Operating System – Project2 Wiki

2019073181 심기호

Design

명세에서 요구하는 조건에 대해서 어떻게 구현할 계획인지, 어떤 자료구조와 알고리즘이 필요한지, 자신만의 디자인을 서술합니다.

Exec2 system call은 기존의 xv6 exec system call과 유사하게 생각하고, 가드용 페이지, 실제로 사용할 페이지를 정하는 2*PGSIZE를 살짝 수정할 것이다,

Pmanager는 입력을 받고, 입력 문자열에 대해서 parsing을 한 후 해당하는 명령어에 대해서 명세에서 원하는 대로 구현을 할 예정이다.

LWP구현은 먼저 쓰레드 구조체를 만들어서 실질적인 쓰레드를 만들고 프로세스가 자신의 쓰레드 배열을 가지면서, 쓰레드 배열의 0번째 인덱스의 쓰레드가 메인쓰레드(프로세스)가 되게 해줄 생각이다. 그리고 스케줄러에서는 프로세스의 각 쓰레드들이 순차적으로 돌면서 cpu를 받아서 일을 하게 할 예정이다. 각 쓰레드들은 프로세스의 kstack, tf, context에 정보를 주입해서 쓰레드의 정보를 가진 프로세스가 작업의 단위가 되고, 스케줄러에서 context switch를 한 후 프로세스가 일을 끝마치고 다시 스케줄러로 돌아왔을 땐 프로세스의 정보를 다시 쓰레드에게 넘겨주는 식으로 구현할 예정이다. 다른 함수들이나 부가적인 기능, 세부적인 기능들은 Implement part에서 소개하겠다.

Implement

0. process 구조체

-process 구조체는 기존의 xv6 proc 구조체에 thread 배열을 담을 thrlist, 다음번 thread id를 지정해줄 nexttid, 현재 돌고있는 thread가 thread배열에서 몇번째 인덱스인지 알려줄 curThrIdx, 그리고 메모리 사이즈를 정해줄 limit 변수를 추가하고, exec함수를 갔다왔는지에 대해서 execVisited 변수를 추가했다.

```
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
  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
  {
    struct thread thread[NTHR];
  } thrlist;
  uint nexttid; // 다음번에 올 tid번호
  uint curThrIdx;  // 현재 돌고있는 thread가 thrlist에서 몇번째인지
  int limit;     //메모리 사이즈
};
```

1. thread 구조체

-thread 구조체는 kstack, tf, context, chan, pid, tid, retval, thr_state를 가지고 execvisited라는 변수를 가진다. execvisited라는 변수는 exec systemcall을 수행하면 현재 쓰레드가 0번(메인 쓰레드)로 옮겨져야 하기 때문에 execVisited라는 변수를 두어서 스케줄러로 다시 돌아왔을 때 0번 쓰레드로 정보를 담기로 했다. (추후에 더 자세히 설명) 그리고 쓰레드 상태를 저장하는 enum 배열을 하나 선언했다.

```
struct thread
{
  char *kstack;               // 쓰레드의 커널스택
  enum threadstate thr_state; // 쓰레드 상태
  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 pid;                    // Process ID
  uint tid;                   // thread ID
  void *retval;               // 리턴값
  uint execVisited;           //exec system call을 실행했는지
};
```

2. 프로세스 작동방식 & 스케줄러

- 스케줄러가 동작하는 과정은 다음과 같다. 한 프로세스에서 쓰레드 배열을 돌면서 thr_runnable한 쓰레드의 정보들을 프로세스의 커널스택, trapframe, context에 대입을 한 후 그 프로세스가 context switch를 통해 동작한다. 작업을 마치고 돌아온 프로세스는 아까 정보를 받은 쓰레드한테 커널스택, trapframe, context 정보들을 다시 대입한다. 한마디로 말하자면 프로세스는 작업을 하는 단위이고, 쓰레드의 정보들을 일시적으로 가진 채 쓰레드 대신 일을 하고 다시 쓰레드에게 정보를 반환한다.

```
//프로세스에 쓰레드의 정보를 넣어주는 과정
        p->curThrIdx = idx;
        p->kstack = t->kstack;
        p->context = t->context;
        p->tf = t->tf;
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&(c->scheduler), p->context);
        switchkvm();
```

