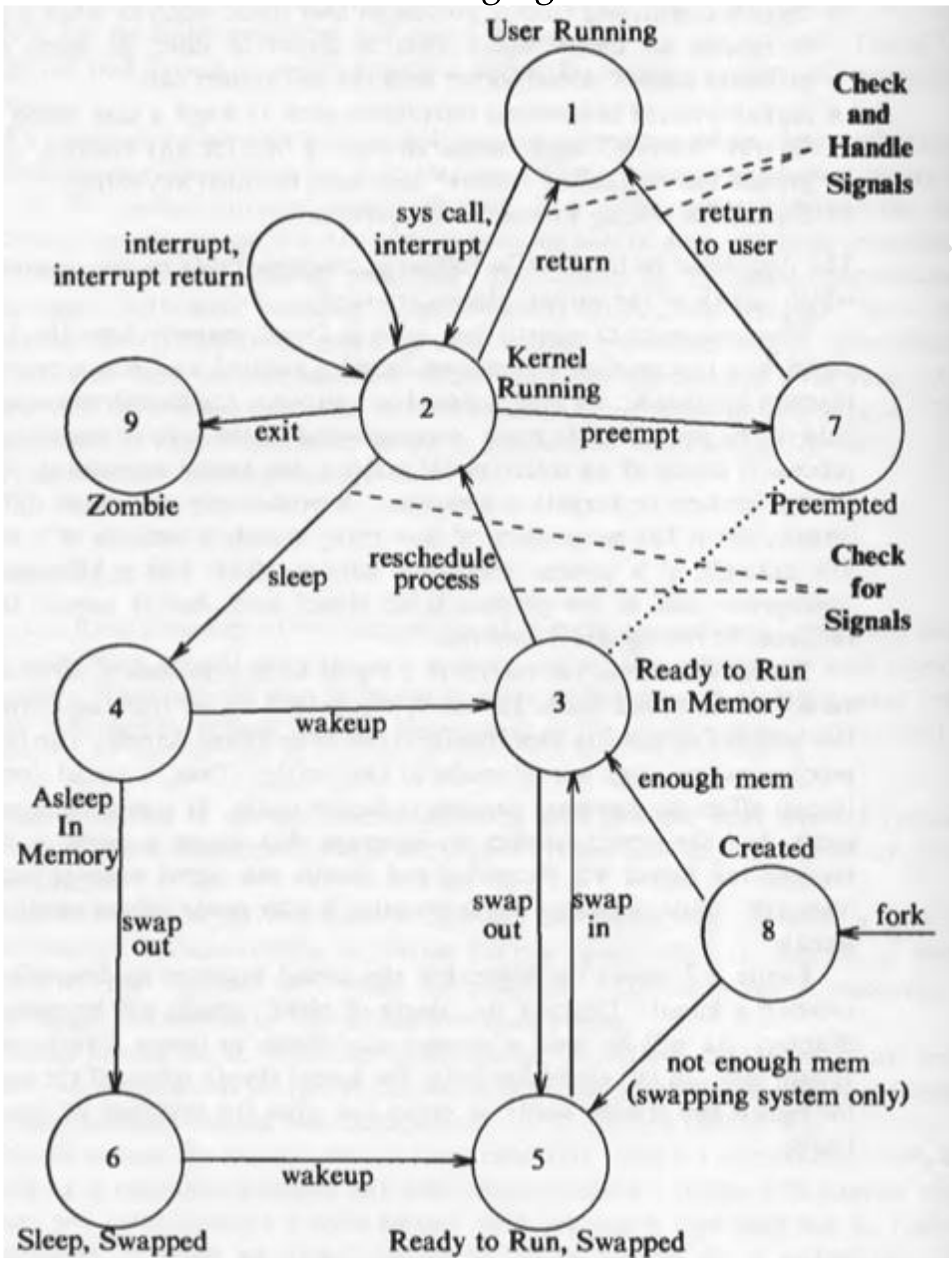
Signals

*Signal*s inform processes of the occurrence of asynchronous events. Processes may send each other *signal*s 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, send when a process *exit*s 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 has been released
- Signals caused by an unexpected error condition during a system call, such as making a nonexistent system call, writing a pipe that has no reader processes, or using an illegal "reference" value for the *lseek* system call. It would be more consistent to return an error on such system calls instead of generating a signal, but the use of signals to abort misbehaving processes is more pragmatic.
- 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 hands 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.

