BILKENT UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING
CS342 OPERATING SYSTEMS

LECTURE 3
PROCESSES

(CHAPTER 3)

Dr. İbrahim Körpeoğlu
http://www.cs.bilkent.edu.tr/~korpe

References

- *The slides here are adapted/modified from the textbook and its slides: Operating System Concepts, Silberschatz et al., 7th & 8th editions, Wiley.*

REFERENCES

- Operating System Concepts, 7th and 8th editions, Silberschatz et al. Wiley.
- Modern Operating Systems, Andrew S. Tanenbaum, 3rd edition, 2009.

Outline

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

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

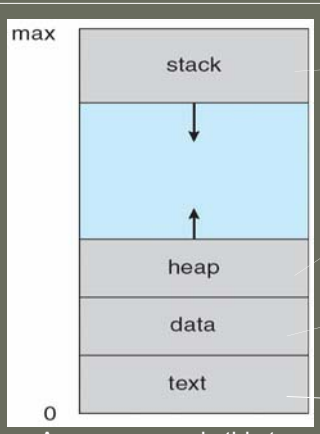
Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- We will use the terms *job* and *process* almost interchangeably

- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - text – code – section (program counter – PC)
  - stack section (stack pointer)
  - data section

  - set of open files currently used
  - set of I/O devices currently used

Process in Memory



A process needs this to be in memory
(address space; memory image)

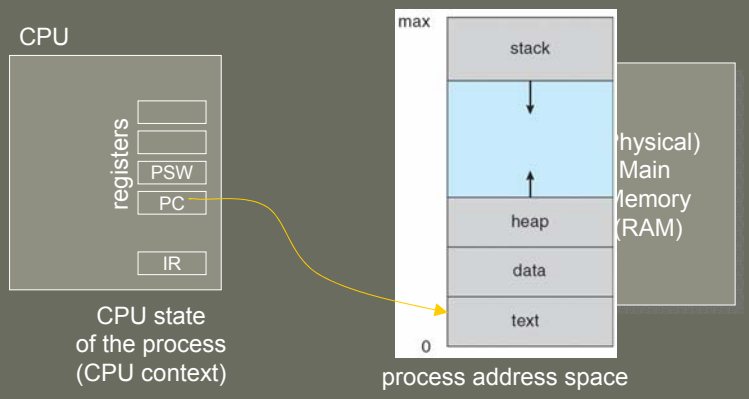
Stack segment
(holds the called function parameters, local variables)

Storage for dynamically allocated variables

Data segment
(includes global variables, arrays, etc., you use)

Text segment
(code segment)
(instructions are here)

Process: program in execution

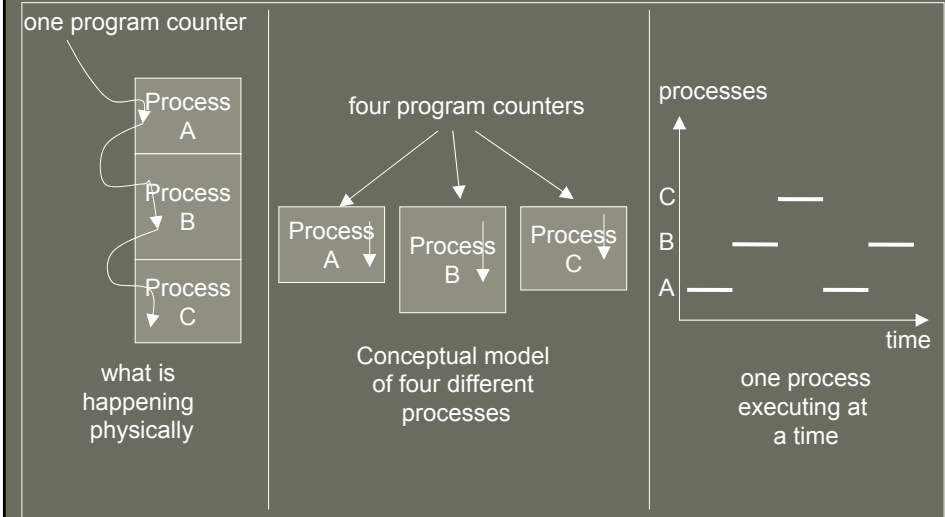


(currently used portion of the address space must be in memory)

Process: program in execution

- If we have a single program running in the system, then the task of OS is easy:
  - load the program, start it and program runs in CPU
  - (from time to time it calls OS to get some service done)
- But if we want to start several processes, then the running program in CPU (current process) has to be stopped for a while and other program (process) has to run in CPU.
- To do this switch, we have to save the state/context (register values) of the CPU which belongs to the stopped program, so that later the stopped program can be re-started again as if nothing has happened.

