udp

Tarea N°2

Ramo: Sistemas Operativos.
Sección: 1.
Profesor: Martin Guitierrez.
Alumnos: Felipe Ulloa y Pedro Olivares.

1.Resumen

En esta tarea se realizaron 2 ejercicios relacionados con paralelismo y sincronización. El primer ejercicio constó en realizar una calendarización de actividades en un hipotético caso de semana de pruebas, para así obtener el orden y momento óptimos en los cuales se debe estudiar, respectos a parámetros y su prioridad respectivamente. El segundo ejercicio consistió en abordar un problema común de sincronización “Escritor-Lector”, pero con la particularidad de tener 2 escritores.

2.Introducción

La mayoría de los computadores en la actualidad, posee más de 1 nucleo y 2 hilos, sin embargo, no todos las aplicaciones aprovechan estas características, por lo que es de suma importancia, que aumenta respecto al aumento de estas características, el aprendizaje resolución de problemas, de manera paralela, manteniendo la sincronización para mantener la integridad de los resultados y lograr reducir el tiempo de ejecución de una aplicación, que solo funciona se forma secuencial.

El paralelismo se puede lograr de 2 formas, mediantes múltiples programas, o múltiples hilos, el primer caso tiene escasas medidas de sincronización, por lo que a distintas ejecución obtendrán resultado distintos, por otro lado, los hilos tienen más medidas de sincronización, donde se pueden utilizar estructuras de sincronización, tales como semáforos, mutex_lock y variables de condición.

Semáforos: Es una estructura de sincronización en el cual permite ingresar a su zona crítica solamente “n” hilos seteados en un contador, cuando un hilo sale de la zona crítica, aumenta el contador, para que otro hilo entre, y cuando un hilo entra, lo decrementa. La zona crítica es la porción de código que encierran los métodos inicial y final de una estructura de sincronización.

mutex_lock: Es una estructura en la cual solo se permite el ingreso a la zona crítica a un hilo a la vez, por lo tanto, para que un segundo hilo entre a la zona, requiere esperar a que el hilo en la zona crítica abra el lock.

variables de condición: Es un concepto en el cual permite pausar la ejecución de un hilo, a la espera a la señal de retomar la ejecución, proveniente desde otro hilo, y así sincronizar los hilos.

3.Desarrollo

3.1 Schedulling-Earliest deadline first no expropiativo.

3.1.1 Explicación de la solución.

Se trabajó con un sistema de arrays independientes que manejan tanto tiempo como hora como hilo como día que se ejecuta para cada hilo que es una actividad que se debe organizar.
Entonces el hilo[i] tiene un tiempo total[i] y debe terminar antes del dia[i] a la hora[i] y asi sucesivamente, esto se puede hacer con inputs y for pero para trabajar en un ambiente controlado se setearon los arrays de forma manual, su modificación por input no debería modificar el funcionamiento final.
Además se tiene un contador de horas por día, en total 24 horas por día.

```
int k[10]={0,1,2,3,4,5,6,7,8,9};
int hilo[10]= {1,2,3,4,5,6,7,8,9,10};
int total[10]={6,1,7,8,2,4,8,1,6,9};
int dia[10]=  {0,4,2,3,5,6,1,2,3,4};
int hora[10]={12,2,4,8,16,22,7,9,21,7};

int lunes=24;
int martes=24;
int miercoles=24;
int jueves=24;
int viernes=24;
int sabado=24;
int domingo=24;
```

A continuación lo que se hizo fue generar un sort de días es decir los hilos puesto en orden 1,2,3,etc se sortean en función de los días

```
Original array:
0 4 2 3 5 6 1 2 3 4

Ordenado por dias array:
1 7 3 8 4 9 2 10 5 6
6 8 7 1 8 6 1 9 2 4
0 1 2 2 3 3 4 4 5 6
12 7 4 9 8 21 2 7 16 22
```

