BUS Exercise 5
Group 23

Thilo Metzlaff
406247

Mats Frenk
393702

Emma van Emelen
406008

June 17, 2020

Contents

Introduction

Welcome back for another Excercise, with me writing every single thing in LaTeX. I am happy to announce, that i have finally found the period that we were looking for. I have also fired that unpaid intern that deleted all our data last time. It is currently 3.43 AM and i have procrastinated way too much... this seems to be my deserved punishment. I must do this, for i have an obligation to do bullshit. There may be fewer jokes... i also made code examples on github[1], even for the pseudocode stuff (i did it in C... i don't like pseudocode), if you want, you can look at that if i remember to push before dying from sleep deprivation...
Welp, i am finally done, i can now die in pieces. Fun fact: i haven't watched a single one of the lectures. man pages, google and persistence. Takes less time than one might think.

THE FORMAT

- Every file will be named similar to the sections in here, so `2.1-stack_exercise.c` is Exercise 2, section 1.
- Every Solution **WILL** be in this pdf, but not necessarily anything predefined by the exercise.
- Any explanation will be both in this PDF as well as in each file.
- This explanation will be in each PDF, in case someone who doesn't know the format tries to correct the exercises
- **WARNING:** Humor may or may not be used. If you are allergic to humor, that sounds like a personal problem.
- **WARNING:** Backing up your data is important. Although linux doesn't have the necessary shame to remove itself, unlike windows, please do back up your data. And try to keep track of your periods. . . they seem to be notoriously hard to find

[1] `https://github.com/Dragonsight91/Study-Courses-Note-collection/tree/master/BUS/exercises/5`

1 Senpai Notice me

That solution will not work. Why? Simple: One reporter signals twice. This is like your parent telling you to do a thing, after they just told you, while you scheduled it. Suddenly you have to process another signal JUST to find out it's the same thing. In this case, it will just make one guy emulate another guy. Kinda weird... but it ain't gay because one said nohomo[2].
How do we fix it?
Remove one signal. That's it. Code should work now. Go and be free now, you deserve it.

2 I like trains

2.1 queues before sephora

If you want a simple and boring answer, just look at the code below. if you want a working example in C i have an inofficial thing for you in the github repo mentioned earlier, but i digress. To solve this, we need a list, which we have. then we need two counting semaphores. one to prevent deletion, one to prevent adding. Now we crosswire them, just like the royals used to do it, and we get THIS:

```
int MAX_LEN = 10; the maximum length of the list
sem_init(enqu, MAX_LEN) // initialize the insert blocking semaphore
sem_init(dequ, 0)       // initialize the remove blocking semaphore
sem_init(sync, 1)       // a synchronizer mutex

void enqueue(element){
    sem_wait(enqu); // can we insert AT ALL? if no, wait here

    sem_wait(sync); // is someone using my toys? let's just... let them play.. we
  can always clean up later
    queue.add(element); // add our element
    sem_post(sync); // unlock write access again

    sem_post(dequ); // grant UNLIMITED POWAH
}

element dequeue() {
    sem_wait(dequ); // can we dequeue?

    sem_wait(sync); // I HAVE THE HIGH GROUND
    element out = queue.pop();
    sem_post(sync) // DON'T UNDERESTIMATE MY POWER

    sem_post(enqu); // AGATHE BAUAH
}

```

[2] SPOILER: it's still gay...

2.2 Escort(ing) services

1. Processes are executed ASYRONONCHUSLY[3] Meaning that we have a read/write asynchonicity problem here. Two guy tryna get the same girl but she just take's 'em both. Nothing wrong with that, but somehow neither of the guys know, and have the same outdated status. In our case, we don't get busy. Let's say we have two cars that arrive at the same time. both will skip the busy loop and suddenly we have two cars on the platform. Now, i don't know about you, but i haven't seen anyone stack cars recenmtly.

