Operating System Project 2 Wiki

2019092306 구본준

Design

1. Process management

exec2() 함수는 기존의 *exec()* 함수를 그대로 이용하되, 스택용 페이지를 할당해주는 부분만 스택 크기를 인자로 받아 설정하도록 변경할 것이다. *setmomorylimit()* 함수는 *proc* 구조체에 *limit*이라는 이름의 변수를 추가하여 *proc*의 메모리 크기를 나타내는 *sz* 변수가 *limit*를 넘어가지 않도록 구현할 것이다. *setmemorylimit()* 함수에서는 단순히 *limit* 변수에 값을 넣어주며, 메모리 크기를 변경하는 *growproc(), exec2()* 등의 함수에서 만약 *limit* 값보다 큰 사이즈로 변경을 요청하는 경우 실행하지 않도록 만들 것이다.
pmanager은 사용자로부터 입력을 받아 다섯가지 기능을 수행하게 된다. *list* 명령어의 경우 새로운 시스템 콜을 만들어 단순히 그 시스템 콜을 호출하면 프로세스의 정보가 전부 출력되도록 구현할 것이다. *kill, execute, mimlim* 명령어는 미리 구현되어 있는 *kill(), exec2(), setmomorylimit()* 시스템 콜을 호출하는 방식으로 구성되는데 이때 *execute* 명령어는 pmanager위에서 실행되면 안 되기 때문에 *fork*() 함수를 통해 프로세스를 하나 생성한 후 거기에서 *exec()*를 호출할 것이다. 마지막으로 *exit* 명령어는 단순히 사용자의 입력을 받는 반복문을 빠져나오도록 할 것이다.

2. Light-weight process

LWP 역시 하나의 프로세스 형태라는 아이디어에서 착안하여 프로세스와 동일하게 생성을 하되, 부모 프로세스의 데이터를 복사해오는 것이 아닌 단순히 가리키도록 구현할 것이다. 따라서 *thread_create()* 함수는 프로세스 생성 과정과 동일하게 *allocproc()* 함수를 통해 *ptable*의 빈 공간을 찾아 할당 받은 후 *fork()* 함수와 유사하게 스레드를 생성할 것이다. 다만 페이지 디렉토리를 복사하는 부분과 스택 포인터를 설정해주는 부분은 스레드를 위해 적절히 다르게 해줄 것이다. *thread_exit()* 함수는 *exit()* 함수와 유사하게 구현할 것이다. 차이점은 메모리 전체를 해제하는 영역을 제거해야 한다는 점이다. *thread_join()* 함수는 *wait()* 함수와 유사하게 구현할 것이며 다른 점은 인자로 받은 특정 스레드 번호를 기다리도록 한다는 점이다. 이를 위해 프로세스 구조체에서 thread id를 위한 변수, return value를 저장하는 변수를 추가하여 주어야 한다.

스레드를 기존의 *ptable* 구조체를 그대로 이용하여 구현하므로 프로세스와 스레드의 개수 합은 64개가 최대이며 NPROC 값을 변경하면 이를 늘리거나 줄일 수 있다. 기존에 있던 시스템 콜인 *fork(), exec(), exit()* 함수와 스케줄러 등은 기존 프로세스 구조가 크게 변하지 않으므로 살짝만 고치거나 아예 고치지 않아도 정상적으로 동작할 것이다.

Implement

1. proc

```
int limit;                  // Limit of memory size
int spnum;                  // The number of stack pages
thread_t tid;               // Thread ID
void *threadretval;         // Return value of thread exit
```

proc 구조체이다. 기존 구조에 *limit, spnum, tid, threadretval* 변수를 추가하였다. *limit* 변수는 *setmemorylimit()* 함수를 통해 설정할 수 있는 메모리의 제한선을 저장한다. *spnum* 변수는 pmanager의 *list* 명령어를 통해 출력해주어야 하는 스택용 페이지의 개수를 저장한다. *tid* 변수는 스레드마다 할당되는 고유 번호이며 *thread_t* type은 int형으로 선언된다. *threadretval* 변수는 *thread_exit()* 함수에서 전달하는 스레드의 반환 값을 저장한다.

