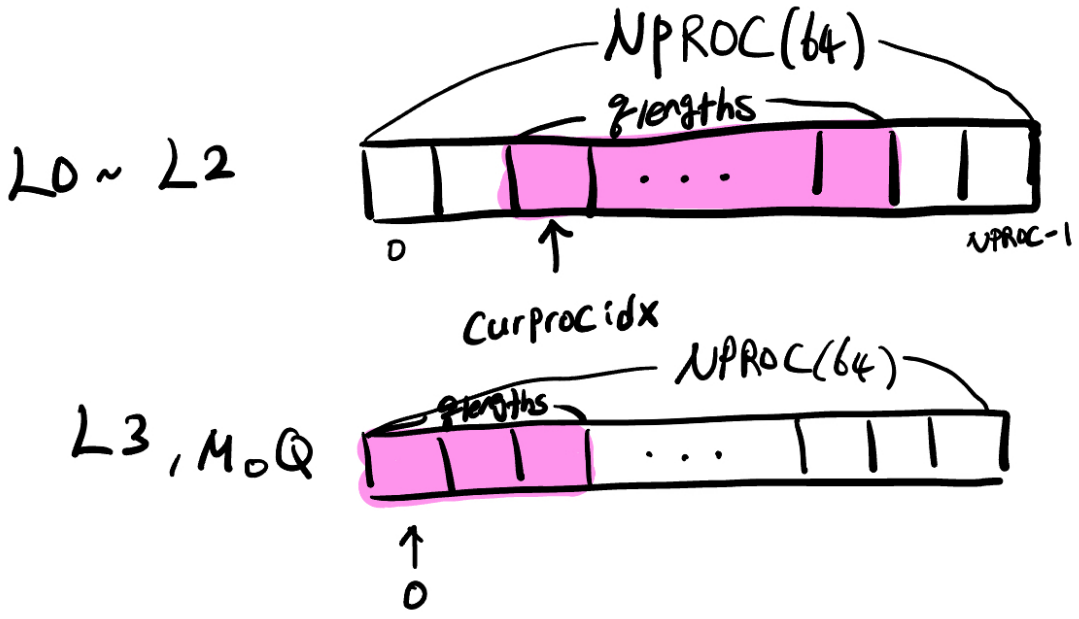
OS Project02 - Wiki

2019040591
컴퓨터소프트웨어학부
박진수

Design:

제가 구현한 큐의 작동 방식을 간단히 그림으로 나타내면 아래와 같습니다.



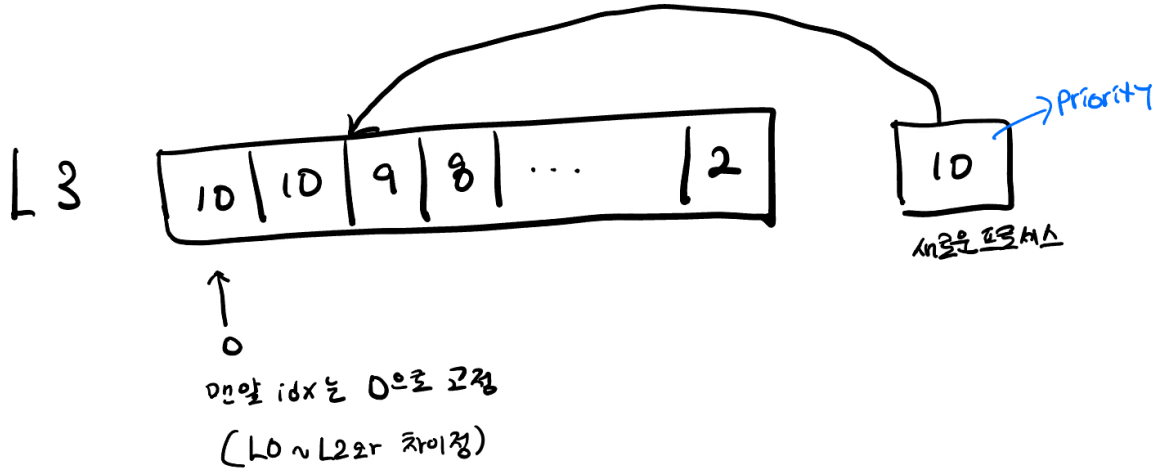
먼저, 저의 경우 L0~L2를 동일한 로직으로 구현하였고, L3와 MoQ를 유사한 로직으로 구현하였습니다.

L0~L2큐의 경우 Round-Robin 정책을 따릅니다. 각 큐에서는 먼저 들어온 순서대로 프로세스가 스케쥴링 되며, 해당 큐에서 프로세스가 주어진 시간(Time quantum) 안에 실행이 완료되지 못한 경우, 그보다 낮은 우선순위의 큐(L0→L3의 순서)로 내려갑니다.

xv6 운영체제의 ptable(모든 프로세스들을 관리)에서 다룰 수 있는 프로세스의 수가 최대 NPROC(64)개이므로, 먼저 각각의 큐들을 크기 NPROC짜리 proc*(프로세스를 가리키는 포인터) 배열로 설정했습니다. 그리고 현재 담겨있는 프로세스들 중 가장 왼쪽 프로세스(가장 먼저 들어온 프로세스)의 인덱스를 curprocidx 변수에, 그리고 들어있는 프로세스의 총

개수를 qlengths 변수에 저장하여 위의 분홍색 하이라이트 된 부분처럼 관리하게끔 하였습니다. 예를 들어 스케쥴러가 L0큐에서 프로세스를 실행시키면 가장 먼저 들어온 프로세스가 빠져나가 curprocidx를 1 증가시키고, fork()등의 이유로 프로세스가 L0큐에 들어오면 qlengths를 1 증가시키는 식입니다. 그리고 물론 해당 큐에서 다 실행되지 못한 프로세스는 우선순위가 낮은 아래의 큐의 맨 뒤에 붙여지고, 그 큐의 qlength는 1 증가합니다.

L3큐의 경우는 priority가 높은 프로세스가 먼저 실행됩니다. 그래서 위의 L0~L2처럼 먼저 들어온 순서 관계가 유지되지 않기 때문에, 제일 먼저 실행될 프로세스의 인덱스는 0으로 고정하고, 프로세스가 들어올 때마다 재정렬하는 방식으로 구현했습니다. 그리고 프로세스의 priority가 같은 경우 먼저 들어온 프로세스를 먼저 실행시키는게 fairness 측면에서 옳다고 생각하여, 아래 그림과 같이 새로운 프로세스가 들어올 때마다 제일 오른쪽(낮은 priority) 프로세스들 부터 priority 값을 순서대로 비교해가면서 새로 들어온 프로세스의 priority가 같거나 작아지는 위치에 삽입되도록 하였습니다.(비교된 프로세스들은 한칸씩 오른쪽으로 밀려난 상태입니다.)



MoQ의 경우, 먼저 들어온 순서대로 왼쪽부터 담기게 됩니다. 제일 먼저 들어온 프로세스가 0번 인덱스에 담기며, 해당 프로세스가 스케쥴러에 의해 선택되어 실행되어 빠져나가면, 오른쪽의 프로세스들이 한칸씩 왼쪽으로 이동하여 다시 0번 인덱스부터 담기는 형식으로 재배열됩니다.

아래는 MLFQ와 MoQ의 구현을 위한 구조체입니다.

1. struct proc (proc.h)

```
struct proc {
  ...
  // For Project 02
  int qnum;                    // Indicates which queue is it
  uint usedtq;                 // Used time quantum
```

```
  uint priority;               // Priority for L3 queue
  uint ismoq;                  // wheter if a process is in m
};
```

각 프로세스를 나타내는 구조체로, qnum의 경우 해당 프로세스가 어떤 큐에 위치해있는지에 대한 정보를 담습니다. 예를 들어 L0의 경우 0, MoQ의 경우 99를 담게 됩니다. usertq의 경우 해당 프로세스가 사용한 time quantum 정보를 담습니다. trap.c의 trap()에서 tick이 1씩 증가하는 부분에서 RUNNING인 경우 이 값을 1 증가시키게끔 구현하였습니다. priority의 경우 L3에서만 사용되는 정보로, 각 프로세스의 우선순위를 나타냅니다. ismoq의 경우 해당 프로세스가 MoQ에 위치해 있는지 여부를 나타냅니다.

