Operating System <Project 01>

2023017856 한진호

(1) Design, Implementation, Troubleshooting

1. sysproc.c

//getlev

```
uint64
sys_getlev(void)
{
  if (schedState){
      return 99;
  }
  else {
    struct proc *p = myproc();
    if (p){
      return myproc()→queueLV;
    }
    else {
      return -1;
    }
  }
}
```

proc.c 에서 현재 scheduler가 어떤 모드인지 간편하게 파악하고, 바꿀 수 있게 하기 위해 int schedState = 1; 를 global variable로 선언하였었다. (schedState == 0 → MLFQ / schedState == 1 → FCFS)

하지만 해당 변수에 접근하려 해봤을 때 컴파일러가 해당 변수가 무엇인지 모르는 문제가 발생하였다.

proc.c 내에서의 global variable 이였기에 다른 파일에서는 읽지 못해 생기는 문제였다.

이 문제를 해결하기 위해 defs.h 내에 extern int schedState; 를 하여 해당 문제를 해결하였다.

if else 문을 통해 1일 때는 FCFS이므로 99를 return하고 0인 경우는 MLFQ 이므로 현재 프로세스가 어떤 level 의 queue에 속해있는지를 return 하도록 design 하였다.

//setpriority

```
uint64
sys_setpriority(void)
{
  int pid, priority;
  argint(0, &pid);
  argint(1, &priority);
  if (priority < 0 || priority > 3){
    return -2;
  }
  struct proc *this;
  this = findProc(pid);
  if (this){
    this→priority = priority;
    return 0;
  }
  else {
    return -1;
  }
}
```

proc.c 내 proc [NPROC] (process를 담고 있는 배열)를 for 문을 통해 순회하며 pid를 하나하나 비교하며 일치하는 경우의 process를 따로 빼서 구현하고자 하였다.

하지만 system call을 선언하기 위해 만든 mysyscall.c 에서는 NPROC를 읽을 수 없을 뿐만 아니라 proc [NPROC]가 proc.c 내 정의된 변수이므로 외부 파일인 mysyscall.c 에서 읽을 수 없음을 깨달았다.

이를 해결하기 위해 proc.c 내에 위에서 말한 logic으로 동작하는 함수 findProc를 만들고, defs.h 에 선언하여 mysyscall.c 에서 findProc를 불러와 구현하였다.

//yield

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

과제 명세서에서 void yield(); 가 kernel function 이므로 system call로 만들어주라고 하여 만들었다.

//fcfsmode

```
uint64
sys_fcfsmode(void)
{
    if (schedState){
        printf("Error: CPU scheduler is already FCFS mode.\n");
        return -1;
    }

    for (int i=0; i<NPROC; i++){
      proc[i].queueLV = -1;
      proc[i].tickProc = -1;
      proc[i].priority = -1;
    }
    schedState = 1;
    globaltick = 0;
    return 0;
}
```

if (schedState) 를 통해 해당 조건문이 true인 경우 (schedState == 1) FCFS 이므로 error message 출력 후 -1 을 return 하게 design 하였다.

과제 명세서에서 FCFS 일 때 process’s priority, level, time quantum을 모두 -1로 초기화하라 하여 for 문을 통해 모든 process에 대해 초기화를 진행하였고, schedState = 1; globaltick = 0; 로 초기화한 후 return 0; 을 하였다.

//mlfqmode

```
uint64
sys_mlfqmode(void)
{
  if (!schedState){
      printf("Error: CPU scheduler is already MLFQ mode.\n");
      return -1;
  }
  // for 문으로 모든 proc 접근 후 싹 다 L0 queue에 enqueue해서 넣고자 함.
  // mysyscall.c 에서는 proc[NPROC]에 접근 불가하므로 proc.c 에서 함수를 만들어 (
  schedState = 0;
  globaltick = 0;
  cleanQueue();
  makeQueue();
  return 0;
}
```

if (!schedState) 로 현재 schedState = 0 이면 해당 조건문을 통해 error message 출력 후 return -1; 하도록 하였다.

schedState = 0; globaltick = 0; 으로 초기화 하였고 proc.c 내에 만들어 놓은 cleanQueue(), makeQueue()로 L0 queue 내에 현재 RUNNABLE 한 process를 넣어 놓았다.

