Project02 Scheduler.

1. Design.

- 스케줄러에 대한 전반적인 디자인을 소개합니다.
- 어떤 자료구조를 사용했는지, 어떤 방식으로 RR을 구현 했는지 서술합니다.
- 구현 전 설계 단계입니다.

2. Implement.

- 실질적 구현 단계입니다.
- 구현에 대한 코드 설명으로 이루어져 있습니다.

3. Result.

- 결과 단계입니다.
- 결과에 대한 과정 및 어떤 문제가 있었는지 서술합니다.

4. Trouble Shooting.

- 결과에 대한 문제 상황에 대한 고찰입니다.
- 어떤게 문제였는지, 어떤 상황 때문에 이런 문제가 생겼는지에 대한 과정을 담아두고 있습니다.

1. Design.

- 스케줄러에 대한 전반적인 디자인을 말해보겠습니다.
- 먼저 스케줄러를 짜기 전, 명세에서 무엇을 요구하는지에 대해 살펴보도록 하겠습니다.
- 먼저 필요한 건 MLFQ와 FCFS(MoQ)입니다. 먼저 MLFQ부터 살펴보도록 하겠습니다.

- MLFQ.

- MLFQ는 총 4단계의 큐로 나뉘어져 있습니다.
  - L0, L1, L2, L3 큐로 나뉘어져 있으며, 각 큐마다 다른 역할을 지니고 있습니다.
  - L0큐는 가장 먼저 실행되는 가장 우선순위의 큐입니다.
  - L1, L2는 L0큐의 time_quantuam을 다 사용하였을 경우 L1, L2로 흘러들어가는 과정으로 되어 있습니다.
  - L3큐는 우선순위 큐로 구성되어 있습니다.

- 각각의 큐는 time_quantuam이 다릅니다. 각각의 큐에다가 다른 time_quantuam을 설정해야 할 것입니다.

- L0 L3까지의 4개의 큐 자료구조가 필요해 보입니다. 큐 자료구조는 linked_list구조로 짜면 쉽게 해결 될 것 같습니다.

- 각각의 큐는 process 노드 구조체로 구성되어 있어야 구현이 쉬울 것 같습니다.

- 프로세스들에는 추가적으로 priority, tick이 필요할 것 입니다. 또, 프로세스 구조체를 queue의 노드로 사용할 것이기에, 자기의 다음 번 노드를 가리키는 `struct* proc* next` 구조체 변수가 필요할 것 같습니다.

- 어느정도의 구현을 생각해 놓았다면 이번엔 FCFS에 대해서 살펴보겠습니다.

- FCFS(MoQ).

- FCFS(MoQ)는 모노폴라이즈 큐로 우선순위가 가장 높은 특수한 큐에 해당합니다.

- 이 큐는 평소엔 실행되지 않다가, `monopolize()`함수가 호출이 되면 그때부터 자신 안에 있는 큐들의 프로세스부터 처리합니다.

- 이 큐는 CPU를 독점적으로 사용하여 `unmonopolize()`함수가 호출이 되기 전까지 계속해서 실행됩니다.

- 이 큐는 priority boosting이 일어나지 않고, 먼저 큐에 들어온 프로세스가 먼저 스케줄링 됩니다.

- 그럼 이 큐는 MLFQ와는 별개의 큐로 구현해야 할 것입니다. 다행히 일반적인 큐 자료구조의 특성을 띤다고 생각하면 될 것 같습니다.

- 어느정도 자료구조체에 대한 설정이 끝났다면, 이번엔 시스템콜과 구현할 함수에 대해서 살펴보겠습니다.

- System Call.

- 우리가 구현해야할 시스템 콜은 총 6개 입니다.
  - `void yield(void)`
    - 자신이 점유한 cpu를 양보하는 시스템 콜입니다.
  - `int getlev(void)`
    - 프로세스가 속한 큐의 레벨을 반환하는 시스템 콜입니다.
  - `int setpriority(int pid, int password)`
    - 특정 pid를 가지는 프로세스의 우선순위를 설정하는 시스템 콜입니다.
  - `int setmonopoly(int pid, int password)`
    - 특정 pid를 가진 프로세스를 MoQ로 이동시키는 시스템 콜입니다. 이때 인자로 독점 자격을 증명할 암호를 얻습니다.
  - `void monopolize(void)`
    - MoQ의 프로세스가 CPU를 독점하여 사용하도록 설정해주는 시스템 콜입니다.
  - `void unmonopolize(void)`
    - 독점적 스케줄링을 끝내고, 기존의 MLFQ로 돌아가는 시스템 콜입니다.
- 어려워 보이지만 크게 어렵지 않습니다. 자세한 구현은 Implement에서 다루면 될 것 같습니다.

- Function.

