Project02 wiki

Design&Implement

Multilevel queue 와 MlFQ 구현에 앞서, make 를 할 때 두 스케줄러를 구분해주기 위하여 아래와 같이 make file을 수정하였습니다.

```
SCHED_POLICY = DEFAULT
MLFQ_K=0

CC = $(TOOLPREFIX)gcc
AS = $(TOOLPREFIX)gas
LD = $(TOOLPREFIX)ld
OBJCOPY = $(TOOLPREFIX)objcopy
OBJDUMP = $(TOOLPREFIX)objdump
CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -fno-omit-frame-pointer
CFLAGS += $(shell $(CC) -fno-stack-protector -E -x c /dev/null >/dev/null 2>&1 && echo -fno-stack-protector)
CFLAGS += -D $(SCHED_POLICY) -D MLFQ_K=$(MLFQ_K)
```

Make 시 SHCED_POLICY 에 들어가는 값을 통하여 어떤 스케줄러를 사용할지 구현했습니다.

또한 만약 MLFQ 스케줄러를 사용시에는 MLFQ_K의 값을 받아 큐의 개수를 조정하게 만들었습니다.

MULTILEVEL QUEUE SCHDULER:

```
  for (p2 = ptable.proc; p2 < &ptable.proc[NPROC]; p2++){
          if( (p2->pid!=2) && (p2->pid)%2==0&&p2->state==RUNNABLE ){
              even_exist=1;

          }

  }
 //cprintf("as: %d/",even_exist);
```

우선 위와 같이 우선 현재 ptable을 전부 탐색하여 지금 Runnable하고 짝수의 pid를 가지는

process가 들어와있는지 확인합니다. 만약 그렇다면 even_exist 변수에 1을 할당하여 존재한다는 표시 후 다음 코드로 넘어갑니다.

```
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)//hear, find process with  minimum pid
{
    //cause it mean faster in, which means  come in first


  if (p->state != RUNNABLE)continue;
  if( (p->pid)==1||p->pid==2){//exception. shell and init is essential
      tmp_process=p;

  }
    //}
  else if( even_exist==1){//even process in so even must in
        if( (p->pid)%2==0){
            tmp_process=p;
        }
        else {

                push(p);
                continue;
        }
    }
    else if (even_exist==0){
        if(p->pid%2==1){
                push(p);
        }
        continue;
    }

    c->proc =tmp_process;//put process in cpu
    switchuvm(tmp_process);//context switch
    tmp_process->state = RUNNING;//make state running
    swtch(&(c->scheduler), tmp_process->context);
    switchkvm();
    //if(cchhkk==1) cprintf("edd\n");

    c->proc=0;
    //cprintf("ed\n");


}
```

위 코드는 스케줄러가 cpu에 할당해줄 process를 찾고, cpu에 할당해주는 작업을 수행합니다.

또한 위 코드는 2가지 케이스로 동작하게 됩니다.

1) 스케줄러에 짝수 pid를 가지는 runnable한 process 가 존재한다.

   이 경우, 앞서 찾은 even_exist 값이 1이므로 해당 else if 문이 실행됩니다.

   for문은 짝수, 홀수 관계 없이 모든 process들을 탐색하기에, 지금 p에 들어와있는 프로세스의 pid 가 짝수인지 확인 후, 짝수라면 tmp_process에 p를 대입 해준 뒤 Cpu에 바로 할당해주게 됩니다. 이때 할당된 process 는 time interupt를 통하여 이미 들어온 process가 완료되지 않더라도 주기적으로 cpu에 새로운 프로세스를 할당해주게 됩

니다. 만약 홀수라면, push 통해 제가 만든 queue자료 구조에 중복없이 누적해주고 할당 없이 continue 하게 됩니다.

2) 스케줄러에 짝수의 pid를 가지는 runnable한 process가 존재하지 않는다.

이때는 홀수 pid를 가지는 process들의 스케쥴을 관리하게됩니다.

이 경우, even_exist값은 그대로 0이기에, else if 문으로 들어갑니다. Else if 문에서는

현재 p가 홀수의 pid인지 확인하고, 홀수라면 process를 큐에 중복없이

저장하며 for문을 돌게됩니다. 그리고 ptable을 끝까지 돈 뒤 빠져 나오게됩니다.

