

WIKI

이름: 문준영

학과: 컴퓨터소프트웨어학부

학번: 2022006135

Design

Thread는 먼저 process와 동일한 취급을 해줘야 합니다. 서로 공유하는 부분이 있고 서로 독립적인 부분이 있습니다. 저는 때문에 Thread를 다음과 같이 구현했습니다.

Thread:

- Process 그 자체를 main thread로 둡니다.
- Proc 구조체에 int tid와 void *retval를 추가합니다.
- Tid는 thread_id이고 retval은 thread의 리턴값입니다.
- Thread 구조체를 사용하지 않고 Proc 구조체를 그대로 사용해 ptable로 thread를 관리하고 스케줄링해줍니다.
- Thread가 생성될 때 새로운 stack 공간을 할당 받습니다.
- Thread는 마스터 스레드와 pgdir을 공유합니다.

Thread_create

- Thread_create의 주체를 master thread로 바꾸어 thread가 master thread로 부터 정보를 받아올 수 있도록 합니다.
- allocproc 함수를 통해 새로운 스레드의 커널 스택을 할당 받습니다.
- fork 함수와 마찬가지로 master thread의 정보들을 가져옵니다.
 - 여기서 fork와의 차이점은 fork에서는 parent process의 pgdir을 copyuvvm해서 복사해 왔습니다. 하지만 Thread끼리는 서로 Pgdir을 공유해야 하기 때문에 master thread의 pgdir을 reference로 가져옵니다.
- Thread만의 새로운 stack 영역을 할당해줍니다.
- 새로운 스택 영역만큼 master thread의 sz도 증가시켜 줍니다.
 - 이는 마스터 스레드가 프로세스 내의 모든 스레드를 관리하는 주체라고 생

각하여, 마스터 스레드에서 계속 stack을 위로 증가하는 방향으로 쌓아가려는 것입니다. (자세한 설명은 구현단계에서 하겠습니다.)

- Thread가 fake return 값과 start_routine에 넣어줄 argument를 stack에 넣어줍니다
- eip 레지스터가 start_routine을 가리키게 하고 thread의 state를 바꿔 스케줄링의 대상이 되게 합니다.

Thread_exit

- 본래의 xv6에 존재하는 exit함수와 거의 동일하지만 retval를 받습니다.

Thread_join

- 본래의 xv6에 존재하는 wait함수와 거의 동일합니다.

Locking:

- Peterson's algorithm을 사용해서 atomic한 함수를 만듭니다. 수업시간에 배운 peterson's algorithm은 두 개의 프로세스에 대한 것이므로 이를 조금 더 확장해서 peterson's algorithm을 사용합니다.

Implementation Thread:

Proc.h

```
xv6-public 2 > C proc.h > 8.8 proc
38 ~ struct proc {
39     uint sz;                      // Size of process memory (byte)
40     uint base;                    // Base of process stack
41     pde_t* pgdir;                // Page table
42     char *kstack;                // Bottom of kernel stack for t
43     enum procstate state;        // Process state
44     int pid;                     // Process ID
45     struct proc *parent;         // Parent process
46     struct trapframe *tf;        // Trap frame for current sysca
47     struct context *context;     // swtch() here to run process
48     void *chan;                  // If non-zero, sleeping on cha
49     int killed;                 // If non-zero, have been kille
50     struct file *ofile[NFILE];   // Open files
51     struct inode *cwd;           // Current directory
52     char name[16];               // Process name (debugging)
53     int tid;                     // Thread ID (-1 for master thread
54     void *retval;                // Return value for thread
55 };
```

Proc 구조체 안에 int tid와 void *retval를 추가하였습니다.

Proc.c

thread_create 함수

```
int
thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg){
    struct proc *np;
    //Find master thread
    struct proc *curproc;
    uint ustack[2];
    struct proc *p;
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(myproc()>pid == p->pid && p->tid == -1){
            curproc = p;
            break;
        }
    }
    release(&ptable.lock);
```

Ptable을 돌면서 tid == -1 (master thread)를 찾아 thread_create에서 참조할 대상을 master thread로 만듭니다.

