

# Taller de IPC

## Sistemas Operativos

### 2do Cuatrimestre de 2016

Christian Heitman



- Conceptos básicos de IPC.
- Comunicación vía pipes.
- Comunicación vía sockets.
- ¡Taller!

Un proceso en un sistema operativo se dice que es:

**Independiente** si no se relaciona con otros procesos.

**Cooperativo** si comparte datos con otros procesos.

Podemos estar interesados en compartir datos si queremos:

- que varios procesos puedan acceder a un mismo archivo o recurso.
- aprovechar varias CPUs para paralelizar cálculos.
- modularizar un proceso y separar sus funcionalidades.

# IPC vía Memoria Compartida

Una opción para dos procesos que se comunican es compartir una región de memoria.

- Un proceso crea una región de memoria compartida en su heap asignándole ciertos permisos.
- Otros procesos se asocian a dicha región.
- Todos pueden usar dicha región según los permisos establecidos.

# IPC vía Memoria Compartida

Una opción para dos procesos que se comunican es compartir una región de memoria.

- Un proceso crea una región de memoria compartida en su heap asignándole ciertos permisos.
- Otros procesos se asocian a dicha región.
- Todos pueden usar dicha región según los permisos establecidos.

## Shared Memory APIs

```
int    shm_open(const char *name, int oflag, mode_t mode);
int    shm_unlink(const char *name);

int    shmget(key_t key, size_t size, int shmflg);
void *shmat(int shmid, const void *shmaddr, int shmflg);
int    shmdt(const void *shmaddr);
```

Otra opción para que dos procesos se comuniquen es el pasaje de mensajes entre ellos.

- Un proceso **P** envía un mensaje a otro proceso **Q** (**send**).
- El proceso **Q** recibe dicho mensaje (**receive**).

## Bloqueante:

- Un proceso se bloquea al enviar un mensaje, se desbloquea cuando ese mensaje es recibido.
- Un proceso se bloquea al intentar recibir un mensaje, se mantiene bloqueado mientras el buzón esté vacío.

## Bloqueante:

- Un proceso se bloquea al enviar un mensaje, se desbloquea cuando ese mensaje es recibido.
- Un proceso se bloquea al intentar recibir un mensaje, se mantiene bloqueado mientras el buzón esté vacío.

## No Bloqueante:

- Un proceso envía el mensaje y sigue ejecutando, sin importar si este es recibido o no.
- Un proceso intenta recibir un mensaje y lo obtiene, si es que hay alguno esperando en el buzón, o obtiene un mensaje nulo en caso contrario.



# Memoria Compartida vs. Pasaje de Mensajes

- Pasar mensajes es útil para pequeños datos, pero es lento porque requiere una llamada al sistema para cada mensaje.
- Memoria compartida es más veloz ya que requiere llamadas al sistema sólo para la inicialización.

# Memoria Compartida vs. Pasaje de Mensajes

- Pasar mensajes es útil para pequeños datos, pero es lento porque requiere una llamada al sistema para cada mensaje.
- Memoria compartida es más veloz ya que requiere llamadas al sistema sólo para la inicialización.
- El paso de mensajes se puede utilizar para comunicar procesos que residen en distintas computadoras.
- La memoria compartida requiere que los procesos estén en la misma computadora.

# Comunicación vía pipes

Recordemos, **pipes**, escritos como “|”.

Por ejemplo, qué sucede si escribimos en **bash**:

```
echo -n "sistemas es lo más" | wc -c
```

# Comunicación vía pipes

Recordemos, **pipes**, escritos como “|”.

Por ejemplo, qué sucede si escribimos en **bash**:

```
echo -n "sistemas es lo más" | wc -c
```

1. Se llama a `echo -n`, un programa que escribe su parámetro por **stdout**.
2. Se llama a `wc -c`, un programa que cuenta cuántos caracteres entran por **stdin**.
3. Se conecta el **stdout** de `echo` con el **stdin** de `wc -c`.

¿Cuál es el resultado?

# Comunicación vía pipes

Recordemos, **pipes**, escritos como “|”.

Por ejemplo, qué sucede si escribimos en **bash**:

```
echo -n "sistemas es lo más" | wc -c
```

1. Se llama a `echo -n`, un programa que escribe su parámetro por **stdout**.
2. Se llama a `wc -c`, un programa que cuenta cuántos caracteres entran por **stdin**.
3. Se conecta el **stdout** de `echo` con el **stdin** de `wc -c`.

¿Cuál es el resultado? **19.**

# Demo pipe

# Comunicación vía pipes

Para crear un pipe usamos:

```
int pipe(int pipefd[2]);
```

Luego de ejecutar pipe, tenemos:

- En pipefd[0] un **file descriptor** que apunta al extremo del pipe en el cual se **lee**.
- En pipefd[1] otro **file descriptor** que apunta al extremo del pipe en el cual se **escribe**.

