System Programming Midterm Report

Yusuf Patoglu

May 2, 2021

1 Problem Definition

This project aims to achieve **Interprocess Communication** between processes. We will be simulating a clinic where nurses bring vaccines to the clinic from the depot, vaccinators apply vaccines to citizens and citizens are waiting to be called. The communication between processes will be provided by **POSIX Named Semaphores**.

2 Program flow

- ***Parsing command-line arguments:*** The program starts by parsing command-line arguments with *getopt* function. If the parameters are not valid then it returns an error message and terminates the program.

```
void
parse_args(int argc, char**argv, int *n, int *v, int *c, int
    *b, int *t, char *input_file_path);
```

- ***Creating and mapping the POSIX sharing memory:*** To open and map a shared memory we have to follow these steps:

```
    /*Open shared memory object.*/
    s_fd = robust_shm_open(shared_mem_name, O_RDWR | O_CREAT);

    /*Allocate space for shared memory.*/
    robust_ftruncate(s_fd, sizeof(struct shared_area));

    /*Map the shared memory into a local pointer.*/
    ptr = robust_mmap(NULL, sizeof(struct shared_area),
        PROT_READ | PROT_WRITE, MAP_SHARED, s_fd, 0);
```

- ***Creating and mapping the POSIX sharing memory:*** Just after gathering a local pointer to a shared memory now we are good to initialize the shared memory.

  We don't need to have protection here because we are **100%** sure that only the parent process is accessing here. I will talk about the semantics of the variables in the **Shared Memory** section.

```
/*Initialize the shared memory.*/
ptr = robust_mmap(NULL, sizeof(struct shared_area),
    PROT_READ | PROT_WRITE, MAP_SHARED, s_fd, 0);

ptr->vaccine_1 = 0;
ptr->vaccine_2 = 0;
ptr->counter = 0;
ptr->finished_nurse_count = 0;
ptr->total_vaccine = 2 * t * c;
ptr->read = 0;
ptr->curr_citizen_count = c;
```

  The shared memory is kept in the global area so those variables are already initialized to zero. But I reinitialized them for the sake of readability.

- ***Opening POSIX Named Semaphores:*** After finishing about the shared memory operations now it's time to create **6** semaphores. We could create the semaphores before creating the shared memory. The order doesn't matter. Named semaphores are generally designed for unrelated processes but we can use them for related processes too.

  I will explain the purpose of each semaphore in **Semaphores** section. One of the semaphores are opened like this:

```
mutex = sem_open(_mutex, O_CREAT | O_EXCL, 0666, 1);
if(mutex == SEM_FAILED) //The process is either failed or
    it tries to open second time.
{
    perror("sem_open");
    exit(EXIT_FAILURE);
}
```

- ***Spawning Processes:***: Since we have a number of distinct processes we're good to create them. Simply one parent process will spawn $N + V + C$ processes and wait for them:

```
 /*Create n + v + c processes.*/
//Some of the deallocations are deleted for simplicity in
    LaTeX
for(i = 0; i < n + v + c; i++)
{
    cur_pid[i]= fork();
    if(cur_pid[i] == 0)
    {
        robust_close(filedes[0]);
        //Nurse Section.
        if(i < n){
            nurse(fd,n, t, c, i, b);
            robust_semclose(mutex);
            free(cur_pid);
            _exit(EXIT_SUCCESS);
        }
        //Vaccinator section.
        else if((i >= n) && (i < n + v)){
            int vaccinated_count = vaccinator(fd, n, t, c,
                b, i);
            sprintf(child_message, "Vaccinator %d (pid=%d)
                vaccinated %d doses.\n", i - n + 1,
                getpid() ,vaccinated_count);
            robust_write(filedes[1], child_message,
                sizeof(child_message));
            robust_semclose(mutex);
            free(cur_pid);
            _exit(EXIT_SUCCESS);
        }
        //Citizen section.
        else if(i >= n + v){
            citizen(fd, n, t, c, v, i);
            robust_semclose(mutex);
            free(cur_pid);
            _exit(EXIT_SUCCESS);
        }
    }
    else if(cur_pid[i] == -1){
        perror("Fork:");
        fprintf(stderr, "FORK FAILURE. EXITING.\n");
        robust_semclose(mutex);
        free(cur_pid);
        exit(EXIT_FAILURE);
    }
}
```

