

[ELE3021] Operating System  
Project #2 Wiki  
2019014266 Lim Kyu Min

# Contents

## I. Introduction

Project #2에 대한 개요이며, Makefile 수정사항에 대해 서술합니다.

## II. Changes in Proc.h

과제 명세에 맞춰 기존과 달라진 Proc.h에 대해 서술합니다.

## III. System Call exec2(), setmemorylimit() Design

과제 요구 사항 중 하나인 exec2(), setmemorylimit()의 구현에 대해 서술합니다.

## IV. Pmanager

과제 요구 사항 중 하나인 pmanager의 구현에 대해 서술합니다.

## V. LWP (Threads)

과제 요구 사항 중 하나인 LWP의 구현 방법과  
작동 방식에 대해 서술합니다.

## VI. Result Preview

지금까지 구현한 내용을 바탕으로  
실제 xv6에서 작동하는 것을 보여줍니다.

## VII. Trouble Shooting & Limitations

원래 구현하려고 하던 방향이 무엇이었고, 해당 방법을 사용하지 못한 이유와,  
지금 최종 제출 본에서도 완벽하게 해결하지 못한 문제에 대해 서술합니다.

\* 코드에서 /\* 로 시작하는 주석은 해당 프로젝트 구현 도중 제가 단 주석이며, 이는 코드 구현  
의 의도와 내용을 이해하는 데에 도움을 줄 것입니다.

\* Project #1와 마찬가지로 서술할 내용이 많아 분량이 많습니다. (총 29페이지) 제가 정말 열심히  
한 과제이고, 그만큼 설명해드리고 싶은 부분이 많은 것과, 아쉬움이 많이 남는 과제라는 뜻이니,  
너그러운 마음으로 봐주시면 감사하겠습니다! 😊

# I. Introduction

Project #2 는 크게 exec2.c, pmanager, LWP 3 가지를 구현하는 과제라고 판단을 했고, 각각 순서대로 구현을 완료했습니다. LWP 같은 경우는 많은 시행착오와 시간이 들어간 프로젝트였고, 이 위키를 제출하는 지금도 완벽하다고 말하기는 어려울 것 같습니다. 이 위키에는 제가 각 요구 사항들을 어떻게 구현했는지, 어떤 워크 플로우를 갖고 있는지를 보여주고, 마지막 Trouble Shooting & Limitations 에서는 제가 원래 구현하려던 방향과 실패한 이유, 그리고 현재로서 예상되는 문제점을 서술함으로서 해당 위키를 마무리하겠습니다.

## - Makefile

```
#Project 2 Implementation: exec2.o, wrapper.o thread.o

OBJS = \
    bio.o \
    console.o \
    exec.o \
    exec2.o \
    file.o \
    fs.o \
    ide.o \
    ioapic.o \
    kalloc.o \
    kbd.o \
    lapic.o \
    log.o \
    main.o \
    mp.o \
    picirq.o \
    pipe.o \
    proc.o \
    sleeplock.o \
    spinlock.o \
    string.o \
    swtch.o \
    syscall.o \
    sysfile.o \
    sysproc.o \
    trapasm.o \
    trap.o \
    uart.o \
    vectors.o \
    vm.o \
    wrapper.o \
    thread.o
```

Makefile, OBJS.

- Project #2에 추가된 object file들입니다. 각각 exec2.c, wrapper.c, thread.c 3가지입니다. exec2.c에는 exec2 구현 관련 내용이, wrapper.c에는 Project #2에 사용된 System Call중, User Program에서도 사용할 수 있도록 하게 해주는 wrapper functions들이, thread.c는 LWP 구현 관련 코드가 있습니다.

UPROGS=\_cat\_\_echo\_\_forktest\_\_grep\_\_init\_\_kill\_\_ln\_\_ls\_\_mkdir\_\_rm\_\_sh\_\_stressfs\_\_wc\_\_zombie\_\_dev\_\_pmanager\_\_thread_test\_\_thread_exec\_\_thread_exit\_\_thread_kill\_\_hello_thread\_\_	EXTRA=\_mkfs.c \_ulib.c \_user.h \_cat.c \_echo.c \_forktest.c \_grep.c \_kill.c \_ln.c \_ls.c \_mkdir.c \_rm.c \_stressfs.c \_wc.c \_zombie.c \_printf.c \_umalloc.c \_dev.c \_pmanager.c \_thread test.c \_thread exec.c \_thread kill.c \_thread exit.c \_hello thread.c \_README \_dot-bochsrc \_*.pl \_toc.* \_runoff \_runoff.list \_gdbinit tmpl \_gdbutil
--	---

Makefile, UPROGS & EXTRA.

- Project #2에 추가된 User Program들입니다. 각각 pmanager.c, dev.c, 그리고 제공된 tese code 들입니다. pmanager.c는 pmanager 구현 관련 내용이, dev.c에는 thread관련 테스트 프로그램이 정의되어 있습니다.

## II. Changes in Proc.h

Project #2 의 특성상, 기존의 proc.h 내의 proc structure 의 많은 부분이 수정되었습니다. 어떤 부분이 수정되거나 추가되었는지 서술하겠습니다.

```
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[NFILE];              // Open files
    struct inode *cwd;                     // Current directory
    char name[16];                          // Process name (debugging)
    int stacksize;                          // Allocated Stack Page Size
    int memlim;                            // Memory Limit (Unlimited if 0 assigned)
    int threadnum;                          // total thread number
    int isthread;                           // flag variable shows current process is thread or not.
    struct thread_t *thread;               // contains thread information.
    int thctr;                             // number of thread created: used for making new thread id.
};
```

proc.h – Proc Structure Definition

- int stacksize: exec(), exec2() 호출 시, 해당 system call에 알맞는 stacksize 값이 저장됩니다.
- int memlim: setmemorylimit() 호출 시, 부여 받은 memory limit 값을 저장합니다. growproc()에서는 해당 값을 참조하고 반영하여 메모리를 할당합니다. 0 을 기본값으로 갖고, 0 은 제한 없음 (Unlimited)를 의미합니다.
- int threadnum: 현재 프로세스가 가지고 있는 스레드의 총 개수를 나타냅니다. 기본적으로 0 을 가지고 있으며, 해당 프로세스를 부모로 하는 스레드의 개수만큼 늘어납니다.
- int isthread: 해당 프로세스가 스레드인지 판단합니다. 제가 구현한 xv6에서는 ptable.proc의 일부를 스레드로 활용을 하며, 해당 값이 0 이면 프로세스, 1 이면 스레드라고 xv6 가 인식합니다.
- struct thread\_t \*thread: 스레드 관련 정보를 가리키는 thread 포인터입니다. 해당 thread\*를 통해 스레드에 대한 정보에 접근할 수 있습니다. 해당 structure에 대한 설명은 V. LWP (Threads)에 자세히 설명하겠습니다.
- int thctr: 스레드 카운터입니다. 해당 값을 바탕으로 스레드의 id 인 tid 가 결정됩니다. 하나의 스레드가 만들어질 때마다 하나씩 증가합니다.

```
#define NPROC      200 // maximum number of processes
#define KSTACKSIZE 4096 // size of per-process kernel stack
```

param.h – Changed value of NPROC

- 새로 만든 기능인 Thread 또한 ptable.proc를 사용하니, 좀 더 많은 NPROC 값이 필요하다고 판단이 되었습니다. 해당 값을 64 에서 200 으로 늘렸습니다.

### III. System Call exec2(), setmemorylimit() Design

Project #2 의 첫번째 요구사항인 exec2()와 setmemorylimit()의 구현에 대해 서술하겠습니다.

#### I. int exec2(char \*path, char \*\*argv, int stacksize)

- exec2는 기존 exec.c를 기반으로 하여 만들었습니다. exec2.c에 해당 내용이 구현되어 있습니다.
- 기존 exec에서 달라진, 즉 추가되거나 수정된 부분에 대해서만 기술하겠습니다.

```
/* exec2: Process Execution with various stacksize
int
exec2(char *path, char **argv, int stacksize)
{
    char *s, *last;
    int i, off;
    uint argc, sz, sp, ustack[3+MAXARG+1];
    struct elfhdr elf;
    struct inode *ip;
    struct proghdr ph;
    pde_t *pgdir, *oldpgdir;
    struct proc *curproc = myproc();

    /* Check if current process is thread
    if(curproc->isthread == 1){
        /* Eliminate other threads except this.
        purgethreads(curproc->thread->parent, curproc);
    }

    /* Check if current process contains thread
    if(curproc->threadnum > 0){
        /* Eliminate all thread.
        purgethreads(curproc, 0);
    }

    begin_op();

    /* Check Stack Size
    if(stacksize < 1 || stacksize > 100){
        end_op();/* If stacksize is not valid, end current operation (Invalid Call)
        cprintf("exec2: invalid stack size\n");
        return -1;
    }

    if((ip = namei(path)) == 0){
        end_op();
        cprintf("exec2: fail\n");
        return -1;
    }
    ilock(ip);
    pgdir = 0;
```

exec2.c [1/2] – Thread management and Allocate Stack Size

- 기존 exec에서 변경점 중 첫번째 부분입니다. 이하 ELF header, Loading Program의 코드는 모두 동일합니다.
- 먼저 thread관련하여 작업을 처리합니다. 이는 thread에서 exec가 호출이 되거나, thread를 가진 process에서 exec가 호출될 때 어떻게 처리가 되어야 하는지에 대한 기능이며, exec와 동일하게 exec2또한 해당 기능을 가지고 있습니다.
  - 해당 기능의 자세한 부분은 V. LWP(Thread)에서 exec()관련 수정사항에 자세히 서술하겠습니다.
- 그 이후로는 stacksize를 받습니다. 해당 stacksize는 1 이상, 100 이하의 정수이며, 이 값에 맞게 스택이 할당되게 됩니다. 가드페이지 또한 명세에 맞게 할당이 됩니다.

```

    /* exec2: allocate stack size, 1 guard page. (stacksize + 1)
if((sz = allocuvm(pgdir, sz, sz + (stacksize + 1)*PGSIZE)) == 0)
    goto bad;
clearpteu(pgdir, (char*)(sz - (stacksize + 1)*PGSIZE));
sp = sz;

/* Set Stack & Memory info of process
curproc->stacksize = stacksize;
curproc->memlim = 0;

```

exec2.c [2/2] – Allocate virtual memory based on stacksize.