2. Another R/W asynchronicity problem. One car arrives but an amount of n visitors arrives. all have a chance to skip the busy loop due to outdated information and suddenly we have an amount of n visitors in 2 seats. I've seen clowns do that, they seem to be able to stack efficiently... but for anyone else, this tends to be a problem.

2.3 Need for Seat

```c
sem_init(carQueue,1);  //Mutex to show if platform is available.
sem_init(passQueue,0); //Counter Semaphore for the passenger queue
sem_init(seatAvail,0); //Counter Semaphore to check seat avalability. no seat = no car

void AnkunftWagen(){

        // wait for the platform to be available. Just how we do with shoes.
        wait(carQueue);

        fahreAufPlattform();
        oeffneTueren();

        // promise a "real bunny"
        sem_post(seatAvail);
        sem_post(seatAvail);

        // wait for victims to get in van
        sem_wait(passQueue);
        sem_wait(passQueue);

        schliesseTueren();
        verlassePlattform();

        // let ne
        sem_post(carQueue);
}

void AnkunftBesucher(){
        sem_wait(seatAvail); // wait for seats to be available. no seat = no car
        betreteWagen();
        sem_post(passQueue); // let the driver know you're inside
}

```

[3] fun adventures in human multithreading

2.4 Pascha 7th Floor

We can add another semaphore that counts VIPs. If there's someone in there, we prioritize them. If there aren't enough VIP, we fill the rest with peasants. it can be done like this:

- get value of VIP semaphore
- if >0, we add n VIP to the car. then we we fill up with $2 - n$ peasants.
- otherwise we add 2 peasants.

3 Eating out. waiiiit a minute...

Since all explanations are in the comments, i will keep it short.

1. we need to add a synchronization mutex. only one person can write at a time.
2. since shared memory is being deregistered and properly marked for destruction, there's no need to care for that.
3. i wanted to implement zombie process handling, but i didn't. Daddy does not know about the death of his children.
4. we sync every time we read or write to/from shared memory. otherwise we get a gangbang problem (many try to access one resource. can work, can go wrong).
5. i only use an 8-bit integer for the waiter's run flag, because i only need 1 or 0.
6. `masks` is a zero initialized semaphore. No one has a mask.
7. `insert_unnecessarily_long_free_space_variable_name`[4] is initialized to `MAX_CUSTOMERS` it holds the amount of free space
8. It is now `5.15AM` sleep is overrated, i'm a programmer.
9. the waiter only gives anyone a mask, if there are enough people in the queue. otherwise we randomly give masks to no one, which sometimes creates weird problems.
10. Customers are first invited to the restaurant and only then do they wait to get a mask. suboptimal, but masks can be available, as long as there are people in the queue.
11. i don't know why, but i decided to just do te most annoying thing and put an lstlisting with all the code on the next.. few pages.... it's the exact same thing as the code file.

[4] `free_space_inside`, could be shortened to fSpace

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <semaphore.h>
#include <errno.h>
#include <string.h>
#include <time.h>
#include <signal.h>

#define MAX_CUSTOMERS 8
#define MAX_QUEUE_LENGTH 10
#define MAX_PROCESSES (MAX_CUSTOMERS + MAX_QUEUE_LENGTH)
#define MASK_MIN 3
#define MASK_MAX 5

struct restaurant_s
{
    pid_t pid[MAX_PROCESSES]; /*  PIDs of child processes */
    sem_t free_space_inside;  /* this is the actual bouncer */
    sem_t shsync;             // memory sync semaphore
    sem_t masks;              // mask counter semaphore
    pid_t waiter;             // waiter PID, only needs one
    int customers_in_queue;
    int customers_in_restaurant;
};

int shouldEnd = 0; /* to terminate for-loop */
int8_t run = 1;    // waiter run variable. we only need a bit, this can never overflow, so why use more than 8 bit?

