Project1

Project1-1 : getppid() 시스템 콜 구현

System call

what is system call?

시스템 콜은 운영 체제의 커널이 제공하는 서비스에 대해, 프로그램이 커널에 접근하기 위한 인터페이스이다. 운영체제는 Kernel mode 와 User mode로 구성되어 있다. 커널 모드는 시스템에 대해 가장 많은 권한을 가지고 있다. 그렇기 때문에 커널 모드에서만 실행할 수 있는 운영 체제의 기능을 사용하기 위해서는 반드시 시스템 콜을 거쳐야만 한다.

System call process in xv6

xv6에서 myfunction (실습 때 작성했던) 이라는 이름의 시스템 콜이 호출되는 과정을 살펴보자.

1. **응용 프로그램 → usys.S**

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

int
main(int argc, char *argv[])
{
    char *buf = "Hello xv6!";
    int ret_val;
    ret_val = myfunction(buf);

    printf(1, "Return value : 0x%x\n", ret_val);
    exit();
};
```

fig 1-1. my_userapp.c

```c
26    int myfunction(char*);
```

fig 1-2. user.h

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
SYSCALL(myfunction)
```

fig 1-3. usys.S

```c
#define SYS_myfunction 22
```

fig 1-4. syscall.h

```c
#define T_SYSCALL       64
```

fig 1-5. traps.h

my_userapp.c 라는 프로그램에서 user.h (fig 1-2) 에 선언되어있는 시스템 콜인 myfunction 이 호출되는 것을 fig 1-1에서 볼 수 있다. myfunction이 호출되면 usys.S (fig

1-3) 에 있는 `SYSCALL(myfunction)` 을 통해 `$SYS_ ## name` 은 `$SYS_myfunction` 으로, fig 1-4 에서의 정의로 인해 `$SYS_myfunction` 은 22로 바뀌게 되어 eax 레지스터에는 22가 담기게 된다.

그 다음 `int $T_SYSCALL` (fig 1-3) 으로 인터럽트를 발생 시키는데, `T_SYSCALL` 은 traps.h (fig 1-5) 에 정의되어있다.

2. Interrupt 발생 → trap.c (IDT)

```
struct gatedesc idt[256];
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);
```

fig 1-6. trap.c

```
.globl vector64
vector64:
  pushl $0
  pushl $64
  jmp alltraps
```

fig 1-7. vectors.S

인터럽트가 발생되면 trap.c (fig 1-6) 에 있는 IDT가 사용된다. IDT는 미리 정의되어 있는 인터럽트들의 번호와 실행 코드를 가리키는 주소들이 저장되어 있는 table 이다. 컴퓨터 부팅 시에 운영체제가 IDT에 인터럽트들을 기록하고, 인터럽트가 발생하면 IDT를 확인하여 해당하는 함수를 호출하여 인터럽트를 처리하게 된다. IDT 에는 `alltraps` 를 분기하는 코드가 있는데, vectors.S (fig 1-7) 에 있는 jmp를 통해 `alltraps` 함수로 이동하게 된다.

3. trapsam.S → trap.c

```
# Call trap(tf), where tf=%esp
pushl %esp
call trap
addl $4, %esp
```

fig 1-8. trapasm.S

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

fig 1-9. trap.c

trapsam.S (fig 1-8) 에는 `alltraps` 가 있는데, `alltraps`는 trap frame을 만든다. trap frame이란 어떤 Interrupt routine이 수행되기 직전에 사용되던 레지스터의 값들을 stack 공간에 저장하는 것을 말한다. user level에서 kernel level으로 넘어가며 system call을 실행하는 과정에서 필수적인 코드이다. `alltraps` 내부에서 `call trap` instruction을 통해 trap.c (fig 1-9) 의 trap 함수를 호출한다. 현재 trap frame의 `trapno`는 `T_SYSCALL` 이기 때문에 해당 분기로 넘어가서 syscall 함수가 호출된다.

