Prácticas 9 y 10

Barreras en multiprocesadores

1. Objetivos

El objetivo de la presente práctica es estudiar, mediante el simulador de multiprocesadores, el acceso a variables compartidas de sincronización, en concreto se estudia la implementación de una barrera y su rendimiento. También se pone de manifiesto el problema de la falsa compartición.

Para conseguir estos objetivos el estudiante tendrá que desarrollar varios programas de prueba para hacer frente a cada uno de los problemas expuestos.

2. Desarrollo

En la sesión anterior se aprendió la utilización del simulador limes obteniendo resultados de la ejecución de diversas aplicaciones de prueba. En esta sesión se le pide al alumno que genere sus propias aplicaciones para poner de manifiesto la influencia de las cachés en sistemas multiprocesadores. Estas aplicaciones paralelas se realizan utilizando la serie de macros de paralelización que se explican a continuación.

2.1 Creación de aplicaciones paralelas: macros ANL

Las aplicaciones paralelas que se van a realizar están basadas en la creación de procesos o hilos a partir de un proceso padre. Los hilos comparten el espacio de direcciones y variables tanto del padre como del resto de hilos, por lo que se trata de procesos de tipo *sproc* más que de procesos creados con *fork*.

Para la administración de estos hilos y la realización de programas paralelos en general, existen una serie de macros de compilación llamadas ANL (*Argonne National Lab*). Las tres macros que se encargan de controlar los hilos son las siguientes:

- `CREATE(nombre_funcion)` Esta macro crea un hilo pasando a ejecutar la función indicada por `nombre_funcion`. Cuando la función termina el hilo termina. Esta función no admite argumentos y no devuelve nada.
- `WAIT_FOR_END(numero)` Esta macro entra en un bucle en espera de que acaben un numero `numero` de hilos, pero de hecho este argumento se ignora puesto que esta macro se espera hasta que hayan terminado todos los procesos hijos. Esta macro la suele invocar el proceso padre al final de una fase paralela y antes de entrar en la siguiente.
- `CLOCK(variable)` Esta macro devuelve en la `variable` el número de ciclos de reloj de tiempo real desde que se inició el programa. De esta forma podemos medir con exactitud el tiempo de ejecución de las diferentes partes del programa. La variable es de tipo entero sin signo (`unsigned int`).

Con estas macros básicas ya se puede realizar casi cualquier programa de ejecución paralela. Sin embargo, el conjunto de macros ANL incluye muchas más. En concreto existen también macros para implementar cerrojos que son el mecanismo básico de sincronización en multiprocesadores de memoria compartida. Las macros para el manejo de cerrojos son las siguientes:

- `LOCKDEC(nombre_cerrojo)` Esta macro se pone en la parte de declaraciones del programa y sirve para declarar un cerrojo con el nombre `nombre_cerrojo`.
- `LOCKINIT(nombre_cerrojo)` Esta macro inicializa el cerrojo `nombre_cerrojo` a un estado libre para que cualquiera lo pueda coger. Normalmente es el proceso padre el encargado de ejecutar esta macro antes de lanzar ningún otro proceso.
- `LOCK(nombre_cerrojo)` Sirve para tomar posesión del cerrojo `nombre_cerrojo`. La ejecución se para aquí hasta que se consigue el cerrojo.
- `UNLOCK(nombre_cerrojo)` Hace lo contrario de la anterior, es decir, libera el cerrojo para que otros lo puedan adquirir.

Junto a estas macros existen otras, como las propias de las barreras, que son las siguientes:

- `ALOCKDEC(nombre_cerrojo,tamanyo_vector)` Esta macro se pone en la parte de declaraciones del programa y sirve para declarar un cerrojo con el nombre `nombre_cerrojo` que protege a un vector de tamaño `tamanyo_vector`.

- `ALOCKINIT(nombre_cerrojo,tamanyo_vector)` Esta macro inicializa el cerrojo `nombre_cerrojo` a un estado libre para que cualquiera lo pueda adquirir. Este cerrojo protege a un vector de tamaño `tamanyo_vector`. Normalmente es el proceso padre el encargado de ejecutar esta macro antes de lanzar ningún otro proceso.
- `ALOCK(nombre_cerrojo,elemento)` Sirve para tomar posesión del cerrojo `nombre_cerrojo` que protege al elemento `elemento` de un vector. La ejecución se para aquí hasta que se consigue el cerrojo.
- `AULOCK(nombre_cerrojo,elemento)` Hace lo contrario de la anterior, es decir, libera el cerrojo para que otros lo puedan adquirir. (Al parecer hay un fallo en las fuentes y en el nombre falta una N, hubiera sido más lógico llamar a esta macro `AUNLOCK`, ya que además es el nombre que recibe en el manual de Limes.)

También hay otras macros como son MAIN_ENV, EXTERN_ENV, MAIN_INITENV y MAIN_END que se usan para inicializar diferentes parámetros y estructuras de las macros ANL, pero que en el caso de Limes no son necesarias.

