4. Processes

Sungyoung Lee

College of Engineering
KyungHee University

Contents

- *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

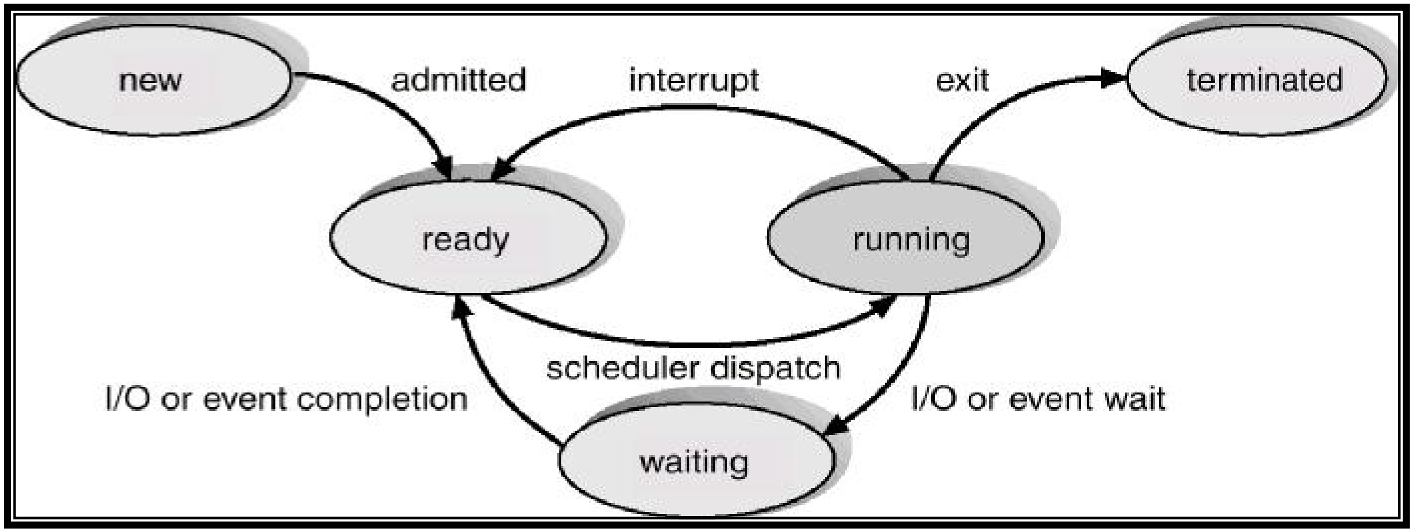
Process Concept

- What is the process?
  - An instance of a program in execution
  - An encapsulation of the flow of control in a program
  - A dynamic and active entity
  - The basic unit of execution and scheduling
  - A process is named using its process ID (PID)

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 process
  - **terminated**: The process has finished execution

Diagram of Process State



Process State Transition

- Linux example

```
xterm
  339 ?        S      0:21 clanmgr
  340 ?        S      0:00 clanmgr
  345 ?        S      0:00 clanagent
  346 ?        S      0:00 clanagent
  596 ?        S      0:00 syslogd -m 0
  601 ?        S      0:00 klogd -x
  621 ?        S      0:00 portmap
  649 ?        S      0:00 rpc.statd
  761 ?        S      0:00 /usr/sbin/apmd -p 10 -w 5 -W -P /etc/sysconfig/apm-sc
  821 ?        S      0:00 /usr/sbin/automount --timeout 300 /user file /etc/aut
  843 ?        S      0:00 /usr/sbin/sshd
  863 ?        S      0:00 xinetd -stayalive -reuse -pidfile /var/run/xinetd.pid
  905 ?        S      0:00 sendmail: accepting connections
  924 ?        S      0:01 gpm -t imps2 -m /dev/mouse
  942 ?        S      0:00 crond
 1016 ?        S      0:00 xfs -droppriv -daemon
 1052 ?        S      0:00 /usr/sbin/atd
 1059 ?        S      0:00 login -- jinsoo
 1060 tty2     S      0:00 /sbin/mingetty tty2
 1061 tty3     S      0:00 /sbin/mingetty tty3
 1062 tty4     S      0:00 /sbin/mingetty tty4
 1063 tty5     S      0:00 /sbin/mingetty tty5
 1064 tty6     S      0:00 /sbin/mingetty tty6
27499 ?        SW     0:00 [rpciod]
27500 ?        SW     0:00 [lockd]
27501 tty1     S      0:00 -tcsh
 5365 ?        S      0:00 /usr/sbin/sshd
 5367 pts/0    S      0:00 -tcsh
 5394 pts/0    R      0:00 ps ax
[oz0:/-5]
```