4. **syscall.c → prac_syscall.c**

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
```

fig 1-10. syscall.c

```
// Simple system call
int
myfunction(char *str)
{
    cprintf("%s\n", str);
    return 0xABCDABCD;
}

int
sys_myfunction(void)
{
    char *str;

    // Decode argument using argstr
    if (argstr(0, &str) < 0)
        return -1;
    return myfunction(str);
}
```

fig 1-11. prac_syscall.c

syscall 함수 (fig 1-10) 가 호출되면, `curproc→tf→eax`에는 아까 usys.S 에서 담긴 22가 들어가게 되고, `syscalls[22]` 에는 sys_myfunction가 담겨있기에 이를 호출한다. sys_myfunction은 prac_syscall.c (fig 1-11)에 정의되어 있는데, argstr을 통해 argument를 decode한 후 myfunction을 호출한다.

The implementation of getppid()

Definition

getpid() 와 getppid() 시스템 콜의 목적은 process의 id를 얻는다는 점에서는 같다. 하지만, getpid()는 현재 process의 id를, getppid()는 부모 process의 id를 반환해야 한다는 점에서는 다르다. 여기서 process란 실행 중인 프로그램을 말한다.

An analysis of getpid()

getpid() 와 getppid() 는 id를 찾는 대상은 다르지만, 기능은 유사하기 때문에 이미 구현되어 있는 getpid() 시스템 콜을 먼저 분석해 보기로 했다. getpid() 의 호출 과정은 위의 myfunction 시스템 콜의 호출 과정과 유사하지만 세 가지 차이점이 있다.

1. usys.S 에서 SYSCALL 매크로 함수 내부의 `$SYS_getpid` 는 syscall.h에서 11로 정의되어 있다.
2. sys_getpid() 는 sysproc.c에 정의되어 있고, 내부에서는 따로 argument를 decode하는 과정이 존재하지 않다.
3. sys_getpid() 내부 동작이 다르다.

이 세 가지 차이점을 제외한 호출 과정은 위에서 설명한 myfunction과 동일하다.
sys_getpid() 는 `myproc()→pid`를 반환하는데, myproc() 이 뭔지 분석해야한다.

myproc() 는 proc.c 에 정의되어 있다. cpu 구조체 포인터 변수 `c`와 proc 구조체 포인터 변수 `p`를 선언한 후, `c`에 mycpu() 의 반환값을 담는다. mycpu() 는 cpu 구조체 포인터를 반환하는데, cpu 구조체에는 proc 구조체의 포인터인 proc 변수가 존재한다. 이후 `c→proc` 을 `p`에 담아준 후 `p`를 반환해준다. 즉 myproc() 을 호출하면 proc 구조체의 포인터가 반환되는데, proc 구조체는 다음과 같이 정의되어 있다.

```
// proc.h
struct proc {
  ...
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  ...
};
```

이 중 getpid() 에서 중요한 부분은 pid와 parent인데, getpid()의 경우 현재 process의 id를 반환하는 것이므로 단순히 `myproc()→pid` 를 반환하면 되지만, getppid() 의 경우 `myproc()→parent→pid` 를 반환해야 한다는 것을 깨달았다.

Create getppid() system call

이제 getppid()는 다음과 같이 간단하게 구현되는 것을 알았다.

