Module 14: Unix Kernel Architecture

The kernel runs the show, i.e. it manages all the operations in a Unix flavored environment. The kernel architecture must support the primary Unix requirements. These requirements fall in two categories namely, functions for process management and functions for file management (files include device files). Process management entails allocation of resources including CPU, memory, and offers services that processes may need. The file management in itself involves handling all the files required by processes, communication with device drives and regulating transmission of data to and from peripherals. The kernel operation gives the user processes a feel of synchronous operation, hiding all underlying asynchronism in peripheral and hardware operations (like the time slicing by clock). In summary, we can say that the kernel handles the following operations :

1. It is responsible for scheduling running of user and other processes.
2. It is responsible for allocating memory.
3. It is responsible for managing the swapping between memory and disk.
4. It is responsible for moving data to and from the peripherals.
5. it receives service requests from the processes and honors them.

All these services are provided by the kernel through a call to a system utility. As a result, kernel by itself is rather a small program that just maintains enough data structures to pass arguments, receive the results from a call and then pass them on to the calling process. Most of the data structure is tables. The chore of management involves keeping the tables updated. Implementing such a software architecture in actual lines of code would be very small. The order of code for kernel is only 10000 lines of C and 1000 lines of assembly code.

Kernel also aids in carrying out system generation which ensures that Unix is aware of all the peripherals and resources in its environment. For instance, when a new disk is attached, right from its formatting to mounting it within the file system is a part of system generation.

14.1 User Mode and Kernel Mode

At any one time we have one process engaging the CPU. This may be a user process or a system routine (like ls, chmod) that is providing a service. A CPU always engages a process which is \runnable". It is the task of the scheduler to choose amongst the runnable

processes and give it the control of CPU to execute its instructions. Upon being scheduled to run, the process is marked now to be in \running" state (from the previous runnable state).

Suppose we trace the operation of a user process. At some point in time it may be executing user instructions. This is the user mode of operation. Suppose, later in the sequence, it seeks to get some data from a peripheral. In that event it would need to make a system call and switch to kernel mode.

The following three situations result in switching to kernel mode from user mode of operation:

1. The scheduler allocates a user process a slice of time (about 0.1 second) and then system clock interrupts. This entails storage of the currently running process status and selecting another runnable process to execute. This switching is done in kernel mode. A point that ought to be noted is: on being switched the current process's priority is re-evaluated (usually lowered). The Unix priorities are ordered in decreasing order as follows:
    - HW errors
    - Clock interrupt
    - Disk I/O
    - Keyboard
    - SW traps and interrupts
2. Services are provided by kernel by switching to the kernel mode. So if a user program needs a service (such as print service, or access to another file for some data) the operation switches to the kernel mode. If the user is seeking a peripheral transfer like reading a data from keyboard, the scheduler puts the currently running process to “sleep" mode.
3. Suppose a user process had sought a data and the peripheral is now ready to provide the data, then the process interrupts. The hardware interrupts are handled by switching to the kernel mode. In other words, the kernel acts as the via-media between all the processes and the hardware as depicted in Figure 14.1.

The following are typical system calls in Unix:

Intent of a process The C function call

- Open a file open
- Close a file close
- Perform I/O read/write
- Send a signal kill (actually there are several signals)
- Create a pipe pipe
- Create a socket socket
- Duplicate a process fork
- Overlay a process exec
- Terminate a process exit

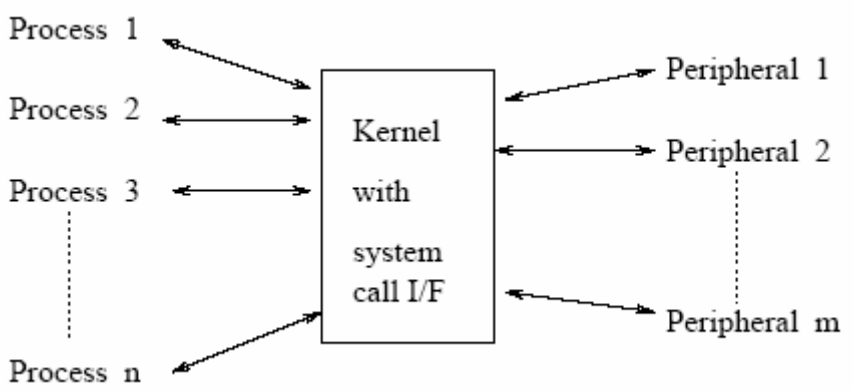

Figure 14.1: The kernel interface.

