

Process Description and Control

1 Process Description

Operating systems are considered as a manager of the underneath various hardware resources, however this management is not the extreme goal, but a way in which the processes may access the resources reasonably. Thus we can think of the operating system as an entity that manages the use of system resources by processes, which may be illustrated in Figure 1, where a solid arrow from a process to a resource entity indicates the former occupies the latter, while a dotted one means a process needs the corresponding resource but does not occupy it anyway. Figure 1 actually shows a snapshot of the resource allocation.

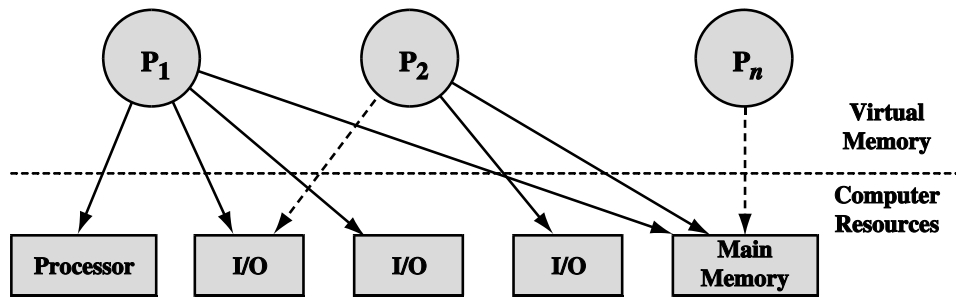


Figure 1: Processes and resources

An operating system, as a service provider, naturally also imposes control on processes that consume the services. That is both processes and resources are under the operating system's control. To enable this control, some facilities must be provided, which is *control tables* we are covering.

Control structures

The operating system constructs and maintains tables of information about each entity that it is managing. Figure 2 illustrates that four different types of tables are maintained by the operating system:

- **Memory tables:** are used to keep track of both main memory and secondary memory. Part of main memory is reserved for use by the operating system;

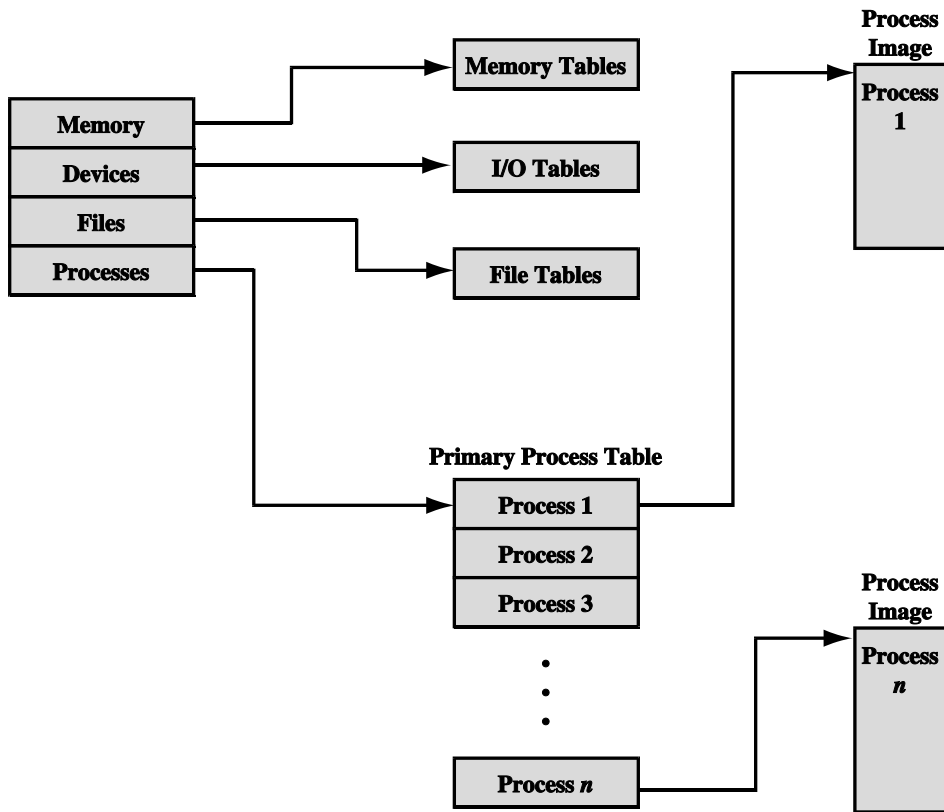


Figure 2: Operating system control tables

the remainder is available for use by processes. Processes may also reside on secondary memory due to the utilization of swapping mechanism. The memory tables generally include the following information:

- the allocation of main memory to processes
 - the allocation of secondary memory to processes
 - protection attributes of blocks of main or virtual memory
- **I/O tables:** are used by the operating system to manage I/O devices. They should record:
 - the availability of each particular device
 - the status of I/O operations relating to each device and the location in main memory being used as the source or destination of the I/O transfer.
 - **File tables:** provides information about
 - the existence of files
 - their location on secondary memory

- their current status and attributes
- **Process tables:** contains what the operating system must know to manage and control processes, including:

- **Process location:**

First a program statically consists of a set of instructions that manipulate data, thus the operating system needs to allocate space for its *code* and *data*. In addition, the dynamic execution of a program requires a *stack* that is used to keep track of procedure calls and parameter passing between procedures. Finally, each process has associated with it a number of attributes that are used by the operating system for process control. Typically the operating system needs to maintain a structure called **process control block (PCB)** containing these attributes. The collection of the above code, data, stack, and attributes is referred to as **process image**.

The location of a process image depends on how the memory management is implemented. Most modern operating systems use a memory-management scheme in which a process image consists of a set of blocks that need not be stored contiguously. The blocks may be variable length (usually called *segments*), or fixed length (called *pages*), or a combination. This scheme allows the operating system to bring in only a portion of any particular process. Therefore process tables must show the location of each segment and/or page of each process image. Figure 2 depicts a primary process table with one entry for each process.

- **Process attributes:**

Process attributes are stored in PCB. Different systems organize this information in different ways, so here we examine only *what* type of information should be included as attributes, instead of *how*. The typical elements of a PCB are:

- * Process identification

All operating systems need to assign a unique identifier to each process in the systems so as to refer to them conveniently. The identifier may be numeric, and for example simply the index of the corresponding entry in the primary process table; otherwise, a mapping must be available to allow the operating system to locate the right entry based on the identifier.

Almost every process is created by another process, its parent process, on behalf of a user. Their identifiers should also be present in the PCB of the child process.

* Process state information

It consists of the contents of processor registers. When a process is interrupted, the context of the processor must be saved so that it can be restored when the process resumes execution. Typically the context includes user-visible registers, control and status registers, and stack pointers.

* Process control information

This may include scheduling and state information, data structuring, process privileges, and resource ownerships. For example in Figure 3, the various queues that we discussed in the preceding class could be implemented as linked lists of PCBs with the help of the data structuring fields. We will talk about more details of control information in succeeding chapters.

