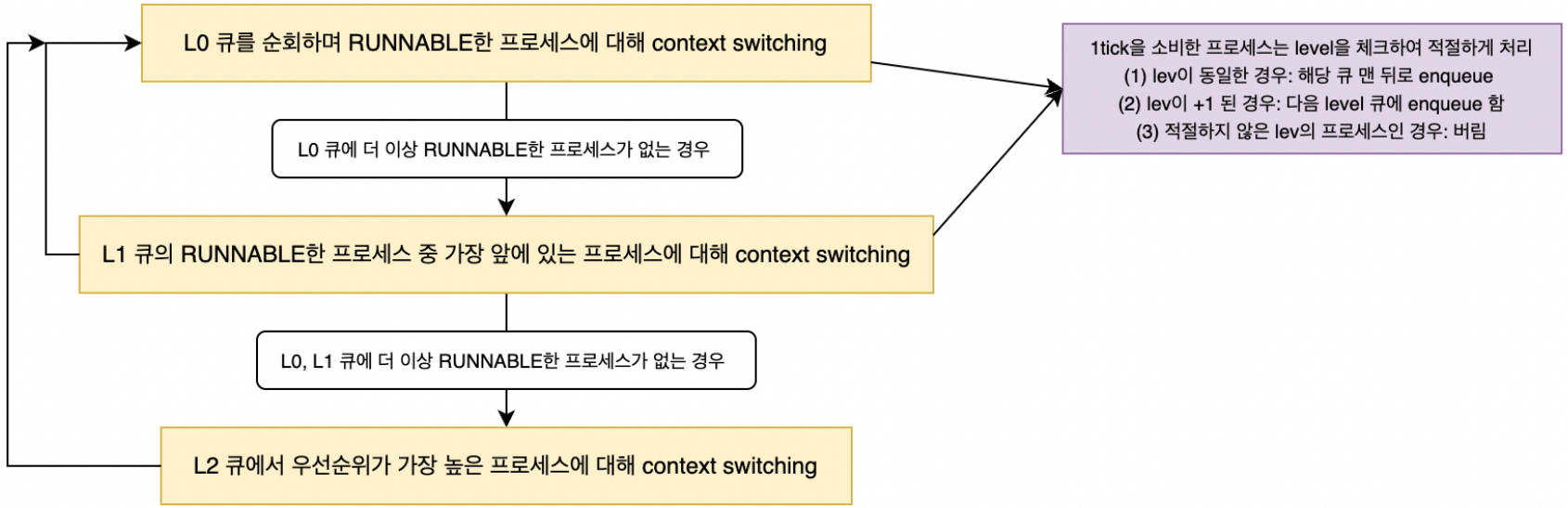
🎨 Design

아래와 같은 구조로 디자인 하였습니다.

```
scheduler(void)
    for(;;) {
```




```
    }
```

디자인 단계에서 고민한 부분은 아래와 같습니다.

1. MLFQ 구조에 대한 고민

(1) queue 자료구조를 꼭 사용해야 할까?

- ptable의 프로세스를 처음부터 그대로 순회하고 level에 따라서 처리만 해 주면 되지 않을까?
- allocproc의 과정을 보면, ptable을 처음부터 순회하면서 UNUSED 자리가 있을 때 프로세스를 초기화 해 주고 submit 하게 됩니다.
- 프로세스가 차곡차곡 쌓이는 동안은 ptable의 앞쪽 프로세스가 뒤쪽 프로세스보다 먼저 생성되었다고 할 수 있지만, 프로세스의 생성-소멸이 반복되다 보면 ptable 의 인덱스가 낮다고 해서 먼저 생성된 프로세스라고 할 수 없습니다.

- 그런데 우리 프로젝트에서는 같은 레벨의 큐 안에 있는 프로세스는 생성 순서에 따라 처리되어야 하는 규칙이 있습니다.
- 따라서 이 방법은 기각하고, 큐 자료구조를 생성하기로 결정하였습니다.

*그렇다면, ptable을 없애고 애초에 allocproc가 L0 큐에 프로세스를 할당하도록 할까?

➔ ptable을 큐로 바꾸면 너무 많은 곳에서 수정이 필요합니다.
kill, procdump, exit, wait와 같은 함수에서 스케줄링 목적이 아니라 필요한 프로세스를 순회하며 찾기 위해 ptable을 사용하고 있습니다.

