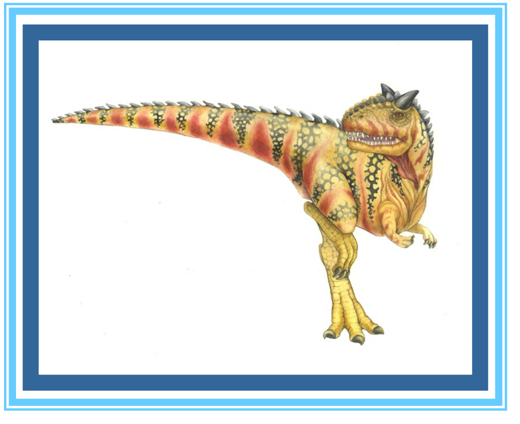
Lecture 3: Processes



Lecture 3: Processes

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

Objectives

- To Introduce Notion of a Process
- To Describe Various Features of Processes
  - Scheduling, Creation / Termination, & Communication
- To Explore Interprocess Communication
  - Shared Memory
  - Message Passing

Reminder

- **Early Computers**
  - Only one program in execution at a time
  - Program had complete control of system
- **Current Computers Allow**
  - Multiple programs to be loaded into memory
  - Multiple programs executed concurrently
- ➔ Notion of Process

Process Concept

- **Process** – a Program in Execution
  - Process execution must progress in sequential fashion
- A System Consists of a Set of Processes
  - OS processes executing system code
  - User processes executing user code
- Textbook Uses terms ***Job*** and ***Process*** almost Interchangeably
  - Batch system: **jobs**
  - Time-shared systems: **user programs** or **tasks**

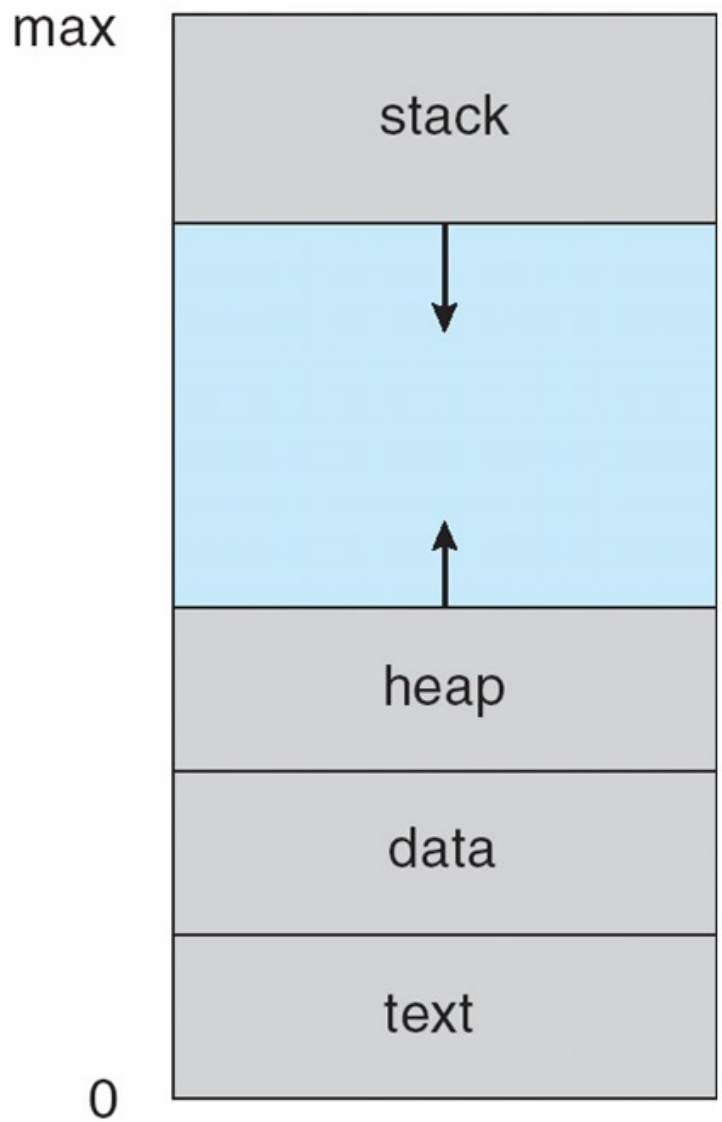
Process Concept (Cont.)

- Multiple Parts
  - Program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

Process Concept (Cont.)

- Program is ***Passive*** Entity Stored on Disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of Program Starts via:
  - GUI mouse clicks or
  - Command line entry of its name
- One Program can be Several Processes
  - Consider multiple users executing same program

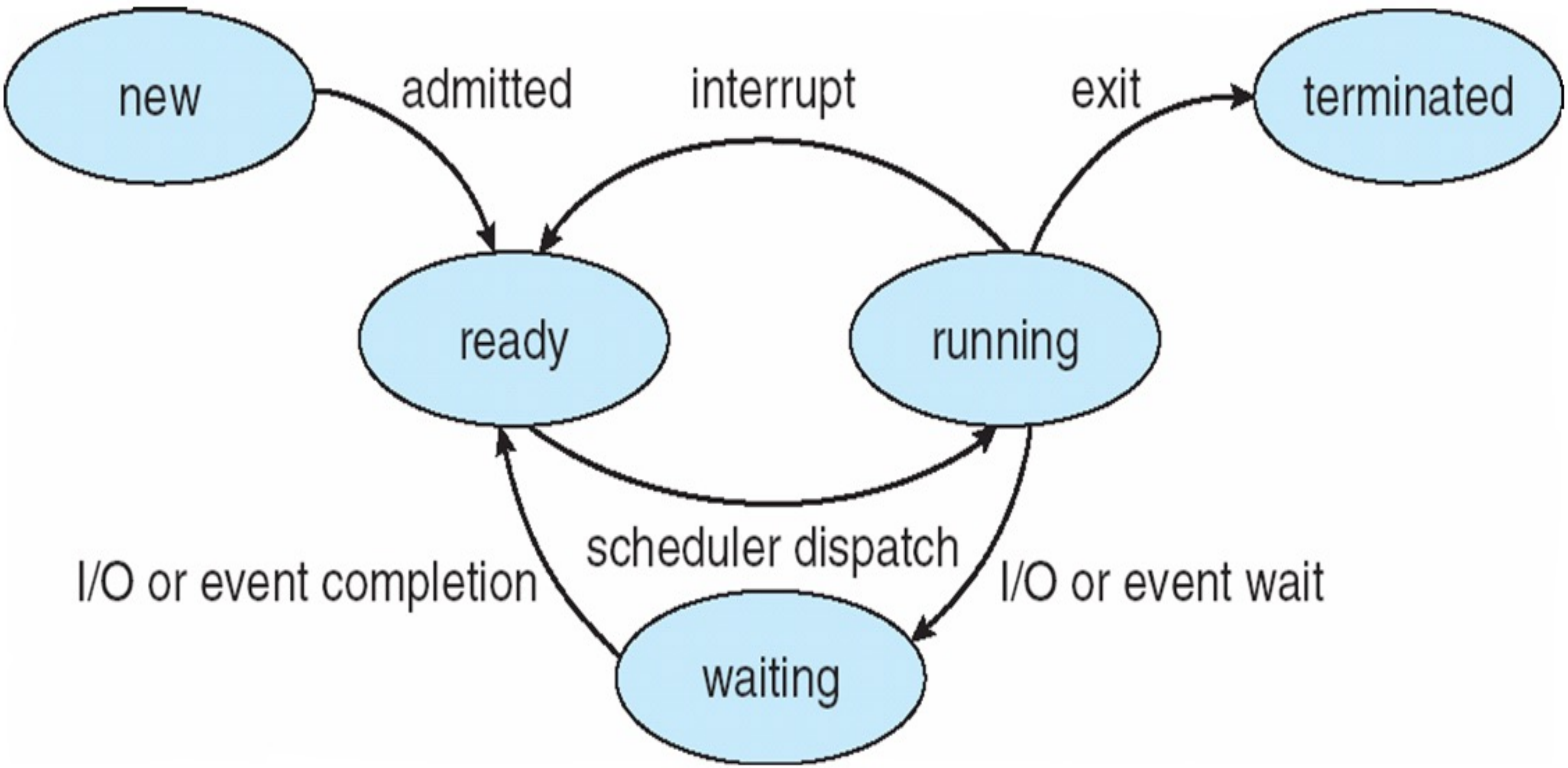
Process in Memory



Process State

- As a Process Executes, it Changes **State**
  - **New**: Process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: Process is waiting for some event to occur
  - **Ready**: Process is waiting to be assigned to a processor
  - **Terminated**: Process has finished execution

Diagram of Process State



Process Control Block (PCB)

- Info Associate with each Process
  - Aka, task control block
- Process State
  - Running, waiting, ready, new, & termin.
- Program Counter
  - Location of instruction to next execute
- CPU Registers
  - Contents of all process-centric registers: GP registers, stack register
- CPU Scheduling Info
  - Priorities, scheduling queue pointers

