CSC 453 Operating Systems

Lecture 4 : Processes

The Process Concept

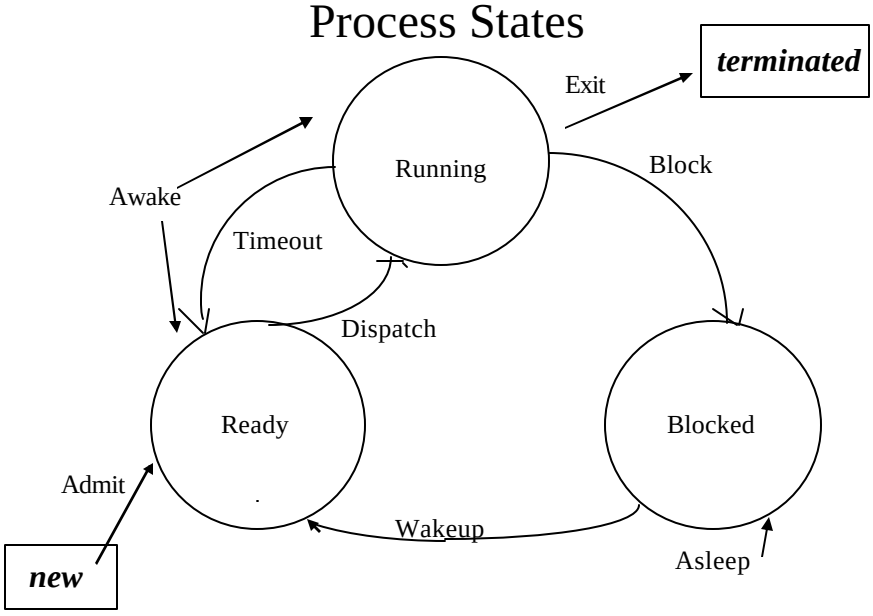
- Originally, computers ran only one program at a time, which had total access to all of the computer's resources.
- Modern systems run many programs concurrently, with the operating system running its own tasks to do its own work.
- The concept of the ***process*** allows us to compartmentalize everything being done on the computer.

What is a Process

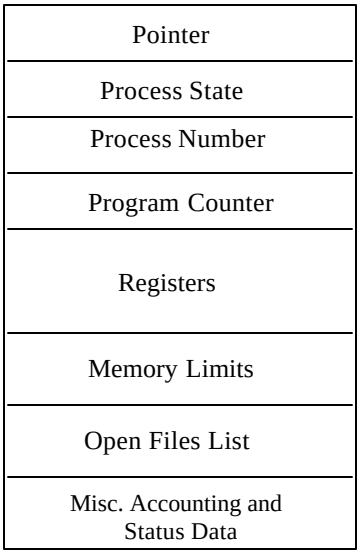
Harvey Deitel offers several definitions of a ***process***:

- a program in execution
- an asynchronous activity
- the "animated spirit" of a procedure
- the "locus of control" of a procedure in execution
- that which is manifested by the existence of a process control block" in the operating system
- that entity to which processors are assigned
- the "dispatchable" unit

