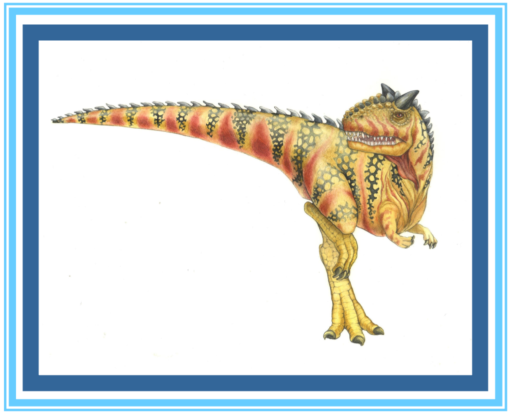
Chapter 3: Processes



Chapter 3: Processes

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

AGENDA

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that uses pipes and POSIX shared memory to perform interprocess communication.
- Describe client-server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.

Recap

- OS structures (Simple, monolithic, multikernel, layered, microkernel,)
- UNIX/LINUX environment, file/directory structures, important directories (bin, dev, usr, lib, root, ~, ., .., $HOME)
- Important commands (ls, pwd, cd, cp, mkdir, mv, rm, gcc)
- Absolute versus relative path
- Process concept (text/code, PC/CPU state, stack, data section, heap, PCB, environment)
- PCB (PID, PPID, Process status, CPU state, Memory status, accounting info, CPU scheduling, I/O status)
- Process status (new, ready, wait, run, exit)
- CPU Bound, I/O Bound processes
- Process queues (Jobs, Ready queue, waiting queue)
- Context switching (dispatcher)
- Process Scheduling (STS (CPU); LTM (Job Scheduler); MTS (swapper))

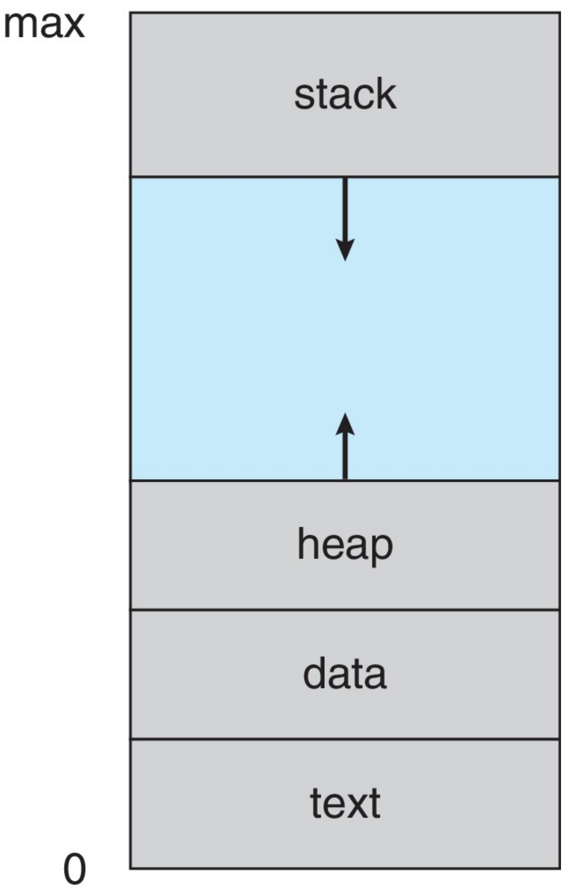
Process Concept

- 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
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **Program Counter**, processor registers, **CPU State**
  - **Stack** containing temporary data
    - 4 Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time
  - Process control block (**PCB**)
  - Environment (**Kernel DS**)

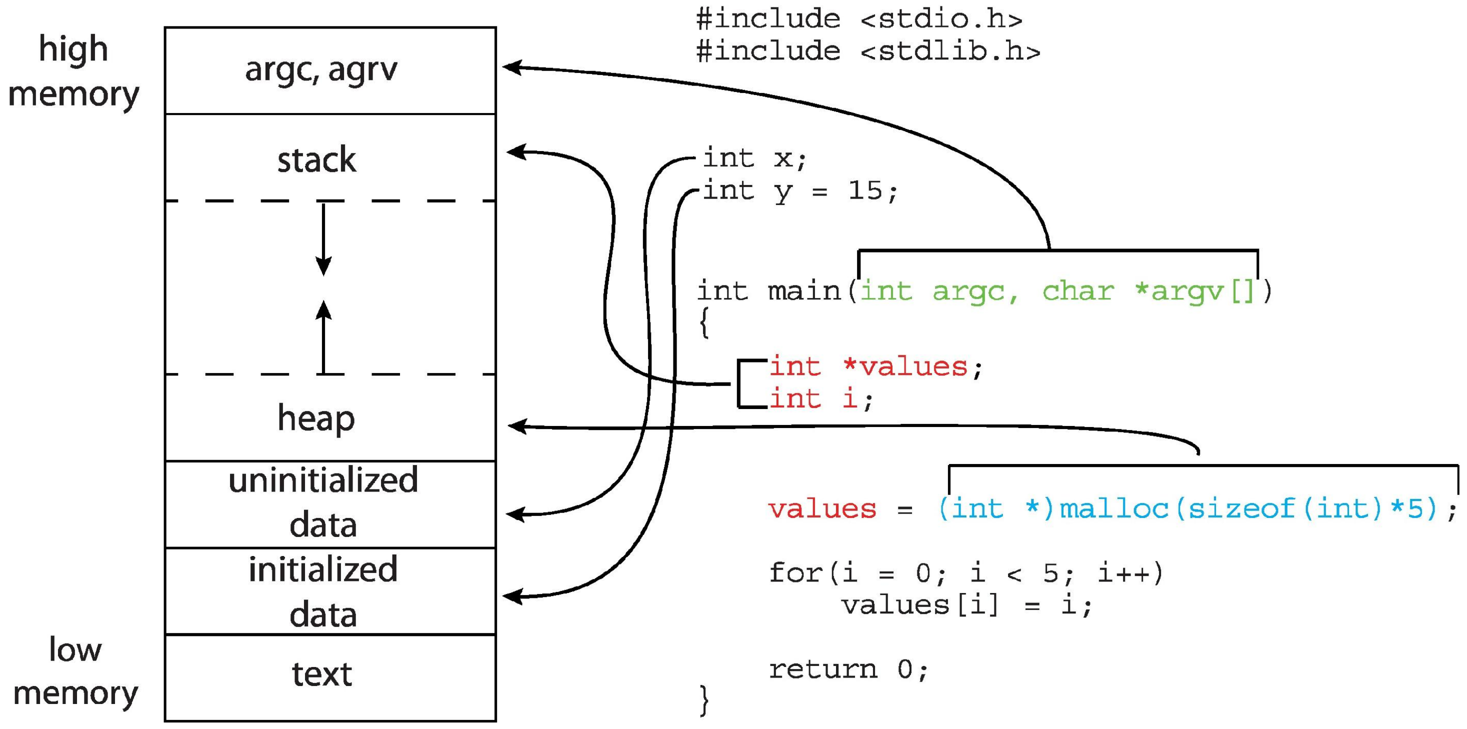
Process Concept (Cont.)

