Procesos e hilos: cálculo de π

Departamento de Automática

Universidad de Alcalá

/gso>

Índice

Descripción general

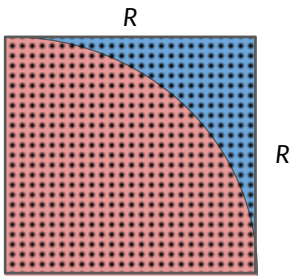
- Objetivo principal: obtener una aproximación al valor de π empleando el método Monte Carlo
- Otros objetivos:
  - Utilizar la biblioteca POSIX para la creación de hilos
  - Aplicar primitivas de sincronización básicas entre hilos

Especificación de la práctica

Implementar un programa que obtenga una aproximación del valor de π empleando el método Monte Carlo. El programa deberá utilizar hilos para paralelizar el cálculo, y mecanismos de sincronización para controlar su ejecución y el acceso a las variables compartidas

Método Monte Carlo para aproximar el valor de π

- Es un método basado en probabilidades estadísticas:
  1. Se parte de un cuadrante circular de radio R inscrito en un cuadrado de lado R
  2. A continuación se proyectan un conjunto de puntos aleatorios de forma uniforme por la superficie del cuadrado
  3. Se contabilizan los puntos que caen dentro de la sección circular
  4. La relación entre el número de puntos dentro de la sección (*nsec*) y el número de puntos totales (*N*) es una aproximación a la relación entre áreas:

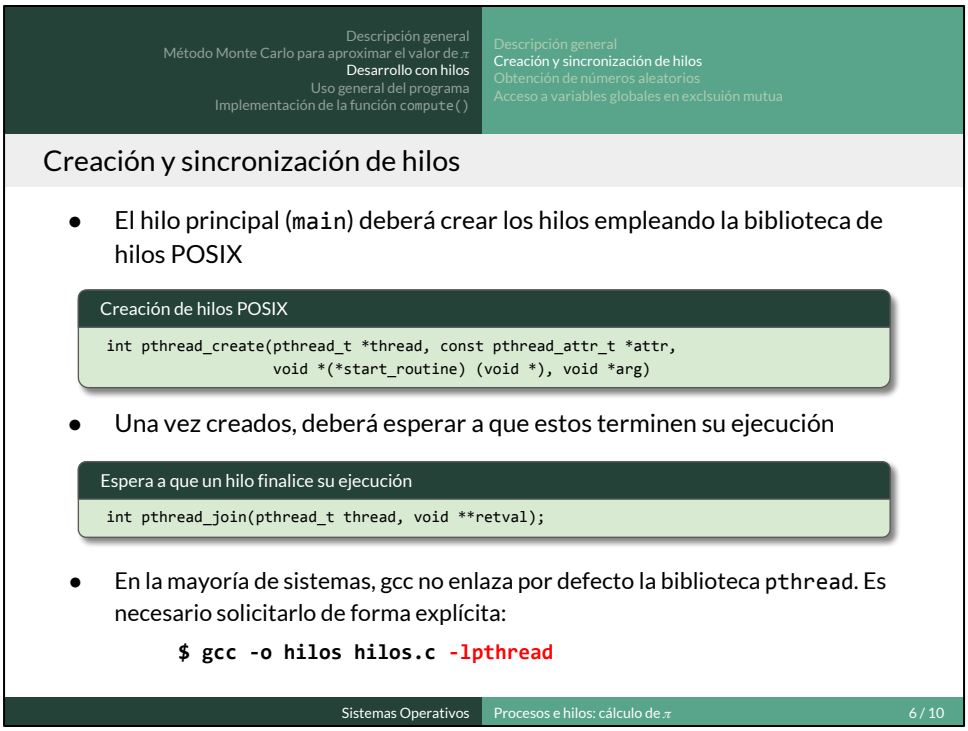


$$\frac{nsec}{N} \cong \frac{\text{área de la sección circular}}{\text{área del cuadrado}} = \frac{(\pi * R^2) / 4}{R^2} = \frac{\pi}{4}$$

Desarrollo con hilos

Descripción general

- El programa deberá emplear hilos para contabilizar los puntos
  - Cada hilo tendrá asignado de antemano un número de puntos
- Los hilos deberán realizar las siguientes operaciones:
  1. Obtener las coordenadas aleatorias de cada punto
  2. Comprobar si el punto (x, y) está dentro de la sección circular de radio $R = 1 \Rightarrow x2 + y2 \leq 1$.
     - Si es así, incrementar el valor de una variable contador
  3. Repetir los pasos 1 y 2 por cada uno de los puntos asignados
  4. Por último, deberá sumar la variable contador local del hilo a una variable contador global
     - El acceso a la variable global estará protegido por un semáforo



