

**Sistemas Operativos .  
Ejercicios de Programación  
en Lenguaje C, Parte I.**



**Mg. Guillermo R. Cherencio**



## Indice

Indice.....	3
Agradecimientos.....	4
Capítulo I. Introducción .....	5
Capítulo II. Procesos Pesados.....	6
Capítulo III. Procesos Livianos (Threads).....	15
Capítulo IV. Señales (Signals).....	26
Capítulo V. Tuberías I (Pipes).....	37
Capítulo VI. Tuberías II (FIFO's).....	45
Capítulo VII. IPC, Cola de Mensajes ( Queues).....	65
Capítulo VIII. IPC,Memoria Compartida y Semáforos (Shared Memory & Semaphores).....	76
Capítulo IX. Entrada - Salida (Input - Output).....	98
Palabras Finales.....	114
Bibliografía .....	115

## **Agradecimientos**

A mi familia: Liliana, Carolina y Santiago.

A mis alumnos, sin ellos, esto no tendría sentido ni razón de ser.

A mis maestros, sin ellos, esto no podría haberse hecho.

A la Editorial del Centro de Estudiantes de Ingeniería Tecnológica, Universidad Tecnológica Nacional, Facultad Regional Buenos Aires, gracias a su trabajo, esto se hizo realidad.

A la Universidad Tecnológica Nacional, Facultad Regional Delta, sin su apoyo, esto no hubiera sido posible.

## Capítulo I. Introducción

El presente libro tiene como objetivo ser una introducción al uso y aplicación de los recursos e interfaces que expone -en este caso- el Sistema Operativo Unix a los desarrolladores de aplicaciones. De cómo pueden resolverse problemas típicos, de cómo podemos comunicarnos con el núcleo (kernel) del sistema operativo, de cómo aplicar técnicas básicas de programación, de cómo desarrollar aplicaciones típicas que pueden aplicarse con o sin variantes en entornos laborales reales.

El libro no pretende dar explicación de conceptos puntuales de Sistemas Operativos, para ello recomendamos la lectura de la bibliografía perteneciente a autores tales como William Stallings, Andrew Tanenbaum o Silberschatz.

El libro no pretende aplicar conceptos de sistemas operativos a través de la ejercitación de los mismos, para ello recomendamos la lectura del libro "Recopilación de Ejercicios de Sistemas Operativos" de Carlos Neetzel de esta misma Editorial.

El libro pretende mostrar la aplicación de conceptos relacionados con sistemas operativos a través del desarrollo de aplicaciones o programas.

El libro plantea una serie de ejercicios que Ud. puede resolver, de forma autoguiada, con ejemplos y fragmentos de código en Lenguaje ANSI C; no obstante, también puede descargar todo el código fuente usado en este libro desde <http://www.grch.com.ar/docs/so/sop1.zip>.

Los requisitos necesarios para trabajar con este texto son: tener conocimientos básicos de programación en Lenguaje ANSI C (manejo de punteros, estructuras, tipos definidos por el usuario, arreglos, archivos, librería standard, etc.)<sup>1</sup>, estar cursando o haber cursado la asignatura "Sistemas Operativos" (en cualquier Universidad o Instituto Superior) o contar con conocimientos básicos de sistemas operativos, conocimientos básicos de Unix o Linux, manejo de un glosario básico de términos en inglés que serán mencionados en el texto (se hará el esfuerzo de traducirlos muy sintéticamente).

El código de este libro está probado con el compilador GNU C/C++ GCC versión 4.3 en Linux Debian 5.05 32 bits, recomendamos que utilice este compilador en una plataforma linux igual o superior de 32 bits (i386) o 64 bits (amd64). Algunas partes del código no funcionarán en distribuciones GCC para Windows, tales como cygWin o mingW. Tampoco funcionará gran parte del código en compiladores nativos para Windows (ejemplo: C++Builder, MS Visual C/C++, etc.) puesto que el código es para la interfaz Unix.

---

<sup>1</sup> Además de los muchos tutoriales disponibles que hay, también puede consultar on-line los apuntes de mi autoría en cuanto a cuestiones básicas del lenguaje ANSI C (ver bibliografía).

## Capítulo II. Procesos Pesados.

**Objetivo:** Desarrollar una serie de funciones que encapsulen cierta funcionalidad vinculada con la gestión de procesos y uso de consola para luego reutilizarlas en la construcción de un Shell (intérprete de comandos).

### Funcionalidades (features) del Shell propuesto

- ◆ Ejecución de procesos en primer plano (foreground)
- ◆ Ejecución de procesos en segundo plano (background)
- ◆ Redirección de la entrada y la salida standard en ambos tipos de ejecución de procesos
- ◆ Notificar al usuario acerca del id o identificación de proceso creado en segundo plano (background)

### Introducción

El shell o intérprete de comandos del SO es un programa que corre como una aplicación más de usuario y nos permite (a través de comandos y acciones definidas en el programa) poder realizar en forma interactiva algunas acciones sobre el SO sin necesidad de escribir un programa. Una de las principales acciones es la ejecución de procesos.

Cuando el usuario interactúa directamente con el shell, el esquema es el siguiente:

**usuario <=> Intérprete de comandos (shell) <=> llamadas al sistema (system calls) <=> núcleo SO (kernel) <=> hardware**

Cuando el usuario ejecuta un proceso en primer plano (foreground) a través del shell, el proceso pasa a ser un proceso hijo del proceso shell y el control de la terminal del usuario (que antes estaba bajo control del shell) pasa al proceso hijo. El esquema es el siguiente:

**usuario <=> proceso <=> Intérprete de comandos (shell) <=> llamadas al sistema (system calls) <=> núcleo SO (kernel) <=> hardware**

Cuando el usuario ejecuta un proceso en segundo plano (background) a través del shell, el proceso también pasa a ser un proceso hijo del shell, pero en este caso el control del terminal sigue estando en manos del proceso shell y habría una ejecución concurrente entre el proceso shell y hijo ejecutándose en segundo plano. El esquema es el siguiente:

**usuario <=> Intérprete de comandos (shell) <=> llamadas al sistema (system calls) <=> núcleo SO (kernel) <=> hardware**

proceso  $\Leftrightarrow$  Intérprete de comandos (shell)  $\Leftrightarrow$  llamadas al sistema (system calls)  $\Leftrightarrow$  núcleo SO (kernel)  $\Leftrightarrow$  hardware

### Acerca de la Entrada, la Salida y el Error

Todo proceso tiene asociado tres flujos de datos (streams): la entrada standard (stdin, asociado al dispositivo teclado), la salida standard (stdout, asociado al dispositivo monitor) y el error standard (stderr, asociado al dispositivo monitor). Estos flujos pueden redirigirse hacia otros destinos o bien desde otros orígenes. La redirección puede hacerse a través de instrucciones dentro del programa:

```
//en un sistema gnu, puede redirigirse directamente, conectando un nuevo flujo
//a stdin, stdout, stderr
//Ejemplo: ahora la salida standard se guardara en el archivo salida.txt
fclose(stdout);
stdout = fopen("salida.txt", "w");
```

Otra forma de uso de redirección por programa, sería duplicar el descriptor de un archivo, podríamos hacer -por ejemplo- que stdout (que habitualmente está asociado al descriptor de archivo 1 (STDOUT\_FILENO)) también se redirija a "salida.txt". Se puede utilizar la función fcntl() (archivo de cabecera <fcntl.h>), pero existe la función dup2() (archivo de cabecera <unistd.h>) de uso más cómodo. Esto es particularmente útil cuando queremos redirigir un proceso hijo del shell:

```
//abre el archivo salida.txt de output, si no existe lo crea, si existe trunca
//su contenido.
//Asocia este archivo con la salida standard del proceso actual
outfile = open("salida.txt", O_WRONLY|O_CREAT|O_TRUNC);
dup2(outfile, STDOUT_FILENO);
close(outfile);
```

En la redirección interactiva se utilizan los signos de mayor (>), menor (<).  
Ejemplo: enviar la salida de prog1 al archivo salida.txt:

```
$ prog1 > salida.txt
```

que prog1 tome la entrada del archivo entrada.txt:

```
$ prog1 < entrada.txt
```

que prog1 tome la entrada del archivo entrada.txt y la salida la envíe a salida.txt:

```
$ prog1 < entrada.txt > salida.txt
```

### Acerca de los Procesos en Primer Plano y Segundo Plano

La ejecución de procesos en primer plano (foreground) implica que el proceso tiene el control de la terminal o consola que está operando el usuario, por lo general, se trata de procesos interactivos y es la forma habitual de invocar a los procesos.

La ejecución de procesos en segundo plano (background) es un tipo de ejecución no interactiva, que remite a los tiempos de los sistemas de procesos por lotes (batch); muchos procesos no requieren de la interacción con el usuario. No obstante, el usuario, como operador del sistema, puede monitorear todo tipo de procesos a través del comando `ps` (estado de los procesos (process status)):

```
$ ps -af
UID      PID  PPID  C STIME TTY      TIME CMD
knoppix  3721  3697  0 18:13 tty5      00:00:00 /bin/bash /usr/bin/adriane
knoppix  4069  3721  0 18:13 tty5      00:00:00 dialog --title ADRIANE --no-ok
knoppix  5095  4690  0 18:27 pts/0      00:00:00 ps -af
$
```

Para ejecutar un proceso en segundo plano (background) se utiliza el símbolo & al final del comando; por ejemplo, para ejecutar `prog1` en segundo plano, suponiendo que este programa tiene algún tipo de entrada y salida, lo apropiado sería redirigirla<sup>2</sup>:

```
$ prog1 < entrada.txt > salida.txt &
```

En este caso, nuestro shell debería informar al usuario el número de proceso creado a fin de poder seguirlo con el comando `ps`. El comando `jobs` también es utilizado para visualizar los procesos en background:

```
$ jobs -l
[1]+  5564 Stopped (tty output)      vi
$
```

Se recomienda que el shell realice en forma automática la redirección de salida, en caso de que el usuario no lo indique.

## Sugerencias

El programa principal del shell propuesto, podría tener la siguiente forma:

```
// shell - programa principal
int main(void) {
    inicio();           // inicializo variables, estructuras, etc.
    ingreso();          // ingreso comando
    while(!salir) {     // mientras no sea el comando para salir ...
        proceso();       // proceso comando ingresado
        libero();         // libero memoria dinamica usada
        ingreso();        // ingreso comando
    }                   // fin-mientras
    return 0;
}
```

<sup>2</sup> Si el programa requiere algún tipo de entrada por teclado y estuviera en segundo plano, quedaría bloqueado esperando una entrada que no se concretaría.

}

La función `ingreso()` trata del ingreso del comando en la consola por parte del usuario y luego su análisis (parsing) para determinar qué tipo de comando es: un comando propio del shell (ejemplo: comando `salir` para salir del programa) o un proceso a ejecutar. Efectivamente, se trata de un programa que iniciará la ejecución de otro programa (spawning) indicado por el usuario.

Como el shell es interactivo, se supone que no puede haber más de un `ingreso()` al mismo tiempo, es decir, que se procesa "de a un comando ingresado por vez", por lo tanto, podríamos contar con una estructura en donde almacenar toda la información del comando actual:

```
// shell - rutina de ingreso de comando por teclado
struct {
    pid_t pid;          // id de proceso del shell
    pid_t pgid;         // id del grupo de proceso del shell
    int interactivo;   // ¿comando en foreground?
    int input;          // descriptor de archivo de stdin del shell
    char *comando;      // comando ingresado por el usuario
    char *error;         // error (de parsing,etc.) del ultimo comando ingresado
    char *arg[64];       // cada elemento es una parte significativa del comando
                         // ingresado por el usuario
    int nargs;          // cantidad de partes significativas que tiene el
                         // comando actual (no puede superar de 64)
    int op;              // luego del parsing, se determino este tipo de comando
    int salir;            // ¿salgo del shell?
    char *entrada;       // nombre de archivo al cual redirigir entrada
    char *salida;        // nombre de archivo al cual redirigir salida
} sh;
```

El ingreso del usuario podría tener la siguiente forma:

```
// shell - rutina de ingreso de comando por teclado
void ingreso() {
    prompt();           // simbolo a mostrar por pantalla
    ingresoLinea();    // ingreso de una linea de texto por teclado
    parsecLinea();     // separacion de la linea ingresada en "tokens"
}
```

## Resuelve:

1. El código de la función `ingresoLinea()` teniendo en cuenta que no sabemos cuántos caracteres puede ingresar el usuario hasta presionar la tecla Intro. Sugerencia: leer de a un carácter de `stdin`:

```
char c;
...
c = (char) getc(stdin);
```

usar `malloc()` para asignar dinámicamente un buffer `n bytes`, ir almacenando en dicho buffer los caracteres leídos, cuando se llene el buffer, volver a pedir más memoria con `realloc()`:

```
// nbuf cuenta los caracteres almacenados en buf
char *buf = (char *) malloc(nbytes); // puede devolver NULL
...
if ( (nbuf % nbytes) == 0 ) {
    nbufsize+=nbytes;
    buf = (char *) realloc(buf,nbufsize); // puede devolver NULL
}
...
sh.comando = buf; // agregar un carácter nulo al final: buf[nbuf]='\0';
```

2. El código de la función `parseoLinea()`. Esta función tiene como entrada `sh.comando` (el comando tipiado por el usuario), debe hacer algunas validaciones, separar a `sh.comando` en las partes significativas del comando<sup>3</sup> y actualizar: `sh.arg`, `sh.narg`, `sh.entrada`, `sh.salida`, `sh.error` (si encontrara algún error), etc.. Para separar `sh.comando` se recomienda usar `strtok()`, haciendo una copia previa del string a parsear:

```
// instrucciones posibles, parte de la función parseoLinea()
...
int i=0;
// asigno espacio para que buffer pueda contener una copia de sh.comando
char *buffer = (char *) malloc(strlen(sh.comando)+1);
...
strcpy(buffer,sh.comando); // buffer=sh.comando
// separa a buffer en i partes separadas por uno o mas espacios
buffer = strtok(sh.comando, " ");
while (i < 64 && buffer != NULL) { // mientras haya mas partes ...
    sh.arg[i] = buffer; // apunto a la i-esima parte
    buffer = strtok(NULL, " "); // busco la parte siguiente
    i++; // incremento el contador de partes
} // fin-mientras
// i es igual a la cantidad de partes en que se dividió sh.comando
```

<sup>3</sup> El usuario puede tipar n espacios entre el nombre del programa a ejecutar y el primer argumento o el segundo, etc. en el arreglo `sh.arg[]` solo deben guardarse el programa, los argumentos, etc. (con un máximo de 64 elementos, en este caso), sin espacios innecesarios.

```
sh.narg=i;
```

```
...
```

### 3. Compile y execute el siguiente programa C:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main(void) {
    int a=4;
    printf("padre: id de proceso=%d\n", (int) getpid());
    printf("padre: a=%d\n", a);
    printf("padre: lanza fork()\n");
    pid_t pid = fork();
    switch (pid) {
        case -1: // error, padre
            printf("padre:Error en fork()\n");
            break;
        case 0: // hijo
            printf("hijo: id de proceso=%d\n", (int) getpid());
            printf("hijo: a=%d\n", a);
            printf("hijo: sumo 1 a variable a\n", a);
            a++;
            printf("hijo: a=%d\n", a);
            break;
        default: // padre
            printf("padre: sumo 10 a variable a\n", a);
            a+=10;
            printf("padre: a=%d\n", a);
            break;
    }
    return 0;
}
```

#### Responda:

3.1 ¿Qué sucede luego de la llamada a la función `fork()`?

3.2 ¿Cuál es el valor final de la variable a para el proceso padre? ¿Por qué?

3.3 ¿Cuál es el valor final de la variable a para el proceso hijo? ¿Por qué?

3.4 Ejecute varias veces el programa. Es posible que no vea todos los mensajes enviados a la consola con `printf()` de la forma esperada. ¿A qué se debe esto?

4. La familia de funciones `exec` (`execv()`, `execl()`, `execve()`, `execle()`, `execvp()`, `execpl()`) permiten reemplazar la imagen del proceso actual con la imagen de un programa guardado en disco. Por lo tanto, suelen utilizarse luego de ejecutar `fork()` para reemplazar al proceso hijo con una imagen de otro proceso. Esto es lo que debería hacer nuestro shell cuando un usuario invoque un comando. Modifique el programa del punto 3, en la sección de código correspondiente al proceso hijo, entre la última instrucción `printf("hijo: a=%d\n", a);` y la instrucción `break;` que le sigue, introduzca el siguiente código C:

```
...
switch(pid) {
```

```

...
    case 0:    // hijo
...
    printf("hijo: a=%d\n",a);
    const char *programa = "ls";
    char *argumentos[] = { "ls", "-l", NULL };
    int ret = execvp(programa, argumentos);
    printf("hijo: termine de ejecutar execvp()=%d\n",ret);
    break;
...
}
...

```

*Nota: se pretende ejecutar el comando "ls" (list directory) con el argumento "-l" (guion "ele", long format).*

**Responda:**

- 4.1 ¿Por qué no aparece en la consola el printf() enviado por el proceso hijo luego de ejecutar la función execvp()?
- 4.2 ¿Existe alguna diferencia en la ejecución de este programa con respecto a la versión anterior, hasta antes de ejecutar la función execvp()?
- 4.3 ¿Hay algún cambio en la secuencia de instrucciones ejecutadas por el proceso padre?
- 4.4 Averigüe todo lo que más pueda en cuanto a la funciones exec()

**4 b1s).** Si Ud. es muy observador, posiblemente habrá notado que debe presionar Intro para volver nuevamente al shell, luego del cambio agregado en el punto anterior. También podrá observar que el proceso padre imprime todos sus mensajes y luego puede verse la ejecución del proceso hijo, esto implicaría que el padre finalizó su ejecución antes que el proceso hijo (este sería un proceso zombie). Modifique el programa del punto anterior, en la sección de código correspondiente al proceso padre, entre la última instrucción printf("padre: a=%d\n",a); y la instrucción break; que le sigue, introduzca el siguiente código C:

```

...
switch(pid) {
...
    case default:    // padre
...
    printf("padre: a=%d\n",a);
    printf("padre: id proceso hijo=%d\n", (int) pid);
    printf("padre: esperando que hijo termine...\n");
    int estado;
    waitpid(pid, &estado, WUNTRACED);
    printf("padre: hijo termino con estado=%d\n",estado);
    break;
}
...

```

**Responda:**

- 4 b1s.1 Observe ahora la ejecución del programa y su salida, ¿Cuál es el último mensaje que aparece en pantalla?
- 4 b1s.2 Averigüe todo lo que más pueda en cuanto a la función waitpid()

4 b1s.3 Haga un pequeño cambio en el programa y agregue otro argumento más a la lista de argumentos del comando ls, por ejemplo "afafjakfjasfkjasñlfjkaskl", ejecute nuevamente el programa y observe si cambia el estado de retorno del proceso hijo

4 b1s.4 En el código correspondiente al proceso padre, ¿Qué valor representa la variable pid?

5. Combinando lo aprendido en los puntos anteriores, la función proceso() podría tener el siguiente formato:

```
void proceso() {
    ...
    switch(sh.op) {
        case 0:           // comando salir
            sh.salir=1;   // salir = true
            break;
        case 1:           // usuario pretende ejecutar comando
            pid_t pid = fork();
            switch (pid) {
                case -1:      // error, padre
                    printf("Error en fork()!\n");
                    break;
                case 0:        // hijo
                    ...
                    setpgrp(); // proceso hijo como nuevo líder del grupo de procesos
                    pid_t chpid = getpid(); // id de proceso del hijo
                    if (sh.interactivo) // si es un proceso interactivo (foreground)
                        tcsetpgrp(sh.input, chpid); // debe controlar la terminal
                    else          // proceso batch (background)
                        printf("Proceso [%d] ejecutado en background\n", (int) chpid);

                    ejecutoProceso();
                    if (printError()) exit(EXIT_FAILURE); // retorno con error
                    else           exit(EXIT_SUCCESS); // retorno Ok
                    break;
                default:       // padre
                    ...
                    if (sh.interactivo) {
                        tcsetpgrp(sh.input, pid); // el nuevo proceso controla la terminal
                        // espero por la terminación del proceso hijo interactivo
                    (foreground)
                        int terminationStatus;
                        while(waitpid(pid, &terminationStatus, WUNTRACED | WNOHANG) == 0);
                        tcsetpgrp(sh.input, sh.pgid); // el shell retoma control de terminal
                    }
                    break;
            }
            break;
    }
    ...
}
```

Responda:

5.1 ¿Qué operaciones debería llevar a cabo la función ejecutoProceso()?

5.2 ¿En qué caso se ejecutaría la instrucción if (printError()) ... ?

6. Termine de unir las piezas faltantes (ejecutoProceso(), inicio(), libero(), etc.) para lograr un primera versión de este shell de comandos.

7. Hasta aquí solo hemos visto una única instrucción vinculada con la sincronización de procesos, ¿A qué instrucción nos estamos refiriendo?
8. Modifique el programa para agregarle el comando `cd` , cuando el usuario tipée este comando, deberá mostrar en pantalla el directorio de trabajo actual. Sugerencia: ver función `getcwd()` .
9. Modifique el programa para agregarle el comando `cd <nuevo directorio>`, cuando el usuario tipée este comando, el programa deberá cambiar el directorio actual por `<nuevo directorio>`. Sugerencia: ver función `chdir()` .
10. Cuando aprenda algo más acerca de la sincronización de procesos y señales, vuelva a esta primera versión del programa shell y modifíquelo para que el mismo le informe al usuario los procesos batch que van finalizando y en qué estado van finalizando.

## Capítulo III. Procesos Livianos (Threads) .

**Objetivo:** Desarrollar una serie de programas que permitan comprobar distintas funcionalidades vinculadas con la gestión de procesos livianos y su sincronización, a través de la utilización de la librería pthread (POSIX Threads).

### Introducción

Puede acceder on-line a un tutorial sobre pthread y otros temas relacionados en [https://computing.llnl.gov/tutorials/parallel\\_comp/](https://computing.llnl.gov/tutorials/parallel_comp/) . Todo el material necesario para realizar esta actividad lo encontrará en este sitio, en esta URL podrá leer el artículo titulado "Introduction to Parallel Computing" de Blaise Barney [Barney]; allí encontrará información relevante acerca de procesos concurrentes y su campo de aplicación; también se menciona la taxonomía de Flynn en cuanto a la clasificación de sistemas con procesamiento en paralelo.

En el mismo sitio, en <https://computing.llnl.gov/tutorials/pthreads/> encontrará el artículo "POSIX Threads Programming" de Blaise Barney [Barney] que será de gran ayuda para realizar los puntos de esta práctica, luego de haber leído el artículo mencionado anteriormente del mismo autor.

1. Realice el programa tp31.c que permite indicar por línea de comando la cantidad de threads a crear. Cada thread ejecutará una función que mostrará un mensaje en pantalla indicando un número único de thread (que es un simple valor entero pasado como parámetro de la función). Sugerencias: No olvide utilizar las cabeceras stdio.h, stdlib.h, pthread.h. Declare a la función main() como int main(int argc,char \*argv[]), si pasa un argumento en la línea de comando, argc vale 2 y en argv[1] se encuentra el argumento en forma de string. Para pasar de string a entero, puede utilizar la función de librería atoi(). Compile de la siguiente forma: gcc -Wall tp31 tp31.c -pthread y ejecútelo como: ./tp31 5 si pretende que main() cree 5 threads.

2. Modifique el programa, agregue el archivo de cabecera unistd.h, el cual contiene el acceso a varias llamadas al sistema (system calls) Unix tales como: open(), read(), write(), close(), seek() y sleep(). Esta función permite detener el proceso actual la cantidad de segundos que se le indique como argumento; por ejemplo, sleep(3) detiene el proceso 3 segundos. Esto coincide con el comando Unix sleep, por lo tanto, desde la consola, Ud. puede tipear sleep 3 y observara que el shell queda detenido durante 3 segundos.

Modifique la función que es ejecutada repetidamente como un thread, para que en medio de dos mensajes que imprima en la consola llame a la función sleep, por ejemplo:

```
printf("Mensaje de thread %d\n", tid);
sleep(3);
```

```
printf("fin Mensaje de thread %d\n", tid);
```

Agregue dos mensajes al final de la función `main()` (antes y después de la llamada a la función `pthread_exit()`, por ejemplo:

```
printf("fin main():1\n");
pthread_exit(NULL);
printf("fin main():2\n");
```

Compile y ejecute el programa, podrá obtener una salida como esta:

```
grchere@debian:~/gccwork/src/thr1$ ./tp32 5
main(): creando thread 0
Mensaje de thread 0
main(): creando thread 1
Mensaje de thread 1
main(): creando thread 2
Mensaje de thread 2
main(): creando thread 3
Mensaje de thread 3
main(): creando thread 4
Mensaje de thread 4
fin main():1
fin Mensaje de thread 0
fin Mensaje de thread 1
fin Mensaje de thread 2
fin Mensaje de thread 3
fin Mensaje de thread 4
grchere@debian:~/gccwork/src/thr1$
```

Observando los mensajes podemos concluir que la función `main()` termina su ejecución antes que los threads que ella misma dispara, lo cual no parece ser algo saludable o razonable. El proceso que dispara estos threads debería terminar cuando el último de los threads que él mismo disparó haya finalizado. Por lo visto, no tiene sentido escribir sentencias luego de `pthread_exit()`, puesto que éstas nunca se ejecutarán.

3. Copie el programa anterior en `tp33.c`. Modifique el programa para que la función `main()` no termine antes que los threads que ella dispara.

Sugerencia: La operación `join` hace que el proceso que la llama quede bloqueado, esperando hasta que el thread que se le indique a la función `join` termine. Modifique la función `main()` para que ésta haga un `join` a cada uno de los threads que ejecuta, de manera tal, que `main()` termine cuando ya hayan terminado todos los threads que él lanzó. Utilice la función `pthread_join()`. Declare a los threads como "joinables" (crear atributo, indicarlo como `joinable` y luego asociarlo a cada uno de los threads con `pthread_create()`):

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
```

Cuando ya no necesite el atributo creado, destrúyalo para liberar recursos:

```
// libero recurso que ya no necesito
pthread_attr_destroy(&attr);
```

Debería obtener una salida similar a esta:

```
grchere@debian:~/gccwork/src/thr1$ ./tp33 5
main(): creando thread 0
Mensaje de thread 0
main(): creando thread 1
Mensaje de thread 1
main(): creando thread 2
Mensaje de thread 2
main(): creando thread 3
Mensaje de thread 3
main(): creando thread 4
Mensaje de thread 4
main(): haciendo join con thread 0
fin Mensaje de thread 0
fin Mensaje de thread 1
main(): termino join con thread 0 status= 0
main(): haciendo join con thread 1
main(): termino join con thread 1 status= 0
main(): haciendo join con thread 2
fin Mensaje de thread 2
main(): termino join con thread 2 status= 0
main(): haciendo join con thread 3
fin Mensaje de thread 3
main(): termino join con thread 3 status= 0
main(): haciendo join con thread 4
fin Mensaje de thread 4
main(): termino join con thread 4 status= 0
fin main():1
grchere@debian:~/gccwork/src/thr1$
```

Observe los mensajes, mientras que `main()` está detenido por el join con el thread 0, vemos que termina el thread 0 y el 1, entonces `main()` se libera y pretende bloquearse con el join con el thread 1, pero éste ya había terminado, por lo tanto, este join, también termina inmediatamente, sin bloquear a `main()`. Luego se van alternando los bloqueos de `main()` con las finalizaciones de los threads. Pero el último mensaje impreso en la consola es la finalización de la función `main()`, ésta termina luego de que todos sus threads han terminado.

Se observa también que el valor de retorno de cada uno de los threads que recupera la función `main()` (usando la función `join`) es siempre cero.

4. Copie el programa anterior en `tp34.c`. Modificar el programa para que los threads que se ejecutan retornen a `main()` un número distinto de cero, podemos usar el número de thread más uno.

Debería obtener una salida similar a esta:

```

grchere@debian:~/gccwork/src/thr1$ ./tp34 5
main(): creando thread 0
Mensaje de thread 0!
main(): creando thread 1
Mensaje de thread 1!
main(): creando thread 2
Mensaje de thread 2!
main(): creando thread 3
Mensaje de thread 3!
main(): creando thread 4
Mensaje de thread 4!
main(): haciendo join con thread 0
fin Mensaje de thread 0!
fin Mensaje de thread 1!
main(): termino join con thread 0 status= 1
main(): haciendo join con thread 1
main(): termino join con thread 1 status= 2
main(): haciendo join con thread 2
fin Mensaje de thread 2!
main(): termino join con thread 2 status= 3
main(): haciendo join con thread 3
fin Mensaje de thread 3!
fin Mensaje de thread 3!
main(): termino join con thread 3 status= 4
main(): haciendo join con thread 4
main(): termino join con thread 4 status= 5
fin main():1
grchere@debian:~/gccwork/src/thr1$
```

5. Problema: Una empresa tiene 5 sucursales de las cuales registra las ventas de un mes. Existe un proceso que carga los datos de estas ventas. Luego esas ventas deben ser sumarizadas por sucursal para obtener un total general. Debemos diseñar una solución a este problema a través de un programa que trabaje en paralelo usando Threads. Siguiendo el apunte "Introduction to Parallel Computing" [Barney] podemos pensar en qué tipo de *particionado* (Partitioning) podríamos hacer de este problema. Podríamos implementar la función cargoVentas() como un thread que nos permita generar los datos de ventas de cada una de las sucursales en forma aleatoria de la siguiente manera:

```

#define NUM_OF_SUC 5

// datos de ventas de las sucursales
double mvta[NUM_OF_SUC][31]; // se asume un maximo de 31 dias para el mes en curso
double totalgral=0.0;
double totalsuc[NUM_OF_SUC];
...
void *cargoVentas(void *p) {
    int s,d;
    printf("thread cargoVentas\n");
    for(s=0;s<NUM_OF_SUC;s++) {
        for(d=0;d<31;d++) {
            mvta[s][d] = ((double) random()) / 10000000.0;
        }
        sleep(2); // para simular una mayor duración de este proceso de carga
    }
}
```

```

    }
    printf("fin thread cargoVentas\n");
}
...

