Operating System Lab 1

운영체제(SW) 2분반, 32170578, 김산

[Lab 1 Scheduler Algorithm Simulator]

A. Implementation

Data Structure

1. Process

- 프로세스에 대한 자료구조는 다음과 같이 작성하였습니다.

```
typedef struct {
    char p_name;    // Process name
    int ariv_t;     // Arrival time
    int rema_t;     // Remain time
    int serv_t;     // Service time
    int turn_t;     // Turnaround time
    int resp_t;     // Response time
    int time_q;     // time quatum
    int qlevel;     // Queue level(=priority)
    int ticket;     // Ticket num
} Process;
```

2. Ready queue

- 프로세스가 도착하면(arrive time == cpu time) 레디큐에 저장하도록 하였습니다.
- 레디큐의 자료구조는 원형큐로 작성하였습니다.

```
typedef struct{
    Process **array;    // Queue buffer
    int size;           // Queue size
    int count;          // Number of processes in the queue
    int front;          // Front index of queue
    int rear;           // Rear index of queue
} ReadyQueue;
```

- 레디큐의 관련 함수는 다음과 같습니다. 해당 함수들은 lab1_ready_q.c에 구현하였습니다.

```
void initReadyQueue(unsigned size);     // initialize ready queue
void del_data(Process *);               // delete data from queue
void enqueue(Process *);                // push item to ready queue
Process *dequeue();                     // pop item from ready queue
Process *getFront();                    // get first process of ready queue
```

- 로터리 스케줄링의 경우 큐 중간에 있는 프로세스가 종료되는 상황을 고려하여 중간의 값을 제거할 수 있도록 `del_data(Process *)` 함수를 작성하였습니다.

3. 프로세스 초기화 및 파일 Parsing

- 초기화와 관련된 함수들은 lab1_init.c에 정리하여 구현하였습니다.
- Workload인풋값은 외부의 input.txt파일을 통해 받아올수 있도록 구현하였습니다.
- init()을 통해 전역변수로 선언된 Process 배열인 `task[MAX_PROC_NUM]`을 초기화하고 `sortInput()`을 통해 arrive time순으로 배열을 정렬합니다.
- 각 스케줄링 함수(`fcfs(),spn(),etc...`) 호출시 초기화 함수 `init()`을 수행하여 `task[MAX_PROC_NUM]`을 초기화 합니다.

```
/* lab1_init.c : Process initialize functions */
void init();            // initialize process
void parseInput();      // parse from input.txt file
void sortInput();       // sort task by arrival time
void setTotalTime();    // set Totaltime variable by sum of tasks service time
void setTimeQ(int);     // set time quantum
```

1) FCFS

```
void fcfs() {
    init();
    char output[total_time];
    int next_idx = 0;
    int cpu_time = 0;

    Process *cur= &task[next_idx++];
    ...
```

- init() 함수를 통해 초기화한 전역변수 total_time을 통해 전체 수행시간 값을 얻고, 각 수행시간에 스케줄링 되어있는 값을 출력하기 위한 `output[total_time]`배열을 선언합니다.
- `next_idx`는 다음 `rq(readyqueue)`에 삽입할 프로세스 구조체를 가리킵니다.

```
    ...
    while (cpu_time < total_time) {
        /* enqueue arrived task to ready queue */
        while (next_idx < proc_num && task[next_idx].ariv_t == cpu_time) {
            enqueue(&task[next_idx++]);
        }
    ...
```

- 전체 cpu_time을 순회하면서 매 `cpu_time`마다 도착한 프로세스를 확인합니다
- 이때 `total_time`은 각 프로세스의 `service_time`의 합이기 때문에 만약 `cpu_time = 0`일때 도착한 프로세스가 아무것도 없다면, 또는 아무것도 스케줄링 된 프로세스가 없는경우 이 두가지 경우에 대해서는 예외가 발생합니다. 이부분까지 고려한다면 완료된 프로세스의 개수(`complete_process`)를 두고, 전체 프로세스의 개수(`proc_num`)에 대해서 `while(complete_process < proc_num)`와 같이 구현하는 방법이 더 좋을것 같다는 생각이 들었습니다.

```
        ...
        run(cur);
        /* complet porcess & schedule new process from ready queue */
        if(cur->rema_t == 0) {
            cur = dequeue();
        }
    }
    ...
```

- `run()` 함수를 통해 현재 스케줄링 되어있는 프로세스(`cur`)의 `remain time`을 감소시킵니다.
- 만약 `remain time == 0`이면 프로세스가 완료되었으므로 종료하고, 레디큐의 새로운 프로세스를 가져옵니다.
- 이때 레디큐에는 프로세스가 도착한 순서대로 삽입되어 있기 때문에 FCFS의 방식으로 스케줄 됩니다.

```
    ...
    printf("fcfs        : ");
    printOutput(output);
}
```

- 앞서 언급한 output배열을 순회하여 전체 스케줄링 결과를 출력합니다.

