OS Project01 - Wiki

2019040591
컴퓨터소프트웨어학부
박진수

TODO:

1. **getgpid() (get grandparent process id) 시스템 콜 구현**
2. **User program 구현**

Implementation:

1. getgpid() (get grandparent process id) 시스템 콜 구현

먼저, **xv6-public** 폴더 안에 시스템 콜(system call)들을 정의할 **project01_syscall.c** 파일을 생성했습니다.

시스템콜은 유저레벨의 프로그램이 커널(kernel)의 기능들을 사용하기 위해 만들어진 인터페이스(접근 도구)입니다. 직접 커널에 접근할 수 없는 유저 프로그램의 프로세스가 **시스템 콜을 호출하여** 커널에 특정 기능을 요청하면, 커널이 그 일을 대신 처리해주고 그 결과를 다시 **시스템 콜이 유저 프로세스에게 반환**해줍니다.

project01_syscall.c 파일 안에는 다음과 같은 내용을 작성했습니다.

제가 xv6의 코드를 조금 살펴본 결과, 시스템 콜들은 **sysproc.c** 파일 안에 모여있었지만, 추후 유지보수의 편의를 위해 제가 작성한 시스템콜은 따로 떼어내어 새로운 파일(project01_syscall.c)에 작성했습니다.

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

```
#include "syscall.h"

// System call for getppid()
int
sys_getppid(void)
{
  struct proc *currentProcess = myproc();  // Fetch the curre
  // Check if the current process or its parent is NULL.
  if(currentProcess == 0 || currentProcess->parent == 0)
    return -1;
  return currentProcess->parent->pid;
}

// System call for getgpid()
int
sys_getgpid(void)
{
  struct proc* currentProcess = myproc();  // Fetch the curre
  // Null check
  if(currentProcess == 0 || currentProcess->parent == 0 || cu
    return -1;
  return currentProcess->parent->parent->pid;
}
```

위 코드에서 **#include** 부분의 순서를 변경하면, 컴파일 과정에서 에러가 발생하니 주의해야 합니다. 처음 코드를 작성할 때, 코드 자동 포맷(auto format) 기능이 켜져있어서 자동으로 formatter가 줄의 순서를 바꿨는데, 왜 에러가 발생하는지 한동안 못찾아 애를 먹었던 기억이 납니다. 그래서 xv6 코드를 작성할 땐 해당 기능을 끄고 진행하는 것을 추천합니다.

위의 코드 중 **sys_getppid()**는 현재 프로세스의 부모의 pid를 반환하는 시스템콜입니다. 조부모(grandparent)의 pid를 호출하는 시스템콜(sys_getgpid())은 있는데, 부모의 pid를 반환하는 시스템콜은 없으면 뭔가 어색할 것 같아 임의로 추가했습니다.

sys_getppid() 와 **sys_getgpid()** 는 구현 방법이 거의 같아, 해당 과제의 요구 내용은 **sys_getgpid()** 에 대해서만 자세히 뜯어보도록 하겠습니다.

```
// System call for getgpid()
int
sys_getgpid(void)
{
```

```
  struct proc* currentProcess = myproc();  // Fetch the curre
  // Null check
  if(currentProcess == 0 || currentProcess->parent == 0 || cu
    return -1;
  return currentProcess->parent->parent->pid;
}
```

위 함수는 int형을 반환하는데, 정상적으로 실행되었을 때는 조부모의 pid를, 무언가 에러가 발생했을 때(null pointer exception)에는 -1을 반환하도록 했습니다. 해당 -1값은 추후 이 시스템콜에서 값을 전달받은 유저 함수가 예외 처리를 하는데에 사용됩니다.

위에서 **myproc()**의 경우, 현재 실행중인(running)프로세스의 구조체인 proc의 포인터를 반환합니다. **proc** 구조체에는 해당 프로세스에 대한 다양한 정보(pid 값, 부모 포인터, 상태, 메모리 레이아웃(layout), 컨텍스트(context) 정보 등) 들을 담고 있습니다.

아래는 proc 구조체의 모습입니다.

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

‘**현재 실행중인 process**’에 관해서 제가 헷갈렸던 부분이 있었는데, 유저프로그램이 실행될 때와 커널프로그램이 실행될 때 프로세스가 바뀌는지에 대한 여부였습니다. 결론적으로 말하면, 프로세스는 동일합니다. 맨 처음 유저 프로그램이 실행되어 새로운 프로세스가 생성되면, 그 프로세스 내에서 user mode, kerne mode 정보만 바뀔 뿐, 동일한 프로세스가 kernel 프로그램도 실행합니다.

