ELE3021_project02_12298_2019042497

Design

1. proc 구조체가 추가로 가지고 있어야 하는 값

   메모리의 최대 제한 정보인 **memory limit**, 스택용 페이지의 개수인 **stack size**, thread의 ID인 **thread ID**, thread_create를 **호출한 proc**의 정보, 자신의 **stack 시작 위치**, 스레드를 종료한 후 join 함수에서 받아갈 값인 **retval**가 필요하다고 생각했습니다. 여기서 반드시 초기화 되어야 할 thread ID와 호출한 proc의 정보는 allocproc() 함수 안에서 초기화를 각각 해줄 것입니다.

2. Process with various stack size

   인자로 받는 stacksize 개수만큼의 stack용 페이지를 할당해야 합니다. 따라서, 기존 함수를 가져와서 stack용 페이지를 할당하는 부분만 수정을 해줄 것입니다. 여기에 추가로 stacksize의 크기는 1 이상 100 이하의 정수여야 하기 때문에 이 범위 체크도 진행해줄 것입니다. 그리고 나중에 출력하기 편하게 하기 위해 proc 구조체에 만들어 놓은 stack_size 변수에 설정한 stacksize의 값을 저장해줄 것입니다.

3. Process memory limitation

   할당 받을 수 있는 메모리의 최대치를 제한하는 함수를 구현해야 합니다. 따라서, ptable을 순회하면서 맞는 pid를 찾으면 위에서 말했듯 proc 구조체의 memory limit 변수에 해당 값을 넣어줄 것입니다. 그리고 다양한 예외를 처리해주고, 실제 추가적인 메모리를 할당 받을 때 memory limit을 넘는 경우 할당 받지 않아야 하기 때문에 이는 해당 함수가 아닌 실제 추가적인 메모리를 할당 받는 부분인 sbrk(growproc) 함수에 처리를 해줄 것입니다.

4. Process manager

   sh.c에서 console의 입력을 받아오는 방식을 가져와서 사용해줄 것입니다. 개행을 기준으로 입력을 buf에 한 줄 받는데, 이를 공백(' ') 또는 0(NULL 값)을 기준으로 나누었을 때 첫 번째 입력 값을 명령어로 보고 구분을 해줄 것입니다. list에서는 해당 정보들을 출력할 수 있는 함수를 proc.c에 만들어서 이를 사용할 것입니다. kill에서는 pid 값이 들어 와있을 것이기 때문에 이를 atoi()로 int형으로 바꿔서 kill 함수에 pid 값을 인자로 넘겨줄 것입니다. execute에서는 path와 stacksize를 인자로 받기 때문에 공백(' ')을 기준으로 또 나눠주고, stacksize는 atoi()로 int형으로 바꿔서 이전에 만든 exec2()에 인자로 넘겨줄 것입니다. memlim에서는 pid와 limit 값이 들어오기 때문에 공백(' ')을 기준으로 또 나눠주고, 둘 다 atoi()로 int형으로 바꿔서 setmemorylimit() 함수에 인자로 넘겨줄 것입니다. 마지막으로 exit에서는 exit()을 바로 해줄 것입니다. 이 외의 명령은 실행되지 않기 때문에 굳이 다른 예외처리를 하지 않을 것입니다.

5. Pthread in xv6

thread_create() 함수는 fork()와 exec()(중에서도 stack 수정 부분)에서 일부 가져와서 수정하여 구현할 것입니다. 마찬가지로 thread_exit() 함수는 exit()에서 일부 가져와서 수정할 것이고, thread_join() 함수는 wait()에서 일부 가져와서 수정하여 구현할 것입니다. 이외에도 기존 코드에서 수정할 점은 thread의 정보가 추가되었기 때문에 각종 초기화를 하는 부분에서 필요하다면 추가로 넣어줄 것입니다. 또한, exec()은 기존 process의 모든 thread들이 정리되어야 하기 때문에 이 부분도 pid가 같은 것들을 다 정리하는 방식으로 구현할 것입니다. sbrk에서 요구한 대로 메모리를 할당해주는 공간이 서로 겹치면 안 되기 때문에 sz를 수정하게 된다면 같은 pid를 가진 thread들의 sz를 다 같이 갱신해주는 것이 가장 중요하다고 생각합니다. 저는 여기서 계속 커지는 방향으로 sz를 증가시켜 할당해줄 것입니다. 또한, 하나 이상의 thread가 kill 되면 process 내의 모든 thread가 종료되어야 하기 때문에 exit()에서 pid가 같은 thread의 자원들을 다 정리하고 회수하도록 해 조건을 충족시켜줄 것입니다.

Implement

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int mem_limit;               // memory limit
  int stack_size;              // stacksize
  int tid;                     // thread ID
  struct proc *called;         // thread_create를 호출한 proc
  uint stack_start;            // 자신의 stack 시작 위치
  void *retval;                // 스레드를 종료한 후 join 함수에서 받아갈 값
};
```

먼저, proc.h 안의 proc 구조체 안에 새로운 값 들을 추가해주었습니다. mem_limit는 memory limit, stack_size는 stacksize, tid는 thread ID, called는 thread_create를 호출한 proc, stack_start는 자신의 stack 시작 위치, retval는 스레드를 종료한 후 join 함수에서 받아갈 값입니다.

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->mem_limit = 0;
  p->stack_size = 0;
  p->tid = 0;
  p->called = p;
  p->stack_start = 0;
  p->retval = 0;
```

각 값들의 초기화는 proc.c 안의 allocproc(void)의 해당 위치에서 해주었습니다.

