ECE30021/ITP30002 Operating System

Processes

(OSC : Chapter 3)

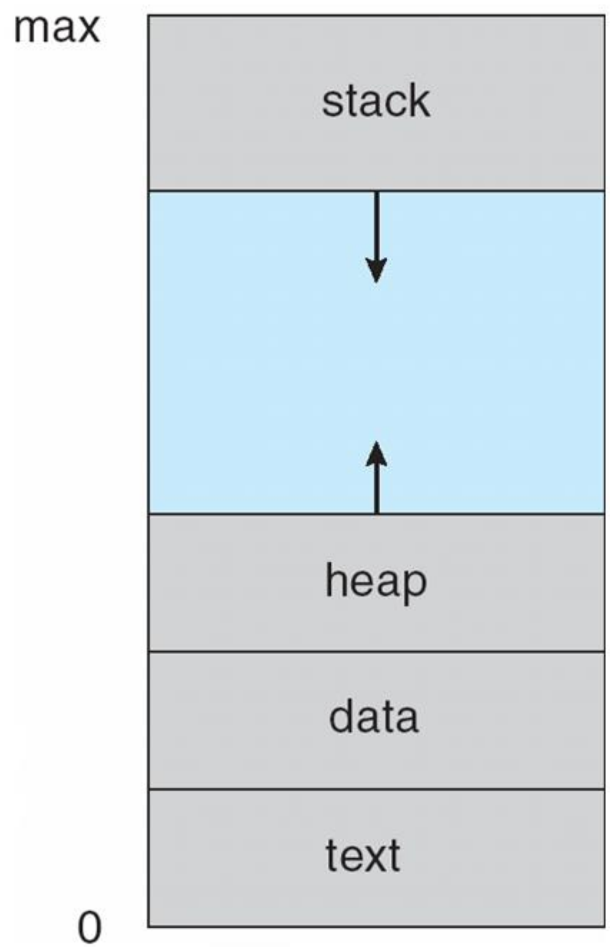
This lecture note is taken from the instructor's resource of Operating System Concept, 9/e and then partly edited/revised by Shin Hong.

