Course Name: Intro to Computer Systems

Course Number and Section: **14:332:434**

Assignment: [Homework 3]

Instructor: Striki

Date Performed: 4/29/19

Date Submitted:

Submitted by:

[Omar Atieh 170001765]

Problem 1:

1. The C code below is an attempt to the 'Dining Philosophers Problem' via an implementation of a monitor using semaphores; the first being 'mutex', keeping track if a process is occupying the monitor at any time, and a list of monitors, 'self[]', keeping track of each semaphore variable of each of the five philosopher threads.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t mutex, self[5];
typedef enum state_enum {THINKING,HUNGRY,EATING} state;
state phil_states[5] = {THINKING,THINKING,THINKING,THINKING,THINKING};
//state phil_states[5] = {HUNGRY,HUNGRY,HUNGRY,HUNGRY,HUNGRY};

void print_states(){
	int t;
	for(t=0;t<5;t++){
		if(phil_states[t]==THINKING)
			printf("THINKING");
		else if(phil_states[t]==HUNGRY)
			printf("HUNGRY");
		else
			printf("EATING");
		if(t<4)
			printf(",");
	}
	printf("'\n");
}

void test(int i){
	if(phil_states[(i+4)%5]!=EATING && phil_states[i]==HUNGRY && phil_states[(i+1)%5]!=EATING){
		phil_states[i] = EATING;
		//printf("Changed thread%d to EATING\n",i);//DELETE
		print_states();
		sem_post(&self[i]);
	}
}

void* dining_philosophers(void* arg){
	int t = *((int*)arg);
	int i;
	for(i=0;i<10;i++){
		if(phil_states[t]==THINKING && rand()%5==0){
			phil_states[t] = HUNGRY;
			//printf("Changed thread%d to HUNGRY\n",t);//DELETE
			print_states();
		}

		sem_wait(&mutex);//Beginning of monitor

		if(phil_states[t]==HUNGRY){
			//pickup
			test(t);
			if(phil_states[t]!=EATING){
```

```
                                        //printf("Thread%d is waiting!\n",t);//DELETE
                                        //sem_post(&mutex);
                                        //printf("Thread%d about to wait\n",t);//DELETE
                                        sem_wait(&self[t]);
                                        //printf("Thread%d finished waiting\n",t);//DELETE
                                        //sem_wait(&mutex);
                                }
                                //print_states();
                                //putdown
                                phil_states[t] = THINKING;
                                //printf("Changed thread%d to THINKING\n",t);//DELETE
                                print_states();
                                test((t+4)%5);
                                test((t+1)%5);
                        }

                        sem_post(&mutex);//End of monitor
                        //sleep(1);
                }
}

int main(){
        srand(time(0));
        sem_init(&mutex,0,1);
        int t;
        for(t=0;t<5;t++){
                sem_init(&self[t],0,0);
        }
        pthread_t phil_threads[5];
        print_states();
        for(t=0;t<5;t++){
                pthread_create(&phil_threads[t],NULL,dining_philosophers,&t);
                sleep(1);
        }
        for(t=0;t<5;t++){
                pthread_join(phil_threads[t],NULL);
        }
        sem_destroy(&mutex);
}
```

2. The same code can be used from part one; all that needs to be adjusted is replacing the semaphores with POSIX locks. The semaphores in part one were pretty much being used as locks in the first place.

3. Between the two parts, the code in part one seems to be faster than the code in part 2. This can probably be attributed the mutex semaphore allowing all three philosopher threads into its critical section, making the procedure faster than using locks.

Problem 2: (8 pts)

1. (4 pts) Show (pseudo-code) how to implement the wait() and signal() semaphore operations in multiprocessor environments using test and set() instruction. The solution should exhibit minimal busy waiting.

Implementation of wait() and signal() semaphore operations in multiprocessor environments can be done with test() and set() instructions. The following is one implementation;

The wakeup() removes one of processes in the waiting queue and places it in the ready queue.
The way I implemented this was taking the wait(semaphore *S) and signal(semaphore *s) and adding the test and set within the functions. This will exhibit minimal busy waiting because as soon as a the process is added to waiting queue the next process will execute.
PseudoCode