- 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
    - Compiler
    - Text editor

Process in Memory



Memory Layout of a C Program



high memory

argc, agrv

stack

heap

uninitialized data

initialized data

text

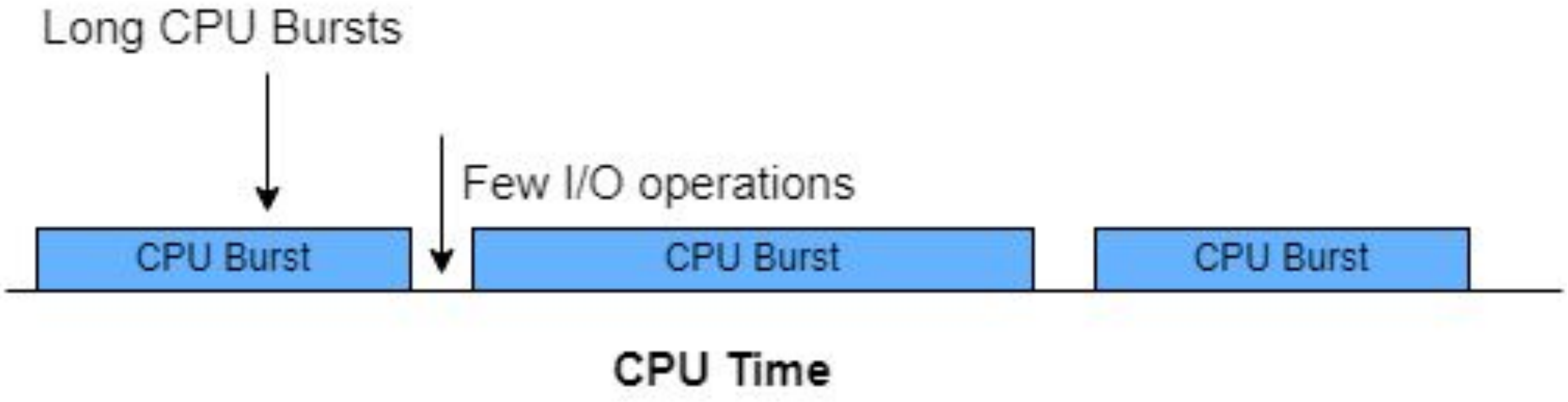
low memory

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

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

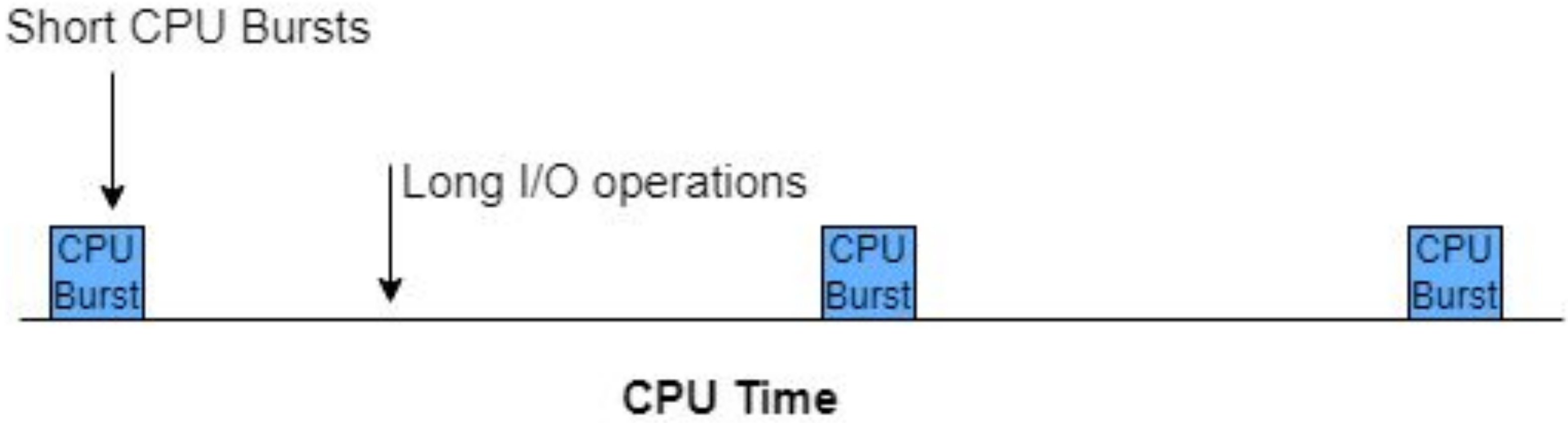
CPU and I/O Bound Processes

Processes can be described as either:

CPU-bound process (Spends more time in computations)



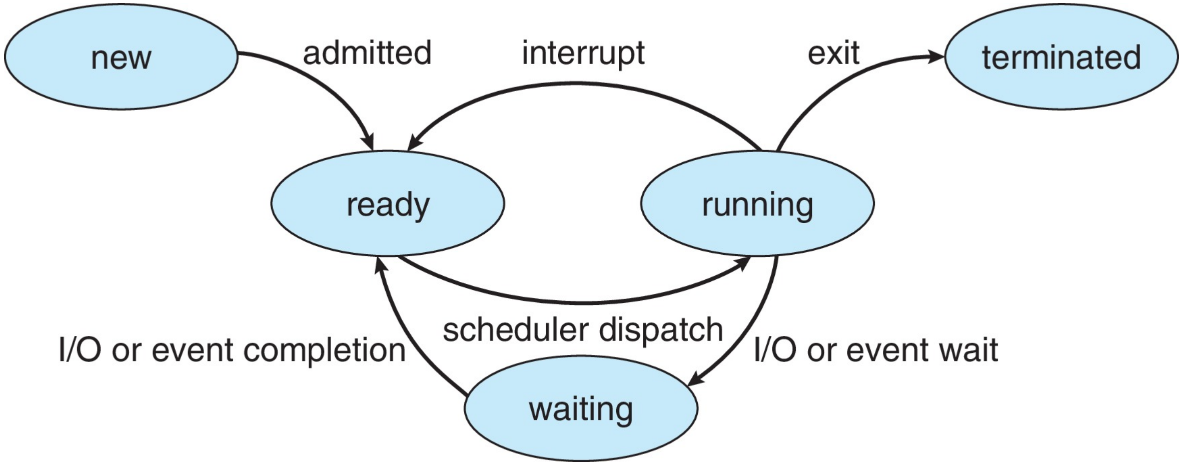
I/O-bound process (Spends more time in I/O operations)



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

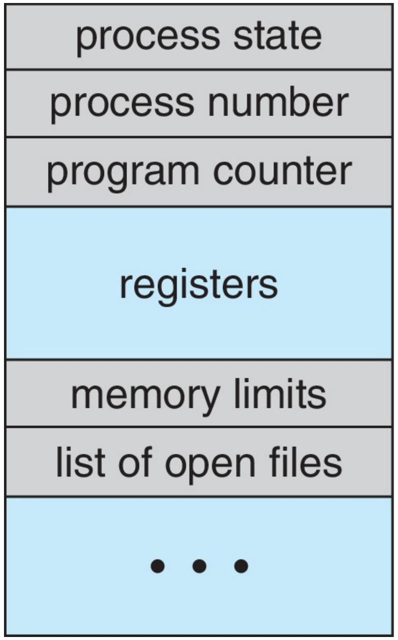
Diagram of Process State



Process Control Block (PCB)

