Operating Systems, Project 6

2013-11392 김지현

개요

이번 과제는 커널 전용 스레드를 구현한 뒤, preemptive priority scheduling을 구현하는것이다. 더불어 멀티코어 환경을 무시할 수 있었던 project 4, 5와는 다르게 멀티코어 환경을 지원 해야만 한다. 문제 description에 주어진 상황들을 모두 잘 고려해 코너케이스를 놓치지 않도록 구현하는 것 뿐만 아니라, 멀티코어로 인해 발생할 수 있는 동시성 문제들도 모두 함께 처리해줘야만 하는 어려운 프로젝트이다.

구현 설명

언제 Yield 해야하는가?

이 과제에서 구현해야 하는 스케줄러는 Preemptive priority scheduler로, 매 순간 “effective priority가 가장 높은 스레드만이 실행되고 있어야 한다”는 가정을 유지해줘야 한다. 이는 상당히 강력한 invariant로 한곳에서라도 실수를 하면 쉽게 가정이 깨질 수 있다.

현재 xv6 구현상, 위 invariant를 지키기 위해 yield를 직접 호출해야하는 경우는 아래와 같다.

- 우선순위가 더 높은 새 커널스레드가 생성되었을 경우
  기존에 실행중이던 커널스레드보다 더 높은 우선순위를 가질 수 있다.
- 실행중이던 스레드의 우선순위가 낮아졌을 경우
  현재 스레드가 더 이상 “effective priority가 가장 높은 스레드”가 아니게 될 수 있다.
- Sleeplock을 release했을 경우
  Sleeplock을 release하는것이 현재 스레드의 effective priority를 낮출 수 있다.

위 세가지 경우만 고려하면 된다. 예를들어 실행중이지 않던 스레드의 우선순위가 변하는 경우는 고려하지 않아도 된다. 문제 스펙상 스레드의 우선순위가 변하는 경우는 단 두개뿐인데

1. kthread_set_priority() 함수가 호출됨
2. 다른 스레드가 이 스레드에 우선순위를 donate함

2번의 경우 priority donation이 발생했다는 것은, 다른 실행중이던 스레드가 sleep에 들어갔다는 뜻이여서 어차피 스케줄러가 호출되기 때문이다. 따라서 1번 케이스만 고려해도 된다.

위와 같이 구현할 경우 sleeplock이 릴리즈되는 때에도 가장 우선순위가 높은 스레드가 실행된다는 것이 보장된다. 이후 구현할 때 위 세가지경우에서 yield 호출을 잊지 않도록 주의해야한다.

함수 kthread_create() 구현

가장 먼저 kthread_create() 함수를 구현했다. kthread_create() 함수는 새로운 커널 전용 스레드를 만들어주는 함수이다. 문제 description에 주어진것과 같이, 이미 있는 proc 구조체를 활용하되 커널 전용 스레드에 필요 없는 기능들은 빼는 방식으로 구현하였다.

커널 전용 스레드가 일반 프로세스와 달라지는 부분은 아래와 같다.

- 커널 스택 메모리만 갖고있고, 유저 프로세스와 같이 전용 코드영역이나 별도의 힙영역을 갖고있지 않다. 따라서 p->sz = PGSIZE이다.
- 커널 메모리 공간에서 실행된다. 따라서 p->pagetable = kernel_pagetable 이다.
- 별도의 트랩프레임을 갖고있지 않다. 트랩프레임은 유저영역에서 실행되던 코드가 트랩을 만나 커널영역으로 넘어올 때 유저 레지스터들을 저장하는 용도로 사용되는 영역인데, 커널모드에서만 실행되므로 트랩프레임이 필요 없다.
- forkret() 함수에서 실행되는 일반 프로세스들과는 달리, “kthread_entrypoint”라는 이름의 커널 스레드 전용 특수 함수에서 실행되기 시작한다.
- Priority가 120으로 고정되어있는 유저 프로세스들과는 다르게, kthread_create 함수의 인자로 주어진 priority 값을 갖고 시작한다.

