Operating System Project 3 Wiki

201903309 윤무원

Design

Thread 실체 정의

프로젝트를 시작하면서 제일 먼저 생각을 한 것은 쓰레드를 실체화 하는 방법이다. 개념적으로는 한 프로세스 내에 쓰레드가 여러개 존재하여 텍스트, 데이터 섹션을 공유하고 프로세스 내의 스택, 힙 영역을 나눠 갖는 형태이다. 하지만 쓰레드를 LWP(Light Weight Process), 즉 하나의 프로세스로써 xv6의 기본 기능을 제대로 수행할 수 있게 기존 xv6의 process를 그 자체로 쓰레드로 취급하여 하나의 독립적인 기본 실행 단위로 사용하였다.

이를 위해서 쓰레드끼리 공유되어야 하는 자원은 공유를 하고 독립적으로 가져야할 자원은 독립적으로 관리를 할 수 있도록 기능을 구현하였다. 또한 쓰레드가 기본 실행 단위이므로 스케쥴링 또한 프로세스를 선택해서 그 프로세스에 속한 쓰레드를 스케쥴링 하는 것이 아닌 각각의 쓰레드가 독자적으로 스케쥴링 되게하였다. 물론 의미적으로는 같은 프로세스에 속한 쓰레드들은 한 덩어리로 묶여서 관리되어야 했기에 후술할 ttable이라는 구조체를 만들어서 관리하였다.

proc 구조체 수정

LWP를 구현하기위해 proc 구조체에 수정이 약간 필요했다.

```
// proc.h
struct proc {
  //shared data
  uint sz;                     // Size of process memory (byt
  pde_t* pgdir;                // Page table
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
```

```
  //thread data
  int killed;                  // If non-zero, have been kill
  char *kstack;                // Bottom of kernel stack for
  enum procstate state;        // Process state
  struct trapframe *tf;        // Trap frame for current sysc
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on ch
  thread_t tid;                // Thread id
  void *retval;                // return value
  struct proc *joining_thread;    // A thread waiting for thi
};
```

수정사항이 많지는 않고 그저 원래의 프로세스 정보 중에서 쓰레드들이 공유되어야 할 것, 독자적으로 가져야할 것으로 나누어주었다. 몇 가지 추가된 변수가 있는데,

`thread_t tid` : 쓰레드의 id. pid와는 별개로 프로세스나 쓰레드가 하나 만들어질때마다 각각의 tid가 부여된다.

`void *retval` : 과제의 명세에 맞게 쓰레드가 종료될때 반환할 값을 저장할 변수이다.

`struct proc *joining_thread` : 내가 죽기를 기다리는 쓰레드, 즉 나를 조인하고 있는 쓰레드의 주소를 담고 있는 변수이다.

프로세스와 쓰레드의 생성 및 종료

그 다음으로 생각한 것이 프로세스와 쓰레드의 생성 및 종료의 동작을 구분하는 것이다. 설계상 프로세스 자체를 쓰레드로 취급하였지만 생성과 종료의 동작은 프로세스와 쓰레드가 서로 다르게 수행되어야 한다.

1. 프로세스 생성 : 새로운 페이지테이블을 만들어서 할당해주고 새로운 pid를 부여해서 다른 프로세스와 구분이 될 수 있게하였다. 또한 새롭게 만들어진 프로세스도 그 자체로 쓰레드이기 때문에 새로운 tid도 부여를 하고 ttable에 해당 프로세스의 공간을 할당해주었다.
2. 쓰레드 생성 : 프로세스==쓰레드로 정의했기에 새로운 프로세스가 생성이 되기는 하나 이는 쓰레드로 취급해야하므로 쓰레드를 생성한 프로세스의 페이지테이블과 pid를 공유하고 새로운 tid를 부여하였다.
3. 프로세스 종료 : 프로세스가 종료될때는 현재 프로세스내에 있는 모든 쓰레드가 종료되어야함으로 종료되는 프로세스의 pid와 동일한 pid를 갖는 쓰레드를 모두 찾아서 종료를 해주었다.

