

Project03

2020055350 강현중

I . pthread

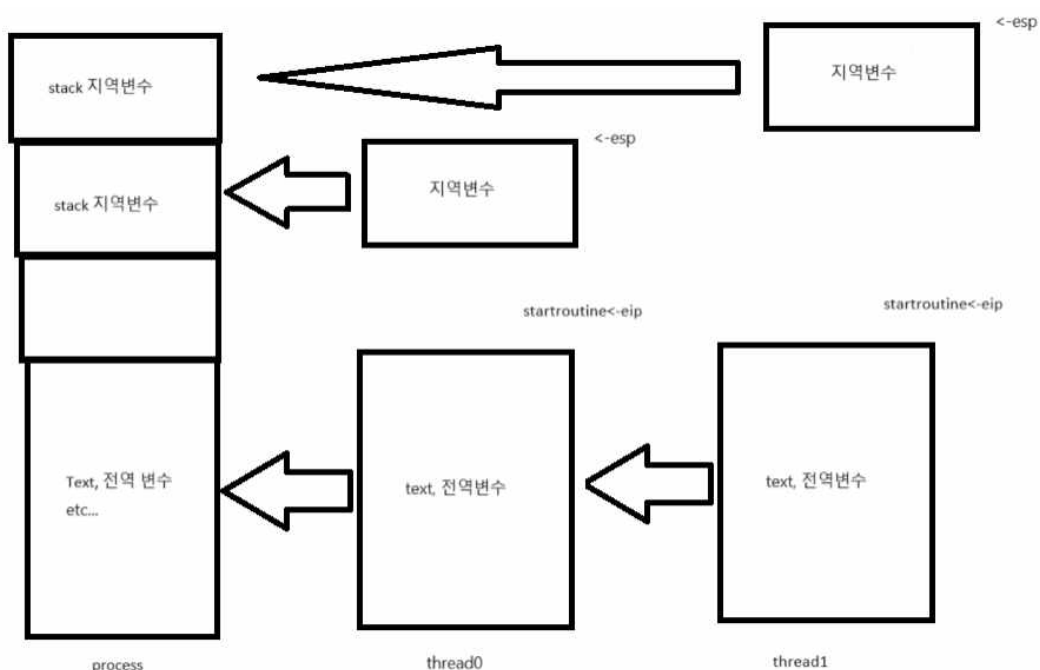
1. Design

가. 구현 목표

기존 xv6의 RR 스케줄러를 사용하기 위해 xv6의 기존 proc.h의 프로세스 구조체를 조금 수정하여 쓰레드를 프로세스처럼 취급하여 스케줄링 할 수 있도록 한다. 추가하는 요소를 최소로 하여 단순한 구조를 만드는 것을 목표로 한다. 각 쓰레드들은 기존의 프로세스와 pgdir을 공유해 주소공간을 공유하고, 각 쓰레드들은 각각의 지역변수 섹션(스택)을 가리키는 esp와 실행 부분인 eip값을 가져, 각각 다른 지점에서 자신만의 매개 변수로 실행흐름을 있도록 한다.

이때, 쓰레드들은 각각 쓰레드를 생성한 프로세스(t_leader)와 쓰레드마다 다른 id(tid)를 자료구조에 저장하게 되고, pid는 부모와 동일한 값을 가지게 한다. 이를 통해, 같은 프로세스군(한 프로세스에서 생성된 쓰레드)에 속하는 쓰레드들을 쉽게 찾아내어, kill이나 exec등 다른 모든 프로세스를 찾아내야 할 때, 프로세스를 사용해서 찾거나, 생성한 프로세스를 찾을 때, 빠르게 찾아낼 수 있도록 한다.

기존의 프로세스와 비슷하게 만든다면 시스템 콜 구현에서 장점이 있다. 기존의 시스템 콜과 유사한 형태로 pthread create, exit, join 시스템 콜을 만들 수 있고, 이는 각각 기존의 fork & exec, exit, wait와 대응된다. 또, 다른 시스템 콜과 상호작용 할 때도, 프로세스와 다를 바가 없어, 간단하게 만들 수 있을 것이다.



<1. 구현 할 자료구조>

나. 시스템 콜 구현을 위한 기존 시스템 콜 분석

1) fork

```
.....
if((np = allocproc()) == 0){
    return -1;
}
acquire(&ptable.lock);
// 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;
```

<2. fork system call>

핵심이 되는 부분은 allocproc()을 통해 메모리를 할당 받고, copyuvm을 통해 자신의 주소 공간에, 부모 프로세스의 주소 공간을 복사해서 넣는 부분이다. 이를 통해 fork 직후에는 부모와 동일한 프로세스가 만들어지게 된다. pthread_create도 이와 유사하지만 pgdir을 공유하게 만들 것이다.

2) exec

```
.....
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;

// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
        goto bad;
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
        goto bad;
    ustack[3+argc] = sp;
}
ustack[3+argc] = 0;

ustack[0] = 0xffffffff; // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4; // argv pointer

sp -= (3+argc+1) * 4;
if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;

// commit to the user image.
oldpgdir = curproc->pgdir;
curproc->pgdir = pgdir;
curproc->sz = sz;
curproc->tf->eip = elf.entry; // main
curproc->tf->esp = sp;
```

<3. exec system call>

먼저 PGROUNDUP을 통해 사이즈를 PGSIZE (=4096)의 배수로 반올림 주는데, 프로세스를 확장할 때, 페이지 단위로 확장하기 위해서이다. allocuvm으로 2페이지를 확장하는데, 한 페이지는 프로세스가 활용하는 부분, 한 페이지는 guard section으로 사용한다. 이후 argv(매개변수)와 fake return pc를 ustack이라는 배열에 넣고, 할당받은 공간에 copyout으로 넣어 준다. 그리고 기존의 pgdir을 버리고 새로운 pgdir로 변경하고, eip와 esp를 지정해 실행흐름을 변경된 프로세스로 바뀐다.

pthread_create의 함수 시작 부분을 이와 비슷하게 구현하되, 주소공간은 기존의 프로세스와 공유하여 기존의 pgdir을 유지하는 방향으로 설계할 것이다.

