Taller de IPC

Sistemas Operativos

Segundo cuatrimestre de 2020

En este taller se implementarán dos formas de comunicación entre procesos. La primera es a través del uso de PIPES, y la comunicación está orientada a realizarse localemente, y la segunda forma se orienta a la comunicación remota utilizando SOCKETS, en un esquema cliente-servidor.

Ejercicio 1: One ring to rule them all

Para este primer ejercicio la idea es lograr implementar un esquema de comunicación en forma de anillo para intercomunicar los procesos. En un esquema de anillo existen al menos tres procesos conectados formando un lazo cerrado. Cada uno de ellos se encuentra comunicado con exáctamente dos procesos: su antecesor y su sucesor. Del antecesor recibe un mensaje, y al sucesor se lo envía. En este caso, dicha comunicación se realizará a través de **PIPES**.

Inicialmente, alguno de los procesos del anillo recibirá un número entero como mensaje a comunicar. Este mensaje será enviado al siguiente proceso en el anillo el cual, luego de recibirlo, lo incrementa en uno y luego lo envía al próximo proceso en el anillo. Este comportamiento se repetirá hasta que el proceso que inició la comunicación reciba, del último proceso, el resultado del mensaje inicialmente enviado.

Se sugiere que el programa inicial cree un conjunto de procesos hijos, a los que debe organizar para formar el anillo. Ejemplo: el hijo 1 recibe el mensaje, lo incrementa, y se lo envía al hijo 2, éste lo vuelve a incrementar y se lo envía al hijo 3, y así sucesivamente, hasta llegar al último hijo, el cual incrementa el valor por última vez, para luego enviárselo al proceso padre. El cual puede mostrar en la salida estándar el resultado final del proceso de comunicación.

Se espera que el programa pueda ejecutarse como `./anillo <n><c><s>`, donde:

- `<n>` es la cantidad de procesos del anillo.
- `<c>` es el valor del mensaje inicial.
- `<s>` es el número de proceso que inicia la comunicación.

Para tener en cuenta:

- En un *pipeline*, todos los procesos se pueden ejecutar al mismo tiempo (de manera **concurrente**).
- Es importante no dejar abiertos los file descriptors (FD) de los *pipes* que no se utilicen. En este caso, ¿qué sucede si no se cierran algunos FD de los *pipes*? ¿Por qué?
- En el campus, en la sección de clases prácticas encontrarán información útil sobre este tema y también algunos ejemplos de código. Recuerden que las páginas del manual son de mucha utilidad, por ejemplo: `man 7 pipe`, `man 2 pipe` y `man 2 dup`,

Ejercicio 2: Todos para... uno?

En un esquema de comunicación cliente-servidor, se les provee un código a completar de un cliente (CLIENT.C) que, al ejecutarse, toma como parámetro la dirección IP de un servidor y establece una conexión con éste a través del protocolo TCP en el puerto 8001.

El programa cliente lee desde la entrada estándar los mensajes que enviará al servidor. Cuando lee el comando "chau", terminará su ejecución.

Se debe completar el código del cliente CLIENT.C para que se comunique con el servidor de la siguiente manera:

1. Conectar a un socket del servidor.
2. Recibir un mensaje de bienvenida por parte del servidor.
3. Esperar un mensaje por entrada estándar.
4. Enviar este mensaje al servidor.
5. Imprimir en pantalla cada mensaje que envíe el servidor.
6. Cuando el servidor envíe CMDSEP se debe dejar de esperar más mensajes.
7. Volver al ítem 3.

Recomendamos fuertemente leer el material de las clases teórica y práctica y buscar el comportamiento de las funciones en el manual de Linux.

También es muy importante que lean el siguiente apunte:

Apunte para el ejercicio de SOCKETS

Sockets

Un *file descriptor*, en particular un *fd* de *socket*, tiene tipo `int`. Recordar de la clase que se puede crear un *socket* usando:

```
int socket(int domain, int type, int protocol);
```

Para trabajar con *sockets* de internet usaremos el `domain AF_INET`. Para TCP, usaremos el `type SOCK_STREAM`. Recordar que en `protocol` en general se utiliza un `0` (ver `/etc/protocols`).

Un socket se cierra con `close(int socket)`.

Direcciones de internet

Para representar una dirección de internet se usa la estructura presentada a continuación:

```
struct sockaddr_in {
    unsigned short sin_family;   /* dominio, usamos AF_INET */
    in_port_t      sin_port;     /* número de puerto */
    struct in_addr sin_addr;     /* dirección IP */
    unsigned char  sin_zero[8];  /* padding (no se usa) */
};
```

Donde la estructura que contiene la dirección IP es la siguiente:

```
struct in_addr {
    in_addr_t s_addr; /* Esto es un número de 32 bits */
};
```

Network byte order

Las estructuras mencionadas arriba necesitan tener el **puerto** y la **dirección IP** almacenadas en un formato conocido como *Network byte order*[1]. Para ello contamos con funciones de conversión:

- `uint16_t htons(uint16_t hostshort)` convierte un *uint16_t* del *host* (máquina local) en un *uint16_t* de la red.
- `uint32_t htonl(uint32_t hostlong)` análoga pero convierte *uint32_t*.

Resolver direcciones IP

Para convertir una cadena de caracteres que contiene una dirección IP (por ejemplo: “127.0.0.1”) en una estructura `in_addr` usamos:

```
int inet_aton(const char *cp, struct in_addr *inp);
```

[1] Se trata de un estándar *big-endian*

Esta función ya nos deja la dirección IP en formato *Network byte order*.

¡Ojo! Esta función devuelve `0` en caso de error (sí, es al revés que la mayoría de las funciones de sistema).

Conexion TCP

Una conexión TCP desde el lado del servidor requiere de tres pasos: `bind`, `listen` y `accept`. **Identifique a través del manual qué realiza cada uno de estos pasos.**

```
int bind(int s, sockaddr* a, socklen_t len);
```

```
int listen(int s, int backlog);
```

```
int accept(int s, sockaddr* a, socklen_t* len);
```

Enviar y recibir paquetes

Una vez que un cliente solicita conexión y esta es aceptada por el servidor pueden 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);
```

Otras funciones útiles

- `ssize_t getline(char **lineptr, size_t *n, FILE *stream);`
  Leer una línea (hasta "`\n`") desde `stream`.
- `int strncmp(const char *s1, const char *s2, size_t n);`
  Comparar dos cadenas `s1` y `s2` de longitud a lo sumo `n`.
- `int system(const char *command)`
  Ejecuta el comando `command` en un shell.