

운영체제 Project2 - scheduler

2019019016 서시연

<디자인>

[FCFS]

1. 먼저 생성 (fork())된 프로세스가 먼저 스케줄링 되어야 한다.

- fork에서 allocproc()을 호출

```
// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
int
fork(void)
{
    int i, pid;
    struct proc *np;
    struct proc *curproc = myproc();

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

- allocproc()을 보면 프로세스의 pid = nextpid++; 이므로 먼저 생성된 프로세스의 pid가 더 작은 것을 알 수 있다.

```
//PAGEBREAK: 32
// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
static struct proc*
allocproc(void)
{
    struct proc *p;
    char *sp;

    acquire(&table.lock);

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

    release(&table.lock);
    return 0;

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

=> pid가 작은 프로세스를 먼저 스케줄링 해야한다.

2. 기본적으로 스케줄링된 프로세스는 종료되기 전까지는 switch-out되지 않는다.

- 항상 pid가 가장 작은 프로세스를 선택해서 실행하면, 해당 프로세스가 SLEEPING 상태가 되는 등의 예외 상황이 아니면 그 프로세스가 종료되기 전까지 그 프로세스만 선택되므로 위의 조건이 지켜진다.

3. 프로세스가 스케줄링된 이후 200ticks가 지날때까지 종료되거나 SLEEPING 상태로 전환되지 않으면 강제 종료 시켜야 한다.

- 프로세스가 처음 스케줄링 되었을 때의 ticks를 저장하고 매번 현재 ticks - 저장한 ticks를 계산해 200이 넘으면 강제종료 시킨다.

- 프로세스가 처음 스케줄링 되었을 때의 ticks를 저장하기 위해 struct proc 에 int stick 을 추가한다.

- stick은 allocproc()의 pid를 처음 셋팅하는 부분에서 0으로 초기화한다.

- 또한 프로세스가 SLEEPING 상태이면 stick을 0으로 초기화한다.

- 프로세스가 스케줄링 되었을 때 (context switch를 위해 선택됐을 때) stick이 0이면 stick에 ticks를 저장해준다.

- pid를 받아 pid에 해당하는 프로세스를 kill하는 kill()을 보면 해당 프로세스의 killed를 1로 바꿔준다.

```
// Kill the process with the given pid.
// Process won't exit until it returns
// to user space (see trap in trap.c).
int
kill(int pid)
{
    struct proc *p;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            p->killed = 1;
        }
    }
}
```

-> 200tick동안 종료되거나 sleeping 상태로 전환되지 않은 프로세스의 killed를 1로 바꿔서 강제 종료시켜준다.

4. 실행중인 프로세스가 SLEEPING 상태로 전환되면 다음으로 생성된 프로세스가 스케줄링 되며, SLEEPING 상태이면서 먼저 생성된 프로세스가 깨어나면 다시 그 프로세스가 스케줄링 된다.

- state가 RUNNABLE인 프로세스 중 가장 작은 pid를 갖는 프로세스를 실행하면, 먼저 생성된 프로세스가 깨어나면 SLEEPING에서 RUNNABLE로 상태가 바뀌기 때문에 위와 같은 조건이 만족된다.

[Multilevel Queue]

1. pid가 짝수인 프로세스들은 round robin으로 스케줄링되며, pid가 홀수인 프로세스들은 FCFS(먼저 생성된 프로세스를 우선적으로 처리)으로 스케줄링 되며 pid가 짝수인 프로세스가 항상 먼저 스케줄링 되어야 한다.

- xv6의 기본 스케줄러는 round robin 방식이므로 아래의 for문에서 pid가 짝수이면 그대로 실행하고, p의 pid가 홀수인 경우에는 struct proc* 타입의 변수를 만들어, 실행되기 전에 pid값을 비교해 가장 작은 pid를 갖는 프로세스를 변수에 저장하고 continue;를 해서 실행되지 않게 한다.

