ELE3021_project01_12299_2019030991

> *본 위키는 commit 기준 Real Finish에 맞추어 작성되어 있습니다.*
> *12299_2019030991_홍정범*

Design

process 탐색 방식의 결정

우선 가장 처음 결정한 것은 **Queue를 직접 만들어서 데이터를 동적으로 관리할 것인가, 아니면 ptable을 for문을 순회하면서 스케줄링을 해줄 것인가** 이었다. 만약에 단지 ptable을 순회하면서 스케줄리을 할 수 있다면 굳이 큐를 만들어서 관리해주는 수고를 덜어도 될 것이라고 초기에 판단, 그래서 ptable을 순회하여 스케줄리을 하기로 마음먹었다.

proc 구조체 구성 방식의 결정

ptable을 순회하면서 어떤 proces를 찾을지 결정할 것이기 때문에 기본적을 각 프로세스는 자신이 어느 큐에 속해 있는지에 대한 정보를 가지고 있어야 한다. 그리고 타임인터럽트의 발생으로 인한 q_lv 및 priorioty의 변경이 일어나야 하므로 각 프로세스는 자신이 얼마나 일을 했는지에 대한 정보를 가지고 있어야 한다. 그러려면 당연히 priolrity 값도 가지고 있어야 한다. 그래서 3개의 변수를 추가해 주었다.

```
// Per-process state
struct proc {
  int priority;               // Scheduling prioriy
  int tq;                     // Time quantum
  int q_lv;                   // Queue level
..
}
```

그리고 처음 실행된 프로세스는 가장 높은 레벨의 큐에 들어가야 하므로 allocproc() 함수에서 초기화를 해주었다.

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->q_lv = 0;
  p->tq = 0;
  p->priority = 3;
```

process 결정 방식의 결정

처음에 발상한 방식은 이러했다. L0 큐의 runnable한 프로세스가 없을 경우 L1을 스케줄링 해야하고, L1큐의 runnable한 프로세스가 없을 경우 L2를 스케줄링 해야한다는 것을 그대로 적용했다.

⚙️ 1. ptable을 0~63 모두 순회하면서 q_lv = 0인 프로세스를 찾는다.
2. 가장 앞에있는 q_lv = 0인 프로세스를 실행하고 tq와 q_lv등을 변경해준다.
3. ptable에 q_lv = 0인 프로세스가 없다면 이제 q_lv=1인 프로세스를 찾는 방식으로 1번부터 반복한다.

물론, 당연히 이렇게 단순한 알고리즘으로 코드를 구성하고 스케줄러를 만드려고 하니 전혀 진척이 없었다. 우선 필요 이상으로 ptable을 많이 도는 경우가 발생했고, L1이나 L2 큐를 찾아보아야 하는 상황에 새로운 프로세스가 실행되어 L0에 프로세스가 들어온다던지 하는 예외를 처리하는 것이 만만치 않았으며, priority 계산을 하는 것도 쉽지 않았다. 그러다 문득 굳이 L0,1,2,3를 따로 찾아보아야할까..? 라는 생각이 들었다. ptable을 한 번 순회하는 것으로 지금 실행하고 싶은 프로세스를 찾을 수 있지 않을까?
즉, ptable을 한 바퀴 다 돌아 " 지금 가장 우선적으로 처리해줘야할 프로세스를 고르자 " 라는 방식으로 알고리즘을 구현하기로 했다.

⚙️ 1. running_p라고 하는 빈 proc 구조체를 선언해준다.
2. ptable을 순회하며 running_p를 가장 우선순위가 높은 process로 초기화 시킨다
→ 큐의 레벨과 priority를 비교하여 running_p를 찾는다. / ex) q_lv이 0인 process가 q_lv이 1인 process보다 먼저 실행되어야한다.
3. running_p를 실행하고 나서 다시 ptable을 순회하며 우선순위가 가장 높은 process를 찾는다.

원래는 for문을 돌면서 해당 q_lv와 일치하면 context switching을 발생시켜 일을 하고, 스케줄러로 돌아와서 다시 찾는 방식이었다면, 바뀐 방식에서는 우선 ptable을 모두 순회한 다음, ptable에서 가장 우선도가 높은 process를 for문에서 가지고 나와 context switching을 발생. 그리고 다시 포문으로 진입하는 방식으로 바꾸었다.

time interrupt 처리

time interrupt가 발생하는 trap.c 파일을 좀 읽어보았는데 코드 위쪽에서 global ticks를 하나 더해주고 아래쪽에서 yield()를 해주는 형식을 띄고 있었다. yield함수는 결국 현재 프로세스가 cpu를 놓치고 스케줄러를 다시 호출하는 함수이므로, 이 아래쪽에서 타임 인터럽트 처리 후 스케줄러로 돌아간다는 사실을 이해했다. trap.c에 들어와 yield 함수를 발생시켰다는 것은 time interrupt가 발생했다는 것이니까, 이 때 process의 tq와 q_lv, priority 등을 컨트롤해주면 편리하겠다 라는 생각이들었다. 그래서 yield 전에 time quantum을 눌려주고 그 늘어난 time quantum을 기준으로 q_lv을 바꿔주는 디자인을 하기로 했다. Priority의 경우에도 L2 에서 일을하다가 time quantum을 다 소모하면 priority를 올려주고 time quantum도 초기화 해주어야 하므로 여기서 처리해주는 것으로 디자인했다.

priority boosting 구현방식의 결정

우선 global ticks를 어떻게 다뤄주어야 할까를 고민을 많이 했다. system call에 uptime이라고 하는 함수를 하나 발견했는데, 사용해보니 trap.c에 저장되어있는 ticks. 즉 xv6의 global ticks를 반환해주는 함수였다. 그래서 이것을 내가 마음대로 변환을 해도 되는 것일까 생각을 조금 해보았는데, 나의 결론은 이러했다. 만약 xv6를 사용하는 유저가 ticks를 시스템콜로 받아와 사용해야하는 일을 수행하고자 마음먹었다고 가정을 해보자. 내가 xv6의 스케줄러를 업데이트하는 과정에서 priority boosting과 같은 현상이 발생한다는 이유로 ticks를 계속 0으로 초기화 한다면 유저는 원래 사용하고자 하는 의도와 다른 ticks를 얻게 될 것이다. 그래서 우선은 **ticks 변수를 건들지말고, 따로 나만의 global tick을 만들어 관리하자.** 라는 결론에 도달했다. 그럼 이 변수를 어디다 선언해주는 것이 좋을까 고민을 해보았는데, priority boosting을 proc.c에서 처리해줄

예정이었기 때문에 proc.c에 전역변수로 선언할까 하다가 그것보다는 조금 더 의미를 부여하고 싶어서 ptable 구조체 내부에다가 선언을 하기로 했다. 이 global ticks는 온전히 process를 관리하는 과정에서만 필요한 global ticks이니까.

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
  int upper_bound;
  ..
} ptable; // table of process
```

그 후 구현에 관한 디자인은 간단했다. 스케줄러 코드 내부에서 context switching 코드가 완료되고 나면, time interrupt가 발생해서 컨텍스트 스위칭 코드가 완료되었으므로 ticks가 1 늘었다고 판단. ptable에 선언한 global_ticks에 +1을 해주고 이 값이 100보다 크거나 같아지면 ptable을 처음부터 끝까지 순회하면서 time quantum, q_lv, priority등을 초기화해주었다.

