Operating System <Project02>

2023017856 한진호

1. Design, Implementation

(1) proc.h (struct proc)

```
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

  int is_thread;               // main process = 0, thread = 1
  struct proc *mythread[NPROC]; // thread list
  uint64 ustack;

  // 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)
};
```

이 proc 가 thread 인가 아닌가를 판단하기 위해서 is_thread 를 추가하였다.

해당 proc 가 clone 을 부른 proc 라면 clone 으로 만들어진 thread 들이 존재할 것이다. 이 thread 들을 관리하기 위해서 mythread 라는 proc 배열을 선언해 clone 때 담아주었다.

clone 때 parameter 로 받은 thread 의 stack 공간인 stack 을 담아주고자 ustack 을 만들었다. 이후 join 에서 free 하기 위해 stack 에 thread 의 stack 공간 주소를 받아가는데 이때 ustack 을 통해 값을 넘겨주고자 이를 추가하였다.

(2) int clone (void(fcn)(void*, void*), void *arg1, void *arg2, void *stack);

이번 project02 과제에서는 xv6 risc-v가 thread를 지원하지 않기 때문에 PCB를 통해 thread를 구현하였다.

thread 지만 구조 자체는 process 이기 때문에 clone 의 flow 를 fork 에서 참고하여 작성하였다.

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

  for(th = proc; th < &proc[NPROC]; th++) {
    acquire(&th->lock);
    if(th->state == UNUSED) {
      th->pid = allocpid();
      th->state = USED;
      memset(&th->context, 0, sizeof(th->context));
      th->context.ra = (uint64)forkret;
      th->context.sp = th->kstack + PGSIZE;
      break;
    }
    else {
      release(&th->lock);
    }
  }

  th->pagetable = p->pagetable;
  th->sz = p->sz;
```

```
th->trapframe = kalloc();
if(th->trapframe == 0){
  freeproc(th);
  th->state = UNUSED;
  release(&th->lock);
  return -1;
}
memmove(th->trapframe, p->trapframe, sizeof(struct trapframe));

th->trapframe->epc = (uint64)fcn;
th->trapframe->a0 = (uint64)arg1;
th->trapframe->a1 = (uint64)arg2;
th->trapframe->sp = (uint64)stack + PGSIZE;

th->trapframe->kernel_sp = th->kstack + PGSIZE;

th->trapframe_va = TRAPFRAME - PGSIZE * th->pid;
th->ustack = (uint64)stack;

if(mappages(th->pagetable, th->trapframe_va, PGSIZE, (uint64)(th->trapfram
  return -1;

for (int i = 0; i < NOFILE; i++) {
  if (p->ofile[i])
    th->ofile[i] = filedup(p->ofile[i]);
}
th->cwd = idup(p->cwd);

safestrcpy(th->name, p->name, sizeof(p->name));
th->is_thread = 1;

acquire(&wait_lock);
th->parent = p;
p->mythread[th->pid] = th;
release(&wait_lock);

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

```
  return th→pid;
}
```

fork 에서는 프로세스를 allocproc() 함수를 통해 만든다. 하지만 allocproc() 를 관찰해보면 이 안에서 proc_pagetable(p) 를 통해 자신만의 고유한 pagetable을 만든다. thread 는 자신만의 고유한 pagetable을 가지는 상황이 아니라 thread 자신을 만든 parent와 pagetable을 공유하여야 한다. 따라서 allocproc() 로 처리해 thread 의 틀을 만든다면 사용하지 않는 pagetable 을 만든 것이기에 memory leak 가 발생한다고 판단하였다. 이러한 이유로 allocproc로 처리하는 것이 아닌 allocproc 의 flow 를 따라가되 고유한 pagetable을 만들지 않음으로써 memory leak를 피하였다.

th→pagetable = p→pagetable; 을 통해 p (clone을 invoke한 process) 와 pagetable을 공유함으로써 thread 의 특성을 살리고자 하였다.

clone 은 arg1, arg2 이라는 parameter 값을 지닌 상태로 fcn 위치부터 실행하기를 기대하고 동작하는 함수이다. 따라서 trapframe→a0, trapframe→a1 위치에 각 parameter 를 담았다.

clone 이 system call 이기에 kernel에서 실행된 이후 trapframe 값을 통해 user로 돌아갈 것이다.

