ELE3021_project02

Path of Project02 & How to execute xv6

- project 2의 위치

  Project_2/xv6-public에 구현했습니다. (노란색 형관펜으로 표시된 곳)

Name	Last commit	Last update
.vscode	UDT: mlfq 갈아엎고 임시 커밋	1 month ago
Project_1/xv6-public	FIN: L2 queue	1 month ago
Project_2	UDT: kfree	15 hours ago
lab4_systemcall	FIX: lab4 - system call - comment fix	2 months ago
lab6-sync	CRET: mlfq_test 추가, 시스템콜 이름 변경, 내일 다...	1 month ago
xv6-public	FIX: proc_ticks 초기화 오류 fix, mlfq 완성(?)	1 month ago

- 실행방법
  Project_2/xv6-public 디렉토리에서

```
make clean
./makemake.sh
./run-xv6.sh
```

  를 쉘에 입력하여 실행합니다.

  빌드 화면 예시

```
:~/Desktop/operating_system/2023_ele3021_2020005269/Project_2/xv6-public$ ./makemake.sh
in -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -fno-pic -O -nostdinc -I. -c bootmain.c
in -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -fno-pic -nostdinc -I. -c bootasm.S
Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
ock.asm
```

Design

- exec2

  기존 exec는 user stack을 하나(4096 bytes) 할당 받지만, 인자인 stacksize만큼 유저 스택을 할당 받게하는 함수입니다. 만약 stacksize = 1이라면 기존 exec 함수와 동일하게 동작하게 되며, stacksize = 3이라면 가드 페이지 한 장과 유저 스택 3장을 받게 됩니다. stacksize는 [1, 100] 범위의 정수입니다.

- setmemorylimit

일반적인 시스템 콜을 구현하듯 구현할 예정입니다.

메모리 리밋을 바꾸려고 하는 프로세스를 p라고 하면

- p→pid == pid를 만족하는 p가 없다면 -1을 반환합니다.
- 만약 limit < p→sz라면 -1을 반환합니다.

위 두 경우에 해당하지 않으면 p→limit 변수를 새로운 limit 변수로 바꾸고 0을 반환합니다.

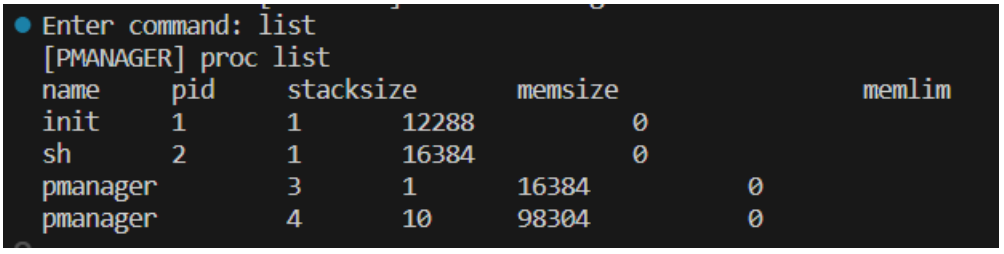
- pmanager
  - list

    proc.c에 pmlist()라는 시스템 콜을 구현하여 출력할 것입니다.

    프로세스(=메인 스레드)가 아닌 서브 스레드의 정보는 출력하지 않고, 프로세스 중에서 SLEEPING, RUNNABLE, RUNNING에 해당하는 프로세스의 정보만 출력합니다.

    출력 되는 항목들은 다음과 같습니다. (프로세스 p에 대해)

    - name: p→name을 출력합니다.
    - pid: p→pid를 출력합니다.
    - stacksize: 스택 페이지 개수를 의미합니다. p→stacksize를 출력합니다. 이는 **가드 페이지는 제외한 유저 스택 페이지의 개수**입니다.
    - memsize: 현재 할당받은 메모리 크기를 출력합니다. 이는 **p→sz값**입니다.
    - memlim: 프로세스가 받을 수 있는 최대 메모리 크기인 p→memlim을 출력합니다. 이 값은 setmemorylimit() 시스템 콜로 변경될 수 있습니다. memlim == 0이면 limit이 없음을 의미합니다.



