

Project01 Wiki

Design

▼ int sys_getpid(void) 분석

```
int
sys_getpid(void)
{
    return myproc()->pid;
}
```

▼ myproc() - proc.c: 현재 cpu에서 실행 중인 process 구조체 포인터 반환

```
struct proc*
myproc(void) {
    struct cpu *c;
    struct proc *p;
    pushcli();
    c = mycpu();
    p = c->proc;
    popcli();
    return p;
}
```

▼ pushcli(): 현재 cpu의 interrupt를 비활성화(cli() 함수)하는 함수

```
void
pushcli(void)
{
    int eflags;

    eflags = readeflags();
    cli();
    if(mycpu()->ncli == 0)
        mycpu()->intena = eflags & FL_IF;
    mycpu()->ncli += 1;
}
```

1. cli() 함수를 통해 현재 cpu의 interrupt를 비활성화
2. 이 함수가 처음으로 호출되면(if(mycpu()->ncli == 0)), mycpu()->intena에 원래 interrupt flag 상태를 저장

a. ncli: pushcli()가 중첩된 횟수

3. mycpu()→ncli를 1 증가

▼ **popcli()**: 현재 cpu의 interrupt를 활성화(sti() 함수)하는 함수

```
void
popcli(void)
{
    if(readeflags() & FL_IF)
        panic("popcli - interruptible");
    if(--mycpu()->ncli < 0)
        panic("popcli");
    if(mycpu()->ncli == 0 && mycpu()->intena)
        sti();
}
```

1. 현재 interrupt가 비활성화 된 상태인지 확인 (if(readeflags() & FL_IF))
 2. mycpu()→ncli를 1 감소한 뒤에 그 값이 0이고, mycpu()→intena가 참일 경우(pushcli() 전에 interrupt가 활성화되어 있었던 경우)에 sti()를 호출하여 현재 cpu의 interrupt를 활성화
- 현재 cpu에서 실행 중인 process를 참조하기 전에 interrupt를 비활성화하는 이유
 - user가 myproc() 호출하고 process를 참조하기 전에 cpu scheduling이 발생하면 user가 원하는 process가 반환되지 않을 수 있기 때문에 interrupt를 잠시 막아준다.
 - **sys_getpid() - sysproc.c**: 현재 cpu에서 실행 중인 process의 pid를 반환

▼ **int getpid(void)**

- **user.h**

```
...
int getpid(void);
...
```

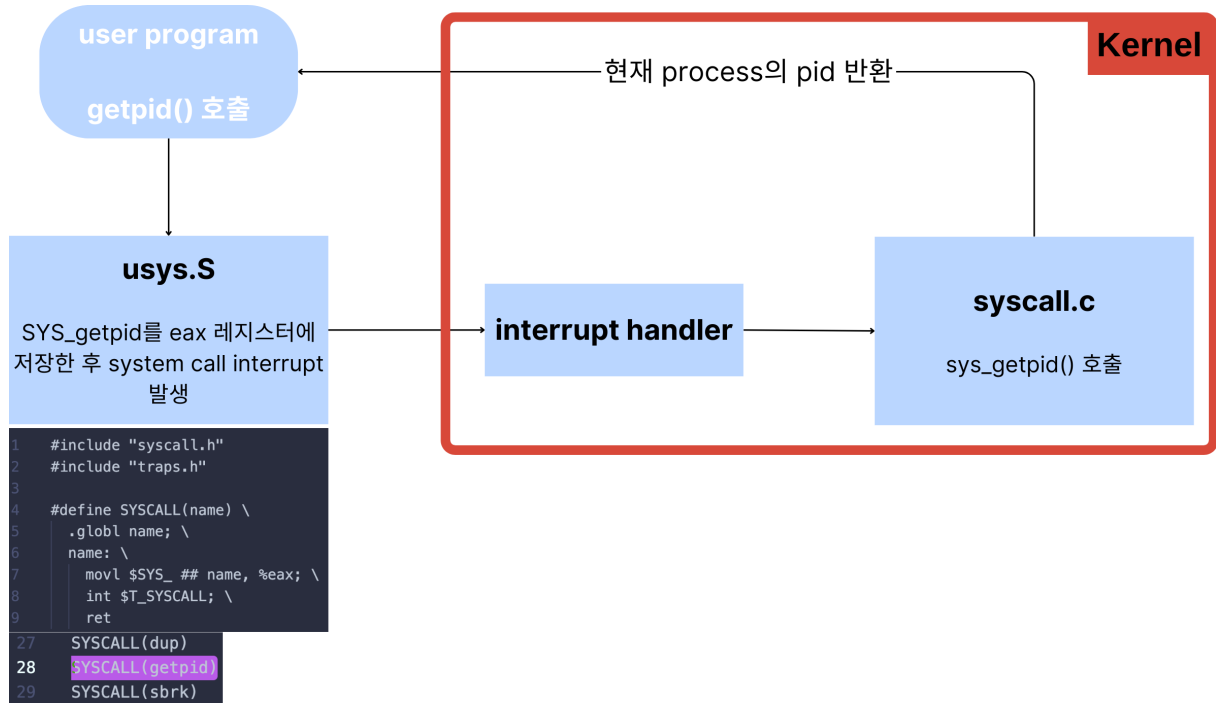
- getpid() 시스템콜은 user.h에 선언만 되어 있고 따로 정의되어 있지 않다.
- user가 program에서 getpid() 시스템 콜을 호출하면 wrapper function sys_getpid()가 현재 실행 중인 process의 pid를 반환하기 때문에 따로 getpid() 시스템 콜을 정의할 필요가 없다.