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

Allocproc을 하여 새로운 프로세스를 생성합니다. Allocpro에 인자를 받도록 합니다

```
// OTHERWISE RETURN 0.
static struct proc*
allocproc(int is_thread)
{
```

```
found:  
    p->state = EMBRYO;  
    if(!is_thread){  
        p->pid = nextpid++;  
        //Master thread  
        p->tid = -1;  
    }  
    else{  
        //Copy process's pid  
        struct proc *temp;  
        int max_tid = -1;  
        for(temp = ptable.proc; temp < &ptable.proc[NPROC]; temp++){  
            if(temp->pid == myproc()->pid){  
                if(temp->tid > max_tid){  
                    max_tid = temp->tid;  
                }  
            }  
        }  
        p->tid = max_tid + 1;  
        p->pid = myproc()->pid;  
    }  
}
```

allocproc 함수에서 is_thread라는 인자를 받게 합니다. 이 argument는 allocproc을 통해 만들 주체가 process인지 thread인지 알려줍니다. (0이면 포르세스, 1이면 스레드) 때문에 thread가 아니면 일반적인 process를 위한 pid작업을 수행합니다. 이에 더하여, 프로세스 자체를 마스터 스레드로 보기에 tid를 -1로 설정해줍니다 하지만 만약 스레드를 생성하는 거라면 그 프로세스 안에서 제일 큰 tid를 할당받습니다 또한 pid를 현재 스레드와 동일하게 가집니다.

다시, Thread_create로 돌아오겠습니다.

```
*thread = np->tid;

//Reference master_thread
np->pgdir = curproc->pgdir;
np->sz = curproc->sz;
np->base = curproc->sz;
*np->tf = *curproc->tf;
np->parent = curproc;
```

Argument로 들어온 thread에 tid값을 넣어줍니다. 그리고 생성한 스레드에 마스터 스레드로부터 정보를 복사/reference 해옵니다.

```
//Round up page for alignment
uint stack_bottom = PGROUNDUP(np->sz);
uint stack_sz = 2 * PGSIZE;
uint stack_top = stack_bottom + stack_sz;

//Allocate stack for new thread with a guard page
if((np->sz = allocuvm(np->pgdir, stack_bottom, stack_top)) == 0){
    freeproc(np);
    return -1;
}
//하나의 thread가 추가되어 master_thread의 size는 증가
curproc->sz += 2 * PGSIZE;

//Set up guard page
clearpteu(np->pgdir, (char *)(np->sz - 2*PGSIZE));
```

Thread만의 스택을 할당 받습니다. Stack의 크기는 총 $2 * \text{PGSIZE}$ 인데 그 중 한 페이지는 가드 페이지로 활용합니다. 때문에 Stack은 1 PGSIZE 만큼 할당받은 것과 동일합니다. Stack을 할당받고 master thread의 size 정보도 업데이트 해줍니다. 마스터 스레드가 결국 새로운 스레드를 할당할 때마다 size가 커지는 방향으로 갑니다. 이말인 즉슨, 새로운 스레드가 할당할 때마다 전체 스레드를 관리하는 master thread는 그 sz가 $2 * \text{PGSIZE}$ 씩

커진다는 얘기입니다.

```
np->tf->esp = np->sz - sizeof(ustack);
//fake return PC
ustack[0] = 0xffffffff;
ustack[1] = (uint)arg;

if(copyout(np->pgdir, np->tf->esp, ustack, sizeof(ustack)) < 0){
    freeproc(np);
    curproc->sz -= 2 * PGSIZE;
    return -1;
}

np->tf->eip = (uint)start_routine;
```

Thread의 stack에 start_routine의 argument와 fake return값을 넣어줍니다. 또한 eip(instruction pointer)가 start_routine을 가리켜 시작 함수를 start_routine으로 시작하게 합니다.

```
//copy process file
for(int i = 0; i < NOFILE; i++){
    if(curproc->ofile[i]){
        np->ofile[i] = fileup(curproc->ofile[i]);
    }
}

np->cwd = idup(curproc->cwd);
//copy process name
safestrcpy(np->name, curproc->name, sizeof(curproc->name));
acquire(&ptable.lock);
np->state = RUNNABLE;
release(&ptable.lock);

return 0;
}
```

Master thread의 파일과 이름을 복사합니다. 생성된 thread의 state를 RUNNABLE로 바꿔 스케줄링이 일어날 수 있도록 합니다.

Thread_exit 함수

Thread_exit함수는 원래의 xv6의 exit함수와 거의 동일하게 구현하였습니다. 하지만 exit 함수를 조금 변형했습니다. 먼저 thread_exit부터 살펴보도록 하겠습니다.