> ⚠ pmanager의 list의 정보들은 tab(\t)으로 구분되어 출력됩니다.
> 따라서 특정 항목의 길이가 길면 다음 항목이 밀려 보일 수 있으나, 순서대로 name, pid, stacksize, memsize, memlim을 의미합니다.
>
> 예를 들면 위 사진의 맨 아래 줄 'pmanager' 프로세스를 보면 stacksize가 4인 것처럼 보일 수 있으나 실제로는 순서대로 읽어야하므로
> name: pmanager | pid: 4 | stacksize: 10 | memesize: 98304 | memlim: 0
> 임을 의미합니다.

  - kill

    해당 pid를 가진 프로세스를 kill하고, 성공하면 '[PMANAGER] Successfully killed pid n'이라고 출력하고, 실패하면 '[PMANAGER] Fail to kill pid n'이라고 출력합니다.

  - execute <path> <stacksize>

stacksize는 가드 페이지를 제외한 유저 페이지의 수로 인식합니다.

즉 <stacksize> == 3이라면 1개의 가드페이지와 3개의 유저 스택 페이지로 해당 path를 실행합니다.

실패 시에만 실패했다는 메시지를 출력합니다.

- memlim <pid> <limit>

  setmemorylimit()을 호출하여 새로운 limit 값을 설정합니다. 성공 또는 실패 메시지를 출력합니다.

- exit

  exit() 시스템 콜을 호출하여 종료합니다.

- LWP
  - 프로세스 struct에 스레드 구현에 필요한 몇 가지 변수(ex. tid)를 추가하고, tid > 0 이면 서브 스레드, tid == 0이면 프로세스(=메인 스레드)로 구분하는 방법을 사용하였습니다.
  - thread_create

    allocuvm()을 이용하여 프로세스(=메인 스레드)의 힙 영역에서 가드 페이지 한 장, 유저 페이지 한 장을 할당받아 스레드의 페이지로 할당해줍니다.
    즉 메인 스레드의 영역을 늘려서 나눠주는 것이므로, 메인 스레드를 포함한 해당 프로세스에 속한 스레드의 모든 sz값이 증가합니다. 그리고
    그 메모리 영역은 프로세스가 exit()을 호출하고 부모 프로세스가 회수하므로 프로세스가 exit()하기 전까지 프로세스의 sz값은 감소하지 않습니다.
    또한 allocuvm으로 할당해주기 이전에 main→memlim 값과 스레드가 생성된다면 바뀔 메인 스레드의 sz값을 비교하여 스레드를 만들 수 있을지 없을지를 판단합니다.

    즉 **main→memlim 값과, main→sz+2*PGSIZE 값을 비교**합니다.

    ex.
    어떤 프로세스의 memlim이 4096*4인 상태에서, thread를 하나 만들면 memlim에 딱 맞는 메모리 영역을 차지하게 됩니다. 여기서 thread가 thread_exit을 호출하더라도 아직 프로세스가 exit하지 않았기에 메모리 영역이 회수 되지 않았고, sz값도 줄어들지 않기에 더 이상의 스레드를 만들 수 없게 됩니다.
  - thread_exit

    해당 함수를 통해 스레드를 종료합니다. 이 함수에서 스레드의 state를 ZOMBIE로 바꿉니다. 그리고 자신을 create한 parent를 깨웁니다.
  - thread_join

    기다리는 스레드의 state가 ZOMBIE가 될 때까지 sleep하며 기다립니다. 해당 스레드가 ZOMBIE로 바뀌면 해당 스레드의 필드를 초기화 시키고 커널 스택을 반납합니다. 여기서 user stack은 반납하지 않고, 프로세스가 exit()될 때 한꺼번에 반납합니다.

