Operating Systems Project 2 Wiki

2018008904 이성진

Contents

1. Design

- xv6가 기본적으로 사용하는 scheduler는 Round-Robin 방식으로 스케줄링을 진행합니다. 1 tick(약 10ms)마다 타이머 인터럽트가 발생하면, yield() 함수가 호출되어 현재 실행 중이었던 프로세스의 상태를 RUNNING에서 RUNNABLE로 전환합니다. 그리고 scheduler로 context switching을 진행합니다. scheduler는 프로세스들의 배열에서 RUNNABLE한 프로세스 중 다음 인덱스의 프로세스를 선택하여 swtch 함수를 사용하여 다시 context switching을 진행합니다.

```
  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  if(myproc() && myproc()->state== RUNNING &&
    tf->trapno== T_IRQ0+IRQ_TIMER){
      yield(); // Calls sched() inside to enter scheduler
```

[part of **trap.c**]

```
 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      // 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 = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&(c->scheduler), p->context);
```

[part of **proc.c**]

- 다음 프로세스로 context switching이 일어난 이후 sleep, yield 등의 함수가 실행되면 다시 스케줄러로 control flow가 넘어와서 같은 과정을 지속하게 됩니다. 이러한 흐름에서 스케줄러로 context switching이 일어나는 곳은 항상 같은 위치이기 때문에, 결과적으로 p table을 순차적으로 순환하며 Round-Robin 방식으로 스케줄링이 이루어지게 됩니다.

Multilevel Queue scheduler design

- Multilevel Queue의 경우에는 pid를 기준으로 짝수 pid를 가진 프로세스와 홀수 pid를 가진 프로세스를 분리하여 스케줄링합니다. scheduler가 다음번 context switching을 진행할 프로세스를 선택할 때 항상 짝수 pid를 가진 프로세스를 먼저 선택한 뒤 짝수 pid를 가진 RUNNABLE한 프로세스가 더 이상 남아있지 않다면 홀수 pid를 가진 프로세스를 선택할 수 있습니다.
- 이러한 구조를 만들기 위해서 scheduler() 함수 내부 프로세스를 선형적으로 탐색하는 반복문 구조를 변경하여 짝수와 홀수 pid를 각각 따로 탐색하도록 하는 두 개의 반복문 구조를 만들어야 합니다. 그리고 홀수 pid를 스케줄링하는 도중에도 새로운 짝수 pid를 가진 RUNNABLE 프로세스가 나타날 수 있으므로, 홀수 pid를 스케줄링하는 반복문의 각 loop에서 짝수 pid를 가진 RUNNABLE 프로세스가 존재하는지 매번 확인해야 합니다.
- 짝수 pid를 가진 프로세스들은 Round-Robin 방식으로 순환하며 실행되므로 기존의 스케줄링 방식을 그대로 사용하면 됩니다. 하지만 홀수 pid를 가진 프로세스들은 FCFS 방식으로 처리되어야 하므로, 가장 pid가 작은 프로세스가 먼저 처리되도록 하는 조건을 추가해야 합니다.

MLFQ scheduler design

- Multilevel Feedback Queue 같은 경우는 2~5개의 서로 다른 level을 가진 queue에 각 프로세스들이 속하게 됩니다. scheduler가 매번 프로세스를 선택할 때마다 가장 level이 작은 queue에서 먼저 스케줄링을 진행하고, 각각의 queue 내부에서는 프로세스마다 가지게 되는 priority를 기준으로 스케줄링을 진행합니다. 각각의 프로세스는 처음 생성될 때 최우선 L0 queue에 속하게 되지만, 각 level에 따른 quantum(level*4 + 2)를 모두 사용하고 나면 다음 queue로 이동하게 됩니다.
- 가장 하위 레벨의 queue로 이동한 뒤 주어진 모든 시간을 사용한 프로세스는 100 tick마다 일어나는 priority boosting이 일어나기 전 까지는 다시 선택되지 않게 됩니다. 이를 위하여 타이머 인터럽트가 100번 일어날 때 마다 모든 프로세스들을 L0 queue로 이동시키고 실행 시간을 초기화하는 과정을 구현해야 합니다.
- 이러한 구조를 구현하기 위해 우선 각각의 프로세스들의 메타데이터를 저장하는 구조체 proc에 현재 속한 queue의 level, priority 그리고 실행 시간을 저장할 수 있는 멤버 변수를 추가해야 합니다.