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

2. SLEEPING 상태에 있는 프로세스는 무시해야 한다. (RR에 RUNNABLE인 프로세스가 없다면 FCFS 큐를 실행해야 함)

- 위의 xv6 기본 스케줄러를 보면, for문의 처음에 state가 RUNNABLE이 아니면 continue;를 해서 무시한다. 따라서 그대로 구현하면 위의 조건이 만족된다.

- int cnt; 를 선언한 뒤 for문에 들어가기 전에 0으로 초기화하고, for문에서 pid가 짝수여서 실행되었다면 cnt를 증가시킨다.

- 만약 For문이 종료된 이후에도 cnt가 0이면 pid가 짝수인 프로세스 중 가장 pid가 작은 프로세스를 실행시킨다.

[MLFQ]

1. L0, L1 두 개의 큐로 이루어져 있고, L0의 우선순위가 더 높으며, 처음 실행된 프로세스는 모두 가장 높은 레벨의 큐(L0)에 들어간다.
 - 모든 프로세스는 L0또는 L1에 포함되므로 struct proc에 int level을 추가해 L0에 포함되어 있는 프로세스는 level이 0, L1은 level을 1로 설정한다.
 - stick과 마찬가지로 allocproc()의 pid를 처음 셋팅하는 부분에서 level을 0으로 초기화한다.
2. L0 큐는 기본 RR 정책을 따르며 L0의 모든 프로세스가 SLEEPING 상태가 되면 L1 큐의 프로세스들을 실행한다.
 - Multilevel Queue에서처럼 기본 스케줄러의 for문을 똑같이 사용해서 프로세스의 level이 1이면 continue;를 해서 실행되지 않게 한다.
3. L1 큐는 priority scheduling을 한다.
 - 각 프로세스의 priority를 저장하기 위해 struct proc에 int priority를 추가한다.
 - 2.에서 level이 1일때 continue를 하기 전, priority를 비교해 priority 값이 가장 큰 프로세스를 찾는다. (priority가 같은 경우 pid가 더 작은 프로세스를 선택)
 - level과 마찬가지로 allocproc()의 pid를 처음 셋팅하는 부분에서 priority를 0으로 초기화한다.
4. 프로세스는 기본적으로 각 레벨에서 주어진 quantum동안 실행한 후에 switch-out된다.
 - level 0의 경우 기본 스케줄러의 for문 안에 for문을 만들어 4tick동안 실행한다.
 - 예외 상황(sleep, yield등)에는 한번에 quantum만큼 실행되지 않을 수도 있으므로 한 번 실행될 때마다 struct proc의 stick을 1씩 증가시켜서, 4번 실행한 뒤에 L1 큐로 내려갈 수 있게 한다.
 - 만약 유저가 시스템콜 yield(), monopolize()를 호출했다면 for문을 빠져나간다.
 - stick이 quantum과 같아지면 level을 0으로 바꾸고 stick을 0으로 초기화한 뒤 break로 for문을 빠져 나간다.
 - L0에 RUNNABLE 상태인 프로세스가 없으면 L1에 있는 우선순위가 가장 높은 프로세스를 for문을 통해 8tick동안 실행한다.
 - level 0의 경우와 동일하게 동작
5. priority boosting
 - int tick을 만들어, 매 tick마다 1씩 증가시킨 뒤, tick이 200이상이 되면 모든 프로세스의 level을 0으로 바꿔주고, priority와 stick을 0으로 리셋해준다.
6. MLFQ 스케줄링을 무시하고 프로세서를 독점해서 사용할 수 있게 하는 시스템콜 monopolize()가 추가되어야 한다.
 - 인자로 받은 암호가 학번과 일치하지 않으면 현재 프로세스의 killed를 1로 바꿔주어 강제 종료시키고, 메시지를 출력한다.

