OS assignment2

Design

pthread 구현

1. pthread 구현방식

- thread는 기존의 `proc` 구조체와 상당한 유사점을 가지고 있다. 그렇기에 thread구조체를 따로 만들지 않고 기존 `proc` 구조체를 그대로 활용하기로 하였다.
- 그리고 프로세스를 `mainthread`로 지정해주고 `next_thread`에 linkedlist 방식으로 thread를 매달아 주었다.

2. pthread 스케줄링

- 프로세스간의 공정성과 thread들간의 공정성을 둘다 유지해야한다고 생각하였다.
- 그래서 ptable을 순회하며 각 프로세스마다 하나의 thread를 스케줄링하고 다음 프로세스로 넘어가게끔 설계하였다.
- 프로세스 내에서도 current_thread 변수를 사용하여 그 프로세스 내에서 스케줄링 된 thread를 가리키고 다음에 그 프로세스가 스케줄링 될때는 `current_thread`의 다음 thread를 스케줄링 하게 하여서 thread간의 공정성도 유지하였다.

implement

1. 기본선언

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  struct proc *next_thread;    // thread를 linkedlist를 위한 프로세스 포인터
  struct proc *main_thread;    // Main thread
  struct proc *current_thread; // 프로세스 내에서 현재까지 실행된 thread
  struct proc *join_thread;    // 프로세스가 생성되고 join을 해줄 thread
  int is_thread;               // thread이면 1 mainthread이면 0
  int tid;                     // thread의 tid
  int create_num;              // tid를 위해 프로세스에서 생성했던 thread수
  void *retval;                // 반환값
  int memory_limit;            // memory_limit
};
```

proc의 구조는 다음과같다 즉 thread의 구조도 다음과 같다

`next_thread` : thread를 생성하면 next_thread에 연결해준다.

`main_thread` : main_thread를 나타낸다 즉 맨처음 생성된 프로세스

`current_thread` : 스케줄링을 위한 변수이다. 한 프로세스내에서 현재까지 실행된 thread를 저장하고 다음 스케줄링이 될때 current_thread의 다음 thread를 스케줄링 한다.

`join_thread` : thread를 생성할때 생성한 thread를 join_thread로 지정하고, thread_exit이 호출되면 join_thread를 깨워 자원을 회수한다.

`is_thread` : thread이면 1 mainthread 즉 process이면 0의 값을 갖는다.

`tid` : thread의 tid, main thread이면 0을 갖는다.

`create_num` : tid값을 부여하기 위하여 process 가 thread를 생성할 때 마다 값을 늘려주고 그 값을 tid로 부여한다.

`memory_limit` : setmemorylimit을 통하여 mainthread의 memory limit을 설정해준다.

```
static struct proc*
allocproc(void)
{
  ...
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->is_thread = 0;
  p->tid = 0;
  p->create_num = 0;
  p->current_thread = p;
  p->main_thread = p;
  ...
}
```

`allocproc()`에서 `mainthread`를 위한 초기화를 해주었다. 이후에 `thread_create`에서 thread인 경우에 다시 할당해준다.

---

2. thread 관련 함수

proc.c

- thread_create()

```
int thread_create(thread_t *thread, void *(*start_routine)(void*), void *arg) {
    struct proc *np;
    struct proc *curproc = myproc();
    struct proc *mainThread=curproc->main_thread;
    uint usp,sz,newsz;

    // 프로세스 할당
    if((np = allocproc()) == 0){
      return -1;
    }
    acquire(&ptable.lock);
    np->main_thread = mainThread; //mainthread 지정
    np->parent= mainThread->parent; //부모 프로세스 지정
    np->pid = mainThread->pid; // mainthread와 pid공유
    *thread=++mainThread->create_num; // thread 번호 저장
    np->tid = mainThread->create_num; // thread 번호 tid에 저장
    np->pgdir = curproc->pgdir; // pgdir 복사
    // Clear %eax so that fork returns 0 in the child.

