관찰

기존 Round-Robin 스케줄러

기본적으로 구현된 `Round-Robin` 스케줄러의 메커니즘을 먼저 살펴보자.

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run.
//  - swtch to start running that process.
//  - eventually that process transfers control
//    via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

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

    int found = 0;
    for(p = proc; p < &proc[NPROC]; p++) { // 순회하면서 실행 가능한 프로세스를 찾음
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;


        printf("CONTEXT SWITCH Process name: %s, pid: %d\n", p->name, p->pid);
        swtch(&c->context, &p->context); // context switch 발생

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
        found = 1;
      }
      release(&p->lock);
    }
    if(found == 0) { // 실행할 프로세스가 하나도 없음
      // nothing to run; stop running on this core until an interrupt.
      intr_on();
```

```
      asm volatile("wfi"); // wait for interrupt의 약자
    }
  }
}
```

기본적으로 구현된 `Round-Robin` 스케줄러는 매 `tick` 마다 타이머 인터럽트가 발생하면서 `context swtich` 된다. `context switch` 가 되는 것을 확인하기 위해 `context switch` 되는 다음 프로세스의 정보를 출력하는 코드를 넣었다. 스케줄러가 올바르게 동작하고 있는 지 확인하기 위해서는 `cpu bound job` 을 수행하는 프로그램을 하나 만들고 정보를 출력하며 확인해보면 된다. 우선 `/user/schedtest.c` 라는 유저 프로그램을 만들고 아래와 같이 구현해보자. 유저 프로그램은 `Makefile` 의 `UPROGS` 변수에 등록하여 올바르게 실행될 수 있도록 한다.

```
#include "kernel/types.h"
#include "user/user.h"

// 그냥 for문 계속 도는 cpu bound job
void busy_work(void) {
  int i, j;
  for(j = 0; j < 100000000; j++){
    for(i = 0; i < 100000000; i++){
      ;
    }
  }
}

int main(void) {
  int i, pid;
  int iterations = 50;  // 각 프로세스가 실행할 횟수

  // 3개의 자식 프로세스 생성
  for(i = 0; i < 3; i++){
    pid = fork();
    if(pid < 0){
      printf("fork failed\n");
      exit(1);
    }
    if(pid == 0) { // 자식 프로세스
      for(i = 0; i < iterations; i++){
        busy_work();
        printf("Child process, pid: %d, iteration: %d\n", getpid(), i);
      }
      exit(0);
    }
  }

  // 부모 프로세스도 iterations 번 busy_work() 실행
  for(i = 0; i < iterations; i++){
    busy_work();
    printf("Parent process, pid: %d, iteration: %d\n", getpid(), i);
  }
```

```
  // 자식 프로세스가 모두 종료될 때까지 기다림
  for(i = 0; i < 3; i++){
    wait(0);
  }

  exit(0);
}
```

해당 프로그램은 부모 프로세스가 `fork`를 통해 자식 프로세스를 세 번 만들고 매 `iteration`마다 `cpu bound job`인 `busy_work`를 수행한다. 어느 프로세스가 몇 번째 `iteration`을 수행 중인지 확인하기 위해 매 `busy_work`를 수행한 뒤에 정보를 출력해준다. 아래는 `schedtest`유저 프로그램을 실행한 결과이다.

```
$ schedtest
CONTEXT SWITCH Process name: sh, pid: 2
CONTEXT SWITCH Process name: sh, pid: 3
CONTEXT SWITCH Process name: sh, pid: 3
CONTEXT SWITCH Process name: sh, pid: 3
CONTEXT SWITCH Process name: sh, pid: 3
CONTEXT SWITCH Process name: sh, pid: 3
ParentCONTEXT SWITCH Process name: schedtest, pid: 4
Child process, pid: 4, iteration: 0
Child process, pid: 4, iteration: 1
...
Child process, pid: 4, iteration: 8
Child process, pid: 4, iteration: 9
ChiCONTEXT SWITCH Process name: schedtest, pid: 5
Child process, pid: 5, iteration: 0
Child process, pid: 5, iteration: 1
...
Child process, pid: 5, iteration: 21
Child process, pid: 5, iteration: 22
ChCONTEXT SWITCH Process name: schedtest, pid: 6
Child process, pid: 6, iteration: 0
Child process, pid: 6, iteration: 1
...
Child process, pid: 6, iteration: 26
Child process, pid: 6, iteration: 27
Child processCONTEXT SWITCH Process name: schedtest, pid: 3
 process, pid: 3, iteration: 0
Parent process, pid: 3, iteration: 1
Parent process, pid: 3, iteration: 2
...
Parent process, pid: 3, iteration: 25
Parent process, pid: 3, iteration: 26
Parent process, pid: 3, iteration: CONTEXT SWITCH Process name: schedtest, pid: 4
ld process, pid: 4, iteration: 10
Child process, pid: 4, iteration: 11
Child process, pid: 4, iteration: 12
...
Child process, pid: 4, iteration: 35
```

```
Child process, pid: 4, iteration: 36
Child process, pid: 4, iteratCONTEXT SWITCH Process name: schedtest, pid: 5
ild process, pid: 5, iteration: 23
Child process, pid: 5, iteration: 24
Child process, pid: 5, iteration: 25
...
Child process, pid: 5, iteration: 48
Child process, pid: 5, iteration: 49
CONTEXT SWITCH Process name: schedtest, pid: 6
, pid: 6, iteration: 28
Child process, pid:CONTEXT SWITCH Process name: schedtest, pid: 3
27
Parent process, pid: 3, iteration: 28
Parent process, pid: 3, iteration: 29
...
Parent process, pid: 3, iteration: 48
Parent process, pid: 3, iteration: 49
CONTEXT SWITCH Process name: schedtest, pid: 4
ion: 37
Child process, pid: 4, iteration: 38
Child process, pid: 4, iteration: 39
Child process, pid: 4, iteration: 40
Child process, pid: 4, iteration: 41
Child process, pid: 4,CONTEXT SWITCH Process name: schedtest, pid: 6
 6, iteration: 29
Child process, pid: 6, iteration: 30
Child process, pid: 6, iteration: 31
...
Child process, pid: 6, iteration: 48
Child process, pid: 6, iteration: 49
CONTEXT SWITCH Process name: schedtest, pid: 3
CONTEXT SWITCH Process name: schedtest, pid: 4
 iteration: 42
Child process, pid: 4, iteration: 43
Child process, pid: 4, iteration: 44
Child process, pid: 4, iteration: 45
Child process, pid: 4, iteration: 46
Child process, pCONTEXT SWITCH Process name: schedtest, pid: 4
id: 4, iteration: 47
Child process, pid: 4, iteration: 48
Child process, pid: 4, iteration: 49
CONTEXT SWITCH Process name: schedtest, pid: 3
CONTEXT SWITCH Process name: sh, pid: 2
```

출력 도중에 `context switch`가 발생하는 경우들 때문에 실행 결과가 한 눈에 안들어올 수 있다. 요약하면 아래와 같다. 부모 프로세스의 `pid`는 3이고 만들어진 세 자식 프로세스의 `pid`는 차례대로 4, 5, 6이다.

```
출력이 온전히 끝난 기준으로 iteration이라 가정
CONTEXT SWITCH Process name: schedtest, pid: 4 -> pid4로 CONTEXT SWITCH
pid: 4, iteration: 0 ~ 9
CONTEXT SWITCH Process name: schedtest, pid: 5 -> pid5로 CONTEXT SWITCH
```

