

## Lab 2

**Sistemas Operativos**Fredy Velásquez - 201011

---

**Ejercicio 1**

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?

```
os@debian:~/Desktop/Lab2$ ./exercisela
hello
hello
hello
hello
hello
hello
hello
hello
hello
os@debian:~/Desktop/Lab2$ ./exerciselb
hello
hello
hello
hello
hello
os@debian:~/Desktop/Lab2$ hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
os@debian:~/Desktop/Lab2$ █
```

Ejecución programa 1a y 1b

En el primero con 4 forks se crea la misma cantidad de procesos que en el segundo con ciclo for, se crean 15 procesos + el padre, dando un total de 16.

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

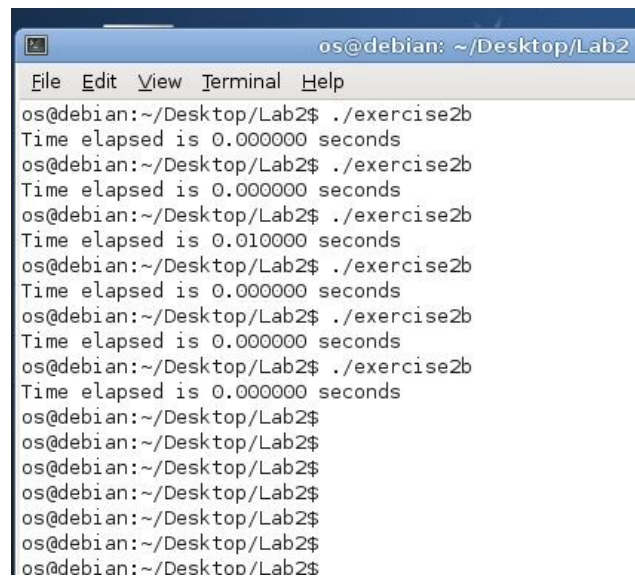
En el primero con 4 forks, se crean 1 proceso hijo por cada proceso, y estos a su vez nuevos hijos. En el segundo con ciclo for, en cada iteración del bucle for padre, se llama a un fork (). Después de cada llamada a fork (), *i* se incrementa, por lo que cada hijo comienza un ciclo for desde *i* antes de que se incremente. Con eso ya podemos asegurar que se crean  $2^n - 1$  procesos hijos, dando un total de 16 contando al padre.

## Ejercicio 2



```
os@debian: ~/Desktop/Lab2
File Edit View Terminal Help
os@debian:~/Desktop/Lab2$ ./exercise2a
Time elapsed is 0.010000 seconds
os@debian:~/Desktop/Lab2$ ./exercise2a
Time elapsed is 0.010000 seconds
os@debian:~/Desktop/Lab2$ ./exercise2a
Time elapsed is 0.000000 seconds
os@debian:~/Desktop/Lab2$ ./exercise2a
Time elapsed is 0.010000 seconds
os@debian:~/Desktop/Lab2$ ./exercise2a
Time elapsed is 0.010000 seconds
os@debian:~/Desktop/Lab2$
```

Ejecución programa 2a



```
os@debian: ~/Desktop/Lab2
File Edit View Terminal Help
os@debian:~/Desktop/Lab2$ ./exercise2b
Time elapsed is 0.000000 seconds
os@debian:~/Desktop/Lab2$ ./exercise2b
Time elapsed is 0.000000 seconds
os@debian:~/Desktop/Lab2$ ./exercise2b
Time elapsed is 0.010000 seconds
os@debian:~/Desktop/Lab2$ ./exercise2b
Time elapsed is 0.000000 seconds
os@debian:~/Desktop/Lab2$ ./exercise2b
Time elapsed is 0.000000 seconds
os@debian:~/Desktop/Lab2$ ./exercise2b
Time elapsed is 0.000000 seconds
os@debian:~/Desktop/Lab2$
os@debian:~/Desktop/Lab2$
os@debian:~/Desktop/Lab2$
os@debian:~/Desktop/Lab2$
os@debian:~/Desktop/Lab2$
os@debian:~/Desktop/Lab2$
```

Ejecución programa 2b

- ¿Cuál, en general, toma tiempos más largos?

### Promedio de tiempos

Ejercicio 2a: 0.008 segundos.

Ejercicio 2b: 0.002 segundos.

