OPERATING SYSTEMS AND NETWORK

By: Prisha

Roll:2021101075

Assignment 5

Question 1:

So, here I'll explain concept / approach used by me for this question

- Initially the input was taken as array of structs where structure of the struct was like:

```
struct student_thread
{
    int entry_time;
    int washing_time;
    int patience_time;
    int index;
    pthread_cond_t signal;
};
```

- Semaphore was initialized for m machines which prevented multiple students to use machine at once, along with it an array of thread was created as students are being considered thread, so array of thread for students. And initializing the lock for counter (that maintains count of number of students who washed / went away).

```
pthread_mutex_init(&variable_lock, NULL); // FOR THE STUDENT COUNTER
pthread_t thread[N];
sem_init(&machine_semaphore, 0, M); // SEMAPHORE FOR M MACHINES
for (int i = 0; i < N; i++)
{
    pthread_cond_init(&struct_array[i].signal, NULL);
}
```

- Since it was desired that students should use washing machines in order of how they came so for this reason array of sorted based on arrival time. To sort, bubble sort was used although qsort can also be used (and is commented just for reference). Here is how the array was sorted.

```c
void sort(struct student_thread *studentthread, int N)
{
    int i, j;
    for (i = 0; i < N - 1; i++)
    {
        for (j = i + 1; j < N; j++)
        {
            struct student_thread *i1 = studentthread + i;
            struct student_thread *i2 = studentthread + j;
            if (i1->entry_time > i2->entry_time)
            {
                struct student_thread temp = *i1;
                *i1 = *i2;
                *i2 = temp;
            }
            else if (i1->entry_time == i2->entry_time)
            {
                if (i1->index > i2->index)
                {
                    struct student_thread temp = *i1;
                    *i1 = *i2;
                    *i2 = temp;
                }
            }
        }
    }
}
```

- Threads were created using pthread_create function which called main_function where we used semaphore and mutex locks in order to ensure that multiple students don't use machine at the same time, busy waiting isn't there and principal pf fcfs is served as will be explained . Note input to the main function was passed as struct.

```c
for (int i = 0; i < N; i++)
{
    if (i != 0 && struct_array[i].entry_time == struct_array[i - 1].entry_time)
    {
        sleep(0.01); // IN CASE THEY ARRIVE EQUALLY INTROUDUCING A DELAY.
    }
    struct input *arg = malloc(sizeof(struct input));
    arg->entry_time = struct_array[i].entry_time;
    arg->index = struct_array[i].index;
    arg->M = M;
    arg->N = N;
    arg->patience_time = struct_array[i].patience_time;
    arg->iterator = i;
    arg->washing_time = struct_array[i].washing_time;
    // SENDING ARGUMENT AS THIS STRUCT INPUT
    pthread_create(&thread[i], NULL, &main_function, arg); // STUDENTS SERVE AS THREADS SO TOTAL
}
```

- So, in main function as the thread was created it was sent to sleep for the arrival time. Here time gone variable is used to keep track of the time person waited before he went back.

