운영체제 Project 02 Wiki

해당 문서는 운영체제 Project 02(Process management & LWP)에 대한 wiki문서이다.

 한양대학교 컴퓨터소프트웨어학부 2019067429 한승우

1. Design

1) Process Management

a. exec2

- exec2는 exec와 비슷한 동작을 수행하지만, stacksize를 추가로 받는다는 것에 차이가 있다.
- 즉, exec에서 스택용 페이지 1개, 가드 페이지 1개 할당 받는 것 대신에, **stacksize만큼 스택용 페이지를 할당 받는 것으로 구현 할 것이다.**

b. setmemorylimit

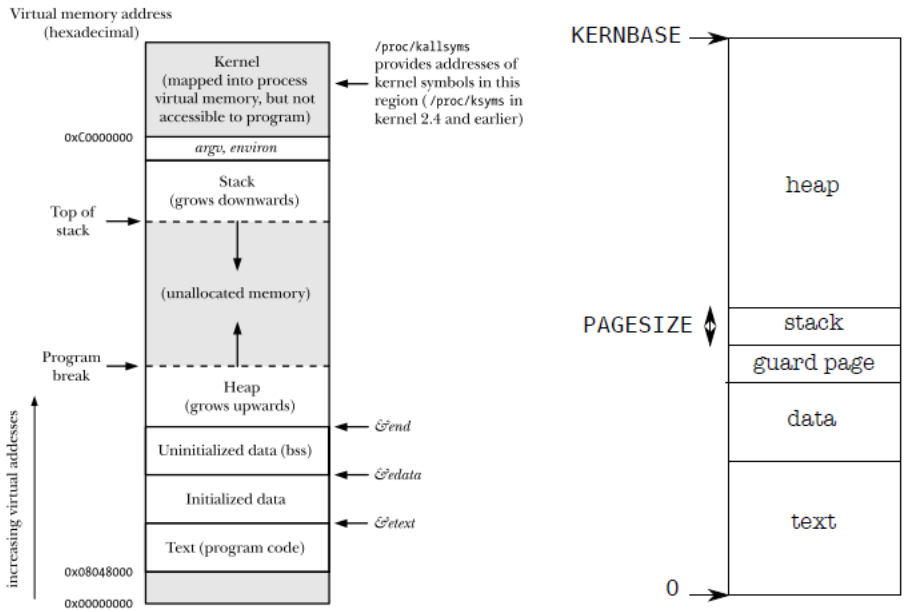
- setmemorylimit는 pid를 통해서 메모리의 limit를 결정하는 system call이다. **proc.h에 memorylimit를 결정하는 속성을 만든 뒤 해당 값만 수정해 줄 것이다.**

c. pmanager

- **sh**와 비슷한 구조로 입력을 받아 명령어들을 수행하도록 만들 것이다.

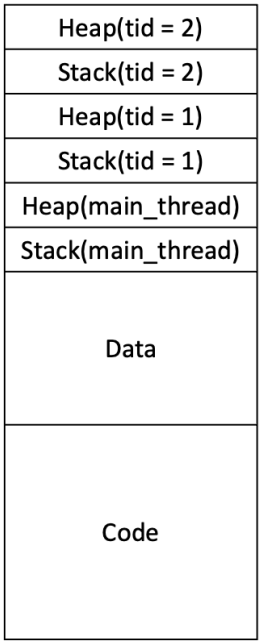
2) LWP

a. LWP의 구조



왼쪽은 리눅스에서의 프로세스, 오른쪽은 xv6에서의 프로세스

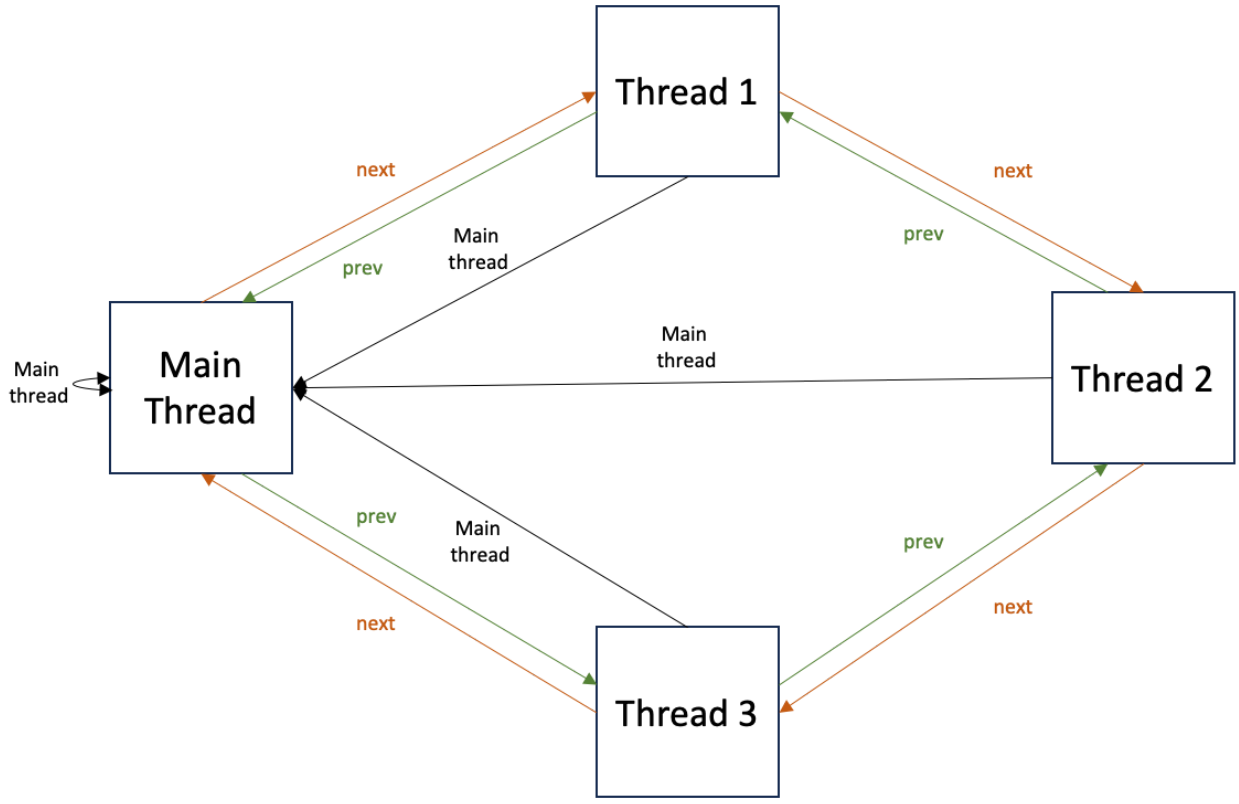
- xv6에서의 프로세스가 가지는 메모리는 **원래 알고 있던 메모리 공간 할당과는 조금 다르게 작동한다.**
- LWP를 구현하기 위해서는 프로세스가 어떻게 메모리 공간을 가지는지 확실히 알 필요가 있는데, thread는 프로세스와 동일한 heap, data, text영역을 참조하고, **stack영역만 각자가 가져야한다.**
- 이를 구현하기 위해서 원래 프로세스에서 쓰레드가 생성되면 그에 해당하는 **stack영역만 계속 쌓아주는 구조로 구현할 것이다.**
- 즉, **process 자체를 main thread로 정의**하고, 해당 process가 thread를 생성 해줄 때마다, **main thread의 메모리 영역을 공유하되, stack영역은 thread마다 따로 가질 수 있도록 할 것이다.**



Process의 메모리 구조, thread의 개수는 2개

- 요약하자면 프로세스가 생성되면 stack, data, text, heap영역을 가지는데, 이때 생성된 stack 영역은 process, 즉 main thread가 가지는 영역이고, **이후 thread가 가지는 stack과 heap영역을 쌓아주는 구조로 구현할 것이다.**
- 이때, 만든 process가 main thread면 process **그렇지 않다면 thread로 취급한다.**

b. LWP의 동작



LWP의 동작방식

- thread의 관리는 이중연결리스트를 통해 구현 할 것이다. **모든 쓰레드들은 main thread를 가르키고 있고, 각 쓰레드의 앞, 뒤를 연결하여 circular하게 구현할 것이다.**
- 이와 같이 구현해야 main thread에서 각 **thread를 관리하는 것에 용이할 것 같다는 생각이 들어**, 해당 구조를 떠올렸다.

c. thread_create

- 해당 구조에서, 사실상 thread는 **새로운 process를 만드는 것이며 다른점은, process안의 모든 thread들이 메모리 영역을 공유하고 stack 영역만 각 thread가 소유하는 방식이다.**
- **새로운 process를 만드는 것 → fork**
- **memory stack 할당 → exec**
- 즉, fork와 exec를 적절히 혼합하여 thread_create를 구현 할 것이다.

d. thread_exit

- thread_exit는 기본적으로 exit과 비슷하게 구현하지만, **proc.h의 속성에 retval을 추가하여, 인자로 넘겨받은 retval을 종료하려고 하는 thread에 저장해 줄 것이다.**

e. thread_join

- thread_join은 wait와 비슷한 동작을 한다.
- 하지만, 자식 중 zombie상태인 모든 process를 정리하는 wait와는 다르게 **인자로 넘겨받은 tid가 같은 thread가 있다면 정리하고, 해당 thread의 return value를 반환하게 해줄 것이다.**

---

2. Implement

1) Changed Process

a. proc.h

```
// 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)

  //For process management
  uint mem_limit;              // mem limit
  uint pages;                  // number of pages

  //For LWP
  thread_t tid;                // Thread ID
  int maxtid;                  // Numbers of Thread
  int is_thread;               // Check for main thread
  struct proc *main_thread;    // Main thread
  struct proc *next_thread;    // Next thread
  struct proc *prev_thread;    // Prev thread
  uint ustack;                 // stack address that the thread is using
  int stack[NTHREAD];          // To fine empty space
  void *retval;                // return value
};
```

- thread와 process 모두, 동일한 proc 구조체를 가지면서 ptable을 통해 스케줄링의 대상이 된다.
- 그 이유는 **process도 main thread라고 볼 수 있기에, 각 thread들과, process의 구조체는 동일하게 사용하였다.**
- 요약하자면 모든 thread들을 **메모리 영역만 공유하는 각각의 process로 생각하여** 해당 구현을 진행하였다.
- 구조체의 각 변수들을 설명하자면,
  - mem_limit → 메모리의 limit를 결정하기 위한 변수이다.
  - pages → 각 thread, process가 가지는 page의 개수이다.
  - tid → thread의 ID number이다.
  - maxtid → process가 가지고 있는 thread의 개수이다.
  - is_thread → 해당 proc 변수가 thread인지 process인지 확인하기 위한 변수이다.
  - main_thread → thread들의 main thread를 가르키는 변수이다.

- next_thread → 다음 thread를 가르키는 변수이다.
- prev_thread → 이전 thread를 가르키는 변수이다.
- ustack → 해당 thread가 사용하고 있는 stack 주소를 담기 위한 변수이다.
- stack[] → process(main thread)에서 비어있는 stack 공간을 찾기 위한 변수이다
- retval → thread의 return value를 담기 위한 변수이다.

