Universidad del Valle de Guatemala
Sistemas Operativos
Docentes: Erick Pineda; Tomás Gálvez P.
Semestre I, 2021

Laboratorio #2

Fecha de Entrega: 26 de febrero, 2021.

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

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.

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.
13. Compare los resultados de tiempos de cada uno de sus programas, y responda:
    - ¿Cuál, en general, toma tiempos más largos?
    - ¿Qué causa la diferencia de tiempo, o por qué se tarda más el que se tarda más?

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 diríjase en ella al directorio donde estén los programas que hizo en el ejercicio anterior.
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
```

6. Realice acciones en la interfaz gráfica (*e.g.*, mueva la ventana donde ejecutó `pidstat` o abra una ventana nueva) y observe el efecto que esto tiene sobre el proceso `Xorg`.

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

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

7. 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.
8. Cancele la ejecución de `pidstat` con `Ctrl-C`. Escriba (pero no ejecute) en la primera terminal el siguiente comando:

```
pidstat -w X 1
```

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

9. Escriba (pero no ejecute) en la segunda terminal el comando para ejecución del programa sin `fork()`s.
10. Ejecute primero el `pidstat` y luego, lo más inmediatamente posible, el programa sin `fork()`s, y espere a que este último termine.
11. Anote el número de cambios de contexto de cada tipo para el proceso correspondiente a la ejecución de su programa. Si no percibe resultados, reste 1 a `X` en `pidstat` e intente de nuevo.
12. Repita los pasos anteriores para el programa con `fork()`s que realizó en el ejercicio anterior.
    - ¿Qué diferencia hay en el número y tipo de cambios de contexto de entre programas?
    - ¿A qué puede atribuir los cambios de contexto voluntarios realizados por sus programas?
    - ¿A qué puede atribuir 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?
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?

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 programa escrito en el inciso anterior en una de las terminales. En la otra ejecute el siguiente comando:

```
ps -ael
```

   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?
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.
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?
7. Vuelva a ejecutar `ps -ael`.
   - ¿Quién es el padre del proceso que quedó huérfano?

Ejercicio 5 (40 puntos)

Escriba un programa en C llamado `ipc.c` que reciba desde terminal un número que llamaremos n y una letra que llamaremos x (investigue sobre `int argc, char** argv`). **Dos instancias de este programa se ejecutarán concurrentemente**.

El código de su programa deberá abrir un espacio de memoria compartida y luego enviar el *file descriptor* de este espacio a la otra instancia de sí mismo. Si el espacio de memoria compartida ya fue abierto por la otra instancia deberá detectarlo, recibir el *file descriptor* de su otra instancia y luego desplegarlo en pantalla. NO debe usar este *file descriptor* para *mappear* la memoria compartida.

El programa, sin importar que haya creado el espacio de memoria compartida o que se haya adherido a un espacio existente, eventualmente *mappeará* la memoria compartida a un puntero. Luego del *mappeo*, deberá engendrar un proceso usando `fork()`.

El proceso padre iniciará un ciclo `for` con un número de iteraciones igual al tamaño asignado a la memoria compartida. En el ciclo se revisará si el número de iteración es divisible entre n, en cuyo caso se enviará x al proceso hijo, "notificándole" que pasó por una iteración divisible entre n. El proceso hijo deberá estar pendiente de la comunicación del proceso padre y deberá escribir cada x recibida en la memoria compartida.

Una vez concluido el ciclo `for`, el proceso padre esperará al hijo mientras éste escribe, luego desplegará el contenido de la memoria compartida y, por último, cerrará las comunicaciones. **Nota**: se calificará programación defensiva contra posibles errores en el manejo de `fork()`s, *pipes*, etc., y que sus programas tomen las medidas adecuadas en el manejo de sus comunicaciones y archivos (cerrar archivos abiertos, quitar el *mappeo* de memoria compartida, etc.).

Finalmente, escriba un segundo programa que únicamente realice un `fork()`, y que en cada una de sus ramas ejecute, con `execl`, el programa `ipc` con diferentes símbolos para x. El resultado debe ser que se despliegue el contenido de memoria compartida haciendo evidente que ambas instancias de `ipc` han escrito en ella.

A continuación, un ejemplo de cómo podría verse el *output* de su último programa, que ejecuta dos instancias de `ipc`:

```
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: bbbbbbbbbbbbbbbbbbbbbbbbbb
os@debian:~$
a: Shared memory has: bbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaaaa
```

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?
- ¿Por qué no se debe usar el *file descriptor* de la memoria compartida producido por otra instancia para realizar el `mmap`?
- ¿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`).
- ¿Cómo puede asegurarse de que ya se ha abierto un espacio de memoria compartida con un nombre determinado? Investigue y explique ***errno***.
- ¿Qué pasa si se ejecuta `shm_unlink` cuando hay procesos que todavía están usando la memoria compartida?
- ¿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.
- 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"?
- 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`.