```
// sysproc.c
int getppid() {
  return myproc()->parent->pid;
}
```

그렇다면 어떤 파일을 생성하고 변경 해야 할까?

1. **project01.c:** 시스템 콜 동작 테스트를 위해 getpid() 와 getppid()가 호출되는 프로그램

2. **syscall.h:** 인터럽트가 발생했을 때, usys.S 의 SYSCALL 함수에서 `$SYS_getppid` 값을 eax 레지스터에 담아야 하기 때문에 SYS_getppid 가 정의되어야 한다. → `#define SYS_getppid 23`
3. **syscall.c:** trap.c 에서 syscall이 호출될 때 syscall 함수 내부에서 `syscalls[SYS_getppid]` 를 통해 sys_getppid 를 호출해야 한다. 따라서 syscalls 함수 포인터 배열에 sys_getppid가 추가되어야 한다. → `[SYS_getppid] sys_getppid,`
4. **sysproc.c:** sys_getppid() 시스템 콜 정의가 있어야 한다.
5. **user.h:** user에게 사용되는 getppid() 의 선언이 있어야 한다.
6. **usys.S:** SYSCALL(getppid) 가 추가되어야 한다.
7. **Makefile**

Result

```
// project01.c

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

int
main(int argc, char *argv[])
{
    int pid, ppid;
    pid = getpid();
    ppid = getppid();

    printf(1, "My pid is %d\n", pid);
    printf(1, "My ppid is %d\n", ppid);
    exit();
};
```

위와 같이 project01.c 를 작성했다. 이후 터미널에서 make → make fs.img → ./bootxv6.sh 순서로 입력한 후 xv6를 실행시켰다.

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 328
init: starting sh
$ project01
My pid is 3
My ppid is 2
$
```

project01 이라고 shell 에 작성하면 다음과 같이 결과가 잘 나오는 것을 확인할 수 있다.

getpid() 의 호출 sequence와 유사하기 때문에 구현하는데 오류를 경험할 수 없었다. 하지만, Makefile을 작성하는 것이 익숙하지 않기 때문에 새로운 파일을 만들고 Makefile을 수정하는 연습이 필요했다.

Project1-2 : xv6 분석하기

xv6

what is xv6?

xv6는 MIT PDOS lab에서 운영체제에 대한 집중적인 학습을 위해 자체 개발한 연습용 운영체제이다. 이전에는 Unix version 6를 이용해서 실습을 했는데, 이후 xv6로 대체 되었다고 한다. v6는 single processor만을 지원하지만, xv6는 multi-processor에서 동작 가능하도록 설계되었다. xv6에서 kernel은 응용 프로그램이 하드웨어에 접근할 수 있도록 여러 서비스를 제공한다. project1-1 에서 설명한 system call이 그러한 역할을 해준다.

Kernel mode vs User mode

kernel mode는 process의 권한이 가장 높은 것을 말하는데, 모든 자원에 접근하고 명령을 내릴 수 있다. 반대로 user mode는 권한이 낮기 때문에 함부로 CPU, memory에 접근할 수 없다. 왜 이렇게 나누어 졌을까? 바로 악의적인 프로그램으로부터 우리의 컴퓨터를 보호하기 위해서 이다. 하지만 우리의 프로그램은 대부분 컴퓨터의 자원이 필요하므로 이에 접근할 수 있는 권한이 필요하다.

1. User mode → Kernel mode

   user mode에서 kernel mode로 넘어가기 위해서는 system call이 필요하다. system call은 위의 myfunction이 호출되는 과정을 통해 설명하였기에 생략한다.

2. Kernel mode → User mode

   커널이 system call로 받은 요청을 처리하고 그 결과값을 반환하면서 user mode로 전환된다.

Booting xv6

처음 PC에 전원을 주면 가장 먼저 실행되는 것은 BIOS (basic input output system)이다. BIOS는 다음 프로그램의 실행을 위해 하드웨어를 초기화하고 디스크의 첫번째 섹터에 존재하는 boot loader 를 실행한다.

xv6의 boot loader는 bootasm.S 와 bootmain.c로 구성되어있다. boot loader의 c 부분인 bootmain.c 는 디스크의 두 번째 섹터부터 kernel 실행 파일의 복사본을 찾는다. kernel 실행 파일은 ELF 형식이기 때문에 이 ELF 헤더의 엑세스 하기 위해 bootmain.c 의 bootmain 함수의 `readseg((uchar*)elf, 4096, 0);` 와 같이 disk에서 첫 page 를 로드한다. 이후 `if(elf->magic != ELF_MAGIC)` 로 가져온 ELF가 실행 가능한지 확인한다. 그 다음 각 프로그램의 세그먼트를 로드한다. 마지막으로 `entry();` 로 xv6 kernel의 entry를 실행한다. entry.S 의 끝부분에 `mov $main, %eax \ jmp *%eax` 에서 보이듯이 main.c 의 main 함수를 호출한다.