2. struct ptable (proc.h)

```
typedef struct ptable{
  struct spinlock lock;
  struct proc proc[NPROC];

  // Multi Level Feedback Queue structure
  struct proc* queues[NQUEUE][NPROC];  // Process queues (4 l
  uint qlengths[NQUEUE];               // Count of processes
  uint timequantums[NQUEUE];            // Time quantum of ea
  uint curprocidx[NQUEUE];              // Current idx in eac

  struct proc* moq[NPROC];  // MOQ
  uint moqlength;           // length of moq
  int ismonopolized;        // if monopolize() is on
} ptable_t;

extern ptable_t ptable;
```

원래 MLFQ와 MoQ와 관련한 구조체를 따로 작성하였었는데, 이처럼 구현하고 각각의 lock을 acquire하고 release 하는 부분에서 deadlock이 발생하여, ptable 구조체에 MLFQ와 MoQ 구현에 필요한 데이터들을 포함시켜 작성하였습니다. 예를 들어 L0를 L1로 옮기는 과정에서 proc의 상태를 바꾸고, MLFQ 구조체의 멤버 정보도 바꾸어야 하는데, 이때 ptable과 mlfq에 모두 lock을 걸게 되면 문제가 발생했던 것 같습니다. 따라서 중복하여 lock을 걸게 되는 상황을 피하려 위처럼 구현하였습니다. 또한 ptable이 trap.c와 같은 다른 파일들에서도 사용되어야 하기에 proc.c에 있던 이부분을 proc.h로 옮겼고, extern으로 설정하였습니다.

struct proc* queues[NQUEUE][NPROC] 의 경우, L0~L3의 큐를 담는(실제로는 포인터) 2차원 배열로 구현하였습니다. 그래서 NQUEUE는 4, NPROC는 64로 총 크기는 256입니다. 앞서 언급했듯이, NPROC이 64라서 이 중 최대 64칸만 사용됩니다. uint qlengths[NQUEUE]의 경우, 각 큐(L0~L3)에 몇개의 프로세스들이 들어가 있는지에 대한 정보를 담는 배열입니다. uint timequantums[NQUEUE]의 경우 각 큐에 할당된 time quantum이 몇인지에 대한 정보를 담습니다. 명세와 같이, 각각 2, 4, 6, 8의 값이 담깁니다. uint curprocidx[NQUEUE]의 경우 앞서 설명드렸던 바와 같이, 각 큐에서 제일 먼저 스케쥴링 될 프로세서의 인덱스를 담습니다. L3의 경우 항상 제일 앞의 인덱스를 0으로 고정하기에, curprocidx의 값은 사용되지 않습니다.

MoQ의 경우, 각 프로세스들은 struct proc* moq[NPROC]에 담깁니다. MoQ도 마찬가지로, 제일 앞의 프로세스는 0번 인덱스로 고정되며, moqlength는 해당 MoQ에 담겨있는 프로세스의 개수를 나타냅니다. ismonopolized의 경우 monopolize() 시스템콜이 호출되면 1, 호출되지 않은 상황이면 0이 담기는 flag 값으로, 스케쥴러에서 이 값에 따라 다른 방식으로(MLFQ 또는 MoQ) 스케쥴링을 진행합니다.

위의 내용이 각 큐들의 기본적인 설계 방식이며, 세부적인 동작이나 코드들은 아래의 Implementation 부분에서 설명하도록 하겠습니다.

Implementation:

1. trap.c

```
void
trap(struct trapframe *tf)
{
    ...
    switch(tf->trapno){
    case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      // For Project 02
      if(myproc() != 0 && myproc()->state == RUNNING){
        myproc()->usedtq += 1;  // Increment timequantum of r
      }
      wakeup(&ticks);
      release(&tickslock);
```

```
    }
    lapiceoi();
    break;
    ...
}
```

위 부분은 타이머 인터럽트를 핸들링하는 부분입니다. 타이머 인터럽트가 발생할 때 마다 위의 코드가 실행되게 되며, global tick인 ticks 뿐만 아니라, 현재 CPU에 할당된 프로세스가 있고, 그 프로세스의 state가 RUNNING일 경우, 해당 프로세스의 usedtq를 1증가시키는 코드를 추가했습니다. 이 코드를 추가함으로 인해, 해당 프로세스가 얼마만큼의 timequantum을 사용했는지 저장할 수 있고, trap()의 아랫 부분에서 timequantum을 다 사용했을 경우 해당 프로세스가 CPU를 반납하도록 했습니다.

```
// Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to ch
  // Also, if a process is in moq, timer interrupt doesn't ha
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER && myproc()->ismoq == 0){
      if(myproc()->usedtq == ptable.timequantums[myproc()->qn
        switch(myproc()->qnum){  // the queue number where my
          // 스케쥴러에서 골라질 때 이미 레디큐에서 빼놔서 집어넣기만 하
          case 0: // L0 큐에서 실행되던 프로세스인 경우
            if(myproc()->pid % 2 == 1){ // pid가 홀수인 프로세스
              putintoL1(myproc());
            } else{  // pid가 짝수인 프로세스들은 L2 큐로 내려가고,
              putintoL2(myproc());
            }
            break;
          case 1:  // L1 큐에서 실행되던 프로세스인 경우
            putintoL3(myproc());
            break;
          case 2:  // L2 큐에서 실행되던 프로세스인 경우
            putintoL3(myproc());
            break;
          case 3: // L3 큐에서 실행되던 프로세스인 경우
            // TODO: priority를 1 감소시키고, timequantum초기화.
            if(myproc()->priority > 0)  // priority가 0에서는 더
              myproc()->priority--;
```

```
            myproc()->usedtq = 0;
            // 스케쥴러에서 골라졌을때(RUNNING)이미 해당 레디큐에서 빠
            putintoL3(myproc());
            break;

          default:
            panic("invalid pid value!!(trap.c)");
          }
          yield();  // give up the CPU held to myproc()
        }
      }
```

위의 코드는 trap()의 아랫부분으로, 현재 프로세스가 RUNNING state이고, monopolized() 상태가 아닌 상황에서, 해당 프로세스가 주어진 time quantum을 다 사용했을 경우, 해당 프로세스를 낮은 우선순위의 큐로 내리거나, L3 프로세스의 경우 priority를 1 감소시키는 부분을 구현한 것입니다. 또한 L0 큐에서 프로세스가 time quantum을 다 사용했을 경우, 해당 프로세스의 pid를 확인하여 홀수일 경우, L1큐에 집어넣고, 짝수일 경우 L2 큐에 집어넣도록 하였습니다. 해당 프로세스가 기존 큐에서 빠지는 부분은, proc.c의 scheduler() 부분에서 구현되어 있습니다. 그리고 해당 처리를 완료하고 난 이후, yield()를 통해 해당 프로세스가 CPU를 반납하도록 하였습니다. 그리고 추가적으로, 처음에는 영어로 주석을 작성하면서 진행했으나, 코드가 많아지면서 제가 한눈에 이해하기가 어려워 중간부터 한글 주석으로 작성하였기에 언어가 섞여있는 점 이해 부탁드립니다.

```
// Priority boosting
  // Global tick이 100ticks가 될 때 마다 모든 프로세스들을 L0 큐로 자
  // moq가 비어있어야만 priority boosting 발생
  if(tf->trapno == T_IRQ0+IRQ_TIMER && ticks % 100 == 0 && pt
    // 먼저 L0에 있는 프로세스들의 tq 0으로 초기화
    int i;
    uint curprocidxL0 = ptable.curprocidx[0]; // L0큐의 첫 프로
    for(i = 0; i < ptable.qlengths[0]; i++){
      ptable.queues[0][(curprocidxL0+i)%NPROC]->usedtq = 0; /
    }
    // L1 차례
    uint curprocidxL1 = ptable.curprocidx[1]; // L1큐의 첫 프로
    uint lengthL1 = ptable.qlengths[1]; // L1 큐의 프로세스의 개
    for(i = 0; i < lengthL1; i++){
```

```
      ptable.queues[1][(curprocidxL1+i)%NPROC]->usedtq = 0; /
      ptable.queues[1][(curprocidxL1+i)%NPROC]->qnum = 0; //
      ptable.queues[0][(curprocidxL0+ptable.qlengths[0])%NPRO
      ptable.qlengths[0]++;
      ptable.qlengths[1]--;
    }
    if(ptable.qlengths[1] != 0)  // for debugging
      panic("Size of L1 not 0 when gloabl boosting!");
    // L2 차례
    uint curprocidxL2 = ptable.curprocidx[2]; // L2프로세스의 첫
    uint lengthL2 = ptable.qlengths[2];  // L2 큐의 프로세스의 가
    for(i = 0; i < lengthL2; i++){
      ptable.queues[2][(curprocidxL2+i)%NPROC]->usedtq = 0; /
      ptable.queues[2][(curprocidxL2+i)%NPROC]->qnum = 0; //
      ptable.queues[0][(curprocidxL0+ptable.qlengths[0])%NPRO
      ptable.qlengths[0]++;
      ptable.qlengths[2]--;
    }
    if(ptable.qlengths[2] != 0)  // for debugging
      panic("Size of L2 not 0 when gloabl boosting!");
    // L3 차례
    // 어차피 L3는 첫 인덱스가 0임
    uint lengthL3 = ptable.qlengths[3];  // L3 큐의 프로세스의 가
    for(i = 0; i < lengthL3; i++){
      ptable.queues[3][i]->usedtq = 0; // curprocidxL3부터 프로
      ptable.queues[3][i]->qnum = 0; // L0 소속으로 변경
      ptable.queues[0][(curprocidxL0+ptable.qlengths[0])%NPRO
      ptable.qlengths[0]++;
      ptable.qlengths[3]--;
    }
    if(ptable.qlengths[3] != 0)  // for debugging
      panic("Size of L3 not 0 when gloabl boosting!");
  }
```

