Project02

1. Design

이번 프로젝트에서 구현해야 하는 것은 스레드 한 가지입니다. 스레드를 구현하기 위해서 clone과 join 함수를 만드는 것이 중요합니다.

1. clone
2. join

Common

함수를 설명하기 전에 스레드를 위해 proc 구조체에 여러 변수들을 추가해줘야 합니다. 메인 스레드인지 구분하기 위한 변수가 필요합니다. 또한 명세에서 유저 함수인 thread_join에서 user stack을 free해줘야 한다고 되어 있습니다. 그렇기에 나중에 유저 스택의 위치를 불러올 수 있도록 값을 저장합니다.

Diffrence

명세에서 기존 코드에 이미 변경 사항이 있다고 되어 있습니다. 변경 사항은 trapframe_va라는 변수가 proc 구조체에 추가된 것입니다.
기존 코드에서는 프로세스만이 존재했기에 하나의 pagetable에 하나의 trapframe만 존재합니다. 그렇기에 pagetable의 고정된 위치에 trapframe을 매핑합니다.
하지만 이번 과제에서는 스레드를 구현해야 합니다. 스레드는 서로 pagetable을 공유합니다. 그렇기에 기존 코드처럼 고정된 위치에 매핑을 할 경우 여러 스레드가 같은 곳에 매핑하는 문제가 발생합니다. 그렇기에 trapframe_va라는 변수를 두어 서로 다른 위치에 매핑을 하도록 한 것입니다.
proc_pagetable 함수에서는 프로세스가 처음 pagetable을 만들 때 매핑을 하는 것이므로 trapframe_va = TRAPFRAME으로 하여 고정된 값으로 매핑을 합니다.
이후 제가 스레드 생성을 구현할 때에는 스레드끼리 서로 다른 곳에 매핑을 하도록 해야 합니다.

clone

먼저 스레드를 생성하는 clone 함수가 필요합니다. 전체적인 코드는 fork와 유사하게 진행됩니다.

1. allocthread를 해줍니다.
2. pagetable을 main 스레드의 것을 공유하도록 합니다.
3. trapframe은 공유하지 않고 main의 것을 복사하여 사용합니다.
4. pagetable을 공유하기 때문에 trapframe을 스레드끼리 겹치지 않도록 trapframe_va 값을 설정한 후 매핑을 합니다.
5. file descriptor는 main의 것을 공유하도록 합니다.

join

clone으로 스레드를 생성했으면 스레드가 종료될 때 정리해주는 함수가 필요합니다. 전체적인 코드는 wait과 유사하게 진행됩니다.

1. proc 배열을 순회하며 main이 본인이며 스스로는 아니고 현재 state가 ZOMBIE인 스레드를 찾습니다.
2. 찾은 경우 user stack을 전달해줍니다.
3. 그 스레드를 정리해줍니다.
4. 만약 찾지 못한 경우 sleep을 하여 스레드를 기다립니다.

thread_create

clone을 호출하는 유저 함수입니다.

1. user stack에 사용하기 위해 malloc으로 공간을 할당합니다.
2. 그 stack을 인자로 clone을 호출합니다.

thread_join

join을 호출하는 유저 함수입니다.

1. stack 변수를 만든 뒤 join을 호출합니다.
2. join 함수에 얻은 user stack 값을 통해 stack을 free합니다.

2. Implementation

proc

스레드를 위해 proc 구조체에 변수를 추가했습니다.

```
//proc.h
struct proc {
  ...

  uint64 tid;
  struct proc *main;
  void *ustack;
  uint64 temp_va;
};

//proc.c
int nexttid = 1;
```

proc.h

스레드를 매핑할 때 trapframe_va를 다르게 설정하기 위해서 tid를 만들었습니다.
특정 스레드의 main을 확인하기 위해 main을 struct proc*으로 저장합니다.
user stack을 join에서 전달해주기 위해 ustack을 만들었습니다.
temp_va는 exec에서 필요해서 만들었습니다. 이후 exec 부분에서 자세히 설명하겠습니다.

proc.c

nextpid처럼 nexttid를 만들어 스레드가 생성될 때마다 1씩 증가 시켜 줍니다.

clone

clone을 구현하기 위해 먼저 allocthread 함수가 필요합니다.
allocproc과 유사하지만 스레드에 필요한 변수들을 할당하기 편하도록 함수를 구분했습니다.

allocthread를 설명하기 전에 allocproc의 변경점부터 설명하겠습니다.

