OS project1 - Implementing simple system schedulers

2021007138 서교빈

Design

💡 Outline of implementation approach

이번 프로젝트의 디자인 목표는 xv6에서의 scheduling policies를 구현하는 것이다.

xv6 시스템은 기본적인 Round-Robin scheduling을 사용한다. timer interrupt가 약 100 milliseconds 마다 발생하며 다음 process를 실행한다.

직접 구현해야 할 scheduler는 크게 2가지로 나뉜다.

1. FCFS

- 원래의 scheduler를 FCFS로 구현한다.
- 기본적인 first come, first served 방식으로 process가 완료되거나, 자발적으로 양보하기 전까지 CPU를 점유한다.

2. MLFQ

- L0, L1, L2 순서의 우선순위를 가진 3-level feedback queue로 구성된다.
- L0, L1 queue는 Round-Robin(RR) scheduling을 따르고, L0의 process를 우선적으로 할당한다.
- L2는 priority scheduling을 사용하며 priority는 0~3으로 구분된다. 초기값은 3으로 설정한다.
- setpriority system call을 호출하여 priority를 수정한다.
- L2의 p rocess 가 time quantum을 모두 사용하면 priority를 1 줄이고 time quantum을 초기화한다.
- 각각의 queue는 2i + 1 의 time quantum을 가진다.
- global tick이 50이 경과할 때마다 Priority boosting이 필요하다. **Priority boosting :** 모든 process를 L0으로 옮기고 time quantum과 priority를 초기화한다.

3. system calls 구현사항

`void yield(), int getlev(), int setpriority(), int mlfqmode(), int fcfsmode()`

Details

💡 Details of approach in code

우선 FCFS에서의 유의사항은 기존에 존재하는 Process Control Block에 새로운 변수 추가 없이 구현하는 것이다. FCFS는 가장 먼저 생성된 process가 반드시 우선순위를 가지므로 생성 시간을 구분할 수 있는 변수가 필요하다.

```
int
allocpid()
{
...
  pid = nextpid;
  nextpid = nextpid + 1;
  release(&pid_lock);

  return pid;
}
```

allocpid() 함수를 확인하면 pid를 부여할 때 1씩 증가하는 단순한 구조이므로 process의 pid를 기준으로 scheduler에서 최소 pid를 선택하도록 구현하고자 했다.

이 외의 특별한 요구사항은 없다.

MLFQ를 구현하기 위해선 proc 구조체에 대한 변수 추가가 필요하다.

```
struct proc {
...
int level;
  int priority;
  int tick;
};

struct circular_queue {
  struct proc *q[NPROC];
  int front;
  int rear;
  int size;
  int level;
  int timequantum;
};
```

각 process가 level과 priority를 가지며, CPU 제어권 획득 횟수를 저장하기 위해 tick을 추가했다. MLFQ는 각 큐마다 사용하는 policy가 다르기 때문에 level별로 process를 담기 위한 원형 큐 구조체를 채택했다. process의 maximum 수가 NPROC으로 정해져 있으므로 크기가 정해진 배열로 접근이 용이하게 했다.

```
void //해당 proc 넣기
cq_push(struct proc *p)
{
  int l = p->level;
  if(mlfq[l].size == NPROC) return;
  mlfq[l].q[mlfq[l].rear] = p;
  mlfq[l].rear = (mlfq[l].rear + 1) % NPROC;
  mlfq[l].size++;
}

struct proc* //front pop
cq_pop(struct circular_queue *cq) {
  if (cq->size == 0) return 0;
  struct proc *p = 0;
```

```
  p = cq->q[cq->front];
  cq->front = (cq->front + 1) % NPROC;
  cq->size--;
  return p;
}

struct proc* //해당 proc 꺼내기
cq_ppop(struct circular_queue *cq, struct proc *p)
{
  if(cq->size == 0) return 0;
  for (int i = cq->front; i != cq->rear; i = (i + 1) % NPROC) {
    if(cq->q[i] == p) {
      for(int j = i; j != cq->rear; j = (j + 1) % NPROC) {
        cq->q[j] = cq->q[(j+1) % NPROC];
      }
      cq->rear = (cq->rear + NPROC - 1) % NPROC;
      cq->size--;
      break;
    }
  }
  return 0;
}
```

proc.c 파일에 queue와 관련된 함수를 추가 정의했다. queueinit, mlfqinit, push, pop 등 주로 scheduler에서 사용한다. L2 큐에서는 높은 priority를 가지는 process로 switch하므로 특정 process를 지정하는 ppop을 별도로 정의했다. push시 rear, pop시 front의 값을 조정하고 원형 큐로 정의했기에 NPROC으로 나눈 나머지를 인덱스로 활용한다.

```
  ...
np->state = RUNNABLE;
  if(mycpu()->sched_mode == MLFQ) {
    cq_push(np);
  } ...
  //userinit(), fork(), yield(), wakeup(), kill()
```

