Chapter 3: Processes

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

Degree of multiprogramming:
nr. of processes in the memory

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
  - Run
    - :~$pstree -p // with names and PIDs
    - :~$pstree -np // arranged in ascending PID
      - To see the list of processes on your linux machine

- Generally, process identified and managed via **a process identifier** (**pid**)

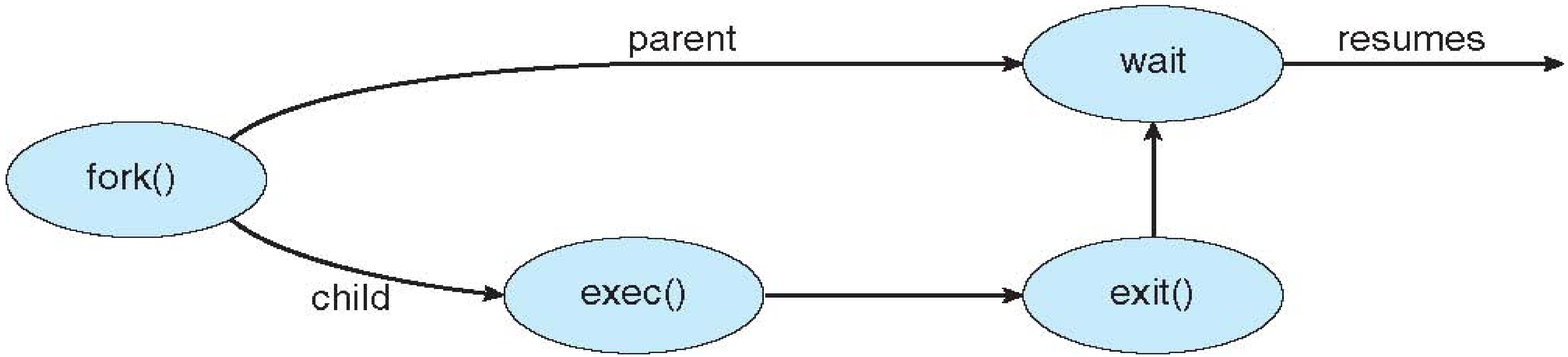
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources

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

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

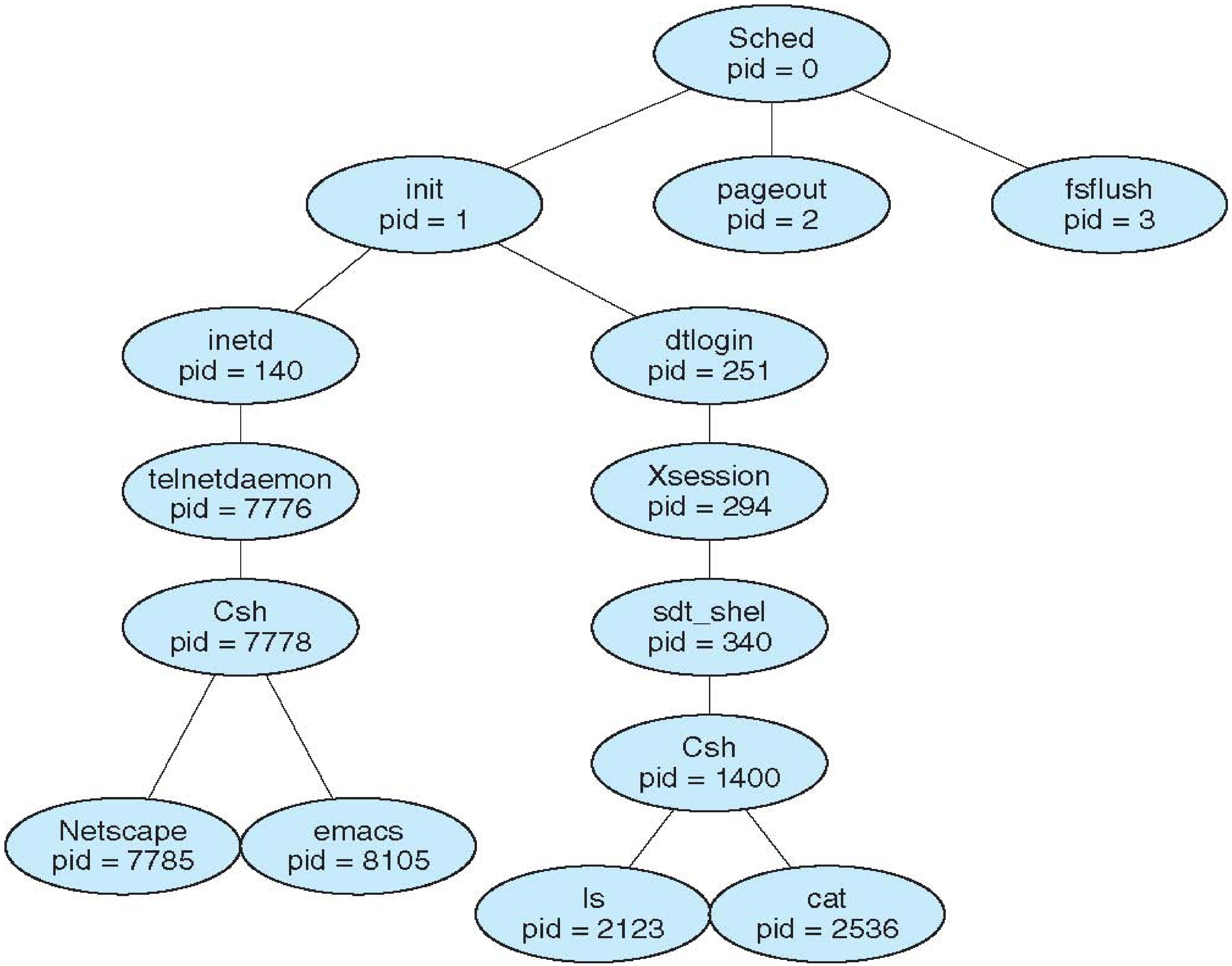
Process Creation



C Program Forking Separate Process

```
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
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 */
        wait (NULL);
        printf ("Child Complete");
    }
    return 0;
}
```

