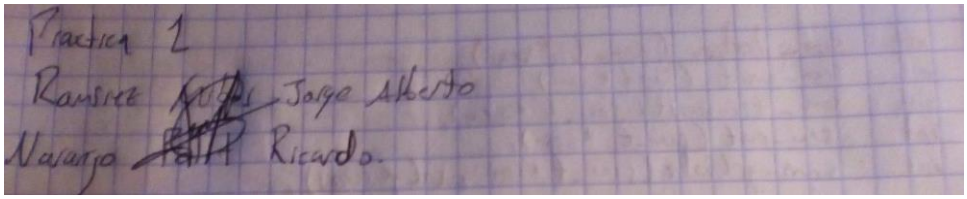
Instituto Politécnico Nacional

Escuela Superior de Cómputo

Práctica #1
Hilos



Fecha de entrega: 06/08/2018

Maestro: Rangel González Josué

Materia: “Aplicaciones para comunicaciones de red”

Grupo: 3CV8

Equipo:

Naranjo Polit Ricardo

Ramirez Gibbs Jorge Alberto

Introducción

Un proceso es el resultado de una técnica que emplean los sistemas operativos para permitir la ejecución simultánea de distintas aplicaciones. Esta técnica consiste en dejar correr a una aplicación por un tiempo, digamos 10 ms. Cuando se agota el tiempo, el kernel del SO retoma el control y se lo entrega a otra aplicación. Al mismo tiempo que el SO cambia el control de una aplicación a otra, también intercambia, por cada aplicación, información adicional tal como:

- Qué archivos están abiertos.
- La memoria de la aplicación.
- La pila de ejecución.

O sea, al salto se le suma el intercambio de información de contexto. Y a la unidad compuesta por toda esa información contextual se la conoce como *proceso*.
El concepto de thread implica solamente saltar, pero mantener el contexto. Es decir que distintas partes de la aplicación se ejecutan concurrentemente, compartiendo memoria y archivos abiertos (la pila de ejecución no se intercambia).
Cuando Linux era jovencito, no tenía threads. Los demás sistemas operativos se pavoneaban exponiendo complejos mecanismos de threads. En estos sistemas (como también es hoy en Windows), los threads son conceptos de primer nivel en el sistema. Es decir que están implementados como algo especial y fijo.
Mientras tanto Linux crecía y crecía, pero solamente tenía procesos. Las primeras versiones de Java tenían que, en Linux, simular la existencia de los threads. Linus Torvalds, el creador de Linux, siempre pensó que los threads eran algo muy parecido a los procesos, tan parecidos que debían ser lo mismo. Mientras que un proceso nace de un fork "completo", los threads vendrían a ser el resultado de un fork parcial, en el que, en vez de duplicar todo el contexto, el proceso hijo mantenga parte de él compartido con el padre. Entonces se creó una llamada al sistema que permite decirla al sistema cuáles partes del contexto se desean duplicar, esta llamada es **clone()**.
Sin embargo, la forma estándar de manipular threads en UNIX (conocida como pthreads) no era del todo compatible con esta conceptualización. Por ejemplo, espera que cada thread vea el mismo "identificador de proceso", y con el mecanismo de Linux eso no era posible, ya que cada thread era un proceso distinto.
A través de los años Linux fue añadiendo funciones para dar mejor soporte a una API pthreads. La implementación actual de pthreads (conocida como NPTL) llegó actualmente a ofrecer el manejo exacto que ofrecen otros sistemas.

Veamos cómo es esa API...