R: Runnable
S: Sleeping
T: Traced or Stopped
D: Uninterruptible Sleep
Z: Zombie

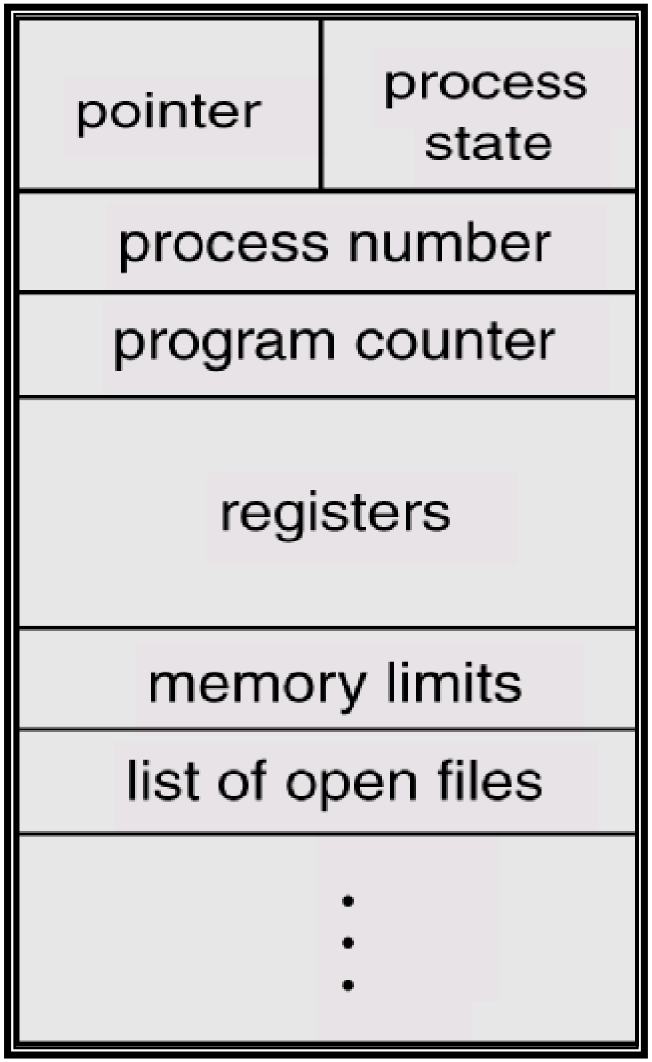
W: No resident pages
<: High-priority task
N: Low-priority task
L: Has pages locked into memory

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

- *Cf) task_struct in Linux*
  - 1456 bytes as of Linux 2.4.18

Process Control Block (PCB)



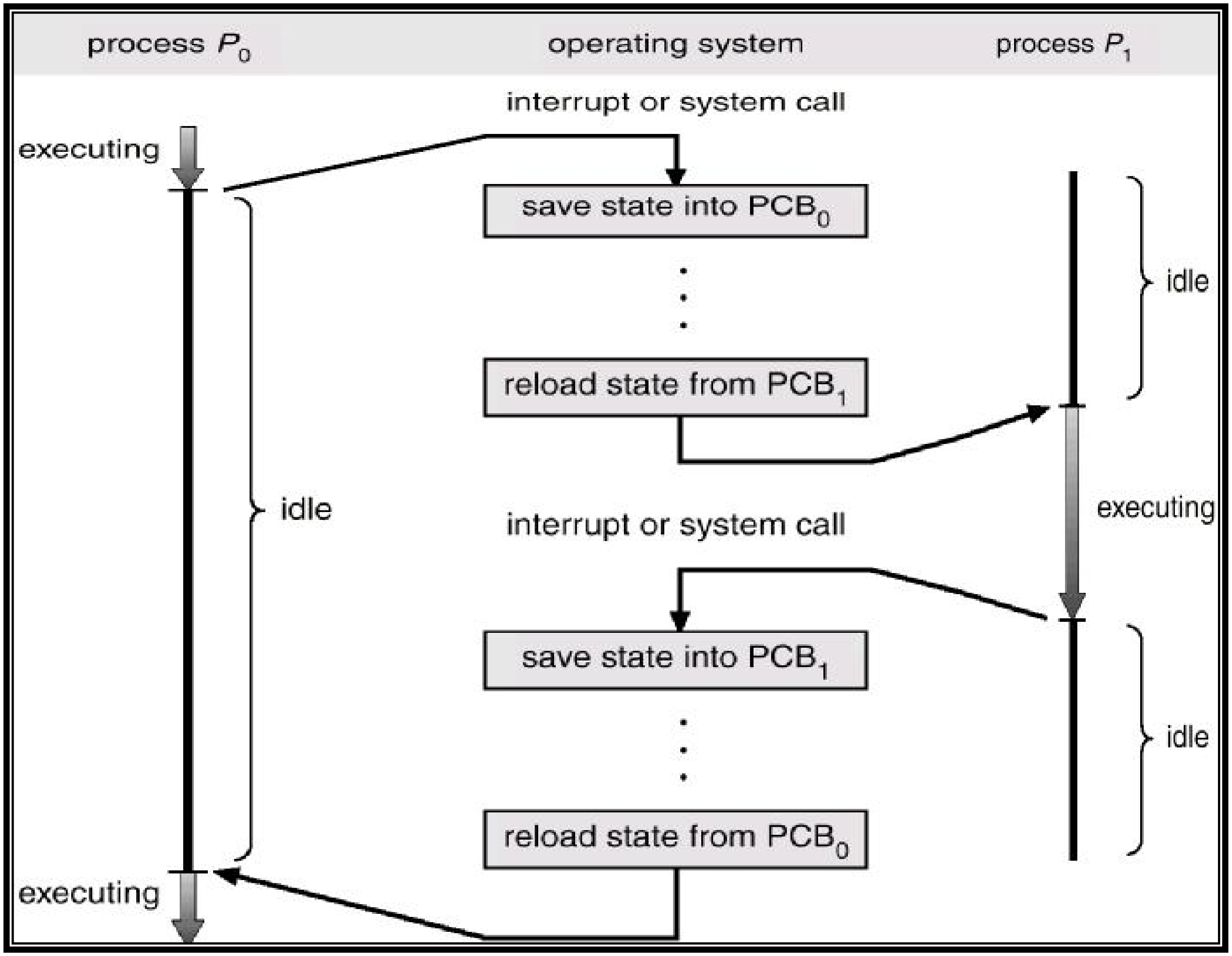
Process Control Block (PCB)

