Operating Systems
Project #2 Wiki [유민수 교수님]

2019087192 이예진

목차

1. Design

이번 과제의 디자인은 크게 2 가지로 나눠서 설명할 수 있습니다. (헷갈림을 방지하기 위해서 thread에서도 exec함수를 exec2로 변경하여 진행하였으며 테스트 코드를 수정하였습니다.)

1) Pmanager 2) Thread

1.1 Pmanager에서의 exec2설계

스텍용 페이지를 여러개 할당 받을 수 있게 하는 시스템 콜 exec2를 새롭게 함수로 만들었습니다. 첫 번째와 두 번째 인자의 의미는 기존 exec 시스템 콜과 동일하며 기본 구조 역시 exec과 동일합니다. stacksize라는 인자를 추가적으로 받아오며 이는 스텍용 페이지를 의미합니다.

또한 proc 구조체에 stackpagenum이라는 인자를 만들어 stacksize를 저장해줍니다.

```
// 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 + (stacksize + 1)*PGSIZE)) == 0) //stacksize + guardpage(1)
  goto bad;
clearpteu(pgdir, (char*)(sz - (stacksize + 1)*PGSIZE));
sp = sz;
```

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->stackpagenum= 1;
  p->threads[0].state = EMBRYO;
  p->threads[0].tid = nexttid++;
  p->limit = 0;
  release(&ptable.lock);
```

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

  //if less than 1 or larger than 100, return -1

  if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0 || argint(2,&stacksize)
  < 0){
    return -1;
  }
  if(stacksize < 1 || 100 < stacksize)
    return -1;

  memset(argv, 0, sizeof(argv));
  for(i=0;; i++){
    if(i >= NELEM(argv))
      return -1;
    if(fetchint(uargv+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);
}
```

exec2시스템 콜 생성

1.2 Pmanager에서의 명령어 설계

• List: 현재 실행중인 프로세스들의 정보를 출력합니다. 각 1) 프로세스의 이름, 2) pid, 3) 스택용 페이지의 개수, 4) 할당 받은 메모리의 크기, 5) 메모리의 최대 제한

이 있습니다.

```
// Per-process state
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[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  struct thread threads[NTHREAD]; //threads in process
  thread_t curtid;              //currund thread's id
  int limit;                    //limit
  int stackpagenum;             //stack page number
  uint  user_stack_pool[NTHREAD]; // user_stack_pool
  void* retval;                // Temporary save return value
};
```

이를 위해 limit, stackpagenum, limit를 proc구조체 안에 추가해주었습니다.

각각은 proc의 name, pid, stackpagenum, sz, limit를 나타냅니다.

• Kill: pid를 가진 프로세스를 kill합니다. Kill시스템 콜을 사용하면 됩니다. 성공 여부를 출력합니다. 이는 proc.c에 있는 kill 시스템 콜을 그대로 사용합니다. 따라서 pmanager.c에 그 kill함수를 그대로 불러 오기만 하면 됩니다.

• Execute: execute <path> <stacksize> 이 함수를 통해 setmemorylimit 함수를 호출하게 됩니다. syscall.h, usys.S, user.h, defs.h 에 setmemorylimit 함수를 정의합니다. Setmemorylimit 함수에서는 limit 를 지정해 줍니다. 이 때 pid 가 exist 하는지 여부를 check_point 를 넣어서 판단하고, received memory(p->sz)가 limit 보다 작은지 판별합니다.

• Memlim: memlim <pid> <limit>

Pid를 가진 프로세스의 메모리 제한을 limit로 설정합니다. 이 때 limit는 exec2함수 및 에서 초기화 해주고 그외 proc.c에서 proc 구조체를 초기화 할 때 함께 초

기화 해줍니다.

