Operating System

Project 6

2014-15703 송화평

1. 프로젝트 요약

본 프로젝트의 목적은 항상 커널 공간에서만 동작하는 커널 스레드를 구현하고, 이 커널 스레드에게 우선순위를 부여하기 위한 preemptive priority scheduling 또한 구현하며, 마지막으로 preemptive priority scheduling에 존재하는 문제점인 priority inversion을 방지하기 위해 priority donation을 적용하는 것이다. Preemptive priority scheduling 시 커널 스레드는 [0, 100), 유저 스레드는 [100, 140)의 priority value를 가지게 되며, 낮을 수록 더 높은 priority를 가진 것으로 간주하여 동작하게 된다. 이 때 priority value가 가장 낮은 스레드를 택하되, 그러한 스레드가 여러 개 있으면 그러한 스레드끼리는 Round-Robin 방식으로 스케쥴링하였다. 또한 높은 priority를 가진 스레드가 낮은 priority를 가진 스레드가 자원을 차지하고 있기 때문에 block되는 현상인 priority inversion을 해결하기 위해, 낮은 priority의 스레드가 해당 자원을 최대한 빨리 사용함으로써 높은 priority의 스레드가 빨리 사용할 수 있도록 priority donation을 구현하였다.

2. 구현 내용

2.1. kthread

kernel/kthread.c 모듈에 있는 함수로는 kthread_create, kthread_exit, kthread_yield, kthread_set_prio, kthread_get_prio가 있다. kthread_create는 새로운 스레드를 만드는 함수이다. 이 때 스레드는 기존의 프로세스 구현과 같이 struct proc 구조체의 형태로 구현하였다. kthread_create를 구현할 때 kernel/proc.c의 fork 함수를 많이 참고하였다. proc[] array에서 비어있는 proc 구조체를 가져오는 allocthread 함수 역시 allocproc 함수와 비슷하게 구현하였다. 이 때 allocthread() 함수에서 특기할 점은 trapframe과 pagetable을 위한 page를 할당하지 않았다는 것이다. 이는 trapframe은 유저 프로세스의 context switch를 위한 공간이고, pagetable은 커널의 pagetable을 이용하면 되기 때문이다. kthread_create는 이렇게 받아온 proc 구조체에 fork 함수와 비슷한 방법으로 나머지 정보를 채워넣는다. fork와 다른 점은 context의 ra, 즉 return address 레지스터에 kernel/proc.c에 정의되어있는 threadret 함수의 주소를 저장하였다는 것이고, s10 레지스터와 s11 레지스터에 각각 fn 함수의 주소, fn 함수에 사용할 arg 인자를 저장하였다는 것이다. 이후 새로 만드는 스레드의 priority가 더 높으면 kthread_yield를 통해 잠시 CPU를 내려놓은 후 return한다. threadret 함수에서는 s10 레지스터와 s11 레지스터로 받아온 정보를 토대로 fn(arg)와 같이 최종적으로 실행을 요청받은 함수를 실행한다.

```
int
kthread_create(const char *name, int prio, void (*fn)(void *), void *c
{
  int i, tid;
  struct proc *nth;
  struct proc *th = myproc();

  if((nth = allocthread()) == 0){
```

```
    if((nth = allocthread()) == 0){
      return -1;
    }

    tid = nth->tid;

    nth->sz = th->sz;

    nth->parent = th->parent;

    for(i = 0; i < NOFILE; i++){
      if(th->ofile[i])
        nth->ofile[i] = th->ofile[i];
    }

    nth->cwd = th->cwd;

    safestrcpy(nth->name, name, sizeof(name));

    nth->context.ra = (uint64) threadret;
    nth->context.s10 = (uint64) fn;
    nth->context.s11 = (uint64) arg;

    nth->prio = prio;
    nth->base_prio = prio;

    nth->state = RUNNABLE;

    if(holding(&nth->lock))
      release(&nth->lock);

    if(th->prio > nth->prio)
      kthread_yield();

    return tid;
  }
```

kthread_exit 함수는 kernel/proc.c의 exit 함수와는 달리 스레드의 state를 ZOMBIE가 아닌 UNUSED로 전환한다. 이는 사망한 스레드를 받아줄 kthread_join 함수가 없기 때문이다.

ktherad_yield 함수는 kernel/proc.c의 yield 함수와 동일하게 구현하였다.

kthread_set_prio 함수는 현재 스레드가 donation을 받은 상태이고, 새로 들어오는 priority가 현재의 effective priority보다 더 높은 priority라면 기존의 donation을 취소하고 새로운 priority로 설정한다. 만약 새로 들어오는 priority가 더 낮다면 단순히 base priority로만 설정한다. 만약 donation이 일어나지 않은 상태라면 base priority와 effective priority 모두 새로운 priority로 설정한 후 kthread_yield()로 CPU를 반환한다.

kthread_get_prio 함수는 현재의 effective priority를 반환한다.

