Chapter 3: Processes

Programs in execution

Thanks to the author of the textbook [**SGG**] for providing the base slides. I made several changes/additions.
These slides may incorporate materials kindly provided by Prof. Dakai Zhu.
So I would like to thank him, too.
Turgay Korkmaz

Chapter 3: Processes

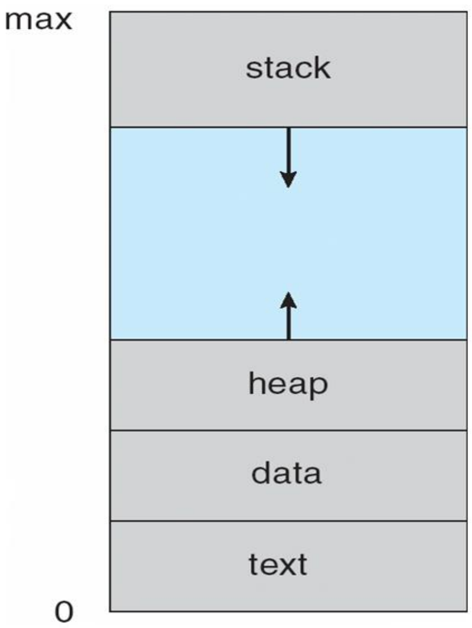
- Process Concept *
- Process Scheduling **** (more in ch 5)
- Operations on Processes ***
- *Interprocess Communication* ****
- *Communication in Client-Server Systems ***** (more later)*
  - *Socket, RPC, RMI*
- Examples of IPC Systems *

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 (more later)

Process Concept

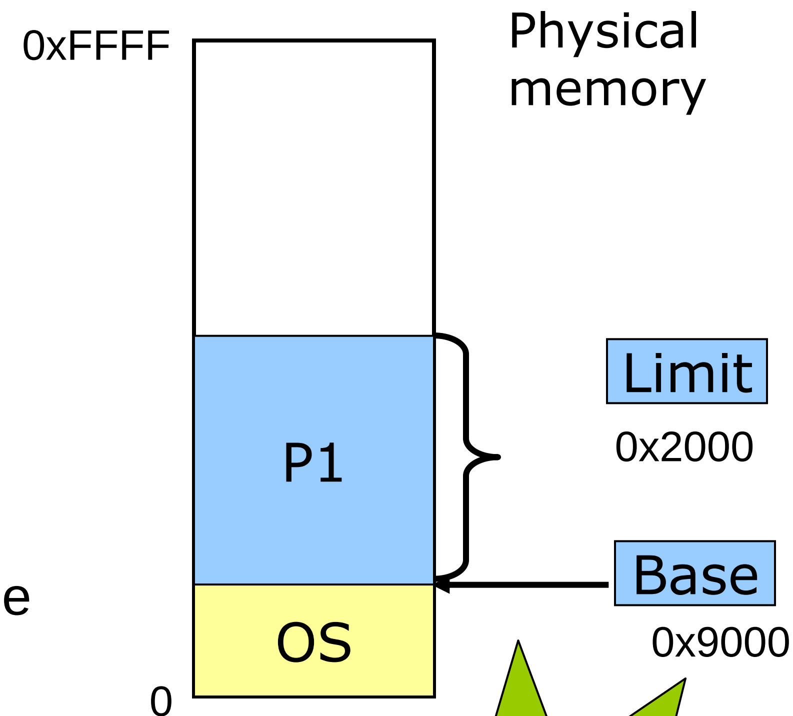
- **Program**: a set of instructions
  - Passive entity, stored as files on disk
- **Process**: *a program in execution*
  - **Dynamic** concept, an **active** entity in memory
  - A process includes:
    - **code** section (text segment),
    - **data** section (global variables),
    - **stack** (temporary data or local variables and return address etc.)
    - **heap**: memory for dynamically allocated data
    - Auxiliary: environment variables and command line arguments
  - Process execution must progress in sequential fashion (single thread)
- An OS executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – user programs or **tasks**
  - The terms *job* and *process* are used interchangeably

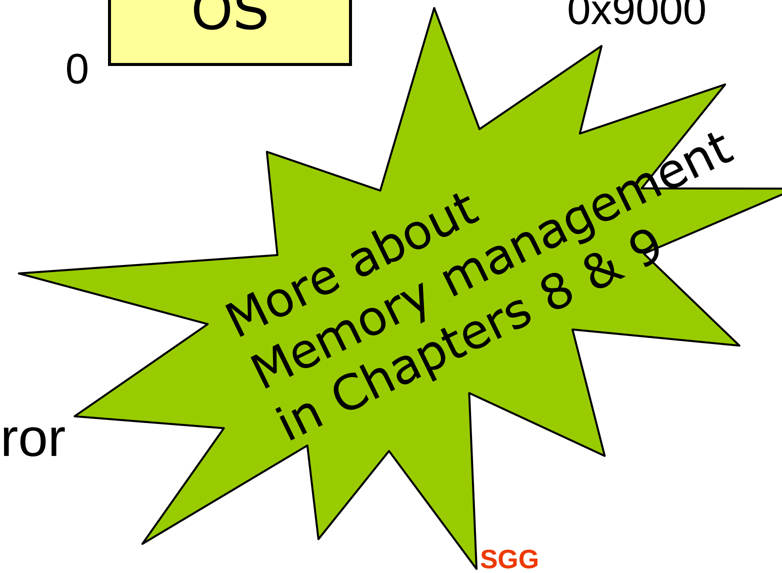


How do we run a program? What are the steps to create a process?

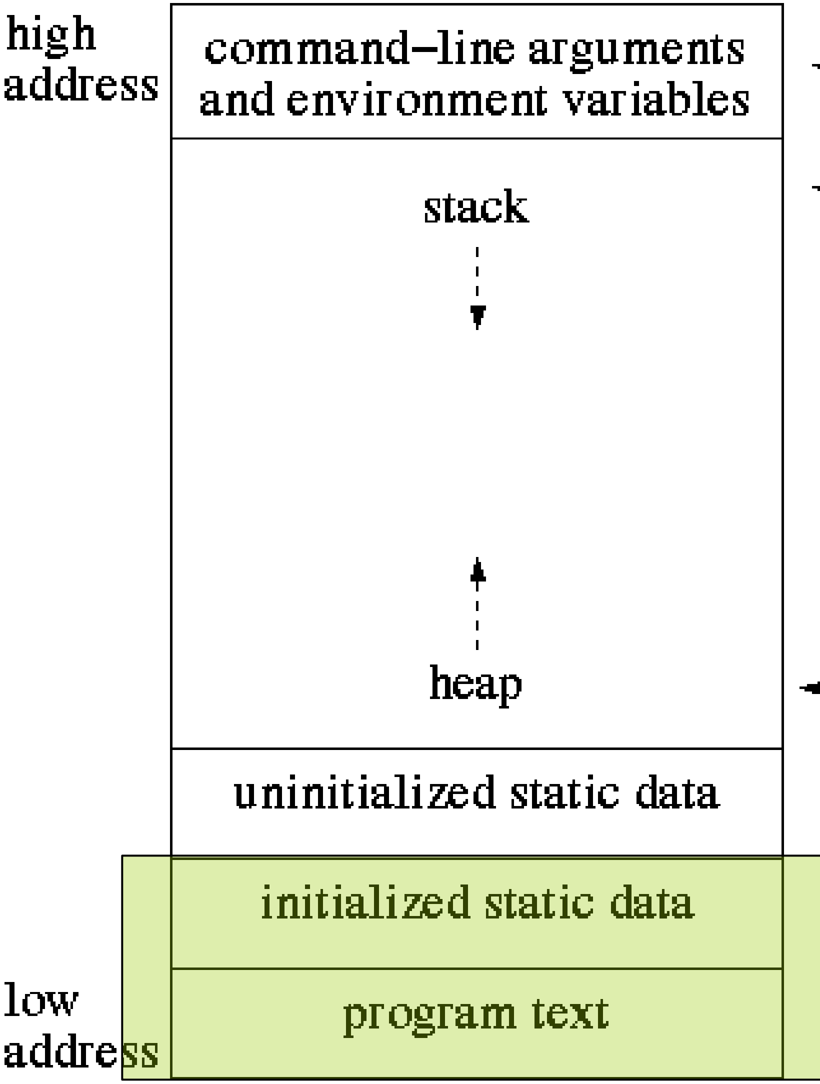
Load to Memory

- Special CPU registers
  - **Base register**: start of the process's memory partition
  - **Limit register**: length of the process's memory partition
  - Access limited to system mode
- Address *translation*
  - **Logical address**: location from the process's point of view
  - **Physical address**: location in actual memory
  - Physical = base + logical address
    - Logical address: 0x1204
      Physical address:0x1204+0x9000 = 0xa204
  - Logical address larger than limit → error





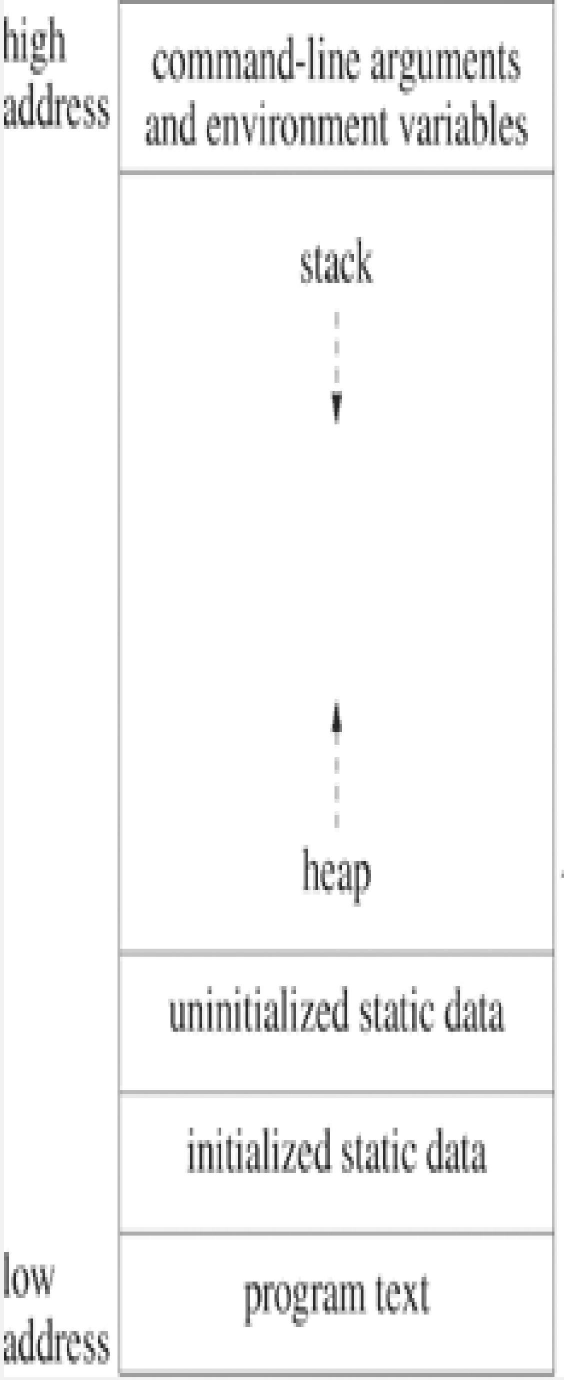
Program Image in Memory (Fig. 2.1 USP)



