Chapter 3: Processes

Coverage:

- Process Concept
- Process Scheduling
- Operations on Processes
- Cooperating Processes
- Inter-process Communication

What is a Process?

- An operating system executes a variety of programs:
  - Batch system – jobs (different than processes)
  - Time-shared systems – user programs or tasks

- Process – a program in execution; process execution must progress in sequential fashion.
  - Textbook uses the terms *job* and *process* “almost” interchangeably.

What is a Process?

- A process includes:
  - instructions (code text section)
  - program counter, other registers
  - stack
  - data section (static) and heap

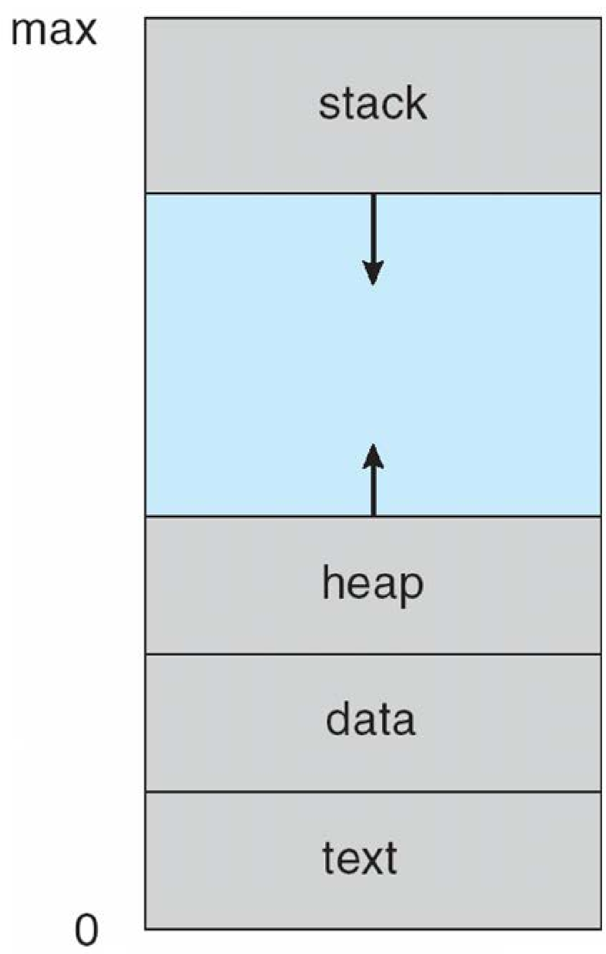
```
#include <stdio.h>

main()
{
        FILE  *file;
        int value;

        if ((file=fopen("abc", "r")) != NULL) exit(1);
        if (fread((char*)&value, sizeof(int), 1, file) == 1)
            printf("%d\n", value);
        fclose(file);
}
```

Opening and reading a file via a stream. Read man pages here and here.

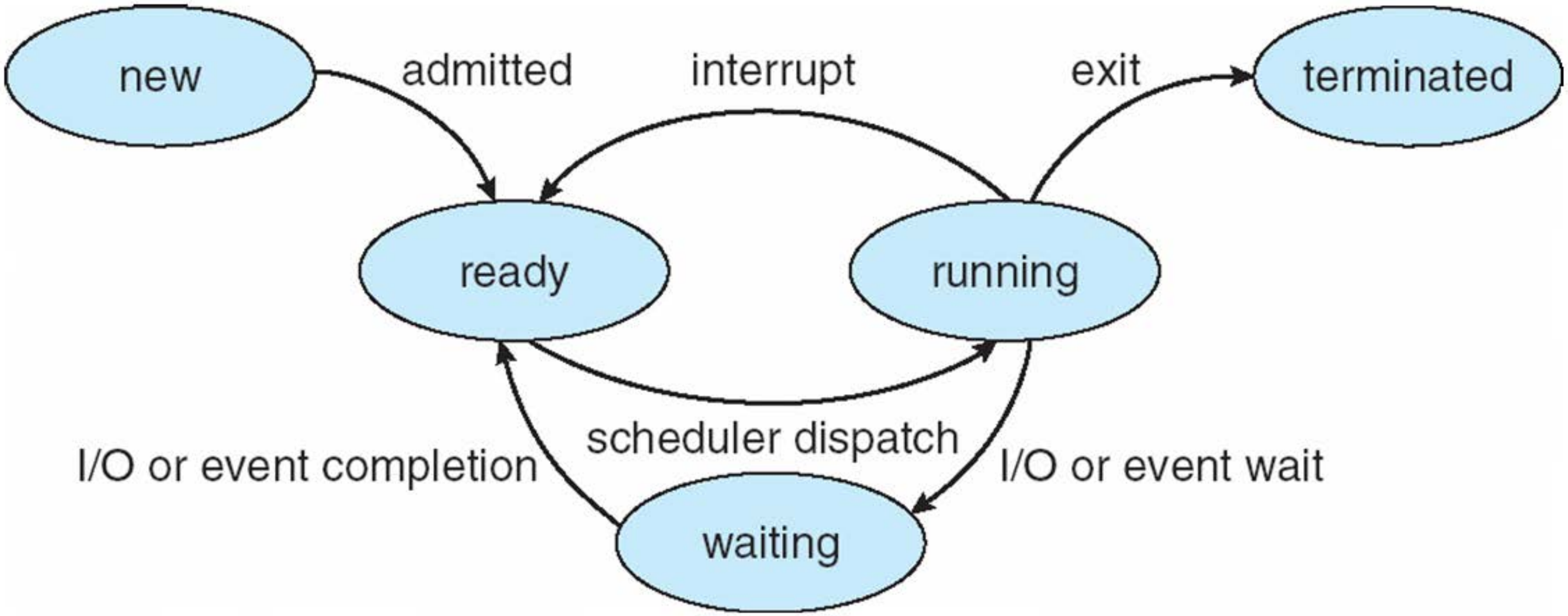
Process in Memory



Process State

- As a process executes, it changes **state.**
  - **new**: The process is being created.
  - **running**: Instructions are being executed.
  - **waiting**: The process is waiting for some event to occur.
  - **ready**: The process is waiting to be assigned to a processor.
  - **terminated**: The process has finished execution.

