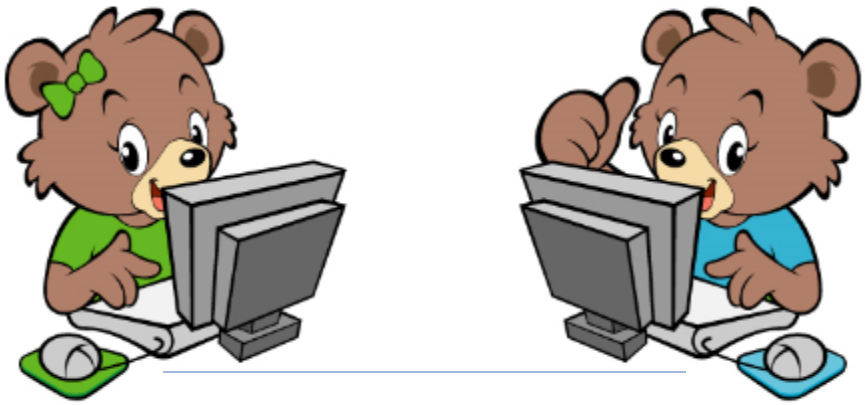
Operating System : LAB1

운영체제 (3분반) – 최 종무 교수님

– 송인호 32152332



학번 : 32152332

학과 : 소프트웨어학과

이름 : 송인호

제출일 : 20. Apr. 20

기한일 : 22. Apr. 20

단국대학교
DANKOOK UNIVERSITY

32152332 소프트 송인호

Lab1과제 : 스케쥴 시뮬레이터 구현 에 대해서 간단히 말씀드리겠습니다.

본 과제는 각각의 스케쥴링 정책들을 구현하면서 그 결과를 보이는 과제라고 할 수 있습니다. 시뮬레이터라고는 하나, 제가 설계함에 있어서 몇 가지 자료구조들이 필요하고(특히 task struct 같은), 또 이 외에도 JobQueue에서 RunQueue로 Task의 상태가 READY로 변하여 스케쥴링을 기다리는 것 또한 설계할 필요가 있다고 판단하였습니다.

간략하게 사용한 자료구조에 대해서 나열하겠습니다.

1. 사용한 자료구조들

<table>
<tr><th>필요한 개념</th><th>실제 과제의 자료구조 이름</th><th>주요 정보</th></tr>
<tr><td>Task struct</td><td>struct task_strct</td><td>

```
typedef struct Task_strct{
    char            pid;// A B C D
    int             id;
    int             state;
    int             sched_policy;
    //task_strct      *next_sched;
    //cpu_state     *my_cpu
    //void *        stack;

    //Task's cpu time
    int             total_time;
    //Time that task will spent cpu
    int             spent_time;
    //Time that has used cpu
```

(이하는 생략)

프로세스의 Pid와 id
Spent_t : 수행되어야 할 시간
Total_t : 현재까지 수행된 시간 등

현재 Queue의 주소(Q * myrq)
현재 Queue에서의 시간
Vruntime과 STRIDE, ticket(STRIDE) 등</td></tr>
<tr><td>Run Queue</td><td>struct sched_queue</td><td>

```
typedef struct Sched_queue{
   //What if FCFS?
       // normal queue, task_struct
    //What if RR?
       // context save/restore,
    //What if MLFQ?
       // Multi Queue.... -> After
       // different TIME like 2**n
    int             time_slice; //ti
    int             policy;
    task_strct      *front;
    task_strct      *rear;
    task_strct      *next_task; //<-
    #ifdef MLFQ_SCHED
        #define HIGHEST_PRIORITY 0
        #define LOWEST_PRIORITY 2
        int             my_level;
    #endif

}sched_queue;
```

타임quantum 정보
(정수로 define된) 정책Policy
(큐에 달린 task자료구조들)
Front task .. & Rear task...
MLFQ일 때, 자신의 우선순위를 나타내는 my_level</td></tr>
<tr><td>Job Queue</td><td>struct tasklist
(코드 내에서는 Joblist라고 불리웁니다.)</td><td>

```
typedef struct Tasklist{
    int arriv_T;
    //arr time for current pointing task
    task_strct * current;
    tasklist * next_item;

}tasklist;
```