이때 돌아가는 위치는 trapframe→epc 이기 때문에 이 위치에 fcn (내가 thread를 통해 실행하고자 하는 function의 address) 를 담음으로써 clone 을 의도대로 동작할 수 있도록 상태를 저장하였다.

이번 project02 에서는 기존 project01 과 달리 struct proc 내부에 uint64 trapframe_va 라는 변수가 추가되었다. 이는 각 thread가 같은 trapframe 을 가지게 된다면 서로 epc, register 등의 상태가 같아져 버리기에 thread 가 독립적으로 서로의 일을 처리할 수 없을 것이다. 따라서 서로 다른 trapframe을 가져야 하기 때문에 이를 구현하고자 추가된 변수라고 판단하였다.

```
if(mappages(pagetable, p→trapframe_va = TRAPFRAME, PGSIZE,
            (uint64)(p→trapframe), PTE_R | PTE_W) < 0){
   uvmunmap(pagetable, TRAMPOLINE, 1, 0);
   uvmfree(pagetable, 0);
   return 0;
 }
```

기존 fork에 trapframe_va 를 TRAPFRAME 위치로 mapping 한 코드가 추가된 것을 볼 수 있다.

이와 같이 process 의 경우 trapframe의 virtual address 가 TRAPFRAME 으로 mapping 하면 문제가 없었지만 thread 는 이와 같이 코드를 작성해버린다면 서로 같은 pagetable을 사용하기에 trapframe 이 mapping 되는 위치가 전부 같은 위치 (해당 pagetable 에서의 TRAPFRAME 위치로 mapping 했으므로) 이므로 문제가 발생한다.

위 문제를 피하고자 trapframe_va 를 TRAPFRAME 으로 설정하는 것이 아니라 TRAPFRAME - th→pid * PGSIZE 로 설정함으로써 각 thread 끼리 겹치지 않도록 trapframe의 영역을 정해주었다.

그 후 mappages(th→pagetable, th→trapframe_va, PGSIZE, (uint64) (th→trapframe), PTE_R | PTE_W) 를 통해 th→trapframe 를 TRAPFRAME - pid * PGSIZE 위치로 mapping 시켜주었다.

```
int thread_create(void (*start_routine)(void*, void*), void *arg1, void *arg2) {
    //Page Size의 2배만큼 할당함.
    //내가 지금 할당받은 범위 내에는 사용하고 있는 addr이 없다는 것이 분명하므로
    //이 범위 내에서 offset bit를 0으로 밀어서 stack이 다른 데이터는 안들고 있고
    //page의 시작 부분을 가리키도록 설정한다.

    void *tmp = sbrk(PGSIZE*2);
    void *stack = (void*)(((uint64)tmp + PGSIZE - 1) & ~(PGSIZE - 1));
    //malloc 으로 공간을 할당하긴 하였으나 실제로 쓰지 않았을 때
    //실제 주소로 할당이 안되어 있을 수 있기에 실제 페이지가 매핑되도록 유도
    memset(stack, 0, PGSIZE);

    if (stack == 0 || (uint64)stack % PGSIZE != 0) {
        free(tmp);
        return -1;
    }
    return clone(start_routine, arg1, arg2, stack);
}
```

처음에는 void *stack = malloc(PGSIZE); 와 같이 PGSIZE (page 의 크기 - 4096 Byte) 만큼 memory를 할당하여 thread의 stack 공간을 마련하려고 하였다. 하지만 직접 코드를 돌려보았을 때 stack 의 주소값이 page size 의 배수로 나타나지 않을 수 있음 (offset 이 0이 아닌 상황. 즉, page-aligned 가 아닌 상황) 을 알게 되었다. 이를 해결하기