Vemos que el primer programa que no es concurrente, se ejecuta en un tiempo mayor.

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

Una sola CPU puede ejecutar solo una operación a la vez, por lo que ejecutar dos procesos significa ejecutar el primero por un tiempo, luego cambiar para ejecutar el segundo y así sucesivamente. Este problema solo se puede

resolver utilizando varias CPU (que al final nuestra máquina virtual tiene 4 núcleos). Entonces los procesos descendientes se están ejecutando al mismo tiempo (concurrente gracias al *fork()*) y, dependiendo de la cantidad de núcleos y el scheduler del OS, es posible que incluso se ejecuten literalmente al mismo tiempo.

En resumen, los procesos en el último programa se ejecutarán al mismo tiempo. Este es básicamente el punto importante de los procesos.

### Ejercicio 3

Time	PID	cs	sm	Command
10:14:25 PM	4981	3.00	0.00	gedit
10:14:25 PM	9964	105.00	4.00	gnome-terminal
10:14:25 PM	10166	1.00	1.00	pidstat
dksnajdknsakdnsakd				
10:14:25 PM	PID	cs	sm	Command
10:14:26 PM	1	1.00	0.00	init
10:14:26 PM	4	1.00	0.00	ksoftirqd/0
10:14:26 PM	15	1.00	0.00	events/0
10:14:26 PM	16	25.00	0.00	events/1
10:14:26 PM	17	1.00	0.00	events/2
10:14:26 PM	18	47.00	0.00	events/3
10:14:26 PM	24	1.00	0.00	sync_supers
10:14:26 PM	170	1.00	0.00	ata/3
10:14:26 PM	1554	35.00	0.00	acpid
10:14:26 PM	1793	65.00	122.00	Xorg
10:14:26 PM	2200	1.00	0.00	gnome-settings-daemon
10:14:26 PM	2203	54.00	0.00	metacity
10:14:26 PM	2207	2.00	0.00	gnome-panel
10:14:26 PM	2210	2.00	0.00	udisks-daemon
10:14:26 PM	2221	2.00	0.00	nautilus
10:14:26 PM	4981	1.00	0.00	gedit
10:14:26 PM	9964	107.00	0.00	gnome-terminal
10:14:26 PM	10166	1.00	0.00	pidstat

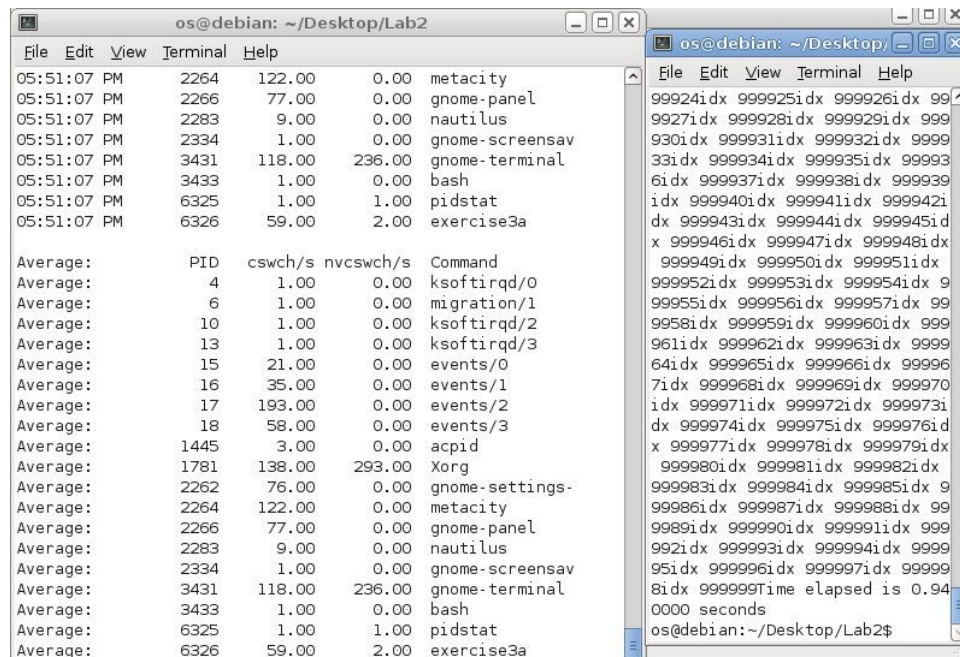
Ejecución de *sysstat*

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

Xorg proporciona el entorno gráfico para Linux, utilizado por GNOME. Los cambios de contexto voluntarios se ven en los procesos de gnome-terminal (107 csch/s, número total de cambios de contexto voluntarios de la tarea realizada por segundo) y Xorg (65 csch/s). Ahora, el único efecto que tiene la entrada de teclado es sobre los cambios de contexto no voluntarios Xorg (122 ncswch/s).