시간(arriv_T)에 따른 fork되어야 할 Task 포인터(current)</td></tr>
</table>

CPU	struct cpu_st와 cpu()함수	Cpu_state (Cpu의 상태)
		`#define CPU_EMPTY 0`
		`#define CPU_RUNNING 1`
		`typedef struct Cpu_state{`
		`int cpu_state;`
		`//int cpu_no; for cacl`
		`//u64 cpu_running; 얼`
		`//Exception Divid`
		`//Exception Inter`
		`//...`
		`}cpu_state;`

Heap(STRIDE)	Struct heap_stride	`typedef struct STR_Heap{`
		`int maxsize;`
		`int add_point;`
		`task_strct ** arrOfTaskStrct;`
		`}heap_stride;`
		Maxsize : 2**n-1로 표현된 힙의 최대 크기.
		Add_point: 다음에 들어올 task의 위치(=length+1)
		arrOfTaskStrct : 배열로 표현된 Heap자료구조(task주소를 value로 가짐)

Workload 와 Scenario("A 0 3"과 같은)	Char * scenario []	"A 0 3" "B 2 6"과 같이 나중에 시간을 기준으로 joblist에 추가되어야 할 정보들.

2. 워크로드와 _Module_fork(), cpu()함수 설계

워크로드는 교수님이 말씀하신 예제처럼 구현한 것과 (test1), 수행 시간은 같지만 도착시간을 제가 임의로 수정한 테스트(test2)를 테스트용 워크로드로 설정하였습니다.

이렇게 String으로 주어진 워크로드를 기반으로, 시간 순서에 따른 Joblist를 만들었습니다. 이 리스트에는 1. 해당 task 의 도착 시간 과 2. task자료구조가 포인터로 매달려 있는 형태입니다.

각각의 스케쥴링 함수 내에서는 Job리스트에서 시간에 맞게 Task자료구조를 꺼내고 리턴해줍니다. 이러한 작업을 하는 함수를 전부 모아서 _Module_fork()라는 래퍼 함수 개념으로 설계하였습니다.

```
task_strct *
_Module_fork(tasklist * joblist, int t);
```

```
void cpu
(cpu_state * state , task_strct * task, int timestamp)
{
    state->cpu_state = CPU_RUNNING;
    task->state = TASK_RUNNING;
    if(task->spent_time == 0)
        task->res_time = timestamp;
    task->spent_time +=1 ;
    if(task->spent_time == task->total_time){
        state->cpu_state = CPU_EMPTY;
        context_save(task);
    }
    footprint[task->id][timestamp+1] = 1;
    printf("%c", task->pid);
}
```

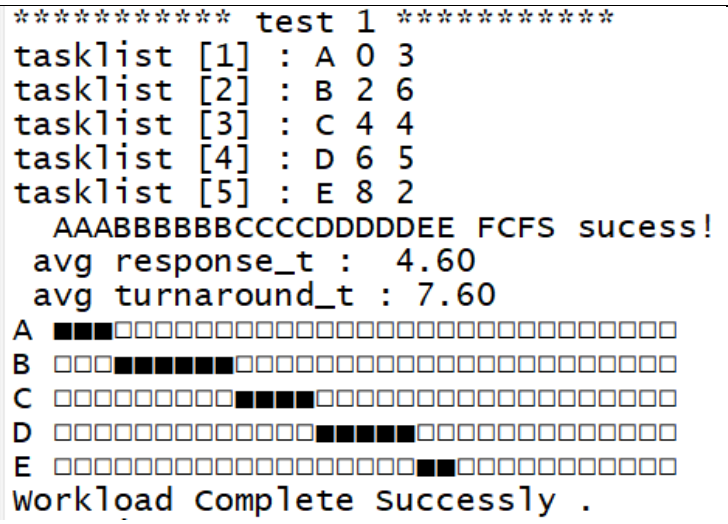
CPU자료구조와 cpu를 사용하여 task를 수행한다는 것을 표현한 cpu()함수를 간략하게 보이겠습니다.