Multiple Processes

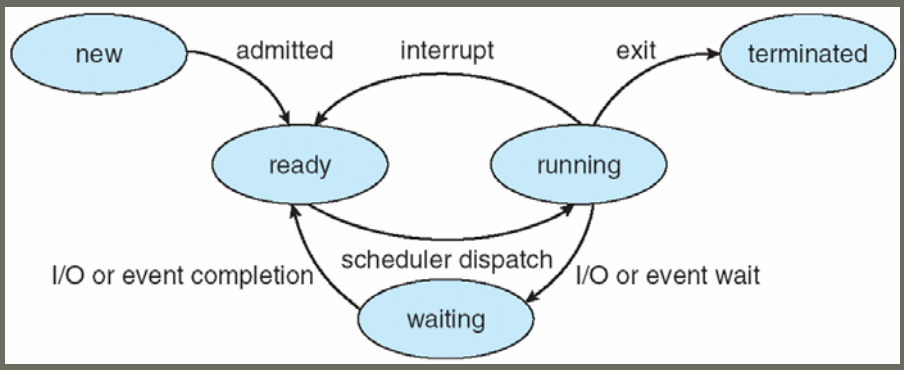


Process State

- 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

In a single-CPU system, only one process may be in running state; many processes may be in ready and waiting states.

Diagram of Process State



Process Control Block

Information associated with each process

- Process state (ready, running, waiting, etc)
- Program counter (PC)
- CPU registers
- CPU scheduling information
  - Priority of the process, etc.
- Memory-management information
  - text/data/stack section pointers, sizes, etc.
  - pointer to page table, etc.
- Accounting information
  - CPU usage, clock time so far, …
- I/O status information
  - List of I/O devices allocated to the process, a list of open files, etc.

Process Control Block (PCB)

Process management
Registers
Program Counter (PC)
Program status word (PSW)
Stack pointer
Process state
Priority
Scheduling parameters
Process ID
Parent Process
Time when process started
CPU time used
Children's CPU time

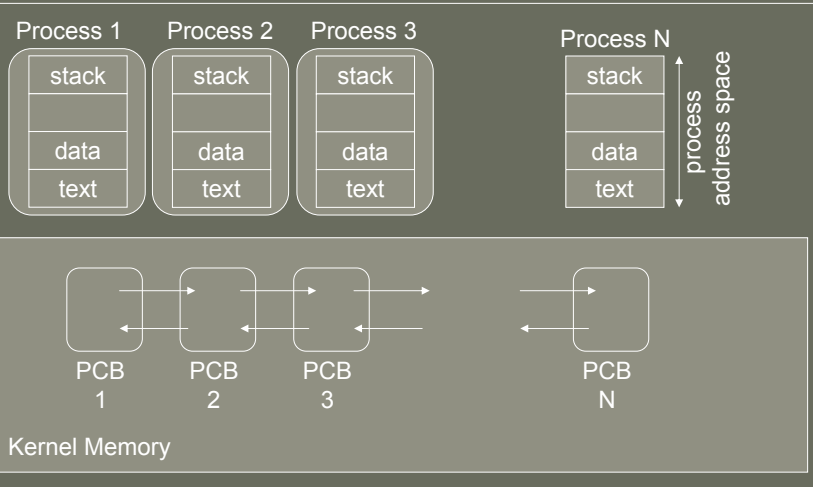
Memory management
Pointer to text segment info
Pointer to data segment info
Pointer to stack segment info

File management
Root directory
Working directory
File descriptors
User ID
Group ID

……more

a PCB of a process may contain this information

PCBs



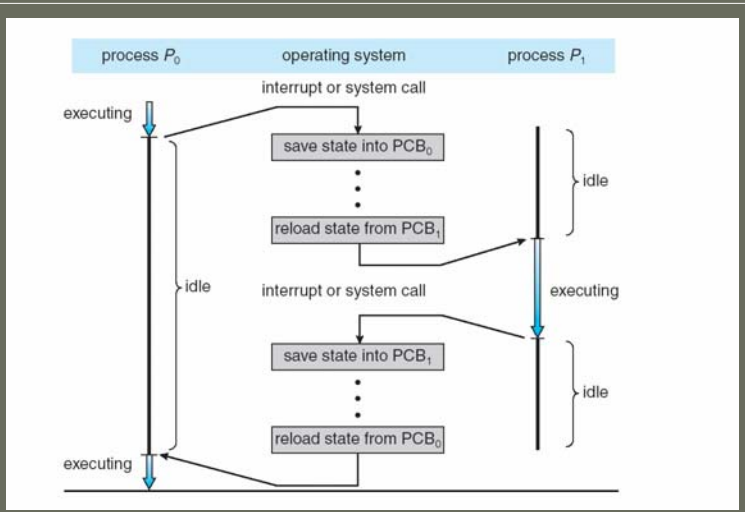
Kernel mains a PCB for each process. They can be linked together in various queues.

