TP III – Procesos Livianos - Threads

Objetivo: Desarrollar una serie de programas que permitan comprobar distintas funcionalidades vinculadas con la gestión de procesos livianos y su sincronización, a través de la utilización de la librería pthread (POSIX Threads).

Introducción

Puede acceder on-line a un tutorial sobre pthread y otros temas relacionados en https://computing.llnl.gov/tutorials/parallel_comp/ . Todo el material necesario para realizar esta actividad ha sido previamente bajada de este sitio y otros y esta disponible en CD-ROM de programación entregado en esta asignatura, a partir de la carpeta `/Software/Win32/gcc/docs/pthread`. En la página `Introduction to Parallel Computing.htm` encontrará información relevante acerca de procesos concurrentes y su campo de aplicación; también se menciona la taxonomía de Flynn vista en clase en cuanto a la clasificación de sistemas con procesamiento en paralelo.

Utilice la página `POSIX Threads Programming.htm` para realizar los puntos de esta práctica, luego de haber leído `Introduction to Parallel Computing.htm`.

1. Realice el programa `tp31.c` que permite indicar por línea de comando la cantidad de threads a crear. Cada thread ejecutará una función que mostrará un mensaje en pantalla indicando un número único de thread (que es un simple valor entero pasado como parámetro de la función). Sugerencias: No olvide utilizar las cabeceras `stdio.h, stdlib.h, pthread.h`. Declare a la función `main()` como `int main(int argc,char *argv[])`, si pasa un argumento en la línea de comando, argc vale 2 y en argv[1] se encuentra el argumento en forma de string. Para pasar de string a entero, puede utilizar la función de librería `atoi()`. Compile de la siguiente forma: `gcc -Wall tp31 tp31.c -pthread` y ejecútelo como: `./tp31 5` si pretende que `main()` cree 5 threads.

2. Modifique el programa, agregue el archivo de cabecera `unistd.h`, el cual contiene el acceso a varias llamadas al sistema Unix tales como: `open(), read(), write(), close(), seek()` y `sleep()`. Esta función permite detener el proceso actual la cantidad de segundos que se le indique como argumento; por ejemplo, `sleep(3)` detiene el proceso 3 segundos. Esto coincide con el comando Unix `sleep`, por lo tanto, desde la consola, Ud. puede tipear `sleep 3` y observara que el shell queda detenido durante `3` segundos.
Modifique la función que es ejecutada repetidamente como un thread, para que en medio de dos mensajes que imprima en la consola llame a la función `sleep`, por ejemplo:

```
printf("Mensaje de thread %d!\n", tid);
sleep(3);
printf("fin Mensaje de thread %d!\n", tid);
```

Agregue dos mensajes al final de la función `main()` (antes y después de la llamada a la función `pthread_exit()`, por ejemplo:

```
printf("fin main():1\n");
pthread_exit(NULL);
printf("fin main():2\n");
```

Compile y ejecute el programa, podrá obtener una salida como esta:

```
grchere@debian:~/gccwork/src/thr1$ ./tp32 5
main(): creando thread 0
Mensaje de thread 0!
main(): creando thread 1
Mensaje de thread 1!
main(): creando thread 2
Mensaje de thread 2!
main(): creando thread 3
Mensaje de thread 3!
main(): creando thread 4
Mensaje de thread 4!
fin main():1
fin Mensaje de thread 0!
fin Mensaje de thread 1!
fin Mensaje de thread 2!
fin Mensaje de thread 4!
fin Mensaje de thread 3!
grchere@debian:~/gccwork/src/thr1$
```

Observando los mensajes podemos concluir que la función `main()` termina su ejecución antes que los threads que ella misma dispara, lo cual no parece ser algo saludable o razonable. El proceso que dispara estos threads debería terminar cuando el ultimo de los threads que él mismo disparó haya finalizado. Por lo visto, no tiene sentido escribir sentencias luego de `pthread_exit()`, puesto que éstas nunca se ejecutarían.