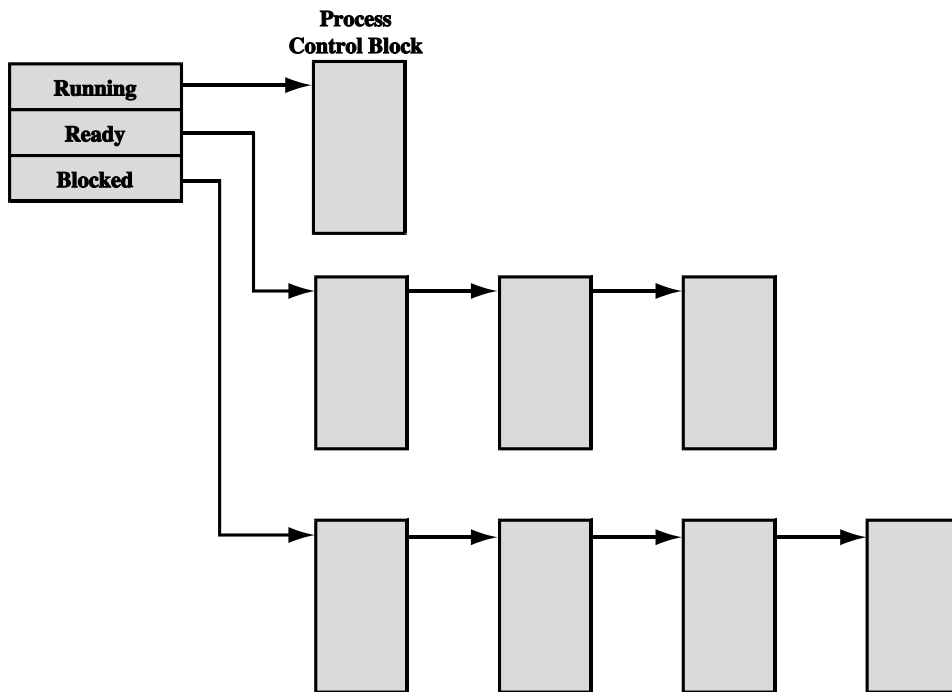


Figure 3: Process list structures

– The role of the PCB

The PCB is the most important structure in an operating system in the sense that each PCB contains all of the information about a process and the set of PCBs defines the state of the operating system.

Its unique position in the operating system thus brings up an important design issue. Within the operating system, a number of modules will need access to PCB. If the direct access is allowed, then a bug in some module

may damage PCBs, which could cause the inability of the operating system to manage the processes. In addition, a design change in the structure or semantics of the PCB could affect a number of modules in the system. A solution to it is setting up an interface for visiting PCBs, which regulates all access and thus separates PCBs from the other parts of the operating system. The strategy is similar to the object-orientation method, where local variables are invisible and cannot be visited directly, and public methods make up the access interface.

2 Process Control

2.1 Models of execution

An operating system are in nature software, consisting of a set of instructions, however it plays a unique role in the computer system since the operating system can do some privileged operations, such as accessing PCBs, while user programs cannot. To meet this requirement, most processors support at least two modes of execution: the less-privileged one or *user mode*, and the more-privileged one or *kernel mode* (also called *system mode*). The operating system runs in the kernel mode and thus may use certain instructions that can only be executed in this mode. In this mode, the software has complete control of the processor registers and memory. User programs run in the user mode, and may not read or alter a control register, or access certain regions of memory.

With multiple modes, there are two questions to be answered:

- How does the processor know in which mode it is to be executed? Typically there is a bit in PSW that indicates the mode of execution.
- How is the mode changed? This is done by executing an instruction that changes the mode, for example the CHM (**CH**ange **M**ode) instruction on the VAX¹ machines. When the user makes a system service call or when an interrupt transfers control to a system routine, the routine executes CHM to enter the kernel mode

¹short for Virtual Address eXtension, the most successful minicomputer design in industry history, possibly exceeding its immediate ancestor, the PDP-11. Between its release in 1978 and its eclipse by killer micros after about 1986, the VAX was probably the hacker's favorite machine, especially after the 1982 release of 4.2BSD Unix. Especially noted for its large, assembly code-programmer-friendly instruction set - an asset that became a liability after the RISC revolution.

and executes it again to enter the user mode before returning control to the user process.

2.2 Process creation

Before we have roughly discussed what happens when a process is created, now it is clearer what is involved in the period after we know the control structures for processes. First, a unique identifier is assigned to the new process and a new entry is added to the primary process table. Second, the space for all elements of a process image is allocated, and then the PCB is initialized. The values for some fields of the PCB structure are already known, e.g. the ID of the parent process, program counter, and system stack pointers, while others fields will use default values, e.g. the process state initialized to Ready or Ready/Suspend, or simply be filled with zero. Finally the PCB will be linked into some data structures and other structures may be created for billing and performance assessment purposes.

2.3 Process switching

When to switching processes

It is mentioned before that multiprogramming systems perform process switching when the currently running process has to wait for the completion of I/O operation and thus keep the processor busy for higher efficiency. Upon the arrival of interrupt signal, the operating system will move all the blocked processes waiting for the signal to the Ready state, and then decide whether to resume execution of the process currently in the Running state or to preempt that process for a higher-priority Ready process. This relates to process scheduling, what we will talk about next week.