위의 부분은 trap()에서 더 아래 부분으로, global tick인 ticks 변수를 100의 배수가 될 때마다 모든 RUNNABLE 프로세스들을 L0큐로 이동시키는 부분을 구현한 것입니다. 마찬가지로 monopolized 상태인 경우 priority boosting이 동작하지 않기에 그 부분도 if문에 추가하였습니다. 그리고 위의 경우에도 fairness를 위해, L1큐, L2큐, L3큐 순서로 L0로 이동

시켜주었습니다. L0~L2큐의 경우, 프로세스들이 curprocidx 부터 담겨있었으므로, 해당 부분 부터 각 큐에 담겨있는 프로세스의 개수만큼 옮겨주었고, 옮길 때 마다 빠지는 큐에서는 qlength(해당 큐에 담겨있는 프로세스의 개수)를 1씩 감소 시키고, 옮겨지는 큐의 qlength는 1씩 증가시켜 주었습니다. 다 옮기고 나서 qlength가 0이 되지 않는 상황인 경우(의도되지 않은 상황), panic으로 오류가 발생하였음을 나타내었습니다.

2. proc.c

proc.c 파일의 경우 프로세스 관리와 관련된 주요 커널 함수들이 들어있습니다.

```
ptable_t ptable;
```

위의 부분은 proc.h에서 extern으로 설정한 ptable에 실제 메모리를 할당해주는 부분입니다.

```
void
pinit(void)
{
  int i, j;

  for (i = 0; i < NQUEUE; i++){
    ptable.qlengths[i] = 0;  // initialize number of processe
    ptable.timequantums[i] = 2 * (i + 1);  // set time quantu
    ptable.curprocidx[i] = 0;  // set idx to search as 0
    for (j = 0; j < NPROC; j++){
      ptable.queues[i][j] = 0;  // set each as NULL
    }
  }
  // Initialize moq related members of mlfq struct
  for(i = 0; i < NPROC; i++){
    ptable.moq[i] = 0;
  }
  ptable.moqlength = 0;
  ptable.ismonopolized = 0;
```

```
  initlock(&ptable.lock, "ptable");
}
```

위의 부분은 ptable을 초기화해주는 부분으로, 저는 ptable 구조체에 MLFQ, MoQ와 관련된 멤버들을 모두 포함시켰기에 초기화 시켜주었습니다. 이 pinit() 부분은 main.c의 main()에서 실행이 되게 됩니다. 여기서 2, 4, 6, 8의 순서로 time quantum이 L0~L3에 할당됩니다.

```
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size

  p = allocproc(); // 여기서 ptable내의 빈 공간에 들어감

  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_init
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  // this assignment to p->state lets other cores
  // run this process. the acquire forces the above
  // writes to be visible, and the lock is also needed
```

```
  // because the assignment might not be atomic.
  acquire(&ptable.lock);

  p->state = RUNNABLE;
  // For project 02
  p->usedtq = 0; // initialize used timequantum of the proces
  p->priority = 0; // initialize priority of the process to 0
  p->ismoq = 0; // initialize ismoq to 0 since it's not in mo

  putintoL0(p);  // Insert into L0 queue

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

위의 부분은 첫 유저 프로세스가 생성되는 부분입니다. 이 프로세스의 경우에도 MLFQ에 들어가야 하므로, 해당 프로세스의 state를 RUNNABLE로 바꾸는 부분 아래에, 해당 프로세스의 usedtq 변수, priority 변수, ismoq 변수 등, 제가 proc struct에서 추가로 만들었던 멤버 변수들을 초기화했습니다. 그리고 해당 프로세스를 L0 큐에 삽입하기 위해, putintoL0(p);를 호출하였습니다. proc가 위치해 있는 큐를 나타내는 qnum 멤버 변수는 putintoL0(p)에서 설정됩니다.

```
int
fork(void)
{
    ...
    np->state = RUNNABLE;

  // Added for Project 02
  np->usedtq = 0; // initialize used timequantum of the proce
  np->priority = 0; // initialize priority of the process to
  np->ismoq = 0;  // initialize ismoq as 0 since it's not in
  putintoL0(np);  // add the new process np into L0 queue

  release(&ptable.lock);
```

```
  return pid;
}
```

새로운 프로세스를 생성할 때 사용되는 fork()의 경우에도 userinit()와 마찬가지로 동일한 코드를 추가한 후, L0큐에 삽입되도록 하였습니다. 기존 xv6의 스케쥴링 방식은 일반적인 RR 방식이었기에, 해당 프로세스의 state를 RUNNABLE로 설정만 하면, scheduler 에서 ptable을 무한히 순회하며 RUNNABLE프로세스를 실행시키는 방식이었기에 위처럼 state만 변경시키면 되었는데, 과제의 경우 MLFQ를 순회하며 동작해야 하기에 위처럼 추가적인 작성이 필요했습니다.

```
int
wait(void)
{
...
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    ...
        if(p->state == ZOMBIE){
            ...
            p->state = UNUSED;
        // For Project 02
        // Clean up the added members of proc state
        p->qnum = 0;
        p->usedtq = 0;
        p->priority = 0;
        p->ismoq = 0;
        release(&ptable.lock);
        return pid;
      }
     ...
}
```

그리고 위의 경우에는 과제의 핵심 부분은 아니지만, ZOMBIE 상태인 자식 프로세스를 부모 프로세스가 발견했을 경우, 해당 프로세스의 상태를 UNUSED의 상태로 설정하고 proc의 멤버들을 초기화하는 과정에서 제가 추가한 멤버 변수도 초기화하는 부분입니다.

그리고 아래의 scheduler()가 이번 과제에서 프로세스 스케쥴링을 담당하는 주요 부분입니다.

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  for(;;){
    // Enable interrupts on this processor.
    sti();

    acquire(&ptable.lock);
    // Loop over ptable.queues looking for process to run.
    if(ptable.ismonopolized == 0){ // monopolize() 가 꺼져있으면
      int i;
      for(i = 0; i < NQUEUE; i++){ // loop through L0 ~ L3
        if(ptable.qlengths[i] == 0) // L0 큐부터 살펴보면서 빈큐는
          continue;
        if(i == 3){ // L3의 경우 삽입시 그냥 priority가 높은 프로세
          p = ptable.queues[i][0]; // L3 큐의 맨 앞의 프로세스가 
          // 이론 상, 스케쥴러가 해당 프로세스를 선택했으면 Ready 큐를
          int j;
          for(j = 1; j < ptable.qlengths[3]; j++){ // 1번 인덱스
            ptable.queues[3][j-1] = ptable.queues[3][j];
          }
          ptable.qlengths[3]--; // 일단 현재 실행된 프로세스가 빠져
          // timer interrupt가 발생하면(아직 안끝났다는 의미이니, 거기

        }else{  // L0, L1, L2 큐의 경우 RR 정책을 따르므로 curproc
          // 프로세스가 들어있는 큐를 L0에서부터 시작해 찾았다면 해당 큐
          p = ptable.queues[i][ptable.curprocidx[i]]; // 해당

          ptable.curprocidx[i] += 1; // 다음에는 그 다음번 프로세스
          if(ptable.curprocidx[i] == NPROC)
            ptable.curprocidx[i] = 0;  // 리스트의 범위를 벗어났으
          // 마찬가지로 여기서도 스케쥴러에서 골라지면 해당 레디큐를 빠져
```

```
          ptable.qlengths[i]--;  // 해당 큐의 프로세스 개수를 하나
        }
        // 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
        c->proc = 0;
      }  // end of for(i = 0; i < NQUEUE; i++)

    } else{ // if(ptable.ismonopolized == 1) monopolize()가 켜
      if(ptable.moqlength > 0){
        p = ptable.moq[0]; // moq의 맨 앞 프로세스를 실행할 프로세스
        int i;
        for(i = 1; i < ptable.moqlength; i++){ // 그 오른쪽 프로
          ptable.moq[i - 1] = ptable.moq[i];
        }
        ptable.moqlength--; // moq의 길이 1 감소

        // 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
```

```
        c->proc = 0;
        if(ptable.moqlength == 0){  // 모든 프로세스들이 종료된 경우
          release(&ptable.lock);
          unmonopolize();
          acquire(&ptable.lock);
        }
      }else {  // MoQ의 길이가 0인 상태에서 monopolize가 실행되었으면
        release(&ptable.lock);
        unmonopolize();
        acquire(&ptable.lock);
      }
    }
    release(&ptable.lock);
  }
}
```

먼저, 스케쥴러는 xv6가 실행될 동안 계속 동작해야하기 때문에, 기존의 scheduler()와 마찬가지로 무한 for문이 필요합니다.

