COL331 Assignment 1

Process Management and System Calls

Utkarsh Singh, 2015ME10686

February 17, 2018

Objective

To modify the xv6 kernel to achieve the following functionalities:

1. System Call Tracing
   (a) Print out a line for each system call invocation, and the number of times it has been called.
   (b) Introduce a sys_toggle() system call to toggle the system trace printed on the screen.
2. sys_add() System Call - To add a system call that takes two integer arguments and return their sum.
3. sys_ps() System Call - To add a system call that prints a list of all the current running processes.

Procedure

Each part is implemented in the order mentioned above.

Part 1(a)

In order to print out the system trace along with it's count, the following two global arrays are introduced in *syscall.c*

```
int count_syscalls[24];//Maintains count of each syscall
char *name_syscalls[] //Array containing all syscall names
= {"sys_fork","sys_exit","sys_wait","sys_pipe","sys_read",
"sys_kill","sys_exec","sys_fstat","sys_chdir","sys_dup",
"sys_getpid","sys_sbrk","sys_sleep","sys_uptime","sys_open",
"sys_write","sys_mknod","sys_unlink","sys_link","sys_mkdir",
"sys_close", "sys_toggle", "sys_add", "sys_ps"};
```

(Note that the extra system calls that will be introduced later have already been accounted for in these arrays. Their declarations and definitions are explained later.)

The count_syscalls array keeps the count of the number of times each system call has been called since booting xv6, and the name_syscalls array stores the names of all system calls.

The syscall() function in *syscall.c* was modified as follows to print the system trace, along with keeping track of the system call count. Below is the syscall() function from *syscall.c*, with the modifications mentioned as "added".

```
void syscall(void)
{
  int num;
  struct proc *curproc = myproc();
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    count_syscalls[num-1] = count_syscalls[num-1]+1; //added
    if(check_systrace()) //is explained later
      cprintf("%s %d\n",
      name_syscalls[num-1], count_syscalls[num-1]); //added
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

(Note that check_systrace() is a part of Part 1(b) and will be defined later).

This finishes Part 1(a).

Part 1(b)

For this, we have to declare and define the sys_toggle system call. toggle() will be the user function that will call the system call. This is done as follows.

1. Add the following in *syscall.c*.

   ```
   extern int sys_toggle(void);
   extern int check_systrace(void);
   ```

   To see how check_systrace() is being used, refer to snippet in Part 1(a).

2. Add the following in *syscall.h*.

   ```
   #define SYS_toggle 22
   ```

3. Add the following to the array of functions in *syscall.c*.

   ```
   [SYS_toggle]  sys_toggle
   ```

4. Now let's define the sys_toggle system call and check_systrace() (it is a helper function). These definitions will be given in *sysproc.c*. Add the following snippet to this file.

```
int systrace_mode;
int sys_toggle(void)
{
  systrace_mode = 1-systrace_mode;
  return 0;
}
int check_systrace(void) //Not a syscall
{
  if(systrace_mode == 0)
    return 1;
  return 0;
}
```

5. Add the following in *usys.S*.

```
SYSCALL(toggle)
```

6. Finally add the following in *user.h*

```
int toggle(void);
```

This completes Part 1(b).

Part 2

For this, we have to declare and define the sys_add system call. add() will be the user function that will call the system call. This is done as follows.

1. Add the following in *syscall.c*.

```
extern int sys_add(void);
```

2. Add the following in *syscall.h*.

```
#define SYS_add 23
```

3. Add the following to the array of functions in *syscall.c*.

```
[SYS_add]   sys_add
```

4. Now let's define the sys_add system call. The definition will be given in *sysproc.c*. Add the following snippet to this file.

```
int sys_add(int a, int b)
{
  argint(0, &a);
  argint(1, &b);
  return a+b;
}
```

5. Add the following in *usys.S*.

   ```
   SYSCALL(add)
   ```

6. Finally add the following in *user.h*

   ```
   int add(int, int);
   ```

This completes Part 2.

Part 3

For this, we have to declare and define the sys_ps system call. ps() will be the user function that will call the system call. This is done as follows.

1. Add the following in *syscall.c*.

   ```
   extern int sys_ps(void);
   ```

2. Add the following in *syscall.h*.

   ```
   #define SYS_ps 24
   ```

3. Add the following to the array of functions in *syscall.c*.

   ```
   [SYS_ps]  sys_ps
   ```

4. Now let's define the sys_ps system call. The definition will be given in *sysproc.c*. Add the following snippet to this file.

```
extern void get_pid_name(void);
int sys_ps(void)
{
  get_pid_name();
  return 0;
}
```

5. Add the following in *usys.S*.

```
SYSCALL(ps)
```

6. Finally add the following in *user.h*

```
int ps(void);
```

7. The sys_ps system call used a function get_pid_name() which does the main job of printing the process name and id. The definition will be given in *proc.c*. Add the following snippet to this file.

```
void get_pid_name(void)
{
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    int check = p->state;
    if(check == UNUSED || check == EMBRYO || check == ZOMBIE)
      continue;
    cprintf("pid:%d name:%s\n", p->pid, p->name);
  }
}
```

This completes Part 3, and also concludes the Assignment.