Project04

LWP

1. LWP란?

- LWP란 `Light Weight Process`의 줄임말으로, 서로 자원과 주소공간을 공유한다.
- 보통 Thread라고 많이 불리므로, 본 wiki에서는 Thread라고 부르도록 하겠다.
- Thread를 사용함으로써 가지게 되는 장점은 다음과 같다:
    1. 자원을 공유하므로 메모리 사용량이 낮다.
    2. 새로운 프로세스를 fork하게 되면 자원을 복사해야하기 때문에 overhead가 커지지만, thread를 만들면 자원을 공유하기 때문에 overhead가 낮다.
    3. 유저 레벨에서 멀티태스킹이 가능해진다.

2. Thread 구현

- 각 Thread를 `Light Weight Process`로 보았다. 이 말은, 모든 쓰레드는 `proc.h`의 `struct proc`이며 프로세스가 `thread`를 만들지 않는다면 그 프로세스는 `single threaded process`, 만들었다면 `multi threaded process`로 보았다. 이를 위하여, `struct proc`에 몇가지 `elements`를 추가하였다.

```
struct proc {

... other elements

        int tid; // Thread ID

        void* retval; // Return value

        uint upper_bound; // Upper bound of stack

        int is_main; // Is main thread

};
```

- 이와 같이 구현한 이유는 다음과 같다:
    1. xv6는 모두 `process`기반으로 동작하는데, 새롭게 `struct thread`를 정의한다면 xv6의 모든 함수에서 `struct proc`에 대한 함수 대신 `struct thread`에 대한 함수로 바꿔주어야하여 구현에 대한 난이도가 높아진다.
    2. `pde_t* pgdir`이 포인터이기 때문에 프로세스끼리 주소 공간을 공유하는 것에 대한 overhead가 없다.
- 새롭게 추가한 `elements`에 대한 `description`은 다음과 같다:

1. `int tid`: 프로세스가 pid를 가지는 것처럼, `thread`는 tid를 가지게 된다.
2. `void* retval`: `thread`가 return하는 값이다.
3. `uint upper_bound`: `process`가 얼만큼 메모리를 차지하고 있는지에 대한 변수이다.
4. `int is_main`: 1일 때 맨 처음 만들어진 thread임을 표시한다.

- 구현의 편의성을 위하여 `main`이라는 개념을 도입하였다.
  - 가장 처음 만들어진 `process/thread`가 `main thread`이다.
  - `is_main == 1`이면 `main thread`이고, `main thread`의 `upper_bound`를 기준으로 `user stack allocation`과 `user stack deallocation`이 일어난다.
  - 모든 자원 할당과 회수는 `main thread`에서 일어난다.

3. 새로운 System Call과 함수

> 모든 Thread 관련 함수들은 `thread.c`에 구현되어있다.

I. `static struct proc* allocthread(void)`

```
static struct proc*

allocthread(void) // allocate thread and kernel stack

{

        struct proc* curproc = myproc();

        struct proc *p;

        char *sp;



        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

                if(p->state == UNUSED)

                        goto found;

        }

        return 0;

found:

        p->state = EMBRYO;

        p->pid = curproc->pid;
```

```
        p->tid = nexttid++;

        if((p->kstack = kalloc()) == 0){

                p->state = UNUSED;

                return 0;

        }

        sp = p->kstack + KSTACKSIZE;



        sp -= sizeof *p->tf;

        p->tf = (struct trapframe*)sp;



        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;

}
```

- `proc.c`의 allocproc을 참고하여 만들었다.
- `thread`에 `kernel stack`을 할당한다.
- `nexttid`는 `thread.c`에 전역변수로 선언되어있다.