2) Process Management

a. exec2

```
//만약 현재 limit보다 새로 할당받은 후의 sz값이 크다면 수행하지 않음.
  if(sz+ (stacksize+1)*PGSIZE > curproc->mem_limit && curproc->mem_limit != 0){
    return -1;
  }
  //stacksize만큼 stack page를 할당, 1은 guard page를 위한 것.
  if((sz = allocuvm(pgdir, sz, sz + (stacksize+1)*PGSIZE)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz - (stacksize+1)*PGSIZE));

...

//현재 process의 스택용 페이지를 설정
curproc->pages = stacksize;
```

- exec2는 기본적으로 exec과 동일하게 동작한다.
- 바뀐 부분은 exec는 2*PGSIZE만큼 페이지를 할당하지만, **exec2는 인자로 받은 stacksize + 1(가드용 페이지) 만큼 페이지를 할당해야 한다는 점이다.**
- 이후, exec2를 실행시킨 curproc의 pages를 인자로 받은 stacksize로 설정해 주었다.

b. setmemorylimit

```
//set memory limit
int setmemorylimit(int pid, int limit) {
  struct proc *curproc = myproc();
  struct proc *p;
  struct proc *p1;
  acquire(&ptable.lock);

  //process가 존재하는지에 대한 여부
  int possible = 0;

  //인자의 pid와 동일한 process가 있는지 확인.
  for(p=ptable.proc; p < &ptable.proc[NPROC];p++){
    if(p->pid==pid && p->is_thread == 0){
      possible = 1;
      break;
    }
  }

  //process가 존재하지 않는 경우
  if(possible == 0){
    release(&ptable.lock);
    return -1;
  }

  //process는 존재하지만, limit의 값이 음수이거나 이미 process의 sz값이 limit보다 큰 경우
  if(limit < 0 || p->sz > limit){
    release(&ptable.lock);
    return -1;
  }

  //정상적으로 동작한 경우
  else{
    p = p->main_thread;
    p->mem_limit = limit;
```

```
    for(p1 = p->next_thread ; p1 != p; p1 = p1->next_thread)
      p1->mem_limit = limit;
    release(&ptable.lock);
  }
  return 0;
}
```

- setmemorylimit 시스템 콜은 pid와 limit를 인자로 받아 해당 process의 memory limit을 결정한다.
- ptable을 순회하며 인자로 받은 pid와 같은 process가 있는지 확인한다.
- 만약 process가 존재하지 않다면 -1을 return한다.
- 만약 해당 process의 sz값이 이미 limit보다 크거나, 인자로 받은 limit의 값이 음수라면 -1을 return한다.
- **정상적으로 동작한 경우 process의 mem_limit를 limit로 설정해주고 0을 return한다.**

c. pmanager

```
#include "types.h"
#include "user.h"
#include "fcntl.h"

// Parsed command representation
#define LIST  1
#define KILL  2
#define EXECUTE  3
#define MEMLIM  4
#define EXIT  5

#define MAX_INPUT_SIZE 100

int fork1(void);

//Fork
int
fork1(void)
{
  int pid;

  pid = fork();
  if(pid == -1);
  return pid;
}

void
runcmd(char * arg)
{

    //list 명령
    if (arg[0] == 'l' && arg[1] == 'i' && arg[2] == 's' && arg[3] == 't') {
        print_all();
    }

    //kill 명령
    else if(arg[0] == 'k' && arg[1] == 'i' && arg[2] == 'l' && arg[3] == 'l'){
        //parsing
        char *temp = (char*)malloc(sizeof(char)*15);
        int index = 0;
        for(int i = 5; i<strlen(arg); i++){
            temp[index] = arg[i];
            index++;
        }
        temp[index] = '\0';
        int pid  = atoi(temp);

        //pid를 가진 process를 kill
        if(kill(pid) == 0){
            printf(1, "Success!\n");
        }
        else{
            printf(1, "Fail!\n");
        }
    }
```

```
//execute 명령
else if(arg[0] == 'e' && arg[1] == 'x' && arg[2] == 'e' && arg[3] == 'c' && arg[4] == 'u' && arg[5] == 't' && arg[6] == 'e'){
    //parsing
    char *temp1 = (char*)malloc(sizeof(char)*15);
    char *temp2 = (char*)malloc(sizeof(char)*15);

    int index = 8;
    int j = 0;
    int k = 0;

    while(1){
        if(arg[index] == 32){
            break;
        }
        temp1[j] = arg[index];
        index++;
        j++;
    }
    temp1[j] = '\0';
    index++;
    while(1){
        if(arg[index] == '\0'){
            break;
        }
        temp2[k] = arg[index];
        index++;
        k++;
    }
    temp2[k] = '\0';
    int stack = atoi(temp2);
    char *argv2[1] = {temp1, 0};

    //이어서 pmanager가 실행되기 위해 fork 후 exec
    if(fork1() == 0){
        if(exec2(temp1, argv2, stack) == 0){
            printf(1, "Success!\n");
        }
        else{
            printf(1, "Fail!\n");
        }
    }
}

//memlim 명령
 else if(arg[0] == 'm' && arg[1] == 'e' && arg[2] == 'm' && arg[3] == 'l' && arg[4] == 'i' && arg[5] == 'm'){
    //parsing
    char *temp1 = (char*)malloc(sizeof(char)*15);
    char *temp2 = (char*)malloc(sizeof(char)*15);
    int index = 7;
    int j = 0;
    int k = 0;

    while(1){
        if(arg[index] == 32){
            break;
        }
        temp1[j] = arg[index];
        index++;
        j++;
    }
    temp1[j] = '\0';
    int pid = atoi(temp1);
    index++;
     while(1){
        if(arg[index] == '\0'){
            break;
        }
        temp2[k] = arg[index];
        index++;
        k++;
    }
    temp2[k] = '\0';
    int limit = atoi(temp2);

    //memory limit를 설정
    if(setmemorylimit(pid, limit) == 0){
        printf(1, "Success!\n");
```

```
        }
        else{
            printf(1, "Fail!\n");
        }

    }
    exit();
}

int main(int argc, char *argv[]) {
    while (1) {
        printf(1, "-> ");
        char arg[MAX_INPUT_SIZE];
        gets(arg, MAX_INPUT_SIZE);
        arg[strlen(arg)-1] = 0;

        //Exit 명령, pmanger를 종료
        if(strcmp("exit", arg) == 0){
            printf(1, "Exit pmanager\n");
            break;
        }
        //fork 후 명령을 수행
        if(fork1() == 0)
            runcmd(arg);
        wait();
    }
    exit();
}
```

- pmanager의 구조는 기본적으로 sh.c와 비슷하도록 구현하였다.
- 무한 루프를 사용하여 명령을 계속해서 받아오고, **fork를 수행한 후 해당 명령을 동작하도록 해주었다.**

1. **list command**

```
void print_all(void){
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc ; p< &ptable.proc[NPROC]; p++){
    //실행중인 프로세스가 아닌경우
    if(p->state == UNUSED || p->state == EMBRYO || p->state == ZOMBIE)
      continue;
    //쓰레드가 아닌 프로세스인 경우
    if(p->is_thread == 0 && p->killed != 1)
      cprintf("1.process name : %s     2.pid : %d     3.number of stack pages : %d      4.stack size : %d      5.memory limit : %d
  }
  release(&ptable.lock);
  return 0;
}
```

   - list 명령은 print_all이라는 시스템 콜을 사용하여 구현해 주었다.
   - print_all 시스템 콜은 **ptable을 모두 순회하며 실행 중인 프로세스들의 정보를 출력한다.**
   - 이때, 실행 중인 프로세스라고 하는 것은 현재 혹은 미래에 스케줄링의 대상이 될 수 있냐에 따라 구분 하였다. **(UNUSED, EMBRYO, ZOMBIE는 현재에도 미래에도 스케줄링이 될 수 없다.)**
   - 이때, **thread인 경우 출력하지 않아야 하기에, 프로세스인지 확인 해주어 출력해 주었다.**

2. **kill command**
   - kill 명령은 단순히, 넘겨 받은 **pid에 해당하는 process에 대해서 kill 시스템 콜을 수행한다.**
3. **execute command**
   - execute 명령은 path와 stacksize를 받은 후 , exec2 시스템 콜을 수행한다.
   - 이때, exec2를 수행한 뒤에도 pmanager가 계속 동작 할 수 있도록 **fork를 사용하여 자식 프로세스가 exec를 수행 할 수 있도록 하였다.**
4. **memlim command**

- memlim 명령은 **setmemorylimit 시스템 콜을 이용해서 수행하게 해주었다.**

5. **exit command**
   - exit 명령이 들어오게 되면 **pmanager process가 exit를 수행하게 해주었다.**

3) LWP

a. thread_create

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

  //routine like fork
  //thread를 create하는 것을 체크
  t_create = 1;
  if ((np = allocproc()) == 0) {
    return -1;
  }


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

  for(int 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));

  np->pgdir=curproc->pgdir;

  //routine like exec

  //stack에 빈공간이 있는지 확인
  for(i = 0; i < NTHREAD; i++)
    if(curproc->stack[i] != -1)
      break;

  //빈공간이 있다면, 그 자리에 할당
  if(i != NTHREAD) {
   sp = curproc->stack[i];
   np->ustack = sp;
   curproc->stack[i] = -1;
  }
  //없다면, 새로 공간을 할당
  else {
    sz = curproc->sz;
    sz = PGROUNDUP(sz);
    //프로세스의 메모리 제한 확인
    if(sz+ 2*PGSIZE > curproc->mem_limit && curproc->mem_limit != 0){
      return -1;
    }
    //제한 사항이 없다면 allocuvm 수행
    if( (sz = allocuvm(curproc->pgdir, sz, sz + 2*PGSIZE)) == 0) {
      kfree(np->kstack);
      np->kstack=0;
      np->parent = 0;
      np->name[0] = 0;
      np->killed = 0;
      np->is_thread = 0;
      np->mem_limit = 0;
      np->pages = 0;
      np->retval = 0;
      np->pid = 0;
      np->tid = 0;
      np->state = UNUSED;
```

```
      for(i = 0; i < NTHREAD; i++)
        np->stack[i] = -1;
      release(&ptable.lock);
      return -1;
    }
    clearpteu(curproc->pgdir, (char*)(sz - 2*PGSIZE));

    sp = sz;
    curproc->sz = sz;
    np->ustack = sz;
  }

  //thread의 구조체 변수 초기화
  np->sz=curproc->sz;
  np->is_thread=1;
  np->pid = curproc->pid;
  np->parent = curproc->parent;
  np->main_thread = curproc;
  np->tid=curproc->maxtid++;
  *np->tf=*curproc->tf;
  np->next_thread = curproc;
  np->prev_thread = curproc->prev_thread;
  np->prev_thread->next_thread = np;
  curproc->prev_thread = np;

  *thread=np->tid;

