운영체제 Project 2 WIKI

2019008813 문원찬

0. proc 구조체

WIKI에 앞서 이번 과제를 통해 프로세스 구조체에 추가한 내용을 바탕으로 구현 내용을 간략히 정리해보았다.

proc.h

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)

  uint s_size;                 // stack size
  uint m_limit;                // memory limit (bytes)

  int tid;                     // main thread: 0
  uint sp;                     // virtual add base
  void *retval;                // return value
  struct proc *main;           // main
};
```

(uint)s_size는 stack size를 설정하는 exec2 시스템콜에서 설정되며 pmanager의 list 명령어에서 실행중인 프로세스의 stack size를 출력하기 구조체에 저장하였다. (uint)m_limit은 setmemorylimit 시스템콜로 설정되며 proc.c의 growproc에서 해당 프로세스의 메모리 제한을 알고 비교하기 위해 저장하였다. 아래 4개는 thread를 구현하기 위한 변수이다. 우선 쓰레드를 프로세스와 동등하게 생각하고 관리를 메인 쓰레드가 하는 방식으로 구현하기 때문에, 프로세스 마다 메인 쓰레드를 가리키는 포인터를 저장한다. tid가 0이면 메인 쓰레드이며 쓰레드를 늘릴 때마다 하나씩 늘린다. (uint)sp는 각 쓰레드마다 stack의 가장 아래를 나타내는 변수이다. 쓰레드들의 스택을 메인 프로세스의 메모리 가장 위에 차곡차곡 쌓기 때문에 나중에 삭제할 때 시작주소인 sp와 끝 주소인 sz를 이용한다. retval은 return value로 thread_join등에서 사용된다.

1. exec2

a. Design

exec.c의 내용을 간단히 바꾸어 구현하였다. exec.c는 stack page를 할당하는 부분에서 2개의 page를 할당해서 아래는 가드용, 하나는 user stack용으로 사용한다. exec2.c에선 이를 (1+stacksize)만큼 할당하고 가장 아래가 가드용, 나머지는 user stack용으로 구현했다. 프로세스에게 넘겨주는 sz 또한 늘어난 만큼의 sz로 넘겨주도록 했다. 또한 stacksize를 proc구조체에 저장해 나중에 확인이 가능하도록 했다. stacksize에 대해서도 1~100사이의 정수만 가능하게 했다. 마지막으로 명세에 맞게 이를 시스템콜로 구현하였다.

b. Implement

exec2.c

```
// Check stacksize
if(stacksize < 1 || stacksize > 100){
  end_op();
  cprintf("exec2: fail(stacksize)\n");
  return -1;
}
```

```
// Allocate (1+stacksize) pages at the next page boundary.
// Make the first inaccessible.  Use the next as the user stack.
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + (1+stacksize)*PGSIZE)) == 0) // 1 guard + stacksize pages
  goto bad;
clearpteu(pgdir, (char*)(sz - (1+stacksize)*PGSIZE));
sp = sz;

// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
  if(argc >= MAXARG)
    goto bad;
  sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
  if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
  ustack[3+argc] = sp;
}
ustack[3+argc] = 0;

ustack[0] = 0xffffffff;  // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4;  // argv pointer




sp -= (3+argc+1) * 4;
if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
  goto bad;
```

위 사진을 통해 stacksize를 검사했고 아래 사진에서 allocuvm 함수의 3번째 인자를 sz+(1+stacksize)*PGSIZE로 설정해 (1+stacksize)만큼의 페이지를 할당받게 했다. clearpteu 함수로 가장 아래 페이지는 가드페이지로 설정했다.

2. setmemorylimit

a. Design

proc.c에 함수로 추가하였다. ptable을 돌면서 해당하는 pid를 가지는 프로세스 구조체에 limit을 설정한다(m_limit). growproc 함수에서 해당 limit보다 크면 -1을 return하게 했다.

b. Implement

proc.c

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

  if(limit < 0)
    return -1;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid && p->tid == 0){
      if(p->sz > limit && limit != 0){
        release(&ptable.lock);
        return -1;
      }
      else{
        p->m_limit = limit;
        release(&ptable.lock);
        return 0;
      }
    }
  }
  release(&ptable.lock);
  return -1;
}
```

프로세스 구조체에 m_limit을 만들어 해당 프로세스의 limit을 설정하게 했다. limit인자가 음수면 -1을 return한다. ptable을 돌며 해당되는 pid를 찾고 limit이 구조체의 (int)m_limit과 비교하여 크거나 0이면 m_limit을 limit으로 설정한다. 나중에 쓰레드 구현에서 (tid = 0)인 main 프로세스가 모든 쓰레드 메모리를 관리하므로 해당 프로세스에서만 m_limit을 설정했다.