이때, 큐는 for문을 돌며 process 가 들어온 순서대로 queue에 누적해주게 됩니다.

```
//add
struct proc* arr[81];
int front=0,rear=0;




int in_qu(struct proc* a){
    for(int i=front;i<rear;i++){
        if(arr[i]->pid==a->pid)
            return 1;
    }
    return 0;
}

void push(struct proc* a){
    if(in_qu(a)==0){//not in qu
        arr[(rear)%80]=a;
        rear=(rear+1)%80;
    }

}

void pop(){
    if(front==rear)
        return;
    else{
    arr[front]=0;
    front=(front+1)%80;
    }

}


struct proc*  get(){
    if(front==rear)
        return 0;
    return arr[front];

}

int emp(){
    if(front==rear)
        return 1;
    else
        return 0;

}
```

이 코드는 제가 추가한 큐 자료구조 입니다.

In_qu는 이미 큐에 중복된 process 가 있는지 pid를 통해 체크하고, pop함수는 일반적인 큐와 동일하게 돌아가지만,

Push 함수는 In_qu함수를 통해 중복된 값이 있다면 push 하지 않아 중복된 프로세스가 큐에 들어가있는 것을 방지합니다.

아래는 queue를 적용하여 스케쥴하는 코드입니다.

```
if(even_exist==0&&emp()==0&&get()!=0){
     struct proc * nxt_odd=0;
    nxt_odd=get();

    if(nxt_odd->state!=RUNNABLE){
        pop();
    }
    else{
     c->proc = nxt_odd;//put process in cpu
     switchuvm(nxt_odd);//context switch
     nxt_odd->state = RUNNING;//make state running
     swtch(&(c->scheduler), nxt_odd->context);
     switchkvm();
    }
```

큐가 비어있다면 스킵하고, 만약 큐에 process가 있지만 runnable 하지 않을 시, Pop을 해주고 다음으로 넘어갑니다. 그리고 만약 큐의 front에 runnable한 process가 있다면, cpu에 할당해줍니다. Queue의 FIFO 특성을 이용하여 FCFS를 구현하였습니다.

MLFQ SCHDULER

멀티레벨 피드백 큐 스케줄러는 MLFQ_K로 입력받은 수 만큼의 큐를 가지고 운용되고, 각 큐마다 4*i+2개의 time quantum 을 가집니다. 그리고 이 time quantum을 모두 사용시 상위 레벨큐로 올려주게됩니다. 그리고 만약 최상위큐(K-1)에서 time quantum을 모두 소모한 경우, priority boost 가 일어날 때 까지 더 이상 할당되지 않습니다.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
//project2 added
  uint ticks;
  int level;//queue level
  int priority;
  int last_out;

};
```

우선 proc.h의 구조체 proc에 몇가지 변수들을 추가하였습니다. 프로세스의

큐 레벨을 알려주는 level, priority, '프로세스' 의 tick을 체크해줄 변수 ticks, 그리고 마지막 큐에서 타임아웃이 일어날

시 그것을 알려주는 last_out 변수를 새롭게 만들었습니다.

```
#ifdef MLFQ_SCHED
    acquire(&ptable.lock);
    // Multilevel feedback queue scheduling
    //See which processes have expired their time slices
struct proc *tmp=0;//tmp contain process with highest level(low num) queue with highest priority
     int chk=0,mn_qu=99,mx_prio=-1; //minimum num queue == highest priority so find min to get highest prio q
     for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        // if(p->last_out==1)  >> trouble!
        //   p->state=SLEEPING;

         if( (p->state) != RUNNABLE)continue;
         //cprintf("l0: %d/n",p->last_out);

         //if(p->last_out){

        //  if( p->last_out!=0) continue;
        //}

        if((p->pid)==1){
            tmp=p;

        }

        else if(!tmp){//initialize
            tmp=p;
         }

        else if (p->last_out==1)continue;// if it goes up, no process problem occur

         else if( (p->level) ==tmp->level){// find process with same level queue
             if(p->priority>tmp->priority){// find process with highest prio same high level queue.
                 tmp=p;
             }

