운영체제_PROJ_#1

ELE3021_project01_12299_2018009216_최현용

목표

기존의 Round-Robin 방식의 스케줄러를 대신할 새로운 MLFQ 스케줄러를 xv6에 구현

실행방법

1. make
2. make fs.img
3. qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512

기본적인 동작구조

```
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
```

```
// Common CPU setup code.
static void
mpmain(void)
{
  cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
  idtinit();       // load idt register
  xchg(&(mycpu()->started), 1); // tell startothers() we're up
  scheduler();     // start running processes
}
```

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  queue_init(&L0);
  queue_init(&L1);
```

이런 순서를 따라서 실행하다가 scheduler를 호출하게 되고 scheduler에 있는 for문 무한루프를 돌면서 계속 실행됩니다.

이렇게 스케줄러가 계속 동작하고있고, 동작중 오는 trap은 trap.c에 있는 trap method에서 처리해줍니다.

DESIGN

각 디자인에 대한 간단한 설명만 있고, 각 구현에 대한 자세한 설명은 IMPLEMENT와 원문코드의 주석에 있습니다!

LOCK

저는 이 프로젝트에서 다음과 같은 lock을 이용하였습니다.

1. ptable.lock
   a. 기존에 존재하는 lock, process table에대한 동시접근을 제한합니다.
2. tickslock
   a. 기존에 존재하는 lock, global tick에 대한 동시접근을 제한합니다.
3. sched_lock
   a. schedulerLock(), schedulerUnlock() 구현을 위해 추가한 lock입니다.
   b. 해당 lock이 있는 경우 해당프로세스만 실행하고 MLFQ는 실행되지 않습니다.
   c. 해당 lock이 없는 경우 MLFQ가 실행됩니다.
4. queue_lock

a. 해당 queue에 대한 동시접근을 제한합니다.

b. queue lock은 각각의 큐마다 존재합니다. 즉, L0, L1, L2_0, L2_1, L2_2, L2_3 총 6개의 queue lock이 존재합니다.

AT proc.h

헤더에 구현되어 있는 proc구조체에 아래의 변수들을 추가했습니다.

```
int queue_state;              // 0 : queue에 안들어있는거, 1 : queue에 들어있는거
int queue_level;              // queue를 구분해줌 0,1,2 존재
int priority;                 // 해당 프로세스의 priority를 표시해줌
int ticks;                    // 해당프로세스의 time quantum
```

AT proc.c

1. queue
   a. queue 구조체를 구현하고, 또한 queue에 spinlock을 추가하여 동시접근을 막았습니다. queue가 가질 수 있는 최대프로세스는 NPROC의 크기입니다.
   b. enqueue, dequeue를 구현하였습니다. 이 함수를 이용하여 queue에 접근할 때 함수 내부에서 lock을 이용하여 동시접근을 제한 하였습니다.
   c. 과제명세에 있는 ‘해당 프로세스를 L0큐의 맨앞으로 넣기’를 위해 enqueue_L0_front 함수를 따로 구현했습니다.
   d. L0, L1큐를 생성하였고, L2의 경우 priority에 따라서 실행순서가 달라지기 때문에 L2 큐의 경우는 L2_0, L2_1, L2_2, L2_3 4개의 큐로 구성하였습니다.
2. scheduler
   a. 기존의 RR형태의 스케줄러를 MLFQ로 바꾸어 구현하였습니다.
   b. 우선 for문을 통해 각각의 프로세스들의 level과 L2의 경우 priority까지 확인하며, 해당 queue에 넣어줍니다.
   c. for 문에서의 enqueue가 끝난 경우 L0 → L1 → L2_0 → L2_1 → L2_2 → L2_3 순으로 확인하며 우선순위에 따라 dequeue를 통해 해당 프로세스를 실행시킵니다.
   d. mulit level queue에 대한 부분은 proc.c 의 scheduler()에서 처리하여주었고, tick에 관한 즉 time quantum에 관한 부분은 trap.c에 있는 interrupt handler부분에서 따로 처리하였습니다.

AT proc.c 에 존재하는 trap.c에서 쓰이는 함수들

1. void degrade()
   a. 이 함수는 현재 실행중인 프로세스가 time quantum을 다쓴경우 level을 높이거나, level 이 2인경우 priority를 낮추는 함수입니다.
2. priorityBoosting()
   a. 이 함수는 priorityBoosting을 위해 구현한 함수입니다.
   b. 기본적으로 global tick을 초기화해주고, enqueue와 dequeue를 이용해 모든 process들을 L0 queue에 enqueue합니다.

AT proc.c // 각종 SYSTEM CALL 구현

1. void setPriority(int pid, int priority);
   a. 해당 함수는 해당 pid를 가진 process의 priority를 설정해주는 함수입니다.
   b. ptable.lock을 잡고 ptable을 for문으로 돌면해 해당 pid를 가진 process를 찾습니다.
   c. 해당 프로세스의 priority를 설정해주는데, queue level이 2인경우 문제가 발생합니다.
      i. 이 경우 priority에 따라 들어가야하는 queue가 달라지기 때문에 ,enqueue, dequeue로 이를 처리했습니다.
2. getLevel()
   a. myproc()를 이용하여 해당 프로세스에 접근해서 queue_level정보에 접근해서 해당 값을 return합니다.
3. void yield()
   a. 이미 처음부터 구현되어있습니다.
4. void schedulerLock(int password)
   a. 이는 앞서 말한 sched_lock을 이용하여 구현하였습니다.

b. 단순히 cpu가 1개인 상황이므로, lock을 잡고있는 상태이면 timer interrupt handling부분에서 cpu를 넘겨주는 yield가 실행되지 않도록 하였습니다. 그렇게 함으로써, 현재 실행중인 프로세스만이 계속 실행될 수 있게 하였습니다.

c. 또한 비밀번호가 틀린경우 명세에 맞게 구현하였습니다.

d. 추가로, lock을 이미 잡고있는데 또 호출한 경우 이경우 cpintf를 통해 해당사실을 알리고, 해당 syscall을 종료하였습니다.

e. 또한 interrupt로도 호출가능하기 위해 interrupt handler에서 이를 처리할 수 있게 하였습니다.

5. void schedulerUnlock(int password)

a. 이는 앞서 말한 sched_lock을 이용하여 구현하였습니다.

b. 해당 비밀번호가 틀린경우 명세에 맞고 구현하였습니다ㅏ.

c. ptable.lock을 잡고 priority, queue_level, time quantum들을 알맞게 설정하고 해당 프로세스를 큐의 앞에 넣어주었습니다.

d. 또한 interrupt로도 호출가능하기 위해 interrupt handler에서 이를 처리할 수 있게 하였습니다.

AT proj1_syscall.c

이 파일에서는 위에서의 system call을 위한 wrapper function이 정의되어 있습니다.

AT trap.c

1. tvinit(void) : trap vector init : 두 함수 schedulerLock, schedulerUnlock도 실습에서의 128 interrupt를 발생시키는 실습과 마찬가지로 user_level에서 실행할 수 있게 설정하였습니다.

