ELE3021_project01_12298_2019042497

Design

1. MLFQ를 위해 각 process들이 추가로 가지고 있어야 하는 값

   해당 process가 속한 **queue의 level**, 각 process가 해당 queue에서 실행되고 있던 시간인 **time quantum**, 해당 process의 우선순위를 담는 **priority**, 각 queue에 들어온 순서를 담는 변수 **order**, schedulerLock()을 호출한 process임을 표시하는 값인 **qualification**이 필요하다고 생각했습니다. 따라서 각 process의 상태를 담는 구조체 proc 안에 각각 구현을 해줄 것입니다. 또한, 각 queue마다 순서를 지정하기 위해서 (가장 끝에 할당된 숫자 +1)의 값을 가지고 있는 전역변수를 각 queue마다 만들어줄 것입니다.

   그리고 이 값들은 allocproc() 함수 안에 초기화를 각각 해줄 것입니다. q_level, t_quantum, qualification은 0으로, priority는 3으로, order에는 전역변수로 L0 queue의 순서 번호를 가지고 있는 값을 넣어주고 1 증가해서 다음 순서 번호를 대비해줄 것입니다.

2. L0, L1, L2의 구현

   기본적으로 scheduling이 일어날 때마다 scheduler() 함수에서 모든 ptable에서 RUNNABLE한 process를 탐색해 가장 우선순위가 높은 최적의 process를 찾아내는 방식으로 구현할 것입니다. 처음 우선순위는 scheduler lock이 걸린 process, 다음부터는 q_level이 더 낮은 process, q_level이 같다면 L0, L1에서는 queue에 들어온 순서가 빠른 process, L2에서는 priority가 더 높은(값이 작은) process, priority까지 같다면 들어온 순서가 빠른 process 순서로 scheduling을 해줄 것입니다. 자세한 알고리즘은 Implement에서 부연 설명하겠습니다.

3. priority boosting

   기존의 ticks를 0으로 계속 초기화하기엔 제대로 된 컴퓨터 동작이 안전하게 이루어지지 않을 수 있다는 생각이 들어서 따로 global_ticks라는 변수를 만들어서 ticks와 같은 단위로 증가는 하지만 boosting이 일어날 때 초기화 해도 전체 xv6에는 안전성을 보장하게 해줄 것입니다. 추가적으로 전역 변수를 통해 각 queue의 order를 저장하는 값에서 overflow가 일어나지 않도록 초기화를 해줄 생각입니다. 이에 각 process마다 priority boosting할 때 임시로 순서를 담는 변수가 필요하다고 생각되어 넣어줄 것입니다. 이후 모든 process들이 L0으로 초기화됨과 동시에 전역 변수의 값도 L0에 현재 process의 개수+1, L1과 L2에는 1의 값을 넣어 초기화해줄 것입니다.

   그리고 중요한 것은 L0으로 합칠 때의 순서가 L0,L1,L2의 queue에 있던 기존 순서를 지키면서 재조정되어야 하기 때문에 모든 process가 조정될 때까지 반복하면서 우선순위대로 정렬을 해서 순서를 지켜줄 것입니다. 그 이후 global_ticks 값도 0으로 초기화를 해줄 것입니다.

4. schedulerLock(), schedulerUnlock()

schedulerLock()이 호출되면 password가 학번과 같을 경우, 각 process 안의 qualification이라는 값에 1을 주어서 해당 process가 최우선적으로 처리되어야 할 자격을 줄 것입니다. 실행되는 방식은 1tick마다 scheduler() 함수 안에는 항상 들어오지만 다음 넘겨줄 process를 탐색을 하다가 qualification 값에 1인 process가 있으면 무조건 이 process를 가지고 그 다음에도 실행되도록 해줄 것입니다. schedulerUnlock()이 호출되면 password가 학번과 같으면서 해당 process가 지금 lock을 잡고 있었다면 qualification 값을 0으로 바꿔주고, queue level을 0, priority를 3, time quantum을 0으로 초기화해줄 것입니다. 또한 L0의 가장 앞의 순서로 와야 하기 때문에 order에 0 값을 줄 것입니다.

Implement

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int q_level;                 // queue level
  int t_quantum;               // time quantum
  int priority;                // priority
  uint order;                  // L0, L1, L2 안에 들어온 order
  uint boosting_tmp;           // priority boosting할 때 임시로 담는 순서
  int qualification;           // 우선 처리되어야 할 qualification
};
```

먼저 proc.h 안의 proc 구조체(각 process가 가지고 있는 값을 나타내는 구조체) 안에 새로운 값들을 추가해주었습니다.

차례대로 q_level은 queue level, t_quantum은 time quantum, priority는 priority, order는 L0, L1, L2 안에 들어온 order, boosting_tmp은 priority boosting할 때 임시로 담는 순서, qualification은 우선 처리되어야 할 자격을 담는 변수입니다.

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->q_level = 0;
  p->t_quantum = 0;
  p->priority = 3;
  p->order = MLFQ_order[0]++;
  p->boosting_tmp = 0;
  p->qualification = 0;
```

각 값들의 초기화는 proc.c 안의 allocproc(void)의 해당 위치에서 해주었습니다.

q_level, t_quantum, boosting_tmp, qualification은 각각 0을 넣어주었고, priority는 3을 넣어주었습니다. 또한, order는 전역변수로 각 queue에 들어온 순서를 담고 있는 MLFQ_order 변수에서 L0을 의미하는 MLFQ_order[0] 값을 넣고 다음을 위해 해당 전역 변수 값을 하나 증가시켜주었습니다.