2) SPN

```
        ...
        sortbyServ();
        ...
```

- 전반적으로 FCFS의 구현과 동일하나 도착한 프로세스를 확인하고 레디큐에 집어넣은뒤 `service time`에 따라 레디큐를 정렬하는 함수 `sortbyServ()`를 작성하여 삽입하였습니다.
- 레디큐는 매번 새로운 스케줄링을 하기 전에 `service time` 오름차순으로 레디큐를 정렬히므로 해당 시점에서 가장 `service time`이 적은 프로세스를 스케줄 합니다.

3) RR

- 라운드로빈의 경우 매 time quantum이 만료될때 마다 context switch를 해야하므로 `setTimeQ(q)`를 통해 모든 프로세스에 time quantum을 할당해 주었습니다.

```
        ...
        /*The flag at this location ensures that the currently
            running process follows the newly arrived process. */
        if(flag == 1) {
            enqueue(tmp);
            cur = dequeue();
            flag = 0;
        }
        ...
```

- time quantum이 만료되어 context switch가 발생할때 다음 cpu time에서 새로 도착한 프로세스 뒤에 이전에 수행중이던 프로세스를 삽입해야 하기 때문에 `flag`를 통해 이를 구현하였습니다.
- time quantum이 만료되면 `flag`를 1로 설정하고 다음 cpu time에서 도착한 프로세스를 큐에 삽입한 후 `flag==1`이면 이전에 수행중이던 완료되지 않은(`cur->rema_t > 0`) 프로세스를 뒤에 삽입합니다.

```
...
if(cur->rema_t <= 0) {
    cur = dequeue();
} else if(cur->time_q <= 0) {
    tmp = cur;
    tmp->time_q = q;
    flag = 1;
}
...
```

- `run(cur)`을 수행한 후`cur->rema_t <= 0`이면 해당 프로세스는 완료되었으므로 새로운 프로세스를 큐에서 꺼내고 수행할 프로세스`cur`에 할당합니다.
- `cur->rema_t > 0` 이고 `cur->time_q <= 0`이면 프로세스가 종료되지는 않았지만 time quantum이 만료되었으므로, time quantum을 다시 원래대로 설정하고 `flag = 1`를 수행하여 앞서 언급한 작업을 다음 순회에서 진행할 수 있도록 합니다.

4) FeedBack (q=1, q=2^i)

q=1

```
...
if(flag == 1) {
    if (rq.count > 0) {
        promote(cur,1);
        enqueue(cur);
        sortbylevel();
        cur = dequeue();
    } else {
        cur->time_q = 1;
    }
    flag = 0;
}
if(flag == 2) {
    sortbylevel();
    cur = dequeue();
    flag = 0;
}
...
```

- q레벨을 올리는 것을 `promote()` 함수를 통해 구현하였습니다. `promote()`는 큐 level을 올리고 time quantum을 해당 큐의 레벨에 맞게 다시 설정합니다.
- time quantum이 만료되었을 때 레디큐에 아무것도 없다면 해당 프로세스의 큐 level을 올리지 않습니다.
- `sortbylevel()`함수를 통해 레디큐의 프로세스를 큐 level이 낮은 순서대로 정렬합니다. 이를 통해 레디큐에서 우선순위가 높은(큐 level이 낮은)프로세스가 먼저 수행됩니다.

q=2^i

```
        if(flag == 1) {
            if (rq.count > 0) {
                promote(cur,2);
                enqueue(cur);
                sortbylevel();
                cur = dequeue();
            } else {
                cur->time_q = pow_2(cur->qlevel);
            }
            flag = 0;
        }
```

- `q=1` 일 때와 달리 `q=2^i` 의 경우 q level에 따라 time quantum이 달라지도록 구현하였습니다.
- pow_2함수는 2의 i 승을 반환합니다.

5) Lottery

```
    while(cpu_time < total_time) {
        while (next_idx < proc_num && task[next_idx].ariv_t == cpu_time) {
            enqueue(&task[next_idx++]);
        }
        cur = get_winner();
        run(cur);
        output[cpu_time++] = cur->p_name;
        if(cur->rema_t <= 0) del_data(cur);
    }
```

```
/* Get winner process at lottery scheduling */
Process* get_winner() {
    int counter = 0;
    int index = 0;
    int winner = get_random();
    for (int i = 1; i <= rq.count; i++) {
        index = (rq.front + i) % rq.size;
        counter += rq.array[index]->ticket;
        if(counter > winner) {
            return rq.array[index];
        }
    }
}
```

- 현재 수행할 프로세스를 `get_winner()` 함수를 통해 레디큐에서 선정합니다.
- ticket의 개수는 `init()` 함수에서 임의로 설정하여 모든 프로세스의 ticket이 100개가 되도록 설정하였습니다.