---

schedulerLock / schedulerUnlock 구현방식의 결정

기본적인 아이디어는 간단했다. ptable 구조체에다가 lock이 되어있는지 알려주는 변수 하나, lock이 된 process를 저장하는 구조체 하나를 선언하고, `schedulerLock` 함수가 호출이 되면 현재 실행되고 있는 프로세스를 가져와 저장 후 저장되어 있다는 의미롤 변수를 1로 변경해주는 아이디어였다. 그 후, mlfq에 들어가기전 lock이 되어있는지 알려주는 변수의 값을 확인하고 0이라면 mlfq를 실행, 1이라면 ptable에 저장된 lock된 process를 실행해준다. 그리고 나서 상황이 두 가지로 나뉘는데, 첫번째는 주어진 100ticks 안에 process가 종료되었을 경우로 그렇다면 process는 zombie 상태로 들어가게 될 것이기에 스케줄러 내부에서 그것을 확인해 `schedulerUnlock`을 호출했고, 두번째는 100ticks가 되었는데도 process가 종료되지 못했을 경우로, 이 경우에는 처음에 `schedulerLock` 함수가 호출 될 때 global tick이 0으로 초기화 되는 조건을 이용. ptable에 저장된 ticks가 100이 된다면 `schedulerUnlock`을 호출하게 했다. 그리고 Lock함수와 Unlock 함수는 각각 과제 명세에 나와있는 조건을 충실히 수행할 수 있도록 했고, 실습 과제로 학습하였던 인터럽트 추가도 그대로 구현했다.

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
  int upper_bound;
  int is_lock;
  struct proc* lock_process;
} ptable; // table of process
```

그리고 가장 많은 생각을 하게 했던 것이 lock이 해제될 때 MLFQ 스케줄러의 L0 큐 가장 앞으로 이동해야 한다는 명세였다. 내가 만든 mlfq에서는 queue라는 실체가 없기 때문에 가장 앞이라는 것을 따지기가 어려웠기 때문이다. 이것은 Trouble shooting에서 자세하게 이야기하겠다.

Implement

Design 부분에서 process 구조체를 어떻게 구성할 것인지에 관한 설명을 했다. 하지만 좀 더 완벽한 구성을 하기 위해서 Trouble shooting을 해주었다.

Trouble shooting 1

---

proc.h & allocproc()

```
// Per-process state
struct proc {
  int priority;                // Scheduling prioriy
  int tq;                      // Time quantum
  int q_lv;                    // Queue level
  int time;                    // queue arrive time

  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)
};

// Process memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap
```

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->q_lv = 0;
  p->tq = 0;
  p->priority = 3;
  p->time = ticks;

  release(&ptable.lock);

...
}
```

`priority` : L2 큐에서 사용하는 process의 priority. 과제 명세의 priority와 같다. 처음에 3으로 초기화 해준다.

`tq` : 각 프로세스가 실행된 시간. 과제 명세의 Time Quantum와 같다. 처음에 0으로 초기화해준다.

`q_lv` : 각 프로세스가 자신이 속한 큐의 레벨을 명시한다. 처음에는 L0 큐에 해당하니 0으로 초기화해준다.

`time` : 프로세스가 큐에 도착한 시간을 저장한다. 처음에는 프로세스의 탄생 당시의 global ticks로 초기화해준다.

---

proc.c

> ***ptable struct***

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
  int upper_bound;
  int is_lock;
  struct proc* lock_process;
} ptable; // table of process
```

struct ptable

`upper_bound` : priority boost를 할 때 사용하는 global ticks이다.

`is_lock` : 현재 스케줄러가 락이 걸린상태인지 아닌지에 대한 정보를 저장한다. lock이 걸렸다면 1을 저장하고 평상시에는 0이다.

`lock_process` : 스케줄러가 락을 걸었을 때 락이 된 프로세스를 저장하는 구조체이다.

---

systemcall 01. getlevel()

```
int
getLevel(void)
{
  return myproc()->q_lv;
}
```

현재 실행되고 있는 process에 구조체에 저장되어 있는 q_lv를 반환하게 했다.

yield는 원래 system call과 동일하다

systemcall 02. setPriority()

```
void
setPriority(int pid, int priority)
{
  struct proc* p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->pid == pid) {

      //exception handling. if input is not id 0~3
      if(priority<0) p->priority = 0;
      else if(priority>3) p->priority = 3;
      else p->priority = priority;
      break;
    }
  }
  release(&ptable.lock);
}
```

입력으로 들어온 pid에 해당하는 process를 ptable을 순회하며 찾고, 찾으면 해당 프로세스의 priority를 입력으로 들어온 priority로 재설정해준다. 만약에 유저가 0~3 사이의 값을 입력하지 않았다면 0보다 작은 값은 0으로, 3보다 큰 값은 3으로 예외처리를 해준다.

---

systemcall 03. schedulerLock()

```
void
schedulerLock(int password)
{
  acquire(&ptable.lock);

  //exception handling. p is sleeping
  if(myproc()->state == SLEEPING) {
    cprintf("This process is sleeping..\n");
    release(&ptable.lock);
    return;
  }

  // exception handling. already Locked
  if(ptable.is_lock == 1) {
    cprintf("The scheduler is already locked..\n");
    release(&ptable.lock);
    return;
  }
  // check password
  if(password == 2019030991) {
    // edit is_lock and lock_process
    ptable.upper_bound = 0;
    ptable.is_lock = 1;
    ptable.lock_process = myproc();
  }
  else {
    // password ERROR
    cprintf("Password Error Exception / Pid : %d / Time Quantum : %d / Queue_level : %d\n", myproc()->pid, myproc()->tq, myproc()->q_lv);

    release(&ptable.lock);
    exit();
    acquire(&ptable.lock);
  }
  release(&ptable.lock);
}
```

systemcall 04. schedulerUnlock()

```
void
schedulerUnlock(int password)
{
  struct proc* p;

  acquire(&ptable.lock);

  // exception handling. Not locked
  if(ptable.is_lock == 0) {
    cprintf("The scheduler is not locked..\n");
    release(&ptable.lock);
    return;
  }

  // check password
  if(password == 2019030991) {
    ptable.is_lock = 0;

    p = ptable.lock_process;

    p->q_lv = 0;
    p->tq = 0;
    p->priority = 3;

    //make this process L0 queue first
    p->time = -1;

    cprintf("Unlock process : %d / Time Quantum : %d / Queue_level : %d\n", p->pid, p->tq, myproc()->q_lv);
  }

  else {
    // password ERROR
    cprintf("Password Error Exception / Pid : %d / Time Quantum : %d / Queue_leve

    release(&ptable.lock);
    exit();
    acquire(&ptable.lock);
  }
```

우선 lock이 걸릴 수 있는 프로세스는 하나 밖에 없으므로, 이미 다른 프로세스가 lock이 걸려있는데 다시 lock을 하려하면 lock을 해주지 않는다. 그리고 간혹 sleeping된 process가 lock이 걸리는 현상이 목격되어 sleeping인 process는 lock을 걸지 못하게 설정하였다. lock을 걸 수 있는 상황이라면 password가 나의 학번과 같은지 확인하고 학번과 같다면 과제 명세와 맞추기 위하여 upper_bound를 0으로 초기화한다. 그리고 is_lock을 1로 설정하여 lock이 걸려있다는 것을 명시하고 lock된 process를 lock_process에 저장해준다. password가 오류가 있다면 오류 메시지를 출력하는데 과제 명세와 같이 해당 프로세스의 pid와 time quantum, queue level을 출력하고 exit 함수를 호출하여 프로세스를 종료시킨다.

