CSE344 – Systems Programming

Midterm Project Report

Oğuzhan Agkuş – 161044003

The main idea of this project is simulating a mess hall.

There are 3 actors involved:

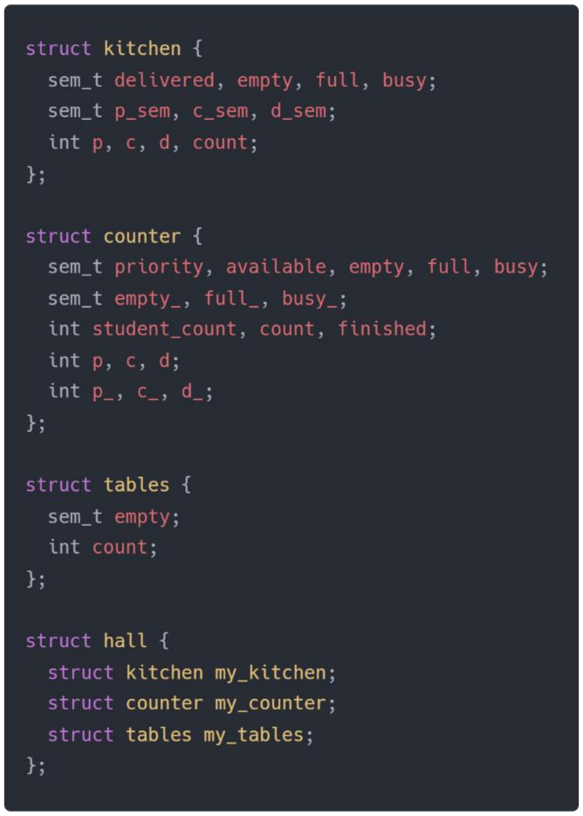
- supplier: deliver food plates to kitchen
- cooks: server food plates to counter
- students: get food plates from counter and eat them

There are 3 locations involved:

- kitchen: supplier delivers to kitchen, cooks get from kitchen
- counter: cooks serves to counter, students get from counter
- tables: after getting meal students goes to tables and eat

The locations have some limits. So, we need to synchronize their actions. This project provides this synchronization.

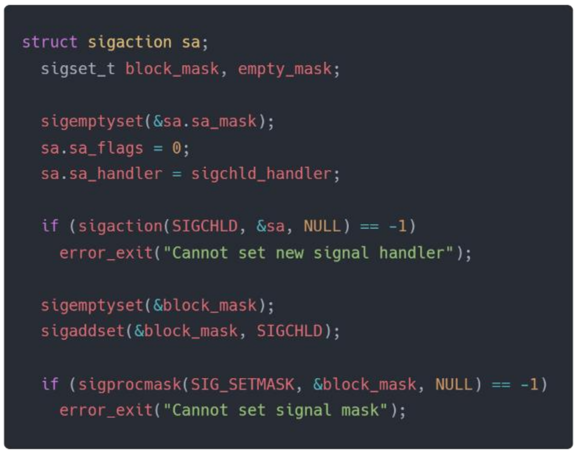
The project used, POSIX unnamed semaphores and shared memory. Shared memory simulates the locations. I defined some structures that represent my mess hall.



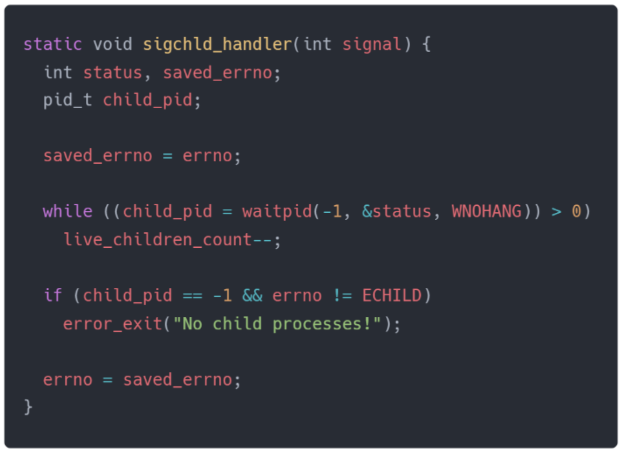
```
struct kitchen {
  sem_t delivered, empty, full, busy;
  sem_t p_sem, c_sem, d_sem;
  int p, c, d, count;
};

struct counter {
  sem_t priority, available, empty, full, busy;
  sem_t empty_, full_, busy_;
  int student_count, count, finished;
  int p, c, d;
  int p_, c_, d_;
};

struct tables {
  sem_t empty;
  int count;
};

struct hall {
  struct kitchen my_kitchen;
  struct counter my_counter;
  struct tables my_tables;
};
```