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER){
  global_ticks++;                                          // global_tick 1 증가
  myproc()->t_quantum++;                                   // 해당 process의 time quantum 1 증가

  if(myproc()->t_quantum == 2*(myproc()->q_level)+4){      // 만약 해당 process의 time quantum이 2*n + 4일 경우
    if(myproc()->q_level == 2){                            // L2 queue에 있는 process라면
      if(myproc()->priority != 0){                         // priority가 0이 아니라면
        setPriority(myproc()->pid, (myproc()->priority)-1);// priority를 1 감소
      }
      myproc()->t_quantum = 0;                             // 해당 process의 time quantum을 0으로 초기화
    }
    else{                                                  // 만약 L0나 L1 queue에 있는 process라면
      myproc()->q_level++;                                 // queue level을 1 증가
      myproc()->t_quantum = 0;                             // 해당 process의 time quantum을 0으로 초기화
      myproc()->order = MLFQ_order[myproc()->q_level]++;   // 해당 process의 순서를 해당 queue의 마지막으로 보냄
    }
  }
  yield();                                                 // 다음 process에게 CPU를 양보
}
```

다음은 trap.c 안에서 현재 실행 중인 process가 있으면서 time interrupt가 발생하여 ticks가 하나 증가되는 상황일 경우를 보겠습니다.

먼저 따로 만들어 놓은 global_ticks와 해당 process의 time quantum 값을 1 증가시켜주었습니다.

그 이후 해당 process가 가지고 있는 time quantum의 값이 각 queue가 가진 time quantum 값과 같은 경우에 대해 처리를 해주었습니다.

만약 queue level이 2인 경우에는 priority 값이 이미 0이 아니라면 setPriority() 함수를 사용해 해당 process의 priority 값을 1 감소하게 하였습니다. 그 이후 time quantum을 초기화해주었는데 이것은 priority와 관계없이 초기화 되도록 해주었습니다.

만약 queue level이 0과 1인 경우에는 queue level을 1 증가하고, time quantum을 0으로 초기화했습니다. 그리고 새로운 queue에 들어간 상황이므로 해당 process의 order를 증가한 해당 queue level의 queue의 마지막으로 들어가도록 해주었습니다.

이후 다음 process에게 CPU를 양보하는 yield() 함수를 실행시켜주었습니다.

여기서 proc.c 안에 들어있는 yield()와 setPriority()에 대해 알아보겠습니다.

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  if (myproc()->q_level != 2)                          // L0과 L1에 있는 process일 경우
    myproc()->order = MLFQ_order[myproc()->q_level]++;  // 그 queue의 마지막 순서로 넣음
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

먼저 yield()는 기존에 있던 ptable.lock을 걸고 실행 중이던 process를 RUNNABLE로 바꾼 이후 sched() 함수를 실행해서 context switch를 하고 ptable.lock를 해제해주는 구문에서 2줄만 추가해 주었습니다. yield() 함수는 과제의 명세에 "다음 프로세스에게 프로세서를 양보합니다."라고 적혀 있기 때문에 구현에 따라 달라질 수 있다고 생각했습니다. 저는 yield() 함수가 본인도 RUNNABLE로 바꾸고 다음 process를 잡기 때문에 각 queue 마다 돌아가는 알고리즘에 의해 구현되어야 한다고 생각했습니다.

먼저 Round-Robin을 사용하는 L0와 L1 queue에서는 해당 process가 1tick만큼 일을 수행하고 yield()가 호출되면 그 queue의 끝 순서로 밀려난다고 구현하였기 때문에 각 queue의 마지막 순서로 이동하도록 해주었습니다. 다시 말해 1tick 만큼 실행되면 그 queue에 들어온 순서 맨 뒤로 보내게 해 같은 우선순위의 다른 process들도 1tick씩 실행되도록 Round-Robin을 구현하였습니다.

이와 달리 priority 스케줄링을 하고, priority가 같은 process끼리는 FCFS로 스케줄링이 되는 L2 queue는 해당 process가 CPU를 놓아 양보해도 다시 스케줄링은 priority -> FCFS 순서로 이루어집니다. 따라서 Round Robin처럼 새로 해당 queue의 끝에 넣어주는 것이 아니라 FCFS의 순서를 가지고 있어야 하기 때문에 L2 queue에 들어온 순서를 유지해주었습니다.

L2 queue에 대한 자세한 스케줄링(priority -> FCFS)은 이후 scheduler() 함수에서 부연 설명을 하겠습니다.

```
// yield 함수의 system call 함수
int
sys_yield(void)
{
  yield();  // yield를 실행
  return 0; // 0을 return
}
```

추가로 yield()를 system call 함수로 만들어주기 위해 해당 함수를 만들어주었습니다.

