운영체제 Project1 Wiki

이름: 권도현

학번: 2023065350

학과: 컴퓨터소프트웨어학부

Design

- **Goal**: 이 과제의 목표는 기존 xv6의 Round Robin의 형식의 scheduler를 mode에 따라 FCFS mode라면, First come first served (이하 FCFS) 방식으로 스케줄링하고 MLFQ mode라면 Multi Level Feedback Queue (이하 MLFQ) 방식으로 스케줄링 할 수 있도록 구현하는 것이다.
  - MLFQ에서는 기본적으로 level 0부터 level 2 queue를 확인한다.
  - **Level 0:** 하나의 process가 1 tick 동안 cpu를 사용할 때까지, Round robin 방식을 이용한다. 1 tick 동안 process가 실행되었다면, level 1 queue로 이동한다.
  - **Level 1:** 하나의 process가 3 ticks 동안 cpu를 사용할 때까지, Round robin 방식을 이용하고, 3 ticks 동안 실행되었다면 level 2 queue로 이동한다.
  - **Level 2:** Priority가 높은 process가 계속 실행될 수 있어야 하고, 5 ticks 동안 사용되었다면 priority를 1 줄여야 한다.
- 내가 수정하거나 구현해야 하는 부분은 기존의 scheduler(), timer interrupt 처리 관련 부분을 수정해야 하고, scheduling, mode switch를 도와주는 system call, test code 등에서 사용할 system call을 구현해야 한다.
- FCFS와 MLFQ를 구현하기 위해 기본 circular queue 자료구조를 기준으로 FCFS에서는 circular queue에 몇 가지 기능만 추가하고, MLFQ는 circular queue를 Level에 따라 3개를 가질 수 있도록 구현하였다.
- 각 queue를 관리하기 위해 대표적인 queue 자료구조 관리 함수인 push, pop, empty, full을 미리 구현했고, 초기화를 위해 init 함수까지 구현하였다.
- 아래 **Structure**에서 구조를 살펴보며 더 구체적인 Design 과정을 설명하도록 하겠다.

Structure

이번 과제에서는 level0, level1의 Round Robin 방식과, FCFS, priority가 큰 것을 우선하여 실행하는 스케줄링 방식을 구현해야 한다. 따라서 이번 과제의 핵심은 다음 두 가지이다.

1. Timer Interrupt 처리 과정 (xv6 기본 동작 과정)

A. Timer interrupt가 발생
B. Usertrap() / kerneltrap() 이 user mode에서 발생한 interrupt인지, kernel mode에서 발생한 interrupt인지에 따라 호출된다.
C. (B) 과정에서 호출된 각 함수에서 devintr()를 호출한다.
D. devcintr() 에서 해당 interrupt가 Timer interrupt임이 확인되면, clockintr()를 호출한다.
E. Clockintr()에서 tick 증가 및 wakeup 동작을 한다.

- 이 과정을 보고, FCFS에서는 하나의 tick마다 따로 동작 처리를 안 해줘도 되지만, MLFQ에서는 Level 0, level 1에 따라 하나의 Tick마다 cpu를 yield 해야 하고, 다시 scheduling 하도록 해야 한다. 추가로 level 1, level 2의 경우 하나의 tick마다 각각 이전 level의 queue에 process가 존재하는지, 혹은 우선 순위가 더 높은 process가 존재하는지 확인해야 한다.
- 이를 Clockintr()에서 구현할 것이다.

2. yield()를 통한 scheduler 호출 과정

A. yield()의 경우, sched()를 호출하도록 구현되어 있다.
B. sched()는 직전에 실행 중이던 process의 context를 저장하고 cpu의 context로 전환하고 scheduler()를 호출한다.
C. scheduler()는 기본적으로 Round robin 방식으로 구현되어져 있다.

- 우리가 구현해야 하는 것은 mode에 따라 scheduling 방식을 바꾸어야 한다. MLFQ의 경우에는 같은 mode라고 하더라도 level에 따라 scheduling 방식이 다르다. 따라서 scheduler() 함수를 각 mode, level에 맞추어 각각 다른 방식으로 구현할 것이다.

Implementation

1. Queue / FCFS queue / MLFQ queue

기본적으로 각 queue가 모든 같은 방식을 사용하도록 하기 위해 기본적인 queue를 만들었다. FCFS에서는 하나의 queue를 이용하고, MLFQ에서는 세 개의 queue를 이용할 수 있도록 queue[3]을 사용하고 index를 level로 관리하였다.

A. proc.h: 기존 proc 구조체에 mlfq를 위한 field를 추가하였다.

```
struct queue {
  struct proc* q[64];
  int front;
  int rear;
  int num;
};

struct fcfs {
  struct queue entry;
};

struct mlfq {
  struct queue entry[3];
};
```

```
int level;                // Queue level
int priority;             // Process priority (3 - Highest value)
int time_quantum;         // Number of ticks
int limits;               // 2*Levels + 1
```