    // thread 스택 공간 확보
    sz=mainThread->sz;
    sz=PGROUNDUP(sz);
```

```
  newsz = allocuvm(curproc->pgdir, sz, sz + 2*PGSIZE);

  // 메모리 limit check
  uint memory_limit = mainThread->memory_limit;
  if (memory_limit > 0 && newsz > memory_limit) {
    release(&ptable.lock);
    return -1;  // 메모리 제한 초과시, 오류 반환
  }

  np->sz = newsz;
  mainThread->sz = newsz;
  clearpteu(curproc->pgdir, (char*)(newsz - 2*PGSIZE));

  //유저 스택 값 넣기
 *np->tf = *curproc->tf;
  np->tf->eip = (uint)start_routine; // start routine 지정
  usp = newsz;
  usp -= 8;
  uint ustack[4];
  ustack[0] = 0xffffffff;
  ustack[1] = (uint)arg;

  // 스택 포인터 이동하여 인자공간 확보
  if(copyout(np->pgdir, usp, ustack, 8) < 0){
      np->state = UNUSED;
      return -1;
  } // 스택에 인자 입력
  np->tf->esp = (uint)usp;

  // 파일 복사
  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));


  //생성된 thread linkedlist에 달아주기
  struct proc *p;
  for (p = curproc; p->next_thread != 0; p = p->next_thread);
  p->next_thread = np;
  np->next_thread = 0;

  np->state = RUNNABLE;
  np->is_thread = 1;
  np->join_thread = curproc; //join을 위한 현재 프로세스 정보 저장
  //프로세스 내의 sz 복사
  for(p = curproc->main_thread; p ; p= p->next_thread)
  {
    p->sz = newsz;
  }
  release(&ptable.lock);
```

```
    return 0;
}
```

우선 프로세스를 할당하고, 이 프로세스에 `mainthread`와 부모 프로세스, pid 등 필요한 정보를 지정한다. 그런 다음 thread의 스택 공간을 확보하고 메모리 제한을 확인합니다. 메모리 제한을 초과하는 경우, 함수는 에러(-1)를 반환한다.

그 다음, 유저 스택 값과 `start routine`을 설정하고, 스택 포인터를 이동하여 인자 공간을 확보한다. 이 과정에서 문제가 발생하면 에러를 반환한다. 그 후, 파일을 복사하고 생성된 스레드를 linked list에 추가한다. 추가된 스레드의 상태를 `RUNNABLE`로 설정하고 `join_thread`를 현재 프로세스로 지정한다.

마지막으로, sbrk 를 위해 프로세스 내의 모든 스레드의 `sz`를 갱신한다.

- thread_exit()

```
void thread_exit(void *retval){
  struct proc * curproc=myproc();
  struct proc * p;
  int fd;
  // 만약 현재 프로세스가 메인 스레드라면, exit를 호출
  if (curproc->main_thread == curproc)
  {
    exit();
  }

  // 현재 프로세스에서 열려있는 모든 파일을 닫음
  for(fd=0;fd<NOFILE;fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd]=0;
    }
  }

  // 작업 시작
  begin_op();
  // 현재 프로세스의 작업 디렉토리를 닫음
  iput(curproc->cwd);
  // 작업 종료
  end_op();
  curproc->cwd=0;

