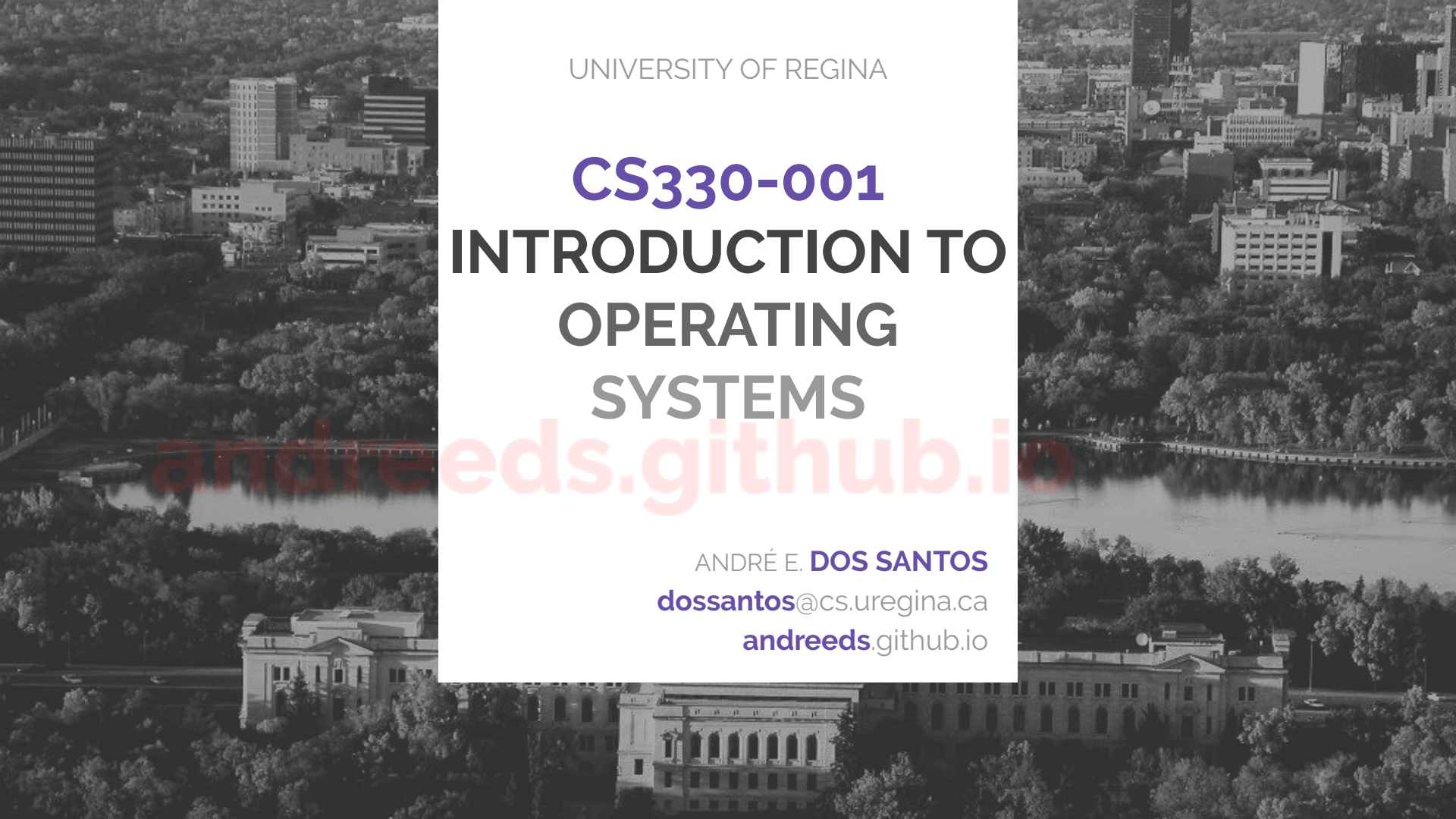
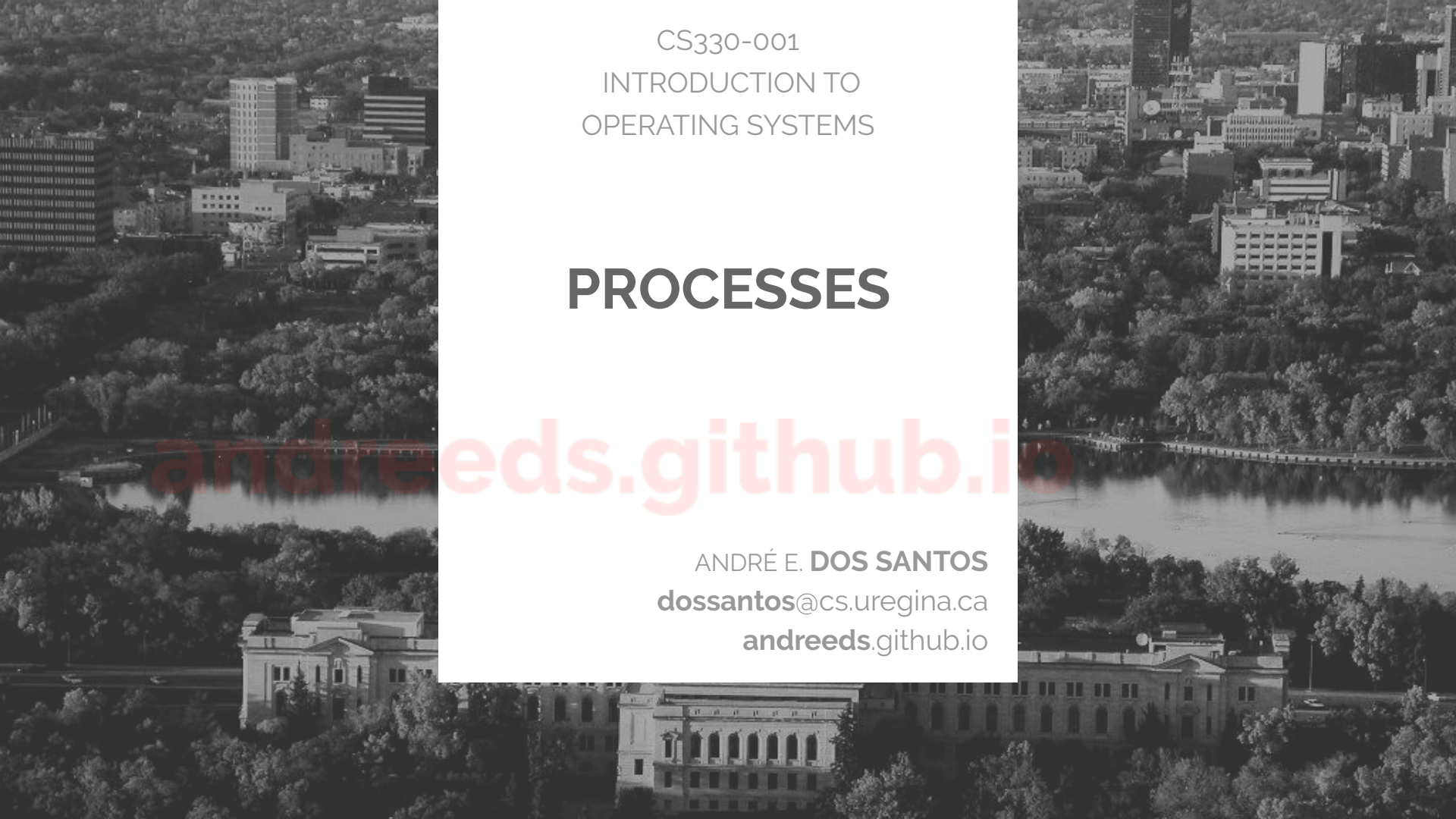