2. Implement

Editing Makefile & adding new kernel files

- make 명령의 argument로 SCHED_POLICY를 입력하여 이에 따라 서로 다른 scheduler가 작동하도록 해야하기 때문에, Makefile의 내용을 일부 변경하였습니다. SCHED_POLICY가 지정되지 않은 경우 기본 Round-Robin scheduer가 사용되도록 ROUND_ROBIN 값을 지정하도록 하였습니다(과제 명세에는 없는 optional part입니다). scheduler와 관련된 구조체 및 함수들은 거의 모두 proc.c 파일에 속해있기 때문에, schduler의 종류에 맞게 컴파일되도록 proc_multilevel.c 와 proc_mlfq.c 파일을 별도로 분리해서 작성하고 SCHED_POLICY에 따라 컴파일될 수 있도록 했습니다.

```
ifndef SCHED_POLICY
SCHED_POLICY := RR_SCHED
endif

ifeq ($(SCHED_POLICY), RR_SCHED)
   PROC := proc.o
endif

ifeq ($(SCHED_POLICY), MULTILEVEL_SCHED)
   PROC := proc_multiLevel.o
endif

ifeq ($(SCHED_POLICY), MLFQ_SCHED)
   PROC := proc_MLFQ.o
   ifndef MLFQ_K
   MLFQ_K := 3
   endif
endif
```

```
OBJS = \
  bio.o\
  console.o\
  exec.o\
  file.o\
  fs.o\
  ide.o\
  ioapic.o\
  kalloc.o\
  kbd.o\
  lapic.o\
  log.o\
  main.o\
  mp.o\
  picirq.o\
  pipe.o\
  $(PROC)\
  sleeplock.o\
```

[part of **Makefile**]

- MLFQ schduler의 경우에는 타이머 인터럽트가 발생했을 경우 특정 작업을 수행해야 하기 때문에, 인터럽트 관련 작업을 수행하는 코드가 있는 trap.c와 yield, sleep system call을 정의하는 sysproc.c 역시 일부 수정할 필요가 있었고 MLFQ_K 값을 make 명령에서 받아서 사용해야 했습니다. 그래서 makefile에서 SCHED_POLICY가 MLFQ_SCHED이며 MLFQ_K 값이 정의된 경우 .c파일을 빌드하는 gcc flag에 -D 명령을 이용하여 MLFQ_K 값을 전처리 과정에서 정의하도록 했습니다. trap.c와 sysproc.c 같은 코드 파일은 #ifdef MLFQ_K 전처리문을 사용해 MLFQ scheduler를 사용할 때 추가적인 작업을 하도록 설정했습니다.

```
ifeq ($(SCHED_POLICY), MLFQ_SCHED)
$(info MLFQ_K is $(MLFQ_K))
CFLAGS += -D MLFQ_K=$(MLFQ_K)
endif
```

[part of **Makefile**]

Implementing Multilevel Queue scheduler

- Multilevel Queue scheduler를 구현하는 과정은 MLFQ에 비해 비교적 간단합니다. proc.c에 있는 scheduler() 함수 내부의 프로세스 테이블 탐색을 진행하는 반복문을 두 부분으로 나누고, 앞 반복문에서는 선형적으로 pid가 짝수인 프로세스들을, 그리고 뒷 부분에서는 pid가 홀수인 프로세스들을 먼저 실행된 순서(pid가 작은 순서)대로 선택하도록 합니다.

```
// Round-Robin scheduling for processes with even pid
   even:
   for(p = ptable.proc; p <&ptable.proc[NPROC]; p++){

    if((p->pid % 2) != 0 || p->state != RUNNABLE){
     continue;
    }
// Context switch operations will be executed
```

[part of scheduler() in **proc_multilevel.c**]

- 스케줄링할 프로세스를 선택한 뒤에는 기존 스케줄러가 했던 프로세스 상태를 RUNNING으로 변경하고 context switching을 진행하는 작업을 수행해주면 됩니다.
- 실행 중이던 프로세스가 타이머 인터럽트, yield 또는 sleep을 통해 다시 scheduler로 control flow를 넘겨주면, swtch 함수를 실행한 부분에서 다시 스케줄링을 시작하는데, 이때 만약 홀수 pid를 스케줄링하는 경우 프로세스가 실행되는 과정에서 새로운 짝수 pid 프로세스가 생성되었을 수 있기 때문에, 짝수 pid 프로세스가 있는지 확인하고 있다면 짝수 pid를 스케줄링하는 부분으로 이동하도록 label과 goto문을 이용했습니다.

```
// First-Come-First-Serve scheduling for processes with odd pid
   for(;;){
    int tmp = 0;
    struct proc* p = 0;
    for(struct proc* p1 = ptable.proc; p1 <&ptable.proc[NPROC]; p1++){
     if(p1->state != RUNNABLE)
      continue;
     else if((p1->pid % 2) != 0){
      if(tmp == 0 || tmp >p1->pid){
       tmp = p1->pid;
       p = p1;
      }
      continue;
     }
     else{
// Processes with even pid is not finished scheduling yet. Goto even.
      goto even;
     }
    }
    if(p == 0){
     break;
    }
// Context switch operations will be executed here.
```

