Sistemas Operativos I - LCC
Práctica 1
Introducción a procesos e hilos

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 `-lpthread`.

NOTA: Puede utilizar el SVN de la materia creando un subdirectorio por alumno/grupo en:
`https://svn.dcc.fceia.unr.edu.ar/svn-no-anon/lcc/R-322/Alumnos/2018/`

1. Pipe entre comandos

Modifique en el pequeño shell dado en clase para que acepte dos comandos por vez. Mediante el uso de la función `int pipe(int pip[2])`, haga que la salida del primer comando sea la entrada del segundo.

2. El jardín ornamental

En un jardín ornamental se organizan visitas guiadas y se desea contar cuánta gente entra por día. Hay dos molinetes, uno en cada una de las dos entradas y se ha implementado el sistema para contar los visitantes con pthreads como sigue (código fuente aquí):

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

#define N_VISITANTES 1000000

int visitantes = 0;
```

```
void *molinete(void *arg)
{
  int i;
  for (i=0;i<N_VISITANTES;i++)
    visitantes++;
}

int main()
{
  pthread_t m1, m2;
  pthread_create(&m1, NULL, molinete, NULL);
  pthread_create(&m2, NULL, molinete, NULL);
  pthread_join(m1, NULL);
  pthread_join(m2, NULL);

  printf("Hoy hubo %d visitantes!\n", visitantes);
  return 0;
}
```

- Por cada molinete entran `N_VISITANTES` personas, pero al ejecutar el programa es difícil que el resultado sea `2*N_VISITANTES`. Explique a qué se debe esto.
- Ejecute el programa 5 veces con `N_VISITANTES` igual a 10. ¿El programa dio el resultado correcto siempre? Si esto es así, ¿por qué?
- ¿Cuál es el mínimo valor que podría imprimir el programa? ¿Bajo qué circustancia?
- Implemente una solución utilizando un mutex.

3. Cena de los Filósofos (Dijkstra)

Cinco filósofos se sientan alrededor de una mesa redonda y pasan su vida comiendo y pensando. Cada filósofo tiene un plato de fideos y un tenedor a la izquierda de su plato. Para comer los fideos son necesarios dos tenedores y cada filósofo sólo puede tomar los que están a su izquierda y derecha. Primero toman el que está a su derecha y luego el que está a su izquierda. Si cualquier filósofo toma un tenedor y el otro está ocupado, se quedará esperando, con el tenedor en la mano, hasta que pueda tomar el otro tenedor, para luego empezar a comer.

Una vez que termina de comer deja los tenedores sobre la mesa y piensa por un momento hasta que luego empieza a comer nuevamente.

Una implementación con pthreads es como sigue (código fuente aquí):

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

Figura 1: Filósofos a la mesa

```
#include <pthread.h>

#define N_FILOSOFOS 5
#define ESPERA 5000000

pthread_mutex_t tenedor[N_FILOSOFOS];

void pensar(int i)
{
  printf("Filosofo %d pensando...\n",i);
  usleep(random() % ESPERA);
}

void comer(int i)
{
  printf("Filosofo %d comiendo...\n",i);
  usleep(random() % ESPERA);
}

void tomar_tenedores(int i)
{
  pthread_mutex_lock(&tenedor[i]); /* Toma el tenedor a su derecha */
  pthread_mutex_lock(&tenedor[(i+1)%N_FILOSOFOS]); /* Toma el tenedor a su izquierda */
}

void dejar_tenedores(int i)
{
  pthread_mutex_unlock(&tenedor[i]); /* Deja el tenedor de su derecha */
  pthread_mutex_unlock(&tenedor[(i+1)%N_FILOSOFOS]); /* Deja el tenedor de su izquierda */
}

void *filosofo(void *arg)
{
  int i = (int)arg;
  for (;;)
  {
```

```
    tomar_tenedores(i);
    comer(i);
    dejar_tenedores(i);
    pensar(i);
  }
}

