**LABORATORIO 2**

**Ejercicio 1.**

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

Se crean 16 procesos.

Graphical user interface, text

Description automatically generated

**Figura 1.** Creación de procesos mediante 4 forks consecutivos.

Text

Description automatically generated

**Figura 2.** Llamadas a sistema forks mediante ciclo iterativo for.

* **¿Por qué hay tantos procesos en ambos programas cuando uno tiene 4 llamadas fork() y el otro sólo tiene una?**

La llamada a sistema fork crea un nuevo proceso por cada uno de los procesos que actualmente están en el programa, entonces, por ejemplo, al inicio está un proceso A, y este realiza la llamada a sistema fork() por lo que se crea un nuevo proceso B. Si ahora se hace nuevamente una llamada a sistema fork(), ahora se crearán 2 procesos nuevos, 1 por el proceso A y otro por el proceso B. En general, se sabe que la cantidad de procesos creados por la llamada a sistema fork() es 2^n – 1, donde n es la cantidad de llamadas a sistema que se hacen. (GeeksforGeeks, 2019b)

Por otro lado, en cuanto al programa que cuenta con un solo fork, este, debido que se encuentra en un ciclo for, lo que indica es que se realizará la llamada la cantidad de veces que el ciclo se ejecute, en este caso particular 4. Por ende, se realizan 4 llamadas forks, representando el mismo caso del programa anterior.

Ejercicio 2.

**Text

Description automatically generated**

**Figura 3.** Ejecución de 3 ciclos for con un solo proceso.

**Text

Description automatically generated**

**Figura 4.** Ejecución de 3 for vacíos por medio de llamadas a sistema fork.

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

Como se puede observar en las **figuras 3 y 4** en general, el programa que más tiempo de ejecución toma es aquel en el cual no se hacen llamadas a sistema fork.

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

Como se mencionaba en las instrucciones del laboratorio, en el que se hacen las llamadas a sistema fork, los for se están ejecutando concurrentemente, esto quiere decir que existen 4 procesos que están ejecutando cada uno parte del programa completo, su for. A comparación del programa sin forks el cual solamente un proceso se encarga de la ejecución de este. Sin embargo, algo a tomar en consideración es que para que la tarea del proceso nieto se considere como completada, debido a que se está utilizando wait, sí se debe de esperar a que el proceso bisnieto termine de ejecutar el for.

Debido a esto, como se puede observar en la **figura 4** los tiempos de ejecución se reducen significativamente, ya que a diferencia de la **figura 3** en la cual únicamente había 1 solo proceso ejecutando de manera secuencial los fors, cuando se realizan las 3 llamadas a sistema forks, ahora se cuentan con 4 procesos ejecutándose concurrentemente.

Ejercicio 3.

* **Cambios de contexto:** se conoce como cambio de contexto, a guardar el estado de un proceso en ejecución para que luego pueda ser restaurado y continuar la ejecución en el punto que había quedado. En general, se produce cuando el sistema operativo asigna el procesador a un nuevo proceso. Por otro lado, permite que los sistemas con un solo CPU puedan ser compartido entre múltiples procesos. (Duke Courses, s. f.)
  + **Voluntarios:** un proceso explícitamente cede el CPU a otro. En este caso, es el proceso quien realiza la llamada a sistema. (*What does «high involuntary context» switches mean?*, 2014)
  + **Involuntarios:** es el sistema operativo el cual hace el cambio de contexto. El *scheluder* del sistema detiene la ejecución de un *thread* activo y cede el control del CPU a otro proceso. (Duke Courses, s. f.)

**Graphical user interface

Description automatically generated**

**Figura 5.** Ejecución de pidstat y comparación de acciones sobre la interfaz en el proceso Xorg.

**Graphical user interface

Description automatically generated**

**Figura 6.** Ejecución de pidstat cuando se realizan entradas de teclado.

* **¿Qué tipos de cambio de contexto incrementa notablemente en cada caso y por qué?**

En el caso de realizar acciones en la interfaz gráfica, los cambios de contexto que incrementaban significativamente eran los voluntarios, como se puede observar en la **figura 5**, en el caso del proceso xorg 1seg antes de mover la ventana de la terminal, este realizaba 336 cambios de contexto voluntarios, y luego de que se movió la ventana se realizaron 1364 cambios de contexto.