다음은 exec2() 구현입니다. 대부분 exec()를 가져와서 사용하였기 때문에 여기서는 기존 exec()과 달라진 부분과 추가된 부분을 설명하겠습니다.

```
if(stacksize < 1 || stacksize > 100) // stacksize가 1 이상 100 이하의 정수가 아니면
  goto bad;                          // 오류로 취급
```

stacksize는 1 이상 100 이하의 정수여야 하기 때문에 해당 조건을 넣어주었습니다.

```
// Allocate (stacksize+1) pages at the next page boundary.
// Make the first inaccessible.  Use the rest as the user stack.
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + (stacksize+1)*PGSIZE)) == 0) // stacksize + 1만큼의 가상 메모리 공간을 할당
  goto bad;
clearpteu(pgdir, (char*)(sz - (stacksize+1)*PGSIZE));         // 가드용 페이지를 설정
sp = sz;
```

그리고 메모리를 할당하는 allocuvm()에서 기존에 stacksize가 1이었을 때 2를 곱해주어 가드용 페이지 공간까지 생각을 해줬기 때문에 여기서는 stacksize+1만큼의 공간을 할당 받고, 가드용 페이지까지 맞춰서 가상 주소상 스택 페이지들의 바로 아래에 있게 설정해줬습니다.

```
curproc->stack_size = stacksize;       // stack용 page의 개수
```

마지막으로 구조체의 stack_size 값에 stacksize 값을 넣어주었습니다.

다음은 해당 함수를 구현하며 exec()에도 바꿔준 부분입니다.

```
curproc->stack_size = 1;       // stack용 page의 개수
```

기존 exec()에도 stack_size를 1로 넣어주어 혹시나 발생할 수 있는 혼선을 방지해주었습니다.

다음은 sysfile.c 에 만들어 놓은 sys_exec2()입니다.

```
if(argint(2, &stacksize) < 0) // stacksize 인자가 int로 들어오지 않았다면
  return -1;
return exec2(path, argv, stacksize);
```

마찬가지로 sys_exec()에서 앞부분은 똑같이 구현하고, 뒤에 해당 부분만 추가해 구현했습니다.

다음은 setmemorylimit() 구현입니다.

```
// 특정 프로세스에 대해 할당받을 수 있는 메모리의 최대치를 제한하는 함수
int
setmemorylimit(int pid, int limit)
{
  if (limit < 0) // limit의 값이 0보다 작으면
    return -1;
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ // ptable 처음부터 끝까지 순회
    if(p->pid == pid){ // pid가 같으면
      if(limit == 0){ // limit가 0이면
        p->mem_limit = 0; // mem_limit에 0을 넣음
        release(&ptable.lock);
        return 0;
      }
      if(limit >= p->sz){ // 기존에 할당 받은 메모리보다 limit가 크면
        p->mem_limit = limit; // mem_limit에 limit를 넣음
        release(&ptable.lock);
        return 0;
      }
    }
  }
  release(&ptable.lock);
  return -1;
}
```

limit의 값은 0 이상의 정수이기 때문에 음수이면 오류를 return 하게 해주었습니다. 이후 ptable을 순회하다가 pid가 같은 p를 찾으면 limit를 설정해줍니다. 이때 limit가 0이면 mem_limit에 그냥 0을 넣어주고 0을 반환하였고, 기존 할당 받은 메모리보다 limit가 작으면 for문을 나와서 -1이 return 되어야 하기 때문에 크거나 같으면 mem_limit를 limit로 설정해주고 0을 반환해주었습니다. 마찬가지로 pid가 존재하지 않으면 자동으로 for문을 벗어나 -1을 반환하게 됩니다.

setmemorylimit()의 syscall 함수입니다.

```
// setmemorylimit 함수의 system call 함수
int
sys_setmemorylimit(void)
{
  int pid, limit;
  if(argint(0, &pid) < 0 || argint(1, &limit) < 0) // pid, limit 인자가 int로 들어오지 않았다면
    return -1;
  return setmemorylimit(pid, limit); // 인자를 넣어 setmemorylimit 함수 실행
}
```

인자로 pid, limit가 들어오는데 둘 다 int가 맞는지 확인하고 인자를 넣어 실행되게 하였습니다.