II. `int thread_create(thread_t *thread, void (*start_routine)(void*), void *arg)

```
int

thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg)

{
```

```
        int i;
        struct proc *np;
        struct proc* curproc = myproc();

        uint sz, sp, ustack[2];
        acquire(&ptable.lock);
        struct proc *main = findmain(myproc());

        if((np = allocthread()) == 0) {
        goto bad;
        }

        main->upper_bound = PGROUNDUP(main->upper_bound);
        if((main->upper_bound = allocuvm(main->pgdir, main->upper_bound, main-
>upper_bound + 2*PGSIZE)) == 0) {
                kfree(np->kstack);
                np->kstack = 0;
                cprintf("allocuvm failed\n");
                goto bad;
        }
        clearpteu(main->pgdir, (char*)(main->upper_bound - 2*PGSIZE));
        np->sz = main->upper_bound;
        sz = main->upper_bound;
        sp = sz;
        ustack[0] = 0xffffffff;
```

```
        ustack[1] = (uint)arg;

        sp -= sizeof(ustack);

        if(copyout(main->pgdir, sp, ustack, sizeof(ustack)) < 0){

        cprintf("copyout failed\n");

        goto bad;

        }

        np->parent = main->parent;

        *np->tf = *curproc->tf;

        np->tf->eip = (uint)start_routine; // thread should start at
start_routine

        np->tf->esp = sp;

        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));


        np->pgdir = main->pgdir;


        upperboundsync();

        *thread = np->tid;

        np->state = RUNNABLE;

        release(&ptable.lock);

        return 0;
```

```
bad:
        np->state = UNUSED;
        release(&ptable.lock);
        return -1;
}
```

- `proc.c`의 `fork`, `exec.c`의 `exec`를 참고하여 구현하였다.
- `findmain` 함수로 `main thread`를 찾은 다음 `main thread`의 `upper_bound` 위로 새로운 `thread`의 `sz`를 잡아준다. `sz`를 각 `thread`가 "메모리를 얼마나 차지하고 있느냐"보다 "각 `thread`의 `frame pointer`"를 나타내게 하였다.
- 페이지 공유를 위하여 각 `thread`가 같은 `pgdir`를 가지게 하였다.
- 각 `thread`는 `start_routine`에서 시작하여야 하므로, `program counter`를 가리키는 `eip`를 `start_rouine`을 가리키게 하였으며 그것에 대한 `argument`인 `arg`를 `user stack`에 넣어주었으며, `stack pointer`가 `arg`를 가리키게 하였다.
- `memory allocation`과정에서 문제가 생기면 -1을 return, 문제가 없다면 0을 return하게 하였다.

III. `int thread_exit(void *retval)

```
void
thread_exit(void *retval)
{
        struct proc *curproc = myproc();
        struct proc *main = findmain(curproc);
        struct proc *p;
        int fd;
        // 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->retval = retval;

        curproc->cwd = 0;


        acquire(&ptable.lock);

        // Parent might be sleeping in wait().

        wakeup1(main);


        // Pass abandoned children to init.

        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

                if(p->parent == curproc){

                        p->parent = main;

                        if(p->state == ZOMBIE)

                                wakeup1(main);

                }

        }

        // Jump into the scheduler, never to return.

        curproc->state = ZOMBIE;


        sched();

        panic("zombie exit");
```

```
}
```

- `proc.c`의 `exit`함수를 참고하여 구현하였다.
- 해당 `thread`를 `zombie`상태로 바꿔주어 더 이상 `scheduling`이 일어나지 않게 하였다.
- `file`을 닫아주고 `main thread`가 자고있으면 `main thread`를 `wakeup`하여 자원 회수를 하게 만들었다.

iv. **`int thread_join(thread_t thread, void **retval)`**

```
int

thread_join(thread_t thread, void **retval)

{

        struct proc *p;

        struct proc *curproc = myproc();

        struct proc *main = findmain(curproc);

        acquire(&ptable.lock);

        for(;;){

                for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

                if(p->state == ZOMBIE && p->tid == thread && p->is_main == 0 &&
p->pid == curproc->pid){

                        kfree(p->kstack);

                        p->kstack = 0;

                        p->pid = 0;

                        p->parent = 0;

                        p->name[0] = 0;

                        p->killed = 0;

                        p->state = UNUSED;

                        *retval = p->retval;

                        release(&ptable.lock);
```

```
                        return 0;
                }
        }
        // Wait for children to exit. (See wakeup1 call in proc_exit.)
        sleep(main, &ptable.lock); //DOC: wait-sleep
        }
}
```

