# Project03 LWP

# DESIGN

## * Project 3의 내용은 git 에 올라온 xv6-public_thread 폴더에 구현했음을 참고 부탁드립니다.

## * LWP 구현 관련 사항

1) LWP(Light-weight-process)는 다른 프로세스와 메모리 자원과 공간을 공유하여 멀티태스킹을 합니다.

2) 메모리를 공유하기 때문에 Multi Process 에 비해 속도가 메모리 사용량이 적고 속도가 빠릅니다.

## * 구체적인 구현

## * init thread_create(thread_t *thread, void (*start_routine)(void ), void *arg);

1. 새로운 thread 를 생성합니다.
2. thread : 해당 주소에 thread id를 저장해 줍니다.
3. start_routine : 스레드가 시작할 함수를 지정합니다. 즉, 새로운 스레드가 만들어지면 그 스레드는 start_routine 이 가리키는 함수에서 시작하게 됩니다.
4. arg : thread의 start_routine에 전달할 인자입니다.
5. return : 성공했다면 0을 그렇지 않다면 0 이 아닌 값을 반환합니다.

## * void thread_exit(void *retval);

1. 스레드를 종료하고 값을 반환합니다. 모든 스레드는 반드시 이 함수를 통해 종료하고, 시작 함수의 끝에 도달하여 종료하는 경우는 고려하지 않습니다.
2. retval : thread를 종료한 후, join 함수에 받아갈 값입니다.

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

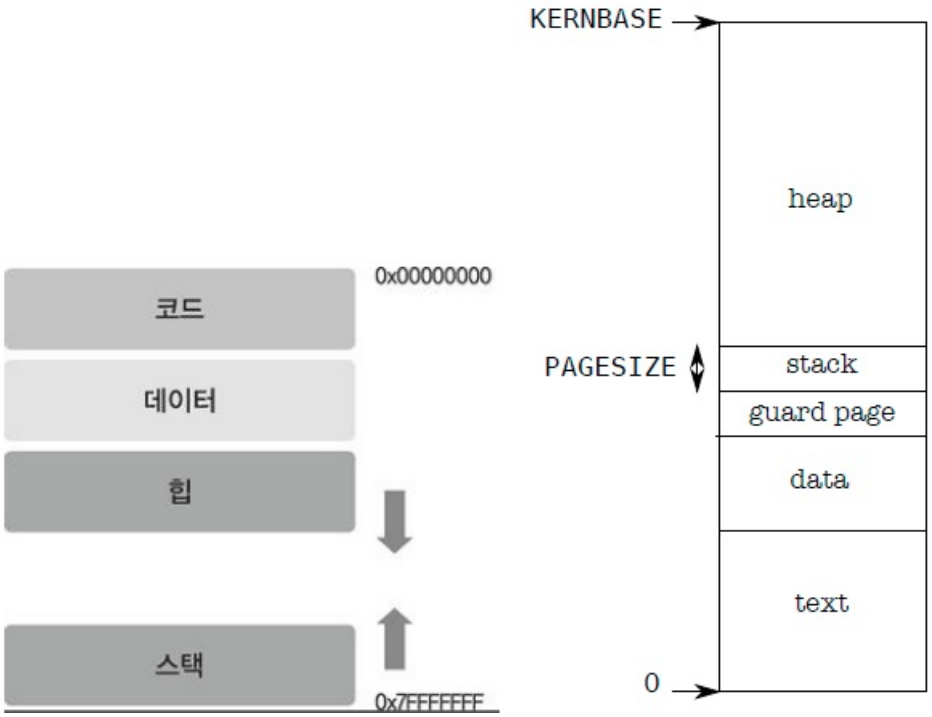
1. 지정한 스레드가 종료되기를 기다리고, 스레드가 thread_exit 을 통해 반환한 값을 받아옵니다. 스레드가 이미 종료되었다면 즉시 반환합니다.
2. 스레드가 종료된 후, 스레드에 할당된 자원들 (페이지 테이블, 메모리, 스택 등)을 회수하고 정리해야 합니다.
3. thread : join할 thread의 id 입니다.
4. retval : thread가 반환한 값을 지정해 줍니다.
5. return : 정상적으로 return 됐다면 0을 아니라면 0 아닌 값을 반환합니다.
6. Scheduler : 기본 scheduler 인 RR(Round Robin)을 사용합니다.