- ***Parent waits for all child processes and listens to vaccinators:*** All processes including the parent spawned simultaneously. In order to close the clinic and print the information, the parent process will wait for all the child processes with the help of the wait system call. The parent also gathers some information from the vaccinator processes. With the help of unnamed pipes, vaccinator processes write their number and how many times did they apply vaccines to citizens.

  ```
  for(i = 0; i < n + v + c; i++) {
      int status = 0;
      pid_t childpid = wait(&status);
  }
   while(robust_read(filedes[0], child_message,
       sizeof(child_message)))
   {
      printf("%s\n", child_message);
   }
  ```

  There could be other design choices here:

  - ***Vaccinator processes return their EXIT_STATUS to parent:*** I think this would be the most elegant and simple solution. But exit statuses are limited to 256 bits. So if a vaccinator has applied more than 255 vaccines, then that would be a problem so I didn't choose this method.
  - ***Parent waits for all pipes to close:*** This would be a nice solution too because pipes have a blocking mechanism provided by the kernel. But we don't interact with all processes like nurse and citizen processes so I didn't choose this too.

3 Shared Memory

In the **Program flow** section I explained how I created and initialized the shared memory. Now it's time to talk about its content. The struct which mapped into shared memory is like this:

```
struct
shared_area
{
    int vaccine_1;
    int vaccine_2;
    int counter;
    int finished_nurse_count;
    int total_vaccine;
    int read;
    int last_vaccinator_pid;
    int last_vaccinator_num;
    int curr_citizen_count;
    //int vaccinator_pids[]
}shared_area;
```

- ***vaccine_1 and vaccine_2:*** As the name implies they hold the total count of different types of vaccines.

4 Semaphores

The keystone of the synchronization is the semaphores. They're integers provided by kernel which prevents busy waiting. The limit of semaphore count was 7 and I exactly used 7 semaphores:

- ***mutex:*** This semaphore behaves like a mutex so I named it as mutex. It protects the shared area between the producer and consumer(In our case nurse and the vaccinator.)
- ***items_1 and items_2:*** These semaphores represent the first and second type of vaccins which is type 1 and 2.
- ***spaces:*** This semaphore represents the available slots in the buffer.
- ***bring_citizen:*** This semaphore is for signalling the citizen by the vaccinator.
- ***mutex_citizen:*** This semaphore is for protecting the shared memory inside the citizen function.
- ***vaccinate:*** This is another mutex variable for protecting single shared integer variable. Actually GCC provides that they'll be atomic but I took precautions.

5 Actors:

In this section I will talk about actors and their implementations.

– ***Nurses:*** Nurses have the producer role in our problem.(Multiple Producers). We enter our Nurse function by waiting two semaphores like in classical producer model. So in this case If the producer arrives when the buffer is full, it will wait for a customer to remove an item before proceeding.

```
/*We will write one byte so space--;*/
robust_sem_wait(spaces);
/*Lock*/
robust_sem_wait(mutex);
```

We know that nurses will bring vaccines from the depot. But only one vaccine will be gathered for each turn. To do that we have to read the file in synchronised fashion.

