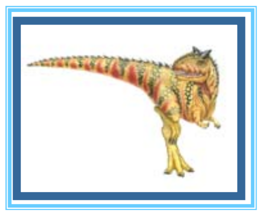
Chapter 3: Processes



(slides selected/reordered/modified by R. Doemer, 04/06/10)

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)

Objectives

- To introduce the notion of a process –
  a program in execution,
  which forms the basis of all computation
- To describe the various features of processes,
  including scheduling, creation and termination, and communication
- To describe communication in client-server systems

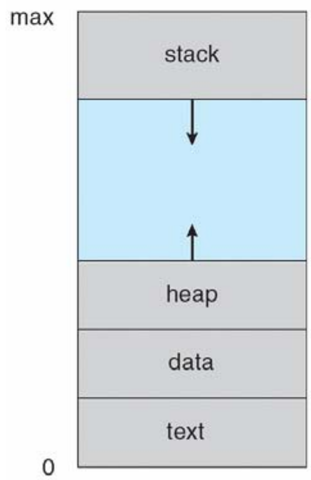
(slide modified by R. Doemer, 04/02/10)

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)

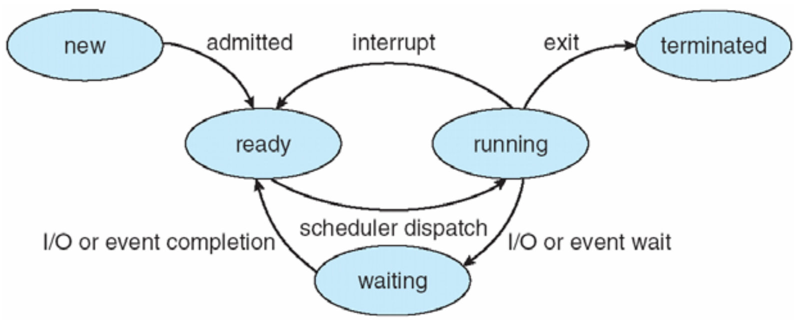
Process in Memory



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



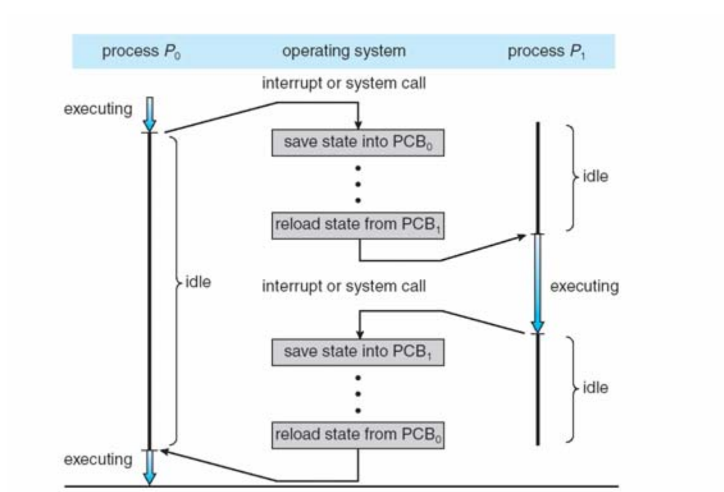
Process Control Block (PCB)

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)

