

Universidad del Valle De Guatemala

Facultad de Ingeniería

Sistemas Operativos



Lab 2

Javier Mombiola

Carne: 20067

Sección: 21

Guatemala, 8 de febrero 2023

Ejercicio 1

```
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ gcc -o 1a ejercicio1A.c
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ ./1a

Process

ProcessProcess
Process
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ Process

ProcessProcess

ProcessProcess
Process
Process
Process
Process
Process
Process
Process
Process

javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ gcc -o 1b ejercicio1B.c
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ ./1b

Process
Process
Process
Processjavimombiela@javimombiela-VirtualBox:~/Documents/Lab2$
Process
Process
Process
Process

ProcessProcess
Process
Process
Process
Process
Process
Process
Process
```

¿Cuántos procesos se crean en cada uno de los programas?

- En el primer programa, se puede observar que se crearon 16 procesos.
- En el segundo programa, también se puede observar que se crearon 16 procesos.

¿Por qué hay tantos procesos en ambos programas cuando uno tiene cuatro llamadas fork() y el otro solo tiene una?

- Podemos observar que en ambos programas hay 16 procesos en total, a pesar de que en el primer programa hay 4 fork() y en el segundo solo hay un fork(). Debido a que en el segundo programa, estamos llamando al fork() adentro del for loop con 4 iteraciones, esto crea 4 fork() al igual que el primer programa. Son 16 procesos, debido a que cada fork() crea 2 procesos, por lo tanto, el número de procesos totales serán 2 elevado a la n. En este caso n es igual a 4 por lo cual 2^4 es igual a 16.

Ejercicio 2

Ejecute su programa varias veces (tres o cinco veces suele exhibir el comportamiento deseado) y apunte los resultados de cada vez.

Programa 1

```
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ ./2a
Tiempo total: 0.004999 segundos
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ ./2a
Tiempo total: 0.004624 segundos
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ ./2a
Tiempo total: 0.005491 segundos
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ ./2a
Tiempo total: 0.004499 segundos
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ ./2a
Tiempo total: 0.006733 segundos
```

Programa 2

```
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ ./2b
Tiempo total: 0.000048 segundos.
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ ./2b
Tiempo total: 0.000081 segundos.
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ ./2b
Tiempo total: 0.000048 segundos.
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ ./2b
Tiempo total: 0.000046 segundos.
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ ./2b
Tiempo total: 0.000050 segundos.
```

Compare los resultados de tiempos de cada uno de sus programas, y responda:

¿Cuál, en general, toma tiempos más largos?

- En general, podemos observar que el programa 1 toma tiempos más largos que el programa concurrente. El promedio de tiempo del programa 1 es de: 0.00527, mientras que el promedio de tiempo del programa concurrente es de 0.000054.

¿Qué causa la diferencia de tiempo, o por qué se tarda más el que se tarda más?