- 기존 exec에서 변경점 중 두번째 부분입니다. 이하 나머지 코드는 모두 동일합니다.
- 기존과 달리 stacksize + 1 만큼의 PGSIZE를 할당합니다. 해당 프로세스는 할당 받은 stacksize만큼의 size를 갖게 되고, 해당 stack page 바로 하단에 guard page 또한 할당 받습니다.
- 이후 해당 stacksize 정보를 저장하고, memlim은 (아직 설정이 되지 않았으니) 0으로 초기화 합니다.

```

int
sys_exec2(void)
{
    char *path, *argv[MAXARG];
    int stacksize;
    int i;
    uint uargv, uarg;

    if(argv[0] == 0 || argv[1] == 0 || argv[2] == 0){
        return -1;
    }
    memset(argv, 0, sizeof(argv));
    for(i=0;; i++){
        if(i >= NELEM(argv))
            return -1;
        if(fetchint(uarg+4*i, (int*)&uarg) < 0)
            return -1;
        if(uarg == 0){
            argv[i] = 0;
            break;
        }
        if(fetchstr(uarg, &argv[i]) < 0)
            return -1;
    }
    return exec2(path, argv, stacksize);
}

```

sys\_exec2() – Wrapper function for exec2.

- User Program에서의 exec2 System call을 위해 정의된 wrapper function입니다. Exec2()를 실행 한다는 점 외에는 기존의 sys\_exec()와 동일합니다.

## II. Int setmemorylimit(int pid, int limit)

- setmemorylimit()은 proc.c 내부에 정의되어 있으며, 현재 동작하는 프로세스의 최대 메모리 공간을 제한하는 memory limit 기능을 수행합니다. 이때 limit은 byte 단위입니다.

```

/* setmemorylimit: set memory limit for process.
int
setmemorylimit(int pid, int limit){
    struct proc *p;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid != pid || p->state == UNUSED)
            continue;

        /* Process Found
        break;
    }

    if(p >= &ptable.proc[NPROC]){
        /* Process Not Found.
        sprintf("setmemorylimit: Process Not Found.\n");
        return -1;
    }

    if(limit < 0){
        /* Invalid Limit Value.
        sprintf("setmemorylimit: Invalid Memory Limit Value.\n");
        return -1;
    }
    if(limit != 0 && limit < p->sz){
        /* Requested limit is less than size. (0: no limitation)
        sprintf("setmemorylimit: Less limit value than allocated size.\n");
        return -1;
    }

    /* Limit Allocation
    p->memlim = limit;
    /* This limitation will be enforced in growproc();

    return 0;
}

```

setmemorylimit() – Receive and set memory limit to current process.

- 구조 자체는 굉장히 간단합니다. 우선 pid에 해당하는 프로세스를 찾습니다.
- 만약 찾았으면, 넘겨받은 memory limit 값을 memlim에 저장하는데, 다음 2가지 경우에는 저장이 되지 않고 -1을 반환합니다.
  - 만약 해당 pid에 해당하는 프로세스를 찾지 못하였을 경우.
  - 만약 넘겨받은 인자가 음수일 경우.
  - 만약 해당 값이 0이 아님에도 (즉, Unlimited memory limit이 아님에도), 기존에 할당된 process memory size보다 적을 경우.
- 만약 넘겨받은 memory limit이 음수인 경우, 0으로 바꾼 후, Unlimited에 해당하는 값인 0으로 바꾼 후 계속 실행합니다.
- 이외의 경우에는 정상적으로 해당 값을 할당한 후, 0을 반환합니다.

```

int
growproc(int n)
{
    uint sz;
    struct proc *curproc = myproc();

    sz = curproc->sz;

    /* Check memory limit
    if(curproc->memlim != 0 && sz + n > curproc->memlim) { /* Memory limit exists, but current process grows more than its limitation.
        sprintf("FATAL ERROR: Out of memory - allocated more than its limitation.\n");
        return -1;
    }

    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;
    switchuvm(curproc);
    return 0;
}

```

growproc() – Grow process memory space until its limit.

- 기존의 growproc()과는 달리, memory limit이 무제한이 아님에도 불구하고 할당하고자 하는 메모리가 제한 이상일 시, Error Message를 할당한 후, 해당 작업을 취소합니다.

```

int
sys_setmemorylimit(void){
    int pid, lim;
    if(argint(0, &pid) < 0)
        return -1;

    if(argint(1, &lim) < 0)
        return -1;

    return setmemorylimit(pid, lim);
}

```

```

int
atoi(const char *s)
{
    int n;
    char neg = 0;
    /* Allows Negative Value.

    n = 0;
    while('0' <= *s && *s <= '9') || *s == '-'{
        if(*s == '-'){
            neg = 1;
            s++;
            continue;
        }
        n = n*10 + *s++ - '0';
    }

    if(neg == 1){
        return 0 - n;
    }
    return n;
}

```

sys\_setmemorylimit() – Wrapper Function

atoi() – allows negative value

- exec2()와 동일하게 setmemorylimit() 또한 wrapper function을 만들었습니다.
- 명세에는 음수를 입력 받는 경우도 고려하는 것 같아, atoi()를 수정하여 음수 값 또한 반영할 수 있게 만들었습니다.(기존에는 음수가 들어올 시, 0을 return합니다.)
  - 위의 코드는 45-20와 같이 작정하고 오타를 내는 케이스는 제대로 계산을 하지 못 하지만 (아마 결과로 -4520이 나올 것입니다), 해당 케이스는 이번 명세 외 케이스 아니, 정상적으로 앞에 음수가 붙은 경우 (-10, -4, -105), 해당하는 음수 값을 반환하도록 수정했습니다.

## IV. Pmanager

Project #2의 두번째 요구사항인 Pmanager에 대해 서술합니다. 해당 기능은 pmanager.c에 정의가 되어 있습니다. Pmanager 를 구성하는 여러 function 들 및 기능을 모두 서술하겠습니다. Pmanager.c는 sh.c를 굉장히 많이 참고하였고, 해당 기능을 바탕으로 새로 재구성하여 만들었습니다.

### 1. Pmanager.h

- Pmanager 에는 필요한 #define constants 와 function prototype 을 보며, 어떤 함수가 정의되어 있는지 서술하겠습니다.

```
/* Defined Commands
#define LIST 1
#define KILL 2
#define EXEC 3
#define MEMLIM 4
#define EXIT 5

/* Defined constants
#define BUF_SIZE 100
#define MAX_ARGV 10

/* Function Prototypes
int recv_cmd(char*, int); /* receive command from user
void run_cmd(int, char*); /* run current command
int parse_cmd(char*); /* parse command, make it recognizable to pmanager.
char* null_eliminate(char*, int); /* eliminate null at the end of the command.
char** parse_argument(char*, char**, int, int*); /* parse arguments.
int forkproc(); /* Fork process, panic for failure.
```

pmanager.h – Definition of constants & function prototypes required for manager

#### i. Constants

- LIST, KILL, EXEC, MEMLIM, EXIT: pmanager 가 수행하는 각 명령어에 해당하는 operation 값입니다. 해당 값을 기준으로 pmanager 은 어떠한 작업을 진행할지 결정합니다.
- BUF\_SIZE: Buffer 의 크기입니다. Pmanager 가 받을 수 있는 최대 명령어 글자 수이며, 100 으로 제한을 두었습니다.
- MAX\_ARGV: 각 명령어마다 최대 argument 개수입니다. 최대 개수를 10 으로 제한했습니다.

#### ii. Function Prototypes

- recv\_cmd: 유저에게 command, 즉 명령어를 받습니다. 여기에는 arguments 도 포함됩니다.
- parse\_cmd: 유저가 입력한 command 를 실행할 수 있는 형식으로 바꿉니다.
- parse\_argument: 유저가 입력한 argument 별로 배열로 저장하여 사용할 수 있게 만들어줍니다.
- run\_cmd: parse 되어 실행 가능해진 명령어와 나머지 arguments 를 바탕으로 command 를 실행합니다.
- null\_eliminate: 입력시 들어가는 null 관련 문자로 인해 발생하는 오류를 막기 위해 제거해줍니다.
- forkproc: 일반적인 fork 와 같으나, 실패할 시 -1 를 return 합니다. (sh.c 의 fork1()와 같은 역할입니다.)

