운영체제 Project #2

2019082279 김영현

[1. Design]

1-1. process manager

Process manager의 구상은 쉘에서 영감을 받았다. Xv6의 쉘은 사용자가 명령어를 입력하면 해당 명령어에 알맞은 기능을 실행하는 인터페이스 역할을 한다. Process manager 또한 shell에서 실행이 되면, 사용자로부터 프로세스와 관련된 명령어 kill, memlim, execute 등을 입력 받으며 적절한 기능을 수행한다. 그렇게 하기 위해선 다음과 같은 과정이 필요하다.

1. shell에서 프로세스 매니저 역할 pmanager 실행

2. pmanager가 루프를 돌며 사용자의 입력을 받음

3. 사용자의 **입력을 parsing해서 분석 후, 잘못된 입력시 예외처리**

4. 입력에 맞는 기능 수행, 다시 2번 수행

쉘과 마찬가지로 console을 열고 파일 디스크립터를 받는 과정을 반복하여 표준 입력, 출력, 오류 출력을 위해 3번 받는다. 그 다음, 사용자의 편의를 위해 pmanager의 명령어를 기록한 mannual을 출력한다. 그 이후, 위와 같은 방식으로 작동한다.

1-2. exec2

Xv6의 exec와 매우 유사하다. Xv6의 exec는 하나의 스택용 페이지와 하나의 가드용 페이지를 할당하여, 총 2개의 페이지를 할당 받는다. 그후, 해당 새로 할당 받은 pgdir, sz 로 프로세스를 새 프로세스로 탈바꿈 시키며 해당 프로세스가 수행하도록 한다, 예전 프로세스가 가지고 있던 pagetable은 oldpgdir에 저장하여 할당 해제한다. Exec 함수를 사용하여 새로운 프로그램을 실행하면, 프로그램 간의 전환 및 새로운 환경에서의 실행이 가능해진다.

Exec2는 기존의 exec에서 할당받는 stack page의 개수를 원하는 만큼 늘린 것에 불과하다. 가드용 페이지는 동일하게 1개를 할당받는다.

1-3. setmemorylimit

Setmomorylimit은 특정 프로세스에 대해 할당받을 수 있는 메모리의 최대치를 제한하는 것이다.

여기서 메모리가 추가적인 메모리를 할당받을 때 초과하지 않는 것이다. 여기서 추가적인 메모리를 할당받는 경우를 프로세스에게 메모리를 더 할당하는 sbrk 시스템 콜을 호출할 때라 해석하였으며, 스레드를 만들 때와 같은 상황은 스레드 또한 프로세스이며, 프로세스가 처음 생성될 때는 할당받는 제한이 없기 때문에 sbrk만 고려했다. **Sbrk의 경우, proc.c에 정의된 growproc 을 통해서 메모리를 더 할당하거나 감소시킨다. 따라서 growproc에서 메모리를 limit를 넘어서서까지 할당하려하면 할당하지 않는 방식으로 구현했다.**

1-4. LWP(Light-weight process)

프로세스는 서로 독립적으로 실행되고, 자원을 공유하지 않으며 서로 별개의 주소 공간과 파일 디스크립터를 가진다. 하지만 LWP의 경우, 스레드와 유사하게 다른 LWP와 자원과 주소 공간 등을 공유한다.

여기서 공유하는 주소 공간은 페이지 테이블을 의미한다. 페이지 테이블은 가상 주소를 물리 주소로 변환하기 위해 사용된다. Xv6에서는 각각의 프로세스에게 페이지 테이블을 할당한다. Vm.c를 보면 알 수 있듯이, 각각의 가상 주소에서 0 부터 KERNBASE(0x80000000) 까지는 유저 가상 메모리에 해당하며, text data stack heap 영역이 존재한다. Xv6의 경우 이러한 프로세스에 메모리를 할당하는데 있어서 stack이나 heap 등 명확한 구분 없이 프로세스의 sz값을 pgsize의 배수만큼 가상 주소를 늘려가며 할당하는 것을 발견할 수 있었다.

이러한 기존의 xv6 방식과 유사하게, 프로세스의 스레드를 생성하면 가상 메모리 상의 크기를 의미하는 sz값을 증가시키는 방식을 채택하였고, 스레드 또한 프로세스와 유사한 기능들을 수행하며, exec를 실행시 스레드가 프로세스로 변환되어야 했고, 어차피 스케줄러의 process table의 구성 요소는 NPROC으로 정해져 있으며, 스케줄러 상에 돌아갈 때 스레드가 하나의 프로세스처럼 구동되더라도 문제가 없다고 판단하였다. 그래서 프로세스 구조체에 스레드의 기능을 담당할 멤버 변수를 추가했다.

스레드의 create은 프로세스의 생성과 유사하다. 프로세스는 결국 fork와 exec 과정을 통해서 프로세스 테이블 상의 UNUSED 프로세스를 RUNNABLE 독립적인 프로세스를 만들어낸다. 스레드 또한 이와 유사한 방식으로 진행하였다. Thread_create 또한 마찬가지이다. Allocproc을 호출하여 페이지 테이블 상의 unused 프로세스를 찾아내어 해당 프로세스를 스레드에 맞게 변환시키는 과정을 거친다. 프로세스의 생성과의 차이점이라면 스레드는 페이지 테이블을 프로세스와 공유하기 때문에, 스레드의 pgdir 값을 메인 프로세스의 값으로 복사하는 것이다. 여기서 스레드는 스레드만의 id를 가지는 것이 프로세스와 구별하는데 편하며 thread join에서의 인자 타입을 만족시키기 쉽다고 판단하여 pid와 별개로 tid는 스레드 생성시에 할당받는 것으로 했다. 스레드는 메인 프로세스와 pgdir를 공유한다고 판단해서 복사해주는 과정을 거친다. 나머지는 fork와 매우 유사하게 구현한다. 메인 프로세스로부터 정보를 copy한 이후로, stack page를 할당하는 과정으로 넘어간다. **Sz 값을 늘리면서 할당하기 이전에 메인 프로세스의 spare array에 저장되어 있는 가상 주소상의**

빈공간이 존재하다면, 그 공간에 할당하고 그렇지 않다면 sz 값을 늘려서 할당하는 방식으로 수행한다. 이후, 스택 상에 argument인자 값과 fake return PC값을 복사해준다. 이후, 스레드 자체 스레드 기능을 위한 값들을 조정한다. 이후, 해당 스레드를 할당하는데 sz 값에 영향을 주므로, sz 값을 모든 스레드가 동일하게 만들어주는 과정을 거친다.

