Homework 2 Report

Elif Akgün
1801042251

1. Design Explanation

1.1 Thread Table Structure

First of all, I created thread table as struct data type to keep threads' informations. Also I created ThreadFunction struct to keep thread function's arguments.

Figure 1: Thread Table Structure

```
struct ThreadFunction{
  void *(*funcName)(void *);
  void *arg;
};

struct ThreadTable{
  int id;
  char name[15];
  unsigned long PC;
  unsigned long SP;
  reg_word registers[R_LENGTH];
  char state[15]; //RUNNING, READY, BLOCKED, TERMINATED
  ThreadFunction* function;
  void* stack;
  int stackSize;
  ucontext_t context;
};
```

Threads share same address spaces. Some information are different between threads. This entries are PC, SP, registers, and state. I keep these and other entries for context switching. When context switch occured, threads save this informations in thread table. Then, when it runs again, it keep going where it left. Each thread has 4 states: RUNNING, READY, BLOCKED, TERMINATED.

For mutexes, I created Mutex struct. It keeps only a integer lock variable.

Figure 2: Mutex Structure

```
struct Mutex{
  int lock;
};
```

1.2 Syscalls

I write 6 syscalls which are init_program, thr_create, thr_join, thr_exit, mtx_lock, and mtx_unlock. When a syscall called, relevant thread function is called. For example when thr_create syscall is called, it calls thread_create function. I explain all function below, so details are in below.

1.3 Thread Functions

I implemented some thread functions. I explained all functions and syscalls one by one.

1.3.1 thread_create

When thr_create syscall is called, this function is called in this syscall. It creates a new thread and sets its information: id, name etc. Threads' names are in this format: "Thread-" + thread.id except main thread. Main thread is created before first thread created to initialize some information. Its name is Main Thread. After setting the information and context of the thread, it runs the thread function.

1.3.2 thread_join

When thr_join syscall is called, this function is called in this syscall. It finds current thread from thread table, then it checks that whether this thread is terminated. For this purpose, it uses thread's state. If thread's state is TERMINATED, then it returns. Otherwise it calls thread_yield function for context switching. thread_join checks the thread until thread state becomes TERMINATED.

1.3.3 thread_yield

This is a helper function. It only calls contextSwitch function. To indicate threads are co-operate, I wrote this function.

1.3.4 thread_exit

When thr_exit syscall is called, this function is called in this syscall. It finds current thread from thread table, then sets its state as TERMINATED.

1.3.5 mutex_init

It only inits the mutex as 0.

1.3.6 mutex_lock

When mtx_lock syscall is called, this function is called in this syscall. If it locks, it returns; otherwise it calls thread_yield function for context switching.

1.3.7 mutex_unlock

When mtx_unlock syscall is called, this function is called in this syscall. It unlocks the lock by updating lock with 0.

1.4 Scheduler

Before I explain scheduler algorithm, let me explain which data structure I used to keep threads. I used list data structure, because Round Robin Scheduling is like a circular list. First thread runs in the list, and when time interrupt is occurred, if the thread did not terminate, it pushed to end of the list. Head of list removed, then again, first thread is in the readyQueue becomes runningThread. It goes like this until readyQueue becomes empty. I did not keep blocked threads because no threads are blocked. In merge sort problem, all threads are ready all the time because they change different parts of array. It is written in homework file "using only mutexes should be enough for inter process communication" so in producer consumer problem, if a thread could not take the mutex, it does context switch, it is not blocked.

When time interrupt is happened, or a thread yields, context switch is happend and it prints thread table informations.

2. Experiment Results

2.1 Merge Sort

In this part, I used the algorithm given in the homework file. Because of I used only kernel program as asm, there is a few time interrupt occured, sometimes it never occures. So to test the context switches, I call SPIM_timerHandler function manually in thread function as seen below. I tested it with different numbers of threads. I added the output that contains 4 threads to the report.

Figure 3: Calling the SPIM_timerHandler

```
// evaluating mid point
int mid = low + (high - low) / 2;
if (low < high) {
  merge_sort(low, mid);
  SPIM_timerHandler();
  merge_sort(mid + 1, high);
  merge(low, mid, high);
}
```

