ELE3021_project02_12299_2018009216

목표 : p_manager & LWP 만들기

실행방법

1. make
2. make fs.img
3. qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512

DESIGN

struct thread

- process가 가지고 있는 thread들을 정의한 구조체입니다.
- process와 thread의 차이를 고려하여 정의하였습니다.

struct proc

- 기존의 process에서 thread를 추가하여 정의하였습니다.
- 기존의 process에서 thread가 가지고 있는 부분은 제거했고, 64개의 thread를 갖고있게 수정하였습니다.
- thread배열인 tarr의 첫번째 thread가 mainthread입니다.

PROCESS MANAGER-DESIGN

int exec2(char *pathm char **argv, int stacksize);

- exec2의 경우 exec를 참조하여 만들었기 때문에 디자인이 거의 유사합니다.
- 따로 exec2.c로 파일을 구분하여 만들었습니다.
- 다른 부분은 stacksize를 조절할 수 있는 부분입니다.
- 따라서 exec에서 기존의 2개의 page를 할당하는 부분을 stacksize 만큼 할당하게 바꾸어주었습니다.

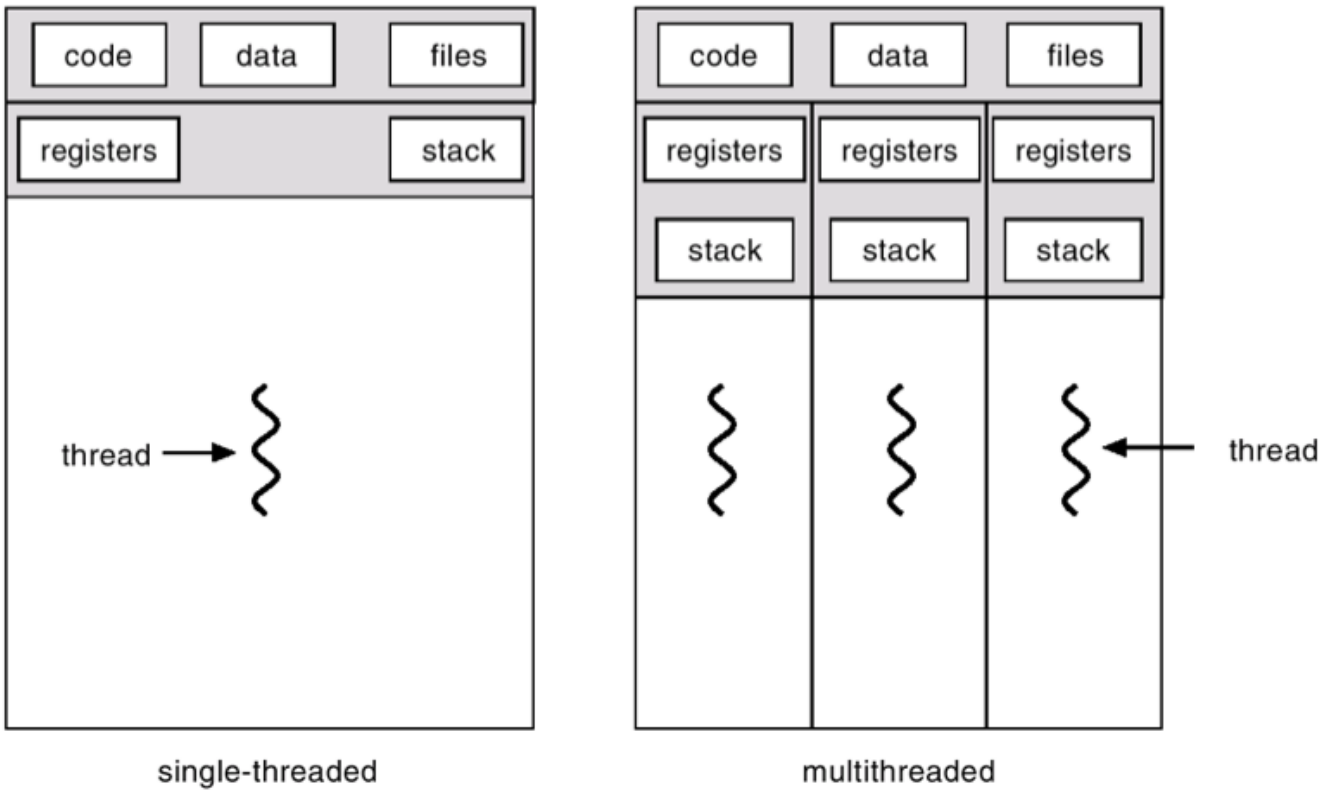
int setmemorylimit(int pid, int limit);