2. allocproc()

```
p->tid = nexttid++;
```

allocproc() 함수 내부에 추가한 코드이다. 스레드를 생성할 때에도 호출을 하므로 스레드 번호를 부여하도록 구현하였다. 이때 프로세스도 하나의 메인 스레드로 간주하여 스레드 번호를 부여해준다.

3. growproc()

```
if(curproc->limit != 0 && sz + n > curproc->limit)
  return -1;
```

growproc() 함수 내부에 추가한 코드이다. *limit* 변수를 확인하여 현재 확장하려는 메모리의 크기가 한계를 넘어가지 않는지 확인한다.

```
// Update sz variable of the other threads.
for(t = ptable.proc; t < &ptable.proc[NPROC]; t++)
  if(t->pid == curproc->pid)
    t->sz = sz;
```

증가한 메모리의 크기에 대한 정보가 모든 스레드에게 공유되어야 하므로 같은 프로세스

스레드의 *sz* 변수를 업데이트해준다.

4. exit()

```
  // Clean up all other threads.
  thread_clear1();
```

exit() 함수 내부에 추가한 코드이다. 다른 부분은 기존의 코드와 동일하며 스레드를 정리해주기 위한 기능을 추가하였다. 스레드가 여러 개 있는 경우 한 스레드에서 *exit()* 함수를 호출한다면 모든 스레드가 종료되어야 한다. 따라서 *thread_clear1()* 이라는 함수를 만들어 이를 구현하였다. *kill()* 함수의 경우 기존과 동일하게 구현되어 있지만 *is_killed* 변수를 세팅하여 *exit()* 함수를 호출할 때 모든 스레드가 종료되므로 *kill()* 함수 역시 kill한 프로세스의 모든 스레드가 종료되는 방식으로 동작한다.

5. procdump2()

```
// Print a process listing to console.  For pmanager
// Runs when user types list on pmanager.
// No lock to avoid wedging a stuck machine further.
void
procdump2(void)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if((p->state != RUNNING && p->state != RUNNABLE && p->state != SLEEPING)
        || p->pid == p->parent->pid)
      continue;
    cprintf("**************************************\n");
    cprintf("name                 : %s\n", p->name);
    cprintf("pid                  : %d\n", p->pid);
    cprintf("stack page number     : %d\n", p->spnum);
    cprintf("allocated memory size : %d\n", p->sz);
    if(p->limit == 0)
      cprintf("memory maximum limit  : no limit\n");
    else
      cprintf("memory maximum limit  : %d\n", p->limit);
    cprintf("**************************************\n");
  }
}
```

Pmanager의 *list* 명령어를 처리해주기 위한 함수이다. 시스템 콜로 만들어져 있으며 프로세스의 이름, *pid,* 스택 페이지 개수, 할당된 메모리 크기, 메모리 최대 제한을 한 줄씩 출력해준다. 이 때 *limit* 변수가 0인 경우 *limit* 변수의 값 대신 제한이 없다는 문구를 출력해준다. 프로세스 간의 출력 내용은 '*'로 이루어진 줄을 통해 구분을 하였다.

6. setmemorylimit()

```
// Set the limit of process memory
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->pid == pid){
    // If the limit is smaller than current process memory size.
    if(limit != 0 && limit < p->sz) {
      release(&ptable.lock);
      return -1;
    }
    p->limit = limit;
    pexist = 1;
  }
}
```

Parameter로 입력 받은 *pid*의 *limit* 변수를 세팅해준다. 이 때 스레드가 존재하는 경우 스레드끼리는 같은 *pid*를 공유하므로 모든 스레드의 *limit* 값이 업데이트된다. 예외처리로 *limit* 값이 현재 메모리 크기보다 작은 경우 '-1'을 반환하도록 하였다.

```
// If the process that matches pid does not exist.
if(pexist == 0)
  return -1;
```

다른 예외 처리로 입력받은 *pid*와 일치하는 *pid*를 가진 프로세스를 찾지 못한 경우 '-1'을 반환하도록 하였다.