다음은 pmanager 구현입니다.

sh.c에서 console의 입력을 받아오는 방식을 가져왔습니다. 이후, pmanager 명세에 맞게 세세한 부분을 수정해주었습니다. 같은 부분의 설명을 제외하겠습니다.

```
// Read and run input commands.
while(getcmd(buf, sizeof(buf)) >= 0){
  buf[strlen(buf)-1] = 0;  // chop \n
  int i;
  for(i = 0; i < 100; i++){
    if (buf[i] == ' ' || buf[i] == 0){ // 공백이나 0(NULL)을 만나면
      temp[i] = 0;                     // 0으로 temp의 마지막을 닫아줌
      break;
    }
    temp[i] = buf[i];                  // temp에 buf 값 복사
  }
  i++;                                 // 한 칸 뒤로 이동
  int n = i;                           // 현재 위치 기억
```

buf에 개행을 기준으로 사용자의 입력이 들어오게 됩니다. 여기서는 공백이나 0(NULL) 기준으로 입력을 나눴을 때 첫 번째 입력을 temp에 담습니다. 이는 유저가 실행하고자 하는 명령어를 의미합니다.

```
if(!strcmp(temp, "list")){ // 만약 list 명령이라면
  printlist();             // 정보 출력
}
```

가장 먼저, list 명령이라면 만들어 놓은 system call인 printlist() 함수를 실행하도록 하였습니다.

```
// 현재 실행 중인 프로세스들의 정보를 출력하는 함수
void
printlist()
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ // ptable 처음부터 끝까지 순회
    if(p->state == RUNNABLE || p->state == RUNNING || p->state == SLEEPING){ // 현재 실행 중인 process라면
      cprintf("name: %s, pid: %d, pages for stack: %d\n", p->name, p->pid, p->stack_size);
      if (p->mem_limit == 0)                          // mem_limit이 0이면
        cprintf("memory size: %d, memory limit: unlimited\n", p->sz);
      else                                            // mem_limit이 0이 아니면
        cprintf("memory size: %d, memory limit: %d\n", p->sz, p->mem_limit);
    }
  }
  release(&ptable.lock);
}
```

printlist() 함수는 현재 실행 중인 process의 정보를 출력합니다. 실행 중인 process는 RUNNABLE, RUNNING, SLEEPING 한 것들만 해당된다고 파악하였습니다. 또한, mem_limit이 0이면 제한이 없음을 의미하기 때문에 이를 나눠서 출력해줬습니다.

printlist() 함수는 system call로 실행되어야 합니다.

```
// printlist 함수의 system call 함수
int
sys_printlist(void)
{
  printlist();
  return 0;
}
```

따라서 위와 같이 printlist()의 system call 함수를 만들어주었습니다. 인자가 없기 때문에 그냥 실행해주었습니다.

```
else if(!strcmp(temp, "kill")){ // 만약 kill 명령이라면
  char bufpid[20];
  int pid;
  for(; i < 100; i++){
    if (buf[i] == 0){           // 0(NULL)을 만나면
      bufpid[i - n] = 0;        // 0으로 bufpid의 마지막을 닫아줌
      break;
    }
    bufpid[i - n] = buf[i];     // bufpid에 buf 값 복사
  }
  pid = atoi(bufpid);           // bufpid에 있는 값을 int로 변환해 pid에 담음
  if(!kill(pid)){               // kill(pid)를 실행했을 때 0이 반환되면
    printf(2, "kill %d successed\n", pid);
  }
  else{
    printf(2, "kill %d failed\n", pid);
  }
}
```

그 다음, kill 명령이라면 인자로 뒤에 pid를 받아와야 합니다. 따라서 buf에서 명령어 다음 부분부터 끝날 때까지 pid를 bufpid로 받아왔고, 이를 atoi()를 이용해 int로 바꿔서 pid에 넣었습니다. 이때 pid의 크기는 0 이상 10억 이하의 정수이므로 길이가 10이 최대이기 때문에 넉넉하게 bufpid 배열의 크기를 20으로 잡았습니다.

이후, kill(pid)를 실행해 pid를 가진 process를 kill 했고, 성공 여부 출력을 위해 반환 값으로 0이면 성공, 0이 아니면 실패를 출력하도록 했습니다.

```
else if(!strcmp(temp, "execute")){            // 만약 execute 명령이라면
  char* argv[10] = { 0, };
  char bufpath[70];
  char bufsize[20];
  int stacksize;
  for(; i < 100; i++){
    if (buf[i] == ' '){                       // 공백을 만나면
      bufpath[i - n] = 0;                     // 0으로 bufpath의 마지막을 닫아줌
      break;
    }
    bufpath[i - n] = buf[i];                  // bufpath에 buf 값 복사
  }
  strcpy(argv[0], bufpath);                   // argv[0]에 bufpath 값 복사
  i++;                                        // 한 칸 뒤로 이동
  n = i;                                      // 현재 위치 기억
  for(; i < 100; i++){
    if (buf[i] == 0){                         // 0(NULL)을 만나면
      bufsize[i - n] = 0;                     // 0으로 bufsize의 마지막을 닫아줌
      break;
    }
    bufsize[i - n] = buf[i];                  // bufsize에 buf 값 복사
  }
  stacksize = atoi(bufsize);                  // bufsize에 있는 값을 int로 변환해 stacksize에 담음
  if(fork() == 0){                            // fork() 했을 때 자식 process이면
    if(exec2(bufpath, argv, stacksize) == -1) // 인자를 알맞게 넣고 exec2() 실행했을 때 -1이 반환되면
        printf(2, "execute failed\n");
    break;
  }
}
```

그 다음, execute 명령이라면 인자로 뒤에 path와 stacksize를 받아와야 합니다. 따라서 buf에서 명령어 다음 부분부터 공백을 만날 때까지 path를 bufpath로 받아왔고, 이를 exec2()에서 사용하기 위해 argv[0]에 넣어주었습니다. 이후, 그 다음 부분부터 끝날 때까지 stacksize를 bufsize로 받아왔고, atoi()를 이용해 int로 바꿔서 stacksize에 넣었습니다. 이때 path의 길이는 50을 넘지 않으므로 넉넉하게 bufpath 배열의 크기를 70으로 잡아주었고, stacksize의 크기는 0 이상 10억 이하의 정수이므로 길이가 10이 최대이기 때문에 넉넉하게 bufsize 배열의 크기를 20으로 잡았습니다.