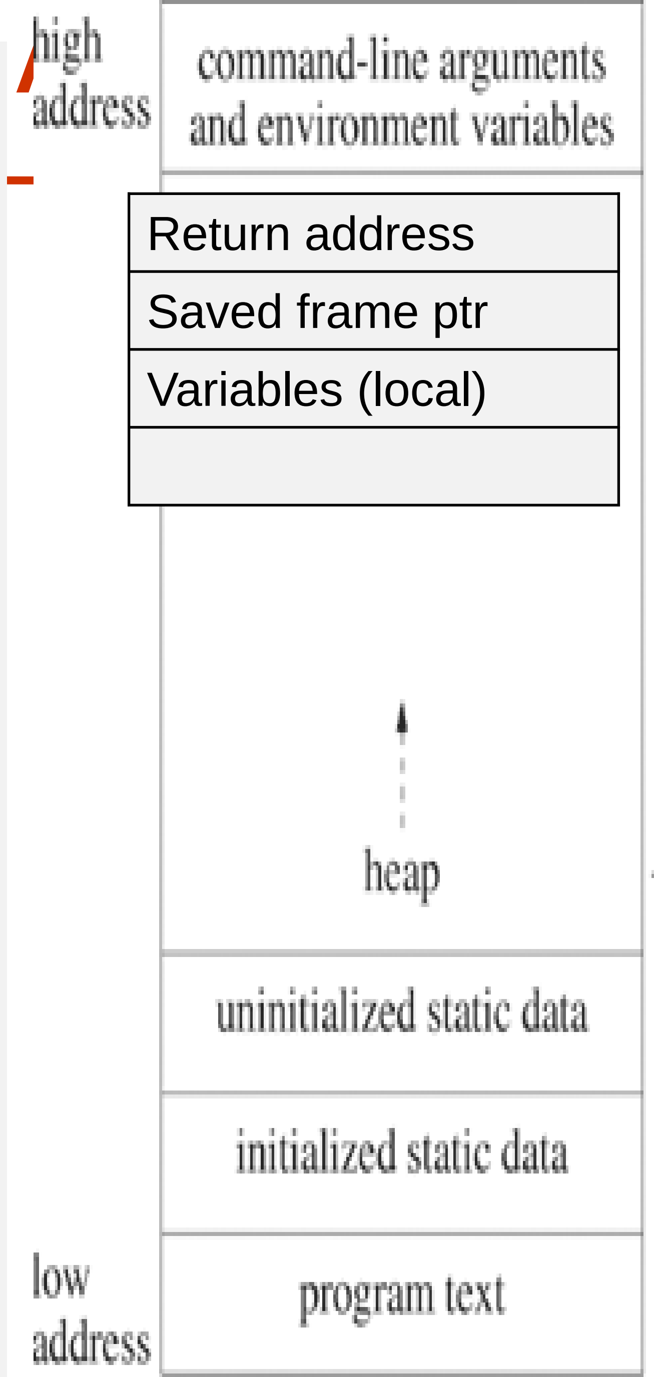
int A[8]; vs. int A[8] = {3,5};

```
> gcc prog.c
> ls -la a.out
… 8710 … a.out
… 8766 … a.out

> size a.out
text    data    bss    dec    hex

filename

2351    568     136    3055   bef
a.out
2351    616     104    3071   bff
a.out
```

```
int A[8];          int B[8]={3};     int i, *ptr;
int main(int argc, char *argv[]) {
   int C[8];  int D[8] ={6};
   printf("argc    at %p contains %d \n", &argc, argc);
   printf("argv    at %p contains %p\n", argv, *argv);
   printf("argv[0] at %p contains %s\n", &argv[0], argv[0]);
   printf("argv[%d] at %p contains %s\n",argc,  &argv[argc]

   printf("C: [0]  at %p contains %d\n", &C[0], C[0]);
   printf("C: [7]  at %p contains %d\n", &C[7], C[7]);
   printf("D: [0]  at %p contains %d\n", &D[0], D[0]);
   printf("D: [7]  at %p contains %d\n", &D[7], D[7]);
   foo(5);
   for(i=0; i<5; i++) {
      ptr = (int *) malloc(sizeof(int));
      printf("ptr     at %p points to %p\n", &ptr, ptr);
   }
   printf("A: [0]  at %p contains %d\n", &A[0], A[0]);
   printf("A: [7]  at %p contains %d\n", &A[7], A[7]);
   printf("B: [0]  at %p contains %d\n", &B[0], B[0]);
   printf("B: [7]  at %p contains %d\n", &B[7], B[7]);
   printf("foo     at %p\n", foo);
   printf("main    at %p\n", main);
   return 0;
}
int foo(int x){
   printf("foo     at %p x is at %p contains %d\n", foo, &x,
   if (x > 0) foo(x-1);
   return x;
}
```

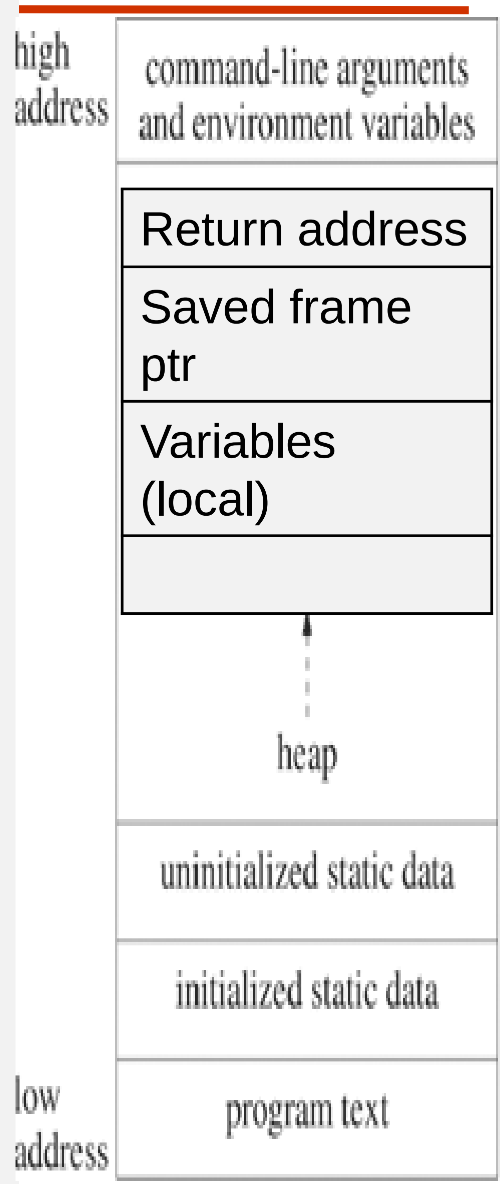


```
argc     at 0x7fff378e0e3c contains 1
argv     at 0x7fff378e0f68 contains 0x7fff378e2a0b
argv[0] at 0x7fff378e0f68 contains a.out
argv[1] at 0x7fff378e0f70 contains (null)
C:  [0]  at 0x7fff378e0e40 contains 1
C:  [7]  at 0x7fff378e0e5c contains 0
D:  [0]  at 0x7fff378e0e60 contains 6
D:  [7]  at 0x7fff378e0e7c contains 0
foo      at 0x4007ba x is at 0x7fff378e0e1c contains 5
foo      at 0x4007ba x is at 0x7fff378e0dfc contains 4
foo      at 0x4007ba x is at 0x7fff378e0ddc contains 3
foo      at 0x4007ba x is at 0x7fff378e0dbc contains 2
foo      at 0x4007ba x is at 0x7fff378e0d9c contains 1
foo      at 0x4007ba x is at 0x7fff378e0d7c contains 0
ptr      at 0x6010a0 points to 0x88e010
ptr      at 0x6010a0 points to 0x88e030
ptr      at 0x6010a0 points to 0x88e050
ptr      at 0x6010a0 points to 0x88e070
ptr      at 0x6010a0 points to 0x88e090
A:  [0]  at 0x6010c0 contains 0
A:  [7]  at 0x6010dc contains 0
B:  [0]  at 0x601060 contains 3
B:  [7]  at 0x60107c contains 0
foo      at 0x4007ba
main     at 0x40057d
```



What might go wrong here? How can we fix?