- scheduler와 monopolize를 구현할 proc.c에 전역변수 struct proc* mono;를 선언한다. 처음에 mono를 0으로 초기화한다.
- 암호가 학번과 일치하면 mono가 0인지 확인하고 0인 경우 해당 프로세스를 mono에 저장해 준다. mono에 프로세스가 저장되어 있는 경우, 해당 프로세스가 monopolize를 한 번 더 호출한 것이므로 해당 프로세스의 priority와 level을 0으로 바꿔주고, mono를 0으로 다시 초기화한다.
- scheduler에서 MLFQ 스케줄링을 하기 전에 mono가 0인지 확인하고, 0이 아니라면 mono를 실행한다.

<구현>

[struct proc]

- int stick
 - FCFS에서는 해당 프로세스가 (SLEEPING상태에서 깨어나) 처음 스케줄링 되었을 때의 ticks를 저장
 - MLFQ에서는 해당 프로세스가 quantum만큼 실행되었는지 확인하기 위해 실행된 횟수를 저장
- int priority
 - MLFQ의 L1에서 priority scheduling을 하기 위해 priority를 저장
- int level
 - MLFQ에서 L0, L1 중 어떤 큐에 속한 프로세스인지 판단하기 위한 값, L0에 속한 경우 0, L1에 속한 경우 1이다.
- int yield
 - MLFQ에서 quantum만큼 실행한 뒤 switch-out하기 위해, for문을 통해 quantum만큼 해당 프로세스를 실행하는데, 이때 이 프로세스가 유저 프로그램에서 시스템콜 yield()를 호출하면 다음 프로세스에 CPU를 넘겨줘야 하므로, 유저 프로그램에서 시스템콜 yield()를 호출했는지 확인하기 위한 값. (1인 경우 호출한 것)
 - sysproc.c에 정의된 sys_yield()에서 myproc()->yield를 1로 바꿔준다.

```
struct proc {
    uint sz;
    pde_t* pgdir;
    char *kstack;
    enum procstate state;
    int pid;
    struct proc *parent;
    struct trapframe *tf;
    struct context *context;
    void *chan;
    int killed;
    struct file *ofile[NOFILE];
    struct inode *cwd;
    char name[16];

    int stick;
    int priority;
    int level;
    int yield;
};
```

[allocproc()]

- 해당 프로세스의 stck, priority, level, yield를 0으로 초기화한다.

```

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

    acquire(&table.lock);

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

        release(&table.lock);
        return 0;
    }

found:
    p->state = EMBRYO;
    p->pid = nextpid++;
    p->stick=0;
    p->level=0;
    p->priority=0;
    p->yield=0;

    release(&table.lock);
}

```

[struct proc *mono]

- proc.c 에 전역변수 mono 를 선언하고 0 으로 초기화한다.

[FCFS]

- #ifdef FCFS_SCHED
- for문으로 ptable에 있는 모든 프로세스를 돌면서, firstProc와 p의 pid를 비교해 pid가 더 작은 것을 firstProc에 저장한다.
- 만약 프로세스의 state가 SLEEPING이면 해당 프로세스의 stick을 0으로 초기화 한다.
- 프로세스의 state가 RUNNABLE이 아니면 continue; 한다.
- for문이 끝나면 firstProc (RUNNABLE인 프로세스 중 pid가 가장 작은 프로세스)를 실행한다.
- 이때, 해당 프로세스의 stick이 0이면 (처음 스케줄링 된 경우) 현재 ticks를 저장해준다.
- stick이 0이 아니고, ticks - p->stick>=200이면 p->killed=1;을 실행해 프로세스를 강제 종료한다.

```

struct proc *firstProc=0;
for(p = table.proc; p < &table.proc[NPROC]; p++){
    // if proc is SLEEPING -> reset stick
    if(p->state == SLEEPING) p->stick=0;

    if(p->state != RUNNABLE)
        continue;
    if(firstProc==0){
        firstProc=p;
    }
    else{
        if(p->pid < firstProc->pid){
            firstProc=p;
        }
    }
}
if(firstProc!=0){
    p=firstProc;
    c->proc = p;
    switchvm(p);
    p->state = RUNNING;
    // save ticks when proc in scheduler first
    if(p->stick==0) p->stick=ticks;

    switch(&(c->scheduler), p->context);
    switchkvm();

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

    // if proc run while more than 200ticks, kill
    if(p->stick!=0 && ticks-p->stick>=200){
        p->killed=1;
    }
}

