# Operating System

Ch03: Processes

#### BeomSeok Kim

Department of Computer Engineering KyungHee University passion0822@khu.ac.kr

### Program vs. Process

#### Program

- ✓ Executable file on a disk
- ✓ Loaded into memory and executed by the kernel

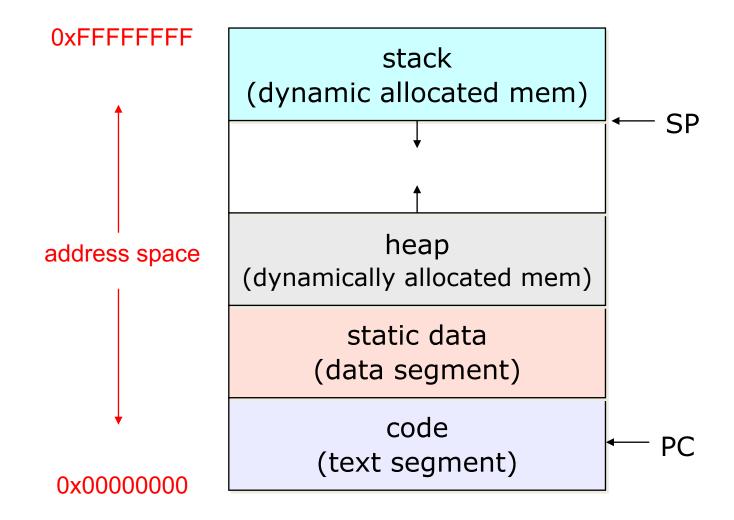
#### Process

- ✓ Executing instance of a program
- ✓ The basic unit of execution and scheduling
- ✓ A process is named using its process ID (PID)

### **Process Concept**

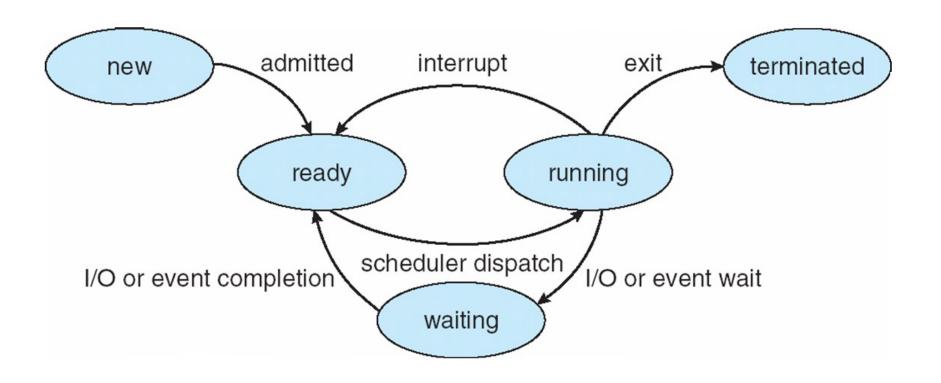
- What is the process?
  - ✓ An instance of a program in execution
  - ✓ An encapsulation of the flow of control in a program
  - ✓ A dynamic and active entity
  - ✓ The basic unit of execution and scheduling
  - ✓ A process is named using its process ID (PID)

### Process Address Space



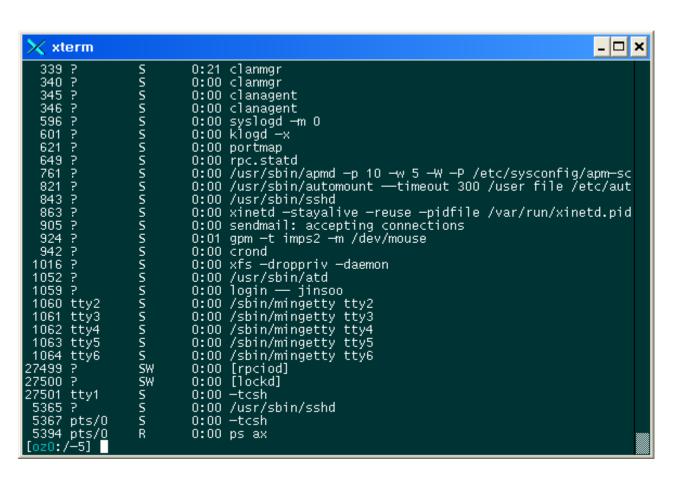
### **Process State**

State diagram



#### **Process State Transition**

Linux example



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)**

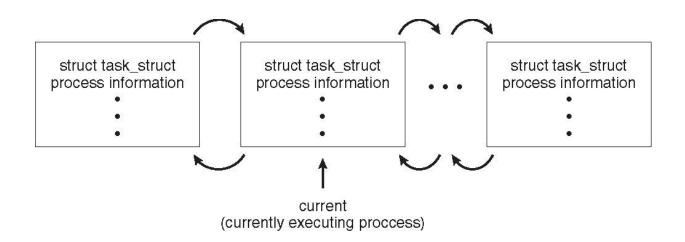
- Information associated with each process
  - Process state
  - ✓ Program counter
  - ✓ CPU registers
  - ✓ CPU scheduling information
  - Memory-management information
  - ✓ Accounting information
  - √ I/O status information
- Cf) task struct in Linux
  - √ 1456 bytes as of Linux 2.4.18