  //ustack 할당
  ustack[0] = 0xffffffff;
  ustack[1] = (uint) arg;

  sp -= 8;

  if(copyout(np->pgdir,sp,ustack,8)<0){
    kfree(np->kstack);
    np->kstack=0;
    np->parent = 0;
    np->name[0] = 0;
    np->killed = 0;
    np->is_thread = 0;
    np->mem_limit = 0;
    np->pages = 0;
    np->retval = 0;
    np->pid = 0;
    np->tid = 0;
    np->state = UNUSED;
    for(i = 0; i < NTHREAD; i++)
      np->stack[i] = -1;
    release(&ptable.lock);
    return -1;
  }


  np->tf->eip=(uint)start_routine; //thread의 시작 위치
  np->tf->esp=sp;

  //process들의 모든 thread들의 stack size 동기화
  struct proc* p;

  for(p=ptable.proc;p<&ptable.proc[NPROC];p++){
    if(p->pid==np->pid){
      p->sz=np->sz;
    }
  }

  np->state=RUNNABLE;
  release(&ptable.lock);
  return 0;
}
```

- thread를 만드는 것은, 단순히 다른 process들과 주소 공간을 공유하는 **또 다른 process를 만드는 것이라고 생각했다.**
- 이 유추의 결론은, **thread create는 단순히 fork와 exec를 적절히 혼합하여 구현하면 된다는 것이다.**
- thread create의 진행 순서는 다음과 같다.

1. **allocproc**

   → thread들을 단지 주소 공간을 공유하는 또다른 process들이라고 생각했기에 **allocproc을 진행하여 process를 만들어주었다.**

2. **fork**

   → fork에서 마킹을 한 부분은,새로 만든 프로세스(thread)의 ofile을 설정하는 것과, 구조체 변수들을 설정하는 부분이다.

3. **exec**

   → exec에서 마킹을 해온 부분은, allocuvm을 사용하여 새로 2개의 page table을 할당하는 부분과, ustack을 할당하고 copyout 함수를 수행하여 할당한 사용자 영역의 메모리를 커널영역에 복사해주는 부분이다. 이때, **할당한 stack pointer를 trap frame의 esp에 넣어준다.**

   → 추가로, exec에서는 trap frame의 eip를 elf.entry로 설정하는 것과는 다르게, **thread create에서는 인자로 넘겨받은 start_routine을 설정해준다.**

4. **sz 설정**

   → 같은 process에 속한 모든 thread들의 sz 값은 동일해야 하므로, **ptable을 순회하면서 sz의 값을 동기화 시켜준다.**

- 이때, thread가 종료되어 해당 thread가 사용하던 **stack 공간이 비어있음을 확인하여 재사용하게 해주었다.**
  - **비어 있으면 해당 공간에 새로 할당한 thread를 넣어주고, 그렇지 않다면 allocuvm을 사용하여 다시 메모리 공간을 확장해 주었다.**

b. thread_exit

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

  //Close all open files.
  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->main_thread);

  // 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);
    }
  }

  // Jump into the scheduler, never to return.
  curproc->state=ZOMBIE;
  // thread의 return value를 설정
  curproc->retval=retval;
  sched();
  panic("zombie exit");
}
```

- thread_exit는 기존 exit의 대부분을 마킹해왔다.
- 달라진 부분은, 자신의 state를 ZOMBIE로 만들고 부모 프로세스를 깨우는 동작방식과는 다르게, **자신의 main thread를 깨우는 것이다.**
- 또한, 종료하는 thread의 return value를 인자로 받은 retval로 설정해준다.

c. thread_join

```
//thread_join
int thread_join(thread_t thread, void **retval){

  struct proc *p;
  struct proc *curproc = myproc()->main_thread;
  int i, havethread;

  acquire(&ptable.lock);

  for(;;){
    havethread = 0;
    //연결된 thread들을 순회
    for(p = curproc->next_thread ; p != curproc; p = p->next_thread){
      if(p->tid!=thread)
        continue;
      //인자로 들어온 tid와 같은 thread가 있는 경우
      havethread = 1;
      if(p->state == ZOMBIE){
        kfree(p->kstack);
        for(i = 0; i < NTHREAD; i++)
          if(curproc->stack[i] == -1)
            break;
        curproc->stack[i] = p->ustack;
        p->kstack = 0;
        p->next_thread->prev_thread = p->prev_thread;
        p->prev_thread->next_thread = p->next_thread;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->is_thread = 0;
        p->main_thread = 0;
        p->prev_thread = 0;
        p->next_thread = 0;
        p->mem_limit = 0;
        p->pages = 0;
        p->tid = 0;
        p->state = UNUSED;

        *retval=p->retval;
        release(&ptable.lock);
        return 0;
      }

    // No point waiting if we don't have any thread.
    if(!havethread || curproc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for thread to exit.
    }
    sleep(myproc(), &ptable.lock);  //DOC: wait-sleep

  }
    return -1;
}
```

- thread_join은 기본적으로 기존 wait와 비슷하게 동작한다.
- **join을 요청받은 thread는 연결 리스트로 연결 되어 있는 thread들을 순회하며 인자로 받은 tid와 일치하는 thread가 있는지 확인한다. 만약 존재한다면 정리해주고 0을 반환한다.**
- 이때, 정리한 thread의 stack 공간을 재사용 하기위해 **main thread의 stack 배열에 정리한 thread의 stack 주소를 넣어준다.**

d. fork

```
  np->sz = curproc->main_thread->sz;
//새로 만든 process의 page, memory limit를 설정
```

```
  np->mem_limit = curproc->mem_limit;
  np->pages = curproc->pages;
```

- 현재 구현방식에서 thread들은 단지, **메모리 영역을 공유하는 process에 불과하기에, 기존 fork에서 변화할 부분이 크게 없다.**
- 하지만, fork시에 자식 프로세스는 부모 프로세스의 **memory limit, stack page, stack size등을 동일하게 가져야 하므로 해당 코드만을 추가해 주었다.**

e. exec

```
//exit_thread
void exit_thread(struct proc * curproc){

  struct proc *p;
  struct proc *mthread = curproc->main_thread;
  int i, fd;

  //close all open files
  for(p = curproc->next_thread ; p != curproc; p = p->next_thread){
      if(p->state != ZOMBIE) {
        for(fd = 0; fd < NOFILE; fd++) {
          if(p->ofile[fd]) {
            fileclose(p->ofile[fd]);
            p->ofile[fd] = 0;
          }
        }

        begin_op();
        iput(p->cwd);
        end_op();
        p->cwd = 0;
      }
  }
  acquire(&ptable.lock);

  //exec을 호출한 thread가 main thread가 된다.
  curproc->pid = mthread->pid;
  curproc->parent = mthread->parent;
  curproc->killed = mthread->killed;
  curproc->tid = 0;
  curproc->maxtid = 0;
  curproc->is_thread = 0;

  for(i = 0; i < NTHREAD; i++)
    curproc->stack[i] = -1;


  //자신을 제외한 모든 thread를 정리
  for(p = curproc->next_thread; p != curproc; p = p->next_thread){
      kfree(p->kstack);
      p->kstack=0;
      p->parent = 0;
      p->name[0] = 0;
      p->killed = 0;
      p->is_thread = 0;
      p->mem_limit = 0;
      p->pages = 0;
      p->retval = 0;
      p->pid = 0;
      p->tid = 0;
      p->state = UNUSED;
      for(i = 0; i < NTHREAD; i++)
        p->stack[i] = -1;
  }
  curproc->main_thread = curproc;
  curproc->next_thread = curproc;
  curproc->prev_thread = curproc;
  release(&ptable.lock);

}
```

- exec이 실행되게 되면, **기존 process의 모든 thread들이 정리 되어야한다.**
- 이때, 크게 2가지 경우가 존재한다.
  - **process (main thread가 호출)**

    → process가 exec를 호출하게 된다면, 단순히 자신 안에 있는 모든 thread들을 정리하고 exec를 수행하면 된다.
  - **thread가 호출**

    → 만약 process가 아닌 thread가 호출하게 된다면, 해당 thread말고는 main thread(process)를 포함하여 모든 thread들이 정리되게 해주었다.

    → 위와 같이 구현한 이유는, thread가 exec을 호출한다 하더라도, exec를 수행한 뒤에는 더이상 thread가 아닌 **process(main thread)가 되어야 하기 때문이다.**
- 이를 구현하기 위해, **proc.c에서 exit_thread 함수를 추가해 주었다. (그 이유는 ptable lock을 걸어야하지만 exec에서는 불가능하기 때문이다.)**
- 요약하자면 **exec를 호출한 쓰레드는 exit_thread 함수를 통해 자신이 main thread가 되며 나머지 thread들을 모두 정리해준다.**

f. sbrk

```
//memelimit 제한을 확인
  if(sz+n > curproc->mem_limit && curproc->mem_limit != 0){
    return -1;
  }
...
//모든 thread들의 sz를 동기화
  struct proc *p;

  for(p=ptable.proc; p<&ptable.proc[NPROC]; p++){
    if(p->main_thread == curproc->main_thread || p == curproc->main_thread){
      p->sz=curproc->sz;
    }
  }
```

- sbrk 시스템 콜은 내부에서 growproc을 호출한다.
- 이때, 모든 thread들은 할당된 메모리 공간을 공유해야 하므로, **ptable을 한번 순회하며 sz값을 동기화 시켜준다.**
- 또한, **allocuvm을 통해서 메모리 공간을 할당해주기 전에 memlimit를 확인 해주어 예외처리 해준다.**

g. kill

```
int
kill(int pid)
{
  struct proc *p;
  struct proc *p1;

  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->state == SLEEPING)
        p->state = RUNNABLE;

      //kill all threads
      p = p->main_thread;

      for(p1 = p->next_thread; p1 != p; p1 = p1->next_thread){
        p1->killed = 1;
```

```
        if(p1->state == SLEEPING)
          p1->state = RUNNABLE;
      }
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

- kill 시스템 콜은 thread 하나에 대해 kill 시스템 콜을 수행하면 해당 프로세스 내의 모든 thread가 종료 되어야한다.
- 만약, 인자로 받은 pid와 동일한 process를 발견하게 된다면, **해당 process에 속한 모든 thread들도 똑같이 killed 변수를 1로 설정하게 해주었다.**

h. sleep

- 해당 구현에서 thread들은 process와 동일하게 취급 받기에, **sleep 시스템 콜은 동일하게 구현하였다.**

i. pipe

- sleep와 마찬가지로 thread들은 **process와 동일하게 취급 받기에 기존 xv6와 동일하게 구현하였다.**

---

3. Result

1)Pmanager

```
#include "types.h"
#include "stat.h"
#include "user.h"