함수 kthread_entrypoint()는 커널 스레드를 시작하기 위한 특수한 함수로, 새로 시작되는 커널 스레드에 argument를 전달하기 위해 존재한다.

Struct proc 안에 커널 컨텍스트 스위치시 레지스터들을 보존하기 위한 struct context가 이미 존재하지만, struct context는 RA(return address), SP(stack pointer) 레지스터와 S0-11 레지스터만을 보존시키는 반면, RISC-V 콜링컨벤션상 함수 인자는 A0 레지스터로 전달해야해서 struct context로는 새로 시작하는 커널스레드에 인자를 넘겨줄 수 없다. 이 때문에 함수 kthread_entrypoint()가 필요하며, 아래와 같이 struct proc에서 엔트리포인트 주소와 함수 인자를 읽어와 호출하는 방식으로 구현되어있다.

```
static void
kthread_entrypoint(void)
{
  // 맨 처음 스케줄링되면, 프로세스가 락된 상태임
  struct proc *p = myproc();
  release(&p->lock);

  // Jump into the entrypoint, never to return.
  p->entry(p->entry_arg);
  panic("Kernel thread finished without calling kthread_exit()");
}
```

Figure 1. Function kthread_entrypoint()

이 때, kthread_entrypoint 함수가 맨 처음 실행되었을때는 스케줄러가 프로세스를 실행한 직후로, 프로세스가 lock되어있어 이를 풀어줘야 함을 유의해야한다.

kthread_entrypoint 함수를 구현하기 위해, struct proc 안에 커널 스레드의 코드 시작지점 주소와 첫번째 파라미터를 저장할 필요가 있어 아래와 같이 struct proc을 확장시켜주었다. 더불어, 커널 스레드가 갖는 priority 정보도 함께 저장하였고, 일반 프로세스와 커널스레드를 구분하기위한 필드인 is_kernel_thread도 추가하였다. Is_kernel_thread 필드는 이후에 스케줄러를 구현할 때 요긴하게 사용된다.

```
kernel/proc.h
-----------------------------------------------------------------------------
 struct proc {
93
   int killed;                  // If non-zero, have been killed
   int xstate;                  // Exit status to be returned to parent's wait
   int pid;                     // Process ID
   int is_kernel_thread;
   int base_prio;

   // these are private to the process, so p->lock need not be held.
   uint64 kstack;               // Bottom of kernel stack for this process
 struct proc {
105
   struct file *ofile[NOFILE];  // Open files
   struct inode *cwd;           // Current directory
   char name[16];               // Process name (debugging)
   void (*entry)(void *);
   void *entry_arg;
 };
```

Figure 2. New fields of struct proc

커널 스레드 생성이 끝나면, 앞 문단에서 논의하였듯이 새로 만들어진 스레드가 현재 스레드보다 우선순위가 더 높은지 검사하고, 더 높을경우 yield 하는것을 잊지 말아야한다.

kthread_exit() 구현

함수 kthread_exit()에서는 kthread_create()의 역순으로 커널 스레드의 내부 정보를 해제하면 된다. 특별히 할 일은 없고, struct proc 내부를 0으로 초기화하면 끝난다. 이 때 kthread_exit()은 일반 프로세스 exit() 과는 다르게, 종료되었을 때 프로세스가 ZOMBIE state가 되는 것이 아니라 곧바로 UNUSED state가 되어야 한다. 이 커널 스레드의 xstate를 읽어서 ripping해줄 부모가 존재하지 않기때문이다.

프로세스를 모두 초기화 한 뒤 맨 마지막에 sched() 함수를 호출해 다른 프로세스를 실행하면 kthread_exit()의 구현이 완료된다.

kthread_yield() 구현

커널 스레드는 일반 프로세스와는 다른 객체이긴 하지만, yield는 이미 정의되어있는 yield() 함수를 그대로 호출해도 아무 문제가 없다. Kthread_yield에서 할 일은 현재 struct proc을 잠그고 state를 RUNNABLE로 바꾼 뒤 sched() 함수를 부르는것인데, 기존 yield() 함수와 아무 차이가 없기 때문이다.