3. pmanager

a. Design

sh.c 코드에서 약간의 수정을 통해 필요한 명령어들이 작동하게 만들었다. 주로 buf 배열을 참조하여 명령어들을 받을 수 있게 했다. 각 buf의 size는 100으로 넉넉히 잡았다. 예외처리는 err변수로 다루었다.

list 명령어는 시스템콜로 proc.c의 list 함수를 만들어 작동하게 했다. list 함수는 시스템콜로써 ptable에서 메인 쓰레드이면서 SLEEPING, RUNNING, RUNNABLE인 프로세스만 상태를 출력한다.

kill 명령어는 kill 시스템콜을 사용했다.

execute 명령어는 sh.c의 exec방식을 수정하여 path를 건네주고 stacksize를 전역변수로 설정해 나중에 exec2 시스템콜을 부르게 했다.

memlim 명령어는 setmemorylimit 시스템콜을 사용했다. exit 명령어는 exit 시스템콜을 사용했다.

b. Implement

pmanager.c

```
// Read and run input commands.
while(getcmd(buf, sizeof(buf)) >= 0){
  err = 0;

  //list
  if(buf[0] == 'l' && buf[1] == 'i' && buf[2] == 's' && buf[3] == 't' && buf[4] == '\n'){
    //list systemcall
    list();
    continue;
  }
  //kill <pid>
  if(buf[0] == 'k' && buf[1] == 'i' && buf[2] == 'l' && buf[3] == 'l' && buf[4] == ' '){
    for(i = 5;buf[i]!='\n';i++){
      if('0' <= buf[i] && buf[i] <= '9'){
        buf2[i-5] = buf[i];
        continue;
      }
      err = 1;
    }
    if(err){
      printf(2, "failed\n");
      continue;
    }
    if(kill(atoi(buf2)) < 0)
      printf(2, "failed\n");
    else
      printf(2, "kill %s\n", buf2);
    continue;

  }
```

pmanager.c 의 main 함수의 일부분이다. 다음과 같이 while문을 돌며 buf를 받고, buf를 읽어 명령어를 수행한다. kill 명령어 작동을 보면 pid인자를 받기 위해 buf2를 만들어 저장했다. 숫자가 아니면 err를 1로 하여 fail하게 작동한다. execute, memlim의 인자들 또한 buf2, buf3에 저장했다 atoi 함수를 사용해 숫자로 바꾼 뒤 시스템콜의 인자로 넣어줬다.

proc.c

```
void
list(void)
{

  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state >= 2 && p->state <= 4 && p->tid == 0) //main thread if문 추가
      cprintf("name: %s pid: %d stacksize: %d memorysize: %d memorylimit: %d", p->name, p->pid, p->s_size, p->sz, p->m_limit);
    else
      continue;
    cprintf("\n");
  }
}
```

다음은 list 시스템콜 구현을 위한 list 함수이다. state가 2~4이면 SLEEPING, RUNNING, RUNNABLE 이므로 해당 프로세스 중 쓰레드는 출력하면 안되니 tid가 0인 메인 쓰레드만 출력한다. 만약 쓰레드가 여럿 있어도 memory관리는 메인 쓰레드에서만 하므로 memorysize는 메인 쓰레드의 sz 값으로 구할 수 있다.

c. Result

```
$ pmanager
[pmanager] list
name: init pid: 1 stacksize: 0 memorysize: 12288 memorylimit: 0
name: sh pid: 2 stacksize: 0 memorysize: 16384 memorylimit: 0
name: pmanager pid: 9 stacksize: 0 memorysize: 16384 memorylimit: 0
[pmanager] fork
[pmanager] a
[pmanager]
[pmanager]
[pmanager] memlim 2 5
failed
[pmanager] list
name: init pid: 1 stacksize: 0 memorysize: 12288 memorylimit: 0
name: sh pid: 2 stacksize: 0 memorysize: 16384 memorylimit: 0
name: pmanager pid: 9 stacksize: 0 memorysize: 16384 memorylimit: 0
[pmanager] memlim 2 17000
memlim 2 17000
[pmanager] list
name: init pid: 1 stacksize: 0 memorysize: 12288 memorylimit: 0
name: sh pid: 2 stacksize: 0 memorysize: 16384 memorylimit: 17000
name: pmanager pid: 9 stacksize: 0 memorysize: 16384 memorylimit: 0
[pmanager] exit
$
$ pmanager
[pmanager] list
name: init pid: 1 stacksize: 0 memorysize: 12288 memorylimit: 0
name: sh pid: 2 stacksize: 0 memorysize: 16384 memorylimit: 17000
name: pmanager pid: 11 stacksize: 0 memorysize: 16384 memorylimit: 0
[pmanager] kill 11
$
```

pmanager 실행 결과이다. list는 잘 작동하며 buf에 예외처리 또한 잘 되었다. memlim 함수에서는 사이즈보다 많이 적게 제한하면 실패하고 사이즈보다 크게 제한하면 제대로 설정된 모습이다. exit

과 kill도 잘 작동한다.