When a kernel or a process sends a signal to another process, a bit in the process table entry of that process is set, with respect to the type of signal received. If the process is asleep at an interruptible priority, the kernel awakens it. A process can remember different types of signals but it cannot remember how many times a signal of a particular type was received.

The kernel checks for receipt of a signal when a process about to return from kernel mode to user mode and when it enters or leaves the sleep state at a suitably low scheduling priority. The kernel handles signals only when a process returns from kernel mode to user mode. This is shown in the following figure:



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.

Handling 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 *exit*s 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.

old function = 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 (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.

These signals do no imply that anything is wrong with the process. The *quit* signal, however, induces a core dump even though it is initiated outside the running process. If a process decides to ignore the signal, or receipt of the signal, the signal field is not reset and the process will continue ignoring the signal.

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. The user stack looks as if the process had called 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.

An important thing to note about signals is that, if a process invokes the *signal* system call with "death of child" parameter, the kernel sends the calling process a "death of child" signal if it has child processes in the zombie state. The reason behind this is discussed later.

Process Groups

The system has to identify processes by "groups" in some cases, for instance, a signal might relate to all the processes which are ancestors of the login shell. The kernel uses the *process group ID* to identify groups of related processes that should receive a common signal for certain events. It saves the group ID in the process table.

The *setpgrp* system call initializes the process group number of a process and sets it equal to the value of its process ID.

```
grp = setpgrp();
```

where grp is the new process group number. A child retains the process group number of its parent during *fork*.

Sending Signals from Processes

Processes use the *kill* system call to send signals.

```
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 superuser, 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 superuser, 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 the UNIX system exit by executing the *exit* system call. When a process *exit*s, it enters the zombie state, relinquishes all of its resources, and dismantles its context except for its process table entry.

```
exit (status);
```

where status is the exit code returned to the parent. The process may call *exit* explicitly, but the startup routine in C calls *exit* after the *main* function returns. The kernel may call *exit* on receiving an uncaught signal. In such cases, the value of *status* is the signal number. In the "write accounting record" step in the above algorithm, it saves its exit code and user and kernel execution time of itself and its descendants in the process table entry. It also *write*s accounting data to a global accounting file, containing various run time statistics such as user ID, memory and CPU usage and amount of I/O for the process. The kernel never schedules a zombie process to execute.

Awaiting Process Termination

A process can synchronize its execution with the termination of a child process by executing the *wait* system call.

```
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.

The kernel adds the accumulated time the child process executed in user and in the kernel mode to the appropriate fields in the parent process u-area.

If the signal is "death of child", the process responds differently:

- In the default case, it will wake up from its sleep in *wait*, and *sleep* invokes algorithm *issig* to check for signals. *issig* recognizes the special case of "death of child" signals and returns "false". Consequently, the kernel does not "long jump" from *sleep*, but returns *wait*. The kernel will restart the *wait* loop, find a zombie child -- at least one is guaranteed to exist, release the child's process table slot, and return from the *wait* system call.
- If the process catches "death of child" signals, the kernel arranges to call the user signal-handler routine, as it does for other signals.
- If the process ignores "death of child" signals, the kernel restarts the *wait* loop, frees the process table slots of zombie children, and searches for more children.

Invoking Other Programs

The *exec* system call overlays the address space of a process with the contents of an executable file.

```
execve (filename, argv, envp)
```

where filename is name of the file being invoked, argv is a pointer to array of character pointers which are arguments to the program in the executable file, and envp is a pointer to array of character pointers which are the *environment* of the executed program. There are several library functions that call *exec*, like *execl*, *execv*, *execle*, and so on.

All call *execve* eventually. The character strings in the *envp* array are of the form, "name=value".

The kernel examines the file header to determine the layout of the executable file. The logical format of an executable file as it exists in the file system is shown in the diagram below:

```
![Logical layout of executable file](Diagrams/Screen Shot 2017-06-22 at 10.02.55 AM.png)
```

It consists of four parts:

1. The primary header describes how many sections are in the file, the start address for process execution, and the *magic number*, which gives the type of the executable file.
2. Section headers describe each section in the file, giving the section size, the virtual addresses the section should occupy when running in the system, and other information.

3. The sections contain the "data", such as text, that are initially loaded in the process address space.
4. Miscellaneous sections may contain symbol tables and other data, useful for debugging.

The magic number is a short integer, which identifies the file as a load module and enables the kernel to distinguish run-time characteristics about it. Magic number plays an important role in paging systems .

