Sistemas Operativos

Trabajo Práctico Especial 1

Filesystems, IPCs y Servidores Concurrentes

Alumnos:

Alderete, Facundo - 51063

Buireo, Juan Martín - 51061

Perez Cuñarro, Javier - 49729

ITBA - Cuatrimestre 2 - 2012

Índice

1. Introducción

Este trabajo consistió en simular un intérprete de un lenguaje de programación especificado por la cátedra. Este lenguaje consiste en instrucciones simples que modifican valores numéricos en memoria (por ejemplo, `inc(2)`), instrucciones que modifican la posición actual en memoria (`como mr(5)`) e instrucciones condicionales que afectan el flujo del código (como `while(9, cz)`).

El simulador es una aplicación del tipo cliente/servidor en la cual los clientes envían programas al servidor para que sean interpretados. El servidor fue pensado para poder recibir cientos de pedidos de clientes distintos a la vez, y ejecutar los programas de forma concurrente. Al correr, estos programas modifican la zona de memoria propia de cada cliente. El resultado de la ejecución, entonces, se obtiene devolviendo la memoria modificada al cliente.

Para resolver este trabajo, se utilizó el lenguaje *C* principalmente debido a los requerimientos de bajo nivel de la consigna. Se utilizó el entorno de desarrollo *Eclipse* junto con aplicaciones propias de Unix que corren en terminal — tales como `ps` o `sockstat` — para supervisar el estado de los IPCs creados. Se creó un proyecto en *Github* para aprovechar el sistema de control de versiones y para poder desarrollar sobre varios *branches* a la vez.

2. Objetivo

De acuerdo al enunciado, el objetivo de este trabajo es familiarizarse con el uso de sistemas cliente-servidor concurrentes, implementando el servidor mediante la creación de procesos hijos utilizando `fork()` y mediante la creación de threads. Al mismo tiempo, ejercitar el uso de los distintos tipos de primitivas de sincronización y comunicación de procesos (IPC) y manejar con autoridad el filesystem de *Linux* desde el lado usuario.

3. Diseño e implementación

3.1. Estructuras de datos

Al momento de recibir el archivo de texto como única información, pensamos en la forma de traducir las instrucciones en algo que pueda ser manipulado ágilmente por un programa. Entonces desarrollamos una estructura de datos sencilla consistente en un "nodo", el cual es capaz de almacenar información y de conectarse con otros nodos como él. Gracias a esta estructura y, dependiendo del flujo que siga el programa que ingresa al server, se podrán formar estructuras con forma de listas, árboles o grafos, aunque de ahora en adelante nos referiremos a él como "el grafo", ya que es éste el término más general. A modo de ejemplo, se incluye en el apéndice un programa típico junto con su respectivo árbol de ejecución (ver página 6).

Además se utiliza una estructura Stack de manera auxiliar durante la formación del grafo, para poder conectar el principio con el final de un ciclo o condición, al mismo tiempo que se respeta el anidamiento de estas instrucciones.

En cuanto al envío y recepción de mensajes, se creó una estructura llamada message_t. La misma se forma así:

```
#define MAX_BUFFER_SIZE 4000
typedef struct {
        int pid;
        char buffer[MAX_BUFFER_SIZE];
        int error;
} message_t;
```

Se ha decidido que todos los IPCs reciban y envíen esta misma estructura. De este modo es posible cambiar el IPC que se utiliza sin que se altere ningún otro aspecto del programa. Simplemente se debe compilar toda la aplicación con el archivo .C correspondiente al IPC deseado.

La variable entera pid almacena el process id (es decir el número del proceso) del cliente que envía su pedido. Más adelante en este informe se explica el motivo de uso de este número.

La variable error se utiliza para que el servidor, al devolver la respuesta al cliente, pueda informar si todo salió bien o hubo algún error, ya sea en el procesamiento del archivo o durante la ejecución del programa. El vector buffer es el encargado de almacenar el path del archivo a leer y posteriormente, el vector de respuesta. Se eligió como tamaño 4000 ya que dicha respuesta sería un vector de enteros de 1000 posiciones lo cual ocupa dicho tamaño si lo representamos como un vector de caracteres. Ampliar el tamaño de este buffer consistiría en simplemente modificar el valor del define MAX_BUFFER_SIZE.