4. 쓰레드 종료 : 쓰레드의 상태를 좀비로 바꾸고 `struct proc *joining_thread` 변수를 활용하여 종료된 쓰레드의 자원을 회수하였다.

쓰레드 관리를 위한 ttable 추가

내가 설계한 방식으로는 모든 쓰레드들이 독립적으로 작동을 하기에 쓰레드 관리의 편의를 위해 ttable이라는 구조체 배열을 만들어서 각 프로세스별로 가지고 있는 쓰레드의 정보를 저장하였다.

```
// proc.c
struct {
  int pid;
  struct proc *thread_list[10];
  int num_thread;
} ttable[NPROC];
```

`int pid` : 쓰레드가 속해있는 프로세스의 pid.

`struct proc *thread_list[10]` : 프로세스 내에 존재하는 쓰레드를 저장하는 배열. 쓰레드 갯수의 조건이 따로 명시되어있지 않아서 임의로 최대갯수를 10개로 설정하였다.

`int num_tread` : 프로세스 내에 존재하는 쓰레드의 갯수.

Lock method 설계

동기화를 보장하는 방법으로는 소프트웨어로 구현하는 방법과 하드웨어의 지원을 받아 atomic operation이 가능한 함수를 사용하는 방법이 있다. 내 생각으로는 소프트웨어로 구현하는 방법이 과제 출제의 의도와 더 맞다고 판단하여 소프트웨어로 구현하기로 결정했다.

동기화의 대표적인 소프트웨어 솔루션에는 피터슨 알고리즘이 있다.

```
// Peterson's algorithm

while(1) {                          // 프로세스i의 진입 영역

  flag[i] = ture;                   // 프로세스i가 임계구역에 진입
하기 위해 진입을 알림.
  turn = j;                         // 프로세스j에게 순서를 양보
함.

  while (flag[j] && turn = j);      // 프로세스i의 차례가 될 때까
```

```
지 대기를 함.

    // critical section

  flag[i] = false                    // 임계구역 사용완료를 알림.

}
```

하지만 피터슨 알고리즘은 프로세스의 갯수가 2개일때만 동기화가 보장이 되고 프로세스가 3개 이상이 되면 동기화를 보장하지않는다. 그렇기에 기존 피터슨 알고리즘을 활용하여 3개 이상의 쓰레드가 있는 환경에서 동기화가 보장될 수 있도록 함수를 설계하였다.

```
while(1) {

  flag[i] = ture;
  priority[i] = max(priority[0] ~ priority[n-1]);
  for(int j = 0; j < n; j++){
      while(flag[j] == true && (priority[j] < priority[i]


    // critical section

  flag[i] = false

}
```

`turn` 변수 대신 `priority`라는 정수형 배열을 사용해 critical section에 진입하려는 각각의 thread에게 우선순위를 부여하였다. 진입하려는 순서가 늦을 수록 낮은 우선순위가 부여된다. 먼저 각각의 쓰레드는 critical section에 진입할때 자신의 `flag`를 `true`로 바꾸고 현재 시점에서 가장 후순위의 우선순위를 부여한다. 이후 모든 쓰레드의 `flag`를 순회하면서 해당 쓰레드가 진입할 의지가 있고 우선순위가 현재 쓰레드보다 높으면 기다린다. 여기서 우선순위는 여러 쓰레드가 동시에 부여받아 똑같은 우선순위를 가지는 경우가 발생할 수 있다. 그렇기에 우선 순위가 같으면 tid가 낮은 순서대로 먼저 진입하도록 하였다.