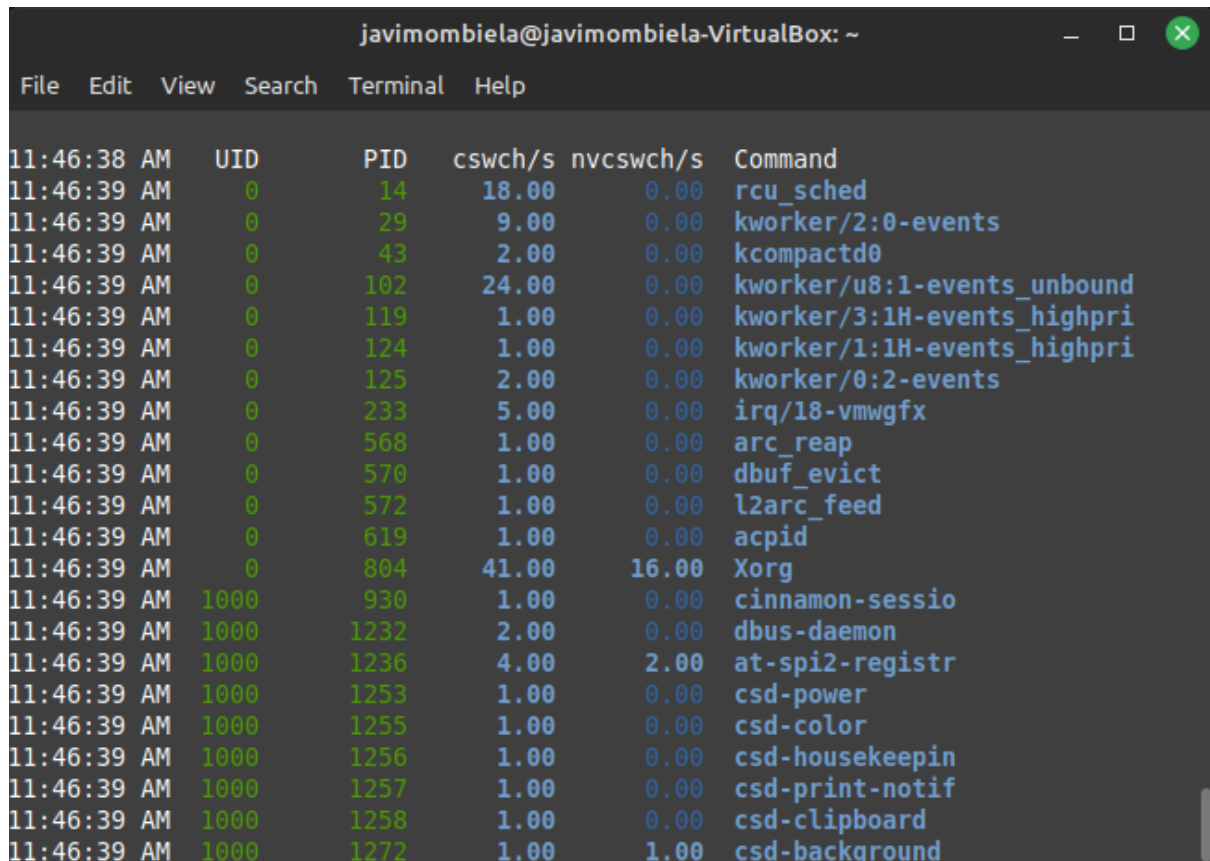
- El primer programa ejecuta de forma secuencial y lineal sin crear procesos adicionales. Utiliza tres ciclos for de un millón de iteraciones cada uno y mide el tiempo total de ejecución de los tres ciclos.
- El segundo programa utiliza la función `fork()` para crear nuevos procesos y ejecutar los ciclos for de forma paralela. El programa padre crea un proceso hijo que a su vez crea un proceso nieto y bisnieto. Cada proceso realiza un ciclo for de un millón de iteraciones que no realizan ninguna operación.
- La diferencia es entonces como ejecutan las instrucciones: el primero de forma secuencial y el segundo de forma paralela utilizando procesos adicionales. El uso del paralelismo puede mejorar el rendimiento de un programa en algunas situaciones, especialmente cuando se trata de tareas intensivas en CPU que se pueden paralelizar.

Ejercicio 3

Investigue un poco sobre los cambios de contexto voluntarios e involuntarios.

- En el contexto de sistemas operativos, los cambios voluntarios e involuntarios en un texto se pueden comparar con los cambios voluntarios e involuntarios en los procesos del sistema operativo. Los cambios voluntarios pueden incluir la creación o eliminación de procesos por parte del usuario o el programador, mientras que los cambios involuntarios pueden ser causados por errores en el código o problemas en el hardware. Estos cambios pueden tener un impacto significativo en el rendimiento del sistema operativo y en la estabilidad general del sistema.

Ejecute el comando `pidstat` en la primera terminal con la opción `-w` para desplegar el número de cambios de contexto que se realizan por proceso; y agregue el parámetro `1` al final para que se realice este reporte cada segundo



Time	UID	PID	cswch/s	nvcschw/s	Command
11:46:38 AM					
11:46:39 AM	0	14	18.00	0.00	rcu_sched
11:46:39 AM	0	29	9.00	0.00	kworker/2:0-events
11:46:39 AM	0	43	2.00	0.00	kcompactd0
11:46:39 AM	0	102	24.00	0.00	kworker/u8:1-events_unbound
11:46:39 AM	0	119	1.00	0.00	kworker/3:1H-events_highpri
11:46:39 AM	0	124	1.00	0.00	kworker/1:1H-events_highpri
11:46:39 AM	0	125	2.00	0.00	kworker/0:2-events
11:46:39 AM	0	233	5.00	0.00	irq/18-vmwgfx
11:46:39 AM	0	568	1.00	0.00	arc_reap
11:46:39 AM	0	570	1.00	0.00	dbuf_evict
11:46:39 AM	0	572	1.00	0.00	l2arc_feed
11:46:39 AM	0	619	1.00	0.00	acpid
11:46:39 AM	0	804	41.00	16.00	Xorg
11:46:39 AM	1000	930	1.00	0.00	cinnamon-sessio
11:46:39 AM	1000	1232	2.00	0.00	dbus-daemon
11:46:39 AM	1000	1236	4.00	2.00	at-spi2-registr
11:46:39 AM	1000	1253	1.00	0.00	csd-power
11:46:39 AM	1000	1255	1.00	0.00	csd-color
11:46:39 AM	1000	1256	1.00	0.00	csd-housekeepin
11:46:39 AM	1000	1257	1.00	0.00	csd-print-notif
11:46:39 AM	1000	1258	1.00	0.00	csd-clipboard
11:46:39 AM	1000	1272	1.00	1.00	csd-background