,

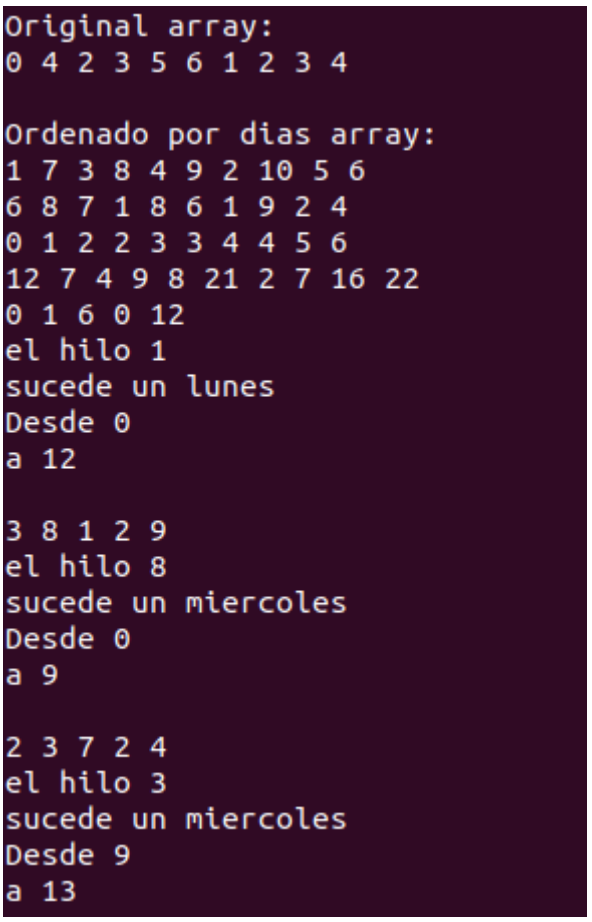
como se puede observar el nuevo orden de los hilos no es 1 2 3 4 5 6 de forma secuencial lo que se tiene es un orden basado en los días que deben ser ejecutados, luego esa información entra en un for para generar los hilos.

```c
selectionSort(dia,hilo,total,hora, n);
printf("\nOrdenado por dias array: \n");
printArray(hilo, n);
printArray(total, n);
printArray(dia, n);
printArray(hora, n);
pthread_t th[8];
int i;
pthread_mutex_init(&mutex, NULL);
for (i = 0; i < 10; i++) {
    int* a = malloc(sizeof(int));
    //int* b = malloc(sizeof(int));
    //int* c = malloc(sizeof(int));
    //int* d = malloc(sizeof(int));
    *a=i;

    //*b=hilo[i];
    //*c=hilo[i];
    //*d=hilo[i];
    if (pthread_create(&th[i], NULL, &schedulling,a) != 0) {
        perror("Failed to create thread");
        return 1;
    }

    //printf("Thread %d has started\n", i);
}
for (i = 0; i < 10; i++) {
    if (pthread_join(th[i], NULL) != 0) {
        return 2;
    }
    //printf("Thread %d has finished execution\n", i);
}
pthread_mutex_destroy(&mutex);
```

Como se puede observar se crea la función schedulling para los hilos, como se sabe al trabajar de manera paralela hay datos que pueden peligrar por eso es primordial identificar la sona critica que resta las horas del día disponibles de cada dia.

```c
void* schedulling(void* arg) {
    pthread_mutex_lock(&mutex);
        int a = *(int*)arg;
        //int b = *(int*)arg2;
        //int c = *(int*)arg3;
        //int d = *(int*)arg4;
        printf("%d ",a);

        printf("%d ",hilo[a]);
        printf("%d ",total[a]);
        printf("%d ",dia[a]);
        printf("%d ",hora[a]);
        int a1=hilo[a];
        int a2=total[a];
        int a3=dia[a];
        int a4=hora[a];
        int b=0;
        int c=0;
        printf("\n");
        //printf(b);
        //printf(c);
        //printf(d);
        if(a3==0){
            if(a4<lunes){
                printf("el hilo %d\n",a1);
                printf("sucede un lunes\n");
                printf("Desde %d\n",24-lunes);
                lunes=lunes-a4;
                printf("a %d\n",24-lunes);
                if(lunes<0){
                    b=lunes*-1;
                    lunes=0;
                    martes=martes-b;
                }
            }
        }
```

La función schelling es simple debido a que no es una función expropiativa la que se seleccionó, entonces, como se mencionó anteriormente es necesario ocupar un mutex lock para asegurar la estabilidad de nuestra información por el trabajo en paralelo de los hilos.
además como en la actividad se pedía probar cuando un horario o tiempo total fuera mayor de 24 se agregó un método de seguridad que hace que una actividad pase al dia siguiente si las horas del dia comienzan a ser negativas asi nos aseguramos de que la actividad termine.
este sistema se hace símil con todos los días de la semana logrando un schedulling final:



como producto final tenemos el horario de ejecución de cada hilo como schedulling final, el único “problema” visible es el hecho de que cada hilo trabaja en paralelo por ende no se imprimen siempre en orden, a pesar de eso se genera un schelling correcto.