Motivation

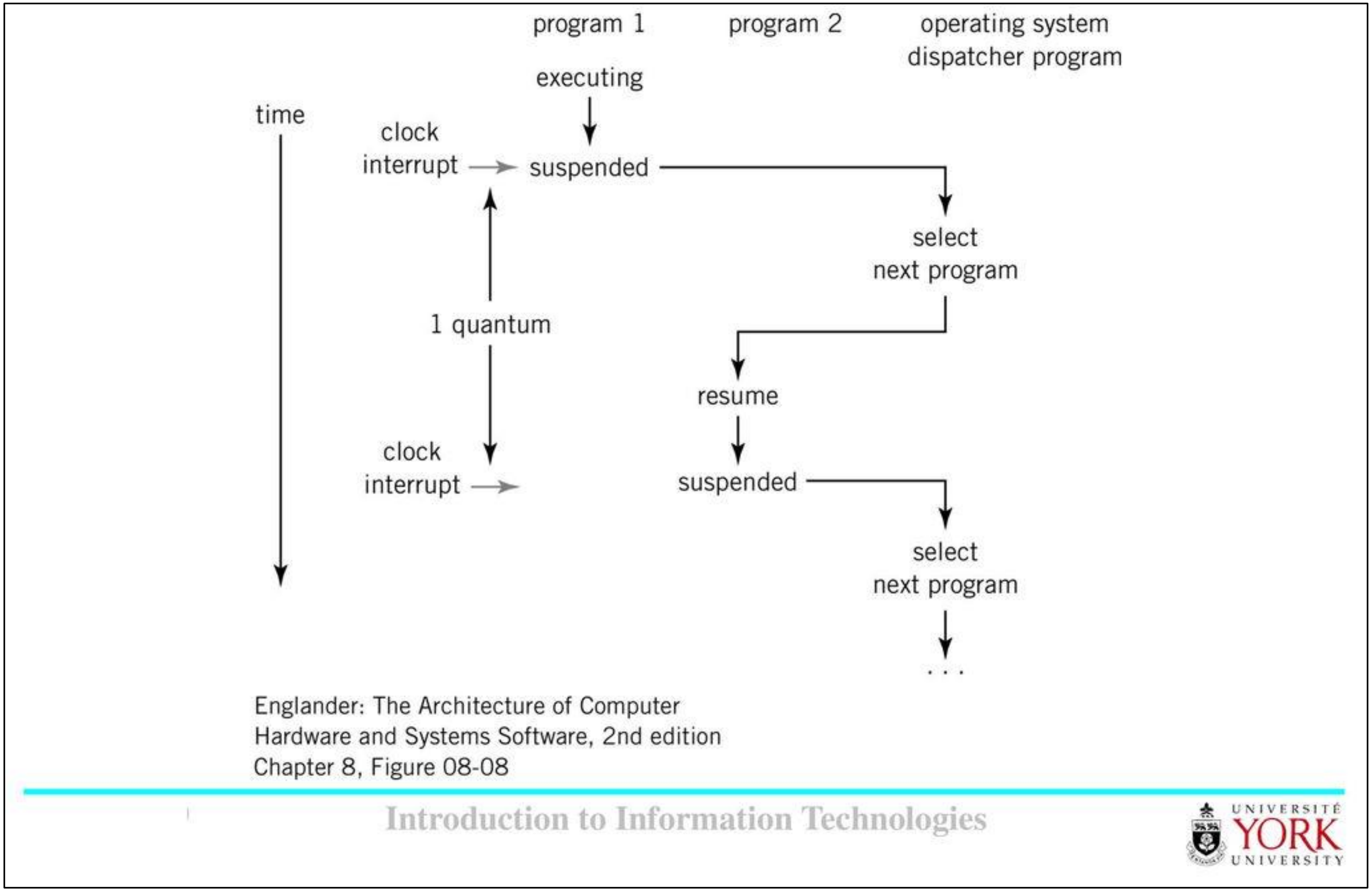
Process Concept

- **Process**: a running instance of a program
  - a process progresses in a instruction-by-instruction fashion
  - program is a *passive* entity stored on disk (executable file), process is *active*
    - Program becomes process when executable file loaded into memory
    - Multiple users may run multiple processes executing the same program
- A process consists of multiple entities
  - current state of the processor including program counter and registers
  - program code (also called text section)
  - stack containing temporary data
    - Function parameters, return addresses, local variables
  - data section containing global variables
  - heap containing memory dynamically allocated during run time

Process in Memory

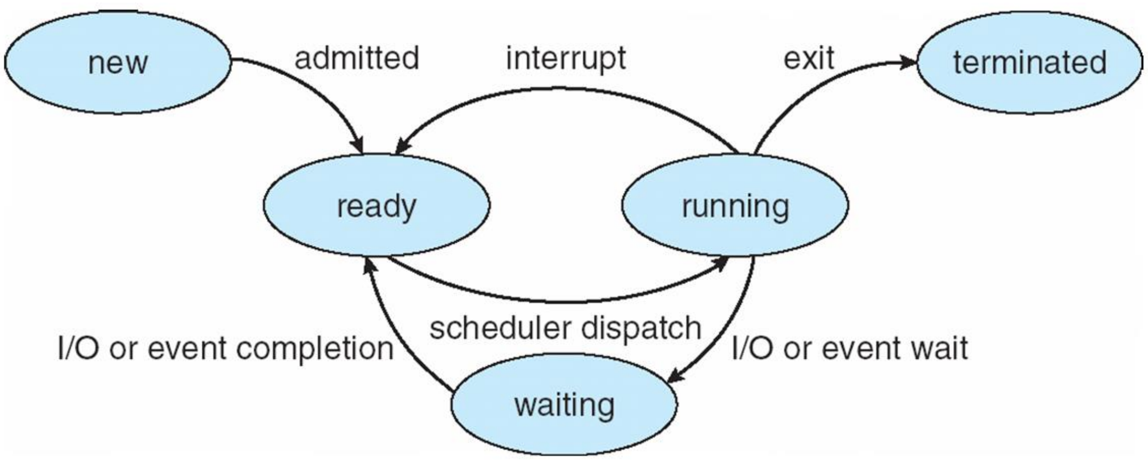


Time Sharing of Multiple Processes



Process State

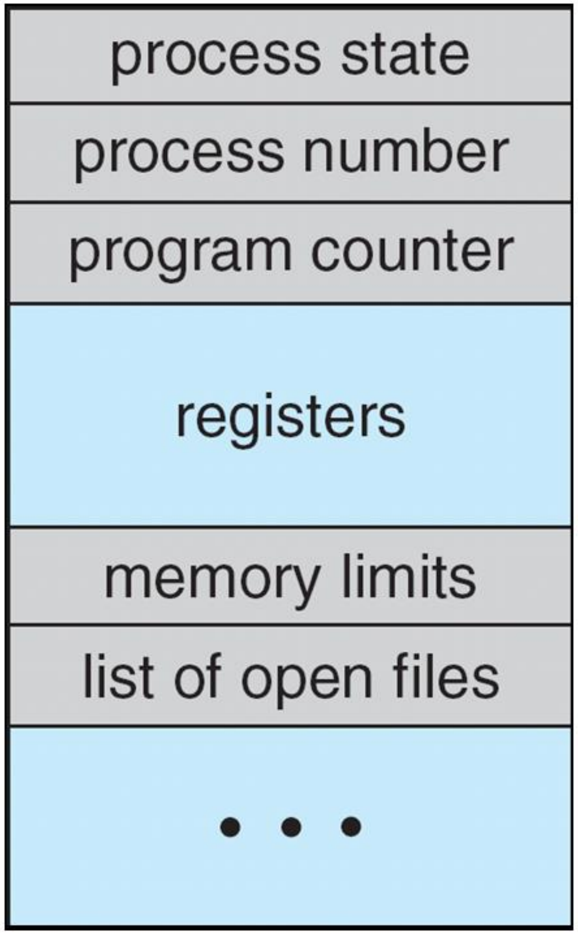
- As a process executes, the process state changes
  - **new**: The process is being created
  - **ready**: The process is waiting to be assigned to a processor
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **terminated**: The process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