• Exit: pmanager를 종료합니다. 이는 proc.c에 있는 exit함수를 그대로 사용하면 됩니다. 따라서 kill함수와 같이 그대로 불러와주었습니다.

proc.c에서는 getcmd라는 함수를 통해 cmd에서 입력되는 것들을 받아올 수 있게 제작하였습니다.

```
int
getcmd(char *buf, int nbuf)
{
  printf(2, "> ");
  memset(buf, 0, nbuf);
  gets(buf, nbuf);

  if(buf[0] == 0)
    return -1;
  return 0;
}
```

Pmanager.c

1.3 Thread

Xv6는 프로세스 단위로 작동 되는데 이를 thread를 이용하는 구조로 바꾸었습니다. 한 개의 process 당 64의 thread를 할당 하였습니다. 같은 process에 속하기 때문에 속한 thread는 모두 같은 address space를 공유합니다. 이외에 각각의 thread를 위해 stack을 할당하였습니다. 같은 프로세스의 thread간의 context switch address space를 공유하기 때문에 stack 영역만 변경해주면 됩니다. 따라서 context switch cost가 낮습니다. Thread를 구조체로 만들고, 이를 리스트 형태로 proc구조체에 할당하였습니다. Threads[0]에는 main thread를 넣었습니다. 또한 thread가 새롭게 실행 될 때마다 그 thread의 index를 proc의 curtid에 저장합니다.

(1) user_stack_pool: 하나의 프로세스에서 많은 thread를 생성하고 종료할 때 생기는 memory allocation 부하를 막기 위해 설정해주었습니다.

(2) curtid: 현재 running중인 thread의 index를 저장합니다.

(3) threads: 프로세스에서 수용하는 threads의 집합입니다.

```
// Per-process state
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[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  struct thread threads[NTHREAD]; //threads in process
  thread_t curtid;              //currund thread's id
  int limit;                    //limit
  int stackpagenum;             //stack page number
  uint  user_stack_pool[NTHREAD]; // user_stack_pool
};
```

아래와 같이 thread에 필요한 요소들을 다 넣어주었습니다.

(1) tid : thread의 id를 저장합니다.

(2) state : process의 state 구성 요소를 그대로 가져왔습니다.

(3) retval : 각 thread에서 return 해주는 값입니다.

(4) context : process와 동일합니다.

(5) chan : process와 동일합니다.

(6) kstack : kstack 은 thread에 할당하는 kernel stack 입니다.

(7) tf : thread실행과 관련된 field입니다.

(8) killed : 종료된 thread인지 판별합니다. 0이 아니면 종료된 thread입니다. 하나 이상의 스레드가 kill되면 프로세스 내의 모든 스레드가 종료되어야 합니다.

```
struct thread
{
  thread_t tid; // Id of Thread
  enum procstate state;
  void *retval; //save the return value of thread
  struct context *context;  // swtch() here to run process
  void *chan;               // If non-zero, sleeping on chan
  char *kstack;             // Bottom of kernel stack for this thread
  struct trapframe *tf;     // Trap frame for current syscall
  int killed;               // If non-zero, have been killed

};
```

• int thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg);

allocproc함수와 비슷합니다. 새롭게 만들어진 LWP에 start runtine을 시작해하기 때문에 kernel stack을 변경하였습니다. tf-eip를 start_routine의 주소로 두었습니다.

exec2에서

• int thread_join(thread_t thread, void **retval);

해당 thread의 종료를 기다리고 스레드가 thread_exit을 통해 반환한 값을 반환합니다. chan을 tid로 설정하고 자신의 tid와 동일한 thread를 wake up 합니다. thread는 join할 스레드의 id이기 때문에 int형 thread_t를 정의해주었습니다. 이는 wait함수와 비슷한 구성으로 만들었습니다. thread를 정리할 때 retval을 저장해줍니다.

```
typedef unsigned int   uint;
typedef unsigned short ushort;
typedef unsigned char  uchar;
typedef uint pde_t;
typedef int thread_t;
```

types.h에서 type 정의된 thread_t

• void thread_exit(void *retval);

thread_exit에서는 return value를 thread의 retval에 저장합니다.