```

Una vez generadas las ventas, podríamos utilizar un thread por sucursal para realizar la suma, a través de la función `sumoVentas()`, la cual recibe como argumento el id de sucursal (número entero de 0 a `NUM_OF_SUC` – 1):

```

void *sumoVentas(void *sucursalid) {
    int sucid;
    sucid = (int) sucursalid;
    printf("thread %d sumoVentas sucursal %d \n", sucid, sucid);
    totalsuc[sucid]=0.0;
    int d;
    for (d=0;d<31;d++) {
        totalsuc[sucid]+=mivta[sucid][d];
    }
    printf("fin thread %d sumatoria de sucursal %d
    $%10.2f\n",sucid,sucid,totalsuc[sucid]);
}

```

Una vez que termine la suma de todas las sucursales, se puede calcular el total general, desde la función `main()`:

```

...
int t;
for(t=0; t<NUM_OF_SUC; t++) totalgral+=totalsuc[t];
printf("main():Total Gral $ %12.2f\n",totalgral);
...

```

Debemos pensar acerca de la ejecución concurrente de `cargoVentas()` y `sumoVentas()`. Existe una *partición de datos* en `sumoVentas()` puesto que ésta trabaja por sucursal, con lo cual, si sumo las ventas de la sucursal 0 ello no debería interferir con estar sumando las ventas de la sucursal 1 al mismo tiempo. El problema estaría en generar datos de ventas al mismo tiempo que pretendo sumarlos. Por lo tanto, se propone implementar una forma de exclusión mutua: no se pueden generar datos y sumarlos al mismo tiempo.

*Nota: el código de las funciones antes descriptas está incompleto, falta todo código relacionado con el tratamiento de threads.*

Sugerencias: Copie el programa anterior en `tp35.c`, elimine todo el código vinculado con la línea de comandos (a este programa ya no se le indica con cuántas sucursales va a trabajar), implemente el código propuesto anteriormente, agregue todo el código de threads que sea necesario, al comienzo del programa, inicializar los datos numéricos (a cero), esto puede hacerse utilizando la función `memset()`:

```

...
memset (&mivta,0,sizeof(double)*NUM_OF_SUC*31);
memset (&totalsuc,0,sizeof(double) *NUM_OF_SUC);
...

```

ara usar `memset()`, debe incluir la cabecera `string.h`

**Responda:**

5.1. ¿Desde el punto de vista del diseño de programas en paralelo, el thread `cargoVentas()` a qué tipo de descomposición corresponde?

5.2. Idem anterior con los threads `sumoVentas()` ¿A qué tipo de descomposición corresponde?

Debería obtener una salida similar a esta:

```
grchere@debian:~/gccwork/src/thr1$ ./tp35
main(): creando thread para carga de ventas
thread cargoVentas
main(): creando thread 0
thread 0 esperando carga de sucursal 0
main(): creando thread 1
thread 1 esperando carga de sucursal 1
main(): creando thread 2
thread 2 esperando carga de sucursal 2
main(): creando thread 3
thread 3 esperando carga de sucursal 3
main(): creando thread 4
thread 4 esperando carga de sucursal 4
main(): haciendo join con thread 0
fin thread cargoVentas
thread 0 sumoVentas sucursal 0
fin thread 0 sumatoria de sucursal 0 $ 3337.60
thread 1 sumoVentas sucursal 1
fin thread 1 sumatoria de sucursal 1 $ 3421.03
thread 2 sumoVentas sucursal 2
fin thread 2 sumatoria de sucursal 2 $ 3904.04
thread 3 sumoVentas sucursal 3
fin thread 3 sumatoria de sucursal 3 $ 3514.70
thread 4 sumoVentas sucursal 4
fin thread 4 sumatoria de sucursal 4 $ 3326.67
main(): termino join con thread 0 status= 1
main(): haciendo join con thread 1
main(): termino join con thread 1 status= 2
main(): haciendo join con thread 2
main(): termino join con thread 2 status= 3
main(): haciendo join con thread 3
main(): termino join con thread 3 status= 4
main(): haciendo join con thread 4
main(): termino join con thread 4 status= 5
main(): Total Gral $ 17504.03
fin main():1
grchere@debian:~/gccwork/src/thr1$
```

Observando la salida podemos ver que todas las sumarizaciones de sucursales quedan bloqueadas mientras que `cargoVentas()` esta en ejecución.

Efectivamente, habíamos dicho que estas operaciones no podían ser concurrentes, se ha implementado una exclusión mutua entre ambas: mientras que se generan datos, no se pueden sumar. Una vez que `cargoVentas()` finaliza, comienzan a liberarse una a una las tareas que suman las sucursales. Cuando termina el ultimo sumador (como

`main()` hizo un `join` a cada uno de estos sumadores, éste queda bloqueado hasta que termine el último sumador), `main()` se libera, hace el total general, imprime el resultado y termina.

6. Intencionalmente (para no complejizar demasiado el código de una sola vez), he introducido un error en el razonamiento del diseño del programa anterior, ¿Puede advertir cuál es?, observe nuevamente la salida del programa anterior:

1. comienza la ejecución de `main()`
2. se ejecuta `cargoVentas()`
3. se ejecutan los sumadores por sucursal
4. se termina de ejecutar `main()`

¿No le parece un código secuencial? ¿Qué clase de provecho le estamos sacando al paralelismo? Evidentemente la solución propuesta no es la más adecuada y requiere algún tipo de refinamiento. El error está en cuanto a la exclusión mutua, no es posible sumarizar mientras se generan los datos, pero una vez que se ha generado los datos de la sucursal *n* ya puede desbloquearse el sumador de la sucursal *n*, ¿para qué dejar bloqueado el sumador de la sucursal *n* cuando ya se han "producido" todos sus datos y éstos pueden ser "consumidos"?

Se requiere poder hacer un bloqueo por sucursal, pero no es un bloqueo al estilo de exclusión como se hizo en el punto anterior, sino que, *este bloqueo debe realizarse a condición de que la sucursal n aún no esté generada*. Una solución posible es utilizar *variables de condición*, una por sucursal, entonces el sumador de la sucursal *n* estaría esperando por la variable de condición *n*, cuando `cargoVentas()` haya terminado la generación de los datos de la sucursal *n* puede cambiar el valor de la variable e indicarle al sumador *n* que ya puede hacer su trabajo.

Sugerencia: Copie el programa anterior en `tp36.c`. Se puede utilizar un arreglo de enteros, uno por sucursal, a modo de flags o banderas, para indicar que la sucursal *n* ha sido generada; inicialmente todo el vector está en cero (falso):

```
...
int cargadas[NUM_OF_SUC];
...
memset(&cargadas, 0, sizeof(int) * NUM_OF_SUC); //cargadas[n] indica falso
(cero)
...
```

Habrá una variable de condición y de exclusión mutua (mutex) por cada sucursal:

```
...
pthread_mutex_t cargada_mutex[NUM_OF_SUC];
pthread_cond_t cargada_cv[NUM_OF_SUC];
...
```

Ahora la función `cargoVentas()` deberá colocar un valor distinto de cero en el elemento *n* del arreglo `cargadas` para indicar que terminó con la generación de los

datos de la sucursal  $n$  e indicar que se dio la condición por la cual está esperando el thread sumador de sucursal  $n$ :

```
....  
void *cargoVentas(void *p) {  
    ...  
    cargadas[n]=1;  
    pthread_cond_signal(&cargada_cv[n]);  
    ...  
}
```

Entonces, la función `sumoVentas()` deberá esperar para obtener el candado de la sucursal  $n$  a sumar, preguntar por la variable de condición y quedarse en espera de la misma, antes de hacer su trabajo:

```
....  
void *sumoVentas(void *threadid) {  
    ...  
    printf("thread %d esperando carga de sucursal %d \n", tid,tid);  
    pthread_mutex_lock(&cargada_mutex[tid]);  
    if ( !cargadas[tid] ) {  
        printf("thread %d esperando condicion de cargada para sucursal %d \n",  
tid,tid);  
        pthread_cond_wait(&cargada_cv[tid], &cargada_mutex[tid]);  
    }  
    printf("thread %d sumo Ventas sucursal %d \n", tid,tid);  
    ...  
    pthread_mutex_unlock(&cargada_mutex[tid]);  
    ...  
}
```

*Nota: No olvide que si Ud. hace un `lock()` sobre una variable de condición, debe también hacerse su correspondiente `unlock()`.*

Debería obtener una salida similar a esta:

```
grchere@debian:~/gccwork/src/thr1$ ./tp36  
main(): creando thread para carga de ventas  
thread cargoVentas  
main(): creando thread 0  
thread 0 esperando carga de sucursal 0  
thread 0 sumo Ventas sucursal 0  
fin thread 0 sumatoria de sucursal 0  $  3337.60  
main(): creando thread 1  
thread 1 esperando carga de sucursal 1  
thread 1 esperando condicion de cargada para sucursal 1  
main(): creando thread 2  
thread 2 esperando carga de sucursal 2  
thread 2 esperando condicion de cargada para sucursal 2  
main(): creando thread 3  
thread 3 esperando carga de sucursal 3  
thread 3 esperando condicion de cargada para sucursal 3  
main(): creando thread 4  
thread 4 esperando carga de sucursal 4
```

```

thread 4 esperando condicion de cargada para sucursal 4
main(): haciendo join con thread 0
main(): termino join con thread 0 status= 1
main(): haciendo join con thread 1
thread 1 sumo Ventas sucursal 1
fin thread 1 sumatoria de sucursal 1 $ 3421.03
main(): termino join con thread 1 status= 2
main(): haciendo join con thread 2
thread 2 sumo Ventas sucursal 2
fin thread 2 sumatoria de sucursal 2 $ 3904.04
main(): termino join con thread 2 status= 3
main(): haciendo join con thread 3
thread 3 sumo Ventas sucursal 3
fin thread 3 sumatoria de sucursal 3 $ 3514.70
main(): termino join con thread 3 status= 4
main(): haciendo join con thread 4
thread 4 sumo Ventas sucursal 4
fin thread 4 sumatoria de sucursal 4 $ 3326.67
main(): termino join con thread 4 status= 5
main(): haciendo join con carga de ventas
fin thread cargoVentas
main(): termino join con carga de ventas status= 0
main(): Total Gral $ 17504.03
fin main():1
grchere@debian:~/gccwork/src/thr1$
```

Se observa que esta versión tiene un mayor nivel de concurrencia, la tarea `cargoVentas()` se está ejecutando durante casi todo el tiempo, `main()` debe hacer también un `join` con `cargoVentas()` para que éste no pretenda calcular el total general antes de que haya terminado la generación y sumarización por sucursal. Este código da oportunidad de que se ejecute, por ejemplo, la sumarización de la sucursal 0, mucho antes de que termine `cargoVentas()`.

Como ayuda a la comprensión de las variables de condición y siguiendo el código anterior, he realizado el siguiente pseudo-código; supongamos el siguiente programa principal:

```

variable compartida v1
declaro mutex m1
declaro condvar cv1

main()
    inicializo m1
    inicializo cv1

    lanzo thr1
    lanzo thr2

    join thr1
    join thr2
fin main()
```

Como se puede observar siempre una variable de condición (`cv1`) tiene asociado un `mutex` (`m1`) y existe un área de datos compartido (`v1`) entre más de un `thread` (`thr1`, `thr2`) que requiere ser utilizado de diferente forma (consumiendo o produciendo). Las

variables de condición permiten que un thread consumidor (thr2) no deba gastar tiempo de cpu en preguntar repetidamente ¿ya puedo consumir? ..¿ya puedo consumir? ... en vez de ello, queda bloqueado (ya sea por el mutex m1 o bien por la espera sobre la variable de condición cv1) hasta que el thread productor (thr1) le avisa (señalo cv1) que ya terminó su producción y por lo tanto, se permite su consumo. El pseudo-código para el productor (thr1) y el consumidor (thr2) sería el siguiente:

```
thr1()
    mientras proceso
        lock m1
        sección crítica 1
            si condición entonces
                señalo cv1
            fin si
        unlock m1
    fin mientras
fin thr1()

thr2()
    mientras proceso
        lock m1
        si no condición entonces
            espero cv1
        fin si
        sección crítica 2
        unlock m1
    fin mientras
fin thr2()
```

Sección crítica 1 se refiere al proceso de producción y sección crítica 2 se refiere al proceso de consumo. Se denominan secciones críticas porque requieren de un mecanismo de exclusión mutua, en este caso, implementado a través del mutex m1. Puede deducirse que espero cv1 en su interior hace un unlock m1, queda en espera de la variable de condición cv1 y cuando cv1 es señalada, hace un lock m1. Estas acciones se corresponden con las realizadas por thr1 para evitar un abrazo mortal entre ambas tareas.

7. Verifique el código de la función cargoVentas () en el programa tp35.c y tp36.c, ambas versiones de esta función deberían tener una llamada idéntica a la función sleep () para simular una mayor duración del proceso de generación de datos de sucursales. Este sleep () no pretende tener ningún tipo de relación con la sincronización de estos procesos (es un grave error intentar, a partir de esperas o tiempos arbitrarios, lograr la sincronización de procesos). Verifique que en ambas versiones de esta función tengan la misma demora.

Utilice el comando Unix time para hacer una ejecución del programa tp35 y otra del programa tp36, de la siguiente forma:

```
grchere@debian:~/gccwork/src/thr1$ time ./tp35
.... <observará la salida habitual de tp35>...
```

```
real 0mXXXXXs
user 0mXXXXXs
sys 0mXXXXXs
```

```
grchere@debian:~/gccwork/src/thr1$ time ./tp36
.... <observará la salida habitual de tp35>...
real 0mYYYYYs
user 0mYYYYYs
sys 0mYYYYYs
```

El comando **time** permite sumarizar la utilización de recursos del sistema por parte de un proceso. Real indica el tiempo real transcurrido en la ejecución del programa (wall clock) indicado en horas, minutos y segundos. Realice varias corridas y compare los tiempos obtenidos en Real para ambos procesos.

**Responda:**

6.1 ¿Cuál de los dos procesos obtiene valores menores de real?

6.2 ¿Por qué Ud. cree que esos valores de real son menores?

Cambie (en ambos procesos) el tiempo de espera dentro de la función sleep() de 2" a 5" y vuelva a comparar los tiempos

6.3 ¿Encontró alguna diferencia?

6.4 ¿El proceso que tiene menor valor de Real continúa siendo el mismo?

## Capítulo IV. Señales (Signals) .

**Objetivo:** Desarrollar una serie de programas que permitan comprobar distintas funcionalidades vinculadas con la gestión señales y su aplicación.

### Introducción

Puede acceder on-line al manual de referencia de la librería GNU C, consultar allí el capítulo 24 Señales (Signals) [Loosemore et. al.]. Los prototipos de las funciones a utilizar se encuentran en `signal.h`.

1. Desarrolle un programa que capture las señales `SIGUSR1` y `SIGUSR2`. Puede usar una misma función para manipular ambas señales, de la siguiente forma:

```
signal(SIGUSR1, sig_usr);
signal(SIGUSR2, sig_usr);
```

en este caso, la función que manipulara ambas señales es `sig_usr()`, cuyo prototipo es:

```
void sig_usr(int signo);
```

donde `signo` corresponde la numero de señal que se ha generado y se ha enviado a este proceso. La captura de la señal puede fallar, en tal caso, la función `signal()` devolverá `SIG_ERR`, el programa deberá controlar eso y emitir un mensaje de error en caso de no poder capturar las señales. Luego de capturar las señales, el programa entrará en un loop infinito y dentro del mismo pondrá un mensaje indicativo de lo que esta haciendo y llamará a la función `pause()`:

```
for ( ; ; ) {
    printf("main():ingreso en pause()\n");
    pause();
}
```

Dentro de la función `sig_usr()` emita un mensaje por cada señal que recibe indicando si se trata de `SIGUSR1` o `SIGUSR2` o cualquier otra señal que llegue; imprima también el numero de la señal capturada.

Compile el programa como `tp41` (`tp41.c`).

2. Ejecute el programa desde el shell en background, de la siguiente forma: `./tp41 &`  
Obtendrá una salida como esta:

```
grchere@debian:~/gccwork/src/signals$ ./tp41 &
[1] 2905
main():ingreso en pause()
grchere@debian:~/gccwork/src/signals$
```

donde 2905 (en este caso) se corresponde con el id de proceso (process id) asignado por el kernel del sistema operativo. Observe que, a pesar de ser un proceso ejecutado en background este shell permite que los datos enviados a `stdout` por parte de un proceso en background se impriman en la consola, por ello vemos el `printf()` que pusimos dentro del loop infinito.

Ejecute el comando `ps -l` para ver los procesos que su usuario tiene en ejecución:

```
grchere@debian:~/gccwork/src/signals$ ps -l
F S  UID  PID  PPID C PRI  NI ADDR SZ WCHAN  TTY      TIME CMD
0 S 1000 2709 2707 0 80 0 - 1428 -      pts/0    00:00:01 bash
0 S 1000 2905 2709 0 80 0 - 403 -      pts/0    00:00:00 tp41
0 R 1000 2916 2709 0 80 0 - 872 -      pts/0    00:00:00 ps
grchere@debian:~/gccwork/src/signals$
```

puede observar que `tp41` es un proceso hijo de `bash` (el shell que lo lanzó), obviamente `bash` hace un `fork()` de `tp41` y lo pone en background, tal como Ud. debió hacerlo en su propia versión de shell.

3. Utilice el comando `kill` para enviar una señal al proceso `tp41`, de la siguiente forma (en este caso, para el id de proceso 2905):

```
grchere@debian:~/gccwork/src/signals$ kill -SIGUSR1 2905
grchere@debian:~/gccwork/src/signals$ sig_usr():recibi SIGUSR1 (10)
main():ingreso en pause()
grchere@debian:~/gccwork/src/signals$
```

Puede comprobar que el kernel generó la señal 10 (SIGUSR1), la envió al proceso 2905 (tp41), el proceso capturó la señal a través de la función `sig_usr()`, se ejecutó `sig_usr()`, terminó `sig_usr()`, terminó la función `pause()` (provocó el retorno de la misma, puesto que, al momento de recibir la señal, tp41 estaba ejecutando la función `pause()`). Según la bibliografía que tenemos, en este caso, la función `pause()` debería terminar retornando -1 y asignando la variable global `errno` al valor `EINTR`.

Pruebe el envío de dos o más veces la señal SIGUSR1 o SIGUSR2, si tp41 continua respondiendo, esto implica que su sistema operativo no resetea las señales a su valor por defecto. Si tp41 dejara de responder, implicaría que la captura de la señal sólo aplica a la primera ocurrencia de la misma y si deseamos continuar manipulándola, debemos volver a llamar a la función `signal()` dentro del manipulador para "remanipularla". Modifique y recompile tp41 si fuese necesario para que pueda responder a n ocurrencias de las señales SIGUSR1 y SIGUSR2.

4. Para terminar un proceso, en realidad hay que enviar la señal de terminación (SIGKILL) a dicho proceso; ejecute el comando `kill` correspondiente:

```
grchere@debian:~/gccwork/src/signals$ kill -SIGKILL 2905
grchere@debian:~/gccwork/src/signals$
```

```
[1]+  Killed          ./tp41
grchere@debian:~/gccwork/src/signals$
```

5. Copie tp41.c como tp42.c, modifique tp42.c para adaptarlo según lo dicho en el punto 3 (sólo si fuese necesario). Quite el mensaje que emite el programa en el loop infinito (antes de llamar a la función pause()), ahora cuando el programa reciba la señal SIGUSR2 el programa deberá terminar normalmente (puede hacer esto modificando el loop infinito para que ahora trabaje con una variable global cuyo valor se modifica desde el manipulador de SIGUSR2 y provoca la terminación normal de main()). El programa terminará devolviendo 0 al shell (return 0, exit(0)). Compile y ejecute tp42.

6. Pruebe el correcto funcionamiento de tp42 según lo indicado en el punto anterior: envíe la señal SIGUSR1 varias veces y luego SIGUSR2 y compruebe con el comando ps que el programa termina su ejecución.

```
grchere@debian:~/gccwork/src/signals$ gcc -Wall -o tp42 tp42.c
grchere@debian:~/gccwork/src/signals$ ./tp42 &
[1] 2955
grchere@debian:~/gccwork/src/signals$ kill -SIGUSR1 2955
grchere@debian:~/gccwork/src/signals$ sig_usr():recibi SIGUSR1 (10)

grchere@debian:~/gccwork/src/signals$ kill -SIGUSR1 2955
grchere@debian:~/gccwork/src/signals$ sig_usr():recibi SIGUSR1 (10)

grchere@debian:~/gccwork/src/signals$ kill -SIGUSR2 2955
grchere@debian:~/gccwork/src/signals$ 
[1]+  Done                  ./tp42
grchere@debian:~/gccwork/src/signals$ ps -l
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2709  2707  0  80  0 - 1428 -      pts/0    00:00:02 bash
0 R  1000  2956  2709  0  80  0 -  872 -      pts/0    00:00:00 ps
grchere@debian:~/gccwork/src/signals$
```

7. Copie tp42.c como tp43.c, modifique tp43.c acorde con el siguiente enunciado: La empresa X posee 10 sucursales, este proceso (tp43) acumula el total de ventas de estas sucursales (datos provenientes de otros sistemas, que serán simulados), cada vez que este proceso recibe la señal SIGUSR1 se muestran por pantalla los totales de ventas de cada una de las sucursales (los totales pueden ir cambiando, no necesariamente serán siempre los mismos: puede usar la función random() para generar números aleatorios). Podría obtener una salida como esta:

```
grchere@debian:~/gccwork/src/signals$ ./tp43 &
[1] 2735
grchere@debian:~/gccwork/src/signals$ ps -l
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2679  2677  0  80  0 - 1428 -      pts/0    00:00:00 bash
0 S  1000  2735  2679  0  80  0 -  402 -      pts/0    00:00:00 tp43
0 R  1000  2736  2679  0  80  0 -  872 -      pts/0    00:00:00 ps
grchere@debian:~/gccwork/src/signals$ kill -SIGUSR1 2735
grchere@debian:~/gccwork/src/signals$ 
suc[0]=282.949175
suc[1]=219.742091
```

```

suc[2]=246.506147
suc[3]=281.715697
suc[4]=400.264556
suc[5]=239.175226
suc[6]=208.506593
suc[7]=319.014392
suc[8]=90.060582
suc[9]=249.309716
grchere@debian:~/gccwork/src/signals$ kill -SIGUSR1 2735
suc[0]=286.449696
suc[1]=271.901628
suc[2]=275.976403
suc[3]=454.411340
suc[4]=433.911134
suc[5]=325.277379
suc[6]=236.378879
suc[7]=342.380904
suc[8]=304.577989
suc[9]=296.180029
grchere@debian:~/gccwork/src/signals$ kill -SIGUSR1 2735
grchere@debian:~/gccwork/src/signals$ 
suc[0]=396.601089
suc[1]=452.099608
suc[2]=407.539806
suc[3]=517.983646
suc[4]=570.824441
suc[5]=437.867196
suc[6]=342.375018
suc[7]=551.282750
suc[8]=367.395490
suc[9]=461.827833

grchere@debian:~/gccwork/src/signals$ kill -SIGUSR2 2735
grchere@debian:~/gccwork/src/signals$ kill -SIGUSR2 2735
bash: kill: (2735) - No such process
[1]+  Done                  ./tp43
grchere@debian:~/gccwork/src/signals$ 
grchere@debian:~/gccwork/src/signals$ 

```

**8. Copie tp43.c como tp44.c. Modifique el programa tp44.c acorde con el siguiente enunciado: La Empresa X necesita de un programa que se ejecute una vez por minuto y muestre el total de ventas de cada sucursal. Este programa, al igual que tp43, terminará su ejecución cuando se reciba la señal SIGUSR2.**

Sugerencia: utilizar la función `alarm(nsec)` que permite lanzar la señal `SIGNALRM` cada `nsec` segundos, capturar la señal `SIGNALRM` y en la función de manipulación de esta señal enviar la señal `SIGUSR2` al proceso `tp43` que debe estar ejecutándose concurrentemente con este proceso. Se puede indicar por línea de comando el número de proceso del proceso `tp43`, ya que este dato se necesitará para enviar la señal desde `tp44` a `tp43`.

Podrá obtener una salida como esta (imprimiendo, una vez por minuto los totales de ventas de las sucursales generados por tp43):

```
grchere@debian:~/gccwork/src/signals$ ./tp43 &
```

```

[1] 2831
grchere@debian:~/gccwork/src/signals$ ./tp44 2831 &
[2] 2832
grchere@debian:~/gccwork/src/signals$ 
suc[0]=282.949175
suc[1]=219.742091
suc[2]=246.506147
suc[3]=281.715697
suc[4]=400.264556
suc[5]=239.175226
suc[6]=208.506593
suc[7]=319.014392
suc[8]=90.060582
suc[9]=249.309716
suc[0]=286.449696
suc[1]=271.901628
suc[2]=275.976403
suc[3]=454.411340
suc[4]=433.911134
suc[5]=325.277379
suc[6]=236.378879
suc[7]=342.380904
suc[8]=304.577989
suc[9]=296.180029
grchere@debian:~/gccwork/src/signals$ kill -SIGUSR2 2832
[2]+  Done                  ./tp44 2831
grchere@debian:~/gccwork/src/signals$ ps
  PID TTY      TIME CMD
 2679 pts/0    00:00:00 bash
 2831 pts/0    00:00:00 tp43
 2833 pts/0    00:00:00 ps
grchere@debian:~/gccwork/src/signals$ kill -SIGUSR2 2831
[1]+  Done                  ./tp43
grchere@debian:~/gccwork/src/signals$ ps
  PID TTY      TIME CMD
 2679 pts/0    00:00:00 bash
 2834 pts/0    00:00:00 ps
grchere@debian:~/gccwork/src/signals$
```

9. El diseño propuesto en el punto anterior debería tener de nuestra parte, algunas mínimas consideraciones:

- 9.1) ¿Qué sucede si el id de proceso indicado en la línea de comando no existe?
- 9.2) ¿Qué sucede si tp43 ha dejado de funcionar mientras se está ejecutando tp44?
- 9.3) ¿Qué sucede si el id de proceso indicado es 1 (proceso init)?

Ejecute estos procesos y utilice el shell para probar el comportamiento de estos programas en estos casos.

Copie tp44.c como tp45.c. Modifique el programa tp45.c acorde con las siguientes respuestas:

9.1) usar la señal nula (null signal), la cual se corresponde con el valor 0, para enviarla al proceso indicado, si kill() devuelve -1 y la variable global errno es igual a ESRCH, implicaría que el proceso no existe. Si el proceso indicado no existe, mostrar mensaje de error y terminar programa. Ejemplo:

```
if (kill((pid_t) procid, 0) == -1) {
    fprintf(stderr, "main():Error, %d es un proceso invalido", procid);
    exit(2);
}
```

**9.2)** controlar el retorno de `kill()` en cada envío, si el envío fue satisfactorio `kill()` devolverá 0. En caso de detectar un envío erróneo, emitir mensaje por `stderr` y terminar normalmente la ejecución del programa, para ello puede utilizar la función `raise()` para enviarse a sí mismo la señal `SIGUSR2` que le permita salir normalmente del programa:

```
if (kill((pid_t) idproc, SIGUSR1) != 0) {
    fprintf(stderr, "sig_alarm():Error enviando señal SIGUSR1 a %d\n", idproc);
    raise(SIGUSR2); // envío señal para obtener totales de ventas
}
```

**9.3)** chequear lo recibido por la linea de comandos, si el id de proceso es  $\leq 1$ , mostrar mensaje de error y finalizar programa. Usar valores distintos de retorno de la función `main()` para cada tipo de situación de error que detecte este programa.

Compilar y probar el programa en estas situaciones.

```
grchere@debian:~/gccwork/src/signals$ ./tp45
tp44 <tp43 proc.id>
Donde:
<tp43 proc.id>=nro. de proceso de tp43 que debe estar previamente en ejecución
grchere@debian:~/gccwork/src/signals$ ./tp45 -10
main():-10 no es un id de proceso valido
tp44 <tp43 proc.id>
Donde:
<tp43 proc.id>=nro. de proceso de tp43 que debe estar previamente en ejecución
grchere@debian:~/gccwork/src/signals$ ./tp45 0
main():0 no es un id de proceso valido
tp44 <tp43 proc.id>
Donde:
<tp43 proc.id>=nro. de proceso de tp43 que debe estar previamente en ejecución
grchere@debian:~/gccwork/src/signals$ ./tp45 1
main():1 no es un id de proceso valido
tp44 <tp43 proc.id>
Donde:
<tp43 proc.id>=nro. de proceso de tp43 que debe estar previamente en ejecución
grchere@debian:~/gccwork/src/signals$ ps
  PID TTY      TIME CMD
 2844 pts/0    00:00:01 bash
 2923 pts/0    00:00:00 tp43
 2933 pts/0    00:00:00 ps
grchere@debian:~/gccwork/src/signals$ ./tp45 2930
main():2930 no es un id de proceso valido
tp44 <tp43 proc.id>
Donde:
<tp43 proc.id>=nro. de proceso de tp43 que debe estar previamente en ejecucion
grchere@debian:~/gccwork/src/signals$
```

```
grchere@debian:~/gccwork/src/signals$ ./tp43 &
[1] 2951
grchere@debian:~/gccwork/src/signals$ ./tp45 2951 &
[2] 2952
grchere@debian:~/gccwork/src/signals$ 
suc[0]=282.949175
suc[1]=219.742091
suc[2]=246.506147
suc[3]=281.715697
suc[4]=400.264556
suc[5]=239.175226
suc[6]=208.506593
suc[7]=319.014392
suc[8]=90.060582
suc[9]=249.309716
grchere@debian:~/gccwork/src/signals$ kill -SIGUSR2 2951
grchere@debian:~/gccwork/src/signals$ 
[1]-  Done                  ./tp43
grchere@debian:~/gccwork/src/signals$ 
grchere@debian:~/gccwork/src/signals$ ps
  PID TTY      TIME CMD
 2844 pts/0    00:00:01 bash
 2952 pts/0    00:00:00 tp45
 2954 pts/0    00:00:00 ps
grchere@debian:~/gccwork/src/signals$ 
grchere@debian:~/gccwork/src/signals$ sig_alarm():Error enviando señal SIGUSR1
a 2951
[2]+  Done                  ./tp45 2951
grchere@debian:~/gccwork/src/signals$ ps
  PID TTY      TIME CMD
 2844 pts/0    00:00:01 bash
 2955 pts/0    00:00:00 ps
grchere@debian:~/gccwork/src/signals$
```

### Observaciones en cuanto al uso de señales

1. Es muy importante determinar si su sistema "resetea" o no los manipuladores de señales, acorde con lo realizado en el punto 3. Si un sistema "resetea" el manipulador a su valor por defecto y nosotros "recapturamos" la señal inmediatamente que entramos en la función de manipulación (esto es lo habitual), en general, suele ser la opción correcta, pero no siempre es así. Por ejemplo, la señal SIGCLD funciona distinto que SIGCHLD y su "recapturación" podría provocar un loop infinito: cada vez que usamos la función `signal()` para capturar la señal SIGCLD, el kernel chequea si hay procesos hijos por los cuales esperar y en tal caso, ejecuta el manipulador; si dentro del manipulador volvemos a ejecutar la función `signal()` capturándola otra vez, entraremos en loop. Para evitar esto, debemos capturar la señal luego de una llamada a la función `wait()`. El programa 10.3 [Stevens, Pág. 282] muestra esta situación.
2. Un programa puede funcionar correctamente en un estado de carga de trabajo normal para el sistema operativo (SO), mientras que podría entrar en una condición de carrera (race condition) cuando el SO tenga una mayor carga de trabajo. Esto podría ocurrir con señales que se supone ocurrirán una única vez: por ejemplo, supongamos el uso de la función `alarm()` para ejecutar cierto código dentro de  $n$  segundos y luego llamamos a la función `pause()` para detener el programa hasta la ocurrencia de la señal SIGALRM, ¿Qué sucede si el programa es interrumpido por el SO después de la ejecución de `alarm()` y antes de la ejecución de `pause()`? La lentitud del sistema es tal, que para cuando el SO le vuelve a asignar tiempo de CPU al proceso, resulta que el timer ya ha expirado, con lo cual debe ejecutar el manipulador de SIGALRM (antes de entrar en `pause()`), una vez ejecutado el manipulador, el cual hace su trabajo y retorna; se ejecuta `pause()` esperando la ocurrencia de una señal, si no hay ninguna otra señal capturada en el programa, esta función `pause()` nunca retornará, por lo tanto, el programa quedará bloqueado indefinidamente.
3. Se debe ser muy cuidadoso con el código que se escribe dentro de un manipulador de una señal, por lo general es un código muy pequeño y que activa/desactiva ciertas banderas (flags) que luego son tratados por el programa principal. Tener en cuenta que se trata de una ejecución asincrónica y que existen funciones no re-entrantes, algunas recomendaciones pueden consultarse en la Página 611 del Capítulo 24 "Signal Handling" [LooseSandra, Pág. 611] del manual de referencia de la librería GNU C/C++.
4. Hay una cuestión de permisos vinculados con las señales, no cualquier proceso puede enviar una señal a otro, esto está en relación con quien es el dueño (owner) del proceso y del grupo de proceso que envía la señal y del proceso que la recibe.
5. Las señales son mensajes asincrónicos que se encolan para cada proceso, otra particularidad es que, cuando se encola  $n$  veces una misma señal en un proceso determinado en un lapso de tiempo muy pequeño, dicha señal se suele enviar (puesto que este comportamiento puede variar según el tipo de Unix utilizado) sólo una vez y no tantas veces como se generó.
6. Un mismo programa puede manejar  $n$  señales, con lo cual, cuando se está ejecutando un manipulador de una señal, éste puede ser interrumpido para pasarle el control a otro manipulador. Supongamos que tenemos que realizar la ejecución de una sección crítica de código, se pueden suspender determinadas señales durante la ejecución de dicha sección y restaurarlas luego de haber salido de la sección crítica evitando que el manipulador sea interrumpido (ver ejemplo en Capítulo 10 "Signals" [Stevens]).