#define NUM_THREAD 5

void*
sbrkthreadmain(void *arg)
{
  int tid = (int)arg;
  char *oldbrk;
  char *end;
  char *c;
  oldbrk = sbrk(1000);
  end = oldbrk + 1000;
  for (c = oldbrk; c < end; c++){
    *c = tid+1;
  }

  for (c = oldbrk; c < end; c++){
    if (*c != tid+1){
      printf(1, "panic at sbrkthreadmain\n");
      exit();
    }
  }
  thread_exit(0);
}

int main(int argc, char* argv[])
{
  thread_t threads[NUM_THREAD];
  int i;
  void *retval;

  for (i = 0; i < NUM_THREAD; i++){
    sleep(100);
```

```
    if (thread_create(&threads[i], sbrkthreadmain, (void*)i) != 0){
      printf(1, "panic at thread_create\n");
      return -1;
    }
  }
  for (i = 0; i < NUM_THREAD; i++){
    if (thread_join(threads[i], &retval) != 0){
      printf(1, "panic at thread_join\n");
      return -1;
    }
  }

  exit();
}
```

- pmanager가 정상적으로 동작하는지 여부에 대해서 테스트 해보기 위한 테스트 코드는 위와 같다.
  - **memlim이 정상적으로 동작하는지 확인해주기 위해 thread가 sbrk를 수행하도록 하였다.**

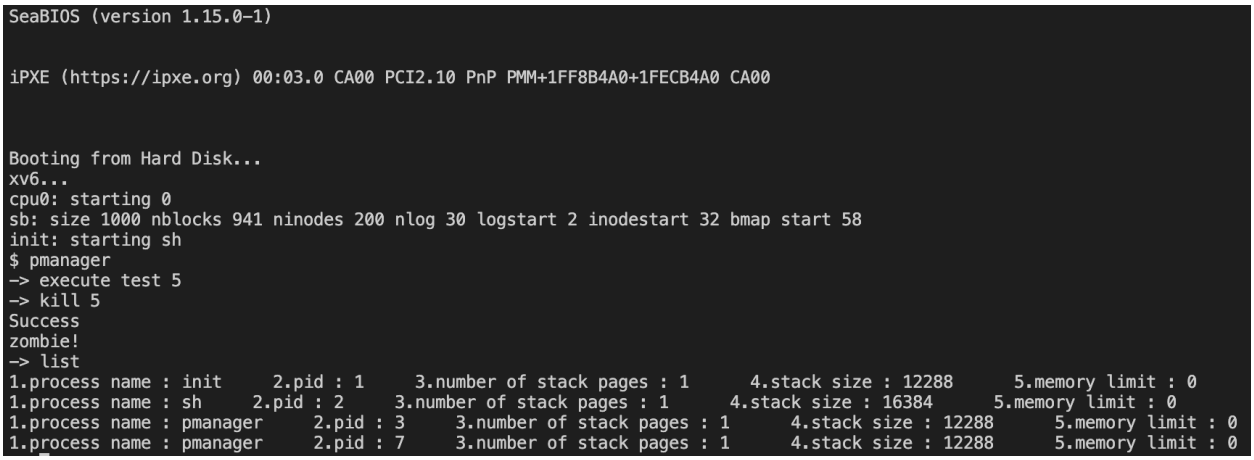
a. list

```
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
-> execute test 5
-> list
1.process name : init     2.pid : 1     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
1.process name : sh     2.pid : 2     3.number of stack pages : 1      4.stack size : 16384      5.memory limit : 0
1.process name : pmanager     2.pid : 3     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
1.process name : pmanager     2.pid : 6     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
1.process name : test     2.pid : 5     3.number of stack pages : 5      4.stack size : 28672      5.memory limit : 0
```

list 명령

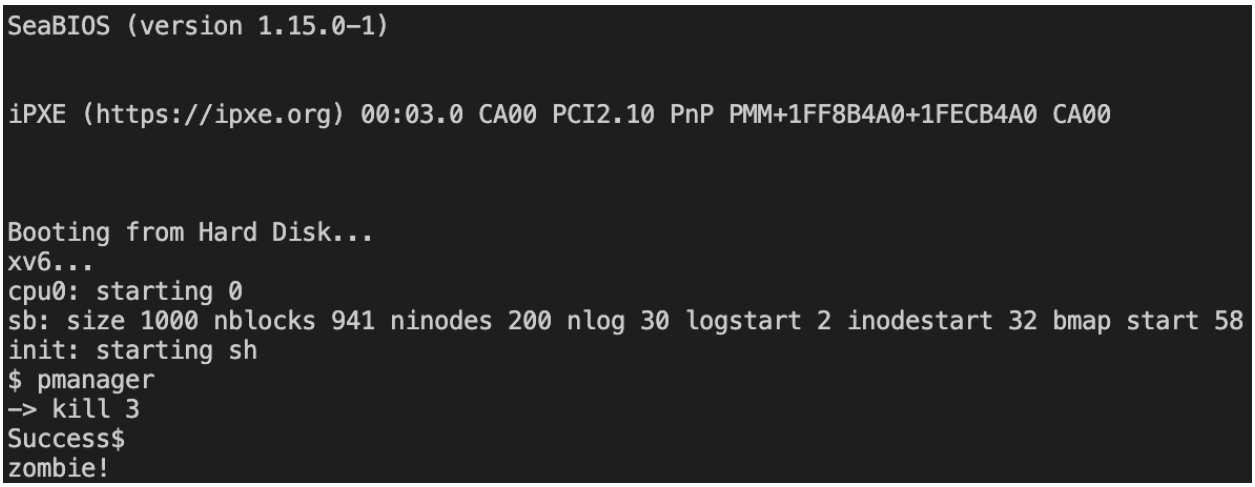
- list명령은 현재 실행중인 프로세스의 정보를 출력한다.
- thread인 경우 출력하지 않는 경우를 체크하기 위해 5개의 thread를 생성하는 test case를 execute하였다.
- 결과는 정상적으로 **process(main thread)만의 정보를 출력하고 stack page, stack size도 정상적으로 출력이 되었다.**
- 한가지 의문점은 thread를 5개 만들때, 각 thread에 page table을 2개씩 할당하므로 적어도 49152(thread들이 10개, process(main thread)가 2개) byte의 크기를 가져야한다고 생각하였다.
- 하지만, **stack공간을 재활용하는 구현의 특성 상 thread_create 함수와, thread_joint함수의 호출 순서에 따라 stack이 재활용 되므로 thread로 인한 stack size는 정확히 알 수 없다는 결론이었다.**
- 추가로, pmanager는 한 번의 fork 후에 명령어를 수행하므로 **list출력 시에 2개의 pmanager가 나온다.**

b. kill



kill 명령(kill testcase)

- kill 명령은 인자로 넘겨준 pid를 가진 process를 kill한다.
- 해당 테스트 케이스에서는 execute한 test 프로세스를 kill하였고, 성공 메시지와 함께 정상적으로 종료되었다.
- 이때, 해당 **test case process 내에 5개의 thread가 있었지만 정상적으로 종료됨을 확인하였다.**



kill 명령(kill pmanager)

- 조금 더 확실하게 process kill 여부를 테스트 하기 위해, **현재 pid가 3인 pmanager자체를 kill하는 test를 진행해보았다.**
- 정상적으로 해당 process가 kill됨을 확인하였다.

c. execute

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 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
$ pmanager
-> execute test 5
-> list
1.process name : init     2.pid : 1     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
1.process name : sh     2.pid : 2     3.number of stack pages : 1      4.stack size : 16384      5.memory limit : 0
1.process name : pmanager     2.pid : 3     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
1.process name : test     2.pid : 5     3.number of stack pages : 5      4.stack size : 37864      5.memory limit : 0
1.process name : pmanager     2.pid : 6     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
-> zombie!
```

execute 명령

- execute 명령을 통해 하나의 test 프로세스를 실행시킨 결과값이다.
- **인자로 stack page는 5개를 넘겨주었기에, stack size 5를 정상적으로 출력하는 것을 볼 수 있다.**

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 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
$ pmanager
-> execute test 5
-> execute test 5
-> execute test 5
-> execute test 5
-> list
1.process name : init     2.pid : 1     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
1.process name : sh     2.pid : 2     3.number of stack pages : 1      4.stack size : 16384      5.memory limit : 0
1.process name : pmanager     2.pid : 3     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
1.process name : pmanager     2.pid : 12     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
1.process name : test     2.pid : 5     3.number of stack pages : 5      4.stack size : 28672      5.memory limit : 0
1.process name : test     2.pid : 7     3.number of stack pages : 5      4.stack size : 28672      5.memory limit : 0
1.process name : test     2.pid : 9     3.number of stack pages : 5      4.stack size : 28672      5.memory limit : 0
1.process name : test     2.pid : 11     3.number of stack pages : 5      4.stack size : 28672      5.memory limit : 0
```

4개의 process를 execute했을 때,

- 다음은 **여러개의 process를 동시에 execute하는 테스트를 진행하였다.**
- 정상적으로 4개의 process 모두, 실행 가능한 process로 인식되었고, execute 후에도 pmanager가 이어서 실행됨을 확인하였다.

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 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
$ pmanager
-> execute echo 5
-> execute echo 10
-> list
1.process name : init     2.pid : 1     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
1.process name : sh     2.pid : 2     3.number of stack pages : 1      4.stack size : 16384      5.memory limit : 0
1.process name : pmanager     2.pid : 3     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
1.process name : pmanager     2.pid : 8     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
1.process name : echo     2.pid : 5     3.number of stack pages : 5      4.stack size : 28672      5.memory limit : 0
1.process name : echo     2.pid : 7     3.number of stack pages : 10      4.stack size : 49152      5.memory limit : 0
-> zombie!
zombie!
```

echo(stack page 5, stack page 10)

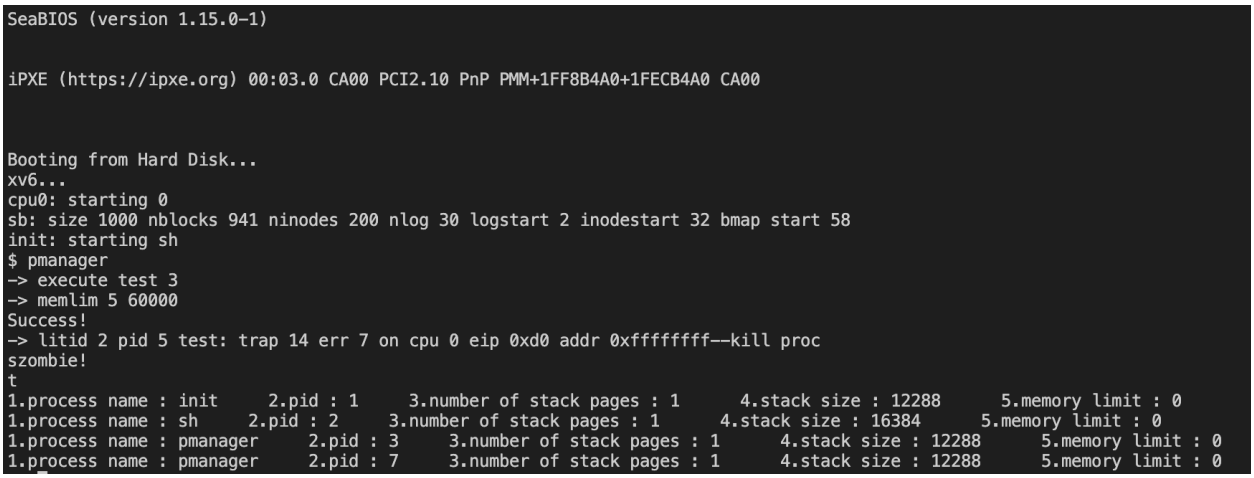
- stack page 할당에 따른 stack size 변화 값을 정확히 측정하기 위해, echo.c를 사용하여 test를 진행하였다.
- **총 11개의 page를 할당 받은 echo process와 6개의 page를 할당 받은 echo process가 존재하였으며 정상적으로 결과 값이 나오는 것을 확인 할 수 있었다.**

d. memlim

```
$ pmanager
-> list
1.process name : init     2.pid : 1     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
1.process name : sh     2.pid : 2     3.number of stack pages : 1      4.stack size : 16384      5.memory limit : 0
1.process name : pmanager     2.pid : 5     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
1.process name : pmanager     2.pid : 6     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
-> memlim 5 70000
Success!
-> list
1.process name : init     2.pid : 1     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
1.process name : sh     2.pid : 2     3.number of stack pages : 1      4.stack size : 16384      5.memory limit : 0
1.process name : pmanager     2.pid : 5     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 70000
1.process name : pmanager     2.pid : 8     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 70000
```