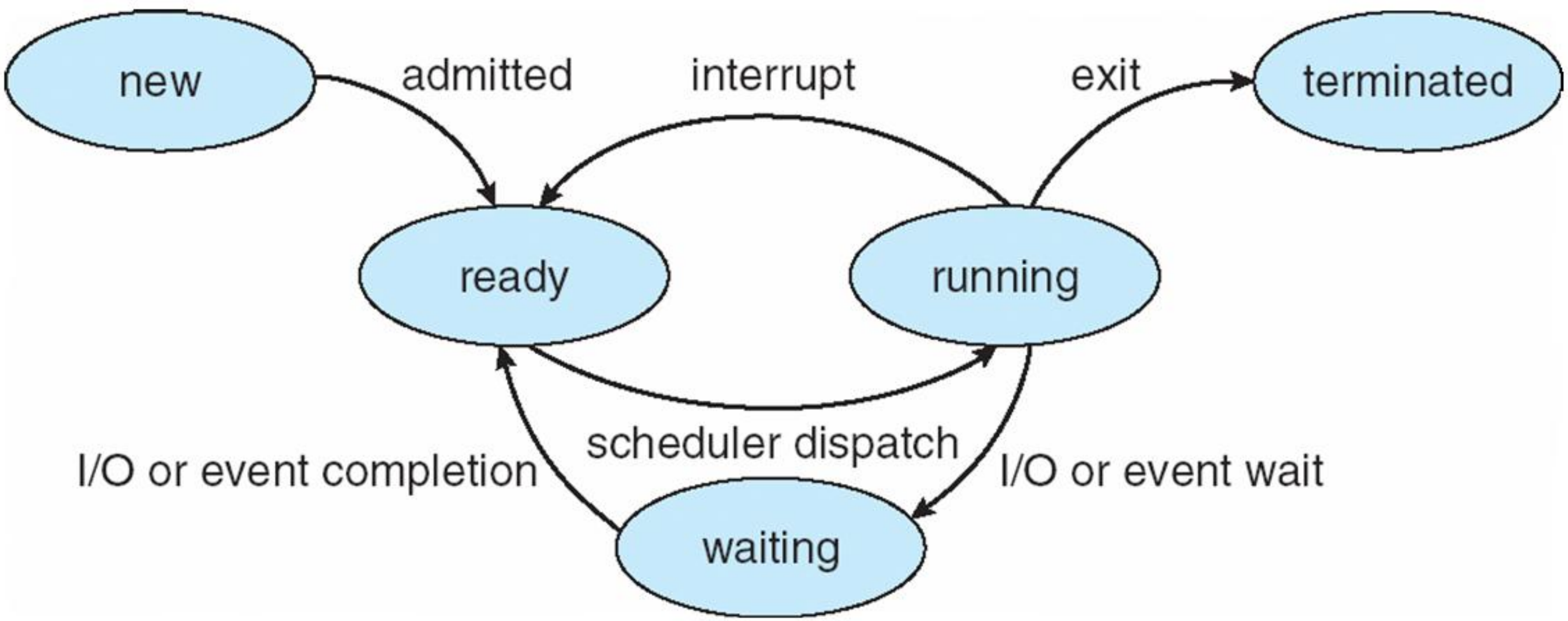
```
char *get_me_a_name(){
      char buff[100];
      scanf("%s", buff);
      return buff;
}
char *get_me_a_name(){
      static char buff[100];
      scanf("%s", buff);
      return buff;
}
char *get_me_a_name(){
      char *buff;
      buff = malloc(100); // if NULL
  ?
      scanf("%s", buff);
      return buff;
}
```



Process State

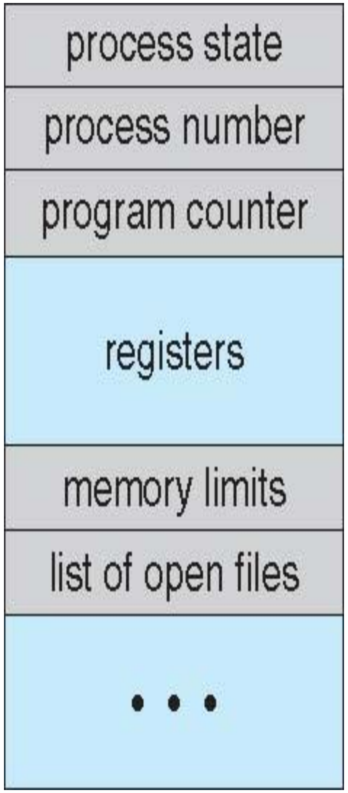
CPU Switch From Process to Process

- 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



Process Control Block (PCB)

- Process state
- **Registers**: in addition to general registers
  - **Program Counter (PC):** contains the memory address of the next instruction to be fetched.
  - **Stack Pointer (SP**): points to the top of the current *stack* in memory. The stack contains one frame for each procedure that has been entered but not yet exited.
  - **Program Status Word (PSW**): contains the condition code bits and various other control bits
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information
- Thread synchronization and communication resource: semaphores and sockets



/linux-3.6.5/include/linux/sched.h

```
struct task_struct {
        volatile long state;
        void *stack;
        atomic_t usage;
        unsigned int flags;
        unsigned int ptrace;
        /* ... ~1.7K
         360 lines */
}
```

Double linked list to maintain PCBs

Threads

- A process can have multiple threads

but…

- We just consider single thread here….
- Multiple threads will be covered later (Chapter 4)….

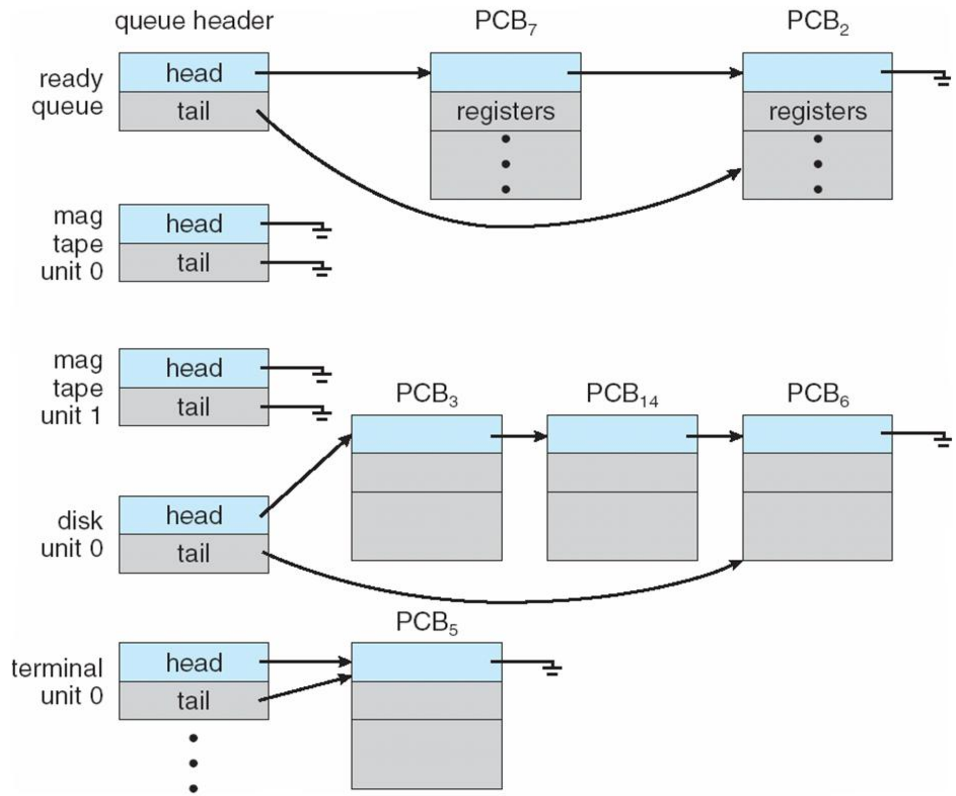
Maximize CPU utilization in time sharing system

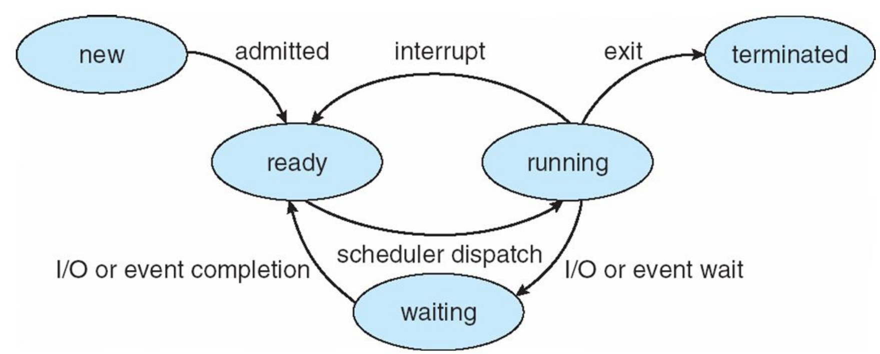
(More in Chapter 5)….

PROCESS SCHEDULING

Scheduling Queues

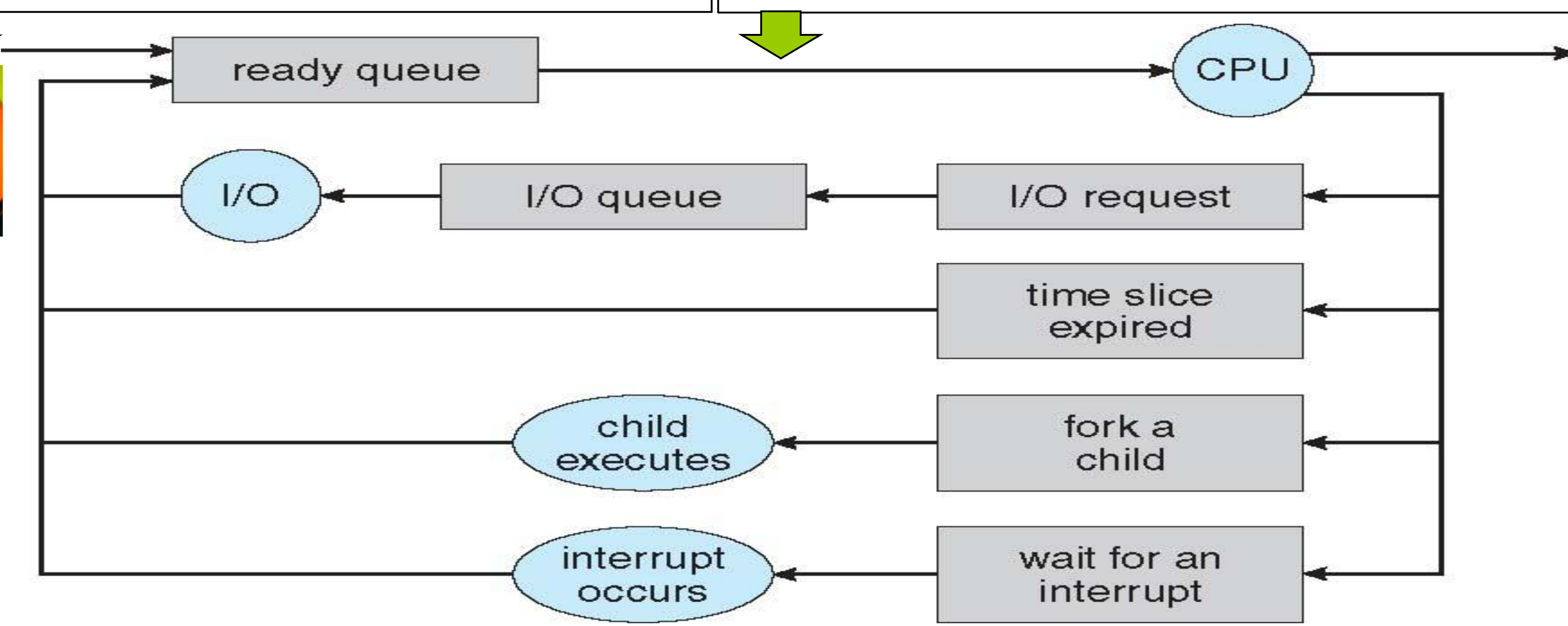
- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues
- **Dispatcher** takes the next task from ready queue and executes it





