

# Operating Systems

Isfahan University of Technology  
Electrical and Computer Engineering Department

Zeinab Zali

## Process Management

Reference: Operating System Concepts book slides



# Process Concept

- An operating system executes a variety of **programs** that run as a **process**.
- **Process** – a **program** in **execution**; process execution must progress in sequential fashion. **No parallel** execution of instructions of **a single process**
- **Program** is **passive** entity stored on disk (**executable file**); **process** is **active**
- **Program** becomes **process** when an **executable** file is **loaded** into memory
  - Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- **One program** can be **several processes**
  - Consider multiple users executing the same program





# Process Parts

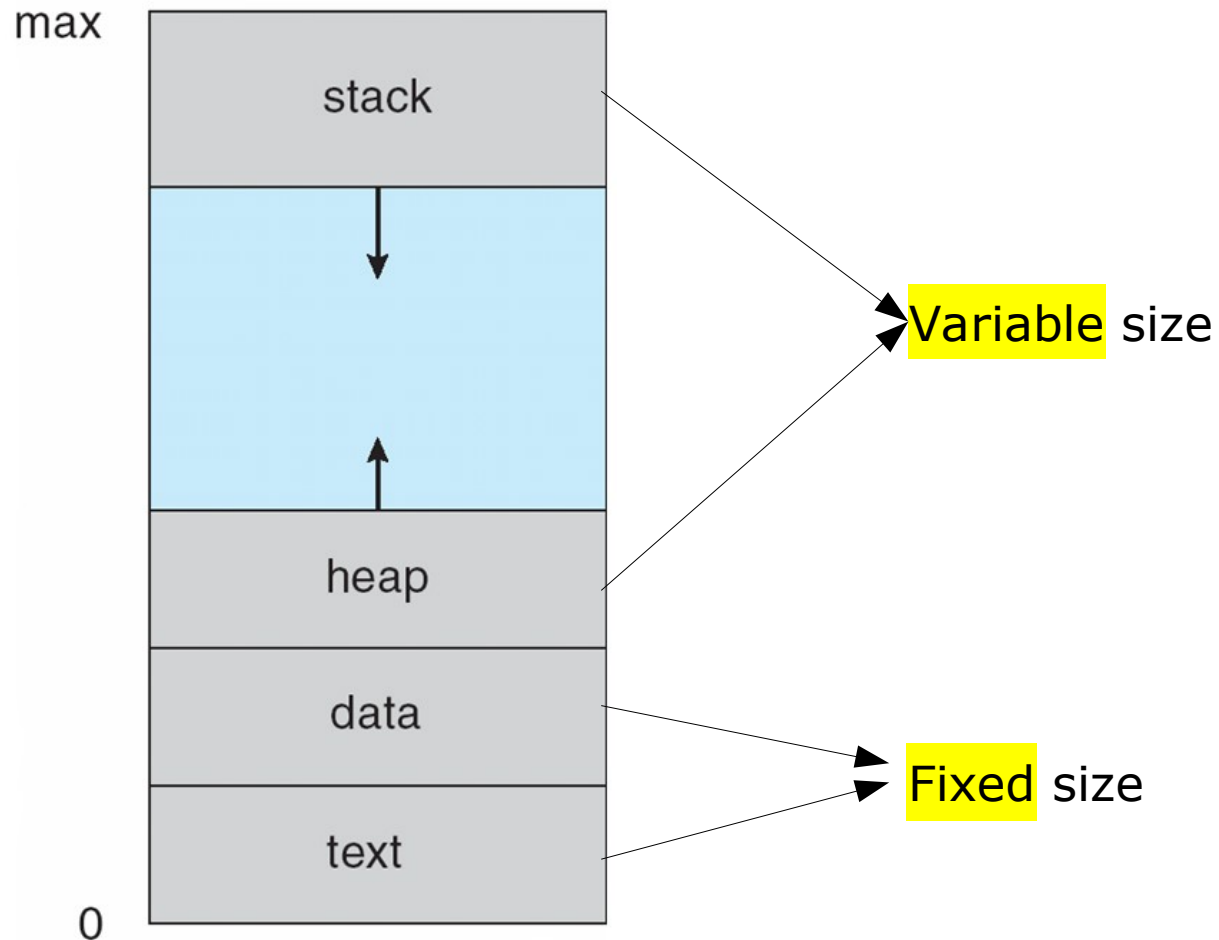
---

- **Text section:** the executable code
- **Data section:** global variables
- **Heap section:** memory that is dynamically allocated during program run time
- **Stack section:** temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)