정상적으로 memory limit이 수행된 모습

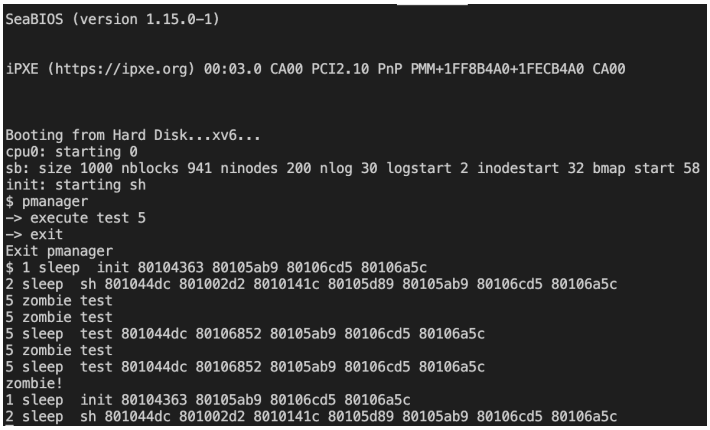
- memlim 명령을 수행하면 해당 pid를 가진 process의 memory limit을 설정해주어야한다.
- 해당 test에서는 pmanager의 memory limit을 70000으로 설정해 주었는데, **pmanager는 fork를 통해 명령들을 수행하므로, list 명령을 수행 할 때의 pmanager 또한 70000으로 설정되어 부모 프로세스의 memory limit을 정상적으로 마킹해오는 것을 알 수 있다.**



memory limit을 초과하여 종료되는 모습

- sbrk를 지속적으로 수행하는 test case를 execute한 후, memory limit을 해당 test case의 요구량보다 낮게 설정해 주었다.
- 결과 값은, **60000에 도달하기 전까지 정상적으로 수행하다 60000을 초과하는 상황에서는 sbrk를 정상적으로 수행하지 못하여 page fault가 발생하는 모습이다.**

e. exit



exit 명령(execute 명령 후)

- exit명령은 pmanger를 즉시 종료하는 명령이다.
- exit명령은 단순히 pmanger를 exit 하면 되는 명령이기에 간단하지만, execute와 동시에 사용 했을 때, pmanager에서 execute를 사용한 process는 pmanager가 exit되었음에도 계속 남아있었다.
- execute 명령이 fork를 한번 더 진행한 후, path에 있는 유저 프로그램을 실행시키기에 옳은 결과 값이였다.
- exit 명령이 수행 되면, **즉시 pmanager를 종료하고 종료 메시지를 출력하게 해주었다.**

2) LWP

```
#include "types.h"
#include "stat.h"
#include "user.h"
```

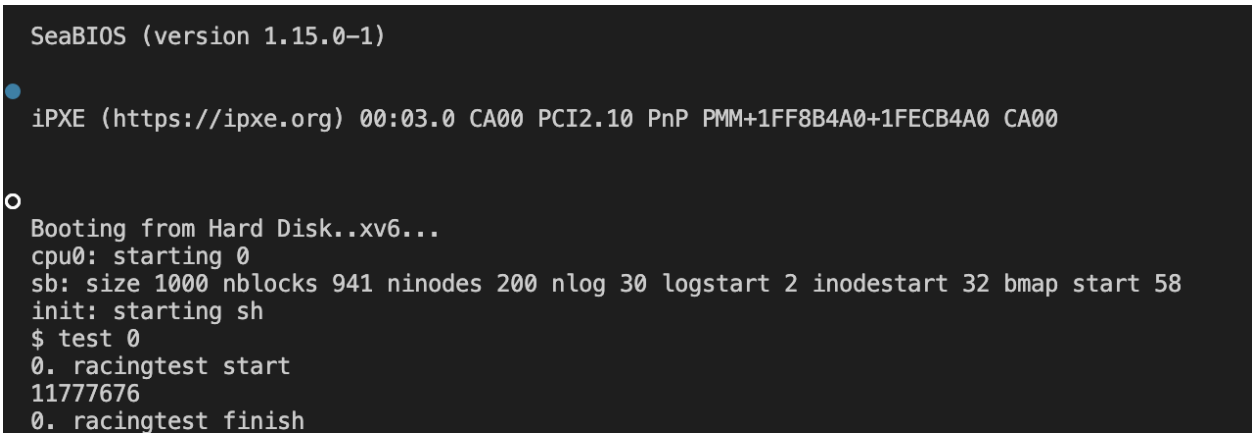
```
#define NUM_THREAD 10
#define NTEST 14

// Show race condition
int racingtest(void);

// Test basic usage of thread_create, thread_join, thread_exit
int basictest(void);
int jointest1(void);
int jointest2(void);

// Test whether a process can reuse the thread stack
int stresstest(void);

// Test what happen when some threads exit while the others are alive
int exittest1(void);
int exittest2(void);

// Test fork system call in multi-threaded environment
int forktest(void);

// Test exec system call in multi-threaded environment
int exectest(void);

// Test what happen when threads requests sbrk concurrently
int sbrktest(void);

// Test what happen when threads kill itself
int killtest(void);

// Test pipe is worked in multi-threaded environment
int pipetest(void);

// Test sleep system call in multi-threaded environment
int sleeptest(void);

volatile int gcnt;
int gpipe[2];

int (*testfunc[NTEST])(void) = {
  racingtest,
  basictest,
  jointest1,
  jointest2,
  stresstest,
  exittest1,
  exittest2,
  forktest,
  exectest,
  sbrktest,
  killtest,
  pipetest,
  sleeptest,
};
char *testname[NTEST] = {
  "racingtest",
  "basictest",
  "jointest1",
  "jointest2",
  "stresstest",
  "exittest1",
  "exittest2",
  "forktest",
  "exectest",
  "sbrktest",
  "killtest",
  "pipetest",
  "sleeptest",
};

int
main(int argc, char *argv[])
{
  int i;
  int ret;
  int pid;
  int start = 0;
  int end = NTEST-1;
```

```
  if (argc >= 2)
    start = atoi(argv[1]);
  if (argc >= 3)
    end = atoi(argv[2]);

  for (i = start; i <= end; i++){
    printf(1,"%d. %s start\n", i, testname[i]);
    if (pipe(gpipe) < 0){
      printf(1,"pipe panic\n");
      exit();
    }
    ret = 0;

    if ((pid = fork()) < 0){
      printf(1,"fork panic\n");
      exit();
    }
    if (pid == 0){
      close(gpipe[0]);
      ret = testfunc[i]();
      write(gpipe[1], (char*)&ret, sizeof(ret));
      close(gpipe[1]);
      exit();
    } else{
      close(gpipe[1]);
      if (wait() == -1 || read(gpipe[0], (char*)&ret, sizeof(ret)) == -1 || ret != 0){
        printf(1,"%d. %s panic\n", i, testname[i]);
        exit();
      }
      close(gpipe[0]);
    }
    printf(1,"%d. %s finish\n", i, testname[i]);
    sleep(100);
  }
  exit();
}
```

a. racingtest

```


void*
racingthreadmain(void *arg)
{
  int tid = (int) arg;
  int i;
  int tmp;
  for (i = 0; i < 10000000; i++){
    tmp = gcnt;
    tmp++;
  asm volatile("call %P0"::"i"(nop));
    gcnt = tmp;
  }
  thread_exit((void *)(tid+1));

  return 0;
}

int
racingtest(void)
{
  thread_t threads[NUM_THREAD];
  int i;
  void *retval;
  gcnt = 0;

  for (i = 0; i < NUM_THREAD; i++){
    if (thread_create(&threads[i], racingthreadmain, (void*)i) != 0){
      printf(1, "panic at thread_create\n");
      return -1;
    }
  }
  for (i = 0; i < NUM_THREAD; i++){
    if (thread_join(threads[i], &retval) != 0 || (int)retval != i+1){
      printf(1, "panic at thread_join\n");
      return -1;
```

```
    }
  }
  printf(1,"%d\n", gcnt);
  return 0;
}
```



racingtest

- 10개의 thread가 동시에 data영역에 속한 process(main thread)에 존재하는 변수에 접근할 때를 가정한 test case이다.
- 기댓값은, **thread들이 해당 data영역을 공유하고 있기에 여러 thread가 동시에 접근한다면 data race문제로 정상적인 sum 값이 나오지 않을 것이라 예상했다.**
- 실제로는, **1억번의 increase를 시행하지만 턱없이 적은 숫자가 출력**됨을 확인 할 수 있다.

b. basictest

```
void*
basicthreadmain(void *arg)
{
  int tid = (int) arg;
  int i;
  for (i = 0; i < 100000000; i++){
    if (i % 20000000 == 0){
      printf(1, "%d", tid);
    }
  }
  thread_exit((void *)(tid+1));

  return 0;
}

int
basictest(void)
{
  thread_t threads[NUM_THREAD];
  int i;
  void *retval;

  for (i = 0; i < NUM_THREAD; i++){
    if (thread_create(&threads[i], basicthreadmain, (void*)i) != 0){
      printf(1, "panic at thread_create\n");
      return -1;
    }
  }
  for (i = 0; i < NUM_THREAD; i++){
    if (thread_join(threads[i], &retval) != 0 || (int)retval != i+1){
      printf(1, "panic at thread_join\n");
      return -1;
    }
```

```
  }
  printf(1,"\n");
  return 0;
}
```

```
1. basictest start
012345678912346705898457102369758469130281467935O2
1. basictest finish
```

basictest

- basic test는 thread_create, thread_joint, thread_exit가 정상적으로 수행되는 지, return value는 정확하게 전달 되는지 확인하는 test case이다.