Firstly, I do input check. If any of them is invalid, it prints the error message and exit. Also, I check the input file if it has sufficient and correct data. If it has not, program will terminate.

Then I create a signal mask and a signal handler to catch SIGCHLD signals, because I do not want zombie processes. The main program will fork each actor separately. We must keep track of these child processes.

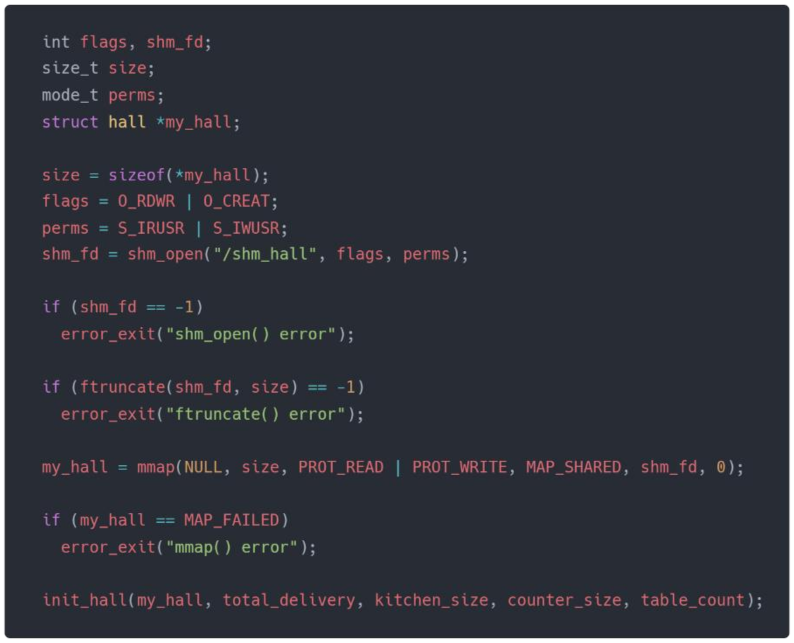


```
struct sigaction sa;
  sigset_t block_mask, empty_mask;

  sigemptyset(&sa.sa_mask);
  sa.sa_flags = 0;
  sa.sa_handler = sigchld_handler;

  if (sigaction(SIGCHLD, &sa, NULL) == -1)
    error_exit("Cannot set new signal handler");

  sigemptyset(&block_mask);
  sigaddset(&block_mask, SIGCHLD);

  if (sigprocmask(SIG_SETMASK, &block_mask, NULL) == -1)
    error_exit("Cannot set signal mask");
```



```
static void sigchld_handler(int signal) {
  int status, saved_errno;
  pid_t child_pid;

  saved_errno = errno;

  while ((child_pid = waitpid(-1, &status, WNOHANG)) > 0)
    live_children_count--;

  if (child_pid == -1 && errno != ECHILD)
    error_exit("No child processes!");

  errno = saved_errno;
}
```

After that I create a new shared memory. I use single shared memory because my struct is contain all places. They are all together. I think it makes it easy to understand.



