Threads

Lab 3

Marin Iuliana

Synchronization - semaphores

- Concurrency in multi-tasking applications may introduce race conditions → we need to protect shared resource
- Semaphores are examples of structure aiming at solving such a problem in multi-process applications
- Semaphores are usually system objects managed by the OS kernel
- Semaphores can be thought as counters that we can manipulate by performing two actions: wait and post
- If counter value > 0, wait decrements the counter and allows the task to enter the critical section
- If counter value = 0, wait blocks the tasks in a waiting list
- post increments the counter value□ If the previous value was 0, a task is woken up from the waiting list

Synchronization - Semaphores

- POSIX semaphores
  - sem_open() – opening/creation of a named semaphore. Useful for synchronization among unrelated processes
  - sem_wait() – Decrement the counter and lock if counter = 0. Initial counter value can be set to > 1
  - sem_post() – Increment the count and unlock the critical section if counter > 0
  - sem_close() – Close all the references to the named semaphore
  - sem_unlink() – Destroy semaphore object. If all the references have been closed

  - Link to POSIX real-time and threads extension library to build (gcc … -lrt -pthread)

Synchronization - Semaphores

```
// C program to demonstrate working of Semaphores
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t mutex;

void* thread(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("\nEntered..\n");

    //critical section
    sleep(4);

    //signal
    printf("\nJust Exiting...\n");
    sem_post(&mutex);
}


int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&mutex);
    return 0;
}
```

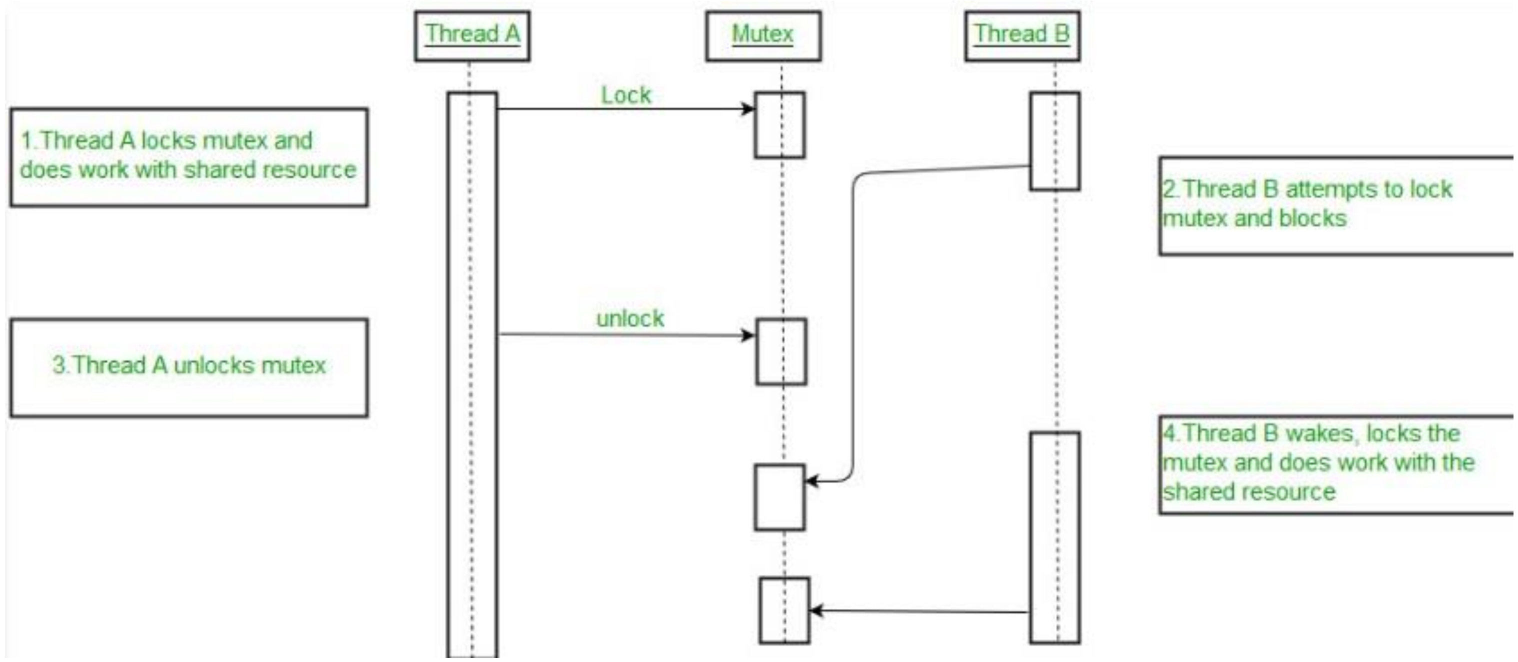
2 threads are being created, one 2 seconds after the first one. But the first thread will sleep for 4 seconds after acquiring the lock. Thus the second thread will not enter immediately after it is called, it will enter 4 – 2 = 2 secs after it is called.

