XV6 MLFQ Scheduler implementation

mlfq는 기본적으로 xv6 기존의 배열 + RUNNABLE한 프로세스만 모아 배열에 참조 가능한 포인터의 큐로 구현했다. 배열에 모든 layer와 priority값을 섞어둘 경우 큐의 순서를 구현하기 힘들고, L2에서 찾는 경우 많은 순회를 해야 한다. 배열을 없앤다면 memory allocation으로 구현해야 하므로 구현 난이도가 높아지고, queue에서 pop할때 4바이트 리턴값이 아닌 PCB 전체의 데이터를 리턴해야 하므로 그만큼 느려지게 된다. 변수의 의미들은 다음과 같다.

1. 여러 프로세스가 동시에 접근하여 race condition이 일어나지 않게 하는 lock
2. mlfq의 크기
3. schedulerLock 함수가 실행될 경우 이 변수에 값이 들어가 mlfq보다 먼저 꺼내진다.
4. L0와 L1는 queue 방식으로 동작하므로 circular queue로 구현하였다.
5. L2는 priority와 time(L1→L2로 이동한 시간)을 이용하는 priority queue로 구현하였다.
6. proc 배열

```
10  // mlfq & related function definition part
11  struct circular_queue {
12      struct proc* proc[NPROC];
13      int head;
14      int tail;
15      int size;
16  };
17
18  struct priority_queue {
19      struct proc* proc[NPROC];
20      int size;
21      int time;
22  };
23
24  struct mlfq {
25      struct spinlock lock;
26      int size;
27      struct proc* schedulerLocked;
28
29      struct circular_queue L0;
30      struct circular_queue L1;
31      struct priority_queue L2;
32
33      struct proc proc[NPROC];
34  };
35
```

queue와 관련된 함수들은 다음과 같다. schedulerLock 이후 time quantum이 100을 넘었을 때 프로세스가 큐의 앞으로 가는것을 구현하기 위해 push_front를 추가했다. priority queue의 정렬기준은 priority가 작은 순서 → time이 작은 순서로 정렬하도록 했다.

```
36  // circular queue function
37  inline int circular_isfull(struct circular_queue *cq){
38      if (cq->size == NPROC) return 1;
39      return 0;
40  }
41  inline int circular_isempty(struct circular_queue *cq){
42      if (cq->size == 0) return 1;
43      return 0;
44  }
45
46  int circular_enqueue(struct circular_queue *cq, struct proc *p){
47      if (circular_isfull(cq)) return -1;
48
49      cq->proc[cq->tail] = p;
50      cq->tail = (cq->tail + 1) % NPROC;
51      cq->size++;
52      return 0;
53  }
54
55  struct proc* circular_dequeue(struct circular_queue *cq){
56      struct proc *ret;
57      if (circular_isempty(cq)) return 0;
58
59      ret = cq->proc[cq->head];
60      cq->head = (cq->head + 1) % NPROC;
61      cq->size--;
62      return ret;
63  }
64
```

```
65  int circular_pushfront(struct circular_queue *cq, struct proc *p){
66      if (circular_isfull(cq)) return -1;
67
68      cq->head = (cq->head - 1) % NPROC;
69      cq->proc[cq->head] = p;
70      cq->size++;
71      return 0;
72  }
73
74  // priority queue function
75  inline int comp(struct proc *p, struct proc *q){
76      if (p->priority < q->priority) return 1;
77      if (p->time < q->time) return 1;
78      return 0;
79  }
80
81  inline int priority_isempty(struct priority_queue *pq){
82      if (pq->size == 0) return 1;
83      return 0;
84  }
85
86  int priority_insert(struct priority_queue *pq, struct proc *p){
87      int i = pq->size;
88      struct proc *temp;
89      if (pq->size == NPROC) return -1;
90
91      pq->proc[i] = p;
92      while (i > 0 && comp(pq->proc[i], pq->proc[(i - 1) / 2])) {
93          temp = pq->proc[i];
94          pq->proc[i] = pq->proc[(i - 1) / 2];
95          pq->proc[(i - 1) / 2] = temp;
96          i = (i - 1) / 2;
97      }
98      pq->size++;
99      return 0;
100 }
```

```
101
102 struct proc* priority_pop(struct priority_queue *pq){
103     struct proc *ret, *temp;
104     int i, j;
105     if (pq->size == 0) return 0;
106
107     ret = pq->proc[0];
108     pq->size--;
109     pq->proc[0] = pq->proc[pq->size];
110     i = 0;
111     while (1) {
112         if (2 * i + 1 <= pq->size - 1 && comp(pq->proc[2 * i + 1], pq->proc[i])) j = 2 * i + 1;
113         else if (2 * i + 2 <= pq->size - 1 && comp(pq->proc[2 * i + 2], pq->proc[i])) j = 2 * i + 2;
114         else break;
115
116         temp = pq->proc[j];
117         pq->proc[j] = pq->proc[i];
118         pq->proc[i] = temp;
119         i = j;
120     }
121     return ret;
122 }
123
```