Implement

앞서 설명한 디자인대로 xv6가 작동할 수 있게 여러 새로운 함수를 추가하고 기존의 함수들을 수정하였다. 하나씩 코드를 보면서 알아보자.

allocproc()

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

  acquire(&ptable.lock);

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

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->tid = nexttid++;

  release(&ptable.lock);

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;
```

```
  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;

  return p;
}
```

fork와 thread_create 모두 allocproc을 호출해 proc 구조체를 생성하고 ptable에 할당한다. 그렇기에 fork가 실행될때에도 문제없이 allocproc을 통해 새로운 프로세스가 생성되어야하므로 수정을 최소화하였다. allocproc 내에선 pid와 tid만 부여하고 이후 `thread_create()` 내에서 proc 구조체가 쓰레드로써 의미를 가지도록 구성하였다.

thread_create()

```
int thread_create(thread_t *thread, void *(*start_routine)(vo
  struct proc *curproc;
  struct proc *new_thread;
  uint sz, sp, ustack[2];

  curproc = myproc();
  if((new_thread = allocproc()) == 0) return -1;

  acquire(&ptable.lock);

  // share process data
  new_thread->pgdir = curproc->pgdir;
  new_thread->pid = curproc->pid;
  nextpid--; // to maintain pid
  new_thread->parent = curproc->parent;
  safestrcpy(new_thread->name, curproc->name, sizeof(curproc-
  for(int i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
```

```
    new_thread->ofile[i] = filedup(curproc->ofile[i]);
  new_thread->cwd = idup(curproc->cwd);
  *new_thread->tf = *curproc->tf;
  *thread = new_thread->tid;

  //allocate thread data
  sz = curproc->sz;
  sz = PGROUNDUP(sz);
  if((sz = allocuvm(new_thread->pgdir, sz, sz + 2*PGSIZE)) ==
    goto bad;
  clearpteu(new_thread->pgdir, (char*)(sz - 2*PGSIZE));

  curproc->sz = sz;
  new_thread->sz = sz;

  //modify ttable
  int index = find_process(new_thread->pid);
  for(int i = 0; i < 10; i++){
    if(ttable[index].thread_list[i] == 0) {
      ttable[index].thread_list[i] = new_thread;
      break;
    }
  }
  ttable[index].num_thread++;
  for(int i = 0; i < ttable[index].num_thread; i++){
    ttable[index].thread_list[i]->sz = sz;
  }
  sp = sz;

  ustack[0] = 0xffffffff;  // fake return PC
  ustack[1] = (uint)arg;
  sp -= sizeof(ustack);

  if(copyout(curproc->pgdir, sp, ustack, sizeof(ustack)) < 0)
    goto bad;

  new_thread->tf->eax = 0;
  new_thread->tf->eip = (uint)start_routine;
```

```
  new_thread->tf->esp = sp;
  switchuvm(curproc);

  new_thread->state = RUNNABLE;

  release(&ptable.lock);

  return 0;

  bad:
  if(curproc->pgdir)
    freevm(curproc->pgdir);
  release(&ptable.lock);
  return -1;
}
```

기존의 `fork()`와 `exec()` 코드를 활용하여 구현하였다. `allocproc`을 통해 새로운 proc 구조체를 생성하고 `pgdir`에 현재 프로세스(thread_create를 호출한 프로세스)의 pgdir을 똑같이 저장하여 페이지 테이블을 공유하도록 하였다. 그외 pid와 parent, open file 등등 쓰레드끼리 공유하는 프로세스의 정보를 저장하였다. 이후 `allocuvm()`을 사용해 쓰레드만의 스택 공간을 할당하고 `ttable`의 `thread_list`에 추가해주었다.

thread_exit

```
void thread_exit(void *retval){
  struct proc *curthread;
  int fd;
  int index;

  curthread = myproc();

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curthread->ofile[fd]){
      fileclose(curthread->ofile[fd]);
      curthread->ofile[fd] = 0;
    }
  }
```

```
  begin_op();
  iput(curthread->cwd);
  end_op();
  curthread->cwd = 0;

  acquire(&ptable.lock);
  index = find_process(curthread->pid);

  for(int i = 0; i < 10; i++){
    if(ttable[index].thread_list[i]->tid == curthread->tid){
      ttable[index].thread_list[i] = 0;
    }
  }
  ttable[index].num_thread--;

  // if there is joining thread, wakeup it
  if(curthread->joining_thread != 0) wakeup1(curthread->joini

  // store retval
  curthread->retval = retval;

  // Jump into the scheduler, never to return.
  curthread->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