3) exit

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

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();
```

```
for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
        fileclose(curproc->ofile[fd]);
        curproc->ofile[fd] = 0;
    }
}
```

<4. exit system call>

먼저 열려 있는 파일에 대한 참조를 해제해준다. 아직 xv6의 파일 시스템에 대해 다루지 않아 간단히 정리하면 file close를 통해, 현재 열려있는 파일에 대한 참조 수를 하나 줄이고, 만약 참조하는 파일이 없으면 파일을 닫는 방식으로, 파일을 정리해 준다. 그 이후 파일 begin_op, iput, end_op를 통해 현재 디렉토리에 대한 공유를 포기한다.

다음으로는 실질적으로 종료하는 부분이다. 먼저 자신의 부모 프로세스가 자신이 종료하는 것을 wait에서 sleep상태로 기다릴 수 있기 때문에 먼저 깨워준다, 그리고 ptable을 돌면서 자신을 부모로 하는 프로세스가 있다면, 그 프로세스의 부모를 initproc으로 바꿔주고, 만약 그 프로세스가 종료되어 ZOMBIE 상태라면, initproc을 깨워 정리를 일임한다.

마지막으로 자기 자신을 ZOMBIE로 만들고 sched 함수를 호출하여, 방금 깨운 부모프로세스가, 정리할 수 있도록 한다.

구현할 exit_thread에서도 이와 비슷한 방법으로 thread_join에서 기다리고 있는 t_leader를 깨우고, 자신을 ZOMBIE로 만드는 방법으로 구현할 것이다.

4) wait

```
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;
            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 wakeup call in proc_exit.)
    sleep(curproc, &ptable.lock); //DOC: wait-sleep
}
```

<5. wait system call>

ptable을 전부 순회 하며, 자신을 부모로 하는 프로세스가 있는지 찾고, 만약 그 프로세스가 ZOMBIE라면, 그 프로세스의 kstack, pgdir의 할당을 해제하고 기타 변수들을 정리한다. 만약 내 프로세스가 kill 됐거나, 나를 부모로 하는 프로세스가 없다면, wait의 대상이 없으므로 -1을 반환한다. 만약 자식프로세스는 있는데, 아직 종료만 되지 않은 상태라면, sleep waiting 해준다. thread_join도 이와 유사하게 구현하되, pgdir은 공유하므로, 이에 대한 할당 해지는 쓰레드가 종료할 때는 하지 않을 것이다.

2. Implement

가. 쓰레드 디자인

```
void * retval;
int tid; // thread_id -1 == proc;
int nexttid; // next thread_id;
struct proc* tleader; // thread leader;
};
```

<6. proc에 추가한 요소>

먼저 proc.h의 proc에 구성요소를 추가했다. 단순히 만들기 위해, return value를 저장할 retval, 그리고 한 프로세스 군에서 pid를 같게 만들 것이기 때문에 내부에서 구분하기 위한 tid와 이를 할당하기 위한 nexttid, 마지막으로 프로세스의 parent 역할을 하는 쓰레드를 만든 프로세스 tleader로 구성해, 빠르게 쓰레드를 만든 프로세스를 찾아 갈 수 있게 구성했다. 또 프로세스의 tid는 -1로 지정해, 프로세스와 쓰레드 간의 구분이 가능하게 만들었으며, <pid,tid> 쌍을 통해 각각 구분할 수 있게 만들었다.

프로세스와 구조상 동일 하기 때문에, 기존의 시스템콜도 약간 수정하여 사용할 수 있고, 기본 스케줄러도 수정없이 이용할 수 있게 만들었다.

나. system call(pthread_create, exit, join) 구현

1) system call 등록

```
int sys_thread_create(void){
    thread_t* tid;
    void* start_routine(void*);
    void* arg;
    if(argptr[0], (char**) &tid, sizeof(tid)) <= argptr[1], (char**) &start_routine, sizeof(start_routine)) <= argptr[2], (char**) &arg, sizeof(arg)) <= 0
        return -1;
    return thread_create(tid, start_routine, arg);
}

int sys_thread_exit(void){
    void* retval;
    if(argptr[0], (char**) &retval, sizeof(retval)) <= 0
        return -1;
    thread_exit(retval); //void type function
    return 0;
}

int sys_thread_join(void){
    thread_t* tid;
    void* retval;
    if(argptr[0], (char**) &tid, sizeof(tid)) <= argptr[1], (char**) &retval, sizeof(retval)) <= 0
        return -1;
    return thread_join(tid, retval);
}
```

<7. sysproc.c>

```
[SYS_thread_create] sys_thread_create, extern int sys_thread_create(void);
[SYS_thread_exit] sys_thread_exit, extern int sys_thread_exit(void);
[SYS_thread_join] sys_thread_join, extern int sys_thread_join(void);
```

<8. syscall.c>

```
26 #define SYS_thread_exit 25
27 #define SYS_thread_join 26
28 #define SYS_t_create 27
```

<9. syscall.h>

```
int thread_create(thread_t *thread, void *(*start_routine)(void*), void * arg);
void thread_exit(void *retval);
int thread_join(thread_t thread, void**retval);
```

<10. user.h>

```
int thread_create(thread_t *thread, void *(*start_routine)(void*), void * arg);
void thread_exit(void *retval);
int thread_join(thread_t thread, void**retval);
```

<11. defs.h>

```
34 SYSCALL(thread_create)
35 SYSCALL(thread_exit)
36 SYSCALL(thread_join)
```

<12. usys.S>

먼저 시스템 콜 3개를 등록해 준다. 기존에 하던대로 sysproc.c에 wrapper function을 만들고 syscall.c, syscall.h, user.h, defs.h에 시스템콜을 등록해준다. 그리고 usys.S에서 매크로로 등록해 시스템 콜로 동작하게 한다. 시스템콜의 내용은 proc.h에 작성했기 때문에 Makefile에서의 수정 내용은 없다.

2)thread_create

```
if((th = allocproc()) == 0) return -1;
th->pgdir = proc->pgdir; // share page table --> share address space

th -> tleader = proc;
th -> parent = proc -> parent;
th -> tid = ++ proc -> nexttid; // -1 == process, start from 0
*thread = th->tid; // trouble shooting
th -> pid = proc->pid;
*(th->tf) = *(proc -> tf); // copy trapframe
```

<13. thread_create 프로세스 할당 부분 1>

먼저 fork와 유사하게 allocproc을 통해 프로세스를 할당 받고, pgdir을 공유해 프로세스와 주소공간을 공유한다. 다음으로 쓰레드의 tleader를 현재 프로세스로 설정하고, 쓰레드의 부모를 현재 프로세스의 부모로 지정해 준다. 다음으로 ++nexttid를 tid로 설정하고, pid는 프로세스와 동일하게 설정해, <pid,tid>로 각 쓰레드가 어디에 속하는지, 몇 번째 쓰레드인지 구분할 수 있게 한다. 마지막으로 trapframe을 복사한다.