allocproc

```
//proc.c
static struct proc*
allocproc(void)
{
  ...
```

```
found:
  p->pid = allocpid();
  p->state = USED;
  p->tid = 0;
  p->main = p;

  ...
}
```

allocproc으로 생성되는 것은 프로세스이기에 tid에 0을 넣고 main을 본인으로 지정합니다.

allocthread

```
//proc.c
static struct proc*
allocthread(struct proc *main_thread)
{
  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 = main_thread->pid;
  p->state = USED;
  p->tid = nexttid++;
  p->main = main_thread;

  if((p->trapframe = (struct trapframe *)kalloc()) == 0){
```

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

  memset(&p->context, 0, sizeof(p->context));
  p->context.ra = (uint64)forkret;
  p->context.sp = p->kstack + PGSIZE;

  return p;
}
```

allocproc과 다른 점은 found에서 변수 할당 부분과 pagetable 할당 부분입니다.

- pid는 main 스레드의 pid를 사용하고, tid는 nexttid++를 하여 스레드가 다른 값을 가지도록 합니다. 또한 main을 본인이 아닌 main 스레드로 해줍니다.
- pagetable의 경우 스레드 간에는 공유를 해야 하므로 proc_pagetable을 호출하지 않습니다.

clone

```
//proc.c
int
clone(void(*fcn)(void*,void*), void *arg1, void *arg2, void *stack) {
  struct proc *np;
  struct proc *curproc = myproc();
  struct proc *main_thread;

  main_thread = curproc->main;

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

  void *aligned = (void *)PGROUNDUP((uint64)stack);
  np->pagetable = main_thread->pagetable;
  np->sz = main_thread->sz;
  np->ustack = (void*)((uint64)stack);
```

```
  *(np->trapframe) = *(main_thread->trapframe);
  np->trapframe->epc = (uint64)fcn;
  np->trapframe->sp = (uint64)aligned + PGSIZE;
  np->trapframe->a0 = (uint64)arg1;
  np->trapframe->a1 = (uint64)arg2;

  np->trapframe_va = TRAPFRAME - (np->tid * PGSIZE);
  if(mappages(np->pagetable, np->trapframe_va, PGSIZE,
          (uint64)(np->trapframe), PTE_R | PTE_W) < 0){
   release(&np->lock);
   return -1;
  }

  for(int i = 0; i < NOFILE; i++)
   if(main_thread->ofile[i])
     np->ofile[i] = filedup(main_thread->ofile[i]);
  np->cwd = idup(main_thread->cwd);

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

  release(&np->lock);

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

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

  return np->pid;
 }
```

전체적인 흐름은 fork와 유사합니다.

1. 먼저 allocthread를 호출하여 기본적인 설정을 해줍니다.
2. user stack이 page-aligned되어야 한다 했으니 PGROUNDUP을 하여 align해줍니다.

3. pagetable은 main 스레드의 pagetable을 사용하고 sz도 main 스레드의 것을 사용합니다.
4. ustack에는 align되지 않은 stack을 저장합니다. align된 것을 저장하면 malloc을 했을 때의 값을 가지고 있지 않고 정렬된 이후의 값을 가지므로 완전한 free를 하지 못하게 됩니다. 그래서 인자로 받은 stack을 저장합니다.
5. trapframe은 main 스레드의 것을 복사하여 사용하고, epc에는 fcn, sp에 정렬된 stack, a0와 a1에 arg1, arg2를 넣습니다.
6. trapframe_va는 TRAPFRAME - (np→tid * PGSIZE)를 하여 다 다른 값을 가지도록 하고 pagetable에 매핑을 해줍니다.
7. file descriptor는 main 스레드의 것을 공유합니다.
8. clone한 스레드의 pid를 return합니다.

join

join을 구현하기 위해 먼저 freethread 함수가 필요합니다.
freeproc과 유사하지만 pagetable을 해제하면 안되는 등 여러 차이가 있기에 함수를 구분했습니다.

freethread를 설명하기 전에 freeproc의 변경점부터 설명하겠습니다.

freeproc

```
//proc.c
static void
freeproc(struct proc *p)
{
  if(p→trapframe)
    kfree((void*)p→trapframe);
  p→trapframe = 0;
  p→trapframe_va = 0;
  if(p == p→main) {
    struct proc *pp;
    for(pp = proc;pp < &proc[NPROC];pp++) {
      if (pp→trapframe_va != 0 && pp→main == p && pp != p) {
        freethread(pp);
      }
    }
    proc_freepagetable(p→pagetable, p→sz);
```

```
  }
  p->tid = 0;
  p->main = 0;