```

[Multilevel Queue]

- for문으로 ptable에 있는 모든 프로세스를 돌면서, pid가 홀수인 경우(pid%2==1) FCFS에서와 마찬가지로 fcfs와 p의 pid를 비교해 pid가 더 작은 것을 fcfs에 저장하고 continue;한다.
- pid가 짝수인 경우, 해당 프로세스를 실행하고 cnt를 1 증가시킨다.
- for문이 종료된 후, cnt가 0이고, fcfs에 프로세스가 저장되어 있으면

```

// no RUNNABLE proc in RR (even)
// run least pid proc (odd)
if(cnt==0 && fcfs!=0){
    p=fcfs;

    c->proc = p;
    switchvm(p);
    p->state = RUNNING;

    switch(&(c->scheduler), p->context);
    switchkvm();

    c->proc=0;
}

```

```

cnt=0;
fcfs=0;
for(p = table.proc; p < &table.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
        continue;

    // find least pid proc (for odd pid)
    if(p->pid%2==1){
        if(fcfs==0) fcfs=p;
        else if(fcfs->pid > p->pid) fcfs=p;
        continue;
    }

    // run proc if pid == even
    c->proc = p;
    switchvm(p);
    p->state = RUNNING;

    switch(&(c->scheduler), p->context);
    switchkvm();

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

```

pid가 홀수인 프로세스 중 가장 pid가 작은 fcfs를 실행한다.

[MLFQ]

- scheduler 함수의 처음에 struct proc* L1, int tick, int cnt를 선언하고 0으로 초기화한다.
- tick이 200이상이면 priority_boosting()을 실행하고, tick에 0을 저장한다.
 - priority_boosting()에서는 for문을 돌면서 모든 프로세스의 level, priority, stick을 0으로 초기화 한다.
- mono가 0이 아니고, state가 RUNNABLE이면 해당 프로세스가 독점적으로 실행되어야 하므로, mono를 실행하고 tick을 1 증가시킨다.
- 위의 경우가 아니면 MLFQ로 동작한다.
- 매번 for문 시작 전에 cnt와 L1을 0으로 초기화
- for문으로 ptable에 있는 모든 프로세스를 돌면서, 프로세스의 level이 1이면 L1과 p의 priority, pid를 비교해 우선순위가 높은 프로세스를 L1에 저장한 뒤 continue;한다.
- 만약 프로세스의 level이 0이면, 기본적으로 quantum (4tick)동안 실행한 뒤 switch-out해야하므로, for문을 이용해 4번 실행되도록 한다.
 - 이때, monopolize가 실행되었거나, 유저가 yield 시스템 콜을 호출한 경우, break를 한다.
 - 그리고 check_time_quantum()을 호출해 프로세스의 stick이 quantum과 같은지 확인하고, 같다면(1이 리턴되면) break해준다.
 - check_time_quantum()은 넘겨받은 프로세스의 stick과 그 프로세스가 속한 level의 quantum을 비교해, 같으면 stick을 0으로 초기화하고, level이 0이면 1로 바꿔주고, 1이면 priority를 1감소시키고 1을 리턴한다. 같지 않으면 0을 리턴한다.
- for문을 빠져나온 뒤에도 cnt가 0이고, L1이 0이 아니라면, L1을 실행한다. 이때 위와 마찬가지로 quantum만큼 연속으로 실행하기 위해 for문을 이용한다.

```
cnt=0;
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
        continue;

    // find high priority proc in L1
    if(p->level==1){
        if(L1==0) L1=p;
        else if(L1->priority < p->priority) L1=p;
        else if(L1->priority==p->priority && L1->pid > p->pid) L1=p;
        continue;
    }

    // run proc for 4ticks
    for(int i=0; i<4; i++){
        if(p->state!=RUNNABLE) break;
        c->proc = p;
        switchvm(p);
        p->state = RUNNING;

        swtch(&(c->scheduler), p->context);
        switchvm();
        c->proc=0;

        // run time for proc
        p->stick++;

        // called monopolize() -> break
        if(mono!=0) break;
        // user call system call yield -> break
        if(p->yield==1){
            check_time_quantum(p);
            p->yield=0;
            break;
        }
        // stick == quantum -> break
        if(check_time_quantum(p)) break;
        tick++;
    }
    cnt++;
    if(mono!=0) break;
}
```