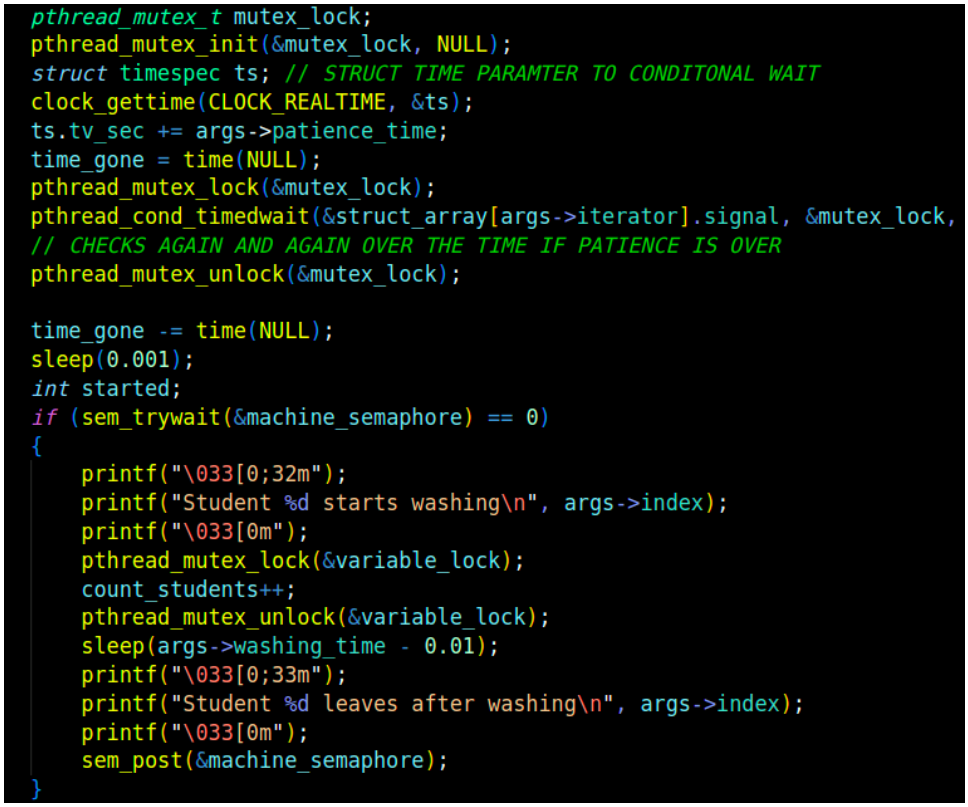
```c
void *main_function(void *myarg)
{

    struct input *args = (struct input *)myarg;
    sleep(args->entry_time - 0.01);
    int time_gone = 0; // CHECK HOW MUCH TIME PERSON WAITED AFTER HE ARRIVED

    printf("\033[0;37m");
    printf("Student %d arrives\n", args->index);
    printf("\033[0m");
```

- After this there are two possible cases, either sem_wait would return 0 i.e. one of the machines is unlocked or it would return a non-zero value indicating a non-empty machine. SO, in the initial case we say person starts washing and then make it sleep for time required for washing clothes, then sem_post to unlock the machine . AND the counter is incremented within locks so that multiple threads don't make changes to counter simultaneously.

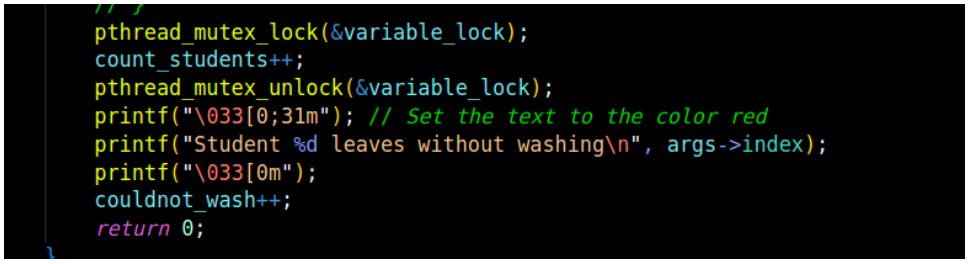
```c
if (sem_trywait(&machine_semaphore) == 0) // CHECKS IF ANY OF MACHINE IS FREE
{

    printf("\033[0;32m");
    printf("Student %d starts washing\n", args->index);
    printf("\033[0m");
    pthread_mutex_lock(&variable_lock); // COUNT _STUDENTS KEEPS TRACK OF STUDENTS COMING
    count_students++;
    pthread_mutex_unlock(&variable_lock);
    sleep(args->washing_time - 0.01); // SLEEPS FOR WASHING TIME

    printf("\033[0;33m");
    printf("Student %d leaves after washing\n", args->index);
    printf("\033[0m");

    sem_post(&machine_semaphore); // unlock THE MACHINE
}
```

- IN the latter case where sem_wait returns a non-zero value, it means none of the machines is free, so we use conditional pthread for time wait, struct time spec is used to pass argument to conditional pthread function call, it checks regularly in patience time is semaphore got unlocked by sending signal.

