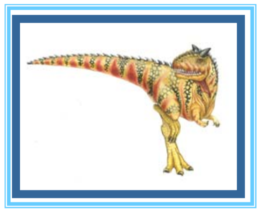
Chapter 3: Processes



(slides selected/reordered/modified by R. Doemer, 01/06/11)

Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

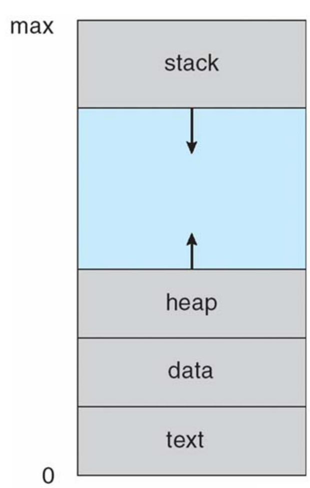
(slide modified by R. Doemer, 04/06/10)

Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process:
  - a program in execution
  - process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section

(slide modified by R. Doemer, 04/02/10)

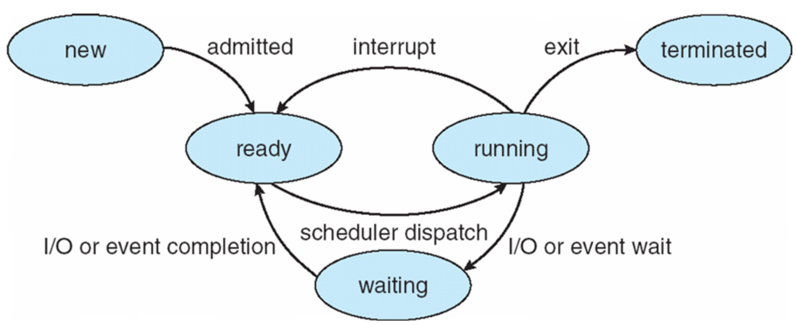
Process in Memory



Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

Diagram of Process State

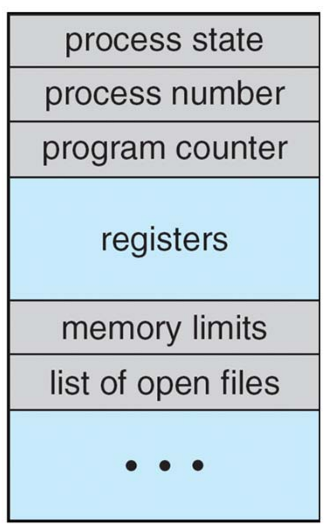


Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process Control Block (PCB)

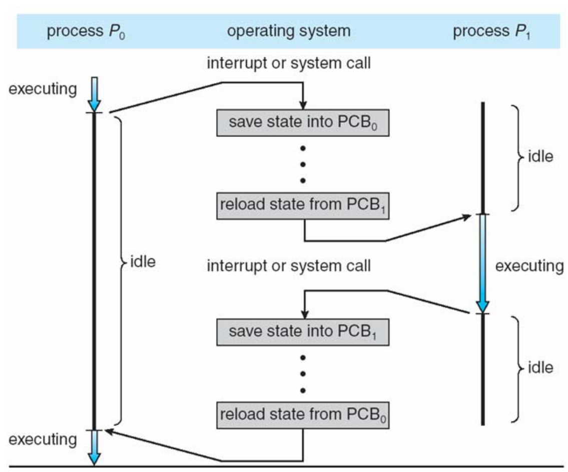


Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**
- **Context** of a process is represented in the PCB
- Context-switch time is *overhead*; the system does no useful work while switching
- Context-switch time is dependent on hardware support

(slide modified by R. Doemer, 04/02/10)