7. thread_create()

```
// Check the limit of the current process.
if(curproc->limit != 0 && curproc->sz + PGSIZE > curproc->limit)
    return -1;
```

스레드의 스택을 위한 페이지를 할당해야 하는데 이 때 메모리의 크기가 *limit*를 넘어가면 안 되므로 이를 확인해주는 코드이다.

```
// Allocate a new stack for this thread.
  if((curproc->sz = allocuvm(curproc->pgdir, curproc->sz, curproc->sz +
PGSIZE)) == 0)  {
    kfree(t->kstack);
    t->kstack = 0;
    t->state = UNUSED;
    return -1;
  }
```

allocproc() 함수를 통해 *ptable*의 공간과 *kstack*을 할당 받은 이후 스레드의 스택 공간을 위하여 메모리 한 페이지를 할당받고 *sz*를 그만큼 증가시킨다. 만약 페이지 할당에 실패했을 경우 *allocproc()*으로 할당 받은 자원을 해제해주고 '-1'을 반환한다.

```
// Share thread state with current process.
```

```
  t->pid = curproc->pid;
  t->sz = curproc->sz;
  t->pgdir = curproc->pgdir;
  t->limit = curproc->limit;

  // Copy thread trap frame state from current process.
  *t->tf = *curproc->tf;
```

현재 프로세스의 *pid*, 메모리 사이즈, 페이지 디렉토리, 메모리 최대 제한 값을 공유 받는다. 트랩 프레임의 값은 스레드가 각각 개별적으로 갖고 있어야 하므로 공유하는 것이 아닌 현재 값을 복사 받는다.

```
// Check if the curproc is main thread.
  if(curproc->parent->pid == curproc->pid)
    t->parent = curproc->parent;
  else
    t->parent = curproc;

  // Increase the stack page number of the main thread
  t->parent->spnum++;
```

현재 프로세스가 메인 스레드인지 확인하여 새로 생성한 스레드의 부모를 메인 스레드가 되도록 만들어준다. 이후 메인 스레드의 스택용 페이지 수를 증가시킨다. *spnum* 변수의 경우 pmanager가 메인 스레드만 확인하여 출력하므로 메인 스레드의 값만 설정해주는 것이다.

```
// Strat from the start routine and set sp to top of the page
  t->tf->eip = (uint)start_routine;
  t->tf->esp = (uint)t->sz;

  // Pass the argument to the stack
  t->tf->esp -= 4;
  *(uint*)t->tf->esp = (uint)arg;

  // Set the fake return address.
  t->tf->esp -= 4;
  *(uint*)t->tf->esp = 0xffffffff;
```

생성한 스래드 트랩 프레임의 *eip*값을 인자로 입력받은 *start_routine* 값으로 설정하여 시작 루틴 함수에서부터 스레드가 동작하도록 만들어준다. *esp*값은 직전에 새로 할당 받은 페이지의 가장 끝으로 세팅해준다. 이후 입력 받은 인자와 가짜 반환 주소를 순서대로 스택에 넣어준다.

```
// Copy the address of the file descriptor without fileup()
  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      t->ofile[i] = curproc->ofile[i];
```

```
  // Copy the address of the current derectory widout idup().
  t->cwd = curproc->cwd;

  safestrcpy(t->name, curproc->name, sizeof(curproc->name));
```

현재 프로세스의 파일 디스크립터와 디렉토리 주소를 복사하는데 이 때 스레드에서의 접근은 동일한 프로세스에서의 접근으로 간주되기 때문에 *fileup()* 함수와 *idup()* 함수를 사용하지 않고 복사한다. 이후 현재 프로세스의 이름을 그대로 복사해온다.

```
// Update sz variable of the other threads.
  for(t = ptable.proc; t < &ptable.proc[NPROC]; t++)
    if(t->pid == curproc->pid)
      t->sz = curproc->sz;
```

이미 존재하는 다른 스레드의 *sz* 변수의 값이 동일해야 하므로 이를 위해 전체 *ptable*을 돌며 *sz* 변수를 업데이트 해준다.

