Laboratorio 1 - Sistemas operativos

Jeanlee Barreto

Octubre 2020

Ejercicio 1

Enunciado

Para calcular el valor de π se puede usar la tecnica Monte Carlo la cual consiste en:

1. Imaginar que se tiene un círculo dentro de un cuadrado (asumir el $radio = 1$).
2. Generar una serie de puntos random como coordenadas simples (x,y). Estos puntos deben estar en los límites del cuadrado y algunos estarán dentro del círculo.
3. Estimar:

$$\pi = 4 \cdot (numero\_puntos\_en\_circulo)/(numero\_total\_de\_puntos)$$

Escribir una versión multithreaded del algoritmo en C (usar POSIX), que cree una thread separada para generar un número de puntos random.
Nota: generar números random tipo double entre -1 y +1. La thread contará el número de puntos que caen dentro del círculo y guardará el resultado en una variable global. Solamente cuando la thread child termine, la thread parent calculará e imprimirá el valor estimado de π. Se recomienda experimentar con el número puntos random generados, mientras mayor sea el número la aproximación a π será mayor.

Análisis del problema

Como el método de Mote Carlo para calcular π se basa en generar puntos random dentro de un espacio cuadrado que presenta un circulo circunscrito, definiremos dicho espacio. Nuestro circulo posee un radio de 1, y el cuadrado se encuentra centrado en el eje de coordenadas. A continuación se muestra un diagrama con la disposición de dichas figuras.

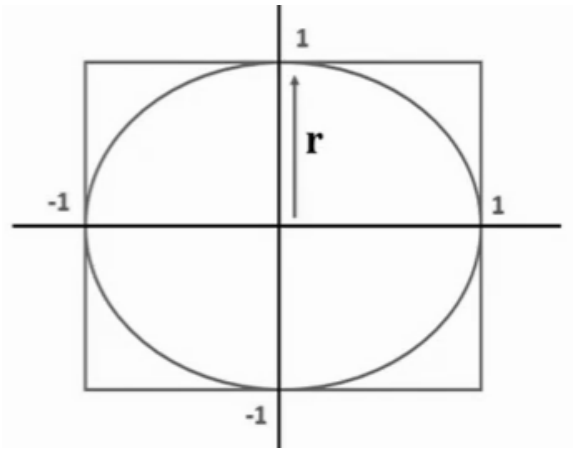

Figura 1: Cuadrado con un circulo circunscrito de radio 1

Ahora bien, para un punto lanzado con coordenadas (a, b) que cae justo en la circunferencia del circulo, su distancia desde el origen será igual a r. En este caso, usando el teorema de Pitágoras se obtiene que:

$$a^2 + b^2 = r^2 = 1$$

Teniendo esto presente, y considerando que el punto se encontraba en el límite del círculo, se puede afirmar que para un punto P, con coordenadas (x, y), este va a estar dentro del circulo si y solo sí cumple que:

$$x^2 + y^2 \leq 1$$

Con esta condición ya podemos seguir con la implementación de la solución del ejercicio.

Implementación

En general, para la resolución de este y los siguientes ejercicios se utilizará la implementación del estándar POSIX thread para el lenguaje C, *pthread*.

Para la implementación se define una macro N_POINTS que representa el número de puntos aleatorios a generar, por defecto se encuentra en 1000000. También se tiene una variable global $POINTS\_IN$, inicializada en cero, que será el contador para los puntos aleatorios cuyas coordenadas se encuentran dentro del circulo.
Así mismo, se crea una función *double_ramdom* encargada de generar números decimales aleatorios en el rango [-1,0 - 1.0].
La función *montecarlo* es la encargado de crear los puntos aleatorios y verificar si estos se encuentran o no dentro del circulo, y de ser así incrementa el contador $POINTS\_IN$ en +1.
Por ultimo, la función principal *main* es la que crea un hilo *child*, con el método *pthread_create*, hilo que toma la función *montecarlo* para realizar toda la lógica del problema. A su vez, *main* se encarga de calcular π con los datos que se obtiene luego de que el hilo haya acabado (se usa *pthread_join*) y se imprime en pantalla el resultado.

Resultados

Los resultados que se obtuvieron fueron los siguientes:

Prueba 1:

```
Total puntos    1000000
Puntos dentro   786337
PI aproximado   3.145348
```

.
Prueba 2:

```
Total puntos    1000000
Puntos dentro   785425
PI aproximado   3.141700
```

.
Prueba 3:

```
Total puntos    1000000
Puntos dentro   785594
PI aproximado   3.142376
```

Ejercicio 2

Enunciado

Extender el programa anterior para n threads, cada una de las cuales genera puntos random y determina si estos están dentro del círculo. Cada thread debe actualizar la cuenta global de todos los puntos que caen en el circulo. Usar mecanismos de exclusión mutua POSIX mutex_lock para proteger la variable global compartida de race conditions en la actualización de valores.

Análisis del problema

Como se dice en el enunciado, lo que se tiene en esta ocasión son varios hilos tratando de actualizar la misma variable. Evidentemente nos encontramos con un problema de race condition cuya región crítica es la variable de conteo global para los puntos que se encuentran dentro del círculo.
En nuestra implementación del ejercicio anterior teníamos la variable global $POINTS\_IN$. Es ha dicha variable que debemos de proteger con un método de exclusión mutua. Por tanto, y como el problema lo indica, usaremos mutex.