Process Representation in Linux

In Linux kernel source tree, the file **include/linux/sched.h** contains the definition of the structure **task_struct**, which is the PCB for a process.

```
struct task_struct {
        long state;                 /* state of the process */
        ….
        pid_t pid;                  /* identifier of the process */
        …
        unisgned int time_slice;    /* scheduling info */
        …
        struct files_struct *files;     /* info about open files */
        ….
        struct mm_struct *mm;     /* info about the address space of this process */
        …
}
```


CPU Switch from Process to Process

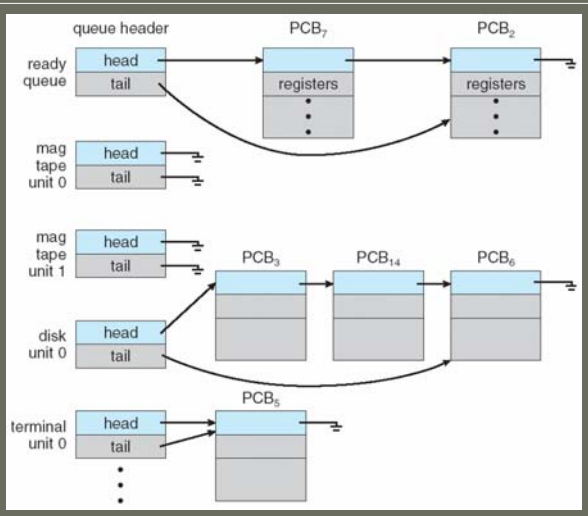


Process Scheduling Queues

- **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

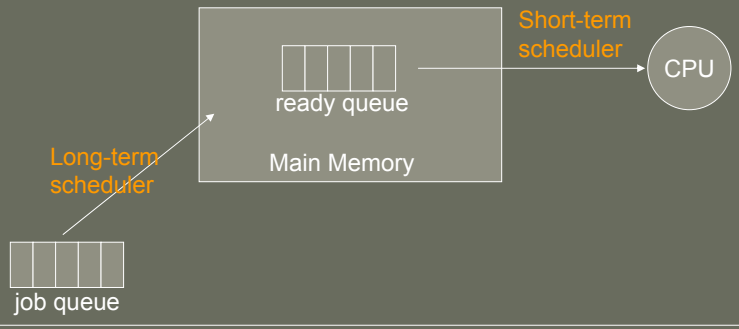
- Processes migrate among the various queues

Ready Queue And Various I/O Device Queues

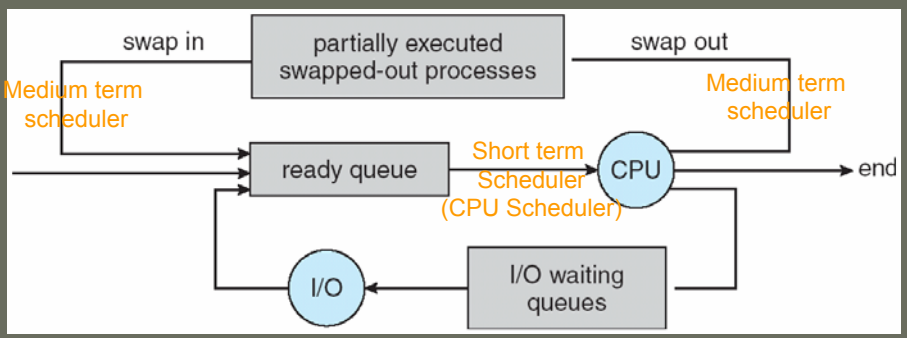


Schedulers

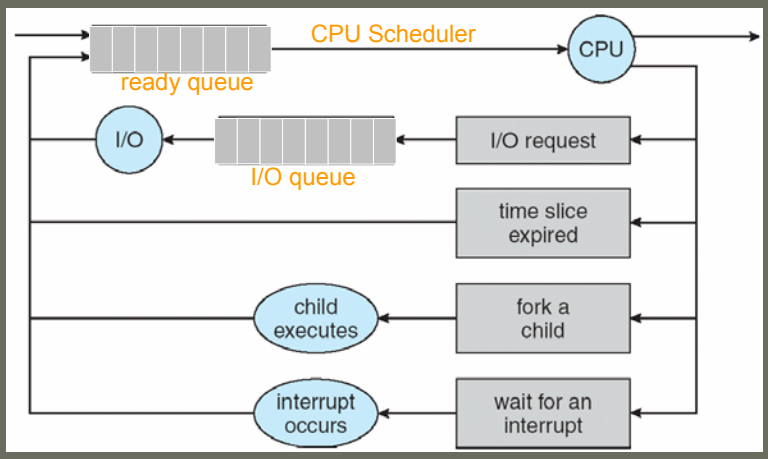
- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU



Addition of Medium Term Scheduling



Representation of Process Scheduling



Schedulers (Cont)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (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

