EI338 Lab Report Chapter5&7

Bao Xiaoyi 517030910306

November 14, 2019

1 Project 5 Scheduling Algorithms

1.1 Environment

- VirtualBox 5.2.18
- Ubuntu 14.04 (64 bit) running on the virtual machine

1.2 Assignment

Implement several different process scheduling algorithm with either C or Java. Some supporting files have been given to read in the schedule of tasks, insert the tasks into a list, and invoke the scheduler. What is left for us to do is to realize the following schedulers. To be more specific, the following scheduling algorithms will be implemented:

- First come, first served(FCFS).
- Shortest-job-first(SJF)
- Priority scheduling
- Round-Robin(RR) scheduling
- Priority with Round-Robin

1.3 Decomposition and Analysis

Here, we try to decompose each of the original task into smaller parts, and solve the sub problems one by one.

1. Design a good structure to store the information about a specific task.

   Solution. *Apart from the given attributes(name, tid, priority, burst), we add attributes remain, arrival_time, first_served_time, finished_time. The main reason is that these attributes can help with our calculation later.*

2. Reverse and sort the original link list.(for SJF and priority scheduling)

Solution. *As we know, scheduling is usually accomplished with a queue. Given the link list, some pre-operations are needed to satisfy our need in the future. Specifically, we add two functions, which can reverse the list and sort the list separately.*

3. calculate average turnaround time, average waiting time, and average response time.

   Solution. *The information about each task is already preserved. What is left is to record the tasks we have encountered. Actually, the counter increases itself by 1 when each task is deleted from the queue.*

4. Decide the next chosen task.

   Solution. *Things are relatively easy with regard to FCFS, SJF and Priority scheduling, because we only need to sort the list in a proper way, and deal with the items one by one. However, when it comes to round robin, we need to decide whether the current task has finished(by remaining time), but we also need to find the next chosen task if it hasn't.*
   There are two special cases we need to consider. One, when the current task locates in the tail of the link list. Two, when the next task is not of the same priority as the current one. In both cases, we move the pointer to the head of the link list again.

5. Decide the time allocated to the current task, especially in Round Robin

   Solution. *We compare the remaining time for the current task with the given slice to make a decision.*

1.4 Details

In this part, some codes are shown for better illustration.

- We wrap up all information about a task before inserting it into the link list. There are three things we need to pay attention to:
  - We should allocate some space, and use *strcpy* to copy the name, rather than directly assigning the pointer to the string.
  - For synchronization, the function *__sync_fetch_and_add(&tid_count, 1)* is used to atomically increment an integer value

```
/* add a task to the link list*/
int tid_count = 0;
void add(char *name, int priority, int burst){
        Task *t = malloc(sizeof(Task));
        t -> tid = tid_count;
```

```
        __sync_fetch_and_add(&tid_count, 1);
        t -> name = malloc(sizeof(char) * (strlen(name) + 1));
        strcpy(t -> name, name);
        ...
        insert(&listForTask, t);
}
```

- In *reverse()* and *sort()* function, we treat the whole link list as two parts, the head and the tail. When initialized, the head contains only one element. In each iteration, an element is moved from the tail part to the head part. For example, here is the implementation of function *reverse*:

```
void reverse(){
        struct node *tail = listForTask;
        struct node *head = listForTask->next;
        tail -> next = NULL;
        struct node *temp;
        while (head){
                temp = head;
                head = temp -> next;
                temp -> next = tail;
                tail = temp;
        }
        listForTask = tail;
}
```

- In order to decide what is the next job to be chosen, we need to pass a pointer to the previous task as a parameter. Take priority with RR for example:

```
Task *pickNextTask(struct node *pre){
        if (pre->next ==NULL){
                return listForTask->task;
        }
        else{
                if (pre->next->task->priority ==pre->task->priority){
                        return pre->next->task;
                }
                else{
                        return listForTask->task;
                }
        }
}
```

1.5 Result

We use a toy example to test our scheduling algorithms.

```
T1, 4, 20
T2, 3, 25
T3, 3, 25
T4, 5, 15
T5, 5, 20
T6, 1, 10
T7, 3, 30
T8, 10, 25
```