¿Qué tipo de cambios de contexto incrementa notablemente en cada caso, y por qué?

- Los dos tipos de cambios de contexto que se pueden medir al mover ventanas y escribir en la terminal son "cswch/s" y "nvcschw/s". La primera medida, "cswch/s", muestra cuántas veces por segundo el sistema operativo cambia de un proceso a otro. La segunda medida, "nvcschw/s", representa cuántas veces por segundo un proceso decide dejar voluntariamente la CPU. Si se mueve la ventana de la terminal, es posible que aumente la cantidad de cambios de contexto por segundo. Por otro lado, si se ingresa una entrada en la consola, es posible que la medida "nvcschw/s" cambie.

Modifique los programas de su ejercicio anterior para que desplieguen el índice de sus ciclos con cada iteración. En la segunda terminal, ejecute cada programa y tome el tiempo en segundos que toma cada uno en terminar.

Programa 1

```
Ciclo 3: 999997
Ciclo 3: 999998
Ciclo 3: 999999
Tiempo total: 5.454566 segundos
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$
```

Programa 2

```
Ciclo 999997 del proceso nieto
Ciclo 999998 del proceso nieto
Ciclo 999999 del proceso nieto
Tiempo total: 0.000064 segundos.
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$
```

Ejecute primero el `pidstat` y luego, lo más inmediatamente posible, el programa sin `fork(s)`, y espere a que este último termine. Anote el número de cambios de contexto de cada tipo para el proceso correspondiente a la ejecución de su programa.

Programa sin `fork()`

```
javimombiela@javimombiela-VirtualBox:~$ pidstat -w 5 1 | grep ejer2a
12:00:17 PM 1000      4061      86.85 109511.35  ejer2a
```

Programa con `fork()`

```
javimombiela@javimombiela-VirtualBox:~$ pidstat -w 2 1 | grep ejer2b
11:59:11 AM 1000      4046        0.50      0.00  ejer2b
11:59:11 AM 1000      4047    4530.85  14690.05  ejer2b
11:59:11 AM 1000      4048    4609.45  15078.61  ejer2b
11:59:11 AM 1000      4049    4472.64  18426.87  ejer2b
```

¿Qué diferencia hay en el número y tipo de cambios de contexto de entre programas?

- El número y tipo de cambios de contexto es diferente entre los dos programas. El primer programa es un programa secuencial que utiliza un solo proceso para ejecutar los tres ciclos `for`. No hay cambios de contexto involucrados en este programa, ya que solo se ejecuta un único proceso. El segundo programa, en cambio, utiliza múltiples procesos creados con la llamada al sistema `fork()`. Cada vez que se realiza un `fork()` se crea un nuevo proceso y, por lo tanto, se produce un cambio de contexto. En este caso, se crean un total de cuatro procesos: el padre y sus tres hijos.

¿A qué puede atribuir los cambios de contexto voluntarios realizados por sus programas?

- Los cambios de contextos voluntarios realizados por los programas, son producidos a la hora que los procesos creados por el segundo programa, esperar a que los otros procesos terminan antes de ellos poder continuar con su ejecución. Esto significa que los procesos pueden ceder la CPU de manera voluntaria para poder permitir que los otros procesos se ejecuten.

¿A qué puede atribuir los cambios de contexto involuntarios realizados por sus programas?

- En estos programas, los cambios de contexto involuntarios pueden deberse a la planificación del sistema operativo. El sistema operativo puede decidir cambiar de un proceso a otro para asignar recursos de manera eficiente y maximizar el rendimiento. Además, en el segundo programa, los procesos pueden ser interrumpidos por el proceso padre al finalizar su ejecución y esperar a que los procesos hijos terminen.