2.2. Preemptive Priority Scheduler

본 프로젝트의 스케쥴러는 Preemptive priority scheduler로서, 이름이 제시하는 대로 각 스레드의 priority를 기준으로 스케쥴링이 이루어져야 한다. 이 때 priority는 effective priority를 사용하며, 가장 높은

priority를 가진 스레드가 두 개 이상일 경우 그러한 각 스레드끼리는 Round-Robin 방식으로 돌아야 한다. 이는 다음과 같이 구현하여 최소한의 수정만을 가하였다.

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  int min_prio;

  c->proc = 0;
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    for(p = proc; p < &proc[NPROC]; p++) {
      min_prio = get_min_prio();

      acquire(&p->lock);

      if(p->state == RUNNABLE && p->prio == min_prio){
        p->state = RUNNING;
        c->proc = p;

        swtch(&c->scheduler, &p->context);

        c->proc = 0;
      }

      release(&p->lock);
    }
  }
}
```

한 timeer tick이 지나 스케쥴링이 이루어지면, 가장 높은 우선순위는 몇인지 알아낸 후 해당 우선순위를 가진 RUNNABLE인 스레드를 실행시킨다. 이 때 기존의 xv6 스케쥴러가 Round-Robin으로 구현되어있음에 착안하여 가장 높은 우선순위를 가진 스레드들이 여러 개 있다면 이들끼리는 마치 기존의 xv6 스케쥴러 위에서 구동되는 것과 같이 구현하였다.

2.3. Priority Donation

Priority donation이 올바르게 작동하게 하기 위하여, struct donation 구조체를 만들어 하나의 donation 정보를 담도록 하였다. 또한, 이 donation 정보들을 기록하기 위해 ledger라는 struct donation의 array를 만들고, 이를 concurrent하게 관리하기 위해 ledgerlock이라는 spinlock을 kernel/sleeplock.c에 만들었다. struct donation은 다음과 같이 선언하였다.

```
struct donation {
  struct sleeplock *lk;
  int donor_tid;
  int donee_tid;
  int prio;
  int old_prio;
};
```

lk은 해당 donation이 발생하게 된 sleeplock의 주소이고, donor_tid는 donation을 한 스레드의 tid이며, donee_tid는 donation을 받은 스레드의 tid이다. 또한 prio는 donation을 받은 결과 갖게 된 priority를 의미하며, old_prio는 donation 전에 가지고 있었던 priority를 뜻한다. 이 때 old_prio는 반드시 base priority가 아니라는 점에 유의해야 할 필요가 있었다.

acquiresleep 함수는 아래와 같이 수정하였다. 해당 sleeplock을 가지고 있는 스레드와 현재 스레드가 다른 경우, 즉 donation이 발생해야 하는 경우 prio_donate 함수를 호출하여 donation이 일어나도록 한 후, sleep 함수를 통해 lk에서 block될 수 있도록 하였다.

```
void
acquiresleep(struct sleeplock *lk)
{
  struct proc *th;
  struct proc *tgt_th;

  acquire(&lk->lk);
  while (lk->locked) {
    th = myproc();
=    if(lk->tid != th->tid){
      tgt_th = findthread(lk->tid);

      prio_donate(lk, th, tgt_th);
    }

    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  lk->tid = myproc()->tid;
  release(&lk->lk);
}
```

releasesleep 함수도 비슷하게 아래와 같이 구현하였다. 현재 스레드의 prio와 base_prio가 같지 않은 경우, 즉 donation을 받은 경우 releasesleep 전에 donation을 되돌려주어야 하기 때문에 prio_cancel 함수를 호출하고 release를 한 후, 가장 마지막에 kthread_yield()를 통해 되돌려준 donation의 결과로 새로이 스케쥴링을 수행하도록 하였다. 이 후에는 Preemptive priority scheduler의 특성 덕에 가장 높은 priority의 스레드가 실행될 것이다.

