

Laboratorio #2

Fecha de Entrega: 26 de febrero, 2021.

Estudiante: **Estuardo Ureta 17010**

Descripción: este laboratorio reforzará el concepto de proceso, terminación, relación padre-hijo y cambio de contexto. También se implementarán tres medios comunes de comunicación entre procesos: memoria compartida, *pipes* ordinarios y *pipes* nombrados. Es obligatorio el uso de estos tres medios de comunicación entre procesos. Deberá entregar un documento con las respuestas a las preguntas planteadas en cada ejercicio (incluyendo diagramas o escaneos si es necesario), junto con todos los archivos de código que programe.

Materiales: necesitará una máquina virtual con Linux, como la OSC-2016. Los demás materiales necesarios se descargarán a lo largo del laboratorio.

Contenido:

Ejercicio 1 (10 puntos)

Si no está apagada, apague su máquina virtual y revise que esté usando más de un procesador (recomendable, cuatro) en el menú de configuración de VirtualBox. De no ser así, configúrela para que use más de un procesador (o cámbiese de computadora *host* a una multinúcleo).

Cree un programa en C que ejecute cuatro `fork()`s consecutivos. Luego cree otro programa en C que ejecute `fork()` dentro de un ciclo `for` de cuatro iteraciones.

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

16 procesos

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

Hay tantos procesos en ambos programas porque el número total de procesos = 2^n , donde 'n' es el número de llamadas al sistema con `fork()`. Entonces aquí $n = 4$ por lo tanto son 16 procesos (2^4).

```
estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/SO
$ ls
a.out ej1 ej1a ej2 ej2a ej2aej3 ej2ej3
(base)
estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/SO
$ ./ej1
fork
fork
fork
(base)
estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/SO
$ fork
fork
fork
fork
fork
fork
fork
fork
fork
fork
fork
fork
fork
fork
fork
fork
./ej1a → ciclo
fork
fork
fork
fork
fork
fork
fork
fork
fork
```

Ejercicio 2 (20 puntos)

1. Cree un programa en C que #incluya los encabezados <stdio.h> y <time.h>. Este programa deberá ejecutar tres ciclos for consecutivos, de un millón de iteraciones cada uno. Ninguno de los ciclos deberá desplegar ni hacer nada.
2. Declare, al principio de su programa, dos variables de tipo clock_t. Ejecute la función clock() justo antes del primer ciclo for, almacenando el resultado en una variable de tipo clock_t. También ejecute la llamada a clock() justo después del último for y almacene el resultado en la segunda variable clock_t.
3. En el programa, luego de los tres ciclos, almacene en una variable de tipo double el resultado de la resta entre las variables clock_t (la variable que está antes se resta a la que está después). Tome en cuenta que, por almacenarse en una variable double, la resta debe ser *casteada* a este tipo.
4. Haga que el programa despliegue el contenido de la variable double en pantalla. El especificador de formato que debe usar es %f, que sirve para números de punto flotante.
5. Ejecute su programa varias veces (tres o cinco veces suele exhibir el comportamiento deseado) y apunte los resultados de cada vez.

Resultados:	7415	8219	7249	7031	7150
-------------	------	------	------	------	------

```
estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/S0
$ ./ej2
empezando ciclo de un millon de interacciones..., primero = 28500
Terminaron los 3 loops... Segundo = 35915
Tiempo de CPU: 7415.000000
(base)
estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/S0
$ ./ej2
empezando ciclo de un millon de interacciones..., primero = 794
Terminaron los 3 loops... Segundo = 9013
Tiempo de CPU: 8219.000000
(base)
estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/S0
$ ./ej2
empezando ciclo de un millon de interacciones..., primero = 844
Terminaron los 3 loops... Segundo = 8093
Tiempo de CPU: 7249.000000
(base)
estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/S0
$ ./ej2
empezando ciclo de un millon de interacciones..., primero = 1655
Terminaron los 3 loops... Segundo = 8686
Tiempo de CPU: 7031.000000
(base)
estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/S0
$ ./ej2
empezando ciclo de un millon de interacciones..., primero = 793
Terminaron los 3 loops... Segundo = 7943
Tiempo de CPU: 7150.000000
```

