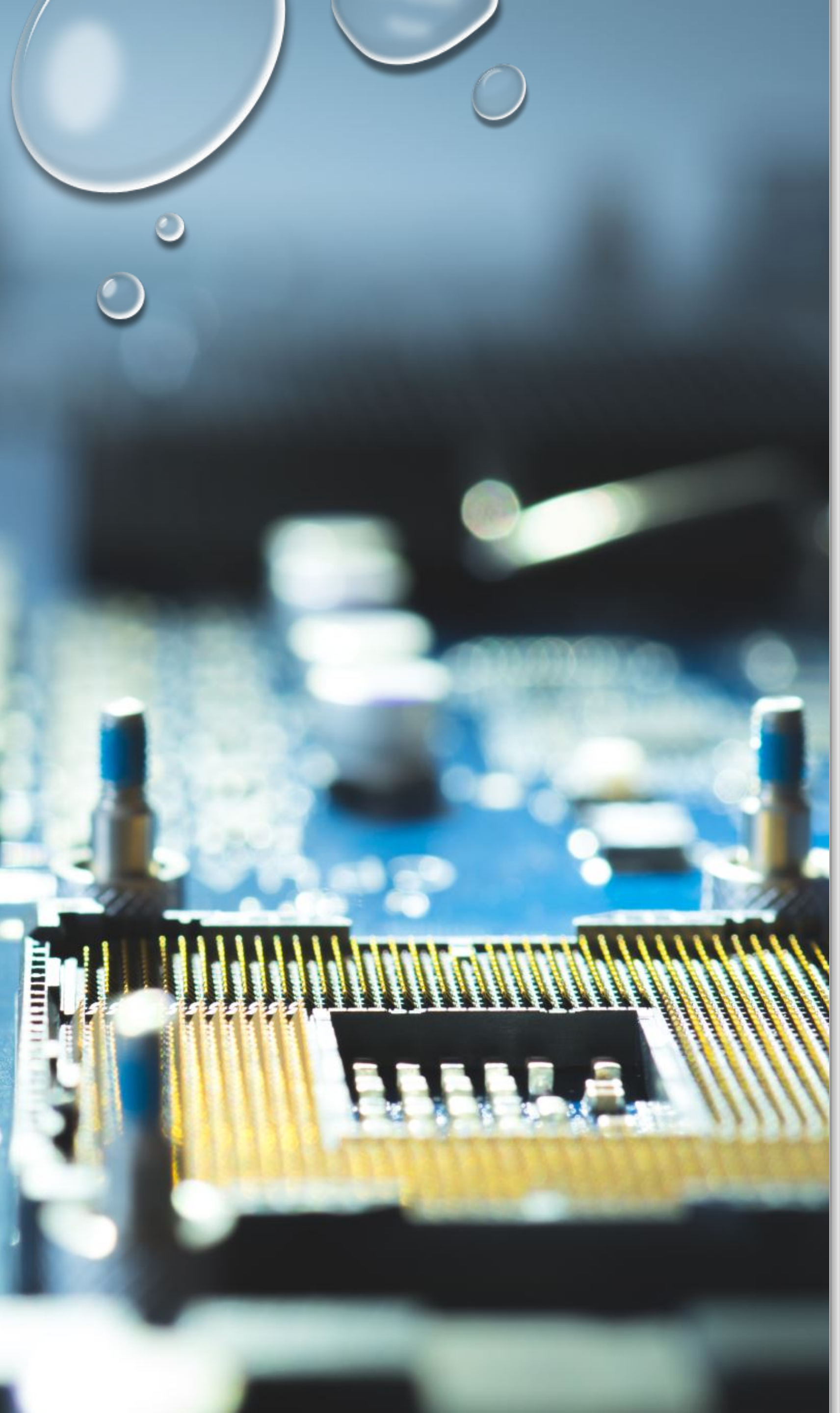
Processes

Operating Systems (CS-2006)
SPRING 2022, FAST NUCES



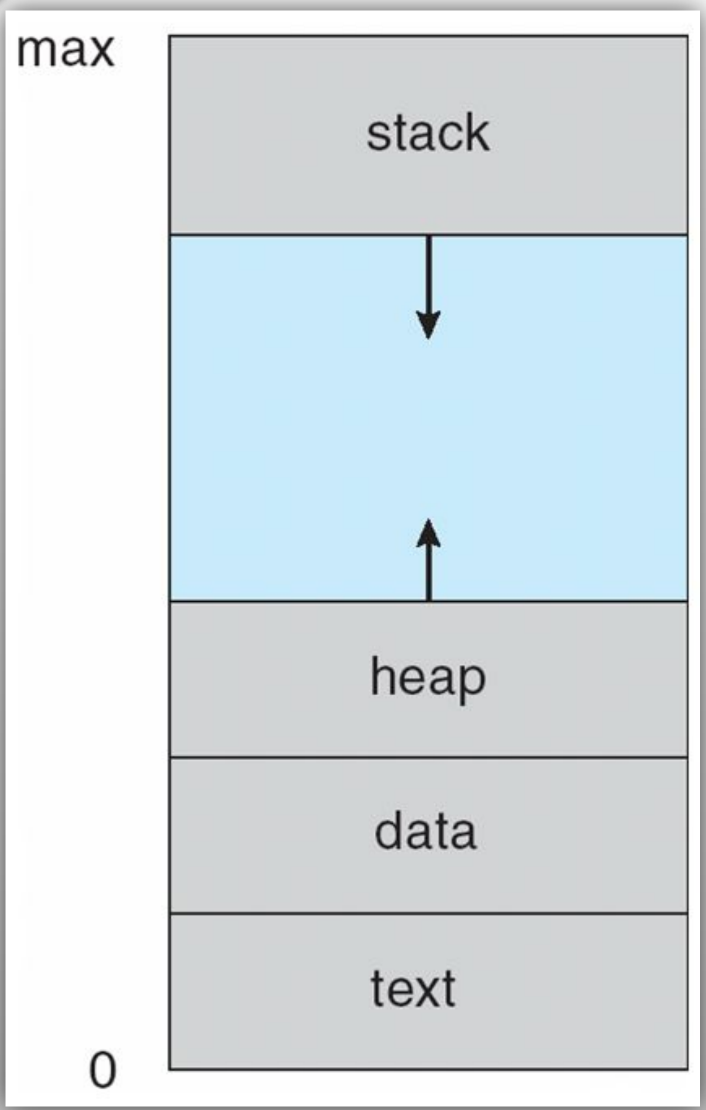
ROAD MAP

1. PROCESS CONCEPT
2. PROCESS SCHEDULING
3. OPERATIONS ON PROCESSES
4. INTERPROCESS COMMUNICATION
5. EXAMPLES OF IPC SYSTEMS
6. COMMUNICATION IN CLIENT-SERVER SYSTEMS