Implementation

이번 과제를 수행하기 위해 proc.h에 있는 struct proc의 필드를 추가하였습니다.

```
struct proc
{
```

```
  uint sz;                     // Size of process memory (bytes) -> 공유
  pde_t *pgdir;                // Page table -> 공유
  int pid;                     // Process ID -> 공유
  struct proc *parent;         // 부모 스레드
  int killed;                  // If non-zero, have been killed -> 공유
  struct file *ofile[NOFILE]; // Open files ->
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)

  // [Project 2] 추가 변수 (프로세스에서 필요한 애들)
  uint memlim;                 // 메모리 리밋. 0이면 리밋없음
  int stacksize;               // 스택 몇장 받았는지.
  int tnum;                    // 이 프로세스가 몇개의 스레드 갖고 있는지, (자신 제외)
  int parent_pid;              // 부모 프로세스의 pid.


  // [Project 2] 스레드끼리 각자 가지고 있는 애들
  char *kstack;              // Bottom of kernel stack for this process -> 각자
  enum procstate state;    // Process state -> 각자
  struct trapframe *tf;    // Trap frame for current syscall -> 각자
  struct context *context; // swtch() here to run process -> 각자
  void *chan;                  // If non-zero, sleeping on chan -> 각자
  struct proc *main;           // 메인 스레드. 즉 어떤 프로세스의 스레드인지를 나타냄


  // [Project 2] 추가 변수 (스레드에서 필요한 애들)
  thread_t tid;                // 스레드이면 1이상의 값(nexttid로 결정), 프로세스이면
  void* retval;             // exit, join에서 쓰이는 return value
};
```

- Pmanager
  - pmanager user program의 구현

    pmanager.c에 구현해놓았습니다.

    while(1){..} 무한 포문을 돌며, 커맨드를 입력받고 해당 커맨드를 파싱하여 적절한 결과를 보여줍니다. 커맨드를 구분할 때 strcmp()함수를 사용하였습니다.

    “exit”이라는 커맨드가 입력되면 프로그램을 종료합니다.
  - pmanager 관련 시스템 콜
    - pmlist()

      “list” 명령이 들어오면 실행되는 시스템 콜로 proc.c에 구현해 놓았습니다.

      ptable을 돌며 tid == 0 이면서 (tid가 0이라는 것은 서브 스레드가 아닌 메인 스레드 즉, 프로세스임을 의미) state가 SLEEPING, RUNNABLE, RUNNING인 프로세스의 정보들을 출력합니다.
    - setmemorylimit()

      프로세스가 할당 받을 수 있는 메모리의 최대치를 설정하는 시스템 콜로 proc.c에 구현하였습니다.

tid == 0이면서 pid가 일치하는 프로세스를 찾아 struct proc에 있는 필드인 memlim을 변경해줍니다. 일치하는 프로세스를 찾지 못했다면 -1을 반환합니다.

- exec2
  exec.c에 구현했습니다. 기존 exec 코드와 유사하지만 다음 부분을 추가, 변경 하였습니다. 기존엔 가드 페이지 하나, 유저 페이지 하나를 받기 위해 2장의 페이지를 할당 받았지만, 여기서는 stacksize+1장을 할당 받습니다.

```
// 프젝 2. stacksize가 유효한지 검사
  if(stacksize <= 0 || stacksize > 100)
    goto bad;

  acquire(&ptable.lock);
  // Allocate two pages at the next page boundary.
  // Make the first inaccessible.  Use the second as the user stack.
  sz = PGROUNDUP(sz);
  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE*(1+stacksize))) == 0) {
    release(&ptable.lock);
    goto bad;
  }
  clearpteu(pgdir, (char*)(sz - 2*PGSIZE*(1+stacksize)));
  sp = sz;
  release(&ptable.lock);


  // [프젝 2] 만약 sz가 memlim보다 크면? (memlim == 0 일때 말고)
  if(curproc->memlim > 0 && curproc->memlim < sz)
    goto bad;

  curproc->stacksize = stacksize; // exec2는 스택 stacksize장
```

- 그외 exit(), kill()

  기존에 구현되어 있는 exit(), kill() 시스템 콜을 이용했습니다. (lwp 구현으로 인해 두 함수의 내부 내용은 변경되었습니다.)

- Light Weight Process
  - thread_create, thread_exit, thread_join 시스템 콜들은 proc.c에 구현해 놓았습니다.
  - thread_create:

    allocproc(), fork(), exec() 함수의 내용을 참고하여 구현했습니다.

    ptable을 돌며 UNUSED인 스레드 t를 찾습니다.