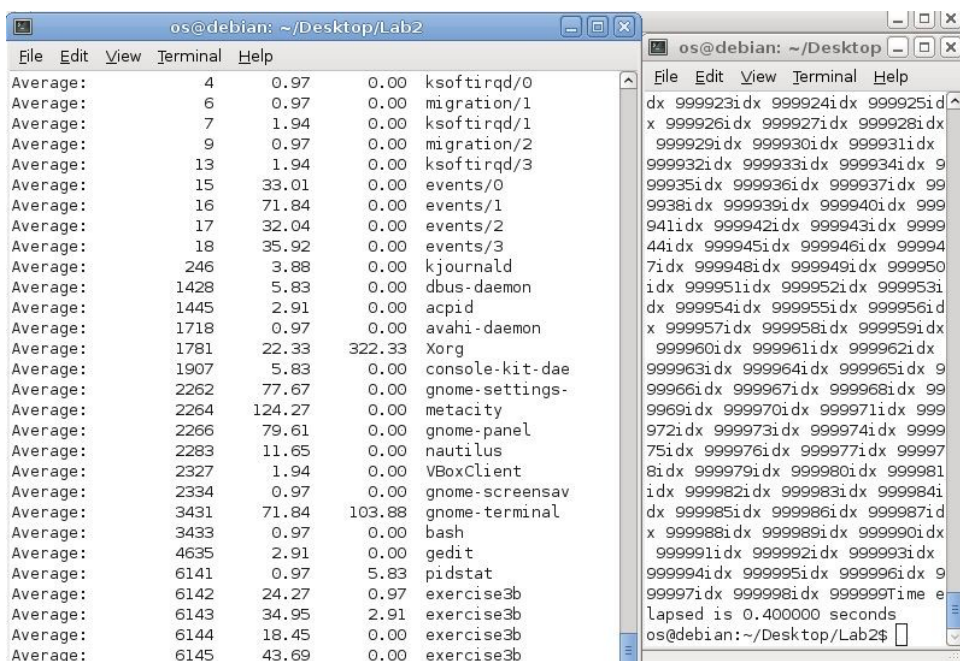
El sistema reconoce como cambios no voluntarios cuando la GUI cambia de manera repentina y recurrente, y esto puede ocurrir también cuando un proceso con mayor prioridad comienza a molestar por recursos mientras se están ejecutando los de prioridad normal.

### Ejecutando pidstat en ambos programas



	PID	cswch/s	nvcsch/s	Command
Average:	4	1.00	0.00	ksoftirqd/0
Average:	6	1.00	0.00	migration/1
Average:	10	1.00	0.00	ksoftirqd/2
Average:	13	1.00	0.00	ksoftirqd/3
Average:	15	21.00	0.00	events/0
Average:	16	35.00	0.00	events/1
Average:	17	193.00	0.00	events/2
Average:	18	58.00	0.00	events/3
Average:	1445	3.00	0.00	acpid
Average:	1781	138.00	293.00	Xorg
Average:	2262	76.00	0.00	gnome-settings-
Average:	2264	122.00	0.00	metacity
Average:	2266	77.00	0.00	gnome-panel
Average:	2283	9.00	0.00	nautilus
Average:	2334	1.00	0.00	gnome-screensav
Average:	3431	118.00	236.00	gnome-terminal
Average:	3433	1.00	0.00	bash
Average:	6325	1.00	1.00	pidstat
Average:	6326	59.00	2.00	exercise3a

