scheduler 과제

20181259 조수민

개발환경

2021 m1 맥북 프로 MacOs Monteray

i686-elf-gcc 사용

scheduling 함수 설명

struct proc *p : 다음에 실행할 프로세스를 탐색하여 그 주소값을 저장하는 포인터

sutruc cpu *c : cpu의 현재 정보를 가리키는 포인터

1. **sti()**
   - 무한루프에서 인터럽트를 허용한다.
2. **acquire(&ptable.lock);**
   - 프로세스 테이블의 락을 획득한다. 이 말은 동시에 이전의 프로세스가 실행이 끝나고 락을 반납했다는 뜻이다.
3. **for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){**
   if(p->state != RUNNABLE)
   continue;
   - 실행 가능(RUNNABLE)한 프로세스를 찾아 p가 가리키게 한다. RUNNABLE하지 않으면 다음 프로세스를 탐색한다.
4. **c->proc = p;**
   switchuvm(p);
   - cpu의 실행 프로세스를 탐색해서 찾은 p로 초기화 한다. switchuvm을 통해서 cpu 구조체의 여러 변수가 프로세스 가상 주소값을 참조할 수 있게 설정한다. 즉 컨텍스트를 전환해준다.
5. **p->state = RUNNING;**

- p의 상태를 RUNNING으로 바꾼다.

6. **switchkvm();**
   - 프로세스가 스케줄러로 돌아오면 커널모드로 전환한다.
7. **c->proc = 0;**
   - cpu가 실행하는 프로세스를 초기화 한다.
8. **release(&ptable.lock);**
   - 프로세스 테이블의 락을 반납하고 다시 무한루프로 들어간다.

priority와 시스템콜

proc.h

프로세스 구조체에 priority와 프로세스가 실행된 횟수 count 추가

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int priority;                // Process priority
  long count;                   // Process count
};
```

proc.c

set_proc_priority와 get_proc_priority 추가

```
// set_proc_priority 시스템 콜
int set_proc_priority(int pid, int priority)
```

```
{
  struct proc *p;
  int priority_return = -1;

  if(priority < 1 || priority > 10)
    return priority_return;
  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->pid == pid)
    {
      priority_return = p->priority;
      p->priority = priority;
      break;
    }
  }

  release(&ptable.lock);
  return priority_return;
}

// get_proc_priority 시스템 콜
int get_proc_priority(int pid)
{
  struct proc *p;
  int priority_return = -1;

  if(pid < 0)
    return priority_return;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->pid == pid)
    {
      priority_return = p->priority;
      break;
    }
  }

  release(&ptable.lock);
  return priority_return;
}
```

sysproc.c

시스템콜을 위한 래퍼 함수

```

int
sys_set_proc_priority(void)
{
```

```
  int pid;
  int priority;

  if(argint(0, &pid) < 0 || argint(1, &priority) < 0)
    return -1;
  return set_proc_priority(pid, priority);
}

int
sys_get_proc_priority(void)
{
  int pid;

  if(argint(0, &pid) < 0)
    return -1;
  return get_proc_priority(pid);
}
```

priority scheduler 구현

proc.c

```
void scheduler(void) {

  //스케쥴링 기준이 되는 상수
  const long priority_slice = 100;
  //프로세스 레디큐
  struct proc *readyQueue[NPROC];

  struct proc *p;
  struct proc *run_proc;
  struct cpu *c = mycpu();
  c->proc = 0;

  for (;;) {
    sti();
    acquire(&ptable.lock);

    //runnable한 프로세스를 큐에 추가함
    int length = 0;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if (p->state != RUNNABLE)
        continue;
      readyQueue[length++] = p;
    }

    //큐에 있는 프로세스를 priority 오름차순으로 정렬
    for (int i = 0; i < length - 1; i++) {
      for (int j = i + 1; j < length; j++) {
```

