Lecture Note

CS431: Introduction to Operating Systems

Introduction to Process

V Kumar
School of Computing and Engineering
University of Missouri-Kansas City

Process

Main function of an operating system

a. Accept a job
b. Provide it its desired resources
c. Execute it and store the results for the user.

User perception: Every user thinks that he/she has the entire machine, even though a number of users are serviced by the O/S simultaneously.

Formally: An operating system creates one **Virtual Machine** for one user. To manage a virtual machine the O/S must keep track of

a. Memory uses.
b. Data/Files used by jobs
c. Status of every job (failed, running etc.).
d. Association of CPU with a job.

Program: Sequence of instructions representing relevant operations for manipulating data to get the desired result. In other words it is the coding of the solution of a problem in a programming language. It does not contain any command to the O/S.

Job: A sequence of commands to O/S for executing the program and the program code. The sequence of commands tells the OS how to treat the program, what to do with the result (print or save etc.) and how to terminate the program. Conventionally, the terms jobs and process have been used to indicate the same thing; however, technically they are different. A job usually is a background unit of processing. However, most of the time we will follow the convention.

Behavior of the CPU: CPU is shared among a number of jobs. An instruction of a program tells the CPU the operation to be performed. Before the execution begins, all necessary information, such as, program size, object code location, files to be used, priority etc., about the program is saved in a special place (RAM or disk). Let us call this a package and when this package is ready, the program can begin execution as soon as the CPU is available. During execution the information stored inside the package may change and all such changes are recorded. Once the package is ready the status of the program is identified as "**being in execution**" but not necessarily being executed at a particular moment. When a job enters this state, i.e., when all information about the program and its intention is know to the system to put it in "**being in execution**" status then the program does not remain a program but termed as a **process**.

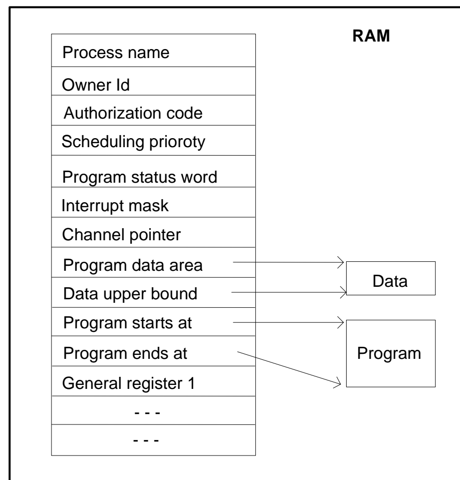
We, therefore, realize that inside a system it is meaningless to talk in terms of programs, since do not exist there, i.e., every activity is a process. From now on, we will use process to indicate a job or a program. Sometime we will not differentiate among process, job and program. However, when confusion is likely to arise, we will be specific.

A program, therefore, must be converted into a process before it can be executed by the O/S. This implies that one cannot build an O/S without defining process. The O/S must know the intention of a program and it is only possible if the program becomes "alive", i.e., converted into a process. It is the process which represents a user program in the O/S.

There are a number of additional properties of a process. Strictly speaking a process must finish in a finite time. If a program runs for ever, then it is not regarded a process. However, there are some system routines that run continuously as long as the system is up, for example, poling routines, routines that check the status of peripheral devices, etc. These modules are also called process. But we will use the formal definition where a process must terminate in a finite time. We assume that the speed of a process is unpredictable since it is impossible to keep track of the speed of a process execution.