- **usys.S**

```
...
SYSCALL(getpid)
...
```

- user program에서 getpid() 시스템 콜을 호출할 경우 wrapper function sys_getpid()가 호출될 수 있도록 해주는 preprocess를 정의

- defs.h에서는 선언되지 않은 이유
 - getpid() 시스템 콜은 user program에 제공되는 인터페이스이기 때문에 커널의 내부 구현과 관련된 함수들의 선언을 담당하는 defs.h에는 선언하지 않는 것이 적합하다.



▼ int getpid() design

getpid()와 sys_getpid()

1. `int getpid()` 함수는 `getpid()` 함수와 마찬가지로 `user.h`에 선언만 한다.
2. wrapper function `int sys_getpid()`를 kernel 영역(`sysproc.c`)에 정의한다.
 - a. user program에서 `getpid()` 시스템 콜을 호출하면 `pid`를 반환하는 과정은 `sys_getpid()` 함수에서 모두 처리하도록 설계한다.

sys_getpid() 설계

1. 현재 실행 중인 process의 parent process의 parent process 구조체를 찾는다.
 - a. 위의 `mycpu()` 함수와 `struct proc` 참고
2. 이 구조체의 `pid`를 반환한다.
 - a. 위의 `sys_getpid()` 참고

Implement

▼ int getgpid(void)

- **user.h**

```
...
// project01 - getgpid()
int getgpid(void);
```

- user.h에 getgpid() 시스템콜을 선언함으로써 user program에서 시스템콜을 호출할 수 있도록 한다.
- gpid 반환과 관련된 모든 작업은 kernel 영역에서 정의된 sys_getgpid() 함수에서 처리하기 때문에 getgpid() 시스템콜은 선언만 한다.

- **sysproc.c**

```
...
// return grandparent pid
int
sys_getgpid(void)
{
    struct proc *gp;
    gp = myproc()->parent->parent;
    return gp->pid;
}
```

- 함수의 정의
 1. 현재 process의 grand parent process 포인터 값을 가질 proc 구조체 gp를 선언한다.
 2. 현재 cpu에서 실행 중인 process 구조체의 포인터를 myproc() 함수를 통해 찾고, 이 process의 parent process pointer의 parent process pointer 값을 gp에 assign한다.
 3. gp의 pid(gpid)를 반환한다.