Una descripción más detallada de todas estas macros se puede encontrar en un pequeño manual que se encuentra en la página web del laboratorio. También el propio manual de Limes trae una breve descripción de las macros.

2.2 Pasos a seguir para crear nuestra propia aplicación

Para crear nuestra propia aplicación paralela y simularla bajo Limes, lo único que hay que hacer es copiar el fichero *makefile* de alguna aplicación de las que vienen con Limes (por ejemplo *fft*) y modificar las variables `APPCS`, donde pondremos el nombre de nuestro fichero fuente, y `TARGET`, donde se pondrá el nombre del fichero ejecutable que generará.

Se pueden modificar otras variables del fichero *makefile*. Es especialmente importante asegurarnos de que existe seleccionado algún protocolo de coherencia de caché, porque si se simula con la memoria ideal es posible que no se vea nada.

Esta nueva aplicación que hagamos la podemos poner en cualquier directorio, por lo que resulta bastante interesante que se ponga fuera del árbol de directorios de Limes. De esta manera se puede borrar Limes sin que se pierdan nuestros datos.

A continuación se muestra una aplicación de ejemplo en la que se lanzan `PROC_MAX` procesos (incluido el padre) que se reparten un vector sobre el que realizan unas operaciones de carga.

```
#include <stdio.h>
#include <math.h>
#include <unistd.h>

#define LONGIT  10000    /* Longitud del vector    */
#define PROC_MAX   64    /* Numero de procesadores */

MAIN_ENV     /* No es necesario con Limes */

unsigned int inicio,final;
double vector[LONGIT];  /* Vector compartido                      */
int PrId;               /* Identificador de proceso               */
LOCKDEC(PrIdLock)       /* Cerrojo usado para la variable anterior */

/* Este es la rutina ejecutada por cada hilo lanzado */
void Proceso(void)
{
int i,trozo,pr;
  LOCK(PrIdLock);
  pr=PrId;         /* Cada identificador de proceso debe ser unico */
  PrId++;          /* por eso se protege con un cerrojo            */
  UNLOCK(PrIdLock);
  trozo=LONGIT/PROC_MAX;
  for (i=trozo*pr;(i<trozo*(pr+1)) & (i<LONGIT) ;i++)
  {
     vector[i]=M_PI;
     vector[i]=sqrt(vector[i]);       /* Calculos tontos para gastar tiempo */
     vector[i]=vector[i]*vector[i]+1;
  }
}

main()
{
int i;
```

```
  MAIN_INITENV();     /* No es necesario con Limes          */
  CLOCK(inicio);      /* tomamos cuenta del tiempo inicial  */
  PrId=0;             /* El padre es el proceso # 0         */
  LOCKINIT(PrIdLock); /* Inicializa cerrojo                 */
  for (i=1;i<PROC_MAX;i++)
  {
     CREATE(Proceso); /* Este bucle lanza PROC_MAX-1 hilos  */
  }
  Proceso();          /* El padre tambien hace su faena     */
  WAIT_FOR_END(PROC_MAX-1);  /* Se espera a que terminen los hijos */
  CLOCK(final);       /* Medimos el tiempo al terminar      */
  printf("%14ld\n",final-inicio);
  MAIN_END;           /* No es necesario con Limes          */
}
```

Este programa será la base para los programas que se piden y se puede bajar de la página web del laboratorio.

3. Trabajo a realizar

3.1 Programa con barrera "sense-reversing"

Se parte de una matriz de 4 filas y *n* columnas (donde *n* es el número de procesadores que hay). Se deben realizar cuatro iteraciones sobre el vector:

1. En la primera iteración se inicia cada elemento de la primera fila con el número del procesador que se ocupa de dicho elemento. Es decir: `vector[0][pr-1]=pr;` (el índice de la columna es *pr-1* puesto que los procesadores se deben numerar a partir de 1).
2. En la iteración *i*, se le asigna a cada elemento de la fila *i* el elemento de la fila *i-1* pero de la columna anterior. Si la columna es la primera se le asigna la última (rotación a derechas). Es decir: `vector[i][pr-1]=vector[i-1][(pr-2+n)%n];`

Para realizar esto se lanzarán todos los procesos y cada uno por separado debe realizar 4 iteraciones. Para que funcione bien y se obtenga el resultado deseado, todos los procesadores deben terminar una etapa antes de empezar la siguiente. Por lo tanto, al final de cada iteración pondremos una barrera.

Una vez los procesos hayan terminado se debe sacar por pantalla el vector de la última fila para ver si todo ha ido bien. Si no se implementa una barrera entre iteración e iteración el resultado no será el esperado (se puede probar para ver que es cierto).

La primera implementación de este programa se hará mediante una barrera simple de tipo *"sense-reversing"*. Es una de las implementaciones de barrera más sencillas que consiste simplemente en tener un contador protegido con cerrojo y una variable compartida que indica la liberación. Cada vez que se entra en la barrera la comparación de la variable de liberación cambia para evitar que un proceso vuelva a la barrera antes de que el resto haya salido. Este cerrojo ha sido explicado en la clase de teoría, donde además se ha incluido el código, por lo que no debe presentar mayor complicación.

