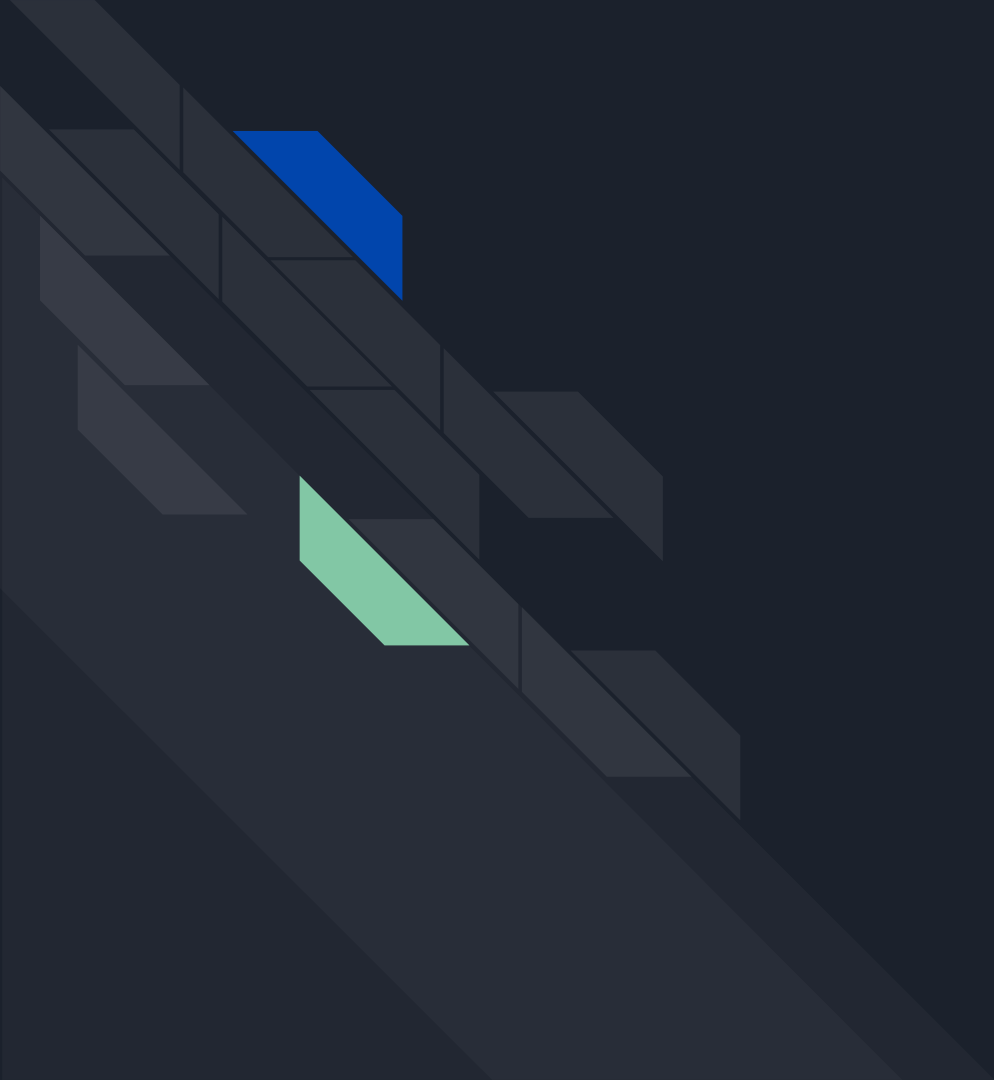




# ARQUITECTURA Y SISTEMAS OPERATIVOS

## CLASE 3

PROCESOS



REPASO



# INTERRUPCIONES

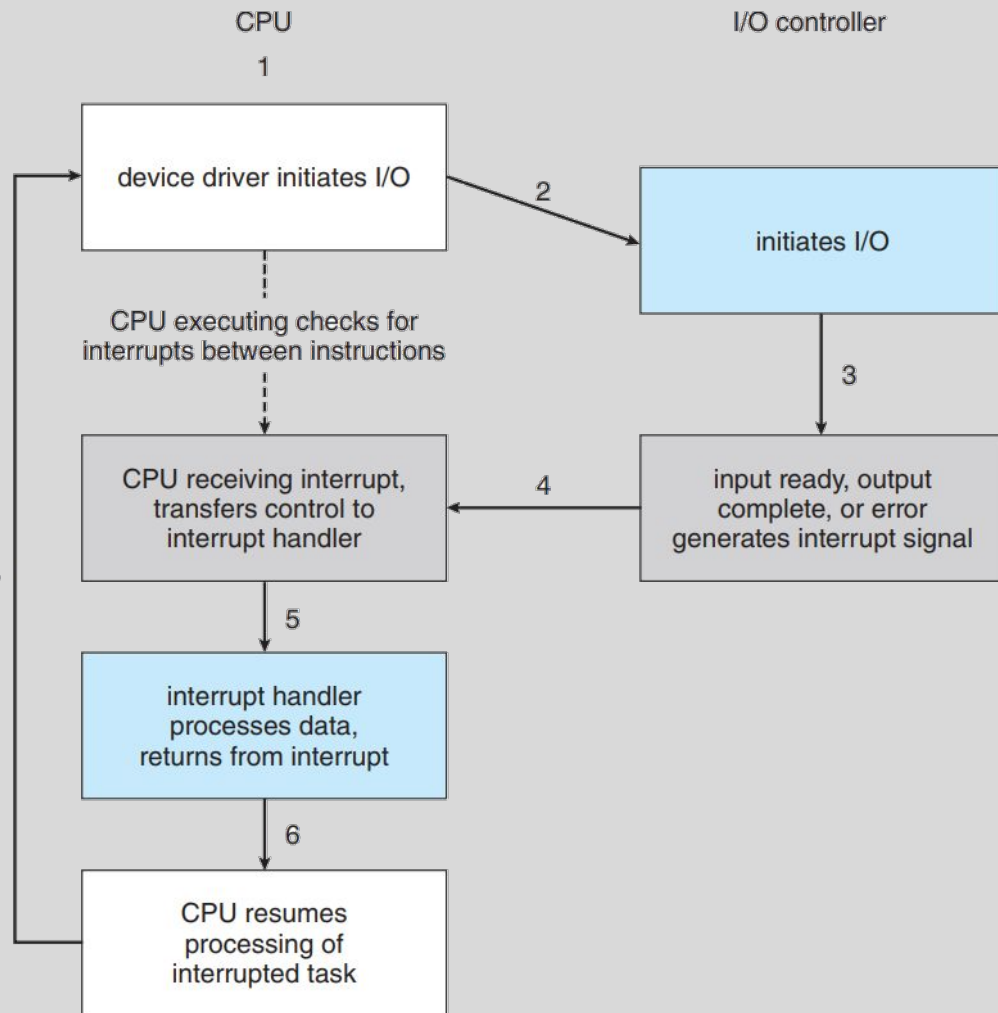
Consideren una operación típica: un programa que lee el valor ingresado por una tecla.

Para iniciar una operación de I/O, el *device driver* carga los registros correspondientes en el *device controller*.

El *device controller*, examina el contenido de estos registros para determinar la acción a realizar ("leer un carácter del teclado"). Una vez presionada la tecla, inicia la transferencia de datos **desde** el dispositivo **hacia** su búfer local y, cuando se completada la transferencia, el *device controller* informa al *device driver* que ha finalizado la operación.

Pero ¿cómo informa el *device controller* al *device driver* que ha finalizado la operación?

Mediante una **interrupción**.



# INTERRUPCIONES

El hardware puede activar una interrupción en cualquier momento enviando una señal a la CPU.



Cuando la CPU se interrumpe, detiene lo que está haciendo y transfiere inmediatamente la ejecución a una ubicación fija que contiene la dirección de inicio de la rutina de servicio de la interrupción.

La rutina de servicio se ejecuta.

Si la interrupción necesita modificar el estado del procesador, debe guardar explícitamente el estado actual y luego restaurarlo antes de regresar el control.