UNIVERSITY OF REGINA

CS330-001
INTRODUCTION TO OPERATING SYSTEMS

ANDRÉ E. **DOS SANTOS**
dossantos@cs.uregina.ca
andreeds.github.io



CS330-001
INTRODUCTION TO
OPERATING SYSTEMS

PROCESSES

ANDRÉ E. **DOS SANTOS**
dossantos@cs.uregina.ca
andreeds.github.io

PROCESS CONCEPT

Textbook uses the terms **job** and **process** almost interchangeably

- **Process** – a program in execution; process execution must progress in sequential fashion
- Program is **passive** entity stored on disk (**executable file**), process is **active**
  - **Program becomes process when executable file 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

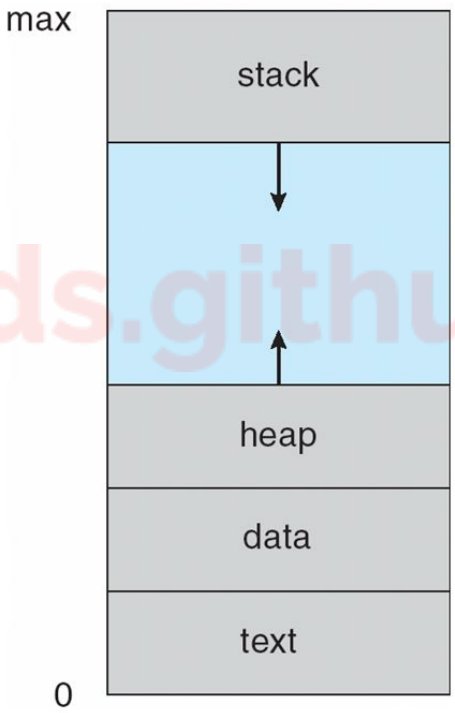
PROCESS CONCEPT

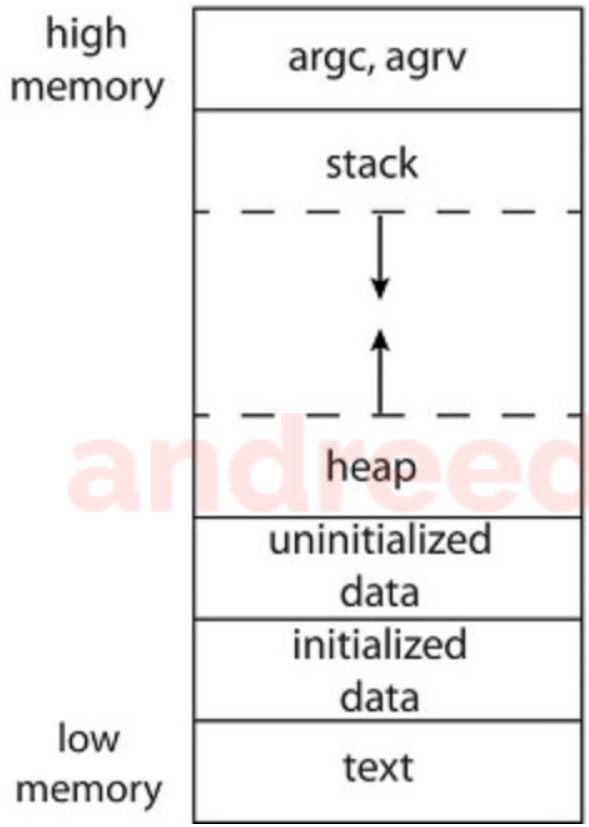
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap**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