PCB에는 다음 4가지 변수를 추가했다.

layer: 해당 프로세스가 몇번째 queue에 있는지 확인

timequantum: 해당 프로세스가 다음 queue로 가기까지 몇 tick 남았는지 확인

priority: 해당 프로세스의 priority 확인

time: 해당 프로세스가 L1→L2로 이동한 시간 확인

```
37  // Per-process state
38  struct proc {
39    uint sz;                     // Size of process memory (bytes)
40    pde_t* pgdir;                // Page table
41    char *kstack;                // Bottom of kernel stack for this process
42    enum procstate state;        // Process state
43    int pid;                     // Process ID
44    struct proc *parent;         // Parent process
45    struct trapframe *tf;        // Trap frame for current syscall
46    struct context *context;     // swtch() here to run process
47    void *chan;                  // If non-zero, sleeping on chan
48    int killed;                  // If non-zero, have been killed
49    struct file *ofile[NOFILE];  // Open files
50    struct inode *cwd;           // Current directory
51    char name[16];               // Process name (debugging)
52
53    // Added for assignment
54    int layer;                  // Existing layer
55    int timequantum;            // Remaining time to move next layer
56    int priority;               // Needed for priority scheduling
57    int time;                   // Time when process get into L2 layer (need to FCFS)
58
59  };
```

새롭게 추가된 변수는 globaltick, time, unlocked이 있다.

globaltick은 tick계산을 위해, time은 L1→L2 시간을 체크하기 위해 사용한다.

```
124  // proc.c starts
125  struct mlfq ptable;
126
```

```
169   static struct proc *initproc;
170
171   int nextpid = 1;
172   int globaltick = 0;
173   int time = 0;
174   int unlocked = 0;
175   extern void forkret(void);
176   extern void trapret(void);
177
178   static void wakeup1(void *chan);
```

allocproc 함수는 다른 부분은 같지만 프로세스가 할당될 때 추가해준 변수에 layer = 0, timequantum = 4 priority = 3으로 초기값을 정해주었다.

```
230  allocproc(void)
231  {
232    struct proc *p;
233    char *sp;
234
235    acquire(&ptable.lock);
236
237    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
238      if(p->state == UNUSED)
239        goto found;
240
241    release(&ptable.lock);
242    return 0;
243
```

```
244  found:
245    p->state = EMBRYO;
246    p->pid = nextpid++;
247
248    ptable.size++;
249    p->layer = 0;
250    p->timequantum = 4;
251    p->priority = 3;
252
253    release(&ptable.lock);
254
255    // Allocate kernel stack.
256    if((p->kstack = kalloc()) == 0){
257      p->state = UNUSED;
258      ptable.size--;
259      return 0;
260    }
261    sp = p->kstack + KSTACKSIZE;
262
263    // Leave room for trap frame.
264    sp -= sizeof *p->tf;
265    p->tf = (struct trapframe*)sp;
266
267    // Set up new context to start executing at forkret,
268    // which returns to trapret.
269    sp -= 4;
270    *(uint*)sp = (uint)trapret;
271
272    sp -= sizeof *p->context;
273    p->context = (struct context*)sp;
274    memset(p->context, 0, sizeof *p->context);
275    p->context->eip = (uint)forkret;
276
277    return p;
278  }
```

