Sistemas Operativos I - LCC - 2015
Práctica 2
Concurrencia y paralelismo con OpenMP y POSIX Threads

Introducción a OpenMP

OpenMP es una interfaz de programación de aplicaciones para la programación multiproceso de memoria compartida en múltiples plataformas. Permite añadir concurrencia a los programas escritos en C, C++ y Fortran sobre la base del modelo de ejecución fork-join. En C/C++ se implementa como directivas al compilador con `#pragma`. Para utilizar OpenMP en gcc debe compilar sus programas `-fopenmp`.

Introducción a POSIX Threads

POSIX Threads (o PThreads) es un estándar POSIX para la creación, manejo y destrucción de hilos. POSIX Threads ofrece métodos para sincronizar los hilos (locks, semáforos, variables de condición, etc). Para compilar un programa C usando POSIX threads debe decir al gcc que linkee la librearía POSIX Threads con la opción `-pthread`.

1. El Jardín Ornamental

- Convierta la implementación Cilk del jardín ornamental dada aquí en una implementación en paralelo usando OpenMP, sin proteger la región crítica.
- Implemente el algoritmo de Lamport para N procesos para proteger la región crítica.
- Aun si el punto anterior está bien implementado, el programa puede seguir fallando al ejecutarse en una máquina con varios núcleos. Explique cómo la memoria caché puede generar este problema.

- Utilice la instrucción de assembler *mfence* para remediarlo.

2. Multiplicación de Matrices

Implemente en OpenMP la multiplicación de dos matrices dadas A y B en paralelo.

Puede encontrar una implementación secuencial aquí.

```
#include <stdio.h>
#include <stdlib.h>

#define N 200

int A[N][N],B[N][N],C[N][N];

void mult(int A[N][N], int B[N][N], int C[N][N])
{
    int i, j, k;
    for (i=0;i<N;i++)
        for (j=0;j<N;j++)
            for (k=0;k<N;k++)
                C[k][i] += A[k][j]*B[j][i];
}

int main()
{
    int i, j;
    for (i=0;i<N;i++)
        for (j=0;j<N;j++) {
            A[i][j] = random() % 10;
            B[i][j] = random() % 10;
        }

    mult(A, B, C);

    return 0;
}
```

- Compare la performance con la solución secuencial para matrices cuadradas de tamaño 200x200, 500x500 y 1000x1000. ¿Qué relación aproximada puede inferir entre los tiempos en uno y otro caso?
- Si se cambia el orden de los índices, ¿se puede mejorar el rendimiento? ¿Por qué?
- Si tuviese que computar la multiplicación de $A \times B^T$, ¿se puede mejorar el rendimiento? ¿Por qué?

3. Búsqueda del Mínimo

Escriba utilizando OpenMP un algoritmo que dado un arreglo de $N = 50000000$ enteros busque el mínimo. Compare la performance con la implementación secuencial usando `omp_get_wtime()` y distinto número de hilos.

4. Quicksort

Aquí puede encontrar una versión secuencial del algoritmo de Quicksort:

```
#include <stdio.h>
#include <stdlib.h>

#define N 1000

void swap(int *v, int i, int j)
{
    int tmp=v[i];
    v[i]=v[j];
    v[j]=tmp;
}

int colocar(int *v, int b, int t)
{
    int i;
    int pivote, valor_pivote;
    int temp;

    pivote = b;
    valor_pivote = v[pivote];
    for (i=b+1; i<=t; i++){
        if (v[i] < valor_pivote){
            pivote++;
            swap(v,i,pivote);
        }
    }
    temp=v[b];
    v[b]=v[pivote];
    v[pivote]=temp;
    return pivote;
}

void QuicksortSeq(int* v, int b, int t)
{
    int pivote;
    if(b < t){
        pivote=colocar(v, b, t);
        QuicksortSeq(v, b, pivote-1);
        QuicksortSeq(v, pivote+1, t);
```

```
    }
}

int main(int argc, char **argv)
{
    int *a,i;

    a = malloc(N*sizeof(int));

    for(i=0;i<N;i++)
        a[i]=random()%N;

    QuicksortSeq(a,0,N-1);

    free(a);
    return 0;
}
```