[part of scheduler() in **proc_multilevel.c**]

Implementing Multilevel Feedback Queue scheduler

- MLFQ schduling을 구현하기 위해선 우선 기존에 정의된 프로세스 메타데이터를 저장하는 구조체 struct proc에 새로운 멤버를 추가해야 합니다. MLFQ_K가 make 과정에서 전처리된 경우에만 추가하도록 #ifdef 전처리문을 활용하여 MLFQ 스케줄링에 필요한 level, priority, 그리고 실행 시간을 저장하는 멤버 변수를 정의했습니다.

```
// Per-process state
struct proc {
  uint sz;           // Size of process memory (bytes)
  pde_t* pgdir;         // Page table
  ...
  char name[16];        // Process name (debugging)
#ifdef MLFQ_K
  int priority;         // Priority of each process
  int level;          // MLFQ queue level
  int extime;          // Time passed during execution
#endif
};
```

[part of **proc.h**]

- 각각의 프로세스가 속한 레벨에 따른 time quantum을 계산하기 위한 전처리문도 같은 파일에 정의했습니다.

```
#ifdef MLFQ_K
#define QT(LEVEL) (4*LEVEL)+2
#endif
```

[part of **proc.h**]

- 또한 현재 OS에서 사용중인 프로세스들의 목록을 저장하는 ptable 구조체에 MLFQ 스케줄링을 효율적으로 진행하기 위한 정보를 저장하는 멤버들을 추가했습니다. 바로 각 레벨에 가장 최근에 실행한 프로세스를 저장하는 heads와 현재 각 레벨에 RUNNABLE한 프로세스의 개수를 저장하는 numproc입니다.

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
   struct proc* heads[MLFQ_K]; // Stores pid of most recently executed process at each level
   int numproc[MLFQ_K];      // Stores number of runble process at each level
} ptable;
```

[part of **proc_mlfq.c**]

- 프로세스가 처음 생성되고 초기화되는 allocproc 함수 내부에 프로세스 할당 과정에서 proc 구조체에 새로 추가된 멤버 변수들을 초기화하는 코드를 삽입했습니다.

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->priority = 0;
  p->extime = 0;
  p->level = 0;
```

[part of allocproc() in **proc_mlfq.c**]

- 또한 프로세스가 RUNNABLE로 설정되는 매 순간마다 현재 레벨에 해당하는 프로세스의 개수를 카운트하기 위해 ptable.numproc[레벨]의 값을 증가시킵니다.

```
  p->state = RUNNABLE;
  ptable.numproc[p->level]++;
```

[part of **proc_mlfq.c**]

- ptable.numproc의 값은 현재 RUNNABLE한 프로세스의 개수를 카운팅하는데 사용되므로, scheduler가 각 프로세스를 선택해서 context switching을 하기 직전에 반드시 해당 프로세스가 속한 레벨의 개수를 한 개 줄입니다.

```
      // 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 = p;
      ptable.heads[p->level] = p;
      switchuvm(p);
      p->state = RUNNING;
      ptable.numproc[p->level]--;
```

[part of scheduler() in **proc_mlfq.c**]

- 타이머 인터럽트가 발생하는 매 tick 마다, 현재 실행중인 프로세스가 있을 경우 해당 프로세스의 실행 시간을 1만큼 증가시키고, 만약 증가시킨 이후의 실행 시간이 해당 레벨의 time quantum 이상일 경우 해당 프로세스의 레벨을 1 증가시킵니다. 이 경우 각 레벨에서 가장 최근에 실행된 프로세스를 저장하는 ptable.heads[레벨]의 값을 0으로 초기화하고, 프로세스의 상태가 RUNNABLE한 경우 ptable.numproc의 값을 변경합니다.
- 프로세스의 레벨을 더 이상 증가시킬 수 없을 때는, 실행 시간을 -1로 설정합니다. 실행 시간이 -1인 priority boost가 발생하기 전에는 스케줄링 될 수 없게 됩니다.

```
// Called at timer interrupt. Increase the execution time of
// current process by 1 and change level and exec time if needed.
void
degrade(void)
{
  acquire(&ptable.lock);
  struct proc* p = myproc();
  if(!p){
   release(&ptable.lock);
   return;
  }
  p->extime++;
  if(p->extime >= QT(p->level)){
   ptable.heads[p->level] = 0;
   if(p->level == MLFQ_K - 1)
    p->extime = -1;
   else{
    if(p->state == RUNNABLE){
     ptable.numproc[p->level]--;
     ptable.numproc[p->level + 1]++;
    }
    p->extime = 0;
    p->level++;
  }}
  release(&ptable.lock);
}
```