- `proc.c`의 `wait`을 참고하여 만들었다.
- `main`이 자원을 회수한다.
- 모든 `thread`는 같은 `pid`를 가지고 있으므로, `ptable`를 돌면서 `zombie`상태이면서 `pid`가 같은 `thread`의 자원을 회수하는 과정을 거친다.
- `retval`을 `thread`의 `retval`로 바꿔준다.

v. `void upperboundsync(void)`

```
void
upperboundsync(void)
{
        struct proc* curproc = myproc();
        struct proc* main = findmain(curproc);
        for(struct proc* p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                if(p->pid == curproc->pid) {
                        p->upper_bound = main->upper_bound;
                }
        }
}
```

- 현재 `thread`와 같은 `pid`를 가지는 `thread`의 `upper_bound`를 모두 똑같이 맞춰준다.

- `sbrk`를 위하여 구현한 함수이다.

4. 기존 System Call 및 함수 수정

I. `int fork(void)`

```
int

fork(void)

{
        ... 기존 코드

        np->sz = curproc->sz;

        np->parent = curproc;

        *np->tf = *curproc->tf;

        np->upper_bound = np->sz; // upper_bound를 sz로 바꿈



// Clear %eax so that fork returns 0 in the child.

        np->tf->eax = 0;

        ...기존 코드
}
```

- 크게 수정하지 않았다.
- `upper_bound`에 `sz`를 넣어준 부분을 추가하였다.

II. `int wait(void)`

```
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.
                if(p->is_main == 1) {
                        freevm(p->pgdir);
                }
                p->killed = 0;
                pid = p->pid;
                kfree(p->kstack);
                p->kstack = 0;
                p->pid = 0;
                p->parent = 0;
                p->name[0] = 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;
}
```

- 크게 수정하지 않았다.
- 만약에 `main`이 `zombie`상태라면, `freevm`함수를 호출하여 자원을 정리하도록 하였다.

iii. `int kill(int pid)`

```
int
kill(int pid)
{
        struct proc *p;
        int flag = 0;
        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;
                        flag = 1;
                }
        }
```

```
        release(&ptable.lock);
        if(flag == 1)
                return 0;
        return -1;
}
```

- 모든 `ptable`을 돌면서 pid가 같은 `process`의 `killed`를 1로 초기화하였다.
- 이렇게 되면 process들이 scheduling되면 `trap`이 걸려 자기 자신을 `exit()`하게 한다.

iv. `int growproc(int n)`

```
int
growproc(int n)
{
        uint sz;
        struct proc *main;
        struct proc *curproc = myproc();
        upperboundsync();
        int flag = 0;
        if(curproc->is_main == 1) {
                main = curproc;
        } else {
                main = findmain(curproc);
                flag = 1;
        }
        sz = main->upper_bound;
        if(n > 0){
                if((sz = allocuvm(main->pgdir, sz, sz + n)) == 0) {
```

```
                cprintf("allocuvm failed\n");
                release(&ptable.lock);
                return -1;
                }
        } else if(n < 0){
                if((sz = deallocuvm(main->pgdir, sz, sz + n)) == 0){
                cprintf("deallocuvm failed\n");
                release(&ptable.lock);
                return -1;
                }
        }
        if(flag == 1) {
                main->upper_bound = sz;
        }
        else main->sz = sz;
        upperboundsync();
        switchuvm(curproc);
        return 0;
}
```

- `sbrk`를 위한 수정이다.
- `main thread`의 `upper_bound`부터 `allocation`이 이루어지도록 하였다.
- 마지막에 `sbrk`를 위하여 같은 `pid`를 가지는 모든 `thread`의 `upper_bound`를 맞추어주었다.