3. Copie el programa anterior en `tp33.c`. Modifique el programa para que la función `main()` no termine antes que los threads que ella dispara.
Sugerencia: La operación join hace que el proceso que la llamada quede bloqueado esperando hasta que el thread que se le indique a la función join termine. Modifique la función `main()` para que ésta haga un join a cada uno de los threads que ejecuta, de manera tal, que termine cuando ya hayan terminado todos los threads que el lanzó. Utilice la función `pthread_join()`. Declare a los threads como "joineables" (crear atributo, indicarlo como joineable y luego asociarlo a cada uno de los threads con `pthread_create()`):

```
      pthread_attr_t attr;
      pthread_attr_init(&attr);
      pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
```

. Cuando ya no necesite el atributo creado, destrúyalo para liberar recursos:

```
      // libero recurso que ya no necesito
      pthread_attr_destroy(&attr);
```

Debería obtener una salida similar a esta:

```
grchere@debian:~/gccwork/src/thr1$ ./tp33 5
main(): creando thread 0
Mensaje de thread 0!
main(): creando thread 1
Mensaje de thread 1!
```

```
main(): creando thread 2
Mensaje de thread 2!
main(): creando thread 3
Mensaje de thread 3!
main(): creando thread 4
Mensaje de thread 4!
main(): haciendo join con thread 0
fin Mensaje de thread 0!
fin Mensaje de thread 1!
main(): termino join con thread 0 status= 0
main(): haciendo join con thread 1
main(): termino join con thread 1 status= 0
main(): haciendo join con thread 2
fin Mensaje de thread 2!
main(): termino join con thread 2 status= 0
main(): haciendo join con thread 3
fin Mensaje de thread 3!
main(): termino join con thread 3 status= 0
main(): haciendo join con thread 4
fin Mensaje de thread 4!
main(): termino join con thread 4 status= 0
fin main():1
grchere@debian:~/gccwork/src/thr1$
```

Observe los mensajes, mientras que `main()` esta detenido por el join con el thread 0, vemos que termina el thread 0 y el 1, entonces `main()` se libera y pretende bloquearse con el join con el thread 1, pero éste ya habia terminado, por lo tanto, este join, también termina inmediatamente, sin bloquear a `main()`. Luego se van alternando los bloqueos de `main()` con las finalizaciones de los threads. Pero el último mensaje impreso en la consola es la finalización de la función `main()`, ésta termina luego de que todos sus threads han terminado.
Se observa también que el valor de retorno de cada uno de los threads que recupera la función `main()` (usando la función join) es siempre cero.

4. Copie el programa anterior en `tp34.c`. Modificar el programa para que los threads que se ejecutan retornen a `main()` un numero distinto de cero, podemos usar el numero de thread más uno.

Debería obtener una salida similar a esta:

```
grchere@debian:~/gccwork/src/thr1$ ./tp34 5
main(): creando thread 0
Mensaje de thread 0!
main(): creando thread 1
Mensaje de thread 1!
main(): creando thread 2
Mensaje de thread 2!
main(): creando thread 3
Mensaje de thread 3!
main(): creando thread 4
Mensaje de thread 4!
main(): haciendo join con thread 0
fin Mensaje de thread 0!
fin Mensaje de thread 1!
main(): termino join con thread 0 status= 1
main(): haciendo join con thread 1
main(): termino join con thread 1 status= 2
```

```
main(): haciendo join con thread 2
fin Mensaje de thread 2!
main(): termino join con thread 2 status= 3
main(): haciendo join con thread 3
fin Mensaje de thread 4!
fin Mensaje de thread 3!
main(): termino join con thread 3 status= 4
main(): haciendo join con thread 4
main(): termino join con thread 4 status= 5
fin main():1
grchere@debian:~/gccwork/src/thr1$
```

5. Problema: Una empresa tiene 5 sucursales de las cuales registra las ventas de un mes. Existe un proceso que carga los datos de estas ventas. Luego esas ventas deben ser sumarizadas por sucursal para obtener un total general. Debemos diseñar una solución a este problema a través de un programa que trabaje en paralelo usando Threads. Siguiendo el apunte `Introduction to Parallel Computing.htm` podemos pensar en qué tipo de *particionado* podríamos hacer en este problema. Podríamos implementar la función `cargoVentas()` como un thread que nos permita generar los datos de ventas de cada una de las sucursales en forma aleatoria de la siguiente manera:

```
#define NUM_OF_SUC 5

// datos de ventas de las sucursales
double mivta[NUM_OF_SUC][31];  // se asume un maximo de 31 dias para el mes en
curso
double totalgral=0.0;
double totalsuc[NUM_OF_SUC];
...
void *cargoVentas(void *p) {
   int s,d;
   printf("thread cargoVentas\n");
   for(s=0;s<NUM_OF_SUC;s++) {
      for(d=0;d<31;d++) {
         mivta[s][d] = ((double) random())/10000000.0;
      }
      sleep(2);  // para simular una mayor duración de este proceso de carga
   }
   printf("fin thread cargoVentas\n");
}
...
```