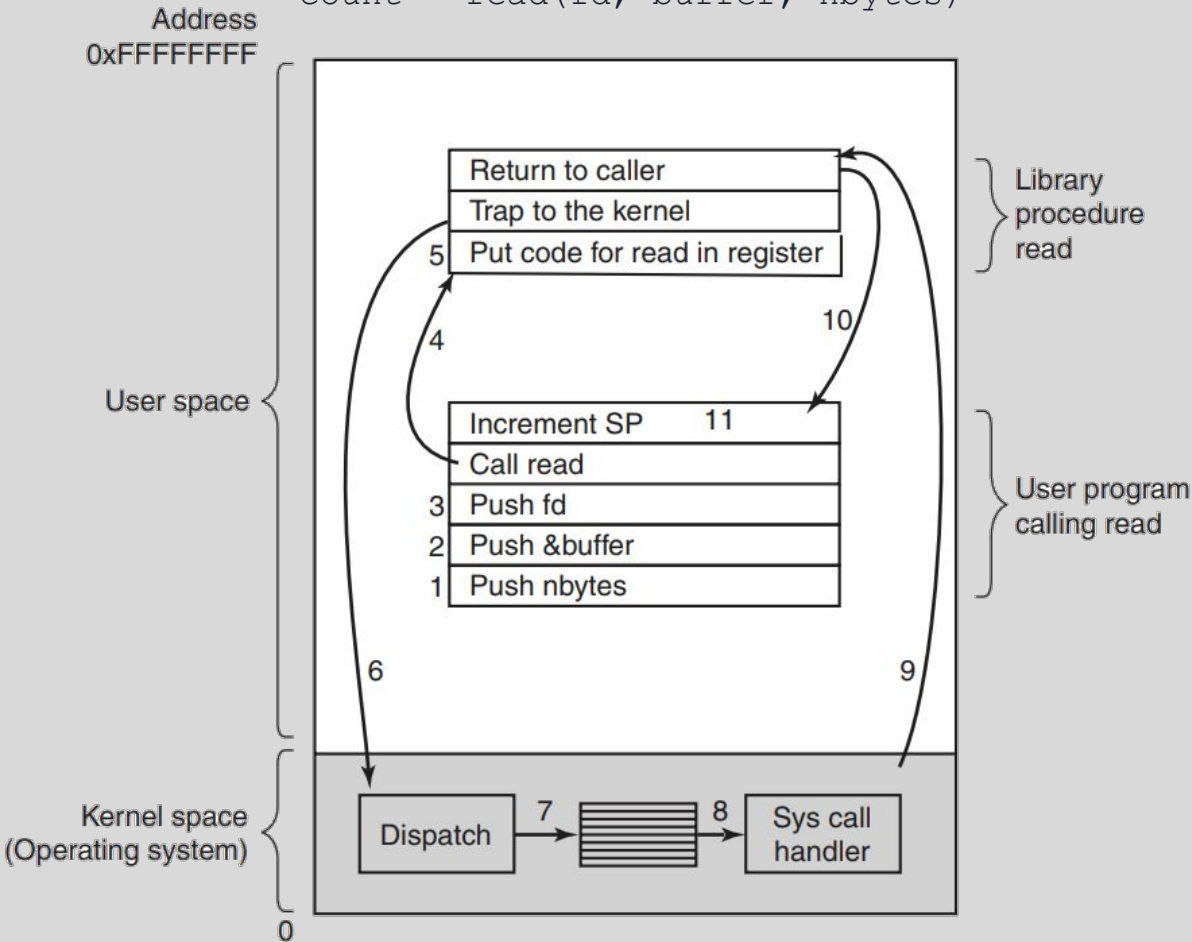
Una vez atendida la interrupción, el cálculo interrumpido se reanuda como si la interrupción no se hubiera producido.

# INTERRUPCIONES

src > 01-introduccion >  read\_input\_example.c >  main()

```
1  #include <stdio.h>
2
3  int main()
4  {
5      char input[100]; // Buffer to store user input
6
7      printf("Enter a value: ");
8      if (fgets(input, sizeof(input), stdin) != NULL) {
9          printf("You entered: %s", input);
10     } else {
11         printf("Error reading input.\n");
12     }
13
14     return 0;
15 }
```

```
count = read(fd, buffer, nbytes)
```



# SYSTEM CALL

Las **system calls** proporcionan los medios para que un programa de usuario solicite al SO que realice tareas reservadas en nombre del programa de usuario. Una **system call** generalmente lleva la forma de un *salto* a una ubicación específica en la tabla de interrupciones.

Las *system calls* se realizan en una serie de pasos.

# SYSTEM CALLS

```
src > 01-introduccion > C read_file_example.c > main()
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define BUFFER_SIZE 1024
5
6  int main()
7  {
8      FILE *file;
9      char buffer[BUFFER_SIZE];
10
11     // Open file
12     file = fopen("../recursos/tup.txt", "r");
13     if (file == NULL) {
14         perror("Error opening file");
15         return 1;
16     }
17
18     printf("Contenido del archivo TUC.txt:\n");
19     printf("-----\n");
20
21     // Read and print file content
22     while (fgets(buffer, BUFFER_SIZE, file) != NULL) {
23         printf("%s", buffer);
24     }
25
26     // Close file
27     if (fclose(file) != 0) {
28         perror("Error closing file");
29         return 1;
30     }
31
32     return 0;
33 }
```

PROCESOS







# BIBLIOGRAFIA

- Operating System Concepts. By Abraham, Silberschatz.
  - Capitulo III

# TEMAS DE LA CLASE

- Concepto de Proceso
  - El proceso
  - Estados de un Proceso
  - Bloque de control de Procesos
- Planificación de Procesos
  - Colas de Planificación
  - Planificación del CPU
  - Cambio de contexto
- Operaciones sobre Procesos
  - Creación de Procesos
  - Terminación de Procesos
- Comunicación entre Procesos (IPC)
- IPC en Sistemas de Memoria Compartida

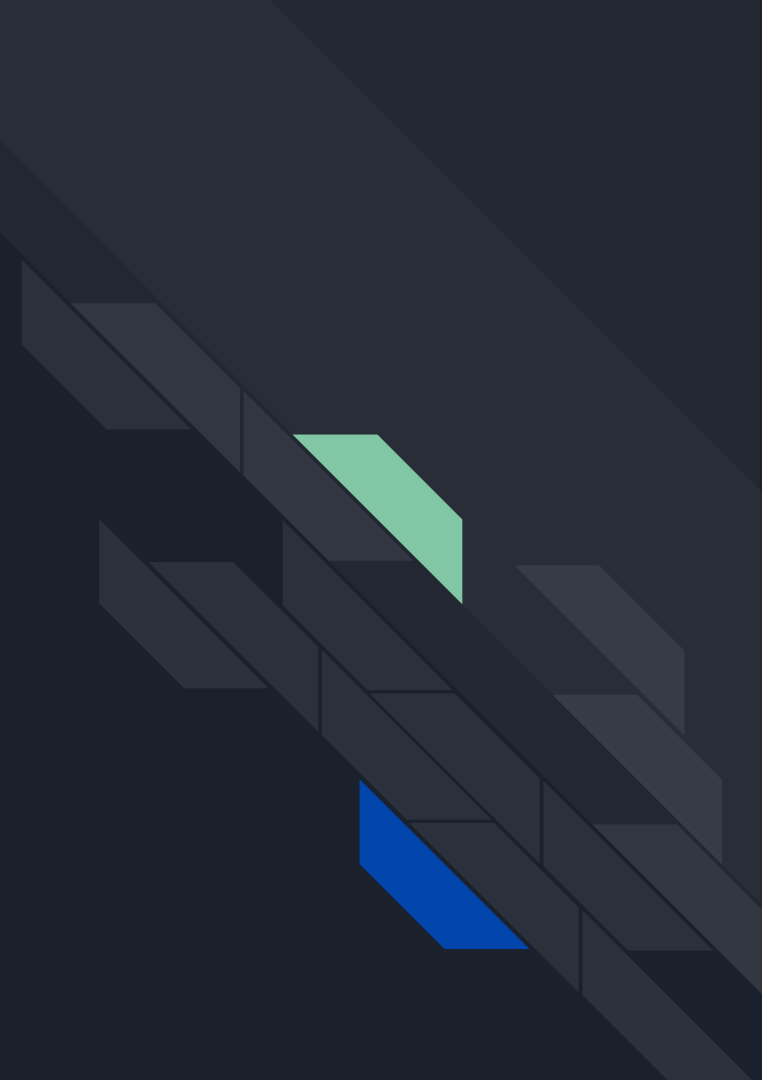


# TEMAS DE LA CLASE

- IPC en Sistemas de pasaje de Mensajes
  - Naming (cómo se referencia de un proceso a otro)
  - Sincronización
  - Buffering
- Examples of IPC Systems
  - POSIX Shared Memory
  - Pipes
- Comunicación en Sistemas Cliente-Servidor
  - Sockets
  - Llamadas a Procedimientos Remotos



# ADMINISTRACIÓN DE PROCESOS





# PROCESOS

Un proceso es un **programa en ejecución**, y es la **unidad de trabajo** en los sistemas operativos.

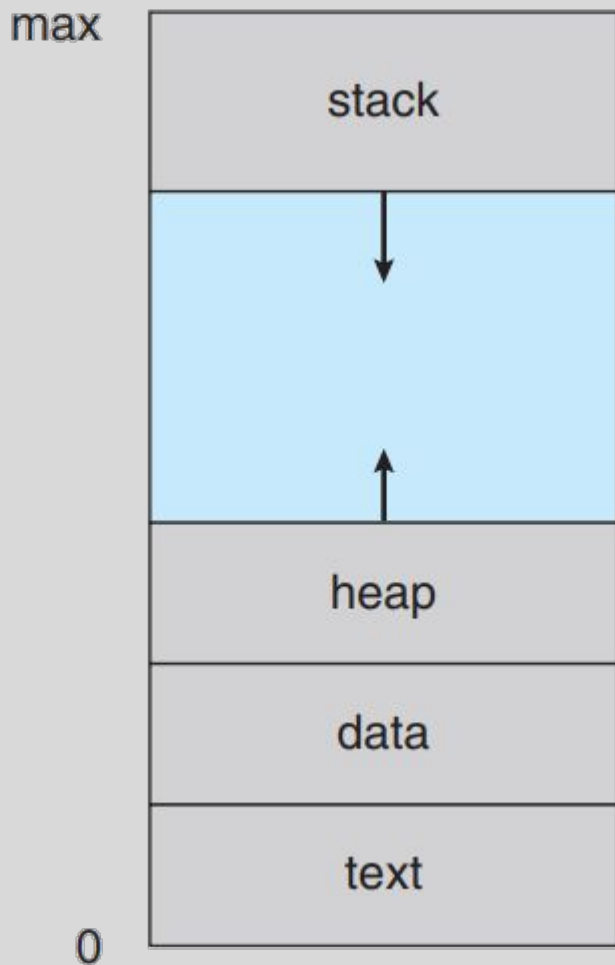
Un sistema está formado por una colección de procesos, algunos de los cuales:

- ejecutan código de usuario
- y otros código de sistema operativo

Potencialmente, todos estos procesos pueden ejecutarse **simultáneamente**.