userinit 함수에는 RUNNABLE로 바뀔 때 L0큐로 들어가도록 하였다.

```
282  void
283  userinit(void)
284  {
285    struct proc *p;
286    extern char _binary_initcode_start[], _binary_initcode_size[];
287
288    p = allocproc();
289
290    initproc = p;
291    if((p->pgdir = setupkvm()) == 0)
292      panic("userinit: out of memory?");
293    inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
294    p->sz = PGSIZE;
295    memset(p->tf, 0, sizeof(*p->tf));
296    p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
297    p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
298    p->tf->es = p->tf->ds;
299    p->tf->ss = p->tf->ds;
300    p->tf->eflags = FL_IF;
301    p->tf->esp = PGSIZE;
302    p->tf->eip = 0;  // beginning of initcode.S
303
304    safestrcpy(p->name, "initcode", sizeof(p->name));
305    p->cwd = namei("/");
306
307    // this assignment to p->state lets other cores
308    // run this process. the acquire forces the above
309    // writes to be visible, and the lock is also needed
310    // because the assignment might not be atomic.
311    acquire(&ptable.lock);
312
313    p->state = RUNNABLE;
314
```

```
315    // insert process pointer into L0 queue
316    if (circular_enqueue(&ptable.L0, p) != 0) panic("ERROR1");
317    release(&ptable.lock);
318  }
```

fork 함수도 마찬가지로 프로세스 공간을 alloc받은 후 L0큐로 들어가게 했다.

```
344  int
345  fork(void)
346  {
347    int i, pid;
348    struct proc *np;
349    struct proc *curproc = myproc();
350
351    // Allocate process.
352    if((np = allocproc()) == 0){
353      return -1;
354    }
355
356    // Copy process state from proc.
357    if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
358      kfree(np->kstack);
359      np->kstack = 0;
360      np->state = UNUSED;
361      ptable.size--;
362      return -1;
363    }
364    np->sz = curproc->sz;
365    np->parent = curproc;
366    *np->tf = *curproc->tf;
367
368    // Clear %eax so that fork returns 0 in the child.
369    np->tf->eax = 0;
370
371    for(i = 0; i < NOFILE; i++)
372      if(curproc->ofile[i])
373        np->ofile[i] = filedup(curproc->ofile[i]);
374    np->cwd = idup(curproc->cwd);
375
376    safestrcpy(np->name, curproc->name, sizeof(curproc->name));
377
378    pid = np->pid;
```

```
379
380    acquire(&ptable.lock);
381
382    np->state = RUNNABLE;
383
384    // insert process pointer into L0 queue
385    if (circular_enqueue(&ptable.L0, np) != 0) panic("ERROR2");
386
387    release(&ptable.lock);
388
389    return pid;
390  }
```

exit 함수에서는 scheduler lock시킨 프로세스가 exit할 경우 자동으로 unlock되도록 했다. (밑의 schedulerLock 참고)

```
395  void
396  exit(void)
397  {
398    struct proc *curproc = myproc();
399    struct proc *p;
400    int fd;
401
402    if(curproc == initproc)
403      panic("init exiting");
404
405    // Close all open files.
406    for(fd = 0; fd < NOFILE; fd++){
407      if(curproc->ofile[fd]){
408        fileclose(curproc->ofile[fd]);
409        curproc->ofile[fd] = 0;
410      }
411    }
412
413    begin_op();
414    iput(curproc->cwd);
415    end_op();
416    curproc->cwd = 0;
417
418    acquire(&ptable.lock);
419
420    if (ptable.schedulerLocked != 0 && ptable.schedulerLocked->pid == curproc->pid){
421      cprintf("Auto unlock the scheduler because the process which locked it is about to exit.\n");
422      ptable.schedulerLocked = 0;
423    }
424    // Parent might be sleeping in wait().
425    wakeup1(curproc->parent);
```

```
426
427    // Pass abandoned children to init.
428    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
429      if(p->parent == curproc){
430        p->parent = initproc;
431        if(p->state == ZOMBIE)
432          wakeup1(initproc);
433      }
434    }
435    // Jump into the scheduler, never to return.
436    curproc->state = ZOMBIE;
437    sched();
438    panic("zombie exit");
439  }
```

위의 global tick 값이 100이 될때마다 priorityBoosting 함수가 실행된다.

