# ELE3021_project02_12299_2019030991

> *본 위키는 commit 기준 pmanager finish - first all complete에 맞추어 작성되어 있습니다.*
> *12299_2019030991_홍정범*

## Design

### LWP

> ***thread의 실체 만들기***

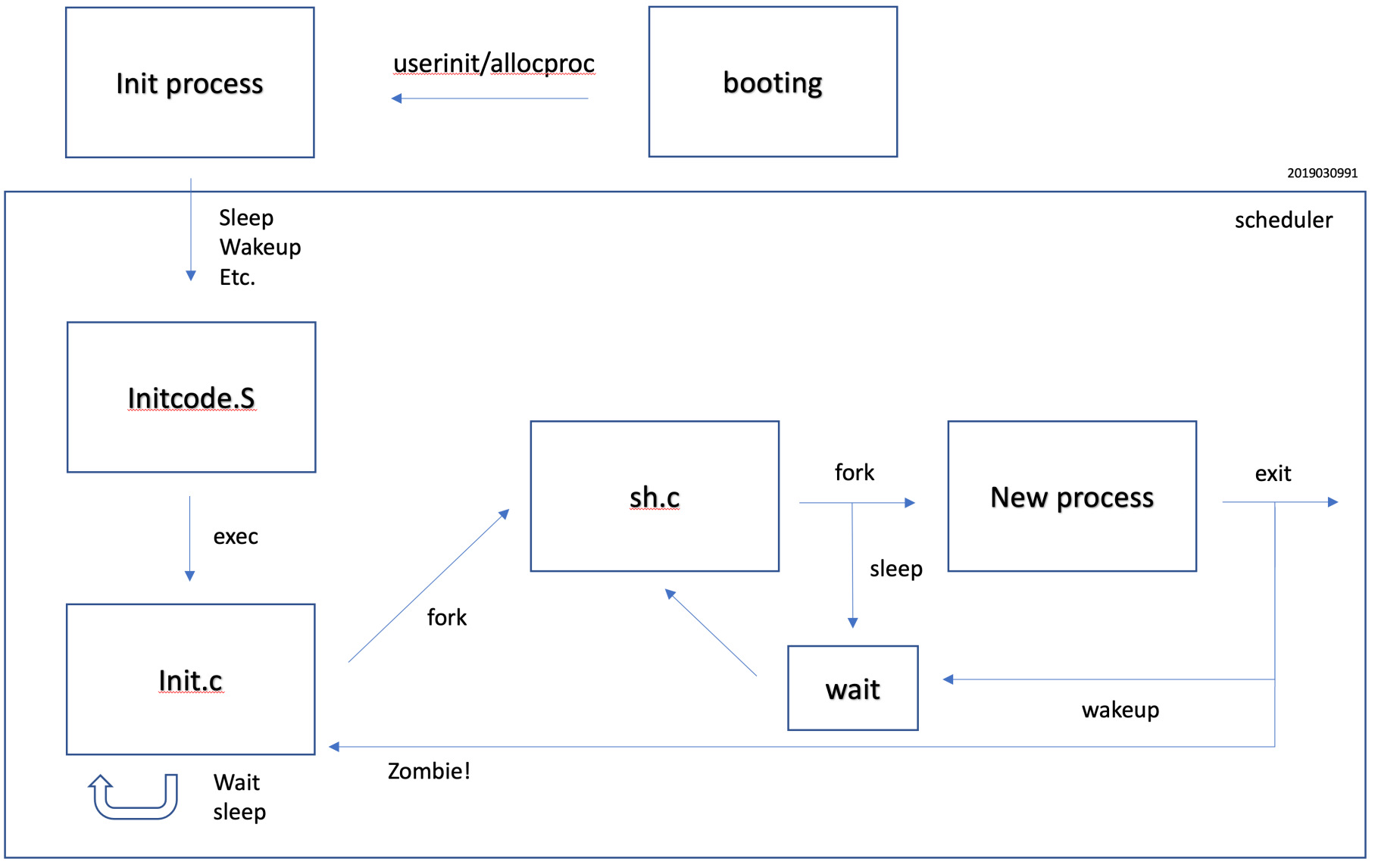
가장 처음 고민했던 것은 “기존에 있는 proc 구조체의 활용” 이었다. 우선 기본적으로 xv6 자체가 proc 구조체로 구성되어있는 ptable을 기준으로 실행되게 되어있기 때문에 proc구조체를 기본적인 실행단위로 고정해야겠다 라고 생각했다. 하지만 쓰레드의 스택을 기반으로 일을 해야하기 때문에, proc이라는 빈 통에 쓰레드를 넣어 일을 시켜야겠다고 생각했다. 즉, 나의 아이디어는 한 프로세스안에 실행되고 있는 쓰레드들을 번갈아가며 proc 구조체 안에 갈아끼우자 라는 것이었다.

이를 위해 개념적으로 몇 가지를 정의했는데, 원래 xv6에서 allocproc으로 할당받게 되는 프로세스는 main thread라고 부르기로 했다. 그리고 proc 구조체 안에 배열을 만들어 쓰레드들을 저장하기로 했다. 한 프로세스가 가질 수 있는 최대 쓰레드는 main thread를 포함해서 10개로 제한했고 main thread는 항상 배열의 첫번째 칸에 위치하게 디자인했다. thread와 proc 구조체는 서로 독립적인 state를 가지며 각 thread의 id는 배열에서의 번호, 즉 0 ~ 9 만 가질수 있게 디자인했다. (이는 test code를 운영할 때 예민하게 처리해주어야 한다.)

```
struct thread {

  //thread state
  thread_t tid;                // thread ID (0~9)
  enum procstate state;        // thread state (independent)

  //thread data
  char *kstack;                // Bottom of kernel stack for this process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process

  // sleep data
  void *chan;                  // If non-zero, sleeping on chan

  // talk about it later
  uint start;                  // thread pool idx;
  void *retval;                // Return value
};
```

```
struct proc {

  // shared data
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  int killed;                  // If non-zero, have been killed
  char name[16];               // Process name (debugging)
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory

  //thread table
  struct thread ttable[10];    // thread table (the main thread is in the ttable[0])

  // talk about it later
  uint thread_pool[10];        // thread reallocate address
  uint sz_limit;               // limit of memory

  // copy from thread
  char *kstack;                // Bottom of kernel stack for this process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  int cur_thread;              // current thread ID
};
```

## *single-thread 기반 흐름 구성하기*




xv6 process 실행 대략적인 흐름도

### allocproc()

원래 proc 구조체 안에 있던 kernel stack에 공간을 할당받고 이것저것 초기 데이터를 설정해 주었지만, 나의 디자인에서는 proc 구조체는 쓰레드의 stack을 잠시 담아 일을 시켜줄 뿐이기에, 본래 스택의 주인인 쓰레드에 데이터를 할당해주었다. 처음 프로세스를 만들 때에는 단일 쓰레드의 프로세스로, main thread를 만들어주는 과정이기 때문에 main thread의 위치인 table의 0번칸에 할당해주었다.

### userinit()

allocproc을 통해 main thread에 스택을 할당받은 proc 구조체를 넘겨받게 되는데, init proceess에 대한 데이터 역시 main thread에 trapframe에 기록을 해준다.

### scheduler()

스케줄러에서는 기본적인 RR 방식을 그대로 살리기 위하여 가장 마지막에 실행한 thread를 proc에 저장하여 table을 순회하게 했다. 그리고 context switching이 일어나기 전에 thread를 proc 구조체로 복사해오고, process가 일을 마치고 다시 스케줄링이 될 때 proc 구조체를 thread로 복사하도록 디자인했다. 이것을 구현하기 위하여 proc → thread 로 자원을 옮겨주는 함수(copy_process)와 thread → proc 로 자원을 옮겨주는 함수(copy_thread)를 만들어 사용했다.