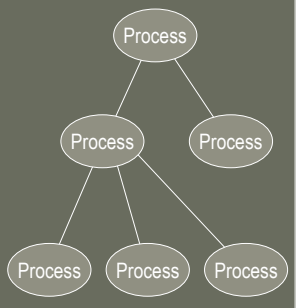
    - CPU burst: a time period during which the process wants to continuously run in the CPU without making I/O

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 represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

Process Creation

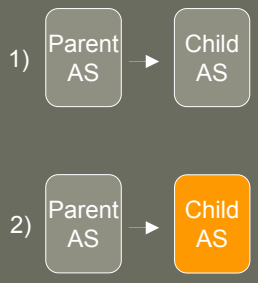
- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **a process identifier** (**pid**)
- Resource sharing alternatives:
  - Parent and children share all resources
  - Children share subset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



Process Creation (Cont)

- Child's address space?

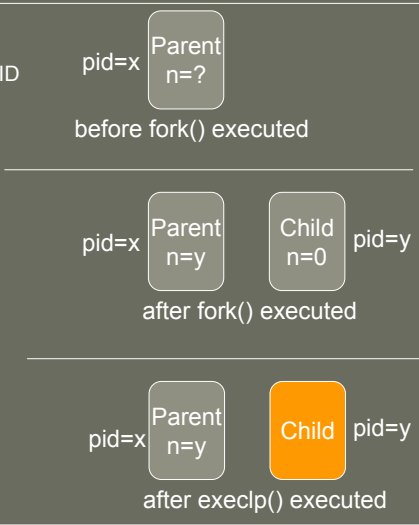
  Child has a new address space.
  Child's address space can contain:
  - 1) the copy of the parent (at creation)
  - 2) has a new program loaded into it



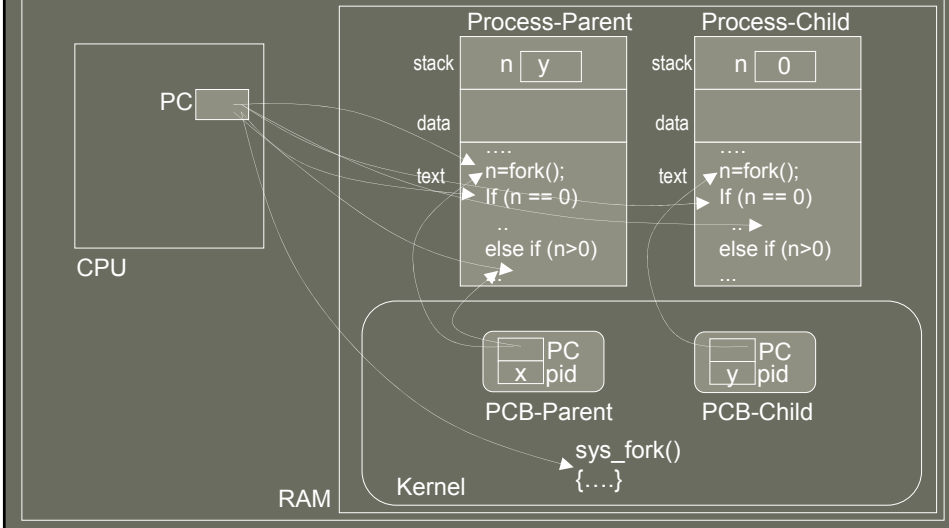
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

C Program Forking Separate Process

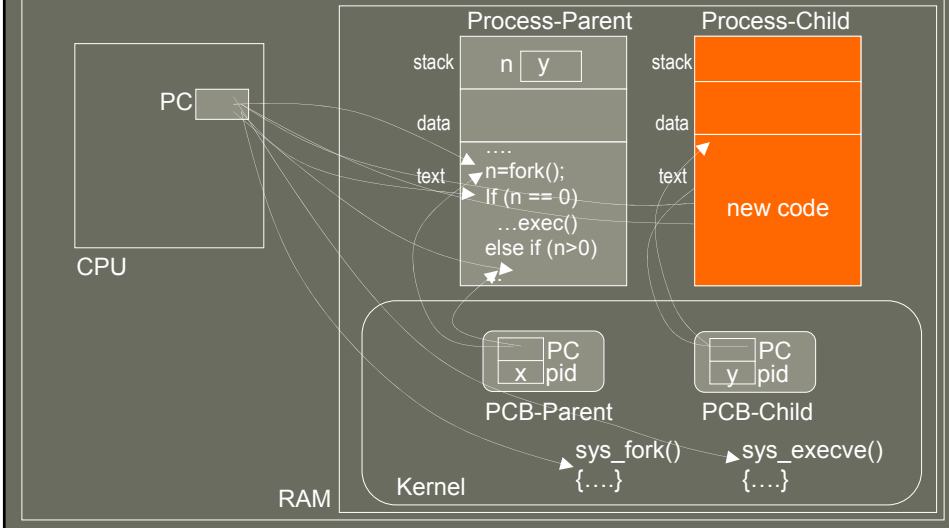
```
int main()
{
    pid_t  n; // return value of fork; it is process ID
    /* fork another process */
    n = fork();
    if (n < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (n == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
            /* parent will wait for the child
               to complete */
            wait (NULL);
            printf ("Child Complete");
            exit(0);
    }
}
```