Diagram of Process State



Process Control Block (PCB)

Information associated with each process

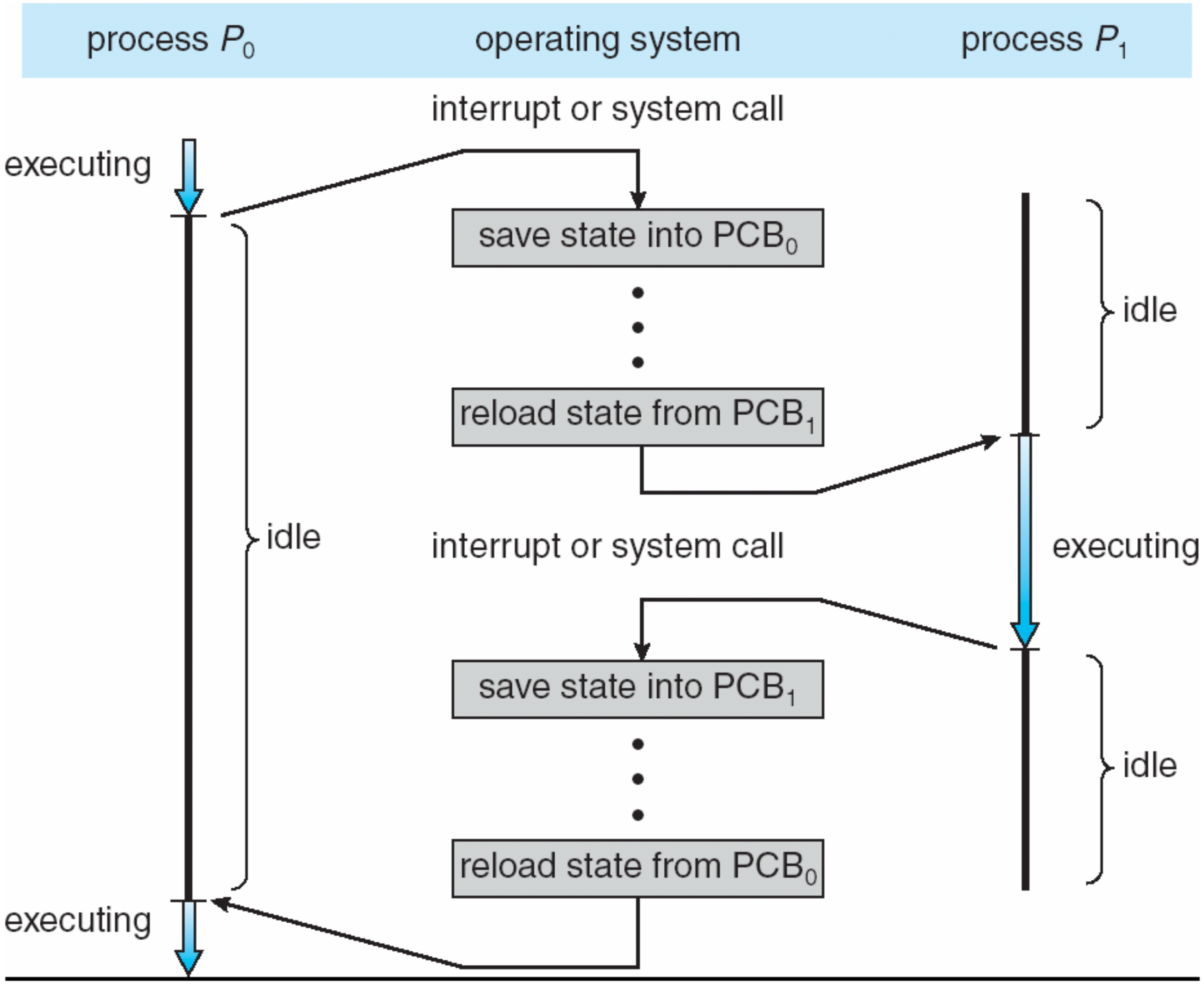
(also called **Task Control Block (TCB)**):

- 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.

Context Switch

- CPU switches to another process; OS kernel:
  - Save the state of the old process;
  - Load the saved state for the new process.
    PCB represents the context.
- Context-switch time is overhead; no useful work while switching. Up to thousands of instructions run during a single process context switch.
  - Store/load registers
  - Update the PCB
  - Run the scheduling algorithm
  - Maintain the queues
  - Update process profiles, etc
- Time: dependent on hardware support;
  Special hardware support for context switch.