Therefore I've kept an offset on my shared memory and in order to prevent race conditions between processes I used *pread* system call instead of *read* system call. That way the offset of the file never changes we give the offset with the help of shared memory offset variable. So we can read file without a problem:

```
/*Read single byte from file with the help of
    ptr->counter offset..*/
read_count = robust_pread(fd, buf, 1, ptr->counter);
/*Increment the offset*/
ptr->counter++;
/*Don't forget to place null character*/
buf[read_count] = '\0';
/*Convert str to int*/
vaccine_type = atoi(buf);
```

Since we've read the next character from file now we have to increment the number of vaccines according to their types:

```
if(vaccine_type == 1){
    ptr->vaccine_1++;
    printf("Nurse %d (pid = %d) has brought vaccine 1:
        the clinic has %d vaccine1 and %d vaccine2.
        %d\n",
    i+1,getpid(), ptr->vaccine_1, ptr->vaccine_2,
        ptr->counter);}
else if(vaccine_type == 2){
```

```
            ...same operation...
        }
```

After incrementing the specified vaccine now we're good to release our mutex semaphore and post the specified semaphore(item1 or item2). But before that I have to mention about the end case of the nurse function. We know that we have at least *2 * t * c* vaccines in our buffer. We can compare our file offset value with *2 * t * c* to understand if the storage is run out of vaccines or not:

```
        if(ptr->counter >= 2 * t * c)
        {
            ptr->finished_nurse_count++;
            /*If all nurses have terminated, notify.*/
            if(ptr->finished_nurse_count == n)
            {
                printf("Nurses have carried all vaccines to the
                    buffer, terminating.\n");
            }
            /*Let consumers terminate.*/
            robust_sem_post(items_1);
            robust_sem_post(items_2);
            /*We didn't add anything to buffer, increment the
                spaces back.*/
            robust_sem_post(spaces);
            /*Unlock and exit. Two statements are not atomic
                together but no problem with that.*/
            robust_sem_post(mutex);
            return;
        }
```

Here another point is we're posting both vaccines to let consumers terminate. Since they're protected by a mutex semaphore we can do it sequentially. We also increment the finished nurse count to print required statement. After posting both vaccines now we're good to release mutex, return and release sources for that nurse process.

– ***Vaccinators:*** Vaccinators have the consumer role in our problem.(Multiple Consumers). Our produces were producing items one by one but in consumers they will consume it two by two. Therefore we need two semaphores for two types of vaccines. Actually we could do it with one semaphore but then we would have to check the distinct vaccines like this in consumer. So this would be **wrong** and an example for poor synchronization:

```
/*If one of the vaccines are missing we shouldn't consume
    anything. We should wait producer to add the missing
    vaccine. But we're relying on context switches here.
    BAD!!!*/
       if(ptr->vaccine_1 <= 0 || ptr->vaccine_2 <= 0)
       {
           printf("Busy waiting\n");
           /*Check if the producers brought all of the
               vaccines or not.
           /*We didn't consume two vaccines, increment them
               again*/
           robust_sem_post(items_1);
           robust_sem_post(items_2);
       }
```

As explained in the comment line we're relying on kernel for context switches. Lots of CPU cycles might be lost there. So in order to prevent that we're using two semaphores. We're starting by waiting two semaphores:

```
/*We'll be popping 2 elements so call items twice.*/
      robust_sem_wait(items_1);
      robust_sem_wait(items_2);
      robust_sem_wait(mutex);/*Lock*/
```

After popping two elements we have to bring citizen to apply vaccine. But before that let's talk about end condition of a consumer:

```
/*If buffer's size is an odd number then the remaning vaccine
    will be 1. If it's even then it will be 0.*/
       if(ptr->total_vaccine <= 1){
           /*We didn't consume 2 items, give them back and
               break*/
           robust_sem_post(items_1);
           robust_sem_post(items_2);
           robust_sem_post(mutex);
           break;
       }
```

Actually it's specified that the buffer says will always be an even number. I could do it as == 0. But they both refer to the exact same case. So here total_vaccine is a global variable that starts with the 2 * t * c value. it's decremented by two for each iteration of consumer. In short it says; if there are no more vaccines give the two elements back, release the mutex and break from the loop to free resources.

The else condition of the end condition is the consume part. Here we increment the vaccinated count and decrement both of the vaccines and after that we signal/bring citizen to apply vaccine.

```
ptr->total_vaccine-=2;
vaccinated_count+=2;
ptr->last_vaccinator_pid = getpid();
ptr->last_vaccinator_num = i;
robust_sem_post(bring_citizen);
robust_sem_post(mutex);
```

As you can see we're also saving the caller vaccinator's pid and index. This is help us to print which vaccinator applied vaccines in citizen process.