# Process in Memory

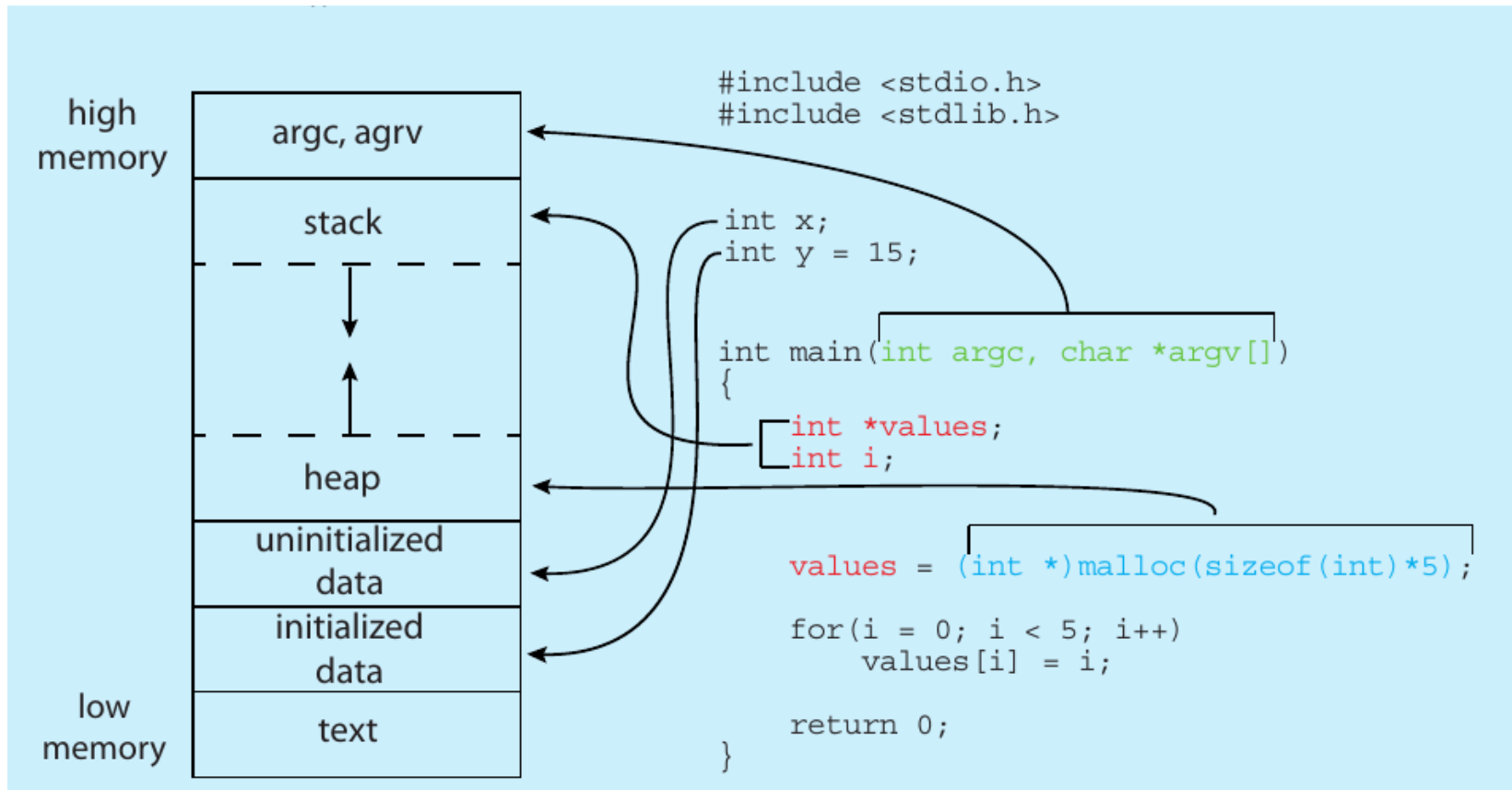


**stack** and **heap** sections grow toward one another, the operating system must ensure they **do not overlap** one another





# Memory Layout of a C Program





# Process State

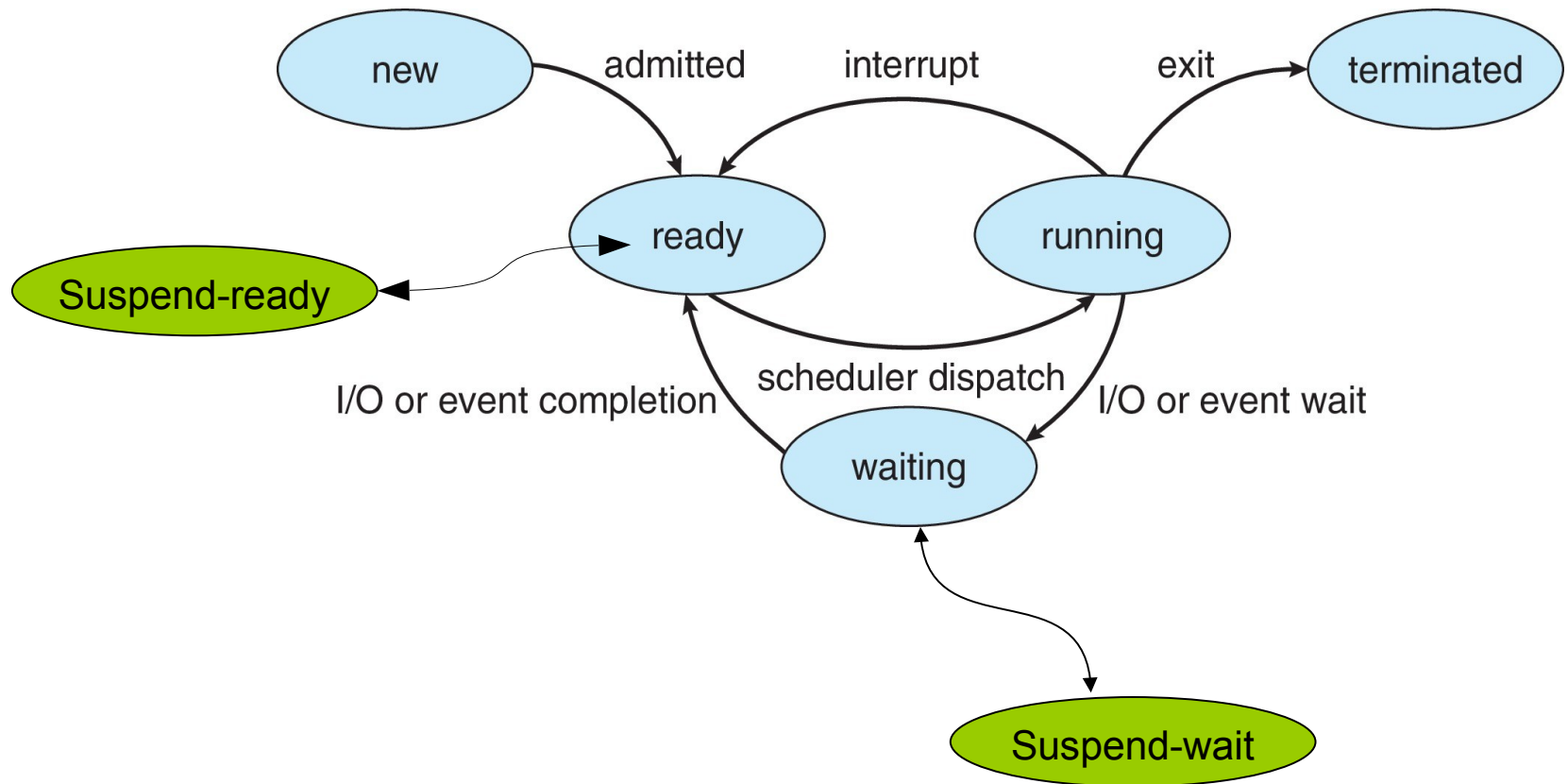
- As a process executes, it changes **state**
  - **New:** The process is being created
  - **Running:** Instructions are being executed
  - **Waiting:** The process is waiting for some event to occur
  - **Ready:** The process is waiting to be assigned to a processor
  - **Terminated:** The process has finished execution

With single CPU, only One program is running while many programs may be ready or waiting





# Diagram of Process State





# 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

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

All processor designs include a register or set of registers, often known as the **program status word (PSW)**, that contains status information.







