

## Laboratorio #2:

### Comunicación entre procesos del sistema operativo

#### Ejercicio 1 (10 puntos):

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
//compilar en terminal es gcc -o ejerci1o1 ejerci1o1.c
// ejecutar es ./ejerci1o1 y enter
int main(){
    printf("Prueba 1\n");
    fork();
    printf("Prueba 2\n");
    fork();
    printf("Prueba 3\n");
    fork();
    printf("Prueba 4\n");
    fork();
    printf("Prueba 5\n");

    return 0
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
//compilar en terminal es gcc -o ejerci1o1 ejerci1o1.c
// ejecutar es ./ejerci1o1 y enter
int main(){
    int sum, count = 0;
    for(count = 0; count<=3; ++count){
        fork();
        printf(" Hola \n");
    }

    return 0;
}
```

Figura #1: Programas creados usando fork.

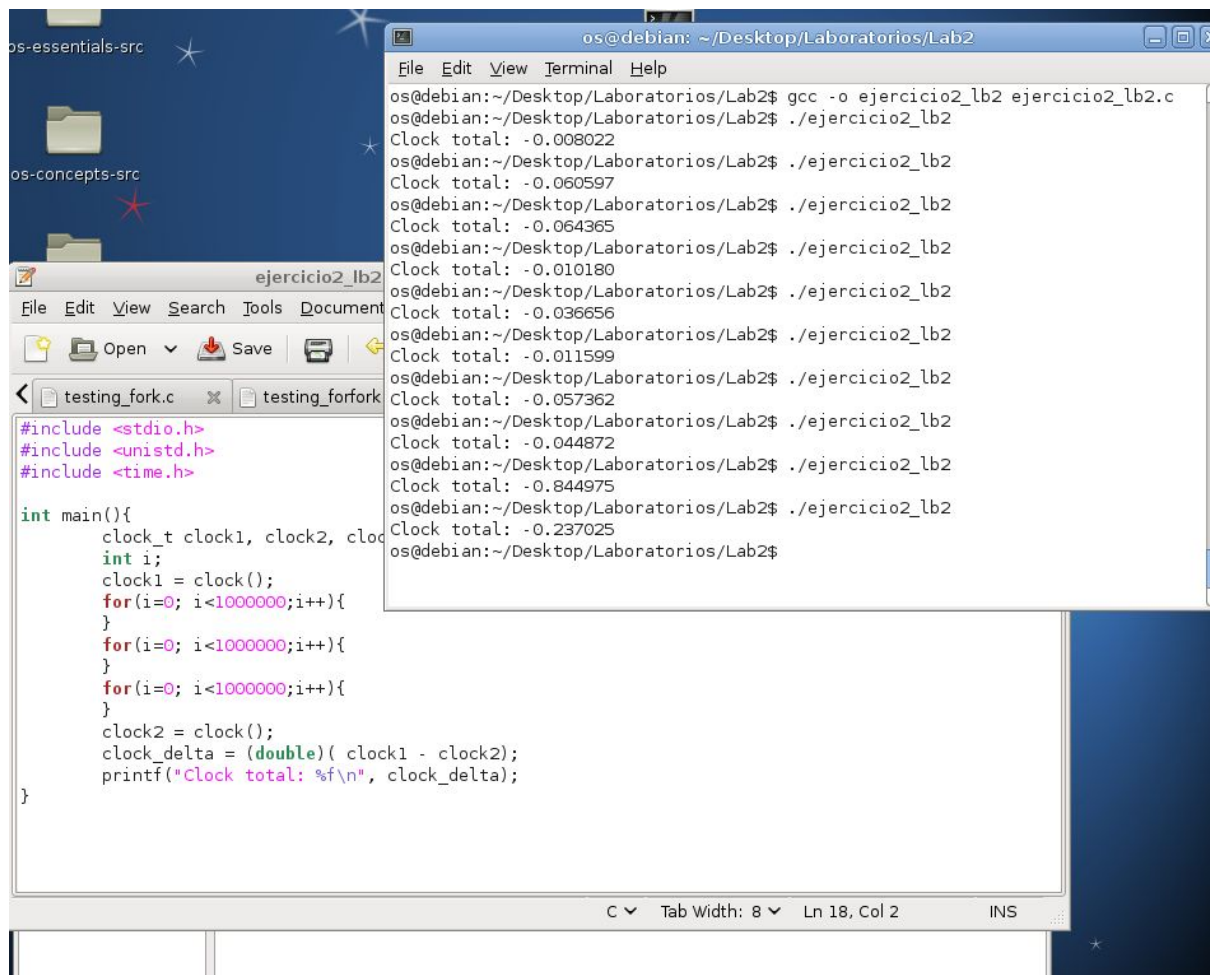
¿Cuántos procesos se crean en cada uno de los programas?

**R//** En este caso, el primer programa sin for, contiene 16 procesos, siendo  $2^4 = 16$  *procesos*, por los 4 forks. En el programa del *for* son 30 procesos, a pesar de tener 4 forks.

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

**R//** Podemos darnos cuenta que a diferencia del programa sin fork, es que dentro del for, la segunda vez que se ejecuta el ciclo, lo ejecutan el padre y el hijo: cada uno ejecuta fork (), y a partir de ese momento hay 4 procesos: los 2 primeros y sus nuevos hijos. Dentro de esas iteraciones, los hijos harán una iteración menos, aumentando la cantidad de procesos ejecutados, que en este caso fueron 30 procesos.

## Ejercicio 2 (20 puntos):