  // 프로세스 테이블 잠금
  acquire(&ptable.lock);
```

```
  // join을 기다리는 스레드 깨우기
  wakeup1(curproc->join_thread);
  // 모든 자식 프로세스를 확인해서 부모 프로세스를 initproc으로 변경하고,
  //상태가 ZOMBIE라면 initproc 깨우기
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }
  // 현재 프로세스 상태를 ZOMBIE로 변경
  curproc->state = ZOMBIE;
  // 반환값 설정
  curproc->retval = retval;
  // 스케줄러 호출
  sched();
  // 이 위치에 도달했다면 문제가 있는 것이므로 panic 호출
  panic("zombie exit");
  }
```

현재 스레드가 메인 스레드인지 확인한다. 만약 메인 스레드라면 `exit` 함수를 호출하여 프로세스를 종료시킨다.

그렇지 않다면, 현재 스레드에서 열려 있는 모든 파일을 닫고, 현재 스레드의 작업 디렉토리를 닫는다. 이 과정에서 파일 시스템 연산을 보호하기 위해 `begin_op`와 `end_op`를 호출합니다.

이후 프로세스 테이블을 잠그고, `join`을 기다리는 스레드를 깨웁니다. 그리고 현재 thread의 모든 자식 프로세스를 확인하면서, 부모 프로세스를 `initproc`으로 변경하고, 자식 프로세스의 상태가 `ZOMBIE`라면 `initproc`을 깨웁니다.

마지막으로 현재 스레드의 상태를 `ZOMBIE`로 변경하고, 반환값을 설정합니다. 그 후 스케줄러를 호출하여 다른 thread에게 넘긴다.

- thread_join()

```
int thread_join(thread_t thread, void **retval) {
    struct proc *curproc = myproc();
    struct proc *mainThread = curproc->main_thread;
    int found = 0;

    acquire(&ptable.lock);

    // join 할 thread 찾기
    struct proc *p;
    for (p = mainThread->next_thread; p ; p = p->next_thread) {
```

```
        if (p->tid == thread && p->join_thread == curproc) {
            found = 1;
            break;
        }
    }

    // thread를 찾지 못하면 -1 반환
    if (!found) {
        release(&ptable.lock);
        return -1;
    }

    // thread가 종료될 때까지 기다림
    while (p->state != ZOMBIE) {
        sleep(curproc, &ptable.lock);
    }

    // 요청된 경우 retval 설정
    // thread의 반환 값을 가져옴
    if (retval != 0) {
        *retval = p->retval;
    }

    // thread 리스트에서 제거
    if (p == mainThread->next_thread) {
        mainThread->next_thread = p->next_thread;
        if(p->main_thread->current_thread == p){
            p->main_thread->current_thread = mainThread;
        }
    } else {
        struct proc *prev;
        for (prev = mainThread->next_thread; prev->next_thread != p;
prev = prev->next_thread)
            prev->next_thread = p->next_thread;
        if(p->main_thread->current_thread == p){
            p->main_thread->current_thread = prev;
        }
    }

    // join된 thread의 자원회수
    kfree(p->kstack);
    p->kstack = 0;
    p->parent = 0;
    p->name[0] = 0;
    p->state = UNUSED;
    p->tid = 0;
    p->join_thread = 0;
    p->main_thread = 0;
    p->pid = 0;
    p->next_thread = 0;
    p->is_thread = 0;

    release(&ptable.lock);
```

```
    return 0; // 성공적으로 thread join 완료
}
```

현재 프로세스와 관련된 메인 스레드를 가져온다.

이 함수는 메인 스레드와 연결된 thread 목록에서 `join`을 기다리는 thread를 찾는다. 이 thread를 찾지 못하면 함수는 -1을 반환하고 종료한다.

thread를 찾았다면, `thread_join` 함수는 이 thread가 종료될 때까지 기다린다.

thread가 종료되면, thread의 반환값을 회수한다. 그리고 이 thread를 thread 목록에서 제거하고, 이 스레드가 사용하던 자원을 해제한다.

- scheduler()

```
void scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  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;

    struct proc *current_thread = p->main_thread->current_thread;
    //현재 프로세스의 이전까지 실행했던 thread 가져옴

    //실행가능한 다음 thread찾기
    do {
        current_thread = current_thread->next_thread;
        if (!current_thread)
            current_thread = p->main_thread;
    } while (current_thread->state != RUNNABLE);

    p->main_thread->current_thread = current_thread;//current thread 업데이트

      c->proc = current_thread;
      switchuvm(current_thread);
      current_thread->state = RUNNING;
      swtch(&(c->scheduler), current_thread->context);
      switchkvm();
```

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

      // Move to the next thread or loop back to the main thread
    }
    release(&ptable.lock);
  }
}
```