2. proc.h

```
struct queue {
  struct proc *data[NPROC];
  int front;
  int rear;
  int capacity;
};

extern struct queue L[3];
```

MLFQ 구현을 위해 queue 라는 struct와 L0, L1, L2 queue 를 표현할 queue 배열을 proc.h 내에 정의하였다.

```
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  int queueLV;                 // MLFQ 에서 현재 Process가 포함 되어 있는 queue의 Lev
  int tickProc;                // MLFQ 에서 Process가 현재 사용한 tick 수
  int priority;                // MLFQ 의 L2 queue 내 priority

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

struct proc 내 member variable로 현재 프로세스가 위치하는 queue의 level (queueLV), 지금까지 프로세스가 사용한 tick의 수 (tickProc), 현재 프로세스의 priority (priority)를 정의하였다.

3. proc.c

//FCFS scheduler

```
struct proc*
FCFS_scheduler(void)
{
   int arr[NPROC];
   for (int i=0; i<NPROC; i++)
      arr[i] = i;

   for (int i=0; i<NPROC-1; i++){
      if (proc[arr[j]].pid < proc[arr[this]].pid){
         this = j;
      }

      int temp = arr[i];
      arr[i] = arr[this];
      arr[this] = temp;
   }

   //...
};
```

FCFS를 구현하고자 pid가 작은 프로세스부터 순차적으로 실행하고자 하였다.

process table을 직접적으로 정렬해버린다면 lock, reference 등에 대해 문제점이 발생하기에

process의 index만을 담고 있는 int 배열 (arr)을 만들어 이를 처리하고자 하였다.

해당 배열을 만들어 우선 Initialize 해주고 selection sort를 진행하여 의도대로 pid가 오름차순이 되도록 배열을 정렬하였다.

하지만 여기서 for 문을 process table의 처음부터 끝까지 돌고자 설정하였는데, 특정 i가 가리키는 process의 index 위치에 process가 존재하지 않는다면 segment fault가 발생할 수 있다.

아래에서 for 문을 통해 proc[arr[i]] 와 같은 꼴로 순차적으로 프로세스를 실행시키고자 하였으나, 이러면 ready 상태의 process가 아닌 process도 무작정 실행시키게 될 수 있다는 문제점을 발견하여 해당 logic을 철회하였다.

```
void
FCFS_scheduler(struct proc *p, struct cpu *c)
```

```
{
  struct proc *this = 0;

  // 가장 작은 pid 가진 RUNNABLE 프로세스 하나 선택
  for (p = proc; p < &proc[NPROC]; p++)
  {
    acquire(&p->lock);
    if (p && p->state == RUNNABLE)
    {
      if (this == 0 || p->pid < this->pid)
      {
        if (this)
          release(&this->lock);  // 이전 후보 lock 해제
        this = p;
        continue;
      }
    }
    release(&p->lock);
  }