### sleep() / wakeup()

thread별로 sleep을 할 수 있어야 하기 때문에 thread에게 chan 변수를 독자적으로 가질 수 있게 하였고, 한 쓰레드가 잠에 들어도 다른 쓰레드는 일을 할 수 있기 때문에 proc의 state는 runnable, thread의 state는 sleeping이 되게 하였다. wakeup 함수에서는 chan이 thread에 있기 때문에 각 프로세스의 쓰레드 배열을 확인하여 쓰레드의 상태를 runnable로 바꿔주는 과정을 거쳤다.

### exec()

여기선 헛갈리지 말아야할 것이, 이 이후에 다시 스케줄러로 돌아가게 되면 e 과정을 거치면서 proc 구조체에 있는 스택 정보를 thread로 내려주게 된다. 쓰레드에 할당하면 역으로 할당한 효과가 사라지므로 그냥 프로세스에다가 할당해주어 알아서 스케줄러에서 테이블로 내려주게 했다.

### fork()

fork는 다시 헛갈리면 안되는 것이 새로운 프로세스를 만들고 그 프로세스에 자신과 같은 스택을 넣어주는 것이다. 그러므로 새로운 프로세스를 만드는 것과 같이 thread, 즉 table에다가 스택 관련 정보를 업데이트 해주어야한다.

---

### wait()

wait은 exit하는 자식 프로세스의 자원을 회수해야하는데, 이는 쓰레드에 자원이 있기 때문에 쓰레드의 자원을 정리해주게 디자인했다. 물론 thread들이 공유하는 process의 자원들도 정리해 주었다.

---

### kill()

kill은 killed 변수를 proc구조체가 가지고 있어 process가 exit을 호출하게 디자인했다. 지금까지 만들어온 구조와 일맥상통하게 해준 셈이다. 그리고 깨워주는 것은 process가 아니라 thread를 깨워주게 했다.

---

### exit()

exit은 thread_exit과 차별점을 두어 process 자체를 terminate하는 기능을 가지고 있어야 된다고 생각하고 디자인했다. 그래서 변경점은 거의 없으며, wait함수에서 thread와는 독자적인 state를 가지고 있는 process의 state가 zombie면 모든 쓰레드를 정리해주게 디자인했다.

---

## *multi-thread 기반으로 디자인하기*

multi-thread를 구현하기 위한 함수들을 구현하기 이전에 우선적으로 고려해주어야할 기본 함수들이 있다.

---

### exec() / wait()

바로 이 두 함수인데, 이유는 바로 쓰레드의 자원정리 때문이다. 우선 wait의 경우에는 간단했다. zombie가 된 자식 process의 모든 thread의 자원을 회수하기 위해 순환문을 돌아 처리를 하면 되었기 때문이다. exec의 경우 조금 더 복잡했는데, 이 경우에는 하나의 thread를 살려두고 이 쓰레드를 main-thread로 하는 프로세스를 만들어주어야 했기 때문이다. 그래서 첫째로 “exec를 호출한 쓰레드”를 제외한 모든 쓰레드를 정리, 둘째로 exec를 호출한 쓰레드를 main thread의 자리로 옮겨주는 과정이 필요했다. 그리고 여기에서 thread table에서 복사해온 thread의 정보를 항상 제자리에 옮겨놓지는 않는다는 것을 발견했다. 그래서 스케줄러에서 전의 쓰레드의 위치가 아닌 배열의 위치를 골라서 돌아갈 수 있게 코드를 디자인했다.

```
copy_process(p,&(p->ttable[p->cur_thread]));
```

---

우선, 나의 디자인에서는 무조건 thread_create,  thread_join을 main_thread만 호출할 수 있게 해주었다. 쓰레드를 사용하는 이유는 기본적으로 process의 병렬 진행을 추구하기 위함이라고 생각하는데, thread가 thread를 만드는 것이 일반적인 process의 병렬화라고 생각하지 않았다. 그것은 마치 thread를 병렬화 하고 싶은 느낌이라고 고려했고 그래서 main_thread, 즉 원래 process였던 thread만 자신의 병렬화를 할 수 있게 만들어 주었다.

### thread_create

***코드 디자인***

1. thread를 할당받을 공간을 찾는다. ttable을 돌며 찾는다.
→ 예외 0 : 더 이상 할당할 공간이 없을 때
→ 예외 1 : 할당받을 공간을 선택했는데 해당 쓰레드가 메인쓰레드일 때
→ 예외 2 : 메인쓰레드가 아닌 쓰레드가 thread_create을 호출할 때

2. 찾은 공간에 thread를 할당 받는다. allocproc과 결을 같이한다.

3. 할당받은 thread에 실행할 함수정보를 저장한다. exec와 결을 같이한다.

---

### thread_exit

⚙️ ***코드 디자인***

exit을 호출한 thread의 state를 zombie로 바꿔주며 join을 호출했을 수 있는 main thread를 깨워준다. exit 함수와 결을 같이한다.
→ 예외 : 메인쓰레드가 thread_exit을 호출할 때

---

## thread_join

⚙️ ***코드 디자인***

인자로 받은 thread를 sleep 상태로 기다리며 thread 가 zombie가 되며 exit을 호출하면 해당 쓰레드의 자원을 정리해준다. wait 함수와 결을 같이한다.
→ 예외 0 : 메인쓰레드가 아닌 thread_join을 호출할 때

---

## user stack recycle

기본적인 디자인은 이렇고, 이제 추가적으로 고려해준 부분은 thread의 유저 스택 재활용이었다. 쓰레드를 굉장히 많이 만들었다 지웠다를 반복하면 유저 스택이 계속 늘어나기만 하고 전에 사용했던 공간은 사라지지 않고 남아있어 공간 활용성 측면에서 안좋다고 판단되어 재활용을 하기로 결정했다.