Ejecución de `sysstat -w 1 1` para el ejercicio 3a



	PID	cswch/s	nvcsch/s	Command
Average:	4	0.97	0.00	ksoftirqd/0
Average:	6	0.97	0.00	migration/1
Average:	7	1.94	0.00	ksoftirqd/1
Average:	9	0.97	0.00	migration/2
Average:	13	1.94	0.00	ksoftirqd/3
Average:	15	33.01	0.00	events/0
Average:	16	71.84	0.00	events/1
Average:	17	32.04	0.00	events/2
Average:	18	35.92	0.00	events/3
Average:	246	3.88	0.00	kjournald
Average:	1428	5.83	0.00	dbus-daemon
Average:	1445	2.91	0.00	acpid
Average:	1718	0.97	0.00	avahi-daemon
Average:	1781	22.33	322.33	Xorg
Average:	1907	5.83	0.00	console-kit-dae
Average:	2262	77.67	0.00	gnome-settings-
Average:	2264	124.27	0.00	metacity
Average:	2266	79.61	0.00	gnome-panel
Average:	2283	11.65	0.00	nautilus
Average:	2327	1.94	0.00	VBoxClient
Average:	2334	0.97	0.00	gnome-screensav
Average:	3431	71.84	103.88	gnome-terminal
Average:	3433	0.97	0.00	bash
Average:	4635	2.91	0.00	gedit
Average:	6141	0.97	5.83	pidstat
Average:	6142	24.27	0.97	exercise3b
Average:	6143	34.95	2.91	exercise3b
Average:	6144	18.45	0.00	exercise3b
Average:	6145	43.69	0.00	exercise3b

Ejecución de `sysstat -w 1 1` para el ejercicio 3b

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

Para el primer ejercicio (el que es sin *forks()*) aparece solamente un comando que sería el de “exercise3a”, con un promedio de total de cambios de contexto por segundo de 59 cswch/s (voluntarios) y 2 nvcswh/s (no voluntarios).

Para el segundo ejercicio (el que es con *forks()*) aparecen cuatro comandos con nombre “exercise3b”, con un total de cambios de contexto por segundo de 29 cswch/s (voluntarios) y 1 nvcswh/s (no voluntarios).

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

Los cambios de contexto voluntarios fueron realizados cuando el proceso salió de la CPU porque no tiene nada más que hacer (mientras espera que suceda algo externo). Ocurren cuando el proceso cede el control a la CPU (es decir, esperando I/O, o en *sleep*).

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

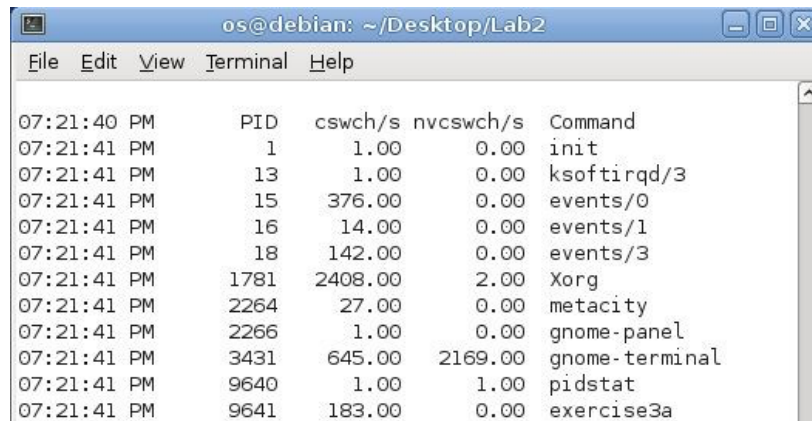
Los cambios de contexto involuntarios fueron realizados al querer continuar con algunos cálculos, pero el OS decide que es hora de cambiar a otro proceso. Ocurren cuando el Process Scheduling interrumpe el proceso (es decir, el intervalo de tiempo expiró).

- ¿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?

Se muestran cuatro procesos ya que fue la cantidad de procesos que fueron creados (padre, hijo, nieto y bisnieto). Un cambio de contexto no voluntario ocurre cuando un proceso deja de responder, sin embargo, también ocurre cuando la tarea no se completa dentro del intervalo de tiempo dado. Por lo tanto, que un proceso tenga 0 cambios de contexto no voluntarios se debe a la llamada al sistema para así esperar debidamente a que el proceso hijo termine (gracias al *wait()*).