이후, fork()를 실행해 자식 process이면 exec2()에 인자를 넣고 실행하도록 했습니다. 이때, -1이 반환되면 실패한 것이므로 메시지를 출력하도록 했습니다. 그리고 부모 process는 계속 pmanager를 실행해야 하므로 아무 조치도 안 취해주었습니다.

```
else if(!strcmp(temp, "exit")){ // 만약 exit 명령이라면
  exit();                       // exit() 호출
}
```

만약 exit 명령이라면 바로 exit()을 호출해 pmanager가 종료되도록 해주었습니다.

```
else if(!strcmp(temp, "memlim")){  // 만약 memlim 명령이라면
  char bufpid[20], buflim[20];
  int pid, limit;
  for(; i < 100; i++){
    if (buf[i] == ' '){            // 공백을 만나면
      bufpid[i - n] = 0;           // 0으로 bufpid의 마지막을 닫아줌
      break;
    }
    bufpid[i - n] = buf[i];        // bufpid에 buf 값 복사
  }
  pid = atoi(bufpid);              // bufpid에 있는 값을 int로 변환해 pid에 담음
  i++;                             // 한 칸 뒤로 이동
  n = i;                           // 현재 위치 기억
  for(; i < 100; i++){
    if (buf[i] == 0){              // 0(NULL)을 만나면
      buflim[i - n] = 0;           // 0으로 buflim의 마지막을 닫아줌
      break;
    }
    buflim[i - n] = buf[i];        // buflim에 buf 값 복사
  }
  limit = atoi(buflim);            // buflim에 있는 값을 int로 변환해 lim에 담음
  if(!setmemorylimit(pid, limit)){ // 인자를 알맞게 넣고 setmemorylimit() 실행했을 때 0이 반환되면
    printf(2, "memlim %d: %d successed\n", pid, limit);
  }
  else{
    printf(2, "memlim %d: %d failed\n", pid, limit);
  }
}
```

만약 memlim 명령이라면 인자로 뒤에 pid와 limit를 받아와야 합니다. 따라서 따라서 buf에서 명령어 다음 부분부터 끝날 때까지 pid를 bufpid로 받아왔고, 이를 atoi()를 이용해 int로 바꿔서 pid에 넣었습니다. 이후, 그 다음 부분부터 끝날 때까지 limit를 buflim로 받아왔고, atoi()를 이용해 int로 바꿔서 limit에 넣었습니다. 이때 pid와 limit의 크기는 0 이상 10억 이하의 정수이므로 길이가 10이 최대이기 때문에 넉넉하게 bufpid와 buflim 배열의 크기를 20으로 잡았습니다.

이후, 인자를 넣고 setmemorylimit()를 실행해 0이 반환되면 성공, 그렇지 않다면 실패로 구분해 메시지를 출력해주었습니다.

각 명령은 해당 명령의 형식을 항상 따르고, 명세에 주어지지 않은 명령은 실행되지 않기 때문에 다른 예외처리는 딱히 하지 않았습니다.

다음은 LWP 구현입니다.

우선 type.h에 아래와 같이 thread_t를 선언해주었습니다. Thread의 id를 지정하는 변수이므로 int 형으로 선언해주었습니다.

```
typedef int thread_t;
```

먼저, thread_create() 함수는 새 스레드를 생성하고 시작하는 함수입니다. 따라서 fork()와 exec()에서 일부 가져와서 수정하였습니다.

```
// 새 스레드를 생성하고 시작하는 함수
int
thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg)
{
  int i;
  struct proc *np;
  struct proc *p;
  struct proc *curproc = myproc();
  uint sz, sp, ustack[2];

  // fork에서 변형
  if((np = allocproc()) == 0){ // 새로운 thread를 위한 공간을 np에 할당
    return -1;                 // 종료
  }

  if(curproc->pgdir == 0){ // 현재 curproc의 page directory가 0이면
    np->state = UNUSED;    // 상태를 UNUSED로 바꾸고
    return -1;             // 종료
  }
  np->parent = curproc->parent; // 새로운 thread의 parent를 현재 curproc의 parent로 설정
  *np->tf = *curproc->tf;       // 새로운 thread의 trap frame을 현재 curproc의 trap frame으로 설정

  np->tf->eax = 0;  // Clear %eax so that fork returns 0 in the child.

  for(i = 0; i < NOFILE; i++)                  // 0부터 file table의 최대 크기까지
    if(curproc->ofile[i])                      // file table에서 해당 file이 비어있지 않으면
      np->ofile[i] = filedup(curproc->ofile[i]); // 새로운 thread의 file을 현재 curproc의 file으로 설정
  np->cwd = idup(curproc->cwd);                // 새로운 thread의 cwd을 현재 curproc의 cwd으로 설정

  safestrcpy(np->name, curproc->name, sizeof(curproc->name)); // 새로운 thread의 name을 현재 curproc의 name으로 설정

  acquire(&ptable.lock);
```

```
np->pid = curproc->pid; // np의 pid를 현재 curproc의 pid로 설정

np->tid = nexttid++;    // np의 tid를 설정

np->called = curproc;   // thread_create를 호출한 curproc의 정보 저장

release(&ptable.lock);
```