El ejemplo de programa que se podría realizar es uno que cree directamente un conjunto de hilos. Estos hilos deberán realizar las siguientes operaciones:

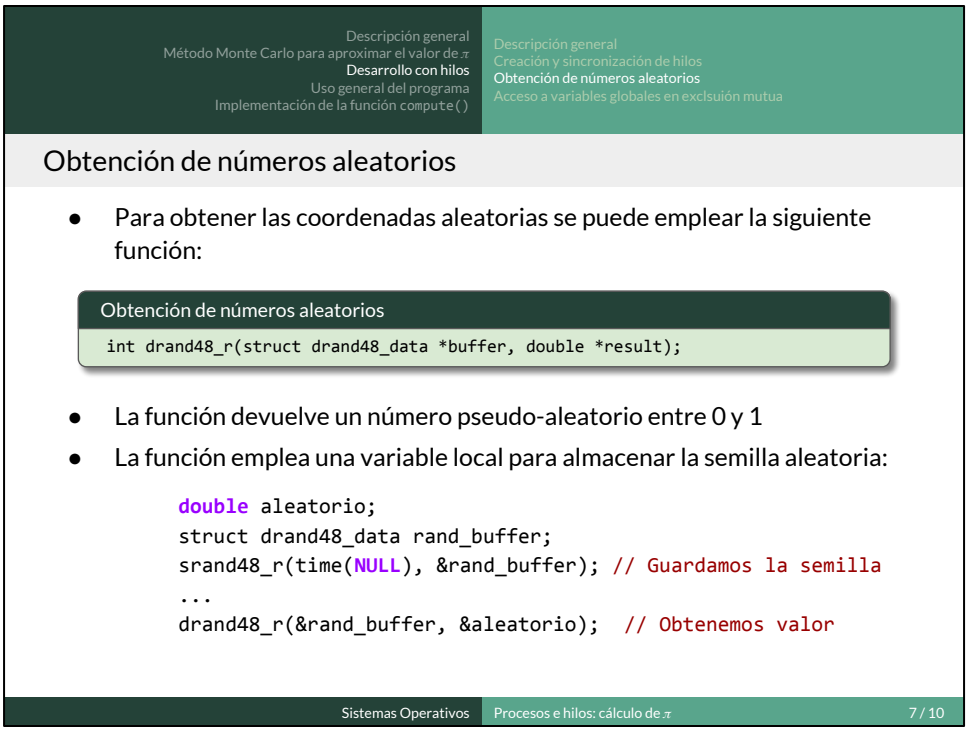
1) Incrementar el valor de una variable global en una unidad
2) Realizar una espera por un número de microsegundos aleatorios empleando la función `usleep()`
3) Decrementar el valor de la misma variable global en una unidad
4) Realizar una segunda espera por un número de microsegundos aleatorios empleando la función `usleep()`
5) Terminar su ejecución empleando la función `pthread_exit()`

La función principal deberá crear los hilos empleando la función `pthread_create()` y a continuación esperar a que estos terminen su ejecución con la función `pthread_join()`. Una vez hayan terminado de ejecutarse todos los hilos, el hilo principal deberá mostrar el contenido de la variable global.

En esta primer versión, se podrá comprobar que, en la mayoría de ejecuciones, el valor final de la variable global es distinto de 0. Esto es debido a que se producen condiciones de carrera al acceder a la variable global. Es necesario proteger el acceso mediante semáforos. Los semáforos se explican en una transparencia posterior.