8. thread_exit()

```
  // Main thread exiting.
  if(curproc->pid != curproc->parent->pid)
    exit();
```

만약 메인 스레드가 *thread_exit()* 함수를 호출했을 경우 *exit()* 함수가 호출되도록 한다.

```
for(fd = 0; fd < NOFILE; fd++)
      curproc->ofile[fd] = 0;
  curproc->cwd = 0;

  curproc->threadretval = retval;
```

아직 다른 스레드에서 파일 디스크립터와 현재 디렉토리 정보를 사용하고 있을 수 있으므로 단순히 주소 값을 0으로 초기화해 준다. 이후 인자로 받은 *retval*을 프로세스 구조체의 *threadretval* 변수에 입력해준다.

```
// Another thread might be sleeping in wait().
  for(t = ptable.proc; t < &ptable.proc[NPROC]; t++)
    if(t->pid == curproc->pid && t->tid != curproc->tid)
      wakeup1(t);
```

*thread_join()*을 어떤 스레드에서 호출하고 있을 지 모르므로 모든 스레드를 *wakeup1()* 함수를 호출하여 깨워준다. 나머지 과정은 기존에 구현되어 있던 *exit()* 함수와 동일하다.

9. thread_join()

```
// Wait itself.
  if(thread == curproc->tid)
    return -1;
```

입력 받은 스레드 번호가 현재 스레드 번호와 같은 경우 예외처리로 '-1'을 반환해준다.

```
        *retval = t->threadretval;
        t->threadretval = 0;
```

기다리는 thread를 찾아 기다리는 과정은 기존에 구현되어 있던 *wait()* 함수와 동일하며 다른 부분은 *retval* 값을 설정해주는 부분과 전체 페이지 디렉토리를 할당 해제해주는 부분 뿐이다

10. thread_clear1()

```
  // Make the current thread to the main thread
  if(curproc->parent->pid == curproc->pid) {
    curproc->spnum = curproc->parent->spnum;
    curproc->parent = curproc->parent->parent;
  }
```

현재 스레드를 제외하고 다른 스레드를 정리해주는 함수로 먼저 현재 스레드를 메인 스레드로 만들어준다. 이는 *exec()* 함수가 특정 스레드에서 호출되었을 때 그 스레드가 메인 스레드가 되어 프로그램을 실행시킬 수 있도록 하기 위함이다. 메인 스레드가 되었기 때문에 *spnum* 변수 값을 갖고 있어야 하므로 이를 메인 스레드로부터 복사해온다.

```
// Pass abandoned children to init.
    if(t->parent->pid == curproc->pid){
      t->parent = initproc;
      if(t->state == ZOMBIE)
        wakeup1(initproc);
    }
```

스레드를 제거해야 하는데 이 때 특정 스레드에서 *fork()* 함수를 통해 자식 프로세스를 생성했을 경우 그 프로세스의 부모를 *init* 프로세스로 만들어준다.

```
// Clear all other threads
    if(t->pid != curproc->pid || t->tid == curproc->tid)
      continue;
    for(fd = 0; fd < NOFILE; fd++)
      t->ofile[fd] = 0;
    t->cwd = 0;
    kfree(t->kstack);
    t->kstack = 0;
    t->pgdir = 0;
    t->pid = 0;
    t->tid = 0;
    t->parent = 0;
    t->name[0] = 0;
    t->killed = 0;
    t->limit = 0;
    t->spnum = 0;
```

```
    t->threadretval = 0;
    t->state = UNUSED;
```

*thread_exit()*과 유사하게 스레드를 정리해준다.

11. thread_clear()

```
void
thread_clear(void)
{
  acquire(&ptable.lock);
  thread_clear1();
  release(&ptable.lock);
}
```

exec() 함수에서 안전하게 *thread_clear1()* 함수를 호출할 수 있도록 *ptable lock*을 부여한 함수이다.

12. exec()

```
 // Check the memory limit
  if(curproc->limit !=0 && sz > curproc->limit)
    goto bad;

  curproc->spnum = 1;
```

limit 변수를 확인하여 할당 받은 메모리 크기가 이를 넘을 경우 예외처리를 해준다. *exec()* 함수의 경우 스택용 페이지를 한 개 할당하기 때문에 *spnum*은 1로 설정해준다.

```
  // Clean up all other threads.
  thread_clear();
```

한 스레드에서 *exec()* 함수가 호출된 경우 나머지 스레드는 모두 정리되어야 하므로 *thread_clear()* 함수를 호출한다.

