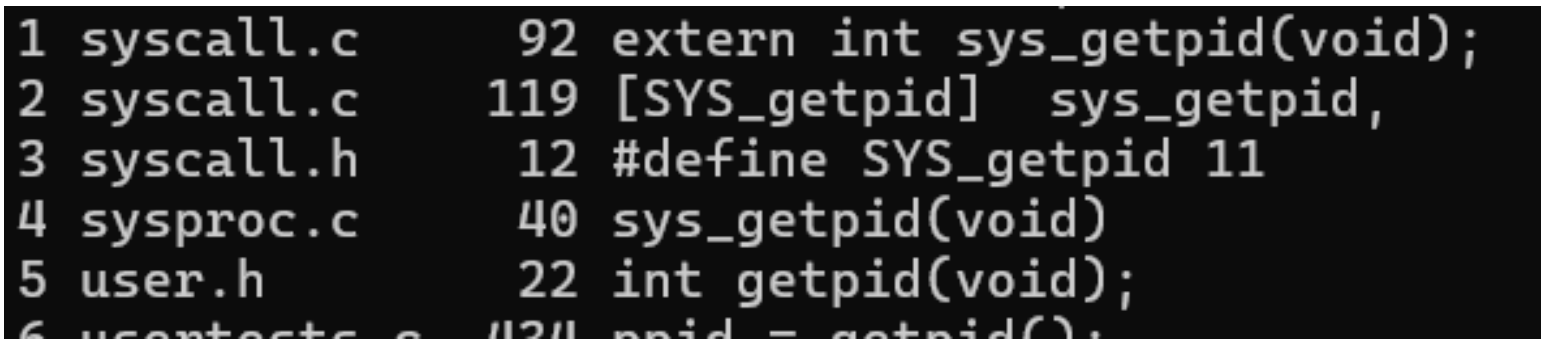
project01 wiki - 2022028522

---

1. Design

> 📌 명세에서 요구하는 조건에 대한 구현 계획을 서술합니다.

- `getgpid()`를 구현하기 위해 과제 명세에 나와있는 Tips 부분에서 cscope를 활용해 `getpid()` 함수를 찾아보았습니다.



이 중 `sysproc.c` 파일에 들어가 봤더니 시스템 콜인 `sys_getpid(void)` 함수의 코드를 확인할 수 있었습니다.

```
39 int
40 sys_getpid(void)
41 {
42   return myproc()->pid;
43 }
```

여기 나와있는 `myproc()`이 무엇인지 알아보기 위해 cscope를 이용해 `myproc()`을 또 찾아보았더니, xv6-public의 `proc.c`에 `myproc()` 함수에 대한 정보가 나와있었습니다.

```
55 // Disable interrupts so that we are not rescheduled
56 // while reading proc from the cpu structure
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 }
```

이 `myproc(void)` 함수의 return type은 `proc` 구조체의 포인터 형태이기 때문에, 다시 cscope을 활용하여 `proc` 구조체에 대한 정보를 찾아보았더니, `proc.h`에 `proc` 구조체에 대한 정보가 들어있었습니다.

```
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 this process
```

```
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 syscall
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 };
```

마침 proc 구조체의 필드에는 parent process를 나타내는 포인터인 `struct proc *parent`와 프로세스의 아이디를 나타내는 `int pid`가 있었기 때문에 이것을 활용하여 `myproc()->parent->parent->pid`의 방식으로 `getgpid()`를 구현할 수 있겠다는 생각을 하였습니다.

- 시스템콜을 추가하고 유저 프로그램에서 사용될 수 있게 하기 위해 다음과 같은 과정을 통해 구현하고자 하였습니다.
  - 새로운 시스템 콜을 정의하기 위해 `prac_syscall.c` 파일을 작성합니다.
  - `prac_syscall.o`를 컴파일하도록 Makefile을 수정합니다.
  - `defs.h` 파일에 `getgpid()` 함수를 추가하여 다른 파일에서 이 함수를 사용할 수 있도록 합니다.
  - 시스템 콜을 호출하기 위한 래퍼 함수인 `sys_getgpid()`을 작성합니다.
  - `syscall.h`에 새로운 시스템 콜 번호를 정의하고, `SYS_getgpid`을 추가하여 새로운 시스템 콜을 등록합니다.
  - `syscall.c`에 `sys_getgpid()` 함수를 등록하여 시스템 콜이 호출될 때 실행되도록 합니다.
  - `user.h`에 `getgpid()` 함수를 등록하여 사용자 프로그램에서 이를 호출할 수 있도록 합니다.
  - 사용자 프로그램에서 시스템 콜을 호출할 수 있도록 `usys.S`에 매크로를 추가합니다.
  - 새로운 시스템 콜을 사용하는 사용자 프로그램인 `project01.c`를 추가하여 새로운 시스템 콜을 호출합니다.
  - 새로운 사용자 프로그램을 빌드할 수 있도록 Makefile을 수정합니다.