v. `int exec(char *path, char **argv)`

```
int
exec(char *path, char **argv)
```

```
{
... 기존 코드

        struct proc *curproc = myproc();

        int fd;

        curproc->is_main = 1;

        for(struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {

                if(p->pid == curproc->pid && p != curproc) {

                        p->state = UNUSED;

                        for(fd = 0; fd < NOFILE; fd++){

                                if(p->ofile[fd]){

                                        fileclose(p->ofile[fd]);

                                        p->ofile[fd] = 0;

                                }

                        }

                }

        }


        begin_op();

...기존 코드

}
```

- exec를 호출하면 모든 스레드를 정리하도록 하였다.
- 현재 `thread`를 `main`로 만들고 나머지 스레드를 정리한다.

vi. `int sys_sbrk(void)`

```
int

sys_sbrk(void)
```

```
{
        acquire(&ptable.lock);
        int addr;
        int n;

        if(argint(0, &n) < 0)
                return -1;
        addr = myproc()->upper_bound;
        if(growproc(n) < 0)
                return -1;
        release(&ptable.lock);
        return addr;
}
```

- sbrk를 호출하면 upper_bound를 주고 거기서 allocation이 일어나도록 하였다.
- 여기서 `race condition`이 일어날 수 있기 때문에 앞 뒤로 `ptable.lock`을 잡아주었다.
- 원래는 `addr = myproc()->sz`였는데, 설계상 `sz`가 `frame pointer`역할을 하므로 `upper_bound`에서 `allocation`이 일어나도록 하였다.

sleep, pipe는 수정하지 않았다.

5. test 결과

I. thread_test.c

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

Test 2: Fork test
Thread 0 start
TThread 2 start
Thread 3 start
Thread 4 start
Child of thread 2 start
Child of thread hread 1 start
4 start
Child of thread 1 start
Child of thread 3 start
Child of thread 0 start
Child of thread 2 end
Thread 2 end
Child of thread 4 end
Thread 4 end
Child of thread 1 end
Thread 1 end
Child of thread 0 end
Child of thread 3 end
Thread 0 end
Thread 3 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!
```

II. thread_exec.c

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

III. thread_exit.c

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

IV. thread_kill.c

```
$ Thread kill test start
Killing process 17
This code should be executed 5 timThis code should be executed 5 times.
This code should be executed 5 times.
This code should be eThis code should be executed 5 times.
es.
xecuted 5 times.
Kill test finished

$
```

모든 테스트를 통과하는 것을 볼 수 있다.

Lock

```
typedef struct {

        volatile int flag;

} spinlock_t;



spinlock_t spinlock = {0};



void lock(spinlock_t *lock) {

        __asm__ volatile (

                "1:\n\t"

                "movl $1, %%eax\n\t"

                "xchg %%eax, %0\n\t"

                "test %%eax, %%eax\n\t"

                "jnz 1b\n\t"

                : "=m"(lock->flag)

                : "m"(lock->flag)

                : "eax", "memory"

