Chapter 4
Process Management

Revised: Jan. 30, 2011

4.1 INTRODUCTION

The central purpose of any computer system is to perform useful work for the system's users. This work is carried out through the execution of application programs. The activities performed by these programs are represented in most operating systems by a model known as a **process**.

The precise meaning of the process concept varies with each operating system. Informally, you could think of a process as a unit of activity or a program in execution. It is important to note that a process is an active concept. A program stored in a file or in memory is not a process. It becomes a process only when it is "in execution." This does not imply that a process is actually running at all times; in fact, partially executed processes often must wait while the computer performs other actions. However, each process is "working on" a program, even if its activity is temporarily stopped.

Another name often used for process is **task**. This term is widely used in some environments, especially in connection with real-time executives or large IBM systems. Adding to the confusion, some systems use both terms, with subtle distinctions between them. For consistency in this text we will usually stick with the term process, except when discussing real-time systems specifically. When considering a particular OS, you must refer to the system manuals to be certain how these terms will be used.

In most cases a process may be characterized more precisely as a **sequential process**, to emphasize the fact that each process represents a single sequence of actions that take place one at a time. However, multiprogramming techniques can be used to share a computer's resources among many processes at a time. Using these techniques, a set of processes can all be in progress at the same time, taking turns at execution by the CPU. The execution of these processes is said to be **interleaved**. Interleaved processes provide the illusion of concurrent execution, and form a system of **concurrent sequential processes**.

An operating system that supports multiple processes executing in an interleaved fashion is referred to as a **multiprogrammed** system or a **multitasking** system. It is important to avoid the temptation to use the term multiprocessing; this is reserved for systems with more than one physical processor.

True simultaneous execution may be possible in a multiprocessor or distributed system, which includes more than one processor. This type of system introduces some subtle difficulties not found with a single CPU. Although single-processor environments are still the norm, multiple processor systems are becoming more common. In this discussion we will generally assume that there is only one processor, and that concurrent execution means interleaved execution. However, we will use the term processor rather than CPU, to allow for the possibility of multiple processors when this causes no great complications.

To do useful work, a process must execute a **program**. However, it is not necessary that each program correspond to exactly one process. In some systems (e.g., VMS), a single process may execute a series of different programs. In other environments (such as UNIX), it is common for a program to consist of several distinct processes, each representing a separate sequence of activities that logically continue at the same time.

The process model is a powerful concept that can effectively represent most of the activities that present-day operating systems support. Most operating systems represent application programs by some type of process. In many cases, major parts of the operating system are represented by processes as well. When it is necessary to distinguish these cases, we will use the terms **application process** or **system process**, respectively. A number of systems also support a simplified type of process, which may have limited capabilities but can be more efficiently managed than a standard process. Such processes are known as **lightweight processes** or **threads**. In a system supporting threads, an increasingly common model, each process may be divided into multiple threads, which share the same data.

To become an effective mechanism for managing a set of activities, a process must be formally defined and represented by an operating system. The basic ideas are handled by different OSs in many different ways. Often the choice of techniques has a major impact on the ways in which processes can be used to support various applications.

Processes may be classified into one of three distinct categories, which strongly affect their characteristics and requirements, and the characteristics of the systems that support them. Process categories are introduced in the next section.

Although a process is not a data object, the OS must represent each process by a data structure in order to keep track of its status and manage its activities. This data structure is usually called a **process control block (PCB)**. Some systems maintain a large set of descriptive and status information in the PCB of each process, allowing more intelligent operations on processes, but perhaps requiring considerable space and time to maintain the process control blocks.

Throughout its operation, a process makes use of a set of **resources**. The exact resources needed will vary as time goes on. The two resources that are always needed by each process are a processor and some memory. Other resources that may sometimes be required include files, I/O devices, and shared information in main memory. By their nature, many of these resources may be used by only one process at a time. Because of this, processes may be required to wait when needed resources are not available.

-

Processes and resources are complex subjects, raising issues that must be studied with care. One type of problem arises in the management of the resources that every process needs, namely processors and memory. Because of the intense competition for these resources, scheduling becomes a significant problem. A second set of problems is caused by the interaction of processes that execute concurrently. This interaction may be intentional, as when processes exchange messages, but often it is an unpleasant side effect of the need to share limited resources.

Because of the need to wait when necessary resources are in use, each process alternates between running and waiting. A waiting process may be waiting for a variety of reasons, and these reasons determine how the process should be treated. To represent these distinctions, a process is considered to move through a series of distinct **states**.

A running process makes use of the registers and other physical elements of a CPU. When that process is required to wait, information in these registers must be saved in some way, so it can be restored intact when the process receives another turn at execution. The information to be saved is called the **context** of the process. This information may also be referred to as the **processor state**. The act of saving the context of one process and restoring that of another is called **context switching**. If this switch takes too much time, it may limit the frequency with which processes may reasonably be switched.

Some older operating systems that made use of the process concept supported only a small, permanent set of processes with little explicit interaction. Today many systems allow processes to be freely created and destroyed, and to establish various types of relationships. These operations are considered at the end of this chapter.

4.2 PROCESS CATEGORIES

In most operating systems, each application process may be considered to be in one of the following categories:

- **batch processes**, which perform a job submitted by a user as a whole. Usually a batch process performs a relatively long computation. Batch processes do not interact with users during their execution.

- **interactive processes**, which respond to requests made by a user at an interactive terminal. In general, they are short computations that may interact with the user during execution.

- **real-time processes**, which monitor and control external events and must meet rigid timing constraints. They usually interact with one or more I/O devices.

The roles of each process type are contrasted in Figure 4-1. Each type has a distinct set of characteristics and requirements that must be considered for effective process management.

-

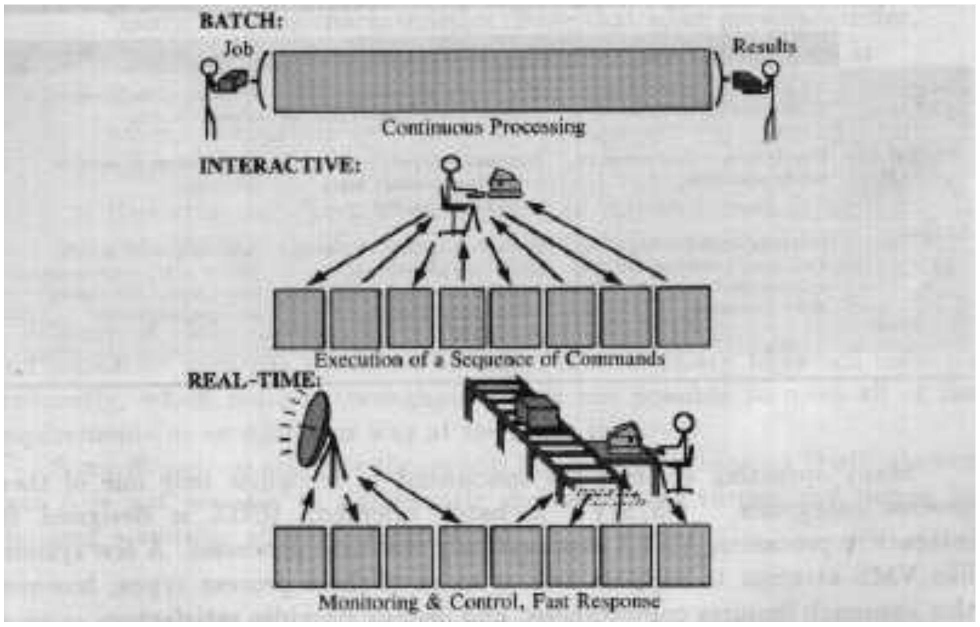
BATCH:
Job
Results
Continuous Processing
INTERACTIVE:
Execution of a Sequence of Commands
REAL-TIME:
Monitoring & Control, Fast Response

Figure 4-1: Process Categories

Batch processes are often submitted with good estimates of their anticipated running time, and their need for memory and other resources. In addition, they may include instructions that explicitly determine the relative importance and scheduling requirements of different processes. The user measures the effectiveness of batch process management by the speed with which final results are returned. Since most batch computations have substantial time and resource requirements, equally sizable processing delays may be accepted.

