Project02 wiki

컴퓨터소프트웨어학부

2016025141

고세진

디자인

각 스케줄러를 어떻게 구현하느지, 추가 변경되는 컴포넌트가 어떻게 상호작용, 어떤 자료구조가 필요한자

FCFS SCHEDULER

명세

1. 먼저 생성된 프로세스가 먼저 스케줄링 되어야 한다
2. 프로세스는 종료 전까지 switch out 되지 않는다.
3. 200ticks가 자닐 때까지 종료되거나 SLEEPING 상태가 아니라면 강제로 종료시킨다.
4. SLEEPING 상태로 전환된다면 다음으로 생성된 프로세스가 스케줄링 된다.

디자인

1. 프로세스의 pid가 작을수록 먼저 생성된 프로세스이다. 따라서, pid가 작은 프로세스를 먼저 스케줄링한다.
2. 프로세스가 처음 스케줄 된 ticks를 기록하기 위한 변수를 proc 자료구조에 추가한다.
3. 해당 변수에 프로세스가 처음 스케줄링된 ticks를 기록하고, 타임 인터럽트 발생 시점에서 200ticks이 지났는지 여부를 확인한다.
4. 200ticks이 지났다면 프로세스를 exit()함수를 이용해 프로세스를 종료한다.

MULTILEVEL SCHEDULER

명세

1. Round Robin 방식, FCFS 방식의 큐가 각각 존재한다. pid가 짝수인 프로세스는 Round Robin에서 스케줄링하고 pid가 홀수인 프로세스는 FCFS에서 스케줄링한다.

2. Round Robin 큐가 우선순위가 더 높아야 한다.
3. Round Robin 큐에 RUNNABLE 프로세스가 없으면 FCFS 큐에서 스케줄링한다.

디자인

1. 전체 프로세스 테이블을 순회하면서 pid가 짝수인 프로세스를 라운드 로빈 방식으로 스케줄링 한다.
2. 한번 순회가 끝난 후, 테이블을 한번 더 순회하면서 짝수 PID를 가진 프로세스가 RUNNABLE하면 goto를 이용해 1번을 반복하고 없다면 이용해 3번으로 간다.
3. 전체 테이블을 순회하면서 홀수 PID를 가진 RUNNABLE 프로세스를 스케줄링한다. 만약 테이블 순회 중 짝수 PID를 가진 RUNNABLE 프로세스가 발견되면 1번으로 돌아간다.

MULTILEVEL FEEDBACK QUEUE SCHEDULER

명세

1. L0, L1의 두 가지 큐가 존재하며, 각각 4ticks, 8ticks의 time quantum을 가진다.
2. 프로세스는 각 레벨에서 주어진 quantum동안 실행한 후에 switch out된다.
3. L0 큐에서 실행된 프로세스가 quantum을 모두 사용한 경우 priority 값이 1 감소한다.
4. 200 ticks 마다 모든 프로세스들이 L0큐로 올라가고, priority가 0으로 리셋된다.

디자인

1. proc 자료구조에 프로세스가 각 레벨에서 가지고 있는 time quantum, 현재 위치한 queue의 level을 저장하는 변수를 작성한다.
2. L0 큐에서 프로세스 자료구조의 레벨 변수값이 0인 프로세스를 Round Robin 방식으로 4 ticks 실행한 후 프로세스 자료구조의 레벨 변수를 1로 변경하는 방식으로 L1큐로 프로세스를 보낸다.
3. 큐 레벨이 0이면서 SLEEPING 상태가 아닌 프로세스가 더이상 없다면 L1 큐로 점프한다.
4. L1 큐는 priority당 여러 프로세스의 포인터를 저장할 수 있는 구조체로 다룬다.
5. priority가 높은 순으로 반복문을 시작하고, 해당 priority의 프로세스 중 pid가 가장 작은 프로세스를 먼저 스케줄링해 8ticks동안 실행한다.
6. 해당 priority에서 실행 가능한 모든 프로세스를 실행한 후, 낮은 우선순위의 프로세스들을 FCFS로 실행한다.

7. L0큐가 먼저 실행되므로, L0 큐의 루프 앞부분에서 ticks%200 == 0인 경우 모든 프로세스의 큐 레벨을 0으로 설정하고 우선순위를 0으로 만들고, L1구조체의 인덱스를 초기화한다.

SYSTEM CALLS

1. getlev : proc 구조체에 현재 queue level을 저장할 수 있는 변수를 생성하고, myproc()함수로 접근하여 리턴한다
2. monopolize: 학번을를 받아 cpu를 독점할 수 있도록 한다.
3. setpriority: proc 구조체의 priority 변수를 수정하고, L1 큐 구조체에서 해당 프로세를 다른 priority에 해당하는 배열로 옮겨준다.
    - 성공시 0, 존재하지 않는 pid의 경우 -1, priority가 범위를 벗어난 경우 -2 리턴

구현

FCFS SCHEDULER

— proc.h 의 proc 구조체에 에 처음으로 스케줄링 된 틱을 기록하는 변수를 추가하였음

```
struct proc {
...
#ifdef FCFS_SCHED
  uint tick_first_scheduled;
#endif
...
};
```

— proc.c 의 scheduler 함수에서 minimum pid를 가진 프로세스를 가장 먼저 스케줄링하고, 처음 스케줄링된 시점의 ticks를 프로세스 구조체에 지정

```
void
scheduler(void)
{
...

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

    struct proc *nextproc = 0;
    acquire(&ptable.lock);
```

```
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      if(nextproc == 0){
        nextproc = p;
      }else if(nextproc->pid > p->pid){
         nextproc = p;
      }
    }
  if(nextproc == 0){
    release(&ptable.lock);
    continue;
  }
  if(nextproc->tick_first_scheduled == 0)
    nextproc->tick_first_scheduled = ticks;
  c->proc = nextproc;
 }
}
...
```

— 프로세스가 200틱이 지난 시점에 종료하기 위하여, 타임 인터럽트 발생시 프로세스가 스케줄링된 후 지난 ticks을 체크해서 200틱 이상 지난 경우 exit()을 이용하여 종료시킨다. init, sh 프로세스는 예외처리 해준다.

```
#ifdef FCFS_SCHED
 // Force process exit if a process not termintated or slept
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER &&
   myproc()->pid != 1 &&
   myproc()->pid != 2 &&
     ticks - myproc()->tick_first_scheduled > 200
    && (tf->cs&3) == DPL_USER)
    exit();
#endif
```

MULTILEVEL SCHEDULER

1) 전체 프로세스 테이블을 순회하면서 pid가 짝수인 프로세스를 라운드 로빈 방식으로 스케줄링 한다.

2) 한번 순회가 끝난 후 RUNNABLE하면서 짝수 PID를 가진 프로세스가 있으면 goto를 이용해 다시 라운드 로빈 큐로 간다.

3) 짝수 PID를 가진 프로세스가 RUNNABLE하지 않으면 FCFS큐를 이용해 스케줄링한다. 다만, 홀수 PID를 가진 프로세스 탐색 중 RUNNABLE한 짝수 PID를 가진 프로세스를 발견하면 Round Robin 스케줄러로 점프한다.

```
//1)
rr:
```

```
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
          continue;
        if( p->pid % 2 == 1)
          continue;
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&(c->scheduler), p->context);
      switchkvm();
      c->proc = 0;
    }
//2)
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
          continue;
        if( p->pid % 2 == 1)
          continue;
      goto rr;
    }

...
struct proc *nextproc = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      if(p->pid%2 == 0)
        goto rr; //pid가 짝수인 프로세스가 있다면 점프한다.

      if(nextproc == 0){
        nextproc = p;
      }else if(nextproc->pid > p->pid){
        nextproc = p;
      }
    }
...
```

MULTILEVEL FEEDBACK QUEUE SCHEDULER

1) L1큐를 구현하기 위한 구조체. priority당 6400개의 프로세스 포인터를 저장할 수 있게 구현하였음

```
#elif MLFQ_SCHED
struct L1 {
    struct proc* queue[NPROC*100][10];
    int idx[10];
};
struct L1 l1;
```

2) 200ticks당 부스팅

```
if(ticks %200 == 0 && boosted == 0){
      boosted = 1;
      //cprintf("boostring!\n");
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if (p->state == UNUSED)
          continue;
        p->queue_level=0;
        p->priority=0;
      }
      for (i=0; i< 10; i++){
        l1.idx[i] = -1;
      }
    }
    if(ticks %200 != 0)
      boosted = 0;
```

3) L0 queue

```
...
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->queue_level !=0){
        continue;
      }
      if(p->state != RUNNABLE){
        continue;
      }
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      if(p->pid != 1 && p->pid != 2){
        p->quantum_level_0--;
        if(p->quantum_level_0 == 0)
          p->queue_level = 1;
      }
      swtch(&(c->scheduler), p->context);
      switchkvm();
      c->proc = 0;
    }
    release(&ptable.lock);

    //L1 큐로 점프하는 플래그. L0큐에 실행 가능한 프로세스가 있다면 0으로 변경
    int flag_q1 = 1;
    acquire(&ptable.lock);

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state == UNUSED){
        continue;
      }
      else if(p->queue_level == 0 && p->state != SLEEPING){
        flag_q1 = 0;
        break;
      }
    }
```

```
    release(&ptable.lock);
    if(flag_q1 == 0)
      continue;
    else
      goto level1;
...
```

4) L1 queue

```
//프로세스를 L1큐에 priority를 이용하여 저장
...
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->queue_level != 1)
      continue;
    if(p->state != RUNNABLE)
      continue;

      //store procs by priority
    l1.idx[p->priority]++;
    l1.queue[l1.idx[p->priority]][p->priority] = p;
  }
...

//우선순위가 높은 순으로 L1큐에서 프로세스를 꺼내어 FCFS 스케줄링
int pri, index;
  for(pri = 9; pri >= 0; pri--){
    //fcfs

    for(;;){
      struct proc *nextproc = 0;
      struct proc *p = 0 ;
      //find proc with minpid

      for(index = 0; index <= l1.idx[pri]; index++){
        p = l1.queue[index][pri];
        if(p->pid == 0){
          continue;
        }
        if(p->state != RUNNABLE){

         continue;
        }
        else if(p->quantum_level_1 == 0)
          continue;
        if(nextproc == 0 || nextproc->pid > p->pid){
          nextproc = p;

        }
      }
      //no runnable proc in this priority
      if(nextproc == 0)
        break;

      nextproc->quantum_level_1--;
      c->proc = nextproc;
```

```
      switchuvm(nextproc);
      nextproc->state = RUNNING;
      swtch(&(c->scheduler), nextproc->context);
      switchkvm();
      c->proc = 0;
    }
  }
...
//time quantum을 모두 소비한 경우 proirity를 1 낮춰주고 quantum을 다시 8로 설정해준다.
for(index = 0; index < l1.idx[0];index++){
   l1.queue[index][0] -> quantum_level_1 = 8;
  }

  for(pri = 1; pri <= 9; pri++){
    for(index = 0; index < l1.idx[pri]; index++){
      struct proc * p = l1.queue[index][pri];
      if(p->quantum_level_1 == 0){
        setpriority(p->pid, p->priority -1);
        p->quantum_level_1 = 8;
      }

    }
  }
```

system calls

```
int
getlev(void)
{
  return myproc()->queue_level;
}
int
setpriority(int pid, int priority)
{

  if(priority < 0 || priority > 10)
    return -2;
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      l1.queue[l1.idx[p->priority]][p->priority] = 0;
      p->priority = priority;
      l1.idx[p->priority]++;
      l1.queue[l1.idx[p->priority]][p->priority] = p;
     release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
```

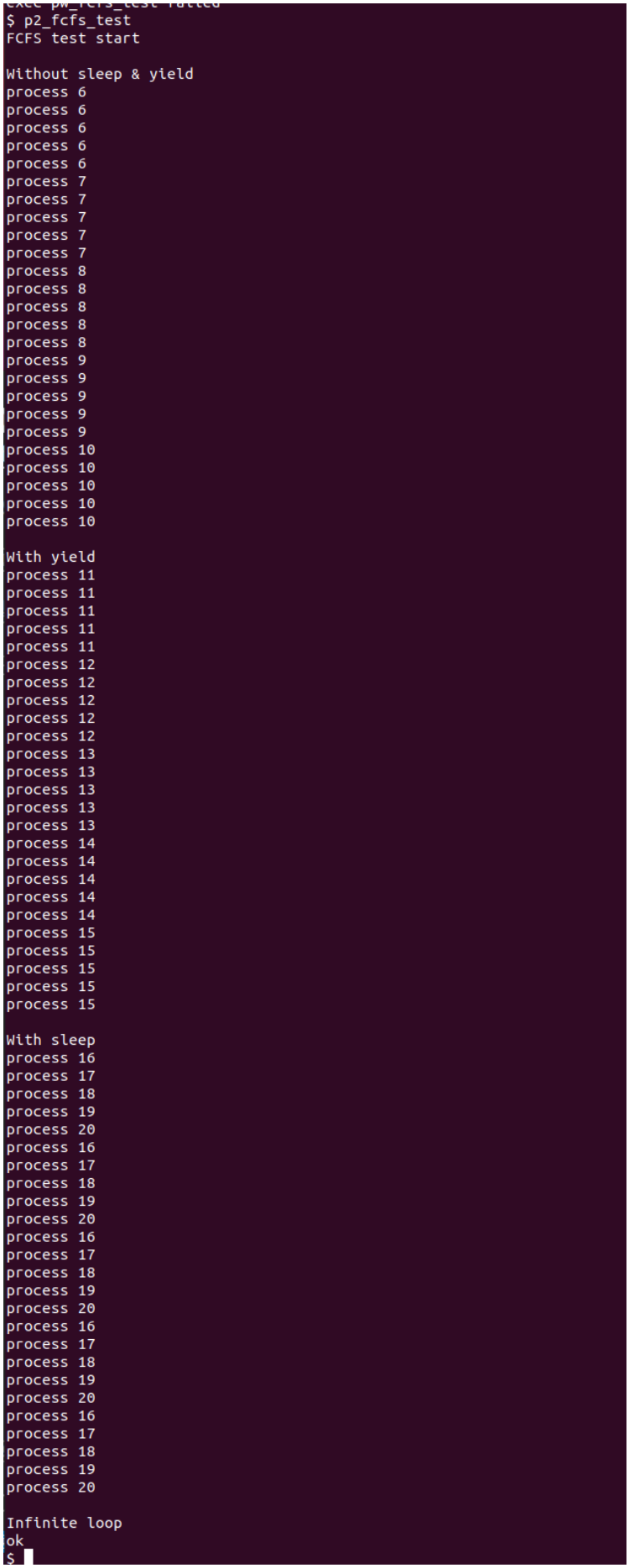
```
}

void
monopolize(int password)
{
  struct proc* p = myproc();
  if(password != 2016025141)
    p->killed = 1;
  if(p->monopolized == 1);
  else if (p->monopolized == 0);

}
```

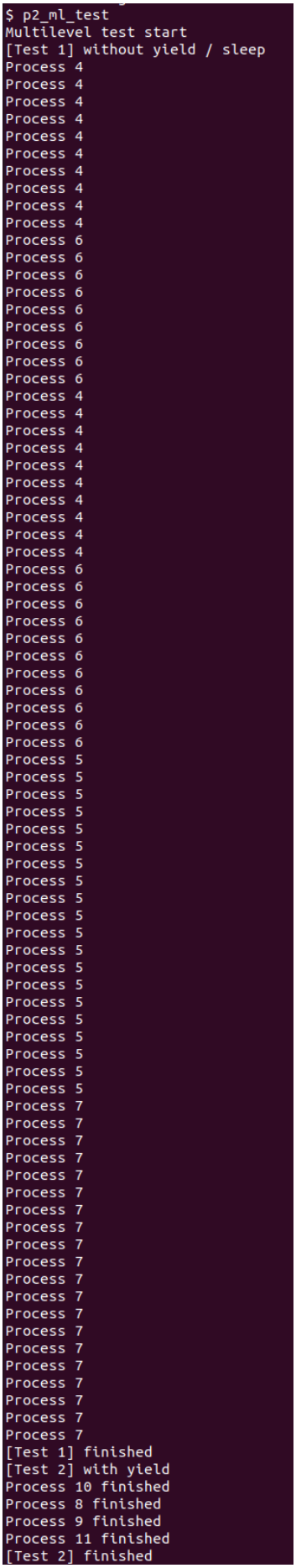
실행 결과

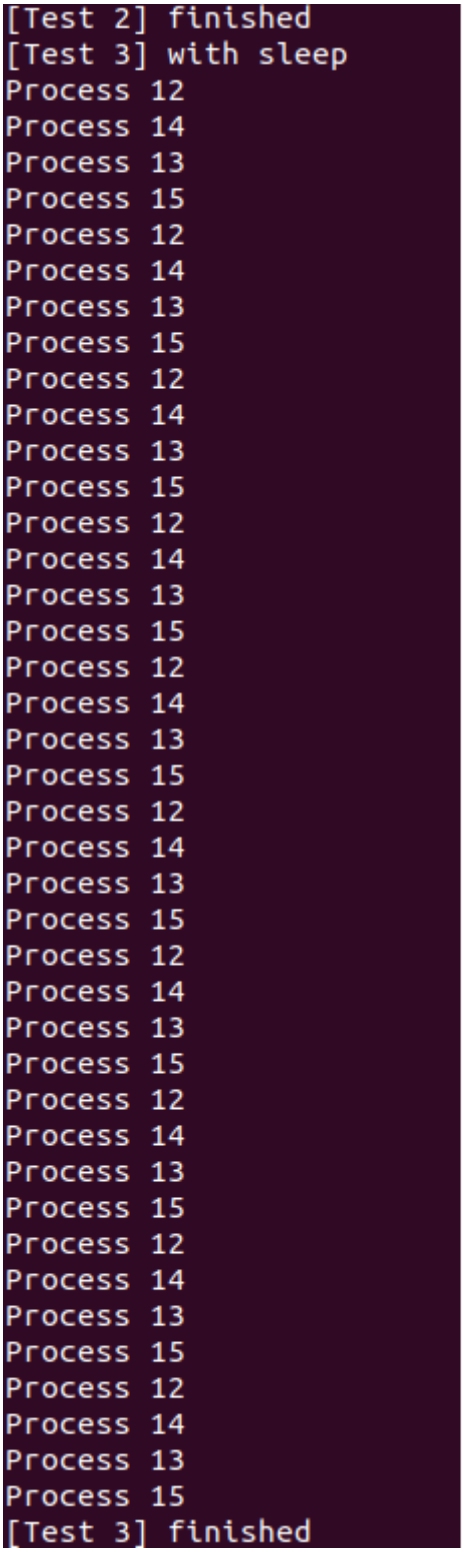
FCFS SCHED


$ p2_fcfs_test
FCFS test start
Without sleep & yield
process 6
process 6
process 6
process 6
process 6
process 7
process 7
process 7
process 7
process 7
process 8
process 8
process 8
process 8
process 8
process 9
process 9
process 9
process 9
process 9
process 10
process 10
process 10
process 10
process 10
With yield
process 11
process 11
process 11
process 11
process 11
process 12
process 12
process 12
process 12
process 12
process 13
process 13
process 13
process 13
process 13
process 14
process 14
process 14
process 14
process 14
process 15
process 15
process 15
process 15
process 15
With sleep
process 16
process 17
process 18
process 19
process 20
process 16
process 17
process 18
process 19
process 20
process 16
process 17
process 18
process 19
process 20
process 16
process 17
process 18
process 19
process 20
process 16
process 17
process 18
process 19
process 20
Infinite loop
ok
$

1. yield를 적용하지 않은 경우 FCFS 스케줄러이기 때문에 프로세스가 순서대로 끝이 났다. yield를 적용한 경우에도 자식 프로세스들이 순서대로 yield를 병렬 호출하기 때문에 결국 처음 만들어진 프로세스가 가장 먼저 실행되고 끝나게 된다.
2. 슬립을 적용한 경우 프로세스 state가 SLEEPING 상태로 전환되어 pid가 작아도 스케줄링 대상이 되지 않는다. 다만 pid가 가장 작은 자식프로세스가 먼저 sleeping 상태가 되기 때문에 wakeup도 빠르게 하여 먼저 끝난다.
3. 프로세스당 200ticks가 지나면 타임 인터럽트를 핸들링하는 과정에서 process를 exit()한다.

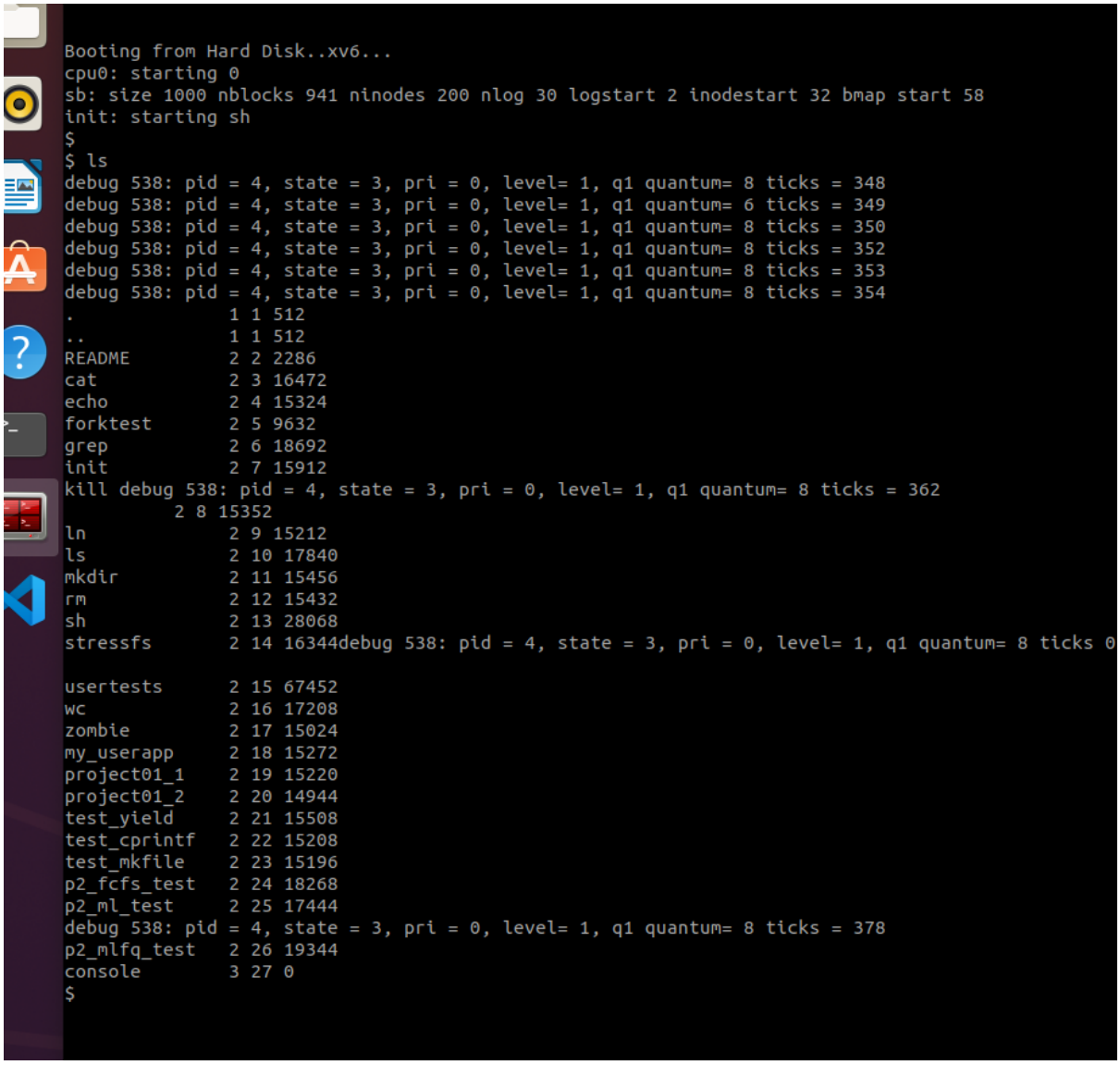
MULTILEVEL SCHED

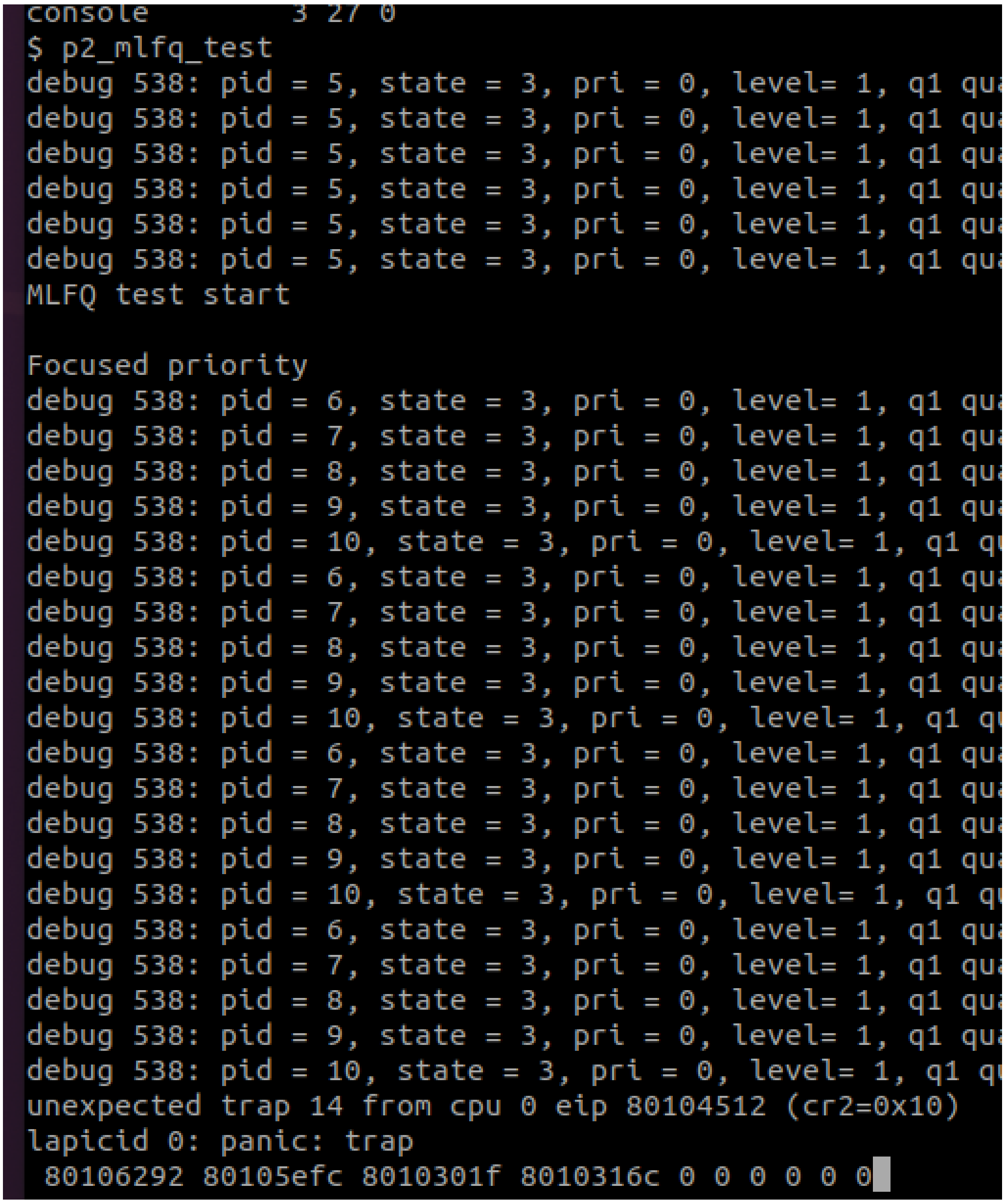

$ p2_ml_test
Multilevel test start
[Test 1] without yield / sleep
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
[Test 1] finished
[Test 2] with yield
Process 10 finished
Process 8 finished
Process 9 finished
Process 11 finished
[Test 2] finished


[Test 2] finished
[Test 3] with sleep
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
[Test 3] finished

짝수 pid를 가진 프로세스가 항상 먼저 실행되고 완료된다.

MLFQ





정상적으로 구동이 되고 ls등의 유저프로그램 실행이 가능하나 되나 테스트 프로그램 실행 시 trap panic에러 발생. setprioirty 시스템콜 내부에서 L1큐 구조체 동작 구현이 완전하지 않아 발생하는 현상으로 추정됨.

트러블슈팅

MLFQ의 boosting 구현 중, 한 틱당 scheduler의 loop가 여러번 돌아서 ticks%200 == 0 인 경우가 여러번 발생해 부스팅이 여러번 되는 경우가 발생하여 boosted 변수를 두고 부스팅이 된 경우 1로 바꿔준 후, ticks%200 ≠ 인 경우에 다시 0으로 설정하는 방법으로 부스팅이 한번만 되도록 하였음

```
if(ticks %200 == 0 && boosted == 0){
      boosted = 1;
      //cprintf("boostring!\n");
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if (p->state == UNUSED)
          continue;
        p->queue_level=0;
        p->priority=0;
      }
      for (i=0; i< 10; i++){
        l1.idx[i] = -1;
      }
    }
if(ticks %200 != 0)
  boosted = 0;
```