# Back To College

## Logic

In my logic each of the entities are threads i.e we have Student Threads, Vaccination Zone Threads and Company Threads. So the student threads busy wait on Vaccination Zone, until they get one and Vaccination Zones busy wait on Companies for a batch of vaccines. So we have arrays of structures of each entity and there is mutex lock per Company and Vaccination Zone. There is also a global mutex for the whole system. **The code is Commented Extensively to explain the logic**

## Implementation and Code-Break Down

### Student

The `student()` is the function that is passed as thread. It sleeps initially for sometime to simulate coming of students at random time. A student runs till he has been tested positive for antibodies or he has completed 3 rounds of vaccination. It calls the `waitforVZone()` where the student waits and gets vaccinated. When a student it also decreases the global variable which stores the number of students that are left to be processed. I am using a common mutex lock for the global variables that represent the whole system.

```
// Student Thread
void *student(void *studentargs)
{
    sleep(getRandom(1, 5)); //Sleep for a random time between 1-5
    int id = ((struct Student *)studentargs)->id;
    while(students[id].status!=1&&students[id].round<=3){  //while the
student has not participated in three rounds and he has not been tested
antibody positive participate in vaccination
        printf("\n\033[0;36mStudent %d has arrived for %d round of
Vaccination\n\033[0m",id,students[id].round);
        waitforVZone(id); //wait for zone allocation
    }
    if(students[id].status!=1){ //if he exited the loop after getting
rounds greater than 3 sedlyf for him.
        printf("\n\033[0;36mStudent %d is leaving for home. The college
gave up on him. Sedlyf\n\033[0m", id);
    }
    pthread_mutex_lock(&global_lock);//concurrency
    num_students_left--;//he left reduce number of students left
    pthread_mutex_unlock(&global_lock);
    return NULL;
}
```

The `waitforVzone()` is called by a student thread. A student busy waits here to get a slot to be available at a vaccination. For the student to register the vaccination zone should be not in vaccination phase and should have atleast one slot left. When the student access a vaccination zone data represented by

`zones[i]` I lock that `i` th zone with `zones[i].vaccination_lock` which is a mutex lock for cuncurrency so that no other entity accesses it at the same time creating issues. I am also using global lock for concurrency in global variables representing the whole system. Now once the student finds the vaccination zone with free slots and is not in vaccination phase it enters it and reduces its slots. I also reduce the number of waiting students. Now this student will busy wait here for the vaccination zone to enter into vaccination phase and get vaccinated. Once the zone enters vaccination phase the student stops busy waiting and gets vaccinated. So it reduces the number of vaccines left and reduces the registered slots. Now the student proceeds to test himself. To test the function calls `isSuccess()` which returns 1 if the student is antibody negative otherwise 0. So if it is zero the student can enter college or he has to appear for another round. This is kept track of by the status attribute of student structure. The code snippet below explains it with comments.

```
//the student gets vaccinated here
void waitforVZone(int id){
    int i=0;
    pthread_mutex_lock(&global_lock);//lock
    printf("\n\033[0;36mStudent %d is waiting to be allocated a slot on a
vaccination zone\n\033[0m", id);
    waitingStudents++; //A new student is waiting so add him
    pthread_mutex_unlock(&global_lock);//unlock

    while (1)
    {
        pthread_mutex_lock(&zones[i].vaccination_lock); // lock so number
of slots do not get accessed by many students at the same time.
        if (zones[i].numslots >= 1 && !zones[i].isVaccinating) // allow a
student to get a slot when the zone is not in vaccination phase and there
are slots left
        {   pthread_mutex_lock(&global_lock);
            waitingStudents--;// A student is assigned a slot, so reduce
waiting students
            pthread_mutex_unlock(&global_lock);
            zones[i].numslots--;//This student took a slot
            printf("\n\033[1;36mStudent %d has been assigned a slot on the
vaccination zone %d. Waiting to be vaccinated\n\033[0m", id, i);
            pthread_mutex_unlock(&zones[i].vaccination_lock);//unlocks the
lock so that multiple students can get vaccinated at the same time at the
same vaccination zone

            while(!zones[i].isVaccinating){
                //busy wait to start vaccination
            }
            sleep(1); //simulate time to vaccinate
            printf("\n\033[1;36mStudent %d on Vaccination Zone %d has been
Vaccinated which has success Probabilty %lf\n\033[0m", id,
i,zones[i].currp);
            pthread_mutex_lock(&zones[i].vaccination_lock);//lock zone for
concurrency
            zones[i].batchleft--;//This student will get vaccinated here so
reduce size to simulate simultaneous vaccination on signal.
            zones[i].vaccinatedslots--;//reduce vaccinatedslots
```

```
            pthread_mutex_unlock(&zones[i].vaccination_lock);//unlock
            int status= isSuccess(zones[i].currp);
            if(!status){
                sleep(1); //simulate time for testing
                printf("\n\033[1;32mStudent %d  has been tested positive
for antibodies\n\033[0m",id);
                students[id].status=1; // he is free to go
                return;
            }
            else{
                sleep(1); //simulate time for testing
                printf("\n\033[0;31mStudent %d  has been tested negative
for antibodies\n\033[0m",id);
                students[id].status=0; //he has to start all again
                students[id].round++;//increment rounds
                return;

            }
        }
        pthread_mutex_unlock(&zones[i].vaccination_lock);// unclock if the
if statement didn't get execute
        i = (i + 1) % m; // go through all zones again
    }

}
```

```
struct Student
{
    int id; //stores id of student
    int status; //stores status of student
    int round; //stores the number of rounds the student participated in

};
```