CPU Switch From Process to Process



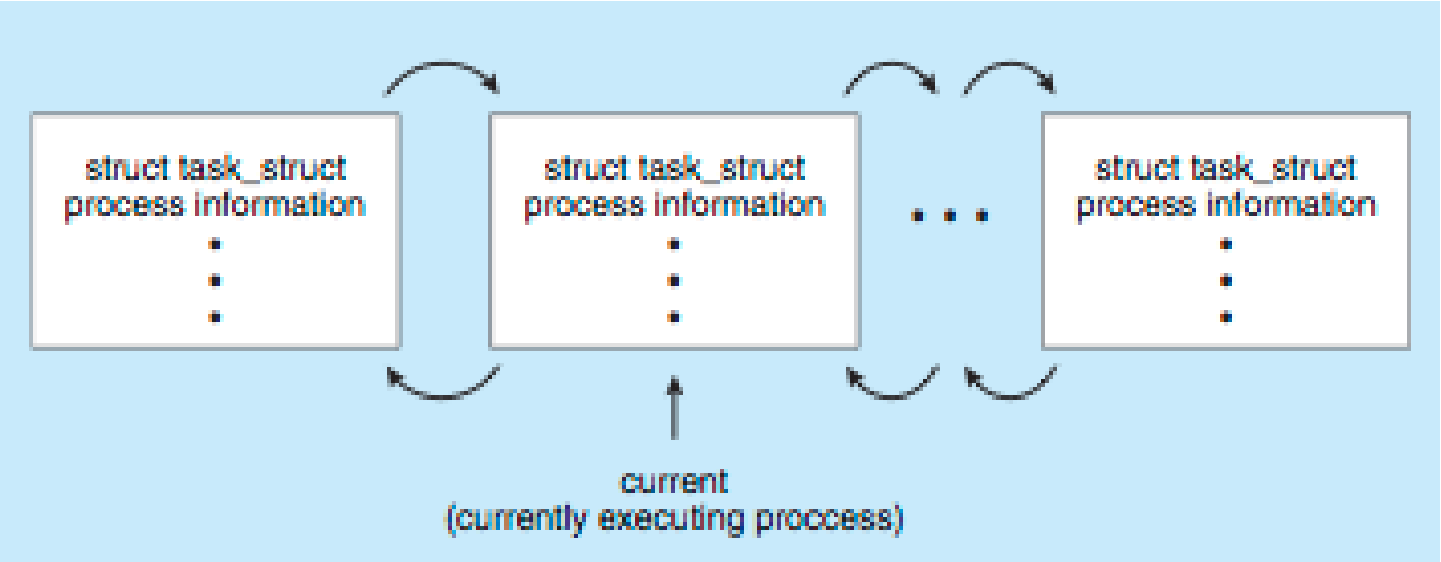
Threads

- So far, process has a single thread of execution.
- 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.
- See chapter 4.

Process Representation in Linux

- Represented by the C structure `task_struct`

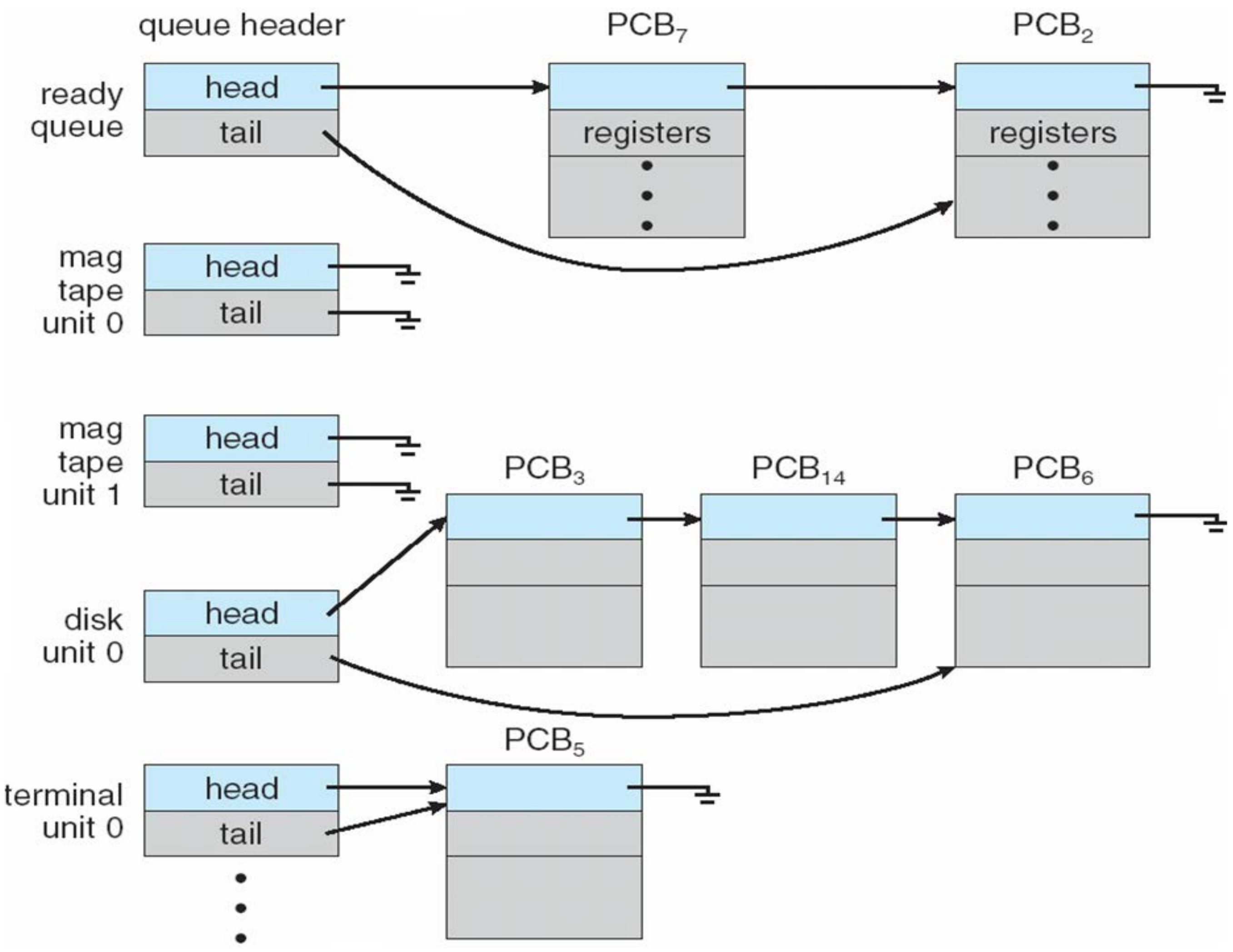
```
pid t pid; /* process identifier */
long state; /* state of the process */
unsigned int time slice /* scheduling information */
struct task struct *parent; /* this process's parent */
struct list head children; /* this process's children */
struct files struct *files; /* list of open files */
struct mm struct *mm; /* address space of this process */
```