- **usys.S**

```
...
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(myfunction)
SYSCALL(getgpid)
```

- usys.S 맨 아래에 위의 코드를 추가하여, user program에서 시스템콜을 호출할 때 어떻게 preprocessing할지 정의한다.

- **syscall.h**

```
...
#define SYS_mkdir 20
#define SYS_close 21
```

```
#define SYS_myfunction 22
#define SYS_getgid 23
```

- syscall.c

```
...
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_myfunction(void);
extern int sys_getgid(void);

static int (*syscalls[])(void) = {
...
    [SYS_close]    sys_close,
    [SYS_myfunction] sys_myfunction,
    [SYS_getgid]   sys_getgid,
};
```

- syscall.h와 syscall.c에 위의 코드를 추가하여, 사용자가 getgid() 시스템콜을 호출하면 wrapper function sys_getgid()가 호출될 수 있도록 한다.

▼ User program - project01

▼ void printf(int, const char*, ...) 분석

- user.h

```
// ulib.c
...
void printf(int, const char*, ...);
...
```

▼ code of print() - printf.c

```
printf(int fd, const char *fmt, ...)
{
    char *s;
    int c, i, state;
    uint *ap;

    state = 0;
    ap = (uint*)(void*)&fmt + 1;
    for(i = 0; fmt[i]; i++){
        c = fmt[i] & 0xff;
        if(state == 0){
            if(c == '%'){
                state = '%';
```

```

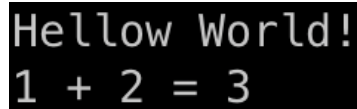
    } else {
        putc(fd, c);
    }
} else if(state == '%'){
    if(c == 'd'){
        printint(fd, *ap, 10, 1);
        ap++;
    } else if(c == 'x' || c == 'p'){
        printint(fd, *ap, 16, 0);
        ap++;
    } else if(c == 's'){
        s = (char*)*ap;
        ap++;
        if(s == 0)
            s = "(null)";
        while(*s != 0){
            putc(fd, *s);
            s++;
        }
    } else if(c == 'c'){
        putc(fd, *ap);
        ap++;
    } else if(c == '%'){
        putc(fd, c);
    } else {
        // Unknown % sequence. Print it to draw attention.
        putc(fd, '%');
        putc(fd, c);
    }
    state = 0;
}
}
}

```

- Argument
 - int fd: 출력할 대상의 File Descriptor
 - 우리는 표준 출력을 해야하므로 1로 설정한다.
 - const char *fmt: 화면에 출력할 format string
 - 변수를 넣고 싶은 부분에 %d와 같이 format
 - %d는 10진수 integer, %s는 string, %c는 character 등등
 - ... : format string에 따라 처리되는 가변 인자 리스트 (string에 넣을 변수)
- Example

```
printf(1, "Hello World!\n");

int a, b;
a = 1;
b = 2;
printf(1, "%d + %d = %d\n", a, b, a+b);
```



```
Hello World!
1 + 2 = 3
```

▼ project01.c

```
#include "types.h"
#include "user.h"

int main(int argc, char *argv[])
{
    int my_pid, my_gpid;
    printf(1, "My student id is 2021042842");

    my_pid = getpid();
    my_gpid = getgpid();
    printf(1, "My pid is %d", my_pid);
    printf(1, "My gpid is %d", my_gpid);
    exit();
}
```

▼ 헤더 파일

- types.h
 - uint, int, char 등의 기본적인 data type을 정의한 헤더파일
- user.h
 - user program에서 사용할 수 있는 구조체, 함수, 시스템 콜이 선언되어 있는 헤더파일

▼ main문

1. my_pid, my_gpid 변수 선언
 - a. my_pid: 현재 cpu에서 실행 중인 process의 pid를 assign할 변수
 - b. my_gpid: 현재 cpu에서 실행 중인 process의 grand parent process pid를 assign할 변수
2. 학번 출력 - 위의 printf()함수 분석을 참고하여 호출한다.
 - a. first argument: 표준 출력이기 때문에 1
 - b. second argument: printf에서 지원하는 출력할 format string