L1큐를 하나하나 pop한 다음 L0에 집어넣고 L2큐를 다시 하나하나 pop한 다음 L0에 집어넣는다. 그 후 프로세스 배열을 돌며 값들을 초기화해준다.

```
// Priority Boosting
void priorityBoosting(){
  struct proc *temp;
  int i;

  while (!circular_isempty(&ptable.L1)){
    temp = circular_dequeue(&ptable.L1);
    if (circular_enqueue(&ptable.L0, temp) != 0) panic("ERROR11");
  }
  while (!priority_isempty(&ptable.L2)){
    temp = priority_pop(&ptable.L2);
    if (circular_enqueue(&ptable.L0, temp) != 0) panic("ERROR12");
  }
  for (i = 0; i < NPROC; i++){
    if (ptable.proc[i].state != UNUSED){
      ptable.proc[i].layer = 0;
      ptable.proc[i].priority = 3;
      ptable.proc[i].timequantum = 4;
    }
  }

}
```

scheduler 함수이다.

우선 scheduler가 Lock되어 있는지 확인한다. scheduler가 비어있거나 RUNNABLE하지 않으면(예외처리) mlfq를 한 층씩 순회한다. 없으면 위로 올라가 다시 순회한다.

timer interrupt가 발생하거나 running중인 프로세스가 sleep이나 exit을 호출하여 context switching이 일어나야 할 때, sched 함수가 실행되고 sched 함수는 현재 running중인 프로세스와 scheduler 함를 context switch 해준다. 따라서 프로세스가 바뀔때마다 scheduler 함수는 550줄 이후부터 실행되는데, 이 곳에 globaltick++를 해주어 tick을 계산할 수 있게 했다. tick이 100이 되면 priorityBoosting 함수가 실행되고 만약 scheduler가 잠겨있었다면 scheduler를 풀어준다.

```
517  void
518  scheduler(void)
519  {
520    struct proc *p;
521    struct cpu *c = mycpu();
522    c->proc = 0;
523
524    for(;;){
525      // Enable interrupts on this processor.
526      sti();
527
528      // Loop over process table looking for process to run.
529      acquire(&ptable.lock);
530
531      if (ptable.schedulerLocked != 0 && ptable.schedulerLocked->state == RUNNABLE)
532        p = ptable.schedulerLocked;
533      else if (!circular_isempty(&ptable.L0))
534        p = circular_dequeue(&ptable.L0);
535      else if (!circular_isempty(&ptable.L1))
536        p = circular_dequeue(&ptable.L1);
537      else if (!priority_isempty(&ptable.L2))
538        p = priority_pop(&ptable.L2);
539      else {
540        release(&ptable.lock);
541        continue;
542      }
543      // Switch to chosen process.  It is the process's job
544      // to release ptable.lock and then reacquire it
545      // before jumping back to us.
546      c->proc = p;
547      switchuvm(p);
548      p->state = RUNNING;
549
```

```
550      swtch(&(c->scheduler), p->context);
551      switchkvm();
552
553
554      // Process is done running for now.
555      // It should have changed its p->state before coming back.
556      c->proc = 0;
557
558      // count up global tick. if 100, do priority boosting and unlock the scheduler if locked.
559      globaltick++;
560      if (globaltick == 100){
561        globaltick = 0;
562        priorityBoosting();
563
564        if (ptable.schedulerLocked != 0){
565          cprintf("Auto unlock the scheduler because 100 tick passed after it has been locked.\n");
566          circular_pushfront(&ptable.L0, ptable.schedulerLocked);
567          ptable.schedulerLocked = 0;
568        }
569      }
570
571      release(&ptable.lock);
572    }
573
574  }
```