MLFQ에서 큐는 Ready Queue 간주한다. RUNNABLE state를 갖지 않는 process들은 따로 관리되지 않는다. process의 state를 RUNNABLE로 변화시키는 로직에 push를 추가해 해당 level에 위치하도록 한다. userinit(), fork(), yield(), wakeup(), kill()에 동일한 로직으로 구현했다.

```
  ...
if (c->sched_mode == MLFQ && global_ticks >= 50) {
      priority_boost();
      global_ticks = 0;
    } ...
```

priority boosting은 전체 tick이 50단위로 도달하면 수행한다. process내의 tick 외에 global tick을 저장하여 priority_boost가 호출되도록 한다.

```
  ...
// give up the CPU if this is a timer interrupt.
  if(which_dev == 2 && myproc() != 0) {
    if(mycpu()->sched_mode == MLFQ) {
      yield();
    }
  }
```

scheduling mode가 MLFQ일 때만 yield()를 호출한다. timer interrupt 발생 시마다 CPU scheduling를 하고 time quantum과 priority에 대한 관리도 함께 한다.

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

uint64
sys_getlev(void)
{
 // FCFS - 99, MLFQ - Q lev
 if(mycpu()->sched_mode == FCFS) {
   return 99;
 } else {
   struct proc *p = myproc();
   return p->level;
 }
}

uint64
sys_setpriority(void)
{
 int pid;
 int priority;
 argint(0, &pid);
 argint(1, &priority);
 return setpriority(pid, priority);
}

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

uint64
sys_mlfqmode(void)
{
 return mlfqmode();
}
```

총 구현해야 할 system call은 5개이다. 대부분 wrapper function의 모습을 갖고 proc.c에 함수를 정의했다. 예외적으로 getlev의 경우 모드별 level값을 반환한다.

```
 extern uint64 sys_yield(void);
extern uint64 sys_getlev(void);
extern uint64 sys_setpriority(void);
extern uint64 sys_fcfsmode(void);
extern uint64 sys_mlfqmode(void);
...
[SYS_yield]   sys_yield,
[SYS_getlev]  sys_getlev,
[SYS_setpriority] sys_setpriority,
[SYS_fcfsmode] sys_fcfsmode,
[SYS_mlfqmode] sys_mlfqmode,
//syscall.c

entry("yield");
entry("getlev");
entry("setpriority");
entry("fcfsmode");
entry("mlfqmode");
//usys.pl
```

kernel의 system call을 사용하기 위해 syscall.c에 mapping할 수 있도록 추가했다. user.h와 usys.pl 엔트리에 system call을 추가했다.

Code Implementation

💡 Key code modification and purpose

```
//proc.c

struct circular_queue mlfq[3];

void
queueinit(struct circular_queue *cq, int level)
{
 cq->front = 0;
 cq->rear = 0;
 cq->size = 0;
 cq->level = level;
 if(level == 0) cq->timequantum = L0_TQ;
 else if (level == 1) cq->timequantum = L1_TQ;
 else cq->timequantum = L2_TQ;
}

void
mlfqinit(void)
{
 for(int i =0; i < 3; i++)
   queueinit(&mlfq[i], i); //.....1
...
void
priority_boost(void) {
 int size_1 = mlfq[1].size;
 int size_2 = mlfq[2].size;
 struct proc* p;
 for (int i = 0; i < size_1; i++) {
   p = cq_pop(&mlfq[1]);
   p->level = 0;
   p->tick = 0;
   p->priority = 3;
   cq_push(p);
 }
 for (int i = 0; i < size_2; i++) {
   p = cq_pop(&mlfq[2]);
   p->level = 0;
   p->tick = 0;
   p->priority = 3;
   cq_push(p);
 }
 for (int i = 1; i < 3; i++) {
   mlfq[i].size = 0;
   mlfq[i].rear = 0;
   mlfq[i].front = 0;
 }
} //.....2
...
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;

 if (mycpu()->sched_mode == MLFQ) {
   p->level = 0;
   p->priority = 3;
```

```
    p->tick = 0;
  }
  else {
    p->level = -1;
    p->priority = -1;
    p->tick = -1;
  } //.....................................................................3
  ...