이러한 이유로 ptable을 삭제하는 것은 적절하지 않다고 판단하고, L0, L1, L2 큐를 따로 구현하여 스케줄러에서 사용하기로 결정하였습니다.

(2) 큐를 구현하기 위해 어떤 자료구조를 사용할까?

각 레벨의 큐에서 FIFO를 보장하기 위해 실제 물리적인 큐가 필요하다는 판단 하에, 큐 자료구조를 구현하고자 했습니다. xv6는 최대 64개의 프로세스만을 감당할 수 있도록 구현되어 있으므로, 정적 배열을 이용한 원형 큐를 사용하였습니다.

2. Scheduler 로직에 대한 고민

(1) 새로 생성된 프로세스의 반영

새로 생성된 프로세스는 L0 큐에 들어가는데, 스케줄러가 L0 큐를 "언제"확인하게 할지 고민하였습니다. 예를 들어, L1큐에서 스케줄링 된 A 프로세스에서 fork()로 B 프로세스가 생성되었고, L1에 A뒤 C, D 프로세스가 남아 있을 때,

A -> B -> C -> D

A -> C -> D -> B

두 가지 순서가 가능합니다. 명세에 의하면 하위 레벨의 큐의 프로세스가 처리되는 중에도 상위 레벨의 큐를 항상 체크해야 하므로, A -> B -> C -> D의 순서로 프로세스를 처리하도록 디자인하였습니다.

(2) 큐에서 프로세스를 언제 제거해야 할까?

처음에는 큐에서 프로세스를 하나씩 꺼내며 RUNNABLE한 프로세스면 RUNNING으로 바꾸어 context switching하고, 아니라면 큐에서 제거하는 것으로 디자인하였습니다. 그런데 프로세스 state 중에는 RUNNABLE이 아니지만 다시 RUNNABLE로 전환될 여지가 있는 state가 있고, 이 프로세스는 큐 대기 순서를 유지한 채 그대로 큐에 남아 있어야 했습니다. 따라서 UNUSED (프로세스가 완전히 종료된 상태), ZOMBIE (프로세스가 종료되고 부모 프로세스에서 회수되기를 기다리는 상태) state 여부를 기준으로 큐에서 제거하거나 큐의 뒤쪽으로 밀어 넣는 로직을 구현하였습니다. 다만 제 로직에서 time quantum을 모두 사용하지 않은 프로세스는 1tick을 사용한 순간 큐의 마지막 순서로 밀려나므로, stride 스케줄링이 반영되어 있습니다.

예를 들어,
L0
L1 A B C
L2
에서 A를 처리한 후 timer interrupt가 발생합니다.
L0 D
L1 B C A
L2
새로운 프로세스가 L0 큐에 들어와서 time quantum을 소비할 때까지 실행됩니다. 이 때 L1큐의 순서는 B C A로 조정되어 있으므로, D가 종료된 후에 스케줄러는 B 프로세스를 스케줄링합니다.

(3) scheduler에서는 ptable을 더 이상 사용하지 않는데 ptable.lock을 그대로 사용해도 될까?
Project #1에서는 CPU가 1개임을 가정하므로 lock에 대한 고려가 필요하지 않지만, CPU가 2개 이상인 상황에서도 queue의 접근에 ptable.lock을 사용해도 된다고 판단하였습니다. mlfq queue도 ptable이 담고 있는 주소값을 큐의 모양으로 담았을 뿐 같은 주소값에 접근하기 때문입니다.

3. SchedulerLock(), SchedulerUnlock()의 비밀번호 파라미터
schedulerLock(), schedulerUnlock()의 경우 시스템 콜 뿐 아니라 인터럽트로도 구현되어야 하는데, 인터럽트 구현 시 비밀번호를 하드코딩하면 보안상 좋지 않다고 생각했습니다. 커널 소스 코드에 접근할 수 있는 개발자는 Makefile에도 접근할 수 있는 maintainer라고 생각하여, 컴파일 옵션으로 비밀번호를 define하게 구성하였습니다. 여전히 시스템 콜에 대해서는 비밀번호를 자유롭게 입력할 수 있습니다.

