Threads, Semaphores (Φροντιστήριο για την 2η σειρά)

Michalis Diamantaris

Threads

- A thread is a light - weight process
- A thread exists within a process, and uses the process resources
- It is asynchronous
- The program in C calls the pthread.h header file
- How to compile:

  gcc hello.c –pthread –o hello

Creating a thread

```
int pthread_create( pthread_t * thread,
                    pthread_attr_t *attr,
                    void * (*func)(void *),
                    void *arg );
```

Returns 0 for success, (>0) for error

- 1st arg (*thread) – pointer to the identifier of the created thread.
- 2nd arg (*attr) – thread attributes. If NULL, then the thread is created with default attributes
- 3rd arg (*func) – pointer to the function the thread will execute
- 4th arg (*arg) – the argument of the executed function

A thread that prints “Hello World”

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void *hello_world(void * ptr) {
        printf("Hello World! I am a thread!\n");
        pthread_exit(NULL);
}

int main(int argc, char * argv[]) {
        pthread_t thread;
        int rc;
        rc = pthread_create(&thread, NULL, hello_world, NULL);
        if (rc) {
                printf("ERROR: return code from pthread_create() is %d\n",rc);
                exit(-1);
        }
        pthread_join( thread, NULL );
}
```

Thread synchronization mechanisms

- Mutual exclusion (mutex)
  - Guard a critical section
  - **It is a locking mechanism**

- Semaphores
  - A generalized mutex
  - Can send signals between threads
  - **It is a signaling mechanism**

Mutexes

- Guard against multiple threads modifying the same shared data simultaneously

- Provide locking/unlocking critical code sections where shared data is modified

- Each thread waits for the mutex to be unlocked (by the thread who locked it) before performing the code section

Mutexes – basic functions

```
int pthread_mutex_lock(pthread_mutex_t*mutex);
...
int pthread_mutex_unlock(pthread_mutex_t*mutex);

int pthread_mutex_trylock(pthread_mutex_t*mutex);
```

- A mutex is like a key (to access the code section) that is handed to only one thread at a time
- The lock/unlock functions work together

- A mutex is unlocked **only by the thread that has locked it**

Mutex example

```
#include <pthread.h>
pthread_mutex_t my_mutex;
int main() {
        int tmp;
        …
        // initialize the mutex
        tmp= pthread_mutex_init( &my_mutex, NULL );
        …
        // create threads
        …
        pthread_mutex_lock( &my_mutex);
        do_something_private();
        pthread_mutex_unlock( &my_mutex);
        …
        pthread_mutex_destroy(&my_mutex);
        return 0;
}
```

Whenever a thread reaches the lock/unlock block, it first determines if the mutex is locked.

If so, it waits until it is unlocked.

Otherwise, it takes the mutex, locks the succeeding code, then frees the mutex and unlocks the code when it's done.

Semaphores

Counting Semaphores:

- Permit a limited number of threads to execute a section of the code
- Similar to mutexes (if we use binary semaphores it’s the same)
- Should include the semaphore.h header file
- Semaphore functions do not have pthread_prefixes
    - They have sem_prefixes

Semaphores – basic functions

- Creating a semaphore:
  int sem_init (sem_t*sem, int pshared, unsigned int value);
  - Initializes a semaphore object pointed to by sem
  - pshared is a sharing option; a value of *0 means the semaphore is local to the calling process*
  - gives an initial value value to the semaphore

- Terminating a semaphore:
  int sem_destroy (sem_t*sem);
  - Frees the resources allocated to the semaphore sem
  - An error will occur if a semaphore is destroyed for which a thread is waiting

Semaphores – basic functions

- Semaphore control:
  int sem_post(sem_t*sem);
  - Atomically increases the value of a semaphore by 1, i.e., when 2 threads call sem_post simultaneously, the semaphore's value will also be increased by 2 (there are 2 atoms calling)

  int sem_wait(sem_t*sem);
  - Atomically decreases the value of a semaphore by 1; but always waits until the semaphore has a non-zero value first

```
#include <pthread.h>
#include <semaphore.h>
…
void *thread_function( void *arg );
…
sem_t semaphore; // also a global variable just like mutexes
…
int main()  {
int tmp;
        // initialize the semaphore
        /*Semaphore: 0 and 1 --> (locked/unlocked)*/
        tmp = sem_init( &semaphore, 0, 0 );

        // create threads
        pthread_create( &thread[i], NULL, thread_function,
NULL );

        while ( still_has_something_to_do() ) {
                sem_post( &semaphore );
        }
pthread_join( thread[i], NULL );
sem_destroy( &semaphore );
return 0;
}
```

```
void *thread_function( void *arg ) {
        sem_wait( &semaphore );
        perform_task_when_sem_open();
        pthread_exit( NULL );
}
```

- Main thread increments the semaphore's count value in the while loop

- The threads wait until the semaphore's count value is non-zero before performing perform_task_when_sem_open() and further

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>#include <semaphore.h>
sem_t sem;

void *print_Hello( void *ptr ){
        printf("Hello ");
        sem_post(&sem);  //semaphore unlocked (Up)!
}

void *print_World( void *ptr ){
        sem_wait(&sem); //semaphore locked (Down)!
        printf("World\n");
}

int main(int argc, char * argv[] ){
        pthread_t t1, t2;
        int rc, rc2;
        /*Semaphore: 0 and 1 --> (locked/unlocked)*/
        sem_init(&sem, 0, 0 ); /*Initialize semaphore with intraprocess scope*/
        rc   =  pthread_create(&t1, NULL, print_Hello, NULL);
        rc2 =  pthread_create(&t2, NULL, print_World, NULL);

        pthread_join(t1, NULL); /*Wait for the thread to finish*/
        pthread_join(t2, NULL);
}
```

This program sometimes prints "Hello World", sometimes prints "World Hello".

Using a semaphore we can synchronize them.

t2 will never be executed before t1.

Το πρόβλημα του τυλίγματος του γύρου

- 1 **Σεφ** και 3 **μάγειρες**
- Ο Γυρος αποτελείται από **κρέας**, **πατάτες** και **πίτα**
- Κάθε μάγειρας έχει απεριόριστο αριθμό από ένα από τα τρία υλικά

- Ο Σεφ επιλέγει τυχαία δύο από τα υλικά και τα τοποθετεί στο τραπέζι ενημερώνει τον Μάγειρα με το τρίτο υλικό
- Ο μάγειρας με το τρίτο υλικό αφαιρεί τα δύο υλικά από το τραπέζι και χρησιμοποιώντας και το δικό του, παρασκευάζει ένα γύρο και τον καταναλώνει για μερικά δευτερόλεπτα
- Η διαδικασία επαναλαμβάνεται συνέχεια

Requirements

- Πρέπει αναγκαστικά να χρησιμοποιήσετε **τέσσερα threads**
- Χρησιμοποιήστε **semaphores** για τον συγχρονισμό
- Η λύση στο συγκεκριμένο πρόβλημα **δεν πρέπει να καταλήγει σε deadlock**
- Κάθε εκτέλεση του προγράμματος **πρέπει να είναι διαφορετική**
  - Hint: srand() + time()
- Χρησιμοποιήστε την συνάρτηση sleep() για την “δημιουργία και την κατανάλωση του γύρου”
- Compile using -pthread