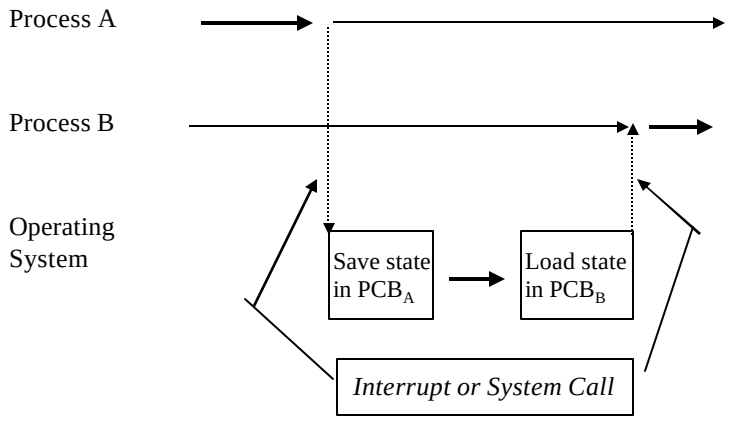
Process States



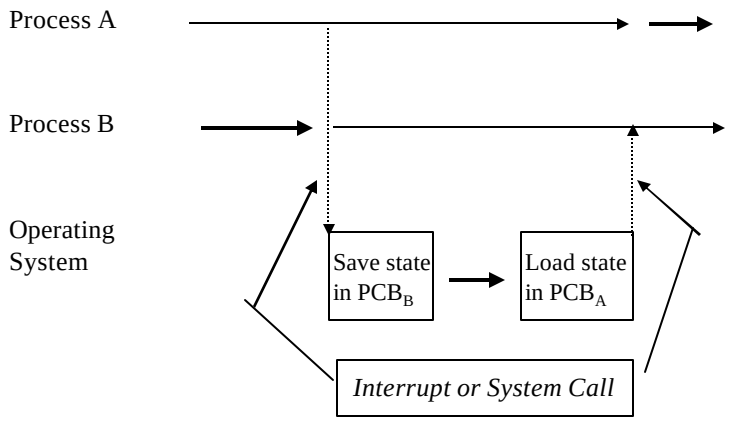
Process Control Block



Switching Processes



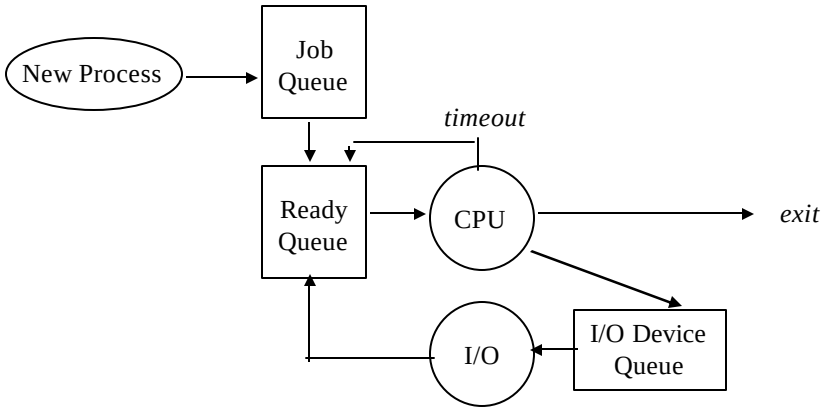
Restoring Processes



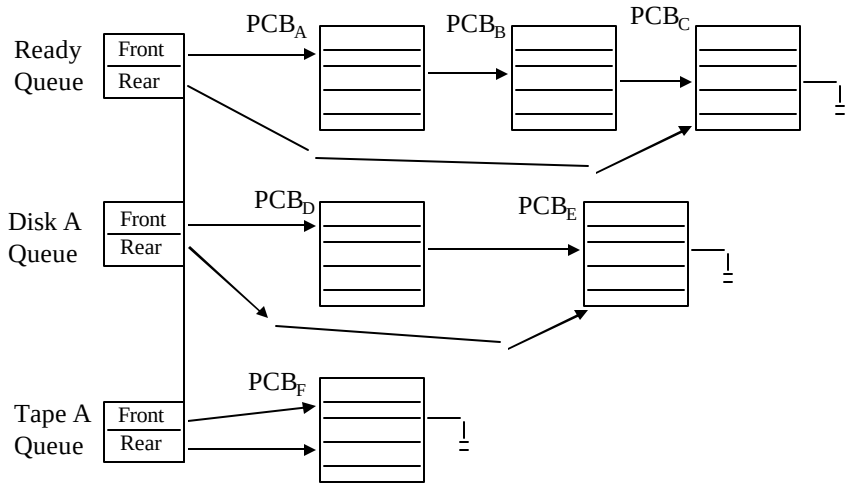
Context Switch

- ***Context switch*** – saving the state of one process and loading the saved state of another process.
- Context switch times varies depending on the hardware, with typically times of 1 to 1000 μsec.