4. Priority Boosting 방법에 대한 고민
위에서 언급한 UNUSED, ZOMBIE 프로세스를 큐에서 제거하는 로직은, L0, L1의 경우엔 스케줄러와 priority boosting 함수에, L2의 경우엔 priority boosting 함수에만 구현되어 있습니다. L0, L1 큐는 항상 큐 맨 앞의 프로세스를 처리하므로 스케줄러에서 바로 dequeue 하여 제거할 수 있지만, L2 큐의 경우 큐를 순회하다가 우선순위를 고려하여 프로세스를 선택하기 때문입니다. Priority Boosting은 명세에 따라 L1, L2 큐의 프로세스를 순서대로 L0 큐에 옮기는데, 그 과정에서 프로세스 레벨 (lev 변수)과 state를 확인합니다. Boosting 시점에 유효한 프로세스만 L0 큐로 옮겨지면서, 프로세스가 정리됩니다.

🛠️ Implement

0. 구조체 변경 및 변수 선언, 큐 구현

MLFQ 구조에 맞게 proc 구조체에 새로운 변수를 선언하였습니다.

```
// 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 // UNUSED로 초기화됨
  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 lev;                     // queue level (0~2)
  uint tq;                     // time quantum
  int priority;                // process priority (for L2 queue)
  int locked;                  // has the process got locked?
};
```

lev: 프로세스가 현재 속한 큐 레벨을 저장합니다.

tq: 프로세스의 time quantum 값을 저장합니다.

priority: 프로세스의 priority 값을 저장합니다.

locked: 프로세스가 스케줄러 lock을 잡고 있는지 여부를 저장합니다. (1: locked)

-초기화: allocproc(void) 안에서 각 변수를 초기값으로 설정하고 L0 큐에 프로세스가 등록됩니다.

```
p->lev = 0;
p->tq = 0;
p->priority = 3;
p->locked = 0;
enqueue(&mlfq.l0, p); // push process to (L0) queue
```

L0, L1, L2는 각각 한개의 원형 큐로 구성하고, proc.c 안에 세 개의 큐를 묶어 전역변수 mlfq로 선언하였습니다. 기존 xv6에서 최대 프로세스 값을 64개로 제한하기 때문에, L0, L1, L2도 각각 최대 64개의 프로세스를 저장할 수 있는 배열을 사용합니다.

```
struct {
  struct Queue l0;
  struct Queue l1;
  struct Queue l2;
} mlfq;
```

1. proc.c / scheduler(void)

```
361     // MLFQ
362     /* schedule process in L0 */
363     runnable_flag = 0;
364     size = mlfq.l0.count;
365     while(size--){
366       p = top(&mlfq.l0);
367       if(p->lev == 0 && p->state == RUNNABLE){ // RUNNABLE한 프로세스인 경우 switch
368         runnable_flag = 1;
369         c->proc = p;
370         switchuvm(p);
371         p->state = RUNNING;
372
373         swtch(&(c->scheduler), p->context);
374         switchkvm();
375         c->proc = 0;
376         if(!ticks) // priority boosting이 발생한 경우
377           break;
378
379         if(dequeue(&mlfq.l0) != p)
380           panic("Unexpected modify in L0 Queue");
```

```
381
382         if(p->lev == 1){ // tq을 다 쓴 프로세스는 l1로 넘겨줌 (이때, state에 대해서는 고려하지 않고, l1에서 처리되도록 함)
383           enqueue(&mlfq.l1, p);
384           #ifdef DEBUG
385           cprintf("L0: Runned - %d\n", p->pid);
386           cprintf("0:");printQueue(&mlfq.l0);
387           cprintf("1:");printQueue(&mlfq.l1);
388           cprintf("2:");printQueue(&mlfq.l2);
389           #endif
390         }
391         else if(p->lev == 0)
392           enqueue(&mlfq.l0, p);
393       }
394       else{ // RUNNABLE한 프로세스가 아닌 경우 switch 하지 않음.
395         dequeue(&mlfq.l0);
396         if(p->lev == 0 && p->state != UNUSED && p->state != ZOMBIE){
397           enqueue(&mlfq.l0, p); // 맨 뒤로 보냄
398         }
399       }
400     }
```

가장 먼저 L0 큐를 순회하며 RUNNABLE 한 프로세스면 스케줄링, 그렇지 않은 경우 프로세스 state를 확인하여 큐 뒤쪽으로 보냅니다.

RUNNABLE한 프로세스를 발견하여 switching이 1회 이상 발생한 경우 runnable_flag가 1로 바뀌고,

```
401     if(runnable_flag){
402       release(&ptable.lock);
403       continue;
404     }
```

위 코드에 의해 다시 L0큐를 순회하게 됩니다.