위해서 tmp 라는 임시로 memory를 할당받고자 하는 변수를 만들어 2page 만큼 memory 를 할당하였다. 이렇게 2page 를 할당 받는다면 내가 지금 할당받은 범위 내에는 사용하고 있는 address가 없다는 것이 분명하므로 이 범위 내에서 (uint64)tmp + PGSIZE - 1) & ~(PGSIZE - 1) 로 offset bit를 0으로 밀어서 stack이 page의 시작 부분 (offse을 0으로 만듦으로써 page-aligned) 을 가리키도록 설정하였다.

(3) int join (void **stack);

join은 child thread 가 terminate 될 때까지 기다리는 함수이다. 이는 process 차원에서 wait 와 매우 흡사하므로 wait 의 구조에서 thread 에게 적용될 수 있도록 약간의 수정을 통해 join을 design 하였다.

```
int
join(void **stack) {
  struct proc *th;
  int havekids, pid;
  struct proc *p = myproc();

  acquire(&wait_lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(th = proc; th < &proc[NPROC]; th++){
      if(th->parent == p && th->is_thread){
        // make sure the child isn't still in exit() or swtch().
        acquire(&th->lock);
        havekids = 1;
        if(th->state == ZOMBIE){
          // Found one.
          pid = th->pid;
          if(!stack || copyout(p->pagetable, (uint64)stack, (char *)&th->ustack, siz
            freeproc(th);
            release(&th->lock);
            release(&wait_lock);
            return -1;
          }
          freeproc(th);
```

```
        release(&th->lock);
        release(&wait_lock);
        return pid;
      }
      release(&th->lock);
    }
  }
  // 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.
  sleep(p, &wait_lock);  //DOC: wait-sleep
 }
}
```

join은 thread 차원의 wait 라 생각하여 내가 proc[NPROC] 를 순회하며 관심을 가지고 볼 proc (thread) 는 parent 가 join 을 invoke한 프로세스이고, thread 인 얘들이다. 이를 if (th->parent == p && th->is_thread) 로 걸러내었다. wait 에서 parameter 로 받은 addr 변수에 xstate 내용을 받아오는 것과 유사하게 parameter 로 받은 stack 변수에 thread의 stack 공간 주소를 받아왔다.

```
int thread_join() {
  void **stack = malloc(sizeof(void*));

  int tid = join(stack);

  if (tid < 0)
    return -1;

  if (stack && *stack) {
    free(*stack);
  }
  return tid;
}
```

stack 은 void* type 의 값을 담아올 변수이기에 void* size 만큼 공간을 할당하였다. 그 후 join(stack) 으로 stack 에 thread의 stack 공간 주소를 받아오고, 이를 free(*stack); 으로 deallocate 하여 memory leak를 피하고자 하였다. (*stack 의 값이 thread의 stack 공간 주소이므로 이를 free)

```
static void
freeproc(struct proc *p)
{
  if(p->is_thread == 0 && p->trapframe)
    kfree((void*)p->trapframe);
  p->trapframe = 0;
  if (p->is_thread && p->trapframe_va)
    uvmunmap(p->pagetable, p->trapframe_va, 1, 1);
  p->trapframe_va = 0;
  if(p->is_thread == 0 && p->pagetable)
    proc_freepagetable(p->pagetable, p->sz);
  p->pagetable = 0;
  p->sz = 0;
  p->parent->mythread[p->pid] = 0;
  p->pid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->chan = 0;
  p->killed = 0;
  p->xstate = 0;
  p->ustack = 0;
  p->is_thread = 0;
  for (int i = 0; i < NPROC; i++) {
    p->mythread[i] = 0;
  }
  p->state = UNUSED;
}
```

위와 같이 join 을 design 하고 실행해보면 문제가 발생함을 볼 수 있다. 문제를 피하기 위해 여러 수정을 거쳐보던 중 기존 freeproc 은 내 thread 에 대해서 문제가 생길 수 있는 부분을 여럿 발견하였다.