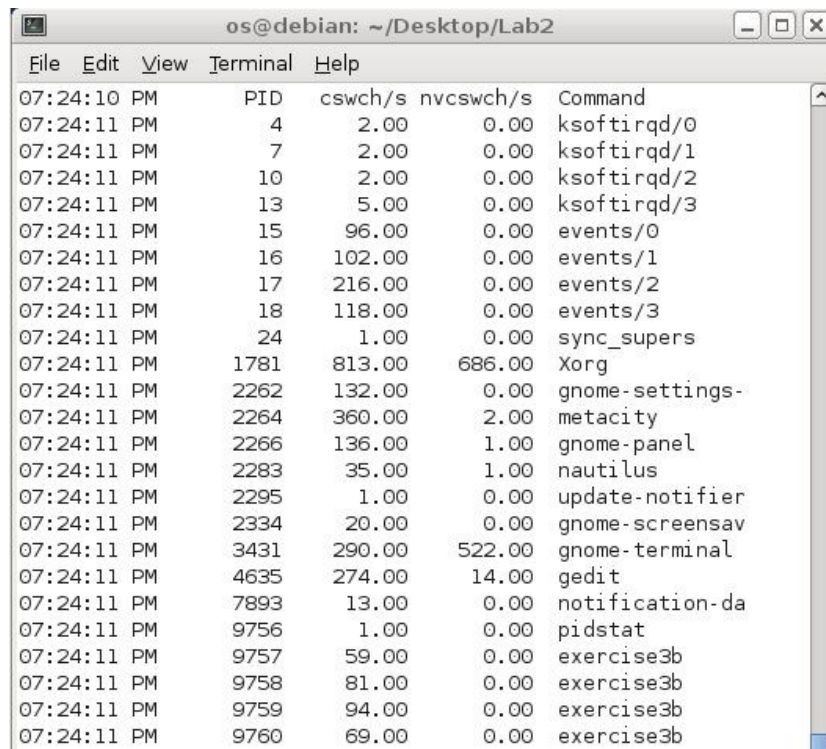
## Ejecutando pidstat interviniendo en la GUI



A terminal window titled 'os@debian: ~/Desktop/Lab2' showing the output of the `sysstat -w 1` command. The output is a table with columns: Time, PM, PID, cswch/s, nvcswch/s, and Command. The data shows system processes like `init`, `ksoftirqd/3`, `events/0`, `events/1`, `events/3`, `Xorg`, `metacity`, `gnome-panel`, `gnome-terminal`, `pidstat`, and `exercise3a`.

Time	PM	PID	cswch/s	nvcswch/s	Command
07:21:40	PM				
07:21:41	PM	1	1.00	0.00	init
07:21:41	PM	13	1.00	0.00	ksoftirqd/3
07:21:41	PM	15	376.00	0.00	events/0
07:21:41	PM	16	14.00	0.00	events/1
07:21:41	PM	18	142.00	0.00	events/3
07:21:41	PM	1781	2408.00	2.00	Xorg
07:21:41	PM	2264	27.00	0.00	metacity
07:21:41	PM	2266	1.00	0.00	gnome-panel
07:21:41	PM	3431	645.00	2169.00	gnome-terminal
07:21:41	PM	9640	1.00	1.00	pidstat
07:21:41	PM	9641	183.00	0.00	exercise3a

Ejecución de `sysstat -w 1` para el ejercicio 3a



A terminal window titled 'os@debian: ~/Desktop/Lab2' showing the output of the `sysstat -w 1` command. The output is a table with columns: Time, PM, PID, cswch/s, nvcswch/s, and Command. The data shows system processes like `ksoftirqd/0`, `ksoftirqd/1`, `ksoftirqd/2`, `ksoftirqd/3`, `events/0`, `events/1`, `events/2`, `events/3`, `sync_supers`, `Xorg`, `gnome-settings-`, `metacity`, `gnome-panel`, `nautilus`, `update-notifier`, `gnome-screensav`, `gnome-terminal`, `gedit`, `notification-da`, `pidstat`, `exercise3b`, and `exercise3b`.