7. Tener en cuenta la modificación de variables globales desde `main()` y desde los manipuladores de señales, es conveniente declarar las variables globales con el cualificador `volatile` esto impide que se interrumpa el programa mientras se está actualizando el valor de la variable. También ANSI C define el tipo de dato `sig_atomic_t` para lo mismo, ambos pueden combinarse:

```
static volatile sig_atomic_t bandera;
```

Si desea profundizar el tema, recomendamos la lectura del capítulo 10 "Signals" del libro de Stevens en donde se tratan estos problemas y sus propuestas de solución que exceden el alcance de esta publicación. Existe muchísimo más acerca de señales: un proceso puede bloquear señales, trabajar con un conjunto de señales (signal set), tener asociado una máscara de señales, ver si hay señales pendientes, etc. Muchas de las funciones que Ud. ya conoce hacen uso de señales, tales como: `abort()`, `system()`, `sleep()`, etc.

8. Existen operaciones bloqueantes, por ejemplo, una operación `read()` puede dejar el proceso bloqueado indefinidamente esperando una recepción de datos que nunca llegará. En este caso, podemos utilizar la función `select()` o `poll()` para indicar allí un período de time-out para una operación de lectura, a modo de evitar esta situación. Pero, ¿qué sucede con el resto de funciones bloqueantes que no tienen esta posibilidad? podemos usar la función `alarm()` antes de la función bloqueante, de la siguiente forma:

```
alarm(5); // por ejemplo, establezco time-out de 5 segundos
llamada_a_funcion_bloqueante();
alarm(0); // desactivo alarma
```

entonces, el manipulador de la señal `SIGALRM` será ejecutado en caso de que la función bloqueante no retorne; desde el manipulador de la señal se puede tomar alguna acción. Pero, ¿qué sucede si pretendo volver de esta función de manipulación de la señal a otra parte del código? para ello debo utilizar las funciones `sigsetjmp()` y `siglongjmp()`; estas funciones derivan de las funciones `setjmp()` y `longjmp()`, pero están adaptadas para trabajar con señales. Veamos una nueva versión de `tp45.c` utilizando esta técnica:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
#include <setjmp.h>

void usage();
void sig_usr(int signo);
void sig_alarm(int signo);
int salir=0;
int idproc=0;
sigjmp_buf jmpbuf;

int main(int argc,char *argv[]) { // inicio main()
```

```

if ( argc != 2 ) {
    usage();
    exit(1);
}
idproc=atoi(argv[1]);
if ( idproc <= 1 ) {
    fprintf(stderr,"main():%d no es un id de proceso valido\n",idproc);
    usage();
    exit(3);
}
if ( kill((pid_t) idproc,0) == -1 ) {
    fprintf(stderr,"main():%d no es un id de proceso valido\n",idproc);
    usage();
    exit(2);
}
if ( signal(SIGALRM, sig_alarm) == SIG_ERR) {
    fprintf(stderr,"main():imposible capturar SIGALRM\n");
    exit(4);
}
if ( signal(SIGUSR2, sig_usr) == SIG_ERR) {
    fprintf(stderr,"main():imposible capturar SIGUSR2\n");
    exit(5);
}
if (sigsetjmp(jmpbuf,1)) {
    printf("main():dentro de 60 segundos vuelvo a pedir totales\n");
}
alarm(60);
for ( ;!salir; ) {
    pause();
}
return 0;
} // fin main()

void sig_usr(int signo) { /* signo=nro de señal */
if (signo == SIGUSR2)  salir=1;
else printf("sig_usr():recibi señal %d\n", signo);
return;
}
void sig_alarm(int signo) { /* signo=nro de señal */
if (kill((pid_t) idproc,SIGUSR1) != 0) {
    fprintf(stderr,"sig_alarm():Error enviando señal SIGUSR1 a %d\n",idproc);
    raise(SIGUSR2); // envío señal de salida
}
siglongjmp(jmpbuf,1);
}
void usage() {
    printf("tp44 <tp43 proc.id>\n");
    printf("Donde:\n");
    printf("<tp43 proc.id>=nro. de proceso de tp43 que debe estar previamente\n");
    en ejecucion\n");
}

```

la salida a obtener por este programa sería (suponiendo que el programa que emite los totales de ventas de las sucursales se está ejecutando previamente con el id de proceso 3079):

```
grchere@debian:~/gccwork/src/signals$ ./tp46 3079
main():dentro de 60 segundos vuelvo a pedir totales!
```

```
suc[0]=1498.857343
suc[1]=1204.173758
suc[2]=1590.140845
suc[3]=2070.052891
suc[4]=1668.643526
suc[5]=1735.123461
suc[6]=1237.069390
suc[7]=1876.513227
suc[8]=1316.073510
suc[9]=1736.023419
main(): dentro de 60 segundos vuelvo a pedir totales!
suc[0]=1514.862496
suc[1]=1408.207045
suc[2]=1601.421418
suc[3]=2182.057773
suc[4]=1706.484476
suc[5]=1786.676463
suc[6]=1408.395217
suc[7]=2033.849564
suc[8]=1457.069481
suc[9]=1943.772091
....
```

analizando la salida, podrá deducir el funcionamiento de `sigsetjmp()`, `siglongjmp()`.

## Capítulo V. Tuberías I (Pipes).

**Objetivo:** Desarrollar una serie de programas que permitan comprobar distintas funcionalidades vinculadas con la gestión de tuberías (pipes) y su aplicación.

### Introducción

Las tuberías o pipes son una de las más viejas formas de comunicación entre procesos (IPC, inter process communication) provistas por Unix, podemos encontrar prototipos de sus funciones en `unistd.h`, sin embargo, las tuberías tienen dos limitaciones:

1. Son half-duplex, es decir, el flujo de datos es unidireccional.
2. Pueden ser usados entre procesos que tengan un ancestro en común.

Normalmente una tubería es creada por un proceso padre, quien llama a `fork()`, opcionalmente llama a `exec()` y luego la tubería es utilizada para comunicar padre con hijo.

1. Problema: La empresa X posee 10 sucursales, este proceso (tp51) le permite al usuario ingresar por teclado un código de sucursal (número comprendido entre 0 y 9) y muestra por pantalla el total de ventas de dicha sucursal (el dato de venta es proveniente de otro sistema que será simulado utilizando la función `random()`). Si el usuario ingresa un código de sucursal inválido, el programa devolverá -1 como total de ventas para dicha sucursal. La interacción con el programa podría ser la siguiente:

```
grchere@debian:~/gccwork/src/pipes$ ./tp51
1
suc[1]=180.428938
grchere@debian:~/gccwork/src/pipes$ ./tp51
99
suc[99]=-1.000000
grchere@debian:~/gccwork/src/pipes$ ./tp51
hola
suc[0]=180.428938
grchere@debian:~/gccwork/src/pipes$ ./tp51
-1
suc[-1]=-1.000000
grchere@debian:~/gccwork/src/pipes$
```

Sugerencia: para el ingreso por teclado se recomienda utilizar la función `read()`, sobre el descriptor de archivo `STDIN_FILENO` quien representa la entrada estandar. Compile el programa como tp51 (tp51.c).

2. Problema: La empresa X está muy conforme con su programa interactivo que muestra los totales de ventas por sucursal, dicho programa (tp51) es muy viejo y fue desarrollado por el consultor Y. No se tienen los fuentes de dicho programa, ni tampoco se sabe cómo accede a la información de los totales de ventas, ni se sabe dónde se encuentra el consultor Y. Ahora la empresa X lo contrata a Ud. para que realice un programa que obtenga los totales de ventas de cada una de las sucursales y los

muestre en pantalla (o genere un grafico de barras, etc.), de forma tal, que la salida de este programa también pueda ser redirigida a un archivo. Un ejemplo de ejecución de este programa podría ser:

```
grchere@debian:~/gccwork/src/pipes$ ./tp55
suc[0]=185.698899
suc[1]=48.488474
suc[2]=17.118268
suc[3]=201.864525
suc[4]=171.574494
suc[5]=33.204262
suc[6]=109.972127
suc[7]=80.015225
suc[8]=49.890014
suc[9]=20.500763
grchere@debian:~/gccwork/src/pipes$
```

Otra posibilidad es redirigiendo la salida:

```
grchere@debian:~/gccwork/src/pipes$ ./tp55 > salida.txt
grchere@debian:~/gccwork/src/pipes$ cat salida.txt
suc[0]=169.214392
suc[1]=31.285207
suc[2]=107.657237
suc[3]=185.225414
suc[4]=47.221871
suc[5]=124.386318
suc[6]=94.252531
suc[7]=63.131976
suc[8]=140.837584
suc[9]=110.428725
grchere@debian:~/gccwork/src/pipes$
```

Para lograr esto, se pueden utilizar tuberías o pipes. La idea sería tomar al programa tp51 como una "caja negra" de la cual sólo sabemos sus entradas posibles y por ende, sus salidas posibles<sup>4</sup>. Podemos crear un nuevo programa, el cual hará un `fork()`, luego un `exec()` de tp51 (quien se transformará en hijo de este nuevo programa) y creará una tubería entre el nuevo programa y su hijo, simulando una interacción que será "*no humana*", la interacción será "*entre procesos*": el padre "*escribirá*" en la entrada standard del hijo y "*leerá*" desde la salida standard del hijo. De esta forma, se podrán obtener los datos de cada una de las sucursales, juntarlos y por último imprimirlas todos juntos (graficarlos, etc.).

Iremos paso a paso, primero crearemos el programa tp52 que hará un `fork()` y quedará esperando hasta la finalización del hijo (Ej.: `waitpid(pid, &status, WUNTRACED);`). Introducimos algunos mensajes:

```
grchere@debian:~/gccwork/src/pipes$ ./tp52
main(): hago fork()!
hijo!
padre esperando hijo!
fin padre esperando hijo!
```

<sup>4</sup> Acorde con el Modelo Determinista de la programación.

```
grchere@debian:~/gccwork/src/pipes$
```

Sugerencia: copie<sup>5</sup> código del Capítulo II (proyecto de shell primitivo), observe la parte del código en donde se hace un `fork()`, `exec()`, etc.. Compile el programa como `tp52 (tp52.c)`.

3. Copie `tp52.c` en `tp53.c`. Modifique `tp53.c`, ahora, luego de hacer el `fork()`, en la parte de código correspondiente al hijo, vamos a agregar una llamada a `execvp()` para reemplazar la imagen del proceso actual por la del proceso `tp51`, para ello podemos usar un código como este:

```
char *prog[] = {"./tp51", "./tp51", NULL};
if (execvp(prog[0], prog) == -1) {
    printf("main():Error en execvp()\n");
}
```

Ahora el proceso hijo se transformará en `tp51`, provocando el ingreso por teclado y la impresión del total de ventas de la sucursal ingresada por teclado:

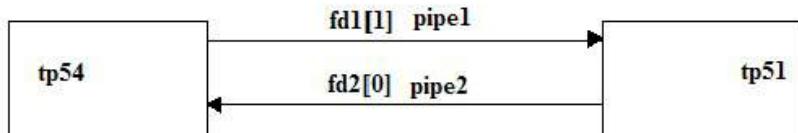
```
grchere@debian:~/gccwork/src/pipes$ ./tp53
main(): hago fork()
hijo1
padre esperando hijo1
2
suc[2]=180.428938
fin padre esperando hijo1
grchere@debian:~/gccwork/src/pipes$
```

4. Copie `tp53.c` en `tp54.c`. Hemos logrado que `tp53` ejecute a `tp51`, no obstante, el problema es que todo el proceso continúa siendo interactivo, ahora necesitamos establecer una comunicación entre procesos y sin intervención "humana". Podemos establecer un pipe de comunicación entre `tp54` y `tp51`, adicionalmente, el proceso hijo deberá conectar un extremo del pipe con su `stdin` (entrada Standard, descriptor `STDIN_FILENO`) y otro extremo del pipe con su `stdout` (salida Standard, descriptor `STDOUT_FILENO`).

La descripción anterior nos indica que `tp54` y `tp51` actúan como *co procesos*, en donde uno de los procesos corre en *background* y conecta su `stdin` y `stdout` a otro proceso que lo leerá y escribirá según corresponda.

Trabajando con pipes unidireccionales (half-duplex) el esquema sería:

<sup>5</sup> Aquí se debería utilizar el término "reutilizar", pero ello implicaría la creación y compartición de librerías, lo cual está fuera del alcance de esta publicación, debido a ello, en la mayoría de los casos se propondrá copiar el código, lo cual es una práctica no recomendable. Instamos a que Ud. investigue el tema y cree sus propias librerías para reusar este código.



La función `pipe()` (`unistd.h`) nos permite crear un pipe representado por un arreglo de enteros que contiene dos descriptores de archivos<sup>6</sup>: uno sobre el cual podemos grabar (posición 1 del arreglo) y otro sobre el cual podemos leer (posición 0 del arreglo). Necesitamos dos pipes para comunicar estos procesos. La función `pipe()` devuelve un valor < 0 cuando hay un error:

```
int fd1[2],fd2[2];
// crear pipes
if ( pipe(fd1) < 0 || pipe(fd2) < 0 ) {
    fprintf(stderr,"main():Error creando pipes!\n");
    exit(1);
}
```

Hay dos extremos de los pipes que no serán utilizados por el padre: `fd1[0]` (tp54 no pretende leer del `stdin` de tp51) y `fd2[1]` (tp54 no pretende grabar en el `stdout` de tp51). Por lo tanto estos extremos pueden ser cerrados por el proceso padre (tp54.c):

```
close(fd1[0]);
close(fd2[1]);
```

Idem anterior, hay dos extremos que no serán utilizados por el proceso hijo: `fd1[1]` (tp51 no pretende escribir en su `stdin`) y `fd2[0]` (tp51 no pretende leer de su `stdout`). Por lo tanto estos extremos pueden ser cerrados por el proceso hijo (tp51.c):

```
close(fd1[1]);
close(fd2[0]);
```

El hijo deberá conectar su `stdin` con el pipe `fd1[0]` y su `stdout` con el pipe `fd2[1]`, para ello podemos usar la función `dup2()`, ya utilizada en el proyecto del shell del capítulo II:

```
...
if (fd1[0] != STDIN_FILENO) {
    if ( dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
        fprintf(stderr,"ejecutoHijo():dup2 error en stdin\n");
    close(fd1[0]);
}
```

<sup>6</sup> Los descriptores de archivos son datos de tipo entero, por lo tanto, nada prohíbe que podamos almacenarlos en un arreglo de enteros.

```

if (fd2[1] != STDOUT_FILENO) {
    if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
        fprintf(stderr, "ejecutoHijo():dup2 error en stdout\n");
    close(fd2[1]);
}
...

```

Una vez creados los pipes y conectados de la forma apropiada, sólo nos queda leer y escribir de ellos, tarea a ser realizada por el proceso padre. Para leer y escribir se pueden utilizar las funciones `read()` y `write()` (`unistd.h`) puesto que el contenido de los arreglos `fd1` y `fd2` son descriptores de archivos<sup>7</sup>. Por ejemplo, para pedir y obtener el total de ventas de la sucursal 2:

```

...
strcpy(buffer, "2\n"); // solicito sucursal 2
int l = strlen(buffer);
int n = write(fd1[1], buffer, l);
if (n != l) printf("pidoVentas:grabe %d y esperaba grabar %d en
suc=2\n", n, l);
n = read(fd2[0], buffer, MAX_BUFFER);
buffer[n] = '\0'; // no olvidar finalizar string con \0
printf("main():lei de prog. hijo: [%s]", buffer);
...

```

Luego de enviar y recibir los datos, el proceso padre continua con la espera de la finalización del hijo, compilamos y probamos `tp4.c`, podríamos obtener la siguiente salida:

```

grchere@debian:~/gccwork/src/pipes$ ./tp54
main(): hago fork()
hijo:
main():lei de prog. hijo: [suc[1]=180.428938
]
padre esperando hijo:
fin padre esperando hijo:
grchere@debian:~/gccwork/src/pipes$
```

Puede observarse que no hay una interacción *"humana"* sino entre procesos. En este caso, `tp54` solicita el código de la sucursal 1 (envía "1\n") y recibe como respuesta "suc[1]=180.428938\n"; exactamente los caracteres que ingresa el operador como entrada de `tp51` y los caracteres que `tp51` emite como salida.

5. Copie `tp54.c` en `tp55.c`. Ahora debemos agregar un loop de 0 a 9 para solicitar las ventas de las 10 sucursales. El diseño general del programa es el siguiente:

<sup>7</sup> Las funciones `open()`, `read()`, `write()`, `seek()`, `close()` trabajan con descriptores de archivos (tipo de dato: `int`) y permiten hacer entrada-salida sin buffers; mientras que las funciones `fopen()`, `fread()`, `fwrite()`, `fseek()`, `fclose()` trabajan con streams (tipo de dato: `FILE *`) y permiten hacer entrada-salida con buffers.

inicio loop sucursal	
creo pipes	
hago fork()	
<b>proceso padre</b>	<b>proceso hijo</b>
cierro pipes innecesarios	cierro pipes innecesarios
envio codigo sucursal	conecto stdin
recibo total de ventas	conecto stdout
imprimo total de ventas	exec de tp51
espero finalización de hijo	
fin loop sucursal	

Una salida posible del programa tp55:

```
grchere@debian:~/gccwork/src/pipes$ ./tp55
main(): hago fork()
hijo1
main(): lei de prog. hijo: [suc[0]=63.149707
]padre esperando hijo1
fin padre esperando hijo1
main(): hago fork()
hijo1
main(): lei de prog. hijo: [suc[1]=31.934346
]padre esperando hijo1
fin padre esperando hijo1
main(): hago fork()
hijo1
main(): lei de prog. hijo: [suc[2]=109.752318
]padre esperando hijo1
fin padre esperando hijo1
main(): hago fork()
hijo1
main(): lei de prog. hijo: [suc[3]=185.506887
]padre esperando hijo1
fin padre esperando hijo1
main(): hago fork()
hijo1
main(): lei de prog. hijo: [suc[4]=155.639549
]padre esperando hijo1
fin padre esperando hijo1
main(): hago fork()
hijo1
main(): lei de prog. hijo: [suc[5]=125.171518
]padre esperando hijo1
fin padre esperando hijo1
main(): hago fork()
hijo1
main(): lei de prog. hijo: [suc[6]=94.452528
]padre esperando hijo1
fin padre esperando hijo1
main(): hago fork()
hijo1
main(): lei de prog. hijo: [suc[7]=173.261329
]padre esperando hijo1
fin padre esperando hijo1
main(): hago fork()
```

```
hijo1
main(): lei de prog. hijo: [suc[8]=142,342423
|padre esperando hijo1
fin padre esperando hijo1
main(): hago fork()
hijo1
main(): lei de prog. hijo: [suc[9]=112,486605
|padre esperando hijo1
fin padre esperando hijo1
grchere@debian:~/gccwork/src/pipes$
```

Ahora modificamos tp55, quitamos los mensajes que están de mas:

```
grchere@debian:~/gccwork/src/pipes$ ./tp55
suc[0]=185.698899
suc[1]=48.488474
suc[2]=17.118268
suc[3]=201.864525
suc[4]=171.574494
suc[5]=33.204262
suc[6]=109.972127
suc[7]=80.015225
suc[8]=49.890014
suc[9]=20.500763
grchere@debian:~/gccwork/src/pipes$
```

Redirigiendo la salida:

```
grchere@debian:~/gccwork/src/pipes$ ./tp55 > salida.txt
grchere@debian:~/gccwork/src/pipes$ cat salida.txt
suc[0]=169.214392
suc[1]=31.285207
suc[2]=107.657237
suc[3]=185.225414
suc[4]=47.221871
suc[5]=124.386318
suc[6]=94.252531
suc[7]=63.131976
suc[8]=140.837584
suc[9]=110.428725
grchere@debian:~/gccwork/src/pipes$
```

Volvemos a modificar tp55, debemos capturar la señal SIGPIPE la cual será lanzada cuando grabamos en un pipe en donde su lector ya ha terminado. Implementamos la captura, en caso de producirse la señal SIGPIPE, enviamos mensaje de error y retornamos 2 al shell, comprobamos nuevamente el funcionamiento de tp55:

```
grchere@debian:~/gccwork/src/pipes$ ./tp55
suc[0]=189.557790
suc[1]=158.801467
suc[2]=127.814084
suc[3]=97.586370
suc[4]=65.977004
suc[5]=144.760503
```

```
suc[6]=6.487983
suc[7]=83.616891
suc[8]=159.872828
suc[9]=21.453911
grchere@debian:~/gccwork/src/pipes$
```

### En cuanto al uso de pipes

Lo visto en este texto es sumamente elemental en cuanto a pipes, también existen implementaciones full-duplex de pipes, la posibilidad de duplicar streams usando pipes, el uso de FIFO's o "named pipes" que también pueden ser un medio muy propicio para la programación client-server. En este texto se optó por un camino algo más largo pero más detallado para la explicación y comprensión de los pipes, pero cabe aclarar que existen las funciones `popen()` y `pclose()` (`stdio.h`) que, si bien tienen algunas limitaciones, son muy prácticas y fáciles de usar para la implementación de co procesos.

## Capítulo VI. Tuberías II (FIFO's) .

**Objetivo:** Desarrollar una serie de programas que permitan comprobar distintas funcionalidades vinculadas con la gestión de tuberías (pipes) de tipo FIFO y su aplicación. Introducir al desarrollo de programas servidores y clientes. Estimular una programación modular, con código reusable. Generalizar funciones. Usar técnicas avanzadas de I/O y manejo de punteros.

### Introducción

En Capítulo V Tuberías I (Pipes) se habló de dos limitaciones en cuanto a los pipes, para salvar el problema de unidireccionalidad de los pipes se pueden utilizar los *stream pipes*, mientras que para salvar el problema de que los pipes requieren de un ancestro en común, se pueden utilizar pipes de tipo FIFO (first-input-first-output) también llamados *named pipes*.

Un programa servidor puede crear un FIFO que sea conocido por todos sus clientes utilizando la función `mkfifo()`, este FIFO queda asociado a un archivo dentro del sistema de archivos del SO, por ejemplo: `/tmp/myfifo` se pueden realizar operaciones de I/O sobre dicho FIFO para comunicar procesos como si se tratase de un archivo más del sistema de archivos y sin necesidad de que los procesos tengan una relación padre-hijo.

Las operaciones de I/O se considerarán atómicas siempre y cuando la operación involucre un número de bytes menor que `PIPE_BUF` (valor predefinido para el SO).

Esto permite el desarrollo de aplicaciones client-server a través del uso de *named pipes* o FIFO's; un servidor puede crear un FIFO, abrirlo de modo lectura, mientras que los clientes abrirán ese mismo FIFO de modo grabación. Los clientes "grabarán" sus peticiones y el servidor "leerá" las peticiones para su proceso. Sin embargo, queda una cuestión por resolver: ¿Cómo puede el servidor saber cuales son sus clientes? ¿Cómo podemos implementar un "ida y vuelta" entre cliente y servidor? El cliente envía la petición, pero necesita enterarse si la misma fue procesada o no y en tal caso, el resultado del proceso.

Una posible solución podría ser: los clientes crearán un FIFO en una ubicación conocida tanto por cliente como por servidor, el nombre del FIFO incluirá el número de proceso del cliente (de modo tal, que sea un FIFO distinto para cada cliente); los clientes, junto con su petición, enviarán su id de proceso al servidor; el servidor tomará su id de proceso y grabará la respuesta de su petición en el FIFO previamente abierto por cliente. De esta forma, tendremos la posibilidad de implementar una "vuelta" por parte del servidor; en el párrafo anterior explicamos una forma de "ida".

Una vez que el cliente ha terminado de enviar y recibir peticiones, cerrará el pipe con el servidor y borrará el pipe creado para leer las respuestas del mismo.

Una vez que el servidor ha terminado de trabajar, cerrará el pipe creado para atender a sus clientes y lo borrará del sistema de archivos.

De esta forma, tanto clientes como servidores dejarán "limpio" al sistema de archivos.

1. El Analista de Sistemas Y de la Empresa X ha podido comprobar las ventajas de una arquitectura client-server. Entonces ahora pretende implementar un servidor que pueda atender a clientes. Este servidor se encargará de enviar a los clientes los totales de ventas de las sucursales que los clientes requieran. El diseño de este cliente y servidor, responde a lo expuesto en la sección anterior.

Crear la carpeta `fifo` para almacenar este proyecto.

Dentro de la carpeta `fifo`, crear la carpeta `v1` para almacenar la versión 1 de este proyecto.

Dentro de `v1` vamos a crear el programa `tp61.c`.

El servidor tendrá el siguiente diseño:

```
main()
    crear FIFO en sistema de archivos
    si pudo crear FIFO Ok entonces
        otorgar permisos a FIFO en sistema de archivos
        abrir FIFO de R/W
        crear buffer de MAX_BUFFER caracteres
        mientras VERDAD (loop infinito)
            leer buffer de FIFO
            si lectura FIFO Ok entonces
                imprimir buffer
            fin si
        fin mientras
        cerrar FIFO
        borrar FIFO del sistema de archivos
    sino
        error en creacion de FIFO!
    fin si
fin main()
```

Este servidor permitirá comprobar si recibimos los mensajes que enviarán los clientes, pero requerirá de que terminemos anormalmente el proceso, puesto que quedará atrapado en un loop infinito. Comencemos a desmenuzar este código.

Ejemplo de crear FIFO:

```
int mf = mkfifo (MY_FIFO, O_CREAT);
```

donde `MY_FIFO` está definido dentro de `myfifo.h` el cual es un archivo de cabecera compartido entre cliente y servidor para indicar allí el nombre del FIFO, en este caso:

```
#define MAX_BUFFER 255          // deberia ser menor que PIPE_BUF
#define MY_FIFO "/tmp/myfifo"
```

¿Cómo saber si el FIFO se pudo crear Ok?:

```
if ( mf == 0 || errno == EEXIST ) {
    <<FIFO creado Ok>>
} else {
    <<error en creacion de FIFO >>
}
```

¿Cómo otorgar permisos a un archivo?, usando la función chmod() (análoga al comando chmod de Unix/Linux):

```
chmod (MY_FIFO, 0777);
```

en este caso, otorgamos todo tipo de permisos (rwx) para el dueño del archivo, el grupo al que pertenece y el resto de usuarios.

Abrir un FIFO o cualquier archivo:

```
int fdi = open (MY_FIFO, O_RDWR, 0777);
```

open() devuelve un descriptor de archivo (file descriptor) a través del cual podemos hacer todas las operaciones de I/O.

Crear un buffer no es mas que reservar memoria automática (en este caso) para un conjunto de bytes en donde almacenaremos la petición de los clientes:

```
char buffer[MAX_BUFFER+1];
```

donde MAX\_BUFFER esta definido dentro de myfifo.h.

Leer de un archivo:

```
int nread = read(fdi,buffer,MAX_BUFFER);
```

donde nread indica la cantidad efectiva de bytes leídos, cuidando de no sobrepasar la capacidad de buffer. Si nread es igual a -1, indica que hubo un error en la lectura.

Para cerrar un FIFO o cualquier archivo:

```
close(fdi);
```

Para borrar el FIFO del sistema de archivos, usar la función remove():

```
remove (MY_FIFO);
```

Consulte el manual de referencia de la librería GNU C para más información [Loosemore et. al.].

Algunos de los headers que deberá utilizar para implementar este servidor y demás programas:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <string.h>
#include <limits.h>

#include "myfifo.h" // librería propia del servidor de la Empresa X
```

Implemente y compile tp6.1. Ejecute tp61 en background, puesto que quedará bloqueado en la función read().

