The Unix Kernel

Mark Sitkowski C.Eng, M.I.E.E
http://www.designsim.com.au

When Unix was first introduced, it was usually described as having a 'shell', or user interface, which surrounded a 'kernel' which interpreted the commands passed to it from the shell.
With the passage of time, and the advent of X-windows and TCP/IP, both of which have direct paths to the kernel, this model is a bit strained at the edges, but it still provides a useful mental image of the system.
The following is not intended to be an exhaustive treatise on the inner workings of the Unix kernel, nor is it specific to any particular brand of Unix. However, it is essential to broadly understand certain important functions of the kernel, before we can take advantage of some of its features, to improve the way in which it handles our process.

The Unix kernel possesses the following functionality, much of which is of interest to us, in pursuit of this goal:

- **System calls**.
  All of the most basic operating system commands are performed directly by the kernel. These include:
  - open()
  - close()
  - dup()
  - read()
  - write()
  - fcntl()
  - ioctl()
  - fork()
  - exec()
  - kill()

  Since the above commands are actually executed by the kernel, the 'C' compiler doesn't need to generate any actual machine code to perform the function. It merely places a 'hook' in the executable, which instructs the kernel where to find the function.
  Modern, third party compilers, however, are ported to a variety of operating systems, and will generate machine code for a dummy function, which itself contains the 'hook'.

- **Process scheduling and control**
  The kernel determines which processes will run, when and for how long. We will examine this mechanism, in detail, later.

- **Networking**
  That, which became networking, was originally designed so that processes could communicate. It is this ability, to pass information quickly and efficiently from one running process to another, which makes the Unix operating system uniquely

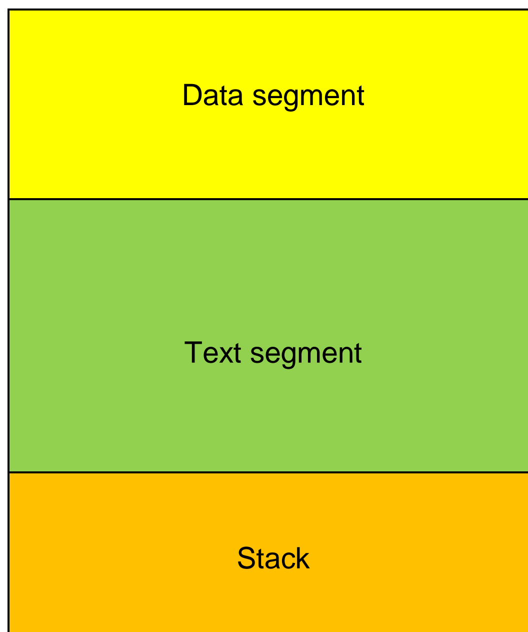
capable of multi-dimensional operation. All of the most important communication commands are the system calls.

- socket()
- connect()
- bind()
- listen()
- accept()
- send()
- recv()

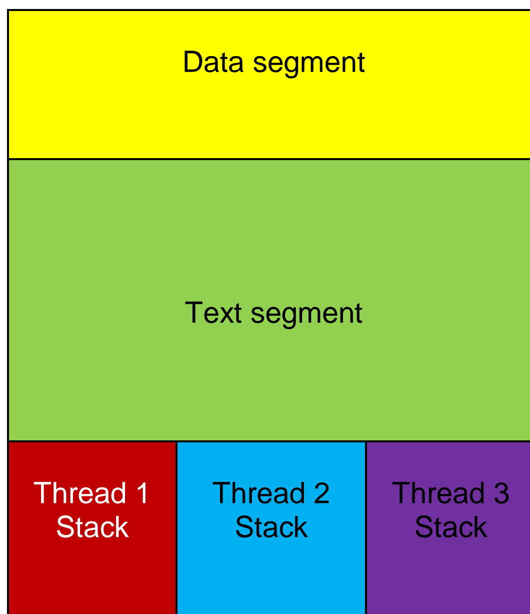
- **Device drivers for all supported hardware devices.**
  Unlike other operating systems, where device drivers are separate programs, which are individually loaded into memory, the Unix kernel inherently contains all of the machine's drivers. Contrary to what may be supposed, the entries in the /dev directory are not drivers, they are just access points into the appropriate kernel routine. We will not concern ourselves unduly with the Unix device drivers, as they are outside the scope of this book.

Anatomy of a process

Single-threaded:



Multi-threaded:



When an executable is invoked, the following events occur, although not necessarily in this order.