Process Scheduling



Scheduling Queues



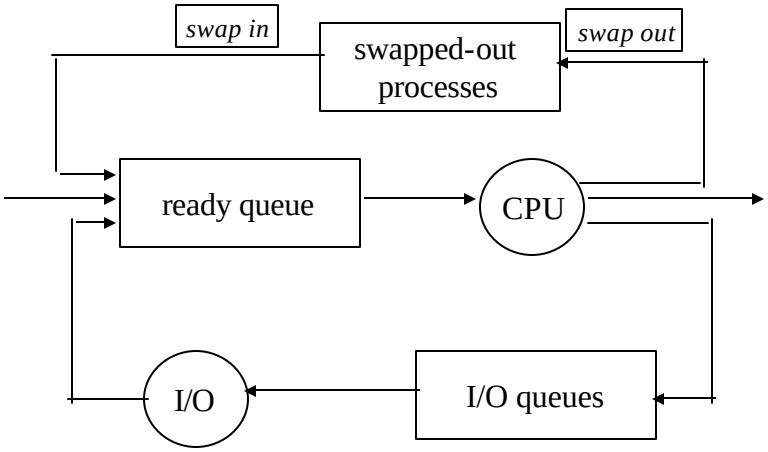
Long-term and Short-term Scheduling

- The long-term scheduler selects SPOOLed processes and loads them into memory.
  - The long-term scheduler executes less frequently.
  - The long-term scheduler controls the degree of ***multiprogramming***.
- The short-term scheduler selects the next process which will have the active use of the CPU.
  - The short-term scheduler executes quite frequently; therefore it must be quick .

CPU-Bound and I/O Bound Processes

- ***CPU-bound processes*** - Processes that spend more of their time doing computational work than input and output
- ***I/O-bound processes*** – Processes that spend more of their time doing input and output than computational work.

Medium-Term Scheduling

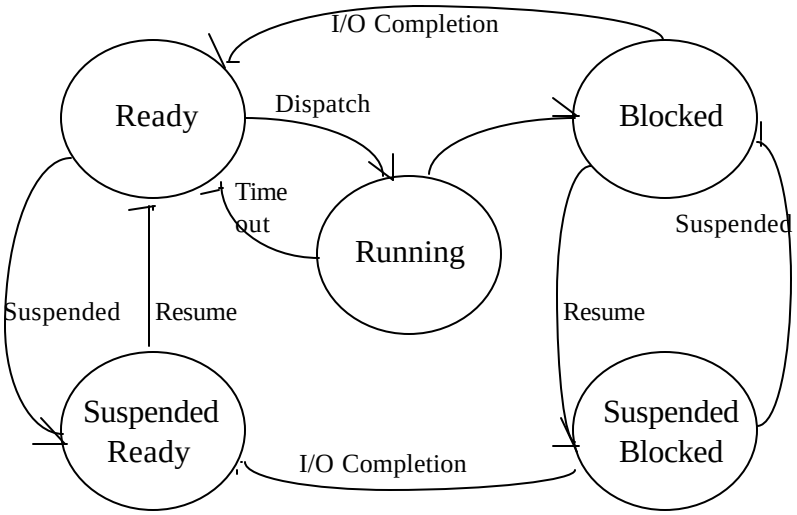


Why Suspend A Process?

There are several reason for suspending a process:

- If a system is functioning poorly and may fail, then current processes may be suspended to be resumed after the problem is corrected.
- A user suspicious about the partial result of a process may suspend it (as opposed to aborting it) until ascertaining whether the process is functioning correctly.
- Some processes may be suspended to ease the load on a system until the load reaches normal levels.

Suspended and Active Processes



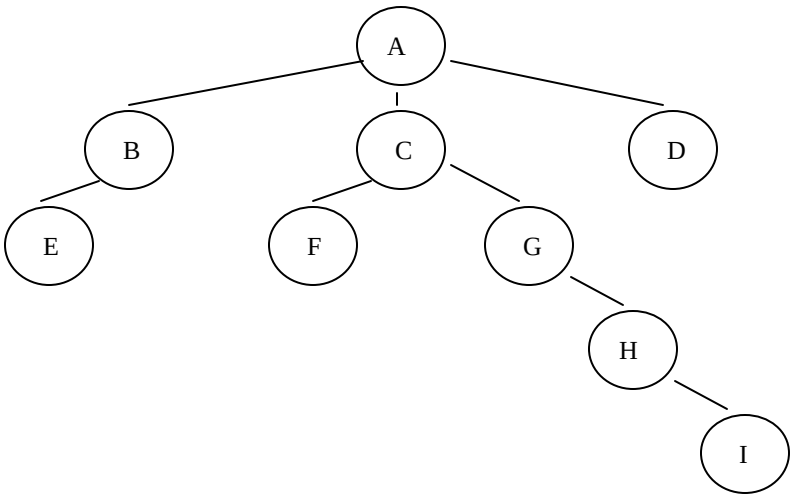
Suspending A Process

- A process may suspend itself (a running process on a uniprocessor system would have to do this) or it can be suspended by another process.
- While one may argue that it is better to wait for I/O to be completed before suspending a blocked process, the process may never reach the ready state; therefore, suspending a blocked process may be the only option.