## 2. Pmanager.c

- Pmanager.c는 여러 함수로 구분이 되어 있습니다. 각 함수마다 구분하여 서술하겠습니다.

```
/* Process Manager
int
main(){
    static char buf[BUF_SIZE];
    printf(0, "Initializing pmanager...\\n");

    while(recv_cmd(buf, sizeof(buf)) >= 0){

        int cmd = parse_cmd(buf);

        if(cmd == 5)
            goto exit; /* Exit!

        if(forkproc() == 0){
            run_cmd(cmd, buf);
        }
        wait();
    }

exit:
    printf(0, "Exiting Pmanager... Bye Bye~\\n");
    exit();
}
```

pmanager.c [1/8] – main

- Pmanager 가 처음 실행되는 main입니다.
- recv\_cmd 를 통해 유저에게 실행할 명령어를 받습니다. (recv\_cmd 함수는 하단에 설명되어 있습니다.)
  - 유저가 아무것도 입력하지 않더라도, '\n'은 기본적으로 입력이 되기 때문에, exit 을 입력하지 않는 이상, 해당 함수는 계속 실행되며 계속 유저에게 명령어를 받습니다.
- parse\_cmd()를 통해 유저가 실행하려고 하는 명령어를 구분하고, 실행 가능한 형태로 변환해줍니다. (parse\_cmd는 하단에 정의되어 있습니다.)
  - 만약, 해당 명령어가 exit 이라면 (cmd == 5), goto exit;을 실행하여 함수를 나가고, 해당 pmanager 를 종료시킵니다.
- 나머지 명령어는 해당 명령어를 실행할 process 를 forkproc() 를 통해 형성한 후 (background 에서 실행, sh.c 의 backcmd 참고), 해당 작업이 끝날때 까지 wait()을 해줍니다.
  - forkproc()이 실패할 시, 오류 메세지를 띄우고 Pmanager 를 종료시킵니다.
- 명령어에 해당하는 작업이 끝날 시, 다시 recv\_cmd()를 통해 다음 명령어를 받습니다.

```
int
recv_cmd(char *buf, int buf_size){
    printf(2, "[PMANAGER] > ");
    memset(buf, 0, buf_size);
    gets(buf, buf_size);

    buf = null_eliminate(buf, buf_size);
    if(buf[0] == 0){ /* Error: EOF
        return -1;
    }

    return 0;
}
```

pmanager.c [2/8] – recv cmd

- Pmanager에서 유저에게 command를 받는 함수입니다.
- 먼저 memset을 활용하여 buf를 0으로 초기화를 해준 후, 해당 buf에 입력을 받습니다.
- 입력 받은 이후, null\_eliminate를 통해 의미가 없는 빈칸들을 모두 제거합니다.
- 만약 오류로 인해 아무것도 없는 빈 buffer 가 올 경우, -1 을 return 하며 pmanager 를 종료시킵니다.

```

char* null_eliminate(char* cmd, int buf_size){
    int start = 0;
    int mv = 0;
    int idx = 0;
    /* Eliminate meaningless whitespaces in front of command.

    while(idx < buf_size){
        if(cmd[idx] == ' ') { /* meaningless white space
            start++;
            idx++;
            continue;
        }

        if(start == 0) /* no need to eliminate
            break;

        /* Move All Command foward.
        for(int j = start; j < buf_size; j++){
            cmd[mv] = cmd[j]; /* move command;
            cmd[j] = 0;

            if(cmd[mv] == '\n') /* moved all command; break.
                break;

            mv++;
        }

        break; /* Move Complete
    }

    return cmd;
}

```

pmanager.c [3/8] – null eliminate

- 유저가 입력한 명령어 중, 앞쪽의 의미 없는 null 문자나 whitespace를 제거합니다.
- 앞 쪽에 whitespace 가 존재할 경우, 해당 whitespace는 무시, 즉 명령어의 일부로 취급하지 않습니다.
- 첫 글자가 발견될 경우, 해당 시점부터 명령어로 인식하며, escape sequence 가 발견될때까지 옮깁니다.
- 해당 작업을 통해서 초반에 입력된 whitespace로 인한 오류는 발생하지 않습니다.
  - ‘list’, ‘list’ 모두 동일한 list 명령어로 취급이 됩니다.

```

int
parse_cmd(char* cmd){
    if(cmd[0] == 'l' && cmd[1] == 'i' && cmd[2] == 's' && cmd[3] == 't' && (cmd[4] == '\n' || cmd[4] == ' ')){
        return LIST; /* Current command is list command.
    }
    else if(cmd[0] == 'k' && cmd[1] == 'i' && cmd[2] == 'l' && cmd[3] == 'l' && (cmd[4] == '\n' || cmd[4] == ' ')){
        return KILL; // Current command is kill command
    }
    else if(cmd[0] == 'e' && cmd[1] == 'x' && cmd[2] == 'e' && cmd[3] == 'c'
            && cmd[4] == 'u' && cmd[5] == 't' && cmd[6] == 'e' && (cmd[7] == '\n' || cmd[7] == ' ')){
        return EXEC; // Current command is execute command
    }
    else if(cmd[0] == 'm' && cmd[1] == 'e' && cmd[2] == 'm' && cmd[3] == 'l'
            && cmd[4] == 'i' && cmd[5] == 'm' && (cmd[6] == '\n' || cmd[6] == ' ')){
        return MEMLIM; // Current command is memlim command
    }
    else if(cmd[0] == 'e' && cmd[1] == 'x' && cmd[2] == 'i' && cmd[3] == 't' && (cmd[4] == '\n' || cmd[4] == ' ')){
        return EXIT; // Current command is exit command
    }
    return -1;
}

```

pmanager.c [4/8] – parse cmd

- 유저가 입력한 명령어를 기반으로, pmanager 가 실행할 수 있는 형식으로 바꿔줍니다.
- 더욱 나은 방법이 있었겠지만, LWP 구현에 더 많은 시간을 쓰기 위해 해당 방법은 Hard Coding 방법으로 구현했습니다.
- 각 위치의 character 를 비교하여 명령어를 찾아내고, 마지막 문자는 띄어쓰기 및 escape sequence 둘 중 하나가 오면 인식하도록 했습니다. 해당 방법으로 명령어를 입력하고 바로 enter를 누르는 경우, whitespace를 입력한 후에 enter를 누르는 경우 모두 동일한 명령어를 실행하도록 구현했습니다.
- 각 내용은 다음과 같은 명령어로 실행합니다.
  - list
  - kill <pid>
  - execute <path> <args> <stacksize>
  - memlim <pid> <limit>
  - exit

```
int forkproc(){
    int pid;

    pid = fork();
    if(pid < 0){
        return -1;
    }

    return pid;
}
```

pmanager.c [5/8] – forkproc

- sh.c 의 fork1()와 비슷한 역할입니다. Fork 가 실패할 시 -1 을 return 합니다.

```
void
run_cmd(int cmd, char *buf){
char* argv[MAX_ARGV]; /* String For Argument Parsing
int pid;
int rear = 0; /* Indicates the last element. e.g.) execute's stacksize.

switch(cmd){
    case LIST: /* list
        if(list() < 0){
            printf(0, "Pmanager: Unexpected Error Occured while executing command \"%s\", shutting down pmanager.\n", buf);
            exit(); /* Unexpected Error
        }
        wait();
        break;

    case KILL: /* kill
        parse_argument(buf, argv, BUF_SIZE, &rear);

        pid = atoi(argv[0]);

        if(pid <= 0){ /* Invalid pid
            printf(0, "Invalid Pid. Pid must be integer bigger than 0\n");
            break;
        }

        /* Kill process
        if(kill(pid) < 0){
            printf(0, "Kill command failed; There are no such process with pid %d\n", pid);
        }
        else {
            printf(0, "Successfully killed proccess %d\n", pid);
        }
        break;
}
```

pmanager.c [6/8] – run cmd (1/2)

- run\_cmd()의 구현 코드입니다. 코드의 양이 많아 2 개로 나누어서 설명하겠습니다.
- Rear 는 exec2 에 필요한 변수입니다. 이 변수에 대해서는 run\_cmd (2/2)부분에 서술하겠습니다.
- parse\_cmd()에서 반환된 cmd 값을 활용하여 각 명령어에 맞는 내용을 실행합니다.

➤ LIST

- list()를 실행하여 현재 실행중인 process들의 전체 리스트를 출력합니다.
  - 만약 오류가 나와 list()에서 -1이 return 될 경우, 오류 메세지를 보여준 후, 해당 cmd를 종료 시킵니다.

➤ KILL

- 대상 pid를 argument로 받은 후, kill 작업을 실행합니다. Kill은 기존의 kill() system call을 사용합니다.
- parse\_argument()를 실행하여 command에서 argument를 추출하여 배열에 할당합니다.
- 할당받은 argument를 atoi()를 활용하여 사용할 수 있는 pid로 바꾸고, kill 작업을 실행합니다.
  - Pid가 음수거나, kill이 실패했을 경우, 해당 오류 메세지를 출력한 후 해당 작업을 마칩니다.
  - 성공할 경우, kill한 pid를 출력한 후 해당 작업을 마칩니다.
- list(), parse\_argument()는 run\_cmd에 대한 설명을 마친 후 설명하겠습니다.

```
case EXEC:
    /* Execute

    parse_argument(buf, argv, BUF_SIZE, &rear);
    int stacksize = atoi(argv[rear]); /* End of argument: stack size

    argv[rear] = 0; /* Remove current stacksize argument
    exec2(argv[0], argv, stacksize); // argv[0]: path, argv[1]: stacksize
    //wait();
    break;

case MEMLIM:
    parse_argument(buf, argv, BUF_SIZE, &rear);

    pid = atoi(argv[0]);
    int lim = atoi(argv[1]);
    if(setmemorylimit(pid, lim) < 0){
        printf(0, "REJECTED: No such pid or limit is less than it's allocation\n");
    }
    break;

case EXIT: /* exit
exit();
break;

default: /* No such command
printf(0, "Pmanager: Unknown Command\n");
}

exit(); /* Command Executed Successfully
}
```

pmanager.c [7/8] – run\_cmd (2/2)

- run\_cmd()의 계속된 설명입니다.