PROCESS CONCEPT

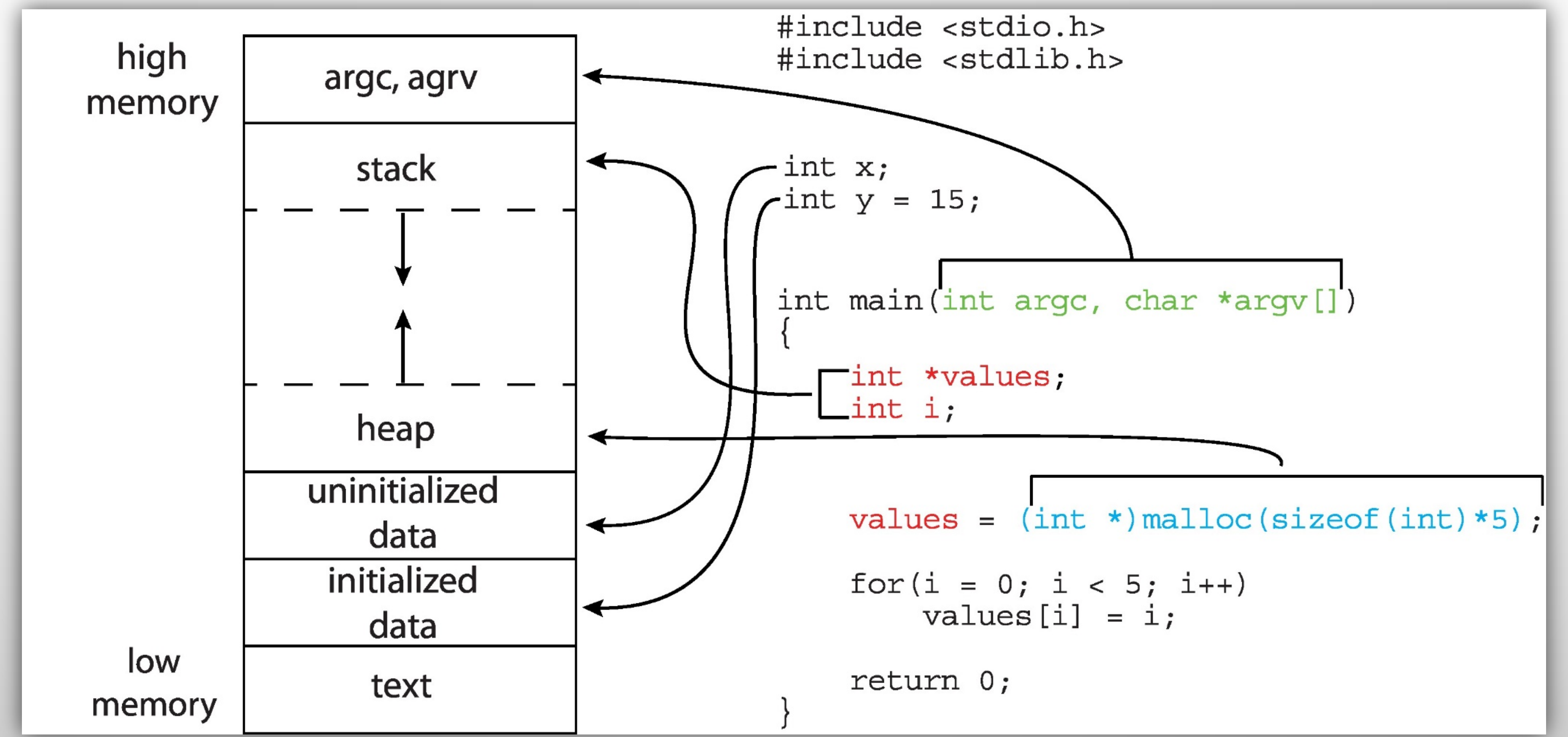
- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** –A process is an instance of a program in execution. Process execution must progress in sequential fashion.
- Program is passive entity, process is active
  - ❖ Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, cmd line etc

PROCESS IN MEMORY



- A process includes:
  - ❖ The program code, also called **text section**
  - ❖ Current activity including **program counter**, processor registers
  - ❖ **Stack** containing temporary data
    - ❖ Function parameters, return addresses, local variables
  - ❖ **Data section** containing global variables
  - ❖ **Heap** containing memory dynamically allocated during run time.

