OS Project02 Wiki

Author: 2020021949 구건모

이 프로젝트는 운영체제(ELE3021-11742)과목 Project02에 관한 문서입니다.
이 문서는 다음 네 가지 정보를 담고 있습니다.

- Desin
- Implement
- Result
- Trouble shooting

1. Design

이 프로젝트에서는 다음 세 가지 feature를 구현하였습니다.

1. 과제 명세에 나온 Scheduling 구현 (L0, L1, L2, L3, Monopoly)
2. Scheduling을 관리하는 System Call 구현 (priority 변경, 큐 모드 변경 등)
3. 이를 테스트해볼 수 있는 유저 프로그램 작성

Scheduling

스케쥴링을 위해 필요한 자료구조는 다음과 같습니다.

- `Queue` : (L0, L1, L2, Monopoly)
- `Heap` : (L3)

`ptable`에 할당되어 있는 프로세스를 `Queue` 혹은 `Heap`에서 관리하면서 우선순위가 높은 프로세스를 먼저 실행하도록 스케쥴링을 구현하면 됩니다.

System Call for Handling Scheduling

스케쥴링을 관리하기 위한 시스템 콜은 다음과 같습니다.

- `yield`: 명시적으로 CPU 반납
- `getlev`: 현재 프로세스의 레벨 반환 (L0, L1, L2, L3, Monopoly)
- `setpriority`: 프로세스의 우선순위 변경 (L3에서만 의미있는 값)
- `setmonopoly`: 해당 프로세스를 MoQ로 이동
- `monopolize`: 현재 CPU 실행 모드를 Monopoly로 변경
- `unmonopolize`: 현재 CPU 실행 모드를 Monopoly에서 일반 모드로 변경

How to implement?

먼저 스케쥴링 방법을 변경해야 합니다. 과제 명세에서 제공하는 스케쥴링은 다음과 같이 디자인할 수 있습니다.

1. `Queue`와 `Heap`을 구현합니다.
2. 새로운 프로세스를 할당하거나 잠들어있는 프로세스가 깨어날 때 `Queue` 혹은 `Heap`에 삽입하는 코드를 작성합니다.
3. `timer interrupt`가 발생할 때마다 CPU를 반납할 것인지, 아니면 계속 진행할 것인지 결정하는 코드를 작성합니다.

4. CPU 반납 시 Feedback을 받아 우선순위를 변경할 수 있도록 코드를 작성합니다.
5. CPU 반납 외에도 `priority_boosting` 등 글로벌 틱에 따라 우선순위를 변경할 수 있도록 코드를 작성합니다.

스케쥴링 코드를 작성했다면 다음과 같이 유저 테스트 코드를 작성할 수 있습니다. (시스템 콜을 등록하는 부분은 Project01에 나와있으므로 생략했습니다.)

1. 여러 프로세스를 생성하고 각 프로세스가 특정 레벨 큐에 머무를 수 있도록 코드를 작성합니다.
2. 해당 프로세스가 특정 레벨에 머무는지, 또, 스케쥴링은 잘 되는지 확인할 수 있는 코드를 작성합니다.
3. 실행 결과를 보고 스케쥴링이 잘 되는지 확인합니다.

2. Implement

먼저 기존 스케쥴링이 어떻게 동작하는지 분석했습니다.

(origin) scheduler

초기에 제공하는 xv6의 스케쥴러는 다음과 같습니다.

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

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

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // 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);
```

```
    }
  }
```

`ptable`에 있는 모든 프로세스를 순회하며 `RUNNABLE` 상태인 프로세스를 찾아 실행하는 코드입니다.

`swtch` 함수를 통해 실행 가능한 프로세스로 context switching을 하고, 다시 scheduler로 context switching이 되면, 다음 프로세스를 순차적으로 찾아 실행하는 구조입니다.

그렇다면, 어떤 상황에서 다시 스케줄러로 context switching을 하는지 알아보겠습니다.

(origin) trap

특정 프로세스가 CPU를 언제 반납하는지 알기 위해서는 `trap` 함수를 분석해야 합니다.

```
//FILE: trap.c
void
trap(struct trapframe *tf)
{
  struct proc* p;

  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE+1:
    // Bochs generates spurious IDE1 interrupts.
    break;
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_COM1:
```

```
      uartintr();
      lapiceoi();
      break;
    case T_IRQ0 + 7:
    case T_IRQ0 + IRQ_SPURIOUS:
      cprintf("cpu%d: spurious interrupt at %x:%x\n",
              cpuid(), tf->cs, tf->eip);
      lapiceoi();
      break;

    //PAGEBREAK: 13
    default:
      if(myproc() == 0 || (tf->cs&3) == 0){
        // In kernel, it must be our mistake.
        cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
                tf->trapno, cpuid(), tf->eip, rcr2());
        panic("trap");
      }
      // In user space, assume process misbehaved.
      cprintf("pid %d %s: trap %d err %d on cpu %d "
              "eip 0x%x addr 0x%x--kill proc\n",
              myproc()->pid, myproc()->name, tf->trapno,
              tf->err, cpuid(), tf->eip, rcr2());
      myproc()->killed = 1;
    }

    // Force process exit if it has been killed and is in user space.
    // (If it is still executing in the kernel, let it keep running
    // until it gets to the regular system call return.)
    if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
      exit();

    if (myproc() && myproc()->state == RUNNING &&
       tf->trapno == T_IRQ0 + IRQ_TIMER) {
      yield();
    }


