Chapter 3: Process Concept

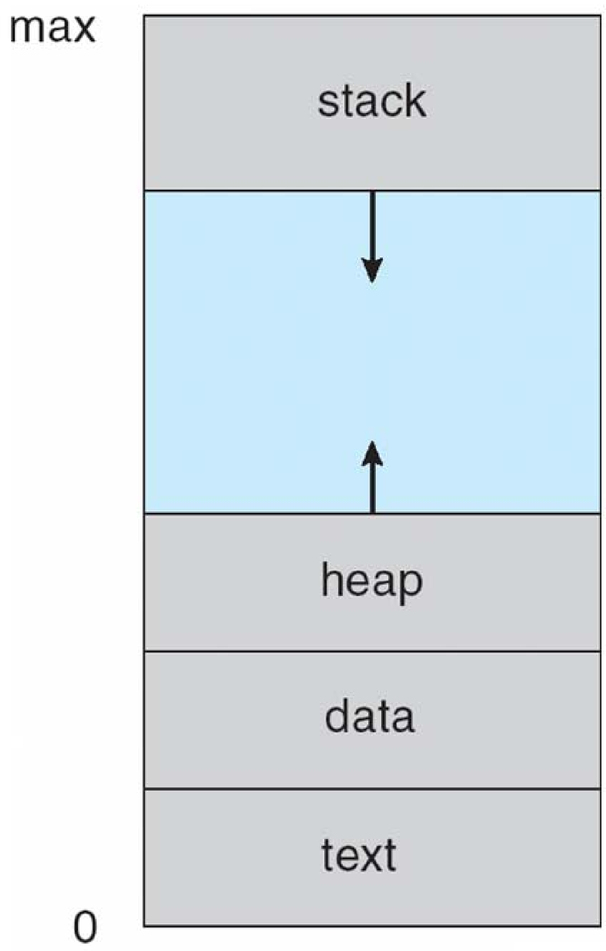
- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

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
- **Program** is ***passive*** entity stored on disk (**executable file**), **process** is ***active***
  - **Program** becomes **process** when executable file loaded into memory
  - **Process** – a program in execution
- Execution of **program** **started via** ***GUI mouse clicks***, ***command line entry of its name***, etc
- A **program** (e.g. notepad.exe) may be **executed several times** =>
  - different processes run currently
  - each process has its own ***data***, ***stack***, and ***heap***

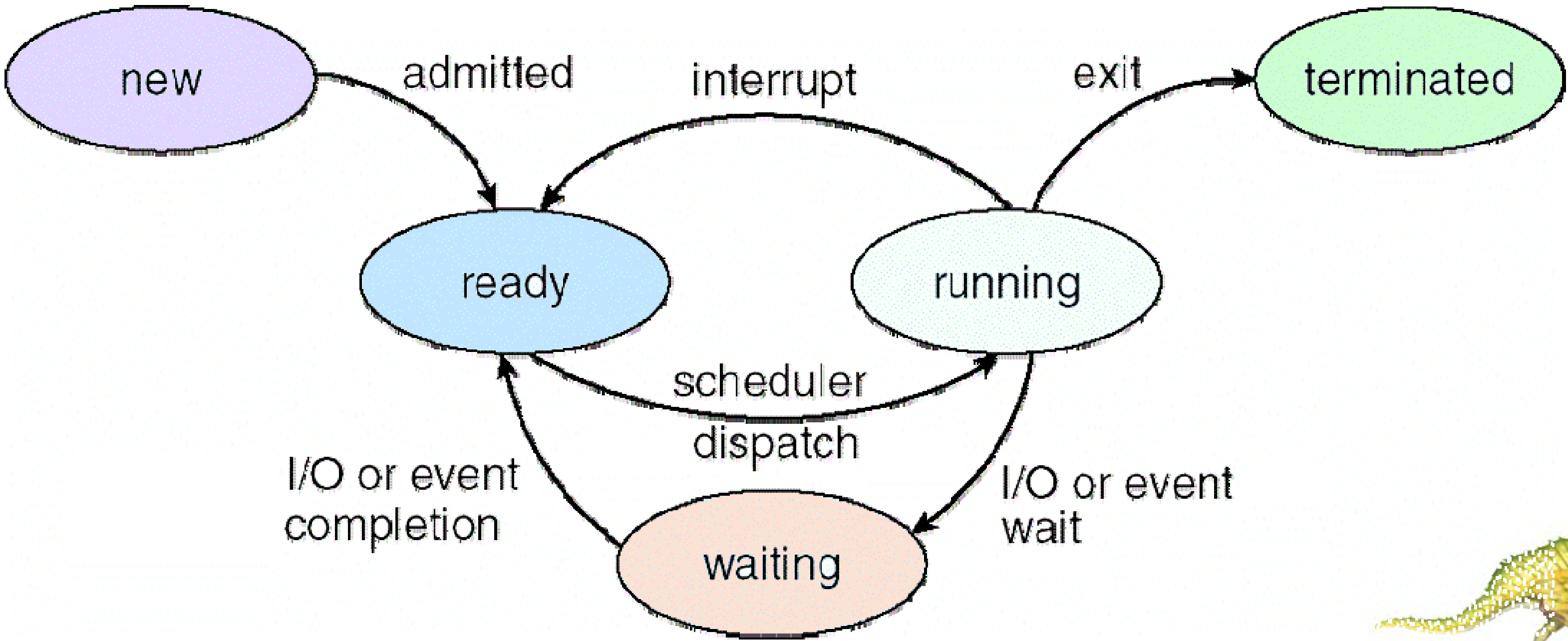
Process in Memory

- A **process** includes:
  - **program counter**, **registers**
  - **stack**
    - containing *function parameters*, *return addresses*, *automatic variable*
  - **heap**
    - containing *memory dynamically allocated* during run time
  - **data** (*static variables*, *constants*)
  - **text** (*object code*)
- A **object program file** just includes **data** and **text** sections



Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: **Instructions** (object code) are being executed (occupy **processor**)
  - **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)