Figure 1: Toy test example

Here are the results of our experiment:

```
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch5$ ./fcfs schedule.txt
Running task = [T1] [4] [20] for 20 units.
Task = [T1], tid = [0], arrival time = [0], first served time = [0], finished ti
me = [20] .

Running task = [T2] [3] [25] for 25 units.
Task = [T2], tid = [1], arrival time = [0], first served time = [20], finished t
ime = [45] .

Running task = [T3] [3] [25] for 25 units.
Task = [T3], tid = [2], arrival time = [0], first served time = [45], finished t
ime = [70] .

Running task = [T4] [5] [15] for 15 units.
Task = [T4], tid = [3], arrival time = [0], first served time = [70], finished t
ime = [85] .

Running task = [T5] [5] [20] for 20 units.
Task = [T5], tid = [4], arrival time = [0], first served time = [85], finished t
ime = [105] .

Running task = [T6] [1] [10] for 10 units.
Task = [T6], tid = [5], arrival time = [0], first served time = [105], finished
time = [115] .

Running task = [T7] [3] [30] for 30 units.
Task = [T7], tid = [6], arrival time = [0], first served time = [115], finished
time = [145] .

Running task = [T8] [10] [25] for 25 units.
Task = [T8], tid = [7], arrival time = [0], first served time = [145], finished
time = [170] .

All tasks finished.
Average turnaround time = [94.00].
Average waiting time = [73.00].
Average response time = [73.00].
```

Figure 2: Result: FCFS

```
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch5$ ./sjf schedule.txt
Running task = [T6] [1] [10] for 10 units.
Task = [T6], tid = [5], arrival time = [0], first served time = [0], finished time = [10] .

Running task = [T4] [5] [15] for 15 units.
Task = [T4], tid = [3], arrival time = [0], first served time = [10], finished time = [25] .

Running task = [T5] [5] [20] for 20 units.
Task = [T5], tid = [4], arrival time = [0], first served time = [25], finished time = [45] .

Running task = [T1] [4] [20] for 20 units.
Task = [T1], tid = [0], arrival time = [0], first served time = [45], finished time = [65] .

Running task = [T8] [10] [25] for 25 units.
Task = [T8], tid = [7], arrival time = [0], first served time = [65], finished time = [90] .

Running task = [T3] [3] [25] for 25 units.
Task = [T3], tid = [2], arrival time = [0], first served time = [90], finished time = [115] .

Running task = [T2] [3] [25] for 25 units.
Task = [T2], tid = [1], arrival time = [0], first served time = [115], finished time = [140] .

Running task = [T7] [3] [30] for 30 units.
Task = [T7], tid = [6], arrival time = [0], first served time = [140], finished time = [170] .

All tasks finished.
Average turnaround time = [82.00].
Average waiting time = [61.00].
Average response time = [61.00].
```

Figure 3: Result: SJF

```
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch5$ ./priority schedule.txt
Running task = [T8] [10] [25] for 25 units.
Task = [T8], tid = [7], arrival time = [0], first served time = [0], finished time = [25] .

Running task = [T4] [5] [15] for 15 units.
Task = [T4], tid = [3], arrival time = [0], first served time = [25], finished time = [40] .

Running task = [T5] [5] [20] for 20 units.
Task = [T5], tid = [4], arrival time = [0], first served time = [40], finished time = [60] .

Running task = [T1] [4] [20] for 20 units.
Task = [T1], tid = [0], arrival time = [0], first served time = [60], finished time = [80] .

Running task = [T2] [3] [25] for 25 units.
Task = [T2], tid = [1], arrival time = [0], first served time = [80], finished time = [105] .

Running task = [T3] [3] [25] for 25 units.
Task = [T3], tid = [2], arrival time = [0], first served time = [105], finished time = [130] .

Running task = [T7] [3] [30] for 30 units.
Task = [T7], tid = [6], arrival time = [0], first served time = [130], finished time = [160] .

Running task = [T6] [1] [10] for 10 units.
Task = [T6], tid = [5], arrival time = [0], first served time = [160], finished time = [170] .

All tasks finished.
Average turnaround time = [96.00].
Average waiting time = [75.00].
Average response time = [75.00].
```

Figure 4: Result: priority

1.6 Future work

There are several small points that we may improve in the future:

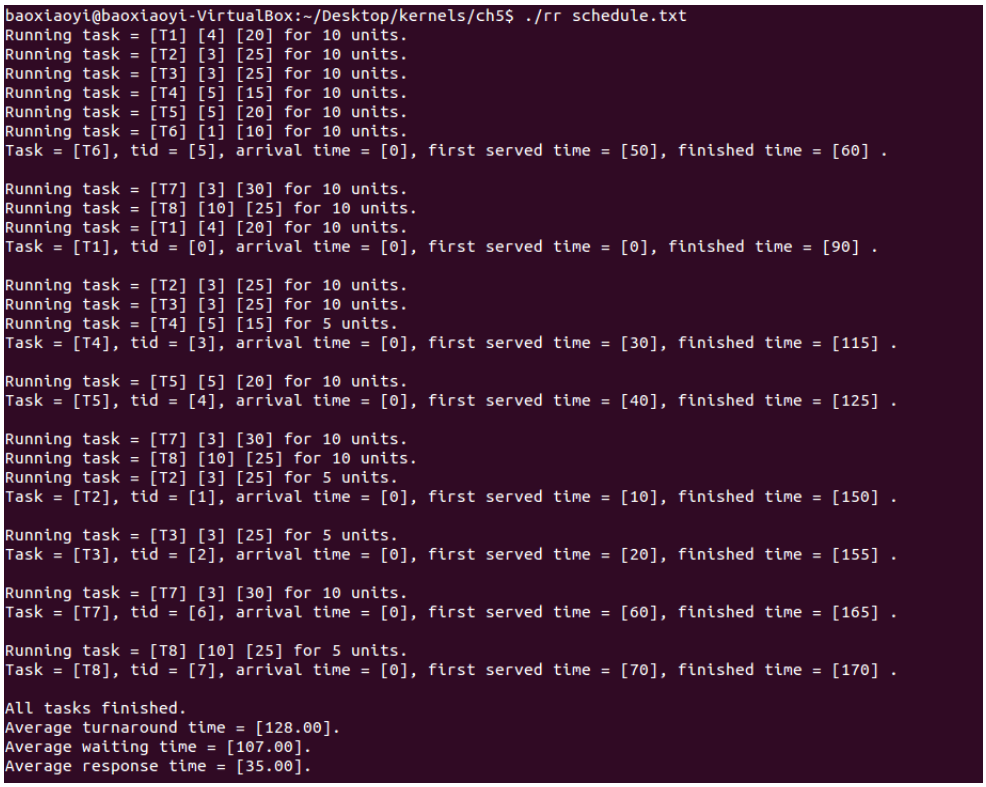

Figure 5: Result: RR

- Currently, the display function is inside *schedule*. It would be better if we can separate it and write a new function.
- We may implement the task as a class, and add functions to update it, rather than update it in *schedule*

```
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch5$ ./priority_rr schedule.txt
Running task = [T8] [10] [25] for 10 units.
Running task = [T8] [10] [25] for 10 units.
Running task = [T8] [10] [25] for 5 units.
Task = [T8], tid = [7], arrival time = [0], first served time = [0], finished time = [25] .

Running task = [T4] [5] [15] for 10 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T4] [5] [15] for 5 units.
Task = [T4], tid = [3], arrival time = [0], first served time = [25], finished time = [50] .

Running task = [T5] [5] [20] for 10 units.
Task = [T5], tid = [4], arrival time = [0], first served time = [35], finished time = [60] .

Running task = [T1] [4] [20] for 10 units.
Running task = [T1] [4] [20] for 10 units.
Task = [T1], tid = [0], arrival time = [0], first served time = [60], finished time = [80] .

Running task = [T2] [3] [25] for 10 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T2] [3] [25] for 10 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T2] [3] [25] for 5 units.
Task = [T2], tid = [1], arrival time = [0], first served time = [80], finished time = [145] .

Running task = [T3] [3] [25] for 5 units.
Task = [T3], tid = [2], arrival time = [0], first served time = [90], finished time = [150] .

Running task = [T7] [3] [30] for 10 units.
Task = [T7], tid = [6], arrival time = [0], first served time = [100], finished time = [160] .

Running task = [T6] [1] [10] for 10 units.
Task = [T6], tid = [5], arrival time = [0], first served time = [160], finished time = [170] .

All tasks finished.
Average turnaround time = [105.00].
Average waiting time = [83.00].
Average response time = [68.00].
```

Figure 6: Result: priority with RR

2 Project 7-1 Designing a Thread Pool

2.1 Environment

- VirtualBox 5.2.18
- Ubuntu 14.04 (64 bit) running on the virtual machine

2.2 Assignment

Implement a thread pool with either Pthreads and POSIX synchronization or Java. The thread pool should support creating and basic managing. To be more specific, the following functions should be provided:

- Initialization. The threads should be created as well as the mutex lock and semaphore.
- Submit a job into the pool.
- Shut down the pool.