    // Check if the process has been killed since we yielded
    if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
      exit();
  }
```

`lapic.c`를 확인해보면, 약 10ms마다 타이머 인터럽트가 발생하며, 해당 타이머 인터럽트는 트랩 코드에서 캐치합니다. 유저 프로세스를 실행하는 도중 인터럽트가 발생하면 trap함수에서 인터럽트를 처리하게 되고, 만약 해당 인터럽트가 타이머 인터럽트이면서 실행 중인 프로세스에서 발생했다면, `yield` 함수를 호출합니다.

따라서, 유저 프로세스는 약 10ms마다 발생하는 타이머 인터럽트가 발생했을 때 CPU를 반납합니다.

따라서, 기존 xv6에서 유저 프로세스는 10ms 마다 CPU를 반납하며, 스케쥴러로 돌아가 다음 실행 가능한 프로세스를 순차적으로 실행하는 Round Robin 스케쥴링을 구현하고 있는 것을 확인할 수 있습니다.

Preparing for new scheduler

명세대로 스케쥴링을 구하기 위핸 `Queue`와 `Heap`을 구현해야 합니다.

L0, L1, L2, Monopoly 큐는 RR 방식이므로 `Queue`로 구현하고, L3은 우선순위 큐이므로 `Heap`으로 구현하면 됩니다.

```
//FILE: proc.c
struct rn_queue {
  int size;
  int front;
  int tail;
  struct proc* proc[NPROC * 2];
};

static void rn_queue_insert(struct rn_queue* q, struct proc* p) {
  if (q->size == NPROC * 2) return;
  q->proc[q->tail] = p;
  q->tail = (q->tail + 1) % (NPROC * 2);
  q->size++;
}

static int rn_queue_is_empty(struct rn_queue* q) {
  return q->size == 0;
}

static struct proc* rn_queue_pop(struct rn_queue* q) {
  struct proc* ret;

  if (q->size == 0) return 0;

  ret = q->proc[q->front];
  q->front = (q->front + 1) % (NPROC * 2);
  q->size--;
  return ret;
}

static int rn_queue_size(struct rn_queue* q) {
  return q->size;
}

struct rn_heap {
  int size;
  struct proc* proc[NPROC * 2 + 1];
};

static void rn_heap_swap(struct rn_heap* h, int i, int j) {
  struct proc* tmp;

  tmp = h->proc[i];
  h->proc[i] = h->proc[j];
  h->proc[j] = tmp;
}
```

```
static void rn_heap_decrease(struct rn_heap* h, int i) {
  int max_i;

  for (; ;) {
    if ((i << 1) <= h->size && h->proc[i << 1]->priority > h->proc[i]-
>priority) {
      max_i = i << 1;
    } else max_i = i;

    if ((i << 1 | 1) <= h->size && h->proc[i << 1 | 1]->priority > h-
>proc[max_i]->priority) {
      max_i = i << 1 | 1;
    }
    if (max_i != i) {
      rn_heap_swap(h, i, max_i);
      i = max_i;
    } else break;
  }
}

static struct proc* rn_heap_pop(struct rn_heap* h) {
  struct proc* ret;

  if (h->size == 0) return 0;
  ret = h->proc[1];
  h->proc[1] = h->proc[h->size];
  h->size--;
  rn_heap_decrease(h, 1);
  return ret;
}

static void rn_heap_ify(struct rn_heap* h, int i) {
  int max_i;

  for (; i > 0; ) {
    if ((i << 1) <= h->size && h->proc[i << 1]->priority > h->proc[i]-
>priority) {
      max_i = i << 1;
    } else max_i = i;

    if ((i << 1 | 1) <= h->size && h->proc[i << 1 | 1]->priority > h-
>proc[max_i]->priority) {
      max_i = i << 1 | 1;
    }
    if (max_i != i) {
      rn_heap_swap(h, i, max_i);
      i >>= 1;
    } else break;
  }
}

static void rn_heap_insert(struct rn_heap* h, struct proc* p) {
  if (h->size == NPROC * 2) return;
  h->size++;
```

```
  h->proc[h->size] = p;
  rn_heap_ify(h, h->size >> 1);
}

static int rn_heap_is_empty(struct rn_heap* h) {
  return h->size == 0;
}

static int rn_heap_delete(struct rn_heap* h, struct proc* p) {
  int i, ret;

  ret = 0;
  for (i = 1; i <= h->size; ++i) {
    if (h->proc[i] == p) {
      h->proc[i] = h->proc[h->size];
      h->size--;
      rn_heap_decrease(h, i);
      ret = 1;
      break;
    }
  }

  return ret;
}
```

스케쥴링 특성 상 최대 NPROC * 2개까지 프로세스 큐에 들어갈 수 있으므로, 이를 고려하여 큐를 구현했습니다.

> 간단한 자료구조이므로, 큐와 힙의 상세한 구현 방식은 문서에서 생략했습니다.

ptable

이후 기존 `ptable`에 해당 큐를 추가했습니다. 큐는 포인터로 `ptable.proc` 원소를 가리키도록 했습니다.

```
//FILE: proc.c
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
  struct rn_queue l[3];
  struct rn_heap l3;
  struct rn_queue monopoly;
  int monopoly_size;
} ptable;
```

cpu

이번 프로젝트에서는 스케쥴링을 할 때, 기본적으로 `MLFQ` 방식으로 동작하되, 유저가 `monopolize`를 호출하면 `MoQ` 방식으로 등록해놓은 프로세스만 실행해야 합니다.

따라서, `cpu` 구조체에 어떤 방식으로 스케쥴링할지 저장할 수 있는 변수를 추가했습니다.

```
//FILE: proc.h
// Per-CPU state
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null

  // rn's custom values
  int cpu_schedule_mode;       // CPU scheduling mode / 0: MLFQ, 1: MoQ
};
```

스케쥴러에서 `cpu.cpu_schedule_mode`를 보고 스케쥴링 방식을 결정하도록 했습니다.

proc

`MLFQ` 방식으로 동작할 때, 해당 프로세스는 정해진 타임 인터럽트 회수만큼 CPU를 사용하고, CPU를 반납해야 합니다.

또, L3 큐에 존재할 경우 priority를 프로세스별로 가지고 있어야 합니다.

따라서, 해당 프로세스가 어떤 큐에 존재하는지, priority는 무엇인지, CPU를 사용하고 있다면, 현재까지 얼마나 사용했는지를 저장하는 변수를 추가했습니다.