Esto se debe a que, por ejemplo, en el caso particular de la tarea xorg, esta se encarga de proporcionar el entorno gráfico, por lo tal, en el caso de mover mover la ventana de la terminal, este necesita esperar a que termine de mover la terminal para luego nuevamente desplegar la información en la nueva posición de la ventana. Por ello, debido a que el proceso está esperando a que se termine de mover la pantalla se bloquea realizando un cambio de contexto voluntario.

Por otro lado, al momento de realizar input por teclado se puede observar que el tipo de cambio de contexto que se incrementan más notablemente son los involuntarios, ya que, en el caso de gnome-shell antes de la entrada de teclado realizaba 199 cambios de contexto y luego de escribir en terminal realizó 357. Esto ocurre ya que en ese caso el proceso no está esperando un recurso, sino más bien, se realiza la entrada desde el teclado, esta es detectada y el sistema operativo realiza el cambio de contexto para atender lo que se escribió.

Graphical user interface

Description automatically generatedGraphical user interface

Description automatically generated

**Figura 7.** Cambios de contexto realizados por el programa sin llamada a sistema fork.

Graphical user interface

Description automatically generatedGraphical user interface

Description automatically generated with medium confidence

**Figura 8.** Cambios de contexto realizados por el programa con 3 llamadas a sistema fork.

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

Como se puede notar en las figuras anteriores en el caso del programa que trabaja con un único proceso, este realiza una mayor cantidad de cambios involuntarios, existiendo una diferencia entre los cambios voluntarios e involuntarios de 287,458 cambios de contexto.

En el caso del programa con 4 procesos, si bien, también existe una diferencia significativa entre los cambios voluntarios e involuntarios, predominando los involuntarios, la diferencia es menor comparado a cuando solo existe un proceso.

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

Los cambios involuntarios pueden ser atribuidos a los wait realizados en los procesos. Wait en este caso, espera a que el proceso hijo finalice, pero, por ejemplo, si el proceso padre ha finalizado de recorrer su for pero el proceso hijo aún no, entonces el proceso padre se bloquea hasta que el proceso hijo finalice, haciendo así cambio de contexto voluntario para que otros procesos puedan usar el CPU.

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

Los cambios de contexto involuntarios son aquellos que realiza el sistema operativo porque el proceso se ha estado ejecutando por cierto tiempo sin realizar un cambio de contexto voluntario y existen procesos en colar esperando por el CPU. En este caso, se puede tomar de ejemplo al proceso bisnieto, este empieza a ejecutar el for, y debido a que son varias iteraciones este se ejecuta por un largo tiempo, sin embargo, en la espera también están los procesos nietos e hijos, entonces para empezar a ejecutar los for dentro de estos procesos el sistema operativo hace un cambio de contexto involuntario dejando que, por ejemplo, el proceso nieto ejecute una parte de su for. Y así sucesivamente hasta que todos los procesos hayan finalizado de ejecutar cada uno su for.

* **¿Por qué el reporte de cambios de contexto para su programa con forks se muestra cuatro procesos, uno de los cuales reporta cero cambios de contexto?**

El reporte de cambios de contexto para el programa con forks muestra 4 procesos, debido a que se hacen 3 llamadas a fork dentro del programa lo que crea un proceso por cada fork, sin embargo, hay que considerar que cada una de estas llamadas solo es realizada por los procesos hijos.

Y en el caso del proceso el cual reporta 0 cambios de contexto, en este caso es el proceso padre, el cual dentro del programa no se encuentra realizando ninguna operación mientras el resto de los procesos está ejecutando los for, sino más bien, espera a que todos estos terminen. Debido a que para empezarse a ejecutar el proceso padre el resto de los procesos ya tuvo que haber finalizado entonces el sistema operativo no cuenta con procesos en cola a los cuales deba de darle recursos de CPU, evitando así los cambios de contexto involuntarios.

Graphical user interface, text

Description automatically generated

**Figura 9.** Cambios de contexto realizados mientras la ejecución de un programa y la intervención al escribir en un editor de texto.

* **¿Qué efecto percibe sobre el número de cambio de contextos de cada tipo?**

Al observar la **figura 9** se encuentra que luego de tener interacción en el editor de texto la cantidad de cambios de contexto involuntarios del programa se redujeron a diferencia de los cambios de contexto voluntarios los cuales incrementaron. Por otro lado, si bien se puede observar, gedit, el editor de texto, ya se encontraba abierto al momento de hacer el reporte de los cambios de contexto, sin embargo, cuando se escribe en el editor, se incrementan los cambios de contexto, tanto los voluntarios como involuntarios

Ejercicio 4.