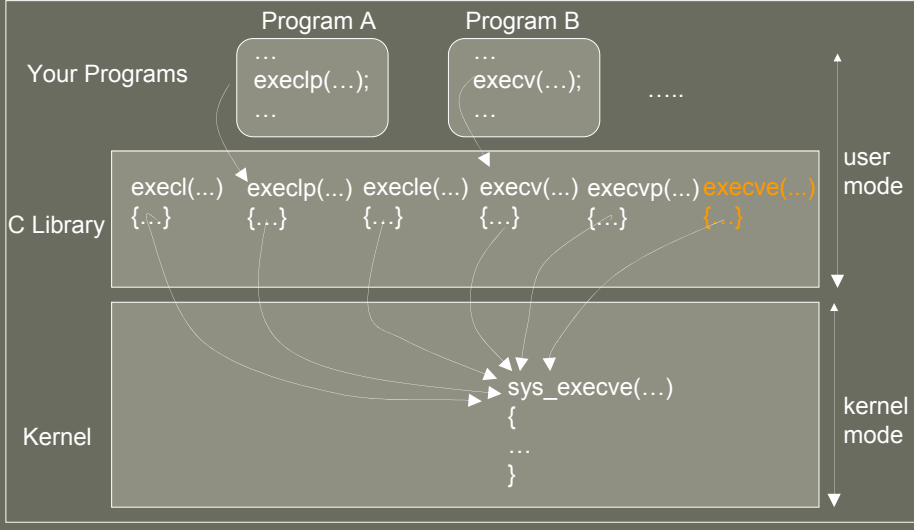
Execution Trace: fork()



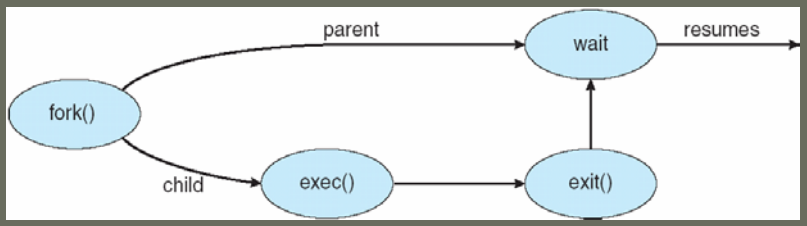
Execution Trace: fork() with execlp()



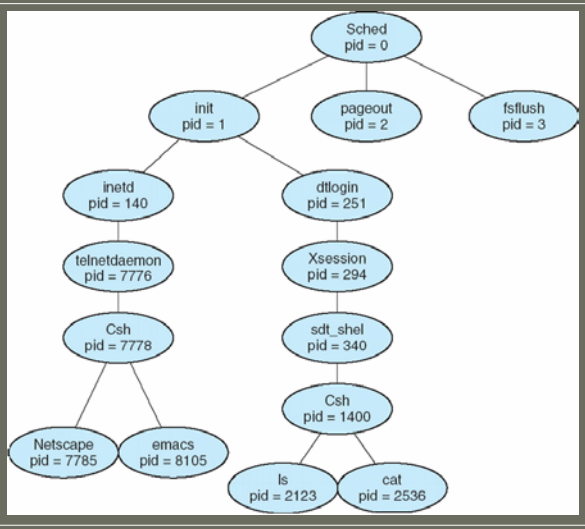
Family of exec() Functions in Unix



Process Creation



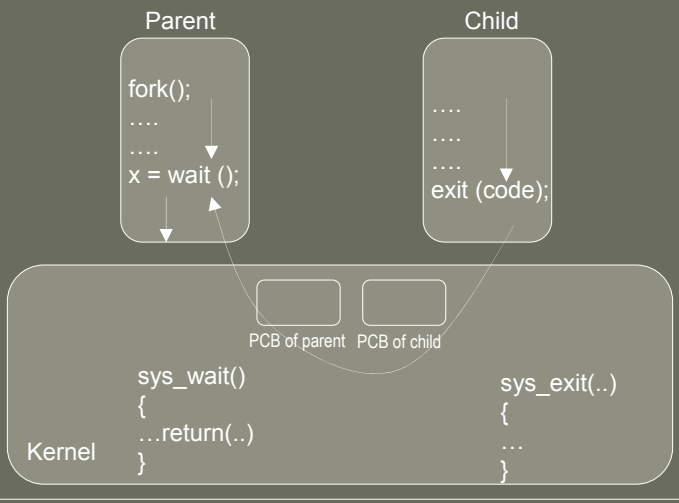
A tree of processes on a typical Solaris