L0 큐에서 RUNNABLE한 프로세스를 찾지 못해 runnable_flag가 0으로 남아 있는 경우, L1 큐에서 가장 앞에 있는 RUNNABLE한 프로세스를 찾아 실행합니다.

```
406     /* schedule process in L1 */
407     size = mlfq.l1.count;
408     runnable_flag = 0;
409     while(size--){ // 가장 앞에 있는 RUNNABLE한 프로세스 찾기
410       p = top(&mlfq.l1);
411       if(p->lev == 1 && p->state == RUNNABLE){
412         runnable_flag = 1;
413         c->proc = p;
414         switchuvm(p);
415         p->state = RUNNING;
416 
417         swtch(&(c->scheduler), p->context);
418         switchkvm();
419         c->proc = 0;
420 
421         if(!ticks) // priority boosting이 발생한 경우
422           break;
423 
424         if(dequeue(&mlfq.l1) != p)
425           panic("Unexpected modify in L1 Queue");
426 
427         if(p->lev == 2){ // tq을 다 쓴 프로세스는 dequeue 하고, l2로 넘겨줌
428           enqueue(&mlfq.l2, p);
429           #ifdef DEBUG
430           cprintf("L1: Runned - %d\n", p->pid);
431           cprintf("0:");printQueue(&mlfq.l0);
432           cprintf("1:");printQueue(&mlfq.l1);
433           cprintf("2:");printQueue(&mlfq.l2);
434           #endif
435         }
436         else if(p->lev == 1)
437           enqueue(&mlfq.l1, p);
438         break;
439       }
440       else{
441         dequeue(&mlfq.l1);
442         if(p->lev == 1 && p->state != UNUSED && p->state != ZOMBIE)
443           enqueue(&mlfq.l1, p); // RUNNABLE로 전환될 여지가 남아 있음, L1의 맨 뒤로 보냄
444       }
445     }
```

L1 룹에서 RUNNABLE한 프로세스가 발견되어 한 번 switching이 일어나고 나면, L0 큐가 변경되었을 수 있으므로 L1룹을 break하고 L0 큐를 확인하게 됩니다.

L1 큐에서 RUNNABLE한 프로세스가 없어, runnable_flag가 0으로 설정되어 있는 경우, L2 큐를 순회합니다.

```
446     /* schedule process in L2 */
447     // L1큐에 실행할 프로세스가 없는 경우
448     // L2큐를 순회하며 실행할 프로세스 고르기
449     // UNUSED 된 프로세스는 priority boosting에서 처리되므로, lev을 확인해서 switching 해 주어야 함
450     if(!runnable_flag){
451       p = NULL;
452       priority_min = 4; // max: 3 이므로 처음 초기화를 위해 4로 설정하고 시작함
453       // L2의 프로세스 중 우선순위가 높고 L2에 먼저 들어온 프로세스 찾기
454       size = mlfq.l2.count;
455       int i = (mlfq.l2.front + 1) % (NPROC + 1);
456       while(size--){
457         now = mlfq.l2.q[i];
458         if(now->state == RUNNABLE && now->priority < priority_min){
459           p = now;
460           priority_min = p->priority;
461         }
462         i = (i + 1) % (NPROC + 1);
463       }
```

L2 큐의 경우 priority에 따라 프로세스가 처리되므로, 큐를 순회하면서 실행 조건에 맞는 프로세스를 찾습니다.

```
465         if(p && p->lev == 2){ // RUNNABLE한 프로세스 중 가장 우선순위가 높은 프로세스를 찾았다면!
466           c->proc = p;
467           switchuvm(p);
468           p->state = RUNNING;
469
470           swtch(&(c->scheduler), p->context);
471           switchkvm();
472           c->proc = 0;
473
474           #ifdef DEBUG
475           if(p->tq == 0){
476             cprintf("L2: Runned - %d\n", p->pid);
477             cprintf("0:");printQueue(&mlfq.l0);
478             cprintf("1:");printQueue(&mlfq.l1);
479             cprintf("2:");printQueue(&mlfq.l2);
480           }
481           #endif
482
483         }
```

찾은 프로세스를 실행하고 L0, L1큐에 RUNNABLE한 프로세스가 있는지 다시 확인합니다.