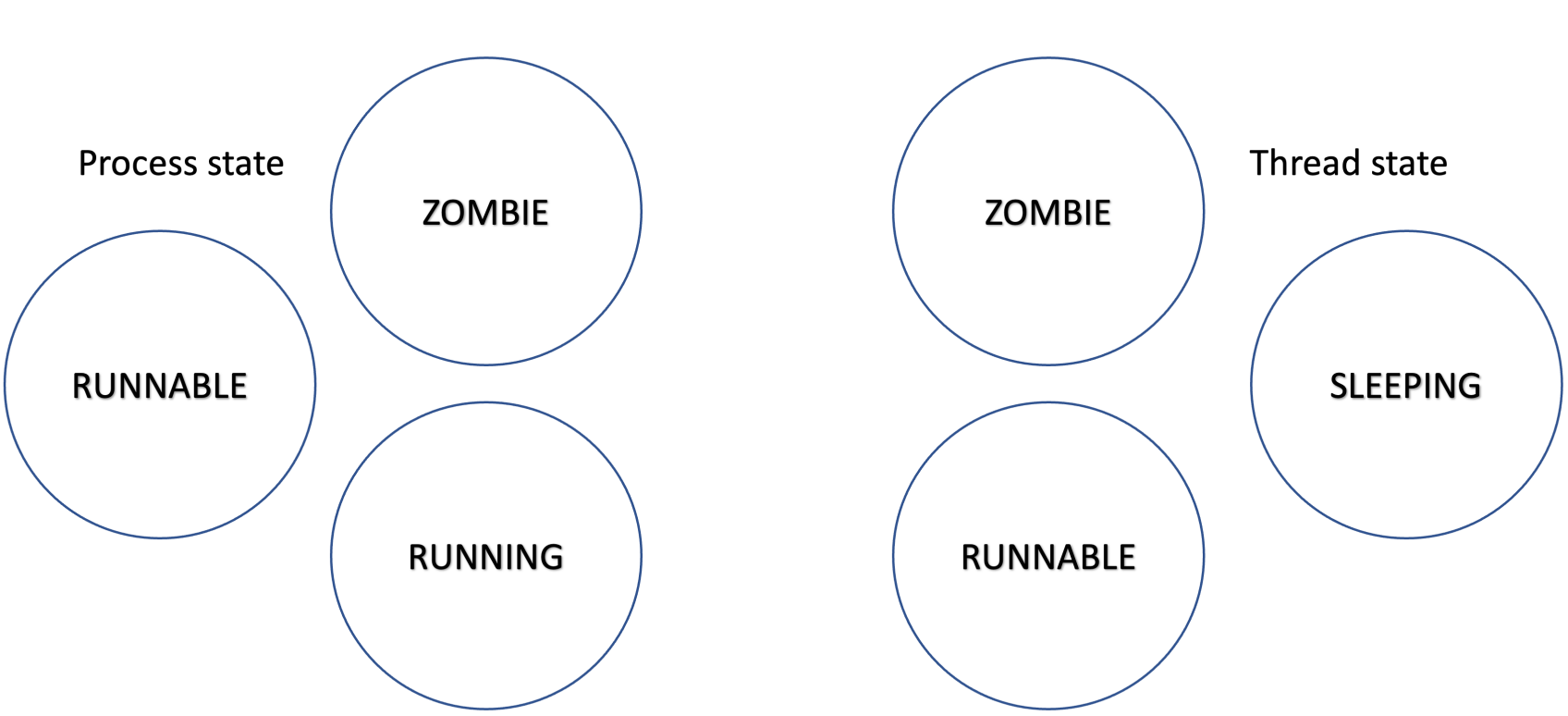
⚙️ 스택 재활용 알고리즘

우선 proc 구조체 안에 thread_pool 이라고 하는 배열을 하나 만든다. 이 배열 안에는 thread의 user stack을 할당 받을 때 시작 주소를 저장한다. 그리고 그것을 저장하기 위하여 thread는 start라는 변수를 하나 갖는다.

1. thread_create할 때 페이지를 할당해 주기 전에 시작 sz를 thread→start에 저장한다.
2. proc 구조체 안에 thread_pool 배열에 sz인자가 저장되어있는지 확인한다.
3-1. 저장되어 있다면 페이지를 할당하지 않고 이 주소에 유저 스택을 할당한다.
3-2 저장되어 있지 않다면 페이지를 할당해준다.
4. thread_join할 때 쓰레드의 자원을 정리해주는 과정에서 thread_pool에 빈칸을 찾아 thread→start를 저장해준다.

---

## process & thread possible state




process & thread state

그래서 디자인 결과, process는 실행중에 runnable, running, zombie, 이 세가지 상태만 가질 수 있다. 모든 쓰레드가 sleeping 중이어도 runnable인 상태를 유지하며, 대신 스케줄이 될 쓰레드가 없으므로 스케줄링이 되지 못한다. 그리고 프로세스가 zombie 상태라는 것은 모든 쓰레드를 종료하고 process 자체를 종료하라는 의미가 된다. thread는 running이 될 수 있는지 나타내는 runnable만 표시하고 실제 running을 하는 주체는 process가 된다. 쓰레드는 다른 쓰레드들과 독립적으로 sleep 상태에 들어갈 수 있어야 하므로 sleeping 상태를 가질 수 있고, 쓰레드가 zombie 상태가 된다는 것은 해당 쓰레드만 자원을 회수 당하고 종료되어야 된다는 것을 의미한다.

---

# Implement

## exec2

```
if(stacksize > 100) {
    cprintf("ERROR : stacksize bigger than 100\n");
    return -1;
  }

  if(stacksize < 1) {
    cprintf("ERROR : stacksize smaller than 1\n");
    return -1;
  }
```

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

exec2에 인자로 받아온 stacksize에 가드용 페이지 하나를 더한 후 할당해 주었다. 그리고 stacksize의 범위가 넘어가면 exec2를 실행할 수 없게 해두었다. 나머지 코드 부분은 기존 exec와 동일하게 구성을 해주었다. system call 함수의 경우에도 동일하게 exec와 동일하게 구성하고 stacksize를 인자로 받는 것만 추가했다.

---

## setmemorylimit

⚙️ 코드 디자인
1. 인자로 받은 pid에 해당하는 process를 찾는다.
2. 과제 명세에 맞추어 예외를 처리해준다.
3. proc 구조체 안에 sz_limit이라는 변수를 만들어 limit를 설정해준다.

---

```
int
setmemorylimit(int pid, int limit)
{
  struct proc *p;
  int pid_chk = 0;

  acquire(&ptable.lock);

  // 1. process 찾기

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->pid == pid) {
      pid_chk = 1;
      break;
    }
  }
  release(&ptable.lock);

  // 2. 예외처리

  if(pid_chk == 0) {
    cprintf("EXCEPTION 0 : non-exit pid error\n");
    return -1;
  }

  if(limit < 0) {
    cprintf("EXCEPTION 1 : negative limit\n");
    return -1;
  }
```

```
  if(p->sz != 0) {
    if(p->sz > limit) {
      cprintf("EXCEPTION 2 : sz already bigger than limit\n");
      return -1;
    }
  }

  // 3. limit 설정하기

  p->sz_limit = limit;

  return 0;
}
```

그리고 설정한 limit를 이용하기 위하여 `growproc()` `exec()` `thread_create()` 등에 allocuvm을 호출하기 전 limit를 확인하고 할당할지 말지를 결정하는 조건문을 달아주었다. 0일때는 무한정 커질 수 있어야 하기에 0 이라면 조건문에 상관없이 일을 할 수 있게 하였다.

```
if(curproc->sz_limit) {
    if(sz+(stacksize+1)*PGSIZE > curproc->sz_limit) {
      cprintf("EXCEPTION : memory limit\n");
      goto bad;
    }
  }
```

---

## pmanager

⚙️ 코드 디자인

1. getcmd 함수를 통해서 명령줄 입력을 받는다.
2. 입력이 무언가 들어오면 parse 함수를 통해서 parsing해 args 배열에 넣어준다
3. 가장 앞에 들어있는 단어를 기준으로 명세 맞추어 일을 하게 한다.