B. proc.c

i. 먼저 각 queue를 관리할 수 있는 함수들을 만들었다. 기본적으로 자료구조 시간에 배운 circular queue 방식을 참고하였고 FCFS와 MLFQ에서 queue를 관리하는 것을 비교적 쉽게 표현하기 위해 이 함수들을 구현했다.

```
void queue_init(struct queue* q) {
  acquire(&queuelock);
  q->front = 0;
  q->rear = 0;
  q->num = 0;

  for (int i = 0; i < NPROC; i++) { // queue 초기화
    q->q[i] = 0;
  }
  release(&queuelock);
}

int queue_full(struct queue* q) {
  int result = 0;
  acquire(&queuelock);
  if (q->num == NPROC) result = 1;
  release(&queuelock);
  return result;
}

int queue_empty(struct queue* q) {
  int result = 0;
  acquire(&queuelock);
  if (q->num == 0) result = 1;
  release(&queuelock);
  return result;
}

struct proc* queue_pop(struct queue* q) {
  acquire(&queuelock);
  int temp = q->front;
  q->front = (q->front + 1) % NPROC;
  q->num--;
  release(&queuelock);

  return q->q[temp];
}

void queue_push(struct queue* q, struct proc* p) {
  acquire(&queuelock);
  q->q[q->rear] = p;
  q->num++;
  q->rear = (q->rear + 1) % NPROC;
  release(&queuelock);
}
```

ii. FCFS에서는 단일 circular queue를 사용하는 것과 다를 바가 없어 단순히 queue 관리 함수를 호출하는 경우가 대부분이다.

```
void fcfs_init(void) {
  queue_init(&fcfs.entry);
}

int fcfs_full(void) {
  return queue_full(&fcfs.entry);
}

int fcfs_empty(void) {
  return queue_empty(&fcfs.entry);
}

struct proc* fcfs_pop(void) {
  return queue_pop(&fcfs.entry);
}

void fcfs_push(struct proc* p) {
  if (p->state == RUNNABLE) queue_push(&fcfs.entry, p);
}
```

iii. MLFQ는 각 레벨마다 queue, 총 3개의 queue를 가지고 있어 각 level의 queue을 각각 따로 관리할 수 있도록 level이라는 매개변수를 추가하였다.

```
void mlfq_init(void) {
  for (int i = 0; i < 3; i++) {
    queue_init(&mlfq.entry[i]);
  }
}

int mlfq_full(int level) {
  return queue_full(&mlfq.entry[level]);
}

int mlfq_empty(int level) {
  return queue_empty(&mlfq.entry[level]);
}

struct proc* mlfq_pop(int level) {
  return queue_pop(&mlfq.entry[level]);
}

void mlfq_push(int level, struct proc* p) {
  if (p->state == RUNNABLE) queue_push(&mlfq.entry[level], p);
}
```

2. System call

과제 명세서를 따라 구현해야 하는 5개의 system call을 제외하고, 테스트 용을 위해 fcfs와 mlfq queue에 들어있는 process를 출력해주는 system call을 추가로 구현하였다.

실제 system call 구현 전, system call을 등록하기 위한 작업을 먼저 진행했다. 실제 system call 동작 구현 부분은 proc.c에 구현되어져 있다.

A. syscall.h

```
#define SYS_yield  22
#define SYS_getlev 23
#define SYS_setpriority 24
#define SYS_mlfqmode 25
#define SYS_fcfsmode 26
#define SYS_showfcfs 27
#define SYS_showmlfq 28
```

B. syscall.c

```
extern uint64 sys_yield(void);
extern uint64 sys_getlev(void);
extern uint64 sys_setpriority(void);
extern uint64 sys_mlfqmode(void);
extern uint64 sys_fcfsmode(void);
extern uint64 sys_showfcfs(void);
extern uint64 sys_showmlfq(void);
```

```
[SYS_yield]       sys_yield,
[SYS_getlev]      sys_getlev,
[SYS_setpriority] sys_setpriority,
[SYS_mlfqmode]    sys_mlfqmode,
[SYS_fcfsmode]    sys_fcfsmode,
[SYS_showfcfs]   sys_showfcfs,
[SYS_showmlfq]   sys_showmlfq
```

C. sysproc.c

```
uint64
sys_yield(void)
{
  yield();
  return 0;
}

uint64
sys_getlev(void)
{
  return getlev();
}

uint64
sys_setpriority(void) // sys call argument는 sys call number를 제외하고 없다.
{
  int pid;
  int priority;

  argint(0, &pid); // 직접 가지고 와야한다.
  argint(1, &priority);

  return setpriority(pid, priority);
}

uint64
sys_mlfqmode(void)
{
  return mlfqmode();
}

uint64
sys_fcfsmode(void)
{
  return fcfsmode();
}

uint64
sys_showfcfs(void)
{
  return showfcfs();
}

uint64
sys_showmlfq(void)
{
  return showmlfq();
}
```

D. defs.h

```
void            yield(void);
int             getlev(void);
int             setpriority(int pid, int priority);
int             mlfqmode(void);
int             fcfsmode(void);
int             showfcfs(void);
int             showmlfq(void);
```

E. proc.c

i. Yield: 현재 cpu를 양보해야 하는 process의 state를 RUNNABLE로 바꾸고 level에 맞는 queue에 push 한다. FCFS의 경우, Non-preemptive이기 때문에 yield를 사용할 필요가 없다. 따라서 MLFQ의 경우에만 동작하도록 한다.

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  if (checkmode == 1) { // MLFQ 일때만 yield
    struct proc *p = myproc();

    acquire(&p->lock);
    p->state = RUNNABLE;
    mlfq_push(p->level, p); // Level에 맞는 queue 뒤에 넣는다.
    sched();
    release(&p->lock);
  }
}
```

ii. getlev: 각 process가 관리하고 있는 level을 return 하도록 한다. P에 대한 race condition이 일어나지 않도록 lock을 걸어주었다.

