Week 9– Assignments

1. Write a C program that implements alarm clock.

```
// C implementation of digital clock

#include <stdio.h>

#include <time.h>


// driver code

int main()

{

        time_t s, val = 1;

        struct tm* current_time;


        // time in seconds

        s = time(NULL);


        // to get current time

        current_time = localtime(&s);


        // print time in minutes,

        // hours and seconds

        printf("%02d:%02d:%02d",

                current_time->tm_hour,

                current_time->tm_min,

                current_time->tm_sec);
        return 0;

}
```

2. Write a C program that displays current time.

```
/* localtime example */
#include <stdio.h>
#include <time.h>

int main ()
{
  time_t rawtime;
  struct tm * timeinfo;

  time ( &rawtime );
  timeinfo = localtime ( &rawtime );
  printf ( "Current local time and date: %s", asctime (timeinfo) );

  return 0;
}
```

3. Write a C program for Test-and-Set Algorithm (A/P)

. Test and Set:
Here, the shared variable is lock which is initialized to false. TestAndSet(lock) algorithm works in this way – it always returns whatever value is sent to it and sets lock to true. The first process will enter the critical section at once as TestAndSet(lock) will return false and it'll break out of the while loop. The other processes cannot enter now as lock is set to true and so the while loop continues to be true. Mutual exclusion is ensured. Once the first process gets out of the critical section, lock is changed to false. So, now the other processes can enter one by one. Progress is also ensured. However, after the first process, any process can go in. There is no queue maintained, so any new process that finds the lock to be false again can enter. So bounded waiting is not ensured.

4. Write a C program for Peterson’s Algorithm (A/P)

```
// Filename: peterson_spinlock.c

// Use below command to compile:

// gcc -pthread peterson_spinlock.c -o peterson_spinlock


#include <stdio.h>

#include <pthread.h>

#include"mythreads.h"


int flag[2];

int turn;

const int MAX = 1e9;

int ans = 0;


void lock_init()

{

	// Initialize lock by resetting the desire of

	// both the threads to acquire the locks.

	// And, giving turn to one of them.

	flag[0] = flag[1] = 0;

	turn = 0;

}


// Executed before entering critical section

void lock(int self)

{
        // Set flag[self] = 1 saying you want to acquire lock

        flag[self] = 1;


        // But, first give the other thread the chance to

        // acquire lock

        turn = 1-self;


        // Wait until the other thread looses the desire

        // to acquire lock or it is your turn to get the lock.

        while (flag[1-self]==1 && turn==1-self) ;

}


// Executed after leaving critical section

void unlock(int self)

{

        // You do not desire to acquire lock in future.

        // This will allow the other thread to acquire

        // the lock.

        flag[self] = 0;

}


// A Sample function run by two threads created

// in main()

void* func(void *s)

{

        int i = 0;
        int self = (int *)s;
        printf("Thread Entered: %d\n", self);

        lock(self);

        // Critical section (Only one thread
        // can enter here at a time)
        for (i=0; i<MAX; i++)
                ans++;

        unlock(self);
}

// Driver code
int main()
{
        // Initialized the lock then fork 2 threads
        pthread_t p1, p2;
        lock_init();

        // Create two threads (both run func)
        pthread_create(&p1, NULL, func, (void*)0);
        pthread_create(&p2, NULL, func, (void*)1);

        // Wait for the threads to end.
        pthread_join(p1, NULL);
        pthread_join(p2, NULL);

        printf("Actual Count: %d | Expected Count: %d\n",
                                                          ans, MAX*2);

        return 0;
}
```

5. Write a C Program for Swap Algorithm (A/P)

```
#include<stdio.h>
 int main()
{
int a=10, b=20;
printf("Before swap a=%d b=%d",a,b);
a=a+b;//a=30 (10+20)
b=a-b;//b=10 (30-20)
a=a-b;//a=20 (30-10)
printf("\nAfter swap a=%d b=%d",a,b);
return 0;
}
```

