Systems Programming and Operating Systems	**Induprakas Kodukula**
CS414, Summer 1996	**Class#2, 7/9/96**

1 Class #2

As already discussed, the fundamental purpose of the operating system is to provide a certain set of services to application programs. In a multiprogrammed environment, it is also necessry to present an illusion that each process owns the machine completely when it is executing. This section examines the fundamental abstractions that a typical operating system uses for these purposes.

1.1 Process

For the purpose of the discussion of this section, a fundamental abstraction of service will be used, called a **process**. A **process** is the fundamental unit of service in modern operating systems like UNIX. A process encapsulates the interaction between the operating system and the application program.

For the purposes of this discussion, a process is an *executing program* and an associated *address space*. The address space of a process is the set of memory locations that a process may access. These locations may be either *program* locations or *data* locations. The control point of execution of a process tracks the sequence of instructions being executed and is typically represented by a hardware register such as the program counter. In a multi programming environment, when several processes share a machine, it is important to present a *virtual* machine to each process. Typically, each process has its own memory and registers, but must depend on the kernel to able to do I/O.

1.1.1 Process Notation

We'll first introduce static and dynamic notations for processes. Static notation represents how the program is coded up. A dynamic notation represents how processes behave dynamically.

1.1.2 Static Notation

A *static* notation for a process is represented by a program fragment. We'll use the following program constructs: A **sequential** process is specified by a piece of code contained inside the keywords *process* and *end*. This piece of code may contain anything other than a **process** keyword.

```
process .... end
```

1.1.3 Dynamic Notation

The dynamic invocation of a process is performed using the *fork* command. A fork command looks like the following:

```
fork w
```

The meaning of a fork is to go to *label* w and start executing the code from there concurrently with whatever is currently executing. A process can call the

```
quit
```

function to terminate.

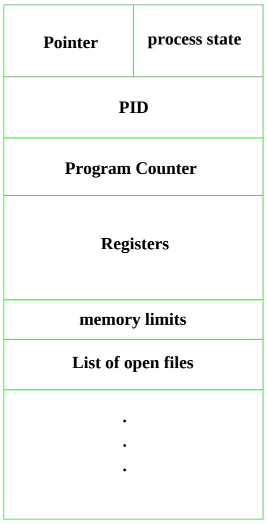

Figure 1: The Process Control block

1.1.4 State, context, segements

Every process has a *state* associated with it. A process is *new* when it has just been created. A process is *running* when its instructions are being executed. A process is *blocked* when it is waiting for some event to happen (typically an I/O request to complete). A process is *ready* when it is not *blocked*, and is waiting for processor assignment. A process is *terminated* when it has finished execution.

Every process has associated with it a *context* often represented as stored as a **process control block**. The fields in the pcb are typically the following:

The state of the process

Program Counter

CPU Registers such as integer registers, floating point registers, program counter, condition code registers.

CPU Scheduling information, such as process priority

Memory management information (to be discussed later)

Accounting information, such as the amount of CPU resources used.

I/O status information, such as the list of I/O devices the process is waiting on or allocated to the process.

The context contains all the information associated with a process that is necessary to restart a process from its current point of execution. For example, the registers, memory contents, any outstanding I/O requests all form a part of the *context*. When the CPU switches from one process from another, the context of the old process needs to be saved and the context of the new process needs to be restored. This process is known as a **context switch**. The contexts are typically saved and restored from secondary storage such as a disk. Context switches are perhaps the biggest performance bottleneck in current OSes.

A running process typically consists of differnt *segments*, which can be usually thought of as contiguous portions of memory. Typically, a running process has *code*, *data* and *stack* segements.

A running process also has an identifier assigned to it, which is unique among all the processes running at any given time. This identifier is typically a number and called the **pid**.

1.1.5 Processes in Unix

An operating system such as Unix uses a data structure similar to the **process control block** shown in figure 1. Unix maintains a data structure called the process table, which has one entry of a **process control block** for every process that is running currently. Every running process has a unique number identifying it, which is called the process identification number, called the **pid**.

Every process running under UNIX has a **context** associated with it. The context consists of information such as

- The program counter of the next instruction
- Permissions of the process, i.e. is it allowed to execute privileged instructions or not?
- Processor status register
- Stack pointer, etc

The **context** of a process under UNIX is sometimes subdivided into **static** and **dynamic** context. **Static** context is the portion of the context of a process that is invariant over the lifetime of the process, such as the **pid** of a process. The **dynamic** context on the other hand represents information that changes over the lifetime of process execution, for example, the stack of the process. It is important to realize that a process is represented by its context.

