COL331 Assignment 1: Report

Aneeket Yadav (2022CS11116)

August 2024

1 Introduction

This report outlines the basic design choices for implementing the required commands.

2 Steps to Add a New System Call

Since this is a recurring theme across different parts, the following general scheme was followed to implement a sycall:

Modify `kernel/syscall.h` to assign a unique system call number:

```
#define SYS_hello  22  // Assign a new unique number
```

Ensure that the number does not conflict with existing system calls. Modify `kernel/syscall.c` to include the new system call:

```
extern uint64 sys_hello(void);
// In the syscalls array:
[SYS_hello] = sys_hello,
```

Modify `kernel/sysproc.c` to define the system call:

```
uint64
sys_hello(void)
{
    printf("Hello from kernel\n");
    return 0;  // Success
}
```

Modify `user/user.h` to declare the system call:

```
int hello(void);
```

Modify `usys.S` to generate the system call stub:

```
SYSCALL(hello)
```

3 Part-2 : Enhanced Shell for xv6

The `login()` function was defined in `init.c`. The logic is straightforward. Macros `USERNAME` and `PASSWORD` were used, initialised via the Makefile. `login()` function was then called just before the infinite for loop in the main function of `init.c`.

4 Part-3 : Shell command - history

An array of 64 integers - `struct history_entry history[MAX_HISTORY]` was maintained to store the processes that are logged in history. `int history_count` was maintained as the position of the first vacant entry in the aforementioned array.

`struct history_entry` maintains the `pid`, `name`, and `mem_usage` of a process in the log.

The kernel function `sys_gethistory()` corresponding to `SYS_gethistory` is mapped to the user-space equivalent `gethistory()`.

A function `add_to_history(const struct proc *p)` is responsible for adding processes to `history[MAX_HISTORY]`.

`sh` is not included in history, unless it is explicitly killed by the command `kill <pid>`, in which case, both `sh` and `kill` are printed to console. Care has been taken to avoid that any incorrect command is not logged in history, which essentially implies that `exec()` syscall failed. Thus, we add an attribute `int exec_failed` to `struct proc` in `proc.h` and initialised to 0 in `exec.c`. If `(ip = namei(path)) == 0`, it means that arguments were not properly provided, following which the process is killed, thus we set `exec_failed=1`. We add a process to history when it exits only if its `exec_failed==1`.

The user space function is invoked in `sh.c` by matching the characters stored in the buffer.

5 Part-4 : Shell command - block

The standard procedure of defining a syscall remains same as before and will not be elaborated here for the sake of brevity.

We add the following components to the `struct proc` of a process-
`int syscall_bitmask[NUM_SYSCALLS]` and `int real_sh`. The former is an integer array whose each index indicates whether the corresponding syscall has been disabled in that shell or not - 1 for diabled and 0 otherwise. We defer an explanation of the latter's significance. Any process's `syscall_bitmask` is initialised with all zeroes i.e no restrictions exist when the process is first spawned.

`sys_block(int syscall_id)` sets `syscall_bitmask[syscall_id]=1`. Likewise, `sys_unblock(int syscall_id)` sets it to 0.

Exempla gratia, let us consider the following sequence of commands:

```
1 block 7
2 echo 3
3 sh
4 echo 5
```

As per the discussion on Piazza, 3 should be printed to the console whereas 5 should not because the intermediate shell spawned by the current shell should be able to call `execv()`, and since `echo` itself does not call `sys_execv()`, it should have no problem writing to the console. However, `sh` explicitly spawns a new shell and any child process of this should be unable to call `execv()`. We implement fine distinction between the two `sh` processes which must be recognised before one calls execv() by defining the attribute `int real_sh`, as hinted to earlier. Essentially, if the process is not a shell, then it may take its value to be 0 or undefined, 1 if it is a temporary shell and 2 if it was explicitly spawned by entering a command on the terminal. We then execute the below code logic to determine whether a syscall should be blocked or not in `syscall.c`. It is worth paying some attention to the difference between treatment of `SYS_exec` and other syscalls. Notice, the ancestral traversal to check if a particular syscall had been disabled for an ancestor(lines 18-30). Also, note that the syscalls for `init` and the first `sh` have not bee interfered with. Lets us consider

the condition at line 31(when `num=7` i.e `sys_exec` has been called.) We allow it to execute only when it is not an explicit shell and the corresponding disabling bit has not been set to 1 in any of its proper ancestors(i.e excluding the parent) - this allows us to distinguish between the intermediate `sh` processes - one spawned by the current shell and the other by its explicit child shell(the former is allowed to execute the syscall whereas the latter is not). The remaining explanation is trivial and is left an exercise to the reader.

```
1  void syscall(void)
2  {
3    int num;
4    struct proc *curproc = myproc();
5    num = curproc->tf->eax;
6
7    if (num > 0 && num < NELEM(syscalls) && syscalls[num])
8    {
9      if (curproc->pid <= 2)
10     {
11       curproc->tf->eax = syscalls[num]();
12     }
13     if (curproc->pid > 2) // Ignore init/system processes
14     {
15       int bit = curproc->parent->syscall_bitmask[num];
16       if (num == 7)
17       {
18         int is_bit_set_in_ancestors_except_parent = 0;
19         struct proc *p = myproc();
20         p = p->parent;
21         while (p->parent->pid != 1)
22         {
23           cprintf("p->pid=%d\n", p->pid);
24           if (p->parent->syscall_bitmask[num] == 1)
25           {
26             is_bit_set_in_ancestors_except_parent = 1;
27             break;
28           }
29           p = p->parent;
30         }
31         if ((curproc->real_sh != 2) && (is_bit_set_in_ancestors_except_parent == 0)) //
               Proper precedence
32         {
33           curproc->tf->eax = syscalls[num](); // Allow
34         }
35         else if (curproc->real_sh == 2)
36         {
37           curproc->tf->eax = syscalls[num](); // Allow
38         }
39         else
40         {
41           cprintf("Syscall␣%d␣blocked␣for␣PID␣%d\n", num, curproc->pid);
42           curproc->tf->eax = -1; // Return error to indicate syscall is blocked
43         }
44       }
45       else
46       {
47         if (bit == 0)
48         {
49           curproc->tf->eax = syscalls[num]();
50         }
51         else
52         {
53           cprintf("Syscall␣%d␣blocked␣for␣PID␣%d\n", num, curproc->pid);
54           curproc->tf->eax = -1; // Return error to indicate syscall is blocked
55         }
56       }
57     }
58   }
59 }
```