```
// 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)

  // rn's custom values
  int q_level;                 // Process queue level
  int priority;                // Process priority
  int time_quantum;            // Process time quantum
};
```

(new) scheduler

새로운 스케줄러는 다음과 같이 구현했습니다.

```
static void scheduler_run_process(struct cpu* c, struct proc* p) {
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;
  p->time_quantum = 0;
  swtch(&(c->scheduler), p->context);
  switchkvm();
  c->proc = 0;
}

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

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

    acquire(&ptable.lock);

    if (c->cpu_schedule_mode == 0) { // MLFQ
      for (i = 0; i < 3; ++i) {
        // process L_i (0 <= i <= 2)
        for (; !rn_queue_is_empty(&ptable.l[i]); ) {
          p = rn_queue_pop(&ptable.l[i]);

          if (p->state != RUNNABLE) continue;

          if (p->q_level != i) { // lazy propagate queue level
            // if changed queue level is monoplized then skip
            if (p->q_level != 99) {
              if (p->q_level >= 0 && p->q_level <= 2)
rn_queue_insert(&ptable.l[p->q_level], p);
              else rn_heap_insert(&ptable.l3, p);
            }
            // if queue level is changed then break and recheck from L0
            goto cont;
          }
          scheduler_run_process(c, p);

          if (p->state == RUNNABLE) {
            if (p->q_level >= 0 && p->q_level <= 2)
rn_queue_insert(&ptable.l[p->q_level], p);
            else rn_heap_insert(&ptable.l3, p);
```

```
            }

            goto cont;
          }
        }
        // process L3
        for (; !rn_heap_is_empty(&ptable.l3); ) {
          p = rn_heap_pop(&ptable.l3);

          if (p->state != RUNNABLE) continue;

          if (p->q_level != 3) { // lazy propagate queue level
            // if changed queue level is monoplized then skip
            if (p->q_level != 99) {
              if (p->q_level >= 0 && p->q_level <= 2)
rn_queue_insert(&ptable.l[p->q_level], p);
              else rn_heap_insert(&ptable.l3, p);
            }
            // if queue level is changed then break and recheck from L0
            goto cont;
          }

          scheduler_run_process(c, p);

          if (p->state == RUNNABLE) {
            if (p->q_level >= 0 && p->q_level <= 2)
rn_queue_insert(&ptable.l[p->q_level], p);
            else rn_heap_insert(&ptable.l3, p);
          }
          goto cont;
        }
      } else { // MoQ
        for (; !rn_queue_is_empty(&ptable.monopoly); ) {
          p = rn_queue_pop(&ptable.monopoly);
          if (p->state != RUNNABLE) continue;

          scheduler_run_process(c, p);

          if (p->state == ZOMBIE) {
            --ptable.monopoly_size;
          } else if (p->state == RUNNABLE) {
            rn_queue_insert(&ptable.monopoly, p);
          }
        }
        if (ptable.monopoly_size <= 0) unmonopolize();
      }
  cont:
      release(&ptable.lock);
    }
  }
```

해당 스케쥴러는 다음과 같이 동작합니다.

1. `cpu_schedule_mode`에 따라 MLFQ 혹은 MoQ로 스케쥴링을 결정합니다.
2. MLFQ의 경우, L0, L1, L2, L3 큐를 순회하며 실행 가능한 프로세스를 찾아 실행합니다.
3. 큐에 존재하는 프로세스가 실행가능하지 않다면, 큐에서 제거하고 다음 프로세스를 찾습니다.
4. 큐에 존재하는 프로세스가 실행 가능하다면, 해당 프로세스로 context switching을 합니다.
5. 이후 실행 시간이 다 됐고, RUNNABLE한 상태라면, 다시 큐에 넣습니다.
6. 만약 실행 가능한 상태이지만, 큐 레벨이 변경됐다면, 해당 레벨 큐에 넣고 다시 탐색을 시작합니다. (lazy propagation - 주어진 타임 퀀텀을 다 썼을 경우, trap.c 에서 queue level을 변경하므로, 이를 반영하기 위함입니다. 문서 아래에 자세한 설명이 있습니다.)
7. MoQ의 경우, MoQ 큐에서 실행 가능한 프로세스를 찾아 실행합니다.
8. MoQ 큐에 존재하는 프로세스가 실행 가능하지 않다면, 큐에서 제거하고 다음 프로세스를 찾습니다.
9. MoQ 큐에 존재하는 프로세스가 실행 가능하다면, 해당 프로세스로 context switching을 합니다.
10. 이후 실행 시간이 다 됐고, RUNNABLE한 상태라면, 다시 큐에 넣습니다.
11. 실행 가능한 프로세스가 없다면 (모든 Monopoly 프로세스가 종료됐다면), MoQ 모드를 해제합니다.

> q_level Feedback을 받아 큐 레벨을 변경할 때, 모노폴리 큐로 넣어주지 않는 이유는, Feedback을 받아 큐 레벨을 변경할 때, Monopoly 큐로 이동할 일이 없기 때문입니다. 따라서 넣어주게 된다면, `setmonopoly`에서만 넣어줘야 합니다. 그렇지 않으면 중복으로 큐에 들어가게 됩니다.

(new) trap

기존의 trap에서는 타이머 인터럽트가 발생했을 때, 무조건 `yield` 함수를 호출했습니다.
하지만 과제 명세 상, L0, L1, L2, L3 큐에서는 각각 (2 * q_level + 2)번째 타이머 인터럽트가 발생했을 때 CPU를 반납해야 합니다.
또한, MoQ 모드로 CPU가 동작하고 있을 때는 CPU를 반납하면 안 됩니다.

따라서, trap 함수를 다음과 같이 수정했습니다.

현재 실행 중인 프로세스에서 타이머 인터럽트가 발생했을 때 다음과 같이 동작합니다.

1. 글로벌 틱이 100이 되면, 모든 프로세스의 우선순위를 높여주는 `priority_boosting` 함수를 호출합니다. 이후 무조건 CPU를 반납합니다. (CPU를 반납하지 않아도 되지만, 의도적으로 우선순위를 변경해주기 위해 CPU를 반납합니다.) Monopoly 모드로 동작하고 있다면, `priority_boosting` 함수를 호출하지 않습니다.
2. Monopoly 모드로 동작하고 있다면, CPU를 반납하지 않고 무시합니다.
3. 이후 주어진 타임 퀀텀(2 * q_level + 2)만큼 CPU를 사용했다면, 큐 레벨을 변경(Feedback)하고 CPU를 반납합니다. (L3 라면 priority를 0미만으로 내려가지 않게 1 감소합니다.)

> 큐 레벨을 변경할 때 실제로 큐를 조작해 변경하지 않습니다. 프로세스의 큐 레벨이 변경됐다는 정보만 기록하고 넘어간 뒤, 다음 스케쥴링 작업에서 큐 레벨을 변경합니다.
> 이렇게 처리해도 되는 이유는 Feedback 과정에서 L1->L0 처럼 상위 큐로 이동하지 않기 때문에 가능합니다. L1->L3 같은 변경이 일어났을 때, L1큐에 들어있는 프로세스의 큐 레벨을 L3로 변경해놓고 실제 스케쥴링 시 L1 큐에서 L3 큐로 이동하게 됩니다. 만약 L1->L0같은 변경이 일어났다면, L1 큐에 남아있는 프로세스를 L0큐에 있는 작업을 마칠 때까지 발견하지 못하는 이슈가 있을 수 있지만, L3에 있어야 하는 프로세스가 L1에서 먼저 발견되어 그때 L3로 옮기는 행동은 문제가 없기 때문입니다.

따라서, 명세에 맞게 CPU를 할당받은만큼 사용하고, Feedback을 받아 큐 레벨을 변경하며, 글로벌 틱이 100이 되면 모든 프로세스의 우선순위를 높여주는 코드를 작성했습니다. 또 같은 레벨에서 계속 CPU를 할당 받아 사용할 경우, `Queue` 구조에 의해 Round Robin 스케쥴링이 이루어집니다.

