Project 01 – 1

- getppid() system call을 구현해야 함. getppid() system call은 진행중인 프로세스의 부모 프로세스 ID를 가져오는 기능을 함.

1 What to do

1.1 일단 parent process의 ID에 접근하기 위해서는 xv6에서 process ID에 접근하는 방식을 먼저 알아야 한다.

1.2 그 안에서 부모 프로세스에 접근할 수 있는 방식을 찾아낸다.

1.3 실습 시간에 배운 Steps to add a new system call을 통해 커널에서 해주어야 하는 작업을 해준다.

1.4 User가 쓸 수 있는 프로그램을 정의해주고 make될 수 있도록 해준다.

1.5 Make를 통해 오류를 확인하고 xv6를 실행시켜 확인한다.

2 How to do

```
Cscope 태그: getpid
   #   줄  파일 이름 / 콘텍스트 / 줄
   1    92  syscall.c <<<unknown>>>
            extern int sys_getpid(void);
   2   120  syscall.c <<<unknown>>>
            [SYS_getpid]  sys_getpid,
   3    12  syscall.h <<<unknown>>>
            #define SYS_getpid 11
   4    40  sysproc.c <<<unknown>>>
            sys_getpid(void)
   5    22  user.h <<<unknown>>>
            int getpid(void);
   6   434  usertests.c <<<unknown>>>
              ppid = getpid();
   7  1498  usertests.c <<<unknown>>>
              ppid = getpid();
   8    28  usys.S <<<unknown>>>
            SYSCALL(getpid)
```

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

```
11 #define SYS_dup    10
12 #define SYS_getpid 11
13 #define SYS_sbrk   12
```

```
119 [SYS_dup]    sys_dup,
120 [SYS_getpid] sys_getpid,
121 [SYS_sbrk]   sys_sbrk,
```

```
135 void
136 syscall(void)
137 {
138   int num;
139   struct proc *curproc = myproc();
140
141   num = curproc->tf->eax;
142   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
143     curproc->tf->eax = syscalls[num]();
144   } else {
145     cprintf("%d %s: unknown sys call %d\n",
146             curproc->pid, curproc->name, num);
147     curproc->tf->eax = -1;
148   }
149 }
```

2.1 cscope를 활용하여 process ID를 가져오는 기능을 가진 system call인 getpid()를 먼저 분석한다. cscope로 getpid를 찾아보면 6,7번에 있는 usertest.c를 제외하고 총 6개 구간에서 구현되어 있는것을 확인할 수 있다. 앞에서부터 보면 syscall.c라고 하는 InterruptHandler 안의 syscall(void)함수에서 해당하는 번호의 system call이 호출되면 interrupt를 발생시키도록 되어 있다. 그리고 syscall.c의 윗부분에 sys_getpid()가 지정되어 있고 해당 인덱스는 syscall.h안에 숫자 11로 지정되어있다. 여기서 interupt가 발생하면 sysproc.c 안에 선언되어 있는 sys_getpid(void)라 선언된 Wrapper function으로 이동한다. sys_getpid(void)는 sysproc.c 파일 안에서 함수로 구현되어있다. 직접 들어가서 확인해 보면 int형으로 myproc()의 pid값을 return하는 형태다. 이 함수를 통해 process ID에 접근함을 알 수 있다.

2.2 myproc()의 구조를 확인하면 부모 프로세스에 접근할 수 있는 방식을 확인할 수 있다. 들어가서

```
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 }
```

```
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 };
```

확인하면 mycpu()라는 함수의 proc 구조체를 return하는것을 확인할 수 있다. proc 구조체에 들어가서 다시 확인하면, 다양한 proc의 요소가 있는데 이 중에서 우리가 확인해야할 것은 정수로 선언된 pid값과 또 다른 proc 구조체로 선언된 parent의 주소값이다. xv6에서 각각의 process는 부모

process의 포인터도 가지고 있다는 것을 알 수 있다. 결국 sys_getpid(void)함수는 myproc()의 mycpu()의 proc 구조체안의 pid를 return하는 것이다. 그리고 부모 process의 pid에 접근하려면 myproc()의 parent 구조체의 pid로 접근하면 된다는 것을 알 수 있다.

2.3 이제 getpid system call을 참고하여 커널 수준에서 해주어야 하는 작업들을 해주면 된다. 일단 함수는 새로 c 파일을 만들어도 되지만, getpid처럼 sysproc.c 파일 안에 만들어주겠다. 이 함수는

```
45 int
46 sys_getppid(void)
47 {
48     return myproc()->parent->pid;
49 }
50
```

```
23 #define SYS_myfunction 22
24 #define SYS_getppid 23
```

```
131 [SYS_myfunction] sys_myfunction,
132 [SYS_getppid] sys_getppid,
133 };
```

```
25 int uptime(void);
26 int myfunction(char*
27 int getppid(void);
```

```
106 extern int sys_myfunction(void);
107 extern int sys_getppid(void);
```

```
31 SYSCALL(uptime)
32 SYSCALL(myfunction)
33 SYSCALL(getppid)
```

wrapper 함수로써 작용할 것이다. 그리고 해당 함수가 system call로서 작용할 수 있도록 syscall.h에 번호로 지정 해줄 것이다. 맨 끝에 있는 23을 선택했다. syscall.c에는 내가 짠 함수가 interrupt를 발생시킬 수 있도록 getpid와 유사하게 만들어 줄 것이다. 그리고 user.h에 user program에 보일 수 있도록 선언 해 주고, system call을 user program에서 사용할때 실행되도록 usys.S에 지정해 준다.