Creating A Process

- Processes may be created by other processes with the ***parent process*** partitioning off resources for the ***child process*** and passing it data.
- The child process may also obtain resources directly from the operating system.
- A hierarchy can be created as child processes create their own child processes.

Hierarchy of Processes



Tasks In Process Creation

Creating a process requires that the operating system:

- name the process
- insert it in the process table
- determine its initial priority
- create the process control block
- allocate its initial resources

Parent and Child Processes

- Once a child process is created:
  - both parent and child may execute concurrently or
  - the parent become inactive until the child process terminates.
- Different possibilities exist for the child's ***core image***:
  - The child is a duplicate of the parent
  - The child is a separate and distinct program

Process Creation in UNIX

A simplified version of the shell

```
while (TRUE)      {     /* repeat forever */
  read_command(command, parameters);
            /* read terminal input */

  if (fork() != 0)
            /* fork off child process */
       /* Parent code */
       waitpid(-1, &status, 0);
  else
       execve(command, parameters, 0);
            /* execute command */
}
```

Process Termination

- Normally, a process terminates at the end of program execution by using the **`exit`** system call.
  - All its resources are either freed or returned to the parent's control. Data may also be returned to the parent.
  - A parent may also terminate a child process by using an **`abort`** system call.

Why Kill A Child Process?

A parent process might terminate a child process because:

- the child process used more resources than it is allowed.
- it is no longer needed.
- the parent process is terminating. The process of processes killing their descendent processes is called ***cascading***.

Cooperating Processes

- Cooperating Processes are processes that can affect or can be affected by other processes.
- Allowing processes to cooperate allows:
  - information sharing
  - computational speedup
  - modularity
  - convenience

Consumer-Producer Problem

- The consumer-producer problem is a classic example of how processes cooperative.
- Consumers consume information produced by the producers but must wait when there is nothing to consume.
- Producers produce information for the consume to consume but must wait when there is no place to put it.

Common Code For Consumer and Producer

```
const int numbuffers = …;
typedef … item;
item buffer[numBuffers];
int  in = 0, out = 0;
     /* in and out have values
     in the range 0 to numbuffers-1 */
```

The Producer Process

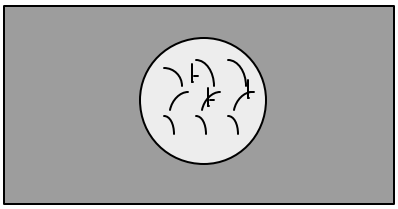
```
item nextp;
do   {
     Produce an item in nextp;
     … … …
     while ((in +1) % numbuffers == Out)
               ;
     buffer[in] = nextp;
     in = (in + 1) % numbuffers
} while True;
```

The Consumer Process

```
item      nextc;
do   {
     while (in == out)
          ;
     nextc = buffer[out];
     out = (out + 1) % numbuffers;
     … … …
     Consume the item in nextp;
} while True;
```

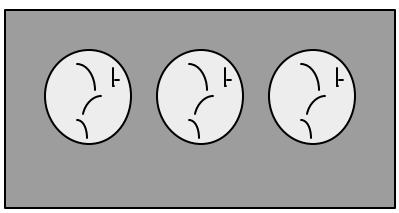
What are threads?

- ***Threads*** are lightweight processes which are their own thread of control but share an address space, allowing them to more easily coordinate their actions



Why Threads?

- Processes each have at least one thread of control, a series of instructions performed in synchronous fashion.
- Processes each have their own address space their own set of resources and require a mechanism provided by the operating system to communicate with each other or to share those resources.



Examples of Multithread Processes

- Examples of natural applications for threads:
  - A file server process, which each thread scheduling a file server request.
  - World Wide Web browsers.

Thread Structure

- Some systems fully support threads.
  - Some of the process table entries belong to the individual threads.
  - The operating system is fully cognizant of the use of multiple threads per process.
  - In such a system, when a thread is blocked, the operating system decides which thread becomes active.
- Other systems manage threads solely within user space. A thread being blocked chooses its successor as the active thread.