# 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**
- Explore in detail in Chapter 4



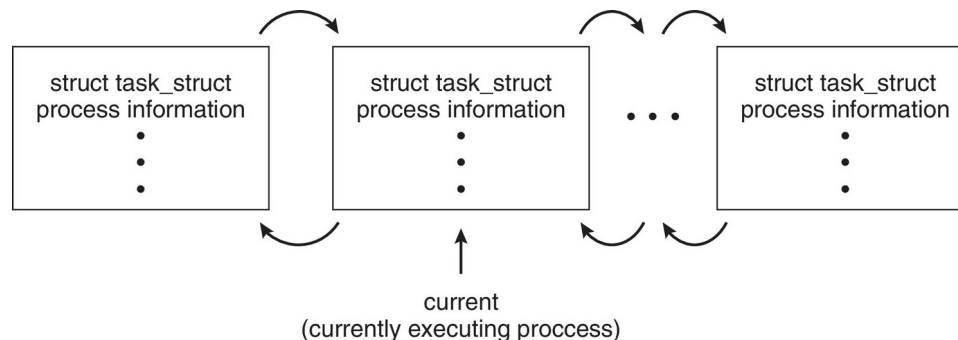


# Process Representation in Linux

Represented by the C structure `task_struct`

`<include/linux/sched.h>`

```
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 */
```



Look inside sched.h and find the corresponding data structures to this chapter

```
<linux/sched.h>
```

```
rb_node
```

```
Rq (running queue)
```

```
State (runnable, unrunnable, stopped)
```



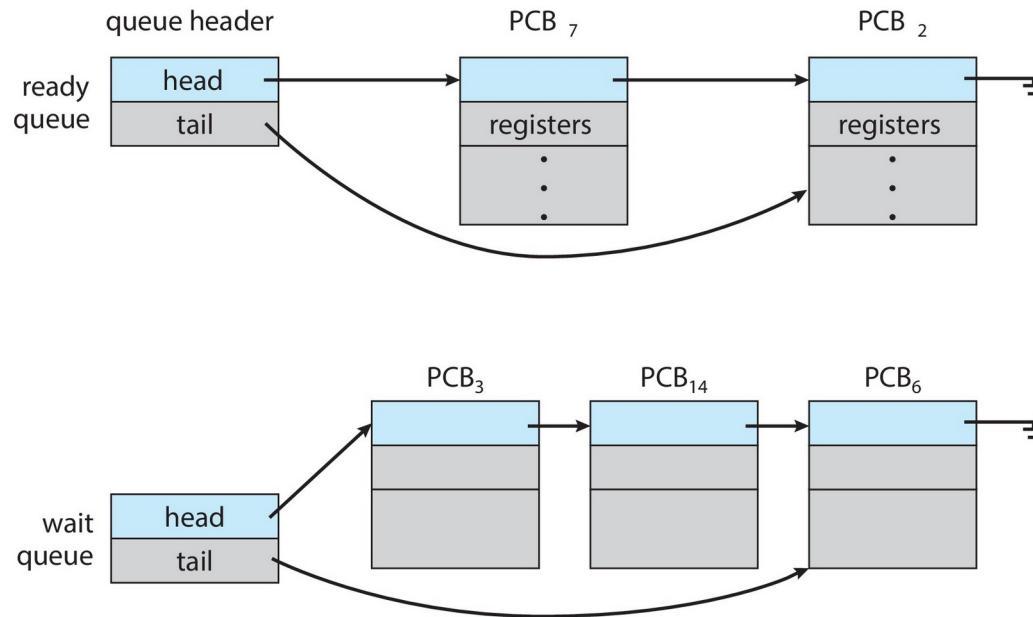
# Process Scheduling

- **Process scheduler** selects among available processes for next execution on **CPU core**
- Goal -- **Maximize CPU** use, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes **residing** in **main memory**, **ready** and **waiting** to execute
  - **Wait queues** – set of processes **waiting** for an **event** (i.e., I/O)
  - **Processes** **migrate** among the **various queues**