기존의 exit()의 코드를 활용해서 구현하였다. open file을 모두 닫고 `ttable`의 `thread_list`에서 해당 쓰레드를 제거하였다. 또한 `joining_thread`를 확인해 이 값이 0이 아니라면 자신을 조인하고 있는 쓰레드가 존재하고 있는 것이므로 해당 쓰레드를 wakeup하고 상태를 좀비로 바꿔주었다.

thread_join

```
int thread_join(thread_t thread, void **retval){
  struct proc *join_thread;
  struct proc *curthread;
```

```
  int index;

  curthread = myproc();


  acquire(&ptable.lock);
  for(;;){
    for(join_thread = ptable.proc; join_thread < &ptable.proc
      if(join_thread->pid == 0 || join_thread->tid != thread)
        continue;
      join_thread->joining_thread = curthread;
      if(join_thread->state == ZOMBIE){
        // Found one.
        kfree(join_thread->kstack);
        join_thread->kstack = 0;
        // check num thread
        index = find_process(join_thread->pid);
        if(ttable[index].num_thread == 0){
          freevm(join_thread->pgdir);
        }
        join_thread->pid = 0;
        join_thread->parent = 0;
        join_thread->name[0] = 0;
        join_thread->killed = 0;
        join_thread->state = UNUSED;
        join_thread->tid = 0;
        *retval = join_thread->retval;
        join_thread->joining_thread = 0;
        release(&ptable.lock);
        return 0;
      }
    }

    sleep(curthread, &ptable.lock);  //DOC: wait-sleep
  }
  release(&ptable.lock);
  return -1;
}
```

기존의 `wait()`의 코드를 활용하여 구현하였다. ptable을 순회하여 인자로 받은 id에 해당하는 쓰레드를 찾은 뒤 해당 쓰레드의 `joining_tread` 를 현재 쓰레드로 저장한후 상태를 확인, 좀비면 자원을 회수하고 좀비가 아니면 sleep을 하였다.

find_process()

```
int find_process(int pid){
  int i;

  for(i = 0; i < NPROC; i++){
    if(ttable[i].pid == pid){
        return i;
    }
  }
  return -1;
 }
```

`ttable`에서 인자로 받은 `pid`와 동일한 프로세스가 저장되어있는 인덱스를 반환하는 함수이다.

clear_thread()

```
void clear_thread(struct proc *thread){
  int index;
  acquire(&ptable.lock);

  index = find_process(thread->pid);

  for(int i = 0; i < 10; i++){
    if((ttable[index].thread_list[i] != 0) && (ttable[index].
      if(ttable[index].thread_list[i]->tid == 0) continue;
      kfree(ttable[index].thread_list[i]->kstack);
      ttable[index].thread_list[i]->kstack = 0;
      // check num thread
      ttable[index].thread_list[i]->pid = 0;
      ttable[index].thread_list[i]->parent = 0;
      ttable[index].thread_list[i]->name[0] = 0;
      ttable[index].thread_list[i]->killed = 0;
```

```
      ttable[index].thread_list[i]->state = UNUSED;
      ttable[index].thread_list[i]->tid = 0;
      if(ttable[index].thread_list[i]->joining_thread != 0) w
      ttable[index].thread_list[i]->joining_thread = 0;
    }
  }
  ttable[index].num_thread = 1;

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

현재 쓰레드를 제외하고 해당 프로세스의 모든 쓰레드를 정리하는 함수이다. `ttable` 의 `thread_list` 를 순회하면서 현재 쓰레드를 제외한 모든 쓰레드의 자원을 회수하였다. 그 과정에서 종료된 쓰레드를 조인하고 있는 쓰레드가 있으면 `wakeup1` 로 깨워주었다.

userinit()

```
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size

  p = allocproc();

  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_init
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S
```

```
  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  for(int i = 0; i < NPROC; i++){
    ttable[i].pid = 0;
    ttable[i].num_thread = 0;
    for(int j = 0; j < 10; j++){
      ttable[i].thread_list[j] = 0;
    }
  }
```

ttable에 대한 초기화 작업을 userinit 함수 내에서 처리해주었다.

growproc()