Un programa en sí mismo **no** es un proceso. Un programa es una entidad **pasiva**, un archivo que contiene una lista de instrucciones almacenadas en disco. En cambio, un proceso es una entidad **activa**, con un PC que especifica la siguiente instrucción a ejecutar y un conjunto de recursos asociados (registros, archivos, memoria, etc).

Así se representa el **estado de la actividad actual** de un proceso, es el valor del PC y el contenido de los registros del procesador.



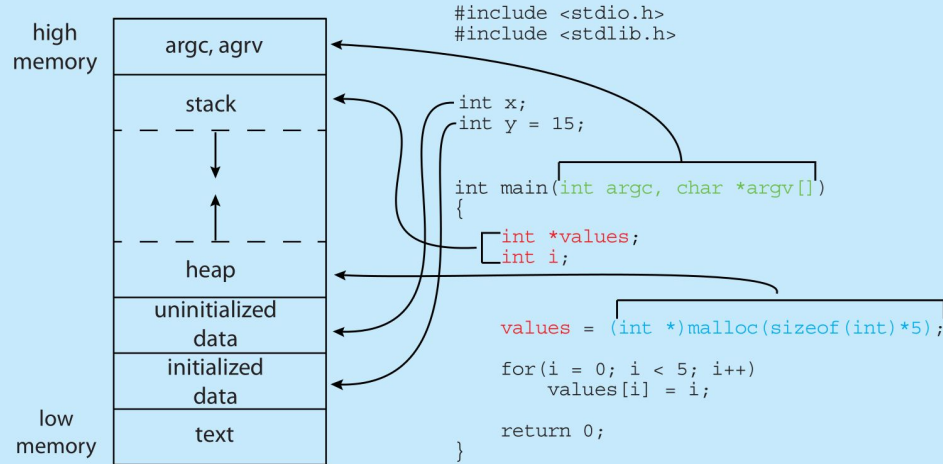
# EL PROCESO

El diagrama de un proceso en memoria se divide en secciones:

- *Text*: el código ejecutable.
- *Data*: variables globales.
- *Heap*: memoria asignada dinámicamente durante la ejecución del programa.
- *Stack*: almacenamiento temporal al invocar funciones (parámetros, direcciones de retorno, variables locales).

Notar las secciones de texto y datos son fijas, no cambian durante la ejecución del programa. En cambio el stack y el heap se pueden expandir y contraer dinámicamente durante la ejecución del programa.

## MEMORY LAYOUT OF A C PROGRAM



The GNU `size` command can be used to determine the size (in bytes) of some of these sections. Assuming the name of the executable file of the above C program is `memory`, the following is the output generated by entering the command `size memory`:

text	data	bss	dec	hex	filename
1158	284	8	1450	5aa	memory

The `data` field refers to uninitialized data, and `bss` refers to initialized data. (`bss` is a historical term referring to *block started by symbol*.) The `dec` and `hex` values are the sum of the three sections represented in decimal and hexadecimal, respectively.

## REPRESENTACIÓN DE LA MEMORIA DE UN PROGRAMA EN C

Esta imagen ilustra de manera más real la disposición de un programa en C en memoria. Notar las diferencias con la imagen anterior:

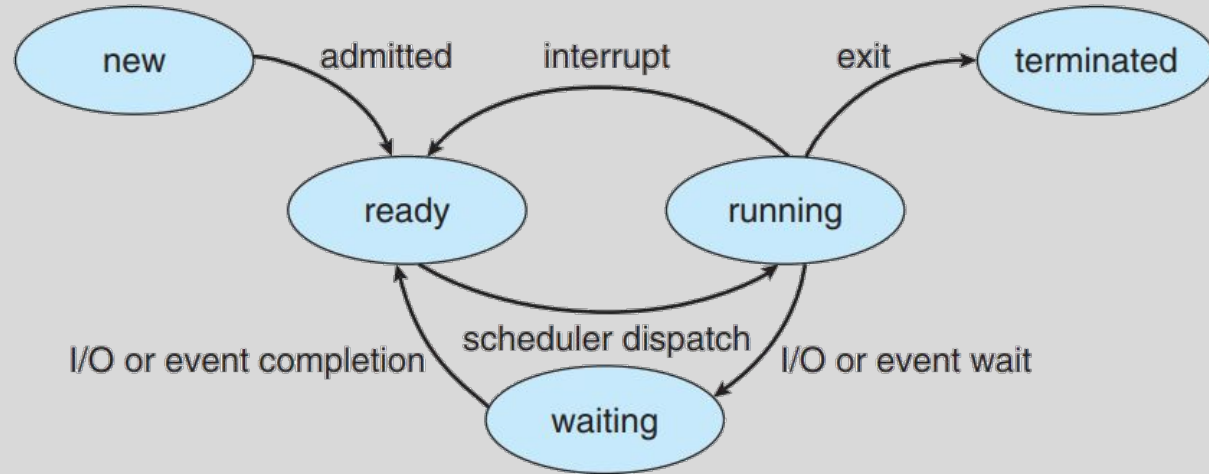
- La sección *data* (donde se almacenan las variables globales) se divide en (a) datos inicializados y (b) datos no inicializados.
- Además, existe una sección separada para los parámetros `argc` y `argv` pasados a la función `main()`.

# ESTADOS

Mientras se ejecuta, el proceso cambia de estado.

- **Nuevo:** el proceso es creado.
- **Ejecutando:** las instrucciones están siendo ejecutadas.
- **Esperando:** el proceso espera la ocurrencia de un evento.
- **Listo:** el proceso espera ser asignado a un procesador.
- **Terminado:** el proceso ha finalizado su ejecución.

Es importante notar que solo un proceso puede estar ejecutándose en un núcleo del procesador en un instante. Sin embargo, muchos procesos pueden estar listos y esperando.







## BLOQUE DE CONTROL (PCB)

Cada proceso es representado en el sistema operativo por un **Bloque de Control de Procesos** (*Process Control Block*). El cual contiene mucha información asociada a un proceso:

- Estado del Proceso (new, ready, etc.)
- PC
- Registros del CPU.
- Información sobre la Planificación del CPU.
- Información sobre la Administración de Memoria.
- Información contable (cantidad de CPU, pid, etc).
- Información del Estado E/S.

# PLANIFICACIÓN DE PROCESOS



# PLANIFICACIÓN DE PROCESOS

El objetivo de la multiprogramación (*multiprogramming*) es tener siempre un proceso en ejecución para maximizar el uso de la CPU. Y la cantidad de procesos actualmente en memoria se conoce como **grado de multiprogramación** (*degree of multiprogramming*).

El objetivo de *timesharing* es asignar un núcleo de CPU entre procesos con tanta frecuencia, que los usuarios puedan interactuar con cada programa mientras se ejecuta como si fuera el único.

Para lograr esto, el **planificador de procesos** selecciona un proceso disponible para la ejecución (*ready*) y lo asigna a un núcleo del CPU.

En general, la mayoría de los procesos pueden describirse como limitados por E/S o limitados por la CPU.

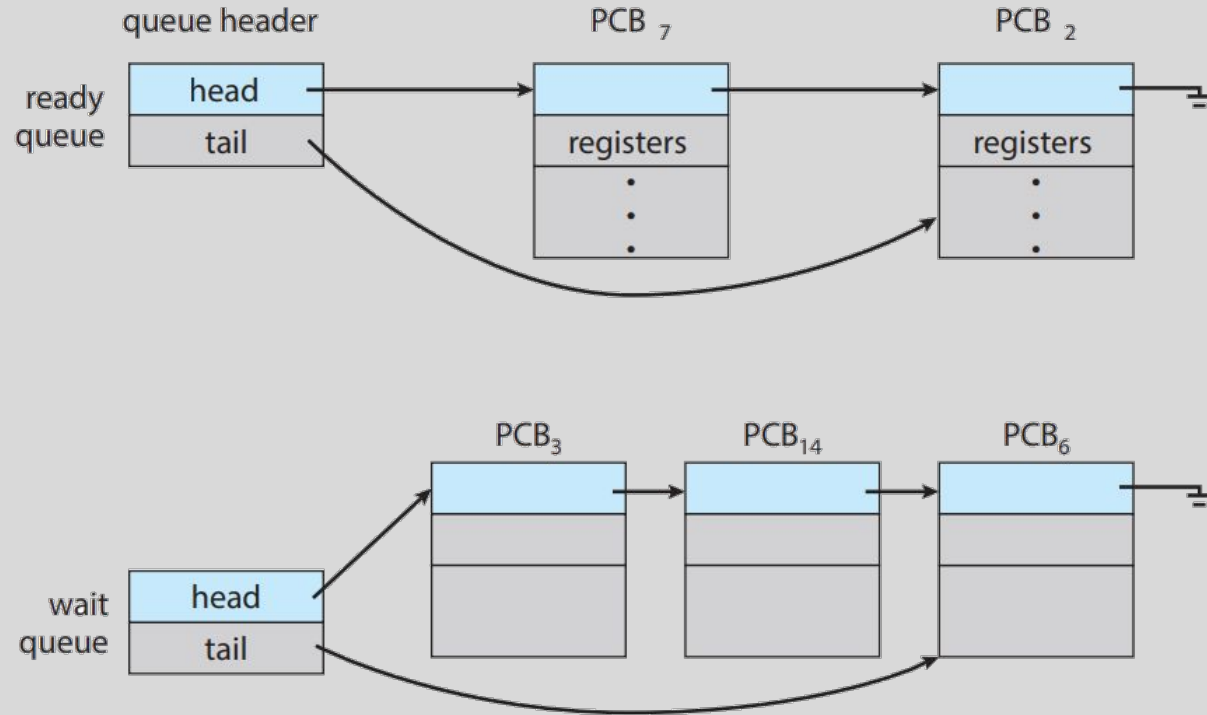
- **Limitado por E/S:** se dedica más tiempo a realizar E/S que a realizar cálculos.
- **Limitado por CPU:** genera pocas solicitudes de E/S y dedica más tiempo a realizar cálculos.



