Assignment 10: Write C program using thread concept to count no of character entered from the keyboard and also utilize the concept of semaphore for thread synchronization

Study material for assignment 10:

Concurrent access to shared data may result in data inconsistency. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes. A **race condition** is a situation where two or more processes access shared data concurrently and final value of shared data depends on timing. The section of code that is under the race condition is called the **critical section** of any process.

Fortunately, there is a set functions specifically designed to provide better ways to control the execution of threads and access to critical sections of code. *semaphores*, which act as gatekeepers around a piece of code, and *mutexes*, which act as a mutual exclusion (hence the name mutex) device to protect sections of code. For example, controlling access to some shared memory, which only one thread can access at a time, would most naturally involve a mutex. However, controlling access to a set of identical objects as a whole, such as giving one telephone line out of a set of five available lines to a thread, suits a counting semaphore better.

We will use POSIX Realtime Extensions and used for threads. we look at the simplest type of semaphore, a *binary* semaphore that takes only values 0 or 1.There is also a more general semaphore, a *counting* semaphore that takes a wider range of values. Normally, semaphores are used to protect a piece of code so that only one thread of execution can run it at any onetime. For this job a binary semaphore is needed.

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 (which we discuss more in a moment), 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.

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.

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.

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);

Solution to the assignment:

```
#include<stdio.h>
#include<sys/types.h>
#include<string.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<semaphore.h>
void *count(void *arg);
sem_t s;
char area [1024];
int main()
{
int res;
pthread_t tid1;
void *result;
res=sem_init(&s,0,0);
if(res!=0)
{
perror("Semaphore creation failed");
exit(EXIT_FAILURE);
}
res=pthread_create(&tid1,NULL,count,NULL);
if(res!=0)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}
printf("Enter some text. Enter end to finish\n");
while(strncmp("end",area,3)!=0)
{
fgets(area,1024,stdin);
sem_post(&s);
}
printf("\nWaiting for thread to finisgh\n");
res=pthread_join(tid1,&result);
if(res!=0)
{
perror("Thread joined failure");
exit(EXIT_FAILURE);
}
printf("Thread joined%s\n",(char *)result);
sem_destroy(&s);
exit(EXIT_SUCCESS);
}
void *count(void *arg)
{
sem_wait(&s);
while(strncmp("end",area,3)!=0)
        {
        printf("You input %d Character\n",strlen(area)-1);
        sem_wait(&s);
        }
```

```
pthread_exit("Thnaks for CPU time & Count funtion running with
semamphore\n");
}
```

The first important change is the inclusion of semaphore.h to provide access to the semaphore functions.
Then you declare a semaphore and some variables and initialize the semaphore *before* you create new thread.

```
sem_t s;
char area[1024];
int main() {
int res;
pthread_t tid1;
void *result;
res = sem_init(&s, 0, 0);
if (res != 0) {
perror("Semaphore initialization failed");
exit(EXIT_FAILURE);
}
```

Note that the initial value of the semaphore is set to 0. When you initialize the semaphore, you set its value to 0. Thus, when the thread's function starts, the call to sem_wait blocks and waits for the semaphore to become nonzero.

In the function main, after you have started the new thread, you read some text from the keyboard, load your area, and then increment the semaphore with sem_post.

```
printf("Input some text. Enter 'end' to finish\n");
while(strncmp("end", area, 3) != 0) {
fgets(area, 1024, stdin);
sem_post(&s);
}
```

In the new thread, you wait for the semaphore and then count the characters from the input. In the main thread, you wait until you have some text and then increment the semaphore with sem_post, which immediately allows the other thread to return from its sem_wait and start executing. Once it has counted the characters, it again calls sem_wait and is blocked until the main thread again calls sem_post to increment the semaphore.

```
sem_wait(&s);
while(strncmp("end", area, 3) != 0) {
printf("You input %d characters\n", strlen(area) -1);
sem_wait(&s);
}
```

While the semaphore is set, you are waiting for keyboard input. When you have some input, you release the semaphore, allowing the second thread to count the characters before the first thread reads the keyboard again.
Again both threads share the same area array. Again, we have omitted some error checking, such as the returns from sem_wait to make the code samples more succinct and

easier to follow. However, in production code you should always check for error returns unless there is a very good reason to omit this check.

Assignment 11: Write a C program to create several threads in the same program and then collect them again in an reverse order from that in which they were started.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<sys/types.h>
#include<math.h>
void *multithread(void *arg);
int main()
{
int i,res,n;
void *result;
printf("\n enter the no of thread you want create");
scanf("%d",&n);
pthread_t tid[n];
for(i=0;i<n;i++)
{
res=pthread_create(&tid[i],NULL,multithread,(void *)i);
if(res!=0)
{
perror("Thread creation failed");
exit("EXIT_FAILURE");
}
sleep(1);
}
printf("\nWaiting for thread to finish.......");
for(i=n-1;i>=0;i--)
{
res=pthread_join(tid[i],&result);
if(res==0)
{
printf("Picked up a thread %d\n",(int *)result);
}
else
{
perror("Thread join failed");
}
}
printf("All done\n");
exit(EXIT_SUCCESS);
}
void *multithread(void *arg)
{
int *j=(int )arg;
//int l;
printf("\nThread funtion is running argument was %d\n",j);
//l=1+(int)(9.0*rand()/(RAND_MAX+1.0));
//sleep(l);
printf("Bye from %d\n",j);
pthread_exit(j);
```

}