  ...
}
```

먼저 trapframe_va, tid와 main을 초기화해줍니다.
그리고 명세에 나와있듯 main 스레드가 먼저 exit을 호출할 경우 같은 프로세스 내의 모든 스레드를 종료해야 합니다. main 스레드는 설계 상 exit을 할 경우 wait에서 부모 프로세스에게 잡혀 freeproc만을 호출합니다. 그렇기에 freeproc을 호출한 상태일 때 다른 스레드가 남아있는 경우 모두 종료 시켜주면 되는 것이므로 proc 배열을 순회하면서 같은 프로세스 내의 모든 스레드를 freethread해줍니다.

freethread

```
//proc.c
static void
freethread(struct proc *t)
{
  if(t->trapframe)
    kfree((void*)t->trapframe);
  t->trapframe = 0;
  if (t->trapframe_va != 0) {
    uvmunmap(t->pagetable, t->trapframe_va, 1, 0);
  }
  t->trapframe_va = 0;
  t->sz = 0;
  t->pid = 0;
  t->tid = 0;
  t->parent = 0;
  t->main = 0;
  t->ustack = 0;
  t->name[0] = 0;
  t->chan = 0;
  t->killed = 0;
  t->xstate = 0;
```

```
  t->state = UNUSED;
}
```

freeproc과 다른 점은 proc_freepagetable을 호출하지 않는 것입니다.
스레드는 pagetable을 서로 공유하고 있습니다. 그래서 다른 스레드가 현재 pagetable을 사용하고 있을 수도 있기 때문에 pagetable을 free해서는 안됩니다. 대신 pagetable에서 unmap은 해줘야 하므로 uvmunmap을 추가합니다.

join

```
//proc.c
int
join(void **stack)
{
  struct proc *p = myproc();
  struct proc *t;
  int havekids, pid;
  acquire(&wait_lock);
  for (;;) {
    havekids = 0;
    for (t = proc; t < &proc[NPROC]; t++) {
      if(t->main == p && t != p) {
        acquire(&t->lock);
        havekids = 1;
        if (t->state == ZOMBIE) {
          pid = t->pid;
          if(copyout(p->pagetable, (uint64)stack, (char*)&t->ustack,sizeof(void*))
            release(&t->lock);
            release(&wait_lock);
            return -1;
          }
          freethread(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);
  }
}
```

전체적인 흐름은 wait과 유사합니다.

1. proc 배열을 순회하며 main이 본인이며 스스로는 아니고 현재 state가 ZOMBIE인 스레드를 찾습니다.
2. 찾은 경우 user stack을 copyout을 하여 전달해줍니다.
3. 그 스레드를 freethread해줍니다.
4. free해준 스레드의 pid를 return합니다.
5. 만약 찾지 못한 경우 sleep을 하여 스레드를 기다립니다.

thread.c

앞에서 만든 clone과 join을 이용하는 유저 함수가 필요합니다.

thread_create

```
//thread.c
int
thread_create(void (*start_routine)(void *, void *), void *arg1, void *arg2)
{
  void *stack = malloc(PGSIZE*2);
  if(stack == 0)
    return -1;
  int pid = clone(start_routine, arg1, arg2, stack);
  if(pid < 0) {
    free(stack);
    return -1;
  }
```

```
  return pid;
}
```

1. 먼저 stack에 malloc으로 PGSIZE*2만큼 할당합니다.
2. thread_create의 인자로 받은 값과 stack으로 clone을 호출합니다. 만약 pid가 음수인 경우 stack을 free해줍니다.
3. 성공적으로 clone 된 경우 pid를 반환합니다.

thread_join

```
//thread.c
int
thread_join(void)
{
  void *stack;
  int pid = join(&stack);
  if(pid < 0)
    return -1;
  free(stack);
  return pid;
}
```

1. stack 변수를 만들어둡니다.
2. join을 호출하여 stack에 user stack의 값을 저장합니다.
3. 성공적으로 join이 된 경우 stack을 free하고 pid를 반환합니다.

System calls

아래는 명세에서 요구한 system call의 구현입니다. 명세에서 요구한 것 외에도 추가적으로 수정한 함수가 있으면 포함되어 있습니다.

exit

```
//proc.c
void
exit(int status)
{
```

```
  struct proc *p = myproc();
  if(p == initproc)
    panic("init exiting");
  if(p->tid == 0) {
    struct proc *pp;
    for(pp = proc; pp < &proc[NPROC]; pp++) {
      acquire(&pp->lock);
      if(pp->main == p && pp != p) {
        pp->killed = 1;
        if(pp->state == SLEEPING) {
          pp->state = RUNNABLE;
        }
      }
      release(&pp->lock);
    }
  }

