Lesson 7, 8

Objectives

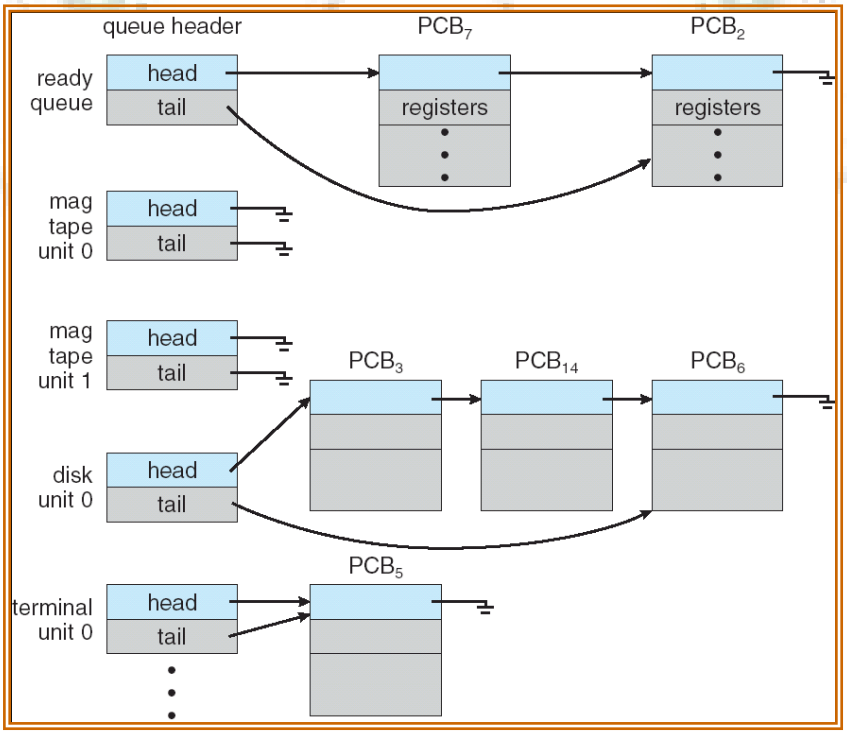
- Process Scheduling
- Operations on processes
- Cooperating processes

PROCESS SCHEDULING

The objective of multiprogramming is to have some process running all the times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. For a uni-processor system it is not possible to run more than one processes at same time, so rest of the processes have to wait for CPU to be free. This is called process scheduling.

*P*rocess Scheduling Queues

- **Job queue** – set of all processes in the system that are just entered
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute. This queue is generally stored as a linked list. A ready queue header will contain pointer to the first and last PCBs in the list
- **Device queues** – set of processes waiting for an I/O device (waiting state)

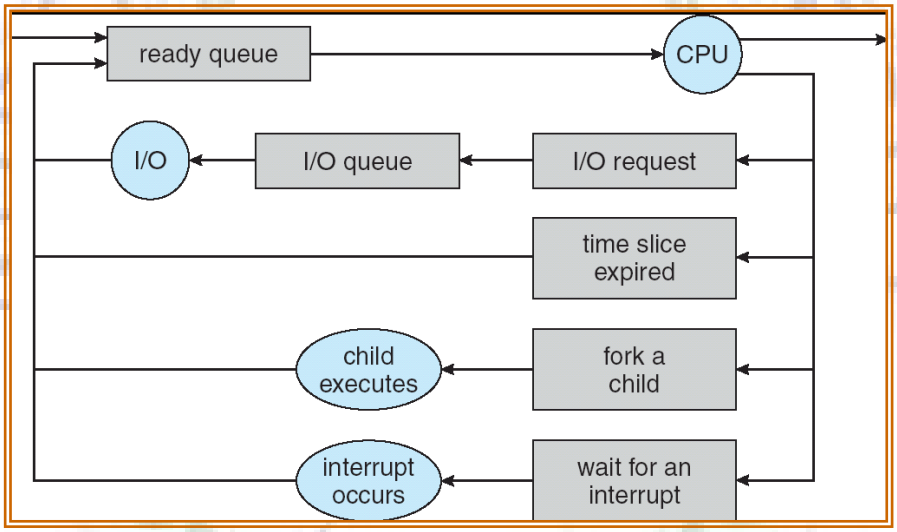


A new process is initially put into a *ready queue*. It waits for its selection for execution in CPU. Once it is given the CPU and is executing, one of several events could occur, like

- The process could issue an I/O request, and then be placed in an I/O queue
- The process could create a new sub-process and wait for its termination
- The process could be removed forcibly from CPU, as a result of an interrupt, and may be put back into ready queue

In above cases process switches between the waiting to ready states, this cycle goes on until process is terminated. Then it is removed from all queues and has its PCB and resources de-allocated.

Following is given a *queuing diagram representation* of process scheduling.



