과제 1

이름 : 김재형

학번 : 2018008068

목표

1. getgpid() (get grand parent process id) 시스템 콜 구현
2. User program 구현

처음 과제를 보고 느낀점 및 궁금증

과제를 읽고 다음과 같은 궁금증이 우선적으로 생겨났다.

1. gpid를 어디에 구현해야 하는지?
2. 시스템 콜을 을 추가해준다면 어디에다가 추가해야할지?
3. 일반프로그램을 추가하는것과 무엇이 다른지?
4. User 프로그램은 어디에다가 구현해야 하는지?

개념정리

● 시스템 콜 호출하는행위 를 어떻게 정의를 해야할까?

시스템 콜은 사용자 모드에서 실행되는 프로그램이
커널 모드에서 실행되는 운영 체제의 서비스를 요청할 수 있는 메커니즘이다.

사용자 프로그램은 직접적으로 커널 함수를 호출할 수 없으며, 대신 시스템 콜을 통해 안전하게 서비스를 요청한다.

시스템 콜이라는것은 일반 프로그램 이 하드웨어와 소통하기 위해서 운영체제에서 대신 호출해주는 함수 호출 이를 통해서 커널을 통하여 하드웨어 와 소통한다 이렇게 생각하면 좋을것같다.

● 시스템 콜 호출시 우리가 정의하는 커널이란?

소프트웨어와 하드웨어간의 소통 창구 같은 느낌이라고 이해하면 좋을것같다.

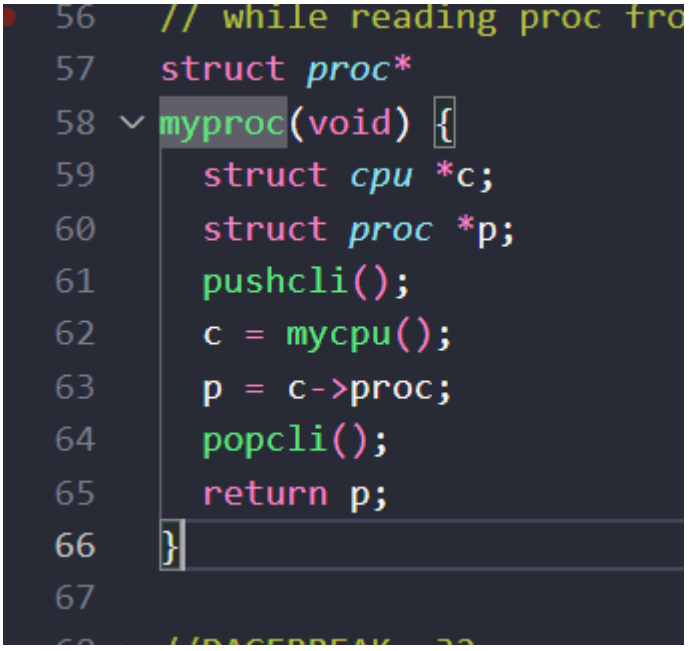
참고 자료

1. 조교님께서 실습시간에 직접 진행하신 예제 프로그램 구현
2. 교수님 proccess and thread 강의
3. getgpid는 아니지만 xv6에서 memset 구현한 예시 자료 [https://dmansp.tistory.com/20]