14.2 System Calls

The kernel lies between the underlying hardware and other high level processes. So basically there are two interfaces: one between the hardware and kernel and the other between the kernel and other high level processes. System calls provide the latter interface.

System call interacts with the kernel employing a syscall vector. In this vector each system call vector has a fixed position. Note that each version of Unix may individually differ in the way they organize the syscall vector. The fork() is a system call. It would be using a format as shown below.

syscall fork-number

Clearly, the fork-number is the position for fork in the syscall vector. All system calls are executed in the kernel mode. Typically, Unix kernels execute the following secure seven steps on a system call:

1. Arguments (if present) for the system call are determined.
2. Arguments (if present) for the system call are pushed in a stack.
3. The state of calling process is saved in a user structure.
4. The process switches to kernel mode.
5. The syscall vector is used as an interface to the kernel routine.
6. The kernel initiates the services routine and a return value is obtained from the kernel service routine.
7. The return value is converted to a c version (usually an integer or a long integer). The value is returned to process which initiated the call. The system also logs the userid of the process that initiated that call.

14.2.1 An Example of a System Call

Let us trace the sequence when a system call to open a file occurs.

- User process executes a syscall open a file.
- User process links to a c runtime library for open and sets up the needed parameters in registers.
- A SW trap is executed now and the operation switches to the kernel mode.
  - The kernel looks up the syscall vector to call "open"
  - The kernel tables are modified to open the file.
  - Return to the point of call with exit status.
- Return to the user process with value and status.
- The user process may resume now with modified status on file or abort on error with exit status.

14.3 Process States in Unix

Unix has the following process state transitions:

- idle ----> runnable -----> running.
- running ----> sleep (usually when a process seeks an event like I/O, it sleeps awaiting event completion).

- running ----> suspended (suspended on a signal).
- running ----> Zombie (process has terminate but has yet to return to its exit code to parent. In unix every process reports its exit status to the parent.)
- sleeping ---> runnable
- suspended---> runnable

Note that it is the sleep operation which gives the user process an illusion of synchronous operation. The Unix notion of suspended on a signal gives a very efficient mechanism for process to respond to awaited signals in inter-process communications.

14.4 Kernel Operations

The Unix kernel is a main memory resident \process". It has an entry in the process table and has its own data area in the primary memory. Kernel, like any other process, can also use devices on the systems and run processes. Kernel differs from a user process in three major ways.

1. The first major key difference between the kernel process and other processes lies in the fact that kernel also maintains the needed data-structures on behalf of Unix. Kernel maintains most of this data-structure in the main memory itself. The OS based paging or segmentation cannot swap these data structures in or out of the main memory.
2. Another way the kernel differs from the user processes is that it can access the scheduler. Kernel also maintains a disk cache, basically a buffer, which is synchronized ever so often (usually every 30 seconds) to maintain disk file consistency. During this period all the other processes except kernel are suspended.
3. Finally, kernel can also issue signals to kill any process (like a process parent can send a signal to child to abort). Also, no other process can abort kernel.

A fundamental data structure in main memory is page table which maps pages in virtual address space to the main memory. Typically, a page table entry may have the following information.

1. The page mapping as a page frame number, i.e. which disk area it mirrors.
2. The date page was created.
3. Page protection bit for read/write protections.

4. Bits to indicate if the page was modified following the last read.
5. The current page address validity (vis-a-vis the disk).

Usually the page table area information is stored in files such as immu.h or vmmac.h.

Processes operating in \user mode" cannot access the page table. At best they can obtain a copy of the page table. They cannot write into page table. This can be done only when they are operating in kernel mode.