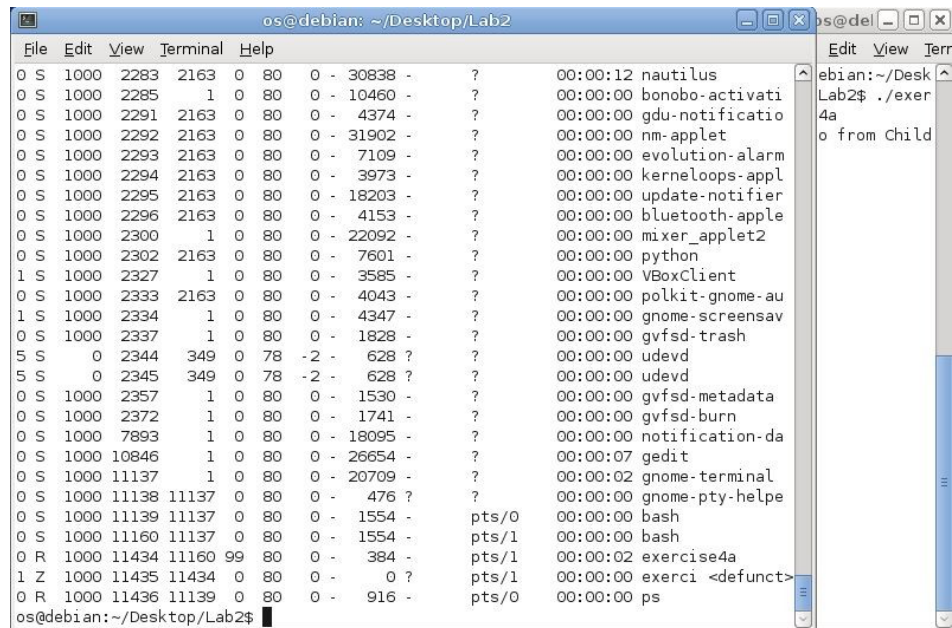
Time	PM	PID	cswch/s	nvcswch/s	Command
07:24:10	PM				
07:24:11	PM	4	2.00	0.00	ksoftirqd/0
07:24:11	PM	7	2.00	0.00	ksoftirqd/1
07:24:11	PM	10	2.00	0.00	ksoftirqd/2
07:24:11	PM	13	5.00	0.00	ksoftirqd/3
07:24:11	PM	15	96.00	0.00	events/0
07:24:11	PM	16	102.00	0.00	events/1
07:24:11	PM	17	216.00	0.00	events/2
07:24:11	PM	18	118.00	0.00	events/3
07:24:11	PM	24	1.00	0.00	sync_supers
07:24:11	PM	1781	813.00	686.00	Xorg
07:24:11	PM	2262	132.00	0.00	gnome-settings-
07:24:11	PM	2264	360.00	2.00	metacity
07:24:11	PM	2266	136.00	1.00	gnome-panel
07:24:11	PM	2283	35.00	1.00	nautilus
07:24:11	PM	2295	1.00	0.00	update-notifier
07:24:11	PM	2334	20.00	0.00	gnome-screensav
07:24:11	PM	3431	290.00	522.00	gnome-terminal
07:24:11	PM	4635	274.00	14.00	gedit
07:24:11	PM	7893	13.00	0.00	notification-da
07:24:11	PM	9756	1.00	0.00	pidstat
07:24:11	PM	9757	59.00	0.00	exercise3b
07:24:11	PM	9758	81.00	0.00	exercise3b
07:24:11	PM	9759	94.00	0.00	exercise3b
07:24:11	PM	9760	69.00	0.00	exercise3b

Ejecución de `sysstat -w 1` para el ejercicio 3b

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

El efecto que producen los eventos de I/O ayudan a disminuir los cambios de contexto no voluntarios, pero han incrementado de manera significativa los cambios de contexto voluntarios. Los voluntarios fueron incrementados porque el proceso no puede obtener los recursos necesarios. Por ejemplo, cuando los recursos del sistema, como I/O (mouse) son insuficientes, se produce un cambio de contexto voluntario.