La función principal es obviamente la que permite crear un thread. Esta función es pthread_create:

```
int pthread_create(pthread_t * thread, pthread_attr_t * attr, void *(*funcion)(void *), void *arg);
```

Los parámetros son los siguientes:

thread
Un puntero en donde la función guardará el identificador del nuevo thread creado.

attr
Atributos del thread, puede ser NULL.

funcion
Acá empieza la ejecución del nuevo thread. Es un puntero a una función declarada de esta manera: void mi_thread(void *arg);.

arg
Parámetro a pasar a la función del nuevo thread

Esta función devuelve el control inmediatamente... mientras tanto, el nuevo thread inicia su recorrido por la función apuntada por funcion. ¿Hasta cuándo? Como mucho hasta que termina ese método, cuando sale, termina el thread. Si un thread necesita esperar a que otro termina (por ejemplo el thread padre esperar a que termine el hijo) puede usar la función pthread_join(). ¿Por qué se llama así? Bueno, crear un proceso es como una bifuración, se abren dos caminos... que uno espere a otro es lo contrario, una unificación (join en inglés).

Algoritmos genéticos

El algoritmo genético es un algoritmo evolutivo clásico basado en el azar. Al azar aquí nos referimos a que para encontrar una solución usando el GA, se aplicaron cambios aleatorios a las soluciones actuales para generar nuevas. Tenga en cuenta que GA se puede llamar Simple GA (SGA) debido a su simplicidad en comparación con otros EA.

GA se basa en la teoría de la evolución de Darwin. Es un proceso lento y gradual que funciona haciendo cambios para hacer cambios lentos y lentos. Además, GA realiza pequeños cambios en sus soluciones lentamente hasta obtener la mejor solución.

GA trabaja en una población que consiste en algunas soluciones donde el tamaño de la población (popsize) es el número de soluciones. Cada solución se llama individual. Cada solución individual tiene un cromosoma. El cromosoma se representa como un conjunto de parámetros (características) que define al individuo. Cada cromosoma tiene un conjunto de genes. Cada gen se representa de alguna manera, como ser representado como una cadena de 0 y 1.

Además, cada individuo tiene un valor de aptitud. Para seleccionar las mejores personas, se usa una función de aptitud. El resultado de la función de aptitud física es el valor de aptitud que representa la calidad de la solución. Cuanto mayor sea el valor de la aptitud, mayor será la calidad de la solución. La selección de los mejores individuos en función de su calidad se aplica para generar lo que se denomina un grupo de apareamiento donde el individuo de mayor calidad tiene una mayor probabilidad de ser seleccionado en el grupo de apareamiento.

Las personas en el grupo de apareamiento se llaman padres. Cada dos padres seleccionados del grupo de apareamiento generarán dos hijos (hijos). Al simplemente aparear individuos de alta calidad, se espera obtener una descendencia de mejor calidad que sus padres. Esto matará a las personas malas de generar más malas personas. Al mantener la selección y el apareamiento de individuos de alta calidad, habrá mayores posibilidades de mantener buenas propiedades de las personas y dejar de lado las malas. Finalmente, esto terminará con la solución óptima o aceptable deseada.

Pero los descendientes generados actualmente usando los padres seleccionados solo tienen las características de sus padres y nada más sin cambios. No se le agrega nada nuevo y por lo tanto los mismos inconvenientes en sus padres realmente existirán en los nuevos descendientes. Para superar este problema, se aplicarán algunos cambios a cada descendencia para crear nuevos individuos. El conjunto de todas las personas recién generadas será la nueva población que reemplazará a la población anterior utilizada anteriormente. Cada población creada se llama generación. El proceso de reemplazar la población anterior por la nueva se llama reemplazo.

Descripción del problema

(A)
Modificar el algoritmo para introducir hilos. Los diferentes hilos ejecutarán el programa con diferentes semillas que generarán diferentes poblaciones.

-Ejecutar el algoritmo sin hilos y medir su tiempo de ejecución.

-Ejecutar el algoritmo con 2, 4, 6 hilos y medir sus tiempos.

-Obtener las gráficas de tiempos y concluir los resultados.

(B)
Modificar la función “fuerte” para que esa función se ejecute con 2 hilos.

Descripción de la solución

-Ejecutar el algoritmo sin hilos y medir su tiempo de ejecución.