```
// 해당 pid의 process의 priority를 설정하는 함수
void
setPriority(int pid, int priority)
{
  struct proc *p;                                   // process를 찾는 for문을 돌리기 위해 필요한 process를 담는 변수
  int check = 0;                                    // priority가 설정되었는지 check하는 역할을 하는 변수

  if(priority < 0 || priority > 3)                  // priority의 값이 0~3 사이가 아닌 경우
    cprintf("setPriority error\n");                 // error 문구 출력
  else{                                             // priority의 값이 0~3 사이인 경우
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);                          // ptable에 접근해 값을 수정해야 하기 때문에 lock을 얻음
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){   // ptable의 process 처음부터 끝까지 탐색
      if(p->pid == pid){                            // 해당 pid를 가진 process를 찾으면
        p->priority = priority;                     // 해당 process에 priority 값을 넣음
        check = 1;                                  // priority가 설정되었으므로 check를 1로
        break;                                      // riority가 설정되었으면 반복문 탈출
      }
    }
    release(&ptable.lock);                          // ptable lock을 해제
    if(check == 0)                                  // priority를 설정할 pid가 없었을 경우
      cprintf("setPriority error\n");               // error 문구 출력
  }
}
```

setPriority()는 해당 pid의 프로세스의 priority를 설정하는 함수입니다.

먼저 인자로 입력 받은 priority의 값이 0~3 사이가 아니라면 error 문구를 띄우도록 하였습니다. 만약 priority의 값이 정상 값이라면 for문을 통해 해당 pid의 값을 찾고 그 process의 priority 값을 바꿔주었습니다. 이 때 check 변수를 통해 priority 값이 설정이 되었다면 1을 넣어주었습니다. 만약 인자로 입력 받은 pid의 값을 가진 process가 없다면(check가 0이라면) error 문구를 띄우도록 하였습니다.

```
// setPriority 함수의 system call 함수
int
sys_setPriority(void)
{
  int pid;                          // setPriority 함수가 받는 pid 인자
  int priority;                     // setPriority 함수가 받는 priority 인자

  if(argint(0, &pid) < 0)           // pid 인자가 int로 들어오지 않았다면
    return -1;                      // -1 return해 오류임을 표시
  if(argint(1, &priority) < 0)      // priority 인자가 int로 들어오지 않았다면
    return -1;                      // -1 return해 오류임을 표시
  setPriority(pid, priority);       // 인자를 알맞게 넣어 setPriority를 실행
  return 0;                         // 0을 return
}
```

setPriority()는 system call로 구현되어야 하기 때문에 다음과 같은 함수를 만들어주었습니다. pid와 priority라는 int형 인자를 받았는지 확인을 해야 하기 때문에 각각의 인자가 int형인지 확인을 하도록 해주고 문제가 없다고 실행하도록 했습니다.

다음은 proc.c 안의 scheduler() 함수 안의 중요한 부분을 보겠습니다.

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

  // Loop over process table looking for process to run.
  acquire(&ptable.lock);
  struct proc *new_p = 0;                          // 다음 실행될 process를 담는 변수
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ // ptable의 process 처음부터 끝까지 탐색
    if(p->state != RUNNABLE)                       // RUNNABLE 하지 않다면
      continue;                                    // 다음 process로 넘어감
    if(p->qualification){                          // 만약 qualification이 1인 process가 있다면
      new_p = p;                                   // 실행될 process를 담는 변수에 넣어줌
      break;                                       // lock이기 때문에 무조건 실행되어야 하므로 break
    }
    if(!new_p || new_p->q_level > p->q_level)      // 만약 기존에 아무것도 없었거나, 기존에 있던 process보다 queue level이 낮은 경우
      new_p = p;                                   // 실행될 process를 담는 변수에 넣어줌
    else if(new_p->q_level == p->q_level){         // 기존의 process와 queue level이 같은 경우
      if(p->q_level == 0 || p->q_level == 1)       // L0이나 L1에 있는 process인 경우
        if(new_p->order > p->order)                // 현재 queue 안의 order가 더 낮다면
          new_p = p;                               // 실행될 process를 담는 변수에 넣어줌
      if(p->q_level == 2){                         // L2에 있는 process인 경우
        if(new_p->priority > p->priority)          // priority 값이 더 낮으면
          new_p = p;                               // 실행될 process를 담는 변수에 넣어줌
        else if(new_p->priority == p->priority){   // priority 값이 같다면
          if(new_p->order > p->order)              // 현재 queue 안의 order가 더 낮다면
            new_p = p;                             // 실행될 process를 담는 변수에 넣어줌
        }
      }
    }
  }
```

ptable에 있는 process들 중에서 RUNNABLE한 process 중에서 골라야 하기 때문에 RUNNABLE 하지 않은 process는 continue로 다음 process를 보도록 넘겨주었습니다.

만약 탐색을 하다가 qualification이 1인(scheduler lock이 걸린) process를 만나면 우선적으로 스케줄링 되어야 하므로 바로 골라주고 탐색을 그만두도록 break로 빠져나오게 했습니다.