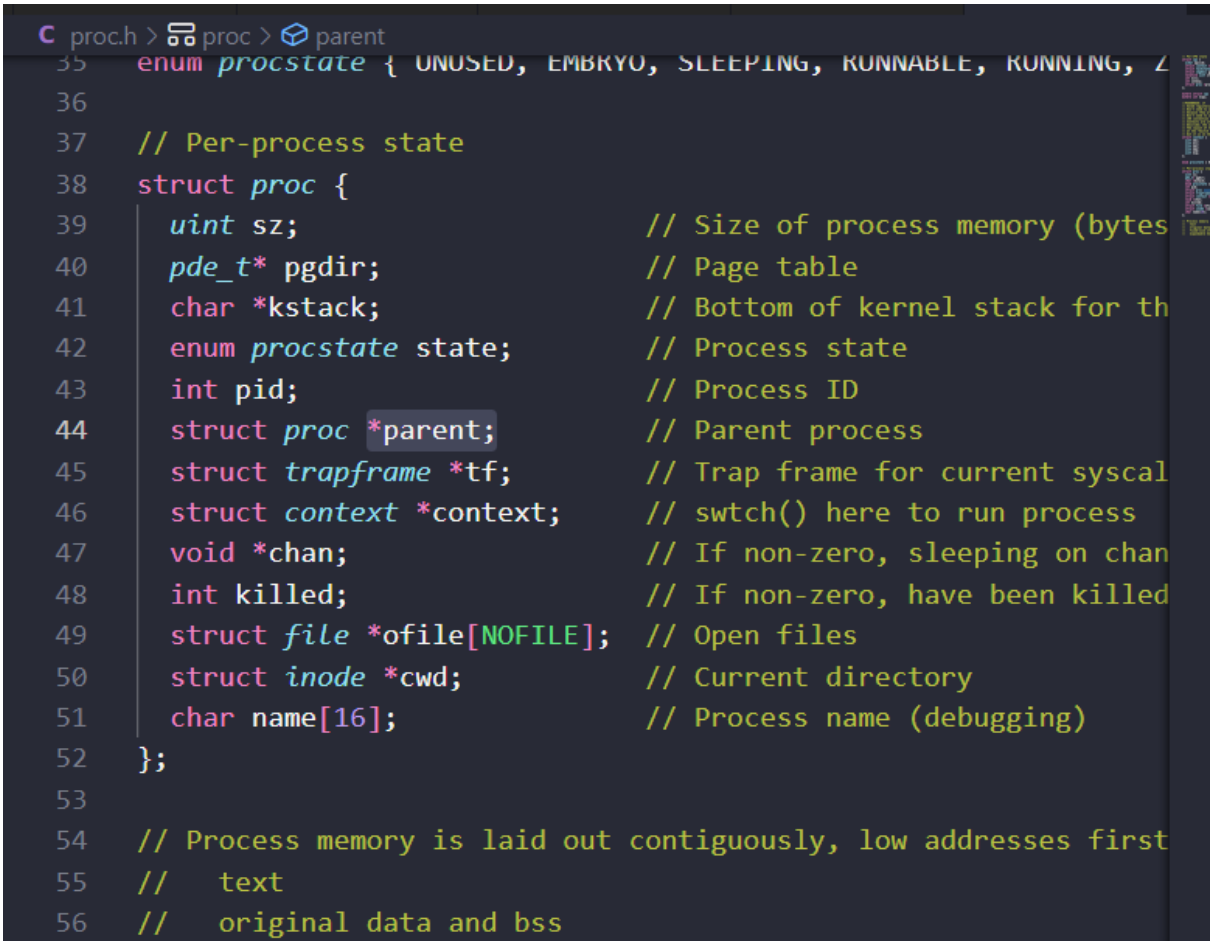
Design 명세

명세에서 요구하는 조건에 대한 구현계획

1. user.h에서 getgpid() 추가한다.
2. usys.S에서 getgpid() 추가
3. syscall.h에서 getgpid() 추가
4. syscall.c에서 getgpid() 추가
5. sysproc.c에서 getgpid시스템 콜 구현
6. my_userapp.c에서 main User 프로그램 구현
7. make clean → make 후에 qemu emulator 실행



```
56  // while reading proc fro
57  struct proc*
58  myproc(void) {
59    struct cpu *c;
60    struct proc *p;
61    pushcli();
62    c = mycpu();
63    p = c->proc;
64    popcli();
65    return p;
66  }
67
```



```
proc.h > proc > parent
35  enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, Z
36
37  // Per-process state
38  struct proc {
39    uint sz;                     // Size of process memory (bytes
40    pde_t* pgdir;                // Page table
41    char *kstack;                // Bottom of kernel stack for th
42    enum procstate state;        // Process state
43    int pid;                     // Process ID
44    struct proc *parent;         // Parent process
45    struct trapframe *tf;        // Trap frame for current syscal
46    struct context *context;     // swtch() here to run process
47    void *chan;                  // If non-zero, sleeping on chan
48    int killed;                  // If non-zero, have been killed
49    struct file *ofile[NOFILE];  // Open files
50    struct inode *cwd;           // Current directory
51    char name[16];               // Process name (debugging)
52  };
53
54  // Process memory is laid out contiguously, low addresses first
55  //   text
56  //   original data and bss
```

가장 중요한 부분은 proc.h 의 proc struct 와

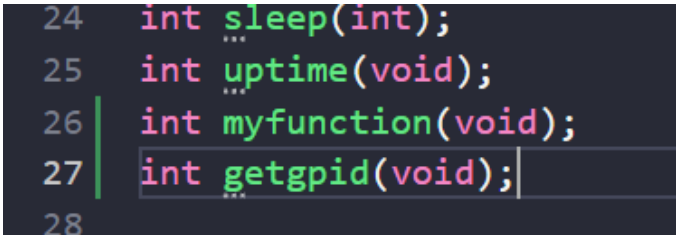
proc.c 의 proc struct 인것같다.

myproc 을 호출한 이후에

proc struct 내에서 구조체 내의 변수를 가져오는 방식으로 설계하였다.