Interactive processes are assumed to perform work on behalf of a user who remains present at an interactive terminal. Many of these processes have very short running times and resource needs. They may display information for the user as they progress, or require terminal input at various points in their processing. Interactive processes are rarely provided with information on resource use or scheduling requirements. The user initiating such a process expects rapid and consistent response throughout its execution, not only when the process is complete.

Real-time processes are designed to interact with external events and objects, such as tracking air traffic or controlling the flow of chemicals in a laboratory experiment. The resource requirements of real-time processes are generally known. They are further characterized by absolute timing requirements, which must be met regardless of other activities in the total system.

The process categories supported have such an impact on process and resource management that they are considered to define the type of the system itself. Many operating systems are specialized to recognize only one of these process categories. IBM's z/OS is primarily batch oriented; Linux is designed for interactive processes; VxWorks supports only real-time processes. Some systems like Windows attempt to support two or more of these process types; however, this approach requires compromises, and usually provides satisfactory support in, at most, one process category. When more than one type of process is present, a

-

single type is always dominant. A system that contains any real-time processes is a real-time system. If there are interactive processes, but no real-time processes, the system is an interactive system. Finally, systems that recognize only batch processes are considered to be batch systems.

4.3 PROCESS REPRESENTATION

The principal data structure used by an OS to represent a process is the **process control block (PCB)**. The PCB contains all the information the OS elects to maintain about a process to describe its attributes and keep track of its status. The size and content of a PCB varies considerably. A real-time executive like VxWorks squeezes all necessary information into a few dozen bytes, while more complex OSs like Linux or Windows 7 require hundreds of bytes per process.

Despite this wide variation, a fairly standard set of information may be found in most process control blocks. This information includes:

- **process name**, by which the process may be identified by users or by other processes. Sometimes there is a distinct external name for users (usually a character string) and an internal name (usually an integer).

- **process state**, a code representing the current state for this process, or a link to a queue that represents that state.

- **process context (CPU state),** a storage area for register contents and other information that must be saved when the process is not running.

- **memory use**, information about the memory allocated to this process and the current location of its program code and data.

- **resource use**, information about other resources currently in use by this process, and any restrictions or quotas on future resource use.

- **process priority**, a value or set of values to help determine the relative priority to be given to this process when scheduling CPU use and allocating resources. This information may be more complex for real-time processes.

- **relationships**, information about relationships that may exist between this process and other processes, such as those it has created.

- **accounting information**, records of time and other resources that have been used by this process, and who owns the process, to be used for billing or analysis.

In a multiple processor environment, the PCB may also contain information about the set of processors that may execute this particular process. A variety of specialized information representing other aspects of the execution of a process may be found in specific PCBs. Most of the information contained in a typical PCB will be described later in more detail.

-

An operating system may be organized with a fixed array of PCBs in a suitable data area, or may allocate PCBs as needed, collecting them on a linked list. A possible declaration for a PCB data type might take the following form:

```
TYPE PCB:
/* pointer to next PCB */
next_PCB: pointer to PCB;

/* process identifiers */
external_name: char[20];
internal_name: integer;

/* scheduling priority */
priority: integer;

/* state and associated queue */
state: integer;
state_queue: pointer to PCB;

/* data for CPU state */
context: integer[32];

/* relatives */
parent: pointer to PCB;
children: pointer to PCB queue;

/* memory use */
program_start: address;
program_size: integer;
data_start: address;
data_size: integer;

/* resource control block */
resources: pointer to resource CB;

/* up to 10 file control blocks */
files: FCB[10];

/* owner's id code */
owner: integer;

/* time spent executing and performing I/O */
CPU_time: integer;
IO_time: integer;

END PCB;
```

-

4.4 PROCESSES AND RESOURCES

To make further progress at any moment in carrying out its assigned activities, a process must be able to use certain resources that, in general, must be shared with other processes. One obviously necessary resource is a processor to execute the instructions of the process. Other important resource categories include main memory space, secondary storage space, and I/O devices. All of these are **physical resources**, that is, permanent components of the computer. Usually they must be controlled by the operating system.

Although files are not physical resources, they are usually treated in this category because of their possibly long lifetimes and ability to be shared. Thus, files are also controlled by the OS.

Another important category of resources is formed by information structures stored within files or main memory. These are **logical resources**, which often have a much shorter lifetime than physical resources. Many logical resources are not under the direct control of the operating system. They are considered resources only when a set of processes agrees to share them.

By their nature, physical resources are **reusable**. They are not destroyed when used, so they may be used repeatedly. In most cases, however, such resources may only be used by one process at a time. Two processes cannot use the same memory at once for separate purposes, nor output data at random to the same printer. A reusable resource that can be used by only one process at a time is called **serially reusable**. Other reusable resources may be shared by more than one process under some conditions.

Some logical resources are **consumable** rather than reusable. They are created by one process and destroyed after use by another. An example is the blocks of data in a message.

To maintain the integrity of the resources it manages, the OS must assign them to a process as needed. This procedure is called **resource allocation**. The OS must keep records showing the current status of each resource (either "available" or "allocated to process P"), and must have a strategy for deciding when to allocate each resource to a particular process, and when to deallocate the resource as well. Finally, for effective control, the OS must be able to prevent use of the resource by processes to which it has not been properly allocated.

There is a similar need for an allocation strategy for resources not controlled by the operating system, but this strategy must be established by voluntary cooperation among the processes involved. An important goal of such a strategy is to ensure **mutual exclusion,** preventing processes from improperly accessing resources while they are in use by other processes. This is an important problem with a number of subtle difficulties. We will not discuss this issue further here.

The best way to solve these allocation problems depends on the nature of the resource. As already suggested, two resource types demand special treatment: processors and memory. These resources have two special characteristics:

1. They are accessed directly, not through system calls.

2. They are always needed by every process.

-

The first of these observations is true because each process uses a processor and accesses memory during execution of each machine instruction. The OS cannot monitor and control each use; it must use other means to prevent misuse of these resources. The tools required for effective control include privileged instructions, memory protection mechanisms, and an interrupting timer.

Because of the second observation, the OS must assume that any process that does not currently have its processor and memory resources allocated is waiting for their use. Thus, there will often be many processes waiting, and a strategy is needed for deciding which waiting process should receive each resource as it becomes available. This leads to the problem of scheduling. Many issues must be considered in developing a suitable scheduling strategy for processors and memory; we will explore these issues in the next chapter.

Other system-controlled resources, including files and I/O devices, do not have the special characteristics listed above. Actual use of these resources, such as reading a file or printing a line on a printer, is accomplished by system calls. The operating system may require that these be preceded by requests to allocate the resources. The OS will reject a request for use if the resource has not been allocated. It will defer a request for allocation, requiring the process to wait, if the resource is not currently available.

Because resources in this category are needed only occasionally, processes are expected to signal their needs by explicitly requesting allocation. There will be few processes waiting for each specific resource, and the scheduling problem is greatly simplified. However, other problems arise in the efficient management of these resource requests, especially the potentially serious problem of deadlock, in which resources that some processes are waiting for cannot be made available because they are held by other processes, which in turn are waiting for other unavailable resources.

The OS must maintain a record of processes waiting for each distinct resource it manages. This record usually takes the form of a queue, whose entries can be added or removed in some systematic order. The queue is an important data structure for process management. In most cases, the most natural form for such a queue is a linked list; but other representations, such as tables or arrays, are also used. The most common order of service for such a queue is first in, first out. In an operating system queue, however, many other orderings are possible. This is especially true for process scheduling queues.

A process is entered into a queue by linking its PCB to the list in some manner. For some queues, especially those used for CPU scheduling, a link field may be reserved in the PCB itself. Each PCB on the queue contains a pointer to the next in this dedicated field. This is an efficient technique for queues on which most processes are frequently entered, but it requires dedicating a special location in every PCB. For other queues that are less frequently used, separate entries may be constructed with pointers to each PCB in the queue. These techniques are shown in Figure 4-2.