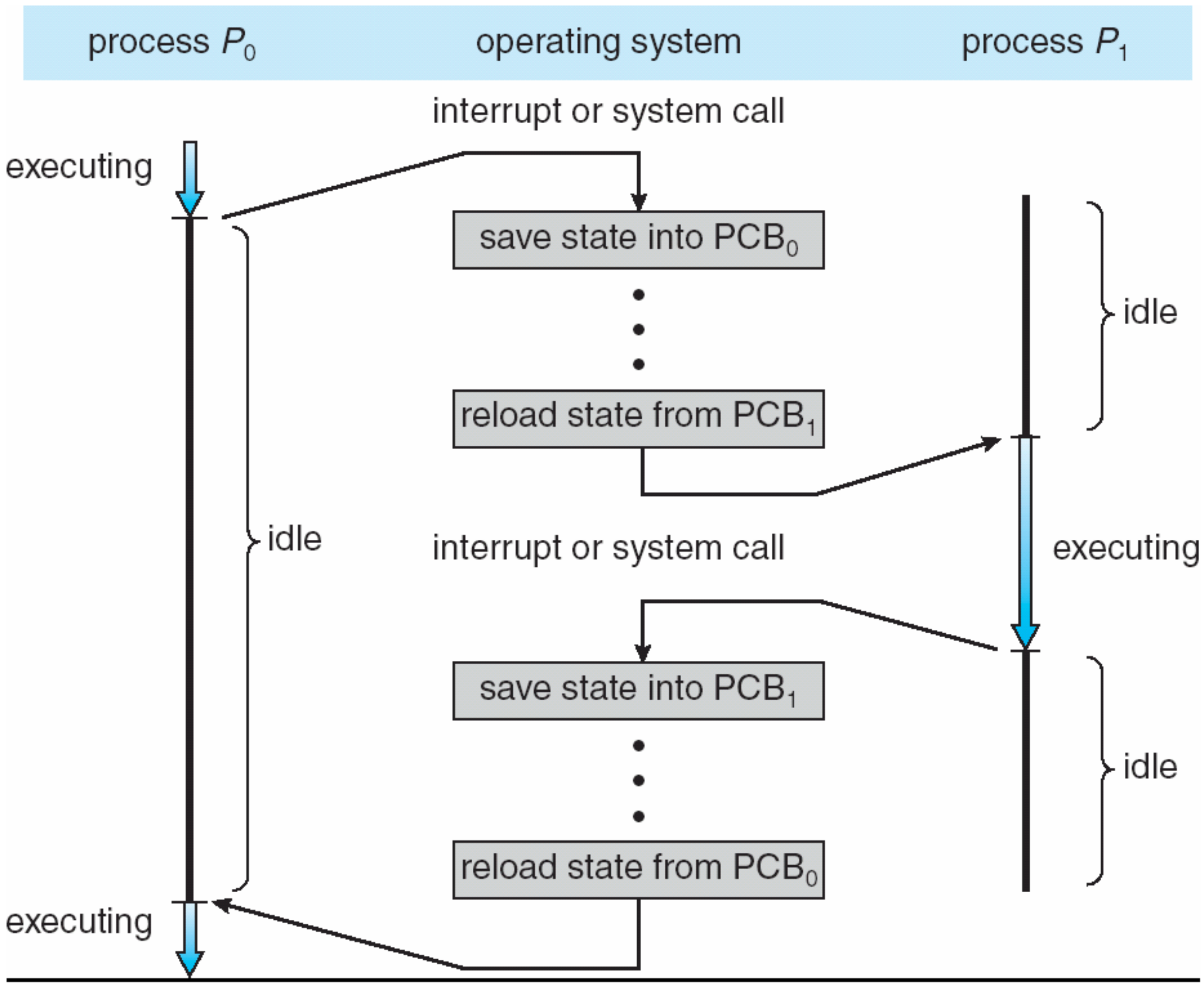
process state
process number
program counter
registers
memory limits
list of open files
• • •

Process Control Block (PCB) (cont.)

- Memory-Management Info
  - Memory allocated to process
  - Page or segment tables
- Accounting Info
  - CPU used
  - Clock time elapsed since start
  - Time limits
- I/O Status Info
  - I/O devices allocated to process
  - List of open files

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

CPU Switch From Process to Process



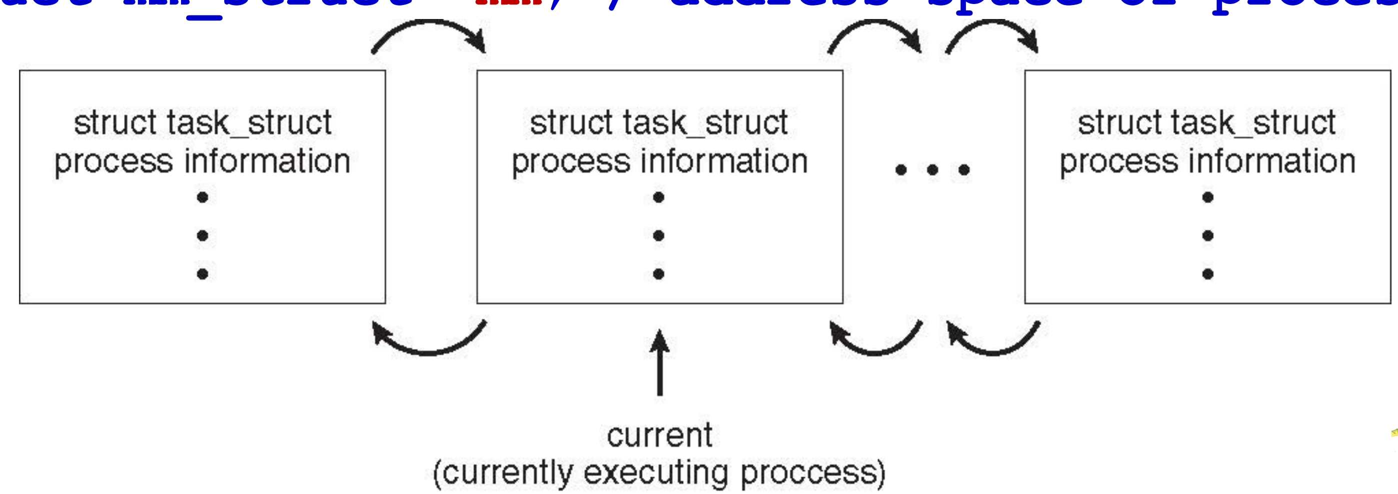
Threads

- Originally, Process has a Single Thread of Execution
  - E.g, : cannot do different searches with Mozilla at the same time (or typing & spell check)
- Consider Having **Multiple** Program Counters per Process
  - Multiple locations can execute at once
    - Multiple threads of control ➔ **threads**
- Must then Have Storage for Thread Details
  - Multiple program counters in PCB
  - Will be covered in detail next lecture

Process Representation in Linux

- **Doubly Linked** List of **Task_Struct**
  - **Kernel Maintains a Pointer to Current Process**

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



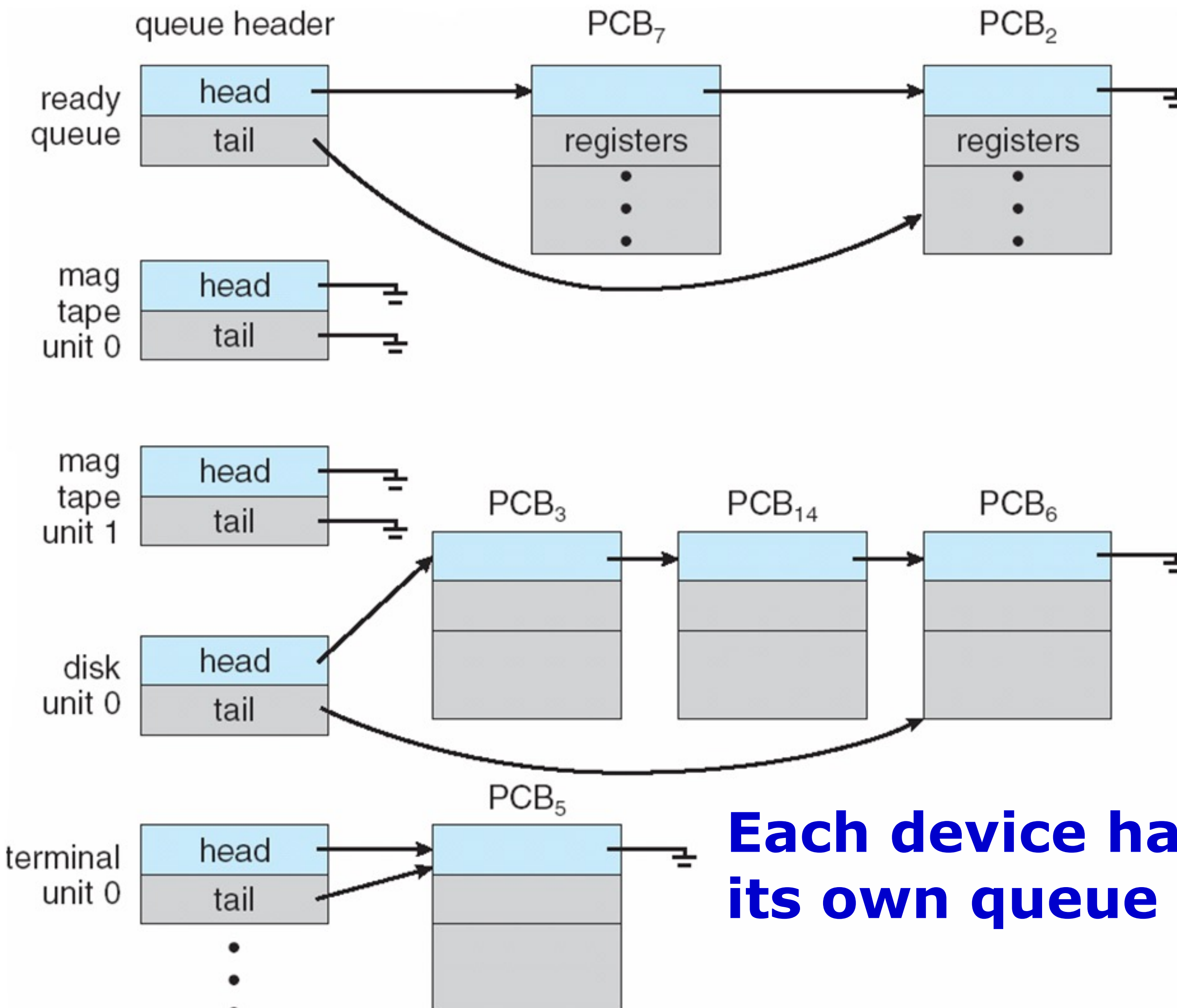
Process Scheduling

- **Process Scheduler**
  - Selects among available processes for next execution on CPU
- Possible Goals of Process Scheduling
  - To maximize CPU utilization
    - Quickly switch processes onto CPU for time sharing
  - To meet deadline of each process
    - In real-time applications
  - To meet fairness among processes

Process Scheduling (cont.)