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

PCBs and Hardware State

- When a process is running:
  - its hardware state is inside the CPU: PC, SP, registers
- When the OS stops running a process:
  - it saves the registers' values in the PCB
- When the OS puts the process in the running state:
  - it loads the hardware registers from the values in that process' PCB

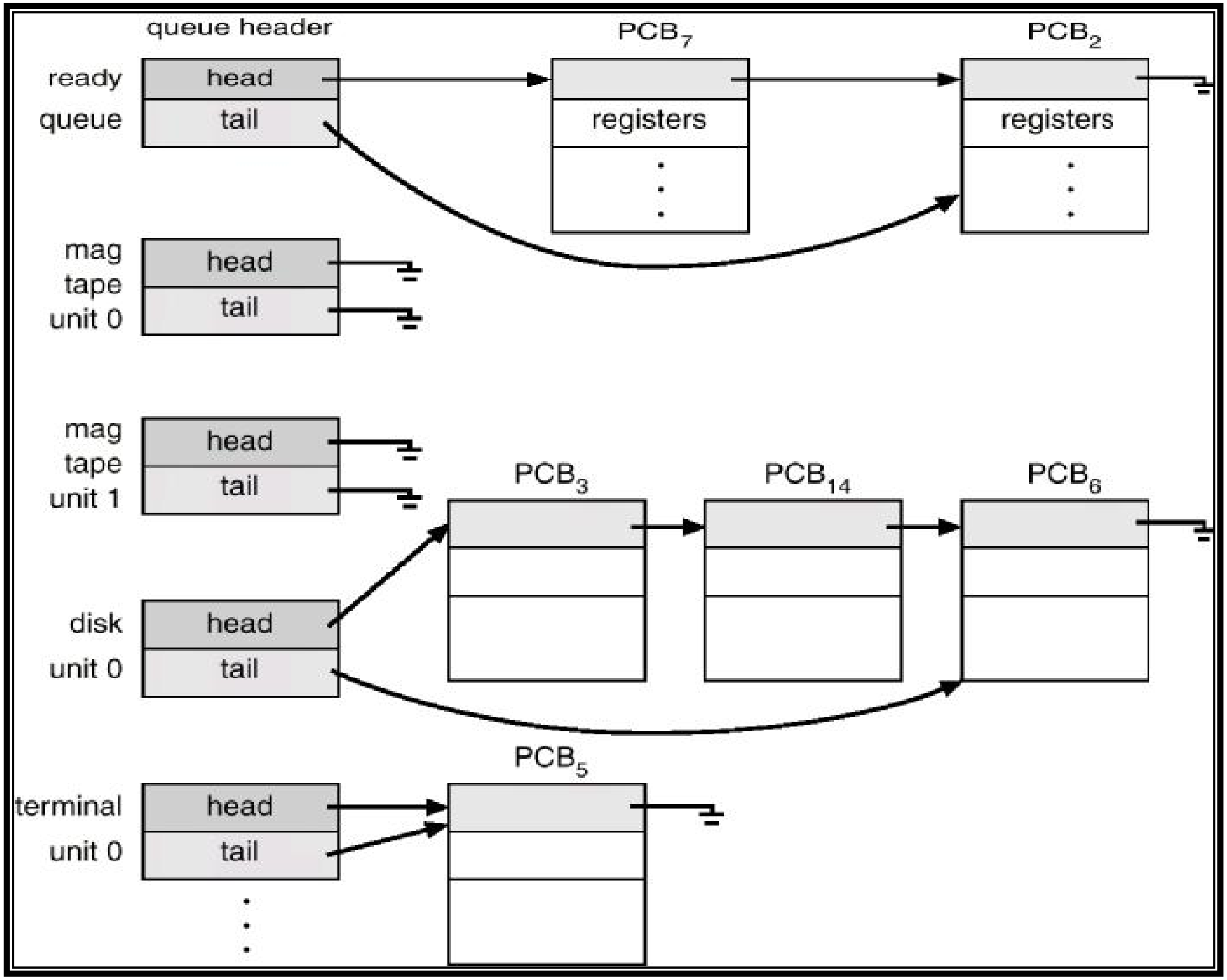
CPU Switch From Process to Process



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
- Process migration between the various queues

