# BLG 312E - Operating Systems

Homework 3

Feyza Nur Keleş
820220332

May 15, 2025

## Contents

# 1 1. Semaphore Fundamentals

## 1.1 (a)

Semaphore is counter. It show how many can go in. If value is 0, wait. If more than 0, pass.

## 1.2 (b)

- `-1` is wrong. Not work. - `0` means block all. - `1` is mutex. - >`1` means many thread can go.

## 1.3 (c)

We need queue to save who is waiting. Like ticket line.

## 1.4 (d)

Mutex is same like semaphore with 1 value. We use macros:

```
#define mutex_init(sem) sem_init(sem, 1)
#define mutex_lock(sem) sem_wait(sem)
#define mutex_unlock(sem) sem_post(sem)
#define mutex_destroy(sem) sem_destroy(sem)
```

# 2 2. Thread Scheduling and ucontext.h

## 2.1 (a)

I use `ucontext.h` to switch thread. When I do `swapcontext`, it go to another thread context.

## 2.2 (b)

Timer call signal, then call `thread_yield` or `thread_schedule`. Then it switch to next thread.

### 2.3 (c)

I make two queue: high and low. If high is not empty, pick from there. Else pick from low.

### 2.4 (d)

I check name. If name is "Player1" or "Player3", I put thread in high queue. So even if create later, they go first.

## 3 3. Synchronization

### 3.1 (a)

If two thread inside mutex and one use global var wrong, it still break. Mutex not fix logic bug.

### 3.2 (b)

Use less shared data. Also put all shared work in one mutex block. Use atomic if can.

## 4 4. File Descriptor and Communication

### 4.1 Why everything is file?

Because it is simple. Read, write, open all work same. We can use same code for socket, file, etc.

### 4.2 write vs dprintf

- `write` is low level. No format. - `dprintf` can format like `printf`.

### 4.3 send vs dprintf

- `send` is for socket. It can set flags. - `dprintf` can send string, but no flag.

### 4.4 recv vs read

- `recv` is like `read` but for socket. - `recv` can do more with flags.

### 4.5 Why no dscanf?

Because it hard to know when input finish. `scanf` need stream, socket is not same.

## 5 5. My Implementations

### 5.1 queue.c

**queue_init:**

```
void queue_init(Queue* q) {
    q->front = NULL;
    q->rear = NULL;
}
```

**queue_enqueue:**

```
void queue_enqueue(Queue* q, void* data) {
    QueueNode* node = malloc(sizeof(QueueNode));
    node->data = data;
    node->next = NULL;
    if (q->rear)
        q->rear->next = node;
    else
        q->front = node;
    q->rear = node;
}
```

**queue_dequeue:**

```
void* queue_dequeue(Queue* q) {
    if (!q->front) return NULL;
    QueueNode* temp = q->front;
    void* data = temp->data;
    q->front = temp->next;
    if (!q->front) q->rear = NULL;
    free(temp);
    return data;
}
```

**queue_is_empty:**

```
int queue_is_empty(Queue* q) {
    return q->front == NULL;
}
```

**queue_remove:**

```
void* queue_remove(Queue* q, void* data) {
    QueueNode *prev = NULL, *curr = q->front;
    while (curr) {
        if (curr->data == data) {
            if (prev) prev->next = curr->next;
            else q->front = curr->next;
            if (q->rear == curr) q->rear = prev;
            void* d = curr->data;
            free(curr);
            return d;
        }
        prev = curr;
        curr = curr->next;
    }
    return NULL;
}
```

**queue_destroy:**

```
void queue_destroy(Queue* q) {
    while (!queue_is_empty(q)) {
        queue_dequeue(q);
    }
}
```

## 5.2 semaphore.c

**sem_init:**

```
void sem_init(Semaphore* sem, int value) {
    sem->count = value;
    queue_init(&sem->waiting_queue);
}
```

**sem_wait:**

```
void sem_wait(Semaphore* sem) {
    sem->count--;
    if (sem->count < 0) {
        queue_enqueue(&sem->waiting_queue, current_thread);
        thread_block();
    }
}
```

**sem_post:**

```
void sem_post(Semaphore* sem) {
    sem->count++;
    if (sem->count <= 0) {
        Thread* t = queue_dequeue(&sem->waiting_queue);
        thread_unblock(t);
    }
}
```

**sem_destroy:**

```
void sem_destroy(Semaphore* sem) {
    queue_destroy(&sem->waiting_queue);
}
```

### 5.3 thread.c (priority)

I add 2 queues:

```
Queue high_priority_queue;
Queue low_priority_queue;
```

In thread_create:

```
if (strstr(thread->name, "Player4") || strstr(thread->name, "Player2"))
    queue_enqueue(&high_priority_queue, thread);
else
    queue_enqueue(&low_priority_queue, thread);
```

In thread_schedule:

```
if (!queue_is_empty(&high_priority_queue))
    next = queue_dequeue(&high_priority_queue);
else
    next = queue_dequeue(&low_priority_queue);
```

## 6 References

- https://man7.org/linux/man-pages/man3/ucontext.3.html
- https://www.youtube.com/watch?v=mb6U5z4G0to
- https://www.geeksforgeeks.org/semaphores-in-process-synchronization/
- My lecture notes from BLG 312E (ITU)
- https://www.man7.org/linux/man-pages/