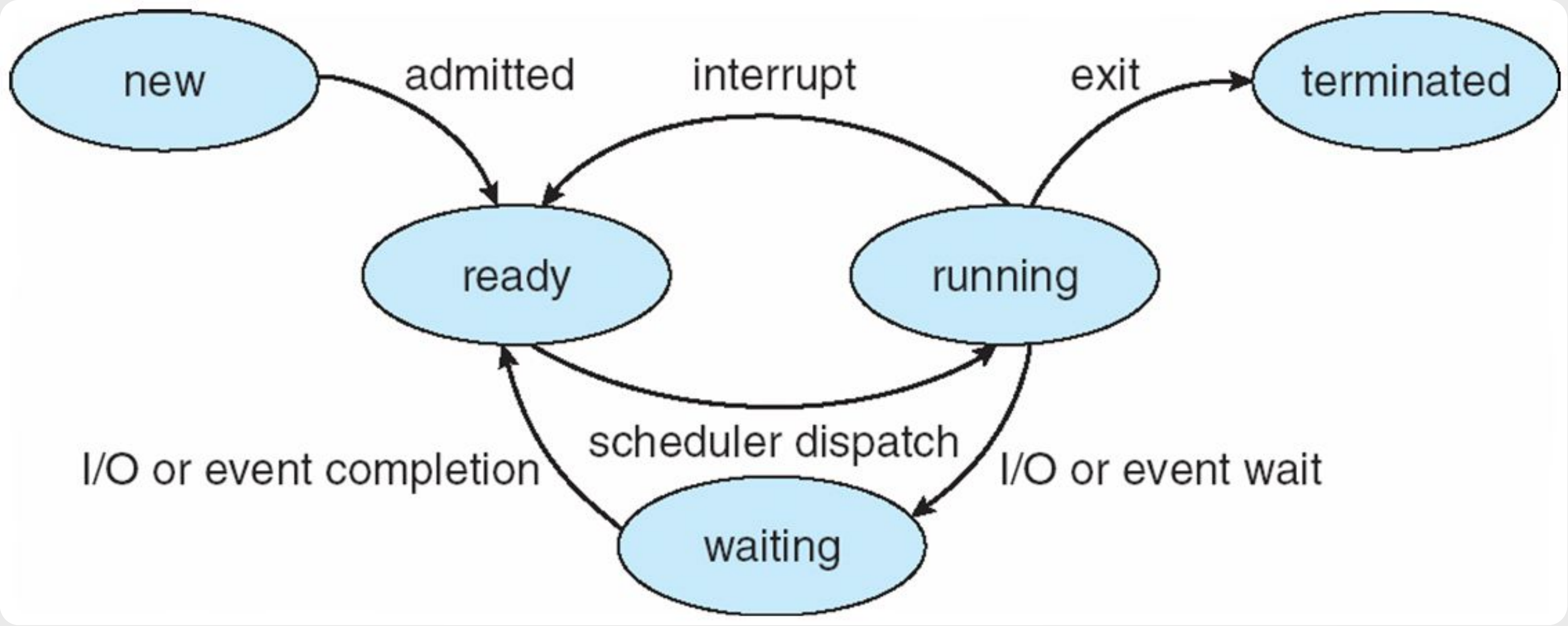
MEMORY LAYOUT OF A C PROGRAM



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

STATE TRANSITION DIAGRAM OF A PROCESS

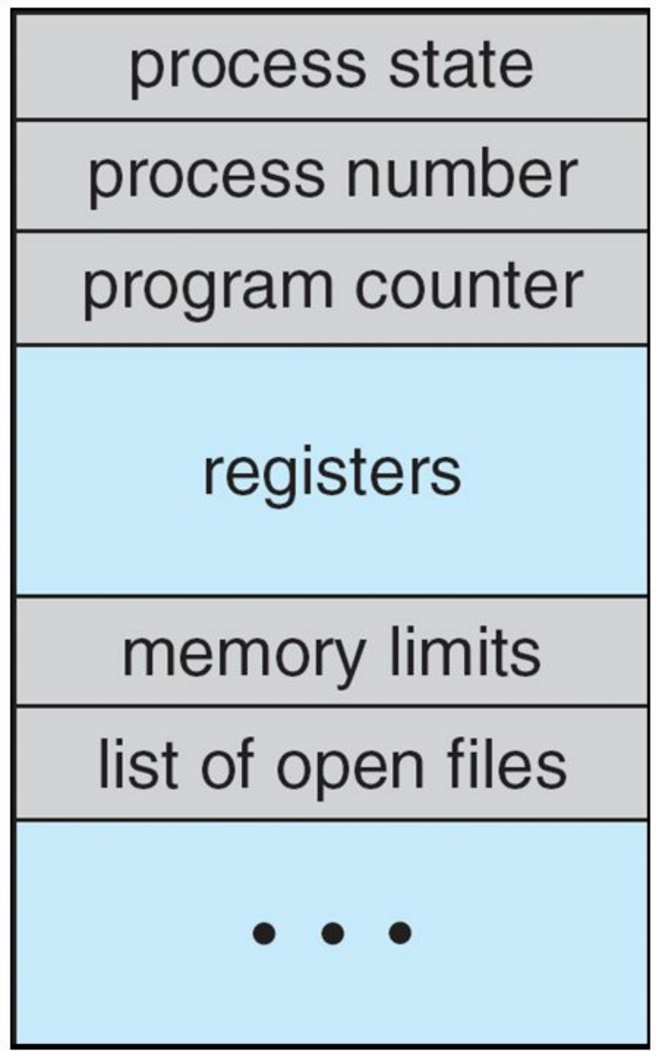


PROCESS CONTROL BLOCK (PCB)

Information associated with each process

(also called **TASK CONTROL BLOCK**)

- **Process state** – running, waiting, etc.
- **Process ID**, and parent process ID
- **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 SCHEDULING

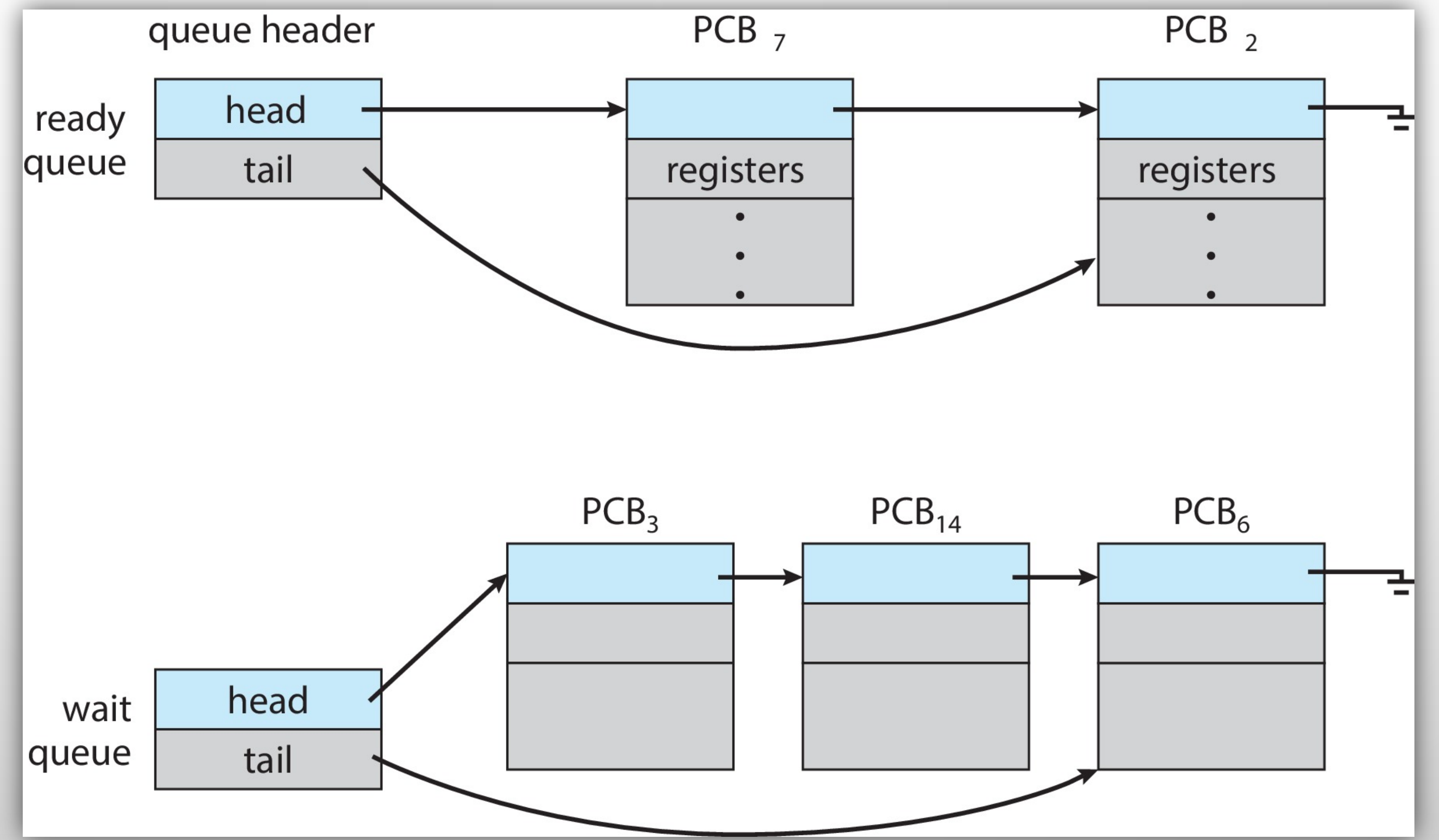
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU and maintains **scheduling queues** of processes

PROCESS SCHEDULING QUEUES

- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Wait queues** – set of processes waiting for an I/O device

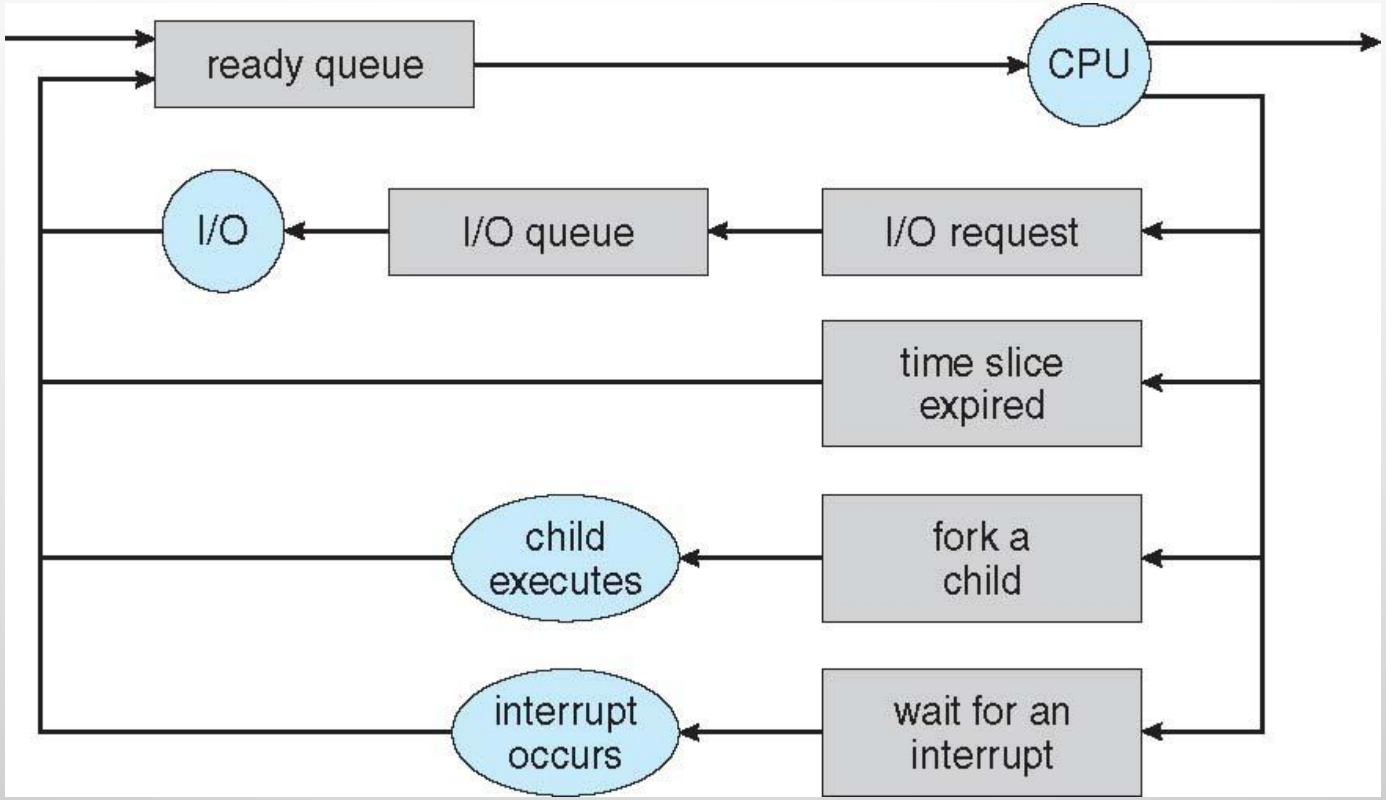
- Processes migrate among the various queues