2. trap.c / trap(struct trapframe *tf)

1tick 단위로 yield()를 호출하면서 time quantum 과 프로세스 level을 바꾸는 부분은 trap에 구현되어 있습니다.

```
57          if (myproc() && myproc()->state == RUNNING)
58            myproc()->tq++; // 현재 프로세스의 time quantum 1 증가
```

먼저 실행 중인 프로세스에 대해 tq값을 1tick마다 증가시킵니다.

```
123     // Force process to give up CPU on clock tick.
124     // If interrupts were on while locks held, would need to check nlock.
125     if(myproc() && myproc()->state == RUNNING &&
126        tf->trapno == T_IRQ0+IRQ_TIMER){
127         if(!(ticks % 100)){
128           if(myproc()->locked)
129             schedulerUnlock(STUDENTID); // just unlock no password issue
130           priorityBoosting();
131         }
132        if(myproc()->locked == 0 &&
133           myproc()->tq == myproc()->lev * 2 + 4){ // 큐가 tq을 모두 소비한 경우
134           if(myproc()->lev != 2)
135             myproc()->lev++;
136           else if(myproc()->lev == 2){
137             if(myproc()->priority > 0)
138               myproc()->priority--;
139           }
140           myproc()->tq = 0;
141         }
142         if(myproc()->locked == 0)
143           yield();
144     }
```

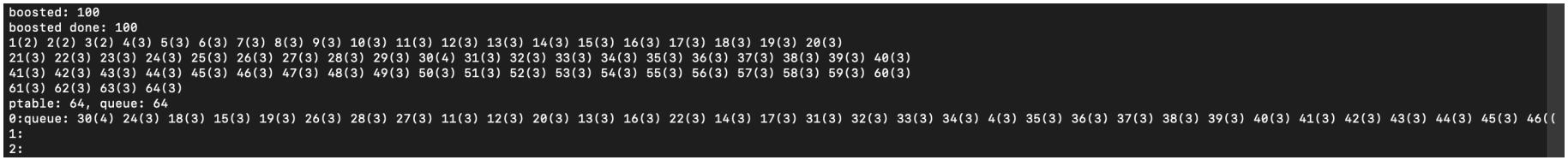
프로세스의 time quantum 값이 lev * 2 + 4와 같아지는 시점에 level을 조정 해 주거나 priority 값을 조정 해 주고, time quantum을 초기화한 후 yield()로 context switching 합니다. 이후 스케줄러는 스위칭으로 돌아온 프로세스에 대해 lev변경 여부를 확인하여 큐 위치를 조정합니다.

3. proc.c / priorityBoosting(void)

ticks (global tick) 변수가 100이 될 때마다 trap에서 priorityBoosting() 함수가 호출되어 priority boosting이 발생합니다.

```
670  void
671  priorityBoosting(void)
672  {
673    struct proc *p;
674    int size;
675
676    pushcli(); // disable timer interrupts
677    acquire(&ptable.lock);
678    size = mlfq.l0.count;
679    while(size--){
680      p = dequeue(&mlfq.l0);
681      if(p->state != UNUSED && p->state != ZOMBIE){
682        p->tq = 0;
683        p->lev = 0;
684        p->priority = 3;
685        enqueue(&mlfq.l0, p);
686      }
687    }
688    while(!isEmpty(&mlfq.l1)){
689      p = dequeue(&mlfq.l1);
690      if(p->lev == 1 && p->state != UNUSED && p->state != ZOMBIE){
691        p->tq = 0;
692        p->lev = 0;
693        p->priority = 3;
694        enqueue(&mlfq.l0, p);
695      }
696    }
697    while(!isEmpty(&mlfq.l2)){
698      p = dequeue(&mlfq.l2);
699      if(p->lev == 2 && p->state != UNUSED && p->state != ZOMBIE){
700        p->tq = 0;
701        p->lev = 0;
702        p->priority = 3;
703        enqueue(&mlfq.l0, p);
704      }
705    }
706    release(&ptable.lock);
707    ticks = 0;
708    popcli(); // enable timer interrupts
709  }
```

우선, 제 코드는 priority boosting 과정에서 timer interrupt가 발생할 경우 dequeue / enqueue 순서가 섞이면서 프로세스가 누락될 위험이 있습니다. 이를 확인하기 위해 아래 사진처럼 Priority boosting 시작과 끝에 tick을 출력하여 중간에 timer interrupt가 발생하는지 확인 해 보았습니다. 그 결과 프로세스 개수가 Max 일 때에도 Priority boosting 과정이 10ms 이내로 수행되어 문제가 없음을 확인했습니다. 다만 예방 차원에서 pushcli(), popcli()를 통해 timer interrupt를 잠시 disable 시켜 주었습니다.