yield 시스템 콜을 호출하거나 timer interrupt가 발생하면 yield로 넘어오므로 yield 함수에서는 time quantum을 관리해 준다. time quantum에 따라 layer와 priority도 바뀌도록 설정했다. 그리고 sched 함수 실행 전에 RUNNABLE로 바뀌므로 다시 큐에 넣어주었다.

```
603  void
604  yield(void)
605  {
606    struct proc *p = myproc();
607
608    acquire(&ptable.lock);  //DOC: yieldlock
609
610    p->state = RUNNABLE;
611
612    if (ptable.schedulerLocked == 0){
613      // reduce timequantum by 1. if 0, move to next layer
614      if (p->timequantum == 0){
615        switch (p->layer){
616          case 0:
617            p->layer = 1;
618            p->timequantum = 6;
619            break;
620          case 1:
621            p->layer = 2;
622            p->timequantum = 8;
623            p->time = time++;
624            break;
625          case 2:
626            p->timequantum = 8;
627            p->priority = p->priority != 0 ? p->priority - 1 : 0;
628          }
629      } else p->timequantum--;
630
```

```
632      // put process to ready queue(runnable process)
633      if (unlocked) {
634        proctest();
635        circular_pushfront(&ptable.L0, p);
636        unlocked = 0;
637        proctest();
638      } else{
639        switch (p->layer){
640        case 0:
641          if (circular_enqueue(&ptable.L0, p) != 0) panic("ERROR3");
642          break;
643        case 1:
644          if (circular_enqueue(&ptable.L1, p) != 0) panic("ERROR4");
645          break;
646        case 2:
647          if (priority_insert(&ptable.L2, p) != 0) panic("ERROR5");
648        }
649      }
650    }
651
652    sched();
653    release(&ptable.lock);
654  }
```

sleep 함수에서 lock을 걸어버리고 자면 안되므로 자동으로 unlock을 해주었다. 또 yield가 실행되지 않고 sleep→sched로 넘어갈 수도 있으므로 이 때에도 timequantum을 관리해 준다.

```
672  void
673  sleep(void *chan, struct spinlock *lk)
674  {
675    struct proc *p = myproc();
676
677    if(p == 0)
678      panic("sleep");
679
680    if(lk == 0)
681      panic("sleep without lk");
682
683    // Must acquire ptable.lock in order to
684    // change p->state and then call sched.
685    // Once we hold ptable.lock, we can be
686    // guaranteed that we won't miss any wakeup
687    // (wakeup runs with ptable.lock locked),
688    // so it's okay to release lk.
689    if(lk != &ptable.lock){  //DOC: sleeplock0
690      acquire(&ptable.lock);  //DOC: sleeplock1
691      release(lk);
692    }
693    // Go to sleep.
694    p->chan = chan;
695    p->state = SLEEPING;
696
697    if (p->timequantum == 0){
698      switch (p->layer){
699        case 0:
700          p->layer = 1;
701          p->timequantum = 6;
702          break;
703        case 1:
704          p->layer = 2;
705          p->timequantum = 8;
706          p->time = time++;
707          break;
```

```
708         case 2:
709           p->timequantum = 8;
710           p->priority = p->priority != 0 ? p->priority - 1 : 0;
711         }
712       } else p->timequantum--;
713
714       if (ptable.schedulerLocked != 0 && ptable.schedulerLocked->pid == p->pid){
715         cprintf("Auto unlock the scheduler because the process which locked it is about to sleep.\n");
716         ptable.schedulerLocked = 0;
717       }
718
719       sched();
720
721       // Tidy up.
722       p->chan = 0;
723
724       // Reacquire original lock.
725       if(lk != &ptable.lock){  //DOC: sleeplock2
726         release(&ptable.lock);
727         acquire(lk);
728       }
729     }
```