아무튼, 본론으로 돌아와서 결과적으로 **currentProcess** 변수에는 제가 실행하게 될 유저 프로그램인 **project01**의 프로세스를 가리키게 됩니다. 그리고 그 아랫줄에 if 문은, 만약 해당 프로세스를 불러오는 과정에서 알 수 없는 이유로 NULL 값을 리턴한다거나, 부모 프로세스가 NULL이거나, 혹은 조부모 프로세스가 NULL인 경우 정상적인 결과를 출력할 수 없으므로 -1을 리턴하도록 하였습니다.

xv6의 경우 NULL값이 defined되어있지 않아, NULL과 동일한 의미인 0과 비교하도록 하였습니다.

그리고 마지막으로 정상적으로 조부모 프로세스에 접근하게 되었을 경우, 해당 조부모 프로세스의 pid값을 반환하도록 하였습니다.

이로써 project01_syscall.c 작성을 완료했습니다.

Implementation에서 잠시 벗어나서 아직도 헷갈리는 부분을 기록하려 합니다. 저는 과제를 진행하면서 실습 자료인 lab02를 참고하였는데요, 해당 자료에서는 prac_syscall.c(이번 과제의 project01_syscall.c 와 유사)에 myfunction 이라는 kernel function과 sys_myfunction이라는 wrapper function을 만듭니다. 그런데 구글 검색 결과 wrapper function은 user-space에서 유저들이 쉽게 시스템콜을 사용할 수 있도록 만든 인터페이스라고 설명이 되어 있습니다. 그래서 왜 myfunction과 sys_myfunction 이라는 두 개의 함수가 필요한지가 궁금합니다. 그리고 sys_myfunction을 실행할 때는, 아직 user mode 인건지 궁금합니다. 그래서 이번 과제에서는 제가 이해한 바에 따라서 오직 시스템콜에 해당하는 sys_getgpid() 함수 하나만 만들었습니다. 해당 wiki는 3월 28일에 작성하고 있고, 해당 내용은 실습 시간인 내일 질문을 드릴 예정입니다. 해당 과제를 구현하면서 느꼈던 점들을 기록하는 것도 의미가 있다고 생각하여 Wiki에 쓰게 되었습니다.

이후 Makefiles내 OBJS에 아래와 같이 추가합니다. 이 부분은 컴파일 과정에서 project01_syscall.o 파일을 포함시키라는 의미로, 해당 파일이 빌드될 수 있도록 만듭니다.

```
OBJS = \
...
project01_syscall.o\
```

실습 자료에서는 defs.h에 다음과 같이 다른 c 파일들에서 해당 함수를 찾을 수 있도록 추가하는 부분이 있었지만, 저는 유저 프로그램에서만 호출될 시스템콜 함수를 구현할 예정이므로 추가하지는 않았습니다.

```
// prac_syscall.c
int             myfunction(char*);
```

그 다음 syscall.h에 아래와 같이 각 시스템콜에 각각의 고유한 번호(식별자)를 부여합니다. 해당 번호는 커널과 유저 프로그램에서 해당 시스템콜을 참조할 때 사용됩니다. 유저 프로그램에서 해당 번호를 이용해 시스템콜을 호출하면, 커널에서는 그 번호를 통해 어떤 시스템콜이 요청되었는지를 알 수 있습니다.

```
#define SYS_getppid 23
#define SYS_getgpid 24
```

그 다음 syscall.c에 아래와 같은 코드를 추가하여, 방금 구현했던 시스템콜 함수(sys_getppid, sys_getgpid)를 해당 파일에서 사용할 수 있도록 함수를 선언합니다.

```
extern int sys_getppid(void);
extern int sys_getgpid(void);
```

위의 **extern** 키워드의 경우, 해당 함수의 구현(implementation)은 다른 파일에 있다고 표시한 것입니다.