Process Scheduling

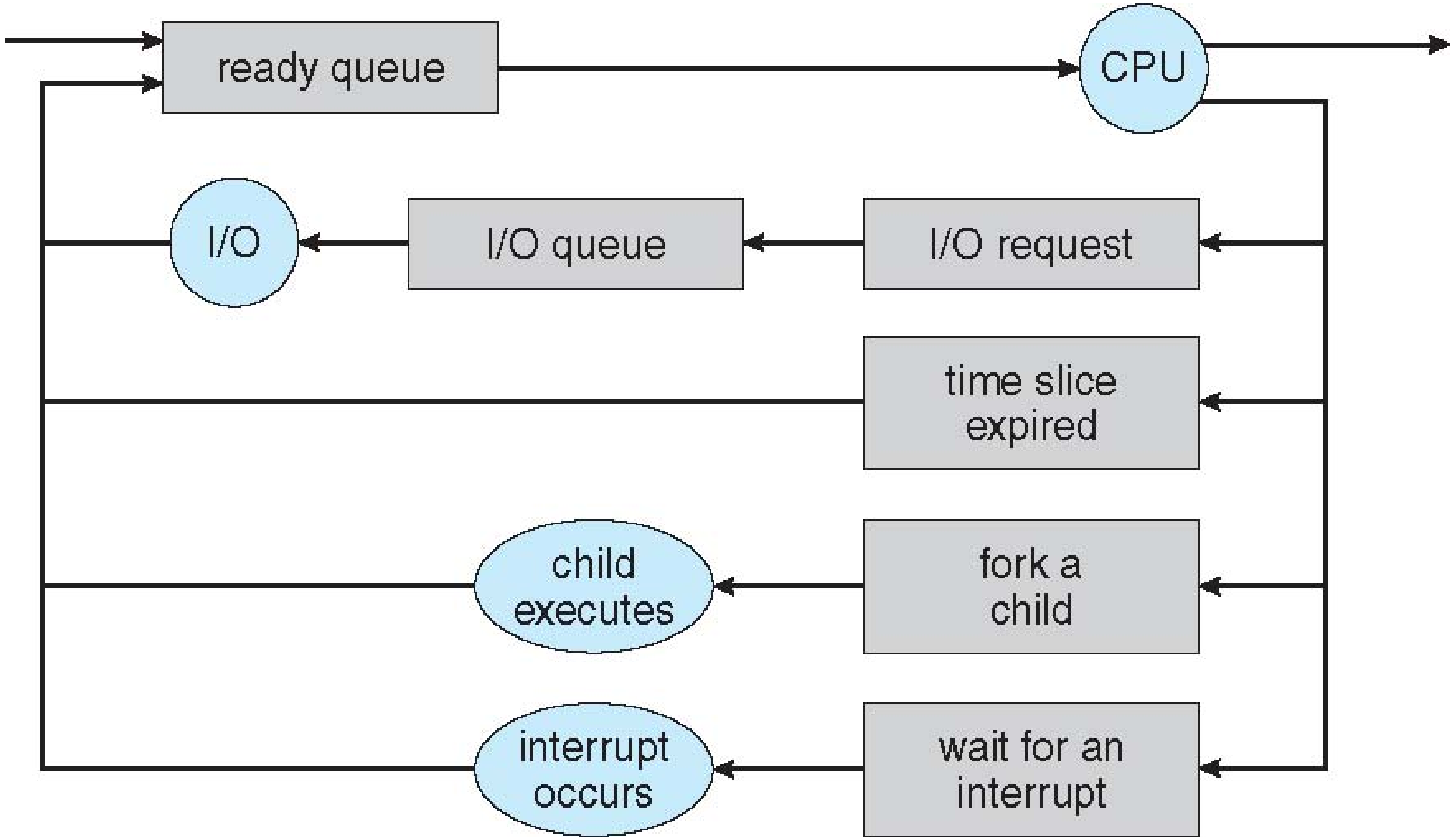
- Maximize CPU use, quickly switch processes onto CPU for time sharing.
- **Process scheduler** selects among available processes for next execution on CPU.
- Maintains **scheduling queues** of processes.
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute.
  - **Device queues** – set of processes waiting for an I/O device.
  - Processes migrate among the various queues.

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

Queueing diagram represents queues, resources, flows.