wakeup 함수는 프로세스가 SLEEP→RUNNABLE로 변하므로 다시 QUEUE에 넣어주었다.

```
717   static void
718   wakeup1(void *chan)
719   {
720     struct proc *p;
721
722     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
723       if(p->state == SLEEPING && p->chan == chan){
724         p->state = RUNNABLE;
725         switch (p->layer){
726           case 0:
727             if (circular_enqueue(&ptable.L0, p) != 0) panic("ERROR6");
728             break;
729           case 1:
730             if (circular_enqueue(&ptable.L1, p) != 0) panic("ERROR7");
731             break;
732           case 2:
733             if (priority_insert(&ptable.L2, p) != 0) panic("ERROR8");
734         }
735       }
736
737   }
```

kill 함수 또한 SLEEPING→RUNNABLE일 경우 QUEUE에 넣어주었고 kill될 때까지 scheduler를 잡고 있으면 안되므로 자동으로 unlock되도록 했다.

```
751  int
752  kill(int pid)
753  {
754    struct proc *p;
755
756    acquire(&ptable.lock);
757    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
758      if(p->pid == pid){
759        p->killed = 1;
760        // Wake process from sleep if necessary.
761        if(p->state == SLEEPING){
762          p->state = RUNNABLE;
763          switch (p->layer){
764            case 0:
765              if (circular_enqueue(&ptable.L0, p) != 0) panic("ERROR9");
766              break;
767            case 1:
768              if (circular_enqueue(&ptable.L1, p) != 0) panic("ERROR10");
769              break;
770            case 2:
771              if (priority_insert(&ptable.L2, p) != 0) panic("ERROR11");
772          }
773        }
774        if (ptable.schedulerLocked->pid == p->pid){
775          cprintf("Auto unlock the scheduler because the process which locked it is about to be killed.\n");
776          ptable.schedulerLocked = 0;
777        }
778        release(&ptable.lock);
779        return -1;
780      }
781    }
782    release(&ptable.lock);
783    return -1;
784  }
```

아래는 모두 syscall에서 사용하는 함수이다.

schedulerLocker 는 누가 ptable을 lock하고있는지 확인하는 함수이다.

getLayer 는 현재 프로세스의 layer를 반환하는 함수이다.

changePriority는 현재 프로세스의 priority를 바꾸는 함수이다.

```
823  int schedulerLocker(void){
824    int pid;
825
826    acquire(&ptable.lock);
827    pid = ptable.schedulerLocked != 0 ? ptable.schedulerLocked->pid : 0;
828    release(&ptable.lock);
829
830    return pid;
831  }
832
833  // used to syscall
834  int getLayer(void){
835    struct proc *p = myproc();
836    return p->layer;
837  }
838
839  int changePriority(int pid, int pri){
840    struct proc* p;
841    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
842      if (p->pid == pid){
843        p->priority = pri;
844        return 0;
845      }
846    }
847    return -1;
848  }
```

setSchedulerLock은 scheduler를 Lock/Unlock 하는 함수이다.

우선 인자로 받은 패스워드를 비교해 틀렸으면 명세의 정보들을 반환하고 프로세스를 종료한다.

그 다음은 이미 Lock/Unlock이 되었는지 확인한다. 이미 되어있으면 로그를 출력하고 함수를 종료한다.

그 후 ptable의 lock 변수에 접근해서 정보를 바꿔준다. lock을 걸 경우 globaltick, 해제할 경우 명세대로 데이터를 세팅해준다. 또 unlock될 경우 unlock 변수를 1로 설정하고 yield 함수를 호출해 yield에서 큐의 맨 앞으로 가도록 예외처리를 해두었다.