Information associated with each process

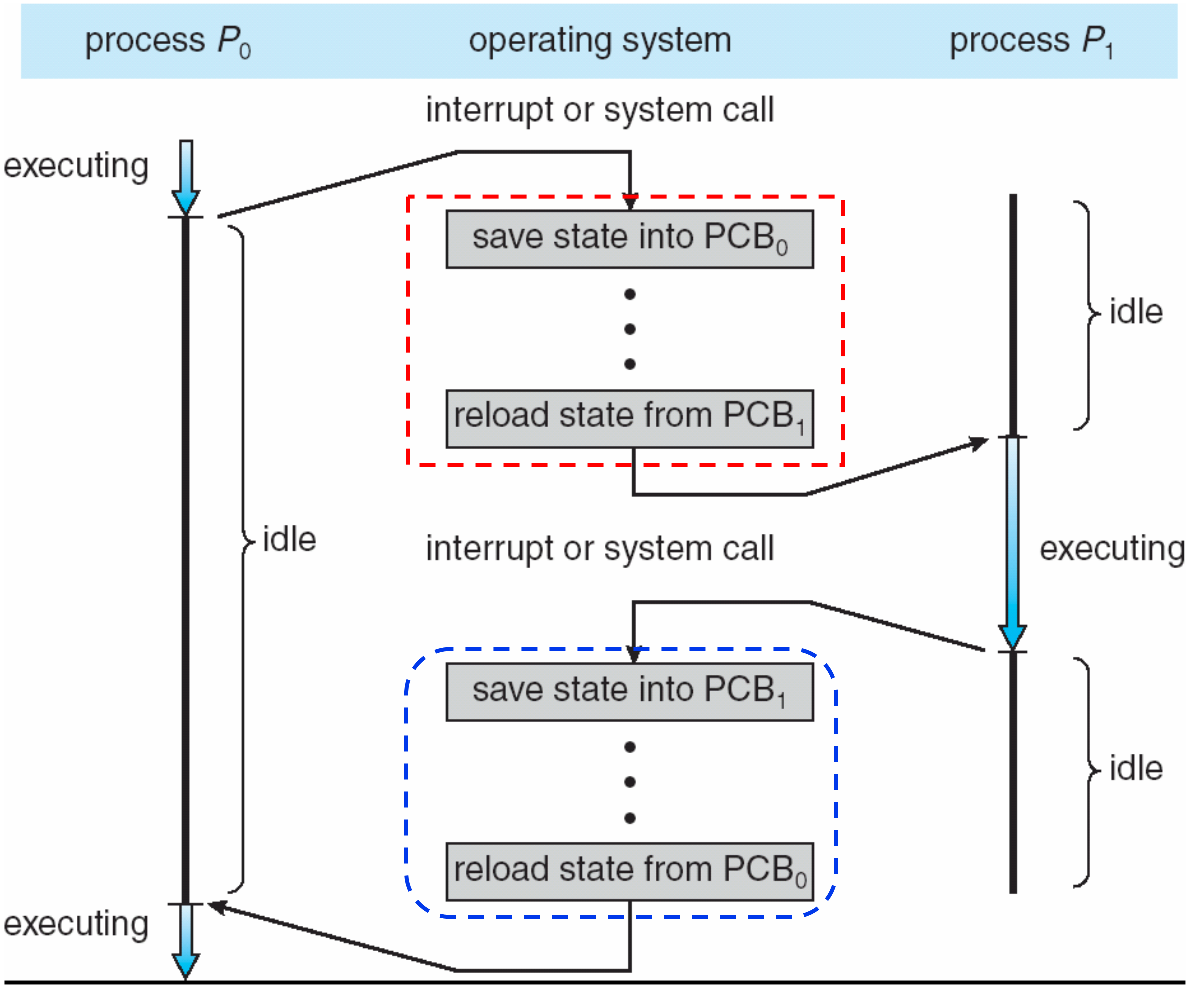
(also called **task control block**)

- **Process state** – running, waiting, etc
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling information** - priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files

pid (process id)