만약 for문을 탐색하다가 기존에 아무런 process가 없었거나, 기존의 process보다 지금 for문으로 돌고 있는 process가 queue level이 더 낮은 경우 더 먼저 실행되어야 하기 때문에 new_p에 넣어주었습니다.

그렇지 않고 기존의 process와 지금 for문으로 돌고 있는 process가 queue level이 같은 경우에는 우선 순위를 자세히 따져봐야 합니다.

먼저, L0과 L1에 있는 process인 경우 order가 빠른 process가 먼저 실행되는 우선순위가 있으므로 order 값이 낮다면 new_p에 넣어주었습니다.

이에 반해 L2에 있는 process인 경우 먼저 priority 값이 낮은지를 판단해 낮다면 우선순위가 높은 것이므로 new_p에 넣어주었습니다. 만약 여기서 priority 값이 같다면 FCFS로 실행되어야 하기 때문에 order 값이 더 낮을 경우에 new_p에 넣어주었습니다.

이렇게 L0, L1은 Round-Robin으로, L2는 priority -> FCFS로 구현되었습니다.

이렇게 for문을 나오게 되면 다음에 실행될 process가 new_p에 들어있을 것입니다. (RUNNABLE한 process가 없다면 여전히 new_p는 0)

```
// Switch to chosen process.  It is the process's job
// to release ptable.lock and then reacquire it
// before jumping back to us.
if(new_p){
  c->proc = new_p;
  switchuvm(new_p);
  new_p->state = RUNNING;

  swtch(&(c->scheduler), new_p->context);
  switchkvm();

  // Process is done running for now.
  // It should have changed its p->state before coming back.
  c->proc = 0;
}
if(global_ticks == 100){ // global_ticks이 100이 되었다면
  priority_boosting();   // Starvation을 막기 위해 priority boosting
}
release(&ptable.lock);
```

for문을 나오게 되면 그 이후에는 기존에 for문 안에 있던 context switch를 하는 부분을 for문 밖에 꺼내 놓았습니다. 현재 돌아가고 있던 process와 내가 우선순위에 의해 선택한 new_p를 context switch를 진행해 바꿔주는 코드입니다. 기존에 있던 코드이니 자세한 설명은 생략하겠습니다.

마지막에는 global_ticks를 검사해 이 값이 100이 된다면 Starvation을 막기 위해 priority boosting이 실행되어야 합니다. 따라서 global_ticks가 100이 되면 priority_boosting() 함수가 실행되도록 했습니다.

```
// Priority boosting 함수
void
priority_boosting()
{
  struct proc *p;                                  // process를 찾는 for문을 돌리기 위해 필요한 process를 담는 변수
  uint now_l0_order = MLFQ_order[0] - 1;           // 현재 L0이 사용한 order 값
  int check = 1;                                   // 모든 process가 boosting이 되었는지 check하는 변수
  while(check){                                    // check가 1인 경우 while문 반복
    struct proc *tmp = 0;                          // priority boosting 시 순서를 유지하기 위해 다음 순서를 담는 변수
    check = 0;                                     // check를 0으로 설정
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  // ptable의 process 처음부터 끝까지 탐색
      if(p->state != RUNNABLE || p->boosting_tmp != 0)   // RUNNABLE 하지 않거나, 이미 boosting 된 대상이면
        continue;                                  // 다음 process로 넘어감
      check = 1;                                   // 그렇지 않다면 boosting 대상이 있으므로 check를 1로 바꿈
      if(p->qualification){                        // 만약 qualification이 1이라면
        p->qualification = 0;                      // lock을 해제함
        tmp = p;                                   // 다음 boosting의 대상으로 넣음
        break;                                     // 가장 먼저 L0에 들어가야 하기 때문에 for문 탈출
      }
      if (!tmp || tmp->q_level > p->q_level)       // 만약 기존에 아무것도 없었거나, 기존에 있던 process보다 queue level이 낮은 경우
        tmp = p;                                   // 다음 boosting의 대상으로 넣음
      else if (tmp->q_level == p->q_level) {       // 기존의 process와 queue level이 같은 경우
        if (p->q_level == 0 || p->q_level == 1)    // L0이나 L1에 있는 process인 경우
          if (tmp->order > p->order)               // 현재 queue 안의 order가 더 낮다면
            tmp = p;                               // 다음 boosting의 대상으로 넣음
        if (p->q_level == 2) {                     // L2에 있는 process인 경우
          if (tmp->priority > p->priority)         // priority 값이 더 낮으면
            tmp = p;                               // 다음 boosting의 대상으로 넣음
          else if (tmp->priority == p->priority) { // priority 값이 같다면
            if (tmp->order > p->order)             // 현재 queue 안의 order가 더 낮다면
              tmp = p;                             // 다음 boosting의 대상으로 넣음
          }
        }
      }
    }
```

```
    if(check == 0)                                 // 더이상 boosting의 대상이 남아있지 않으면(check가 0)
      break;                                       // while문 탈출
    tmp->boosting_tmp = MLFQ_order[0]++;           // boosting의 대상을 L0의 다음 순서로 넣음
  }
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  // ptable의 process 처음부터 끝까지 탐색
    if(p->boosting_tmp == 0)                       // boosting의 대상이 아니면
      continue;                                    // 다음 process로 넘어감
    p->q_level = 0;                                // queue level 0으로 초기화
    p->priority = 3;                               // priority 3으로 초기화
    p->t_quantum = 0;                              // time quantum 0으로 초기화
    p->order = p->boosting_tmp - now_l0_order;     // order에 1부터 넣음
    p->boosting_tmp = 0;                           // 임시로 순서 담았던 변수 0으로 초기화
  }
  MLFQ_order[0] = MLFQ_order[0] - now_l0_order;    // 현재 L0에 있는 개수로 표시
  MLFQ_order[1] = 1;                               // L1에는 아무 process도 없으므로 1로 초기화
  MLFQ_order[2] = 1;                               // L2에는 아무 process도 없으므로 1로 초기화
  global_ticks = 0;                                // global_ticks 0으로 초기화
}
```