Figure 4: Output with 4 threads-1

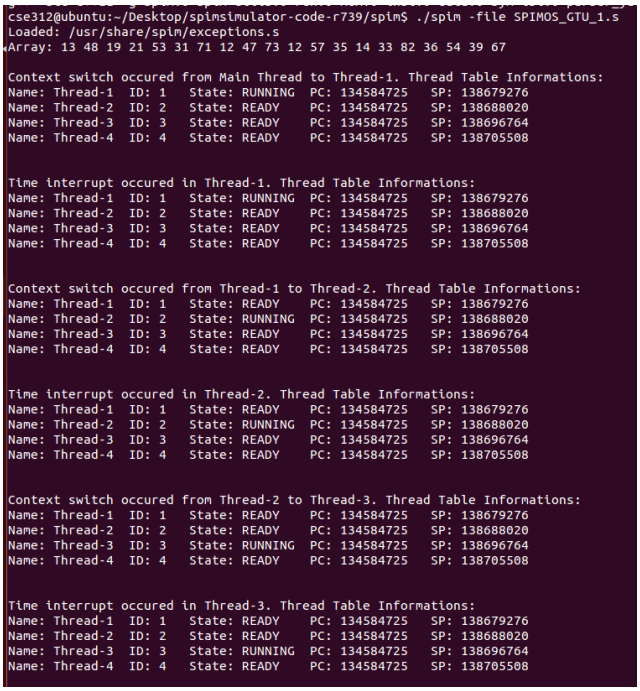


Figure 5: Output with 4 threads-2

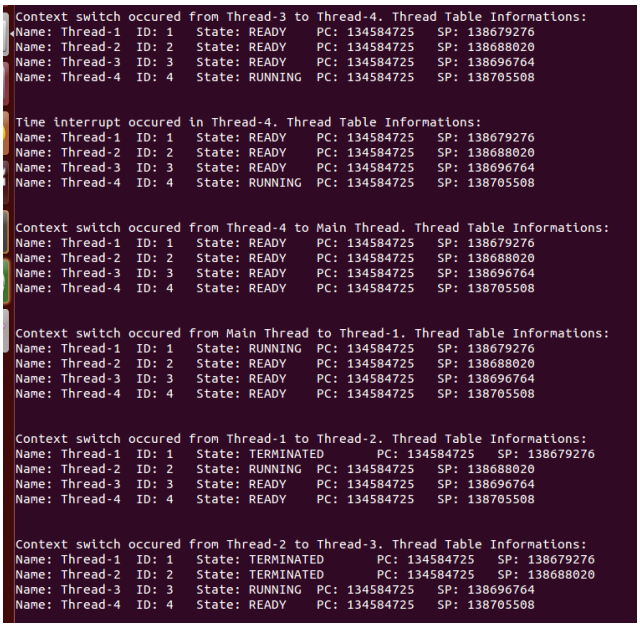
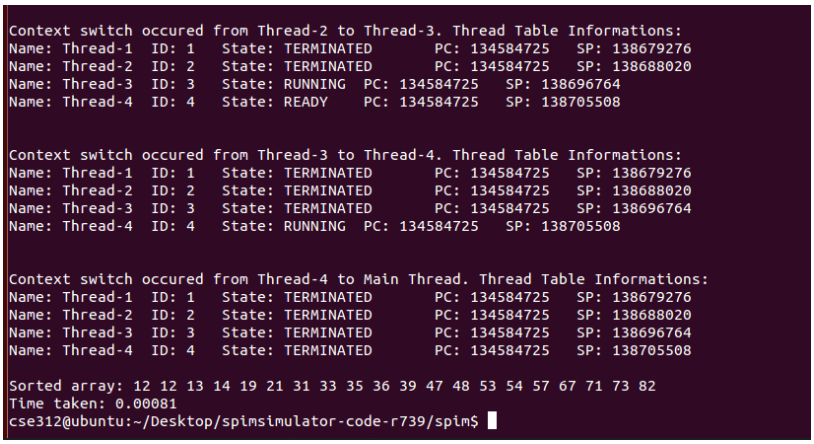


Figure 6: Output with 4 threads-3



2.2 Producer Consumer

In this part, like merge sort, I call SPIM_timerHandler function manually in thread function as seen below. I create a producer and a consumer. In producer function, producer increases buffer by one in a for loop. For loop goes to BUFFER_SIZE and it is 2 in my experiments. In consumer function, consumer decreases buffer by one in a for loop. For loop goes to BUFFER_SIZE. To understand what is happening, I wrote some messages to console. In first case, I did not use mutexes and I showed race condition in output. Second case, I used mutexes and race condition is not happened.

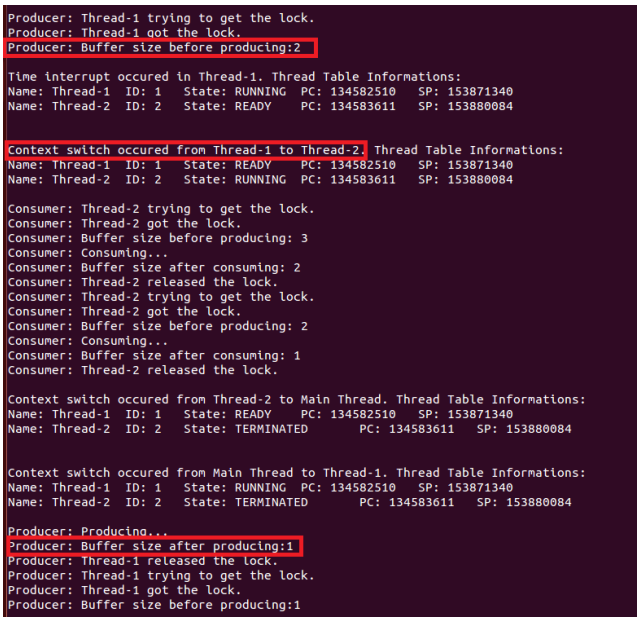
Figure 7: Calling the SPIM_timerHandler

```
void *producer(void *ptr){
  for(int i = 0; i<BUFFER_SIZE; ++i){
    cout<<"Producer: "<<runningThread->name<<" trying to get the lock."<<endl;
    mutex_lock(mutex);
    cout<<"Producer: "<<runningThread->name<<" got the lock."<<endl;
    cout<<"Producer: Buffer size before producing:"<<buffer<<endl;
    ++buffer;
    SPIM_timerHandler();
    cout<<"Producer: Producing..."<<endl;
    cout<<"Producer: Buffer size after producing:"<<buffer<<endl;
    cout<<"Producer: "<<runningThread->name<<" released the lock."<<endl;
    mutex_unlock(mutex);
  }
  thread_exit();
}
```

2.2.1 Case 1 - Without Mutexes

As seen in the below, while producer was producing, time intterupt is occured, then consumer start to run. Consumer decreases buffer and context switch occured again. Buffer size was 2 before producer produces, but producer writes "Buffer size after producing:1". This is wrong.

Figure 8: Output without mutexes



2.2.2 Case 2 - With Mutexes

In this case, as seen in the below, while producer was producing, time intterupt is occured, then consumer start to run. Consumer tried to take mutex but it could not. Then it calls thread_yield function for context switching. And then, after producer produces, it writes right buffer size.

Figure 9: Output with mutexes