c. jointest1

```
void*
jointhreadmain(void *arg)
{
  int val = (int)arg;
  sleep(200);
  printf(1, "thread_exit...\n");
  thread_exit((void *)(val*2));

  return 0;
}

int
jointest1(void)
{
  thread_t threads[NUM_THREAD];
  int i;
  void *retval;

  for (i = 1; i <= NUM_THREAD; i++){
    if (thread_create(&threads[i-1], jointhreadmain, (void*)i) != 0){
      printf(1, "panic at thread_create\n");
      return -1;
    }
  }
  printf(1, "thread_join!!!\n");
  for (i = 1; i <= NUM_THREAD; i++){
    if (thread_join(threads[i-1], &retval) != 0 || (int)retval != i * 2 ){
      printf(1, "panic at thread_join\n");
      return -1;
    }
  }
  printf(1,"\n");
  return 0;
}

int
jointest2(void)
{
  thread_t threads[NUM_THREAD];
  int i;
  void *retval;

  for (i = 1; i <= NUM_THREAD; i++){
    if (thread_create(&threads[i-1], jointhreadmain, (void*)(i)) != 0){
      printf(1, "panic at thread_create\n");
      return -1;
    }
  }
  sleep(500);
  printf(1, "thread_join!!!\n");
  for (i = 1; i <= NUM_THREAD; i++){
    if (thread_join(threads[i-1], &retval) != 0 || (int)retval != i * 2 ){
      printf(1, "panic at thread_join\n");
```

```
      return -1;
    }
  }
  printf(1,"\n");
  return 0;
}
```

```
2. jointest1 start
thread_join!!!
thread_ethread_exit...thread_exit...
thread_exit...
thread_exit...
threthread_exit...
threathread_exit...ad_exit...
d
thxit...
_exit...
read_e
xit...

2. jointest1 finish
```

jointest1

- jointest1은 thread의 **start_routine에서 thread_exit전에 sleep를 수행하여 thread_join이 먼저 수행 된 경우** 정상적으로 동작하는 지 확인하는 test case이다.

d. jointest2

```
3. jointest2 start
thread_thread_exthread_exiet...
xit.it...
thread_exit...
thread_exit...
thread_exit...
threathread_exit...
thread_exthread_exit...
it.....
d_
exit...
thread_join!!!

3. jointest2 finish
```

jointest2

- jointest2는 jointest1과 반대로 **main thread에서 thread_join이 수행되는 시간을 늦추는 test case이다.**

e. stresstest

```
void*
stressthreadmain(void *arg)
{
  thread_exit(0);

  return 0;
}

int
stresstest(void)
{
  const int nstress = 35000;
  thread_t threads[NUM_THREAD];
  int i, n;
  void *retval;

  for (n = 1; n <= nstress; n++){
```

```
    if (n % 1000 == 0)
      printf(1, "%d\n", n);
    for (i = 0; i < NUM_THREAD; i++){
      if (thread_create(&threads[i], stressthreadmain, (void*)i) != 0){
        printf(1, "panic at thread_create\n");
        return -1;
      }
    }
    for (i = 0; i < NUM_THREAD; i++){
      if (thread_join(threads[i], &retval) != 0){
        printf(1, "panic at thread_join\n");
        return -1;
      }
    }
  }
  printf(1, "\n");
  return 0;
}
```

```
4. stresstest start
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
14000
15000
16000
17000
18000
19000
20000
21000
22000
23000
24000
25000
26000
27000
28000
29000
30000
31000
32000
33000
34000
35000

4. stresstest finish
```

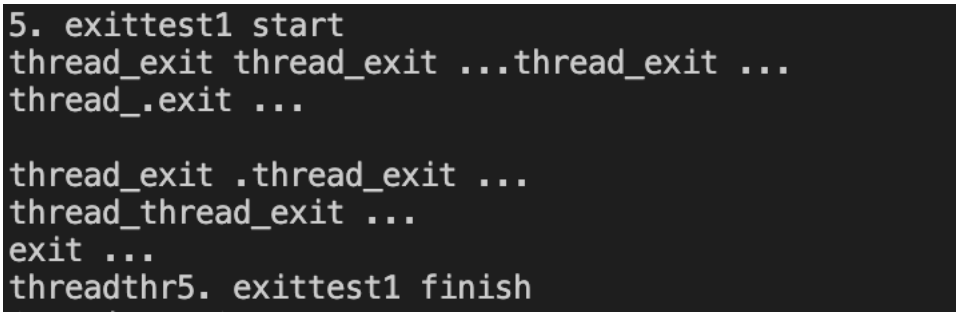
stresstest

- 10개의 thread를 35000번 생성하는 test case이다.
- 처음에 해당 test case를 실행 했을 때는, **정리한 thread의 stack을 재사용하지 않아 에러가 발생하였는데 stack을 재사용 한 후 정상적으로 각 thread가 할당되는 것을 확인하였다.**

e. exittest1

```
void*
exitthreadmain(void *arg)
{
```

```
  int i;
  if ((int)arg == 1){
    while(1){
      printf(1, "thread_exit ...\n");
      for (i = 0; i < 5000000; i++);
    }
  } else if ((int)arg == 2){
    exit();
  }
  thread_exit(0);

  return 0;
}

int
exittest1(void)
{
  thread_t threads[NUM_THREAD];
  int i;

  for (i = 0; i < NUM_THREAD; i++){
    if (thread_create(&threads[i], exitthreadmain, (void*)1) != 0){
      printf(1, "panic at thread_create\n");
      return -1;
    }
  }
  sleep(1);
  return 0;
}

int
exittest2(void)
{
  thread_t threads[NUM_THREAD];
  int i;

  for (i = 0; i < NUM_THREAD; i++){
    if (thread_create(&threads[i], exitthreadmain, (void*)2) != 0){
      printf(1, "panic at thread_create\n");
      return -1;
    }
  }
  while(1);
  return 0;
}
```



exittest1

- process(main thread)가 종료 된다면 해당 프로세스의 모든 thread들이 종료되는지 확인하는 test case이다.
- process가 exit을 호출한다면 기존 process의 모든 thread들이 정리되야함이 자명하므로, 정상적으로 수행이 되었다.

f. exittest2

```
6. exittest2 start
6. exittest2 finish
```

exittest2

- exittest2는 thread에서 exit를 호출 했을 때 모든 thread들이 종료 되는지 확인하는 test case이다.
- 처음에는 thread가 exit을 호출 하면, thread_exit를 호출하며 해당 thread만 종료되게 구현했으나, 해당 구현의 문제점은
  - **exit와 thread_exit의 차별점이 없는 점.**
  - **process를 여러개로 나누어 병렬 처리를 위해서 만든 것이 thread의 정의라는 점**
- 해당 두 가지 점을 고려하여 **해당 process와 그에 해당하는 thread 모두를 정리하게 구현하였다.**

g. forktest

```
void*
forkthreadmain(void *arg)
{
  int pid;
  if ((pid = fork()) == -1){
    printf(1, "panic at fork in forktest\n");
    exit();
  } else if (pid == 0){
    printf(1, "child\n");
    exit();
  } else{
    printf(1, "parent\n");
    if (wait() == -1){
      printf(1, "panic at wait in forktest\n");
      exit();
    }
  }
  thread_exit(0);

  return 0;
}

int
forktest(void)
{
  thread_t threads[NUM_THREAD];
  int i;
  void *retval;

  for (i = 0; i < NUM_THREAD; i++){
    if (thread_create(&threads[i], forkthreadmain, (void*)0) != 0){
      printf(1, "panic at thread_create\n");
      return -1;
    }
  }
  for (i = 0; i < NUM_THREAD; i++){
    if (thread_join(threads[i], &retval) != 0){
      printf(1, "panic at thread_join\n");
      return -1;
    }
  }
  return 0;
}
```

```
7. forktest start
parent
parent
pareparent
parent
parent
parent
parent
parent
child
child
chilchild
child
child
child
child
child
child
nt
parent
d
7. forktest finish
```

forktest

- forktest는 thread에서 fork를 수행했을 때, 정상적으로 수행 되는지 확인하기 위한 test case이다.
- **thread에서 fork를 통해서 만들어진 child process는 child를 출력하고, 부모가 되는 thread는 parent를 출력한다.**
- 정상적으로 출력 되며 thread가 wait를 통해 기다린 후 자식을 정리 할 수 있음을 확인 하였다.

h. exectest

```
void*
execthreadmain(void *arg)
{
  char *args[3] = {"echo", "echo is executed!", 0};
  sleep(1);
  exec("echo", args);

  printf(1, "panic at execthreadmain\n");
  exit();
}

int
exectest(void)
{
  thread_t threads[NUM_THREAD];
  int i;
  void *retval;

  for (i = 0; i < NUM_THREAD; i++){
    if (thread_create(&threads[i], execthreadmain, (void*)0) != 0){
      printf(1, "panic at thread_create\n");
      return -1;
    }
  }
  for (i = 0; i < NUM_THREAD; i++){
    if (thread_join(threads[i], &retval) != 0){
      printf(1, "panic at thread_join\n");
      return -1;
    }
  }
  printf(1, "panic at exectest\n");
  return 0;
}
```

exectest

- exectest는 thread에서 exec을 실행하고, 기존 프로세스의 모든 thread들이 정리 될 수 있는지 확인하는 testcase이다.
- 예상 기댓값은 처음에 **exec을 수행시킨 thread가 main thread를 포함한 모든 thread들을 정리 한 후, exec을 수행하는 것이다.**
- 예상대로 **echo를 하나의 thread가 한번 exec을 수행하고, 다른 모든 thread들이 종료됨을 확인 할 수 있다.**

i. sbrktest

```
void*
sbrkthreadmain(void *arg)
{
  int tid = (int)arg;
  char *oldbrk;
  char *end;
  char *c;
  oldbrk = sbrk(1000);
  end = oldbrk + 1000;
  for (c = oldbrk; c < end; c++){
    *c = tid+1;
  }
  sleep(1);
  for (c = oldbrk; c < end; c++){
    if (*c != tid+1){
      printf(1, "panic at sbrkthreadmain\n");
      exit();
    }
  }
  thread_exit(0);

  return 0;
}

int
sbrktest(void)
{
  thread_t threads[NUM_THREAD];
  int i;
  void *retval;

  for (i = 0; i < NUM_THREAD; i++){
    if (thread_create(&threads[i], sbrkthreadmain, (void*)i) != 0){
      printf(1, "panic at thread_create\n");
      return -1;
    }
  }
  for (i = 0; i < NUM_THREAD; i++){
    if (thread_join(threads[i], &retval) != 0){
      printf(1, "panic at thread_join\n");
      return -1;
    }
  }

  return 0;
}
```

sbrktest

- sbrktest는 **10개의 thread가 sbrk를 호출하더라도 정상적으로 할당 공간이 겹치지 않고 수행 할 수 있는지 확인하는 test case이다.**

j. killtest

```
void*
killthreadmain(void *arg)
{
  kill(getpid());
  while(1);
}

int
killtest(void)
{
  thread_t threads[NUM_THREAD];
  int i;
  void *retval;

  for (i = 0; i < NUM_THREAD; i++){
    if (thread_create(&threads[i], killthreadmain, (void*)i) != 0){
      printf(1, "panic at thread_create\n");
      return -1;
    }
  }
  for (i = 0; i < NUM_THREAD; i++){
    if (thread_join(threads[i], &retval) != 0){
      printf(1, "panic at thread_join\n");
      return -1;
    }
  }
  while(1);
  return 0;
}
```