1. cpu함수 내에서는 task자료구조의 수행시간을 증가시키고
2. 결과 화면 출력시에 필요한 부분을 출력하였습니다.
3. 또한 만약 task가 수행시간을 모두 사용했다면, cpu는 자신의 상태를 CPU_EMPTY

	라는 상태로 바꾸어 이후 스케쥴링 될 수 있도록 구현하였습니다.

- 수행결과 화면

3. FCFS의 구현



*t=0부터 수행된다고 가정.

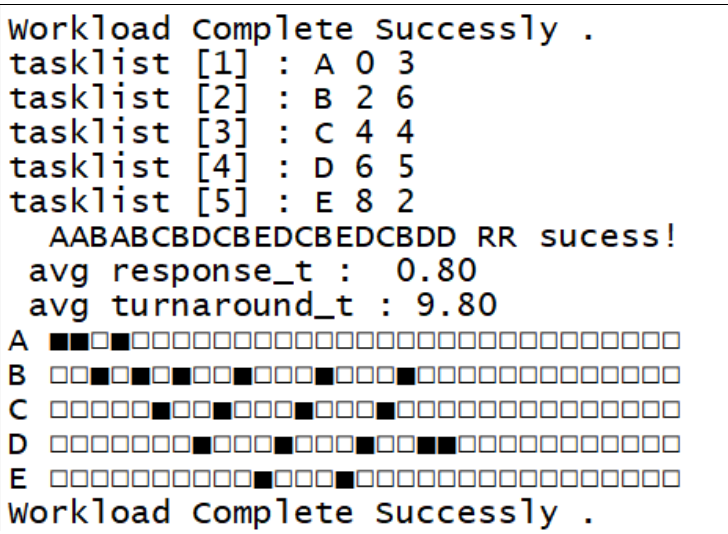
Ave_Res = (0+(3-1)+(9-4) +(13-6) +(18-8))=23 (23/5 = 4.6)

FCFS정책으로 구현한 결과 화면입니다.

FCFS정책은 선점하는 부분이 없기 때문에 그저 현재 수행하고 있는 task가 수행이 끝나고 종료가 되면, 다음 task를 스케쥴링하는 간단한 기법입니다.

Task가 수행이 끝났는지는 task자료구조 내에 현재 수행된 시간과 총 수행된 시간을 보고 판단할 수 있고, 이를 통해서 CPU를 나타내는 자료구조내의 CPU의 상태변수가 CPU_EMPTY라는 ,제가 지정한 상수로 바뀐다면 Task가 종료된 것 임을 알렸습니다.

4. RR의 구현



* t=0부터 수행된다고 가정.

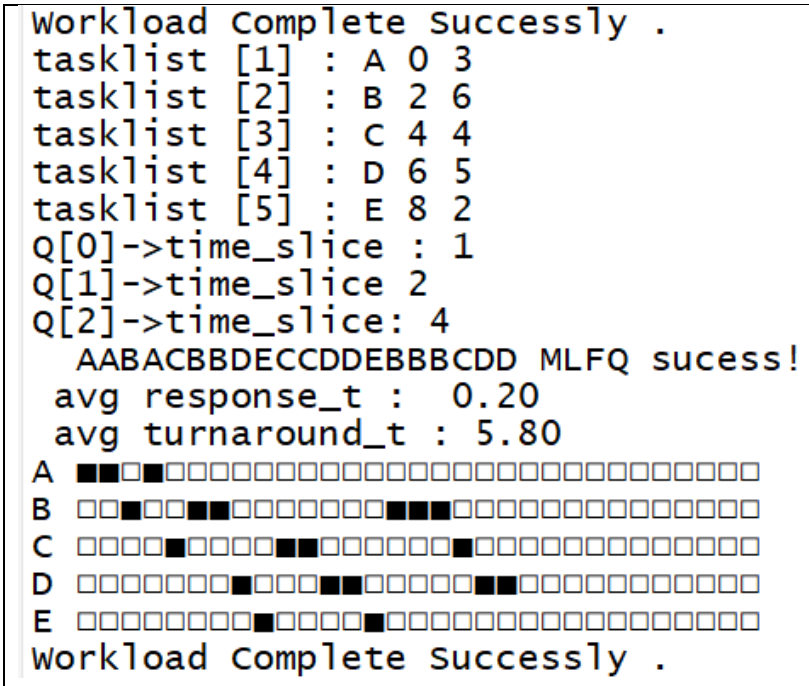
RR정책으로 구현한 결과 화면입니다.