THE PROCESS

p106

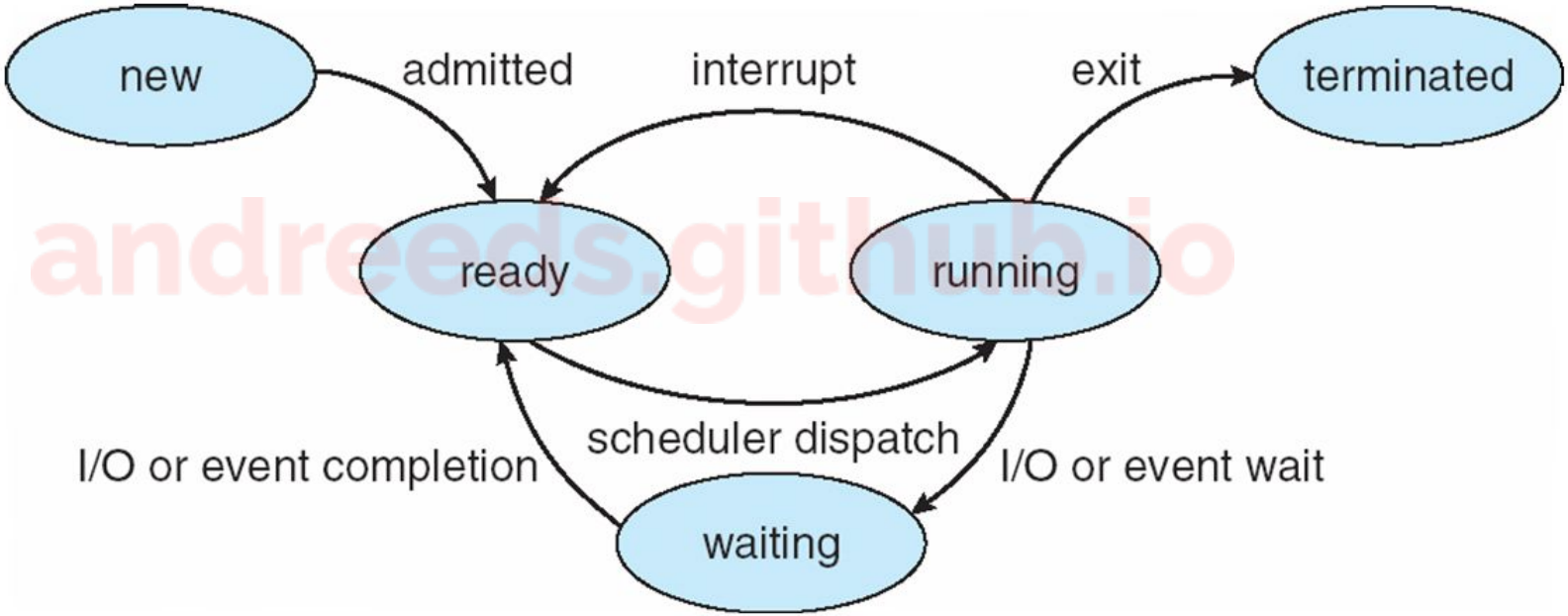
Layout of a process in memory






```c
#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;
}
```

PROCESS STATE



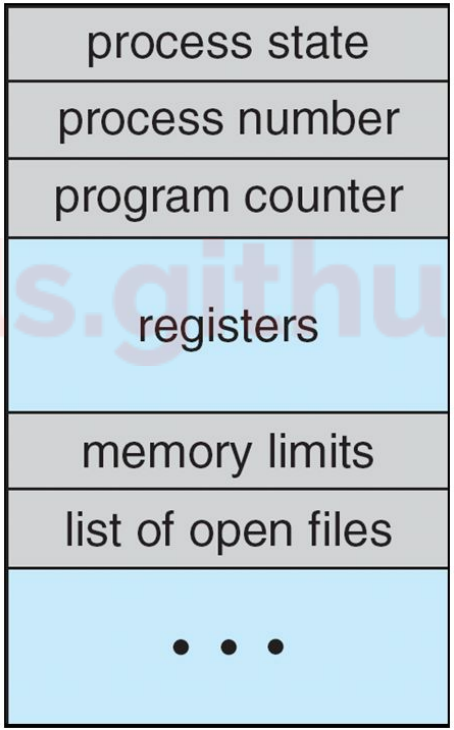
PROCESS STATE

- Multiple parts
  - 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

PROCESS CONTROL BLOCK (PCB)

- Each process is **represented** in the operating system by a **process control block (PCB)**—also called a **task control block**.
- It contains many pieces of **information** associated with a specific process, including these:
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information