killtest

- killtest는 thread가 kill 상태가 된다면, 프로세스 내의 모든 thread가 종료 될 수 있는지 확인하는 test case이다.
- **thread가 kill 될 때, main thread까지 같이 정리하지 못한다면 main thread가 종료되지 못하는데, 정상적으로 main thread까지 종료됨을 확인 할 수 있다.**

k. pipetest

```
void*
pipethreadmain(void *arg)
{
  int type = ((int*)arg)[0];
  int *fd = (int*)arg+1;
  int i;
  int input;
  for (i = -5; i <= 5; i++){
    if (type){
      read(fd[0], &input, sizeof(int));
      __sync_fetch_and_add(&gcnt, input);
      //gcnt += input;
    } else{
      write(fd[1], &i, sizeof(int));
    }
  }
  thread_exit(0);

  return 0;
```

```
}

int
pipetest(void)
{
  thread_t threads[NUM_THREAD];
  int arg[3];
  int fd[2];
  int i;
  void *retval;
  int pid;

  if (pipe(fd) < 0){
    printf(1, "panic at pipe in pipetest\n");
    return -1;
  }
  arg[1] = fd[0];
  arg[2] = fd[1];
  if ((pid = fork()) < 0){
      printf(1, "panic at fork in pipetest\n");
      return -1;
  } else if (pid == 0){
    close(fd[0]);
    arg[0] = 0;
    for (i = 0; i < NUM_THREAD; i++){
      if (thread_create(&threads[i], pipethreadmain, (void*)arg) != 0){
        printf(1, "panic at thread_create\n");
        return -1;
      }
    }
    for (i = 0; i < NUM_THREAD; i++){
      if (thread_join(threads[i], &retval) != 0){
        printf(1, "panic at thread_join\n");
        return -1;
      }
    }
    close(fd[1]);
    exit();
  } else{
    close(fd[1]);
    arg[0] = 1;
    gcnt = 0;
    for (i = 0; i < NUM_THREAD; i++){
      if (thread_create(&threads[i], pipethreadmain, (void*)arg) != 0){
        printf(1, "panic at thread_create\n");
        return -1;
      }
    }
    for (i = 0; i < NUM_THREAD; i++){
      if (thread_join(threads[i], &retval) != 0){
        printf(1, "panic at thread_join\n");
        return -1;
      }
    }
    close(fd[0]);
  }
  if (wait() == -1){
    printf(1, "panic at wait in pipetest\n");
    return -1;
  }
  if (gcnt != 0)
    printf(1,"panic at validation in pipetest : %d\n", gcnt);

  return 0;
}
```

pipetest

- pipetest는 **thread 환경에서도 각각 화면에 출력하는 데에 문제가 없는지 확인하는 test case이다.**

l. sleeptest

```
void*
sleepthreadmain(void *arg)
{
  sleep(1000000);
  thread_exit(0);

  return 0;
}

int
sleeptest(void)
{
  thread_t threads[NUM_THREAD];
  int i;

  for (i = 0; i < NUM_THREAD; i++){
    if (thread_create(&threads[i], sleepthreadmain, (void*)i) != 0){
        printf(1, "panic at thread_create\n");
        return -1;
    }
  }
  sleep(10);
  return 0;
}
```

sleeptest

- sleeptest는 한 thread가 sleep를 호출하면 그 thread만 sleep할 수 있는지 확인하는 test case이다.
- **main thread가 다른 thread들보다 먼저 종료되며 나머지 thread들을 모두 정리하게 된다.**

m. thread makes thread again

```
#include "types.h"
#include "stat.h"
#include "user.h"

#define NUM_THREAD 5
thread_t thread[NUM_THREAD];
thread_t thread2[NUM_THREAD];
void* thread_main2(void *arg)
{
    int val1 = (int)arg;
    printf(1, "Thread %d start", val1);
    printf(1, " Root\n");
    thread_exit(0);
}
void* thread_main(void* arg)
{
    int j;
    int val = (int)arg;
    int *retval2;
    printf(1, "Thread %d start\n", val);
    for (j = 0; j < NUM_THREAD; j++) {
        thread_create(&thread2[j], thread_main2, (void*)j);
    }
    sleep(100);
    thread_exit(0);
}

int main(int argc, char* argv[])
{
    int i;
```

```
    void *retval;
    printf(1, "Thread of thread test start\n");
    for (i = 0; i < NUM_THREAD; i++) {
        thread_create(&thread[i], thread_main, (void*)i);
    }
    for( i = 0 ; i<NUM_THREAD; i++){
        thread_join(thread[i], &retval);
    }
    exit();
}
```

- thread가 추가로 thread_create를 수행하는 경우에 대해서 test case를 작성하였다.
- **thread에서 thread를 만들어도 사실상 original thread의 main thread와 같은 memory 공간을 공유 하므로 해당 thread도 main thread의 thread라고 봐도 무방하다고 생각하였다.**
- 그래서, 이에 대해서 정상적으로 작동하는지 확인하기 위한 test case를 작성하였다.

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 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
$ pmanager
-> execute testThreadkill 10
-> Thread of thread test start
Thread 0Thread 2 stThread 3 start
Thread 4 start
Thread 1 start RootThread 2 start Thread 3 startThread 4 start Root
Thread 1 starThread 2 start Roo start
Tart
Thre
Root
 Root
t Root
t
Thread 4 starhread 1ad 0 startThread 3 stat R Root
rt Roooot
Thread 0t
TThread 1 start Thread 2 start Root
TThread 4 start Root
ThreaThread 1 startThread 2 start Root
hrd 0 start R Root
Thread 3 sThread 4 start start
tart R oot
Root
 starThread 0 start Thread 1 start RoThread 2 sThread 3 start Thread 4 start Root Root
hreadt
 0 start RRoot
oot
ead 3 stoot
Root
art Root
ot
tart Root
Root
zombie!
list
1.process name : init     2.pid : 1     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
1.process name : sh     2.pid : 2     3.number of stack pages : 1      4.stack size : 16384      5.memory limit : 0
1.process name : pmanager     2.pid : 3     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
1.process name : pmanager     2.pid : 6     3.number of stack pages : 1      4.stack size : 12288      5.memory limit : 0
```

retult values of thread makes thread again

- 각 thread가 정상적으로 생성되었고, 각 thread가 또한 새로운 thread를 정상적으로 생성하였다.
- 이때, **thread가 만든 thread를 정리하는 주체는 main thread이기에 main thread가 만든 thread와 thread가 만든 thread모두 정상적으로 main thread에서 정리하는 것을 확인 할 수 있다.**

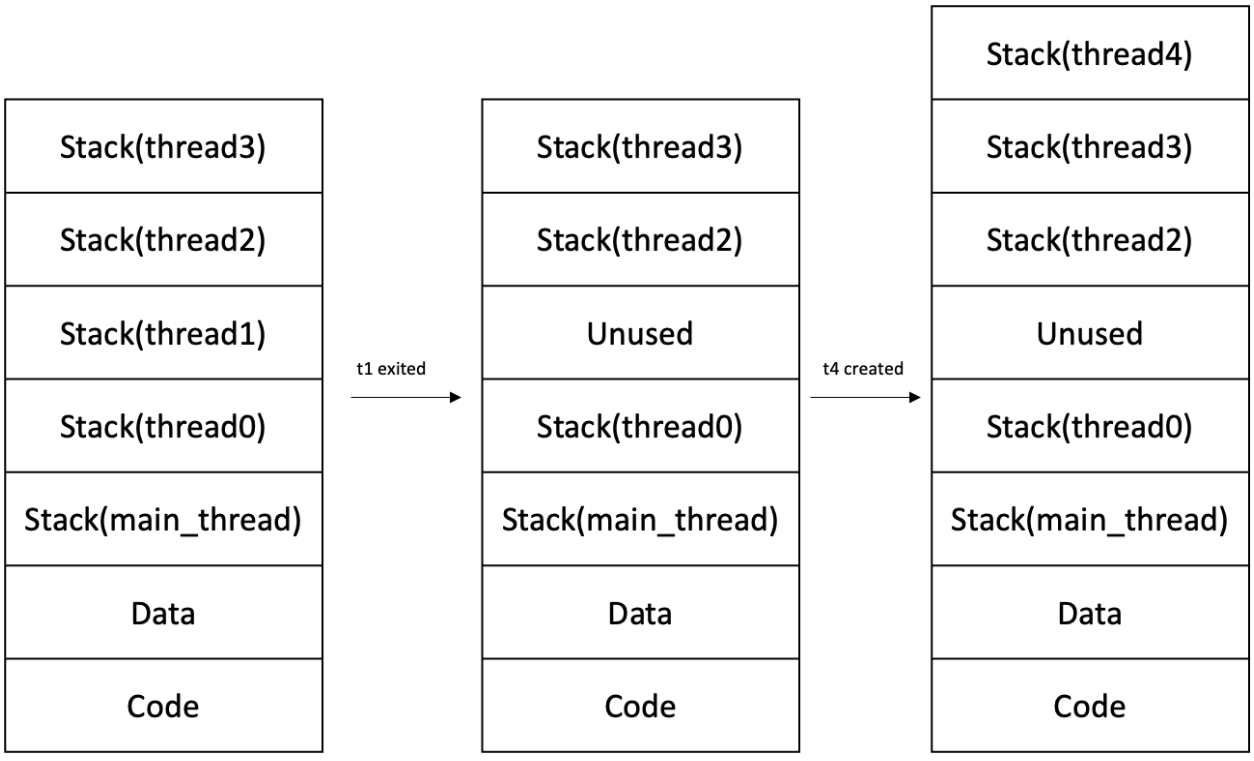
4. Trouble Shooting

1) Sbrk

- thread_create를 반복하면서 각 thread가 sbrk 시스템 콜을 수행 했을때, page fault가 발생하였다.
- 각 thread들의 sz값이 동기화가 되지 않아 발생한 문제라고 유추를 하였다.
- **한 개의 thread가 stack size를 늘려도, 해당 sz값은 main thread에만 반영을 시켜 주었기에, 다른 thread가 sbrk를 수행하면 stack size가 늘어남을 인지하지 못했다.**
- 그래서 해당 코드를 추가하여 모든 thread들의 sz값을 동기화 시켰다.

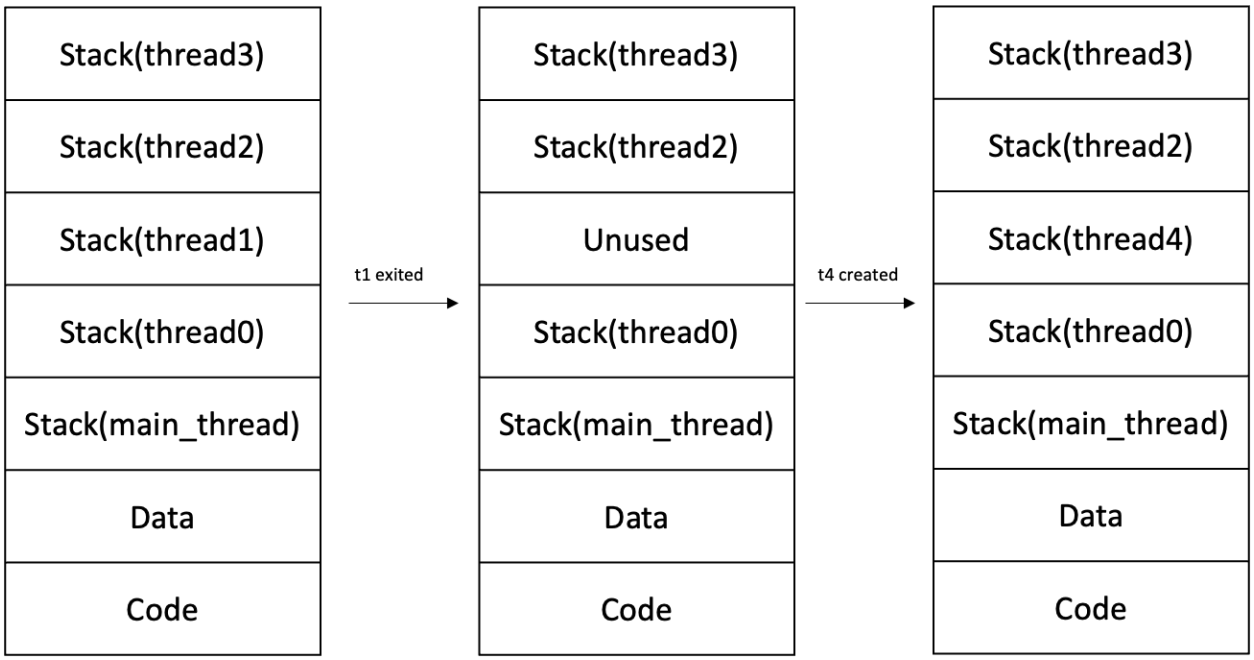
```
//process들의 모든 thread들의 stack size 동기화
  struct proc* p;