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

A picture containing graphical user interface

Description automatically generatedGraphical user interface

Description automatically generated with low confidence

**Figura 10.** Informe de los procesos activos.

La segunda columna identificada con S indica el estado del proceso, en este caso Z se define para procesos “zombie” en los cuales el proceso ha finalizado, pero no ha sido manejado o eliminado por el proceso padre, en inglés eso se conoce como *reaping*.

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

Graphical user interface, text

Description automatically generated

**Figura 11.** Terminal en la que se ejecuta el programa luego de haber eliminado el proceso padre.

En la ventana en la cual se ejecuta el programa no se percibe ningún cambio, dicho programa sigue ejecutándose como si nada hubiera pasado e incluso al final, debido a que el programa cuenta con un while infinito este no termina de ejecutarse. Básicamente no se percibe que se haya eliminado el proceso padre.

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

Text

Description automatically generated**Figura 12.** Informe de procesos activos luego de eliminar el proceso padre.

Como se observar en la **figura 12** ahora el proceso padre del proceso que quedó ejecutando el programa es el proceso con id 1, conocido como systemd. Es el primer proceso que se ejecuta y actúa como el inicializador del sistema creando los servicios de modo usuario.

Ejercicio 5.

* **main(int argc, char\*\* argv):** argc y argv son argumentos de líneas de comando que se procesan en un programa, **argc** será la cantidad de argumentos que se pasan al programa mediante la línea de comandos y **argv** será el arreglo de argumentos.
* **¿Qué diferencia hay entre realizar comunicación usando memoria compartida en lugar de usar un archivo de texto común y corriente?**

Al utilizar como método de comunicación entre procesos un archivo común de texto es importante destacar que entre los procesos no se está compartiendo memoria, y tampoco se obtienen los beneficios en cuanto a *performance* de la memoria compartida. (Stephen, 2020)

Al utilizar archivo de texto común es necesario explícitamente utilizar llamadas a sistema para enviar y recibir mensajes entre procesos. Esto implica realizar una mayor cantidad de llamadas a sistema a diferencia de usar memoria compartida y también de que el kernel copie información. Por otro lado, para tener acceso a la información escrita dentro de un archivo, debido a que este proceso no se hace inmediato, es necesario “cargar” nuevamente el archivo para percibir los cambios en el mismo. Además, si se cuenta con más de un proceso que esté escribiendo será necesario establecer un “lock” para mantener la integridad de información. (Stephen, 2020)

Por otro lado, en el caso de usar memoria compartida no es necesario utilizar llamadas a sistema de lectura o escritura y tampoco copiar la información desde el kernel, además, en este caso, mientras que un proceso escribe en el espacio de memoria y casi inmediatamente visible para el otro proceso. Se podría decir que el proceso de escritura/lectura se realiza en tiempo real. (Stephen, 2020)

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

No se utiliza el *file descriptor* producido por otra instancia debido a que dicho *file descriptor* está asociado al proceso que lo instancia. En particular, se puede asegurar que es el *file descriptor* con el número más bajo no abierto previamente dentro del proceso. Por lo tal, si se utiliza el *file descriptor* producido por otra instancia puede generar error al momento de hacer el mapeo, ya que dicho *file descriptor* puede no ser el esperado, es decir, el más bajo de los previamente abiertos dentro del proceso, y por ende, podría existir otro *file descriptor* igual en los ya abiertos dentro del proceso.

* **¿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)**

Para poder redireccionar el output de un programa ejecutado por medio de exec se puede utilizar dup2. El cual duplica un *file descriptor* que permite redirigir el output del programa ejecutado. Básicamente como funciona es que redirige el output a un archivo y algo interesante a destacar es que en el caso de que se utilice con forks y execs el proceso hijo podrá redirigir el output a un archivo incluso después de que termine de ejecutarse exec. (Song, s. f.)

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

Para poder validar si el espacio de memoria ya ha sido creado, *shm\_open* cuenta con diferentes *flags* que se pueden agregar al momento de crear el objeto de memoria. En particular existe O\_EXCL el cual, en conjunto con O\_CREAT valida si el objeto de memoria compartida ya ha sido creado retorna valor de -1 con y configura *errno* para indicar cuál fue el error.