➤ EXEC

- exec2()를 실행하는 작업입니다. 실행하기 위해서는 stacksize를 필수로 받습니다.
  - 항상 stacksize는 마지막 인자로 옵니다. (e.g. execute sbbk 16834 3 (stacksize = 3))
  - 해당 stacksize는 rear이라는 항상 마지막 인자의 위치를 가리키는 변수를 통해 access합니다. 해당 마지막 자리는 execute을 실제로 하는 path에는 필요하지 않은 argument이니, 0을 넣어서 실행하는데 지장을 주지 않도록 합니다.
- 받은 argv, stacksize의 정보를 활용하여 exec2()를 실행합니다.

➤ MEMLIM

- Pid에 해당하는 프로세스에 limit를 적용합니다.
- parse\_arguemnt()의 결과로 추출된 argv 정보를 활용하여 setmemorylimit()을 실행합니다.
  - 만약 setmemorylimit()이 실패했을 시, 오류 메세지를 출력하고 종료합니다.

➤ EXIT

- Pmanager를 종료시킵니다.

```
char** parse_argument(char* cmd, char** argv, int buf_size, int *rear){  
    int argvsz = 0; /* argument size  
    int start = 0; /* Argument starts  
    int argvnum = 0; /* number of arguments  
  
    /* Find argument starting place.  
    while(start < buf_size){  
        if(cmd[start++] != ' ') { /* skip the command part  
            continue;  
        }  
  
        /* next part will be starting point of argument  
        break; /* starting position found.  
    }  
  
    for(int i = start; i < buf_size; i++){  
        if(cmd[i] != ' ' && cmd[i] != '\n'){ /* Argument  
            argvsz++;  
            continue;  
        }  
        /* Whitespace found: End of argument.  
        if(argvsz == 0) /* no argument size: skip.  
            continue;  
  
        argv[argvnum] = malloc(argvsz);  
        for(int tgt = 0; tgt < argvsz; tgt++){  
            argv[argvnum][tgt] = cmd[start + tgt];  
        }  
        if(cmd[i] == '\n') /* if current position is 0, it is endpoint.  
            break;  
  
        argvsz = 0;  
        argvnum++;  
        (*rear)++; /* rear moved  
        start = i+1; /* start from next pos  
    }  
  
    return argv;  
}
```

pmanager.c [8/8] – parse\_argument

- Pmanager에서 argument가 필요한 명령어의 경우, 해당 명령어를 사용할 수 있는 형태 (char\*\* array)로 변환해주는 함수입니다.
- 우선 첫번째 command 부분은 넘겨 argument에서 제외합니다. list, execute와 같은 command는 skip합니다.
- 이후 문자가 발견될 때, 해당 문자를 argument라고 인식, 다음 whitespace가 나올 때까지 argv의 element로 복사합니다.
  - argv의 각 element는 각 argument의 첫번째 자리를 pointing하며, argv의 element 수는 곧 argv의 개수를 의미합니다.
- rear에는 마지막 argv의 위치를 가져옵니다. 이 정보는 exec2에서 마지막 인자인 stacksize를 가져오는데 사용됩니다.
- 모든 작업이 끝났으면, argv를 return합니다.

이하는 proc.c에 정의된 list()에 대한 설명을 서술하겠습니다.

```

int
list(){
    struct proc *p;
    struct proc *t;
    uint procsz;

    cprintf("-----\n");
    cprintf("[ PID] name / number of stack page / allocated memory (byte) / memory limit (byte) / Number of running threads |\n");
    cprintf("-----\n");
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        procsz = 0; /* Init
        if(p->state == UNUSED || p->isthread == 1 || p->state == ZOMBIE) /* skip the unused space and thread.
            continue;

        procsz = p->sz;
        if(p->threadnum > 0){
            /* Find threads and sum it all.
            for(t = ptable.proc; t < &ptable.proc[NPROC]; t++){
                if(t->isthread == 1){
                    if(t->thread->parent == p){
                        procsz += t->sz; /* Add thread process size
                    }
                }
            }
        }

        cprintf("%d / %s / %d stacks / %d bytes allocated /", p->pid, p->name, p->stacksize, procsz);
        p->memlim == 0 ? cprintf(" UNLIMITED /") : cprintf(" %d byte(s) /", p->memlim); /* memlim will be 0 if there is no memory limitation.
        cprintf(" Running %d threads\n", p->threadnum);
    }

    return 0;
}

```

list() – List all process running

- Pmanager 의 list 명령어 호출 시 실행되는 함수입니다.
- 프로세스 중 사용되지 않았거나, Zombie 상태의 (종료 예정인) 프로세스, thread 는 출력 대상에서 제외됩니다.
  - Zombie 프로세스는 실행 ‘종료’가 되어 죽은 프로세스임으로, 출력 대상에서 제외하였습니다.
- 만약 thread를 가진 프로세스일 경우, 해당 thread의 memory size까지 합하여서 출력합니다.
- 만약 memlim 이 unlimited (0)일 경우, memory limit 에 대해 UNLIMITED 를 출력합니다.

```

int
sys_list(){
    return list();
}

```

sys\_list() – Wrapper function for list()

- list()에 대한 wrapper function 이며, wrapper.c 에 정의되어 있습니다.

Pmanager에 대한 실행 코드는 이하 VI. Result Preview에 서술하겠습니다.

## V. LWP (Threads)

일단 해당 부분에 대한 위키를 적기전에 앞서, 저는 제가 구현한 스레드가 완벽하지 않을 뿐더러, 문제가 많은 코드라는 것을 인지한 상태입니다. 하지만 그 사실을 알았을 때에는 이미 시간이 많이 흘러간 이후였고, 일단 제가 구현한 코드들을 적고 이후에 Limitations 에 제가 생각하는 단점을 적어서 제출할 계획입니다.

LWP 는 총 3 개의 thread 관련 syscall 을 구현하는 것이었고, 각각 thread\_create, thread\_exit, thread\_join 입니다. 각각을 서술 시작하겠습니다.

### 1. thread.h

- Thread 관련 헤더파일입니다. Thread 관련 function prototype 과 constants 는 defs.h 가 아닌, thread.h 에 정의 했습니다. 유저 프로그램과 커널 프로그램 모두 스레드를 사용하려면 이 thread.h 를 #include 해야합니다.

```
/* Definitions
#define TSTACKSIZE 2000
#define THREADSIZE 4096
#define TMAXPERPROC 100

/* thread_t structure definition
typedef struct thread_t{
    ushort          tid;
    ushort          pid;
    ushort          exitcalled;
    ushort          occupied;
    void*           retval;
    void*(*start_routine)(void*); 
    struct proc*    parent;
    void*           arg;
}thread_t;

/* Function Prototypes
int thread_create(thread_t*, void*(*start_routine)(void*), void* );
void thread_exit(void* );
int thread_join(thread_t, void**);
/* Function in service
thread_t thread_self();
int allocthread(thread_t* );
void terminatethread(void* );
void cleanupthread(struct proc* );
int waitthread(thread_t, void**);
void purgethreads(struct proc*, struct proc*);

/* thread syscalls.
int            fetchthread(uint, thread_t* );
int            argthread(int, thread_t* );
int            argvptr(int, void**, int);
```

thread.h – Wrapper function for list()

- thread\_t
  - tid: thread 의 id 입니다.
  - pid: parent process 의 id 입니다.
  - exitcalled: 만약 thread\_exit 를 통해 exit 이 불려졌다면, 해당 변수는 1 이 됩니다. (default: 0)
  - occupied: 해당 thread 는 배열 형태로 정의되어 있는데, 배열이 어떤 thread 에 의해 사용중이면 해당 변수는 1 이 됩니다. (default: 0)
  - retval: return value 가 저장되는 곳입니다.
  - start\_routine: thread 의 start\_routine 입니다.
  - parent: 부모 프로세스를 가리킵니다.
  - arg: start\_routine 의 argument 를 가리킵니다.
- Functions
  - thread\_create, thread\_exit, thread\_join: 명세에 나와있는 필수 system call 들입니다.

- allocthread(): thread\_create()에 사용되는 함수입니다.
- terminatethread(): thread\_exit()에 사용되는 함수입니다.
- waitthread(), terminatethread(): thread\_join()에 사용되는 함수입니다.
- purgethread(): 어떠한 프로세스의 모든 스레드를 제거하는데 사용되는 함수입니다.
- fetchthread(), argthread(): user program 의 스레드를 불러오는데 사용하는 함수입니다.

이 외에 설명 안 된 것들은 사용이 되지 않는 부분입니다.

## 2. thread\_create() & function calls

- thread.c 와 proc.c 에 정의되어 있는 thread\_create()와 여기에 사용되는 함수 들에 대해서 서술합니다.