1.4. syscall 설정

```
#define SYS_close  21
#define SYS_thread_create 22
#define SYS_thread_exit 23
#define SYS_thread_join 24
#define SYS_exec2    25
#define SYS_list 26
#define SYS_setmemorylimit  27
```

```
[SYS_thread_create] sys_thread_create,
[SYS_thread_exit]   sys_thread_exit,
[SYS_thread_join]   sys_thread_join,
[SYS_exec2]    sys_exec2,
[SYS_list]   sys_list,
[SYS_setmemorylimit]   sys_setmemorylimit,

};

void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();
  struct thread *curthread = &CURTHREAD(curproc);

  num = curthread->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // curproc->tf->eax = syscalls[num]();
    curthread->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %d %s: unknown sys call %d\n",
            curproc->pid,curthread->tid, curproc->name, num);
    curthread->tf->eax = -1;
  }
}
```

```
int             thread_create(thread_t *, void *(*)(void *), void *);
void            thread_exit(void *);
int             thread_join(thread_t, void **);
void            list(void);
int             setmemorylimit(int,int);
```

```
int
sys_thread_create(void)
{
  thread_t *thread;
  void *(*start_routine)(void*);
  void *arg;

  if (argptr(0, (char **)&thread, sizeof thread) < 0)
    return -1;

  if (argptr(1, (char **)&start_routine, sizeof start_routine) < 0)
    return -1;

  if (argptr(2, (char **)&arg, sizeof arg) < 0)
    return -1;


  return thread_create(thread, start_routine, arg);
}

int
sys_thread_exit(void)
{
  void* retval;
  if (argptr(0, (char **)&retval, sizeof retval) < 0)
    return -1;
  thread_exit(retval);

  return 0;
}

int
sys_thread_join(void)
{
  thread_t thread;
  void **retval;

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

  if(argptr(1, (char **)&retval, sizeof retval) < 0)
    return -1;

  return thread_join(thread, retval);
}


int
sys_list(void)
{
  list();
  return 0;
}

int
sys_setmemorylimit(void){
  int pid, limit;
```

차례로 syscall.h, syscall.c, defs.h, sysproc.c

2. Implementation

2.1. 초기화

proc.c에서 변수들을 초기화해줍니다. exec2시스템 콜을 통해 만들어진 프로세스는 mode=0, memory limit = 0, stacksize = 1로 초기화해줍니다.

```
// 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 + (stacksize + 1)*PGSIZE)) == 0) //stacksize +
guardpage(1)
  goto bad;
clearpteu(pgdir, (char*)(sz - (stacksize + 1)*PGSIZE));
sp = sz;
```

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->stackpagenum= 1;
  p->threads[0].state = EMBRYO;
  p->threads[0].tid = nexttid++;
  p->limit = 0;
  release(&ptable.lock);
```

```
// Commit to the user image.
oldpgdir = curproc->pgdir;
curproc->pgdir = pgdir;
curproc->sz = sz;
if(curproc->curtid != 0) {
  curproc->user_stack_pool[0] = sz;
  curproc->threads[0] = curproc->threads[curproc->curtid];
  curproc->threads[curproc->curtid].kstack = 0;
}

curproc->threads[0].tf->eip = elf.entry;  // main
curproc->threads[0].tf->esp = sp;
curproc->limit = 0;
curproc->stackpagenum = stacksize;
curproc->curtid = 0;

cprintf("*****EXEC2*****\n");
for(curthread = &curproc->threads[1]; curthread < &curproc->threads[64]; curthread++) {
  if (curthread->kstack)
    kfree(curthread->kstack);
  curthread->kstack = 0;
  curthread->tid = 0;
  curthread->retval = 0;
  curthread->state = UNUSED;

  curproc->user_stack_pool[curthread - curproc->threads] = 0;
}
switchuvm(curproc);
freevm(oldpgdir);
return 0;
```

exec.2에서

proc.c의 allocproc함수에서 p변수들 초기화 (중간)

exec2에서 pmanager, thread에 쓰이는 변수들 초기화 (아래)

2.2. pmanager 관련 구현

(1) list: proc구조체 안의 것들을 출력해줍니다. 이 때, RUNNABLE, SLEEPING, RUNNING상태인 proc들만 출력 대상이 되도록 구현하였습니다. 또한 Race condition을 고려하여 함수 앞부분과 뒷부분에 ptable.lock을 걸어주었습니다.