---

2. Implement

> 📌 새롭게 구현하거나 수정한 부분이 기존과 어떻게 다른지, 해당 코드의 목적이 무엇인지에 대해 구체적으로 서술합니다.

- `prac_syscall.c` 파일을 xv6-public 디렉토리에 추가하여 `getgpid()` 함수의 구현하였습니다. `getgpid()` 함수의 구현은 아래와 같습니다.

```
$ vim prac_syscall.c

1 #include "types.h"
2 #include "defs.h"
3 #include "param.h"
4 #include "mmu.h"
5 #include "proc.h"
6
7 //Simple system call
8 int
```

```
9 getgpid(char *str)
10 {
11  cprintf("%s\n", str);
12  return myproc()->parent->parent->pid;
13 }
14
15 //Wrapper for my_syscall
16 int
17 sys_getgpid(void)
18 {
19  char *str;
20  //Decode argument using argstr
21  if(argstr(0, &str) < 0)
22      return -1;
23  return getgpid(str);
24 }
```

`getgpid()` 함수는 문자열을 받아들이고, `cprintf` 함수를 사용하여 입력된 문자열을 커널 로그에 출력한 후 `myproc()->parent->parent->pid`를 사용하여 현재 프로세스의 조부모 프로세스의 pid를 반환합니다.

`sys_getgpid` 함수는 새로운 시스템 콜을 호출하는 래퍼 함수로, `argstr()`을 사용하여 사용자로부터 받은 문자열 인자를 가져옵니다. 문자열 인자가 없는 경우 `-1`을 반환하고, 그렇지 않으면 `getgpid()` 함수를 호출하고 그 결과를 반환합니다.

- 다음으로 `prac_syscall.o`를 컴파일하도록 Makefile을 수정합니다. 이를 위해 `OBJS`에 `prac_syscall.o`를 추가합니다.

```
$ vim Makefile

1 OBJS = \
2 bio.o\
3 console.o\
…
28 vectors.o\
29 vm.o\
30 prac_syscall.o\      // 새로 추가된 부분
```

- `defs.h` 파일에 `getgpid()` 함수를 추가하여 이 헤더 파일을 include하는 다른 파일에서 이 함수를 사용할 수 있도록 합니다.

```
$ vim defs.h

1 struct buf;
2 struct context;
3 struct file;
…
186 int copyout(pde_t*, uint, void*, uint);
187 void clearpteu(pde_t *pgdir, char uva);
188
189 //prac_syscall.c
190 int getgpid(char);        // 새로 추가된 부분
```

- `syscall.h`와 `syscall.c`를 수정합니다. `syscall.h`에 `SYS_getgpid`을 추가하여 새로운 시스템 콜 번호를 정의하고 새로운 시스템 콜을 등록하고, `syscall.c`에 `sys_getgpid()` 함수를 등록하여 시스템 콜이 호출될 때 실행되도록 합니다.

```
$ vim syscall.h
```

```
$ vim syscall.c
```

```
1 // System call numbers
2 #define SYS_fork 1
3 #define SYS_exit 2
…
22 #define SYS_close 21
23 #define SYS_getgpid 22
        // 새로 추가된 부분
```

```
1 #include "types.h"
2 #include "defs.h”
…
105 extern int sys_uptime(void);
106 extern int sys_getgpid(void);
        // 새로 추가된 부분
…
129 [SYS_close] sys_close,
130 [SYS_getgpid] sys_getgpid,
        // 새로 추가된 부분
131 };
```

- `user.h`에 `getgpid()` 함수를 등록하여 사용자 프로그램에서 이를 호출할 수 있도록 합니다.

```
$ vim user.h

1 struct stat;
2 struct rtcdate;
3
4 // system calls
5 int fork(void);
…
25 int uptime(void);
26 int getgpid(char*);        // 새로 추가된 부분
```

- 사용자 프로그램에서 시스템 콜을 호출할 수 있도록 `usys.S`에 매크로를 추가합니다.

```
$ vim usys.S

1 #include “syscall.h”
2 #include “trap.h”
3
4 #define SYSCALL(name) \
…
30 SYSCALL(sleep)
31 SYSCALL(uptime)
32 SYSCALL(getgpid)        // 새로 추가된 부분
```

- 새로운 시스템 콜을 사용하는 사용자 프로그램인 project.c를 작성합니다.

```
$ vim project.c

1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4
5 int
6 main(int argc, char *argv[])
7 {
8   char *buf = "My student id is 2022028522";
9   int pid, gpid;
10  pid = getpid();
11  gpid = getgpid(buf);
```

```
12  printf(1, "My pid is %x\n", pid);
13  printf(1, "My gpid is %x\n", gpid);
14  exit();
15 }
```

`buf` 변수는 문자열을 가리키는 포인터로, `"My student id is 2022028522"`라는 문자열을 가리킵니다. 이는 나중에 getgpid 함수의 인자로 들어가 커널 로그에 출력하는 역할을 합니다.