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

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



process state

process number

program counter

registers

memory limits

list of open files

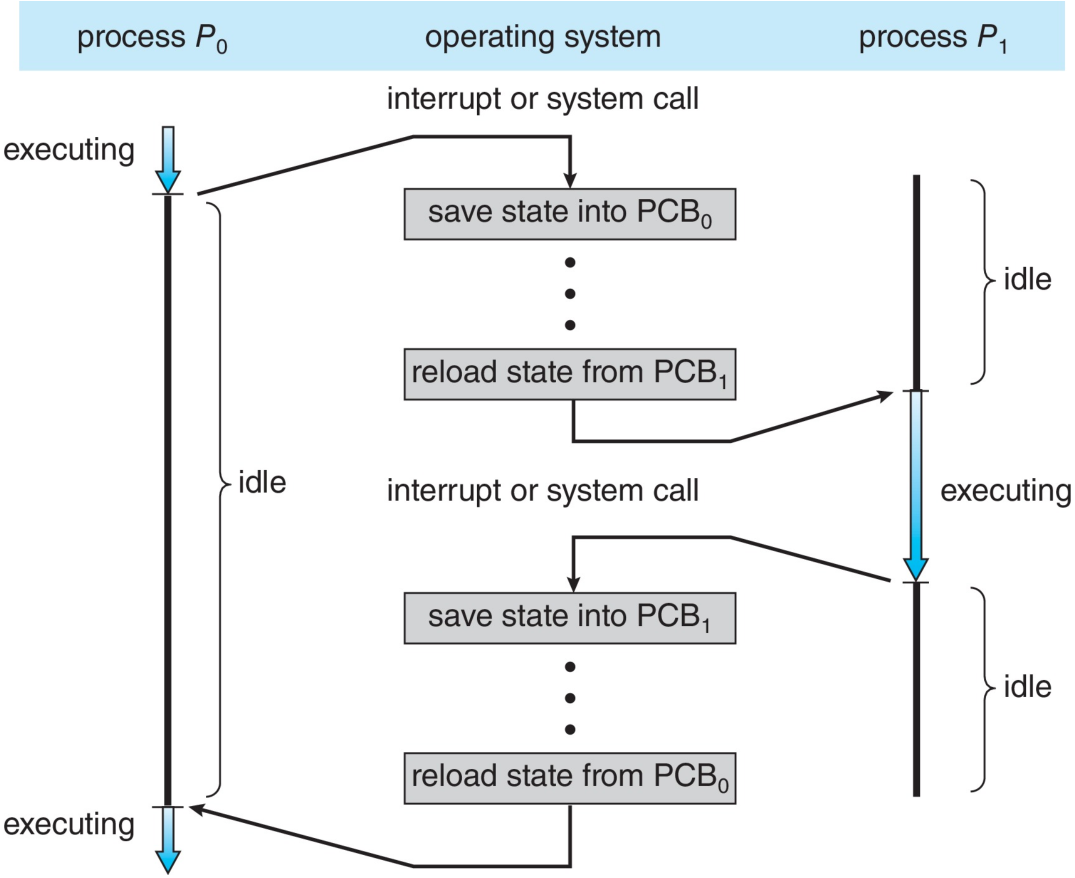
• • •

Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - 4 Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- Explore in detail in Chapter 4

CPU Switch From Process to Process

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



process P_0

operating system

process P_1

interrupt or system call

executing

save state into PCB_0

idle

reload state from PCB_1

idle

interrupt or system call

executing

save state into PCB_1

idle

reload state from PCB_0

executing

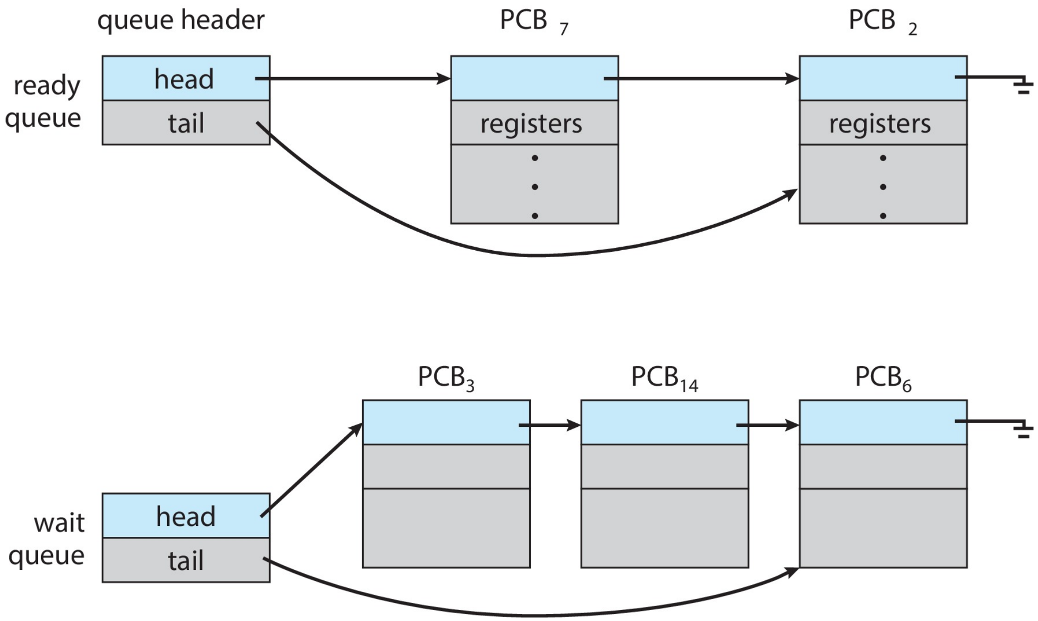
Context Switch

- 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 (Dispatcher)**
- **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 □ multiple contexts loaded at once

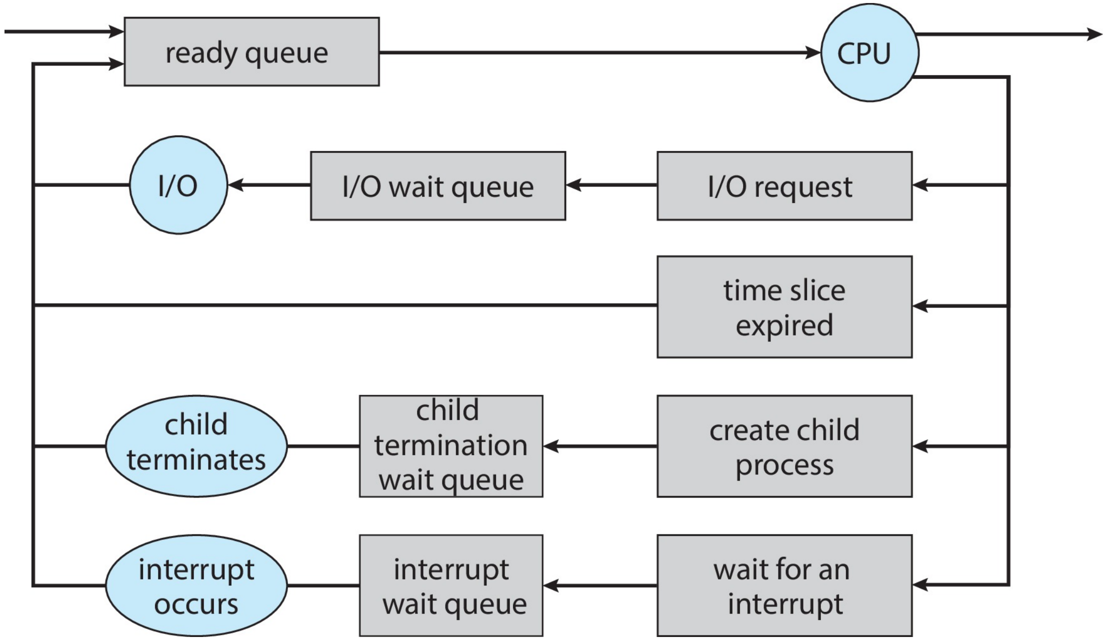
Process Scheduling

- **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
    - **Job queue** – set of all processes in the system
    - **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
        - **Reasons:** Memory requirements of the process exceeds
        - Signal from OS, wait for I/O, wait for event