```

acquire(&ptable.lock);
proc->sz = PGROUNDUP(proc->sz);
sp = proc->sz;
if((sp = allocvm(proc->pgdir, sp, sp+2*PGSIZE))==0) /
{
    //cprintf("1");
    th->state = UNUSED;
    return -1;
}
//cprintf("%d\n", sp);
clearpteu(proc->pgdir, (char*)(sp-2*PGSIZE));
proc->sz += 2*PGSIZE;
th->sz = proc->sz;
ustack[0]=0xFFFFFFFF;
ustack[1]=(uint)arg;
sp -= 8;
//cprintf("%d\n", sp);
if(copyout(proc->pgdir, sp, ustack, 8)<0)
{
    //cprintf("2");
    th->state = UNUSED;
    deallocvm(proc->pgdir, sp+8, sp-2*PGSIZE+8);
    return -1;
}
th->tf->eax=0;
th->tf->eip=(uint)start_routine;
th->tf->esp=sp;

```

<14. start_routine과 매개변수 지정>

위에서 살펴본 exec의 함수 실행부분과 유사하게 먼저 allocvm을 통해 2페이지를 새로 할당 받는다. 그 후, 프로세스의 sz를 늘려주고, ustack을 통해 fake return address와 매개변수를 넣어주고, 이를 copyout을 통해 stack에 넣는다. 다음으로 새로 만든 쓰레드의 eip를 start_routine으로 esp를 스택의 시작 주소로 지정해 실행흐름을 넘겨준다.

이 때, ptable에 대한 lock을 좀 크게 잡는데, sz를 변경하는 중에 sbrk등 sz를 변경하려는 시도가 있으면, race condition이 발생하는 경우가 있어, 이를 변경하는 create 함수의 내부에서 lock을 잡았다.

```

for(int i=0;i<NOFILE;i++)
if(proc->ofile[i])
th->ofile[i]=filedup(proc->ofile[i]); //open file

th->cwd=idup(proc->cwd); //Current directory
safestrncpy(th->name, proc->name, sizeof(proc->name));

th->state = RUNNABLE;
int ppid = proc->pid;
for(th = ptable.proc; th < &ptable.proc[NPROC]; th++){
if(th->pid == ppid&& th->tid>=0) // find thread;
{
    th->sz = proc->sz;
}
}
//cprintf("asdf");
release(&ptable.lock);

```

<15. thread_create 프로세스 할당 부분 2>

마지막으로 기존의 fork와 유사하게, 파일을 복사하고, 상태를 RUNNABLE로 만들어 준다. 추가로 같은 pid를 가진 프로세스들의 sz를 전부 동기화 해주는데, 처음에는 growproc에만 있는 부분이었는데, 이를 create에서도 넣지 않으면 remap이 발생해 다른 쓰레드들에도 sz변경을 알려주는 부분을 추가했다.

3)thread_exit

```
void thread_exit(void *retval)
{
    struct proc *th = myproc(); // thread;

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

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

        acquire(&ptable.lock);
        // Parent might be sleeping in wait().
        wakeup1(th->tleader); // wait in thread_join
        // Pass abandoned children to init.
        struct proc *p;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->parent == th){
                p->parent = initproc;
                if(p->state == ZOMBIE)
                    wakeup1(initproc);
            }
        }
        th->retval = retval;
        // Jump into the scheduler, never to return.

        th->state = ZOMBIE;
        //cprintf("exit %d %d %d\n", th->pid, th->tid, (int)th->state);
        sched(); //release lock
        panic("zombie exit");
    }
}
```

<16. thread_exit>

기존 시스템콜 exit와 유사하게 동작한다. 먼저 열려있던 파일에 대한 참조를 종료한다. 다음으로 thread_join에서 sleep상태로 대기 중인 th->leader(쓰레드를 기다리고 있는 프로세스)를 깨워준다. 만약 이 쓰레드에서 fork 되어 나를 부모로 하는 프로세스가 있다면, 그 프로세스의 부모를 initproc으로 바꾸어 준 뒤, 만약 그 프로세스가 ZOMBIE라면 대신 처리해 줄 initproc을 깨워준다. 마지막으로, 명세에서 요구한 retval을 내 쓰레드의 retval에 저장해 join에서 가져갈 수 있게 하고, 내 상태를 ZOMBIE로 바꾼 뒤, sched를 통해 빠져나가게 한다.

4) thread_join

```
..
struct proc *th;
struct proc *proc = myproc();
int ppid = proc->pid;
int find = 0;
acquire(&ptable.lock);
for(;;){
    // Scan through table looking for exited children.
    find = 0;
    for(th = ptable.proc; th < &ptable.proc[NPROC]; th++){
        if((th->pid == ppid) && (th->tid == thread)) // find thread;
        {
            find = 1;
            //cprintf("find %d %d %d\n join", th->pid, th->tid, (int)th->state);
            if(th->state == ZOMBIE){
                // Found one.
                kfree(th->kstack);
                th->kstack = 0;
                //freevn(th->pgdir);
                th->pid = 0;
                th->parent = 0;
                th->name[0] = 0;
                th->killed = 0;
                th->state = UNUSED;
                release(&ptable.lock);
                *retval = th->retval;
                //cprintf("%d", (uint**)retval);
                return 0;
            }
        }
    }
}
if(find == 1) sleep(proc, &ptable.lock);
else if(find == 0 || proc->killed == 1)
{
    release(&ptable.lock);
    return -1;
}
}
```

<17. thread_join>

thread_join은 wait와 유사한 방식으로 매우 간단하게 구현 했다. 먼저 ptable을 순회하며, 내가 지정한 쓰레드가 현재 있는지 확인한다. 만약 그 쓰레드가 있고, ZOMBIE 상태라면, wait처럼 커널 스택의 할당을 취소해주고, 기타 요소들도 정리해 준다. 단 여기서 pgdir은 공유 중이므로 프로세스의 종료전까지 그대로 유지하도록 한다. 또, 명세의 요구대로 exit에서 지정한 반환값이 th->retval이 가리키는 주소에 있으므로 이 주소를 가져가고 0을 return 하도록 한다. 마지막으로 ptable을 한바퀴 돌았는데, 쓰레드는 있으나, ZOMBIE가 아니라면 아직 종료 되지 않은 것이므로 sleep하도록 하고, 만약 그 쓰레드가 없다면, 정상적으로 join할 수 없으므로 ptable의 lock을 풀고, -1을 반환하도록 한다.

가. 다른 시스템 콜과의 상호작용

1) fork