1. trapframe_va
   기존 freeproc은 trapframe_va 에 대해 값을 0으로 할당하는 동작밖에 없다.
   process에 대해서는 trapframe_va에 대해 문제가 생기지 않을 것이다. fork에서 trapframe_va 를 TRAPFRAME 위치로 mapping 하였기에 proc_freepagetable 에서 uvmunmap(pagetable, TRAMPOLINE, 1, 0); 으로 pagetable에서 unmap 후 uvmfree 에서 적절하게 지우기에 문제가 발생하지 않는다.
   하지만 내가 만든 thread는 TRAPFRAME으로 mapping 하지 않았다. 이 때문에 trapframe_va 에 대해서 panic: freewalk: 문제가 발생하였다.
   어차피 우리는 process 에 대해서는 앞서 말한 이유로 문제가 생기지 않음을 알고 있다. 그래서 나는 thread에 대해서만 따로 trapframe_va 를 uvmunmap 함으로서 control 하였다.
   (uvmunmap 에서 do_free = 1 로 설정하여 kfree 까지 진행하였기에 if (p→trapframe) kfree((void*) p→trapframe) 까지 해버리면 같은 위치를 2번 kfree 하고자 하는 것이므로 문제가 발생하였었다. 이를 피하기 위해서 이 부분에도 thread인지 확인 후 process 에 대해서만 해당 kfree 를 진행하도록 control 하였다.)
2. pagetable
   기존 freeproc 는 pagetable이 존재하면 proc_freepagetable을 호출함으로서 pagetable을 정리하였다.
   process는 서로 개별의 pagetable 을 갖기에 이에 대해 아무런 문제가 발생하지 않는다.
   이제 thread에 대해서 생각해보자. 여러 thread 가 아직 동작하고 있고 어떤 한 thread는 종료되어 freeproc에서 proc_freepagetable을 실행할 것이다. 이 thread가 서로 공유하고 있던 pagetable을 지워버림으로서 남은 thread는 갑자기 pagetable이 사라져버려 아무것도 할 수 없는 상태가 되어버린다.
   이러한 문제를 thread가 아닌 process에 대해서만 proc_freepagetable 을 할 수 있게 함으로서 피하였다.

(4) sbrk

```
int
growproc(int n)
{
  uint64 sz;
  struct proc *p = myproc();

  acquire(&wait_lock);
  if (p→is_thread)
```

```
    sz = p->parent->sz;
  else
    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;
  if(p->is_thread)
    p->parent->sz = sz;
  release(&wait_lock);
  return 0;
}
```

test 4 는 thread 에서 sbrk 가 잘 되는가를 보는 test이다. sbrk 를 고치기 위해 flow 를 따라가다보면 growproc 을 통해 memory allocate 부분이 진행되기에 이 부분을 수정하였다.

수정 이전에는 thread 별로 sz가 공유되어 있지 않고 개별적인 sz를 지니고 있었다. 이 때문에 thread 중 하나가 sbrk 를 꽤나 큰 크기로 진행한다면 다른 thread 의 공간을 침범하게 된다. (예를 들어 thread1 이 growproc 에서 uvmalloc으로 pagetable에 할당한 위치가 thread2 의 sz 위치를 넘어버린 상황을 생각해보자. 그 후 thread2 가 growproc 을 수행할 때 sz 위치부터 uvmalloc 할텐데 이 위치는 이미 thread1 이 사용해버렸으므로 이에 대해 panic: mappages: remap 이 발생하게 된다.)

process 에 대해서는 어차피 독립적으로 pagetable 을 가지고 있기 때문에 다른 얘가 해당 위치에 uvmalloc 해버리는 일이 없기 때문에 문제가 생기지 않는다. 그래서 나는 thread 인지 확인 후 별도로 control 해주었다.