Ready and Wait Queues



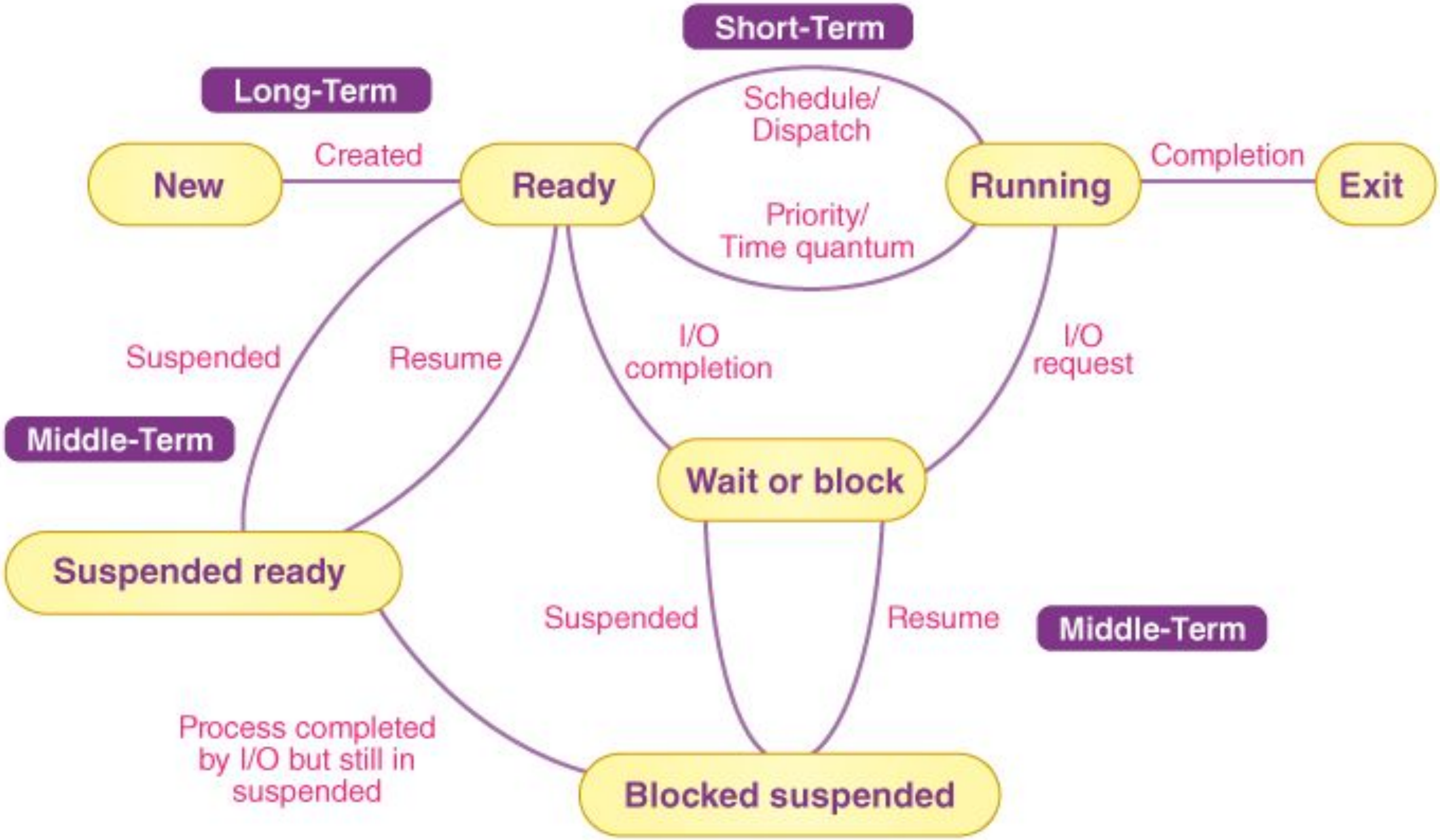
Representation of Process Scheduling



Schedulers

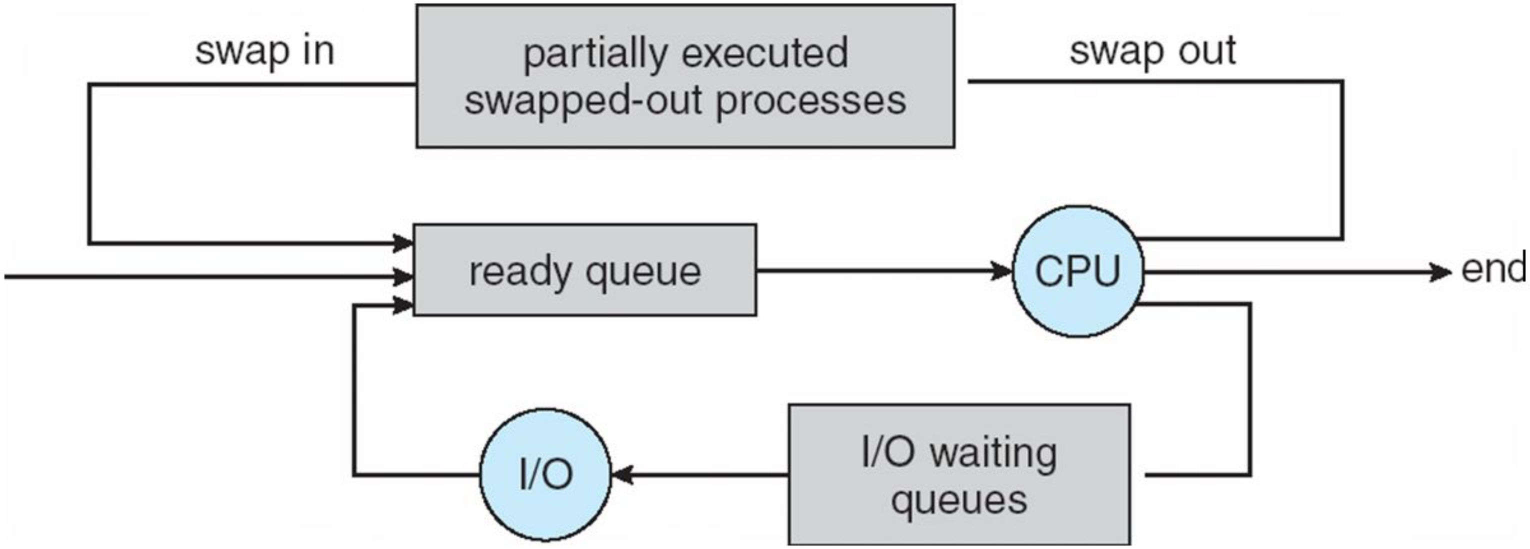
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Invoked frequently ⇒ (must be fast)
  - Invokes context switching- Highly time crucial
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Invoked infrequently ⇒ (may be slow)
  - Controls the **degree of multiprogramming**
  - Long-term scheduler strives for good **process mix**
- **Medium Term Schedulers** (or **Swapping scheduler**)
  - Process from main memory to disk temporarily (swap space)
  - Medium speed for swap in and swap out

Schedulers


Short-Term
Long-Term
Schedule/ Dispatch
New
Created
Ready
Running
Completion
Exit
Priority/ Time quantum
Suspended
Resume
I/O completion
I/O request
Middle-Term
Wait or block
Suspended ready
Suspended
Resume
Middle-Term
Process completed by I/O but still in suspended
Blocked suspended