*Errno* es la abreviatura de *error number*, básicamente es el número del último error ocurrido en el sistema. Cuando dentro de un sistema se detecta un error, un valor es almacenado en errno y dependiendo de dicho valor especifica qué sucedió en el programa. Para el caso particular de *shm\_open* en caso de que un objeto de memoria compartida con un nombre determinado ya ha sido creado, retorna valor de -1 y al obtener el valor de errno EEXIST el cual indica que un archivo ya existe.

Por lo mencionado anteriormente, para poder validar que un espacio de memoria ya ha sido abierto, al momento de crearlo se utilizan las *flags* O\_CREATE | OEXCL, entonces al momento de que un proceso intente crear dicho espacio nuevamente *shm\_open* retornará error y solo falta validar si el código errno corresponde a EEXIST.

* **¿Qué pasa si se ejecuta shm\_unlink cuando hay procesos que todavía están usando la memoria compartida?**

*Shm\_unlink* se encarga de eliminar la asociación de un nombre con un objeto de memoria compartida. Por lo tal, realmente depende del lugar en el cual se ejecute *shm\_unlink* dentro del programa. Por ejemplo, si este se ejecuta luego de crear el objeto de memoria compartida, luego de *shm\_open*, o en general, si se realiza antes de hacer el mapeo al objeto de memoria, esto lo que provocará es que se trabajen con dos objetos de memoria completamente diferentes. Si, por otro lado, se ejecuta *shm\_unlink­* luego de hacer el mapeo esto no tendrá ningún efecto en cuanto al objeto de memoria compartida en el cual trabajaran los procesos ya que el file descriptor se creó con el nombre aún vinculado al objeto.

* **¿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 sobre escribir sobre ello.**

Un puntero es una variable estática que almacena una dirección de memoria no un carácter o número, por lo tal es necesario utilizar un operador que permita acceder a la información en ese espacio de memoria. El operador “\*” retorna el contenido en una posición de memoria y generalmente se le conoce como operador de indirección. Para utilizarlo, simplemente se coloca antes del nombre del puntero, por ejemplo:

char \*pc; → define un puntero de tipo carácter

Para acceder a la información en la dirección de memoria almacenada por p solo se colocar el operador de indirección antes de la variable.

\*p → contenido en la dirección de memoria almacenada por p

* **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 liberar espacio de memoria compartida de forma manual existe el comando *ipcrm* el cual se encarga de eliminar ciertos recursos de objetos IPC, dichos objetos pueden ser memoria compartida, cola de mensajes o semáforos. La eliminación del objeto de memoria compartida se realiza hasta que todos los procesos mapeados hayan desvinculado el objeto de su espacio de direcciones virtuales. Se puede utilizar con la opción *-M* la cual elimina el segmento de memoria creado *shmkey* o también se puede usar *-m* para eliminar segmentos de memoria identificados por *shmid*.

Por otro lado, algo interesante es que si se desconoce la clave o el identificador del segmento de memoria compartida este se puede obtener por medio de *ipcs* el cual muestra la información de los recursos de comunicación entre procesos. Para específicamente obtener información sobre los segmentos de memoria activos se puede utilizar la *flag* -m.

* **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.**

Text

Description automatically generated

**Figura 13.** Ejemplo de una instancia finalizando mucho antes de la otra provocando que ambas instancias crearan su propio espacio de memoria.

En general se sabe que fork es bastante rápido, comúnmente menos de un milisegundo y en ocasiones hasta puede durar docenas de microsegundos. Usleep es una función que suspende la ejecución del proceso que realiza la llamada por al menos usec microsegundos, donde usec es la cantidad de microsegundos que se desean

**Graphical user interface, text

Description automatically generated**

**Figure 14.** Ejemplo de 3 ejecuciones de un programa que ejecuta 2 instancias de un programa que implementa memoria compartida.