Schedulers

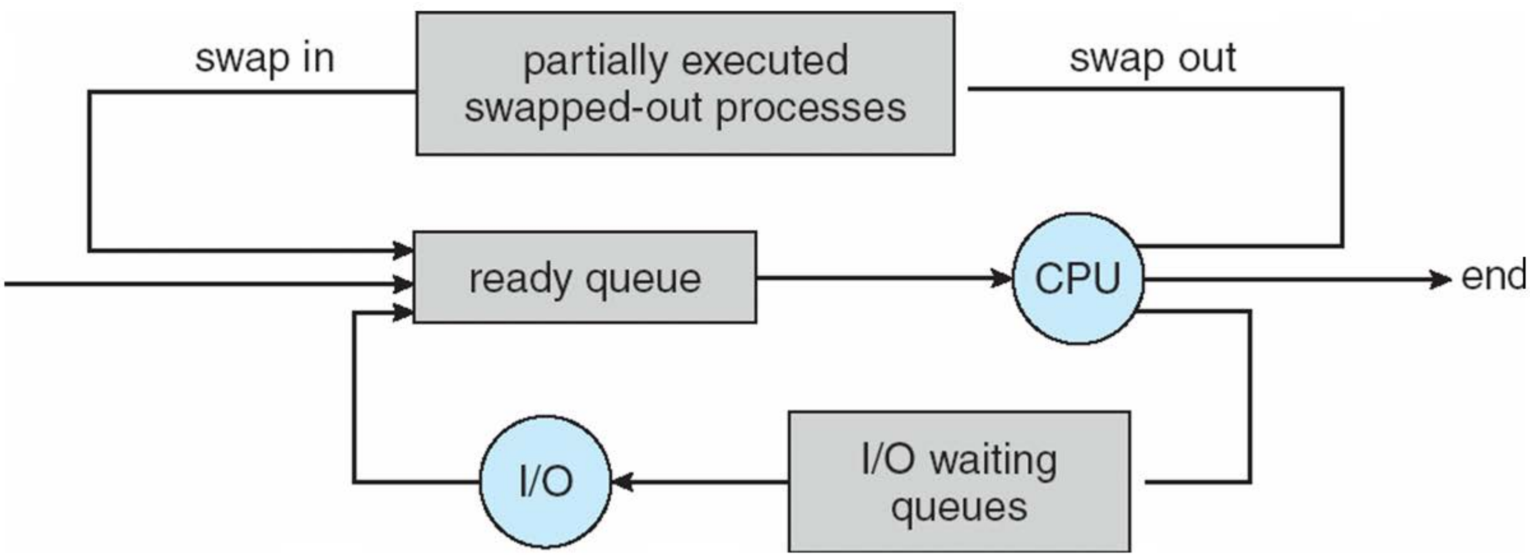
- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue.
  - Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow).
  - The long-term scheduler controls the *degree of multiprogramming.*
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU.
  - Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast).

A process 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

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease.
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping.**



Multitasking in Mobile Systems

- Early systems allowed only one process to run, others suspended.
- Due to screen real estate and user interface limits, iOS has a
  - Single **foreground** process- controlled via user interface.
  - Multiple **background** processes– in memory, running, but not on the display, and with limits.
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback.
- Android (Unix) runs foreground and background, with fewer limits.
  - Background process uses a **service** to perform tasks.
  - Service can keep running even if background process is suspended.
  - Service has no user interface, and small memory use.

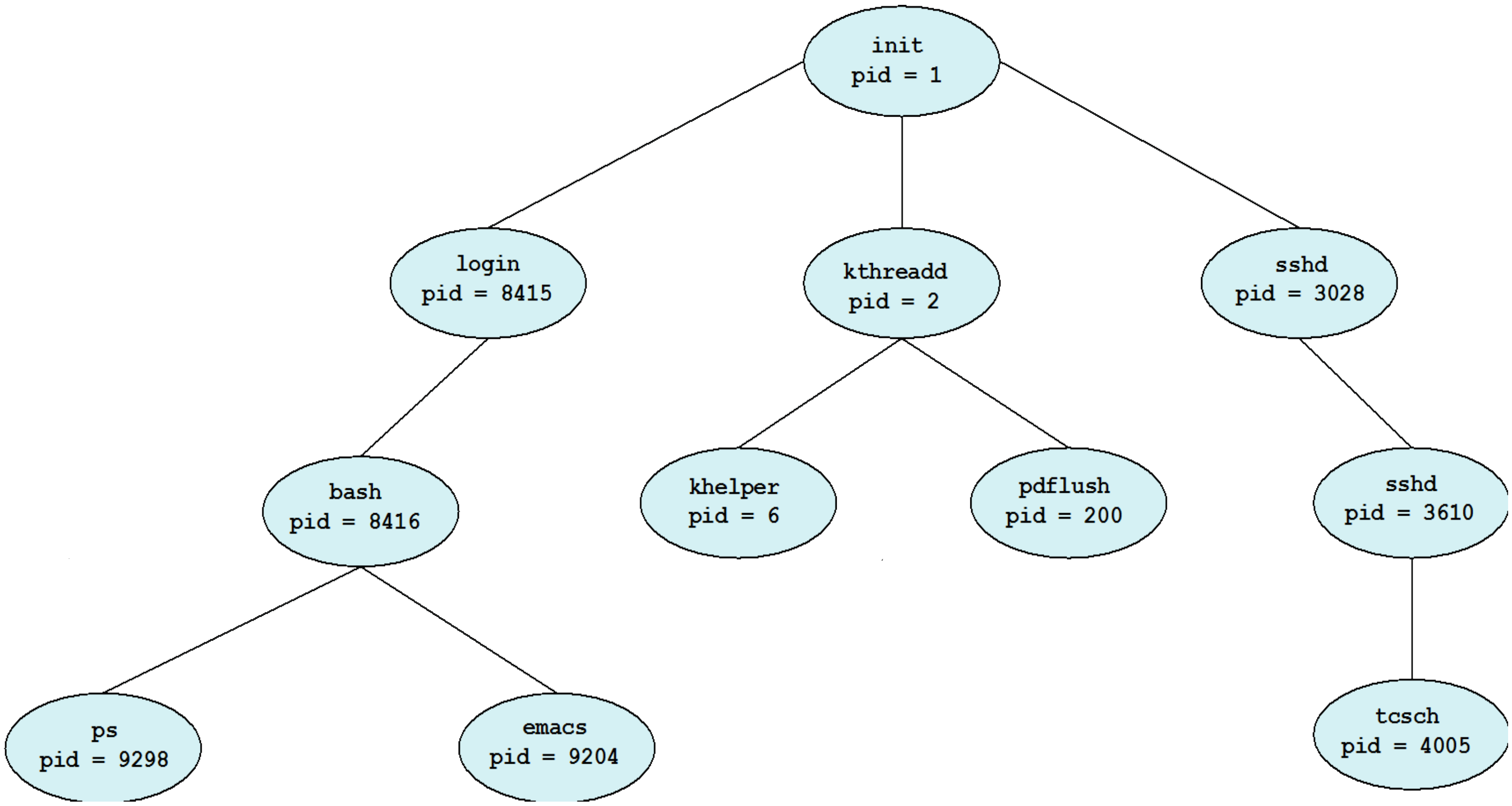
Operations on Processes

- System must provide mechanisms for process creation, termination, and so on as detailed next.