3.2 Escritor-Lector.

3.2.1 Explicación de la solución.

Para esta problemática, se tiene 3 “Entidades” que interactúan entre sí, 2 escribas y los alumnos. La solución se abordó desde la perspectiva de dividir el curso de 4 alumnos en 2 grupos, resultando en 4 “Entidades”, cada escrito y grupo tardan 1 segundo en leer y escribir correspondientemente, implicando que el curso completo tarda 2 segundos en poder leer una materia.

G.Alumnos 1	G.Alumnos2	Escriba1	Escriba2

La solución se abordó de forma paralela, generando hilos por cada alumno y escriba. Para poder sincronizar los alumnos se implementó un semáforo(students), una variable de condición para los escribas y alumnos(ordenEstudiar), otra variable de condición entre los escribas(Btermino), y para controlar las iteraciones de un mismo hilo mutex_lock(pergaminoA y pergaminoB, alumno).

```
 9   pthread_mutex_t pergaminoA, pergaminoB, alumno ;
10   pthread_cond_t ordenEstudiar, Btermino;
11   sem_t students;
```

3.2.1.1 Explicacion Escriba2.

La rutina del segundo escriba, consiste en simular la escritura en el PergaminoB. Se divide en 2 etapas, La de escritura y la de saltar el turno cuando se encuentre ocupado el PergaminoB.

```
26  void* Escriba2(void* arg){
27      pthread_mutex_lock(&pergaminoB);
28      turnoB+=1;
29      if(Bused==true && turnoB<=materias){
30          //Saltar Turno
31          if(contadorEscriturasB>0){contadorEscriturasB=0;}
32          sleep(1);
33          if(turnoB==materias){pthread_cond_signal(&Btermino);}
34          pthread_mutex_unlock(&pergaminoB);
35      }else if(Bused==false && turnoB<=materias){
36          //Escribir
37          printf("Escribire B: %d , con  %d escrituras\n", *(int*)arg, contadorEscriturasB+1);
38          sleep(1);
39          contadorEscriturasB+=1;
40          pthread_mutex_unlock(&pergaminoB);
41          if(turnoB==materias){sleep(1);pthread_cond_signal(&Btermino);}
42      }
43      free(arg);
44  }
```

Como se comentó anteriormente, se utiliza un mutex_(pergaminoB), para evitar que otro hilo pueda acceder a la porción de código, hasta el unlock. Dicho de otra manera, la siguiente escritura se realizará una vez la primera ya se haya realizado.

De la imagen () se puede apreciar en la línea 33, que una vez se esté en la última ejecución del segunda escriba, se mandará una señal a la variable de condición “Btermino”, que será recibida por el primer escriba.

3.2.1.2 Explicacion Escriba1.

La rutina del primer escriba consiste en simular la escritura en el pergaminoA, consta de 3 etapas, 2 semejantes a las comentadas en la explicación de la rutina del segundo escriba, pero además se tiene una tercera etapa que consiste en la "Autodestruccion" que consiste en realizar la última operación del segundo escriba, antes de morir. Lo que se realiza es esperar a la señal de término del segundo escriba (Linea 69), y mantenerse mandando la señal a la variable de condición, que hace esperar a los alumnos, para que entren a ejecución a tiempos de escritura, pero sin estar escribiendo.