```

} found:
{
    p->state = EMBRYO;
    p->pid = nextpid++;
    p->tid = -1; // process;
    p->nexttid = -1;
    p->tleader = myproc();
}
np->parent = curproc;
*np->tf = *curproc->tf;
np->tleader = np;

```

<18. allocproc과 fork 변경점>

디자인한 쓰레드가 기존의 프로세스와 차이가 거의 없어, fork에서는 수정할 부분이 거의 없다. 먼저 프로세스를 할당받는 allocproc에서, 일단 프로세스를 만든다고 가정해 초기 tid를 -1로 변경해주고, nexttid도 -1부터 시작, tleader도 자기 자신으로 설정하기 때문에, fork에서는 기존의 fork처럼 부모의 주소공간을 복사하고, parent를 생성한 프로세스로 지정해 주면 된다. 또한 np->tleader = np를 통해 부모를 지정하는 것과 동일하게 자기 자신을 tleader로 지정해 준다.

2) exec

```

exit_all_thread(curproc);
curproc->tid = -1;
curproc->nexttid = -1;
curproc->tleader = curproc;

void exit_all_thread(struct proc *proc)
{
    int ppid = proc->ppid;
    int ptid = proc->tid;
    acquire(&ptable.lock);
    struct proc *th;
    for(th = ptable.proc; th < &ptable.proc[NPROC]; th++){
        if((th->pid == ppid) && (th->tid != ptid))
        {
            release(&ptable.lock);
            for(int i=0; i<NOFILE; i++){
                if(th->ofile[i]){
                    fclose(th->ofile[i]);
                    th->ofile[i]=0;
                }
            }
            begin_op();
            input(th->cwd);
            end_op();
            th->cwd = 0;
            acquire(&ptable.lock);
        }
    }
    acquire(&ptable.lock);
    wakeup1(th->parent);
    kfree(th->kstack);
    th->kstack = 0;
    //freem(th->pgdir);
    th->pid = 0;
    th->parent = 0;
    th->name[0] = 0;
    th->killed = 0;
    th->state = UNUSED;
}
release(&ptable.lock);

```

<19. exec 변경점과 exit_all_tread>

명세에서 요구한 대로, exec이 호출되었을 때, 한 스레드만 제외하고, 전부 종료 시키기 위해 exit_all_thread라는 함수를 proc.h에 정의하고 proc.c에서 선언했다. exec.c가 proc.h를 include 하고 있어 proc.c에 정의하여 사용할 수 있게 했다. 이 함수는 기존의 create_exit와 join의 부분을 합쳐 ptable을 돌며 호출한 쓰레드 외의 나머지 모든 쓰레드를 종료시키고, 자원을 회수한다. 반환 값이 지정되지 않았기에, join의 역할까지 이 함수에서 모두 진행하도록 만들었다. 다만 이 프로세스의 종료를 기다리는 부모 프로세스가 있을 수 있기에 이는 한번 wakeup1로 깨워주었다. 모두 다 종료되는데 pgdir은 남겨두는데, 이는 exec에서 old pgdir을 버리고, pgdir을 채운뒤, 기존의 것은 freemv 하기 때문이다.

exec에서는 exit_all_thread를 호출하고, 프로세스로 분화하기 위해, tid, tleader를 초기화하고, 기존의 exec와 동일한 코드를 실행한다.

3) sbrk

```
uint sz;
struct proc *curproc = myproc();
//struct proc *proc = curproc;
struct proc *th;
acquire(&ptable.lock);
uint ppid = curproc->pid;
/*for(th = ptable.proc; th < &ptable.proc[NPROC]; th++){
    if(th->pid == ppid && th->tid == -1) // find thread;
    {
        proc = th;
        cprintf("%d", proc->sz);
    }
}
*/
sz = curproc->sz;
if(n > 0){
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
        return -1;
} else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
        return -1;
}
curproc->sz = sz;

for(th = ptable.proc; th < &ptable.proc[NPROC]; th++){
    if(th->pid == ppid) // find thread;
    {
        th->sz = sz;
    }
}
release(&ptable.lock);
switchuvm(curproc);
return 0;
```

<20. sbrk>

기존의 sbrk에서 추가된 부분은 아래쪽에 ptable을 순회하는 부분이다. 공유하는 모든 쓰레드가 접근 할 수 있어야 하기 때문에, 현재 스레드와 pid가 같은 모든 쓰레드의 sz를 동기화해준다. pgdir은 공유하기 때문에 기존의 코드에서 크게 건드리는 부분은 없다. 다만 여기에서도 sz를 접근하는 부분이 커서 lock을 함수의 시작과 끝에 잡아뒀다. 그러지 않았을 경우 sz에 대한 race condition이 발생해, remap이 발생하는 것을 확인했다.

4) kill