Process Termination

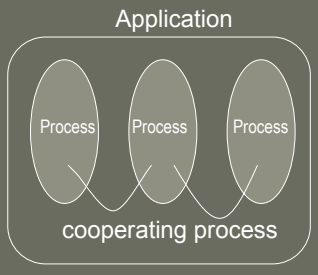
- Process executes last statement and asks the operating system to delete it (can use **exit** system call)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded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

Process Termination



Cooperating Processes

- Processes within a system may be **independent** or **cooperating**
- **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
- Reasons for process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity (application will be divided into modules/sub-tasks)
  - Convenience (may be better to work with multiple processes)



The overall application is designed to consist of cooperating processes

IPC Mechanisms

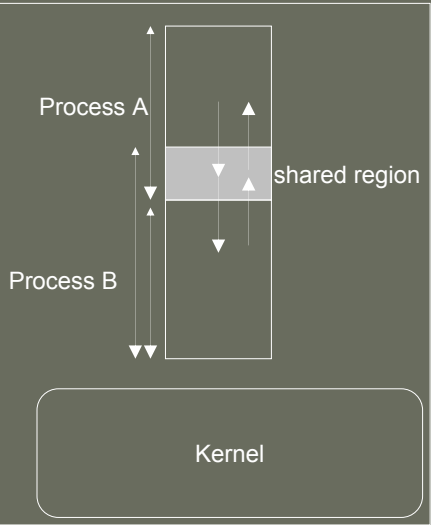
- Cooperating processes require a facility/mechanism for interprocess communication (IPC)
- There are two basic mechanism provided by most systems:

  1) Shared Memory

  2) Message Passing

Shared Memory IPC Mechanism

- A region of shared memory is established between (among) two or more processes.
- Establishment of that shared region is done via the help of the operating system kernel.
- Then, processes can read and write shared memory region (segment) directly as ordinary memory access (like they are accessing memory variables directly without kernel help)
  - During this time, kernel is not involved.
  - Hence it is fast



Shared Memory IPC Mechanism

- To illustrate use of the shared memory IPC mechanism, a general model problem, called producer-consumer problem, can be used.
- Producer-consumer problem: we have a producer, a consumer, and data is sent from producer to consumer.
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

We can solve this problem via shared memory IPC mechanism

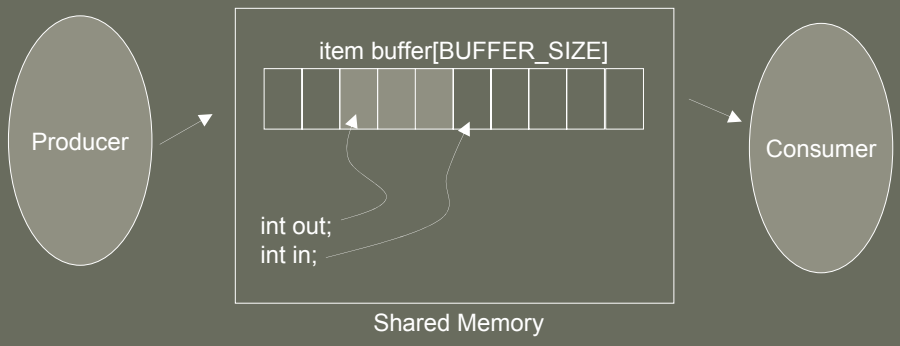
Bounded-Buffer – Shared-Memory Solution

- Shared data

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

item buffer[BUFFER_SIZE];
int in = 0;  // next free position
int out = 0; // first full position
```

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

Buffer State in Shared Memory

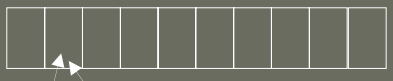


Buffer State in Shared Memory

Buffer Full

((in+1) % BUFFER_SIZE == out) : considered full buffer

Buffer Empty



In == out : empty buffer

Bounded-Buffer – Producer and Consumer Code

Producer

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

Consumer

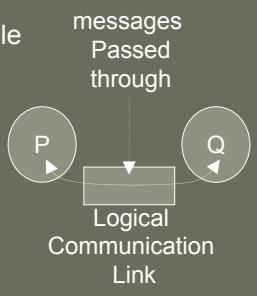
```
while (true) {
        while (in == out)
item buffer[BUFFER_SIZE]; ; // do nothing -- nothing to consume

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

```
int in = 0;
int out = 0;
```