```
SETGATE(idt[T_SCHED_LOCK], 1, SEG_KCODE<<3, vectors[T_SCHED_LOCK], DPL_USER);
SETGATE(idt[T_SCHED_UNLOCK], 1, SEG_KCODE<<3, vectors[T_SCHED_UNLOCK], DPL_USER);
```

2. trap 함수내부 // trap handler

a. 129 interrupt처리 (schedulerLock interrupt)

a. 또한 interrupt를 통한 호출또한 비밀먼호를 인자로 받기 위해 eax 레지스터를 이용하였습니다.

```
if(tf->trapno == T_SCHED_LOCK) //이 if 문을 통해 129번 interrupt를 처리해 줍니다.
```

b. 130 interrupt를 호출한 경우 (schedulerUnlock 처리)

a. 또한 interrupt를 통한 호출또한 비밀먼호를 인자로 받기 위해 eax 레지스터를 이용하였습니다.

```
if(tf->trapno == T_SCHED_UNLOCK) //이 if 문을 통해 129번 interrupt를 처리해 줍니다.
```

c. timer interrupt를 호출한 경우 → tick에 관련된 모든 처리를 담당하도록 설계했습니다.

a. timer interrupt가 호출될 때마다 tickslock을 잡고, global ticks을 1 늘려줍니다.

b. 그리고 현재 실행중인 프로세스의 time quantum을 1 줄여줍니다.

c. 그리고 global ticks이 100이 될때마다. proc.c에 정의되어있는 priorityBoosting을 작동시킵니다.

a. 이때 scheduler lock이 있는 상태라면 lock을 unlock하고 boosting을 시작하고, 해당 프로세스를 queue L0의 맨앞으로 보내줍니다.

d. 모든 처리가 끝난경우 tickslock을 반환합니다.

d. preemptive를 실행하는 부분

```
if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER && myproc()->tick
```

이때는 이미 구현되어 있는 yield()를 통해 다음 프로세스에게 cpu를 넘겨줍니다.

단, scheduler lock을 확인하여 lock이 있는경우 yield()를 호출하지 못하게 하여 schedular lock을 구현하였습니다.

또한 “myproc()->ticks <= 0”을 추가하여 time quantum을 따 쓴 경우, 또는 해당 프로세스를 다 실행한 경우 2가지 경우에만 cpu를 넘겨주도록 구현하였습니다.

예를 들어 L0큐의 경우 4ticks동안 실행하고 넘겨주면 L1큐의 경우 한번에 6ticks 을 실행하고 cpu를 넘겨줍니다.

IMPLEMENT

분량이 너무 길어지지 않게 코드의 일부분만 가져왔습니다. 모든 원형확인을 위해 어디서 온 코드인지 표시해 두었습니다. 함수에 대한 간단한 소개가 있고, 더욱 자세한 코드설명은 원본에 주식으로 있습니다.

1. queue 및 관련 함수 소개