4. LWP

a. design

우선 [**0. proc 구조체**]에서 적었듯이 쓰레드와 프로세스를 같게 생각한다. 쓰레드를 처음 만드는 프로세스가 메인 쓰레드가 되고 모든 쓰레드들의 메모리를 관리한다. 모든 쓰레드들은 각자의 스택을 가지고 있는데, 이 스택들은 모두 메인 쓰레드의 총 메모리의 위로 2페이지씩 쌓인다. 첫 페이지는 가드용이고 다음은 유저 스택용이다. 유저 스택에는 제일 위에 가짜 pc가 들어가고 다음은 인자로 받은 arg가 들어간다. 각 쓰레드마다 시작주소를 가지고 있어서 thread_exit 때 해당하는 공간만 지울 수 있다. thread_join에선 메인 쓰레드가 호출하여 메인 쓰레드들을 제외한 나머지를 다 지우고, 마지막에 메인 쓰레드를 zombie로 만들고 sched를 호출한다. thread_create는 fork와 exec를 참고하였고 thread_exit은 exit을, thread_join은 wait을 참고했다. 전부 시스템콜로 구현했다.

fork, exec, sbrk, sleep, pipe의 시스템콜 또한 쓰레드에 맞게 수정해준다. fork의 경우, 메인에서 실행한게 아니면 총 메모리의 크기가 같은 메모리의 크기가 아닐 수도 있기에 메인의 메모리의 크기를 참조하게 한다. exec의 경우 메인이 아닐 경우, 해당 쓰레드의 커널 스택을 메인의 커널 스택으로 설정하는 등 메인에게 옮겨주고 현재 프로세스를 메인으로 바꿔준다. 메인에서 exec가 다 진행되면 마지막에 메인이 아닌 쓰레드들을 다 정리해준다. kill의 경우, pid에 해당하는 프로세스 중 메인을 골라 그것만 killed= 1로 설정한다. sleep과 pipe는 쓰레드랑 프로세스랑 같은 개념으로 보기에 수정할 필요가 없다.

thread_create, thread_exit, thread_join

b. Implement

proc.c

```c
//like fork and exec
int
thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg)
{
  int i;
  struct proc *np;
  struct proc *curproc = myproc();
  uint sz, sp, ustack[2];
  struct proc *main;

  main = curproc;
  if(curproc->tid != 0)
    main = curproc->main; //main thread에서 총 관리

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

thread_t 자료형은 쓰레드랑 프로세스를 같게 보기에 따로 구조체를 만들지 않고 uint로 대체해서 구현했다. 메인 쓰레드를 제외한 나머지는 모두 main에 메인을 가지고 있다. 처음엔 fork와 같이 새로운 np를 할당한다.

proc.c

```
// Make new ustack (exec.c 참고)
sz = PGROUNDUP(main->sz);
if((sz = allocuvm(main->pgdir, sz, sz + 2*PGSIZE)) == 0){ //main thread 위에 두 페이지 추가
  np->state = UNUSED;
  return -1;
}
clearpteu(main->pgdir, (char*)(sz - 2*PGSIZE)); //guard 설정
sp = sz;
ustack[0] = 0xffffffff;
ustack[1] = (uint)arg;
sp -= 2*4;
if(copyout(main->pgdir, sp, ustack, 2*4)<0){
  np->state = UNUSED;
  return -1;
}

// Copy pagetable from proc and set np's state
np->pid = main->pid;
np->tid = nexttid++;
np->pgdir = main->pgdir;
np->sp = sz - 2*PGSIZE;
np->sz = sz;
main->sz = sz; //main은 항상 늘어난 sz를 가지고 있어서 다음에 thread 만들 때, 또 그 위에 할당
np->parent = main;
np->main = main;
*np->tf = *main->tf;

*thread = np->tid;
```

앞에서 설명한 내용을 구현했다. np를 새로운 쓰레드로 생각하고 값을 설정해준다.

proc.c

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

  if(curproc->tid==0) //main은 x
    return;


  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  acquire(&ptable.lock);

  // main might be sleeping in wait().
  curproc->retval = retval; //retval 넘김
  wakeup1(curproc->main);

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

thread_exit에선 retval을 사용해 return value를 반환하게 했다.

proc.c