// handle SIGINT in root
void signal_handler(int signalNum)
{
    printf("Owner: Got interrupted, will shutdown the restaurant now \n");
    shouldEnd = 1;
}

// handle SIGINT in waiter
void waiter_exit(int signalNum)
{
    printf("Waiter: Going home. \n");
    run = 0;
}

int main(int argc, char **argv)
{
    int id, free_slot; /* "id" of Shared Memory Segment
                  change value stored in shared memory with *shar_mem */
    struct restaurant_s *shar_mem;
    pid_t f_pid; /* the pid after fork (customer pid) */

    /* request shared memory segment (get), attach to process (shmat), and set to 0 */
    id = shmget(IPC_PRIVATE, sizeof(struct restaurant_s), IPC_CREAT | 0644);
    shar_mem = (struct restaurant_s *)shmat(id, 0, 0);
    memset(shar_mem, 0, sizeof(struct restaurant_s));

    /* initialize pids to -1, i.e. not in restaurant */
    for (int i = 0; i < MAX_PROCESSES; i++)
    {
        shar_mem->pid[i] = -1;
    }

    /* shar_mem->free_space_inside ist eine Zaehlsemaphore die angibt, wieviel Platz im Restaurant ist
     *
     * Am Anfang kann sie also MAX_CUSTOMERS viele Leute reinlassen
     * Was waere also ein guter start Wert? Wann blockt die Semaphore?
     *
     * Achtung: shar_mem->free_space_inside ist die Semaphore um zu zaehlen, wieviele Leute im Club sind,
     * sie sichert nicht den gemeinsamen Speicherbereich (shar_mem), dazu benoetigen Sie eine 2. unbenannte
     * Semaphore, die sie noch anlegen muessen.
     */

    /* initialize random number generator */
    srand(time(NULL));

    // initialize semaphores
    sem_init(&shar_mem->masks, 1, 0); // a counter semaphore, there are no masks available, no one has a mask
    sem_init(&shar_mem->shsync, 1, 1); // a mutex, only one process is allowed to write at a time
    sem_init(&shar_mem->free_space_inside, 1, MAX_CUSTOMERS); // the free space semaphore. it's just the amount of customers that fits

    /* catch interrupts */
    signal(SIGINT, signal_handler);

    // create the waiter and save its PID in shared memory
    if ((shar_mem->waiter = fork()) == 0)
    {
        // handle the SIGINT
        signal(SIGINT, waiter_exit);
        printf("Waiter: arriving at restaurant\n"); // waiter was created
        int sleep;

        int masks; // the random amount of masks

        // run as long as no SIGINT was sent
        while (run)
        {
            masks = ((random() % 3)+3) ; // generate a random integer between 3 and 5
            sleep = ((rand() % 3001) + 6000)*1000; // sleep time in microseconds

            // we only give masks, if there are enough people in the cueue, otherwise we give masks to nonexistent people
            if (shar_mem->customers_in_queue >= masks)
            {
                // print status
                printf("Waiter: Handing out %d masks\n", masks);

                // hand out 3-5 masks
                for (size_t i = 0; i < masks; i++) // 1 iq, 3 masks; (1%3)-3 = 0
                {
                    sem_post(&shar_mem->masks);
                }
                usleep(sleep); // sleep for 3-6 seconds
            }
        }
        exit(0); // exit
    }
```

```c
    printf("Waiter PID: %d\n", shar_mem->waiter); // doesn't need sync, we are in root and it is only written once

    // root process
    while (!shouldEnd)
    {
        /* we are exclusive now */
        if (shar_mem->customers_in_queue < MAX_QUEUE_LENGTH)
        {
            /* there is space for at least one more */
            for (free_slot = 0; free_slot < MAX_PROCESSES; free_slot++)
            {
                if (shar_mem->pid[free_slot] == -1)
                {
                    break;
                }
            }

            /* enque customer in line */
            // synchronize memory access.
            sem_wait(&shar_mem->shsync);
            shar_mem->customers_in_queue++;
            sem_post(&shar_mem->shsync);