기존의 scheduling 방식에서 스케줄할 process를 찾는 부분을 바꾸었다

1. thread나 프로세스가 선택이되면 해당 프로세스의 메인 스레드를 타고가서 이전까지 실행했던 current thread를 가져온다..
2. 그 다음으로, 실행 가능한 다음 스레드를 찾는다. 만약 linkedlist의 끝까지 찾지 못하면 메인 스레드로 되돌아간다.
3. 실행 가능한 thread를 찾았다면, 메인 thread의 `current_thread`를 업데이트한다.

- fork

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

...

safestrcpy(np->name, curproc->main_thread->name, sizeof(curproc->main_thread->name));

...
```

기존 `fork` 함수에서 복사하는 부분을 fork를 실행하는 프로세스의 `mainthread`를 참조하게 해주었다.

- exec

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

uint memory_limit = curproc->main_thread->memory_limit;
  if (memory_limit > 0 && sz > memory_limit) {
    return -1;  // 메모리 limit 확인후 에러처리
  }
  curproc->tf->eip = elf.entry;  // main
  curproc->tf->esp = sp;
  kill_for_exec(curproc);

...

}
```

- `kill_for_exec`을 호출하여 thread를 정리해준다.

```
// 현재 프로세스(curproc)에 대해 exec를 위한 종료 과정을 수행하는 함수
void
kill_for_exec(struct proc * curproc){
  struct proc *p;
  struct proc *next_thread;
  int fd;

  for(p = curproc->main_thread; p; p= next_thread ){
    next_thread = p->next_thread;

    // 현재 스레드가 현재 프로세스가 아닌 경우
    if(p != curproc){

      // 스레드 상태가 ZOMBIE가 아닌 경우
      if(p->state !=ZOMBIE){

        // 열려있는 파일을 모두 닫는다
        for(fd = 0; fd < NOFILE; fd++) {
          if(p->ofile[fd]) {
            fileclose(p->ofile[fd]);
            p->ofile[fd] = 0;
          }
        }

        // 현재 작업 디렉토리를 해제
        begin_op();
        iput(p->cwd);
        end_op();
```

```
        p->cwd = 0;
      }
    }
  }

  acquire(&ptable.lock);

  // 다시 한번 현재 프로세스의 메인 스레드를 순회하며
  for(p = curproc->main_thread; p; p= next_thread ){
    next_thread = p->next_thread;

    // 현재 스레드가 현재 프로세스가 아닌 경우
    if(p != curproc){

      // 프로세스의 커널 스택을 해제
      kfree(p->kstack);
      p->kstack = 0;

      // 프로세스 정보를 초기화
      p->parent = 0;
      p->name[0] = 0;
      p->state = UNUSED;
      p->tid = 0;
      p->join_thread = 0;
      p->main_thread = 0;
      p->pid = 0;
      p->next_thread = 0;
      p->is_thread = 0;
    }
  }

  // 현재 프로세스 정보를 업데이트
  curproc->main_thread = curproc;
  curproc->is_thread = 0;
  curproc->tid = 0;
  curproc -> current_thread = curproc;
  curproc -> create_num = 0;
  curproc -> next_thread = 0;

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

- sbrk

```
int
growproc(int n)
{
  uint sz;
  struct proc *curproc = myproc();
  acquire(&ptable.lock);
  sz = curproc->main_thread->sz;
  if(n > 0){
    uint new_sz = sz + n;
    uint memory_limit = curproc->main_thread->memory_limit;
    if (memory_limit > 0 && new_sz > memory_limit) {
      release(&ptable.lock);
      return -1;  // 메모리 limit 확인후 에러발생시 -1 return
    }
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0){
      release(&ptable.lock);
      return -1;
    }
  } else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0){
      release(&ptable.lock);
      return -1;
    }
  }
  struct proc* p;
  for(p = curproc->main_thread; p ; p= p->next_thread)
  {
    p->sz = sz;
  }//변경된 sz값 해당 프로세스의 thread에 모두 적용
  release(&ptable.lock);
  switchuvm(curproc);
  return 0;
}
```

메모리를 변경해주고 변경된 메모리를 해당 thread가 속해있는 프로세스의 모든 thread에게 적용시킨다.

- exit