## * xv6-public 내부 함수 변경 사항

- fork : thread 에서 fork 가 호출되면, 기존의 fork 루틴을 문제없이 실행되어야 합니다. 즉, 해당 스레드의 주소 공간의 내용을 복사하고 새로운 프로세스를 시작할 수 있어야 하고, wait 시스템 콜로 기다릴 수도 있어야 합니다.
- exec : exec 가 실행되면 기존 프로세스의 모든 thread들이 정리되어야 하며, 그 중 하나의 thread에서 새로운 프로세스가 시작하고 나머지 thread 는 종료되어야 합니다.
- sbrk : 프로세스에게 메모리를 할당하는 시스템 콜입니다. 여러 스레드가 동시에 메모리 할당을 요청하더라도 할당해주는 공간이 서로 겹치면 안 되며 (예를 들어, 두 스레드에서 동시에 각각 2개의 페이지에 해당하는 메모리 할당을 요청했다면 총 4개의 페이지가 할당되어야 합니다), 요청받은 크기만큼을 올바르게 할당할 수 있어야 합니다. sbrk 에 의해 할당된 메모리는 프로세스 내의 모든 스레드가 공유 가능합니다.
- kill : 하나 이상의 스레드가 kill 되면 그 스레드가 속한 프로세스 내의 모든 스레드 모두 정리되고 자원을 회수해야 합니다.
- sleep : 한 스레드가 sleep 을 호출하면 그 스레드만 요청된 시간 동안 잠들어 있어야 합니다. 자고 있는 상태에서도 kill에 의해서 종료될 수 있어야 합니다.
- pipe : pipe 는 각 스레드에서 각각 화면에 출력하는 데에 문제가 없도록만 합니다.

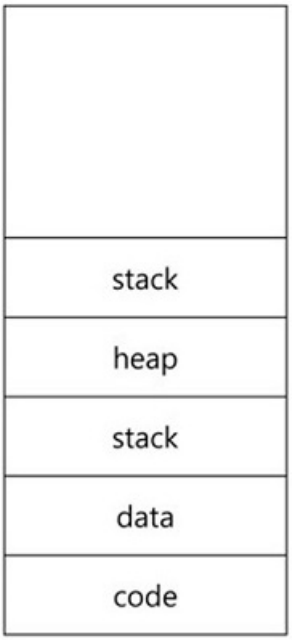
## * 실제 memory address space와 xv6 에서의 memory address space의 차이점

기존 linux의 address space 와 달리 xv6에서는 stack의 크기가 고정적이다. 그리고 linux에서 스택의 시작 위치가 메모리의 맨 끝인것에 비해 xv6는 data영역 바로 위에 stack이 할당되고 그 위로 heap 공간이 할당된다.




[왼쪽 기존의 address space, 오른쪽 xv6의 address space]

따라서, xv6 에서 구현을 진행할 때, stack 방식을 사용하여 아래 번지에서 차근차근 stack과 heap공간을 쌓아나갔다.




추가적으로 exit의 경우, exit한 해당 thread의 stack 공간을 비워주고, 새로운 thread의 stack 공간 할당이 필요하다면 비어있는 공간이 있나 확인 후, 없다면 최상단에 stack을 쌓고 그렇지 않다면, 비워준 공간에 채워주는 방식을 설계를 시도했습니다.