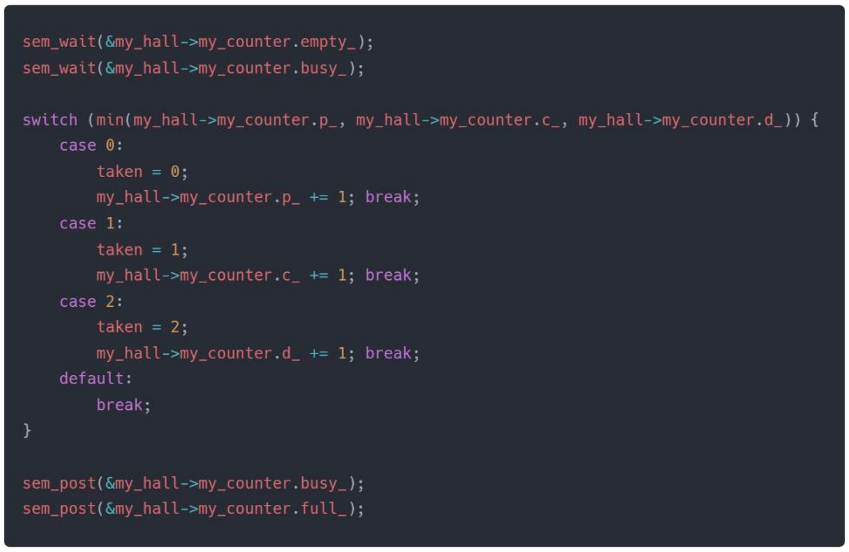
```
int flags, shm_fd;
size_t size;
mode_t perms;
struct hall *my_hall;

size = sizeof(*my_hall);
flags = O_RDWR | O_CREAT;
perms = S_IRUSR | S_IWUSR;
shm_fd = shm_open("/shm_hall", flags, perms);

if (shm_fd == -1)
  error_exit("shm_open() error");

if (ftruncate(shm_fd, size) == -1)
  error_exit("ftruncate() error");

my_hall = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);

if (my_hall == MAP_FAILED)
  error_exit("mmap() error");

init_hall(my_hall, total_delivery, kitchen_size, counter_size, table_count);
```

Then I start to fork actors. It will have a supplier, N cooks and M students. I added the bonus part. Students consist of undergraduate and graduate students. Graduate students have priority.

The synchronization problems that I encountered are like that:

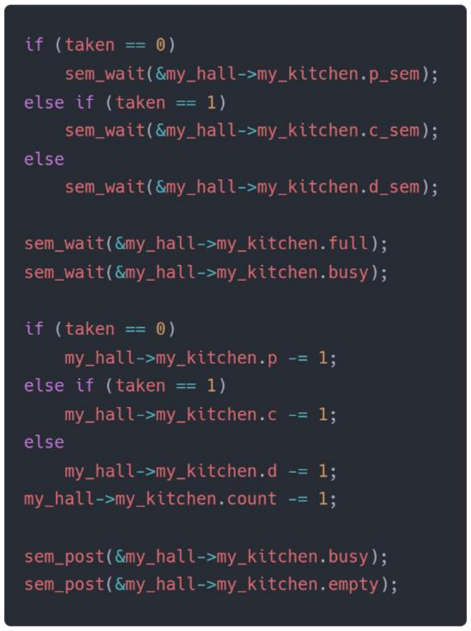
- The supplier waits until the kitchen is empty
- The cook waits until the kitchen is full
- The cook waits until the counter is empty
- The students do not get from the counter until 3 types of dishes available
- The students wait until any table is available
- Cooks must choose appropriate dishes to block deadlocks

The most important one is choosing appropriate dishes. The satisfy this I declare some extra semaphores and variables in the counter. They are not real variables of counter. They only help to cooks to choose correct dishes. It works like that: It choose the minimum counted dish. Then update the variable values. By doing this, the caller process "I will get x type dishes, you should choose another type." to others.