지금 문제는 서로 sz가 공유되지 않기에 다른 thread 가 어디 위치까지 썼는지 모르기 때문에 생기는 문제이다. 이를 해결하기 위해 clone 을 호출하여 thread 를 만든 프로세스 (p->parent 에 해당하는 process) 의 sz 를 기준으로 삼고자 하였다. p->parent 의 sz에 growproc 로 인한 변경 사항을 계속 update 해주고 thread 라면 이 sz 를 받아와 사용하면 서로 sz를 공유하는 것과 같은 효과라 판단하여 해당 logic 으로 growproc 를 수정해주었고 문제를 해결하였다.

추가적으로 sz 를 각 thread 가 공유하는 변수로 만들었기에 이 sz 가 변경되는 코드는 critical section 이다. 여러 thread 가 동시에 sz 에 대해 접근하여 race condition 이 발

생할 수 있기 때문에 growproc 에 lock 을 걸어 synchronization 을 하였다.

(5) kill

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

  for(p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);
    if(p->pid == pid){
      struct proc *target = p->is_thread ? p->parent : p;
      for (int i = 0; i < NPROC; i++){
        if (target->mythread[i] && target->mythread[i]->pid != pid){
          target->mythread[i]->killed = 1;
          check++;
        }
      }
      if (p->is_thread){
        target->killed = 1;
        check++;
      }

      while (check > 0){
        release(&p->lock);
        check--;
        yield();
        acquire(&p->lock);
      }
      p->killed = 1;
      if(p->state == SLEEPING){
        // Wake process from sleep().
        p->state = RUNNABLE;
      }
      release(&p->lock);
      return 0;
```

```
    }
    release(&p->lock);
  }
  return -1;
}
```

과제 명세서를 보면 kill 에 대해 한 thread 가 terminating 될 때 그 프로세스 내 모든 thread 가 terminate 되도록 하라고 적혀있다. 이 부분은 지금 내 thread 가 속해있는 프로세스 내 모든 thread 에 대해 접근해 killed = 1 로 설정해버리면 구현이 될 것이다.

여기서 크게 2가지 case로 나뉘게 된다.
case 1 - kill 을 호출한 얘가 process (thread 를 만들었던 process 를 뜻한다.)

case 2 - kill 을 호출한 얘가 thread

case 1에 대해서는 자신의 mythread 배열, case 2에 대해서는 자신의 parent 의 mythread 배열 + 자신의 parent를 찾아보면 속해있는 모든 thread 를 찾아낼 수 있다. 이를 깔끔하게 구현하기 위해 target 이라는 변수를 만들어 p→is_thread 가 1이면 parent, 0 이면 자신을 담았다.

그럼 이제 target 의 mythread 를 관찰하면 속해있는 thread 를 모두 찾을 수 있으니 for 문으로 순회하며 killed = 1 을 설정해주고 check 를 +1 하며 자신을 제외하고 속해있는 모든 thread의 수를 count 하였다.

(+ 여기서 자기 자신은 어차피 아래에서 killed = 1 을 해줄 것이므로 pid 를 따져 kill 을 호출한 thread 와 같다면 skip 해주었다.)

kill 은 killed = 1 로 설정함으로서 나중에 해당 프로세스가 종료되도록 flag를 설정해주는 것뿐이지 직접적으로 종료시키는 함수가 아니다. 종료되는 것은 timer interrupt 가 일어났을 때와 같은 상황에서 발생한다. 하지만 나는 kill 내에서 바로 다른 thread 들이 종료되는 것을 원했기에 check 만큼 yield() 를 진행하였다. procdump() 를 통해 프로세스의 상태를 관찰해보았을 때 내가 원하는 대로 잘 동작함을 확인할 수 있었다.

(6) exec

```
int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  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();
uint64 oldtrapframe_va = p→trapframe_va;

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->is_thread ? p->parent->sz : 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;