---

# IMPLEMENT

- thread 구현 이전에 proc 구조체에 thread ID, Manage 할 process, return 할 value 값, stack 에 넣어줄 NPROC 크기의 배열, stack의 시작 주소와 끝 주소, 마지막으로 stack의 virtual memory를 추가해 줍니다.

```
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)
```

```
    int tid;            // Thread ID
    struct proc* manager;       // Manager Process
    void *retval;           // return value

    uint stacklist[NPROC + 1];   // Freed stack list
    int start;              // start of stack list
    int end;                // end of stack list
    uint sva;               // Virtual address of stack
};
```

---

## * thread_create

- thread_create 를 구현 시, fork 일 경우와 exec의 파트 두 부분으로 나누어 진행합니다.
- 전체 코드 내용

```
int
thread_create(thread_t *thread, void* (*start_routine)(void *), void *arg)
{
        int i;
        uint sz, sp, ustack[2];
        pde_t *pgdir;
        struct proc *np;
        struct proc *curproc = myproc();

        if(curproc->manager != curproc) {
            return -1;
        }

        // fork part
        // allocproc 호출
        if((np = allocproc()) == 0) {
            return -1;
        }

    // 새로운 프로세스의 공간을 np에 할당
       pgdir = curproc->pgdir;
       if(pgdir == 0) {
        np->state = UNUSED;
        return -1;
       }
       np->parent = curproc->parent;
       *np->tf = *curproc->tf;

       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));

        // manager process의 page table을 가져와 변수를 초기화해주고, tid의 값은 1증가 시켜준다.
       acquire(&ptable.lock);
       np->tf->eax = 0;
       np->pid = curproc->pid;
       np->manager = curproc;
       np->tid = nexttid++;
       release(&ptable.lock);

       // exec part
       // stack list 가 비어있는지 확인 해준다.
       // stack 이 비어있다면 빈공간의 시작주소를 sz에 넣는다.
       if(curproc->start == curproc->end) sz = curproc->sz;
       // 그렇지 않다면, manager process의 sz에 넣어준다.
       else sz = curproc->stacklist[curproc->start];

        // alloccuvm 함수를 호출합니다.
       if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0) {
```

```
            np->state = UNUSED;
            return -1;
        }
        clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
        sp = sz;

        ustack[0] = 0xffffffff;
        sp -= 4;
        ustack[1] = (uint)arg;
        sp -= 4;
        if(copyout(pgdir, sp, ustack, 2*4) < 0)
         {
            np->state = UNUSED;
            return -1;
         }

        np->virtualAddress = sz - 2*PGSIZE;
        if(curproc->start == curproc->end) curproc->sz = sz;
        else
        {
            curproc->sz = curproc->sz;
            curproc->start = (curproc->start+1) % (NPROC+1);
        }

      // stack영역의 초기화가 끝났으면 스레드의 stack 시작 주소를 virtualAddress변수에 넣고 sz, pgdir, eip, esp 변수들도 마저 초기화 해준다.
        np->sz = curproc->sz;
        np->pgdir = pgdir;
        np->tf->eip = (uint)start_routine;
        np->tf->esp = sp;

        *thread = np->tid;
        acquire(&ptable.lock);
        np->state = RUNNABLE;
        release(&ptable.lock);

        return 0;
    }
```

---

## * fork 부분

- allocproc 을 통해 프로세스를 호출합니다.

```
if((np = allocproc()) == 0) {
       return -1;
}
```

- 새로운 프로세스의 공간을 np에 할당해 줍니다.

```
pgdir = curproc->pgdir;
if(pgdir == 0) {
        np->state = UNUSED;
        return -1;
}
np->parent = curproc->parent;
*np->tf = *curproc->tf;

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));
```

- proc.h 에 선언한 manager process의 ptable을 가져온 후, 변수를 초기화 해주고, tid(thread id)를 1증가 시켜줍니다.

```
acquire(&ptable.lock);
np->tf->eax = 0;
np->pid = curproc->pid;
np->manager = curproc;
np->tid = nexttid++;
```

```
release(&ptable.lock);
```