Process Loading

1. The loader

- Fetches the executable file from the disk
- Allocates memory for all of the global variables and data structures, ('the data segment') and loads the variables into that area of memory.
- Loads the machine code of the executable itself ('the text segment') into memory. With demand-paged executables, this is not strictly the case, as the amount of code actually loaded into memory is several 4k pages. The remainder is put into the swap area of the disk.
- Searches the header portion of the executable, for any dynamically-linked libraries or modules.
- Checks to see if these are already loaded and, If not, the modules are loaded into memory, otherwise, their base addresses are noted.
- Makes available the base addresses of dynamically-linked modules/libraries to the process.
- Allocates an area of memory for the stack. If the process is multi-threaded, a separate stack area is allocated for each thread.

2 The kernel

- Sets the program counter to the first byte of executable code.
- Allocates a slot in the process table to the new process
- Allocates a process ID to the new process
- Allocates table space for any file descriptors
- Allocates table space for any interrupts.
- Sets the 'ready to run' flag of the process.

All of the above resources, allocated to a given process, constitute the 'context' of a process. Each time the kernel activates a new process, it performs a context switch,

by replacing the resources of the previously running process with those of the current one.

At this point, the scheduling algorithm takes over.

The Process Scheduling algorithm

While the process is running, it runs in one of two modes:

1. **Kernel mode**
   All system calls are executed by the kernel, and not by machine code within the process. Kernel mode has one very desirable characteristic, and that is the fact that system calls are atomic and, hence, cannot be interrupted. One of the most important factors in writing code for high-performance applications, is to ensure that your process executes as much in kernel mode as possible. This way, you can guarantee the maximum CPU time for a given operation.
   Kernel threads, such as pthreads, run in kernel mode. It is sometimes worth using a thread, even when doing so doesn't constitute parallel operation, purely to get the advantage of running in kernel mode.
   If an interrupt occurs, while in this mode, the kernel will log the signal in the interrupt table, and examine it after the execution of the current system call. Only then will the signal be actioned.
2. **User mode**
   The ordinary machine code, which makes up much of the executable, runs in user mode. There are no special privileges associated with user mode, and interrupts are handled as they arrive.

It may be seen that, during the time that a process runs, it is constantly switching between kernel mode and user mode. Since the mode switch occurs within the same process context, it is not a computational burden.

A Unix process has one of the following states:

- Sleep
- Run
- Ready to Run
- Terminated

The scheduling algorithm is a finite-state machine, which moves the status of the process between states, depending on certain conditions.

Basically, what happens is this.

A process begins to execute. It runs until it needs to perform I/O, then, having initiated the I/O, puts itself to sleep. At this point, the kernel examines the next process table slot and, if the process is ready to run, it enables its execution.
If a process never performs I/O, such as processes which perform long series of floating point calculations, the kernel permits it to only run for a fixed time period, of between 20 and 50 milliseconds, before pre-empting it, and enabling the next eligible process.
When the time comes for a process to run, an additional algorithm determines the priority of one process over another. The system clock sends the kernel an interrupt, once per second, and it is at this time, that the kernel calculates the priorities of each process. Leaving aside the user-level priority weighting, determined by 'nice' the

kernel determines priority based on several parameters, of which these are significant::

- How much CPU time the process has previously used
- Whether it is waking up from an I/O wait or not
- Whether it is changing from kernel mode to user mode or not

Performance Note

It may be seen from the above, that processes, which are designed for performance-critical applications, should avoid doing physical I/O until it is absolutely necessary, in order to maximise the amount of contiguous CPU time. If it is at all possible, all of the I/O operations should be saved up, until all other processing has completed, and then performed in one operation, preferably, by a pthread.
Further, if we consider a situation, where we have 100 processes running on a machine, and one of them is ours, then we would expect to use 1% of the CPU time. However, if 25 of them are ours, we would be eligible to use 25% of the CPU time. Thus, If an application can split itself into several processes, running concurrently, then, quite apart from the obvious advantages of parallelism, we will capture more of the machine's resources, just because each child process occupies a separate process table slot.
This also helps, when the kernel assigns priorities to processes. Even though we may be penalised for using a lot of CPU time, the priority of each process is rated against that of other processes. If many of these belong to one application then even though the kernel may decide to give one process priority over another, the application, as a whole, will still get more CPU time.
Additionally, if we are running on a multi-processor machine, then we can almost guarantee to be given a separate CPU for each child process. The kernel may juggle these processes over different CPU's, as a part of its load-balancing operations, but each child will still have its own processor.
This type of architecture forms the cornerstone of multi-dimensional programming.

Swapping and Paging

Of course, the execution of a process is never that straightforward. Only a portion of the code is loaded into memory, meaning that it can only run until another page needs to be fetched from the disk. When this occurs, the process generates a 'page fault' which causes the pager to go and fetch the appropriate page. A similar situation occurs when a branch instruction is executed, which takes the execution point to a page other than those stored in memory. The paging mechanism is fairly intelligent, and contains algorithms similar to those found in CPU machine instruction pipeline controllers. It tries to anticipate branch instructions, and pre-fetch the anticipated page, more or less successfully, depending on the code structure.