```
//like wait
int
thread_join(thread_t thread, void **retval)
{
  struct proc *p;
  int havekids;
  struct proc *curproc = myproc();

  if(curproc->main!=0){
    return -1;
  }

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited thread.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->main != curproc || p->tid != thread)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        *retval = p->retval;
        kfree(p->kstack);
        p->kstack = 0;

        // freevm(p->pgdir);
        // deallocuvm(p->pgdir, p->sz, p->sp);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;

        release(&ptable.lock);
        return 0;
      }
    }
    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

join에서 메인이면 기다리면서 나머지 쓰레드들을 정리해준다.

proc.c

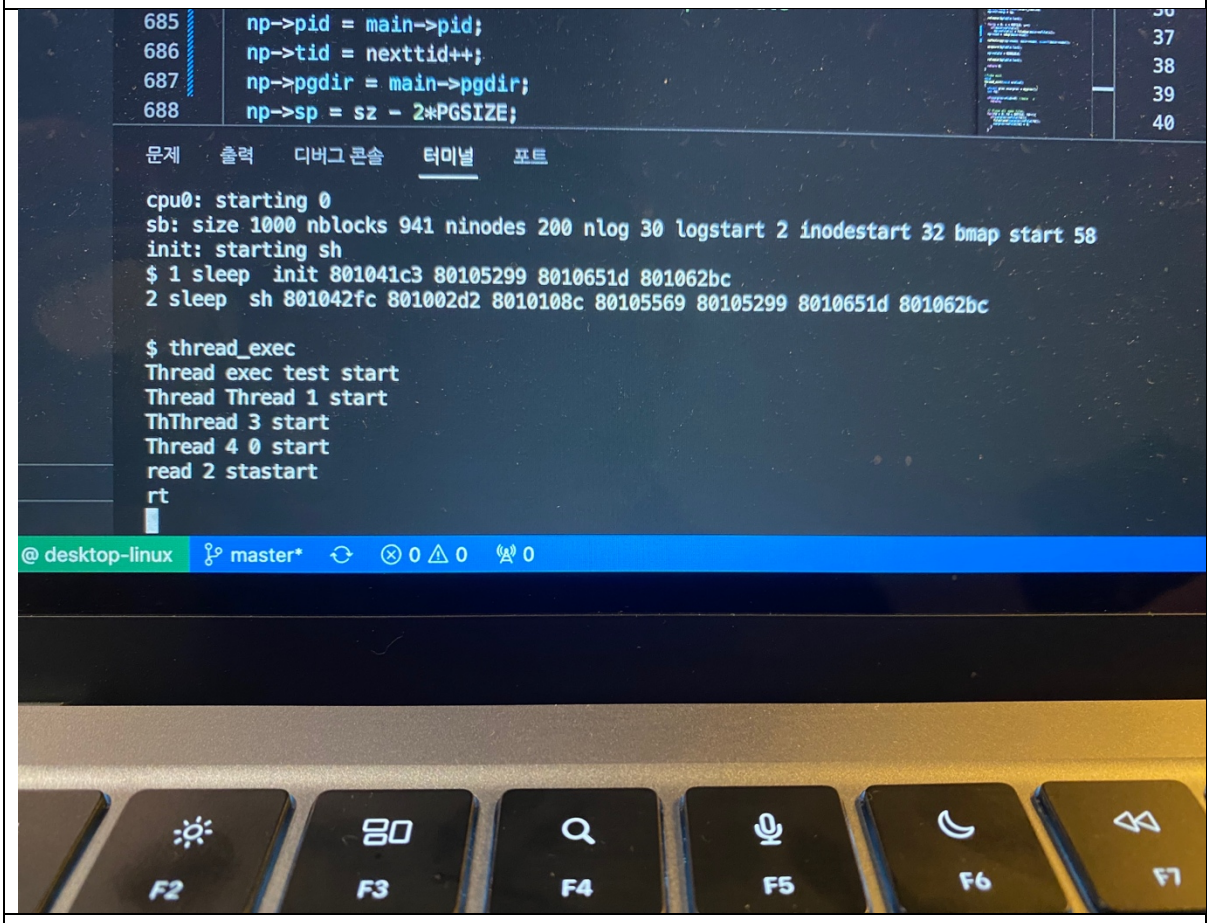
```
//exec시 call
void
kill_thread(int pid){
  struct proc* p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid != pid || p->tid == 0)
      continue;

    // Found one.
    p->killed = 1;
    wakeup1(p);
    }
  release(&ptable.lock);
}
```

exec하면 나머지 쓰레드들 지우기 위해 함수를 구현했다.

c. Result and trouble



모든 테스트에서 하나만 출력되고 다시 쉘이 실행되는 현상이 발생했다. 이를 보아 스택을 잘못 접근하는 듯해 보였다. 하지만 이를 수정하기엔 시간이 부족해서 수정하지 못했다.