¿Por qué el reporte de cambios de contexto para su programa con fork()s muestra cuatro procesos, uno de los cuales reporta cero cambios de contexto?

- La razón por la cual uno de los cambios de contexto reporta 0 es porque el proceso padre espera a que el proceso hijo termine su ejecución y luego se termina a él mismo, por lo que este proceso no realiza cambios de contexto adicionales. Por lo tanto, el reporte muestra que este proceso no reporta cambios de contexto.
Por otro lado, los procesos hijo, nieto y bisnieto ejecutan cada uno un for de 1000000 de iteraciones, lo que significa que cada uno realizará múltiples cambios de contexto mientras se ejecutan. El reporte muestra el número de cambios de contexto realizados por cada proceso durante el tiempo en que se ejecutaron.

Vuelva a realizar los procedimientos de medición de cambios de contexto, pero esta vez haga que se muestre el reporte cada segundo una cantidad indefinida de veces.

Mientras pidstat se ejecuta en una terminal, en la otra ejecute cualquiera de sus programas del ejercicio anterior. Intente intervenir en la ejecución de este programa jugando con la interfaz gráfica o escribiendo en la terminal (o en un editor de texto).

12:59:51 PM 1000 5508 2029.00 5948.00 ejer2b	Ciclo 999994 del proceso bisnieto
afasdfsad12:59:52 PM 1000 5506 2967.00 16244.00 ejer2b	Ciclo 999995 del proceso bisnieto
12:59:52 PM 1000 5507 3182.00 14634.00 ejer2b	Ciclo 999996 del proceso bisnieto
12:59:52 PM 1000 5508 3406.00 6403.00 ejer2b	Ciclo 999997 del proceso bisnieto
fsadfsadf12:59:53 PM 1000 5506 1366.00 5383.00 ejer2b	Ciclo 999998 del proceso bisnieto
12:59:53 PM 1000 5507 1519.00 6838.00 ejer2b	Ciclo 999999 del proceso bisnieto
12:59:53 PM 1000 5508 1467.00 8704.00 ejer2b	Tiempo total: 0.000098 segundos.
dssasdfs	javimombiela@javimombiela-VirtualB

1.00 1202.00 pidstat	Ciclo 999993 del proceso bisnieto
143.00 2373.00 ejer2b	Ciclo 999994 del proceso bisnieto
1395.00 21590.00 ejer2b	Ciclo 999995 del proceso bisnieto
14145.00 0.00 kworker/u8:2-events_unbound	Ciclo 999996 del proceso bisnieto
374.00 683.00 firefox-bin	Ciclo 999997 del proceso bisnieto
26.00 368.00 Socket Process	Ciclo 999998 del proceso bisnieto
14.00 11.00 Web Content	Ciclo 999999 del proceso bisnieto
1.00 0.00 Chroot Helper	Tiempo total: 0.000068 segundos.

¿Qué efecto percibe sobre el número de cambios de contexto de cada tipo?

- A la hora de jugar con la terminal, podemos observar que hay un cambio en el número cambios de contexto en el reporte. En la segunda imagen, podemos observar que abrimos Firefox mientras se corría el programa, esto causa un aumento en el número de cambios de contexto involuntarios, debido a que esta tarea consumo muchos recursos del sistema, por lo que el sistema debe de reasignar los recursos a este diferente proceso creado. En la primera imagen, solo estamos escribiendo en la terminal, esta es una tarea que no requiere muchos recursos, por lo que no hay cambios de contexto significativos.

Ejercicio 4

Ejecute el programa escrito en el inciso anterior en una de las terminales. En la otra ejecuté el siguiente comando: `ps -ael`.

```
6020 pts/0    R+      0:03 ./ejer4a SHELL=/bin/bash SESSION_MANAGER=
6021 pts/0    Z+      0:00 [ejer4a] <defunct>
6022 pts/1    R+      0:00 ps -ael SHELL=/bin/bash SESSION_MANAGER=
javimombiela@javimombiela-VirtualBox:~$
```

¿Qué significa la Z y a qué se debe?

- La letra "Z" en la segunda columna de la salida del comando "ps -ael" indica que el proceso asociado se encuentra en estado zombi. Un proceso zombi es un proceso que ya termino, pero aún hay registro de él en el PCB porque su padre no lo ha cosechado. En este programa, se debe a que se generó un proceso hijo, el cual termino antes que el padre, por lo que se volvió un proceso zombi y se quedara así hasta que el padre lo elimine del PCB.