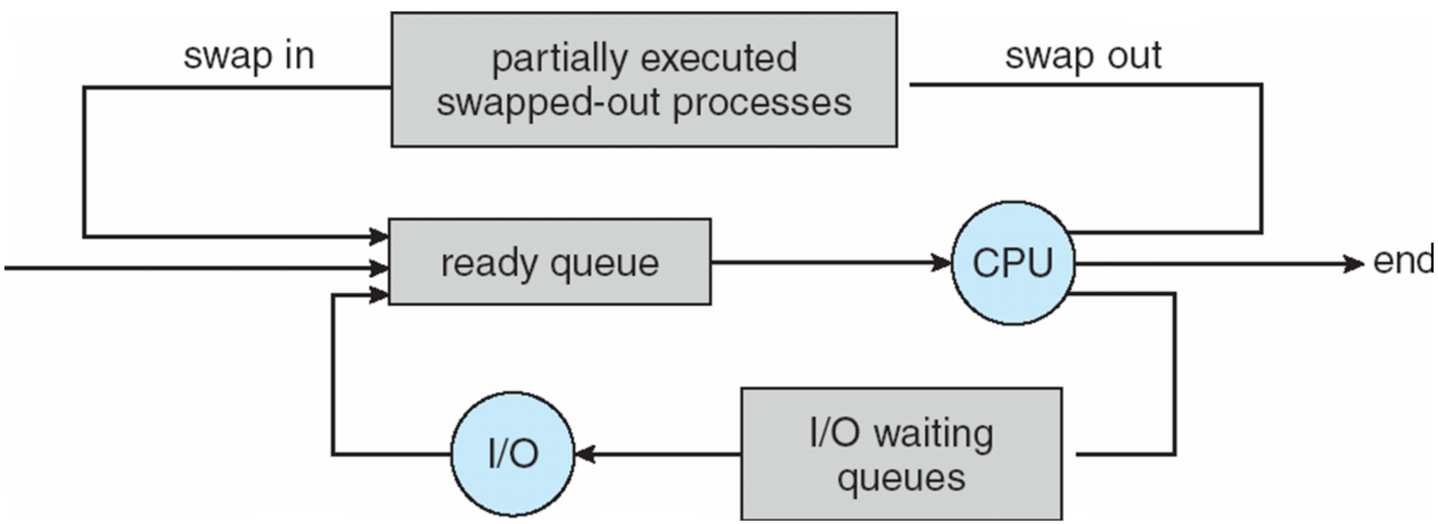
Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
  - Less frequent
  - Controls degree of multiprogramming
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - More frequent (e.g., every 100 ms)
  - Must be fast (if it takes 10ms, then we have ~10% performance degradation)



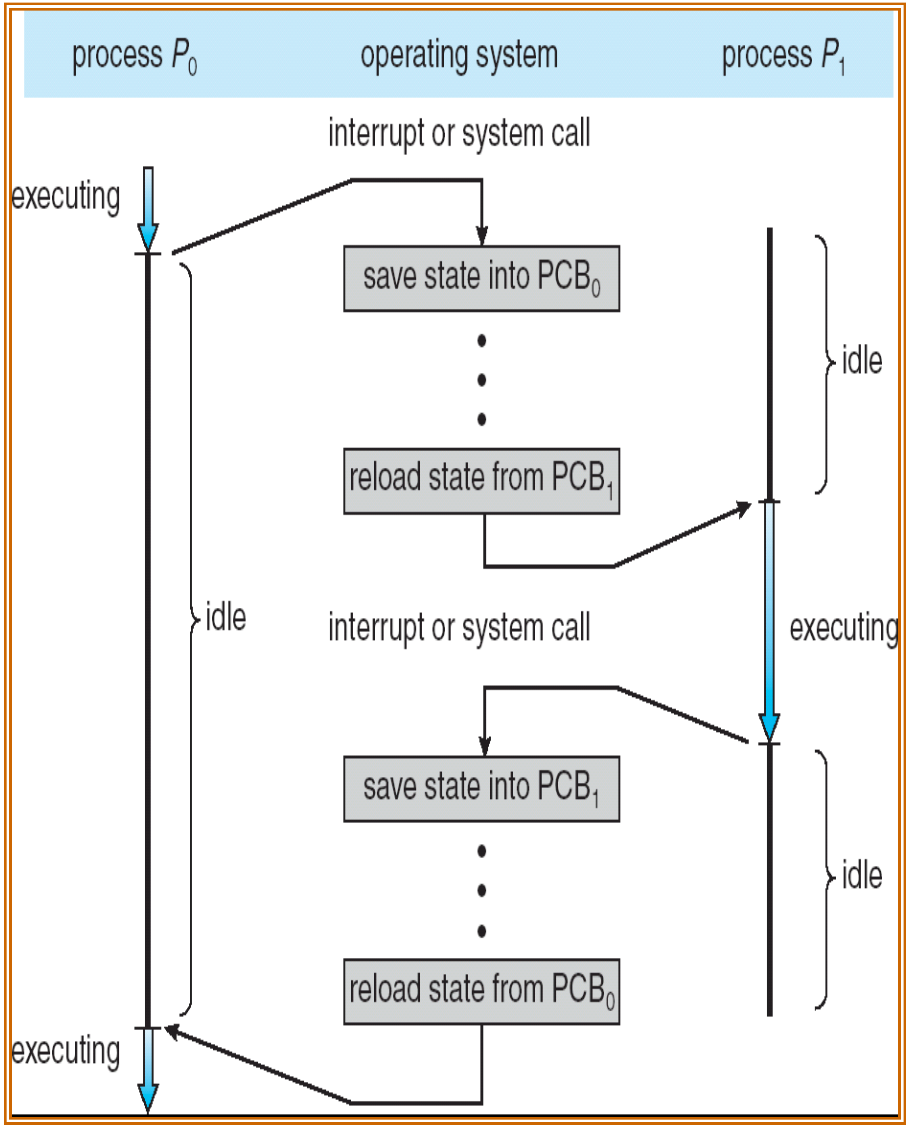
Schedulers (Cont.)

- 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
- Addition of Medium Term Scheduling
  - Fine tune degree of multiprogramming



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
- Hardware support
  - Multiple set of registers then just change pointers
- Other performance issues/problems
  - Cache content: locality is lost
  - TLB content: may need to flush



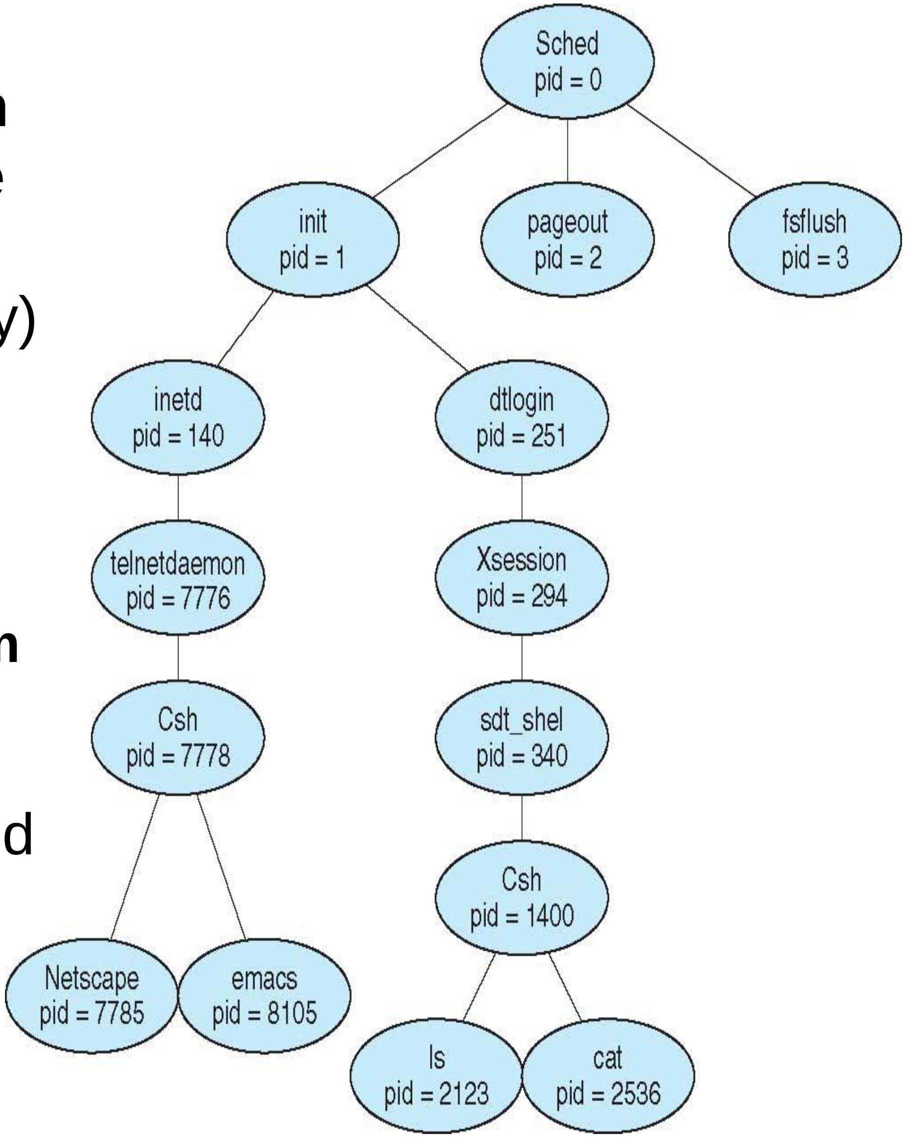
Let's try to see context switching in action?

```
> vi prog.c
  for(i=0; i<1000; i++) {
     // …
     printf("%s", argv[1]);
  }
> prog A &; prog B &; prog C &
```