---

```
int
getcmd(char *buf, int nbuf)
{
  printf(2, "(PMG) : ");
  memset(buf, 0, nbuf);
  gets(buf, nbuf);
  if(buf[0] == 0) // EOF
    return -1;
  return 0;
}
```

```
int parse(char *buf, char **args)
{
  int argc = 0;
  while (*buf != '\0') {
    // Skip leading whitespace
    while (*buf == ' ' || *buf == '\t' || *buf == '\n')
      *buf++ = '\0';

    // If reached the end of the buffer, break
    if (*buf == '\0')
      break;

    // Save the current argument
    args[argc++] = buf;

    // Find the end of the current argument
    while (*buf != '\0' && *buf != ' ' && *buf != '\t' && *buf != '\n')
      buf++;
  }

  // Null-terminate the argument list
  args[argc] = 0;

  return argc;
}
```

```
int
main(int argc, char *argv[])
{
    static char buf[100];
    printf(1, "Process manager start\n");

    while(getcmd(buf, sizeof(buf)) >= 0){

        char *args[10];  // Maximum 10 arguments
        parse(buf, args);

        if (args[0] == 0) continue;

        if (strcmp(args[0], "list") == 0) {
            printf(1, "Running the list command\n");
            procdump();

        }
        else if (strcmp(args[0], "kill") == 0) {
            if(args[1] != 0) {
                printf(1, "Running the kill command\n");

                uint pid = atoi(args[1]);

                if(kill(pid) == 0) {
                    printf(1,"SUCCESS : pid %d killed\n",pid);
                }
                else printf(1,"ERROR : pid %d \n",pid);

                }
            }
        else if (strcmp(args[0], "execute") == 0) {
            if (args[1] != 0 && args[2] != 0) {
            printf(1, "Running the execute command with path: %s and stacksize: %s\n", args[1], args[2]);

            char* path = args[1];
            char *val[2] = {path,0};
            uint stacksize = atoi(args[2]);

            int pid = fork();

            if(pid == 0) {
                if(exec2(path,val,stacksize) == -1) printf(1,"ERROR : exec2 fail\n");
                exit();
                }
            }
        }
        else if (strcmp(args[0], "memlim") == 0) {

            if (args[1] != 0 && args[2] != 0) {
                printf(1, "Running the memlim command with pid: %s and limit: %s\n", args[1], args[2]);
                int pid = atoi(args[1]);
                int limit = atoi(args[2]);
                if(setmemorylimit(pid,limit) == 0) printf(1, "SUCCESS : set memory limit");
                else printf(1, "ERROR : set memory limit");
            }
        }
        else if (strcmp(args[0], "exit") == 0) {
            printf(1, "Exiting the process manager\n");
            exit();
        }
        else {
            printf(1, "Invalid command: %s\n", args[0]);
        }
    }
}
```

```
void
procdump(void)
{
  static char *states[] = {
  [UNUSED]    "unused",
  [EMBRYO]    "embryo",
  [SLEEPING]  "sleep ",
  [RUNNABLE]  "runble",
  [RUNNING]   "run   ",
  [ZOMBIE]    "zombie"
  };
  int i;
  int pnum = 0;
  struct proc *p;
  struct thread *t;
  char *state;
```

```
  uint pc[10];

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;

    pnum++;
    cprintf("\n-------------\n");
    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
      state = states[p->state];
    else
      state = "???";
    cprintf("%d. pid : %d\n", pnum, p->pid);
    cprintf("state : %s | name : %s\n", state, p->name);
    cprintf("current thread : %d\n", p->cur_thread);
    cprintf("stack pages : %d | memory size : %d | memory limit %d\n",(p->sz)/4096, p->sz, p->sz_limit);
    if(p->state == SLEEPING){
      getcallerpcs((uint*)p->context->ebp+2, pc);
      for(i=0; i<10 && pc[i] != 0; i++)
        cprintf(" %p", pc[i]);
    }
    cprintf("\n\n");

    cprintf("threads\n");

    for(t = p->ttable; t < &(p->ttable[10]); t++) {
      if(t->state == UNUSED) continue;
      if(t->state >= 0 && t->state < NELEM(states) && states[t->state])
        state = states[t->state];
      else
        state = "???";
      cprintf("%d %s", t->tid, state);
      if(p->state == SLEEPING){
        getcallerpcs((uint*)t->context->ebp+2, pc);
        for(i=0; i<10 && pc[i] != 0; i++)
        cprintf(" %p", pc[i]);
      }
      cprintf("\n-------------\n");
      cprintf("\n\n");
    }
  }
}
```

list의 경우에는 proc.c의 procdump()를 system call화 하고 여러가지 데이터를 출력할 수 있게 변경했다.

execute의 경우 stacksize를 정해줄 수 있어야하므로 exec2를 이용하여 구현하였다. 그리고 fork를 해서 exec2를 호출하기 때문에 원래 프로세스는 계속해서 pmanager를 실행할 수 있다. 그 후 zombie 상태의 자식 프로세스는 wait으로 기다려주지 않기 때문에 나중에 pmanager가 실행이 종료되면 init process가 “zombie!”를 출력하며 정리해준다.

---

## LWP

> ***allocproc()***

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  struct thread *main_thread;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED) {
      main_thread = &(p->ttable[0]);
      goto found;
    }

  release(&ptable.lock);
  return 0;

found:
  //process(main thread) info
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->cur_thread = 0;
  p->sz_limit = 0;

  //thread info
```

```
  main_thread->state = EMBRYO;
  main_thread->tid = 0;

  release(&ptable.lock);

  // Allocate kernel stack.
  if((main_thread->kstack = kalloc()) == 0){
    p->state = UNUSED;
    main_thread->state = UNUSED;
    return 0;
  }
  sp = main_thread->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *main_thread->tf;
  main_thread->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 *main_thread->context;
  main_thread->context = (struct context*)sp;
  memset(main_thread->context, 0, sizeof *main_thread->context);
  main_thread->context->eip = (uint)forkret;

  return p;
}
```

unused한 state에 있는 process를 찾는 것 까지는 동일하고, 찾은 프로세스의 main_thread(ttable의 0번칸)에다가 본래 할당받던 스택과 정보들을 초기화 해준다. proc 구조체안에도 기본적인 process가 가져야할 정보, 즉 thread가 공유하는 정보들을 초기화 해준다.

---

### *userinit()*

```
void
userinit(void)
{
  struct proc *p;
  struct thread *main_thread;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();
  main_thread = &(p->ttable[0]);

  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(main_thread->tf, 0, sizeof(*main_thread->tf));
  main_thread->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  main_thread->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  main_thread->tf->es = main_thread->tf->ds;
  main_thread->tf->ss = main_thread->tf->ds;
  main_thread->tf->eflags = FL_IF;
  main_thread->tf->esp = PGSIZE;
  main_thread->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  // this assignment to p->state lets other cores
  // run this process. the acquire forces the above
  // writes to be visible, and the lock is also needed
  // because the assignment might not be atomic.
  acquire(&ptable.lock);

  p->state = RUNNABLE;
  main_thread->state = RUNNABLE;

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

init process에 대한 정보 또한 main thread에다가 저장을 해준다. pgdir과 sz, process의 name과 같은 정보는 thread들이 공유할 예정이기 때문에 원래 저장하던 것처럼 저장을 해주었다.

---

### *copy_thread() / copy_proc()*

```
void
copy_thread(struct proc* running_p, struct thread* running_thread)
{
  running_p->kstack = running_thread->kstack;
  running_p->tf = running_thread->tf;
  running_p->context = running_thread->context;

  running_p->cur_thread = running_thread->tid;
}
```

```
void
copy_process(struct proc* running_p, struct thread* running_thread)
{
  running_thread->kstack = running_p->kstack;
  running_thread->tf = running_p->tf;
  running_thread->context = running_p->context;
}
```

이 함수들은 thread의 정보를 process로 가지고 올라왔다 내려갔다 할 때 필요해서 proc.c에 구현했다. kstack, tf, context를 스케줄링할 때 위 쓰레드와 proc 구조체 안에서 옮겨주었다. cur_thread는 현재 어떤 thread가 실행되고 있는지 저장해주는데, 이는 초기화 되지 않고 남기 때문에 가장 마지막에 어떤 thread가 실행되었는 지 알 수 있다. 이를 통해 RR 스케줄링을 제대로 구현할 수 있었다.