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**



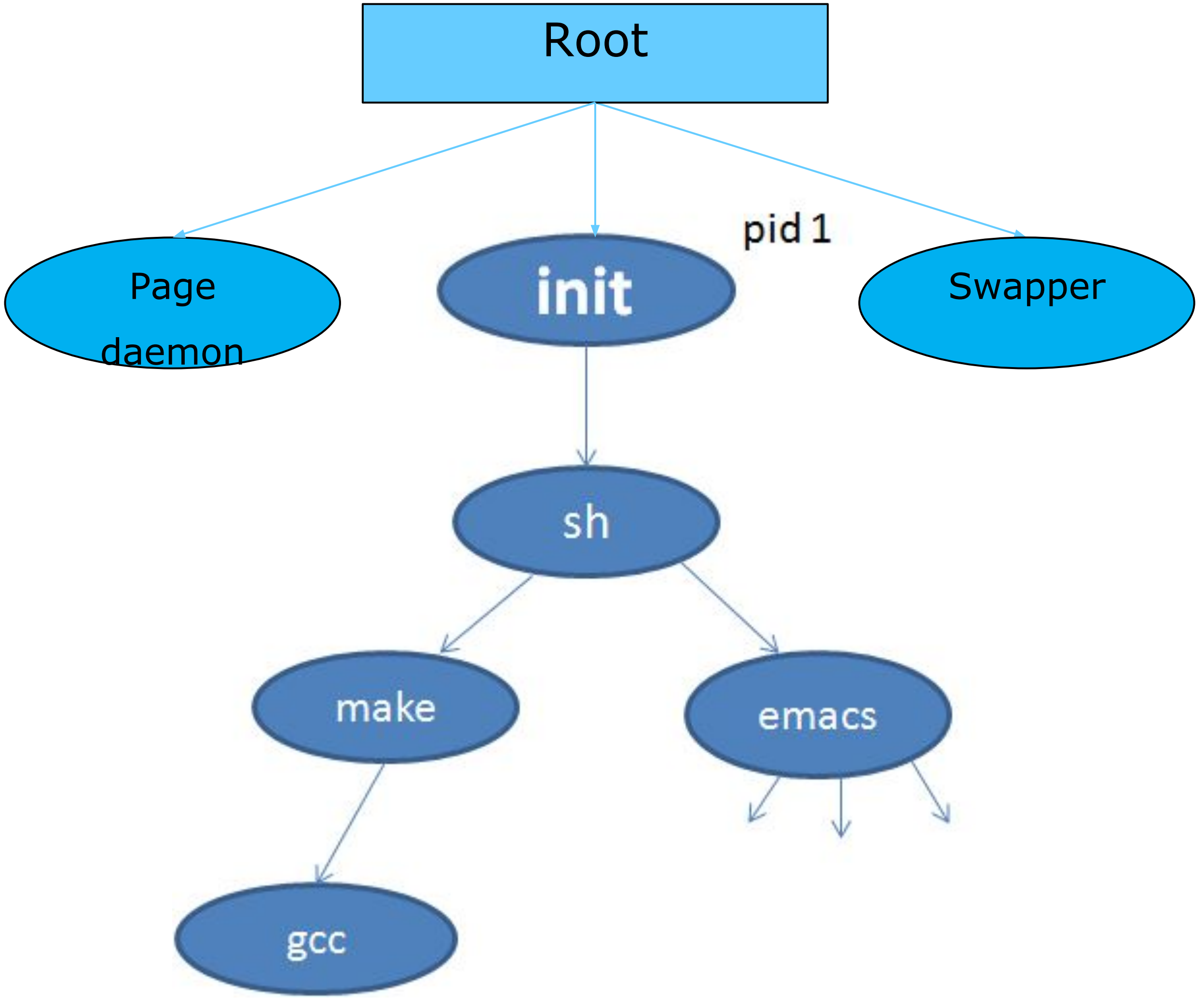
Operations on Processes

- System must provide mechanisms for:
  - Process creation
  - Process termination

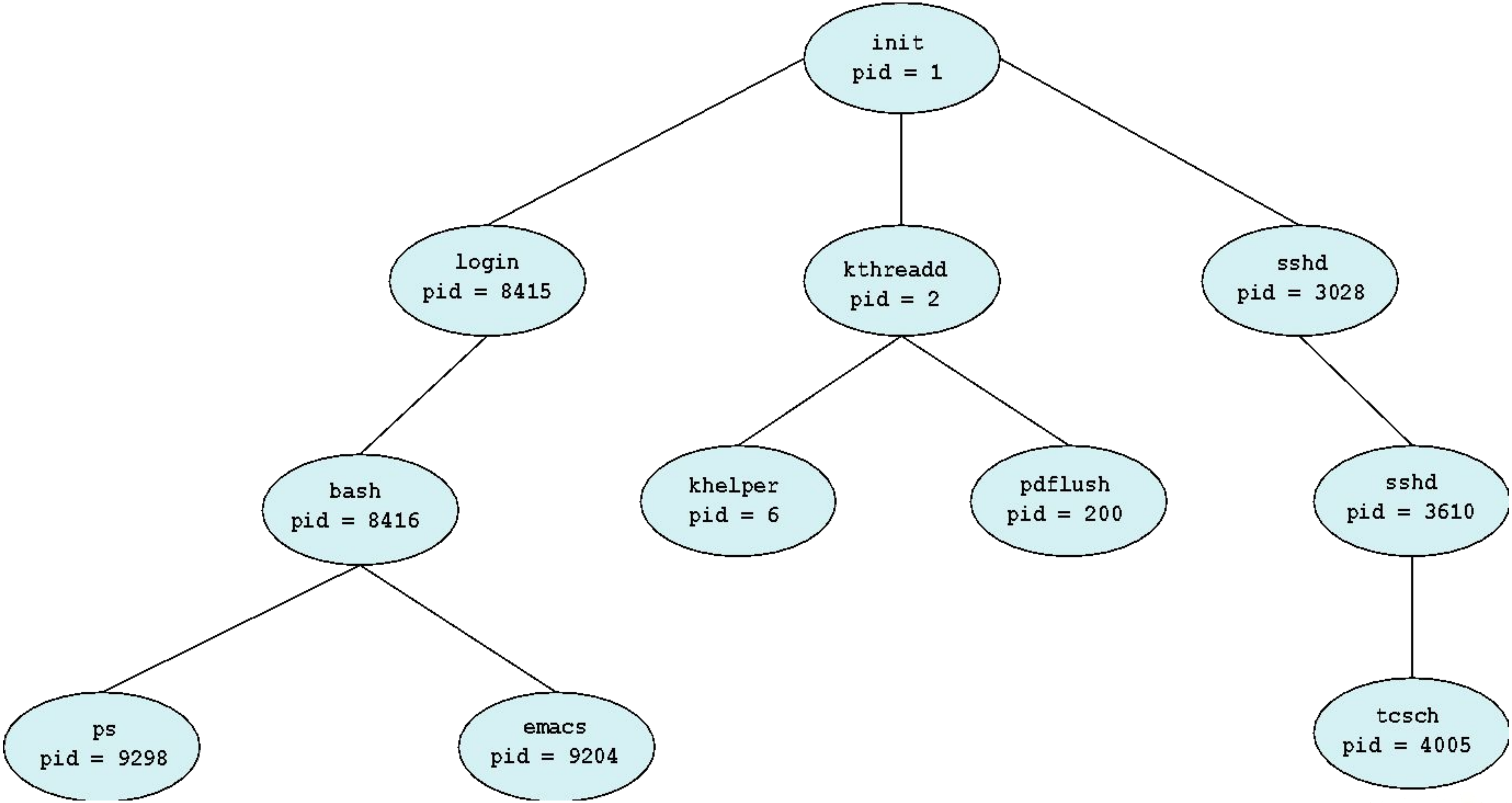
Process Creation

- **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

Process Tree on UNIX

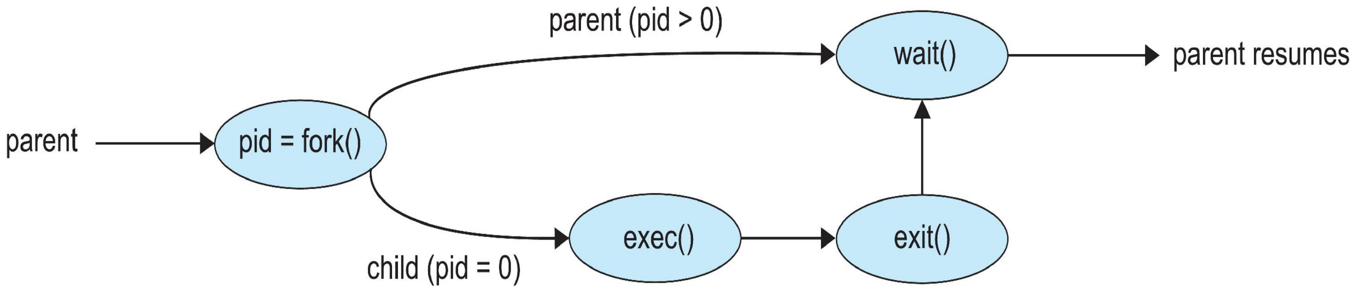


A Tree of Processes in Linux



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



Process creation

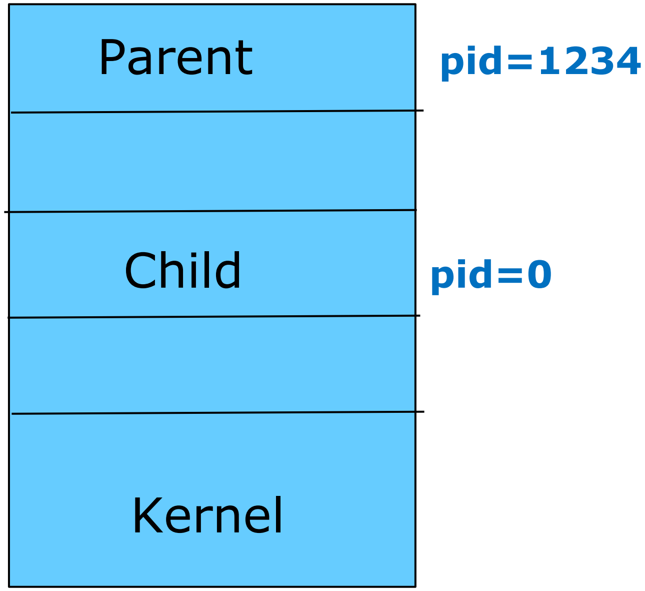
- The return code for fork is **0 for the child process** and the process identifier is returned to the parent process.
  - On success, both the processes continue execution at the instruction after the fork
  - The child process may be assigned a task and the parent can be made to wait for the child
  - **On failure, -1** is returned to the parent process and **errno** (kernel variable) is set to the appropriate value to indicate the reason of failure and no child is created
  - **Errno** is an important information for the parent to understand the **issue/reason** with child creation i.e.
    - 4 Number of child is exceeded from predefined number
    - 4 Child exceeds the resources
    - 4 Systems child process limit is reached
    - 4 Swap space or memory does not has space