Process Creation

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

A Tree of Processes in Linux



Process Tree vs Precedence Graph

Process Tree: Captures process creation (parent-child) history among processes.

Precedence Graph: Captures fork/join precedences among processes over a single address space.
Nodes: Compound statements.
Edges: Precedence constraints.

Process Creation

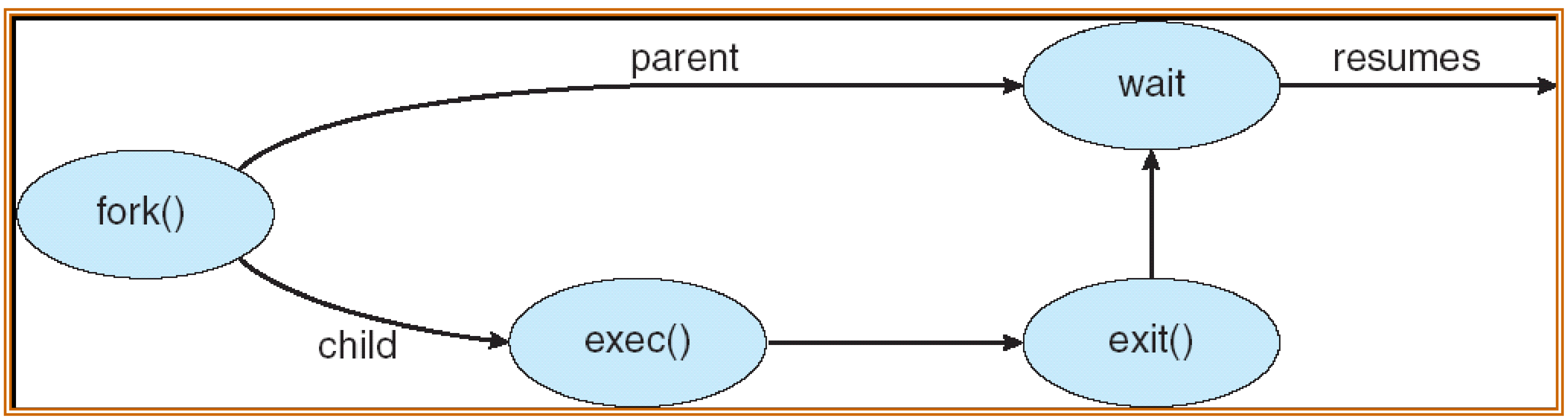
- Parent process creates children processes, which, in turn create other processes, forming a tree of processes--process tree (versus precedence graph).
  - e.g., shell program
  - Process ID

```
sxj63@volatile:~$ ps -l
F S   UID   PID     PPID  C PRI  NI ADDR SZ  WCHAN TTY     TIME     CMD
0 S  5618  22326 22325  0 75    0  -       809 wait     pts/1   00:00:00 bash
0 R  5618  22332 22326  0 76    0  -       588 -          pts/1   00:00:00 ps
```

- 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 **wait()** 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. Alternative: vfork().
  - **exec()** system call (execl(), execlp), etc) used after a **fork()** to replace the process' memory space with a new program.



An Example of Process Creation

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

main()
{
        int pid;

        pid = fork();
        if (pid < 0) { /* error occurs */
                perror("fork() failed");
                exit(1);
        } else if (pid == 0) { /* child process */
                sleep(10);
                execlp("/bin/sleep", "sleep", "10", NULL);
        } else { /* parent process */
                /* parent will wait for the child to complete */
                wait(NULL);
                printf("child complete\n");
                exit(0);
        }
}
```

```
sxj63@volatile:~$ ps -l
F S  UID   PID   PPID  C PRI NI ADDR SZ  WCHAN  TTY    TIME     CMD
0 S  5618 18816 18815  0  75  0  -   811 wait   pts/1  00:00:00 bash
0 S  5618 18983 18816  0  76  0  -   350 wait   pts/1  00:00:00 a.out
0 S  5618 18984 18983  0  77  0  -   449 -      pts/1  00:00:00 sleep
0 R  5618 18985 18816  0  76  0  -   588 -      pts/1  00:00:00 ps
```

Changing Process Address Space

exec system call: The only way in which a program is executed in Unix is for an existing process to issue an exec system call.

exec() call does not change the pid.

The program invoked by exec() inherits: pid, ppid, cwd, file locks, real uid.

The new program can get a new "effective id":
If the set-user-id bit of the program being exec'ed is set then the effective uid is changed to the user-id of the owner of the program file.

Execve(filename, argv, envp)

Variations: execve, execlp, execv, execvp, execle:
p: use current PATH for searching executable.
L: list of arguments
v: argv[] vector
e: pass own environment variable list.

Process Termination

- Process executes last instruction, and asks the operating system to terminate it (**exit()**).
  - Output data from child to parent (via **wait()**).
  - Process's resources are reclaimed by operating system.

- Parent may terminate child processes (e.g., by sending a signal to the child processes, signal KILL).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.
    - An operating system may not allow a child to continue if its parent terminates.
    - Cascading termination.

- *Orphan* and *Zombie*
  - Orphan: A process whose parent has terminated and who has been adopted by the process **init(1)**.
  - Zombie (defunct): A process that has terminated and whose parent has not yet received notification of its termination.

Orphan and Zombie

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

main()
{
        int pid;

        pid = fork();
        if (pid < 0) { /* error occurs */
                exit(1);
        }
        else if (pid == 0) { /* child process */
                sleep(10);
        }
        else { /* parent process */
                exit(0);
        }
}
```