If semaphore returns zero, the same is executed else count of student who went back without washing is incremented.



```
pthread_mutex_t mutex_lock;
pthread_mutex_init(&mutex_lock, NULL);
struct timespec ts; // STRUCT TIME PARAMTER TO CONDITONAL WAIT
clock_gettime(CLOCK_REALTIME, &ts);
ts.tv_sec += args->patience_time;
time_gone = time(NULL);
pthread_mutex_lock(&mutex_lock);
pthread_cond_timedwait(&struct_array[args->iterator].signal, &mutex_lock,
// CHECKS AGAIN AND AGAIN OVER THE TIME IF PATIENCE IS OVER
pthread_mutex_unlock(&mutex_lock);

time_gone -= time(NULL);
sleep(0.001);
int started;
if (sem_trywait(&machine_semaphore) == 0)
{
    printf("\033[0;32m");
    printf("Student %d starts washing\n", args->index);
    printf("\033[0m");
    pthread_mutex_lock(&variable_lock);
    count_students++;
    pthread_mutex_unlock(&variable_lock);
    sleep(args->washing_time - 0.01);
    printf("\033[0;33m");
    printf("Student %d leaves after washing\n", args->index);
    printf("\033[0m");
    sem_post(&machine_semaphore);
}
```



```
    pthread_mutex_lock(&variable_lock);
    count_students++;
    pthread_mutex_unlock(&variable_lock);
    printf("\033[0;31m"); // Set the text to the color red
    printf("Student %d leaves without washing\n", args->index);
    printf("\033[0m");
    couldnot_wash++;
    return 0;
}
```

- Now after this say if multiple threads are waiting for the machine, then principle of FCFS is followed for the waiting processes where for the intermediate time process is sent to sleep. This ensures that the process that came first should be executed just as the machine gets unlocked as in ensure fcfs function.

```c
void ensure_fcfs(struct input *args, int time_already, int students)
{
    int time_come = args->entry_time + args->washing_time + time_already; // COMPARES WHO CAME FIRST
    int abhi_time = struct_array[students].entry_time;
    if (time_come < abhi_time)
    {
        sleep(abhi_time - time_come - 0.01); // FCFS IS ENSURED USING SLEEP
    }

    pthread_cond_signal(&struct_array[students].signal);
}
```

- After the complete process mutex lock were destroyed using kill function, threads were joined using pthread_join

```c
void kill_function(int N)
{
    pthread_mutex_destroy(&variable_lock);
    for (int i = 0; i < N; i++)
    {
        pthread_cond_destroy(&struct_array[i].signal);
    }
}
```

- Since we already have total students as count for the students who left without eating hence this can be used to calculate percentage if less than 25 percent than no washing machine is needed else we will require the washing machine.

```c
printf("%d\n", couldnot_wash);
int threshold = 0.25 * N; // CHECKING IF PEOPLE WHO WENT WITHOUT WASHING IS GREATER THAN 25 % OR NOT
if (couldnot_wash > threshold)
{
    printf("\033[0;37m");
    printf("YES\n");
    printf("\033[0m");
}
else
{
    printf("\033[0;37m");
    printf("NO\n");
    printf("\033[0m");
}
return 0;
```

QUESTION 3 :

Although I didn't code this question up but here is the approach that I was thinking to use , so it's a server client problem which can be done as :

- ✓ We initially take the input of nodes and the corresponding delay.
- ✓ So, since they are doing full precomputing hence, we are supposed to go to each of the nodes, ping all its Neighbours and get their Neighbours since a node has info only about its immediate neighbors in beginning.
- ✓ Then we precompute the routing table using any of the shortest path algorithms say Dijkstra.
- ✓ The we initialize the N nodes .basically N sockets of different ports and make another thread using thread create which communicates with the client then using routing table made earlier we send it to socket.