그리고 작업을 마치고 다시 스케줄러로 돌아왔을 때에는 정보들을 반납한다. Exec 시스템콜을 수행한 후에는 0번 쓰레드에 정보를 반납하고, exec이 아니였다면 다시 원래 쓰레드에 반납한다.

```
if (p->state != ZOMBIE)//kill되지 않았다면..
        {
          if (t->execVisited == 1)//exec을 갔다온 뒤라면
          {
            p->curThrIdx = 0;//curThrIdx를 0으로 바꿔주고, 메인 쓰레드에 kstack, context, tf 대입.
            p->thrlist.thread[0].kstack = p->kstack;
            p->thrlist.thread[0].context = p->context;
            p->thrlist.thread[0].tf = p->tf;
            //원래 쓰레드의 자원 정리. kfree는 하면 안됨. 포인터만 0으로 바꿔준다.
            t->execVisited = 0;
            t->kstack = 0;
            t->chan = 0;
            t->thr_state = THR_UNUSED;
            t->retval = 0;
            t->tid = 0;
            t->pid = 0;
            t->context = 0;
            t->tf = 0;
          }
          else
          {
            //프로세스의 상태를 쓰레드에 대입한다.
            t->kstack = p->kstack;
            t->context = p->context;
            t->tf = p->tf;
          }
        }
```

3. thread_create(thread_t *thread, *thread, void start_routine)(void *), void arg)

-thread_create는 기존 xv6의 allocproc, fork, exec 함수들을 보면서 구현했다. 먼저 쓰레드 배열을 돌면서 thr_unused인 thread를 찾는다. 그 후 tid 배정, 커널스택 할당, 커널스택에 trapframe, trapret, forkret 정보들을 넣어주고, limit을 확인하고 유저스택 페이지를 할당하게 한다. 유저스택에 인자와 fake return PC를 넣어준다. 그리고 eip에 start_routine, esp에 stackpointer을 넣어준다.

```
found:
  // allocproc의 과정처럼 정보를 넣어준다. kstack할당, tf 대입, context 대입
  t->thr_state = EMBRYO;
  t->tid = *thread;

  release(&ptable.lock);

  // Allocate kernel stack.
  if ((t->kstack = kalloc()) == 0)
  {
    t->thr_state = UNUSED;
    goto bad;
  }
  sp = t->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *t->tf;
  t->tf = (struct trapframe *)sp;

  // 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);
  t->context->eip = (uint)forkret;
  // t->tf에 프로세스의 현재 쓰레드 trapframe을 대입.
  *t->tf = *p->thrlist.thread[p->curThrIdx].tf;
  // 그 후 유저스택에 인자를 넣어줌.
  uint ustacks[2];
  uint stackpointer = 0;
  pde_t *pagedir = p->pgdir;
  sz = PGROUNDUP(p->sz);
  if (p->limit == 0) // unlimied 상태
  {
    if ((sz = allocuvm(pagedir, sz, sz + 2 * PGSIZE)) == 0)
    {
      goto bad;
    }
  }
  else if (p->limit != 0)
  {
    if (p->limit >= sz + 2 * PGSIZE) // 할당해도 되는지 검사
    {
      if ((sz = allocuvm(pagedir, sz, sz + 2 * PGSIZE)) == 0)
      {
        goto bad;
      }
    }
    else
    {
      goto bad;
    }
  }
  clearpteu(pagedir, (char *)(sz - 2 * PGSIZE));
  p->sz = sz;
  stackpointer = sz;
  ustacks[0] = 0xffffffff; // fake return PC
  ustacks[1] = (uint)arg;
  stackpointer -= 8;
  // 그 후 pagedir에 ustack 복사.
  if (copyout(pagedir, stackpointer, ustacks, 2 * 4) < 0)
    goto bad;
  // eip에 start_routine을 넣어주고, esp에 sp를 넣어준다.
  t->tf->eip = (uint)start_routine;
  t->tf->esp = stackpointer;
  // 쓰레드의 상태를 runnable로 바꿔준다.
  t->thr_state = THR_RUNNABLE;
  p->pgdir = pagedir;
  switchuvm(p);
  return 0;
```