[Required system call]

- int getlev() : myproc()->level을 리턴
- int setpriority(int pid, int priority)
 - priority<0 || priority>10이면 -2를 리턴
 - myproc()->priority = priority
- void monopolize(int password)
 - password가 학번과 일치하고 mono가 0이면, mono=myproc()
 - password가 학번과 일치하고 mono가 0이 아니면 mono->priority=0, mono->level=0, mono=0
 - password가 학번과 일치하지 않으면 myproc()->killed=1
 - 이때 mono가 0이 아니면 mono=0

[FCFS]

```

$ ./p2_fcfs_test
FCFS test start

Without sleep & yield
process 4
process 4
process 4
process 4
process 4
process 5
process 5
process 5
process 5
process 5
process 6
process 6
process 6
process 6
process 6
process 7
process 7
process 7
process 7
process 7
process 8
process 8
process 8
process 8
process 8

With yield
process 9
process 9
process 9
process 9
process 10
process 10
process 10
process 10
process 10
process 10
process 11
process 11
process 11
process 11
process 11

process 11
process 11
process 11
process 11
process 11
process 12
process 12
process 12
process 12
process 12
process 13
process 13
process 13
process 13

With sleep
process 14
prprocess 14
ess process 14
1process 14
5process 14

prprocess 15
cess 16
procesprocess 15
prprocess 15
cess 16
$ 1process 15
process 16
7process 16

process 16
process 17
rprocess 17
cess 18
process 17
process 18
process 17
process 18
process 18
process 18

Infinite loop
ok
$ █

```

- without sleep & yield
 - pid가 작은 프로세스가 먼저 실행되므로 4, 5, 6, 7, 8 순서로 각각의 프로세스가 종료될 때까지 실행된다.
- with yield
 - 실행 중인 프로세스가 yield를 호출해서 CPU를 양보한 뒤, 다음 실행될 프로세스를 선택할 때도 pid가 가장 작은, 같은 프로세스가 선택되므로 9, 10, 11, 12, 13 순서로 실행된다.
- with sleep
 - 14번 프로세스가 먼저 실행되다가 sleep을 하면 다음으로 pid가 가장 작은 15번 프로세스가 실행되고, 14번 프로세스가 깨어나면 14번이 실행된다. 따라서 이웃한 pid의 프로세스들이 종료되기 전까지 번갈아가면서 실행된다.

- Infinite loop
 - 200tick이 지난 후 child 프로세스들이 강제 종료되고 child 프로세스들이 모두 종료되고 나면, parent process가 exit_child()에서 리턴되어 ok를 출력한다.