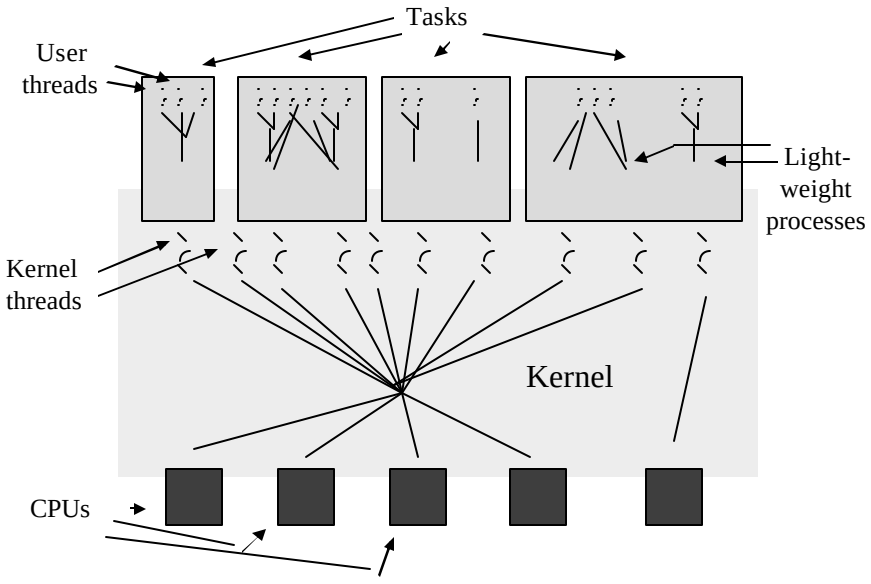
Threads and Operating Systems

- Operating systems using threads include Windows NT, Solaris, Mach and OS/2.
- UNIX is a one thread per process operating system.

Supporting Threads

- Some systems have threads supported by the kernel. In other systems, it is supported on the user level by library calls.
  - It takes longer to switch kernel-supported threads.
  - User-level support can cause the whole process to wait on account of one thread and can leads to unfair scheduling.
- Some operating systems, such as Solaris 2, support both mechanisms.

Example: Threads in Solaris 2



Interprocess Communication

- An easy way to have processes cooperate is to have the operating system provide an interprocess communication (IPC) facility .
- The basic structure of a message system are the operations: **`send`**(*message*) and **`receive`**(*message*).
- These operations assume that we have the ability to open a communications channel between these processes.

Implementation Issues for Interprocess Communication

- In establishing a communications link between processes, we must consider several questions:
  - How do we establish links?
  - How many processes can share a link?
  - How many links can processes share?
  - What is the link's capacity?
  - How large can messages be?
  - Is communication one-way or two-way?

Implementing A Link

- There are several ways to implement link:
  - Direct or Indirect Communications
  - ***Symmetric*** or ***Asymmetric*** Communications
  - ***Automatic*** or ***Explicit*** Buffering
  - ***Send by Copy*** or ***Send By Reference***
  - Fixed-Size or Variable-Sized Messages

Direct Communication

- In this case, we define our operations as:
  - send(P, message)
  - receive(Q, message)
- This automatically establishes exactly one link between exactly 2 processes.
- This link is most likely unidirectional but can be bidirectional.

Direct Communication and the Producer-Consumer Problem

The producer process:

```
REPEAT
    … … …
    produce an item in nextp
    … … …
    send (consumer, nextp);
  UNTIL False;
```

The consumer process:

```
REPEAT
    receive(producer, nextc);
    … … …
    consume the item in nextp
    … … …
UNTIL False;
```

Indirect Communication

- In this case, messages are sent and received via a mailbox (or port).
- We define our operations as
  - send(BoxA, message)
  - receive(BoxA, message)
- Links can now be established between more than 2 processes.
- 2 processes can share more than one link.
- Links can be unidirectional or bidirectional.

Message Buffering

- Links may have the capacity to store message temporarily before they are received. This depends on the link's buffering capacity.
- The buffering capacity can be zero, bounded or unbounded, which determines whether the sender is delayed.

Exception Conditions

- A message system in a distributed environment must be able handled communications failures that may not result in the failure of the entire system.
- Such failures may involve:
  - messages to or from a process that has terminated
  - messages that are lost
  - messages that are deliverable but are erroneous.

Example: Message Passing in Mach

- Mach was developed at Carnegie-Mellon University
- Communications, including system calls are made by message, which are sent to and received from ***ports***.
- Although ports can be full, the sending thread has the choice of whether to wait, how long to wait or whether to ***cache*** the message.

Example: Message Passing in Windows NT

- Application programs communicate via a message passing facility called the Local Procedure Call facility (LPC), which uses indirect communication.
- NT uses message passing even for rudimentary functions such as graphics - this tends to slow down the system.