스레드 exit 또한 프로세스의 exit과 유사하다. 공유하는 page table을 제외하고 다른 자원들을 할당 해제하고, 메인 스레드이자 프로세스에게 종료한다는 것을 알리고 스케줄링 된다.

스레드 join 도 마찬가지로 프로세스의 wait과 유사하다. Wait할 thread의 id를 받고 해당 스레드만 종료를 기다리는 것에서 부모 프로세스가 자식 프로세스의 종료를 기다리는 기존 프로세스의 wait과 차이가 있을 뿐이다. 추가로, 스레드가 종료되었다면, **스레드에 할당된 스택 페이지를 회수하고 커널 스택 등의 자원들을 해제한다. 이후, 메인 스레드의 spare array에 해제한 가상 주소를 저장한다.**

스레드를 fork 하는 경우, 가상 메모리를 pgdir 기준으로 pgsize씩 늘려가면서 copyuvm하기 전에, 해당 스레드의 메인 스레드가 spare array에 요소를 가지고 있는지 확인한다. Spare array는 스레드에 가상메모리를 해제할 때 빈 공간의 시작 주소를 저장하는 array이다. 해당 spare array는 circular queue를 배열로 구현한 형태이다**. 0부터 sz까지 빈 공간이 존재하지 않는다면 원래 하던 방식대로 copyuvm을 하고, 빈 공간이 존재한다면, copyuvmTHD인 spare array에 기록된 주소를 제외하고 copy하는 함수를 이용한다. 이후, copyuvmTHD를 이용했다면 spare array또한 복사해준다.** 프로세스를 copyuvmTHD방식으로 spare array에 있는 가상 주소를 제외하고 복사했기 때문에, **wait에서 pgdir를 free해줄 때, 기존의 freevm이 아닌, spare array를 고려한 freevmTHD를 이용한다.**

스레드를 exec를 할 경우**, 해당 스레드를 제외한 나머지 스레드를 thread_cleaner를 호출해 할당 해제 시키고 UNUSED상태로** 만든다. 이후, 이때도 스레드의 메인 스레드의 spare를 고려한 **freevmTHD를 이용해서 이전의 페이지 테이블을 free해준다**.

스레드를 sbrk 호출할 경우, 스레드의 메모리 할당을 단순히 스택 페이지를 늘리면서 전체 메모리를 늘리고 sz값을 늘렸기 때문에, 어차피 같은 **페이지 테이블을 공유하기 때문에 sz 값을 동일하게 해주는 thread_family_sz_same을 호출한다.**

스레드를 kill 할 경우, 메인 프로세스의 pid를 공유하기 때문에 pid가 동일한 스레드의 killed를 1로 만든다. 이때, flag를 활용하여 일반적인 kill과 구별할 수 있도록 한다.

[2. Code Summary]

```
typedef int thread_t;/*for project02 thread id*/
typedef struct proc Thread;/*for project02 thread*/
```

(proc.h) 명세의 thread_t type과 proc 구조체를 스레드에도 활용하기 위해 typedef했다.

```
  /*project02*/
  int mem_limit;              // Memory limit of process
  int sPage_cnt;              // Number of assigned stack pages
  thread_t tid;               // Thread ID
  void *retval;               // Thread return value
    /*for 0~sz smoothe*/
  struct proc *tmaker;        // Thread Maker
  uint spare[NTHREAD];        // 0~sz spare page virtual address - management
    int front;                  // front spare array
    int rear;                   // rear spare array
  uint thdva;               //thd allocated page start address
```

(proc.h) project2를 하면서 proc 구조체에 새로 추가해준 멤버 변수들이다. Mem_limit과 spage_cnt는 process manager를 구현하기 위해 사용했다. 나머지는 스레드의 구현을 위해 추가했다. Tmaker는 스레드들의 메인 스레드이자 프로세스를 가리킨다. 메인 스레드는 자기 자신을 가리킨다.

Spare array는 circular queue 형태로, 스레드를 create하면서 virtual memory를 할당하는 과정에서 sz 값을 늘리게 되는데, thread_join의 명세를 만족시키기 위해 스레드를 deallocuvm하는 과정을 거치다 보면 가상 주소 상의 0부터 sz 사이에 메모리가 할당 해제된 공간이 존재하게 된다. 따라서 이러한 빈 공간을 그대로 fork해서 copyuvm을 호출할 경우, 메모리 할당 해제된 공간을 가리키게 되어 오류가 발생가능하다. 또한 무한정 sz를 늘리는 방식은 0부터 sz 사이에 빈공간이 있는데도 불구하고 메모리를 낭비한다 생각하여 만든 배열이 바로 spare array이다.