2. Estando en ejecución tp61 verificar el directorio /tmp, ejecutar el comando ls -l /tmp y chequear si allí se puede ver el archivo asociado con el FIFO del servidor. Recuerde que tp61 quedará bloqueado, deberá usar el comando kill para enviar la señal de terminación de proceso a tp61: kill -SIGKILL <id de proceso tp61> también puede probar: kill -SIGINT <id de proceso tp61>.

3. Desarrolle un programa que será el cliente que realice peticiones al servidor del punto anterior. Recuerde que siempre debe ejecutarse primero el servidor y luego el cliente. El diseño del cliente sería el siguiente:

```
main()
    abrir FIFO de W
    crear buffer de MAX_BUFFER caracteres
    para <suc> 0 hasta 9 hacer
        armar en buffer mensaje a enviar al servidor para <suc>
        grabar buffer en FIFO
        imprimir buffer indicado lo enviado al servidor
    fin para
    cerrar FIFO
fin main()
```

Según lo indicado en la introducción, debemos enviar como dato al servidor el id de proceso del cliente y el número de sucursal de la cual el cliente requiere su total de ventas. Para facilitar el parsing de este mensaje por parte del servidor, se podría optar por el siguiente formato:

"<id proceso cliente/sucursal>"

una forma posible de armar el buffer, sería:

```

...
char buffer[MAX_BUFFER+1];
...
int suc;
for(suc=0;suc<10;suc++) { // solicito total ventas de sucursal suc
    sprintf(buffer,MAX_BUFFER,"%d/%d", (int) getpid(),suc);
    ...
}
...

```

La función `snprintf()` se encuentra en el header `stdio.h` y es similar a `printf()` aunque en este caso la salida se envía a un string en vez de enviarla a un dispositivo de salida y además de ello, se establece un número máximo de caracteres a enviar (a diferencia de `sprintf()`, la cual no impone ningún límite, pudiendo provocar un desbordamiento del string destino, por ello debe usarse con cuidado de no superar la cantidad de caracteres previamente asignados (allocated)).

Implemente el programa `tp62.c` y complejelo como `tp62`. Ejecute `tp62` en foreground (recuerde que antes debe estar ejecutándose `tp61`). Recomendamos que utilice dos consolas, una para ejecutar el servidor y otra para ejecutar el cliente a modo de no confundirse con los mensajes. Observe los mensajes enviados por `tp62` y los recibidos por `tp61`:

```

consola servidor:

grchere@debian:~/gccwork/src/fifo/v1$ ./tp61 &
[1] 2813
grchere@debian:~/gccwork/src/fifo/v1$ main():inicio servidor FIFO!
main():Tamaño maximo de operaciones atomicas sobre PIPES: 4096
main():Tamaño maximo de buffer de servidor: 255

grchere@debian:~/gccwork/src/fifo/v1$ 
main():lei: [<2838/0>] de FIFO [/tmp/myfifo]
main():lei: [<2838/1>] de FIFO [/tmp/myfifo]
main():lei: [<2838/2>] de FIFO [/tmp/myfifo]
main():lei: [<2838/3>] de FIFO [/tmp/myfifo]
main():lei: [<2838/4>] de FIFO [/tmp/myfifo]
main():lei: [<2838/5>] de FIFO [/tmp/myfifo]
main():lei: [<2838/6>] de FIFO [/tmp/myfifo]
main():lei: [<2838/7>] de FIFO [/tmp/myfifo]
main():lei: [<2838/8>] de FIFO [/tmp/myfifo]
main():lei: [<2838/9>] de FIFO [/tmp/myfifo]

grchere@debian:~/gccwork/src/fifo/v1$ 
grchere@debian:~/gccwork/src/fifo/v1$ ps
  PID TTY      TIME CMD
 2794 pts/0    00:00:00 bash
 2813 pts/0    00:00:00 tp61
 2839 pts/0    00:00:00 ps
grchere@debian:~/gccwork/src/fifo/v1$ 
[1]+  Terminated                  ./tp61
grchere@debian:~/gccwork/src/fifo/v1$ 
grchere@debian:~/gccwork/src/fifo/v1$ ls -l /tmp

```

```

total 12
drwx----- 2 grchere grchere 4096 2009-06-25 14:17 gconfd-grchere
drwx----- 2 grchere grchere 4096 2009-06-25 14:16 keyring-SehzsZ
prwxrwxrwx 1 grchere grchere 0 2009-06-25 14:36 myfifo
drwx----- 2 grchere grchere 4096 2009-06-25 14:17 orbit-grchere
prwxr-xr-x 1 grchere grchere 0 2009-06-25 14:33 ScITE.2822.in
grchere@debian:~/gccwork/src/fifo/v1$
```

**consola cliente:**

```

grchere@debian:~/gccwork/src/fifo/v1$ ./tp62
main(): inicio cliente FIFO!
main(): envie 8 bytes a servidor [<2038/0>]
main(): envie 8 bytes a servidor [<2038/1>]
main(): envie 8 bytes a servidor [<2038/2>]
main(): envie 8 bytes a servidor [<2038/3>]
main(): envie 8 bytes a servidor [<2038/4>]
main(): envie 8 bytes a servidor [<2038/5>]
main(): envie 8 bytes a servidor [<2038/6>]
main(): envie 8 bytes a servidor [<2038/7>]
main(): envie 8 bytes a servidor [<2038/8>]
main(): envie 8 bytes a servidor [<2038/9>]
main(): fin cliente FIFO!
grchere@debian:~/gccwork/src/fifo/v1$
```

```

grchere@debian:~/gccwork/src/fifo/v1$ kill -SIGTERM 2813
grchere@debian:~/gccwork/src/fifo/v1$
```

4. Debemos encontrar un medio para que el servidor no quede permanentemente bloqueado en la función `read()`, a la espera -tal vez interminable- de un cliente, una forma posible sería multiplexando las operaciones de I/O (I/O multiplexing) , es una técnica avanzada de I/O que puede lograrse a través de la función `select()` o bien de la función `poll()`. Permitirá darle a la función `read()` cierto time-out, es decir, que no quede bloqueada por siempre. La idea sería invocar a la función `poll()`, ésta nos dirá cuando hay una operación de I (input) pendiente (devuelve 1) y recién allí invocaríamos a `read()` con un time-out preestablecido. Luego de atendido el cliente, volveríamos nuevamente a la función `poll()`, el esquema es el siguiente:

```

main()
    crear FIFO en sistema de archivos
    si pude crear FIFO Ok entonces
        otorgar permisos a FIFO en sistema de archivos
        abrir FIFO de R/W
        crear buffer de MAX_BUFFER caracteres
        mientras VERDAD (loop infinito)
            polling de FIFO con 10 segundos de time-out
            si hay lectura pendiente entonces
                leer buffer de FIFO
                si lectura FIFO Ok entonces
                    imprimir buffer
                fin si
            sino
                error en polling
            fin si
        fin mientras
        cerrar FIFO
        borrar FIFO del sistema de archivos
```

```

sino
    error en creacion de FIFO!
fin si
fin main()

```

¿Qué requiere la función poll() ? Básicamente, hay que indicarle sobre cuáles descriptores de archivos (file descriptors) estamos interesados y sobre cuáles eventos le puedan ocurrir a esos descriptores. Esta información se proporciona a través de la estructura pollfd. Debemos agregar los headers: <stropts.h> y <poll.h> requeridos por esta función. Para nuestro caso, lo podemos hacer de la siguiente forma:

```

...
// polling info
int rpoll;
struct pollfd fdpoll[1];
fdpoll[0].fd = fdi; // fdi es el descriptor de archivo de MYFIFO
fdpoll[0].events = POLLIN | POLLPRI;
...
while(1) { // loop infinito
    rpoll = poll(fdpoll,1L,10000); // time out de 10 segundos para read()
...
}
...

```

Copie tp61.c como tp63.c, implemente el polling propuesto, compile como tp63 y ejecute en background. Incluya varios mensajes en esta nueva versión del servidor para verificar si queda bloqueado como antes o no. Puede continuar usando el mismo programa cliente.

5. Si bien hemos avanzado en cuanto al bloqueo, no obstante el servidor continúa dentro de un bucle infinito. Como Ud. ya sabe sobre señales, se propone implementar una forma elegante de terminar este servidor utilizando la señal SIGUSR2, tal como ya lo ha hecho antes. La idea es que cuando el servidor reciba la señal SIGUSR2, éste salga del loop infinito y termine normalmente. Debemos capturar la señal SIGUSR2, y cuando ésta se produzca, en la función que manipula esta señal, debemos poner la variable global salir en 1 (true, verdad, activado) y ahora el loop infinito pasará a ser while(!salir) { ... } se puede declarar a salir como una variable entera fuera de la función main(), puesto que debe ser global.

Copie tp63.c como tp64.c, implemente la manipulación de la señal, la variable global, cambie el loop infinito, Compile como tp64, pruebe el sistema, podría obtener una salida como esta:

```

consola server:
grchere@debian:~/gccwork/src/fifo/v1$ ./tp64 &
main():inicio servidor FIFO!
main():para salir envie señal SIGUSR2 a proceso 2847
main():Tamaño maximo de operaciones atomicas sobre PIPES: 4096
main():Tamaño maximo de buffer de servidor: 255
[1] 2847
grchere@debian:~/gccwork/src/fifo/v1$
```

```

grchere@debian:~/gccwork/src/fifo/v1$ main():lei:
[<2849/0><2849/1><2849/2><2849/3><2849/4><2849/5><2849/6><2849/7><2849/8><2849
/9>] de FIFO [/tmp/myfifo]

grchere@debian:~/gccwork/src/fifo/v1$ 

consola cliente:
grchere@debian:~/gccwork/src/fifo/v1$ ./tp62
main():inicio cliente FIFO
main():envie 8 bytes a servidor [<2849/0>]
main():envie 8 bytes a servidor [<2849/1>]
main():envie 8 bytes a servidor [<2849/2>]
main():envie 8 bytes a servidor [<2849/3>]
main():envie 8 bytes a servidor [<2849/4>]
main():envie 8 bytes a servidor [<2849/5>]
main():envie 8 bytes a servidor [<2849/6>]
main():envie 8 bytes a servidor [<2849/7>]
main():envie 8 bytes a servidor [<2849/8>]
main():envie 8 bytes a servidor [<2849/9>]
main():fin cliente FIFO
grchere@debian:~/gccwork/src/fifo/v1$ 

```

Al enviar la señal SIGUSR2 al proceso tp64 debería suceder:

```

consola cliente:
grchere@debian:~/gccwork/src/fifo/v1$ kill -SIGUSR2 2847
grchere@debian:~/gccwork/src/fifo/v1$ 

consola server:
grchere@debian:~/gccwork/src/fifo/v1$ main():recibi señal de salida
main():fin servidor FIFO| retorno=0

[1]+ Done                  ./tp64
grchere@debian:~/gccwork/src/fifo/v1$ 

```

6. Ahora debemos hacer que el servidor pueda obtener un total de ventas para cada sucursal pedida por los clientes, para ello, vamos a simular dichos valores, de la misma forma que lo hicimos en tp51.c. Podríamos tener un arreglo global para los totales: double ventas[10]; y agregar la función init() que se llamaría la comienzo del servidor, inicializando el vector:

```

void init() {
    srand(time(0));
    int i;
    for(i=0;i<10;i++) ventas[i]=((double) random())/10000000.0;
}

```

Otro problema es que el servidor debe interpretar las peticiones de los clientes (*parsing*) para ello implementamos la función parse() y luego, una vez que sabemos lo que el cliente quiere, podemos llamar a la función proceso():

```
// analiza buffer enviado por clientes para determinar id de proceso cliente
```

```

// y nro de sucursal solicitada
// formato de buffer: "<id proceso/nro sucursal>...<id proceso/nro sucursal>"
void parse(char *buffer) {
    char id[MAX_BUFFER];
    char suc[MAX_BUFFER];
    int iid,isuc;
    char *p;
    while(*buffer) {
        memset(id,0,MAX_BUFFER);
        memset(suc,0,MAX_BUFFER);
        while( *buffer && *buffer != '<' ) buffer++;
        if ( *buffer == '<' ) buffer++;
        p=id;
        while( *buffer && *buffer != '/' ) *p++=*buffer++;
        if ( *buffer == '/' ) buffer++;
        p=suc;
        while( *buffer && *buffer != '>' ) *p++=*buffer++;
        if ( *buffer == '>' ) buffer++;
        // final de parsing, ¿que obtuve?
        printf("parse(): id proceso [%s] sucursal [%s]\n",id,suc);
        iid=atoi(id);
        isuc=atoi(suc);
        printf("parse(): id proceso [%d] sucursal [%d]\n",iid,isuc);
        proceso(iid,isuc);
    }
}

// luego de parsing, proceso id de proceso, id de sucursal
void proceso(int id,int suc) {
    printf("proceso(): id proceso [%d] sucursal [%d] total $ %lf\n",id,suc,ventas[suc]);
}

```

Copie tp64.c como tp65.c, implemente las funciones propuestas, llame a estas funciones en el lugar indicado. Compile como tp65, pruebe el sistema, podria obtener una salida como esta:

```

consola servidor:
grchere@debian:~/gccwork/src/fifo/v1$ ./tp65 &
main(): inicio servidor FIFO
main(): para salir envie señal SIGUSR2 a proceso 2858
main(): inicializo total de ventas de sucursales
main(): Tamaño maximo de operaciones atomicas sobre PIPES: 4096
main(): Tamaño maximo de buffer de servidor: 255
[1] 2858
grchere@debian:~/gccwork/src/fifo/v1$ main():lei: [<2860/0>] de FIFO
[/tmp/myfifo]
parse(): id proceso [2860] sucursal [0]
parse(): id proceso [2860] sucursal [0]
proceso(): id proceso [2860] sucursal [0] total $ 138.479047
main():lei: [<2860/1>] de FIFO [/tmp/myfifo]
parse(): id proceso [2860] sucursal [1]
parse(): id proceso [2860] sucursal [1]
proceso(): id proceso [2860] sucursal [1] total $ 55.640795
main():lei: [<2860/2>] de FIFO [/tmp/myfifo]
parse(): id proceso [2860] sucursal [2]
parse(): id proceso [2860] sucursal [2]

```

```

proceso(): id proceso [2860] sucursal [2] total $ 99.352217
main():lei: [<2860/3>] de FIFO [/tmp/myfifo]
parse(): id proceso [2860] sucursal [3]
parse(): id proceso [2860] sucursal [3]
proceso(): id proceso [2860] sucursal [3] total $ 138.450279
main():lei: [<2860/4>] de FIFO [/tmp/myfifo]
parse(): id proceso [2860] sucursal [4]
parse(): id proceso [2860] sucursal [4]
proceso(): id proceso [2860] sucursal [4] total $ 10.434312
main():lei: [<2860/5>] de FIFO [/tmp/myfifo]
parse(): id proceso [2860] sucursal [5]
parse(): id proceso [2860] sucursal [5]
proceso(): id proceso [2860] sucursal [5] total $ 84.119174
main():lei: [<2860/6>] de FIFO [/tmp/myfifo]
parse(): id proceso [2860] sucursal [6]
parse(): id proceso [2860] sucursal [6]
proceso(): id proceso [2860] sucursal [6] total $ 45.279634
main():lei: [<2860/7>] de FIFO [/tmp/myfifo]
parse(): id proceso [2860] sucursal [7]
parse(): id proceso [2860] sucursal [7]
proceso(): id proceso [2860] sucursal [7] total $ 212.639716
main():lei: [<2860/8>] de FIFO [/tmp/myfifo]
parse(): id proceso [2860] sucursal [8]
parse(): id proceso [2860] sucursal [8]
proceso(): id proceso [2860] sucursal [8] total $ 54.377959
main():lei: [<2860/9>] de FIFO [/tmp/myfifo]
parse(): id proceso [2860] sucursal [9]
parse(): id proceso [2860] sucursal [9]
proceso(): id proceso [2860] sucursal [9] total $ 82.784439
main():recibi señal de salida
main():fin servidor FIFO| retorno=0

[1]+ Done ./tp65
grchere@debian:~/gccwork/src/fifo/v1$
```

```

consola cliente:
grchere@debian:~/gccwork/src/fifo/v1$ ./tp62
main():inicio cliente FIFO|
main():envie 8 bytes a servidor [<2860/0>]
main():envie 8 bytes a servidor [<2860/1>]
main():envie 8 bytes a servidor [<2860/2>]
main():envie 8 bytes a servidor [<2860/3>]
main():envie 8 bytes a servidor [<2860/4>]
main():envie 8 bytes a servidor [<2860/5>]
main():envie 8 bytes a servidor [<2860/6>]
main():envie 8 bytes a servidor [<2860/7>]
main():envie 8 bytes a servidor [<2860/8>]
main():envie 8 bytes a servidor [<2860/9>]
main():fin cliente FIFO|
grchere@debian:~/gccwork/src/fifo/v1$ kill -SIGUSR2 2858
grchere@debian:~/gccwork/src/fifo/v1$
```

7. Modificar el servidor para grabar en un FIFO propio para cada cliente el total de ventas de la sucursal, con el siguiente formato: <sucursal/total ventas> . El FIFO en donde grabará la respuesta el servidor es un FIFO previamente creado por el cliente y su nombre tiene el formato: MY\_FIFO.<id proceso cliente>. Por lo tanto, cliente y

servidor, ambos, crean FIFO's; aquí tenemos código que puede compartirse y es repetitivo. Entonces vamos a crear una biblioteca de funciones compartidas myfifo.c allí implementaremos la función `crearfifo()`:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <string.h>
#include <limits.h>

#include "myfifo.h"

// crea pipe, devuelve true si pudo hacerlo, caso contrario devuelve false
int crearfifo(char *pipe) {
    int mf = mkfifo(pipe, 0_CREAT);
    if (mf == 0 || errno == EEXIST) {
        chmod(pipe, 0777);
        return 1;
    } else return 0;
}
```

Ahora la función `proceso()`, debería contestarle al cliente el total de ventas de la sucursal solicitada:

```
// luego de parsing, proceso id de proceso, id de sucursal
void proceso(int id, int suc) {
    char fifo[MAX_BUFFER], buffer[MAX_BUFFER];
    sprintf(fifo, MAX_BUFFER, "%s.%d", MY_FIFO, id);
    printf("proceso(): id proceso [%d] sucursal [%d] total $ %lf
fifo:[%s]\n", id, suc, ventas[suc], fifo);
    //crearfifo(fifo); //se supone previamente creado por el cliente
    int fdo = open(fifo, O_RDWR, 0777);
    sprintf(buffer, MAX_BUFFER, "<%d/%lf>", suc, ventas[suc]);
    write(fdo, buffer, strlen(buffer));
    close(fdo);
}
```

Copie `tp65.c` como `tp66.c`, implemente los cambios propuestos, compile como: `gcc -Wall -o tp66 tp66.c myfifo.c`

8. Modificar el cliente para crear FIFO cliente y recibir respuestas del servidor. Ahora el diseño del cliente sería:

```
main()
    obtengo id de proceso
    armo nombre de fifo cliente con MY_FIFO+id de proceso
    si pude crearfifo() entonces
        abrir FIFO cliente de RW
        abrir FIFO server de W
        crear buffer de MAX_BUFFER caracteres
```

```

    para <suc> 0 hasta 9 hacer
        armar en buffer mensaje a enviar al servidor para <suc>
        grabar buffer en FIFO server
        imprimir buffer indicado lo enviado al servidor
        leer buffer de FIFO cliente
        imprimir buffer indicado lo recibido del servidor
    fin para
    cerrar FIFO server
    cerrar FIFO cliente
    borrar FIFO cliente del sistema de archivos
sino
    error creando FIFO cliente!
fin si
fin main()

```

Copie tp62.c como tp67.c, implemente los cambios propuestos, compile como: `gcc -Wall -o tp67 tp67.c myfifo.c` . Ejecute server, cliente, pruebe el sistema, podrá obtener una salida como esta:

```

consola server:
grchere@debian:~/gccwork/src/fifo/v1$ ./tp66 &
main(): inicio servidor FIFO!
main(): para salir envie señal SIGUSR2 a proceso 2863
main(): inicializo total de ventas de sucursales
main(): Tamaño maximo de operaciones atomicas sobre PIPES: 4096
main(): Tamaño maximo de buffer de servidor: 255
[1] 2863
grchere@debian:~/gccwork/src/fifo/v1$ main():lei: [<2866/0>] de FIFO
[~/tmp/myfifo]
parse(): id proceso [2866] sucursal [0]
parse(): id proceso [2866] sucursal [0]
proceso(): id proceso [2866] sucursal [0] total $ 159.773684
fifo:[~/tmp/myfifo.2866]
main():lei: [<2866/1>] de FIFO [~/tmp/myfifo]
parse(): id proceso [2866] sucursal [1]
parse(): id proceso [2866] sucursal [1]
proceso(): id proceso [2866] sucursal [1] total $ 53.234007
fifo:[~/tmp/myfifo.2866]
main():lei: [<2866/2>] de FIFO [~/tmp/myfifo]
parse(): id proceso [2866] sucursal [2]
parse(): id proceso [2866] sucursal [2]
proceso(): id proceso [2866] sucursal [2] total $ 101.434253
fifo:[~/tmp/myfifo.2866]
main():lei: [<2866/3>] de FIFO [~/tmp/myfifo]
parse(): id proceso [2866] sucursal [3]
parse(): id proceso [2866] sucursal [3]
proceso(): id proceso [2866] sucursal [3] total $ 209.114185
fifo:[~/tmp/myfifo.2866]
main():lei: [<2866/4>] de FIFO [~/tmp/myfifo]
parse(): id proceso [2866] sucursal [4]
parse(): id proceso [2866] sucursal [4]
proceso(): id proceso [2866] sucursal [4] total $ 4.183333
fifo:[~/tmp/myfifo.2866]
main():lei: [<2866/5>] de FIFO [~/tmp/myfifo]
parse(): id proceso [2866] sucursal [5]
parse(): id proceso [2866] sucursal [5]
proceso(): id proceso [2866] sucursal [5] total $ 183.813549
fifo:[~/tmp/myfifo.2866]

```

```

main():lei: [<2866/6>] de FIFO [/tmp/myfifo]
parse(): id proceso [2866] sucursal [6]
parse(): id proceso [2866] sucursal [6]
proceso(): id proceso [2866] sucursal [6] total $ 67.664777
fifo:[/tmp/myfifo.2866]
main():lei: [<2866/7>] de FIFO [/tmp/myfifo]
parse(): id proceso [2866] sucursal [7]
parse(): id proceso [2866] sucursal [7]
proceso(): id proceso [2866] sucursal [7] total $ 118.434233
fifo:[/tmp/myfifo.2866]
main():lei: [<2866/8>] de FIFO [/tmp/myfifo]
parse(): id proceso [2866] sucursal [8]
parse(): id proceso [2866] sucursal [8]
proceso(): id proceso [2866] sucursal [8] total $ 173.126028
fifo:[/tmp/myfifo.2866]
main():lei: [<2866/9>] de FIFO [/tmp/myfifo]
parse(): id proceso [2866] sucursal [9]
parse(): id proceso [2866] sucursal [9]
proceso(): id proceso [2866] sucursal [9] total $ 104.360250
fifo:[/tmp/myfifo.2866]

grchere@debian:~/gccwork/src/fifo/v1$ 
grchere@debian:~/gccwork/src/fifo/v1$ main():recibi señal de salida
main():cierra FIFO server [/tmp/myfifo]
main():elimino FIFO server [/tmp/myfifo]
main():fin servidor FIFO| retorno=0

[1]+  Done                  ./tp66
grchere@debian:~/gccwork/src/fifo/v1$ 

consola cliente:
grchere@debian:~/gccwork/src/fifo/v1$ ./tp67
main():inicio cliente FIFO|
main():creando FIFO [/tmp/myfifo.2866]
main():abro FIFO [/tmp/myfifo] para grabar
main():abro FIFO [/tmp/myfifo.2866] para leer y grabar
main():envie 8 bytes a servidor [<2866/0>]
main():recibi 14 bytes de servidor [<0/159.773684>]
main():envie 8 bytes a servidor [<2866/1>]
main():recibi 13 bytes de servidor [<1/53.234007>]
main():envie 8 bytes a servidor [<2866/2>]
main():recibi 14 bytes de servidor [<2/101.434253>]
main():envie 8 bytes a servidor [<2866/3>]
main():recibi 14 bytes de servidor [<3/209.114185>]
main():envie 8 bytes a servidor [<2866/4>]
main():recibi 12 bytes de servidor [<4/4.183333>]
main():envie 8 bytes a servidor [<2866/5>]
main():recibi 14 bytes de servidor [<5/183.813549>]
main():envie 8 bytes a servidor [<2866/6>]
main():recibi 13 bytes de servidor [<6/67.664777>]
main():envie 8 bytes a servidor [<2866/7>]
main():recibi 14 bytes de servidor [<7/118.434233>]
main():envie 8 bytes a servidor [<2866/8>]
main():recibi 14 bytes de servidor [<8/173.126028>]
main():envie 8 bytes a servidor [<2866/9>]
main():recibi 14 bytes de servidor [<9/104.360250>]
main():cierra FIFO's
main():elimino FIFO cliente [/tmp/myfifo.2866]

```

```

main(): fin cliente FIFO!
grchere@debian:~/gccwork/src/fifo/v1$ 
grchere@debian:~/gccwork/src/fifo/v1$ ls -l /tmp
total 12
drwx----- 2 grchere grchere 4096 2009-06-25 14:17 gconfd-grchere
drwx----- 2 grchere grchere 4096 2009-06-25 14:16 keyring-SehzsZ
prwxrwxrwx 1 grchere grchere 0 2009-06-25 14:48 myfifo
drwx----- 2 grchere grchere 4096 2009-06-25 14:17 orbit-grchere
prwxr-xr-x 1 grchere grchere 0 2009-06-25 14:33 SciTE.2822.in
grchere@debian:~/gccwork/src/fifo/v1$ 
grchere@debian:~/gccwork/src/fifo/v1$ kill -SIGUSR2 2863
grchere@debian:~/gccwork/src/fifo/v1$ 

```

9. Analicemos el código y la funcionalidad de ambos programas (servidor y cliente), seguramente el cliente también necesitará hacer un *parsing* de lo recibido por parte del servidor, cuyo formato es idéntico al mensaje enviado por cliente, sólo cambia su contenido. Aquí tenemos otra oportunidad de generalizar código, la función *parse()*. El problema es que *parse()*, en su interior, llama a la función *proceso()* y ésta es propia del servidor (en el caso del cliente, se necesita "otra versión de la función *proceso()*"). Necesitamos hacer que esta función pueda ser *"intercambiable"* desde el exterior de *parse()*. La idea sería que el cliente use su propia función *proceso()*, distinta a la del servidor. ¿Cómo puede lograrse esto en *parse()*? Pasándole a *parse()* un puntero a la función *proceso()* a utilizar. En C/C++, de la misma forma que existen punteros a variables, también existen punteros a funciones, muy útiles en casos como este. El prototipo de la función *parse()* sería:

```
void parse(char *buffer, void (*proceso)(char *campo1, char *campo2));
```

El primer argumento, es el buffer a "parsear", el segundo es un puntero a la función *proceso()* a usar, a su vez, la función *proceso()* tiene el prototipo:

```
void proceso(char *campo1, char *campo2);
```

En donde *campo1* será id de proceso o sucursal, según sea el caso.

En donde *campo2* será sucursal o total de ventas, según sea el caso.

La función *parse()* debería ser más genérica, simplemente obtener los dos campos del mensaje y llamar (para cada mensaje "parseado") a *proceso()* pasando los campos como argumentos. Los campos son de tipos de datos distintos, puesto que el id de proceso es un entero, mientras que el total de ventas es un double. Para compatibilizar esto, podríamos pensar en una función *proceso()* genérica que reciba dos cadenas de caracteres y que cada "proceso()" haga las conversiones pertinentes.

Esta nueva versión de *parse()* deberá ser puesta dentro de *myfifo.c*, puesto que pasaría a ser código genérico, reutilizable por cliente y servidor:

```
// analiza buffer recibido de los clientes para determinar
// id de proceso cliente y nro de sucursal solicitada
// formato de buffer: "<campo1/campo2><campo1/campo2>....<campo1/campo2>"
```

```

void parse(char *buffer, void (*proceso)(char *campo1, char *campo2)) {
    char id[MAX_BUFFER];
    char suc[MAX_BUFFER];
    //int iid, isuc;
    char *p;
    while(*buffer) {
        memset(id, 0, MAX_BUFFER);
        memset(suc, 0, MAX_BUFFER);
        while( *buffer && *buffer != '<' ) buffer++;
        if ( *buffer == '<' ) buffer++;
        p=id;
        while( *buffer && *buffer != '/') *p++=*buffer++;
        if ( *buffer == '/' ) buffer++;
        p=suc;
        while( *buffer && *buffer != '>' ) *p++=*buffer++;
        if ( *buffer == '>' ) buffer++;
        // final de parsing, ¿que obtuve?
        printf("parse(): campo1 [%s] campo2 [%s]\n", id, suc);
        //iid=atoi(id);
        //isuc=atoi(suc);
        //printf("parse(): id proceso [%d] sucursal [%d]\n", iid, isuc);
        (*proceso)(id, suc); // hago uso del puntero, ejecuto función!
    }
}

```

¿Cómo sería la invocación de parse() desde el cliente o desde el servidor? simplemente:

```
parse(buffer, proceso);
```

En donde proceso, es el nombre de la función a ser invocada por parse() cada vez que se desea obtener el par: campo1 y campo2 !!.

- 9.1. Dentro de la carpeta fifo, crear la carpeta v2 para almacenar la versión 2 de este proyecto.
- 9.2. Dentro de v2 copie todo el contenido actual de v1.
- 9.3. Modifique myfifo.c, mueva la función parse() de tp66.c a myfifo.c
- 9.4. Agregue dentro de myfifo.h el prototipo de la función parse().
- 9.5. Cambie los prototipos de las funciones proceso() tanto en tp66 (servidor) como en tp67 (cliente).
- 9.6. Implemente la función proceso() en tp67 como:

```

// luego de parsing, proceso sucursal y total de venta recibido
void proceso(char *s, char *v) {
    int suc=atoi(s);
    double vta=atof(v);
    printf("proceso(): sucursal [%d] total $ %lf\n", suc, vta);
}

```

- 9.7. Implemente la función proceso() en tp66 como:

```

// luego de parsing, proceso id de proceso, id de sucursal
void proceso(char *sid, char *ssuc) {

```