일단 fork()에서 가져온 부분입니다. np에 새로운 thread를 위한 공간을 할당해주었습니다. 이후 원래는 현재 curproc의 pgdir를 복사해서 np에 할당해주지만 thread는 pgdir를 공유하고 stack 부분만 더 쌓기 때문에 그 부분을 없애고 curproc의 pgdir이 있는 지만 확인해주었습니다. 그리고 sz는 이후 exec()에서 가져온 부분에서 결정해줄 것이기 때문에 그 부분을 뺐습니다. np의 parent는 curproc의 parent여야 하기 때문에 그렇게 할당해주었습니다. 이후 fork()처럼 초기화 해주었고 np의 pid를 curproc의 pid와 같게 하고, tid는 nexttid를 넣어주어 차별화를 두고 하나 증가시켜주었습니다.

```
// stack 수정 부분 (exec에서 살짝 변형)
sz = curproc->sz; // sz에 현재 curproc의 sz를 할당

if((sz = allocuvm(curproc->pgdir, sz, sz + 2*PGSIZE)) == 0) // 2만큼의 가상 메모리 공간을 할당
  goto bad;
clearpteu(curproc->pgdir, (char*)(sz - 2*PGSIZE));          // 가드용 페이지를 설정
sp = sz; // stack pointer에 sz를 할당

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

sp -= 2*4; // sp를 2칸 감소
if(copyout(curproc->pgdir, sp, ustack, 2*4) < 0) // ustack의 data를 pgdir에 복사함
  goto bad;

np->stack_start = sz - 2*PGSIZE; // np의 stack의 시작 위치를 저장
curproc->sz = sz;                // 현재 curproc의 sz에 바뀐 sz 값을 할당

np->sz = sz;                     // np의 sz에 sz 값을 할당
np->pgdir = curproc->pgdir;      // np의 pgdir에 현재 curproc의 pgdir를 할당
np->tf->eip = (uint)start_routine; // instruction pointer에 start_routine를 저장
np->tf->esp = sp;                // stack pointer에 sp를 담음

*thread = np->tid;               // thread에 np의 tid를 넣음
```

```
  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->pid == curproc->pid) // 현재 pid와 같은 pid를 가졌다면
      p->sz = sz;              // sz를 갱신

  np->state = RUNNABLE;              // np의 상태를 RUNNABLE로 설정

  release(&ptable.lock);

  return 0;

 bad:
  np->state = UNUSED;                // np의 상태를 UNUSED로 설정
  return -1;
}
```

다음은 stack을 수정하는 부분입니다. 이는 exec()에서 밑에 부분을 가져왔습니다. stack은 curproc가 가지고 있던 sz에서 2(스택용, 가드용)만큼 더 크게 메모리 공간을 할당했습니다. 그리고 내용은 인자인 arg를 갖고 있어야 하기 때문에 arg를 넣어주었습니다. 이때, 원래는 void *형이었기 때문에 (uint)로 형변환을 해서 넣어주었습니다. ustack의 data를 pgdir에 복사해서 잘 넣고 나서는 np의 stack의 시작 주소를 stack_start에 넣어주었습니다. 그리고 sz를 잘 할당하고, eip에 start_routine 함수의 정보를 넣어주기 위해 또 (uint)로 형변환을 해주었습니다. 이후, thread에 np의 thread id인 tid를 넣어주었습니다. 이후에는 같은 pid를 가진 thread들은 sz값을 다 똑같이 갖고 있어야 하기 때문에 ptable을 순회하면서 pid가 같다면 sz를 전부 갱신해주었습니다. 마지막으로 np의 상태를 RUNNABLE로 표시해 시작해주었습니다.

```
// thread_create 함수의 system call 함수
int
sys_thread_create(void)
{
  int thread, start_routine, arg;

  if(argint(0, &thread) < 0 || argint(1, &start_routine) < 0 || argint(2, &arg) < 0)
    return -1;
  return thread_create((thread_t *)thread, (void *)start_routine, (void *)arg);
}
```

thread_create() 함수의 system call 함수입니다. 인자가 제대로 들어왔는지 확인하고, 그 인자를 넣어서 thread_create()를 실행해주었습니다.

```
// 스레드를 종료하고 값을 반환하는 함수
void
thread_exit(void *retval)
{
  struct proc *curproc = myproc();
  int fd;

  if(curproc == initproc)   // curproc가 initproc인 경우
    panic("init exiting");

  curproc->retval = retval; // retval값을 지정해줌

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){ // 0부터 NOFILE까지
    if(curproc->ofile[fd]){ // 열린 file이 있다면
      fileclose(curproc->ofile[fd]); // 열린 file을 닫아줌
      curproc->ofile[fd] = 0; // 참조한 값 초기화
    }
  }

  begin_op(); // file system 동기화를 위해 호출
  iput(curproc->cwd); // 현재 cwd를 file system에 반환
  end_op(); // file system 동기화를 종료
  curproc->cwd = 0; // 참조한 값 초기화

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(curproc->called); // exit()하려는 curproc을 호출한 called를 wakeup

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

다음으로 thread_exit() 함수는 스레드를 종료하고 값을 반환하는 함수입니다. 따라서 exit()에서 일

부 가져와서 수정하였습니다.

일단 retval 값을 join에서 받아가야 하기 때문에 먼저 넣어 놓습니다. 그리고 기존 exit()과 같이 열린 file을 다 닫아주고, file system 동기화 해 현재 cwd를 file system에 반환해주고 초기화 해주었습니다. 그 다음에 wakeup1로 parent를 깨웠었는데 여기서는 thread를 호출한 주체인 called를 wakeup하게 했습니다. 이후 상태를 ZOMBIE로 설정해주었습니다.

이외에 크게 바꾼 점은 없습니다.