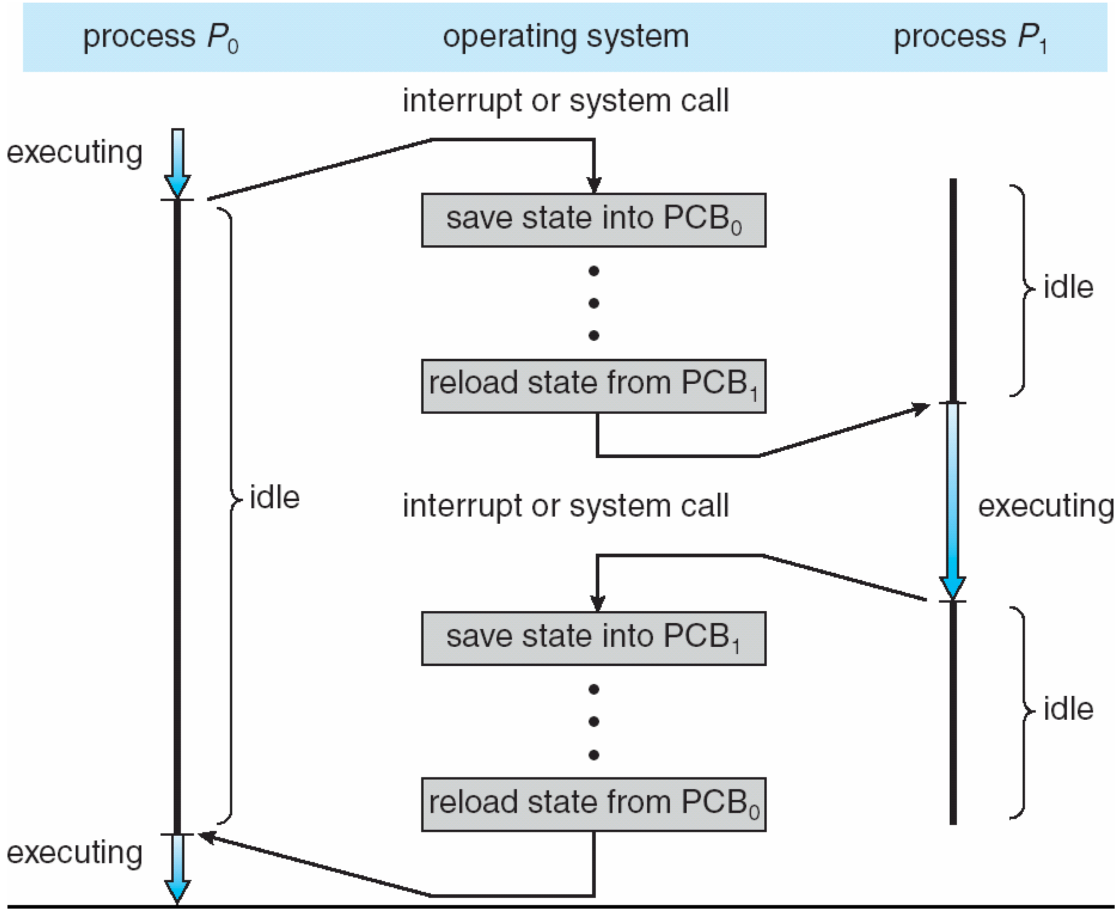
Process Control Block (PCB)

Information associated with each process (also called task control block)

- Process state
- Program counter: a 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
- I/O status information: I/O devices allocated to process, list of open files
- Accounting information: CPU used, clock time elapsed since start, time limits