---

## * exec 부분

- proc.h 에 선언한 stack list가 비어있는지 확인을 한 후, 비어있다면, 시작주소를 sz에 할당하고, 비어있지 않다면, manager process에 sz를 할당합니다.

```
// stack list 가 비어있는지 확인 해준다.
// stack 이 비어있다면 빈공간의 시작주소를 sz에 넣는다.
if(curproc->start == curproc->end) sz = curproc->sz;
// 그렇지 않다면, manager process의 sz에 넣어준다.
else sz = curproc->stacklist[curproc->start];
```

- allocuvm 함수를 호출합니다.

```
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0) {
     np->state = UNUSED;
     return -1;
}
```

- sz부터 sz+2*PGSIZE만큼의 공간을 할당해줍니다. 이는 sz ~ sz+PGSIZE 만큼의 protection을 해줍니다.

```
np->virtualAddress = sz - 2*PGSIZE;
if(curproc->start == curproc->end) curproc->sz = sz;
else
{
        curproc->sz = curproc->sz;
        curproc->start = (curproc->start+1) % (NPROC+1);
}
```

- stack영역의 초기화가 끝났으면 스레드의 stack 시작 주소를 proc.h 에 선언한 virtualAddress변수에 넣고 sz, pgdir, eip, esp 변수들도 모두 초기화 해줍니다.

```
np->sz = curproc->sz;
np->pgdir = pgdir;
np->tf->eip = (uint)start_routine;
np->tf->esp = sp;
*thread = np->tid;
```

---

## * thread_exit

- thread_exit 함수는 manager process 에서 남은 thread 자원을 정리한 후, state를 zombie 로 바꿔줍니다.
- 전체 코드

```
void
thread_exit(void *retval)
{
        struct proc *curproc = myproc();
        int fd;

        if(curproc->tid == 0) return;

    //manager process에서 thread의 자원을 정리할 수 있도록 return value를 지정해준다.
        curproc->retval = retval;

    // exit part
    // thread의 state를 ZOMBIE로 바꿔준다.
        for(fd = 0; fd < NOFILE; fd++) {
    if(curproc->ofile[fd]) {
        fileclose(curproc->ofile[fd]);
        curproc->ofile[fd] = 0;
    }
    }

   begin_op();
   iput(curproc->cwd);
   end_op();
   curproc->cwd = 0;

   acquire(&ptable.lock);
   wakeup1(curproc->manager);
   curproc->state = ZOMBIE;
   sched();
   panic("zombie exit");
```

```
}
```

---

- manager process에서 thread의 자원을 정리할 수 있도록 return value를 지정해준다.

```
curproc->retval = retval;
```

---

- thread의 state를 ZOMBIE로 바꿔준다.

```
for(fd = 0; fd < NOFILE; fd++) {
        if(curproc->ofile[fd]) {
            fileclose(curproc->ofile[fd]);
            curproc->ofile[fd] = 0;
        }
}

begin_op();
iput(curproc->cwd);
end_op();
curproc->cwd = 0;

acquire(&ptable.lock);
wakeup1(curproc->manager);
curproc->state = ZOMBIE;
```

---

## * thread_join

- ZOMBIE가 된 스레드들의 자원을 정리해주고, 정리된 stack영역을 stacklist에 넣어줌으로써 빈 공간을 관리한다.
- 전체 코드

---

```
int
thread_join(thread_t thread, void **retval)
{
        struct proc *curproc = myproc();
        struct proc *p;
        int virtualAddress;
        int havekids;
        if(curproc->tid != 0) {
            return -1;
        }

        acquire(&ptable.lock);
        for(;;){
            havekids = 0;
            for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                if(p->manager != curproc || p->tid != thread)
                    continue;
                havekids = 1;
          // ZOMBIE가 된 스레드들의 자원을 정리해준다.
                if(p->state == ZOMBIE){
                    kfree(p->kstack);
                    p->kstack = 0;
                    p->pid = 0;
                    p->parent = 0;
                    p->name[0] = 0;
                    p->killed = 0;
                    p->state = UNUSED;

                    p->manager = 0;
                    p->tid = 0;
                    virtualAddress = p->virtualAddress;
                    p->virtualAddress = 0;

                    deallocuvm(p->pgdir, virtualAddress + 2*PGSIZE, virtualAddress);

                    *retval = p->retval;

                    release(&ptable.lock);
                     // 스레드의 stack영역 역시 정리해주어야 하는데, 정리된 stack영역을 stacklist에 넣어줌으로써 빈 공간을 관리한다.
```

```
                curproc->stacklist[curproc->end] = virtualAddress;
                curproc->end = (curproc->end+1) % (NPROC+1);

                return 0;
            }
          }

          if(!havekids || curproc->killed) {
            cprintf("error\n");
            release(&ptable.lock);
            return -1;
          }
          sleep(curproc, &ptable.lock);
        }
      }
```