- wrapper function은 따로 setmemorylimit.c에 정의하였습니다.
- pid를 가지고 해당 process를 찾아야 하므로 ptable에 접근해야 합니다. 따라서 setmemorylimit 함수는 proc.c에 정의하였습니다.
- 그리고 process에 해당 limit를 가지고 있다는 것을 표시해야 하므로 proc.h에 있는 struct proc에 int limit를 추가하였습니다.
- 그리고 해당 limit를 넘는 size를 해당 프로세스가 가지면 안되므로 exec2에서 limit보다 큰 size가 큰경우에 대해 처리해주었습니다.
- 그리고 limit == 0 인경우 예외처리를 해주었습니다.

pmanager

- pmanager는 process manager입니다.
- 글자 그대로 process를 관리하는 user program입니다.
- uesr로 부터 입력을 받아 해당 명령을 수행할 수 있도로 구현하였습니다.

Single and Multithreaded Processes



LWP - DESIGN

- thread는 process와 깊은 관련이 있기 때문에 process table이 있는 proc.c에서 관련함수들을 정의하기로 하였습니다.
- 각 프로세스는 최대 64개의 thread를 가질 수 있습니다.
- LWP의 구현에 필요한 각 함수는 exec, fork, exit, wait 등 기존의 process에서 동작하는 시스템콜을 참조하여 구현하였습니다.

int thread_create(thread_t *thread, void (start_routine)(void *), void *arg);

- fork() 와 exec()를 참조하여 만들었습니다.
- 해당 함수는 새로운 쓰레드를 생성하고 start_routine을 실행하는 것입니다.
- 즉 fork()를 참조하여 새로운 쓰레드를 생성하도록 구현하였고, exec()를 참조하여 start_routine이 실행되도록 구현하였습니다.
- 구현에서 process와 thread의 차이점을 고려하면서 함수를 구현하였습니다.
- 즉, fork()의 경우 자식프로세스는 부모프로세스의 주소공간을 복제하지만, thread는 그럴 필요가 없으므로 이 부분은 구현하지 않았습니다.
- 또한 process는 모두 독립적인 주소공간을 갖지만, 쓰레드는 이와 다른게 공유하는 부분도 있으므로 이를 고려하여 구현하였습니다.

int thread_exit(void *retval);

- exit(), wait() 시스템콜을 참고하여 만들었습니다.
- 이 함수를 호출한 thread를 종료시킵니다. ⇒ 추가로 해당 process에 있는 모든 thread를 종료시켜 줍니다.
- exit()와 마찬가지로 해당 함수가 직접종료시키는 것은 아니라 zombie상태로 바꾸어줍니다.
- 이 thread의 실제 종료는 wait()에서 실행합니다.
- 또한 exit()와 다르게 void *retval가 값을 받게 했습니다.
- 따라서 wait()에서 해당 thread의 자원을 회수할 수 있게 수정해 주었습니다.

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

- wait()를 참조하여 만들었습니다.
- 이 함수는 해당 thread_t thread를 가진 thread가 종료될 때까지 기다립니다.
- 따라서 해당 thread를 찾고 종료를 기다립니다. 해당 thread가 종료되면 프로세스에서 부모가 자식의 자원을 회수하는 것처럼 해당 thread의 자원을 회수합니다.

fork

- fork()시 새로운 process를 생성하고 기존의 thread들을 모두 복사하는 것이 아닌 fork()를 호출한 thread만 복사합니다.
- 따라서 fork()를 호출한 thread를 새로운 process의 mainThread로 만듭니다.

exec

- exec()호출시 exec()를 호출한 thread를 제외한 기존의 모든 thread들을 정리하고 해당 프로세스의 mainThread에서 실행시켜주었습니다.
- 따라서 해당 스레드를 제외한 스레드들의 자원을 반환합니다.

sbrk

- 기존의 메모리의 사이즈를 결정하는 sz는 thread에서 관리하는 것이 아닌 기존의 process에서 관리하므로 수정한 부분이 없습니다.