```
  release(&ptable.lock);
}
```

lock이 걸려있지 않은데 unlock을 할 순 없으니 lock이 되어있는지 우선 확인한다. lock이 되어 있고 password가 나의 학번과 일치한다면 is_lock 변수를 0으로 초기화해 lock이 해제되었다는 것을 명시해주고 과제 명세와 맞추기 위하여 tq와 priority를 설정해준다. 그리고 time을 -1로 설정하여 L0큐에서 가장 먼저 실행될 수 있게 설정한다. password가 오류가 있다면 오류 메시지를 출력하는데 과제 명세와 같이 해당 프로세스의 pid와 time quantum, queue level을 출력하고 exit 함수를 호출하여 프로세스를 종료시킨다.

systemcall → sysproc.c

```
int
sys_setPriority(void)
{
  int pid, priority;
  if(argint(0,&pid)<0) return -1;
  if(argint(1,&priority)<0) return -1;
  setPriority(pid,priority);
  return 0;
}

int
sys_yield(void)
{
  myproc()->q_lv = 2;
  myproc()->tq = 0;
  myproc()->time = ticks;
  myproc()->priority = 3;

  yield();
  return 0;
}

int
sys_getLevel(void)
{
  int ret = getLevel();
  if(ret < 0 || ret > 2) return -1;
  return ret;
}
```

```
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;
}
```

레퍼 function들은 모두 sysproc.c에 작성하였다. 입력을 받아야 하는 system call들은 입력이 없다면 return -1을 하게 했으며, getLevel은 정상적이지 않은 level을 반환할 경우 return -1을 하게 했다. yield에 경우에는 시스템콜로 유저가 호출할 경우, L2 queue에 위치하게 디자인했다. yield를 주기적으로 반복해서 다른 프로세스들이 L0와 L1에서 실행되는것을 막는 현상을 막기 위해서이다.

systemcall setting

```
void setPriority(int,int);
void yield(void);
int getLevel(void);
void schedulerLock(int);
void schedulerUnlock(int);
```

user.h

```
#define prac            128
#define T_SCHEDLOCK     129
#define T_SCHEDUNLOCK   130
```

traps.h

스케줄러 lock과 unlock을 interrupt로도 처리해야하므로 설정했다.

```
void            yield(void);
void            setPriority(int, int);
int             getLevel(void);
void            schedulerLock(int);
void            schedulerUnlock(int);
```

defs.h

```
#define SYS_yield 23
#define SYS_getLevel 24
#define SYS_setPriority 25
#define SYS_schedulerLock 26
#define SYS_schedulerUnlock 27
```

syscall.h

```
extern int sys_setPriority(void);
extern int sys_yield(void);
extern int sys_getLevel(void);
extern int sys_schedulerLock(void);
extern int sys_schedulerUnlock(void);
```

syscall.c

```
SYSCALL(setPriority)
SYSCALL(yield)
SYSCALL(getLevel)
SYSCALL(schedulerLock)
SYSCALL(schedulerUnlock)
```

usys.S

```
[SYS_setPriority] sys_setPriority,
[SYS_yield] sys_yield,
[SYS_getLevel] sys_getLevel,
[SYS_schedulerLock] sys_schedulerLock,
[SYS_schedulerUnlock] sys_schedulerUnlock,
```

syscall.c

실습시간에 배웠던 방식으로 시스템콜을 그대로 구현했다.

User_systemcall function

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

int
main(int argc, char *argv[])
{
    if(argc <= 2) {
        exit();
    }

    int ipt_pid = atoi(argv[1]);
    int ipt_priority = atoi(argv[2]);

    setPriority(ipt_pid,ipt_priority);
    printf(1, "setPriority Complete.\n");

    exit();
};
```

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

int
main(int argc, char *argv[])
{
    if(argc <= 1) {
        exit();
    }

    int password = atoi(argv[1]);

    schedulerLock(password);

    printf(1, "SchedulerLock finish\n");

    exit();
};
```

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

int
main(int argc, char *argv[])
{
    yield();
    printf(1, "yield complete\n");
    exit();
};
```

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

int
main(int argc, char *argv[])
{
    if(argc <= 1) {
        exit();
    }

    int password = atoi(argv[1]);

    schedulerUnlock(password);

    printf(1, "SchedulerUnLock finish\n");

    exit();
};
```

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

int
main(int argc, char *argv[])
{
    int lvl = getLevel();
    printf(1, "Running Process level : %d\n", lvl);
    exit();
};
```

모든 유저 함수에서는 함수를 실행하고 잘 실행되었다고 print 후 바로 꺼지게 만들었다

scheduler

⚙️ ***scheduler flow***

1. scheduler lock process
2. mlfq scheduler process
→ 2-1 selecting running_p process
→ 2-2 context switching running_p
3. priority boost & scheduler unlock process

scheduler lock process. *스케줄러가 락이 걸려있을 때 처리해주는 부분이다.*

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

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

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);

    struct proc* running_p = 0; // chosen process to run
    int chk = 0; //

    if(ptable.is_lock) {
      //sched lock process.

      running_p = ptable.lock_process;

      //In the middle of the lock processing, process can sleep or die.
      //so we have to check state of running_p
      if(running_p->state == RUNNABLE) {

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.

      c->proc = running_p;
      switchuvm(running_p);
      running_p->state = RUNNING;

      //cprintf("!!LOCKED!! pid : %d | q_lv : %d | tq : %d| priority : %d | upper_bound : %d  \n", running_p->pid, running_p->q_lv , running_p->tq, running_p->priority,

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

      running_p->time = ticks;

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
      }
    }
```

`running_p` : 해당 스케줄러가 스케줄링을 통해 컨텍스트 스위칭을 해 running으로 state를 변경해줄 프로세스

`chk` : mlfq 스케줄러가 가장 처음에 running_p가 비어있을 때 스케줄링을 할 수 있게 해주는 변수

schedulerLock 함수를 호출하면 ptable에 저장된 is_lock을 1로 바꿔주고 lock process에 해당 프로세스를 저장해주기 때문에, lock이 걸려있다면 if문에 조건에 맞아 scheduler lock process에 진입하게 된다. 그 후 running_p를 lock_process로 설정해주고 컨텍스트 스위칭을 해준다. time은 다시 큐에 들어갈 때의 global ticks로 초기화 해준다. 하지만 중간에 프로세스가 runnable한 상태가 아니라 zombie가 되거나 sleeping에 들어가게 된다면 더이상 일을 하면 안되니 통과시켰다. zombie 프로세스에 경우에는 아래에서 스케줄러 락을 해제하게 시켰고, sleep에 경우에는 아직 프로세스가 다시 깨어날 수 있는 가능성이 존재하니 100틱이 찰 때 까지는 공회전을 하게 했다.

> **mlfq scheduler process.** *lock이 걸려있지 않을 때 mlfq 스케줄링을 해주는 부분이다.*

> **selecting running_p process.** *runnable한 process 중 running 시켜줄 프로세스를 선택하는 부분이다.*

```
else{
      //mlfq_sched
      //find which process is have to running-process
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if(p->state != RUNNABLE) continue;

      // if queue level different, compare queue level
      if(!chk || running_p->q_lv > p->q_lv) {
        running_p = p;
        chk = 1; // to put first process in running_p
      }