Un filósofo pensó lo siguiente: dado que en cada paso Quicksort divide el arreglo en dos partes que se pueden ordenar de manera independiente, podemos aprovechar los varios nucleos de una computadora si hacemos que cada una de las partes la ordene un hilo distinto, en paralelo. Realizó la implementación en PThreads que se encuentra aquí:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define N 1000

typedef struct {
    int *v;
    int b, t;
} qsparams;

void swap(int *v, int i, int j)
{
    int tmp=v[i];
    v[i]=v[j];
    v[j]=tmp;
}

int colocar(int *v, int b, int t)
{
    int i;
    int pivote, valor_pivote;
    int temp;

    pivote = b;
    valor_pivote = v[pivote];
```

```
    for (i=b+1; i<=t; i++){
        if (v[i] < valor_pivote){
            pivote++;
            swap(v,i,pivote);
        }
    }
    temp=v[b];
    v[b]=v[pivote];
    v[pivote]=temp;
    return pivote;
}

void *Quicksort(void *p)
{
    qsparams *params = (qsparams *)p;
    int *v = params->v;
    int b = params->b;
    int t = params->t;

    int pivote;
    if(b < t){
        pthread_t t1, t2;
        pivote=colocar(v, b, t);
        qsparams params1, params2;
        params1.v = v;
        params1.b = b;
        params1.t = pivote-1;
        params2.v = v;
        params2.b = pivote+1;
        params2.t = t;
        pthread_create(&t1, 0, Quicksort, (void *)&params1);
        pthread_create(&t2, 0, Quicksort, (void *)&params2);
        pthread_join(t1, NULL);
        pthread_join(t2, NULL);
    }
}

int main(int argc, char **argv)
{
    int *a,i;
    pthread_t t;

    a = malloc(N*sizeof(int));

    for(i=0;i<N;i++)
        a[i]=random()%N;

    qsparams params;
    params.v = a;
    params.b = 0;
```

```
    params.t = N-1;
    pthread_create(&t, 0, Quicksort, (void *)&params);
    pthread_join(t, NULL);

    free(a);
    return 0;
}
```

- El quicksort del programa anterior es más lento que el secuencial para casi cualquier tamaño de arreglo. No solo eso, para arreglos de más de cierto tamaño empieza a fallar aunque la versión secuencial termina normalmente. ¿A qué se debe esto?
- Implemente una versión en PThreads que, aprovechando los varios nucleos de la máquina, ejecute más rápido que la secuencial sin importar el tamaño del arreglo.

5. Lectores y Escritores (Parnas)

El problema de los lectores y escritores consiste en M hilos lectores y N escritores tratando de acceder a un arreglo en memoria compartida con las siguientes restricciones:

1. No puede haber un lector accediendo al arreglo al mismo tiempo que un escritor.
2. Varios lectores pueden acceder al arreglo simultáneamente.
3. Sólo puede haber un escritor a la vez.

El siguiente código es una implementación del problema de los lectores y escritores utilizando POSIX Threads (código fuente aquí):

```
#include <stdio.h>
#include <pthread.h>

#define N 2
#define ARRLEN 1024

int arr[ARRLEN];

void *escritor(void *arg)
{
    int i;
    int num = *((int *)arg);
    for (;;) {
        sleep(random()%3);
        for (i=0; i<ARRLEN; i++) {
            arr[i] = num;
        }
```