          }
          else if( (p->level) <tmp->level){// find process with high level queue
                  tmp=p;
            }
        }

        if (tmp&&tmp->state==RUNNABLE){//double check
       // cprintf("lq: %d/",tmp->last_out);
        c->proc = tmp;
        switchuvm(tmp);
        tmp->state = RUNNING;
        swtch(&(c->scheduler), tmp->context);

        switchkvm();
        //cprintf("5\n");
        c->proc = 0;
        }

//must add : as time quantum run out, lower qu lev, if t_q =100 prio boost
    release(&ptable.lock);

#endif
```

위 코드는proc.c에서 구현된 코드입니다. 우선 for문을 돌며 runnable하지 않은 프로세스는 패스하고, 만약 아직 아무런 프로세스가 할당되어있지 않다면 tmp에 할당하여 initialize 해줍니다.

그리고 제가 proc 구조체에 새롭게 만든 변수 last_out을 확인하게 되는데, 만약 1의 값이 할당되어있다면 마지막 큐에서 time quantum을 모두 사용한 proces이기에 제외합니다.

그 이후, 만약 탐색중에 더 낮은 레벨의 큐에 할당된 process가 존재한다면 tmp변수에 갱신해주고, 같은 큐레벨이라면 더 높은 priority를 가지는 process를 tmp에 갱신에주게 됩니다.

for문을 끝까지돌고나면, 할당하지 말아야할 조건들을 만족하고, 가장 낮은 큐에있고 가장 높은

우선순위를 가진 process가 tmp에 들어가있게 되고, 이후 cpu에 해당 process를 할당해주게 됩니다.

```
#ifdef MLFQ_SCHED

      //ticks= time  interupt come every 1 cycle (round robin) ticks is tick for queue
      //proc-> ticks is tick for 'process'
        if(  (ticks%100) == 0){//priority boost
            //cprintf("boost!\n");
            //struct proc *p;
            //acquire(&ptable.lock);
            //for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                if(myproc()->pid > 0){//double check

                        myproc()->level=0;
                        myproc()->ticks = 0;
                        myproc()->last_out = 0;// if process slepp becuase of runout from last queue, reset

                    //}
                }
            //}
            //release(&ptable.lock);

        }
        myproc()->ticks++;

        int k=MLFQ_K; // get k
        int qu_num= myproc()->level ;


        if ( (myproc()->last_out)==0   && myproc()->ticks >= ( 4* qu_num +2 )  ) {//exhaust all time quantum
            if(myproc()->level<k-1){//not in last queue
                myproc()->level = qu_num+1;//down level when tick run out
                //cprintf("qu_%d+1\n",qu_num);
                myproc()->ticks=0;
            }
            else{
                myproc()->last_out=1;
                //myproc()->state=SLEEPING;// if in last queue and run out all time_quantum checkpoint
                //cprintf("last_qu sleep\n"); d  //sleeping >> yiled make runnable
            }
            //myproc()->ticks=0;
        }
        yield();

#else
```

위 코드는 trap.c 에 구현된 코드입니다. 이 코드는 큐레벨에 따른 time quantum, 그리고

Priority boost를 구현한 코드입니다. 매 타임 인터럽트가 발생할때마다, 현재 할당되어 있는 process의 ticks이 1씩 증가하게 되어 cpu에 현재 얼마나 있었는지를 확인할 수 있습니다.

또한 'procss' 의 tick 이 아닌, 전체의 tick을 나타내는 ticks 또한 존재하는데요, 이경우는

Myproc()->ticks 와 같이 proc 구조체에 생성된 것이 아닙니다. 이 전체의 ticks가 100이 될때마다,

모든 큐들을 최하위(L0)큐에 올려주는 작업을 수행하게 되는데요, 이때 만약 마지막 큐에서 timequantum 을 초과하여 last_out의 값이 1이된 process 가 있다면 다시 0으로 초기화하여

Cpu에 할당될 수 있게합니다. 그리고 만약 process가 level에 맞는 time quantum을 전부소모하게 된다면

1) 만약 최상위 큐에 위치하지 않는다면, 큐 레벨을 1 증가시킨 뒤 시간을 초기화해줍니다.

2) 만약 마지막 큐에서 time quantum을 전부 쓴 경우 last_out 변수에 1을 할당하여 표시해주어 cpu에 할당되지 않도록 합니다.

Set priority, get level, yield