  ...

  if(p->tid == 0 || p->killed == 1) {
    wakeup(p->parent);
  }
  else {
    wakeup(p->main);
  }

  ...
}
```

main 스레드가 exit을 호출하면 모든 스레드가 종료되고, 다른 스레드가 exit을 호출하면 그 스레드만 종료되는 것을 구현하기 위해 exit을 수정했습니다. 만약 exit을 호출한 것이 main인 경우 proc 배열을 순회하면서 자신을 main으로 갖는 모든 스레드를 kill해줍니다. kill을 하지 않더라도 main 스레드가 종료되고 wait에서 freeproc될 때 모든 스레드가 freethread가 되긴 하지만 wait에서 수거되기 전에 다른 스레드가 먼저 실행될 수 있기에 kill하여 실행을 막습니다.
그리고 main 스레드가 아닌 다른 스레드가 exit을 호출했을 때 join으로 수거가 되야 합니다. join을 호출한 main 스레드는 sleep 상태이므로 main을 wakeup해줍니다.
또한 만약 특정 스레드가 kill된 경우 그 프로세스에 속하는 모든 스레드가 kill된 것입니다.

이때는 main 스레드가 join을 해주지 못하므로 wait을 통해 모든 스레드가 free됩니다. 그렇기에 p→killd == 1을 통해 kill이 된 경우 parent를 wakeup해줍니다.

fork

fork는 수정 사항이 없습니다.
수정하지 않더라도 호출한 스레드를 부모로 지정하여 생성이 잘 됩니다.

exec

```
//exec.c
int
exec(char *path, char **argv)
{
  ...

  uint64 temp_va = p→trapframe_va;

  ...

  p→tid = -1;
  p→temp_va = temp_va;
  proc_freepagetable(oldpagetable, oldsz);
  p→main = p;
  p→tid = 0;
  p→temp_va = 0;
  return argc; // this ends up in a0, the first argument to main(argc, argv)

    ...
}
```

exec를 호출하면 본인을 제외한 모든 스레드를 제거하고 새로운 프로세스로 바뀌어야 합니다. 그렇기에 old pagetable을 free하기 전에 모든 스레드의 trapframe을 unmap하고 freethread해주면 됩니다.
저는 exec에서 free를 해주지 않았고 proc_freepagetable에서 했습니다.

1. 이때 proc_freepagetable은 freeproc에서도 사용되므로 exec를 호출했을 때와 구분하기 위해서 임시로 tid를 -1로 설정해 실행합니다.

2. 그리고 exec 실행 중에 호출한 스레드의 trapframe_va 값이 변합니다. 그렇기에 proc 구조체에 temp_va를 저장하여 저 값을 이용하여 unmap해줍니다.

```
//proc.c
void
proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
  struct proc *p = myproc();
  struct proc *pp;
  if(p->tid == -1) {
    for(pp = proc;pp < &proc[NPROC];pp++) {
      if(pp->pid == p->pid && pp != p && pp->tid != 0) {
        freethread(pp);
      }
      if(pp->pid == p->pid && pp != p && pp->tid == 0) {
        if(pp->trapframe)
          kfree((void*)pp->trapframe);
        pp->trapframe = 0;
        pp->trapframe_va = 0;
        pp->sz = 0;
        pp->pid = 0;
        pp->tid = 0;
        pp->parent = 0;
        pp->name[0] = 0;
        pp->chan = 0;
        pp->killed = 0;
        pp->xstate = 0;
        pp->state = UNUSED;
      }
    }
    if(p->temp_va != TRAPFRAME) {
      uvmunmap(pagetable, p->temp_va, 1, 0);
    }
  }
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  uvmfree(pagetable, sz);
}
```

3. proc_freepagetable에서 tid가 -1일 경우 모든 스레드를 종료하는 코드를 실행합니다.
4. 본인이 아니고 main 스레드가 아닌 경우 freethread를 호출합니다.
5. 본인이 아니고 main 스레드인 경우 freeproc을 호출하지 않고 직접 초기화해줍니다. freeproc을 호출할 경우 proc_freepagetable을 또 호출하기 때문에 문제가 생깁니다.
6. 그 다음 만약 본인이 main일 경우 기존 proc_freepagetable의 uvmunmap(pagetable, TRAPFRAME, 1, 0)으로 unmap이 되므로 두 번 unmap하지 않기 위해 temp_va가 TRAPFRAME이 아닌 경우에만 직접 unmap을 해줍니다.
7. 이후 기존 코드대로 unmap을 3번 진행합니다.
8. 다시 exec으로 돌아와 main을 본인으로 지정하고, tid는 0, temp_va도 초기화합니다.

sbrk

```
//sysproc.c
uint64
sys_sbrk(void)
{
  uint64 addr;
  int n;

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

메모리 사이즈는 스레드 간에 공유하므로 addr를 본인 스레드의 sz가 아닌 main의 sz를 사용하도록 합니다.