Process

kernel은 실행중인 프로그램에 서비스를 제공하는 프로그램이다. 각각 실행되는 프로그램을 process라고 하는데, 이 process들은 명령어, 데이터, 스택을 포함하는 메모리를 가진다. xv6의 process는 kernel이 관리하는 process의 state로 이루어진다. kernel은 process를 proc 구조체를 통해 관리한다.

proc 구조체는 ptable 이라는 proc 구조체의 배열로 관리되는데, xv6에서 최대로 생성될 수 있는 process는 코드에도 나타나듯이 64개이다. (`NPROC` 이 64이기 때문이다.)

```
// proc.c
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state;
  int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];
};
```

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

main.c 에 있는 main() 이 실행되면 가장 먼저 각종 init() 함수들을 호출하며 하드웨어의 자원들과 kernel을 초기화한다. 이후 마지막에 userinit() 을 통해 첫번째 user process를 생성한다. userinit() 의 첫번째 작업은 allocproc() 을 호출하는 것이다. allocproc() 은 process table을 보면서 UNUSED proc 이 있는지 확인하고 `p->state = EMBRYO;` 로 사용중 플래그를 설정함과 동시에 `p->pid = nextpid++;` 로 pid를 부여한다. 그 다음 kernel thread에 kernel stack을 할당한다.

kernel stack을 할당하는 이유는 새로 생성된 process가 scheduling 될 때 실행될 함수와 레지스터를 미리 세팅하기 위해서이다. context 내부의 다음에 실행할 명령어의 주소를 가지고있는 eip register를 forkret() 함수의 주소로 설정함으로써 생성된 process가 scheduling 되면 forkret() 함수가 실행 되도록 한다. 이후 forkret() 함수가 실행된 이후 이어서 trapret() 함수가 실행될 수 있도록 한다.

forkret() 은 이름에서도 알 수 있듯이 fork return의 약자로 새로운 process를 생성한 이후 `release(&ptable.lock)` 에서 처럼 잡고있던 ptable의 lock을 놓아준다. trapret() 은 trapasm.S 에 정의되어 있는데, stack의 trapframe 에 저장되어있던 register를 복구함으로써 user mode로 전환하는 함수이다.

다시 userinit() 으로 돌아와서 allocproc() 의 반환값을 `proc* p` 지역 변수에 저장한다. `p→state = RUNNABLE` 을 통해 process가 scheduling될 수 있게 만들어준다.

Scheduling

scheduling 은 multi-programming 을 가능하도록 하는 OS의 동작 기법이다. OS는 process 에게 CPU 등의 자원을 적절히 할당함으로써 system의 성능을 높일 수 있다.

다시 main.c 로 돌아와 userinit() 이 끝난 이후 mpmain()이 호출된다. mpmain() 은 main.c 에 존재하는데 proc.c 의 scheduler() 를 실행함으로써 process를 실행하도록 만들어준다.

```
// scheduler(void) at proc.c
for(;;){
    sti();
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();
      c->proc = 0;
    }
    release(&ptable.lock);
  }
```

`if(p->state != RUNNABLE) continue;` 에서 처럼 `RUNNABLE` 인 경우에만 process가 scheduling 된다. 위의 userinit() 마지막에서 `p→state = RUNNABLE` 을 통해 해당 user process 가 scheduling 될 수 있게 된다. 이 scheduler() 함수는 계속해서 루프를 돌며 (`for(;;)`)scheduling을 하게 된다.

fork(void)

fork() 는 fork 시스템 콜을 호출한 process를 부모로 하고 새롭게 만든 process를 자식으로 하여, 자식 process에 부모 process의 데이터를 모두 복사하는 것이다. 기존에 실행되고 있던 부모 process에 있는 것들을 자식 process에 복사한다. fork() 의 return value 로 process의 id가 나오는데 0일 경우에는 자식 process의 id 이고, 0이 아닌 양수일 경우 부모 process의 id이다. 그런데 어떻게 fork() 는 return을 2번이나 할 수 있는지 궁금하다.