      // if queue level same, compare priority and time
      else if (running_p->q_lv == p->q_lv) {

        if(running_p->q_lv == 2){ // L2 Queue
          // if priority is different, compare priority. if priority is same, compare time
          if((running_p->priority > p->priority)) running_p = p;
          else if((running_p->priority == p->priority) && (running_p->time > p->time)) running_p = p;
        }

        else { // L0, L1 Queue. compare time
          if(running_p->time > p->time) running_p = p;
        }
      }
    }
```

ptable을 처음부터 끝까지 확인하면서 running_p를 고른다.

1. runnable한 프로세스가 아니라면 스케줄링 대상에서 제외한다.
2. runnable 하다면, process의 큐 레벨을 비교한다. 큐 레벨이 다르다면, 우선순위가 높은 큐 레벨의 process를 running_p로 설정한다. 단, 처음에 running_p는 ptable에 들어있는 프로세스가 아니어서 초기화 되어있지 않으므로 비교할 수 없다. 그러므로 가장 처음 process는 chk 변수가 0이므로 무조건 넣어주고 chk를 1로 만들어 준다.
3. 비교할 두 process의 큐 레벨이 같다면
   → 3-1. L0,L1에 경우에는 round-robin과 FCFS를 보장해주기 위해 time 변수를 비교하여 더 먼저 큐에 들어온 process를 스케줄링 대상으로 설정한다.
   → 3-2. L2에 경우에는 priority를 비교해 priority가 우선도가 더 높은 process를 실행. 만약에 priority 마저 같다면 L0,L1과 같이 time 변수를 비교해 더 먼저 큐에 들어온 process를 스케줄링.

> **context switching running_p.** *선택한 running_p를 컨텍스트 스위칭해준다.*

```
if(chk == 1) {
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.

      c->proc = running_p;
      switchuvm(running_p);
      running_p->state = RUNNING;

      //cprintf("pid : %d | q_lv : %d | tq : %d| priority : %d | upper_bound : %d  \n", running_p->pid, running_p->q_lv , running_p->tq, running_p->priority, ptable.upp

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

      running_p->time = ticks;

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;

    }
  }
```

running_p를 context switching 해준다. 그 후 큐에 들어가기 전에 time 변수를 global ticks로 초기화 해준다.

> **priority boost & scheduler unlock process.** *100 ticks 마다 priority boost를 발생시켜준다. scheduler lock이 걸려있었다면 lock을 상황에 따라 해제해줄지 결정한다.*

```
//  if scheduler lock_process finish, unlock scheduler
    if(ptable.is_lock == 1 && ptable.lock_process->state == ZOMBIE ) {
      release(&ptable.lock);
      schedulerUnlock(2019030991);
      acquire(&ptable.lock);
    }

    ptable.upper_bound++;

    if(ptable.upper_bound >= 100) { //Priority boosting
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        p->tq = 0;
        p->q_lv = 0;
        p->priority = 3;
        p->time = 0;
      }

      // if scheduler lock and global ticks == 100, unlock scheduler
      if(ptable.is_lock == 1) {
        release(&ptable.lock);
        schedulerUnlock(2019030991);
        acquire(&ptable.lock);
      }
      ptable.upper_bound = 0;
    }
      release(&ptable.lock);
  }
}
```

context switching이 종료되면 priority boost global ticks을 증가시켜준 뒤 두 가지를 체크해준다. priority boosting을 해야하는가, 그리고 지금 lock을 해제해야하는가.

priority boost에 경우 지금 upper_bound가 100에 도달했다면 ptable에 있는 모든 프로세스를 과제 명세와 맞게 모두 초기화 해주고 q_lv을 0으로 설정해주어 L0 프로세스로 모두 복귀시켜준다. time도 모두 0으로 초기화 해주어 다시 ptable 기준 가장 앞에 있는 프로세스부터 스케줄링 대상이 될 수 있게 해준다.

lock이 해제되는 기준은 두 가지가 있는데 하나는 lock_process가 100ticks 안에 종료되었을때, 그리고 100 ticks를 모두 소모했을 때이다. lock_process가 종료되었다면 종료되었는지 확인하고 `schedulerUnlock`을 호출한다. 100 ticks를 모두 소모했다면 ptable 초기화 이후 schedulerUnlock을 호출해준다. 그래야 unlock된 process의 time 변수만 -1로 초기화해 L0큐 가장 앞에 놓을 수 있다.

> ***etc.***

```
int
fork(void)
{
  int i, pid;
```

```
  struct proc *np;
  struct proc *curproc = myproc();

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

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  np->tq = 0;
  np->q_lv = 0;
  np->time = ticks;
  np->priority = 3;
```

fork로 인하여 새로 생긴 프로세스는 무조건 부모 프로세스와 mlfq 스케줄링 관련 정보를 공유하지 않도록 조정하였다.

trap.c

tvinit()

```
void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
  SETGATE(idt[T_SCHEDLOCK], 1, SEG_KCODE<<3, vectors[T_SCHEDLOCK], DPL_USER);
  SETGATE(idt[T_SCHEDUNLOCK], 1, SEG_KCODE<<3, vectors[T_SCHEDUNLOCK], DPL_USER);

  initlock(&tickslock, "time");
}
```

128번과 129번에 `schedulerlock` 과 `schedulerunlock`이 interrupt를 실행할 수 있도록 추가해주었다.

sched Lock / Unlock Interrupt

```
if(tf->trapno == T_SCHEDLOCK){
    schedulerLock(2019030991);
    exit();
  }

  if(tf->trapno == T_SCHEDUNLOCK){
    schedulerUnlock(2019030991);
    exit();
  }
```

과제 명세에 맞추기 위하여 128번과 129번 interrupt가 호출되면 각 함수를 호출했다.

time interrupt 처리

```
if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER) {

      // calculate time quantum
      myproc()->tq++;

      // check queue level
      if(myproc()->tq == (myproc()->q_lv)*2 + 4) {

        // if L2, change priority
        if(myproc()->q_lv == 2) {
          setPriority(myproc()->pid, (myproc()->priority) - 1);
        }

        // else, chagne queue level
        else myproc()->q_lv++;

        // make tq == 0
        myproc()->tq = 0;
      }

      yield();
     }
```

time interrupt가 발생했을 때 yield하기 직전에 time quantum가 queue level을 관리해주는 코드를 작성했다. 매 time interrupt마다 해당 프로세스의 tq를 하나 증가시켜주고 해당 q_lv에서 time quantum을 다 사용했다면 q_lv을 올려주고, L2에 경우에는 priority를 올려주었다. `setPriority` 함수에서 예외 처리를 해 두었기 때문에 그냥 -1을 계속해도 priority가 0으로 고정된다.

Result

```
290     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
291       if(p->parent != curproc)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00


Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
```

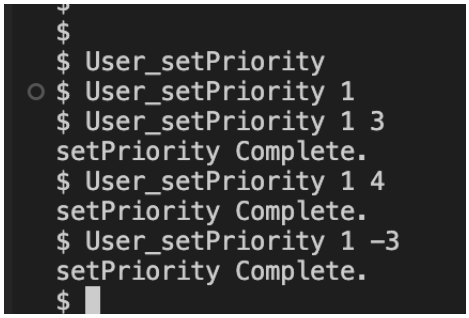
우선 성공적으로 booting이 된 모습이다.

> ***systemcall***

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15652
echo           2 4 14532
forktest       2 5 8968
grep           2 6 18488
init           2 7 15152
kill           2 8 14620
ln             2 9 14516
ls             2 10 17084
mkdir          2 11 14640
rm             2 12 14620
sh             2 13 28668
stressfs       2 14 15552
wc             2 15 16068
zombie         2 16 14192
myapp          2 17 14640
prac2_usercall 2 18 14056
User_setPriori 2 19 14696
User_yield     2 20 14376
User_getLevel  2 21 14460
User_schedlock 2 22 14584
User_schedunlo 2 23 14592
int_test       2 24 14048
mlfq_test      2 25 15596
console        3 26 0
$
```

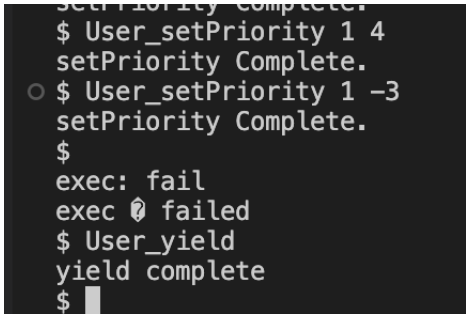
ls를 호출했을 때 나타나는 화면이다.