kill ⇒ exit ⇒ wait

kill

- kill 호출시 해당프로세스의 모든 thread들도 같이 정리해주어야 합니다.
- kill 이 직접 종료시키는 것이 아닌 kill flag에 표시해두고, 나중에 trap handler에서 이 flag를 보고 exit()을 호출하여 종료시킵니다.
- 따라서 exit()를 수정하여 thread들도 같이 종료시키도록 해야합니다.

exit

- 기존과의 차이점은 thread의 유무이므로 해당 thread들도 exit될 수 있게 해당 process의 모든 thread의 상태를 좀비로 바꿔줍니다.

wait

- 기존의 process의 자원을 회수하는 것에 대해서 해당 process가 갖고 있는 자원들도 회수하도록 자꾸었습니다.

sleep

- 특정 thread만 재우는 시스템콜입니다.
- 따라서 해당하는 thread를 찾고 상태를 SLEEPING로 바꾸어 재웁니다.

pipe

- pipe는 process간의 통신을 해주는 시스템콜입니다.
- thread를 추가했지만 thread의 자원을 사용하지 않으므로 딱히 수정이 필요하지 않습니다.

SCHEDULER

- 기존의 process RR에서 thread단위로 실행하는 RR으로 변경시켜줘야 합니다.
- 따라서 process단위의 실행에서 thread단위의 실행으로 변경하였습니다. 또한 process의 상태를 확인해서 process가 RUNNABLE이 아니면 해당 thread들을 모두 실행x 입니다.

IMPLEMENT

struct thread

```
// thread 구조체
struct thread {
  char *kstack;                 // Bottom of kernel stack for this thread
  enum procstate state;         // Thread state
  int tid;                      // Thread ID
  struct trapframe *tf;         // Trap frame for current syscall //Trap frame : kernel 에서 user
  struct context *context;      // swtch() here to run process
  void *chan;                   // If non-zero, sleeping on chan //chan : 프로세스가 대기중인 이벤트를
  int killed;                   // If non-zero, have been killed //해당 프로세스가 종료되야 하는지 여부를
  //주로 signal처리에 사용, 다른 process에서 해당 process에 신호를 보내어 killed를 1로 변경하여 해당 프로세스를
  void *retval;
};
```

struct proc

```
struct proc {
  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 //해당 프로세스가 종료되야 하는지 여부를
  // 주로 signal처리에 사용, 다른 process에서 해당 process에 신호를 보내어 killed를 1로 변경하여 해당 프로세스
  struct file *ofile[NOFILE];  // Open files // 파일 디스크립터 배열을 가리키는 포인터 // 파일 디스크립터
  // ofile은 NOFILE상수로 정의된 크기르 가짐, 일반적으로 파일 디스크립터의
  // 프로세스가 파일을 열거나 생성하면, 해당 파일 디스크립터를 'ofile' 배열에 할당, 파일 참조
  // 파일을 닫을 때는 해당 파일 디스크립트를 해제, ofile 배열에서 제거
  // 이를 통해서 프로세스는 여러 개의 파일을 동시에 열고 사용할 수 있음

  struct inode *cwd;            // Current directory
  char name[16];                // Process name (debugging)

  // for pmanager
  int limit;
  int stacksize;


  // for thread
  // NTREAD(64) 64개
  struct thread tarr[NTHREAD]; //해당 프로세스가 갖고있는 threads 64개
};
```

PROCESS MANAGER

int exec2(char *pathm char **argv, int stacksize);

```
int exec2(char *path, char **argv, int stacksize)
// 기존의 exec와 다른 부분은 stacksize부분의 추가입니다.
...
...
if((sz = allocuvm(pgdir, sz, sz + (1+stacksize)*PGSIZE)) == 0)//allocuvm : 실행 파일의 세그먼트를 메
    goto bad; //sz == 0인 경우 : 메모리 할당 실패한 경우
```

```
clearpteu(pgdir, (char*)(sz - (1+stacksize)*PGSIZE));
// pgdir에서 sz - 2*PGSIZE부터 sz까지의 페이지테이블 엔트리를 초기화, 페이지테이블 엔트리는 해당 페이지를 참조하지
// 기존의 1개의 kernel stack + 1개의 user stack에서
//      1개의 kernel stack + stacksize+1개의 user stack을 할당받도록 수정하였습니다.
...
...
myproc()->stacksize = stacksize + 1;
// 이를 위해 proc.h에 struct proc에 stacksize라는 변수를 추가해주었으므로 이를 update 해줍니다.
...
```