void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  struct proc *selected = 0;
  struct proc *next_proc = 0;
  c->sched_mode = FCFS;

  c->proc = 0;
  mlfqinit();
  for(;;){
    intr_on();

    if (c->sched_mode == MLFQ && global_ticks >= 50) {
      priority_boost();
      global_ticks = 0;
    }

    if(c->sched_mode == FCFS) {
      selected = 0;
      for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if(p->state == RUNNABLE) {
          if(selected == 0 || p->pid < selected->pid) {
            if(selected)
              release(&selected->lock);
            selected = p;
            continue;
          }
        }
        release(&p->lock);
      }

    if(selected) {
      selected->state = RUNNING;
      c->proc = selected;
      swtch(&c->context, &selected->context);
      c->proc = 0;
      release(&selected->lock);
    }

    else {
      intr_on();
      asm volatile("wfi");
        // nothing to run; stop running on this core until an interrupt.
    }
  } else { //MLFQ
      next_proc = 0;
      next_proc = cq_pop(&mlfq[0]);

      if(!next_proc) next_proc = cq_pop(&mlfq[1]);

      if (!next_proc) {
      int max_priority = -1;
      for (int i = 0; i < mlfq[2].size && next_proc == 0; i++) {
        p = mlfq[2].q[i];
        if(max_priority < p->priority) {
          max_priority = p->priority;
          next_proc = p;
        }
      }
      cq_ppop(&mlfq[2], next_proc);
    }

      if (next_proc) {
        acquire(&next_proc->lock);
        next_proc->state = RUNNING;
        c->proc = next_proc;
        swtch(&c->context, &next_proc->context);
        c->proc = 0;
        release(&next_proc->lock);
```

```
    }
    else {
      intr_on();
      asm volatile("wfi");
        // nothing to run; stop running on this core until an interrupt.
    }
   }
  }
} //.....4
...
int
setpriority(int pid, int priority) {
 if(priority < 0 && priority > 3) {
  return -2;
 }

 struct proc *p;
 for(p = proc; p < &proc[NPROC]; p++) {
  acquire(&p->lock);
  if(p->pid == pid){
   p->priority = priority;
   release(&p->lock);
   return 0;
  }
  release(&p->lock);
 }
 return -1;
}

int
fcfsmode(void) {
 struct proc *p;
 struct cpu *c = mycpu();

 if(c->sched_mode == FCFS) {
  printf("Already in FCFS mode\n");
  return -1;
 }

 c->sched_mode = FCFS;
 global_ticks = 0;

 // 모든 프로세스의 MLFQ 관련 변수 초기화
 for(p = proc; p < &proc[NPROC]; p++) {
  acquire(&p->lock);
  if(p->state == RUNNABLE || p->state == RUNNING || p->state == SLEEPING) {
   p->priority = -1;
   p->level = -1;
   p->tick = -1;
  }
  release(&p->lock);
 }
 return 0;
}

int
mlfqmode(void) {
 struct proc *p;
 struct cpu *c = mycpu();
 if(c->sched_mode == MLFQ) {
  printf("Already in MFLQ\n");
  return -1;
 }
 else {
  c->sched_mode = MLFQ;
  global_ticks = 0;
  for(p = proc; p < &proc[NPROC]; p++) {
   acquire(&p->lock);
    p->priority = 3;
    p->level = 0;
    p->tick = 0;
    if(p->state == RUNNABLE)
     cq_push(p);
   release(&p->lock);
  }
 }
 return 0;
} //.....5
...
void
yield(void)
{
```