2.3 Decomposition and Analysis

Here, we try to decompose each of the original task into smaller parts, and solve the sub problems one by one.

1. Design a good structure to store the information all the tasks.

   Solution. *We design a structure for one task, and store all the tasks in a task list. Actually, the task is implemented as a circular queue, and we also record its head, tail and the current number of items.*

2. *enqueue* operation for the queue.

   Solution. *Before modifying the queue, pthread_mutex_lock is used to check whether there is any other process executing its critical code. The function does nothing but returning 1 if the queue is already full. Otherwise, it add the task to the queue, and returns 0 to indicate success.*

3. *dequeue* operation for the queue.

   Solution. *Similar to enqueue operation, it doesn't enter its critical code unless no other process is in the critical code. An item is retrieved from its head and returned.*

4. Communication with semaphore.

   Solution. *When a task is submitted to the pool, if it is successfully added to the queue, a semaphore posts the message, and the worker starts to work as soon as he receives it.*

5. Initialize the pool before using it, and shut down the pool after all tasks have been submitted.

   Solution. *The semaphore and the mutex lock is initialized as well as all the threads. Similarly, they are all destroyed in the shutdown function.*

2.4 Details

In this part, some codes are shown for better illustration.

- A worker keeps busy waiting until a new task is added.

```
task work_to_do;
while(TRUE){
        sem_wait(&sem_count);
        work_to_do = dequeue();
        execute(work_to_do.function, work_to_do.data);
}
```

- When we shut down the pool, we need to pay special attention to the order, that is, how we shut down all the relative resourse one by one. In contrast, we don't need to do so in initialization.

```
void pool_shutdown(void)
{
        int i;
        for (i=0; i<NUMBER_OF_THREADS; i++){
                pthread_cancel(bee[i]);   //cancel without signal
                pthread_join(bee[i], NULL);
        }
        sem_destroy(&task_count);
        pthread_mutex_destroy(&queue_lock);
}
```

- In *reverse()* and *sort()* function, we treat the whole link list as two parts, the head and the tail. When initialized, the head contains only one element. In each iteration, an element is moved from the tail part to the head part. For example, here is the implementation of function *reverse*:

```
void reverse(){
        struct node *tail = listForTask;
        struct node *head = listForTask->next;
        tail -> next = NULL;
        struct node *temp;
        while (head){
                temp = head;
                head = temp -> next;
                temp -> next = tail;
                tail = temp;
        }
        listForTask = tail;
}
```

- In the main function, we create a number of jobs according to the requirement from the user.

```
printf("Please input the number of tasks: ");
    scanf("%d", &num_of_task);
// initialize the thread pool
pool_init();
struct data *data_arr = malloc(sizeof(struct data) * num_of_task);
for (i=0; i<num_of_task; i++){
```

```
T1, 4, 20
T2, 3, 25
T3, 3, 25
T4, 5, 15
T5, 5, 20
T6, 1, 10
T7, 3, 30
T8, 10, 25
```

Figure 7: Toy test example

```
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch7/7-1$ ./example
Please input the number of tasks: 5
All work has been submited.
I add two values 0 and 1 result = 1
I add two values 1 and 2 result = 3
I add two values 2 and 3 result = 5
I add two values 3 and 4 result = 7
I add two values 4 and 5 result = 9
```

Figure 8: Result: 7-1, 1

```
        data_arr[i].a = i;
        data_arr[i].b = i+1;
        while (pool_submit(&add, &data_arr[i]));
        // if error, return 1, keep submitting
}
```

2.5 Result

We use the toy example mentioned above to test our scheduling algorithms. Here are the results of our experiment:

Here are the results of our experiment:

2.6 Future work

There is one small point that we may improve in the future:

```
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch7/7-1$ ./example
Please input the number of tasks: 10
All work has been submited.
I add two values 0 and 1 result = 1
I add two values 1 and 2 result = 3
I add two values 2 and 3 result = 5
I add two values 3 and 4 result = 7
I add two values 4 and 5 result = 9
I add two values 5 and 6 result = 11
I add two values 6 and 7 result = 13
I add two values 7 and 8 result = 15
I add two values 8 and 9 result = 17
I add two values 9 and 10 result = 19
```

Figure 9: Result: 7-1, 2

- We want to implement a circular queue, so we add one more space in the queue. In this way, we actually don't need to keep track the number of items in the queue. Some basic mathematical operation is enough to decide whether the queue is full or not.

3 Project 7-4 The Producer–Consumer Problem

3.1 Environment

- VirtualBox 5.2.18
- Ubuntu 14.04 (64 bit) running on the virtual machine

3.2 Assignment

Design a programming solution to the bounded-buffer problem using producer and consumer processes. To be more specific, the following functions should be provided:

- Initialize information about the buffer and the synchronization tools before use, and destroy the memory after use.
- Producer thread.
- Consumer thread.

3.3 Decomposition and Analysis

Here, we try to decompose each of the original task into smaller parts, and solve the sub problems one by one.

1. Use some tools provided by Pthread and POSIX for synchronization

   Solution. *Here, we use two standard counting semaphores for empty and full and a mutex lock, rather than a binary semaphore, to represent mutex.*

2. Insert item into the storage.

   Solution. *You may refer to enqueue operation in the previous subsection.*

3. Remove item from the storage.

   Solution. *You may refer to dequeue operation in the previous subsection,*

4. Communication with semaphore.

   Solution. *The producer waits for empty, while the consumer waits for full. Also, the producer signals full after adding an item into the storage, while the consumer signals empty after removing an item from the storage.*

5. Initialize the pool before using it, and shut down the pool after all tasks have been submitted.

   Solution. *The semaphore and the mutex lock is initialized as well as all the threads. Similarly, they are all destroyed in the shutdown function.*

6. Wait for a random period of time before producing/consuming the next item.

   Solution. *As the return value of rand() function can be very large, we use mod operation to limit the number of time waiting. A constant, MAX_SLEEP_TIME is provided.*

3.4 Details

In this part, some codes are shown for better illustration.

- Sleep a randomized period of time.

```
temp_sleep_time = 1 + rand() % MAX_SLEEP_TIME;
sleep(temp_sleep_time);
```

- Producer waits for *empty* and signals *full*.

```
sem_wait(&empty);
insert_item(item);
printf("producer produced %d\n", item);
sem_post(&full);
```

- Consumer waits for *full* and signals *empty*.

```
sem_wait(&full);
remove_item(&item);
printf("consumer consumed %d\n", item);
sem_post(&empty);
```

3.5 Result

We get sleeping time, number of producers and number of consumers from the user.

Here are the results of our experiment:

3.6 Future work

No future work.

```
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch7/7-4$ ./pc 5 2 2
sleep time = 5, number of producers = 2, number of consumers = 2.
Initialization finished.
Sleep for 5 second(s) before exit.
producer produced 1957747793
consumer consumed 1957747793
producer produced 1649760492
consumer consumed 1649760492
producer produced 1025202362
producer produced 783368690
consumer consumed 1025202362
consumer consumed 783368690
Destroy finished.
```

Figure 10: Result: 7-4, 1

```
baoxiaoyi@baoxiaoyi-VirtualBox:~/Desktop/kernels/ch7/7-4$ ./pc 8 4 2
sleep time = 8, number of producers = 4, number of consumers = 2.
Initialization finished.
Sleep for 8 second(s) before exit.
producer produced 719885386
consumer consumed 719885386
producer produced 1189641421
consumer consumed 1189641421
producer produced 783368690
producer produced 2044897763
consumer consumed 783368690
producer produced 1540383426
consumer consumed 2044897763
consumer consumed 1540383426
producer produced 521595368
producer produced 1726956429
producer produced 861021530
producer produced 233665123
producer produced 468703135
consumer consumed 521595368
consumer consumed 1726956429
consumer consumed 861021530
producer produced 1369133069
producer produced 1059961393
Destroy finished.
```

Figure 11: Result: 7-4, 2