3.2. Procesos y paralelismo

Una de las características fundamentales de nuestra aplicación es que es capaz de atender a un gran número de clientes en simultáneo (limitado generalmente por los recursos de hardware de la máquina y por las características del IPC utilizado), al mismo tiempo que se es capaz de continuar "escuchando" en caso de que se produzcan más llamadas de otros clientes.

Esto es posible gracias a que se utilizó la herramienta fork provista por Unix, que nos permitió crear servidores idénticos al original, capaces de dedicarse exclusivamente a la atención del cliente que los llamó, para luego terminar su ejecución. El código necesario (pseudocódigo) para llevar a cabo esta tarea es el siguiente:

```
while (TRUE) {
Recibo el pedido de un cliente.
        switch (pid = fork()) {
        case -1:
                ERROR;
                exit(-1);
                break;
        case 0:
                Ejecutar programa;
                exit(1);
        default:
                break;
        }
}
```

3.3. Esquema de comunicaciones

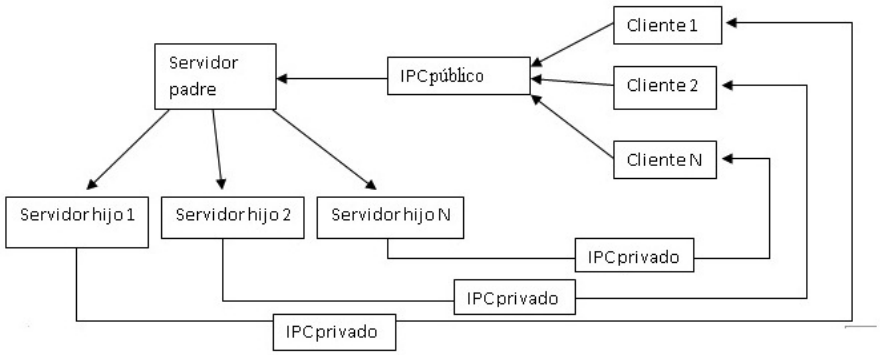

Figura 1: Representación gráfica de las comunicaciones entre clientes y servidor

Como se puede observar en el gráfico, hay un servidor padre corriendo de fondo en todo momento. En cuanto llega un cliente, el mismo se conecta a través del IPC público y envía el mensaje correspondiente. Cuando el servidor recibe el mensaje, éste se realiza un fork de sí mismo, de manera que sigue escuchando a nuevos clientes mientras atiende al cliente que le llegó. Una vez que el servidor hijo termina de atender al cliente, devuelve la respuesta por el IPC privado (salvo en message queue) y este servidor hijo termina su ejecución. El cliente que recibe la respuesta, la imprime en un archivo y también finaliza su ejecución.

3.4. FIFO

Tal como lo pedía la consigna, se podían utilizar tanto pipes como fifos en esta parte. Luego de leer acerca de ellos y comprender sus diferencias se llegó a la conclusión de que los fifos (debido a sus características) nos serían de mayor ayuda que los pipes. Esto se debe a que los fifos, al ser simplemente pipes con nombre, nos permiten identificarlos mediante el mismo a la hora de comunicar varios clientes con un servidor. La lógica que se llevó a cabo para implementarlos fue establecer un número fijo para el servidor para que de esta manera cualquier cliente pueda conocer su canal. Así, se crea un fifo público por el cual todos los clientes envían sus datos al servidor, y luego, cada cliente recibe la respuesta por su propio canal privado.

Los fifos (que se almacenan en el directorio /tmp) se nombran con la forma ipc(pid), donde pid es el número del proceso. Como se mencionó previamente, al establecer un número fijo para el servidor, cada cliente tiene acceso al canal público. Cuando un cliente se comunica con el servidor, el mismo abre un canal privado y envía a través del público su pid para que de esta manera, el servidor luego pueda saber por dónde enviar la respuesta correspondiente. Esto nos permite que un cliente no reciba la respuesta de un programa que él no envió.