13. exec2()

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

스택용 페이지를 입력 받은 *stacksize* 인자만큼 할당받는다.

```
curproc->spnum = stacksize;
```

spnum 변수를 *stacksize* 변수의 값으로 설정해준다. 나머지 과정은 *exec()* 함수와 동일하다.

14. pmanager

```
 gets(cmd, CMDLENGTH);
       for(int i = 0; cmd[i]; i++) {
           if(cmd[i] == ' ' || cmd[i] == '\n' || cmd[i] == '\r') {
```

```
                arg[argnum++][j] = '\0';
                j = 0;
                continue;
            }
            arg[argnum][j++] = cmd[i];
        }
        arg[argnum][j] = '\0';
```

gets() 함수를 통해 사용자에게서 명령어를 입력 받는다. 이후 공백을 기준으로 문자열을 잘라서 *arg* 배열에 저장한다.

```
if(!strcmp(arg[0], "list")) {
            procdump2();
        }
```

list 명령어는 단순히 *procdump2()* 시스템 콜을 호출한다.

```
else if(!strcmp(arg[0], "kill")) {
            if(kill(atoi(arg[1])) == 0)
                printf(1, "kill success.\n");
            else
                printf(1, "kill fail.\n");

        }
```

kill 명령어는 *kill()* 함수를 호출하는데 이때 *kill()* 함수가 0을 반환하지 않는 경우 실패했다는 메시지를 띄워준다.

```
else if(!strcmp(arg[0], "execute")) {
            char *execargv[PATHLENGTH];
            execargv[0] = arg[1];
            if(fork() == 0) {
                if(exec2(arg[1], execargv, atoi(arg[2])) == -1)
                    printf(1, "execute fail.\n");
                exit();
            }
        }
```

execute 명령어는 *exec2()* 함수를 호출하는데 pmanager 위에 실행이 되면 안되므로 *fork()* 함수를 먼저 호출한 이후 자식 프로세스 위에 프로그램을 올려 실행한다. 마찬가지로 실패 시 메시지를 띄워준다.

```
else if(!strcmp(arg[0], "memlim")) {
            if(setmemorylimit(atoi(arg[1]), atoi(arg[2])) == 0)
                printf(1, "memlim success.\n");
            else
                printf(1, "mimlim fail.\n");
        }
```

memlim 명령어는 *setmemorylimit* 시스템 콜을 호출하며 실패 여부를 출력해준다.

```
else if(!strcmp(arg[0], "exit")) {
            break;
        }
```

exit 명령어는 명령어를 받는 반복문을 빠져나오게 하며 반복문 밖에 있는 *exit()* 함수를 통해 pmanager가 종료된다.

Result

1. 컴파일 및 실행과정

```
make CPUS=1
make fs.img
```

컴파일 시 CPUS=1 인자를 통해 CPU 개수를 하나로 설정해주었다.

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

부팅을 위한 명령어이다.

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 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
```

부팅이 정상적으로 완료된 모습이다. 이후 thread_test등의 테스트 파일 이름을 입력하면 정상적으로 테스트 코드가 실행된다.

2. Thread basic 테스트 코드

```
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 Child of threadChild of thread 4 start
thrChild of thread 1 start
 2 start
Child oead 0 start
f thread 3 start
Child of thread 4 end
Thread 4 end
Child of thread 2 end
Thread 2Child of thread 0 end
Child of thread 1 end
Child of thread 3 end
Thread 0 end
Thread 1 end
 end
Thread 3 end
Test 2 passed

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

All tests passed!
```

스레드의 기본적인 동작을 테스트하는 코드이다. 첫 번째 테스트는 단순히 스레드를 생성한 후 종료하는 테스트이다. 문제없이 동작하는 것을 볼 수 있다. 두 번째 테스트는 스레드를 생성한 후 *fork()* 함수를 호출하여 그 스레드의 자식 프로세스를 만드는 테스트이며 세 번째 테스트는 스레드에서 *malloc()* 함수를 호출하여 *sbrk()* 시스템 콜이 정상적으로 동작하는지 확인하는 테스트이다. 두 테스트 다 스레드 간의 출력이 섞이긴 하였지만 정상적으로 완료된 것을 볼 수 있다.

3. Thread kill 테스트 코드