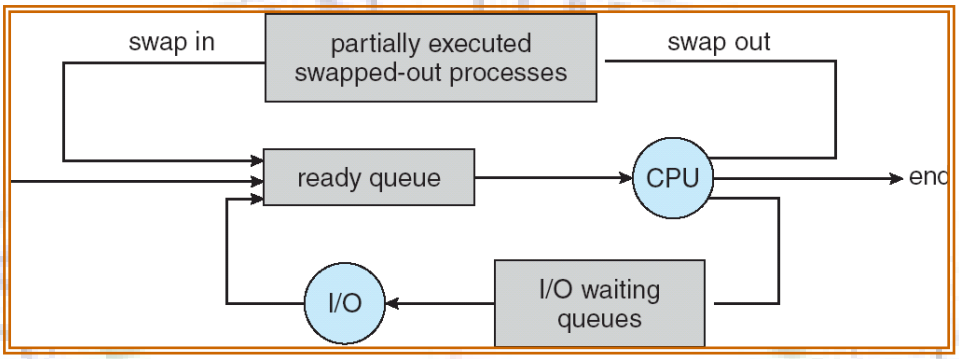
Schedulers

Processes migrate among the various queues through their lifetime. So operating system must select these processes from these queues in some fashion. This is carried out by an *scheduler*.

There are three types of schedulers.

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue. It is less frequent. It controls the *degree of multiprogramming* (the number of the processes in the memory) so it has much time to decide which process should be en-queued. Since processes are of two types *I/O bound* which takes much time in I/O than in execution, and *CPU bound* are the processes which take more CPU time compared to I/O activities.

- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocated CPU. This should be very fast since its decision time is subject to CPU utilization also it is utilized very frequently.
- **Medium-term scheduler** (or swapper) – it offers medium level of scheduling. Sometime it is advantageous to remove from main memory and CPU, while this process need to come back again, so instead of removing it permanently it is kept in medium-term scheduler such that can be accessed back for convenience.



***C*ontext Switch**

Switching the CPU to another process requires saving the state of old process and loading the saved state for the new process. This task is called *context switch*. It is an overhead since in that while CPU sits idle. Its speeds varies in the range (1-1000µsec) form machine to machine depending on

- Memory speed
- Number of register which may be copied
- Existence of special instruction for save

OPERATIONS ON PROCESSES

The processes in the system can be executed concurrently (occur at the same time), and must be created and deleted dynamically. Hence OS gives a mechanism to carry on these steps.

***P*rocess Creation**

1. A process can create several processes, via a *CreateProcess* system call, during execution. The creating process is called *parent process*, while created process called its *child process*
2. It is possible that a sub-process share a subset of its parent resources

3. Principal events that cause process creation
    a. System initialization

Foreground and background processes (**Foreground-background** is a scheduling algorithm that is used to control execution of multiple processes on a single processor. It is based on two waiting lists, the first one is called **foreground** because this is the one in which all processes initially enter, and the second one is called **background** because all processes, after using all of their execution time in foreground, are moved to background.

When a process becomes ready it begins its execution in foreground immediately, forcing the processor to give up execution of current process in the background and execute newly created process for a predefined period. This period is usually 2 or more quanta. If the process is not finished after its execution in the foreground it is moved to background waiting list where it will be executed only when the foreground list is empty. After being moved to background, process is then run longer than before, usually 4 quanta. The time of execution is increased because the process obviously needs more than 2 quanta to finish (this is the reason it was moved to background). This gives the process the opportunity to finish within this newly designated time. If the process does not finish after this, it is then preempted and moved to the end of the background list.

The advantage of the foreground-background algorithm is that it gives process the opportunity to execute immediately after its creation, but scheduling in the background list is pure round-robin scheduling**).**

    b. Execution of a process creation system call by a running process
        i. Concept of independent interacting processes
    c. User request to create a new process
    d. Initiation of a batch job
4. System calls to create new process are **Fork** in UNIX and **Create_Process** in windows
5. Execution:
    a. A parent continues to execute concurrently with its children
    b. A parent waits until some or all of its child terminated
6. Address space:
    a. Child duplicate of parent
    b. Child has a program loaded into it
7. UNIX examples
    a. **fork** system call creates new process
    b. **exec** system call used after a **fork** to replace the process' memory space with a new program

***P*rocess Termination**

Conditions which terminate processes:

1. Normal exit (voluntary) — successful termination

2. Error exit (voluntary) — erroneous termination
3. Fatal error (involuntary) — terminated by operating system
4. Killed by another process (involuntary) — parent kills child

For example, the parent may terminate the child due to the following reasons:

- The child has exceeded its usage of some of the resources allocated
- The task assigned to the child is no longer required
- The parent is exiting and the operating system does not allow a child to continue if its parent terminates it is called *cascading termination*

COOPERATING PROCESSES

In operating system processes can either be independent—when they does not affect or effected by any other process, or cooperating –which may affect and effected by other processes. This is done due to the following reasons.

***I*nformation Sharing**

Since several users may interested in same piece of information (for example, a shared file)**,** we must provide an environment to allow concurrent access to these types of resources.

***C*omputation Speedup**

One way to make a task speedy we can divide it to subtasks, each executing in parallel to other. This can be obtained in multiprocessing environment.

***M*odularity**

A system may be modularized in order make more smooth and speedy.

***C*onvenience**

It is convenient for an individual user to do many tasks in single time. Like editing, printing, compiling etc.

Example

To illustrate the concept of cooperating system, let us take an example of **consumer-producer processes.** Producer produces the information while consumer consumes it. Here are two scenarios for this one is called *unbounded-buffer* and other is *bounded-buffer*.

In unbounded buffer, producer can produce unlimited information in buffer or buffer has no practical limit. If the buffer is empty then consumer has to wait for the new item.

In bounded buffer, producer can produce up to filling of buffer then he waits if buffer is full, till some portion is utilized, similarly consumer has to wait for new item if buffer is empty earlier.