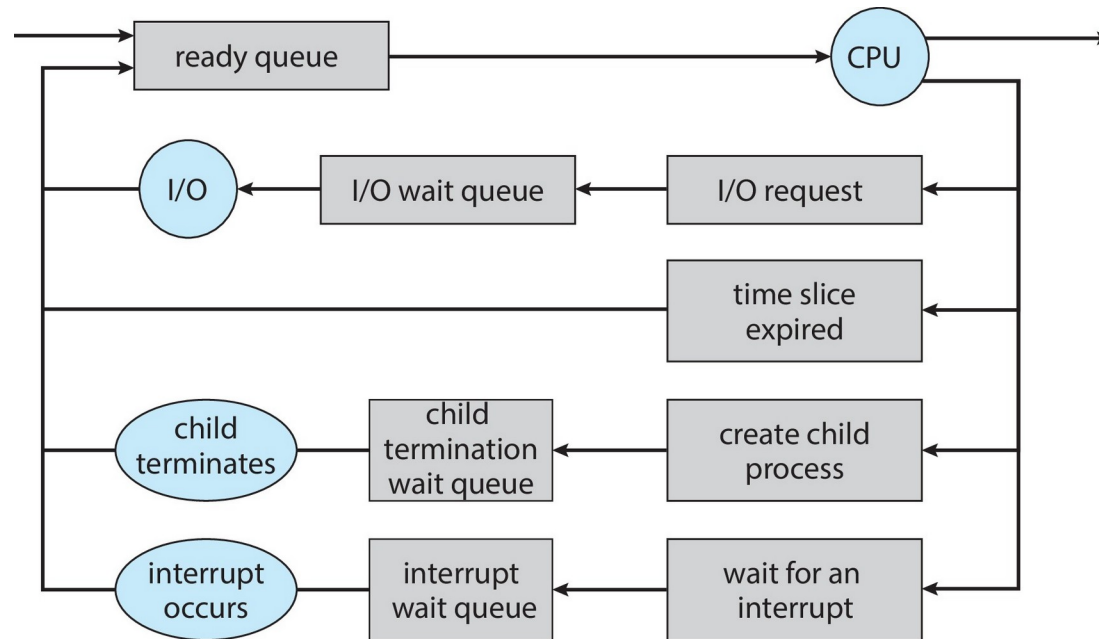


# Ready and Wait Queues





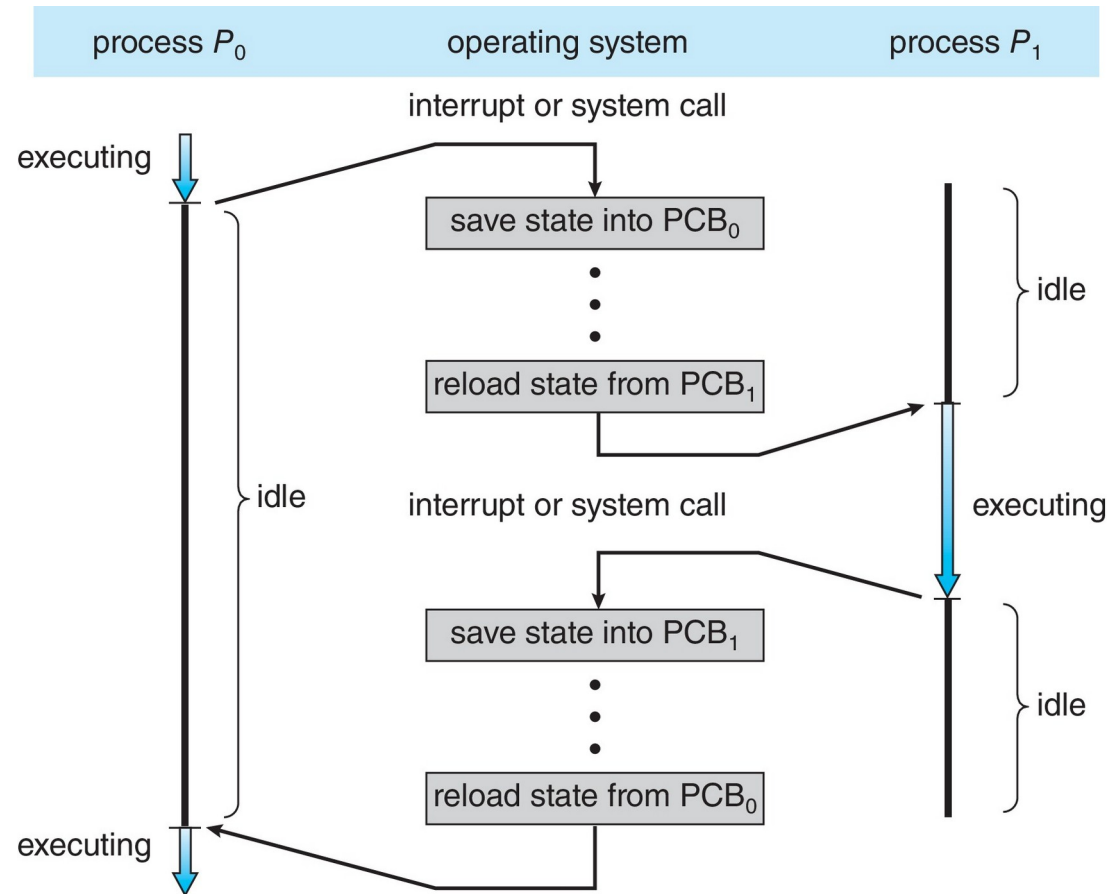
# Representation of Process Scheduling





# CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.

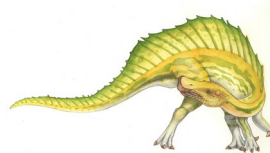




# Context Switch

- 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 **pure overhead**; the system does no useful work while switching
  - The more **complex** the OS and the PCB, the longer the context switch
- Time dependent on hardware support
  - Some hardware provides **multiple sets of registers** per CPU, so **multiple contexts loaded** at once

What is the difference between Interrupt handling and context switch?



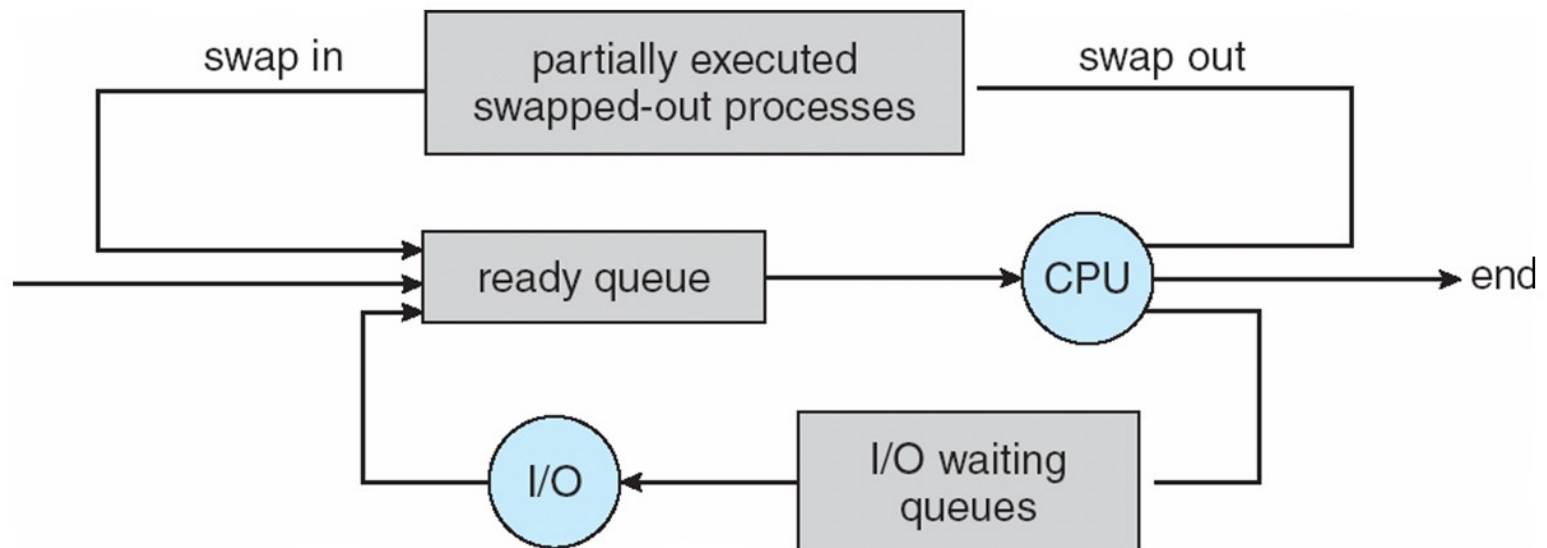


# Schedulers

- **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**) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which **processes** should be brought into the **ready queue**
  - Long-term scheduler is invoked **infrequently** (**seconds, minutes**) ⇒ (may be slow)
  - 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**
- **Load balancing** of I/O and CPU bound processes
  - Long-term scheduler **strives for good *process mix***