```
//set_priority syscall ass1_195423
int setpriority(int in_pid, int new_priority)
{

    //prio range error!
  if(new_priority<0||10<new_priority)
      return -2;


  //scan process table and find process with
  struct proc *p;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == in_pid)// find process
    {

        //cprintf("%d",p->ppid);
        if(myproc()&&( (p->parent)->pid )==myproc()->pid){//if calle is caller's children, it work
            //acquire(&ptable.lock);
        //if(myproc()){
            p->priority=new_priority;
            //release(&ptable.lock);
            return 0;

        }

    }
  }


  return -1;// error




}

int getlev(void){
    int gt=myproc()->level;
    return gt;
}
```

위 코드는 proc.c에 구현되었습니다. Setpriority는 부모의 priority를 할당해주되, '직접' 만든 자식 프로세스의 priority만 설정할 수 있게 if문을 통하여 구현하였습니다. Getlev 또한 현재 프로세스의 level을 리턴하도록 구현하였습니다.

```
#ifdef MLFQ_SCHED
  if(num==23){//syscall num of yield  PJ2
      myproc()->level=0;
      myproc()->ticks=0;
    }
  if(num==13){//syscall num of sleep  PJ2
      myproc()->level=0;
      myproc()->ticks=0;
    }
#endif
```

위 코드는 syscall.c 에서 구현하였습니다. Yield와 sleep 시스템콜을 사용한 경우, 프로세스가 작업을 마친 것으로 간주해야되기 때문에, 해당 시스템 콜이 호출될경우 프로세스의 레벨과 tick을 초기화합니다.

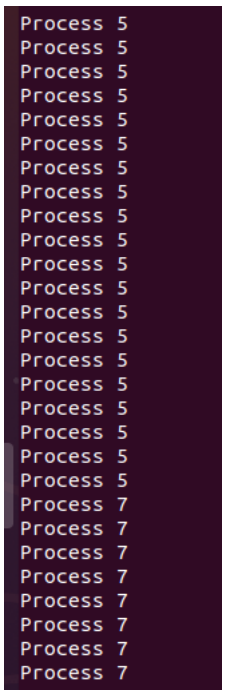
Result

