Chapter 3: Processes

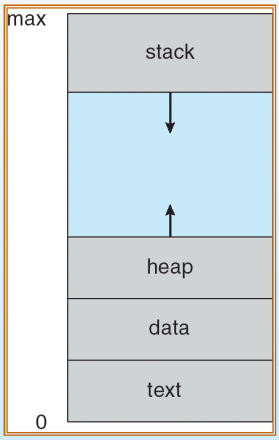
Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Cooperating Processes
- Interprocess Communication
- Communication in Client-Server Systems

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
  - Text section / program code
- More that *just* the program code
  - Program is not a process == passive entity
  - Process == active entity

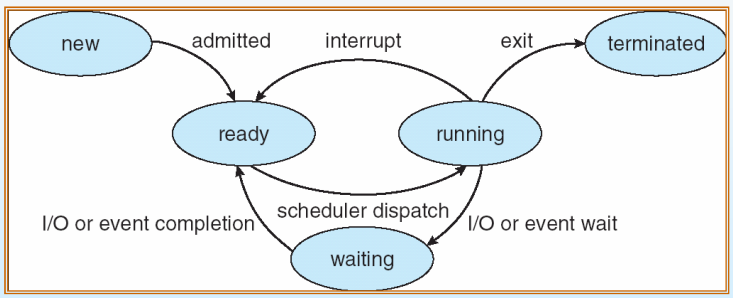
Process in Memory



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



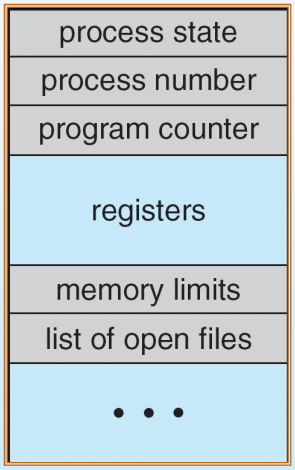
Only one process can be running on any processor at any instant

Process Control Block (PCB)

Information associated with each process

- Process state (new, ready, running, waiting, halted…)
- Program counter (@ of the next instruction)
- CPU registers (accumulators, stack pointer, condition-code information)
- CPU scheduling information
  - Priority, ptr to scheduling queues (chap 5)
- Memory-management information
  - Base, limit, page table (chap 8)
- Accounting information (CPU, real time)
- I/O status information (list of I/O allocated, open files)

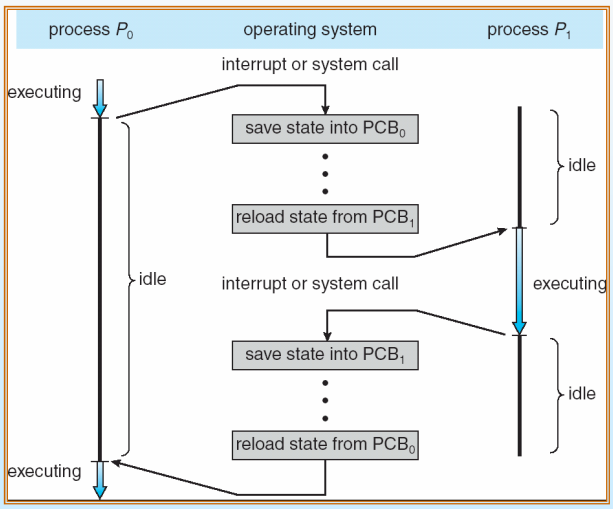
Process Control Block (PCB)



Process Scheduling

- Multiprogramming == maximize CPU utilization
  - → have some process running at all times
- Time sharing == switch the CPU among processes so frequently
  - → user can interact which each program
- **Process Scheduler** selects an available process