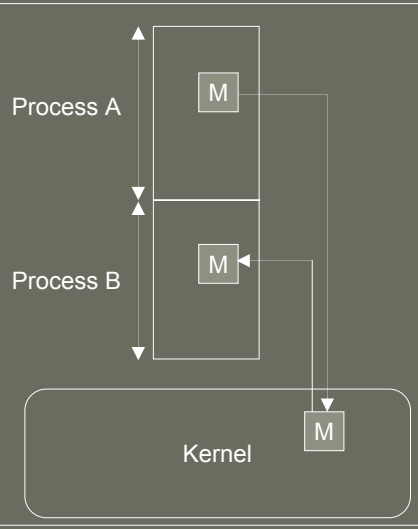
Shared Memory

Message Passing IPC Mechanism

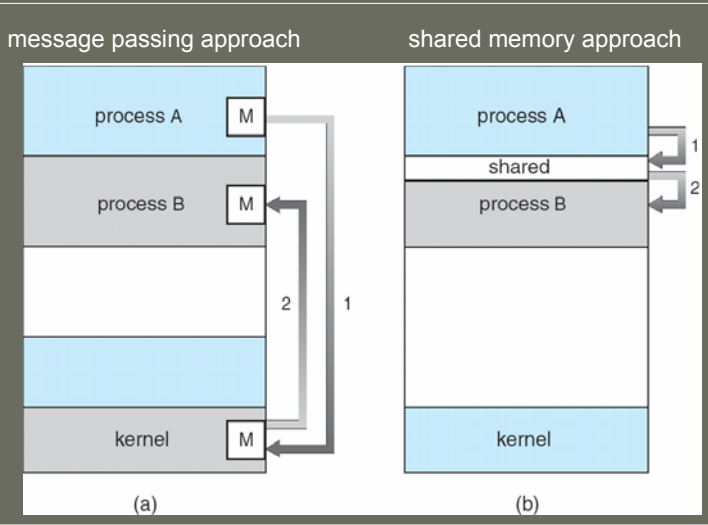
- Another mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- This IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a (logical) *communication link* between them
  - exchange messages via send/receive



Message Passing



Communication Models



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?

Issues to Consider

- Naming
  - Direct
  - Indirect
- Synchronization
  - Blocking send/receive
  - Non-blocking send/receive
- Buffering
  - Zero capacity
  - Bounded capacity
  - Unbounded capacity

Direct Communication

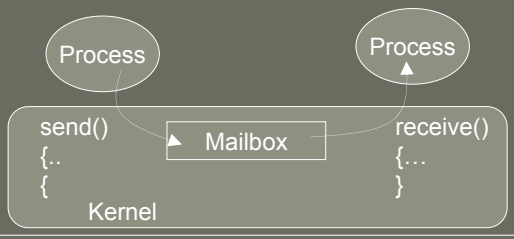
- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically (i.e. implicitly by the kernel)
  - A link is associated with 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

Examples of IPC Systems –Unix/Linux Shared Memory

- There are two different API that provide functions for shared memory I in Unix/Linux operating system
  - 1) System V API
    - System V is one of the earlier Unix versions that introduced shared memory
  - 2) POSIX API
    - POSIX (Portable Operating System Interface) is the standard API for Unix like systems.

Examples of IPC Systems – Unix System V Shared Memory

- System V Shared Memory
  - Process first creates shared memory segment

```
segment id = shmget(IPC PRIVATE, size, S IRUSR | S
  IWUSR);
```

  - Process wanting access to that shared memory must attach to it

```
shared memory = (char *) shmat(id, NULL, 0);
```

  - Now the process could write to the shared memory

```
sprintf(shared memory, "Writing to shared memory");
```

  - When done a process can detach the shared memory from its address space

```
shmdt(shared memory);
```

Examples of IPC Systems – Unix POSIX Shared Memory

- The following functions are defined to create and manage shared memory in POSIX API
  - shm_open():
    - create or open a shared memory region/segment (also called shared memory object)
  - shm_unlink():
    - remove the shared memory object
  - ftruncate():
    - set the size of shared memory region
  - mmap():
    - map the shared memory into the address space of the process. With this a process gets a pointer to the shared memory region and can use that pointer to access the shared memory.

Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer

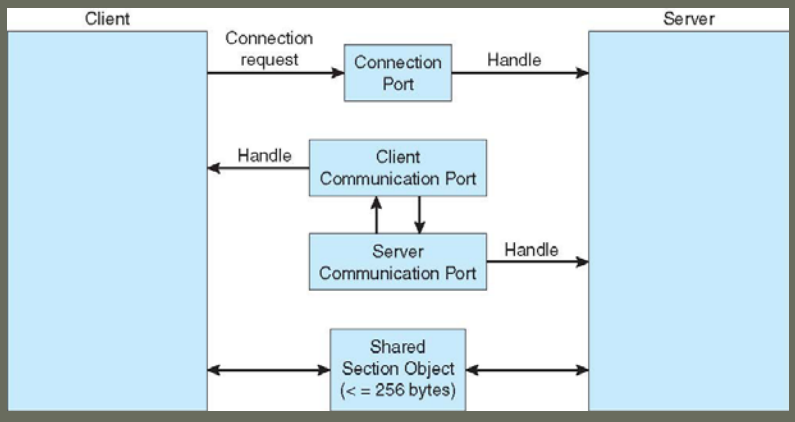
    `msg_send(), msg_receive(), msg_rpc()`
  - Mailboxes needed for commuication, created via