6. Cree un nuevo programa en C que #incluya los encabezados <stdio.h>, <time.h>, <unistd.h> y <sys/wait.h>. Al principio del programa declare tres variables de tipo pid_t y dos variables de tipo clock_t.
7. Este programa hará lo mismo que en el primero, pero de forma concurrente (recuerde que esto no es sinónimo de “paralelo”). Para lograrlo comience por realizar y almacenar el resultado de una llamada a clock() justo antes de un fork(), y almacene el resultado de cada llamada en las variables con tipos correspondientes.
8. Haga que el proceso hijo realice otro fork(), y que este nuevo proceso (sería el proceso nieto) haga también un nuevo fork().
9. El proceso bisnieto (el creado por el fork() más anidado) debe realizar un ciclo for de un millón de iteraciones que no hagan nada. El proceso nieto debe realizar lo mismo que el bisnieto en el inciso anterior, pero de forma exclusiva. Es decir, en el else del if que restringe el for al inicio de este inciso al proceso bisnieto. Asegúrese de que el proceso nieto espere, luego de completar su for, a que termine el proceso bisnieto con wait(NULL).
10. El proceso hijo debe realizar lo mismo, y en las mismas condiciones, que el proceso nieto en el inciso anterior.
11. En el proceso raíz o padre exclusivamente (es decir, en el else menos anidado) espere a que termine la ejecución del proceso hijo y luego ejecute clock(). Despliegue la diferencia entre las variables clock_t y deje terminar el programa.
12. Ejecute este programa la misma cantidad de veces que ejecutó el programa no concurrente.

Resultado No Concurrente	7415	8219	7249	7031	7150
Resultado Concurrente	385	1262	41344	254	33926

```

estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/S0
$ ./ej2a
Empezando..., tiempo1 = 738
tiempo de cpu: 385.000000
(base)
estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/S0
$ ./ej2a
Empezando..., tiempo1 = 731
tiempo de cpu: 1262.000000
(base)
estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/S0
$ ./ej2a
Empezando..., tiempo1 = 694
tiempo de cpu: 41344.000000
(base)
estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/S0
$ ./ej2a
Empezando..., tiempo1 = 754
^[[A
tiempo de cpu: 254.000000
(base)
estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/S0
$ ./ej2a
Empezando..., tiempo1 = 651
tiempo de cpu: 33926.000000

```

13. Compare los resultados de tiempos de cada uno de sus programas, y responda: • ¿Cuál, en general, toma tiempos más largos?

En general: el no concurrente toma tiempos más largos

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

Seguramente es la concurrencia ya que al ser concurrente los procesos pueden soportar dos o más acciones en progreso (aunque no sean precisamente al mismo tiempo).

Ejercicio 3 (20 puntos)

1. Descargue e instale el paquete sysstat usando apt-get en una terminal (o el manejador de paquetes del sistema operativo que tenga, *e.g.*, yum o dpkg):

```
sudo apt-get install sysstat
```

2. Investigue un poco sobre los **cambios de contexto voluntarios e involuntarios**. 3. Abra una segunda terminal y dirijase en ella al directorio donde estén los programas que hizo en el ejercicio anterior.