그리고 이 무한 for 문 안의 구조를 크게 살펴 보면 아래와 같이 볼 수 있습니다.

```
if(ptable.ismonopolized == 0){ // monopolize() 가 꺼져있으면
    ...
}else{ // if(ptable.ismonopolized == 1) monopolize()가 켜져있으면
...
}
```

scheduler의 매 iteration 마다 ptable 구조체의 ismonopolized 멤버변수의 값을 확인하여, 해당 값이 0이면, 즉 monopolize() 가 꺼져있는 상태이면 일반 MLFQ scheduling을 수행하는 위의 if문이 수행되고, 해당 값이 1인 경우 MoQ 스케쥴링이 구현되어 있는 아래의 else문이 실행됩니다.

먼저 ptable.ismonopolized == 0 일때 수행되는 MLFQ 스케쥴링 부분을 살펴보겠습니다.

```
if(ptable.ismonopolized == 0){ // monopolize() 가 꺼져있으면
      int i;
      for(i = 0; i < NQUEUE; i++){ // loop through L0 ~ L3
        if(ptable.qlengths[i] == 0) // L0 큐부터 살펴보면서 빈큐는
          continue;
```

```
      if(i == 3){ // L3의 경우 삽입시 그냥 priority가 높은 프로세
        p = ptable.queues[i][0]; // L3 큐의 맨 앞의 프로세스가 
        // 이론 상, 스케쥴러가 해당 프로세스를 선택했으면 Ready 큐를
        int j;
        for(j = 1; j < ptable.qlengths[3]; j++){ // 1번 인덱
          ptable.queues[3][j-1] = ptable.queues[3][j];
        }
        ptable.qlengths[3]--; // 일단 현재 실행된 프로세스가 빠져
        // timer interrupt가 발생하면(아직 안끝났다는 의미이니, 거

      }else{  // L0, L1, L2 큐의 경우 RR 정책을 따르므로 curproc
        // 프로세스가 들어있는 큐를 L0에서부터 시작해 찾았다면 해당 큐
        p = ptable.queues[i][ptable.curprocidx[i]]; // 해당

        ptable.curprocidx[i] += 1; // 다음에는 그 다음번 프로세스
        if(ptable.curprocidx[i] == NPROC)
          ptable.curprocidx[i] = 0;  // 리스트의 범위를 벗어났으
        // 마찬가지로 여기서도 스케쥴러에서 골라지면 해당 레디큐를 빠져
        ptable.qlengths[i]--;  // 해당 큐의 프로세스 개수를 하나
      }
      // 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
      c->proc = 0;
    }  // end of for(i = 0; i < NQUEUE; i++)

  }
```

```
    else{ // if(ptable.ismonopolized == 1) monopolize()가 켜져
    ...
```

먼저 L0~L3의 순으로 스케쥴링 우선순위가 높기 때문에, for 문으로 0부터 3까지 순회하며 각 큐의 길이가 0인지 아닌지 확인합니다. 0이면 해당 큐가 비어있다는 뜻이므로 다음 큐의 길이를 조사합니다. 큐의 길이가 0이 아닌 경우, 즉 비어있지 않으면 L3인지 아니면 나머지 L0~L2큐인지 여부로 분기됩니다. 먼저 L0~L2 큐의 길이(담겨있는 프로세스의 개수)가 0이라 L3큐가 스케쥴링 되는 부분을 살펴보겠습니다.

```
if(i == 3){ // L3의 경우 삽입시 그냥 priority가 높은 프로세스가 배열
          p = ptable.queues[i][0]; // L3 큐의 맨 앞의 프로세스가 
          // 이론 상, 스케쥴러가 해당 프로세스를 선택했으면 Ready 큐를
          int j;
          for(j = 1; j < ptable.qlengths[3]; j++){ // 1번 인덱
            ptable.queues[3][j-1] = ptable.queues[3][j];
          }
          ptable.qlengths[3]--; // 일단 현재 실행된 프로세스가 빠져
          // timer interrupt가 발생하면(아직 안끝났다는 의미이니, 거

        }
```

먼저 L3큐의 제일 앞부분, 즉 인덱스 0에 위치해 있던 프로세스의 포인터를 p라는 변수에 담습니다. 이 p는 뒷부분에서 RUNNING state로 전환된 후, 실행이 되게 됩니다. 그리고 p에 담고 난 이후에는 그 프로세스가 L3 큐에서 빠져나갔다고 생각하고, 그 오른쪽에 있던 프로세스들을 한칸씩 왼쪽으로 당겨줍니다. 그래서 다시 0번 인덱스 부터 프로세스들이 채워지게 됩니다. 이후 해당 L3큐의 길이를 나타내는 qlengths[3] 변수를 1 감소시켜줍니다.

아래의 부분은 L0~L2큐가 비어있지 않은 경우 실행되는 부분으로, 실제로 동작할 때는 위의 L3큐 스케쥴링 보다 먼저 실행됩니다.

```
else{  // L0, L1, L2 큐의 경우 RR 정책을 따르므로 curprocidx 사용
          // 프로세스가 들어있는 큐를 L0에서부터 시작해 찾았다면 해당 큐
          p = ptable.queues[i][ptable.curprocidx[i]]; // 해당

          ptable.curprocidx[i] += 1; // 다음에는 그 다음번 프로세스
          if(ptable.curprocidx[i] == NPROC)
```

```
            ptable.curprocidx[i] = 0;  // 리스트의 범위를 벗어났으
          // 마찬가지로 여기서도 스케쥴러에서 골라지면 해당 레디큐를 빠져
          ptable.qlengths[i]--;  // 해당 큐의 프로세스 개수를 하나
        }
```

앞서 언급드렸듯이, L0~L2는 curprocidx라는 변수에 논리상 제일 왼쪽(먼저 스케쥴링 되어야 할) 프로세스의 인덱스 정보를 담게 됩니다. 그래서 curprocidx 인덱스에 위치하고 있던 프로세스의 포인터를 p에 담고, curprocidx 값을 1 증가시켜 주어 그 바로 오른쪽에 있던 프로세스에 위치시켜줍니다. 그래서 다음번에 스케쥴링 될 때는 이 프로세스가 제일 먼저 실행되게 됩니다. 그리고 해당 큐에 담겨있는 프로세스의 개수를 담고있는 qlengths 변수를 1 감소시켜 줍니다. 그리고 위처럼 curprocidx와 qlengths를 가지고 큐의 기능을 구현하도록 하는 경우, 실제 해당 큐를 담고있는 배열의 크기를 넘어가는 상황이 발생할 수 있습니다. 이 경우 나머지 연산을 활용하여, 배열의 끝을 넘어설 경우 0번 인덱스에 채워지고, 그 다음에는 1, 2, 3, .. 이런 식으로 인덱스가 채워지게 구현하였습니다.

그 이후에는 기존의 scheduler()에서 동작하던 것 처럼, 프로세스의 상태를 RUNNING으로 바꾸어주고 그 프로세스의 context로 switch가 일어나게 됩니다.