Spare array는 기본적으로 NTHREAD크기이며, 이는 process table의 사이즈를 고려한 결과이다. Thdva는 스레드 각각의 할당된 가상 주소이다.

```
int nextTid = 1;/*for project02*/
```

(proc.c) 스레드의 id를 따로 표현하기 위해서 만들었다.

```
  /*for project02*/
  p->mem_limit = 0;
  p->sPage_cnt = 0;
  p->tid = 0;
  p->retval = 0;
  p->front = 0;
  p->rear = 0;
```

```
  p->tmaker = p;
```
(proc.c – allocproc) 멤버변수를 초기화해준다.

```
    if(curproc->mem_limit > 0 && sz > curproc->mem_limit) {
```
(proc.c – growproc) mem_limit 값 보다 크면 다시 alloc한 값을 되돌린다.

```
    thread_family_sz_same(curproc->pid, sz);
```
(proc.c – growproc) 스레드가 growproc후, 변경된 sz 값을 동일한 메인 스레드를 공유한 스레드의 sz값을 모두 동일하게 만든다.

```
int
setmemorylimit(int pid, int limit)
```
(proc.c – setmemorylimit) process manager에서 memlim 명령을 수행한다. ptable에서 pid에 해당하는 메인 스레드를 찾고 메인 스레드의 sz값을 기준으로 memory limit 성공 여부를 판단한다. sz 값이 limit보다 작다면 프로세스의 mem_limit값을 limit로 저장한다.

```
void
listProc(void){
```
(proc.c – listProc) process manager에서 list 명령을 수행한다. ptable에서 스레드가 아니고 RUNNABLE이거나 RUNNING인 프로세스를 찾아서 정보를 출력한다. 할당받은 메모리 크기는 스레드가 할당 받은 값까지 포함해서 출력하며, 어차피 sz 값을 늘리는 방식을 채택했기 때문에 sz 값을 출력한다.

```
  if(!curproc->tid && curproc->front == curproc->rear){//if not has spare
                                                       list, there is no mem leak
    if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
```
(proc.c – fork) 일반적인 fork의 경우이다. Spare array가 비어 있기 때문에 copyuvm을 해준다.

```
 }else{//if there is spare list, copy without that area
    struct proc *procmaker = curproc->tmaker;
    if((np->pgdir = copyuvmTHD(procmaker->pgdir,procmaker->sz, procmaker-
                               >spare, procmaker->front, procmaker->rear))==0){
생략
 if(casefr > 0){//rear > front
  생략
}else if(casefr < 0){ //front > rear
```
(proc.c – fork) spare array가 비어 있지 않은 경우의 fork이다. copyuvmTHD를 이용한다. 이때 spare array는 circular queue 방식이기 때문에 spare array를 fork된 자식 프로세스에 복사해주기 위해서 front와 rear를 비교해서 경우에 맞게 복사해준다.

```
 freevmTHD(p->pgdir, p->tmaker->spare, p->tmaker->front, p->tmaker->rear);
```
(proc.c – wait) 프로세스의 spare array가 존재할 수 있기 때문에 일반 freevm이 아니라 freevmTHD를 이용한다

```
      killFlag = 1;//
    }
  }
  if(killFlag){
    release(&ptable.lock);
    return 0;
  }
```

(proc.c – kill) 일반적으로 kill을 할 경우, p->killed = 1을 하고 RUNNABLE로 만든 다음 바로 종료되는데 그러면 스레드의 경우 모든 스레드를 kill할 수 없기 때문에 killFlag로 스레드 kill과 일반 프로세스 kill까지 모두 만족시켰다.

```
.   /*********************************

  thread_create- copy proc to thread
  *********************************/
```

(proc.c - thread_create) thread_create 함수에서 fork부분과 유사한 부분이다. 어차피 스레드와 프로세스를 fork하는 것의 차이는 pgdir 를 공유하느냐 안하느냐의 차이라고 판단했기 때문에 해당 부분을 제외하고는 fork와 비슷하다. 스레드 또한 프로세스이기 때문에 스레드의 parent는 메인 프로세스의 parent와 동일하며 pid는 메인 스레드의 pid이다.

```
  /*********************************
  thread_create-allocate stack pages
  *********************************/
```

(proc.c – thread_create) thread_create 함수에서 exec부분과 유사한 부분이다. 스레드는 메인 스레드와 페이지 테이블을 공유하지만 별도의 스택 페이지를 할당 받아야 한다고 판단했다. 페이지를 할당하는 exec에서 영감을 받았다.

```
  hasSpare = (front != rear);
  if(hasSpare){
    sz = curproc->spare[front];
  }else{
    sz = curproc->sz;
  }
```

(proc.c – thread_create) stack page 할당 부분에서 메인 스레드의 spare array를 확인해서 중간에 빈 공간에 할당해줄지, sz 값을 늘리면서 할당해줄지 정한다.

```
  ustack[0] = 0xffffffff;  // fake return PC
  ustack[1] = (uint)arg;
```

(proc.c – thread_create) arg인자를 sp기준으로 저장해준다.

```
  /*setting start_routine and thread info*/
  if(hasSpare){
    curproc->sz = curproc->sz;//not change
    curproc->front = (front+1) % NTHREAD;
  }else{
    curproc->sz = sz;//added version update
```

```
  }
```
(proc.c – thread_create) sz 값을 늘렸냐 안늘렸냐 메인 스레드의 sz정보를 update 해주거나 안해주거나 한다.

```
  nThread->thdva = sz - 2*PGSIZE;

  nThread->tf->eip = (uint)start_routine;
  nThread->tf->esp = sp;
  nThread->tid = nextTid++;
  nThread->tmaker = curproc;
  *thread = nThread->tid;/*set thread argument*/
  nThread->state = RUNNABLE;
  thread_family_sz_same(curproc->pid, curproc->sz)
```
(proc.c – thread_create) 이후 스레드 정보를 정해준다. 그리고 sz값을 스레드에 모두 공유한다.

```
void
thread_exit(void *retval){/*thread : exit*/
```
(proc.c – thread_exit) 프로세스의 exit과 매우 유사하다. Retval을 지정해준다.

```
int
thread_join(thread_t thread, void **retval){
```
(proc.c – thread_join) 프로세스의 wait과 매우 유사하다. 다만 thread인 스레드의 id를 채널로 하여 sleep하게 된다.

```
        deallocuvm(thd->pgdir, tmpva + 2*PGSIZE , tmpva);
        curprocPar->spare[curprocPar->rear] = tmpva;
        curprocPar->rear = (curprocPar->rear+1) % NTHREAD;
        thd->state = UNUSED;
```
(proc.c – thread_join) kfree, deallocuvm을 통해 스레드의 자원을 할당 해제하고, 빈공간을 메인 스레드의 spare array에 저장해준다.

```
void
thread_cleaner(Thread *curthd){
```
(proc.c - thread_cleaner) 메인 프로세스가 exit될 때와 exec를 실행했을 때 사용된다. 커널 스택을 할당 해제해주고 메모리는 thread_cleaner를 호출할 경우 이후, freevm을 하는 경우기 때문에 신경 쓰지 않는다. Tid는 어차피 allocproc 에서 다시 초기화 해주기 때문에 신경쓰지 않는다.

```
int
thread_family_sz_same(int pid, uint sz){
```
(proc.c – thread_family_sz_same) 메인 프로세스를 공유한 스레드들의 sz 값을 모두 동일하게 해준다.

```
void freevmTHD(pde_t *pgdir, uint *spare, int front, int rear) {
```
(vm.c – freevmTHD) 메인 스레드의 spare array를 고려하여 deallocuvmTHD를 호출한다. Spare 가 비어있다면 일반 freevm을 해준다.

```
int deallocuvmTHD(pde_t *pgdir, uint oldsz, uint newsz, uint *spare, int
front, int rear) {
```