Fork () inherits

- Sample:

```
#include <sys/types.h>
#include<unistd.h>
pid_t fork();
```

- The child process inherits the following attributes from the parent.
  - Environment
  - Open file descriptor table
  - Signal handling settings
  - Current working directory
  - Root directory
  - File mode creation mask (unmask)
  - Nice value (priority)
- The child differs from the parent.
  - Different child ID (PID)
  - Different parent ID (PPID)
  - Child has its own copy of parents file descriptors



wait ()

- The wait system call suspends the calling process until one of its immediate children terminates.
  - Wait returns prematurely if a signal is received from the system,
  - If all the child process stopped or terminated prior to the call on wait, return is immediate.
  - If the wait call is successful, the process ID of the terminating child is returned
  - If the parent terminates, all the child processes are assigned a new parent the **init** (granddaddy of all process) process. Thus the children still have a parent to return their status and execution statistics.

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()`**)
  - The process of parent process deallocating a child process resources is called reaping
  - A child whose parent doesn't reap it is known as **zombie process**
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call.
- Some reasons for child termination:
  - 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 (Reasons)

- Some operating systems 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.
  - 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);
```

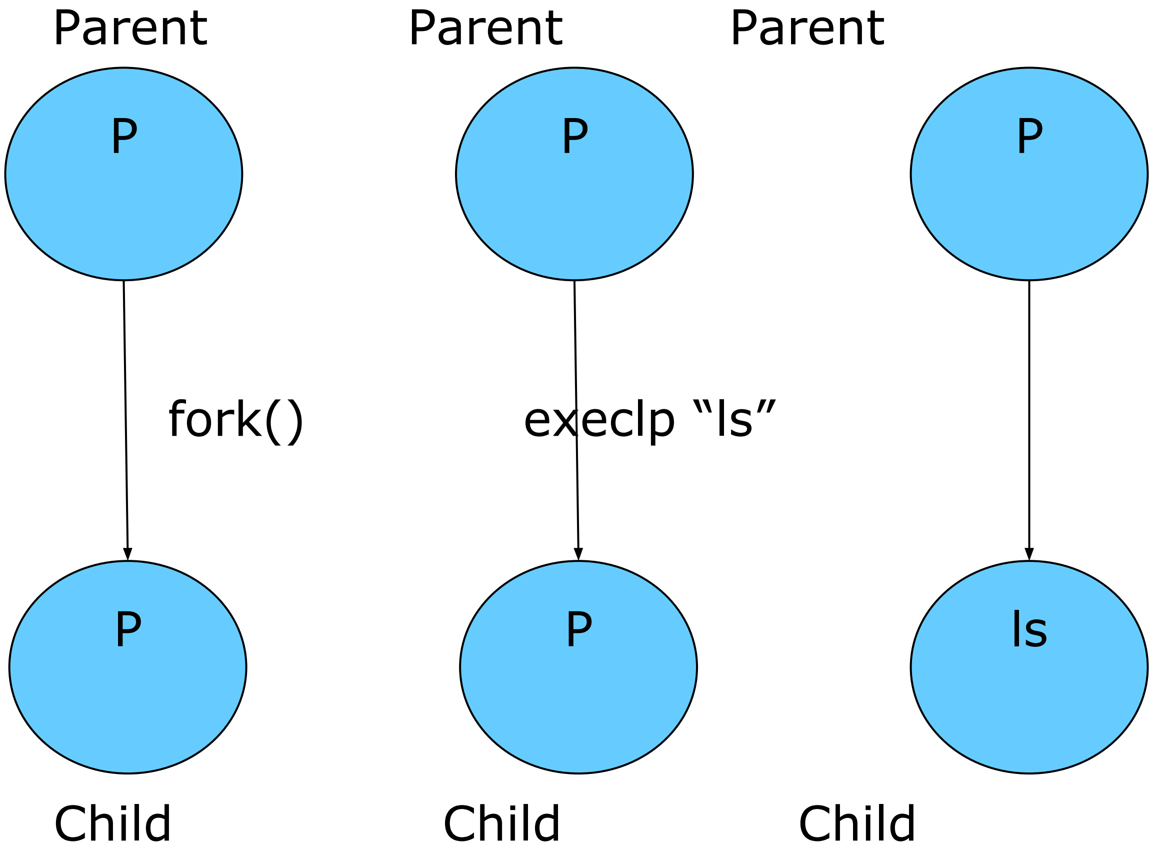
- If no parent waiting (did not invoke `wait()`) process is a **zombie or defunct** a process that has completed execution but still has an entry in the process table as its parent process didn't invoke an wait() system call
- If parent terminated without invoking `wait()`, process is an **orphan i.e. a** computer process whose parent process has finished or terminated, though it (child process) remains running itself.

exec ()

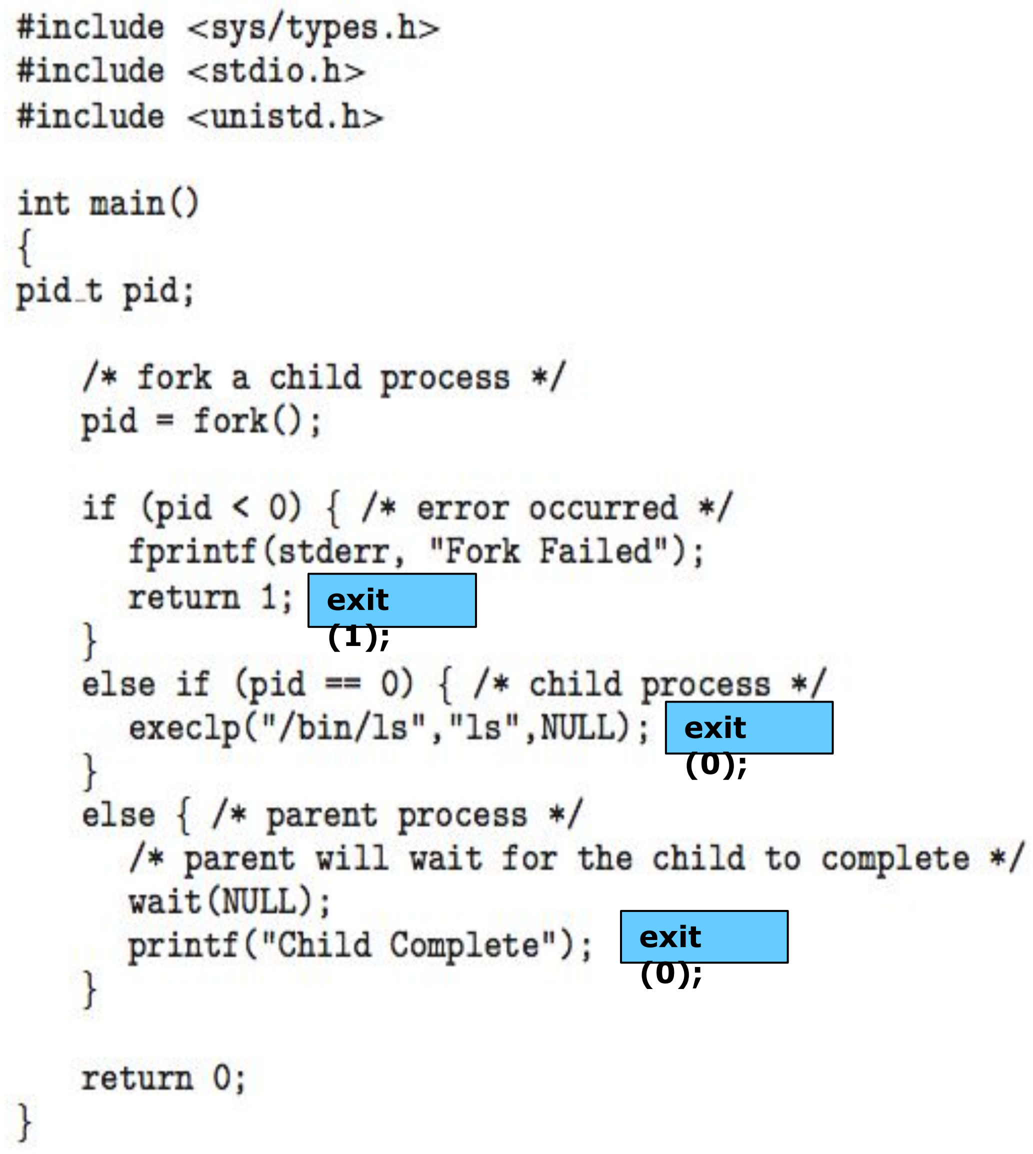
- The exec system call is used after the fork system call by one of the two processes to replace the process memory space with a new executable program.
    - The new process image is constructed from the other executable file (e.g. from /bin folder)
    - There is no return from a successful exec() system call i.e. the process may not go back to its original memory image
    - The calling process image is overlaid by the new process image but the PID remains the same

