Operating Systems
- Project01 Wiki–

2017029916
양동해

Ⅰ. Project01-1

1. 디자인

부모프로세스의 pid를 가져오기 위해, 우선 자기 자신의 pid를 가져오는 system call을 확인해 보았다. sysproc.c 파일에 sys_getpid()가 정의되어 있었는데, 여기서 pid 값을 myproc()가 반환하는 포인터를 통해 가져온다는 것을 알았다. myproc()은 proc.c 파일에 정의되어 있었으며, 이 함수가 반환하는 반환형인 struct proc는 proc.h 파일에 정의되어 있었다.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

[proc.h 파일 안에, struct proc의 구조]

struct proc안에는 부모 프로세스를 가리키는 포인터인 parent가 존재했고, 이를 통해 부모 프로세스의 pid를 가져올 수 있을 것이라 생각했다.

자기 자신의 프로세스는 myproc()을 통해 접근 가능하므로, myproc()->parent->pid를 하게 되면 부모프로세스의 pid를 가져올 수 있다.

2. 구현

우선 커널 프로그램인 getppid부터 만들어야 하므로, getppid.c 파일을 생성해 다음과 같이 작성했다. Wrapper 함수도 작성했지만 어차피 바로 getppid()를 호출하기 때문에 큰 의미는 없다.

```
#include "types.h"
#include "x86.h"
#include "defs.h"
#include "date.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"

int getppid(void) {
    return myproc()->parent->pid;
}

int sys_getppid(void) {
    return getppid();
}
```

[getppid.c 파일]

그 다음, 오브젝트 파일을 관리하기 위해 Makefile 파일의 OBJS에 getppid.o를 추가해주었고, 커널 전체에서 getppid()의 호출이 가능하도록 defs.h에 int getppid(void)를 선언해주었다. 자세한 코드 부분은 Makefile 파일의 31번 줄, defs.h 파일의 193번 줄에서 확인이 가능하다.

그리고 방금 만든 getppid를 system call로 추가 해주기 위해, syscall.h 파일과 syscall.c 파일에 각각 알맞는 값을 선언해주었다. syscall.h 파일에서 SYS_getppid를 23으로 정의했기 때문에, syscall.c 파일의 알맞은 위치에 배열 값을 추가해주었다.

```
#define SYS_close  21
#define SYS_myfunction  22
#define SYS_getppid 23
```

[syscall.h 파일 안에, 선언한 SYS_getppid 값]

```
extern int sys_uptime(void);
extern int sys_myfunction(void);
extern int sys_getppid(void);
```

[syscall.c 파일 안에, 선언한 sys_getppid(void)]

```
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_myfunction]  sys_myfunction,
[SYS_getppid] sys_getppid,
};
```

[syscall.c 파일 안에, static int (*syscalls[])(void)의 일부분]

마지막으로 유저 프로그램에서 이 system call을 사용할 수 있게 user.h 파일 안에 int getppid(void)를 선언해주었고, usys.S 파일 안에도 SYSCALL(getppid)를 추가해주었다. 자세한 코드 부분은 user.h 파일의 27번 줄과 usys.S 파일의 33번 줄에서 확인이 가능하다.

이제 system call과 관련된 부분은 모두 작성 했으므로, 이를 실행할 유저 프로그램만 만들면 된다. 유저 프로그램의 이름은 project01_1이며, 명세에 맞게 다음과 같이 작성했다.

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

int main(int argc, char *argv[]) {
    int myPid = getpid();
    int myPPid = getppid();
    printf(1, "My pid is %d\n", myPid);
    printf(1, "My ppid is %d\n", myPPid);
    exit();
}
```

[project01_1.c 파일]

만든 유저 프로그램을 실행하기 위해, Makefile 파일의 UPROGS 부분과 EXTRA 부분에 알맞는 값을 추가해주었다. 자세한 코드 부분은 Makefile 파일의 187번 줄과 259번 줄에서 확인이 가능하다.

3. 실행 결과

이를 실행하기 위해, make clean, make, make fs.img를 차례로 수행하고 xv6를 실행시키면 다음과 같은 결과가 나타난다. (권한 설정만 해준다면, xv6의 실행은 실습시간에 만든 bootxv6.sh의 사용으로도 가능하다.)

```
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ project01_1
My pid is 3
My ppid is 2
$
```

[xv6에서 project01_1 실행 결과]

4. 트러블슈팅

우선 매우 기본적이고 당연한 이슈이지만, 처음에 어떻게 접근해야 할지 막막해서 실습시간에 한 내용을 복습하고 대다수의 헤더 파일과 C 파일을 차근차근 읽어 보았다. 그러면서 system call이 어디서 선언되는지, 어떻게 호출되는지 등을 알 수 있었고, 프로세스 상태를 나타내는 구조체인 proc에 대해서도 이해할 수 있었다.

개발하는 과정에서도 이슈가 존재했는데, getppid.c 파일에서 맨 처음에 types.h 파일과 defs.h 파일만 include 하여 문제가 있었다.

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

int getppid(void) {
    return myproc()->parent->pid;
}

int sys_getppid(void) {
    return getppid();
}
```

[처음에 작성한 getppid.c 파일]