```
// thread_exit 함수의 system call 함수
int
sys_thread_exit(void)
{
  int retval;

  if(argint(0, &retval) < 0)
    return -1;
  thread_exit((void *)retval);
  return 0;
}
```

thread_exit() 함수의 system call 함수입니다. 인자가 제대로 들어왔는지 확인하고, 그 인자를 넣어서 thread_exit()를 실행해주었습니다.

```
// 해당 스레드의 종료를 기다리고, 스레드가 thread_exit을 통해 반환한 값을 반환하는 함수
int
thread_join(thread_t thread, void **retval)
{
  struct proc *p;
  int havekids;
  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++){ // ptable 처음부터 끝까지 순회
      if(p->called != curproc || p->tid != thread) // p을 호출한 called가 curproc이 아니거나, tid가 thread가 아닌 경우
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){ // 상태가 ZOMBIE인 경우
        // Found one.
        kfree(p->kstack);
        p->kstack = 0;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        // 기존의 각종 초기화 + 새로 만든 값 초기화
        p->called = 0;
        p->tid = 0;
        p->stack_start = 0;
        *retval = p->retval; // retval에 p에 넣어놨던 retval 값을 할당
        release(&ptable.lock);
        return 0;
      }
    }
```

다음으로 thread_join() 함수는 해당 스레드의 종료를 기다리고, 스레드가 thread_exit을 통해 반환한 값을 반환하도록 하는 함수입니다. 기존의 wait() 함수에서 하는 일이기 때문에 여기서 일부 가져와서 수정하였습니다.

If문 안의 조건이 원래는 parent가 curproc이 아니면 넘어가는 거였는데, thread 기준으로 바뀌다 보니 해당 thread를 호출한 called가 다르면서 인자로 넘어온 thread 값이 tid와 다른 경우를 넘어가게 했습니다. 이후 ZOMBIE 상태라면 각종 초기화를 해주었습니다. 여기서 pgdir는 다른 thread와 공유를 하고 있기 때문에 freevm(p->pgdir)을 뺐습니다. 그리고 called와 tid, stack_start 같은 변수들도 다 초기화했습니다. 마지막으로는 retval에 p가 갖고 있던 retval 값을 넣어주었습니다. 이 이후에는 기존의 함수와 같이 동작하도록 해주었습니다.

```
// thread_join 함수의 system call 함수
int
sys_thread_join(void)
{
  int thread, retval;

  if(argint(0, &thread) < 0 || argint(1, &retval) < 0)
    return -1;
  return thread_join((thread_t)thread, (void **)retval);
}
```

thread_join() 함수의 system call 함수입니다. 인자가 제대로 들어왔는지 확인하고, 그 인자를 넣어서 thread_join()를 실행해주었습니다.

다음으로는 thread가 잘 돌아갈 수 있도록 몇몇 system call 함수들을 수정해주었습니다.

먼저, exec()에서 추가해준 부분입니다.

```
exec_exit(curproc->pid, curproc->tid); // pid가 같으면서 tid가 다른 thread 정리
curproc->tid = 0;
curproc->called = curproc;
curproc->retval = 0;
```

exec가 실행되면 기존 프로세스의 모든 스레드들이 정리되어야 하기 때문에 나머지 스레드를 종료하도록 해주었습니다. 나머지 스레드를 종료하기 위해서는 ptable의 순회가 필요하기 때문에 proc.c에 exec_exit() 함수를 만들어주어 사용하였습니다. 이후, 초기화를 위해 tid를 0으로, called는 curproc으로 retval는 0으로 초기화해주었습니다.

```
// exec에서 pid가 같으면서 tid가 다른 thread 정리하는 함수
void
exec_exit(int pid, int tid)
{
  struct proc *p;

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

여기서는 간단하게 pid가 같으면서 tid는 다른 thread들을 모두 정리해준 부분입니다. 마찬가지로 thread로 동작하기 때문에 freevm(p->pgdir)을 지우고 나머지만 정리해주었습니다.

```
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->pid == curproc->pid && p != curproc){ // pid가 같고 curproc이 아니라면
    kfree(p->kstack);
    p->kstack = 0;
    p->pid = 0;
    p->parent = 0;
    p->name[0] = 0;
    p->killed = 0;
    p->state = UNUSED;
  } // 자원 할당 해제 부분, wait()에서 해제한 것과 동일 (page table은 제외)
}
release(&ptable.lock);
```

다음은 exit() 함수에서 추가해준 부분입니다.