```
        if (readyQueue[j]->priority < readyQueue[i]->priority) {
          struct proc *temp = readyQueue[i];
          readyQueue[i] = readyQueue[j];
          readyQueue[j] = temp;
          }
      }
    }

    //레디큐 순서대로 실행
    for (int i = 0; i < length; i++) {
      run_proc = readyQueue[i];
        //실행 횟수는 slice(100)에서 중요도*10을 한만큼 뺀 값
        //즉 priority가 낮을수록(중요도가 높을수록) 많이 실행된다
        for(int j = 0; j < priority_slice-((run_proc->priority-1) * 10); j++)
        {
        if(run_proc->state != RUNNABLE)
          break;

        c->proc = run_proc;
        switchuvm(run_proc);
        run_proc->state = RUNNING;

        (run_proc->count)++;
        swtch(&(c->scheduler), run_proc->context);
        switchkvm();
      }
      c->proc = 0;
    }
  release(&ptable.lock);
  }
}
```

처음에는 count에 따라 priority를 낮추는 로직을 짰다. starvation은 예방할 수 있었지만, 무한루프로 테스트를 하다보니 어느 순간 이후로 중요도가 모두 1로 같아지게 되었다. **나는 count 혹은 시간에 따라 중요도를 낮추는 aging을 사용한 이 방법이 priority에 가깝다고 생각했는데, 지금 조건으로는 한계가 있었다.**

리팩토링한 방법은 사실 aging보다는 Round Robin을 베이스로 한 priority scheduling 이다. 한번의 스케쥴링 싸이클(레디큐를 생성)에 모든 runnable한 프로세스를 실행하게 된다. priority 오름차순으로 실행 순서가 먼저 배정되고, priority가 낮을수록 실행 횟수도 더 많다.

우려되는 점은 시간복잡도가 n^ 이라는 점이다. count와 priority를 더해서 중요도를 정하는 방식으로 시간복잡도 n으로 짤 수 있지 않을까 생각해봤지만, 기아문제를 더 철저하게 보완할 수 있다는 점에서 지금의 방식을 택하게 됐다.

test code

loop.c

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

//프로세스 루프 생성
//exec을 통해 실행
int loop()
{
  int pid = getpid();

  printf(1, "runing process:%d priority:%d\n", pid, get_proc_priority(pid));
  while (1)
  {
    //starvation 체크에 필요한 조건문 2000번 실행되면 sleep
    if(get_proc_count(pid) > 2000)
    {
      sleep(5000);
      break;
    }
  }
  exit();
}
```

test.c

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

int fork_process_with_priority(int priority)
{
  int pid = fork();
  if (pid > 0)
  {
    set_proc_priority(pid, priority);
  }
  else if (pid == 0)
  {
    char *argv[] = {"loop", 0};
    exec(argv[0], argv);
    exit();
  }
  return pid;
}

int default_priority_test()
{
  printf(1, "\ndefault priority test\n");
  printf(1, "priority의 디폴트 값이 5인지 테스트\n\n");
```

```
  //프로세스가 생성되자 마자 디폴트 우선순위를 확인
  int priority = get_proc_priority(getpid());
  if (priority == 5)
    printf(1, "default priority:%d\ndefault priority test success✅\n", get_proc_prior
ity(getpid()));
  else
    printf(1, "default priority:%d\ndefault priority test failed❌\n", get_proc_priori
ty(getpid()));
  printf(1, "\n----------------------------------------\n\n");

  return 0;
}

int fork_priority_test()
{
  printf(1, "fork priority test\n");
  printf(1, "fork한 프로세스의 우선순위가 부모 프로세스와 같은지 테스트\n\n");