OPERATIONS ON PROCESSES

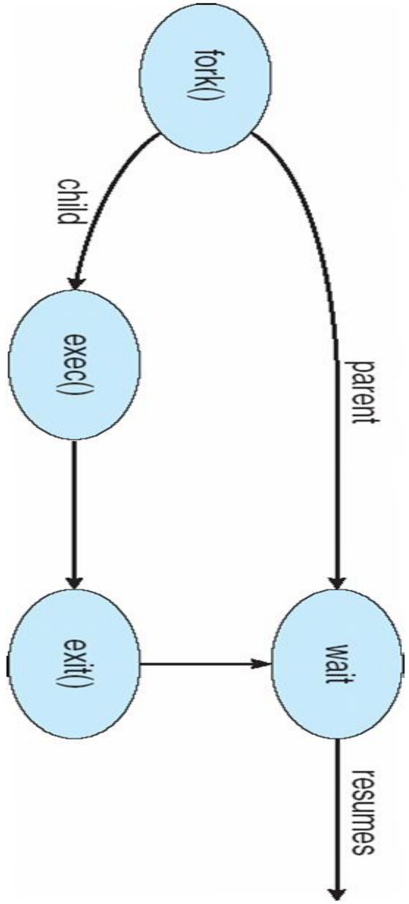
Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes (Process hierarchy)
  - System initialization
  - User request to create a new process
  - Running processes use **system call** to create new process
- Generally, process identified and managed via **a process identifier** (**pid**)



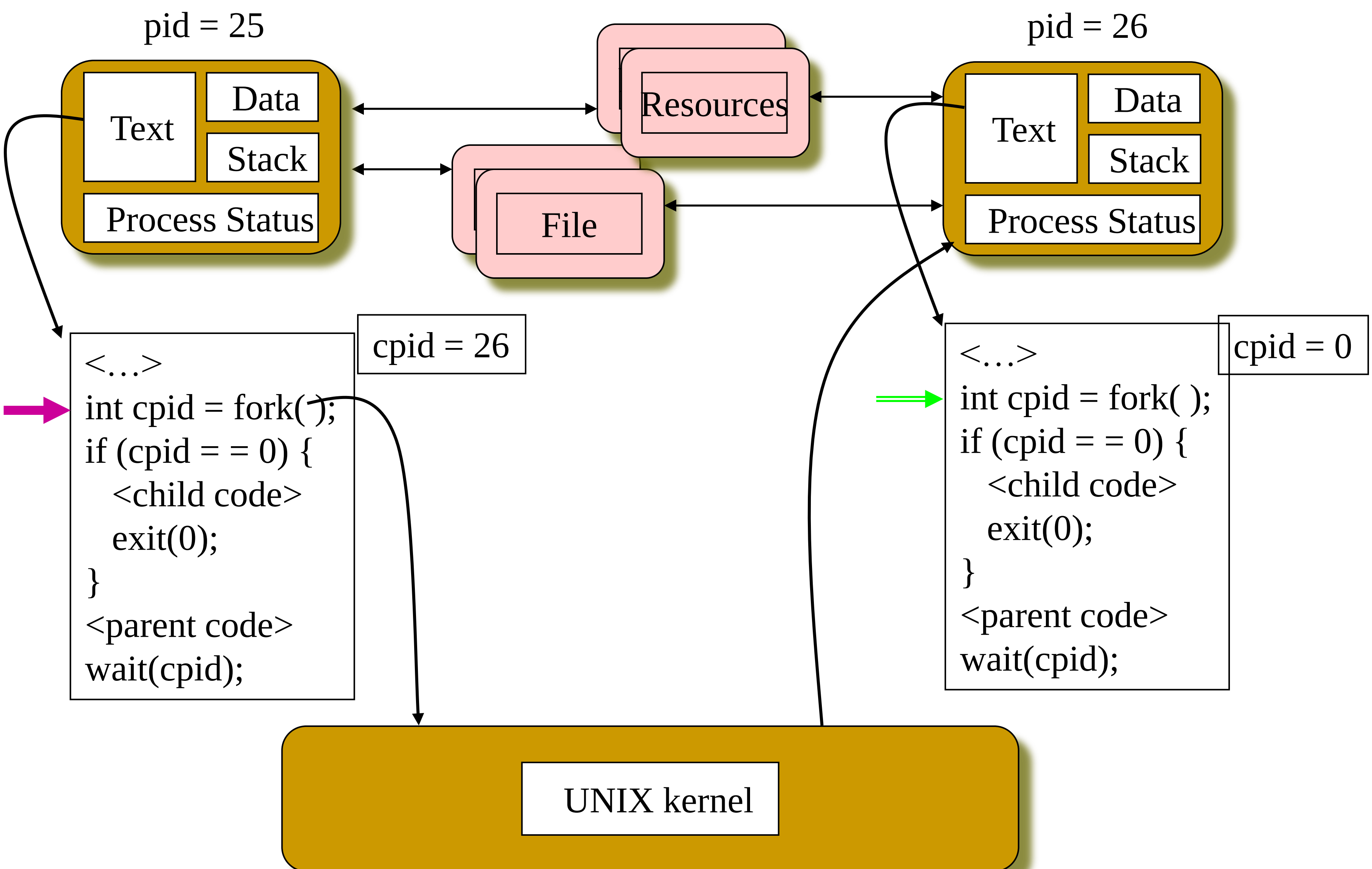
Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- UNIX examples
  - **fork** system call creates new process
    - The child process has a ***separate*** *copy* of the parent's address space.
    - Both the parent and the child continue execution at the instruction following the ***fork*** system call
    - Return value of 0 → new (child) process continues
  - **exec** system call used after a **fork** to replace the process' memory space with a new program



*

An Example: Unix `fork( )`



C Program Forking Separate Process

```
int main()
{
pid_t  pid;
   pid = fork();                          /* fork another process */

   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);
   }
}
```

Process Creation in POSIX and Win32

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

int main()
{
pid_t pid;

    /* fork a child 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);
    }
}
```

```c
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    // allocate memory
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // create child process
    if (!CreateProcess(NULL, // use command line
      "C:\\WINDOWS\\system32\\mspaint.exe", // command line
      NULL, // don't inherit process handle
      NULL, // don't inherit thread handle
      FALSE, // disable handle inheritance
      0, // no creation flags
      NULL, // use parent's environment block
      NULL, // use parent's existing directory
      &si,
      &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    // parent will wait for the child to complete
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    // close handles
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Process Creation in Java

```
import java.io.*;

public class OSProcess
{
  public static void main(String[] args) throws IOException {
    if (args.length != 1) {
      System.err.println("Usage: java OSProcess <command>");
      System.exit(0);
    }

    // args[0] is the command
    ProcessBuilder pb = new ProcessBuilder(args[0]);
    Process proc = pb.start();

    // obtain the input stream
    InputStream is = proc.getInputStream();
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);

    // read what is returned by the command
    String line;
    while ( (line = br.readLine()) != null)
      System.out.println(line);

    br.close();
  }
}
```

JVM is created as an ordinary application.

Each JVM supports multiple threads but not process model in a JVM. Why?

Java allows to create external processes using ProcessBuilder class…

Process Termination

- **Voluntarily**
  - process finishes and asks OS to delete it (**exit**).
  - Output data from child to parent (**wait** or **waitpid**).
  - Process' resources are de-allocated by OS.
- **Involuntarily**
  - parent terminate execution of children processes (e.g. **TerminateProcess()** in Win32, **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**)
    - Some operating system do, and init owns them
- Parent process is terminated (e.g., due to errors)
  - What will happen to the children process?!