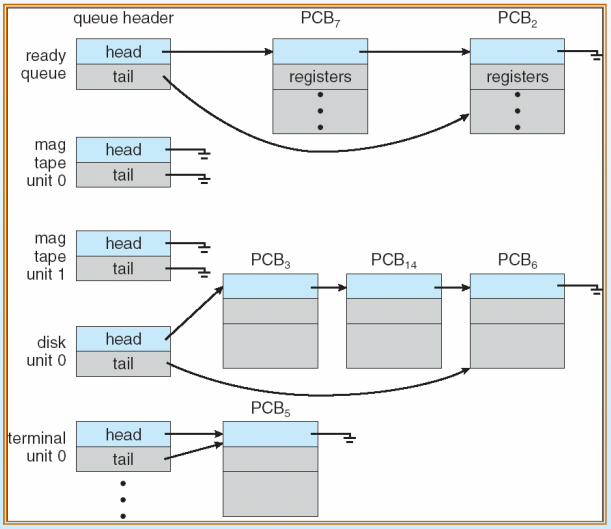
CPU Switch From Process to Process



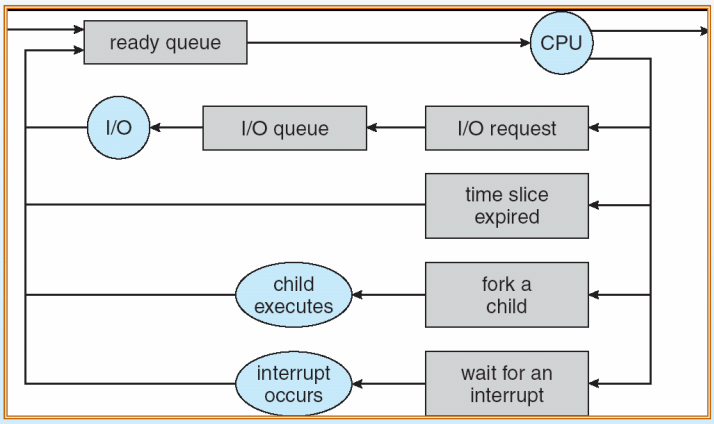
Process Scheduling Queues

- **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

- **A process migrates among the various scheduling queues during its lifetime**
- **The selection process is carried out by the appropriated scheduler**
- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
  - Batch style : more processes than can be executed. Pool on a disk
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

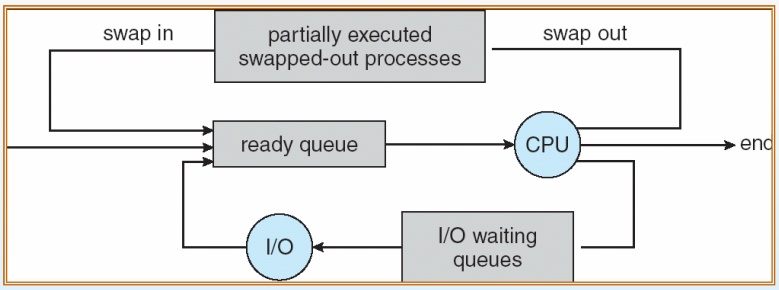
Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
  - Once every 100 millisecond, 10 millisecond to decide
  - → 9% wasted
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
  - → *# of processes in memory*
  - *Stable == rate of proc. creation == proc. Departure*
  - → *invoked only when a process leaves the system*
- 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

Long –term scheduler

- Must select a good **process mix**
- All processes are I/0 bound
  - Ready queue will almost always be empty
  - Short term scheduler will have little to do
- All process are CPU Bound
  - I/0 waiting queue will almost always be empty
  - Devices will go unused → system will be unbalanced
- Best performance → combination of CPU-bound **and** I/O-bound

Addition of Medium Term Scheduling



Long-term scheduler may be absent or minimal (UNIX / Microsoft – time sharing)

Advantageous to remove processes from the memory → reduce the degree of multiprogramming (**swapping**)

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
  - State save / state restore
- Context-switch time is **pure overhead**; the system does no useful work while switching
- Time dependent on hardware support
  - Sun UltraSPARC have multi set of registers
  - → context switch == changing the ptr to the current register set
  - Address space must be preserved
  - How / what amount of work depends on the memory-management (Chap. 8)

Operation on Processes

- Processes can execute **concurrently**
- May be created and deleted **dynamically**
- System must provide a way for
  - **Creation**
  - **Termination**

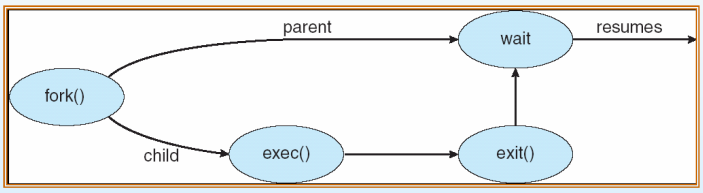
Process Creation

- **Parent** process create **children** processes (*create-process* system call)
  - which, in turn create other processes,
  - forming a tree of processes
  - **Unique process identifier (pid)**
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources (advantage ?)
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

Process Creation



C Program Forking Separate Process

```
#include <sys/types.h>
#include <unistd.h>
int main() {
    pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-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");
        exit(0);
    }
}
```

fork(): new process == copy of the address space of the original → communication between parent and child is easy

Both continue execution

Return value for the child is 0

Exec() system call used to replace the program in memory.