다음은 proc.c 안의 priority_boosting() 함수입니다. Priority boosting이 일어났을 때 모든 process들은 L0 queue로 재조정됩니다. 이때 순서는 L0 -> L1 -> L2의 순서가 유지되어야 합니다. 따라서 저는 스케줄링이 되는 우선순위에 따라서 재조정되도록 구현했습니다. 여기서 주목할 점은 boosting_tmp라는 변수입니다. 이 변수는 탐색을 하면서 이미 boosting이 된 process임을 표시함과 동시에 L0으로 들어가는 순서를 임시로 담을 수 있는 변수입니다.

스케줄링이 되는 우선순위에 따라 먼저 boosting이 되는 tmp이 결정되면 boosting_tmp에 L0의 다음 순서 값을 넣도록 했습니다. 더 이상 boosting의 대상이 되는 process가 없다면 check 변수가 0에서 바뀌지 않을 것이기 때문에 계속 돌던 while문을 탈출할 것입니다.

이후 다시 ptable을 탐색하면서 boosting의 대상인 경우 queue level, priority, time quantum을 명

세에 맞게 초기화하였습니다. 그리고 order의 순서를 1부터 나타내기 위해서 이전에 미리 저장해 놨던 now_l0_order(기존에 L0이 사용했던 번호의 최댓값)의 값을 빼고 order에 넣어주었습니다. 그리고 boosting을 마쳤으므로 boostin_tmp의 값을 0으로 초기화해 boosting의 대상에서 벗어났음을 나타냈습니다.

이후 boosting이 다 끝나면 전역변수 MLFQ_order의 값을 각 queue에 들어있는 개수 +1의 값으로 다시 초기화해주었습니다. 마지막으로 global_ticks의 값을 0으로 초기화했습니다.

```
// 해당 프로세스가 우선적으로 스케줄링 되도록 하는 함수
void
schedulerLock(int password)
{
  int check = 0;                                                    // scheduler lock이 걸린 process가 있는지 확인하는 변수
  if(password == 2019042497){                                       // 암호가 학번과 일치할 시
    struct proc *p;                                                 // process를 찾는 for문을 돌리기 위해 필요한 process를 담는 변수
    acquire(&ptable.lock);                                          // ptable lock
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){             // ptable의 process 처음부터 끝까지
      if(p->qualification == 1 && (p->state == RUNNABLE || p->state == RUNNING)){  // scheduler lock 상태인 process가 있다면
        check = 1;                                                  // check를 1로
        break;                                                      // 찾아서 넣었으면 반복문 탈출
      }
    }
    release(&ptable.lock);                                          // ptable lock 해제
    if (check == 0){                                                // scheduler lock이 걸린 proecss가 없다면 (check가 0이라면)
      myproc()->qualification = 1;                                  // 우선으로 처리되어야 할 자격을 얻음
      global_ticks = 0;                                             // global tick은 priority boosting 없이 0으로 초기화
    }
  }
  if(password != 2019042497 || check == 1){                         // 암호가 일치하지 않거나 이미 scheduler lock이 걸린 process가 있는 경우
    cprintf("pid: %d, time quantum: %d, current queue level: %d\n",
            myproc()->pid, myproc()->t_quantum, myproc()->q_level); // 프로세스의 pid, time quantum, 현재 위치한 큐의 level을 출력
    exit();                                                         // 종료
  }
}
```

다음은 proc.c 안의 schedulerLock() 함수입니다. 기본적으로 스케줄러를 lock하는 process가 존재할 경우 MLFQ 스케줄러는 동작하지 않고, 해당 process가 최우선적으로 스케줄링 되어야 합니다. 그렇기 때문에 저는 해당 함수에서 올바른 password가 들어오고, 이미 qualification이 1인 process가 없다면 scheduler() 함수에서 무조건 먼저 실행되도록 qualification의 값을 1로 바꿔주었습니다. 그리고 성공적으로 실행되었다면, global_ticks를 0으로 초기화 시켜주었습니다.