Se modificó el programa para que se pudieran iniciar varias poblaciones en una misma ejecución del programa, se logró esto al mover el código que estaba en la función main y

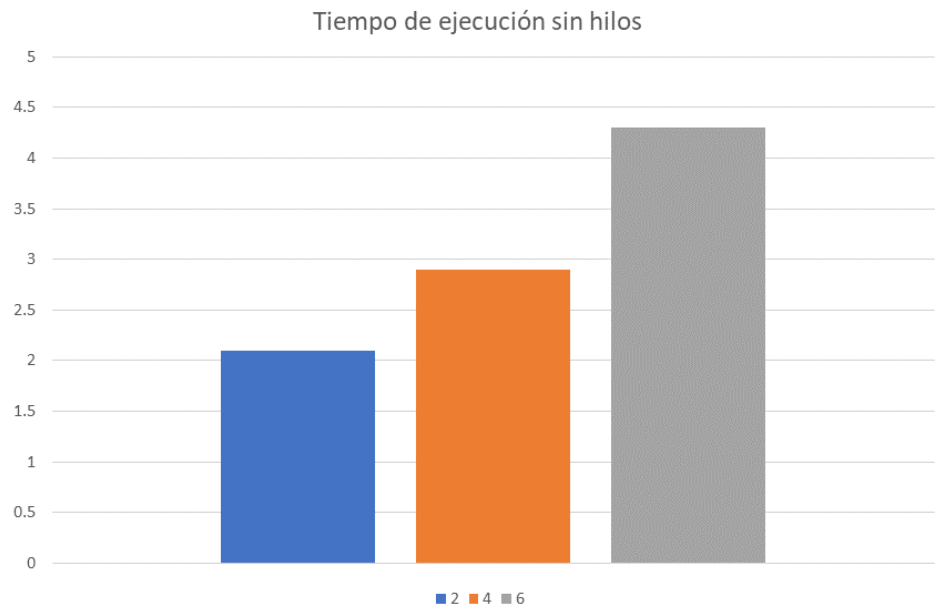
colocarlo en una función extra llamada “creaSociedad” para así mandar a llamar varias veces a esta función y que se creen diferentes poblaciones, también se modificó para que la función necesite de una semilla y esta se utiliza para crear los valores aleatorios de la población.

```
int main( ){

  int socNum = 6;

  int *semillas;

  semillas = (int *) malloc (socNum*sizeof(int ));

  for(int i = 0; i < socNum; i++){
    semillas[i] = i+10;
    creaSociedad(&semillas[i]);
  }

  return 0;
}
```

Se ejecutó el programa con 2, 4 y 6 poblaciones sin hacer uso de hilos , dando los siguientes resultados en tiempos.

```
ricardo@ricardo-chrome:~/Documents/REDES2$ g++ practica1A.cpp -o sinhilos -lpthread
ricardo@ricardo-chrome:~/Documents/REDES2$ time ./sinhilos

real    0m2.115s
user    0m2.100s
sys     0m0.006s
ricardo@ricardo-chrome:~/Documents/REDES2$ #prueba del inciso A sin hilos con 2 poblaciones
ricardo@ricardo-chrome:~/Documents/REDES2$ g++ practica1A.cpp -o sinhilos -lpthread
ricardo@ricardo-chrome:~/Documents/REDES2$ time ./sinhilos

real    0m2.996s
user    0m2.975s
sys     0m0.005s
ricardo@ricardo-chrome:~/Documents/REDES2$ #prueba del inciso A sin hilos con 4 poblaciones
ricardo@ricardo-chrome:~/Documents/REDES2$ g++ practica1A.cpp -o sinhilos -lpthread
ricardo@ricardo-chrome:~/Documents/REDES2$ time ./sinhilos

real    0m4.326s
user    0m4.282s
sys     0m0.006s
ricardo@ricardo-chrome:~/Documents/REDES2$ #prueba del inciso A sin hilos con 6 poblaciones
```

Representándolos en un gráfico se obtienen los siguientes resultados. El tiempo de ejecución más largo sucedió cuando se emplearon 4 hilos.