```
    }
    return NULL;
}

void *lector(void *arg)
{
    int v, i, err;
    int num = *((int *)arg);
    for (;;) {
        sleep(random()%3);
        err = 0;
        v = arr[0];
        for (i=1; i<ARRLEN; i++) {
            if (arr[i]!=v) {
                err=1;
                break;
            }
        }
        if (err) printf("Lector %d, error de lectura\n", num);
        else printf("Lector %d, dato %d\n", num, v);
    }
    return NULL;
}

int main()
{
    int i;
    pthread_t lectores[N], escritores[N];
    int arg[N];

    for (i=0; i<ARRLEN; i++) {
        arr[i] = -1;
    }
    for (i=0; i<N; i++) {
        arg[i] = i;
        pthread_create(&lectores[i], NULL, lector, (void *)&arg[i]);
        pthread_create(&escritores[i], NULL, escritor, (void *)&arg[i]);
    }
    pthread_join(lectores[0], NULL); /* Espera para siempre */
    return 0;
}
```

En el código no hay ningún tipo de sincronización. Implemente una solución utilizando variables de condición de POSIX (`phtread_cond_t`) y explíquela.

6. Lectores y Escritores II

En el problema anterior la única restricción que tienen los lectores es que no pueden acceder al arreglo si hay un escritor escribiendo. Esto puede traer un

problema si el número de lectores es grande y tratan de leer continuamente el arreglo: si un escritor quiere modificar el arreglo y hay lectores leyendo, puede quedar esperando eternamente si a medida que los lectores terminan de leer son reemplazados por nuevos lectores.

Modifique la solución anterior de manera que cuando un escritor quiera escribir, los lectores que están leyendo terminen su lectura pero no se puedan empezar nuevas lecturas hasta que el escritor escriba. Explique el funcionamiento de la modificación.

7. El Problema del Barbero (Dijkstra)

Una barbería tiene una sala de espera con N sillas y un barbero. Si no hay clientes para atender, el barbero se pone a dormir. Si un cliente llega y todas las sillas están ocupadas, se va. Si el barbero está ocupado pero hay sillas disponibles, se sienta en una y espera a ser atendido. Si el barbero está dormido, despierta al barbero. Escriba un programa que coordine el comportamiento del barbero y los clientes y explíquelo.

8. Servidor de Eco

La siguiente es una implementación secuencial de un servidor de eco (en el puerto 8000) (código fuente aquí):

```
#include <stdio.h>

#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>

#define BUFF_SIZE 1024

int id=-1;

void handle_client(int conn_s)
{
    char buffer[BUFF_SIZE],buffer2[BUFF_SIZE];
    int res;
    id++; // Nuevo cliente
    fprintf(stderr,"New client %d connected\n",id);
    while(1) {
        res=read(conn_s,buffer,BUFF_SIZE);
        if (res<=0) {
            close(conn_s);
            break;
        }
```

```
        buffer[res]='\0';
        sprintf(buffer2,"Response to client %d: %s",id,buffer);
        write(conn_s,buffer2,strlen(buffer2));
    }
}

int main()
{
    int list_s,conn_s=-1,res;
    struct sockaddr_in servaddr;
    char buffer[BUFF_SIZE],buffer2[BUFF_SIZE];
    if ( (list_s = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
        fprintf(stderr, "ECHOSERV: Error creating listening socket.\n");
        return -1;
    }
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(8000);

    if ( bind(list_s, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0 ) {
        fprintf(stderr, "ECHOSERV: Error calling bind()\n");
        return -1;
    }

    if ( listen(list_s, 10) < 0 ) {
        fprintf(stderr, "ECHOSERV: Error calling listen()\n");
        return -1;
    }

    while (1) {
        if ( (conn_s = accept(list_s, NULL, NULL) ) < 0 ) {
            fprintf(stderr, "ECHOSERV: Error calling accept()\n");
            return -1;
        }
        handle_client(conn_s);
    }
    return 0;
}
```

- El servidor actualmente puede atender a un cliente a la vez. Utilizando POSIX Threads conviértalo en un servidor que atienda a múltiples clientes a la vez creando un hilo para cada cliente.
- ¿En su solución hay variables compartidas entre los hilos? Si es así asegúrese de proteger su acceso.

Puede probarlo corriendo el servidor y ejecutando en otra terminal:

```
# telnet 127.0.0.1 8000
```

o bien usando el cliente que se encuentra aquí. Para probar con una cantidad grande de conexiones (mayor a `ulimit -n`) debe cambiar el límite con `ulimit -n X`, con `X` suficientemente grande. Note que para esto posiblemente tenga que ser superusuario.