-

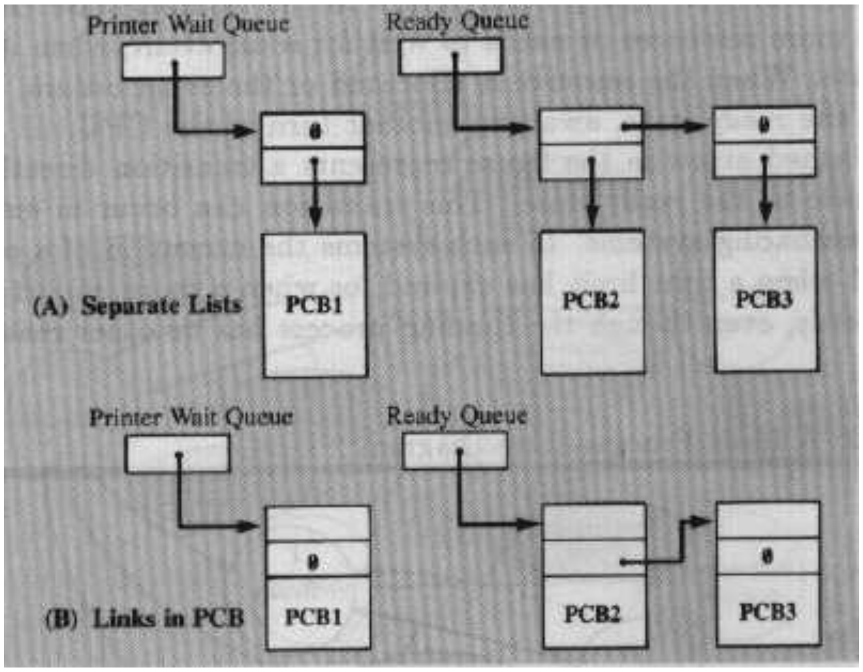

Fig 4-2: Queue Structures for Process Management

4.5 PROCESS STATES

The concept of **process state** is widely used to keep track of the current status of each process: At each moment, a process is considered to be in one of a small set of states which represent its current activity or resource needs.

An active process is either running or waiting; if not running, it is waiting for some resources (including at least a processor) required for further progress. In addition, it may be waiting for the occurrence of one or more specific **events**, such as a signal from another process, or the completion of an I/O operation. The simplest possible model for process states would include two states: **running** and **waiting**.

However, because processors are such critical resources, no process is given a processor while it is waiting for something else. Thus, processes waiting for processors are treated in two distinct categories, depending on whether they are also waiting for something else. A process that is waiting only for a processor is in a **ready** state, while one that is waiting also for other resources, or for the occurrence of specific events, is considered to be in a **blocked** state.

A useful representation for understanding the process states of a particular system is the **state diagram**. A state diagram illustrating the three principal states for a process is shown in Figure 4-3. The circles in this diagram represent possible states, and the arrows represent permissible transitions from state to state. In the usual sequence, a process begins in the ready state. When its turn comes, a processor is assigned to it. The process is now in the running state.

It runs until it requires more resources or needs to wait for some event, when it enters the blocked state. When the resource is allocated or the event occurs, the process returns to the ready state, awaiting another turn at the processor.

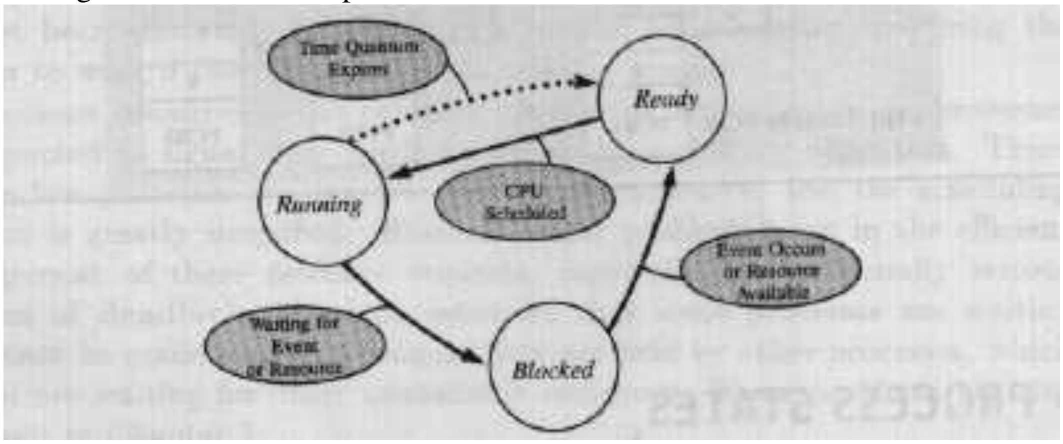
Time Quantum Expires
Ready
Running
CPU Scheduled
Event Occurs or Resource Available
Waiting for Event or Resource
Blocked

Fig 4-3: A Basic Process State Diagram

A dashed arrow in the figure shows a transition directly from the running state to the ready state. This transition is common in timesharing systems. In such systems the execution of a process may be stopped when a time limit has expired, or when a more important process becomes ready, even though the running process has no other reason to wait. A second dashed arrow indicates a transition from ready to blocked. This may occur when resources are taken away from a ready process and given to a more important process. This operation, known as **preemption,** occurs primarily in real-time systems**.**

The names we have used for the states of a process are typical ones; the exact names used by each OS vary widely. In addition, the blocked state may be partitioned in various ways, depending on what the process is waiting for. A useful distinction can be made between waiting for events that are expected to occur very soon (e.g., completion of a disk transfer), or for events that may not occur for a long time, if ever (e.g., a signal from another process or terminal user). A process that is waiting for long-term events may be placed in a **suspended** state. Often, the programs and data for a suspended process are removed from memory and swapped to disk. When the awaited event occurs, the process is no longer blocked by that event but may have to wait further to be restored to memory. This leads to a more complex state diagram such as the one shown in Figure 4-4. Also shown in this figure is an **initial** state for processes that have been created but have not yet begun to execute, and a **terminal** state for processes that have completed execution. The purpose of the terminal state is to allow final status information to be obtained from a process before it is completely destroyed.

If the computer system has one processor, only one process can be running at any instant in time. To achieve an illusion of concurrency, processes execute in an interleaved manner using multiprogramming. Even if there is more than one processor, interleaving will be necessary unless there is a processor for every process. Multiprogramming becomes efficient because use of a processor by one process can take place while other processes are waiting for I/O operations to complete, or for other events to occur. Each process then goes through many cycles of alternate running and waiting.