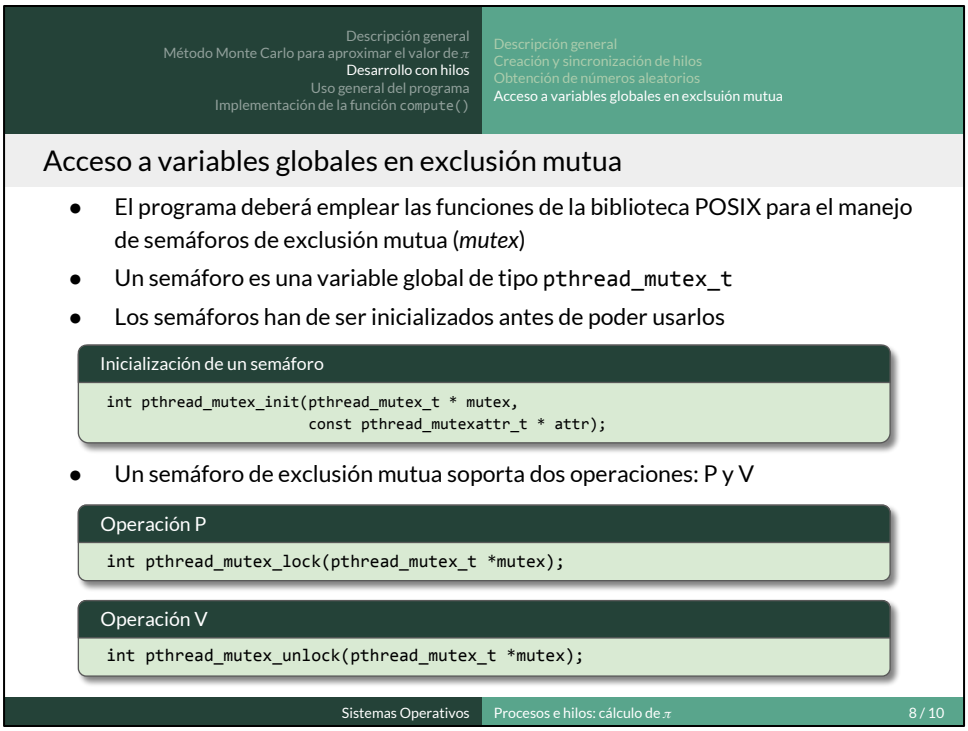
NOTA: Es probable que si ejecutamos el programa en una máquina virtual, el valor resultante sea cero en la mayoría de los casos, incluso sin usar semáforos. Esto es

debido a que, dentro de la máquina virtual, la granularidad temporal que ofrece la implementación de usleep() no es lo suficientemente pequeña como para forzar que las interrupciones salten en los instantes necesarios para que se produzcan las condiciones de carrera.



Los números aleatorios generados por la función `drand48_r()` se obtienen mediante la aplicación de un algoritmo a un valor inicial llamado semilla (*seed*). En llamadas sucesivas a la función, se aplicará el mismo algoritmo sobre el resultado anterior obtenido (la semilla inicial solo hay que establecerla **una única vez antes de obtener el primer número aleatorio**). En el caso de la función `drand48_r()` la semilla se guarda en una variable local a la función de tipo `struct drand48_data`.

Si la función aleatoria se inicializa siempre con la misma semilla, producirá siempre los mismos valores “aleatorios”. Por eso, es conveniente emplear una semilla “aleatoria” para inicializar el algoritmo, de tal forma que dos ejecuciones sucesivas del mismo programa empleen distintas semillas. Una forma sencilla de hacerlo es emplear como semilla el tiempo actual del sistema. Mediante la función `time()` podemos obtener el número de segundos transcurridos desde La Época. Suponiendo que el reloj del sistema funciona correctamente, dos llamadas sucesivas al mismo programa resultarán en valores distintos del tiempo de sistema y, por tanto, utilizarán dos semillas distintas.



En la función `pthread_mutex_init()` se pueden establecer los atributos iniciales del semáforo a través del segundo parámetro de la función. Si al realizar la llamada de inicialización el segundo parámetro es `NULL`, el semáforo se inicializará con los valores por defecto.

Aquí se puede completar el programa propuesto en la transparencia 5, protegiendo el acceso a la variable compartida con un semáforo de exclusión mutua.

Uso general del programa

- El programa principal recibe como argumentos el número de puntos y el número de hilos

```
$ ./computepi PUNTOS HILOS
```

- El código que analiza los argumentos ya se encuentra implementado en el esqueleto de la práctica
  - El número máximo de puntos que el programa acepta es 500,000,000
- Una vez analizados los argumentos, la función `main()` llama a la función `compute()`

Prototipo de la función compute()

```
void compute(int npoints, int nthreads);
```

- La función recibe como parámetros el número de puntos (`npoints`) y el número de hilos (`nthreads`)

Implementación de la función `compute()`

- La función `compute()` deberá realizar las siguientes operaciones:
  1. Inicialización del semáforo de exclusión mutua `mutex`
  2. Creación de todos los hilos del programa
     - El número de hilos dependerá del valor pasado como parámetro a la función compute()
     - Los hilos deberán ejecutar la función `thread_function()`
     - Cada hilo recibirá como argumento el número de puntos que deberá calcular
     - Si la división del número de puntos entre el número de hilos no es exacta, el último hilo creado calculará el resto de puntos
  3. Esperar a que todos los hilos terminen su ejecución
  4. Calcular el valor estimado de π e imprimirlo por pantalla
  5. Destruir el semáforo con la función `pthread_mutex_destroy()`

El parámetro inicial que se le pasa a los hilos durante la inicialización es de tipo **`void`** *, esto es, un puntero genérico. Este puntero genérico contendrá la dirección donde estará el entero que contiene el número de puntos asignados.

Para esperar por los hilos, el programa deberá emplear la función `pthread_join()`.