Zombies

Wait for Processes

```
#include <sys/wait.h>
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

- **wait** : parent blocks until the child finishes
  - If a child terminated, return its pid
  - Otherwise, return -1 and set `errno`
- **waitpid** : parent blocks until a specific child finishes
  - Allow to wait for a particular process (or all if pid=-1);
  - NOHANG option: return 0 if there is a specified child to wait for but it has not yet terminated
- Important values of `errno`
  - ECHILD no unwaited for children;
  - EINTR a signal was caught

!!!!!!! (more later) !!!!!

Information Sharing

Computation speedup

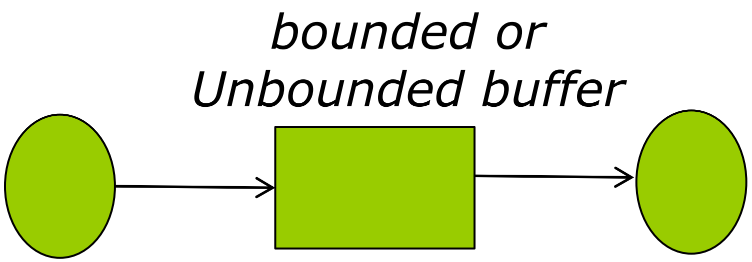
Modularity

Convenience (user can do multiple things…)

INTERPROCESS COMMUNICATION

Interprocess Communication

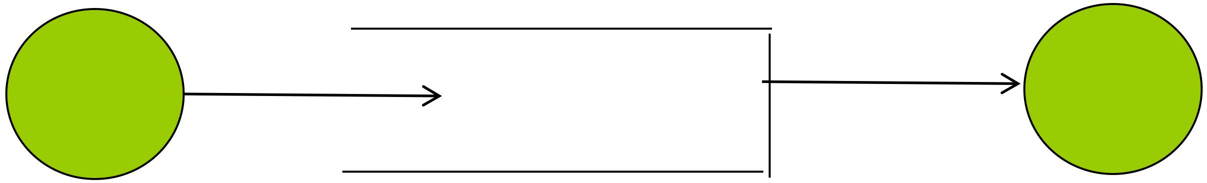
- Processes within a system may be **independent** or **cooperating**
- **Cooperating process** can affect or be affected by other processes, including sharing data
- Example: Producer-Consumer Problem
  - *producer* process produces information that is consumed by a *consumer* process
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience



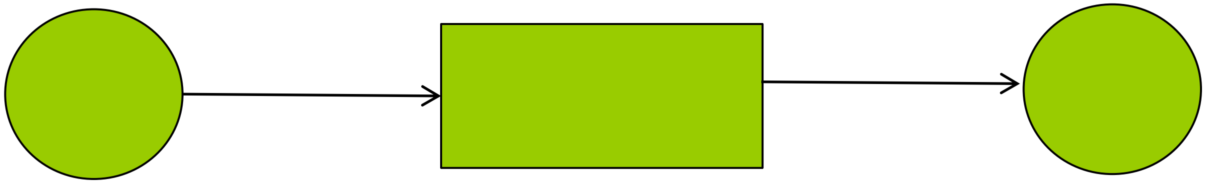
- Cooperating processes need **interprocess communication** (**IPC**)
  - Shared memory
  - Message passing

Producer-Consumer Problem

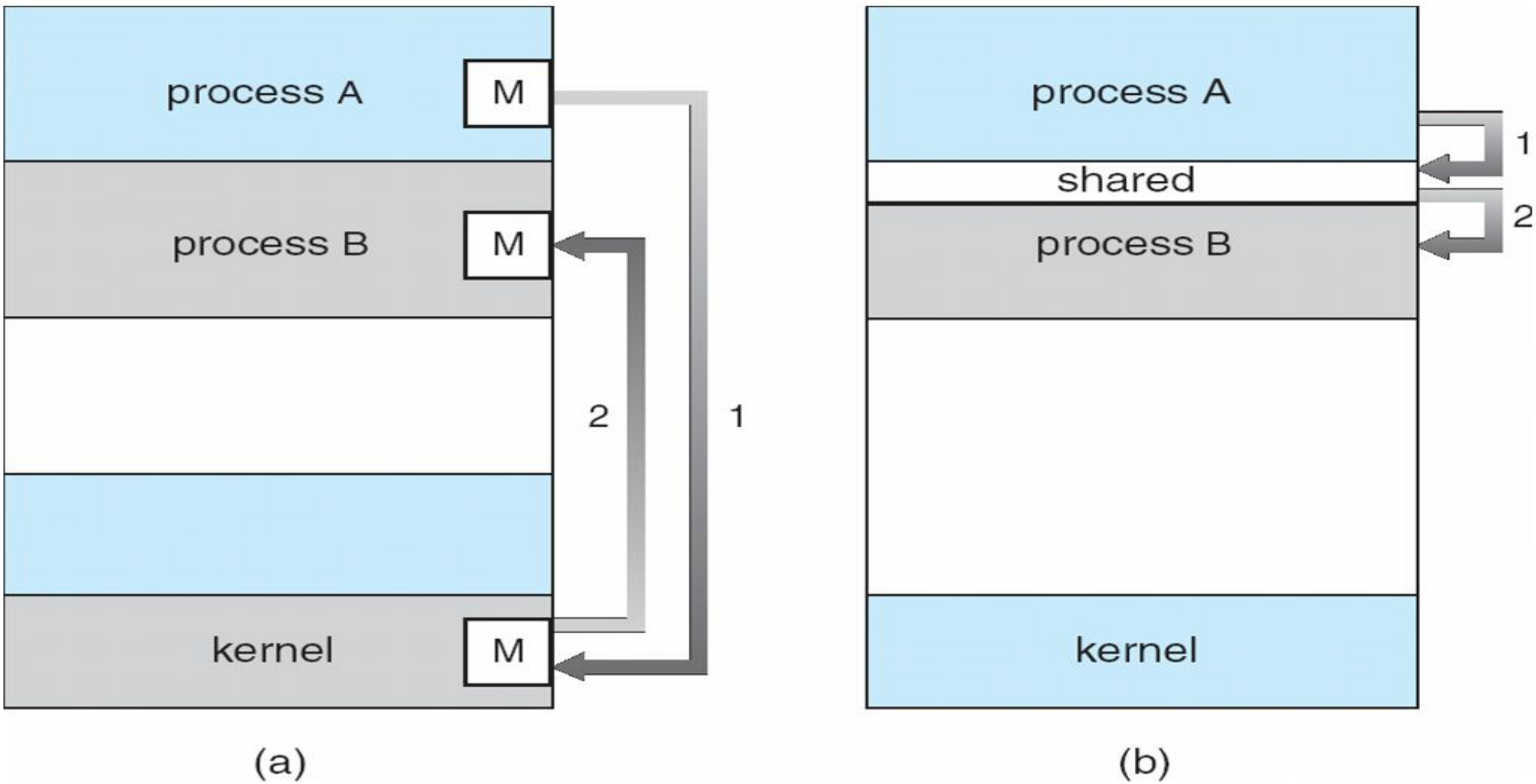
- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer



  - *bounded-buffer* assumes that there is a fixed buffer size



Communications Models



Message Passing

Shared Memory

Self-study

SHARED-MEMORY

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BSIZE 10
typedef struct {
    . . .
} itemT;

itemT buffer[BSIZE];
itemT item;
int in = 0;
int out = 0;
```

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

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

```
while (true) {
    while (in == out)
        ; // do nothing -- nothing to consume
    item = buffer[out];
    out = (out + 1) % BSIZE;
    /* Consume the  item */
}
```

POSIX Shared-Memory APIs

- POSIX 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);
```

- `shmctl`: alter the permission of the shared segment

```
shmctl(shm_id, cmd, *buf);
cmd: SHM_LOCK, SHM_UNLOCK, IPC_STAT, IPC_SET, and IPC_RMID
```

IPC with Shared Memory (POSIX/C)

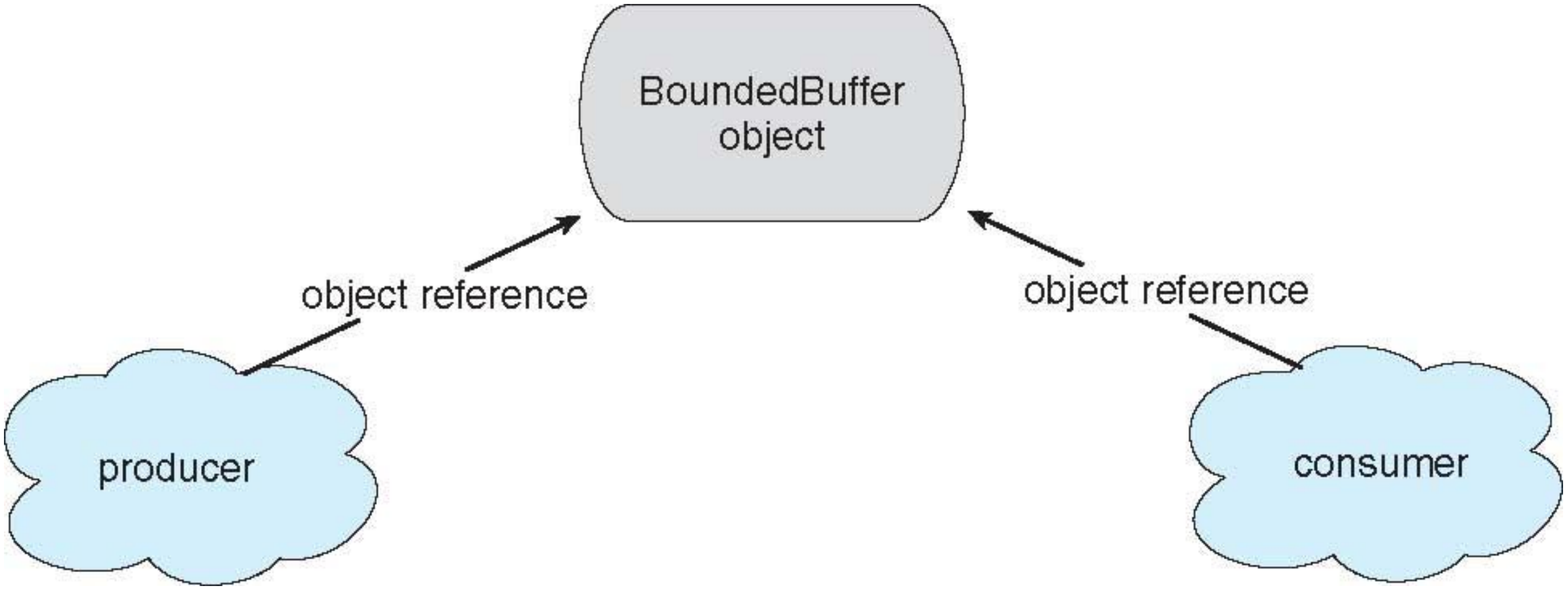
```
#include <sys/ipc.h>
#include <sys/shm.h>
int main(int argc, char *argv[]) {
 int shmid;        char *data;
 /* create the shared segment: */
 shmid = shmget(100, 1024, 0644 | IPC_CREAT))
 /* attach it to a pointer */
 data = shmat(shmid, (void *)0, 0);
 /* write some data */
 sprintf(data, "Hi, I am writing share memory");
 shmdt(data); /* detach from the segment: */
 /*remove the shared segment*/
 shmctl(shmid, IPC_RMID, NULL);
 return 0;
}
```

Communicate via Shared Memory

```
int main(int argc, char *argv[]) {
 int shmid;
 char * data;
 … … /* setup the shared segment: */
 if ( (cpid = fork())==0 ){ //child process
   sprintf(data, “Child: using SM!”);
   sleep(1); //give parent a chance
   printf(“%s\n”, data);      exit(0);
 } else if (cpid >0){ //parent process
   sleep(1); //let child first
   sprintf(data, “Parent: changing SM”);
   wait(cpid); //wait child to finish
 }
 shmdt(data);
 shmctl(shmid, IPC_RMID, NULL);
 return 0;
}
```

Simulating Shared Memory in Java

Java does not provide support for shared memory, but it can be emulated …



!!!!!!! (more later) !!!!!

MESSAGE PASSING

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 at least 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)

When to use shared memory vs. message passing?

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?

Naming and 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

Naming and 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
- 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

Naming and Indirect Communication (cont'd)

- 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.

Message Passing: 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

Message Passing: 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

Example: Producer-Consumer with Message Passing in Java