-

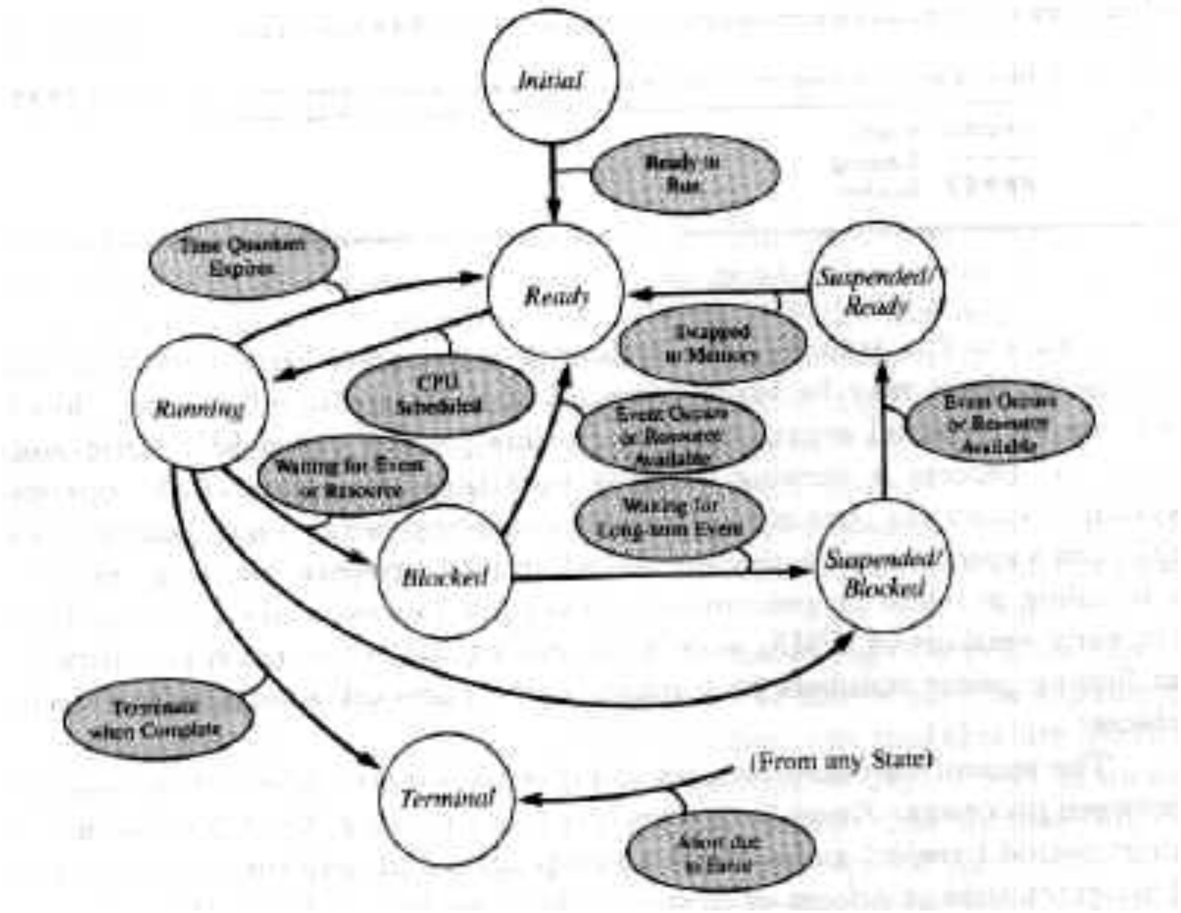

Fig 4-4: State Diagram with Additional States

Figure 4-5 illustrates a series of successive phases that might be experienced by a typical set of processes running in a multiprogramming operating system. At any moment, only one process per processor can be in the running state. Other processes vary between the ready and blocked states. However, because a number of processes are available, some process is running at all times.

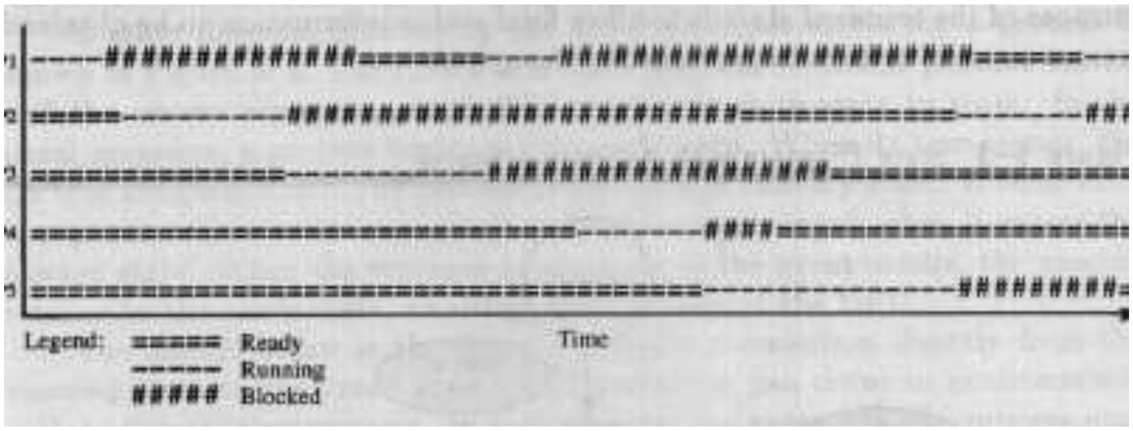

Fig 4-5: Process States vs. Time (Simplified)

Figure 4-5 is actually unrealistic in two respects. First, even with many processes, there may be times when all of them are simultaneously blocked waiting for external events, resulting in time periods called **idle time** during which no process is running. During such times a portion of the operating system is executing, but not necessarily doing any useful work. Sometimes the OS runs a special system process, called an **idle process**, which simplifies the scheduling problem by guaranteeing that some process is always ready to run. On early versions of UNIX, such a process was put to work on problems such as finding prime numbers or computing the constant **e** to a million decimal places.

The second unrealistic aspect of Figure 4-5 is its neglect of the time used between processes. Every time a process stops running, the OS must run for a short period to select and activate a new process and perform a context switch. This extra time is a form of overhead that can be significant if processes are changed too frequently or if the scheduling strategy is too complicated. A more realistic example is presented in Figure 4-6.

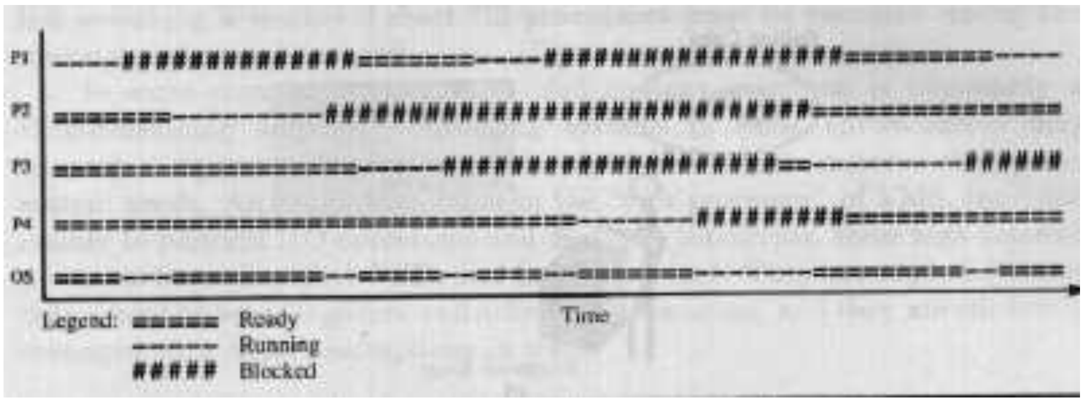

Fig 4-6: Process States vs. Time (Realistic)

4.6 CONTEXT SWITCHING

Since processes running in a multiprogramming operating system must periodically give up control of the processor, we must have a way to preserve the information stored in registers, status flags, and the like while the processor is in use by the OS or by other processes. In switching from one process to another, it is necessary to save all information that the first process maintained in the processor registers in some convenient way, and to replace it with the corresponding information for the second process. As we noted, this information is known as the **context** of the process, and the activity is called **context switching**. It may occur dozens of times a second in large timesharing systems.

The importance of keeping track of the exact status of a process can be shown by an analogy from human experience. Our minds may be capable of keeping track of a couple of simultaneous activities, but mistakes are likely if the number of activities becomes large, as illustrated in Figure 4-7. If we are using a recipe in baking a cake, we have a series of steps that

-

we must follow. If we are on step three---add the first of the two cups of flour required---and the telephone rings, our activity will temporarily change from baking to speaking on the telephone (in other words, a "human mind context switch" takes place). Upon completing the telephone conversation, hanging up and returning to the activity of baking, we must remember exactly where we were in the recipe. If we remember adding the flour but forget we had only added the first cup of two, the result will not be the cake that the recipe is supposed to produce. Similarly, a failure to save all of a process's registers or status upon a context switch will cause the process to produce incorrect results.