int setmemorylimit(int pid, int limit);

```
int setmemorylimit(int pid, int limit) // memory의 limit를 정합니다.
{
  struct proc *p;
  acquire(&ptable.lock); // lock을 얻습니다.
  // 해당 pid를 가진 process를 찾습니다.
  // limit는 현재의 메모리보다 작을 수 없기 때문에 이를 처리합니다.
  // 문제가 없는 경우 이를 정상적으로 limit를 지정합니다.
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p -> pid == pid){
    if(p->sz > limit && p->limit != 0){
      cprintf("current size %d is larger than limit %d! setmemorylimit fail!\n",p->sz, limit);
      release(&ptable.lock); // 수정 한것
      return -1;
    }else{
      p->limit = limit;
    }
  }
}
if(p == &ptable.proc[NPROC]){ // bad인 경우 bad label을 지정하는 대신 이렇게 한번 해봤습니다.
  release(&ptable.lock); // 수정 한것
  cprintf("pid %d doesn't exist!\n",pid);
  return -1;
}
release(&ptable.lock); // 수정 한것
return 0;
}
```

pmanager

pmanger를 위해 추가로 정의한 int printProcess(void) 함수입니다.

```
// pmanager를 위해 정의한 함수입니다.
// 이는 현재 실행중인 프로세스들의 정보를 보여줍니다.
int printProcess(void){
  cprintf("proc.c/printProcess start\n");
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    //page num모를갯음
    if(p->pid != 0){
      cprintf("process name : %s, pid : %d, page num : %d, memory size : %d, memory limit : %d\n
    }
  }
```

```
  release(&ptable.lock);
  cprintf("proc.c/printProcess end\n");
  return 0;
}
```

user program인 pmanager.c입니다.

```
// 유저로 부터 입력을 받고 해당 명령어를 실행시켜줍니다.
// 특별한 것은 없는 프로그램입니다.
// list, kill, execute, memlim (memory limit), exit 명령어를 수행합니다.

#include "types.h" //order!
#include "stat.h"
#include "user.h"

int
main(int argc, char *argv[])
{
    //printf(1, "pmanager.c start\n");
    printf(1,"pmanager.c start\n");
    int input_num = 0;
    int int_arg1 = 0;
    char buf[50];
    int pid_for_kill;
    for(;;){
        printf(1,"\nlist\n");
        printf(1,"kill <pid>\n");
        printf(1,"execute <path> <stacksize>\n");
        printf(1,"memlim <pid> <limit>\n");
        printf(1,"exit\n\n");
        input_num = read(0, buf,sizeof(buf));
        buf[input_num-1] = '\0';
        if(strcmp(buf, "list") == 0){
            printf(1,"\nlist/your input : %s\n",buf);
            printProcess();
        }else if((buf[0]=='k') & (buf[1]=='i') & (buf[2]=='l') & (buf[3]=='l')){ //이거 뒤부터 인지
            printf(1,"\nkill/your input : %s\n",buf);
            int temp;
            pid_for_kill = 0;
            for(int i = 0;i<46;i++){
                if(buf[5+i] == '\0'){
                    break;
                }
                temp = buf[5+i] - 48;
                pid_for_kill = pid_for_kill * 10 + temp;
            }
            //printf(1,"sum : %d\n", pid_for_kill);
            kill(pid_for_kill);
        }else if((buf[0]=='e') &(buf[1]=='x') &(buf[2]=='e') &(buf[3]=='c') &(buf[4]=='u') &(buf
            printf(1,"\nexecute/your input : %s\n",buf);
            int pid;
            pid = fork();
            if(pid == -1){
                printf(1,"pmanager.c/main execute fail!\n");
            }else if(pid == 0){ // 자식프로세스인 경우
                int pid1;
                pid1 = fork();
```

```
                if(pid1 == 0){ // 자식의 자식인 경우
                    char arg1[50];
                    int stacksize = 0, i, temp = 0;
                    for(i = 0;i<50;i++){
                        if(buf[8+i] == ' '){
                            break;
                        }
                        arg1[i] = buf[5+i];
                    }
                    i++;
                    for(;i<50; i++){
                        if(buf[8+i] == '\0'){
                            break;
                        }
                        temp = buf[8+i] - 48;
                        stacksize = stacksize * 10 + temp;
                    }
                    printf(1,"pmanager.c/main/execute : stacksize : %d\n",stacksize);
                        char *argvExec2[] = {arg1,0};
                        exec2(arg1, argvExec2, stacksize);
                    }
                    wait(); // 자식의 부모프로세스는 자식의 자식프로세스를 기다림
                }
            }else if((buf[0]=='m') &(buf[1]=='e') &(buf[2]=='m') &(buf[3]=='l') &(buf[4]=='i') &(buf
                printf(1,"\nmemlim/your input : %s\n",buf);
                int i;
                int pid = 0, limit = 0, temp = 0;
                for(i = 0;; i++){
                    if(buf[i + 7] == ' '){
                        break;
                    }
                    temp = buf[i + 7] - 48;
                    pid = 10 * pid + temp;
                }
                for(;;i++){
                    if(buf[i+7] == '\0'){
                        break;
                    }
                    temp = buf[i + 7] -48;
                    limit = 10 * limit + temp;
                }
                if(setmemorylimit(pid, limit) == -1)
                    printf(1,"pmanager.c/memlim fail!\n");
                printf(1,"pamanger.c/memlim success!");
            }else if(strcmp(buf, "exit") == 0){
                printf(1,"\nexit/your input : %s\n",buf);
                break;
            }else{
                printf(1,"non-exist instruction! re - input!\n");
                continue;
            }
        }
        exit();
        printf(1,"%d\n",input_num+int_arg1);
    }
```