```
int
getlev(void)
{
  if (checkmode == 0) return 99;
  else {
    struct proc *p = myproc();
    // process에 대한 접근을 atomic 하게 만든다.
    acquire(&p->lock);
    int l = p->level;
    release(&p->lock);
    return l;
  }
}
```

iii. setpriority: 과제 명세서에 따라 priority 범위를 벗어나면 -2 return, Loop를 돌며 찾고자 하는 pid와 일치하는 process가 있다면 해당 process의 priority를 세팅한다. 일치하는 process가 없다면 -1을 return한다.

```
int
setpriority(int pid, int priority)
{
  if (priority < 0 || priority > 3) return -2;

  for (struct proc *p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if (p->pid == pid) {
      p->priority = priority;
      release(&p->lock);
      return 0;
    }
    release(&p->lock);
  }

  return -1;
}
```

Mode switch에 앞서 proc.c에 전역 변수로 checkmode라는 정수형 변수를 만들어

checkmode의 값이 0이면 FCFS mode, 1이면 MLFQ mode를 나타내도록 하였다.

iv. mlfqmode: 이미 mlfqmode라면 -1을 return하고, fcfs queue에 있던 process들을 초기화하고, mlfq의 level 0 queue에 추가한다. 현재 실행하고 있는 프로세스는 fcfs queue에 없으므로 mlfq mode에 맞게 초기화만 해주고 지속적으로 실행은 되도록 한다. Modeswitch 이후에는 init 함수로 field와 queue를 초기화했다. Modelock을 이용하여 mode switch가 atomic함을 보장한다.

```
int
mlfqmode (void)
{
  if (checkmode == 1) { // 이미 MLFQ mode인지 확인
    //printf("no changes are made\n"); - testcase에서 출력
    return -1;
  }

  acquire(&modelock);

  checkmode = 1;
  global_tick_count = 0;

  // FCFS 큐에 있던 프로세스를 모두 L0으로 이동
  while (!fcfs_empty()) {
    struct proc *p = fcfs_pop();

    acquire(&p->lock);
    if (p->state == RUNNABLE) {
      p->level = 0;
      p->priority = 3;
      p->time_quantum = 0;
      p->limits = 1;
      mlfq_push(0, p);
    }
    release(&p->lock);
  }

  // 현재 실행 중인 프로세스를 L0으로 초기화
  // 계속 실행은 되도록 한다.
  struct proc *now = myproc();
  acquire(&now->lock);
  if (now->state == RUNNING) {
    now->level = 0;
    now->priority = 3;
    now->time_quantum = 0;
    now->limits = 1;
  }
  release(&now->lock);

  fcfs_init();
  release(&modelock);
  return 0;
}
```

i. fcfsmode: 우선 이미 FCFS mode라면 -1을 return하게 한다. Mlfq mode에서 fcfs mode로 돌아갈 때, 가장 중요한 부분은 pid 즉, 생성 시간에 따라서 fcfs queue에서 정렬되어야 한다는 것이다. 따라서 임시 저장 (runnable_proc) 배열을 만들어서 level 0부터 level 2까지 돌며 임시 저장 배열에 넣고, 임시 저장 배열을 pid가 작은 순으로 정렬하고 순서대로 FCFS queue에 넣도록 구현하였다. 정렬 방식은 시간이 오래 걸리더라도 가장 명확한 Bubble sort 방식을 사용하였다. mlfqmode()와 동일하게 현재 실행하고 있는 프로세스는 mlfq queue에 없으므로 fcfs mode에 맞게 초기화만 해주고 지속적으로 실행은 되도록 한다. Modeswitch 이후에는 init 함수로 field와 queue를 초기화했다. Modelock을

이용하여 mode switch가 atomic함을 보장한다.

```
int
fcfsmode (void)
{
  if (checkmode == 0) { // 이미 FCFS인지 확인
    // printf("no changes are made\n"); - testcase에서 출력
    return -1;
  }

  acquire(&modelock);

  checkmode = 0;
  global_tick_count = 0;

  int index = 0;
  struct proc* temp; // 임시 저장 용
  struct proc* runnable_proc[NPROC];

  for (int i = 0; i < 3; i++) {
    while (!mlfq_empty(i)) {
      temp = mlfq_pop(i);

      acquire(&temp->lock);
      if (temp->state == RUNNABLE) {
        temp->level = -1;
        temp->priority = -1;
        temp->time_quantum = -1;
        temp->limits = -1;
        runnable_proc[index++] = temp;
      }
      release(&temp->lock);
    }
  }

  // 현재 실행 중인 프로세스를를 FCFS으로 초기화
  // 계속 실행은 되도록 한다.
  struct proc *now = myproc();
  acquire(&now->lock);
  if (now->state == RUNNING) {
    now->level = -1;
    now->priority = -1;
    now->time_quantum = -1;
    now->limits = -1;
  }
  release(&now->lock);

  // Bubble sort로 pid 작은 순대로 정렬
  for (int i = 0; i < index - 1; i++) {
    for (int j = i + 1; j < index; j++) {
      if (runnable_proc[i]->pid > runnable_proc[j]->pid) {
        temp = runnable_proc[i];
        runnable_proc[i] = runnable_proc[j];
        runnable_proc[j] = temp;
      }
    }
  }

  for (int i = 0; i < index; i++) {
    fcfs_push(runnable_proc[i]);
  }

  mlfq_init(); // queue를 초기화

  release(&modelock);
  return 0;
```

v. showfcfs / showmlfq: 각각 현재 FCFS / MLFQ queue에 들어있는 process들을 모두 출력하도록 구현하였다. Lock을 걸어 출력 중에 queue 안에 저장되어 있는 process가 변하지 않도록 구현했다.