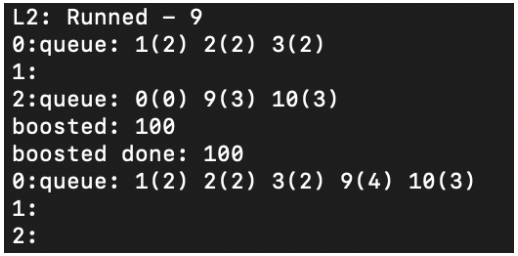
boosted: 100은 priority boosting 시작 시 tick 값

boosted done: 100은 priority boosting을 끝마쳤을 때 tick 값

아래 n(p) 형식의 숫자는 pid와 procstate 번호인데, queue의 누락을 확인하기 위해 ptable 요소를 먼저 출력한 후에 queue 값을 출력하였습니다.

ptable: 64, queue: 64는 현재 UNUSED, ZOMBIE 상태가 아닌 프로세스의 개수를 의미하고, 항상 같아야 합니다.

제 프로그램에서 Priority boosting은 명세에서 요구한 기능 이외에 프로세스를 정리하는 기능을 포함하고 있습니다. 후술할 lock, unlock 과정에서 레벨이 변경된 프로세스, L0~L2 큐에서 스케줄링 중에 종료되었지만 아직 큐에 남아 있는 프로세스들을 큐에서 제거 해 주기 위함입니다. 특히 L2 큐의 경우 스케줄러에서 dequeue 없이 큐를 순회하므로, priority boosting에서 정리되기 전까지 UNUSED상태로 큐에 남아 있습니다. defs.h 파일에서 DEBUG를 define 하여 디버깅 모드로 확인 해 보면 아래처럼 UNUSED 된 프로세스가 L2에 남아 있다가, priority boosting 후에 없어지는 것을 확인하실 수 있습니다.



4. proc.c / schedulerLock(int password), schedulerUnlock(int password)

스케줄러 lock, unlock은 프로세스의 locked 변수를 이용합니다. Lock을 잡은 프로세스는 최대 100 tick 동안 cpu를 독점하여 사용할 수 있는데, 저는 명세 해석 시에 스케줄러를 Lock(disable)하는 기능도 중요하다고 생각하였고, lock이 걸린 경우 해당 프로세스가 정상적으로 실행되는 동안 trap에서 yield를 막는 방법으로 구현하였습니다.

```
745  void
746  schedulerLock(int password)
747  {
748    if(password == STUDENTID){
749      if(myproc()->locked)
750        return;
751      myproc()->locked = 1;
752    #ifdef DEBUG
753      cprintf("[%d] lock!\n", myproc()->pid);
754      cprintf("0:");printQueue(&mlfq.l0);
755      cprintf("1:");printQueue(&mlfq.l1);
756      cprintf("2:");printQueue(&mlfq.l2);
757    #endif
758      acquire(&tickslock);
759      ticks = 0;
760      release(&tickslock);
761    }
762    else{
763      cprintf("SchedulerLock: invalid password, pid: %d, time quantum: %d, level: %d\n", myproc()->pid, myproc()->tq, myproc()->lev);
764      exit();
765    }
766  }
```

인자로 넘어 온 password가 학번과 일치하면서, lock이 이미 걸려 있지 않은 경우에 myproc()의 locked 변수를 1로 설정합니다. global tick은 0으로 초기화합니다.

```
123   // Force process to give up CPU on clock tick.
124   // If interrupts were on while locks held, would need to check nlock.
125   if(myproc() && myproc()->state == RUNNING &&
126      tf->trapno == T_IRQ0+IRQ_TIMER){
127       if(!(ticks % 100)){
128         if(myproc()->locked)
129           schedulerUnlock(STUDENTID); // just unlock no password issue
130         priorityBoosting();
131       }
132      if(myproc()->locked == 0 &&
133         myproc()->tq == myproc()->lev * 2 + 4){ // 큐가 tq을 모두 소비한 경우
134         if(myproc()->lev != 2)
135           myproc()->lev++;
136         else if(myproc()->lev == 2){
137           if(myproc()->priority > 0)
138             myproc()->priority--;
139         }
140         myproc()->tq = 0;
141       }
142       if(myproc()->locked == 0)
143         yield();
144   }
```