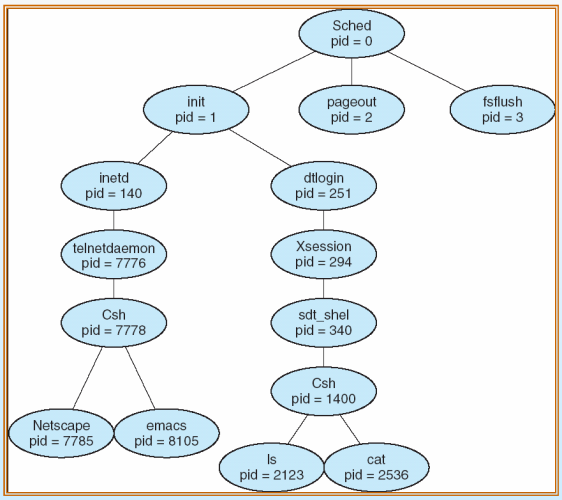
Wait() system call to move off the ready queue

Note sur fork

- **fork** crée un processus fils qui diffère du processus parent uniquement par ses valeurs PID et PPID et par le fait que toutes les statistiques d'utilisation des ressources sont remises à zéro. Les verrouillages de fichiers, et les signaux en attente ne sont pas hérités.

- Sous Linux, **fork** est implementé en utilisant une méthode de copie à l'écriture. Ceci consiste à ne faire la véritable duplication d'une page mémoire que lorsqu'un processus en modifie une instance
  - . Tant qu'aucun des deux processus n'écrit dans une page donnée, celle-ci n'est pas vraiment dupliquée. Ainsi les seules pénalisations induites par fork sont le temps et la mémoire nécessaires à la copie de la table des pages du parent ainsi que la création d'une structure de tâche pour le fils.

A tree of processes on a typical Solaris



Managing memory & file system

Root parent process for all user processes

Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
    - Physical / virtual memory, open files, I/O buffers
- 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 systems (VMS) do not allow child to continue if its parent terminates
      - All children terminated - *cascading termination*
    - *On Linux, children are assigned as their new parent the init process*

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

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
    - Consumer may have to wait
    - Producer can always produce new item
  - *bounded-buffer* assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Shared buffer == circular array
- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Insert() Method

```
while (true) {
  /* Produce an item */
  while (((in = (in + 1) % BUFFER SIZE count)  == out)
    ;   /* do nothing -- no free buffers */
   buffer[in] = item;
   in = (in + 1) % BUFFER SIZE;
}
```

Bounded Buffer – Remove() Method

```
while (true) {
    while (in == out)
        ; // do nothing -- nothing to consume

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
return item;
}
```

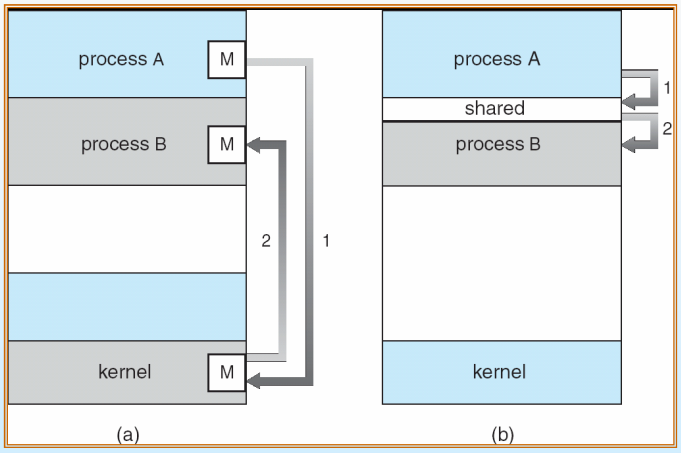
Interprocess Communication (IPC)

- Mechanism for processes to communicate and to 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)

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?