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

kill(p->pid);
p->killed = 0;

// Commit to the user image.
oldpagetable = p->pagetable;
p->pagetable = pagetable;
if (p->is_thread && oldtrapframe_va != TRAPFRAME){
  uvmunmap(oldpagetable, oldtrapframe_va, 1, 1);
  uvmunmap(p->pagetable, p->trapframe_va, 1, 1);
  p->trapframe_va = TRAPFRAME - PGSIZE * p->pid;
  if (mappages(p->pagetable, p->trapframe_va, PGSIZE, (uint64)(p->trapfram
    goto bad;
}
p->sz = sz;
// arguments to user main(argc, argv)
// argc is returned via the system call return
// value, which goes in a0.
p->trapframe->a1 = sp;
p->trapframe->epc = elf.entry;  // initial program counter = main
p->trapframe->sp = sp; // initial stack pointer

if (p->is_thread == 0)
  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;
}
```

exec 는 모든 thread 를 clean up 하고 새로운 프로세스를 시작하라고 과제 명세서에 적혀 있다. 나는 여기서 모든 thread 를 clean up 하고 프로세스를 새로 만들고 시작하는 것과 프로세스 내 모든 thread 중 exec 를 수행 중이던 thread 만 남기고 진행하는 것을 동치라 판단하여 exec 를 수행 중인 thread 만 남기고 나머지는 모두 종료 시키는 방향으로 exec 를 수정하였다.

위에서 kill 을 thread 라면 그 프로세스 내에 속한 다른 모든 thread 들에게 killed = 1 을 설정해주도록 수정하였다. 이를 사용하고자 하였다. kill(p→pid); 함으로써 다른 thread 들이 있는지 체크한 후 종료해주었다. (kill 에서 yield 를 통해 바로 thread 들이 종료되도록 하였으므로 kill 만으로도 충분하다.) kill(p→pid) 를 수행하고 나오면 현재 exec 를 수행하고 있는 thread 도 killed = 1 로 설정되어 있는 상태이다. 하지만 지금 이 thread 는 종료되면 안되기에 killed = 0 으로 설정함으로서 exec 를 수행 중인 thread 만 남기고 나머지는 모두 종료 시켜주었다.

exec 의 flow 를 살펴보면 마지막에 기존 pagetable 을 oldpagetable 이라는 변수에 담고, p→pagetable 위치에 새로 할당한 pagetable을 넣어주며 oldpagetable 을 proc_freepagetable 로 정리해준다. 하지만 여기서 thread 일 때 panic: freewalk: leaf 가 발생하였다.

1. p→sz
   나는 어떤 thread에 대해서 sz 값을 그 thread 가 growproc 을 실행하는 상황에서만 업데이트 되도록 구현하였었다. 이로 인해 growproc 을 실행하지 않았다면 그 thread 는 clone 으로 만들었을 당시의 sz 값을 계속 가지고 있는 상황이다.
   이게 exec 에서 문제가 되었다. exec 는 oldpagetable 을 정리할 때 oldsz 변수로 어느 범위까지 지울지를 결정하였는데 이 oldsz 는 p→sz 값으로 할당된다. 하지만 test 6 에서 보면 growproc 을 호출하는 일이 없다.

즉, oldsz 의 값이 p 라는 thread 가 clone 으로 만들어졌을 당시의 sz 값이다.
여기서 test 6과 같이 thread 를 총 5개 만들고, 첫 번째로 만들어진 thread 가 exec 를 실행하는 상황을 생각해보자. 그러면 이 oldsz 는 예전의 값이기에 뒤늦게 만들어진 나머지 thread 에 대한 정보는 sz 의 범위를 넘어가버렸으므로 지워지지 못한다. 이로 인해 panic: freewalk: leaf 가 발생한다.
이를 올바르게 수정하기 위해서는 oldsz 에 가장 최신의 sz 값이 들어갔어야 한다.