4. thread_exit(void *retval)

```
void thread_exit(void *retval)
{
  struct proc *p = myproc();
  acquire(&ptable.lock);
  struct thread *t = &p->thrlist.thread[p->curThrIdx];
  //쓰레드의 상태를 zombie로 바꿔주고, retval을 대입해준다.
  p->thrlist.thread[0].thr_state = THR_RUNNABLE;
  t->thr_state = ZOMBIE;
  t->retval = retval;
  p->state = RUNNABLE;
  sched();
}
```

Thread_exit은 현재 쓰레드의 상태를 좀비로 바꿔주고, retval을 저장해주고, 0번 쓰레드를 runnable로 바꿔줌으로써 0번 쓰레드가 모든 쓰레드의 부모 역할을 하여 자원을 회수하게 한다.

5. thread_join(thread_t thread, void **retval)

```
found:
  for (;;)
  {//zombie라면
    if (t->thr_state == THR_ZOMBIE)
    {//retval 저장
      *retval = t->retval;
      if (t->kstack != 0)
      {
        kfree(t->kstack);
      }
      t->thr_state = THR_UNUSED;
      t->context = 0;
      t->tf = 0;
      t->kstack = 0;
      // t->retval = 0;
      release(&ptable.lock);
      return 0;
    }

    if (p->killed)//죽었다면 끝내기
    {
      release(&ptable.lock);
      return -1;
    }

    sleep(&p->thrlist.thread[0], &ptable.lock);
  }
}
```

Thread_join은 wait함수와 비슷하게 구현했다. 무한 for loop을 돌면서 쓰레드의 상태가 좀비라는 것은 exit되었다는 소리이기 때문에 자원을 정리해줘야 하고, 프로세스가 죽었다면 바로 리턴 -1을 한다. 좀비가 아니고 프로세스가 죽지 않았다면 채널을 main 쓰레드를 채널로 해서 자게한다.

6. kill system call

```
if (p->pid == pid)
    {
      p->killed = 1;
      for (t = p->thrlist.thread; t < &p->thrlist.thread[NTHR]; t++)
      {
        //사용되지 않은 쓰레드가 아니라면 모두 runnable로 바꿔준다.
        if (t->thr_state == THR_UNUSED)
          continue;
        t->thr_state = THR_RUNNABLE;
      }
      release(&ptable.lock);
      return 0;
    }
```

-pid가 맞는 프로세스를 찾아 killed를 1로 바꿔주고, thread배열을 돌면서 unused가 아니라면 runnable로 상태를 바꿔주어서 나중에 스케줄러로 가서 cpu를 받아서 trap에서 정리되게끔 한다.

7. wakeup1 system call

```
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    for (t = p->thrlist.thread; t < &p->thrlist.thread[NTHR]; t++)
    {
      if (t->thr_state == THR_SLEEPING && t->chan == chan)
      {//쓰레드가 자고 있고, chan정보가 맞다면, thr_state는 runnable로 바꿔준다.
        t->thr_state = THR_RUNNABLE;
      }
    }
  }
```

쓰레드의 상태가 자고 있고, chan과 같다면 쓰레드가 일어나야 하므로, runnable로 바꿔준다.

8. sleep system call

```
t->chan = chan;
t->thr_state = THR_SLEEPING;
p->state = RUNNABLE;
sched();

// Tidy up.
t->chan = 0;

// Reacquire original lock.
if (lk != &ptable.lock)
{ // DOC: sleeplock2
  release(&ptable.lock);
  acquire(lk);
}
```

Sleep은 chan을 지정해주고, 상태를 sleeping으로 바꿔준다. 프로세스의 상태는 runnable로 유지한다.