    찾았다면 해당 스레드 t의 상태를 EMBRYO로 바꿉니다.

    - 커널 스택의 할당:
      t의 kstack을 할당하고 그 kstack에 trapframe의 자리를 만들고, 그 trapframe은 부모 스레드의 내용을 복사해 옵니다. 다음에 커널 스택에 trapret을 넣을 자리를 만들고 trapret을 저장합니다. 다음에 context가 들어갈 자리를 만들고, 해당 context의 eip에 forkret을 저장합니다. create 함수의 마지막 부분에 tf의 eip에 start_routine을 저장하고, esp에는 유저 스택의 포인터를 넣어줍니다.

이렇게 해서 스레드는 forkret으로 점프 → trapret으로 점프 → start routine으로 점프하여 실행하게 됩니다.

- 스레드 변수의 초기화:
  스레드의 필드들을 메인 스레드 값을 참고하여 적절하게 초기화 해줍니다.
  tid는 스레드의 id를 의미합니다. 이 값은 nexttid라는 전역변수를 증가시켜서 이전 스레드와 겹치지 않도록 설정해줍니다.

```
// 메인 스레드 가져오기
  main = curproc->main;

  pgdir = main->pgdir;


  t->tid = nexttid++;
  *thread = t->tid;

  t->parent = curproc; // 부모는 현재 스레드이다. 얘가 join 해줌
  t->main = main; // 메인 스레드 넣어주기
  t->pid = main->pid;
  t->pgdir = pgdir;

  // 프젝 2에서 추가된 변수들도 다 카피
  t->memlim = curproc->memlim;
  t->stacksize = curproc->stacksize;
  t->tnum = curproc->tnum;

  t->retval = 0; // 그냥 초기화 -> 사실 안해도 될것 같은데
  t->parent_pid = main->pid;   // exit함수를 위한 변수
```

- 유저 스택의 할당:
  유저 스택은 main의 힙 영역을 늘려서 스레드에게 나눠주는 방식으로 할당합니다. allocuvm을 이용해서 할당해주며, 할당 전에 스레드를 만든 후 sz값이 memlim보다 작거나 같은지 확인해줍니다. allocuvm으로 스레드의 스택을 할당해 줄 때, 나와 같은 프로세스에 속한 모든 스레드의 sz의 값을 동기화 시켜주어야 하기 때문에, vm.c의 allocuvm함수의 마지막 부분에 다음과 같은 부분을 추가했습니다.

```
for(t = ptable.proc; t < &ptable.proc[NPROC]; t++){

    if(t->pid == p->pid) t->sz = newsz;
  }
```

  유저 스택을 할당받은 후 위 한 페이지는 가드페이지로 만들어주고, arg와 fake return address를 넣어줍니다.

```
if((sz = allocuvm(pgdir, sz, sz+2*PGSIZE)) == -1) { // 얘는 exit에서 f
    t->state = UNUSED;
    kfree(t->kstack);
```

```
    release(&ptable.lock);
    return -1;
  }
  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));


  t->sz = sz; // 초기 sz는 페이지 2장
  main->sz = sz;
  st = sz;


  safestrcpy(t->name, main->name, sizeof(main->name));

  st -= 4;
  *(uint *)st = (uint)arg;
  st -= 4;
  *(uint *)st = (uint)0xffffffff;
  t->sz = sz;


  t->tf->eip = (uint)start_routine; // 여기도 문제가 아닌거 같고
  t->tf->esp = (uint)st; //이게 문제가 아니네
```