```
#include <stdio.h>
#include <unistd.h>
#include <time.h>

int main(){
    clock_t clock1, clock2, clock_delta;
    int i;
    clock1 = clock();
    for(i=0; i<1000000;i++){
        for(i=0; i<1000000;i++){
            for(i=0; i<1000000;i++){
                clock2 = clock();
                clock_delta = (double)( clock1 - clock2);
                printf("Clock total: %f\n", clock_delta);
            }
        }
    }
}
```

```
os@debian: ~/Desktop/Laboratorios/Lab2
File Edit View Terminal Help
os@debian:~/Desktop/Laboratorios/Lab2$ gcc -o ejercicio2_lb2 ejercicio2_lb2.c
os@debian:~/Desktop/Laboratorios/Lab2$ ./ejercicio2_lb2
Clock total: -0.008022
os@debian:~/Desktop/Laboratorios/Lab2$ ./ejercicio2_lb2
Clock total: -0.060597
os@debian:~/Desktop/Laboratorios/Lab2$ ./ejercicio2_lb2
Clock total: -0.064365
os@debian:~/Desktop/Laboratorios/Lab2$ ./ejercicio2_lb2
Clock total: -0.010180
os@debian:~/Desktop/Laboratorios/Lab2$ ./ejercicio2_lb2
Clock total: -0.036656
os@debian:~/Desktop/Laboratorios/Lab2$ ./ejercicio2_lb2
Clock total: -0.011599
os@debian:~/Desktop/Laboratorios/Lab2$ ./ejercicio2_lb2
Clock total: -0.057362
os@debian:~/Desktop/Laboratorios/Lab2$ ./ejercicio2_lb2
Clock total: -0.044872
os@debian:~/Desktop/Laboratorios/Lab2$ ./ejercicio2_lb2
Clock total: -0.844975
os@debian:~/Desktop/Laboratorios/Lab2$ ./ejercicio2_lb2
Clock total: -0.237025
os@debian:~/Desktop/Laboratorios/Lab2$
```

**Figura 2:** Programa no concurrente que muestra la diferencia de los tiempos.

The image shows a Gedit editor window titled 'ejercicio2b\_lb2.c (~/Desktop/Laboratorios/Lab2) - gedit' and a terminal window titled 'os@debian: ~/Desktop/Laboratorios/Lab2'. The Gedit window contains the following C code:

```
(){\n    clock_t clock1, clock2, clock_delta;\n    pid_t ciclo1, ciclo2, ciclo3;\n    int i;\n    clock1 = clock();\n    if ((ciclo1 = fork()) == 0){\n        //printf("Hijo \\n");\n    }\n    else{\n        if(( ciclo2 = fork()) == 0){\n            //printf("Nietao \\n");\n        }\n        else{\n            if ((ciclo3 = fork()) == 0)\n                //printf("Bisnieto \\n");\n            for(i=0; i<1000000;i++)\n                ;\n        }\n        else{\n            for(i=0; i<1000000;i++)\n                ;\n            wait(NULL);\n        }\n        for(i=0; i<1000000;i++){ }\n        wait(NULL);\n    }\n    wait(NULL);\n    clock2 = clock();\n    clock_delta = (double)( clock1 - clock2);\n    printf("Clock total: %f\\n", clock_delta);\n    return 0;\n}
```

The terminal window shows the execution of the program. It displays the output of the program, which is the clock total for each execution. The output is as follows:

```
Clock total: -0.126431\nClock total: -0.126431\nClock total: -0.126431\nos@debian:~/Desktop/Laboratorios/Lab2$ ./ejercicio2b_lb2\nClock total: -1.240578\nClock total: -1.240578\nClock total: -1.240578\nos@debian:~/Desktop/Laboratorios/Lab2$ ./ejercicio2b_lb2\nClock total: -0.024398\nClock total: -0.024398\nClock total: -0.024398\nos@debian:~/Desktop/Laboratorios/Lab2$ ./ejercicio2b_lb2\nClock total: -0.309935\nClock total: -0.309935\nClock total: -0.309935\nos@debian:~/Desktop/Laboratorios/Lab2$ ./ejercicio2b_lb2\nClock total: -0.131015\nClock total: -0.131015\nClock total: -0.131015\nos@debian:~/Desktop/Laboratorios/Lab2$ ./ejercicio2b_lb2\nClock total: -0.102133\nClock total: -0.102133\nClock total: -0.102133\nos@debian:~/Desktop/Laboratorios/Lab2$
```

**Figura 3:** Programa concurrente que muestra la diferencia de los tiempos con forks.

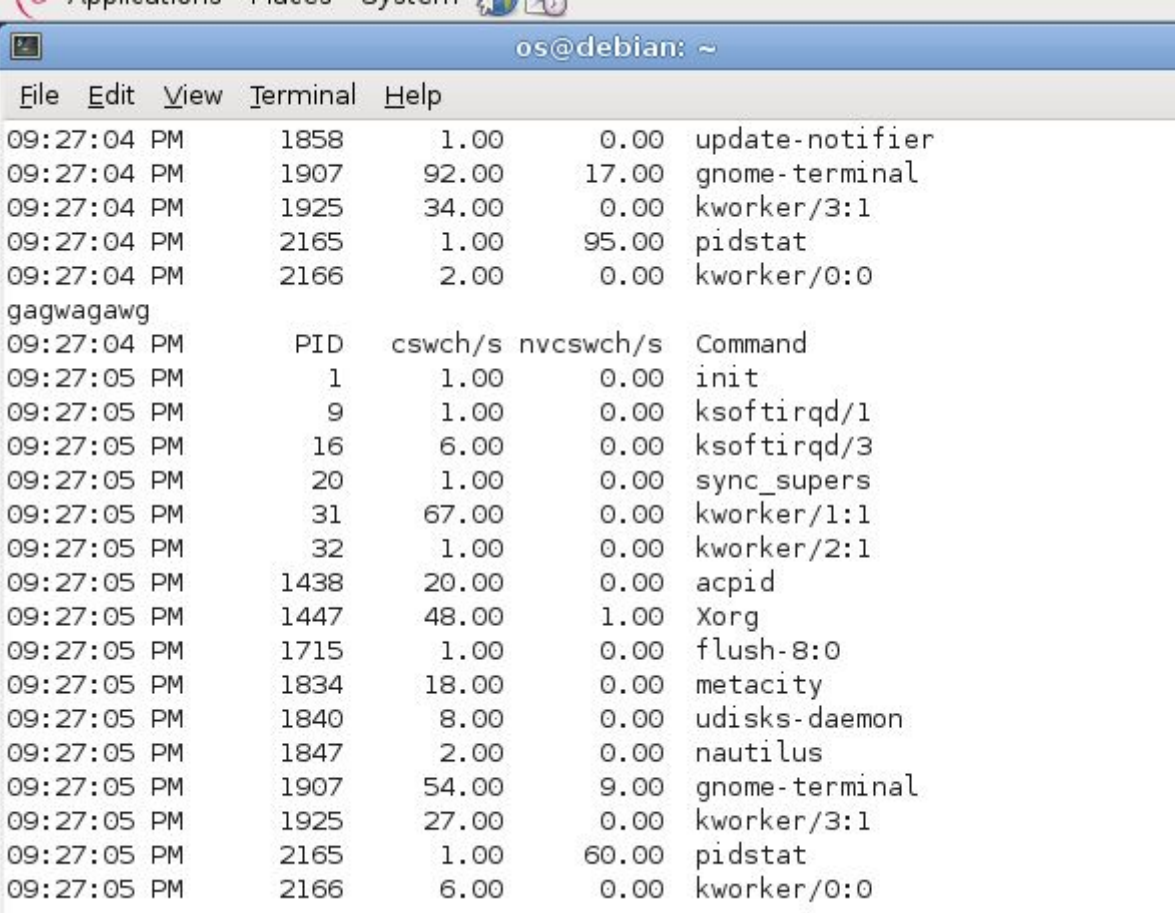
¿Cuál, en general, toma tiempos más largos?

**R//** En este caso, tarda más el programa concurrente, ya que, incluso promediando los resultados, el promedio del programa concurrente es mayor que el programa sin forks.

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