9. wait system call

```
  if (p->state == ZOMBIE)
  {
    // Found one.
    pid = p->pid;
    //쓰레드 배열을 돌면서 프로세스의 쓰레드들의 자원을 회수해주고 정리해준다.
    for (idx = 0, t = p->thrlist.thread; t < &p->thrlist.thread[NTHR]; idx++, t++)
    {
      if (t->thr_state != THR_UNUSED)
      {
        kfree(t->kstack);
      }
      t->kstack = 0;
      t->context = 0;
      t->thr_state = THR_UNUSED;
      t->tid = 0;
      t->chan = 0;
      t->retval = 0;
      t->tf = 0;
    }
    //프로세스의 정보도 회수&정리
    p->kstack = 0;
    freevm(p->pgdir);
    p->pid = 0;
    p->parent = 0;
    p->name[0] = 0;
    p->state = UNUSED;
    release(&ptable.lock);
    return pid;
  }
}
```

Wait은 current 프로세스의 자식 프로세스를 찾고 그 프로세스의 쓰레드 배열을 돌면서 unused된 쓰레드를 제외하곤 커널스택 초기화, context 초기화 등 초기화 작업을 해주고, 자식 프로세스의 정보도 회수하고 초기화한다.

10. setmemorylimit(int pid, int limit)

```
// limit이 0보다 작다면 out
if (limit < 0)
  goto bad;

for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{
  // ptable을 돌면서 pid를 만나고, limit이 p->sz보다 작다면 p->limit을 limit으로 정해준다.
  if (p->pid == pid)
  {
    if (p->sz > limit)
    {
      goto bad;
    }
    else
    {
      p->limit = limit;
      return 0;
    }
  }
}
goto bad;
```

Ptable을 돌면서 해당 pid의 Limit이 0미만, pid가 없는 상황, sz가 limit보다 큰 상황이라면 return -1을 하고 그 외의 상황이라면 정상이기 때문에 프로세스의 메모리 limit을 인자로 지정한다.

11. exit system call

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

// Pass abandoned children to init.
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{
  if (p->parent == curproc)
  {
    p->parent = initproc;
    if (p->state == ZOMBIE)
      wakeup1(initproc);
  }
}
//현재 프로세스의 상태를 좀비로 바꾸고
curproc->state = ZOMBIE;
//쓰레드의 상태를 unused가 아니면 좀비로 바꿔준다.
for (t = curproc->thrlist.thread; t < &curproc->thrlist.thread[NTHR]; t++)
{
  if (t->thr_state != THR_UNUSED)
  {
    t->thr_state = THR_ZOMBIE;
  }
}
```

기존 xv6의 exit system call을 유지하면서 exit하는 프로세스의 쓰레드 배열도 순회하면서 unused가 아니라면 thr_zombie로 바꿔준다.

12. fork system call

```
// 현재 프로세스의 페이지 테이블을 복사.
if ((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0)
{
  kfree(np->thrlist.thread[0].kstack);
  np->thrlist.thread[0].kstack = 0;
  np->thrlist.thread[0].thr_state = UNUSED;
  np->state = UNUSED;
  return -1;
}
//sz, parent, curThrIdx 정보 초기화
np->sz = curproc->sz;
np->parent = curproc;
np->curThrIdx = 0;

// Swap stacks of current thread index and index 0.
//tf정보도 복사
*np->thrlist.thread[0].tf = *curproc->tf;

// Clear %eax so that fork returns 0 in the child.
np->thrlist.thread[0].tf->eax = 0;

for (i = 0; i < NOFILE; i++)
  if (curproc->ofile[i])
    np->ofile[i] = filedup(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;
np->thrlist.thread[0].thr_state = THR_RUNNABLE;
release(&ptable.lock);
return pid;
}
```

Fork는 현재 프로세스의 페이지 테이블을 그대로 복사하고, sz도 복사하고, parent로 지정 그 후 프로세스가 새로 생긴 것이기 때문에 np의 curThrIdx를 메인쓰레드 번호인 0으로 지정해준다. 그 후 0번 쓰레드의 tf에 기존 프로세스의 tf를 복사하고, 자식 프로세스의 eax에는 리턴값 0을 넣어주고, 기존의 fork가 하던 대로 파일 정보를 복사하고 상태를 runnable로 바꿔준다.

13. list_info()