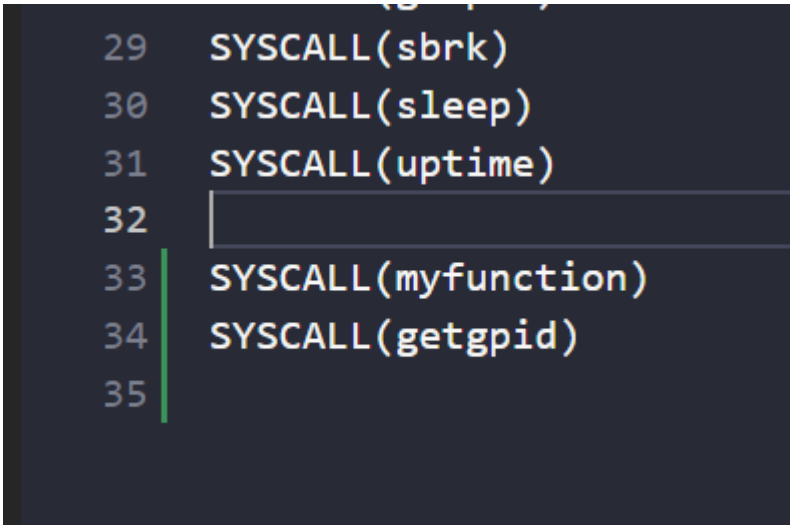
Implement

1. user.h에 getgpid 추가한 부분

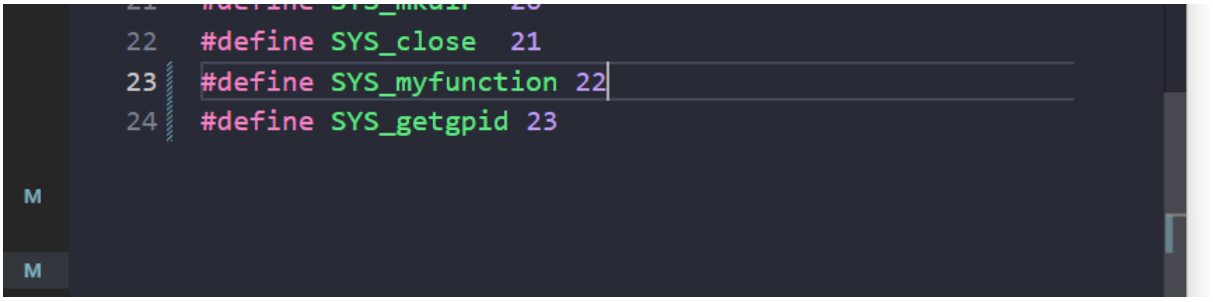


● User.h 의 파일 역할이란? **xv6의 시스템 호출 정의**

2. usys.S에서 getgpid() 추가



3. syscall.h에서 getgpid() 추가



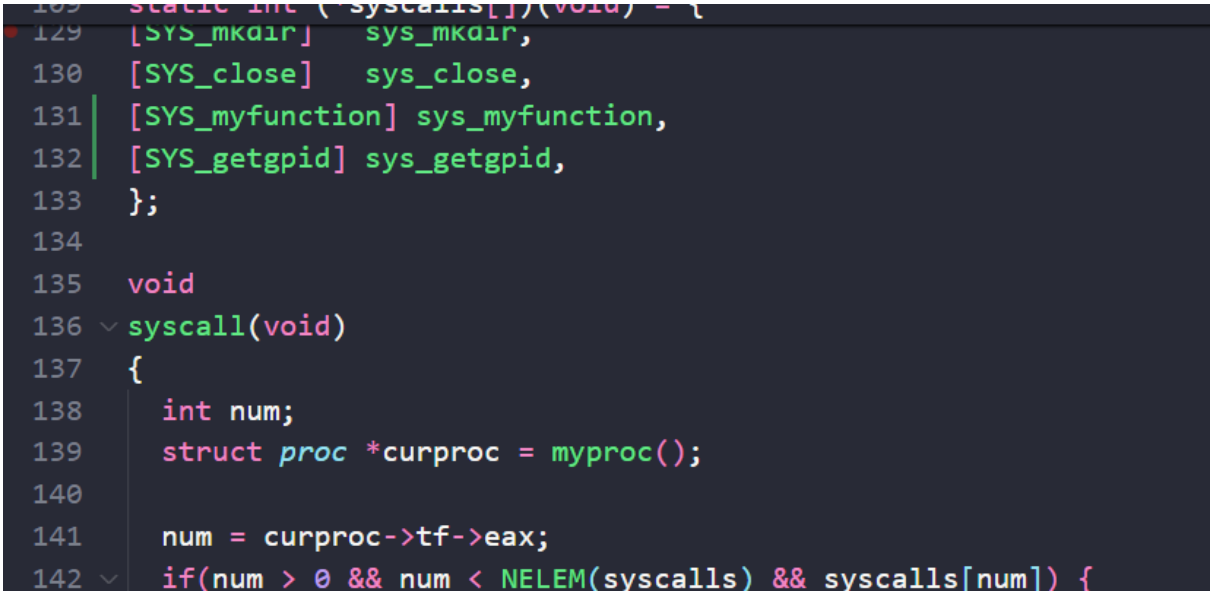
시스템 호출 번호 매핑 → 새 시스템 호출을 위해 새로운 매핑 추가해야 함

syscall.h의 주요기능 정리

1. 시스템 콜 번호 정의
2. 매크로 정의
3. 타입 정의 및 함수 프로토타입

4. syscall.c에서 getgpid() 추가

시스템 호출 인수를 구문 분석하는 함수 및 실제 시스템 호출 구현에 대한 포인터



5. sysproc.c에서 getgpid시스템 콜 구현

프로세스 관련 시스템 호출 구현 → 시스템 호출 코드 추가해야 함

```
// return grand parent pid
int sys_getgpid(void)
{
  struct proc *curproc = myproc();
  if (curproc->parent == 0) return -1; // 부모가
  if (curproc->parent->parent == 0) return -1; //

  return curproc->parent->parent->pid; // 부모의
}
```

my process 구조체를 curproc 라는 포인터로 받은이후에

부모가 존재하는지 존재하지 않는지 확인을 한다.

부모도 존재하고 조부모 역시 존재를 할시 , pid를 return 하는 함수를 구현할수 있었다.

6. my_userapp.c에서 main User 프로그램 구현

```