```
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
        p->killed = 1;
        flag=1;
        // Wake process from sleep if necessary.
        if(p->state == SLEEPING)
            p->state = RUNNABLE;
    }
}
if(flag == 1)
{
    release(&ptable.lock);
    return 0;
}
release(&ptable.lock);
return -1;
}
```

<21. kill>

기존의 kill과 대부분 유사하지만, 차이점은 그 쓰레드가 속한 프로세스의 모든 쓰레드가 kill의 대상이 된다는 것이다. 따라서 ptable을 돌며, kill의 대상과 pid가 같은 모든 쓰레드를 kill 해준다. ptable 순회가 끝나면 kill이 성공했을 땐 flag가 1이므로 0을, 아니면 대상이 없는 것이므로 -1을 return 한다. 만약 sleep 상태라면 기존과 동일하게 상태를 RUNNABLE로 만들어 sleep 상태의 프로세스도 종료 될 수 있게 한다.

5) sleep, pipe

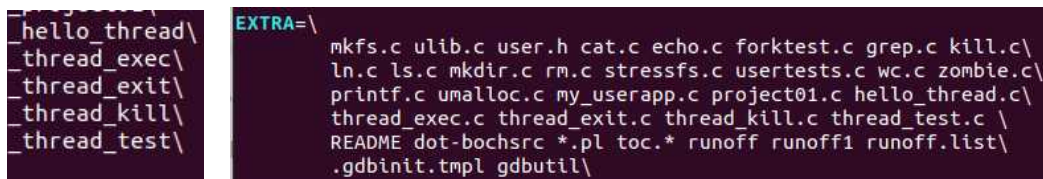
sleep과 pipe는 기존의 시스템 콜에서 변화가 없다. 구현한 쓰레드의 구조가 프로세스처럼 동작하기 때문에, sleep은 기존의 프로세스처럼 한 쓰레드의 상태를 SLEEPING으로 바꾸고 자게 되고, 자는 중에도 4)에서 언급한대로 그 쓰레드가 RUNNABLE로 바뀌어 kill의 대상이 될 수 있다.

pipe 또한 기존의 프로세스와 구조상 차이가 없기 때문에, 별다른 조작을 하지 않고 남겨두었다.

3. Result

가. 컴파일 및 실행

1) Makefile



```

hello_thread\
thread_exec\
thread_exit\
thread_kill\
thread_test\

EXTRA=\
mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
printf.c umalloc.c my_userapp.c project01.c hello_thread.c\
thread_exec.c thread_exit.c thread_kill.c thread_test.c \
README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
.gdbinit.tmpl gdbutil\

```

<22. Makefile>

Makefile의 UPROGS와 EXTRA 부분에 제공된 유저 프로그램을 추가한다.

2) 컴파일 및 실행

```

$ make clean
$ make
$ make fs.img
$ make ./bootxv6.sh
-----boot xv6-----
$ thread_test
$ thread_exec
$ thread_exit
$ thread_kill

```

<23. 컴파일 및 실행>

3) 실행결과

가) thread_test

```
$ thread_test
Test 1: Basic test
Thread 0 start
Thread 1 start
t
Thread 0 end
Parent waiting for children...
Thread 1 end
Test 1 passed
```

```
Test 3: Sbrk test
Thread Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
0 start
Test 3 passed
All tests passed!
```

```
Test 2: Fork test
Thread 0 start
ThrThread 2 start
Thread 3 start
Thread 4 start
Child of thread 0 start
Child of Child of thread 3 start
Child of thread 4 start
ead 1 start
thread 2 start
Child of thread 1 start
Child of thread 0 end
Thread 0 end
Child of thread 3 end
Child of thread 4 end
Thread 3 end
Child of thread 2 end
Child of thread 1 end
Thread 1 end
Thread 2 end
Thread 4 end
Test 2 passed
```

<24. thread_test>

첫 번째 테스트 프로그램인 thread_test는 3가지 테스트로 구성되어 있다. 그중 첫 번째 테스트는 기본적인 시스템 콜 3개 pthread_create, exit, join을 테스트한다. 먼저 2개의 스레드를 만들고, 첫 번째 스레드는 즉시 종료하고, 2번째 스레드는 2초 후 종료한다. 이후 두 스레드를 join하는데, 이 때, exit전에 join이 return 되거나, 예상하지 못한 값이 return 되었을 때, join이 0이 아닌 값이 반환되었을 때, 실패하게 된다.

테스트 결과에서는 2개의 스레드가 create되고 0번 스레드는 즉시 종료되고, 1번 스레드는 2초뒤에 종료되는데, 이후 정상적으로 join 되는 것을 알 수 있다.

두 번째 테스트에서는 5개의 스레드를 생성해 각각 fork를 실행하고, 반환 값이 0이 아니면 부모 프로세스로 스레드들은 자신이 생성한 프로세스를 wait한다. fork의 반환 값이 0이면 자식 프로세스로 start와 end를 출력하고 종료하는데, 이 때, wait하는 중인데, 기다릴 자식 프로세스가 없거나, 부모와 자식이 주소 공간을 공유하면 에러가 발생한다.

테스트 결과에서는 0~4번의 스레드가 각각 자식을 fork하고, 자신의 자식 프로세스가 종료된 후에, 부모가 종료되는 것을 확인 할 수 있다. 이를 통해 fork()가 정상적으로 작동하고, 이를 기다리는 wait()도 조건에 맞게 작동함을 알 수 있다. 또 에러가 나지 않았으므로, 부모와 자식이 주소공간을 공유하지 않고 있음을 알 수 있다.

세 번째 테스트에서는 5개의 스레드를 생성해, 0번 스레드에서는 malloc을 통해 메모리를 할당 받은 뒤, 다른 스레드들이 접근 가능한지 확인한다. 다음으로 5개의 스레드들이 각각 malloc을 통해, 메모리를 할당받고, 각각 접근한다. 이 때, 할당 받은 공간에, 스레드마다 다른 val을 넣고, 다시 한 번 돌면서, 저장된 값이 val과 다른지 확인한다. 다르다면, 할당된 공간에 다른 스레드가 값을 쓴 것이므로, 중복되는 공간을 할당 받은 것으로 에러가 발생한다.

만약 정상적으로 실행됐다면, 할당받은 공간을 free하고, 메모리를 할당 받는 과정을 2000번 반복해, 여러번 확인한다.

테스트 결과에서는 에러 없이 5개의 스레드가 정상적으로 종료 되었다. 이를 통해 sbrk를 통해 공간을 할당 받고, 이를 공유 할 수 있음을 알 수 있고, 서로 다른 스레드들이 공간을 할당 받아도, 중복 없이 각각 다른 공간을 할당 받을 수 있음을 확인했다.

세가지 테스트를 통해, 기본적인 시스템 콜과 fork, 뉴가, wait가 정상적으로 수행됨을 알 수 있다.

나) thread_exec