-Modificar el algoritmo para que utilice hilos.
En este inciso modificamos el main para emplear hilos , siemplemente usando pthread_create( ) en lugar de llamar directamente a la función creaSociedad. Le asignamos dicha función como rutina de inicio y le pasamos las semillas como argumentos mediante pthread_create().

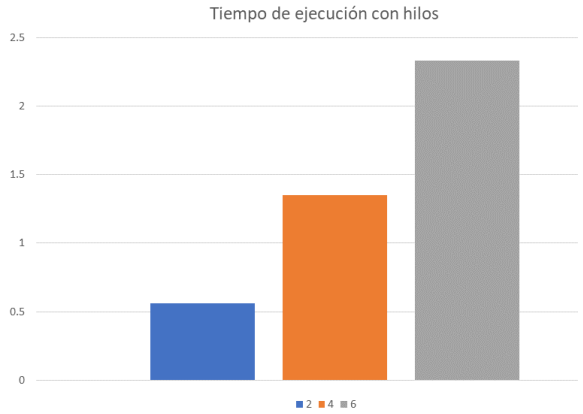
practica1A.cpp | practica1.cpp

```
int main( ){

  int healt;
  int socNum = 6;
  int *semillas;

  pthread_t *hilos;
  semillas = (int *) malloc (socNum*sizeof(int ));
  hilos = (pthread_t *) malloc(sizeof(pthread_t)*socNum);

  for(int i = 0; i < socNum; i++){
    semillas[i] = i+10;
    healt = pthread_create(&hilos[i],NULL,creaSociedad,&semillas[i]);


    if(healt == -1){
      cout << "Error al crear el hilo: " << i << endl;
      exit(0);
    }
    //cout << "Hilo" << i <<"creado satisfactoriamente"<< endl;


  }

  for (int i = 0; i < socNum; i++) {
    pthread_join(hilos[i], NULL);
    //cout << "Join: " << i << endl;
  }

  return 0;
}

void *creaSociedad(void *a){

  int *seed = (int *) a;
  srand ((*seed));
  double valorvars;
```

Ejecutando el programa obtenemos los siguientes tiempos de ejecución del programa con uso de hilos. Se nota un decremento en los tiempos de ejecución del programa cuando éstos se usan.

```
ricardo@ricardo-chrome:~/Documents/REDES2$ g++ practica1.cpp -o conhilos -lpthread
ricardo@ricardo-chrome:~/Documents/REDES2$ time ./conhilos

real    0m0.567s
user    0m1.033s
sys     0m0.012s
ricardo@ricardo-chrome:~/Documents/REDES2$ #prueba del inciso A con 2 hilos
ricardo@ricardo-chrome:~/Documents/REDES2$ g++ practica1.cpp -o conhilos -lpthread
ricardo@ricardo-chrome:~/Documents/REDES2$ time ./conhilos

real    0m1.355s
user    0m2.479s
sys     0m0.028s
ricardo@ricardo-chrome:~/Documents/REDES2$ #prueba del inciso A con 4 hilos
ricardo@ricardo-chrome:~/Documents/REDES2$ g++ practica1.cpp -o conhilos -lpthread
ricardo@ricardo-chrome:~/Documents/REDES2$ time ./conhilos

real    0m2.339s
user    0m4.009s
sys     0m0.038s
ricardo@ricardo-chrome:~/Documents/REDES2$ #prueba del inciso A con 6 hilos
```

El comportamiento del programa cambia cuando se usan hilos. A juzgar por la gráfica los tiempos de ejecución en este caso son proporcionales al número de hilos que se utilicen.



-Modificar la función “fuerte” para que se ejecute con 2 hilos.

Si ejecutamos el programa sin el uso de threads obtendremos los siguientes tiempos de ejecución.