    `port_allocate()`

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



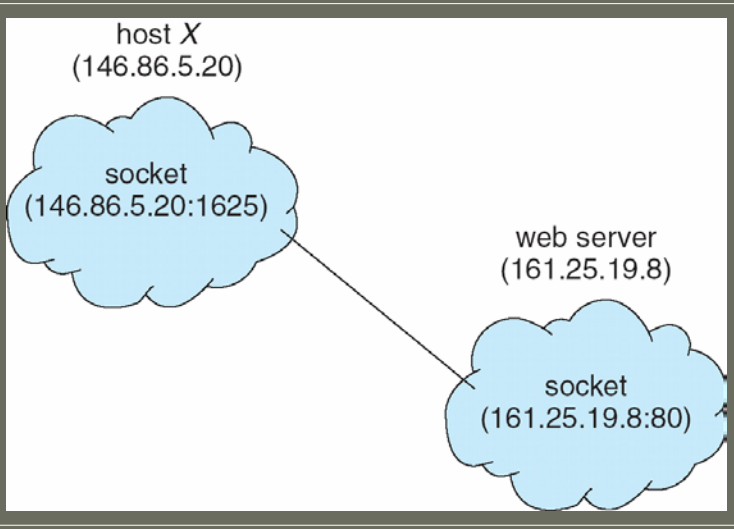
Communications in Client-Server Systems

- 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

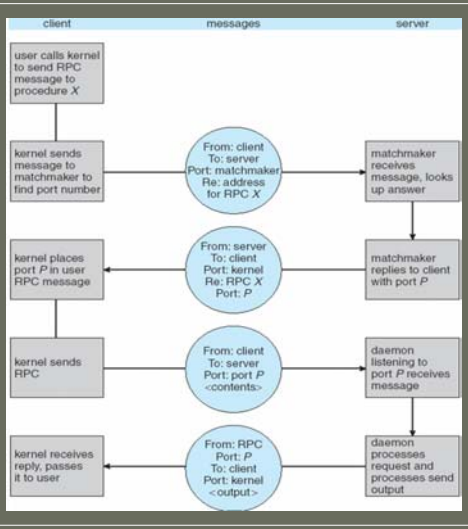
Socket Communication



Remote Procedure Calls

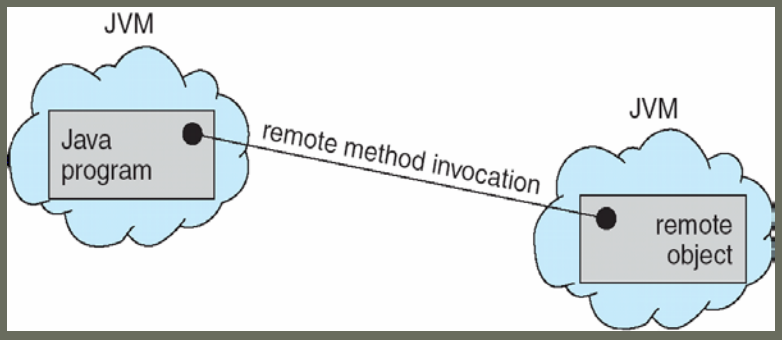
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and *marshalls* the parameters
- The server-side stub receives this message, unpacks the marshalled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

Execution of RPC

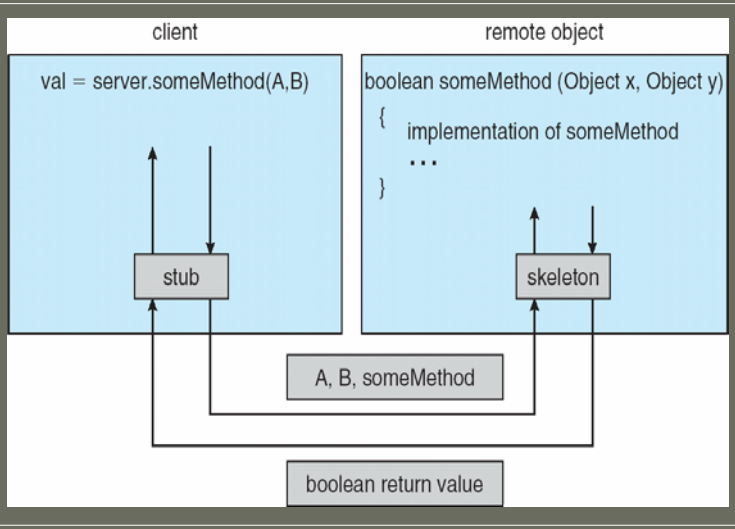


Remote Method Invocation

- 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 Lecture