```java
import java.util.Vector;
public class MessageQueue<E>
          implements Channel<E>
{
   private Vector<E> queue;
   public MessageQueue() {
          queue = new Vector<E>();
   }

  public void send(E item) {
          queue.addElement(item);
   }

   public E receive() {
       if (queue.size() == 0)
          return null;
       else
          return queue.remove(0);
   }
}
```

```java
public interface Channel<E>
{
          public void send(E item);
          public E receive();
}
```

```java
import java.util.Date;
public class Test
{
  public static void main(String[] args){
     Channel<Date> mailBox =
        new MessageQueue<Date>();
        mailBox.send(new Date());

        Date rightNow =
                mailBox.receive();
        System.out.println(rightNow);
   }
}
```

!!!!!!! (more later) !!!!!

Sockets

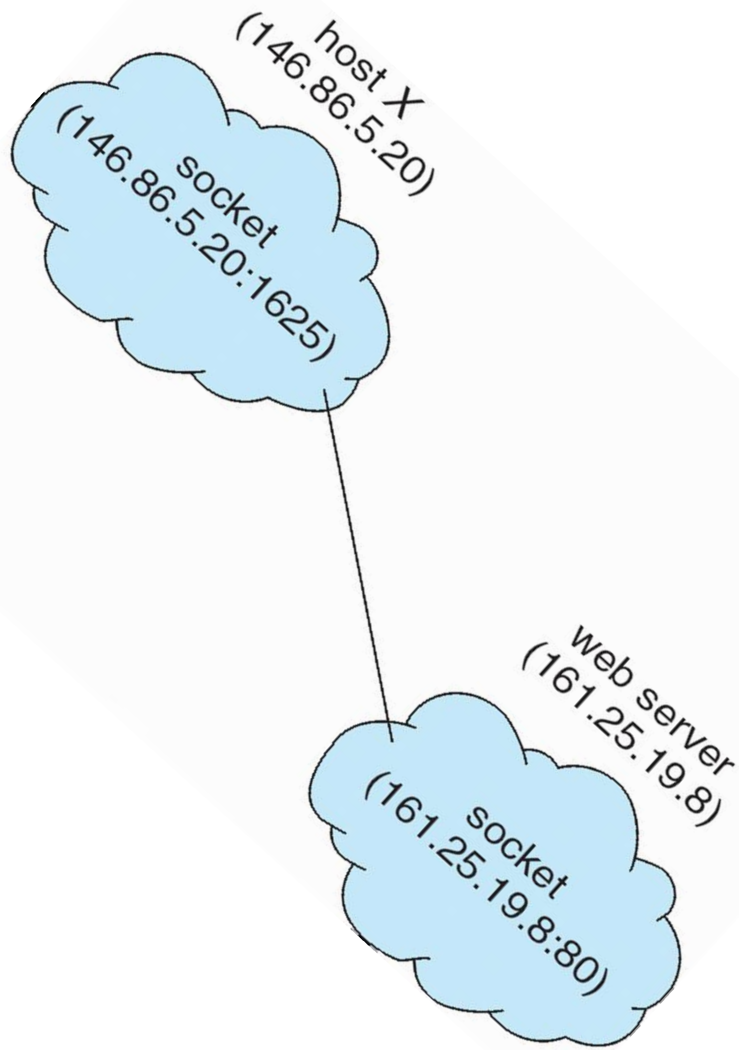
Remote Procedure Calls (RPC)

Remote Method Invocation (RMI) Java

COMMUNICATIONS IN CLIENT-SERVER SYSTEMS

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



IPC with Message Passing (socket)

- C/C++ (sys/socket.h, netinet/in.h)
  - Server
    - Create a socket with the `socket()`
    - Bind the socket to an address using the `bind()`
    - Listen for connections with the `listen()`
    - Accept a connection with the `accept()` system call.
  - Client
    - Create a socket with the `socket()` system call
    - Connect to server using the `connect()` system call
    - `read()` and `write()`
- Java
  - Server: `ServerSocket`
  - Client: `Socket`

Socket Communication in Java

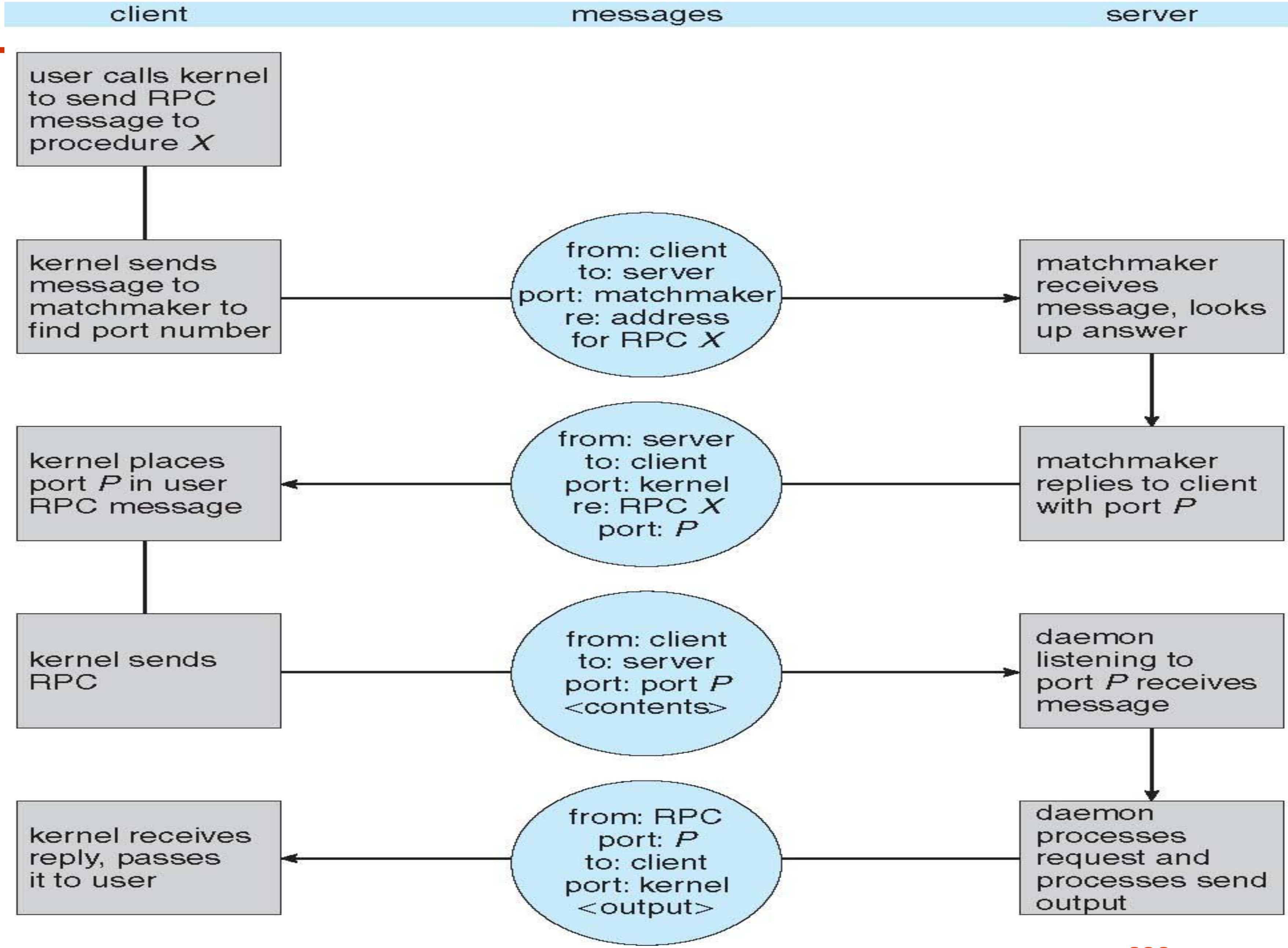
```
public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         // now listen for connections
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            // write the Date to the socket
            pout.println(new java.util.Date().toString());