맨처음 현재 fork() 를 호출한 process의 proc 구조체를 가져오기 위해 `struct proc* curproc = myproc()` 을 한다. 새롭게 생기는 process의 공간을 할당하기 위해 allocproc() 을 호출하여 반환값을 `np` 에 담아주고, `np→pgdir = copyuvm(proc→pgdir, proc→sz)` 와 같이 신규 자식의 proc 구조체에 부모 proc의 내용을 복사한다. 이후 `np->parent = curproc;` 부모 프로세스를 설정해 준다. 그 다음 `np→ofile[i] = filedup(curproc→ofile[i]);` 와 `np→cwd = idup(curproc→cwd);` 에서 부모의 열려진 파일 목록과 현재 directory 관련 정보를 자식에게 복사한다. 이후 부모는 자식의 pid를 가지고 return을 하게 된다. 여기서 아까 allocproc() 을 분석하면서 나왔던 forkret() 과 trapret() 이 사용된다.

```
// proc.c

static struct proc* allocproc(void) {
  ...
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;

  return p;
}
```

allocproc() 의 `*(uint*)sp = (uint)trapret;` 에서 처럼 forkret이 끝난 후 trapret으로 돌아가게 하고, `p->context->eip = (uint)forkret;` 에서 현재 생성되는 process인 `p` 의 다음 실행 부분을 forkret() 으로 바꿔준다. 이렇게 되면 fork() 에서 생성된 `np` , 즉 자식 process는 fork() 가 끝나면 forkret() → trapret() 의 순서를 거치고 fork()를 호출한 부분으로 복귀하게 된다. 이때, fork() 에서 `np->tf->eax = 0;` 로 eax를 0으로 설정하였기 때문에 0이 반환되는 것을 알 수 있다.

exec(char *path, char **argv)

exec()은 fork() 와 달리 새로운 process를 위한 메모리를 할당하지 않고 exec() 에 의해 호출된 프로세스만 메모리에 남게 된다. fork() 의 경우 프로세스가 하나 더 생기는 것이지만,

exec()은 이 system call을 호출한 process의 pid를 새로운 process가 사용하게 되며, 기존의 process는 덮이게 된다.

initcode.S 에는 usys.S 에서 보던 코드와 비슷한 코드가 존재한다.

```
// usys.S
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

```
// initcode.S
// exec(init, argv)
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0
  movl $SYS_exec, %eax
  int $T_SYSCALL
```

프로그램에서 exec() system call을 호출하면 myfunction이 호출되었던 sequence와 유사하게 exec() system call이 호출된다. exec() 은 exec.c 에 정의되어있다. exec은 `if((ip = namei(path)) == 0)` 와 같이 namei 함수를 통해 `path` 에 해당하는 binary를 읽어온다. `if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))` 에서 처럼 ELF header를 읽어오고 ELF binary가 맞는지 확인을 한 다음 새로운 page table을 setupkvm() 을 통해 할당한다. 그 후 allocuvm() → loaduvm() 을 통해 메모리에 각각의 ELF segment를 load 한다. 이제 allocuvm() 을 통해 두개의 page를 할당하여 두번째 page를 user stack으로써 활용하는데, 이 stack에 argv, argc 그리고 fake return pc를 넣는다. 마지막으로 new image를 commit하고 old image를 해제시킨다. 즉 process가 다른 process로 대체된다.