  // 2. 선택된 프로세스 실행
  if (this)
  {
    this->state = RUNNING;
    c->proc = this;
    swtch(&c->context, &this->context);
    c->proc = 0;
    release(&this->lock);
  }
  else
  {
    intr_on();
    asm volatile("wfi");
  }
};
```

최종적으로 해당 코드를 짠 후 사용하게 되었다.

위 코드와 flow 자체는 별 차이가 없으나 굳이 int 배열을 선언한 후 selection sort를 진행하며 배열에 프로세스를 pid 순으로 넣지 않았으며, 앞서 문제가 발생할 수 있다고 판단했던 null일 때 상황과 RUNNABLE이 아닌 프로세스인 상황을 if (p && p→state == RUNNABLE) 로 방어하였다.

proc[NPROC] 로 정의되어 있던 기존 process 배열은 pid 순으로 정렬되어 있다는 보장이 없으므로, 직접 for 문을 순회하며 현재까지 가장 작은 pid를 가지고 있던 process를 this에 담고 다음 차례에서 p(proc[NPROC]에서 꺼낸 어떤 한 프로세스)와 this의 pid를 비교하며 업데이트하였다.

만약 this == 0 (첫 for 문 실행 중) 일 때 this = p 로 첫 프로세스를 담아놓고 그 이후로는 p→pid < this→pid 일 때만 if 문이 성립하며 해당 상황은 this를 p로 업데이트 해야하는 상황이다.

for문 순회 때마다 맨 처음에 acquire(&p→lock); 로 걸어놓은 lock을 특별한 상황(p→pid < this→pid이 true인 경우)가 아니라면 끝날 때 release(&p→lock); 으로 lock을 풀어주어 이중 lock이 되어 panic: acquire이 되는 상황을 방어하였고, p→pid < this→pid이 true인 경우에는 if 문 안에서 release(&this→lock); 으로 lock을 풀었다.

이렇게 하면 최종적으로 가장 작은 pid를 지닌 process가 this에 담기게 될 것이다.

RUNNABLE state의 process가 없다면 this가 0일텐데 이런 경우에서 context switch를 진행하게 되면 문제가 발생할 것이므로 if (this)로 방어했고, 그 안에서 context switch를 진행하였다.

//fork

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy user memory from parent to child.
  if(uvmcopy(p→pagetable, np→pagetable, p→sz) < 0){
```

```
  freeproc(np);
  release(&np->lock);
  return -1;
}
np->sz = p->sz;

// copy saved user registers.
*(np->trapframe) = *(p->trapframe);

// Cause fork to return 0 in the child.
np->trapframe->a0 = 0;

// increment reference counts on open file descriptors.
for(i = 0; i < NOFILE; i++)
  if(p->ofile[i])
    np->ofile[i] = filedup(p->ofile[i]);
np->cwd = idup(p->cwd);

safestrcpy(np->name, p->name, sizeof(p->name));

pid = np->pid;

release(&np->lock);

acquire(&wait_lock);
np->parent = p;
release(&wait_lock);

acquire(&np->lock);
np->state = RUNNABLE;

//fork에서 enqueue 진행해버림
if (schedState == 0 && np) {
  np->queueLV = 0;
  np->tickProc = 0;
  np->priority = 3;
  enqueue(np,&L[0]);
}
```

```
  release(&np->lock);

  return pid;
}
```

프로세스가 새로 생겼을 때 proc[NPROC] 에서 탐색해 찾아 L0 queue로 넣는 방법도 가능할 수 있으나, 편리하게 fork() 에서 프로세스가 만들어졌을 때 schedState == 0(MLFQ)이며 복제된 프로세스 np가 0(null)이 아닌 경우 바로 L0 queue에 넣어주는 방법을 채택하여 구현하였다.

//findProc

```
struct proc*
findProc(int pid)
{
  struct proc *this;
  for (int i=0; i<NPROC; i++){
    this = &proc[i];
    if (this && this->pid == pid){
      return this;
    }
  }
  return 0;
}
```

parameter로 내가 찾고자 하는 프로세스의 pid 값을 받아오고, for 문으로 proc[NPROC] 전체를 search 하며 proc[NPROC] 내 pid 와 비교 후 값이 같다면 해당 프로세스를 return 하도록 design 하였다.

(queue를 search 한다면 L0, L1, L2 모두 찾아봐야 하므로 현재 존재하는 모든 프로세스를 관리하는 proc[NPROC] 를 찾아봄으로써 처리하였다.)

//dequeue

```
struct proc*
dequeue(struct queue *q)
{
```

```
  //If queue is empty..
  if (q->front == q->rear && !q->capacity)
  {
    return 0;
  }

  struct proc *p;
  for (int i=q->front; i<q->front+NPROC; i++){
    p = q->data[i%NPROC];
    if (p && p->state == RUNNABLE){
      q->front = (i+1)%NPROC;
      return p;
    }
  }
  return 0;
}
```

만약 front 와 rear가 가리키는 위치가 동일하고 capacity가 없다면 자명하게 queue가 empty 상태임을 알 수 있다. 해당 경우와 의도대로 queue 내에 프로세스를 꺼내주지 못한 경우에 return 0 (null) 하였다.

queue는 front 위치부터 값을 꺼내주어야 하므로 loop의 시작을 front로 하였으며, queue의 front, 상태가 나도 모르게 잘못 관리 되어있을 수 있으므로 최대로 배열 한 바퀴를 다 돌도록 하였다. (for (int i=q->front; i<q->front+NPROC; i++))

loop를 돌며 p에 현재 i가 가리키고 있는 위치의 queue 내 프로세스를 담았다.