```
$ thread_exec
Thread exec test start
Thread 0 start
Thread 1 start
Thread 3 start
Thread 4 start
Thread 2 start
Executing...
Hello, thread!
$
```

<25. thread_exec 실행 결과>

두 번째 테스트 프로그램에서는 5개의 스레드를 생성한 뒤, 한 스레드가 exec를 통해, 자신의 주소공간을 바꿔 hello_thread를 실행한다. 이 때, 한 프로세스의 속하는 나머지 프로세스는 명세의 조건에 따라 종료 되어야 한다. 만약, 종료되지 않는다면, sleep중인 나머지 스레드가 깨어나, “This code shouldn’t be executed!!”를 출력하고 종료할 것이고 exec가 정상적으로 수행되지 않아도 같은 출력이 나올 것이다.

테스트 결과 5개의 스레드가 실행되었지만, exec가 호출된 뒤, 한 스레드를 제외하고 모두 종료되고, 남은 한 스레드는 hello_thread를 실행해 “Executing...”과 “Hello, thread!”가 출력됨을 볼 수 있다. 이를 통해 exec() 시스템 콜 또한 정상적으로 작동함을 알 수 있다.

다) thread_exit

```
$ thread_exit
Thread exit test start
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Exiting...
```

<26. thread_exit>

thread_exec와 마찬가지로 5개의 스레드를 만들고, 그 중 하나가 exit()의 대상이 된다. 이 때, 그 스레드와 같은 프로세스에 속하는 모든 스레드가 전부 종료되어야 한다. 이 때, 종료되

지 않고 남은 쓰레드가 있다면 “This code shouldn’t be executed!!”가 출력되어 잘못되었음을 알려준다.

테스트 결과에서는 5개의 쓰레드가 생성되고 한 번의 exit() 호출로 5개의 쓰레드가 한번에 종료되어 쓰레드를 대상으로 한 exit()도 정상적으로 시행됨을 알 수 있다.

라) thread_kill

```
$ thread_kill
Thread kill test start
Killing process 34
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
$
```

<27. thread_kill>

2개의 프로세스로 fork되어 나뉘진 뒤, 각각 5개의 쓰레드를 생성한다. 부모 프로세스의 쓰레드 하나가, 자식프로세스 쓰레드를 하나 kill 하는데 이 때, 그 쓰레드가 속한, 프로세스의 나머지 프로세스들은 전부 kill 되어야 하며, 그 중 하나라도 kill 되지 않는다면, “This code shouldn’t be executed!!”이 출력되어 오류를 알려 준다. 또한 부모 프로세스는 영향을 받지 않아 부모 프로세스의 쓰레드 들은 각각 “This code should be executed 5 times.”를 출력해 5줄에 걸친 출력이 나온다.

테스트 결과 자식프로세스의 쓰레드들은 전부 kill의 대상이 되었고, 부모 프로세스는 영향을 받지 않아 정상적으로 수행되었다. 또한, 자식 프로세스는 생성되고 잠시 sleep되는데, 이 때도 부모프로세스의 kill의 대상이 되어 종료 될 수 있으므로, 명세에서 요구한 sleep 상태인 프로세스도 정상적으로 kill이 됨을 알 수 있다.

4. Trouble Shooting

가. 처음에는 malloc을 통해 외부 스택을 할당 받고, 그 주소를 쓰레드 마다 저장하게 해서 각각의 스택을 가지게 하려고 했는데, malloc이 선언이 안 되어 있다는 에러가 떠서, 현재와 같은 방법으로 우회하게 되었습니다.

나. exec에서 다른 쓰레드들을 종료시킬 때, 직접 ptable을 순회하면서 종료하려고 했는데, exec.c에서는 ptable에 대한 참조를 할 수 없어서, 프로세스의 쓰레드 하나를 제외한 나머지 쓰레드를 전부 종료시키는 함수를 proc.h에 선언하고 proc.c에 정의하여 해결했습니다. 시스템 콜로 만들까라는 생각도 했지만, 번거롭고, 한 번만 호출될 함수라 이렇게 정의했습니다.

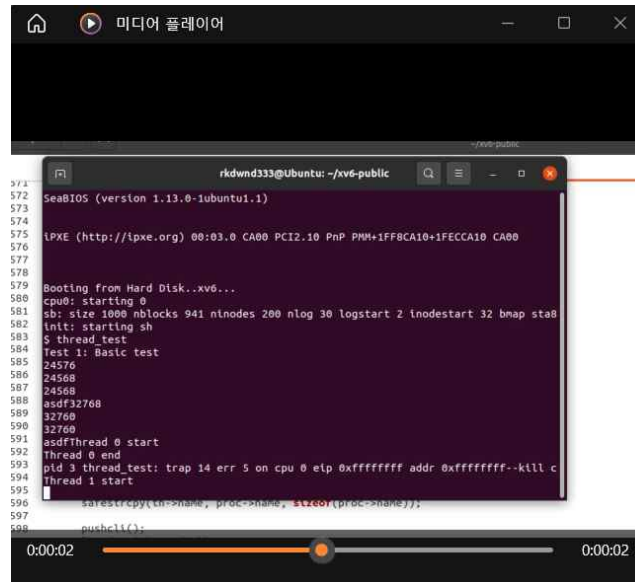
다. pthread_create에서 sz를 쓰레드마다 동기화 해야 할까? growproc할 때만 전부 동기화하고 create에서는 생성한 쓰레드와 프로세스만 하면 되지 않을까 했는데, remap이 떠서 전부 sz를 변경해주었습니다. NPROC=64번만 돌면 되기에 큰 문제는 되지 않을 것이라 생각했습니다.

라. 사소한 실수로 tid를 allocproc과 fork에서 둘 다 초기화하는 코드를 넣어 하나는 -1부터 시작, 하나는 0으로 초기화 해서, tid가 잘못 나오는 오류가 있었는데, 찾는데 오래걸렸

습니다.

마. 자료구조를 단순화 하고 싶어서 최대한 빼 보려고 했는데, 실패했습니다. 만약 하나를 빼다면 tleader를 빼고, tleader를 tid = -1 인 프로세스를 ptable을 돌며 찾는 방법을 사용할 것 같습니다.

바. create와 exec에서 오류가 발생했는데 출력이 한 번 나오고 xv6가 재부팅 되는 오류였습니다. 출력이 뭔지 알고 싶어서 캡처후 0.25 배속으로 재생해서 문제점을 파악해서 해결했습니다. start routine으로 넘어갈 때, eip, esp와 ustack의 내용이 잘못 설정되어 발생한 오류로 이를 수정해 해결 했습니다.



사. 뭔가 기능적인 측면을 알아 보기 쉽게 하고싶어서, 중복으로 사용하거나, 아무 동작도 하지 않는 코드를 넣었습니다. allocproc에서 tleader를 자기 자신으로 설정하는데, fork할 때 한 번 더 넣거나, 자신의 pgdir을 자신의 pgdir로 교체하는 코드 등이 있습니다. 쓰레드를 생성할 때, fork와 exec의 역할을 같이 함을 알아보기 쉽게 하기 위해 추가했습니다.

아. xv6의 파일 시스템을 아직 배우지 않아, 파일을 열고, 닫는 함수를 쓰레드에는 넣지 않았었는데, 쓰레드가 종료된다고 다른 쓰레드가 참조 할 가능성이 있는 파일을 닫는게 이상해서 그런 생각을 했습니다. 마지막으로 코드를 정리할 때, 문득 생각이 나서, xv6 문서를 읽어봤는데, 참조 수만 줄이고 늘리고, 참조하는 프로세스가 없으면 종료하는 방식이어서, 그 부분을 추가했습니다.

자. lock을 최대한 작은 범위에만 걸고 싶었는데, 걸지 않아도 돌아가는 부분을 한쪽으로 몰기엔 가독성이 떨어져서, 최대한 조정하는 선에서 그쳤습니다.

II. Lock

1. Design

가. 구현목표

기존 pthread의 api (pthread_mutex)를 사용하지 않고, lock을 구현 해 critical section에 접근을 막고 race condition을 해결하는 것이 목표이다.

나. 생각한 구현 방법

1) 피터슨 알고리즘이나 세마포어 사용

강의 시간에 배운 피터슨 알고리즘이나 세마포어를 사용해 P(), V()로 감싼다면 해결 될 것이라 생각하였으나 P, V 호출이 atomic 하거나 앞,뒤로 인터럽트를 끄고 켜서, timeout이 발생하지 않도록 해야 lock 대한 race condition이 생기지 않는다. 하지만 C에서는 인터럽트를 끄고 켜는 함수를 제공하지 않아 이는 불가능하다.

또 피터슨 알고리즘이 강의에서는 2개의 프로세스로 생각했는데, 프로세스가 여러 개일 때는, 다른 프로세스들이 들어가려고 하는지 확인하는 절차가 복잡해진다. 하지만 무엇보다도 락을 걸고 해제하는 함수가 atomic하게 만들 수 없어 다른 방법을 생각했다. 스레드의 수와 반복 수가 적을 때는 성공했으나 숫자가 커지니 lock에대한 race condition이 생겼다.

2) inline assembly

C에서는 atomic한 함수를 지원하지 않는다고 해서, 어셈블리에서 제공하는 xchg는 atomic 하게 두 데이터를 swap 할 수 있다고 해, 이를 C코드에 인라인으로 넣으면 되지 않을까 라는 생각이 들었다.

3) <stdatomic.h>

하드웨어의 도움으로 atomic swap이나 test and set으로 lock을 구현 할 수 있다는 내용을 강의 시간에 배웠는데, 이를 지원하지 않을까 해서 찾아보았는데, standard library header <stdatomic.h> 가 있었다. 여기서 atomic_exchange라는 atomic한 exchange를 보장하는 함수를 제공한다. 또는 atomic_compare_exchange를 사용해 만들 수 있을 것이라 생각했다. 문서와 헤더 파일을 읽어본 결과 ptr의 값을 val로 바꾸고, 기존에 저장되어있던 값을 반환한다고 한다.