```
void list(void){
  // struct proc *curproc = myproc();
  acquire(&ptable.lock);
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    //RUNNABLE, RUNNING, SLEEPING
    if(p->pid != 0 && p->killed != 1 &&(p->state == RUNNABLE || p->state == SLEEPING ||
    p->state == RUNNING)){
      cprintf("name: %s\npid: %d \nstackpagenum: %d\nsz: %d\nlimit: %d\n\n", p->name,p->pid,
      p->stackpagenum,p->sz,p->limit);
    }
  }
  release(&ptable.lock);
  return;
}
```

[proc.c] list

(2) setmemorylimit:

```
int setmemorylimit(int pid, int limit) {
  struct proc *p;
  int check_point = 1;

  if(limit <0) {
    return -1;
  }

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid) {
      check_point = 0;
      if(p->sz <= limit) {
        //if the received memory is less than limit return
        release(&ptable.lock);
        return -1;
      } else {
        p->limit = limit; //set limit.
        break;
      }
    }
  }
  // pid doesn't exist return -1
```

setmemorylimit에서는 pid가 같고, p->sz > limit 일 때 limit를 설정합니다.

2-3. thread 관련 구현

```
int thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg)
{
  struct proc *curproc = myproc();
  struct thread *t;
  int t_idx;
  uint sz;
  // struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for (t = curproc->threads; t < &curproc->threads[NTHREAD]; ++t)
    if (t->state == UNUSED)
      goto found;

  release(&ptable.lock);

  return -1;

found:
  t_idx = t - curproc->threads;
  t->tid = nexttid++;
  t->state = EMBRYO;

  // Allocate kernel stack.
  if((t->kstack = kalloc()) == 0){
    cprintf("Issue!\n thread_create\n");
    t->state = UNUSED;
    t->tid = 0;
    t->kstack = 0;
    return -1;
  }
  sp = t->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *t->tf;
  t->tf = (struct trapframe*)sp;
  *t->tf = *CURTHREAD(curproc).tf;
  // Set up new context to start executing at forkret,
  // which returns to trapret.

  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *t->context;
  t->context = (struct context*)sp;
  memset(t->context, 0, sizeof *t->context);
  //return address forkret.
  t->context->eip = (uint)forkret;

  if (curproc->user_stack_pool[t_idx] == 0)
  {
    sz = PGROUNDUP(curproc->sz);
    if ((sz = allocuvm(curproc->pgdir, sz, sz + PGSIZE)) == 0)
    {
      cprintf("cannot alloc user stack\n");
      goto bad;
```

thread_create를 통해 thread를 초기화합니다. 이때 user_stack_pool도 초기화 해줍니다. 이 때 stact_routine으로 함수를 받아서 실행시킬 수 있도록 t->tf->eip = (uint)start_routine;

t->tf->esp = (uint)sp; 코드를 넣어줍니다.

```
int thread_join(thread_t thread, void **retval){
  //wait the exit of the thread and return the value returned by thread_exit().
  struct proc *p;
  struct thread *t;

  acquire(&ptable.lock);

  for (p = ptable.proc; p < &ptable.proc[NPROC]; ++p)
    if (p->state == RUNNABLE)
      for (t = p->threads; t < &p->threads[NTHREAD]; ++t)
        if (t->tid == thread && t->state != UNUSED){
          cprintf("*********thread_join***********\n");
          goto found;
        }
  release(&ptable.lock);
  return -1;

found:
  if(t->state != ZOMBIE) {
    sleep((void*)thread, &ptable.lock);
  }

  // if (retval != 0)
  *retval = t->retval;
  // *retval = p->retval;
  kfree(t->kstack);
  t->kstack = 0;
  t->retval = 0;
  t->tid = 0;
  t->state = UNUSED;
  release(&ptable.lock);

  return 0;
}
```

thread_join

thread join에서는 retval을 저장해주고 thread의 값을 초기화 해줍니다. Wait 함수와 비슷하기 때문에 wait함수를 많이 참고해서 코드를 구현하였습니다.