만약 p가 null이 아니고 RUNNABLE 하다면 현재 실행 가능한 프로세스를 p가 나타내고 있다는 뜻이므로 front를 한 칸 뒤로 밀어주고 p를 return 해줌으로써 dequeue를 design 하였다.

(여기서 dequeue인데 queue에서 데이터만 밖으로 넘겨주고 delete 해주지 않은 것에 대해 의문을 가질 수 있다. 이는 MLFQ에서 프로세스가 끝날 때까지 실행시키는 것이 아니라 1 tick 만큼의 시간만 할당해주므로 프로세스가 끝나지 않은 상황이 default일 것이다. 여기서 delete를 해버린다면 다음에 이 프로세스를 또 실행시켜줘야 하는데 queue에서 찾지 못하므로 그 프로세스는 영원히 scheduler의 선택을 받지 못하고 RUNNABLE로 남아 문제가 발생할 수 있기 때문에 delete에 대해서는 따로 checkDelete 라는 함수를 만들어 여기서 처리하도록 하였다.)

//enqueue

```
void
enqueue(struct proc *p, struct queue *q)
{
  for (int i=q->rear; i<q->rear+NPROC; i++){
    if (!q->data[i%NPROC]){
      q->data[i%NPROC] = p;
      q->rear = (i+1)%NPROC;
      q->capacity++;
      break;
    }
  }
  return;
}
```

queue 내에 다음 프로세스가 들어갈 위치를 가리킬 rear 부터 시작하여 0(프로세스가 들어있지 않고 텅 비어있는 경우)인 위치가 발견되면 그 위치에 바로 process를 집어넣었다.

프로세스가 queue 내에 들어갔으니 rear를 한 칸 옆으로 밀어주었고, capacity를 1 증가시켜주어 queue를 관리하였다.

해당 작업 (queue 내 프로세스 넣음 + rear, capacity 관리)가 실행되었다면 break; 를 통해 loop를 바로 끝내주어 엉뚱한 위치에 프로세스가 들어가는 일과 rear, capacity가 현재 상태와 다르게 망가지는 문제를 방지하였다.

//saveQueue

```
void
saveQueue()
{
  int idx = 0;
  struct proc *p;
  for (int i=0; i<3; i++){
    for (int j=0; j<L[i].capacity-1; j++){
      p = dequeue(&L[i]);
      p->queueLV = p->tickProc = 0;
      p->priority = 3;

      temp[idx] = p;
```

```
      idx++;
    }
  }
}
```

우선 Priority boosting 할 때 기존 queue 내 순서를 보존하기 위해 struct proc *temp[NPROC]; 를 선언하여 *proc 를 담고자 하였다.

temp에 L0, L1, L2 순으로 프로세스를 담도록 for (int i=0; i<3; i++) loop 를 설정했고, 현재 넣고자 하는 L (queue)의 capacity 만큼 temp에 담도록 for (int j=0; j<L[i].capacity-1; j++)를 설정하였다. (j<L[i].capacity가 아닌 j<L[i].capacity-1 로 설정한 이유는 현재 RUNNING 중인 프로세스도 capacity로 카운트 되었기에 RUNNABLE state만 미리 담아놓기 위해서이다.)

프로세스가 현재 어떤 큐에 있는지, time quantum, priority 를 초기 값으로 초기화한 후 temp에 차례대로 담아두었다.

//cleanQueue

```
void
cleanQueue(void)
{
  for (int i=0; i<3; i++)
  {
    for (int j=0; j<NPROC; j++){
      L[i].data[j] = 0;
    }
    L[i].capacity = 0;
    L[i].front = 0;
    L[i].rear = 0;
  }
  return;
}
```

priority boosting 할 때 L0, L1, L2 queue 전체를 처음 상태 (다 비워져 있는 상태)로 initialize 하기 위해 design 하였다.

//makeQueue

```
void
makeQueue()
{
  for (int i=0; i<NPROC; i++){
    if (temp[i])
      enqueue(temp[i],L);
    else
      break;
  }
}
```

앞서 saveQueue에서 순서대로 temp에 프로세스를 담아놓았으므로 이를 enqueue 를 통해 L0 queue로 옮겼다.