**R//** En el caso del programa concurrente, se tarda más, debido a que los hijos de cada generación o fork espera a que el proceso hijo termine su ejecución y sus forks, lo cual vemos que son las funciones wait(). Una vez terminado, procede el padre a hacer su ejecución, hasta llegar a la raíz.

### Ejercicio 3 (20 puntos):



| 09:27:04 PM | 1858 | 1.00    | 0.00     | update-notifier |
|-------------|------|---------|----------|-----------------|
| 09:27:04 PM | 1907 | 92.00   | 17.00    | gnome-terminal  |
| 09:27:04 PM | 1925 | 34.00   | 0.00     | kworker/3:1     |
| 09:27:04 PM | 2165 | 1.00    | 95.00    | pidstat         |
| 09:27:04 PM | 2166 | 2.00    | 0.00     | kworker/0:0     |
| gagwagawg   |      |         |          |                 |
| 09:27:04 PM | PID  | cswch/s | nvcsch/s | Command         |
| 09:27:05 PM | 1    | 1.00    | 0.00     | init            |
| 09:27:05 PM | 9    | 1.00    | 0.00     | ksoftirqd/1     |
| 09:27:05 PM | 16   | 6.00    | 0.00     | ksoftirqd/3     |
| 09:27:05 PM | 20   | 1.00    | 0.00     | sync_supers     |
| 09:27:05 PM | 31   | 67.00   | 0.00     | kworker/1:1     |
| 09:27:05 PM | 32   | 1.00    | 0.00     | kworker/2:1     |
| 09:27:05 PM | 1438 | 20.00   | 0.00     | acpid           |
| 09:27:05 PM | 1447 | 48.00   | 1.00     | Xorg            |