Se retorna 0 si el llamado fue exitoso, -1 en caso contrario.

# Comunicación vía pipes

Para leer de un **file descriptor** usamos:

```
ssize_t read(int fd, void *buf, size_t count);
```

- **fd** **file descriptor**.
- **buf** puntero al **buffer** donde almacenar lo leído.
- **count** cantidad máxima de bytes a leer.

Se retorna la cantidad de bytes leídos, -1 en caso de error.



# Comunicación vía pipes

Para leer de un **file descriptor** usamos:

```
ssize_t read(int fd, void *buf, size_t count);
```

- **fd** **file descriptor**.
- **buf** puntero al **buffer** donde almacenar lo leído.
- **count** cantidad máxima de bytes a leer.

Se retorna la cantidad de bytes leídos, -1 en caso de error.

`read` es bloqueante, aunque pueden configurarse ciertos **flags** para que no lo sea.

Los **pipes** requieren que los procesos que se comunican tengan un ancestro en común que configure ambos extremos de la conexión.

**¿Qué pasa si los procesos no se conocen entre sí?**

Los **pipes** requieren que los procesos que se comunican tengan un ancestro en común que configure ambos extremos de la conexión.

**¿Qué pasa si los procesos no se conocen entre sí?**

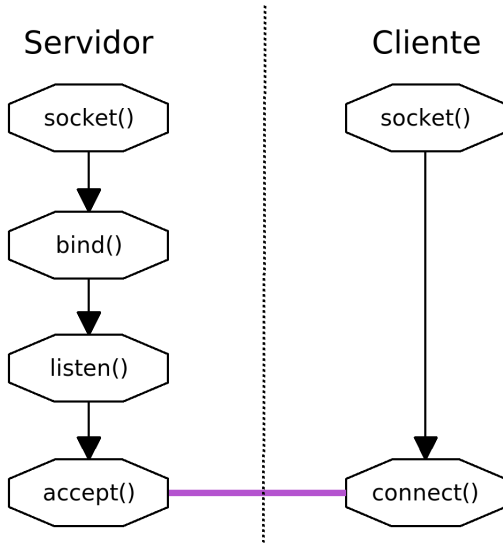
## Sockets

- Un **socket** es el extremo de una conexión y tiene asociado un nombre.
- Dos procesos que se quieren comunicar entre sí se ponen de acuerdo en dicho nombre.

- Los **sockets** pueden trabajar en distintos dominios.
- En el dominio UNIX las direcciones son un nombre de archivo.
- Dos procesos que se ponen de acuerdo en tal nombre de archivo se pueden comunicar.

# Demo socket

# Estableciendo la Comunicación con sockets



# Estableciendo sockets: Servidor

- 1 `int socket(int domain, int type, int protocol);`  
Crear un nuevo **socket**.
- 2 `int bind(int fd, sockaddr* a, socklen_t len);`  
Conectar un **socket** a una cierta dirección.
- 3 `int listen(int fd, int backlog);`  
Esperar conexiones entrantes en un **socket**.
- 4 `int accept(int fd, sockaddr* a, socklen_t* len);`  
Aceptar la próxima conexión en espera en un **socket**.

# Estableciendo sockets: Cliente

- 1 `int socket(int domain, int type, int protocol);`  
Crear un nuevo **socket**.
- 2 `int connect(int fd, sockaddr* a, socklen_t* len);`  
Conectarse a un **socket** remoto que debe estar escuchando.



# Comunicación vía sockets: Mensajes

Una vez que un cliente solicita conexión y esta es aceptada por el servidor puede comenzar el intercambio de mensajes con:

```
ssize_t send(int s, void *buf, size_t len, int flags);
```

```
ssize_t recv(int s, void *buf, size_t len, int flags);
```

# Comunicación vía sockets: ¿Bloqueante?

`accept()`, `send()`, `recv()` se bloquean hasta que la otra parte cumple con su parte.

# Comunicación vía sockets: ¿Bloqueante?

`accept()`, `send()`, `recv()` se bloquean hasta que la otra parte cumple con su parte.

¿Cómo hacemos si tenemos un servidor que se comunica con dos clientes a la vez?

## Demo

Ahora el servidor muestra por pantalla lo que **dos** clientes le indican.

# Comunicación vía sockets: ¿Bloqueante?

Si usamos llamadas bloqueantes podemos esperar indefinidamente por el primer cliente mientras el segundo tiene mucho que enviarnos y nunca se lo pedimos.