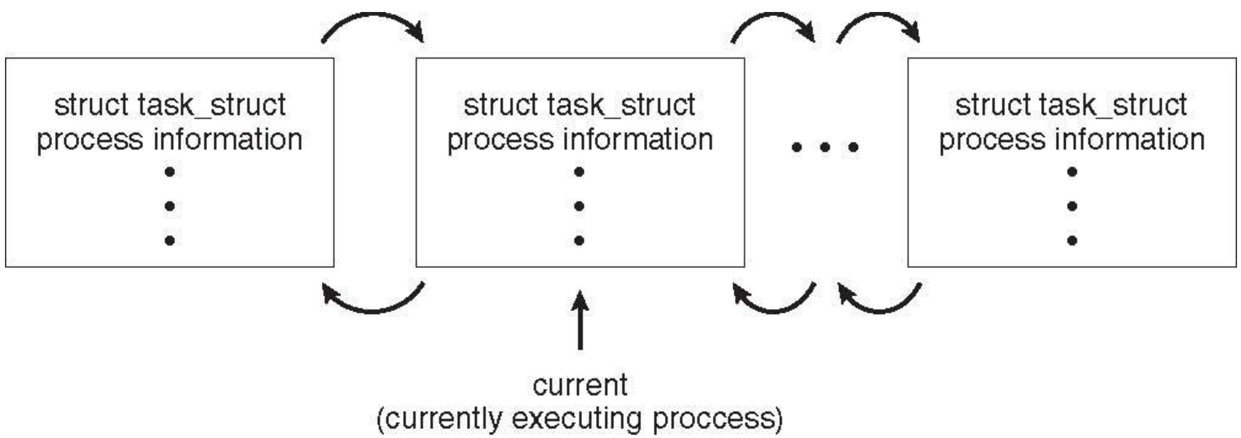
CPU Switch From Process to Process



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 info. */
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;         /* addr. space of this process */
```



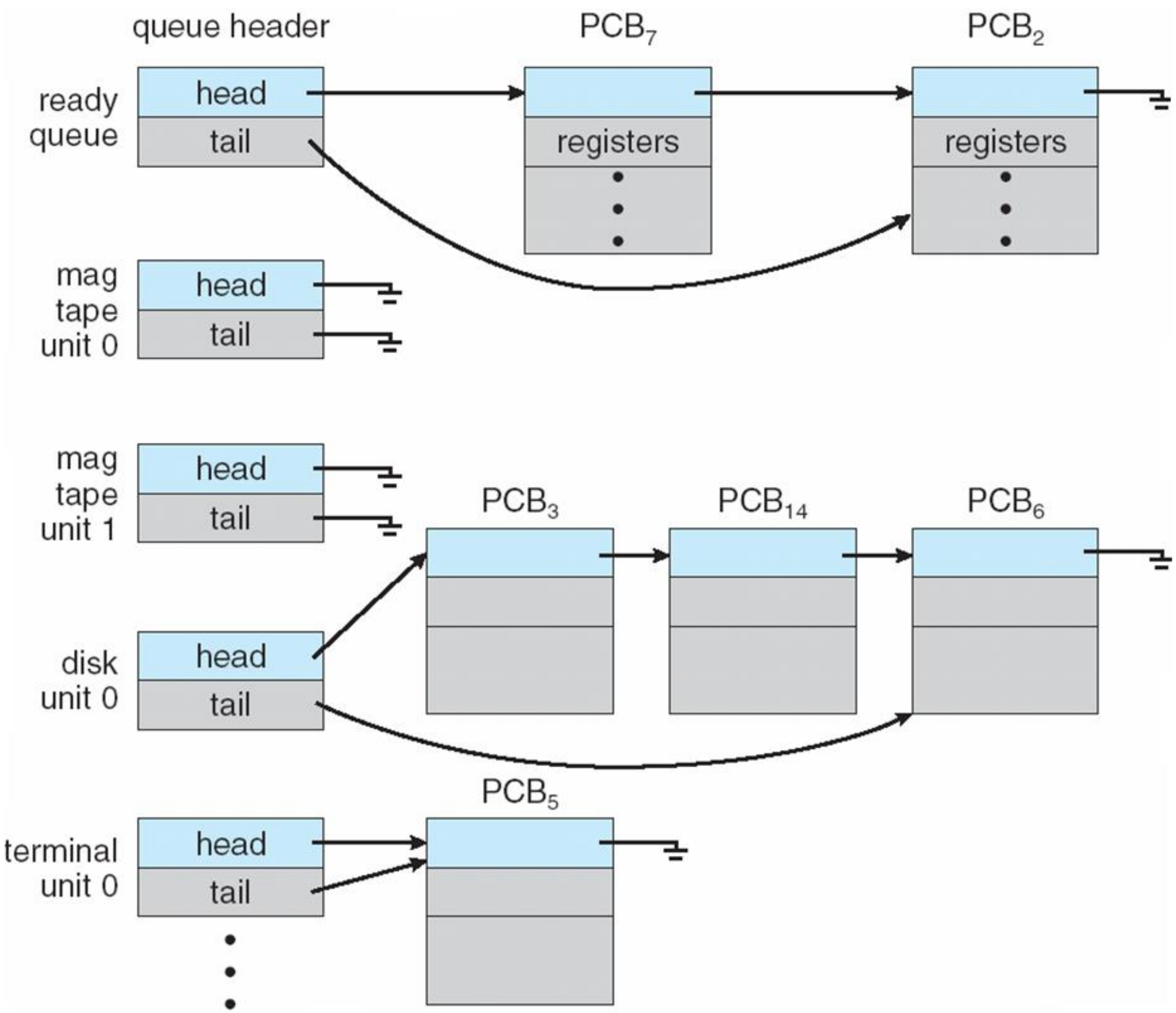
Threads

- So far, process has a single flow of execution
- Consider having multiple program counters per process. Then, multiple locations can execute simultaneously
    - Having multiple execution (control) flows is called *multi-threading*
- Must then have storage for thread details, multiple program counters in PCB
- See next chapter

Process Scheduling

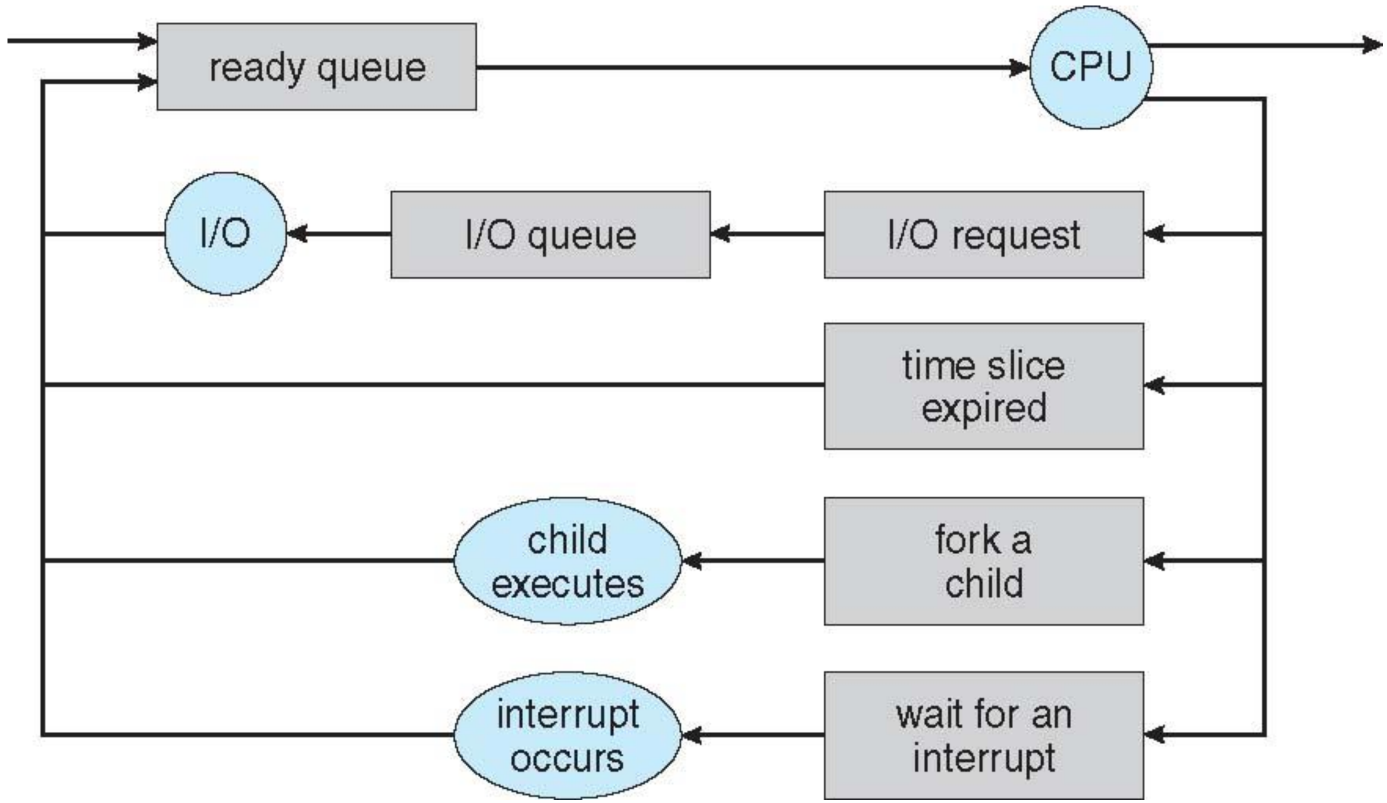
- Process scheduler selects one of available processes (ready state) for next execution on CPU
  - a scheduler is object to maximize CPU utilization and/or minimize response latency
- OS maintains scheduling queues of processes
  - **Job queue**: a set of all processes in the system
  - **Ready queue**: a set of all processes residing in main memory, ready and waiting to execute
  - **Device queues**: a set of processes waiting for an I/O device

Ready Queue And Various I/O Device Queues

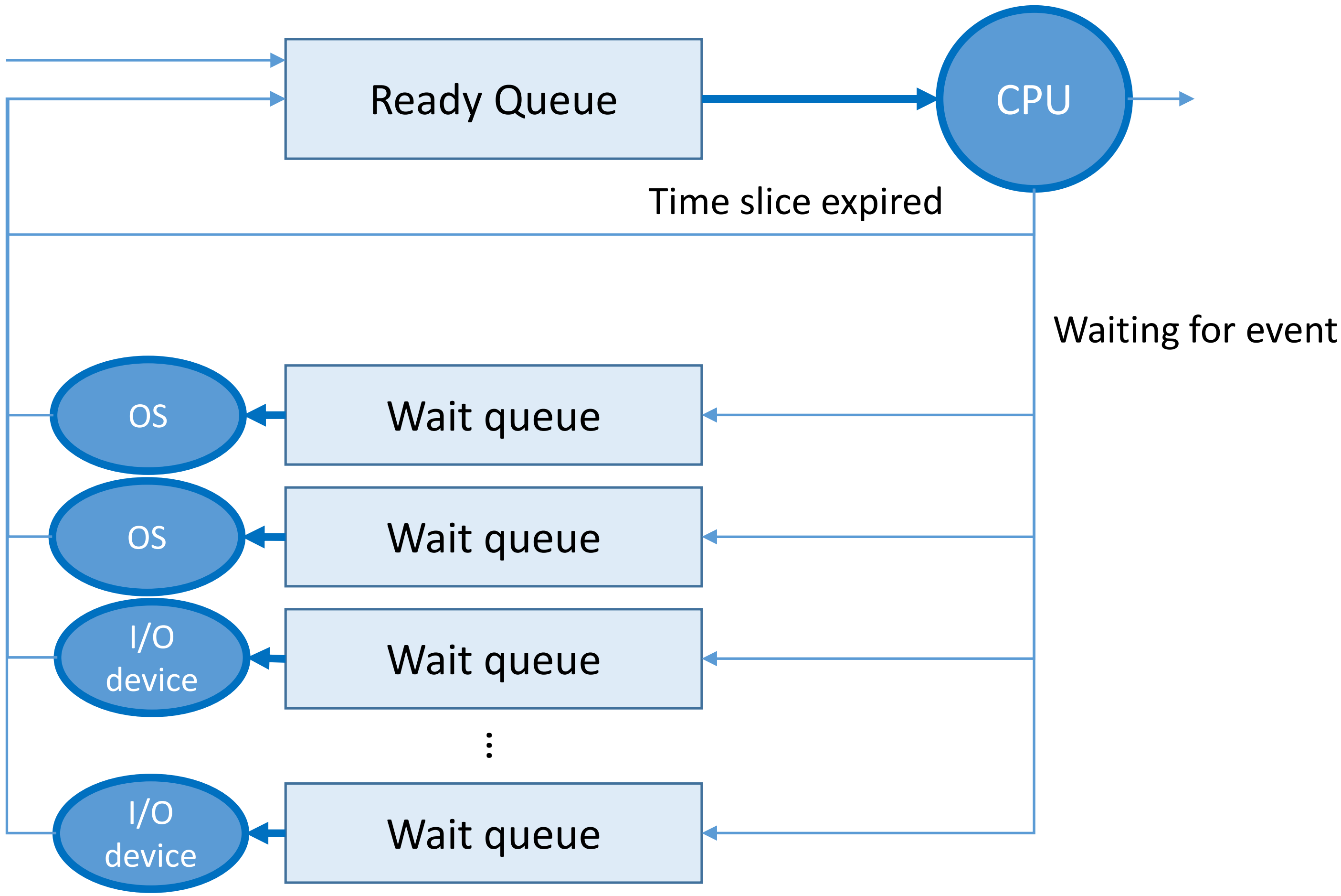


Representation of Process Scheduling

Queueing diagram represents queues, resources, flows



Scheduling Lifecycle of Process

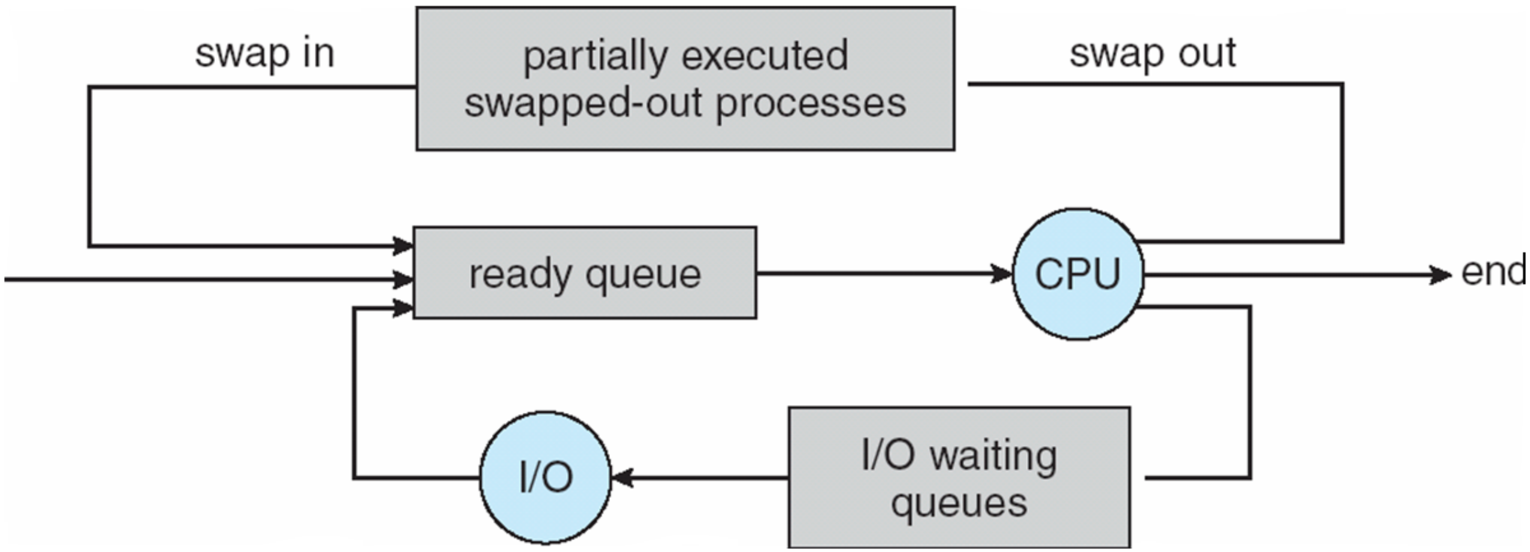


Schedulers

- Short-term scheduler (or CPU scheduler): selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds)