```
void thread_exit(void *retval)
{
  cprintf("*********thread_exit********** RETVAL: %d\n", retval);
  struct proc *curproc = myproc();
  struct thread *curthread = &CURTHREAD(curproc);

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1((void*)curthread->tid);

  // Jump into the scheduler, never to return.
  // curthread->retval = retval;
  curproc->retval = retval;
  curthread->state = ZOMBIE;
  curproc->threads[curproc->curtid].retval = retval;


  sched();
  panic("zombie exit");
}
```

thread_exit

curthread->tid를 보내서 wake1함수를 실행시킵니다.

Race condition이 발생하지 않도록 중간중간에 lock을 걸어주었고 특히 여러 thread에서 동시에 growproc을 호출하면 race가 발생하기 때문에 ptable.lock을 이용해 race를 방지해주었습니다. growproc에 적절하게 lock을 걸어주었습니다.

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

  acquire(&ptable.lock); //ptable.lock
  sz = curproc->sz;

  if(curproc->limit < sz+n && curproc->limit != 0){
    return -1;
  }
  if(n > 0){
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0) {
      release(&ptable.lock);
      return -1;
    }
  } else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0) {
      release(&ptable.lock);
      return -1;
    }
  }
  release(&ptable.lock);
  curproc->sz = sz;
  switchuvm(curproc);
  return 0;
}
```

[proc.c] growproc에 적절히 lock을 걸어준 모습입니다.

Thread

Sbrk를 호출 할 때 발생하는 racing문제를 해결하기 위해

Racing 문제를 해결하기 위해 growproc에

3. Result

3-1 pmanager test result

pipe가 잘 작동하는지 확인하기 위해 pipetest 함수를 pmanager가 실행될 때 활용할 수 있도록 하였습니다.

```
void    int fds[2]
pipete
{       fds
  int fds[2], pid;
  int seq, i, n, cc, total;

  if(pipe(fds) != 0){
    printf(1, "pipe() failed\n");
    exit();
  }
  pid = fork();
  seq = 0;
  if(pid == 0){
    close(fds[0]);
    for(n = 0; n < 5; n++){
      for(i = 0; i < 1033; i++)
        buf[i] = seq++;
      if(write(fds[1], buf, 1033) != 1033){
        printf(1, "pipetest failed 1\n");
        exit();
      }
    }
    exit();
  } else if(pid > 0){
    close(fds[1]);
    total = 0;
    cc = 1;
    while((n = read(fds[0], buf, cc)) > 0){
      for(i = 0; i < n; i++){
        if((buf[i] & 0xff) != (seq++ & 0xff)){
          printf(1, "pipetest failed 2\n");
          return;
        }
      }
```