PROCESS CONTROL BLOCK (PCB)



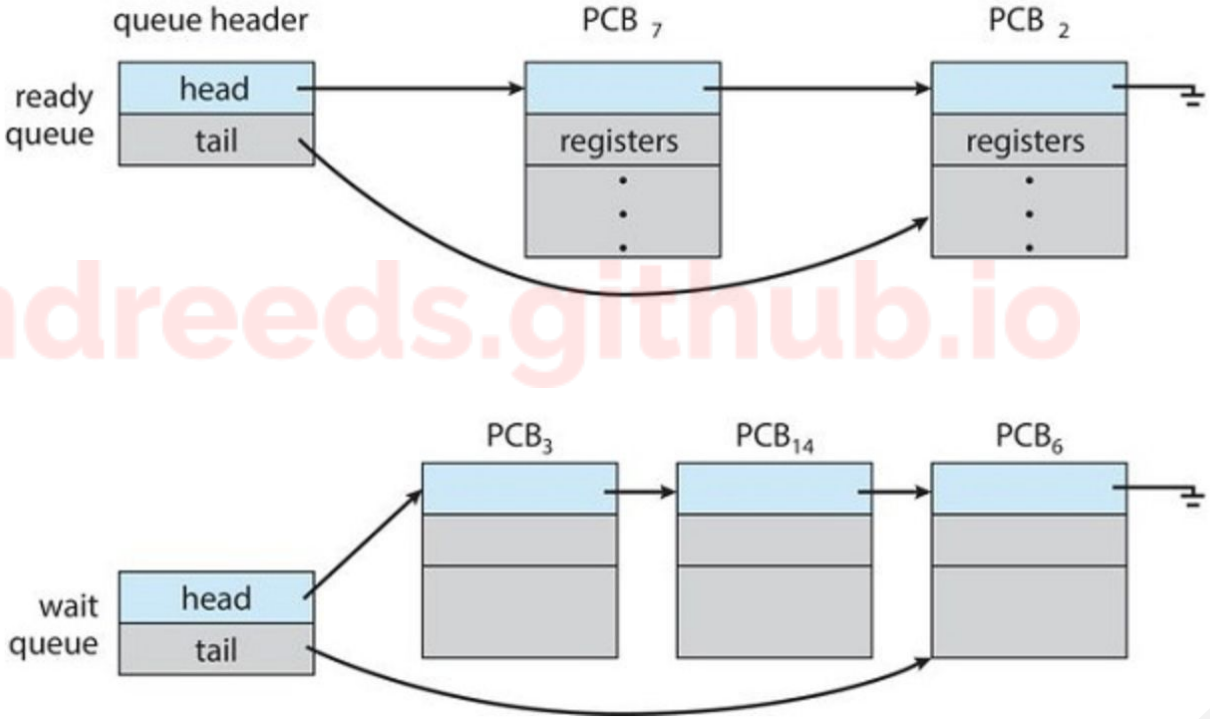
THREADS

- 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

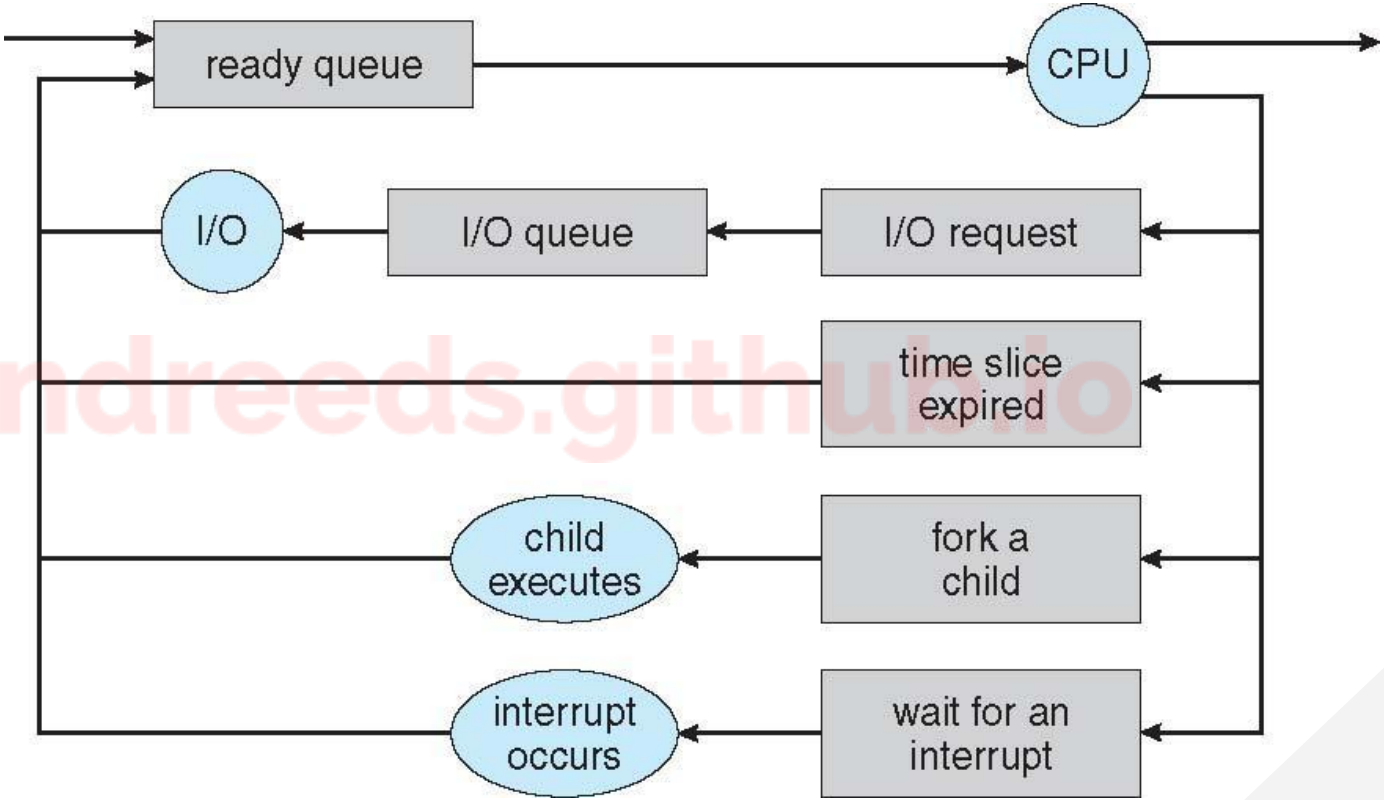
PROCESS SCHEDULING

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

READY QUEUE AND VARIOUS I/O DEVICE QUEUES



QUEUEING DIAGRAM



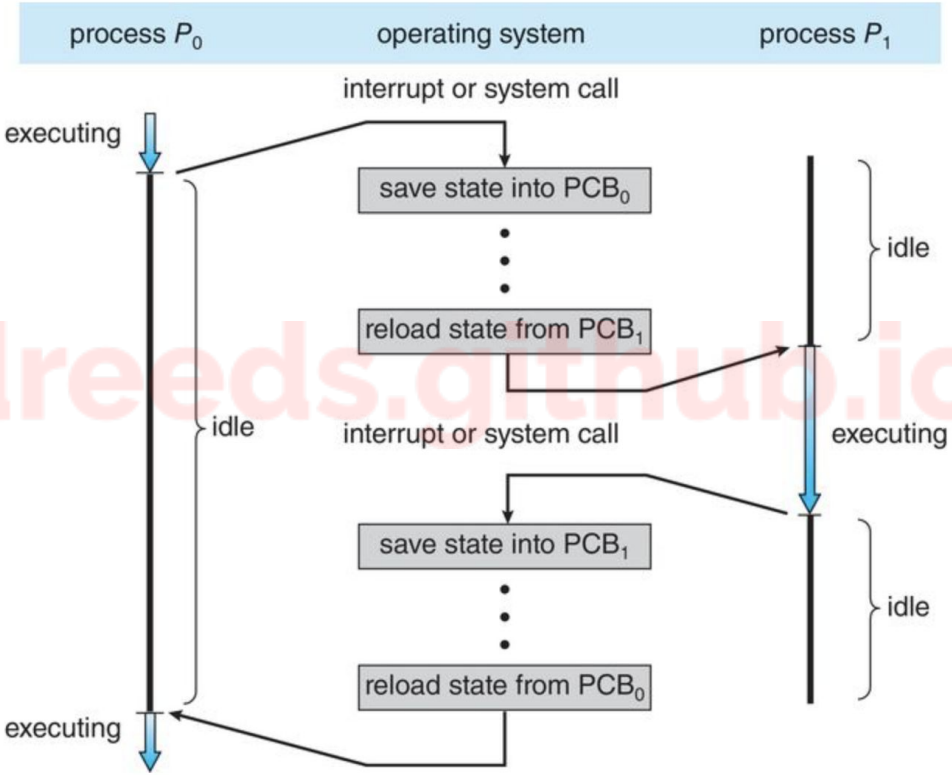
CPU SCHEDULING

- A process migrates among the ready queue and various wait queues throughout its lifetime
- The role of the CPU scheduler is to select from among the processes that are in the ready queue and allocate a CPU core to one of them
- The CPU scheduler must select a new process for the CPU frequently

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
    - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support

CONTEXT SWITCH



OPERATIONS ON PROCESSES

- System must provide mechanisms for:
  1. **process creation,**
  2. **process termination,**