```
//All FROM proc.c

//queue 자료구조, NPROC만큼의 process를 가질수 있고, spinlock을 추가해 동시접근을 막았습니다.
//또한 원형큐로 구현해서 지속적인 enqueue, dequeue에서 문제가 발생하지 않도록 만들었습니다.
//queue에 대한 모든 함수와, 변수는 proc.c에 존재합니다.
struct queue {
  struct spinlock lock;
  struct proc* procs[NPROC];
  int front, rear;
  int size;
};

//앞으로 사용할 queue를 생성하였습니다. L0은 L0 queue이고, L2_0는 L2큐의 priority가 0인 큐입니다.
struct queue L0;
struct queue L1;
struct queue L2_0;
struct queue L2_1;
struct queue L2_2;
struct queue L2_3;

//큐를 초기화 해주는 함수입니다. 원형은 proc.c에 존재합니다.
void queue_init(struct queue *q)

//기존적인 enqueue, dequeue함수입니다. 제가 보통의 enqueue,dequeue와 다른점만 설명하겠습니다.
//다른점은 enqueue, dequeue내부에서 시작하기전 acquire(&q->lock)
//나가기전에 release(&q->lock);를 해서 해당 queue에 대한 동시접근을 막습니다.
//또한
int enqueue(struct queue *q, struct proc *p);
struct proc *dequeue(struct queue *q);
//이 함수에서 p->queue_state를 통해 해당 프로세스가 queue내부에 존재하는지 확인합니다.
//1인경우 queue에 들어가 있는 상태이고, 0인경우 queue에 들어가있지 않은 상태입니다.

//이 함수는 과제명세의 구현을 위해 정의한 함수입니다.
//이 함수는 p process를 L0 queue의 맨 앞에 enqueue합니다.
void enqueue_L0_front(struct proc* p);
```

2. scheduler구현 설명

```
//ALL from proc.c / void scheduler(void)

//먼저 스케줄러에서 사용할 queue들을 초기화 합니다.
queue_init(&L0);
queue_init(&L1);
queue_init(&L2_0);
queue_init(&L2_1);
queue_init(&L2_2);
queue_init(&L2_3);

//ptable을 for문으로 돌면서 해당하는 queue에 process를 넣어줍니다.
//L2 queue의 경우 위의 DESIGN에서 설명한것과 같이 priority에 따라 다른 큐에 넣어주기 때문에 또 따로 처리했습니다
//process의 상태가 RUNNALBE하지 않다면 큐에 넣지 않습니다.
```

```
//또한 queue_state 를 확인하여 이미 queue에 들어가 있는 상태이면 queue에 넣지 않습니다.
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
        continue;
    if(p->queue_state == 1)//이미 queue에 들어가있다면 enqueue할 필요가 없으므로
        continue;
    if(p->queue_level == 0){
        enqueue(&L0, p);
    }else if(p->queue_level == 1){
        enqueue(&L1, p);
    }else if(p->queue_level == 2){
        if(p->priority == 0)
            enqueue(&L2_0, p);
        else if(p->priority == 1)
            enqueue(&L2_1, p);
        else if(p->priority == 2)
            enqueue(&L2_2, p);
        else if(p->priority == 3)
            enqueue(&L2_3, p);
    }
}

//ptable 전체를 for문으로 돌면서 각각의 프로세스들을 해당하는 큐에 넣어주었습니다.
//따라서, 이제는 해당 큐를 우선순위에 따라서 확인하면서 가장우선순위가 프로세스하나를 dequeue해주어 실행합니다.
//또한 실행할 process가 없는 경우를 따로 뒤서 실행할 프로세스가 없는 경우 for 무한루프를 계속 돌도록 했습니다.
//dequeue후 프로세스포인터가 NULL일 경우 따로 예외 처리했습니다.
if(L0.size > 0){
    p = dequeue(&L0);
}else if(L1.size > 0){
    p = dequeue(&L1);
}else if(L2_0.size > 0){
    p = dequeue(&L2_0);
}else if(L2_1.size > 0){
    p = dequeue(&L2_1);
}else if(L2_2.size > 0){
    p = dequeue(&L2_2);
}else if(L2_3.size > 0){
    p = dequeue(&L2_3);
}else{ //예외상황
    release(&ptable.lock);
    continue;
}
if(p==0){
    cprintf("result of dequeue is NULL\n");
    continue;
}