"voluntario": Proceso realiza llamada al sistema que implica esperar por evento Transición de en ejecución a bloquea

"involuntario": S.O. le quita la UCP al proceso Transición de en ejecución a listo

4. Coloque las terminales en pantalla de forma que pueda ver ambas a la vez.
5. 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. La instrucción debe verse así:

```
pidstat -w 1
```

```
estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/SO
$ pidstat -w 1
Linux 4.19.128-microsoft-standard (LAPTOP-5IN4BIR3) 02/26/21 _x86_64_ (4 CPU)

18:25:40      UID      PID  cswch/s nvcschw/s  Command
18:25:41    1000      41    15.53    0.00    node
18:25:41    1000     141     0.97    0.00    node
18:25:41    1000    5537     4.85    0.00    node
18:25:41    1000    5552     0.97    0.00    pidstat

18:25:41      UID      PID  cswch/s nvcschw/s  Command
18:25:42    1000      41    11.00    0.00    node
18:25:42    1000     141     1.00    0.00    node
18:25:42    1000    5537     4.00    0.00    node
18:25:42    1000    5552     1.00    1.00    pidstat

18:25:42      UID      PID  cswch/s nvcschw/s  Command
18:25:43    1000      41    11.00    1.00    node
18:25:43    1000     141     4.00    0.00    node
18:25:43    1000    5537     9.00    0.00    node
18:25:43    1000    5552     1.00    0.00    pidstat

18:25:43      UID      PID  cswch/s nvcschw/s  Command
18:25:44    1000      41    20.00    0.00    node
18:25:44    1000     141     1.00    0.00    node
18:25:44    1000    5537    23.00    0.00    node
18:25:44    1000    5552     1.00    0.00    pidstat

Average:      UID      PID  cswch/s nvcschw/s  Command
Average:    1000      41    14.39    0.25    node
Average:    1000     141     1.74    0.00    node
Average:    1000    5537    10.17    0.00    node
Average:    1000    5552     0.99    0.25    pidstat
```

- En la terminal que está ejecutando pidstat, teclee cosas y observe qué procesos aparecen o responden ante estas interacciones.

- Xorg hace referencia a la interfaz gráfica y por eso incrementa cuando interactuamos con ella. pidstat al teclear y abrir otras pantallas genera o ejecutar más procesos. Estas conjuntamente con gnome son los procesos principales que note para correr programas.**

- [illegible]

- donde X es la cantidad (o el entero inferior inmediato) de segundos que calculó para el programa sin fork().

- Ya que el segundo programa cuenta con forks implementados con concurrencia el número de cambios voluntarios incrementa y el número de cambios involuntarios disminuye. Estos cambios, claro está, en relación con el primer problema**

- A los procesos generados con fork ya que no empieza otro proceso hasta que termine el anterior.**

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

Esto está ligado a los procesos que por definición, en este caso, vienen en el sistema operativo.

- ¿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?

Probablemente el que muestra cero cambios no se está ejecutando o está bloqueado, sino son los hijos (3) los que sí están ejecutándose.

13. 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 con el siguiente comando:

```
pidstat -w 1
```

14. 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).

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

Aumentan los cambios involuntarios ya que entran a la cola y toman prioridad.

Ejercicio 4 (10 puntos)

1. Escriba un programa en C que realice un fork(). En el proceso hijo debe desplegarse un mensaje en pantalla únicamente, y en el proceso padre (exclusivamente) debe ejecutarse un ciclo while infinito.
2. Abra dos terminales.
3. Ejecute el progrma escrito en el inciso anterior en una de las terminales. En la otra ejecute el siguiente comando:

```
ps -ael
```

```
estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/SO
$ ./ej4a
Proceso Hijo
```

En el resultado verá dos procesos con el nombre de su programa, uno de los cuales tendrá añadido <defunct>. Este proceso también desplegará una Z en la segunda columna.

- ¿Qué significa la Z y a qué se debe?

La Z significa que es un proceso finalizado pero, aún se debe hacer algo con él, está en un estado zombie.