| 09:27:05 PM | 1715 | 1.00    | 0.00     | flush-8:0       |
| 09:27:05 PM | 1834 | 18.00   | 0.00     | metacity        |
| 09:27:05 PM | 1840 | 8.00    | 0.00     | udisks-daemon   |
| 09:27:05 PM | 1847 | 2.00    | 0.00     | nautilus        |
| 09:27:05 PM | 1907 | 54.00   | 9.00     | gnome-terminal  |
| 09:27:05 PM | 1925 | 27.00   | 0.00     | kworker/3:1     |
| 09:27:05 PM | 2165 | 1.00    | 60.00    | pidstat         |
| 09:27:05 PM | 2166 | 6.00    | 0.00     | kworker/0:0     |

**Figura #4:** Terminal uno que muestra cambios de contexto incrementando por cada vez que se escriba.

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

Se cambió Xorg, algunos kworkers, y gnome terminal. Entre todos los programas que estén preparados para ejecutarse, la rutina selecciona uno de ellos siguiendo algún algoritmo equitativo, en cada iteración que muestra la terminal. Se toma el estado previamente copiado en la memoria principal y se vuelca en los registros del microprocesador. Finalmente, la rutina termina su ejecución saltando a la instrucción que estaba pendiente de ejecutar en el programa seleccionado.



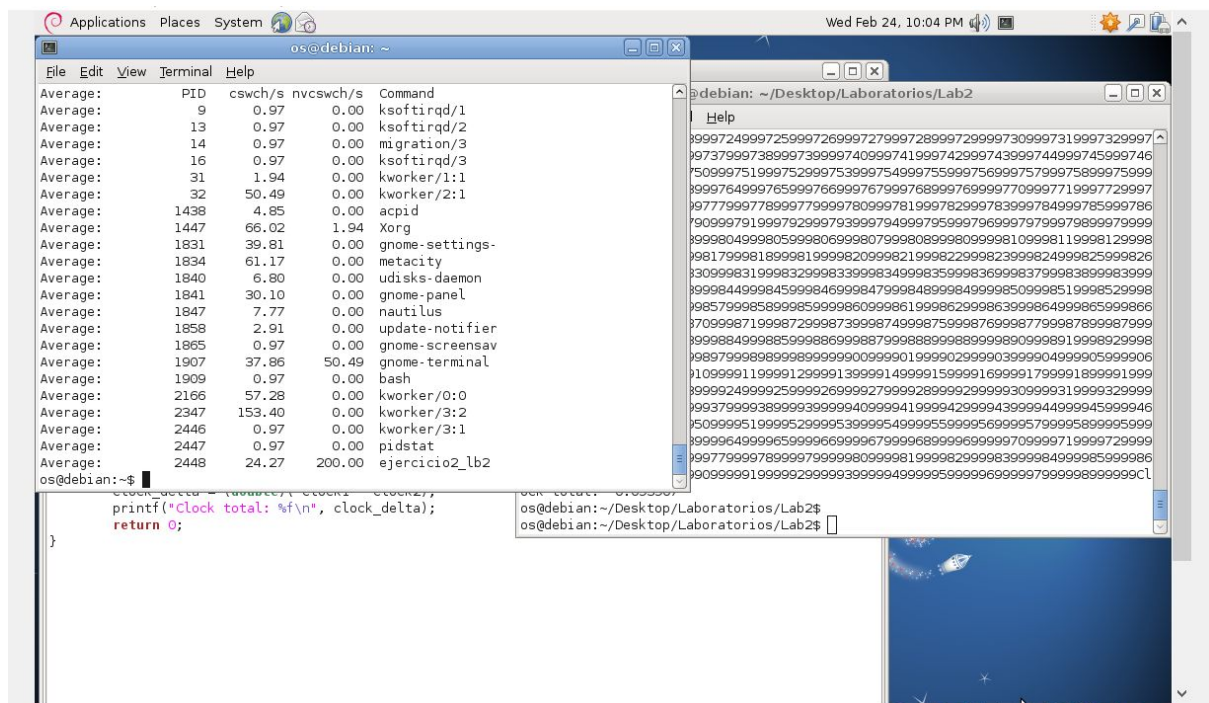