base, limit registers

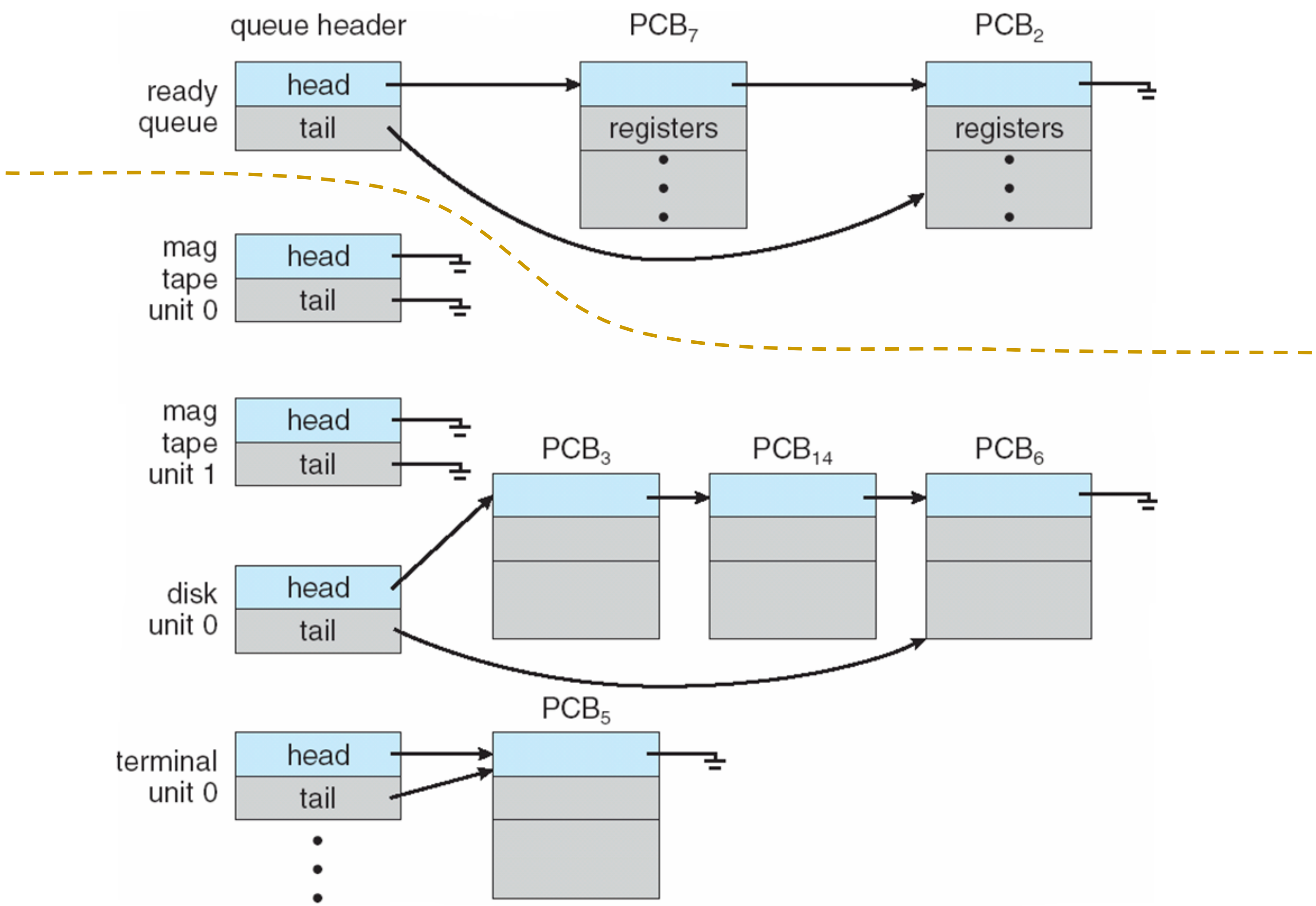
CPU Switch From Process to Process



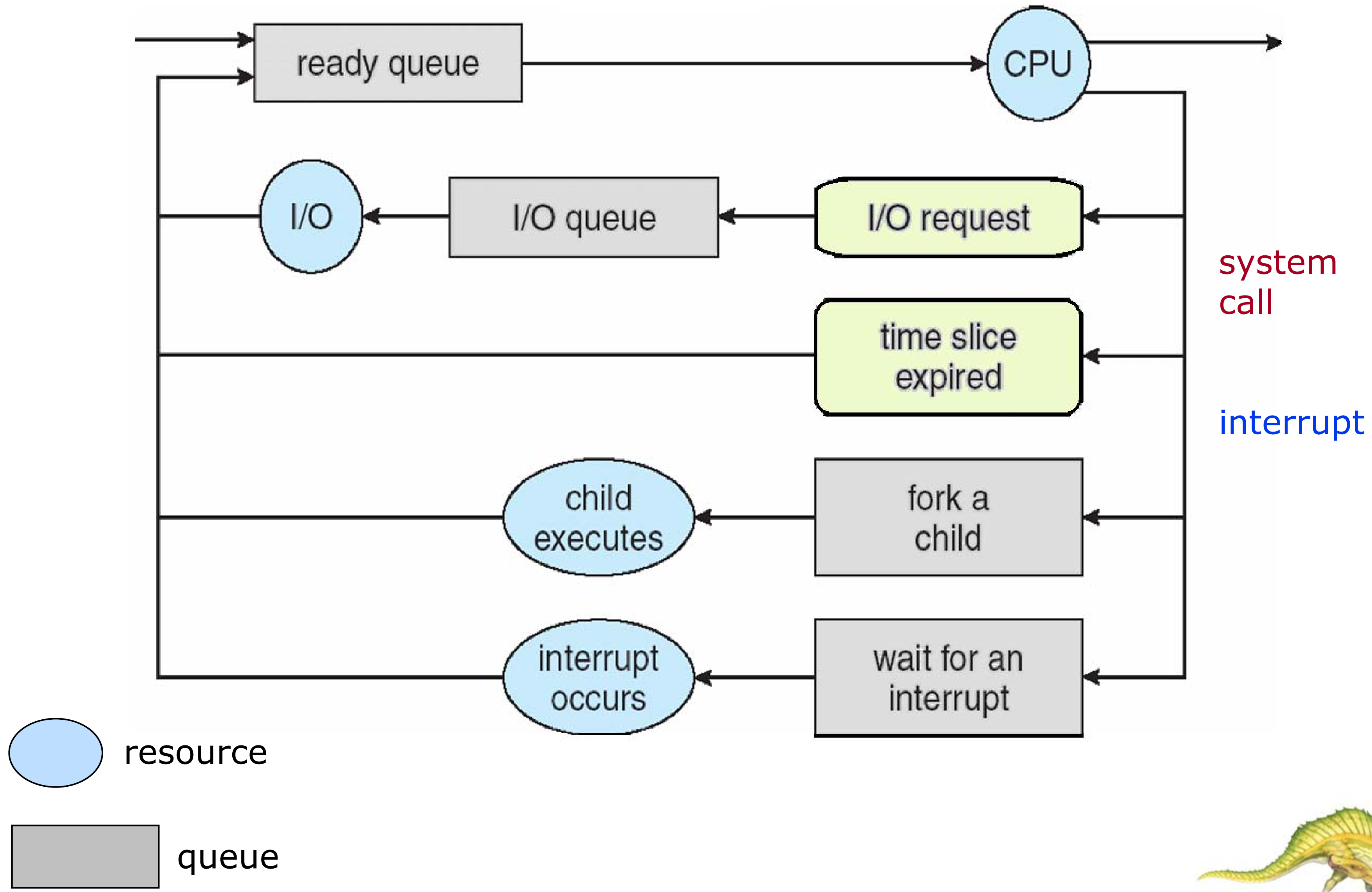
Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes. Processes migrate among the various queues
  - **Job queue** – set of all jobs in the system
    - **batch :** executes commands when **system load level** permit, e.g. the **load average** drops below 0.8
  - **Ready queue** – set of all processes residing in main memory, **ready** and waiting to execute (**wait CPU**)
  - **Device queues** – set of processes waiting for an I/O device
  - **Event queues** – waiting for an event to occur, e.g. timer, semaphore