### 2.1. thread\_create() (thread.c)

```
/* thread_create
int
thread_create(thread_t *thread, void*(*start_routine)(void*), void *arg){
    /* Thread Creation

    memset(thread, 0, sizeof(thread));
    thread->start_routine = start_routine;
    thread->arg = arg;

    if(allocthread(thread) < 0) /* Error: Thread allocation Failed.
        return -1;

    return 0;
}
```

thread\_create() – create thread process

- 유저 프로그램에서 사용할 thread 와 실행할 함수인 start\_routine, 그리고 함수의 argument 를 받아옵니다.
- 해당 스레드에 사용할 thread 를 kernel 에서 사용할 수 있도록 메모리에 설정을 해준 후, 해당 thread 에 start\_routine 과 argument 를 설정해줍니다.
- proc.c 에 정의된 allocthread() 를 호출하여 thread 를 만듭니다.

### 2.2. sys\_thread\_create() (wrapper.c)

```
/*thread_create - thread.c
int
sys_thread_create(void){
    thread_t *thread;
    void*(*start_routine)(void*) = 0;
    void* arg;

    if(argptr(0, (char**)&thread, sizeof(thread_t *)) < 0 || argptr(1, (char**)&(start_routine), sizeof(void*)) < 0
       || argptr(2, (char**)&arg, sizeof(void*)) < 0){
        printf("sys_thread_create: read args failed.\n");
        return -1;
    }

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

sys\_thread\_create() – Wrapper function for thread\_create()

- thread\_create()의 wrapper function 입니다.
- 차례로 thread, start\_routine, arg 를 읽어온 후, thread\_create 에 해당 인자들을 넘겨줍니다.
- 실패할 경우, 오류 메세지를 띄운 후, -1 을 return 합니다.

### 2.3. allocthread() (proc.c)

```

/* allocthread(thread_t *thread)
/* Referred: thread_create()
/* allocate memory space to thread.
/* Implemented based on code of fork() and exec().
int
allocthread(thread_t *thread){
    struct proc* curproc = myproc();
    struct thread_t* destthread;
    uint sp = 0;
    uint ustack[2]; /* size: basic stack 2:
                      // fake return counter, address of argument, termination
    int tid = ++(curproc->thctr); /* Thread counter will be new thread id.
    struct proc* newthread;
    pde_t *pgdir = 0;

    /* Step 1) Thread init
    /* Find Vacant Thread.
    for(destthread = threadlist; destthread < &threadlist[NPROC]; destthread++){
        if(destthread->occupied == 0){
            break;
        }
    }

    if(destthread >= &threadlist[NPROC]){
        /*Cannot find space.
        cprintf("Allocation Failed while finding thread space.\n");
        goto failed;
    }

    /* Allocation
    thread->pid = curproc -> pid;
    thread->tid = tid;
    thread->parent = curproc;
    thread->retval = 0;
    thread->exitcalled = 0;

    destthread->pid = curproc->pid;
    destthread->tid = tid;
    destthread->parent = curproc;
    destthread->retval = 0;
    destthread->exitcalled = 0;
    destthread->start_routine = thread->start_routine;
    destthread->arg = thread->arg;
    /*Now Occupied
    destthread->occupied = 1;
}

```

allocthread() [1/3] – assign & set thread information.

- allocthread()의 상반부입니다. 필요한 변수들을 정의하고, thread 를 사용할 수 있도록 설정합니다.
- proc.c 의 전역 변수로 정의된 threadlist 에서 비어 있는 (occupied == 0)인 thread element를 찾아, destthread 로 정의합니다.
- 해당 destthread 와 넘겨 받은 thread 모두에 필요한 정보를 저장해줍니다.
- 마지막으로 destthread 에 사용된다는 의미로 occupied 값을 1로 바꾸어줍니다.

```

/* 2) Thread Allocation.
newthread->thread = destthread;

/* 3) Executable Stack Allocation.
/* Copy parent's page directory. - References fork();
if((pgdir = copyuvm(curproc->pgdir, curproc->sz)) <= 0){
    cprintf("Allocation Failed while copying parent's page\n");
    goto failed;
}
//pgdir = shareuvm(pgdir, curproc);

/* Allocate stack frame.
ustack[0] = (uint)0xffffffff;
ustack[1] = (uint)newthread->thread->arg;
/* clone stack

sp = newthread->sz;
sp -= (uint)(sizeof(ustack));

//void* stack_mem = kalloc();
//memmove(stack_mem, ustack, sizeof(ustack) + 1);
if(copyout(pgdir, sp, ustack , (sizeof(ustack))) < 0){
    cprintf("Allocation failed while copying argument stacks\n");
    goto failed;
}

/* 4) Entry Point Setting
newthread->tf->eax = 0;
newthread->tf->ip = (uint)newthread->thread->start_routine;
newthread->tf->esp = sp;

/* 5) file copy - referred fork()
/* I think this was unnecessary
/* however if I remove this, I get weird deadlock :
for(int i = 0; i < NOFILE; i++){
    if(curproc->ofile[i]){
        newthread->ofile[i] = fildup(curproc->ofile[i]);
    }
}
newthread->cwd = idup(curproc->cwd);

```

allocthread() [2/3] – Set thread and pgdir to thread

- 위에서 설정된 destthread 를 연결해줍니다.
- fork()에서 사용되는 copyuvvm()을 사용하여 부모 pgdir 을 가져옵니다.
  - 현재 위키를 쓰는 이 시점에서, 이 방법이 잘못되었다는 점을 인지하는 상태입니다. 해당 방법은 아예 다른 메모리 공간을 형성하고, 유저 프로그램에서 넘겨받는 포인터 값 또한 달라져 return value 가 저장되지 못할 수도 있습니다.
  - 해당 방법을 고치려고 어떤 시도를 했고, 결국에 하지 못했던 이유는 VII. Touble Shooting & Limitations 에 대해서 서술하겠습니다.
- ustack 에는 return 을 해주는 fake return address 와, 다음 위치에는 argument 를 가리키는 포인터를 넣어줍니다.
- 해당 ustack 을 저장하기 위해, sp 를 ustack 의 공간 만큼 옮긴 후, copyout 을 활용하여 해당 정보를 stack 에 넣어줍니다.
- eip 에는 start\_routine 을, esp 에는 바뀐 sp 를 넣어주어 argument 를 가리키게 합니다.
- 이후에는 fork()를 참고하여 file 을 copy 합니다.

```

    /* 6) connect pgdir and miscellaneous things.
    /* copy name
    safestrcpy(newthread->name, curproc->name, sizeof(curproc->name));
    /* give completed pgdir.
    newthread->pgdir = pgdir;
    /* Make it runnable state.
    acquire(&ptable.lock);
    newthread->state = RUNNABLE;
    //newthread->thread = thread;
    release(&ptable.lock);

    //newthread->thread = thread;
    ++(newthread->threadnum);
    return 0;

failed:
    if(pgdir){ /* * Free allocated paged bc failed.
        freevm(pgdir);
    }

    cprintf("thread allocation failed\n");
    return -1;
}

```

allocthread() [3/3] – Finish creating thread.

- Thread creation 을 마무리하고 프로세스의 thread number 를 1 증가시킵니다.
- fork()을 기반으로 만들었습니다.
- 만약 allocthread 과정 중 실패한 부분이 있을 시에는, failed:로 넘어오게 됩니다. 해당 루틴에서는 pgdir 을 (형성되었을 경우) 다시 free 해주고, 실패의 의미로 -1 를 return 합니다.

### 3. thread\_exit() & function calls

- thread 를 종료시키는 system call 인 thread\_exit 과 이에 사용되는 함수들에 대해 서술합니다.

#### 3.1. thread\_exit() (thread.c)

```

    /* thread_exit
    void
    thread_exit(void* ret_val){
        terminatethread(ret_val);
        return;
    }

```

thread\_exit() – Finish current thread

- 스레드를 종료시킵니다. Void pointer 형태의 return value 를 받은 후, terminatethread 함수의 인자로 넘겨줍니다.

### 3.2. sys\_thread\_exit() (wrapper.c)

```
/*thread_exit - thread.c
int
sys_thread_exit(void){
    void* retval;

    if((argptr[0], (char**)&retval, sizeof(void*)) < 0){
        cprintf("sys_thread_exit: read args failed.\n");
        return -1;
    }

    thread_exit(retval);

    return 0;
}
```

sys\_thread\_exit() – Wrapper function for thread\_exit()

- thread\_exit()의 Wrapper Function입니다.
- 차례로 return value를 가져온 뒤, thread\_exit으로 해당 값을 넘겨줍니다.

### 3.3 terminatethread() (proc.c)

```
/* terminatethread()
 * Referred: thread_exit()
 * made based on exit()
void
terminatethread(void* retval){
    struct proc* curthread = myproc(); /* This process must be thread.
    //int fd;

    acquire(&ptable.lock);
    /* Step 1) set return value.
    curthread->thread->retval = retval;
    curthread->thread->exitcalled = 1; /* This thread had just been ended.
    /* Not closing opened file in here
    /* Removing it triggers panic('acquire') at before swtch() call in scheduler.