timer interrupt가 발생하여 trap함수가 실행되고, locked 변수가 1로 설정되었으므로 yield 없이 tick을 소비합니다. Ticks가 100이 되면 Unlock이 정확한 비밀번호로 실행되고, priority boosting이 발생합니다.

```
768  void
769  schedulerUnlock(int password)
770  {
771    if(password == STUDENTID){
772      if(!myproc()->locked) // 이미 unlock 되어 있는 경우 skip
773        return;
774      myproc()->lev = 0;
775      myproc()->priority = 3;
776      myproc()->tq = 0;
777      if(top(&mlfq.l0) != myproc()) // l0에서 lock된 경우가 아닐 때만 l0에 넣어줌. unlock되고 나면 스케줄러에서 lev이 다르므로 skip하게 됨
778        push(&mlfq.l0, myproc());
779    #ifdef DEBUG
780      cprintf("[%d] unlock!\n", myproc()->pid);
781      printQueue(&mlfq.l0);
782      printQueue(&mlfq.l1);
783      printQueue(&mlfq.l2);
784    #endif
785      myproc()->locked = 0;
786    }
787    else{
788      cprintf("SchedulerUnlock: invalid password, pid: %d, time quantum: %d, level: %d\n", myproc()->pid, myproc()->tq, myproc()->lev);
789      myproc()->tq = 0;
790      myproc()->locked = 0;
791      exit();
792    }
793  }
```

Unlock 함수는 우선 입력된 비밀번호와 locked 여부를 확인한 후, 프로세스의 레벨과 priority, tq 값을 초기화합니다. 이후 이 프로세스가 L0에서 Lock을 잡은 프로세스가 아닌 경우에만 L0의 맨 앞에 push 해 줍니다. 이렇게 되면 Unlock된 프로세스가 L0 큐 맨 앞에 위치하게 됩니다.

Lock을 잡은 프로세스의 위치는 L0 맨 앞, L1 맨 앞, L2 세 가지 경우로 나눌 수 있습니다. 세 가지 경우 중 L0의 맨 앞이었던 경우 Unlock 이후에 자연스럽게 원래 스케줄링으로 되돌아갑니다. L1 맨 앞이었던 경우에, 스케줄러는 L0에서 해당 프로세스를 실행 한 후 L1에 내려왔을 때 lev 이 맞지 않는 이 프로세스를 버려서 정리합니다. L2 큐에서 lock이 걸린 경우에도 마찬가지로 RUNNING으로 넘기지 않다가, priority boosting 과정에서 프로세스의 레벨을 확인하고 정리합니다.

(unlock 된 프로세스는 context switching 이후 바로 L0 프로세스를 확인하므로, L1, L2에 남아 있다고 해서 해당 프로세스가 Unlock 직후 L1, L2에서 실행되는 것은 아닙니다.)

*예외처리

Lock을 잡은 프로세스가 다시 Lock을 호출한 경우, Unlock 되어 있는 상태에서 다시 Unlock 함수가 호출된 경우 – warning이나 error 없이 return 합니다.
Wrong password 에 대한 체크: 프로세스 Lock, unlock 여부보다 우선하여 체크합니다. 따라서 정확한 비밀번호로 lock을 호출한 프로세스가 실행 중에 틀린 비밀번호로 다시 lock을 호출한 경우 또는 정확한 비밀번호로 unlock 된 프로세스가 틀린 비밀번호로 다시 unlock을 호출한 경우 모두 비밀번호 오류에 대한 에러 메시지를 띄우고 프로세스를 종료(exit)합니다.

🌈 Result (Test)

1. 컴파일 및 실행 과정

- make clean;make;make fs.img로 컴파일합니다.
- Makefile의 PASSWORD 변수에 interrupt schedulerLock, schedulerUnlock에 대한 비밀번호를 입력할 수 있습니다.
- 입력하지 않는 경우(컴파일 옵션에서 -DPASSWORD 주석 처리)에는 학번으로 자동 define 되어 정상 동작하게 구현 해 두었습니다.

- qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512 로 xv6를 실행합니다.

- 테스트 코드를 실행하여 동작을 확인합니다.
- defs.h 에 #define DEBUG를 주석 해제하면 L0, L1, L2 큐 변화 과정을 확인하실 수 있습니다.

2. 실행 결과 (테스트 결과)

unit test, scenario test, mlfq_test (제공 해 주신 테스트)
세 가지 테스트를 진행하였습니다.
(1) Unit test – test_mlfq_unit
구현해야 하는 기능이 동작하는지 간단히 함수를 호출 해 보며 확인하였습니다.

