

Comunicación interprocesos (IPC)

Sistemas operativos

Iván Raskovsky

basada en la clase de Nicolás Rosner, Verano 2010

24 de agosto de 2010

- POSIX.
- Creación de procesos: `fork`, `execvp`, `wait`.
- Conceptos básicos de IPC.
- Comunicación vía *pipes*.
- Comunicación vía *sockets*.
- Taller: MINI-TELNET.

POSIX: *Portable Operating System Interface [for UNIX]*

- Una familia de estándares de llamadas al sistema operativo definidos por la IEEE.
- Persiguen generalizar las interfaces de los sistemas operativos para que una misma aplicación pueda ejecutarse en distintas plataformas.

Creación de procesos con POSIX

```
pid_t fork();
```

Con `fork()` un proceso puede crear un proceso hijo.

- El hijo es una copia del espacio de direcciones del proceso original.
- Padre e hijo continúan ejecutando concurrentemente.
- La ejecución en ambos prosigue desde la instrucción siguiente al `fork()`.

Creación de procesos con POSIX

```
pid_t fork();
```

Con `fork()` un proceso puede crear un proceso hijo.

- El hijo es una copia del espacio de direcciones del proceso original.
- Padre e hijo continúan ejecutando concurrentemente.
- La ejecución en ambos prosigue desde la instrucción siguiente al `fork()`.
- La única diferencia es que la llamada a `fork()` le devuelve:

Al hijo: un 0.

Al padre: el *pid* del proceso hijo.

Demo fork

Si el sistema implementa `fork` de forma ingenua, esta puede ser muy ineficiente.

Si el sistema implementa `fork` de forma ingenua, esta puede ser muy ineficiente.

Sin embargo, copiar todo el espacio de direcciones no siempre es necesario.

Si el sistema implementa `fork` de forma ingenua, esta puede ser muy ineficiente.

Sin embargo, copiar todo el espacio de direcciones no siempre es necesario.

Copy on write

- El hijo utiliza las mismas regiones de memoria que el proceso padre.
- Al intentar (padre o hijo) modificar alguna, se copia únicamente esa porción.
- El resto de las porciones siguen siendo compartidas.

¿Qué pasa si no queremos que el hijo sea una copia del padre, sino otro proceso distinto?

Creación de procesos con POSIX

¿Qué pasa si no queremos que el hijo sea una copia del padre, sino otro proceso distinto?

```
int execlp(char *file, char *arg1, char *arg2, ...);
```

Con `execlp(...)` se reemplaza la ejecución del proceso llamador por otro ejecutable que es llamado con argumentos `arg1`, `arg2`, ...

El último argumento debe ser `NULL` para avisar el final de la lista.

Demo exec1p

Creación de procesos con POSIX

A veces el padre termina antes que el hijo, retornando al *shell* mientras su hijo sigue escribiendo. ¡Muy feo!

Creación de procesos con POSIX

A veces el padre termina antes que el hijo, retornando al *shell* mientras su hijo sigue escribiendo. ¡Muy feo!

Para eso tenemos:

```
pid_t wait(int *status);
```

El proceso llamador espera hasta que algún hijo suyo cambia de estado. Esto sucede, entre otras cosas, cuando el proceso hijo finaliza. El valor de retorno es el *pid* del hijo que cambió de estado y en *status* se almacena el estado al cual transicionó.

Demo wait

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.

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.

No lo detallaremos en esta clase.

IPC vía paso de mensajes

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

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

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

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

La comunicación puede ser:

Directa: se indica a qué proceso se envía, o de qué proceso se recibe.

Indirecta: se indica a qué buzón (o puerto) se envía o desde qué buzón (o puerto) se recibe.

Asimismo, la comunicación puede ser:

- 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.

Asimismo, la comunicación puede ser:

- 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. paso 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. paso 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 "la vaca es un poco flaca" | wc -c
```


Comunicación vía *pipes*

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

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

```
echo "la vaca es un poco flaca" | wc -c
```

- 1 Se llama a `echo`, un proceso 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?

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

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

