Trabajo Obligatorio
Sistemas Operativos III

Profesor : Carlos Fidalgo

Integrantes:

Nombre	**C.I.**
Emiliano Fernández	4496822-0
Juan Aparicio	4820381-8

Índice

Introducción

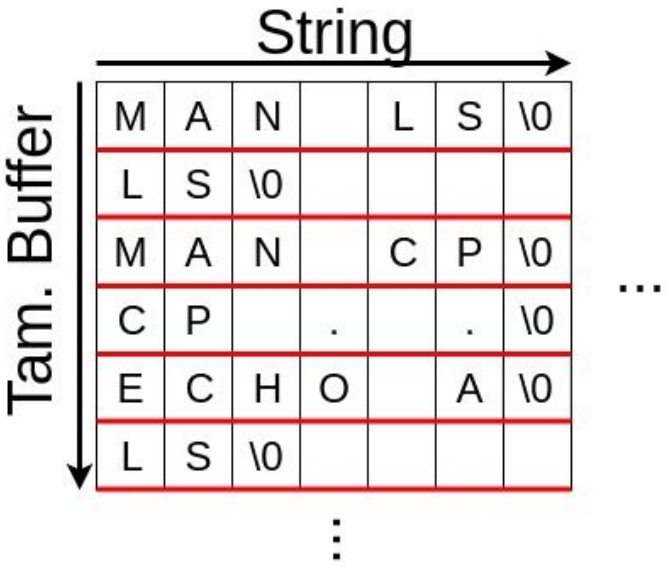
Lo que se pide es un programa que lea comandos de un archivo y este se los pase a otros procesos para que los puedan ejecutar y guardar su salida en archivos de log.

La solución al problema se realizó mediante el uso de semáforos, memoria compartida y pipes.
Se creó un proceso (productor) que reserva un espacio de memoria compartida, y a medida que lee los comandos del archivo, los ubica en una estructura predefinida en ese espacio de memoria compartida. Luego se cuenta con otro programa (consumidor) que se puede instanciar tantas veces como se quiera, que accede al espacio de memoria compartida definido por el proceso productor y lee los comandos que son ubicados allí, para su ejecución.

Solución al problema

Para poder sincronizar el proceso productor y los consumidores se tuvo que hacer uso de un semáforo (mutex, es decir iniciado con el valor 1) que se usa para que solamente un proceso pueda acceder a la memoria compartida a la vez. Si no se hiciese uso del semáforo, pueden ocurrir errores si el proceso consumidor lee a la vez que el productor escribe.
La estructura definida para almacenar los comandos es un buffer circular. Este buffer circular se implementó utilizando una matriz, en donde cada fila representa una entrada del buffer (un String) y la cantidad de columnas el largo máximo del String. Esta se ve de la siguiente forma:

```
typedef struct {
   char buffer[TAM_BUFFER][TAM_STRING];
   int inicio;
   int fin;
   int largo;
} buffer_t;
```



Los índices `inicio` y `fin` sirven para saber a partir de qué fila se lee la siguiente entrada y en qué fila se escribe la siguiente entrada.
El buffer cuenta con los métodos `push` y `pop`, que sirven para escribir al siguiente espacio disponible del buffer y obtener el primer dato disponible del buffer respectivamente (el buffer es una cola FIFO):

```
int buffer_push(buffer_t *b, string_p_t dato);

int buffer_pop(buffer_t *b, string_p_t dato);
```

A su vez uno de los requerimientos del programa es que los procesos consumidores se enteren cuando es que el proceso padre finalizó de leer los comandos del archivo. Para esto se hizo otro estructurado que envuelve el buffer junto con otro int que se setea en 1 si se termino de leer los comandos del archivo y por defecto es inicializado en 0. Este se ve de la siguiente forma:

```
typedef struct{
   buffer_t bufferCirc;
   int finArchivo;
}datos_sh_mem_t;
```

Para el manejo de los semáforos, se utiliza un `sem_wait()` y un `sem_post()` antes y después de hacer una llamada que sea de R/W del buffer respectivamente:

```
...
/*Inicio seccion critica*/
sem_wait(semaforo);
int finalizado = datos_sh_mem_finalizado(shMemData);
sem_post(semaforo);
/*fin seccion critica*/
...
...
/*Inicio seccion critca*/
sem_wait(semaforo);
buffer_push(&(shMemData -> bufferCirc), linea);
sem_post(semaforo);
/*Fin seccion critica*/
...
```

Para la ejecución de los comandos se decidió por utilizar la función popen. Esta función hace varias cosas dentro:

1. Inicia un nuevo proceso (mediante fork)
2. Crea un pipe
3. Invoca el comando en la shell

La función popen se ve de la siguiente manera:

```
FILE *popen(const char *command, const char *type);
```

(Se puede ver la implementación aquí: git.musl-libc.org/cgit/musl/tree/src/stdio/popen.c)

Donde:

- **`const char *command`** es el comando a ejecutar
- **`const char *type`** es un `String` que admite **“`w`”** si se quiere escribir a la entrada del proceso o **“`r`”** si se quiere leer la salida del proceso.