우리는 앞서 sz 값에 대해서는 thread 가 아닌 process 가 항상 최신의 sz 값을 가지도록 업데이트 해주었다. 따라서 oldsz 값에 thread 가 아닌 process 의 sz 를 할당하면 해결될 것이다.
이를 위해 oldsz = p→is_thread ? p→parent : p ; 로 수정함으로서 sz 값을 올바르게 control 하였다.

2. p→trapframe_va
   p (thread) 에 대해서 p→trapframe_va 를 TRAPFRAME - pid * PGSIZE 로 mappages 해주었다. 기존에 process 에 대해서는 proc_freepagetable 내에서 uvmunmap(pagetable, TRAPFRAME, 1, 0) 으로 control 해주었지만 thread 는 이 line 으로는 thread 의 trapframe 을 unmap 해주지 못한다. (trapframe 위치가 TRAPFRAME - pid * PGSIZE 이기 때문임.)
   따라서 이로 인해 panic: freewalk: leaf 가 발생하였다.
   이를 freeproc 때와 같이 thread 라면 uvmunmap 으로 직접 unmap 시키는 방식을 채택하였다.
   우선 exec 시작 부분에 oldtrapframe_va 라는 변수를 만들어 exec 호출 이전의 trapframe_va 값을 저장하였다. 그 후 pagetable 을 바꿔주는 타이밍에 oldtrapframe_va 를 uvmunmap 하여 unmap 해주었다. 또한 pagetable 을 만들 때 proc_pagetable 함수를 사용하였는데 여기서는 trapframe_va = TRAPFRAME 으로 하여 mappages 해준다. 하지만 thread 가 exec 하는 경우에는 TRAPFRAME 위치가 아닌 TRAPFRAME - pid * PGSIZE 위치에 자신의 trapframe 이 있다.
   그러므로 나는 uvmunmap 으로 TRAPFRAME 위치로 mapping 된 trapframe_va 를 날리고, 새롭게 TRAPFRAME - pid * PGSIZE 위치로 mapping 해주었다.
   이렇게 한 후 실행해보았을 때 완벽하게 수정된 줄 알았으나 프로그램이 끝나지 않고 무한정 대기하는 문제에 빠지게 되었다.
   그에 대한 이유는 내가 trapframe 에 대해 위와 같은 수정을 하기 이전에 p→trapframe→a0 = sp 와 같이 trapframe 을 수정한 작업이 있었기 때문이다. 다시 말하자면, exec 에서 정상적인 flow 로 흘러가도록 trapframe 을 수정해주었는데 내가 위와 같이 trapframe 을 날려버린 후 새롭게 mapping 해주었으니 trapframe 이 망가졌던 것이다. trapframe 이 망가진 상태이기에 exec 이후 제대로 흘러가지 않고 문제가 발생하였다.
   이를 어떻게 피할까 고민하던 중 trapframe 위치에 대한 수정을 다 마친 후에 값을 바

꿔준다면 trapframe 이 망가지는 문제를 피할 수 있다고 판단하여 위와 같이 수정 후 해당 test 에 대한 panic 문제를 해결하였다.

2. Results

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

```
[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#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#4]
Thread 0 sbrk: old break = 0x0000000000035000
Thread 0 sbrk: increased break by 14000
new break = 0x0000000000051010
Thread 1 size = 0x0000000000051010
Thread 2 size = 0x0000000000051010
Thread 3 size = 0x0000000000051010
Thread 4 size = 0x0000000000051010
Thread 0 sbrk: free memory
Thread 0 end
Thread 1 end
Thread 2 end
Thread 3 end
Thread 4 end
TEST#4 Passed


[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
TEST#5 Passed
```

```
[TEST#6]
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Executing...
Thread exec test 0
TEST#6 Passed