```
void
releasesleep(struct sleeplock *lk)
{
  struct proc *th;

  acquire(&lk->lk);

  th = myproc();
  acquire(&th->lock);

  if(th->prio != th->base_prio){
    prio_cancel(lk, th);
  }
```

```
  release(&th->lock);

  lk->locked = 0;
  lk->pid = 0;
  lk->tid = 0;
  wakeup(lk);
  release(&lk->lk);

  kthread_yield();
}
```

또한 prio_donate 함수는 아래와 같이 구현하였다. ledger에 기록이 이루어지니만큼 시작부터 끝까지 ledgerlock을 들고 있으며, ledger의 빈 칸에 정보를 기입한다. 이 때 기입하기 전에 donee가 donation을 이미 받은 상태이고, 새로 donation을 받는 priority가 이보다 낮다면 아무것도 하지 않고 return하며, 그렇지 않다면 이전의 donation을 cancel하고 새로이 donation을 받도록 하였다.

```
void
prio_donate(struct sleeplock *lk, struct proc *donor_th, struct proc *
{
  int i, j;
  struct donation *entry;

  acquire(&ledgerlock);
  for(i = 0; i < NDONATION; i++){
    // find an empty entry
    if(ledger[i].lk == 0){

      if(donee_th->prio != donee_th->base_prio){
        if(donee_th->prio <= donor_th->prio){
          release(&ledgerlock);
          return;
        }
        else{
          entry = find_donation(lk, donee_th);

          if(entry != 0)
            prio_cancel(lk, donee_th);
        }
      }


      ledger[i].lk = lk;
      ledger[i].donor_tid = donor_th->tid;
      ledger[i].donee_tid = donee_th->tid;
      ledger[i].old_prio = donee_th->prio;
      ledger[i].prio = donor_th->prio;

      donee_th->prio = ledger[i].prio;

      release(&ledgerlock);

      if(donee_th->chan != 0){
        for(j = i; j >= 0; j--){
          if(ledger[j].donor_tid == donee_th->tid){
            prio_donate(ledger[j].lk, donee_th, findthread(ledger[j].d
          }
```

```
        }
      }

      return;
    }
  }
  release(&ledgerlock);

  panic("ledger full");
}
```

prio_cancel 함수는 아래와 같이 구현하였다. lk과 donee의 정보를 가지고 ledger를 검색하여, donee의 priority보다 ledger에 기록된 priority가 낮으면 해당 donation이 일어난 이후 다른 lock을 통해 더 낮은 priority를 빌려왔다고 판단하여 해당 donation을 취소한다고 하더라도 donee의 priority에는 변화가 없어야 한다. 다만 모든 장부를 뒤져 해당 donation이 취소되었으니 old_prio값을 수정해주어야 한다. 또한 만약 장부상의 priority와 donee의 priority가 같다면 donee의 priority를 장부상에 기록된 old_prio값으로 바로 덮어씌워줌으로써 donation을 cancel한다. 이 작업이 끝나면 해당 장부의 lk 값을 0으로 만드는데, 이것은 lk 주소값이 장부를 검색하는 key 값으로 사용되기 때문에 취소되었고 추후 해당 자리에 다른 donation을 기록할 수 있음을 의미한다. 이러한 작업들을 통하여 multiple donation과 nested donation이 일어나는 상황에 대처할 수 있도록 하였다.

```
void
prio_cancel(struct sleeplock *lk, struct proc *donee_th)
{
  int i, j;

  for(i = NDONATION-1; i >= 0; i--){
    if (ledger[i].lk == lk && \
        ledger[i].donee_tid == donee_th->tid){

      if(donee_th->prio < ledger[i].prio){
        for(j = 0; j < NDONATION; j++){
          if(ledger[j].donee_tid == donee_th->tid && ledger[j].old_pri
            ledger[j].old_prio = ledger[i].old_prio;
          }
        }
      }
      else if(donee_th->prio == ledger[i].prio){
        donee_th->prio = ledger[i].old_prio;
      }

      ledger[i].lk = 0;

      return;
    }
  }

  return;
}
```

3. 참고

남은 프로젝트 시간이 많지 않아 Concurrency control 부분은 구현하지 못하였다. 또한 커널이 무거워져 실행시간이 늦어진 탓인지 sh의 터미널 문자열이 뜨는 시간보다 입력이 들어가는 시간이 더 빨라 테스트에서 FAIL이 뜨는 경우가 생겼는데, 교수님께서 말씀하신 것 처럼 run-test.py의 33번째의 time.sleep(1)을 time.sleep(3) 정도로 바꾸니 모두 PASS가 뜨는 것을 볼 수 있었다.

똥같은 코드도 코드라고 내놔서 죄송합니다... 한 학기 동안 수고 많으셨고 감사드립니다!