```
$ thread_kill
Thread kill test start
Killing process 10
This code should beThis code should be executed 5 This code should be executed 5 times.
This code should be exeThis code should be executed 5 times.
 executed 5 times.
times.
cuted 5 times.
Kill test finished
```

스레드에서 *kill()* 시스템 콜이 정상적으로 동작하는지 테스트하는 코드이다. *fork()* 함수를 통해 프로세스를 하나 생성한 후 자식 프로세스에서는 실행돼서는 안 되는 코드를, 부모 프로세스에서는 자식 프로세스의 스레드를 죽이는 코드를 시작 루틴으로 갖는 스레드를 생성한다. 정상적으로 실행된다면 "This code should be executed 5 times" 라는 문구가 다섯 번 출력되어야 한다. 스레드 간의 출력이 섞이긴 하였지만 정상적으로 테스트가 완료된 것을 볼 수 있다.

3. Thread exit 테스트 코드

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

스레드에서 *exit()* 시스템 콜이 정상적으로 동작하는지 테스트하는 코드이다. 스레드를 여러 개 생성한 다음 가장 처음 만든 스레드에서만 *exit()*를 호출한다. 이 때 다른 스레드도 모두 종료되는지 테스트해준다. 정상적으로 모두 종료된 것을 볼 수 있다.

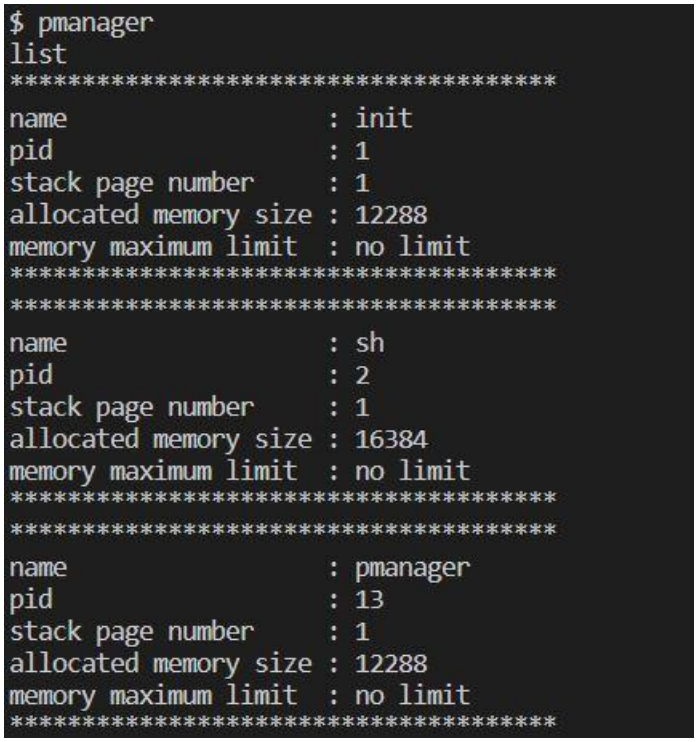
4. Thread exec 테스트 코드

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

스레드에서 *exec()* 시스템 콜이 정상적으로 동작하는지 테스트하는 코드이다. 스레드를 여러 개 생성한 다음 한 스레드에서 *exec()*를 호출한 다음 프로그램이 정상적으로 실행되는 지와 나머지 스레드가 종료되었는지 테스트한다. 정상적으로 동작하는 것을 볼

수 있다.

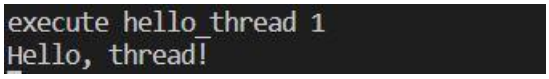
5. pmanager 실행 결과



list 명령어를 실행한 결과이다. 현재 *sleeping*, *runnable*, *running* 상태인 프로세스 정보를 출력해준다.

*pid*가 13인 pmanager을 kill하는 명령어이다. 정상적으로 pmanager이 종료되어 shell로 나간 것을 볼 수 있다.

존재하지 않는 *pid*를 kill하는 명령어를 입력해보았다. kill이 실패하였다는 메시지가 정상적으로 잘 출력된다.



hello_thread 파일을 실행하는 명령어이다. hello_thread 파일이 정상적으로 실행되어 출력되는 것을 볼 수 있다.