# COLAS DE PLANIFICACIÓN

Cuando un proceso entra al sistema, se suma a la **cola de listos** (*ready*), donde esperan la asignación en un núcleo de la CPU.

En general, esta cola se implementa como una lista enlazada; un *header* que mantiene una referencia a la primera PCB de la lista, y a su vez, cada PCB incluye una referencia que apunta a la siguiente PCB de la lista.

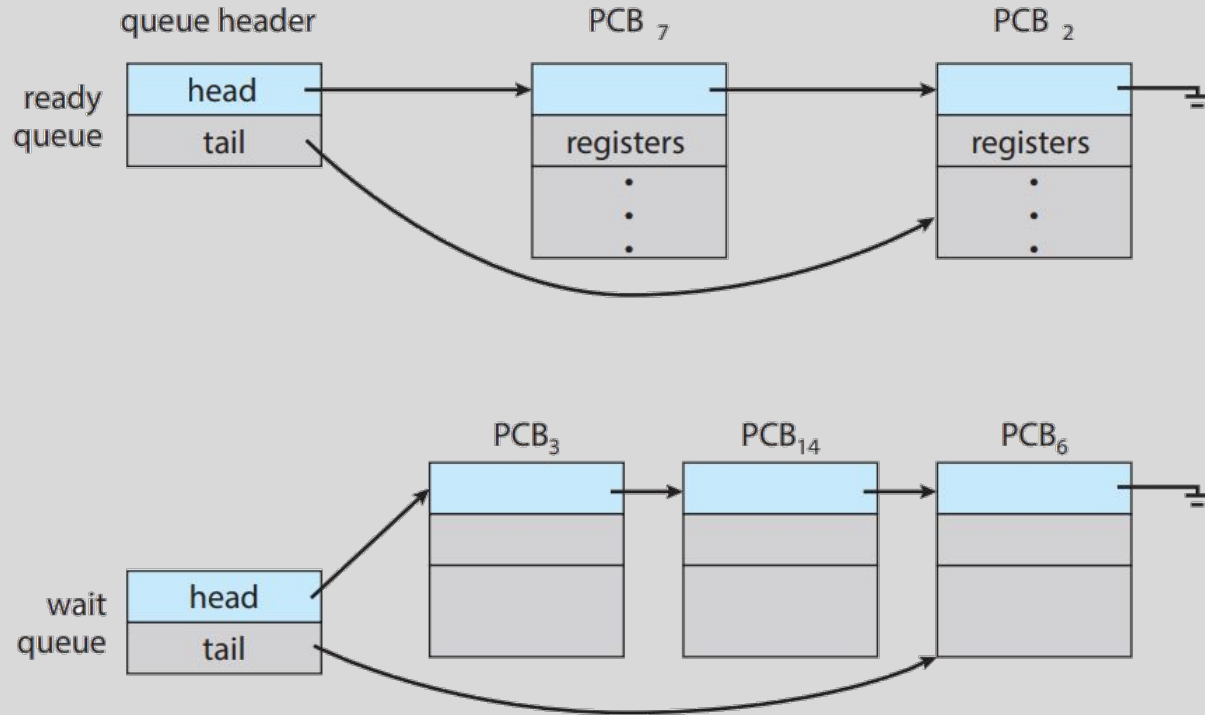


# COLAS DE PLANIFICACIÓN

El sistema también incluye otras colas de planificación.

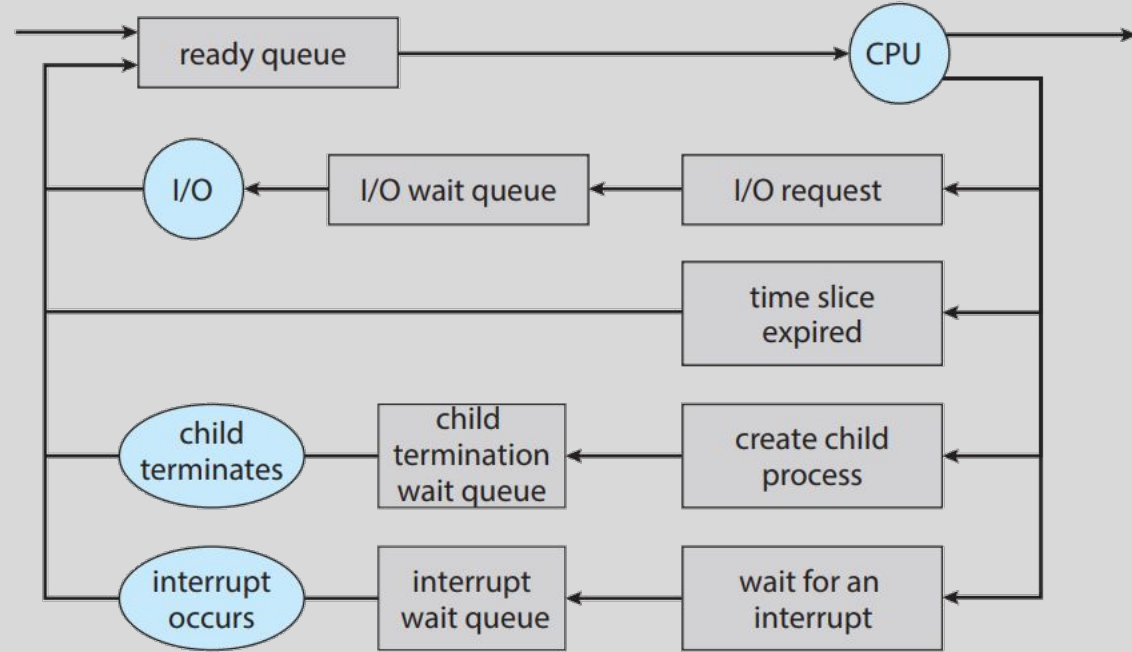
Por ejemplo, como hemos visto, los dispositivos periféricos funcionan mucho más lento que los procesadores, entonces si el proceso solicita una operación de E/S, tendrá que esperar a que la respuesta esté disponible.

Estos procesos que esperan un evento determinado, como la finalización de E/S, se colocan en una **cola de espera**.



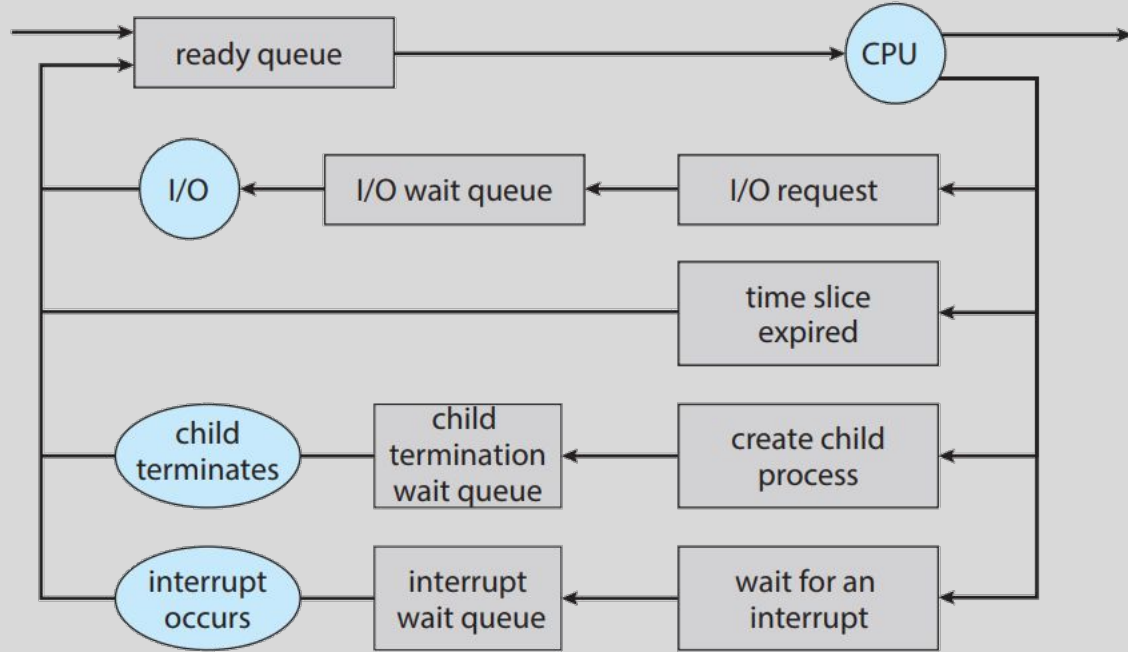
# COLAS DE PLANIFICACIÓN

Así, el ciclo de vida de un proceso podría ser: inicialmente se suma en la cola de listos y espera hasta que es seleccionado para su ejecución. Una vez asignado a un núcleo de CPU y se está ejecutando, puede ocurrir alguno de estos eventos:



# COLAS DE PLANIFICACIÓN

- El proceso puede emitir una solicitud de E/S y colocarse en una cola de espera de E/S.
- Puede crear un nuevo proceso hijo y luego colocarse en una cola de espera mientras espera su finalización.
- Puede ser eliminado forzosamente del núcleo, por resultado de una interrupción o por expiración del tiempo asignado, y volver a la cola de listos.
- O puede continuar este ciclo hasta que termina, momento en el que se elimina de todas las colas, y se liberan su PCB y sus recursos.