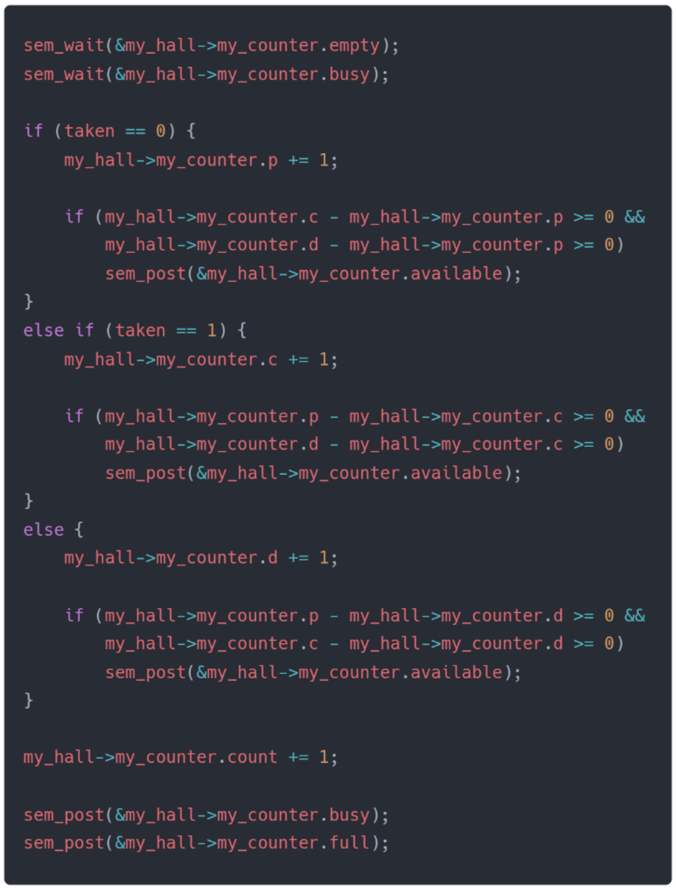
```c
sem_wait(&my_hall->my_counter.empty_);
sem_wait(&my_hall->my_counter.busy_);

switch (min(my_hall->my_counter.p_, my_hall->my_counter.c_, my_hall->my_counter.d_)) {
    case 0:
        taken = 0;
        my_hall->my_counter.p_ += 1; break;
    case 1:
        taken = 1;
        my_hall->my_counter.c_ += 1; break;
    case 2:
        taken = 2;
        my_hall->my_counter.d_ += 1; break;
    default:
        break;
}

sem_post(&my_hall->my_counter.busy_);
sem_post(&my_hall->my_counter.full_);
```

After choosing appropriate dish, the cook goes to the kitchen and wait until the dish is available. When it is available get it and leave from the kitchen.



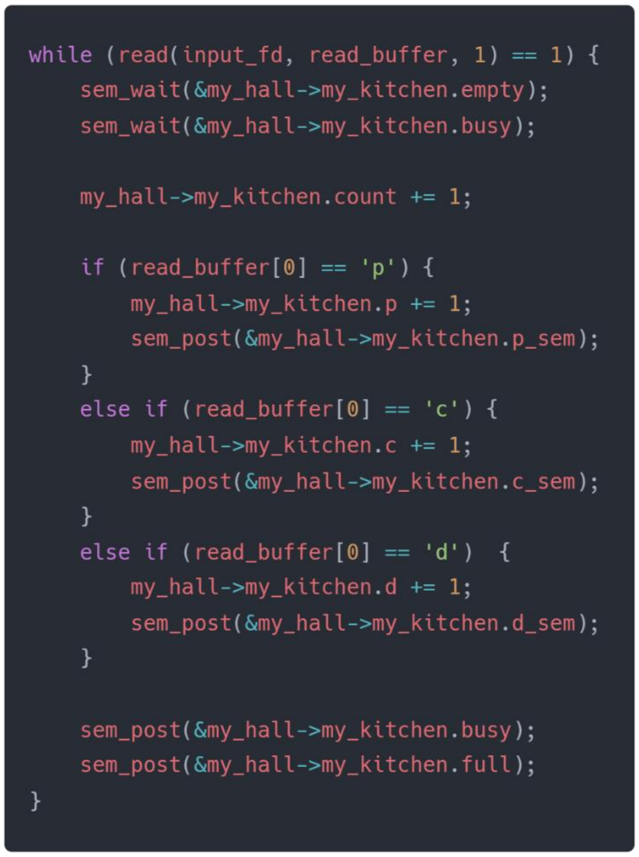
```c
if (taken == 0)
    sem_wait(&my_hall->my_kitchen.p_sem);
else if (taken == 1)
    sem_wait(&my_hall->my_kitchen.c_sem);
else
    sem_wait(&my_hall->my_kitchen.d_sem);

sem_wait(&my_hall->my_kitchen.full);
sem_wait(&my_hall->my_kitchen.busy);

if (taken == 0)
    my_hall->my_kitchen.p -= 1;
else if (taken == 1)
    my_hall->my_kitchen.c -= 1;
else
    my_hall->my_kitchen.d -= 1;
my_hall->my_kitchen.count -= 1;

sem_post(&my_hall->my_kitchen.busy);
sem_post(&my_hall->my_kitchen.empty);
```

Then it waits until the counter is empty. When it is empty, it puts the dish to dinner and check if counter is available for a student. If it is, it posts the available semaphore.