A Tree of Processes on Solaris



Process Termination

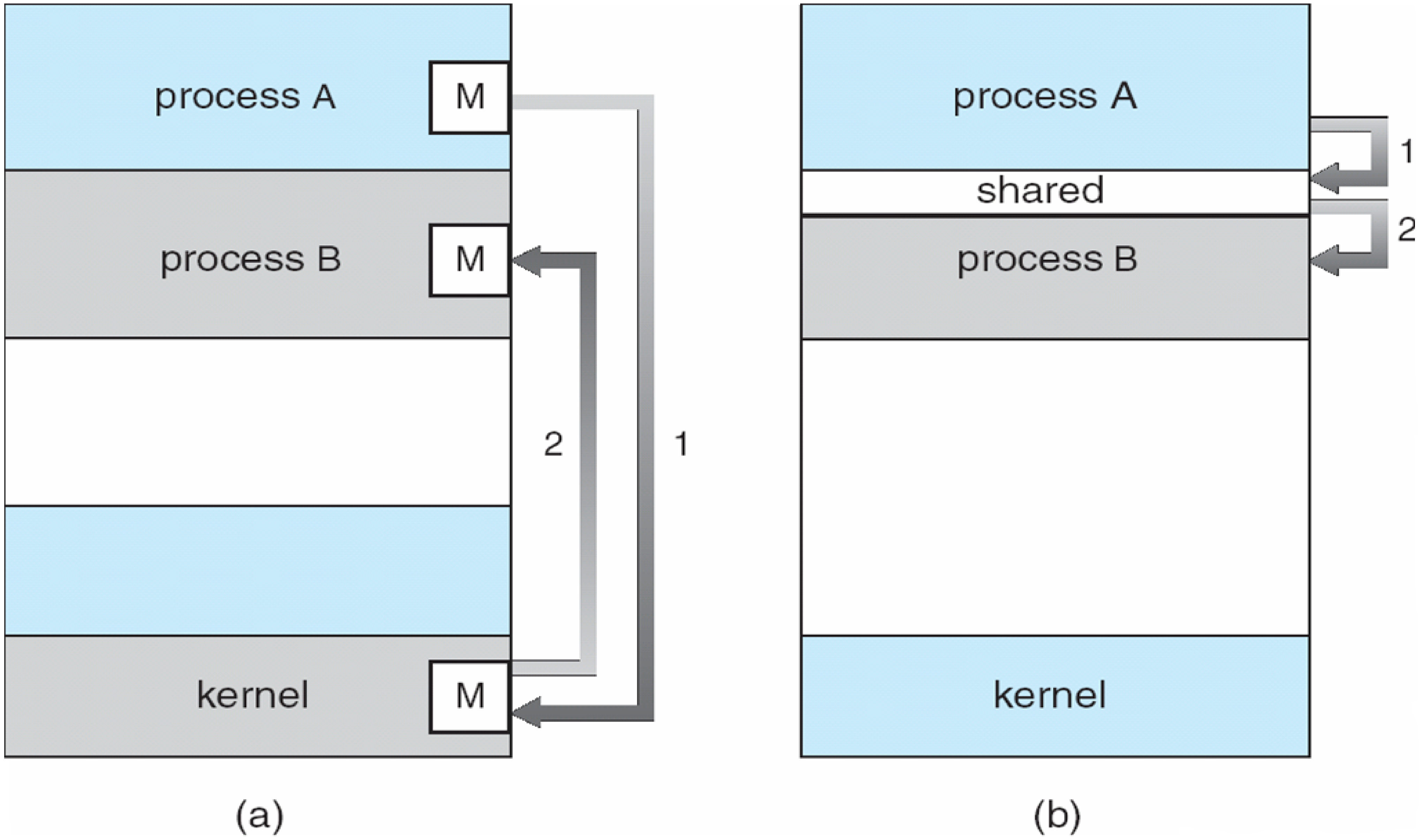
- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - 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**

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

- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - Shared memory
  - Message passing

Communications Models



Message Passing Vs. Shared Memory

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

```
item nextProduced;
  while (true) {
   /* Produce an item */
     while (((in + 1) % BUFFER SIZE)  == out);
                                        /* if buffer is full do nothing
*/
     buffer[in] = item;
     in = (in + 1) % BUFFER SIZE;
  }
```

Bounded Buffer – Consumer

```
item nextConsumed;
while (true) {
          while (in == out);
               // if buffer empty do nothing --
    //nothing to consume

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

Interprocess Communication – Message Passing

- 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 processes *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive

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 (DIRECT)
  - Between each pair there exists exactly one link
  - The link may be either unidirectional or bi-directional

Indirect Communication

- Messages are sent to 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
  - Consequently, 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
  1. create a new mailbox
  2. send and receive messages through mailbox
  3. When done, destroy a mailbox

- Primitives are defined as: A is a mailbox

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

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

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** (is considered **synchronous)**
  - **Blocking send: S**ender blocks until the message is received by the receiver
  - **Blocking receive: R**eceiver blocks until a message is available i.e. sent by the sender
- **Non-blocking** (is considered **asynchronous)**
  - **Non-blocking** send: Here the sender sends the message and continues on to its next instruction
  - **Non-blocking** receive: receiver receive a valid message when required and available otherwise 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; receiver must wait if empty
  3. Unbounded capacity – infinite length
     Sender never waits

IPC Systems – POSIX Shared Memory

- POSIX Shared Memory STEPS: (shall include libraries: unistd, sys/ipc.h, sys/shm.h)
  - Create,
  - Attach,
  - Use and
  - Detach

1. Process first creates shared memory segment:

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

2. Process wanting access to that shared memory must attach to it:

```
    shared memory = (char *) shmat(id, NULL, 0);
// *shmat(int shmid ,void *shmaddr ,int shmflg)
// address of the attached shared memory segment
```

3. Now the process could write to the shared memory:

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

4. When the process wants to detach the shared memory from its address space:

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

Communications in Client-Server Systems

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

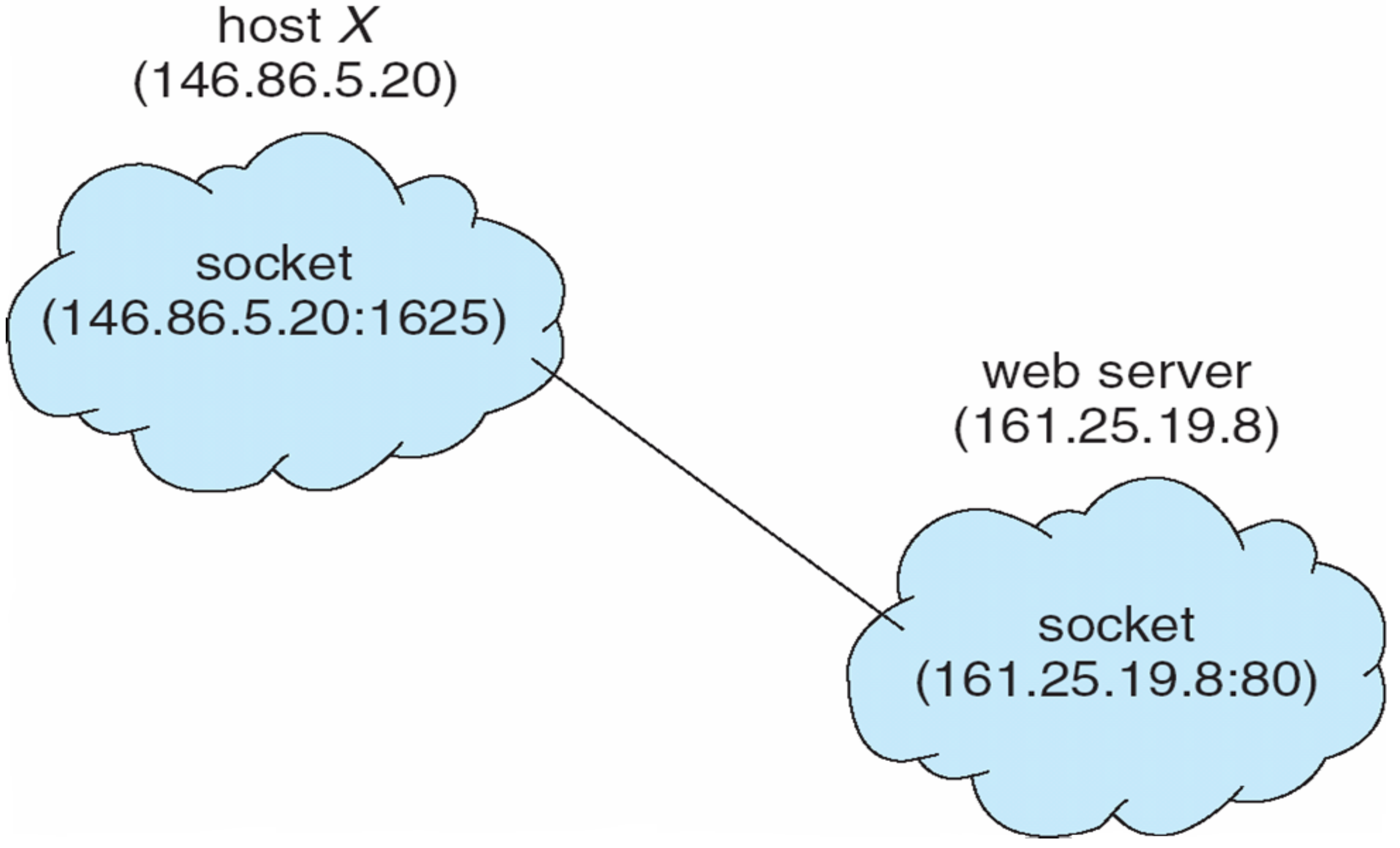
1. Sockets -based Communication

- A **socket** is defined as an *endpoint for communication:*
  - *IP_Address:Port_Number*

Concatenation of IP address and port

- **Examples:**
  - **161.25.19.8:1625**
    - refers to port **1625** on host **161.25.19.8**

- Communication consists between a pair of sockets



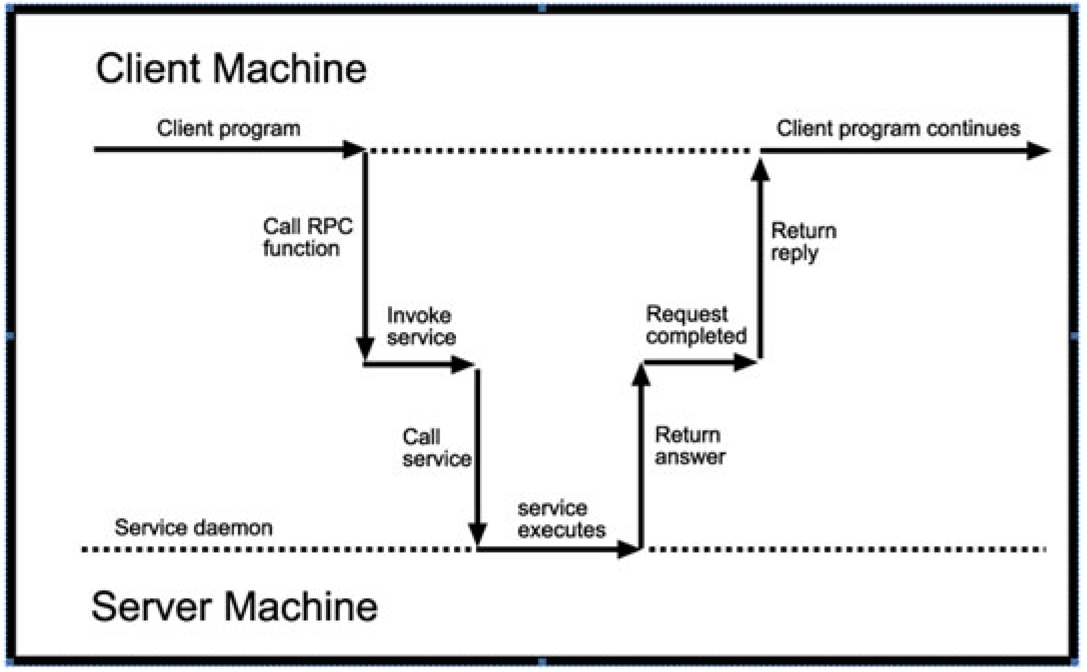
2. Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - The same in the Java OO paradigm is called RMI, there is a whole library and framework for the same

- **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



3. Pipes

- Acts as a conduit allowing two processes to communicate
- Pipe maybe one of the most widely used IPC methods
- Through pipes DATA flows sequentially between connected processes “FIFO”
- PIPE: conceptually an area of main memory that is treated as a “virtual file”
  - One process P1 can write to this “virtual file”
  - Anothter process P2 can read from it