Ready Queue & I/O Device Queues



Queueing-diagram of Process

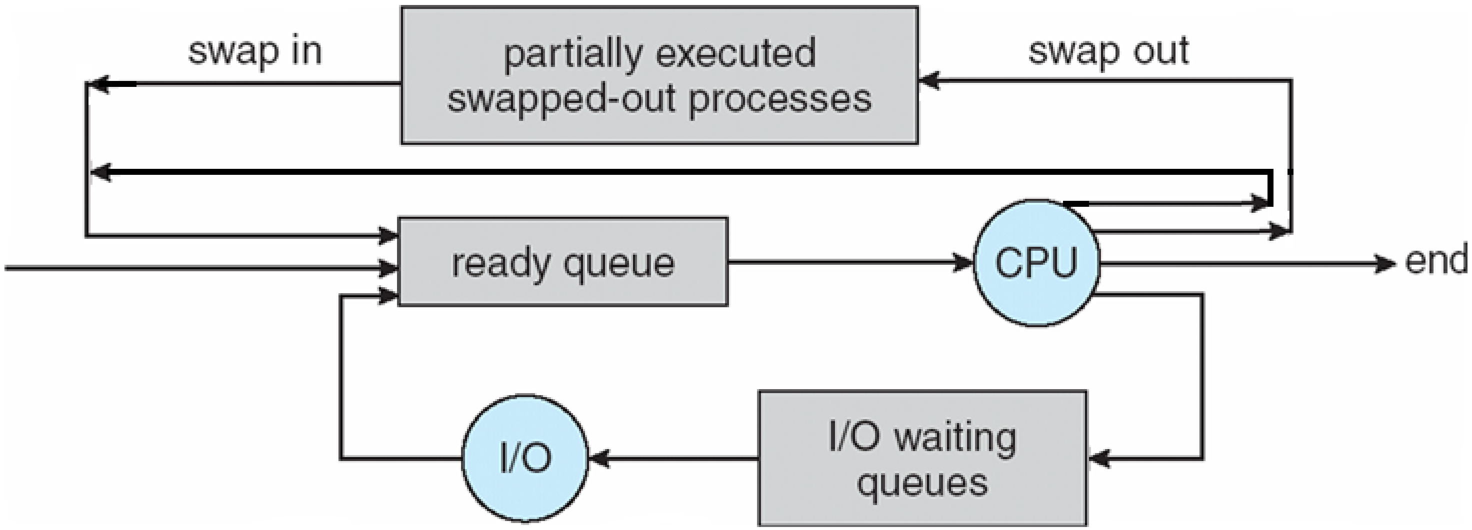


Schedulers

- **Long-term scheduler** (or job scheduler)
  - **selects** which **processes** should be brought into the **ready queue**
  - controls the **degree** of **multiprogramming** (but by user in MS windows)
- **Short-term scheduler** (or **CPU** scheduler)
  - **selects** which process should be executed next and **allocates CPU**
  - **invoked very frequently => must be fast**.
    - If it take 10 ms for 100 ms, then 10 / (10+100) = **9%**

Medium Term Scheduling

- **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
- **Remove process** from memory, **store on disk**, **bring back in** from disk to continue execution (swapping)
- To **improve** the **process mix** in some **timesharing systems**



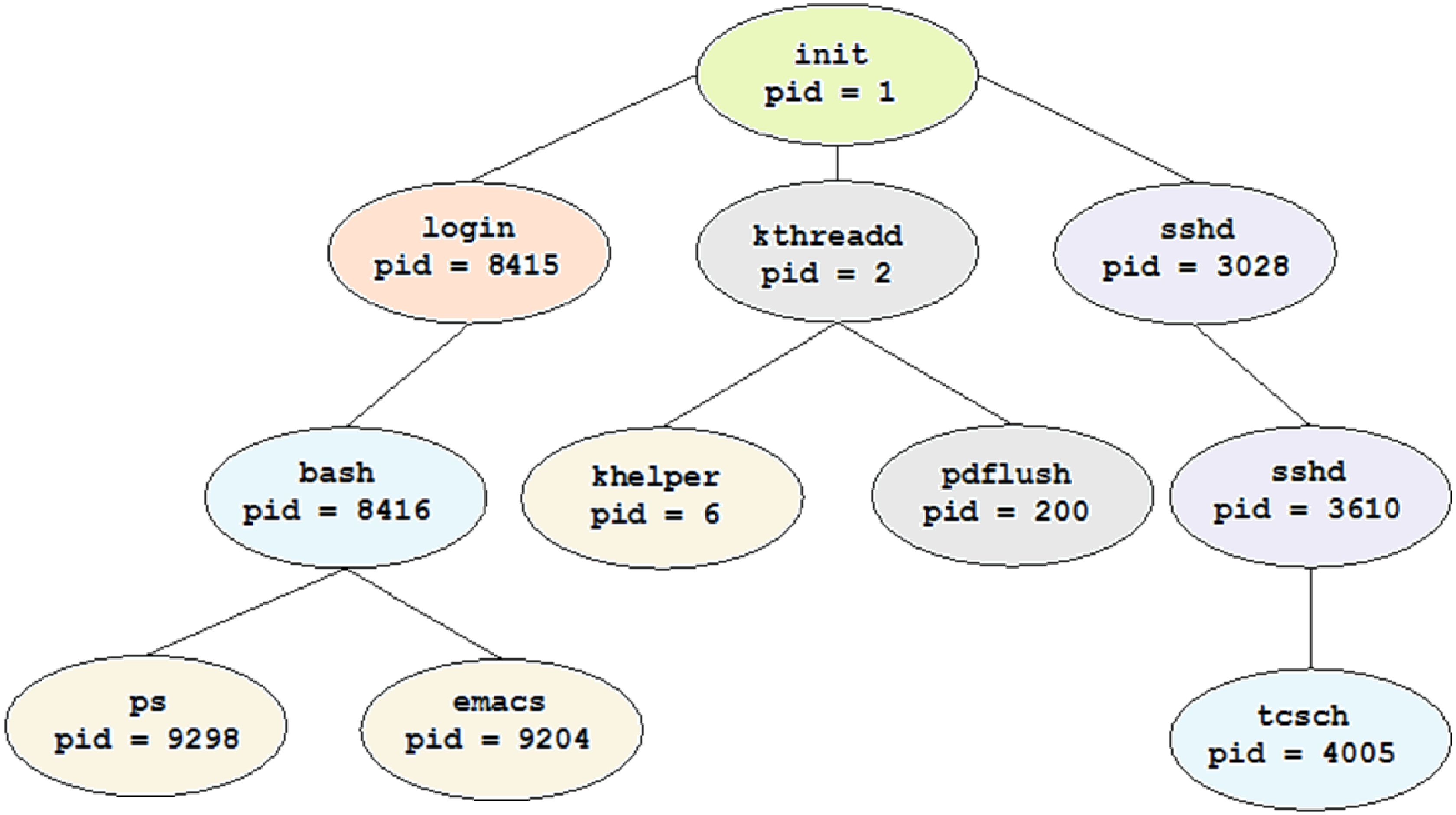
Context Switch

- 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
  - The more **complex** the OS and the PCB => longer the context switch
- **Time dependent** on **hardware support**
  - Some hardware provides **multiple sets of registers** per CPU => multiple contexts loaded at once

Process Creation (1)