//Priority boosting

```
void
priorityBoosting(void)
{
  if (schedState)
    return;

  saveQueue();
  cleanQueue();
  makeQueue();

  for (int i=0; i<NPROC; i++){
    temp[i] = 0;
  }
  globaltick = 0;
  return;
}
```

saveQueue 를 통해 L0 에 옮길 프로세스들을 미리 temp 내에 담아놓은 후, cleanQueue 로 L0, L1, L2 를 아무것도 없는 상태로 초기화 해주었다. 그 다음 makeQueue로 temp 내 프로세스를 L0로 옮겨 담아 priority boosting 을 구현하였다.

그 후 temp 내에 값이 존재한다면 다음 Priority boosting 때 의도와 다르게 동작할 수 있기에 temp와 global tick을 0으로 초기화하였다.

(위 코드의 flow를 보면 현재 priorityBoosting function을 수행하는 프로세스 (RUNNING state)는 따로 queue 에 넣지 않았는데 이는 trap.c 에서 처리해주었다.)

//deleteElt

```
void
deleteElt(struct queue *q)
{
  struct proc *p = myproc();
  for (int i=q->front-1; i<q->front-1+NPROC; i++){
    if (p && p == q->data[i%NPROC]){
      q->data[i] = 0;
      if (q->capacity>0) {
        q->capacity--;
      }
      return;
    }
  }
}
```

for문을 통해 queue 전체를 순회하며 queue 내에서 현재 deleteElt 를 실행하고 있는 프로세스와 같은 프로세스를 마주친다면 그 위치를 delete 하도록 function을 design 하였다. (이 함수는 demote에서 사용하기 위해 세팅하였다.)

//demote

```
void
demote(void)
{
  struct proc *p = myproc();
  if (p->queueLV == 0)
  {
    p->queueLV = 1;  // L0에서 L1으로 강등
    p->tickProc = 0;
    enqueue(p, &L[1]);  // L1 큐에 추가
```

```
    deleteElt(&L[0]);
  }
  else if (p->queueLV == 1)
  {
    p->queueLV = 2;  // L1에서 L2로 강등
    p->tickProc = 0;
    enqueue(p, &L[2]);  // L2 큐에 추가
    deleteElt(&L[1]);
  }
  else if (p->queueLV == 2)
  {
    if (p->priority > 0)
      p->priority--;
    p->tickProc = 0;
  }
  return;
}
```

process가 queue 에서 할당된 time quantum을 다 쓴 경우를 demote로 관리하였다.

L0 queue 에 있던 경우 (if (p->queueLV == 0)) L1 queue로 넣어주었으며, 여기서 끝내버린다면 해당 프로세스가 L0, L1 에 모두 존재하는 상황일 것이다. (L0 queue 내에서 지워주지 않았으므로)

이건 의도에 어긋나는 상황이므로 방지하기 위해 deleteElt를 통해 현재 demote를 실행하고 있는 프로세스를 L0에서 찾아 delete 하였다.

(L1 queue 에 있던 경우도 위와 같게 생각하여 처리해주었다.)

L2 queue 에 있던 경우는 queue 간 이동이 없으니 priority 만 감소시킴으로써 관리해주었다.

//MLFQ scheduler

```
void
MLFQ_scheduler(struct proc *p, struct cpu *c)
{
  if ((p = dequeue(&L[0])))
  {
    //L0 queue에 process가 있는 경우
```

```
  acquire(&p->lock);
  if (p->state == RUNNABLE)
  {
    p->state = RUNNING;
    c->proc = p;
    swtch(&c->context, &p->context);
    c->proc = 0;
  }
  release(&p->lock);
}
else if ((p = dequeue(&L[1]))) //L0 queue에 process가 없어서 L1 queue 관찰
{
  //L1 queue에 process가 있는 경우
  acquire(&p->lock);
  if (p->state == RUNNABLE)
  {
    p->state = RUNNING;
    c->proc = p;
    swtch(&c->context, &p->context);
    c->proc = 0;
  }
  release(&p->lock);
}
else if (L[2].capacity) //L1 queue에 process가 없어서 L2 queue 관찰
{
  //L2 queue에 process가 있는 경우
  struct proc *this = 0;
  int idx = 0;

  for (int i = L[2].front; i < L[2].front+NPROC; i++) {
    p = L[2].data[i%NPROC];
    if (!p)
      continue;
    acquire(&p->lock);
    if (p->state == RUNNABLE) {
      if (this == 0 || p->priority > this->priority) {
        if (this)
          release(&this->lock);
```

```
          this = p;
          idx = (i+1)%NPROC;
          continue;
        }
      }
      release(&p->lock);
    }

    L[2].front = idx%NPROC;

    if (this)
    {
      this->state = RUNNING;
      c->proc = this;
      swtch(&c->context, &this->context);
      c->proc = 0;
      release(&this->lock);
    }
  }
  else
  {
    for (struct proc *p = proc; p<&proc[NPROC]; p++){
      if (p && p->state == RUNNABLE){
        enqueue(p,L);
      }
    }
    globaltick = 0;
    intr_on();
    asm volatile("wfi");
  }
}
```