(2) Scenario test – test_mlfq_scenario
자식 프로세스를 여러 개 생성하며 가능한 시나리오를 구성한 후 의도대로 동작하는지 확인하였습니다.
Test1 – 자식 프로세스 3개로 기본적인 동작 과정을 확인합니다.

Test2 – scheduler lock, unlock을 각 케이스별로 확인합니다.

1) Lock 후에 unlock을 정상적으로 실행한 경우
2) Lock wrong password
3) Unlock wrong password
4) Priority boosting에 의해 Unlock 된 후 unlock wrong password – exit하도록 처리
5) 프로세스가 실행 중간에 Lock을 잡은 경우 (L1, L2 큐에서 lock 되었을 때 정상 동작하는지 보기 위함)
6) priority boosting에 의해 Unlock 된 후 correct password

Test3 – scheduler lock by interrupts, 컴파일 옵션으로 PASSWORD를 바꿔 보며 테스트

Test4 – Max process, 프로세스 개수가 64개일 때 정상 동작하는지 테스트 - 프로세스가 많고, 프로세스 개수가 약 25개를 넘어가면 L1큐에 어떤 프로세스도 내려가지 못하고 priority boosting이 반복되기 때문에 테스트 시간이 오래 걸립니다. (1분 정도 소요)

(3) mlfq_test

4개 프로세스에 대해 100000번의 룹 동안 L0, L1, L2 큐에 머문 횟수를 세고 wrong level이 없는지 체크합니다.

🤯 Trouble shooting

물리적인 L0, L1, L2 큐를 만들어서 실제로 스케줄링이 진행 될 때마다 큐에서 프로세스를 넣고 빼고 반복하도록 구현했기 때문에, 프로세스의 누락을 관리하는 것이 가장 어려웠습니다. 특히 Priority boosting을 구현하면서 종료되지 않은 프로세스가 큐를 이탈하는 문제를 자주 맞닥뜨렸고, ptable의 프로세스 개수와 비교하면서 디버깅하였습니다. 가장 오래 시간을 쏟은 누락 문제는 trap 함수에서 global tick 체크 순서 때문에 발생한 것이었습니다.

```
123     // Force process to give up CPU on clock tick.
124     // If interrupts were on while locks held, would need to check nlock.
125     if(myproc() && myproc()->state == RUNNING &&
126        tf->trapno == T_IRQ0+IRQ_TIMER){
127        if(!(ticks % 100)){
128          if(myproc()->locked)
129            schedulerUnlock(STUDENTID); // just unlock no password issue
130          priorityBoosting();
131        }
132        if(myproc()->locked == 0 &&
133          myproc()->tq == myproc()->lev * 2 + 4){ // 큐가 tq을 모두 소비한 경우
134          if(myproc()->lev != 2)
135            myproc()->lev++;
136          else if(myproc()->lev == 2){
137            if(myproc()->priority > 0)
138              myproc()->priority--;
139          }
140          myproc()->tq = 0;
141        }
142        if(myproc()->locked == 0)
143          yield();
144     }
```

처음에 trap 함수를 구현하면서 tq을 확인하고 레벨, priority 값을 바꿔 주는 132 ~ 141번 라인을 priority boosting 구간인 127 ~ 131 번 라인 위에 두었고, 이 경우 RUNNING 상태의 프로세스 레벨이 실제 속한 큐의 레벨과 맞지 않아 priority boosting 과정에서 누락되었습니다.

```
690     while(!isEmpty(&mlfq.l1)){
691       p = dequeue(&mlfq.l1);
692       if(p->lev == 1 && p->state != UNUSED && p->state != ZOMBIE){
```

692 라인에서 표시 된 대로 lev을 확인하여 dequeue된 프로세스를 L0큐로 enqueue 할지 결정하는데, ticks % 100 == 0 이고 myproc()->tq == myproc()->lev * 2 + 4인 특정 경우에 해당 프로세스가 누락됩니다.

Memo

최대 프로세스 개수로 테스트를 진행하면서 프로세스 개수가 많고 모두 RUNNABLE한 상태로 cpu를 기다릴 경우, L1, L2에 도달하지 못하고 Priority

boosting에 의해 L0에서 계속 스케줄링 될 수 있음을 깨달았습니다. L1, L2 큐의 크기를 64로 할당 해 두었지만, 두 개의 큐가 full이 되는 경우는 발생하지 않습니다. 하지만 라이브러리로 구현된 큐의 크기를 동적으로 변화시키는 것이 프로그램을 더 복잡하게 만든다고 생각하여 큐 크기를 수정하지 않았습니다.