

Project03 Wiki

2020081958 송재휘

1) LWP Wiki

1. Design

우선 스레드를 구현하기 위해 기존의 proc 구조체에 해당 프로세스가 스레드인지 여부를 나타내는 `int isThread`, 해당 스레드의 `tid`를 나타내기 위한 `int tid`, 만약 스레드를 포함하는 상위 프로세스인 경우 해당 프로세스가 가지고 있는 스레드의 수를 저장하기 위한 `int numThread`, 나중에 return value를 `thread_join`에 넘겨줄 때 사용할 `void *retval`, 해당 스레드가 속한 프로세스를 저장하기 위한 `struct proc *master`의 특성을 추가해주었다. 이후 project03에서 구현해야 할 3개의 system call인 `thread_create`, `thread_exit`, `thread_join`을 먼저 구현할 것이다. 시스템콜 구현을 마치고, 이미 존재하는 다른 여러 시스템콜에 대한 대처를 할 수 있도록 코드를 수정할 계획이다. 이를 위해, 우선 새로운 프로세스를 할당해주는 `allocproc` 함수를 수정할 것이다. `Allocproc` 함수에서 새로운 프로세스를 만들 때 앞서 추가한 특성들의 값을 초기화 해줄 것이다. 이후, `thread_create`에서는 xv6에 존재하는 `fork`와 `exec` 시스템콜을 참고해 만들 계획이다. 전반적인 계획은 Fork에서 수행하는 `allocproc`을 우선 진행하고, `exec`에서 수행하는 페이지 할당을 해주고, 스레드를 생성한 프로세스와 페이지 테이블을 공유하도록 할 것이다. 이렇게 함으로써 스레드는 상위 프로세스와 주소공간을 공유하게 된다. 이후, 상위 프로세스의 `pid`, 페이지 테이블, 페이지 사이즈, `trapframe`, `parent`를 공유하도록 설정하고, 앞서 추가한 `isThread`를 1로, 스레드의 `master`를 상위 프로세스로, 또 `thread_create`의 인자로 받은 `thread_t *thread`에 현재 스레드의 `tid` 값을 넣어줄 것이다. 이때 `tid`는 앞서 추가한 상위 프로세스의 `numThread`값을 사용해 정하고, `tid`를 특정 스레드에 부여한 이후에는 `numThread` 값을 1 올려준다. 추가적으로, 스레드 생성 시, 실행할 `start_routine`을 인자로 넘겨주기 때문에 `exec` 시스템콜을 참고해 `ustack array`를 생성하고, `thread_create`의 마지막 인자로 넘겨받은 `void *arg`를 `ustack`을 통해 전달한다. 이후, 생성된 스레드의 `eip`에 `start_routine`을, `esp`에 앞서 언급한 `ustack`을 저장한 이후의 `stack pointer`값을 저장해준다. 이후 파일 디스크립터를 할당한다. 이후 상위 프로세스의 이름을 스레드에 똑같이 저장해주고, 상위 프로세스의 페이지를 로드한 후, 스레드의 상태를 `RUNNABLE`로 바꾸어 줄 계획이다. 다음으로 `thread_exit` 시스템콜은 기존의 `exit` 시스템콜을 참고해 만들었다. 우선 `thread_exit` 시스템콜을 실행하는 것이 스레드인지를 확인한다. 만약 아니라면, `panic`을 발생시킨다. 이후 현재 스레드로 인해 열려 있는 모든 파일 디스크립터를 닫고, 현재 작업 디렉터리를 해제한다. 이후 `ptable`을 돌면서 해당 스레드를 `parent`로 하는 프로세스 혹은 스레드가 있는 경우 `parent`를 초기 프로세스인 `initproc`으로 바꿔준다. 만약 해당

프로세스 혹은 스레드가 ZOMBIE 상태이면 `initproc`을 깨워준다. 이후 인자로 받은 `retval`을 해당 스레드의 `retval`에 저장해주고, 해당 스레드가 속한 프로세스의 `numThread`를 1 감소시킬 것이다. 이후 만약 해당 스레드가 상위 프로세스의 마지막 스레드인 경우 상위 프로세스를 깨우고, 스레드 상태를 ZOMBIE로 만든 후 `sched`를 통해 다른 프로세스로 스케줄링을 넘길 것이다. 이때 `sched`함수로 넘기기 위해 `phtable`에 대한 lock을 획득한 상태를 유지할 것이다. 마지막으로 `thread_join`에서는 `wait` 시스템콜을 참고해 만들 것이다. Lock 획득 후 `phtable`을 돌며 해당 프로세스 하위의 스레드가 끝나길 기다린다. 이후 끝나게 되면 `return value`를 해당 시스템콜의 인자인 `void **retval`에 저장하고, 해당 스레드의 커널 스택을 해제시켜줄 것이다. 이후 해당 스레드의 특성 값을 0으로 초기화해주고, `state` 또한 `UNUSED`로 변경해줄 것이다. 만약 끝난 스레드가 없다면, 상위 프로세스는 `sleep`하게 된다. 이후 앞서 구현한 3가지 시스템콜을 사용하기 위해 차례로 `defs.h`, `syscall.h`, `syscall.c`, `user.h`, `usys.S`에 등록할 것이다. 추가적으로 `thread_t`라는 데이터 타입을 여러 파일에서 사용할 것임으로, `types.h`에 `typedef int thread_t`를 추가해 줄 것이다. 이제 남은 건 이미 존재하는 시스템콜에 대처하는 것이다. Project03 pdf파일에 언급된 것들을 중심으로 살펴보겠다. 변경이 필요한 system call을 위주로 얘기하겠다. 우선 `exec`의 경우, 해당 시스템콜 실행 시 기존 프로세스의 스레드를 모두 정리하고, 그 중 하나의 스레드에서 새로운 프로세스를 시작해야 하기 때문에 `exec.c` 파일을 수정해 줄 것이다. 여기서도 추가해야 할 것은 해당 프로세스의 스레드를 정리해주는 코드이기 때문에 `proc.c`에 `thread_remove(int pid)`라는 함수를 하나 정의할 것이다. 해당 함수를 통해 특정 `pid`를 가지는 process 혹은 스레드 중 현재 `curproc`만을 남기고 모두 정리하고 자원을 회수할 것이다. 해당 함수의 구현은 이후 구현 파트에서 설명하겠다. 이후, 남은 `curproc`을 대표 프로세스로 설정한 뒤, 해당 프로세스의 페이지 테이블과 크기, `eip`, `esp`를 새로운 내용으로 갈아 끼고, 이전의 `old` 페이지 테이블을 삭제해줄 것이다. 그 다음은 `sbrk` 시스템콜인데 `sbrk`에 의해 할당된 메모리는 프로세스 내의 모든 스레드가 공유해야 한다. 따라서 상위 프로세스의 페이지 사이즈를 `addr`로 활용한다. `Sbrk`에서는 `growproc`이라는 함수를 실행하는데, 해당 함수에서 스레드인 경우 혹은 상위 프로세스인 경우로 나눠 페이지의 추가적인 할당 혹은 축소를 인자인 `n`의 부호에 맞게 실행해 줄 것이다. 이때 메모리에 할당하는 공간이 서로 겹치면 안되기 때문에 `sbrk`를 위한 `sbrklock`을 따로 정의해 할당 과정에서 사용할 것이다. 그 다음은 `kill` 시스템콜인데, 해당 시스템콜에서는 인자로 받은 `pid`의 프로세스의 `killed`를 1로 변경한다. 이때 `sleep` 상태에서도 `kill`에 의해 종료될 수 있다. `Kill` 자체의 코드는 따로 수정하지 않고, 대신 `exit`의 코드를 수정할 것이다. `Kill`에서 하나의 스레드만 `kill` 되더라도 그 스레드가 속한 프로세스 내의 모든 스레드가 정리 돼야 하기 때문에 여기서도 `thread_remove`함수를 통해 같은 `pid`를 가진 스레드를 정리하고, 자원을 회수할 것이다. `Exec`과 같이 남은 `curproc`을 상위 프로세스로 설정하고, Parent 프로세스를 깨울 것이다. 이후 `phtable`을 돌면서 `parent`를 `initproc`으로 변경

해주는 과정은 동일하다. 이후 curproc의 상태를 ZOMBIE로 바꾸고 똑같이 ptable lock을 유지한채로 sched함수를 실행 할 것이다.

2. Implementation

1) proc.h

proc 구조체에 스레드 구현을 위해 isThread, tid, numThread, *retval, *master를 추가해 준다. 각 특성의 설명은 위의 design 파트에 적어두었다.

```
// Per-process state
struct proc {
    uint sz;                // Size of process memory (bytes)
    pde_t* pgdir;           // Page table
    char *kstack;           // Bottom of kernel stack for this process
    enum procstate state;   // Process state
    int pid;                // Process ID
    struct proc *parent;    // Parent process
    struct trapframe *tf;   // Trap frame for current syscall
    struct context *context; // swtch() here to run process
    void *chan;             // If non-zero, sleeping on chan
    int killed;             // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;      // Current directory
    char name[16];          // Process name (debugging)
    int isThread;           // if thread, value = 1 else 0
    int tid;                // thread id
    int numThread;          // # of threads for a master process
    void *retval;           // return value of a thread
    struct proc *master;    // master process of a thread
};
```

2) proc.c

Allocproc의 found 아래부분에 스레드를 위한 특성을 초기화해준다.(디폴트==프로세스)

```
// initialize added attributes for thread
p->isThread = 0;
p->tid = -1;
p->numThread = 0;
p->master = p;
```

Thread_create 부분은 앞의 design 파트에서 설명한대로 fork와 exec 시스템콜을 참고해 만들었다. 우선 allocproc을 통해 할당을 한 후, 페이지 할당을 해준다. 이후, thread에 맞게 특성값을 교체해준다. 특히 isThread = 1로 설정해 thread임을 명시하고, pid, sz

pgdir, tf를 공유해 주소공간을 공유하며, 해당 프로세스 하위의 스레드임을 나타낸다. (master라는 특성은 직접 해당 스레드가 속한 프로세스를 가리킨다.) tid를 프로세스 하위의 스레드 개수로 결정한다. 이후 인자의 *thread에 생성된 스레드의 tid를 저장한다. 이후 start_routine 설정을 위해 ustack을 정의하고 stack pointer가 가리키는 주소에 차례로 저장한다. 이후 변경된 sp와 start_routine을 각각 해당 스레드의 trapframe의 esp와 eip에 저장해준다.(이렇게 함으로써 스레드는 해당 start_routine에서 시작하게 됨 exec과 유사) 이후엔 fork의 코드로 파일 디스크립터를 할당한 후 프로세스의 name을 copy하고, 프로세스의 페이지 테이블을 로드한다. 이후, 준비가 다 된 스레드를 RUNNABLE 상태로 돌리면서 thread_create는 끝이 난다.

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

    // Allocate process.
    if ((np = allocproc()) == 0) {
        return -1;
    }
    acquire(&sptable.lock);
    curproc->sz = PGROUNDUP(curproc->sz);
    if ((curproc->sz = allocuvm(curproc->pgdir, curproc->sz, curproc->sz + 2 * PGSIZE)) == 0) {
        return -1;
    }
    sp = curproc->sz;
    // update proc attributes for thread
    np->isThread = 1;
    np->pid = curproc->pid; // thread pid == master process pid
    np->tid = curproc->numThread++;
    np->master = curproc;
    np->pgdir = curproc->pgdir; // thread share page table
    np->sz = curproc->sz; // size should be equal
    np->parent = curproc->parent; // parent process should be equal
    *np->tf = *curproc->tf;
    *thread = np->tid; // save tid value to *thread
    release(&sptable.lock);

    // save arg for start routine
    ustack[0] = 0xffffffff;
    ustack[1] = (uint)arg;
    sp -= 8;

    if (copyout(curproc->pgdir, sp, ustack, 8) < 0)
        return -1;

    // change trapframe
    np->tf->eax = 0;
    np->tf->eip = (uint)start_routine; // set eip with start_routine
    np->tf->esp = sp;

    for (int 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));

    switchuvm(curproc);
    acquire(&sptable.lock);
    np->state = RUNNABLE;
    release(&sptable.lock);
}
```

Thread_exit 시스템콜의 경우 exit 시스템콜과 유사하다. 우선 thread_exit을 호출한 것이 스레드인지 판단하고, 아니라면 panic을 발생시킨다. 이후 해당 스레드로 인해 열린 파일을 모두 닫고, 현재 작업 디렉토리를 해제한다. 이후 ptable 접근을 위해 ptable lock을 획득하고, ptable을 돌며 만약 특정 프로세스 혹은 스레드의 parent가 현재 스레드인지 확인한다. 만약 그렇다면 parent를 초기 프로세스인 Initproc으로 바꿔준다. 이후 인자로 받은 retval 값을 스레드의 retval이라는 특성에 저장해주고, 해당 스레드가 속한 프로세스의 총 스레드 개수를 1 감소시킨다. 이후 만약 해당 프로세스가 해당 프로세스의 마지막 스레드라면, 상위 프로세스를 깨워준다. 이후 현재 스레드 상태를 ZOMBIE로 바꾸고, 다른 프로세스 혹은 스레드의 스케줄링을 위해 sched를 호출한다.(이때 ptable lock 유지)

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

    // if curproc is not thread
    if(!curproc->isThread)
        panic("not thread error");

    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();
    input(curproc->cwd);
    end_op();
    curproc->cwd = 0;

    acquire(&ptable.lock);

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

    // set return value of thread
    curproc->retval = retval;
    // decrement total # of threads of a process by 1
    curproc->master->numThread--;
    //master might be sleeping in thread_join()
    if (curproc->master->numThread == 0)
        wakeup1(curproc->master);
    curproc->state = ZOMBIE;
    sched();
    panic("zombie exit");
}
```

Thread_join에서는 인자로 전달된 thread라는 tid를 가진 스레드가 종료되길 기다리고, thread_exit를 통해 반환된 값을 void **retval에 저장한다. 이를 위해 ptable을 돌면서 현재 프로세스 혹은 스레드와 동일한 프로세스에 속해있으면서 인자의 thread와 일치하는 tid를 가진 스레드의 종료 여부를 확인한다. 종료된 경우 해당 스레드의 커널 스택을 해제하고, 스레드를 정리하기 위해 특성을 초기화해준다. 만약 해당 스레드가 끝나지 않았다면 curproc은 잠깐 sleep을 하며 기다리게 된다.

```
int thread_join(thread_t thread, void **retval)
{
    struct proc *p;
    struct proc *curproc = myproc();

    acquire(&ptable.lock);
    for(;;){
        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            // check threads that belongs to curproc and tid of them
            if (p->tid != thread || p->master != curproc->master)
                continue;
            if(p->state == UNUSED){
                release(&ptable.lock);
                return 0;
            }
            if (p->state == ZOMBIE) {
                *retval = p->retval;
                kfree(p->kstack);
                p->kstack = 0;
                p->pid = 0;
                p->parent = 0;
                p->name[0] = 0;
                p->killed = 0;
                p->state = UNUSED;
                release(&ptable.lock);
                return 0;
            }
        }
    }

    // Wait for children threads to exit. (Wakeup1 call in thread_exit.)
    sleep(curproc, &ptable.lock);
}
}
```

Thread_t 데이터 타입을 xv6의 파일 내에서 사용할 수 있도록 types.h에 정의했다.

```
typedef int thread_t;
```

위 시스템콜에 대한 wrapper function은 sysproc.c에 정의하였고, 다음과 같다. 포인터가 붙은 인자들은 argptr을 활용해 인자를 받았고, 포인터가 붙지 않은 thread_t의 경우 argint를 사용해 인자를 받았다. 처음에는 &를 사용해 argptr의 인자를 받으려고 했지만 타입 에러가 발생해 (char**)를 사용해 argptr의 인수 타입을 맞춰 주었다.

```
// wrapper function for thread_create
int
sys_thread_create(void)
{
    thread_t *thread;
    void *(*start_routine)(void *);
    void *arg;

    if(argptr(0, (char **)&thread, sizeof(thread)) < 0)
        return -1;
    if(argptr(1, (char **)&start_routine, sizeof(start_routine)) < 0)
        return -1;
    if(argptr(2, (char **)&arg, sizeof(arg)) < 0)
        return -1;

    return thread_create(thread, start_routine, arg);
}
```

```
// wrapper function for thread_exit
int
sys_thread_exit(void)
{
    void *retval;
    if(argptr(0, (char **)&retval, sizeof(retval)) < 0)
        return -1;

    thread_exit(retval);
    return 0;
}
```

```
// wrapper function for thread_join
int
sys_thread_join(void)
{
    thread_t thread;
    void **retval;

    if(argint(0, &thread) < 0)
        return -1;
    if(argptr(1, (char **)&retval, sizeof(retval)) < 0)
        return -1;

    return thread_join(thread, retval);
}
```

앞의 3개의 시스템콜은 차례로 defs.h, syscall.h, syscall.c, user.h, usys.S에 등록 과정을 거치게 된다. 이에 대한 코드는 다음과 같다. (thread_remove 또한 proc.c에 추가로 정의한 함수이며 이에 대해서는 아래에서 설명하겠다.)

defs.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);
void     thread_remove(int pid);
```

syscall.h

```
#define SYS_thread_create 24
#define SYS_thread_exit 25
#define SYS_thread_join 26
```

syscall.c

```
extern int sys_thread_create(void);
extern int sys_thread_exit(void);
extern int sys_thread_join(void);
```

```
[SYS_thread_create] sys_thread_create,
[SYS_thread_exit]   sys_thread_exit,
[SYS_thread_join]   sys_thread_join,
```

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);
```

usys.S

```
SYSCALL(thread_create)
SYSCALL(thread_exit)
SYSCALL(thread_join)
```


이 함수는 exec과 exit에서 특정 프로세스에 속한 스레드를 정리해주기 위해 만들었다. 특정 프로세스의 pid를 인자로 넣으면 ptable lock을 획득하고 ptable을 돌며 해당 함수를 실행하는 curproc을 제외한 해당 pid의 다른 스레드로 인해 열려 있는 모든 파일 디스크립터를 닫고, 현재 작업 디렉터리를 해제한다. 이후 해당 스레드를 정리하기 위해 커널 스택을 해제하고, 특성을 0으로 초기화해준다. State를 UNUSED로 바꿔줌으로써 해당 스레드는 없어지게 된다. 이후 ptable lock을 내려놓고 함수가 끝난다.

```
void thread_remove(int pid) {
    struct proc *curproc = myproc();
    struct proc *p;

    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid && p != curproc) {
            int fd;
            for (fd = 0; fd < NOFILE; fd++)
            {
                if (p->ofile[fd])
                {
                    fileclose(p->ofile[fd]);
                    p->ofile[fd] = 0;
                }
            }
            release(&ptable.lock);

            begin_op();
            input(p->cwd);
            end_op();
            p->cwd = 0;

            acquire(&ptable.lock);
            kfree(p->kstack);
            p->kstack = 0;
            p->pid = 0;
            p->parent = 0;
            p->name[0] = 0;
            p->killed = 0;
            p->tid = -1;
            p->isThread = 0;
            p->master = 0;
            p->state = UNUSED;
        }
    }
    release(&ptable.lock);
}
```

위 코드는 exec.c에 추가한 코드로, exec 호출 시 현재 exec을 실행한 프로세스 혹은 스레드만을 남기고, 해당 pid를 가진 다른 모든 스레드를 정리한다. 위의 기능은 앞서 구현한 thread_remove 함수를 통해 구현했다. 이후 현재 스레드를 상위 프로세스로 설정하고, 이에 맞게 특성값을 변경해준다. 이후 해당 스레드에서 새로운 프로세스가 시작된다.

```
// terminate other threads of a given process
thread_remove(curproc→pid);
curproc→master = curproc;
curproc→isThread = 0;
curproc→numThread = 0;
curproc→tid = -1;
```

이후 sbrk 시스템 콜에 대응하기 위해 여러 스레드가 메모리 할당을 요청하더라도 겹치지 않게 하기 위해 sbrklock을 proc.c파일에 선언해주었다.

```
struct spinlock sbrklock;
```

또한 sbrk에 의해 할당된 메모리는 프로세스 내의 모든 스레드가 공유 가능해야 하기 때문에 sysproc.c에서 sbrk의 wrapper function의 addr 값을 상위 프로세스의 사이즈로 설정해주었다.

```
int
sys_sbrk(void)
{
    int addr;
    int n;

    if(argint(0, &n) < 0)
        return -1;
    addr = myproc()→master→sz;
    if(growproc(n) < 0)
        return -1;
    return addr;
}
```

Sbrk 시스템콜에서는 메모리 변동 사이즈 n을 인자로 받고, growproc(n)을 실행한다..

이때 growproc에서는 직접적인 메모리 할당을 수행하는데. 이때 curproc이 스레드인 경우와 상위 프로세스인 경우 두가지로 구분해 처리를 해주었다. 스레드라면 상위 프로세

스의 페이지와 사이즈 값을 활용해 n 이 0보다 클 때와 작을 때를 나누어 추가적인 할당 혹은 축소를 수행했다. 반대로 `curproc`이 상위 프로세스인 경우 현재 프로세스의 페이지와 사이즈를 활용해 같은 방식으로 할당 및 축소를 수행했다. 이후 이렇게 바뀐 메모리 크기의 공간으로 로드한 후 해당 함수가 종료된다. 이때 이 과정 전후로 앞서 선언한 `sbrklock`을 활용해 메모리 할당 공간이 겹치는 것을 방지하도록 설계했다.

```
// Grow current process's memory by n bytes.
// Return 0 on success, -1 on failure.
int
growproc(int n)
{
    uint sz;
    struct proc *curproc = myproc();

    // acquire sbrk lock
    acquire(&sbrklock);

    // if curproc is thread
    if(curproc->isThread){
        sz = curproc->master->sz;
        if(n > 0){
            if((sz = allocuvm(curproc->master->pgdir, sz, sz + n)) == 0)
                return -1;
        } else if(n < 0){
            if((sz = deallocuvm(curproc->master->pgdir, sz, sz + n)) == 0)
                return -1;
        }
        curproc->master->sz = sz;
    }
    // if curproc is not a thread
    else {
        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;
    }

    release(&sbrklock);
    switchuvm(curproc);
    return 0;
}
```

마지막으로 수정을 한 부분은 exit인데 kill이라는 시스템 콜 호출 시 하나의 스레드가 kill이 되더라도 그 스레드가 속한 프로세스 내의 모든 스레드가 정리 되고, 자원이 회수 되어야 한다. Kill에서는 해당 Pid의 프로세스 혹은 스레드의 killed를 1로 수정하는데, 결국 trap.c에서 killed가 1인 스레드의 경우 exit() 시스템콜을 호출하게 된다. 이를 처리하기 위해 앞서 exec에서와 같이 thread_remove를 통해 curproc을 제외한 해당 프로세스의 다른 모든 스레드를 종료하도록 설계하였다 이후, 남은 curproc을 상위 프로세스로 설정 하고, 그에 맞게 특성 값을 변경해준다. 이후 동일하게 ptable lock을 획득한 뒤 parent 프로세스를 깨우고, 바뀌어야 한다면 initproc으로 parent를 변경해준 뒤 state를 ZOMBIE로 바꾸고, sched()를 통해 스케줄링을 진행한다. (해당 부분의 코드는 앞서 설명한 thread_exit과 동일하다)

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

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

    thread_remove(curproc->pid);
    curproc->master = curproc;
    curproc->isThread = 0;
    curproc->numThread = 0;
    curproc->tid = -1;

    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");
}
```

여기까지 구현하게 되면 존재하는 xv6의 다른 시스템콜과 함께 스레드를 효과적으로 사용할 수 있게 된다.

3. Result.

Project03 Test를 진행하기 위해 올려주신 p3_test 내의 파일들을 차례로 추가해주었고, Makefile의 UPROGS와 Extra부분에 차례로 등록을 해주었다. 컴파일 및 실행을 위해 이전 project와 동일하게 make -> make fs.img -> ./bootxv6 명령어를 사용했다. 이후 실행 결과는 다음과 같다. (Project03-Test에서 주어진 모든 test case를 통과했다.)

1) thread_test.c

Test1의 경우 스레드 API의 기본적인 기능(create, exit, join)과 스레드 사이에서 메모리가 자 공유되는지를 평가하는 테스트이다. 해당 테스트에서 요구하는 것처럼 스레드 0은 곧바로 종료되고, 스레드1은 2초 후 종료되는 걸 확인해볼 수 있었다. 이 과정에서 메인 스레드에서 join을 하며 기다리게 되고, 두 스레드 종료 후 메인 스레드에서 하고있던 join이 종료되며 성공적으로 테스트가 끝나는 것을 확인할 수 있었다.

```
$ 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
```

Test2의 경우 스레드 내에서 fork를 수행하는 테스트이다. 이때 자식 프로세스는 부모 프로세스와 분리된 주소 공간에서 동작해야 한다. 테스트2에서 역시 스레드 내에서 fork를 했을 때에도 오류 없이 주소 공간이 분리되어 실행되는 것을 확인할 수 있었다.

```
Test 2: Fork test
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Child of thread Child of thread 1 start
Child of thread 3 start
Child of thre0 start
ad 4 start
Child of thread 2 start
Child of thread 1 end
Child of thread 3 end
Thread 3 end
Child of thread 0 end
Child of thread 4 eChild of thread 2 end
Thread 0 end
Thread 1 end
Thread 2 ennd
d
Thread 4 end
Test 2 passed
```

Test3의 경우 스레드에서 sbrk가 올바르게 동작하는지 평가하는 테스트이다. 이때 thread_test.c에서 malloc을 사용하는데, 이 과정에서 내부적으로 sbrk가 호출된다. 이때 할당된 메모리를 특정 프로세스 하위의 스레드가 모두 공유를 해야 하며, 연속된 메모리 할당 요청을 받아도 겹치지 않도록 처리해주어야 한다. 앞서 살펴본 sbrk관련된 부분에서 sbrklock을 활용하여 상위 프로세스 기준으로 sbrk를 처리했기 때문에 이번 테스트에서도 꼬이는 문제 없이 메모리가 할당되고 해제되는 것을 확인할 수 있었다.

```
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!
$ █
```

2) thread_exec.c

이번 테스트는 스레드에서 exec 시스템콜을 호출했을 때 올바르게 동작하는지 평가하는 것이다. 스레드에서 exec을 실행하는 순간 해당 프로세스 하위의 모든 스레드를 종료하고, exec을 실행한 스레드만을 남겨 해당 스레드에서 새로운 프로세스가 실행되어야 한다. 해당 테스트에서는 생성되는 스레드 중 하나가 exec으로 hello_thread 프로그램을 실행하게 되고, 이때 다른 스레드는 정리된다. 이후 해당 스레드에서 Hello, thread! 문구를 띄우고, 테스트는 종료된다. 이 테스트를 통해 스레드 내에서 exec을 수행했을 때 해당 프로세스 하위의 모든 스레드가 성공적으로 정리되는 것을 확인할 수 있었다.

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

3) thread_exit.c

주어진 명세에서는 하나 이상의 스레드에서 `exit` 호출 시 해당 프로세스의 모든 스레드가 종료 된다고 명시하고 있다. 이번 테스트는 특정 스레드에서 `exit`을 호출했을 때 해당 프로세스에 속한 모든 스레드가 종료되는지를 평가하는 테스트였다. 첨부한 결과와 같이 한 스레드 내에서 `exit`이 호출됐을 때 해당 프로세스의 모든 스레드가 성공적으로 종료되고, 바로 쉘로 빠져나오는 것을 확인할 수 있었다.

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

4) thread_kill.c

이번 테스트에서는 프로세스가 Kill 됐을 때 해당 프로세스의 모든 스레드가 올바르게 종료되는지 확인한다. 해당 테스트 프로그램의 전반적인 흐름은 프로그램이 `fork`를 통해 두 개의 프로세스로 나뉘지고, 각각 5개씩의 스레드를 생성한다. 이후 부모 프로세스의 스레드 하나가 자식프로세스를 kill하게 되고, 이때 자식 프로세스의 모든 스레드는 즉시 종료된다. 위 테스트를 실행해 본 결과, `kill`을 실행했을 때 자식 프로세스의 모든 스레드가 즉시 종료되는 것을 오류 없이 확인할 수 있었다.

```
$ thread_kill
Thread kill test start
Killing process 34
This code should be executed 5 times.
TThis code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
his code should be executed 5 times.
Kill test finished
$ █
```

4. Trouble Shooting

이번 프로젝트를 하면서 마주한 가장 큰 문제는 lock을 잡는 과정에서 아직 lock이 release 되지 않아 발생하는 panic: acquire 에러였다. 디버깅을 하는 과정에서 정말 많은 acquire 오류를 경험했는데, 복잡한 구조에서 lock이 acquire & release되는 구조이다 보니 해당 flow를 이해하는 데에 많은 시간이 걸렸다. 디버깅 과정에서 이를 해결하기 위해 acquire(&ptable.lock);을 하는 코드와 전체 흐름을 cprintf를 통해 전부 찍어보며 해당 문제를 해결할 수 있었다.

```
Booting from Hard Disk.. xv6...
userinit
allocproc 1
allocproc 2
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
exec.c
init: starting sh
fork syscall
allocproc 1
allocproc 2
exec.c
$ thread_test
fork syscall
allocproc 1
allocproc 2
growproc
exec.c
Test 1: Basic test
thread_create
allocproc 1
allocproc 2
in thread_create before acquire 1
in thread_create after acquire 2
lapicid 0: panic: acquire
80104daa 801043f4 80104ab9 801005a4 80101270 80105529 801051d9 8010634d 801060e4 0
```

두번째로는, 특정 프로세스의 하위 스레드를 정리하는 과정에서 어려움을 겪었다. 처음 구현을 할 때에는 thread_remove 함수를 따로 만들지 않고, ptable을 돌며 현재 스레드만을 남기고, 특정 프로세스의 다른 모든 스레드를 종료를 해주었는데, 해당 과정이 예상한 대로 실행되지 않았다. 정확한 이유는 잘 모르겠지만, 예상하곤 한번 ptable을 돌 때 감지하지 못한 스레드의 경우 초기화가 되지 않고, 지나가는 것 같았다. 한번에 하위 스레드가 정리되지 못하고, 남아 있는 것 같았다. 이로 인해, 테스트 프로그램을 실행할 때 결과는 올바르게 출력됐지만, 해당 테스트가 끝나고 셸로 돌아오지 않았다. 이를 해결하기 위해, 테이블을 도는 코드를 여러 번 수정해봤지만 잘 동작하지 않았고, 특정 pid를 인자로 전달 받는 thread_remove라는 함수를 따로 구현해주었다. 해당 함수 내에서 curproc을 제외한 해당 프로세스의 스레드를 모두 정리해주도록 했더니, 올바르게 종료되는 것을 확인했다.

2) Locking Wiki

1. Design

우선 locking을 구현하기 위해 운영체제 수업시간에 배운 Peterson algorithm과 block/wakeup semaphore를 직접 구현해보고자 한다. NUM_ITERS와 NUM_THREADS의 값을 변경하면서 Peterson algorithm을 활용해 테스트 해본 결과 각 값이 80이 넘어갈 때 busy waiting으로 인해 꽤나 많은 시간이 걸리는 것을 알 수 있었다. 따라서 condition_check 함수를 정의해 NUM_ITERS와 NUM_THREADS 값이 80이 넘어가면 block/wakeup semaphore 알고리즘을 사용하고, 그게 아니면 Peterson algorithm을 사용하도록 구현해보려고 한다.(condition 전역변수로 알고리즘 구분, 0 -> Peterson / 1-> block/wakeup) 이때 semaphore 변수를 Peterson algorithm으로 감싸 해당 변수의 race condition을 방지하려고 한다. (이미 구현된 semaphore가 아닌 struct를 통해 직접 구현할 것이다). Block/Wakeup 방식에서는 수업시간에 배운 것과 같이 특정 스레드가 critical section에 들어가지 못하는 상황인 경우 해당 스레드를 busy waiting 시키는 것이 아니라 이를 재운다. 이후 critical section을 끝낸 스레드가 생긴 경우 자고 있는 스레드를 깨워 주도록 구현할 것이다.

2. Implementation

Peterson algorithm을 위해 전역변수로 의도를 나타내는 flag와 교통정리를 해줄 turn을 선언해주었다. (main 함수에서 thread 생성 전에 초기화)

```
// flag and turn global variable for peterson algorithm
int flag[NUM_THREADS];
int turn[NUM_THREADS];
```

앞서 얘기한대로 NUM_THREADS와 NUM_ITERS값에 기반해 peterson과 block&wakeup 알고리즘 중 선택을 하도록 했다.

```
int condition = 0; // if 0 -> peterson algorithm, if 1 -> block&awake algorithm

void condition_check(void) {
    // if # of threads and iters are large, use sleep&wakeup algorithm
    if (NUM_THREADS >= 80 && NUM_ITERS >= 80)
        condition = 1;
    // else use peterson algorithm
    else
        condition = 0;
}
```

Peterson algorithm의 entry 부분으로 NUM_THREADS만큼 루프를 돌며 현재 tid의 인덱스에 해당하는 flag 값을 1로 변경한다. \hookrightarrow (critical section에 들어가려고 함을 의미) 또한 tid의 turn 값을 현재 i로 설정한다.(이는 일종의 우선순위 역할을 해주어 먼저 온 스레드가 먼저 실행될 수 있도록 해준다.) 이후, wait 변수를 선언하고 do while문을 실행한다. 이때 NUM_THREADS만큼 다시 한번 루프를 돌면서 critical section을 실행 중인 스레드가 있는지 확인한다. 이때 사용한 조건문은 일단 현재 tid 이외의 스레드를 찾는 것이므로 $j \neq i$ 이고, j라는 프로세스의 의도를 나타내는 flag[j]가 1이어야 하고(critical section에 들어갈 의도가 있음), 앞서 설정한 tid의 turn값이 j인덱스의 turn보다 크거나 혹은 같은 경우 tid가 j보다 크다면 busy waiting을 돌게 된다. (이때 turn값이 작을 수록 우선순위를 갖도록 만들었다.) 만약 실행 중인 스레드가 없다면 critical section에 들어가게 된다.

```
void peterson_entry(int tid)
{
    for (int i = 0; i < NUM_THREADS; i++) {
        flag[tid] = 1; // thread with tid try to enter critical section
        turn[tid] = i;

        int wait; // busy waiting if wait == 1, else enter critical section
        do {
            wait = 0;
            for (int j = 0; j < NUM_THREADS; j++)
            {
                // if there exists a thread running critical section
                if (j != i && flag[j] && (turn[tid] > turn[j] || (turn[tid] == turn[j] && tid > j)))
                {
                    wait = 1;
                    break;
                }
            }
        } while (wait);
    }
}
```

Peterson algorithm에 exit 부분으로 tid가 critical section을 빠져나왔음으로 의도를 나타내는 flag[tid]를 0으로 설정해준다.

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

이 부분은 block/wakeup 알고리즘에 사용할 부분으로 block/wakeup semaphore를 구현하고자 한다. 이때 semaphore 구조체를 정의해 접근 가능한 스레드 수 통제를 위한 int value와 조건이 충족되지 않을 경우 블록 시킬 스레드를 매달아두기 위해 thread를 위한 list를 구현하고, 해당 리스트에 마지막에 연결하기 위해 tail이라는 특성 또한 추가하였다. 또한 threadNode 구조체를 구현해 각 스레드를 매달아두기 위해 이를 나타낼 수 있는 tid값과 thread_list 내에서 매달린 스레드의 순서를 알기 위해 prev와 next를 추가해주었다. 이후 전역 변수로 Semaphore State를 선언하였다.

```
// struct node to save blocked threads
typedef struct threadNode{
    int tid;
    struct threadNode *prev;
    struct threadNode *next;
} threadNode;

// implemented semaphore
typedef struct {
    int value;    // value for semaphore
    struct threadNode *tail;
    struct threadNode *thread_list;
} Semaphore;

// Global State and LinkedList for blocked threads
Semaphore State;
```

이 부분은 lock에 대한 함수로 앞서 설정한 condition 값에 따라 다른 종류의 locking 알고리즘을 사용한다. Condition 0인 경우, Peterson algorithm이기 때문에 단순히 Peterson_entry(tid);를 호출하고, Condition이 1인 경우, block/wakeup algorithm을 구현하고자 했다. 이때 State라는 공용 변수에 접근할 때 발생할 수 있는 race condition을 막기 위해 내부적으로 Peterson 알고리즘으로 감싸주었다. 그 내부에서는 Block/wakeup semaphore와 비슷하게 State.value를 1 감소시킨다. 이때 isSleep이라는 변수는 특정 프로세스를 블록시켰을 때 Peterson_exit의 중복실행을 방지하기 위함이다. 이후 조건을 통해 State.value < 0인 경우 이는 critical section을 실행 중인 스레드가 있음을 의미하기 때문에 새로운 threadNode를 생성하고 tid값을 설정해준 뒤 State.thread_list에 매달아준다. 이후 block을 통해 해당 스레드를 재우려고 했지만, block이라는 시스템콜을 지원하지 않기 때문에 이미 구현된 library를 사용하지 않고, 이를 구현하지 못했다. 유동적으로 스레드가 끝나면 block/wakeup을 수행하는 코드를 구현하지 못했기 때문에 sleep으로 일정시간 재우는 방식을 어쩔 수 없이 작성하였다.

```
void lock(int tid)
{
    if(condition)
    {
        peterson_entry(tid);
        int isSleep = 0;
        State.value--;
        if (State.value < 0)
        {
            // malloc new thread Node
            threadNode *newNode = (threadNode*)malloc(sizeof(threadNode));
            newNode->tid = tid;
            newNode->next = NULL;
            newNode->prev = NULL;
            if (State.thread_list == NULL)
            {
                State.thread_list = newNode;
            }
            else
            {
                newNode->prev = State.tail;
                State.tail->next = newNode;
            }
            State.tail = newNode;
            isSleep = 1;
            peterson_exit(tid);
            //block();    -> since block() is not supported, used sleep instead
            sleep(100);
        }
        if (isSleep)
            peterson_exit(tid);
    }
    else
    {
        peterson_entry(tid);
    }
}
```

이 부분은 lock을 내려놓는 unlock에 해당하는 부분으로 마찬가지로 condition 값을 기준으로 알고리즘을 나눠 놓았다. Peterson algorithm의 경우 단순 Peterson_exit(tid);를 실행해 주었고, block/wakeup 알고리즘에서는 wakeup을 수행하고자 했다. 여기서도 마찬가지로, Peterson 알고리즘으로 State의 접근을 감싸주었고, State.value <= 0인 경우 기다리는 스레드가 있다는 것을 의미하기 때문에 thread_list의 head에 있는 스레드를 wakeup시켜주고 해당 스레드를 linkedlist에서 삭제하고자 했지만, 특정 tid를 가진 스레드를 깨우는 것을 구현하지 못했다. 기존의 library 함수를 이용하지 않고, 특정 tid의 스레드를 깨우는 것을 구현하고자 했지만 이에 실패했기 때문에, 앞서 설정한 sleep()으로 이를 대체할 수 밖에 없었다. 이후 깨우고자 했던 노드를 free 시켜준다.

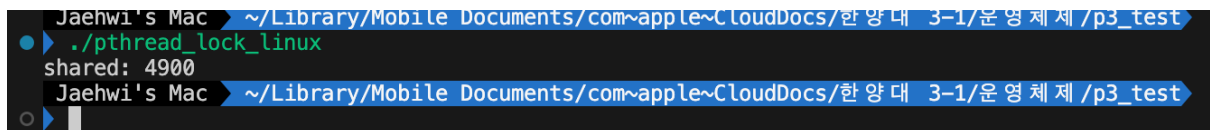
```
void unlock(int tid)
{
    if(condition)
    {
        peterson_entry(tid);
        State.value++;
        if (State.value <= 0)
        {
            threadNode *Node = State.thread_list;
            State.thread_list->next->prev = NULL;
            State.thread_list = State.thread_list->next;
            // wakeup(Node->tid); // want to wakeup a thread with particular tid, but it is not supported.
            free(Node);
        }
        peterson_exit(tid);
    }
    else
    {
        peterson_exit(tid);
    }
}
```

이 부분은 main 함수 내의 코드로 thread 생성 전 flag와 turn을 0으로 초기화 해준다. 주석 처리 해둔 condition_check()를 통해 NUM_THREAD와 NUM_ITERS 값을 기반으로 사용할 알고리즘을 선택하고자 했다. 하지만 block/wakeup 알고리즘의 block과 wakeup 기능을 성공적으로 구현하지 못했기 때문에 해당 함수를 주석처리 해 두었다. Condition 이 1인 경우 block/wakeup 알고리즘을 위해 State의 특성을 초기화 해준다.

```
// initialize flags and turn
for (int i = 0; i < NUM_THREADS; i++) {
    flag[i] = 0;
    turn[i] = 0;
}
// failed to implement, Block/Wakeup so fixed condition to 0
// condition_check();
// initialize LinkedList when using Block/Wakeup algorithm (but failed to implement)
if (condition)
{
    State.value = 1;
    State.thread_list = NULL;
    State.tail = NULL;
}
```

3. Result

Peterson 알고리즘을 활용했을 때 결과는 race condition없이 잘 나온다. 그러나 NUM_THREADS와 NUM_ITERS 값이 커질 때 많은 시간이 걸리게 된다.

A terminal window screenshot showing a command prompt on a Mac. The prompt is 'Jaehwi's Mac ~/Library/Mobile Documents/com~apple~CloudDocs/한양대 3-1/운영 체제 /p3_test'. The command entered is './pthread_lock_linux'. The output is 'shared: 4900'. The prompt is repeated below the output.

4. Trouble Shooting

이외의 코드는 제공해주신 파일의 코드와 일치하다. 위의 아이디어를 기반으로 상황에 따라 busy waiting의 여부를 결정해 속도를 빠르게 하고자 했지만, 결국 busy waiting 최소화하는 block/wakeup 알고리즘을 라이브러리 함수의 사용 없이 구현하지 못했다. 특정 tid를 가진 스레드의 wakeup을 구현할 수만 있었다면 locking 구현을 원하는 대로 할 수 있었을텐데, 이 방법을 잘 모르겠다.