한 가지 예외처리는 프로세스가 sleep, exit, killed할 때 자신이 lock을 걸었다면 자동으로 unlock되도록 했는데 우선적으 처리해야 할 프로세스가 사라진 상황이므로 더이상 scheduler를 lock하고 있을 이유가 없다고 판단하였다.

```
875  void setSchedulerLock(int pwd, int is){
876    struct proc *p;
877    p = myproc();
878
879    if (pwd != 2019082079){
880      cprintf("Wrong password!!\npid: %d, time quantum: %d, level: %d proc has terminated\n", p->pid, p->timequantum, p->layer);
881      exit();
882      return;
883    }
884
885    acquire(&ptable.lock);
886
887    if (ptable.schedulerLocked != 0 && is == 1){
888      cprintf("The schelduler is already locked\n");
889      release(&ptable.lock);
890      return;
891    }
892
893    if (ptable.schedulerLocked == 0 && is == 0){
894      cprintf("The schelduler is already unlocked\n");
895      release(&ptable.lock);
896      return;
897    }
898
```

```
899    if (is == 1) {
900      ptable.schedulerLocked = p;
901      globaltick = 0;
902      release(&ptable.lock);
903    } else{
904      ptable.schedulerLocked = 0;
905      p->layer = 0;
906      p->priority = 3;
907      p->timequantum = 4;
908      unlocked = 1;
909      release(&ptable.lock);
910      yield();
911    }
912
913    return;
914  }
```

시스템 콜을 구현하였다.

yield 함수가 이미 존재하므로 yield1함수로 구현했고 proc.c의 yield 함수를 그대로 호출하도록 했다. 다만 scheduler가 locked되어있다면 context switching이 일어나지 않도록 했다.

이외의 함수들도 예외처리를 제외하면 proc.c의 함수를 호출해 원하는 동작을 수행하도록 했다.

```c
#include "types.h"
#include "defs.h"

void yield1(void){
    int pid = schedulerLocker();
    if (pid != 0) cprintf("pid: %d has locked the scheduler. context swithcing will not arise.\n", pid);
    else yield();
}

int getLevel(void){
    return getLayer();
}

void setPriority(int pid, int pri){
    if (pri > 3 || pri < 0) cprintf("priority must be 0~3.\n");
    else changePriority(pid, pri);
}

void schedulerLock(int password){
    setSchedulerLock(password, 1);
}

void schedulerUnlock(int password){
    setSchedulerLock(password, 0);
}

void procTest(void){
    proctest();
}

```

```c
int sys_getLevel(void){
    return getLevel();
}

int sys_setPriority(void){
    int pid;
    int pri;

    if (argint(0, &pid) != 0 || argint(1, &pri) != 0)
        return -1;
    cprintf("pid: %d pri: %d\n", pid, pri);
    setPriority(pid, pri);
    return 0;
}

int sys_schedulerLock(void){
    int password;
    if (argint(0, &password) != 0) return -1;
    schedulerLock(password);
    return 0;
}

int sys_schedulerUnlock(void){
    int password;
    if (argint(0, &password) != 0) return -1;
    schedulerUnlock(password);
    return 0;
}

```

trap.c에서는 DPL을 바꿔주고, T_SCHEDLOCK(129)와 T_SCHEDUNLOCK(130)을 인터럽트로 받을 경우 시스템 콜을 호출하도록 했다.