```

int id=atoi(sid),suc=atoi(ssuc);
char fifo[MAX_BUFFER],buffer[MAX_BUFFER];
snprintf(fifo,MAX_BUFFER,"%s.%d",MY_FIFO,id);
printf("proceso(): id proceso [%d] sucursal [%d] total $ %lf
fifo:[%s]\n",id,suc,ventas[suc],fifo);
snprintf(buffer,MAX_BUFFER,<%d/%lf>,suc,ventas[suc]);
int fdo = open(fifo,O_RDWR,0777);
write(fdo,buffer,strlen(buffer));
close(fdo);
}

```

Entonces, `parse()` es compartida, y por lo tanto, está en `myfifo.c`, su prototipo está en `myfifo.h`, es distribuible, reusable, independiente de la implementación particular de server o cliente que se realice.

Entonces, `proceso()` es local, propia de cada programa server o cliente, no reusable, una versión exclusiva para cada caso, por lo tanto, está implementada en `tp66.c` y en `tp67.c`. No hace falta que se llame `proceso()` en ambos programas, puede tener cualquier nombre, sólo debe tener el prototipo (formato genérico de la función) que requiere `parse()`.

**9.8.** Modifique `tp66.c` y `tp67.c` para que hagan la llamada a la función `parse()` de la forma indicada previamente.

**9.9.** Compile cliente y servidor como se indicó en el punto anterior. Ejecute server, cliente, pruebe el sistema, podrá obtener una salida como esta:

```

consola server:
grchere@debian:~/gccwork/src/fifo/v2$ ./tp66 &
[1] 2877
grchere@debian:~/gccwork/src/fifo/v2$ main():inicio servidor FIFO
main():para salir envie señal SIGUSR2 a proceso 2877
main():inicializo total de ventas de sucursales
main():Tamaño maximo de operaciones atomicas sobre PIPES: 4096
main():Tamaño maximo de buffer de servidor: 255

grchere@debian:~/gccwork/src/fifo/v2$ 
grchere@debian:~/gccwork/src/fifo/v2$ main():lei: [<2883/0>] de FIFO
[/tmp/myfifo]
parse(): campo1 [2883] campo2 [0]
proceso(): id proceso [2883] sucursal [0] total $ 127.815824
fifo:[/tmp/myfifo,2883]
main():lei: [<2883/1>] de FIFO [/tmp/myfifo]
parse(): campo1 [2883] campo2 [1]
proceso(): id proceso [2883] sucursal [1] total $ 124.333209
fifo:[/tmp/myfifo,2883]
main():lei: [<2883/2>] de FIFO [/tmp/myfifo]
parse(): campo1 [2883] campo2 [2]
proceso(): id proceso [2883] sucursal [2] total $ 104.190440
fifo:[/tmp/myfifo,2883]
main():lei: [<2883/3>] de FIFO [/tmp/myfifo]
parse(): campo1 [2883] campo2 [3]
proceso(): id proceso [2883] sucursal [3] total $ 9.342347
fifo:[/tmp/myfifo,2883]
main():lei: [<2883/4>] de FIFO [/tmp/myfifo]

```

```

parse(): campo1 [2883] campo2 [4]
proceso(): id proceso [2883] sucursal [4] total $ 25.764340
fifo:[/tmp/myfifo,2883]
main():lei: [<2883/5>] de FIFO [/tmp/myfifo]
parse(): campo1 [2883] campo2 [5]
proceso(): id proceso [2883] sucursal [5] total $ 102.006486
fifo:[/tmp/myfifo,2883]
main():lei: [<2883/6>] de FIFO [/tmp/myfifo]
parse(): campo1 [2883] campo2 [6]
proceso(): id proceso [2883] sucursal [6] total $ 124.583868
fifo:[/tmp/myfifo,2883]
main():lei: [<2883/7>] de FIFO [/tmp/myfifo]
parse(): campo1 [2883] campo2 [7]
proceso(): id proceso [2883] sucursal [7] total $ 89.308833
fifo:[/tmp/myfifo,2883]
main():lei: [<2883/8>] de FIFO [/tmp/myfifo]
parse(): campo1 [2883] campo2 [8]
proceso(): id proceso [2883] sucursal [8] total $ 172.868113
fifo:[/tmp/myfifo,2883]
main():lei: [<2883/9>] de FIFO [/tmp/myfifo]
parse(): campo1 [2883] campo2 [9]
proceso(): id proceso [2883] sucursal [9] total $ 76.690872
fifo:[/tmp/myfifo,2883]
main():recibi señal de salida
main():cierra FIFO server [/tmp/myfifo]
main():elimina FIFO server [/tmp/myfifo]
main():fin servidor FIFO! retorno=0

[1]+ Done                  ./tp66
grchere@debian:~/gccwork/src/fifo/v2$
```

#### consola cliente:

```

grchere@debian:~/gccwork/src/fifo/v2$ ./tp67
main():inicio cliente FIFO!
main():creando FIFO [/tmp/myfifo,2883]
main():abro FIFO [/tmp/myfifo] para grabar
main():abro FIFO [/tmp/myfifo,2883] para leer y grabar
main():envie 8 bytes a servidor [<2883/0>]
parse(): campo1 [0] campo2 [127.815824]
proceso(): sucursal [0] total $ 127.815824
main():envie 8 bytes a servidor [<2883/1>]
parse(): campo1 [1] campo2 [124.333209]
proceso(): sucursal [1] total $ 124.333209
main():envie 8 bytes a servidor [<2883/2>]
parse(): campo1 [2] campo2 [104.190440]
proceso(): sucursal [2] total $ 104.190440
main():envie 8 bytes a servidor [<2883/3>]
parse(): campo1 [3] campo2 [9.342347]
proceso(): sucursal [3] total $ 9.342347
main():envie 8 bytes a servidor [<2883/4>]
parse(): campo1 [4] campo2 [25.764340]
proceso(): sucursal [4] total $ 25.764340
main():envie 8 bytes a servidor [<2883/5>]
parse(): campo1 [5] campo2 [102.006486]
proceso(): sucursal [5] total $ 102.006486
main():envie 8 bytes a servidor [<2883/6>]
parse(): campo1 [6] campo2 [124.583868]
proceso(): sucursal [6] total $ 124.583868
```

```

main(): envie 8 bytes a servidor [<2003/7>]
parse(): campo1 [7] campo2 [89.308833]
proceso(): sucursal [7] total $ 89.308833
main(): envie 8 bytes a servidor [<2003/8>]
parse(): campo1 [8] campo2 [172.868113]
proceso(): sucursal [8] total $ 172.868113
main(): envie 8 bytes a servidor [<2003/9>]
parse(): campo1 [9] campo2 [76.690872]
proceso(): sucursal [9] total $ 76.690872
main(): cierro FIFO's
main(): elimino FIFO cliente [/tmp/myfifo.2883]
main(): fin cliente FIFO
grchere@debian:~/gccwork/src/fifo/v2$ kill -SIGUSR2 2877
grchere@debian:~/gccwork/src/fifo/v2$
```

**10.** Analizando la biblioteca `myfifo.c` podemos observar que hay código repetitivo y poco reutilizable dentro de la función `parse()`, básicamente, ésta necesita el auxilio de otra función que pueda copiar un string que se encuentra dentro de otro string y delimitado por dos caracteres, se podría reutilizar una función tal como:

```
char *strcpyentre(char *to, char *from, char desde, char hasta);
```

Esta función devuelve un puntero al desplazamiento ocurrido dentro de la cadena `from` a medida que se fue haciendo el *parsing*.

**10.1.** Dentro de la carpeta `fifo`, crear la carpeta `v3` para almacenar la versión 3 de este proyecto.

**10.2.** Dentro de `v3` copie todo el contenido actual de `v2`.

**10.3.** Agregar el prototipo de `strcpyentre()` en `myfifo.h`, implementar la función `strcpyentre()` dentro de `myfifo.c`, para luego reutilizarla desde `parse()`, el código de la función `strcpyentre()` sería:

```

// to es un cadena que se encuentra dentro de from y esta delimitada
// por dos caracteres: desde y hasta
// si no encuentra la subcadena, devuelve una cadena vacia (inicializada a
// NULL)
// to es una cadena que tiene previamente asignado -al menos-
// strlen(from)+1 caracteres.
// devuelve el desplazamiento realizado en from durante el parsing
char *strcpyentre(char *to, char *from, char desde, char hasta) {
    memset(to, 0, strlen(from)+1); // to se rellena con nulos
    while( *from && *from != desde) from++;
    if ( *from == desde ) from++;
    while( *from && *from != hasta) *to++=*from++;
    return from;
}
```

Dentro de `myfifo.c`, el código de la función `parse()` ahora pasaría a ser:

```

// analiza buffer recibido de los clientes para determinar id de proceso
// cliente y nro de sucursal solicitada
// formato de buffer: "<campo1/campo2><campo1/campo2>....<campo1/campo2>"
```

```

void parse(char *buffer, void (*proceso)(char *campo1, char *campo2)) {
    char id[MAX_BUFFER];
    char suc[MAX_BUFFER];
    while(*buffer) {
        buffer=strcpyentre(id,buffer,'<','/');
        buffer=strcpyentre(suc,buffer,'/','>');
        if (*id && *suc) { // algo obtuve
            // final de parsing, ¿que obtuve?
            //printf("parse(): campo1 [%s] campo2 [%s]\n",id,suc); // trace
            (*proceso)(id,suc);
        }
    }
}

```

**10.4.** El manejo de pipes está vinculado con la señal **SIGPIPE** (también lo están los sockets tcp). Si un proceso pretende grabar sobre un pipe en donde el lector de dicho pipe ha terminado, se generará esta señal. Implementar una función de manipulación de esta señal para que emita el mensaje de error correspondiente en caso de recibir esta señal. Implemente esta señal tanto en cliente como en servidor.

**10.5.** Compile cliente y servidor como se indicó en el punto anterior. Ejecute server, cliente, pruebe el sistema, la salida a obtener sería la misma, pero con una librería más reutilizable y mejor estructurada.

**10.6.** Elimine todos los mensajes innecesarios enviados a la consola, recompile, una salida posible sería:

```

consola server:
grchere@debian:~/gccwork/src/fifo/v3$ ./tp66 &
[1] 2890
grchere@debian:~/gccwork/src/fifo/v3$ main():inicio servidor FIFO
main():para salir envie señal SIGUSR2 a proceso 2890
main():inicializo total de ventas de sucursales
main():Tamaño maximo de operaciones atomicas sobre PIPES: 4096
main():Tamaño maximo de buffer de servidor: 255
proceso(): proceso [2891] suc [0] total $ 97.467573 fifo [/tmp/myfifo.2891]
proceso(): proceso [2891] suc [1] total $ 29.156740 fifo [/tmp/myfifo.2891]
proceso(): proceso [2891] suc [2] total $ 182.218157 fifo [/tmp/myfifo.2891]
proceso(): proceso [2891] suc [3] total $ 197.326834 fifo [/tmp/myfifo.2891]
proceso(): proceso [2891] suc [4] total $ 41.050789 fifo [/tmp/myfifo.2891]
proceso(): proceso [2891] suc [5] total $ 76.399383 fifo [/tmp/myfifo.2891]
proceso(): proceso [2891] suc [6] total $ 149.939370 fifo [/tmp/myfifo.2891]
proceso(): proceso [2891] suc [7] total $ 127.515767 fifo [/tmp/myfifo.2891]
proceso(): proceso [2891] suc [8] total $ 104.658367 fifo [/tmp/myfifo.2891]
proceso(): proceso [2891] suc [9] total $ 142.723383 fifo [/tmp/myfifo.2891]
main():recibi señal de salida
main():cierra FIFO server [/tmp/myfifo]
main():elimina FIFO server [/tmp/myfifo]
main():fin servidor FIFO| retorno=0

[1]+  Done                  ./tp66
grchere@debian:~/gccwork/src/fifo/v3$ 

consola cliente:
grchere@debian:~/gccwork/src/fifo/v3$ ./tp67
main():inicio cliente FIFO
main():creando FIFO [/tmp/myfifo.2891]

```

```
main(): abro FIFO [/tmp/myfifo] para grabar
main(): abro FIFO [/tmp/myfifo.2891] para leer y grabar
proceso(): suc [0] total $ 97.467573
proceso(): suc [1] total $ 29.156740
proceso(): suc [2] total $ 182.218157
proceso(): suc [3] total $ 197.326034
proceso(): suc [4] total $ 41.050789
proceso(): suc [5] total $ 76.399383
proceso(): suc [6] total $ 149.939370
proceso(): suc [7] total $ 127.515767
proceso(): suc [8] total $ 104.656367
proceso(): suc [9] total $ 142.723383
main(): cierro FIFO's
main(): elimino FIFO cliente [/tmp/myfifo.2891]
main(): fin cliente FIFO
grchere@debian:~/gccwork/src/fifo/v3$ kill -SIGUSR2 2890
grchere@debian:~/gccwork/src/fifo/v3$
```

### En cuanto a los FIFO's

Habrá notado que los FIFO's que pudieron haber sido abiertos de solo lectura se abrieron como R/W (read/write, lectura-escritura). Esto está en relación con el hecho de no obtener un EOF cuando el servidor ya no tenga ningún cliente que atender o bien cuando el cliente ya no tenga más mensajes que recuperar del servidor.

## Capítulo VII. IPC, Cola de Mensajes (Queues).

**Objetivo:** Desarrollar una serie de programas que permitan comprobar distintas funcionalidades vinculadas con la gestión de cola de mensajes como una forma de comunicación entre procesos (IPC: Inter Process Communication) provistas por el kernel del SO. Introducir al desarrollo de programas servidores y clientes. Estimular una programación modular, con código reusable.

### Introducción

Existen 3 formas de IPC's:

1. Cola de Mensajes (Message Queues)
2. Semáforos (Semaphores)
3. Memoria Compartida (Shared Memory)

Dentro del kernel se identifican a través de un número entero positivo, este número es de tipo `key_t` y está definido dentro del archivo de cabecera `sys/types.h`. De alguna forma, si dos o más programas pretenden compartir cualquiera de estos recursos de IPC's deberán conocer este número para poder accederlo de forma unívoca. Para compartir este número se podría hacer lo siguiente:

1. Un programa servidor podría crear un IPC de tipo `IPC_PRIVATE`, obtener su identificador y luego guardarlo en forma persistente en el sistema de archivos para que luego el o los programas cliente puedan leerlo.
2. El programa servidor y cliente acuerdan un número fijo de clave, indicado por ejemplo en un archivo de cabecera común a ambos programas e incorporado en la compilación de ambos.
3. El programa servidor y cliente acuerdan un path (una ruta de acceso, la ubicación de una carpeta dentro del sistema de archivos) y un id de proyecto (un simple carácter). Este path y id de proyecto pueden estar en un archivo de cabecera común a ambos. Con estos dos datos se puede invocar a la función `ftok()`, la cual, a partir de estos dos datos genera una clave única de IPC para que ambos programas puedan acceder al mismo recurso.

Cada una de estas alternativas tiene sus ventajas y desventajas, en nuestro caso, se optó por la opción 3.

¿Qué es una cola de mensajes? Es una lista enlazada de mensajes almacenada dentro del kernel del SO e identificada a través de un id (de tipo `key_t`).

¿Qué estructura tiene cada mensaje? Todos los mensajes tienen idéntica estructura y la misma está compuesta por un número de tipo `long`, que representa el tipo de mensaje y luego continúa con una serie de caracteres acorde con la necesidad de cada aplicación. Esto permite gran flexibilidad de implementación.

Esta simple estructura de cola de mensajes nos permite pensar en una implementación client-server. Se podría implementar un servidor que espera recibir mensajes de tipo 1 de la cola de mensajes, estos mensajes son peticiones de los clientes, una vez recibido el mensaje, lo procesa y ... ¿cómo devolver una respuesta a los clientes? Se nos ocurrió la siguiente alternativa: dentro de la petición del cliente, incluir su id de proceso, entonces, el servidor, como respuesta a la petición cliente puede introducir un mensaje en la cola de mensajes de tipo X en donde X es el id de proceso cliente a quien el servidor le está respondiendo.

El Analista Y pretende diseñar un servidor que informe a los clientes acerca de los totales de ventas de las sucursales de la Empresa X bajo este enfoque, los mensajes que enviarán los clientes tendrán el formato: <id de proceso cliente/número de sucursal> y serán de tipo 1, mientras que el formato del mensaje a devolver por parte del servidor será: <número de sucursal/total de ventas> y será un mensaje de tipo <id de proceso cliente>.

El formato propuesto en los mensajes a intercambiar entre cliente y servidor, permiten los siguientes diseños:

#### Programa servidor:

```
main()
    inicializar totales de ventas de sucursales
    crear clave IPC (según path y id proyecto)
    crear cola de mensajes usando clave IPC
    si pude crear Cola Ok entonces
        mientras no salir
            recibir mensaje de tipo 1
            parsear mensaje recibido ("<id proceso cliente/sucursal>")
            si número de sucursal es correcto entonces
                obtener total de ventas de sucursal
                enviar mensaje a cola de tipo <id proceso cliente>
            fin si
        fin mientras
        borrar cola
    sino
        error creando cola
    fin si
fin main()
```

#### Programa cliente:

```
main()
    crear clave IPC (según path y id proyecto)
    obtener cola de mensajes usando clave IPC
    si pude obtener Cola Ok entonces
        obtener id de proceso
        para cada sucursal hacer
            armar mensaje "<id proceso/sucursal>"
            enviar mensaje a cola de tipo 1
            si pude enviar mensaje Ok entonces
                recibir mensaje de tipo <id proceso> de cola
                si pude recibir mensaje Ok entonces
                    parsear mensaje recibido("<id sucursal/total
ventas>")
```

```

        mostrar mensaje recibido
    fin si
    fin si
    fin para
sino
    error obteniendo cola
fin si
fin main()

```

1. El Analista de Sistemas Y está muy entusiasmado con IPC y está convencido de que este no será el último desarrollo en esta línea, por lo tanto, se propone implementar una pequeña librería para el manejo de cola de mensajes, semáforos y memoria compartida: myipc.c cuyo archivo de cabecera será myipc.h. Este archivo de cabecera y librería será compartido tanto por cliente como por servidor, por lo tanto, aquí podemos acordar el path y id de proyecto según lo dicho en la introducción, así que dentro de myipc.h debemos incluir:

```

#define IPC_PATH "."
#define IPC_KEY 'm'

```

También se deberá incluir dentro de myipc.h una serie de archivos de cabecera a ser utilizados por los programas servidor y cliente, para que ambos accedan a las funciones de ipc y otras librerías necesarias por esta implementación:

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <ctype.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <limits.h>
#include <time.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <signal.h>

```

La API (Application Program Interface<sup>8</sup>) de myipc.c definida dentro de myipc.h y compartida por clientes y servidores será:

```

int borrar_cola(int qid);
int crear_cola(key_t key);
int enviar_msg(int qid, void *qbuf, int msglen, long type);
void init(double *ventas);
key_t obtener_clave(char *path, char car);
int obtener_cola(key_t key);

```

<sup>8</sup> Define el protocolo de comunicación (interfaz) con una capa (layer) de software en particular, en este caso, con el modulo myipc.c. Esto permite que toda otra capa de software que interactúe con ésta lo haga en los términos de esta interfase previamente establecida. Es muy importante definirla de manera clara, simple y reusable; de manera tal, que no deba tener que realizarse cambios sobre la misma a futuro (a lo sumo, ampliaciones) aunque si existan cambios en su implementación (cambios al interior de esta capa de software), de forma tal, que permita independencia y encapsulación del código con respecto a las otras capas de software que la utilizan.

```
int recibir_msg(int qid, void *qbuf, int msglen, long type);
```

Tanto cliente como servidor ejecutarán:

```
key_t clave = obtener_clave(IPC_PATH, IPC_KEY);  
para generar una clave IPC con la cual se identifique a la cola de mensajes.  
El servidor creará la cola de mensajes:  
int idcola = crear_cola(clave);  
mientras que el cliente se conectará a una cola previamente creada:  
int idcola = obtener_cola(clave);  
Estas funciones devolverán -1 cuando exista alguna condición de error.  
El servidor inicializará el total de ventas de las sucursales (generará totales de ventas):
```

```
double ventas[10];  
...  
init(&ventas[0]);
```

Tanto para el envío como para la recepción de mensajes: `qid` se refiere al id de la cola de mensajes (`idcola`), `qbuf` se refiere al mensaje a enviar o recibir, `msglen` se refiere al largo del mensaje<sup>9</sup> y `type` al tipo de mensaje a enviar (1 o bien <id de proceso cliente>).

`Myipc.c` permitirá un uso más abstracto de IPC y permitirá aislar el código del Analista Y de los detalles de implementación. ¿Para qué podría servirle esto? Hay varias razones para hacerlo, pero mencionemos al menos una: portabilidad. El Analista Y desarrollará sus servidores y clientes en términos de la API definida por `myipc.h`, no es el caso de la cola de mensajes, pero nosotros utilizaremos lo que se denomina una implementación System V R4 (SVR4) de semáforos, pues resulta que también existen los semáforos POSIX, para ambos tipos de semáforos se utilizan API's muy distintas para comunicarse con el kernel. El Analista Y podría tener distintas versiones de `myipc.c` (la implementación de la API `myipc.h`), pero todas con la misma API (`myipc.h`), según la plataforma donde corra su código o acorde con las características de la aplicación a desarrollar podrá usar una u otra, sin que ni siquiera lo noten los programadores que trabajarán con El, puesto que ellos hacen todas las operaciones en términos de `myipc.h`.

Comencemos con la implementación de esta librería (`myipc.c`), ¿Qué implica obtener una clave? simplemente llamar a la función `ftok()` para que ésta haga su trabajo:

```
key_t obtener_clave(char *path, char car) {  
    return ftok(path, car);  
}
```

Para crear u obtener acceso a una cola de mensajes podemos usar la función `msgget()` indicándole la clave y el modo (formado por el tipo de acción a realizar

<sup>9</sup> Cabe aclarar que este número no incluye lo que ocupa el tipo de mensaje. Únicamente se contabiliza el largo de los caracteres del mensaje.

(IPC\_CREAT para crear una cola, sino se trata solo de obtener el id de una cola previamente creada) y el permiso que tendrá este proceso sobre dicho recurso). Aquí queda de manifiesto que hay una cuestión de permisos vinculada con IPC, no cualquier proceso puede crear recursos dentro del kernel<sup>10</sup>:

```
/* Crea cola de mensajes */
int crear_cola(key_t key) {
    return msgget(key, IPC_CREAT|0660);
}

/* Obtiene cola de mensajes previamente creada */
int obtener_cola(key_t key) {
    return msgget(key, 0660);
}
```

Una vez que hemos creado una cola u obtenido su id, podemos cambiar algunas características de la misma utilizando la función msgctl(), ésta también nos servirá para destruir la cola:

```
/* Destruye cola de mensajes */
int borrar_cola(int qid) {
    return msgctl(qid, IPC_RMID, 0);
}
```

Sabemos que todo mensaje comienza con un long y continúa con una serie de caracteres (totalmente variable, según nuestra necesidad), para las funciones de envío y recepción de mensajes (msgsnd() y msgrcv() respectivamente) obviamente<sup>11</sup>, el mensaje deberá simbolizarse como algo de tipo void \* (un puntero o referencia a un tipo de dato void, un tipo genérico, al cual pueda adaptarse cualquier mensaje). Nuestra API continuará con esta idea, pero obviamente habrá que indicar el largo del mensaje (que no incluye al long inicial). También se agregó que se pueda indicar el tipo, de esta forma, el programador no tiene necesidad de cargar el tipo de mensaje en este long inicial, ya que lo haremos nosotros haciendo uso de la función memcpy() y del hecho que sabemos que todo mensaje comienza con este long:

```
/* Envía un mensaje de determinado tipo a la cola */
int enviar_msg(int qid, void *qbuf, int msgsize, long type) {
    memcpy(qbuf, &type, sizeof(long)); //el efecto genérico de: qbuf->mtype = type;
    return msgsnd(qid, qbuf, msgsize, 0);
}

/* Obtiene un mensaje de la cola de un determinado tipo */
int recibir_msg(int qid, void *qbuf, int msgsize, long type) {
```

<sup>10</sup> En este sentido, tenga presente con qué usuario está ejecutando su aplicación, podría tener problemas en la ejecución de programas que utilizan IPC si el mismo no posee los privilegios suficientes. En tal caso, pruebe de ejecutar el código utilizando el usuario root (administrador).

<sup>11</sup> ¿Qué otra alternativa tenía el diseñador de las funciones msgsnd() y msgrcv()? No tenía otra posibilidad más que definirlas en forma genérica, independiente del tipo de dato del mensaje a enviar y ello en ANSI C sólo puede hacerse a través de un puntero de tipo void, lo cual es un puntero a una dirección de memoria, queda por resolver el problema de ¿Cuántos bytes hay en dicha dirección de memoria? ¿Dónde termina lo apuntado? Para ello quedan siempre 2 alternativas: indicar el tamaño de lo apuntado usando otro argumento o bien una convención: lo apuntado termina -por ejemplo- con un carácter nulo (0).

```
    memcpy(qbuf, &type, sizeof(long));
    return msgrecv(qid, qbuf, msgsize, type, 0);
}
```

Esto otorga gran flexibilidad, por ejemplo, nuestro servidor podría declarar el formato de los mensajes de la cola como:

```
#define SUC_MSG_LEN 20
...
struct sucmsg {
    long mtype;
    char mtext[SUC_MSG_LEN];
};
...
```

y luego hacer uso de la librería para recibir mensajes de tipo 1 (1L) con largo SUC\_MSG\_LEN:

```
...
struct sucmsg qbuf;
...
if ( recibir_msg(idcola, &qbuf, SUC_MSG_LEN, 1L) != -1) {
    printf("main():recibi [%s] como peticion\n", qbuf.mtext);
}
...
...
```

Puede completar el resto de la API, hacer un pequeño programa de prueba de la librería y compilar el código<sup>12</sup>.

2. Utilice el programa de prueba de la librería del punto anterior y cópíelo como tp71.c. Implemente en este programa el servidor de la cola de mensajes, tal como se indicó en la introducción. Esta primera versión simplemente creará la cola de mensajes, esperará por mensajes, mostrará el contenido de los mensajes recibidos.

Compile y ejecute tp71.c. El programa quedará bloqueado esperando mensajes de una cola a quien nadie envía nada, efectivamente, la función msgrecv() es bloqueante<sup>13</sup>, deje por ahora el programa bloqueado y abra otra consola<sup>14</sup>, ejecute el comando: ipcs (consulte su sintaxis: man ipcs<sup>15</sup>), este comando Unix es muy importante porque permite la administración (por fuera de las aplicaciones) de los recursos IPC actualmente existentes en el kernel del SO, de este comando podrá obtener una salida como:

<sup>12</sup> Para simplificar la compilación, evitaremos crear una librería estática, simplemente podríamos hacer lo siguiente:  
\$ gcc -Wall -o prueba prueba.c myipc.c  
en donde prueba es el ejecutable, prueba.c es el programa de prueba y myipc.c es nuestro modulo de librería.

<sup>13</sup> El proceso que llame a la función msgrecv() será bloqueado por el sistema operativo hasta que éste sea notificado de que hay -al menos- un mensaje a ser leído de la cola de mensaje. Es la misma situación en donde un proceso requiere realizar una operación de entrada - salida (I/O) que requiera el uso de DMA (Direct Memory Access); dicho proceso quedará bloqueado hasta que se complete la operación de entrada - salida.

<sup>14</sup> Se refiere a abrir una terminal para acceder al intérprete de comandos (shell) del sistema operativo.

<sup>15</sup> Con el comando Unix man puede consultar los manuales en línea del sistema operativo, el cual también incluye documentación de funciones C/C++ y llamadas al sistema (system calls); siempre y cuando tenga instalada la documentación en su sistema. En el caso de Debian, recomendamos que instale el paquete (package) gcc-doc.

```
...
grchere@debian:~/gccwork/src/msg$ ipcs

----- Shared Memory Segments -----
key      shmid  owner    perms      bytes  nattch  status
0x00000000 32768  grchere  600  196608  2  dest
0x00000000 65537  grchere  600  196608  2  dest
0x00000000 98306  grchere  600  393216  2  dest
0x00000000 131075 grchere  600  393216  2  dest
0x00000000 163844 grchere  600  12288  2  dest
0x00000000 196613 grchere  600  196608  2  dest
0x00000000 229382 grchere  600  393216  2  dest
0x00000000 262151 grchere  600  393216  2  dest
0x00000000 360456 grchere  600  393216  2  dest
0x00000000 393225 grchere  600  196608  2  dest
0x00000000 720906 grchere  600  393216  2  dest
0x00000000 753675 grchere  600  196608  2  dest

----- Semaphore Arrays -----
key      semid  owner    perms      nsems

----- Message Queues -----
key      msgqid  owner    perms      used-bytes  messages
grchere@debian:~/gccwork/src/msg$
```

Tendrá que presionar **Ctrl+C** para interrumpir el programa servidor, en tal caso, podrá quedar la cola de mensajes creada, use el comando **ipcs** para destruirla en caso de que sea necesario.