Kthread_set_priority() 구현

우선순위를 바꾸는 일 자체는 현재프로세스를 잠근 뒤 p->base_prio 필드를 변경하면 끝으로, 전혀 어렵지 않다. 신경써야하는것은 우선순위가 낮아졌을 때 yield하는것인데, 문제 지문에 쓰여있듯이 base priority가 아니라 effective priority가 낮아졌는지 비교해야한다.

```
void
kthread_set_prio(int newprio)
{
  struct proc *p = myproc();
  acquire(&p->lock);

  int before = kthread_get_prio_of_locked(p);
  p->base_prio = newprio;
  int after = kthread_get_prio_of_locked(p);

  release(&p->lock);

  if (after > before) {
    kthread_yield();
  }
}
```

Figure 3. Source code of kthread_set_prio()

여기서 특기할점은, Figure 3에서 볼 수 있듯이 kthread_set_prio()가 effective priority 비교를 위해 곧바로 kthread_get_prio() 함수를 호출하지 않았다는 점이다.

Struct proc의 base_prio 필드는 kthread 함수들과 스케줄러 등 여럿이 동시에 읽고 쓸 가능성이 있는 필드여서 락을 잡아줘야한다. 이 때 kthread_set_prio() 함수에서도 프로세스 락을 잡는데 그 안에서 kthread_get_prio() 함수를 호출할 경우 데드락이 발생하기 때문이다.

이러한 순환참조를 막기 위해, kthread_get_prio_of_locked() 라는 함수를 새로 만들었다. 이 함수는 "이미 프로세스에 락이 잡혔다고 가정하여 추가적인 락을 잡지 않는 함수"여서, 지금과 같은 경우에 사용하면 용이하다.

kthread_get_prio(), kthread_get_prio_of_locked() 함수 구현

위에서 kthread_get_prio_of_locked 함수를 새로 만들었기 때문에, kthraed_get_prio() 함수는 단순히 프로세스의 락을 잡은 뒤 kthread_get_prio_of_locked() 함수를 호출하는 래퍼가 된다.

kthread_get_prio_of_locked 함수가 사실상 이번 과제의 핵심인데, 아래와 같이 재귀적인 구조로 만들면 된다.

1. 현재 프로세스가 acquire중인 락이 있는지 검사한다.
2. 현재 프로세스가 acquire중인 락이 없다면, base_prio를 리턴하고 끝낸다.
   현재 프로세스가 acquire중인 락이 있다면, 3으로 진행한다.
3. 현재 프로세스가 acquire중인 모든 락 L에 대해 아래 작업을 수행한다.
   a. L을 acquire하지 못해 sleep중인 모든 프로세스 p들의 목록을 만든다.
   b. 모든 p들의 effective priority를 **kthread_get_prio_of_locked 함수를 재귀호출하여** 얻어낸다.
   c. 이중 최소값 X를 얻어낸다.

   a~c를 를 수행하면 락별로 값 X가 얻어지는데, 모든 X들의 최소값인 min_prio를 찾는다.
4. min_prio를 반환한다.

위와 같은 방식으로 구현하면 방문했던 프로세스를 여러 번 지속적으로 방문하게되어 시간복잡도가 상당히 좋지 못하다. 그래프나 힙과 같은 자료구조를 동원하면 시간복잡도를 개선시킬 수 있는 여지가 많지만, 이는 이 과제의 범위를 넘어간다고 생각되어 일단은 위와 같은 brute force 알고리즘으로 작성하였다.

이렇게 구현하면 priority donation이 아무 변수도 수정하지 않는 immutable한 방식으로 구현되기 때문에, 수많은 골치아픈 invariant 문제에서 해방될 수 으며, multiple priority donation과 nested priority donation도 재귀적으로 자동으로 구현된다.

