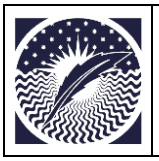

Sistemas Operativos
Grado en Ingeniería Informática en Sistemas de Información

Enseñanzas de Prácticas y Desarrollo
PRÁCTICA 7: Synchronization Problems

Objectives

1. Learn how to synchronize threads
2. Implement some classical synchronization problems

Concepts

In theory class we have seen that there should not be a way to access the value of a semaphore. Well, that's in theory. In practice, library semaphone.h provides a system call that allows us to check the value of a counting semaphore:

```
int sem_getvalue(sem_t *sem, int *sval);
```

The first argument is the semaphore we want to inspect, while the second parameter is an integer passed as a references where the value of the semaphore will be stored. If one or more threads are blocked waiting for this semaphore, the library may returns two values, either a 0 or a negative number, whose absolute value indicates how many threads are currently waiting for the semaphore. The function returns a 0 if there are no errors, otherwise it returns a -1.

Bibliografía

1. Tanenbaum. Sistemas Operativos Modernos, 2ª edición. Pág. 100-131.
2. The Linux Programmer's Guide (véase WebCT)
3. http://www.opengroup.org/onlinepubs/007908799/xsh/semaphore.h.html
4. http://greenteapress.com/semaphores/

Experimentos

E1 (20 min.) Analizar, compilar y ejecutar el siguiente código (solución a el problema 1 de la EPD6). Poner atención en como se logra la exclusión mutua y la sincronización. Use la función sem_getvalue para inspeccionar el valor de los semáforos genéricos.

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


#define QUEUESIZE 4
#define LOOP 20

void *producer (void *args);
void *consumer (void *args);
//semaforo binario
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
//SEMAFOROS GENERICOS, TIENEN QUE SE INICIALIZADOS EN EL MAIN
sem_t notFull;
sem_t notEmpty;


typedef struct {
      int buf[QUEUESIZE];
      long head, tail;
```

```
      int full, empty;
} queue;

queue *queueInit (void);
void queueDelete (queue *q);
void queueAdd (queue *q, int in);
void queueDel (queue *q, int *out);

int main ()
{
      queue *fifo;
      pthread_t pro, con;

      fifo = queueInit ();
      if (fifo ==  NULL) {
            fprintf (stderr, "main: Queue Init failed.\n");
            exit (1);
      }
      //initialize semaphores
      sem_init(&notEmpty, 0, 0);
      sem_init(&notFull, 0, QUEUESIZE);

      pthread_create (&pro, NULL, producer, fifo);
      pthread_create (&con, NULL, consumer, fifo);

      //wait for the producer and consumer to terminate
      pthread_join (pro, NULL);
      pthread_join (con, NULL);
      //delete the queue
      queueDelete (fifo);

      return 0;
}

void *producer (void *q)
{
      queue *fifo;
      int i;

      fifo = (queue *)q;

      for (i = 0; i < LOOP; i++) {
          //wrong, mutex should go just before critical region

            if (fifo->full) {
                  printf ("producer: queue FULL.\n");

            }
            sem_wait(&notFull);
            pthread_mutex_lock(&mut);

            queueAdd (fifo, i);

            pthread_mutex_unlock(&mut);
            sem_post(&notEmpty);
            printf("producer: inserted %d\n",i);
            usleep (100000);
      }

      return (NULL);
}
```

```
void *consumer (void *q)
{
      queue *fifo;
      int i, d;

      fifo = (queue *)q;

      for (i = 0; i < LOOP; i++) {
            if (fifo->empty) {
                  printf ("consumer: queue EMPTY.\n");
            }
            sem_wait(&notEmpty);
            pthread_mutex_lock(&mut);

            queueDel (fifo, &d);

            pthread_mutex_unlock(&mut);
            sem_post(&notFull);
            printf ("consumer: recieved %d.\n", d);
            usleep(200000);
      }

      return (NULL);
}


queue *queueInit (void)
{
      queue *q;
      q = (queue *)malloc (sizeof (queue));
      if (q == NULL) return (NULL);
      q->empty = 1;
      q->full = 0;
      q->head = 0;
      q->tail = 0;
      return (q);
}

void queueDelete (queue *q)
{
      free (q);
}

void queueAdd (queue *q, int in)
{
      q->buf[q->tail] = in;
      q->tail++;
      if (q->tail == QUEUESIZE)
      q->tail = 0;
      if (q->tail == q->head)
      q->full = 1;
      q->empty = 0;
      return;
}

void queueDel (queue *q, int *out)
{
      *out = q->buf[q->head];
      q->head++;
      if (q->head == QUEUESIZE)
```

```
        q->head = 0;
        if (q->head == q->tail)
        q->empty = 1;
        q->full = 0;
        return;
}
```

Problemas

P1. (90 min.) Implementar en C el problema de la cena de los filósofos visto en clase de teoría.

Ampliación de Problemas

AP1 (90 min.) Implementar el problema del barbero dormilón, descrito en Tanembaum Sistemas Operativos Modernos 2ª edición, pág. 129, y visto en clase de teoría.

AP2 (90 min.) Implementar una solución al problema de los lectores y escritores, que modela el acceso a una base de datos. Un búfer compartido, cuatro threads lectores y un thread escritor. Los lectores leen datos del búfer, y el escritor pon datos en el búfer. Solo puede utilizar el búfer compartido un thread y solo uno, es decir, o bien un thread estará escribiendo o bien leyendo, pero nunca ocurrirá simultáneamente (teniendo en cuenta que si no lo esta utilizando nadie, tendrá preferencia el escritor ante el lector). Este problema está descripto en Tanenbaum, Sistemas Operativos Modernos, 2ª edición Pág. 128.