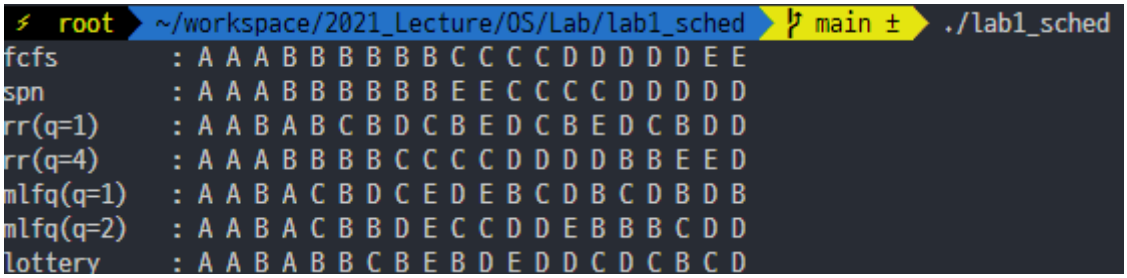
B. Results

Case 1

Work Load

Process	Arrival Time	Service Time	
A	0	3	
B	2	6	
C	4	4	
D	6	5	
E	8	2	

Result

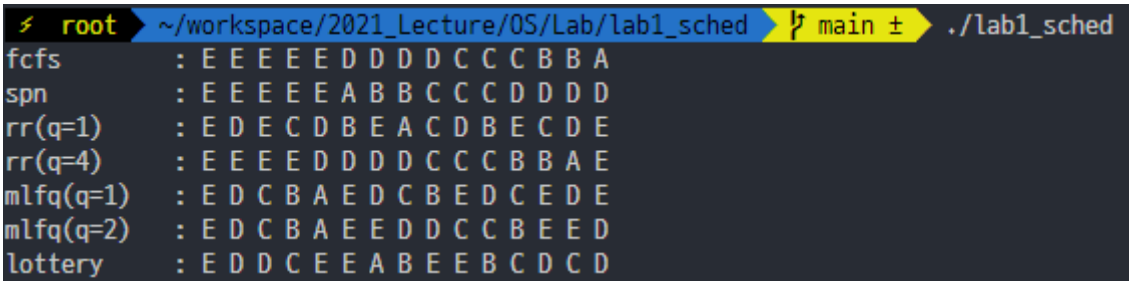


Case 2

Work Load

Process	Arrival Time	Service Time	
A	4	1	
B	3	2	
C	2	3	
D	1	4	
E	0	5	

Result

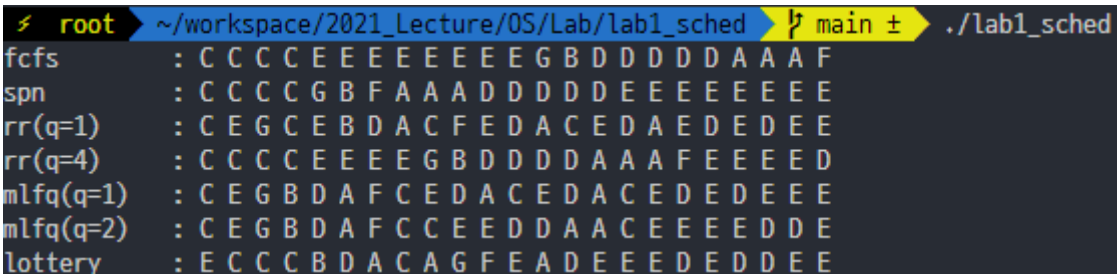


Case 3

Work Load

Process	Arrival Time	Service Time	
A	4	3	
B	3	1	
C	0	4	
D	3	5	
E	0	8	
F	5	1	
G	1	1	

Result



C. Discussion

구현하면서 많은 의문이 들었습니다. 이 부분에서는 전역변수를 써도 될지, 전역변수를 이렇게 남용해도 될지 부터 시작을 해서, 전역변수를 쓰지 않는다면 함수에 인자로 전달해야 하는데 이로인한 trade off는 어떨지, 또 반복되는 부분들을 어떻게 모듈화 하면 더 좋을지 고민이 많았던 과제였습니다.

또, 이번 과제를 하면서 삼았던 한가지 목표는 굳이 주석을 통해 설명하지 않아도 남들이 이해하기 좋은 코드를 짜고싶었는데, 이를 위해 좀더 모듈화 하고 구조적으로 짜려고 했지만 부족한 부분을 많이 느꼈습니다. 개인적으로 tovalds님의 깃허브에 올라와 있는 리눅스 코드를 보면서 그러한 부분에 대해서 많이 참고를 했지만, 아직은 좀더 익숙해지고 구현을 많이 해봐야겠다 라는 생각이 들었습니다.