```
int
growproc(int n)
{
  uint sz;
  int index;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  index = find_process(curproc->pid);
  sz = curproc->sz;
  for(int i = 0; i < 10; i++){
    if(ttable[index].thread_list[i] != 0 && ttable[index].thr
      sz = ttable[index].thread_list[i]->sz;
    }
  }
  if(n > 0){
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0){
      release(&ptable.lock);
      return -1;
    }
  } else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0){
```

```
      release(&ptable.lock);
      return -1;
    }
  }
  curproc->sz = sz;
  for(int i = 0; i < ttable[index].num_thread; i++){
    ttable[index].thread_list[i]->sz = sz;
  }

  switchuvm(curproc);
  release(&ptable.lock);
  return 0;
}
```

과제 명세에 맞게 `sbrk()` 가 수행될 수 있도록 수정을 하였다. 여러 쓰레드가 동시에 할당을 요청해도 할당해주는 공간이 겹치지 않도록 `ptable.lock` 으로 동기화를 해주었다. 또한 할당 받은 공간을 프로세스 내의 모든 쓰레드가 공유할 수 있도록 하기 위해 추가로 메모리를 할당 받은 후 모든 쓰레드의 `sz` 를 동일하게 수정해주었다.

fork()

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

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

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0)
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
```

```
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  for(int i = 0; i < NPROC; i++){
    if(ttable[i].pid != 0) continue;
    ttable[i].pid = np->pid;
    ttable[i].thread_list[0] = np;
    ttable[i].num_thread = 1;
    break;
  }

  release(&ptable.lock);

  return pid;
}
```

`proc`구조체 자체를 thread로 취급하였기 때문에 큰 수정사항 없이도 쓰레드에서 fork를 호출해도 기존의 루틴이 문제없이 수행된다. 프로세스를 생성한후 새롭게 만들어진 프로세스의 ttable을 할당해주었다.

exit()

```
void
exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  if(curproc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  clear_thread(curproc);

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
```

```
    }
  }

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

exit를 통해 프로세스가 종료되려면 해당 프로세스 내의 모든 쓰레드가 종료되어야하므로 `clear_thread()` 를 통해 프로세스 내의 모든 쓰레드를 정리한뒤 현재 쓰레드의 상태를 좀비로 바꿔주었다.

wait()

```
int
wait(void)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
```

```
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        p->joining_thread = 0;
        p->tid = 0;
        // remove from thread_list
        int index = find_process(pid);
        for(int i = 0; i < 10; i++){
          ttable[index].thread_list[i] = 0;
        }
        ttable[index].num_thread = 0;
        ttable[index].pid = 0;
        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

`fork()` 와 마찬가지로 proc 구조체 그 자체를 쓰레드로 취급하였기에 큰 수정사항 없이도 쓰레드가 `wait()` 를 호출해도 문제없이 기존 루틴을 수행한다. 자식 프로세스를 정리한후에는 해당 프로세스의 `ttable` 도 같이 정리해주었다.

kill()

```
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;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING)
        p->state = RUNNABLE;
      release(&ptable.lock);
      clear_thread(p);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

하나 이상의 쓰레드가 kill 되면 그 쓰레드가 속한 프로세스 내의 모든 쓰레드도 정리가 되어야 하므로 `clear_thread()` 를 호출해 쓰레드를 정리해주었다.

exec()

```
// exec.c
int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  uint argc, sz, sp, ustack[3+MAXARG+1];
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pde_t *pgdir, *oldpgdir;
  struct proc *curproc = myproc();

  begin_op();

  ....
```

```
  clear_thread(curproc);
  switchuvm(curproc);
  freevm(oldpgdir);
  return 0;

    bad:
  if(pgdir)
    freevm(pgdir);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
}
```

현재 쓰레드를 제외한 기존 프로세스의 모든 쓰레드를 정리하기 위해 `clear_thread()` 를 사용하였다.

lock()