그 다음으로는 ptable.ismonopolized == 1 인 경우입니다. 이 경우 MoQ 스케쥴링이 동작하게 됩니다.

```
else{ // if(ptable.ismonopolized == 1) monopolize()가 켜져있으면
      if(ptable.moqlength > 0){
        p = ptable.moq[0]; // moq의 맨 앞 프로세스를 실행할 프로세스
        int i;
        for(i = 1; i < ptable.moqlength; i++){ // 그 오른쪽 프로
          ptable.moq[i - 1] = ptable.moq[i];
        }
        ptable.moqlength--; // moq의 길이 1 감소

        // 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
        c->proc = 0;
        if(ptable.moqlength == 0){  // 모든 프로세스들이 종료된 경
          release(&ptable.lock);
          unmonopolize();
          acquire(&ptable.lock);
        }
      }else {  // MoQ의 길이가 0인 상태에서 monopolize가 실행되었으면
        release(&ptable.lock);
        unmonopolize();
        acquire(&ptable.lock);
      }
    }
```

MoQ의 경우 L3와 마찬가지로 0번 인덱스부터 프로세스들이 채워지게 됩니다. 위의 경우 MoQ의 길이가 0보다 클 경우 주요 기능이 실행되고, MoQ의 길이가 0일 때 monopolize()가 실행되었을 경우 바로 unmonopolize()가 호출되어 global tick이 0으로 초기화 되고 다시 ismonopolized 값을 0으로 바꾸도록 만들었습니다. 하지만 예상되는 주요 시나리오는 아닙니다. 실제로 예상되는 상황은 MoQ의 길이가 0보다 큰 상황에서 monopolize()가 호출된 상황으로, 이 경우에는 MoQ의 제일 앞, 즉 0번 인덱스의 프로세스부터 p에 담기게 되며, 담긴 이후에는 그 오른쪽 프로세스들을 한 칸씩 앞으로 당기며, MoQ의 길이 값인 ptable.moqlength 값도 1 감소시켜 줍니다. p에 실행될 프로세스가 담긴 이후에는 이전과 마찬가지로 RUNNABLE로 state가 바뀌고 context switch가 일어나 해당 프로세스가 실행되게 됩니다. 그리고 trap.c의 trap()에서 설정된 바와 같이, MoQ스케쥴링이 일어나는 경우에는 timequantum에 상관없이 해당 프로세스가 끝날 때 까지 yield()를 당하지 않습니다. 그리고 ismonopolized 값은 변경되지 않으므로, scheduler()의 무한 for문에서 계속 이 MoQ 스케쥴링 부분으로 진입하게 됩니다. 그리고 계속 MoQ 스케쥴링을 처리하다 결국 그 길이가 0이 되는 경우, unmonopolize()를 실행하여 ismonopolized 값을 0으로 바꾸고, global ticks 값도 0으로 초기화합니다.

그리고 아래는 wakeup1(void *chan) 함수로, chan이라는 wait channel에서 기다리고 있는(SLEEP) 프로세스들을 깨우는 역할을 합니다.

```
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;
      // For Project 02
      // 깨어났으면 다시 sleep 전 해당 그 큐로 보내는 정책을 사용할 예정
      if(p->ismoq == 1){ // moq의 프로세스였던 경우
        int i;
        for(i = ptable.moqlength - 1; i >= 0; i--){
          ptable.moq[i + 1] = ptable.moq[i]; // 한칸씩 오른쪽으로
        }
        // moq의 맨 앞자리에 다시 집어넣기
        ptable.moq[0] = p;
        ptable.moqlength++; // moq의 길이 1 늘리기
      } else { // 일반 mlfq의 프로세스였던 경우
        // usedtq는 각 putintoL{qnum}에서 0으로 다시 초기화해줌. (다
        int qnum = p->qnum; // sleep 이전에 위치해있던 mlfq의 큐 번
        switch(qnum){
          case 0:
            putintoL0(p);
            break;
          case 1:
            putintoL1(p);
            break;
          case 2:
            putintoL2(p);
            break;
          case 3:
            putintoL3(p);
            break;
          case 99:
            ptable.moq[ptable.moqlength] = p;
            ptable.moqlength++;
            break;
          default:
            panic("unexpected qnum!");
```

```
        }
      }
    }
  }
}
```

SLEEP상태로는 RUNNING인 상태에서만 바뀌게 되는데, 해당 프로세스의 상태가 RUNNING인 상태에서는 어느 어느 READY큐(MLFQ, MoQ)에도 들어있지 않게 됩니다. 그래서 SLEEP에서 깨어난 이후, 원래 있던 큐로 다시 집어 넣어주는 과정이 위의 과정입니다. 원래 MoQ에 있다가 실행이 되던 도중 SLEEP 하게된 프로세스의 경우, 다시 맨 MoQ의 맨 앞에 집어 넣어주어 바로 다음번 스케쥴링에서 실행될 수 있도록 하였습니다. 그리고 MLFQ에 있던 프로세스의 경우, p→qnum 값에 원래 어느 큐에 있었는지에 대한 정보를 담고 있어서 이 정보를 바탕으로 각 큐에 집어넣게 됩니다. 이 경우에는 대기큐의 제일 앞에 집어넣는 것이 아니라, 해당 큐의 정책에 따라 들어가게 하였습니다.

여기서부터는 제가 직접 추가한 커널 함수들입니다.

먼저 아래는 프로세스를 큐에 집어 넣기 위해 만든 함수입니다.

```
// Additionally added kernel functions by me


// Generic function to append a process into a given Queue le
void _putintomlfq(struct proc *p, int targetqueue)
{
  struct proc *np = p;  // new process to put into L{targetqu

  uint indextoput = (ptable.curprocidx[targetqueue] + ptable.

  ptable.queues[targetqueue][indextoput] = np;  // append poi
  ptable.qlengths[targetqueue] += 1;  // increment num of pro

  np->qnum = targetqueue;  // Set current queue number as tar
  np->usedtq = 0;  // Set used tick value to 0.
}

// Append a new process into L0 queue
```

```
void
putintoL0(struct proc *p)
{
  _putintomlfq(p, 0);
}

// Append a new process into L1 queue
void
putintoL1(struct proc *p)
{
  _putintomlfq(p, 1);
}

// Append a new process into L2 queue
void
putintoL2(struct proc *p)
{
  _putintomlfq(p, 2);
}
```

L0, L1, L2큐에 프로세스를 집어넣을 때, 모두 동일한 로직을 사용하므로 _putintomlfq()라는 제너릭 함수를 만들어 두었습니다. 그리고 L0, L1, L2큐 에 집어넣는 putintoL0(), putintoL1(), putintoL2()를 정의할 때 모두 해당 함수를 호출하는 방식으로 구현하였습니다. _putintomlfq() 함수의 경우, 집어넣을 프로세스인 struct proc *p와 집어넣을 큐의 번호(L0이면 0)을 인자로 받도록 하였습니다. 그리고 curprocidx + qlength 위치가 지금까지 채워져있던 프로세스의 바로 오른쪽 빈 공간이기 때문에, 그곳에 프로세스 p를 집어넣고 qlength를 1 증가시켰습니다. 그리고 해당 프로세스 구조체의 멤버도 바뀐 정보에 맞추어 수정하여 주었습니다.

그리고 아래는 L3 큐에 프로세를 집어넣는 기능을 하는 함수입니다.

```
// Append a new process into L3 queue
// 프로세스간 Priority의 내림차순이 보장되도록 삽입함.
void
putintoL3(struct proc *p)
{
  struct proc *np = p;  // new process to put into L3
  uint priority = np->priority;   // priority of the new proc
```

```
  uint qlength = ptable.qlengths[3];  // number of processes
  int indextoput = 0;
  int i;
  for(i = qlength - 1; i >= 0; i--){  // L3의 맨 오른쪽(뒤) 프로세
    if(priority <= ptable.queues[3][i]->priority){ // 집어넣을
      indextoput = i + 1;
      break;
    }
  }
  // if문에서 안걸렸으면 priority = 0으로 설정한 초기값으로 유지됨. 제일
  // 이제 오른쪽으로 한칸씩 옮겨 indextoput에 빈자리를 만들어아햠(집어넣
  for(i = qlength - 1; i >= indextoput; i--){
    ptable.queues[3][i + 1] = ptable.queues[3][i];
  }
  // indextoput에 생긴 빈 자리에 새로운 프로세스 집어넣기
  ptable.queues[3][indextoput] = np;

  ptable.qlengths[3] += 1;  // increment num of processes in

  np->qnum = 3;  // Set current queue number as 3 (L3)
  np->usedtq = 0;  // Set used tick value to 0.
}
```

Wiki 제일 처음에서 언급했듯이, L3의 경우 프로세스들은 0번 인덱스 부터 담기게 되며, 새로 프로세스가 들어오는 경우, 새로운 프로세스의 priority와 기존 L3큐에 담겨있던 프로세스들의 priority를 맨 오른쪽 부터 비교해가며, 새로운 프로세스의 priority가 작거나 같아지는 부분에 해당 큐를 삽입합니다. 그리고 비교해가는 과정에서 기존에 있던 프로세스들은 한칸씩 오른쪽으로 밀려납니다. 그리고 L3큐에 삽입이 완료된 이후에는, ptable 구조체의 MLFQ 관련 멤버변수와 새로운 프로세스의 proc 구조체의 멤버 변수를 이에 맞게 수정해주었습니다.