RR정책은 주어진 timeslice (sched_queue자료구조 내에 time_slice라는 이름으로 존재하는) 만큼 수행을 하거나, 수행도중 종료가 되면 스케쥴링 하는 기법이 되겠습니다.

한 가지 구현시에 생각할 만한 조건이 있다면, 현재 대기중인 RunQ가 비었다면, 계속 수행한다는 조건을 추가하였습니다.

선점이 된다는 RR의 특성 상 context_save()라는 함수를 만들어 현재 task가 Done 상태인지, Ready 혹은 Run상태인지 저장하였습니다.

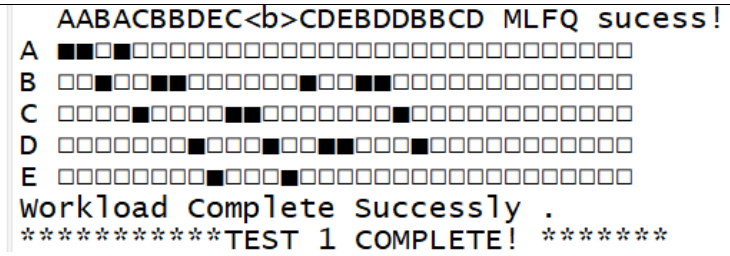
5. MLFQ 구현



*t=0부터 수행된다고 가정.

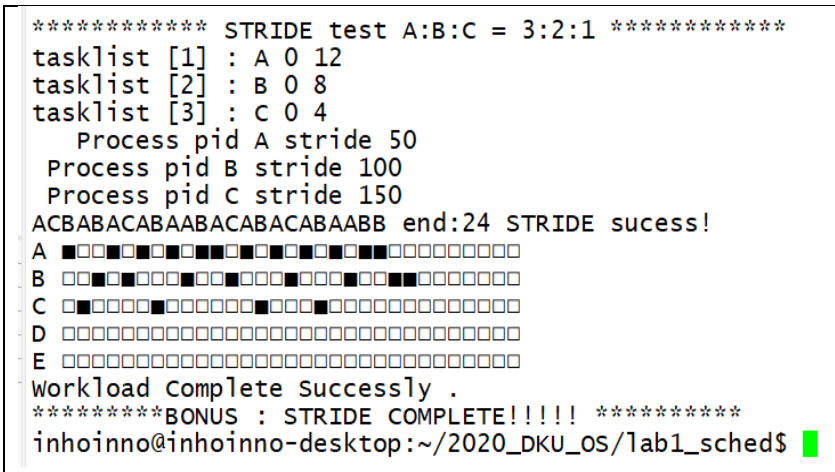
대망의 MLFQ정책으로 구현한 결과 화면입니다.

MLFQ정책은 앞서 설명드린 RR을 포함하면서, 현재 Task의 수행 시간에 따라서 TimeQuantum이 큰 Queue로 우선순위를 낮추는 기법입니다.
수행하는 도중에 새로운 task가 생성되어 현재 수행되고 있는 task보다 우선순위가 높다면 새로운 task를 수행시키는 것이 생각할 만한 부분이었습니다.
생각보다 오류가 많아서 코딩/테스팅 시에 꽤 까다로운 부분이 몇 가지 있었습니다.



이것은 t =10일 때 boosting을 한 결과 화면입니다.
비교해 보신다면 기존의 수행은 C DD E BBB C DD인데 부스팅되어서 C D E B DD BB C D로 수행되는 결과를 볼 수 있었습니다.