- Maintains **Scheduling Queues** of Processes
  - **Job queue** – set of all processes in system
  - **Ready queue** – set of all processes residing in main memory, ready, and waiting to execute
    - Pointers to the 1st and final PCB in the list
  - **Device queues** – set of processes waiting for an I/O device (**one queue** for **each device**)
- Processes Migrate among Various Queues
  - Job queue ➔ ready queue
  - Device queue ➔ ready queue

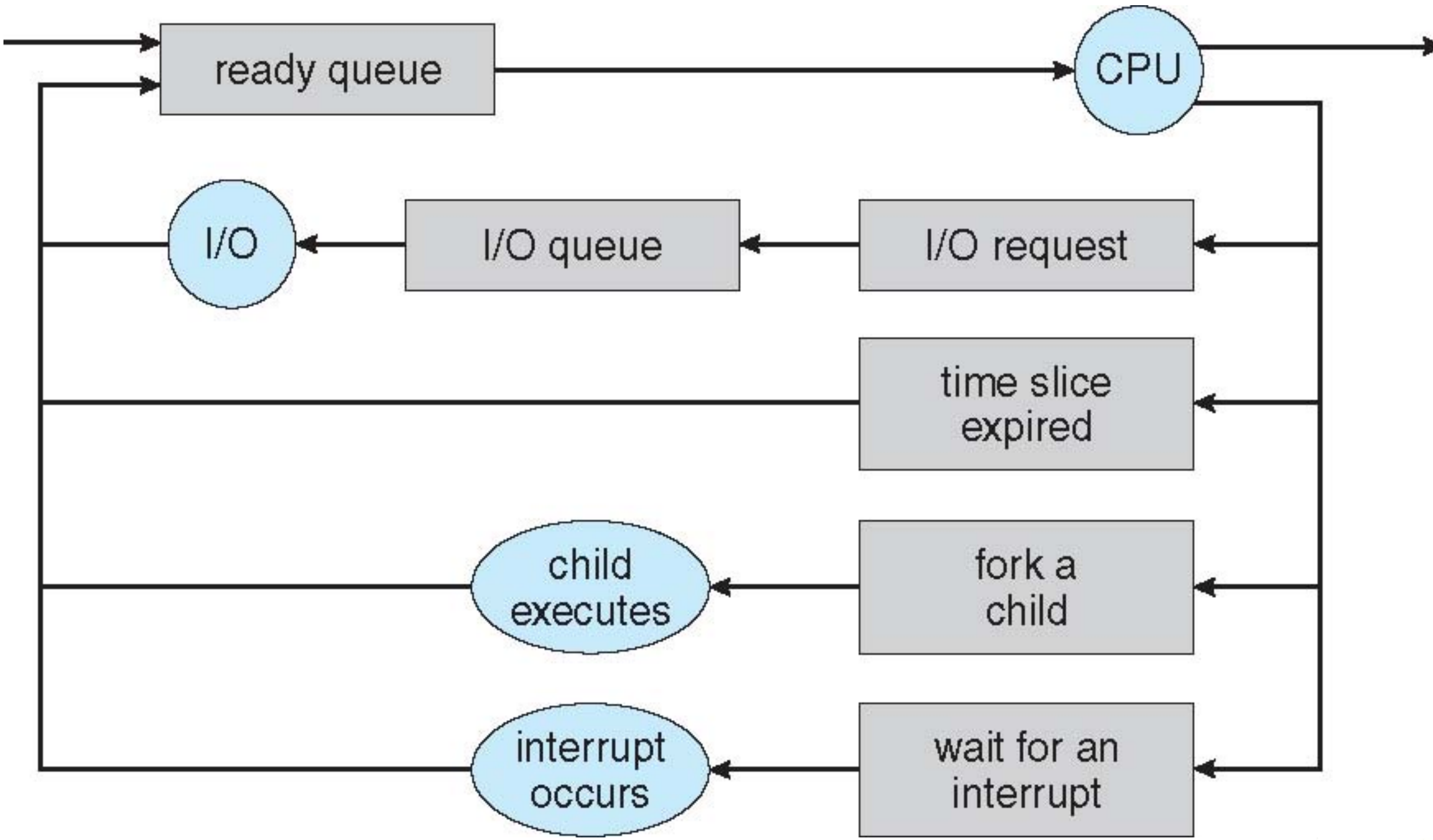
Ready Queue and Various I/O Device Queues



Each device has its own queue

Representation of Process Scheduling

- **Queuing Diagram** represents queues, resources, and flows



Schedulers

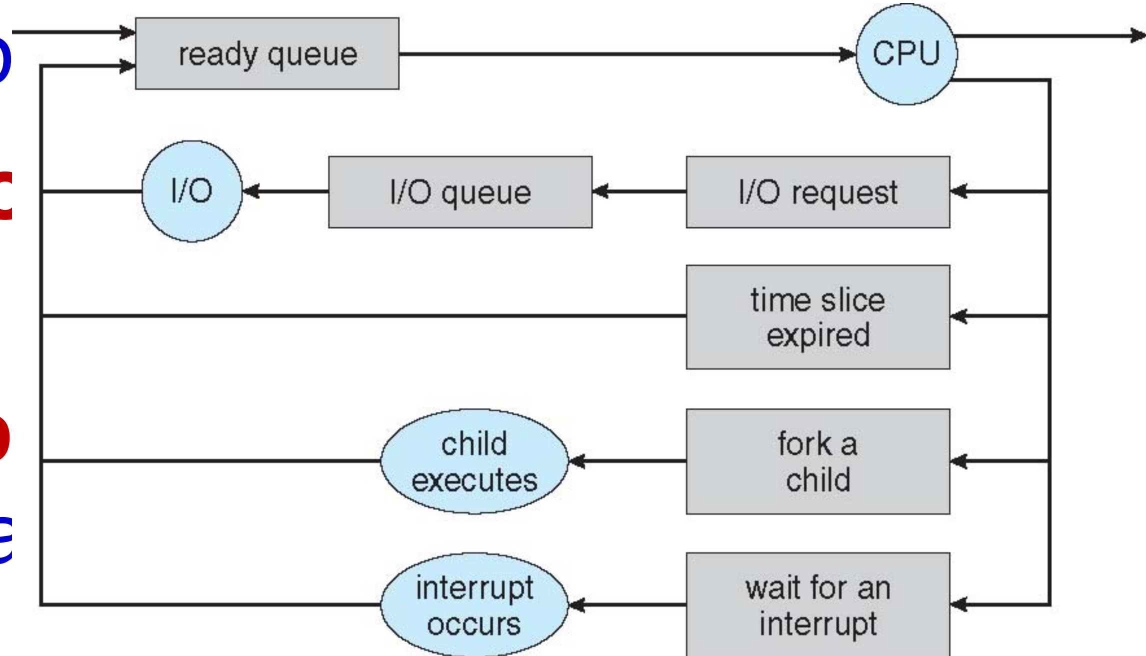
- A Process Migrates Among various Scheduling Queues throughout its Lifetime
  - OS selects (schedules) processes in queues
- **Short-Term (ST) Scheduler** (CPU Scheduler)
  - Selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Is invoked frequently (milli-sec)
    - Must be fast
    - Example: scheduler takes 10ms to run and it is invoked every 100ms ➔ ~9% OS overhead on CPU

Schedulers (cont.)

- **Long-Term (LT) scheduler** **(job scheduler)**
  - Selects which processes should be brought into ready queue
  - Invoked infrequently (sec~min) $\Rightarrow$ (may be slow)
  - Controls **degree of multiprogramming**
    - **i.e., determines no of processes in memory**
  - **Steady-State:** long-term scheduler should be invoked only a process leaves the system
    - Assuming degree of multiprogramming is stable

Schedulers (cont.)