# PLANIFICACIÓN DE LA CPU

El planificador de la CPU debe seleccionar uno de los procesos en la cola de listos y asignarlo a un núcleo de la CPU. Esta selección ocurre con mucha frecuencia. Para describir la idea:

- Un proceso limitado por E/S puede ejecutarse pocos milisegundos antes de esperar por una solicitud de E/S.
- Y aunque un proceso limitado por CPU requerirá más tiempo de CPU, es poco probable que el planificador le otorgue un período prolongado. Seguramente el planificador esté diseñado para forzar la salida de la CPU del proceso y programar la ejecución de otro.

Por lo tanto, podríamos decir que el planificador de la CPU se ejecuta muchas veces en 1 segundo, al menos una vez cada 100 ms (aunque normalmente ocurre con mucha más frecuencia).

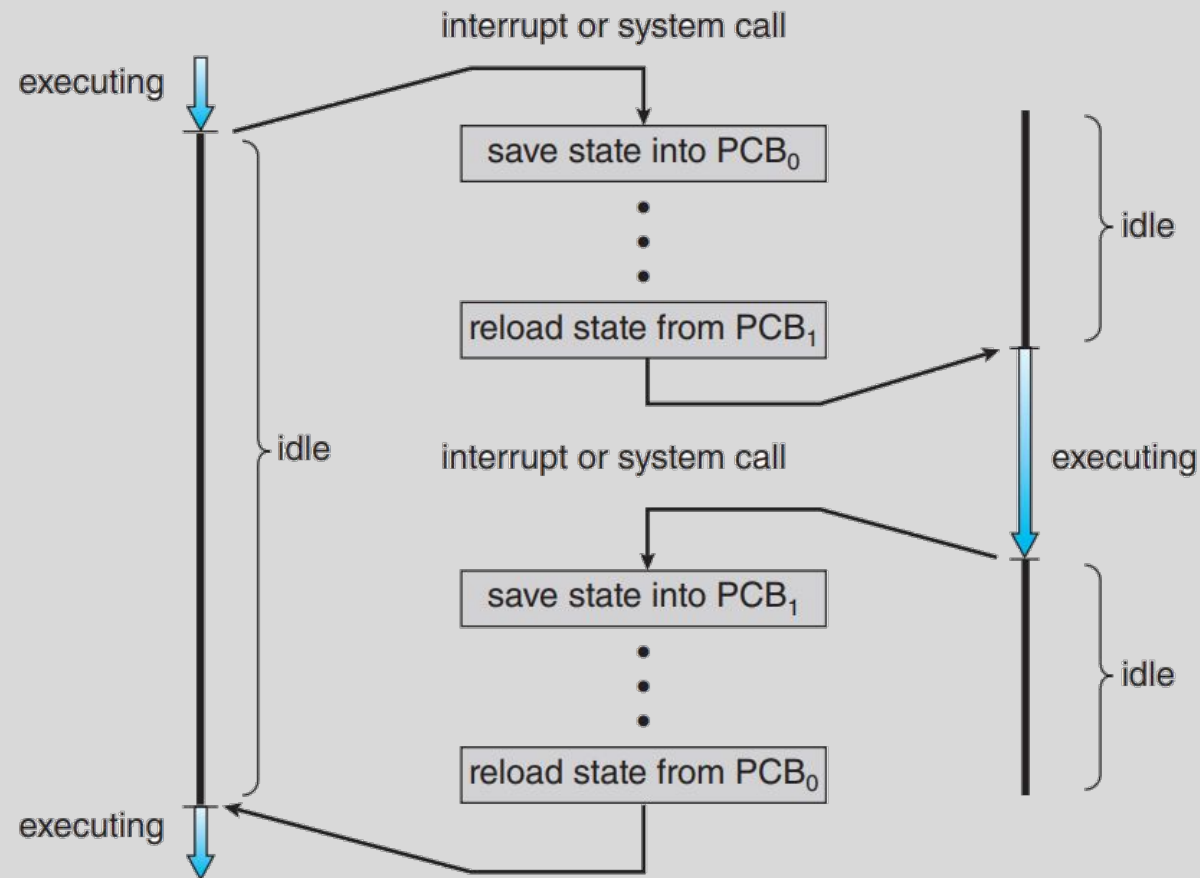
Lo que produce esta selección es un **cambio de contexto**.





process  $P_0$ 

operating system

process  $P_1$ 

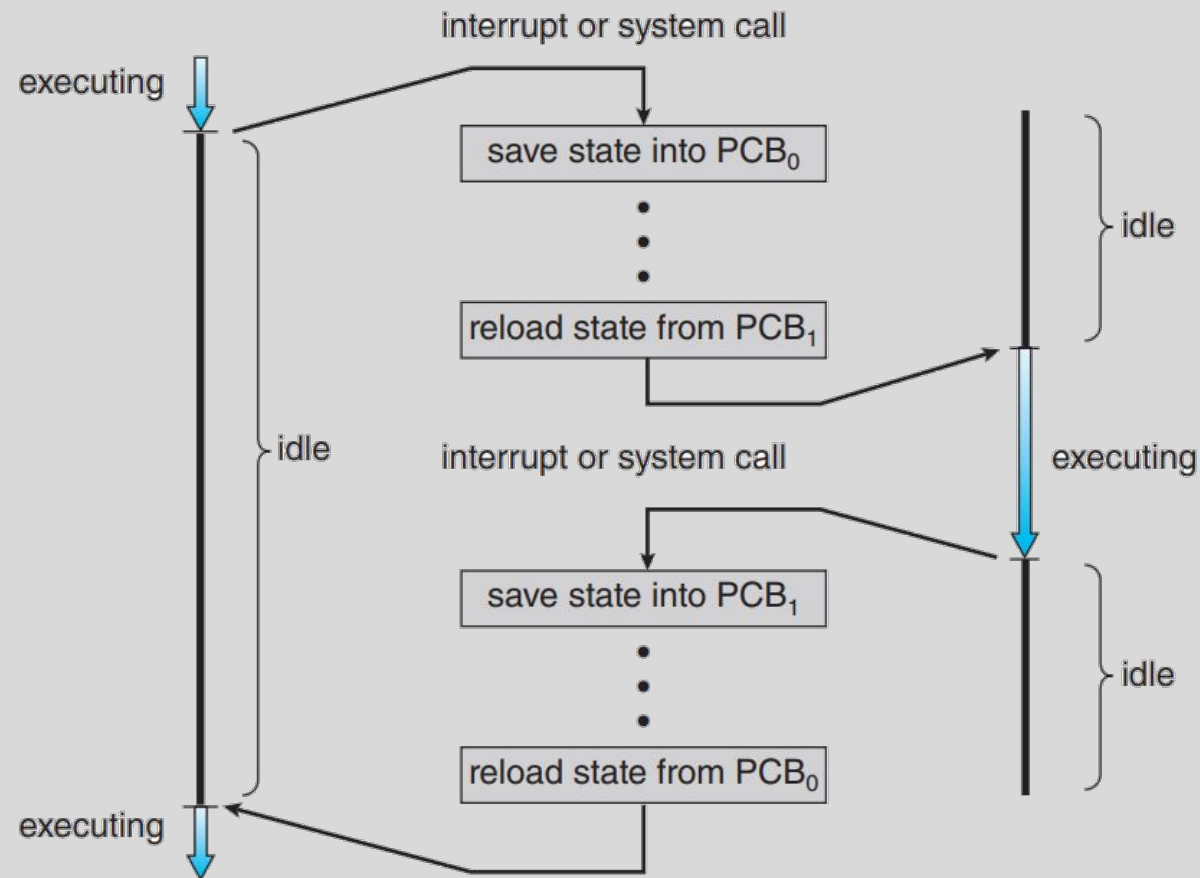
## CAMBIO DE CONTEXTO

Cuando se produce una interrupción, el sistema necesita **guardar el contexto actual** del proceso que se ejecuta en un núcleo de la CPU, para poder restaurarlo una vez finalizado su procesamiento (o sea, **suspender** y luego **reanudar**).

Este contexto se representa en el PCB del proceso. Generalmente, se realiza el **guardado** del estado actual del núcleo de la CPU, ya sea en modo kernel o de usuario, y cuando la interrupción termina, el estado es **restaurado** para poder reanudar las operaciones.

process  $P_0$ 

operating system

process  $P_1$ 

# CAMBIO DE CONTEXTO

Este procedimiento se denomina, **cambio de contexto**. Y cuando se produce, el kernel guarda el contexto del proceso anterior en su PCB, y carga el contexto guardado del nuevo proceso programado para ejecutarse.

El tiempo de cambio de contexto es pura **sobrecarga**, ya que el sistema no realiza ninguna **función útil** durante el cambio. Una velocidad típica es de varios microsegundos. Y dependen en gran medida de la compatibilidad del hardware.

# OPERACIONES SOBRE PROCESOS





# CREACIÓN DE PROCESOS

Durante su ejecución, un proceso puede crear varios procesos nuevos.