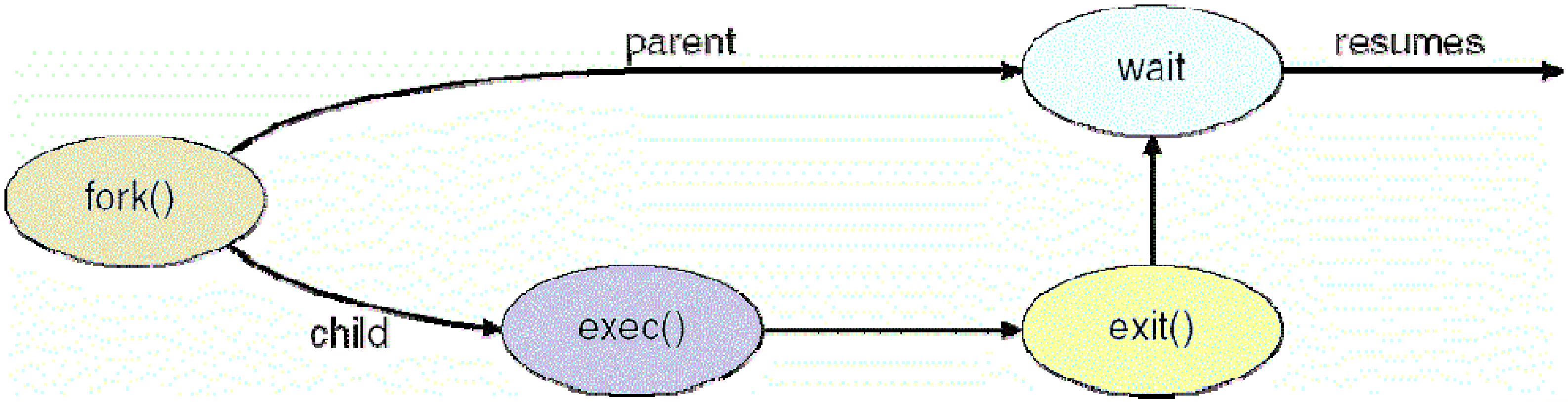
- **Parent** process create **children** processes, which, in turn create other processes, forming **a tree of processes**
- **Process** identified and managed via a **process identifier** (**pid**)
- **Resource sharing**
  - Parent and children **share all resources**
  - Children **share subset** of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

A tree of processes on a Linux



Process Creation (2)

- **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



C Program Forking Separate Process

Parent process

```
#include <unistd.h>
int main()
{
    pid_t  pid;

    /* fork another 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);
    }
}
```

Child process

Creating Separate Process Using Win32 API

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

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

    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 starting 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 Termination

- **Process** executes last statement and asks the **OS** to delete it, e.g. **exit**(1)
  - Output data from child to parent (via **wait()**)
    - `pid = wait(&status);`
  - Process' resources are deallocated by operating system
- **Parent** may terminate **children processes**, e.g. **kill**(pid, signal)
  - Child has exceeded **allocated resources**
  - Task assigned to child is **no longer required**
  - If parent is exiting
    - <u>Some OS do not allow child to continue if its parent terminates</u>
      - All children terminated -- **cascading termination**
- **If no parent waiting**, then terminated process is a **zombie**
  - Its entry in **process table** must remain there until the parent call **wait()**
- **If parent terminated**, process is **orphan** **(**assgning ***init*** as new parent)
  - e.g., ***nohup*** *abc &*

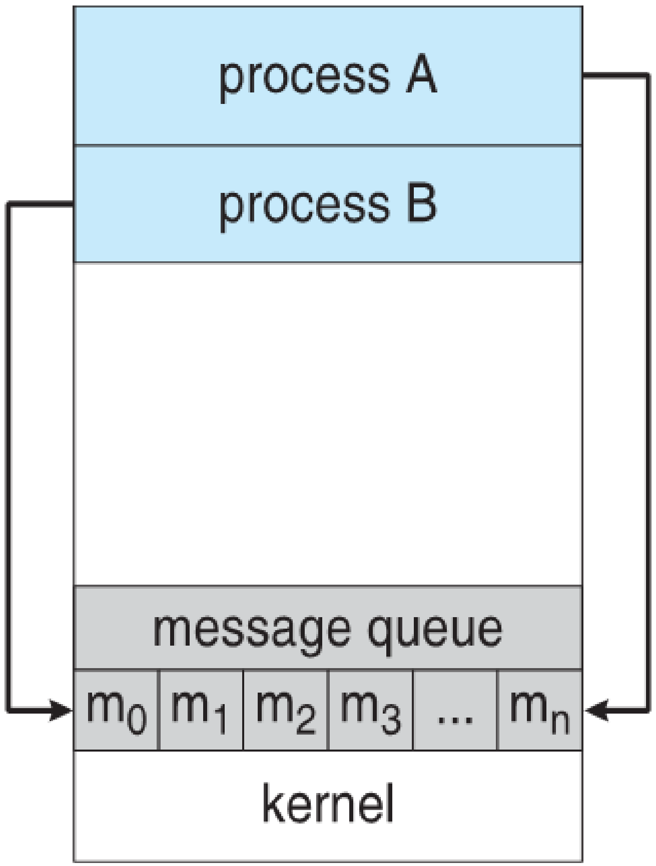
Multiprocess Architecture – Chrome Browser

- Many **web browsers** ran as single process
  - If one web site causes trouble, entire browser can **hang** or **crash**
- **Google Chrome Browser** is multiprocess with 3 categories
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript, **new one for each website opened**
    - Run in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

Interprocess Communication

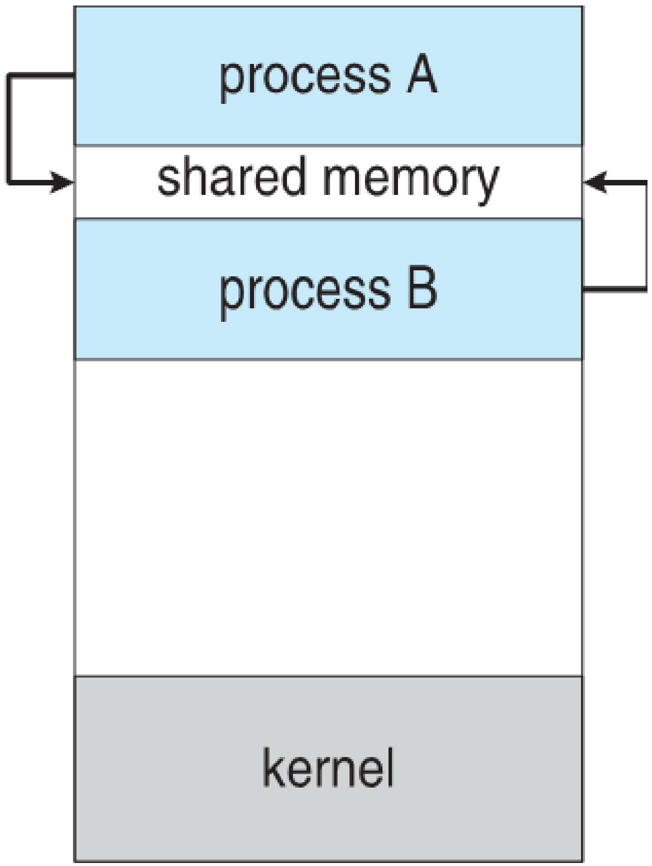
- Processes within a system may be **independent** or **cooperating**
- **Reasons** for **cooperating processes**:
  - Information sharing
  - Computation speedup
  - Modularity (e.g. ls -l | grep abc)
  - Convenience
- **Cooperating process** can **affect** or be **affected** by other processes, including sharing data
- **Cooperating processes** need **interprocess communication (IPC)**
- **Models of IPC**
  - **Shared memory**
  - **Message passing**