int main()
{
  int i;
  pthread_t filo[N_FILOSOFOS];
  for (i=0;i<N_FILOSOFOS;i++)
    pthread_mutex_init(&tenedor[i], NULL);
  for (i=0;i<N_FILOSOFOS;i++)
    pthread_create(&filo[i], NULL, filosofo, (void *)i);
  pthread_join(filo[0], NULL);
  return 0;
}
```

- Este programa puede terminar en deadlock. ¿En qué situación se puede dar?
- Cansados de no comer los filósofos deciden pensar una solución a su problema. Uno razona que esto no sucedería si alguno de ellos fuese zurdo y tome primero el tenedor de su izquierda.

  Implemente esta solución y explique por qué funciona.
- Otro filósofo piensa que tampoco tendrían el problema si todos fuesen diestros pero sólo comiesen a lo sumo $N - 1$ de ellos a la vez.

  Implemente esta solución y explique por qué funciona. Para ello va a necesitar un semáforo de Dijkstra. Puede utilizar los *POSIX Semaphores*. En la cabecera `semaphore.h` puede encontrar los prototipos de las funciones necesarias:

  ```
  int sem_init(sem_t *sem, int pshared, unsigned int value);
  int sem_destroy(sem_t *sem);
  int sem_wait(sem_t *sem);
  int sem_post(sem_t *sem);
  int sem_getvalue(sem_t *sem, int *valp);
  ```

4. El Problema de los Fumadores (Patil)

Tres procesos tratan de fumar cada vez que pueden. Para hacerlo necesitan tres ingredientes: tabaco, papel y fósforos. Cada uno tiene una cantidad ilimitada de uno de estos ingredientes. Esto es, un fumador tiene tabaco, otro tiene papel y el último tiene fósforos.

Los fumadores no se prestan los ingredientes entre ellos, pero hay un cuarto proceso, el agente, con cantidad ilimitada de todos los ingredientes, que repetidamente pone a disposición de los fumadores dos de los tres ingredientes elegidos al azar. Cada vez que esto pasa, el fumador que tiene el ingrediente restante procede a hacarse un cigarrillo y fumar.

A continuación se puede ver una implementación con pthreads (código fuente aquí):

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

sem_t tabaco, papel, fosforos, otra_vez;

void agente()
{
    for (;;) {
        int caso = random() % 3;
        sem_wait(&otra_vez);
        switch (caso) {
            case 0:
                sem_post(&tabaco);
                sem_post(&papel);
                break;
            case 1:
                sem_post(&fosforos);
                sem_post(&tabaco);
                break;
            case 2:
                sem_post(&papel);
                sem_post(&fosforos);
                break;
        }
    }
}

void fumar(int fumador)
{
    printf("Fumador %d: Puf! Puf! Puf!\n", fumador);
    sleep(1);
}

void *fumador1(void *arg)
{
    for (;;) {
        sem_wait(&tabaco);
        sem_wait(&papel);
        fumar(1);
```

```
        sem_post(&otra_vez);
    }
}

void *fumador2(void *arg)
{
    for (;;) {
        sem_wait(&fosforos);
        sem_wait(&tabaco);
        fumar(2);
        sem_post(&otra_vez);
    }
}

void *fumador3(void *arg)
{
    for (;;) {
        sem_wait(&papel);
        sem_wait(&fosforos);
        fumar(3);
        sem_post(&otra_vez);
    }
}

int main()
{
    pthread_t s1, s2, s3;
    sem_init(&tabaco, 0, 0);
    sem_init(&papel, 0, 0);
    sem_init(&fosforos, 0, 0);
    sem_init(&otra_vez, 0, 1);

    pthread_create(&s1, NULL, fumador1, NULL);
    pthread_create(&s2, NULL, fumador2, NULL);
    pthread_create(&s3, NULL, fumador3, NULL);
    agente();

    return 0;
}
```

- ¿Cómo puede ocurrir un deadlock?
- Implemente una solución y explíquela.