User_setPriority



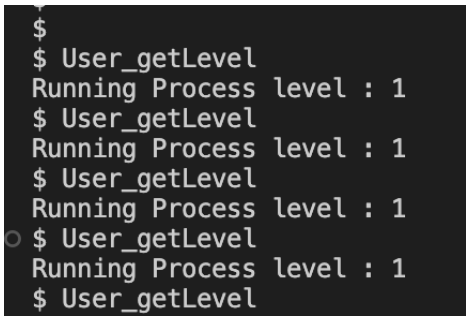
입력으로 숫자 2개가 들어오지 않으면 실행되지 않게 하였고, 2개가 들어오면 setPriority를 수행하도록 했다. priority 범위를 넘어가는 숫자에 대해서도 예외처리를 해서 처리가 가능하게 했다. setPriority 함수 같은 경우에는 trap.c에서 활발하게 이용하고 있기 때문에 mlfq가 잘 작동한다면 그 효과가 올바르다고 볼 수 있을 것 같다.

User_yield



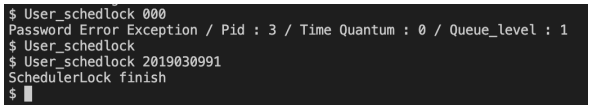
정말 단순하게 proc.c에 기본적으로 구현되어있는 yield 시스템 콜을 호출하고 yield가 되었다고 print만 하는 함수이다.

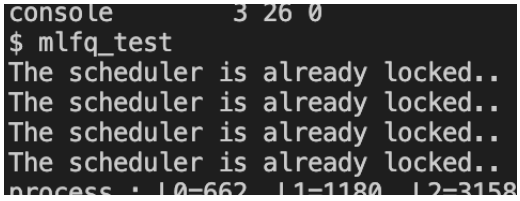
User_getLevel



이 함수도 상당히 단순하게 myproc()에 접근하여 q_lv을 반환하게 하였다. 그리고 나서 잘 반환했으면 print하라 했다.

User_schedlock





password가 올바르지 않다면 pid와 tq, q_lv를 말해주고 process가 죽고, 비밀번호가 들어오지 않는다면 아예 실행되지 않는다. password가 올바르게 들어 왔다면 완료되었다고 print를 해준다. 잠시 후에 볼 fork를 여러번 하는 실험 코드에서 lock이 걸린 상태에서 여러번 lock을 하려 하면 lock이 이미 되어있어 할 수 없다는 메시지를 내보낸다.

lock이 걸린상태에서 잘못된 비밀번호로 lock이 실행되려 해도 lock이 이미 설정되어 있기 때문에 lock을 다시 거는 것 자체를 금하는 디자인을 했다.

User_schedunlock

```
$ User_schedunlock
$ User_schedunlock 123
The scheduler is not locked..
SchedulerUnLock finish
$ User_schedunlock 2019030991
The scheduler is not locked..
SchedulerUnLock finish
$ QEMU: Terminated
root@011ac41df8a3:/OS/xv6-public#
```

password가 아예 들어오지 않는다면 실행되지 않는다. 그리고 기본적으로 lock이 걸려있어야 lock을 해제해주므로 user_schedunlock의 기본 출력은 락이 되어있지 않아 실행할 수 없다는 메시지이다.

unlcok도 lock과 마찬가지로 이미 해제 되어있을 때 unlock을 한다고 해서 프로세스를 종료 시키진 않았다. 해제가 되어있는데 해제를 하려하는 것 자체를 디자인에서 막았다.

```
$ mlfq_test
The scheduler is already locked..
The scheduler is already locked..
The scheduler is already locked..
The scheduler is already locked..
Password Error Exception / Pid : 4 / Time Quantum : 0 / Queue_level : 2
$ The scheduler is not locked..
The scheduler is not locked..
The scheduler is not locked..
The scheduler is not locked..
zombie!
zombie!
zombie!
zombie!
zombie!
```

mlfq_teset 함수에서 부모에 lock을 걸고 unlock에 비밀번호를 0000으로 주고 실행한 결과인데, 비밀번호가 맞지 않으니 성공적으로 종료되고 부모가 먼저 종료되어 다섯 개의 좀비프로세스가 생성된 모습이다. 부모 프로세스는 unlock에 fail해 exit함수를 호출하여 죽은 후, scheduler에서 zombie 상태가 된 것을 확인하고 unlock을 자동으로 해준다.

나의 lock 코드에서는 기본적으로 유저의 실수로 좀비프로세스가 생생되는 것을 막아주진 않았다.

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 b
init: starting sh
$ mlfq_test
The scheduler is already locked..
The scheduler is already locked..
The scheduler is already locked..
The scheduler is already locked..
The scheduler is not locked..
The scheduler is not locked..
The scheduler is not locked..
The scheduler is not locked..
The scheduler is not locked..
$ QEMU: Terminated
```

lock도 unlock도 제대로 걸어 terminate 되는 process 없이 잘 실행되고 zombie도 발생하지 않는 모습이다.

---

lock(2019030991) → lock(2019030991) / lock(0000) = 이미 락되어 락을 할 수 없다.

lock(2019030991) → unlock(2019030991) = 성공적으로 lock 풀림

lock(2019030991) → unlock(0000) = lock된 프로세스가 exit되어 죽고 스케줄러가 lock process가 죽은것을 확인하고 unlock 처리함

unlock(201903991)/unlock(0000) = lock된 프로세스가 없으므로 unlock 을 할 수 없다.

---

int_test

int_test.c

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

int
main(int argc, char *argv[])
{
    __asm__("int $130");
    return 0;
}
```

```
stressfs        2 14 15552
wc              2 15 16068
zombie          2 16 14192
myapp           2 17 14640
prac2_usercall 2 18 14056
User_setPriori 2 19 14684
User_yield      2 20 14376
User_getLevel   2 21 14460
User_schedlock 2 22 14584
User_schedunlo 2 23 14592
int_test        2 24 14048
mlfq_test       2 25 15584
console         3 26 0
$ int_test
The scheduler is not locked..
$
```

interrupt로도 lock과 unlock 함수를 실행할 수 있는지 테스트해 볼 수 있는 함수이다. 보는 것과 같이 잘 실행된다.

mlfq_test - fork()