[part of **proc_mlfq.c**]

- 타이머 인터럽트가 100번 발생할 때 마다 모든 프로세스의 레벨과 실행 시간을 0으로 초기화시키는 priority boosting을 담당하는 함수 boost를 정의했습니다.

```
// Called when 100 ticks are passed
// Send all process to queue with highest level(L0)
// and reset execution time of boosted process
void boost(void)
{
  acquire(&ptable.lock);
  for(struct proc* p = ptable.proc; p <&ptable.proc[NPROC]; p++){
    if(p->state == RUNNABLE){
     ptable.numproc[p->level]--;
     ptable.numproc[0]++;
    }
    p->level = 0;
    p->extime = 0;
  }
  release(&ptable.lock);
}
```

[part of **proc_mlfq.c**]

- degrade와 boost 함수는 각각 타이머 인터럽트가 1, 100번 발생할 때 마다 호출되어야 하므로, trap.c에서 타이머 인터럽트가 발생하는 부분에 해당 함수를 호출합니다.

```
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
   if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
#ifdef MLFQ_K
    degrade();
    if(!(ticks % 100))
     boost();
#endif
    wakeup(&ticks);
    release(&tickslock);
   }
   lapiceoi();
   break;
```

[part of trap(tf) in **trap.c**]

- 위와 같은 과정을 바탕으로 저장한 메타데이터들을 바탕으로, MLFQ 스케줄링을 구현하기 위해 scheduler() 함수를 아래와 같이 수정하였습니다. Multilevel Queue scheduler와 유사하게, 각 레벨에 따라 독립된 반복문 안에서 계속해서 프로세스를 탐색합니다.
- 한 프로세스의 실행이 종료된 이후에는 더 높은 우선순위의 큐에 프로세스가 남아있는지 확인하며 만약 존재한다면 해당 레벨에서 다시 탐색할 수 있도록 goto 문을 활용하여 반복문 실행 전으로 되돌아가도록 했습니다.
- 더 높은 우선순위의 queue에 RUNNABLE한 프로세스가 없는 경우, 현재 레벨의 프로세스들을 priority 순으로 선택하여 스케줄링합니다.

```
  for(;;){
   // Enable interrupts on this processor.
   sti();
   // Loop over process table looking for process to run.
   acquire(&ptable.lock);
   schbegin:
   for(int level = 0; level <MLFQ_K; level++){
    do{
     schsearch:
     for(p = ptable.proc; p <&ptable.proc[NPROC]; p++){
     // Check whether runble process in higher level queue exists
      for(int i = 0; i <level; i++)
       if(ptable.numproc[i] >0){
        goto schbegin;
       }
     // Process p must be in runble state and stays in current level queue
      if(p->state != RUNNABLE || p->level != level)
       continue;
     // Check if current process have highest priority in current queue
      struct proc* p1 = 0;
      int pflag = 0;
      if(ptable.numproc[level] >1){
       for(p1 = ptable.proc; p1 <&ptable.proc[NPROC]; p1++){
        if(p1->state == RUNNABLE && p1->level == level && p1->priority >
            p->priority){
         pflag = 1;
         break;
        }
       }
      }
      if(pflag){
       if(p1 <p)
        goto schsearch;
       else
        continue;
      }

      // Context switch operations will be executed here.

     }
    }while(ptable.numproc[level] >0);
   }
   release(&ptable.lock);
  }
```

[part of scheduler() in **proc_mlfq.c**]

- 유저 프로그램이 현재 프로세스가 속한 queue의 level을 알 수 있게 하는 system call인 getlev는 아래와 같이 정의한 후 기존 system call들을 구현한 것처럼 유저 프로그램에서 사용 가능하도록 했습니다.

```
int getlev(void){
   return myproc()->level;
}
```

[part of **proc_mlfq.c**]

- 마찬가지로 유저 프로그램이 우선순위를 지정할 수 있는 setpriority system call을 아래와 같이 구현했습니다.
- 프로세스 목록을 순환하며 setpriority의 대상이 현재 사용중인 프로세스이며 system call을 호출한 프로세스의 자식인 것이 확인되면 정상적으로 priority를 지정하고 0을 return하고, 아닌 경우 상황에 맞는 오류 코드를 return합니다.

```
int setpriority(int pid, int priority){
  if(priority < 0 || priority > 10) return -2;
  struct proc* curproc = myproc();
  acquire(&ptable.lock);
  for(struct proc* p = ptable.proc; p <&ptable.proc[NPROC]; p++){
   if(p->state == UNUSED) continue;
   if(p->parent != curproc || p->pid != pid) continue;
   else{
    p->priority = priority;
    release(&ptable.lock);
    return 0;
   }
  }
  release(&ptable.lock);
  return -1;
}
```