Communications Models



(a)

message passing



(b)

shared memory

Producer-Consumer Problem

- **Paradigm** for **cooperating processes**
  - ***producer*** process produces information that is consumed by a ***consumer*** process
  - ***bounded-buffer*** assumes that there is a **fixed buffer size**
- **One Solution : Bounded-Buffer on Shared-Memory**

```
 /* Declaration of the shared data */
#define BUFFER_SIZE  10
typedef struct {
   . . .
} item;

struct  shrd {
     item buffer[BUFFER_SIZE];
     int  in = 0;
     int  out = 0;
} *mp;
```

```
mp = shmat( sizeof(shrd) );
mp->in ++;
```

Bounded-Buffer Producer

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

```
item  nextProduced ;
while (true) {
     /* Produce an item in nextProduced */

    while (  ( (in + 1) % BUFFER_SIZE)  == out  )
         ;   /* do nothing -- no free buffer */

    buffer[ in ] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Bounded-Buffer Consumer

```
item  nextConsumed ;
while (true)  {

     while (in == out)
          ;  // do nothing

     nextConsumed = buffer[ out ];
     out = (out + 1) % BUFFER_SIZE;
     /* consume the item in nextConsumed */
}
```

IPC – Message Passing

- **Mechanism** for processes to communicate and to synchronize their actions
- **Message system** – processes communicate with each other without resorting to shared variables
- **Message passing** 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***, and exchange messages via **send**/**receive**
- **Implementation** of **communication link**
  - **direct** or **indirect** communication, i.e. **naming**
  - **Synchronous** or **asynchronous** communication
  - automatic or explicit **buffering**

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
  - If the name of *P* or *Q* changed, all such names must be changed
- Properties of **communication link**
  - Links are established **automatically**
  - A link is associated with exactly one pair of communicating processes
  - The link may be **unidirectional**, but is usually **bi-directional**

Indirect Communication

- **Messages** are sent to and received from **mailboxes** (ports)
  - Each **mailbox** has a unique **id**
  - Processes can communicate only if they share a mailbox
- **Primitives** are defined as:

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

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

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.

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

    e.g. SendMessage(hWnd, MsgID, WParam, LParam)

  - **Blocking receive** has the receiver block until a message is available

    e.g. GetMessage(lpMsg, hWnd, FirstMsgID, LastMsgID)

- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** has the sender send the message and continue

    e.g. PostMessage()

  - **Non-blocking receive** has the receiver receive a valid message or null

    e.g. PeekMessage()

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: POSIX Shared Memory

- Process first **creates** shared memory segment

```
shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);
```

  - Also used to open an existing segment to share it

- **Set the size** of the object

```
ftruncate(shm_fd, 4096);
```

- **Map** the shared memory object

```
ptr = mmap(0,4096,PROT_WRITE,MAP_SHARED,shm_fd,0);
```

- Now the process could **write to** the shared memory

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

IPC POSIX Producer

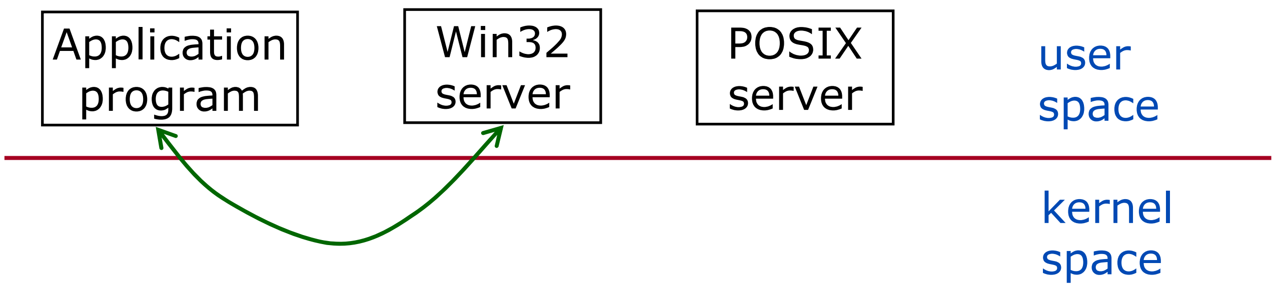
```c
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main()
{
        const int SIZE = 4096;
        const char *name = "OS";
        const char *message0= "Studying ", *message1= "Operating Systems ";
        const char *message2= "Is Fun!";
        int shm_fd;
        void *ptr;
                /* create the shared memory segment */
        shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

                /* configure the size of the shared memory segment */
        ftruncate(shm_fd,SIZE);

                /* now map the shared memory segment in the address space of the process */
        ptr = mmap(0,SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
        if (ptr == MAP_FAILED) {
                printf("Map failed\n");     return -1;
        }

        /* Now write to the shared memory region.
         * Note we must increment the value of ptr after each write.  */
        sprintf(ptr,"%s",message0);    ptr += strlen(message0);
        sprintf(ptr,"%s",message1);    ptr += strlen(message1);
        sprintf(ptr,"%s",message2);    ptr += strlen(message2);
        return 0;
}
```

IPC POSIX Consumer

```
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main()
{
        const char *name = "OS";
        const int SIZE = 4096;
        int shm_fd;
        void *ptr;
        int i;
                /* open the shared memory segment */
        shm_fd = shm_open(name, O_RDONLY, 0666);
        if (shm_fd == -1) {
                printf("shared memory failed\n");           exit(-1);
        }

                /* now map shared memory segment in the address space of the process */
        ptr = mmap(0,SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
        if (ptr == MAP_FAILED) {
                printf("Map failed\n");                      exit(-1);
        }