//뒷부분은 원래의 코드와 같아서 따로 설명하지 않겠습니다.
```

3. 시스템콜 구현 설명

```
//from proc.c

//현재 실행중인 프로세스의 level을 return합니다.
int getLevel(void){
```

```
  return myproc()->queue_level;
}
```

```
//from proc.c

//구현 설명
//ptable.lock을 잡고 for문을 이용해 돌면서 해당 pid와 같은 pid를 가지고 있는 process정보를 갖고옵니다.
//그뒤에 L0, L1인경우 priority르 바꿔줍니다.
//하지만 L2의 경우 priority에 따라 실행순서가 바뀌고, 앞에 DESIGN에서 설명했듯이,
//L2의 경우 priority에 따라 queue가 다르므로 queue를 이동시켜줍니다.
void setPriority(int pid, int priority){
  //cprintf("proc.c/setPriority\n");
  if(priority > 3 || priority < 0){  //예외처리를 위한 부분입니다.
    //cprintf("setPriority error!, priority is 0~3, out of bound!\n");
    return;
  }
  struct proc *p;
acquire(&ptable.lock); //ptable에 접근하는 거니까 lock이 필요하겠지???
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->pid == pid){
    if(p->queue_state == 0){ //queue에 안들어가있는 상태이면
      p->priority = priority;
      break;
    }else{ //해당 process가 queue에 들어가있는 상태
      if(p->queue_level != 2){ //L2큐를 제외하고 나머지 큐는 priority가 큐에서 상관없으므로
        p->priority = priority;
        break;
      }else{ //L2인 상태에서 priority는 순서에 영향을 미치므로
        struct proc* temp_proc1;
        if(p->priority == priority)
          break;
        //priority에 따라 queue 를 바꿔줘야함
        //해당 pid에 맞는 process를 찾는과정
        //enqueue와 dequeue안에는 locking protocal이 구현되어 있으므로 같은 큐내에선 동시에 접근x
        if(p->priority == 0){
          while(1){
            temp_proc1 = dequeue(&L2_0);
            if(temp_proc1->pid == pid){
              break;
            }
            enqueue(&L2_0, temp_proc1);
          }
        }else if(p->priority == 1){
          while(1){
            temp_proc1 = dequeue(&L2_1);
            if(temp_proc1->pid == pid){
              break;
            }
            enqueue(&L2_1, temp_proc1);
          }
        }else if(p->priority == 2){
          while(1){
            temp_proc1 = dequeue(&L2_2);
            if(temp_proc1->pid == pid){
              break;
            }
            enqueue(&L2_2, temp_proc1);
```

```
          }
        }else if(p->priority == 3){
          while(1){
            temp_proc1 = dequeue(&L2_3);
            if(temp_proc1->pid == pid){
              break;
            }
            enqueue(&L2_3, temp_proc1);
          }
        }else{
          cprintf("proc.c/setPriority err12\n");
          break;
        }
        //이제 찾은 queue에서 찾은 process를 알맞은 priorty queue에 넣어준다.
        if(priority == 0){
          enqueue(&L2_0, temp_proc1);
        }else if(priority == 1){
          enqueue(&L2_1, temp_proc1);
        }else if(priority == 2){
          enqueue(&L2_2, temp_proc1);
        }else{
          enqueue(&L2_3, temp_proc1);
        }
      }
    }
  }
  release(&ptable.lock); //lock 해제
}
```

```
//from proc.c

//password를 인자로 받고 틀린경우 pid, time quantum, queue level을 출력하고 종료합니다.
//예외처리로 이미 누군가가 lock을 잡고있는경우 이를 출력해주고 빠져나옵니다.
//위의 상황들을 통과하면, lock을 획득합니다.
//lock을 획득하면 trap.c에 있는 interrupt handler에 있는 yield()를 실행하지 못하게 하여 현재프로세스만 실행하
//global tick은 priority boosting없이 0으로 설정
//모든 프로세스들의 time quantum은 초기화됨
//기존에 존재하는 프로세스들만 호출할 수 있음
//global tick == 100 일때 priority boosting이 발생하면 기존의 MLFQ로 돌아감
//129번 인터랍트 발생시 실행
void schedulerLock(int password){ //password : 2018009216 //tickslock을 사용함
  if(STUDENT_NUM != password){ //password가 틀린경우
    cprintf("wrong schedulerLock password!\n");
    cprintf("pid : %d, time quantum : %d, queue level : %d\n",myproc()->pid, myproc()->ticks, my
    if(sched_lock.locked == 1)//
      release(&sched_lock);
    exit();
  }
  if(sched_lock.locked > 0){ //누군가가 이미 lock을 잡고 있으면 그냥 빠져나옴
    cprintf("other process already has schedulerLock!\n");
    return;
  }
  cprintf("\n");
  cprintf("");
  cprintf("schedulerLock start\n");
  acquire(&sched_lock);
```

```
  acquire(&tickslock);
  ticks = 0;
  release(&tickslock);
  //스켜줄링이 안되게 함
}
```