```
//FILE: trap.c
void
trap(struct trapframe *tf)
{
  struct proc* p;
  int nticks;
  nticks = 1;

  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      nticks = ticks;
      wakeup(&ticks);
      if (mycpu()->cpu_schedule_mode == 0 && nticks % 100 == 0) {
        priority_boosting();
      }
      release(&tickslock);
    }
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE+1:
    // Bochs generates spurious IDE1 interrupts.
    break;
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_COM1:
    uartintr();
    lapiceoi();
    break;
  case T_IRQ0 + 7:
  case T_IRQ0 + IRQ_SPURIOUS:
    cprintf("cpu%d: spurious interrupt at %x:%x\n",
            cpuid(), tf->cs, tf->eip);
    lapiceoi();
    break;
```

```
  //PAGEBREAK: 13
  default:
    if(myproc() == 0 || (tf->cs&3) == 0){
      // In kernel, it must be our mistake.
      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
              tf->trapno, cpuid(), tf->eip, rcr2());
      panic("trap");
    }
    // In user space, assume process misbehaved.
    cprintf("pid %d %s: trap %d err %d on cpu %d "
            "eip 0x%x addr 0x%x--kill proc\n",
            myproc()->pid, myproc()->name, tf->trapno,
            tf->err, cpuid(), tf->eip, rcr2());
    myproc()->killed = 1;
  }

  // Force process exit if it has been killed and is in user space.
  // (If it is still executing in the kernel, let it keep running
  // until it gets to the regular system call return.)
  if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();

  if (myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0 + IRQ_TIMER && nticks % 100 == 0 &&
     mycpu()->cpu_schedule_mode == 0) {
    yield();
  } else if (myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0 + IRQ_TIMER) {
    // Force process to give up CPU on clock tick.
    // If interrupts were on while locks held, would need to check nlock.
    p = myproc();
    p->time_quantum++;
    if (p->q_level != 99) { // If the process is monoplized then don't
yield cpu
      if (p->time_quantum >= p->q_level * 2 + 2) { // If the process has
used up its time quantum
        p->time_quantum = 0;

        if (p->q_level == 0) {
          if (p->pid & 1) { // make the process to be in the L1 queue
            p->q_level = 1;
          } else { // make the process to be in the L2 queue
            p->q_level = 2;
          }
        } else if (p->q_level < 3) { // If the process is in the L1 or L2
queue
          // make the process to be in the L3 queue
          p->q_level = 3;
        } else if (p->q_level == 3) { // If the process is in the L3 queue
          // sub process's priority by 1
          p->priority--;
          // priority can't be negative
          if (p->priority < 0) p->priority = 0;
```

```
        }
        yield();
      }
    }
  }


  // Check if the process has been killed since we yielded
  if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();
}
```

Queue Handling

위 코드와 설명은, 큐에 실행 가능 프로세스가 잘 담겨있다는 가정 하에 작동하는 코드입니다.

따라서, 실행 가능한 프로세스를 큐에 관리하는 코드가 추가되어야 합니다.

먼저, 실행 가능한 프로세스를 관리하는 부분은 다음과 같습니다.

1. `myproc.c - userinit`
2. `myproc.c - fork`
3. `myproc.c - wakeup1`
4. `myproc.c - kill`

1, 2는 새로운 프로세스를 만들 때 호출되고, 3, 4는 자고있거나 다른 상태(!= RUNNABLE)인 프로세스를 다시 실행 가능한 프로세스로 만들 때 호출됩니다.

그 외에는 프로세스 상태를 실행 가능하게 변경해주지 않습니다. 따라서 해당 함수에서 큐에 프로세스를 추가해주는 코드를 작성해야 합니다.

1, 2에서는 다음 코드를 추가해야 합니다.