```
#include <unistd.h>

int execlp(const char *filepath, const char *arg0,….. const char
*argn, (char *), () );


execlp (“/bin/ls”,“ls”, “-l”, NULL)
```

exec ()


Parent
Parent
Parent
P
P
P
fork()
execlp "ls"
P
P
ls
Child
Child
Child

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;
}
```

exit (1);

exit (0);

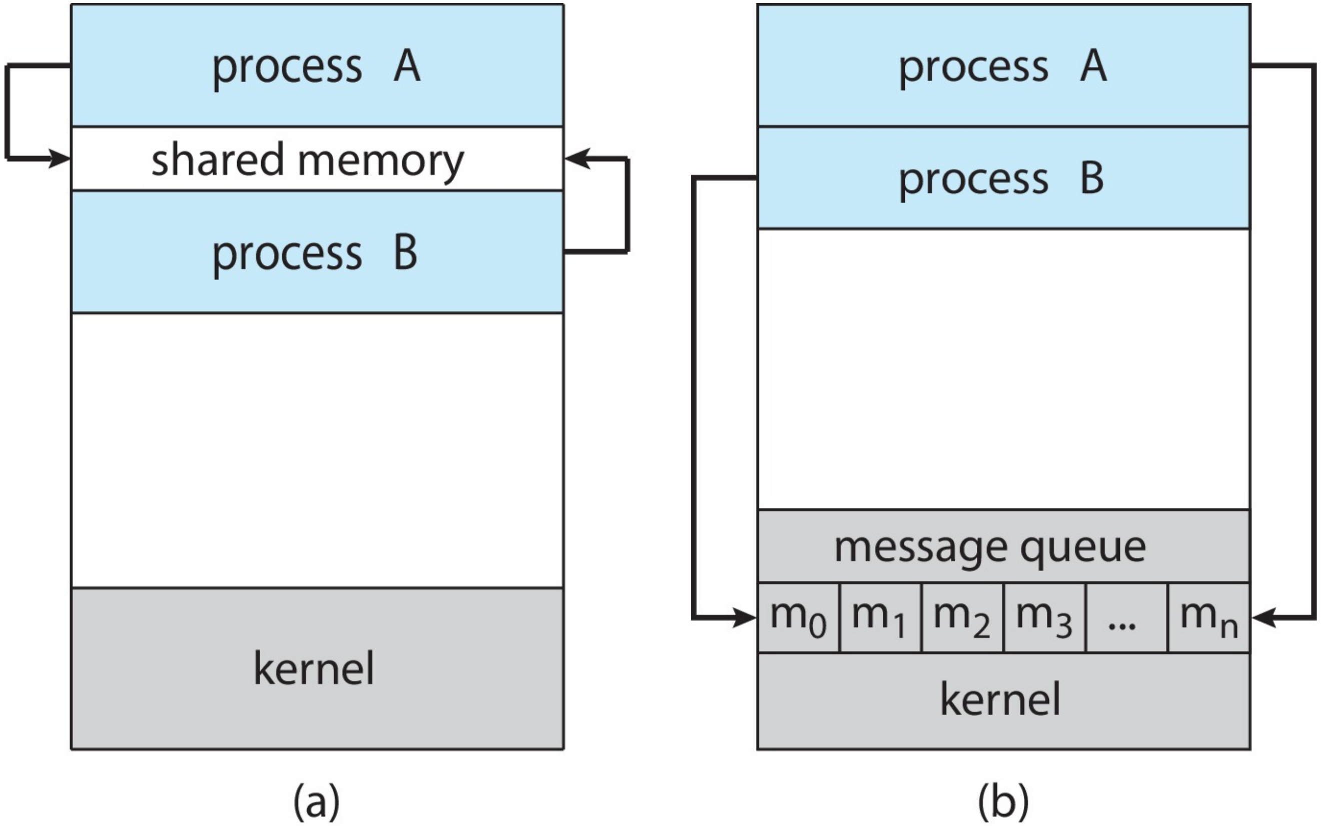
exit (0);

Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- **Independent process** does not affect or get affected by another process
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons/advantages for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory** (under the control of users-PC scenario)
  - **Message passing** (under the control of OS-Pipes)

Communications Models

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



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:
    - 4 Producer never waits
    - 4 Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - 4 Producer must wait if all buffers are full
    - 4 Consumer waits if there is no buffer to consume

Shared Memory Solution

- 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.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapters 6 & 7.

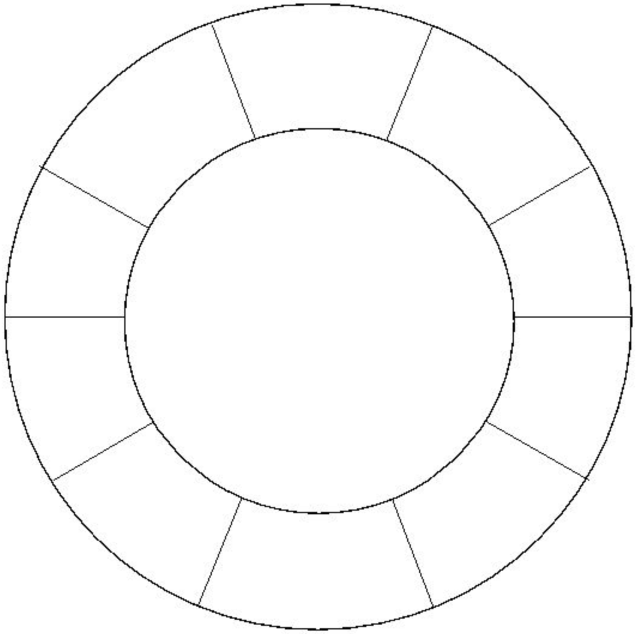
Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution presented in next slides is correct, but can only use **`BUFFER_SIZE-1`** items; that is: 9 items

Bounded-Buffer (Cont.)



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 and 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 `register1 = counter` {register1 = 5}
  S1: producer execute `register1 = register1 + 1` {register1 = 6}
  S2: consumer execute `register2 = counter` {register2 = 5}
  S3: consumer execute `register2 = register2 - 1` {register2 = 4}
  S4: producer execute `counter = register1` {counter = 6 }
  S5: consumer execute `counter = register2` {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.

IPC – Message Passing

- Processes communicate with each other without resorting to shared variables

- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)

- The *message* size is either fixed or variable

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
- 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?

Implementation of Communication Link

- 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, 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
- 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

Indirect Communication (Cont.)