```
void list_info()//ptable의 process중 unused되지 않은 프로세스의 정보를 출력
{
  struct proc *p;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state !=UNUSED)
    {
      cprintf("name : %s\n", p->name);
      cprintf("pid is %d\n", p->pid);
      cprintf("stack page is %d\n", p->sz/4096);
      cprintf("Memory size is %d\n", p->sz);
      cprintf("Memory limit is %d\n", p->limit);
      cprintf("\n");
    }
  }
}
```

Ptable을 순회하면서 unused가 아닌 프로세스들에 대해서는 정보들을 출력한다.

14. pmanager-parse_input

```
void parse_input(char *input, char **args, int *argc)
{
  int i = 0;
  int len = strlen(input);
  int arg_cnt = 0;
  int arg_len = 0;

  while (i < len && arg_cnt < MAX_ARGS - 1)
  {
    if (input[i] == ' ' || input[i] == '\r' || input[i] == '\n')
    {
      if (arg_len > 0)
      {
        args[arg_cnt] = (char *)malloc(arg_len + 1);
        memmove(args[arg_cnt], input + i - arg_len, arg_len);
        args[arg_cnt][arg_len] = '\0';
        arg_cnt++;
        arg_len = 0;
      }
    }
    else
    {
      arg_len++;
    }
    i++;
  }

  if (arg_len > 0)
  {
    args[arg_cnt] = (char *)malloc(arg_len + 1);
    memmove(args[arg_cnt], input + i - arg_len, arg_len);
    args[arg_cnt][arg_len] = '\0';
    arg_cnt++;
  }

  args[arg_cnt] = 0;
  *argc = arg_cnt;
}
```

Parse_input함수에서는 들어온 input에 대해서 공백을 만나면 그때까지의 문자열을 args에 복사한 후, 다시 초기화 시켜서 다시 읽는다. I는 전체 문자열에 대한 index, len은 input 문자열의 공백포함 총 길이, arg_cnt는 전체 들어온 인자의 개수, arg_len은 각 인자들의 길이를 의미한다. Memmove 함수로 args[arg_cnt]에 지금 읽고 있는 인덱스로부터 arg_len까지의 길이를 복사한다.

15. pmanager-main

```c
int main(void)
{
  char input[1000];

  while (1)
  {
    printf(1, "$$$ pmanager cmd : "); // 쉘 프롬프트 출력

    memset(input, 0, sizeof(input)); // 입력 버퍼 초기화
    gets(input, sizeof(input));      // 사용자로부터 입력 받기

    // 입력을 인자로 구문 분석
    char *args[MAX_ARGS];
    int argc = 0;

    parse_input(input, args, &argc);

    if (strcmp(args[0], "list") == 0)
    {
      list_info();
    }
    else if (strcmp(args[0], "kill") == 0)
    {
      int pid = atoi(args[1]);
      int result = kill(pid); // kill 시스템 콜을 사용하여 프로세스 종료
      if (result == -1)
      {
        printf(1, "Failed to kill process with PID %d\n", pid);
      }
      else
      {
        printf(1, "Process with PID %d killed\n", pid);
      }
    }
    else if (strcmp(args[0], "execute") == 0)
    {
      char *path = args[1];
      int stack_size = atoi(args[2]);
      char *exec_args[2];
      exec_args[0] = path;
      exec_args[1] = 0;
      int pid = fork();
      if (pid == 0)
      {