[part of **proc_mlfq.c**]

- yield나 sleep system call을 유저가 호출할 때 마다 해당 프로세스의 사용이 종료되었다고 가정하고 실행 시간과 레벨을 초기화시키므로, 해당 작업을 수행하는 rst 함수를 정의하고 yield 및 sleep system call의 내부에서 호출하도록 했습니다.
- yield, getlev, setpriority는 sysproc.c와 sysproc.h 파일에 새로 정의했습니다. 전처리 지시문을 통해 MLFQ 스케줄러일 때 정상 작동하도록 설계했습니다.

```
// Called when yield &sleep system call is called.
// Assume that the process which called syscall has ended executing
// so that the kernel can initialize the level and execution time of the
process.
void rst(void)
{
  acquire(&ptable.lock);
  struct proc* p = myproc();
  if(p->state == RUNNABLE){
   ptable.numproc[p->level]--;
   ptable.numproc[0]++;
  }
  p->level = 0;
  p->extime = 0;
  release(&ptable.lock);
}
```

[part of **proc_mlfq.c**]

```c
int
sys_sleep(void)
{
  int n;
  uint ticks0;
#ifdef MLFQ_K
  rst();
#endif
  ...
  release(&tickslock);
  return 0;
}
```

```c
// System call version of yield
int
sys_yield(void)
{
#ifdef MLFQ_K
  rst();
#endif
  yield();
  return 0;
}
```

```c
// System call getlev
int
sys_getlev(void){
#ifdef MLFQ_K
  return getlev();
#else
  return -1;
#endif
}
```

```c
// System call setpriority
int
sys_setpriority(void){
#ifdef MLFQ_K
  int a, b;
  if(argint(0, &a) <0)
   return -1;
  if(argint(1, &b) <0)
   return -2;
  return setpriority(a, b);
#else
  return -1;
#endif
}
```

[part of **sysproc.c**]

3. Result

Multilevel Queue scheduling test result

- make 과정에서 SCHED_POLICY=MULTILEVEL_SCHED를 지정하고 컴파일한 후 ml_test 테스트 프로그램을 실행한 결과는 아래와 같습니다.

```
girinman@DESKTOP-NLJQQDD:~/my_repos/OS/xv6-public$ make CPUS=1 SCHED_POLICY=MULTILEVEL_SCHED
```

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 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
$ ml_test
```

- Test 1 결과, 짝수 프로세스가 홀수 프로세스보다 먼저 스케줄링되는 것을 확인할 수 있습니다.
- 짝수 프로세스들은 Round-Robin 방식으로 스케줄링되기 때문에 4번과 6번 프로세스가 서로 뒤섞여 실행되지만, 홀수 프로세스는 FCFS 방식으로 스케줄링되기 때문에 5번 프로세스가 실행이 끝난 뒤에야 7번 프로세스가 실행되는 것을 볼 수 있습니다.

```
Multilevel test start
[Test 1] without yield / sleep
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
```

```
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
[Test 1] finished
```

- Test 2 결과, 각 프로세스들이 yield를 지속적으로 호출하는데, 짝수 프로세스들은 거의 동시에 실행이 종료되지만, 홀수 프로세스들은 pid가 작은 순서대로 순차적으로 완료됐습니다.

```
[Test 2] with yield
Process 10 finished
Process 8 finished
Process 9 finished
Process 11 finished
[Test 2] finished
```

- Test 3 결과, 각 프로세스들이 sleep을 지속적으로 호출하는데, sleep 상태인 도중에는 다른 프로세스가 스케줄링될 수 있기 때문에 짝수 프로세스들이 실행된 후 sleep 상태에 들어서면 홀수 프로세스들이 pid 순으로 실행되는 방식으로 각 프로세스들이 순차적으로 실행되는 것을 확인할 수 있습니다.