```
echo "la vaca es un poco flaca" | wc -c
```

- 1 Se llama a `echo`, un proceso 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? **25.**

Standard streams

Los *flujos* estándar son canales de entrada y salida preconectados para conectar un programa y su entorno.

stdin: Son datos de entrada al proceso, generalmente texto.

Standard streams

Los *flujos* estándar son canales de entrada y salida preconectados para conectar un programa y su entorno.

stdin: Son datos de entrada al proceso, generalmente texto.

stdout: Es el *stream* donde el programa escribe su output.

Standard streams

Los *flujos* estándar son canales de entrada y salida preconectados para conectar un programa y su entorno.

stdin: Son datos de entrada al proceso, generalmente texto.

stdout: Es el *stream* donde el programa escribe su output.

stderr: Es donde el programa generalmente escribe errores o diagnosticos.

Demo pipe

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.

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.

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.

Standard streams como fd

Cada standard stream tiene asociado un file descriptor.

`stdin`: tiene asociado el `fd==0`

`stdout`: tiene asociado el `fd==1`

`stderr`: tiene asociado el `fd==2`

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?

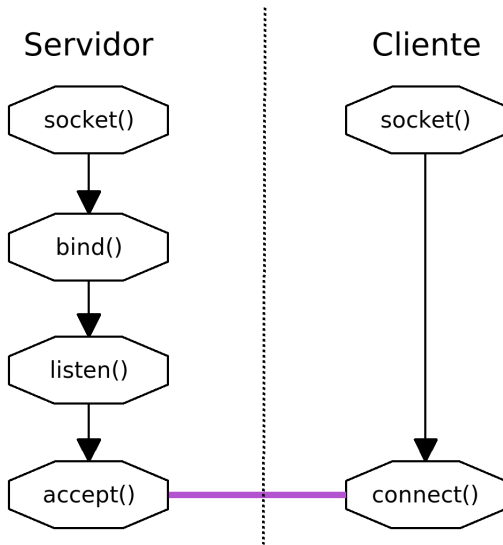
Socket

- 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

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

Estableciendo *sockets*: cliente

- ❶ `int socket(int domain, int type, int protocol);`
Crear un nuevo *socket*.
- ❷ `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 para atender cada cliente.
(Esta última 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!

Solución 2: usar `select()`

Demo: llamadas bloqueantes y `select()`

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.

Solución 2: usar `select()`

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.

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

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.

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:

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.

Vamos a hacer un cliente de MINI-TELNET¹.

Parte 1.

- El servidor está dado por nosotros.
 - Recibe mensajes UDP. Puerto 5555. Longitud máxima 1024.
 - Ejecuta el comando recibido.
 - El output de dicho comando **no se envía de vuelta al cliente**.
- El cliente lo deben programar ustedes.
 - Debe recibir como parámetro la IP del servidor.
 - Envía mensajes hasta que alguno es “chau” y termina.

¹Esto es un invento de la cátedra. No existe en el “mundo real” tal cosa como el protocolo MINI-TELNET

¡Manos a la obra!

Vamos a hacer un cliente de MINI-TELNET².

Parte 2.

- El servidor lo tienen que programar (o modificar) ustedes.
 - Recibe mensajes TCP. Puerto 5555. Longitud máxima 1024.
 - Ejecuta el comando recibido.
 - El output de dicho comando **se envía de vuelta al cliente**. Tanto stdout como stderr.
- El cliente lo deben programar (o modificar) ustedes.
 - Debe recibir como parámetro la IP del servidor.
 - Debe mostrar el output de cada comando ejecutado en el servidor.
 - Envía mensajes hasta que alguno es “chau” y termina.

²Esto es un invento de la cátedra. No existe en el “mundo real” tal cosa como el protocolo MINI-TELNET

¡Manos a la obra!

Vamos a hacer un cliente de MINI-TELNET².

Parte 2.

- El servidor lo tienen que programar (o modificar) ustedes.
 - Recibe mensajes TCP. Puerto 5555. Longitud máxima 1024.
 - Ejecuta el comando recibido.
 - El output de dicho comando **se envía de vuelta al cliente**. Tanto stdout como stderr.
- El cliente lo deben programar (o modificar) ustedes.
 - Debe recibir como parámetro la IP del servidor.
 - Debe mostrar el output de cada comando ejecutado en el servidor.
 - Envía mensajes hasta que alguno es “chau” y termina.

Ayuda: breve apunte con pistas para la implementación.

²Esto es un invento de la cátedra. No existe en el “mundo real” tal cosa como el protocolo MINI-TELNET

Deben enviar el código debidamente comentado, a la lista de docentes con subject:

Cliente MT: grupo Apellido1, Apellido2

Reemplazando Apellido_i por los apellidos de los **2** integrantes del grupo.