Como la función internamente usa un pipe, y este es por su definición unidireccional, no se puede leer y escribir al mismo tiempo dentro de la misma función `popen`, por eso se pasa el parámetro `type`, para que sea `read-only` o `write-only`.

Como en nuestro caso solo se quiere leer la salida del comando para luego escribirla a un archivo, se utiliza la bandera **“`r`”**.

Asignación de la memoria

Para compilar el programa se utilizo el comando:

```
gcc main.c String.c Buffer.c DatosShMem.c -pthread -o main
```

Esto genera un archivo binario llamado main. Al correrlo (`./main`) se puede usar el comando `pgrep main` que obtiene el pid del proceso que se llama `main`. En uno de los casos de ejecución, devuelve el PID 28513. Luego de obtenido el PID se utiliza el comando `pmap 28513` para obtener un mapeo de la memoria del proceso:

```
28513:   ./main
000056464da7c000      4K r---- main
000056464da7d000      4K r-x-- main
000056464da7e000      4K r---- main
000056464da7f000      4K r---- main
000056464da80000      4K rw--- main
000056464f6bf000    132K rw---   [ anon ]
00007f1110cab000     12K rw---   [ anon ]
00007f1110cae000    136K r---- libc-2.28.so
00007f1110cd0000   1324K r-x-- libc-2.28.so
00007f1110e1b000    304K r---- libc-2.28.so
00007f1110e67000      4K ----- libc-2.28.so
00007f1110e68000     16K r---- libc-2.28.so
00007f1110e6c000      8K rw--- libc-2.28.so
00007f1110e6e000     16K rw---   [ anon ]
00007f1110e72000     24K r---- libpthread-2.28.so
00007f1110e78000     60K r-x-- libpthread-2.28.so
00007f1110e87000     24K r---- libpthread-2.28.so
00007f1110e8d000      4K r---- libpthread-2.28.so
00007f1110e8e000      4K rw--- libpthread-2.28.so
00007f1110e8f000     24K rw---   [ anon ]
00007f1110ec5000      4K rw-s- sem.sem_tr_ob1
00007f1110ec6000      4K rw-s-   [ shmid=0x688003 ]
00007f1110ec7000      8K r---- ld-2.28.so
00007f1110ec9000    124K r-x-- ld-2.28.so
00007f1110ee8000     32K r---- ld-2.28.so
00007f1110ef0000      4K r---- ld-2.28.so
00007f1110ef1000      4K rw--- ld-2.28.so
00007f1110ef2000      4K rw---   [ anon ]
00007fff5b0ef000    132K rw---   [ stack ]
00007fff5b13a000     12K r----   [ anon ]
00007fff5b13d000      8K r-x--   [ anon ]
total             2448K
```

Aqui se puede ver las páginas dedicadas a las librerías: (libc, libpthread y ld), el espacio reservado para el semáforo (sem.sem_tr_ob1) y el id de la memoria compartida (`0x688003`) el cual se puede ver también mediante el comando `ipcs -mp`.

Asignación de almacenamiento secundario

Los archivos de log se generan en la misma carpeta en la que se está ejecutando el proceso consumidor. Para ver la asignación de inodos de estos se pueden utilizar varios comandos:

```
ls -i -l | grep .log
```

Este comando lista todos los archivos dentro de una carpeta junto con el número de inodo al principio:

```
9830964 -rw-r--r-- 1 juan juan  1212 Nov 18 09:35 2018-11-18_9:35_ifconfig.log
9834349 -rw-r--r-- 1 juan juan   765 Nov 18 09:35 2018-11-18_9:35_ls -l.log
```

En la salida se puede ver que al archivo `2018-11-18_9:35_ifconfig.log` se le asignó el i-nodo `9830964`.