```
  struct proc *p = myproc();
  acquire(&p->lock);
  p->state = RUNNABLE;
  if(mycpu()->sched_mode == MLFQ) {
    //context switch 이후 timequantum 증가
    if(++p->tick >= 2*p->level + 1) {
      if(p->level < 2) {
        p->level++;
        p->tick = 0;
      } //L2라면 priority 감소
      if(p->level == 2 && p->priority > 0) {
        p->priority--;
      }
    }
    cq_push(p);
  }
  sched();
  release(&p->lock);
} //.....6
```

1️⃣ queueinit과 mlfqinit을 추가해 MLFQ scheduling 시 사용할 queue를 생성한다.

2️⃣ priority boosting 시 L1, L2의 process를 초기화하고 L0에 삽입한다. L1, L2도 기본값으로 초기화한다.

3️⃣ allocproc에서 process 할당 시 FCFS와 MLFQ 모드에 따른 초기화를 진행한다.

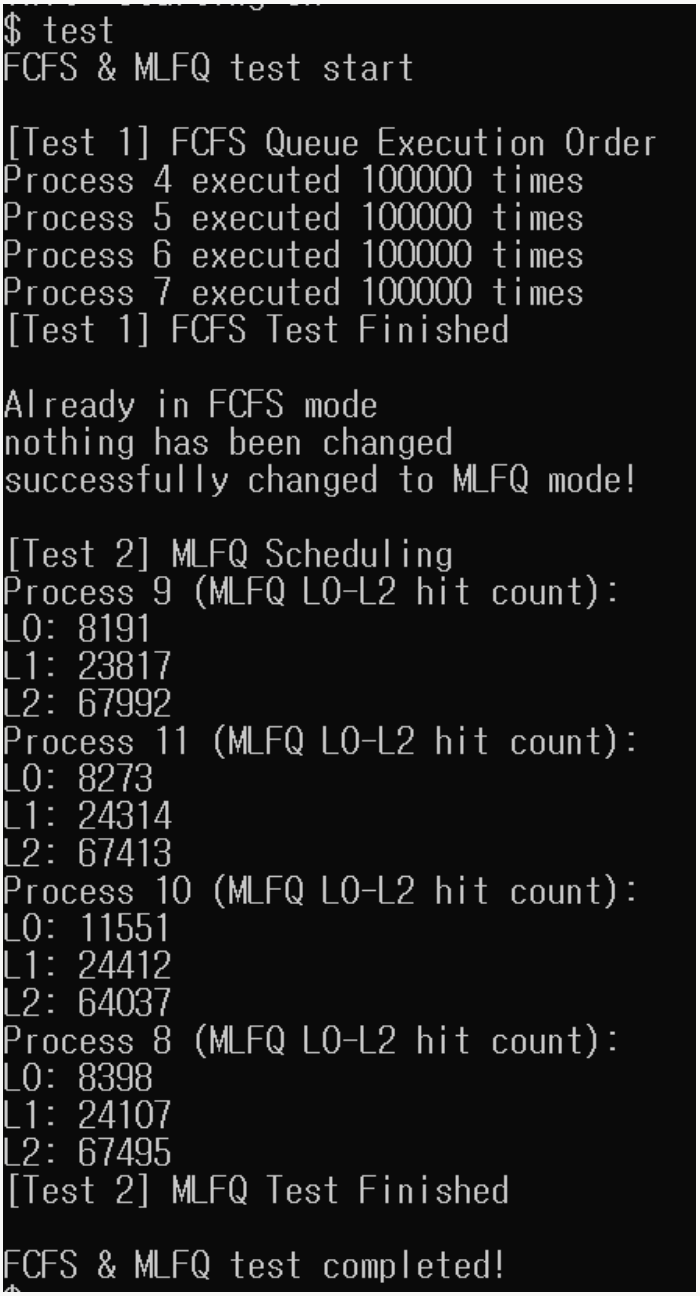
4️⃣ FCFS일 경우 pid가 작은 process를 찾아 context switch를 진행하고, MLFQ일 경우 low level의 queue부터 후보 process를 찾아 context switch를 한다.

5️⃣ setpriority는 pid값을 받아 결과를 반환한다. fcfsmode에서 MLFQ를 위해 추가한 변수의 값을 초기화한다. mlfqmode에서 모든 process의 변수를 초기화하고 RUNNABLE state의 process를 push한다. 각 mode switch마다 global tick(priority boosting 과정 적용)을 초기화한다.

6️⃣ yield 호출마다 running중인 process를 RUNNABLE로 바꿔 ready queue에 추가한다. time quantum과 priority를 조정해 push할 경우 기존 큐의 위치와 달라질 수 있다.

Results

💡 Compliation process, program flow and working code



Test 1은 FCFS 모드로 NUM_LOOP 횟수만큼 process가 실행되었다. Test 2 도입 시 queue 별로 hit count를 계산해 process마다 나타내었다.

TroubleShooting

💡 Encountered problem, applied solutions

❗MLFQ 구현 시 기존의 방법은 process가 alloc될 때마다 L0의 큐에 저장하고, RUNNABLE한 process 발생 시에도 큐에 저장해 scheduling 마다 RUNNABLE한 process를 찾는 작업을 추가로 진행했었다. mode switch, quantum time 계산 등 어려운 점이 많아 alloc하는 process 대신 RUNNABLE한 Process 발생 시마다 큐에 저장하는, 즉 Ready 상태의 process만 저장하여 L0, L1에서 탐색 과정없이 바로 process를 할당할 수 있었다.

❗MLFQ 스케줄링 시 반드시 낮은 level의 queue부터 우선탐색하며, L2의 queue의 priority까지 비교하는 로직을 구현하기 어려웠다. 위의 내용처럼 Ready queue로 구현하여 process 추출 과정을 간단히 하였고, queue가 비어있을 시 0을 반환해 상위 level의 queue의 if조건을 간단히 하도록 구현했다.