```
memlim 13 15000
memlim success.
list
************************************
name                 : init
pid                  : 1
stack page number    : 1
allocated memory size : 12288
memory maximum limit  : no limit
************************************
************************************
name                 : sh
pid                  : 2
stack page number    : 1
allocated memory size : 16384
memory maximum limit  : no limit
************************************
************************************
name                 : pmanager
pid                  : 13
stack page number    : 1
allocated memory size : 12288
memory maximum limit  : 15000
************************************
```

memlim 명령어를 통해 *pid* 13을 갖고 있는 pmanager의 메모리 제한을 설정하는 과정이다. 이후 list 명령어를 통해 프로세스 상태를 확인해보면 정상적으로 memory maximum limit가 설정된 것을 확인할 수 있다.

Trouble Shooting

1. 메인 스레드 설정 문제

```
  // Check if the curproc is main thread.
  if(curproc->parent->pid == curproc->pid)
    t->parent = curproc->parent;
  else
    t->parent = curproc;
```

thread_create() 함수에서 단순히 생성된 스레드의 부모를 현재 실행되고 있는 프로세스로 설정하니 어떤 스레드가 메인 스레드인지 구분하기 어려워지는 문제가 생겼다. 이를 해결하기 위하여 어떤 스레드에서 스레드를 생성하든 부모는 무조건 메인 스레드가 되도록 설정하여 문제를 해결하였다.

2. 메인 스레드 외에서의 스레드 생성 문제

```
  // Update sz variable of the other threads.
  for(t = ptable.proc; t < &ptable.proc[NPROC]; t++)
```

```
    if(t->pid == curproc->pid)
      t->sz = curproc->sz;
```

메인 스레드에서만 스레드를 생성한다면 계속 *sz* 변수가 업데이트 되므로 정상적으로 스레드에게 스택용 페이지를 할당해 줄 수 있다. 하지만 다른 스레드에서 스레드를 생성한다며 *sz* 변수가 자신이 생성될 당시의 값으로 고정되어 있어 오류가 나는 문제가 발생하였다. 이를 해결하기 위하여 스레드를 생성할 때마다 같은 프로세스 내 모든 스레드의 *sz* 변수를 업데이트 해주도록 설정하였고 오류가 해결되었다.

2. 스레드 wakeup 문제

```
  // Another thread might be sleeping in wait().
  for(t = ptable.proc; t < &ptable.proc[NPROC]; t++)
    if(t->pid == curproc->pid && t->tid != curproc->tid)
      wakeup1(t);
```

thread_exit() 함수는 *exit()* 함수를 이용하여 비슷하게 구현하였다. 그 과정에서 부모 프로세스(메인 스레드)가 *sleeping* 상태일 때 이를 깨워주도록 하였는데 메인 스레드가 아닌 스레드에서 *thread_join()*을 호출한 경우에는 정상적으로 동작하지 않았다. 따라서 어떤 스레드에서 기다리든 정상적으로 깨워 줄 수 있도록 *ptable*을 돌면서 *pid*가 같은 경우 모두 깨우도록 만들어 문제를 해결하였다.

3. 스레드 정리 관련 문제

```
if(t->parent->pid == curproc->pid){
      t->parent = initproc;
      if(t->state == ZOMBIE)
        wakeup1(initproc);
    }
```

exec(), exit() 함수를 호출하면 현재 스레드를 제외하고 다른 스레드를 정리하기 위하여 *thread_clear()* 함수를 호출하게 되는데 이때 단순히 스레드들을 종료시켰더니 그 스레드들의 자식 프로세스가 부모를 잃는 문제가 생겼다. 스레드 내에서 *fork()* 함수를 호출한다면 그 스레드의 정보를 복사하여 새로운 프로세스가 생성이 되는데 이 때 그 프로세스의 부모는 *fork()*를 호출한 스레드가 되므로 부모에 대한 추가적인 설정이 필요했다. *exit()* 함수에서 구현된 것과 동일하게 *init* 프로세스가 부모가 되도록 하여 그 프로세스들의 자원을 회수해주도록 구현하였으며 문제가 해결되었다.