### Vaccination Zone

This thread runs till all students are processed. It uses `zones[id].vaccination_lock` and `companies[i].company_lock` mutexes to lock the zone represented by id and company represented by `i`. The thread runs till all students are processes that is `while(num_students_left>0)`. Initially we have no slots and we are not in vaccinating phase. Now we busy wait on companies to get some vaccines `while(num_students_left)` if a company has a batch I take its data and reduce its batches after locking it and after taking we unlock it. Then I break from the inner loop. Now we produce to slot declaration. The value of slots is derived as told in the assignment pdf. We busy wait for there to be some waiting students. But if there are waiting students we move on to declare slots. If `num_students_left` becomes 0 when we wait, we leave the thread. Now we declare k, the no. of slots as `k = (getRandom(1,minna(8,zones[id].batchleft,waiters)));` where `minna(a,b,c)` returns minimum of 3 nos. Then I update the vaccination zone and the zone declares its slot. Now it busy waits for the students to register i.e. `while(zones[id].numslots>0 && waitingStudents>0)` Now if all slots

are filled or there are no waiting students the zone moves on to Vaccination phase. Now it busy waits for all registered students to get vaccinated i.e. `while(zones[id].vaccinatedslots)`. After all students are vaccinated, the vaccination zone checks if it has more vaccines, `if(zones[id].batchleft>0)` we goback to slot declaration, otherwise it means that we used all the vaccines that we took from that company. We inform that company by doing `companies[mycompany].deliverable--` and zone moves into non vaccinating phase, and the process continues from the start.

```
void *vZone(void * zoneargs)
{
    int id = ((struct VZone *)zoneargs)->id; //id of vzone
    int i=0;
    int k;//slots in iteration
    int mycompany;//the company from where we got the vaccines from
    while (num_students_left > 0)  //run the thread till number of students
is greater than zero
    {
        pthread_mutex_lock(&zones[id].vaccination_lock); //lock zone for
concurrency
        zones[id].isVaccinating=0; // the zone is not vaccinating currently
        pthread_mutex_unlock(&zones[id].vaccination_lock);//unlock
        mycompany=-1; // the company that gave it vaccines, now no one
        while(num_students_left){ //while there are some students left
            pthread_mutex_lock(&companies[i].company_lock);//lock so that
only one one zone gets access to a company
            if(companies[i].r>=1){ //if this company has a batch take it
                companies[i].r--; //we took a batch
                pthread_mutex_lock(&zones[id].vaccination_lock); //lock
this zone for concurrency
                zones[id].batchleft=companies[i].p; //update this zone
                zones[id].currp=companies[i].x;
                mycompany=companies[i].id;
                pthread_mutex_unlock(&zones[id].vaccination_lock); //unlock
zone, company
                pthread_mutex_unlock(&companies[i].company_lock);
                printf("\n\033[1;34mPharmaceutical Company %d is delivering
a vaccine batch to Vaccination Zone %d which has success probabiltity
%lf\n\033[0m",mycompany,id,companies[i].x);
                sleep(1);//simulate time
                printf("\n\033[1;33mPharmaceutical Company %d has delivered
vaccines to Vaccination Zone %d, resuming vaccination
now\n\033[0m",mycompany,id);
                break; //we got a batch now need not search for more
companies, so break and wait for slots to fill.

            }
            pthread_mutex_unlock(&companies[i].company_lock);
            i = (i + 1) % n;
        }
        int waiters;//the no of waiters
        slot_declaration:
        while((waiters=waitingStudents)<=0){ //wait till we have some
waiting students to move onto slot declaration
```

```
            //here we busy wait for some students to arrive
            if(num_students_left==0){
                return NULL;
            }
        }
        k = (getRandom(1,minna(8,zones[id].batchleft,waiters))); // get
slots that are to be allocated

        pthread_mutex_lock(&zones[id].vaccination_lock); // lock this zone
we are going to update, concurrency matters
        zones[id].numslots=k; //update
        printf("\n\033[1;33mVaccination Zone %d is ready to vaccinate with
%d slots\n\033[0m",id,k);
        zones[id].isVaccinating=0;
        printf("\n\033[1;33mVaccination Zone %d is not in Vaccination Phase
currently\n\033[0m",id);
        pthread_mutex_unlock(&zones[id].vaccination_lock);//unlock, before
busy waiting
        while(zones[id].numslots>0 && waitingStudents>0){
            //busy wait for maximum slots to get filled , if there are no
waitingstudents then enter vaccination phase
            if(num_students_left==0){
                return NULL;
            }
        }
        pthread_mutex_lock(&zones[id].vaccination_lock);//lock for
concurrency
        zones[id].vaccinatedslots=k-zones[id].numslots; //vaccinatedslots
has no of students who registered for vaccination
        printf("\n\033[1;33mVaccination Zone %d entering Vaccination
Phase\n\033[0m",id);
        zones[id].isVaccinating=1; //now entering vaccination phase
        pthread_mutex_unlock(&zones[id].vaccination_lock);//unlock before
waiting
        while(zones[id].vaccinatedslots){
            //wait for all of em to get vaccinated
            if(num_students_left==0){
                return NULL;
            }
        }
        pthread_mutex_lock(&zones[id].vaccination_lock);//for concurrency
        if(zones[id].batchleft>0){ // there are some more vaccines left in
this batch, going to allocate more slots
            pthread_mutex_unlock(&zones[id].vaccination_lock);//unlock to
avoid deadlocks
            goto slot_declaration;// go back to declaring more slots
        }
        zones[id].isVaccinating=0;// now it is not vaccinating
        zones[id].numslots=0;//set slots back to zero
        printf("\n\033[1;33mVaccination zone %d has run out of
Vaccines\n\033[0m",id);
        pthread_mutex_unlock(&zones[id].vaccination_lock);
        pthread_mutex_lock(&companies[mycompany].company_lock);
        companies[mycompany].deliverable--;// reduce the batch of the
```

```
company from which we took vaccines from
        pthread_mutex_unlock(&companies[mycompany].company_lock);

    }

    return NULL;
}
```