- Processes can b
  - **I/O-bound prc** doing I/O than
  - **CPU-bound p** doing computa



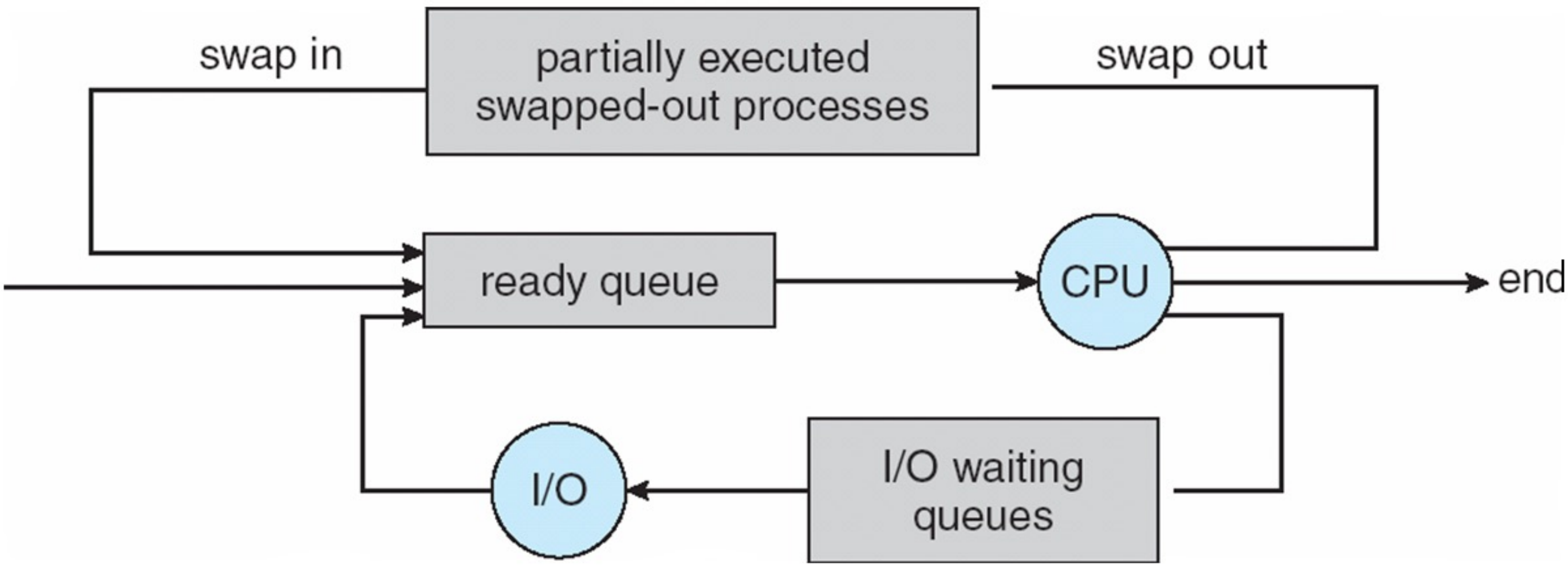
- LT Scheduler Strives for Good ***process mix***
  - All processes I/O bound ➔ ready queue will almost always be empty ➔ ST scheduler idle
  - All processes CPU bound ➔ I/O waiting queue will almost always be empty ➔ devices idle

Schedulers (cont.)

- LT Scheduler May be Absent or Minimal
  - E.g., UNIX and Windows have no LT scheduler and put every new process in memory
  - ➔ Can adversely affect performance
  - ➔ So, users may quit and their processes are terminated

Addition of Medium Term Scheduling

- **Medium-Term Scheduler** Used if Degree of Multi-Programming Needs to Decrease
  - **Swapping:** remove process from memory, store on disk (swapped out), bring back in from disk to continue execution (swapped in)
  - Used to improve process mix



Context Switch

- When CPU Switches to another Process
  - ➔ System must **save state** of old process
  - ➔ OS, then, loads **saved state** for new process via a **context switch**
- **Context** of a Process Represented in PCB
- Context-Switch Time is Overhead
  - System does no useful work while switching
  - More complex OS and PCB ➔ longer context switch
  - More context switching time when using VMs

Context Switch (cont.)

- Context-Switch Time Depends on HW Support
  - Memory speed
  - # of registers in RF
  - Special instruction to copy RF
    - Also, HW (micro-architecture) support
  - Some HW provides multiple sets of registers per CPU ➔ multiple contexts loaded at once
    - Context switch: changing a pointer to a target RF

Operations on Processes

- System must Provide Mechanisms for
  - Process creation
  - Process termination
  - and so on as detailed next

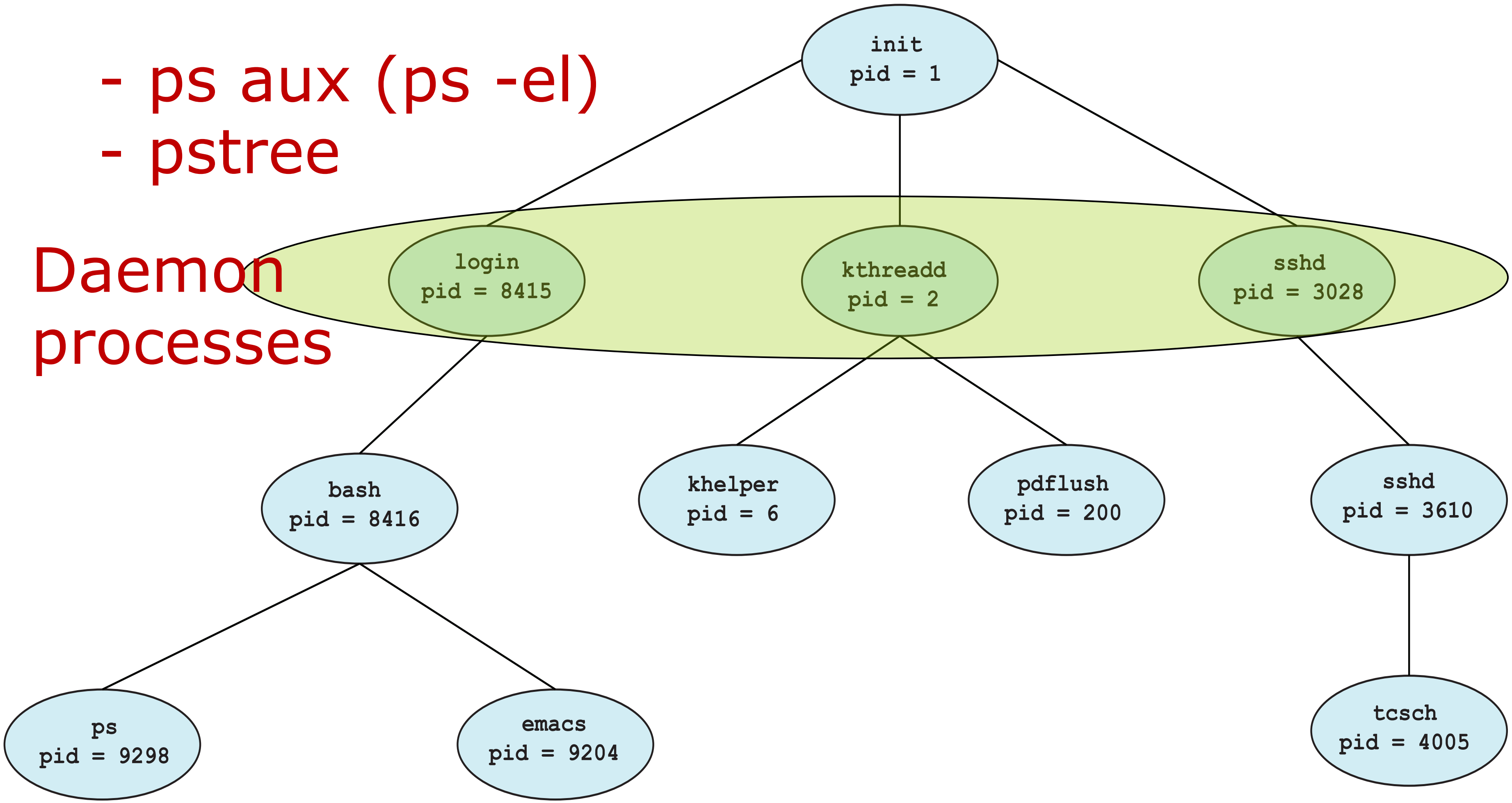
Process Creation

- **Parent** Process Creates **Children** processes
  - Which, in turn creates other processes, forming a **tree** of processes
- Process Identified and Managed via a **Process Identifier (pid)**
- Root Process in Linux: “init”
- Root Process in Solaris: “sched”
  - Children processes: “Init”, “pageout”, “fsflush”
    - “Init” root of user processes