```
[Test 3] with sleep
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
[Test 3] finished
```

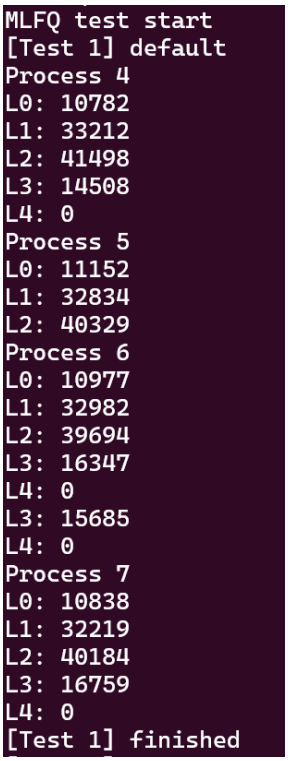
Multilevel Feedback Queue scheduling test result

- make 과정에서 SCHED_POLICY=MLFQ_SCHED MLFQ_K=5를 지정하고 컴파일한 후 mlfq_test 테스트 프로그램을 실행한 결과는 아래와 같습니다.

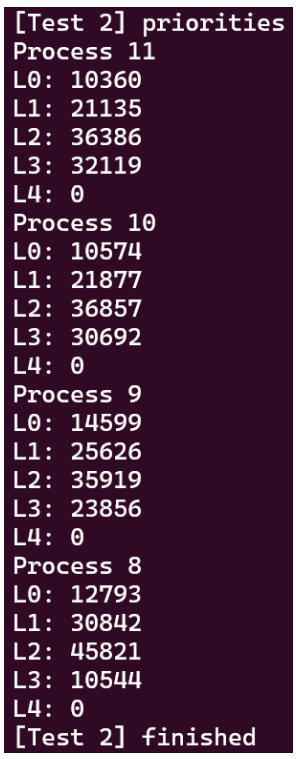
```
girinman@DESKTOP-NLJQQDD:~/my_repos/OS/xv6-public$ make CPUS=1 SCHED_POLICY=MLFQ_SCHED MLFQ_K=5
```

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 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
$ mlfq_test
```

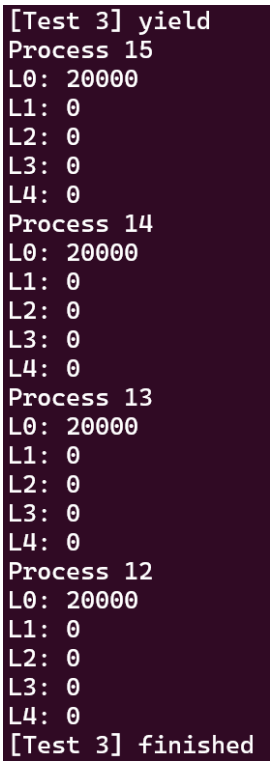
- Test 1 결과, 사실상 Round-Robin 방식으로 스케줄링 되는 각 프로세스가 거의 같은 시간을 소모하면서 거의 비슷한 시기에 프로세스가 종료되고, 일부 출력이 꼬이는 등 반드시 pid 순서로만 호출되지 않는다는 것을 확인할 수 있습니다.



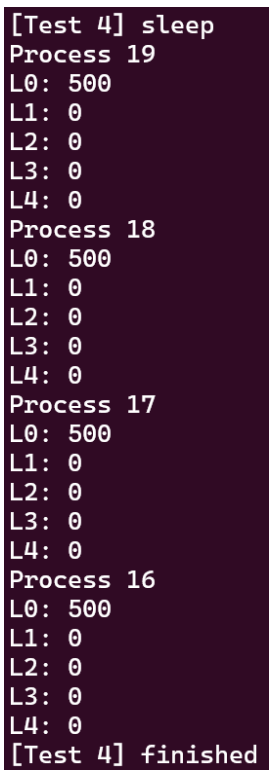
- Test 2 결과, 전체적인 실행시간은 유사하지만 pid가 큰 프로세스에 더 높은 priority가 부여되었기 때문에 더 먼저 종료될 확률이 높았고, 실제로 pid가 높은 순서대로 실행 완료된 것을 알 수 있습니다.



- Test 3 결과, 각 프로세스가 루프마다 yield system call을 호출하기 때문에, 각 프로세스는 항상 L0 queue에 속해 있게 됩니다. pid가 높은 순서대로 높은 priority가 부여되었기 때문에, L0에 속해 있는 프로세스들은 pid가 큰 순서대로 스케줄링 되어 pid가 큰 순서대로 먼저 실행을 완료하게 되는 것을 알 수 있습니다.



- Test 4 결과 역시 마찬가지로, 각 프로세스가 루프마다 sleep system call을 호출하기 때문에, 각 프로세스는 항상 L0 queue에 속해 있게 됩니다. pid가 높은 순서대로 높은 priority가 부여되었기 때문에, L0에 있는 프로세스들은 pid가 큰 순서대로 스케줄링 되어 pid가 큰 순서대로 먼저 실행을 완료하게 되는 것을 알 수 있습니다.