Compilation should be done with
gcc a.c -lpthread –lrt

The output is:
Entered.. Just Exiting... Entered.. Just Exiting...

Synchronization - mutex

- A Mutex is a lock that we set before using a shared resource and release after using it.
- When the lock is set, no other thread can access the locked region of code.
- So we see that even if thread 2 is scheduled while thread 1 was not done accessing the shared resource and the code is locked by thread 1 using mutexes then thread 2 cannot even access that region of code.
- So this ensures a synchronized access of shared resources in the code.


Thread A
Mutex
Thread B
Lock
1.Thread A locks mutex and does work with shared resource
2.Thread B attempts to lock mutex and blocks
unlock
3.Thread A unlocks mutex
4.Thread B wakes, locks the mutex and does work with the shared resource

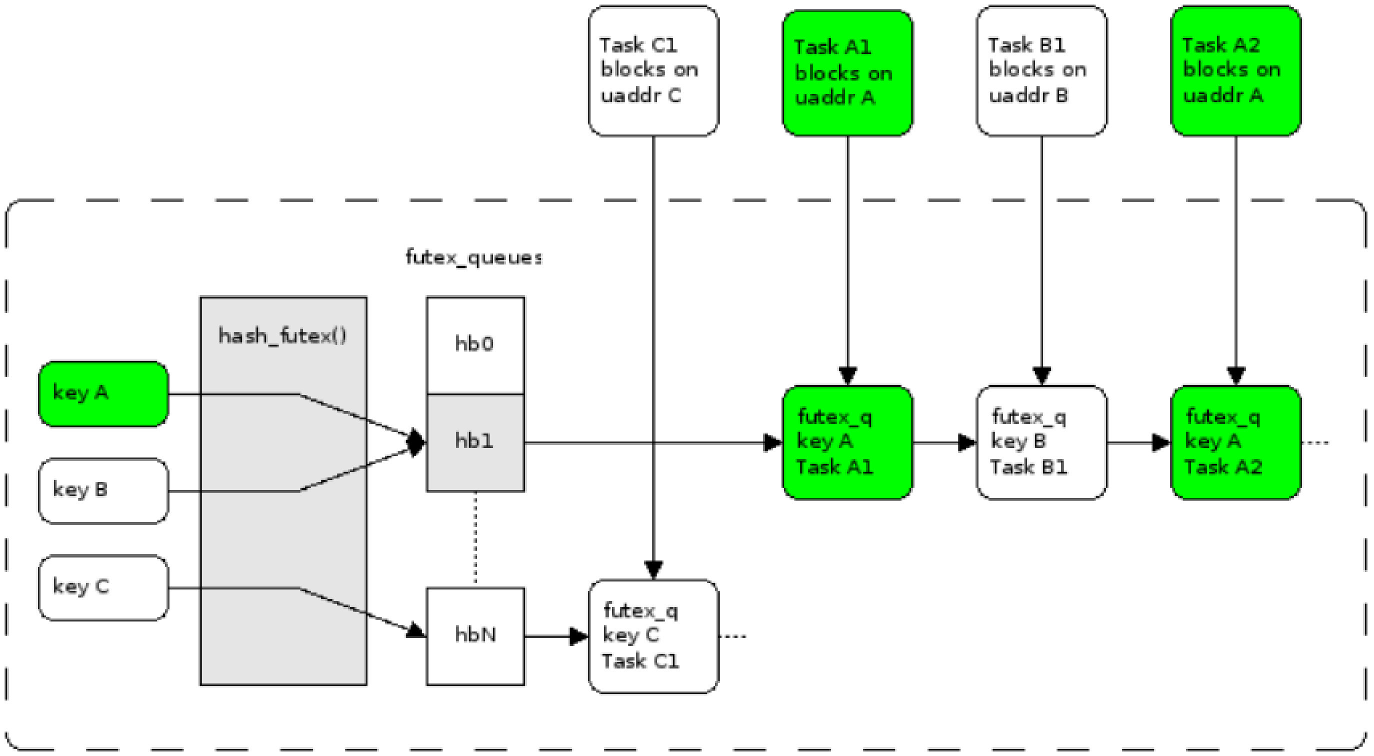
Synchronization - MUTEX

```
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* trythis(void *arg) {
    pthread_mutex_lock(&lock);
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);
    for(i=0; i<(0xFFFFFFFF);i++);
    printf("\n Job %d has finished\n", counter);
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void) {
    int i = 0;
    int error;
    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return 1;
    }
    while(i < 2) {
        err = pthread_create(&(tid[i]), NULL, &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created :[%s]", strerror(error));
        i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

A mutex is initialized in the beginning of the main function. The same mutex is locked in the 'trythis()' function while using the shared resource 'counter'. At the end of the function 'trythis()' the same mutex is unlocked. At the end of the main function when both the threads are done, the mutex is destroyed.
Output :
Job 1 started Job 1 finished Job 2 started Job 2 finished