3   #include "user.h"
4
5   int main(int argc, char * argv[])
6   {
7       int gpid;
8       gpid = getgpid();
9
10      printf(1, "My student id is 2018008068\n");
11      printf(1, "My pid is %d\n", getpid());
12      printf(1, "My gpid is %d\n", gpid);
13      exit();
14  }
```

7. makefile 추가

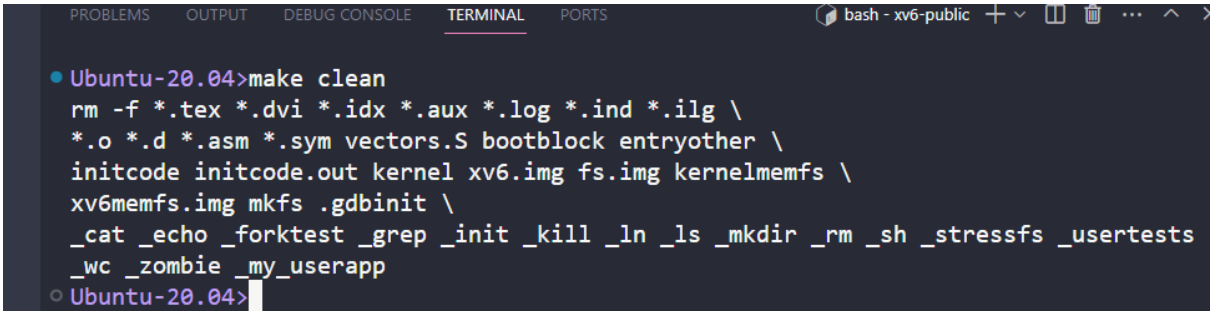
```
251
252 EXTRA=\
253     mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
254     ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
255     printf.c umalloc.c _my_userapp.c\
256     README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
257     .gdbinit.tmpl gdbutil\
258
259 dist:
```

```
169 UPROGS=\
170     _cat\
171     _echo\
172     _forktest\
173     _grep\
174     _init\
175     _kill\
176     _ln\
177     _ls\
178     _mkdir\
179     _rm\
180     _sh\
181     _stressfs\
182     _usertests\
183     _wc\
184     _zombie\
185     _my_userapp\
186
```