LWP

int thread_create(thread_t *thread, void *(start_routine)(void *), void *arg);

```
int thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg){
  thread_t tid;
  //int i;
  struct proc *curproc = myproc();
  struct thread *nt; // new thread
  char *sp;
  uint sz;
  acquire(&ptable.lock);

  for(nt = curproc->tarr; nt < &curproc->tarr[NTHREAD]; nt++){
    if(nt->state == UNUSED){
      goto found;
    }
  }
  release(&ptable.lock);
  cprintf("proc.c/thread_create no UNUSED thread space\n");
  return -1;

found:

  tid = nexttid++;
  *thread = tid; // thread는 왜 따로 받는 거지? thread구조체의 tid에 저장되어 있는데...
  nt->tid = tid;
  nt->state = EMBRYO;

  if((nt->kstack = kalloc()) == 0){
    cprintf("proc.c/thread_create kalloc fail!\n");
    goto bad;
  }
  sp = nt->kstack + KSTACKSIZE;

  sp -= sizeof *nt->tf;
  nt->tf = (struct trapframe*)sp;
  *(nt->tf) = *(curproc->curthread->tf); // 이거 수정 필요 // 이거 왜 하는 거지?

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret; // 스택에 현재 실행 중인 프로세스가 복귀할 함수 주소 저장

  sp -= sizeof(*nt->context); // context를 저장할 공간 확보
  nt->context = (struct context*)sp; //
  memset(nt->context, 0, sizeof(*nt->context));
  nt->context->eip = (uint)forkret; //

  // Allocate process
  // 현재실행중인 프로세스에서 thread의 자리가 남아있으면 tid를 할당, kernel stack을 할당해줌


  //-----여기까지 kernel stack fork 참조해서 함---
    ////////////////////////////////////////
  //-----여기부터 exec 참조

  //struct proc* p;
```

```
  //p = myproc();
  sz = PGROUNDUP(curproc->sz); // page 크기에 맞춰서 올림해줌

  //cprintf("thread_create test 1.0\n");
  //page table, old size, new size
  if((sz = allocuvm(curproc->pgdir, sz, sz + PGSIZE)) == 0){ // 할당받은 size return
    cprintf("proc.c/thread_create allocuvm fail!\n");
    goto bad;
  }
  //cprintf("thread_create test 1.1\n");
  sp = (char*)sz;
  myproc()->sz = sz; // 새로운 stack을 1 PGSIZE만큼 추가 할당받았으므로 update
  //int thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg)
  //cprintf("thread_create test 1.3\n");
  // copyout을 사용하지 않기 때문에 직접 스택을 조절해준다.
  sp -= 4;
  *(uint *)sp = (uint)arg;
  sp -= 4;
  *(uint *)sp = 0xffffffff;
  //cprintf("thread_create test 1.4\n");

  nt->tf->eip = (uint)start_routine; // start_routine을 저장 // eip : instruction pointer : pc re
  nt->tf->esp = (uint)sp; // esp : stack pointer
  nt->state = RUNNABLE; //
  nt->retval = arg;
  //cprintf("proc.c/thread_create arg : %d\n",arg);
  //cprintf("proc.c/thread_create retval : %d\n",nt->retval);

  //cprintf("thread_create test 1.5\n");
  release(&ptable.lock);

  return 0; // 정상종료

bad:

  //thread는 process와 다르게 부모의 정보를 복사할필요가 있나??? => 없는거 아님?
  cprintf("proc.c/thread_create bad label\n");
  nt->state = UNUSED;
  nt->tid = 0;
  nt->kstack = 0;

  release(&ptable.lock);

  return -1; // error
}
```

