Operation System Assignment 1 Wiki

2017033654 류동원

1. Design

A. XV6 스케줄러

i. 기본적으로 Round-Robin 정책을 사용

ii. 타이머 인터럽트가 발생하면 현재 실행중인 프로세스를 RUNNABLE 상태로 전환시키고, 프로세스의 배열에서 다음 인덱스의 프로세스를 실행시킨다. 배열의 마지막 프로세스까지 실행시켰다면 배열의 처음 프로세스부터 다시 시작한다.

```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }
  lapiceoi();
  break;
```

```
// 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();
```

iii. 타이머 인터럽트는 1 tick (10 ms) 마다 발생하며, 이때마다 yield()가 호출되어 프로세스 간의 context switch가 발생한다.

```
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);
  switchkvm();

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

B. 3-level feedback queue

i. L0 ~ L2, 총 3개의 큐로 이루어져 있고, 숫자가 작을수록 우선순위 높음

ii. 각 큐는 {4, 6, 8}의 time quantum을 가짐

iii. 처음 실행된 프로세스는 가장 높은 레벨의 큐 (L0)에 들어감

iv. 프로세스가 실행되어 자신의 큐의 할당된 time quantum을 다 소진하면 다음 레벨의 큐로 이동하는 방식

v. L2 큐에서 실행된 프로세스가 time quantum을 다 사용한 경우, 해당 프로세스의 priority 값이 1 감소하고 time quantum은 초기화 됨. Priority 값이 0일 경우 더 이상 값을 감소시키자 않고 0으로 유지

vi. 정책

1. 기본적으로 Runnable한 프로세스를 L0큐 -> L1큐 -> L2큐 순으로 탐색
2. L0 큐와 L1는 큐
   A. 기본 Round-Robin 정책과 First Come First Serve (FCFS) 정책
3. L2 큐
   A. Priority 스케줄링
   B. 프로세스가 처음 실행될 때 3으로 설정, setPriority() 시스템 콜로 변경 가능
   C. 우선순위가 같은 프로세스끼리는 FCFS 스케줄링을

vii. Priority Boosting

1. Starvation을 막기 위하여 Global tick이 100 ticks가 될 때마다 모든 프로세스들은 L0 큐로 재조정 함.
2. 이 때 모든 프로세스들의 priority 값은 3, time quantum 값은 0으로 초기화

viii. System call 함수 작성 – 5개의 system call 함수를 작성해야 함.

1. yield() : 다음 프로세스에게 프로세서를 양보함.
2. getLevel() : 프로세스가 속한 큐의 레벨을 반환.
3. setPriority() : 해당 pid의 프로세스의 priority를 설정함/
4. schedulerLock() : 해당 프로세스가 우선적으로 스케줄링 되도록 함.
5. schedulerUnlock() : 해당 프로세스가 우선적으로 스케줄링 되던 것을 중지함.

6. schedulerLock(), schedulerUnlock() 시스템 콜은 인터럽트를 통해 실행가능 해야 함.

2. Implementation

A. [1]MLFQ 큐를 구현하기 위해서 기존의 proc 구조체에 아래와 같은 멤버를 추가한다.

```
int level;                  // Queue level of MLFQ
int used_time;              // used time of Process in each level
int priority;               // priority value of Process
int arrival_time;           // process가 생성된 시간
int locked;                 // 프로세스를 lock함
```

[proc.h 파일]

i. level은 현재 프로세스가 속해있는 큐의 레벨을 나타냄

ii. used_time은 현재 실행된 프로세스가 각 레벨의 큐에서 사용된 time quantum을 나타냄

iii. priority는 프로세스의 우선순위를 나타냄

iv. arrival_time은 프로세스가 각 레벨의 큐에 도착한 시간을 나타냄

v. locked는 현재 실행중인 프로세스가 schedulerLock() 시스템 콜에 의해서 우선적으로 처리되고 있는지 여부를 나타냄

B. 큐의 레벨 사이즈를 정의해줍니다.