3.5. Message Queue

Partiendo del IPC FIFO, resultó bastante fácil desarrollar la cola de mensajes, ya que sólo se trató de una adaptación. Fue por eso que en un principio se decidió crear una cola de mensajes pública y una privada para devolver el resultado a cada cliente.
Luego de pensarlo un poco nos dimos cuenta de que era muy ineficiente hacer esto ya que se estaba desaprovechando en gran medida la capacidad de la cola de mensajes de manejarse asignando prioridades a sus usuarios. Fue por eso que finalmente se logró manejar tanto al servidor como a todos los clientes con una única cola de mensajes. Nuevamente, aprovechando la idea de que cada proceso tiene su propio pid (y que el número del servidor es conocido por todos los clientes) se decidió que cada cliente sería capaz de enviar sus datos a través de la cola de mensajes con prioridad "server", para que el servidor los reciba con dicha prioridad. De esta manera, un cliente no sacaría lo que otro cliente puso en la cola. Asimismo, una vez que se terminó de ejecutar el programa, el servidor introduce en la cola la respuesta con prioridad "pid del cliente" y dicho cliente la levanta con la misma prioridad, asegurándose así que un cliente no reciba una respuesta errónea o de otro cliente.

3.6. Shared memory

Para implementar este IPC se siguió una lógica parecida a la de los fifos y la cola de mensajes, es decir, contamos con una zona de memoria pública donde todos los clientes pueden escribir, y una zona de memoria privada para cada cliente en la cual se recibe la respuesta.

A diferencia de un fifo, que automáticamente se bloquea, en este caso hubo

que implementar semáforos para lograr sincronización. De otro modo, un cliente podría levantar el mensaje que otro cliente envió, o el servidor podría leer datos erróneos de una zona de memoria que aún no fue escrita.

Se utilizaron semáforos a los cuales se les asignó un nombre, y se almacenan en devshm con la identificación ipc(pid), siendo pid el número de proceso. De esta forma, cada proceso puede acceder al semáforo deseado utilizando su pid.

La implementación de los semáforos se llevó a cabo de la siguiente forma. Se crearon dos semáforos públicos (uno de escritura y uno de lectura) y un semáforo privado por cada cliente. Para comprender mejor el funcionamiento de los semáforos detallamos a continuación un pseudocódigo de lo que hace el servidor y uno de lo que hace el cliente.

```
Servidor
Abre el canal publico;
While(1) {
       Mensaje = recibir();
       Fork() {
              padre: sigue escuchando;
              hijo: atiende el programa;
       }
}

Hijo() {
       Conexion al canal privado;
       Envia(respuesta);
}
```

```
Cliente
Conexion al canal publico;
Abre el canal privado;
Envia(mensaje por canal publico);
Recibe(respuesta por el canal privado);
```

Al comenzar el programa el servidor siempre está en ejecución en background. Cuando un cliente llega, se conecta al mismo y así comienza el envío y la recepción de datos. Al comenzar el servidor corriendo de fondo, si no hubiera semáforos, el mismo leería datos erróneos de la zona de memoria, ya que nadie ha escrito aún en la zona pública. Es por eso que el semáforo público de lectura comienza en 0, para que el servidor espere a que algún cliente escriba y así levante dicho semáforo. Por otro lado, el semáforo público de escritura comienza en 1 para que el primer cliente que llegue pueda escribir. Cuando un cliente escribe baja el semáforo de escritura y lectura asegurándose así que nadie escriba en ese momento ni que nadie intente leer.

El semáforo privado es más sencillo ya que sólo se relaciona con un cliente.

Dicho semáforo comienza en 0 de manera que el cliente se queda esperando que el servidor escriba en la zona de memoria privada para poder leer. En el momento en que el servidor escribe, dicho semáforo se levanta y así el cliente que lo tiene asignado puede leer los datos que le corresponden.

En cuanto al acceso a la memoria, la misma se manejó a través del pid ya que la llamada shmget recibe un key. En este caso, como se mencionó, se utilizó el pid como base para construir el key.

3.7. Sockets

Este fue el último de los IPCs implementados. Como ya se había logrado realizar de manera genérica el resto de los IPCs (definiendo un ipc.h a modo de "interfaz" e implementando cada IPC según su uso), tuvimos que lograr adaptar este mismo al código genérico que estaba hecho.

Comenzamos intentando implementar TCP el cual nos trajo muchos conflictos debido a que un cliente al conectarse al canal público automáticamente abría un canal de respuesta.

Allí decidimos cambiar la implementación por UDP con dominio UNIX. Esto nos facilitó seguir el esquema de código genérico, ya que usa la creación de sockets mediante archivos. De esta manera, creamos en el directorio /tmp un socket con el nombre ipc(pid) donde pid es el número de proceso. Igual que en los IPCs anteriores, al conocer el número de identificación del servidor, cada cliente sabe a qué socket conectarse, y a su vez el servidor sabe en qué socket debe depositar su respuesta.