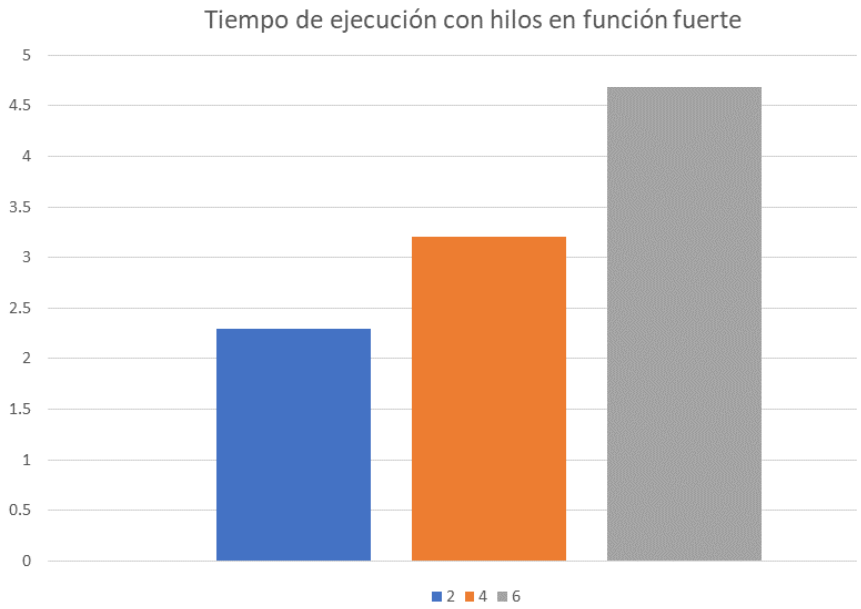
Modificaremos la función fuerte para que trabaje con dos hilos.

```
int * poblacion::fuerte(){
  sumar();
  porcentaje();
  acomula();
  alea();
  Gan.clear();
  //Declaración y reserva de memoria de hilo
  pthread_t hilo1;
  pthread_t hilo2;
  int healt;
  struct estructuraHilos *estructura;
  //Se obtienen los ganadores de la poblacio
  for(int i = 0; i < indi.size(); i++){
    for(int j = 0; j < indi.size(); j++){
      if(aleatorio[i]<=zAcum[j]){
        Gan.push_back(j);
        break;
      }
    }
  }
```

Declaramos los hilos y estructuraHilos, una estructura con la cual pasaremos los individuos y el vector cubeta
Obtenemos los siguientes tiempos de ejecución.