만약 이미 scheduler lock이 걸린 process가 있거나, 인자로 들어온 password가 제 학번이 아닌 경우에는 해당 process의 pid, time quantum, 현재 위치한 queue의 level을 출력하고 강제로 종료되도록 해주었습니다.

```
// schedulerLock 함수의 system call 함수
int
sys_schedulerLock(void)
{
  int password;                 // schedulerLock 함수가 받는 password 인자

  if(argint(0, &password) < 0)  // password 인자가 int로 들어오지 않았다면
    return -1;                  // -1 return해 오류임을 표시
  schedulerLock(password);      // 인자를 알맞게 넣어 schedulerLock 함수 실행
  return 0;                     // 0을 return
}
```

다른 system call 함수와 마찬가지로 인자를 검사하고 실행하도록 해주었습니다.

```
// 해당 프로세스가 우선적으로 스케줄링 되던 것을 중지하는 함수
void
schedulerUnlock(int password)
{
  if(password == 2019042497 && myproc()->qualification == 1){ // 암호가 일치하고, 자격이 있을 시
    myproc()->qualification = 0;                              // 우선으로 처리되어야 할 자격을 해제
    myproc()->q_level = 0;                                    // L0 queue로 이동
    myproc()->priority = 3;                                   // priority를 3으로 설정
    myproc()->t_quantum = 0;                                  // time quantum 초기화
    myproc()->order = 0;                                      // L0 queue의 가장 앞 순서로 지정
  }
  else{                                                       // 암호가 일치하지 않거나 자격이 없을 시
    cprintf("pid: %d, time quantum: %d, current queue level: %d\n",
            myproc()->pid, myproc()->t_quantum, myproc()->q_level);
    // 프로세스의 pid, time quantum, 현재 위치한 큐의 level을 출력
    exit();                                                   // 종료
  }
}

// schedulerUnlock 함수의 system call 함수
int
sys_schedulerUnlock(void)
{
  int password;                 // schedulerUnlock 함수가 받는 password 인자

  if(argint(0, &password) < 0)  // password 인자가 int로 들어오지 않았다면
    return -1;                  // -1 return해 오류임을 표시
  schedulerUnlock(password);    // 인자를 알맞게 넣어 schedulerUnlock 함수 실행
  return 0;                     // 0을 return
}
```

다음은 proc.c 안의 schedulerUnlock() 함수입니다. 이 함수는 우선적으로 처리되어야 할 process의 자격을 해제하기 위한 함수입니다. 따라서 qualification이 1인 (scheduler lock이 걸린) process가 아닌 process가 호출하면 오류로 처리하도록 해주었습니다. 그래서 인자로 받은 password와 qualification이 알맞은지 확인하고, 맞다면 qualification을 해제해야하기 때문에 0으로 초기화를 해주고, L0 queue의 가장 앞으로 이동해야하기 때문에 q_level과 order도 0으로 초기화해주었습니다. 그리고 priority는 3, time quantum은 0으로 초기화해주었습니다.

password가 제 학번이 아니거나 scheduler lock이 걸린 process가 아닌 경우에는 process의 pid, time quantum, 현재 위치한 queue의 level을 출력하고 강제로 종료되도록 해주었습니다.

```
SETGATE(idt[129], 1, SEG_KCODE<<3, vectors[129], DPL_USER);
SETGATE(idt[130], 1, SEG_KCODE<<3, vectors[130], DPL_USER);
```

```
if(tf->trapno == 129){
  schedulerLock(2019042497);
}
if(tf->trapno == 130){
  schedulerUnlock(2019042497);
}
```

추가로 각각 129, 130번 인터럽트를 호출할 경우 실행되어야 하기 때문에 해당 조건은 trap.c에 추가해주었습니다.

```
// process가 속한 queue의 level을 반환하는 함수
int
getLevel(void)
{
  int level = myproc()->q_level;  // 해당 process의 queue level을 level에 담음
  return level;                   // queue level 반환
}

// getLevel 함수의 system call 함수
int
sys_getLevel(void)
{
  return getLevel(); // getLevel()에서 return 된 queue level 값을 return
}
```

마지막으로 proc.c 안의 getLevel() 함수입니다. 단순하게 해당 process의 q_level 값을 return 하도록 해주었습니다. 그리고 system call 함수에서도 그 return 값을 고스란히 받아서 return 하도록 해주었습니다.

마지막으로 제가 등록한 system call 함수 5개를 등록한 과정을 설명하겠습니다.

```
void            priority_boosting();
void            yield(void);
extern uint     global_ticks;
extern uint     MLFQ_order[3];
int             getLevel(void);
void            setPriority(int pid, int priority);
void            schedulerLock(int password);
void            schedulerUnlock(int password);
```

먼저 다른 file에서도 해당 함수를 사용할 수 있도록 defs.h에 다른 file들이 사용할 수 있도록 함수들을 넣어주었습니다. priority_boosting()은 scheduler() 함수에서만 실행하지만 해당 함수 밑에 구현해놓았기 때문에 넣어주었습니다. 또한 global_ticks와 MLFQ_order[3] 변수는 proc.c에 있지만 trap.c에서도 사용할 수 있게 하기 위해서 정의를 넣어주었습니다.