### Pharmaceutical Company

A Pharmaceutical Company thread runs till there are some students left to be processed that is `while(num_students_left>0)`. It gets a random number between 1,5 (inclusive) as batches. It gets a number between 10,20 (inclusive) as the number of vaccines in each batch. Then it sleeps for some time to simualate production. Then we lock this company and update it. We use the company lock of this company. Then the company busy waits for all vaccines to get over. After everything gets over it moves on to produce more.

```
void *pcompany(void *companyargs)
{
    int id = ((struct Pcompany *)companyargs)->id; //id of company
    int r,p;
    int flag;
    while (num_students_left > 0) //continue the process until all students
are processed
    {

        r=getRandom(1,5); // get the number of batches to prepare
        p=getRandom(10,20); // get the number of vaccines
        printf("\n\033[1;34mPharmaceutical Company %d is preparing %d
batches of vaccines which has success probability
%lf\n\033[0m",id,r,companies[id].x);
        sleep(getRandom(2, 5)); //sleep for time to simulate manufacturing
        printf("\n\033[1;34mPharmaceutical Company %d has prepared %d
batches of vaccines which has success probability %lf. Waiting for all
vaccines to be used to resume production\n\033[0m",id,r,companies[id].x);
        pthread_mutex_lock(&companies[id].company_lock);//lock the company
for concurrency
        companies[id].p=p; //set the new batch
        companies[id].r=r;
        companies[id].deliverable=r;
        pthread_mutex_unlock(&companies[id].company_lock);//unlock
        // now busy wait for all batches to get over
        while(companies[id].deliverable){ //deliverable is the variable
that keeps count of batches that are in still in use. when there are no
batches left continue production
                if(num_students_left==0){
                    return NULL;
                }
        }
        printf("\n\033[1;34mAll the vaccines produced by Pharmaceutical
```

```
Company %d are emptied. Resuming Production now\n\033[0m",id);
        //resume

    }
    return NULL;
}
```

### Structures and Global Variables

These are explained by comments and names. They are trivial.

```
// the inputs
int n,m,o;
//num_students_left in simulation
int num_students_left;
//num of waiting students
int waitingStudents;
//mutex_lock_for global variables
pthread_mutex_t global_lock;

struct Pcompany
{
    int id;//id of company
    double x;//probabilty of success
    int p;//no. of vaccines in each batch
    int r;//no. of batches left
    pthread_mutex_t company_lock;//lock
    int deliverable;
};

struct Pcompany *companies;

struct VZone
{
    int id; //id of zone
    int batchleft; //no. of vaccinizes left
    int numslots;// no. of slots left
    double currp;// the curr probabilty
    int isVaccinating;// 1 if vaccinating otherwise zero;
    int vaccinatedslots;//keeps count of vaccinated slots
    pthread_mutex_t vaccination_lock; //the lock

};

struct VZone *zones;

struct Student
{
    int id; //stores id of student
    int status; //stores status of student
    int round; //stores the number of rounds the student participated in
```

```
};
struct Student *students;
```

### Helper Functions

```
//gets a random number including lower and upper bounds
int getRandom(int lb, int ub)
// returns success of vaccine on a student given probabilty
int isSuccess(double p)
//return min of 3 nos
int minna(int num1, int num2 , int num3)
```

### Main

It initialises, takes inputs, launches threads and joins them back. Then it moves on to cleaning up and exits.