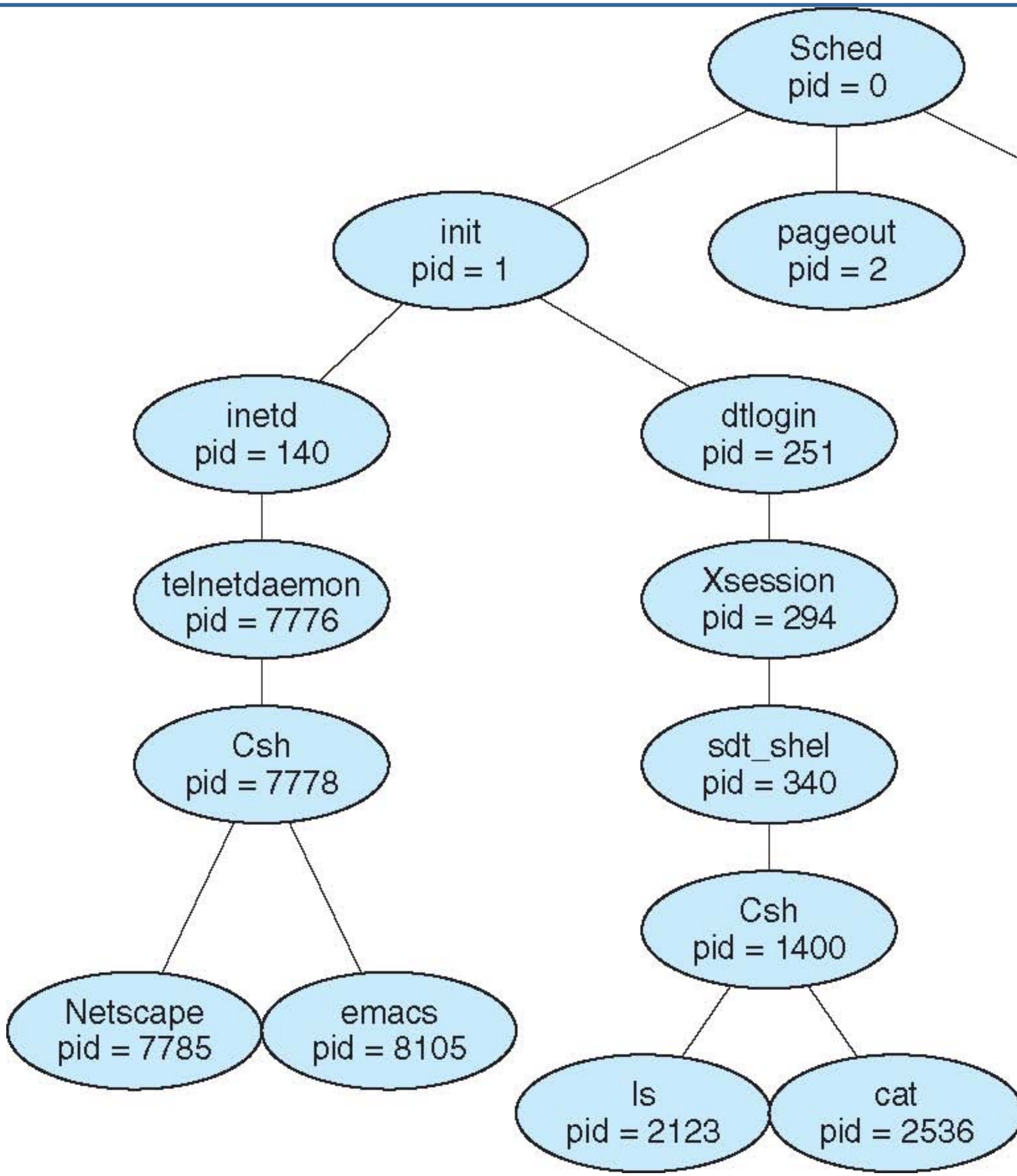
A Tree of Processes in Linux

- ps aux (ps -el)
- pstree

Daemon processes



A Tree of Processes in Solaris



Process Creation (cont.)

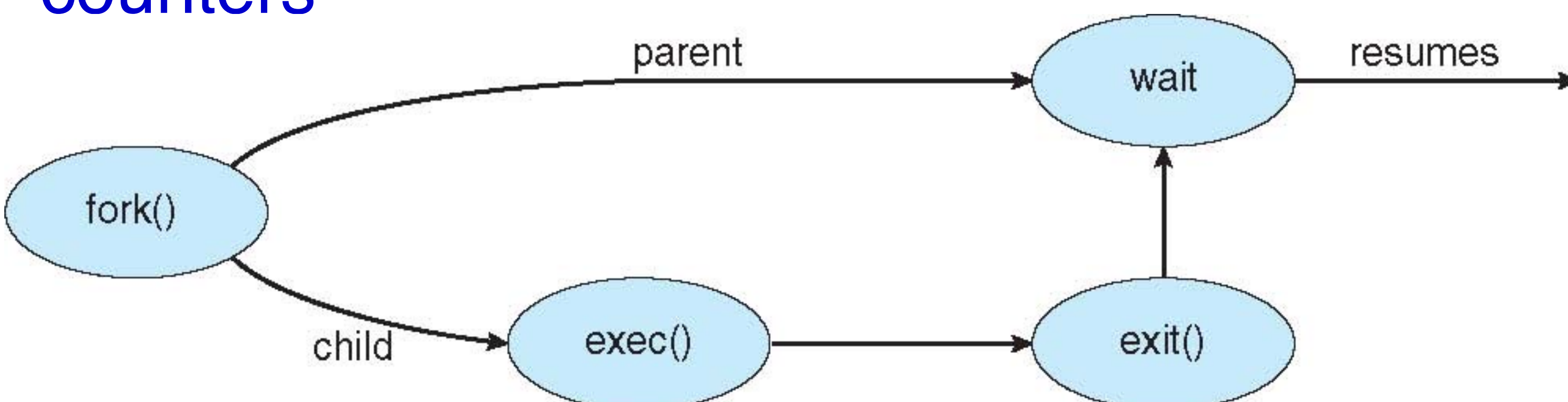
- Resource **Sharing Options**
  - Parent and children share all resources
  - Children share subset of parent's resources
    - Prevents any process from overloading system by creating too many sub-processes
  - Parent and child share no resources
- **Execution Options**
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- Address Space
  - Child duplicate of parent
  - Child has a new program loaded into it
- UNIX Examples
  - **fork()** system call creates new process
    - New process consists of a copy of address space of original process
    - Convenient communication between two processes
  - **exec()** system call used after a **fork()** to replace process' memory space with a new program

Process Creation (Cont.)

- What Happens after “fork()”?
  - Both processes continue execution at the instruction after “fork”
  - Child gets unique process ID
  - Child’s PPID = parent’s PID
  - Reset child’s resource utilization and CPU time counters



Process Creation (Cont.)

- What Does not Child Inherits from Parent?
  - Parents memory locks
  - Parent’s timers
  - Semaphore’s adjustments and pending signals
- What Child Process Inherits from Parent?
  - Privileges and scheduling attributes
  - Certain resources such as open files

C Program Forking Separate Process

```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");
       return 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");
    }

    return 0;
}
```

Creating a Separate Process via Windows API

```
#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 */
    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 Termination

- Process Executes Last Statement
  - Then asks OS to delete it
    - Using **`exit()`** system call
  - Returns status data from child to parent via **`wait()`**
  - Process' resources are de-allocated by OS
- Parent may Terminate Execution of Children Processes Using **`abort()`** System Call
  - Or TerminateProcess() in Win32

Process Termination (cont.)

- Reasons to Terminate Child Processes
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - Parent is exiting and OS does not allow a child to continue if its parent terminates
  - Or …

Process Termination (cont.)

- Some OSes do not Allow Child to Exists
  - If its parent has terminated
  - If a process terminates, then all its children must also be terminated
  - **Cascading termination**
    - All children, grandchildren, etc. are terminated
  - Termination is initiated by OS
- Parent process may Wait for Termination of a Child Process by Using **`wait()`** System Call

Process Termination (cont.)

- Call Returns Status Info and pid of Terminated Process

```
pid = wait(&status);
```

- Zombie Process
  - If no parent waiting (did not invoke `wait()`)
  - Process has completed execution but still has an entry in process table
  - Entry needed for possible reading of exit status by its parent

Process Termination (cont.)

- Orphan Process
  - Parent terminated without Invoking **wait**
  - In UNIX, any orphan process is immediately adopted by "init" process
    - Called "re-parenting"
  - A process can become orphan intentionally or unintentionally
    - Process crash
    - To run a process indefinitely (in the background)

Process Termination (cont.)

- Example Code 1:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
   printf("Hello \n");
   int pid=fork();
   printf("pid = %d \n", getpid());
   sleep(20);
   printf("Sample cod for fork \n");
   return 0;
}
```

- Two Scenarios
  - Case 1: Terminate parent process ➔ reparenting
  - Case 2: Terminate child process ➔ Zombie process

Process Termination (cont.)