# 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**





# Process Creation

- **Parent** process create **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**

ps -el

pstree

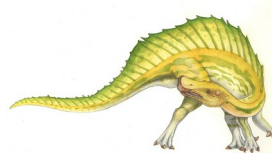
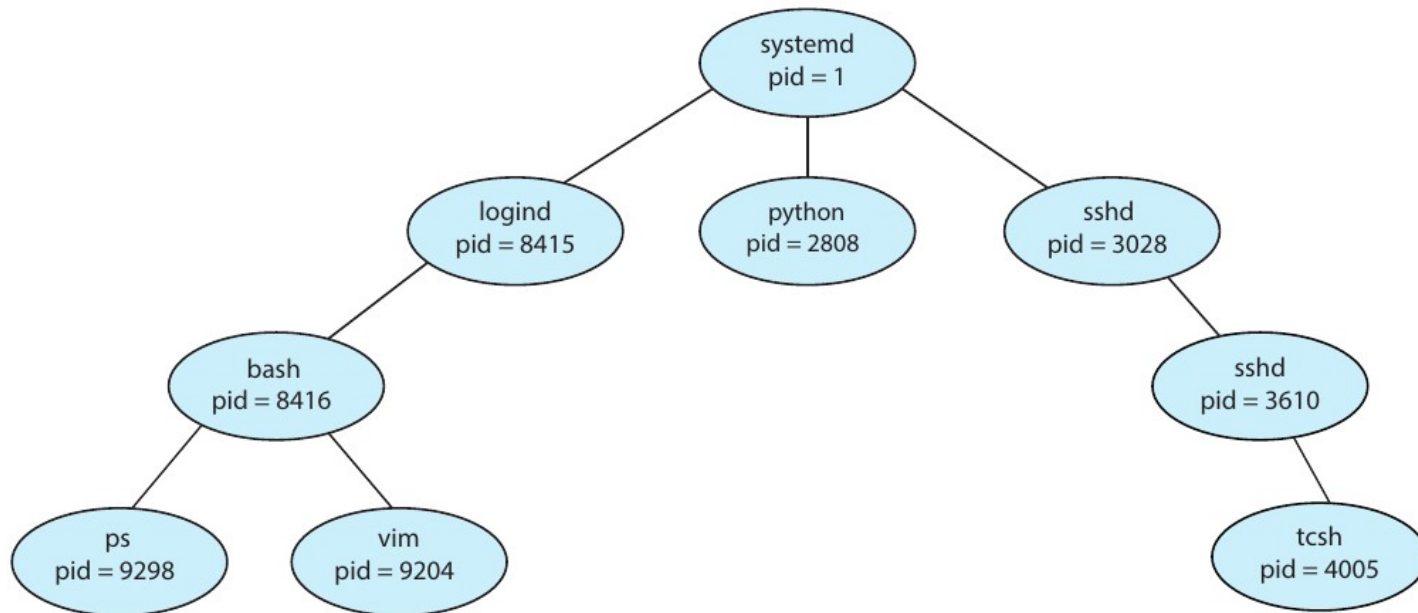
Prevent overloading the system by too many child processes





# A Tree of Processes in Linux

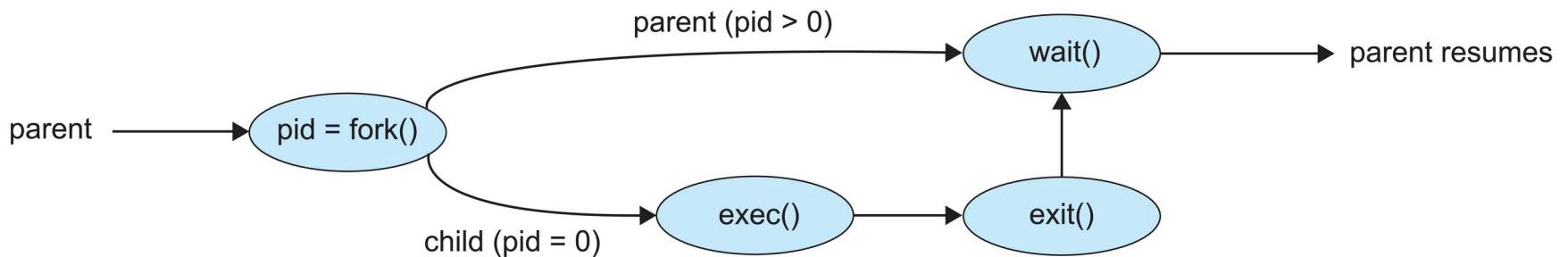
pstree





# Process Creation (Cont.)

- 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
  - Parent process calls **wait()** waiting for the child to terminate





# C Program 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;
}
```





# Practice

---

What is the output of following code? How many processes are created?

```
Main(){
    pid_t pid1, pid2, pid3;
    pid1 = fork();
    wait()
    pid2 = fork();
    if (pid1 == 0 or pid2==0){
        printf('new child process');
    }
    pid3 = fork();
    printf("End of the process");
}
```





# 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 `kill()` 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 operating systems do not allow 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);
```
- A process that has terminated, but whose parent has not yet called `wait()`, is known as a **zombie process**
- If a parent did not invoke `wait()` and instead terminated, thereby leaving its child processes as **orphans**





# Process Termination

- Some operating systems do not allow 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.

All processes transition to zombie state when they terminate, but generally they exist as zombies only briefly.

Once the parent calls `wait()`, the process identifier of the zombie process and its entry in the process table are released.

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

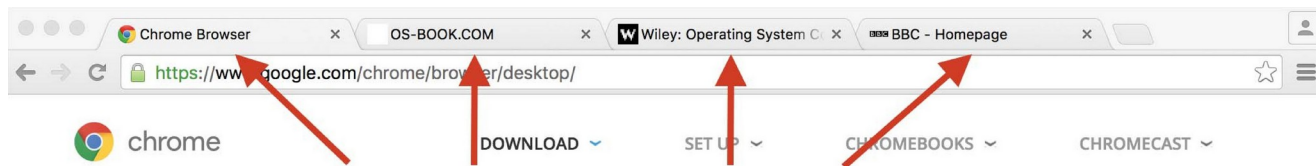
- A process that has terminated, but whose parent has not yet called `wait()`, is known as a **zombie process**
- if a parent did not invoke `wait()` and instead terminated, thereby leaving its child processes as **orphans**





# Multiprocess Architecture – Chrome Browser

- Many web browsers run as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 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



Each tab represents a separate process.