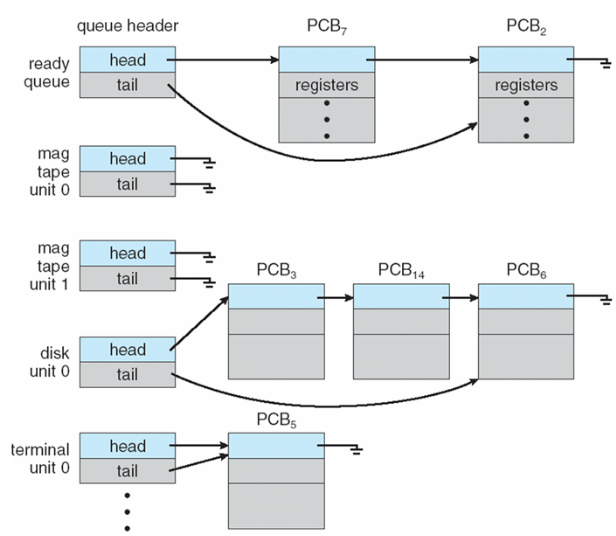
CPU Switch From Process to Process



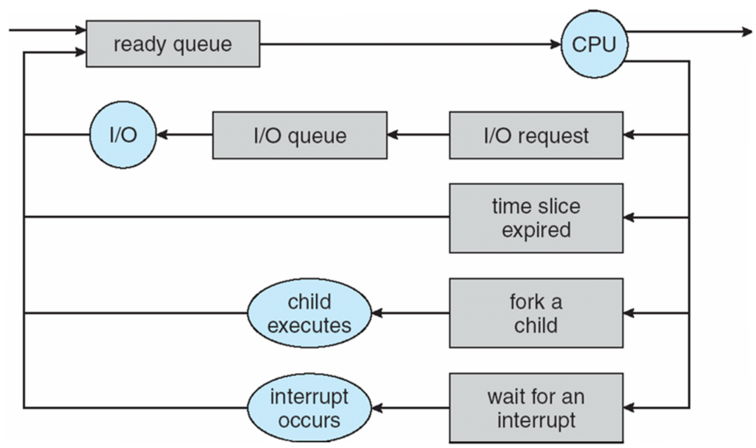
Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



Process Creation

- **Parent** process create **child** processes,
  which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options:
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options:
  - Parent and children execute concurrently
  - Parent waits until children terminate

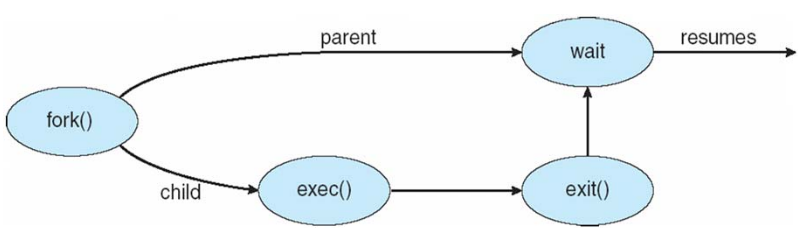
(slide modified by R. Doemer, 04/02/10)

Process Creation (Cont)

- Address space options:
  - Child is a duplicate of parent
  - Child has a program loaded into it
- UNIX example
  - **fork** system call creates new process
    (as an almost identical copy of the parent)
  - **exec** system call is used after a **fork**
    to replace the process' memory space with a new program (from disk)
  - **wait** system call allows parent to wait for child completion

(slide modified by R. Doemer, 04/02/10)

Process Creation in Unix



(slide modified by R. Doemer, 04/02/10)

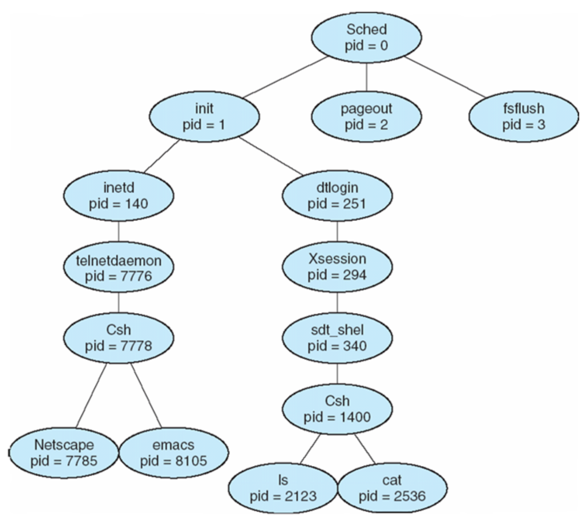
C Program Forking a Child Process

```
int main()
{
   pid_t  pid;

    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
         fprintf(stderr, "Fork Failed");
         return 1;
    }
    else if (pid == 0) { /* child process */
         execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
         /* parent will wait for the child to complete */
         wait(NULL);
         printf ("Child Complete");
    }
   return 0;
}
```

(slide modified by R. Doemer, 04/02/10)

A tree of processes on a typical Solaris system



(slide modified by R. Doemer, 04/02/10)

Process Termination

- Process executes last statement (returns from main()), or asks the operating system to delete it (**exit**)
  - Output *status* from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**

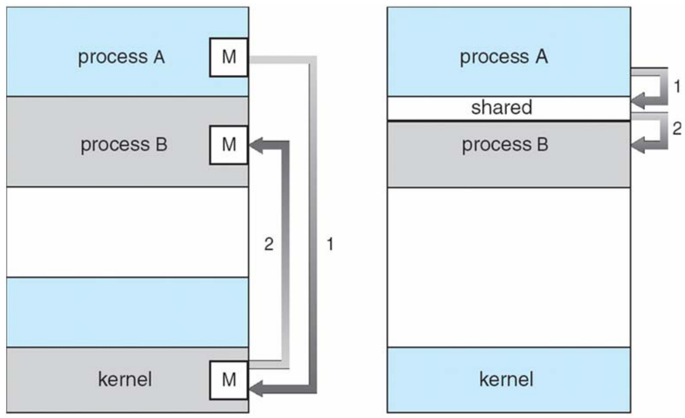
(slide modified by R. Doemer, 04/02/10)

Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - Shared memory
  - Message passing

(slide modified by R. Doemer, 04/02/10)

Inter-Process Communications Models



(a) Message Passing

(b) Shared Memory

(slide modified by R. Doemer, 04/02/10)

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

(slide modified by R. Doemer, 04/06/10)

End of Chapter 3