- Example Code 2:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
  printf("Hello \n");
  int pid=fork();
  printf("pid = %d \n", getpid());
  sleep(20);
  printf("Sample cod for fork \n");
  wait(NULL);
  sleep(10);
  return 0;
}
```

- Test Scenario
  - Terminate child process ➔ Zombie process ➔ removed after wait call by parent

Process Termination (cont.)

- Other Variations of Wait
  - waitid()
  - waitpid()
- Waits for specific child process

Multiprocess Architecture – Chrome Browser

- Many Web Browsers Ran as Single Process (Some Still Do)
  - If one web-site causes trouble, entire browser can hang or crash
- Google Chrome Browser is Multi-Process with Three Different Types of Processes

Multiprocess Architecture – Chrome Browser

- Google Chrome Browser is Multi-Process with Three Different Types of Processes
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript.
  - A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

Interprocess Communication

- Processes: ***Independent*** or ***Cooperating***
- ***Independent*** Process
  - Cannot affect or be affected by execution of another Process
  - Does not share data with any other processes
- ***Cooperating*** Process can Affect or be Affected by Execution of another Process
  - Including shared data

Cooperating Processes

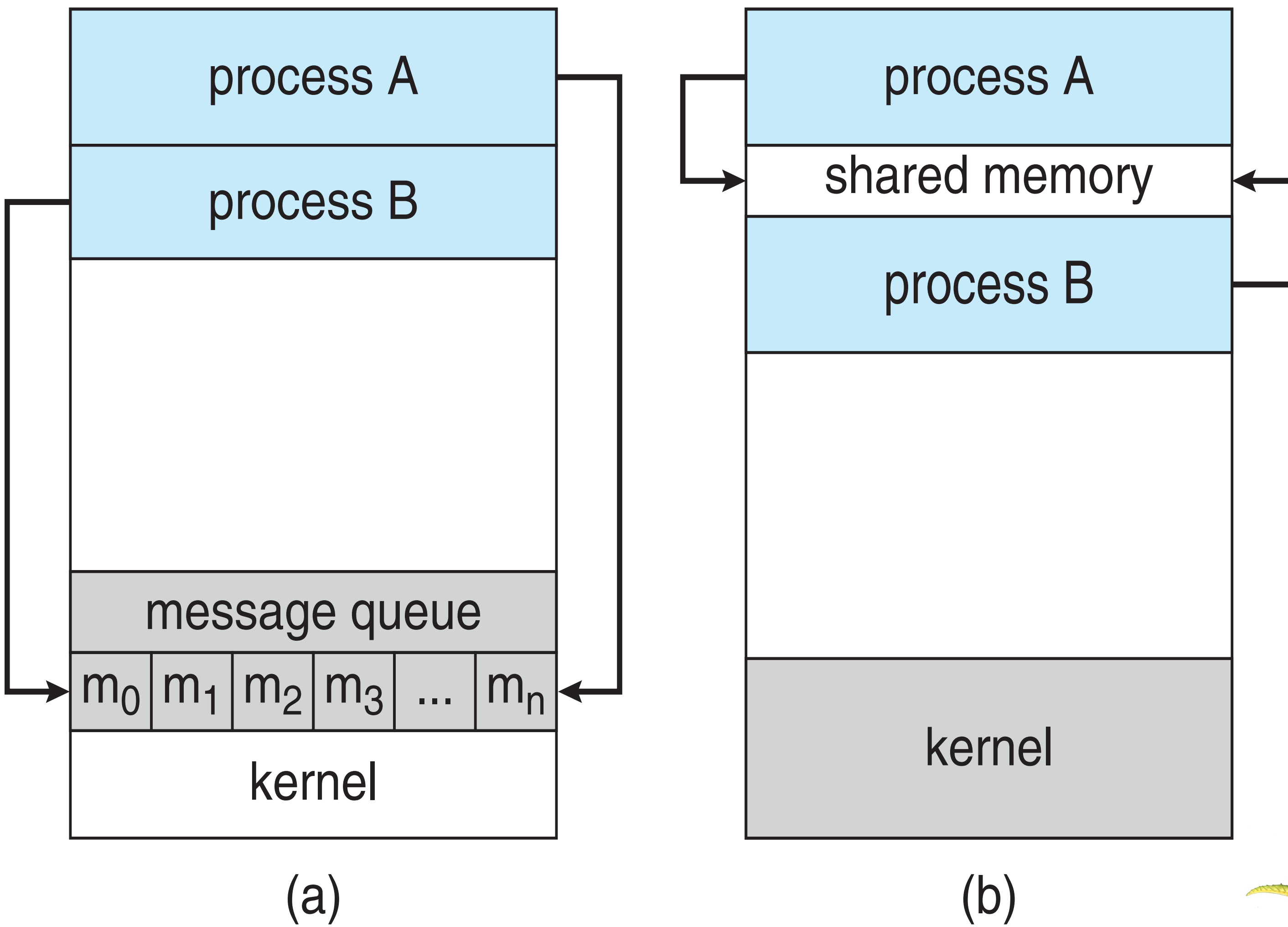
- Motivations of Process Cooperation
  - Information sharing
    - E.g., shared file
  - Computation speed-up
    - Breaking into subtasks & executing on multiple cores
    - E.g., computing Pi
  - Modularity
    - Constructing a system in a modular fashion
    - Dividing system functions into separate processes
  - Convenience
    - Several tasks of a single user (editing, printing, …)

Interprocess Communication (cont.)

- Cooperating Processes need **Interprocess Communication (IPC)**
- Two Models of IPC
  - **Shared memory**
  - **Message passing**
- Most OSes Implement Both Models

Communications Models

(a) **Message passing** (b) **Shared memory**



Interprocess Communication (cont.)

- **Shared memory**
  - Maximum speed
  - Convenience of communication
  - System calls used only to establish shared-memory regions
    - Thereafter, all accesses are treated as normal memory accesses
- **Message passing**
  - Useful for exchanging smaller amounts of data
  - No conflicts need to be avoided
  - Easier to be implemented for inter-computer communication
  - Implemented by system calls → time-consuming

Shared-Memory Solution: Producer-Consumer Problem

- Paradigm for Cooperating Processes
  - *Producer* process produces info that is consumed by a *consumer* process
  - **Unbounded-buffer** places no practical limit on size of 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 next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer: Consumer

```
item next_consumed;
while (true) {
     while (in == out)
          ; /* do nothing */
     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     /*consume item in next consumed*/
}
```

Interprocess Communication: Shared Memory

- Main Idea
  - An area of memory shared among processes that wish to communicate
- Communication is under Control of Users Processes not OS
- Major Issue
  - To provide mechanism that will allow user processes to synchronize their actions when they access shared memory
  - Synchronization will be discussed in details in next lectures

Interprocess Communication: Message Passing

- Mechanism for Processes to Communicate and to Synchronize their Actions
- Processes Communicate with each other
  - Without resorting to shared variables
  - Without having a shared address space
- Significantly useful in a Distributed Systems
  - E.g., a chat program
- Two Major Operations
  - **`send`**(*message*)
  - **`receive`**(*message*)

Message Passing (Cont.)

- If Processes *P* and *Q* Wish to Communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- *Message* size is either Fixed or Variable
- Fixed Message Size
  - Straightforward implementation ☺
  - Difficulty for programmer ☹

Message Passing (Cont.)

- Implementation Issues:
  - 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 capacity of a link?
  - Is size of a message that link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing: Communication Link

- Communication Link can be Viewed at either
  - Physical or Logical
- Physical
  - Shared memory
  - Hardware bus
  - Network
- Logical
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering

Direct Communication

- Processes must Name each other Explicitly
  - **send** (*P*, *msg*): send a message to process P
  - **receive**(*Q*, *msg*): receive a message from Q
- Properties of Direct 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
  - Link may be unidirectional, but is usually bidir
    - Bidirectional: symmetric (example as above)
    - Directional: Asymmetric

Indirect Communication

- Main Issue in Direct Communication
  - Identifiers must **explicitly stated** (e.g, process P or Q)
- 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 an **Indirect** 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 bidirectional

Indirect Communication

- Operations
  - Create a new mailbox (port)
  - 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 (cont.)

- Mailbox Sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets 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 system to select arbitrarily receiver
  - Sender is notified who receiver was

Synchronization

- Message Passing may be either Blocking or Non-Blocking
- **Blocking** considered **Synchronous**
  - **Blocking send** -- sender is blocked until message is received
  - **Blocking receive** -- receiver is blocked until a message is available

Synchronization (cont.)

- **Non-Blocking** Considered **Asynchronous**
  - **Non-blocking send** -- sender sends message and continue
  - **Non-blocking receive** -- receiver receives:
    - A valid message, or
    - Null message
- Different Combinations Possible
  - If both send and receive are blocking → we have a **rendezvous**

Synchronization (Cont.)

- Producer-Consumer becomes Trivial

```
message next_produced;
 while (true) {
      /*produce an item in next produced*/
 send(next_produced);
}
```

```
message next_consumed;
while (true) {
    receive(next_consumed);
    /*consume the item in next consumed*/
}
```

Buffering

- Implemented in one of Three Ways

  1. **Zero capacity** – no messages are queued on a link
     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

Pipe Sample Code

```
#include<stdio.h>
#include<unistd.h>
int main() {
  int pipefds[2];
  int returnstatus;
  int pid;
  char writemessages[2][20]={"Hi", "Hello"};
  char readmessage[20];
  returnstatus = pipe(pipefds);
  if (returnstatus == -1) {
    printf("Unable to create pipe\n");
    return 1;
  }
  pid = fork();
// Child process
if (pid == 0) {
      read(pipefds[0], readmessage, sizeof(readmessage));
      printf("Child Process - Reading from pipe – Message 1 is %s\n", readmessage);
      read(pipefds[0], readmessage, sizeof(readmessage));
      printf("Child Process - Reading from pipe – Message 2 is %s\n", readmessage);
  } else { //Parent process
      printf("Parent Process - Writing to pipe - Message 1 is %s\n", writemessages[0]);
      write(pipefds[1], writemessages[0], sizeof(writemessages[0]));
      printf("Parent Process - Writing to pipe - Message 2 is %s\n", writemessages[1]);
      write(pipefds[1], writemessages[1], sizeof(writemessages[1]));
  }
  return 0;
}
```

Pipe: Two-Way Sample Code