Una vez generadas las ventas, podríamos utilizar un thread por sucursal para realizar la suma, a través de la función `sumoVentas()`, la cual recibe como argumento el id de sucursal (número entero de `0` a `NUM_OF_SUC - 1`):

```
void *sumoVentas(void *sucursalid) {
   int sucid;
   sucid = (int) sucursalid;
   printf("thread %d sumoVentas sucursal %d \n", sucid,sucid);
   totalsuc[sucid]=0.0;
   int d;
   for(d=0;d<31;d++) {
      totalsuc[sucid]+=mivta[sucid][d];
   }
```

```
   printf("fin thread %d sumatoria de sucursal %d
$%10.2f\n",sucid,sucid,totalsuc[sucid]);
}
```

Una vez que termine la suma de todas las sucursales, se puede calcular el total general, desde la función `main()`:

```
...
int t;
for(t=0; t<NUM_OF_SUC; t++) totalgral+=totalsuc[t];
printf("main():Total Gral $ %12.2f\n",totalgral);
...
```

Debemos pensar acerca de la ejecución concurrente de `cargoVentas()` y `sumoVentas()`. Existe una *particion de datos* en `sumoVentas()` puesto que ésta trabaja por sucursal, con lo cual, si sumo las ventas de la sucursal 0 ello no debería interferir con estar sumando las ventas de la sucursal 1 al mismo tiempo. El problema estaría en generar datos de ventas al mismo tiempo que pretendo sumarlos. Por lo tanto, se propone implementar una forma de exclusión mutua: no se pueden generar datos y sumarlos al mismo tiempo.

Nota: el código de las funciónes antes descriptas esta incompleto, falta todo código relacionado con el tratamiento de threads.

Sugerencias: Copie el programa anterior en `tp35.c`, elimine todo el código vinculado con la línea de comandos (a este programa ya no se le indica con cuántas sucursales va a trabajar), implemente el código propuesto anteriormente, agregue todo el código de threads que sea necesario, al comienzo del programa, inicializar los datos numéricos (a cero), esto puede hacerse utilizando la función `memset()`:

```
...
memset(&mivta,0,sizeof(double)*NUM_OF_SUC*31);
memset(&totalsuc,0,sizeof(double)*NUM_OF_SUC);
...
```

Para usar `memset()`, debe incluir la cabecera `string.h`

Responda:
5.1. ¿Desde el punto de vista del diseño de programas en paralelo, el thread `cargoVentas()` a qué tipo de descomposición corresponde?
5.2. Idem anterior con los threads `sumoVentas()` ¿A qué tipo de descomposición corresponde?

Debería obtener una salida similar a esta:

```
grchere@debian:~/gccwork/src/thr1$ ./tp35
main(): creando thread para carga de ventas
thread cargoVentas
main(): creando thread 0
thread 0 esperando carga de sucursal 0
main(): creando thread 1
thread 1 esperando carga de sucursal 1
main(): creando thread 2
thread 2 esperando carga de sucursal 2
```

```
main(): creando thread 3
thread 3 esperando carga de sucursal 3
main(): creando thread 4
thread 4 esperando carga de sucursal 4
main(): haciendo join con thread 0
fin thread cargoVentas
thread 0 sumoVentas sucursal 0
fin thread 0 sumatoria de sucursal 0  $   3337.60
thread 1 sumoVentas sucursal 1
fin thread 1 sumatoria de sucursal 1  $   3421.03
thread 2 sumoVentas sucursal 2
fin thread 2 sumatoria de sucursal 2  $   3904.04
thread 3 sumoVentas sucursal 3
fin thread 3 sumatoria de sucursal 3  $   3514.70
thread 4 sumoVentas sucursal 4
fin thread 4 sumatoria de sucursal 4  $   3326.67
main(): termino join con thread 0 status= 1
main(): haciendo join con thread 1
main(): termino join con thread 1 status= 2
main(): haciendo join con thread 2
main(): termino join con thread 2 status= 3
main(): haciendo join con thread 3
main(): termino join con thread 3 status= 4
main(): haciendo join con thread 4
main(): termino join con thread 4 status= 5
main():Total Gral $     17504.03
fin main():1
grchere@debian:~/gccwork/src/thr1$
```

Observando la salida podemos ver que todas las sumarizaciones de sucursales quedan bloqueadas mientras que `cargoVentas()` esta en ejecución.
Efectivamente, habíamos dicho que estas operaciones no podían ser concurrentes, se ha implementado una exclusión mutua entre ambas: mientras que se generan datos, no se pueden sumar. Una vez que `cargoVentas()` finaliza, comienzan a liberarse una a una las tareas que suman las sucursales. Cuando termina el ultimo sumador (como `main()` hizo un join a cada uno de estos sumadores, éste queda bloqueado hasta que termine el último sumador), `main()` se libera, hace el total general, imprime el resultado y termina.

