운영체제 Project 1 WIKI

2019008813 문원찬

1. Design

3-level MLFQ를 구현하기 위해 큐 자료구조를 직접 만드는 것이 아닌, 프로세스 자료구조에 몇 가지 변수를 추가해 간접적으로 3-level MLFQ를 구현했다. 프로세스에 추가한 변수로는 속한 큐의 레벨을 나타내는 변수(int L), 큐에서 프로세스의 순서를 나타내는 변수(int order), 우선순위를 나타내는 변수(int priority), 각 큐에서 얼마나 실행되었는지를 나타내는 변수(uint runningtime)이다.

L0, L1은 round robin 방식으로 작동하기 위해 1tick마다 실행 중인 프로세스의 runningtime을 1 증가시키고, runningtime이 각 큐에서 4, 6이 되면 큐의 레벨을 내리는 함수를 만들었다. 또한, 이 함수에선 L2에 있는 프로세스의 runningtime이 8이 되면 priority를 감소시키는 역할도 하게 된다.

100ticks가 되면 Priority boosting을 구현하기 위해 부스팅을 하는 함수를 만들었다. 이 함수는 L0부터 변수들의 오버플로우를 방지하기 위해 들어온 순서대로 order를 1부터 설정해준다. L1, L2도 들어온 순서 그래도 L0의 뒤에 추가해준다. 이때 모든 프로세스들의 priority는 3, runningtime은 0으로 설정해준다.

SchedulerLock(), schedulerUnlock()은 나의 학번인 '2019008813'을 인자로 받아 일치하면 실행되게 구현하였다. 일치하지 않으면 그 프로세스를 종료하고 필요한 정보들을 터미널에 출력해준다. 스케줄러 락 상태일 때 해당 프로세스가 또 스케줄러 락을 하려하면 아무 일도 일어나지 않게 하고, 언락을 호출하면 L0의 맨앞, runningtime = 0, priority = 3으로 만든다. 스케줄러 락 상태에서 100ticks가 되면 L0의 맨앞으로 이동하고 boosting이 되게 구현했다. 스케줄러 락이 걸리지 않은 상태에서 스케줄러 언락을 호출하면 아무일도 일어나지 않게 하였다. 또한 두 함수는 시스템 콜일 뿐만 아니라 인터럽트를 통해서도 실행 될 수 있게 수정해주었다.

구현해야 하는 시스템 콜로는 yield, getLevel, setPriority, schedulerLock, schedulerUnlock이 있는데 yield는 이미 xv6에 존재하므로 시스템 콜로 추가만 해주었다. getLevel은 현재 프로세스의 레벨을 반환하는 함수를 만들고 시스템 콜에 추가했다. SetPriority는 pid와 priority를 인자로 받는데, 해당하는 pid가 없거나 priority가 0~3이 아닐경우, 아무 일도 일어나지 않게 하였다. schedulerLock과 schedulerUnlock은 위에서 설명한 데로 만들고 시스템콜에 추가했다.

자세한 알고리즘 및 설명은 Implement에서 사진과 함께 다뤘다.

2. Implement

a. proc.h 및 proc.c 변수 추가

proc.h

```
// 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)

  int L;                       // L: 0~2, -1: schedulerlock
  int order;                   // order: 1~100, 0 is for locked process.
  int priority;                // priority: 0~3
  uint runningtime;             // time quantum in queue
};
```

우선 프로세스의 자료구조를 수정하기 위해 proc.h에 존재하는 다음의 구조체 proc을 수정해주었다. L은 속한 큐의 레벨을 표현하는 변수로 -1~2의 값으로 존재한다. 0~2는 각 L0, L1, L2를 나타내며, -1은 스케줄러 락을 위한 값으로, 부스팅이나 레벨증감을 하는 함수에 영향을 받지 않게 하기 위해 존재한다. order는 큐에 들어온 순서를 나타내는 변수로, 아무리 순서가 늘어나도 100ticks가 되면 boosting으로 순서를 초기화 하기 때문에 100이상으로론 존재할 수 없다. 0은 스케줄러 락을 잡고 풀릴 때 L0의 첫 순서로 가기 위해 존재한다. priority는 프로세스의 우선순위를 나타내는 변수로 0~3이다. runningtime은 각 큐에서 얼마나 실행되었는가를 나타낸다. 음수는 존재할 수 없기에 혹시 모를 overflow를 방지하기 위해 uint로 설정했다.

proc.c

```
int schedLock = 0; //schedulerLock: 0 or 1
struct proc *p_locked; //process call schedulerLock
//init order of each queue.
int L0_order = 0;
int L1_order = 0;
int L2_order = 0;
```

스케줄러 락이 걸렸는지 확인하는 변수 schedLock을 만들었다. 1이면 락 상태이며 0이면 언락 상태이다. p_locked는 현재 락이 걸려있는 프로세스를 가리킨다. 아래 order들은 각 큐의 끝 순서를 가리키는 변수이다.

b. proc.c에 추가한 함수들 및 작동 알고리즘

proc.c – void checkTime(void)

```
void
checkTime(void){ //call when tick is increased and present p's state is RUNNING.
  acquire(&ptable.lock);
  struct proc *p = myproc();
  if(p->L==0){
    if(p->runningtime==4){ //degrade
      p->L = 1;
      p->runningtime = 0;
      p->order = ++L1_order;
    }
    else
      p->order = ++L0_order; //RR
  }
  else if(p->L==1){
    if(p->runningtime==6){ //degrade
      p->L = 2;
      p->runningtime = 0;
      p->order = ++L2_order;
    }
    else
      p->order = ++L1_order; //RR
  }
  else if(p->L==2){
    if(p->runningtime==8){
      if(p->priority>0)
        p->priority -= 1;
      // p->runningtime = 0; //No degrade
    }
    // p->order = ++L2_order; //For FCFS, do not ratate.
  }
  release(&ptable.lock);
}
```

degrade나 priority를 조정하기 위해 checkTime함수를 proc.c에 생성했다. if문들을 통해 지금 실행 중인 프로세스의 레벨을 체크하고 맞는 레벨에 들어가서 각 역할을 수행해준다. 예를 들어 프로세스가 L0일 경우엔 runningtime이 4가 되면, L1으로 degrade를 해주기 위해 L을 1로 설정하고 runningtime 또한 새로운 큐에 들어가기에 초기화 시켜준다. order는 L1의 마지막 순서를 가리키는 L1_order에 1을 추가하고 그 순번을 갖는다. 만약 runningtime이 4가 아니라면(0~3), 라운드 로빈을 구현하기 위해 순번이 가장 빠른 프로세스가 cpu를 잡을 수 있게 설정했으므로 1tick마다 끝 순번으로 가서 다음 프로세스가 cpu를 잡을 수 있게 해준다.

L1도 L0와 같은 라운드 로빈으로 같은 알고리즘으로 동작한다. 하지만 L2의 경우엔 priority queue이므로 runningtime이 8의 배수일 때마다 priority를 낮춘다. 0이면 더 이상 낮추지 않는다. 라운드 로빈이 아니므로 끝 순번으로 가지 않아도 된다. 오히려 priority가 같으면 큐에 들어온 순서를 따지므로 order는 건들지 않는다.

proc.c – void boosting(void) 중 L0

```
void
boosting(void){ //Call when ticks == 100.
  acquire(&ptable.lock);
  struct proc *p;
  int least_order;
  int catch;
  struct proc *temp;

  //L0 init
  int L0_order_before = L0_order; //Remember last process order.
  L0_order = 0; //init L0_order to prevent overflow.
L0:
  least_order = 100;
  catch = 0;
  temp = ptable.proc;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->L!=0 || p->state != RUNNABLE) //p is in L0 and RUNNABLE.
      continue;
    if(L0_order<p->order && p->order<least_order){ //Find least order process, but not boosting.
      temp = p;
      least_order = temp->order;
      catch = 1;
    }
  }
  p = temp; //Set choosed process.
  if(p->order == L0_order_before){ //Last boosting.
      p->order = ++L0_order;
      p->priority = 3;
      p->runningtime = 0;
      catch = 0;
    }
  if(catch){ //No last boosting.
    p->order = ++L0_order;
    p->priority = 3;
    p->runningtime = 0;
    goto L0;
  }
```

다음은 boosting 함수의 L0부분이다. L0는 이미 L0이므로 그대로 있어도 되지만 priority와 runningtime 초기화를 해야하고 추가로, L0_order가 프로세스가 들어올 때마다 증가해 overflow를 방지하기 위해 boosting될 때 초기화를 시켜주었다. 우선 for문에서 L0이며 RUNNABLE이고 가장 빠른 순서를 가진 프로세스를 찾는다. 찾았다면 catch 값을 1로 바꿔주고 마지막 if문으로 가서 필요한 초기화를 해주고 다시 L0로 가서 for문을 돈다. 만약 마지막 순서라면 아래에서 두번째 if 문을 통해 직접 init해주고 catch를 0으로 바꾸고 루프에서 빠져나온다. 만약 L0이며 RUNNABLE 한 프로세스가 없었다면 catch는 항상 0이므로 다음 L1, 그 후 L2로 가게 된다. L1, L2 또한 위와 같은 알고리즘이며 L0로 이동한다는게 다른 점이다.

trap.c – boosting(), checkTime()

```
if(tf->trapno == T_IRQ0+IRQ_TIMER && ticks==100){
  if(schedLock){ //If locked, unlock and boosting.…
  boosting();
  ticks = 0;
  // cprintf("boosting\n");
}


// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER && ticks!=0){
    myproc()->runningtime += 1;
    checkTime();
    yield();
   }
```

위 if문은 100ticks마다 부스팅을 한 후 ticks를 0으로 초기화 해주기 위한 부분이다. 아래 if문은 1tick마다 현재 돌아가는 프로세스의 runningtime을 1증가시키고 아까의 checkTime()를 선언하고 yield()를 통해 cpu를 놓아준다.

proc.c – int getLevel(void), void setPriority(int pid, int priority)

```
//Add some functions.
int
getLevel(void){ //Return RUNNING process's level.
  return myproc()->L;
}

void
setPriority(int pid, int priority){ //Set priority of process which has same pid.
  struct proc *p;
  if(priority<0 || priority>3) //priority must be 0~3.
    return;
  acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ //If there is no process, then nothing happens.
      if(p->pid == pid)
        p->priority = priority;
    }
  release(&ptable.lock);
}
```

다음은 getLevel()과 setPriority(pid, priority) 함수이다. getLevel()은 현재 프로세스의 level을 반환하고 setPriority는 pid와 priority를 인자로 받는데 만약 priority가 0~3이 아니거나 존재하지 않는 pid라면 아무 일도 일어나지 않게 구현했다.

proc.c – void schedulerLock(int password)

```
void schedulerLock(int password){
  acquire(&ptable.lock);
  struct proc *p = myproc();
  if(p==0)
    return;
  if(password != 2019008813){
    cprintf("pid: %d time quantum: %d level: L%d", p->pid, p->runningtime, p->L);
    exit();
  }
  cprintf("Lock ready %d, tick: %d\n", p->pid, ticks);
  if(!schedLock){ //Lock when unlocked.
    ticks = 0;
    p->L = -1;
    p_locked = p;
    schedLock = 1;
    cprintf("Lock %d, tick: %d\n", p->pid, ticks);
  }
  release(&ptable.lock);
}
```

스케줄러 락을 하기 위한 함수이다. 현재 프로세스가 만약 없다면 아무런 일도 일어나지 않는다. 만약 interrupt로 호출 시에 프로세스를 잡지 않고 있을 수도 있기에 안정성을 더했다. 다음 if문을 통해 password가 맞지 않으면 프로세스의 pid, time quantum, level을 출력하고 프로세스를 종료하도록 했다. 또한, 같은 프로세스에서 락을 두 번 거는 일을 방지하기 위해 if문을 사용하였다. 락은 명세에 나와있는 것처럼 tick을 0으로 초기화하고 schedLock변수를 1로 만들며 스케줄러 락이 걸렸음을 스케줄러에게 알린다. boosting이나 checkTime의 대상이 되지 않기 위해 레벨을 따로 -1로 설정했다. 또한 현재 락이 걸린 프로세스의 포인터를 저장해 둠으로써 스케줄링 시 빠르게 동작하고, 언 락을 할 때 해당 프로세스가 언 락을 호출했는지 확인할 수 있다.

proc.c – void schedulerUnlock(int password)

```
void schedulerUnlock(int password){
  acquire(&ptable.lock);
  struct proc *p = myproc();
  if(p==0)
    return;
  if(password != 2019008813){
    // cprintf("pid: %d time quantum: %d level: L%d", p->pid, p->runningtime, p->L);
    exit();
  }
  if(schedLock && p->pid==p_locked->pid){ //Unlock when locked.
    p_locked->L = 0;
    p_locked->order = 0; //First order in L0.
    p_locked->runningtime = 0;
    p_locked->priority = 3;
    schedLock = 0;
    // cprintf("Unlock %d\n", p->pid);
  }
  release(&ptable.lock);
}
```

위 두 개의 if문은 스케줄러 락과 동일하다. 아래 if문은 락이 걸려있고 락이 걸린 프로세스가 맞

으면 언락을 하게 만들었다. 이로써 락을 안걸었는데 언락을 호출하는 경우를 방지했다. 언락은 명세에 나와있는 데로 L0의 첫번째 순서로 가고 runningtime을 0으로 초기화해 time quantum을 초기화하고 priority또한 바꾸고 schedLock을 0으로 바꿈으로써 스케줄러에게 언락을 알렸다.

trap.c – schedLock

```
if(tf->trapno == T_IRQ0+IRQ_TIMER && ticks==100){
  if(schedLock){ //If locked, unlock and boosting.
    p_locked->L = 0;
    p_locked->order = 0;
    //boosting
    p_locked->priority = 3;
    p_locked->runningtime = 0;
    schedLock = 0;
    // cprintf("UNLOCK\n");
  }
  boosting();
  ticks = 0;
  // cprintf("boosting\n");
}
```

락을 건 상태에서 100ticks가 되면 명세에 나와있는 데로 L0 첫 번째로 이동하고 boosting을 하는데 0번은 boosting()에서 다루지 않기에, boosting에서 초기화하는 부분을 따로 입력해 주었다.

defs.h

```
//PAGEBREAK: 16
// proc.c
int             cpuid(void);
void            exit(void);
int             fork(void);
int             growproc(int);
int             kill(int);
struct cpu*     mycpu(void);
struct proc*    myproc();
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            setproc(struct proc*);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(void);
void            wakeup(void*);
void            yield(void);
int             getLevel(void);
void            setPriority(int pid, int priority);
void            boosting(void);
void            checkTime(void);
void            schedulerLock(int password);
void            schedulerUnlock(int password);
```

위 사진은 그동안 추가했던 함수들을 defs.h에 추가했음을 보여준다.

c. 시스템 콜 및 인터럽트 추가

다음은 시스템콜에 추가하기 위해 파일 별 추가한 부분이다.

sysproc.c

```
int
sys_yield(void)
{
  yield();
  return 0;
}

int
sys_getLevel(void)
{
  return getLevel();
}

int
sys_setPriority(void)
{
  int pid;
  int priority;

  if(argint(0, &pid) < 0)
    return -1;
  if(argint(0, &priority) < 0)
    return -1;
  setPriority(pid, priority);
  return 0;
}

int
sys_schedulerLock(void)
{
  int password;

  if(argint(0, &password) < 0)
    return -1;
  schedulerLock(password);
  return 0;
}
```

오른쪽 그림처럼 wrapper functon을 만들고 아래처럼 syscall에 추가한 뒤 user가 사용할 수 있도록 user.h, usys.S에 각 내용들을 추가해주었다.

syscall.c

```
extern int sys_yield(void);
extern int sys_getLevel(void);
extern int sys_setPriority(void);
extern int sys_schedulerLock(void);
extern int sys_schedulerUnlock(void);
[SYS_yield]   sys_yield,
[SYS_getLevel]   sys_getLevel,
[SYS_setPriority]   sys_setPriority,
[SYS_schedulerLock]   sys_schedulerLock,
[SYS_schedulerUnlock]   sys_schedulerUnlock,
```

syscall.h

```
#define SYS_yield  23
#define SYS_getLevel 24
#define SYS_setPriority 25
#define SYS_schedulerLock 26
#define SYS_schedulerUnlock 27
```

user.h

```
void yield(void);
int getLevel(void);
void setPriority(int pid, int priority);
void schedulerLock(int password);
void schedulerUnlock(int password);
```

usys.S

```
SYSCALL(yield)
SYSCALL(getLevel)
SYSCALL(setPriority)
SYSCALL(schedulerLock)
SYSCALL(schedulerUnlock)
```

traps.h

```
//project 1
#define T_schedulerLock 129
#define T_schedulerUnlock 130
```

trap.c

```
case T_schedulerLock: //129 interrupt
  schedulerLock(2019008813);
  lapiceoi();
  break;
case T_schedulerUnlock: //130 interrupt
  schedulerUnlock(2019008813);
  lapiceoi();
  break;
```

위는 스케줄러 락/언락을 인터럽트로 구현하기 위해 추가한 부분이다.

d. 스케줄러 수정 (MLFQ 구현)

proc.c – void scheduler(void)

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

  struct proc *temp;
  int catch; //If find the RUNNABLE process in each queue, set 1.
  int least_order; //Least order in each queue
  int L2_least; //Least priority in L2

  for(;;){
loop:
    // Enable interrupts on this processor.
    sti();


    // Loop over process table looking for process to run.
    acquire(&ptable.lock);

    //schedulerLock
    if(schedLock){
      p = p_locked;
      if(p->state == RUNNABLE) //If p's state is RUNNABLE, goto run.
        goto run;

      release(&ptable.lock);
      goto loop; //loop until p's state is RUNNABLE.
    }
```

다음은 scheduler(void)의 가장 윗 부분이다. catch, least_order, L2_least는 앞에서 다른 함수들의 변수와 같은 의미를 지닌다. 기존 xv6의 스케줄러는 루프를 계속 돌게 되는데 이를 반영하였다. 우선 스케줄러의 락이 걸렸는지 확인하고 걸렸으면 그 프로세스가 RUNNABLE하면 run으로 가서 실행시키고 아니면 다시 스케줄러 락이 풀릴 때까지 loop를 돈다.

proc.c – void scheduler(void)

```
//L0
catch = 0;
least_order = 100;
temp = ptable.proc;
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE||p->L!=0) //Check RUNNABLE in L0.
    continue;
  if(p->order<least_order){ //Find least order process.
    temp = p;
    least_order = temp->order;
    catch = 1;
  }
}
p=temp;
if(catch) //If find the process in L0, goto run.
  goto run;
```

다음은 스케줄러의 L0 부분이다. 스케줄러 락이 걸리지 않았다면 처음 마주치는 부분이다. boosting()과 비슷하게 작동한다. L0이며 RUNNABLE한 프로세스 중 가장 작은 순번을 선택한다. 만약 선택해서 catch가 1이라면 run으로 가서 선택한 프로세스를 실행시킨다. 만약 선택하지 못했다면 바로 아래의 L1으로 간다. L1도 라운드 로빈이므로 똑같은 알고리즘으로 작동한다.

proc.c – void scheduler(void)

```
//L2
catch = 0;
L2_least = 4; //priority < 4
least_order = 100; //order < 100
temp = ptable.proc;
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE||p->L!=2) //Check RUNNABLE in L2.
    continue;
  if(p->priority<=L2_least){ //Find least priority.
    L2_least = p->priority;
  }
}
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE||p->L!=2) //Check RUNNABLE in L2.
    continue;
  if(p->priority==L2_least && p->order<least_order){ //Find least order for equal least priority.
    temp = p;
    least_order = temp->order;
    catch = 1;
  }
}
p=temp;
if(catch)
  goto run;

release(&ptable.lock);
goto loop; //If all queues don't have RUNNBLE process goto loop.
```

스케줄러의 L2 부분이다. L1에서도 catch가 0으로 run으로 가지 않았다면 L2로 내려온다. L2는 L0와 L1과는 달리 priority가 작은 것이 빠른 순번보다 먼저 판단되어야 한다. 따라서 위 for문은 L2이며 RUNNABLE한 프로세스 중 가장 작은 priority를 찾는다. 아래 for문은 priority가 같은데 order가 가장 작은 프로세스를 선택한다. 선택했다면 run으로 가고 아니면 loop로 가서 반복한다.

proc.c – void scheduler(void)

```
run:
    // Switch to chosen process.  It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;

    swtch(&(c->scheduler), p->context);
    switchkvm();

    // Process is done running for now.
    // It should have changed its p->state before coming back.
    c->proc = 0;

    release(&ptable.lock);

  }
}
```

다음은 스케줄러의 가장 마지막인 run 부분이다. 기존 xv6의 스케줄러의 부분을 그대로 run으로 지정했다.

e. allocproc 수정

proc.c – static struct proc* allocproc(void)

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  //Additional init when alloc.
  p->L = 0;
  p->order = ++L0_order;
  p->priority = 3;
  p->runningtime = 0;

  release(&ptable.lock);
```

프로세스를 초기화 할 때 추가적으로 코드를 넣어주었다.

3. Result

proc.c – void scheduler(void)

```
p->state = RUNNING;
cprintf("1. pid: %d, p_name: %s, p_level: %d, p_priority: %d, p_order: %d, runningtime: %d, global_ticks: %d\n",
 p->pid, p->name, p->L, p->priority, p->order, p->runningtime, ticks);

swtch(&(c->scheduler), p->context);
switchkvm();
```

스케줄러의 run 부분에 cprintf를 추가하여 스케줄링 마다 pid, level, priority, order, runningtime, global_ticks를 출력하게 했다.

xv6 start

```
1. pid: 1, p_name: initcode, p_level: 0, p_priority: 3, p_order: 7, runningtime: 0, global_ticks: 5
1. pid: 1, p_name: initcode, p_level: 0, p_priority: 3, p_order: 8, runningtime: 1, global_ticks: 6
1. pid: 1, p_name: initcode, p_level: 0, p_priority: 3, p_order: 9, runningtime: 2, global_ticks: 7
1. pid: 1, p_name: initcode, p_level: 0, p_priority: 3, p_order: 10, runningtime: 3, global_ticks: 8
1. pid: 1, p_name: initcode, p_level: 1, p_priority: 3, p_order: 1, runningtime: 0, global_ticks: 9
1. pid: 1, p_name: initcode, p_level: 1, p_priority: 3, p_order: 2, runningtime: 1, global_ticks: 10
1. pid: 1, p_name: initcode, p_level: 1, p_priority: 3, p_order: 3, runningtime: 2, global_ticks: 11
1. pid: 1, p_name: initcode, p_level: 1, p_priority: 3, p_order: 4, runningtime: 3, global_ticks: 12
1. pid: 1, p_name: initcode, p_level: 1, p_priority: 3, p_order: 5, runningtime: 4, global_ticks: 13
1. pid: 1, p_name: initcode, p_level: 1, p_priority: 3, p_order: 6, runningtime: 5, global_ticks: 14
1. pid: 1, p_name: initcode, p_level: 2, p_priority: 3, p_order: 1, runningtime: 0, global_ticks: 15
```

위 사진은 xv6를 시작할 때 터미널에 나오는 문장들이다. L0~L2까지 프로세스가 runningtime에 맞게 이동하는 모습을 확인할 수 있다.

xv6 start

```
pid: 2, p_name: init, p_level: 2, p_priority: 3, p_order: 3, runningtime: 0, global_ticks: 67
pid: 2, p_name: init, p_level: 2, p_priority: 3, p_order: 3, runningtime: 1, global_ticks: 68
pid: 2, p_name: init, p_level: 2, p_priority: 3, p_order: 3, runningtime: 2, global_ticks: 69
pid: 2, p_name: init, p_level: 2, p_priority: 3, p_order: 3, runningtime: 3, global_ticks: 70
pid: 2, p_name: init, p_level: 2, p_priority: 3, p_order: 3, runningtime: 4, global_ticks: 71
pid: 2, p_name: init, p_level: 2, p_priority: 3, p_order: 3, runningtime: 5, global_ticks: 72
pid: 2, p_name: init, p_level: 2, p_priority: 3, p_order: 3, runningtime: 6, global_ticks: 73
pid: 2, p_name: init, p_level: 2, p_priority: 3, p_order: 3, runningtime: 7, global_ticks: 74
pid: 2, p_name: init, p_level: 2, p_priority: 2, p_order: 3, runningtime: 8, global_ticks: 75
pid: 2, p_name: init, p_level: 2, p_priority: 2, p_order: 3, runningtime: 9, global_ticks: 76
pid: 2, p_name: init, p_level: 2, p_priority: 2, p_order: 3, runningtime: 10, global_ticks: 77
pid: 2, p_name: init, p_level: 2, p_priority: 2, p_order: 3, runningtime: 14, global_ticks: 81
pid: 2, p_name: init, p_level: 2, p_priority: 2, p_order: 3, runningtime: 15, global_ticks: 82
pid: 2, p_name: init, p_level: 2, p_priority: 1, p_order: 3, runningtime: 16, global_ticks: 83
pid: 2, p_name: init, p_level: 2, p_priority: 1, p_order: 3, runningtime: 17, global_ticks: 84
pid: 2, p_name: init, p_level: 2, p_priority: 1, p_order: 3, runningtime: 22, global_ticks: 89
pid: 2, p_name: init, p_level: 2, p_priority: 1, p_order: 3, runningtime: 23, global_ticks: 90
pid: 2, p_name: init, p_level: 2, p_priority: 0, p_order: 3, runningtime: 24, global_ticks: 91
pid: 3, p_name: sh, p_level: 2, p_priority: 0, p_order: 1, runningtime: 30, global_ticks: 46
pid: 3, p_name: sh, p_level: 2, p_priority: 0, p_order: 1, runningtime: 31, global_ticks: 47
pid: 3, p_name: sh, p_level: 2, p_priority: 0, p_order: 1, runningtime: 32, global_ticks: 48
pid: 3, p_name: sh, p_level: 2, p_priority: 0, p_order: 1, runningtime: 33, global_ticks: 49
```

위 사진에서 L2에서 8ticks마다 정상적으로 priority가 높아지는 지 확인할 수 있다. 마지막 사진에선 runningtime이 32에서 더 이상 priority가 0에서 낮아지지 않는 모습을 확인할 수 있다.

xv6 start

```
pid: 2, p_name: init, p_level: 0, p_priority: 3, p_order: 36, runningtime: 2, global_ticks: 99
pid: 2, p_name: init, p_level: 0, p_priority: 3, p_order: 1, runningtime: 0, global_ticks: 0
pid: 2, p_name: init, p_level: 0, p_priority: 3, p_order: 2, runningtime: 1, global_ticks: 1
```

다음은 boosting이 정상적으로 작동하는 사진이다.

앞의 내용은 프로세스가 1개일 때의 경우로 mlfq_test를 통해 2개 이상일 때 스케줄러가 정상 작동하는지 mlfq_test.c를 통해 알아 보았다.

mlfq_test

```
pid: 22, p_name: mlfq_test, p_level: 0, p_priority: 3, p_order: 1, runningtime: 0, global_ticks: 1
pid: 23, p_name: mlfq_test, p_level: 0, p_priority: 3, p_order: 2, runningtime: 0, global_ticks: 2
pid: 24, p_name: mlfq_test, p_level: 0, p_priority: 3, p_order: 3, runningtime: 0, global_ticks: 3
pid: 22, p_name: mlfq_test, p_level: 0, p_priority: 3, p_order: 4, runningtime: 1, global_ticks: 4
pid: 23, p_name: mlfq_test, p_level: 0, p_priority: 3, p_order: 5, runningtime: 1, global_ticks: 5
pid: 24, p_name: mlfq_test, p_level: 0, p_priority: 3, p_order: 6, runningtime: 1, global_ticks: 6
pid: 22, p_name: mlfq_test, p_level: 0, p_priority: 3, p_order: 7, runningtime: 2, global_ticks: 7
pid: 23, p_name: mlfq_test, p_level: 0, p_priority: 3, p_order: 8, runningtime: 2, global_ticks: 8
pid: 24, p_name: mlfq_test, p_level: 0, p_priority: 3, p_order: 9, runningtime: 2, global_ticks: 9
pid: 22, p_name: mlfq_test, p_level: 0, p_priority: 3, p_order: 10, runningtime: 3, global_ticks: 10
pid: 23, p_name: mlfq_test, p_level: 0, p_priority: 3, p_order: 11, runningtime: 3, global_ticks: 11
pid: 24, p_name: mlfq_test, p_level: 0, p_priority: 3, p_order: 12, runningtime: 3, global_ticks: 12
pid: 22, p_name: mlfq_test, p_level: 1, p_priority: 3, p_order: 1, runningtime: 0, global_ticks: 14
pid: 23, p_name: mlfq_test, p_level: 1, p_priority: 3, p_order: 2, runningtime: 0, global_ticks: 15
pid: 24, p_name: mlfq_test, p_level: 1, p_priority: 3, p_order: 3, runningtime: 0, global_ticks: 16
pid: 22, p_name: mlfq_test, p_level: 1, p_priority: 3, p_order: 4, runningtime: 1, global_ticks: 17
pid: 23, p_name: mlfq_test, p_level: 1, p_priority: 3, p_order: 5, runningtime: 1, global_ticks: 18
pid: 24, p_name: mlfq_test, p_level: 1, p_priority: 3, p_order: 6, runningtime: 1, global_ticks: 19
pid: 22, p_name: mlfq_test, p_level: 1, p_priority: 3, p_order: 7, runningtime: 2, global_ticks: 20
pid: 23, p_name: mlfq_test, p_level: 1, p_priority: 3, p_order: 8, runningtime: 2, global_ticks: 21
pid: 24, p_name: mlfq_test, p_level: 1, p_priority: 3, p_order: 9, runningtime: 2, global_ticks: 22
```

22, 23, 24 프로세스들이 정상적으로 1tick이 지날 때마다 다음 순번의 프로세스에게 cpu를 넘겨주는 모습을 볼 수 있다. 또한 각 프로세스 별로 runningtime이 4가 되면 L1으로 degrade된다.

mlfq_test

```
pid: 23, p_name: mlfq_test, p_level: 2, p_priority: 0, p_order: 1, runningtime: 512, global_ticks: 98
pid: 23, p_name: mlfq_test, p_level: 2, p_priority: 0, p_order: 1, runningtime: 513, global_ticks: 99
pid: 24, p_name: mlfq_test, p_level: 0, p_priority: 3, p_order: 1, runningtime: 0, global_ticks: 1
pid: 24, p_name: mlfq_test, p_level: 0, p_priority: 3, p_order: 2, runningtime: 1, global_ticks: 2
```

boosting도 잘 작동되는 모습이다.

mlfq_test

```
SeaBIOS (version 1.15.0-1)


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 start 58
init: starting sh
$ mlfq_test
MLFQ test start
[Test 1] default
Process 6
L0: 949
L1: 1066
L2: 97985
L3: 0
L4: 0
Process 7
L0: 7157
L1: 10513
L2: 82330
L3: 0
L4: 0
Process 4
L0: 10692
L1: 15593
L2: 73715
L3: 0
L4: 0
Process 5
L0: 13935
L1: 21432
L2: 64633
L3: 0
L4: 0
[Test 1] finished
done
$
```

다음은 test코드 결과이다.

4. Trouble shooting

a. SLEEP상태였다가 RUNNABLE로 바뀌는 부분

SLEEP상태였다가 RUNNABLE로 바뀌면 ready 큐에서 나갔다가 들어온 것으로 판단하고 다시 L0로 넣어줘야 하는데 처음에 그렇지 않아서 문제가 생겼었다. 따라서 다음의 wake와 kill함수에 RUNNABLE로 바뀌는 구간에 코드를 추가해 주었다.

proc.c – static void wakeup1(void *chan)

```
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan){
      p->state = RUNNABLE;
      //If p's state changed to RUNNABLE, enter the queue again.
      if(!schedLock){ //If locked, do not go to L0. Just RUNNABLE.
        p->L = 0;
        p->runningtime = 0;
        p->priority = 3;
        p->order = ++L0_order;
      }

    }
}
```

proc.c – int kill(int pid)

```
int
kill(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING){
        p->state = RUNNABLE;
        //If p is RUNNABLE, enter the queue again.
        if(!schedLock){ //If locked, do not go to L0. Just RUNNABLE.
          p->L = 0;
          p->runningtime = 0;
          p->priority = 3;
          p->order = ++L0_order;
        }
      }
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

b. boosting이 제대로 안됨.

mlfq_test

```
pid: 24, p_name: mlfq_test, p_level: 2, p_priority: 0, p_order: 1, runningtime: 1729, global_ticks: 99
pid: 24, p_name: mlfq_test, p_level: 2, p_priority: 0, p_order: 1, runningtime: 1730, global_ticks: 1
pid: 24, p_name: mlfq_test, p_level: 2, p_priority: 0, p_order: 1, runningtime: 1731, global_ticks: 2
pid: 24, p_name: mlfq_test, p_level: 2, p_priority: 0, p_order: 1, runningtime: 1732, global_ticks: 3
```

프로세스가 하나만 있을 때 항상 RUNNING 상태여서 boosting이 제대로 안되는 것이라 판단하고 boosting 대상을 RUNNING도 포함하였다.

proc.c – void boosting(void)

```
boosting(void){ //Call when ticks == 100.
  acquire(&ptable.lock);
  struct proc *p;
  int least_order;
  int catch;
  struct proc *temp;

  //L0 init
  int L0_order_before = L0_order; //Remember last process order.
  L0_order = 0; //init L0_order to prevent overflow.
L0:
  least_order = 100;
  catch = 0;
  temp = ptable.proc;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->L!=0 || ((p->state != RUNNABLE)&&(p->state != RUNNING))) //p is in L0 and RUNNABLE. or RUNNIN
      continue;
    if(L0_order<p->order && p->order<least_order){ //Find least order process, but not boosting.
      temp = p;
      least_order = temp->order;
      catch = 1;
    }
  }
  p = temp; //Set choosed process.
  if(p->order == L0_order_before && catch){ //Last boosting.
      p->order = ++L0_order;
      p->priority = 3;
      p->runningtime = 0;
      catch = 0;
    }
  if(catch){ //No last boosting.
    p->order = ++L0_order;
    p->priority = 3;
    p->runningtime = 0;
    goto L0;
  }
```

또한 마지막에서 2번째 if문에서 catch할 때만 마지막 부스팅으로 판단하게 수정하였다.