```
//FILE: proc.c
p->state = RUNNABLE;
p->priority = 0;
p->q_level = 0;
rn_queue_insert(&ptable.l[0], p);
```

3, 4에서는 다음 코드를 추가해야 합니다.

```
//FILE: proc.c
if (p->q_level == 99) rn_queue_insert(&ptable.monopoly, p);
else if (p->q_level >= 0 && p->q_level <= 2) rn_queue_insert(&ptable.l[p-
>q_level], p);
else rn_heap_insert(&ptable.l3, p);
```

Scheduler Handling Systemcall

스케쥴러를 핸들링하기위한 시스템 콜 목록입니다.

- `yield`: 명시적으로 CPU 반납
- `getlev`: 현재 프로세스의 레벨 반환 (L0, L1, L2, L3, Monopoly)
- `setpriority`: 프로세스의 우선순위 변경 (L3에서만 의미있는 값)
- `setmonopoly`: 해당 프로세스를 MoQ로 이동
- `monopolize`: 현재 CPU 실행 모드를 Monopoly로 변경
- `unmonopolize`: 현재 CPU 실행 모드를 Monopoly에서 일반 모드로 변경

yield

명시적으로 CPU를 반납합니다. 현재 프로세스를 실행 가능한 프로세스로 만들고, 스케쥴러로 돌아가 다음 스케쥴링을 진행합니다.

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

getlev

현재 프로세스의 큐 레벨을 반환합니다.

```
// get current process's queue level
int getlev(void) {
  struct proc *p;
  int ret;

  ret = -1;

  acquire(&ptable.lock);
  p = myproc();
  ret = p->q_level;
  release(&ptable.lock);

  return ret;
}
```

setpriority

해당 pid를 가진 프로세스의 priority를 변경합니다.

Priority는 0 이상 10 이하의 값이어야 합니다. 만약, priority가 범위를 벗어나면 -2를 반환합니다.
만약 pid가 존재하지 않으면 -1을 반환합니다.
정상적으로 priority를 변경했다면 0을 반환합니다.

만약, Heap에 존재한다면, Heap에서 삭제하고, 다시 Heap에 넣어줍니다.

> 이렇게 하는 이유는, Heap은 우선순위가 변경되면 다시 정렬해야 하기 때문입니다. 그렇지 않으면 Heap 구조가 깨져 정상적인 우선순위 스케쥴링이 이루어지지 않습니다.

```
// set process's priority
int setpriority(int pid, int priority) {
  struct proc *p;
  int ret;

  if (priority < 0 || priority > 10) return -2;
  ret = -1;

  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->pid == pid) {
      p->priority = priority;
      ret = 0;
      break;
    }
  }
  if (rn_heap_delete(&ptable.l3, p)) {
    rn_heap_insert(&ptable.l3, p);
  }
  release(&ptable.lock);

  return ret;
}
```

setmonopoly

해당 pid를 가진 프로세스를 Monopoly 큐로 이동합니다.

Password가 일치하지 않으면, -2를 반환합니다.
만약, pid가 존재하지 않으면 -1을 반환합니다.
정상적으로 Monopoly 큐로 이동했다면, Monopoly 큐의 크기를 반환합니다.

만약, 이미 Monopoly 큐에 존재한다면, 작업을 무시합니다.

```
// set pid's process to be monopolized
int setmonopoly(int pid, int password) {
  struct proc *p;
  int ret;

  if (password != 2020021949) return -2;
  ret = -1;
```

```
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->pid == pid) {
      if (p->q_level != 99) { // if process is not in MoQ
        p->q_level = 99;
        rn_queue_insert(&ptable.monopoly, p);
      }
      ret = ++ptable.monopoly_size;
      break;
    }
  }
  release(&ptable.lock);

  return ret;
}
```

monopolize

현재 CPU를 Monopoly 모드로 변경합니다.

mycpu를 호출하기 위해선 인터럽트를 막고 호출해야 합니다. 따라서 pushcli와 popcli를 사용해야 합니다.

monopolize를 했다면, 다시 스케쥴링을 해야 하기 때문에 현재 프로세스의 CPU를 반납하고 다음 스케쥴링을 진행합니다.

```
// make cpu using MoQ
void monopolize(void) {
  struct cpu* c;

  pushcli();
  c = mycpu();
  c->cpu_schedule_mode = 1;
  popcli();

  yield();
}
```

unmonopolize

현재 CPU를 Monopoly 모드에서 일반 모드로 변경합니다.

유저가 호출할 일이 없다는 가정 하에 작성했습니다. 따라서 CPU 모드만 변경 및 글로벌 틱 초기화만 진행하고, 넘어갑니다.

```
// make cpu using MLFQ
void unmonopolize(void) {
  struct cpu* c;

  pushcli();
  c = mycpu();
```

```
  c->cpu_schedule_mode = 0;
  popcli();

  acquire(&tickslock);
  ticks = 0;
  release(&tickslock);
}
```

Priority Boosting

모든 프로세스의 우선순위를 높여주는 함수입니다. Monopoly Queue에 존재하지 않는 모든 프로세스에 대해 q_level을 모두 0으로 변경합니다.

이후 L1, L2, L3 큐에 존재하는 모든 프로세스를 L0 큐로 이동시킵니다.