---

- ptable을 확인 하면서 zombie가 된 프로세스의 자원을 초기화해 줍니다.

---

```
if(p->state == ZOMBIE){
        kfree(p->kstack);
        p->kstack = 0;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;

        p->manager = 0;
        p->tid = 0;
        virtualAddress = p->virtualAddress;
        p->virtualAddress = 0;

        deallocuvm(p->pgdir, virtualAddress + 2*PGSIZE, virtualAddress);

        *retval = p->retval;
```

---

- 스레드의 stack영역을 정리할 때, 정리된 stack영역을 stacklist에 넣어줌으로써 빈 공간을 관리한다.

---

```
curproc->stacklist[curproc->end] = virtualAddress;
curproc->end = (curproc->end+1) % (NPROC+1);
```

---

---

## * fork

- fork를 통해 생성된 자식 프로세스는 부모 프로세스의 모든 값을 복사해온다.

---

```
if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
```

---

- 부모 프로세스의 stack영역을 제외한 다른 스레드들의 stack영역들은 필요 없으므로 모두 할당해제해준 뒤 stacklist에 넣어서 관리해줍니다.

---

```
if(curproc->pid == p->pid && curproc->tid != p->tid) {
     deallocuvm(np->pgdir, p->virtualAddress + 2*PGSIZE, p->virtualAddress);
     np->stacklist[np->end] = p->virtualAddress;
     np->end = (np->end+1) % (NPROC + 1);
}
```

---

```
 np->virtualAddress = curproc->virtualAddress;
 np->sz = curproc->sz;
 np->parent = curproc;
 *np->tf = *curproc->tf;
```

---

---

## * exec

- exec 에서 남은 thread를 정리하기 위해 다음과 같이 clean_threads 라는 함수를 구현합니다.

---

```
void
clean_threads(int pid, int tid) {
        struct proc *p;
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if(pid == 0)
                continue;
            if(p->pid == pid && p->tid != tid) {
                kfree(p->kstack);
                p->kstack = 0;
                p->pid = 0;
                p->parent = 0;
                p->name[0] = 0;
                p->killed = 0;
                p->state = UNUSED;
            }
        }
        release(&ptable.lock);
}
```

---

- exec 함수에서 해당 thread를 제외한 나머지 thread는 위의 clean_threads 함수를 사용하여 초기화해줍니다.

---

```
clean_threads(curproc->pid, curproc->tid);

curproc->tid = 0;
curproc->manager = curproc;
curproc->start = 0;
curproc->end = 0;
curproc->retval = 0;
```

---

## * exit

- exit 역시 exec와 유사하게 현재 thread를 제외한 나머지 thread를 정리해 줍니다. 정리한 후, 현재 thread의 상태를 zombie 상태로 바꿔줍니다.

---

```
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if(p->pid == curproc->pid && p != curproc) {
            kfree(p->kstack);
            p->kstack = 0;
            p->pid = 0;
            p->parent = 0;
            p->name[0] = 0;
            p->killed = 0;
            p->state = UNUSED;
        }
}
release(&ptable.lock);
```

---

## * sbrk

- 모든 process는 동일한 메모리 사이즈를 갖게하기위해 growproc이 호출될 때 마다, 모든 프로세스의 메모리 사이즈를 변경해줍니다.