        if (exec2(path, exec_args, stack_size) == -1)
        {
          printf(2, "Failed to execute program at %s\n", path);
          exit();
        }
        printf(1,"asdf\n");
      }
    }
    else if (strcmp(args[0], "memlim") == 0)
    {
      int pid = atoi(args[1]);
      int limit = atoi(args[2]);
      int result = setmemorylimit(pid, limit); // 사용자 정의 시스템 콜을 통해 프로세스의 메모리 제한 설정
      if (result == -1)
      {
        printf(1, "Failed to set memory limit for process with PID %d\n", pid);
      }
      else
      {
        printf(1, "Memory limit set for process with PID %d\n", pid);
      }
    }
    else if (strcmp(args[0], "exit") == 0)
    {
      exit(); // pmanager 종료
    }
    else
    {
      printf(1, "Invalid command: %s\n", args[0]);
    }
  }
```

Pmanager-main에서는 gets함수로 input 문자열을 받고, parse_input함수로 인자를 분리한 후 strcmp로 첫번째 cmd를 확인하면서 커맨드에 맞는 명령어들을 수행하는 모습이다. List는 list_info함수를 사용하고, kill은 kill system call을 사용, execute는 fork한 후 exec system call을 사용, memlim은 setmemoryliit system call을 사용한다.

16. growproc()

```
int growproc(int n)
{
  uint sz;
  struct proc *curproc = myproc();
  sz = curproc->sz;
  if (n > 0)
  {
    if (curproc->limit == 0) // unlimied 상태
    {
      if ((sz = allocuvm(curproc->pgdir, sz, sz+n)) == 0)
      {
        return -1;
      }
    }
    else if (curproc->limit != 0)
    {
      if (curproc->limit >= sz + n) // 할당해도 되는지 검사
      {
        if ((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
        {
          return -1;
        }
      }
      else
      {
        return -1;
      }
    }
  }
  else if (n < 0)
  {
    if ((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  curproc->sz = sz;
  switchuvm(curproc);
  return 0;
}
```

초기 limit은 0으로 allocproc에서 지정을 해주었고 0이면 limit이 없는 상태로 생각해서 구현했다. Limit이 0이 아니라면 pmanager에서 memlim을 통해 지정해준 상태이기 때문에 curproc->limit이 sz+n보다 크거나 같은지 검사해서 alloc을 할 수 있게 해주었다.

17. exec2 system call

```
sz = PGROUNDUP(sz);
if (curproc->limit == 0) // unlimied 상태
{
  if ((sz = allocuvm(pgdir, sz, sz + (stacksize + 1) * PGSIZE)) == 0)
  {
    goto bad;
  }
}
else if (curproc->limit != 0)
{
  if (curproc->limit >= sz + (stacksize+1) * PGSIZE) // 할당해도 되는지 검사
  {
    if ((sz = allocuvm(pgdir, sz, sz + (stacksize + 1) * PGSIZE)) == 0)
    {
      goto bad;
    }
  }
  else
  {
    goto bad;
  }
}
clearpteu(pgdir, (char *)(sz - (stacksize + 1) * PGSIZE));
```

Page 개수를 2가 아니라 stacksize+1로 설정해서 원하는 페이지 개수 + 가드 페이지 해서 할당을 할 수 잇게 해주었다. Exec2 시스템 콜 코드 앞쪽에 stacksize가 1과 100사이 인지 검사하게 해주었다.

18. exec system call

```
// Commit to the user image.
  // exec함수는 프로그램 카운터를 새 프로세스의 main함수의 주소로 설정.
  oldpgdir = curproc->pgdir;
  curproc->pgdir = pgdir;
  curproc->sz = sz;
  curproc->tf->eip = elf.entry; // main
  curproc->tf->esp = sp;
  switchuvm(curproc);
  if (curproc->curThrIdx != 0) // 현재 돌고있는 번호가 0번(메인 쓰레드)이 아닌 그냥 일반 쓰레드일 때
  {
    curproc->thrlist.thread[curproc->curThrIdx].execVisited = 1;                    // execVisited를 1로 설정
    curproc->thrlist.thread[0].chan = curproc->thrlist.thread[curproc->curThrIdx].chan; // chan, retval, thr_state 변경
    curproc->thrlist.thread[0].retval = curproc->thrlist.thread[curproc->curThrIdx].retval;
    curproc->thrlist.thread[0].thr_state = curproc->thrlist.thread[curproc->curThrIdx].thr_state;
  }
  // kill part
  for (idx = 0, t = curproc->thrlist.thread; t < &curproc->thrlist.thread[NTHR]; idx++, t++)
  {
    if (idx == curproc->curThrIdx || idx == 0) // idx가 0(메인쓰레드)거나 현재 쓰레드 번호일 때는 정보 남겨둔다.
      continue;
    if (t->kstack != 0)
    {
      kfree(t->kstack);
    }