그리고 아래는 과제 명세가 요구하는 시스템콜을 구현하기 위해 추가한 커널 함수들입니다. 유저 레벨에서 호출한 시스템콜을 핸들링하는 핸들러 함수들은 sysproc.c에 구현하였고, 밑부분에서 설명하겠습니다. proc.c에 위치하는 아래의 함수들은 sysproc.c의 sys_{함수명} 꼴의 핸들러 함수들에 의해서 호출되는 커널함수들입니다. 구현상 유저프로그램에서 시스템콜을 호출하면, sysproc.c의 핸들러 함수들이 호출되고, 이 핸들러 함수들이 시스템콜 호출시 전달된 인자들의 예외처리를 진행한 후, proc.c 의 커널 함수를 호출하게 됩니다.

먼저 아래의 setpriority()의 경우, pid라는 프로세스 id와 priority 값을 입력받은 이후 ptable의 proc 배열을 순회하며 해당 pid 값을 가진 프로세스를 찾아 priority 값을 전달받은 인자의 priority로 바꿔주는 함수입니다. 해당 프로세스가 존재하지 않을 경우 -1을 반환하고, 해당 작업을 완료했을 경우 0을 반환합니다. 인자인 priority가 가능한 범위가 아닌 경우 -2를 반환하는 부분은 sysproc.c의 sys_setpriority()에서 구현하였습니다.

```
// 특정 pid를 가지는 프로세스의 priority를 설정합니다.
int
setpriority(int pid, int priority)
{
  struct proc *p;
  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != UNUSED && p->pid == pid){
      p->priority = priority;
      release(&ptable.lock);
      return 0;  // priority 설정에 성공한 경우 0을 반환
    }
  }
  release(&ptable.lock);
  return -1;  // 주어진 pid를 가진 프로세스가 존재하지 않는 경우 -1을
}
```

아래는 pid의 값을 가진 프로세스를 MoQ로 이동하는 함수입니다.

자기 자신을 MoQ로 이동시키려고 하는 경우나, 이미 프로세스가 MoQ에 존재하는 경우의 예외처리를 하였고, 만약 pid 값을 가진 프로세스를 발견한 경우, 해당 프로세스가 어느 큐에 속해있었는지에 따라 해당 큐에서 프로세스를 뽑아낸 이후, 오른쪽의 프로세스들을 왼쪽으로 한칸 씩 옮겨주고, 제일 아랫 부분에서 MoQ에 발견한 해당 프로세스를 집어넣도록 구현하였습니다. L0~L2의 경우 프로세스들이 curprocidx 인덱스 부터 담겨있으므로 해당 프로세스를 찾은 이후부터 왼쪽으로 이동하도록 하였습니다. L3에서 찾은 경우, 인덱스가 0부터 시작하므로 0부터 검사를 진행했습니다.

```
// 특정 pid를 가진 프로세스를 MoQ로 이동합니다
int
setmonopoly(int pid)
{
  struct proc *p;
```

```
  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      if(myproc()->pid == p->pid){
        release(&ptable.lock);
        return -4;  // 자기 자신(RUNNING)을 MoQ로 이동시키려 하는 경
      } else if(p->ismoq == 1){  // 이미 MoQ에 들어있던 프로세스인
        release(&ptable.lock);
        return -3;  // 이미 MoQ에 존재하는 프로세스인 경우 -3을 반환
      } else{  // CPU가 한개이므로 내가 실행될 일은 없음 따라서 RUNNI
        switch(p->qnum){
          case 0: { // L0에 있던 프로세스를 빼야함
            uint curprocidx = ptable.curprocidx[0];
            uint qlength = ptable.qlengths[0];

            int i;
            int isfound = 0; // p를 발견했는지 여부
            for(i = 0; i < qlength; i++){ // L0큐에서 p를 찾아야
              if(isfound == 1){  // 이 전에 p를 찾았으면 그 이후부터
                ptable.queues[0][curprocidx + i - 1] = ptable
              }
              if(ptable.queues[0][curprocidx + i]->pid == pid
                if(i == 0) // 만약 curprocidx위치에 있던 프로세스를
                  ptable.curprocidx[0]++;
                isfound = 1;
              }
            }
            ptable.qlengths[0]--;
            break; // case 0 end
          }
          case 1: { // L1에 있던 프로세스를 빼야함
            uint curprocidx = ptable.curprocidx[1];
            uint qlength = ptable.qlengths[1];

            int i;
            int isfound = 0; // p를 발견했는지 여부
            for(i = 0; i < qlength; i++){ // L1큐에서 p를 찾아야
```

```
      if(isfound == 1){  // 이 전에 p를 찾았으면 그 이후부터
        ptable.queues[1][curprocidx + i - 1] = ptable
      }
      if(ptable.queues[1][curprocidx + i]->pid == pid
        if(i == 0) // 만약 curprocidx위치에 있던 프로세스를
          ptable.curprocidx[1]++;
        isfound = 1;
      }
    }
    ptable.qlengths[1]--;
    break; // case 1 end
  }
  case 2: { // L2에 있던 프로세스를 빼야함
    uint curprocidx = ptable.curprocidx[2];
    uint qlength = ptable.qlengths[2];

    int i;
    int isfound = 0; // p를 발견했는지 여부
    for(i = 0; i < qlength; i++){ // L2큐에서 p를 찾아야
      if(isfound == 1){  // 이 전에 p를 찾았으면 그 이후부터
        ptable.queues[2][curprocidx + i - 1] = ptable
      }
      if(ptable.queues[2][curprocidx + i]->pid == pid
        if(i == 0) // 만약 curprocidx위치에 있던 프로세스를
          ptable.curprocidx[2]++;
        isfound = 1;
      }
    }
    ptable.qlengths[2]--;
    break; // case 2 end
  }
  case 3: { // L3에 있던 프로세스를 빼야함
    // L3는 프로세스의 맨 앞(0)부터 실행이 되기때문에 curproc
    uint qlength = ptable.qlengths[3];

    int i;
    int isfound = 0; // p를 발견했는지 여부
    for(i = 0; i < qlength; i++){ // L3큐에서 p를 찾아야
```

```
              if(isfound == 1){  // 이 전에 p를 찾았으면 그 이후부터
                ptable.queues[3][i - 1] = ptable.queues[3][i]
              }
              if(ptable.queues[3][i]->pid == pid){ // index i
                isfound = 1;
              }
            }
            ptable.qlengths[3]--;
            break; // case 3 end
          }
          default:
            panic("invalid qnum in setmonopoly()!");
        }

        // 이 위에까진 MLFQ 구조 수정
        // 이 아래부턴 MoQ 구조 수정
        p->ismoq = 1;
        p->qnum = 99;
        ptable.moq[ptable.moqlength] = p;
        ptable.moqlength++;
        release(&ptable.lock);
        return ptable.moqlength;  //setmonopolize 설정에 성공한
      }
    }
  }
  release(&ptable.lock);
  return -1;  // 주어진 pid를 가진 프로세스가 존재하지 않는 경우 -1을
}
```

마지막으로 monopolize()와 unmonopolize()의 경우, ptable의 ismonopolized 값을 변경해 주었습니다. 이 값은 trap() 과 scheduler()에서 MoQ 스케쥴링인지 여부를 판단하는 데에 사용됩니다. unmonopolized()의 경우 global ticks도 0으로 초기화 하는 기능을 추가했습니다.

```
// MoQ의 프로세스가 CPU를 독점하여 사용하도록 설정합니다.
int
monopolize(void)
{
```

```
  acquire(&ptable.lock);
  ptable.ismonopolized = 1;
  release(&ptable.lock);
  return 0;
}

// 독점적 스케줄링을 중지하고 기존의 MLFQ part로 돌아갑니다.
int
unmonopolize(void)
{
  acquire(&ptable.lock);
  acquire(&tickslock);
  ptable.ismonopolized = 0;
  ticks = 0; // global tick 0으로 초기화
  release(&tickslock);
  release(&ptable.lock);
  return 0;
}
```

3. proc.h

proc.h의 내용은 Wiki의 맨 처음 proc 와 ptable struct를 설명한 부분에서 이미 언급하였고, 그 외의 부분은 아래와 같습니다. proc.c에서 구현한 함수들을 trap.c에서 사용할 일이 생겨 아래처럼 헤더에 해당 함수들의 선언을 해 놓았습니다.

```
// to use these in trap.c
void putintoL0(struct proc *p);
void putintoL1(struct proc *p);
void putintoL2(struct proc *p);
void putintoL3(struct proc *p);

// to use these in proc.c
int setpriority(int pid, int priority);
int setmonopoly(int pid);
int monopolize(void);
int unmonopolize(void);
```