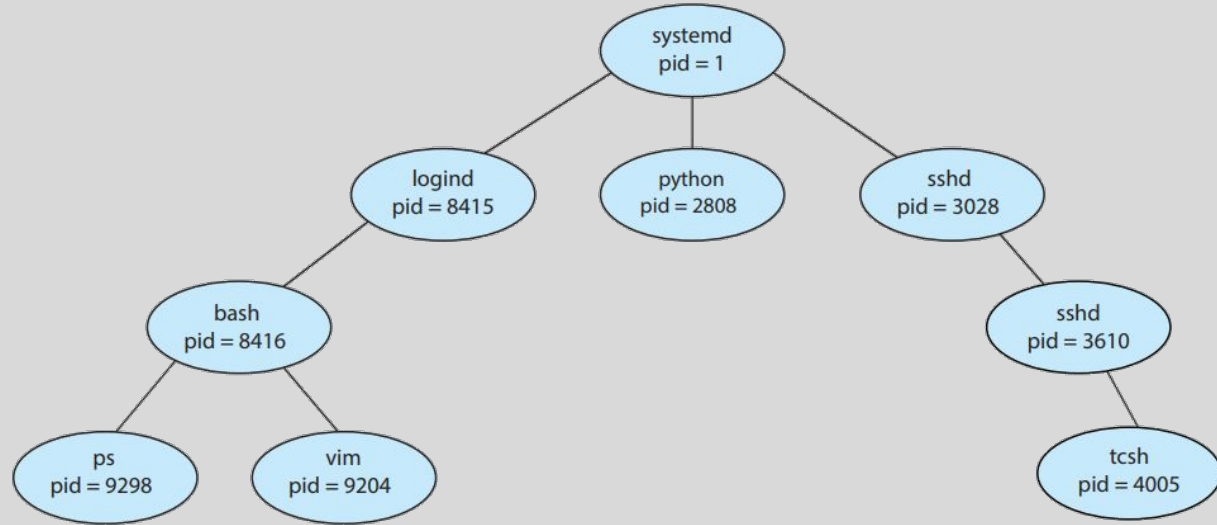
El proceso creador se denomina **proceso padre** y los nuevos procesos se denominan **procesos hijos**. Cada uno de estos nuevos procesos puede, a su vez, crear otros procesos, formando un **árbol de procesos**.

Los sistemas operativos (UNIX, Linux y Windows) identifican a los procesos según un **identificador de proceso único** (o PID), que suele ser un número entero. El PID proporciona un valor único para cada proceso del sistema y puede utilizarse como índice para acceder a sus diversos atributos dentro del kernel.

# CREACIÓN DE PROCESOS

El proceso **systemd** (con PID = 1, siempre) es el proceso raíz para todos los procesos, y es el primer proceso que se crea al arrancar el sistema.

Una vez arrancado el sistema, **systemd** crea otros procesos que proporcionan servicios adicionales, como un servidor web, de impresión, un servidor SSH, etc.





# CREACIÓN DE PROCESOS

Cuando un proceso crea un nuevo proceso hijo, este último necesitará ciertos recursos para realizar su tarea. El proceso hijo puede obtener sus recursos directamente del sistema operativo o estar limitado a un subconjunto de los recursos del proceso padre. Puede darse que el padre deba particionar sus recursos entre sus hijos o compartir algunos entre varios de ellos.

En términos de **ejecución**, cuando un proceso crea un nuevo proceso, existen dos posibilidades:

1. El proceso padre continúa ejecutándose simultáneamente con sus procesos hijos.
2. El proceso padre espera hasta que algunos o todos sus procesos hijos hayan terminado.

En términos de **espacio de memoria**, también existen dos posibilidades para el nuevo proceso:

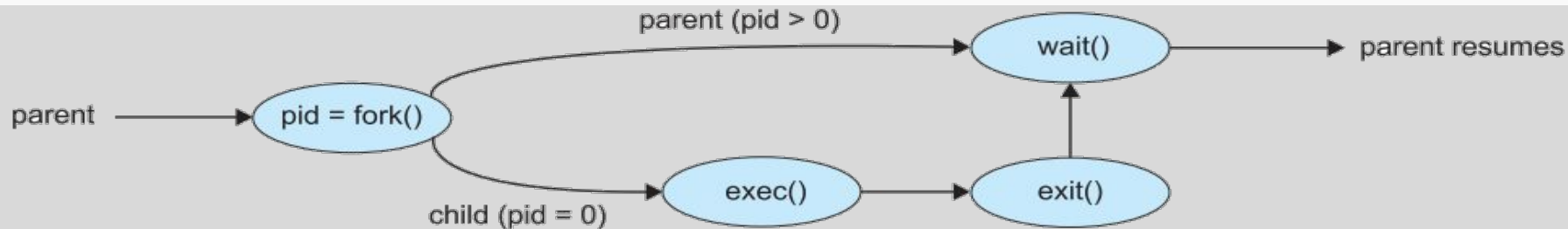
1. El proceso hijo es un duplicado del proceso padre (mismo programa y datos).
2. El proceso hijo tiene un nuevo programa cargado.

# CREACIÓN DE PROCESOS

Un nuevo proceso se crea mediante la *syscall* `fork()`. El nuevo proceso consiste en una copia del espacio de direcciones del proceso original. Este mecanismo permite que el proceso padre se comunique fácilmente con el proceso hijo. Dado que el hijo es una copia del padre, cada proceso tiene su propia **copia** de cualquier dato.

Ambos procesos continúan la ejecución en la instrucción posterior a `fork()`, con una diferencia:

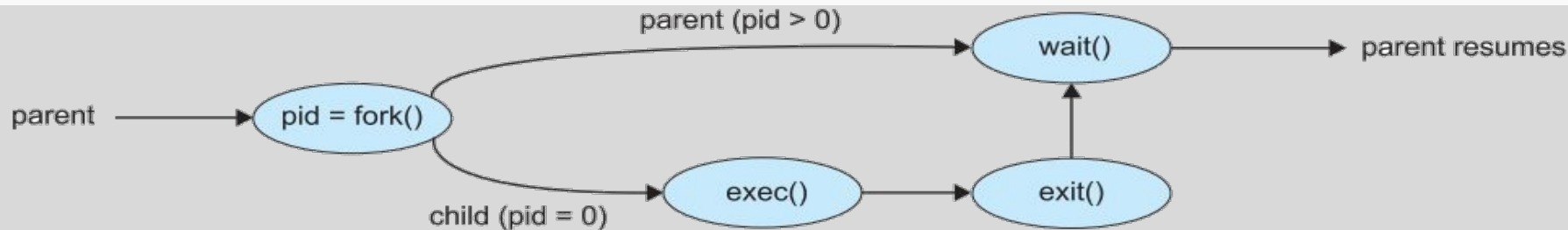
- el código de retorno de `fork()` es cero para el nuevo proceso (hijo)
- mientras que para el padre es positivo y distinto de cero. De hecho, `fork()` retorna el identificador de proceso hijo al padre.



# CREACIÓN DE PROCESOS

Es normal que tras una `syscall fork()`, uno de los dos procesos utilice la `syscall exec()` para reemplazar el espacio de memoria del proceso con un nuevo programa.

La `syscall exec()` carga un archivo binario en memoria (destruyendo la imagen de memoria del programa, una copia del padre) e inicia su ejecución. De esta manera, los dos procesos pueden comunicarse y luego seguir caminos separados. El padre puede entonces crear más hijos; o, puede ejecutar una `syscall wait()` para retirarse de la cola de procesos listos hasta que el hijo termine. Debido a que la llamada a `exec()` superpone el espacio de direcciones del proceso con un nuevo programa, `exec()` no devuelve el control a menos que ocurra un error.







# TERMINACIÓN DE PROCESOS

Un proceso termina cuando finaliza su última sentencia y solicita al SO que lo elimine mediante la `syscall exit()`. El proceso puede devolver un valor de estado a su proceso padre en espera (mediante la `syscall wait()`). El sistema operativo libera y recupera todos los recursos del proceso, la memoria física y virtual, los archivos abiertos y los búferes de E/S.

Un proceso padre puede terminar la ejecución de uno de sus hijos por diversas razones, por ejemplo:

- El hijo ha excedido el uso de algunos de los recursos que tenía asignados. (Para esto, el padre debe tener algún mecanismo para inspeccionar el estado de sus hijos).
- La tarea asignada al hijo ya no es necesaria.
- El padre está finalizando y el sistema operativo no permite que un hijo continúe si su proceso padre termina.



# TERMINACIÓN DE PROCESOS

Algunos sistemas no permiten que un hijo exista si su proceso padre ha terminado. Cuando un proceso termina (ya sea de forma normal o anormal), todos sus procesos hijos también deben terminar. Este fenómeno, es conocido como **terminación en cascada**, y normalmente lo inicia el sistema operativo.

Cuando un proceso termina, el sistema operativo libera sus recursos. Sin embargo, su registro en la tabla de procesos debe permanecer ahí hasta que el proceso principal invoque `wait()`, ya que la tabla de procesos contiene su estado de salida. Un proceso que ha terminado, pero cuyo proceso principal aún no ha invocado `wait()`, se conoce como proceso **zombi**. Todos los procesos pasan a este estado al terminar, pero generalmente solo existen como zombis brevemente. Una vez que el proceso principal invoca `wait()`, se liberan el identificador del proceso zombi y su registro en la tabla de procesos.



# COMUNICACIÓN ENTRE PROCESOS (IPC)



# COMUNICACIÓN ENTRE PROCESOS

Proceso **independiente**: no comparte datos con otros procesos que se ejecutan en el sistema.