Although there are certain similarities, paging, as described above, which is a natural result of process execution, should not be confused with swapping.
If the number of processes grows, to the extent that all available memory becomes used up, the addition of another process will trigger the swapper, and cause it to take a complete process out of memory, and place it in the swap area of the disk.
This is, computationally, an extremely expensive operation, as the entire process, together with its context, has to be written to disk then, when it is permitted once again to run, another process must be swapped out, to make space for it to be reloaded into memory.

System calls

1. fork()

Under pre-Unix operating systems, starting a process from within another process was traditionally performed as a single operation. One command magically placed the executable into memory, and handed over control and ownership to the operating system, which made the new process run.
Unix doesn't do that.
Each process has a hierarchical relationship, with its parent, which is the process which brought it to life, and with its child or children which, in turn, are processes which it, itself, started. All such related processes are part of a *process group*.
The basic mechanism, which initiates the birth of a new process is *fork().*

The fork() system call makes a running copy of the process which called it. All memory addresses are re-mapped, and all open file descriptors remain open. Also, file pointers maintain the same file position in the child as they do in the parent.

Consider the following code fragment:

```
pid_t pid;

   switch((pid = fork()){
      case –1:
         printf("fork failed\n");
      break;
      case 0:
         printf("Child process running\n");
         some_child_function();
      break;
      default:
         printf("Parent process executes this code\n");
      break;
   }
```

At the time that the fork() system call is called, there is only one process in existence, that of the expectant parent. The local variable pid is on the stack, probably uninitialised.
The system call is executed and, now, there are two identical running processes, both executing the same code. The parent, and the new child process both simultaneously check the variable pid, on the stack. The child finds that the value is zero, and knows, from this, that it is the child. It then executes some_child_function(), and continues on a separate execution path.
The parent does not see zero, so it executes the 'default' part of the switch() statement. It sees the process ID of the new child process, and drops through the bottom of the switch(). Note, that, if we do not call a different function in the case 0: section of the switch, that both parent and child will continue to execute the same code, since the child will also drop through the bottom of the switch().

Programmers who know little about Unix, will have a piece of folklore rattling around in their heads, which says 'a fork() is expensive. You have to copy an entire process in memory, which is slow, if the process is large'.
This is true, as far as it goes. There is a memory-to-memory copy of that part of the parent, which is resident in memory, so you may have to wait a few milliseconds.

However, we are not concerned with trivial processes whose total run time is affected by those few milliseconds. We are dealing exclusively with processes whose run times are measured in hours, so we consider a one-time penalty of a few milliseconds to be insignificant.

When a parent forks a child process, on a multi-processor machine, the Unix kernel places the child process onto its own, separate CPU. If the parent forks twelve children, on a twelve CPU machine, each child will run on one of the twelve CPU's. In an attempt to perform load-balancing, the kernel will shuffle the processes around the CPU's but, basically, they will remain on separate processors.
The fork() system call is one of the most useful tools, for the full utilisation of a multi-processor machine's resources, and it should be used whenever one or more functions are called, which can proceed their tasks in parallel. Not only is the total run time reduced to that of the longest-running function, but each function will execute on its own CPU.

2. vfork()
   There is a BSD variant of fork(), which was designed to reduce the memory usage overhead associated with copying, possibly, a huge process in memory. The semantics of vfork() are exactly the same as those of fork(), but the operation is slightly different. Vfork() only copies the page, of the calling process. which is currently in memory but, due to a bug (or feature) permits the two processes to share the same stack. As a result, if the child makes any changes to variables local to the function which called vfork(), the changes will be visible to the parent. Knowledge of this fact has enabled experienced programmers to make use of the advantages of vfork(), while avoiding the pitfalls. However, far more subtle bugs also exist, and most Unix vendors recommend that vfork() only be used, if it is immediately followed by an exec().

3. exec()
   The original thinking behind fork(), was that its primary use would be to create new processes – not just copies of the parent process. The exec() system call achieves this, by overlaying the memory image of the calling process with the new process.
   There is a very good reason for separating fork() and exec(), rather than having the equivalent of VMS's spawn() function, which combines the two. That reason is, because it is sometimes necessary, or convenient, to perform some operations in between fork() and exec(). For example, is may be necessary to run the child process as a different user, like root, or to change directory, or both.

   There is, in fact, no such call as 'exec()', but there are two main variants, execl() and execv().
   The semantics of execl() are as follows:

   execl(char *path, char *arg0, char *arg1…char *argn, (char *) NULL)
   execv(char *path, char *arg0, char **argv)

   It may be seen, that the principal difference between the two variants, is that, whereas the execl() family takes a path, followed by separate arguments, in a NULL terminated, comma-separated list, the execv() variants take a path, and a vector, similar to the argv[] vector, passed to a main() function.