Figura #5: Ejecutable que muestra el tiempo de ejecución del programa sin `fork()`.

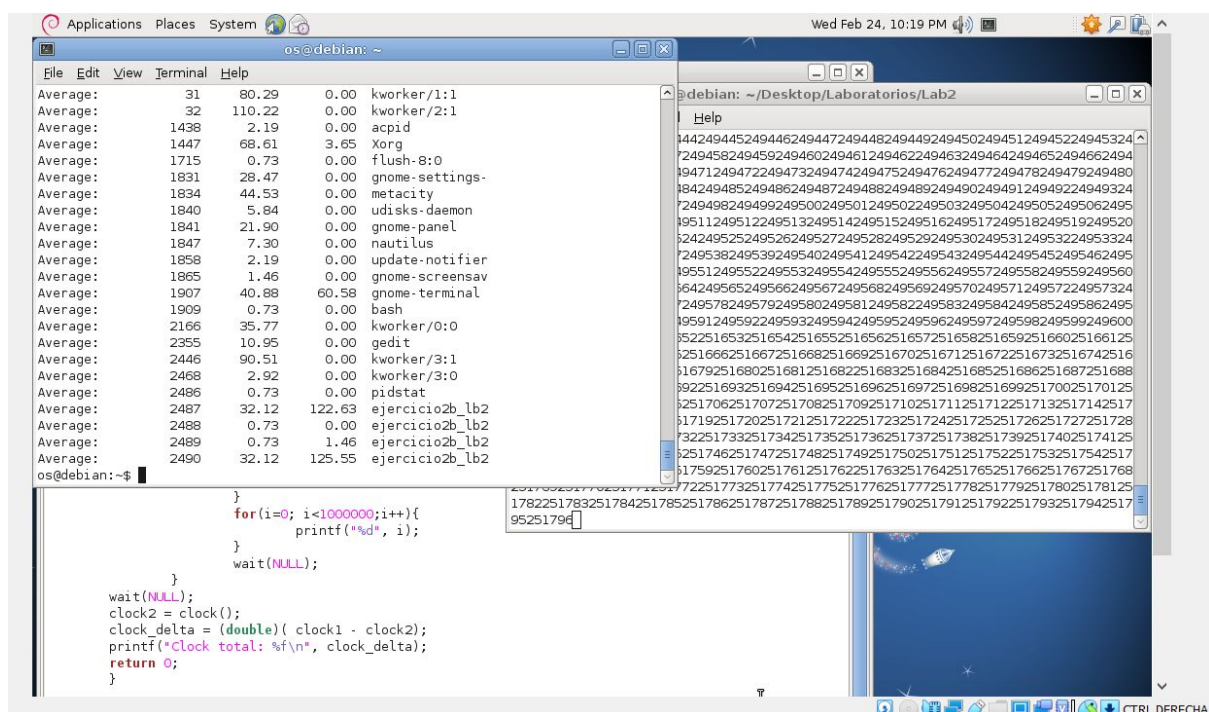


Figura #6: Ejecutable que muestra el tiempo de ejecución del programa con `fork()`.

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

La diferencia entre cada programa es que en el programa del que contiene forks, es ejecutable 4 veces o más, dependiendo de la jerarquía de los hijos, nietos y bisnietos. Podemos verlo de una manera que está mitigando el tiempo de ejecución total del programa, comparado con el programa sin fork.

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

Para que el procesador sea eficiente, provocó estos cambios voluntarios por medio de la terminal, donde fueron ejecutados los programas en la segunda terminal.

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

¿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?

Porque cada vez que se creó un fork, se crea otro proceso del ejecutable. Algunos que tienen cero cambios de contexto, son algunos que están en espera a que terminen la ejecución de sus hijos.

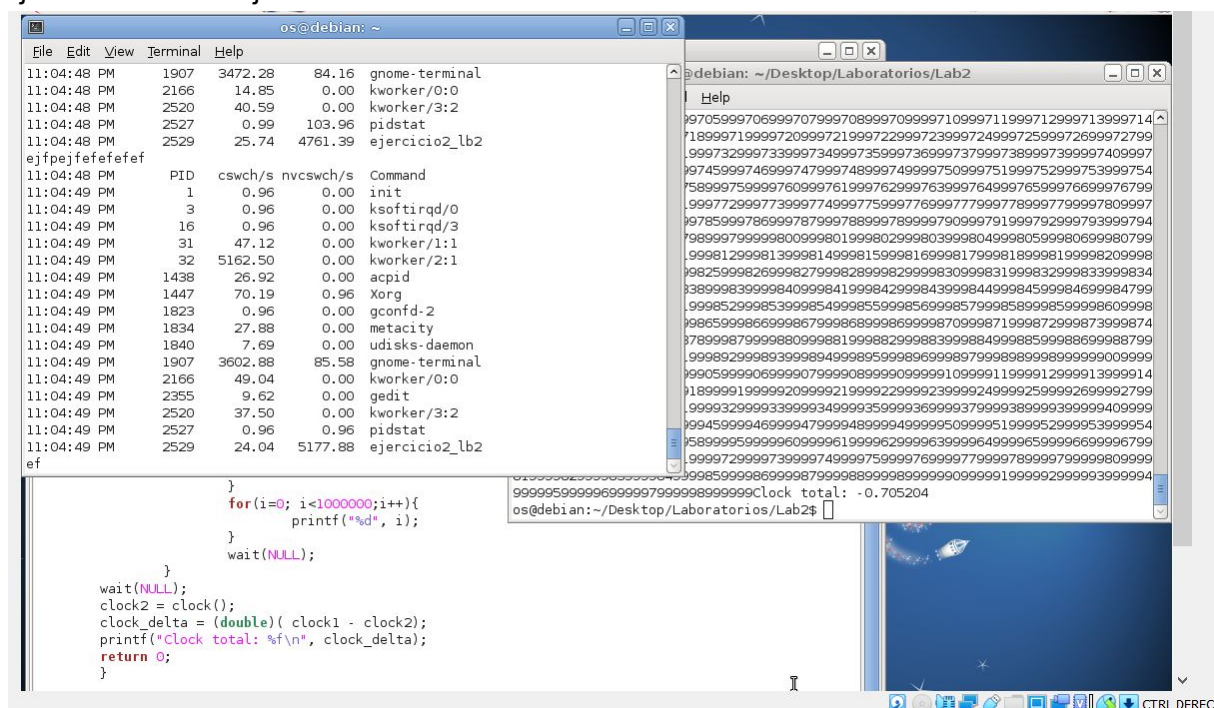


Figura #7: Última parte del ejercicio #3

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