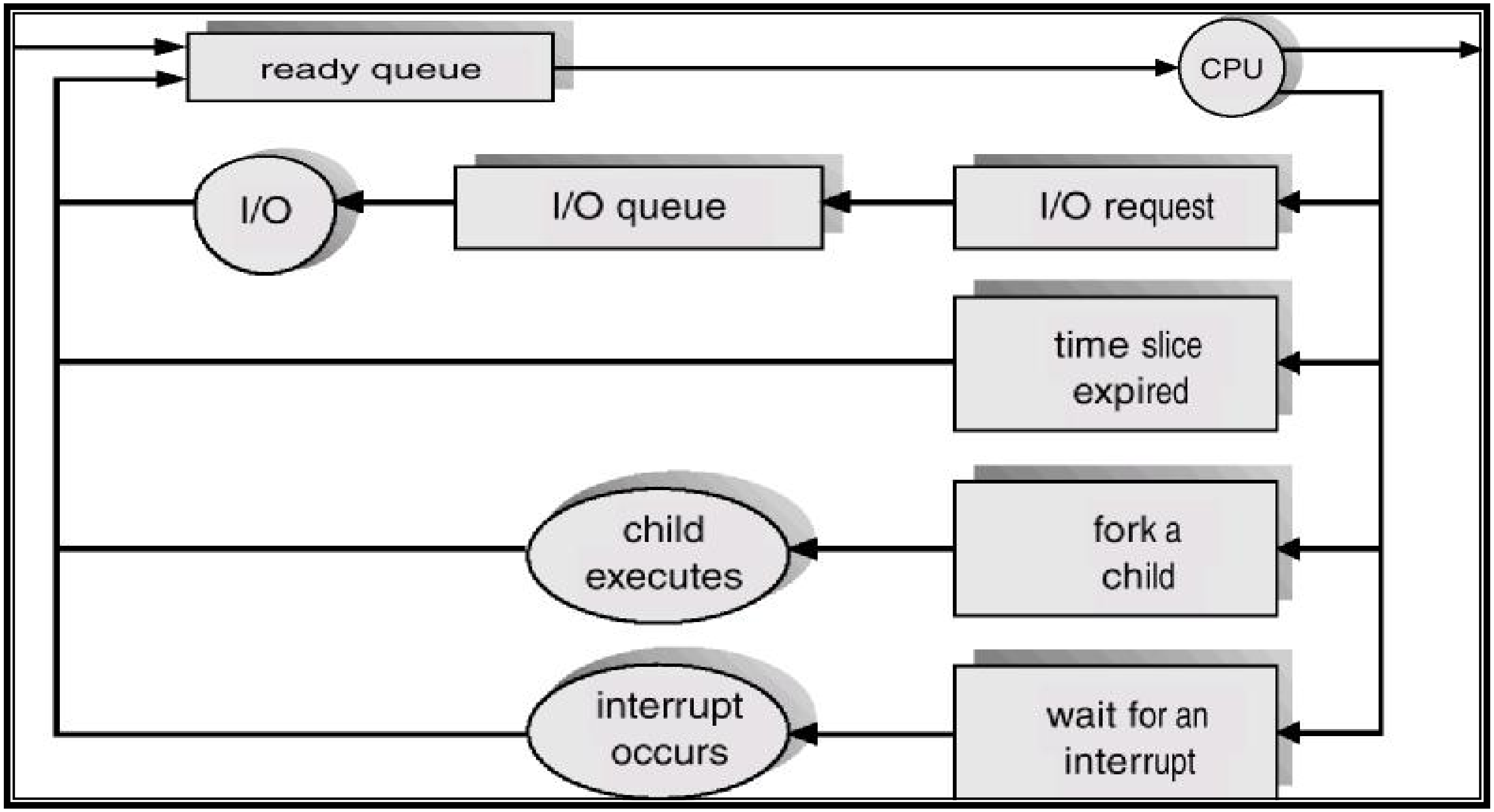
Ready Queue And Various I/O Device Queues



PCBs and Queues

- PCBs are data structures
  - dynamically allocated inside OS memory
- When a process is created:
  - OS allocates a PCB for it
  - OS initializes PCB
  - OS puts PCB on the correct queue
- As a process computes:
  - OS moves its PCB from queue to queue
- When a process is terminated:
  - OS deallocates its PCB

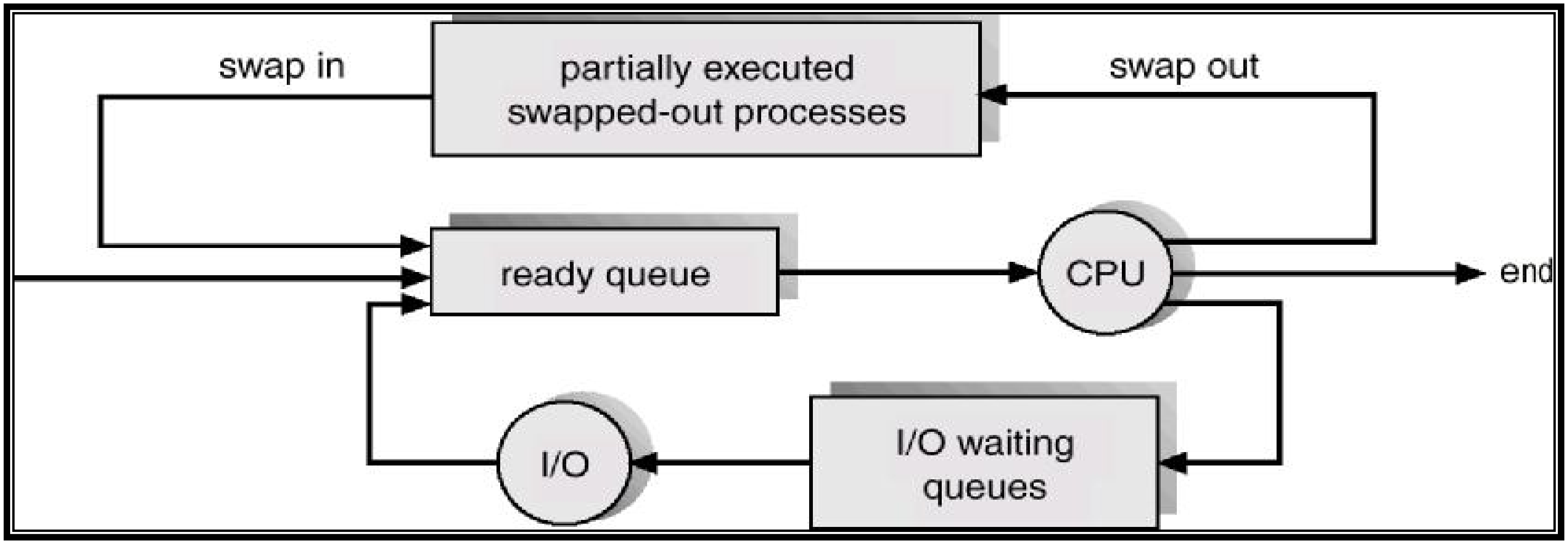
Representation of Process Scheduling



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

Addition of Medium Term Scheduling



Schedulers (Cont'd)