            // close the socket and resume
            // listening for connections
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

```
public class DateClient
{
   public static void main(String[] args) {
      try {
         //make connection to server socket
         Socket sock = new Socket("127.0.0.1",6013);

         InputStream in = sock.getInputStream();
         BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));

         // read the date from the socket
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         // close the socket connection
         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Remote Procedure Calls

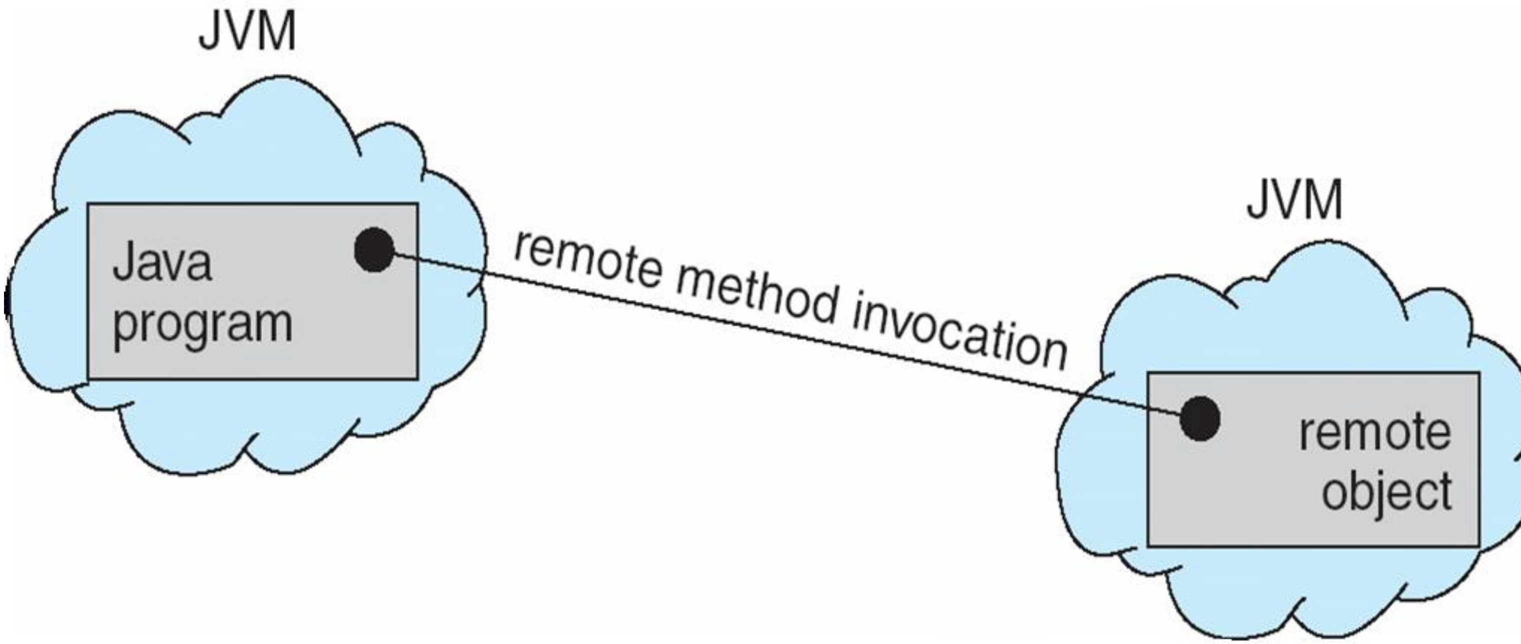
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and *marshalls* the parameters
- The server-side stub receives this message, unpacks the marshalled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

Execution of RPC

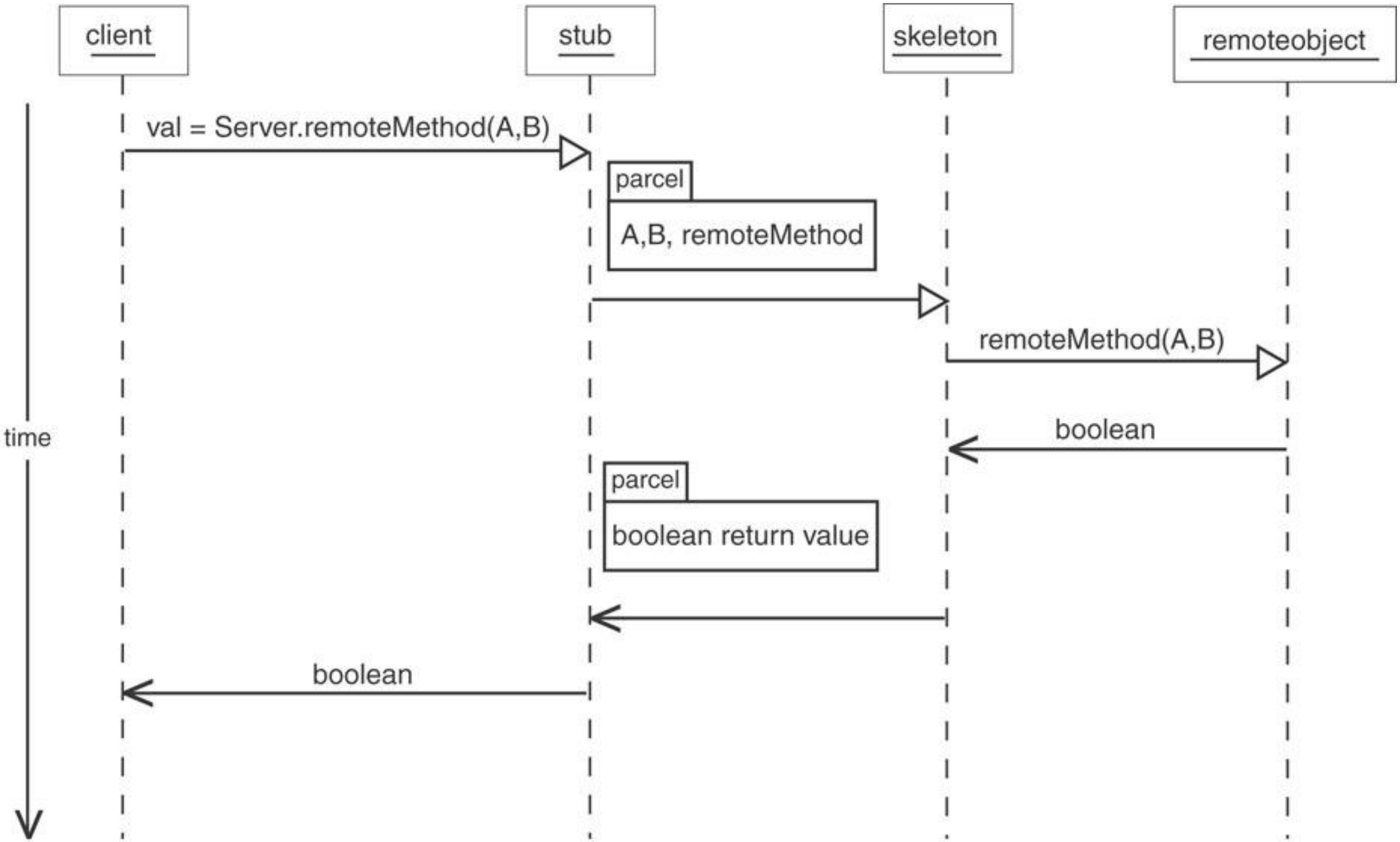


Remote Method Invocation

- 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 (UML seq. diagram)



RMI

- Remote Objects
- Access to the Remote Object
- Running the Programs
  - rmiregistery &
- RMI versus RPC vs Sockets

RMI Example (Appendix D online)

```
public interface RemoteDate extends Remote
{
   public abstract Date getDate() throws RemoteException;
}
```

```
public class RemoteDateImpl extends UnicastRemoteObject
      implements RemoteDate
{
   public RemoteDateImpl() throws RemoteException { }

   public Date getDate() throws RemoteException {
      return new Date();
   }

   public static void main(String[] args) {
      try {
         RemoteDate dateServer = new RemoteDateImpl();

         // Bind this object instance to the name "DateServer"
         Naming.rebind("DateServer", dateServer);
      }
      catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

```
public class RMIClient
{
   public static void main(String args[]) {
      try {
         String host = "rmi://127.0.0.1/DateServer";

         RemoteDate dateServer = (RemoteDate)Naming.lookup(host);
         System.out.println(dateServer.getDate());
      }
      catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

Mach: Message passing

Windows XP : Shared Memory

EXAMPLES OF IPC SYSTEMS

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()`

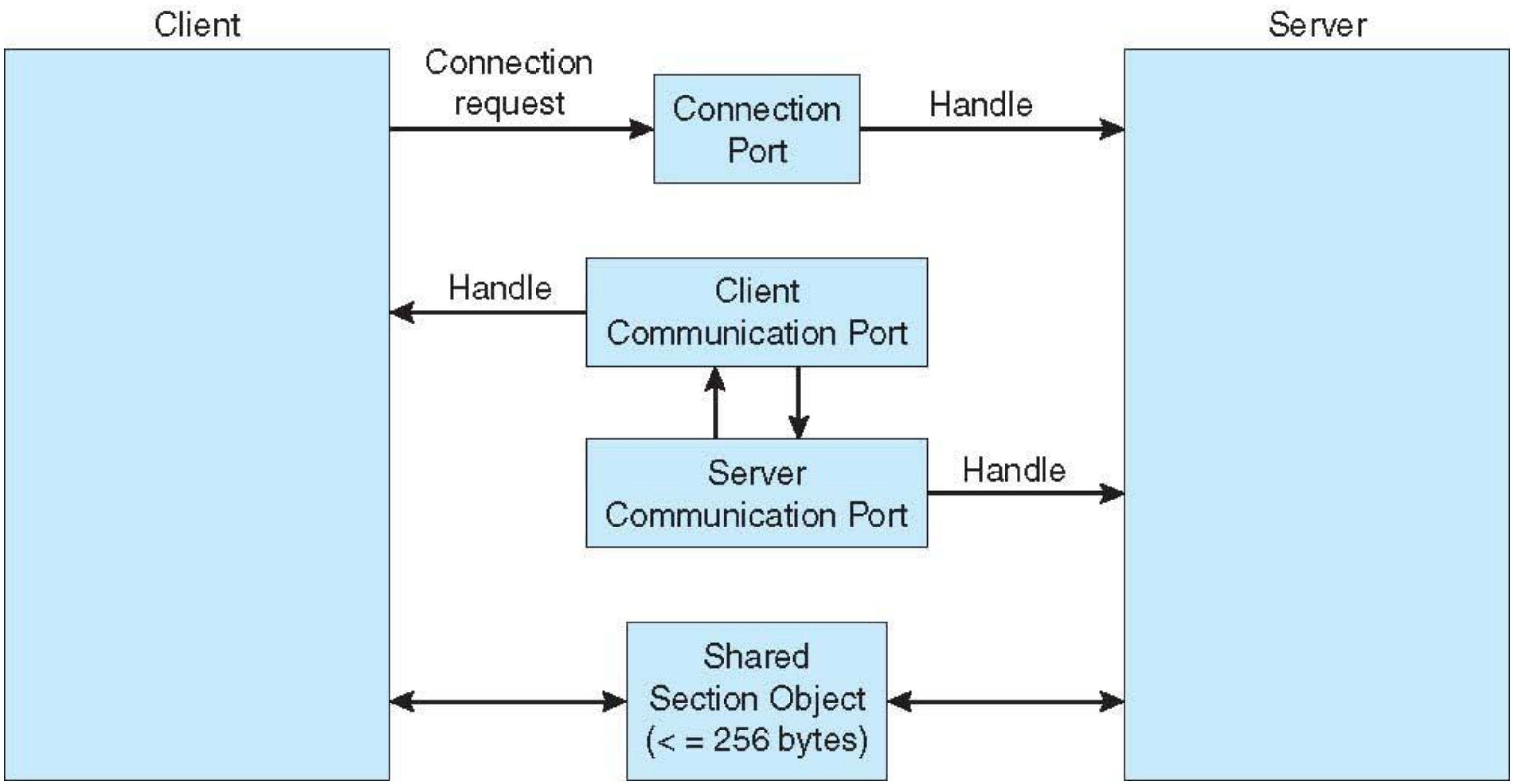
  - Mailboxes needed for commuication, created via

    `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



End of Chapter 3