User processes use the following areas :

- Code area: Contains the executable code.
- Data area: Contains the static data used by the process.
- Stack area: Usually contains temporary storages needed by the process.
- User area : Stores the housekeeping data.
- Page tables : Used for memory management and accessed by kernel.

The memory is often divided into four quadrants as shown in Figure 14.2. The vertical line shows division between the user and the kernel space. The horizontal line shows the swappable and memory resident division. Some Unix versions hold a higher level data structure in the form of region table entries. A region table stores the following information.

- Pointers to i-nodes of files in the region.
- The type of region (the four kinds of files in Unix).
- Region size.
- Pointers to page tables that store the region.
- Bit indicating if the region is locked.
- The process numbers currently accessing the region.

Generally, the above information is stored in region.h files. The relevant flags that help manage the region are:

1. RT_UNUSED : Region not being used
2. RT_PRIVATE: Only private access permitted, i.e. non-shared region
3. RT_STEXT: Shared text region
4. RT_SHMEM: Shared memory region

The types of regions that can be attached to a process with their definitions are as follows:

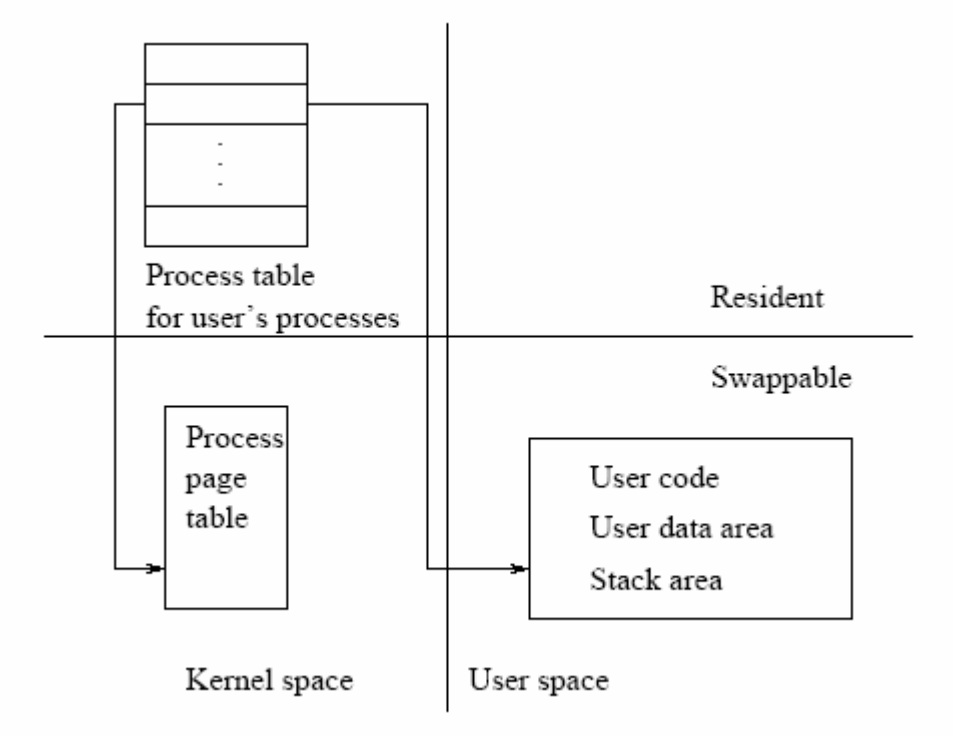

Figure 14.2: User and kernel space.

1. PT_TEXT: Text region
2. PT_DATA: Data region
3. PT_STACK: Stack region
4. PT_SHMEM: Shared memory region
5. PT_DMM: Double mapped memory
6. PT_LIBTEXT: Shared library text region
7. PT_LIBDAT: Shared library data region

In a region-based system, lower-level functions that are needed to be able to use the regions are as follows:

1. *allocreg(): To allocate a region
2. freereg(): To free a region
3. *attachreg(): Attach region to the process
4. *detachreg(): Detach region from the process
5. *dupreg(): Duplicate region in a fork
6. growreg(): Increase the size of region
7. findreg(): Find from virtual address

8. chprot(): Change protection for the region
9. reginit(): Initialise the region table

The functions defined above are available to kernel only. These are useful as the virtual memory space of the user processes needs regions for text, data and stack. The above function calls cannot be made in user mode. (If they were available in user mode they would be system calls. These are not system calls.)