4. Inconvenientes encontrados

En un principio nos resultó muy difícil comprender qué tipos de datos es posible pasar a través de un IPC. Los primeros diseños requerían el pasaje de punteros y estructuras complicadas y, luego de investigar y asesorarnos con alumnos de cursadas anteriores, entendimos que no es posible pasar datos que involucren el uso de punteros a través de un IPC. Según entendimos, esto ocurre porque un proceso no tiene acceso a la zona de memoria del proceso que le pasa los datos, dejando inutilizados a los punteros.

Fue por eso que finalmente, en lugar de enviar la estructura del grafo previamente parseada, decidimos que sería mucho más conveniente enviar simplemente un string con el path del archivo a leer, encargándose el servidor de parsearlo, construir el grafo a partir de él y ejecutarlo, para luego devolver la respuesta.

Por otra parte, analizamos la posibilidad de paralelizar la ejecución del grafo utilizando threads, pero luego de pensarlo mucho, y de consultar con la cátedra, no encontramos forma de llevar esto a cabo, más que nada porque las instrucciones que es capaz de ejecutar la aplicación dependen una de otra en gran medida. Es por esto que finalmente decidimos que no valía la pena sumar complejidad al código por algo que no agregaría eficiencia.

5. Conclusiones

El trabajo sirvió en gran medida para comprender el uso de los IPCs. Pudimos notar que los IPCs son de fundamental importancia para trabajar con varios procesos a la vez. Hasta el momento nunca los habíamos usado por ignorar su utilidad. Definitivamente será una herramienta que tendremos en cuenta ya que nos permite lograr ciertas funciones que hasta ahora hacíamos de forma ineficiente. Por ejemplo, hemos llegado a pasar un mensaje mediante la persistencia de datos en un archivo de texto.

En cuanto a la performance de los IPCs creados, tal y como se esperaba, el IPC más rápido fue shared memory. Esto se observó al correr un número bastante grande de clientes, específicamente 10.000. Y tiene sentido ya que se maneja un bloque de memoria directamente, por lo cual el acceso es muy rápido. El IPC que encontramos más lento fue la cola de mensajes, aunque en este caso hay que notar también que a diferencia del resto de los IPCs, se utilizó solo una cola de mensajes tanto para el canal público como para los privados, por lo cual la misma debía manejar demasiadas cosas.

6. Apéndice

Ejemplo de un programa a ejecutar junto con el árbol correspondiente:

```
INC(20)
MR(2)
WHILE(2, INC(20) IF(2, INC(20)) ML(3) ENDIF(2))
DEC(30)
INC(20)
ENDWHILE(2)
```

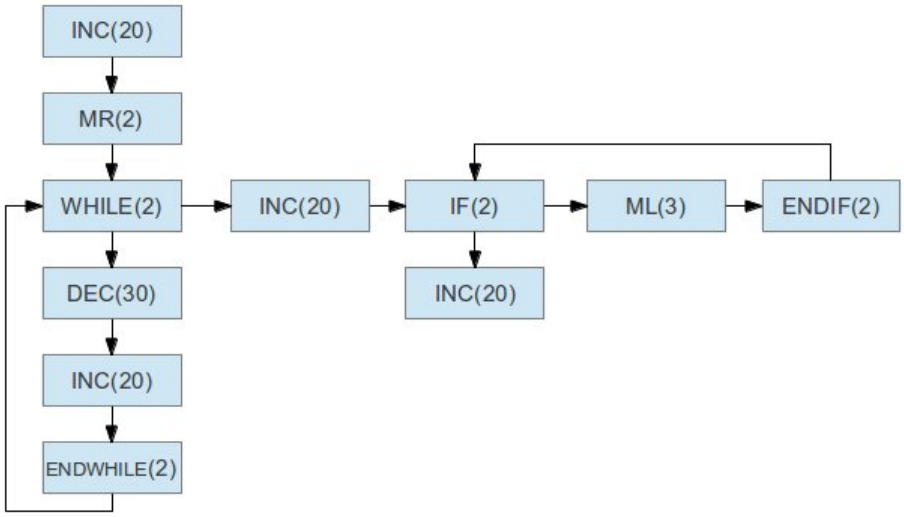

Figura 2: Representación gráfica del árbol de ejecución