Makefile 파일에 대한 설명

Result

1. Make clean



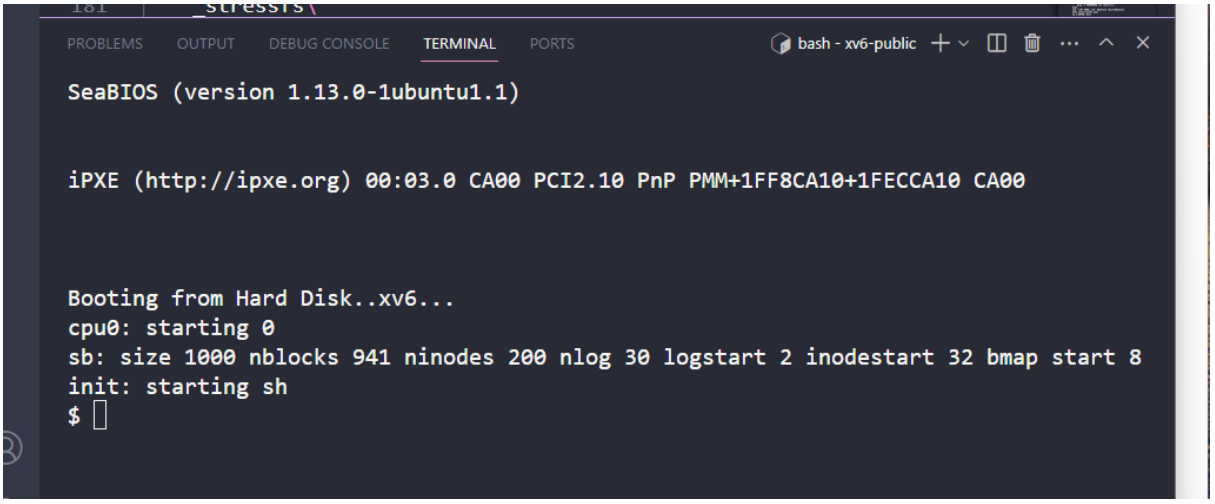
2. make

```
 uart.o vectors.o vm.o gpid_syscall.o  -b binary initcode entryother
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0381589 s, 134 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.0001027 s, 5.0 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
417+1 records in
417+1 records out
213776 bytes (214 kB, 209 KiB) copied, 0.0011405 s, 187 MB/s
```