(vm.c – deallocuvmTHD) 스레드의 spare array를 참조하여 spare array에 있는 virtual address를 제외하고 할당 해제한다.

```
int is_spare_addr(uint addr, uint *spare, int front , int rear) {
```

(vm.c – is_spare_addr) 스레드의 spare array에 들어있는지를 알려주는 helper function이다.

```
pde_t*
copyuvmTHD(pde_t *pgdir, uint sz, uint* spare, int front, int rear)
```

(vm.c – copyuvmTHD) spare array를 고려하여 spare array에 저장된 virtual address를 제외하고 pgdir부터 단방향으로 할당해준 pgdir를 리턴한다.

```
  /*for proejcet 02*/
  uint* spare = curproc->tmaker ->spare;
  int front = curproc->tmaker->front;
  int rear = curproc->tmaker->rear;
  thread_cleaner(curproc);
  생략
  switchuvm(curproc);
  freevmTHD(oldpgdir,spare, front, rear);
```

(exec.c, exec2.c) 스레드가 존재하다면 자기자신을 제외하고 모두 자원을 반환한다. 이후 새롭게 프로세스로 단장한 프로세스를 실행시키고 기존 페이지 테이블은 기존 spare를 고려하여 할당 해제한다.

```
  if ((sz = allocuvm(pgdir, sz, sz + (stacksize+1) * PGSIZE)) == 0)
```

(exec2.c) 지정한 stacksize만큼 추가 할당해준다.

```
int
getcmd(char *buf, int nbuf)//-sh.c
```

(pmanager.c – getcmd) buf를 비우고 입력받는다.

```
  while((fd = open("console", O_RDWR)) >= 0){
```

(pmanager.c) 파일 디스크립터를 open한다

```
  // following mannual
  printf(1, "\n");
  printf(1, "[You must follow this format]\n");
  printf(1, "a.   exit\n");
  printf(1, "b.   list\n");
  printf(1, "c.   kill <pid>\n");
  printf(1, "d.   execute <path> <stacksize> (stacksize [1,100])\n");
  printf(1, "e.   memlim  <pid> <limit>\n");
  printf(1, "\n");
```

(pmanager.c) 매뉴얼을 출력한다.

```
  // Read and run input commands.
  while(getcmd(buf, sizeof(buf)) >= 0){
  명령어 기능 수행 생략
    else{
      printf(2, "given input is not valid: only 'exit', 'list', 'kill <pid>',
'execute <path> <stack size>', 'memlim <pid> <limit>'\n");
    }//case end
    printf(1, "\n");
```

(pmanager.c) getcmd를 반복해서 수행하며 입력받는다. 잘못된 입력을 했을 경우, 경고를 출력한다.

```
    if(buf[0] == 'e' && buf[1] == 'x' && buf[2] == 'i' && buf[3] == 't' &&
/*exit*/
            buf[4] == '\n' )
    else if(buf[0] == 'l' && buf[1] == 'i' && buf[2] == 's' && buf[3] == 't'
&& /*list*/
      buf[4] == '\n')
```

(pmanager.c) exit을 입력받았을 경우, 종료시킨다. List를 입력받았을 경우, proc.c의 listProc을 호출한다.

```
    else if(buf[0] == 'k'&& buf[1] == 'i' && buf[2] == 'l' && buf[3] == 'l'
&&  /*kill <pid>*/
            buf[4] == ' ')
    else if(buf[0] == 'e' && buf[1] == 'x' && buf[2] == 'e' && buf[3] == 'c'
&& buf[4] == 'u' && buf[5] == 't' && buf[6] == 'e' && /*execute
            buf[7] == ' ')
    else if(buf[0] == 'm' && buf[1] == 'e' && buf[2] == 'm' && buf[3] == 'l'
&& buf[4] == 'i' && buf[5] == 'm' && /*memlimit
            buf[6] == ' ')
생략
      /*pid scan*/ - kill
      while(buf[buf_idx] != ' ' && buf[buf_idx] != '\n'){
      /*path scan*/ - execute
      while(buf[buf_idx] != ' ' && buf[buf_idx] != '\n'){
      /*pid scan*/ - memlim
      while(buf[buf_idx] != ' ' && buf[buf_idx] != '\n'){
생략
      /*blank space*/
      option_idx = 0;
      buf_idx++;
생략
      /*stack size scan*/ - execute
      while(buf[buf_idx] != ' ' && buf[buf_idx] != '\n'){
      /*limit scan*/ - memlim
      while(buf[buf_idx] != ' ' && buf[buf_idx] != '\n'){
```

(pmanager.c) 위에서부터 parsing하는 단계로, 가각 kill, execute, memlim의 경우에 해당한다. Idx

를 1씩 더해가며 명령어를 parsing하고 -> pid나 path , pid를 parsing -> 빈공간 -> arg가 더 있는 경우: size, limit parsing -> 각각의 경우에 조건을 만족시키지 못했다면 option_not_valid flag를 1로 만들고, 이후 해당 에러를 처리한다.option_not_valid가 0이라면 해당 명령을 수행한다.