Ahora modifique su programa para que en lugar de desplegar un mensaje en el proceso hijo, despliegue el conteo de 1 a 4,000,000. El objetivo es que los despliegues en pantalla tomen entre 5 y 15 segundos, por lo que puede incrementar el límite del conteo si es necesario.

Ejecute su programa en una de las dos terminales y en la otra vuelva a ejecutar `ps -ael`. Anote los números de procesos de tanto el padre como el hijo.

```
0 R 1000      2443      1780 97  80    0 - 660 -      pts/0      00:00:01 4b
1 S 1000      2444      2443 55  80    0 - 693 wait_w pts/0      00:00:01 4b
```

Repita el inciso anterior de modo que éste y el próximo paso se realicen antes de que termine el conteo. En la terminal donde ejecutó el comando ps ejecute el siguiente

comando: `kill -9 <numproc>` donde `<numproc>` debe ser reemplazado por el número de proceso padre.

```
0 R 1000 2443 1780 97 80 0 - 660 - pts/0 00:00:01 4b
1 S 1000 2444 2443 55 80 0 - 693 wait_w pts/0 00:00:01 4b
4 R 1000 2445 2262 0 80 0 - 3800 - pts/1 00:00:00 ps
javimombiela@javimombiela-VirtualBox:~$ kill -9 2443
javimombiela@javimombiela-VirtualBox:~$
```

¿Qué sucede en la ventana donde ejecutó su programa?

- En la ventana donde se ejecuta el programa no pasa nada, es decir, el conteo sigue de una forma normal, debido a que el hijo sigue en ejecución.

Vuelva a ejecutar `ps -ael`.

```
5185 pts/0 Ss+ 0:00 bash SSH_AUTH_SOCK=/run/user/1000/keyring
5921 pts/1 Ss 0:00 bash SSH_AUTH_SOCK=/run/user/1000/keyring
6431 pts/0 R 0:04 ./ejer4b SHELL=/bin/bash SESSION_MANAGER=
6436 pts/1 R+ 0:00 ps -ael SHELL=/bin/bash SESSION_MANAGER=
```

¿Quién es el padre del proceso que quedó huérfano?

```
1 R 0 2442 2 7 80 0 - 0 - ? 00:00:02 kworker/u8
1 R 1000 2444 924 54 80 0 - 693 - pts/0 00:00:08 4b
4 R 1000 2446 2262 0 80 0 - 3800 - pts/1 00:00:00 ps
javimombiela@javimombiela-VirtualBox:~$
4 S 1000 924 1 0 80 0 - 4322 ep_pol ? 00:00:00 systemd
```

- Como podemos observar, cuando corrimos el comando `ps -ael` la última vez, ya no existe el proceso padre PID 2443, ya que fue el que eliminamos, pero si existe el hijo que tiene el PID 2444, y como podemos ver, en la quinta columna (PPID) podemos ver los padres y ya no es el proceso con PID 2443, pero ahora es el proceso con el PID 924. Si miramos la segunda imagen, podemos ver que este proceso es el `systemd`, lo que tiene sentido, porque normalmente, el sistema operativo hace el reparenting por default con el `systemd`.

Ejercicio 5

```
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ gcc -o ipc ipc.c
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$ ./ejer5
Proceso padre escribió a en iteración 15
Proceso padre escribió b en iteración 10
Proceso hijo recibió b en iteración 15
Proceso hijo recibió b en iteración 10
Contenido de la memoria compartida:
bbbbbbbbbbbbbbbb
Contenido de la memoria compartida:
bbbbbbbbbb
shm unlink: No such file or directory
javimombiela@javimombiela-VirtualBox:~/Documents/Lab2$
```


¿Qué diferencia hay entre realizar comunicación usando memoria compartida en lugar de usando un archivo de texto común y corriente?

- La principal diferencia entre la comunicación por memoria compartida y por archivos de texto es que la memoria compartida es más rápida, ya que no necesita lectura o escritura de archivos en el disco duro. Además, la memoria compartida puede ser más segura y eficiente debido a la implementación de mecanismos de sincronización para evitar conflictos de escritura.

¿Por qué no se debe usar el file descriptor de la memoria compartida producido por otra instancia para realizar el mmap?

- No se debe usar el file descriptor de otra instancia para acceder a una región de memoria compartida porque las instancias pueden tener diferentes permisos para acceder a la región. También puede ocurrir que la región de memoria compartida haya sido eliminada por la otra instancia antes de que se intente usar el file descriptor. Por lo tanto, es necesario crear y abrir una nueva región de memoria compartida para cada instancia que quiera acceder a ella.