---

```
int
sys_sbrk(void)
{
      int addr;
      int n;

      if(argint(0, &n) < 0)
            return -1;
      addr = myproc()->sz;
      // growproc에서 수정을 진행해야한다.
      if(growproc(n) < 0)
```

```
        return -1;
      return addr;
    }
}
```

---

- growproc에서 구체적으로 추가된 부분

---

```
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if(p->pid == 0)
            continue;
            if(p->pid == curproc->pid)
                p->sz = sz;
}
```

---

---

### * 나머지 kill, sleep, pipe 는 xv6에 있는 그대로 구현

---

---

# RESULT

## 1. thread_test

- Test_1 : Basic test

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

- Test_2 : Fork test

```
Test 2: Fork test
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Child of thread 0 start
Child of thread 1 start
Child of thread 2 start
Child of thread 3 sChild of thread 4 start
tart
Child of thread 0 end
Child of thread 1 end
Child of thread 2 end
Thread 0 end
Thread 1 end
Thread 2 end
Child of thread 3 end
Child of thread 4 end
Thread 3 end
Thread 4 end
Test 2 passed
```

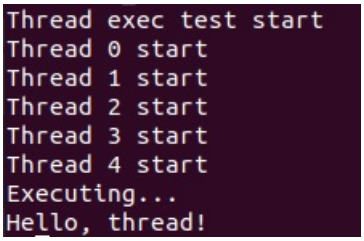
- Test_3 : Sbrk test

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

All tests passed!
```

---

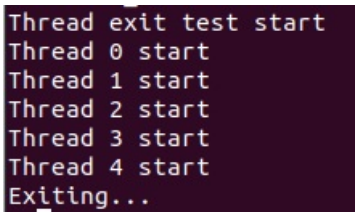
## 2. thread_exec




- 만들어지는 스레드 중 하나가 exec로 hello_thread 프로그램을 실행합니다.
- exec가 실행되는 순간 다른 스레드들은 모두 종료되어야 합니다.

---

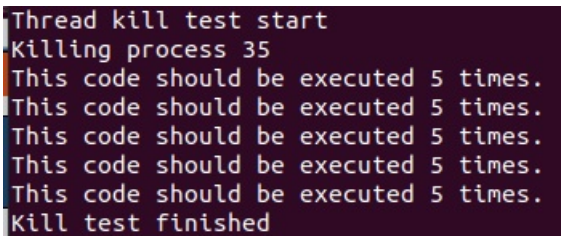
### 3. thread_exit




- 하나의 스레드에서 exit이 호출되면 그 프로세스 내의 모든 스레드가 모두 종료되어야 합니다.

---

### 4. thread_kill




- 프로그램이 fork를 통해 두 개의 프로세스로 나누어진 뒤, 각각 5개씩의 스레드를 생성합니다. 그 중 부모 프로세스 쪽의 스레드 하나가 자식 프로세스를 kill 합니다.

---

## TROUBLE SHOOTING

---

### 1. SBRK

sbrk 는 프로세스에게 메모리를 할당하는 시스템 콜입니다. 처음 메모리를 할당할 경우, sbrk 시스템콜 함수 내에서 구현을 시도하였습니다. 하지만, ptable을 불러올 수 없어서 여러개의 헤더파일을 include하고 했지만, 정상적으로 진행되지 않았습니다. 그리하여 안의 내부에서 호출하는 함수은 growproc 에서 ptable을 불러와 p->sz 에 해당 sz 를 넣어주는 방식으로 진행하여 해결했습니다.

---

### 2. thread 초기화 문제

thread 를 초기화하기 위해 ptable만 정상적으로 구현된다면 문제가 없을 것이라고 생각해서, ptable에 집중하여 구현했습니다. 따라서 proc 구조체에 새로정의한 manager process 에서만 메모리를 할당했으나, 실제 sz 값과 다르다는 것을 cprint 문을 찍어가면서 확인했고, sbrk 와 heap 에 추가적으로 할당해주면서 문제를 해결했습니다.