`pid`와 `gpid` 변수는 각각 현재 프로세스의 PID와 조부모 프로세스의 PID를 저장하는 데 사용되는 변수로, `getpid()` 함수와 `getgpid(buf)` 함수를 호출하여 각각의 변수에 저장합니다.

`printf(1, "My pid is %x\n", pid)`와 `printf(1, "My gpid is %x\n", gpid)`를 호출하여 프로세스의 PID와 조부모 프로세스의 PID를 출력합니다.

마지막으로 `exit()` 함수를 호출하여 프로그램을 종료합니다.

- 새로운 사용자 프로그램을 빌드할 수 있도록 Makefile을 수정합니다.

```
$ vim Makefile
…
169 UPROGS=\
170 _cat\
171 _echo\
…
184 _zombie\
185 _project01\        // 새로 추가된 부분
…
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 project01.c\        // 새로 추가된 부분
```

---

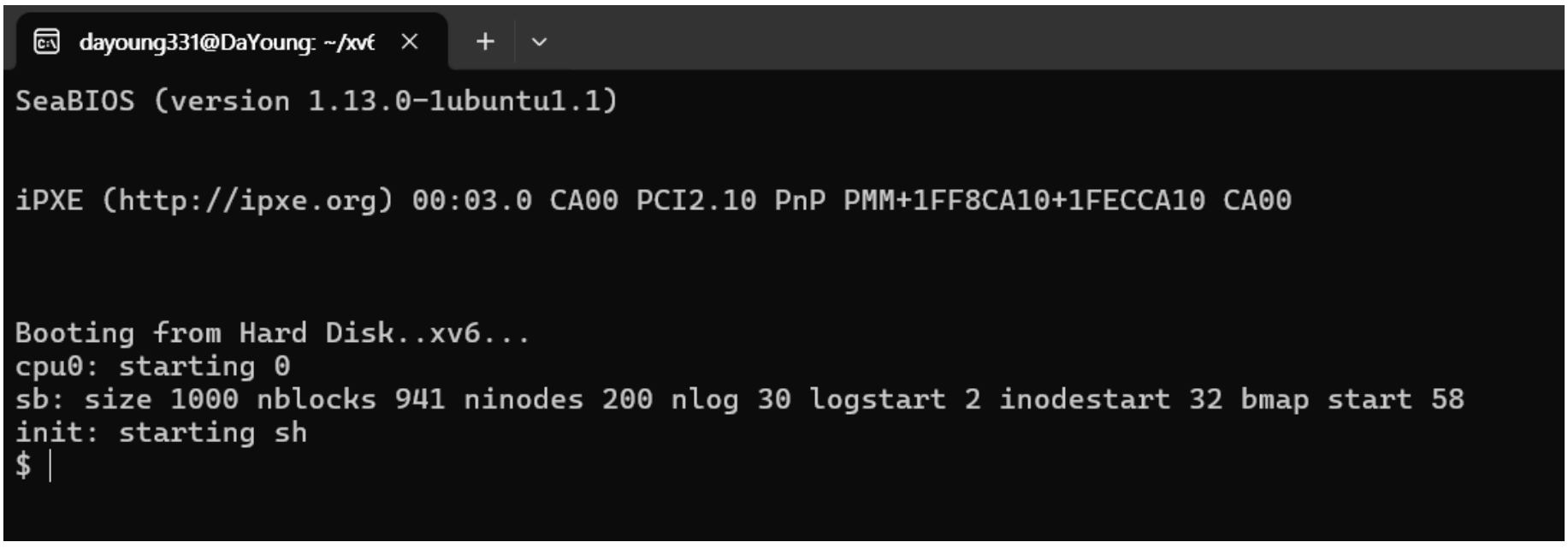
3. Result

📌 컴파일 및 실행 과정과, 해당 명세에서 요구한 부분이 정상적으로 동작하는 실행 결과를 첨부하고, 동작 과정에 대해 설명합니다.