```
int semValue = 0;
int target = 0;

wait(){
      //while test and set returns true
      while (TestAndSet(&target) == 1);
      if (semValue == 0) {
                  //add process to the process waiting queue
                  target = 0;
                  }else {
                              //semValue gets decremented one for wait

                              semValue= semValue -1;

                              target = 0;
            }

}
signal(){
while (TestAndSet(&target) == 1);
//if the semvalue is 0 and if the wait queue has a process on it
if (semValue == 0){
      if(wait queue !=0){

            //wake up() first process in the waiting queue;
      }
      //Must increment the semaphore value
```

```
      else semValue=semValue+1;
      target = 0;
      }
```

2. (4 pts) Describe (pseudo-code) how the compare and swap() instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.

We use compare and swap when ever a process exits the critical section to set the mutex back to 0 to allow another to enter the critical section. This will satisfy mutual exclusion.
The process in the code below can only enter critical section if either one of the wait[i] or turn are false. Progress is met because each time there is a process that wished to enter the critical section and the critical section is empty then they are allowed. I tried to incorporate bounded waiting as well with wait[i] because each process will have a fair share because the program will iterate through the array. I used slide 5.22 to assist me in this problem.

```
boolean wait[n];
boolean lock;
do {
/*similar boolean wait[i] will equal true if it is ready to enter the
critical section*/
wait[i] = true;
//turn indicates the key
   turn = true;
//when both wait[i] and turn are true
while (wait[i] && turn)
//we use the compare and swap to enter the critical section
   turn = compare_and_swap(&lock,0,1);
//we have to set it back to false
   wait[i] = false; /* critical section */
   j = (i + 1) % n;
while ((j != i) && !wait[j])
     j = (j + 1) % n;
   if (j == i)
     lock = false;
Else
     waiting[j] = false;
/* remainder section */
} while (true);
```

Problem 3: (5 points) A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint:

The sum of all unique numbers associated with all the processes currently accessing the file must be less than n. Write a monitor (pseudo-code) to coordinate access to the file.
Solution:

```
monitor file {
//making a monitor called file
//we need to make a current number to coordinate access to the file
int currentInt = 0;
int n;

//condition variable is used to allow one to use wait() and signal()
//separately.

condition variable;

void access(int number) {
     while (curentInt + number >= n)

//the wait() is run in variable

     variable.wait();
     currentInt += number; }

 //This is when you are done accessing the file

void leave(int number) {

//set currentInt back

     currentInt -= number;
     }
}
```

This could have been done using semaphores however using monitor make it a little bit less complicated with monitor it is guaranteed that only one task can be active inside the monitor.
Signaling Functions: wait(): Suspends tasks until signal() is called.
signal(): activates exactly one suspended process

4. In a corporate environment managers work separately than office suppliers. Managers work in their office for a window of time (work_actual()) and take breaks periodically outside (time_off()). The suppliers' personnel never enter the office while there are still managers there and vice versa. The

managers are represented by threads that execute function manager() and the office supplier personnel are represented by threads which execute function supplier().

Part 1:

```
mutex = semaphore(1);
empty = semaphore(1);
int managerCount = 0;

void supplier()
{
	while(;){
		wait(empty);
		supply();
		signal(empty);
		time_off();


	}
}
void manager(){
	while(;){
		wait(mutex);
		managerCount= mangerCount + 1;
		if(mangerCount==1)

			wait(empty);
		signal(mutex);
		work_actual();
		wait(mutex);
		managerCount--;


		if(managerCount == 0)
			signal(empty);
		signal(mutex);
		time_off();

	}
}
```

The problem with the solution above is starvation. For example if you have more managers continuously arriving then the the suppliers dont have a chance to enter the office.

Part 2:

One way of limiting the amount of managers is to add an extra semaphore of max space. So it will unlock only if there is extra space available.