Se tuvo que actualizar otros campos o procesos, como uno en el que se guarde la razón por la que se abandona el estado de ejecución y otros con información de contabilidad, lo cual debe cambiar el estado del proceso a alguno de los otros estados, (listo o bloqueado), que fueron aumentados durante la ejecución del programa más el contacto de la terminal .

## Ejercicio 4 (10 puntos):

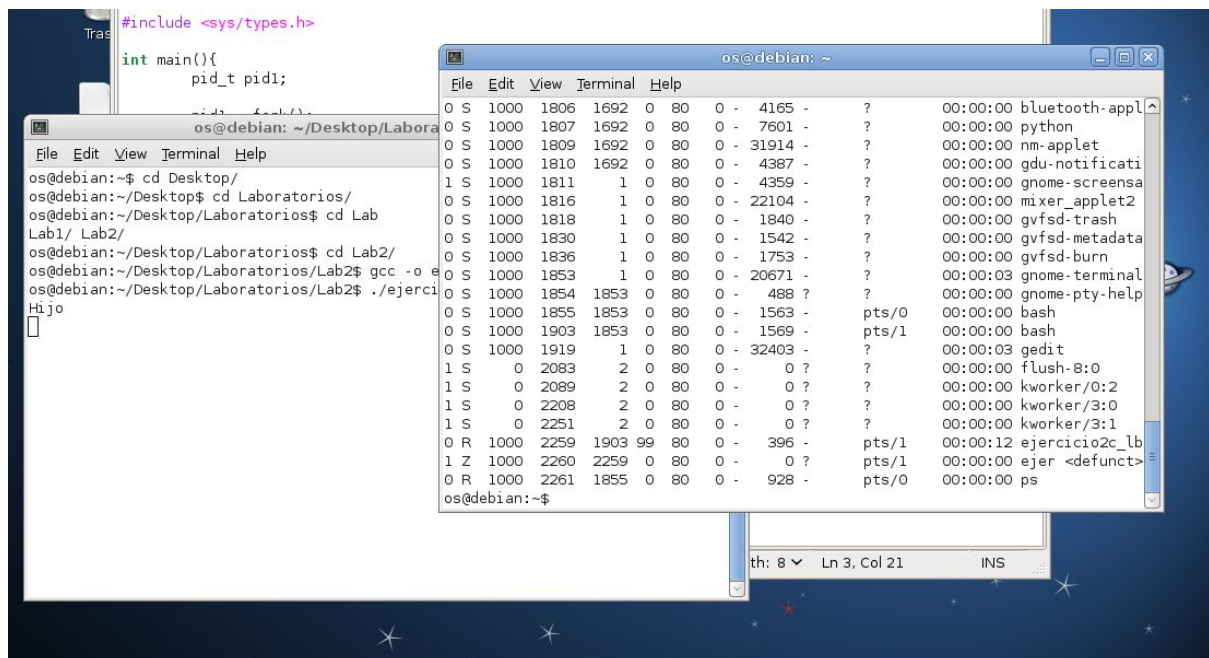


Figura #8: Ejecución del while y inicio de procesos zombies

¿Qué significa la Z y a qué se debe?

Z es denominado un proceso zombie. Es un proceso que ha completado la ejecución pero todavía tiene una entrada en la tabla de procesos: es un proceso en el "estado terminado".

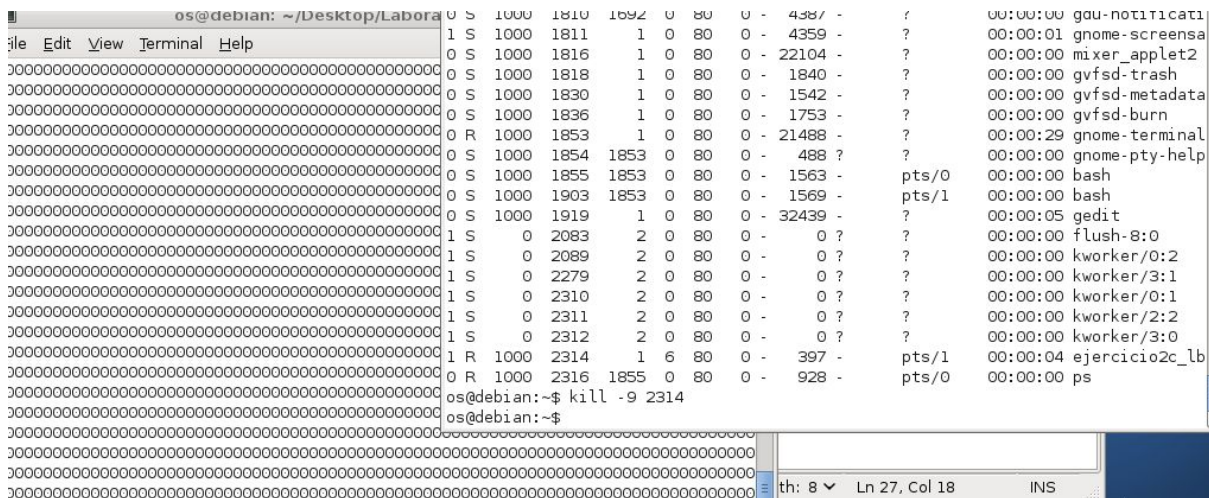


Figura #9: Eliminando el proceso padre.

¿Qué sucede en la ventana donde ejecutó su programa?

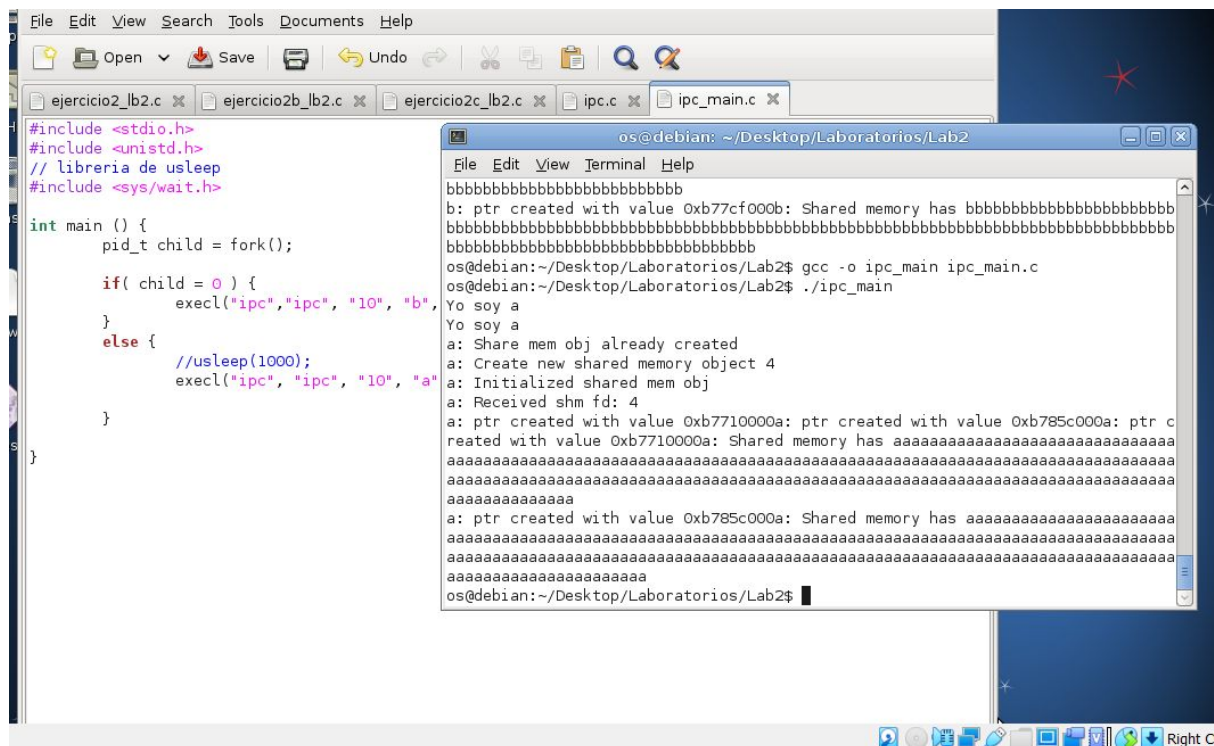
El hijo sigue con su ejecución, ya que, a pesar de eliminar el principal, los hijos son otros procesos creados a partir del padre, y seguirán procesando.

¿Quién es el padre del proceso que quedó huérfano?

El padre del hijo huérfano es el init, que tiene PID 1. Estos procesos no quedan como zombis, debido a que init espera que terminen su ejecución final.



## Ejercicio 5 (40 puntos):