```c
// Debugglingng용으로 FCFS queue와 MLFQ를 출력
int
showfcfs(void)
{
  acquire(&queuelock);
  printf(">>> FCFS queue [%d procs]: ", fcfs.entry.num);
  for(int i = 0; i < fcfs.entry.num; i++){
    int index = (fcfs.entry.front + i) % NPROC;
    struct proc *p = fcfs.entry.q[index];
    if(p) printf("%d ", p->pid);
  }
  printf("\n");
  release(&queuelock);

  return 0;
}

// 각 MLFQ queue에 들어있는 process를 level 0부터 출력
int
showmlfq(void)
{
  acquire(&queuelock);
  for(int l = 0; l < 3; l++){
    struct queue *q = &mlfq.entry[l];
    printf(">>> MLFQ L%d queue [%d procs]: ", l, q->num);
    for(int i = 0; i < q->num; i++){
      int index = (q->front + i) % NPROC;
      struct proc *p = q->q[index];
      if(p) printf("%d ", p->pid);
    }
    printf("\n");
  }
  release(&queuelock);
  return 0;
}
```

이후부터는 User 폴더 내의 코드이다.

F. user.h

```c
int yield(void);
int getlev(void);
int setpriority(int pid, int priority);
int mlfqmode(void);
int fcfsmode(void);
int showfcfs(void);
int showmlfq(void);
```

G. usys.pl

```
entry("yield");
entry("getlev");
entry("setpriority");
entry("mlfqmode");
entry("fcfsmode");
entry("showfcfs");
entry("showmlfq");
```

3. Initialize

새로운 process가 생기거나 sleep 상태이던 process가 wakeup 되면 mode에 따라 process가 가지고 있는 각 field가 초기화 되어야 한다.

각 mode에 맞는 field만 규칙에 맞게 초기화해주면 현재 mode와 관련 없는 부분의 field는 mode switch 시 초기화되기 때문에 현재 mode에 맞는 field만 초기화했다.

userinit(), fork(), wakeup(), kill() 함수 내에서 해당하는 process의 state를 RUNNABLE로 바꾸는 코드 밑에 전부 아래와 같은 초기화 코드를 추가하고, push를 호출하여 queue에 추가하였다..

비록 중복되는 부분이 있더라도, 코드에서 오류를 발생하지 않고, 큰 오버헤드를 가지지 않는 것 같아 모든 코드에 추가했다.

```
if (checkmode == 0) { // FCFS
  p->priority = -1;
  p->level = -1;
  p->time_quantum = -1;
}
else { // MLFQ
  p->level = 0; // L0
  p->priority = 3; // 3 - Highest value
  p->time_quantum = 0;
  p->limits = 1; // L0의 time quantum은 1
}
```

```
//  state 변경 및 push
np->state = RUNNABLE;
if (checkmode == 0)
  fcfs_push(np);
else
  mlfq_push(0, np);
```

4. Scheduling

A. FCFS: FCFS를 non-preemptive로 실행해야 하는 부분은 다른 곳에서 구현하였고, scheduler() 함수에선 단순하게 queue에서 Runnable 하다면 pop하고 실행할 수 있도록 하였고 Runnable 하지 않다면 FCFS의 가장 뒤에 다시 삽입했다. Lock을 이용하여 내가 p에 대한 상태를 확인하는 동안에는 p의 값이 바뀌지 않는 것을 보장하였다.

```
if (checkmode == 0 && !fcfs_empty()) { // FCFS 경우 + FCFS가 비어있지 않으면
  p = fcfs_pop();

  acquire(&p->lock);
  if(p->state == RUNNABLE) { // 더블 체크
    // Switch to chosen process.  It is the process's job
    // to release its lock and then reacquire it
    // before jumping back to us.
    p->state = RUNNING;
    c->proc = p;
    swtch(&c->context, &p->context);

    // Process is done running for now.
    // It should have changed its p->state before coming back.
    c->proc = 0;
    found = 1;
  }
  else { // RUNNABLE이 아닌 경우
    fcfs_push(p);
  }
  release(&p->lock);
}
```

B. MLFQ: 우선 가장 바깥의 if 문을 사용하여 각 level의 queue가 비였는 지부터 확인하도록 하였다.

Level이 비교적 낮은 queue부터 조건을 체크하여 Level이 낮은 queue에 process가 있다면 다른 조건은 확인도 하지 않는 방식으로 Level이 낮은 것에 대한 우선 순위를 부여하였다.

Level 0와 Level 1은 기본적으로 Round robin이고 time_quantum을 제외하면 첫 번째 process를 가져와야 함으로 pop 하도록 구현했다.

Level2는 우선 mlfq의 level2 queue를 돌며, priority가 가장 높은 index를 저장하고 해당 index에 해당하는 process가 실행 가능한 상태라면, 해당 process를 실행하도록 구현하였다.

