

Project03

1. POSIX thread 를 구현해야 함.

A. API

- i. thread_create
- ii. thread_exit
- iii. thread_join

POSIX thread API

1. Desgin

A. 기존 fork 디자인

- i. 프로세스를 할당 해 준다.
- ii. 기존 프로세스에서의 프로세스 메모리 사이즈와 pgdir(page table)을 복사한다.
- iii. 스택은 초기화만 하여 두고, trapframe 은 동일하게, parent는 기존 프로세스로 한다.
- iv. ofile의 값도 옮겨주고, 모든 초기 값을 세팅한다.
- v. 사실상 kstack을 제외한 모든 값을 복사해 온다.

B. 기존 exec 디자인

- i. 해당 프로세스를 초기화하고, code, data, stack 영역을 새로이 할당한다.

C. 기존 exit 디자인

- i. 열었던 ofile 값을 다 닫아준다.
- ii. 자고있던 현재 프로세스의 부모 프로세스를 깨워준다.
- iii. 부모 프로세스를 현재의 프로세스로 바꾸어준다.
- iv. 현재 프로세스를 죽비상태로 바꿔준다.

D. 기존 wait 디자인

- i. 자식프로세스가 없으면 -1을, 있다면 pid를 리턴하고 자식을 위해 wait상태에 들어간다.

E. thread_create

- i. create하는 부분에 fork와 exec의 기능을 둘 다 할 수 있도록 만듬. 기존의 코드를 대부분 사용하고, 따로 만들어준 스레드 아이디를 최대한 활용함.
- ii. 새로운 프로세스 np를 할당한다. 그리고 curproc의 page table을 비롯한 정보들을 초기화 해 준다. tid는 count할 수 있도록 threadNumCount변수를 하나 만들어서 하나씩 더해준다. ustack을 활용해 쓰레드의 stack공간을 할당해 준다. 할당한 공간은 fake return PC와 arg값을 넣어 쓰레드가 활용할 수 있게 한다. 넣어줄 때마다 sz값을 4씩 빼주어 공간을 만들어 준다.

F. thread_exit

- i. 기존 exit과 동일하게 열었던 ofile을 닫아주고 기존의 값을 retval에 집어 넣어준다. 그리고 이전 부모를 깨워준다. 그리고 현재 스레드의 상태를 zombie로 변경해준다.

G. thread_join

- i. 기존 wait과 동일하게 쓰레드의 자원을 정리해준다.

2. Implement

A. thread_create

```

int thread_create(thread_t* thread,
                  void* (*start_routine)(void*), void* arg){
    struct proc *np;
    struct proc *curproc = myproc();
    uint ustack[2];
    int i;

    if((np=allocproc()) == 0){
        return -1;
    }

    curproc->sz = PGROUNDUP(curproc->sz);
    if((curproc->sz= allocuvm(curproc->pgdir,
                                 curproc->sz, curproc->sz + 2* PGSIZE)) == 0){
        return -1;
    }

    clearpteu(curproc->pgdir, (char*)(curproc->sz - 2*PGSIZE));

    if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz))==0){
        kfree(np->kstack);
        np->kstack = 0;
        np->state = UNUSED;
        return -1;
    }
}

```

```

np->tid = curproc->threadNumCount++;
np->pgdir = curproc->pgdir;
np->sz = curproc->sz;
np->parent = curproc;
*np->tf = *curproc->tf;

*thread = np->tid;

ustack[0] = 0xffffffff;
curproc->sz -= 4;
ustack[1] = (uint)arg;
curproc->sz -= 4;

if(copyout(np->pgdir, curproc->sz, ustack, 2*4)<0){
    return -1;
}

np->tf->eax = 0;
np->tf->eip = (uint)start_routine;
np->tf->esp = curproc->sz;

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));
acquire(&ptable.lock);
np->state = RUNNABLE;

release(&ptable.lock);

return 0;
}

```

- i. 기존의 fork에 있는 코드와 exec에 있는 코드를 최대한 활용했다. 추가된 것은 쓰레드 id를 지정해주는 부분이다. 그리고 할당될때마다 쓰레드 번호를 하나씩. 추가하며 달 수 있도록 해준 것이다.