```
pid: 5, iteration: 0 ~ 22
CONTEXT SWITCH Process name: schedtest, pid: 6 -> pid6으로 CONTEXT SWITCH
pid: 6, iteration: 0 ~ 27
CONTEXT SWITCH Process name: schedtest, pid: 3 -> pid3으로 CONTEXT SWITCH
pid: 3, iteration: 0 ~ 26
CONTEXT SWITCH Process name: schedtest, pid: 4 -> pid4로 CONTEXT SWITCH
pid: 4, iteration: 10 ~ 36
CONTEXT SWITCH Process name: schedtest, pid: 5 -> pid5로 CONTEXT SWITCH
pid: 5, iteration: 23 ~ 49 끝! 자식이 먼저 끝나서 ZOMBIE 프로세스가 됨
CONTEXT SWITCH Process name: schedtest, pid: 6 -> pid6으로 CONTEXT SWITCH
pid: 6, iteration: 28
CONTEXT SWITCH Process name: schedtest, pid: 3 -> pid3으로 CONTEXT SWITCH
pid: 3, iteration: 27 ~ 49 끝! wait를 통해 5번 자식 회수, 부모는 나머지 자식들을 기다리며 SLEEPING 상태
가 됨
CONTEXT SWITCH Process name: schedtest, pid: 4 -> pid4로 CONTEXT SWITCH
pid: 4, iteration: 37 ~ 41
CONTEXT SWITCH Process name: schedtest, pid: 6 -> pid6으로 CONTEXT SWITCH
pid: 6, iteration: 29 ~ 49 끝! 자식 회수를 위해 wakeup 호출을 통해 부모 깨우고 부모로 CONTEXT SWITCH
발생
CONTEXT SWITCH Process name: schedtest, pid: 3 -> pid3으로 CONTEXT SWITCH
CONTEXT SWITCH Process name: schedtest, pid: 4 -> pid4로 CONTEXT SWITCH
pid: 4, iteration: 42 ~ 46
CONTEXT SWITCH Process name: schedtest, pid: 4 -> pid4로 CONTEXT SWITCH
pid: 4, iteration: 47 ~ 49 끝! 자식 회수를 위해 wakeup 호출을 통해 부모 깨우고 부모로 CONTEXT SWITCH
발생
CONTEXT SWITCH Process name: schedtest, pid: 3 -> pid3으로 CONTEXT SWITCH
CONTEXT SWITCH Process name: sh, pid: 2 -> 프로그램이 종료되며, 부모인 shell을 깨우고 CONTEXT
SWITCH
```

`Round-Robin`답게 프로세스가 공평하게 3, 4, 5, 6 순으로 `context switch`되는 것을 확인해볼 수 있다.

타이머 인터럽트 실행 메커니즘

타이머 인터럽트는 `CLINT`(Core Local Interrupt)라는 하드웨어 모듈에서 관리된다. `CLINT`는 두 개의 주요한 레지스터가 존재하는데 `mtime`와 `mtimecmp`이 존재한다. `mtime`은 현재 시간 정보를 갖고 있으며 `mtimecmp`는 다음 타이머 인터럽트가 발생하는 시간 정보를 갖고 있다. 우리는 `w_stimecmp`함수를 통해 `mtimcmp`레지스터에 다음 인터럽트가 발생하는 시점을 기록함으로써 타이머 인터럽트를 발생시킨다.

`trap.c`의 `clockintr`함수를 살펴보면 `w_stimecmp`가 구현되어 있는 것을 살펴볼 수 있다. (clockintr부분은 나중에 구현할 때 코드가 일부 수정됩니다. wiki 나중에 설명이 있습니다.)

```
void
clockintr()
{
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;                  // 글로벌 틱 카운터
    wakeup(&ticks);
    release(&tickslock);
  }
```

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

타이머 인터럽트가 발생하면 현재 모드가 `유저모드`인지 `커널모드`인지에 따라 제어가 다르게 넘어간다. `커널모드`에서의 타이머 인터럽트는 유저 프로그램 내에서 시스템 콜 호출로 인해 커널로 제어가 넘어간 상태에서 호출된 타이머 인터럽트이다.

`유저모드`에서 타이머 인터럽트가 발생하는 경우는 `trampoline.S`에서 `usertrap`으로 제어가 넘어간다. 여기서 `which_dev`라는 변수가 2로 설정되며 `yield`가 호출된다.

```
void
usertrap(void)
{
  int which_dev = 0; //디바이스 인터럽트 발생 시 어떤 디바이스인지 저장하는 변수

  ...
  else if((which_dev = devintr()) != 0){ // 0이면 unrecognized, 1이면 other device, 2면
timer interrupt
    // ok
  }
  ...

  // give up the CPU if this is a timer interrupt.
  // FCFS 는 timer interrupt가 발생했을 때 context switch가 발생하면 안됨
  if(which_dev == 2 && sched_mode != FCFS_MODE)
    yield();

  usertrapret();
}
```

`커널모드`에서 타이머 인터럽트가 발생하는 경우는 `kernelvec.S`에서 `kerneltrap`으로 제어가 넘어간다. 여기도 마찬가지로 `which_dev`라는 변수가 2로 설정되고 `yield`가 호출된다.

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

  ...
  // give up the CPU if this is a timer interrupt.
  // FCFS 는 timer interrupt가 발생했을 때 context switch가 발생하면 안됨
  if(which_dev == 2 && myproc() != 0 && sched_mode != FCFS_MODE)
    yield();

  ...

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

```
  w_sstatus(sstatus);
}
```

`yield`는 함수가 호출되면 실행 중이던 프로세스를 `RUNNABLE` 로 만들고 `sched`함수를 호출하여 스케줄러로 `context swtich`를 진행한다.

컨텍스트 스위치 매커니즘

`sched`함수는 `swtch.S`에 정의한 `swtch`함수를 통해 `context switch` 를 실행한다. `&p->context`는 이전 프로세스의 context이고 `&mycpu()->context`는 스케줄러의 context이다.

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();
  ...
  swtch(&p->context, &mycpu()->context);
  mycpu()->intena = intena;
}
```

참고로 스케줄러를 실행할 때 필요한 레지스터를 따로 `context`로 관리한다.

```
// Per-CPU state.
struct cpu {
  struct proc *proc;          // The process running on this cpu, or null.
  struct context context;     // swtch() here to enter scheduler().
  int noff;                   // Depth of push_off() nesting.
  int intena;                 // Were interrupts enabled before push_off()?
};
```

제어가 스케줄러로 넘어가면 스케줄러는 `swtch(&c->context, &p->context);`를 통해 현재 스케줄러의 문맥을 저장하고 다음 실행할 프로세스의 문맥으로 전환한다.

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

  c->proc = 0;
  for(;;){
    ...
    for(p = proc; p < &proc[NPROC]; p++) { // 순회하면서 실행 가능한 프로세스를 찾음
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        ...
        swtch(&c->context, &p->context);
        ...
```

```
      }
      release(&p->lock);
    }
    ...
  }
}
```

정리하자면 `yield`가 호출되면 `yield`의 `sched`가 `swtch`를 통해 `scheduler` 문맥으로 전환하고, `scheduler`는 새로운 프로세스를 찾은 다음 `swtch`를 통해 스케줄러의 문맥에서 새로 실행할 프로세스의 문맥으로 전환하고 이 과정이 반복된다.

xv6에서의 락에 대한 고찰

xv6에서 락을 얻고 푸는 과정을 살펴보자

락을 얻는 경우 `push_off()`를 통해 인터럽트를 끈다.

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
void
acquire(struct spinlock *lk)
{
  push_off(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // On RISC-V, sync_lock_test_and_set turns into an atomic swap:
  //   a5 = 1
  //   s1 = &lk->locked
  //   amoswap.w.aq a5, a5, (s1)
  while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen strictly after the lock is acquired.
  // On RISC-V, this emits a fence instruction.
  __sync_synchronize();

  // Record info about lock acquisition for holding() and debugging.
  lk->cpu = mycpu();
}
```

락을 푸는 경우 `pop_off()`를 통해 인터럽트를 다시 킨다.

```
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");
```

```
  lk->cpu = 0;

  // Tell the C compiler and the CPU to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other CPUs before the lock is released,
  // and that loads in the critical section occur strictly before
  // the lock is released.
  // On RISC-V, this emits a fence instruction.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code doesn't use a C assignment, since the C standard
  // implies that an assignment might be implemented with
  // multiple store instructions.
  // On RISC-V, sync_lock_release turns into an atomic swap:
  //   s1 = &lk->locked
  //   amoswap.w zero, zero, (s1)
  __sync_lock_release(&lk->locked);

  pop_off();
}
```

즉, **"xv6에서는 `critical section`을 실행하는 동안 cpu선점이 발생하지 않는다"**

xv6 부팅 과정에서 프로세스들의 상태에 대한 고찰

부팅 후 xv6가 처음 실행되면 커널의 `main()`이 호출되어 각종 서브시스템이 초기화되고, 그중 `procinit()`이 프로세스 테이블을 초기화한다. `main()`에서 `userinit()`이 호출되면 최초의 프로세스인 `initcode` 프로세스를 만들어 내고 `RUNNABLE` 상태로 만들어준다. 스케줄러가 `initcode`를 실행하고 상태를 `RUNNING`으로 만들며, `exec("/init", argv)`를 실행한다. 아래는 `initcode.S` 파일이 내용이다.

```
# Initial process that execs /init.
# This code runs in user space.

#include "syscall.h"

# exec(init, argv)
.globl start
start:
        la a0, init
        la a1, argv
        li a7, SYS_exec
        ecall

# for(;;) exit();
exit:
        li a7, SYS_exit
        ecall
        jal exit
```

```
# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .quad init
  .quad 0
```

`exec("/init", argv)`이 호출된 이후, `initcode`프로세스는 이제 `init`프로세스가 된다. `init` 프로세스는 `fork()`를 통해 자식 프로세스를 생성하고 `exec("sh", argv)`를 통해 자식을 `sh`프로세스로 만든다. `init`프로세스는 `sh`프로세스가 종료되길 기다리며 `SLEEPING`상태로 들어간다.