  //부모 프로세스의 우선순위를 임의로(8)로 설정하고 fork
  set_proc_priority(getpid(), 8);
  int parent_priority = get_proc_priority(getpid());
  int pid = fork();
  if (pid == 0)
  {
    int child_priority = get_proc_priority(getpid());
    //자식 프로게스의 우선순위가 부모와 같은지 확인
    if (child_priority == parent_priority)
    {
      printf(1, "parent priority:%d\nchild priority:%d\n", parent_priority, child_prio
rity);
      printf(1, "fork priority test success✅\n");
    }
    else
    {
      printf(1, "parent priority:%d\nchild priority:%d\n", parent_priority, child_prio
rity);
      printf(1, "fork priority test failed❌\n");
    }
    exit();
  }
  wait();
  printf(1, "\n----------------------------------------\n\n");
  return pid;
}

int preemptive_test() {
  printf(1, "preemptive test\n");
  printf(1, "우선순위가 높은 프로세스가 실행되는지 테스트\n\n");

  //부모프로세스가 먼저 끝나지 않기 위해 우선순위를 높게 설정
  set_proc_priority(getpid(), 10);

    int num_processes = 10;

    int pids[num_processes];
```

```
    for (int i = 0; i < num_processes; i++) {
        pids[i] = fork_process_with_priority(num_processes - i);
    }
    sleep(100);

    process_info();

    for (int i = 0; i < num_processes; i++) {
        kill(pids[i]);
    }

    for (int i = 0; i < num_processes; i++) {
        wait();
    }

  printf(1, "우선순위인 프로세스부터 실행되는 지 확인, 우선순위대로 스케쥴링 횟수 확인\n");
  printf(1, "\n----------------------------------------\n\n");
  return 0;
}

int starvation_test() {
    printf(1, "starvation test\n");
    printf(1, "starvation을 예방할 수 있는지 테스트\n\n");

    // 부모 프로세스 다시 default priority로 설정
    set_proc_priority(getpid(), 5);

    int num_processes = 5;
    int pids[num_processes];

    // 2개의 프로세스 생성
    for (int i = 0; i < 2; i++) {
        pids[i] = fork_process_with_priority(2);
    }

    pids[2] = fork_process_with_priority(10);

    //우선순위가 더 낮은 프로세스 계속 생성
    for (int i = 0; i < 2; i++) {
        pids[i + 3] = fork_process_with_priority(2);
    }

    sleep(100);

    // 모든 프로세스 정보 출력
    process_info();

    // 모든 프로세스 종료
    for (int i = 0; i < num_processes; i++) {
        kill(pids[i]);
    }

    // 모든 프로세스 종료 대기
    for (int i = 0; i < num_processes; i++) {
        wait();
    }
```

```
    printf(1, "priority 10의 프로세스가 스케줄러에 의해 실행된 적이 있다면 starvation 없다는 의미
\n");
    printf(1, "\n----------------------------------------\n\n");

    return 0;
}

//starvation을 관측하기 위한 테스트 코드
//proc.c에 주석 처리해놓은 scheduling 방식으로 변경하고 실행하면 된다.

// int watch_starvation(){
//   set_proc_priority(getpid(), 5);

//  int pid1 = fork_process_with_priority(1);
//  //pid1을 runnable에 올리기 위해
//  sleep(10);

//  int pid2 = fork_process_with_priority(10);
//  int pid3 = fork_process_with_priority(1);
//  sleep(10);

//  process_info();
//  kill(pid1);
//  kill(pid2);
//  kill(pid3);
//  wait();
//  wait();
//  wait();

//  printf(1, "priority 10의 프로세스가 스케줄러에 의해 실행된 적이 없다면 starvation\n");
// }

int main()
{
  default_priority_test();

  fork_priority_test();

  preemptive_test();

  starvation_test();

  // watch_starvation();

  exit();
}
```

result

default && fork

```
default priority test
priority의 디폴트 값이 5인지 테스트

default priority:5
default priority test success✅

----------------------------------------

fork priority test
fork한 프로세스의 우선순위가 부모 프로세스와 같은지 테스트

parent priority:8
child priority:8
fork priority test success✅

----------------------------------------
```

preemptive(선점형)