```
// priority boosting
void priority_boosting(void) {
  int i;
  struct proc *p;

  acquire(&ptable.lock);
  for (i = 1; i < 3; ++i) {
    for (; !rn_queue_is_empty(&ptable.l[i]); ) {
      p = rn_queue_pop(&ptable.l[i]);
      if (p->q_level < 0 || p->q_level > 3) continue;

      p->q_level = 0;
      rn_queue_insert(&ptable.l[0], p);
    }
  }
  for (; !rn_heap_is_empty(&ptable.l3); ) {
    p = rn_heap_pop(&ptable.l3);
    if (p->q_level < 0 || p->q_level > 3) continue;

    p->q_level = 0;
    rn_queue_insert(&ptable.l[0], p);
  }

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->q_level != 99) p->q_level = 0;
  }

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

user app for testing scheduling

스케쥴링을 잘 처리하는지 확인하기 위해, 다음과 같은 테스트 프로그램을 작성할 수 있습니다.

코드가 길어 첨부하진 않았지만, `rn_sched_test.c` 파일에서 확인할 수 있습니다.

프로세스들이 특정 큐에 잘 머무는지, 큐마다 우선순위가 잘 지켜지는지 확인하기 위해 `lapic.c`를 분석해서 타이머 인터럽트에서 사용하는 타이머값을 이용해 CPU를 사용하면서 x milli second를 기다리는 rn_sleep 함수를 다음과 같이 구현했습니다.

```
void sys_rn_sleep(void) {
  int ms, prev, cur, ms_tick;

  if(argint(0, &ms) < 0)
    return;

  prev = lapic[0x0390 / 4];
  ms_tick = 0;
  for (; ;) {
    cur = lapic[0x0390 / 4];

    if (cur + 1000000 <= prev) {
      if (++ms_tick == ms) break;
      prev -= 1000000;
    } else if (cur >= prev) {
      prev += 10000000;
    }
  }
}
```

이후, 다음과 같은 프로세스로 여러 테스트를 구현했습니다.

test 1

1. L0 큐에 머무르는 프로세스를 세 개 생성하고, 각각 다른 시간을 사용하도록 합니다.
2. 자식 프로세스는 해당 큐에 머물면서 어떤 레벨에서 프로세스를 실행하는지 기록합니다.
3. 이후 부모 프로세스에서 사용 시간이 적은 프로세스부터 순서대로 reaping되는지 확인합니다.
4. 자식 프로세스에서 종료 이전에 프로세스가 실행되면서 어떤 레벨에 머물렀는지 출력하고 종료합니다.

이 테스트를 통해, L0~L0 큐에서만 머무는지, 라운드 로빈이 잘 이루어지는지 확인할 수 있습니다.

test 2

1. L1~L2 큐에 머무르는 프로세스를 네 개 생성하고, 각각 다른 시간을 사용하도록 합니다.
2. 자식 프로세스는 해당 큐에 머물면서 어떤 레벨에서 프로세스를 실행하는지 기록합니다.
3. 이후 부모 프로세스에서 L1큐 우선, L2큐 우선, 사용 시간이 적은 프로세스부터 순서대로 reaping되는지 확인합니다.
4. 자식 프로세스에서 종료 이전에 프로세스가 실행되면서 어떤 레벨에 머물렀는지 출력하고 종료합니다.

이 테스트를 통해, L0~L2 큐에서만 머무는지, 우선순위가 잘 지켜지는지 확인할 수 있습니다. 또한, L0 큐에서 일정 부분 머무른 값이 있다면 priority boosting이 잘 이루어지는지 확인할 수 있습니다.

test 3

1. L3 큐에 머무르는 프로세스를 네 개 생성하고, 각각 다른 시간 및 priority를 사용하도록 합니다.
2. 자식 프로세스는 해당 큐에 머물면서 어떤 레벨에서 프로세스를 실행하는지 기록합니다.

3. 이후 부모 프로세스에서 priority 우선, 사용 시간이 적은 프로세스부터 순서대로 reaping되는지 확인합니다.
4. 자식 프로세스에서 종료 이전에 프로세스가 실행되면서 어떤 레벨에 머물렀는지 출력하고 종료합니다.

이 테스트를 통해, L0~L3 큐에서만 머무르는지, priority가 잘 지켜지는지 확인할 수 있습니다. 또한, L0, L1, L2 큐에서 일정 부분 머무른 값이 있다면 priority boosting이 잘 이루어지는지 확인할 수 있습니다.

test 4

1. Monopoly 큐에 머무르는 프로세스를 세 개 생성하고, 각각 다른 시간을 사용하도록 합니다.
2. 자식 프로세스는 해당 큐에 머물면서 어떤 레벨에서 프로세스를 실행하는지 기록합니다.
3. 이후 부모 프로세스에서 FCFS로 reaping되는지 확인합니다.
4. 자식 프로세스에서 종료 이전에 프로세스가 실행되면서 어떤 레벨에 머물렀는지 출력하고 종료합니다.

이 테스트를 통해, Monopoly 큐에서만 머무르는지, FCFS가 잘 이루어지는지 확인할 수 있습니다.

test 5

1. L0~L3 큐에 머무르는 프로세스를 여섯 개 생성하고, 각각 다른 시간 및 priority를 사용하도록 합니다.
2. 자식 프로세스는 해당 큐에 머물면서 어떤 레벨에서 프로세스를 실행하는지 기록합니다.
3. 이후 부모 프로세스에서 L0, L1, L2, L3 우선, priority 우선, 사용 시간이 적은 프로세스부터 순서대로 reaping되는지 확인합니다.
4. 자식 프로세스에서 종료 이전에 프로세스가 실행되면서 어떤 레벨에 머물렀는지 출력하고 종료합니다.