4. 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.
5. Ejecute su programa en una de las dos terminales y en la otra vuelva a ejecutar ps -ael. Anote los números de proceso de tanto el padre como el hijo.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1238119 a.out ej1a ej2a ej2ej3 ejercicio1a.c ejercicio2a.c ejercicio2ej3.c
1238120 ej1 ej2 ej2aej3 ejercicio1.c ejercicio2.c ejercicio2aej3.c ejercicio4.c
1238121 (base)
1238122 estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/SO
1238123 $ ps -ael
1238124 F S 0 1 0 0 80 0 - 223 - ? 00:00:00 init
1238125 4 S 0 5866 1 0 80 0 - 223 - ? 00:00:00 init
1238126 5 S 0 5866 1 0 80 0 - 223 - ? 00:00:00 init
1238127 1 S 0 5867 5866 0 80 0 - 223 - ? 00:00:00 init
1238128 4 S 1000 5868 5867 0 80 0 - 653 do_wai pts/0 00:00:00 sh
1238129 0 S 1000 5869 5868 0 80 0 - 653 do_wai pts/0 00:00:00 sh
1238130 0 S 1000 5874 5869 0 80 0 - 653 do_wai pts/0 00:00:00 sh
1238131 0 R 1000 5876 5874 3 80 0 - 250588 - pts/0 00:00:10 node
1238132 0 S 1000 5914 5876 0 80 0 - 234619 do_epo pts/0 00:00:02 node
1238133 0 S 1000 5925 5876 0 80 0 - 1353 do_wai pts/1 00:00:00 bash
1238134 0 S 1000 5941 5876 0 80 0 - 1353 do_wai pts/2 00:00:00 bash
1238135 0 R 1000 5964 5925 99 80 0 - 591 - pts/1 00:00:03 ej4
1238136 1 R 1000 5965 5964 99 80 0 - 591 - pts/1 00:00:03 ej4
1238137 1 S 1000 5966 5964 10 80 0 - 604 - pts/1 00:00:00 ej4
1238138 1 S 1000 5967 5965 15 80 0 - 604 tty_wr pts/1 00:00:00 ej4
1238139 0 R 1000 5968 5941 0 80 0 - 1452 - pts/2 00:00:00 ps
1238140 (base)
1238141 estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/SO
1238142 $ ps -ael
1238143 F S 0 1 0 0 80 0 - 223 - ? 00:00:00 init
1238144 4 S 0 5866 1 0 80 0 - 223 - ? 00:00:00 init
1238145 5 S 0 5866 1 0 80 0 - 223 - ? 00:00:00 init
1238146 1 S 0 5867 5866 0 80 0 - 223 - ? 00:00:00 init
1238147 4 S 1000 5868 5867 0 80 0 - 653 do_wai pts/0 00:00:00 sh
1238148 0 S 1000 5869 5868 0 80 0 - 653 do_wai pts/0 00:00:00 sh
1238149 0 S 1000 5874 5869 0 80 0 - 653 do_wai pts/0 00:00:00 sh
1238150 0 S 1000 5876 5874 4 80 0 - 250877 do_epo pts/0 00:00:15 node
1238151 0 S 1000 5914 5876 0 80 0 - 234619 do_epo pts/0 00:00:02 node
1238152 0 S 1000 5925 5876 0 80 0 - 1353 do_wai pts/1 00:00:00 bash
1238153 0 S 1000 5941 5876 0 80 0 - 1353 do_wai pts/2 00:00:00 bash
1238154 0 R 1000 5964 5925 99 80 0 - 591 - pts/1 00:00:56 ej4
1238155 1 R 1000 5965 5964 99 80 0 - 591 - pts/1 00:00:56 ej4
1238156 1 S 1000 5966 5964 6 80 0 - 604 - pts/1 00:00:03 ej4
1238157 1 S 1000 5967 5965 6 80 0 - 604 tty_wr pts/1 00:00:03 ej4
1238158 0 R 1000 5969 5941 0 80 0 - 1452 - pts/2 00:00:00 ps
1238159 (base)
1238160 estuardo8u14@LAPTOP-5IN4BIR3 ~/Desktop/SO
$
```

6. 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.

- ¿Qué sucede en la ventana donde ejecutó su programa?

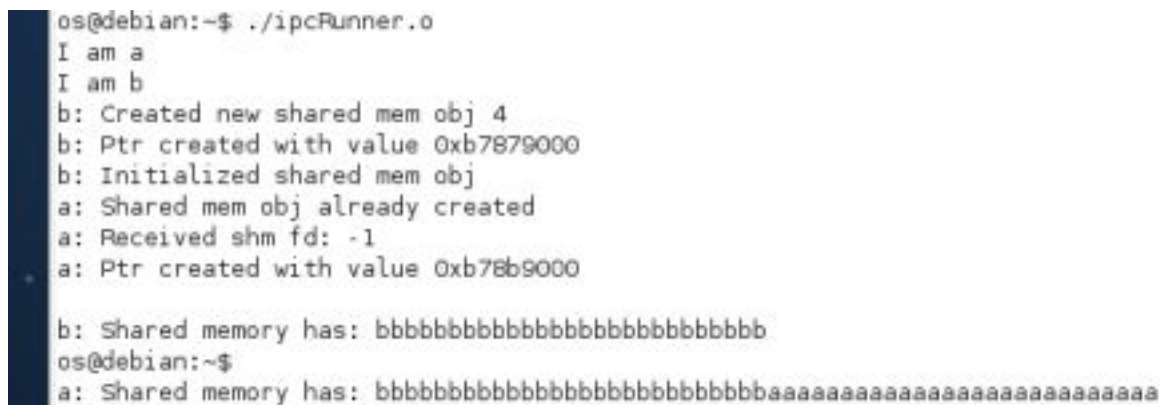
Se mata el programa y termina de correr el proceso.

7. Vuelva a ejecutar ps -ael.

- ¿Quién es el padre del proceso que quedó huérfano?

El padre del proceso huérfano es el primero de cada dos procesos si se trata de un fork.

Ejercicio 5 (40 puntos)