    /* Step 3) Make this state as zombie.
    /* Not disallocate this thread right now; need to collect retval in thread_join();
    curthread->state = ZOMBIE; /* Zombie State

    /* Step 4) Wake up parent process, and call sched().
    wakeup(curthread->thread->parent);
    sched();
    /*release(&ptable.lock);
}
```

terminatethread() – Save return value and make it zombie state

- 해당 스레드를 종료시키는 terminatethread()입니다. 기존의 exit()을 기반으로 구현했습니다.
- Return value를 스레드에 저장합니다.
- 또한 exit 이 불렸음을 알려주기 위해서 exitcalled 을 1로 만듭니다.
- ZOMBIE state 를 부여한 후, 해당 스레드의 부모를 wakeup()한 후, 스케줄러에게 권한을 넘겨줍니다.
- thread\_join()에서 사용될 retval 을 포함한 thread 데이터를 위해 terminatethread()에서 thread 를 disallocate 하지 않습니다.
  - Disallocation 은 이후 thread\_join()에서 이루어집니다.

### 3. thread\_join() & function calls

- 형성된 thread 들이 exit 할때까지 기다린 후, return value 를 thread 에 저장해주는. thread\_join 과 각종 function calls에 대해 서술합니다.

#### 3.1. thread\_join() (thread.c)

```
/* thread_join
int
thread_join(thread_t thread, void** retval){
    if(waitthread(thread, retval) < 0){ /* Error: Thread Join Failed.
        cprintf("thread_join failed\n");
        return -1;
    }

    return 0; /* Successfully terminated.
}
```

thread\_join() – wait created thread to exit

- 형성된 thread 가 exit 할때까지 기다리는 함수입니다.
- waitthread()를 실행하며, 오류가 나서 -1 이 return 될 시, 오류메세지와 함께 -1 을 return 합니다.

### 3.2. sys\_thread\_join() (wrapper.c)

```
/*thread_join - thread.c
int
sys_thread_join(void){
    thread_t thread;
    void** retval;

    if(argthread(0, &thread) < 0 || argptr(1, (char**)&retval, sizeof(void**)) < 0){
        cprintf("sys_thread_join: read args failed.\n");
        return -1;
    }

    thread_join(thread, retval);
    return 0;
}
```

sys thread join() – Wrapper function for thread join()

- thread\_join 의 Wrapper function 입니다
- 차례로 thread, retval을 받아와 thread\_join()에 넘겨줍니다.
- Thread 의 경우, 포인터가 아닌 일반 변수를 받아옴으로, argthread 를 따로 만들어 가져와줍니다.

### 3.3. argthread() & fetchthread() (syscall.c)

```
/* Get the thread from userprogram.
/* Implemented based on argint()
int
argthread(int n, thread_t* tp){
    return fetchthread((myproc()->tf->esp) + 4 + 4*n, tp);
}

/* fetchthread -> get thread from user program.
/* implemented based on fetchint()
int
fetchthread(uint addr, thread_t* tp){
    struct proc *curproc = myproc();

    if(addr >= curproc->sz || addr+sizeof(thread_t) > curproc->sz){
        return -1;
    }

    *tp = *(thread_t*)(addr);
    return 0;
}
```

argthread(), fetchthread() – Get thread variable from user program.

- syscall.c 에 정의된 argthread()와 fetchthread()입니다. 포인터가 아닌, 일반 변수로서의 thread 를 가져옵니다.
- 각각 argint(), fetchint()를 참고하여 만들어졌습니다.

### 3.4. waitthread() (proc.c)

```

    /* waitthread(thread_t thread, void** retval)
    ** Referred: thread_join
    ** wait until thread ends.
    ** implemented based on wait();
    ** Function called thread_join will perform circular wait until thread ends.
int
waitthread(thread_t thread, void** retval){
    /* Step 1) Find targeting thread.
    struct proc* tgtthread = 0;
    struct proc* curproc = myproc();

    acquire(&ptable.lock);

    for(tgtthread = ptable.proc; tgtthread < &ptable.proc[NPROC]; tgtthread++){
        if((tgtthread->isthread == 1) && (tgtthread->thread->tid == thread.tid))
            break; /* Thread Found
    }

    if(tgtthread == 0){
        cprintf("thread_join: invalid thread.\n");
        release(&ptable.lock);
        return -1;
    }

    /*if((tgtthread->thread->parent != curproc) && (tgtthread->thread->parent != initproc)){
        // * Thread can only be joined to it's caller.
        cprintf("thread_join: not a caller\n");
        release(&ptable.lock);
        return -1;
    }*/

    /* Step 2) wait until current thread end.
    while(1){ /* Circular Wait.
    if(((tgtthread->thread->exitcalled == 1) && (tgtthread->state == ZOMBIE))){
        // * Current thread had just been ended.
        // * Step 3) Save Return Value.
        *retval = tgtthread->thread->retval;
        // * Step 4) Clean up thread.
        cleanupthread(tgtthread);
        release(&ptable.lock);
        return 0;
    }

    // * Sleep until it's child calls thread_exit(), waking up its parent(current process).
    sleep(curproc, &ptable.lock);
    }

    release(&ptable.lock);
    return -1;
}

```

waitthread() – Wait until targeting thread exit, handle its finishing routine.

- thread\_join 의 메인 기능을 실행합니다. 기존에 만들어져있는 wait() system call 을 참고하여 만들었습니다.
- 대상으로 하는 thread 가 종료되었는지 종료되지 않았는지 확인합니다.
  - 만일 종료 대상 thread 의 부모가 현재 join 을 호출한 process 가 아니라면, -1 을 return 합니다.
- 대상 Thread를 찾으면 while loop 로 돌아간 후, 다음과 같은 분기로 나누어집니다.
  - Thread 가 종료되지 않았을 경우: sleep()하여 스레드가 종료할 때까지 기다립니다. 이후 해당 스레드의 thread\_exit() 호출, 혹은 다른 이유로 인해 wakeup()이 되게 된다면, 다시 while 의 처음으로 돌아가 분기 조건을 확인합니다.
  - Thread 가 종료되었을 경우: retval 을 해당 스레드에 저장한 후, cleanupthread()를 호출하여 disallocation 을 진행합니다. 성공적으로 완료했을 경우, 0 을 return 합니다.
- 만약 thread 가 종료되는 경우를 제외하고 while 이 어떠한 이유로 인해 break 되어 벗어날 경우, 해당 케이스는 오류로 판단하여 -1 을 return 시킵니다.
  - Thread 가 종료되지 않으면 loop 을 계속 돌 것이고, 종료되었으면 return 하여 해당 함수 자체가 끝나니, 정상적인 케이스에서는 while 문 아래의 명령어들이 실행될 이유가 없습니다.

### 3.5. cleanupthread() (proc.c)

```

    /* cleanupthread(sturct proc* tgtthread)
    /* Referred: waitthread(), thread_join().
    /* deallocate, and clean thread space
    /* CAUTION: MUST CALL WITH LOCKED PTABLE (acquire(&ptable.lock))
void
cleanupthread(struct proc* tgtthread){
    /* Find parent
    struct proc* parent = tgtthread->thread->parent;
    /* Start Cleaning Procedure.
    /* Clean thread space
    tgtthread->thread->pid = 0;
    tgtthread->thread->tid = 0;
    tgtthread->thread->exitcalled = 0;
    tgtthread->thread->retval = 0;
    tgtthread->thread->start_routine = 0;
    tgtthread->thread->parent = 0;
    tgtthread->thread->arg = 0;
    tgtthread->thread->occupied = 0;
    tgtthread->thread = 0;

    /* Clean thread process.
    kfree(tgtthread->kstack);
    tgtthread->kstack = 0;
    freevm(tgtthread->pgdir);
    tgtthread->pid = 0;
    tgtthread->parent = 0;
    tgtthread->name[0] = 0;
    tgtthread->killed = 0;
    tgtthread->state = UNUSED;

    /* Reduce number of thread.
    --(parent->threadnum);
}

```

cleanupthread() – Clean & disallocate targeting thread from memory.

- 대상 thread 를 메모리에서 깨끗하게 disallocate 시키는 함수입니다.
- 해당 함수가 종료된 후, 해당 parent 의 thread number 를 하나 줄입니다.
- 해당 함수는 ptable.lock 을 acquire 한 후에 호출해야하며, 마찬가지로 종료 후에도 해당 함수를 호출한 곳에서 release 를 해야합니다.

## 4. System Calls

- 구현된 thread 를 지원하기 위해 각 systemcall 에 다음과 같은 수정 사항을 적용했습니다.

### 4.1. fork()

```

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

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

    // Copy process state from proc.
    if((np->pgdir = copyvm(curproc->pgdir, curproc->sz)) == 0){
        kfree(np->kstack);
        np->kstack = 0;
        np->state = UNUSED;
        return -1;
    }
    np->sz = curproc->sz;
    if(curproc->isthread == 1){
        np->parent = curproc->thread->parent; /* Prevent process being zombie if thread terminated. */
    }
    np->parent = curproc;
    *np->tfn = *curproc->tfn;

    // Clear eax so that fork returns 0 in the child.
    np->tf->eax = 0;

    for(i = 0; i < NOFILE; i++){
        if(curproc->ofile[i])
            np->ofile[i] = filenum(curproc->ofile[i]);
        np->cwd = idup(curproc->cwd);
    }

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

    pid = np->pid;
    acquire(&ptable.lock);

    np->state = RUNNABLE;
    release(&ptable.lock);
}

return pid;
}

```

fork() – Allocate new process if thread calls fork

- Thread에서 fork()를 호출할 경우를 대비하여 fork()를 수정했습니다.
- Thread에서 fork()를 호출할 시, 새로운 process를 형성합니다.
- 이때, 기존의 thread가 종료되었을 때, ZOMBIE가 되는 것을 막기 위해, parent 또한 해당 thread의 부모 process로 바꾸어줍니다.
  - 해당 xv6에서 thread의 부모가 thread인 경우는 존재하지 않음으로, thread에서 fork를 통해 형성된 새로운 프로세스는 해당 thread의 부모 process를 parent로 갖게되고, 해당 부모 process가 종료될 시, 자동으로 killed, 이후에 사라지게 됩니다. (일반적인 child의 행보.)

#### 4.2. exec()