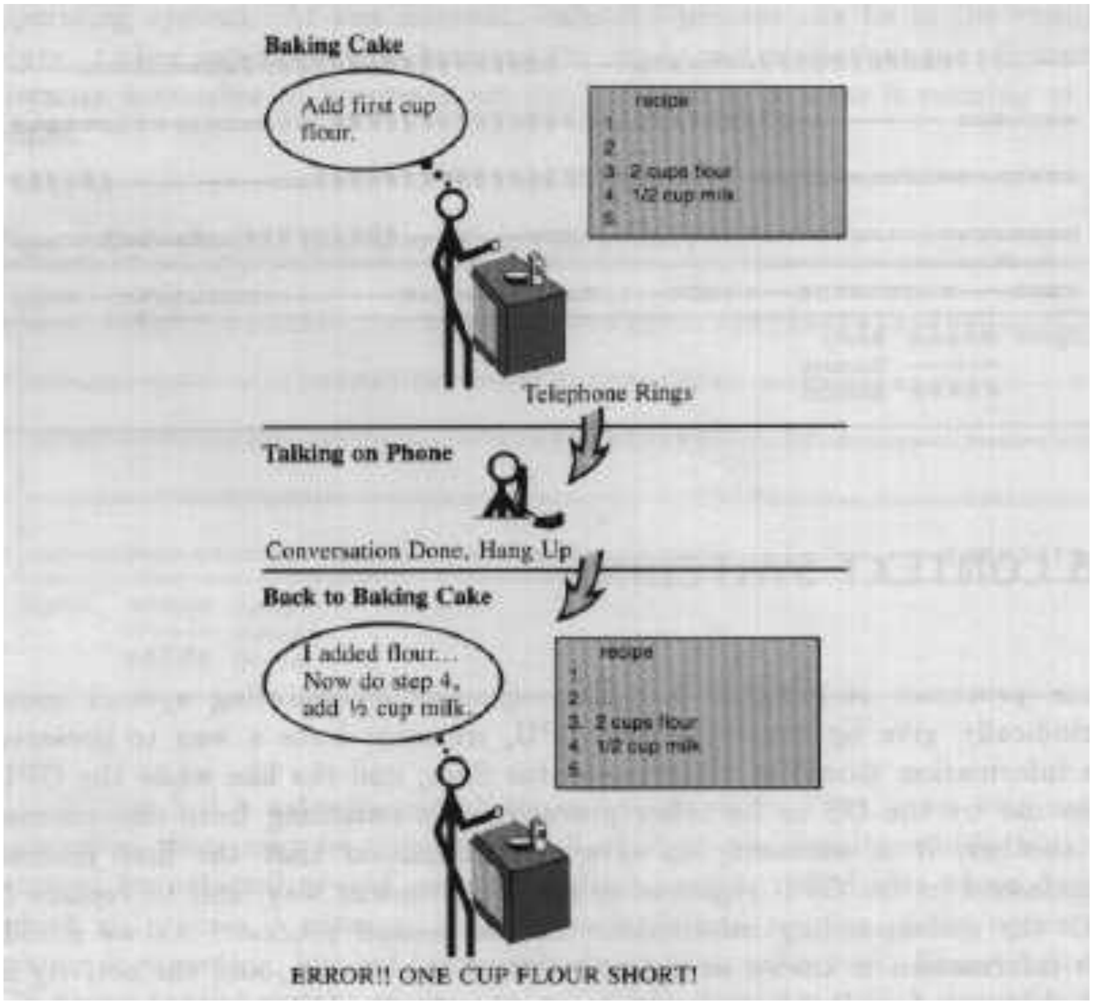

Fig 4-7: Failure to Remember the Status of an Activity (Process)

In the simplest systems, processes give up control of the processor voluntarily at convenient points in their program and can prepare by saving the necessary information. In general, though, loss of control can occur asynchronously as a result of interrupts at any time; hence the loss and regaining of control must be transparent to a process. When the process resumes, everything in the execution environment (register contents, status words, address spaces, stacks, current location in the program, and so on) must appear exactly as it was when the process ceased execution.

-

It is important that a context switch be performed as efficiently as possible. To a large extent, the cost of context switching is determined by the processor architecture. Generally, when there is more information to save, the cost is greater. Many architectures have special instructions providing rapid saving and restoring of the required information to reduce overhead. Some machines provide multiple register sets, so that in some cases information can be left in one set while the next process uses a different one. It is especially helpful to supply the operating system with a private set of registers, so that less switching is needed if short OS procedures must be executed during the operation of a single process.

Because of the high cost of context switching, **lightweight processes** are sometimes introduced to serve limited system needs. An example is the "fork processes" of VMS. Intended mainly to perform I/O operations and deal with interrupts, these high-priority processes had simplified PCBs and limited context. They were expected to use only a limited set of registers and other CPU resources, and they are efficiently managed with these assumptions in mind.

By far the most popular type of simplified process today is a **thread.** Threads hold no independent resources except for a program counter and basic processor registers. Instead, a set of threads shares a single program, memory space, and resource set. Threads are useful for various types of programs that may contain explicit concurrency. They are generally viewed as subdivisions of an ordinary process.

One question that then arises is whether threads should be scheduled directly by the OS (the **kernel threads** model) or only within the schedule of their respective processes (the **user threads** model). Both approaches have benefits and drawbacks. For example, kernel threads ensure that every thread that is ready to run will have a chance, even if other threads in the same process are blocked. User threads ensure that each thread receives a fair share of its parent process' priority. The best choice depends on the nature of the application.

Linux and Windows are examples of common operating systems providing support for threads. Linux allows an application to setup threads following either a kernel threads model or a user threads model, or to run without any threads at all. Windows 7 assumes that every process has at least one thread, and may have additional threads besides. Threads are scheduled using a kernel threads model, but may themselves be divided into a type of "subthread" that Windows refers to as a **fiber**. Fibers are scheduled using a user thread model.

4.7 PROCESS CREATION AND CONTROL

In early multiprogrammed operating systems, processes were permanent entities. The number of application processes was fixed at one per user. The number of system processes, if any, was fixed as well. CDC SCOPE, for example, provided seven application processes plus one system process, while T.H.E. supported 15 processes: five for users, and the rest primarily for controlling I/O devices. PCBs were permanently allocated in memory, and processes were never created or destroyed. These systems provided **static process management**. Handling processes in such environments was relatively simple, but the usefulness of the process concept was limited.

-

By contrast, an OS that supports **dynamic process management** allows processes to be created and destroyed during the operation of the system. In the simplest case, the number of processes changes only as the number of concurrent jobs or users; there is always one process per job. In newer OSs, it is possible for one process to explicitly create another by use of system calls. Thus a user may employ several processes, all running simultaneously, to perform a total job. Processes created by an application may be considered to have equal stature with the main process, or they may be viewed as subprocesses with restricted capabilities.

In some operating systems, processes and subprocesses are relatively expensive to create and maintain. Each one may require a large PCB with extensive links to other data structures throughout the system. This was the case, for example, with MVT and VMS, which expected processes to only occasionally have subprocesses. Other environments, notably Tenex, MULTICS and UNIX, were designed to encourage the use of multiple processes by each user. These systems treat all processes equally, and require efficient algorithms for process creation and management.

The ability of processes to create new ones gives rise to a process hierarchy, as shown in Figure 4-8. A new process is called a **child** of its creator, which is considered the **parent**. Children of the same parent are called **siblings**. Other related terms, such as **grandparent**, **ancestor**, and **descendant,** may also be used in the obvious way.

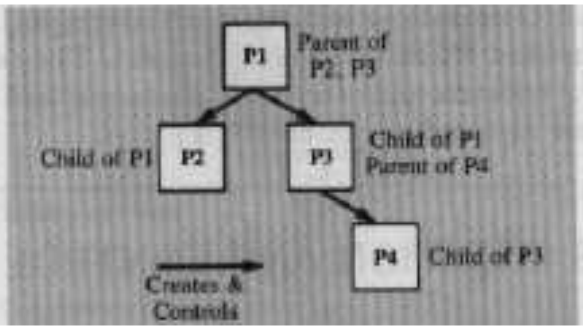

Fig 4-8: A Hierarchy of Processes

In most cases this hierarchy directly reflects the lines of authority that allow certain processes to control others. A parent may control its children (and indirectly, all its descendants) by examining and changing their properties, starting and stopping them, or even destroying them. Usually, a process can exercise little control over other processes that are not among its descendants.