3. getpid() 시스템콜을 호출하여 현재 cpu에서 수행중인 process의 pid를 my_pid 변수에 assign한다.
4. getppid() 시스템콜을 호출하여 현재 cpu에서 수행중인 process의 grand parent process의 pid를 my_gpid 변수에 assign한다.
5. 현재 process의 pid와 gpid 출력 (my_pid, my_gpid 변수 출력)
 - a. first argument: 표준 출력이기 때문에 1
 - b. second argument: printf에서 지원하는 출력할 format string
 - c. reamin argument: format string 변수 자리에 들어갈 변수
6. user process를 종료하기위해 exit() 시스템콜을 호출하여 main문을 마무리한다.

▼ Makefile

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

```
UPROGS=\
...
_wc\
_zombie\
_my_userapp\
project01\
...
```

- user program 코드 파일을 compile하고, shell에서 실행하기 위해 위의 코드를 추가한다.

Result

▼ Cross Compile (MacOS M1)

docker를 이용해 프로젝트를 수행하던 중 불편한 점이 많아 mac os m1 환경에서 xv6를 개발하기 위한 cross compile 방법을 탐색했다.

1. install homebrew

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

2. install qemu

```
$ brew install qemu
```


3. install x86_64-elf tools

```
$ brew install x86_64-elf-gcc
$ brew install x86_64-elf-gdb
$ brew install x86_64-elf-binutils
```

4. Makefile을 다음과 같이 수정한다.

- As is

```
OBJS = \
...
# Cross-compiling (e.g., on Mac OS X)
# TOOLPREFIX = i386-jos-elf
...
# If the makefile can't find QEMU, specify its path here
# QEMU = qemu-system-i386

# Try to infer the correct QEMU
ifndef QEMU
QEMU = $(shell if which qemu > /dev/null; \
    then echo qemu; exit; \
    ...)
endif

CC = $(TOOLPREFIX)gcc
AS = $(TOOLPREFIX)gas
LD = $(TOOLPREFIX)ld
...
CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall
-MD -ggdb -m32 -Werror -fno-omit-frame-pointer
...
LDFLAGS += -m $(shell $(LD) -V | grep elf_i386 2>/dev/null | head -n
1)
```

- To be

```
OBJS = \
...
# Cross-compiling (e.g., on Mac OS X)
ifeq ($(shell uname), Darwin)
    TOOLPREFIX = x86_64-elf-
endif
...
# If the makefile can't find QEMU, specify its path here
```

```
# QEMU = qemu-system-i386

# Try to infer the correct QEMU
ifndef QEMU
    QEMU = /opt/homebrew/bin/qemu-system-i386
endif

CC = x86_64-elf-gcc
AS = $(TOOLPREFIX)gas
LD = x86_64-elf-ld
...
CFLAGS = -fno-pic -std=gnu99 -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -march=i386 -fno-omit-frame-pointer
...
LDFLAGS += -m $(shell $(LD) -V | grep elf_i386 2>/dev/null | head -n 1)
LIBGCC_A = $(shell $(CC) $(CFLAGS) -print-libgcc-file-name)
FS=fs.img
```

5. Build project & Booting xv6 with QEMU

방법1)

```
$ make CPUS=1 qemu-nox
```

방법 2)

```
$ make
$ make fs.img
$ qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp
```