B. thread_exit

```
void thread_exit(void *retval){
    struct proc *curproc = myproc();
    int fd;

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

    if(curproc->tid == 0){
        return;
    }
    curproc->retval = retval;
    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;

    acquire(&ptable.lock);

    wakeup1(curproc->parent);

    curproc->state = ZOMBIE;
    curproc->threadNumCount--;

    sched();
    panic("zombie exit");
}
```

- i. 기존의 exit코드를 최대한 활용했고, retval를 만들어서 쓰레드의 내용들을 지정해주었다. 열린 파일을 모두 닫아주고 상태를 zombie로 바꾸어주었다.

C. thread_join

```
1 int thread_join(thread_t thread, void** retval){
2     struct proc *p;
3     struct proc *curproc = myproc();
4
5     if(curproc->tid!=0){
6         return -1;
7     }
8     acquire(&ptable.lock);
9
10    for(;;){
11        for(p = ptable.proc; p<&ptable.proc[NPROC];p++){
12            if(p->parent != curproc)
13                continue;
14            if(p->state == ZOMBIE && p->tid == thread){
15                *retval = p->retval;
16
17                kfree(p->kstack);
18                p->kstack = 0;
19                p->pid = 0;
20                p->parent = 0;
21                p->name[0] = 0;
22                p->killed = 0;
23                p->state = UNUSED;
24                p->tid = 0;
25                release(&ptable.lock);
26                return 0;
27            }
28        }
29    }
30
31    if(curproc->killed){
32        release(&ptable.lock);
33        return -1;
34    }
35
36    sleep(curproc, &ptable.lock);
37 }
38 }
```

- i. 기존 wait의 코드를 최대한 활용했다. 기존 스레드가 갖고있는 값을 모두 정리해주었다. 그리고 현재 프로세스를 sleep상태로 바꾸어준다.

3. Result

A. thread_test

i. Test 1

```
$ thread_test
Test 1: Basic test
Thread 0 start
Thread 0 end
pid 4 thread_test: trap 14 err 5 on cpu 0 e
Thread 1 start
Parent waiting for children...
Thread 1 end
Test 1 passed
```

ii. Test 2

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

iii. Test 3

```
Test 3: Sbrk test
Thread 0 start
lapicid 0: panic: remap
80106da7 801071d9 80103aef 80105b6f 80104e6d 80106041 80105d84 0 0 0
```

1. Test3에서는 panic:remap 오류가 발생하여 종료되지 않았으나 확실히 처리하지 못했다.

B. Thread_exec

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

C. Thread_exit

```
$ thread_exit
Thread exit test start
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Exiting...
This code shouldn't be executed!!
```

- i. 끝부분에 제대로 종료되지 않았지만 확실히 처리하지 못했다.

D. Thread_kill

```
$ thread_kill
Thread kill test start
Killing process 4
This code should be executed 5 times.
pid 5 thread_kill: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0xffff
This codThis code should be executed 5 times.
This code should be executed This code should be executed 5 times.
e should be executed 5 times.
5 times.
Kill test finished
```

4. Trouble shooting

- A. 3번 test를 통과하지 못했는데 아마 할당 이후에 exit하면서 제대로 할당을 해제하지 못한 이유일거라 판단된다. 추가적으로 비어있는 stack 영역을 관리할 수 있는 공간을 만들어서 할당 해제된 것들을 판단하고 다시 다른 쓰레드가 불릴때 적당히 다시 할당할 수 있도록 해 주어야 하는데 여러번 시도를 했지만 완벽하게 끝내진 못했다.
- B. Thread_exit을 제대로 마무리하지 못했는데 이도 아마 test3를 통과하지 못한 이유와 비슷하게 exit할때 제대로 할당을 해제하지 못한 이유라 생각한다. Stack이 할당 해제되고 다시 할당되는 과정을 다시 추가해주어야 할 것 같다.