1) L0 내에 프로세스가 존재하는 상황 (if ((p = dequeue(&L[0]))))

dequeue에서 front 부터 돌아 현재 RUNNABLE 프로세스를 pick 했다.

이는 queue 내에서 가장 앞 쪽에 위치해있으며 실행될 준비가 되어 있는 프로세스를 꺼냈음을 보장한다고 생각한다.

그래서 p = dequeue(&L[0]) 로 p에 L0 내 가장 앞 쪽의 프로세스를 할당해주고 이를 if로 0인지 체크하여 문제가 없는 상황이라면 context switch를 진행해주었다.

2) L0에 존재하지 않으며 L1 내에 프로세스가 존재하는 상황 (else if ((p = dequeue(&L[1]))))
1) 의 상황과 사실상 동일하므로 똑같이 진행해주었다.

3 L0, L1에 존재하지 않으며 L2 내에 프로세스가 존재하는 상황 (else if (L[2].capacity))

이 상황은 앞서 본 1), 2) 와는 차이가 있다.

간단하게 queue 내 가장 앞 쪽 프로세스를 pick 하는 것이 아니라 priority가 가장 큰 프로세스를 pick 해주어야 한다.

이를 위해 for 문으로 L2 의 front부터 한 바퀴 돌게 loop condition을 설정함으로써 queue 내에서 가장 priority가 큰 process를 찾는 logic을 사용하였다. (FCFS scheduler에서의 idea를 적용하여 그와 유사하게 구현하였다.)

priority가 같은 프로세스가 여럿 있는데도 이전에 pick 된 프로세스가 계속 pick 되는 문제가 있었다.

이를 해결하기 위해 search 하며 선택된 프로세스를 this 에 넣는 타이밍에 그 프로세스가 queue에 있었던 위치에서 한 칸 뒤쪽을 idx에 남겨놓았다.

for 문을 다 돌고 난 뒤 front 값을 idx로 update 함으로써 이전에 실행된 프로세스는 가장 마지막에 찾게 되므로 priority가 같은 프로세스가 있다면 이전에 실행된 프로세스는 잡히지 않을 것이다.

해당 방식으로 코드를 작성하여 위 문제를 해결하였다.

4) L0, L1, L2 queue 내에 프로세스가 없는 경우 (else)

MLFQ mode 로 설정한 후 test 를 진행해봤을 때 프로세스가 모두 끝났는데도 프로그램이 끝나지 않고 무한 대기에 빠지는 문제가 발생했다.

왜인지 분석하기 위해 printf 로 상태를 하나하나 출력해보며 분석한 결과 이는 queue 내 프로세스들은 전부 끝났으나 pid 3 (메인 프로그램을 실행시키고 있는 프로세스)는 queue 내에 들어온 적이 없으니 RUNNABLE 상태임에도 scheduler의 선택을 받지 못해 실행되지 못하고 기다리고 있어 발생하는 문제임을 알아챘다.

이를 해결하기 위해 나도 모르게 queue 내에는 프로세스가 존재하지 않지만 RUNNABLE state 프로세스가 존재하는 상황이 있을 수 있으니 for loop으로 proc[NPROC] 를

search 해보고 있다면 L0 queue에 집어넣었다.

이렇게 하면 pid 3 process가 MLFQ scheduler 의 선택을 받을 수 있으니 해당 문제를 해결할 수 있었다.

이 else 문에 진입한 상황은 모든 작업이 끝난 상황이니 global tick 을 0으로 초기화 해주었다.