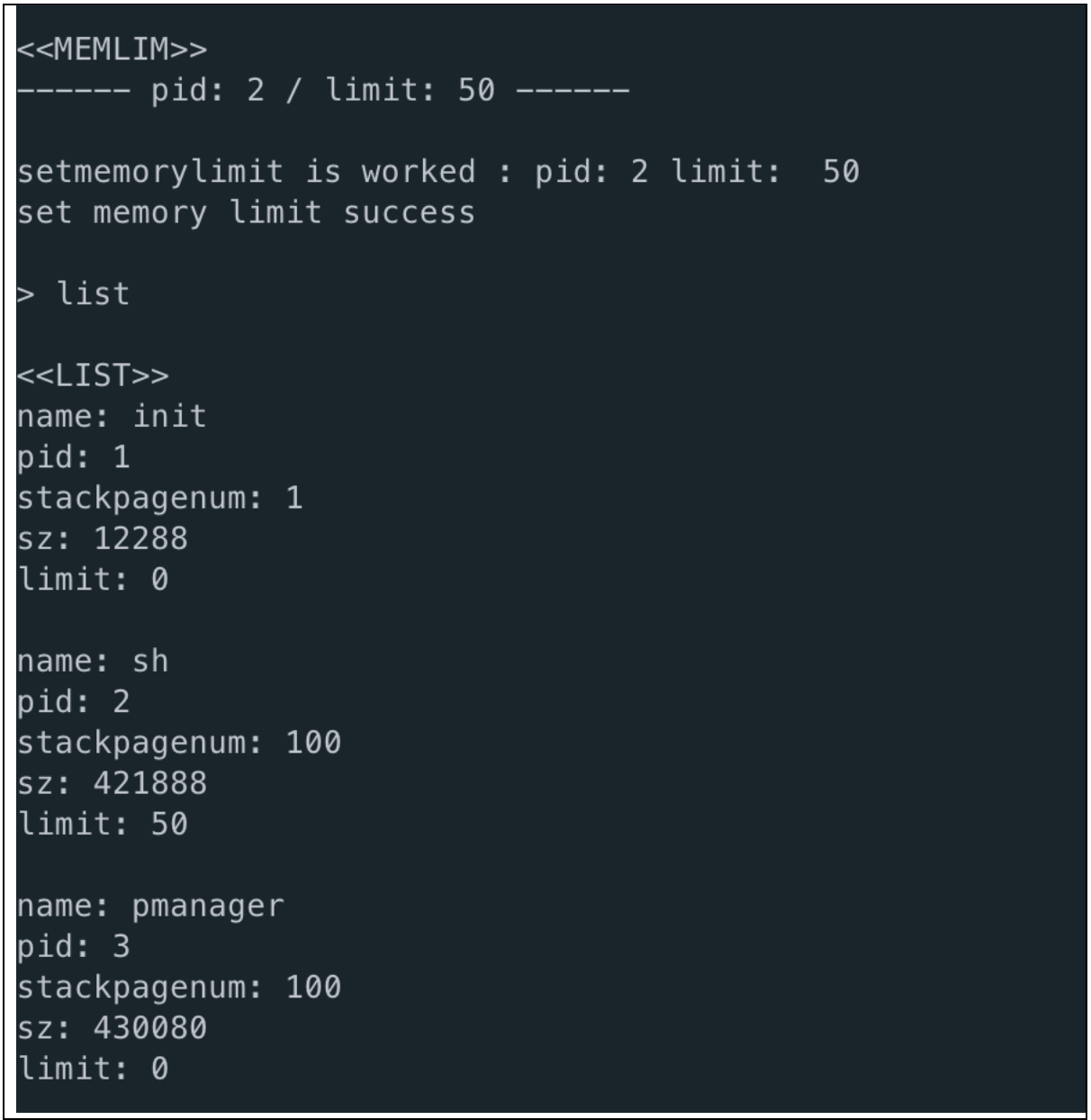
```
pipetest ok
```

pmanager.c에서 pipe test 코드와 결과

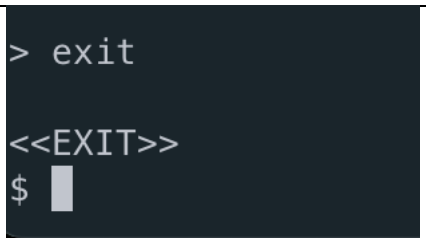
3-2 thread test result

```
[PMANAGER]

> list

<<LIST>>
name: init
pid: 1
stackpagenum: 1
sz: 12288
limit: 0

name: sh
pid: 2
stackpagenum: 100
sz: 421888
limit: 0

name: pmanager
pid: 6
stackpagenum: 100
sz: 430080
limit: 0
```