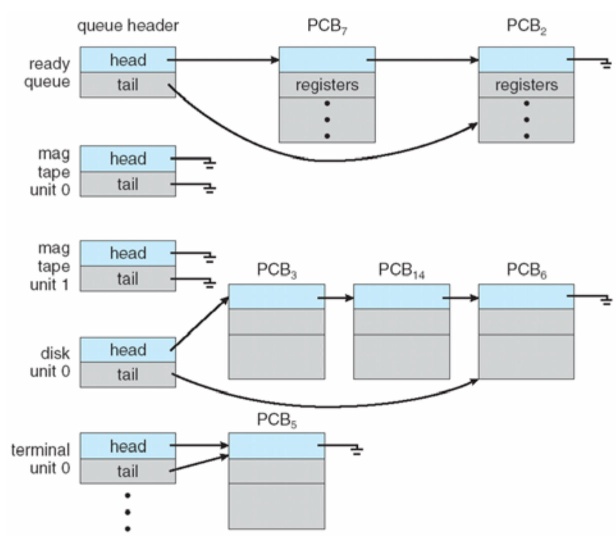
CPU Switch From Process to Process



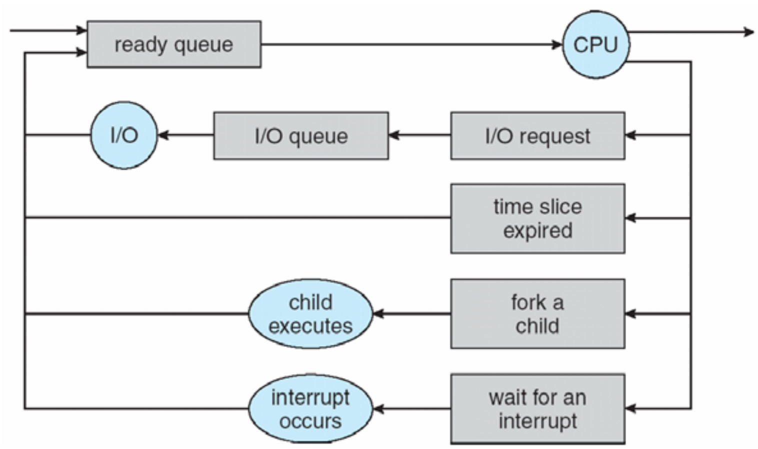
Process Scheduling Queues

- **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



Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

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

Schedulers (Cont)

- Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts

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

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)

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

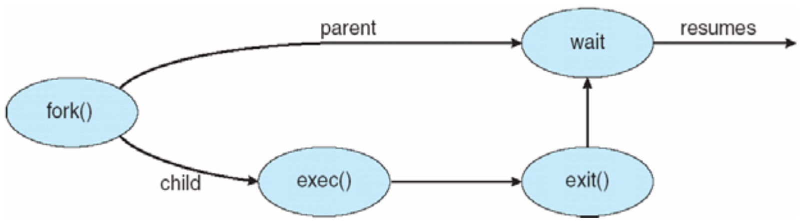
(slide modified by R. Doemer, 04/02/10)

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)