```
void
exit(void)
{
...

struct proc *next_thread;
  for(p = curproc->main_thread; p; p = next_thread){
```

```
    next_thread = p->next_thread;
    if(p->pid == curproc->pid && p->tid != curproc->tid){
    kfree(p->kstack);
    p->kstack = 0;
    p->parent = 0;
    p->name[0] = 0;
    p->state = UNUSED;
    p->tid = 0;
    p->join_thread = 0;
    p->main_thread = 0;
    p->pid = 0;
    p->next_thread = 0;
    p->is_thread = 0;
    }
  }//exit을 호출한 process내의 모든 thread의 자원할당을 해제

...

}
```

`exit()` 를 호출하면 현재 실행중인 프로세스 혹은 thread의 모든 thread를 자원할당해제한다.

- kill

`kill` 호출 시 **trap.c** 에서 `exit` 을 호출한다. `exit` 에서 스레드에 대한 자원할당해제 처리를 해두어 `kill` 에 대해 추가적인 처리는 하지 않았다.

- sleep

프로세스 기반으로 스레드를 구현했기에 `sleep` 를 위해 따로 처리할 내용은 없었다.

- pipe

thread 구현 후 추가적인 처리 없이도 잘 동작하였다.

---

3. Pmanager

- exec2

```
exec.c

int
exec2(char *path, char **argv, int stacksize)
{

...

sz = PGROUNDUP(sz);
  int guardpages = 1; // 가드페이지
  int stackpages = stacksize + guardpages;
  if((sz = allocuvm(pgdir, sz, sz + stackpages*PGSIZE)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
  sp = sz;

...

}
```

기존의 `exec`에서 인자로 받아온 `stacksize`와 `guardpages`만큼 스택용 페이지를 할당해주는 부분이 변경되었다.

- setmemorylimit

```
proc.c
// 프로세스의 메모리 제한을 설정하는 함수
int setmemorylimit(int pid, int limit) {
  struct proc *p;
  if(limit < 0) {
    return -1;} // 제한 값이 유효한지 확인
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {

    // 해당 pid를 가진 프로세스를 찾고, thread가 아닌 경우에만 메모리 제한을 설정
    if (p->pid == pid && p->is_thread == 0) {

      // 프로세스가 존재하는지 확인
      if (p->state == UNUSED)
      {
        release(&ptable.lock);
        return -1;
      }

      // 이미 프로세스가 새로 설정할 제한값보다 많은 메모리를 사용하고 있는지 확인
```

```
      if (limit < p->sz)
      {
        release(&ptable.lock);
        return -1;
      }

      // 메모리 limit 업데이트
      p->main_thread->memory_limit = limit;
      release(&ptable.lock);
      return 0;
    }
  }
  // 해당 pid를 가진 프로세스를 찾지 못한 경우, -1 반환
  release(&ptable.lock);
  return -1;
}
```

- list