    t->kstack = 0;
    t->context = 0;
    t->thr_state = THR_UNUSED;
    t->tid = 0;
    t->chan = 0;
    t->tf = 0;
  }
  freevm(oldpgdir);
  return 0;
```

Exec2 system call처럼 limit을 확인해서 allocuvm함수를 부르게끔 해주고(**사진에는 없음**), **현재 인덱스가 0이 아닌 즉 메인쓰레드가 아니라면 execVisited 변수를 1로 초기화하고, tf, kstack, context 정보를 제외한(스케줄러로 돌아와서 복사할 것) chan retval thr_state 정보를 복사하고 0번쓰레드와 현재 쓰레드를 제외한 나머지 쓰레드들은 kstack을 해제하고 나머지 정보들을 초기화해준다.**

Result

```
root@DESKTOP-HC0THFJ:~/2023_ele3021_2019073181/xv6-public# make clean && make && make fs.img && ./bootxv6.sh
```

-make && make fs.img && ./bootxv6.sh를 쳐서 booting을 한다.

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
```

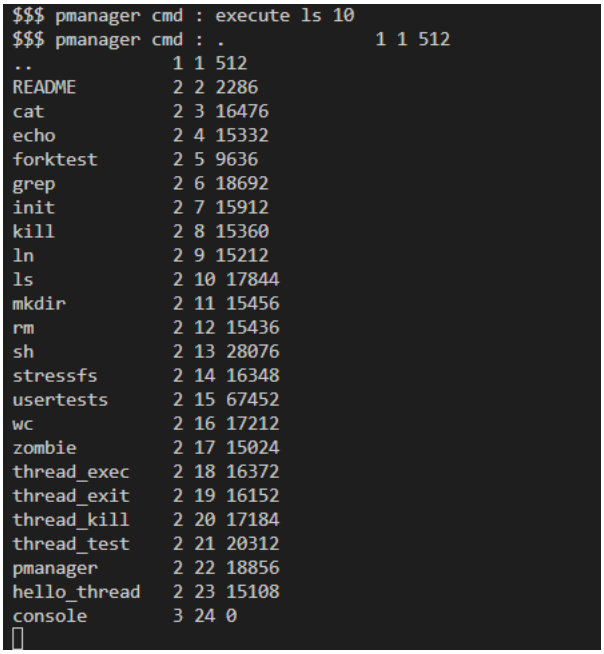
-booting이 되고 init과 sh 프로세스가 실행이 된 모습

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ pmanager
$$$ pmanager cmd : list
name : init
pid is 1
stack page is 3
Memory size is 12288
Memory limit is 0

name : sh
pid is 2
stack page is 4
Memory size is 16384
Memory limit is 0

name : pmanager
pid is 3
stack page is 11
Memory size is 45056
Memory limit is 0
```

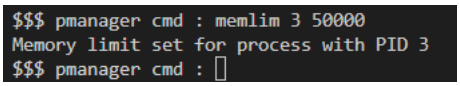
-pmanager에서 list command를 쳤을 때 정상 작동하는 모습

```
$$$ pmanager cmd : kill 100
Failed to kill process with PID 100
$$$ pmanager cmd :
```

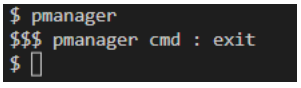
-pmanager에서 kill pid를 했을 때 없는 pid를 했더니 실패 메시지를 띄우는 모습이다.



pmanager에서 execute ls 10을 했을 때 ls가 정상 작동하는 모습을 볼 수 있다.



pmanager에서 memlim pid limit을 했을 때 정상 작동하는 모습을 볼 수 있다.



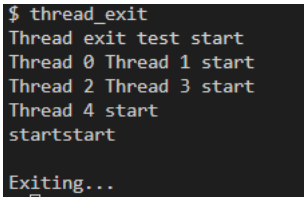
pmanager에서 exit 명령어를 주었을 때 pmanager을 종료하는 모습을 볼 수 있다.