- Long-term scheduler (or job scheduler): selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes)
  - The long-term scheduler controls the degree of multiprogramming
- Processes can be described as either:
  - I/O-bound process: spends more time doing I/O than computations, many short CPU bursts
  - CPU-bound process: spends more time doing computations; few very long CPU bursts

Addition of Medium Term Scheduling

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



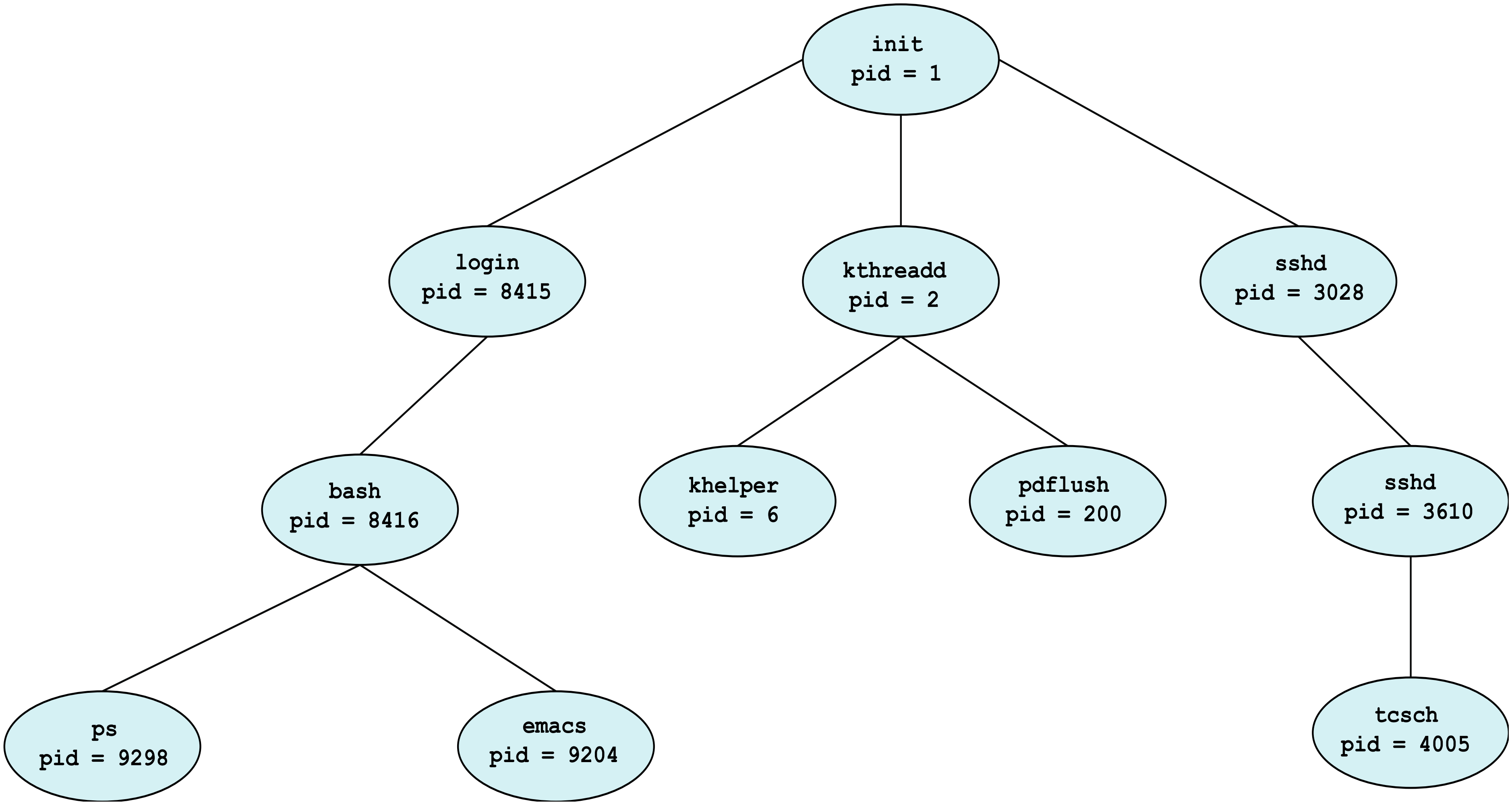
Context Switch

- Context-switch is an action by OS that switches the running process to another one in the ready queue
  - save the state of the current process
  - select one of the ready processes
  - load the saved state for the selected process
- 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, the longer the context switch
  - Some hardware provides multiple sets of registers per CPU to load multiple contexts at once

Process Creation

- A process is identified and managed via a process identifier (pid)
- A parent process can spawn a child process, and a parent process can run concurrently with its child process
  - A process can spawn multiple children processes
  - A child process, in turn create other processes, forming a tree of processes
  - A parent can wait until a child or children terminate
- A parent and its children may share resources
  - Children may share a subset of parent's resources
- UNIX examples
  - a system call **`fork()`** system call creates a new process