process state process number program counter registers memory limits list of open files

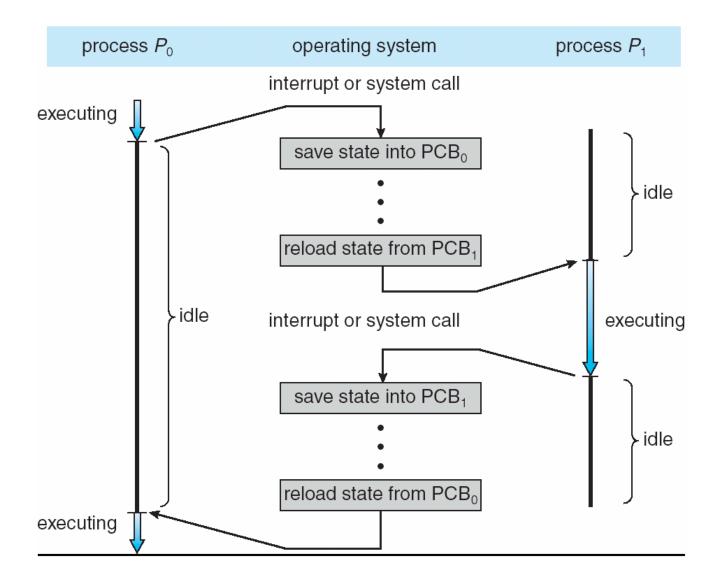
### **Process Control Block (PCB)**

#### Represented by the C structure task\_struct

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



### **Context Switch (CPU Switch)**



#### **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

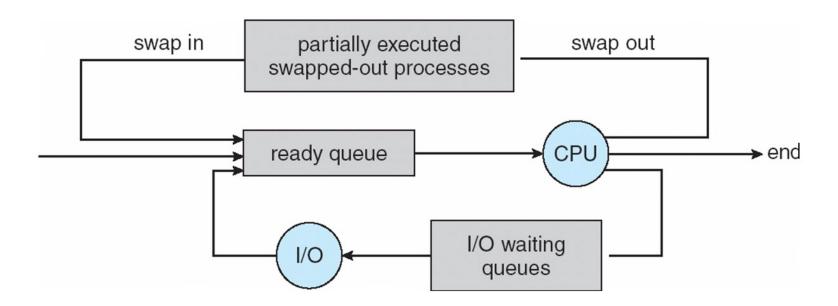
#### **Context Switch**



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

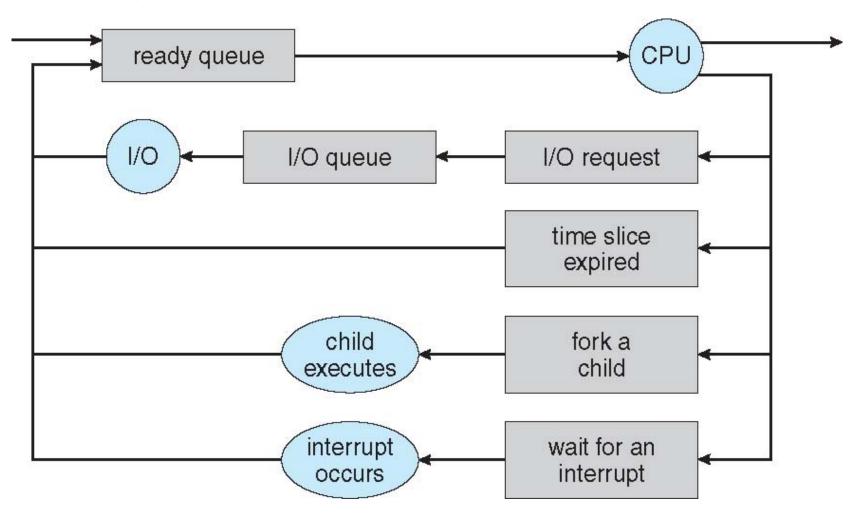
### **Schedulers**

- 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
- Medium-term scheduler (or swapper)