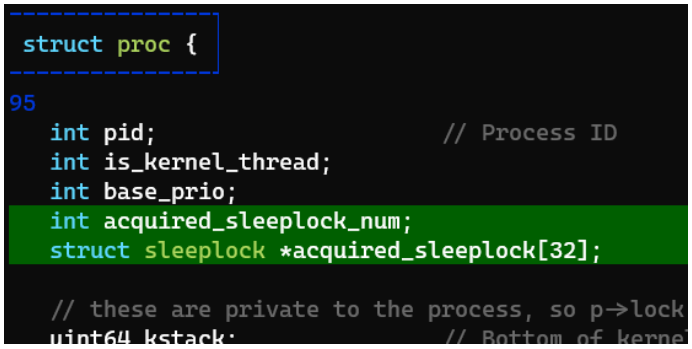

Figure 4. Dynamic array, acquired_sleeplock

여기서 문제는 현재 프로세스가 acquire중인 락이 무엇인지 알 방법이 없다는것인데, 이를 위해 struct proc을 Figure 4와 같이 추가로 확장해야한다. “acquire_sleeplock”은 해당 프로세스가 acquire중인 슬립락들의 목록을 저장하는 배열로, 원소의 개수가 acquire_sleeplock_num 필드에 저장되어있다. 동적할당을 잘 활용하면 길이제한이 없는 버전으로도 구현할 수 있으나, 이는 본 과제의 범위를 넘어간다고 판단해 생략하였다.

Acquiresleep() 함수와 releasesleep() 함수에서 락을 얻고 해제함에 따라 Figure 5, 6과 같이 acquire_sleeplock 배열에 락을 리스트에 적절히 추가하고 삭제하면 된다.

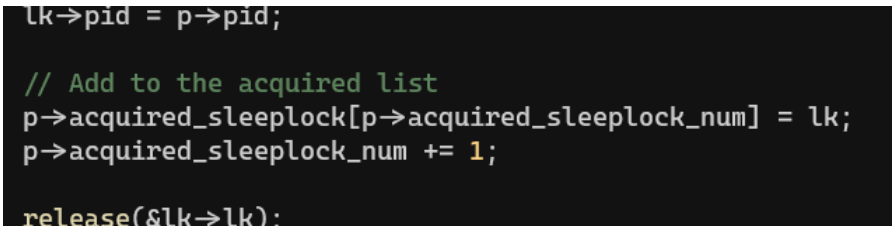

Figure 5. Inserting new acquired lock at acquiresleep()

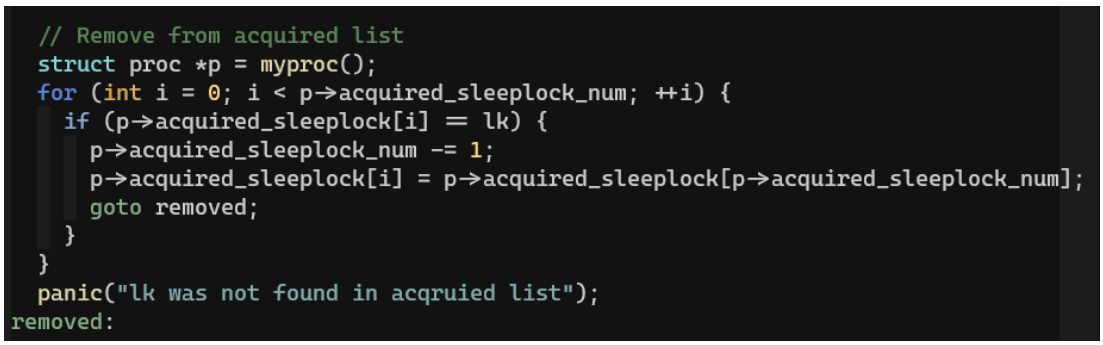

Figure 6. Remove released lock at releasesleep()

또한 앞선 문단에서 논의하였듯이, releasesleep() 함수 맨 마지막에서 yield를 하여야 스케줄러가 올바르게 동작한다.

이런 방식으로 구현할경우, acquiresleep() 함수와 releasesleep() 함수는 상당히 간단한 구조를 가진다.