6. Write a C Program for Bakery Algorithm (A/P)

```
// Importing the thread library
#include "pthread.h"

#include "stdio.h"
```

```
// Importing POSIX Operating System API library
#include "unistd.h"

#include "string.h"

// This is a memory barrier instruction.
// Causes compiler to enforce an ordering
// constraint on memory operations.
// This means that operations issued prior
// to the barrier will be performed
// before operations issued after the barrier.
#define MEMBAR __sync_synchronize()
#define THREAD_COUNT 8

volatile int tickets[THREAD_COUNT];
volatile int choosing[THREAD_COUNT];

// VOLATILE used to prevent the compiler
// from applying any optimizations.
volatile int resource;

void lock(int thread)
{

        // Before getting the ticket number
        //"choosing" variable is set to be true
```

```
choosing[thread] = 1;


MEMBAR;

// Memory barrier applied


int max_ticket = 0;


// Finding Maximum ticket value among current threads

for (int i = 0; i < THREAD_COUNT; ++i) {


        int ticket = tickets[i];

        max_ticket = ticket > max_ticket ? ticket : max_ticket;

}


// Allotting a new ticket value as MAXIMUM + 1

tickets[thread] = max_ticket + 1;


MEMBAR;

choosing[thread] = 0;

MEMBAR;


// The ENTRY Section starts from here

for (int other = 0; other < THREAD_COUNT; ++other) {


        // Applying the bakery algorithm conditions

        while (choosing[other]) {
                    }

                    MEMBAR;

                    while (tickets[other] != 0 && (tickets[other]
                                                                        < tickets[thread]
                                                              || (tickets[other]
                                                                                  ==
tickets[thread]
                                                                        && other <
thread))) {
                    }
          }
}

// EXIT Section
void unlock(int thread)
{

          MEMBAR;
          tickets[thread] = 0;
}

// The CRITICAL Section
void use_resource(int thread)
{
          if (resource != 0) {
                    printf("Resource was acquired by %d, but is still in-use by %d!\n",
                            thread, resource);
          }

          resource = thread;
          printf("%d using resource...\n", thread);

          MEMBAR;
          sleep(2);
          resource = 0;
}

// A simplified function to show the implementation
void* thread_body(void* arg)
{

          long thread = (long)arg;
          lock(thread);
          use_resource(thread);
          unlock(thread);
          return NULL;
}

int main(int argc, char** argv)
{
          memset((void*)tickets, 0, sizeof(tickets));

          memset((void*)choosing, 0, sizeof(choosing));

          resource = 0;


          // Declaring the thread variables

          pthread_t threads[THREAD_COUNT];


          for (int i = 0; i < THREAD_COUNT; ++i) {


                    // Creating a new thread with the function

                    //"thread_body" as its thread routine

                    pthread_create(&threads[i], NULL, &thread_body, (void*)((long)i));

          }


          for (int i = 0; i < THREAD_COUNT; ++i) {


                    // Reaping the resources used by

                    // all threads once their task is completed !

                    pthread_join(threads[i], NULL);

          }


          return 0;

}
```

7. Write a C Program for Monitors in Process Synchronization based on wait and signal operations (A/P)

```
struct semaphore {
    enum value(0, 1);

    // q contains all Process Control Blocks (PCBs)
    // corresponding to processes got blocked
    // while performing down operation.
    Queue<process> q;

} P(semaphore s)
{
    if (s.value == 1) {
        s.value = 0;
    }
    else {
        // add the process to the waiting queue
        q.push(P) sleep();
    }
}
V(Semaphore s)
{
    if (s.q is empty) {
        s.value = 1;
    }
    else {
```

```
                    // select a process from waiting queue

                    Process p = q.front();

                    // remove the process from wating as it has been

                    // sent for CS

                    q.pop();

                    wakeup(p);

            }

}
```

8. Write a C Program for Monitor’s implementation using Semaphores (A/P)

```
#include<semaphore.h>

int availableItemCounts;
int bufferSize;
int itemShouldBeProducedCount;
int itemShouldBeConsumedCount;
int availableItemIndex;//index of last item inserted in buffer.
char *buffer;
sem_t mutex;

typedef struct
{
   sem_t semaphore;
   int numberOfBlockedThreads;
} Condition;

Condition bufferIsFull, bufferIsEmpty;


int countCV(Condition conditionVariable)
{
   return conditionVariable.numberOfBlockedThreads;
}

void waitCV(Condition conditionVariable)
{
   conditionVariable.numberOfBlockedThreads++;
   sem_wait(&(conditionVariable.semaphore));
}

void signalCV(Condition conditionVariable)
{
   if(countCV(conditionVariable)>0)
```

```
    {
        sem_post(&(conditionVariable.semaphore));
        conditionVariable.numberOfBlockedThreads--;
    }
    else
    {
        sem_post(&mutex);
    }
}


void monitorInit(int buffSize, int itemSBPC,int itemSBCC)
{
    availableItemCounts=0;
    availableItemIndex=-1;
    bufferSize=buffSize;
    itemShouldBeProducedCount=itemSBPC;
    itemShouldBeConsumedCount=itemSBCC;
    char tempBuffer[bufferSize];
    buffer=tempBuffer;
    sem_init(&(bufferIsFull.semaphore), 0, 0);
    sem_init(&(bufferIsEmpty.semaphore), 0, 0);
    sem_init(&(mutex), 0, 1);
    bufferIsEmpty.numberOfBlockedThreads=0;
    bufferIsFull.numberOfBlockedThreads=0;
}

void monitor_Insert(char item)
{
    sem_wait(&mutex);
    if (itemShouldBeProducedCount>0)
    {
        if(availableItemCounts==bufferSize)
        {
            sem_post(&mutex);
            waitCV(bufferIsFull);
            sem_wait(&mutex);
        }
        availableItemIndex++;
        buffer[availableItemIndex]=item;
        printf("p:%lu, item: %c, at %d\n", pthread_self(), item, availableItemIndex);
        itemShouldBeProducedCount--;
        availableItemCounts++;
        signalCV(bufferIsEmpty);
    }
    else
    {
        sem_post(&mutex);
        pthread_exit(NULL);
    }

}

void monitor_Remove(char item)
{
    sem_wait(&mutex);
    if (itemShouldBeConsumedCount>0)
    {
        if(availableItemCounts==0)
        {
           sem_post(&mutex);
           waitCV(bufferIsEmpty);
           sem_wait(&mutex);
        }
        item=buffer[availableItemIndex];
        printf("c:%lu, item: %c, at %d\n", pthread_self(), item, availableItemIndex);
        availableItemIndex--;
        itemShouldBeConsumedCount--;
        availableItemCounts--;
        signalCV(bufferIsFull);

     }
     else
     {
        sem_post(&mutex);
        pthread_exit(NULL);
     }
  }
```