  for(p=ptable.proc;p<&ptable.proc[NPROC];p++){
    if(p->pid==np->pid){
      p->sz=np->sz;
    }
  }
```

2) Stack 재사용



기존의 thread 생성 방식

- 처음 thread_create를 구현했을 때에, **thread가 만들어 질 때 마다 단지 stack을 쌓아 올리기만 하였다.**
- 하지만 해당 구현의 문제점은 **빈 stack공간이 존재함에도 계속해서 stack공간을 할당하기 때문에 메모리가 상당히 낭비되는 점이다.**
- 실제로, LWP의 test에서 **실제 물리메모리를 초과할때까지 stack을 할당 받아 allocuvm이 더이상 불가능해졌다**



변경한 thread 생성 방식

- 이를 해결하기 위해 위에 그림과 같이 **빈 stack공간이 있으면, 새로 thread를 만들 때 해당 공간을 재사용 하도록 만들어 주었다.**
- 구현 방법은 단순히, **process(main thread)에서 배열을 통해, 빈 공간이 있는지 찾아주어 해당 부분에 thread를 할당해 주었다.**

```
//stack에 빈공간이 있는지 확인
  for(i = 0; i < NTHREAD; i++)
    if(curproc->stack[i] != -1)
      break;

//빈공간이 있다면, 그 자리에 할당
  if(i != NTHREAD) {
   sp = curproc->stack[i];
   np->ustack = sp;
   curproc->stack[i] = -1;
  }
  //없다면, 새로 공간을 할당
  else {
    sz = PGROUNDUP(sz);

    //제한 사항이 없다면 allocuvm 수행
    if( (sz = allocuvm(curproc->pgdir, sz, sz + 2*PGSIZE)) == 0) {
      kfree(np->kstack);
      np->kstack=0;
      np->parent = 0;
      np->name[0] = 0;
      np->killed = 0;
      np->is_thread = 0;
      np->mem_limit = 0;
      np->pages = 0;
      np->retval = 0;
      np->pid = 0;
      np->tid = 0;
      np->state = UNUSED;
      for(i = 0; i < NTHREAD; i++)
        np->stack[i] = -1;
      release(&ptable.lock);
      return -1;
    }
    clearpteu(curproc->pgdir, (char*)(sz - 2*PGSIZE));

    sp = sz;
    curproc->sz = sz;
```

```
    np->ustack = sz;
  }
```

- thread가 생성 될 때, main thread(process)의 stack 변수를 순회하며 빈공간이 있는지 확인한다. **(빈공간이라 함은 stack의 주소값이 저장되어 있는 인덱스이다.)**
- **빈공간이 있다면 해당 자리에 thread를 넣어주고 그렇지 않다면 allocuvm을 통해 stack을 추가로 할당해준다.**

```
curproc->stack[i] = p->ustack;
```

- join을 통해 thread가 정리 될 때, **해당 thread가 사용하던 stack의 주소값을 main thread에 넘겨주게 하였다.**

3) exit(1)

```
//close all open files
  for(p = curproc->next_thread ; p != curproc; p = p->next_thread){
      if(p->state != ZOMBIE) {
        for(fd = 0; fd < NOFILE; fd++) {
          if(p->ofile[fd]) {
            fileclose(p->ofile[fd]);
            p->ofile[fd] = 0;
          }
        }

        begin_op();
        iput(p->cwd);
        end_op();
        p->cwd = 0;
      }
  }
```

- exit를 수행하게 된다면, 해당 thread 혹은 process에 속하는 모든 thread를 정리해 주어야한다.
- 이때, **지속해서 page fault가 발생하는 문제가 생겼는데 모든 thread의 file을 close해주어야 하는데 main thread의 ofile만 close해서 발생하는 문제였다.**
- 위의 코드를 통해 process안에 속한 모든 thread들의 file을 close해주어서 해결하였다.

4) exit(2)

- process(main thread)가 아닌 thread가 exit를 수행 할때, 어떤 방식으로 구현할 것인지가 문제였다.
  - exit를 수행한 thread만을 종료한다. → thread_exit과 동일.
  - **해당 thread가 속한 모든 thread들과 같이 종료한다. → 기존 (main thread)exit과 동일.**
- 따라서, 2번 방식을 채택하였고, **main thread를 포함한 모든 thread들을 종료하게 구현하였다.**

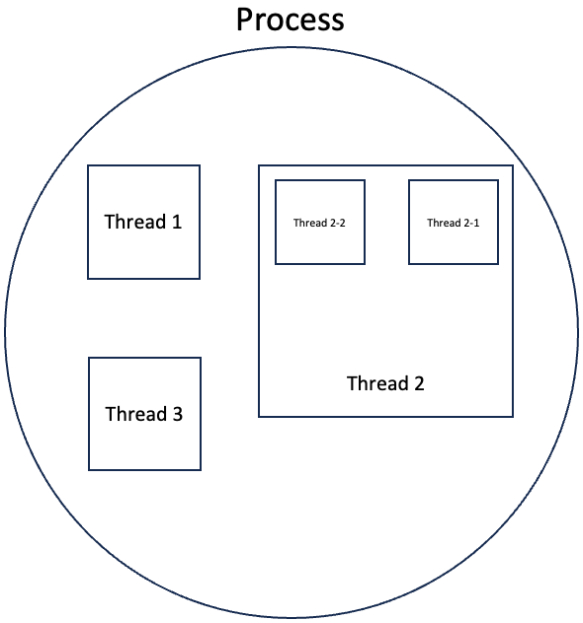
5) exec

- thread가 exec을 수행할 때, 어떤 방식으로 구현할 것인가에 대한 문제가 있었다.
- 처음엔 main thread를 제외한 thread만을 정리 한 후, exec을 수행한 thread가 새로 process가 되게 구현하였다.
- 하지만, exec의 정의는 exec을 수행하는 프로세스가 인자로 들어온 **path에 있는 user program을 수행하는 프로세스가 되는 것이다.**
- 이에 따라 **현재 process의 모든 thread들이 정리된 후, 한 개의 thread에서 새로운 process를 수행하는 것이 맞다고 생각이 들었다.**
- 결론적으로, **main thread를 포함한 모든 thread들을 정리 한 후, 한 개의 thread에서 exec을 수행하였다.**

6) 속도의 문제

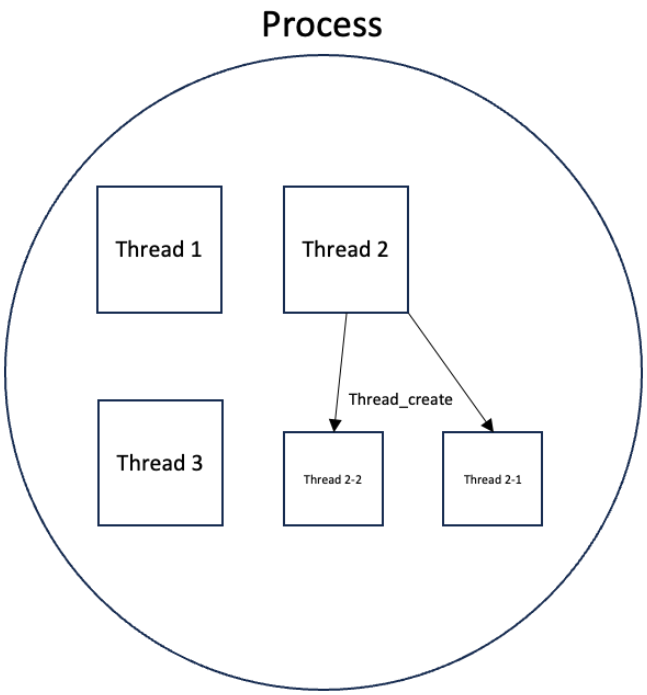
- 초기에는 **thread를 관리 하는 것을 ptable을 전부 순회하여 pid가 같은 것을 찾는 방식으로 구현하였다.**
- 스케줄링 방식이 round-robin이라 스케줄링이 느림을 감안하더라도 예상보다 상당히 진행 속도가 느려, **연결 리스트 구조로 thread들을 연결하여 thread들을 관리해 주었다.**
- 해당 구조를 통해,매 경우 ptable을 순회하지 않아 구조적으로 조금 더 안정적이게 되었다.

7) (not main)thread가 thread를 생성할 때



thread makes thread again

- 위의 그림은 main thread가 아닌 thread가 또 thread를 생성 했을 때를 처음 구상한 그림이다.
- thread안에서 thread가 만들어 지는 구조로, **thread 2에서 생성된 thread들의 main thread는 thread 2가 되는 구조이다.**
- 하지만 사실상 main thread와 각 thread는 메모리 영역을 공유하고 있다.
- **각 thread가 새로 thread를 생성한다고 하더라도, 새로 생성된 thread는 main thread의 main thread 즉 process자체의 main thread와 메모리 영역을 공유한다.**
- 그래서, thread가 새로 thread를 만든다 하더라도 기존 main thread를 참조하는 thread가 생성되게 하였다.


Process
Thread 1
Thread 2
Thread_create
Thread 3
Thread 2-2
Thread 2-1

변경된 구조