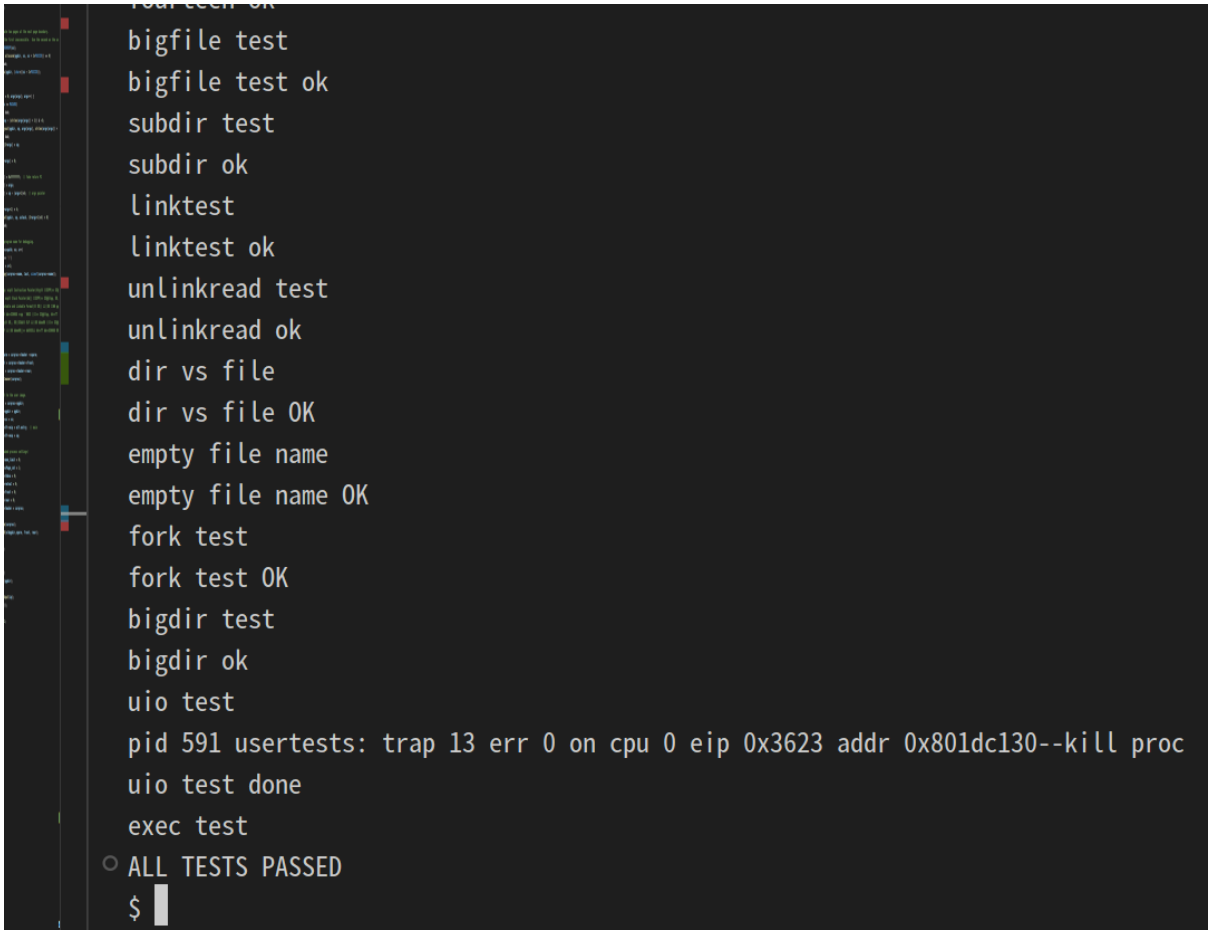
```
        pid *= 10;
        int adder = buf[buf_idx] - '0';
        if(adder < 0 || adder > 9){/*option each number of digit check*/
          option_not_valid = 1;
          break;
        }
        if(++option_digit > 9){/*option digit check 000,000,000*/
          option_not_valid = 1;
          break;
        }
        pid += adder;
        buf_idx++;
```

(pmanager.c) 숫자를 앞에서부터 읽기 때문에 해당 수를 입력 받기 위해 다음과 같이 adder를 사용해 10씩 곱해 더하는 방식으로 처리했다.

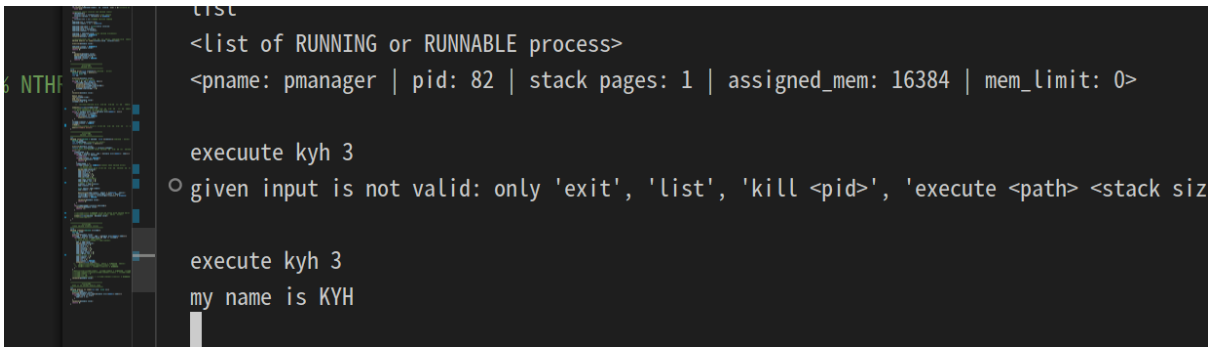
```
      /*normal execute*/
      int pid = fork();
      if(!pid){//for child pmanager
        int cpid = fork();
```

(pmanager.c) execute일 경우, background에서 실행이 가능하도록 만들기 위해 fork를 두 번 해줬다. 자식의 자식 프로세스가 exec를 수행한다.