```
else if (checkmode == 1) { // MLFQ
  if (!mlfq_empty(0)) {
    p = mlfq_pop(0);
    found = 1;
  }
  else if (!mlfq_empty(1)) {
    p = mlfq_pop(1);
    found = 1;
  }
  else if (!mlfq_empty(2)) {
    int max = -1; // Max priority를 찾기 위함
    int index = -1;

    for (int i = 0; i < mlfq.entry[2].num; i++) {
      temp = mlfq.entry[2].q[(mlfq.entry[2].front + i) % NPROC];
      acquire(&temp->lock);
      if (temp->priority > max) {
        max = temp->priority;
        index = i;
      }
      release(&temp->lock);
    }

    if (index != -1) { // index == -1 이면 없는 것
      acquire(&queuelock);
      p = mlfq.entry[2].q[(mlfq.entry[2].front + index) % NPROC];

      // mlfq_pop이 level0, level1 기준이라 강제로 빼고 한 칸씩 앞으로 이동
      for (int i = index; i < mlfq.entry[2].num; i++) {
        int past = (mlfq.entry[2].front + i + 1) % NPROC; // 이전 위치
        int new = (mlfq.entry[2].front + i) % NPROC; // 새로운 위치
        mlfq.entry[2].q[new] = mlfq.entry[2].q[past];
      }
      mlfq.entry[2].rear = (mlfq.entry[2].rear + NPROC - 1) % NPROC; // 음수가 되는 것을 방지해 NPROC를
      mlfq.entry[2].num--;
      release(&queuelock);

      found = 1;
    }
  }
```

마지막으로 FCFS와 MLFQ 동일하게 스케쥴 대상 process를 찾았다면 found = 1로 설정하게 하고, found = 1이라면 현재 p에 저장되어 있는 즉, 실행의 대상이 되는 process의 상태가 RUNNABLE인지 확인하고 실행할 수 있도록 구현하였다.

```
if (p != 0 && found == 1) { // p를 확실히 가져올 수 있는 상황이라면 p를 가져온다.
  acquire(&p->lock);
  if (p->state == RUNNABLE) {
    p->state = RUNNING;
    c->proc = p;
    swtch(&c->context, &p->context);
    c->proc = 0;
  }
  release(&p->lock);
}
```

5. Locking

Queue와 mode switch에 lock을 걸기 위해 spinlock을 활용하였다.

Spinlock queuelock, modelock을 선언하고 procinit()에서 사용할 수 있도록 초기화하였다.

```
void
procinit(void)
{
  struct proc *p;

  initlock(&pid_lock, "nextpid");
  initlock(&wait_lock, "wait_lock");
  initlock(&queuelock, "queue_lock"); // 처음에 한번만 호출되도록
  initlock(&modelock, "modelock");

  for(p = proc; p < &proc[NPROC]; p++) {
      initlock(&p->lock, "proc");
      p->state = UNUSED;
      p->kstack = KSTACK((int) (p - proc));
  }
}
```

6. Timer interrupt 처리

A. FCFS: FCFS에서는 Timer interrupt가 큰 영향을 미치지 못한다. 따라서 Timer interrupt로 인해 process가 yield 되면 안 된다. 따라서 kerneltrap / usertrap 모두에 FCFS mode인 경우 그냥 yield를 호출하도록 하고 yield에 FCFS 경우에는 별 다른 행동을 취하지 않도록 만들어 Non-preemptive 방식의 FCFS를 구현하였다.

```
if(which_dev == 2 && myproc() != 0) {
  if (checkmode == 0) yield(); // FCFS면 그냥 yield
}
```

B. MLFQ: MLFQ의 경우 Timer interrupt가 올 때마다 Level 0, 1에서 yield 해주어야 하고, "scheduling의 대상이 되는 queue level을 바꾸어야 할 지", priority boosting 그리고 "현재 process가 주어진 time quantum을 모두 사용하였다면 process의 level이나 priority를 조절해주는 것이 필요하다. 이 모든 작업은 clockintr() 함수에서 진행한다.

단 이 함수 내에서 cpu를 빼앗지 않는다. 해당 작업은 yield()에서 진행하며, 알맞은 queue에 넣어주는 작업도 yield()에서 진행한다.

우선 clockintr() 호출 될 때마다 priority boosting을 위한 global_tick_count를 1씩 증가시킨다.

i. 주어진 time quantum을 전부 사용했을 경우
Level 0: Level 1으로 이동하고 limits = 3
Level 1: Level 2로 이동하고 limits = 5
Level 2: priority가 0이 아니라면 priority를 1 감소

```
int checkyield = 0; //yield 가능 여부

if (checkmode == 1) { // MLFQ
  struct proc *p = myproc();
  global_tick_count++;

  if (p && p->state == RUNNING) {
    acquire(&p->lock);
    p->time_quantum++;

    if (p->time_quantum >= p->limits) { // 주어진 time quantum을 전부 사용했을 경우
      p->time_quantum = 0; // time quantum을 초기화

      if (p->level == 0) { // Demoted to L1
        p->level = 1;
        p->limits = 3;
      }
      else if (p->level == 1) { // Demoted to L2
        p->level = 2;
        p->limits = 5;
      }
      else if (p->level == 2) {
        if (p->priority > 0) p->priority --;
      }

      p->state = RUNNABLE;
      release(&p->lock);

      checkyield = 1; // yield 가능
```

ii. Level이나 Priority를 바꿔야 하는지 확인

현재 실행 중인 process의 level이나 priority보다 높은 process가 존재한다면 현재 실행 중인 process는 cpu를 양보해야 한다.

Level 0: Time quantum을 전부 사용한 것이 아니라면, cpu를 뺏길 일이 없다.
Level 1: Level 0 queue가 비어 있지 않다면 yield() 호출해야 한다.
Level 2: Level 0 / 1 queue가 비어 있지 않거나, Level 2 queue에 현재 실행 중인 process보다 priority가 높은 process가 있다면 yield()를 호출해야 한다.