Implementación

A nuestra implementación previa le agregaremos una macro $N\_THREADS$ que definirá cuantos hilos se correrán en el programa, por defecto está en 5. Así mismo, creamos nuestra variable $mutex$ del tipo $pthread_mutex_t$ el cual nos ayudará a coordinar los accesos a nustra variable de conteo global.
EL el método $montecarlo$, dentro de nuestra condición para validar si un punto está dentro del circulo, agregamos el $pthread\_mutex\_lock(\&mutex)$ y el $pthread\_mutex\_unlock(\&mutex)$; que encierran a la línea que modifica la variable global $POINTS\_IN$. De este modo aseguramos la exclusión mutua.
FInalmente, las tareas que se le agregó a la función $main$ fue la de crear un arreglo de hilos $th\_children[N\_THREADS]$ y de balancear la cantidad de puntos que cada hilo calculará en el programa. De este modo cada hilo tendrá una carga casi homogénea.

Resultados

Los resultados que se obtuvieron con el programa fueron los siguientes:

Prueba 1:
Para 5 threads

```
Puntos por hilo 200000
Total puntos    1000000
Puntos dentro   784693

PI aproximado   3.138772
```

.
Prueba 2:
Para 6 hilos

```
Puntos por hilo 166666 aprox.
-> se reparten los 4 puntos restantes
Total puntos    1000000
Puntos dentro   785265

PI aproximado   3.141060
```

.
Prueba 3:
Para 7 hilos

```
Puntos por hilo 142857 aprox.
-> se reparten los 1 puntos restantes
Total puntos    1000000
Puntos dentro   785667

PI aproximado   3.142668
```

Ejercicio 3

Enunciado

La secuencia Fibonacci es la series de números 0, 1, 1, 2, 3, 5, 8, ... Formalmente se expresa como:

$$fib_0 = 0$$

$$fib_1 = 1$$

$$fib_n = fib_{n-1} + fib_{n-2}$$

Escribir un programa multithreaded en C (usar POSIX) que genere la secuencia Fibonacci con las siguientes caracterÍsticas:

- El programa recibe como argumento la cantidad de números Fibonacci a generar.
- Luego el programa crea una thread separada que genera los números Fibonacci, colocándolos como datos que pueden ser compartidos por threads (ejemplo: un array es la forma más simple y conveniente par ala estructura de datos).
- Cuando la thread child finaliza ejecución, la thread parent imprime la secuencia generada por la thread child. La thread parent debe esperar que la thread child finalice para poder imprimir.

Análisis del problema

La serie de Fibonacci es una de las más conocidas, sin embargo, su implementación de forma recursiva suele ser muy costosa en tiempo de ejecución cuando se trata de calcular números altos, por ejemplo 50. Para este caso, y lo que se nos propone es realizar una implementación más sofisticada, con el uso de una estructura para guardar los números de la serie.
Este tipo de implementación suele ser llamada como *memorizada*, ya que el algoritmo no vuelve a calcular los números de la serie, que en Fibonacci ocurre constantemente, sino que al ya haberlos calculado se vuelvan a usar para los posteriores cálculos dentro del programa.
Este el enfoque le daremos al ejercicio y, claramente también usaremos hilos para la implementación.

Implementación

En esta oportunidad manejaremos al hilo padre como el programa principal, y al hijo como la thread que este lance en su ejecución.
Primero, se define el tipo de dato $fibo\_t$ que es un *unsigned long long int*. Este es el tipo de dato con más capacidad que se pudo encontrar pudiendo guardar un número tan grade como 18,446,744,073,709,551,615.
Pese a ser un número grande, este pone una limitación a nuestro programa puesto que el $Fibonacci(95)$ supera dicha cantidad. Por ende, se ha definido la macro $FIBO\_MAX$ cuyo valor por defecto es de 93 y servirá para limitar la cantidad de números Fibonacci que se pueden generar.
Así mismo, se ha creado una variable global $fibo\_numbers$ que es el array dinámico (un puntero $fibo_t*$) que almacenará los números Fibonacci generados en ejecución.
También se implementó una función $fibonacci_init$ que verifica si el número aportado está dentro de los límites del programa. De ser así, se genera el espacio en el heap para el array dinámico y se inicializa todos los valores a 0. Cabe aclarar que el índice 1 se setea con el valor 1 para cumplir con la regla inicial de Fibonacci.
La función $m\_fibonacci$ es recursiva y se encarga de calcular los números de Fibonacci con un algoritmo memorizado. Esto es, si el número Fibonacci n que deseo ya se encuentra en el array $fibo\_numbers$ solo retorno el valor, caso contrario procedo a calcularlo con las dos llamadas recursivas para $n-1$ y $n-2$ (usando la formula brindada).
La función $fibonacci$ es la encargada de llenar los n nnumeros de Fibonacci en el array $fibo\_numbers$ al llamar al método $m\_fibonacci$.
Por último, la función principal *main* se encarga de validar si al programa se le pasa un argumento para ejecutarse correctamente. Crea el hilo *child* que utilizará a la función $fibonacci$ para calcular los números. Una vez terminado el calculo de dicah thread se presentan los números en pantalla y se procede a liberar la memoria reservada para el array dinámico (considerando siempre las buenas prácticas).

Resultados

Los resultados que se obtuvieron con el programa fueron los siguientes:

Prueba 1:

Para 5 números.

```
Los primeros 5 numeros Fibonacci:
0
1
1
2
3

PI aproximado    3.138772
```

.

Prueba 2:

Para 20 números

```
Los primeros 20 numeros Fibonacci:
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
```

Ejercicio 4

La implantación se hizo en el archivo Ejercicio4