```
[Test 1] without yield / sleep
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
ProcessProcess 6
Process 6
Process 6
Process 6
Process 6
Proces 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
s 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
```

ml_test 1

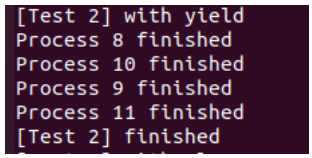
짝수 프로세스가 먼저 스케줄되고, round robin 방식으로 계속해서 프로세스가 바뀌며 돌아갑니다

.



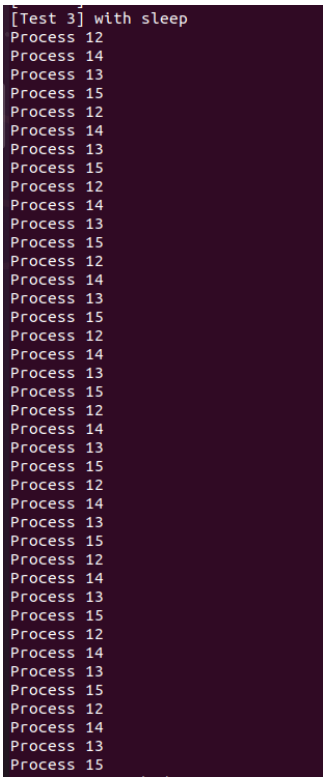
홀수 pid를 가지는 프로세스의 경우, 마지막에 할당되고, 프로세스가 끝날때까지 cpu를 차지하는 FCFS 방식으로 스케줄러가 동작합니다.

ml_test2



짝수인 프로세스들은 yield될때마다 번갈아 실행되어 유사한 시간에 끝나고, 홀수인 프로세스들은 yield되어도 다시 가장 낮은 pid의 프로세스가 할당됩니다.

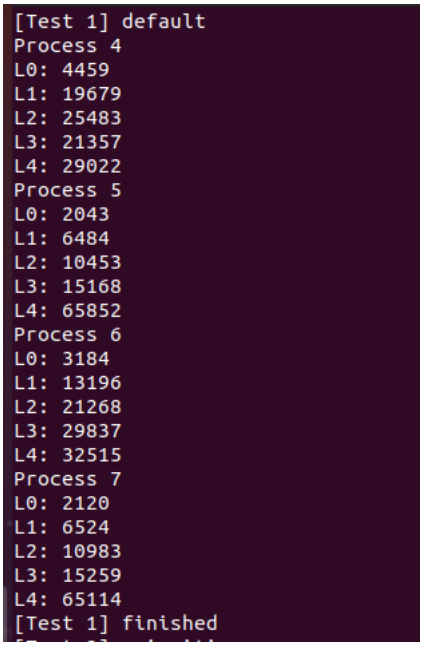
ml_test 3



sleep 상태가되면 프로세스가 할당될 수 없기에 다음 조건에 맞는

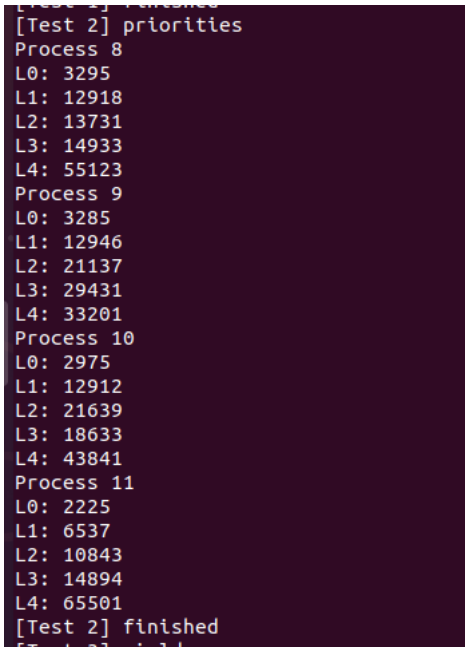
올바른 프로세스가 할당되는 것을 확인 할 수 있습니다.

Mlfq_test



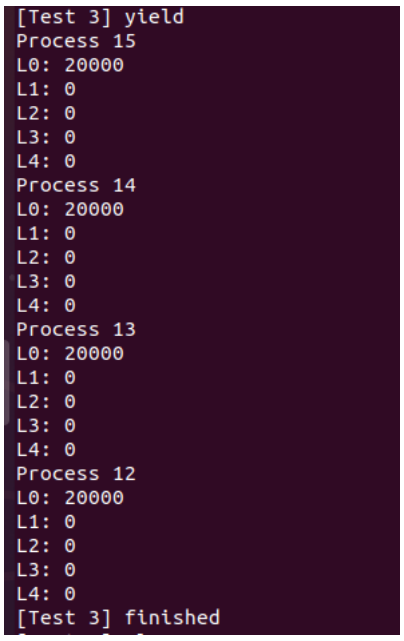
큐의 레벨이 올라감에 더 높은 time quantum을 가지고, 그 결과 더 오랜시간 동안 머무는 것을 확인 할 수 있습니다. 또한

모든 프로세스들이 큐에 머무는 시간이 비슷한 것 또한 확인 할 수 있습니다.

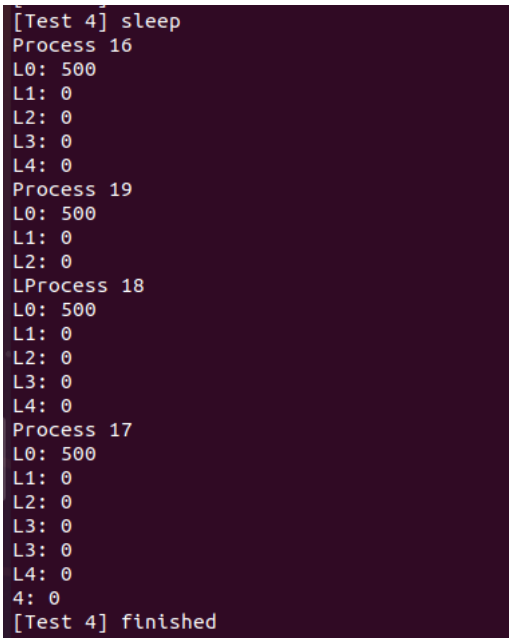


Pid가 더 높은 프로세스가 먼저 나오지만, 결국 전체적인 시간사용량은 유사하기에 비슷한 시간에 종료됩니다.

Mlfq_test 3

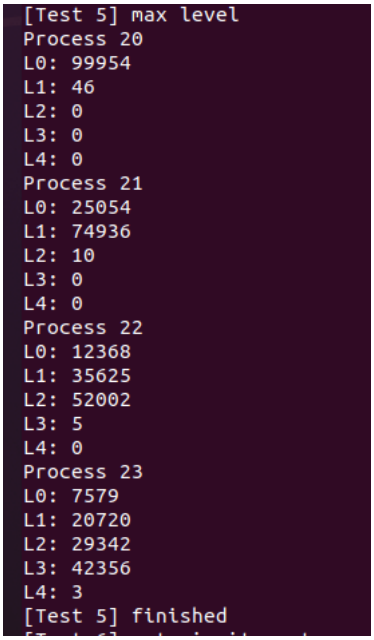


MLfq_test 4



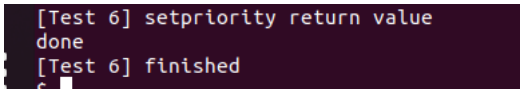
Sleep, yield 시스템콜 호출시 프로세스의 시간과 레벨이 초기화 되기에 계속해서 최하위큐에 머무는 것을 확인 할 수 있습니다.