6. Intencionalmente (para no complejizar demasiado el código de una sola vez), he introducido un error en el razonamiento del diseño del programa anterior, ¿Puede advertir cuál es?, observe nuevamente la salida del programa anterior:

1. comienza la ejecución de main()
2. se ejecuta `cargoVentas()`
3. se ejecutan los sumadores por sucursal
4. se termina de ejecutar `main()`

¿No le parece un código secuencial? ¿Qué clase de provecho le estamos sacando al paralelismo? Evidentemente la solución propuesta no es la más adecuada y requiere de algún tipo de refinamiento. El error está en cuanto a la exclusión mutua, no es posible sumarizar mientras se generan los datos, pero una vez que se ha generado los datos de la sucursal *n* ya puede desbloquearse el sumador de la sucursal *n*, ¿para qué dejar bloqueado el sumador de la sucursal *n* cuando ya se han "producido" todos sus datos y éstos pueden ser "consumidos"?.
Se requiere poder hacer un bloqueo por sucursal, pero no es un bloqueo al estilo de exclusión como

se hizo en el punto anterior, sino que, *este bloqueo debe realizarse a condición de que la sucursal n aún no esté generada*. Una solución posible es utilizar *variables de condición*, una por sucursal, entonces el sumador de la sucursal *n* estaría esperando por la variable de condicion *n*, cuando `cargoVentas()` haya terminado la generación de los datos de la sucursal n puede cambiar el valor de la variable e indicarle al sumador *n* que ya puede hacer su trabajo.

Sugerencia: Copie el programa anterior en `tp36.c`. Se puede utilizar un arreglo de enteros, uno por sucursal, a modo de flags o banderas, para indicar que la sucursal *n* ha sido generada; inicialmente todo el vector esta en cero (falso):

```
...
int cargadas[NUM_OF_SUC];
....
   memset(&cargadas,0,sizeof(int)*NUM_OF_SUC); //cargadas[n] indica falso (cero)
....
```

Habrá una variable de condición y de exclusión mutua (mutex) por cada sucursal:

```
...
pthread_mutex_t cargada_mutex[NUM_OF_SUC];
pthread_cond_t cargada_cv[NUM_OF_SUC];
...
```

Ahora la función `cargoVentas()` deberá colocar un valor distinto de cero en el elemento *n* del arreglo `cargadas` para indicar que terminó con la generación de los datos de la sucursal *n* e indicar que se dió la condición por la cual esta esperando el thread sumador de sucursal *n*:

```
....
void *cargoVentas(void *p) {
...
      cargadas[n]=1;
      pthread_cond_signal(&cargada_cv[n]);
...
}
```

Entonces, la función `sumoVentas()` deberá esperar para obtener el candado de la sucursal *n* a sumar, preguntar por la variable de condición y quedarse en espera de la misma, antes de hacer su trabajo:

```
...
void *sumoVentas(void *threadid) {
...
   printf("thread %d esperando carga de sucursal %d \n", tid,tid);
   pthread_mutex_lock(&cargada_mutex[tid]);
   if ( !cargadas[tid] ) {
      printf("thread %d esperando condicion de cargada para sucursal %d \n",
tid,tid);
      pthread_cond_wait(&cargada_cv[tid], &cargada_mutex[tid]);
   }
   printf("thread %d sumo Ventas sucursal %d \n", tid,tid);
...
...
   pthread_mutex_unlock(&cargada_mutex[tid]);
...
```

```
}
```

Nota: No olvide que si Ud. hace un lock() sobre una variable de condición, debe también hacerse su correspondiente unlock().