[3. Reesult]



Usertests 통과 화면이다.



Pmanager 에서 execute 화면이다. Kyh는 my name is KYH를 출력하는 유저 프로그램이다.

```
$ pmanager

[You must follow this format]
a.   exit
b.   list
c.   kill <pid>
d.   execute <path> <stacksize> (stacksize [1,100])
e.   memlim  <pid> <limit>

list
<list of RUNNING or RUNNABLE process>
<pname: pmanager | pid: 82 | stack pages: 1 | assigned_mem: 16384 | mem_limit: 0>
```

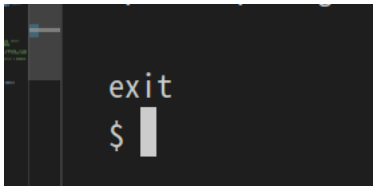
Pmanager 에서 list 화면이다. 메인 프로세스만 출력된다.

```
list
<list of RUNNING or RUNNABLE process>
<pname: pmanager | pid: 82 | stack pages: 1 | assigned_mem: 16384 | mem_limit: 0>

memlim 82 2000
setmemorylimit fail

memlim 82 20000
setmemorylimit success

list
<list of RUNNING or RUNNABLE process>
<pname: pmanager | pid: 82 | stack pages: 1 | assigned_mem: 16384 | mem_limit: 20000>
```

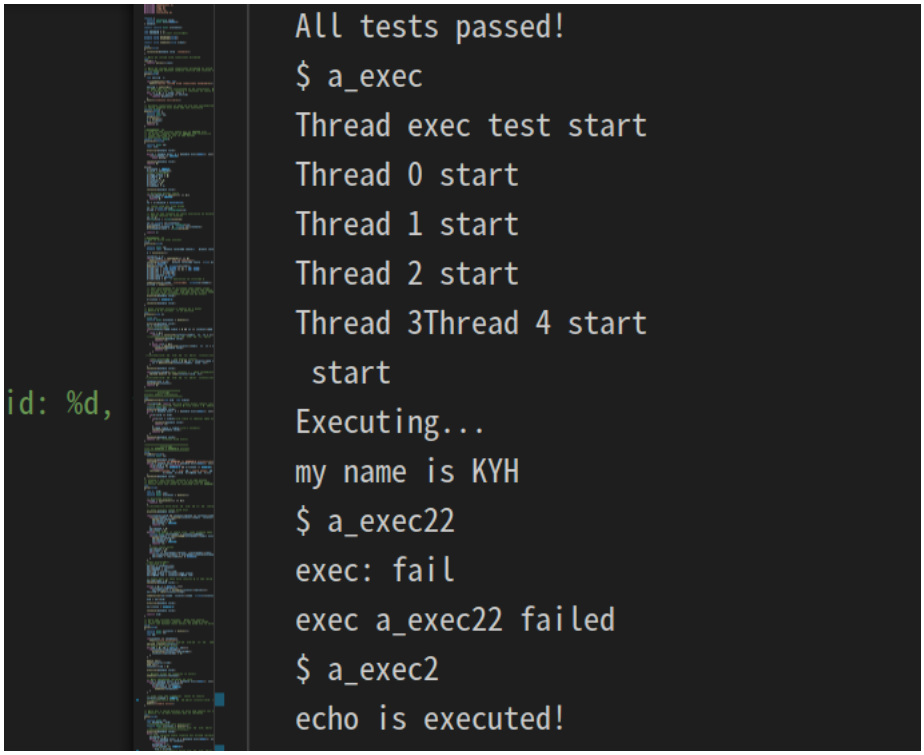
pmanager에서 memlim을 한 화면이다.

```
<pname: pmanager | pid: 82 | stack pages: 1 | assigned_mem: 16384 | mem_limit: 20000>

kill 82
zombie!
$
```

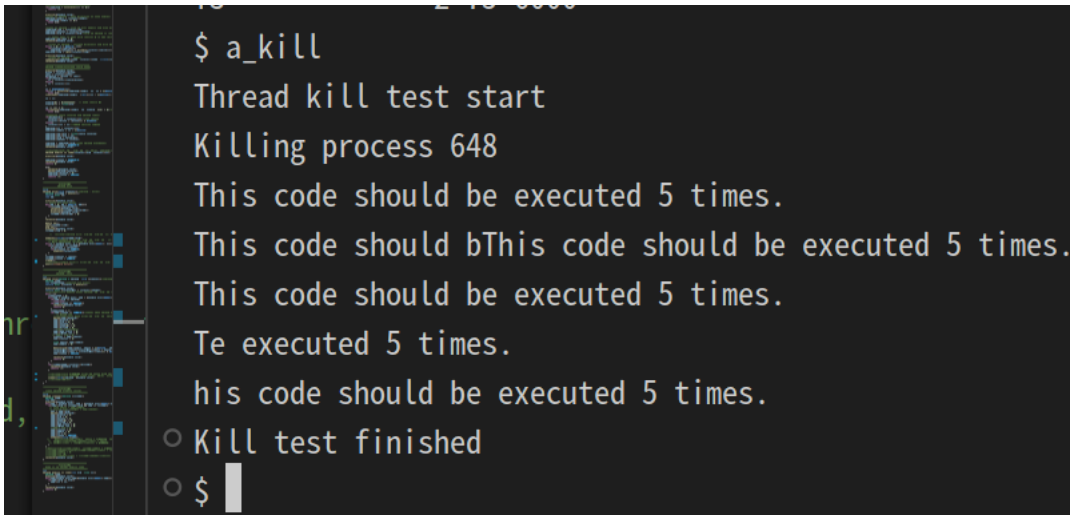
pmanager에서 pmanager를 kill한 화면이다.