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

```
acquire(&p->lock);
sz = p->main->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->main->sz = sz;
release(&p->lock);
return 0;
}
```

growproc에서도 sz를 전부 main의 것을 사용하도록 합니다.

kill

```
//proc.c
int
kill(int pid)
{
  struct proc *p;
  int count = 0;
  for(p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);
    if(p->pid == pid){
      count++;
      p->killed = 1;
      if(p->state == SLEEPING){
        // Wake process from sleep().
        p->state = RUNNABLE;
      }
    }
    release(&p->lock);
  }
  if(count>0)
    return 0;
```

```
  return -1;
}
```

기존 kill은 조건에 해당하는 프로세스를 찾으면 바로 return을 했습니다. 하지만 조건 상 스레드에서는 한 스레드가 죽으면 모든 스레드가 죽어야 합니다. pid 조건에 맞는 스레드를 찾아도 바로 return하지 않고 모든 스레드를 찾도록 해줍니다.

wait

```
//proc.c
int
wait(uint64 addr)
{
  struct proc *pp;
  int havekids, pid;

  ...

        if(pp->tid == 0) {
          freeproc(pp);
        } else {
          freethread(pp);
        }

        ...
}
```

일반적으로 스레드는 join을 통해 free를 하게 됩니다. 하지만 kill이 된 경우 메인 스레드도 kill이 되어 join을 해주지 못하기에 wait을 통해 free를 합니다. 이때 main 스레드가 먼저 wait으로 수거되는 경우 freeproc을 통해 모든 스레드를 free시켜줄 수 있습니다. 하지만 main 스레드보다 다른 스레드가 먼저 wait으로 수거되는 경우 freeproc을 통해 pagetable을 free해서는 안되므로 freethread를 하도록 합니다.

sleep

sleep도 수정 사항이 없습니다. 이미 기존 코드에서 호출한 대상만 sleep이 되도록 구현되어있기에 그대로 사용해도 됩니다.

pipe

pipe도 수정 사항이 없습니다. 스레드여도 프로세스와 동일하게 pipe가 작동합니다.

3. Results

제공된 테스트 코드를 실행시켜 결과를 확인했습니다.

```
xv6 kernel is booting

init: starting sh
$ thread_test

[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 = 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#5]
Thread 0 start, pid 9
Thread 1 start, pid 9
Thread 2 start, pid 9
Thread 3 start, pid 9
Thread 4 start, pid 9
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!!
```

거의 모든 결과가 동일하지만 TEST#5에서 pid 값이 다릅니다. 아마 스레드를 생성할 때 allocproc을 안 써서 다른 것 같습니다. allocproc을 사용해야 nextpid 값이 증가하면서 TEST#5 상황에서 pid가 29가 나옵니다. 저는 스레드 생성 시에 nextpid 값이 증가하지 않기에 다른 결과가 나왔습니다.
또한 TEST#5에서 Thread 0 end가 안뜹니다. kill을 호출한 스레드인 Thread 0도 kill의

대상이기 때문에 저 문장을 출력하기 전에 죽어서 그렇습니다. kill을 호출한 스레드를 kill의 대상으로 지정하지 않으면 제대로 출력이 됩니다.

4. Troubleshooting

이번 과제에서 가장 힘들었던 부분은 panic: freewalk: leaf입니다. pagetable을 free하기 전에 매핑되어 있는 모든 trapframe을 unmap해줘야 한다는 것을 몰랐기에 왜 저런 패닉이 뜨는지 알지 못했습니다. 저 패닉의 원인을 찾는데 많은 시간을 썼습니다. 대부분의 test에서 저 패닉으로 인해 실행이 잘 되지 않았습니다.

그리고 TEST#4에서 메모리 사이즈가 처음에는 초기값이 35000으로 나왔었습니다. 실행은 잘 되기에 크게 신경 쓰지는 않았으나 왜 저렇게 나오는지 알지 못해서 쉽게 고치지 못했습니다. user stack 관련 코드를 수정하니 고쳐졌습니다.