int thread_exit(void *retval);

```
void thread_exit(void *retval){ // xv6에서 스레드는 병렬적으로 실행 x
  // 현재 이걸 호출한 스레드를 어떻게 찾지?
  struct proc *curproc = myproc();
  struct thread *t = curproc->curthread; // 이건 어떻게 찾음? // 현재실행중인 스레드를 어떻게 알지?

  acquire(&ptable.lock);
  wakeup1((void *)t->tid);
```

```
  t->retval = retval;
  t->state = ZOMBIE;

  sched();
  panic("zombie thread exit\n");
}
```

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

```
int thread_join(thread_t thread, void **retval){ // wait과 유사
  struct thread* t;
  struct proc* p;

  //cprintf("proc.c/thread_join before acquire\n");

  acquire(&ptable.lock);
  for(p = ptable.proc; p != &ptable.proc[NPROC]; p++) {
    if(p->state != RUNNABLE)
      continue;
    for(t = p->tarr; t < &p->tarr[NTHREAD]; t++){
      if(thread == t->tid){
        //cprintf("proc.c/thread_join match\n");
        goto MATCH;
      }
    }
  }
  release(&ptable.lock);
  cprintf("proc.c/thread_join can't match tid\n");
  return -1; // error!

MATCH:
  //cprintf("proc.c/thread_join\n");
  for(;;){ // 여기가 문제!!!!!!!1
    if(t->state != ZOMBIE){
      sleep((void *)thread, &ptable.lock);
    }

    t->tid = 0;
    kfree(t->kstack);
    t->kstack = 0;
    t->state = UNUSED;
    *retval = t->retval;
    t->retval = 0;

    release(&ptable.lock);
    return 0; // 정상종료
  }
}
```

fork

```
struct thread *t; // thread도 건드려야 하므로 필요
...

// 기존의 process가 갖고있던 자원중 thread가 대신 갖고있는 자원들
```

```
// copyuvm 실패시 해단 process의 main thread가 갖고 있는 자원할당해제
if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    cprintf("proc.c/fork copyuvm fail!\n");
    kfree(t->kstack);
    t->kstack = 0;
    t->state = UNUSED;
    np->state = UNUSED;
    return -1;
}
...
// tf는 thread가 갖고있으므로
np->tarr->tf->eax = 0; // 자식이 return
...
// thread의 상태도 변경해줌
t->state = RUNNABLE;

//allocproc // 기존과 다른게 mainThread의 자원도 할당받아야 하므로 수정이 필요하다.
static struct proc* allocproc(void)
...
// main thread의 상태도 바꿔주고 tid를 할당해줌
t = p->tarr;
t->state = EMBRYO;
t->tid = nexttid++;
...
// kernel stack은 기존과 다르게 thread에서 갖고있으므로 main thread에 kernel stack할당
if((t->kstack = kalloc()) == 0){
    p->state = UNUSED;
    t->state = UNUSED;
    cprintf("proc.c/allocproc kstack fail\n");
    return 0;
}
sp = t->kstack + KSTACKSIZE;

// trap frame도 기존과 다르게 thread가 갖고 있으므로...
// Leave room for trap frame.
sp -= sizeof *t->tf;
t->tf = (struct trapframe*)sp;
...
// 실행단위가 스레드이므로 문맥교환도 스레드단위로 일어남 따라서 thread에 context할당
sp -= sizeof *t->context; // context를 저장할 공간 확보
t->context = (struct context*)sp; // context의 위치를 stack pointer의 위치로 설정
memset(p->tarr->context, 0, sizeof *t->context); // context구조체를 0으로 초기화
t->context->eip = (uint)forkret; // ret : 어셈블리어 명령어,
```