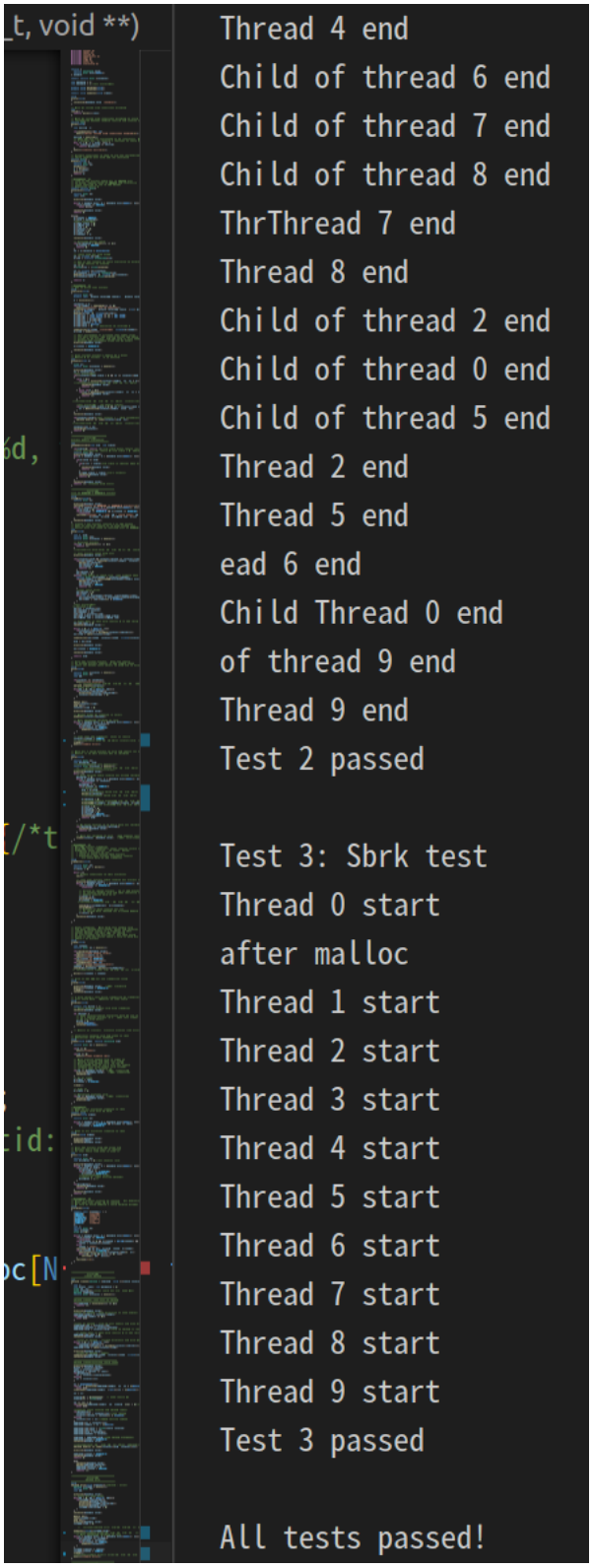
pmanager에서 exit한 화면이다.



a_exec, a_exec2를 통과한 화면이다. 스레드를 만들어서 exec를 수행한다. a_exec는 kyh를 exec한다. 메인 스레드는 sleep하다가 exec호출로 종료된다. a_exec2는 echo를 exec한다. Echo에 echo is excuted를 인자로 넘겨준다. 메인 스레드는 thread_join을 하다가 스레드의 exec 호출로 종료된다.



a_kill을 통과한 화면이다. 스레드를 5개 만들고 한 스레드를 kill한다.



a_test를 통과한 화면이다. Test3까지 있으며 test1은 단순히 스레드들을 생성하고 종료, test2는 스레드의 fork가 정상 작동 되는지 테스트하고, test3는 스레드에 sbrk가 정상 작동되는지 테스트한다.

a_test2를 통과한 화면이다. 스레드들을 create하여 forktest, sbrktest, killtest, pipetest, sleeptest를 수행한다.