- 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

Long-term Scheduler

- Job scheduler
  - Selects which processes should be brought into the ready queue
  - Controls the degree of multiprogramming
  - Should select a good mix of I/O-bound and CPU-bound processes
  - Time-sharing systems such as UNIX often has no long-term scheduler
    - Simply put every new process in memory
    - Depends either on a physical limitation or on the self-adjusting nature of human users

Short-term Scheduler

- CPU scheduler
  - Selects which process should be executed next and allocates CPU
  - Should be fast !
  - Scheduling criteria:
    - CPU utilization
    - Throughput
    - Turnaround time
    - Waiting time
    - Response time

Medium-term Scheduler

- Swapper
  - Removes processes from memory temporarily
  - Reduces the degree of multiprogramming
  - Can improve the process mix dynamically
  - Swapping is originally proposed to reduce the memory pressure

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

- Context-switch time is overhead
  - the system does no useful work while switching

- Time dependent on hardware support

Context Switch

- The act of switching the CPU from one process to another

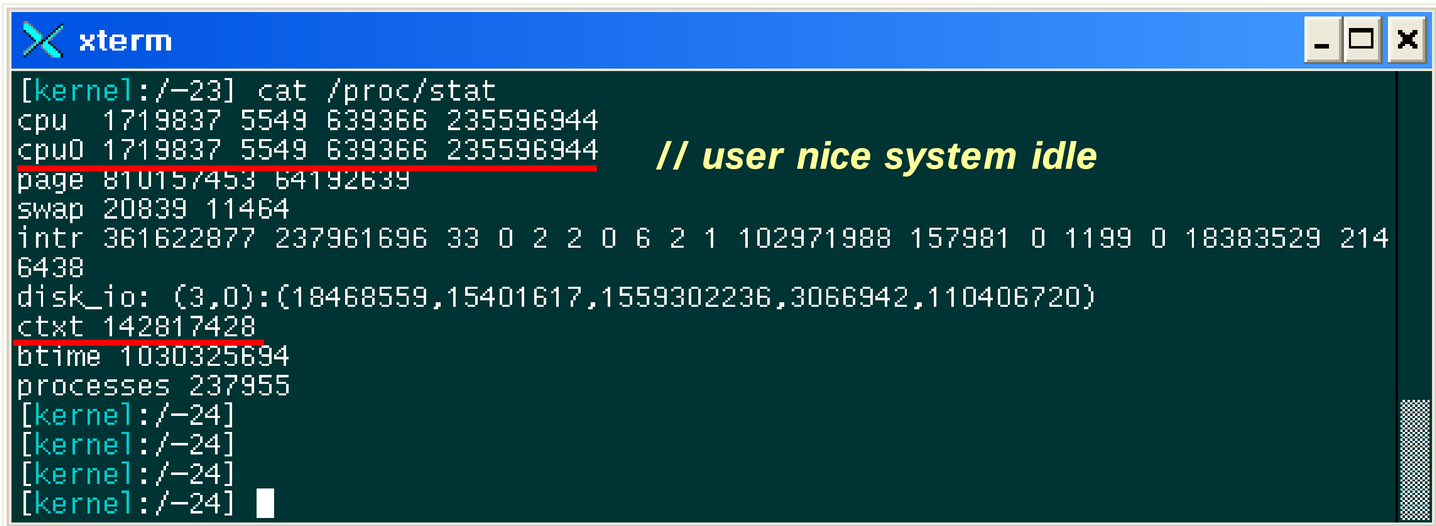
- Administrative overhead
  - saving and loading registers and memory maps
  - flushing and reloading the memory cache
  - updating various tables and lists, etc.

- Context switch overhead is dependent on hardware support
  - Multiple register sets in UltraSPARC
  - Advanced memory management techniques may require extra data to be switched with each context

- 100s or 1000s of switches/s typically

Context Switch (Cont'd)

- Linux example
  - Total 237,961,696 ticks = 661 hours = 27.5 days
  - Total 142,817,428 context switches
  - Roughly 60 context switches / sec



```
[kernel:/-23] cat /proc/stat
cpu  1719837 5549 639366 235596944
cpu0 1719837 5549 639366 235596944
page 810157453 64192639
swap 20839 11464
intr 361622877 237961696 33 0 2 2 0 6 2 1 102971988 157981 0 1199 0 18383529 214
6438
disk_io: (3,0):(18468559,15401617,1559302236,3066942,110406720)
ctxt 142817428
btime 1030325694
processes 237955
[kernel:/-24]
[kernel:/-24]
[kernel:/-24]
[kernel:/-24]
```

Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes

- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources

- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

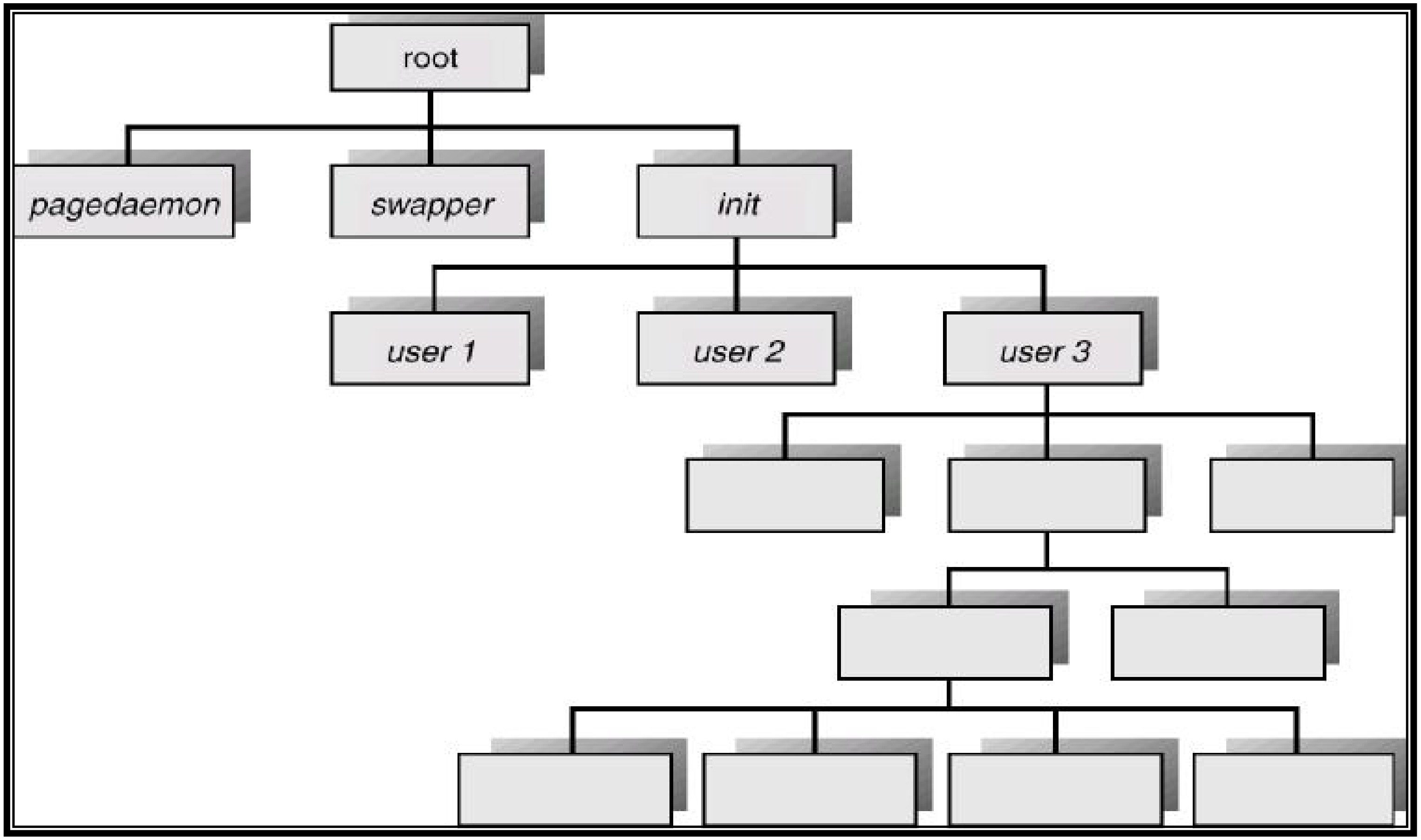
- Cf) Windows has no concept of process hierarchy

Process Creation (Cont'd)

- 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

Processes Tree on a UNIX System



fork()

```
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int pid;

    if ((pid = fork()) == 0)
        /* child */
        printf ("Child of %d is %d\n", getppid(), getpid());
    else
        /* parent */
        printf ("I am %d. My child is %d\n", getpid(), pid);
}
```

fork(): Example Output

```
% ./a.out
I am 31098. My child is 31099.
Child of 31098 is 31099.

% ./a.out
Child of 31100 is 31101.
I am 31100. My child is 31101.
```

Why fork()?

- Very useful when the child…
  - is cooperating with the parent
  - relies upon the parent's data to accomplish its task
  - Example: Web server

```
While (1) {
    int sock = accept();
    if ((pid = fork()) == 0) {
        /* Handle client request */
    } else {
        /* Close socket */
    }
}
```

Simplified UNIX Shell

```
int main()
{
    while (1) {
        char *cmd = read_command();
        int pid;
        if ((pid = fork()) == 0) {
            /* Manipulate stdin/stdout/stderr for
               pipes and redirections, etc. */
            exec(cmd);
            panic("exec failed!");
        } else {
            wait (pid);
        }
    }
}
```

Process Creation: UNIX

int fork()

- fork()
  - Creates and initializes a new PCB
  - Creates and initializes a new address space
  - Initializes the address space with a copy of the entire contents of the address space of the parent
  - Initializes the kernel resources to point to the resources used by parent (e.g., open files)
  - Places the PCB on the ready queue
  - Returns the child’s PID to the parent, and zero to the child

Process Creation: UNIX (Cont'd)

int exec (char *prog, char *argv[])

- exec()
  - Stops the current process
  - Loads the program "prog" into the process' address space
  - Initializes hardware context and args for the new program
  - Places the PCB on the ready queue
    - Note: exec() does not create a new process
  - What does it mean for exec() to return?

Process Creation: NT

BOOL CreateProcess (char *prog, char *args, ...)