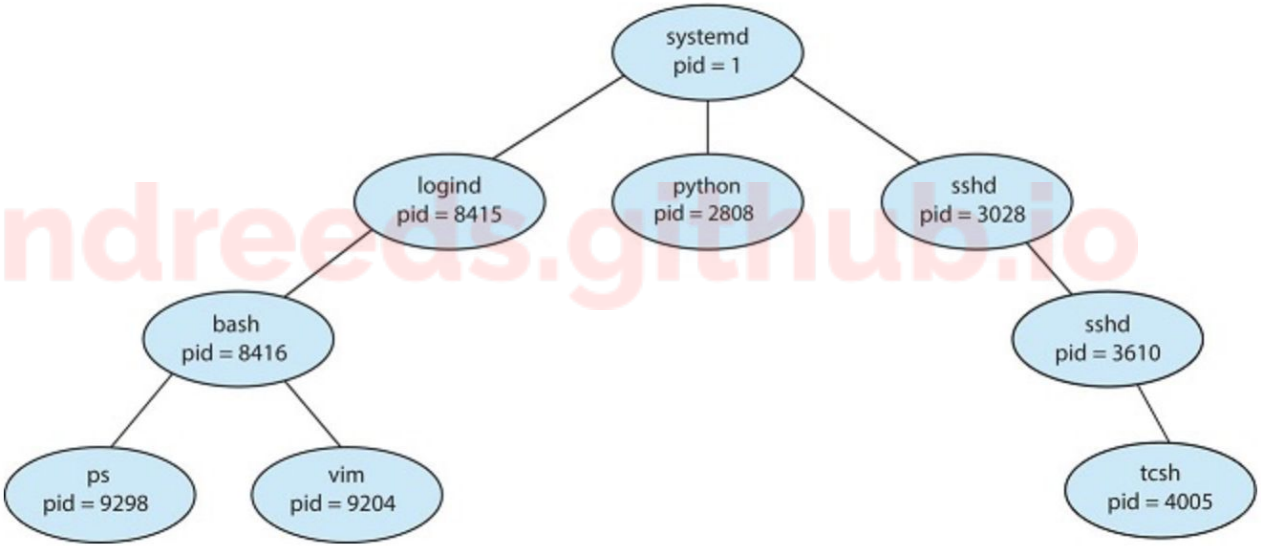
and so on as detailed next

OPERATIONS ON PROCESSES

1 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

A TREE OF PROCESSES IN LINUX

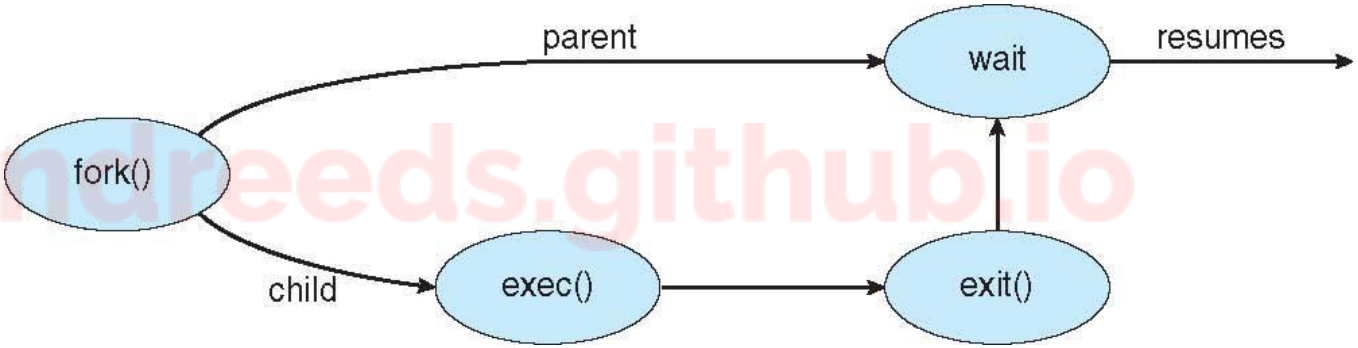


OPERATIONS ON PROCESSES

1 PROCESS CREATION

- 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

Process creation using the `fork()` system call



OPERATIONS ON PROCESSES

2 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

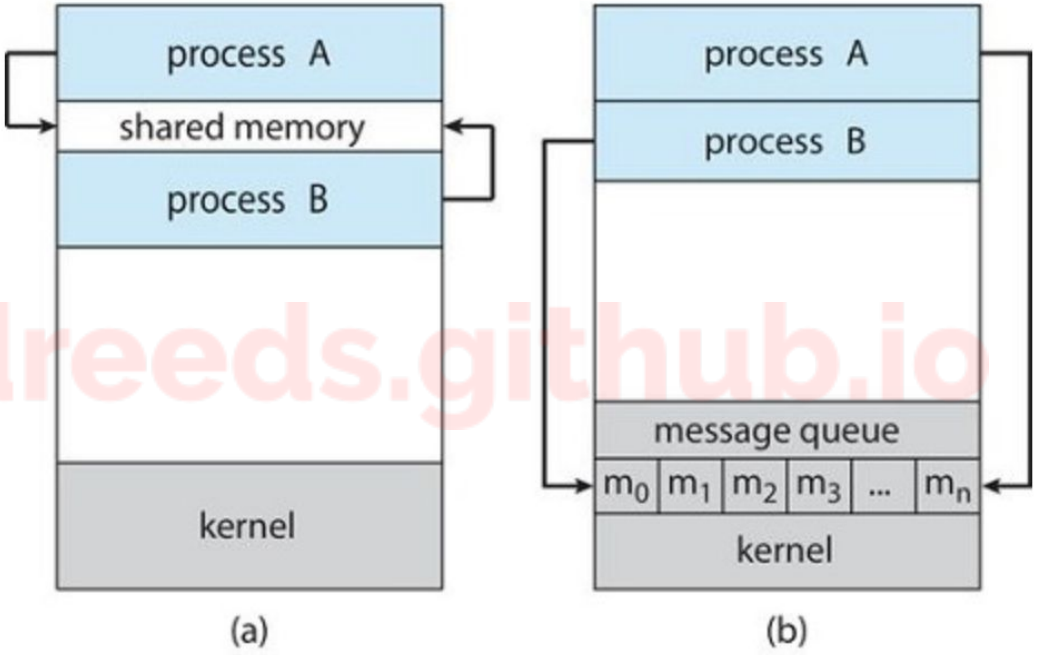
2 PROCESS TERMINATION

- 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**.
    - 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**