```
#include<stdio.h>
#include<unistd.h>
int main() {
  int pipefds1[2], pipefds2[2];
  int returnstatus1, returnstatus2;
  int pid;
  char pipe1writemessage[20] = "Hi";
  char pipe2writemessage[20] = "Hello";
  char readmessage[20];
  returnstatus1 = pipe(pipefds1);
  if (returnstatus1 == -1) {
    printf("Unable to create pipe 1 \n");
    return 1;
  }
  returnstatus2 = pipe(pipefds2);
  if (returnstatus2 == -1) {
    printf("Unable to create pipe 2 \n");
    return 1;
  }
  pid = fork();
  if (pid != 0) // Parent process {
      close(pipefds1[0]); // Close the unwanted pipe1 read side
      close(pipefds2[1]); // Close the unwanted pipe2 write side
      printf("In Parent: Writing to pipe 1 – Message is %s\n", pipe1writemessage);
      write(pipefds1[1], pipe1writemessage, sizeof(pipe1writemessage));
      read(pipefds2[0], readmessage, sizeof(readmessage));
      printf("In Parent: Reading from pipe 2 – Message is %s\n", readmessage);
   } else { //child process
      close(pipefds1[1]); // Close the unwanted pipe1 write side
      close(pipefds2[0]); // Close the unwanted pipe2 read side
      read(pipefds1[0], readmessage, sizeof(readmessage));
      printf("In Child: Reading from pipe 1 – Message is %s\n", readmessage);
      printf("In Child: Writing to pipe 2 – Message is %s\n", pipe2writemessage);
      write(pipefds2[1], pipe2writemessage, sizeof(pipe2writemessage));
   }
   return 0;
}
```

Examples of IPC Systems

- Reading Assignments
  - POSIX shared memory
  - Message passing in Mach OS
  - Winx XP (message passing/shared memory)
  - Sockets
  - Remote Procedure Calls
  - Pipes
  - Remote Method Invocation (Java)

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment

    ```
    shm_fd = shm_open(name, O CREAT | O RDWR, 0666);
    ```

  - Also used to open existing segment to share it
  - Processes wish to access shared memory must attach it to their address space using mmap or

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

  - Now process could write to shared memory

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

```
Finally, to detach: shmdt(shared_mem)
```

IPC POSIX Producer

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

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

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

IPC POSIX Consumer

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

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Examples of IPC Systems - Mach

- Mach Communication is Message Based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- Kernel & Notify
  - Only three system calls needed for message transfer

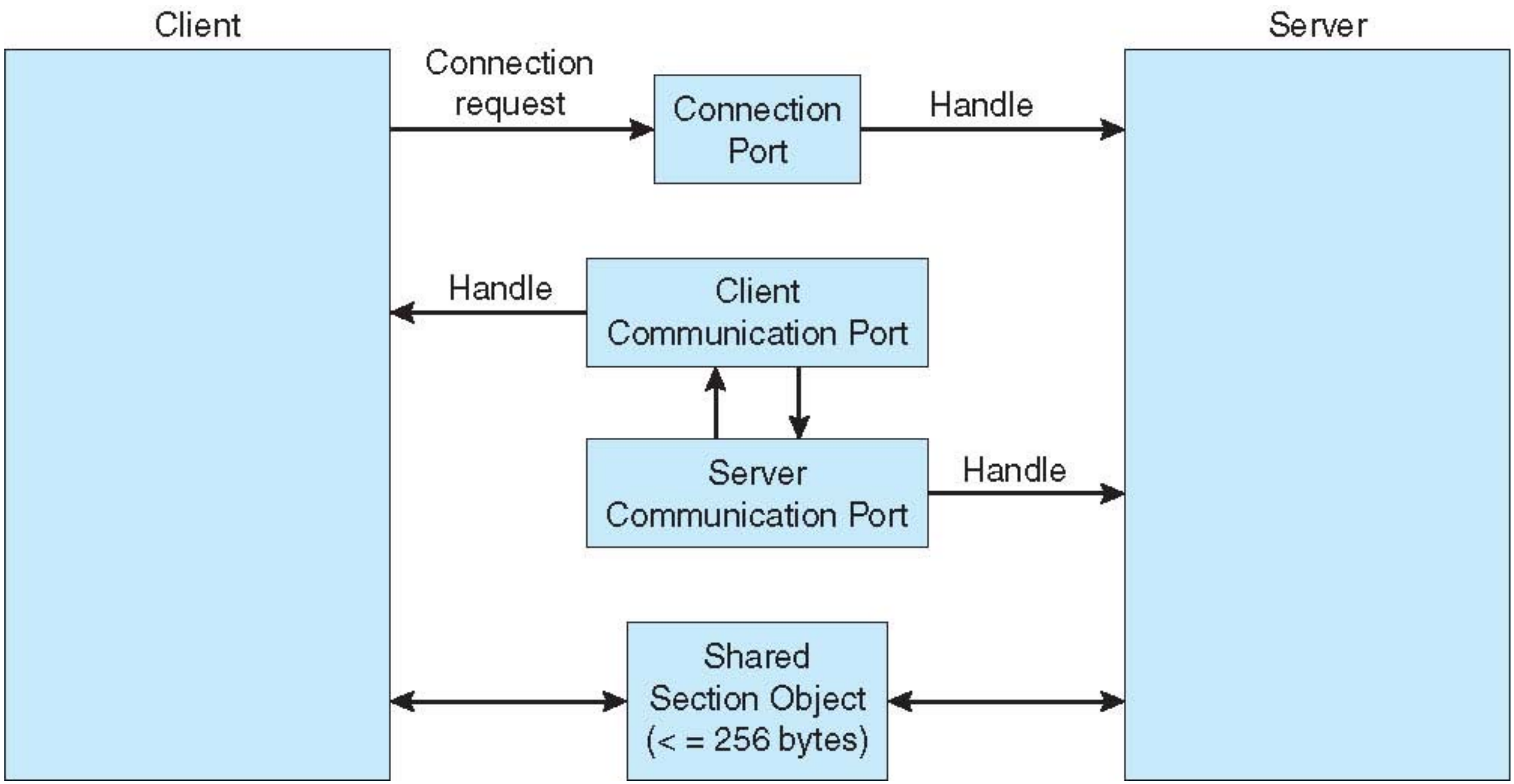
    `msg_send(), msg_receive(), msg_rpc()`
  - Mailboxes needed for commuication, created via

    `port_allocate()`
  - Send and receive are flexible, e.g. 4 options if mailbox full:
    - Wait indefinitely
    - Wait at most *n* milliseconds
    - Return immediately
    - Temporarily cache a message

Examples of IPC Systems – Windows

- Message-Passing Centric via **Advanced 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:
    - Client opens a handle to the subsystem's **connection port** object
    - Client sends a connection request
    - Server creates two private **communication ports** and returns the handle to one of them to the client
    - Client and server use the corresponding port handle to send messages or callbacks and to listen for replies

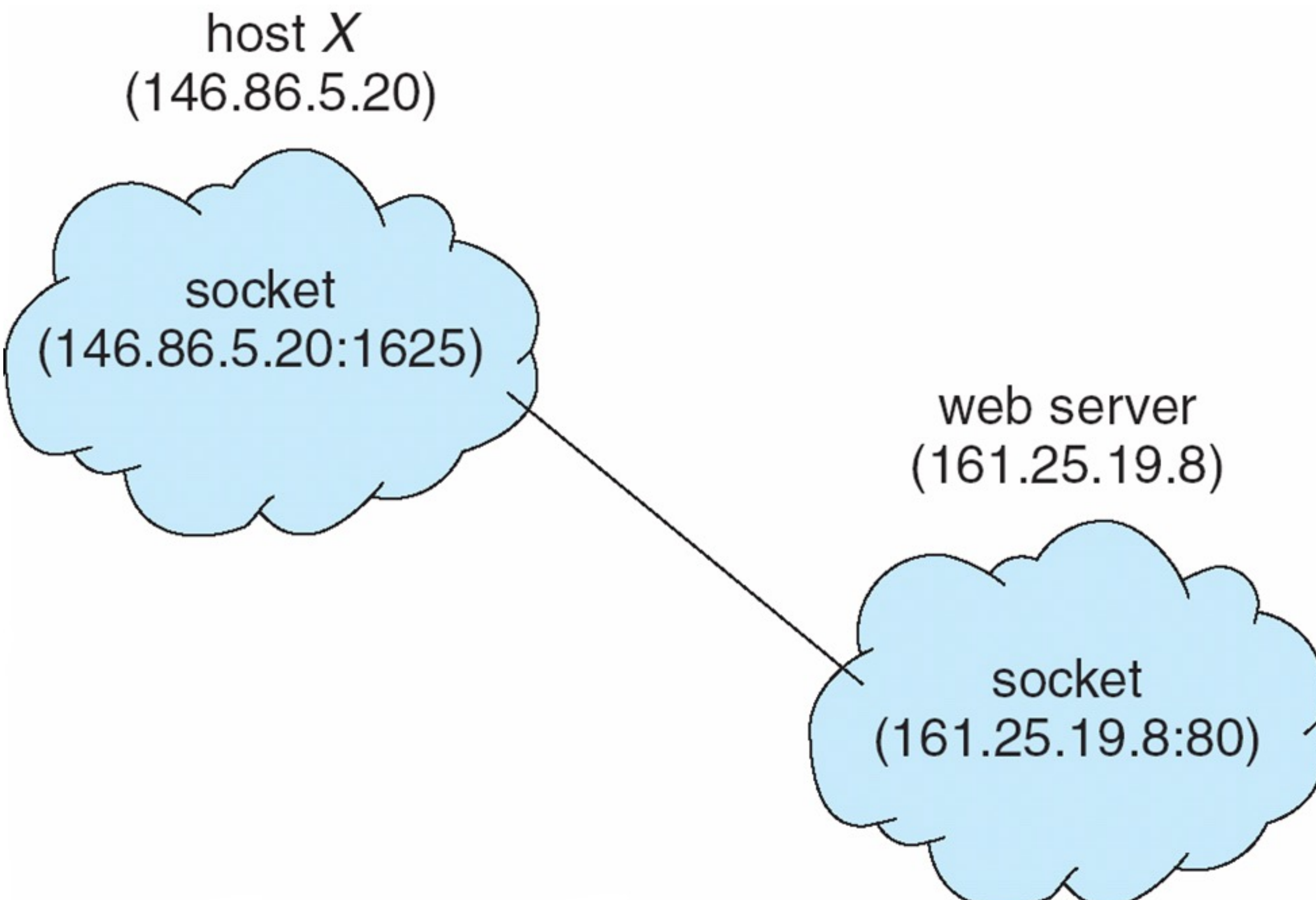
Local Procedure Calls in Windows



Sockets

- **Socket:** an Endpoint for Communication
- Concatenation of IP address and **Port**
  - A number included at start of message packet to differentiate network services on a host
  - Socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All Ports below 1024 are ***Well Known***
  - Used for standard services
- Special IP address 127.0.0.1 (**loopback**)
  - Refers to system on which process is running

Socket Communication



Sockets in Java

- 3 Types of Sockets
  - **Connection-oriented (TCP)**
  - **Connectionless (UDP)**
  - **`MulticastSocket`** class– data can be sent to multiple recipients
- Consider this "Date" server