- thread_exit:
  exit(void) 함수를 참고하여 구현했습니다.
  현재 스레드의 retval 필드에 인자로 들어온 retval을 저장하고, 참조하고 있던 파일을 닫고, 자신의 상태를 ZOMBIE로 바꾼 뒤 부모가 자신의 자원을 회수해줄 수 있게 부모를 깨웁니다.
- thread_join:
  wait()함수를 참고하여 구현했습니다.
  ptable을 돌면서 t→tid == thread인 스레드를 찾고, 그 상태가 ZOMBIE이면 필드를 초기화하고, 커널 스택을 반납합니다. 해당 스레드가 아직 ZOMBIE가 아니라면 sleep을 호출하여 기다립니다.
- exec, exec2:
  exec, exec2가 호출되었을 때, 그 프로세스가 스레드를 가지고 있다면 자신을 제외한 같은 프로세스에 속한 모든 스레드를 정리해야합니다. 따라서 exec.c에 thread_remove라는 함수를 구현하여 exec, exec2 안에서 호출했습니다.
  이 함수는 curproc인 자신을 제외한 모든 스레드를 정리합니다. curproc이 메인 스레드가 아니더라도 메인 스레드도 정리하고 이제부터는 서브 스레드였던 curproc이 메인 스레드가 됩니다.

```
// 나와 pid가 같은 애들을 나빼고 모두 종료
void
thread_remove(struct proc* curproc)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p == curproc) // 나는 죽이면 안됨
      continue;
```

```
    if(p->pid != curproc->pid) // 다른 프로세스의 스레드는 죽이면 안됨
      continue;

    if(p == curproc->main) {
      curproc->parent = p->parent;
    }

    //죽이자
    kfree(p->kstack); // 커널 스택 반납
    p->kstack = 0;

    p->pid = 0;
    p->tid = 0;
    p->parent = 0;
    p->name[0] = 0;
    p->state = UNUSED;

    p->memlim = 0;
    p->stacksize = 0;
    p->tnum = 0;
    p->pgdir = 0;

    p->retval = 0;
  }

  // 내 정보 갱신
  curproc->tid = 0;
  curproc->memlim = 0;
  curproc->stacksize = 0;
  curproc->tnum = 0;
  curproc->main = curproc;

}
```

- exit():

  현재 프로세스가 exit됐다면, 현재 프로세스의 스레드들이 만든 프로세스 또는 스레드의 부모를 init process로 바꿔주어야 합니다.
  그리고 프로세스에 속한 스레드들은 자신의 메인 프로세스의 부모 프로세스로 부모가 변경됩니다. 해당 과정을 담은 코드입니다.

```
  // main을 가져오자
  main = curproc->main;
  parent = main->parent;
  pid = main->pid;
```

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

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent_pid == curproc->pid && p->pid != pid) { // 자신의 부모인
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }

    if(p->pid == pid) {
      p->parent = parent;
      p->state = ZOMBIE;
    }
  }
```

- wait():

  기존의 wait은 자식 프로세스만 정리해주지만, lwp 구현 후에는 그 프로세스에 속한 서브 스레드들도 모두 정리를 해주어야합니다.

  해당 과정을 담은 코드를 wait()에 추가하였습니다.

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;

      havekids = 1;


      if(p->state == ZOMBIE) { // 얘가 좀비라는건 다 exit()됐다는거임
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        pgdir = p->pgdir;
        freevm(pgdir);
        p->pgdir = 0;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;

        // [프젝 2] 새로 추가된 변수들
        p->memlim = 0;
        p->stacksize = 0;
        p->tnum = 0;
```

```
        p->tid = 0;
        p->retval = 0;

        p->state = UNUSED;

        // 나와 같은 프로세스에 속한 스레드 정리
        for(t = ptable.proc; t < &ptable.proc[NPROC]; t++) {
          if(t == p) // 방금 해제한 스레드이면 패스
            continue;

          if(t->pid == pid) { // 정리
            kfree(t->kstack);
            t->kstack = 0;
            t->pid = 0;
            t->parent = 0;
            t->name[0] = 0;
            t->killed = 0;
            t->pgdir = 0;

            t->memlim = 0;
            t->stacksize = 0;
            t->tnum = 0;
            t->tid = 0;
            t->retval = 0;
            t->state = UNUSED;
          }
        }
        release(&ptable.lock);
        return pid;
      }
    }
```

Result

pmanager의 실행화면입니다.