```
#define atomic_exchange_explicit(PTR, VAL, MO) \
__extension__ \
({ \
    __auto_type __atomic_exchange_ptr = (PTR); \
    __typeof__ ((void)0, *__atomic_exchange_ptr) __atomic_exchange_val = (VAL); \
    __typeof__ ((void)0, *__atomic_exchange_ptr) __atomic_exchange_tmp; \
    __atomic_exchange (__atomic_exchange_ptr, &__atomic_exchange_val, \
        &__atomic_exchange_tmp, (MO)); \
    __atomic_exchange_tmp; \
})
```

<29. atomic_exchange>

- Process P_i

```

do {
    /* key: me  lock: he */
    key = true;    /* My intention */
    while (key == true) Swap(lock, key);
    critical section
    lock = false;
    remainder section
}

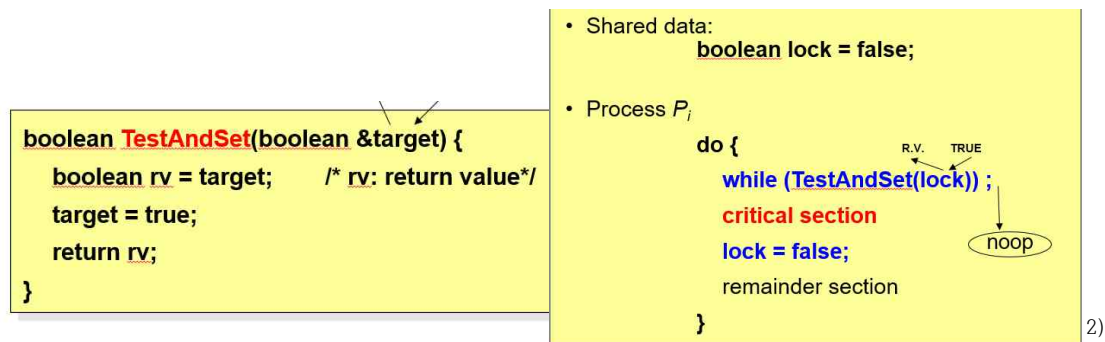
```

Did he set lock ?

1)

<30. atomic swap을 이용한 lock>

atomic swap을 이용한 lock은 다음과 같은 방식으로 이루어진다. key가 true인 동안 lock이 0일 때, swap을 호출하면 key = false가 되고, lock은 true가 되어, 락을 획득해 critical section으로 들어간다. 이 때, 이 swap이 atomic하게 일어나 swap 되는 동안에는 다른 스레드가 lock의 값에 접근할 수 없다. swap 이후에는 lock의 값이 true가 됨으로 swap을 호출해도 1과 1을 교환 하는 것이기 때문에, unlock 되기 전까지는 다른 프로세스가 lock을 획득할 수 없다.



2)

<31. TestAndSet을 이용한 lock>

atomic_exchange를 이용한 lock은 함수의 특성상 TestAndSet을 이용한 lock과 비슷하게 작동시킬 수 있다. atomic_exchange는 target을 항상 true로 만들고 기존의 값을 반환하는데, atomic_exchange도 target의 값을 지정하고, 기존에 저장되어 있던 값을 반환한다는 점에서 유사하게 구현 할 수 있다.

TestAndSet을 이용한 lock에서는 lock을 true로 만들고 기존의 lock 값을 반환한다. 만약 lock이 false였다면, false가 return 되어 lock을 획득하고, critical section에 들어갈 수 있다. 이때, TestAndSet이 atomic하게 동작하기 때문에, 일련의 과정에서, 다른 스레드들은 lock 값에 접근하지 못하고, 한 스레드가 lock을 획득한 후에는 lock이 true가 되기 때문에 unlock이 호출되어 lock이 false가 되기 전에는 lock을 획득할 수 없다

1) 강의 자료에서 발췌 (Synchronization 1)

2) 강의 자료에서 발췌 (Synchronization 1)

2. Implement