아래는 `init.c`의 일부이다.

```
...
int
main(void)
{
  int pid, wpid;

  if(open("console", O_RDWR) < 0){
    mknod("console", CONSOLE, 0);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr

  for(;;){
    printf("init: starting sh\n");
    pid = fork();
    ...
    if(pid == 0){
      exec("sh", argv);
      ...
    }

    for(;;){
      wpid = wait((int *) 0);
      ...
    }
  }
}
```

`sh`프로세스는 이제 사용자의 입력을 기다리며 `SLEEPING`상태로 들어가고 사용자의 명령어가 들어오면 `fork()`와 `exec()`을 통해 사용자의 명령을 실행하게 된다.

SYSTEM CALL

SYSTEM CALL DESIGN

목표: yield, getlev, setpriority, mlfqmode, fcfsmode 시스템 콜들을 올바르게 잘 구현하자

yield - 실행 중이던 현재 프로세스를 RUNNING -> RUNNABLE로 바꾸고 sched를 호출하여 CONTEXT SWITCH 발생

getlev - 현재 프로세스의 level을 반환, FCFS모드면 99 반환

setpriority - 프로세스의 id와 바꿀 priority를 입력받고 해당 프로세스의 priority를 입력받은 값으로 바꿈

fcfsmode - 현재 모드가 fcfs라면 -1을 반환, mlfq라면 sched_mode를 fcfs로 바꾸고 각 프로세스들의 priority, level, time quantum 모두 -1로 설정 후 0 반환.

mlfqmode - 현재 모드가 mlfq라면 -1을 반환, fcfs라면 sched_mode를 mlfq로 바꾸고 각 프로세스들의 priority를 3, level을 0, time quantum을 0으로 설정 후 0 반환.

위 정의에 맞게 구현하는 것이 목표이다.

SYSTEM CALL IMPLEMENTATION

가장 먼저 `usys.pl` 에 해당하는 시스템 콜의 어셈블리 스텁을 생성하게 하는 `entry서브루틴` 을 추가해주자.

```
...
sub entry {
    my $name = shift;
    print ".global $name\n";
    print "${name}:\n";
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}
...
entry("yield");
entry("getlev");
entry("setpriority");
entry("mlfqmode");
entry("fcfsmode");
```

`defs.h` 에 우리가 구현할 시스템 콜 선언을 해준다. `yield` 는 이미 선언되어있다.

```
void            yield(void);
...
int             getlev(void);       //getlev
int             setpriority(int pid, int priority); //setpriority
int             fcfsmode(void);     //fcfsmode
int             mlfqmode(void);     //mlfqmode
```

`proc.c` 에 시스템 콜들을 조건에 맞게 구현해주자.

`yield`는 `FCFS_MODE`에서는 자발적으로만 호출되어야한다(fcfs는 non-preemptive기 때문). `xv6` 내에서 `yield`가 호출되는 케이스는 1. `유저/커널 모드에서의 타이머 인터럽트` 2. `유저 프로그램 내에서 프로세스의 자발적인 yield` 두 종류밖에 존재하지 않는다. `yield`는 현재 `RUNNING` 중이던 프로세스의 상태를 `RUNNABLE` 바꾸고 컨텍스트 스위칭을 위해 `sched()`를 호출해준다. `fcfs`에서 발생하는 자발적인 `yield`는 어차피 방금까지 실행하던 프로세스가 생성 시간이 가장 이른 프로세스였을 것이기 때문에 해당 프로세스를 재개한다. `fcfs`에서의 자발적 `yield`는 멀티코어 환경에서나 유효하다.

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  if(sched_mode == FCFS_MODE){ // fcfs 모드에서 자발적 yield가 호출되어도 방금까지 실행중이던 프로세스가 가장 생성시간이 빨랐을 것이기 때문에 해당 프로세스를 다시 재개
    return;
  }
  struct proc *p = myproc(); // 현재 프로세스
  acquire(&p->lock);
  p->state = RUNNABLE; // 현재 프로세스 RUNNABLE 설정
  sched(); //context switching
  release(&p->lock);
}
```

`getlev`는 현재 프로세스가 몇 번째 레벨의 큐에 속해있는지 반환해준다. `fcfs`면 99를 반환한다.

```
int
getlev(void)
{
  if(sched_mode == FCFS_MODE) // FCFS모드면 99 반환
    return 99;
  struct proc *p = myproc();
  return p->level;
}
```

`setpriority`는 해당 프로세스의 우선순위를 바꿔준다. 프로세스의 우선순위는 `mlfq` 큐 `L2`에서만 유의미하다.

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

  // priority 값이 0~3 범위를 벗어나면 -2 반환.
  if(priority < 0 || priority > 3)
    return -2;

  for(p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);
    if(p->pid == pid){
      p->priority = priority;
      release(&p->lock);
      return 0; // 성공 시 0 반환
    }
```

```
    release(&p->lock);
  }
  return -1; // pid에 대응되는 프로세스가 존재하지 않음, -1 반환
}
```

`fcfsmode`는 스케줄러 모드를 `fcfs`로 바꿔준다. `xv6`가 처음에 부팅될 때 스케줄러는 `fcfs`로 실행된다. `mlfq`에서 `fcfs`로 모드 스위치가 일어나면 `mlfq`와 `fcfs`의 큐를 초기화해주고 프로세스 테이블을 순회하며 `RUNNABLE`한 프로세스를 `fcfs` 큐에 삽입해준다.

왜 mlfq 큐에 있는 프로세스들을 다 pop한 뒤 fcfs로 push하지 않는가?

만약 `fcfsmode`를 그렇게 구현하게 된다면 스케줄링 과정에서 한 프로세스가 `pop`되고 `running`중에 `mode switch`가 일어났다고 가정했을 때 해당 프로세스는 큐에 다시 `push`되지 못한 채 덩그러니 `RUNNABLE`상태로 어느 큐에도 속하지 않는 일이 발생해버린다. 아래와 같이 구현하면 프로세스 테이블이 어차피 `pid(생성시간)` 순으로 정렬되어 있어 구현도 간단해진다. +) `xv6`에서는 `pid`의 재사용이 일어나지 않기 때문에 `pid`가 곧 생성시간과 관련된다.

추가로 `mlfq`를 실행중이던 프로세스 자체는 `RUNNING`상태 였으므로 `fcfsmode`가 호출되면 `RUNNABLE`로 바꿔주자.

```
int
fcfsmode(void)
{
  struct proc *p;

  // 모드 전환 중에는 다른 스케줄링이 일어나지 않도록 보호
  acquire(&mode_lock);

  if(sched_mode == FCFS_MODE){
    printf("already in FCFS mode.\n");
    release(&mode_lock);
    return -1;
  }

  // mlfq 초기화
  mlfq_queue_init();
  // fcfs 초기화
  fcfs_queue_init();

  sched_mode = FCFS_MODE;
  acquire(&tickslock);
    ticks = 0;
  release(&tickslock);

  // 이미 프로세스 테이블 상에서 pid순(생성시간 순)으로 되어있음
  for(p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);
    if(p->state == RUNNABLE || p->state == RUNNING){
      p->state        = RUNNABLE;
      p->priority     = -1;
      p->level        = -1;
      p->time_quantum = -1;
```

```
      fcfs_queue_push(p);
    }
    release(&p->lock);
  }

  release(&mode_lock);
  return 0;
}
```

`mlfqmode`는 스케줄러 모드를 `mlfq`로 바꿔준다. `fcfs`에서 `mlfq`로 모드 스위치가 일어나면 프로세스들을 `fcfs`큐에서 `mlfq` 의 `L0` 큐로 이동시켜준다. `global tick` 값은 0으로 초기화해주고, 프로세스들의 `priority = 3`, `level = 0`, `time_quantum = 0`으로 설정해준다.

추가로 `fcfs`를 실행중이던 프로세스 자체는 `RUNNING`상태 였으므로 `mlfqmode`가 호출되면 `RUNNABLE`로 바꿔주자.

```
int
mlfqmode(void)
{
  struct proc *p;

  // 모드 스위칭 중에는 다른 어떤 프로세스도 진행되지 않도록 atomic하게 만들었다.
  acquire(&mode_lock);

  // 이미 MLFQ 모드인지 확인
  if(sched_mode == MLFQ_MODE){
    printf("already in MLFQ mode.\n");
    release(&mode_lock);
    return -1;
  }

  // fcfs 큐 초기화
  fcfs_queue_init();
  // mlfq 큐 초기화
  mlfq_queue_init();

  // MLFQ mode로 설정
  sched_mode = MLFQ_MODE;

  // global tick값 초기화
  acquire(&tickslock);
  ticks = 0;
  release(&tickslock);

  // fcfs에서의 프로세스들을 mlfq의 L0 큐로 이동
  for(p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);
    if(p->state == RUNNABLE || p->state == RUNNING){
      p->state        = RUNNABLE;
      p->priority     = 3;    // 최고 우선순위
      p->level        = 0;    // 레벨 0
      p->time_quantum = 0;    // 소비 시간 초기화
```

```
      mlfq_queue_push(0, p);
    }
    release(&p->lock);
  }

  release(&mode_lock);
  return 0;
}
```

`Mode switch` 과정은 `mode_lock`을 통해 `atomic`하게 만들어 `inconsistency`를 방지하였다. 추후에 설명할 `priority boost`도 `boost_lock`을 통해 `atomic`하게 만든 것을 확인할 수 있다.

```
struct spinlock mode_lock; // mode lock
struct spinlock boost_lock; // priority boost lock
```

`sysproc.c`에 우리가 구현한 시스템 콜들의 `wrapper function`들을 만들어주자.

