Sistemas Operativos 2/2018
Laboratorio 2

Profesores:
Cristóbal Acosta (cristobal.acosta@usach.cl)
Miguel Cárcamo (miguel.carcamo@usach.cl)
Fernando Rannou (fernando.rannou@usach.cl)
Ayudantes:
Marcela Rivera (marcela.rivera.c@usach.cl)
Esteban Alarcón (esteban.alarcon.v@usach.cl)

I. Objetivos Generales

Este laboratorio tiene como objetivo aplicar los conceptos de concurrencia de *pthread*, tales como hebras, semáforos y barreras, dentro del contexto del problema de transferencia radiativa.

II. Objetivos Específicos

1. Conocer y usar las funcionalidades de `getopt()` como método de recepción de parámetros de entradas.
2. Comprender e implementar el proceso de transferencia radiativa.
3. Utilizar recursos de *pthread* para sincronizar las hebras y proteger los recursos compartidos.
4. Conocer y practicar uso de makefile para compilación de programas.

III. Conceptos

III.A. Concurrencia y Sincronización

Cuando dos o más tareas (procesos o hebras) comparten algún recurso en forma concurrente o paralela, debe existir un mecanismo que sincronice sus actividades.

De no existir una sincronización, es posible sufrir una corrupción en los recursos compartidos u obtener soluciones incorrectas.

III.B. Sección Crítica

Porción de código que se ejecuta de forma concurrente y podría generar conflicto en la consistencia de datos debido al uso de variables globales.

III.C. Mutex

Provee exclusión mutua, permitiendo que sólo una hebra a la vez ejecute la sección crítica.

III.D. Hebras

Los hilos POSIX, usualmente denominados pthreads, son un modelo de ejecución que existe independientemente de un lenguaje, además es un modelo de ejecución en paralelo. Éstos permiten que un programa

controle múltiples flujos de trabajo que se superponen en el tiempo.

Para poder utilizar hebras, es necesario incluir la librería **pthread.h**. Por otro lado, dentro de la función main, se debe instanciar la variable de referencia a las hebras, para esto se utiliza el tipo de dato **pthread_t** acompañado del nombre de variable.

Luego el código que ejecutarán las hebras se debe construir en una función de la forma:

```
void * function (void * params)
```

La cual recibe parámetros del tipo void, por lo cual es necesario castear el o los parámetros de entrada para así poder utilizarlos sin problemas.

Algunas funciones para manejar las hebras son:

- `pthread_create`: función que crea una hebra. Recibe como parámetros de entrada:
  - La variable de referencia a la hebra que desea crear.
  - Los atributos de éste, los cuales no es obligación de modificar, por lo que en caso de no querer hacerlo, simplemente se deja NULL.
  - El nombre de la función que la hebra ejecutará (la cual debe cumplir con la descripción antes mencionada)
  - Por último, los parámetros de entrada (de la función que se ejecutará) previamente casteados.

  Un ejemplo sería:

  ```
  while(i < numeroHebras)
  {
      pthread_create(&hilo[i], NULL, escalaGris, (void *) &structHebra[i].id);
      i++;
  }
  ```

- `pthread_join:` función donde la hebra que la ejecuta, espera por las hebras que se ingresan por parámetro de entrada.

  Un ejemplo sería:

  ```
  while(i < numeroHebras)
  {
      pthread_join(hilo[i],NULL);
      i++;
  }
  ```

- `pthread_mutex_init:` función que inicializa un mutex, pasando por parámetros la referencia al mutex, y los atributos con que se inicializa.

  Para inicializar la estructura se utiliza:

  ```
  while(i < numeroHebras)
  {
      pthread_mutex_init(&MutexAcumulador, NULL);
      i++;
  }
  ```

  Donde `MutexAcumulador`, es una variable (de tipo `pthread_mutex_t`) que representa un mutex, el cual permitirá implementar exclusión mutua a una sección crítica que será ejecutada por varias hebras.