```
void lock(int tid)
{
    int other_thread;
    int max;
    flag[tid] = 1;
    max = priority[0];
    for(int i = 1; i < NUM_THREADS; i++){
        if(max < priority[i])
            max = priority[i];
    }
    priority[tid] = max + 1;

    for (other_thread = 0; other_thread < NUM_THREADS; other_
        while( flag[other_thread] == 1 && (priority[other_thr
                || (priority[other_thread] == priority[tid] &&
    }
```

```
}
```

인자로 쓰레드의 id, `int tid` 를 받는다. 해당 쓰레드의 `flag` 를 1로 바꿔주고 현재 critical section에 진입하려는 thread 중에서 낮은 우선순위를 부여받는다(`priority` 값이 높을 수록 낮은 우선순위). 이후 모든 쓰레드를 순회하면서 `flag` 가 1이면 critical section에 진입할 의지가 있다고 판단, 우선순위를 확인한다. 이 과정을 모든 쓰레드에 대해 진행해서 `flag` 가 1이고 자신보다 우선순위가 높은 쓰레드가 존재하지 않을때까지 기다렸다가 critical section에 진입하도록 구현하였다. 앞서 디자인 파트에서 설명했듯이 우선순위를 부여받는 코드 부분이 여러 쓰레드에서 동시에 실행돼서 여러 쓰레드가 같은 우선순위를 부여받을 수 있으므로 그때는 tid가 작은 순서로 먼저 진입하게 통제하였다.

unlock()

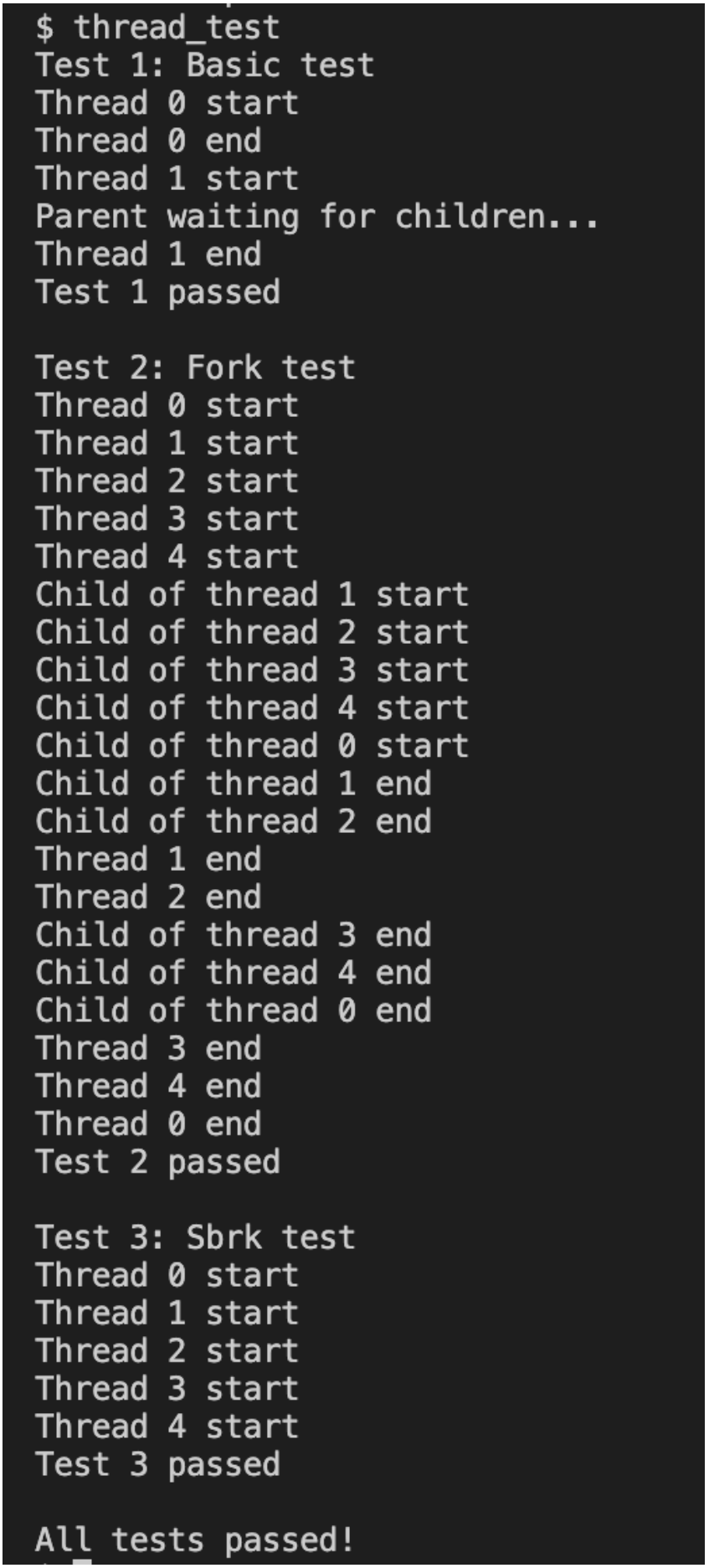
```
void unlock(int tid)
{
    flag[tid] = 0;
}
```