---

### *scheduler()*

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  struct thread *t;
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE){
        continue;
      }

        int i = 0;
        for(t = &(p->ttable[0]); t < &(p->ttable[10]); t++){ // find last_sched thread arr index
          if(t->tid == p->cur_thread)
            break;
          i++;
        }

        int idx, thread_num;
        for(idx = 1; idx <= 10; idx++) { // find which thread have to be scheduling
          thread_num = i + idx;
          if(( t = (&(p->ttable[thread_num%10])) )->state == RUNNABLE){

            copy_thread(p,t);
            // Switch to chosen process.  It is the process's job
            // to release ptable.lock and then reacquire it
            // before jumping back to us.
            c->proc = p;
            switchuvm(p);
            p->state = RUNNING;

            swtch(&(c->scheduler), p->context);
            switchkvm();

            if(p->state != ZOMBIE)
              copy_process(p,&(p->ttable[p->cur_thread]));
            else break;

            // Process is done running for now.
            // It should have changed its p->state before coming back.
            c->proc = 0;
          }
```

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

  }
}
```

⚙️ ***코드 디자인***

1. runnable한 process 탐색
2. runnable한 process의 thread중 runnable 한 thread 탐색
3. 실행하기로 선택된 thread 스택을 proc 구조체에 채워넣음
4. running
5. 실행을 완료한 thread를 thread 구조체에 다시 최신화

runnable한 process를 찾는 과정은 원래 xv6의 방식과 동일하다. runnable한 process를 찾은 이후에는 cur_thread를 기준으로, 즉 가장 최근에 실행되었던 thread를 기준으로 그 다음 쓰레드부터 runnable한 쓰레드가 있는지 탐색을 하는 방식으로 RR을 구현했다. circular하게 구현하기 위하여 모듈러 연산을 활용했다. 그렇게 실행할 thread를 찾게되면 찾은 thread를 `copy_thread` 함수를 통해 proc 구조체에 가져와 붙여준다. context switching을 통해 일을 하고 일을 끝마치고 스케줄러로 돌아오게 된다면 우선 proc의 state가 zombie인지 확인한다. zombie 상태가 되었다면 굳이 copy_process를 해줄 필요 없이 더 이상 스케줄링이 되지 않고 있다가 wait함수에 의해 자원이 수거될 것이다.

---

### *sleep()*

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();
  struct thread *t;

  for(t = p->ttable; t < &(p->ttable[10]); t++) {
    if(t->tid == p->cur_thread) break;
  }

  ...

  // Go to sleep.
  t->chan = chan;
  t->state = SLEEPING;
  p->state = RUNNABLE;

  sched();

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

  ...
}
```

이렇게 state를 변화하는 함수를 수정할 때에는 thread와 process의 상태를 항상 잘 고려해주어야 한다. 많이 수정하진 않았고 이 sleep을 호출한 thread를 cur_thread 정보로 알아내고, thread의 상태는 sleeping, process의 상태는 runnable로 만들어 주었다. 그래야 우리가 구현하는 쓰레드가 쓰레드 별로 독립적으로 sleep에 빠질 수 있고, 다른 쓰레드가 sleep 중이어도 process의 상태가 runnable이라 스케줄링 될 여지가 생긴다. 당연히 chan을 thread에 저장하여 thread가 각자 깰 수 있게 만들었다. wakeup 함수의 호출로 해당 thread가 기상하게 되면 역시 thread의 chan을 초기화 해주게 수정했다.

---

### *wakeup()*

```
static void
wakeup1(void *chan)
{
  struct proc *p;
  struct thread *main_thread;
  struct thread *t;
```

```
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    main_thread = &(p->ttable[0]);
    for(t = main_thread; t< &(p->ttable[10]); t++) {
      if(t->state == SLEEPING && t->chan == chan)
        t->state = RUNNABLE;
      }
  }
}
```

원래 process의 chan을 참조하여 깨우던 방식에서 모든 프로세스의 ttable을 전부 순회하며 chan에서 자고 있는 thread들을 깨워주는 방식으로 수정했다.

---

## *exec()*

```
int
exec(char *path, char **argv)
{

  ...

// check memeory limit
if(curproc->sz_limit) {
  if(ph.vaddr + ph.memsz > curproc->sz_limit) {
    cprintf("EXCEPTION : memory limit - exec - A\n");
    goto bad;
  }
}

for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
  if(ph.type != ELF_PROG_LOAD)
    continue;
  if(ph.memsz < ph.filesz)
    goto bad;
  if(ph.vaddr + ph.memsz < ph.vaddr)
    goto bad;
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
iunlockput(ip);
end_op();
ip = 0;

// Allocate two pages at the next page boundary.
// Make the first inaccessible.  Use the second as the user stack.
sz = PGROUNDUP(sz); // round-up stack size to allocate in memory.

// check memeory limit
if(curproc->sz_limit) {
  if(sz+(2*PGSIZE) > curproc->sz_limit) {
    cprintf("EXCEPTION : memory limit - exec - B\n");
    goto bad;
  }
}

if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
  goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;


...


pushcli();

// 1. 지금까지 실행되던 모든 쓰레드를 종료하고 exec를 호출한 thread 하나만 남게 해야합니다.
struct thread *t;
struct thread *exec_call_thread = 0;
for(int i=9; i>=0; i--) {
  t = &(curproc->ttable[i]);
  if(t->state == UNUSED) continue;

  // 1-1. 붙여넣을 thread, 즉 exec를 호출한 thread는 남겨둡니다.
```

```
    if(t->tid == curproc->cur_thread) {
      exec_call_thread = t;
      continue;
    }

    kfree(t->kstack);
    t->kstack = 0;
    t->tf = 0;
    t->context = 0;
    t->state = UNUSED;
    t->tid = 0;
    t->chan = 0;
  }

  t = &(curproc->ttable[0]);

// 2. main_thread의 위치로 thread를 옮겨주어야 합니다.

  if(exec_call_thread->tid != 0) {
    t->kstack = exec_call_thread->kstack;
    t->tf = exec_call_thread->tf;
    t->context = exec_call_thread->context;
    t->tid = 0;

    exec_call_thread->kstack = 0;
    exec_call_thread->tf = 0;
    exec_call_thread->context = 0;
    exec_call_thread->state = UNUSED;
    exec_call_thread->tid = 0;
  }

  curproc->cur_thread = 0;
  t->state = RUNNABLE;

  // Commit to the user image.
  // 3. proc 구조체 안에 정보를 저장합니다.
  oldpgdir = curproc->pgdir;
  curproc->pgdir = pgdir;
  curproc->sz = sz;
  curproc->tf->eip = elf.entry;
  curproc->tf->esp = sp;
  switchuvm(curproc);
  freevm(oldpgdir);

  popcli();

  return 0;

  ...
}
```

전체적인 구조를 바꾸진 않았고, 우선 `allocuvm`할 때 proc 구조체의 memory limit을 확인하고 메모리보다 실행할 수 없게 했다. 즉, 만약에 limit를 걸고 exec를 진행하게 되면 오류가 exec가 실행되지 않을 수 있다. 안전하게 하려면 limit을 0으로 초기화 해주고 진행해야한다. exec를 통한 공간 할당을 완료한 후에는 exec를 호출한 쓰레드를 제외하고 모두 정리해준다. 그리고선 무조건 지금 실행중인 쓰레드를 0번 쓰레드, main_thread로 만들어주고 현재 process에 tf를 수정해준다. 그 이유는 exec가 끝나면 scheduler로 돌아갈 텐데 그럼 `copy_process` 함수를 통해 process에 있는 정보를 thread로 내려줄 것이기 때문이다.

### *fork()*

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct thread *main_thread;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  main_thread = &(np->ttable[0]);

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    kfree(main_thread->kstack);
    np->kstack = 0;
    main_thread->kstack = 0;
```

```
  np->state = UNUSED;
  main_thread->state = UNUSED;
  return -1;
}
np->sz = curproc->sz;
np->sz_limit = curproc->sz_limit;
np->parent = curproc;
*main_thread->tf = *curproc->tf;

// Clear %eax so that fork returns 0 in the child.
main_thread->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;
main_thread->state = RUNNABLE;

release(&ptable.lock);

return pid;
}
```

fork에서는 copyuvm에 실패하면 unused로 상태 초기화 해주는 것을 바꿔주었고, np는 exec와 다르게 나중에 쓰레드의 정보를 process에 올려주는 것부터 스케줄링이 될 예정이므로 np의 main thread에 정보를 복사해주었다. 그리고 main_thread의 상태를 runnable로 바꿔주는 것도 잊지 않았다.