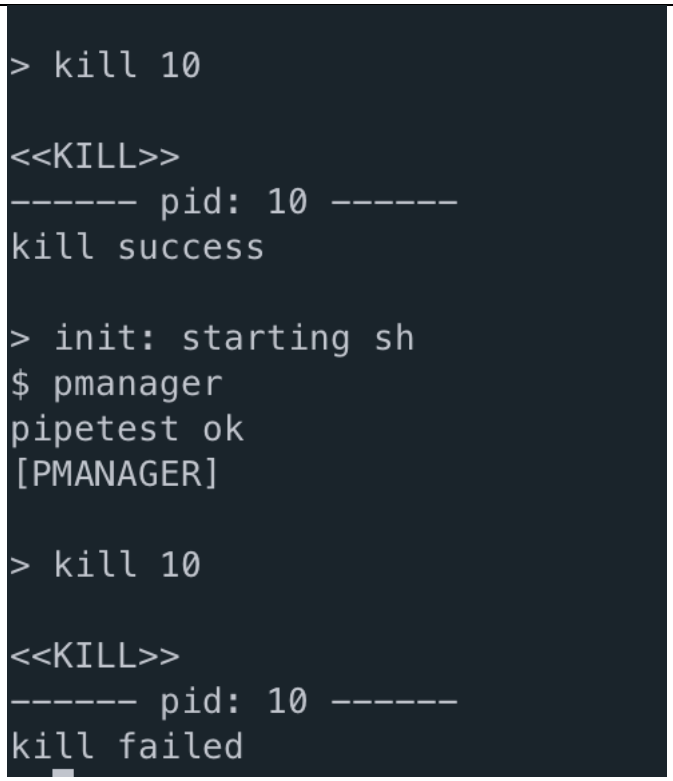
list test 차례대로 list가 찍힌 모습을 확인할 수 있습니다.



memlim test 결과입니다. 이를 통해 다음과 같이 limit를 process에 할당해주는 것을 확인할 수 있습니다.



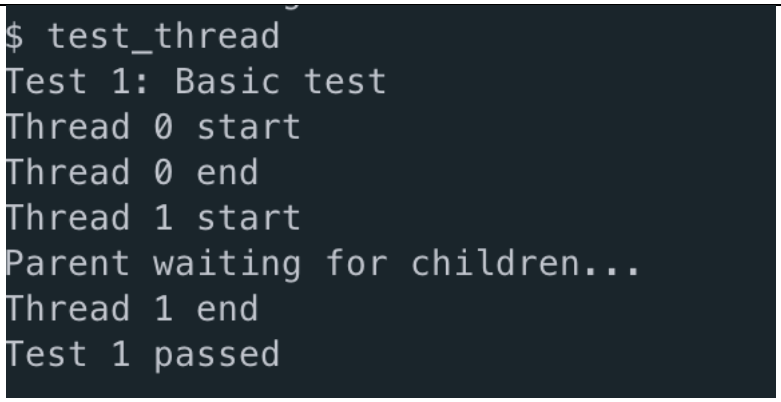
exit test pmanager를 종료하고 다시 xv6로 돌아가는 모습을 확인할 수 있습니다.