```
int processinfo(int index, char *process_name, int *process_pid,
 int *process_stack_pages, int *process_allocated_memory, int *process_memory_limit ) {
  if(index>=NPROC) return -1;
  struct proc *p;
  acquire(&ptable.lock);
  p=ptable.proc;
  while(index>0){
    p++;
    index--;
  }
  if(p->is_thread&&p) return 0;
  else if(p->state !=UNUSED && p->killed == 0){
    safestrcpy(process_name, p->name, strlen(p->name) + 1);
// Copy process PID
    *process_pid = p->pid;
    int stack_pages = p->sz / PGSIZE;
    *process_stack_pages = stack_pages;
    *process_allocated_memory = p->sz;
    *process_memory_limit = p->main_thread->memory_limit;
    release(&ptable.lock);
    return 1;
  }
  release(&ptable.lock);
  return -1;
}
```

list를 보여주기 위한 systemcall

- pmanager.c

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

#define MAX_CMD_LENGTH 256

// 문자열을 토큰으로 나누는 함수
void tokenize(char *str, char **tokens, int max_tokens) {
  int i = 0;
  while (*str != '\0' && i < max_tokens - 1) {
    while (*str == ' ' || *str == '\t' || *str == '\n')
      str++;
    if (*str == '\0')
      break;
    tokens[i++] = str;
    while (*str != '\0' && *str != ' ' && *str != '\t' && *str != '\n')
      str++;
    if (*str == '\0')
      break;
    *str++ = '\0';
  }
  tokens[i] = 0;
}

int main(int argc, char *argv[]) {
  char cmd[MAX_CMD_LENGTH];
  char process_name[16];
  int process_pid;
  int process_stack_pages;
  int process_allocated_memory;
  int process_memory_limit;

  while (1) {
    printf(1, "pmanager> ");
    gets(cmd, MAX_CMD_LENGTH);

    // 명령어를 토큰으로 나눔
    char *tokens[MAX_CMD_LENGTH / 2];
    tokenize(cmd, tokens, MAX_CMD_LENGTH / 2);

    if (tokens[0] == 0)
      continue; //empty command 처리

    if (strcmp(tokens[0], "list") == 0) {
      int i = 0;
      int flag;
      while (1) {
        flag = processinfo(i, process_name, &process_pid,
&process_stack_pages, &process_allocated_memory, &process_memory_limit);
        i++;
```

```
        if (flag == -1)
          break;
        else if (flag == 0)
          continue;
        else {
          printf(1, "Name: %s\n", process_name);
          printf(1, "PID: %d\n", process_pid);
          printf(1, "Stack Pages: %d\n", process_stack_pages);
          printf(1, "Allocated Memory: %d bytes\n", process_allocated_memory);
          printf(1, "Memory Limit: %d bytes\n", process_memory_limit);
          printf(1, "--------------------------\n");

        }
      }
    }
    else if (strcmp(tokens[0], "kill") == 0) {
      char *pid_str = tokens[1];
      if (pid_str == 0) {
        printf(1, "Usage: kill <pid>\n");
        continue;
      }
      int pid = atoi(pid_str);
      if (kill(pid) == 0) {
        printf(1, "Process with PID %d killed successfully.\n", pid);
      } else {
        printf(1, "Failed to kill process with PID %d.\n", pid);
      }
    } else if (strcmp(tokens[0], "execute") == 0) {
      char *path = tokens[1];
      char *stacksize_str = tokens[2];
      char *val[2] = {path,0};
      if (path == 0 || stacksize_str == 0) {
        printf(1, "Usage: execute <path> <stacksize>\n");
        continue;
      }
      int stacksize = atoi(stacksize_str);
      int pid = fork();
      if(pid == 0){
        if (exec2(path, val, stacksize) < 0) {
          printf(1, "Failed to execute %s with stacksize %s.\n", path, stacksize_str);
          }
      }
     } else if (strcmp(tokens[0], "memlim") == 0) {
      char *pid_str = tokens[1];
      char *limit_str = tokens[2];
      if (pid_str == 0 || limit_str == 0) {
        printf(1, "Usage: memlim <pid> <limit>\n");
        continue;
      }
      int pid = atoi(pid_str);
      int limit = atoi(limit_str);
      if (setmemorylimit(pid, limit) == 0) {
        printf(1, "Memory limit set successfully for process with PID %d.\n", pid);
      } else {
```