# 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 (for small amounts of data)**

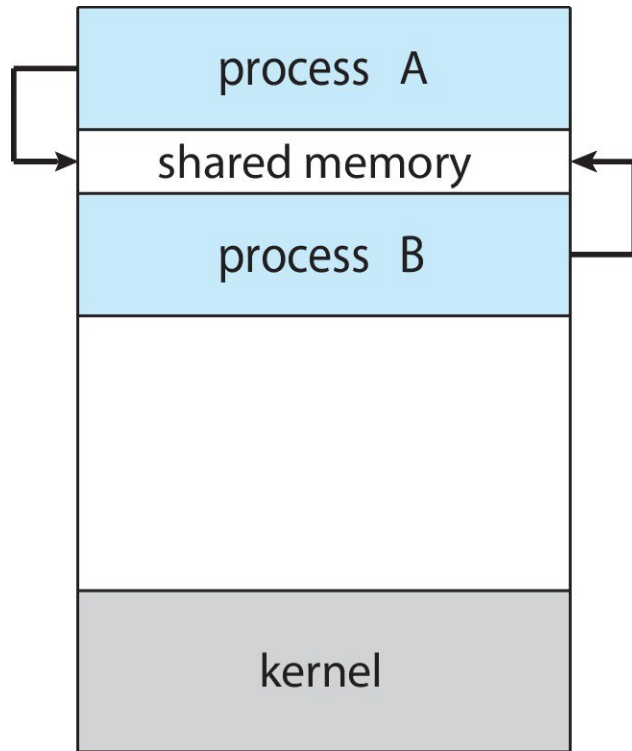




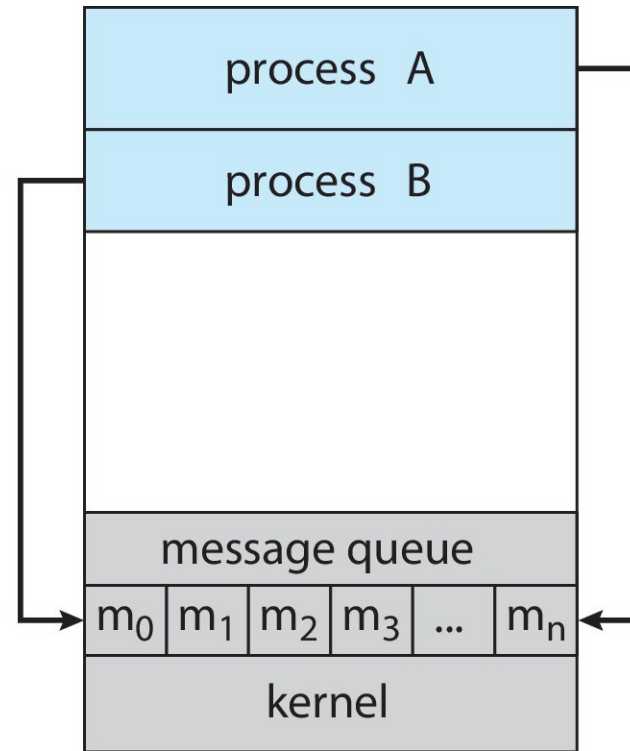
# Communications Models

(a) Shared memory.

(b) Message passing.



(a)



(b)





# Producer-Consumer Problem

---

- Paradigm for cooperating processes:
  - *producer* process produces information that is consumed by a *consumer* process
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - ▶ Producer never waits
    - ▶ Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - ▶ Producer must wait if all buffers are full
    - ▶ Consumer waits if there is no buffer to consume





# Interprocess Communication – Shared Memory

---

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- **Benefits:**
  - Faster than message passing
- **Disadvantages:**
  - Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.





# 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







# Producer Process – Shared Memory

---

```
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;
}
```





# Consumer Process – Shared Memory

---

```
item next_consumed;

while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```





# What about Filling all the Buffers?

---

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
- We can do so by having an integer **counter** that keeps track of the number of full buffers.
- Initially, **counter** is set to 0.
- The integer **counter** is incremented by the producer after it produces a new buffer.
- The integer **counter** is decremented by the consumer after it consumes a buffer.





# Producer

---

```
while (true) {  
    /* produce an item in next produced */  
  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```





# Consumer

---

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in next consumed */  
}
```





# Race Condition

- **counter++** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

S0: producer execute	<b>register1 = counter</b>	{register1 = 5}
S1: producer execute	<b>register1 = register1 + 1</b>	{register1 = 6}
S2: consumer execute	<b>register2 = counter</b>	{register2 = 5}
S3: consumer execute	<b>register2 = register2 - 1</b>	{register2 = 4}
S4: producer execute	<b>counter = register1</b>	{counter = 6}
S5: consumer execute	<b>counter = register2</b>	{counter = 4}





# Race Condition (Cont.)

---

- Question – why was there no race condition in the first solution (where at most  $N - 1$  buffers can be filled?)
- More in Chapter 6.



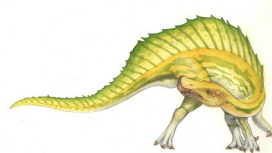


# Examples of IPC Systems - POSIX

---

## ■ POSIX Shared Memory

- Process first creates shared memory segment  
`shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`
- Also used to open an existing segment
- Set the size of the object  
`ftruncate(shm_fd, 4096);`
- Use `mmap()` to memory-map a file pointer to the shared memory object
- Reading and writing to shared memory is done by using the pointer returned by `mmap()`.







# IPC POSIX Producer

```
#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 object */
    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

```
#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 object */
    void *ptr;

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

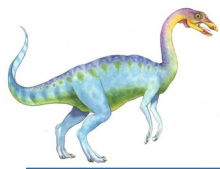
    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

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

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

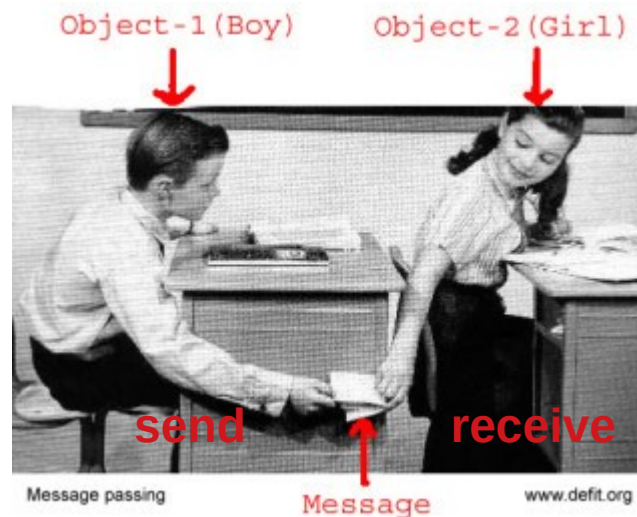
    return 0;
}
```





# IPC – Message Passing

- Message system – processes communicate with each other without sharing the same address space directly
- IPC facility provides two operations:
  - **send(message)**
  - **receive(message)**
- The *message* size is either fixed or variable





# Message Passing

---

- If processes  $P$  and  $Q$  wish to communicate, they need to:
  - Establish a **communication link** between them
  - Exchange messages via send/receive
- 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 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?