The kernel has to copy the parameters passed to exec to a holding space as the address space is going to be overlaid. It usually copies the parameter into kernel memory. The use of kernel stack for saving the copied parameters is also common, but there is a limit to the kernel stack and the parameters can be of arbitrary length.

The kernel usually copies them to a space which is accessible faster. Hence, use of primary memory is preferable that secondary memory (swap device).

The kernel allocates and attaches regions for text and data, loading the contents of the executable file into main memory (algorithms *allocreg*, *attachreg*, and *loadreg*). The data region of a process is (initially) divided into two parts: data initialized at compile time and data not initialized at compile time ("bss").

The initial allocation and attachment of the data region is for the initialized data. The kernel then increases the size of the data region is for the initialized data. The kernel then increases the size of the data region using algorithm *growreg* for the "bss" data, and initializes the value of the memory to 0.

Finally, it allocates a region for the process stack, attaches it to the process, and allocates memory to store *exec* parameters. If the kernel has saved the *exec* parameters in memory pages, it can use those pages for the stack. Otherwise, it copies the *exec* parameters to the user stack.

The kernel takes special action for *setuid* programs and for process tracing . After *exec*, the process ID doesn't change. Only user level context changes. The reason why text and data regions are separate, is primarily to protect the text region. The kernel can use hardware protection for the text region and any attempt to overwrite the text region, results into a protection fault that typically results in termination of the process.

If the text region is protected, it does not change from the time kernel loaded it into the memory. Therefore, several processes can share the text region, saving memory. Thus, when the kernel allocates a text region for a process in *exec*, it checks if the executable file allows its text to be shared, indicated by its magic number.

Of course, the kernel frees the regions only if no processes currently use it. The scenario for *exec* is slightly more complicated if a process *exec*s itself. If a user types sh script the shell *fork*s and the child process *exec*s the shell and executes the commands in the file "script".
If a process *execIs itself and allows sharing of its text region, the kernel must avoid deadlocks over the inode and region locks. That is, the kernel cannot lock the "old" text region, hold the lock, and then attempt to lock the "new" text region, because the old and new regions are one region. Instead, the kernel simply leaves the old text region attached to the process, since it will be reused anyway.

The User ID of a Process

There are two *user ID* associated with a process, the *real user ID* and the *effective user ID* or *setuid* (set user ID). The real user ID 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 *exec*s 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 *exec*s 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.

```
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 superuser, the kernel resets the real and effective user ID fields in the process table and u-area to *uid*. If its not the superuser, 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 *login* calls *setuid* system call. *login* is *setuid* to root (superuser) and therefore runs with *effective user ID* root. When login is successful, it calls *setuid* system call to set its real and effective user ID to that of the user trying to log in (found in fields in the file "/etc/passwd").
The *mkdir* command is a typical *setuid* program. To allow ordinary users to create directories, the *mkdir* command is a *setuid* program owned by root (superuser permission).

Changing the Size of a Process

A process can increase or decrease the size of its data region by using the *brk* system call.

```
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*. 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.
When the user stack overflows, the kernel extends it using *brk*.

The Shell

The shell is a very complex program. But this section will describe the shell, apart from the complex parts. The main loop of the shell looks like this:

```
// read command line until "end of file"
while (read(stdin, buffer, numchars))
{
        // parse command line
        if (/* command line contains & */)
                amper = 1;
        else
                amper = 0;
        // for commands not part of the shell command language
        if (fork() == 0)
        {
                // redirection of IO?
                if (/* redirect output */)
                {
                        fd = creat(newfile, fmask);
                        close(stdout);
                        dup(fd);
                        close(fd);
                        // stdout is now redirected
                }
                if (/* piping *)
                {
                        pipe (fildes);
                        if (fork() == 0)
                        {
                                // first component of command line
                                close(stdout);
                                dup(fildes[1]);
                                close(fildes[1]);
                                close(fildes[0]);
                                // stdout now goes to pipe
                                // child process does command
                                execlp(command1, command1, 0);
                        }
                        // 2nd command component of command line
                        close(stdin);
                        dup(fildes[0]);
                        close(fildes[0]);
                        close(fildes[1]);
                        // standard input now comes from pipe
                }
                execve(command2, command2, 0);
        }
        // parent continues over here...
        // waits for child to exit if required
        if (amper == 0)
                retid = wait(&status);
}
```
The standard input and 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, *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.
If a *&* character is postfixed to the command, the shell runs the *exec*ed process asynchronously. Otherwise, it will wait for the *exec*ed process to finish execution.
When input or output is redirected using the >, <, or >2 characters, the shell redirects the output by closing the *stdin*, *stdout*, whichever applicable, or both and then *creat*ing new file(s) and using *dup* for actual redirection. (In the code above, the redirection of *stdout* is shown, same method is used for *stdin* and *stderr*.)
The code shows how the shell could handle a command line with a single *pipe*, as in
ls -l | wc
This will result in the output of ls -l to be passed to wc as its input. For doing this, the shell pipes the output of ls to the input of wc. Here, the child of shell creates another child (grandchild of the shell).
The shell keeps looping and *read*ing the commands.