- Test 5 실행 결과 pid가 작은 프로세스들은 레벨이 작은 queue에만 있으려고 하고, pid가 큰 프로세스들은 상대적으로 레벨이 큰 queue로 밀려나게 되므로 pid가 작은 순서대로 실행이 끝나게 됩니다. 실제로 pid가 큰 프로세스일수록 레벨이 큰 queue에 더 많이 머물러 있다는 것을 알 수 있습니다.

```
[Test 5] max level
Process 20
L0: 99973
L1: 27
L2: 0
L3: 0
L4: 0
Process 21
L0: 27184
L1: 72810
L2: 6
L3: 0
L4: 0
Process 22
L0: 15106
L1: 38353
L2: 46539
L3: 2
L4: 0
Process 23
L0: 12547
L1: 27268
L2: 36255
L3: 23929
L4: 1
[Test 5] finished
```

-Test 6 실행 결과 부모가 자식 프로세스의 priority를 설정할 때만 사용 가능한 setpriority가 정상 작동한다는 것을 확인할 수 있었습니다.

```
[Test 6] setpriority return value
done
[Test 6] finished
```

4. Troubleshooting

- 기존의 Round-Robin scheduling을 Multilevel Queue scheduling 시스템으로 변경하기 위한 개념을 적용하는 과정에서 아래와 같은 문제 상황들과 그 해결법이 있었습니다.

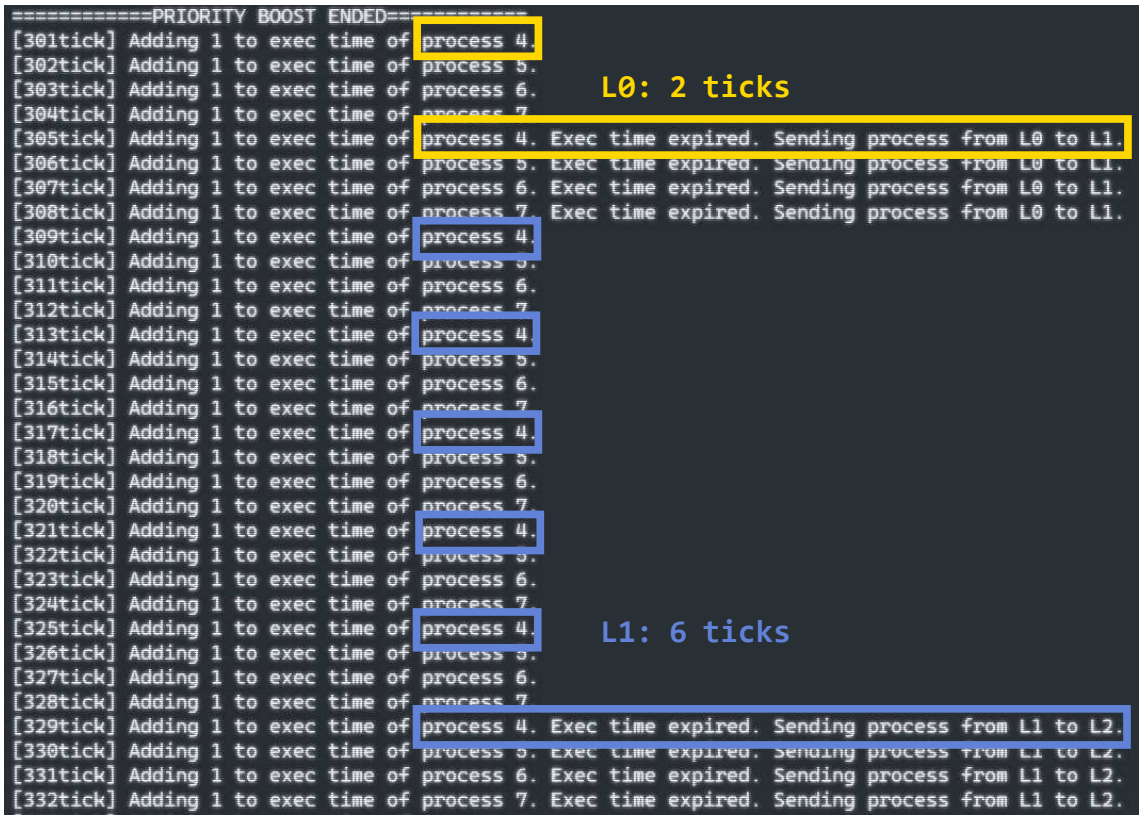
Difficulties during debugging

- xv6는 커널과 유저 영역으로 나뉘어 있으며 가상 하드웨어 위에서 작동하기 때문에, 일반 C/C++ 코드처럼 GDB 등을 활용하여 쉽게 디버깅하기는 다소 어려운 점이 있었습니다. 그래서 특정 코드 실행 시점에 각각의 값들이 의도된 값을 가지는지 출력하도록 했습니다.

```
============PRIORITY BOOST ACTIVATED============
pid=1 boosted from L0 to L0.
pid=2 boosted from L0 to L0.
pid=3 boosted from L1 to L0.
pid=4 boosted from L2 to L0.
pid=5 boosted from L2 to L0.
pid=6 boosted from L2 to L0.
pid=7 boosted from L2 to L0.
All boosted process's exec time initialized.
============PRIORITY BOOST ENDED============
```