READY AND WAIT QUEUES



REPRESENTATION OF PROCESS SCHEDULING

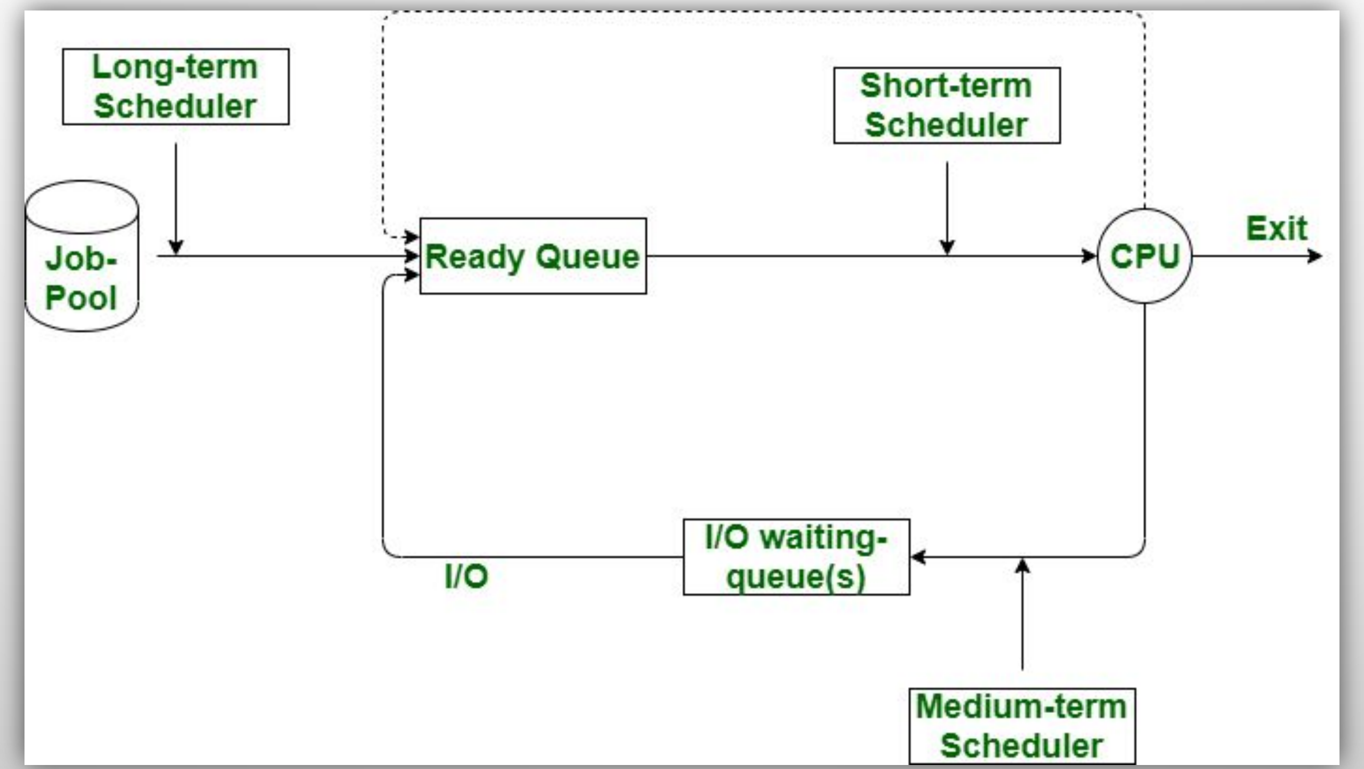
- **Queueing diagram** represents queues, resources, flows



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) $\Rightarrow$ (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) $\Rightarrow$ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**

SCHEDULERS

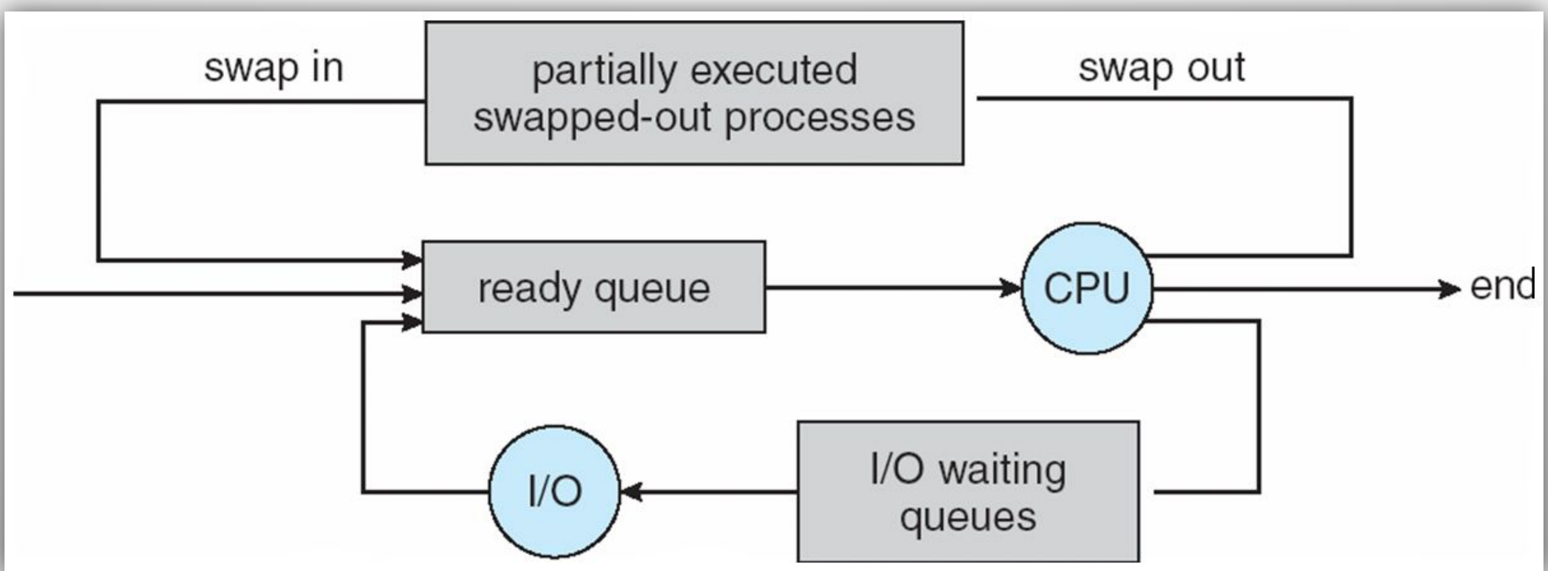

Long-term Scheduler
Short-term Scheduler
Job-Pool
Ready Queue
CPU
Exit
I/O waiting-queue(s)
I/O
Medium-term Scheduler