4.8 PROCESS CONTROL OPERATIONS

Some operating systems that do not provide for process creation or interprocess communication provide few if any system calls for explicit operations on processes. When processes are allowed to create and control other processes and to deliberately interact in various ways, a number of useful operations may be provided. In this section, we discuss operations used by processes to directly control one another.

Process Creation

If an OS supports dynamic process creation, new processes may be created upon request by an existing process. Typically the request takes the form of a **create_process** system call. The calling process may be running a system program, such as the command interface, or an application program. A hierarchical relationship is established among the processes, as we discussed in the previous section. The newly created process is a child of its creator. A parent process generally has special authority and responsibilities for the children it has created.

Creation of a process that is ready to run requires the following distinct activities:

- Create or allocate a process control block;
- Allocate initial memory space for program and data;
- Identify and (usually) load the program to be run;
- Assign initial attributes and resource limits to the process;
- Allocate initial resources to the process, if any;
- Establish the starting state for the process, and setup or complete the PCB.

The method for allocating PCBs varies among different operating systems. Some systems maintain a pool of permanent PCBs. Free PCBs are either linked together in a free PCB list or identified by a status flag in the PCB itself. This method is shown in Figure 4-9.

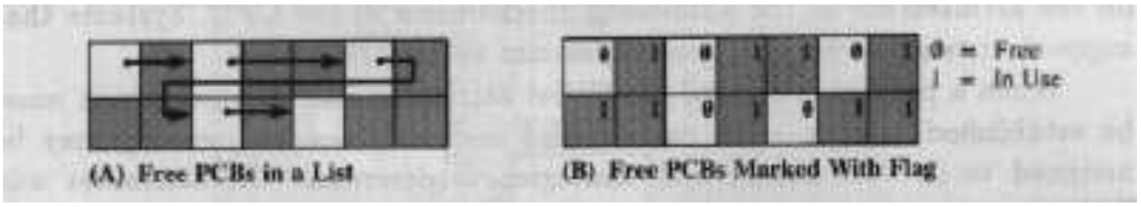

Fig 4-9: PCB Allocation from a Permanent Pool

Other systems use dynamic memory allocation to get memory for a PCB. A specific storage area may be reserved for allocation of PCBs and other system data structures. This is the strategy used by versions of OS/360, such as MVT.

UNIX employs a mixed approach, as shown in Figure 4-10. The PCB is divided into two parts. Critical portions are collected in a permanent data structure called the **process table**. The process table then includes a pointer to the remainder of the PCB, called the **user area**, which is allocated only when needed.

-

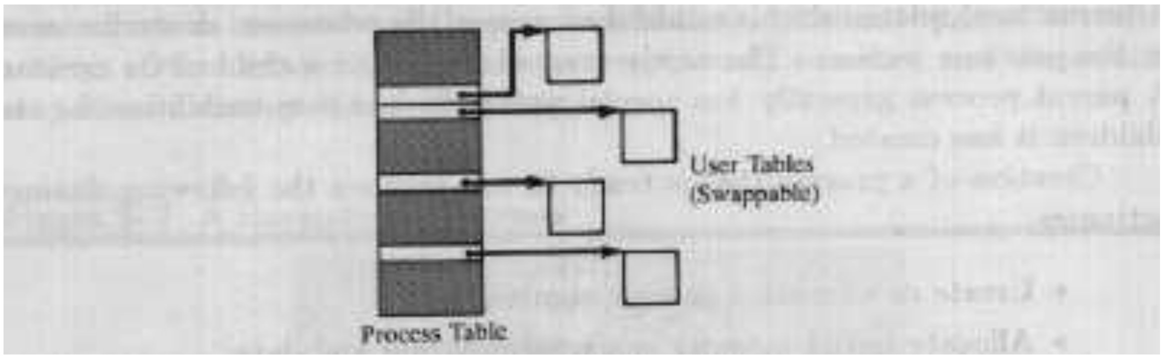

Fig 4-10: The UNIX Process Table

In addition to the allocation of memory for a PCB, memory must be allocated for the (first) program to be loaded and executed. Often the process will not be created unless sufficient memory is available.

The loading of the program is usually done by a relocating loader, a resident component of the operating system. The loader reads programs stored in "executable" form in files, and copies their instructions and initial data into the allocated memory. The loading process may be complicated because the addresses at which the program information will be stored are not known until it is actually loaded. Branch addresses and data references contained in the program may need to be adjusted to reflect the actual locations. As discussed in a later chapter, this relocation process may be carried out by a combination of hardware and software techniques depending on the architecture of the addressing mechanisms of the processor. Systems that support virtual memory, in general, do not require relocation.

When a process is created, its initial attributes and resource limits must be established, and in some cases initial resources besides memory may be assigned to it. Several strategies may be used to determine the attributes and limits to assign to a newly created process. Some examples include:

- Derive attributes from the known properties of the batch job being run (MVT)
- Provide default attributes based on the characteristics of an interactive user (VMS, Windows 7)
- Allow the creator process to specify, within limits, what the child's attributes should be (VMS, Windows 7)
- Assign the child process attributes derived from or identical to those of its parent (UNIX, Linux)

In most cases, a new process is not automatically allocated any resources except memory. Additional resources are assigned only on request. An exception is found in UNIX, in which a child process inherits the allocation of resources, such as open files, from its parent. The newly created process shares access to these resources with its creator.

-

Although it seems most natural to provide a newly-created process with a program and place it immediately in a ready state, these steps are separated in some systems. UNIX provides two distinct operations, **fork** and **exec**, as shown in Figure 4-11. The fork operation creates a new process and makes it ready, but instead of loading a new program, the child process initially runs the same program (from the same location) as its parent. (All programs under UNIX are reentrant and may be shared by multiple processes.) The data areas used by the two processes are independent. The fork procedure returns a result parameter that is different in each process and may be used to distinguish parent from child. Each fork call is normally followed by a test so that each process can take the appropriate actions. Usually the child process then performs an exec call. This operation loads a new program for the calling process, completely replacing the old one. The process then begins executing at the start of the new program.