- **pthread_mutex_lock:** entrega una solución a la sección crítica. Ésta recibe como parámetros la variable que se desea bloquear para el resto de hebras. Un ejemplo de uso sería:

```
pthread_mutex_lock(&MutexAcumulador);
```

  Cabe destacar que la primera hebra en ejecutar **pthread_mutex_lock** podrá ingresar a la sección crítica, el resto de hebras quedarán bloqueadas a la espera de que se libere el mutex.

- **pthread_mutex_unlock:** permite liberar una sección crítica. Ésta recibe como parámetro de entrada, la variable que se desea desbloquear. Su implementación es la siguiente:

```
pthread_mutex_unlock(&MutexAcumulador);
```

  Por otro lado, existen casos donde se debe esperar que se completen varias tareas antes de que pueda continuar una tarea general, para solucionar esto se puede usar la sincronización de barrera . Los hilos **POSIX** especifican un objeto de sincronización llamado barrera, junto con funciones de barrera.

  Las funciones crean la barrera, especificando el número de hebras que se sincronizan en ésta, luego se configuran para realizar tareas y esperan en la barrera hasta que todos las hebras alcanzan la barrera. Cuando llega el último subproceso a la barrera, todos los subprocesos reanudan la ejecución. Para hacer uso de esto, se necesitan las siguientes funciones:

- **pthread_barrier_init:** permite asignar recursos a una barrera e inicializar sus atributos. Su implementación es:

```
pthread_barrier_init(&rendezvous, NULL,NTHREADS);
```

  Donde **rendezvous** es una variable de tipo pthread_barrier_t, **NULL** indica que se utilizan los atributos de barrera predeterminados (se pide usar estos atributos, no debiese ser necesario modificarlos) y **NTHREADS** es la cantidad de hebras que deben esperar en la barrera.

- **pthread_barrier_init:** permite sincronizar los hilos en una barrera especificada. El hilo de llamada bloquea hasta que el número requerido de hilos ha llamado a pthread_barrier_wait() especificando la barrera. Un ejemplo de su uso:

```
funcion1();
pthread_barrier_wait(&rendezvous);
funcion2();
```

  Donde **funcion1** y **funcion2** son funciones que deben ejecutar las hebras pero de manera sincronizada, es decir, no se debe ejecutar funcion2 hasta que las n hebras (indicadas en barrier_init) ejecuten la funcion 1.

- **pthread_barrier_destroy:** cuando ya no se necesita una barrera, se debe destruir. SU implementación es:

```
pthread_barrier_destroy(&rendezvous);
```

IV. Proceso de transferencia radiativa

La **transferencia radiativa** es el fenómeno físico mediante el cual se transfiere energía en la forma de energía electromagnética. Por ejemplo, en el caso de los discos protoplanetarios, la energía lumínica (fotones de luz) de la estrella interacciona con el medio circundante, a través de dos procesos físicos: la absorción y la difusión.

IV.A. Lanzamiento de un fotón

Cada fotón es lanzado desde una estrella, representada mediante un punto del plano, con una distancia a recorrer (l) y una dirección (μ), ambas aleatorias. El vector μ debe ser un vector unitario en el plano (x,y).

IV.B. Eventos: absorción y difusión

Una vez recorrida la distancia l, puede ocurrir un evento de absorción o difusión. Cuál evento ocurre depende del *albedo* a, el cual es simplemente la probabilidad que un fotón sea dispersado. Para simplicidad, a tendrá un valor de 0.5 (ambos eventos tienen la misma probabilidad de ocurrir).

IV.B.1. Absorción

En caso de realizar este evento, el fotón debe agregar energía en la celda donde está ubicado. El valor de la energía a sumar es 1 unidad de energía. Posteriormente, se busca otro l y μ, nuevamente de forma aleatoria.