```
else { // level이나 priority를 바꾸어야되는지 먼저 확인
  if (p->level == 1) {
    release(&p->lock);
    if (!mlfq_empty(0)) checkyield = 1; // L0, L1에서 yield 가능
  }
  else if (p->level == 2) { // priority 방식
    // level0 또는 level1에 프로세스 있으면 yield
    if (!mlfq_empty(0) || !mlfq_empty(1)) {
      release(&p->lock);
      checkyield = 1;
    }

    // level2 안에서 우선순위 비교
    for (int i = 0; i < mlfq.entry[2].num; i++) {
      struct proc *temp = mlfq.entry[2].q[(mlfq.entry[2].front + i) % NPROC];
      if (temp->priority > p->priority && temp->state == RUNNABLE) {
        release(&p->lock);
        checkyield = 1;
        break;
      }
    }
    if (checkyield == 0) release(&p->lock);
  }
}
```

위의 두 경우 중 하나라도 해당되면 yield()를 호출하여 스케줄링 될 수 있도록 한다.

```
if (checkyield == 1) yield(); // yield 가능하면 yield
```

iii. Priority boosting

MLFQ에서 global_tick_count가 50이 될 때마다 level0, level1, level2에 있는 모든 process를 level 0 queue에 넣어야 한다.

나는 이 과정에서 Level 0에 있는 process는 어차피 Level 0 queue에 들어갈 필요도 없고 field 상태도 priority boosting 이후와 동일하므로 level 1, level 2 queue 순으로 field 초기화와 level 0 queue에 push 하도록 구현하였다.
단, 현재 실행 중이던 process도 level 0 queue에 들어가도록 초기화되어야 하므로 실행이 끊기진 않게 level 0 queue에 push 하지는 않고, field만 초기화 해주었다.
마지막으로 global_tick_count = 0으로 리셋해주었다.

```
// Starvation 방지 - Priority boosting
if (global_tick_count >= 50 && checkmode == 1) {
  printf("[BOOST] triggered at tick = %d\n", global_tick_count);
  for (int level = 1; level <= 2; level++) {
    while (!mlfq_empty(level)) {
      struct proc *temp = mlfq_pop(level);

      acquire(&temp->lock);
      if (temp->state != UNUSED) {
        temp->level = 0;
        temp->priority = 3;
        temp->time_quantum = 0;
        temp->limits = 1;
        mlfq_push(0, temp); // L0 큐에 다시 push
      }
      release(&temp->lock);
    }
  }

  // 현재 실행 중인 프로세스도 Level0으로 이동
  struct proc *me = myproc();
  if (me != 0) {
    acquire(&me->lock);
    if (me->state == RUNNING) {
      me->level = 0;
      me->priority = 3;
      me->time_quantum = 0;
      me->limits = 1;
    }
    release(&me->lock);
  }

  global_tick_count = 0;
```

Results

1. Testcase 1 (조교님이 제공해주신 코드)
   조교님께서 제공해주신 test code는 FCFS scheduling, FCFS에서 MLFQ로의 mode switch 그리고 MLFQ scheduling 이 제대로 동작하는지 확인하는 코드이다. 결과는 아래와 같다. FCFS에서 pid가 작은 순으로 실행되며, "successfully changed to MLFQ mode" message를 통해 mode switch가 성공적임을 확인할 수 있고 MLFQ에서도 각 process에서 L0, L1, L2에 대한 hit count가 골고루 분포되어 있는 것으로 보아, 정상 동작한다고 할 수 있다.

```
$ test1
FCFS & MLFQ test start

[Test 1] FCFS Queue Execution Order
Process 4 executed 100000 times
Process 5 executed 100000 times
Process 6 executed 100000 times
Process 7 executed 100000 times
[Test 1] FCFS Test Finished

nothing has been changed
successfully changed to MLFQ mode!

[Test 2] MLFQ Scheduling
Checking level's hit count...
Process 8 (MLFQ L0-L2 hit count):
L0: 8205
L1: 29367
L2: 62428
Checking level's hit count...
Process 9 (MLFQ L0-L2 hit count):
L0: 11553
L1: 35734
L2: 52713
Checking level's hit count...
Process 10 (MLFQ L0-L2 hit count):
L0: 12506
L1: 36832
L2: 50662
Checking level's hit count...
Process 11 (MLFQ L0-L2 hit count):
L0: 11435
L1: 34416
L2: 54149
[Test 2] MLFQ Test Finished