It doesn't post the spaces(available slots). That will be done in the citizen process. Also it can be seen that there is a local variable private to specific process named *vaccinated_count*. This help us to return the vaccinated count at the end of the program. After vaccinator finishes its job it will send these numbers to parent with the help of an *unnamed pipe*.

- ***Citizens:***Each citizen will get vaccinated two by two t times. So citizens lifetime is limited to t iterations. Each citizen waits for a wake up call here. Remember the consumers were posted *bring_citizen*. Now each citizen waits for that post to break the semaphore barrier.

```
robust_sem_wait(bring_citizen);
robust_sem_wait(mutex_citizen);
ptr->vaccine_1--;
ptr->vaccine_2--;
robust_sem_post(spaces);
robust_sem_post(spaces);
robust_sem_post(mutex_citizen);
```

Here we decrement both type of vaccines and with the help of mutex_citizen. Also the we're incrementing spaces semaphores which allows vaccinator to continue without blocking.

As we mentioned earlier we've saved the last vaccinator's index and process ID. Now we're good to use it:

```
 printf("Vaccinator %d (pid=%d) is inviting citizen
     pid=%d to the clinic.\n",
ptr->last_vaccinator_num, ptr->last_vaccinator_pid,
    getpid());
printf("Citizen %d (pid=%d) is vaccinated for the %dth
    time: the clinic has %d vaccine1 and %d
    vaccine2.\n",
process_no - n - v + 1, getpid(),i, ptr->vaccine_1,
    ptr->vaccine_2 );
```

When a citizen gets all of his/her doses it decrements the current citizen count and if it's the last one informs the user about it. Also decrementing current citizen count is capsulated with another mutex. Actually gcc guarantees that those increments are atomic but I did it anyway.

```
robust_sem_wait(vaccinate);
ptr->curr_citizen_count--;
robust_sem_post(vaccinate);
printf("The citizen is leaving. Remaining citizens to
    vaccinate: %d.\n", ptr->curr_citizen_count);
if(ptr->curr_citizen_count == 0)
{
printf("All citizens have been vaccinated.\n");
}
```

6 Handling SIGINT Signal

When the SIGINT signal arrives in a process, it terminates the process as default. In our case the goal is similar but before terminating the process we have to free the resources that we have allocated. It's not limited to heap resources. We also need to close file descriptors that we have opened.

- **What to do when the signal arrives?**:

  So when the CTRL + C is pressed we should clean up resources and terminate. So when the process receives the SIGINT signal it goes to our handler:

```
void clean_explicitly()
{
    /*It's fine to use write system call inside a signal
        handler because it's Reentrant*/
    robust_write(STDOUT_FILENO, "CTRL-C CATCHED. TERMINATING
        GRACEFULLY.\n", sizeof("CTRL-C CATCHED. TERMINATING
        GRACEFULLY.\n"));
    int i;
    int child_status;
    /**
     * FREE RESOURCES FOR PARENT BEGINNING
     * */
    robust_close(s_fd);
    robust_close(fd);
    free(cur_pid);

     * TERMINATE THE CHILD PROCESSES.*/
    for(i = 0 ; i < total_child ; ++i)
    {
        kill(cur_pid[i], SIGTERM);
    }
    /**
     * REAP TERMINATED CHILDREN IN CASE OF ANY ZOMBIES*/
    for (i = 0; i < total_child; i++)
    {
        wait(&child_status);
    }
}
```

According to the homework instructions we also need to print a message just before the program terminates. We can't use functions like printf because those functions are not reentrant. Therefore in order to print the message we'll use *robust_write* function.

References

[1] Little Book of Semaphores

[2] Advanced Programming in the Unix Environment

[3] Advanced Linux Programming

[4] https://linux.die.net/man/