- CreateProcess()
  - ü Creates and initializes a new PCB
  - ü Creates and initializes a new address space
  - ü Loads the program specified by “prog” into the address space
  - ü Copies “args” into memory allocated in address space
  - ü Initializes the hardware context to start execution at main
  - ü Places the PCB on the ready queue

Process Termination

- Process executes last statement and asks the operating system to decide it (**exit**)
  - Output data 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
  - Parent is exiting
    - Operating system does not allow child to continue if its parent terminates
    - Cascading termination

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
  - *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;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer Process

```
item nextProduced;

while (1) {
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded-Buffer – Consumer Process

```
item nextConsumed;

while (1) {
        while (in == out)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
}
```

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?

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**
- **Non-blocking** is considered **asynchronous**
- **send** and **receive** primitives may be either blocking or non-blocking

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

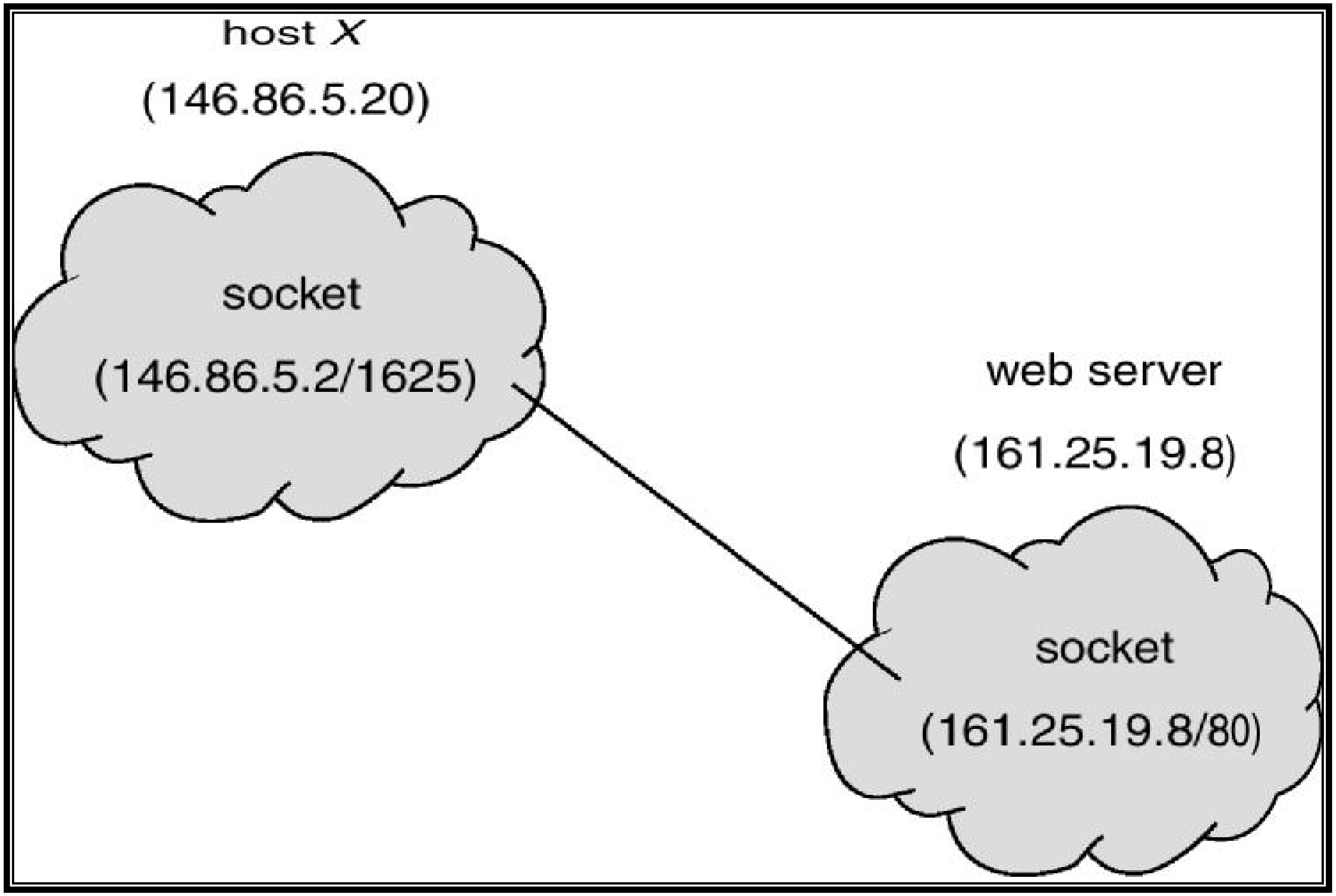
Client-Server Communication

- 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

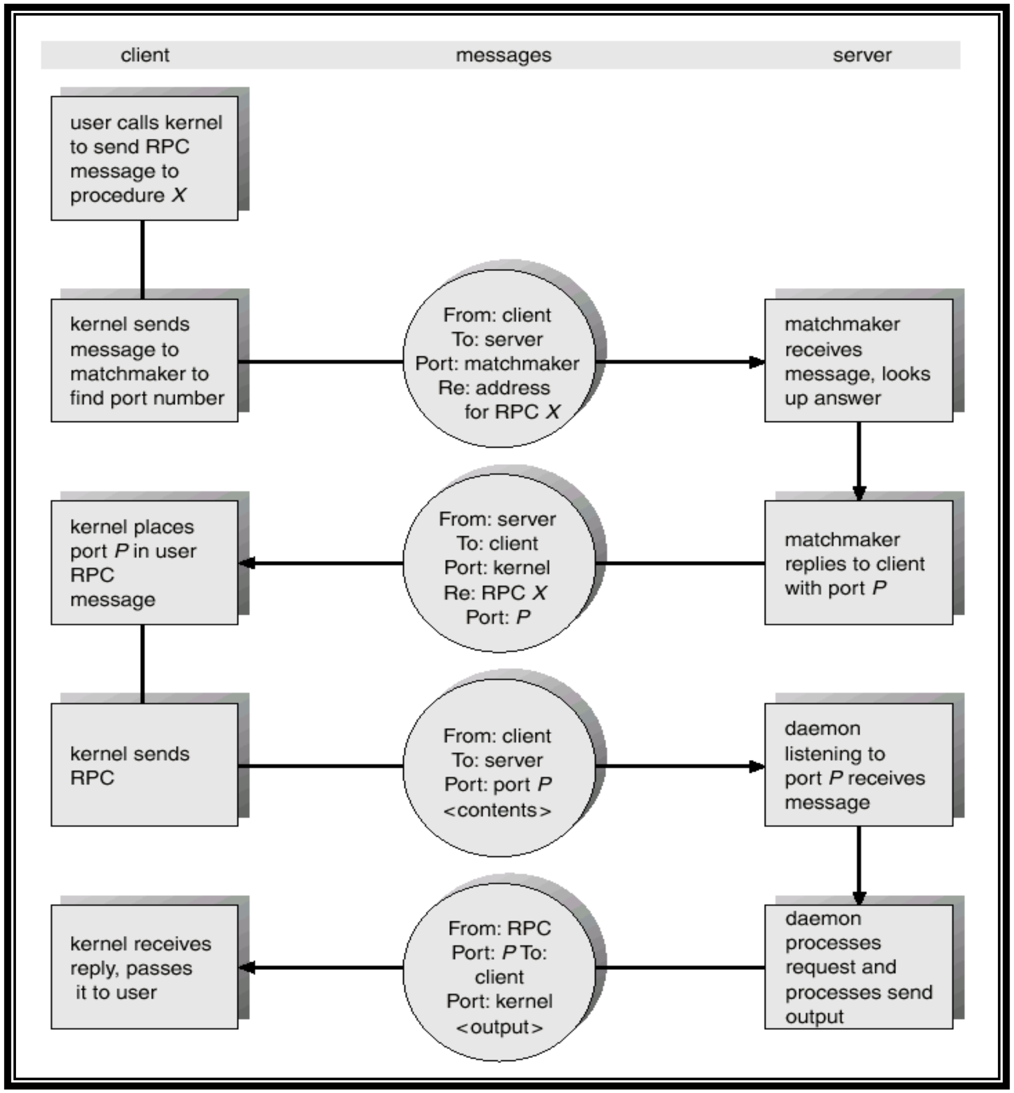
Socket Communication



Remote Procedure Calls

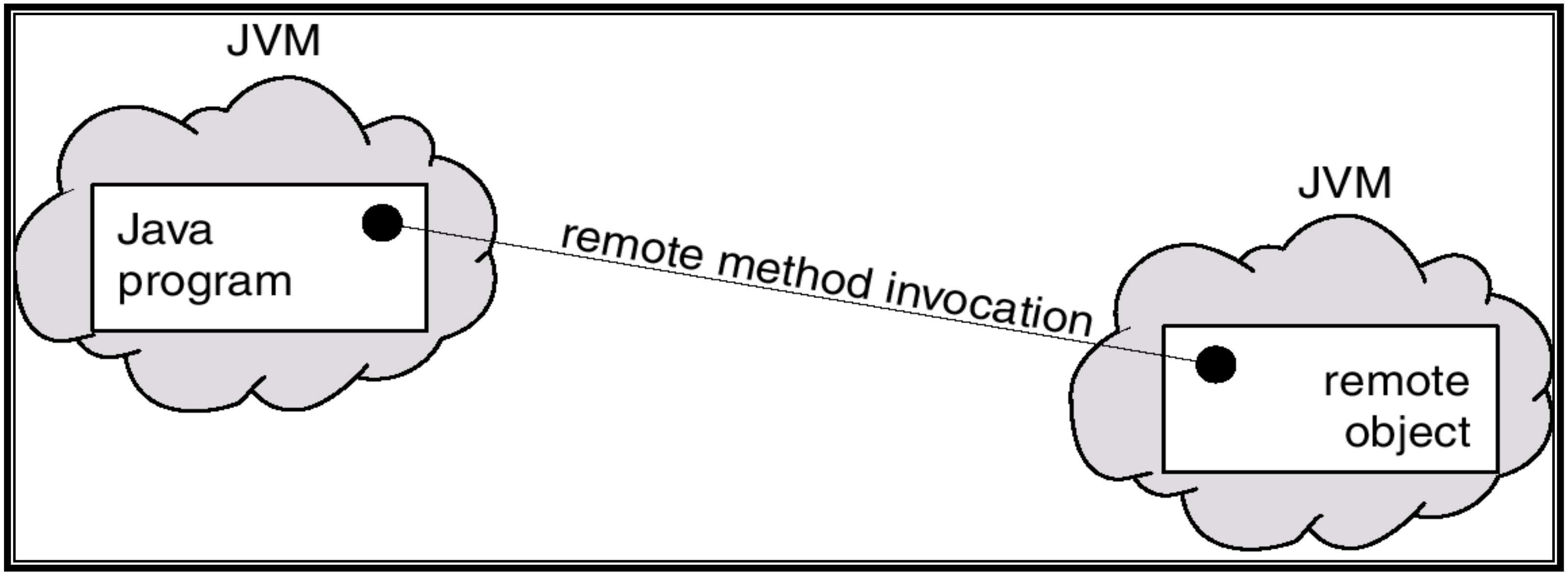
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- **Stubs**
  - client-side proxy for the actual procedure on the server
- The client-side stub locates the server and *marshalls* the parameters
- The server-side stub receives this message, unpacks the marshalled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