이후 앞서 설명했듯 해당 함수 밑에 wrapper function들을 만들어주었습니다.

```
#define SYS_yield  23          [SYS_yield]   sys_yield,                  extern int sys_yield(void);
#define SYS_getLevel 24        [SYS_getLevel]  sys_getLevel,            extern int sys_getLevel(void);
#define SYS_setPriority 25     [SYS_setPriority] sys_setPriority,       extern int sys_setPriority(void);
#define SYS_schedulerLock 26   [SYS_schedulerLock] sys_schedulerLock,   extern int sys_schedulerLock(void);
#define SYS_schedulerUnlock 27 [SYS_schedulerUnlock] sys_schedulerUnlock, extern int sys_schedulerUnlock(void);
```

다음은 syscall.h과 syscall.c에 wrapper function을 등록해주었습니다.

```
void yield(void);                          SYSCALL(yield)
int getLevel(void);                        SYSCALL(getLevel)
void setPriority(int pid, int priority); SYSCALL(setPriority)
void schedulerLock(int password);          SYSCALL(schedulerLock)
void schedulerUnlock(int password);        SYSCALL(schedulerUnlock)
```

마지막으로 user program이 사용할 수 있도록 user.h에 해당 함수들을 넣고 usys.S에도 넣어주었습니다.

Result

```
MLFQ test start
[Test 1] default
Process 4
pid :4, L0: 6797
pid :4, L1: 9997
pid :4, L2: 83206
Process 5
pid :5, L0: 13502
pid :5, L1: 20259
pid :5, L2: 66239
Process 6
pid :6, L0: 18814
pid :6, L1: 20332
pid :6, L2: 60854
Process 7
pid :7, L0: 19931
pid :7, L1: 30440
pid :7, L2: 49629
[Test 1] finished
```

조교님께서 제공해주신 코드를 우리 과제 명세에 맞게 바꾸어 실행해보았습니다.

각 process 당 getLevel()을 100,000번씩 실행시켰기 때문에 각 process마다 L0, L1, L2에 담긴 값의 합은 항상 100,000개가 되어야 합니다. 해당 조건을 만족한 것을 확인했습니다.

또한, fork()가 될 때마다 sleep(10)이 있기 때문에, 해당 queue에서의 순서가 바뀐 상태에서 priority boosting이 일어나는 등의 특수한 경우가 생기지 않는 이상 주로 fork()가 일어난 순서대로 끝나는 것을 확인할 수 있었습니다.

가끔 L0이나 L1에서 Round-Robin의 구현으로 인해 순서가 바뀐 상태에서 priority boosting이 일어나면 순서가 이대로 지켜지기 때문에 다른 process가 먼저 L1, L2로 가게 되고, 먼저 끝나게 되는 경우도 발생했습니다. 하지만 그것은 Round-Robin의 구현이기 때문에 자연스러운 결과라고 생각합니다.

```
[Test 2] priority
pid: 8, priority: 0
pid: 9, priority: 1
pid: 10, priority: 2
pid: 11, priority: 3
Process 8
pid :8, L0: 13295
pid :8, L1: 19942
pid :8, L2: 66763
Process 9
pid :9, L0: 18550
pid :9, L1: 28539
pid :9, L2: 52911
Process 10
pid :10, L0: 19774
pid :10, L1: 29389
pid :10, L2: 50837
Process 11
pid :11, L0: 20581
pid :11, L1: 29789
pid :11, L2: 49630
[Test 2] finished
```

다음은 각 process마다 priority를 설정해주었을 때 잘 돌아가는지 확인하는 코드입니다. 해당 코드 또한 조교님께서 주신 코드를 사용하여 우리 과제 명세에 맞게 바꾸어 사용해주었습니다.

각 process 당 getLevel()을 100,000번씩 실행시켰기 때문에 각 process마다 L0, L1, L2에 담긴 값의 합은 항상 100,000개가 되어야 합니다. 해당 조건을 만족한 것을 확인했습니다.

또한 priority가 낮을수록 L2에서 무조건 더 빨리 실행되어야 하기 때문에 priority boosting이 순서가 바뀐 상태에서 일어나지 않는다면 순서대로 끝나는 것을 확인했습니다.

```
[Test 3] 13 yield
Process 12
pid :12, L0: 6560
pid :12, L1: 9563
pid :12, L2: 83877
Process 14
pid :14, L0: 13145
pid :14, L1: 18139
pid :14, L2: 68716
Process 15
pid :15, L0: 16257
pid :15, L1: 24402
pid :15, L2: 59341
Process 13
pid :13, L0: 8857
pid :13, L1: 14665
pid :13, L2: 76478
[Test 3] finished
```

다음은 13번 process가 실행될 때 의도적으로 계속 yield()를 시켜주었습니다.

13번 process가 항상 가장 늦게 끝나는 것을 확인할 수 있었습니다.

```
[Test 4] setPriority
setPriority error
setPriority error
setPriority error
setPriority error
setPriority error
[Test 4] finished
```

setPriority() 함수를 테스트해보았습니다. 설정이 되는 경우 2가지와, priority를 다른 값을 넣거나 pid를 다른 값을 넣어서 되지 않은 경우 5가지를 해봤습니다.