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

main()
{
        int pid;

        pid = fork();
        if (pid < 0) { /* error occurs */
                exit(1);
        }
        else if (pid == 0) { /* child process */
                exit(0);
        }
        else { /* parent process */
                sleep(10);
        }
}
```

C Program in Unix: Forking Separate Process

```
#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);
}
```

Multiprocess Browser Architectures

- 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 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.
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits.
  - **Plug-in** process for each type of plug-in.

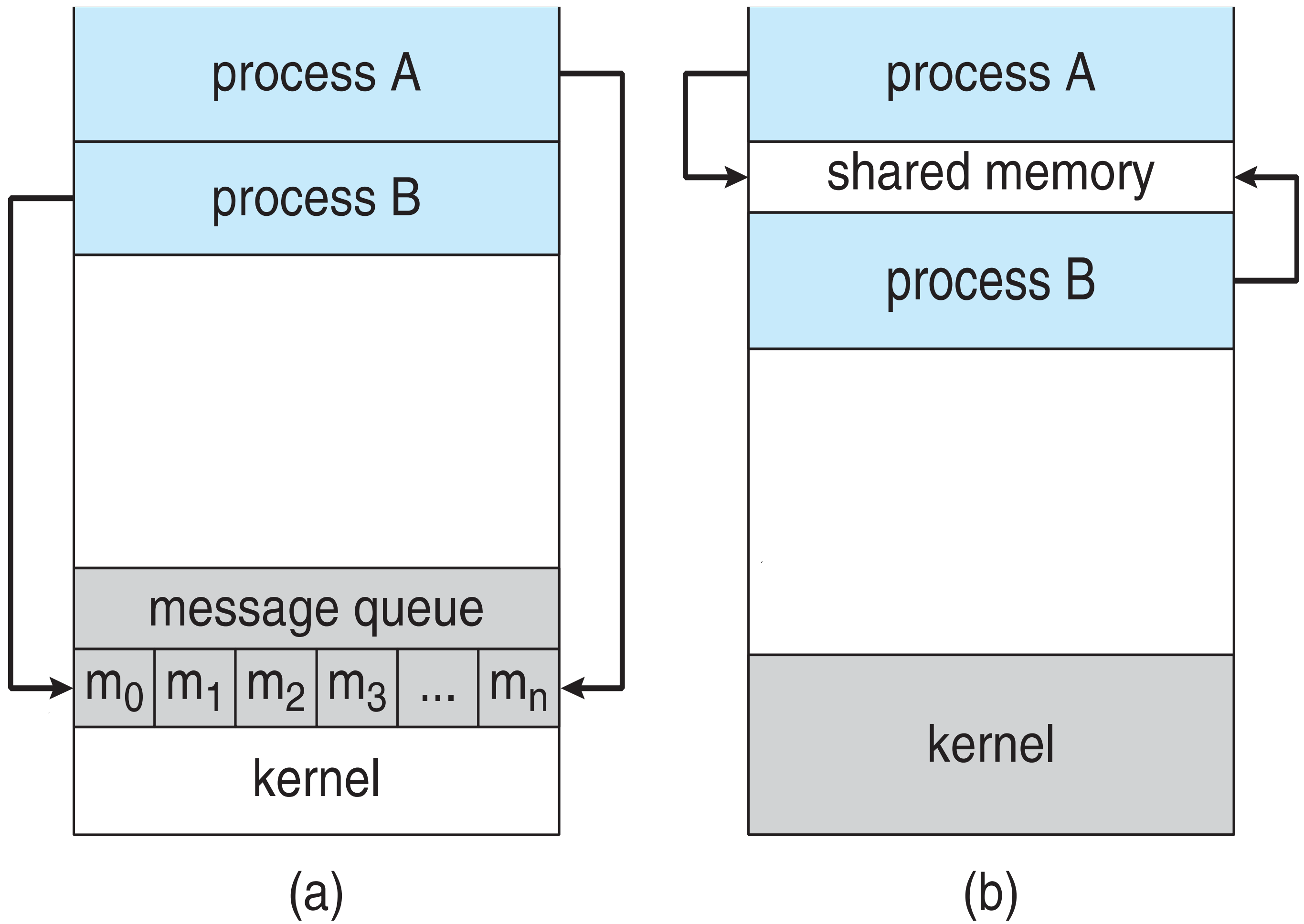
Cooperating Processes

- At the beginning of the course, we saw limited cooperation between parent and child processes
  - wait(), signal(), resource sharing on process creation
  - But these are not enough.
    Example: backend database server process to provide answers to clients.

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

Interprocess Communication Models



Producer-Consumer Problem

- We have seen one paradigm for cooperating processes: Producer-Consumer model:
  - *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.

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*)

Interprocess Communication – Message Passing

- 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., direct or indirect, synchronous or asynchronous, automatic or explicit buffering).

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?

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.

The rest of the slides are FYI only, and are already covered in depth in recitations.

Note: Your next two assignments will need this information.

Examples of IPC Systems - POSIX

- 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);
    ```

  - Now the process could write to the shared memory

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

IPC POSIX Producer