이 테스트를 통해, L0~L3 큐에서만 머무르는지, 우선순위가 잘 지켜지는지 확인할 수 있습니다. 또한, L0, L1, L2 큐에서 일정 부분 머무른 값이 있다면 priority boosting이 잘 이루어지는지 확인할 수 있습니다.

test 6

1. L0~L3, Monopoly 큐에 머무르는 프로세스를 여덟 개 생성하고, 각각 다른 시간 및 priority를 사용하도록 합니다.
2. 자식 프로세스는 해당 큐에 머물면서 어떤 레벨에서 프로세스를 실행하는지 기록합니다.
3. 이후 부모 프로세스에서 L0, L1, L2, L3, priority 우선, 사용 시간이 적은 프로세스부터 순서대로 reaping되는지 확인합니다.
4. 처음에는 MLFQ로 동작하다가 세 번째 자식이 reaping되는 순간, Monopoly 모드로 변경합니다.
5. Monopoly 모드로 변경되면, Monopoly 큐에서 FCFS로 reaping되는지 확인합니다.
6. 자식 프로세스에서 종료 이전에 프로세스가 실행되면서 어떤 레벨에 머물렀는지 출력하고 종료합니다.

이 테스트를 통해, L0~L3 큐에서만 머무르는지, 우선순위가 잘 지켜지는지 확인할 수 있습니다. 또한, L0, L1, L2 큐에서 일정 부분 머무른 값이 있다면 priority boosting이 잘 이루어지는지 확인할 수 있습니다. 마지막으로, Monopoly 모드가 잘 동작하는지 확인할 수 있습니다. (Monopoly <-> MLFQ 전환)

test 7

1. Monopoly 큐에 머무르는 프로세스를 세 개 생성하고, 각각 다른 시간을 사용하도록 합니다.
2. Monopoly 모드로 동작하는 과정에서 sleeping 모드로 전환하면서 스케쥴링 시점에서 모든 프로세스가 자고 있는 상태를 만듭니다.
3. 자식 프로세스는 해당 큐에 머물면서 어떤 레벨에서 프로세스를 실행하는지 기록합니다.
4. 이후 부모 프로세스에서 FCFS로 reaping되는지 확인합니다.
5. 자식 프로세스에서 종료 이전에 프로세스가 실행되면서 어떤 레벨에 머물렀는지 출력하고 종료합니다.

이 테스트를 통해, Monopoly 큐에서만 머무르는지, FCFS가 잘 이루어지는지 확인할 수 있습니다. 또, Monopoly 모드에서 모든 프로세스가 sleeping인 상태에서 unmonopolize가 이상하게 작동하지 않는지 테스트합니다.

test 8

1. L0 큐에 머무르는 프로세스를 세 개 생성하고, 각각 다른 시간을 사용하도록 합니다.
2. 프로세스가 작동하면서 sleeping 모드로 계속 전환합니다.
3. 자식 프로세스는 해당 큐에 머물면서 어떤 레벨에서 프로세스를 실행하는지 기록합니다.
4. 이후 부모 프로세스에서 사용 시간이 적은 프로세스부터 순서대로 reaping되는지 확인합니다.
5. 자식 프로세스에서 종료 이전에 프로세스가 실행되면서 어떤 레벨에 머물렀는지 출력하고 종료합니다.

이 테스트를 통해, L0~L0 큐에서만 머무는지, 라운드 로빈이 잘 이루어지는지 확인할 수 있습니다. 또한, 모든 프로세스가 sleeping인 상태에서 제대로 스케쥴링 되는지 확인합니다.

test 9

1. L0 큐에 머무르는 프로세스를 세 개 생성하고, 각각 다른 시간을 사용하도록 합니다.
2. 프로세스가 작동하면서 sleeping 모드로 계속 전환합니다.
3. 자식 프로세스는 해당 큐에 머물면서 어떤 레벨에서 프로세스를 실행하는지 기록합니다.
4. 이후 부모 프로세스에서 사용 시간이 적은 프로세스부터 순서대로 reaping되는지 확인합니다.
5. 자식 프로세스에서 종료 이전에 프로세스가 실행되면서 어떤 레벨에 머물렀는지 출력하고 종료합니다.
6. 첫 번째 프로세스가 reaping되면서 가장 늦게 끝나야 하는 프로세스를 Kill합니다.

이 테스트를 통해, L0~L0 큐에서만 머무는지, 라운드 로빈이 잘 이루어지는지 확인할 수 있습니다. 또한, 모든 프로세스가 sleeping인 상태에서 제대로 스케쥴링 되는지 확인합니다. 또한, Kill이 잘 작동하는지 확인합니다.

test 10

1. Monopoly 큐에 머무르는 프로세스를 세 개 생성하고, 각각 다른 시간을 사용하도록 합니다.
2. Monopoly 모드로 동작하는 과정에서 sleeping 모드로 전환하면서 스케쥴링 시점에서 모든 프로세스가 자고 있는 상태를 만듭니다.
3. 자식 프로세스는 해당 큐에 머물면서 어떤 레벨에서 프로세스를 실행하는지 기록합니다.
4. 이후 부모 프로세스에서 FCFS로 reaping되는지 확인합니다.
5. 자식 프로세스에서 종료 이전에 프로세스가 실행되면서 어떤 레벨에 머물렀는지 출력하고 종료합니다.
6. 두 번째 Monopoly로 등록된 프로세스를 자기 자신을 Kill 합니다.

이 테스트를 통해, Monopoly 큐에서만 머무르는지, FCFS가 잘 이루어지는지 확인할 수 있습니다. 또, Monopoly 모드에서 모든 프로세스가 sleeping인 상태에서 unmonopolize가 이상하게 작동하지 않는지 테스트합니다. 또한, Kill이 잘 작동하는지 확인합니다.