Proceso **cooperativo** si puede afectar o ser afectado por los demás procesos que se ejecutan en el sistema. Es decir, comparten información con otros procesos.

Razones para proporcionar un entorno que permita la cooperación entre procesos:

- **Intercambio de información.** Los sistemas deben proporcionar un entorno que permita el acceso simultáneo a la información.
- **Aceleración computacional.** Para que una tarea específica se ejecute más rápido, es posible dividirla en subtareas, donde cada una de ellas se ejecutará en paralelo con las demás. Esto es posible en entornos *multicore*.
- **Modularidad.** Se puede construir el sistema de forma modular, dividiendo las funciones del sistema en procesos o hilos separados.



# COMUNICACIÓN ENTRE PROCESOS

Para que los procesos sean cooperativos, es necesario un mecanismo de comunicación entre ellos (IPC) que les permita intercambiar datos entre sí.

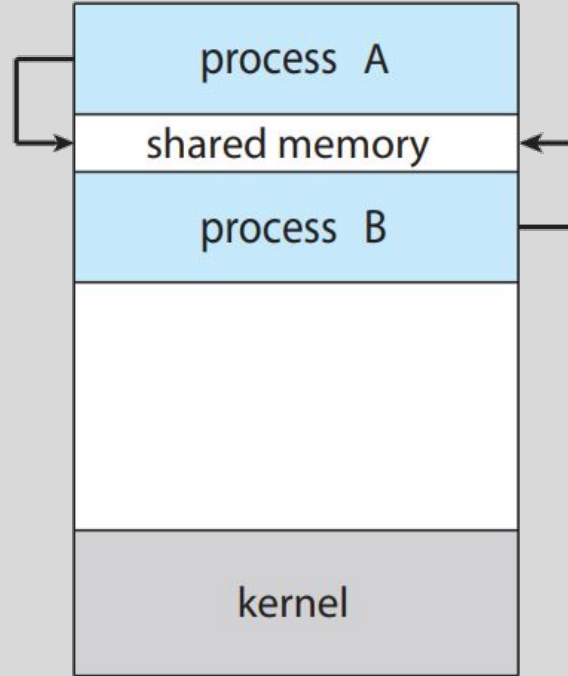
Existen dos modelos fundamentales de comunicación entre procesos:

- **Memoria compartida:** los procesos cooperativos establecen una región de memoria compartida. De esta forma, pueden intercambiar información leyendo y escribiendo datos en la región de memoria compartida.
- **Pasaje de mensajes:** la comunicación se realiza mediante el intercambio de mensajes entre los procesos cooperativos.

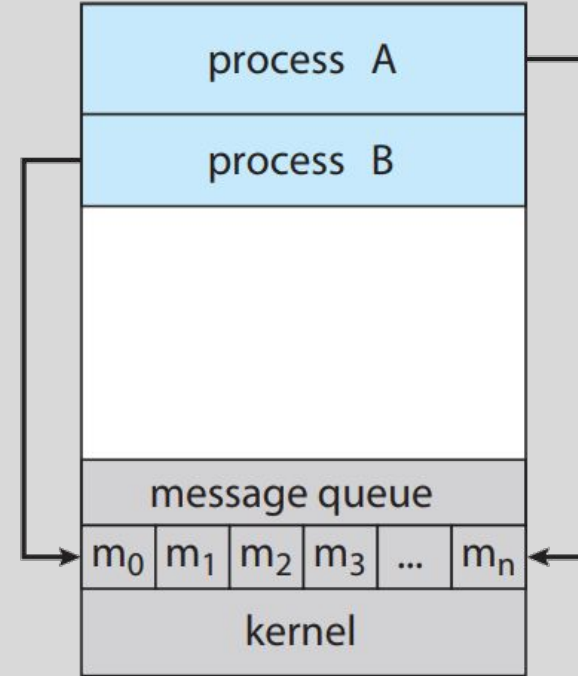
# COMUNICACIÓN ENTRE PROCESOS

La **memoria compartida** puede ser más rápida que el pasaje de mensajes, ya que los sistemas de pasaje de mensajes suelen implementarse mediante *syscalls* y, por esto, requieren la intervención del kernel, lo cual requiere más tiempo.

En los sistemas de **memoria compartida**, las *syscalls* solo se requieren para establecer las regiones de memoria compartida. Una vez establecida, todos los accesos se tratan como accesos rutinarios a la memoria y no se requiere la intervención del kernel.



Shared memory.



Message passing.



# IPC EN SISTEMAS DE MEMORIA COMPARTIDA



# IPC EN SISTEMAS DE MEMORIA COMPARTIDA

La memoria compartida requiere que los procesos involucrados establezcan una región de memoria compartida. Normalmente, esta región reside en el espacio de direcciones del proceso que crea el segmento. Así, otros procesos que deseen comunicarse utilizando este segmento deben adjuntarlo a su espacio de direcciones.

Vale mencionar que normalmente, el sistema operativo intenta impedir que un proceso acceda a la memoria de otro. Por este motivo, la memoria compartida requiere que los procesos acuerden eliminar esta restricción. De esta forma, pueden intercambiar información leyendo y escribiendo datos en las áreas compartidas.

La forma y la ubicación de los datos son determinadas por los procesos y no están bajo el control del sistema operativo. También son responsables de garantizar que no escriban en la misma ubicación simultáneamente.





# IPC EN SISTEMAS DE MEMORIA COMPARTIDA

Para ilustrar estos conceptos de procesos cooperativos, generalmente se utiliza el problema del productor-consumidor. Donde un proceso productor genera (produce) información que es utilizada por un proceso consumidor. Una solución a este problema utiliza memoria compartida.

Para que los procesos productor y consumidor se ejecuten simultáneamente, es necesario disponer de un buffer (lugar donde almacenar) de elementos que el productor pueda llenar y el consumidor pueda vaciar. Este búfer residirá en la región de memoria compartida por ambos procesos. Así un productor puede producir un elemento mientras el consumidor consume otro. Además, el productor y el consumidor deben estar sincronizados para que el consumidor no intente consumir un elemento que aún no se ha producido.

Para esto, se pueden usar dos tipos de búfer:

- Un buffer ilimitado, donde no se impone un límite práctico a su tamaño. El consumidor puede tener que esperar nuevos elementos, pero el productor siempre puede producirlos.
- Un buffer acotado tiene un tamaño fijo. En este caso, el consumidor debe esperar si el búfer está vacío y el productor debe esperar si está lleno.



# IPC EN SISTEMAS CON PASAJE DE MENSAJES



# IPC EN SISTEMAS CON PASAJE DE MENSAJES

El pasaje de mensajes proporciona un mecanismo que permite a los procesos comunicarse y sincronizar sus acciones sin compartir el mismo espacio de direcciones. Es particularmente útil en entornos distribuidos, donde los procesos que se comunican pueden residir en diferentes computadoras conectadas por una red.

Un mecanismo de paso de mensajes proporciona al menos dos operaciones:

- `send(message)`
- `receive(message)`



# IPC EN SISTEMAS CON PASAJE DE MENSAJES

Así, para que los procesos P y Q puedan comunicarse (enviar y recibir mensajes entre sí), debe existir un **enlace de comunicación** (*link*) entre ellos.

Veremos varios métodos para implementar lógicamente un *link* y las operaciones `send()`/`receive()`:

- Comunicación directa o indirecta
- Comunicación síncrona o asíncrona
- Almacenamiento en *buffer*



# IPC EN SISTEMAS CON PASAJE DE MENSAJES - *NAMING*

En **comunicación directa**, cada proceso que desee comunicarse debe nombrar explícitamente al destinatario o remitente de la comunicación. En este esquema, las operaciones `send()` y `receive()` se definen como:

- `send(P, message)`: Envía un mensaje al proceso P.
- `receive(Q, message)`: Recibe un mensaje del proceso Q.

Propiedades del **enlace de comunicación** en este esquema:

- Se establece automáticamente un enlace entre cada par de procesos que deseen comunicarse. Los procesos sólo necesitan conocer la identidad del otro para comunicarse.
- Un enlace está asociado con exactamente dos procesos.
- Entre cada par de procesos, existe exactamente un enlace.



# IPC EN SISTEMAS CON PASAJE DE MENSAJES - *NAMING*

En **comunicación indirecta**, los mensajes se envían y reciben desde buzones o puertos. Un buzón puede considerarse, de forma abstracta, como un objeto en el que los procesos pueden colocar mensajes y del que pueden eliminarlos, y cada buzón tiene una identificación única. Las operaciones `send()` y `receive()` se definen de la siguiente manera:

- `send(A, message)`: Envía un mensaje al buzón A.
- `receive(A, message)`: Recibe un mensaje del buzón A.

Propiedades del **enlace de comunicación** en este esquema:

- Un *link* se establece entre un par de procesos solo si ambos miembros comparten un buzón.
- Un *link* puede estar asociado a más de dos procesos.
- Entre los procesos que se comunican, pueden existir varios *links* diferentes, cada uno correspondiente a un buzón.



# IPC EN SISTEMAS CON PASAJE DE MENSAJES - *SYNCHRONIZATION*

Como vimos, la comunicación entre procesos se realiza mediante llamadas a las operaciones `send()` y `receive()`. Pero existen diferentes opciones de diseño para implementar cada operación. Por eso, el pasaje de mensajes entre procesos puede ser **bloqueante** o **no bloqueante**, también conocido como **síncrono** y **asíncrono**.