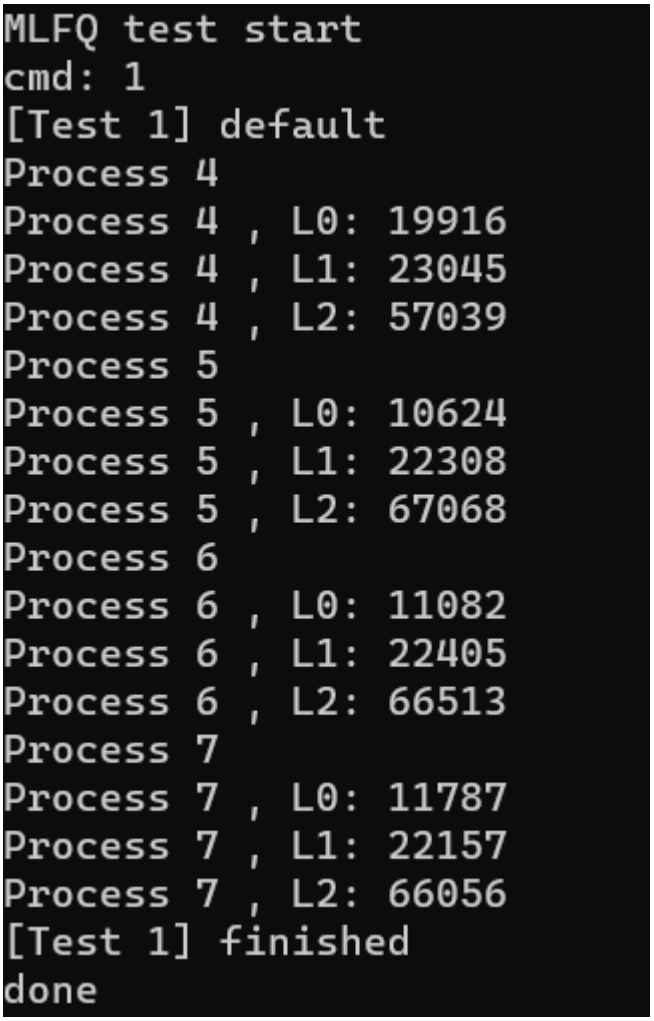
```
17  void
18  tvinit(void)
19  {
20    int i;
21
22    for(i = 0; i < 256; i++)
23      SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
24    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
25    SETGATE(idt[T_SCHEDLOCK], 1, SEG_KCODE<<3, vectors[T_SCHEDLOCK], DPL_USER);
26    SETGATE(idt[T_SCHEDUNLOCK], 1, SEG_KCODE<<3, vectors[T_SCHEDUNLOCK], DPL_USER);
27
28    initlock(&tickslock, "time");
29  }
```

```
38  void
39  trap(struct trapframe *tf)
40  {
41    if(tf->trapno == T_SYSCALL){
42      if(myproc()->killed)
43        exit();
44      myproc()->tf = tf;
45      syscall();
46      if(myproc()->killed)
47        exit();
48      return;
49    }
50
51    if(tf->trapno == T_SCHEDLOCK){
52      if(myproc()->killed)
53        exit();
54      myproc()->tf = tf;
55      setSchedulerLock(2019082079, 1);
56      if(myproc()->killed)
57        exit();
58      return;
59    }
60
61    if (tf->trapno == T_SCHEDUNLOCK){
62      if(myproc()->killed)
63        exit();
64      myproc()->tf = tf;
65      setSchedulerLock(2019082079, 0);
66      if(myproc()->killed)
67        exit();
68      return;
69    }
```

실행 코드와 결과는 다음과 같다.

```
#include "types.h"
#include "stat.h"
#include "user.h"

#define NUM_LOOP 100000
#define NUM_SLEEP 500

#define NUM_THREAD 4
#define MAX_LEVEL 3
int parent;

int fork_children()
{
  int i, p;
  for (i = 0; i < NUM_THREAD; i++) {
    p = fork();
    if (p == 0)
    {
      sleep(10);
      return getpid();
    }
    sleep(50);
  }
  return parent;
}

case '1':
    printf(1, "[Test 1] default\n");
    pid = fork_children();


    for (volatile int i=0; i<10000000; i++){
    }

    if (pid != parent)
```

```
    {
      for (i = 0; i < NUM_LOOP; i++)
      {
        int x = getLevel();
        if (x < 0 || x > 2)
        {
          printf(1, "Wrong level: %d\n", x);
          exit();
        }
        count[x]++;
      }
      printf(1, "Process %d\n", pid);
      for (i = 0; i < MAX_LEVEL; i++)
        printf(1, "Process %d , L%d: %d\n", pid, i, count[i])
    }
    exit_children();
    printf(1, "[Test 1] finished\n");
    break;
```


MLFQ test start
cmd: 1
[Test 1] default
Process 4
Process 4 , L0: 19916
Process 4 , L1: 23045
Process 4 , L2: 57039
Process 5
Process 5 , L0: 10624
Process 5 , L1: 22308
Process 5 , L2: 67068
Process 6
Process 6 , L0: 11082
Process 6 , L1: 22405
Process 6 , L2: 66513
Process 7
Process 7 , L0: 11787
Process 7 , L1: 22157
Process 7 , L2: 66056
[Test 1] finished
done