여기서도 마찬가지로 현재 돌아가고 있는 thread를 제외하고 pid가 같은 나머지 thread들의 자원을 모두 정리해주었습니다. 이는 kill에서 하나 이상의 스레드가 kill 되면 프로세스 내의 모든 스레드가 종료되어야 하고, kill 및 종료된 스레드의 자원들은 모두 정리되고 회수되어야 한다는 명세를 구현하기 위해 추가해주었습니다.

```
int
growproc(int n)
{
  uint sz;
  struct proc *curproc = myproc();
  struct proc *p;

  sz = curproc->sz;
  if(curproc->mem_limit != 0 && sz + n > curproc->mem_limit) // 추가적으로 할당 받는 memory가 limit보다 크다면
    return -1;
  if(n > 0){
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  } else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  }

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->pid == curproc->pid) // 현재 pid와 같은 pid를 가졌다면
      p->sz = sz;              // sz를 갱신
  }
  release(&ptable.lock);

  curproc->sz = sz;
  switchuvm(curproc);
  return 0;
}
```

마지막으로 sbrk(growproc)에서 추가해준 부분입니다.

먼저, mem_limit를 넘는 경우 추가적인 메모리를 할당 받지 못하게 해야 하기 때문에 그 부분을 여기서 구현해주었습니다. mem_limit이 0이라면 제한이 없기에 상관이 없지만 sz+n이 mem_limit 보다 더 커진다면 추가적인 메모리를 할당 받으면 안 되기 때문에 해당 조건문을 넣어주었습니다.

그리고 같은 pid를 가진 thread들은 sz값을 다 똑같이 갖고 있어야 하기 때문에 ptable을 순회하면서 pid가 같다면 sz를 전부 갱신해주었습니다.

마지막으로 system call 함수를 등록한 과정을 설명하겠습니다.

```
int             setmemorylimit(int, int);
void            printlist();
int             thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg);
void            thread_exit(void *retval);
int             thread_join(thread_t thread, void **retval);
```

먼저 다른 file에서도 해당 함수를 사용할 수 있도록 defs.h에 함수들을 넣어주었습니다.

```
#define SYS_exec2  22         extern int sys_exec2(void);          [SYS_exec2]   sys_exec2,
#define SYS_setmemorylimit 23 extern int sys_setmemorylimit(void);[SYS_setmemorylimit] sys_setmemorylimit,
#define SYS_printlist 24      extern int sys_printlist(void);     [SYS_printlist] sys_printlist,
#define SYS_thread_create 25  extern int sys_thread_create(void); [SYS_thread_create] sys_thread_create,
#define SYS_thread_exit 26    extern int sys_thread_exit(void);   [SYS_thread_exit] sys_thread_exit,
#define SYS_thread_join 27    extern int sys_thread_join(void);   [SYS_thread_join] sys_thread_join,
```

다음엔 syscall.h과 syscall.c에 wrapper function을 등록해주었습니다.

```
int exec2(char*, char**, int);
int setmemorylimit(int, int);
void printlist();
int thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg);
void thread_exit(void *retval);
int thread_join(thread_t thread, void **retval);
```

```
SYSCALL(exec2)
SYSCALL(setmemorylimit)
SYSCALL(printlist)
SYSCALL(thread_create)
SYSCALL(thread_exit)
SYSCALL(thread_join)
```

마지막으로 user program이 사용할 수 있도록 user.h에 해당 함수들을 넣고 usys.S에도 넣어주었습니다.

Result

먼저 pmanager 프로그램에서 다양한 명령을 실행해보았습니다.

```
$ pmanager
pmanager$ list
name: init, pid: 1, pages for stack: 1
memory size: 12288, memory limit: unlimited
name: sh, pid: 2, pages for stack: 1
memory size: 16384, memory limit: unlimited
name: pmanager, pid: 3, pages for stack: 1
memory size: 12288, memory limit: unlimited
```

list 명령입니다. 정상적인 출력이 됩니다.

```
name: pmanager, pid: 5, pages for stack: 1
memory size: 12288, memory limit: unlimited
pmanager$ kill 5
$
```

pid 5번인 pmanager를 kill해보았는데 바로 밖으로 잘 빠져나가 kill이 잘 작동됨을 확인했습니다.

```
pmanager$ execute hello_thread 6
pmanager$ Hello, thread!

pmanager$
```

hello_thread를 execute 했고, 잘 실행되는 것을 확인했습니다. 그리고 바로 그 다음에 이어서 pmanager가 실행되는 것을 확인했습니다.

```
pmanager$ memlim 3 13000
memlim 3: 13000 successed
pmanager$ list
name: init, pid: 1, pages for stack: 1
memory size: 12288, memory limit: unlimited
name: sh, pid: 2, pages for stack: 1
memory size: 16384, memory limit: unlimited
name: pmanager, pid: 3, pages for stack: 1
memory size: 12288, memory limit: 13000
pmanager$ memlim 3 0
memlim 3: 0 successed
pmanager$ list
name: init, pid: 1, pages for stack: 1
memory size: 12288, memory limit: unlimited
name: sh, pid: 2, pages for stack: 1
memory size: 16384, memory limit: unlimited
name: pmanager, pid: 3, pages for stack: 1
memory size: 12288, memory limit: unlimited
pmanager$ memlim 3 122
memlim 3: 122 failed
pmanager$ list
name: init, pid: 1, pages for stack: 1
memory size: 12288, memory limit: unlimited
name: sh, pid: 2, pages for stack: 1
memory size: 16384, memory limit: unlimited
name: pmanager, pid: 3, pages for stack: 1
memory size: 12288, memory limit: unlimited
```