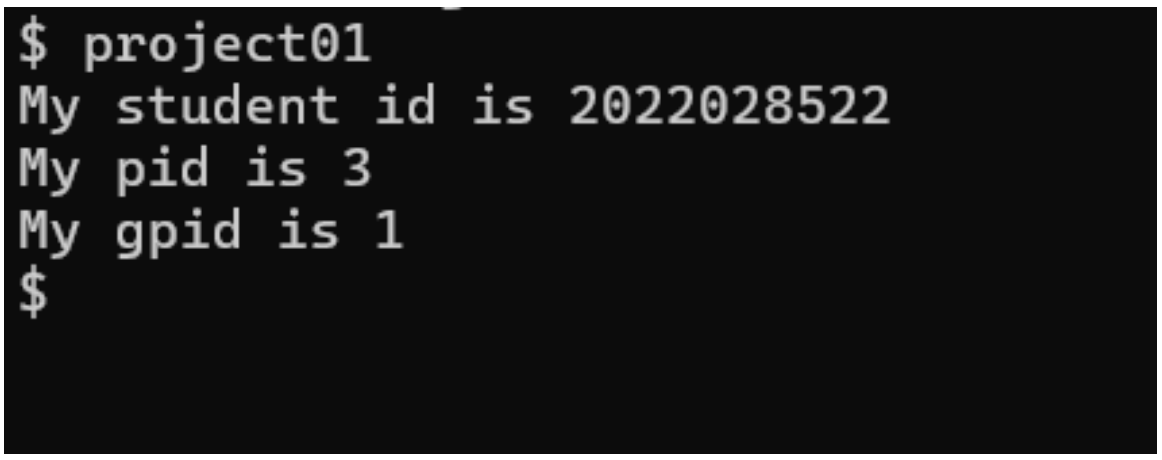
- xv6-public 디렉토리 안에서 다음과 같은 명령어를 차례대로 입력합니다.

```
$ make clean
$ make
$ make fs.img
…
$ ./bootxv6.sh
```

- 마지막 명령어를 통해 QEMU에서 xv6 운영 체제가 부팅되면 셸 프롬프트가 나타납니다. 그 사진은 아래에 첨부합니다.



- 이 상태에서 셸에 project01을 입력하면, 해당 사용자 프로그램이 실행됩니다. 셸에 proeject01을 입력하였을 때 나타나는 결과물은 다음과 같습니다.



- 작동 방식을 설명하기 위해 `project01.c` 코드를 다시 불러오겠습니다.

```
$ vim project.c

1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4
5 int
6 main(int argc, char *argv[])
7 {
8   char *buf = "My student id is 2022028522";
9   int pid, gpid;
10  pid = getpid();
11  gpid = getgpid(buf);
12  printf(1, "My pid is %x\n", pid);
13  printf(1, "My gpid is %x\n", gpid);
14  exit();
15 }
```

`main` 함수 내부에서는 먼저 문자열 `buf`를 정의합니다. 여기서는 과제 명세에서 과제를 수행한 사람의 학번을 출력하라는 매뉴얼이 있었으므로 `buf`를 `"My student id is 2022028522"`로 정의합니다. 다음으로 `getpid()` 함수와 `getgpid()` 함수를 호출하여 현재 프로세스의 PID와 현재 프로세스의 조부모 프로세스의 PID를 얻어 `pid` 변수와 `gpid` 변수에 저장합니다. `getpid()` 함수를 호출할 때에는 문자열 `buf`가 시스템 콜에 인자로 전달됩니다. 마지막으로 `printf` 함수를 사용하여 `pid`와 `gpid`를 출력합니다.

---

4. Trouble Shooting

> 📌 과제 수행 과정에서 겪은 문제가 있다면, 해당 문제와 해결 과정을 서술합니다. 해결하지 못했다면 어떤 문제였고 어떻게 해결하려 했는지 서술합니다.

- 맨 처음에 `make` 명령어를 실행했을 때에는 다음과 같은 오류 메시지가 뜨며 `make`가 되지 않았었습니다.

```
prac_syscall.c: In function ‘getgpid’:
prac_syscall.c:13:17: error: dereferencing pointer to incomplete type ‘struct proc’
   13 |  return myproc()->parent->parent->pid;
      |                 ^~
prac_syscall.c:14:1: error: control reaches end of non-void function [-Werror=return-type]
   14 | }
      | ^
```

위의 Design 부분에서 알 수 있듯이, `proc` 구조체에 대한 정보는 `proc.h`와 `proc.c`에서 찾아볼 수 있었기 때문에, 헤더에 `#include "proc.h"`를 추가해주었습니다.