- **Envío bloqueante:** El proceso emisor se bloquea hasta que el receptor o el buzón reciben el mensaje.
- **Envío no bloqueante:** El proceso emisor envía el mensaje y continúa la operación.
- **Recepción bloqueante:** El receptor se bloquea hasta que haya un mensaje disponible.
- **Recepción no bloqueante.** El receptor recupera un mensaje válido o nulo.



# IPC EN SISTEMAS CON PASAJE DE MENSAJES - *BUFFERING*

Ya sea en comunicación directa o indirecta, los mensajes intercambiados por los procesos residen en una cola temporal. Básicamente, estas colas se pueden implementar de tres maneras:

- **Capacidad cero:** No existe cola de almacenamiento, el enlace no puede tener ningún mensaje esperando. En este caso, el emisor debe bloquearse hasta que el receptor reciba el mensaje.
- **Capacidad limitada:** La cola tiene un almacenamiento finito de  $N$ , es decir, puede almacenar como máximo  $N$  mensajes. Así, cuando se envía un nuevo mensaje, si el almacenamiento de la cola no está lleno, el emisor deja el mensaje y puede continuar su ejecución sin esperar. Pero como la capacidad del enlace es finita, si el almacenamiento está lleno, el emisor debe bloquearse hasta que haya espacio disponible.
- **Capacidad ilimitada.** El almacenamiento de la cola es potencialmente infinito, es decir, puede almacenar cualquier número de mensajes. En este caso, el emisor nunca se bloquea.





# COMUNICACIÓN EN SISTEMAS CLIENTE-SERVIDOR



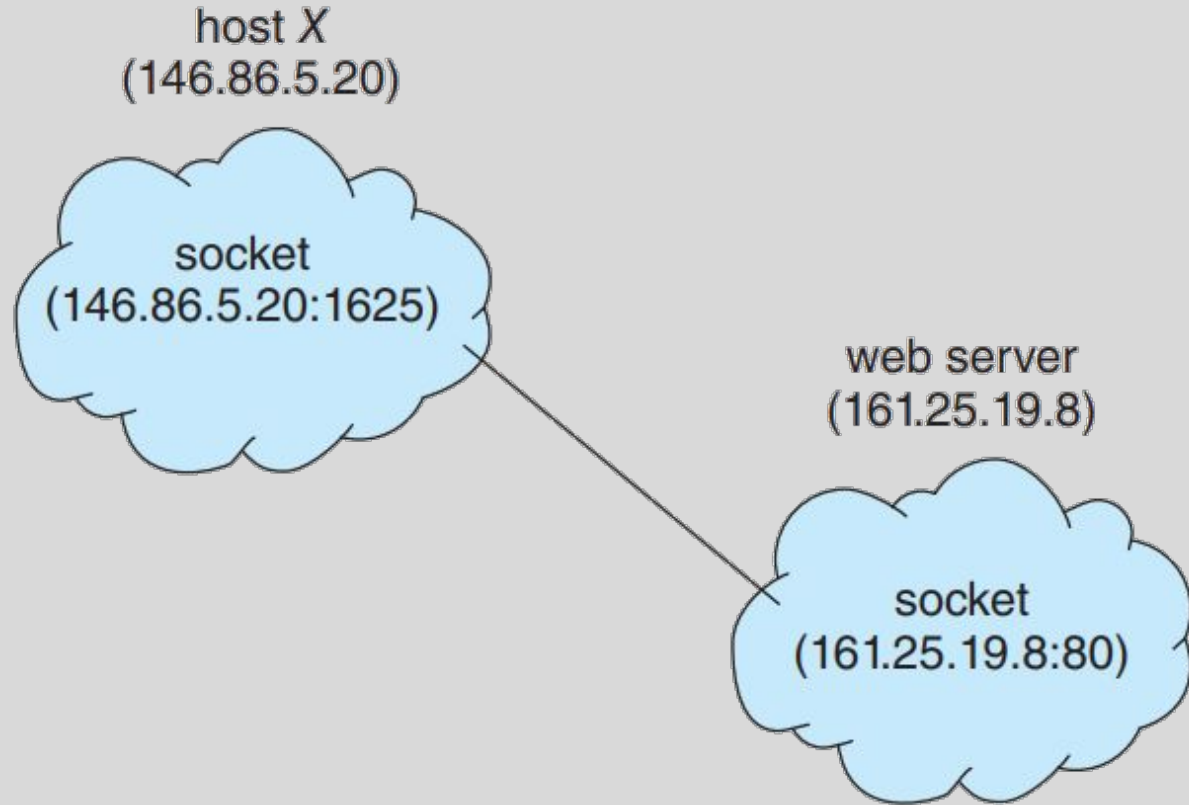
# SOCKETS

Un **socket** se define como un *endpoint* de comunicación. Dos procesos que se comunican a través de una red utilizan un par de sockets, uno para cada proceso. Un **socket** se identifica mediante una dirección IP concatenada con el número de un puerto, y en general, se utilizan en una arquitectura cliente-servidor.

En esta arquitectura, el servidor espera las solicitudes entrantes del cliente escuchando en un puerto específico. Una vez recibida la solicitud, acepta una conexión del *socket* del cliente. Los servidores que implementan servicios específicos (como SSH, FTP y HTTP) escuchan en puertos conocidos (un servidor SSH escucha en el puerto 22; un servidor FTP en el 21; y un servidor web o HTTP, en el 80, HTTPS en el 443). Todos los puertos inferiores a 1024 se consideran conocidos y se utilizan para implementar servicios estándar.

# SOCKETS

Cuando un proceso cliente inicia una solicitud de conexión, su *host* le asigna un puerto. Este puerto tiene un número arbitrario mayor que 1024. Por ejemplo, si un cliente en el host X con la dirección IP 146.86.5.20 desea establecer una conexión con un servidor web en la dirección 161.25.19.8, se le puede asignar al host X el puerto 1625. La conexión constará de un par de sockets: (146.86.5.20:1625) en el host X y (161.25.19.8:80) en el servidor web.

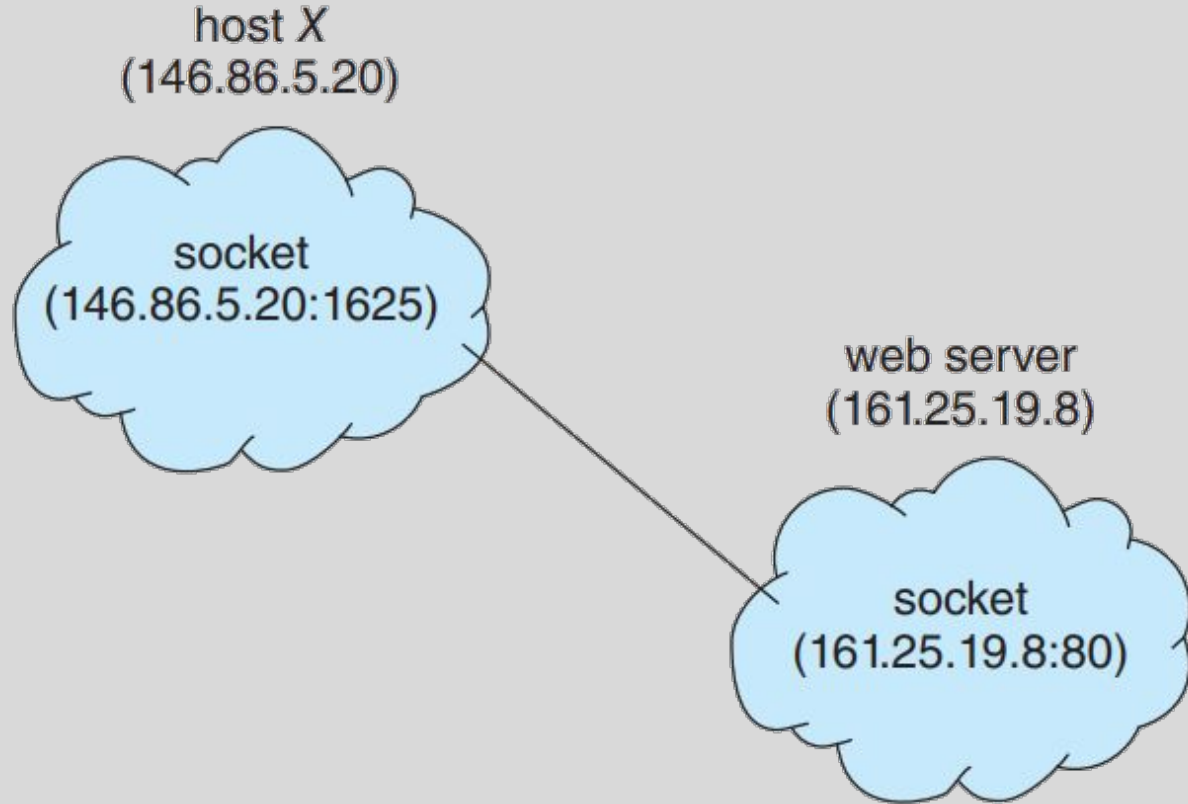


Communication using sockets.

# SOCKETS

De esta manera, los paquetes que viajan entre los hosts se entregan al proceso correspondiente según el número de puerto de destino.

Por esto, todas las conexiones deben ser únicas. Esto es, si otro proceso, también en el host X, deseara establecer otra conexión con el mismo servidor web, se le asignaría un número de puerto mayor que 1024 y distinto de 1625. Esto garantiza que todas las conexiones tengan de un par de sockets único.



Communication using sockets.

# Muchas Gracias

Jeremías Fassi

Javier E. Kinter