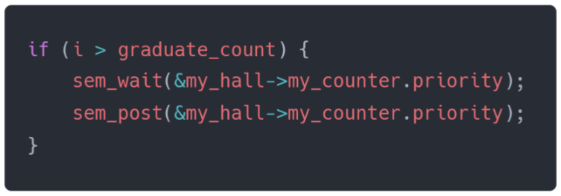
```c
sem_wait(&my_hall->my_counter.empty);
sem_wait(&my_hall->my_counter.busy);

if (taken == 0) {
    my_hall->my_counter.p += 1;

    if (my_hall->my_counter.c - my_hall->my_counter.p >= 0 &&
        my_hall->my_counter.d - my_hall->my_counter.p >= 0)
        sem_post(&my_hall->my_counter.available);
}
else if (taken == 1) {
    my_hall->my_counter.c += 1;

    if (my_hall->my_counter.p - my_hall->my_counter.c >= 0 &&
        my_hall->my_counter.d - my_hall->my_counter.c >= 0)
        sem_post(&my_hall->my_counter.available);
}
else {
    my_hall->my_counter.d += 1;

    if (my_hall->my_counter.p - my_hall->my_counter.d >= 0 &&
        my_hall->my_counter.c - my_hall->my_counter.d >= 0)
        sem_post(&my_hall->my_counter.available);
}

my_hall->my_counter.count += 1;

sem_post(&my_hall->my_counter.busy);
sem_post(&my_hall->my_counter.full);
```

By the way, the supplier delivers to kitchen like that: It read from random generated file.



```c
while (read(input_fd, read_buffer, 1) == 1) {
    sem_wait(&my_hall->my_kitchen.empty);
    sem_wait(&my_hall->my_kitchen.busy);

    my_hall->my_kitchen.count += 1;

    if (read_buffer[0] == 'p') {
        my_hall->my_kitchen.p += 1;
        sem_post(&my_hall->my_kitchen.p_sem);
    }
    else if (read_buffer[0] == 'c') {
        my_hall->my_kitchen.c += 1;
        sem_post(&my_hall->my_kitchen.c_sem);
    }
    else if (read_buffer[0] == 'd')  {
        my_hall->my_kitchen.d += 1;
        sem_post(&my_hall->my_kitchen.d_sem);
    }

    sem_post(&my_hall->my_kitchen.busy);
    sem_post(&my_hall->my_kitchen.full);
}
```

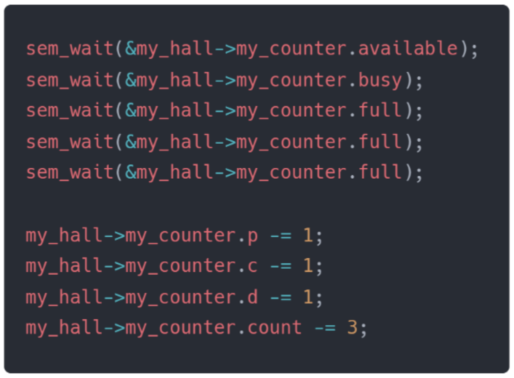
While forking the students, first G students is graduate students and the left U students are undergraduate students. When there is graduate students in the counter, any of graduate student cant get meal. So I need a lock.



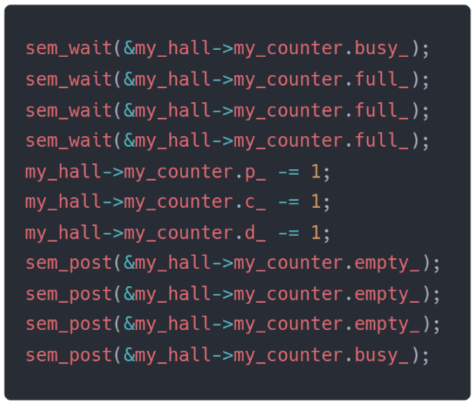
```c
if (i > graduate_count) {
    sem_wait(&my_hall->my_counter.priority);
    sem_post(&my_hall->my_counter.priority);
}
```

All undergraduate students will wait here until the semaphore is unlock by the last graduate student at counter.