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

#define NUM_CHILD 5
#define NUM_LOOP1 50000
#define NUM_LOOP2 1000000
#define NUM_LOOP3 200000
#define NUM_LOOP4 1000000

int me;
int create_child(void){
  for(int i =0  ; i<NUM_CHILD; i++){
    int pid = fork();
    if(pid == 0){
      me = i;
      sleep(10);
      return 0;
    } //else schedulerLock(2019030991);
  }
  return 1;
}

void exit_child(int parent) {
  if (parent)
    while (wait() != -1); // wait for all child processes to finish
  exit();
}

int main()
{
  int p;
  p = create_child();
  //schedulerLock(2019030991);

  if (!p) {
    int arr[3] = {0, };
    for (int i = 0; i < NUM_LOOP1; i++) {
      cnt[getLevel()]++;
    }
    printf(1, "pid %d : L0=%d, L1=%d, L2=%d\n", getpid(), arr[0], arr[1], arr[2]);
  }
  exit_child(p);
}
```

동시에 여러 프로세스들이 스케줄링 되는 모습을 지켜 보아야 mlfq 스케줄러가 스케줄링을 잘 하고 있는지 지켜볼 수 있으므로 fork 명령어를 통해 자식을 생성해 동시에 여러개의 프로세스가 돌아가게 하고, 실행 시간을 NUM_LOOP의 크기를 조절하여 컨트롤했다. 위에서 lock, unlock을 실험할 때 사용한 코드가 이 코드이고, 주석 부분을 이용했다. arr에 프로세스가 각 level에 몇번 들어가는 연산하여 저장했다.

50,000/500,000 연산

```
init: starting sh
$ mlfq_test
pid 4 : L0=7031, L1=10526, L2=32443
pid 5 : L0=9139, L1=13502, L2=27359
pid 6 : L0=9671, L1=14967, L2=25362
pid 7 : L0=10585, L1=15286, L2=24129
pid 8 : L0=10352, L1=15586, L2=24062
```

50000번 연산

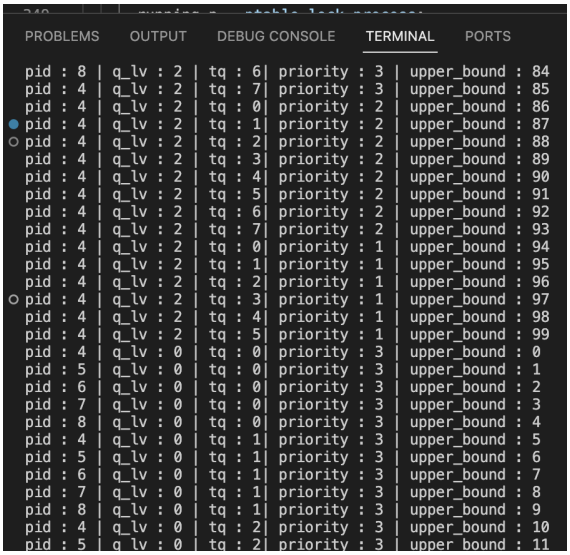
```
process : L0=63457, L1=94495, L2=342048
process : L0=82398, L1=123446, L2=294156
process : L0=91373, L1=137373, L2=271254
process : L0=97880, L1=144902, L2=257218
process : L0=102180, L1=152602, L2=245218
$ QEMU: Terminated
```

실행 결과로 각 프로세스가 50000번씩 연산되고 있으며 L0와 L1의 실행 횟수를 4ticks 6ticks 만큼 사용하니 대략 1: 1.5의 비율을 유지한다는 것을 확인할 수 있다. 그 외 나머지 실행은 L2 큐에서 일어나는 것을 알 수 있다. 500000번 연산에서도 잘 실행된다.

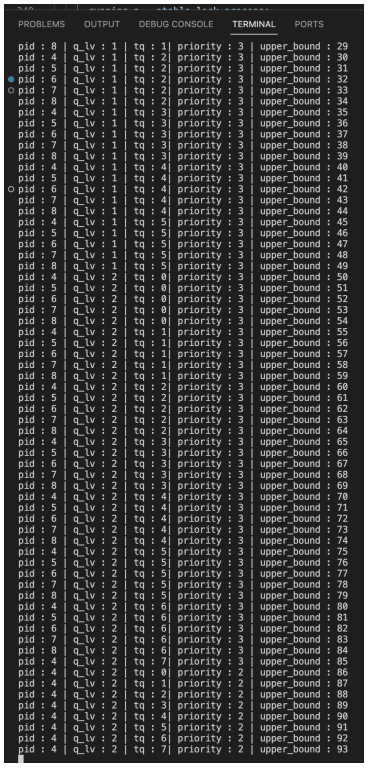
직접 실행된 프로세스 정보 출력해보기

```
if(chk == 1) {
  // Switch to chosen process.  It is the process's job
  // to release ptable.lock and then reacquire it
  // before jumping back to us.

  c->proc = running_p;
  switchuvm(running_p);
  running_p->state = RUNNING;

  cprintf("pid : %d | q_lv : %d | tq : %d| priority : %d | upper_bound : %d  \n", running_p->pid, running_p->q_lv , running_p->tq, running_p->priority, ptable.upper_bound);

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

  running_p->time = ticks;

  // Process is done running for now.
  // It should have changed its p->state before coming back.
  c->proc = 0;
```

어떤 프로세스가 context switching의 대상이 되고 있는지 확인해 보기 위해서 running 직전에 계속해서 information을 찍어 보았다.



priority boost



일반적인 실행모습

mlfq 구현 명세와 같이 잘 돌아간다는 것을 결과값을 통해 확인할 수 있다. 그리고 priority boost가 발생하면 다섯개의 process 모두 다시 L0로 초기화 되어 잘 실행된다는 사실도 결과값을 통해 확인할 수 있다.

Lock 기능 확인

```
if(ptable.is_lock) {
      //sched lock process.

      running_p = ptable.lock_process;

      //In the middle of the lock processing, process can sleep or die.
      //so we have to check state of running_p
      if(running_p->state == RUNNABLE) {

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.

      c->proc = running_p;
      switchuvm(running_p);
      running_p->state = RUNNING;

      cprintf("!!LOCKED!! pid : %d | q_lv : %d | tq : %d| priority : %d | upper_bound : %d  \n", running_p->pid, running_p->q_lv , running_p->tq, running_p->priority, p

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

      running_p->time = ticks;

      // Process is done running for now.
      // It should have changed its p->state before coming back.
```

```
      c->proc = 0;
      } else cprintf("Just Looping\n");
    }
```

```
int main()
{
  int p;
  int yieldcnt = 0;
  p = create_child();
  schedulerLock(2019030991);
..
}
```

```
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 3| priority : 0 | upper_bound : 89
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 4| priority : 0 | upper_bound : 90
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 5| priority : 0 | upper_bound : 91
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 6| priority : 0 | upper_bound : 92
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 7| priority : 0 | upper_bound : 93
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 0| priority : 0 | upper_bound : 94
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 1| priority : 0 | upper_bound : 95
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 2| priority : 0 | upper_bound : 96
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 3| priority : 0 | upper_bound : 97
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 4| priority : 0 | upper_bound : 98
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 5| priority : 0 | upper_bound : 99
pid : 5 | q_lv : 0 | tq : 0| priority : 3 | upper_bound : 0
pid : 3 | q_lv : 0 | tq : 0| priority : 3 | upper_bound : 1
pid : 4 | q_lv : 0 | tq : 0| priority : 3 | upper_bound : 2
pid : 5 | q_lv : 0 | tq : 1| priority : 3 | upper_bound : 3
pid : 3 | q_lv : 0 | tq : 1| priority : 3 | upper_bound : 4
pid : 6 | q_lv : 0 | tq : 0| priority : 3 | upper_bound : 5
pid : 4 | q_lv : 0 | tq : 1| priority : 3 | upper_bound : 6
pid : 5 | q_lv : 0 | tq : 2| priority : 3 | upper_bound : 7
pid : 3 | q_lv : 0 | tq : 2| priority : 3 | upper_bound : 8
pid : 6 | q_lv : 0 | tq : 0| priority : 3 | upper_bound : 9
pid : 4 | q_lv : 0 | tq : 2| priority : 3 | upper_bound : 10
pid : 5 | q_lv : 0 | tq : 3| priority : 3 | upper_bound : 11
```

lock이 풀리고나서 가장 앞으로 가는 모습