System Boot and the Init Process

But procedures vary according to machine types, but the goal is to copy the operating system into machine memory and start executing it. This is done in a series of steps and hence called as the "bootstrap". On UNIX machines, the bootstrap finally reads the boot block (block 0) of a disk and loads it into memory. The program in the boot block loads the kernel from the file system (for example, from the file "/unix"). Then the boot block program transfers control to the start address of the kernel and the kernel starts running. The algorithm *start* is given below:
*/

```
{
        fd = open("/etc/inittab", O_RDONLY);
        while (line_read(fd, buffer))
        {
                // read every line of file
                if (invoked state != buffer state)
                        continue;               // loop back to while
                // state matched
                if (fork() == 0)
```

```
            {
                     execl("process specified in buffer");
                     exit();
            }
            // init process does not wait
            // loop back to while
        }

        while ((id = wait((int *) 0)) != -1)
```

init 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. This is shown below:

init 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. Meanwhile, *init* executes the *wait* system call, monitoring the death of its child processes and the death of processes "orphaned" by *exit*ing parents.

Processes in the UNIX system are either user processes, daemon processes, or kernel processes. Most processes are user processes, associated with users at a terminal. Daemon processes are not associated with any users but do system-wide functions, such as administration and control of networks, execution of time-dependent activities, and so on. They are like user processes in that they run at user mode and make system calls to access system services.

Process Scheduling and time

On a time sharing system, the kernel allocates CPU to a process for a period of time called the time slice or time quantum. After the time quantum expires, it preempts the process and schedules another one. The scheduler in UNIX uses relative time of execution as a parameter to determine which process to schedule next.

Every process has a priority associated with it. Priority is also a parameter in deciding which process to schedule next. The kernel recalculates the priority of the running process when it comes to user mode from kernel mode, and it periodically re-adjusts the priority of every "ready-to-run" process in user mode.

Process Scheduling

The scheduler on the UNIX system belongs to the general class of operating system schedulers knows as *round robin with multilevel feedback*. That means, when kernel schedules a process and the time quantum expires, it preempts the process and adds it to one of the several priority queues.

The algorithm *schedule_process* is given below:

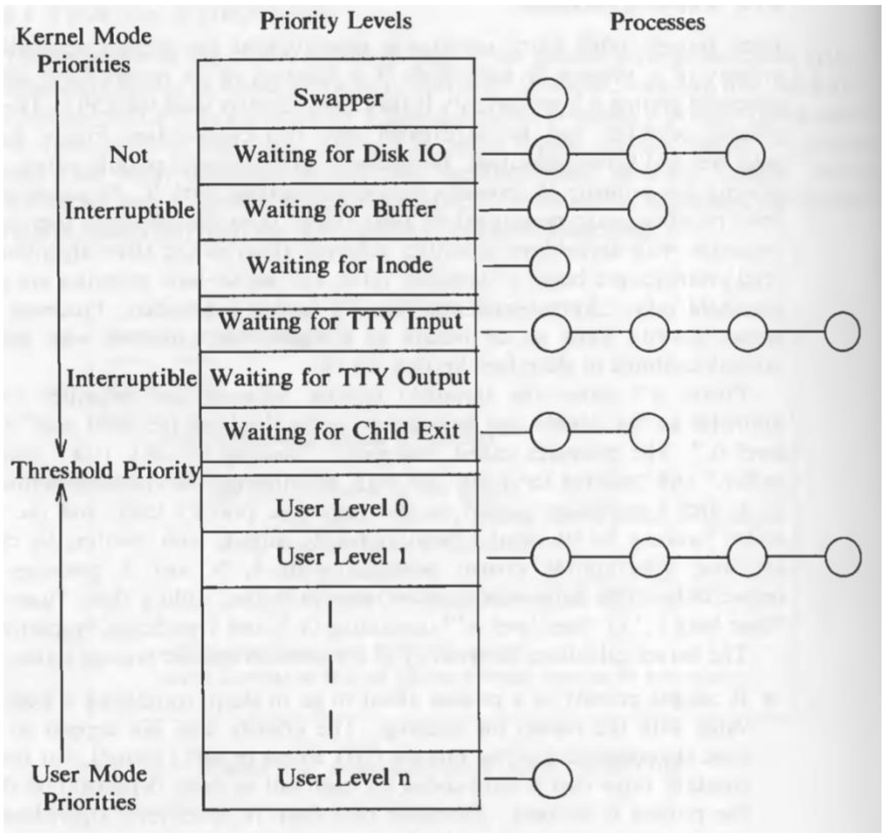
```
/*  Algorithm: schedule_process
 *  Input: none
 *  Output: none
 */

{
        while (no process picked to execute)
        {
                for (every process on run queue)
                        pick highest priority process that is loaded in
memory;
                if (no process eligible to execute)
                        idle the machine;
                        // interrupt takes machine out of idle state
        }
        remove chosen process from run queue;
        switch context to that of chosen process, resume its execution;
}
```