IV.B.2. Difusión

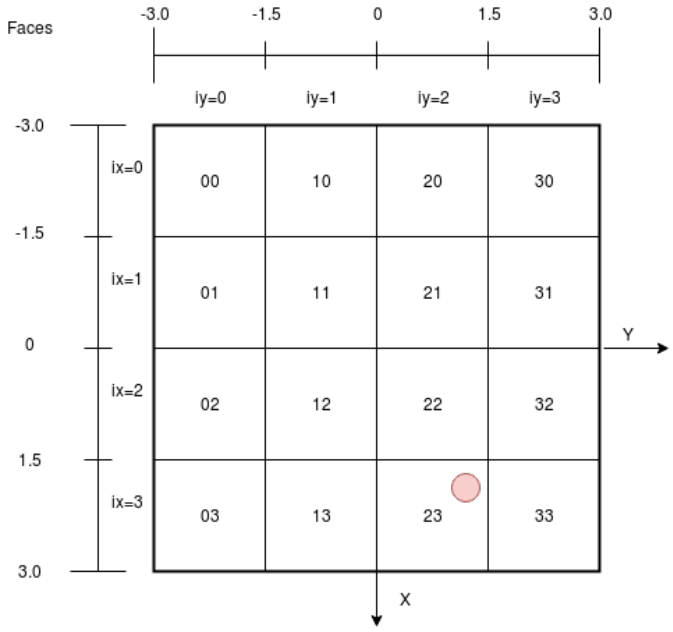
Si el evento escogido es difusión, simplemente se busca aleatoriamente un l y μ.

IV.C. Fin de un fotón

Con el objetivo de saber en qué momento el fotón finaliza su movimiento, se entrega un valor L, el cual es la distancia total que puede recorrer cada fotón. Es importante destacar que el fotón automáticamente acaba su trayectoria si escapa de la grilla.

V. Conceptos para la programación

Para definir el espacio a simular y la grilla, deben entregarse los valores Y (dimensión eje y), X (dimensión eje y) y un delta Δ (distancia continua entre límites de cada celda). La siguiente imagen muestra un espacio y grilla con $X = 4$, $Y = 4$, $\Delta = 1.5$.



El movimiento del fotón es de forma **continua**, y a su vez, esa posición continua equivale a una posición de la grilla (x,y). En la imagen anterior se supone un fotón (círculo rojo), el cual dada una dirección y un l se calcula la posición (1.67, 1.44); esa posición continua debe asociarse a la posición de la grilla x=3, y=2. Por lo tanto, si ocurre un evento de absorción, el fotón debe depositar energía en tal posición de la grilla.

La matriz de la grilla debe ser expresada como [Y][X], siendo Y la dimensión del eje y, y X la dimensión del eje x.

IMPORTANTE: En congruencia con el propósito de este laboratorio, deben lanzarse los fotones de forma concurrente, y cada posición de la grilla debe ser una sección crítica, es decir, no puede haber más de un fotón en la misma posición de esta. Para ello, deben usarse las herramientas que provee *pthread*.

VI. Parámetros de entrada

Los parámetros que deben ser entregados por la terminal son:

- -n: cantidad de fotones.
- -L: distancia MÁXIMA que puede recorrer el fotón.
- -X: Dimensión del eje x de grilla.
- -Y: Dimensión del eje y de grilla.
- -d: Delta, distancia entre cada límite de la grilla.
- -b: Permite ver el seguimiento de fotones en la grilla.

VII. Salida del programa

Como resultado, debe entregar un archivo de texto con los valores de energía en cada posición de la grilla, de la siguiente forma:

```
<valor [0][0]>
<valor [0][1]>
...
<valor [0][X-1]>
<valor [1][0]>
<valor [1][1]>
...
<valor [1][X-1]>
...
<valor [Y-1][X-1]>
```

VIII. Entregables

Debe entregarse un archivo comprimido que contenga al menos los siguientes archivos:

1. *Makefile*: archivo para make que compile los programas.
2. dos o mas archivos con el proyecto (*.c, *.h)

El archivo comprimido debe llamarse: RUTESTUDIANTE1_RUTESTUDIANTE2.
Ejemplo: 19689333k_189225326.zip

IX. Fecha de Entrega

Jueves 13 de Diciembre, hasta las 23:55 hrs.