그리고 아래의 코드는 아까 설정했던 시스템 콜 번호와, 각 번호에 맵핑되는 시스템 콜 함수(kernel space handler)를 연결시키는 역할을 하는 dispatch table을 구현한 모습입니다. index에는 시스템 콜 번호가, 그리고 각 인덱스에 대응되는 value에는 맵핑되는 시스템 콜 함수의 포인터가 위치해 있습니다.

```
static int (*syscalls[])(void) = {
    ...
[SYS_getppid] sys_getppid,
[SYS_getgpid] sys_getgpid,
};
```

그리고 이제 위에서 설정한 시스템콜을 유저 프로그램에서 호출할 수 있도록 인터페이스(접근 도구)를 선언합니다. 이제 유저 프로그램에서는 해당 함수를 불러서 대응되는 시스템 콜을 사용할 수 있습니다.

```
int getppid(void);
int getgpid(void);
```

그리고 usys.S 파일에 다음과 같은 코드를 추가합니다.

```
SYSCALL(getppid)
SYSCALL(getgpid)
```

위의 코드는 어셈블리어로 작성된 미리 작성된 SYSCALL 매크로를 사용하여, 유저 프로그램에서 사용할 시스템콜을 호출할 수 있도록 만들어줍니다. SYSCALL 매크로를 살펴보면, 아까 설정했던 각 시스템 콜에 해당하는 번호를 eax레지스터에 저장하고 시스템콜에 해당하는 소프트웨어 인터럽트를 실행합니다. 그러면 커널은 eax 레지스터에 들어있는 시스템콜 번호로 어떤 시스템 콜이 요청되었는지 알 수 있습니다. 위의 코드가 있어야 유저 프로그램에서 getppid()와 getgpid() 함수를 호출할 수 있습니다.

2. User program 구현

유저 프로그램 구현을 위해 xv6-public 폴더 안에 **project01.c** 파일을 생성합니다. 그리고 다음과 같이 코드를 작성했습니다.

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

int
main(int argc, char *argv[])
{
  int pid = getpid();
  int gpid = getgpid();
  printf(1, "My student id is 2019040591\n");

  // Exception handling
  if(pid < 0){
    printf(1, "getpid failed\n");
    exit();
  } else {
    printf(1, "My pid is %d\n", pid);
  }

  if(gpid < 0){
    printf(1, "getgpid failed\n");
    exit();
```

```
  } else {
    printf(1, "My gpid is %d\n", gpid);
  }

  exit();
}
```

위의 코드에서 “user.h”를 import 하여 우리가 위에서 작성했던 getgpid()함수와, 기존 xv6 에서 제공하는 getpid()의 선언들을 불러들입니다. getpid()함수는 현재 프로세스의 pid를 반환해주는 함수입니다.

제공되는 getpid()함수와 이번에 새로 만든 getgpid() 함수를 호출하여 각각에 해당하는(본인 & 조부모) pid 값을 pid와 gpid변수에 담습니다. 정상적으로 동작하였다면 음수가 아닌 정수값(pid)이 담겼을 것입니다.

그 아래에서 if문을 통해 pid에 음수(-1)가 담겼다면 오류 메시지를 출력하고 프로그램을 종료합니다.

아 그리고 위의 main 함수 안에서 종료시 exit()을 작성하지 않게 되면, 다음과 같은 에러 메세지가 해당 유저프로그램 실행 시 출력되게 됩니다.

```
pid 4 project01: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0
```

검색해보니 페이지 폴트 에러 메세지라고 합니다. 반드시 프로그램 실행이 끝났으면 exit()을 추가하도록 합시다.

그래서 결과적으로 shell에서 project01을 실행시 저의 학번(2019040591), 현재 프로세스의 pid, 조부모 프로세스의 pid가 출력되게 됩니다.

How to run:

다음과 같이 코드를 작성하고, 터미널 창에서 xv6-public 폴더로 이동합니다. 저의 경우 vscode에서 ssh연결을 통해 접속하여 바로 xv6-public 폴더로 연결이 되었지만, 만약 가상머신에서 프로그램을 실행하는 경우 다음과 같은 과정을 거치면 될 것이라고 생각합니다.

먼저, 콘솔에서 다음 명령어를 입력하여 사용자 폴더로 이동합니다. (그 이전에 먼저 이 경로에 xv6-public 폴더를 위치시켜 놓아야 합니다.

```
$ cd ~
```

그리고 다음 명령어를 통해 xv6-public 폴더로 들어갑니다

```
$ cd xv6-public
```

그리고 콘솔창에 다음과 같이 입력합니다.