FCFS & MLFQ test completed!
```

2. Testcase 2 (내가 만들어본 test case)
   테스트 코드는 다음과 같다.

   Test 1: FCFS queue에 생성 순서대로 들어갔는지 확인한다.
   Test 2: FCFS -> MLFQ switch 후 MLFQ queue의 상태 확인
   Test 3: MLFQ 동작 과정 확인 및 Priority boosting이 제대로 일어나는지 확인
   Test 4: MLFQ -> FCFS switch 후 FCFS queue의 상태 확인
   Test 5: Mode switch 후, FCFS queue 정상 동작 확인

   Test 3에서 실제로 tick마다 확인하기가 어려워서 sleep()을 사용하였는데, sleep() system call을 사용하여 의도하지 않게 global_tick_count가 더 증가하는 경우가 발생하였다. 이를 감안하여 sleep 36번째가 첫 번째 priority boosting 발생 시점이라고 하자. (디버깅을 통해 확인했다.)

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

#define NPROCS       3      // 단계별로 fork할 자식 수
#define BOOST_TICKS  60     // priority boost를 위한 tick 수
#define STEP_TICKS   3

int
main(void)
{
  int p_fcfs[NPROCS], p_mlfq[NPROCS], p_final[NPROCS];
  int i, elapsed;

  printf("=== FCFS <-> MLFQ Full Queue Test ===\n\n");
  // [1] FCFS 모드: 초기 FCFS 큐 확인 (자식 유지)
  printf("[1] FCFS initial queue (keep children)\n");
  fcfsmode();
  for (i = 0; i < NPROCS; i++) {
    if ((p_fcfs[i] = fork()) == 0) {
      // FCFS 모드에선 선점이 없으므로,
      // 큐에 계속 머무르려면 잠시라도 실행을 yield 해 줘야 합니다.
      sleep(1);
      // wakeup → fcfs_push() 후 exit()
      exit(0);
    }
  }
  sleep(1);
  printf(" FCFS queue contents:\n");
  showfcfs();
  printf("[1] 테스트 완료\n\n");

  // [2] FCFS -> MLFQ 모드 전환 직후 큐 확인
  printf("[2] Switch FCFS -> MLFQ (children still alive)\n");
  mlfqmode();
  printf(" MLFQ queues immediately after switch:\n");
  showmlfq();
  // Stage1 자식 정리
  for (i = 0; i < NPROCS; i++) kill(p_fcfs[i]);
  for (i = 0; i < NPROCS; i++) wait(0);
  printf("[2] 테스트 완료\n\n");

  // [3] MLFQ 모드: 데모션 & Priority-Boost 확인
  printf("[3] MLFQ demotion & priority-boost (5 ticks step)\n");
  for(i = 0; i < NPROCS; i++){
    if ((p_mlfq[i] = fork()) == 0) {
      while (1) ;
    }
  }

  elapsed = 0;
  while(elapsed < BOOST_TICKS){
    sleep(STEP_TICKS);
    elapsed += STEP_TICKS;
    printf(" after %d sleeps:\n", elapsed);
    showmlfq();
  }
  printf("[3] 테스트 완료\n\n");
```

```
// [4] MLFQ -> FCFS 모드 전환 직후 큐 확인
printf("[4] Switch MLFQ -> FCFS (children still alive)\n");
fcfsmode();
printf(" FCFS queue after switch:\n");
showfcfs();
// Stage3 자식 정리
for (i = 0; i < NPROCS; i++) kill(p_mlfq[i]);
for (i = 0; i < NPROCS; i++) wait(0);
printf("[4] 테스트 완료\n\n");

// [5] FCFS 모드 재검증: 새 자식 fork 후 큐 확인
printf("[5] FCFS final queue (new children)\n");
for (i = 0; i < NPROCS; i++) {
  if ((p_final[i] = fork()) == 0) {
    sleep(1);
    exit(0);
  }
}
sleep(1);
showfcfs();
for (i = 0; i < NPROCS; i++) wait(0);
printf("[5] 테스트 완료\n\n");

printf("=== 모든 테스트 완료 ===\n");
exit(0);
```

- Test 1: FCFS queue에는 생성 순으로 들어가 있다.

```
[1] FCFS initial queue (keep children)
 FCFS queue contents:
>>> FCFS queue [3 procs]: 4 5 6
[1] 테스트 완료
```

- Test 2: Mode switch 이후에 MLFQ의 level 0 queue에 그대로 들어가 있는 것을 확인할 수 있다.

```
[2] Switch FCFS -> MLFQ (children still alive)
 MLFQ queues immediately after switch:
>>> MLFQ L0 queue [3 procs]: 4 5 6
>>> MLFQ L1 queue [0 procs]:
>>> MLFQ L2 queue [0 procs]:
[2] 테스트 완료
```

- Test 3

  3 tick을 실행하고 출력을 하여 처음 level 0 -> level 1은 확인할 수 없다.

  Level 1 -> level 2는 다음 과정에서 확인할 수 있다.

```
>>> MLFQ L0 queue [0 procs]:
>>> MLFQ L1 queue [3 procs]: 7 8 9
>>> MLFQ L2 queue [0 procs]:
 after 15 sleeps:
>>> MLFQ L0 queue [0 procs]:
>>> MLFQ L1 queue [1 procs]: 9
>>> MLFQ L2 queue [2 procs]: 7 8
```

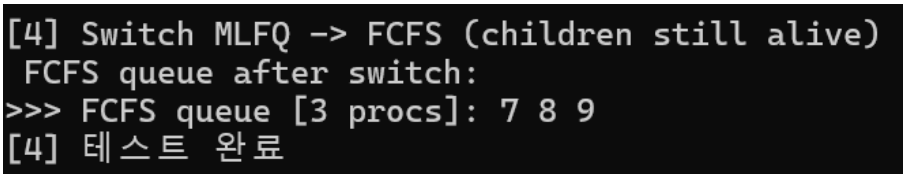
Level 2에서 priority에 따라 순서가 바뀌는 것도 확인할 수 있다.

```
>>> MLFQ L0 queue [0 procs]:
>>> MLFQ L1 queue [0 procs]:
>>> MLFQ L2 queue [3 procs]: 8 9 7
 after 21 sleeps:
>>> MLFQ L0 queue [0 procs]:
>>> MLFQ L1 queue [0 procs]:
>>> MLFQ L2 queue [3 procs]: 7 8 9
 after 24 sleeps:
>>> MLFQ L0 queue [0 procs]:
>>> MLFQ L1 queue [0 procs]:
>>> MLFQ L2 queue [3 procs]: 9 7 8
 after 27 sleeps:
>>> MLFQ L0 queue [0 procs]:
>>> MLFQ L1 queue [0 procs]:
>>> MLFQ L2 queue [3 procs]: 8 9 7
```