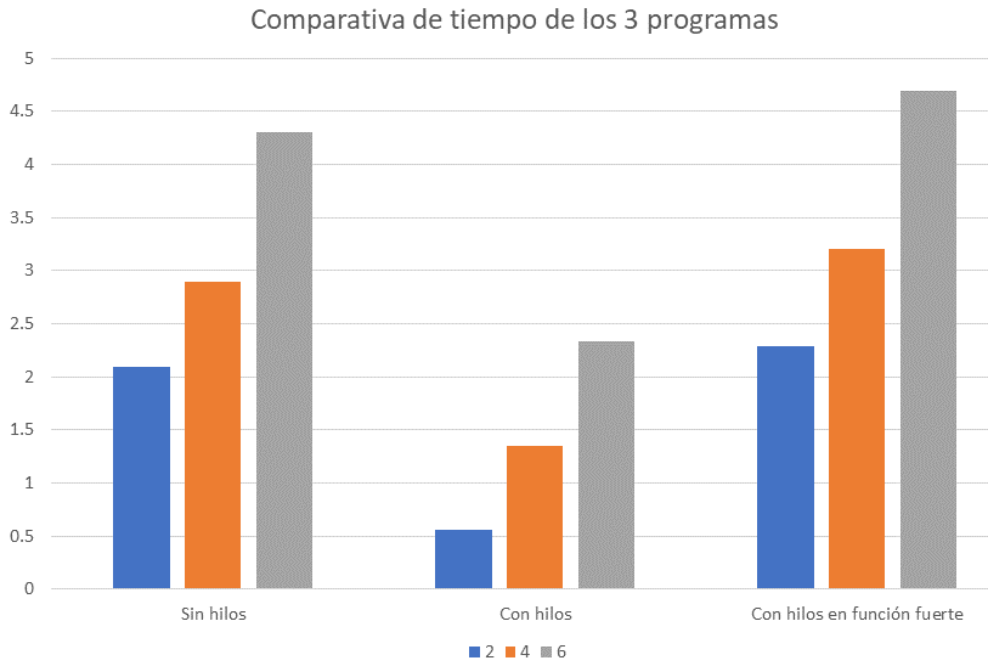
```
ricardo@ricardo-chrome:~/Documents/REDES2$ g++ practica1B.cpp -o conhilosFuerte -lpthread
ricardo@ricardo-chrome:~/Documents/REDES2$ time ./conhilosFuerte

real    0m2.293s
user    0m2.259s
sys     0m0.011s
ricardo@ricardo-chrome:~/Documents/REDES2$ #prueba del inciso B con 2 hilos
ricardo@ricardo-chrome:~/Documents/REDES2$ g++ practica1B.cpp -o conhilosFuerte -lpthread
ricardo@ricardo-chrome:~/Documents/REDES2$ time ./conhilosFuerte
real    0m3.219s
user    0m3.141s
sys     0m0.019s
ricardo@ricardo-chrome:~/Documents/REDES2$ #prueba del inciso B con 4 hilos
ricardo@ricardo-chrome:~/Documents/REDES2$ g++ practica1B.cpp -o conhilosFuerte -lpthread
ricardo@ricardo-chrome:~/Documents/REDES2$ time ./conhilosFuerte
real    0m4.692s
user    0m4.624s
sys     0m0.017s
ricardo@ricardo-chrome:~/Documents/REDES2$ #prueba del inciso B con 6 hilos
```

Representándolos en un gráfico se obtienen los siguientes resultados.



-Obtener las gráficas de tiempos y concluir los resultados

A continuación, se va muestra la comparación entre las tres gráficas, se puede ver claramente que los tiempos en los que fueron usados los hilos correctamente bajaron considerablemente, a diferencia de los tiempos de las otras dos graficas.



Conclusiones

Ricardo Naranjo Polit:

En esta práctica se pudo ver que el uso de hilos es una tarea que puede llegar a ser complicada, y que no siempre usar hilos en los programas es efectivo porque no si no se usan para paralelizar tareas independientes entonces el tiempo que se tarda al hacerse con hilos o sin ellos prácticamente es el mismo.

Al realizar esta práctica se encontraron varios obstáculos, en algún momento de la programación los hilos estaban llegando a un error de “segmentation fault”, por lo cual fue muy complicado encontrar el fallo y corregirlo. También se encontraron complicaciones porque se necesitaban estar mandando vectores a las funciones de los hilos, y cuando se mandaban a veces los hilos las recibían con basura, por lo que se tuvo que tener mucho cuidado al inicializar los vectores para detener ese problema.

Ramírez Gibbs Jorge Alberto

Vimos en esta práctica varias cosas que implican usar hilos y cómo no siempre puede ser una ventaja para programas como el que programamos en la presente práctica. El funcionamiento de los hilos también varía entre máquinas dependiendo de arquitecturas y número de núcleos. Tuvimos que revisar conceptos de paso de parámetros entre hilos y las APIs de pthread para comprender como pasar correctamente vectores a las funciones de rutina de inicio. Tuvimos algunas dificultades (entre ellas el paso de vectores) que nos tomaron tiempo pero al final fueron provocados por un mal uso de funciones de reserva de memoria el cual se solucionó después (los hilos recibían vectores llenos de basura y caíamos en un error de tipo std::bad_alloc).

Si se considera usar hilos se debe de tomar en cuenta si la tarea por realizar se puede paralelizar y si reamente es conveniente paralelizarlo y no hacerlo concurrente.