```
mutex = semaphore(1);
empty = semaphore(1);
int managerCount = 0;
Max_Space = semaphore(N)


void supplier()
{
      while(;){
            wait(empty);
            supply();
            signal(empty);
            time_off();


      }
}
void manager(){
      while(;){
            wait(mutex);
            managerCount= mangerCount + 1;
            if(mangerCount==1)

            wait(empty);
              signal(mutex);
              wait(Max_Space);
              work_actual();
              signal(Max_Space);
              wait(mutex);
              managerCount--;


            if(managerCount == 0)
              signal(empty);
              signal(mutex);
              time_off();

      }
}
```

In the code I wait to see if there is still space and If there is then I can do work_actual(). After I signal it back to increment and continue.

Part 3:
In this code I added another semaphore to limit the amount of space for suppliers. I also shifted the semaphore from problem 2 to account for the space of managers only and not the total space. The supplier code had to be changed to work similarly to manager().

```
mutex = semaphore(1);
mutex1 = semaphore(1);
empty = semaphore(1);
int managerCount = 0;
int supplierCount = 0;
managerSpace = semaphore(N);
supplierSpace= semaphore(M);



void supplier()
{
      while(;){
            //pretty much the same methodology from the manager() function
            wait(mutex1);
            supplerCount++;
            if(supplierCount == 1)

            wait(empty);
            signal(mutex1);
            wait(supplierSpace);
            supply();
            signal(suplierSpace);
            wait(mutex1);
            supplerCount--;

            if(supplierCount == 0)
              signal(empty);
              signal(mutex1);
              time_off();

      }
}
void manager(){
```

```
      while(;){
            wait(mutex);
            managerCount= mangerCount + 1;
            if(mangerCount==1)

            wait(empty);
              signal(mutex);
              wait(managerSpace);
              work_actual();
              signal(managerSpace);
              wait(mutex);

              managerCount--;


            if(managerCount == 0)
              signal(empty);
              signal(mutex);
              time_off();

      }
}
```

Problem 5:
Compilable and Runnable C Code. I have fully implemented the algorithm.
Explanation of Algorithm below code

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>

#define N 5

sem_t sem1;
pthread_mutex_t lock1,lock2,lock3;
int count1 = 0;
short b = 0;


void boarding_bus(){
```

```
    /*resets algorithm to accept N travelers into waiting room*/
    pthread_mutex_lock(&lock2);
       count1++;
    pthread_mutex_unlock(&lock2);

    while(count1<(N*2) && b==0);

    pthread_mutex_lock(&lock3);
       count1--;
       b=1;
    pthread_mutex_unlock(&lock3);

    if(count1<(N+1)){
       count1=0;
       b=0;
       sem_init(&sem1, 0, N);
    }


    printf("boarding bus\n");
}

void cross_check(){

    printf("cross check waiting\n");

    pthread_mutex_lock(&lock1);
       count1++;
    pthread_mutex_unlock(&lock1);

    /*once N travelers(threads) have arrived and cross checked the board
bus as batch*/
    while(count1<=(N-1));

    boarding_bus();
}

void* traveler(void* input){

    /*travelers are here before entering the waiting area*/
    sem_wait(&sem1);
    /*if enough room in waiting area then they may enter*/
```

```
    cross_check();
}


int main(){

    sem_init(&sem1, 0, N);
    pthread_mutex_init(&lock1, NULL);
    pthread_mutex_init(&lock2, NULL);
    pthread_mutex_init(&lock3, NULL);

    /*each new thread is a traveler*/
    while(1){
       pthread_t id;
       pthread_create(&id, NULL, traveler, NULL);
       sleep(1);

    }

}
```

To make an algorithm with specification as described you must do the following…

First in the main thread initialize a semaphore to N and have 3 mutex locks initialized

In the main thread continuously spawn threads(preferably disjointed), these will act as the travelers. Each traveler thread will call the function traveler.

In the traveler function call sem_wait and then call cross_check. Sem_wait is part of the algorithm such that only N number of threads will be able to call cross_check. Cross_check takes place in the waiting area. We only want N number of travelers(threads) in the waiting area at a given time.