                /* now read from the shared memory region */
        printf("%s",ptr);
                /* remove the shared memory segment */
        if (shm_unlink(name) == -1) {
                printf("Error removing %s\n",name);         exit(-1);
        }
        return 0;
}
```

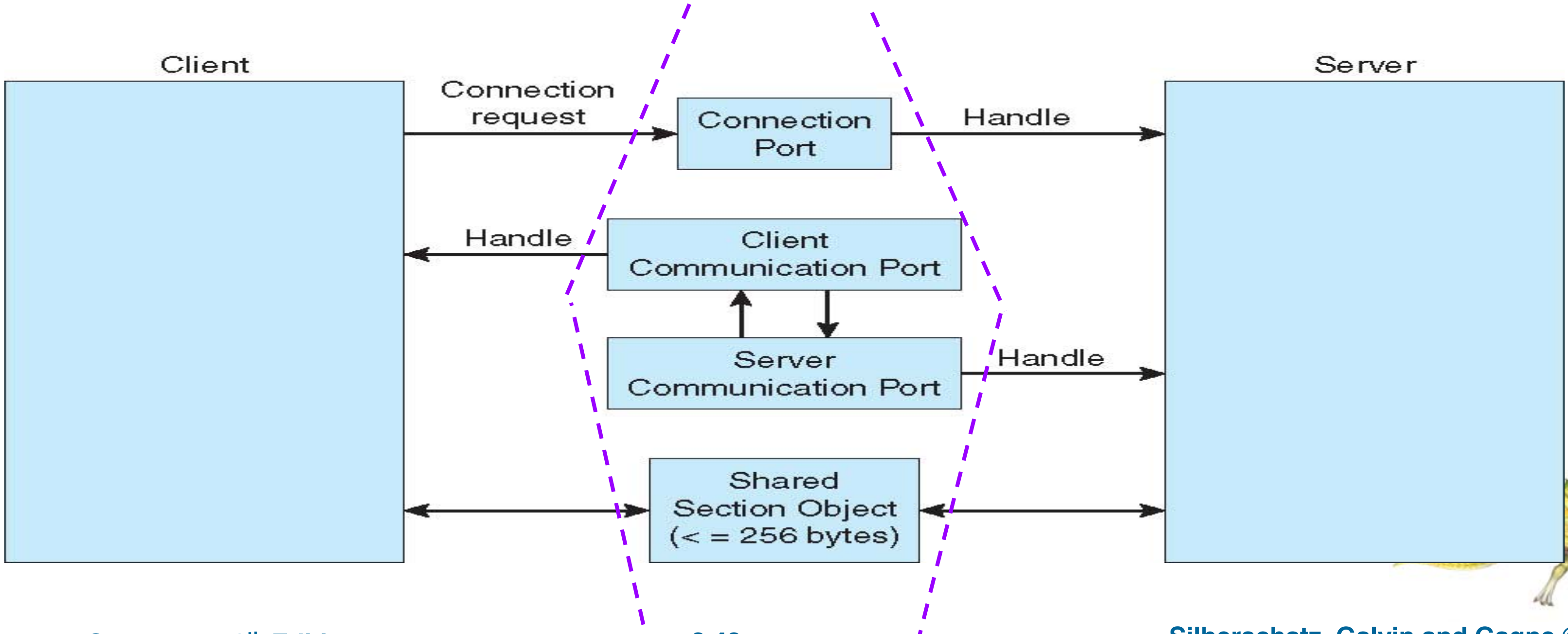
Example: Windows Message Passing

- **Windows** provides support for multiple subsystems (e.g. **Win32** server, **POSIX** server) with which application programs communicate via message passing
- **Message-passing** facility is called **advanced local procedure call** (ALPC)
  - Only works between processes on the same system
  - Uses **ports** (like mailboxes) to establish and maintain communication channels
- The ALPC facility in Windows is not part of the **Win32 API** (i.e. not visible to application programs)



Local Procedure Calls in Windows

- 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

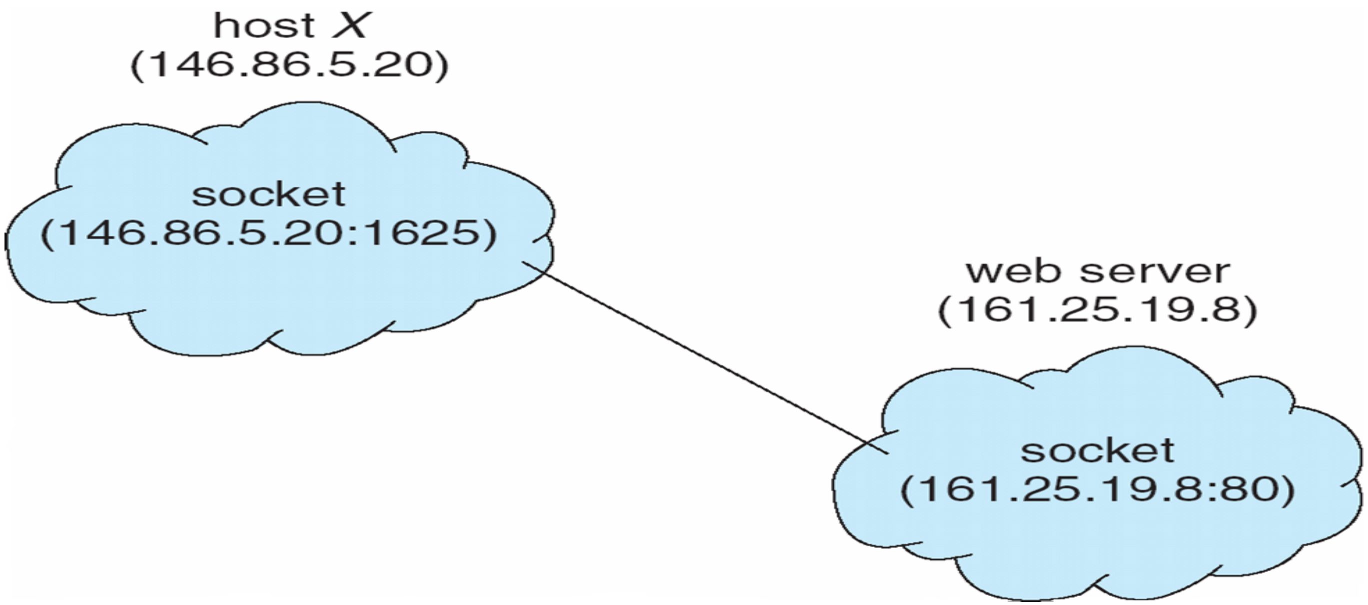


Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes

Socket Communication

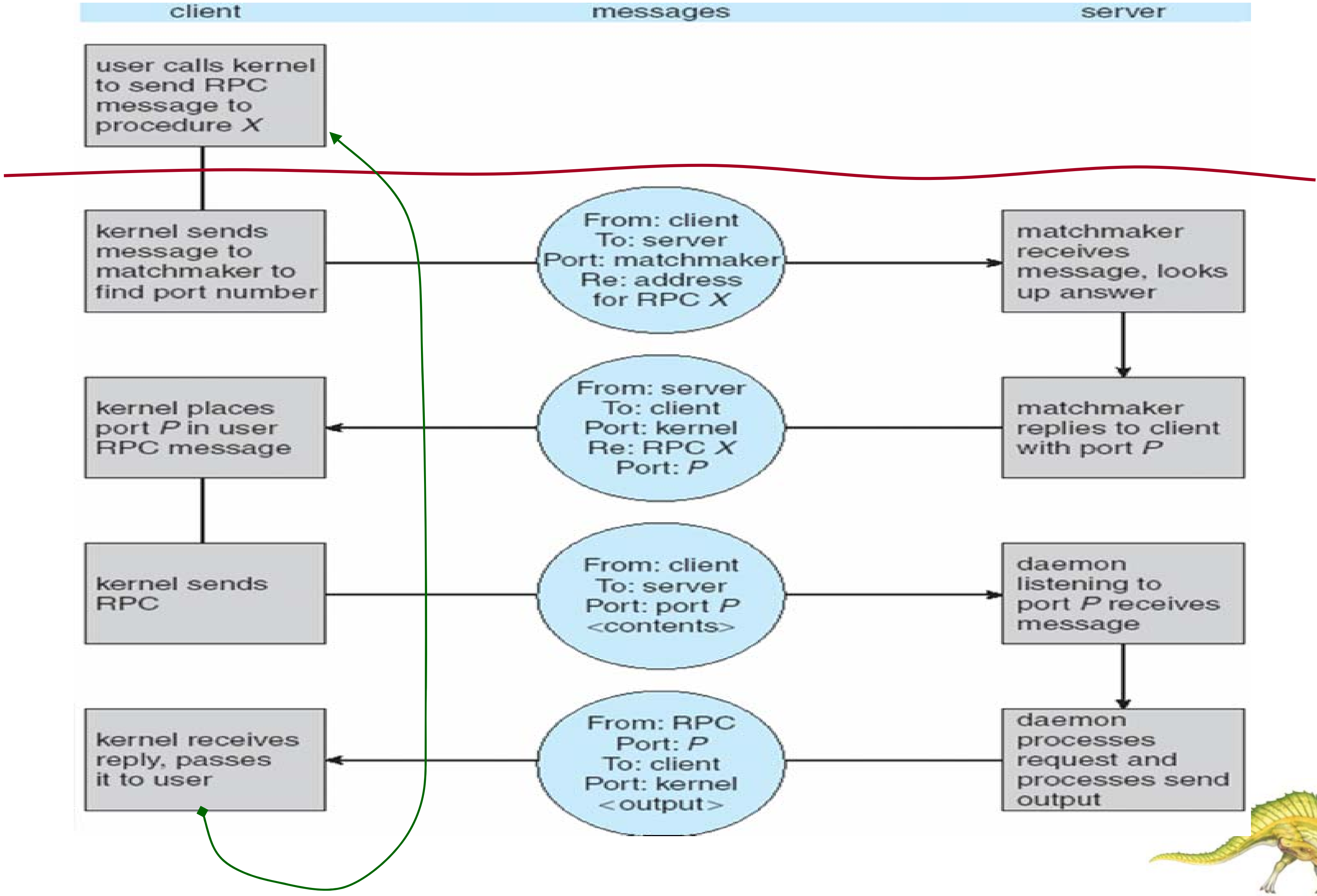
- A socket is defined as an *endpoint for communication*
- Concatenation of **IP** address and **port**
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets



Remote Procedure Calls

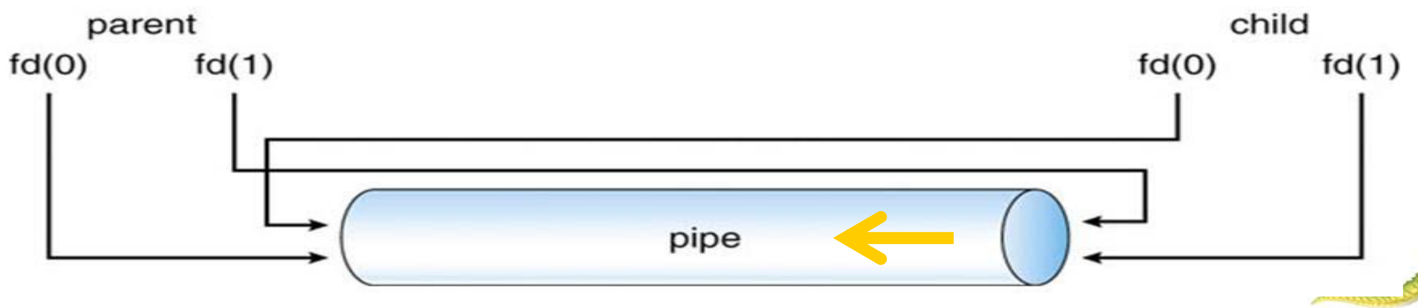
- **Remote procedure call** (**RPC**)
  - abstract procedure calls between **processes** on **networked systems**
  - **Stubs** – client-side proxy for the actual **procedure** on the server
- The **client-side stub locates** the **server** and ***marshalls*** the **parameters**
- The **server-side stub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

Execution of RPC



Pipes

- A **pipe** acts as a conduit allowing two processes to communicate. One of the first **IPC mechanisms** in early **UNIX** systems.