        );
```

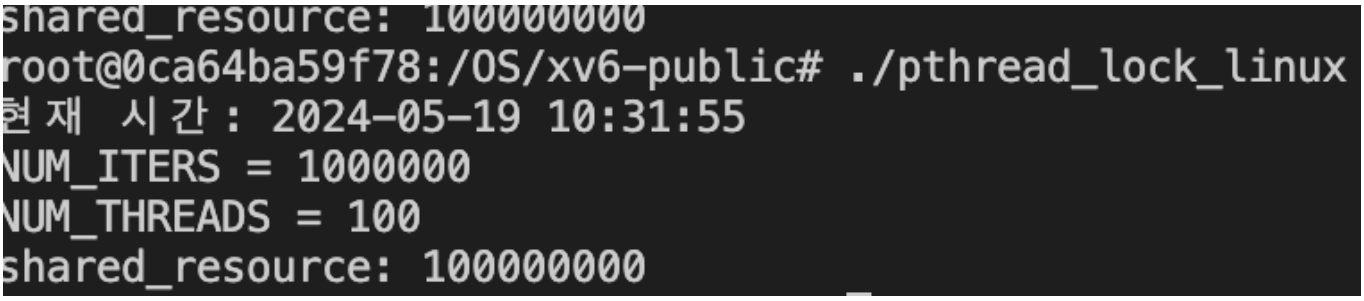
```
}



void unlock(spinlock_t *lock) {
        __asm__ volatile (
                "movl $0, %0\n\t"
                : "=m"(lock->flag)
                :
                : "memory"
        );
}
```

- `xchg` instruction을 사용하여 `swap` 형태의 `spinlock`을 구현하였다.
- 파일은 xv6-public에 존재한다.

Test 결과

1. NUM_ITERS = 1000000, NUM_THREADS = 100

```
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:30:41
NUM_ITERS = 1000000
NUM_THREADS = 100
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:30:54
NUM_ITERS = 1000000
NUM_THREADS = 100
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:31:01
NUM_ITERS = 1000000
NUM_THREADS = 100
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:31:09
NUM_ITERS = 1000000
NUM_THREADS = 100
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:31:16
NUM_ITERS = 1000000
NUM_THREADS = 100
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:31:25
NUM_ITERS = 1000000
NUM_THREADS = 100
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:31:33
NUM_ITERS = 1000000
NUM_THREADS = 100
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:31:47
NUM_ITERS = 1000000
NUM_THREADS = 100
```



여러번 테스트 해도 모두 옳은 결과가 나오는 것을 볼 수 있다.

2. NUM_ITERS = 100000, NUM_THREADS = 1000

```
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:32:49
NUM_ITERS = 100000
NUM_THREADS = 1000
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:32:51
NUM_ITERS = 100000
NUM_THREADS = 1000
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:32:52
NUM_ITERS = 100000
NUM_THREADS = 1000
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:32:53
NUM_ITERS = 100000
NUM_THREADS = 1000
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:32:54
NUM_ITERS = 100000
NUM_THREADS = 1000
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:32:55
NUM_ITERS = 100000
NUM_THREADS = 1000
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:32:56
NUM_ITERS = 100000
NUM_THREADS = 1000
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:32:56
NUM_ITERS = 100000
NUM_THREADS = 1000
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:33:13
NUM_ITERS = 100000
NUM_THREADS = 1000
```

```
NUM_THREADS = 1000
shared_resource: 100000000
```

3. NUM_ITERS = 10000, NUM_THREADS = 10000

```
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:37:20
NUM_ITERS = 10000
NUM_THREADS = 10000
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:37:21
NUM_ITERS = 10000
NUM_THREADS = 10000
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:37:24
NUM_ITERS = 10000
NUM_THREADS = 10000
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:37:26
NUM_ITERS = 10000
NUM_THREADS = 10000
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:37:27
NUM_ITERS = 10000
NUM_THREADS = 10000
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:37:28
NUM_ITERS = 10000
NUM_THREADS = 10000
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:37:30
NUM_ITERS = 10000
NUM_THREADS = 10000
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:37:31
NUM_ITERS = 10000
NUM_THREADS = 10000
shared_resource: 100000000
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:37:33
NUM_ITERS = 10000
NUM_THREADS = 10000
shared_resource: 100000000
```

```
root@0ca64ba59f78:/OS/xv6-public# ./pthread_lock_linux
현재 시간: 2024-05-19 10:37:37
NUM_ITERS = 10000
NUM_THREADS = 10000
shared_resource: 100000000
```

Trouble Shooting

1. `thread_test.c`에서 `sbrk` 테스트를 할 때 growproc 전 `upper_bound`에 `race condition`이 생겨 같은 `upper_bound`에서 두 번 이상 `allocation`이 일어나 `trap 14`가 자주 나왔으며, 이를 같은 pid를 가지는 `thread`의 `upper_bound`를 동기화 하고 앞 뒤로 `ptable.lock`을 걸어주는 것으로 해결하였음.