---

### *wait()*

```
int
wait(void)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;

      if(p->state == ZOMBIE) {
        pid = p->pid;
        struct thread *t;
        for(int i=9; i>=0; i--) {
          t = &(p->ttable[i]);
          if(t->state == UNUSED) continue;

          kfree(t->kstack);
          t->tid = 0;
          t->chan = 0;
          t->tf = 0;
          t->context = 0;
          t->kstack = 0;
          t->state = UNUSED;
        }

        freevm(p->pgdir);
        p->pid = 0;
        p->kstack = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;

        for(int i=0; i<9; i++) p->thread_pool[i] = 0;

        release(&ptable.lock);
```

```
        return pid;
      }
    }


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

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

wait에서는 자식 프로세스의 상태가 zombie가 되면 해당 프로세스의 모든 쓰레드의 여러 자원을 정리해 주었다. 중요한 것은 thread_pool을 항상 0으로 초기화해 주어야 한다는 것이다. 그렇지 않으면 할당받지 못한 공간에 접근할 수 있다.

---

### *kill()*

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

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      thread = &(p->ttable[0]);
      // Wake process from sleep if necessary.
      for(thread = &(p->ttable[0]); thread < &(p->ttable[10]); thread++){
        if(thread->state == SLEEPING)
          thread->state = RUNNABLE;
      }
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

kill은 killed 변수를 process가 가지고 있다가 1이 되면 process state 자체를 zombie로 만들어주기 때문에 thread가 모두 공유하게 했다. 추가한 부분은 thread를 깨워서 exit을 호출할 수 있게 해준 부분이다.

exit의 경우 변경 사항이 없어 넘어가겠다.

---

thread_create, thread_exit, thread_join의 경우 thread.c 라는 파일에 따로 빼서 작업을 진행했습니다. 참고 부탁드리겠습니다.

---

### *thread_create*
### *1단계. ttable 탐색*

```
int thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg)
{
    struct proc *p = myproc();

    pushcli();
    struct thread *t = 0;
    struct thread *main_thread = &(p->ttable[0]);
    struct thread *now_thread;

    for(now_thread = main_thread; now_thread < &(p->ttable[10]); now_thread++) {
        if(now_thread->tid == p->cur_thread) break;
    }

    char *sp;

    // 예외처리
    // A. 쓰레드 테이블이 가득찼다면 "쓰레드를 더이상 만들 수 없음" 이라고 출력하고 return -1
    // B. 쓰레드를 골랐는데 main thread가 선택되었을 때 return -1
    // C. 메인 쓰레드가 아닌 쓰레드가 thread_create를 호출 했을 때 return -1
```

```
// 코드 디자인
// 1. 할당받을 쓰레드를 탐색

int chk_valid_thread = 0;
int thread_index = 0;

for(t = main_thread; t < &(p->ttable[10]); t++) {
    if(t->state == UNUSED) {
        chk_valid_thread = 1;
        break;
    }
    thread_index++;
}

// 예외 A
if(chk_valid_thread == 0) {
    cprintf("EXCEPTION 0 : The maximum number of threads has already been allocated.\n");
    popcli();
    return -1;
}

// 예외 B
if(thread_index == 0) {
    cprintf("EXCEPTION 1 : Main thread terminated.\n");
    popcli();
    return -1;
    }

    // 예외 C
    if(now_thread != main_thread) {
        cprintf("EXCEPTION 2 : Caller is not main thread\n");
        popcli();
        return -1;
    }

    //thread info
    t->state = EMBRYO;
    t->tid = thread_index;
    *thread = t->tid;
```

전체적인 코드 디자인은 위에 설명한 것과 같다. proc 구조체 안에 아직 할당 받지 않은 ttable 공간을 찾아 포인터로 지정해준다. 이제 해당 쓰레드에 스택 할당을 해줄 예정이다. 기본적인 tid, state같은 정보를 업데이트 해주고 tid를 thread에 저장한다.

---

> ***thread_create***
> ***2단계. thread kstack 할당***

```
// 2. thread를 할당받음

    // Allocate kernel stack.
    if((t->kstack = kalloc()) == 0){
        p->state = UNUSED;
        t->state = UNUSED;
        return 0;
    }

    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->tf = *now_thread->tf;
    t->tf->eax = 0;
```

이 과정은 fork의 과정을 일부 차용해왔다. `allocuvm`의 과정을 거친 후 새로 할당받을 쓰레드의 tf에 thread_create을 호출한 쓰레드(main_thread)의 tf를 붙여주었다. 마치 fork를 하고 exec을 하는 것처럼 생각하기 위함이었다.

---

### *thread_create*
### *3단계. thread kstack 할당*

```
// 3. thread에 함수 정보 저장

    uint sz = p->sz;
    uint spt;
    uint ustack[2];
    pde_t *pgdir = p->pgdir;
    int i;

    uint pool_sz;
    int find = 0;

    // thread_pool에 빈 user stack이 있는지 탐색
    for(i=0; i<10; i++) {
     if(p->thread_pool[i] != 0) {
            pool_sz = p->thread_pool[i];
            p->thread_pool[i] = 0;
            find = 1;
            break;
        }
    }

    // 빈 user stack을 발견하지 못했다면 할당을 해주고 발견했다면 할당하지 않고 재활용함
    if(find == 0) {
        // Allocate two pages at the next page boundary.
        // Make the first inaccessible.  Use the second as the user stack.
        sz = PGROUNDUP(sz); // round-up stack size to allocate in memory.
        t->start = sz;
        if(p->sz_limit) {
            if(sz+(2*PGSIZE) > p->sz_limit) {
            cprintf("EXCEPTION : memory limit - thread_create\n");
            goto bad;
            }
        }

        if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
            goto bad;
        clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
        spt = sz;
    }
    else {
        t->start = pool_sz;
        spt = t->start;
    }

    // 스택 영역에 인자 저장

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

    spt = spt - sizeof(ustack);

    if(copyout(pgdir, spt, ustack, sizeof(ustack)) < 0) {
        cprintf("spt : %d\n",spt);
        panic("dead");
        goto bad;
    }

    p->sz = sz;
    p->pgdir = pgdir;
    t->tf->eip = (uint)start_routine;
    t->tf->esp = spt;
    switchuvm(p);

    t->state = RUNNABLE;

    popcli();

    return 0;

  bad:
    if(pgdir) freevm(pgdir);
    popcli();
    return -1;
}
```

본 과정은 exec의 과정을 일부 차용해 왔다. page를 할당해주고 user stack에 정보를 저장해주는 것이 차용해온 부분인데, thread에 올바르게 수정을 해주었다. 우선 나의 thread는 무조건 thread에게 page를 새로 할당해주는 것이 아니라, exit 할 때 thread가 사용했던 stack의 시작 주소를 thread_pool에 저장해주었다가 thread_pool에 인자가 있다면 그 공간을 재사용하는 방식을 선택했다. 이를 통해 정말 많은

thread_create과 thread_exit을 반복해도 올바르게 실행되게 하였다. 페이지 문제를 해결한 이후에는 user stack을 과제 명세 함수 꼴에 맞추어 주고 eip와 esp register를 설정해주었다.

### *thread_exit()*