critical section이 완료되면 해당 쓰레드의 `flag` 를 0으로 바꿔서 critical section에 진입할 의지가 없다는 걸 다른 쓰레드에 알려준다.

Result

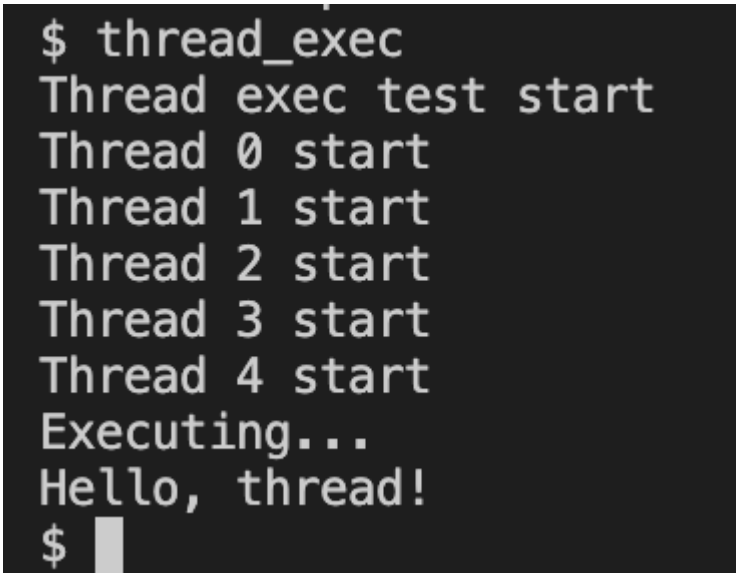
제공되는 테스트 코드를 유저프로그램으로 등록하여 실행을 해보았다.

thread_test.c


$ thread_test
Test 1: Basic test
Thread 0 start
Thread 0 end
Thread 1 start
Parent waiting for children...
Thread 1 end
Test 1 passed
Test 2: Fork test
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Child of thread 1 start
Child of thread 2 start
Child of thread 3 start
Child of thread 4 start
Child of thread 0 start
Child of thread 1 end
Child of thread 2 end
Thread 1 end
Thread 2 end
Child of thread 3 end
Child of thread 4 end
Child of thread 0 end
Thread 3 end
Thread 4 end
Thread 0 end
Test 2 passed
Test 3: Sbrk test
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Test 3 passed
All tests passed!

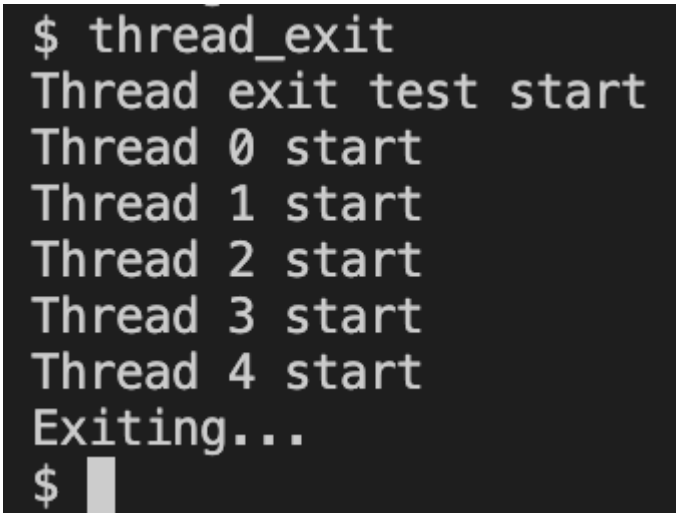
test 1, 2, 3 모두 예시와 동일하게 잘 실행되는 것을 확인할 수 있다.

thread_exec.c



역시 예시와 동일하게 잘 실행된다.

thread_exit.c



예시와 동일하게 잘 실행된다.

thread_kill.c