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



Shared memory **Message passing**

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

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
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory

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

- 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
  - P1, P2, and P3 share **mailbox A**
  - P1, sends; P2 and P3 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**

BUFFERING

- Buffers are generally used when there is a difference between the rate at which data is received and the rate at which it can be processed.
- 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

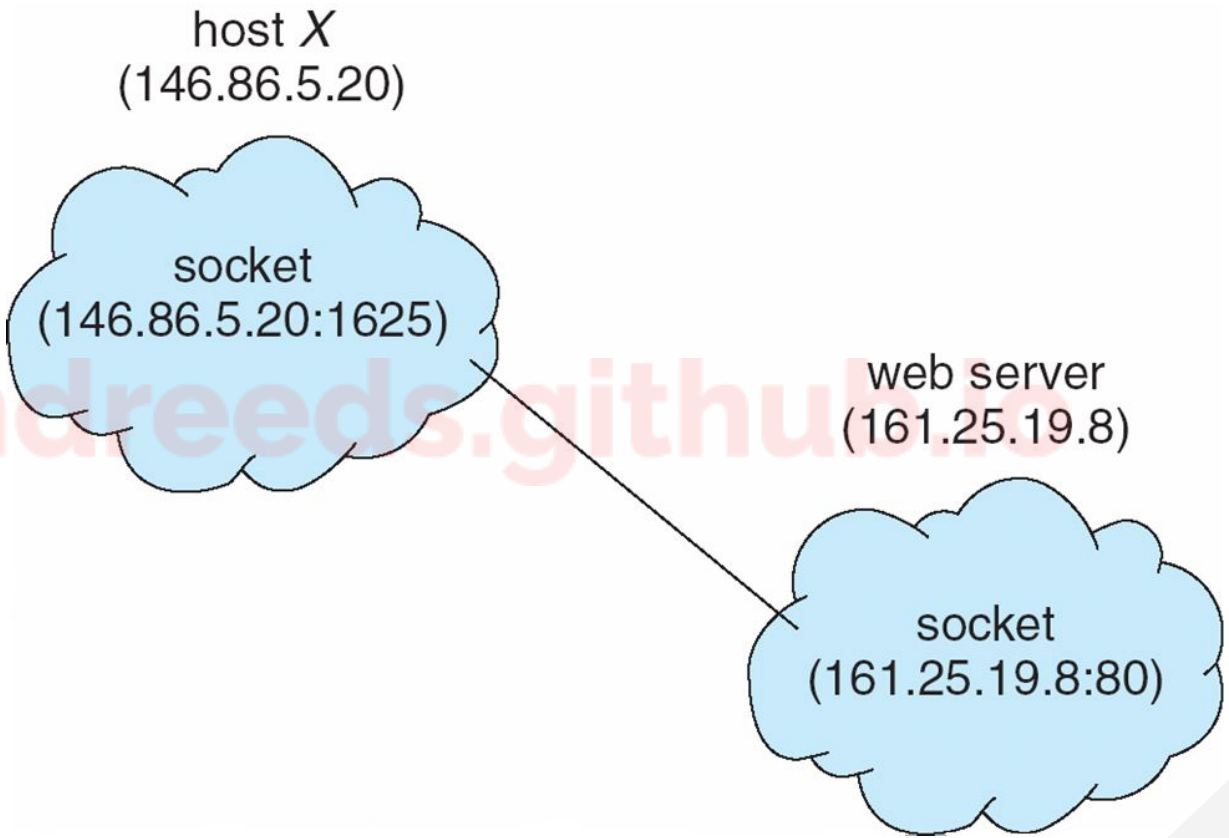
COMMUNICATIONS IN CLIENT-SERVER SYSTEMS

- Sockets
- Remote Procedure Calls
- 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 COMMUNICATION