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

uint64
sys_getlev(void)
{
  return getlev();
}

uint64
sys_setpriority(void)
{
  int pid;
  int priority;

  // 첫 번째 인자 pid, 두 번째 인자 priority
  argint(0, &pid);
  argint(1, &priority);

  return setpriority(pid, priority);
}

uint64
sys_fcfsmode(void)
{
  return fcfsmode();
}

uint64
sys_mlfqmode(void)
```

```
{
  return mlfqmode();
}
```

`syscall.h`에 시스템 콜 번호들을 부여해주자.

```
...
#define SYS_yield  22
#define SYS_getlev 23
#define SYS_setpriority 24
#define SYS_mlfqmode 25
#define SYS_fcfsmode 26
```

`syscall.c`에서 `syscalls`함수형 배열에 다음과 같이 추가해준다.

```
...
extern uint64 sys_yield(void);
extern uint64 sys_getlev(void);
extern uint64 sys_setpriority(void);
extern uint64 sys_mlfqmode(void);
extern uint64 sys_fcfsmode(void);

// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
static uint64 (*syscalls[])(void) = {
...
[SYS_yield]   sys_yield,
[SYS_getlev]  sys_getlev,
[SYS_setpriority]  sys_setpriority,
[SYS_mlfqmode]  sys_mlfqmode,
[SYS_fcfsmode]  sys_fcfsmode
};

void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // Use num to lookup the system call function for num, call it,
    // and store its return value in p->trapframe->a0
    p->trapframe->a0 = syscalls[num]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

여기는 이전 `assignment` 내용이므로 구체적인 설명은 생략하겠습니다.

FCFS

FCFS DESIGN

목표: FCFS 스케줄러를 구현해보자

- FCFS는 non-preemptive 스케줄링 방식이므로 타이머 인터럽트로 인한 yield를 막아야한다.
- FCFS 큐 구조체와 큐 관련 함수들을 정의해야한다. 큐에 `push` 또는 `pop` 할 때 락을 걸지 않으면 `race condition` 이 발생한다.
- 기존에 구현된 `Round-Robin` 스케줄러는 프로세스 테이블을 순회하며 `Runnable` 한 프로세스를 찾았지만 FCFS는 `Ready queue` 에 존재하는 프로세스를 `pop` 하고 스케줄링을 하므로 **"프로세스들이 생성될 때마다 `Ready queue` 에 넣어주는 작업"**이 필요하다. 맨 처음 스케줄링되는 모드가 `FCFS` 이므로 최초의 프로세스도 큐에 넣어줘야 한다.

FCFS IMPLEMENTATION

`FCFS(First Come First Served)` 는 말 그대로 먼저 `ready queue` 에 도착한 프로세스가 먼저 처리되어야 한다는 규칙으로 동작하는 스케줄러이다. `Round-Robin` 과는 다르게 `non-preemptive` 스케줄러이기 때문에 `FCFS` 로 동작하는 도중에는 타이머 인터럽트로 인한 `context switch` 가 발생하면 안된다.

```
// give up the CPU if this is a timer interrupt.
// FCFS 는 timer interrupt가 발생했을 때 context switch가 발생하면 안됨
if(which_dev == 2 && sched_mode != FCFS_MODE) {
  yield();
}
-----------------------------------------------------------------
// give up the CPU if this is a timer interrupt.
// FCFS 는 timer interrupt가 발생했을 때 context switch가 발생하면 안됨
if(which_dev == 2 && myproc() != 0 && sched_mode != FCFS_MODE){
  yield();
}
```

`FCFS` 를 구현하기 위해 필요한 큐 정보를 `defs.h` 에 선언하였다.

```
// fcfs 큐 락 초기화
void fcfs_queue_lock_init(void);
// 큐 초기화
void fcfs_queue_init(void);
// 큐가 비었는지 확인
int fcfs_queue_empty(void);
// 큐가 꽉 찼는지 확인
int fcfs_queue_full(void);
// 큐에 프로세스 삽입
void fcfs_queue_push(struct proc *p);
// 큐에서 프로세스 pop
struct proc *fcfs_queue_pop(void);
```

`proc.h`에 큐 구조체를 선언하였다. 구현의 편의성을 위해 `circular queue`로 구현하였다.

```
// fcfs queue 자료구조
struct fcfs_queue {
  struct proc *procs[NPROC];
  int head;  // 큐의 시작 인덱스
  int tail;  // 큐의 끝 인덱스
};
```

큐 관련 함수들은 `proc.c`에 정의하였다.

```
struct fcfs_queue fcfs_q; // FCFS QUEUE
struct spinlock fcfs_lock; //FCFS LOCK

// fcfs 큐 락 초기화, 처음 한 번만 호출되고 런타임에 호출되면 안됨
void
fcfs_queue_lock_init(void)
{
  initlock(&fcfs_lock, "fcfs_q");
  fcfs_queue_init();
}
// 큐 초기화
void
fcfs_queue_init(void)
{
  acquire(&fcfs_lock);
  fcfs_q.head = 0;
  fcfs_q.tail = 0;
  release(&fcfs_lock);
}
// 큐가 비었는지 확인
int
fcfs_queue_empty(void)
{
  return (fcfs_q.head == fcfs_q.tail);
}
// 큐가 꽉 찼는지 확인
int
fcfs_queue_full(void)
{
  return (((fcfs_q.tail + 1) % NPROC) == fcfs_q.head);
}
// 큐에 프로세스 삽입
void
fcfs_queue_push(struct proc *p)
{
  acquire(&fcfs_lock);
  if(fcfs_queue_full()){
    panic("fcfs_queue_push: ready queue full");
  }
  fcfs_q.procs[fcfs_q.tail] = p;
```

```
  fcfs_q.tail = (fcfs_q.tail + 1) % NPROC;
  release(&fcfs_lock);
}
// 큐에서 프로세스 pop
struct proc *
fcfs_queue_pop(void)
{
  struct proc *p = 0;
  acquire(&fcfs_lock);
  if(!fcfs_queue_empty()){
    p = fcfs_q.procs[fcfs_q.head];
    fcfs_q.head = (fcfs_q.head + 1) % NPROC;
  }
  release(&fcfs_lock);
  return p;
}
```

`Race condition`을 방지하기 위해 `fcfs`큐에 `push`하거나 `pop`할 때 락을 걸어줘야한다. `Fcfs` 큐에 `push`하는 상황에서의 `Race condition`의 예시는 다음과 같다. `tail` 값을 읽고 각 프로세스가 `tail`값을 업데이트하기 전에 `fcfs_q.procs[fcfs_q.tail] = p;`가 호출되면 프로세스가 같은 인덱스에 덮어 써지는 현상이 발생한다. `pop`도 마찬가지이므로 락을 잘 걸어주자.

```
void
fcfs_queue_push(struct proc *p)
{
  acquire(&fcfs_lock);
  if(fcfs_queue_full()){
    panic("fcfs_queue_push: ready queue full");
  }
  fcfs_q.procs[fcfs_q.tail] = p;
  fcfs_q.tail = (fcfs_q.tail + 1) % NPROC;
  release(&fcfs_lock);
}

1 PROCESS1  읽기: tail → 0
2 PROCESS2  읽기: tail → 0
3 PROCESS1  쓰기: fcfs_q.procs[0] = P1
4 PROCESS2  쓰기: fcfs_q.procs[0] = P2
5 PROCESS1  쓰기: tail = 1  tail=1
6 PROCESS2  쓰기: tail = 1  tail=1
-> P1이 사라지고 procs[0]=P2, 3과 4 순서가 바뀐다면 P2가 사라짐
```

`sched_mode`는 맨 처음에 `fcfs`로 되어있기 때문에 우리가 테스트 프로그램을 실행하기 직전까지**(쉘 프로그램이 실행되기까지)**의 프로세스들은 `fcfs_queue`에 넣어주어야 `initcode`과 `shell`프로세스가 `fcfs`스케줄러에 의해 실행이 될 수 있다(결국 테스트 프로그램은 쉘에서 입력하고 실행하기 때문). 프로세스들은 결국 하나의 프로세스로부터 `fork`를 통해 **복제**되거나 `exec`을 통해 **대체**되기 때문에