```

int
exec(char *path, char **argv)
{
    char *s, *last;
    int i, off;
    uint argc, sz, sp, ustack[3+MAXARG+1];
    struct elfhdr elf;
    struct inode *ip;
    struct proghdr ph;
    pde_t *pgdir, *oldpgdir;
    struct proc *curproc = myproc();
    //struct proc *tgt;

    /* Check if current process is thread
    if(curproc->isthread == 1){
        /* Eliminate other threads except this.
        purgethreads(curproc->thread->parent, curproc);
    }

    /* Check if current process contains thread
    if(curproc->threadnum > 0){
        /* Eliminate all thread.
        purgethreads(curproc, 0);
    }

    begin_op();
}

```

exec() – Handle cases for thread

- Process에서 exec이 호출되면 모든 스레드가 정리되어야 합니다.
- Thread에서 exec이 호출되면 해당 thread를 제외한 나머지 thread가 정리되어야 합니다.
- 해당 기능을 수행하기 위해서, 현재 프로세스가 thread인 경우, thread는 아니지만 가지고 있는 thread가 존재할 경우, 위의 기능을 실행해주는 purgethread()를 호출합니다.

##### 4.2.1. purgethread (proc.c)

```

/*purgethreads()
/* Kill all thread it have except exception.
void
purgethreads(struct proc* p, struct proc* exception){
    struct proc* tgt;

    for(tgt = ptable.proc; tgt < &ptable.proc[NPROC]; tgt++){
        if(tgt->isthread == 1){
            if(tgt != exception && tgt->parent == p){
                /* If current thread is not the exception (thread must be alive)
                //and current thread's parent is the same process passed from argument.
                //cleanupthread(tgt);
                acquire(&ptable.lock);
                tgt->killed = 1;
                if(tgt->state == SLEEPING){
                    tgt->state = RUNNABLE;
                }
                release(&ptable.lock);
            }
        }
    }
}

```

purgethread() – Kill all thread except exception.

- Exception을 제외한 해당 프로세스의 모든 thread들을 kill합니다. 기본 system call인 kill()을 참고하여 구현했습니다

#### 5. sbrk()

- 원래라면 thread 의 공유 메모리가 서로 겹치지 않게 하도록 처리하는 기능과, 늘어난 메모리 또한 공유할 수 있도록 설정해야 하나, thread\_create()에서 설명한 분할 공간 문제로 인해 해당 syscall 은 수정 사항이 존재하지 않습니다. 이 사항에 대해서는 VII. Trouble Shooting & Limitations 에 상세히 서술하겠습니다.

## 6. kill()

```
// Kill the process with the given pid.
// Process won't exit until it returns
// to user space (see trap in trap.c).
int
kill(int pid)
{
    struct proc *p;
    struct proc **t;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            p->killed = 1;
            // Wake process from sleep if necessary.
            if(p->isthread != 1 && p->threadnum > 0){
                for(t = ptable.proc; t < &ptable.proc[NPROC]; t++){
                    if(t->isthread == 1){
                        if(t->thread->parent == p){
                            t->killed = 1;
                        }
                    }
                }
            }else if(p->isthread == 1){
                for(t = ptable.proc; t < &ptable.proc[NPROC]; t++){
                    if(t->isthread == 1){
                        if(t->thread->parent == p->thread->parent){
                            t->killed = 1;
                        }
                    }
                }
            }
            if(p->state == SLEEPING)
                p->state = RUNNABLE;
            release(&ptable.lock);
            return 0;
        }
    }
    release(&ptable.lock);
    return -1;
}
```

kill() – Handle cases for thread.

- Thread 가 kill 되었을 경우, 해당 Process 의 모든 thread 가 kill 되어야 합니다.
  - Kill 대상인 Process 가 thread 를 가지고 있거나, thread 일 경우, Process 내의 모든 thread 를 kill 합니다.
  - 나중에 wait 에서 thread 의 경우, cleanupthread()가 호출되어 자원을 회수합니다.

```
if(p->isthread == 1){
    /* go to cleanupthread routine.
    cleanupthread(p);
} else{
```

Part of wait() – call cleanupthread() if current zombie process is thread.

## 7. sleep, pipe

- sleep 과 pipe 모두 thread 일 경우에 분기가 존재하지 않아 따로 수정하지 않았습니다. 이는 제 thread 가 xv6에서 프로세스처럼 여겨지기 때문에, 해당 syscall 의 처리 방법 또한 일반 process 와 동일하기 때문입니다.

## VI. Result Preview

Project #2 의 Pmanager, LWP (Thread)의 결과 및 시연입니다.

<pre>init: starting sh \$ pmanager Initializing pmanager... [PMANGER] &gt; [REDACTED]</pre>	<p>- ‘Pmanager’ 명령어를 입력함으로서 pmanager를 시작합니다</p>
<pre>[PMANGER] &gt; list   [PID] name / number of stack page / allocated memory (byte) / memory limit (byte) / Number of running threads   ----- [1] / init / 1 stacks / 12288 bytes allocated / UNLIMITED / Running 0 threads [2] / sh / 1 stacks / 16384 bytes allocated / UNLIMITED / Running 0 threads [3] / pmanager / 1 stacks / 16384 bytes allocated / UNLIMITED / Running 0 threads [4] / pmanager / 1 stacks / 16384 bytes allocated / UNLIMITED / Running 0 threads [PMANGER] &gt; [REDACTED]</pre>	<ul style="list-style-type: none"><li>- list<ul style="list-style-type: none"><li>➢ 명령어 ‘list’를 입력함으로써 현재 실행되고 있는 프로세스들을 모두 보여줍니다.</li><li>➢ Thread는 출력되지 않으며, 실행중인 프로세스만 출력이 됩니다.</li><li>➢ 스레드를 가지고 있는 프로세스의 경우, 해당 스레드의 size (sz)만큼 크기가 추가로 출력되며, 몇 개의 스레드를 가지고 있는지 출력됩니다.</li></ul></li></ul>
<pre>\$ pmanager Initializing pmanager... [PMANGER] &gt; list   [PID] name / number of stack page / allocated memory (byte) / memory limit (byte) / Number of running threads   ----- [1] / init / 1 stacks / 12288 bytes allocated / UNLIMITED / Running 0 threads [2] / sh / 1 stacks / 16384 bytes allocated / UNLIMITED / Running 0 threads [3] / pmanager / 1 stacks / 16384 bytes allocated / UNLIMITED / Running 0 threads [4] / pmanager / 1 stacks / 16384 bytes allocated / UNLIMITED / Running 0 threads [PMANGER] &gt; kill 3 Successfully killed process\$ 3 zombie! [PMANGER] &gt; kill 10 Kill command failed; There are no such process with pid 10 [PMANGER] &gt; [REDACTED]</pre>	<ul style="list-style-type: none"><li>- kill &lt;pid&gt;<ul style="list-style-type: none"><li>➢ 명령어 ‘kill’과 pid 입력함으로써 프로세스가 종료되는 것을 보여줍니다.</li><li>➢ 현재 3 번 프로세스인 pmanager를 kill 하니, 성공적으로 3 번 프로세스를 종료시켰습니다.</li><li>➢ 해당 예시의 경우, pmanager 가 종료되는 순간 kill 을 실행하던 child 의 부모가 종료되어, 해당 child 는 zombie!를 띄우게 됩니다.</li><li>➢ 존재하지 않는 pid 를 Kill 하게 되면, 존재하지 않는 pid 임을 알려주고 실행을 취소합니다.</li></ul></li></ul>
<pre>[PMANGER] &gt; execute ls 10 . .. README cat echo forktest grep init [REDACTED] [PMANGER] &gt; execute echo HELLO!! 4 HELLO!! [PMANGER] &gt; [REDACTED]</pre>	

- execute <path> (args) <stacksize>
- 명령어 kill 와 함께 path, (필요할 경우 해당 path 에 대한 arguments), stacksize 를 입력합니다.
- 과제 명세 중 하나인 exec2()로 실행이 됩니다. 부여받은 stacksize 만큼의 stack 을 할당해준 후 실행이 됩니다.
- ‘execute ls 10’의 경우, 10 개 만큼의 stack 을 할당해준 후, ls 명령어를 실행합니다.
- ‘execute echo HELLO!! 4’의 경우, 4 개 만큼의 stack 을 할당해준 후, echo 명령어를 실행하며 argument 또한 적용합니다.
- 해당 명령어의 stacksize 는 항상 마지막 위치에 오게 됩니다.