Debería obtener una salida similar a esta:

```
grchere@debian:~/gccwork/src/thr1$ ./tp36
main(): creando thread para carga de ventas
thread cargoVentas
main(): creando thread 0
thread 0 esperando carga de sucursal 0
thread 0 sumo Ventas sucursal 0
fin thread 0 sumatoria de sucursal 0  $   3337.60
main(): creando thread 1
thread 1 esperando carga de sucursal 1
thread 1 esperando condicion de cargada para sucursal 1
main(): creando thread 2
thread 2 esperando carga de sucursal 2
thread 2 esperando condicion de cargada para sucursal 2
main(): creando thread 3
thread 3 esperando carga de sucursal 3
thread 3 esperando condicion de cargada para sucursal 3
main(): creando thread 4
thread 4 esperando carga de sucursal 4
thread 4 esperando condicion de cargada para sucursal 4
main(): haciendo join con thread 0
main(): termino join con thread 0 status= 1
main(): haciendo join con thread 1
thread 1 sumo Ventas sucursal 1
fin thread 1 sumatoria de sucursal 1  $   3421.03
main(): termino join con thread 1 status= 2
main(): haciendo join con thread 2
thread 2 sumo Ventas sucursal 2
fin thread 2 sumatoria de sucursal 2  $   3904.04
main(): termino join con thread 2 status= 3
main(): haciendo join con thread 3
thread 3 sumo Ventas sucursal 3
fin thread 3 sumatoria de sucursal 3  $   3514.70
main(): termino join con thread 3 status= 4
main(): haciendo join con thread 4
thread 4 sumo Ventas sucursal 4
fin thread 4 sumatoria de sucursal 4  $   3326.67
main(): termino join con thread 4 status= 5
main(): haciendo join con carga de ventas
fin thread cargoVentas
main(): termino join con carga de ventas status= 0
main():Total Gral $     17504.03
fin main():1
grchere@debian:~/gccwork/src/thr1$
```

Se observa que esta versión tiene un mayor nivel de concurrencia, la tarea `cargoVentas()` se esta ejecutando durante casi todo el tiempo, `main()` debe hacer también un join con `cargoVentas()` para que éste no pretenda calcular el total general antes de que haya terminado la generación y sumarización por sucursal. Este codigo da oportunidad de que se ejecute, por ejemplo, la sumarización de la sucursal `0`, mucho antes de que termine `cargoVentas()`.

7. Verifique el código de la función `cargoVentas()` en el programa `tp35.c` y `tp36.c`, ambas versiones de esta función deberían tener una llamada idéntica a la función `sleep()` para simular una mayor duración del proceso de generación de datos de sucursales. Este `sleep()` no pretende tener ningún tipo de relación con la sincronización de estos procesos (es un grave error intentar, a partir de esperas o tiempos arbitrarios, lograr la sincronización de procesos). Verifique que en ambas versiones de esta función tengan la misma demora.
Utilice el comando Unix `time` para hacer una ejecución del programa `tp35` y otra del programa `tp36`, de la siguiente forma:

```
grchere@debian:~/gccwork/src/thr1$ time ./tp35
.... <observará la salida habitual de tp35>...
real  0mXXXXXs
user  0mXXXXXs
sys   0mXXXXXXs
```

```
grchere@debian:~/gccwork/src/thr1$ time ./tp36
.... <observará la salida habitual de tp35>...
real  0mYYYYYs
user  0mYYYYYs
sys   0mYYYYYs
```

El comando `time` permite sumarizar la utilización de recursos del sistema por parte de un proceso. `Real` indica el tiempo real transcurrido en la ejecución del programa (wall clock) indicado en horas, minutos y segundos. Realice varias corridas y compare los tiempos obtenidos en `Real` para ambos procesos.

Responda:
6.1 ¿Cuál de los dos procesos obtiene valores menores de real?
6.2 ¿Por qué Ud. cree que esos valores de real son menores?
Cambie (en ambos procesos) el tiempo de espera dentro de la función sleep() de 2" a 5" y vuelva a comparar los tiempos
6.3 ¿Encontró alguna diferencia?
6.4 ¿El proceso que tiene menor valor de Real continúa siendo el mismo?

Bibliografía

- Stevens, Richard, "Advanced Programming in the UNIX Environment", Addison-Wesley Professional Computing Series, 1993, ISBN 0-201-56317-7
- LooseSandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram, Ulrich Drepper, "The GNU C Library Reference Manual", Free Software Foundation, 2007, Boston, USA.

Espero que esta práctica haya sido de vuestro agrado y le permita comprender mejor los mecanismos de sincronización de threads, las cuestiones de diseño que involucran a programas con ejecución en paralelo y la aplicación de la librería pthread (POSIX Threads).