```
#define NMLFQ        3      // size of Queue level
```

[param.h 파일]

C. allocproc(void) 메소드에 의하여 각 프로세스가 생성될 때, 프로세스 멤버들을 초기화

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  // 프로세스 멤버 초기화
  p->priority = 3;
  p->used_time = 0;
  p->level = 0;
  p->locked = 0;
  p->arrival_time = global_ticks;
```

[allocproc() in proc.c] 파일

i. priority = 3, used_time = 0, level = 0 으로 초기화해줍니다.

ii. schedulerLock()이 호출되지 않았으므로 locked 변수도 0으로 초기화해줍니다.

iii. 프로세스의 L0큐의 arrival_time을 현재의 global_ticks 변수로 초기화해줍니다. (FCFS 스케줄링 정책에 활용)

D. Scheduler 함수

i. 특정한 레벨의 자료구조 큐를 직접 구현하지 않고, proc 구조체 안에 level이라는 변수를 사용하여 논리적으로 level을 구분함.

ii. flag 변수는 ptable을 한 바퀴 탐색하면서 runnable한 프로세스를 찾았을 경우 context switch를 실행하는 구간인 execute: 로 점프하고자 하는 기능

iii. 각각의 레벨에서 FCFS 정책을 구현하기 위해, min_time 변수를 사용

1. arrival_time은 앞서 각 프로세스가 각각의 레벨에 진입할 때의 global_ticks로 초기화해줌

A. priority boost로 인해 global_ticks는 100 ticks마다 초기화되므로 arrival_time은 0 ~ 100범위로 나타난다

2. 각 레벨에서 ptable의 runnable한 프로세스를 탐색하면서, 발견된 runnable한 프로세스의 arrival_time이 기존의 runnable한 프로세스의 arrival_time(min_time으로 저장됨)보다 작으면 min_time을 해당 프로세스의 arrival_time으로 update한 후, 해당 프로세스를 FCFS 변수에 저장

3. ptable을 한 바퀴 전부 탐색하면서 runnable 하며, arrival_time이 최소가 되는 (제일 먼저 도착한) 프로세스를 실행시키고자 하는 목적.

iv. 전체적으로 Level 0 -> Level 1 -> Level 2 순으로 runnable한 process를 탐색하며, L0에 없을 경우 L1 탐색, L1에도 없을 경우 L2를 탐색하는 구조.

v. 즉, 낮은 레벨에서 runnable한 프로세스 발견하면 execute: 로 가서 바로실행

vi. level 2에서는 priority 스케줄링을 구현하기 위해 총 네번의 ptable을 탐색하게 됨.

1. 첫 번째는 priority=0이고 runnable한 프로세스를 찾기 위한 ptable 탐색

2. 두 번째는 priority=1이고 runnable한 프로세스를 찾기 위한 ptable 탐색

3. 세 번째는 priority=2이고 runnable한 프로세스를 찾기 위한 ptable 탐색

4. 네 번째는 priority=3이고 runnable한 프로세스를 찾기 위한 ptable 탐색

5. priority가 높고(숫자가 낮은) runnable한 프로세스를 찾았을 경우 priority가 낮

은 (숫자가 큰) runnable한 프로세스는 탐색하지 않고 execute: 로 가서 실행

```
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);
    // arrival_time이 가장 작은 먼저 도착한 프로세스 저장
    struct proc *fcfs = 0;
    // runnable한 process 찾았으면 1로 변경
    uint flag = 0;
    // arrival_time이 가장 작은 프로세스의 도착시간 저장 (FCFS 정책을 위해)
    uint min_time = 999999;

    // level 0 -> level 1 -> level 2 순서로 runnable한 프로세스 탐색
    for(int i = 0; i < NMLFQ; i++) {
      // level 0과 1은 Round Robin + FCFS 스케줄링
      if(i == 0 || i == 1) {
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
          // ptable 탐색하면서 RUNNABLE 이고 level=i 인 프로세스 찾음
          if(p->state != RUNNABLE || p->level != i) {
            continue;
          }
          // FCFS 정책을 위해 같은 level에서는 arrival_time이 제일 작은 프로세스 찾음
          if ( p->arrival_time <= min_time ) {
            min_time = p->arrival_time;
            fcfs = p; // Runnable한 프로세스를 fcfs 포인터 변수에 저장
            flag = 1; // Runnable한 프로세스 찾았으면 flag = 1 로 변경
          }
        }
        // level i의 arrival_time이 최소인 runnable한 프로세스 찾았으면 execute로 가서 실행
        if (flag == 1) {
          p = fcfs;
          goto execute;
```