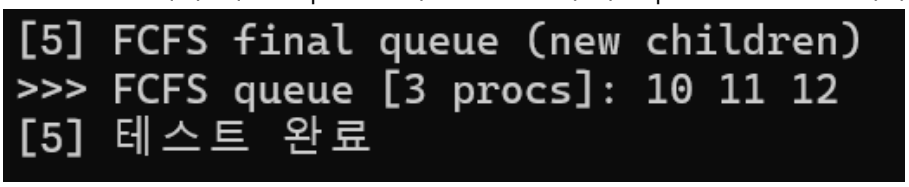
Priority boosting은 36 sleeps 일 때 확인 가능하다.

```
 after 33 sleeps:
>>> MLFQ L0 queue [0 procs]:
>>> MLFQ L1 queue [0 procs]:
>>> MLFQ L2 queue [3 procs]: 9 7 8
 after 36 sleeps:
>>> MLFQ L0 queue [3 procs]: 8 9 7
>>> MLFQ L1 queue [0 procs]:
>>> MLFQ L2 queue [0 procs]:
 after 39 sleeps:
>>> MLFQ L0 queue [0 procs]:
>>> MLFQ L1 queue [3 procs]: 8 9 7
>>> MLFQ L2 queue [0 procs]:
```

- Test 4: MLFQ -> FCFS에서 FCFS mode로 전환되니 pid 순으로 정렬되어 FCFS queue에 있는 것을 확인할 수 있다.



- Test 5: FCFS에서 새로운 process가 생성된 경우에도 pid 순으로 정렬되어져 있다.



Troubleshooting

1. 처음엔 MLFQ에 대해 MLFQ를 구현할 수 있는 구조체를 따로 만들지 않고, proc[NPROC]를 순회하며 각 process의 Level에 맞는 것만 실행할 수 있도록 하여 MLFQ를 구현하려고 하였다.
   - 기존 scheduler, sched code가 proc[NPROC]을 매번 순회하는 방식이라 이 방법대로 구현하기 위해서 생각해낸 방법이었다.
   - 하지만 이 방법대로 구현하면 한 process에서 다음으로 같은 level을 가지는 process의 위치를 확인할 방법이 없어 Level 0 Queue와 Level 1 Queue가 ㅆRound Robin 방식인 것을 구현하기 힘들었다.

2. Level1에서 1 tick마다 queue의 뒤로 보내주어야 하는데 Level1에서 각 process에 time quantum = 3이라는 명세서의 글을 보고 Level 0에 다른 프로세스가 없는 경우에 3 ticks 동안 실행되는 것이 보장되어야 한다고 착각했다. 이는 나중에 구현 과정에서 clockintr() 함수 내부 구현을 바꾸어서 해결하였다.

3. 현재 실행 중인 process는 어떻게 처리가 되어야 하는지에 대한 고민을 처음에는 생각해내지 못하였다. 하지만 테스트 코드를 실행해보는 과정에서 priority boosting이나 mode switch 이후 현재 실행 중인 process에 대한 정보가 출력되지 않는다는 것을 확인하고 현재 실행 중인 process에 대해서는 cpu를 뺏지는 않되, field의 정보는 수정하는 방식으로 구현하여 해결하였다.

4. 테스트 케이스를 실행하던 도중 출력이 잘리는 현상이 발생하였다. 처음엔 내가 Lock을 제대로 걸지 못하여 충돌이 발생하는 줄 알았지만, process간의 충돌은 아니도 출력(printf) 상에서 중복된 호출로 관련 문제가 발생하는 것이었다. 아래처럼 일부분을 수정하여 조금은 개선하였다. Console.c에서 한 글자씩 출력하는 부분에 spinlock을 이용하였다. 그러나 이 방법은 한 글자 출력에 대해서만 보호할 수 있어 여러 글자를 출력하는

경우에 항상 성립하지 않는다는 단점은 존재하였다. 이를 수정하기 위해선 test code 자체 수정을 해야할 것 같아 console.c의 lock을 거는 것까지만 수행해보았다.

```
int
consolewrite(int user_src, uint64 src, int n) // spinlock -> console 출력이 끊기지 않도록 한다.
{
  int i;

  acquire(&prlock);

  for(i = 0; i < n; i++){
    char c;
    if(either_copyin(&c, user_src, src+i, 1) == -1)
      break;
    uartputc(c);
  }

  release(&prlock);
  return i;
}
```

5. 함수들 간의 호출 관계에 있어 어떤 함수에서 어떤 동작까지만 해야 할 지를 정하는 것이 헷갈렸다. 처음에는 clockintr()와 kerneltrap() / usertrap()에서 yield()를 각각 호출하게 되어 하나의 경우에 yield가 여러 번 호출되거나, global_tick_count가 중복되어 count 되는 경우가 있었다. 각 함수에서 어떤 동작까지만 수행할 지 명확히 정하고 구현하니 해결되었다.