All tests passed. Great job!!
```

→ 결과가 Test 예시와 같이 잘 나오지만 Test 4 에 대해서는 로그를 봤을 때 문제가 있다고 판단되었다. 이에 대한 내용은 아래 Troubleshooting 에서 다루도록 하겠다.

3. Troubleshooting

test 4 에서 malloc(4096 * 4 * NUM_thread); 하면서 총 20 page 만큼의 공간을 할당하였다. 이는 로그를 통해서도 잘 할당되었음을 확인할 수 있다. (Thread 0 sbrk: increased break by 14000)

하지만 Thread 0 sbrk: old break = 0x0000000000035000, new break = 0x0000000000051010 는 Test 예시와 거리가 있는 것을 확인할 수 있다. 이에 대한 이유를 각각 설명하도록 하겠다.

1. Thread 0 sbrk: old break = 0x0000000000035000 관련 문제
   나는 thread_create 에서 stack 공간을 만들어줄 때 2 page 크기만큼씩 만들었다. 그래서 thread 를 만들 때마다 sz 가 2 page 크기만큼 커지는데, thread 가 종료될 때 2 page 크기만큼 sz 를 감소시키지 않았었기에 thread_create 가 호출될 때마다 sz 가 2 page 씩 누적되어 커져간다.
   처음 프로그램 실행하는 프로세스 (pid 3) 의 sz 가 86016 (= 0x15000) 이였다.
   test 1 이 끝나면서 test 1 에서 thread 를 5개 만들었으므로 10 page 만큼 더 커져서 sz 는 0x1F000, test 2 가 끝났을 때는 sz 가 0x29000, test 3 이 끝났을 때는 sz 가 0x33000 이 되었다.
   old break 의 값은 test 4 에서 처음으로 만들어지는 thread 의 sz 이다.
   thread 를 만들 때 2 page 씩 sz 가 커져가므로 결과적으로 0x35000 이 출력되게 되었다.

2. new break = 0x0000000000051010 관련 문제
   old break = 0x35000 이고 increased break 가 0x14000 라 로그에 뜨므로 우리가 기대하는 new break 는 0x49010 이였어야 했다. 하지만 기대하는 것과 다르게 0x51010 으로 8 page나 차이가 발생하였다. 어디서 8 page 나 공간이 더 할당되었을까 찾아보기 위해 growproc 에서 로그를 찍어보았다.

아래는 sbrk(0); 을 실행하는 순간 (old break) 에서 찍힌 로그이다.

24 thread sz: 217088 (0x35000) → growproc 시작 타이밍 + growproc 실행하는 thread sz
growproc sz: 249856 (0x3d000) → process 내에서 가장 최신의 sz 값
after growproc sz: 249856 (0x3d000) → growproc(0) 수행 후 sz 값

우리가 처음 old break 로 찍은 로그의 값은 첫 번째 thread 가 clone 된 당시의 sz 값이다. (24 thread sz: 217088 (0x35000)
하지만 growproc 에서 사용되는 sz 값은 thread 의 parent 가 가지고 있는 sz 값이다. 즉, thread 5개를 전부 만든 상황 이후의 sz 값이다. 내가 thread 를 만들 때마다 tmp 를 2* PGSIZE 만큼 sbrk 하였기에 thread 를 만들 때마다 sz는 0x2000 씩 커진다. 따라서 sbrk(0) 에서 사용되는 sz 값은 첫 번째 thread 의 sz 값에서 0x8000 (8 page) 이나 더 증가한 값이 사용된다. (growproc sz: 249856 (0x3d000))
growproc(0) 이후에는 thread 의 sz 값도 가장 최신으로 update 되기 때문에 여기서 첫 번째 thread 의 sz 값은 0x3d000 으로 변경되었다.
sbrk 의 내부를 보면 growproc 하기 이전의 sz 값을 return 하는 구조기에 Test 에서의 로그는 0x35000 를 출력하게 되었지만, 내부적으로 실제 sz 값은 0x3d000 이기 때문에 나중에 new break 시에 0x3d000 에서 0x14000 만큼 증가한 0x51010 이 출력되게 된 것이다.