SCHEDULERS

- 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
- 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**



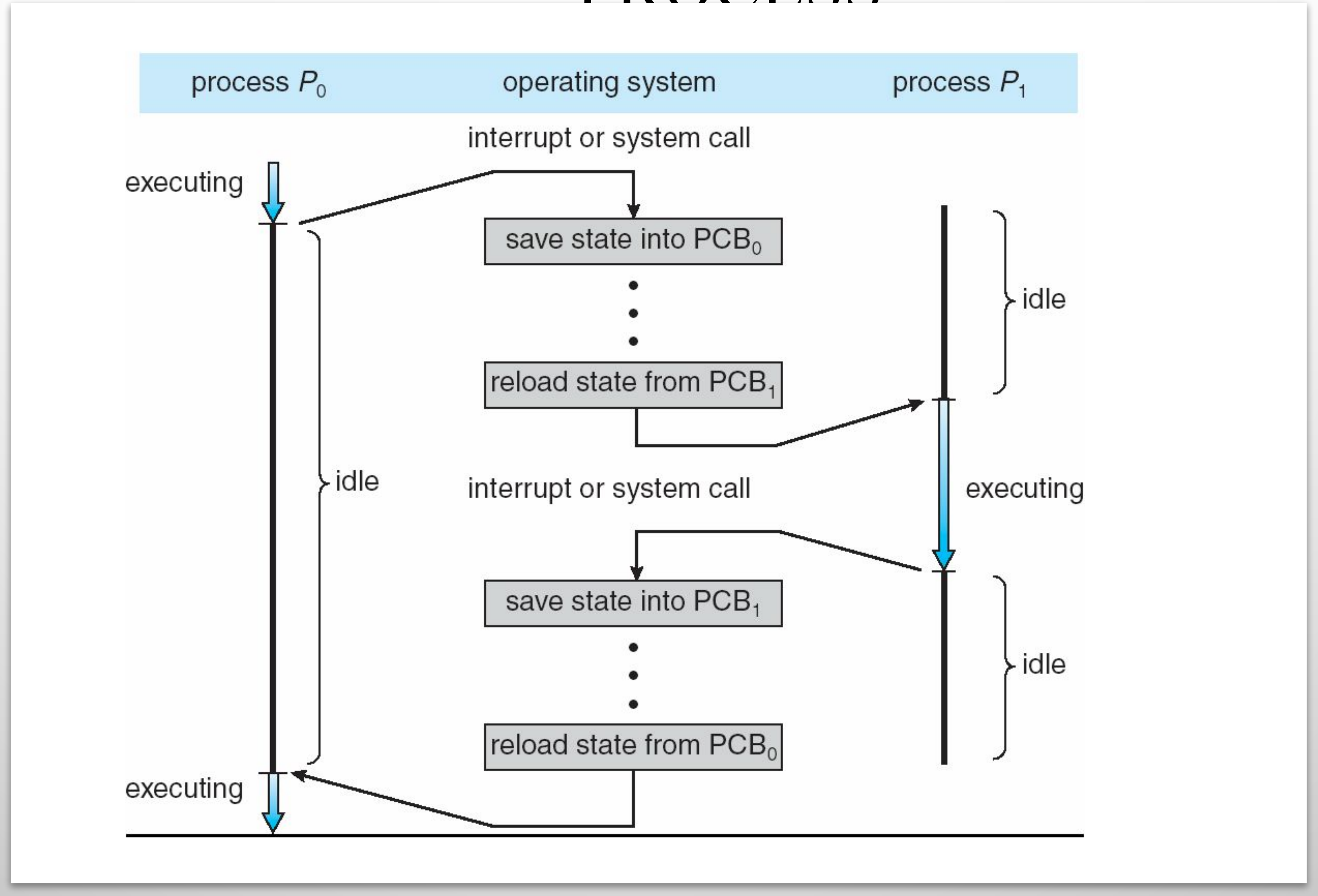
MULTITASKING IN MOBILE SYSTEMS

- Some mobile systems (e.g., Early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for 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 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, small memory use

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 overhead; the system does no useful work while switching

CPU SWITCH FROM PROCESS TO PROCESS



OPERATIONS ON PROCESSES

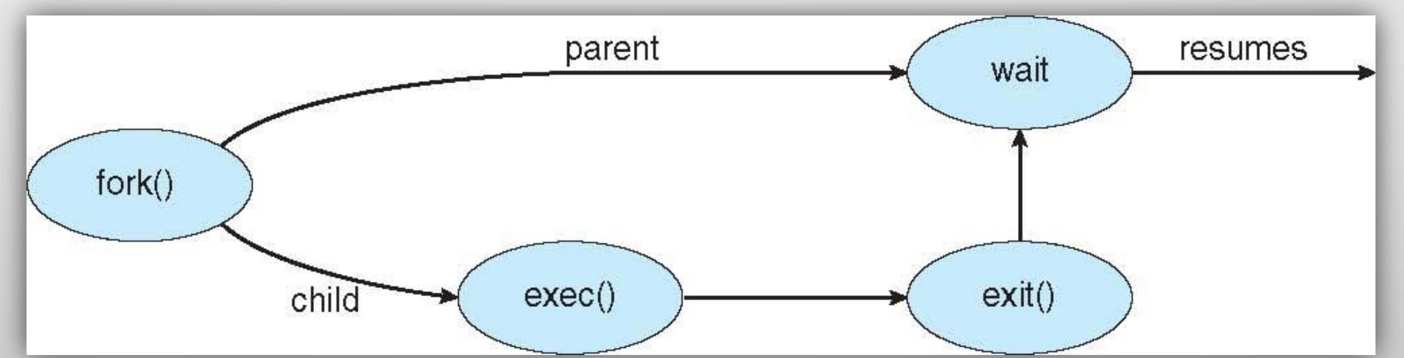
- System must provide mechanisms for:
  - Process creation,
  - Process termination,
  - and so, on as detailed next

