**Laboratorio #2**

**Juan Pablo Pineda 19087**

# Ejercicio 1 (10 puntos)

Si no está apagada, apague su máquina virtual y revise que este usando más de un procesador (recomendable, cuatro) en el menú de configuraci6n 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 () consecutivos. Luego cree otro programa en C que ejecute fork () dentro de un ciclo for de cuatro iteraciones.

* cuantos procesos se crean en cada uno de los programas?
  + Tomando en cuenta que cada una de las llamadas fork() crea dos procesos de uno solo, sabemos que su forma de crecimiento es por lo que se crean 16 procesos.  
    Sabiendo lo mismo, la forma del crecimiento de fork dentro de un ciclo for seria
* por qué hay tantos procesos en ambos programas cuando uno tiene cuatro llamadas fork () y el otro solo tiene una?
  + Por el funcionamiento de la llamada, esta crea dos procesos de cada uno de los procesos individuales.

# 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 esta antes se resta a la que esta después). Tome en cuenta que, para 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.

1. Este programa hará lo mismo que en el primero, pero de forma concurrente (recuerde que esto no es sin6nimo de "paralelo"). Para lograrlo comience para 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.
2. Haga que el proceso hijo realice otro fork(), y que este nuevo proceso (seria el proceso nieto) haga también un nuevo fork ().
3. El proceso bisnieto (el creado por el fork () más anidado) debe realizar un ciclo for de un mill6n 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 e l s e del i f 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).
4. El proceso hijo debe realizar lo mismo, y en las mismas condiciones, que el proceso nieto en el inciso anterior.
5. En el proceso raíz o padre exclusivamente (es decir, en el e l s e menos anidado) espere a que termine la ejecuci6n del proceso hijo y luego ejecute clock (). Despliegue la diferencia entre las variables clock\_t y deje terminar el programa.
6. Ejecute este programa la misma cantidad de veces que ejecut6 el programa no concurrente.
7. Compare los resultados de tiempos de cada uno de sus programas, y responda:
   * ¿Cual, en general, toma tiempos más largos?
     + El segundo programa toma tiempos más largos en promedio
   * ¿Que causa la diferencia de tiempo, o porque se tarda más el que se tarda más?
     + Porque debe pasar la información por varios procesos o instrucciones antes de dividirse y trabajar.

# 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?

Moviendo la ventana y tecleando, los cambios voluntarios son los que mas incrementan en el proceso Xorg

1. 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.

El primero 2.44703

El segundo 0.69633

1. 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().
2. Escriba (pero no ejecute) en la segunda terminal el comando para ejecución del programa sin fork()s.
3. Ejecute primero el pidstat y luego, lo más inmediatamente posible, el programa sin fork()s, y espere a que este último termine.
4. 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. 9397 voluntarios, 65 involuntarios.
5. 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?   
  Los cambios de contexto en el programa que tiene forks son mucho mayores que el que no tiene forks y no presenta cambios involuntarios mayores a 1
* ¿A qué puede atribuir los cambios de contexto voluntarios realizados por sus programas?   
  a el intercambio de mensajes o información entre cada uno de los procesos
* ¿A qué puede atribuir los cambios de contexto involuntarios realizados por sus programas?   
  a errores inesperados u overhead
* ¿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?

1. 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
2. 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.

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

1. 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.
2. 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.
3. 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 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?
4. Vuelva a ejecutar ps –ael.
   1. ¿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.