exec

```
// 기존과 다른점은 exec()를 실행한 thread를 main thread로 바꾸어 주고 나머지 thread들의 자원을 회수하는 것입니
int
exec(char *path, char **argv)
...
// 실행한 thread를 main thread로 바꿔주기
  *curproc->tarr = *curproc->curthread;
  if(curproc->curthread != curproc->tarr){
    cprintf("exec.c curproc->curthread != curproc->tarr\n");
    curproc->curthread->kstack = 0;
  }
```

```
t = curproc->tarr; // 현재 thread의 main thread
t++; // main thread의 경우 자원할당을 해제하면 안되므로 다음 thread로 넘어감
for(; t < &curproc->tarr[NTHREAD]; t++){ // 모든 thread를 돌며 자원을 회수함
    if(t->kstack){
      cprintf("exec.c before kfree\n");
    kfree(t->kstack);
    cprintf("exec.c after kfree\n");
  }
  t->kstack = 0;
  t->tid = 0;
  t->retval = 0;
  t->state = UNUSED;
}
// 호출한 스레드에서 main thread로 바꿔줘야 하므로
curproc->tarr->tf->eip = elf.entry;
curproc->tarr->tf->esp = sp;
curproc->curthread = curproc->tarr;

...
```

sbrk

```
// 수정한 부분 X
// struct proc가 기존의 system call을 이용할 수 있게 만들었으므로 해당 thread를 건드는 부분이 아니라면 코드의
```

kill() → exit() → wait()

- **kill을 호출하면 kill flag를 수정해주고 나중에 trap handler가 해당 flag를 보고 exit()을 호출하므로 exit()을 수정해줘야함**
- **exit()가 실제로 종료하는 것이 아니라 해당 프로세스를 ZOMBIE상태로 만들어 주고 실제의 자원회수와 종료는 wait()에서 처리해주므로 wait()를 수정해주어야함!**

```
// kill()은 수정 X
// struct proc가 기존의 system call을 이용할 수 있게 만들었으므로 해당 thread를 건드는 부분이 아니라면 코드의

//------------------------------------------------------------------
// exit()
void exit(void){
...

// thread들의 상태도 다 zombie로 바꿔줘야 한다.
  for(t = curproc->tarr; t < &curproc->tarr[NTHREAD]; t++){
    if(t->state != UNUSED){
      t->state = ZOMBIE;
    }
  }

...
}

//------------------------------------------------------------------
int wait(void){
...
            // 해당 프로세스를 찾은 경우 기존과 다르게 thread들의 자원도 해제해줌
```

```
    if(p->state == ZOMBIE){
// Found one.
for(t = p->tarr; t < &p->tarr[NTHREAD]; t++){ // 갖고있는 모든 thread들도 없애야함
  if(t->kstack){
    kfree(t->kstack);
  }
  t->kstack = 0;
  t->tid = 0;
  t->state = UNUSED;
}
pid = p->pid;
freevm(p->pgdir);
p->pid = 0;
p->parent = 0;
p->name[0] = 0;
p->killed = 0;
p->state = UNUSED;

...
}
```

sleep

```
// 기존과 다른점은 해당 프로세스가 아닌 해당 스레드를 재워주는 것입니다.
// 따라서, 해당 부분만 수정하면 됩니다!

void
sleep(void *chan, struct spinlock *lk){
...

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

...
}
```

pipe

```
// 수정 x
// thread의 자원을 사용하지 않으므로 수정이 필요없다!
```

SCHEDULER

```
// 기존의 process단위에서 thread단위로 실행할 수 있게 수정했습니다.
// 해당 프로세스에서 thread들을 순차적으로 돌며 round robin방식으로 실행합니다.

void scheduler(void){
...
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      c->proc = p;
      // Switch to chosen process.  It is the process's job
```

```
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      for(t = p->tarr; t < &p->tarr[NTHREAD]; t++){// process thread둘다 RUNNING으로? 언제 runnabl
        if(t->state == RUNNABLE){
          p->curthread = t;
          p->curtid = t->tid;
          switchuvm(p);
          t->state = RUNNING;
          //p->state = RUNNING;
          swtch(&(c->scheduler), t->context);
          switchkvm();
        }
      }
      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
...
}
```

SCHED

```
// 기존과 달리 문맥교환을 process가 아닌 thread가 하게 수정하였습니다.
void sched(void){
    ...
    struct thread *t = p->curthread;
    ...
    swtch(&t->context, mycpu()->scheduler);
    ...
}
```