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

    if(curproc == initproc){
        panic("init exiting");
    }
    if(retval){
        curproc->retval = retval;
    }
}
```

현재 스레드가 initproc이면 패닉을 발생시킵니다. Retval가 존재하면 스레드의 retval에 넣어줍니다.

```

// 파일 종료
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;

```

열려있는 파일들을 닫아줍니다.

```

wakeup1(curproc->parent);

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc && !(curproc->tid == -1 && p->tid >= 0)){
        p->parent = initproc;
        if(p->state == ZOMBIE){
            wakeup1(initproc);
        }
    }
}

curproc->state = ZOMBIE;
sched();
panic("zombie exit");
}

```

Curproc->parent를 깨워줍니다. 여기서 중요한 것은 parent가 master thread라는 것입니다. Master thread가 join에서 대기하고 있을 것이기 때문에 master thread를 깨워 thread_exit을 호출하는 스레드를 정리해줄 수 있도록 합니다. 만약 master thread가 먼저 죽어있다면 initprocess에 고아 스레드/프로세스를 정리하도록 합니다.

Thread_exit은 스레드가 종료하면서 호출하는 함수입니다. 하지만 exit의 경우 다릅니다. Exit의 경우 하나의 스레드가 호출하면 전체 프로세스를 종료해야 합니다. 때문에 다음과

같이 변형하였습니다.

Exit 함수

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

    if(curproc->tid >= 0){
        curproc = curproc->parent;
    }
}
```

Exit의 초반부이자 제 구현에서 가장 핵심적인 부분입니다. 어떤 스레드가 exit를 호출하든 간에 master thread가 exit를 관리하는 주체가 되게 만듭니다.

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

//master thread 종료 전 모든 스레드 종료
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == curproc->pid && p->tid >= 0){
        freeproc2(p);
    }
}
release(&ptable.lock);
```

현재 종료하려는 프로세스가 Init process면 패닉을 발생시킵니다. 그리고 ptable을 돌면

서 종료하려는 Process의 마스터 스레드를 제외한 모든 스레드를 제거합니다.

Freeproc2 함수

```
void
freeproc2(struct proc *p){
    int fd;
    for(fd = 0; fd < NOFILE; fd++){
        if(p->ofile[fd]){
            fileclose(p->ofile[fd]);
            p->ofile[fd] = 0;
        }
    }
    if(p->pgdir){
        deallocuvm(p->pgdir, p->sz, p->base);
        p->pgdir = 0;
    }
    p->sz = 0;
    p->pid = 0;
    p->parent = 0;
    p->name[0] = 0;
    p->killed = 0;
    p->tid = 0;
    p->state = UNUSED;
}
```

Freeproc2 함수에서는 argument로 들어온 스레드를 할당 해제 시켜줍니다.

다시 원래의 exit 함수로 돌아와서 보면

```
//master thread 종료 전 모든 스레드 종료
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC];
    if(p->pid == curproc->pid && p->tid >= 0)
        freeproc2(p);
}
release(&ptable.lock);

// 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;

acquire(&ptable.lock);

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

마스터 스레드가 나머지 스레드를 모두 종료시키고 파일 시스템을 정리합니다. 그리고 마스터 스레드의 부모, 즉, 마스터 스레드를 fork한 주체를 깨웁니다. 그러면 마스터 스레드의 부모가 마스터 스레드를 wait() 함수에서 처리해줍니다.

```
// 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");
```

Thread_exit과 마찬가지로 parent가 zombie상태면 init process에서 마스터 스레드를 회수합니다.

Thread_join 함수

```
int
thread_join(thread_t thread, void **retval){
    acquire(&ptable.lock);
    struct proc *curproc = myproc();
    struct proc *p;
    int found = 0;

    for(;;){
        found = 0;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->pid == curproc->pid && p->tid == thread){
                found = 1;
                if(p->state == ZOMBIE){
                    if(retval){
                        *retval = p->retval;
                    }
                    freeproc(p);
                    release(&ptable.lock);
                    // 스레드가 종료되었고 이의 retval을 회수했으면 종료
                    return 0;
                }
            }
        }
        // 스레드를 찾지 못했거나 master thread가 kill 된 경우 종료
        if(!found || curproc->killed){
            release(&ptable.lock);
            return -1;
        }
        // 스레드가 아직 종료되지 않은것이므로 종료되기까지 대기
        sleep(curproc, &ptable.lock);
    }
}
```

Thread_join함수는 wait함수와 로직이 매우 유사합니다. Ptable을 돌면서 tid가 같은 것을 찾고 그 스레드를 할당 해제를 해줍니다. 만약 당장 그 스레드가 zombie 상태가 아니면 thread_join을 호출한 주체, 즉, 마스터 스레드는 스레드가 종료되기까지 sleep을 합니다. 이 sleep은 thread_exit에서 wakeup1을 통해 풀립니다.