3. Modifique el programa **tp71.c** para permitir una salida más elegante del mismo y que pueda borrar la cola de mensajes sin problemas, para ello implemente la señal **SIGUSR2** en combinación con una variable global (**volatile**) para indicar que el programa debe salir. Al enviar una señal **SIGUSR2** el programa servidor deberá detenerse:

```
grchere@debian:~/gccwork/src/msg$ ./tp71 &
main(): inicio servidor Cola de Mensajes!
main(): para salir envie señal SIGUSR2 a proceso 3265
main(): inicializo total de ventas de sucursales
main(): solicito clave ipc
main(): creo cola de mensajes con clave ipc
main(): quedo a la espera de mensajes en cola [98304]...
[1] 3265
grchere@debian:~/gccwork/src/msg$ ipcs

----- Shared Memory Segments -----
key      shmid  owner    perms      bytes  nattch  status
0x00000000 32768  grchere  600  196608  2  dest
0x00000000 65537  grchere  600  196608  2  dest
0x00000000 98306  grchere  600  393216  2  dest
0x00000000 131075 grchere  600  393216  2  dest
0x00000000 163844 grchere  600  12288  2  dest
0x00000000 196613 grchere  600  196608  2  dest
0x00000000 229382 grchere  600  393216  2  dest
```

```

0x00000000 262151 grchere 600 393216 2 dest
0x00000000 360456 grchere 600 393216 2 dest
0x00000000 393225 grchere 600 196608 2 dest
0x00000000 720906 grchere 600 393216 2 dest
0x00000000 753675 grchere 600 196608 2 dest
0x00000000 851980 grchere 600 393216 2 dest
0x00000000 884749 grchere 600 196608 2 dest

----- Semaphore Arrays -----
key semid owner perms nsems
----- Message Queues -----
key msgid owner perms used-bytes messages
0x6d0229fa 98304 grchere 660 0 0

grchere@debian:~/gccwork/src/msg$ kill -SIGUSR2 3265
grchere@debian:~/gccwork/src/msg$ main():recibi señal de salida
main():fin recepcion de mensajes, borra cola [98304]
main():fin servidor Cola de Mensajes| retorno=0

[1]+ Done ./tp71
grchere@debian:~/gccwork/src/msg$
```

4. Crear el programa `tp72.c`. Este programa será el cliente de `tp71.c`. Hágalo según el diseño indicado en la introducción. Este programa solicitará los totales de ventas de 10 sucursales, indicando su id de proceso y número de sucursal, enviando mensajes de tipo 1 y por cada envío esperará la respuesta, mostrando lo recibido del servidor en la consola. Fácilmente puede armar el mensaje a enviar al servidor utilizando la función `snprintf()` de la siguiente forma:

```
...
snprintf(qbuf.mtext, SUC_MSG_LEN, "<%d/%d>", pid, suc);
...
```

Como siempre, trabaje -al menos- con dos consolas, una para el servidor y otra para el cliente, podrá obtener salidas como estas:

```
consola server:
grchere@debian:~/gccwork/src/msg$ ./tp71 &
main():inicio servidor Cola de Mensajes!
main():para salir envie señal SIGUSR2 a proceso 3316
main():inicializo total de ventas de sucursales
main():solicito clave ipc
main():creo cola de mensajes con clave ipc
main():quedo a la espera de mensajes en cola [131072]...
[1] 3316
grchere@debian:~/gccwork/src/msg$ main():recibi [<3332/0>] como peticion
main():recibi [<3332/1>] como peticion
main():recibi [<3332/2>] como peticion
main():recibi [<3332/3>] como peticion
main():recibi [<3332/4>] como peticion
main():recibi [<3332/5>] como peticion
```

```
main():recibi [<3332/6>] como peticion
main():recibi [<3332/7>] como peticion
main():recibi [<3332/8>] como peticion
main():recibi [<3332/9>] como peticion

grchere@debian:~/gccwork/src/msg$
```

```
consola cliente:
grchere@debian:~/gccwork/src/msg$ ./tp72
main():inicio cliente Cola de Mensajes!
main():solicito clave ipc
main():obtengo cola de mensajes con clave ipc
main():envie [<3332/0>] Ok, ahora espero respuesta
main():envie [<3332/1>] Ok, ahora espero respuesta
main():envie [<3332/2>] Ok, ahora espero respuesta
main():envie [<3332/3>] Ok, ahora espero respuesta
main():envie [<3332/4>] Ok, ahora espero respuesta
main():envie [<3332/5>] Ok, ahora espero respuesta
main():envie [<3332/6>] Ok, ahora espero respuesta
main():envie [<3332/7>] Ok, ahora espero respuesta
main():envie [<3332/8>] Ok, ahora espero respuesta
main():envie [<3332/9>] Ok, ahora espero respuesta
main():fin envio de mensajes a cola [131072]
main():fin cliente Cola de Mensajes! retorno=0
grchere@debian:~/gccwork/src/msg$
```

5. El programa servidor requiere hacer un *parsing* de lo recibido. Copie tp71.c en tp73.c. Vamos a trabajar sobre tp73.c para hacer una nueva versión del servidor. Este código ya lo hemos hecho, debemos reutilizarlo. Si Ud. tiene conocimientos en cuanto a la compilación y creación de librerías estáticas<sup>16</sup>, podría hacer una partiendo de lo hecho en el Capítulo VI. Tuberías II (FIFO's), para reutilizar el código de *parsing*. Caso contrario, para no complejizar aún más este proceso, puede copiar dicho código (myfifo.h, myfifo.c) a su directorio de trabajo y compilar este programa con esta librería (gcc -Wall -o tp73 tp73.c myipc.c myfifo.c). Cabe dejar bien en claro que esto no es reutilizar código, lo cual supone la existencia de código en un único lugar a ser compartido en n proyectos.

Modificar tp73.c, usar strcpyentre() para hacer *parsing* de petición de clientes, obtener id proceso y sucursal, tomar el total de ventas de dicha sucursal y enviar mensaje de respuesta a cliente<sup>17</sup>. Compilar con la librería myfifo.c. El programa cliente también requiere algunos cambios. Copiar tp72.c en tp74.c. Vamos a trabajar sobre tp74.c para hacer una nueva versión del cliente. El cliente deberá mostrar el mensaje recibido del servidor, luego del envío de la petición.

Podrá obtener una salida como esta:

<sup>16</sup> Se recomienda la lectura de todo el apunte ubicado en <http://www.grch.com/docs/p1/ProgramacionI-2008-gcc.pdf> titulado "Trabajando con el Compilador GNU C/C++ GCC" [Chenicio], allí encontrará la forma de crear un ambiente de trabajo simple y la forma de crear/catalogar librerías a ser reutilizadas en n proyectos.

<sup>17</sup> Obsérvese que tal vez Ud. no encontró sentido en los cambios propuestos en el ejercicio 10 del Capítulo VI, en cuanto a la eliminación de código repetitivo y poco reusable. Si no se hubieran implementado esos cambios, aquí no podríamos hacer esta reutilización de código. De eso se trata la programación.

```
consola server:  
grchere@debian:~/gccwork/src/msg$ ./tp73 &  
main(): inicio servidor Cola de Mensajes!  
main(): para salir envie señal SIGUSR2 a proceso 3461  
main(): inicializo total de ventas de sucursales  
main(): solicito clave ipc  
main(): creo cola de mensajes con clave ipc  
main(): quedo a la espera de mensajes en cola [196608]...  
[1] 3461  
grchere@debian:~/gccwork/src/msg$ main():recibi [<3463/0>] como peticion  
main():envie [<0/176.74>] como respuesta a peticion  
main():recibi [<3463/1>] como peticion  
main():envie [<1/139.95>] como respuesta a peticion  
main():recibi [<3463/2>] como peticion  
main():envie [<2/53.98>] como respuesta a peticion  
main():recibi [<3463/3>] como peticion  
main():envie [<3/153.95>] como respuesta a peticion  
main():recibi [<3463/4>] como peticion  
main():envie [<4/184.47>] como respuesta a peticion  
main():recibi [<3463/5>] como peticion  
main():envie [<5/53.13>] como respuesta a peticion  
main():recibi [<3463/6>] como peticion  
main():envie [<6/76.13>] como respuesta a peticion  
main():recibi [<3463/7>] como peticion  
main():envie [<7/129.18>] como respuesta a peticion  
main():recibi [<3463/8>] como peticion  
main():envie [<8/139.09>] como respuesta a peticion  
main():recibi [<3463/9>] como peticion  
main():envie [<9/38.64>] como respuesta a peticion  
  
grchere@debian:~/gccwork/src/msg$
```

```
consola cliente:  
grchere@debian:~/gccwork/src/msg$ ./tp74  
main(): inicio cliente Cola de Mensajes!  
main(): solicito clave ipc  
main(): obtengo cola de mensajes con clave ipc  
main():envie [<3463/0>] Ok, ahora espero respuesta  
main():recibi [<0/176.74>] del servidor de cola de mensajes  
main():envie [<3463/1>] Ok, ahora espero respuesta  
main():recibi [<1/139.95>] del servidor de cola de mensajes  
main():envie [<3463/2>] Ok, ahora espero respuesta  
main():recibi [<2/53.98>] del servidor de cola de mensajes  
main():envie [<3463/3>] Ok, ahora espero respuesta  
main():recibi [<3/153.95>] del servidor de cola de mensajes  
main():envie [<3463/4>] Ok, ahora espero respuesta  
main():recibi [<4/184.47>] del servidor de cola de mensajes  
main():envie [<3463/5>] Ok, ahora espero respuesta  
main():recibi [<5/53.13>] del servidor de cola de mensajes  
main():envie [<3463/6>] Ok, ahora espero respuesta  
main():recibi [<6/76.13>] del servidor de cola de mensajes  
main():envie [<3463/7>] Ok, ahora espero respuesta  
main():recibi [<7/129.18>] del servidor de cola de mensajes  
main():envie [<3463/8>] Ok, ahora espero respuesta  
main():recibi [<8/139.09>] del servidor de cola de mensajes  
main():envie [<3463/9>] Ok, ahora espero respuesta
```

```
main():recibi [<9/38,64>] del servidor de cola de mensajes
main():fin envio de mensajes a cola [196608]
main():fin cliente Cola de Mensajes! retorno=0
grchere@debian:~/gccwork/src/msg$
```

6. Teniendo en cuenta lo aprendido en el Capítulo IV. Señales (Signals), se recomienda poner una alarma de  $n$  segundos en el programa cliente (capturar la señal SIGALRM, combinado con la función alarm()), ya que, si el servidor nunca contesta la petición del cliente, éste quedará bloqueado (en la llamada a la función msgrecv()) esperando una respuesta que nunca llegará.

## Capítulo VIII. IPC, Memoria Compartida y Semáforos (Shared Memory & Semaphores).

**Objetivo:** Desarrollar una serie de programas que permitan comprobar distintas funcionalidades vinculadas con la gestión de memoria compartida y semáforos de tipo System V Release 4 (SVR4) como una forma de comunicación entre procesos (IPC: Inter Process Communication) provistas por el kernel del SO. Introducir al desarrollo de programas servidores y clientes, productores y consumidores. Aplicar la solución propuesta en la teoría en cuanto al problema del Productor - Consumidor.

### Introducción

La memoria compartida es una de las técnicas de IPC más utilizadas y eficientes puesto que los datos están disponibles para todos los procesos y no requieren que éstos realicen "copias privadas" de los datos a intercambiar. La implementación de distintos tipos de servidores y soluciones empresariales basadas en memoria compartida plantea el problema de la exclusión mutua, requiriendo de semáforos para el control de concurrencia entre procesos.

En 1965 se publica el tratado de Dijkstra<sup>18</sup> cuyo principio fundamental es: *"dos o más procesos pueden cooperar por medio de simples señales, de forma tal, que un proceso pueda ser obligado a parar en un lugar determinado hasta que haya recibido una señal específica. Cualquier requisito complejo de coordinación puede ser satisfecho con la estructura de señales apropiada"*. Los semáforos son las señales a las cuales se refería Dijkstra; enviar una señal al semáforo `s = semSignal(s)` y detener el proceso actual hasta recibir la señal del semáforo `s = semWait(s)`.

El Analista Y pretende diseñar un servidor que actualice los totales de ventas de las sucursales en un área de memoria compartida, de esta forma, otros procesos clientes podrán consultarla. Se simularán los totales de ventas de las sucursales y éstas se almacenarán como un arreglo de 10 posiciones de tipo double. Estos totales son información resumida en base a los datos de facturación de las sucursales.

Implementar un área de memoria compartida, implica obtener una clave IPC (tal como se comentó en el capítulo anterior), luego se debe crear el área de memoria compartida usando dicha clave e indicando su tamaño en bytes y los permisos sobre la misma (tal como si se tratase de una cola de mensajes, archivo, proceso, etc.). Una vez creada la memoria compartida, ésta es accesible a través de un puntero local, es decir, nuestro programa obtendrá una referencia al área de memoria compartida previamente creada (nos hemos conectado con el área de memoria compartida). A través de este puntero accedemos a la memoria compartida, nos desplazamos, leemos y cambiamos dicha memoria compartida como si fuese un área de memoria local de nuestro proceso. Estos cambios están disponibles para todos los procesos que acceden a dicha memoria. Por último, debemos desconectar la memoria compartida. En el caso del

<sup>18</sup> Dijkstra, E., "Cooperating Sequential Processes", Technological University, Eindhoven, The Netherlands, 1965.

servidor, cuando éste termine su tarea, se supone no habrá clientes por atender, por lo tanto, borrará o destruirá la memoria compartida creada originalmente.

La descripción anterior coincide con los siguientes diseños:

**Programa servidor:**

```
main()
    crear clave IPC (según path y id proyecto)
    crear Memoria usando clave,tamaño,permisos
    si pude crear Memoria Ok entonces
        conectar Memoria a través de pventas (puntero double *)
        si pude conectar Memoria Ok entonces
            inicializar Memoria (pongo vector en cero)
            generar totales de ventas de sucursales usando pventas
            mostrar totales de ventas
            mientras no salir
                bloquear proceso hasta recibir señal (pause())
                fin mientras
                desconectar Memoria
            sino
                error conectando Memoria
            fin si
            borrar Memoria
        sino
            error creando Memoria
    fin si
fin main()
```

**Programa cliente:**

```
main()
    crear clave de IPC (según path y id proyecto)
    obtener Memoria usando clave,permisos
    si pude obtener Memoria Ok entonces
        conectar Memoria a través de pcliente (puntero double *)
        si pude conectar Memoria Ok entonces
            mostrar totales de ventas usando pcliente
            desconectar Memoria
        sino
            error conectando Memoria
        fin si
    sino
        error creando Memoria
    fin si
fin main()
```

1. El Analista de Sistemas Y está muy entusiasmado con memoria compartida y semáforos; está convencido de que este no será el último desarrollo en esta línea, por lo tanto, se propone ampliar su librería para el manejo de IPC: `myipc.c` cuyo archivo de cabecera es `myipc.h`. Aquí nuevamente, aplicamos el criterio indicado en el punto 5 del Capítulo VII. IPC, Cola de Mensajes (Queues) (puede copiar la librería o recompilarla). Este archivo de cabecera y librería será compartido tanto por cliente como por servidor, por lo tanto, aquí podemos acordar el path y id de proyecto, en este

caso, usaremos otra clave para Memoria Compartida, así que dentro de myipc.h deberemos agregar:

```
#define IPC_MKEY 'h'
```

También se deberán agregar dentro de myipc.h los archivos de cabecera necesarios para utilizar semáforos y memoria compartida:

```
...
#include <sys/shm.h>
#include <sys/sem.h>
...
```

Los prototipos de las funciones a implementar dentro de myipc.c serán:

```
int borrar_memoria(int shmid);
void *conectar_memoria(int shmid, int modo);
int crear_memoria(key_t key, int size, int modo);
int desconectar_memoria(void *buffer);
/* Obtiene memoria compartida previamente creada */
int obtener_memoria(key_t key, int modo);
```

Tanto cliente como servidor ejecutarán:

```
key_t clave = obtener_clave(IPC_PATH, IPC_MKEY);
```

para generar una clave IPC con la cual identificar a la memoria compartida.

El servidor creará la memoria compartida en modo lectura-escritura:

```
int modo = SHM_R|SHM_W; // lectura y escritura
int tamaño = sizeof(double)*10; // son 10 totales de ventas
idmem = crear_memoria(clave, tamaño, modo);
```

El cliente se conectará a una memoria previamente creada en modo lectura:

```
int modo = SHM_R;
idmem = obtener_memoria(clave, modo);
```

Estas funciones devolverán -1 cuando exista alguna condición de error.

Tanto cliente como servidor deberán conectar a la memoria compartida:

```
pventas = (double *) conectar_memoria(idmem, modo);
```

El servidor inicializará el total de ventas de las sucursales y generará los totales:

```
memset(pventas, 0, tamaño);
init(pventas);
```

También es posible agregar otras funciones compartidas y reutilizables a la librería myipc.c, por ejemplo:

```
void mostrar_ventas(double *ventas);
```

esta función permite mostrar el total de ventas de todas las sucursales por la consola, a partir de un puntero al primer total de ventas.

De forma muy similar y consistente con la cola de mensajes, para crear o acceder a un área de memoria compartida debemos utilizar la función `shmget()`, cuando se indica tamaño, implica una creación de un área de memoria compartida y cuando el tamaño es 0, implica un acceso a la misma. Para evitar confusión y errores de los programadores, el Analista Y ha decidido usar dos entradas distintas en la API de `myipc` (`crear_memoria()` y `obtener_memoria()`) para estas dos situaciones:

```
int crear_memoria(key_t key,int size,int modo) {
    return shmget(key, size, IPC_CREAT|modo);
}
int obtener_memoria(key_t key,int modo) {
    return crear_memoria(key,0,modo);
}
```

Se consideraría incorrecto implementar `obtener_memoria()` en función de `shmget()`<sup>19</sup>, esto atentaría contra el reuso y las ventajas que éste nos otorga: si modificamos `crear_memoria()` automáticamente también habremos corregido `obtener_memoria()`.

Para conectar a un área de memoria compartida usamos `shmat()` (share memory attach) y `shmdt()` para desconectarnos (share memory detach):

```
void *conectar_memoria(int shmid,int modo) {
    return shmat(shmid,0,modo);
}
int desconectar_memoria(void *buffer) {
    return shmdt(buffer);
}
```

De igual forma que con la cola de mensajes, una vez que la hemos creado, podemos cambiar algunas características de la misma, utilizando la función `shmctl()`, ésta también nos servirá para destruir el área de memoria compartida:

```
int borrar_memoria(int shmid) {
    return shmctl(shmid, IPC_RMID, 0);
}
```

La API requerida para manipular un área de memoria compartida es aún más simple que el de una cola de mensajes. Puede completar el resto de la API, hacer un pequeño programa de prueba de la librería y compilar el código.

---

<sup>19</sup> Siendo posible el no hacerlo.

2. Utilice el programa de prueba de la librería del punto anterior y cópíelo como `tp81.c`. Implemente en este programa el servidor de memoria compartida, tal como se indicó en la introducción.

Esta primera versión simplemente creará el área de memoria compartida, la inicializará, generará los totales de ventas, entrará en el loop principal y utilizará la función `pause()` para bloquear el proceso hasta que se llegue una señal. Implemente la señal `SIGUSR2` en combinación con la variable `salir (volatile)` para evitar que el servidor quede bloqueado indefinidamente.

Compile y ejecute `tp81.c`. Podrá obtener una salida como esta:

```
consola server:
grchere@debian:~/gccwork/src/sem$ ipcs
----- Shared Memory Segments -----
key      shmid      owner      perms      bytes      nattch      status
0x00000000 32768      grchere    600      196608      2      dest
0x00000000 65537      grchere    600      196608      2      dest
0x00000000 98306      grchere    600      393216      2      dest
0x00000000 131075      grchere    600      393216      2      dest
0x00000000 163844      grchere    600      12288      2      dest
0x00000000 262149      grchere    600      393216      2      dest
0x00000000 294918      grchere    600      196608      2      dest
0x00000000 327687      grchere    600      393216      2      dest

----- Semaphore Arrays -----
key      semid      owner      perms      nsems

----- Message Queues -----
key      msqid      owner      perms      used-bytes      messages
grchere@debian:~/gccwork/src/sem$ 

grchere@debian:~/gccwork/src/sem$ ./tp81
main(): inicio proceso productor!
main(): para salir envie señal SIGUSR2 a proceso 2894
main(): solicito clave ipc
main(): creo memoria compartida de 80 bytes
main(): attachar memoria local a memoria compartida 393224
main(): Inicializo memoria compartida a cero
main(): genero total de ventas de sucursales en memoria compartida
mostrar_ventas(): sucursal[0]=105.933955
mostrar_ventas(): sucursal[1]=118.313756
mostrar_ventas(): sucursal[2]=172.562817
mostrar_ventas(): sucursal[3]=207.236120
mostrar_ventas(): sucursal[4]=19.042746
mostrar_ventas(): sucursal[5]=134.783351
mostrar_ventas(): sucursal[6]=206.104176
mostrar_ventas(): sucursal[7]=46.965428
mostrar_ventas(): sucursal[8]=117.049172
mostrar_ventas(): sucursal[9]=96.927325
main(): Bloqueo proceso hasta recibir señal de salida
```

```
consola cliente:
grchere@debian:~$ ipcs
```

```

----- Shared Memory Segments -----
key      shmid      owner      perms      bytes      nattch      status
0x00000000 32768      grchere    600      196608      2      dest
0x00000000 65537      grchere    600      196608      2      dest
0x00000000 98306      grchere    600      393216      2      dest
0x00000000 131075     grchere    600      393216      2      dest
0x00000000 163844     grchere    600      12288       2      dest
0x00000000 262149     grchere    600      393216      2      dest
0x00000000 294918     grchere    600      196608      2      dest
0x00000000 327687     grchere    600      393216      2      dest
0x69022a15 393224     grchere    600      80          1

----- Semaphore Arrays -----
key      semid      owner      perms      nsems

----- Message Queues -----
key      msgqid      owner      perms      used-bytes      messages

grchere@debian:~$ grchere@debian:~$ kill -SIGUSR2 2894

```

luego de enviada la señal SIGUSR2:

```

consola server:
main():recibi señal de salida
main():Desconecto memoria compartida
main():Destruyo memoria compartida
main():fin proceso productor1 retorno=0
grchere@debian:~/gccwork/src/sem$
```

```

consola cliente:
grchere@debian:~$ ipcs

----- Shared Memory Segments -----
key      shmid      owner      perms      bytes      nattch      status
0x00000000 32768      grchere    600      196608      2      dest
0x00000000 65537      grchere    600      196608      2      dest
0x00000000 98306      grchere    600      393216      2      dest
0x00000000 131075     grchere    600      393216      2      dest
0x00000000 163844     grchere    600      12288       2      dest
0x00000000 262149     grchere    600      393216      2      dest
0x00000000 294918     grchere    600      196608      2      dest
0x00000000 327687     grchere    600      393216      2      dest

----- Semaphore Arrays -----
key      semid      owner      perms      nsems

----- Message Queues -----
key      msgqid      owner      perms      used-bytes      messages

grchere@debian:~$
```

3. Copiar tp81.c en tp82.c. . Este programa será el cliente de tp81.c. Hágalo según lo indicado en la introducción. Tp82.c es un cliente que accede a la memoria compartida y muestra los valores allí contenidos, no tiene datos globales, no especifica

tamaño de memoria compartida, el modo de acceso es SHM\_R, no olvide utilizar obtener\_memoria() en vez de crear\_memoria(). Conecta la memoria compartida a través de un puntero local, usa el puntero local para mostrar las ventas (mostrar\_ventas()), desconecta la memoria compartida y no implementa señales.

Podrá obtener una salida como esta:

```
consola server:
grchere@debian:~/gccwork/src/sem$ ./tp81
main(): inicio proceso productor
main(): para salir envie señal SIGUSR2 a proceso 2986
main(): solicito clave ipc
main(): creo memoria compartida de 80 bytes
main(): attachar memoria local a memoria compartida 458760
main(): Inicializo memoria compartida a cero
main(): genero total de ventas de sucursales en memoria compartida
mostrar_ventas(): sucursal[0]=105.933955
mostrar_ventas(): sucursal[1]=118.313756
mostrar_ventas(): sucursal[2]=172.562817
mostrar_ventas(): sucursal[3]=207.236120
mostrar_ventas(): sucursal[4]=19.042746
mostrar_ventas(): sucursal[5]=134.783351
mostrar_ventas(): sucursal[6]=206.104176
mostrar_ventas(): sucursal[7]=46.965428
mostrar_ventas(): sucursal[8]=117.049172
mostrar_ventas(): sucursal[9]=96.927325
main(): Bloqueo proceso hasta recibir señal de salida
main(): recibi señal de salida
main(): Desconecto memoria compartida
main(): Destruyo memoria compartida
main(): fin proceso productor! retorno=0
grchere@debian:~/gccwork/src/sem$ 

consola cliente:
grchere@debian:~/gccwork/src/sem$ ./tp82
main(): inicio proceso consumidor
main(): para salir envie señal SIGUSR2 a proceso 2987
main(): solicito clave ipc
main(): obtengo memoria compartida
main(): attachar memoria local a memoria compartida 458760
main(): leo memoria compartida
mostrar_ventas(): sucursal[0]=105.933955
mostrar_ventas(): sucursal[1]=118.313756
mostrar_ventas(): sucursal[2]=172.562817
mostrar_ventas(): sucursal[3]=207.236120
mostrar_ventas(): sucursal[4]=19.042746
mostrar_ventas(): sucursal[5]=134.783351
mostrar_ventas(): sucursal[6]=206.104176
mostrar_ventas(): sucursal[7]=46.965428
mostrar_ventas(): sucursal[8]=117.049172
mostrar_ventas(): sucursal[9]=96.927325
main(): Desconecto memoria compartida
main(): fin proceso consumidor! retorno=0
grchere@debian:~/gccwork/src/sem$ kill -SIGUSR2 2986
```

4. Copiar tp81.c en tp83.c. El servidor tp81.c mostraba siempre los mismos totales de ventas. El Analista Y sabe que estos datos son muy dinámicos. Implemente en tp83.c una nueva versión del servidor, ahora el servidor permite cambiar los datos de los totales de ventas de las sucursales cada vez que recibe la señal SIGUSR1.

Esta nueva versión de servidor (más real que la anterior) plantea un problema: tp82 (el cliente) podría ver en una misma consulta- un "mix" de valores actualizados y no actualizados de totales de ventas de sucursales. Esto no compromete al sistema, ni a los procesos, puesto que el acceso de todos los clientes es de lectura. Si no se desea este efecto, es decir, se desea que, ante una consulta de un cliente, ésta muestre todos los datos actualizados de todas las sucursales. De ser así, se requiere de un mecanismo de exclusión mutua: un cliente no puede consultar datos de sucursales cuando estos datos están siendo actualizados por el servidor. Sólo por ahora, no vamos a preocuparnos de este problema.

Sugerencia: utilizar un puntero global que apunte al vector de totales de ventas almacenado en la memoria compartida, cuando se reciba la señal SIGUSR1, recalcular los valores sumando un 10% al valor actual (suponiendo que p es un dato local y pventas es el puntero global):

```
...
double *p = pventas;
int i; for(i=0;i<10;i++,p++) *p *= 1.1;
...
```

Podrá obtener una salida como esta:

```
consola server:
grchere@debian:~/gccwork/src/sem$ ./tp83
main(): inicio proceso productor
main(): para recalcular envie señal SIGUSR1 a proceso 2904
main(): para salir envie señal SIGUSR2 a proceso 2904
main(): solicito clave ipc
main(): creo memoria compartida de 80 bytes
main(): attachar memoria local a memoria compartida 360455
main(): Inicializo memoria compartida a cero
main(): genero total de ventas de sucursales en memoria compartida
mostrar_ventas(): sucursal[0]=116.592117
mostrar_ventas(): sucursal[1]=10.888748
mostrar_ventas(): sucursal[2]=151.835415
mostrar_ventas(): sucursal[3]=59.417490
mostrar_ventas(): sucursal[4]=110.724445
mostrar_ventas(): sucursal[5]=188.109783
mostrar_ventas(): sucursal[6]=193.502843
mostrar_ventas(): sucursal[7]=87.824307
mostrar_ventas(): sucursal[8]=210.795342
mostrar_ventas(): sucursal[9]=57.125540
main(): Bloqueo proceso hasta recibir señal de salida

(luego de recibir SIGUSR1:)

mostrar_ventas(): sucursal[0]=128.251329
mostrar_ventas(): sucursal[1]=11.977622
```

```

mostrar_ventas(): sucursal[2]=167.010956
mostrar_ventas(): sucursal[3]=65.359239
mostrar_ventas(): sucursal[4]=121.796890
mostrar_ventas(): sucursal[5]=206.920762
mostrar_ventas(): sucursal[6]=212.853127
mostrar_ventas(): sucursal[7]=96.606737
mostrar_ventas(): sucursal[8]=231.874876
mostrar_ventas(): sucursal[9]=62.838094
...
main():recibi señal de salida
main():Desconecto memoria compartida
main():Destruyo memoria compartida
main():fin proceso productor1 retorno=0

consola cliente:
grchere@debian:~/gccwork/src/sem$ kill -SIGUSR1 2904
grchere@debian:~/gccwork/src/sem$
```

5. Hasta aquí, hemos realizado lectura y escritura de memoria compartida, con la posible necesidad de exclusión mutua. Copiar tp83.c como tp84p.c, para implementar una nueva versión de este servidor.

La Empresa X tiene un sistema que va registrando las ventas que ocurren en las distintas sucursales. Modifique tp84p.c para que éste guarde (junto con el vector de totales de ventas por sucursal) en la misma área de memoria compartida (a medida que se van reportando las ventas) los siguientes datos que representan una venta en una sucursal:

sucursal (int, número de sucursal)  
fecha (almacenada como un conjunto de caracteres con el formato: dd/mm/yyyy)  
factura (int, número de factura)  
monto (double, importe total de la factura)

estos datos pueden representarse a través de la estructura sucta:

```

struct sucta {
    int sucursal;
    char fecha[11]; // almacena tambien el \0 final de la cadena de caracteres
    int factura;
    double monto;
};
```

luego podemos, por ejemplo, declarar un puntero a una estructura de dicho tipo:

```

...
    struct sucta *psucta;
...
```

a medida que llegan las ventas, este proceso las ubica en un buffer de 10 elementos, es decir, que este programa tiene capacidad para "producir"<sup>20</sup> y almacenar hasta un

<sup>20</sup> Efectivamente, tp84p se trata de un proceso productor. De allí el cambio de nomenclatura en su nombre.

máximo de 10 ventas<sup>21</sup> realizadas en alguna de las sucursales de la Empresa X. Paralelamente, se continúa manteniendo otro buffer de 10 elementos con los totales de ventas de las sucursales (al igual que en tp81). Ambos se encuentran en la misma área de memoria compartida de la siguiente forma:

Memoria Compartida										n bytes									
0										...									
arreglo de 10 elementos para los totales de ventas de las sucursales										arreglo de 10 elementos para las ventas de las sucursales									
d	ouble	d	ouble	d	ouble	d	ouble	d	ouble	s	ucvta	s	ucvta	s	ucvta	s	ucvta	s	ucvta

ambos arreglos se inicializan a 0 cuando comienza el proceso. Por lo tanto, el nuevo tamaño de memoria compartida sería:

```
...
int tamaño = (sizeof(double)*10)+(sizeof(struct sucvta)*10);
...
```

si pventas continúa siendo un puntero al primer arreglo de 10 elementos:

```
...
pventas = (double *) conectar_memoria(idmem, modo);
...
```

Entonces, psucvta (puntero al segundo arreglo en memoria compartida) sería:

```
...
psucvta = (struct sucvta *) (pventas + sizeof(double)*10);
...
```

debemos sumarle el tamaño del primer arreglo<sup>22</sup> para que este puntero quede apuntando al comienzo del segundo arreglo. Es nuestra responsabilidad como programadores hacer un manejo correcto de estos punteros, sin superar ninguno de sus límites<sup>23</sup>.

Se puede implementar la función void generar\_venta(struct sucvta \*) para simular una venta de una sucursal de la siguiente forma:

```
void generar_venta(struct sucvta *sv) {
    sv->sucursal = (int) ((long) random() % 10L);
    int dia = (int) ((long) random() % 31L);
    int mes = (int) ((long) random() % 12L)+1;
```

<sup>21</sup> Para ser más precisos, tp84p es un proceso productor con buffer acotado.