   The first variant of execl() and execv(), adds an environment vector to the end of the argument list:

execle(char *path, char *arg0, ….char *argn, (char *) NULL, char **envp)
execve(char *path, char *arg0, char **argv, char **envp)

The second variant replaces the ‘path’ argument, with a ‘file’ argument. If this latter contains a slash, it is used as a path, otherwise, the PATH environment variable of the calling process is used to find the file.

execlp(char *file, char *arg0, ….char *argn, (char *) NULL, char **envp)
execvp(char *file, char *arg0, char **argv, char **envp)

We can now combine fork() and exec(), to execute lpr from the parent process, In order to print a file:

```
pid_t pid;

switch((pid = fork()){
    case –1:
        printf(“fork failed\n”);
    break;
    case 0:
        printf(“Child process running\n”);
        execl(“/usr/ucb/lpr”, “lpr”, “/tmp/file”, (char *) NULL);
    break;
    default:
        printf(“Parent process has executed lpr to print a file\n”);
    break;
}
```

The above code only has one problem. If the parent process quits, the child process will become an orphan, and be adopted by the ‘init’ process. When lpr has run to completion, it will become a zombie process, and waste a slot in the process table. The same happens if the child prematurely exits, due to some fault.
There are two solutions to this problem:

1. We execute one of the wait() family of system calls.
   A waited-for child does not become a zombie, but the parent must suspend processing, until the child terminates, which may or may not be a disadvantage. There are options, which allow processing to continue, during the wait, but the parent needs to poll waitpid(), which makes our second solution, described below, a much better option.

   If we are waiting for a specific process, the most convenient call is to waitpid(). The synopsis of this call is:

   pid_t waitpid(pid_t pid, int *status, int options);

   The call to waitpid() returns the process ID of the child for which we are waiting, whose process ID is passed in as the first argument, ‘pid’.
   The second argument, ‘status’ is the returned child process exit status, and ‘options’ is the bitwise-OR of the following flags:
   WNOHANG : prevents waitpid() from causing the parent process to hang, if there is no immediate return.
   WNOWAIT : keeps the process, whose status is returned, in a waitable state, so that it may be waited for again.

The options flags are of no use to us, so we set them to zero. The status word, however, provides useful information on how our child terminated, and can be decoded with the macros, described in the man page for 'wstat'.

```
pid_t pid;
int status;

   switch((pid = fork()){
      case –1:
         printf("fork failed\n");
      break;
      case 0:
         printf("Child process running\n");
         execl("/usr/ucb/lpr", "lpr", "/tmp/file", (char *) NULL);
      break;
      default:
         printf("Parent process has executed lpr to print a file\n");
          if(waitpid(pid, &status, 0) == pid){
              printf("lpr has now finished\n");
          }
      break;
   }
```

2. If we don't wish to poll waitpid() repeatedly, but need to do other processing, while the child process goes about its business, then we need to, effectively, disown the child process.
   As soon as the child has successfully forked, we must disassociate it from the process group of the parent. This is accomplished by executing the system call setpgrp(), or setsid(), both of which have the same functionality. These calls create a new process session group, make the child process the session leader, and set the process group ID to the process ID of the child.

   The complete code is as below:

```
pid_t pid;

   switch((pid = fork()){
      case –1:
         printf("fork failed\n");
      break;
      case 0:
          if(setpgrp() == -1){
             printf("Can't set pgrp\n");
         }
         printf("Independent child process running\n");
         execl("/usr/ucb/lpr", "lpr", "/tmp/file", (char *) NULL);
      break;
      default:
         printf("Parent process has executed lpr to print a file\n");
      break;
   }
```

1. open() close() dup() read() write()

These system calls are primarily concerned with files but, since Unix treats almost everything as a file, most of them can be used on any byte-orientated device, including sockets and pipes.

- int open(char *file, int how, int mode);
  open() returns a file descriptor to a file, which it opens for reading, writing or both. The 'file' argument is the file name, with or without a path, while 'how' is the bitwise-OR of some of the following flags defined in fcntl.h:

O_RDONLY	Read only
O_WRONLY	Write only
O_RDWR	Read/write
O_TRUNC	Truncate on opening
O_CREAT	Create if non-existent

  The 'mode' argument is optional, and defines the permissions on the file, using the same flags as chmod.

- Int close(int fd);
  Closes the file, which was originally opened with the file descriptor fd.
- Int dup(int fd);
  Returns a file descriptor, which is identical to that passed in as an argument, but with a different number. This call seems fairly useless, at first glance but, in fact, it permits some powerful operations, like bi-directional pipes, where we need a pipe descriptor to become a standard input or output. Also, client-server systems, need to listen for incoming connections on a fixed socket descriptor, while handling existing connections on different descriptors.