- 이번엔 우리가 필요해보이는 함수들에 대해 생각해봅시다.
- 크게 카테고리를 생각하며 묶어서 밑에다 적어보겠습니다.
  1. 큐 자료구조의 기본적인 함수들(push,pop 등)
  2. time_quantum을 전부 사용했을 때 프로세스를 자신보다 낮은 큐에 넣는 작업을 하는 함수.
  3. priority_boosting을 해주는 함수.
  4. 큐 자료구조의 함수를 가지고 구성하는 MLFQ의 기본적인 함수들.
- 이 정도가 우리가 구현해야하는 함수임을 알 수 있습니다.

어느정도의 디자인이 잡혔다면 이번엔 구현입니다.

2. Implement.

- 크게 4가지로 나누어 생각하면 될 것 같습니다.

- MLFQ 구현.
- FCFS 구현.
- 시스템 콜 구현.
- 함수 구현.

- MLFQ 구현.

- 먼저 MLFQ를 구현 해 보겠습니다.
- MLFQ는 총 3개의 큐와 한 개의 우선순위 큐로 구성되어 있습니다.
- 따라서 3개의 큐와 1개의 우선순위 큐를 가진 구조체를 먼저 선언해줍니다.

```
typedef struct mlfq{
    struct queue* queue[3];
    struct queue* priority_queue;
} mlfq;
```

- 이 구조체 안에 들어가는 함수들은 다음과 같습니다.

```
struct mlfq* mlfq_init();

void mlfq_push(mlfq* mlfq, struct proc* proc, int n);

struct proc* mlfq_pop_targetproc(mlfq* mlfq, struct proc* proc);

struct proc* mlfq_pop(struct mlfq* mlfq);

void mlfq_boost(mlfq* mlfq);
```

- 이 함수들의 각각의 역할은 다음과 같습니다.

  ‣ mlfq_push

  프로세스의 레벨에 맞는 큐에 프로세스를 넣어줍니다.

  ‣ mlfq_pop

  프로세스를 큐에서 빼옵니다.

  ‣ mlfq_boost

  priority_boosting을 해줍니다.

  ‣ 각 함수들의 자세한 구현은 다음과 같습니다.

```
void mlfq_push(struct mlfq* mlfq, struct proc* proc, int n) {
    //TODO : mlfq_push
    if(n == 0) push(mlfq->queue[L0], proc);
    if(n == 1) push(mlfq->queue[L1], proc);
    if(n == 2) push(mlfq->queue[L2], proc);
    if(n == 3) push(mlfq->priority_queue, proc);
    return;
}
```

```
    if(!isempty(mlfq->queue[L0])) {
      p = pop(mlfq->queue[L0]);
      return p;
    }

    else if(!isempty(mlfq->queue[L1])) {
      p = pop(mlfq->queue[L1]);
      return p;
    }

    else if(!isempty(mlfq->queue[L2])) {
      p = pop(mlfq->queue[L2]);
      return p;
    }

    else if(!isempty(mlfq->priority_queue)) {
      p = top_pri_proc(mlfq->priority_queue);
      return p;
    }

    return (void*)0;
```

```
void mlfq_boost(struct mlfq* mlfq) {
  struct proc* p;
  for(p = mlfq->queue[L0]->front; p != mlfq->queue[L0]->rear; p = p->next) {
    p->tick = 0;
  }
  mlfq->queue[L0]->rear->tick = 0;

  cprintf("L0 size : %d\n", mlfq->queue[L0]->size);
  cprintf("L1 size : %d\n", mlfq->queue[L1]->size);
  cprintf("L2 size : %d\n", mlfq->queue[L2]->size);
  cprintf("L3 size : %d\n", mlfq->priority_queue->size);

  for(int i=1; i<3; i++) {
      while(!isempty(mlfq->queue[i])) {
          struct proc* proc = pop(mlfq->queue[i]);
          proc->level = L0;
          proc->tick = 0;
          push(mlfq->queue[L0], proc);
      }
  }

  while(!isempty(mlfq->priority_queue)) {
      struct proc* proc = pop(mlfq->priority_queue);
      proc->level = L0;
      proc->tick = 0;
      push(mlfq->queue[L0], proc);
  }

}
```

- 다음으로는 queue 자료구조입니다.

- 큐 자료구조는 다음과 같은 구조입니다.

```
typedef struct queue {
    struct proc* rear;
    struct proc* front;
    int time_quantuam;
    int size;
} queue;
```

- 큐에는 rear, front가 존재하며, linked_list 구조로 짜져 있습니다. 각 노드는 proc 포인터가 노드의 역할을 해줍니다.

- 그 밖의 기본적인 큐 함수들(push, pop) 등이 존재합니다.

- FCFS 구현.
  - 그 다음으로는 FCFS 구현입니다.

  - FCFS의 경우 그 자체로 큐의 구조를 띠고 있어 구현이 딱히 존재하지 않습니다.