//checkDelete

```
void
checkDelete(void)
{
  for (int i=0; i<3; i++){
    for (int j=0; j<NPROC; j++){
      if (L[i].data[j] && L[i].data[j]→state == ZOMBIE){
        L[i].data[j] = 0;
        if (L[i].capacity > 0)
          L[i].capacity--;
      }
    }
  }
  return;
}
```

스케줄링 이후 프로세스가 끝나 ZOMBIE state인 프로세스가 생길 수 있다.

해당 state의 프로세스들을 queue 에서 빼냄으로써 올바르게 관리하기 위해 해당 함수를 design 하였다.

for (int i=0; i<3; i++) 를 통해 L0, L1, L2 전체를 보고자 하였고, queue 내 몇 번째 인덱스의 프로세스가 ZOMBIE state일지 모르므로 for (int j=0; j<NPROC; j++) 로 queue 내 전체를 탐색하여 프로세스를 queue에서 빼냈다.

//scheduler

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

```
  struct proc *p = 0;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    // The most recent process to run may have had interrupts
    // turned off; enable them to avoid a deadlock if all
    // processes are waiting.
    intr_on();

    if (schedState){
      FCFS_scheduler(p,c);
    }
    else {
      //terminated 된 process 관리
      checkDelete();
      MLFQ_scheduler(p,c);
    }
  }
}
```

scheduler 에서는 schedState 변수로 실행할 scheduler mode를 나누었다.

schedState를 처음에 1로 초기화하여 default scheduler가 FCFS가 되도록 하였다.

schedState == 0 (MLFQ)인 경우에는 실행될 때마다 MLFQ scheduler 이전에 queue 내 ZOMBIE state 프로세스 (terminated된 프로세스)가 있는지 체크하고, 우리는 RUNNABLE state 프로세스들만을 queue에 넣어놓아야 하므로 ZOMBIE state는 queue 내에서 delete 해주었다.

4. trap.c

//clockintr

```
void
clockintr()
{
  struct proc *p = myproc();
  if(cpuid() == 0){
    acquire(&tickslock);
```

```
    ticks++;
    if (p && p->state == RUNNING){
      globaltick++;
      p->tickProc++;
    }

    if (globaltick == 50){
      priorityBoosting();
      //지금 running 중인 process는 queue에 못들어가고 붕 뜨는거 아닌가?
      if (p && p->state == RUNNING){
        p->queueLV = p->tickProc = 0;
        p->priority = 3;
        enqueue(p,&L[0]);
        //해당 프로세스가 cpu를 계속 잡고 있기 때문에 이 프로세스부터 실행된다.. 이 문제
        flag = 1;
      }
    }
  }
  wakeup(&ticks);
  release(&tickslock);

  // ask for the next timer interrupt. this also clears
  // the interrupt request. 1000000 is about a tenth
  // of a second.
  w_stimecmp(r_time() + 1000000);
}
```

clockintr는 timer interrupt가 발생한 경우 호출되는 함수이다.

즉, 이 함수가 실행될 때는 tick이 +1 되는 상황이라고 생각해볼 수 있다.

이 흐름을 따라 현재 clockintr를 실행하고 있는 프로세스의 tick (현재까지 해당 큐에서 사용한 시간을 의미)과 global tick을 1씩 증가시켜주었다.

증가시켜줬을 때 global tick이 50이 되어버리는 경우 priority boosting을 진행해주어야 하므로 priorityBoosting()을 호출하였다.

앞서 priority boosting 에 대한 design, logic 을 작성한 부분에서 말했었던 현재 실행되고 있는 이 프로세스는 boosting이 모두 끝난 후에 enqueue를 통해 집어넣었다. (현재 실행되고 있는 프로세스를 queue 맨 끝에 넣어주기 위함이다.)

이렇게 한 후 test를 진행해보았을 때 L0 queue의 순서를 무시하고 priority boosting이 진행된 당시 프로세스가 실행되는 문제가 발생하였다.

해당 문제가 발생하는 이유는 현재 RUNNING state의 process에 대해 cpu를 뺏어가지 않았기 때문이라 생각하여 이 상황을 control 하기 위해 flag 라는 변수를 두었다.