The cross_check function is our waiting area once cross check is complete we want all the travelers(thread) to leave to board the bus at the same time. This is accomplished using a mutex lock and a counter. Once the counter counts up to a certain value it release all the travelers(threads) to the bus at THE SAME TIME. We then reset the counter later on in the boarding_bus function.

In the boarding bus function the biggest challenge was figuring out how to reset the counter back to 0 in a thread-safe manner. In the boarding bus function we continue to iterate counter1 and trap the threads in a spin lock. But once all the N threads have arrived into this function the spinlock releases and the last thread that gets released from the spinlock resets the counter to 0 and re-initializes the semaphore so the waiting room can take N more new travelers.

The code I included above is an implementation of my algorithm if you would like to test it.

Problem 7:

Part 1

In my solution I decided to use semaphores. The reason I decided to use semaphores was because it allows a certain number of threads into a critical section unlike mutex locks which only allows one. So in case we have let's say 5 taxis and 3 consumers; we can pair the 3 consumers and 3 taxis simultaneously.

So this is how I plan on solving this problem…

In the main thread I will have create a thread that will randomly generate a number of taxis and consumers continuously so we don't run out. Each taxi and consumer are added to their own thread-safe blocking queue. And I will init the semaphore with the initial value of the min between the value of the taxis and consumers. In this was we can accomplish quicker execution as with the 5 taxis, 3 consumers example above. After that we continuously generate a random amount of taxis and consumers and enqueue onto the queue. After that we sem_post since we at least have one more taxi and consumer in the queue.

Back in our main thread in a loop we create new disjointed threads that either call taxi or consumer functions that basically do the same thing. But before we create the threads we call sem_wait to make sure we have at least one taxi and one consumer. The thread will then go ahead and dequeue one taxi and one consumer.

I added sleep to this code to add a level of randomness which is what we would expect in real life scenarios.

Part 2

C Psuedocode

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unist.h>
```

```
sem_t sem;

/*blocking... thread safe queue*/
Queue taxis;
Queue consumers;

void done(int i){

    if(i == 1)
       printf("customer is picked up");

    else
       printf("taxi has been boarded");

}

void *consumer(void *input){

    /*consumer found taxi*/
    consumers.dequeue();
    taxis.dequeue();
    done(2);

}

void *taxi(void *input){

    /*taxi found consumer*/
    taxis.dequeue();
    consumers.dequeue();
    done(1);
}

/*Generates random number of arrivals of
 * consumers and taxis*/
void *generating(void *input){


    int t = (some random amount %3) + 1;
    int c = (some random amount %3) + 1;

    taxis.enqueue() t times;
    sleep(some random amount % 2);
```

```
    consumers.enqueue() c times;
    sleep(some random amount % 2);

    int min = min(t,c);
    sem_init(&sem, 0, min);    //this is the amount of taxis that can be
matched to consumers

    do{
       t = (some random amount %3) + 1;
       c = (some random amount %3) + 1;

       taxis.enqueue() t times;
       sleep(some random amount % 2);

       consumers.enqueue() c times;
       sleep(some random amount % 2);

       sem_post(&sem);    //we are guaranteed atleast 1 matching pair so we
can increase our semaphore
    }while(1);

}


int main(){
    pthread_t tid1;
    pthread_create(&tid1, NULL, generating, NULL);


    while(1){


       if(taxis.isEmpty==false){
             sem_wait(&sem); //make sure enough taxis and consumers
             pthread_create(NULL, disjointed, taxi, NULL);
       }


       if(consumers.isEmpty==false){
             sem_wait(&sem); //make sure enough taxis and consumers
             pthread_create(NULL, disjointed, consumers, NULL);
       }
```

```
        sleep(some random amount % 2);

    }


}
```

Problem 6

The two lines of the code that prevents this system from working are from 15 and 16. We can fix these errors by

if(readcount ==1)

P(w_or_r);

V(mutex);

The readcount need to be initialized to 1 so that there is at least one reader that belongs in the critical section. Without this code the readcounts are being read when the writers aren't synchronized. In general writer synchronization should not happen before unlocking its consequent readers.