여기서 저는 고민을 하던 것이 있습니다. Thread가 모두 동시에 종료하는 것이 아니기에 하나씩 종료하면 어떡할 건지에 대한 고민입니다. 예를 들어, master thread가 있고 이 마스터 스레드가 0번, 1번, 2번 스레드를 생성했다고 합시다. 그렇다면 thread_create에서 각각의 스레드에게 2 * PGSIZE만큼의 stack을 allocuvm을 통해 할당해 줄것입니다. 하지만

만약 0번이 종료하고, 나중에 1번이 종료하고, 더 나중에 2번이 종료하면 문제가 발생할 수 있습니다. 스택의 중간이 비어있는 현상이 발생합니다. 이 때 새로운 스레드를 마스터 스레드가 생성할 때 이는 필시 문제가 될 수 있습니다. 때문에 저는 thread_create에서 매번 마스터 스레드의 sz를 2 * PGSIZE만큼 키워줬습니다. 즉, 중간중간 스레드가 할당하든 말든 어차피 계속 새로운 페이지를 memory 공간에서 증가하는 방향으로 할당해 줄 것입니다.

전체 vm의 경우 프로세스가 종료될 때 wait에서 freevm이 되므로 이 프로세스가 실행될 동안은 memory utilization이 안좋아 질 수 있을 것입니다. 하지만 만약 중간중간 deallocate 된 메모리를 관리하기 위해서는 list를 만들거나 해야할 것이므로 구현이 너무 어려워 그냥 memory utilization을 조금 줄이더라도 할당을 계속 위쪽으로 해줬습니다.

Delete_threads 함수

```
void
delete_threads(struct proc *curproc){
    struct proc *p;
    struct proc *oldparent = curproc;
    acquire(&ptable.lock);
    if(curproc->tid != -1){
        oldparent = curproc->parent;
        curproc->parent = curproc->parent->parent;
        curproc->pgdir = copyuvm(curproc->parent->pgdir, curproc->parent->sz);
        curproc->sz = curproc->parent->sz;
        curproc->tid = -1;
    }
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == curproc->pid && p != curproc){
            if(p->tid == -1){
                p->killed = 1;
                continue;
            }
            freeproc(p);
        }
    }
    release(&ptable.lock);
}
```

이 함수는 exec에서 사용되는 함수입니다. Exec 시스템 콜을 thread가 호출하면 호출한 스레드를 제외한 모든 스레드는 종료되어야 합니다. 때문에 저는 호출한 스레드를 인자로 받아 이를 먼저 마스터 스레드로 만들어 주었습니다. Ptable을 돌면서 스레드를 종료 해주고 master thread인 경우 kill표시를 해주어 master thread의 부모 프로세스가 이를 회수 할 수 있도록 했습니다.

여기서 고민한 것은 처음 실행 될 때 init process나 shell process가 exec를 수행할 때 문제가 생기지 않을까 입니다. 하지만 어차피 이들은 그 자체로 마스터 스레드이기에 ptable을 돌면서 어떠한 것도 종료시키지 않을 것이기에 exec 호출 시에 아무런 문제가 발생하지 않습니다.

```
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();

    delete_threads(curproc);
```

Exec 함수의 초반부입니다. Exec.c에서 ptable.lock을 획득하지 못하기에 proc.c에서 delete_threads 함수를 만들어 호출해주는 방식을 활용했습니다.

]

Kill 함수

```
int
~kill(int pid)
{
    struct proc *p;
    int found = 0;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid && p->tid == -1){
            p->killed = 1;
            found = 1;
        }
        if(p->state == SLEEPING){
            p->state = RUNNABLE;
        }
        if(found){
            release(&ptable.lock);
            return 0;
        }
    }
    release(&ptable.lock);
    return -1;
}
```

Kill 함수의 변형은 상당히 간단했습니다. 왜냐하면 저는 어찌됐든 프로세스의 어떠한 스레드가 exit을 호출하더라도 그 전체 프로세스를 종료해버리게 exit을 수정해 놨기 때문입니다. 때문에 kill에서 pid가 같은 프로세스의 마스터 스레드를 찾고 killed변수를 1로 바꿔줍니다. 이는 trap.c에서 exit을 호출하게 됩니다.

```
void
trap(struct trapframe *tf)
{
    if(tf->trapno == T_SYSCALL){
        if(myproc()->killed)
            exit();
        myproc()->tf = tf;
        syscall();
        if(myproc()->killed)
            exit();
        return;
    }
}
```

Trap.c파일에 trap함수에서 현재 RUNNING중인 프로세스가 killed된 상태면 exit 함수를 호출하게 되어있습니다. 때문에 kill 함수에서 마스터 스레드를 kill함으로 그 프로세스 안에있는 모든 스레드를 종료할 수 있습니다.

Implementation Locking & Trouble Shooting of Locking

Locking을 구현하기 위해 먼저 software로 만들 수 있는 atomic operation을 생각해보았습니다. 이는 Peterson's algorithm이었습니다. 수업 때 다룬 것은 2개의 프로세스에 대한 peterson's algorithm이었기에 이를 확장해 n개의 Process에 대해 peterson's algorithm을 사용했습니다.

```

9 void lock(int);
10 void unlock(int);
11 int flag[NUM_THREADS];
12 int turn[NUM_THREADS];
13 int n = NUM_THREADS;
14
15 void lock(int curr)
16 {
17     flag[curr] = 1;
18     for(int i = 0; i < NUM_THREADS; i++){
19         if(i != curr){
20             //다른 놈한테 턴을 넘겨줌
21             turn[curr] = i;
22             //다른애가 들어갈 의지가 있고 순서가 상대 순서면 기다림
23             while(flag[i] && (turn[curr] == i));
24         }
25     }
26 }
27
28 void unlock(int curr)
29 {
30     flag[curr] = 0;
31 }

```

Lock에서 먼저 flag를 1로 바꾸며 현재 자신이 들어갈 마음이 있다는 것을 보입니다. 그리고 thread 개수만큼 for문을 돌며 턴을 다른 애한테 넘겨주고 만약 그 놈이 들어갈 의지도 있다면 busy waiting을 합니다. 하지만 이 피터슨 알고리즘에는 문제가 있습니다. 예)

- 세 개의 스레드 0, 1, 2가 있다고 가정합시다.
- Flag =[0,0,0], turn = [0,0,0] 으로 초기값이 설정될 것입니다.
- 스레드 0에서 lock(0)을 호출합니다.
- Flag[0] = 1
- For문을 통해 thread 1, 2 turn값을 조정합니다.
- 동시에 1이 확인할 때 turn[0] = 1로 설정하고 flag[1] && (turn[0] == 1)을 확인합니다

- 스레드 1이 critical section에 들어가고 싶으면 스레드 0은 대기합니다
- 동시에 스레드 1이 lock(1)호출합니다
- Flag[1] = 1
- For 문을 통해 다른 스레드의 turn값을 조정합니다
- 스레드 0과 스레드 1이 동시에 서로 대기중일 수 있습니다.

이와 같이 동시에 여러개의 스레드가 critical section에 접근하려고 할 때 서로가 서로를 기다리는 deadlock에 걸릴 수 있습니다. 실제로 실행해본 결과 확률적으로 올바른 답안이 나오고 확률적으로 무한 루프에 걸려 결과값이 출력되지 않았습니다.

고민을 하던 중 semaphore를 활용해볼까도 했는데 이 역시 결국 semaphore 변수 자체가 공유변수이기에 어차피 peterson's algorithm으로 보호해줘야 합니다. 하지만 위에서 보았듯이 peterson's algorithm으로 다중 스레드 상황 synchronization을 보장할 수 없습니다.

때문에 Software상으로 synchronization을 구현하기는 어렵다고 판단했습니다. 하드웨어 상으로 이를 구현하려고 하니 어셈블리어의 xchgl instruction을 활용하면 된다는 사실을 알게되었습니다. Xchg instruction은 하드웨어 상으로 atomic한 instruction이기에 이를 활용하여 하드웨어 상으로 atomic한 함수인 test_and_set을 구현했습니다.

```
test_and_set.s
1 .section .text
2 .global test_and_set
3 test_and_set:
4     movl $1, %eax
5     xchgl %eax, (%rdi)
6     ret
7
```

macOS와 x86 아키텍쳐에서 사용하는 어셈블리어가 다르기에 docker로 작성했습니다. 기본적이 test_and_set과 로직이 같습니다. 먼저 %rdi에 target의 주소가 들어옵니다. 먼저 return 값을 저장하는 %eax 레지스터에 true를 저장합니다. 그리고 원자적으로 target과 return_value를 swap합니다. Return value를 리턴합니다. 이를 통해 다음과 같이 구현했습니다.

```
9
10 void lock();
11 void unlock();
12 extern int test_and_set(int *lock);
13 int target = 0;
14
15 void lock()
16 {
17     while(test_and_set(&target) != 0);
18 }
19
20 void unlock()
21 {
22     target = 0;
23 }
24
```

이와 같이 하드웨어 상으로 구현하니 정상적으로 locking이 작동합니다.

```
gcc test_and_set.s pthread_lock_linux.c -lpthread; ./a.out
```

이와 같이 컴파일 하여 실행시켰습니다.

Result

```
$ 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 0 start
Child of thread 2 start
Child of thread 3 start
Child of thread 1 start
Child of thread 4 start
Child of thread 0 end
Child of thread 2 end
Thread 0 end
Thread 2 end
Child of thread 3 end
Child of thread 1 Child of thread 4 end
Thread 3 end
Thread 4 end
end
Thread 1 end
Test 2 passed

All tests passed!
```

Thread_test.c 파일 테스트 결과입니다. Trouble shooting에서 말씀드리겠지만 저는 sbrk는 수행하지 못했습니다. Thread_test.c 파일에서 test3는 현재 주석처리 되어 있습니다. Basic test와 fork test까지 통과한 모습입니다.

```
$ thread_kill
Thread kill test start
Killing process 4
This code should be executed 5 times.
Kill test finished
```

Thread_kill.c 파일 실행 결과입니다

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

Thread_exit.c 수행결과 입니다

```
[root@13584cc87a11:~/OS/p3_test]$ ./thread_exec
Thread exec test start
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Executing...
Hello, thread!
$
```

Thread_exec.c 수행 결과입니다.

```
[root@13584cc87a11:~/OS/p3_test# vi pthread_lock_linux.c
[root@13584cc87a11:/OS/p3_test# gcc test_and_set.s pthread_lock_linux.c -lpthread; ./a.out
shared: 100
[root@13584cc87a11:/OS/p3_test# gcc test_and_set.s pthread_lock_linux.c -lpthread; ./a.out
shared: 1000
[root@13584cc87a11:/OS/p3_test# gcc test_and_set.s pthread_lock_linux.c -lpthread; ./a.out
shared: 1050
```

Pthread_lock_linux.c 수행 결과입니다. 맨 위는 thread의 개수가 10개이고 iteration 수가 10번인 기본 테스트 케이스입니다. 두 번째 사진은 thread의 개수가 100개이고 iteration 수가 10번인 테스트 케이스입니다. 세 번째 사진은 thread의 개수가 10개 iteration 수가 105번인 테스트 케이스입니다.

Trouble Shooting for Thread (Locking의 trouble shooting은 Implementation 부분에서 설명함)

Thread를 구현하면서 여러 가지 문제를 겪었습니다.

먼저, exit에서 freeproc 호출 시입니다. 저는 freeproc 함수와 freeproc2 함수를 만들었습니다. 이 두 함수의 차이는 하나는 freeproc은 kfree를 해준다는 것이고 freeproc2는 kfree

를 안해준다는 것입니다. Thread_join 할 때는 freeproc을 호출해서 thread를 완전히 정리했습니다. 하지만 exit에서 freeproc을 호출하면 page fault 가 발생했습니다. 이를 해결하기 위해 여러가지 시도를 하던 중 kfree를 안하면 문제가 발생하지 않는다는 것을 깨달았습니다. Exit에서만 이러한 문제가 생겼던 것이므로 exit에서만 freeproc2를 만들어서 kfree 부분을 하지 않는 형태로 함수를 만들어 마스터 스레드가 나머지 스레드를 정리하도록 하였습니다. 사실 아직도 왜 kfree가 문제인지는 잘 모르겠습니다.

다음으로 발생한 문제는 thread에서 fork시 발생하는 copyvm page not present 패닉 문제였습니다. 이는 제 설계상 발생한 문제였습니다. 예를 들어, 어떤 스레드가 fork를 호출하면 fork에서 copyvm을 호출합니다.

```
// Copy process state from proc.  
// 부모 프로세스와 독립적인 페이지 테이블을 사용해야 하므로 페이지 테이블을 복사  
if((np->pgdir = copyuvvm(curproc->pgdir, curproc->sz)) == 0){  
    kfree(np->kstack);  
    np->kstack = 0;  
    np->state = UNUSED;  
    return -1;  
}
```

Copyuvvm에 들어가보면 페이지 디렉토리를 전부 훑으면서 만약 존재하지 않는 page면 패닉을 발생시키도록 하였습니다.

```
for(i = 0; i < sz; i += PGSIZE){  
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)  
        panic("copyuvvm: pte should exist");  
    if(!(pte & PTE_P))  
        panic("copyuvvm: page not present");
```

하지만 제 구현에서 위에서 말씀드렸듯이 마스터 스레드 중간중간에 비어있는 공간이 있을 수 있습니다. 이를 넘어가지 못하게 가드 페이지가 막고 있으니 크게 문제 되지는 않습니다. 때문에 저는 fork시 존재하지 않는 페이지라도 그냥 무시하도록 하였습니다.

```
for(i = 0; i < sz; i += PGSIZE){  
    if((pte = walkpgdir(pgd, (void *) i, 0)) == 0)  
        panic("copyuvm: pte should exist");  
    if(!(pte & PTE_P))  
        continue;  
    // panic("copyuvm: page not present");
```

이와 같이 무시하도록 하였습니다. 이러니 정상적으로 fork가 되고 모든 테스트가 통과되었습니다.

다음으로, exec함수를 수정하는 과정이었습니다. 이것이 너무 어려웠던 것이 저는 모든 것의 주체가 마스터 스레드이기 때문입니다. 하지만 exec의 경우 마스터 스레드가 아닌 일반 스레드가 호출 할 수도 있을 것입니다. 그렇다면 그 스레드를 제외한 모든 스레드, 즉, 마스터 스레드 역시 제거해야 합니다. 하지만 그렇게 하니 정말 여러가지 문제가 생기더군요. 때문에 아예 제거 방식을 바꿨습니다. Exec를 호출하는 현재 스레드를 마스터 스레드로 바꾸어 나머지 스레드를 정리했습니다. 이와 같이 구현하니 문제가 해결되었습니다.

다음으로, exec 수행한 후에 발생하는 문제입니다. Exec 발생한 후에 \$ 표시가 엔터를 한번 더 눌러야 나옵니다. 이게 처음 thread_exec파일을 실행할 때는 안 그러는데 추가적으로 더욱 무언가를 실행하면 그 프로세스가 정상시행되고 나서도 \$ 표시가 엔터 한 번을 쳐야 나옵니다. 이를 해결하지 못했습니다.

마지막으로는 해결하지 못한 문제입니다. Thread_test3를 수행하려고 보니 sbrk를 수정해야 했습니다. 하지만 sbrk의 경우 page를 서로 공유를 해야 합니다. 저는 이를 구현하기 위해 각 스레드의 pgdir를 관리해야 한다는 것을 깨달았고 그렇다면 제가 구현한 것과 달리 중간중간 비어 있는 공간 역시 관리되어야 한다는 것을 깨달았습니다. 너무나 복잡해질 것 같아 솔직히 포기했습니다. 죄송합니다.

마지막으로 하고 싶은 말:

이번 과제를 하면서 솔직히 정말 많이 후회스러웠습니다. 저는 이번 과제를 수행하기 위해 정말 다양한 파일들, 다양한 함수들을 뜯어보고 분석해 보면서 xv6의 전체적인 메모

리 구조에 대해 이해하려고 노력했습니다 때문에 제가 구현한 대부분의 것들은 모두 제 이해 안에서 나온 것입니다. 정말 많은 시간을 들였고 노력을 쏟았습니다. 하지만 대부분 이번 과제를 fork를 배끼고 exec 배끼고 git을 참고하는 등 빠르게 과제를 끝냈습니다. 대부분 제대로 이해를 하지 않은 상태로 코드를 작성한 것입니다. 그러면서 저보다 높은 점수를 받을 생각을 하니 정말 마음이 아프네요. 조교님 그냥 이건 제 푸념입니다. 이해 해주시면 감사하겠습니다...