- Operations
  - Create a new mailbox (port)
  - Send and receive messages through mailbox
  - Delete a mailbox
- Primitives are defined as:
  - **`Send`**(*A, message*) – send a message to mailbox A
  - **`receive`**(*A, message*) – receive a message from mailbox A

Synchronization

Message passing may be either blocking or non-blocking

- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - 4 A valid message, or
    - 4 Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

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 */
    }
```

Buffering

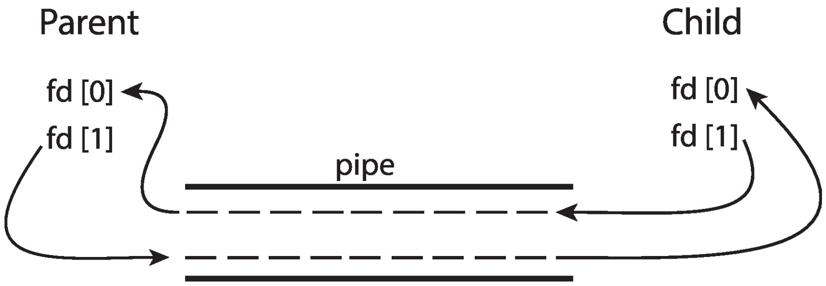
- Queue of messages attached to the link.
- Implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

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

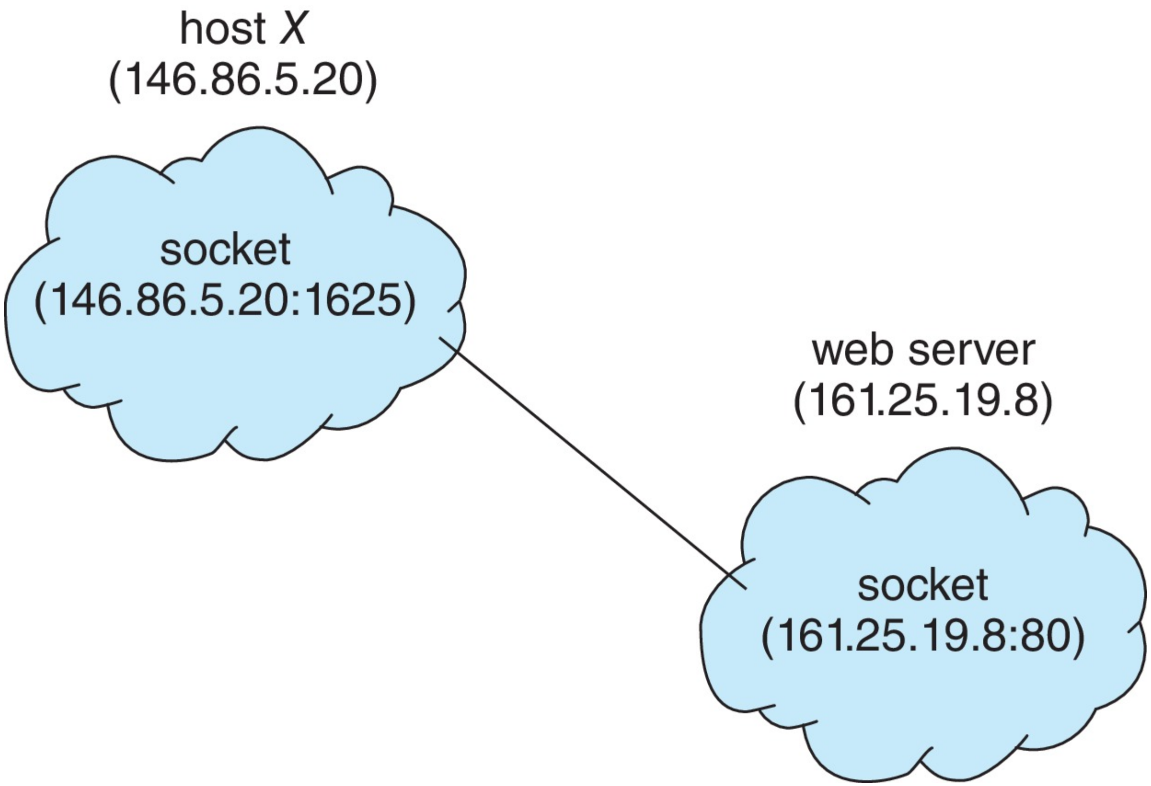
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls

Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port**
  - port is 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

Socket Communication



Sockets in Java

- Three types of sockets
  - **Connection-oriented** (**TCP**)
  - **Connectionless** (**UDP**)
  - **`MulticastSocket`** class– data can be sent to multiple recipients

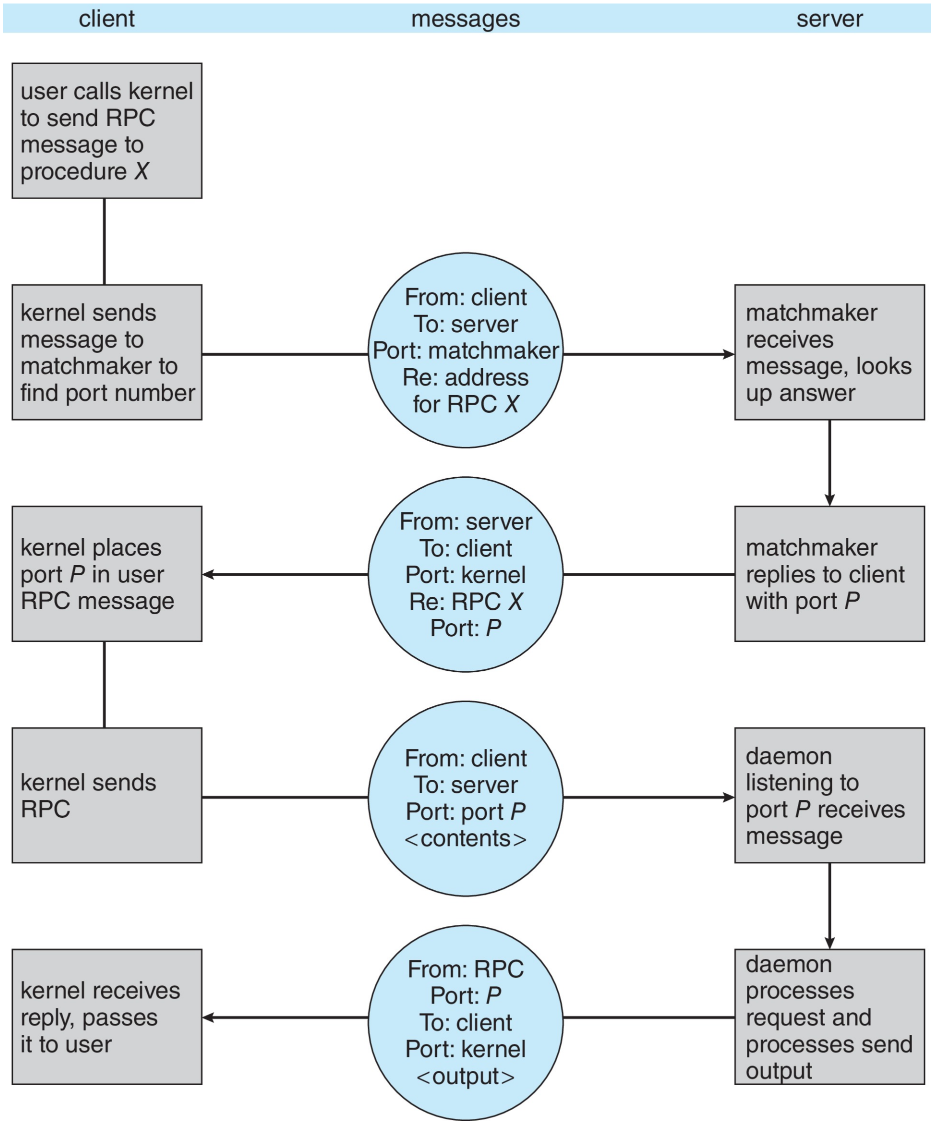
Remote Procedure Calls

- 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 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



client

messages

server

user calls kernel to send RPC message to procedure *X*

kernel sends message to matchmaker to find port number

From: client
To: server
Port: matchmaker
Re: address for RPC *X*

matchmaker receives message, looks up answer

matchmaker replies to client with port *P*

From: server
To: client
Port: kernel
Re: RPC *X*
Port: *P*

kernel places port *P* in user RPC message

kernel sends RPC

From: client
To: server
Port: port *P*
<contents>

daemon listening to port *P* receives message

daemon processes request and processes send output

From: RPC
Port: *P*
To: client
Port: kernel
<output>

kernel receives reply, passes it to user

End of Chapter 3