# Message Passing model

---

## ■ Physical link

- Shared memory, Hardware bus, Network

## ■ Logical link

- Naming
  - Direct (name the process), indirect (name the mailbox or link)
- Synchronization
  - Block, Non-block
- Buffering
  - Zero, bounded, un-bounded





# Producer-Consumer: Message Passing

- Producer

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

- Consumer

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





# Pipes

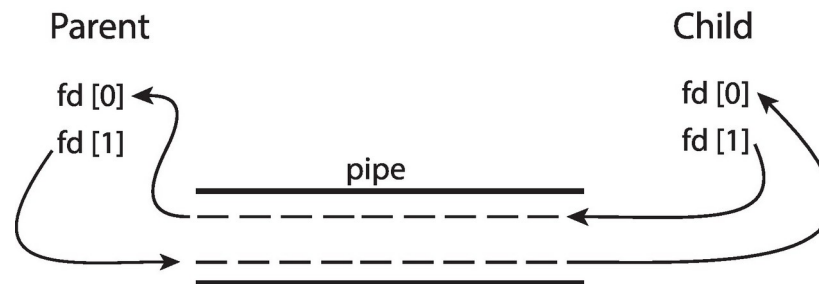
- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., **parent-child**) between the communicating processes?
  - Can the pipes be used over a network?
- **Ordinary pipes** – cannot be accessed from outside the 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 (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**







# Named Pipes

---

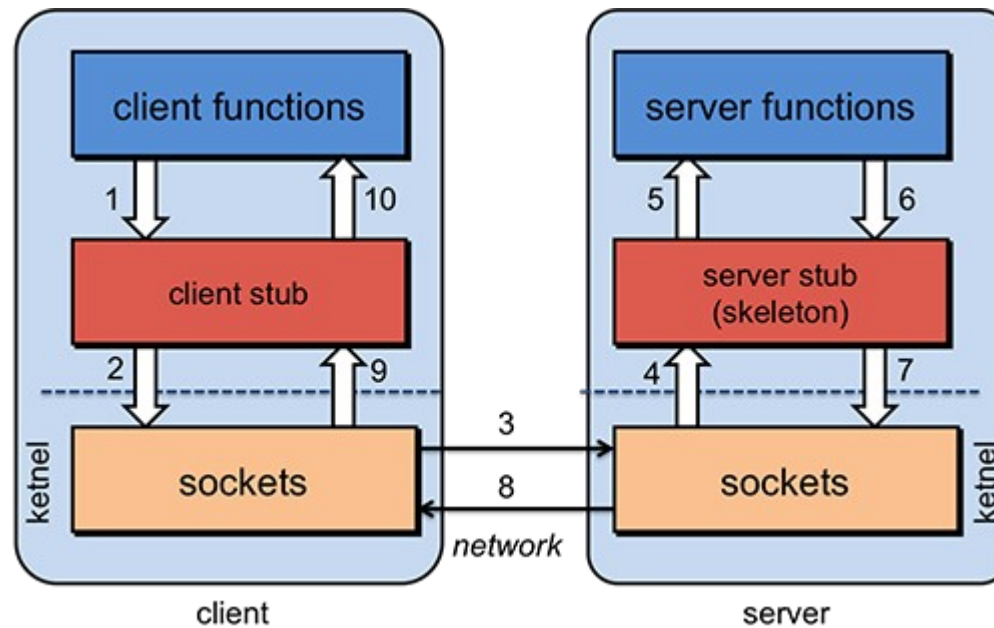
- Named Pipes are more powerful than ordinary pipes
- Communication is **bidirectional**
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems





# Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls (RPC)





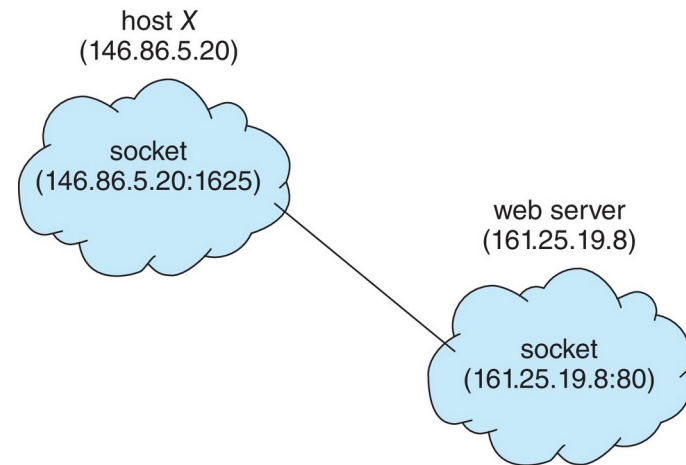
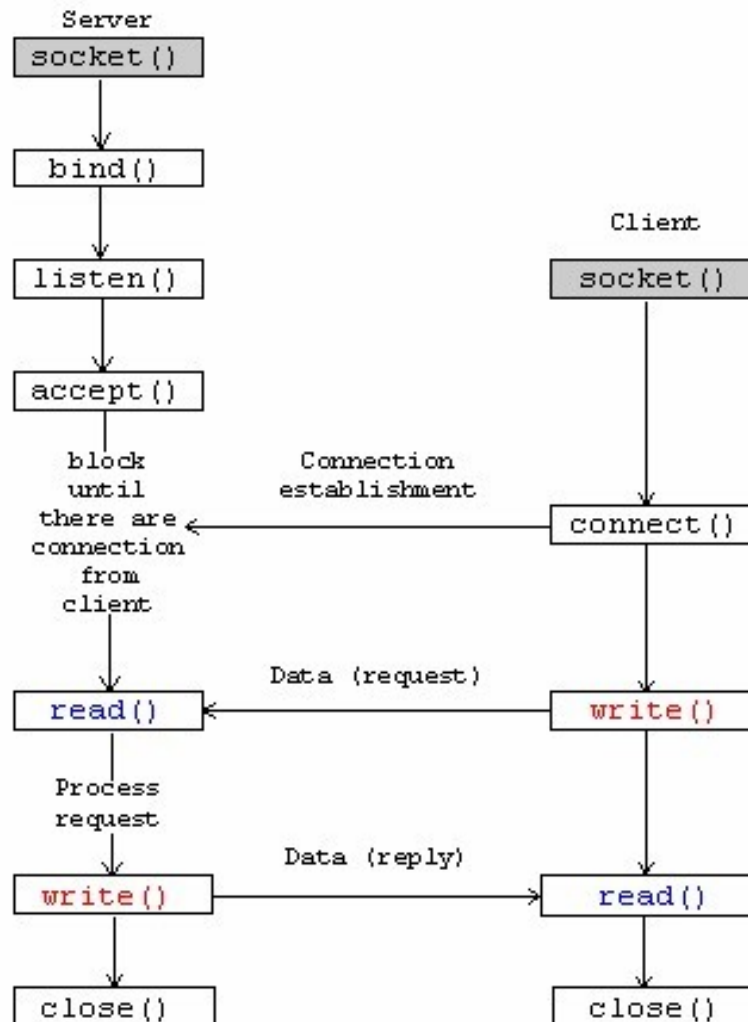
# Sockets