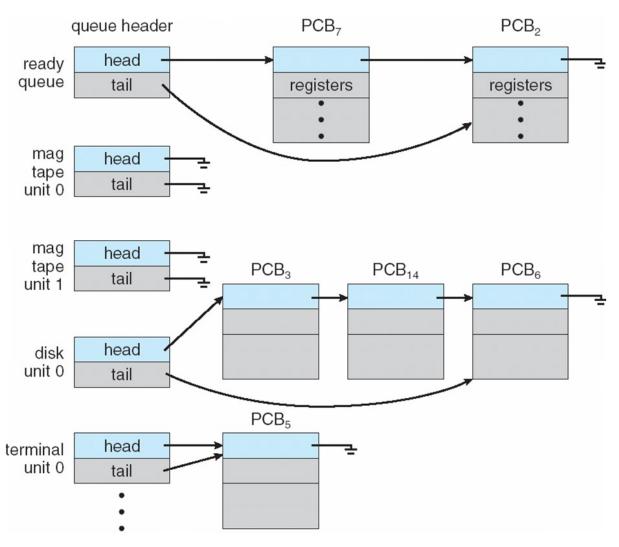
### Representation of Process Scheduling

Queueing diagram



### Representation of Process Scheduling

Ready queue and various I/O device queues



### **Operations on Processes**

- Process creation
  - ✓ fork()
- Process execution
  - ✓ exec()
- Process termination
  - ✓ exit()
  - ✓ \_exit()
  - √ abort()
  - ✓ wait()
- Cooperating processes
  - ✓ Inter-Process Communication (IPC)

### **Process Creation: Unix/Linux**

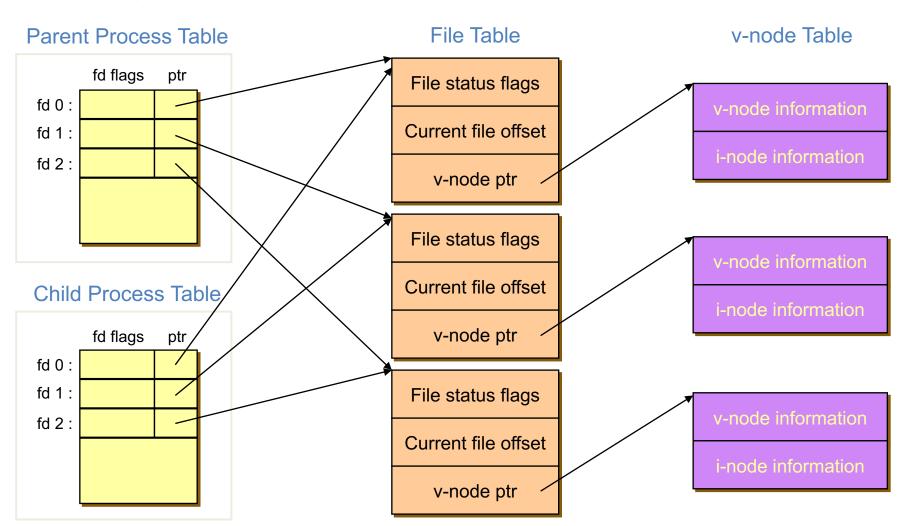
#### int fork()

#### **■** 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/Linux**

Sharing of open files between parent and child after fork



### 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()?

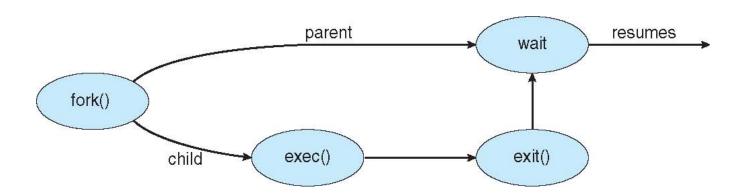
- Very useful when the child...
  - √ is cooperating with the parent
  - ✓ relies upon the parent's data to accomplish its task
  - ✓ Example: Web server