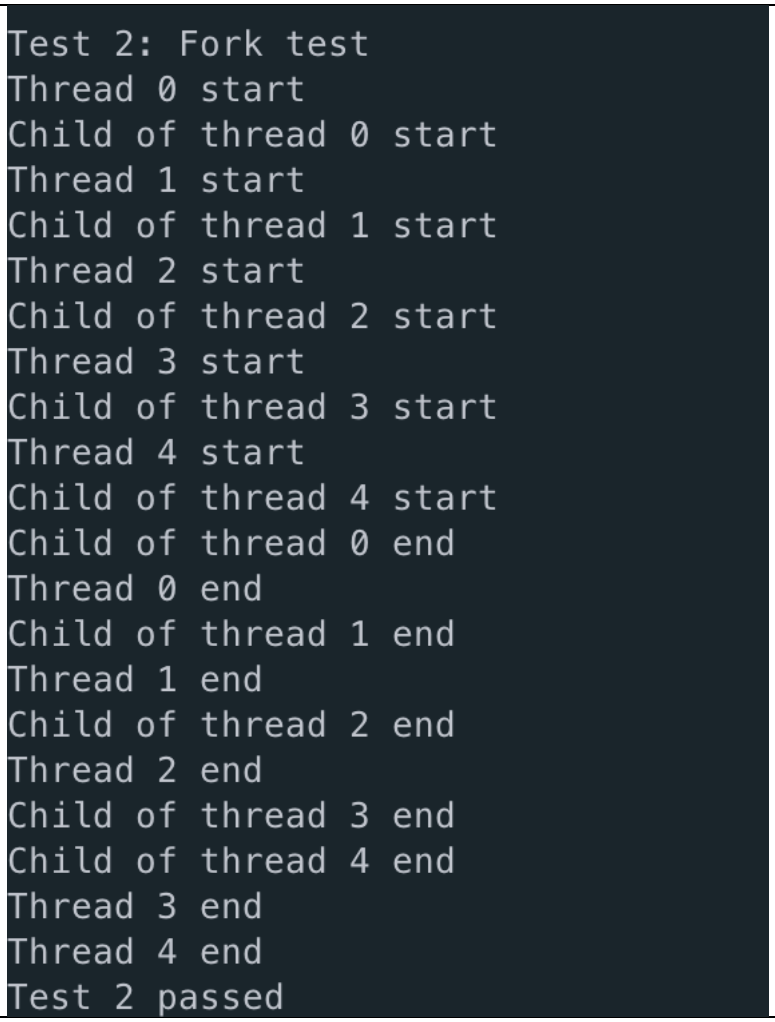
kill test

kill 10를 했을 때 처음에는 success가 뜨지만 다시 kill 10했을 때는 이미 kill이 된 proc이므로 failed가 출력 됩니다.

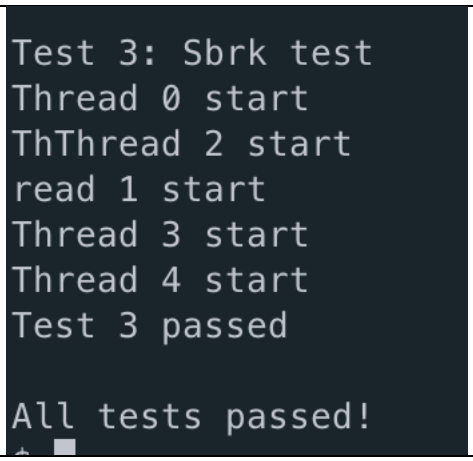
4-1. Thread Test



thread_test.c라는 조교님께서 올려주신 파일을 실행해보았습니다. 다음은 thread_create를 테스트해본 결과입니다.



thread_test.c라는 조교님께서 올려주신 파일을 실행해보았습니다. 이를 통해 Fork를 test해보았습니다.



thread_test.c라는 조교님께서 올려주신 파일을 실행해보았습니다. 이를 통해 join도 test해보았습니다.

4. Trouble shooting

(1) 변경사항이 기하급수적으로 늘어난 Thread의 구조체할당

Thread구조체를 직접 할당 하다 보니 굉장히 바꾸어 줄게 많았습니다. 당연히 proc안에 구조체를 형성 해야 한다고 생각했으나 조교님의 말씀을 듣고 그제서야 꼭 thread를 구조체로 정의하지 않아도 된다는 것을 알게 되었습니다. 만약 다시 작업을 하게 된다면, proc구조체 안에 retval을 설정하고 main thread의 process를 저장하고, 새로운 thread의 기본 주소 등을 저장하여 thread를 구현할 것입니다.

(2) retval이 return을 못해주는 문제

retval이 return을 못해주는 문제가 발생하는데 원인을 도저히 찾을 수 없었습니다. 그래서 첫 번째 과제의 mlfq구조체를 가져와 명확하게 proc을 지정해 주는 과정을 거쳤습니다. 또한 제가 깜빡한 부분이 있었는데 syscall() 함수에서 curproc부분을 curthread로 바꾸어 주는 과정을 거쳤습니다. 이를 통해 current thread -> tf ->eax;의 값을 받아 number에 넣어주었습니다.

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();
  struct thread *curthread = &CURTHREAD(curproc);

  num = curthread->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curthread->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %d %s: unknown sys call %d\n",
            curproc->pid, curthread->tid, curproc->name, num);
    curthread->tf->eax = -1;
  }
}
```