[scheduler() in proc.c 파일]

```
    } else if (i==2) {      // level 2는 Priority + FCFS 스케줄링
      // L2 큐에서는 priority가 낮은 순서대로 탐색함 0 -> 1 -> 2 -> 3
      for (int j = 0; j <= 3; j++) {
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
          // ptable 탐색하면서 RUNNABLE 이고 level=2 이고 특정한 priority를 가지는 프로세스만 찾음
          if (p->state != RUNNABLE || p->level != i || p->priority != j) {
            continue;
          }
          // FCFS 정책을 위해 같은 level, 같은 priority에서는 arrival_time이 제일 작은 프로세스 찾음
          if ( p->arrival_time <= min_time ) {
            min_time = p->arrival_time;
            fcfs = p;
            flag = 1;
          }
        }
        // level 2의 특정한 priority를 가지는 runnable한 프로세스 찾았으면 execute로 가서 실행
        if (flag == 1) {
          p = fcfs;
          goto execute;
        }
      }
    }
  }
  // runnable한 프로세스를 찾지 못했으면 첫 for문으로 돌아가서 재탐색
  release(&ptable.lock);
  continue;

  execute:
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    swtch(&(c->scheduler), p->context);
    switchkvm();
    c->proc = 0;
    release(&ptable.lock);
  }
}
```

[scheduler() in proc.c 파일]

E. Timer Interrupt (1 ticks) 마다 프로세스 yield 함

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

  // Timer Interrupt 호출 시 마다
  global_ticks++;
  myproc()->used_time++;
  yield();

}
```

[trap() in trap.c 파일]

i. Timer Interrupt가 일어날 때 마다 global_ticks 변수를 1씩 증가시키고, 현재 실행중인 프로세스의 used_time 변수도 1씩 증가시킴

ii. 프로세스가 yield가 되면 sched() 함수가 호출되어 context switch가 일어남

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;

  int level = myproc()->level;

  // level 0에 있는 프로세스에서 yield 발생
  if(level == 0 && myproc()->used_time >= 2*level + 4) {
      myproc()->used_time = 0;                // 각 레벨에서 사용한 time quantum 초기화
      myproc()->level++;                      // 프로세스를 다음 레벨로 증가시킴
      myproc()->arrival_time = global_ticks;  // 다음 레벨의 arrival_time을 global_ticks로 초기화

  // level 1에 있는 프로세스에서 yield 발생
  } else if (level == 1 && myproc()->used_time >= 2*level + 4) {
      myproc()->used_time = 0;
      myproc()->level++;
      myproc()->arrival_time = global_ticks;

  // level 2에 있는 프로세스에서 yield 발생
  } else if(level == 2 && myproc()->used_time >= 2*level + 4) {
      myproc()->used_time = 0;
      if (myproc()->priority != 0) {          // priority를 감소시킴 (이미 0이면 감소 x)
        myproc()->priority--;
      }
  }
  sched();
  release(&ptable.lock);
}
```

[yield() in proc.c 파일]

iii. yield( )가 진행될 때 실행중인 프로세스의 used_time(사용 시간)이 자신의 레벨 큐에서 사용할 수 있는 time quantum을 넘었는지 체크함 (각 레벨 time quantum = {4, 6, 8})

1. time quantum을 넘겼으면 level 0, level 1에 있던 프로세스는
   A. level이 1씩 증가하고
   B. used_time (각 레벨 큐에서 프로세스 사용시간)은 0으로 초기화
   C. arrival_time (각 레벨 큐에 도착 시간)은 현재 global_ticks로 초기화
2. time quantum을 넘겼으면 level 2에 있던 프로세스는
   A. level 증가 대신 priority 감소
   B. used_time (각 레벨 큐에서 프로세스 사용시간)은 0으로 초기화

F. System call 함수들 (모두 proc.c 파일에 구현되어 있음)

i. setPriority( )

```
// 해당 pid의 프로세스의 priority를 설정합니다.
void setPriority(int pid, int priority) {
  struct proc* p;

  // 설정되는 priority가 0보다 작거나 3보다 클 경우 리턴
  if(priority<0 || priority>3) {
    return;
  }

  // ptable을 탐색하며 입력 pid와 일치하는 프로세스를 찾아서 priority 재조정
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->pid == pid) {
      p->priority = priority;
    }
  }
  release(&ptable.lock);
  return;
}
```