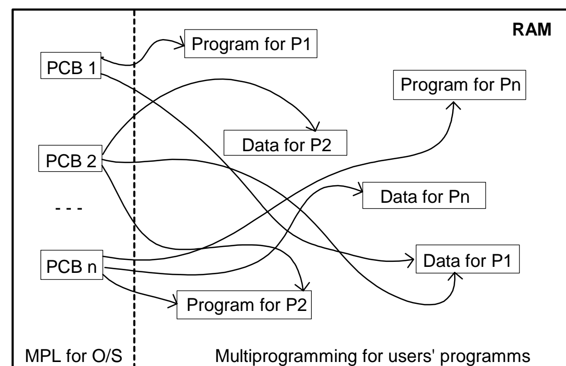
Process Implementation

Implementation of a process means preparing the package and store it at a storage device (disk or RAM). This package is formatted as a table and is referred to as **Process Control Block (PCB)**. A typical PCB may look like as follows:



Process Control Block (PCB)

Each PCB is loaded in RAM and it can only be accessed by the O/S. A PCB contains pointers to other parts of the memory that holds data and object code of your process as shown in the above diagram. The data used by your process can also be used by other processes, i.e., data can be shared among a number of processes. In this case there are pointers from many processes to the same set of data. This creates the problem of process synchronization and will be discussed later.



RAM with a set of processes

A set of PCBs can be scattered all over the main memory and at one time only one PCB will be used by the O/S. In many cases a procedure is also shared by many processes. If a code

sharing is allowed then the O/S must guarantee that the code is pure (pure procedure or reentrant procedure), i.e., the code is not changed by a process or it does not modify itself. So in a multiprogramming system the state of the RAM at any time may look as shown in the above figure.

Process Creation: A process must be created for any program before it can begin execution. If the program already has a process then it may create (spawn) another process for smaller task.

Example: EDIT MYFILE System call: CREATE

Steps

1. Locate the load module file for EDIT.
2. Allocate memory for a PCB
3. Allocate a swap file for the new process and copy the program image (object code) to it.
4. Complete the various fields of the PCB, storing name, program size, disk swap file address, null base register value (the process is initially swapped out), save PSW pointing to the first instruction of the program, and so on.
5. Send a message containing the initialization text MYFILE.
6. Notify the dispatcher that new process is to be initiated and its PCB placed in the ready queue.

PSW: A process may be in various states during its entire execution life. The state of a process is recorded in a word called "Program Status Word (PSW)" and is stored in a special purpose register (hardware). A typical PSW may look like:

run/wait	**sys/proc**	**mask**	**map**	**condition code**	**program counter**

Run/wait: One bit. Indicates whether the process is running or waiting.
System/process: One bit. Indicates operation in either the system state or process state, with the primary objective being the authorization to execute privileged instructions.
Mask: A bit string specifies which interrupt classes are enabled (e.g., I/O or timer interrupt)
Map: One bit. Indicates whether or not memory address mapping is enabled
Condition code: A bit string, which holds the result of the last operation.
Program counter: Holds the address of the next instruction to be executed or of the current instruction if an error occurs.

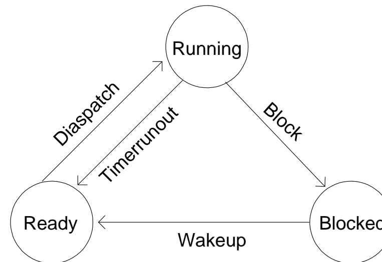
Process State: A process is a dynamic entity, so it can go into several states:

1. Being executed by the CPU: Running.
2. Waiting for a resource: Waiting.
3. Ready for execution (waiting only for CPU): Ready.
4. Blocked by another process: Blocked.
5. Suspended (temporary hold): Suspended.

Process state transition: A process may move from one state to another many times during its execution life. Such movement of processes is caused by a number of factors. Processes change state mainly to improve resource utilization and to satisfy scheduling policy. If there is an infinite number of resources (CPU, files, I/O processors, peripherals, etc.) then a process's request for a resource will never be denied. In this situation the state transition becomes meaningless. No computer system can have an infinite number of resources. If there are n

processes then at time T at least the resource request of process i may not be satisfied and the O/S then must force process i to wait by moving it to the relevant state. Since there are a number of transitions possible a set of rules is defined to manage state transition.