```
atomic_int lck = 0;
#define NUM_ITERS 50000
#define NUM_THREADS 296

void lock();
void unlock();

void lock()
{
    while(atomic_exchange(&lck, 1))
    {
        sleep(0.1);
    }
}

void unlock()
{
    atomic_store(&lck, 0);
}
```

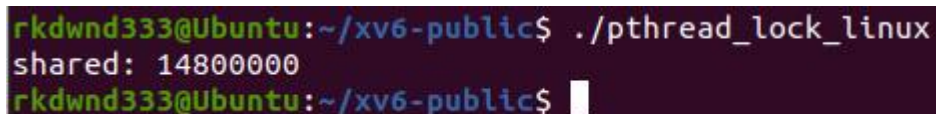
<32. 구현한 lock>

구현은 매우 간단하다. lck 라는 atomic_int 변수를 선언하고, lck가 0일 때는, 락이 해제된 상태, 1일 때는 걸린 상태로 정의 한다. lock에서는 lck의 값을 1로 변경한다. 이 때, 만약 락이 걸린 상태가 아니라면, 기존 lck 값인 0을 반환해, while을 빠져나가고, critical section으로 들어 갈 수 있다. 이 atomic_exchange가 atomic하게 발생하기 때문에, 다른 스레드는 이 사이에 lock을 획득 할 수 없다.(lck에 대한 race condition 발생 하지 않음.) 만약 한 스레드가 락을 획득하면 lck = 1이 되어 다른 스레드는 atomic_exchange에서 1이 return 되어 락을 획득할 수 없다.

while문을 한 번 돌고 sleep하는데, spinlock으로 계속 기다리는게 기다리는 동안 락이 해제 될 수 있는 멀티프로세서에서는 유용하겠지만, 계속 잡고 있지 않게 하기 위해 한 번 확인하고, 잠시 sleep해 스케줄링 되게 만들었다. sleep 시간이 너무 길면, lock이 해제 되어도 다른 프로세스가 전부 자고 있기에 느려지는 경우가 있어 짧게 sleep하도록 했다.

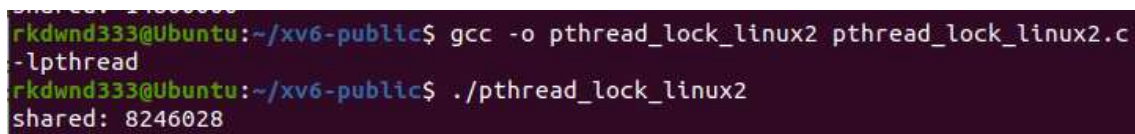
다음으로 unlock() 또한 atomic_store를 통해 lck를 0으로 만들어 락을 해제한다.

3. Result



```
rkdwnd333@Ubuntu:~/xv6-public$ ./pthread_lock_linux
shared: 14800000
rkdwnd333@Ubuntu:~/xv6-public$
```

<33. lock 사용>



```
rkdwnd333@Ubuntu:~/xv6-public$ gcc -o pthread_lock_linux2 pthread_lock_linux2.c -lpthread
rkdwnd333@Ubuntu:~/xv6-public$ ./pthread_lock_linux2
shared: 8246028
rkdwnd333@Ubuntu:~/xv6-public$
```

<34. lock 미사용>

NUM_ITERS = 50000, NUM_THREADS = 296으로 설정해 lock을 사용한 pthread_lock_linux에서는 50000*296 = 14,800,000 이 출력되었고, lock을 사용하지 않은 pthread_lock_linux2를 만들어 테스트해 본 결과 이 때는 race condition이 발생해 그보다 작은 값을 나타냈다.

4. Trouble Shooting

가. 어셈블리로 넣고 싶었는데, 여건이 좋지 않아 다음에 시도해 보겠습니다.

나. 기존에 피터슨으로 만들었는데, 수가 작을 때는, switching 되기 전에 모두 끝나 잘 되는 듯했으나, 개수가 늘어나니, race condition이 발생했습니다.

다. locking도 xv6환경에서 돌아가도록 만들어야 하는 줄 알고, 그에 대한 오해가 있었습니다.

라. 사실 lock을 해제하는 부분에는 `lck = 0;` 으로만 적어도 괜찮습니다. 락을 획득하고자 할 때, lock을 확인하고, lock을 획득,변경하고, 크리티컬 섹션에 들어가는 과정이 atomic 하지 않을 때, mutual exclusion이 보장이 안 되는 것이지, lock을 해제할 때는 이런 과정이 없이 lock을 0으로만 바꾸는 것이기 때문에 상관 없고, 결과로도 확인했습니다. 강의에서도 lock을 해제하는 부분은 단순히 `lock = 0`으로 구현했었습니다. 다만, `atomic_store`를 써 보고 싶어 사용했습니다.

마. `atomic.h`의 버전이 맞지 않을 수도 있다고 생각해 비교적 최근 추가된 `_Atomic` 키워드 대신에 `atomic_int` 타입의 정수를 선언했습니다.

바. `volatile`이라는 키워드가 c에 있어, 찾아보았는데, 가장 사용 빈도가 적은 키워드로 캐시를 거치지 않고, 항상 메모리로 접근 해 최신화 된 값을 받아 오는 키워드가 있었습니다. 이를 통해 lock을 구현 할 수 있을 거라 생각했는데, 읽어오고, 값을 변경하고, 다시 저장하는 일련의 과정에서 스케줄링이 되면, 이 또한 race condition이 발생했습니다.