Course: Operating Systems

Instructor: Dr. Raheel Ahmed Memon

Lab-11

Objectives:

- Using semaphore for synchronization
- Using mutex for synchronization

Semaphores

The semaphore functions do not start with pthread_, as most thread-specific functions do, but with sem_. Four basic semaphore functions are used in threads. They are all quite simple. A semaphore is created with the sem_init function, which is declared as follows:

#include <semaphore.h> int sem_init(sem_t *sem, int pshared, unsigned int value);

This function initializes a semaphore object pointed to by sem, sets its sharing option, and gives it an initial integer value. The pshared parameter controls the type of semaphore. If the value of pshared is 0, the semaphore is local to the current process. Otherwise, the semaphore may be shared between processes. Here we are interested only in semaphores that are not shared between processes. The next pair of functions controls the value of the semaphore and is declared as follows:

```
#include <semaphore.h>
int sem_wait(sem_t *
sem); int
sem_post(sem_t * sem);
```

These both take a pointer to the semaphore object initialized by a call to sem_init.
The sem_post function atomically increases the value of the semaphore by 1. Atomically here means that if two threads simultaneously try to increase the value of a single semaphore by 1, they do not interfere with each other, as might happen if two programs read, increment, and write a value to a file at the same time. If both programs try to increase the value by 1, the semaphore will always be correctly increased in value by 2. The sem_wait function atomically decreases the value of the semaphore by one, but always waits until the semaphore has a nonzero count first. Thus, if you call sem_wait on a semaphore with a value of 2, the thread will continue executing but the semaphore will be decreased to 1. If sem_wait is called on a semaphore with a value of 0, the function will wait until some other thread has incremented the value so that it is no longer 0. If two threads are both waiting in sem_wait for the same semaphore to become nonzero and it is incremented once by a third process, only one of the two waiting processes will get to decrement the semaphore and continue; the other will remain waiting. This atomic “test and set” ability in a single function is what makes semaphores so valuable.The last semaphore function is sem_destroy. This function tidies up the semaphore when you have finished with it. It is declared as follows:

```
#include <semaphore.h>
int sem_destroy(sem_t *
sem);
```

Again, this function takes a pointer to a semaphore and tidies up any resources that it may have. If you attempt to destroy a semaphore for which some thread is waiting, you will get an error. Like most Linux functions, these functions all return 0 on success.

Debug Following Codes:

Code # 1

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t mutex;
void* thread(void* arg) {
sem_wait(&mutex); //wait state
printf("\nEntered into the Critical Section..\n");
sleep(3); //critical section
printf("\nCompleted...\n"); //comming out from Critical section
sem_post(&mutex);
}
main() {
sem_init(&mutex, 0, 1);
pthread_create(&th1,NULL,thread,NULL);
sleep(2);
pthread_create(&th2,NULL,thread,NULL);
//Join threads with the main thread
pthread_join(th1,NULL);
pthread_join(th2,NULL);
}
```

Code # 2

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
void *thread_function(void *arg);
sem_t bin_sem;
char work_area[WORK_SIZE];
int main() {
int res;
pthread_t a_thread;
void *thread_result;
sem_init(&bin_sem, 0, 0);
pthread_create(&a_thread, NULL, thread_function, NULL);
printf("Input some text. Enter 'end' to finish\n");
while(strncmp("end", work_area, 3) != 0) {
fgets(work_area, WORK_SIZE, stdin);
sem_post(&bin_sem);
}
printf("\nWaiting for thread to finish...\n");
pthread_join(a_thread, &thread_result);
printf("Thread joined\n");
sem_destroy(&bin_sem);
exit(EXIT_SUCCESS);
```

```
}
void *thread_function(void *arg)
{
sem_wait(&bin_sem);
while(strncmp("end", work_area, 3) != 0)
{
printf("You input %d characters\n", strlen(work_area) -1);
sem_wait(&bin_sem);
}
pthread_exit(NULL);
}
```

Mutex

The other way of synchronizing access in multithreaded programs is with mutexes (short for mutual exclusions), which act by allowing the programmer to "lock" an object so that only one thread can access it. To control access to a critical section of code you lock a mutex before entering the code section and then unlock it when you have finished.

The basic functions required to use mutexes are very similar to those needed for semaphores. They are declared as follows: **#include <pthread.h>**

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr); int pthread_mutex_lock(pthread_mutex_t *mutex); int pthread_mutex_unlock(pthread_mutex_t *mutex); int pthread_mutex_destroy(pthread_mutex_t *mutex); Example # 3

Code # 3

```
#include <stdio.h>
#include <stdlib.h>
#include <string.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 error;
if (pthread_mutex_init(&lock, NULL) != 0)
{
```

```
printf("\n mutex init has failed\n");
return 1;
}
while (i < 2) {
pthread_create(&(tid[i]),NULL, &trythis, NULL);
 i++;
}
pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);
pthread_mutex_destroy(&lock);
return 0;
}
```

Code # 4

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
void *thread_function(void *arg);
pthread_mutex_t work_mutex;
#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int time_to_exit = 0;
int main() {
pthread_t a_thread;
void *thread_result;
pthread_mutex_init(&work_mutex, NULL);
pthread_create(&a_thread, NULL, thread_function, NULL);
pthread_mutex_lock(&work_mutex);
printf("Input some text. Enter 'end' to finish\n");
while(!time_to_exit) {
fgets(work_area, WORK_SIZE, stdin);
pthread_mutex_unlock(&work_mutex);
while(1) {
pthread_mutex_lock(&work_mutex);
if (work_area[0] != '\0') {
pthread_mutex_unlock(&work_mutex);
sleep(1);
}
else {
break;
}
}
pthread_mutex_unlock(&work_mutex);
printf("\nWaiting for thread to finish...\n");
pthread_join(a_thread, &thread_result);
printf("Thread joined\n");
pthread_mutex_destroy(&work_mutex);
exit(EXIT_SUCCESS);
```

```
}
void *thread_function(void *arg) {
sleep(1);
pthread_mutex_lock(&work_mutex);
while(strncmp("end", work_area, 3) != 0) {
printf("You input %d characters\n", strlen(work_area) -1);
work_area[0] = '\0';
pthread_mutex_unlock(&work_mutex);
sleep(1);
pthread_mutex_lock(&work_mutex);
while (work_area[0] == '\0' ) {
pthread_mutex_unlock(&work_mutex);
sleep(1);
pthread_mutex_lock(&work_mutex);
}
}
time_to_exit = 1;
work_area[0] = '\0';
pthread_mutex_unlock(&work_mutex);
pthread_exit(0);
}
```