In addition, in a time-sharing environment, clock interrupt enables the operating system to determine whether the currently running process has used up its time slice. If so, this process must be switched to a Ready state and another process dispatched.

The above interrupts have the same characteristics. That is they are all due to some sort of event that is external to and independent of the currently running process. There also exist a special kind of interrupts, usually called **trap**, that relate to an error or exception condition generated within the currently running process, such as an illegal file access attempt. With a trap, if the error is fatal, the operating system will move the currently running process into the Exit state and dispatch another process; if not, the system may attempt some recovery procedure or simply notify the user.

The last possibility of process switching comes from **supervisor calls**, invocations of system routines. For example, an I/O request made by a user process will cause a transfer to a system routine and the process is to be placed into the Blocked state.

2.4 Change of process state

When a process switching occurs, the following steps will be taken:

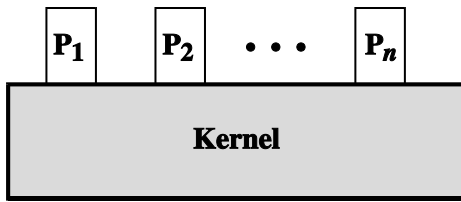
1. Save the context of the processor, including PC and other registers.
2. Update the related fields in the PCB, e.g. the state of the process.
3. Move the PCB to the appropriate queue.
4. Select another process for execution.
5. Update the PCB of the process selected.
6. Load the values of PC and other registers from the PCB, and thus restore the context of the processor to the one when this selected process was last switched out.

Although interrupts usually cause process switching, they do not necessarily do so. In some operating systems, the occurrence of an interrupt may simply involve a mode switching. That is after the interrupt handler has executed the currently running process will resume execution. In this case, all that is necessary is to save the processor state information when the interrupt occurs and restore that information when control is returned to the program that was in progress. The advantage of the mode switch is little overhead involved.

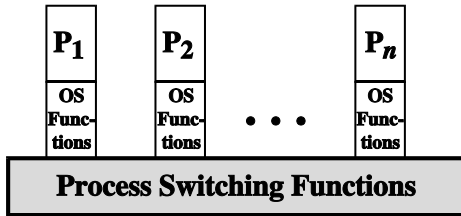
2.5 Execution of the operating system

We know programs run in a computer system as processes, but what about the operating system itself? Does it also take the form of process? Actually with different design approaches of an operating system, the answers also vary. Figure 4 shows three popular options.

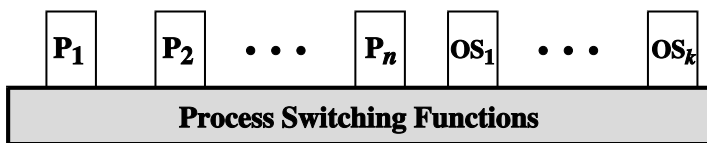
Nonprocess kernel



(a) Separate kernel



(b) OS functions execute within user processes



(c) OS functions execute as separate processes

Figure 4: Relationship between operating system and user processes

A common approach on many older operating systems is to execute the kernel of the operating system outside of any process, as illustrated in Figure 4 (a). With this approach, the concept of process is considered to apply only to user programs. The operating system code is executed as a separate entity the operates in privileged mode.

Execution within user processes

An alternative is to execute virtually all operating system software in the context of a user process. That is that the operating system is primarily a collection of routines that the user calls to perform various functions, executed within the environment of the user's process. This is shown in Figure 4 (b). The advantage of this approach is that when an interrupt, trap, or supervisor call occurs, only a mode switching happens. Upon completion of its work, depending on system design, the operating system may or may not continue to execute the current process; in the latter case, a process

switching occurs.

Process-based operation system

Another alternative, illustrated in Figure 4 (c), is to implement the operating system as a collection of system processes. Although the kernel still executes in a kernel mode, major kernel functions are organized as separate processes. This approach has several advantages:

- It encourages the use of a modular operating system with clean interfaces between the modules.
- Some noncritical operating system functions are conveniently implemented as separate processes.
- The form of processes makes it possible to distribute different parts of the operating system in a multiprocessor environment, improving performance.