```
//from proc.c

//password를 인자로 받고 틀린경우 pid, time quantum, queue level을 출력하고 종료합니다.
//예외처리로 아무도 lock을 잡고있지 않은 경우 이를 출력해주고 빠져나옵니다.
//위의 상황들을 통과하면, unlock을 합니다.
//unlock을 하면 trap.c에 있는 interrupt handler에 있는 yield()를 다시 실행하도록하여 기존의 MLFQ로 돌아갑니
//해당 프로세스는 L0큐의 맨앞으로 이동 priority = 3, time quantum 초기화
//암호가 일치하지 않을 시 pid, time quantum, 현재 위치한 큐의 level을 출려합니다.
//그리고 강제종료
//130번 인터랍트 발생시 실행
void schedulerUnlock(int password){ //password : 2018009216
if(STUDENT_NUM != password){ //password가 틀린경우
  cprintf("wrong schedulerLock password!\n");
  cprintf("pid : %d, time quantum : %d, queue level : %d\n",myproc()->pid, myproc()->ticks, my
  exit();
}
//locked == 0 : unlocked, locked == 1 : locked
if(sched_lock.locked == 0){ //예외상황 처리
  cprintf("scheduler lock is already unlocked! so, can't unlock!\n");
  return;
}
//cprintf("schedulerUnlock start\n");
cprintf("trap.c/pid : %d\n",myproc()->pid);
acquire(&ptable.lock);
myproc()->priority = 3;//명세조건
myproc()->queue_level = 0;//명세조건
myproc()->queue_state = 1;//queue에 들어갔다는 의미
myproc()->ticks = 4; // queue level : 0이므로 time quantum은 4이다.
release(&ptable.lock);
enqueue(&L0, myproc());
struct proc* temp;
for(int i = 0; i < L0.size - 1; i ++){//해당 프로세스를 L0 queue의 맨 앞으로 이동
  temp = dequeue(&L0);
  enqueue(&L0, temp);
 }
 release(&sched_lock);
 //cprintf("trap.c/pid : %d\n",myproc()->pid);
 //cprintf("end sched_unlock\n\n");
}
```

SYSCALL을 위한 wrapper function

```
//ALL from proj1_syscall.c

// Give up the CPU for one scheduling round.
int sys_yield(void){
    yield();
    return 0;
}
//프로세스가 속한 큐의 레벨을 반환
int sys_getLevel(void){
```

```
    return getLevel();
}

//해당 pid의 프로세스의 priority를 설정
int sys_setPriority(void){ //int pid, int priority 2개의 인자를 받음
    int pid;
    int priority;
    if(argint(0, &pid) < 0){
        cprintf("proj1_syscall.c/sys_setPriority argint(0, &pid) error!");
        return -1;
    }
    if(argint(1, &priority) < 0){
        cprintf("proj1_syscall.c/sys_setPriority argint(1, &pid) error!");
        return -1;
    }
    setPriority(pid, priority);
    return 0;//정상종료
}
//129번 인터럽트 호출시 이 시스템콜 실행
//해당 프로세스가 우선적으로 스케줄링 되도록 합니다.
int sys_schedulerLock(void){
    cprintf("sys_schedulerLock called!\n");
    int password;
    if(argint(0, &password) < 0){
        cprintf("proj1_syscall.c/sys_schedulerLock/argint error!\n");
        return -1;
    }
    schedulerLock(password);
    return 0;
}
//130번 인터럽트 호출시 이 시스템콜 실행
//해당 프로세스가 우선적으로 스케줄링 되던 것을 중지합니다.
int sys_schedulerUnlock(void){
    cprintf("sys_schedulerUnlock called!\n");
    int password;
    if(argint(0, &password) < 0){
        cprintf("proj1_syscall.c/sys_schedulerUnlock/argint error!\n");
        return -1;
    }
    schedulerUnlock(password);
    return 0;
}
```

4. interrupt handler (trap) 구현 설명.

```
//ALL from trap.c

//trap vector init할때 sched_lock도 같이 초기화하였습니다.
initlock(&sched_lock, "sched_lock");


//schedulerLock, schedulerUnlock을 interrupt 129, 130으로도 호출할 수 있도록 하기 위해 구현하였습니다.
//실습내용과 다른 내용은 인자를 받아서 이를 schedulerLock or Unlock에 넣어준다는 것입니다.
//이는 trapframe에 있는 eax register를 이용하여 구현하였습니다.
if(tf->trapno == T_SCHED_LOCK){
    if(myproc()->killed)
```

```
      exit();
    int temp = tf->eax;
    //tf->eax = 64;//syscall num
    myproc()->tf = tf;
    schedulerLock(temp);
    if(myproc()->killed)
      exit();
    return;
  }
  if(tf->trapno == T_SCHED_UNLOCK){
    if(myproc()->killed)
      exit();
    int temp = tf->eax;
    //tf->eax = 64;
    myproc()->tf = tf;
    schedulerUnlock(temp);
    if(myproc()->killed)
      exit();
    return;
  }