## Ejercicio 4



```

os@debian: ~/Desktop/Lab2
File Edit View Terminal Help
0 S 1000 2283 2163 0 80 0 - 30838 - ? 00:00:12 nautilus
0 S 1000 2285 1 0 80 0 - 10460 - ? 00:00:00 bonobo-activati
0 S 1000 2291 2163 0 80 0 - 4374 - ? 00:00:00 gdu-notificatio
0 S 1000 2292 2163 0 80 0 - 31902 - ? 00:00:00 nm-applet
0 S 1000 2293 2163 0 80 0 - 7109 - ? 00:00:00 evolution-alarm
0 S 1000 2294 2163 0 80 0 - 3973 - ? 00:00:00 kerneloops-appl
0 S 1000 2295 2163 0 80 0 - 18203 - ? 00:00:00 update-notifier
0 S 1000 2296 2163 0 80 0 - 4153 - ? 00:00:00 bluetooth-apple
0 S 1000 2300 1 0 80 0 - 22092 - ? 00:00:00 mixer_applet2
0 S 1000 2302 2163 0 80 0 - 7601 - ? 00:00:00 python
1 S 1000 2327 1 0 80 0 - 3585 - ? 00:00:00 VBoxClient
0 S 1000 2333 2163 0 80 0 - 4043 - ? 00:00:00 polkit-gnome-au
1 S 1000 2334 1 0 80 0 - 4347 - ? 00:00:00 gnome-screensav
0 S 1000 2337 1 0 80 0 - 1828 - ? 00:00:00 gvfsd-trash
5 S 0 2344 349 0 78 -2 - 628 ? ? 00:00:00 udevd
5 S 0 2345 349 0 78 -2 - 628 ? ? 00:00:00 udevd
0 S 1000 2357 1 0 80 0 - 1530 - ? 00:00:00 gvfsd-metadata
0 S 1000 2372 1 0 80 0 - 1741 - ? 00:00:00 gvfsd-burn
0 S 1000 7893 1 0 80 0 - 18095 - ? 00:00:00 notification-da
0 S 1000 10846 1 0 80 0 - 26654 - ? 00:00:07 gedit
0 S 1000 11137 1 0 80 0 - 20709 - ? 00:00:02 gnome-terminal
0 S 1000 11138 11137 0 80 0 - 476 ? ? 00:00:00 gnome-pty-helpe
0 S 1000 11139 11137 0 80 0 - 1554 - pts/0 00:00:00 bash
0 S 1000 11160 11137 0 80 0 - 1554 - pts/1 00:00:00 bash
0 R 1000 11434 11160 99 80 0 - 384 - pts/1 00:00:02 exercise4a
1 Z 1000 11435 11434 0 80 0 - 0 ? pts/1 00:00:00 exerci <defunct>
0 R 1000 11436 11139 0 80 0 - 916 - pts/0 00:00:00 ps
os@debian:~/Desktop/Lab2$

```

Ejecución de `ps -aef` para el ejercicio 4a

- ¿Qué significa la Z y a qué se debe?  
Significa “zombie” o “defunct” (difunto). Esto se debe a que el proceso ha completado su tarea o ha sido dañado o eliminado, pero sus procesos secundarios aún se están ejecutando o estos procesos principales están monitoreando su proceso secundario.
- ¿Qué sucede en la ventana donde ejecutó su programa?

```

0 R 1000 12178 11160 99 80 0 - 384 - pts/1 00:00:26 exercise4b
1 S 1000 12179 12178 1 80 0 - 385 - pts/1 00:00:00 exercise4b
0 R 1000 12202 11139 0 80 0 - 916 - pts/0 00:00:00 ps

```

El proceso hijo (que es el que imprime los números) sigue funcionando sin ningún problema hasta que termina su ejecución.

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

Ejecución de `kill -9 <numproc>` y luego `ps -aef` para el ejercicio 4b

Su padre ahora es *init* (PID 1), este mismo adopta el proceso hijo huérfano. Cuando los procesos huérfanos mueren, no permanecen como procesos zombis; en su lugar, *init* los espera hasta que finalicen.

## Ejercicio 5

Ejecución exitosa del ejercicio 5

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

Una ventaja del modelo de memoria compartida es que la comunicación de la memoria es más rápida en comparación con el modelo de transmisión de mensajes en la misma máquina. Sin embargo, puede crear problemas como la sincronización y la protección de la memoria que deben tratarse en el código.

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

Diferentes *file descriptors* en diferentes procesos pueden identificar el mismo archivo físico. Si otro proceso asigna el mismo archivo a su espacio de



memoria virtual, ese segundo proceso puede establecer diferentes protecciones. Como tal, es posible que una región marcada como de “solo lectura” en un proceso pueda cambiar mientras el proceso se está ejecutando (establecer otras protecciones nuevas).

- ¿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. La terminal puede redirigir la entrada (I/O) manipulando los *file descriptors* del proceso hijo. Un proceso hijo recién creado hereda los *file descriptors* abiertos de su padre, específicamente el mismo teclado para `stdin` y la pantalla de terminal para `stdout` y `stderr`. En C, se puede usar `dup()` y `dup2()` para lograr esto, pero con la siguiente condición: si conecta un extremo de un *pipe* a la entrada estándar o salida estándar a través de `dup2()` (o `dup()`), se debe cerrar ambos *file descriptors* devueltos por `pipe()`.