Como se puede apreciar en la línea 83, además de las 3 etapas mencionadas, por cada ciclo de escriba, el primer escriba se encarga de enviar la señal que manda a ejecución a los alumnos a la espera de esta señal.

```
46  void* Escriba1(void* arg){
47      pthread_mutex_lock(&pergaminoA);
48      turnoA+=1;
49  >   if(Aused==true && turnoA<materias){//No hacer nada, saltar el turno.…
54  >   }else if(Aused==false && turnoA<materias){//Escribir…
65      }else if(turnoA == materias){
66          //AutoDestruccion
67          contadorEscrituraA+=1;
68          if(Aused==false | turnoA==1){printf("Escribire A : %d , con %d ituras\n", *(int*)arg, contadorEscrituraA);}
69          pthread_cond_wait(&Btermino, &pergaminoA);
70          int ordenes=(materias*2)-turnoA;
71          pthread_mutex_unlock(&pergaminoA);
72          printf("Autodestruccion, ordenes = %d\n", ordenes);
73          for (size_t i = 0; i < ordenes; i++)
74          {
75              pthread_cond_signal(&ordenEstudiar);
76              pthread_cond_signal(&ordenEstudiar);
77              pthread_cond_signal(&ordenEstudiar);
78              pthread_cond_signal(&ordenEstudiar);
79              sleep(1);
80          }
81
82      }
83      for (size_t i = 0; i < cantidadAlumnos; i++){pthread_cond_signal(&ordenEstudiar);}
84      free(arg);
85  }
```

3.2.1.3 Explicacion Alumnos.

La rutina de alumnos costo de implementar una variable de condición, para que los alumnos esperen la señal de ejecución provenientes de los escribas, para que primero se escriba, y de lo escrito luego se lea.

Luego de la sincronización entre los alumnos y escribas, entre los mismos alumnos, para acercarnos a un escenario más realista, en el que no todos el curso puede leer un mismo pergamino, se implementó un semáforo con contador de 2, que permitirá que solo 2 alumnos estén a la vez en el semáforo.

Dentro del semáforo se realiza la bifurcación para dirigir a qué pergamino debe leer el estudiante, mediante la bifurcación sobre la consulta de una variable global, que indica si los pergaminos A y B están utilizados.

En la bifurcación se permitirán contadorEscritura*4 alumnos, ya que por cada escritura realizada, todo el curso, que para este caso es de 4 alumnos, lo debe leer, y como se comentó antes, dado que la lectura tarda 1 segundo, y se permiten en el semaforo 2 hilos a la vez, en leer una materia que tarda 1 segundos, el curso tarda 2 segundos.

```
134     sem_init(&students, 0, 2);
```

```
88  void* Alumnos(void* arg){
89      pthread_mutex_lock(&alumno);
90      pthread_cond_wait(&ordenEstudiar, &alumno);
91      sem_wait(&students);
92      if(Aused==true){
93          contadorAlumnos+=1;
94          if(contadorAlumnos==temp){
95              Bused=true;
96              temp=contadorEscriturasB*4;
97              contadorAlumnos=0;
98              Aused=false;
99          }
100         printf("Leyendo A -> %d  contador Alumnos: %d temp: %d\n", *(int*)arg, contadorAlumnos, temp);
101         pthread_mutex_unlock(&alumno);
102         sleep(1);
103     }else if (Bused==true){…
115     sem_post(&students);
116     free(arg);
117 }
```