State transition rules: State transition is represented by a State Transition Diagram (a digraph). It indicates the next possible state of a process and the operation that initiates the transition. State transition must follow certain rules. Some transitions are not logically possible. A simple state transition diagram looks like:



Process State Transition

A process changes its state when some event occurs. For example, if a process is waiting (blocked) for a line printer that is being used by another process, then the waiting process goes to ready state when the line printer becomes available (event).

State transition modules

Dispatcher: changes the state from ready to running (assignment of CPU to a ready process is called dispatching).

dispatch (P_name): ready → running

Timerrunout: processes timeshare CPU resource (typically less than 1 microsecond). If the process does not voluntarily relinquish the CPU before the time interval expires, the clock generates an interrupt causing the operating system to regain control and the following state transition takes place:

timerrunout (P_name): running → ready. dispatch (P_name): ready → running

Block: if a running process initiates an I/O operation before its CPU share expires, the running process voluntarily relinquishes the CPU (i.e., the process blocks itself pending the completion of the I/O operation). This state transition is:

block (P_name): running → blocked

Wakeup: when the required I/O completes, the blocked process is activated and the transition is:

wakeup (P_name): blocked → ready

IMPORTANT: the only state transition initiated by the user process itself is block; the other three transitions are initiated by entities external to the process.

Sometime it becomes necessary to force a state change externally (by human operator). For example, if a process is using a significant amount of CPU resource or a process needs a data that has not yet been created by another process, for some reason a process is slowing down the performance of the system, etc. In these situations the operator can move a process to one of the suspension states.

Short term suspension: process P1 is waiting for some input from process P2, and P2 has not produced the desired input then P1 will be suspended, not blocked or ready, until P2 sends the data to P1. When P1 gets the data, it goes to ready state.

Long term suspension: a process is not likely to get its desired resources for sometime or the process is affecting other parts of the system (performance etc.) or the operator may wish to suspend a process for the following reasons:

- **a**. System is functioning poorly and may fail.
- **b**. A user suspicious about the partial results of a process may suspend it (rather than aborting it).
- **c**. In response to short-term fluctuations in system load, .

Two new states for managing process suspension,

Suspendready: Suspension of a ready process: **suspend (P_name): ready → suspendready**.

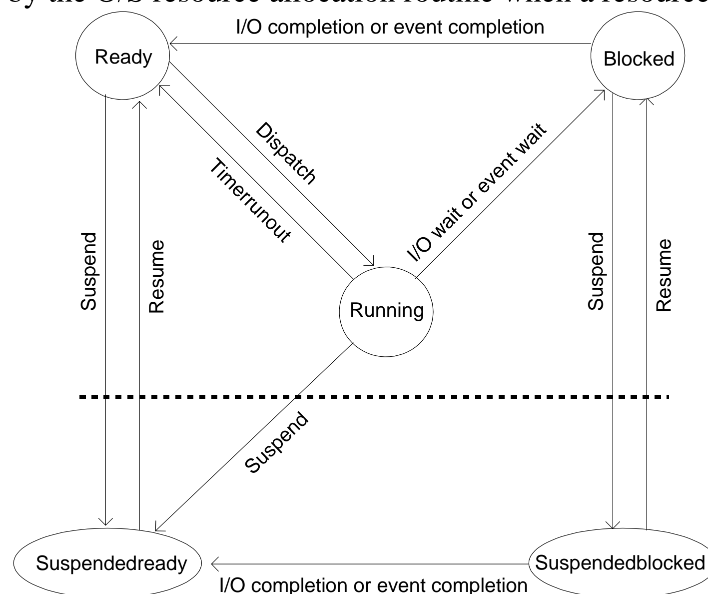
A suspendready process may be resumed: **resume (P_name): suspendready → ready**

Suspendblocked: Suspension of a blocked process: **suspend (P_name): blocked → suspendblocked**

It is possible that a suspendblocked process may be changed to suspendeready on the completion of the desired I/O. In this case the transition is:

completion (P_name): suspendblocked → suspendready.