```
os@debian:~$ ./ipcRunner.o
I am a
I am b
b: Created new shared mem obj 4
b: Ptr created with value 0xb7879000
b: Initialized shared mem obj
a: Shared mem obj already created
a: Received shm fd: -1
a: Ptr created with value 0xb78b9000

b: Shared memory has: bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
os@debian:~$
a: Shared memory has: bbbbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Responda las preguntas en la siguiente página.

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

La diferencia principal es que en el archivo de texto se utilizará data duplicada en la memoria mientras que en la memoria compartida esta data se utiliza en los programas ejecutados sin trastocar.

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

Porque son instancias diferentes y por tanto usan distintos espacios en la memoria.

- ¿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`).

Utilizando pipes, en `ls | less` el comando `ls` imprime el contenido del directorio y `less` muestra el contenido de un archivo o una salida de comando y así navegamos en el archivo.

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

Si se ha abierto un espacio de memoria compartida se puede usar el espacio para hacer operaciones y sino sale un error. Así sabemos si está abierto o no y aparte errno guarda el numero de error producido.

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

Si hay referencias al objeto de memoria compartida cuando se utiliza el comando `shm_unlink` el nombre se elimina antes de que regrese la función. No se hace una eliminación de la memoria hasta que el mapa al objeto de memoria compartida este eliminado.

- ¿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.

El contenido hace referencia a cuando se inicializa se le asigna un valor de coordenadas a donde el puntero apunta, valga la redundancia.

- 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”?

Con el comando `munmap`, eliminando mapeos contenidos dentro del espacio de direcciones del proceso.

- 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`.

Con `clock_gettime(fork())` se nota que es muy rápido, de aproximadamente un milisegundo. `Usleep` retrasa la ejecución de un proceso por ciertos segundos para asegurar que dos instancias no terminan mucho antes de la otra dejando espacio inutilizado. ya que un sistema operativo que no esté haciendo nada en algún momento, no está bien realizado.