**Referencias**

A. (2021, 17 diciembre). *¿Qué es el proceso XORG en Ubuntu?* CompuHoy.com. https://www.compuhoy.com/que-es-el-proceso-xorg-en-ubuntu/

BlackBerry. (s. f.). *Embedded OS, Support and Services | RTOS, Hypervisor | BlackBerry QNX*. Qnx.Com. https://www.qnx.com/index.html

Blakelead. (2015, 10 septiembre). *Warning comparison between pointer and integer*. Stack Overflow. https://stackoverflow.com/questions/32510218/warning-comparison-between-pointer-and-integer

CodeVault. (2020, 4 junio). *How to send a string through a pipe* [Vídeo]. YouTube. https://www.youtube.com/watch?v=ZS3qxIjZ0z0&list=PLfqABt5AS4FkW5mOn2Tn9ZZLLDwA3kZUY&index=23

Duke Courses. (s. f.). *Context Switches: Voluntary and Involuntary*. Courses. https://courses.cs.duke.edu/spring01/cps110/slides/interleave/tsld008.htm

GeeksforGeeks. (2019a, enero 31). *Communication between two process using signals in C*. https://www.geeksforgeeks.org/signals-c-set-2/?ref=lbp

GeeksforGeeks. (2019b, diciembre 9). *fork() in C*. https://www.geeksforgeeks.org/fork-system-call/

GeeksforGeeks. (2021, 9 julio). *POSIX shared-memory API*. https://www.geeksforgeeks.org/posix-shared-memory-api/

GeeksforGeeks. (2022, 3 enero). *C Program to Demonstrate fork() and pipe()*. https://www.geeksforgeeks.org/c-program-demonstrate-fork-and-pipe/?ref=lbp

IBM. (s. f.). *sigaction()--Examine and Change Signal Action*. Ibm.Com. https://www.ibm.com/docs/en/i/7.2?topic=ssw\_ibm\_i\_72/apis/sigactn.htm

*IPC:Shared Memory*. (s. f.). users.cs. https://users.cs.cf.ac.uk/dave/C/node27.html

Linux Manual. (s. f.-a). *errno(3) - Linux manual page*. Man7.Org. https://man7.org/linux/man-pages/man3/errno.3.html

Linux Manual. (s. f.-b). *mmap(2) - Linux manual page*. Man7.Org. https://man7.org/linux/man-pages/man2/mmap.2.html

Linux Manual. (s. f.-c). *pidstat(1) - Linux manual page*. Man7.Org. https://man7.org/linux/man-pages/man1/pidstat.1.html

Linux Manual. (s. f.-d). *ps(1) - Linux manual page*. Man7.Org. https://man7.org/linux/man-pages/man1/ps.1.html

Linux Manual. (s. f.-e). *ps(1) - Linux manual page*. Man7.Org. https://man7.org/linux/man-pages/man1/ps.1.html

Linux Manual. (s. f.-f). *usleep(3) - Linux manual page*. Man7.Org. https://man7.org/linux/man-pages/man3/usleep.3.html

L.M. (s. f.). *ipcs(1) - Linux manual page*. Man7.Org. https://man7.org/linux/man-pages/man1/ipcs.1.html

Man 7. (s. f.). *shm\_open(3) - Linux manual page*. Man7.Org. https://man7.org/linux/man-pages/man3/shm\_open.3.html

man7. (s. f.). *systemd(1) - Linux manual page*. Man7.Org. https://man7.org/linux/man-pages/man1/systemd.1.html

Song, K. (s. f.). *Using dup2() to redirect output*. Â. https://www.cs.utexas.edu/%7Etheksong/2020/243/Using-dup2-to-redirect-output/

Stephen, C. (2020, 22 mayo). *Using files for for shared memory IPC*. Stack Overflow. https://stackoverflow.com/questions/61962399/using-files-for-for-shared-memory-ipc

Universidad Carlos III de Madrid. (s. f.). *Introducción a la gestión de procesos*. ucm. http://ocw.uc3m.es/ingenieria-informatica/sistemas-operativos/material-de-clase-1/mt\_t2\_l3.pdf

*What does «high involuntary context» switches mean?* (2014, 14 febrero). Stack Overflow. https://stackoverflow.com/questions/21777430/what-does-high-involuntary-context-switches-mean

*What does int argc, char \*argv[] mean?* (2010, 11 junio). Stack Overflow. https://stackoverflow.com/questions/3024197/what-does-int-argc-char-argv-mean

*What does reaping children imply?* (2019, 15 noviembre). Stack Overflow. https://stackoverflow.com/questions/58885831/what-does-reaping-children-imply