```
[PMAGER] > list
| [PID] name / number of stack page / allocated memory (byte) / memory limit (byte) / Number of running threads |
[1] / init / 1 stacks / 12288 bytes allocated / UNLIMITED / Running 0 threads
[2] / sh / 1 stacks / 16384 bytes allocated / UNLIMITED / Running 0 threads
[3] / pmanager / 1 stacks / 16384 bytes allocated / UNLIMITED / Running 0 threads
[4] / pmanager / 1 stacks / 16384 bytes allocated / UNLIMITED / Running 0 threads
[PMAGER] > memlim 3 18000
[PMAGER] > list
| [PID] name / number of stack page / allocated memory (byte) / memory limit (byte) / Number of running threads |
[1] / init / 1 stacks / 12288 bytes allocated / UNLIMITED / Running 0 threads
[2] / sh / 1 stacks / 16384 bytes allocated / UNLIMITED / Running 0 threads
[3] / pmanager / 1 stacks / 16384 bytes allocated / 18000 byte(s) / Running 0 threads
[6] / pmanager / 1 stacks / 16384 bytes allocated / UNLIMITED / Running 0 threads
[PMAGER] > memlim 3 0
[PMAGER] > lis
Pmanager: Unknown Command
[PMAGER] > list
| [PID] name / number of stack page / allocated memory (byte) / memory limit (byte) / Number of running threads |
[1] / init / 1 stacks / 12288 bytes allocated / UNLIMITED / Running 0 threads
[2] / sh / 1 stacks / 16384 bytes allocated / UNLIMITED / Running 0 threads
[3] / pmanager / 1 stacks / 16384 bytes allocated / UNLIMITED / Running 0 threads
[9] / pmanager / 1 stacks / 16384 bytes allocated / UNLIMITED / Running 0 threads
[PMAGER] > 
----- 
[PMAGER] > memlim 3 1
REJECTED: No such pid or limit is less than it's allocation
[PMAGER] > memlim 15 18000
REJECTED: No such pid or limit is less than it's allocation
[PMAGER] > 
```

- memlim <pid> <limit>
- 명령어 memlim 와 함께 pid, limit 를 입력합니다.
- 해당 pid 를 가진 process 에 limit 를 부여합니다.
- Limit 에 0 값을 넣을 경우, 다시 메모리 제한 없음 (UNLIMITED) 상태가 됩니다.
- 존재하지 않은 pid, 이미 할당된 메모리보다 적은 limit 의 경우 REJECT 을 합니다.

```
[PMAGER] > exit
Exiting Pmanager... Bye Bye~
$ 
```

- Exit
- Pmanager 를 종료합니다.

## LWP (Threads)

```
int
main(){
    thread_t t_thread[3]; /* Thread Declaration.
    void* res[3]; /* thread result.
    char t1[] = "THREAD_1";
    char t2[] = "THREAD_2";

    /*Thread Creation
    thread_create(&t_thread[0], thread_test, (void*) t1);
    thread_create(&t_thread[1], thread_test2, (void*) t2);

    thread_join(t_thread[0], &res[0]);
    thread_join(t_thread[1], &res[1]);

    printf(0, "thread_join called: Thread 0 finished with value %d\n", *(int*)res[0]);
    printf(0, "thread_join called: Thread 1 finished with value %d\n", *(int*)res[1]);

    exit();
}

void* thread_test(void* data){
    char* threadName = (char*) data; /* Extract data from thread -> Passed by argument.
    printf(0, "[1] Thread Called! Name: %s\n", threadName);
    thread_exit((void*)1);
    return (void*)0;
}

void* thread_test2(void* data){
    char* threadName = (char*)data;
    printf(0, "[2] Thread called! Name: %s\n", threadName);
    fork();
    thread_exit((void*)1);
    return (void*)0;
}
```

- dev.c
- thread 테스트 코드입니다. 2 개의 thread 를 형성하여, 각각 thread\_test(), thread\_test2 를 실행합니다.
- 각각 string 형태의 인자 t1, t2 를 넘겨주며, 각각의 값은 THREAD\_1, THREAD\_2 입니다.
- User program 으로서, ‘dev’라는 명령어로 실행할 수 있습니다.

```
$ dev
[1] Thread Called! Name: THREAD_1
[2] Thread called! Name: THREAD_2
thread_join called: Thread 0 finished with value 1
thread_join called: Thread 1 finished with value 1
$
```

- 실행 결과입니다.
- start\_routine 에 argument 가 잘 전달되었고, exit 이 호출된 후, retval 에 맞는 값이 thread\_join 에서 전달되었습니다.

## VII. Trouble Shooting & Limitations

### Pmanager

- Pmanager 은 기존의 sh.c 를 참고하여 만들었습니다.
- Command parsing 의 경우, 입력되는 명령어의 수가 한정이 되어있어, parse\_cmd()에서 글자들을 각각 비교하고, 마지막 글자는 공란, 혹은 escape sequence 가 입력되면 인식할 수 있게 하였습니다.
- 명령어들을 실행하면 프로세스가 종료되거나 하는 요소들이 존재했는데, 기존의 exec.c 와 sh.c, 그리고 sh.c 의 backcmd 를 참고하여 실행하고 종료되지 않고, 해당 pmanager 가 background에서 동작할 수 있게 설정했습니다.

### LWP (Threads)

- 사실, 해당 LWP 는 완벽하지 않습니다. 일단, copyuvvm 을 하여 개발한 후, testcode 를 돌려보니, 다음과 같은 문제점이 있습니다.
  1. Thread에서 작업을 한 후, 해당 작업사항이 원래 caller 인 parent 에 반영이 되지 않는 경우 존재.
  2. Return Value 가 저장이 안되는 문제.
- 위 사항 모두 copyuvvm 을 사용하여 형성되었던 문제임으로, 이는 Process 와 Thread 간 memory 가 공유되지 않음을 의미했습니다.
- 저는 곧 thread 의 pgdir 을 parent 의 pgdir 로 넣을 생각을 하였습니다. (newthread->pgdir = curproc->pgdir)
- 하지만 다음 경우, trap 14, 즉 Page fault 가 발생하였습니다. 정상적인 경우에는 나면 안되는 오류입니다.
- 이는 형성되지 않은 공간에 pgdir 을 넣는다고 생각하여, shareuvvm()이라는 system call 을 만들어 해결하려고 했습니다.

```
/** Shareuvvm()
 * Unlike copyuvvm, child will share address space with parent.
 * Implemented based on copyuvvm(), loaduvm(), and walkpgdir()
 */
shareuvvm(pde_t* pgdir, struct proc* parent){
    pde_t *dest;
    pte_t *pkern;
    pte_t *parent_pg;
    uint paddr, flags;

    /* STEP 1) init pgdir -> setupkvm();
     * if(dest == setupkvm()) == 0
     * return 0;

    /* STEP 2) Allocate same page with parent to child.
    parent_pg = parent->pgdir;
    /* Start iteration: Will iterate all parent's pgdir
    /* the copy the valid pgdir.
    for(int i = 0; i < NPDENTRIES; i++){ // Iterate Directories Loop
        if((parent_pg[i] & PTE_P) && (parent_pg[i] & PTE_V)){ // Valid. (Present)
            dest[i] = parent_pg[i]; // Share.

        /* STEP 3), now, it will iterate page table entries
        /* and copy the valid entries, just like outer loop.
        /* Got an idea from walkpgdir and
        pgkern = (pte_t*)P2V(PTE_ADDR(pgdir[i])); // Get addr based on kernel base.

        for(int j = 0; j < NPTENTRIES; j++){ // Iterate Page Table Entries Loop
            /* Copy valid entries.
            if((pgkern[j] & PTE_P) && (pgkern[j] & PTE_V)){
                /* Get physical address of the page
                paddr = PTE_ADDR(pgkern[j]);
                flags = PTE_FLAGS(pgkern[j]);

                /* STEP 4) Based on the physical address calculated,
                /* map those address to child.
                /* ch_paddr = ((pte_t *)P2V(PTE_ADDR(dest[i])))|j; // Based on Stack Overflow.
                if(mappages(dest, (void*)(PTE_ADDR(dest[i]) + PGSIZE * j), PGSIZE, paddr, flags) < 0){
                    goto failed;
                }
            }
        }
    }

    return dest;
failed:
    freevm(dest);
    return 0;
}
```

shareuvvm() – Allocate pgdir, and make pgdir to share parent's page directory.

- 전체적인 구현 의도는 다음과 같았습니다.
- Parent process의 pgdir을 share하는 새로운 pgdir을 만들어 return을 하면, 해당 pgdir을 할당하는 것으로 해결하려 했습니다.
- 구현은 copyuvvm(), loaduvvm(), walkpgdir()등을 참고하였고, 추가로 xv6의 작동 방식에 대한 지식 또한 참고하여 만들었습니다.
- Valid한 parent의 pgdir을 모두 복사하고, 각각의 공간에 대해 kernel space을 만들어 준 후 mapping을 해주는 방법으로 해결하려고 시도했습니다.
  
- 하지만 다음과 같은 방법을 사용하였을 시, panic(remap)을 일으켰고, 결국 오늘날까지 해결하지 못해, 일단은 오류가 안나는 이전 모델로 제출을 했습니다.
- 해당 프로세스는 다른 메모리 공간을 가지게 되고, 실제 xv6에서도 프로세스처럼 다루어져 sbrk 와 같은 memory 관련된 syscall에서 (다른 메모리 공간이기에 메모리가 겹칠 일이 없기 때문에) 따로 예외처리를 하지 않았습니다.
- 일단 thread->parent를 통해 간접적으로 parent의 메모리 공간으로 갈 수 있게 하여 간접적인 공유는 할 수 있게 했습니다. 다만, 해당 적용 사항을 실제 작업 시 (메모리 접근 시 thread->parent로도 접근을 하는 등)의 변화는 결국 적용하지 못했습니다.

### Wrap Up

- 굉장히 아쉬움이 남는 구현 과제였습니다. 처음에 fork()의 copyuvvm()에 대한 잘못된 이해 (아예 다른 공간에 새로운 프로세스의 형성이지만, 그저 pgdir을 공유할 수 있도록 메모리 공간을 복사해주는 기능)로 인해 이와 같은 결과를 만든 것 같습니다.
- 하지만 최선을 다 했기에, 후회는 되지 않았고, 가벼운 마음으로 해당 Project #2 을 마무리 짓습니다.

2019014266 임규민

ELE3021 Operating System – Project #2 Wiki

The End.