To keep track of all processes, the O/S maintains list of PCBs allowed. In general, there is a ready list, which contains PCBs of all ready processes, a waiting list which contains PCBs of all waiting processes, etc. By means of these lists, the O/S forms pools of processes in similar states which are examined by the O/S resource allocation routine when a resource becomes available.

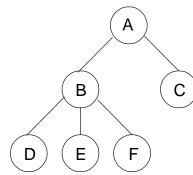


Process State Transition Diagram

State change: Occurrence of some event may lead to a state change. A state change is achieved by moving the relevant PCB from one list to another appropriate list. Since PCBs are always on

the move the common data structure for their management is a linked list. To improve system performance a linked list with multiple pointers are used.

Process hierarchy (Spawning) and process creation: A process may spawn a new process. The creating process is called the parent process and the created process is called the child process. In turn the child process may create another process and thus a hierarchical process structure is created. Processes may be dynamically created or destroyed. In a simple O/S, a set of processes already exist. In such a simple system no more processes are ever required. In more complex systems there are system calls to create a process or child process dynamically. In this way, at any time a tree of processes may exist in the system. For example, during system initialization a special process called boot loads the O/S. It then creates a process for each terminal connected to the system. The following diagram gives a snapshot of an O/S state:



Management of process hierarchy: In many systems when a parent process dies then all its children also die since there is only one PCB for the entire process tree. Some of these children may be "threads" (discussed later) of the parent process. For example, in the above diagram D, E, and F may be threads of process B, which is the child of process A. In other systems the elimination of parent process does not affect children process. This however, is difficult to manage since the O/S must create separate PCBs for each child or grandchild. It must maintain a link between the ancestors and dependents since a dependent any time may request some information from its parent. These requirements make the O/S very complex.

Interrupt

Sometime it is necessary to alter the normal execution of a process. This is done by generating an interrupt. Such break in the execution of a process may be generated by the process itself or by the operating system. An interrupt generated by the operating system (register overflow, division by zero, addressing error, etc.) is usually termed as trap and the interrupt generated by the user process is regarded as a normal interrupt.

Example: Pressing control and C key on the keyboard during the execution of a process, switching off the line printer while printing result, etc.

Management of interrupts: When an interrupt occurs the following steps are taken

1. OS takes control.
2. Save status of the process during which interrupt occurred.
3. Decide which interrupt routine is to be used to manage the interrupt.
4. Load and execute that routine.
5. Resume normal execution of the process (saved in 2) at the end of interrupt processing.

Example

Event: During printing line printer ran out of paper. **User process**: P1.
An I/O interrupt is generated by the I/O program handling the printing process.

1. Save status of the printing process (how far executed, what is the next instruction when stopped and many other things)

2. OS takes control.
3. Looks at the interrupt code and decides which interrupt routine is responsible for detecting the problem.
4. Loads this routine from the library.
5. Creates process (system process) to run this routine.
6. This interrupt routine prints message "LP out of paper".
7. Operator loads paper.
8. Interrupt routine detects this and passes control to OS.
9. OS removes the interrupt process and loads P1.
10. End of interrupt processing.
11. Resumes execution of P1.

Interrupt facility improves CPU efficiency. Interrupts can be generated by hardware or software. All interrupts can be categorized into six different classes:

SVC (supervisor call): A user process, if requires more memory or any other resource asks the OS by SVC instruction. An interrupt is generated and the request is either denied or granted.

I/O interrupt: Generated by I/O system (hardware and software combination) during I/O processing. They can be generated when an I/O completes, error during file transfer, printing etc.

External interrupt: External to the system. By operator or by another CPU in a multiprocessor system.

Restart interrupt: Bootstrap. System reloading.

Program check interrupt: Interrupt during normal execution of a process. For example, register overflow, division by zero etc.

Machine check interrupt: By hardware.

Context Switching: CPU is taken away from one process and assigned to another process. The PCB of the process that has to give up the CPU is saved some place in the memory or on the disk and the new process uses the CPU.