이렇게 파일을 만들고 make를 하게 되면, 아래와 같은 에러를 만날 수 있었다.

```
getppid.c: In function 'getppid':
getppid.c:5:20: error: dereferencing pointer to incomplete type 'struct proc'
     return myproc()->parent->pid;
                    ^~
getppid.c:6:1: error: control reaches end of non-void function [-Werror=return-type]
 }
 ^
cc1: all warnings being treated as errors
<builtin>: recipe for target 'getppid.o' failed
make: *** [getppid.o] Error 1
```

[처음에 작성한 getppid.c 파일로 make를 했을 때 발생한 에러]

이 문제에 대해 여러가지 시도를 한끝에 알아낸 것은, defs.h 파일에 선언된 myproc()만을 가지곤 proc의 형태를 모르기 때문에 에러가 발생한 것이라고 판단했다. 이를 해결하기 위해서 getppid.c 파일에 proc.h 파일을 하나 더 include 했지만 그럼에도 오류가 발생하여, sysproc.c 파일에 존재하는 모든 헤더 파일을 include 하는 것으로 이 문제를 해결했다. (완성된 getppid.c 파일은 "2. 구현"에서 본 것과 같다.)

Ⅱ. Project01-2

1. 디자인

처음 main 함수가 실행될 때, tvinit() 함수를 통해 IDT(interrupt descriptors table)를 초기화 해준다. 그리고 프로그램 실행 도중, interrupt가 호출되면 CPU의 레지스터를 전부 저장하고 trap을 호출한다. 그 trap에서 각각 trap number에 맞춰 나머지 하위 코드가 실행된다.

여기서 우리가 만들 128번 interrupt 또한 우리가 원하는 흐름에 맞게 실행시켜 주기 위해선, 먼저 trap.c 파일에 존재하는 tvinit() 함수에서 128에 해당하는 interrupt를 유저모드에서 호출할 수 있도록 바꾸어 주어야 한다. 그러고 난 뒤 trap() 함수에서 128번 interrupt에 대한 처리를 해주면 된다.

2. 구현

trap.c 파일 안의 tvinit() 함수에서, 128번 interrupt 호출을 유저 프로그램에서 할 수 있도록 SETGATE의 dpl 값을 DPL_USER로 바꾸어 준다. (SETGATE는 mmu.h 파일에서 확인할 수 있었다.)

```
void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
  SETGATE(idt[128], 0, SEG_KCODE<<3, vectors[128], DPL_USER);

  initlock(&tickslock, "time");
}
```

[trap.c 파일 안에, tvinit() 함수]

그리고 이를 처리할 루틴을 trap() 함수 내에 만들어 준다.

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  if(tf->trapno == 128){
    cprintf("user interrupt 128 called!\n");
    exit();
    return;
  }
```

[trap.c 파일 안에, trap() 함수의 일부분]

이제 이를 호출할 유저 프로그램을 만들어준 뒤, 이에 대한 Makefile 파일의 수정도 해준다. project01_2의 코드는 아래 사진으로 첨부했으며, Makefile 파일의 자세한 코드 내용은 188번 줄과 259번 줄에서 확인이 가능하다.

```
int main(int argc, char *argv[]) {
    __asm__("int $128");
    return 0;
}
```

[project01_2.c 파일]

3. 실행 결과

이를 실행하기 위해, make clean, make, make fs.img를 차례로 수행하고 xv6를 실행시키면 다음과 같은 결과가 나타난다. (권한 설정만 해준다면, xv6의 실행은 실습시간에 만든 bootxv6.sh의 사용으로도 가능하다.)

```
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ project01_2
user interrupt 128 called!
$
```

[xv6에서 project01_2 실행 결과]

4. 트러블슈팅

Project01-2도 Project01-1과 마찬가지로, interrupt에 대한 개념이 부족하여 실습시간에 한 내용을 다시 한 번 복습했다.

개발과정에서 발생한 이슈로는, 처음에 project01_2.c 파일에 존재하는 __asm__의 뜻을 몰라 찾아보았는데, 어셈블리어를 실행하는 명령어였다는 것을 알았다. 때문에 C 파일에서 바로 저렇게 사용해도 문제 없이 어셈블리어를 사용할 수 있다는 것을 배웠다.

trap.c 파일을 수정하는 과정에서도, 처음에는 단지 trap() 함수의 switch문에서 128 번의 case만 추가해주면 되는 줄 알았다.

```
case T_IRQ0 + 7:
case T_IRQ0 + IRQ_SPURIOUS:
  cprintf("cpu%d: spurious interrupt at %x:%x\n",
          cpuid(), tf->cs, tf->eip);
  lapiceoi();
  break;
case 128:
  cprintf("user interrupt 128 called!\n");
  break;

//PAGEBREAK: 13
default:
```

[처음에 작성한 trap.c 파일 안에, trap() 함수의 일부]

하지만 이렇게 하면 계속 13번 interrupt로 실행되어서, 여러가지 시도 끝에 처음 tvinit() 부터 바꾸어 주어야 한다는 것을 알았다. 그러고 구조 자체를 SYSCALL 처리와 비슷하게 수정했더니(이 부분은 "2. 구현"에서 본 것과 같다.) 오류없이 128번 interrupt에 대한 처리를 할 수 있었다.