Operation System Project01

Department of Computer Science

2020099743

Dooik Kim

Dooik Kim
Professor Kang
ELE3021
31 March 2024

Operation System Project01

A system call allows an application program to access the kernel and manipulate the CPU. The goals of this project are mainly to create a system call named getgpid() that returns the process ID of the grandparent process and create a user program that prints out pid and gpid. To achieve this goal, it is necessary to analyze how getpid()-the system call that returns the process ID of itself-is implemented in xv6 via cscope. Moreover, a few alterations on ‘defs.h,’ ‘syscall.h,’ and ‘syscall.c’ are required to enable the kernel to access customized syscall. Additionally, changes in ‘user.h' and ‘usys.S’ are requisite to allow the user program to access a user-made syscall.

By using ‘:cs f e getpid’ command on cscope, it can be found that the definition of getpid() is placed in the file sysproc.c. The syscall getpid() returns myproc()->pid, which is the process ID of itself. More information can be noticed by analyzing the file sysproc.c, which is that most of the definitions of syscall functions related to the process are located inside sysproc.c. Hence, it can be inferred that the definition of getgpid() should be stated in sysproc.c since the syscall is associated with the process.

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

Picture 1. Definition of sys_getpid() in sysproc.c

```c
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)
};
```

Picture 2. Definition of struct proc

```c
int
sys_getgpid(void)
{
  return myproc()->parent->parent->pid;
}
```

Picture 3. Implementation of getgpid() in sysproc.c

Furthermore, by using the ':cs f g proc' command on cscope, it can be recognized that struct proc has member variables 'int pid' and 'struct proc *parent' which are process ID and a pointer to the parent process respectively. It is reasonably presumed that 'myproc()->parent->parent->pid' would return the grandparent process's process ID. The implementation of getgpid() is demonstrated in picture 3.

As mentioned before, modifications in 'defs.h', 'syscall.h', and 'syscall.c' are required to enable the kernel to exploit a custom syscall function. 'defs.h' adds a forward declaration to the new syscall, 'syscall.h' defines the position of the system call vector that connects to the implementation, and 'syscall.c' externally defines the function that connects the shell and the kernel.

```c
int             getgpid(void);
```

Picture 4. getpid declaration in defs.h

```c
#define SYS_getgpid 22
```

Picture 5. Syscall number declaration in syscall.h

```c
extern int sys_getgpid(void);
[SYS_getgpid] sys_getgpid,
```

Picture 6. Externally defined sys_getpid and insertion to system call vector in syscall.c

In addition, adjustments in 'user.h' and 'usys.S' are essential to permit a user program to access syscall. 'user.h' is usually included in the user program that manipulates syscall. Consequently, getgpid() must be defined in 'user.h' to be employed in the application program. On the other hand, 'usys.S' uses the macro to define and connect the call of the user to the system call function.

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

Picture 6. getgpid() defined in user.h

```
SYSCALL(getgpid)
```

Picture 7. Macro of getgpid in usys.S

By far, all the preparation to utilize custom syscall is done. The next task is to make an application that displays pid and gpid named 'project01.c.' The code of project01.c is shown below.

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

int
main(int argc, char* argv[])
{
    printf(1, "My student id is 2020099743\n");
    printf(1, "My pid is %d\n", getpid());
    printf(1, "My gpid is %d\n", getgpid());
    exit();
}
```

Picture 8. project01.c

The result of project01 is displayed below. The program project01 prints out the pid of itself 1 and the pid of grandparent process 3.

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
init: starting sh
$ project01
My student id is 2020099743
My pid is 3
My gpid is 1
$
```