```
$ thread_test
Test 1: Basic test
Thread 0 start
ThreThread 1 start
ad 0 end
Parent waiting for children...
Thread 1 end
Test 1 passed

Test 2: Fork test
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 starChild of thread 0 t
start
Child of thrChild of thread ead3 start
Child Child of thre 1 start
of thread 2 starad 4 start
t
Child of thread 0 end
Thread 0 end
Child of thread 3 end
ThreaChild of thread 1 end 1 end
d
ThrChild of threaChild of threead 2 end
d 2 end
ad 4Thread 3  end
end
Thread 4 end
Test 2 passed

Test 3: Sbrk test
Thread 0 sThread 1 start
Thread Thread 3 start
Thread 4 start
tart
2 start
Test 3 passed

All tests passed!
```

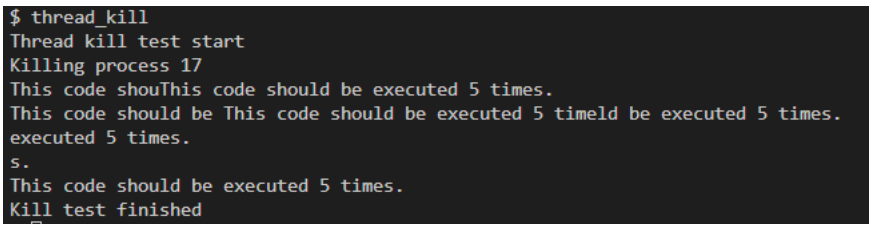
Thread_test 테스트코드를 돌렸을 때 모든 테스트가 정상적으로 작동되는 모습.

```
$ thread_exec
Thread exec test start
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 starThread 4 start
t
Executing...
Hello, thread!
```

Thread_exec을 돌렸을 때 hello thread가 정상적으로 나오는 모습이다.



Thread_exit을 돌렸을 때 정상적으로 exit하는 모습이다.



Thread_kill 테스트코드를 돌렸을 때 정상적으로 kill test를 통과하는 모습이다.

Trouble Shooting

-exec함수를 메인 쓰레드가 아닌 쓰레드가 실행했을 때 메인쓰레드만 남고, 다른 쓰레드는 정리해주고 다시 스케줄러로 돌아왔을 때 프로세스의 정보를 메인쓰레드가 받아야 하는 문제를 해결하는데에 꽤나 많은 고민과 시간이 필요했다. 그래서 해결은 execVisited라는 변수를 만들어서 스케줄러로 돌아왔을 때 이 변수가 1이라면 메인쓰레드로 하여금 프로세스의 정보를 받게했고, 해당 쓰레드는 자원을 정리해주었다.

-그리고 pmanager에서 받은 문자열을 파싱하는데에 어떻게 해야할까 고민을 많이 했고, 특별한 함수가 없었기에 하나하나 문자를 돌면서 공백을 확인하면 끊어서 char* 배열에 저장하는 수밖에 없었다.

-처음에 과제를 시작할 때 코드 이해하는 것들이 쉽진 않았고 프로세스가 쓰레드 배열을 가지고, 동작의 단위는 프로세스이지만, 쓰레드를 프로세스처럼 취급한다는 개념들이 너무나도 헷갈렸다. 그래서 쓰레드를 잠시동안 프로세스 취급을 해주고 작업을 끝나고 와서는 다시 쓰레드의 정보로 넣어주는 아이디어를 떠올리게 되었다.