  - a child process duplicates the memory of its parent

A Tree of Processes in Linux



Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system

- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some OSes do not allow a child to exist 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.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process

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

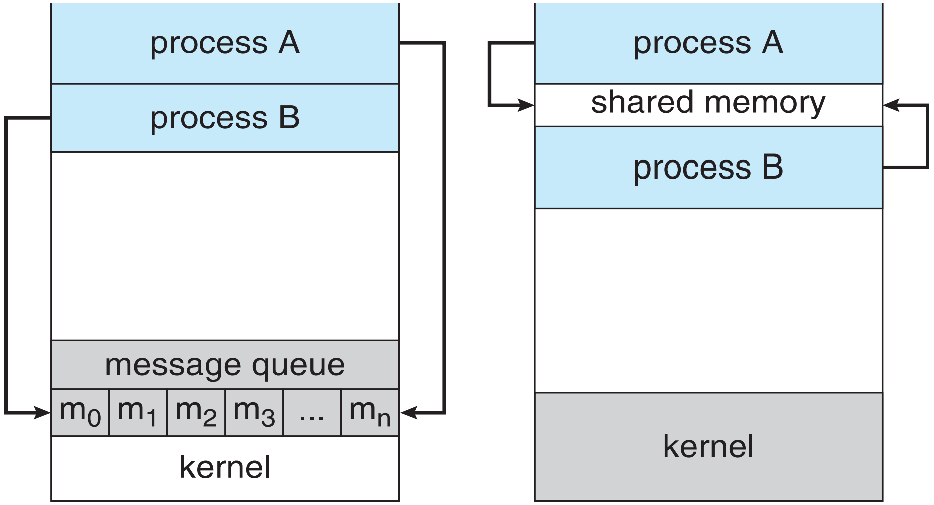
- If no parent waiting (did not invoke `wait()`) process is a zombie
- If parent terminated without invoking `wait`, process is an orphan

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as a single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome is multiprocess with 3 different processes:
  - Browser process manages user interface, disk and network I/O
  - Renderer process renders web pages, deals with HTML and 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 within a system may be ***independent*** or ***cooperating***
  - Cooperating process can affect or be affected by other processes, including sharing data
- Cooperating processes need **inter-process communication** (**IPC**)
  - **Shared memory**
  - **Message passing**



(a) Message passing. (b) shared memory.

Message Passing

- Mechanism for processes to communicate with each other without resorting to shared variables
  - If processes *P* and *Q* wish to communicate, they need to:
    - Establish a ***communication link*** between them
    - Exchange messages via send/receive
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)

- Implementation issue
  - Physical media: shared memory, Hardware bus, Network
  - Direct or indirect
  - Synchronous or asynchronous
  - 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
- 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

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
  - 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
  - Link may be unidirectional or bi-directional

Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - the sender is blocked until the message is received
  - the receiver is blocked until a message is available
- Non-blocking is considered asynchronous
  - the sender sends the message and continue
  - the receiver receives a valid message, or null message
- Different combinations possible
  - If both send and receive are blocking, we have a rendezvous

Buffering

- Queue of messages attached to the link.
- 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

Examples of IPC Systems - POSIX

- How to use POSIX Shared Memory
  - Process first creates/opens shared memory segment as a file

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

  - Set the size of the object

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

  - Map the shared memory file to a memory

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

  - Now the process could write to the shared memory

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

Producer-Consumer Example