RESULT // 컴파일 및 실행과정, 실행결과 , 동작에 대한 설명

```
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
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15684
echo           2 4 14560
forktest       2 5 9000
grep           2 6 18516
init           2 7 15180
kill           2 8 14644
ln             2 9 14544
ls             2 10 17112
mkdir          2 11 14668
rm             2 12 14652
sh             2 13 28700
stressfs       2 14 15576
wc             2 15 16096
zombie         2 16 14220
prac_myuserapp 2 17 14592
myapp          2 18 14572
pmanager       2 19 17836
hello_thread   2 20 14364
thread_exec    2 21 15636
thread_exit    2 22 15368
thread_kill    2 23 16520
thread_test    2 24 19328
console        3 25 0
$
```

PROCESS MANAGER

pmanager실행시

```
$ pmanager
pmanager.c start

list
kill <pid>
execute <path> <stacksize>
memlim <pid> <limit>
exit
```

list실행시

```
list

list/your input : list
proc.c/printProcess start
process name : init, pid : 1, page num : 0, memory size : 12288, memory limit : 0
process name : sh, pid : 2, page num : 0, memory size : 16384, memory limit : 0
process name : pmanager, pid : 4, page num : 0, memory size : 12288, memory limit : 0
proc.c/printProcess end

list
kill <pid>
execute <path> <stacksize>
memlim <pid> <limit>
exit
```

kill 4실행시, 위의 list에서 pid : 4 = pmanager이므로 pmanager가 종료된다.

```
kill 4

kill/your input : kill 4
$
```

execute pmanager시 pmanager가 추가로 생성되었다.

```
execute pmanager

execute/your input : execute pmanager

list
kill <pid>
execute <path> <stacksize>
memlim <pid> <limit>
exit

pmanager.c/main/execute : stacksize : 2
proj2_exec2.c/sys_exec2 1.0
proj2_exec2.c/exec2 start
path : te pmanager, stacksize : 2
exec2: fail

list
kill <pid>
execute <path> <stacksize>
memlim <pid> <limit>
exit

list

list/your input : list
proc.c/printProcess start
process name : init, pid : 1, page num : 0, memory size : 12288, memory limit : 0
process name : sh, pid : 2, page num : 0, memory size : 16384, memory limit : 0
process name : pmanager, pid : 6, page num : 0, memory size : 12288, memory limit : 0
process name : pmanager, pid : 7, page num : 0, memory size : 12288, memory limit : 0
process name : pmanager, pid : 8, page num : 0, memory size : 12288, memory limit : 0
process name : pmanager, pid : 9, page num : 0, memory size : 12288, memory limit : 0
process name : pmanager, pid : 10, page num : 0, memory size : 12288, memory limit : 0
proc.c/printProcess end
```

exit시 pmanager가 종료된다.

```
list
kill <pid>
execute <path> <stacksize>
memlim <pid> <limit>
exit

exit

exit/your input : exit
$
```

LWP

hello_thread실행시

```
$ hello_thread
Hello, thread!
$
```

thread_exec실행시

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

thread_exit실행시

```
$ thread_exit
Thread exit test start
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Exiting...
$
```

thread_kill 실행시

```
$ thread_kill
Thread kill test start
thread_kill parent
thread_kill pid == 0
Killing process 6
This code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
Kill test finished
$
```

thread_test실행시

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

Test 2: Fork test
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Thread 0 returned 167772161, but expected 0
Test failed!
Thread 0 returned 167772161, but expected 0
Test failed!
Thread 0 returned 167772161, but expected 0
Test failed!
Thread 0 returned 167772161, but expected 0
Test failed!
Thread 0 returned 167772161, but expected 0
Test failed!
Thread 0 returned 167772161, but expected 0
Test failed!
Thread 0 end
Thread 1 end
Thread 2 end
Thread 3 end
Thread 4 end
Test 2 passed
```

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

All tests passed!
$
```

TROUBLE SHOOTING

- 기존의 프로젝트에서 이어서 하지 않고 새로운 프로젝트 파일로 시작할려고 했는데 문제가 많았고, 오류를 잡는데 많은 시간을 들였지만 오류를 잡지 못했습니다.
- 그래서, 오류가 나올 때마다 고민하면서 오류가 잡히지 않으면 새로 프로그램을 작성하였습니다.
- 기존의 process와 다르게 thread들을 어떻게 병렬적으로 실행시킬까에 대한 고민이 많았습니다. ⇒ xv6는 이론과 다르게 병렬적으로 thread를 실행시키지 않는다는 것을 알게됬고 기존과 같이 RR(round robin)방식으로 스케줄러를 만들었습니다.