G. getLevel( )

```
// 프로세스가 속한 큐의 레벨을 반환합니다.
int getLevel(void) {
  struct proc *curproc = myproc();

  return curproc->level;
}
```

H. SchedulerLock( )

```
// 해당 프로세스가 우선적으로 스케줄링 되도록 합니다.
void schedulerLock(int password) {
  int num = 2017033654;

  // 입력으로 받은 password와 학번이 일치하면 Lock을 실행
  if(password == num) {
    acquire(&ptable.lock);
    global_ticks = 0;
    // 현재 프로세스를 lock 상태로 변환
    myproc()->locked = 1;
    // level 0의 제일 높은 우선순위를 부여해서 제일 먼저 스케줄링 되도록 함
    myproc()->level = 0;
    myproc()->arrival_time = 0;
    release(&ptable.lock);
  } else {
    // 암호가 일치하지 않으면 프로세스 강제 종료
    cprintf("pid = %d, time quantum = %d, level = %d",
      myproc()->pid, myproc()->used_time, myproc()->level);
    kill(myproc()->pid);
  }
}
```

i. schedulerLock( ) 함수가 호출되면 암호가 일치하는지 확인

1. 암호가 일치하면 현재 프로세스의 locked 변수를 1로 바꿈 (locked 상태)
2. global_ticks = 0으로 초기화 및 현재 프로세스의 level = 0, arrival_time = 0으로 초기화하여 scheduling 우선순위를 제일 높게 만듦

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  // schedulerLocked
  if(p->locked == 1) {
    p->level = 0;
    p->arrival_time = 0;
    p->priority = 3;
    p->used_time = 0;
    // global ticks가 0
    if (global_ticks >= 100) {
      p->locked = 0;
    }
  }
  // global_ticks가 100보다 커지면 priority boost 발생
  if(global_ticks >= 100) {
    priority_boost();
  }
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

[sched() in proc.c 파일]

ii. schedulerLock( ) 함수 호출로 인해 호출한 프로세스의 locked 변수가 1이 되면 yield( )가 일어나서 context switch에 의해 다시 cpu 프로세스 스케줄링이기 되기 직전에 sched( )에서 해당 프로세스의 스케줄링 우선순위를 가장 높여줌

A. level = 0, arrival_time = 0, priority = 3, used_time = 0으로 계속해서 초기화시켜줌 -> 해당 프로세스가 최우선적으로 스케줄링 받을 수 있음

B. locked = 1인 상황에서 global_ticks 변수가 100이상이 되면 locked = 0 (락 해제)이 되고 priority boost 발생

I. global_ticks가 100보다 크거나 같아지면 항상 priority boost 발생

```
void priority_boost(void) {
  struct proc* p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    p->priority = 3;
    p->level = 0;
    p->used_time = 0;
    global_ticks = 0;
  }
}
```

[priority_boost() in proc.c 파일]

1. priority_boost가 호출되면 모든 프로세스들의 priority, level, used_time이 초기화되고 global_ticks도 0으로 초기화된다.

J. SchedulerUnlock()

```
// 해당 프로세스가 우선적으로 스케줄링 되던 것을 중지합니다.
void schedulerUnlock(int password) {
  int num = 2017033654;

  if(password == num) {
    acquire(&ptable.lock);
    // 암호가 일치하면 현재 프로세스를 unlock 상태로 변환
    myproc()->locked = 0;
    myproc()->level = 0;
    myproc()->arrival_time = 0;
    myproc()->priority = 3;
    myproc()->used_time = 0;
    release(&ptable.lock);
  } else {
    // 암호가 일치하지 않으면 프로세스 강제 종료
    cprintf("pid = %d, time quantum = %d, level = %d",
      myproc()->pid, myproc()->used_time, myproc()->level);
    kill(myproc()->pid);
  }
}
```

i. 패스워드와 암호가 일치한다면 프로세스의 변수들을 초기화

1. locked = 0으로 더이상 lock을 하지 않음
2. L0큐의 젤 앞으로 이동하도록 level = 0, arrival_time = 0으로 초기화
3. priority =3, time qunatum의 값인 used_time = 0으로 초기화

ii. 암호가 일치하지 않는다면 프로세스 종료

K. schedulerLock(), schedulerUnlock() 시스템 콜은 129번, 130번 인터럽트 호출시 실행

```
case 129:
  if(myproc()->killed)
    exit();
  myproc()->tf = tf;
  // schedulerLock 시스템 콜 수행
  schedulerLock(2017033654);
  if(myproc()->killed)
    exit();
  break;
case 130:
  if(myproc()->killed)
    exit();
  myproc()->tf = tf;
  // schedulerUnlock 시스템 콜 수행
  schedulerUnlock(2017033654);
  if(myproc()->killed)
    exit();
  break;
```