```
#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;
}
```

```
#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;
}
```

Pipes

- Acts as a channel allowing two processes to communicate
  - **Ordinary pipes** are typically created by a parent process to communicate with a child process; they cannot be accessed from outside the process that created it.
  - **Named pipes** can be accessed without a parent-child relationship
- Ex. Ordinary pipe

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

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

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]);
}
```

Example

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



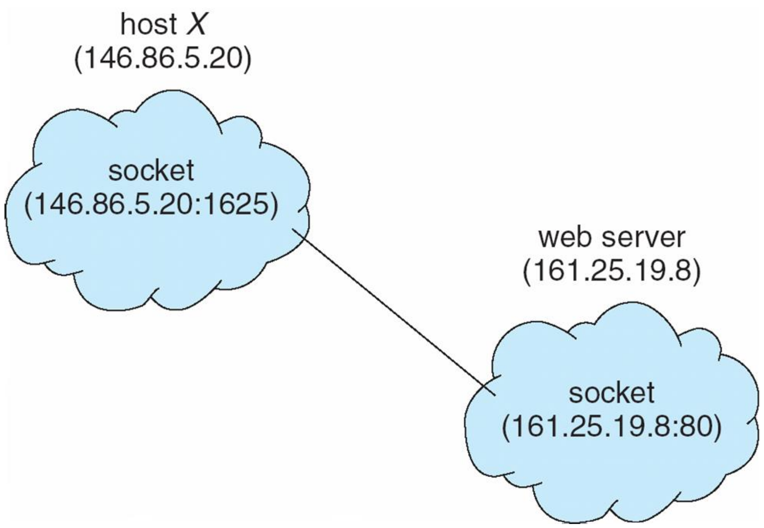
```
/* create the pipe */
if (pipe(fd) == -1) {
  fprintf(stderr,"Pipe failed");
  return 1;
}
```



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

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]);
}
```

Sockets

- A **socket** is defined as 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
- The 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



Remote Procedure Calls (RPC)

- RPC abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- Stubs – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and marshalls the parameters
  - Data representation handled via External Data Representation (XDL) format to account for different architectures
  - e.g., Big-endian and little-endian
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server, and then send back the output to the client
  - Remote communication has more failure scenarios than local
  - OS typically provides a rendezvous (or matchmaker) service to connect client and server

Execution of RPC