```
#include <stdio.h>
#include <unistd.h>
// libreria de usleep
#include <sys/wait.h>

int main () {
    pid_t child = fork();

    if( child == 0 ) {
        execl("ipc", "ipc", "10", "b",
        )
    }
    else {
        //usleep(1000);
        execl("ipc", "ipc", "10", "a"
        )
    }
}

os@debian: ~/Desktop/Laboratorios/Lab2
File Edit View Terminal Help
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
b: ptr created with value 0xb77cf00b: Shared memory has bbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
os@debian:~/Desktop/Laboratorios/Lab2$ gcc -o ipc_main ipc_main.c
os@debian:~/Desktop/Laboratorios/Lab2$ ./ipc_main
Yo soy a
Yo soy a
a: Share mem obj already created
a: Create new shared memory object 4
a: Initialized shared mem obj
a: Received shm fd: 4
a: ptr created with value 0xb771000a: ptr created with value 0xb785c00a: ptr c
reated with value 0xb771000a: Shared memory has aaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaa
a: ptr created with value 0xb785c00a: Shared memory has aaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaa
os@debian:~/Desktop/Laboratorios/Lab2$
```

**Figura #10:** Ejecución del último ejercicio.

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

La velocidad de comunicación entre la memoria es más rápida. Sin embargo, es bien sensible al momento de utilizar este método por la vulnerabilidad que hay al manejar la memoria por medio del código.

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

Al tener varios file descriptors creados en diferentes procesos podrían acceder una misma instancia de datos de un archivo a la memoria. Esto sería vulnerable debido a que estarían haciendo el mismo procedimiento, y por hacerlo en dos procesos, podría tener una incorrección de extraer el archivo a memoria.

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

Si es posible. Los files descriptors permanecen abiertos en todo el proceso hijo y también en el proceso padre. Si llamamos a fork() después de crear un pipe, entonces el padre y el niño pueden comunicarse a través del pipe.



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

```
char *ptr;
// Memory Object en esta linea
shm_fd = shm_open(NAME, O_EXCL | O_CREAT | O_RDWR, 0666);
if (shm_fd == -1){
    fprintf(stderr, "%s: Shared memory not possible to create for memory object: %s.\n", NAME, strerror(errno));
    return EXIT_FAILURE;
}
```

En este caso, podemos asegurarnos por medio de una librería `errno`, de que, El valor en `errno` es significativo solo cuando el valor de retorno de la llamada indicó un error (es decir, -1 de la mayoría de las llamadas al sistema; -1 o NULL de la mayoría de las funciones de la biblioteca), y eso pasaría al momento de crear una memoria compartida que ya existe.

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

Dejaría de manejar el objeto de memoria, pero sus procesos todavía serían ejecutados.

¿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.

Los comandos de `&` e `*`, representan la dirección y el valor de los registros de memoria respectivamente.

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”?

Sencillamente, otro programa que haga la desconexión a la memoria compartida, la cual es el `shm_unlink`.

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`.

Esto dependerá mucho cuánto velocidad contiene los procesadores, pero, en general, un `usleep` de (1000), soluciona para la mayoría de computadores que cuidan las instancias al momento de ejecutar procesos en la memoria compartida.