```
pid : 5 | q_lv : 0 | tq : 1| priority : 3 | upper_bound : 19
pid : 5 | q_lv : 0 | tq : 2| priority : 3 | upper_bound : 20
!!LOCKED!! pid : 5 | q_lv : 0 | tq : 3| priority : 3 | upper_bound : 1
!!LOCKED!! pid : 5 | q_lv : 1 | tq : 0| priority : 3 | upper_bound : 2
!!LOCKED!! pid : 5 | q_lv : 1 | tq : 1| priority : 3 | upper_bound : 3
!!LOCKED!! pid : 5 | q_lv : 1 | tq : 2| priority : 3 | upper_bound : 4
!!LOCKED!! pid : 5 | q_lv : 1 | tq : 3| priority : 3 | upper_bound : 5
!!LOCKED!! pid : 5 | q_lv : 1 | tq : 4| priority : 3 | upper_bound : 6
!!LOCKED!! pid : 5 | q_lv : 1 | tq : 5| priority : 3 | upper_bound : 7
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 0| priority : 3 | upper_bound : 8
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 1| priority : 3 | upper_bound : 9
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 2| priority : 3 | upper_bound : 10
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 3| priority : 3 | upper_bound : 11
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 4| priority : 3 | upper_bound : 12
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 5| priority : 3 | upper_bound : 13
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 6| priority : 3 | upper_bound : 14
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 7| priority : 3 | upper_bound : 15
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 0| priority : 2 | upper_bound : 16
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 1| priority : 2 | upper_bound : 17
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 2| priority : 2 | upper_bound : 18
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 3| priority : 2 | upper_bound : 19
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 4| priority : 2 | upper_bound : 20
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 5| priority : 2 | upper_bound : 21
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 6| priority : 2 | upper_bound : 22
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 7| priority : 2 | upper_bound : 23
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 0| priority : 1 | upper_bound : 24
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 1| priority : 1 | upper_bound : 25
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 2| priority : 1 | upper_bound : 26
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 3| priority : 1 | upper_bound : 27
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 4| priority : 1 | upper_bound : 28
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 5| priority : 1 | upper_bound : 29
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 6| priority : 1 | upper_bound : 30
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 7| priority : 1 | upper_bound : 31
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 0| priority : 0 | upper_bound : 32
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 1| priority : 0 | upper_bound : 33
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 2| priority : 0 | upper_bound : 34
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 3| priority : 0 | upper_bound : 35
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 4| priority : 0 | upper_bound : 36
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 5| priority : 0 | upper_bound : 37
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 6| priority : 0 | upper_bound : 38
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 7| priority : 0 | upper_bound : 39
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 0| priority : 0 | upper_bound : 40
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 1| priority : 0 | upper_bound : 41
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 2| priority : 0 | upper_bound : 42
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 3| priority : 0 | upper_bound : 43
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 4| priority : 0 | upper_bound : 44
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 5| priority : 0 | upper_bound : 45
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 6| priority : 0 | upper_bound : 46
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 7| priority : 0 | upper_bound : 47
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 0| priority : 0 | upper_bound : 48
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 1| priority : 0 | upper_bound : 49
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 2| priority : 0 | upper_bound : 50
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 3| priority : 0 | upper_bound : 51
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 4| priority : 0 | upper_bound : 52
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 5| priority : 0 | upper_bound : 53
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 6| priority : 0 | upper_bound : 54
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 7| priority : 0 | upper_bound : 55
!!LOCKED!! pid : 5 | q_lv : 2 | tq : 0| priority : 0 | upper_bound : 56
```

lock 결과물

lock을 걸고도 print를 해보았는데, 총 다섯개의 프로세스가 모두 락이 걸린 메시지가 출력이 되며, lock이 걸리면 올바르게 한 프로세스만 계속 실행될 수 있게 잘 출력이 되었다.

```
int create_child(void){
  for(int i =0  ; i<NUM_CHILD; i++){
    int pid = fork();
    if(pid == 0){
      me = i;
      sleep(10);
      return 0;
    } else schedulerLock(2019030991);
  }
  return 1;
}

void exit_child(int parent) {
  if (parent)
    while (wait() != -1); // wait for all child processes to finish
  exit();
}

int main()
{
  int p;
  int yieldcnt = 0;
  p = create_child();
  //schedulerLock(2019030991);
```

```
$ mlfq_test
!!LOCKED!! pid : 3 | q_lv : 1 | tq : 1| priority : 3 | upper_bound : 1
The scheduler is already locked..
!!LOCKED!! pid : 3 | q_lv : 1 | tq : 2| priority : 3 | upper_bound : 2
The scheduler is already locked..
!!LOCKED!! pid : 3 | q_lv : 1 | tq : 3| priority : 3 | upper_bound : 3
The scheduler is already locked..
!!LOCKED!! pid : 3 | q_lv : 1 | tq : 4| priority : 3 | upper_bound : 4
!!LOCKED!! pid : 3 | q_lv : 1 | tq : 5| priority : 3 | upper_bound : 5
The scheduler is already locked..
The scheduler is already locked..
Just Looping
Just Looping
Just Looping
Just Looping
Just Looping
Just Looping
Just Looping
Just Looping
Just Looping
Just Looping
Just Looping
Just Looping
Just Looping
Just Looping
Just Looping
```

create child 부분에 else 부분에다가 lock을 걸면 부모 프로세스만 lock을 걸 수 있는데, 그러면 곧 부모 프로세스는 sleep에 들어가게 된다. 그래서 더 이상 많은 일을 하지 않고 공회전을 하다가 lock이 풀리는 모습. print는 실행이 될때만 찍힌다.

---

mlfq_test_piazza

```
int main(int argc, char *argv[])
{
  int i, pid;
  int count[MAX_LEVEL] = {0};
//  int child;
//  MAX_LEVEL = 3
  parent = getpid();

  printf(1, "MLFQ test start\n");

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

  if (pid != parent)
  {
    for (i = 0; i < NUM_LOOP; i++)
    {
      int x = getLevel();
      if (x < 0 || x > 3)
      {
        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, "L%d: %d\n", i, count[i]);
  }
  exit_children();
  printf(1, "[Test 1] finished\n");
  printf(1, "done\n");
  exit();
}
```

```
init: starting sh
$ mlfq_test
MLFQ test start
[Test 1] default
Process 4
L0: 9010
L1: 12644
L2: 78346
Process 5
L0: 13228
L1: 18923
L2: 67849
Process 6
L0: 15238
L1: 22038
L2: 62724
Process 7
L0: 16961
L1: 24132
L2: 58907
[Test 1] finished
done
$
```

조교님께서 제공해주신 test case를 굴려보았다. 기본적인 작동 방식이 내가 전에 사용했던 test와 form이 비슷하기 때문에 결과도 비슷한 양상으로 나오는 모습이다.