¿Es posible enviar el output de un programa ejecutado con exec a otro proceso por medio de un pipe? Investigue y explique cómo funciona este mecanismo en la terminal (e.g., la ejecución de ls | less).

- Sí, es posible enviar la salida de un programa ejecutado con exec a otro proceso mediante un pipe. En la terminal, se logra ejecutando dos comandos en cadena, separados por el carácter "|" (pipe), que redirige la salida del primer comando al segundo. Esto permite al usuario manipular los datos resultantes del primer comando mediante el segundo proceso.

¿Cómo puede asegurarse de que ya se ha abierto un espacio de memoria compartida con un nombre determinado? Investigue y explique errno.

- Para asegurarse de que un espacio de memoria compartida ya ha sido abierto con un nombre determinado, se puede usar la función shm_open con el mismo nombre y un flag específico que indique que solo se debe abrir el espacio de memoria compartida si ya existe. Si la función devuelve un valor menor que cero, es posible verificar si el error es EEXIST, lo que significa que la memoria compartida ya ha sido creada con el nombre especificado.

errno es una variable global en C que se usa para almacenar el número de error generado por funciones del sistema. Cada número de error tiene una macro correspondiente definida en errno.h. Si una función falla, el valor de errno se establece en el número de error correspondiente y se puede usar para obtener más información sobre el error.

¿Qué pasa si se ejecuta shm_unlink cuando hay procesos que todavía están usando la memoria compartida?

- Cuando se elimina el nombre de la memoria compartida con shm_unlink, la memoria en sí no se destruye, los procesos que ya la han abierto pueden seguir usándola. La memoria compartida se destruirá cuando todos los procesos la cierren. Si se intenta abrir la memoria compartida después de eliminar su nombre, el proceso fallará. Es mejor esperar a que los procesos terminen de usar la memoria compartida antes de eliminar su nombre.

¿Cómo puede referirse al contenido de un espacio en memoria al que apunta un puntero? Observe que su programa deberá tener alguna forma de saber hasta dónde ha escrito su otra instancia en la memoria compartida para no escribir sobre ello.

- Para referirse al contenido de un espacio en memoria al que apunta un puntero, se utiliza el operador de indirección (*). Para saber hasta dónde ha escrito otra instancia en la memoria compartida, se puede utilizar un puntero auxiliar o una variable compartida que almacene el tamaño de los datos escritos en la memoria compartida.

Imagine que una ejecución de su programa sufre un error que termina la ejecución prematuramente, dejando el espacio de memoria compartida abierto y provocando que nuevas ejecuciones se queden esperando el file descriptor del espacio de memoria compartida. ¿Cómo puede liberar el espacio de memoria compartida “manualmente”?

- Podemos liberar el espacio de memoria compartida manualmente mediante el uso de la función shm_unlink(), la cual elimina la región de memoria compartida del sistema de archivos, lo que permitirá que nuevas instancias puedan crearla y acceder a ella.

Observe que el programa que ejecute dos instancias de ipc.c debe cuidar que una instancia no termine mucho antes que la otra para evitar que ambas instancias abran y cierren su propio espacio de memoria compartida. ¿Aproximadamente cuánto tiempo toma la realización de un fork()? Investigue y aplique usleep

- La realización de un fork() toma unos pocos milisegundos, pero el proceso hijo creado por fork() tarda un tiempo adicional en completar la copia de la imagen de memoria del proceso padre. Para asegurar que las dos instancias no abran y cierren su propio espacio de memoria compartida, se puede usar la función usleep() para esperar un cierto tiempo antes de continuar la ejecución del programa después de crear el proceso hijo. El tiempo que se espera puede ser ajustado según la necesidad del programa.

```
./ejer5
Proceso padre escribió a en iteración 15
Proceso padre escribió b en iteración 10
Proceso hijo recibió b en iteración 10
Contenido de la memoria compartida:
bbbbbbbbbb
Proceso hijo recibió b en iteración 15
Contenido de la memoria compartida:
bbbbbbbbbbbbbbbb
shm_unlink: No such file or directory
javimombiela@javimombiela-VirtualBox:~/Documents/G
```

```
1 if (pid == 0)
2     { // Child process
3         usleep(5000);
4         execl("./ipc", "ipc", "10", "b", NULL);
5         perror("execl");
6         return 1;
7     }
```