```
$ thread_kill
Thread kill test start
Killing process 18
This code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
Kill test finished
$
```

부모프로세스의 쓰레드는 자식 프로세스의 쓰레드가 호출한 kill에 영향을 받지않으므로 부모 프로세스의 쓰레드가 실행한 코드가 5번 잘 실행되는 걸 확인할 수 있다.

pthread_lock_linux.c

NUM_ITERS와 NUM_THREADS 둘 다 1000으로 설정하고 테스트를 진행하였다.

먼저 lock(), unlock()을 사용하지않고 10번 실행해보았다.

```
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 999274
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 999556
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 998484
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 999043
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 998584
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 999000
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 999028
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 997396
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 1000000
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 997983
root@62b2bd4eb118:/OS/xv6-public#
```

동기화가 보장되지않아 결과값이 다르게 출력되는 것을 확인할 수 있다.

내가 구현한 lock(), unlock()을 사용해서 10번 실행해보자.

```
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 1000000
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 1000000
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 1000000
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 1000000
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 1000000
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 1000000
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 1000000
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 1000000
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 1000000
root@62b2bd4eb118:/OS/xv6-public# ./pthread_lock_linux
shared: 1000000
```

실행해본 결과 10번 모두 결과값이 정확하게 출력되는 것을 확인할 수 있다. 출력값은 정확하게 나오지만 한 번 실행할때마다 약 1분 가량이 소요된다. 속도가 느리다는 소프트웨어 솔루션의 단점이 여실히 드러나는 걸 확인할 수 있다.

Trouble shooting

growproc() panic : remap

sbrk 시스템 콜을 과제 명세에 맞게 구현하기 위해 growproc()을 수정하는 과정에서 자꾸 panic : remap이 발생하였다. panic remap은 잘못된 메모리 접근이 일어나거나 페이지 테이블 항목이 충돌될때 발생된다는 걸 검색을 통해 알게되었고 이를 발생시킬 가능성이 있는 코드들을 살펴보았다. 그 결과 allocuvm()을 통해 해당 프로세스에 추가적인 메모리 공간을 할당되면 해당 프로세스의 모든 쓰레드의 sz 값을 바꿔주어야 하는데 그렇지 않아서 다른 프로세스가 이미 할당된 메모리 공간에 또 다시 할당을 하려는 동작이 수행되서 문제가

발생한 것이었다. 그래서 allocuvm()을 실행하고 추가 메모리 공간을 할당받아 새로운 sz값을 해당 프로세스에 속한 모든 쓰레드에 반영해줌으로써 해결할 수 있었다.

lock() 실행 시간 이슈

반복수와 쓰레드의 갯수를 10에서부터 점차 늘려가면서 테스트를 해보았는데 쓰레드 갯수가 1000 이하일때는 실행 시간이 길어지긴 했지만 1분 내외로 실행이 완료되었다. 하지만 쓰레드의 갯수가 10000 이상이 되는 시점부터는 실행 시간이 너무 길어져 제대로 동작을 하는지 확인을 할 수 없었다. 단순히 실행이 완료되는데 시간이 오래 걸리는건지, 구현한 코드에 헛점이 있어서 데드락이 발생하는건지 판단을 내리지 못하였다.