PROCESS CREATION

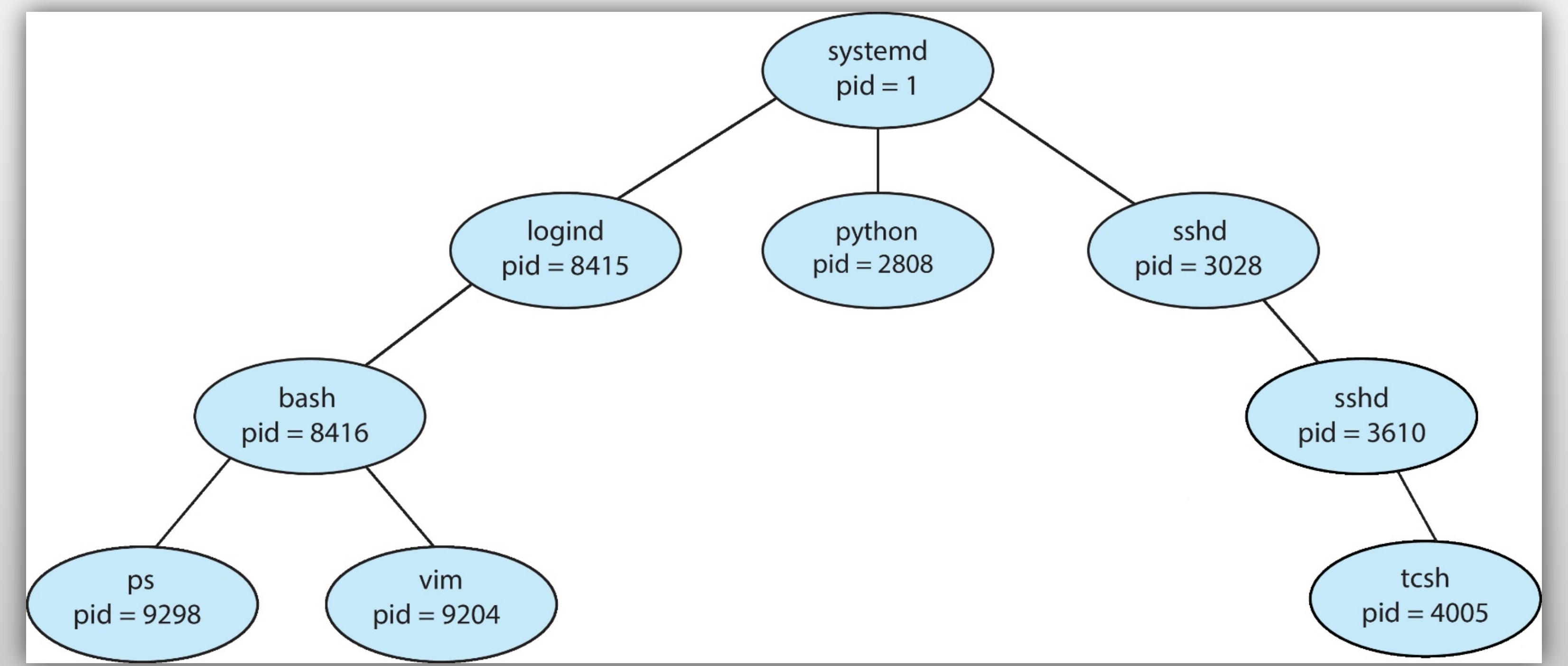
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset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

PROCESS 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



A TREE OF PROCESSES IN LINUX



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;
}
```

PROCESS TERMINATION

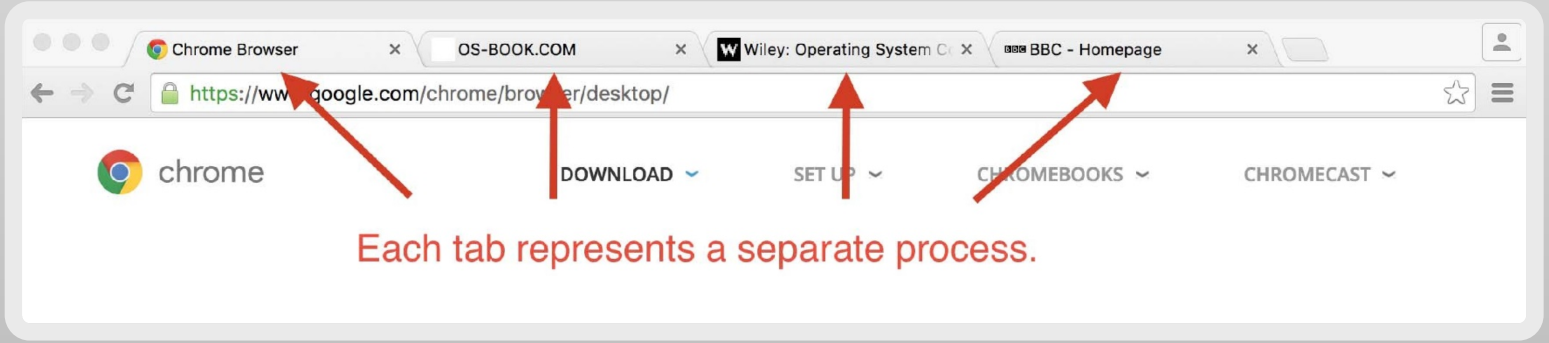
- Process executes last statement and then asks the operating system to delete it using the exit() system call.
  - Returns status data from child to parent (via wait())
  - Process' resources are DE allocated by operating system
- Parent may terminate the execution of children processes using the abort() 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
  - ALL CHILDREN TERMINATED - **CASCADING TERMINATION**

ANDROID PROCESS IMPORTANCE HIERARCHY

- MOBILE OPERATING SYSTEMS OFTEN HAVE TO TERMINATE PROCESSES TO RECLAIM SYSTEM RESOURCES SUCH AS MEMORY.
- Process hierarchy FROM **MOST** TO **LEAST** IMPORTANT:
  - FOREGROUND PROCESS
  - VISIBLE PROCESS
  - SERVICE PROCESS
  - BACKGROUND PROCESS
  - EMPTY PROCESS
- ANDROID WILL BEGIN TERMINATING PROCESSES THAT ARE LEAST IMPORTANT.

MULTI PROCESS ARCHITECTURE – CHROME BROWSER

- MANY WEB BROWSERS RAN AS SINGLE PROCESS (SOME STILL DO)
  - IF ONE WEB SITE CAUSES TROUBLE, ENTIRE BROWSER CAN HANG OR CRASH
- GOOGLE CHROME BROWSER IS 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



COOPERATING PROCESSES

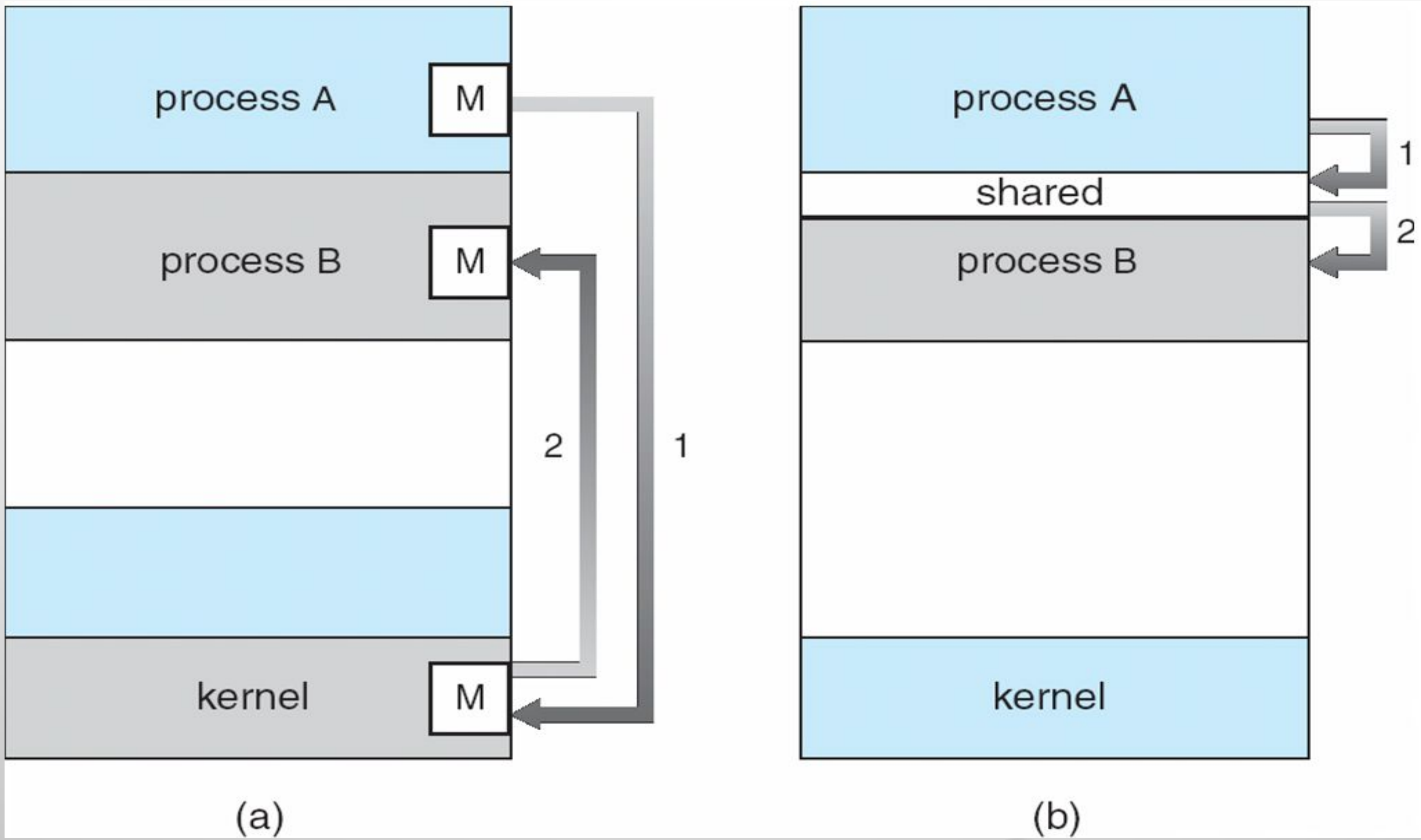
- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

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**

COMMUNICATIONS MODELS

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



PRODUCER-CONSUMER PROBLEM

- Paradigm for cooperating processes, *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

BOUNDED-BUFFER – SHARED-MEMORY SOLUTION

- Shared data