3.2.2 Resultados.

```
felipe@felipe-notebook:~/workspace/SO/T2$ ./eje
----------Tiempo : 0 ----------
Escribire A : 1 , con 1 escrituras
Escribire B: 1 , con  1 escrituras
----------Tiempo : 1 ----------
Escribire B: 2 , con  2 escrituras
Leyendo A -> 1  contador Alumnos: 1 temp: 4
Leyendo A -> 1  contador Alumnos: 2 temp: 4
----------Tiempo : 2 ----------
Escribire B: 3 , con  3 escrituras
Leyendo A -> 1  contador Alumnos: 3 temp: 4
Leyendo A -> 1  contador Alumnos: 0 temp: 8
----------Tiempo : 3 ----------
Escribire A : 4 , con 1 esciturasLeyendo B -> 2  contador Alumnos: 1 temp: 8
Leyendo B -> 2  contador Alumnos: 2 temp: 8
----------Tiempo : 4 ----------
Autodestruccion, ordenes = 4
Leyendo B -> 2  contador Alumnos: 3 temp: 8
Leyendo B -> 2  contador Alumnos: 4 temp: 8
----------Tiempo : 5 ----------
Leyendo B -> 3  contador Alumnos: 5 temp: 8
Leyendo B -> 3  contador Alumnos: 6 temp: 8
----------Tiempo : 6 ----------
Leyendo B -> 3  contador Alumnos: 7 temp: 8
Leyendo B -> 3  contador Alumnos: 0 temp: 4
----------Tiempo : 7 ----------
Leyendo A -> 4  contador Alumnos: 1 temp: 4
Leyendo A -> 4  contador Alumnos: 2 temp: 4
----------Tiempo : 8 ----------
Leyendo A -> 4  contador Alumnos: 3 temp: 4
Leyendo A -> 4  contador Alumnos: 0 temp: 0
felipe@felipe-notebook:~/workspace/SO/T2$
```

```
felipe@felipe-notebook:~/workspace/SO/T2$ ./eje
----------Tiempo : 0 ----------
Escribire A : 1 , con 1 escrituras
Escribire B: 1 , con  1 escrituras
----------Tiempo : 1 ----------
Leyendo A -> 1  contador Alumnos: 1 temp: 4
Leyendo A -> 1  contador Alumnos: 2 temp: 4
Escribire B: 2 , con  2 escrituras
----------Tiempo : 2 ----------
Leyendo A -> 1  contador Alumnos: 3 temp: 4
Escribire B: 3 , con  3 escrituras
Leyendo A -> 1  contador Alumnos: 0 temp: 8
----------Tiempo : 3 ----------
Escribire A : 4 , con 1 esciturasLeyendo B -> 2  contador Alumnos: 1 temp: 8
Leyendo B -> 2  contador Alumnos: 2 temp: 8
----------Tiempo : 4 ----------
Autodestruccion, ordenes = 4
Leyendo B -> 2  contador Alumnos: 3 temp: 8
Leyendo B -> 2  contador Alumnos: 4 temp: 8
----------Tiempo : 5 ----------
Leyendo B -> 4  contador Alumnos: 5 temp: 8
Leyendo B -> 3  contador Alumnos: 6 temp: 8
----------Tiempo : 6 ----------
Leyendo B -> 3  contador Alumnos: 7 temp: 8
Leyendo B -> 3  contador Alumnos: 0 temp: 4
----------Tiempo : 7 ----------
Leyendo A -> 4  contador Alumnos: 1 temp: 4
Leyendo A -> 4  contador Alumnos: 2 temp: 4
----------Tiempo : 8 ----------
Leyendo A -> 3  contador Alumnos: 3 temp: 4
Leyendo A -> 4  contador Alumnos: 0 temp: 0
felipe@felipe-notebook:~/workspace/SO/T2$
```

3.2.1 Análisis de resultados.

Dado los resultados, se puede apreciar que se cumplen los requerimientos de que siempre al menos un escriba está escribiendo, además se cumple que el tiempo de lectura es el doble que escritura, por lo que dado n escrituras, aparecen 4*n lecturas, en el doble 2 tiempos. Sin embargo si se aprecia la diferencia entre los resultados, en el tiempo 5 y 8 entre las imágenes, en la imagen derecha, ocurre que los hilos tratan de leer de pergaminos los cuales no escribieron la materia correspondiente a leer, y esto podría ocurrir porque al mandar la señal de ejecución a los alumnos en espera, a veces, no los ejecuta en el orden secuencial o podría ser que los hilos de alumnos se tardaron en comenzar a ejecutar y llegaron a este wait en desorden, no obstante se cumple el orden correspondiente de a qué pergamino leer, solo a veces falla cual es la materia que me toca leer.

4.Conclusión

Poder implementar nuestras soluciones de manera paralela, incrementa considerablemente la performance en el tiempo de ejecución, tomando el ejemplo del Escritor-Lector, al poder hacer más de una tarea en un mismo tiempo, sin embargo surge otro problema, el de sincronización, el cual no es un problema nuevo, por lo que lenguajes de programación como C implementan estructuras de sincronización el cual permite mantener una sincronización en la ejecución en paralelo, no obstante se requiere cambiar la visión de programacion con estos nuevos conceptos de sincronización, para saber dónde y cómo implementar estas estructuras, con tal de no tener problemas de sincronización.