[Multilevel Queue]

```
$ time ./starting.sh
$ ./ml_test
Multilevel test start
[Test 1] without yield / sleep
Process 4
Process 4
Process 4
Process 4
ProProcess cess 4
P6
Process 6
Process 6
Process 4
ProcProcess 6
Prss 4
Process 6
rocessProcess 6
4
Process 4
Process 6
Process 4
ProcProess 4
Procecess 6
Process 6
Processss 4
Process 4
Proce 6
ss 4
ProcProcess 6
Process 6
ess 4
Process Process 6
Pr4
Process 6
Process 4
Process 4
rocess 6
Process Process 4
Process 6
Process 6
P4
rocess 6
Process 6
Process 6
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
```

- Test 1
 - pid가 짝수인 프로세스가 홀수인 프로세스보다 우선순위가 높으므로 짝수인 프로세스들의 실행이 모두 끝난 다음에 홀수인 프로세스가 실행된다.
 - 짝수인 프로세스들끼리는 RR이므로 서로 번갈아가면서 실행된다.
 - 홀수인 프로세스들끼리는 FCFS이므로 pid가 작은 프로세스의 실행이 끝난 다음에 다음 프로세스가 실행된다.
- Test2
 - yield()를 해도 마찬가지로 우선순위가 높은 pid가 짝수인 프로세스가 먼저 종료되고, 다음으로 홀수인 프로세스가 pid가 작은 순서로 종료된다.
- Test3
 - pid가 짝수인 프로세스가 모두 SLEEPING 상태가 되는 경

우가 있기 때문에 pid가 홀수인 프로세스도 번갈아면서 실행된다.

[MLFQ]

```
Enter selecting on
$ p2_mlfq_test
MLFQ test start

Focused priority
process 8: L0=1003, L1=18997
process 7: L0=1608, L1=18392
process 6: L0=1970, L1=18030
process 5: L0=2336, L1=17664
process 4: L0=2402, L1=17598

Without priority manipulation
process 9: L0=2535, L1=47465
process 10: L0=4030, L1=45970
process 11: L0=4658, L1=45342
process 12: L0=4820, L1=45180
process 13: L0=5117, L1=44883

With yield
process 14: L0=207, L1=9793
process 15: L0=398, L1=9602
process 16: L0=591, L1=9409
process 17: L0=784, L1=9216
process 18: L0=963, L1=9037

Monopolize
process 23: L0=25000, L1=0
process 19: L0=3143, L1=21857
process 20: L0=3311, L1=21689
process 21: L0=3447, L1=21553
process 22: L0=5070, L1=19930
$
```

- Focused priority

- 테스트 프로그램을 보면 프로세스가 생성된 순서대로 priority가 더 높게 설정되며 주기적으로 priority를 업데이트해준다. 처음 L0에서 4tick 실행된 이후, priority boosting이 있기 전까지는 모든 프로세스가 L1에 있기 때문에 priority가 가장 높은 순서대로 (pid가 큰 순서대로 종료된다)

- without priority manipulation

- 마찬가지로 모든 프로세스는 처음 L0에서 4tick 실행된 이후, priority boosting이 있기 전까지는 L1에 있는데, 모든 프로세스의 Priority가 0으로 같기 때문에 FCFS가 되어 pid가 작은 순서대로 종료된다.

- with yield

- 마찬가지로 우선순위가 높은 pid가 작은 순서대로 종료된다.

- Monopolize

- monopolize를 호출한 process 23이 가장 먼저 종료되며, 그동안 MLFQ가 작동하지 않기때문에 level은 모두 0이다. 나머지는 without priority manipulation과 같다.

<트러블슈팅>

- exit()가 프로세스를 강제 종료시키는 함수라고 생각해서 FCFS에서 200tick이 지나면 프로세스를 강제종료 시키기 위해 exit()에서 처럼 state를 ZOMBIE로 만들었는데, ok가 출력되지 않았다.
 - exit()에서는 ZOMBIE로 만든 후 sched()를 호출하길래 sched()를 호출해봤는데 trap14가 발생했다.
 - 넘겨받은 pid를 갖는 프로세스를 kill하는 kill()을 참고해 proc의 killed를 1로 설정하자 종료되었다.
- fcfs에서 for문을 돌고난 후 firstProc이 0인지 확인하지 않고 실행했을 때, lapicid 0 : panic: switchvm : no process에러가 발생 -> 프로세스를 실행하기 전 해당 프로세스가 0인지 검사