```
$ make clean
$ make
$ make fs.img
```

이렇게 되면 빌드가 모두 완료됩니다.

이제, shell 프로그램을 실행시키기만 하면 됩니다. 아래의 명령어를 입력하면 됩니다.

```
$ qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img x
```

그런데 너무 명령어가 길기에, 셸스크립트 파일을 만들어 놓고, 앞으로는 간단한 명령어로 실행시킬 수 있도록 하겠습니다.

먼저 해당 xv6-public 폴더에 **bootxv6.sh** 라는 새로운 파일을 생성합니다. 그리고 다음과 같은 내용을 추가합니다.

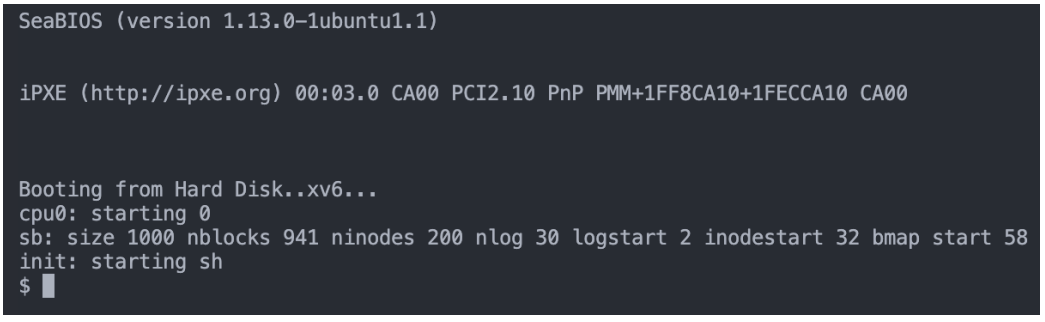
```
#!/bin/bash

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

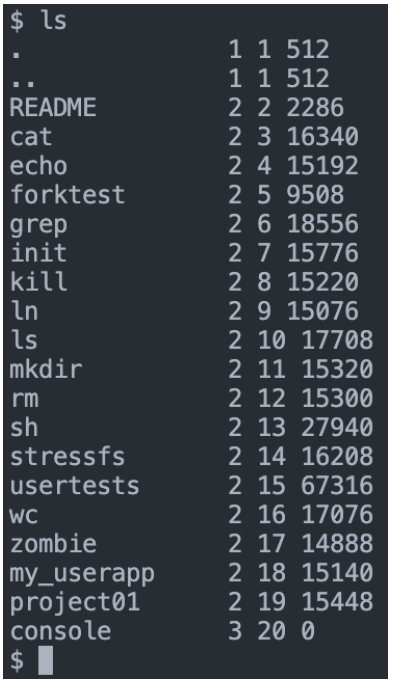
그리고 저장한 후, 콘솔에서 다음과 같이 입력합니다.

```
$ ./bootxv6
```

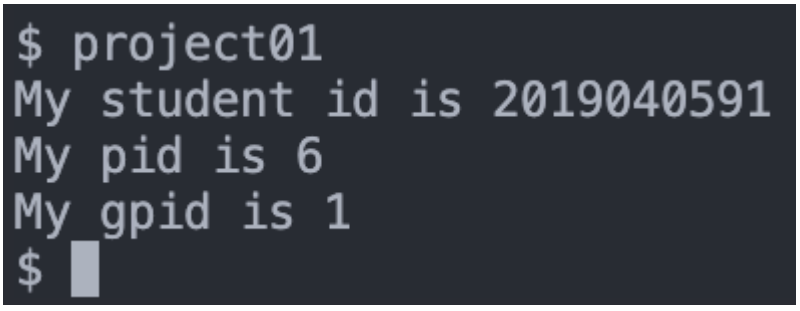
그러면 콘솔에 다음과 같은 shell 프로그램이 실행됩니다.



그리고 현재 실행할 수 있는 유저 프로그램의 정보를 보여주는 명령어인 **ls**를 입력합니다. 그러면 아래와 같이 유저프로그램이 쭉 나열됩니다.



그 후 제가 만든 project01 유저 프로그램을 실행하기 위해 **project01**을 입력합니다. 그러면 아래와 같이 정상적으로 출력되는 것을 확인할 수 있습니다.



이상으로 Wiki 작성을 마칩니다. 긴 글 읽어주셔서 감사합니다.