- **Ordinary pipes** (e.g. ls -l | grep abc)
  - **unidirectional** -- **two pipes** must be used for **two-way communication**.
  - constructed using the **system call**, **pipe**(int **fd**[ ])
  - **fd[0]** is the descriptor for the **read-end**, **fd[1]** is for the **write-end**
  - Once the processes terminate the ordinary pipe ceases to exist
- **Named pipes (FIFOs)**
  - Once created, they appear as typical files in the file system, **mkfifo**()
  - Continue to exist after communicating processes have finished
  - Once established, several processes can use it for communication
- **UNIX** treats a **pipe** as a special type of file; thus, pipes can be accessed using ordinary system calls, **read()** and **write()**



Example: POSIX Pipe

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

#define BUFFER_SIZE 25
#define READ_END    0
#define WRITE_END   1

int main(void)
{
      char write_msg[BUFFER_SIZE] = "Greetings";
      char read_msg[BUFFER_SIZE];
      int fd[2];
      pid_t pid;

   /* create the pipe */
   if (pipe(fd) == -1) {
      fprintf(stderr,"Pipe failed");
      return 1;
   }

   pid = fork();     /* fork a child process */

   if (pid < 0) {  /* error occurred */
      fprintf(stderr, "Fork failed");
      return 1;
   }

   if (pid > 0) {  /* parent process */
      /* close the unused end of the pipe */
      close(fd[READ_END]);

      /* write to the pipe */
      write(fd[WRITE_END], write_msg,
strlen(write_msg)+1);

      /* close the write end of the pipe */
      close(fd[WRITE_END]);
   }
   else {                /* child process */
      /* close the unused end of the pipe */
      close(fd[WRITE_END]);

      /* read from the pipe */
      read(fd[READ_END], read_msg,
BUFFER_SIZE);
      printf("read %s", read_msg);

      /* close the write end of the pipe */
      close(fd[READ_END]);
   }

   return 0;
}
```