```
void thread_exit(void *retval)
{
    struct proc *p = myproc();
    struct thread *main_thread = &(p->ttable[0]);
    struct thread *t;

    acquire(&ptable.lock);

    for(t = main_thread; t < &(p->ttable[10]); t++) {
        if(t->tid == p->cur_thread) break;
    }

    if(t == main_thread) {
        cprintf("Exception 0 : Attempting to exit the main thread");
        return ;
    }

    thread_wakeup(main_thread);

    t->retval = retval;
    t->state = ZOMBIE;
    p->state = RUNNABLE;

    sched();

    cprintf("ZOMBIE thread %d\n",t->tid);
    panic("zombie exit");
}
```

정말 간단하게 구현했다. proc.c 의 exit의 형태를 따르는데 thread가 exit할 때 필요한 것만 다뤄 주었다. 그리고 retval을 전달하는 것을 잊지 않았다.

### *thread_wakeup()*

```
void thread_wakeup(void *chan)
{
    pushcli();
    struct proc *p = myproc();
    struct thread *main_thread = &(p->ttable[0]);

    main_thread->state = RUNNABLE;
    popcli();
}
```

이것은 wakeup1을 굳이 호출하지 않고 main thread를 깨워주기 위해서 따로 만든 함수이다.

### *thread_join()*

```
int thread_join(thread_t thread, void **retval)
{
    struct proc *p = myproc();
    struct thread *join_thread;
    struct thread *main_thread = &(p->ttable[p->cur_thread]);
    int i;

    if(p->cur_thread != 0) {
        cprintf("EXCEPTION 0 : Not mainthread join\n");
        return -1;
    }

    acquire(&ptable.lock);

    for(join_thread = p->ttable; join_thread < &(p->ttable[10]); join_thread++) {
        if(join_thread->tid == thread) break;
    }

    // thread 자원회수
```

```
    for(;;)
    {
        if(join_thread->state == ZOMBIE) {

            kfree(join_thread->kstack);
            join_thread->kstack = 0;
            join_thread->state = UNUSED;
            join_thread->tf = 0;
            join_thread->context = 0;
            join_thread->tid = 0;

            *retval = join_thread->retval;
            join_thread->retval = 0;

            for(i=0; i<10; i++) {
                if(p->thread_pool[i] == 0) {
                    p->thread_pool[i] = join_thread->start;
                    break;
                }
            }

            switchuvm(p);

            release(&ptable.lock);

            return 0;
        }

        sleep(main_thread, &ptable.lock);
    }
}
```

wait함수를 원형으로 해서 간단하게 구현했다. join할 thread를 tid에 기반하여 찾고 해당 thread의 state가 zombie가 될 때까지 main_thread가 잠을 자다가 일어나 자원을 회수해준다. 항상 꼭 thread_pool의 정보를 초기화해 주어야 한다.

---

# Result

## LWP

> ***thread_test.c***

*테스트 파일로 thread_test.c 라는 파일을 넣어두었습니다.다음 사진은 실행 결과 사진입니다. NUMTHREAD == 8*

```
$ thread_test
0. racingtest start
1571695
0. racingtest finish
1. basictest start
12345678345812764123578642568137425813 67
1. basictest finish
2. jointest1 start
thread_join!!!
thread_exit...
thread_exit...
thread_exit...
thread_exithread_exit...
thread_exit...
thread_exit...
thread_exit...
t...

2. jointest1 finish
3. jointest2 start
thread_ethread_exit...
thread_exit...
thread_exit...
thread_exit...
thread_exit...
thread_exit...
thread_exit...
xit...
thread_join!!!

3. jointest2 finish
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
5. exittest1 start
thread_exit ...
thread_exit ...
tttttt5. exittest1 finish
6. exittest2 start
6. exittest2 finish
7. forktest start
child
child
child
child
chilchild
child
parent
parent
parent
parent
parent
parent
parent
parent
child
d
7. forktest finish
8. exectest start
echo is executed!
8. exectest finish
9. sbrktest start
9. sbrktest finish
10. killtest start
10. killtest finish
11. pipetest start
11. pipetest finish
12. sleeptest start
12. sleeptest finish
```

## racing test

```
void nop(){ }

void*
racingthreadmain(void *arg)
{
  int tid = (int) arg;
  int i;
  int tmp;
  for (i = 0; i < 1000000; 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+1];
  int i;
  void *retval;
  gcnt = 0;

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

thread를 여러개 만들고 일부로 race condition을 일으켜 제대로 공유를 하는지 확인해보는 테스트이다. 테스트를 돌릴 때마다 다른 값이 잘 출력된다.

## basic test

```
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+1];
  int i;
  void *retval;

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

```
  for (i = 1; 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;
}
```

thread 8개를 만들고 실행한 thread들이 자신의 tid를 출력하는 것을 보여주고있다. 한 쓰레드만 실행되는 것이 아니라 여러 쓰레드들이 열심히 context switching되며 돌아가고 있다는 것을 확인할 수 있다. 그리고 기본적으로 thread_create, thread_exit, thread_join 함수가 활용되고 있으므로 각 함수가 재역할을 잘 하고 있음도 간접적으로 알 수 있다.

---

## join test

```
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;
}
```

thread_create를 하고 난 이후에 sleep을 이용하여 쓰레드가 죽는 타이밍을 달리하여 보는 테스트 케이스이다. 이를 통해 thread_join이 thread_exit보다 먼저 호출되든 나중에 호출되든 thread가 잘 정리되는 것을 볼 수 있다.

---

## stress test

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

  return 0;
}

int
stresstest(void)
{
  const int nstress = 35000;
  thread_t threads[NUM_THREAD+1];
  int i, n;
  void *retval;

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

thread를 단순히 만들었다 지웠다를 35000번정도 반복하는 테스트 케이스이다. 전에 stack의 재활용을 적용하지 않았을 때는 3000 언저리에서 panic이 발생하거나 xv6가 재부팅 되었는데, 재활용 코드를 추가하고 나서 제대로 실행되는 모습이다.

---

## exit test

```
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;
}
```

main thread가 return0에 의해 직접 zombie가 되는 과정과 thread가 exit을 호출하는 과정을 모두 보여주고 있다. 두 경우에 모두 안정적으로 test가 종료되는 것을 볼 수 있다.

## fork test

```
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;
}
```

thread가 안정적으로 fork를 하는지 테스트해보는 코드이다. 총 8번씩 parent와 child를 안정적으로 출력하는 것을 볼 수 있다.

## exec test

```
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;
}
```

thread가 exec를 호출할 수 있는지 test해보는 코드이다. main thread는 exit을 할 수 없으므로 exec 이후에는 exit함수를 호출해주어야 한다. 안정적으로 echo 함수를 호출하고 종료되는 모습이다.

## kill test

```
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;
}
```

thread가 kill을 호출하여 자신이 속한 process를 kill하는 테스트이다. 안정적으로 잘 죽는 모습이다.

## sbrk test

```
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;
}
```

sbrk를 thread가 잘 실행할 수 있는지 확인하는 코드이다. 아무런 문제 없이 잘 실행된다.

### sleep test