6. Bonus)STRIDE 스케쥴링



* 현재 3:2:1의 비율로 수행을 시켜 보았고,
디버거를 돌리면서 확인 한 결과, 같은 vruntime일 때 알파벳순으로 수행 되지 않는 것을 제외하고는 정상적으로 Heap이 구현됨을 알 수 있었습니다. (A : B : C = 12 : 8 : 4 = 3 : 2 : 1)
각 프로세스의 시간은 종료 시간을 맞추어 비율을 보고자 한 것뿐입니다.

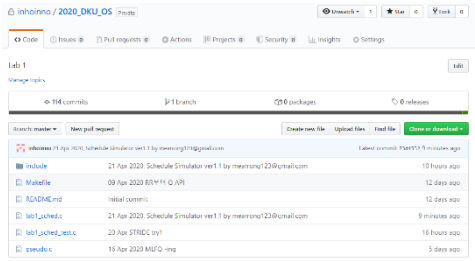
보너스 문제인 STRIDE정책으로 구현한 결과 화면입니다.
STRIDE함수를 실행시키는 _env_STRIDE함수는
1. 새로운 프로세스를 joblist와 _modue_fork로 생성시키고
2. vruntime을 0으로 초기화 하고 minHeap에 insert 합니다
3. 이후 힙의 root를 스케쥴링 하고
4. cpu에 넣어 수행시키고
5. 해당 STRIDE값 만큼 증가(vruntime += stride)시켜 다시 힙에 추가 하는 방법으로 구현하였습니다.

32152332 소프트 송인호

소감.

"이것만큼은 혼자 힘으로 해보자!" 라는 마음이 드는 과제였습니다. 약간의 실수도 많았지만, 덕분에 디버거를 능숙하게 사용하게 되었고, 여러모로 설계 능력, 코딩 능력도 많이 늘어난 것 같은 기분이 들어 뿌듯했던 과제였습니다.

이번 과제:Lab1을 통해서 한 가지 깨달은 점은, 한 번만에 원하는 결과가 만약 나온다면 그것은 구현을 잘못했을 확률이 굉장히 높다는 점입니다. ㅎㅎ



[1]깃허브를 사용해서 조금씩 조금씩 코드를 추가하고, 테스트 하였습니다.

또한 리눅스 커널 소스안에 있는 sched.h를 참고하면서, 설계에 도움을 얻었고, 이름 또한 많은 부분에서 조금씩 인용하였습니다. 간단히 제가 구현한 과제를 요약하면 (MLFQ기준으로 설명 드리자면)

루프를 돌면서(time)

1. 해당 시간에 해당하는 task를 생성하여(Joblist로부터 꺼내), 몇 가지 초기 설정을 한 뒤 런 큐에 집어 넣고, (이전까지 수행하던 프로세스가 있다면 우선순위를 비교하여 조치합니다)
2. TimeQuantum 경과 유무, cpu의 상태, 우선순위인 큐의 상태 등으로 스케쥴링을 해야하는지 판단하고 스케쥴링 합니다(curr_task)
3. 스케쥴링 된 curr_task를 cpu에 집어 넣고(cpu()함수 사용)수행을 시킵니다. 종료가 된다면 CPU의 상태를 나타내는 자료구조에 CPU_EMPTY라는 상태가 있습니다
4. 마찬가지로 TimeQuantum, cpu상태에 따라서 지금까지 수행한 정보를 저장하고 (ContextSave()) 아직 수행할 것이 남았다면, 혹은 현재 큐에서의 시간을 다 썼다면 등으로 조건을 판단후 해당 큐에 Enqueue 해줍니다

1과 2 사이에 (모든 큐가 비었고, CPU의 상태가 Empty고, joblist또한 비었다)면 종료하는 종료함수(EndWorkload())가 위치합니다.

자세한 구현 사항은 운영체제 과목 조교 선생님인 최 건희 선배에게 제출하도록 하겠습니다.

감사합니다!

송인호 올림.

---

[1] https://github.com/inhoinno