  - 시스템콜과 함께 구현되면서 자연스레 구현됩니다.
  - setmonopoly() 함수를 본다면 FCFS가 어떻게 구현되었는지 알 수 있습니다.

```
int setmonopoly(int pid, int password) {
  if(password != 2020028377) return -2;

  struct proc* p = (void*)0;
  for(int i=0; i<3; i++) {
    p = find_process_pid(q->queue[i], pid);
    if(p != (void*)0) break;
  }
  if(p == (void*)0) find_process_pid(q->priority_queue, pid);
  if(p == (void*)0) return -1;

  return MoQ->size;
}
```

  - 다음은 proc.c에서의 MoQ와 MLFQ입니다.

```
struct mlfq* q;
struct queue* MoQ;
```

- System Call 구현.
  - 시스템 콜 구현입니다. 각 시스템 콜은 다음과 같이 구현되어 있습니다.

  - `void yield()`

```
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

- `int getlev()`

```
int getlev(void) {
  //if(isexist(MoQ, myproc())) return 99;
  return myproc()->level;
}
```

- `int setpriority(int pid, int priority)`

```
int setpriority(int pid, int priority) {
  struct proc* p;

  if(priority > 10 || priority < 0) return -2;
  for(int i=0; i<3; i++) {
    p = find_process_pid(q->queue[i], pid);
    if(p != (void*)0) break;
  }
  if(p != (void*)0)
    p->priority = priority;

  return 0;
}
```

- `int setmonopoly(int pid, int password)`

```
int setmonopoly(int pid, int password) {
  if(password != 2020028377) return -2;

  struct proc* p = (void*)0;
  for(int i=0; i<3; i++) {
    p = find_process_pid(q->queue[i], pid);
    if(p != (void*)0) break;
  }
  if(p == (void*)0) find_process_pid(q->priority_queue, pid);
  if(p == (void*)0) return -1;

  return MoQ->size;
}
```

- `void monopolize(void)`

```
void monopolize() {
  monopolize_set = 1;
}
```

- `void unmonopolize(void)`

```
void unmonopolize() {
  monopolize_set = 0;
  ticks = 0;
}
```

- Scheduler 구현.
  - 다음은 스케줄러 구현입니다.

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

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

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    if(monopolize_set == 1) {
      p = pop(MoQ);

      if(p->state == RUNNABLE) {
        // 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;
      }
      if(p->state == RUNNABLE || p->state == SLEEPING) push(MoQ, p);
    }
```

```
else {
  flag = 0;
  if(ticks >= 100) {
    mlfq_boost(q);
    ticks = 0;
  }

  p = mlfq_pop(q);
  if(p == (void*)0) {
    release(&ptable.lock);
    continue;
  }
  cprintf("%d\n", p);
  //cprintf("now process level : %d\n", p->level);
  if(p->state == RUNNABLE) {
    cprintf("%d\n", p->pid);
    // 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;
  }

  p->tick++;
  if(p->tick >= q->queue[p->level]->time_quantuam) {
    p->tick = 0;
    flag = 1;
  }
```

```
      if(!flag) {
        if(p->state == RUNNABLE || p->state == SLEEPING) mlfq_push(q, p, p->level);
      }

      if(flag) {
        if(p->level == L0) {
          if(p->pid % 2 == 1) {
            p->level = L1;
            if(p->state == RUNNABLE || p->state == SLEEPING) mlfq_push(q, p, p->level);
          }
          else {
            p->level = L2;
            if(p->state == RUNNABLE || p->state == SLEEPING) mlfq_push(q, p, p->level);
          }
        }
        else if(p->level == L1 || p->level == L2) {
          p->level = L3;
          if(p->state == RUNNABLE || p->state == SLEEPING) mlfq_push(q, p, p->level);
        }
        else {
          if(p->priority > 0) p->priority--;
          if(p->state == RUNNABLE || p->state == SLEEPING) mlfq_push(q, p, p->level);
        }
      }
    }
    release(&ptable.lock);
  }
}
```

3. Result.

- 하지만 구현 결과 아쉽게도 결과는 잘 나오지 않았습니다.

- 자꾸만 init_shell에서 문제가 생겨 프로그램이 실행되지 못하는 모습입니다.

```
Booting from Hard Disk..xv6...
L0 queue size 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
L0 size : 0
L1 size : 0
L2 size : 0
L3 size : 34146
```

- 따라서 이번 프로젝트는 제게 있어 실패라고 볼 수 있습니다.

4. Trouble Shooting

- 무엇보다도 쉘이 실행이안되는게 너무나 큰 문제입니다.

- 어떤 문제인지 감이 안잡힙니다.

- 또, 제가 실수로 컨테이너를 삭제해 그간의 작업이 전부 날아갔던 기억이 있습니다.

- 또 하나는, 자료구조를 직접 만들어 하여서 꽤나 시간이 오래 걸렸다는 점입니다.

- 아마 이 시간 이후로 다시 코드를 뜯어봐야할 것 같습니다.