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

```
/* Create the shared memory object. */
shm_fd = shm_open(name, O_RDWR | O_CREAT | O_EXCL, 0600);
if (shm_fd == -1) {
    fprintf(stderr, "%s: Cannot create shared memory object: %s.\n", name, strerror(errno));
    return EXIT_FAILURE;
}
```

Luego de crear el objeto de memoria compartida, se puede acceder al error lanzado por `shm_open()` al incluir la librería con `#include <errno.h>`. El archivo de encabezado `<errno.h>` define la variable entera *errno*, que se establece mediante llamadas al sistema y algunas funciones de biblioteca en caso de error para indicar qué salió mal.

Además se puede obtener el nombre del error como string, y así en vez de mostrar el *errno* 17, muestra “*File exists*”:

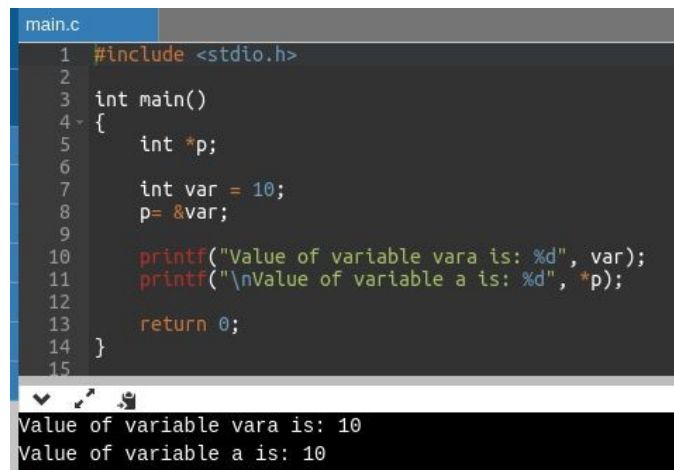
```
/holi: Cannot create shared memory object: File exists.
```

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

Esto solo desvincula el objeto de memoria compartida, incluso si todavía algún proceso está usando el objeto. El objeto se eliminará solo después de que se cierren todas las referencias abiertas.

- ¿Cómo puede referirse al contenido de un espacio en memoria al que apunta un puntero?

Se usa el operador "Address of"(&) y el operador "Value at Address"(\*). Esto, al igual que en lenguajes C, C++ y Go, al usar el operador \* podemos acceder al valor de una variable a través de un puntero.



```
main.c
1  #include <stdio.h>
2
3  int main()
4  {
5      int *p;
6
7      int var = 10;
8      p = &var;
9
10     printf("Value of variable vara is: %d", var);
11     printf("\nValue of variable a is: %d", *p);
12
13     return 0;
14 }
15
```

Value of variable vara is: 10  
Value of variable a is: 10

- 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. 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"?

Para eliminar un segmento de memoria compartida después de que algún programa no pudo separarse de él antes de salir, se debería de ejecutar algo muy parecido a `shm_remove()` si existiera, pero no existe en esta versión de C. Sin embargo, podemos usar `shmctl(shm_id, IPC_RMID, NULL)`; donde `shm_id` es el ID de memoria compartida y `IPC_RMID` indica que se trata de una operación de eliminación.

- 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`.

Depende de la velocidad del procesador. En mi computadora, toma alrededor de 100 a 450 microsegundos. De acuerdo a Pike, Presotto, Ken y Trickey (1991), tomaba alrededor de 700 microsegundos.

```
C time.c
1  #include <sys/time.h>
2  #include <stdio.h>
3
4  int main(){
5
6      struct timeval stop, start;
7      gettimeofday(&start, NULL);
8      //do stuff
9      if (fork() == 0){
10
11      } else {
12          wait(NULL);
13          gettimeofday(&stop, NULL);
14          printf("took %lu us\n", (stop.tv_sec -
15      });
```

```
took 471 us
```

```
took 471 us
```

```
took 502 us
```

### Literatura citada

Pike, R., Presotto, D., Ken, T. y Trickey, H. (1991). Plan 9 and Unix systems. AUUGN, 12(1), 75.