MLFQ_test 5



각 프로세스가 자신이 있고 싶어하는 큐에 편향되게 위치한 것을 확인 할 수 있습니다.

MLFQ_test 6



함수가 에러없이 작동함을 확인할 수 있습니다.

Trouble Shooting

우선 makefile에서 많은 시행착오가 있었습니다. 과제에 명시된대로 make 시에 값을 받아서 코드에 적용시켜야 했는데, 이를 어떻게 구현할지에 고민을 하기위해 makefile을 수없이 수정하였는데, makefile에선 사소한 변화도 크게 작용되어 많은 디버깅이 필요하였습니다.

또한 pid가 1,2 인 프로세스들에 대한 예외처리에서도 많은 문제가 발생하였습니다.

Pid가 1인 프로세스와 2인프로세스는 각각 init, shell 프로세스로, 두 프로세스 모두 필수적이기에 스케줄러에 들어가 다른 프로세스들과 동일한 취급을 받으며 스케줄링될시 에러가 발생하게 되어, if문을 통하여 예외처리를 해주었습니다. MULTILEVEL QUEUE 에서 FCFS 스케쥴시에 문제가 발생하였습니다. 처음엔 단순히 먼저 들어온 process의 아이디는 뒤에 들어온 프로세스보다 낮을 것이라는 이론을 통해 pid가 낮은 것을 우선적으로 할당하는 방식을 사용하였지만, 만약 pid= 1,3,5 인 프로세스가 sleep한뒤, 5,3,1 순으로 깨어났을 때, 5,1,3 순으로 깨어나는 등, pid 기반의 방식에는 한계를 발견하여 이를 queue 자료구조를 사용하여 먼저 들어온 process를 우선적으로 할당하도록 하여 문제를 해결하였습니다.

또한 MLFQ을 구현할 때에, 처음에는 마지막 큐에서 time quantum을 모두 소모한 프로세스를 SLEEP 상태로 만들어주고, priority boost에서 wake 시켜주는 방식을 통하여 구현하려고 하였으나, yield 를 통해 cpu를 넘겨줄 때에 해당 함수에서 process 의 상태를 RUNNABLE하게 만드는 것을 확인하고, 프로세스 구조체인 proc에 마지막 큐에서 time quantum을 모두 소모한 프로세스를 따로 표시하여 예외처리를 하는 방식으로 해결하였습니다. 또한 'tick' 에 대한 개념의 혼동이 제가 과제를 구현할 때 가장 큰 문제가 되었는데, Process 가 cpu를 사용한 시간의 개념으로 작용하는 tick과, 전체 시간의 흐름을 나타내는 tick의 개념을 계속해서 혼동하여 구현시 많은 에러가 발생하였습니다.