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

### **Process Execution: Unix/Linux**

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

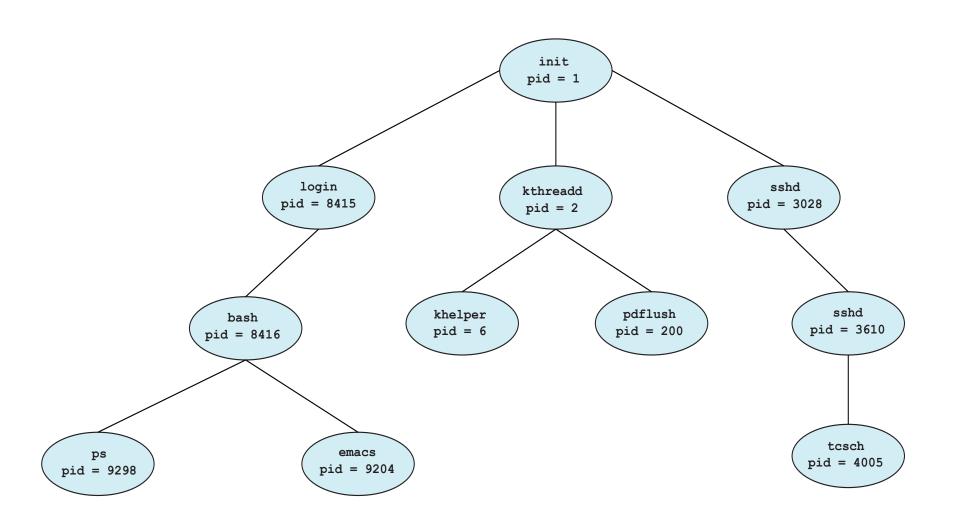
- exec()
  - ✓ Stops the current process
  - ✓ Loads the program "prog" into the process' address space
  - ✓ Initializes hardware context and args for the new program
  - ✓ Places the PCB on the ready queue
    - Note: exec() does not create a new process
  - ✓ What does it mean for exec() to return?



### Simplified Unix/Linux 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);
```

### **A Process Tree in Linux**



#### **Process Creation/Execution: Windows**

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

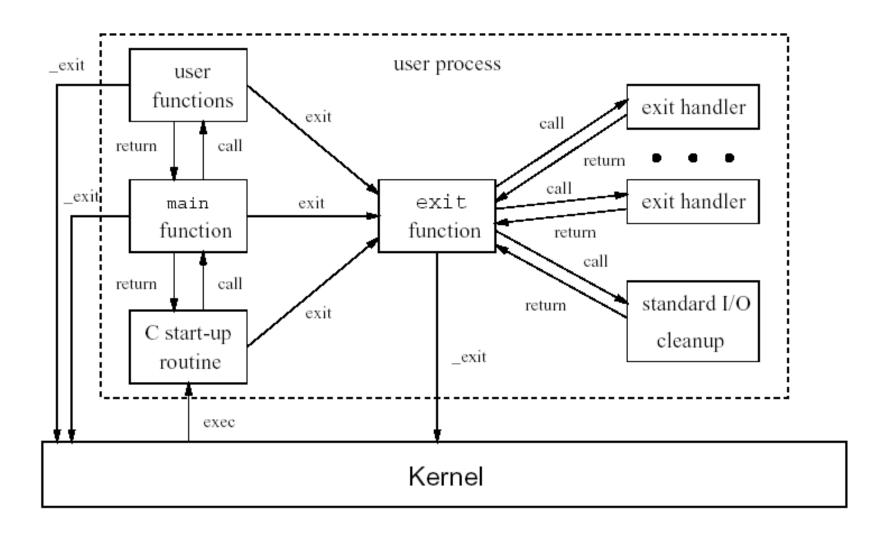
#### CreateProcess()

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

### **Process Termination**

- Normal termination
  - ✓ return from main()
  - ✓ calling exit()
  - ✓ calling \_exit()
- Abnormal termination
  - ✓ calling abort()
  - √ terminated by a signal
- Wait for termination of a child process
  - ✓ calling wait()
  - ✓ If no parent waiting (did not invoke wait()), process is a zombie
  - ✓ If parent terminated without invoking wait, process is an orphan

### **Start and Termination of a C Program**



### **Multiprocess in Application Program**

- Google Chrome Browser is multiprocess with 3 different types of processes:
  - ✓ Browser process manages user interface, disk and network I/O
  - ✓ Renderer process renders web pages, deals with HTML, Javascript
    - > A new renderer created for each website opened
    - > Runs in sandbox restricting disk and network I/O, minimizing effect of security exploits
  - ✓ Plug-in process for each type of plug-in

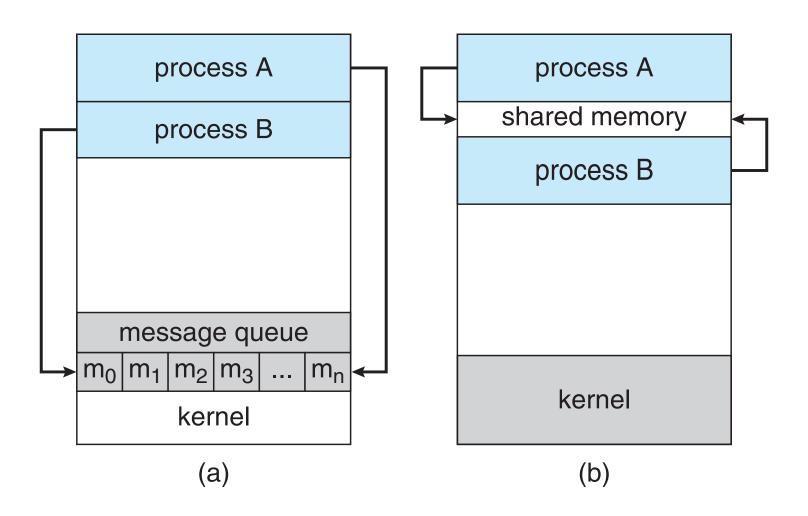


### Multiprocess in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - ✓ Single foreground process- controlled via user interface
  - ✓ Multiple background processes— in memory, running, but not on the display, and with limits
  - ✓ Limits include single, short task, receiving notification of events, specific longrunning tasks like audio playback
- Android runs foreground and background, with fewer limits
  - ✓ Background process uses a service to perform tasks
  - ✓ Service can keep running even if background process is suspended
  - ✓ Service has no user interface, small memory use

### Inter-Process Communication (IPC)

- Communication models
  - ✓ (a) message passing vs. (b) shared memory



### **Inter-Process Communication (IPC)**

- Cooperating processes
  - ✓ Example: Bounded buffer problem (Producer-Consumer problem)

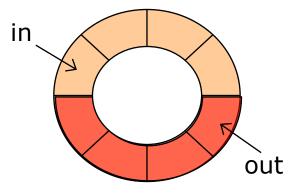
#### Producer