4. sysproc.c

여기에 작성된 함수들은 시스템 콜 핸들러 함수들로, 유저 프로그램에서 시스템 콜을 호출하면 usys.S에서 만들어진 stub 함수들을 통해 미리 설정해둔 각각의 시스템콜에 해당하는 인터럽트를 호출하며, 이 인터럽트 번호가 syscall.c에서 아래의 함수들과 맵핑되어 아래의 핸들러 함수들이 실행되게 됩니다.

```
// Project 02
// Required system calls
// 인자 잘못된 경우 리턴을 원래는 -1을 해야하는데, 명세의 내용이 정해져 있
// 혼동을 방지하기 위해여 -10을 리턴하도록 설정함.
int
sys_yield(void) {
  /**
   * 자신이 점유한 cpu를 양보합니다.
  */
  yield();
  return 0;
}

int
sys_getlev(void)
{
  /**
   * 프로세스가 속한 큐의 레벨을 반환합니다.
   * MoQ에 속한 프로세스인 경우 99를 반환합니다.
  */
  return myproc()->qnum;

}

int
sys_setpriority(void)
{
  /**
   * arguments: (int pid, int priority)
   * 특정 pid를 가지는 프로세스의 priority를 설정합니다.
```

```
   * priority 설정에 성공한 경우 0을 반환합니다.
   * 주어진 pid를 가진 프로세스가 존재하지 않는 경우 -1을 반환합니다.
   * priority가 0 이상 10 이하의 정수가 아닌 경우 -2를 반환합니다.
  */

  int pid, priority;

  // argint fetches the nth 32-bit system call argument
  if(argint(0, &pid) < 0)
    return -10;
  if(argint(1, &priority) < 0)
    return -10;
  if(priority < 0 || priority > 10)
    return -2;  // priority가 0 이상 10 이하의 정수가 아닌 경우 -2를

  return setpriority(pid, priority);  // proc.c
}

int
sys_setmonopoly(void)
{
  /** arguments: (int pid, int password)
   * 특정 pid를 가진 프로세스를 MoQ로 이동합니다. 인자로 독점 자격을 증명
   * 암호가 일치할 경우, MoQ를 반환합니다.
   * 존재하지 않는 포르세스의 pid인 경우 -1을 반환합니다.
   * 암호가 일치하지 않는 경우 -2를 반환합니다.
   * 이미 MoQ에 존재하는 프로세스인 경우 -3을 반환합니다.
   * 자기 자신을 MoQ로 이동시키려 하는 경우 -4를 반환합니다.
  */
  int pid, password;

  // argint fetches the nth 32-bit system call argument
  if(argint(0, &pid) < 0)
    return -10;
  if(argint(1, &password) < 0)
    return -10;

  // 학번을 암호로 입력받음
```

```
  if(password != 2019040591){ // 일치하지 않으면
    return -2;  // 암호가 일치하지 않는 경우 -2를 반환
  } else{
    return setmonopoly(pid);
  }
}

int
sys_monopolize(void)
{
  /**
   * MoQ의 프로세스가 CPU를 독점하여 사용하도록 설정합니다.
  */
  return monopolize();
}

int
sys_unmonopolize(void)
{
  /**
   * 독점적 스케줄링을 중지하고 기존의 MLFQ part로 돌아갑니다.
  */
  return unmonopolize();
}
```

위의 함수들의 경우, 유저 레벨에서 호출한 시스템콜의 인자들이 제대로 들어왔는지 판단하는 예외처리 기능을 수행합니다. 위에서 argint(0, &pid) 함수의 경우, 유저가 시스템콜을 호출할 때 넘겨진 인자들의 배열 중 첫 번째 값을 pid라는 변수에 담는 역할을 합니다. argint(0, &pid)의 리턴 값이 음수인 경우 제대로 인자가 넘겨지지 않았다는 의미이므로 예외처리를 수행했습니다. 그리고 인자들이 정상적인 값으로 판단된 경우, proc.c에 있는 해당하는 커널 함수를 호출하게 하였습니다. 예외적으로 sys_yield()와 sys_getlev()의 경우, 내부적으로 따로 처리할 필요가 없어 바로 yield() 커널함수를 호출하거나, myproc()→qnum을 리턴하도록 했습니다.

5. user.h

```
// For project02
void yield(void);
int getlev(void);
```

```
int setpriority(int pid, int priority);
int setmonopoly(int pid, int password);
void monopolize();
void unmonopolize();
```

유저 프로그램에서는 위의 user.h에서 선언된 함수를 호출하게 되며, 위의 형태가 과제의 명세에서 요구하는 함수의 형태와 정확히 일치합니다. 위의 함수를 호출하면 아래와 같이 usys.S에서 만들어진 stub 함수가 호출되게 됩니다.

6. usys.S

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
...
SYSCALL(yield)
SYSCALL(print_ticks_pid_name)
SYSCALL(getlev)
SYSCALL(setpriority)
SYSCALL(setmonopoly)
SYSCALL(monopolize)
SYSCALL(unmonopolize)
```

위의 경우 각각의 SYSCALL(함수이름)으로 만들어진 stub 함수(Assembly code)는 각 시스템콜에 할당된 번호에 맞는 interrupt를 발생시킵니다. 각 시스템콜에 매핑된 번호는 아래의 파일에 작성되어 있습니다.

7. syscall.h

```
...
#define SYS_yield 25
#define SYS_print_ticks_pid_name 26
#define SYS_getlev 27
```

```
#define SYS_setpriority 28
#define SYS_setmonopoly 29
#define SYS_monopolize 30
#define SYS_unmonopolize 31
```

위처럼 SYS_yield는 25로 매핑되어 있습니다. 그래서 위의 usys.S에서 yield()가 호출되었을 경우, eax레지스터에 SYS_yield에 해당하는 25를 저장시키고, T_SYSCALL이라는 interrupt number를 가진 시스템 인터럽트를 호출합니다. 여기서 시스템 콜에 해당하는 인터럽트를 발생시키면, 커널 레벨 핸들러에서는 이 eax레지스터에 들어있는 숫자를 파악하여 어떤 시스템콜이 호출되었는지 판별할 수 있습니다.

8. syscall.c

아래의 코드는 커널 레벨의 시스템 콜 핸들러 함수들을 extern으로 설정하여, 다른 파일들에서도 해당 함수를 호출할 수 있도록 만든 부분입니다

```
extern int sys_yield(void);
extern int sys_print_ticks_pid_name(void); // 지난 실습에서 추가
extern int sys_getlev(void);
extern int sys_setpriority(void);
extern int sys_setmonopoly(void);
extern int sys_monopolize(void);
extern int sys_unmonopolize(void);
```

아래는 앞서 usys.S에서 설정했던 어셈블리 함수에서 시스템 콜 인터럽트가 호출되면 그것을 처리하여 해당하는 시스템콜 핸들러 함수를 호출하는 부분으로, eax레지스터에 담겨있는 값에 따라 해당하는 sys_{시스템콜명} 함수를 실행합니다. 이렇게 되면 앞서 sysproc.c에서 만들어 두었던 함수들이 실행되게 됩니다.

```
static int (*syscalls[])(void) = {
...
[SYS_yield] sys_yield,
[SYS_print_ticks_pid_name] sys_print_ticks_pid_name, // 지난 실
[SYS_getlev] sys_getlev,
[SYS_setpriority] sys_setpriority,
[SYS_setmonopoly] sys_setmonopoly,
[SYS_monopolize] sys_monopolize,
[SYS_unmonopolize] sys_unmonopolize,
};
```

9. Makefile

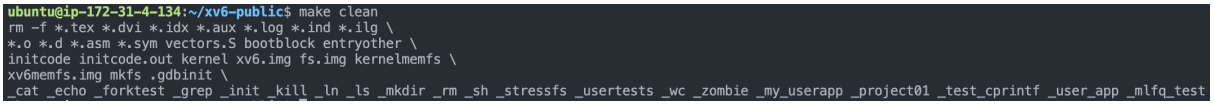
주어진 테스트 파일인 mlfq_test.c를 xv6 코드에서 빌드하기 위해, 아래와 같이 추가하여 줍니다.