3.2 Barrera en árbol

Esta barrera también se ha explicado en clase pero es bastante más difícil de implementar que la anterior. En principio es una barrera que ofrece un mayor paralelismo y por lo tanto se debe esperar un mayor rendimiento.

El programa a realizar será el mismo que en la sección anterior, lo único que cambia es el tipo de barrera. Para la implementación haremos uso de un vector de contadores protegido por un vector de cerrojos. También la liberación consiste en un vector. Este vector de contadores y de liberación tendrá una disposición en forma de árbol tal como se ha visto en clase de teoría.

A efectos prácticos, lo único que necesita un proceso es saber quien es su padre y sus hijos dentro del árbol. Si suponemos un árbol binario siendo la raíz el procesador 1 y rellenamos el árbol de forma consecutiva por niveles, tendremos que el padre del proceso *i* no es más que $\lfloor i/2 \rfloor$ y sus dos hijos son *i**2 e *i**2+1, siempre que estos hijos sean menores que *n* (número de procesadores).

El algoritmo de la barrera es más o menos así:

1. Al inicio cada proceso pone su contador y su variable de liberación a cero.
2. Cuando un proceso "hoja" (sin hijos) entra en la barrera debe incrementar el contador de su padre y se espera hasta la liberación.

3. Cuando un proceso con hijos entra en la barrera se espera a que su contador alcance el número de hijos que tiene.
4. Cuando el contador de un proceso con hijos alcanza el número de hijos que tiene, entonces incrementa el contador de su padre, pone el suyo propio a cero y se espera hasta que sea liberado.
5. Cuando el contador del proceso raíz (proceso 1) es igual al número de hijos que tiene, entonces pone a uno la variable de liberación de cada uno de sus hijos saliendo de la barrera.
6. Cuando un proceso detecta que su variable de liberación está a uno, pone a cero la suya y a uno la de sus hijos saliendo de la barrera.

Hay que prestar atención al hecho de que tanto la liberación como el contador son vectores, por lo que se puede tener el problema de la **falsa compartición**. Se debe por tanto implementar con vectores cuyos elementos se encuentren separados para que no coincidan en la misma línea de caché.

3.3 Falsa compartición

Este último apartado es opcional y se iniciará su implementación cuando se hayan terminado completamente los anteriores. Se pide la realización de un programa que ponga de manifiesto el problema de la falsa compartición. La falsa compartición ocurre cuando dos o más procesadores acceden a la misma línea de caché aunque en realidad están accediendo a variables distintas. Esto se puede dar con frecuencia en el acceso a vectores y matrices donde varios procesadores pueden estar accediendo a diferentes datos del vector, pero al encontrarse cerca comparten la misma línea de caché.

El programa visto al principio del boletín puede servir como punto de partida para realizar el programa que se pide. Se realizarán las siguientes modificaciones:

1. Se debe incluir un bucle para que repita el experimento desde un procesador hasta 64 en potencias de dos. Para esto se tiene que sustituir la constante `PROC_MAX` por una variable compartida que tendrá el número de procesadores (procesos o hilos) utilizados. La salida se debe volcar a un fichero para su posterior tratamiento (hacer gráfica de tiempo de ejecución en función del número de procesadores).
2. Para que el fichero de datos de salida nos salga lo más limpio posible (sin todos los mensajes propios del simulador) podemos ejecutar el programa enviando los mensajes propios del simulador a la salida de error. Esto se hace utilizando la opción `-- -e`.
3. Crear a continuación otro proceso que sustituya al anterior y que reparta el vector de forma entrelazada entre todos los procesadores. En el proceso del ejemplo los elementos se reparten de manera continua, de manera que se minimicen los efectos de la falsa compartición. Se pide hacer este otro proceso completamente entrelazado para medir la influencia de la falsa compartición en el rendimiento total. Y todo siempre en función del número de procesadores.
4. Todo lo anterior debe probarse al menos con protocolo de invalidación (*MESI* o *berkeley*) y otro de actualización (*dragon*). De esta manera se pueden comparar ambos.

Adicionalmente se pueden hacer las gráficas con un número mayor de procesadores, se puede modificar el programa para que directamente se muestre el aumento del rendimiento (*speed-up*), se pueden probar diferentes tamaños de caché, algún otro protocolo de caché diferente de los que se piden, etc.

3.4 Conclusiones

De esta práctica se deben extraer numerosas conclusiones sobre el comportamiento de la caché en sistemas multiprocesadores. También se sacarán conclusiones sobre la forma correcta de realizar programas paralelos en sistemas multiprocesadores basados en bus común. Estas conclusiones reforzarán las ya obtenidas en la práctica anterior para poder contestar correctamente el examen que se plantee.