            /* create the new customer */
            // stack variables don't need to be threadsafe
            f_pid = fork();
            if (f_pid == 0)
            {
                /* this is the customer code (child) */
                struct timespec tv;

                /* childs should not catch SIG_INT */
                signal(SIGINT, SIG_DFL);
                srand(time(NULL));

                /* Check if we can enter the restaurant within 2 seconds */
                /* requires absolute time */
                clock_gettime(CLOCK_REALTIME, &tv);
                tv.tv_sec += 2;
                tv.tv_nsec = 0;

                // a value of -1 means that it had an error. as there is no reason for an error other tan a timeout
                // we can assume that we caught that.
                if (sem_timedwait(&shar_mem->free_space_inside, &tv) == -1)
                {

                    printf("%d: That takes too long, I leave\n", getpid());

                    for (int i = 0; i < MAX_PROCESSES; i++)
                    {
                        if (shar_mem->pid[i] == getpid())
                        {
                            sem_wait(&shar_mem->shsync);
                            shar_mem->pid[i] = -1;
                            sem_post(&shar_mem->shsync);
                            break;
                        }
                    }

                    sem_wait(&shar_mem->shsync);
                    shar_mem->customers_in_queue--;
                    sem_post(&shar_mem->shsync);
                }
                else
                {
                    /* we are in, so we leave the queue */
                    printf("%d: Waiing for mask \n", getpid());
                    sem_wait(&shar_mem->masks);
                    printf("%d: Going inside \n", getpid());

                    sem_wait(&shar_mem->shsync);
                    shar_mem->customers_in_queue--;
                    shar_mem->customers_in_restaurant++;
                    sem_post(&shar_mem->shsync);

                    /* stay here some time to eat - yummy!!! */
                    printf("%d: YUMMY YUM - Delicious! \n", getpid());
                    usleep(((rand() % 5000) + 3000) * 1000);
                    printf("%d: I am full - I go home now\n", getpid());

                    sem_wait(&shar_mem->shsync);
                    shar_mem->customers_in_restaurant--;
                    sem_post(&shar_mem->shsync);

                    for (int i = 0; i < MAX_PROCESSES; i++)
                    {
                        if (shar_mem->pid[i] == getpid())
                        {
                            sem_wait(&shar_mem->shsync);
                            shar_mem->pid[i] = -1;
                            sem_post(&shar_mem->shsync);

                            break;
                        }
                    }

                    // we have to free the space after we leave
                    sem_post(&shar_mem->free_space_inside);
                }
                /* exit, this causes a SIGCHLD at the parent process */
                exit(0);
            }
            else
            {
                /* Root process prints queue size and joined customer
                         * root process now knows about the child processes currently running */
                sem_wait(&shar_mem->shsync);
                shar_mem->pid[free_slot] = f_pid;
                sem_post(&shar_mem->shsync);

                printf("Owner: %d joined the queue, there are %d people in the queue and %d in the restaurant \n",
                       f_pid, shar_mem->customers_in_queue, shar_mem->customers_in_restaurant);
            }
        }

        /* delay everything a bit between 300 and 800 ms */
        usleep(((rand() % 501) + 300) * 1000);
    }

    /* ok we should end here so wait for all children to terminate */
    printf("Owner: Close the kitchen, wait for customers to leave\n");

```

```c
    // destroy the masks semaphore and stop all processes from waiting for masks.
    sem_destroy(&shar_mem->masks);

    for (int i = 0; i < MAX_PROCESSES; i++)
    {
        if (shar_mem->pid[i] != -1)
        {
            printf("%d: Going Home\n", shar_mem->pid[i]);
            waitpid(shar_mem->pid[i], NULL, 0);
        }
    }

    // wait for the waiter to close
    waitpid(shar_mem->waiter, NULL, 0);

    /* detach shared memory */
    shmdt(shar_mem);
    /* remove shared memory identifier */
    shmctl(id, IPC_RMID, 0);

    return 0;
}
```

3–eating_out.c