Universidad Nacional de General Sarmiento

Sistemas Operativos y Redes

Trabajo Práctico nº1
"Sistemas Operativos"

INFORME

Alumna: María Sol Hoyos

Ejercicio 1: Procesos y Fork

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   fork();

   printf("Hola mundo!\n");
   return 0;
}
```

El output de este programa es:

```
Hola mundo!
Hola mundo!
```

Esto se debe a que la llamada fork() crea un nuevo proceso a partir del proceso padre, donde el proceso hijo creado es una copia del padre. De no estar la llamada fork, el printf se ejecutaría una única vez.

Ejercicio 2: Threads y Semáforos

Exclusión Mutua

Pseudocódigo del problema de los lectores y escritores:

Thread escritor	**Thread lector**

int hoja = 0;
 	
while(true){
hoja += 1;
printf(“Escribe”, hoja)
}
	int hoja = 0;
	
	while(true){
	printf(“Lee”, hoja)
	}

En rojo, se indica la condición de carrera de los threads, donde el Thread lector debe leer después que el Thread escritor escribió. Esto se debe a que en esta situación, los dos procesos leen o escriben un dato compartido y el resultado depende de qué proceso corre primero.

A continuación, el pseudocódigo del código que implementé:

```
//Incluir librerias

int hoja = 0; //variable global
pthread_mutex_t mi_mutex;

void* funcionEscribir (){
        pthread_mutex_lock(mi_mutex);

        hoja+=1;
        printf("Escribe", hoja);

        pthread_mutex_unlock(mi_mutex);
}

void* funcionLeer(){
        pthread_mutex_lock(&mi_mutex);

        printf("Lee", hoja);

        pthread_mutex_unlock(mi_mutex);
}

main(){
        //Declaración de threads
        pthread_mutex_init (mi_mutex, NULL);    //Inicialización del mutex

        pthread_create(thLeer, funcionLeer);
        pthread_create(thEscribir, funcionEscribir);

        //pthread_join de los threads

        pthread_mutex_destroy(mi_mutex); //Destrucción del mutex
}
```

Se indica en rojo la sección crítica, en donde utilizando mutex se logra que cuando un proceso esté en su sección crítica, entonces ningún otro proceso pueda ejecutar su sección crítica.
En cuanto a semáforos de sincronización, no fue necesario agregar ninguno.

Sincronización:

Por cada evento del juego(jugar, ganar, perder, descansar, terminar), existe un proceso que lo realiza y además, imprime por pantalla el nombre del evento.
El objetivo es sincronizar los threads para que se imprima (una cantidad de veces que se fija por parámetro) lo siguiente:

```
jugar (ganar ó perder) descansar
jugar (ganar ó perder) descansar
…
terminar
```

Luego de correr el programa con un parametro=20, este es el output obtenido:

```
JUGAR, PERDER, DESCANSAR.
JUGAR, GANAR, DESCANSAR.
JUGAR, PERDER, DESCANSAR.
JUGAR, GANAR, DESCANSAR.
JUGAR, PERDER, DESCANSAR.
JUGAR, GANAR, DESCANSAR.
JUGAR, PERDER, DESCANSAR.
JUGAR, GANAR, DESCANSAR.
JUGAR, PERDER, DESCANSAR.
JUGAR, GANAR, DESCANSAR.
JUGAR, PERDER, DESCANSAR.
JUGAR, GANAR, DESCANSAR.
JUGAR, PERDER, DESCANSAR.
JUGAR, GANAR, DESCANSAR.
JUGAR, PERDER, DESCANSAR.
JUGAR, GANAR, DESCANSAR.
JUGAR, PERDER, DESCANSAR.
JUGAR, GANAR, DESCANSAR.
JUGAR, PERDER, DESCANSAR.
JUGAR, GANAR, DESCANSAR.
TERMINAR!
```

El evento perder sucedió 10 veces, al igual que el evento ganar. Podria decirse que no hay azar en este resultado. Esto se debe a que los threads que ejecutan cada evento, cuentan con el mismo nivel de prioridad.
No se suceden rachas ganadoras ni rachas perdedoras debido a que tanto el evento perder como el evento ganar, son habilitados por el mismo semáforo.

Ejercicio 3: El Poder del Paralelismo

Se pide implementar un ejemplo de paralelismo de tareas y comparar el tiempo de ejecución de la implementación secuencial y la implementación con paralelismo usando el comando time para medir el tiempo. El resultado fue:

Implementacion secuencial:

```
real    0m0.040s
user    0m0.014s
sys     0m0.001s
```

Implementación con paralelismo:

```
real    0m0.004s
user    0m0.000s
sys     0m0.003s
```