The loop runs L times. Students wait for meal, get it, leave the counter and go to the tables and eat. Then repeat L times. Then it will update the helper variables and semaphores for the cooks. Then it goes to tables, wait until to available table. When a table available the student sits and eat.



```c
sem_wait(&my_hall->my_counter.available);
sem_wait(&my_hall->my_counter.busy);
sem_wait(&my_hall->my_counter.full);
sem_wait(&my_hall->my_counter.full);
sem_wait(&my_hall->my_counter.full);

my_hall->my_counter.p -= 1;
my_hall->my_counter.c -= 1;
my_hall->my_counter.d -= 1;
my_hall->my_counter.count -= 3;
```



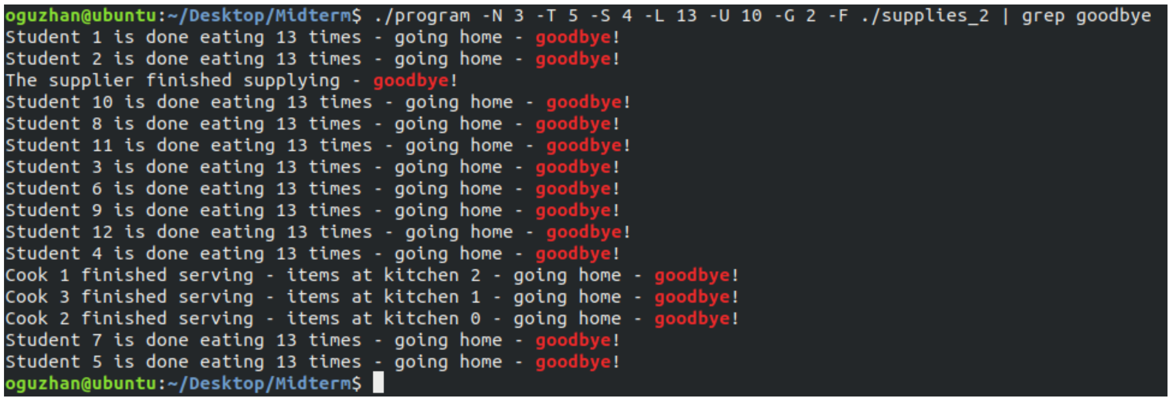
```c
sem_wait(&my_hall->my_counter.busy_);
sem_wait(&my_hall->my_counter.full_);
sem_wait(&my_hall->my_counter.full_);
sem_wait(&my_hall->my_counter.full_);
my_hall->my_counter.p_ -= 1;
my_hall->my_counter.c_ -= 1;
my_hall->my_counter.d_ -= 1;
sem_post(&my_hall->my_counter.empty_);
sem_post(&my_hall->my_counter.empty_);
sem_post(&my_hall->my_counter.empty_);
sem_post(&my_hall->my_counter.busy_);
```

It prints the status change messages.

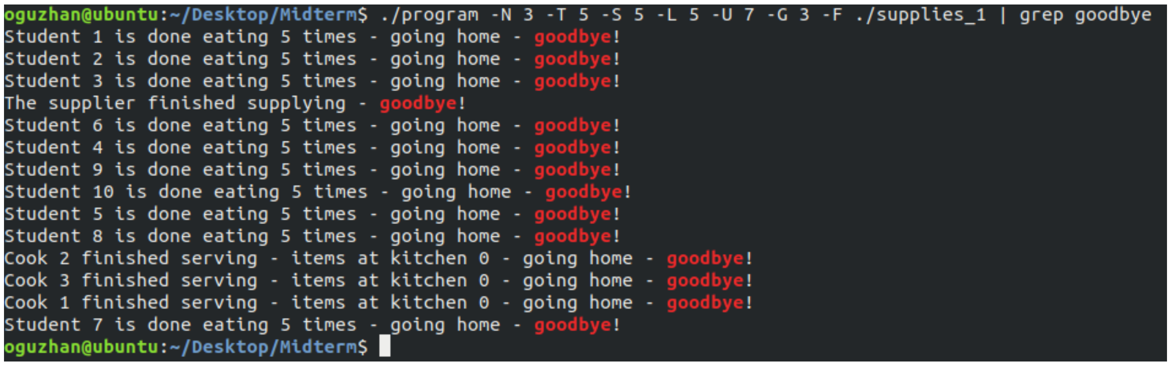
I test my code with random generated files and different numbers. I try a brute force method to catch deadlock, but any deadlock does not occur. So, it is well synchronized.