[100 tick이 지날 때마다 일어나는 priority boost 과정을 출력한 모습]

- 아래 출력된 결과는 mlfq_test 프로그램의 Test 1 진행 과정에서 출력된 것입니다. 300 tick에 priority boost가 발생한 뒤로 모든 프로세스의 레벨이 0이 되고 실행시간이 초기화된 상태입니다. 이후 301 tick부터는 1 tick마다 실행 중인 프로세스의 실행시간을 증가시키고, 퀀텀을 모두 사용했는지 확인합니다. L0에 있는 프로세스들은 2 tick을 사용한 후에 L1으로 이동하고, L1으로 이동한 프로세스들은 6tick을 사용한 후에 L2로 이동하는 것을 볼 수 있습니다.



[1 tick마다 프로세스의 실행시간이 증가하고 퀀텀을 사용한 이후에는 레벨이 변경되는 모습]

- 테스트가 완료되어 정상 작동하는 것을 확인한 뒤에는, 모든 디버깅용 cprintf 함수 호출 전에 if(VERBOSE) 조건을 추가하였습니다. VERBOSE는 0이면 디버깅용 문구를 출력하지 않고, 1이면 출력하도록 하는 조건을 지정하는 값입니다. def.h에 정의하였고, make 명령에서 VERBOSE 값을 지정하는 것을 통해 디버깅할지 안 할지 결정할 수 있도록 해서 손쉽게 OS가 작동하는 모습을 원하는 때에 볼 수 있도록 했습니다.

```
if(VERBOSE) cprintf("==========PRIORITY BOOST ACTIVATED==========\n");
  for(struct proc* pptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state!= UNUSED){
      if(VERBOSE)
        cprintf("pid=%d boosted from L%d to L0.\n", p->pid, p->level);
      if(p->state == RUNNABLE){
        ptable.numproc[p->level]--;
        ptable.numproc[0]++;
       }
     p->level = 0;
     p->extime = 0;
```

[part of boost() in **proc_mlfq.c**]

```
ifndef VERBOSE
VERBOSE:= 0
endif
...
CFLAGS+= -D VERBOSE=$(VERBOSE)
```

[part of **Makefile**]

Where to call rst()?

- 과제 명세에 따르면 유저 프로그램이 **yield** 또는 **sleep**을 호출하면 프로세스의 레벨과 실행시간을 초기화해야 합니다. 이를 위해 **yield**와 **sleep** 함수의 정의 내부에 **rst** 함수를 호출한 결과, 매번 타이머 인터럽트가 발생할 때마다 **yield** 함수가 호출되어 초기화가 일어나고, 유저가 **system call**을 호출한 것이 아닌 커널 내부에서 실행된 **sleep**의 경우에도 초기화가 발생하는 등 의도되지 않게 계속해서 **rst** 함수가 호출되어 결과적으로 **test 3, 4**의 경우처럼 프로세스가 실행 내내 **L0**에서 다른 레벨로 이동하지 않는 상황이 있었습니다.

- 그래서 유저가 **system call**을 실행할 때와 그렇지 않을 때를 구분하기 위해서, **sys_yield**와 **sys_sleep**의 정의 내부에 **rst** 함수를 호출하도록 했습니다. 이를 통해 과제 명세에서 의도된 부분을 정확하게 구현할 수 있었습니다. **Test 3, 4**의 결과가 이를 뒷받침해 줍니다.

```
int
sys_sleep(void)
{
  int n;
  uint ticks0;
#ifdef MLFQ_K
  rst();
#endif
  if(argint(0, &n) < 0)
    return -1;
  acquire(&tickslock);
  ticks0 = ticks;
  while(ticks - ticks0 < n){
    if(myproc()->killed){
      release(&tickslock);
      return -1;
}
    sleep(&ticks, &tickslock);
}
  release(&tickslock);
  return 0;
}
```

```
// System call version of yield
int
sys_yield(void)
{
#ifdef MLFQ_K
  rst();
#endif
  yield();
  return 0;
}
```

[part of **sysproc.c**]