```
#Define BUFFER_SIZE 10
Typedef struct {
  . . .
} item;

Item buffer[buffer_size];
Int in = 0;
Int out = 0;
```

- Solution is correct, but can only use buffer_size-1 elements

BOUNDED-BUFFER – PRODUCER

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

BOUNDED BUFFER – CONSUMER

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

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

EXAMPLES OF IPC SYSTEMS - POSIX

- **POSIX SHARED MEMORY**
  - Process first creates shared memory segment

```
SEGMENT ID = SHMGET(IPC PRIVATE, SIZE, S IRUSR | S IWUSR);
```

  - Process wanting access to that shared memory must attach to it

```
SHARED MEMORY = (CHAR *) SHMAT(ID, NULL, 0);
```

  - Now the process could write to the shared memory

```
SPRINTF(SHARED MEMORY, "WRITING TO SHARED MEMORY");
```

  - When done a process can detach the shared memory from its address space

```
SHMDT(SHARED MEMORY);
```

INTER PROCESS 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*)
  - **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?

MESSAGE PASSING (CONT.)

- 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

- Operations
  - Create a new mailbox (port)
  - Send and receive messages through mailbox
  - Destroy a mailbox
- Primitives are defined as:

Send(*a, message*) – send a message to mailbox A

Receive(*a, message*) – receive a message from mailbox A

INDIRECT COMMUNICATION

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?

- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

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:
    - A valid message, or
    - null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

SYNCHRONIZATION (CONT.)

PRODUCER-CONSUMER BECOMES TRIVIAL

```
MESSAGE NEXT_PRODUCED;
      WHILE (TRUE) {
           /* PRODUCE AN ITEM IN NEXT PRODUCED */
      SEND(NEXT_PRODUCED);
      }
```

```
message next_consumed;
while (true) {
    receive(next_consumed);

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

BUFFERING

- 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 - 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

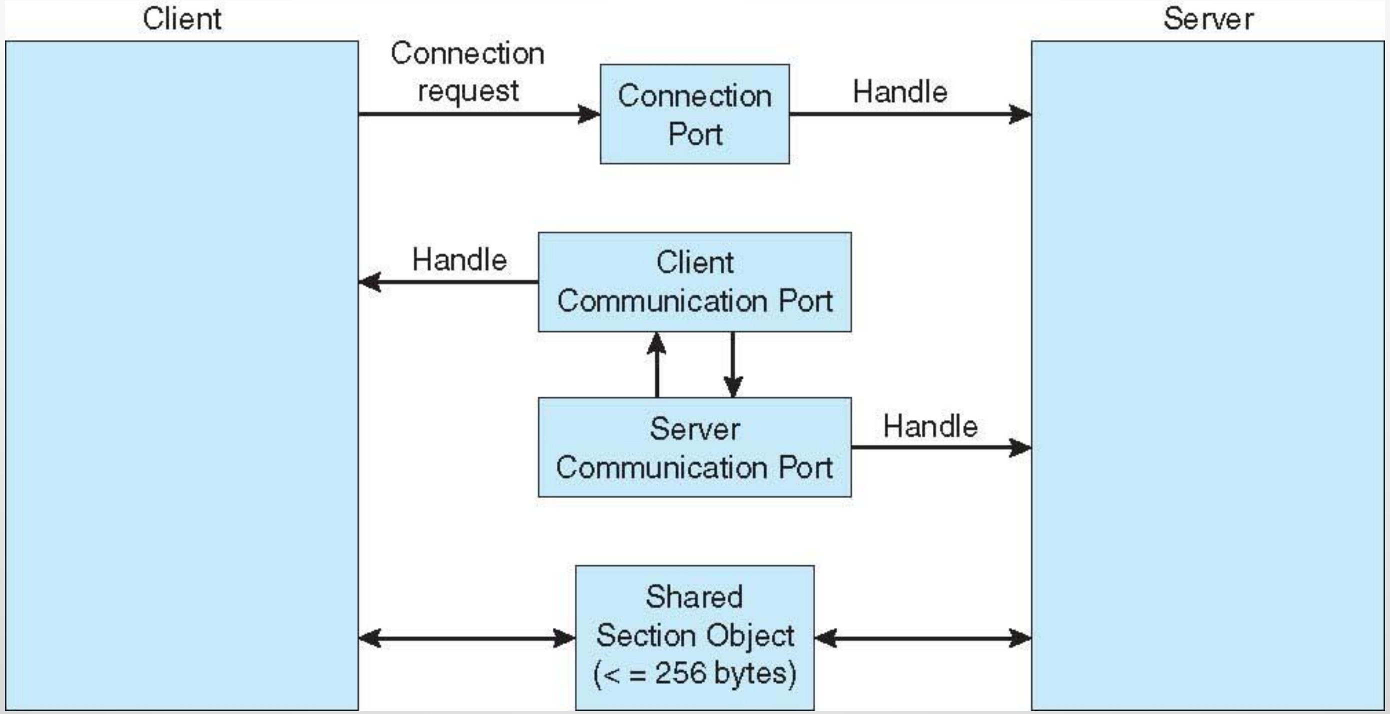
    Msg_send(), msg_receive(), msg_rpc()
  - Mailboxes needed for communication , created via