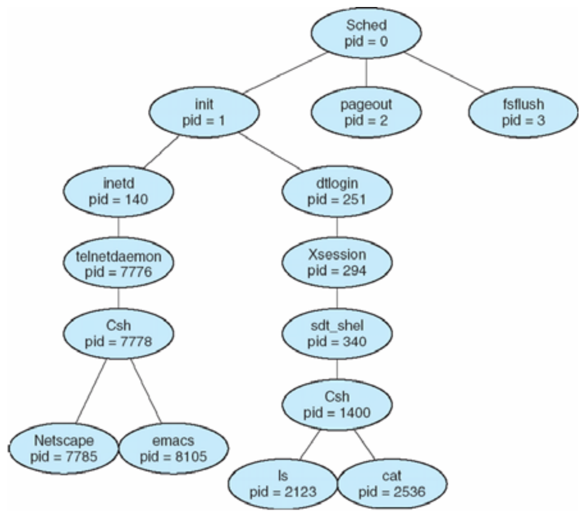
C Program Forking a Child Process

```
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**

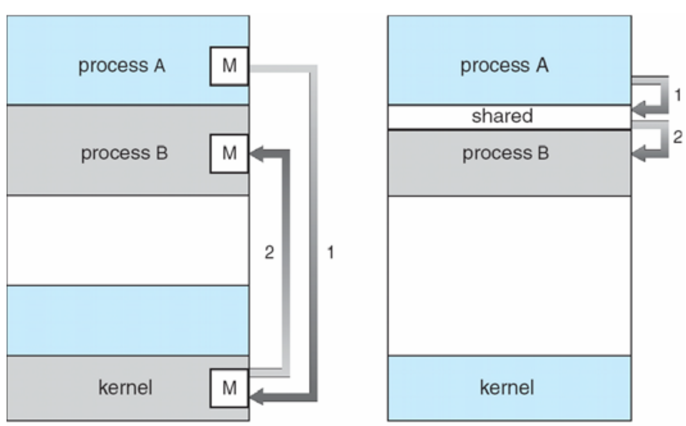
(slide modified by R. Doemer, 04/02/10)

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)

Producer-Consumer Problem

- Paradigm for cooperating processes
  - *Producer* process produces information that is consumed by a *consumer* process
  - *Unbounded-buffer* places no practical limit on the size of the buffer
  - *Bounded-buffer* assumes that there is a fixed buffer size
- EECS111 Note:
  - We will not discuss the *bounded buffer* implementation here because it requires proper synchronization
  - We will postpone this until the discussion of *process synchronization* (Chapter 6)

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

Interprocess Communication – Message Passing

- Mechanism for processes to communicate *and* synchronize their actions
- Message system
  - processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

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

Message Passing: Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be
  between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate
  fixed or variable?
- Is a link unidirectional or bi-directional?

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

Message Passing: Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

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

Message Passing: Indirect Communication

- Messages are directed and received from **mailboxes** (also referred to as *ports*)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

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

Message Passing: Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

  send(*A, message*) – send a message to mailbox A

  receive(*A, message*) – receive a message from mailbox A

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

Message Passing: Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

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

Message Passing: 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)

Message Passing: Buffering

- Queue of messages attached to the link;
  implemented in one of three ways
  1. **Zero capacity** – 0 messages
     Sender must wait for receiver (*rendezvous*)
  2. **Bounded capacity** – finite length of *n* messages
     Sender must wait if link full
  3. **Unbounded capacity** – infinite length
     Sender never waits

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

Examples of IPC Systems - POSIX