Communications Models



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

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

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

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.

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

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

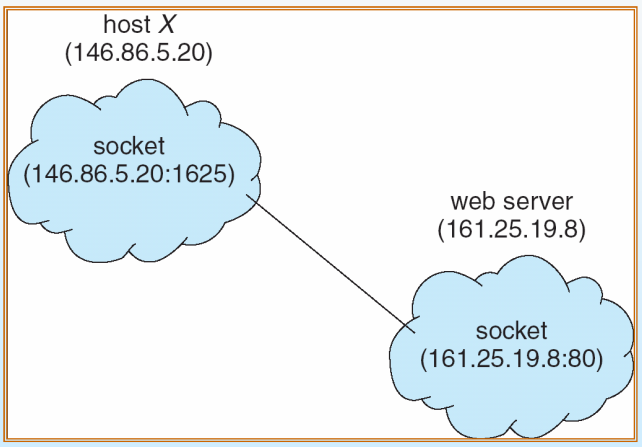
Client-Server Communication

- 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

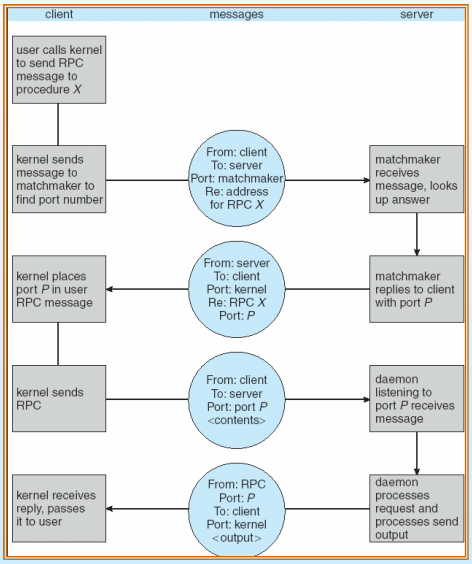
Socket Communication



Remote Procedure Calls

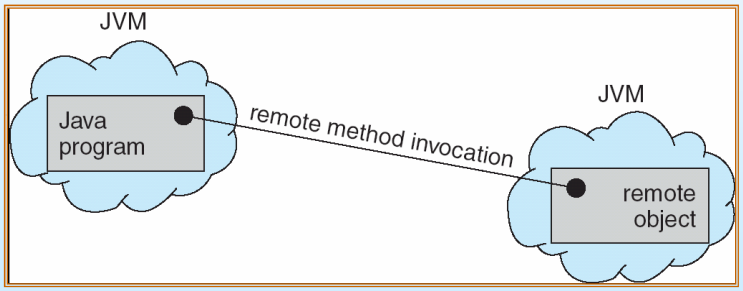
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- **Stubs** – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and *marshalls* the parameters.
- The server-side stub receives this message, unpacks the marshalled parameters, and peforms the procedure on the server.

Execution of RPC

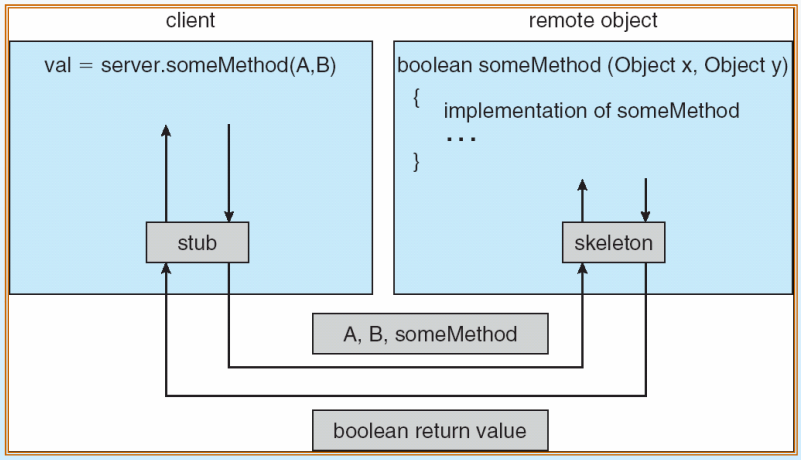


Remote Method Invocation

- 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.



Marshalling Parameters



End of Chapter 3