1. `initcode`프로세스에서의 큐 삽입 처리
2. `fork`에서 자식 프로세스 큐 삽입 처리

3. `fcfs_queue`는 `RUNNABLE`한 프로세스만 관리하므로 `SLEEPING`상태에서 `RUNNABLE`로 바뀌는 과정인 `wakeup`함수 내에서 큐 삽입 처리를 해주면 된다. 정확히는 `shell`에서 입력받은 프로그램은 `shell`로부터 `fork`된 자식 프로세스가 `exec` 호출로 인해 대체되는 것이다. `fcfs`스케줄러가 동작하기 전에 `kernel/main.c`에서 `fcfs_queue`를 큐의 락을 초기화 해주자.

```
void
main()
{
  if(cpuid() == 0){
    fcfs_queue_lock_init(); // fcfs queue init
    ...
  } else {
    ...
  }
  // 스케줄러
  scheduler();
}
```

맨 처음에는 `initcode`프로세스가 실행되므로 `/kernel/proc.c`에 있는 `userinit`함수의 `initcode`프로세스를 `fcfs_queue`에 넣어준다. `initcode`프로세스는 무조건 `fcfs`에서만 스케줄링되기 때문에 조건을 추가할 필요가 없다.

```
// Set up first user process.
void
userinit(void)
{
  p = allocproc(); // 최초의 프로세스 initcode
  ...
  p->state = RUNNABLE;
  fcfs_queue_push(p); // 프로세스가 RUNNABLE 상태로 바뀌었으므로 ready queue에 넣음.
  ...
}
```

그 다음 `shell`프로세스는 `init`프로세스에 `fork`를 통해 생성되므로 `fork`도 수정해준다.

```
// Create a new process, copying the parent.
// Sets up child kernel stack to return as if from fork() system call.
int
fork(void)
{
  ...
  np->state = RUNNABLE;
    // 자식이 RUNNABLE 상태가 되었으므로 ready queue에 넣음
  if(sched_mode == FCFS_MODE) {
    fcfs_queue_push(np);
  } else {
    mlfq_queue_push(np->level, np);
  }
  ...
}
```

`shell` 프로세스는 사용자의 입력을 기다리면서 `SLEEPING` 상태에서 대기한다. 따라서 `wakeup`을 호출함에 따른 큐 push 코드도 추가해야한다. `wakeup` 코드는 기존의 코드에서 완전히 수정한 것을 확인할 수 있다. 그 이유는 유저 프로그램이 실행되는 동안 쉘이 `SLEEPING` 상태로 전환하는데, 만약 유저 프로그램이 `mlfq mode`인 상태로 실행을 종료한다면 쉘이 `wakeup`됨에 따라 `mlfq_queue_push`가 호출되고, 쉘은 `level`이 -1이므로 스케줄링이 불가능해 `panic`이 호출된다. 따라서 유저 프로그램이 종료하면서 쉘을 `wakeup`시키는 경우에 한해 다시 `fcfs mode`로 바꿔주는 로직이 필요했다. **우리가 실행하는 xv6는 무한루프를 돌기 때문에 하나의 유저 프로그램이 실행이 끝났다고 해서 전역 변수가 다시 원래대로 돌아가지 않는다는 것을 간과하면 안된다.** 개인적으로 이번 프로젝트를 수행하며 가장 많은 시간을 고민했던 부분이다.

```
// Wake up all processes sleeping on chan.
// Must be called without any p->lock.
void
wakeup(void *chan)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    if(p == myproc())  // 자기 자신은 건너뛰고
      continue;

    acquire(&p->lock);
    if(p->state == SLEEPING && p->chan == chan) {
      // 쉘 프로세스만 FCFS 모드로 복원
      if (p->pid == 2) {
        acquire(&mode_lock);
        sched_mode = FCFS_MODE;
        release(&mode_lock);
      }

      p->state = RUNNABLE;
      if(sched_mode == FCFS_MODE){
        fcfs_queue_push(p);
      } else {
        mlfq_queue_push(p->level, p);
      }
    }
    release(&p->lock);
  }
}
```

구현된 `FCFS` 스케줄러는 아래와 같다.

1. fcfs 큐에서 프로세스 pop
2. 프로세스 상태를 RUNNING으로 전환 후 해당 프로세스로 context switch

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

```
    c->proc = 0;
    for (;;) {
      // deadlock 방지
      intr_on();

      if (sched_mode == FCFS_MODE) { // fcfs 모드의 경우
        if (!fcfs_queue_empty()) {
          p = fcfs_queue_pop();
          acquire(&p->lock);
          if (p->state == RUNNABLE) {
            p->state = RUNNING;
            c->proc = p;
            swtch(&c->context, &p->context); //실행
            c->proc = 0; // 끝
          }
          release(&p->lock);
        } else {
          intr_on();
          asm volatile("wfi");
        }

      } else { // mlfq 모드의 경우
        ...
      }
    }
  }
```

MLFQ

MLFQ DESIGN

목표: MLFQ 스케줄러를 구현해보자

- `MLFQ`는 `preemptive` 스케줄링 방식이므로 타이머 인터럽트로 인한 `yield`가 발생한다. 프로세스가 자발적으로도 `yield`를 호출하는 점도 생각하자.
- `mlfq`의 구조체 및 큐를 정의해야한다. 큐에 `push`또는 `pop`할 때 락을 걸지 않으면 `race condition`이 발생한다.
- `Priority boost`를 구현하자. `global tick`값을 바꿀 때 락을 걸고 해제해야한다.

MLFQ IMPLEMENTATION

`MLFQ(Multi-Level Feedback Queue)`스케줄러는 `FCFS`와는 달리 `preemptive` 방식으로 동작한다. MLFQ에서는 타이머 인터럽트가 주기적으로 발생하여 `context switch`를 발생시킨다. `mlfq`는 총 3개의 레벨을 가진 큐가 존재하며, 0번째 큐부터 차례대로 순회하며 스케줄링 대상을 찾는다. 0번째와 1번째 큐는 `Round-Robin` 정책을 따르고 2번째 큐는 `priority scheduling`정책을 따른다. 만약 특정 프로세스가 가진 `time quantum`이 모두 소진된다면 다음 레벨로 `demotion`이 발생하고 `time quantum`은 초기화된다.

아래는`defs.h`에서 선언한`MLFQ`의 큐를 구성하는 선언 코드이다.

```
// mlfq 큐 락 초기화
void mlfq_queue_lock_init(void);
// 큐 초기화
void mlfq_queue_init(void);
// 큐가 비었는지 확인
int mlfq_queue_empty(int level);
// 큐가 꽉 찼는지 확인
int mlfq_queue_full(int level);
// 큐에 프로세스 삽입
void mlfq_queue_push(int level, struct proc *p);
// 큐에서 프로세스 pop
struct proc *mlfq_queue_pop(int level);
```

큐 구조체는 `proc.h`에 정의하였다.

```
// mlfq queue 자료구조
struct mlfq_queue {
  struct proc *procs[NPROC];
  int head;
  int tail;
};
```

큐에 대한 정의는 `proc.c`에 있다.

```
struct mlfq_queue mlfq_q[3]; // MLFQ QUEUE
struct spinlock mlfq_lock; // MLFQ LOCK

// mlfq 큐 락 초기화, 처음 한 번만 호출되고 런타임에 호출되면 안됨
void
mlfq_queue_lock_init()
{
  initlock(&mlfq_lock, "mlfq_q"); // mlfq queue 락 초기화
  mlfq_queue_init();
}
// 큐 초기화
void
mlfq_queue_init(void) // 주의!!! mlfq_queue_init은 딱 한 번만 실행되고 런타임 중에는 호출하면 안됨!!!
{
  initlock(&mlfq_lock, "mlfq_q"); // mlfq queue 락 초기화
  acquire(&mlfq_lock);
  for(int i = 0; i < 3; i++){
    mlfq_q[i].head = 0;
    mlfq_q[i].tail = 0;
  }
  release(&mlfq_lock);
}
// 큐가 비었는지 확인
int
mlfq_queue_empty(int level)
{
```

```
  return (mlfq_q[level].head == mlfq_q[level].tail);
}
// 큐가 꽉 찼는지 확인
int
mlfq_queue_full(int level)
{
  int next = (mlfq_q[level].tail + 1) % NPROC;
  return (next == mlfq_q[level].head);
}
// 큐에 프로세스 삽입
void
mlfq_queue_push(int level, struct proc *p)
{
  acquire(&mlfq_lock);
  if(mlfq_queue_full(level)){
    panic("mlfq_queue_push: ready queue full");
  }
  int next = (mlfq_q[level].tail + 1) % NPROC;
  mlfq_q[level].procs[mlfq_q[level].tail] = p;
  mlfq_q[level].tail = next;
  release(&mlfq_lock);
}
// 큐에서 프로세스 pop
struct proc *
mlfq_queue_pop(int level)
{
  struct proc *p = 0;
  acquire(&mlfq_lock);
  if (!mlfq_queue_empty(level)) {
    p = mlfq_q[level].procs[mlfq_q[level].head];
    mlfq_q[level].head = (mlfq_q[level].head + 1) % NPROC;
  }
  release(&mlfq_lock);
  return p;
}
```