<sup>22</sup> A esto se lo denomina "desplazamiento" (offset).

<sup>23</sup> Suele ser la mayor fuente de errores en la programación C/C++, al mismo tiempo, es una de las características que hacen a la performance, la elegancia y lo compacto del código C.

```

int anio = 2009;
snprintf(sv->fecha,11,"%2d/%2d/%4d",dia,mes,anio);
sv->factura = (int) ((long) random() / 1000000L;
sv->monto = ((double) random() / 10000000.0;
}

```

A medida que el servidor recibe ventas de sucursales (genera ventas), se deberá actualizar el correspondiente total de ventas de la sucursal:

```

...
p = psucvta;
for (i=0;i<10;i++,p++) {
    generar_venta(p);
    *(pventas+p->sucursal) += p->monto; // acumulo venta
}
...

```

p es de tipo struct sucvta \* y pventas continua siendo un puntero al primer elemento del arreglo de totales de ventas; obsérvese que pventas + <número de sucursas> es el desplazamiento necesario en memoria para actualizar en el lugar correcto el total de ventas de la sucursal.

Generar ventas cada 10 segundos aproximadamente. Para ello utilizar la función sleep (sleep(10);) dentro del loop principal del servidor, luego de generar las ventas y actualizar los totales de ventas.

Podrá obtener una salida como esta:

```

consola server:
grchere@debian:~/gccwork/src/sem$ ./tp84p
main(): inicio proceso productor
main(): para salir envie señal SIGUSR2 a proceso 3037
main(): solicito clave ipc
main(): creo memoria compartida de 360 bytes
main(): attachar memoria local a memoria compartida 458759
main(): Inicializo memoria compartida a cero
main(): genero total de ventas de sucursales en memoria compartida
main(): Bloqueo proceso hasta recibir señal de salida
mostrar_ventas(): sucursal[0]=352.309974
mostrar_ventas(): sucursal[1]=555.469024
mostrar_ventas(): sucursal[2]=129.552290
mostrar_ventas(): sucursal[3]=659.881827
mostrar_ventas(): sucursal[4]=1056.077428
mostrar_ventas(): sucursal[5]=181.898165
mostrar_ventas(): sucursal[6]=1130.669420
mostrar_ventas(): sucursal[7]=522.252935
mostrar_ventas(): sucursal[8]=741.985095
mostrar_ventas(): sucursal[9]=590.463744
mostrar_sucventas(): suc=[4] fecha=[27/ 4/2009] fac=[1595] monto=[197.529392]
mostrar_sucventas(): suc=[0] fecha=[27/ 8/2009] fac=[778] monto=[145.057775]
mostrar_sucventas(): suc=[8] fecha=[29/ 4/2009] fac=[607] monto=[38.361159]
mostrar_sucventas(): suc=[1] fecha=[ 4/ 5/2009] fac=[1546] monto=[173.807989]
mostrar_sucventas(): suc=[6] fecha=[ 2/ 6/2009] fac=[1294] monto=[160.439974]
mostrar_sucventas(): suc=[9] fecha=[10/ 6/2009] fac=[205] monto=[89.141984]
mostrar_sucventas(): suc=[6] fecha=[16/ 7/2009] fac=[806] monto=[42.004613]

```

```

mostrar_sucventas(): suc=[7] fecha=[27/ 2/2009] fac=[2034] monto=[61.111264]
mostrar_sucventas(): suc=[3] fecha=[27/ 3/2009] fac=[1620] monto=[24.583503]
mostrar_sucventas(): suc=[6] fecha=[16/ 7/2009] fac=[2055] monto=[206.386668]
mostrar_ventas(): sucursal[0]=639.126360
mostrar_ventas(): sucursal[1]=571.337362
mostrar_ventas(): sucursal[2]=202.827572
mostrar_ventas(): sucursal[3]=659.881827
mostrar_ventas(): sucursal[4]=1056.077428
mostrar_ventas(): sucursal[5]=213.922284
mostrar_ventas(): sucursal[6]=1310.140690
mostrar_ventas(): sucursal[7]=580.505195
mostrar_ventas(): sucursal[8]=848.977764
mostrar_ventas(): sucursal[9]=758.659658
mostrar_sucventas(): suc=[0] fecha=[15/ 9/2009] fac=[691] monto=[100.332799]
mostrar_sucventas(): suc=[9] fecha=[ 9/11/2009] fac=[1604] monto=[168.195914]
mostrar_sucventas(): suc=[7] fecha=[28/ 4/2009] fac=[2007] monto=[58.252260]
mostrar_sucventas(): suc=[6] fecha=[ 8/10/2009] fac=[367] monto=[173.445431]
mostrar_sucventas(): suc=[2] fecha=[ 8/ 6/2009] fac=[902] monto=[73.275281]
mostrar_sucventas(): suc=[8] fecha=[13/ 5/2009] fac=[1902] monto=[106.992669]
mostrar_sucventas(): suc=[5] fecha=[12/10/2009] fac=[1205] monto=[32.024119]
mostrar_sucventas(): suc=[1] fecha=[13/ 4/2009] fac=[351] monto=[15.868338]
mostrar_sucventas(): suc=[6] fecha=[27/ 8/2009] fac=[1837] monto=[6.025839]
mostrar_sucventas(): suc=[0] fecha=[22/ 9/2009] fac=[931] monto=[98.483587]
...
main():recibi señal de salida
main():Desconecto memoria compartida
main():Destruyo memoria compartida
main():fin proceso productor! retorno=0
grchere@debian:~/gccwork/src/sem$
```

6. Copiar tp82.c en tp84c.c. Modificar tp84c.c para implementar una nueva versión del programa cliente: la Empresa X tiene uno o más procesos cliente que acceden al área de memoria compartida (en donde el programa servidor genera y actualiza los datos) y retiran desde allí las ventas de las sucursales (cada vez que se retira una venta, esa porción de memoria compartida se inicializa a cero)<sup>24</sup>; los datos de ventas que se extraen de la memoria se guardan en un archivo secuencial para su posterior procesamiento. El nombre del archivo se forma con el id del proceso y la extensión .ventas, con un formato de texto, tal como:

```

...
char archivo[30];
snprintf(archivo,30,"%d.ventas",(int) getpid());
FILE *fpo=fopen(archivo,"a");
...
fclose (fpo);
```

la estructura interna del archivo es la siguiente:

```

número de sucursal (2 posiciones)
fecha (8 posiciones)
factura (10 posiciones numéricas)
monto (12 posiciones totales con punto y dos decimales)
```

<sup>24</sup> Tp84c es un proceso consumidor.

cada registro se separa con `\n` (new line, Intro), de forma tal, que es posible ver/editar su contenido con cualquier archivo de texto.

Puede lograr esto valiéndose de las funciones `fopen()`, `fclose()` (tal como se indicó anteriormente) y puede usar `fprintf()` para grabar los registros con el formato indicado previamente, de la siguiente forma (suponiendo que `sv` es un puntero de tipo `struct sucvta *` y apunta a la posición *i*ésima del vector de ventas de sucursales):

```
...
fprintf(salida, "%2d%s%10d%12.2lf\n",
        sv->sucursal, sv->fecha, sv->factura, sv->monto);
...
```

salida debe ser un stream<sup>25</sup>, algo de tipo `FILE *`. Todo programa C tiene acceso a tres streams (que no requieren ser abiertos en forma explícita): `stdio` (generalmente<sup>26</sup> el teclado), `stdout` (generalmente el monitor), `stderr` (generalmente el monitor). Por lo tanto se podría diseñar una función tal como `void mostrar_sucventas(FILE *, struct sucvta *)`, que nos sirva tanto para mostrar por pantalla como para enviar a un archivo:

```
...
FILE *fout = fopen("miarchivo.txt", "a");
struct sucvta *sv = ....
...
// para mostrar por pantalla
mostrar_sucventas(stdout, sv);
// para grabar en archivo
mostrar_sucventas(fout, sv)
...
fclose(fout);
...
```

Implemente el programa cliente consumidor `tp84c.c`, podrá obtener una salida como esta:

```
consola cliente:
grchere@debian:~/gccwork/src/sem$ ./tp84c
main(): inicio proceso consumidor
main(): solicito clave ipc
main(): obtengo memoria compartida
main(): attachar memoria local a memoria compartida 589831
main(): leo memoria compartida
mostrar_ventas(): sucursal[0]=397.650330
mostrar_ventas(): sucursal[1]=326.846774
mostrar_ventas(): sucursal[2]=5.049239
mostrar_ventas(): sucursal[3]=215.701141
mostrar_ventas(): sucursal[4]=282.191748
mostrar_ventas(): sucursal[5]=299.724144
```

<sup>25</sup> Stream se refiere a un flujo de datos unidireccional, secuencial.

<sup>26</sup> Se utiliza la palabra "generalmente" porque estos flujos o streams pueden ser redirigidos hacia o desde otro dispositivo o archivo.

```

mostrar_ventas(): sucursal[6]=480.924430
mostrar_ventas(): sucursal[7]=413.780045
mostrar_ventas(): sucursal[8]=191.408803
mostrar_ventas(): sucursal[9]=156.867319
mostrar_suventas(): suc=[5] fecha=[22/ 4/2009] fac=[1175] monto=[142.859295]
mostrar_suventas(): suc=[6] fecha=[11/10/2009] fac=[350] monto=[26.475312]
mostrar_suventas(): suc=[0] fecha=[18/ 5/2009] fac=[1268] monto=[30.115256]
mostrar_suventas(): suc=[9] fecha=[25/ 7/2009] fac=[1681] monto=[1.871950]
mostrar_suventas(): suc=[7] fecha=[17/ 4/2009] fac=[1927] monto=[191.437293]
mostrar_suventas(): suc=[6] fecha=[ 8/ 7/2009] fac=[573] monto=[208.439285]
mostrar_suventas(): suc=[1] fecha=[ 0/ 9/2009] fac=[1449] monto=[10.224205]
mostrar_suventas(): suc=[7] fecha=[16/11/2009] fac=[799] monto=[208.244878]
mostrar_suventas(): suc=[1] fecha=[26/ 6/2009] fac=[73] monto=[207.803250]
mostrar_suventas(): suc=[4] fecha=[11/ 5/2009] fac=[1770] monto=[61.441396]
main():Desconecto memoria compartida
main():fin proceso consumidor! retorno=0
grchere@debian:~/gccwork/src/sem$
```

7. La situación descripta en los ejercicios anteriores se denomina productor - consumidor con un buffer acotado. El primer proceso es el productor (tp84p.c), el segundo proceso (tp84c.c) es el consumidor. Ahora se requiere de la implementación de exclusión mutua, teniendo en cuenta de no producir cuando ya no hay lugar dónde guardar la producción y de no consumir cuando ya no hay nada que consumir.

Implementar ambos procesos utilizando memoria compartida y 3 semáforos según lo recomendado por William Stallings en su libro "Sistemas Operativos" 5ta. edición (español) en la página 227, Fig. 5.13, "Una solución al problema productor/consumidor con buffer acotado usando semáforos", la solución propuesta por Stallings es la siguiente:

```

/* programa productor consumidor */
semaphore s = 1;
semaphore n = 0;
semaphore e = /* tamaño del buffer */;
void productor() {
    while (true) {
        producir();
        semWait(e);
        semWait(s);
        anyadir();
        semSignal(s);
        semSignal(n);
    }
}
void consumidor() {
    while (true) {
        semWait(n);
        semWait(s);
        extraer();
        semSignal(s);
        semSignal(e);
        consumir();
    }
}
void main() {
    paralelos (productor,consumidor);
```

]

Probar la ejecución concurrente del productor y uno o más consumidores.

7.1) Los semáforos a utilizar serán de tipo System V Release 4 (SVR4)<sup>27</sup>, nuevamente, se debe ampliar la librería `myipc.c` para dar soporte a semáforos, se propone agregar la siguiente API básica:

```
int crear_semaforo(key_t key,int modo);
int borrar_semaforo(int semid);
int borrar_semaforos(int semid,int nsem);
int semSignalTo(int semid,int nro); // incrementar/decrementar en n un
semáforo
int semSignal(int semid); // incrementar un semáforo
int semWait(int semid); // decrementar un semáforo
int obtener_semaforo(key_t key,int modo);
```

debido al hecho de que, en SVR4 no se crea un semáforo sino un conjunto de semáforos, entonces se podría pensar en una ampliación de esta API básica para agregar otras funciones que trabajen sobre grupos de semáforos y algunas otras operaciones adicionales<sup>28</sup>:

```
int crear_semaforos(key_t key,int nsem,int modo);
int semSignalWaitTo(int semid,int nro);
int semSignalTo(int semid,int nro);
int semSignalWaitsTo(int semid,struct sembuf sb[],int nsem);
int semSignalsTo(int semid,struct sembuf sb[],int nsem);
int semSignalWaitsToNum(int semid,int nro,int nsem);
int semSignalsToNum(int semid,int nro,int nsem);
void obtener_claves(char *path[],char car[],key_t arrkey[],int nkeys);
```

Para especificar el tipo de uso que se pretende hacer del semáforo se pueden usar las macros `SEM_R` (para lectura) y `SEM_A` (para escritura, estas macros no se han encontrado en Linux (por lo menos, en la distribución Debian 5.01), en tal caso, podremos definirlas como:

```
// definición de SEM_R & SEM_A
#ifndef SEM_R
#define SEM_R 0400
#endif
#ifndef SEM_A
#define SEM_A 0200
#endif
```

Si va a crear un conjunto de semáforos, por cada semáforo propuesto por Stallings, se necesitará usar una clave IPC (obtenida a partir de un path más una letra) para cada semáforo (`IPC_SKEYs`, `IPC_SKEYn`, `IPC_SKEYe`). Las macros `SEM_s`,

<sup>27</sup> Vale la aclaración puesto que también existen los semáforos de tipo POSIX (prototipos en `semaphore.h`). Existen varias diferencias entre éstos y los semáforos SVR4 que aquí usaremos, en cuanto a su implementación, su performance, su portabilidad.

<sup>28</sup> Dejamos la implementación de estas funciones a cargo de aquel que quiera profundizar aún más sus conocimientos de la API de semáforos SVR4. Esta tarea está fuera del alcance de este trabajo.

`SEM_n`, `SEM_e`, indican la posición dentro del arreglo usado para facilitar la creación de semáforos como se verá más adelante:

```
#define IPC_SKEYs 's' // letra para semáforo s
#define IPC_SKEYn 'n' // letra para semáforo n
#define IPC_SKEYe 'e' // letra para semáforo e
#define SEM_s 0 // id (dentro del vector de semáforos) para el semáforo s
#define SEM_n 1 // id (dentro del vector de semáforos) para el semáforo n
#define SEM_e 2 // id (dentro del vector de semáforos) para el semáforo e
```

Nuevamente, debemos obtener una clave IPC y con ella intentar crear un conjunto de semáforos que se identificarán a través de un mismo id, para ello usaremos `semget()`, siguiendo la misma lógica de las restantes funciones de IPC:

```
// creamos semáforos "de a uno", a pesar de que la api de svr4
// permite crear n semáforos en una misma llamada
int crear_semaforo(key_t key,int modo) {
    return crear_semaforos(key,1,modo);
}
int crear_semaforos(key_t key,int nsem,int modo) {
    return semget(key, nsem, 0666|IPC_CREAT);
}
```

el o los procesos consumidores deberán obtener un semáforo previamente creado:

```
// obtenemos semáforos "de a uno", a pesar de que la api de svr4
// permite crear n semáforos en una misma llamada
int obtener_semaforo(key_t key,int modo) {
    return semget(key, 0, 0666);
}
```

es decir, que esta API crea un conjunto de 1 solo semáforo por vez y trabaja siempre sobre el semáforo número 0.

Para borrar un semáforo, análogamente utilizamos la función `semctl()`:

```
int borrar_semaforo(int semid) {
    return borrar_semaforos(semid,0);
}
int borrar_semaforos(int semid,int nsem) {
    return semctl(semid, nsem, IPC_RMID, 0);
}
```

Para incrementar o decrementar un semáforo:

```
// incrementar en 1 a un semáforo
int semSignal(int semid) {
    return semSignalTo(semid,1);
}
```

```

// decrementar en 1 a un semáforo
int semWait(int semid) {
    return semSignalTo(semid,-1);
}

// incrementar/decrementar en n a un semáforo
int semSignalTo(int semid,int nro) {
    struct sembuf sb[1];
    sb[0].sem_num = 0;
    sb[0].sem_op = nro;
    sb[0].sem_flg = SEM_UNDO | 0; //vuelve atrás el cambio si se cae este
proceso
    return semop(semid,sb,1);
}

```

en este caso hay alguna diferencia con lo visto anteriormente, puesto que interviene la función `semop()` (realizar operaciones sobre semáforos). Cabe aclarar que cada semáforo del conjunto se representa por una estructura de tipo `sembuf`. Se podrá utilizar un arreglo de tipo `sembuf` para indicar n semáforos (o al menos uno). El elemento `sem_num` coincide con el número de semáforo dentro del conjunto (0, 1, 2 ...), `sem_op` es el número a incrementar o decrementar (si es negativo) del semáforo y `sem_flg` indica el modo y otros parámetros, tales como `SEM_UNDO` que permite indicar que, en caso de que este proceso termine abruptamente, las operaciones realizadas por el mismo se restablecerán.

Modifique `myipc.h` y `myipc.c` en este sentido. Recompile.

**7.2) Identificar la sección crítica de código<sup>29</sup> en `tp84p.c` y `tp84c.c`. Copiar `tp84p.c` en `tp85p.c`. Modificar `tp85p.c` para crear una nueva versión del servidor, incorporando semáforos según Stallings y acorde con librería `myipc`:**

El modo de uso de estos semáforos será de lectura y escritura:

```

...
int sem_modo = SEM_R | SEM_A;
...

```

para las distintas claves asociadas con estos semáforos (según Stallings, los semáforos s,n,e):

```

...
int sem_ids[3];      // id de c/u de los semáforos a crear: s,n,e
key_t sem_keys[3];   // claves IPC para luego generar los ids de cada semáforo
char *sem_paths[] = {IPC_PATH,IPC_PATH,IPC_PATH};
char sem_chars[] = {IPC_SKEYs ,IPC_SKEYn ,IPC_SKEYe };
...

printf("main():solicito claves ipc para semaforos\n");
obtener_claves(sem_paths,sem_chars,sem_keys,3);

```

<sup>29</sup> Dícese del conjunto de instrucciones del programa en donde se requiere la aplicación algún mecanismo de exclusión mutua.

```
...
```

Una vez obtenidas las claves, se pueden crear e inicializar los semáforos y por último borrarlos:

```
...
char *sem_error1[] = {"main():Error creando semaforo s",
                      "main():Error creando semaforo n",
                      "main():Error creando semaforo e",};
char *sem_error2[] = {"main():Error inicializando semaforo s",
                      "main():Error inicializando semaforo n",
                      "main():Error inicializando semaforo e",};
char *sem_error3[] = {"main():Error borrando semaforo s",
                      "main():Error borrando semaforo n",
                      "main():Error borrando semaforo e",};
// valores iniciales de los semáforos s,n,e  según Stallings Pag.227
int sem_init[] = {1,0,1};
...
printf("main():creo semaforos\n");
for(i=0;i<3;i++)
    if ( (sem_ids[i] = crear_semaforo(sem_keys[i],sem_modo)) == -1)
        perror(sem_error1[i]);

printf("main():inicializo semaforos\n");
for(i=0;i<3;i++)
    if ( semSignalTo(sem_ids[i],sem_init[i]) == -1)
        perror(sem_error2[i]);
...
// elimino semaforos
printf("main():borro semaforos\n");
for(i=0;i<3;i++)
    if ( borrar_semaforo(sem_ids[i]) == -1)
        perror(sem_error3[i]);
...
```

Habiendo creado los semáforos de esta forma, se pueden incrementar o decrementar sus valores de la siguiente forma:

```
...
semWait(sem_ids[SEM_e]); // operacion semWait(e)
...
semSignal(sem_ids[SEM_n]); // operacion semSignal(n)
...
```

También se puede usar un único set de semáforos o bien tres semáforos independientes (como se optó en este caso). Eliminar la llamada: `sleep(10);` dentro del loop principal del servidor para generar un mayor nivel de concurrencia con `tp85c`.

Podrá obtener una salida como esta:

```
consola server:
debian:/home/grchere/gccwork/src/sem# ./tp85p
main():inicio proceso productor|
```

```

main(): para salir envie señal SIGUSR2 a proceso 2708
main(): solicito clave ipc
main(): creo memoria compartida de 360 bytes
main(): attachar memoria local a memoria compartida 262151
main(): Inicializo memoria compartida a cero
main(): genero total de ventas de sucursales en memoria compartida
main(): solicito claves ipc para semaforos
main(): creo semaforos
main(): inicializo semaforos
main(): Bloqueo proceso hasta recibir señal de salida
mostrar_ventas(): sucursal[0]=136.105426
mostrar_ventas(): sucursal[1]=145.062103
mostrar_ventas(): sucursal[2]=298.346262
mostrar_ventas(): sucursal[3]=58.790672
mostrar_ventas(): sucursal[4]=13.892183
mostrar_ventas(): sucursal[5]=210.941994
mostrar_ventas(): sucursal[6]=184.822537
mostrar_ventas(): sucursal[7]=288.855166
mostrar_ventas(): sucursal[8]=80.767787
mostrar_ventas(): sucursal[9]=330.083290
913/ 5/2009      1983      85.81
213/ 5/2009      323      186.31
829/10/2009      105      38.55
713/ 3/2009      973      140.58
915/ 7/2009      202      208.76
3 1/ 2/2009      1432      58.66
0 4/ 2/2009      2039      24.81
629/11/2009      102      115.21
128/ 2/2009      167      89.86
223/ 6/2009      2123      10.84
...
main(): borro semaforos
main(): Desconecto memoria compartida
main(): Destruyo memoria compartida
main(): fin proceso productor1 retorno=0
debian:/home/grchere/gccwork/src/sem#

```

**7.3)** Idem anterior. Copiar tp84c.c en tp85c.c. Modificar tp85c.c para incorporar semáforos según Stallings y acorde con librería myipc. Modificar tp85c.c para consumir (en este caso, significa mostrar los totales de ventas por sucursal, mostrar las ventas de sucursales, agregar las ventas de las sucursales en el archivo <id proceso>.ventas, y por ultimo, poner en cero las ventas de sucursales) los totales generados por tp85p.c hasta que reciba la señal SIGUSR2 para salir.

Este proceso no destruye ni crea semáforos, sino que obtiene el id de los semáforos creados por el servidor (usando los mismos arreglos que en el caso anterior):

```

...
printf("main(): solicito claves ipc para semaforos\n");
obtener_claves(sem_paths,sem_chars,sem_keys,3);
printf("main(): obtengo semaforos\n");
for(i=0;i<3;i++)
    if ( (sem_ids[i] = obtener_semaforo(sem_keys[i],sem_modo)) == -1)
        perror(sem_error1[i]);

```

```
...
```

El proceso de "consumir", asumiendo que:

- `fpo` es un puntero de tipo `FILE *` que apunta al stream abierto de modo `append`<sup>60</sup> conectado con el archivo `<id proceso>.ventas`
- `pventas` es un puntero de tipo `double *` que apunta al primer elemento del arreglo de totales de ventas por sucursal
- `psucvta` es un puntero de tipo `struct sucvta *` que apunta al primer elemento del arreglo de ventas realizadas en sucursales podría ser el siguiente:

```
...
```

```
mostrar_suventas(fpo,psucvta);
mostrar_ventas(pventas);
mostrar_suventas(stdout,psucvta);
printf("main():pongo a cero memoria consumida\n");
memset(psucvta,0,sizeof(struct sucvta)*10);
...
```

Usar vectores para facilitar la implementación de los 3 semáforos requeridos. También se puede usar un único set de semáforos o bien tres semáforos independientes.

Una vez implementados los puntos 7.1), 7.2) y 7.3) podrá obtener una salida como esta:

```
consola server:
debian:/home/grchere/gccwork/src/sem# ./tp85p
main():inicio proceso productor
main():para salir envie señal SIGUSR2 a proceso 2922
main():solicito clave ipc
main():creo memoria compartida de 360 bytes
main():atricular memoria local a memoria compartida 393223
main():Inicializo memoria compartida a cero
main():genero total de ventas de sucursales en memoria compartida
main():solicito claves ipc para semaforos
main():creo semaforos
main():inicializo semaforos
main():Bloqueo proceso hasta recibir señal de salida
...
mostrar_ventas(): sucursal[0]=160157.865768
mostrar_ventas(): sucursal[1]=151184.964007
mostrar_ventas(): sucursal[2]=155957.583394
mostrar_ventas(): sucursal[3]=150512.363443
mostrar_ventas(): sucursal[4]=149154.134820
mostrar_ventas(): sucursal[5]=148647.715444
mostrar_ventas(): sucursal[6]=148937.294108
mostrar_ventas(): sucursal[7]=134875.590290
mostrar_ventas(): sucursal[8]=149356.049756
mostrar_ventas(): sucursal[9]=151668.660699
 4 7/10/2009    1865    119,61
 727/ 9/2009    1359    149,10
```

<sup>60</sup> El segundo argumento de la función `fopen()` se refiere al modo de apertura del stream o flujo; el modo `append` implica abrir el flujo para agregar datos al mismo, grabar a partir del final del archivo.

```

917/ 9/2009    1053    160.33
918/ 4/2009    808     178.43
2 7/11/2009    970     108.02
3 7/11/2009    2000    211.54
012/11/2009    806     130.30
320/ 4/2009    2087    93.76
026/ 4/2009    845     53.79
321/ 3/2009    1445    94.11
...
main():borro semaforos
main():Desconecto memoria compartida
main():Destruyo memoria compartida
main():fin proceso productor1 retorno=0
debian:/home/grchere/gccwork/src/sem#
```

---

```

consola cliente:
...
main():pongo a cero memoria consumida
main():leo memoria compartida
mostrar_ventas(): sucursal[0]=160157.865768
mostrar_ventas(): sucursal[1]=151184.964007
mostrar_ventas(): sucursal[2]=155957.583394
mostrar_ventas(): sucursal[3]=150512.363443
mostrar_ventas(): sucursal[4]=149154.134820
mostrar_ventas(): sucursal[5]=148647.715444
mostrar_ventas(): sucursal[6]=148937.294108
mostrar_ventas(): sucursal[7]=134875.590290
mostrar_ventas(): sucursal[8]=149356.049756
mostrar_ventas(): sucursal[9]=151668.660699
4 7/10/2009    1865    119.61
727/ 9/2009    1359    149.10
917/ 9/2009    1053    160.33
918/ 4/2009    808     178.43
2 7/11/2009    970     108.02
3 7/11/2009    2000    211.54
012/11/2009    806     130.30
320/ 4/2009    2087    93.76
026/ 4/2009    845     53.79
321/ 3/2009    1445    94.11
main():pongo a cero memoria consumida
...
main():recibi señal de salida
main():pongo a cero memoria consumida
main():Desconecto memoria compartida
main():fin proceso consumidor1 retorno=0
```

---

```

consola otra consola, durante el proceso:
debian:/home/grchere/gccwork/src/sem# ipcs
```

----- Shared Memory Segments -----						
key	shmid	owner	perms	bytes	nattch	status
0x00000000	32768	grchere	600	196608	2	dest
0x00000000	65537	grchere	600	196608	2	dest
0x00000000	98306	grchere	600	393216	2	dest
0x00000000	131075	grchere	600	393216	2	dest
0x00000000	163844	grchere	600	12288	2	dest

```

0x00000000 196613 grchere 600 393216 2 dest
0x00000000 229382 grchere 600 196608 2 dest
0x68022a15 327687 root 600 360 1

----- Semaphore Arrays -----
key semid owner perms nsems
0x73022a15 196608 root 666 1
0x6e022a15 229377 root 666 1
0x65022a15 262146 root 666 1

----- Message Queues -----
key msqid owner perms used-bytes messages
debian:/home/grchere/gccwork/src/sem#


al terminar proceso no quedan semaforos asignados:

debian:/home/grchere/gccwork/src/sem# ipcs

----- Shared Memory Segments -----
key shmid owner perms bytes nattch status
0x00000000 32768 grchere 600 196608 2 dest
0x00000000 65537 grchere 600 196608 2 dest
0x00000000 98306 grchere 600 393216 2 dest
0x00000000 131075 grchere 600 393216 2 dest
0x00000000 163844 grchere 600 12288 2 dest
0x00000000 196613 grchere 600 393216 2 dest
0x00000000 229382 grchere 600 196608 2 dest

----- Semaphore Arrays -----
key semid owner perms nsems

----- Message Queues -----
key msqid owner perms used-bytes messages
debian:/home/grchere/gccwork/src/sem#

```

Obsérvese que el productor queda bloqueado mientras está imposibilitado de producir, cuando ya no hay ningún consumidor en ejecución.

## Capítulo IX. Entrada - Salida (Input - Output)

**Objetivo:** Desarrollar un programa que permita operar sobre dispositivos de entrada - salida del computador. Interactuar con el programa a través de la línea de comandos. Establecer distintos parámetros de comunicación.

### Introducción

En un entorno UNIX los dispositivos están representados a través de archivos especiales ubicados en el sistema de archivos, al igual que cualquier otro archivo regular del usuario, esto permite un tratamiento uniforme, a través de una misma interfaz.

Los archivos especiales asociados a dispositivos se encuentran en `/dev`.

Los puertos seriales están representados por el archivo `ttys0` (puerto serial 1), `ttys1` (puerto serial 2) y así sucesivamente.

Los puertos usb están representados por el archivo `ttyUSB0` (puerto usb 1), `ttyUSB1` (puerto usb 2) y así sucesivamente.

Los puertos paralelos están representados por el archivo `lp0` (puerto paralelo 1), `lp1` (puerto paralelo 2), etc.; también suele existir el link<sup>31</sup> `/dev/printer`.

Por lo tanto, enviar un carácter a través del puerto serial 1 del computador implica grabar dicho carácter en el archivo `/dev/ttys0`. Idem para la lectura, habrá que leer un carácter de dicho archivo.

La configuración de los parámetros de comunicación se realiza a través de una terminal (usando la función `tcsetattr()`), una vez que la terminal ha sido configurada, luego se asocia dicha terminal con el dispositivo a utilizar. Generalmente antes de cambiar la configuración de una terminal se guarda su configuración (usando la función `tcgetattr()`) actual para luego poder restablecerla<sup>32</sup>.

---

<sup>31</sup> Algo similar a los "Accesos Directos" (shortcuts) de Windows, es un fichero que apunta a otro fichero.