También se puede usar el comando `stat *.log` para ver información más detallada de los archivos `.log` generados por los productores:

```
File: 2018-11-18_9:35_ifconfig.log
 Size: 1212            Blocks: 8          IO Block: 4096   regular file
Device: fe00h/65024d    Inode: 9830964     Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/    juan)   Gid: ( 1000/    juan)
Access: 2018-11-18 09:35:23.220660929 -0300
Modify: 2018-11-18 09:35:23.223994199 -0300
Change: 2018-11-18 09:35:23.223994199 -0300
Birth: -
 File: 2018-11-18_9:35_ls -l.log
 Size: 765             Blocks: 8          IO Block: 4096   regular file
Device: fe00h/65024d    Inode: 9834349     Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/    juan)   Gid: ( 1000/    juan)
Access: 2018-11-18 09:39:51.981485486 -0300
Modify: 2018-11-18 09:35:24.237308351 -0300
Change: 2018-11-18 09:35:24.237308351 -0300
Birth: -
```

Aquí se muestra toda la información: Tamaño del archivo, cantidad de bloques asignados, el tamaño del bloque, el dispositivo en el que se encuentra almacenado, y los accesos junto con los permisos del archivo.

Permisos de los archivos utilizados en la solución

Los programas productor/consumidor requieren de los siguientes archivos:

- productor (archivo binario)
- consumidor (archivo binario)
- comandos.txt (archivo con los comandos a ejecutar)
- archivos .log (archivos donde se almacena la salida de los comandos)

Los **binarios productor y consumidor** son producidos por el comando `gcc` que por defecto le asigna los siguientes permisos:

```
rwxr-xr-x
```

Sin embargo, como el archivo es un ejecutable, solo bastaría con tener el permiso `--x--x--x`. Para esto se ejecutaron los comandos:

```
chmod -r ./productor
chmod -w ./productor
```

Luego se ejecuto el archivo de vuelta y se comprobo que sigue funcionando la ejecución.

El archivo **comandos.txt** se creó mediante el comando `nano comandos.txt` y se le escribieron 2 líneas. El archivo quedó entonces con los permisos

```
rw-r--r--
```

Sin embargo, para la ejecución del programa. se puede tener solo los permisos `r--r--r--` ya que el productor solo va a leer y no escribir. Entonces se ejecuto el comando `chmod -w ./arch.txt` para sacarle los permisos de escritura y que el archivo sea `read-only`. Luego de la ejecución del programa consumidor se comprobó que el archivo puede ser read only.

Los **archivos .log** generados por el proceso consumidor tienen por defecto los permisos

```
rw-r--r--
```

Si no contiene los permisos `r` no se podría leer la salida luego, y si no tuviese el permiso `w` no se podría escribir la salida desde el proceso consumidor.

Solución alternativa con mensajería

Se podría haber llegado a una solución utilizando el pasaje de mensajes. El productor leería los comandos del archivo de comandos, y a medida que los lee, los envía mediante un mensaje a una mailbox que soporte una cantidad n de mensajes que se almacenan allí hasta que son consumidos todos.
UNIX tiene una implementación de Mailbox (XSI message queues) y C cuenta con las llamadas al sistema de estas:

La estructura de un mensaje es la siguiente:

```
struct mymsg {
    long    mtype;       /* Message type. */
    char    mtext[1];    /* Message text. */
}
```

Donde:

- **mtype** es el tipo de mensaje.
- **mtext** es el texto que se quiere enviar en el mensaje.

Se tienen los siguientes métodos para operar con la cola de mensajes:

```
int msgget(key_t key, int msgflg);
```

Obtiene el identificador de la cola de mensajes asociado a la clave key.

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Envía un mensaje a la cola asociada con la cola de mensajes identificada por `msqid`

```
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int
msgflg);
```

Lee un mensaje de la cola asociada con la cola de mensajes por msqid.

El método `msgsnd()` reemplazaría el método `buffer_push()` y el método `msrcv()` reemplazaría el método `buffer_pop()`

Conclusiones

La solución encontrada mediante memoria compartida no es la más eficiente, ya que como se vio en el diagrama, si los comandos no ocupan todo el largo máximo del string dentro del buffer, se desperdicia espacio en los caracteres. Una mejor solución hubiese sido usando mensajería, ya que estos tienen espacio *indefinido* para almacenar los mensajes y usa señales para avisar a los consumidores que pueden leer un mensaje de la casilla, de esta forma no se tienen que hacer un *polling* constante para ver si hay algún dato nuevo en la memoria compartida.