```
preemptive test
우선순위가 높은 프로세스가 실행되는지 테스트

runing process:14 priority:1
runing process:13 priority:2
runing process:12 priority:3
runing process:11 priority:4
runing process:10 priority:5
runing process:9 priority:6
runing process:8 priority:7
runing process:7 priority:8
runing process:6 priority:9
runing process:5 priority:10

pid      name    state           priority        count
1        init    SLEEPING        5               12
2        sh      SLEEPING        5               15
3        test    RUNNING         10              14
5        loop    RUNNABLE        10              11
6        loop    RUNNABLE        9               21
7        loop    RUNNABLE        8               31
8        loop    RUNNABLE        7               41
9        loop    RUNNABLE        6               51
10       loop    RUNNABLE        5               61
11       loop    RUNNABLE        4               71
12       loop    RUNNABLE        3               81
13       loop    RUNNABLE        2               91
14       loop    RUNNABLE        1               205

우선순위인 프로세스부터 실행되는 지 확인, 우선순위대로 스케쥴링 횟수 확인

---------------------------------------
```

우선순위가 높은 프로세스가 먼저 실행되고, 실행 횟수도 더 많은 것을 볼 수 있다.

starvation

```
starvation test
starvation을 예방할 수 있는지 테스트

runing process:15 priority:2
runing process:16 priority:2
runing process:18 priority:2
runing process:19 priority:2
runing process:17 priority:10

pid      name    state          priority        count
1        init    SLEEPING       5               12
2        sh      SLEEPING       5               15
3        test    RUNNING        5               18
15       loop    RUNNABLE       2               180
16       loop    RUNNABLE       2               180
17       loop    RUNNABLE       10              10
18       loop    RUNNABLE       2               180
19       loop    RUNNABLE       2               180

priority 10의 프로세스가 스케줄러에 의해 실행된 적이 있다면 starvation 없다는 의미

---------------------------------------
```

priority가 10인 프로세스의 count횟수가 10회 있다는 것은 우선순위가 낮은 프로세스도 스케쥴링 된다는 것을 의미한다.

starvataion관측

```
$ test
runing process:4 priority:1
runing process:6 priority:1

pid      name    state          priority        count
1        init    SLEEPING       5               21
2        sh      SLEEPING       5               13
3        test    RUNNING        5               13
4        loop    SLEEPING       1               2002
5        test    RUNNABLE       10              0
6        loop    SLEEPING       1               2001

priority 10의 프로세스가 스케줄러에 의해 실행된 적이 없다면 starvation
```

스케쥴링 방식을 무조건 priority가 낮은 프로세스가 돌도록 변경한 후 진행했다.

priority가 1인 두 프로세스가 2000회 돌았지만 10인 프로세스는 돌지 않았다.

유틸 시스템콜

proc.c

```
//count값을 반환해주는 시스템콜
int get_proc_count(int pid)
{
  struct proc *p;
  int count_return = -1;

  if(pid < 0)
    return count_return;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->pid == pid)
    {
      count_return = p->count;
      break;
    }
  }
  release(&ptable.lock);
  return count_return;
}


//프로세스의 정보를 출력한다.
int process_info(void) {
  struct proc *p;
  char *state;

  acquire(&ptable.lock);
  cprintf("\npid\t name\t state\t\t priority\t count\n");
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    switch (p->state) {
      case SLEEPING:
        state = "SLEEPING";
          break;
      case RUNNABLE:
        state = "RUNNABLE";
          break;
      case RUNNING:
        state = "RUNNING";
          break;
      default:
        state = 0;
      }
    if (state)
    cprintf("%d\t %s\t %s\t %d\t\t %d\n", p->pid, p->name, state, p->priority, p->coun
```

```
t);
  }
  cprintf("\n");
  release(&ptable.lock);
  return 0;
}
```