```
UPROGS=\
    ...
    _mlfq_test\
```

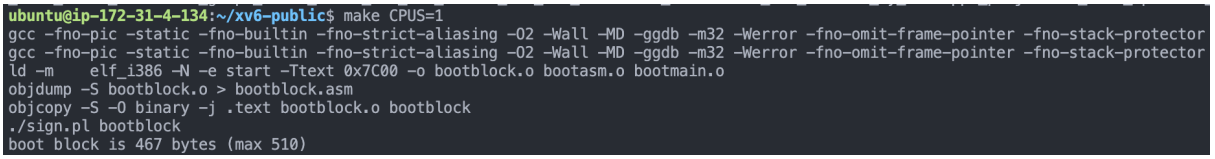
```
EXTRA=\
..mlfq_test.c\
```

Result

먼저 콘솔창에 **make clean**을 입력합니다.



그 다음 **make CPUS=1**를 입력합니다. 작성해둔 bootxv6.sh파일을 보니 -smp=1 옵션이 적혀 있었지만 혹시 몰라 이 과정에서도 CPUS=1 옵션을 입력하였습니다.



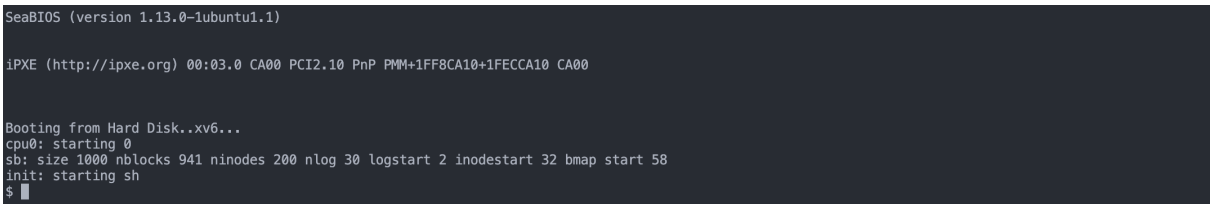
그 다음 **make fs.img CPUS=1**을 입력해줍니다.

그 다음 **./bootxv6.sh**를 입력합니다. bootxv6.sh 파일의 내용은 아래와 같습니다.

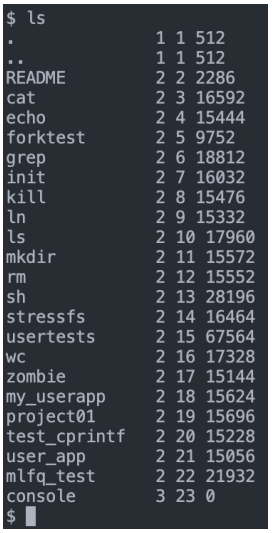
```
#!/bin/bash
```

```
qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6
```

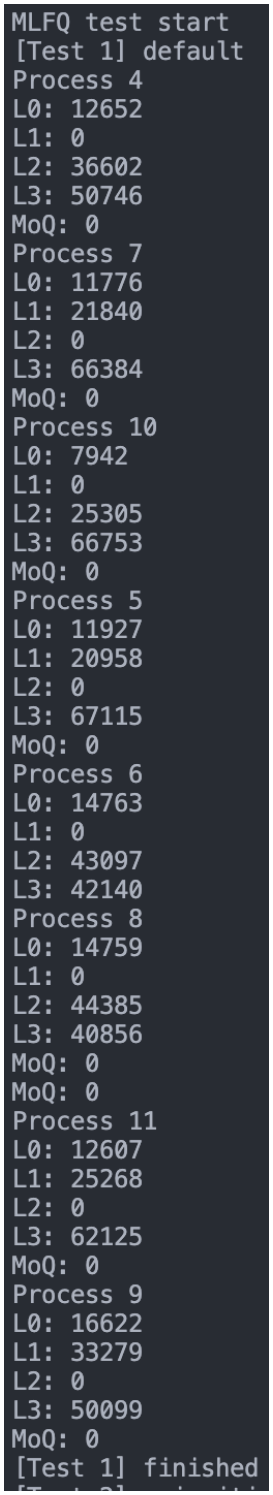
./bootxv6를 입력하면 아래와 같은 화면이 출력됩니다.

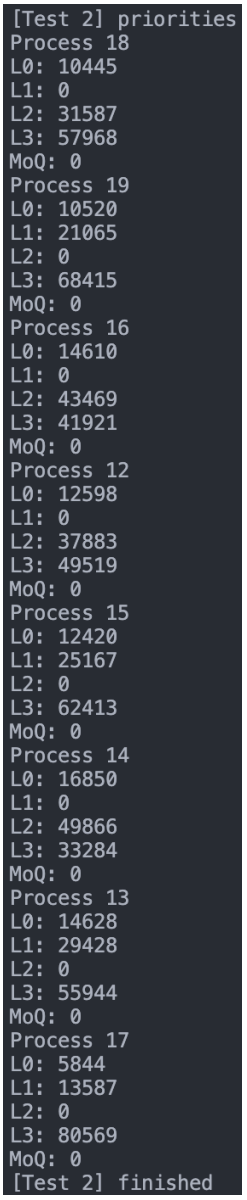


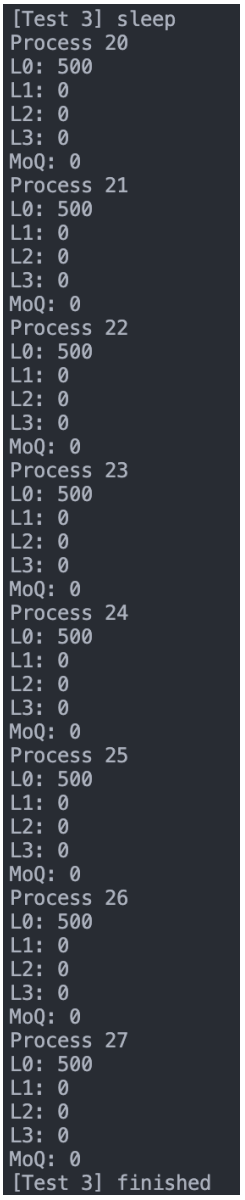
ls를 입력해보면, 테스트를 위해 추가한 mlfq_test가 나타나 있는 것을 확인할 수 있습니다.

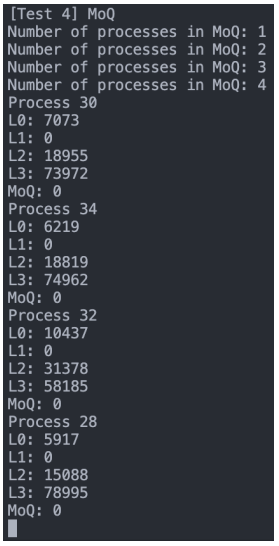


그 다음 테스트를 위해 **mlfq_test**를 입력합니다. 그러면 아래처럼 테스트 프로그램이 실행됩니다.









위에서 L0~L3의 경우 정상적으로 동작하는 것처럼 보였으나, MoQ를 테스트 하는 과정에서 항상 5번째 사이클 부터 진행되지 않고 중단되는 문제가 발생하였습니다.

Trouble shooting

과제 수행 도중 크게 두 가지 문제가 발생하였습니다. 하나는 Deadlock과 관련한 문제였습니다. 처음에는 MLFQ와 MoQ를 구현하기 위해 각각 구조체를 생성하고 각각의 구조체에 struct spinlock lock; 멤버 변수를 만들고 locking 시스템을 구현했었습니다. 그런데 ptable을 순회하면서 동시에 mlfq에 접근해야 하는 경우가 많았습니다. 그래서 아래와 같은 에러가 발생하기도 했습니다.

```
lapicid 0: panic: sched locks
 80103ae5 80103ba0 8010231e 80100191 80101519 80101597 801038
```

처음에는 이 문제가 어떤 문제인지 알지 못해 애를 먹었었으나, 해당 출력이 발생되는 부분을 파일 내에서 찾아본 결과, locking과 관련된 문제라고 인지하였고, 결국 ptable 구조체에 MLFQ와 MoQ에 해당하는 멤버 변수를 모두 포함시킨 후, spinlock을 하나만 사용하므로써 문제를 해결 할 수 있었습니다. 조금 더 좋은 해결 방법이 있을 것 같은데, 약간은 아쉬운 방법이었던 것 같습니다. 조금 더 locking 시스템에 대해서 공부해보고 좋은 해결 방법을 떠올릴 수 있으면 좋을 것 같습니다.

두 번째 문제는 아직까지 해결하지 못한 부분입니다. 위의 결과 부분에서 언급했듯이, MoQ 테스트 중간에서 갑자기 중단되는 문제가 발생했습니다. gdb와 같은 디버거를 사용하면 문제를 찾을 수 있을 것 같은데, 사용 방법을 알지 못하여 cprintf()함수를 코드 중간마다 쳐서 에러가 날 법한 위치를 찾아내는 방식으로 디버깅을 진행했는데, 결국 찾아내지 못했습니다. 이 경우에는 추후에 gdb 사용법을 공부하여 에러가 난 위치를 찾아내야 할 것 같습니다.

비록 요구된 기능들을 완벽하게 동작하게 만들진 못했지만, 과제를 수행하면서 전체적인 XV6 운영체제의 동작 메커니즘을 이해하게 된 것 같아 많이 도움이 되었던 것 같습니다. 긴 글 읽어주셔서 감사합니다.