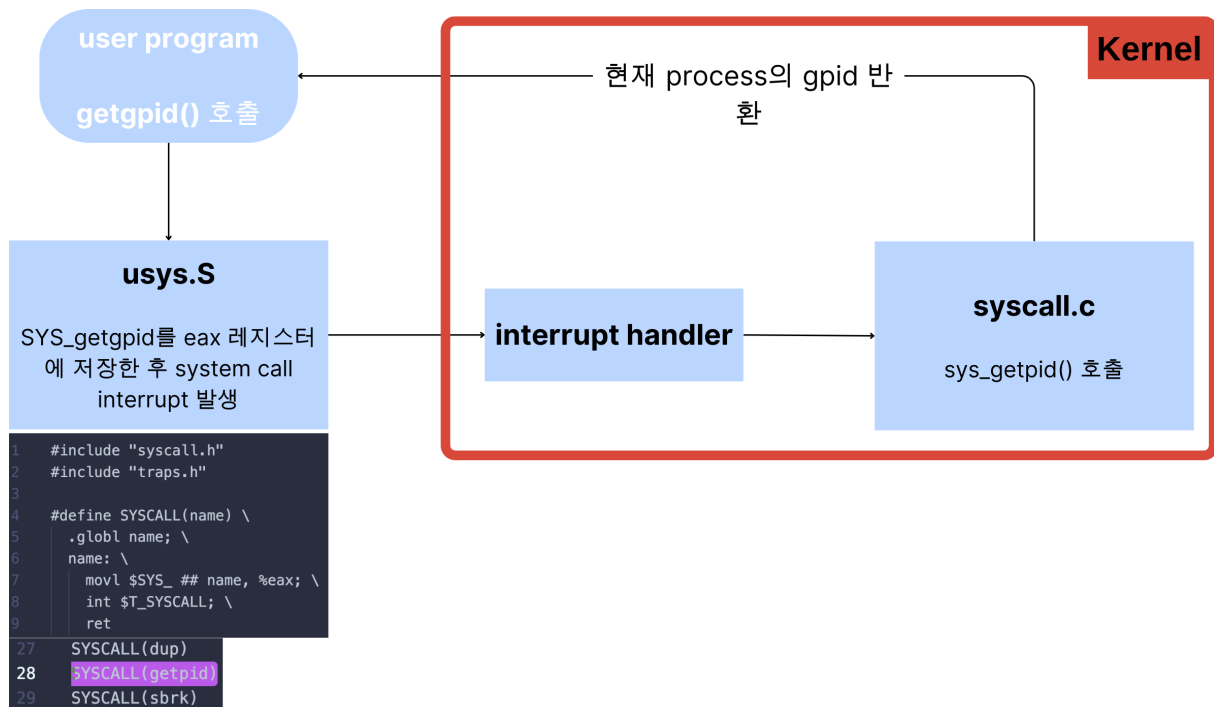
▼ User Program 실행 예시

```
SeaBIOS (version rel-1.16.3-0-ga6ed6b701f0a-prebuilt.qemu.org)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFD0FC0+1EF30FC0 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
$ project01
My student id is 2021042842
My pid is 3
My gpid is 1
$ echo hello
hello
$ project01
My student id is 2021042842
My pid is 5
My gpid is 1
$
```

▼ getpid() 시스템콜 호출 과정



1. user program: `getpid()` 시스템콜을 호출한다.
2. usys.S: `syscall.h`에 정의되어 있는 `SYS_getpid` 값을 `eax` 레지스터에 저장한 후에 system call interrupt를 발생시킨다.
3. interrupt handler에 의해 `syscall.c`에 정의된 `syscall()` 함수가 호출된다.
 - a. `eax` register에 있는 값을 `num` 변수에 assign하고, `syscalls[num]()`을 호출한다.

b. syscall.c

```
static int (*syscalls[])(void) = {  
    ...  
    [SYS_getgid]    sys_getgid,  
};
```

위의 정의에 의해 `syscalls[num]()` 호출은 `sys_getgid()`를 호출하는 것과 같다.

4. 3번에 의해 `getgid()` 시스템콜의 wrapper function `sys_getgid()`이 호출되고 이 함수는 현재 `cpu`가 수행 중인 `process`의 grand parent의 `pid(gid)`를 return한다.

Trouble shooting

▼ `getgid()` 정의 문제

: 이 시스템콜의 작업 과정을 `getgid()`에 정의해야 되는지 wrapper function `sys_getgid()`에 정의해야 되는지를 결정하는 데 어려움이 있었다.

- wrapper `sys_getgid()`에 정의한 이유
 - `getgid()`는 user program의 요청에 의해 user mode에서 호출되는 시스템콜이지만 `sys_getgid()`는 이 요청을 실제로 처리하는 kernel mode의 함수이기 때문이다.