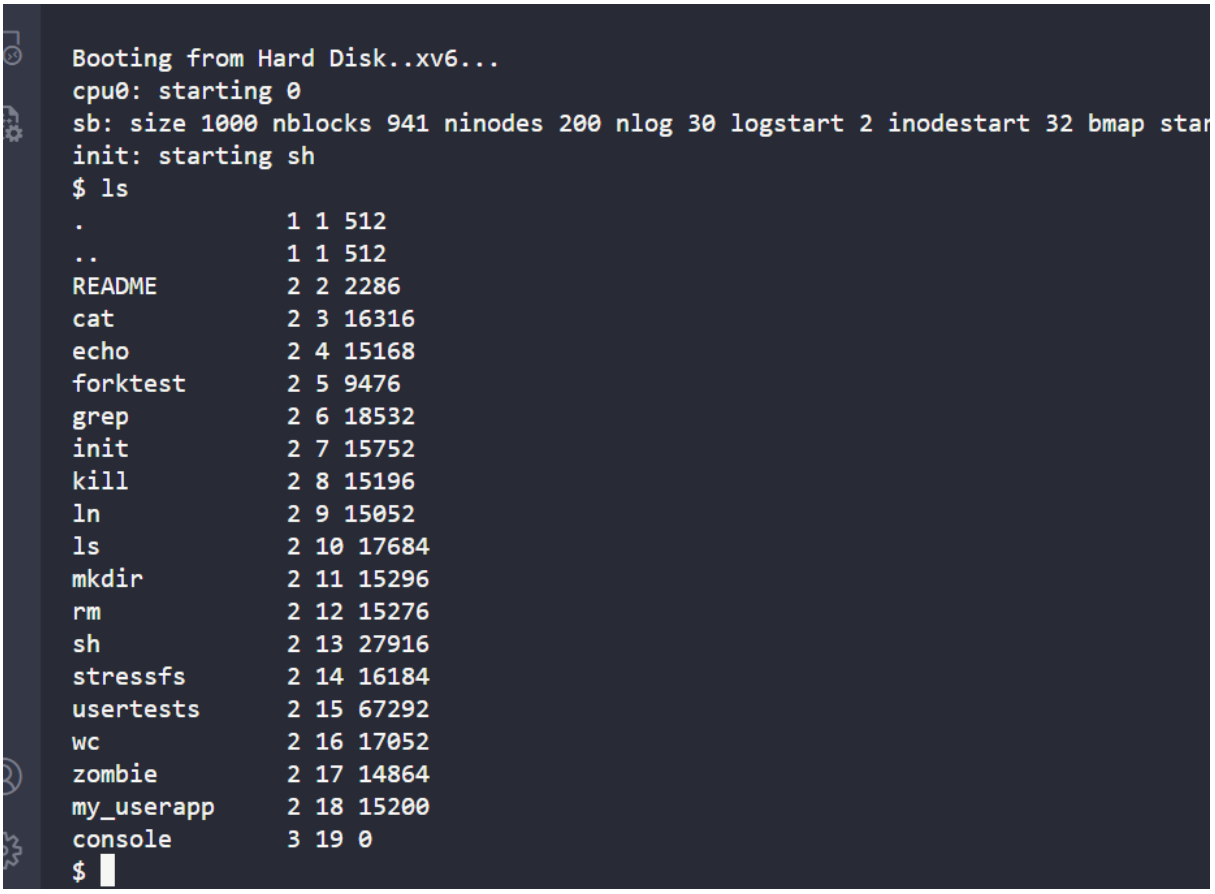
3. make fs.img

```
ld -m    elf_i386 -N -e main -Ttext 0 -o _my_userapp my_userapp.o ulib.o usys.o pri
ntf.o umalloc.o
objdump -S _my_userapp > my_userapp.asm
objdump -t _my_userapp | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > my_userapp.sym
./mkfs fs.img README _cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh
_stressfs _usertests _wc _zombie _my_userapp
nmeta 59 (boot, super, log blocks 30 inode blocks 26, bitmap blocks 1) blocks 941 t
otal 1000
balloc: first 698 blocks have been allocated
balloc: write bitmap block at sector 58
Ubuntu-20.04>
```

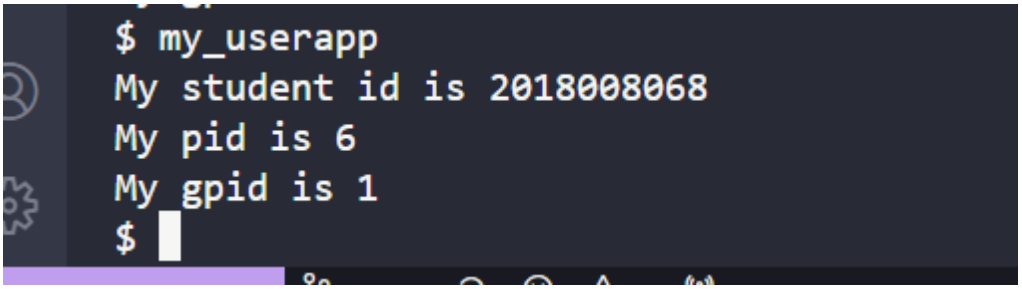
4. bootxv6.sh



5. ls 를 통하여 전체 프로그램 확인



6. my_userapp실행결과



Trouble shooting

원래는

my_userapp.c U | gpid_syscall.c U | usys.S M | syscall.h M | syscall.c M

gpid_syscall.c > ...

```c
#include "types.h"
#include "defs.h"

// Simple system call
int myfunction(void)
{
    return 1;
//    int a= 1;
//   struct proc *curproc = myproc();
//   if (curproc->parent == 0) return -1; // 부모가 없으면 -1 반환
//   if (curproc->parent->parent == 0) return -1; // 부모의 부모가 없으면 -1

//   return curproc->parent->parent->pid; // 부모의 부모의 PID 반환
}

int sys_myfunction(void){
    return myfunction();
}

```

다음과 같이 gpid_syscall.c 에서 wrapper function 과 myfunction 으로 system 콜을 구현할려 하였으나

무슨이유인지 모르겠으나, 컴파일 에러가 나왔다. 따라서 proc.c 에서 직접시스템 콜을 구현하였다.