14.5 The Scheduler

Most Unix schedulers follow the rules given below for scheduling:

1. Usually a scheduler reevaluates the process priorities at 1 second interval. The system maintains queues for each priority level.
2. Every tenth of a second the scheduler selects the topmost process in the runnable queue with the highest priority.
3. If a process is runnable at the end of its allocated time, it joins the tail of the queue maintained for its priority level.
4. If a process is put to sleep awaiting an event, then the scheduler allocates the processor to another process.
5. If a process awaiting an event returns from a system call within its allocated time interval but there is a runnable process with a higher priority then the process is interrupted and higher priority, process is allocated the CPU.
6. Periodic clock interrupts occur at the rate of 100 interruptions per second. The clock is updated for a tick and process priority of a running process is decremented after a count of 4 ticks. The priority is calculated as follows:
   priority = (CPU quantum used recently)/(a constant) + (base priority) + (the nice setting).
   Usually the priority diminishes as the CPU quantum rises during the window of time allocated. As a consequence compute intensive processes are penalised and processes with I/O intensive activity enjoy higher priorities.

In brief:

- At every clock tick: Add 1 to clock, recalculate process priority if it is the 4th clock tick for this process.

- At every 1/10th of the second: Select process from the highest priority queue of runnable processes.
- Run till:
  1. End of the allocated time slot or
  2. It sleeps or
  3. It returns from a system call and a higher priority process is ready to run.
- If it is still ready to run: It is in runnable state, so place it at the end of process queue with the same priority level.
- At the end of 1 second: Recalculate the priorities of all processes.

Having seen the Unix Kernels' structure and operation, let us look at Linux.

14.6 Linux Kernel

The Linux environment operates with a framework having four major parts. These sections are: the hardware controllers, the kernel, the OS services and user applications. The kernel as such has a scheduler. For specialised functions, the scheduler makes a call to an appropriate utility. For instance, for context switching there is an assembly program function which the scheduler calls. Since Linux is available as open source we can get the details of the kernel internals function. For instance, the source description of the scheduler is typically located at /usr/src/linux/sched.c. This program regulates the access of processes to the processor. The memory management module source is at /usr/src/linux/mm. This module supports virtual memory operations.

The VFS or virtual file system takes into a variety of file formats. The VFS is a kind of virtual file interface for communication with the devices. The program at /usr/src/linux/net provides the basic communication interface for the net. Finally, the programs at /usr/src/linux/ipc define the range of inter-process communication capabilities. The net and VFS require device drivers usually found at /usr/src/linux/drivers. The architecture x dependent code, essentially, assembly code is found at /usr/src/linux/arch and /usr/src/linux/include. Between them these define and offer all the configuration information which one may need. For instance, one can locate architecture specific files for processors like i386 and others. Lastly, as mentioned earlier, the Linux kernel uses some utilities written in assembly language instructions to do the process switching.

14.6.1 Linux Sources and URLs

Linux, as we stated earlier, is an open source. The following URLs offer a great deal of information about Linux.

1. www.linux.org The most sought after site for Linux. It gives all the user driven and developed information on Linux.
2. sunshine.unc.edu/mdw/linux.html The home page of linux documentation.
3. www.cl.cam.ac.uk/users/iwj10/linux-faq A very good site on frequently asked questions on Linux.
4. www.XFree86.org The X windows in Linux comes from XFree.
5. www.arm.uk.org/rmk/armlinux.html A non-PC architecture Linux home page.
6. www.maclinux-m68k.org An Apple computer, Motorola home page for Linux.
7. The following three URLS are for linux administration, open source gnu c compiler and Linux kernel. linux.dev.admin linux.dev.gcc linux.dev.kernel

Now that we have studied Unix kernel, we shall reinforce the tools track. The next module shall deal with make tool and its use in large projects.