```
import java.net.*;
import java.io.*;

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);
      }
   }
}
```

Remote Procedure Calls

- **Remote Procedure Call** (RPC)
  - Abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
  - Stubs – client-side proxy for actual procedure on server
  - Client-side stub locates server and marshalls parameters
  - Server-side stub receives this message, unpacks marshalled parameters, and performs procedure on server

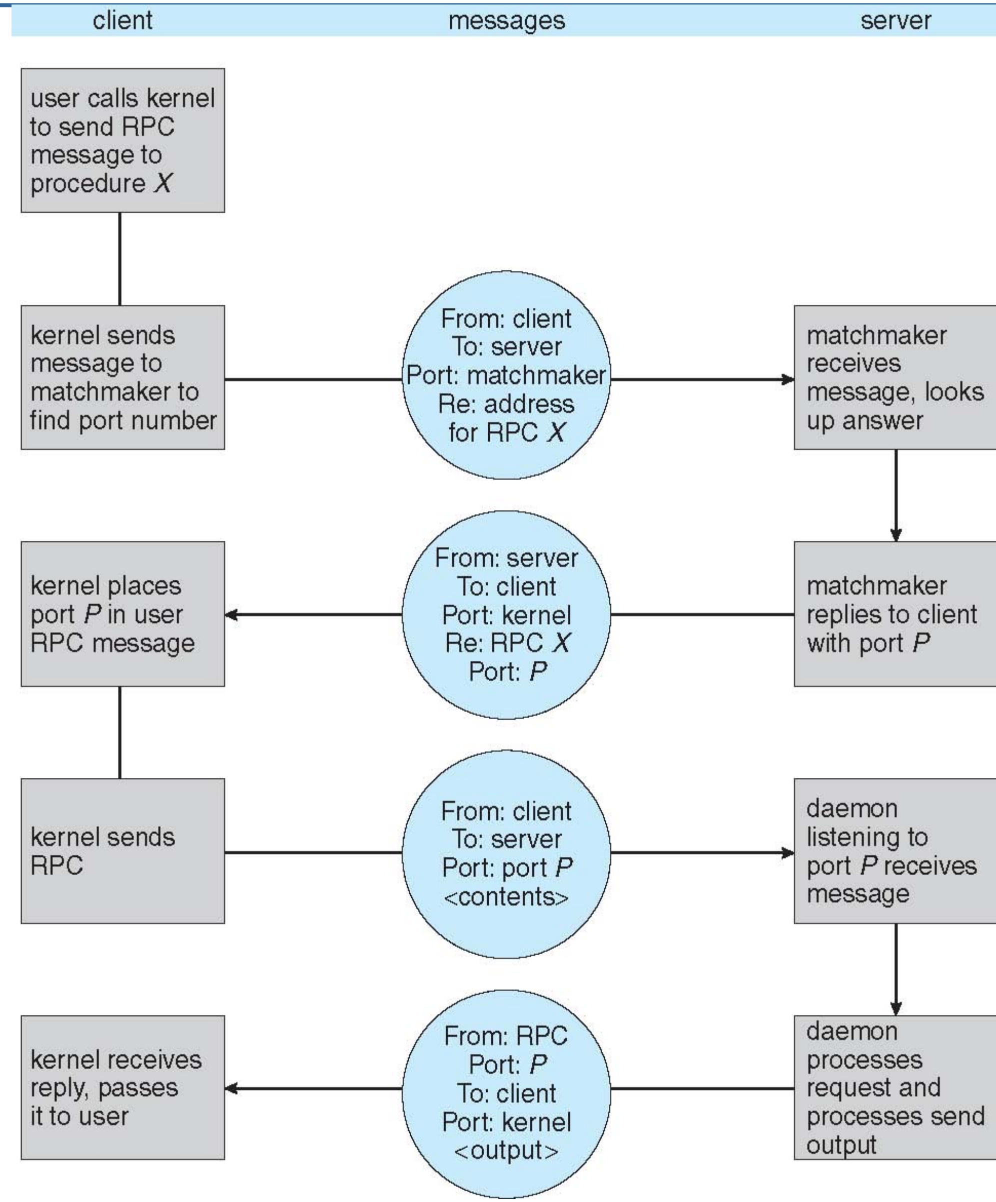
Remote Procedure Calls (Cont.)

- On Windows
  - Stub code compile from specification written in **Microsoft Interface Definition Language (MIDL)**

Remote Procedure Calls (Cont.)

- Data Representation Handled via **External Data Representation (XDL)** Format
  - To account for different architectures
    - **Big-endian** and **little-endian**
- Remote Communication Has more Failure Scenarios than Local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS Typically Provides a Rendezvous (or **matchmaker**) Service to Connect Client and Server

Execution of RPC



Pipes

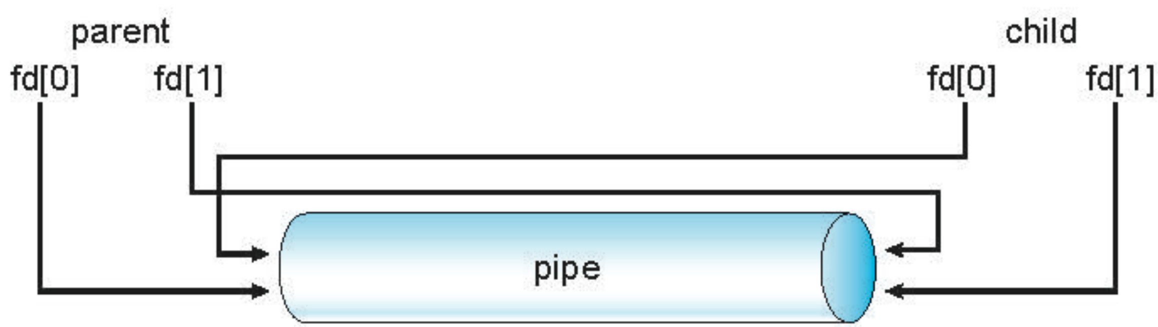
- Acts as a Conduit allowing Two Processes to Communicate
  - 1st IPC mechanism used in early UNIX
- Issues:
  - Is communication unidirectional or bidirectional?
  - In case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can pipes be used over a network?

Pipes (cont.)

- Ordinary Pipes
  - Cannot be accessed from outside process that created it.
  - Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named Pipes
  - Can be accessed without a parent-child relationship

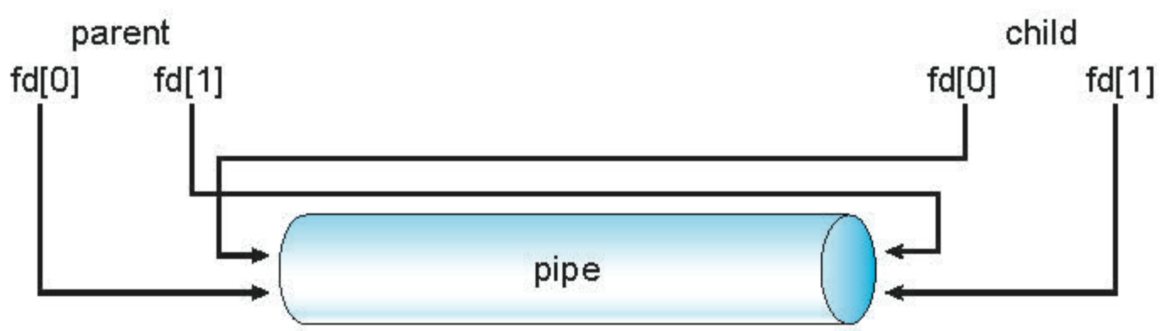
Ordinary Pipes

- Ordinary Pipes allow Communication in Standard Producer-Consumer Style
- Producer Writes to One End (**write-end** of Pipe)
- Consumer Reads from other end (**read-end** of Pipe)



Ordinary Pipes (Cont.)

- Ordinary Pipes are therefore Unidirectional
- Require Parent-Child Relationship between Communicating Processes
- Windows Calls these **Anonymous Pipes**
- See Unix and Windows Code Samples in Textbook



Named Pipes

- Named Pipes More powerful than Ordinary Pipes
- Communication is Bidirectional
- No Parent-Child Relationship is Necessary between Communicating Processes
- Several Processes can Use Named Pipe for Communication
- Provided on both UNIX and Windows Systems

End of Lecture 3