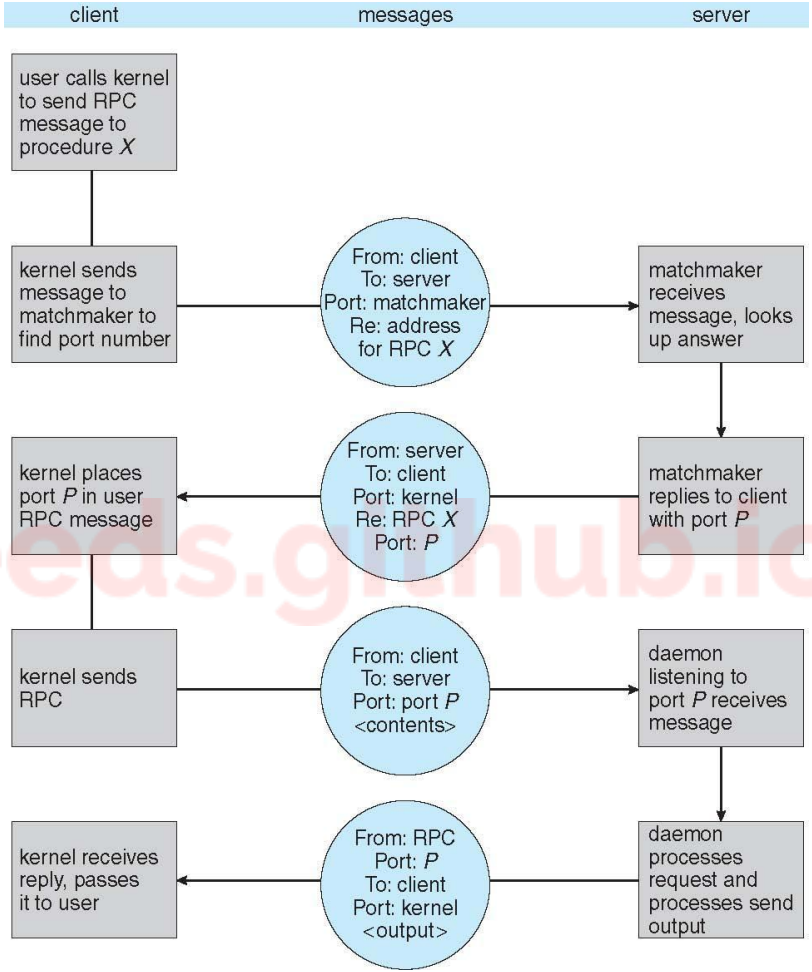
REMOTE PROCEDURE CALLS

- **Remote procedure call (RPC)** is a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details

REMOTE PROCEDURE CALLS

- **Remote procedure call (RPC)** is a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details
- When program statements that use RPC framework are compiled into an executable program, a **stub** is included in the compiled code that acts as the representative of the remote procedure code
- When the program is run and the procedure call is issued, the stub receives the request and forwards it to a client runtime program in the local computer

stub
a small program routine that substitutes for a longer program, possibly to be loaded later or that is located remotely

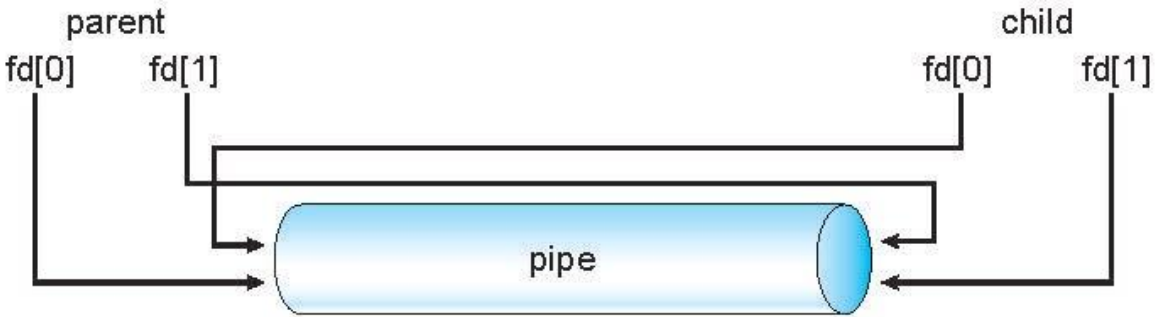


PIPES

- 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



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

REVIEW QUESTIONS
PROCESSES

The ____ of a process contains temporary data such as function parameters, return addresses, and local variables.

A. text section
B. data section
C. program counter
D. stack

A process control block ____.

A. includes information on the process's state
B. stores the address of the next instruction to be processed by a different process
C. determines which process is to be executed next
D. is an example of a process queue

A process may transition to the Ready state by which of the following actions?

A. Completion of an I/O event
B. Awaiting its turn on the CPU
C. Newly-admitted process
D. All of the above

REVIEW QUESTIONS
PROCESSES

A _____ saves the state of the currently running process and restores the state of the next process to run.

A. save-and-restore
B. state switch
C. context switch
D. none of the above

When a child process is created, which of the following is a possibility in terms of the execution or address space of the child process?

A. The child process runs concurrently with the parent.
B. The child process has a new program loaded into it.
C. The child is a duplicate of the parent.
D. All of the above

REVIEW QUESTIONS
PROCESSES

Which of the following statements is true?

A. Shared memory is typically faster than message passing.

B. Message passing is typically faster than shared memory.

C. Message passing is most useful for exchanging large amounts of data.

D. Shared memory is far more common in operating systems than message passing.

A blocking send() and blocking receive() is known as a(n) _____

A) synchronized message

B) rendezvous

C) blocked message

D) asynchronous message

"Under indirect communication, each process that wants to communicate must explicitly name the recipient or sender of the communication." True or False?

REVIEW QUESTIONS
PROCESSES

Imagine that a host with IP address 150.55.66.77 wishes to download a file from the web server at IP address 202.28.15.123. Select a valid socket pair for a connection between this pair of hosts.

A) 150.55.66.77:80 and 202.28.15.123:80
B) 150.55.66.77:150 and 202.28.15.123:80
C) 150.55.66.77:2000 and 202.28.15.123:80
D) 150.55.66.77:80 and 202.28.15.123:3500

When a process creates a new process using the `fork()` operation, which of the following states is shared between the parent process and the child process?

a. Stack
b. Heap
c. Shared memory segments

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

int value = 5;

int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0) { /* child process */
        value += 15;
        return 0;
    }
    else if (pid > 0) { /* parent process */

        wait(NULL);
        printf("PARENT: value = %d",value); /* LINE A */

        return 0;
    }
}
```

Explain what the output will be at LINE A.

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


int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
```

Including the initial parent process, how many processes are created by the program?