Synchronization - FUTEX

- A futex is a queue the kernel manages for userspace convenience. It lets userspace code ask the kernel to suspend until a certain condition is satisfied, and lets other userspace code signal that condition and wake up waiting processes.
- In the Linux kernel, futexes are implemented in kernel/futex.c. The kernel keeps a hash table keyed by the address to quickly find the proper queue data structure and adds the calling process to the wait queue.



FUTEX EXAMPLE

```
#include <stdio.h>
#include <pthread.h>
#include <linux/futex.h>
#include <syscall.h>
#define NUM 5
int futex_addr;
int futex_wait(void* addr, int val1){
  return syscall(SYS_futex,&futex_addr,val1, NULL, NULL, 0);
}
int futex_wake(void* addr, int n){
  return syscall(SYS_futex, addr, FUTEX_WAKE, n, NULL, NULL, 0);
}
void* thread_f(void* par){
     int id = (int) par;
    /*go to sleep*/
    futex_addr = 0;
    futex_wait(&futex_addr,0);
    printf("Thread %d starting to work!\n",id);
     return NULL;
}

int main(){
        pthread_t threads[NUM];
        int i;
        for (i=0;i<NUM;i++){
                pthread_create(&threads[i],NULL,thread_f,(void *)i);
        }
        printf("Everyone wait...\n");
        sleep(1);
        printf("Now go!\n");
         /*wake threads*/
         futex_wake(&futex_addr,5);
         /*give the threads time to complete their tasks*/
        sleep(1);
        printf("Main is quitting...\n");
        return 0;
}
```

Exercises

1) Consider a restaurant where orders are received. The waiter is the producer. He is responsible of adding pizza orders inside the restaurant’s kitchen. The cook is responsible of consuming the pizza orders and announcing when he has finished. Simulate the situation using threads.
2) The restaurant has a parking. The barrier to enter it shouldn’t open if there are no available parking lots. Simulate the situation using threads.