Unix provides two mechanisms for process creation. The first is the regular **fork** system call. The **fork** system call when executed by a process has the effect of creating a child process with a unique **pid**, and the same context as the calling process. In other words, it creates a duplicate copy of the calling process. The **fork** system call works the following way:

- Allocate a slot in the process table
- Assign a unique ID to the child process
- Make a logical copy of the context of the calling process to the context of the new process

The second mechanism the Unix operating system provides to create processes is called the **exec (filename)** system call. The **exec** system call takes an executable file name as an argument. The **exec** system call has the effect of creating a new process from the executable passed to it as an argument. However, the difference from **fork** is that, rather than create a new process, **exec** overwrites the context of the calling process by the context of the new process derived from the file name passed as an argument. Here is a brief description of how **exec** works:

- Verify the file **filename** has executable permissions
- Read file headers and verify that it is a proper executable module.
- Free storage associated with calling process
- Create context for new process in the same place that was freed up by the old process

1.2 Inter Process Communication

For various reasons, running processes need to communicate. This communication is best achieved using IPC (Interprocess communication mechanisms). A message passing mechanism is usually the best way to provide an IPC facility. Message sent by a process can be either of fixed size or of variable size. Fixed sized messages are easy to implement, but programming with them is hard. Variable sized messages on the other hand are harder to implement, but easier to program. A message passing IPC mechanism must at the very least implement at least a **send** and a **receive** operation to a process.

For two processes to communicate, a **communication link** must exist between the two processes. Such a link has several interesting properties that one must be concerned about:

How are links established?

Can a link be associated with more than two processes?

What is the capacity of a link? Is buffer space a concern?

What is the behaviour of message sizes? Fixed? Variable?

Is a link unidirectional or bidirectional? If a link exists from P to Q, can Q use it to send a message to P?

Inter process communication can be either direct communication or indirect communication. In the direct communication mechanism, each process explicitly refers to the other process by name. For example, if there are two processes P and Q, that wish to communicate with each other, they can do the following:

```
send (P, message)     Send a message to process P
recv (Q, message)     Receive a message from process P
```

In this case, the behaviour of the communication links between P and Q is as follows:

A link is established between every pair of processes wanting to communicate

A link is associated with exactly two processes

Between each pair of processes, there is exactly one link

Link is usually bidirectional.

Indirect mode of communication can be thought of as a **mailbox** mechanism. The idea is that there are a number of mailboxes each with unique identifiers. The following two primitives are defined to use with the mailbox.

```
send (M, message)     Send a message to a mailbox
recv (M, message)     Recv a message from a mailbox
```

In this case, for a pair of processors, P and Q, the following is true regarding the behaviour of communication links

A link is established between two processes only when they share a mailbox.

A link may be associated with more than two processes

Between each pair of processes, there can be a large number of links.

Links can be either unidirectional or bidirectional.

A problem with the **indirect communication** mechanism is **contention**. Contention happens when one process sends a message to a mailbox, after which two processes simultaneously do a recv operation. In this case, the operating system must arbitrate among the contending processes.

1.3 Process Scheduling

Processes need to be scheduled. Scheduling is a policy decision. This means the following: there is no *optimal* scheduling strategy in general. This is because of several reasons. One is that it is not always possible to predict the execution time of a process before it has started. Another is that the optimal strategy may depend on the mix of I/O and CPU work that a process needs to perform. This section describes some commonly used scheduling strategies. The following criteria are typically used to evaluate scheduling strategies.

Scheduling time: The time that elapses between the time the job arrives at the machine and the time it gets to run

waiting time: The amount of time a process spends waiting, typically for I/O requests to complete.

turn-around time: The time that elapses between the time a job is submitted and the time it is completed.

1.3.1 FCFS

The simplest scheduling policy is one in which the jobs are allowed to run as they come. Each job runs to completion before the next one is run. The disdavantage of FCFS is that average scheduling time for the processes can be quite long.

1.3.2 Shortest-job-first

At any time, the *job pool* (which is all the jobs that have not yet been run) is examined to find which is the shortest job. That job is run first. This strategy gives the shortest average scheduling time for a given set of input jobs. The disadvantage of this approach is that it is hard to estimate how long a job may run.

1.3.3 Priority Scheduling, no preemption

In this scheme, every job is assigned a prioirity before it starts executing. When a job needs to be run, the *job pool* is examined for the job with the highest priority, which is then run to completion.

1.3.4 Priority Schedulin, w/ preemption

This is similar to the previous strategy, except that if a new job of higher priority than the currently running job is received, the current job is context switched and the new job is started.

1.3.5 Round robin

In this scheme, a *time-slice* is defined as an interval of some fixed length. The job pool is implemented as a FIFO queue. The process at the head of the queue is run until the process completes execution or an interval equal to the *time-slice* elapses. If the process runs to completion, the next process from the queue is started. If the process has not run to completion, it is inserted at the tail of the queue and the next process from the queue is started up.

1.3.6 Multilevel queues

In this scheme, there are several *job queues* in the system with different time-slices for each job queue. When a job arrives in the system, it is placed in one of the job queues depending on some characteristic of the job. Typically, the jobs in the shortest time slice job queue are run first. If that queue is empty, jobs in the queue with the next smallest time slice is run next and so on. Each queue is usually scheduled in round-robin fashion.

1.3.7 Multilevel w/ Feedback queues

In this scheme, there are also several job queues with different time slices (each scheduled in round-robin fashion). However, a new job always goes to the queue with the smallest time slice. However, after the job has spent a certain amount of time in this queue, it is *migrated* to the job queue with the next shortest time slice. This scheme basically ensures that very long running jobs don't end up increasing the response of the system to shorter running jobs. Again, as in the previous case, jobs in a queue with a particular time slice are run only when there are no jobs in any queue with a shorter time slice. UNIX uses a 3 level feedback scheme for its scheduling.