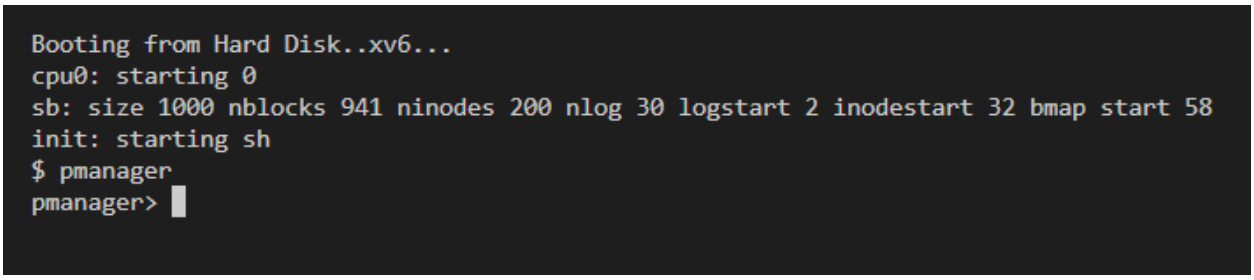
```
        printf(1, "Failed to set memory limit for process with PID %d.\n", pid);
      }
    } else if (strcmp(tokens[0], "exit") == 0) {
      break; //
    } else {
      printf(1, "Invalid command. Please try again.\n");
    }
  }

  exit();
}
```

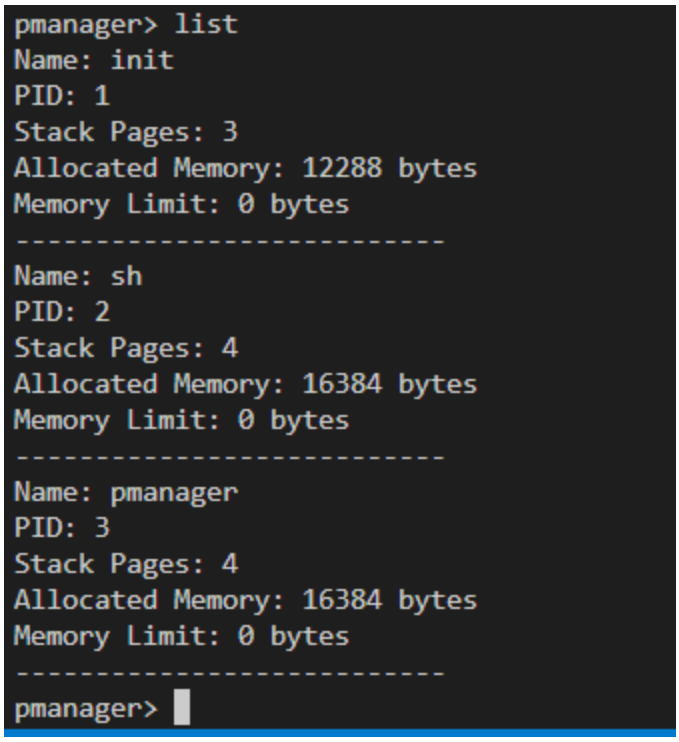
명세에 맞게 문자열을 토큰으로 나누고 해당하는 명령어를 실행하게끔 구현하였다.

---

result



- 부팅후 pmanager 실행


pmanager> list
Name: init
PID: 1
Stack Pages: 3
Allocated Memory: 12288 bytes
Memory Limit: 0 bytes

Name: sh
PID: 2
Stack Pages: 4
Allocated Memory: 16384 bytes
Memory Limit: 0 bytes

Name: pmanager
PID: 3
Stack Pages: 4
Allocated Memory: 16384 bytes
Memory Limit: 0 bytes
---
pmanager>

- list

```
pmanager> execute ls 10
pmanager> .              1 1 512
..              1 1 512
README          2 2 2286
cat             2 3 16484
echo            2 4 15336
forktest        2 5 9644
grep            2 6 18704
init            2 7 15924
kill            2 8 15368
ln              2 9 15224
ls              2 10 17852
mkdir           2 11 15464
rm              2 12 15444
sh              2 13 28088
stressfs        2 14 16356
usertests       2 15 67464
wc              2 16 17220
zombie          2 17 15036
thread_test     2 18 16396
thread_test2    2 19 33068
pmanager        2 20 19460
thread_test3    2 21 20316
thread_kill     2 22 17180
thread_exit     2 23 16148
thread_exec     2 24 16368
hello_thread    2 25 15116
console         3 26 0

pmanager>
```