//switch문 안의 timer interrupt를 처리하는 부분
//기존의 코드와 다른 부분은 myproc()의 ticks을 감소, degrade하는 부분, priorityBoosting() 하는 부분
//이 다른부분에 대한 설명은 해당 함수의 설명에서 이미 했음
case T_IRQ0 + IRQ_TIMER: //
    if(cpuid() == 0){
      //cpuid()는 현재 실행중인 CPU의 ID 반환
      //"cpuid() == 0"은 현재 실행중인 코드가 첫 번째 cpu에서 실행하고 있는지를 확인함
      acquire(&tickslock); //ticks은 trap.c에서 정의된 global variable
      ticks++; //1 tick 은 cpu의 클럭주기 : 주기는 cpu마다 다름, 보통 10ms ~ 15ms
      if(myproc()){
        myproc()->ticks--; //틱을 올리는거
      }
      wakeup(&ticks); // chan이 ticks인 모든 프로세스를 sleeping -> runnable
      if(ticks >= 100){ //priority boosting 이 필요
        //sched_lock을 잡고있는 상태면 풀어줘야함
        if(sched_lock.locked == 1){//sched_lock이 있는 상태이면
          release(&sched_lock);
          enqueue_L0_front(myproc());
        }
        //해당 프로세스는 L0큐 맨 앞으로이동
        //여기 trap.c에선 queue에 접근x -> proc.c에서 함수를 정의해서 사용함
        priorityBoosting();
      }
      release(&tickslock);
    }
    lapiceoi();
    break;

//기존의 preemptive한 실행을 해주는 부분
//timer interrupt가 발생할 때마다 yield를 호출하여 context switching을 수행하고 scheduler를 호출해서 다음
//만약 sched_lock을 잡고있는 경우 -> 즉, schedulerLock을 호출한 상태이면 yield()를 실행하지 못하게 하여, MLF
if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER myproc()->ticks <= 0){
           //이경우 time qantum을 다 썼으므로 degrade시켜준다.
      degrade();
      //현재 실행중인 프로세스를 queue에 저장하고 contexswitch실행해야함
      if(sched_lock.locked == 0){//만약 scheulerLock이 unlock이면
```

```
        yield();
      }
  }
```

5. 이외의 함수 설명

degrade() // 해당 함수는 process가 time quantum을 다 사용한 경우, queue level, priority를 감소시키는 것입니다.

```
//AT proc.c

void
degrade()
{
  //switch문을 통해 구현
  //queue_level에 따라 케이스를 나누어 구현
  switch(myproc()->queue_level){
    case 0: //현재 큐가 L0
  acquire(&ptable.lock);
  enqueue(&L1, myproc());
  myproc()->queue_level++;
  myproc()->ticks = 6; //L1의 time quantum은 6이므로
  release(&ptable.lock);
  break;
case 1: //현재 큐가 L1 -> L2 로 이동해야하므로, L2는 priority에 따라 나뉘므로 switch문으로 case를 나누어
  acquire(&ptable.lock);
  myproc()->queue_level++;
  myproc()->ticks = 8; //L2의 time quantum은 8이므로
  switch(myproc()->priority){
    case 0:
      enqueue(&L2_0, myproc());
      break;
    case 1:
      enqueue(&L2_1, myproc());
      break;
    case 2:
      enqueue(&L2_2, myproc());
      break;
    case 3:
      enqueue(&L2_3, myproc());
      break;
    default: //priority의 범위는 0~3인데 이 이외의 경우 오류출력
      cprintf("proc.c/degrade/default error2 \n");
      break;
  }
  release(&ptable.lock);
  break;
case 2: //현재 큐가 L2 //case 1의 경우와 유사하게 구현함
  acquire(&ptable.lock);
  myproc()->ticks = 8;
  if(myproc()->priority != 0)
    myproc()->priority--;
  release(&ptable.lock);
  switch(myproc()->priority){
    case 0:
      enqueue(&L2_0, myproc());
      break;
    case 1:
```

```
          enqueue(&L2_1, myproc());
          break;
        case 2:
          enqueue(&L2_2, myproc());
          break;
        default: //priority의 범위는 0~3인데 이 이외의 경우 오류출력
          cprintf("proc.c/degrade/default error3 \n");
          break;
      }
      break;
    default: //queue_level의 범위는 0~2인데 이 이외의 경우 오류출력
      cprintf("proc.c/degrade/default error1\n");
      break;
  }
}
```

priorityBoosting() // priorityBoosting을 구현한 함수입니다.

```
//이 함수는 ticks을 사용하지만 이걸 호출하기 전에 이미 tickslock을 잡고 있고
//이 함수가 다 실행뒤 빠져나가고 이 함수를 호출한 곳에서 tickslock을 release하기 때문에 tickslock을 건들 필요기
void
priorityBoosting(void)
{
  struct proc* p;
  ticks = 0;//이미 이걸부르는 timer interrupt handler에서 tickslock을 잡고 있음

  acquire(&ptable.lock); //모든 process의 상태를 변경하기 위해서 ptable.lock을 잡음
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == RUNNABLE){
      p->queue_level = 0;
      p->priority = 3;
      p->ticks = p->queue_level*2 + 4;
    }
  }
  release(&ptable.lock);
  //queue_lock에 문제가 생길거 같지만, 큐가 달라서 락도 다르다!
  //이렇게 실행 -> 순서를 유지할 수 있음 : L0->L1->L2_0->L2_1->L2_2->L2_3 순으로 L0에 들어가 있음
  while(L1.size > 0){
    //cprintf("proc.c/priorityBoosting/while(L1.size > 0)");
    enqueue(&L0, dequeue(&L1));
  }
  while(L2_0.size > 0){
    //cprintf("proc.c/priorityBoosting/while(L2.size > 0)");
    enqueue(&L0, dequeue(&L2_0));
  }
  while(L2_1.size > 0){
    enqueue(&L0, dequeue(&L2_1));
  }
  while(L2_2.size > 0){
    enqueue(&L0, dequeue(&L2_2));
  }
  while(L2_3.size > 0){
    enqueue(&L0, dequeue(&L2_3));
  }
}
```

RESULT

mlfq_test 결과

```
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00



Booting from Hard Disk...
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap st8
init: starting sh
$ mlfq_test
MLFQ test start
[Test 1] default
Process 4
L0: 156
L1: 1089
L2: 98755
L3: 0
L4: 0
Process 5
L0: 5455
L1: 9525
L2: 85020
L3: 0
L4: 0
Process 6
L0: 10173
L1: 16248
L2: 73579
L3: 0
L4: 0
Process 7
L0: 12890
L1: 21264
L2: 65846
L3: 0
L4: 0
[Test 1] finished
done
```

forktest결과

```
$ forktest
fork test
fork test OK
$
```

이외에도 proj1_test.c라는 파일을 만들어 test를 진행하였습니다.

schedulerLock, schedulerUnlock test

```
//schedulerLock, schedulerUnlock test
//반복되는 부분을 전부다 넣기에는 너무 길어져서 반복되는 부분은 ...으로 대체하였습니다.

//test code
//이 코드는 fork()를 이용하여 부모와 자식프로세스를 실행하고
//부모 process에 scheduler lock을 걸어 test를 진행하였습니다.
#include "types.h"
#include "stat.h"
#include "user.h"

int main(int argc, char *argv[]){

    //interrupt 호출 test
    int student_num = 2018009216;
    printf(1, "before interrupt\n");
    //return 0;

    printf(1, "proj1_test start!\n");
    yield();
    printf(1, "after yield\n");

    printf(1, "getLevel : %d\n",getLevel());
    //schedulerLock(1);
    int i;
    i = 0;
    if(fork() == 0) //자식의경우
    {
        while (1){
            printf(1,"child\n");
        }
    }else{ //부모의 경우
        schedulerLock(2018009216);
        while(i++ <30){
            printf(1,"parent\n");
        }
    }
    printf(1,"i : %d\n",i);
    printf(1,"after inf loop\n");

    exit();
}

sys_schedulerLock called!
parent //parent가 계속반복된다.
parent
parent
'''
'''
parent
parent
parent
parent
parent

//tick == 100일때 priorityBoosting이 발생하고,
//parent process가 L0 queue의 맨앞에 들어가므로 먼저 실행된다.
//그리고 parent process, child process가 반복적으로 실행된다.
```

```
------priorityBoosting after sched_lock------
enqueue L0
parent
parent
parent
parent
parent
parent
parent
parent
parent
child
child
child
child
child
child
child
child
child
child
cparent
parent
parent
parent
parent
parent
parent
```

트러블 슈팅

```
# qemu-system-i386 –nographic –serial mon:stdio –hdb fs.img xv6.img –smp 1 –m 512
qemu-system-i386: –nographic: Could not open '–nographic': No such file or directory
```

이런 오류가 뜬다

해결책

해당 명령어는 xv6 운영체제를 실행시키기 위한 QEMU 가상 머신 명령어입니다.

그러나 입력하신 명령어에서 문제가 발생했습니다. 에뮬레이터에서 "-nographic" 옵션을 인식하지 못하고 "–nographic"으로 입력되어 있다는 것입니다. "-"와 "–"은 다른 문자로 인식되기 때문에 명령어 실행이 실패하게 되는 것입니다.

따라서, 올바른 명령어는 다음과 같이 "-"을 사용해 입력해야 합니다.

“-”, 가 달라서 생기는 문제

```
case T_IRQ0 + IRQ_TIMER: //
    if(cpuid() == 0){
      //cpuid()는 현재 실행중인 CPU의 ID 반환
      //"cpuid() == 0"은 현재 실행중인 코드가 첫 번째 cpu에서 실행하고 있는지를 확인함
      //근데.. cpu가 하나라고 설정했으므로 ... ???
      acquire(&tickslock); //ticks은 trap.c에서 정의된 global variable
      ticks++; //1 tick 은 cpu의 클럭주기 : 주기는 cpu마다 다름, 보통 10ms ~ 15ms
      //cprintf("T_IRQ0 + IRQ_TIMER 1.15!\n");
```