```
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;
}
```

thread가 sleeping 일때도 main thread가 return 0을 하면 잘 정리해주는 지 확인해보는 코드이다. 아무런 문제 없이 잘 실행된다.

*해당 위키를 작성하면서 혹시모를 오류가 발생할 수 있을 것 같아 작성하는 내내 테스트 코드를 실행시켜보고 있는데, 오류가 발생하지 않았다.*

## Pmanager

### *list*

```
(PMG) : list
Running the list command

_____________
1. pid : 1
state : runble | name : init
current thread : 0
stack pages : 3 | memory size : 12288 | memory limit 0


threads
0 sleep
_____________



_____________
2. pid : 2
state : runble | name : sh
current thread : 0
stack pages : 4 | memory size : 16384 | memory limit 0


threads
0 sleep
_____________



_____________
3. pid : 3
state : run    | name : pmanager
current thread : 0
stack pages : 4 | memory size : 16384 | memory limit 0


threads
0 runble
_____________
```

pmanager를 실행하면 (PMG) : 라는 멘트가 출력되고 줄 입력을 넣을 수 있다. 이는 list를 입력했을 때 현재 실행되고 있는 process에 관한 정보를 출력해주는 모습이다. thread는 자세한 정보를 출력하지 않고 어떤 thread가 실행되고 있는지와 상태정도만 출력할 수 있게 했다.

## kill

```
(PMG) : kill 3
Running the kill command
$
```

kill 함수를 이용해서 pmanager를 kill 시켜보는 실험을 해보았다. 잘 종료되고 다시 shell이 실행되는 모습이다.

## execute

이 실험을 통해서 exec2가 제대로 실행되는지 알 수 있다.

```
Process manager start
(PMG) : execute forktest 34
Running the execute command with path: forktest and stacksize: 34
(PMG) : fork test
fork test OK
```

```
____________
4. pid : 5
state : zombie | name : forktest
current thread : 0
stack pages : 36 | memory size : 147456 | memory limit 0


threads
0 runble
```

execute로 forktest를 실행해보는 모습이다. stack page도 정상적으로 들어간 모습이다.

```
(PMG) : exit
Exiting the process manager
zombie!
$ zombie!
```

제대로 wait을 걸어주지 않기에 init process가 정상적으로 자원을 정리해주는 모습이다.

---

## memlim

이것이 잘 실행되는지를 통하여 setmemorylimit가 잘 실행되는지 알 수 있다.

우선 list를 통해 원래 process의 정보를 확인해준다.

```
$ pmanager
Process manager start
(PMG) : list
Running the list command

_____________
1. pid : 1
state : runble | name : init
current thread : 0
stack pages : 3 | memory size : 12288 | memory limit 0


threads
0 sleep
_____________



_____________
2. pid : 2
state : runble | name : sh
current thread : 0
stack pages : 4 | memory size : 16384 | memory limit 0


threads
0 sleep
_____________



_____________
3. pid : 3
state : run    | name : pmanager
current thread : 0
stack pages : 4 | memory size : 16384 | memory limit 0


threads
0 runble
_____________
```

memory limit을 10으로 설정해본다. 정상적으로 동작한다.

```
(PMG) : memlim 3 10
Running the memlim command with pid: 3 and limit: 10
SUCCESS : set memory limit(PMG) : list
Running the list command

_____________
1. pid : 1
state : runble | name : init
current thread : 0
stack pages : 3 | memory size : 12288 | memory limit 0


threads
0 sleep
_____________



_____________
2. pid : 2
state : runble | name : sh
current thread : 0
stack pages : 4 | memory size : 16384 | memory limit 0


threads
0 sleep
_____________



_____________
3. pid : 3
state : run    | name : pmanager
current thread : 0
stack pages : 4 | memory size : 16384 | memory limit 10


threads
```

다시 무한을 뜻하는 0을 설정해 본다. 잘 동작하는 모습이다.

```
(PMG) : memlim 3 0
Running the memlim command with pid: 3 and limit: 0
SUCCESS : set memory limit(PMG) : list
Running the list command

______________
1. pid : 1
state : runble | name : init
current thread : 0
stack pages : 3 | memory size : 12288 | memory limit 0


threads
0 sleep
______________



______________
2. pid : 2
state : runble | name : sh
current thread : 0
stack pages : 4 | memory size : 16384 | memory limit 0


threads
0 sleep
______________



______________
3. pid : 3
state : run    | name : pmanager
current thread : 0
stack pages : 4 | memory size : 16384 | memory limit 0


threads
0 runble
```

이번에는 limit을 5로 설정하고 memroy size보다 작은 limit 을 걸어본다. 오류가 제대로 나는 모습이다.

```
-------------
3. pid : 3
state : run     | name : pmanager
current thread : 0
stack pages : 4 | memory size : 16384 | memory limit 5


threads
0 runble
-------------


(PMG) : memlim 3
(PMG) : memlim 3 3
Running the memlim command with pid: 3 and limit: 3
EXCEPTION 2 : sz already bigger than limit
ERROR : set memory limit(PMG) : list
```

> wrong command

```
(PMG) : ㅇㄴ러ㅏ
Invalid command: ㅇㄴ러ㅏ
(PMG) : djfo
Invalid command: djfo
(PMG) :
```

제대로 invalid 하다고 알려준다.

> exit

```
(PMG) : djfo
Invalid command: djfo
(PMG) : exit
Exiting the process manager
$
```

다시 shell이 잘 실행되는 모습이다.

## trouble shooting

```
8. exectest start

_____________
1. pid : 1
state : runble | name : init
current thread : 0
stack pages : 3 | memory size : 12288 | memory limit 0


threads
0 sleep
_____________



_____________
2. pid : 2
state : runble | name : sh
current thread : 0
stack pages : 4 | memory size : 16384 | memory limit 0


threads
0 sleep
_____________



_____________
3. pid : 4
state : runble | name : thread_test
current thread : 0
stack pages : 5 | memory size : 20480 | memory limit 0


threads
0 sleep
_____________



_____________
4. pid : 21
state : runble | name : echo
current thread : 0
stack pages : 3 | memory size : 12288 | memory limit 0


threads
0 sleep
_____________
```

```
$ thre
8. exectest start
echo is executed!
8. exectest finish
$ 8. exectest start
echo is executed!
8. exectest finish
$ 8. exectest start
echo is executed!
8. exectest finish
$ 8. exectest start
echo is executed!
8. exectest finish
$ 8. exectest start
echo is executed!
8. exectest finish
$ 8. exectest start
echo is executed!
8. exectest finish
$ 8. exectest start
echo is executed!
8. exectest finish
$ exec: fail
exec thre failed
$ thread_test 8 8
8. exectest start
echo is executed!
8. exectest finish
thread_test 8 8
thread_test 8 8
thread_test 8 8
thread_test 8 8$
8. exectest start
echo is executed!
8. exectest finish
thread_test 8 8
thread_test 8 8
thread_test 8 8
thread_test 8 8
$ 8. exectest start
echo is executed!
8. exectest finish
thread_test 8 8
thread_test 8 8
$
8. exectest start
echo is executed!
8. exectest finish
thread_test 8 8$ thread_test 8 88. exectest start

echo is executed!
8. exectest finish
$ 8. exectest start
echo is executed!
8. exectest finish
$ 8. exectest start
echo is executed!
8. exectest finish
$ 8. exectest start
echo is executed!
8. exectest finish
$ 8. exectest start
echo is executed!
8. exectest finish
$ 8. exectest start
```

정말 가끔씩 실행을 하다가 모든 process의 main thread의 상태가 sleep에 빠지는 기 현상이 발생했는데, 하필이면 이것이 또 항상 exectest 에서만 발생했다. 이에 대해 좀 고민을 해보았는데 결론을 내리지 못했다.