memlim 명령입니다. 각각 13000, 0, 122를 limit으로 설정해보았습니다. 13000은 원래 memory size인 12288보다 크기 때문에 성공적으로 되는 것을 볼 수 있고, 0도 limit이 없는 것을 나타내기 때문에 성공적으로 되는 것을 확인할 수 있습니다. 하지만 122 같은 경우엔 0도 아니면서 12288보다 작기 때문에 실패한 것을 확인할 수 있습니다.

```
pmanager$ exit
$
```

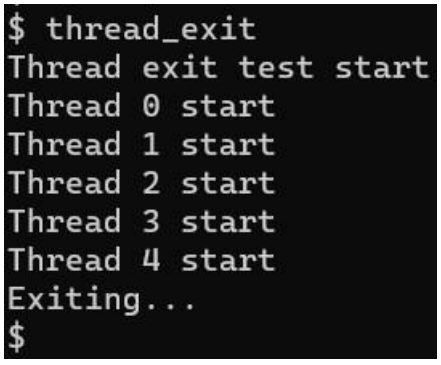
exit 명령입니다. pmanager를 종료한 것을 확인할 수 있습니다.

```
$ 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
Child of thread 0 start
Child of thread 2 start
Child of thread 3 start
Child of thread 4 start
Child of thread 1 start
Child of thread 0 end
Child of thread 2 end
Child of thread 3 end
Thread 2 end
Thread 3 end
Child of thread 1 end
Child of thread 4 end
Thread 0 end
Thread 1 end
Thread 4 end
Test 2 passed
```

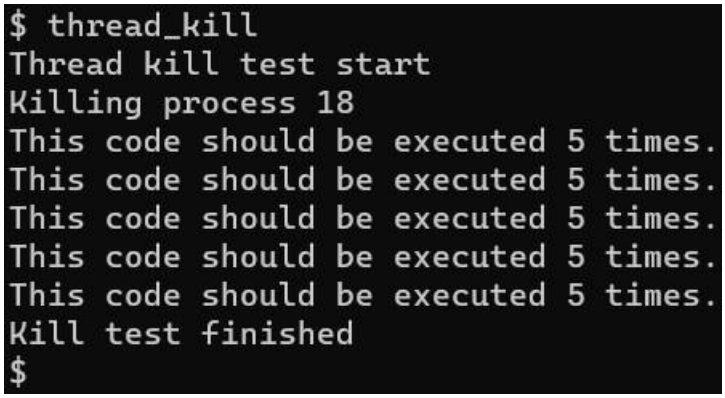
```
Test 3: Sbrk test
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Test 3 passed

All tests passed!
```

thread_test입니다. 기본 test와 fork test, sbrk test가 다 에러 없이 정상적으로 동작하는 것을 확인할 수 있었습니다.

thread_exec입니다. hello_thread 프로그램이 정상적으로 실행되는 것을 확인했고, 다른 스레드가 종료되어 에러가 없는 것을 확인할 수 있었습니다.



thread_exit입니다. Exiting... 출력 이후 바로 쉘로 빠져나가지는 것을 확인할 수 있었습니다.



thread_kill입니다. 프로세스가 kill 되었을 때 그 프로세스 내의 모든 스레드가 올바르게 종료되는 것을 확인했습니다. 좀비 프로세스가 발생하지 않았고, 자식 프로세스들의 스레드는 모두 즉시 종료되는 것을 확인했습니다.

Trouble shooting

1. xv6를 실행했을 때 유저 프로그램이 실행이 안 되는 경우

해결: growproc()에서 mem_limit 보다 할당 받으려는 크기가 커지면 -1을 return하게 했었습니다. 하지만 if 조건문에서 mem_limit가 0일 때를 제외하고 조건을 걸었어야 했는데 그렇게 하지 않아서 여기서 growproc()이 제대로 작동되지 않는 경우가 발생했습니다.

이에 if(curproc->mem_limit != 0 && sz + n > curproc->mem_limit)로 조건문을 고쳐주어 해결해 주었습니다.

2. thread_test를 했을 때 재부팅이 되는 경우

해결: thread_join에서 wait()을 그대로 가져왔더니 freevm(p->pgdir);가 있어서 thread라 함부로 건드리면 안 되는 부분(같은 pid를 가진 thread끼리 공유하는 pgdir)이었음에도 할당 해제를 해서 생기는 문제였습니다.

이에 해당 부분을 지워주어 해결했습니다.

3. thread_test -> test 3: sbrk에서 lapicid 0: panic: remap 오류가 뜨는 경우

해결: growproc에서는 pid가 같은 경우에 sz 값을 업데이트를 해줬는데, thread_create에서는 sz 값을 변경하지만 다른 thread에 sz 값을 업데이트를 해주지 않아서 memory를 잘못 건드리는 경우가 발생했던 것으로 보입니다.

이에 thread_create에서도 pid가 같은 경우에 sz 값을 업데이트 시켜주면서 해결했습니다.