(flag == 1 → RUNNING process 의 cpu를 뺏으라고 알리는 의도)

//usertrap

```
void
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(killed(p))
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sepc, scause, and sstatus,
    // so enable only now that we're done with those registers.
```

```
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else {
    printf("usertrap(): unexpected scause 0x%lx pid=%d\n", r_scause(), p->pid
    printf("            sepc=0x%lx stval=0x%lx\n", r_sepc(), r_stval());
    setkilled(p);
  }

  if(killed(p))
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
  {
    if (flag){
      flag = 0;
      yield();
    }
    if (schedState == 0){
      if (p->queueLV == 0 && p->tickProc == 1){
        demote();
      }
      else if (p->queueLV == 1 && p->tickProc == 3){
        demote();
      }
      else if (p->queueLV == 2 && p->tickProc == 5){
        demote();
      }
      yield();
    }
  }

  usertrapret();
}
```

usertrap과 kerneltrap 에서는 timer interrupt가 발생한 상황 (which_dev == 2)에 대해서 수정하였다.

위 clockintr 에서 말한 문제를 cover 하고자 If (flag) 로 flag == 1 인 상황을 체크한 후 yield로 cpu를 뺏어옴으로써 문제를 해결하였다.

MLFQ의 경우 queue에서 할당된 time quantum을 모두 사용하였으면 하위 queue로 내려주거나 priority를 감소시켜줘야 하므로 if, else if 문을 통해 위 조건이 만족하는지 확인 후 demote를 호출하여 진행하였다.

FCFS의 경우에는 non-preemptive 하게 진행하며 MLFQ의 경우에는 RR이므로 tick마다 cpu를 넘겨주어야 한다. 이를 구분지어 구현하기 위해서 yield()를 if (schedState == 0) 안으로 넣어서 해결하였다.

//kerneltrap

```
void
kerneltrap()
{
  int which_dev = 0;
  uint64 sepc = r_sepc();
  uint64 sstatus = r_sstatus();
  uint64 scause = r_scause();

  if((sstatus & SSTATUS_SPP) == 0)
    panic("kerneltrap: not from supervisor mode");
  if(intr_get() != 0)
    panic("kerneltrap: interrupts enabled");

  if((which_dev = devintr()) == 0){
    // interrupt or trap from an unknown source
    printf("scause=0x%lx sepc=0x%lx stval=0x%lx\n", scause, r_sepc(), r_stva
    panic("kerneltrap");
  }

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2 && myproc() != 0){
    struct proc *p = myproc();
    if (flag){
      flag = 0;
```

```
      yield();
    }
    if (schedState == 0){
      if (p->queueLV == 0 && p->tickProc == 1){
        demote();
      }
      else if (p->queueLV == 1 && p->tickProc == 3){
        demote();
      }
      else if (p->queueLV == 2 && p->tickProc == 5){
        demote();
      }
      yield();
    }
  }

  // the yield() may have caused some traps to occur,
  // so restore trap registers for use by kernelvec.S's sepc instruction.
  w_sepc(sepc);
  w_sstatus(sstatus);
}
```

usertrap 과 동일하게 수정하였다.

(2) Results

```
$ test
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

Error: CPU scheduler is already FCFS mode.
nothing has been changed
successfully changed to MLFQ mode!

[Test 2] MLFQ Scheduling
Process 8 (MLFQ L0-L2 hit count):
L0: 7942
L1: 35078
L2: 56980
Process 9 (MLFQ L0-L2 hit count):
L0: 11521
L1: 35167
L2: 53312


Process 11 (MLFQ L0-L2 hit count):
L0: 11655
L1: 34695
L2: 53650
Process 10 (MLFQ L0-L2 hit count):
L0: 11692
L1: 34939
L2: 53369
[Test 2] MLFQ Test Finished

FCFS & MLFQ test completed!
```

[Figure 1 : 제공된 Project01_test 내 test.c 실행 결과]

해당 실행 결과에서는 process의 개수가 4개밖에 없어 time quantum을 1,3,5 로 나눈 의도와 같이 각 queue 별로 있던 시간이 1:3:5 의 비율과 비슷하게 나타났다.

하지만 process 개수를 8개로 늘려봤을 때 프로세스 개수가 많아 오래 걸리며 이때 priority boosting이 일어나며 다시 L0 queue로 올라가는 상황이 겹쳐 각 queue 별로 있던 시간이 1:3:5 를 유지하지 못하고 1:5:3 과 같이 나타나기도 하였다.