This algorithm is executed at the conclusion of a context switch. It selects the highest priority process from the states "ready to run, loaded in memory" and "preempted". If several processes have the same priority, it schedules the one which is "ready to run" for a long time.

Scheduling Parameters

Each process table entry contains a priority field. The priority is a function of recent CPU usage, where the priority is lower if a process has recently used the CPU. The range of priorities can be partitioned in two classes: user priorities and kernel priorities. It is shown in the diagram below:



Each priority has a queue of processes logically associated with it. The processes with user-level priorities were preempted on their return from the kernel to user mode, and processes with kernel-level priorities achieved them in the *sleep* algorithm.

The kernel calculates process priorities in these process states:

- It assigns priority to a process about to go to sleep. This priority solely depends on the reason for the sleep. Processes that sleep in lower-level algorithms tend to cause more system bottlenecks the longer they are inactive; hence they receive a higher priority than process that would cause fewer system bottlenecks. For instance, a process sleeping and waiting for the completion of disk I/O has a higher priority than a process waiting for a free buffer. Because the first process already has a buffer and it is possible that after the completion of I/O, it will release the buffer and other resources, resulting into more resource availability for the system.
- The kernel adjusts the priority of a process that returns from kernel mode to user mode. The priority must be lowered to a user level priority. The kernel penalizes the executing process in fairness to other processes, since it had just used valuable kernel resources.

- The clock handler adjusts the priorities of all processes in user mode at 1 second intervals (on System V) and causes the kernel to go through the scheduling algorithm to prevent a process from monopolizing use of the CPU.

System Calls for Time

There are several time-related system calls, *stime*, *time*, *times*, and *alarm*. The first two deal with global system time, and the latter two deal with time for individual processes. *stime* allows the superuser to set a global kernel variable to a value that gives the current time:

```
stime(pvalue);
```

where pvalue points to a long integer that gives the time as measured in seconds from midnight before (00:00:00) January 1, 1970, GMT. The clock interrupt handler increments the kernel variable once a second. *time* retrieves the time as set by *stime*.

```
time(tloc);
```

where tloc points to a location in the user process for the return value. *time* returns this value from the system call, too.
times retrieves the cumulative times that the calling process spent executing in user mode and kernel mode and the cumulative times that all zombie children had executed in user mode and kernel mode.

Clock

The functions of the clock interrupt handler are to:

- restart the clock
- schedule invocation of internal kernel functions based on internal timers
- provide execution profiling capability for the kernel and for user processes
- gather system and process accounting statistics,
- keep track of time
- send alarm signals to processes on request
- periodically wake up the swapper process
- control process scheduling

Some operations are done every clock interrupt, whereas others are done after several clock ticks. The clock handler runs with processor execution level set high.

Keeping Time

The kernel increments a timer variable at every clock interrupt, keeping time in clock ticks from the time the system was booted. The kernel saves the process start time in its u-area when a process is created in the *fork* system call, and it subtracts that value from the current time when the process *exit*s, giving the real execution time of a process. Another timer variable, set by the *stime* system call, is updated once a second, keeps track of calendar time.