2.4 이제 user가 쓸 수 있는 새로운 파일을 project01.c라는 이름으로 만들어주고, 과제 형식처럼 출

```
project01.c (~/OS/xv6-public) - VIM
 1 #include "types.h"
 2 #include "stat.h"
 3 #include "user.h"
 4
 5 int main(int argc, char*argv[]){
 6     int pid = getpid();
 7     int ppid = getppid();
 8     printf(1,"My pid is %d\n",pid);
 9     printf(1,"My ppid is %d\n",ppid);
10     exit();
11 };
```

```
185         _my_userapp\
186         _project01\
```

```
253 EXTRA=\
254     mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
255     ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
256     printf.c umalloc.c my_userapp.c project01.c\
```

력될 수 있도록 한다. 그리고 make 될 수 있도록 Makefile에 코드를 집어넣어준다. wrapper 함수를 따로 .c 파일로 만들어준게 아니라 sysproc.c 안에 만들었기 때문에 .o 파일을 Makefile에 넣어줄 필요는 없다.

2.5 이대로 make 해주면 오류없이 잘 동작함을 확인할 수 있고, 실행해도 과제 예시처럼 작동함을 알 수 있다.

```
Booting from Hard Disk..xv6..
cpu0: starting 0
sb: size 1000 nblocks 941 nir
init: starting sh
$ project01
My pid is 3
My ppid is 2
$
```

- 부모 프로세스의 ID를 어디서 구할 수 있는지 알아내는 과정에서 조금의 문제가 있긴 했으나, cscope으로 proc을 분석해서 부모 프로세스의 구조체를 찾아낼 수 있었다. 실습에서 했던 prac_syscall과 다른점은 따로 .c파일로 함수를 만들어준게 아니라 기존에 있는 sysproc.c 파일 안에 함수를 만들어 줬다는 것이고, 이로 인해서 오브젝트 파일(.o)을 구지 만들어줄 필요가 없었다.

Project 01 – 2

- XV6

- System call
  - System call은 user가 어떤 작업(OS 수준에서만 할 수 있는)을 실행할 때 OS의 kernel에 그 작업을 하도록 요청하는 방식을 뜻한다.
  - user는 하드웨어에 직접 접근할 수 없고, kernel만이 하드웨어에 접근할 수 있다. 또한 user는 kernel에도 직접 접근할 수 없고 오직 system call만이 kernel에 접근할 수 있다.
  - 결국 user는 system call을 통해서 kernel에 접근해야 한다.
  - xv6에서 system call은 syscall.h에 들어가면 전체를 다 확인할 수 있다. 처음 설치했을 땐 총 21개의 system call을 확인할 수 있다.

```
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 };
```

- Process
  - 이번 과제를 통해서 proc의 구조에 대해서 어느정도는 파악할 수 있었다.
  - proc 구조체는 이런 내용들을 담고있다. 일단 process는 stack 형태로 kernel에서 관리

```
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 }
67
```

된다는 사실을 알 수 있다. 그리고 proc의 상태, process의 ID, 부모 proc 구조체, 트랩 프레임, 파일 열기, 디렉토리 정보, 이름 등의 내용을 담고 있다는 것을 알 수 있다. 프로세스의 이름은 16자리 문자열을 넘을 수 없게 되어있다.

- myproc(void)의 정보도 확인할 수 있는데 cpu와 proc의 구조체의 주소를 선언하여 결국 mycpu()의 proc 구조체의 주소값을 return하는 형태이다. 따라서 프로세스의 정보에 접근하고 싶다면 myproc() 함수를 호출하면 된다. 그러면 proc이라고 하는 구조체에 담겨 있는 멤버 변수들에 접근할 수 있다.

```
104 //PAGEBREAK: 16
105 // proc.c
106 int             cpuid(void);
107 void            exit(void);
108 int             fork(void);
109 int             growproc(int);
110 int             kill(int);
111 struct cpu*     mycpu(void);
112 struct proc*    myproc();
113 void            pinit(void);
114 void            procdump(void);
115 void            scheduler(void) __attribute__((noreturn));
116 void            sched(void);
117 void            setproc(struct proc*);
118 void            sleep(void*, struct spinlock*);
119 void            userinit(void);
120 int             wait(void);
121 void            wakeup(void*);
122 void            yield(void);
```

- 다시 이번에는 defs.h 헤더에서 proc.c를 위해 선언된 함수들을 확인할 수 있다. cpu와 proc 구조체의 주소값이 선언되어있고, 그 밖에 필요한 함수들이 있다. fork(void) 함수는 새로운 프로세스를 생성하는 역할을 하고 cpuid(void)는 cpu의 id 값을 int로 return 한다. 그 밖에도 exit(void) 처럼 프로세스를 종료하거나 scheduler(void) 나 wait(void) 처럼 프로세스 흐름을 관리를 하는 함수들도 존재한다. 여기서 헤더에 선언을 해 두었기 때문에 proc.c에서 활용할 수 있게 된다. 그리고 함수의 내용들은 myproc(void)처럼 proc.c에서 지정되어있다.