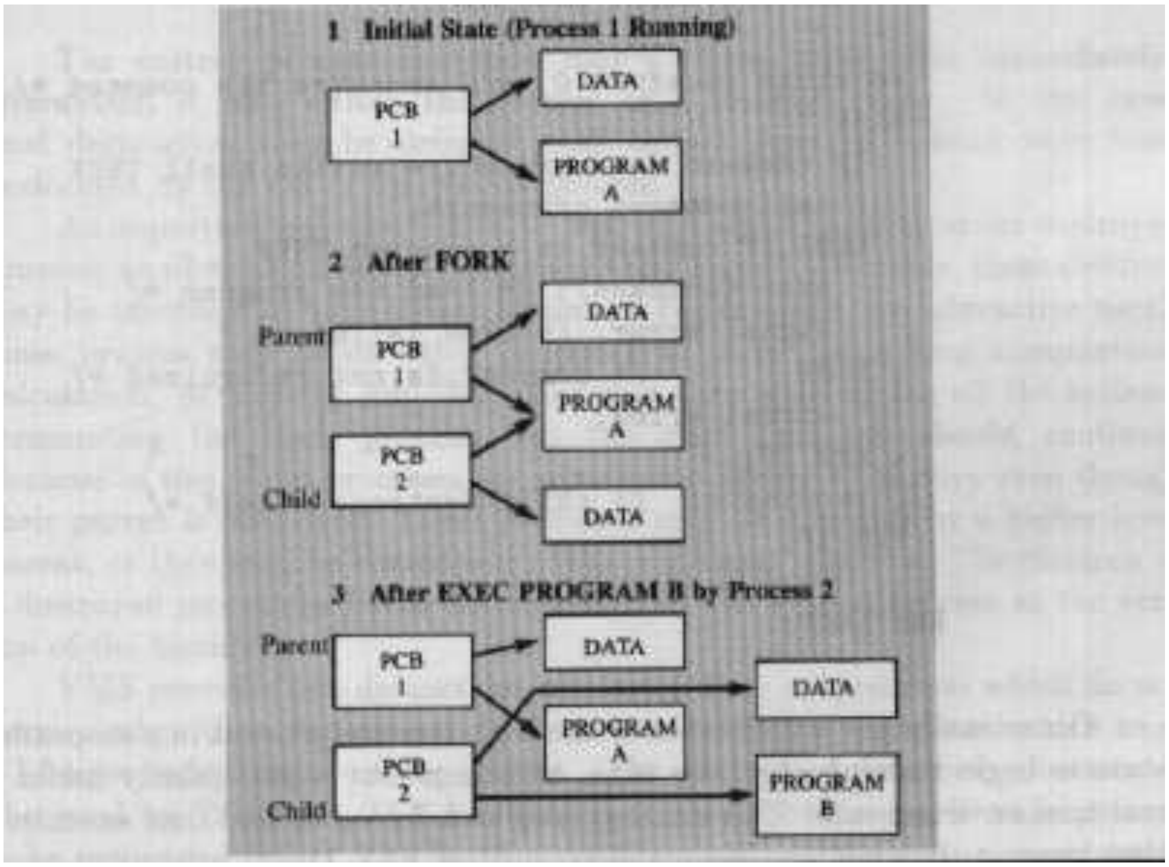

Fig 4-11: Process Creation in UNIX

The UNIX strategy does take some getting used to, but it provides flexibility in special cases. A child process may be created to execute a short procedure in the parent program, without the overhead of loading (or finding space for) a separate program. In this case, the forked process becomes a type of thread. This strategy is followed by standard UNIX shells, which use forking alone to process simple commands whose program code is part of the shell itself. The following algorithm is a typical example:

-

```
          LOOP
            get a command from terminal;

      /* perform fork, return status value
               to both processes */
            status := fork();

      /* signal error if fork failed */
            IF status < 0 THEN    /* fork failed */
              signal error

      /* parent receives child's id and waits */
            ELSE IF status > 0 THEN
              wait for child process

      /* child receives 0, and executes the command */
            ELSE
              IF command is a procedure within shell THEN
                call command procedure;
              ELSE IF command is a program THEN
                exec (command); /* load new program */
                signal error; /* exec failed */
              ELSE        /* command is not recognized */
                signal error;
              END IF;
              terminate;  /* child destroys itself */
            END IF;

          END LOOP;
```

Occasionally you will want to create a process and place it in a suspended state to begin running at a later time, an action that is particularly useful in real-time environments. This strategy is often supported by real-time executives that allow dynamic process creation.

Process Destruction

The removal or destruction of a process can occur for several reasons, such as those in the following list:

- The process itself has requested normal termination
- A nonrecoverable error has occurred in the process execution
- An operator command has terminated the process
- A parent process has terminated either normally or abnormally

-

- An authorized process has requested termination

Destruction may be invoked by a **destroy_process** system call by the process itself or another process, or it may be an automatic consequence of certain events. Destruction of a process involves the following steps:

- Remove PCB from its current state queue, and other queues to which it may be linked
- Destroy or reassign child processes
- Free allocated memory and other resources
- Free the PCB

The destroy_process operation may perform these steps immediately; alternately, it may switch the process to a terminal state. In this case, final destruction may be deferred until certain clean-up actions have been performed by the OS or the parent process.

An important question is how to deal with active children of the destroyed process; an obvious answer is to destroy them as well. However, these children may be intended to have longer lifetimes. For example, an interactive user's main process may create other processes to carry out a long compilation, calculation, or printing job. The user may then want to log off the system, terminating the main process, but the child processes should continue. Because of this, child processes are sometimes allowed to survive even though their parent is destroyed. These processes may be inherited by a higher-level parent, or they may be considered orphans without a parent. The children of a destroyed process in UNIX are inherited by the system process at the very top of the hierarchy.

VMS provided two distinct process categories: subprocesses that do not survive when their parents are destroyed, and detached processes, which have a lifetime independent of any parent. The concept of detached processes was also used by TOPS-20. This mechanism further allows a user's main process to be suspended rather than destroyed when an interactive session terminates. If the session terminates abnormally, such as due to a communication failure, suspension occurs by default. When the same user begins a new session, it is possible to continue with the same process.

Reading and Changing Attributes

Operations may be provided to allow a process to obtain information from its own PCB or those of certain other processes. Most of the information recorded in the PCB could be useful to a process in some cases, and can be obtained simply.

In many systems ranging from small real-time executives to OS/MVT, the PCB is maintained within the address space of the process it controls. In this case, the process can read the PCB (and possibly modify it) directly. If the PCB is not directly accessible, a **read_PCB**

-

system call may be provided, which reads the entire PCB into a buffer in the address space of the calling process. These methods allow a process to examine all attributes in the PCB, but the process must know how to interpret them.

Alternately, there may be distinct calls to obtain various categories of information, such as current state, priority, resources allocated, or CPU time used. This allows programs to be less dependent on the precise format used for PCBs.

Processes may also have a limited ability to change their attributes or those of other processes, such as their children. Information that might be subject to change includes process names, resource allocations and limits, and (within limits) scheduling priority.

Suspension and Wakeup Operations

The OS may allow a process to put itself, or another process it controls, into a suspended state via a suspend or sleep operation. The request for suspension may specify an event that the process is waiting for, which may be caused by action of another process, by an interrupt, or by arrival of a specific time. The OS must then watch for this event and associate the process with it in some way (as by a special queue). When the event occurs, the OS will resume or "awaken" the process by restoring it to a ready state. Alternately, the suspend call may not specify any event. In this case the process relies on an explicit wakeup operation by another process that specifies that the original process should resume its activity.

Examples

Process management operations available in common versions of UNIX include *fork* to create a child process, *kill* to destroy a process (or, more generally, to send a "signal" to a process), and *exit* by which a process terminates itself. Other system calls allow processes to find out their own internal name, get and set priorities and resource limits, and get statistics about their own behavior or that of their children.

Processes may be delayed until a specific time. A more general suspend-wakeup mechanism is provided by allowing processes to send signals to other processes and to wait until signals are received. Generally, each process may control and access only itself and its descendants.

Mach supports both processes (which it calls tasks) and threads. A task is a unit of resource allocation, and a thread (which exists within a task) is a unit of activity. Typical process management operations are supported on both tasks and threads; these include create, destroy, suspend, resume, get information, and a few others. Operations on tasks implicitly affect all threads within the task.

VMS provided system services to allow processes to create and delete other processes; suspend themselves or others, possibly scheduling a wakeup at a predetermined time; resume a suspended process, obtain information from the PCB, and set process names, priorities, and privileges. The authority of each process to control others in various ways is determined by a complex system of privileges assigned to each process.

-

The real-time executive VRTX provides system calls to create, delete, suspend and resume processes, to obtain PCB information, and to change priorities. In VRTX, like many real-time executives, any process can control any other one.

Linux supports traditional UNIX process operations but adds support for optional threads. Windows XP provides a traditional process creation model but directly manages threads and fibers rather than processes.

FOR FURTHER READING

Although there are few books or articles devoted to basic process management, the subject is well covered in a variety of general OS texts. Good overviews are given by Deitel [1984] and Milenkovic [1987]. More theoretical treatments may be found in Coffman and Denning [1973] or Brinch Hansen [1973]. Kaisler [1983] provides a detailed treatment of data structures and algorithms for process management. Goscinski [1991] discusses process management for distributed operating systems, including a number of detailed case studies. Detailed implementations for pedagogical operating systems are presented by Tanenbaum [1987] and Milenkovic [1987]. A set of standard system calls for process management is described in the IEEE MOSI standard [IEEE 1990], while aspects of the UNIX style of process management are codified in the POSIX standard [ISO 1991]. A comprehensive set of process management operations is also defined by the CTRON specifications [TRON 1992].

A number of books and articles cover, in detail, process management in specific real operating systems. Important examples include Bach [1986] for UNIX, Kenah and Bate [1984] for VMS, and Organick [1972] for MULTICS. Merusi [1992] discusses process management in VMS, UNIX, and OS/2, and Goscinski [1991] presents case studies for Mach and six other distributed OS's. Process management in several older systems, such as SCOPE and T.H.E. is described by McKeag et al. [1976]. Case studies of MVS and VM/370, among others, are provided by Deitel [1984]. Silberschatz et al [2003] provides a good description of process management in Linux, Windows XP, and several others.