    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



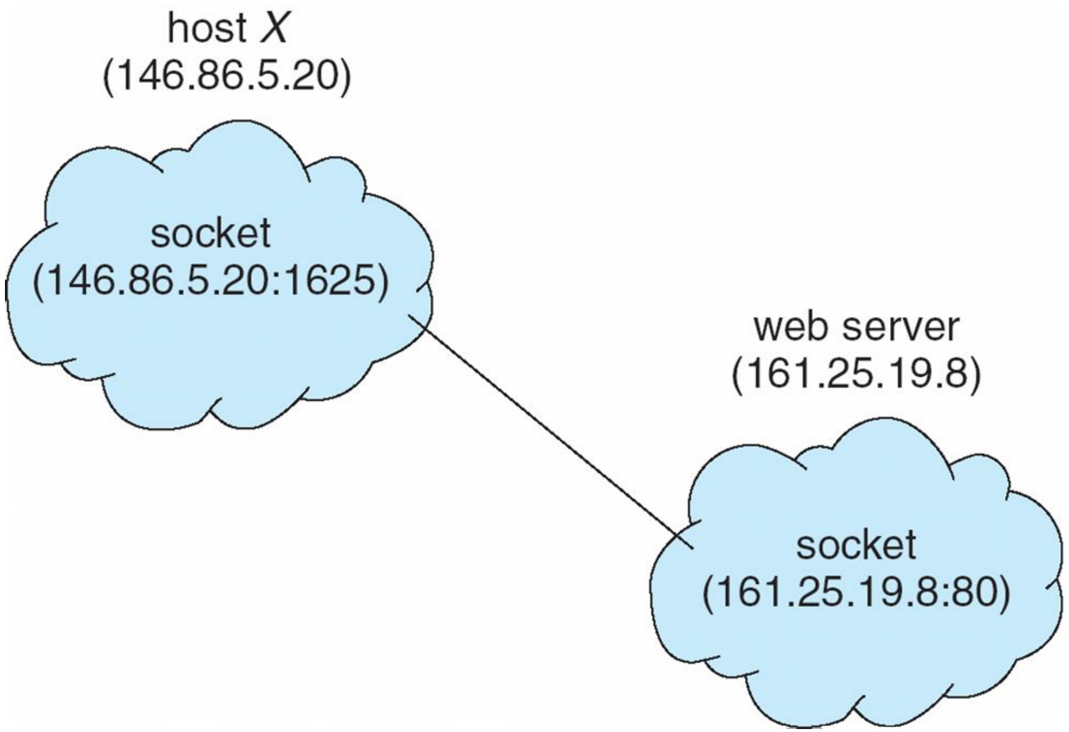
COMMUNICATIONS IN CLIENT-SERVER SYSTEMS

1. Sockets
2. Remote procedure calls
3. Pipes

SOCKETS

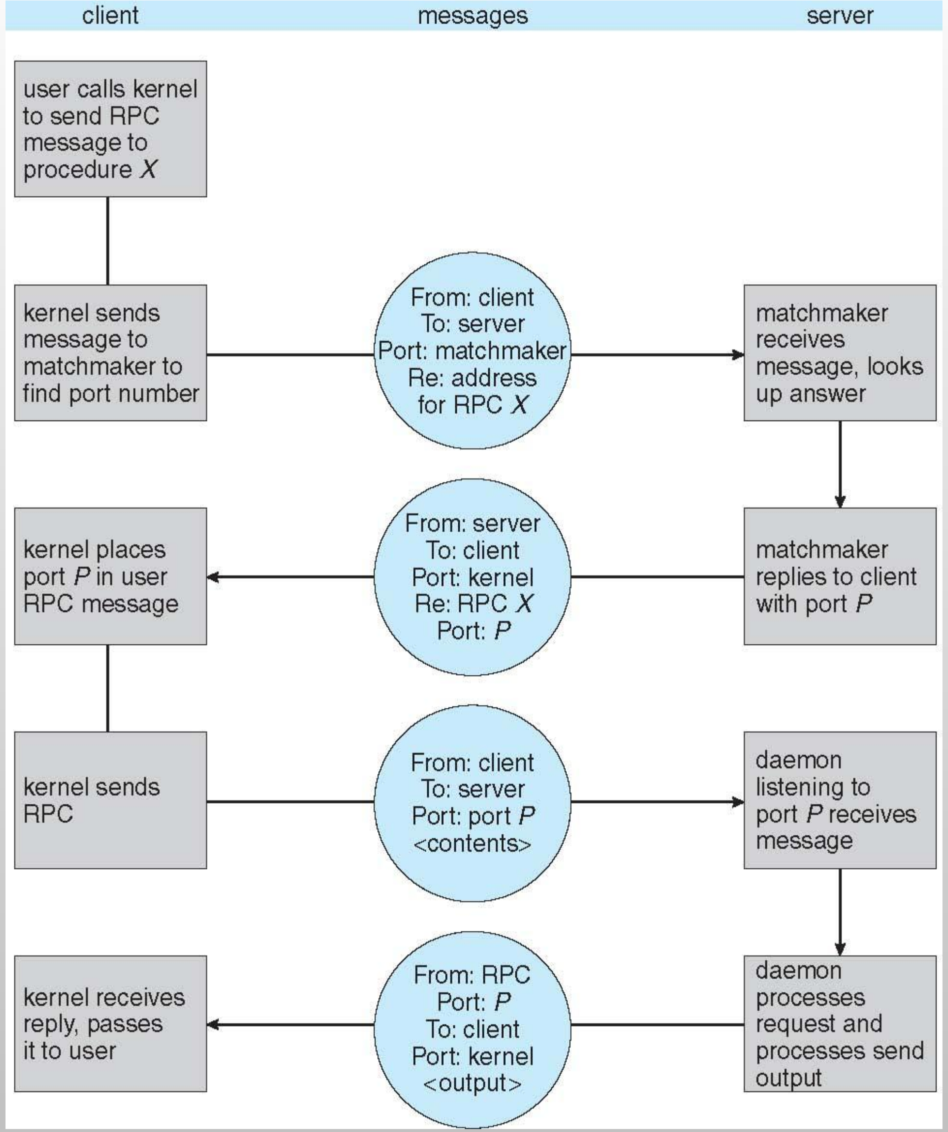
- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between 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

SOCKET COMMUNICATIO N



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



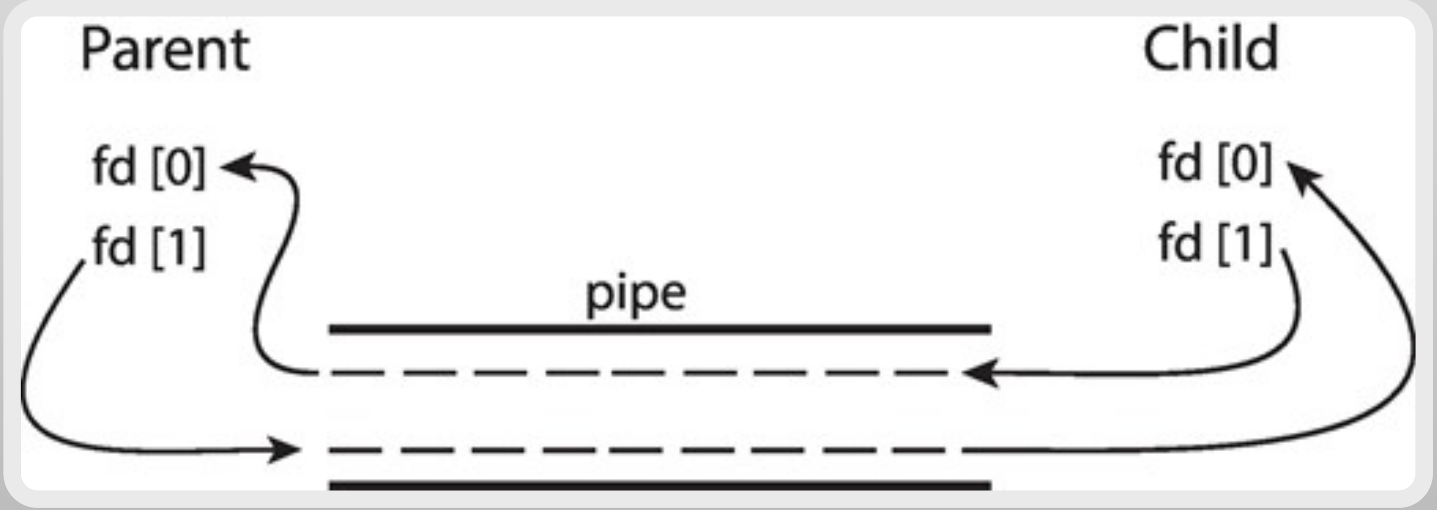
EXECUTION OF RPC

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

- 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

THANK YOU!