- execute

```
pmanager> kill 4
Process with PID 4 killed successfully.
pmanager>
```

- kill

```
$ pmanager
pmanager> memlim 3 30000
Memory limit set successfully for process with.
pmanager>
```

- memlim

```
pmanager> exit
z$ ombie!

$
```

- exit

```
$ thread_test3
Test 1: Basic test
TThread hread 0 start
Thre1 start
ad 0 end
Parent waiting for children...
Thread 1 end
Test 1 passed

Test 2: Fork test
Thread 0 staThread 1 start
Thread 2 start
TThread 4 start
Crt
hread 3 starChild of thread 1 start
hChild oft
i thread 4 start
ChChild of thread 3ld of thread 2 start
i start
ld of thread 0 start
Child of thread 1 end
Thread 1 end
Child of thChild of thread 2 end
rChild Thread 2 end
eChild of thread 0 end
of thread 3 end
TadThreahread 3 4 end
Thread 0 end
 end
d 4 end
Test 2 passed

Test 3: Sbrk test
Thread 0 Thread 1 start
TThread 3 start
Tstart
hread hread 4 2 start
start
Test 3 passed

All tests passed!
$
```

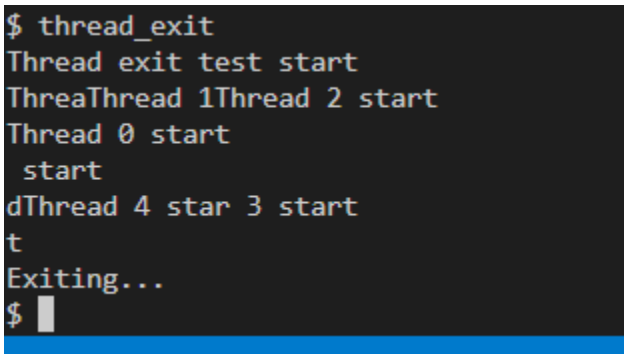
- pizza에 올라온 thread_test.c 파일을 thread_test3.c 이라는 이름으로 생성후 실행결과

```
$ thread_kill
Thread kill test start
Killing prThis code shoulThis code should be executed 5 ocess 22d be executed 5 times.
t
Thiimes.
This cs code should be execuode should be execuThis code should be executed 5 timted t.
es.
5 times.
Kill test finished
$
```

- pizza에올라온 thread_kill.c 파일 실행결과

```
$ thread_exec
Thread exec test start
TThread 1 Thread 2 start
Thread 0 start
sthread 3 start
Tart
hread 4 start
Executing...
Hello, thread!
$
```

- pizza에 올라온 thread_exec.c 파일 실행결과



- pizza에 올라온 thread_exit.c 파일 실행결과

---

trouble shooting

- *sbrk 을 실행하는도중에* `growproc()` *과* `thread_create()` *가 충돌하여 trap 14 오류가 뜨는 경우가 발생하였다.*

→ ptable의 lock처리를 `grwoproc()` 과 `thread_create()` 에 해주었고 thread_create를 할때 sz 값을 모두 동기화 해주는 과정이 생략되어있었다.

- `exit` *함수에서 exit을 호추한 process내의 모든 thread의 자원할당을 해제 해주는 과정에서 현재 cpu를 잡고있는 thread의 자원할당까지 해제해주어서* `wait` *함수에서 깨어난 부모가 깨운 thread의 자원할당을 중복해서 해제해주어서 trap 14 오류가 발생하였다.*

→ exit 함수에서 curproc의 자원할당은 해제해주지 않는 것으로 해결하였다.