- 다시 `make` 명령어를 실행해 보았더니, 이번에는 다음과 같은 오류 메시지가 발생하였습니다.

```
proc.h:5:20: error: field ‘ts’ has incomplete type
    5 |   struct taskstate ts;         // Used by x86 to find stack for interrupt
      |                    ^~
proc.h:6:22: error: ‘NSEGS’ undeclared here (not in a function)
    6 |   struct segdesc gdt[NSEGS];   // x86 global descriptor table
      |                      ^~~~~
proc.h:13:24: error: ‘NCPU’ undeclared here (not in a function)
   13 | extern struct cpu cpus[NCPU];
      |                        ^~~~
proc.h:49:22: error: ‘NOFILE’ undeclared here (not in a function)
   49 |   struct file *ofile[NOFILE];  // Open files
      |                      ^~~~~~
make: *** [<builtin>: prac_syscall.o] Error 1
```

위의 첫 번째 에러 메시지를 통해 `taskstate`라는 구조체인 `ts`를 선언할 때 오류가 있었음을 예상할 수 있었습니다. 그래서 cscope를 통해 `taskstate`를 검색해 보았더니, `mmu.h`에 `taskstate` 구조체에 대한 정보가 들어있는 것을 확인하였습니다.

```
// mmu.h
...

106 // Task state segment format
107 struct taskstate {
108 uint link;         // Old ts selector
109 uint esp0;         // Stack pointers and segment selectors
110 ushort ss0;        //   after an increase in privilege level
111 ushort padding1;
112 uint *esp1;
113 ushort ss1;
114 ushort padding2;
115 uint *esp2;
116 ushort ss2;
117 ushort padding3;
118 void *cr3;         // Page directory base
119 uint *eip;         // Saved state from last task switch
120 uint eflags;
121 uint eax;          // More saved state (registers)
```

```
122 uint ecx;
123 uint edx;
124 uint ebx;
125 uint *esp;
126 uint *ebp;
127 uint esi;
128 uint edi;
129 ushort es;         // Even more saved state (segment selectors)
130 ushort padding4;
131 ushort cs;
132 ushort padding5;
133 ushort ss;
134 ushort padding6;
135 ushort ds;
136 ushort padding7;
137 ushort fs;
138 ushort padding8;
139 ushort gs;
140 ushort padding9;
141 ushort ldt;
142 ushort padding10;
143 ushort t;          // Trap on task switch
144 ushort iomb;       // I/O map base address
145 };
```

그래서 `mmu.h` 헤더를 추가하면 해당 오류가 사라지지 않을까 싶어 `prac_syscall.c`의 맨 처음 부분에 `#include "mmu.h"`를 추가해주었습니다. 다시 `make` 명령어를 실행했을 때, 다음과 같이 오류 메시지 2개가 없어진 것을 확인할 수 있었습니다.

```
In file included from prac_syscall.c:6:
proc.h:13:24: error: ‘NCPU’ undeclared here (not in a function)
   13 | extern struct cpu cpus[NCPU];
      |                        ^~~~
proc.h:49:22: error: ‘NOFILE’ undeclared here (not in a function)
   49 |   struct file *ofile[NOFILE];  // Open files
      |                      ^~~~~~
make: *** [<builtin>: prac_syscall.o] Error 1
```

이번에는 cscope로 `NCPU`를 검색해보았더니, `param.h`라는 파일에 `NCPU`에 대한 정보가 있는 것을 확인하였습니다. 추가로 위의 두 번째 오류인 `NOFILE`에 대한 정보도 있는 것을 확인할 수 있습니다.

```
// param.h

1 #define NPROC        64  // maximum number of processes
2 #define KSTACKSIZE 4096  // size of per-process kernel stack
3 #define NCPU          8  // maximum number of CPUs
4 #define NOFILE       16  // open files per process
5 #define NFILE       100  // open files per system
6 #define NINODE       50  // maximum number of active i-nodes
7 #define NDEV         10  // maximum major device number
8 #define ROOTDEV       1  // device number of file system root disk
9 #define MAXARG       32  // max exec arguments
10 #define MAXOPBLOCKS  10  // max # of blocks any FS op writes
11 #define LOGSIZE      (MAXOPBLOCKS3)  // max data blocks in on-disk log
12 #define NBUF         (MAXOPBLOCKS3)  // size of disk block cache
13 #define FSSIZE       1000  // size of file system in blocks
```

위의 방식과 비슷하게 이번에는 `prac_syscall.c`에 `#include "param.h”`를 추가해주었습니다.

- 이렇게 세 개의 헤더를 추가해주고 나니 더 이상의 오류 메시지 없이 정상적으로 프로그램이 작동하였습니다.