<sup>32</sup> En pruebas realizadas entre Linux y Windows, transfiriendo a través del puerto serie se pudo comprobar que al intentar restablecer la configuración original de la terminal (en el caso del programa de envío) ello provocaba la pérdida de datos transmitidos por parte del programa de recepción. Esto se pudo notar en archivos grandes y con equipos lentos. También las pruebas denotan una gran lentitud en la recepción/transmisión de datos a través del puerto serie en Windows, utilizando este mismo código recompilado con cygwin, posiblemente esto se deba a la intervención de cygwin1.dll que convierte las llamadas Unix en llamadas Windows. Debido a esto se incorporó el parámetro de delay para mejorar la sincronización de equipos con velocidades muy dispares que puedan afectar a la comunicación (a pesar de haberse configurado el puerto a la misma velocidad).

El Analista Y debe interactuar con varios dispositivos conectados a distintos puertos de los distintos computadores que tiene en cada una de las sucursales de la Empresa X. Por ejemplo, puede tratarse de puntos de venta que poseen impresoras fiscales conectadas al puerto paralelo del computador. En otra sucursal tiene un reloj de control de personal conectado a una workstation por puerto serial. Otro sistema tiene una interfaz que interactúa a través de un puerto USB, etc.

La interacción con los dispositivos es muy variada y básicamente se trata del envío y recepción de caracteres a través de un puerto. En algunos casos, se trata de caracteres especiales que son interpretados de alguna forma por el receptor; en otros casos, son simples datos a enviar o recibir.

Otra variante son los parámetros de la comunicación: varían en cuanto a la cantidad de bits de datos, de paridad, en cuanto a la velocidad de comunicación con el dispositivo, en cuanto a los bits de stop y en cuanto al time-out de la comunicación.

Para solucionar estos problemas, el Analista Y ha pensado en el desarrollo de dos programas (uno para el envío y otro para la recepción) que interactúen con los puertos; los programas deberán:

- ◆ Recibir los siguientes parámetros por línea de comandos:
  - archivo (path) a enviar o recibir
  - nombre del dispositivo (/dev/...)
  - velocidad (número de baudios, ejemplo: 1200, 2400, ...)
  - time-out (número de segundos de time-out antes de dar por fallido el envío o recepción, ejemplo: 0, 1, 2, ...)
  - paridad (0 indica que no hay paridad, 1 indica paridad impar, 2 indica paridad par)
  - bits de datos por byte (5,6,7,8)
  - bits de stop por byte (1,2)
  - segundos de demora (segundos de demora luego de un envío o recepción, esto puede ser útil para sincronizar las distintas velocidades entre dispositivos, ejemplo: 0 (indica que no hay demora),1 (demora de un 1 segundo),...)
- ◆ Configurar el puerto utilizando los parámetros indicados
- ◆ Realizar el envío o recepción del archivo indicado

La descripción anterior coincide con el siguiente diseño:

### Programa de envío:

```
main()
    controlar parametros
    si parametros Ok entonces
        si pudo abrir dispositivo en modo grabacion entonces
            si pudo abrir archivo a enviar en modo lectura entonces
                guardar estado actual del dispositivo
                setear modo de control del dispositivo
                setear paridad
                setear bits de datos por byte
                setear modo de transmision
                setear time-out
                setear velocidad
                setear bits de stop por byte
                asignar al puerto los nuevos parametros
                bytes enviados = 0
                mientras haya podido leer un bloque de datos de archivo a enviar
                    grabo en dispositivo los bytes leidos de archivo a enviar
                    bytes enviados = bytes enviados + bytes leidos
                    demoro <segundos de demora>
                fin mientras
                cierro dispositivo
                cierro archivo a enviar
                muestro la cantidad de bytes enviados
            sino
                error en apertura de archivo a enviar
            fin si
        sino
            error en apertura de dispositivo
        fin si
    sino
        error en parametros, mostrar forma de uso
    fin si
fin main()
```

### Programa de recepción (muy similar al anterior):

```
main()
    controlar parametros
    si parametros Ok entonces
        si pudo abrir dispositivo en modo lectura entonces
            si pudo abrir archivo a recibir en modo grabacion entonces
                guardar estado actual del dispositivo
                setear modo de control del dispositivo
                setear paridad
                setear bits de datos por byte
                setear modo de transmision
                setear time-out
                setear velocidad
                setear bits de stop por byte
                asignar al puerto los nuevos parametros
                bytes recibidos = 0
                mientras haya podido leer un bloque de datos del dispositivo
                    grabo en archivo a recibir los bytes leidos del dispositivo
                    bytes recibidos = bytes recibidos + bytes grabados
                    demoro <segundos de demora>
                fin mientras
```

```

        cierro dispositivo
        cierro archivo a recibir
        muestro la cantidad de bytes recibidos
    sino
        error en apertura de archivo a recibir
    fin si
    sino
        error en apertura de dispositivo
    fin si
    sino
        error en parametros, mostrar forma de uso
    fin si
fin main()

```

1. Implemente la librería `serial.c`. Evidentemente, ambos diseños tienen mucho en común, por lo tanto, sería buena idea "poner en un solo lugar todo el código a reusar", para ello, construiremos una librería formada por su archivo de código `serial.c` y su API `serial.h`. Todos los parámetros de comunicación se pueden especificar a través de la interfase de terminal de bajo nivel (ver capítulo 17 "Low-Level Terminal Interface" [LooseSandra, Loosemore, et. al.], también se pueden consultar [Frerking, Gary] y [Lawyer, David S.]), para ello se utiliza la estructura `struct termios`. Cambiar un parámetro implica modificar el contenido de algún elemento dentro de dicha estructura. Si el programa principal tendrá una referencia (puntero) a dicha estructura, la misma deberá ser pasada a cada una de las funciones de esta librería. Una API posible de esta librería podría ser:

```

void setParidad(struct termios *t,int par);
void setDataBits(struct termios *t,int bits);
void setTimeOut(struct termios *t,int sec);
void setVel(struct termios *t,char *velocidad);
void setStopBits(struct termios *t,int bits);

```

Dentro de la estructura `termios` existen una serie de atributos que están formados por un conjunto de bits que representan determinadas características de la comunicación, el más importante de ellos es el atributo `c_cflag` que será utilizado para indicar el control de paridad, bits de datos, bits de stop y velocidad. ¿Cómo es posible indicar tantas cosas distintas en una única variable? A través de la activación y desactivación de determinados bits dentro del conjunto de bits que representa `c_cflag`. Por lo tanto, se requerirá del uso de operadores de manejo de bits. Tomemos como ejemplo el seteo de la paridad de la comunicación, los posibles valores del argumento `par` son:

Valor Argumento <code>par</code>	Significado
0	implica sin control de paridad
1	implica paridad impar
2	implica paridad par

Una forma de uso posible de esta función sería:

```

struct termios newtio;
int paridad;

```

```

...
// se asignan valores a newtio y paridad
...
setParidad(&newtio, paridad);
...

```

Para facilitar la activación/desactivación de bits, se pueden utilizar los operadores:

Operador	Significado
	OR binario
&	AND binario
~	Negación/inversión binaria (donde hay un bit activado lo desactiva y viceversa)

Para facilitar la ubicación del conjunto de bits a cambiar, C nos provee de constantes definidas en `termios.h`<sup>33</sup> que tienen algún significado lógico, a continuación se resumen algunas de las utilizadas en este proyecto<sup>34</sup>:

Constante	Significado	Aplicable sobre Atributo (de estructura <code>termios</code> ) / usado con función:
PARENB	Habilitar control de paridad	<code>c_cflag</code>
PARODD	Habilitar paridad impar	<code>c_cflag</code>
B0, B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400, B4800, B9600, B19200, B38400, B57600, B115200, B230400 ... <sup>35</sup>	Velocidad en bps, 0 baudios (colgado), 50 baudios, 75, 110, ...	<code>c_cflag</code>
CSIZE	Máscara que representa todas las combinaciones posibles de bits de datos (5,6,7,8)	<code>c_cflag</code>
CS5, CS6, CS7, CS8	Habilitar 5,6,7 u 8 bits de datos	<code>c_cflag</code>
CSTOPB	Habilitar 2 bits de stop (caso contrario, se asume 1 bit de stop)	<code>c_cflag</code>
CLOCAL	Indica que la terminal está conectada localmente (ignora	<code>c_cflag</code>

<sup>33</sup> En plataformas Windows utilizando cygwin se encuentran definidas dentro del archivo `\usr\include\termios.h` . En Linux `/usr/include/termios.h` lo redirecciona a `/usr/include/bits/termios.h` .

<sup>34</sup> Para mayor información, favor de consultar el Capítulo 17 "Low-Level Terminal Interface" del "The GNU C Library Reference Manual".

<sup>35</sup> Las velocidades varían según la plataforma, por ejemplo, en Windows utilizando cygwin se cuenta con B128000, B256000; mientras que en Linux éstas no existen. En Linux se cuenta con B460800, B500000, B576000, B921600, B1000000 ; mientras que en Windows éstas no existen.

	la linea de estado del modem)	
CREAD	Habilita el hecho de que la entrada pueda ser leída desde la terminal, caso contrario, la entrada será descartada si bien llegue.	c_cflag
ICANON	Habilita modo canónico de comunicación <sup>36</sup> (caso contrario se asumirá modo no canónico)	c_lflag
ECHO	Habilita el echo de los caracteres recibidos en la terminal	c_lflag
INPCK	Habilita el chequeo de paridad (caso contrario, no habrá control de paridad)	c_iflag (para input) c_oflag (para output)
PARMRK	Los bytes con error de paridad serán marcados (se usa en combinación con INPCK) agregando previamente dos bytes adicionales (0377 y 0)	c_iflag (para input) c_oflag (para output)
ISTRIP	Los bytes recibidos correctamente serán representados con 7 bits (caso contrario, se usarán 8 bits).	c_iflag (para input) c_oflag (para output)
TCIFLUSH	Limpia la cola de input asociada con la terminal.	tcflush()
TCOFLUSH	Limpia la cola de output asociada con la terminal.	tcflush()
TCIOFLUSH	Limpia la cola de input y output asociada con la terminal.	tcflush()
TCSANOW	Setea los atributos de la terminal ahora, inmediatamente (otras constantes pueden indicar que esto se haga en forma diferida).	tcsetattr()

No debemos olvidar que atributos tales como `c_iflag`, `c_oflag`, `c_cflag`, `c_lflag`, etc. tienen un conjunto de bits activados/desactivados y que, para cambiar algún aspecto de la comunicación debemos ser cuidadosos de sólo afectar al conjunto

<sup>36</sup> Los sistemas POSIX soportan dos modos de entrada: canónico o no canónico. El modo de procesamiento canónico implica que la terminal interprete determinados caracteres de control y actúe en consecuencia (por ejemplo, no se leerá el input de la terminal hasta que el usuario presione la tecla de Intro). En modo no canónico, los caracteres no se agrupan en líneas, son un simple flujo de bytes, sobre el cual no se realiza ningún tipo de interpretación.

de bits adecuado. Tomemos como ejemplo a la función `setParidad()`, esta función primero deberá determinar si hay control de paridad (ver tabla argumento `par`):

```
void setParidad(struct termios *t, int paridad) {
    // seteo paridad
    if (!paridad) { // deshabilito control de paridad
        t->c_cflag &= ~PARENB;
    } else { // habilito control de paridad
        t->c_cflag |= PARENB;
    }
    ...
}
```

El uso de un operador de la forma: `<variable> <operador> = <expresión>`; es equivalente a `<variable> = <variable> <operador> <expresión>`. Por lo tanto, `a+=b` equivale a: `a=a+b`. La expresión `t->c_cflag &= ~PARENB` equivale a `t->c_cflag = t->c_cflag & (~PARENB)`.

¿Por qué debo hacer: `t->c_cflag &= ~PARENB;` para deshabilitar el control de paridad? ¿Por qué no hacer otra instrucción? Investiguemos. Observemos el contenido de `termios.h`:

```
...
#define NCCS 18
...
#define PARENB 0x00100
...
typedef unsigned char cc_t;
typedef unsigned int tcflag_t;
typedef unsigned int speed_t;
typedef unsigned char ospeed_t;
...
struct termios
{
    tcflag_t c_iflag;
    tcflag_t c_oflag;
    tcflag_t c_cflag;
    tcflag_t c_lflag;
    char c_line;
    cc_t c_cc[NCCS];
    ospeed_t c_ispeed;
    ospeed_t c_ospeed;
}
...
```

El valor definido para `PARENB` es `0x00100` (hexadecimal), en decimal equivale a 256, en binario `100000000` (el noveno bit activado, el resto desactivado); es decir, que activar el control de paridad equivale a activar el noveno bit dentro de `c_cflag`. El valor `0x00100` es de tipo entero (`int`) al igual que `tcflag_t`, por lo tanto, `PARENB` y `c_cflag` son tipos compatibles.

La operación a realizar debería ser tal, que solo se afectara al noveno bit, dejando todo lo demás igual que antes dentro `c_cflag`. Supongamos que `c_cflag` vale 4 (100 en binario), hagamos las cuentas:

PARENB	100000000
<code>~</code> PARENB	011111111
<code>c_cflag</code>	000000100
<code>c_cflag</code>	000000100
<code>&amp;</code>	<code>~</code> PARENB
	011111111
	-----
	000000100

Podemos observar que el cálculo permite desactivar el noveno bit de `c_cflag` sin alterar el resto de su contenido (100), pero claro, antes de comenzar, el noveno bit de `c_cflag` ya estaba desactivado. Probemos ahora con el noveno bit activado en `c_cflag`:

PARENB	100000000
<code>~</code> PARENB	011111111
<code>c_cflag</code>	100000100
<code>c_cflag</code>	100000100
<code>&amp;</code>	<code>~</code> PARENB
	011111111
	-----
	000000100

El resultado es el mismo, esto prueba que estando o no activado previamente el control de paridad, la sentencia `t->c_cflag &= ~PARENB;` lo desactiva, sin alterar el resto de los bits de `c_cflag`.

El mismo análisis puede hacerse para la activación del bit de control de paridad: `t->c_cflag |= PARENB;` . Comencemos con `c_cflag` valiendo 4 (100):

PARENB	100000000
<code>c_cflag</code>	000000100
<code>c_cflag</code>	000000100
<code> </code>	PARENB
	100000000
	-----
	100000100

Podemos comprobar que activa el noveno bit sin alterar el resto. Ahora probemos con el noveno bit activado en `c_cflag`:

```

PARENB 100000000
c_cflag 100000100

c_cflag 100000100
| PARENB 100000000
-----
100000100

```

El noveno bit de `c_cflag` continua activado y el resto de los bits no se han cambiado.

Ahora Ud. ya puede comprender las operaciones binarias básicas que requiere este programa, veamos el código completo de `setParidad()`:

```

void setParidad(struct termios *t,int paridad) {
    // seteo paridad
    if (!paridad) {
        //PARENB significa habilitar bit de paridad
        //entonces, esto deshabilita el bit de paridad:
        t->c_cflag &= ~PARENB;
    } else { // habilito paridad
        t->c_cflag |= PARENB;
    }
    if (paridad == 1) { // paridad impar
        //PARODD habilita paridad impar
        t->c_cflag |= PARODD;
    } else { // paridad par, implica no paridad impar
        t->c_cflag &= ~PARODD;
    }
}

```

Otras funciones más que son parte de la biblioteca `serial.c`:

```

...
// <stop bits>=1,2 (nro. de bits de stop)
void setStopBits(struct termios *t,int stopbits) {
    //CSTOPB indica 2 bits de stop, caso contrario es solo 1 bit de stop
    if (stopbits == 2) t->c_cflag |= CSTOPB;
    else t->c_cflag &= ~CSTOPB;
}

// bits=5,6,7,8 (nro. de bits por byte)
void setDataBits(struct termios *t,int bits) {
    //CSIZE es una mascara para todos los tamaños de datos en bits,
    //entonces, haciendo un and con su negacion, elimina el
    //seteo actual en cuanto al tamaño de datos en bits
    t->c_cflag &= ~CSIZE;
    switch(bits) {
        case 5:
            t->c_cflag |= CS5;
            break;
        case 6:
            t->c_cflag |= CS6;
            break;
    }
}

```

```

        case 7:
            t->c_cflag |= CS7;
            break;
        case 8:
            t->c_cflag |= CS8;
            break;
        default:
            t->c_cflag |= CS8;
    }
}

// nro.de segundos de time out en input/output (0 indica no time out)
void setTimeOut(struct termios *t,int sec) {
    t->c_cc[VTIME] = sec*10; /* inter-character timer unused */
}
...

```

2. Continuamos ampliando la librería serial.c. Falta implementar `setVel()` para establecer la velocidad de la comunicación. Obsérvese que el argumento de la velocidad es una cadena de caracteres pero las constantes definidas en `termios.h` son de tipo `int`. El usuario ingresará en la linea de comandos `"2400"` y ello internamente deberá transformarse en `B2400` (valor `int` definido dentro de `termios.h`). Para resolver esto, podemos hacer un "traductor" que pase de una cadena a su valor entero correspondiente, para ello podemos definir dentro de `serial.h` un nuevo tipo de dato que ayudará a la conversión:

```

...
typedef struct velocidad {
    char *nombre;
    int valor;
} velocidad;
...

```

Dentro de `serial.c` podemos crear un arreglo global o estático a esta librería de tipo `velocidad` en donde se indiquen las velocidades soportadas por esta librería:

```

...
// dato global de biblioteca
velocidad vel[] = {
    {"0",B0}, {"50",B50}, {"75",B75}, {"110",B110}, {"134",B134}, {"150",B150},
    {"200",B200}, {"300",B300}, {"600",B600}, {"1200",B1200}, {"1800",B1800},
    {"2400",B2400}, {"4800",B4800}, {"9600",B9600}, {"19200",B19200},
    {"38400",B38400}, {"57600",B57600}, {"115200",B115200}, {"230400",B230400},
    {'\0',0}
};
...

```

Cada pareja de valores dentro del arreglo de tipo `velocidad` se corresponden con los valores de `nombre` y `valor`. Por ejemplo, `{"0",B0}` indica que `vel[0].nombre` apunta a `"0"` y `vel[0].valor` vale `B0` (que en `termios.h` está definido como un entero que vale `0x00000`) y así sucesivamente. La última pareja de valores `{ '\0', 0 }` asignará un valor nulo (`NULL`, `'\0'`, valor decimal `0`) en `nombre` para indicar el

final de las velocidades posibles. Usando este arreglo se puede implementar una función que haga una búsqueda secuencial dentro del mismo -por ejemplo- a partir de "2400" y retorne la posición (subíndice) en que se encuentra dentro del arreglo `vel[]` (se implementa usando punteros):

```
...
// busca strVel en tabla de velocidades,
//   si no esta devuelve -1
//   si esta devuelve su posicion
int findVel(char *strVel) {
    velocidad *v = &vel[0];
    while(v->nombre) {
        if (strcmp(v->nombre,strVel) == 0) return v - &vel[0];
        v++;
    }
    return -1;
}
...
```

La función `findVel()` será llamada desde la función `setVel()`.

Para indicar la velocidad de comunicación se puede usar también el flag `C_CFLAG` y/o las funciones `cfsetispeed()`, `cfsetospeed()` (para las velocidades de input y output respectivamente):

```
...
// <velocidad>=<baudios>
void setVel(struct termios *t,char *velocidad) {
    int vpos = findVel(velocidad);
    int ret;
    if ( vpos != -1) {
        t->c_cflag |= vel[vpos].valor;
        ret=cfsetispeed(t, vel[vpos].valor);
        ret=cfsetospeed(t, vel[vpos].valor);
    } else printf("ts baudios no es una velocidad correcta\n",velocidad);
}
...
```

3. Ya tendríamos una biblioteca básica para comunicaciones. Ahora vamos a comenzar por el programa de envío (`sendserial.c`), cuyo diseño se indicó al comienzo. El usuario va a interactuar con el programa a través de la línea de comandos, es decir, que el programa podría ejecutarse de la siguiente forma en Linux utilizando un shell script:

```
#!/bin/bash
#envio desde linux debian 5.01
./sendserial serial.c /dev/ttyS0 9600 5 1 7 1 2
```

Algo similar sería en Windows, utilizando un archivo `.bat`<sup>37</sup>:

```
echo off
rem Envio desde windows
```

<sup>37</sup> Para un programa compilado con cygwin y que éste tenga acceso a `cygwin1.dll`.

```
sendserial.exe serial.c /dev/ttyS0 9600 10 1 7 1 0
```

Los argumentos de la línea de comandos se representan y acceden en C a través de los argumentos declarados en la función main(). Por ejemplo, si se declara la función main() como:

```
...
int main(int argc,char *argv[]) {
    ...
}
```

Para los ejemplos de ejecución anterior, el valor de argc será de 9 y el contenido apuntado por argv[] serían las siguientes cadenas de caracteres (para el caso de Linux):

Posición argv[]	Contenido argv[]	Significado
argv[0]	sendserial	Nombre del Programa
argv[1]	serial.c	Archivo a enviar/recibir
argv[2]	/dev/ttyS0	Dispositivo a utilizar en envío/recepción
argv[3]	9600	Velocidad
argv[4]	5	segundos de time-out en envío/recepción (0..n)
argv[5]	1	Paridad (0,1,2)
argv[6]	7	Bits de datos (5,6,7,8)
argv[7]	1	Bits de stop (1,2)
argv[8]	2	Segundos de demora entre cada envío/recepción (0..n)

En cuanto al diseño del programa "controlar parámetros" se refiere a controlar los valores posibles de argc y cada uno de los elementos de argv[].

Hasta incluso, si el usuario no indica ningún argumento (argc == 1) o el número de argumentos fuese distinto de 9, se podría mostrar una ayuda en cuanto a los argumentos soportados.

"Abrir dispositivo en modo grabación", implica declarar e instanciar un descriptor de archivo (file descriptor, valor entero que indica el número de archivo abierto por este proceso):

```
...
int fd = open(argv[2], O_WRONLY | O_NOCTTY );
if (fd <0) {
    fprintf(stderr,"Error abriendo dispositivo [%s]\n",argv[2]);
    exit(-1);
} else { // pude abrir dispositivo Ok
    ...
}
```

Para abrirlo en modo lectura, indicar flags: `O_RDONLY | O_NOCTTY`

Para "abrir archivo a enviar" (en este caso, apertura de archivo como stream buffered a ser enviado):

```
...
FILE *fpi = fopen(argv[1], "rb");
if (fpi == NULL) {
    perror(argv[1]);
    exit(-2);
}
...
```

Para abrirlo en modo escritura, el segundo argumento de `fopen()` podría ser "`wb`" (write binary).

Para "guardar el estado actual del dispositivo":

```
...
struct termios oldtio;
...
// guardar estado actual del puerto (fd previamente abierto)
tcgetattr(fd, &oldtio);
...
```

Para "setear el modo de control del dispositivo":

```
...
struct termios oldtio;
...
// guardar estado actual del puerto (fd previamente abierto)
tcgetattr(fd, &oldtio);
...
```

Podemos usar a `newtio` para cargar allí todos los nuevos seteos y luego configurar el puerto con esta nueva configuración:

```
...
struct termios newtio;
...
// guardar estado actual del puerto (fd previamente abierto)
tcgetattr(fd, &newtio);
...
```

Para "setear el modo de control del dispositivo":

```
...
newtio.c_cflag = CLOCAL | CREAD;
...
```

Para "setear paridad":

```
...  
    // seteo paridad  
    setParidad(&newtio, atoi(argv[5]));  
...
```

Para "setear bits de datos por byte":

```
...  
    // bits por byte  
    setDataBits(&newtio, atoi(argv[6]));  
...
```

Para "setear modo de transmision" (no canónico, no echo) en el caso de envío:

```
...  
    // set input mode (non-canonical, no echo,...)  
    newtio.c_oflag = 0;  
    newtio.c_lflag &= ~(ICANON|ECHO);  
...
```

Idem en el caso de recepción:

```
...  
    /* set input mode (non-canonical, no echo,...) */  
    newtio.c_iflag = INPCK | PARMRK | ISTRIP;  
    newtio.c_lflag &= ~(ICANON|ECHO);  
...
```

Para "setear time-out":

```
...  
    // set time out  
    setTimeOut(&newtio, atoi(argv[4]));  
...
```

Para "setear velocidad":

```
...  
    // seteo velocidad  
    setVel(&newtio, argv[3]);  
...
```

Para "setear bits de stop por byte":

```
...  
    // seteo bits de stop  
    setStopBits(&newtio, atoi(argv[7]));  
...
```

Para "asignar al puerto los nuevos parámetros":

```
...  
    // paso los nuevos parametros al file descriptor del puerto
```

```
tcflush(fd, TCOFLUSH);
tcsetattr(fd, TCSANOW, &newtio);
...
```

El loop principal del programa de envío:

```
...
char buf[255];
size_t nread=0;
...
// demora
int sdelay = atoi(argv[8]);
if ( sdelay < 0 ) sdelay =0;

// leo archivo y envío a puerto serie
while((nread=fread(buf,1,255,fpi))) {
    nwrite+=write(fd,buf,nread);
    sleep(sdelay);
}
tcflush(fd, TCOFLUSH);
...
```

Para "cerrar dispositivo y archivo":

```
...
// cierro archivo y puerto
close(fd);
fclose(fpi);
...
```

Los archivos de cabecera a utilizar serían:

```
...
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include "serial.h"
...
```

Compile el programa como: `gcc -Wall -o senserial sendserial.c serial.c`

4. Idem anterior, pero ahora desarrolle el programa de recepción (`receiveserial.c`).

Compile el programa como: `gcc -Wall -o receiveserial receiveserial.c serial.c`

5. Si cuenta con un cable serial como el utilizado para las terminales seriales Unix<sup>38</sup>, puede utilizar estos programas para enviar y recibir archivos a través del puerto serial. Si recompila los programas en Windows utilizando cygwin, podría hacerlo entre equipos con distintos Sistemas Operativos<sup>39</sup>.

---

<sup>38</sup> Si desea armar su propio cable, puede consultar el documento Serial-HOWTO que encontrará en <http://tldp.org/HOWTO/Serial-HOWTO.html>, ver sección 19.1 "Pinout of 9-pin and 25-pin serial connectors".

<sup>39</sup> Esta experiencia ya fue probada con este código, compilando ambos programas con cygwin en Windows y recibiendo/transfiriendo archivos entre computadoras con Windows XP Professional Edition y Linux Debian 5.01.

## **Palabras Finales**

Habiendo concluido esta ejercitación, Ud. está en condiciones de utilizar -a través de sus programas- los distintos recursos que nos ofrece un sistema operativo maduro que le serán de gran ayuda en la solución de problemas habituales. Si bien todo el libro está orientado al Sistema Operativo Unix, en el mundo Windows también existen las tuberías, los semáforos, etc.; los conceptos y sus aplicaciones se repiten, solo cambia la API a utilizar desde sus programas.

Queda mucho camino por recorrer: el sistema administrador de archivos permite implementar distintos tipos de archivos que permitirán la solución de distintos tipos de problemas; el sistema administrador de memoria también permitirá hacer otros trucos muy útiles. La comunicación y sincronización de procesos que hemos visto está supeditada a un único computador; aún queda por explorar todo lo relacionado con los sistemas distribuidos en donde podemos realizar interacciones entre procesos que se ejecutan sobre distintos computadores. Esta última área ha cobrado gran importancia en los últimos años y permite el desarrollo de programas servidores y clientes bajo una arquitectura cliente - servidor (client-server) o de n capas (n-tier architecture). Las próximas publicaciones pretenderán cubrir el temario que aun nos queda pendiente.

## Bibliografía

- ◆ Barney, Blaise, "Introduction to Parallel Computing", Lawrence Livermore National Laboratory, On-line: [https://computing.llnl.gov/tutorials/parallel\\_comp/](https://computing.llnl.gov/tutorials/parallel_comp/)
- ◆ Barney, Blaise, "POSIX Threads Programming", Lawrence Livermore National Laboratory, On-line: <https://computing.llnl.gov/tutorials/pthreads/>
- ◆ Cherencio, Guillermo, "Administración de Memoria y Apuntadores en ANSI C", On-line: <http://www.grch.com.ar/docs/p1/ProgramacionI-2008-apuntadores.pdf>
- ◆ Cherencio, Guillermo, "Cuestiones Sintácticas básicas del Lenguaje C", On-line: <http://www.grch.com.ar/docs/p1/ProgramacionI-2008-sintaxis.pdf>
- ◆ Cherencio, Guillermo, "Trabajando con el Compilador GNU C/C++ GCC", On-line: <http://www.grch.com.ar/docs/p1/ProgramacionI-2008-gcc.pdf>
- ◆ Frerking, Gary, "Serial Programming HOWTO", revisión 1.01, Agosto 2001, The Linux Documentation Project (TLPD), On-line: <http://tldp.org/HOWTO/Serial-Programming-HOWTO/index.html>
- ◆ Kernighan, Brian W.; Ritchie, Dennis M., "El lenguaje de programación C" Segunda Edición, Prentice-Hall Hispanoamericana S.A., AT & T Bell Laboratories, Murray Hill, New Jersey, 1991, ISBN-968-880-205-0 (inglés), ISBN 0-13-110362-8 (español)
- ◆ Lawyer, David S., "Serial HOW TO", versión 2.26, Noviembre 2010, The Linux Documentation Project (TLPD), On-line: <http://tldp.org/HOWTO/Serial-HOWTO.html>
- ◆ LooseSandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram, Ulrich Drepper, "The GNU C Library Reference Manual", Free Software Foundation, 2007, Boston, USA. On-line: [http://www.gnu.org/software/libc/manual/html\\_node/index.html](http://www.gnu.org/software/libc/manual/html_node/index.html)
- ◆ Neetzel, Carlos, "Recopilación de Ejercicios de Sistemas Operativos", Editorial CEIT, 2009
- ◆ Silberschatz, Galvin, Gagne, "Operating Systems Concepts", 8th edition, John Wiley & Sons, Inc., 2008, ISBN 0-470-12872-0
- ◆ Stallings, William, "Sistemas Operativos", 5ta. edición, Pearson Educación, Madrid, 2005, ISBN: 978-84-205-4462-5
- ◆ Stevens, Richard, "Advanced Programming in the UNIX Environment", Addison-Wesley Professional Computing Series, 1993, ISBN 0-201-56317-7
- ◆ Tanenbaum, Andrew, "Sistemas Operativos Modernos", 3ra. Edición, Pearson Educación, Mexico, 2009, ISBN: 978-607-442-046-3
- ◆ Wheeler, David A., "Program Library HOW TO", versión 1.20, Abril 2003, The Linux Documentation Project (TLPD), On-line: <http://tldp.org/HOWTO/Program-Library-HOWTO/>