```
printf(1, "[Test 2] yield\n");
   pid = fork_children();

   if (pid != parent)
   {
     for (i = 0; i < NUM_YIELD; i++)
     {
       int x = getLevel();
       if (x < 0 || x > 3)
       {
         printf(1, "Wrong level: %d\n", x);
         exit();
       }
```

```
      count[x]++;
      yield();
    }
    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 2] finished\n");
```

```
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodes
init: starting sh
$ mlfq_test
[Test 2] yield
Process 4
Process 4 , L0: 226
Process 4 , L1: 0
Process 4 , L2: 19774
Process 5
Process 5 , L0: 442
Process 5 , L1: 0
Process 5 , L2: 19558
Process 6
Process 6 , L0: 639
Process 6 , L1: 0
Process 6 , L2: 19361
Process 7
Process 7 , L0: 818
Process 7 , L1: 0
Process 7 , L2: 19182
[Test 2] finished
done
```

만약에 계속 유저가 yield를 호출하게 되면 어떻게 될지 실험해보았다. 나는 유저에 의해 yield되는 함수에 의해서 다른 프로세스들이 mlfq에 의해 스케줄링 되는 것을 방해받는 것이 싫었다. 만약에 yield를 주기적으로 반복하는 프로세스가 L0 queue에 있다면 mlfq관련 정보가 업데이트 되지 않아 그 process만 계속 실행될 것이기 때문에 시스템 전반적으로 좋은 영향을 끼칠 것이라고 생각하지 않았다. 그래서 유저에 의해 yield되는 함수를 L2 queue에 priority 3으로 만들어 주어 우선순위를 가장 낮추었다. time의 경우에도 해당 시점의 global ticks로 변경되어 말그대로 가장 후순위에 스케줄이 되는 process가 된다. 대신에 yield가 된 함수를 제외한 다른 프로세스들은 mlfq를 잘 실행할 수 있다. 유저는 이를 감안해서 yield를 호출하면 후순위에 스케줄링이 될것이라 생각하고 호출을 해야할 것이다. 지속적으로 yield를 호출하는 process를 보면 그래서 L2 queue에서 활동을 하는 것을 알 수 있다.

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

#define NUM_CHILD 5
#define NUM_LOOP1 50000
#define NUM_LOOP2 1000000
#define NUM_LOOP3 200000
#define NUM_LOOP4 1000000

int me;

int create_child(void){
  for(int i =0  ; i<NUM_CHILD; i++){
    int pid = fork();
    if(pid == 0){
      me = i;
      sleep(10);
      return 0;
    } //else schedulerLock(2019030991);
  }
  return 1;
}

void exit_child(int parent) {
  if (parent)
    while (wait() != -1); // wait for all child processes to finish
  exit();
}

int main()
{
  int p;
  int yieldcnt = 0;
  p = create_child();
  //schedulerLock(2019030991);

  if (!p) {
    int arr[3] = {0, };
    for (int i = 0; i < NUM_LOOP1; i++) {
      arr[getLevel()]++;
        if(me/2 == 0) {
        yieldcnt++;
        yield();
      }
      }

    printf(1, "me : %d | L0=%d, L1=%d, L2=%d | Yieldcnt : %d \n",me, arr[0], arr[1], arr[2], yieldcnt);
  }

  exit_child(p);
}
```

```
Booting from Hard Disk...
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
exec: fail
exec _  failed
$ mlfq_test
me : 2 | L0=5673, L1=8341, L2=35986 | Yieldcnt : 0
me : 3 | L0=8002, L1=11806, L2=30192 | Yieldcnt : 0
me : 4 | L0=12360, L1=19060, L2=18580 | Yieldcnt : 0
me : 0 | L0=543, L1=0, L2=49457 | Yieldcnt : 50000
me : 1 | L0=1029, L1=0, L2=48971 | Yieldcnt : 50000
$
```

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ mlfq_test
me : 1 | L0=4935, L1=7328, L2=37737 | Yieldcnt : 0
me : 3 | L0=13058, L1=21127, L2=15815 | Yieldcnt : 0
me : 0 | L0=554, L1=0, L2=49446 | Yieldcnt : 50000
me : 2 | L0=1071, L1=0, L2=48929 | Yieldcnt : 50000
me : 4 | L0=1541, L1=0, L2=48459 | Yieldcnt : 50000
$
```

그럼 특정 프로세스만 연속적으로 yield를 하면 어떻게 될지도 궁금해서 실험을 해보았다. 원래 실험하던 코드가 본인에게 조금 더 편리한 것 같아 가져와서 조금 고쳐보았다. 0과 1 프로세스만 yield를 시켜본 결과, 예상대로 0과 1 프로세스는 후순위로 밀려 출력도 나중에 나왔고, 다른 프로세스가 mlfq를 도는 것을 방해하지도 않았다. 아래있는 사진은 짝수 프로세스만 계속 yield를 호출했을 때의 모습이다.

Trouble shooting

1. time 변수의 추가

   디자인을 하고나니 한가지 고민이 들었다. 내가 구상한 프로세스는 만약 A 라는 프로세스와 B 라는 프로세스 두개가 레벨이 같아 스케줄러 내에서 스케줄링을 해줘야 한다면, A → B → A → B 이런식으로 일을 해야하는데, A → A → A(종료), B→B(종료) 이런식으로 일을하여 Round-Robin의 역할을 잘 해주지 못하는 스케줄러가 될

것이라는 것이다. 누가 먼저 들어왔는지 모르기 때문에 레벨이 같은 프로세스들이 있다면 무조건 ptable 기준으로 앞에있는 프로세스가 실행되게 될것이라는 것이다. 이를 어떻게 하면 해결해 줄 수 있을까를 고민하다가 탄생한 변수가 time 이었다. time 변수는 proc 구조체에다가 선언을 하였고, 해당 변수는 처음에는 process가 생성되었을 때의 global ticks를 저장해준다. 그리고 그 이후에는 context switching이 일어나기 전에 그 때의 global ticks로 초기화 해준다. 즉, 큐에 들어갈 때의 시간 정도라고 생각하면 된다. 이렇게 time변수를 설정하고 스케줄링할 때 같은 레벨 프로세스의 경우 time 변수가 작은 프로세스부터 실행하게 한다면 Round Robin을 구현할 수 있다! 그리고 scheduler lock이 발생하고 lock이 풀리게 된다면 무조건 L0 큐의 가장 앞으로 이동해야하므로 time 변수를 예외적으로 -1 이라고 설정을 했다. 그렇다면 다른 L0 프로세스의 프로세스들 보다 time이 무조건 작기 때문에 무조건 먼저 실행이 된다.

2. print 문제

   print를 어느 위치에서 하느냐에 따라 찍히는 정보가 다른 것에서 많은 고민이 들었다. 매 tick마다 print하게 하는 것이 overhead가 커서 print가 꼬일 수 있다는 생각도 많이 해보았고 print를 찍는 위치의 문제도 있을 수 있다는 생각을 했다. context switching 기준으로 switching 하기 직전, switching 중간, switching 직후 이렇게 종류별로 찍어보았고 time interrupt에서도 yield직전에 찍어 보았는데, 위치에 따라 생각보다 다른 print 결과가 발생했다. 내가 신뢰하기로 한 정보는 정확하게 running_p가 state가 4, 즉 RUNNING으로 설정되어서 실행되기 직전에 print하는 것이 실행될 process를 정확하게 찍어주는 것이라고 생각해서 그렇게 설정했는데, 확실하게 왜 위치마다 달라지는 지는 해석하지 못하였다.