[trap() in trap.c 파일]

```
#define SYS_myfunction 22
#define SYS_getLevel 23
#define SYS_yield 24
#define SYS_setPriority 25
#define SYS_schedulerLock 26
#define SYS_schedulerUnlock 27
```

[syscall.h 파일]

i. 129번 130번 interrupt 호출 시 trap.c 파일의 trap() 메소드에서 해당 interrupt 번호의 switch문으로 이동하여 수행한다.

ii. myproc()->tf = tf을 통해 현재 프로세스에 trapframe 구조체 포인터를 할당해 이후 시스템 콜이 발생했을 때 프로세스 상태를 저장하거나 복원하는데 사용하며,

schedulerLock(), schedulerUnlock() 시스템 콜이 수행되도록 한다.

L. proc_systemcall.c 파일

i. 시스템 콜 함수들의 Wrapper Function을 모아놓은 파일

```c
// Wrapper Function of System call
int sys_setPriority(void) {
  int a, b;
  if(argint(0, &a) < 0) {
    return -1;
  }
  if(argint(0, &b) < 0) {
    return -2;
  }
  setPriority(a, b);
  return 0;
}

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

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

```c
int sys_schedulerLock(void) {
    int a;
    if(argint(0, &a) < 0) {
        return -1;
    }
    schedulerLock(a);
    return 0;
}

int sys_schedulerUnlock(void) {
    int a;
    if(argint(0, &a) < 0) {
        return -1;
    }
    schedulerUnlock(a);
    return 0;
}
```

3. Test

A. mlfq_test

```
root@25056d724edf:/OS/xv6-public# make CPUS=1
```

```
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
$ 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 14584
mlfq_test      2 18 17332
console        3 19 0
$
```

i. fork_children( )

```
$ ./mlfq_test
MLFQ test start
[Test 1] default
Process 8
L0: 6159
L1: 8999
L2: 84842
L3: 0
L4: 0
Process 9
L0: 9873
L1: 15499
L2: 74628
L3: 0
L4: 0
Process 6
L0: 13338
L1: 20120
L2: 66542
L3: 0
L4: 0
Process 7
L0: 16364
L1: 24116
L2: 59520
L3: 0
L4: 0
[Test 1] finished
done
$
```

ii. fork_children2( )

```
$ ./mlfq_test
MLFQ test start
[Test 1] default
Process 7
L0: 6382
L1: 8551
L2: 85067
L3: 0
L4: 0
Process 8
L0: 10542
L1: 15288
L2: 74170
L3: 0
L4: 0
Process 5
L0: 14537
L1: 20256
L2: 65207
L3: 0
L4: 0
Process 6
L0: 16779
L1: 23988
L2: 59233
L3: 0
L4: 0
[Test 1] finished
done
$
```

iii. fork_chilidren3( )

```
$ ./mlfq_test
MLFQ test start
[Test 1] default
Process 7
L0: 6493
L1: 8926
L2: 84581
L3: 0
L4: 0
Process 8
L0: 10297
L1: 15317
L2: 74386
L3: 0
L4: 0
Process 5
L0: 14283
L1: 21153
L2: 64564
L3: 0
L4: 0
Process 6
L0: 16747
L1: 24514
L2: 58739
L3: 0
L4: 0
[Test 1] finished
done
$
```

4. Trouble Shooting

A. 처음 디자인은 L0, L1 Queue 2개, L2 Priority Queue 1개를 실제로 자료구조를 만들어서 구현을 하고 프로세스를 enqueue, dequeue 하면서 관리를 하려고 했다.

B. 이유는 잘 모르겠지만 1번 프로세스가 실행이 계속 되는데 shell program 등 다른 프로그램들이 (예를들어 2번 프로세스)가 실행이 되지 않아서 계속해서 디버깅을 했지만 해결하지 못하였다.

C. 결국 디자인을 proc 구조체에 level 변수를 통한 논리적은 구조로 변경하여서 스케줄링을 진행하게 되었다.