# Comunicación vía sockets: ¿Bloqueante?

Si usamos llamadas bloqueantes podemos esperar indefinidamente por el primer cliente mientras el segundo tiene mucho que enviarnos y nunca se lo pedimos.

Soluciones:

- 1 Llamadas no bloqueantes y espera activa.
- 2 Usar la llamada al sistema `select()`.
- 3 Utilizar un proceso (o thread) para atender cada cliente.  
(*No la veremos en esta clase.*)

# Solución 1: Llamadas No Bloqueantes

Demo: espera activa y llamadas no bloqueantes.

# Solución 1: Llamadas No Bloqueantes

Demo: espera activa y llamadas no bloqueantes.

- Se usa la llamada `fcntl()` para indicar que los sockets del servidor sean no bloqueantes.
- Al intentar recibir mensajes, si se devuelve `-1` tenemos que ver si es por un error verdadero o porque no había nada en espera de ser recibido. Usamos la variable `errno` para discernir esto.

# Solución 1: Llamadas No Bloqueantes

Demo: espera activa y llamadas no bloqueantes.

- Se usa la llamada `fcntl()` para indicar que los sockets del servidor sean no bloqueantes.
- Al intentar recibir mensajes, si se devuelve `-1` tenemos que ver si es por un error verdadero o porque no había nada en espera de ser recibido. Usamos la variable `errno` para discernir esto.

Este enfoque tiene un problema, al realizar una “espera activa”, ocupamos el procesador innecesariamente.

**¡Necesitamos algo mejor!**



# The Ultimate Client-Server Implementation (sin Threads)

Demo: llamadas bloqueantes y `select()`

# The Ultimate Client-Server Implementation (sin Threads)

## Demo: llamadas bloqueantes y `select()`

- La llamada al sistema `select()` toma como parámetro un conjunto de **file descriptors** sobre los cuales se quiere esperar y un **timeout**.
- Se puede separar los **fds** según si se espera por lectura, escritura, etc.
- Vencido el plazo retorna el control y avisa que no hubo eventos.
- Si hay algún evento, nos retorna el **fd** correspondiente.

# The Ultimate Client-Server Implementation (sin Threads)

## Demo: llamadas bloqueantes y select()

- La llamada al sistema `select()` toma como parámetro un conjunto de **file descriptors** sobre los cuales se quiere esperar y un **timeout**.
- Se puede separar los **fds** según si se espera por lectura, escritura, etc.
- Vencido el plazo retorna el control y avisa que no hubo eventos.
- Si hay algún evento, nos retorna el **fd** correspondiente.

De esta forma evitamos la espera activa y logramos el mismo efecto que las llamadas no bloqueantes sin hacer morfar al CPU.

### Pero...

Under Linux, `select()` may report a socket file descriptor as "ready for reading", while nevertheless a subsequent read blocks. This could for example happen when data has arrived but upon examination has wrong checksum and is discarded. There may be other circumstances in which a file descriptor is spuriously reported as ready. Thus it may be safer to use `O_NONBLOCK` on sockets that should not block.

# Sockets de Internet

¿Qué pasa si queremos comunicar dos procesos en computadoras distintas? Quizás incluso se trata de computadoras de sistemas operativos totalmente diferentes.

---

<sup>1</sup>Más sobre protocolos de red y sockets en Teoría de las Comunicaciones

# Sockets de Internet

¿Qué pasa si queremos comunicar dos procesos en computadoras distintas? Quizás incluso se trata de computadoras de sistemas operativos totalmente diferentes.

## Socket de Internet

- En vez de usar nombres de archivo para establecer la comunicación, se usa la dirección IP y un número de puerto.
- Determinados servicios usan puertos estándar: 80 para HTTP, 22 para SSH, etc.

---

<sup>1</sup>Más sobre protocolos de red y sockets en Teoría de las Comunicaciones

# Sockets de Internet

¿Qué pasa si queremos comunicar dos procesos en computadoras distintas? Quizás incluso se trata de computadoras de sistemas operativos totalmente diferentes.

## Socket de Internet

- En vez de usar nombres de archivo para establecer la comunicación, se usa la dirección IP y un número de puerto.
- Determinados servicios usan puertos estándar: 80 para HTTP, 22 para SSH, etc.

Tipos de comunicación<sup>1</sup>:

**TCP:** Garantiza la llegada de los datos, el orden relativo y la integridad de los mismos, etc.

**UDP:** Se envían paquetes independientes uno de otro. No hay garantías de ningún tipo respecto de su arribo, ni la corrección de los datos que llevan.

---

<sup>1</sup>Más sobre protocolos de red y sockets en Teoría de las Comunicaciones

# ¿Preguntas?