REVIEW QUESTIONS

1. Explain in your own words the meaning of the process concept.

2. Give examples of some resources that are (a) serially reusable, (b) reusable and shareable, and (c) consumable.

3. Which two resource types are handled in a fundamentally different way from all others? What is the major difference in handling, and why is it necessary?

4. How does a typical queue structure used within an operating system differ from the usual definition of a queue?

-

5. In the simplified process state diagram of Figure 4-3, explain why there is no transition from the blocked state to the running state.

6. Explain the distinctions among a thread, a lightweight process, and an ordinary process.

7. Identify five explicit operations on processes that an OS might provide in response to system calls.

8. Describe the steps performed by a typical OS during dynamic creation of a new process.

9. Contrast the UNIX fork and exec mechanism with the more common create_process operation.

10. Explain why it may not be desirable to destroy all the descendants of a process when the parent process is destroyed.

ASSIGNMENTS

1. Briefly describe some difficulties a process manager might face if there is more than one processor (i.e., more than one process may be in the running state at the same time). Assume that all processors share the same memory.

2. The definition of system types in Section 4.2 indicates that interactive processes are "more important" than batch processes, and that real-time processes are most important of all. Explain the reasons for this ranking.

3. Explain a possible advantage of each of the two queue structures illustrated in Figure 4-2.

4. In many operating systems, the state of a process is represented implicitly by linking that process on an appropriate queue such as the ready queue. In such cases is it also useful to represent the state explicitly by a code? Explain a possible advantage and disadvantage of this "redundant" state representation.

5. Define a data structure PCB_Q for a collection of PCBs using a high-level language such as C, Pascal, Ada, or another language of your choice. What language features are important for defining a data structure of this type? Why?

6. Draw a process state diagram for an OS in which blocked states waiting for I/O requests are to be kept separate by device. For example, waiting for terminal input, printer output, or disk transfer are all to be considered separate states. What effect would this more detailed division of blocked states have on PCB structure and system mechanisms for state representation? What would be the effect of providing separate states for suspended processes?

-

7. Using a time chart similar to Figure 4-6, contrast the way in which a set of five programs might execute on a uniprogrammed OS and a multiprogrammed one. Are there cases in which the programs might complete their work as soon or sooner without multiprogramming? Why or why not?

8. Draw a diagram representing a process hierarchy with at least ten processes. Your diagram should include three "generations", that is, some child processes should themselves have children, and each process which has any descendants should have at least two. Identify all of the direct relationships that exist for each process in your diagram.

9. Give two examples in which it may be useful for a single user to have two or more processes, each executing a different program at the same time.

10. Give a specific example illustrating the use of two or more processes in a single program.

11. Give an example illustrating the usefulness of multiple threads sharing the same memory and other resources.

12. As shown in Figure 4-10, UNIX separates its PCB into two parts. Critical data which must exist in memory as long as the process has any existence whatsoever is stored in the process table; other PCB data is stored in the swappable user area. Consider the PCB information classes outlined in Section 4.3. If you are designing a PCB using the UNIX approach, what information items should be placed in the process table, and why?

13. Mach separates all processes into tasks, which control resources, and threads, which consist solely of a processor state (including program counter). This organization is used uniformly for tasks with a single thread and for multiple threads. Discuss some advantages and disadvantages of the Mach organization.

14. List all the information included in the context (CPU state) that must be saved for each process for three different computer architectures. Determine what instructions are provided by each architecture to save or switch contexts. Estimate the average time required for a context switch using a typical implementation of each architecture.

15. Determine the process categories that are supported by three different multiprogrammed operating systems of your choice, other than the ones discussed in Section 4.2.

16. List all the information actually contained in a typical PCB for a large operating system, such as Windows 7. Classify this information according to the categories of Section 4.3. Are any additional categories needed?

-

PROJECT

This section describes a programming project to develop a representation for processes using process control blocks (PCBs), and to implement basic operations on the PCBs. You should read the general description in this text in conjunction with the more specific descriptions given in your project manual. The interface to be constructed is an important component of the multiprogramming executive (MPX) project. This project has three parts.

Part 1

Design data structures to represent a collection of PCBs. The PCBs may be statically or dynamically allocated. A method must be provided to systematically access all the PCBs when necessary. In addition, provision must be made for linking the PCB into a doubly-linked queue representing its current state. This may be accomplished using a pair of pointers in the PCB itself, or using separate list elements, as discussed in Section 4.3.

Additional information required in each PCB includes:

- **Process name**: character string name for the process.
- **Process type**: a code identifying the process as either a system process or an application process.
- **Priority**: an integer representing the process priority. The minimum range should be from -126 to +126 for application processes, and from -128 to +127 for system processes.
- **State**: a code representing the process state. The minimum set of states to be represented includes running, ready, and blocked.
- **Suspended**: a flag indicating whether or not the process is currently suspended.
- **Save area**: a buffer to hold the process context. This must be large enough to contain a copy of all processor registers and flags which must be saved for the type of computer you are using.
- **Stack area**: a region large enough to hold the stack information for the process during execution. A pointer to the current "top" of the stack is also needed. Some architectures do not require a stack area.
- **Load address**: the address of the beginning of the memory region reserved for the program instructions and static data for this process.
- **Execution address**: the address within the program at which execution should begin.
- **Memory size**: an integer giving the size of the memory area

-

The number of PCBs in the collection should be easy to change. The minimum number of PCBs to be supported is 10.

Part 2

Implement the following procedures to manipulate the PCB collection you have defined:

1. `alloc_pcb`: This procedure should locate an available PCB by finding an existing free PCB or allocating a new one if possible. Normally it should return a pointer to the allocated PCB. A null pointer is returned if allocation is not possible.

2. `free_pcb`: This procedure should free a currently allocated PCB. A pointer to the PCB is supplied as an input parameter.The procedure should return a code indicating whether the operation was successful. If it was not successful, the code should further indicate the type of error that occurred.

3. `setup_pcb`: This procedure should initialize a newly-allocated PCB. Input parameters include a pointer to the PCB, process name, process type, and initial priority. Further parameters may be included if necessary. The procedure should return a code indicating whether the operation was successful.

4. `find_pcb`: This procedure is intended to locate the PCB for a process specified by its process name. The process name is the input parameter. The procedure should return a pointer to the appropriate PCB, or a null pointer if the search failed.

5. `suspend_proc`: The purpose of this procedure is to place a process in a suspended state by setting its suspended flag to true. The only required input parameter is a pointer to the PCB. The procedure should return a code indicating whether the operation was successful

6. `resume_proc`: In the same manner, this procedure places a process in a non-suspended state by setting its suspended flag to false. The only required input parameter is a pointer to the PCB. The procedure should return a code indicating whether the operation was successful.

7. `set_priority`: This procedure should allow the priority of a process to be changed. Input parameters are a pointer to the PCB and a priority value. The procedure must ensure that the given priority is within the acceptable range. The procedure should return a code indicating whether the operation was successful.

Part 3

Add commands to the MPX command handler (COMHAN) to create and manipulate PCBs and to display the contents of PCBs. The commands to be added should perform the following functions:

1. Allocate and initialize a PCB for a new process. The process name, type, and initial priority are given as command arguments.

2. Set the priority of a process. The process name is given as a command argument.

3. Suspend a specified process.

4. Resume a specified process.

5. Display information about a process specified by name. The display should include at least the following information:
   - PCB address
   - process name
   - process priority
   - process type
   - process state
   - suspended or not suspended
   - next instruction to be executed
   - program load address and size
   - context save area address

6. Display information about all processes in a particular category. The possible categories should include:
   - all processes
   - all application processes
   - all system processes
   - all processes in the ready state
   - all suspended processes
   - all non-suspended processes

Your display may list only limited information for each process, such as the process name, or it may provide more extended information as in the previous command. Remember to ensure that the complete display fits on a single screen, or that appropriate paging control is provided.

-