- If a process tries to read before something is written to the pipe
  - the process is suspended until something is written.
- If a pipe is full, the writing process has to wait for a free space!
- Simple example of using pipe:
- :~$ cat filename | grep hello
  - The effect is
    - The stdout filedescripter of “cat” process gets connected to the input side of piple
    - The stdin file descripter of “grep” process gets connected to output of the pipe

Ordinary Pipes

- **Ordinary Pipes** allow communication in standard producer-consumer style
- Producer writes to one end (the *write-end* of the pipe)
- Consumer reads from the other end (the *read-end* of the pipe)
- Ordinary pipes are therefore unidirectional

Operating System Services

Services provided to user programs: (implemented as Sys. progs)

- Controlling execution of processes
- Scheduling of processes fairly on CPU
- Allocating MM for an exec. Process (via swapping)
- Allocating seconary memory for efficient storage and retrieval. (file sytem, protection)

- I/O operations
  - User program cannot directly access I/O hardware, OS does the low level part for them
  - OS provides controlled access to peripheral devices such as terminals, keyboard, disc drives, etc.

- Communications
  - Both inter-process on the same computer, and between computers over a network
  - via shared memory or through message passing
- Error detection
  - Errors do occur: in the CPU and memory hardware, in I/O devices, in user programs
  - OS should handle them appropriately to ensure correct and consistent computing
  - Low level debugging tools really help

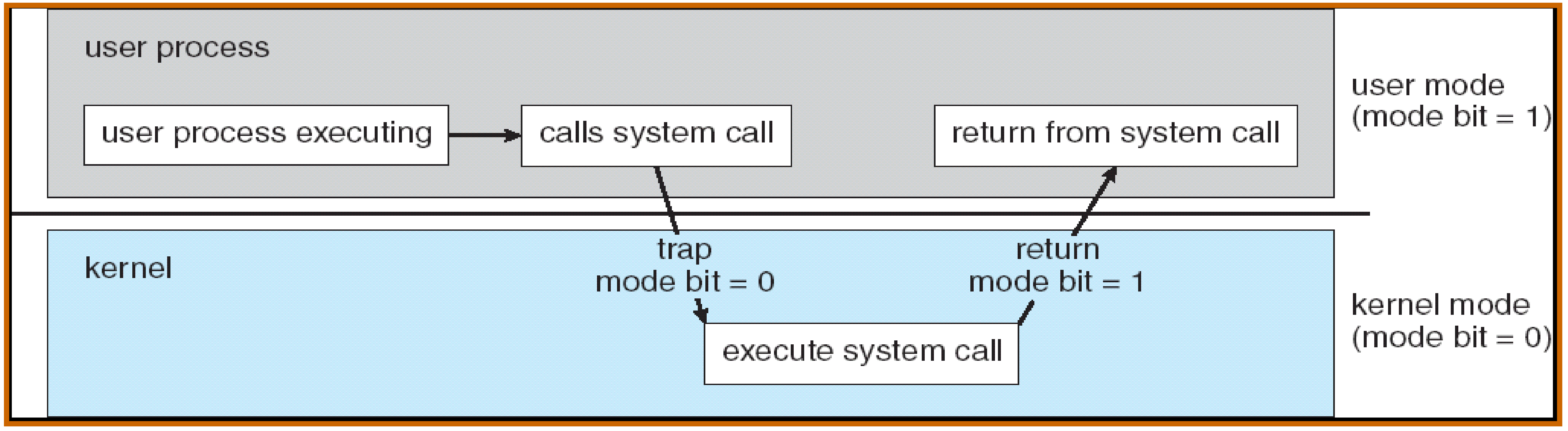
Operating-System Operations

- OS is interrupt driven
- Interrupts raised by hardware and software
  - Mouse click, division by zero, request for operating system service
  - Timer interrupt (i.e. process in an infinite loop), memory access violation (processes trying to modify each other or the operating system)
- Some operations should be performed only by a trusted party
  - Accessing hardware, memory-management registers
  - A rogue user program could damage other programs, steal the system for itself, …

  - Solution: dual-mode operation

Transition from User to Kernel Mode

- Dual-mode operation allows OS to protect itself and other system components
  - User mode and kernel mode
  - Mode bit provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as privileged, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user



More details …..

- The user-mode program inits registers, (stack frame with arguments)
  - indicating what specific service it requires from the operating system
  - User-mode program invokes the trap instruction.
  - Trap_handler checks the reason for trap and if found to be due to system_call
    - CPU switches to kernel mode, and jumps to instructions at a fixed location in memory corresponding to the system_call.
    - system call when done, clears the mode to underprivileged user-mode and returns from the system call