My program catches SIGCHLD signals, so there is no zombie processes detected.

There is two evidence of all processes are exited successfully.



```
oguzhan@ubuntu:~/Desktop/Midterm$ ./program -N 3 -T 5 -S 4 -L 13 -U 10 -G 2 -F ./supplies_2 | grep goodbye
Student 1 is done eating 13 times - going home - goodbye!
Student 2 is done eating 13 times - going home - goodbye!
The supplier finished supplying - goodbye!
Student 10 is done eating 13 times - going home - goodbye!
Student 8 is done eating 13 times - going home - goodbye!
Student 11 is done eating 13 times - going home - goodbye!
Student 3 is done eating 13 times - going home - goodbye!
Student 6 is done eating 13 times - going home - goodbye!
Student 9 is done eating 13 times - going home - goodbye!
Student 12 is done eating 13 times - going home - goodbye!
Student 4 is done eating 13 times - going home - goodbye!
Cook 1 finished serving - items at kitchen 2 - going home - goodbye!
Cook 3 finished serving - items at kitchen 1 - going home - goodbye!
Cook 2 finished serving - items at kitchen 0 - going home - goodbye!
Student 7 is done eating 13 times - going home - goodbye!
Student 5 is done eating 13 times - going home - goodbye!
oguzhan@ubuntu:~/Desktop/Midterm$ 
```



```
oguzhan@ubuntu:~/Desktop/Midterm$ ./program -N 3 -T 5 -S 5 -L 5 -U 7 -G 3 -F ./supplies_1 | grep goodbye
Student 1 is done eating 5 times - going home - goodbye!
Student 2 is done eating 5 times - going home - goodbye!
Student 3 is done eating 5 times - going home - goodbye!
The supplier finished supplying - goodbye!
Student 6 is done eating 5 times - going home - goodbye!
Student 4 is done eating 5 times - going home - goodbye!
Student 9 is done eating 5 times - going home - goodbye!
Student 10 is done eating 5 times - going home - goodbye!
Student 5 is done eating 5 times - going home - goodbye!
Student 8 is done eating 5 times - going home - goodbye!
Cook 2 finished serving - items at kitchen 0 - going home - goodbye!
Cook 3 finished serving - items at kitchen 0 - going home - goodbye!
Cook 1 finished serving - items at kitchen 0 - going home - goodbye!
Student 7 is done eating 5 times - going home - goodbye!
oguzhan@ubuntu:~/Desktop/Midterm$ 
```

Semaphore Organization Draft

Supplier	**Cook**	**Student**
wait(kitchen.empty)	wait(kitchen.delivered)	wait(counter.available)
wait(kitchen.busy)	wait(counter.empty_)	wait(counter.busy)
(put to kitchen)	wait(counter.busy_)	wait(counter.full) x3
post(kitchen.busy)	(choose dishes)	(get from counter)
post(kitchen.full)	post(counter.busy_)	wait(counter.busy_)
	post(counter.full_)	wait(counter.full_) x3
	wait(kitchen.(p or c or d)	(update)
	wait(kitchen.full)	post(counter.empty_) x3
	wait(kitchen.busy)	post(counter.busy_)
	(get from kitchen)	post(counter.empty) x3
	post(kitchen.busy)	post(counter.busy)
	post(counter.empty)	
	wait(counter.empty)	
	wait(counter.busy)	
	(put to counter)	
	post(counter.available)	
	post(counter.busy)	
	post(counter.full)	