결과적으로 오류가 나서 안 되는 경우가 5번 발생함을 확인했습니다.

```
[Test 5] schedulerLock / Unlock, Normal Case
Process 3 scheduler Lock
Process 3 scheduler Lock success
Process 3 scheduler Unlock
Process 3 scheduler Unlock success
Process 19 scheduler Lock
Process 19 scheduler Lock success
Process 19 scheduler Unlock
Process 19 scheduler Unlock success
[Test 5] finished
```

다음은 schedulerLock(), schedulerUnlock()이 잘 동작하는지 테스트해보았습니다.

scheduler Lock을 먼저 출력하고 schedulerLock()에 성공하면 success를 출력하게 하였고, Unlock도 먼저 출력하고 schedulerUnlock()에 성공하면 success를 출력하게 하였습니다.

예상대로 lock에 걸렸을 경우 해당 process만 동작해 하나의 process가 Unlock이 될 때까지 먼저 다 끝나는 것을 다 확인했습니다.

```
[Test 6] schedulerLock / Unlock, Wrong Case1 : PASSWORD ERROR
Process 20 scheduler Lock, Wrong Password
pid: 20, time quantum: 0, current queue level: 0
Process 21 scheduler Lock
Process 21 scheduler Lock success
Process 21 scheduler Unlock, Wrong Password
pid: 21, time quantum: 1, current queue level: 0
[Test 6] finished
```

이번 테스트는 잘못된 password가 입력되었을 때 오류 처리가 잘 되는지 테스트해보았습니다. 먼저 이전과 같은 메커니즘으로 구성되어 있는데, wrong password를 출력한 함수 뒤에는 password +1의 값을 인자로 넣어주었습니다. 따라서 오류로 처리되어 각각의 pid, time quantum, queue level을 잘 출력하는 것을 볼 수 있습니다. Lock과 Unlock에서 모두 잘 되는지 확인했습니다.

```
[Test 7] schedulerLock / Unlock Wrong Case2 : duplication ERROR
Process 22 scheduler Lock
Process 22 scheduler Lock success
Process 22 scheduler Lock, duplication
pid: 22, time quantum: 1, current queue level: 0
Process 23 scheduler Lock
Process 23 scheduler Lock success
Process 23 scheduler Unlock
Process 23 scheduler Unlock success
Process 23 scheduler Unlock, duplication
pid: 23, time quantum: 1, current queue level: 0
[Test 7] finished
```

마지막 테스트는 이미 lock이 걸려있는 상황이거나 이미 unlock을 해준 상황에서 또 호출하면 어떻게 동작하는지 테스트해보았습니다.

먼저 Lock을 잡는데 성공하고, duplication 출력을 하고 한 번 더 Lock을 실행했더니 오류로 잘 처리가 된 것을 확인할 수 있었습니다.

마찬가지로 Lock을 걸고, Unlock을 잘 해주었고 그 이후 duplication을 출력하고 한 번 더 Unlock 함수를 호출해주었더니 오류로 처리된 것을 확인할 수 있었습니다.

Trouble shooting

1. priority_boosting() 함수에서의 오류로 인해 xv6의 작동이 정상적으로 되지 않은 경우

해결: 알고리즘 자체가 모든 RUNNABLE한 process들 중에 순서가 바뀌지 않은 process들만 골라서 스케줄링 순서를 반영해주는 방식인데, 이때 for문을 다 돌고도 찾지 못하면 check라는 flag 역할을 하는 변수가 0이 되어 큰 while문을 빠져나와야 합니다. 하지만 그것을 해결해주는 코드가 없어서 할당되지 않은 tmp에 순서를 넣는 상황이 발생하였고, 오류가 났습니다.

이에 if (check == 0) break;라는 코드를 for문을 나온 시점에 적어 줌으로써 해당 문제를 해결하였습니다.

2. setPriority() 함수의 에러처리가 모호한 경우

해결: 원래 priority error인 경우와 pid error인 경우를 따로 출력하려 했으나 양쪽 다 문제인 경우의 처리가 모호해지는 문제가 발생하여, 그냥 모든 error를 "setPriority error"로 통일해 출력하게 하였습니다.

3. (테스트 7번) 이미 schedulerLock을 호출해 우선으로 처리되어야 하는 process가 있는 시점에 다른 process가 schedulerLock을 호출한 경우

해결: 이미 우선으로 처리해야 하는 process가 있는 경우 오류로 처리해야 하기 때문에 schedulerLock() 함수 안에서 모든 RUNNABLE하거나 RUNNING한 process들 중에서 qualification 이 이미 1인 process가 있다면 schedulerLock 오류로 처리해 강제 종료를 하도록 만들었습니다.

4. L2는 FCFS로 구현을 했어야 하나, L0과 L1처럼 1tick만큼 실행하면 그 queue의 뒤로 넣어주는 Round Robin 방식으로 구현한 경우

해결: yield() 안에서 다음 순서로 넘겨줄 때, queue의 level이 2인 경우만 제외하고 해당 queue의 마지막으로 순서로 들어가도록 하였습니다.