Acquiresleep():

- 락을 acquire하려고 시도하고, 잠겨있을 경우 sleep하는 루프로 들어간다. (기존 acquiresleep 구현 그대로)
- Acquire에 성공할경우, 락에 관련 정보를 기입한다 (기존 acquiresleep 구현 그대로)
- **현재 프로세스의 acquire_sleeplock 리스트에 이 락을 추가한다**

Releasesleep():

- **현재 프로세스의 acquire_sleeplock 리스트에서 이 락을 뺀다**
- 이 락을 대기하며 잠든 프로세스를 모두 깨운다 (기존 releasesleep 구현 그대로)
- **yield한다**

이렇게 구현할경우 가장 우선순위가 높은 스레드가 먼저 실행되기 때문에, 슬립락이 정상적으로 동작하며 우선순위가 가장 높은 스레드가 락을 가져가게된다.

복잡한 로직은 모두 kthread_get_prio_of_locked 함수에 들어있기 때문에 이쪽 로직은 복잡할것이 없다.

스케줄러 구현

앞선 문단에서 논의한것과 같이 yield를 모두 적절히 호출한다면, 스케줄러에서 할 일은 매우 단순하다. 아래의 알고리즘에 따라 동작하면 된다.

1. 가장 마지막에 스케줄링된 프로세스의 index 번호를 뜻하는 “last_idx” 변수의 값을 0으로 초기화한다.
2. 모든 Runnable한 프로세스를 순회하여, effective priority의 최소값인 min_prio가 몇인지 구한다.
    a. Runnable한 프로세스가 한 개도 없을경우, 다시 2를 수행한다.
3. Proc[] 배열을 “last_idx + 1”부터 순회하며, min_prio와 동일한 effective priority 값을 가진 Runnable 프로세스를 찾아 스케줄링한다.
    a. 찾지 못했을 경우, 2로 돌아간다.
4. 찾은 프로세스의 인덱스 번호로 last_idx 변수를 업데이트한 뒤, 이 프로세스를 실행한다.
5. 2~4를 무한반복한다.

이렇게 구현하면 시간복잡도가 좋지 못하고, 힙과 같은 자료구조를 동원할 경우 최적화의 여지가 있다. 그러나 이 역시 과제의 범위를 넘어간다고 판단하여 생략하였다.

락만 잘못잡지 않도록 주의하면 어렵지 않게 스케줄러 구현을 마칠 수 있다.

실험 결과

보너스 문제를 포함해, 모든 주어진 테스트케이스를 성공적으로 통과하였다.

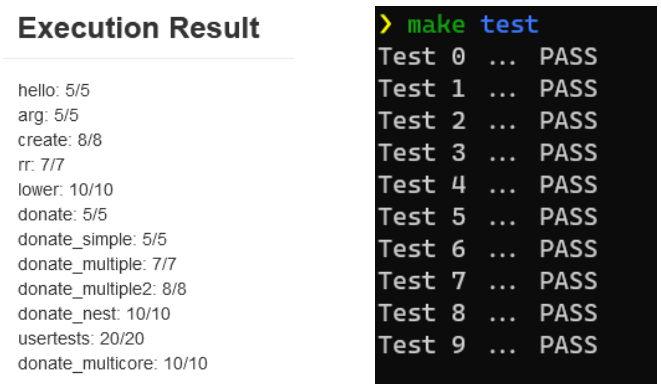

Figure 7. Test result

주어진 테스트케이스 이외에도 간단한 테스트를 몇가지 만들어 동작시킨 결과, 성공적으로 실행됨을 확인할 수 있었다.

결론

복잡한 알고리즘을 구현하며 동시에 동시성문제까지 신경써야해서 상당히 난이도있는 과제였다. 그러나 실제 운영체제에 필요한 기능들을 직접 만들고 동작시키면서, OS 프로그래밍에 대한 상당한 자신감을 얻을 수 있었다. Priority donation이 어떤것인지 익히고, 운영체제를 개발하며 만날 수 있는 다양한 코너케이스들을 처리하며 경험을 쌓을 수 있었던 좋은 프로젝트였다.