입력의 순서는 다음과 같습니다.

list
execute thread_exec 10
execute thread_exit 100
execute thread_kill 1
list
list
exit

```
$ pmanager
[PMANAGER] Hello! Pmanager started
Enter command: list
[PMANAGER] proc list
name     pid     stacksize       memsize                 memlim
init     1       1       12288           0
sh       2       1       16384           0
pmanager         9       1       16384           0

Enter command: execute thread_exec 10
Enter command: Thread exec test start
Thread 0 start
Thread 1 start
Thread 2 staThread 3 start
Thread 4 start
rt
Executing...
Hello, thread!
execute thread_exit 100
Enter command: Thread exit test start
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Exiting...
execute thread_kill 1
Enter command: Thread kill test start
list
[PMANAGER] proc list
name     pid     stacksize       memsize                 memlim
init     1       1       12288           0
sh       2       1       16384           0
pmanager         9       1       16384           0
thread_kill      12      1       61440           0
thread_kill      13      1       61440           0

Enter command: Killing process 13
This code should be executed 5 times.This code should be executed 5 times.
This This code should be executed 5 timThis code should be executcode should be execes.
ed 5 times.

uted 5 times.
Kill test finished
list
[PMANAGER] proc list
```

```
list
[PMANAGER] proc list
name     pid     stacksize       memsize                 memlim
init     1       1       12288           0
sh       2       1       16384           0
pmanager         9       1       16384           0

Enter command: exit
[PMANAGER] Bye!
zombie!
zombie!
$
```

pmanager에서 ‘memlim’ 커맨드를 이용하여 pid == 3인 ‘pmanager’ 프로세스의 memlim을 바꾸는 실행 결과입니다.

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ pmanager
[PMANAGER] Hello! Pmanager started
Enter command: list
[PMANAGER] proc list
name     pid     stacksize       memsize                 memlim
init     1       1       12288           0
sh       2       1       16384           0
pmanager         3       1       16384           0

Enter command: memlim 3 20000
[PMANAGER] Successfully set memlim (pid 3 with limit 20000)
Enter command: list
[PMANAGER] proc list
name     pid     stacksize       memsize                 memlim
init     1       1       12288           0
sh       2       1       16384           0
pmanager         3       1       16384           20000

Enter command: memlim 3 15000
[PMANAGER] Fail to setmemorylimit (pid 3 with limit 15000)
Enter command: list
[PMANAGER] proc list
name     pid     stacksize       memsize                 memlim
init     1       1       12288           0
sh       2       1       16384           0
pmanager         3       1       16384           20000

Enter command: memlim 3 0
[PMANAGER] Successfully set memlim (pid 3 with limit 0)
Enter command: list
[PMANAGER] proc list
name     pid     stacksize       memsize                 memlim
init     1       1       12288           0
sh       2       1       16384           0
pmanager         3       1       16384           0

Enter command:
```

thread_test의 실행 화면입니다.

Trouble Shooting

- usertests의 오류

  lwp를 구현하며 proc.c의 이곳저곳을 고쳤는데, usertests를 실행하면 fourfiles test에서 'balloc: out of blocks'가 뜨는 것을 확인했습니다.

  allocuvm과 같은 메모리 할당과 관련된 문제라고 예상했고 원인을 찾으려고 노력하였지만, 결국 원인을 찾지 못해 고치지 못했습니다.

```
$ usertests
usertests starting
arg test passed
createdelete test
createdelete ok
linkunlink test
linkunlink ok
concreate test
concreate ok
fourfiles test
lapicid 0: panic: balloc: out of blocks
 80101a1e 80101c32 8010232f 80101880 80105dc9 80105a7c 80106df0 80106b34 0 0
```

- 메모리 할당에 대한 에러
  lwp 구현 시, thread_create를 구현하며, 'panic: remap', 'trap 14'와 같은 에러를 많이 만났습니다. 원인은 스레드를 create하고 main의 virtual memory 공간이 늘어났지만 그 프로세스에 속한 스레드들의 sz 값을 동기화 시켜주지 않아서 이미 pgdir에서 할당이 된 곳에 재할당을 하려고 하여 remap이 발생하거나 메모리에 잘못 접근하여 trap 14가 발생한다는 것을 알게 되었습니다. main thread의 sz값이 늘어날 때마다 그 프로세스에 속한 thread의 sz값을 모두 동기화 시켜주었더니 해결되었습니다.