가장 먼저 `main.c`의 메인함수에서 `mlfq`큐의 락을 초기화해주자.

```
// start() jumps here in supervisor mode on all CPUs.
void
main()
{
  if(cpuid() == 0){
    fcfs_queue_lock_init(); // fcfs queue init
    mlfq_queue_lock_init(); // mlfq queue init
    ...
  }

  // 스케줄러
  scheduler();
}
```

매 50틱마다 `priority boost`가 실행되어 프로세스의 `starvation` 을 막아줘야한다. `priority boost`도 모드 스위치할 때와 마찬가지로 큐가 꼬이지 않게 `atomic`하게 만들어주었다. `priority boost`는 프로세스 테이블을 순회하며 `RUNNABLE` 한 프로세스들(큐에 있던 프로세스들)과 `priorityboost`를 호출한`RUNNING`프로세스를 `L0`큐에 삽입해준다.

```
void
priorityboost(void)
{
  struct proc *p;

  acquire(&boost_lock); // atomic 하게 만들어 큐가 꼬이는 일이 발생하지 않게 하자.

  // FCFS 모드에서는 호출되면 안됨
  if(sched_mode == FCFS_MODE){
    printf("cannot call priority boost in FCFS mode\n");
    return;
  }

  // 직전에 부스트가 발생한 시간 갱신
  last_boost = ticks;

  // mlfq 큐 초기화
  mlfq_queue_init();

  for(p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);
    if(p->state == RUNNABLE || p->state==RUNNING) {
      p->state        = RUNNABLE;
      p->level        = 0;
      p->priority     = 3;
      p->time_quantum = 0;
      mlfq_queue_push(0, p);
    }
    release(&p->lock);
  }
  release(&boost_lock);
}
```

구현된 `MLFQ`스케줄러는 아래와 같다.

1. 먼저 틱 값을 확인하여 50틱이 지나면 `priority boost`를 실행해준다.
2. 가장 레벨이 낮은 큐부터 스케줄링 대상 프로세스를 탐색한다.
3. 실행 후 만약 프로세스의 타임퀀텀이 해당 큐의 타임퀀텀 이상이라면 다음 큐로 demotion시킨다. 만약 L2큐였다면 priority를 감소시켜준다. 프로세스가 큐의 타임 퀀텀만큼 실행되지 못한 채 `cpu`를 넘겨주면 해당 프로세스는 다시 원래 있던 큐로 들어간다.

```
void
```

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

  c->proc = 0;
  for (;;) {
    // deadlock 방지
    intr_on();

    if (sched_mode == FCFS_MODE) { // fcfs 모드의 경우
      ...
    } else { // mlfq 모드의 경우
      // priority boost
      if (ticks - last_boost >= 50) {
        priorityboost();
        last_boost = ticks;
      }

      // 우선순위 가장 높은 level0 큐부터 검색
      p = 0;
      for(int level = 0; level < 3; level++){
        if(!mlfq_queue_empty(level)){
          p = mlfq_queue_pop(level);
          break;
        }
      }

      if (p) {
        acquire(&p->lock);
        if (p->state == RUNNABLE) {

          // 실제로 실행
          p->state = RUNNING;
          c->proc = p;
          swtch(&c->context, &p->context); // 실행
          c->proc = 0; // 끝

          // 실행 후 demotion
          if (p->state == RUNNABLE) {
            if (p->level == 0) { // L0→L1
              if (p->time_quantum >= 1) {
                p->level++;
                p->time_quantum = 0;
              }
            } else if(p->level == 1){ // L1→L2
              if (p->time_quantum >= 3) {
                p->level++;
                p->time_quantum = 0;
              }
            } else {
              // L2: quantum 5 소진 시 priority 감소
              if (p->time_quantum >= 5) {
```

```
                p->time_quantum = 0;
                if (p->priority > 0)
                  p->priority--;
              }
            }
            // 다시 해당 레벨 큐로 push
            mlfq_queue_push(p->level, p);
          }
        }
        release(&p->lock);
      } else {
        // idle
        intr_on();
        asm volatile("wfi");
      }
    }
  }
}
```

프로세스가 실행되는 도중에 `fork()`를 통해 자식을 생성하면 자식의 필드를 초기화해줄 필요가 있다. 현재 실행 중인 모드에 따라 `priority`, `level`, `time_quantum`을 알맞게 초기화해주자.

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

  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == UNUSED) {
      goto found;
    } else {
      release(&p->lock);
    }
  }
  return 0;

found:
  p->pid = allocpid();
  p->state = USED;

  // 새 프로세스의 필드를 현재 모드에 맞게 초기화
  if (sched_mode == FCFS_MODE) {
    // fcfs 모드
    p->priority     = -1;
    p->level        = -1;
    p->time_quantum = -1;
  } else {
    // mlfq 모드면
    p->priority     = 3;
    p->level        = 0;
```

```
    p->time_quantum = 0;
  }

  ...
  return p;
}
```

추가로 프로세스의 타임 퀀텀 값을 증가시키기는 로직을 글로벌 틱이 증가되는 부분에서 동시에 이루어질 수 있도록 구현하였다.

```
void
clockintr()
{
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;                  // 글로벌 틱 카운터

    // mlfq에서 실행 중인 프로세스 time quantum 1증가
    struct proc *p = mycpu()->proc;
    if (p && sched_mode == MLFQ_MODE) {
      acquire(&p->lock);
      p->time_quantum++;
      release(&p->lock);
    }

    wakeup(&ticks);
    release(&tickslock);
 }
```

RESULT

아래는 제공된 테스트 코드이다. 파일은 `/user/testFromTA.c` 로 저장되어있다.

테스트 내용은 다음과 같다.

- `fcfsmode`에서 자식들을 4개 만들고 100000번씩 fcfs_count를 증가시킨다. (생성된 순서로 실행되어야 함)
- `mlfqmode`에서 자식들을 4개 만들고 100000번씩 돌면서 자신이 해당 큐에서 실행된 횟수(count[x])를 증가시킨다.

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

#define NUM_LOOP 100000
#define NUM_THREAD 4
#define MAX_LEVEL 3

int parent;
int fcfs_pids[NUM_THREAD];
int fcfs_count[100] = {0};

int fork_children() // 자식 4개 만듦, 본인 포함 총 5개
```

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

void exit_children() // 자식 종료, 부모는 자식 회수 대기
{
  if (getpid() != parent)
    exit(0);
  int status;
  while (wait(&status) != -1);
}

int main(int argc, char *argv[])
{
  int i, pid;
  int count[MAX_LEVEL] = {0};

  parent = getpid();

  printf("FCFS & MLFQ test start\n\n"); // 테스트 시작

  // [Test 1] FCFS test
  printf("[Test 1] FCFS Queue Execution Order\n");
  pid = fork_children(); // 자식 4개 생성

  if (pid != parent) // 자식의 경우
  {
    while(fcfs_count[pid] < NUM_LOOP) // 100000번 돌기
    {
      fcfs_count[pid]++;
    }

    printf("Process %d executed %d times\n", pid, fcfs_count[pid]); // fcfs이므로 먼저 생성된 순으로 끝남
  }
  exit_children(); // 자식 회수
  printf("[Test 1] FCFS Test Finished\n\n");

  // Switch to FCFS mode - should not be changed
  if(fcfsmode() == 0) printf("successfully changed to FCFS mode!\n");
  else printf("nothing has been changed\n"); // 이미 fcfsmode이므로 -1값이 반환되고 이 문장 출력됨

  // Switch to MLFQ mode
  if(mlfqmode() == 0) printf("successfully changed to MLFQ mode!\n"); // fcfs -> mlfq 모드로 전환됨
  else printf("nothing has been changed\n");
```

```
  // [Test 2] MLFQ test
  printf("\n[Test 2] MLFQ Scheduling\n");
  pid = fork_children(); // 자식 4개 생성

  if (pid != parent) // 자식의 경우
  {
    for (i = 0; i < NUM_LOOP; i++) // 100000번 돌기
    {
      int x = getlev(); // 현재 프로세스의 레벨
      if (x < 0 || x >= MAX_LEVEL) // 레벨 값이 이상한 경우
      {
        printf("Wrong level: %d\n", x);
        exit(1);
      }
      count[x]++; // 특정 레벨에서 실행된 횟수를 기록
    }

    printf("Process %d (MLFQ L0-L2 hit count):\n", pid);
    for (i = 0; i < MAX_LEVEL; i++)
      printf("L%d: %d\n", i, count[i]);
  }
  exit_children(); // 자식 수거

  printf("[Test 2] MLFQ Test Finished\n");
  printf("\nFCFS & MLFQ test completed!\n");
  exit(0);
}
```

실행 결과는 아래와 같다.

`fcfs`에서는 프로세스가 생성된 순서(4, 5, 6, 7)대로 실행되었다.