```
#include <stdio.h>
#include <stlib.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_RDRW, 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

```
#include <stdio.h>
#include <stlib.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 and Notify
  - Only three system calls needed for message transfer.

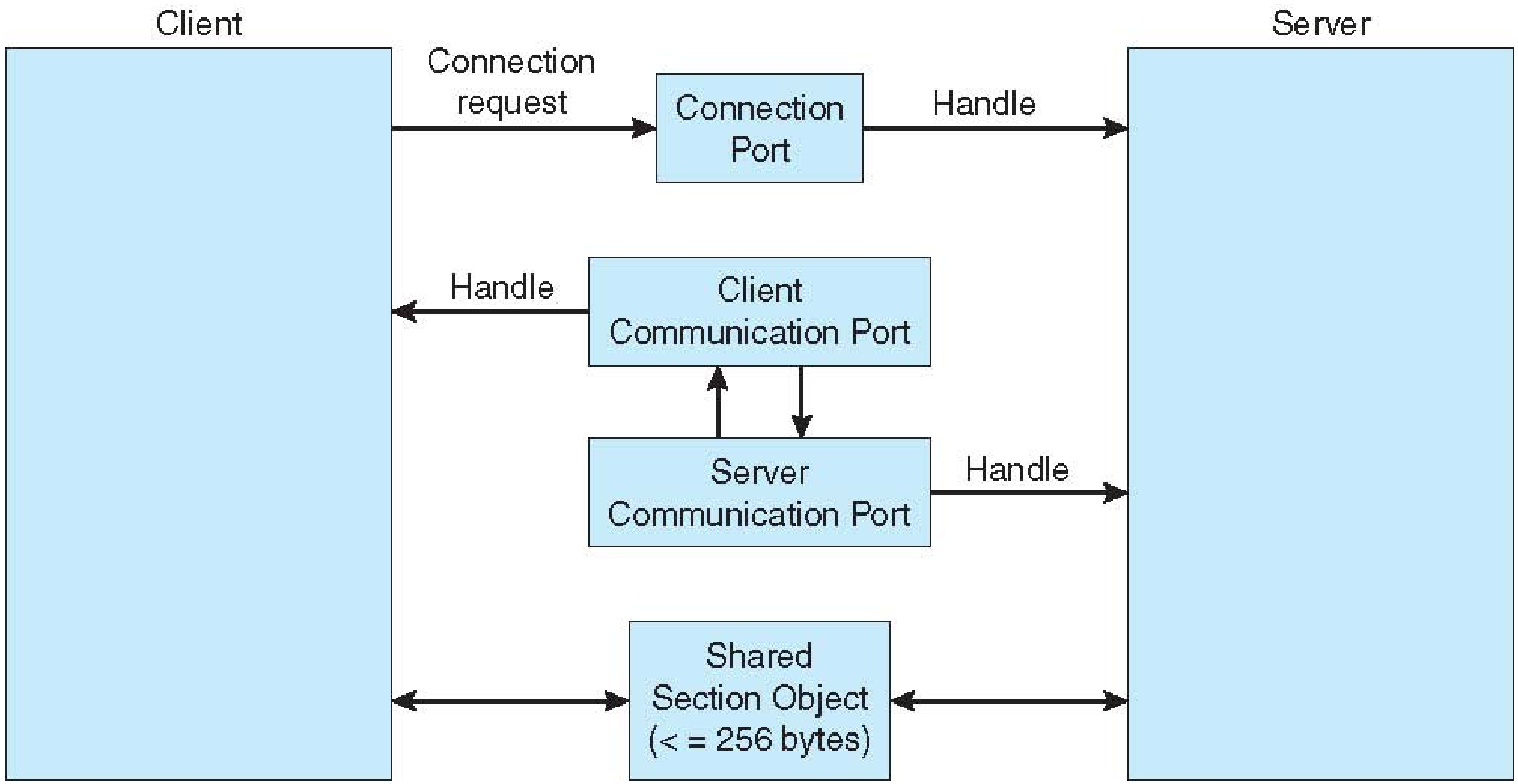
    `msg_send(), msg_receive(), msg_rpc()`
  - Mailboxes needed for communication, created via

    `port_allocate()`
  - Send and receive are flexible, for example four 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:
    - The client opens a handle to the subsystem's **connection port** object.
    - The client sends a connection request.
    - The server creates two private **communication ports** and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Local Procedure Calls in Windows XP



Communications in Client-Server Systems

- Sockets (Used to be covered in this course; now covered in EECS 325).

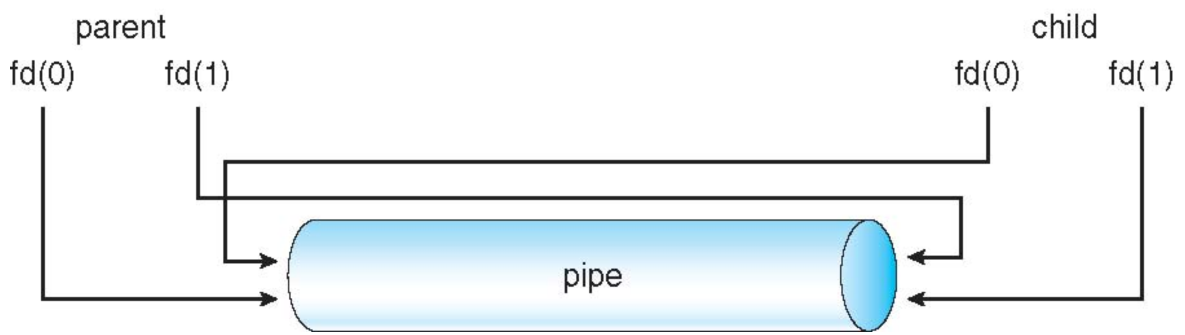
- Remote Procedure Calls (will see in Week 12 approximately, and will have an assignment).

- Pipes (you have already seen in recitations; may have an assignment).

- Remote Method Invocation (Java).

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
  - Require parent-child relationship between communicating processes.



- Windows calls these **anonymous pipes**
- See Unix and Windows code samples in textbook.

Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems