Project02
wiki

이름: 권도현
학번: 2023065350
학과: 컴퓨터소프트웨어학부

Preview

명세서에 구현에 앞서, 자세히 살펴보라고 알려주신 아래 4가지의 코드를 분석하며, 주석을 달아보았다.

세부적인 내용 외에도 전체적인 흐름도 파악하기 위해 Wiki에 따로 정리해보았다.

1. `fork()`

   np를 새롭게 생성할 child process를 가리키는 pointer 변수로 사용하여, `open file`을 복사할 수 있도록 한다.

   Child process가 `Open file`을 참조하기 때문에 file reference를 1 증가시킨다.

   새롭게 생성되는 process의 pid, parent, state을 초기화한다.

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy user memory from parent to child.
```

```
// uvmcopy: 메모리 복사 후 오류 체크까지
if(uvmcopy(p→pagetable, np→pagetable, p→sz) < 0){
  freeproc(np);
  release(&np→lock);
  return -1;
}
// Address space size 복사
np→sz = p→sz;

// copy saved user registers.
*(np→trapframe) = *(p→trapframe);

// Cause fork to return 0 in the child.
np→trapframe→a0 = 0;

// increment reference counts on open file descriptors.
// 해당 file을 참조하는 process 수 증가
for(i = 0; i < NOFILE; i++)
  if(p→ofile[i])
    np→ofile[i] = filedup(p→ofile[i]);
np→cwd = idup(p→cwd);

safestrcpy(np→name, p→name, sizeof(p→name));

// parent process에 child process's pid return
pid = np→pid;

release(&np→lock);

// child process parent & state setting
acquire(&wait_lock);
np→parent = p;
release(&wait_lock);

acquire(&np→lock);
np→state = RUNNABLE;
release(&np→lock);
```

```
  return pid;
}
```

2. `exec()`

   `ELF header`를 검사하여 다른 process를 실행하기 위해 필요한 정보를 얻고 대체되어야 할 process가 실행이 가능한 지 판단한다.

   완전히 새로운 program으로 덮는 것이기 때문에 아래와 같은 동작을 한다.

   1. 새로운 `user stack`을 구성하여 `stack guard`를 설정하고, argument를 저장한다.
   2. 새로 실행할 process에 맞추어 `pagetable`을 새롭게 생성한다.
   3. User level로 돌아가 정상적인 동작을 하도록 `trapframe`을 세팅한다.

```
int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  uint64 argc, sz = 0, sp, ustack[MAXARG], stackbase;
  struct elfhdr elf; // ELF file header
  struct inode *ip; // current directory
  struct proghdr ph; // program 관련 정보
  pagetable_t pagetable = 0, oldpagetable;
  struct proc *p = myproc();

  // process가 파일을 열어 작업 중이면 대기, 아니라면 파일 수정을 위해 공간 확보
  begin_op();

  if((ip = namei(path)) == 0){
    end_op();
    return -1;
  }
  ilock(ip);

  // Check ELF header
  if(readi(ip, 0, (uint64)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;
```

```
if(elf.magic != ELF_MAGIC)
  goto bad;

if((pagetable = proc_pagetable(p)) == 0)
  goto bad;

// Load program into memory.
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
  if(ph.type != ELF_PROG_LOAD)
    continue;
  if(ph.memsz < ph.filesz)
    goto bad;
  if(ph.vaddr + ph.memsz < ph.vaddr)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  uint64 sz1;
  if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz, flags2perm(ph.
    goto bad;
  sz = sz1;
  if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
iunlockput(ip);
// ELF Load 후 문제가 없다면 파일 관리 시스템 종료 표시
end_op();
ip = 0;

p = myproc();
uint64 oldsz = p→sz;

// Allocate some pages at the next page boundary.
// Make the first inaccessible as a stack guard.
// Use the rest as the user stack.
// sz = logical address space size
sz = PGROUNDUP(sz);
```

```
uint64 sz1;
if((sz1 = uvmalloc(pagetable, sz, sz + (USERSTACK+1)*PGSIZE, PTE_W))
  goto bad;
sz = sz1; // 할당 받은 size
uvmclear(pagetable, sz-(USERSTACK+1)*PGSIZE); // Stack guard → 접근
sp = sz;
stackbase = sp - USERSTACK*PGSIZE;

// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
  if(argc >= MAXARG)
    goto bad;
  sp -= strlen(argv[argc]) + 1; // argument legnth만큼 공간 확보
  sp -= sp % 16; // riscv sp must be 16-byte aligned
  if(sp < stackbase)
    goto bad;
  if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
  ustack[argc] = sp; // ustack에 각 argument가 저장된 공간을 저장
}
ustack[argc] = 0; // User program에서 argument가 더 이상 없음을 확인하기 위

// push the array of argv[] pointers.
sp -= (argc+1) * sizeof(uint64);
sp -= sp % 16;
if(sp < stackbase)
  goto bad;
if(copyout(pagetable, sp, (char *)ustack, (argc+1)*sizeof(uint64)) < 0)
  goto bad;

// arguments to user main(argc, argv)
// argc is returned via the system call return
// value, which goes in a0.
p->trapframe->a1 = sp;

// Save program name for debugging.
for(last=s=path; *s; s++)
  if(*s == '/')
```

```
    last = s+1;
  safestrcpy(p->name, last, sizeof(p->name));

  // Commit to the user image.
  // pagetable을 교체 후, epc, sp를 초기화
  oldpagetable = p->pagetable;
  p->pagetable = pagetable; // 새로운 pagetable로 교체
  p->sz = sz; // Logical memory address size도 교체
  p->trapframe->epc = elf.entry;  // initial program counter = main
  p->trapframe->sp = sp; // initial stack pointer
  proc_freepagetable(oldpagetable, oldsz); // 기존 pagetable 할당 해제

  return argc; // this ends up in a0, the first argument to main(argc, argv)

 bad:
  if(pagetable)
    proc_freepagetable(pagetable, sz);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
}
```

3. **exit()**

   현재 process가 `init process`인지 확인하고, init process가 아니라면 종료

   종료할 process의 부모 process가 `sleep` 상태라면 `Wakeup`

   Resource를 할당 해제하지 않고 `ZOMBIE state`로 변경 후, scheduler 호출

   - Resource는 wait()에서 할당 해제할 수 있도록 한다.

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

  // init process는 절대 종료되지 않도록 방지
  if(p == initproc)
```

```
  panic("init exiting");

// Close all open files.
for(int fd = 0; fd < NOFILE; fd++){
  if(p→ofile[fd]){
    struct file *f = p→ofile[fd];
    fileclose(f);
    p→ofile[fd] = 0;
  }
}
// process가 파일을 열어 작업 중이면 대기, 아니라면 파일 수정을 위해 공간 확보
// cwd에는 현재 실행중인 directory의 inode가 (파일 정보) 저장
// iput()을 이용해 inode 정리 및 reference 수 감소
begin_op();
iput(p→cwd);
end_op();
p→cwd = 0;

acquire(&wait_lock);

// Give any children to init.
// Orphan process 생기는 것을 방지
// exit()을 호출하는 process의 child process는 init process을 parnet로 가지도
reparent(p);

// Parent might be sleeping in wait().
wakeup(p→parent);

acquire(&p→lock);

// 종료 상태 코드 저장, 현재 상태 = ZOMBIE
p→xstate = status;
p→state = ZOMBIE;

release(&wait_lock);

// Jump into the scheduler, never to return.
sched(); //  Scheduler 호출
```

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

4. `wait()`

   모든 Process를 확인하며, 현재 실행 중인 process를 parent로 갖는 process가 있는지 확인한다.

   찾은 child process가 종료되어 `ZOMBIE` 상태라면 종료 상태를 반환 받고, 자원을 할당 해제 한다.

   `Child process`가 존재하나, 아직 실행 중이라면 대기한다.

```
int
wait(uint64 addr)
{
  struct proc *pp;
  int havekids, pid;
  struct proc *p = myproc();

  acquire(&wait_lock);

  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(pp = proc; pp < &proc[NPROC]; pp++){
      if(pp->parent == p){ // 현재 process를 parent로 갖는 process가 존재한다면
        // make sure the child isn't still in exit() or swtch().
        acquire(&pp->lock);

        havekids = 1;
        if(pp->state == ZOMBIE){ // Child process가 ZOMBIE 상태라면 resourc
          // Found one.
          pid = pp->pid;
          // Child process의 종료 상태를 Parent process에 전달
          if(addr != 0 && copyout(p->pagetable, addr, (char *)&pp->xstate,
                                  sizeof(pp->xstate)) < 0) {
            release(&pp->lock);
            release(&wait_lock);
            return -1;
```

```
        }
        freeproc(pp); // Child process Resource deallocation
        release(&pp->lock);
        release(&wait_lock);
        return pid;
      }
      release(&pp->lock); // child process가 없는 경우에 release
    }
  }

  // No point waiting if we don't have any children.
  if(!havekids || killed(p)){
    release(&wait_lock);
    return -1;
  }

  // Wait for a child to exit.
  // Chile process 존재 & 아직 종료되지 않음
  sleep(p, &wait_lock);  //DOC: wait-sleep
 }
}
```

위 네 가지의 함수들은 system call을 기반으로 동작하며, user program에서 system call을 호출하면, System call number에 따라 커널이 관련된 함수를 호출한다.

Trap 관련 정리

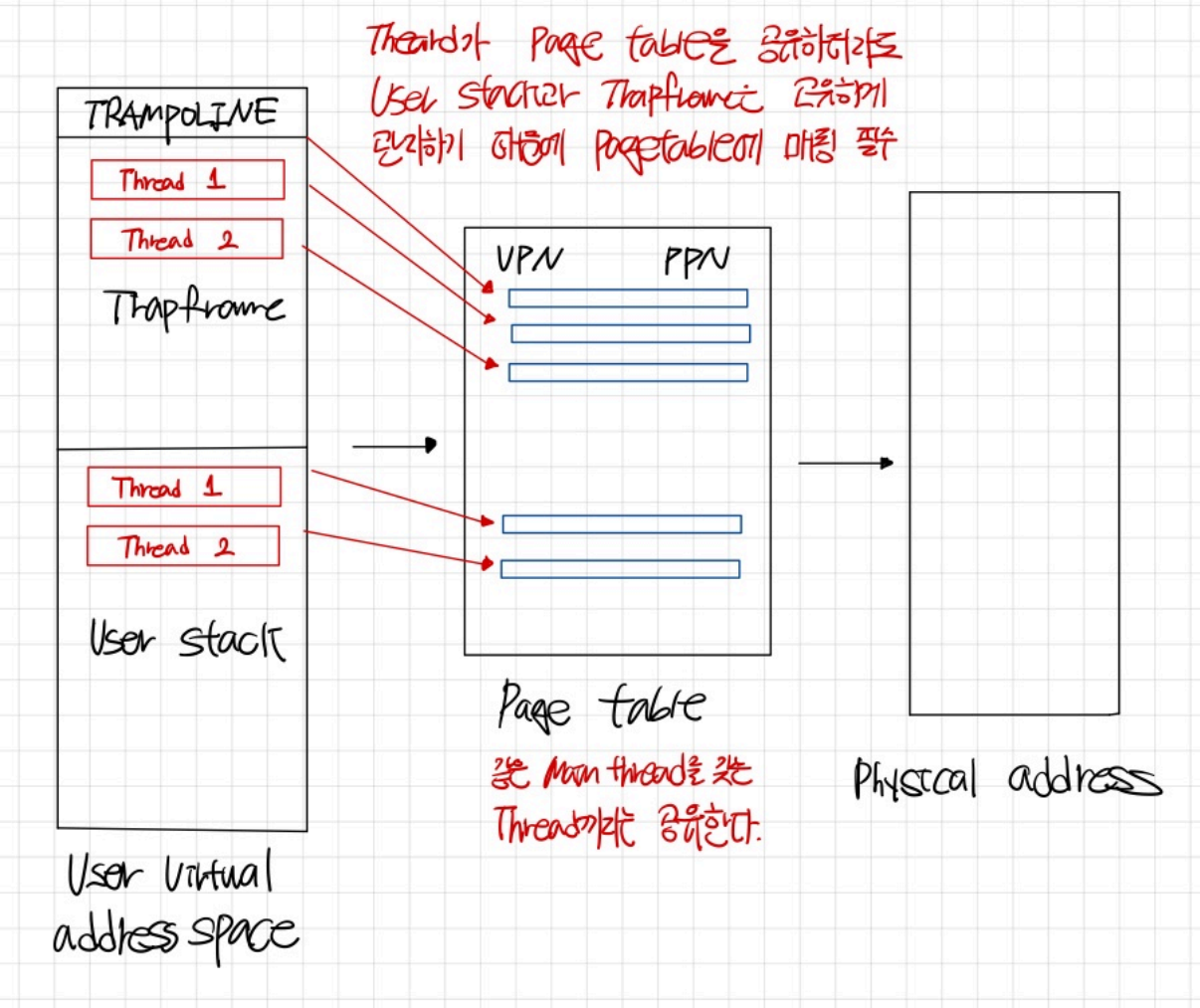
`trap.c` : 여러 예외 사항을 확인하고, return address를 저장한 후에 syscall()을 호출한다.

```
void
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0)
```

```
  panic("usertrap: not from user mode");

// send interrupts and exceptions to kerneltrap(),
// since we're now in the kernel.
w_stvec((uint64)kernelvec);

struct proc *p = myproc();

// save user program counter.
p->trapframe->epc = r_sepc();

if(r_scause() == 8){
  // system call

  if(killed(p))
    exit(-1);

  // sepc points to the ecall instruction,
  // but we want to return to the next instruction.
  p->trapframe->epc += 4;

  // an interrupt will change sepc, scause, and sstatus,
  // so enable only now that we're done with those registers.
  intr_on();

  syscall();
} else if((which_dev = devintr()) != 0){
  // ok
} else {
  printf("usertrap(): unexpected scause 0x%lx pid=%d\n", r_scause(), p->pid
  printf("            sepc=0x%lx stval=0x%lx\n", r_sepc(), r_stval());
  setkilled(p);
}

if(killed(p))
  exit(-1);

// give up the CPU if this is a timer interrupt.
```

```
  if(which_dev == 2)
    yield();

  // User mode로 변경
  usertrapret();
}
```

Memory 관련 정리



- Main thread를 포함한 모든 thread는 `user virtual address space` 내에 존재한다.
- User virtual address space의 최상단은 user ↔ kernel 사이의 전환을 지원하는 `TRAMPOLINE`이 차지한다.
- 아래로 각 thread의 `trapframe, user stack` 영역이 위치한다.

- 각 thread는 각기 다른 Trapframe과 user stack을 가지고, page table은 main thread와 공유한다.
    - 따라서, 각 thread의 trapframe과 user stack을 main thread의 `page table에 매핑` 해주어야 한다.

PROCESS vs THREAD

본 과제는 `Thread`를 지원하기 위해 필요한 함수를 구현하고, 기존에 process를 지원하던 system call 함수들을 수정하는 것이기 때문에 process와 thread의 차이를 살펴보았다.

우선 Thread는 하나의 process 안에서도 여러 개 실행 단위로 나뉠 수 있다.

이 경우, 모든 Thread는 동일한 `Address space (page table)와 파일 디스크립터`를 공유한다.

그러나, `Register 상태(trapframe)와 stack`는 독립적으로 가진다.

이러한 관점에서 Thread를 `Light Weight Process (LWP)`라고도 부른다.

반면, Process는 자원 공유가 불가능하고 독립된 실행 단위이다.

먼저 기존 xv6에서 구현된 `process struct`를 확인해보자

```
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
```

```
  uint64 trapframe_va;        // virtual address of the trapframe
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

Thread를 지원하는 `TCB`는 process struct를 수정하여 사용할 것이다.

- 기존 proc struct에 main_thread를 가리키는 포인터와 main_thread인지 확인하는 필드가 추가로 필요하다.

Thread에서 공유해야 하는 부분은 main_thread의 포인터를 거쳐서 사용하거나, 복사해서 사용한다.

Design

1. `TID`: 과제 명세서에서 clone system call 구현을 설명하는 페이지에 “Returns the thread’s ID (PID)” 라고 적혀있어 `clone / join`에서 return 하는 값은 기존 PID가 부여되는 방식을 그대로 사용하기로 하였다.
    - 그러나 각 thread가 고유의 trapframe을 가지긴 위해서는 tid가 구분되어야 하기 때문에, main_thread 에서 `thread_num field`로 현재 child thread의 개수를 추적하며, `tid`를 부여할 수 있도록 하였다.

2. `exec()` 에서 process table을 순회하며 thread를 제거하기 위해서 proc.c에서 proc 구조체를 `extern`으로 선언한 후, proc.h에서 정의하고 proc.c에서 선언하도록 하였다.

3. proc struct에 `thread`를 지원하기 위해 아래와 같은 필드를 추가하였다.

```
// Thread related
int isThread;               // 0: process, 1: thread
int tid;                    // Thread ID
int thread_num;             // thread의 개수 (tid 할당)
void *stack;

// main_thread == 0: 지금 process가 main thread
```

```
// main_therad != 0: 해당 thread가 속한 process의 main_thread
struct proc *main_thread;
```

- `isThread` : 0이면 main thread임을 나타내고, 1이라면 main thread가 아님을 나타낸다.
- `tid` : 각 thread의 ID
- `thread_num` : main thread가 child thread의 개수를 관리하며 tid를 부여하기 위해 사용한다.
- `stack` : kernel에서 user stack을 관리하기 위한 포인터
- `main_thread` : 각 thread의 main_thread를 가리키는 포인터
  - 각기 다른 thread가 같은 main_thread를 가지는 지는 해당 thread의 main_thread가 같은 proc struct를 가리키는 지를 통해 확인한다.
  - vm.c 파일에 존재하는 함수들은 sz field를 사용하는데, 구현 과정에서 thread마다 관리하니 쉽게 memory 관련 panic을 발생했다. 이를 방지하기 위해, thread가 공유해야 하는 부분 중 sz는 main_thread만이 관리하고, main_thread를 거쳐서 확인할 수 있도록 구현하였다.
  - 그 외 Thread가 공유하는 부분은 다른 proc struct의 필드를 최대한 활용하기 위해 clone 과정에서 main thread의 값을 복사하여 사용하였다.
- 추가로, 기존에 존재하던 `parent field` 는 thread의 경우에는 main_thread를 가리키도록 구현하였다.

4. 우선 기존 process를 지원하던 system call을 참조하여 `clone / join` 을 구현하고 이후에 system call을 수정해나가는 방식으로 구현하였다.

Implements

새롭게 만든 `system call` 을 지원하기 위한 구현과 User 폴더에 추가한 파일들을 지원하기 위해 MAKEFILE을 수정하는 작업은 사전에 완료했다.

clone / join 구현

1. thread_create

```
int
thread_create(void (*start_routine)(void*, void*), void *arg1, void* arg2)
{
   int tid;

   void *stack = malloc(2 * PGSIZE);
   if (stack == 0) return -1;

   if ((tid = clone(start_routine, arg1, arg2, stack)) < 0) {
      free(stack);
      return -1;
   }

   return tid;
}
```

- 기존 xv6에서 process에 새롭게 stack을 할당할 때, `page gaurd`를 위해 `2*PGSIZE`만큼 할당하였다.
- 이를 참고하여 thread에서도 page gaurd를 지원하기 위해 `2*PGSIZE size malloc()`을 했다.

2. sys_clone

```
uint64
sys_clone(void)
{
  uint64 fcn, arg1, arg2, stack;

  argaddr(0, &fcn);
  argaddr(1, &arg1);
  argaddr(2, &arg2);
  argaddr(3, &stack);

  return clone((void (*)(void*, void*))fcn, (void *)arg1, (void *)arg2, (void *)stac
}
```

- arguement를 적절히 넘겨주고 clone()을 호출한다.
- `argaddr()`은 user level의 argument를 kernel 변수로 옮겨준다.

3. clone

```
int
clone(void (*fcn)(void*, void*), void *arg1, void *arg2, void *stack)
{
  struct proc *p = myproc();
  struct proc *nt;

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

  // Thread 관련 부분 처리
  nt->isThread = 1;
  nt->main_thread = p->isThread ? p->main_thread : p;
  nt->parent = nt->main_thread;

  acquire(&nt->main_thread->lock);
  nt->pagetable = nt->main_thread->pagetable;
  nt->tid = ++nt->main_thread->thread_num;
  release(&nt->main_thread->lock);

  // 각 Thread가 File descriptor copy를 가짐
  for(int i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      nt->ofile[i] = filedup(p->ofile[i]);

  nt->cwd = idup(p->cwd);

  safestrcpy(nt->name, p->name, sizeof(p->name));

  // map the trapframe page just below the trampoline page, for
  // trampoline.S.
  if(mappages(nt->pagetable, nt->trapframe_va = TRAMPOLINE - (nt->tid * PGS
```

```
            (uint64)(nt->trapframe), PTE_R | PTE_W) < 0){
    freeproc(nt);
    release(&nt->lock);
    return 0;
  }

  // trapframe 우선 복사
  *(nt->trapframe) = *(p->trapframe);
  // fcn, stack, argument를 넘겨줌
  nt->trapframe->epc = (uint64)fcn;
  nt->trapframe->a0 = (uint64)arg1;
  nt->trapframe->a1 = (uint64)arg2;

  void *user_stack = (void *)(((uint64)stack + PGSIZE - 1) & ~(PGSIZE - 1));
  nt->trapframe->sp = (uint64)user_stack + PGSIZE;

  nt->stack = stack;

  nt->state = RUNNABLE;
  release(&nt->lock);

  return nt->pid;
}
```

- `fork() + exec()`을 참고하여 구현하였다.
- `clone()`은 main thread를 생성하는 코드가 아니기 때문에 `allocproc(1)`을 통해 thread에 맞게 proc struct를 초기화 하였다.
- `파일 디스크립터`는 명세서대로 각 thread가 복사본을 가지도록 구현하였다.
- `Thread filed`를 초기화하는 부분은 위의 DESIGN에서 설명했다.
- `tid`를 각기 다르게 가지도록 하여 각 thread가 pagetable은 공유하지만 trapframe에 매핑되는 `trapframe_va`는 각기 다르게 갖도록 구현하였다.
  - Thread를 처리하는 다른 코드에서는 pid를 사용하지만, tid field를 추가한 이유이다.
  - 위처럼 하지 않으면 thread간의 trapframe이 충돌하여 `panic: remap`이 발생한다.
- 각 thread의 tramframe_va를 main thread의 page table에 매핑한다.

- `clone()`을 호출한 후, user level로 돌아가 arg1, arg2를 argument로 갖는 fcn 함수를 실행할 수 있도록 trapframe을 적절히 설정했다.
- `user stack`이 page alligned될 수 있도록 하고, `$sp`를 stack의 최상단을 가리키도록 한다.

4. thread_join

```
int
thread_join()
{
   int tid;
   void *stack;

   if ((tid = join(&stack)) < 0) {
      return -1;
   }

   free(stack);
   return tid;
}
```

- `join`의 parameter에 &stack을 넘겨주어서 kernel의 join system call에서 할당 해제해야 하는 stack의 주소를 받아온다.
- Thread를 생성할 때, stack을 user level(thread_clone())에서 할당했기 때문에 똑같이 user level(thread_join())에서 할당 해제할 수 있도록 한다.

5. sys_join

```
uint64
sys_join(void)
{
  uint64 stack;

  argaddr(0, &stack);
  return join((void **)stack);
}
```

- argument를 적절히 넘겨주고, join()을 호출하도록 한다.

6. join

```
int
join(void **stack)
{
  int havekids, pid;
  struct proc *p = myproc();
  struct proc *main_thread = p->isThread ? p->main_thread : p;
  struct proc *t;

  acquire(&wait_lock);

  for (;;) {
    havekids = 0;

    for (t = proc; t < &proc[NPROC]; t++) {
      if (t->main_thread == main_thread && t->isThread) {
        acquire(&t->lock);
        havekids = 1;

        if (t->state == ZOMBIE) {
          pid = t->pid;

          copyout(main_thread->pagetable, (uint64)stack, (char *)&t->stack, sizeo

          freeproc(t);

          release(&t->lock);
          release(&wait_lock);

          return pid;
        }

        release(&t->lock);
```

```
    }
   }

   if (!havekids || killed(p)) {
    release(&wait_lock);
    return -1;
   }

   sleep(p, &wait_lock);
  }
 }
```

- `wait()`을 참고하여 구현하였다.
- Resource 회수는 반드시 `main thread`만이 할 수 있기 때문에, 현재 join()을 호출한 thread의 main thread를 찾아서 포인터 변수에 저장하였다.
- proc table을 순회하며 main thread가 아니며, main_thread를 main thread로 갖는 thread를 찾는다.
- 해당 thread가 `ZOMBIE 상태`인 경우 아래 두 가지 동작을 한다.
  - user level에서 부여된 stack을 user level에서 free 해야 하기 때문에 자원을 해제하기 전 해당 thread가 사용하던 stack 주소를 main_thread의 pagetable을 통해 사용자 공간에 복사한다.
  - freeproc()을 호출한다.

기존 코드 수정

7. allocproc

```
static struct proc*
allocproc(int isThread)
{
  struct proc *p;

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

```
    acquire(&p->lock);
    if(p->state == UNUSED) {
      goto found;
    } else {
      release(&p->lock);
    }
  }
  return 0;

found:
  p->pid = allocpid();
  p->state = USED;

  // Allocate a trapframe page.
  if((p->trapframe = (struct trapframe *)kalloc()) == 0){
    release(&p->lock);
    freeproc(p);
    return 0;
  }

  // Set up new context to start executing at forkret,
  // which returns to user space.
  memset(&p->context, 0, sizeof(p->context));
  p->context.ra = (uint64)forkret;
  p->context.sp = p->kstack + PGSIZE;

  // Edited
  if (isThread == 0) {
    // An empty user page table.
    p->pagetable = proc_pagetable(p);
    if(p->pagetable == 0){
      release(&p->lock);
      freeproc(p);
      return 0;
    }

    p->isThread = 0;
    p->tid = 1;
```

```
    p->main_thread = 0;
    p->stack = 0;
    p->thread_num = 1;
  }

  return p;
}
```

- 기존 allocproc()와 달리, `parameter`를 갖도록 하여 main thread와 main thread가 아닌 thread가 다르게 할당 받을 수 있도록 구현하였다.
- 기존에 allocproc()에 있던 코드 중, main thread인지 여부와 관계없이 모두 수행해야 하는 부분을 제외하고, main thread가 관리해야 하는 field들은 `if (isThread) block` 내에서 처리할 수 있도록 하였다.
    - 특히, page table은 모든 thread가 공유해야 하므로 `main thread`만이 `proc_pagetable()`을 호출할 수 있도록 하였다.
- main thread가 아닌 thread가 가져야 하는 필드들은 헷갈리지 않도록 clone()에서 초기화할 수 있도록 구분하여 구현했다.

8. freeproc

```
static void
freeproc(struct proc *p)
{
  if (p->isThread == 0) {
    if(p->pagetable) {
      proc_freepagetable(p->pagetable, p->sz);
      p->thread_num = 1;
      p->sz = 0;
    }
  }
  else {
    p->isThread = 0;
    p->main_thread = 0;
    uvmunmap(p->pagetable, p->trapframe_va, 1, 0);
  }
```

```
  if(p->trapframe)
    kfree((void*)p->trapframe);
  p->trapframe = 0;
  p->trapframe_va = 0;
  p->name[0] = 0;
  p->chan = 0;
  p->killed = 0;
  p->xstate = 0;
  p->state = UNUSED;
  p->pid = 0;
  p->tid = 0;
  p->parent = 0;
  p->pagetable = 0;
}
```

- allocproc()과 비슷하게 main thread와 main thread가 아닌 thread가 다르게 할당 해제할 수 있도록 구현하였다.
- main thread인 경우에만 sz를 초기화하고, page table을 해제하도록 구현했다.
- main_thread가 아닌 경우에는 해당thread의 trapframe_va와 page table과의 매핑을 제거했다.
    - 원래는 `Unmapping`을 `proc_freeapagetable()`에서 호출하지만, main thread가 아닌 경우 page table을 해제하지 않고, mapping만 제거해야 하므로 따로 호출하도록 구현하였다.

SYSTEM CALL 수정

9. fork

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *t = myproc();
```

```
struct proc *p = t->isThread ? t->main_thread : t;

// Allocate process.
if((np = allocproc(0)) == 0){
  return -1;
}

// Copy user memory from parent to child.
if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
  freeproc(np);
  release(&np->lock);
  return -1;
}
np->sz = p->sz;

// copy saved user registers.
// trapframe은 호출한 thread의 trapframe 이용
*(np->trapframe) = *(t->trapframe);

// Cause fork to return 0 in the child.
np->trapframe->a0 = 0;

// increment reference counts on open file descriptors.
for(i = 0; i < NOFILE; i++)
  if(p->ofile[i])
    np->ofile[i] = filedup(p->ofile[i]);
np->cwd = idup(p->cwd);

safestrcpy(np->name, p->name, sizeof(p->name));

pid = np->pid;

release(&np->lock);

acquire(&wait_lock);
np->parent = t;
release(&wait_lock);
```

```
  acquire(&np->lock);
  np->state = RUNNABLE;
  release(&np->lock);

  return pid;
}
```

- fork()를 호출하는 thread가 항상 main thread라는 보장이 없다고 판단했다.
- 따라서 `main thread가 아닌 thread`가 fork()를 호출한 경우를 대비하여, main thread를 찾는 코드를 추가하였다.
- `새롭게 생성되는 process (main thread)`는 `fork()를 호출한 thread의 main thread의 필드`를 복제하여 사용한다.

10. exec

```
int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off, pagetable_changed_allowed = 1;
  uint64 argc, sz = 0, sp, ustack[MAXARG], stackbase;
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pagetable_t pagetable = 0, oldpagetable;
  struct proc *p = myproc();
  struct proc *t;
  struct proc *main = p->isThread ? p->main_thread : p;


  // exec을 호출한 thread를 main_thread로 변화
  if (p != main) {
    pagetable_changed_allowed = 0;

    for (t = proc; t < &proc[NPROC]; t++) {
      acquire(&t->lock);
```

```
    if (t->main_thread == main && t != p) {
      t->main_thread = p;
      t->parent = p;
    }
    else if (t == p->main_thread) {
      t->main_thread = p;
      t->parent = p;
      t->isThread = 1;
      t->tid = p->tid;
    }

    release(&t->lock);
  }

  p->sz = main->sz;
  p->isThread = 0;
  p->main_thread = 0;
  p->tid = 1;
}

for (t = proc; t < &proc[NPROC]; t++) {
  if (t == p) continue;

  acquire(&t->lock);
  if ((t->isThread == 1 && t->main_thread == p&& t->state != UNUSED && t->st
    t->killed = 1;
    if (t->state == SLEEPING) {
      t->state = RUNNABLE;
    }
  }
  release(&t->lock);
}

begin_op();

if((ip = namei(path)) == 0){
  end_op();
```

```
  return -1;
}
ilock(ip);

// Check ELF header
if(readi(ip, 0, (uint64)&elf, 0, sizeof(elf)) != sizeof(elf))
  goto bad;

if(elf.magic != ELF_MAGIC)
  goto bad;

if((pagetable = proc_pagetable(p)) == 0)
  goto bad;

// Load program into memory.
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
  if(ph.type != ELF_PROG_LOAD)
    continue;
  if(ph.memsz < ph.filesz)
    goto bad;
  if(ph.vaddr + ph.memsz < ph.vaddr)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  uint64 sz1;
  if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz, flags2perm(ph.flag
    goto bad;
  sz = sz1;
  if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
iunlockput(ip);
end_op();
ip = 0;

p = myproc();
```

```
uint64 oldsz = p->sz;

// Allocate some pages at the next page boundary.
// Make the first inaccessible as a stack guard.
// Use the rest as the user stack.
sz = PGROUNDUP(sz);
uint64 sz1;
if((sz1 = uvmalloc(pagetable, sz, sz + (USERSTACK+1)*PGSIZE, PTE_W)) == (
  goto bad;
sz = sz1;
uvmclear(pagetable, sz-(USERSTACK+1)*PGSIZE);
sp = sz;
stackbase = sp - USERSTACK*PGSIZE;

// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
  if(argc >= MAXARG)
    goto bad;
  sp -= strlen(argv[argc]) + 1;
  sp -= sp % 16; // riscv sp must be 16-byte aligned
  if(sp < stackbase)
    goto bad;
  if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
  ustack[argc] = sp;
}
ustack[argc] = 0;

// push the array of argv[] pointers.
sp -= (argc+1) * sizeof(uint64);
sp -= sp % 16;
if(sp < stackbase)
  goto bad;
if(copyout(pagetable, sp, (char *)ustack, (argc+1)*sizeof(uint64)) < 0)
  goto bad;

// arguments to user main(argc, argv)
// argc is returned via the system call return
```

```
  // value, which goes in a0.
  p->trapframe->a1 = sp;

  // Save program name for debugging.
  for(last=s=path; *s; s++)
    if(*s == '/')
      last = s+1;
  safestrcpy(p->name, last, sizeof(p->name));

  // Commit to the user image.
  oldpagetable = p->pagetable;
  p->pagetable = pagetable;
  p->sz = sz;
  p->trapframe->epc = elf.entry;  // initial program counter = main
  p->trapframe->sp = sp; // initial stack pointer
  if (pagetable_changed_allowed) proc_freepagetable(oldpagetable, oldsz);

  return argc; // this ends up in a0, the first argument to main(argc, argv)

 bad:
  if(pagetable)
    proc_freepagetable(pagetable, sz);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
}
```

- fork와 마찬가지로 exec()을 호출하는 thread가 main thread라는 보장이 없다고 생각했다.
- exec()을 호출한 thread가 main thread가 아니라면, 우선 exec()을 호출한 thread의 main thread를 찾았다.
- 해당 main thread의 child thread의 `parnet`가 `exec()을 호출한 thread`가 되도록 설정하였다.
- 이후 exec()을 호출한 thread가 main thread처럼 동작하도록 관련 field를 설정하였다.

- 마지막으로, proc table을 순회하며 `exec()을 호출한 thread`를 `parent로 가지는 thread`를 kill하였다.

11. sbrk

```
uint64
sys_sbrk(void)
{
  uint64 addr;
  int n;
  struct proc *p = myproc()->isThread ? myproc()->main_thread : myproc();

  argint(0, &n);
  addr = p->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}

int
growproc(int n)
{
  uint64 sz;
  struct proc *p = myproc()->isThread ? myproc()->main_thread : myproc();

  acquire(&p->lock);

  sz = p->sz;
  if(n > 0){
    if((sz = uvmalloc(p->pagetable, sz, sz + n, PTE_W)) == 0) {
      return -1;
    }
  } else if(n < 0){
    sz = uvmdealloc(p->pagetable, sz, sz + n);
  }
  p->sz = sz;
```

```
  release(&p->lock);
  return 0;
}
```

- DESIGN에서 언급한 것처럼 sz는 main thread가 아니라면 항상 main thread의 sz 값을 이용하도록 구현하였다.
- 따라서 sz를 다루는 sbrk에서 sbrk()를 호출한 thread가 main thread가 아니라면, `sbrk()를 호출한 thread의 main thread의 sz`를 사용하도록 구현하였다.

12. kill

- 기존 xv6 코드를 그대로 사용하였다.

13. sleep

- 기존 xv6 코드를 그대로 사용하였다.

14. pipe

```
int
pipewrite(struct pipe *pi, uint64 addr, int n)
{
  int i = 0;
  struct proc *pr = myproc()->isThread ? myproc()->main_thread : myproc();

  acquire(&pi->lock);
  while(i < n){
    if(pi->readopen == 0 || killed(pr)){
      release(&pi->lock);
      return -1;
    }
    if(pi->nwrite == pi->nread + PIPESIZE){ //DOC: pipewrite-full
      wakeup(&pi->nread);
      sleep(&pi->nwrite, &pi->lock);
    } else {
```

```
      char ch;
      if(copyin(pr->pagetable, &ch, addr + i, 1) == -1)
        break;
      pi->data[pi->nwrite++ % PIPESIZE] = ch;
      i++;
    }
  }
  wakeup(&pi->nread);
  release(&pi->lock);

  return i;
}

int
piperead(struct pipe *pi, uint64 addr, int n)
{
  int i;
  struct proc *pr = myproc()->isThread ? myproc()->main_thread : myproc();
  char ch;

  acquire(&pi->lock);
  while(pi->nread == pi->nwrite && pi->writeopen){  //DOC: pipe-empty
    if(killed(pr)){
      release(&pi->lock);
      return -1;
    }
    sleep(&pi->nread, &pi->lock); //DOC: piperead-sleep
  }
  for(i = 0; i < n; i++){  //DOC: piperead-copy
    if(pi->nread == pi->nwrite)
      break;
    ch = pi->data[pi->nread++ % PIPESIZE];
    if(copyout(pr->pagetable, addr + i, &ch, 1) == -1)
      break;
  }
  wakeup(&pi->nwrite);  //DOC: piperead-wakeup
  release(&pi->lock);
```

```
  return i;
}
```

- fork, exec와 마찬가지로 pipe()을 호출하는 thread가 main thread라는 보장이 없다고 생각했다.
- exec()을 호출한 thread가 main thread가 아니라면, 우선 `exec()을 호출한 thread의 main thread`를 찾았다.
- 이후, `main thread의 paget table`을 이용할 수 있도록 구현하였다.

15. exit

```
void
exit(int status)
{
  struct proc *p = myproc();
  struct proc *main = p->isThread ? p->main_thread : p;

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

  // Close all open files.
  for(int fd = 0; fd < NOFILE; fd++){
    if(p->ofile[fd]){
      struct file *f = p->ofile[fd];
      fileclose(f);
      p->ofile[fd] = 0;
    }
  }

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

  acquire(&wait_lock);
```

```
  // p: 현재 실행 중, main: 현재 실행 중인 thread의 main_thread
  if (p == main) {
    for (struct proc *t = proc; t < &proc[NPROC]; t++) {
      if (t == p) continue;

      acquire(&t->lock);
      if ((t->main_thread == main && t->isThread)) {
        t->killed = 1;
        if (t->state == SLEEPING) {
          t->state = RUNNABLE;
        }
      }
      release(&t->lock);
    }

    // Parent might be sleeping in wait().
    wakeup(p->parent);
  }
  else {
    // main_thread를 깨워야함
    wakeup(main);
  }
  // Give any children to init.
  reparent(p);

  acquire(&p->lock);

  p->xstate = status;
  p->state = ZOMBIE;

  release(&wait_lock);

  // Jump into the scheduler, never to return.
  sched();
  panic("zombie exit");
}
```

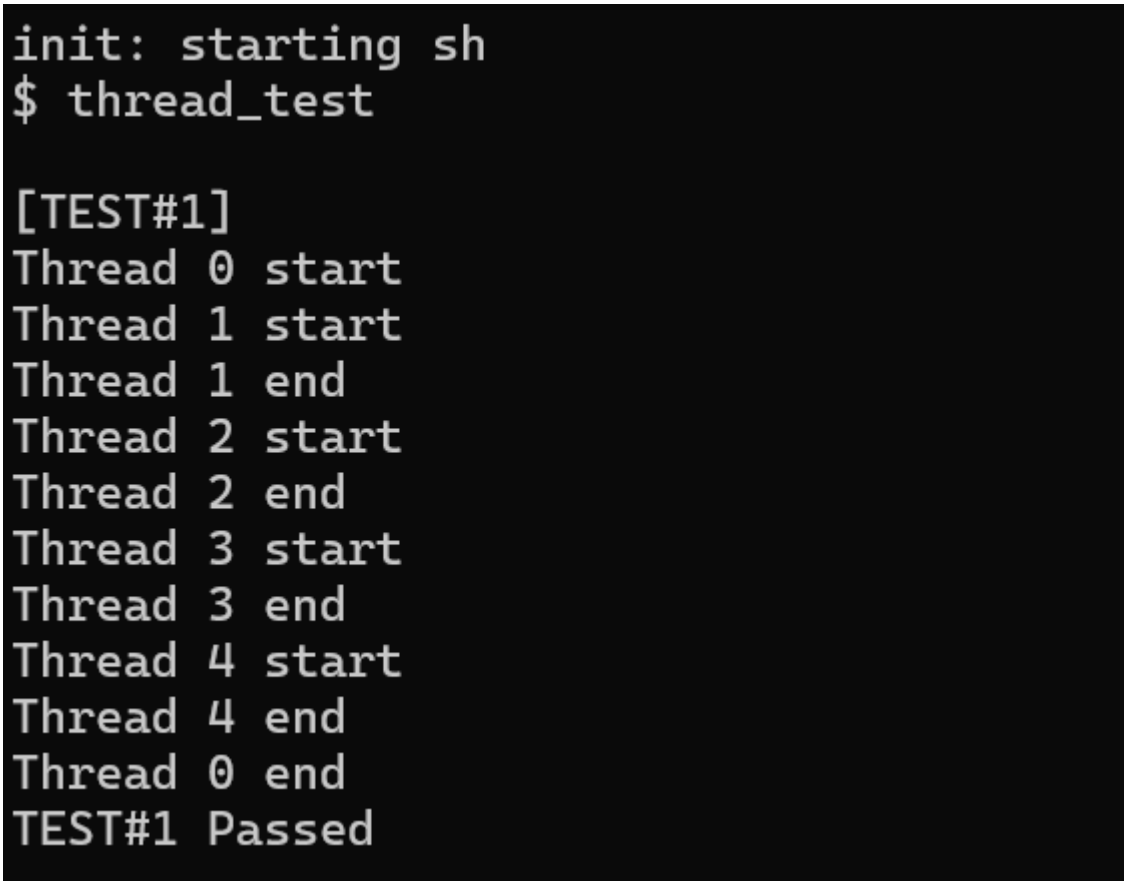
- fork, exec, pipe와 마찬가지로 kill()을 호출하는 thread가 main thread라는 보장이 없다고 생각했다.
- 우선 kill()을 호출한 thread의 main thread를 찾았다.
- `kill()을 호출한 thread가 main thread` 라면, thread의 child thread를 전부 kill하였다.
- `kill()을 호출한 thread가 main thread가 아니라면`, 현재 thread만을 kill하고 현재 thread의 main thread를 깨운다.

Results

Test 1부터 Test 6까지의 전체적인 실행 결과를 스크린샷 할 수가 없어, 각 Test 별로 나누어서 분석할 것이다.

다만, Test1부터 Test 6까지는 모두 끊김 없이 정상적으로 잘 실행되었다.

Test 1



- `Test 1`은 `Thread API`가 정상적으로 동작하는 지와 `thread간의 memory 공유`가 정상적으로 동작하는지 확인한다.
- 명세서의 결과와 동일한 결과를 얻은 것을 보아, thread API가 정상적으로 동작하고 thread간의 memory 공유도 정상적으로 동작하는 것을 확인할 수 있다.
- 명세서대로, 가장 마지막에 `thread 0`이 종료되는 것을 확인할 수 있다.

Test 2

```
[TEST#2]
Thread 0 start, iter=0
Thread 0 end
Thread 1 start, iter=1000
Thread 1 end
Thread 2 start, iter=2000
Thread 2 end
Thread 3 start, iter=3000
Thread 3 end
Thread 4 start, iter=4000
Thread 4 end
TEST#2 Passed
```

- `Test 2`는 각 thread가 정확히 `2개의 argument`를 정확히 넘겨 받고, `shared resource에 write`하는지 확인한다.
- 명세서의 결과와 동일한 결과를 얻은 것을 보아, 각 thread가 `shared resource`에 write하고 있는 것을 확인할 수 있다.
- `iter`를 통해 expected global array에 대한 접근 thread간의 충돌 없이 이루어지는 것을 확인할 수 있다.

Test 3

```
[TEST#3]
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Child of thread 0 start
Child of thread 1 start
Child of thread 2 start
Child of thread 3 start
Child of thread 4 start
Child of thread 0 end
Child of thread 1 end
Child of thread 2 end
Child of thread 3 end
Child of thread 4 end
Thread 0 end
Thread 1 end
Thread 2 end
Thread 3 end
Thread 4 end
TEST#3 Passed
```

- `Test 3`는 각 thread가 `fork()`을 호출한 이후, child thread (process)가 parent와 독립된 address space에서 정상적으로 동작하는지 확인한다.
- 명세서의 결과와 동일한 결과를 얻은 것을 보아, thread에서 parent와 child의 `address space`의 분리가 정확하게 이루어졌음을 확인할 수 있다.

Test 4

```
[TEST#4]
Thread 0 sbrk: old break = 0x0000000000015000
Thread 0 sbrk: increased break by 14000
new break = 0x0000000000029010
Thread 1 size = 0x0000000000029010
Thread 2 size = 0x0000000000029010
Thread 3 size = 0x0000000000029010
Thread 4 size = 0x0000000000029010
Thread 0 sbrk: free memory
Thread 0 end
Thread 1 end
Thread 2 end
Thread 3 end
Thread 4 end
TEST#4 Passed
```

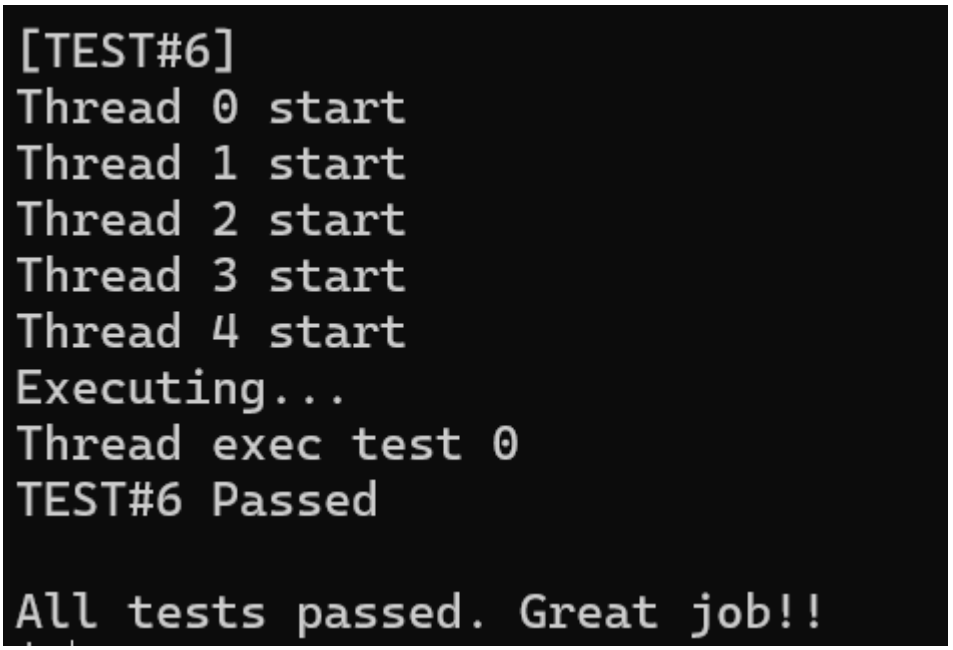
- `Test 4`는 thread가 `sbrk()`를 정확히 호출하고, 요청한 memory에 문제 없이 접근하는지 확인한다.
- 명세서의 결과와 동일한 결과를 얻은 것을 보아, 각 thread가 개인적인 memory를 가지고, 다른 thread와 겹치지 않는 주소를 가지고 있음을 확인할 수 있다.

Test 5

```
[TEST#5]
Thread 0 start, pid 29
Thread 1 start, pid 29
Thread 2 start, pid 29
Thread 3 start, pid 29
Thread 4 start, pid 29
Thread 0 end
TEST#5 Passed
```

- `Test 5`는 `kill()` 호출 시, thread 종료가 정상적으로 이루어지는지 확인한다.
- 명세서의 결과와 동일한 결과를 얻은 것을 보아, kill() 호출이 정상적으로 종료되는 것을 확인할 수 있다.
- Test code를 확인하면, main thread인 thread 0이 종료된 후, 나머지 thread도 모두 종료된다.
    - 이를 통해, main thread가 kill 되었을 때, 다른 thread kill되는 것을 확인할 수 있다.

Test 6



- `Test 6`은 `exec system call`이 thread를 지원하는지 확인한다.
- "Thread exec test 0" 메세지를 통해 Thread 0이 exec을 호출한 후, 나머지 thread는 종료되고 Thread 0에 `새로운 program (thread_fcn)`이 실행된 것을 확인할 수 있다.

Trouble Shooting

사전 `DESIGN`에서 발생한 문제

처음에는 기존 proc 구조체를 수정하여 `새로운 thread 구조체`를 정의하고, 하나의 process가 여러 thread를 table로 가지고, 관리할 수 있도록 구현하고자 하였다.

Process 구조체를 유지하며, Process 구조체를 거쳐서 thread를 관리, 동작하게 만들면 thread를 지원하도록 일부 system call만 수정하면 될 것 같다고 생각했다.

하지만 이 방법으로 구현하고자 하니, 기존에 process를 지원하던 거의 모든 코드를 thread까지 지원하게 수정이 필요해 보였고, 예상치 못한 부분에서 `panic (acquire, kerneltrap)` 이 계속 발생했다.

구현 과정에서 발생한 문제

1. `clone` 에서 아래 코드를 추가하였더니 `panic: remap` 이 발생했다.
    - `Preview` 에서 확인한 것처럼 user stack에 대해서도 `mappages` 가 필요하다고 판단하여 clone()에 아래 코드를 추가하였다.

    ```
    mappages(nt→pagetable, (uint64)stack, 2*PGSIZE, (uint64)kstack, PTE_W
    ```

    - 위 코드를 `clone()` 에서 제거하니 remap이 발생하지 않았다.
    - 찾아보니, user level에서 호출한 `malloc()` 함수에서 내부적으로 page table에 대한 매핑까지 수행한다고 한다.

2. `Test 3` 에서 `thread 0 ~ 4 lost their child` 라는 메세지가 출력 되었다.
    - parent 설정의 문제라고 판단해서 관련된 코드를 살펴보았다.
    - 디버깅을 통해 알아보니 `thread 0 ~ 4` 의 parent가 pid = 4 ~ 8인 thread가 아니라 thread_test (pid = 3)으로 설정되어 있었다.
    - 기존에는 main thread가 아닌 경우에는 parent를 따로 설정하지 않았는데, `일반 thread는 parent가 main_thread` 를 가리키도록 구현하니 해결되었다.

3. `Test 6` 의 결과가 명세서와 달랐다.
    - `thread 0 ~ 4 end` 라는 메세지가 전부 출력 되는 것을 보니 `exec()` 구현이 잘못 되었다고 생각해서 exec() 안에 디버깅 문구를 작성했다.
    - exec()이 정상적으로 호출되었다면 반드시 출력 되어야 할 디버깅 문구가 출력 되지 않는 것을 확인했다.
    - 디버깅을 통해 `sys_exec()` 에서 `fetchaddr()` 호출에서 문제가 발생함을 확인할 수 있었다.

- `fetchaddr()`를 확인해보니 sz값을 사용하는데, 내 구현에서는 sz는 반드시 main thread의 값을 사용하도록 했기 때문에 main thread가 thread가 `fetchaddr()` 를 호출하면 문제가 발생함을 인지하였다.
- 따라서 `fetchaddr()` 와 `fetchstr()`에서 sz값을 반드시 main thread의 값을 사용할 수 있도록 아래와 같이 수정하였다.

```
int
fetchaddr(uint64 addr, uint64 *ip)
{
  struct proc *p = myproc();
  struct proc *main = p->isThread ? p->main_thread : p;
  if(addr >= main->sz || addr+sizeof(uint64) > main->sz) // both tests need
    return -1;
  if(copyin(main->pagetable, (char *)ip, addr, sizeof(*ip)) != 0)
    return -1;
  return 0;
}

// Fetch the nul-terminated string at addr from the current process.
// Returns length of string, not including nul, or -1 for error.
int
fetchstr(uint64 addr, char *buf, int max)
{
  struct proc *p = myproc();
  struct proc *main = p->isThread ? p->main_thread : p;
  if(copyinstr(main->pagetable, buf, addr, max) < 0)
    return -1;
  return strlen(buf);
}
```

4. 세 번째 문제를 해결하고 나니 `exec()`은 정상적으로 호출되는데, `panic: freewalk: leaf` 가 발생했다.
    - 위 panic 메세지가 `pagetable` 관련 문제라는 것을 이전에 알게 되어 page table 관련 부분을 확인했다.
    - page table 관련 코드는 기존 `exec()`의 코드를 그대로 사용하고 있는데 무엇이 문제일지 고민하던 도중 `exec()`을 호출한 thread가 main thread가 아니라면 `old`

`page table을 할당 해제` 하면 문제가 발생한다는 것을 인지하였다.

- 따라서 `exec()` 에 `page table 해제 권한을 확인하기 위한 변수` 하나를 추가하여 해결하였다.

```
int i, off, pagetable_changed_allowed = 1;
...
if (p != main) {
    pagetable_changed_allowed = 0;
}
...

if (pagetable_changed_allowed) proc_freepagetable(oldpagetable, oldsz);
```