`mlfq`에서는 각 프로세스가 특정 큐에서 실행된 횟수가 출력되었다. 이때 L0, L1, L2큐에서 실행된 횟수의 합이 `100000`이어야 정상적으로 작동된 것이다.

```
$ testFromTA
FCFS & MLFQ test start

[Test 1] FCFS Queue Execution Order
Process 4 executed 100000 times
Process 5 executed 100000 times
Process 6 executed 100000 times
Process 7 executed 100000 times
[Test 1] FCFS Test Finished

already in FCFS mode.
nothing has been changed
successfully changed to MLFQ mode!

[Test 2] MLFQ Scheduling
```

```
Process 8 (MLFQ L0-L2 hit count):
L0: 5693
L1: 23916
L2: 70391
Process 10 (MLFQ L0-L2 hit count):
L0: 7959
L1: 24234
L2: 67807
Process 9 (MLFQ L0-L2 hit count):
L0: 10536
L1: 24301
L2: 65163
Process 11 (MLFQ L0-L2 hit count):
L0: 9946
L1: 23964
L2: 66090
[Test 2] MLFQ Test Finished

FCFS & MLFQ test completed!
```

직접 만든 테스트 코드는 `/user/schedtest_full.c`에 구현되어 있다. 조교님의 테스트 코드를 참고해서 구현하였다. 테스트 내용은 아래와 같다.

- **테스트 1** - `FCFS` 동작 확인하기
- **테스트 2** - `mode-switch` 직후 `getlev()` 값 확인
- **테스트 3** - `MLFQ` 동작 확인하기
- **테스트 4** - 자발적 `yield` 확인하기, 단 `mlfq`의 경우 아주 드물게 `timer interrupt`로 인한 `yield`가 사이에 발생하여 `demotion`이 생길 수도 있음.
- **테스트 5** - `setpriority` 테스트하기
- **테스트 6** - `priority boost` 확인, `busy loop`을 돌며 `getlev()`를 호출했을 때 level 2에 머무는 것이 아닌 0 또는 1 레벨이 다시 출력되는지를 통해 확인

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

#define NUM_LOOP    100000
#define NUM_THREAD  4
#define MAX_LEVEL   3

int parent;

// 1) 단순 fork/exit 헬퍼: 자식 NUM_THREAD개 생성
int
fork_children(void)
{
  int i, pid;
  for (i = 0; i < NUM_THREAD; i++) {
    if ((pid = fork()) == 0)
```

```
      return getpid();    // 자식: 자신 PID 반환
  }
  return parent;          // 부모: parent 반환
}

// 2) 자식 회수 헬퍼
void
exit_children(void)
{
  if (getpid() != parent)
    exit(0);              // 자식은 즉시 종료
  int status;
  while (wait(&status) != -1)
    ;                     // 부모는 모든 자식 회수
}

int
main(int argc, char *argv[])
{
  int pid, i;
  int hits[MAX_LEVEL];

  parent = getpid();
  printf("=== FCFS & MLFQ extended test start ===\n\n");

  // ------------------------------------------------------------
  // [Test 1] 기본 FCFS 동작 검증
  //   - non-preemptive: fork된 순서대로 각 자식이 NUM_LOOP회 실행
  // ------------------------------------------------------------
  printf("[1] FCFS 기본 동작\n");
  pid = fork_children();
  if (pid != parent) {
    // 자식: NUM_LOOP 회 루프
    int ctr = 0;
    while (ctr++ < NUM_LOOP) ;
    printf(" Child %d done busy loop\n", pid);
  }
  exit_children();
  printf(" => [1] FCFS 확인 완료\n\n");

  // ------------------------------------------------------------
  // [Test 2] mode-switch 직후 getlev 검증
  //   - fcfsmode() 후 getlev() == 99
  //   - mlfqmode() 후 getlev() == 0
  // ------------------------------------------------------------
  printf("[2] 모드 전환 직후 필드 초기화 확인\n");
  if (fcfsmode() != 0) printf("  fcfsmode: 이미 FCFS 모드\n");
  printf("   getlev() after fcfsmode() == %d  (기대: 99)\n", getlev());

  if (mlfqmode() != 0) printf("  mlfqmode: 이미 MLFQ 모드\n");
  printf("   getlev() after mlfqmode() == %d  (기대: 0)\n\n", getlev());
```

```
  // ------------------------------------------------------------
  // [Test 3] 기본 MLFQ 동작 검증
  //   - 각 레벨별 실행 횟수 누적
  // ------------------------------------------------------------
  printf("[3] MLFQ 기본 스케줄링 분포\n");
  pid = fork_children();
  if (pid != parent) {
    for (i = 0; i < MAX_LEVEL; i++) hits[i] = 0;
    for (i = 0; i < NUM_LOOP; i++) {
      int lvl = getlev();
      if (lvl < 0 || lvl >= MAX_LEVEL) {
        printf(" Wrong level: %d\n", lvl);
        exit(1);
      }
      hits[lvl]++;
    }
    printf(" Child %d level hits: ", pid);
    for (i = 0; i < MAX_LEVEL; i++)
      printf("L%d=%d ", i, hits[i]);
    printf("\n");
  }
  exit_children();
  printf(" => [3] MLFQ 분포 확인 완료\n\n");

  // ------------------------------------------------------------
  // [Test 4] 자발적 yield 테스트
  //   - FCFS 모드: yield() 호출해도 아무 변화 없음을 확인
  //   - MLFQ 모드: yield() 호출 시 즉시 재스케줄링 됨을 확인
  // ------------------------------------------------------------
  printf("[4] 자발적 yield 테스트\n");

  // -- FCFS 모드에서 yield --
  fcfsmode();
  pid = fork_children();
  if (pid != parent) {
    printf("  [FCFS] child %d calling yield() twice...\n", pid);
    yield();
    yield();
    printf("  [FCFS] still here after yield(), pid=%d\n", pid);
  }
  exit_children();

  // -- MLFQ 모드에서 yield, 이 경우 사이에 timer interrupt로 인한 yield가 발생하고 demotion이 일어날
수도 있다는 점.. --
  mlfqmode();
  pid = fork_children();
  if (pid != parent) {
    printf("  [MLFQ] child %d calling yield() twice...\n", pid);
    printf("    before getlev()=%d, pid=%d\n", getlev(), pid);
    yield();
    printf("    after 1st yield getlev()=%d, pid=%d\n", getlev(), pid);
    yield();
```

```
    printf("    after 2nd yield getlev()=%d, pid=%d\n", getlev(), pid);
  }
  exit_children();
  printf(" => [4] yield 동작 확인 완료\n\n");

  // ------------------------------------------------------------
  // [Test 5] setpriority 테스트
  //   - 정상 호출, 범위 벗어난 호출, 잘못된 PID 호출 검증
  // ------------------------------------------------------------
  printf("[5] setpriority 테스트\n");

  pid = fork();
  if (pid == 0) {
    // 자식은 잠깐 대기
    for (i = 0; i < 1000000; i++) ;
    printf("  child %d exiting\n", getpid());
    exit(0);
  }
  // 부모: 정상범위 설정
  if (setpriority(pid, 2) == 0)
    printf("  setpriority(%d,2) 성공\n", pid);
  else
    printf("  setpriority(%d,2) 실패\n", pid);
  // 범위 벗어난 priority
  printf("  setpriority(%d,-1) => %d (기대: -2)\n", pid, setpriority(pid, -1));
  // 잘못된 PID
  printf("  setpriority(9999,2) => %d (기대: -1)\n", setpriority(9999,2));
  wait(0);
  printf(" => [4] setpriority 확인 완료\n\n");

  // ------------------------------------------------------------
  // [6] priority boost 동작 확인
  //   - MLFQ 모드에서 일정 시간(틱) 경과 후 레벨이 다시 0으로 부스트 되는지 확인
  // ------------------------------------------------------------
  printf("[6] priority boost 테스트\n");
  mlfqmode();
  pid = fork_children();
  if (pid != parent) {
    // 짧은 busy loop를 반복하면서 priority boost가 발생하는지 확인
    for (i = 0; i < 200; i++) {
      for (int j = 0; j < 500000000; j++) ;
      if (i % 10 == 0)
        printf("  child %d at iteration %d, level=%d\n",
               getpid(), i, getlev());
    }
  }
  exit_children();
  printf(" => [6] priority boost 확인 완료\n\n");

  printf("=== 모든 테스트 완료 ===\n");
  exit(0);
}
```

아래는 실행 결과이다.

테스트1 - 네 자식(4,5,6,7)이 순서대로 `busy loop`를 돌고 종료

테스트2 - `fcfsmode()` 호출 후 `getlev() == 99`, `mlfqmode()` 호출 후 `getlev() == 0`

테스트3 - 네 자식이 대략 **L0:L1:L2 ≈ 5–15% : 20–25% : 60–70%** 비율로 잘 실행됨 -> `time quantum`과의 연관성

테스트4 -

- **FCFS 모드**: `yield()` 호출해도 여전히 같은 프로세스가 계속 실행됨.
- **MLFQ 모드**:
  - 호출 직후 즉시 스케줄러가 개입해 재스케줄링됨을 확인.
  - 다만, **한 케이스(pid=17)** 에서 2번째 `yield()` 뒤 레벨이 1로 올라간(=demotion) 현상이 보이는데 이것은 **timer interrupt 중간에** `time_quantum`이 채워져서 `demotion` 로직이 작동한 것임

테스트5 - 올바르게 실행 됨. 정상`pid`와 정상 `priority`도 검증 완료

테스트6 -

- `iteration` 20, 30, 50, 100, 150 등 시점마다 일부 프로세스가 **레벨 0으로 회귀**함
- 하지만 **`iteration`=10, 40 등**에서도 `demotion/boost`가 뒤섞여 보임
- 출력이 워낙 많아 실제 “50틱 단위” 부스트 타이밍을 정확히 짚어내긴 어려움 -> 일단 `priority boost`가 올바르게 잘 동작하는 것은 확인 가능.