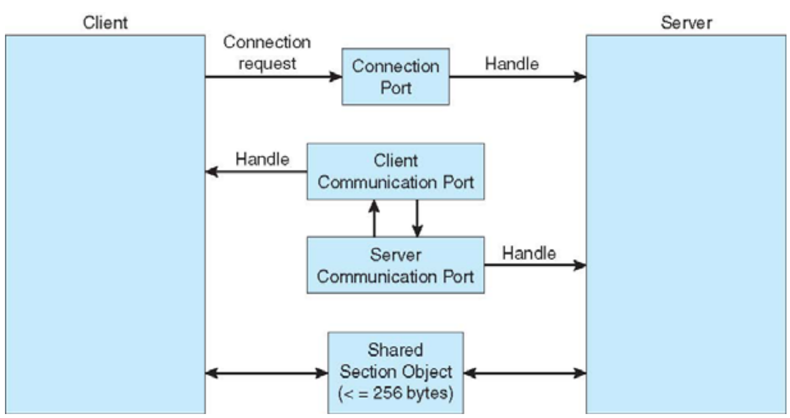
- **POSIX Shared Memory** *(adjusted for EECS111 Solaris servers)*
  - Process first creates a shared memory segment
    `int sid = shmget(IPC PRIVATE, size, SHM_R | SHM_W);`
  - Process wanting access to that shared memory must attach to it
    `void *shm = shmat(sid, NULL, 0);`
  - Now the process could write to the shared memory
    `sprintf(shm, "Writing to shared memory");`
  - When done, a process should
    (1) detach the shared memory from its address space, and
    `shmdt(shm);`
    (2) release the shared memory segment
    `shmctl(sid, IPC_RMID, NULL);`

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

Examples of IPC Systems – Windows XP

- Message-passing centric via local procedure call (LPC) facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem's connection port object
    - The client sends a connection request
    - The server creates two private communication ports and returns the handle to one of them to the client
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies

Local Procedure Calls in Windows XP



Communications in Client-Server Systems

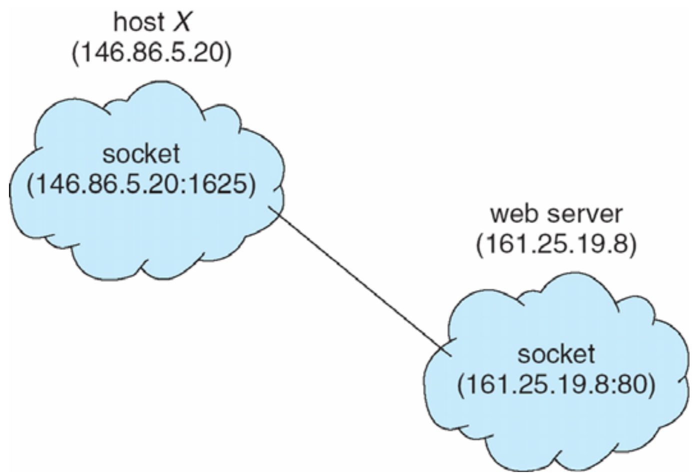
- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Sockets

- A **socket** is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625**
  refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets

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

Socket Communication

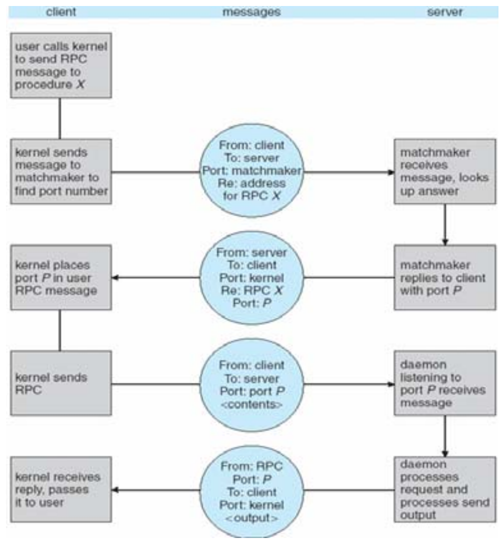


Remote Procedure Calls

- **Remote procedure call** (RPC)
  abstracts procedure calls between processes on networked systems
- **Stubs** – client- and server-side proxies for handling the actual procedure
- The client-side stub locates the server,
  marshalls and packs the parameters, and
  sends a message to the server
- The server-side stub receives the message,
  unpacks the marshalled parameters, and
  performs the procedure on the server
- Result values are returned to the client the same way
- If port numbers are not fixed beforehand,
  a **matchmaker** is used to negotiate ports

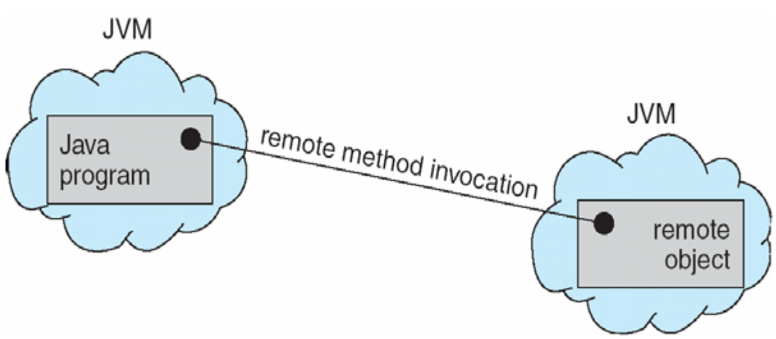
(slide modified by R. Doemer, 04/06/10)

Execution of RPC



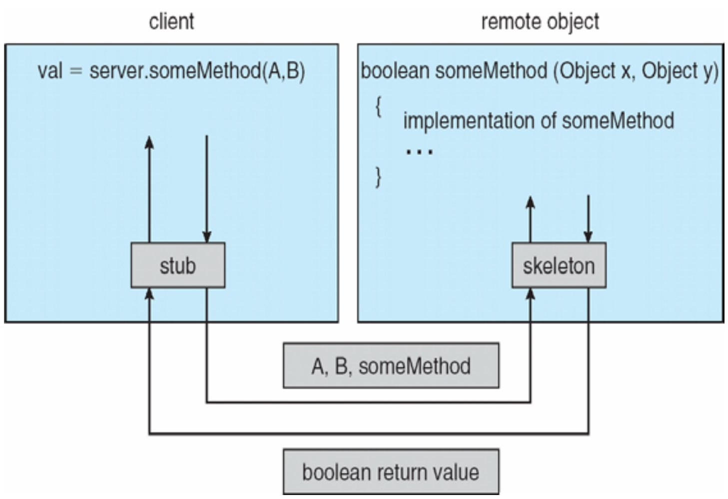
Remote Method Invocation in Java

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs
- RMI allows a Java program on one machine to invoke a method on a remote object



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

Marshalling Parameters (Java RMI)



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

End of Chapter 3