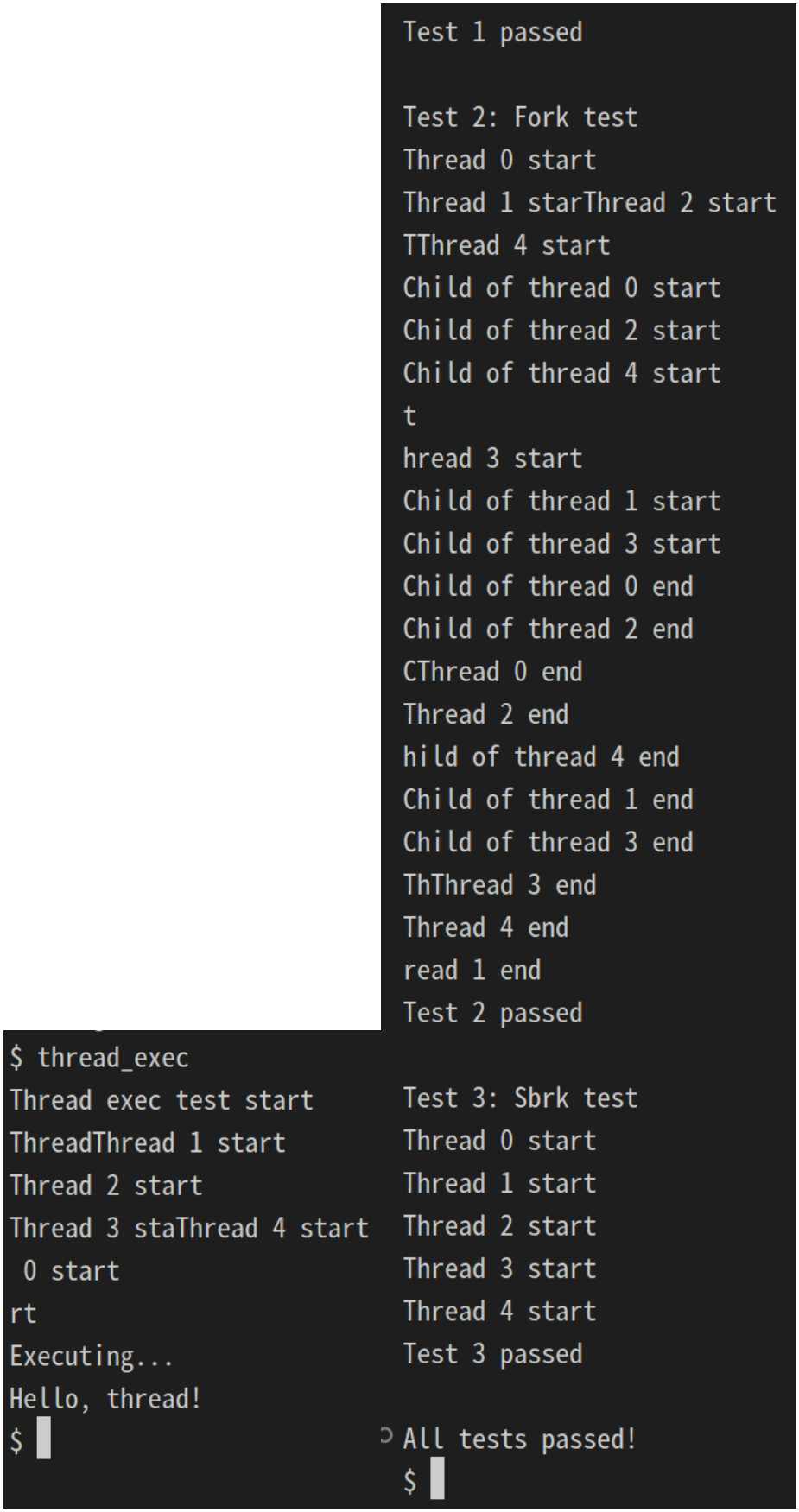
Xv6자체 내장 usertests를 제외한 테스트 코드는 다른 곳에서 가져와서 명세에 맞게 수정했다.

```
$ thread_exit
Thread exit test start
Thread 0 start
Thread 1 start
TThread 3 start
Thread 4 start
hread 2 start
Exiting...
$
```

Thread_exit을 테스트한 결과이다.

```
Hello, thread!
$ thread_kill
Thread kill test start
Killing procThis code should be executed 5 times.
This code should be executed 5 times.
ThThis code should be executed 5 times.
ess 16
This code should be executed is code should be executed 5 times.
5 times.
Kill test finished
$
```

Thread_kill을 test한 결과이다.



각각 thread_exec와 thread_test를 실행한 결과이다.

[4. Trouble Shooting]

스레드를 어떻게 구현할지가 문제였다. 기존의 proc구조체에 멤버변수를 추가해서 구현할지, 따로 스레드 구조체를 만들지, 아니면 proc 구조체에 스레드 배열을 추가할지 결정하기 쉽지 않았다. 많은 고민 끝에 xv6의 기존 작동 방식과 유사하게 스레드를 구현하고자 했다. 어떤 방식이든 어차피 xv6 자체의 ptable은 크기가 정해져 있고 xv6 구현상의 성능 부분에서 한계가 있는 것은 동일하다고 판단하여 어렵게 돌아갈 필요가 없다고 판단했다. 또한 스레드 자체를 프로세스로 취급하며, 기타 구현 과정에서 스레드와 프로세스를 넘나드는 과정이 존재하기 때문에 기존 proc 구조체를 이용하는 것이 적합한 것 같았다. 다만 아쉬운 점은 성능적으로는 스레드 배열을 프로세스에 추가해줘서 구현하는 것이 더 좋지 않을까라는 생각이 들기 때문에 이러한 스레드 설계 부분에서 적당히 타협한 것과 뒤늦게 알아차린 것이 아쉽다.

또한 스레드의 스택 페이지 할당, 해제를 어떻게 할지 고민이었다. Xv6의 경우 프로세스마다 page table을 할당하는데 stack heap code data로 사용 가능한 부분은 virtual address 0부터 KERNBASE(0x80000000) 까지이다. Xv6 자체적으로 프로세스에 메모리를 할당할 때 allocuvm copyuvm 등 pgdir기준으로 증가하는 방향으로 차례대로 증가시키게 설계되어 있었다. Xv6 기존 구현 방식과 유사하게 만들고자 했다. 그래서 처음엔 단순히 sz값을 증가시켜가며 index처럼 활용하여 PGSIZE의 배수만큼 증가시켜 할당했다.

하지만 이는 fork부분에서 copyuvm을 할 경우 문제가 발생했다. 명세에 부합하려면 thread_join시 자원을 할당 해제 해줘야 하는데 thread_join에서 할당 해제할 경우 virtual address 상의 0부터 sz까지 중에서 할당 해제된 부분이 존재하기 때문에 발생한 문제이다. 이 부분은 메인 스레드에 spare 배열을 추가하여 해결했다. 0부터 sz까지 할당 해제된 주소를 저장하는 것으로 해결했다. 또한 그에 맞게 vm.c에 정의된 메모리 할당 복사 해제 함수를 추가해줬다.