```
$ schedtest_full
=== FCFS & MLFQ extended test start ===

[1] FCFS 기본 동작
 Child 4 done busy loop
 Child 5 done busy loop
 Child 6 done busy loop
 Child 7 done busy loop
 => [1] FCFS 확인 완료

[2] 모드 전환 직후 필드 초기화 확인
already in FCFS mode.
  fcfsmode: 이미 FCFS 모드
   getlev() after fcfsmode() == 99  (기대: 99)
   getlev() after mlfqmode() == 0  (기대: 0)

[3] MLFQ 기본 스케줄링 분포
 Child 8 level hits: L0=5182 L1=24216 L2=70602
 Child 10 level hits: L0=8140 L1=23231 L2=68629
 Child 9 level hits: L0=13941 L1=23758 L2=62301
 Child 11 level hits: L0=9648 L1=25340 L2=65012
 => [3] MLFQ 분포 확인 완료

[4] 자발적 yield 테스트
```

```
  [FCFS] child 12 calling yield() twice...
  [FCFS] still here after yield(), pid=12
  [FCFS] child 13 calling yield() twice...
  [FCFS] still here after yield(), pid=13
  [FCFS] child 14 calling yield() twice...
  [FCFS] still here after yield(), pid=14
  [FCFS] child 15 calling yield() twice...
  [FCFS] still here after yield(), pid=15
  [MLFQ] child 16 calling yield() twice...
    before getlev()=0, pid=16
  [MLFQ] child 17 calling yield() twice...
    before getlev()=0, pid=17
  [MLFQ] child 18 calling yield() twice...
    before getlev()=0, pid=18
  [MLFQ] child 19 calling yield() twice...
    before getlev()=0, pid=19
    after 1st yield getlev()=0, pid=16
    after    after 1st yield getlev()=0, pid=18
    after 1st yield getlev()=0, pid=19
    after 2nd yield getlev()=0, pid=16
    after 2nd yield getlev()=0, pid=18
    after 2nd yield getlev()=0, pid=19
 1st yield getlev()=0, pid=17
    after 2nd yield getlev()=1, pid=17
 => [4] yield 동작 확인 완료

[5] setpriority 테스트
  setpriority(20,2) 성공
  setpriority(20,-1) => -2 (기대: -2)
  setpriority(9999,2) => -1 (기대: -1)
  child 20 exiting
 => [4] setpriority 확인 완료

[6] priority boost 테스트
already in MLFQ mode.
  child 21 at iteration 0, level=1
  child 22 at iteration 0, level=1
  child 23 at iteration 0, level=1
  child 24 at iteration 0, level=1
  child 21 at iteration 10, level=2
  child 22 at iteration 10, level=2
  child 23 at iteration 10, level=1
  child 24 at iteration 10, level=1
  ch  child 22 at iteration 20, level=2
ild 21 at iteration 20, level=2
  child 23 at iteration 20, level=0
  child 24 at iteration 20, level=0
  child 22 at iteration 30, level=2
  child 21 at iteration 30, level=2
  child 24 at iteration 30, level=2
  child 23 at iteration 30, level=0
  child 22 at iteration 40, level=1
```

```
  child 21 at iteration 40, level=2
    child 23 at iteration 40, level=2
child 24 at iteration 40, level=2
  child 21 at iteration 50, level=1
  child 22 at iteration 50, level=1
  child 23 at iteration 50, level=2
  child 24 at iteration 50, level=2
  child 22 at iteration 60, level=2
  child 21 at iteration 60, level=2
  child 23 at iteration 60, level=2
  child 24 at iteration 60, level=2
  child 21 at iteration 70, level=2
  child 22 at iteration 70, level=2
  child 23 at iteration 70, level=2
  child 24 at iteration 70, level=2
  child 22 at iteration 80, level=2
  child 21 at iteration 80, level=2
  child 23 at iteration 80, level=2
  child 24 at iteration 80, level=2
  child 21 at iteration 90, level=2
  child 22 at iteration 90, level=2
  child 23 at iteration 90, level=2
  child 24 at iteration 90, level=2
  child 22 at iteration 100, level=1
  child 21 at iteration 100, level=1
  child 23 at iteration 100, level=2
  child 24 at iteration 100, level=2
  child 21 at iteration 110, level=2
  child 22 at iteration 110, level=2
  child 23 at iteration 110, level=2
  child 24 at iteration 110, level=2
  child 22 at iteration 120, level=2
  child 21 at iteration 120, level=2
  child 22 at iteration 130, level=2
  child 23 at iteration 120, level=2
  child 24 at iteration 120, level=2
  child 21 at iteration 130, level=2
  child 22 at iteration 140, level=2
  child 21 at iteration 140, level=2
  child 23 at iteration 130, level=2
  child 24 at iteration 130, level=2
  child 22 at iteration 150, level=1
  child 21 at iteration 150, level=1
  chil  child 23 at iteration 140, level=2
d 24 at iteration 140, level=2
  c  child 21 at iterationhild 22 at iteration 160, level=2
 160, level=0
  child 24 at iteration 150, level=2
  child 23 at iteration 150, level=2
  child 22 at iteration 170, level=2
  child 21 at iteration 170, level=2
  child 24 at iteration 160, level=2
```

```
  child 23 at iteration 160, level=2
  child 22 at iteration 180, level=2
  child 21 at iteration 180, level=2
  child 24 at iteration 170, level=2
  child 23 at iteration 170, level=2
  child 22 at iteration 190, level=2
  child 21 at iteration 190, level=2
  child 23 at iteration 180, level=2
  child 24 at iteration 180, level=2
  child 24 at iteration 190, level=2
  child 23 at iteration 190, level=2
 => [6] priority boost 확인 완료

=== 모든 테스트 완료 ===
```

TROUBLESHOOTING

1. 처음에 `fcfs` 큐를 구현하고 실행했을 때 쉘이 안나오는 오류가 발생하였다. 이것은 쉘 프로세스가 `fcfs ready 큐`에 있지 않아서 스케줄러가 쉘 프로세스를 실행시키지 못하는 오류였다. 최초의 프로세스를 생성할 때, `fork`를 수행할 때, `wakeup`을 수행할 때 `fcfs` 큐에 넣어준 뒤 올바르게 스케줄링이 되었다.
2. `fcfs`에서의 타이머 인터럽트 발생으로 인한 `yield`가 발생하는 것을 확인하였다. `fcfs`는 비선점 스케줄러이므로 타이머 인터럽트가 발생할 때 `yield`를 못하도록 구현해야한다. 추가로 `yield`를 자발적으로 호출했을 때 `fcfs`큐의 맨 뒤로 보내는 것이 아니라 **프로세스 생성시간 기준으로 실행한다는 기준** 하에 다시 해당 프로세스를 실행하도록 하였다.
3. `yield`내부에서 `mlfq` 큐의 해당 레벨로 푸시하는 코드를 잘못 추가하여 스케줄러에서 푸시하는 코드와 중복되는 문제가 발생하였다. 코드의 일관성을 위해서 `yield`부분을 수정해주었다.
4. **특정 유저 프로그램을 실행하는 경우 `panic`이 호출되는 문제가 지속되었다. 이것은 유저 프로그램이 `mlfq`모드로 종료되는 상황에서만 발생하였다. 이유는 `shell`이 `init` 프로세스의 `fork`를 통해 생성되며 생성 당시`fcfs mode`환경이었으므로 `level` 값은 -1인데 `mlfq`모드로 유저 프로그램이 종료될 경우 유저 프로그램이 쉘을 `wakeup`함에 따라 `mlfq ready 큐`로 쉘을 집어넣는 상황이 발생하였다. `mlfq`에 쉘이 들어가는데 쉘의 `level`이 -1이므로 `panic`이 호출되는 것이었다. `wakeup` 대상이 쉘일 경우에는(유저 프로그램이 종료될 때) 모드를 `FCFS모드`로 다시 전환한 후 쉘을 `wakeup` 해주는 방식으로 코드를 수정하였다.**