- 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
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running
- Three types of sockets
  - **Connection-oriented (TCP)**
  - **Connectionless (UDP)**
  - **MulticastSocket** class– data can be sent to multiple recipients
- Consider this “Date” server in Java:





# Socket Communication

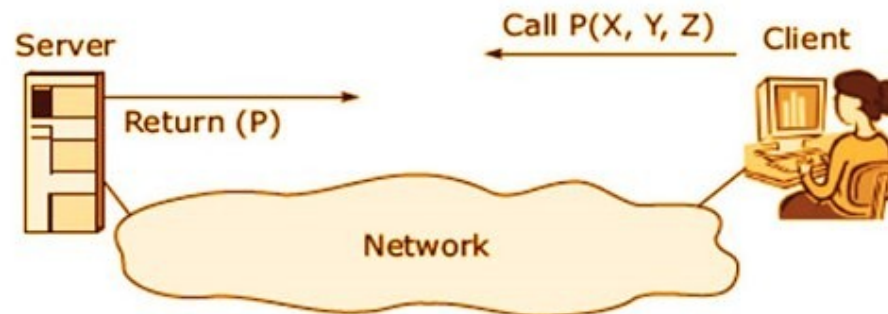




# Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for 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 performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language (MIDL)**

## Remote Procedure Call (RPC)





# Remote Procedure Calls (Cont.)

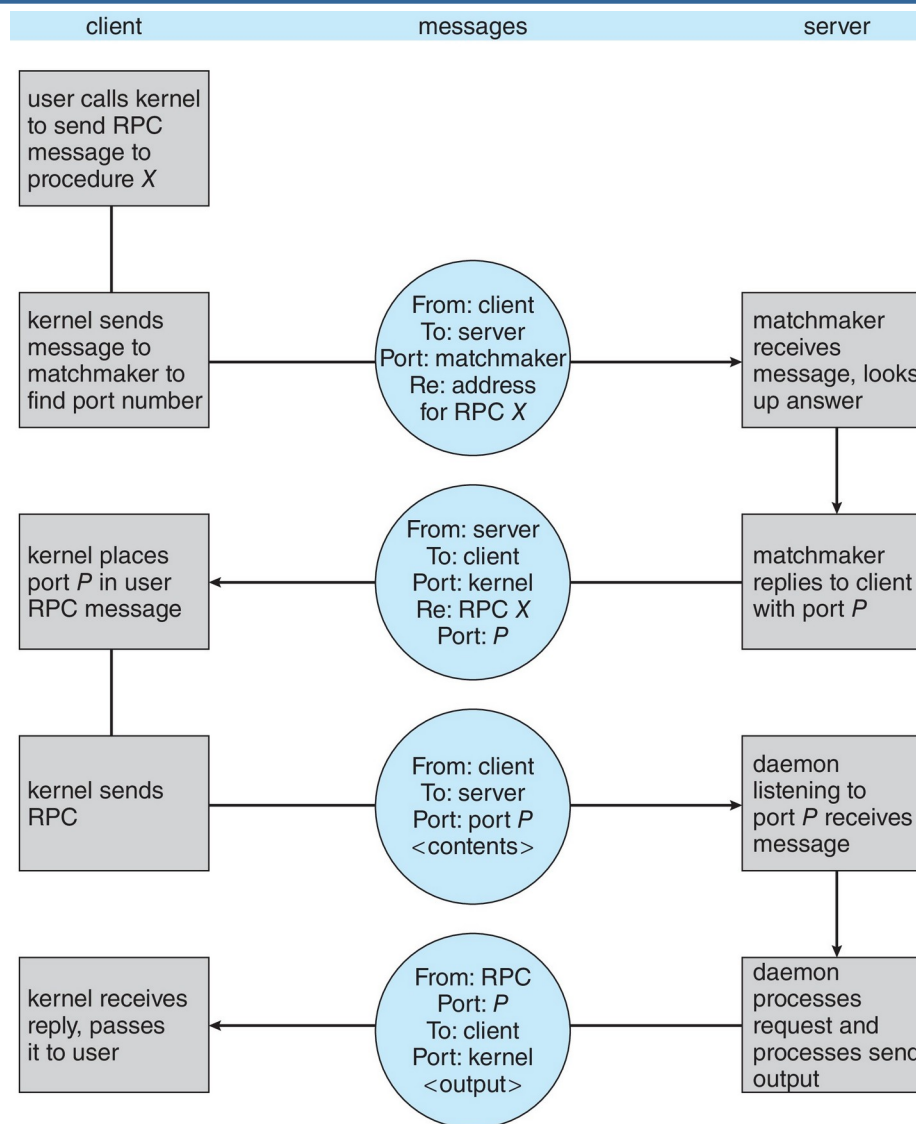
---

- 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





# Examples of IPC Applications

---

- X Server and client
  - Unix-domain sockets, named pipes, ...
- Piping commands in shell
  - pipe
- Database server
  - Socket, RPC
- Web services
  - Kind of RPC
- ...