```
void producer(data)
{

while (count==N);
buffer[in] = data;
in = (in+1) % N;
count++;
}
```

#### int count;

struct item buffer[N];
 int in, out;



#### Consumer

```
void consumer(data)
{

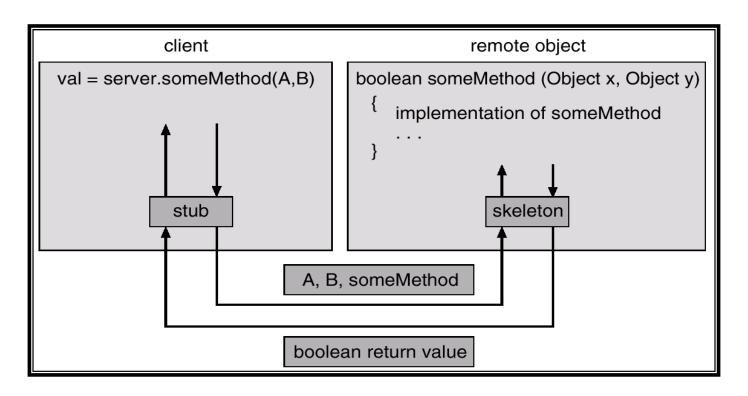
while (count==0);
data = buffer[out];
out = (out+1) % N;
count--;
}
```

### **Inter-Process Communication (IPC)**

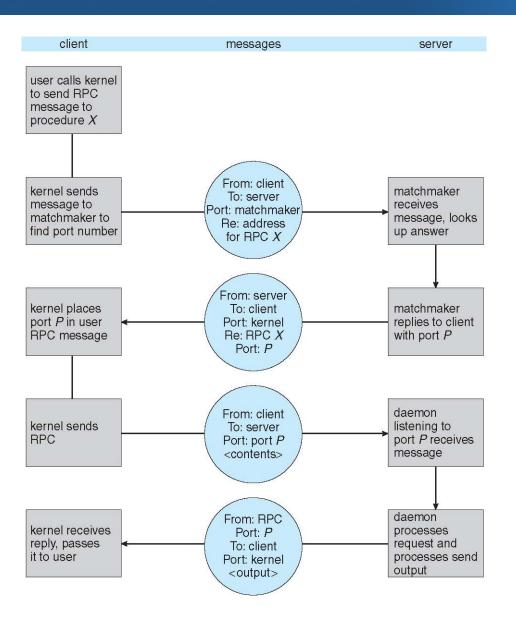
- Unix/Linux IPC
  - ✓ pipes
  - √ FIFOs
  - √ message queue
  - √ shared memory
  - √ sockets

#### **Client-Server Communication**

- Sockets
- Remote Procedure Call (RPC)
- Remote Method Invocation (RMI in Java)
- Marshalling parameters



### **Execution of RPC**



## Thank You! Q&A