```
      myproc()->ticks++;
      //cprintf("T_IRQ0 + IRQ_TIMER 1.2!\n");
      wakeup(&ticks); // chan이 ticks인 모든 프로세스를 sleeping -> runnable
      if(ticks >= 100){ //priority boosting 이 필요
        priorityBoosting();
      }
      release(&tickslock);
    }
    lapiceoi(); //interrupt처리가 끝나면, 해당 인터랩트를 인터랩트 컨트롤러에게 전달받았음을 알림
    break;
```

```
myproc()->ticks++; // 에러가 나는 부분
---------------------------------------------------------
Booting from Hard Disk..xv6...
cpu0: starting 0
start scheduler
scheduler 1.0
scheduler 1.1
tf->trapno : 32
T_IRQ0 + IRQ_TIMER 1.15!
proc.c/myproc 1.0
proc.c/myproc 1.1
proc.c/myproc 1.2
proc.c/myproc 1.3
proc.c/myproc 1.4
proc.c/myproc 1.5
tf->trapno : 14
proc.c/myproc 1.0
proc.c/myproc 1.1
proc.c/myproc 1.2
proc.c/myproc 1.3
proc.c/myproc 1.4
proc.c/myproc 1.5
unexpected trap 14 from cpu 0 eip 801060f2 (cr2=0x84)
lapicid 0: panic: trap
 8010634f 80105dbf 80105dbf 8010303f 8010318c 0 0 0 0 0
```

에러가 발생하는 코드부분은 time quantum을 계산한 부분입니다.

이 부분에서 에러가 발생했으므로, 따로 함수를 정의해서 처리해주었습니다.

스케줄러에서 이런식으로 for문을 변경했더니 p에 접근할 때마다 error가 발생하여 원래의 형태로 바꾸어 주었습니다.

```
for(i = 0; i < NPROC; i++){
```

Lock을 이중으로 잡은 문제

```
switch(tf->trapno){
//1 tick마다 context switching
//global tick이 100이 될때마다 모든 프로세스들은 L0 큐로 재조정
case T_IRQ0 + IRQ_TIMER: //
  if(cpuid() == 0){
    //cpuid()는 현재 실행중인 CPU의 ID 반환
    //"cpuid() == 0"은 현재 실행중인 코드가 첫 번째 cpu에서 실행하고 있는
    //근데.. cpu가 하나라고 설정했으므로 ... ???
    acquire(&tickslock); //ticks은 trap.c에서 정의된 global varia
    ticks++; //1 tick 은 cpu의 클럭주기 : 주기는 cpu마다 다름, 보통 10
    if(ticks >= 100)
      priorityBoosting();
    cprintf("global ticks : %d\n",ticks);
    if(myproc()){ // myproc가 존재하면 //여기서의 myproc는 커널인가?
      myproc()->ticks++; //틱을 올리는거
      cprintf("myproc name : %s\n",myproc()->name);
      cprintf("myproc()->ticks : %d\n",ticks);
      cprintf("cpu()->proc.name : %s\n",mycpu()->proc->name);
    }
    //cprintf("T_IRQ0 + IRQ_TIMER 1.2!\n");
    wakeup(&ticks); // chan이 ticks인 모든 프로세스를 sleeping ->
    //cprintf("T_IRQ0 + IRQ_TIMER 1.3!\n");
```

```
84
85  static struct proc *initproc;
86  //부팅시점에 생성되는 초기 프로세서
87  //static : 정적 전역변수, 해당 변수에 접근할수 있는
88
89  void
90  priorityBoosting(void)
91  {
92    struct proc* p;
93    acquire(&tickslock);//ticks 초기화
94    ticks = 0;
95    release(&tickslock);
96
97    acquire(&ptable.lock);
98    for(p = ptable.proc; p < &ptable.proc[N
99      if(p->state == RUNNABLE){
100       p->queue_level = 0;
101       p->priority = 3;
102       p->ticks = p->queue_level*2 + 4;
103     }
104   }
105   release(&ptable.lock);
```

이미 tickslock를 잡고있지만 priorityBoosting에서 또 lock을 잡으려고 시도해서 문제가 발생

따라서 lock을 고려해서 다시 프로그래밍하여 문제를 해결하였습니다.

QUEUE구현을 실제로 안하고 process의 정보를 담고있는 proc구조체에 int queue_level정보만 포함하여 구현했습니다.

하지만 이를 이용하여 과제명세를 구현하던중 overhead가 너무 큰거 같아서 queue 를 실제로 구현하여 만들었습니다.

+

xv6를 만든 mit에서 제공해주는 xv6에 대한 pdf도 참조했습니다.

https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev10.pdf