3. Result

스케쥴링 테스트 앱을 구현하고 테스트한 결과, 정상적으로 동작하는 것을 확인할 수 있었습니다.

```
$ rn_sched_test
rn_test1 [L0 Only]
 [ process 6 (p3) ]
    - L0:       100
    - L1:       0
```

```
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  0

 [ process 5 (p2) ]
    - L0:       500
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  0

 [ process 4 (p1) ]
    - L0:       1000
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  0

rn_test1 finished successfully


rn_test2 [L1~L2 Only]
 [ process 9 (p3) ]
    - L0:       3
    - L1:       97
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  0

 [ process 7 (p1) ]
    - L0:       6
    - L1:       294
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  0

 [ process 10 (p4) ]
    - L0:       10
    - L1:       0
    - L2:       40
    - L3:       0
    - MONOPOLIZED:  0

 [ process 8 (p2) ]
    - L0:       11
    - L1:       0
    - L2:       189
    - L3:       0
    - MONOPOLIZED:  0

rn_test2 finished successfully


rn_test3 [L3 Only]
```

```
 [ process 12 (p2) ]
    - L0:       1
    - L1:       0
    - L2:       5
    - L3:       54
    - MONOPOLIZED:  0

 [ process 14 (p4) ]
    - L0:       5
    - L1:       0
    - L2:       9
    - L3:       46
    - MONOPOLIZED:  0

 [ process 13 (p3) ]
    - L0:       6
    - L1:       6
    - L2:       0
    - L3:       48
    - MONOPOLIZED:  0

 [ process 11 (p1) ]
    - L0:       8
    - L1:       7
    - L2:       0
    - L3:       45
    - MONOPOLIZED:  0

rn_test3 finished successfully


rn_test4 [MoQ Only - wait 5sec]
monopolize!
monopolized process 15
 [ process 15 (p1) ]
    - L0:       0
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  100

monopolized process 16
 [ process 16 (p2) ]
    - L0:       0
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  60

monopolized process 17
 [ process 17 (p3) ]
    - L0:       0
    - L1:       0
    - L2:       0
```

```
    - L3:       0
    - MONOPOLIZED:  30

rn_test4 finished successfully


rn_test5 [L0, L1, L2, L3]
 [ process 23 (p6) ]
    - L0:       500
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  0

 [ process 22 (p5) ]
    - L0:       1000
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  0

 [ process 21 (p4) ]
    - L0:       9
    - L1:       191
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  0

 [ process 20 (p3) ]
    - L0:       16
    - L1:       0
    - L2:       184
    - L3:       0
    - MONOPOLIZED:  0

 [ process 18 (p1) ]
    - L0:       9
    - L1:       0
    - L2:       9
    - L3:       42
    - MONOPOLIZED:  0

 [ process 19 (p2) ]
    - L0:       2
    - L1:       15
    - L2:       0
    - L3:       43
    - MONOPOLIZED:  0

rn_test5 finished successfully


rn_test6 [L0, L1, L2, L3, MoQ]
 [ process 29 (p6) ]
```

```
    - L0:       500
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  0

 [ process 28 (p5) ]
    - L0:       1000
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  0

 [ process 27 (p4) ]
    - L0:       9
    - L1:       191
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  0

monopolize!
monopolized process 30
 [ process 30 (p7) ]
    - L0:       0
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  30

monopolized process 31
 [ process 31 (p8) ]
    - L0:       0
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  30

 [ process 26 (p3) ]
    - L0:       14
    - L1:       0
    - L2:       186
    - L3:       0
    - MONOPOLIZED:  0

 [ process 24 (p1) ]
    - L0:       9
    - L1:       0
    - L2:       8
    - L3:       43
    - MONOPOLIZED:  0

 [ process 25 (p2) ]
    - L0:       11
    - L1:       5
```

```
    - L2:       0
    - L3:       44
    - MONOPOLIZED:  0

rn_test6 finished successfully


rn_test7 [MoQ Only - Sleep Test]
monopolize!
monopolized process 32
 [ process 32 (p1) ]
    - L0:       0
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  100

monopolized process 33
 [ process 33 (p2) ]
    - L0:       0
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  60

monopolized process 34
 [ process 34 (p3) ]
    - L0:       0
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  30

rn_test7 finished successfully


rn_test8 [L0 Only - All Sleep]
 [ process 37 (p3) ]
    - L0:       50
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  0

 [ process 36 (p2) ]
    - L0:       100
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  0

 [ process 35 (p1) ]
    - L0:       200
    - L1:       0
```

```
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  0

rn_test8 finished successfully


rn_test9 [L0 Only - Sleep & Kill]
 [ process 40 (p3) ]
    - L0:       50
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  0

kill process 38
 [ process 39 (p2) ]
    - L0:       100
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  0

rn_test9 finished successfully


rn_test10 [MoQ Only - Kill Self]
monopolize!
monopolized process 41
 [ process 41 (p1) ]
    - L0:       0
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  100

monopolized process 42
monopolized process 43
 [ process 43 (p3) ]
    - L0:       0
    - L1:       0
    - L2:       0
    - L3:       0
    - MONOPOLIZED:  30

rn_test10 finished successfully
```

실제 CPU를 사용 환경도 다르고, 타이머 인터럽트가 발생하는 시간도 조금씩 다르기 때문에 항상 같은 값을 얻을 수는 없지만, 비율을 생각하면 잘 작동한지 아닌지를 높은 확률로 알 수 있습니다.

직접 컴파일 및 실행하려면 다음 명령어를 사용하면 됩니다. 이후 xv6 내부에서 `rn_sched_test`를 실행하면 위와 비슷한 결과를 확인할 수 있습니다.

```
$ sh boot.sh
```

4. Trouble shooting

이번 과제는 분석해야 하는 부분이 많아서 시간이 많이 소요됐습니다.

또, printf -> write -> filewrite -> begin_op 에서 무조건 sleep을 호출해서 CPU를 반납하는 줄 알고 테스트 코드를 잘못 작성했었습니다. 추가적인 분석 이후 명시적으로 CPU를 반납해야 한다는 것을 알게 되었습니다.

코드가 실행되는 과정에서 발생하는 사이드 이펙트들을 고려하며 코드를 작성하니 훨씬 수월했습니다.