

Assignment 1 Report

Anoop (2015CS10265)

February 17, 2018

Part 1 - System Call Tracing

Printing the Trace

Listing 1: Modification in “syscall” function - syscall.c

```
235 if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {  
236     track_syscalls(curproc);  
237     curproc->tf->eax = syscalls[num]();  
238 } else {  
239     cprintf("%d %s: unknown sys call %d\n",  
240         curproc->pid, curproc->name, num);  
241     curproc->tf->eax = -1;  
242 }
```

At line 236 in Listing 1, the function `track_syscalls` is used to print the syscalls that are issued only if the `toggle_state` is set to 1.

Listing 2: `track_syscalls` function implementation - syscall.c

```
139 int syscall_count[24] = {0};  
140 // Syscall identification  
141 void  
142 track_syscalls(struct proc *p)  
143 {  
144     ++syscall_count[p->tf->eax - 1];  
145     if(toggle_state == 1)  
146     {  
147         switch(p->tf->eax)  
148         {  
149             case 1:  
150                 cprintf("%s %d\n", "sys_fork", syscall_count[p  
->tf->eax - 1]);  
151                 break;  
152             case 2:  
153                 .  
154                 .
```

```

221         default:
222             cprintf("Unknown syscall\n");
223             break;
224     }
225 }
226 }

```

`track_syscalls` function implementation uses a global array (`syscall_count`) to maintain the count of the syscalls issued. The function uses **switch-case** statements to print appropriate syscall count when called only if the `toggle_state` is set to 1.

Toggling the Printing of Trace

Listing 3: `sys_toggle` function implementation - `sysproc.c`

```

93 // toggles the system trace (1 - ON | 0 - OFF)
94 int toggle_state = 1;
95 int
96 sys_toggle(void)
97 {
98     toggle_state = toggle_state == 0 ? 1 : 0;
99     return toggle_state;
100 }

```

For the `sys_toggle` system call implementation, a global variable (`toggle_state`) has been used to indicate the state of the system trace. Corresponding additions of definitions have been done in files - `user.h`, `syscall.h`, `usys.S`. An if statement has been added to `syscall.c` to check the `toggle_state` before printing the trace (see code - Listing 1). `user_toggle.c` file has been added to test the toggle system call implementation.

Part 2 - `sys_add` System Call

Listing 4: `sys_add` function implementation - `sysproc.c`

```

102 // add two numbers
103 int
104 sys_add(void)
105 {
106     int a;
107     int b;
108
109     if(argint(0, &a) < 0)
110         return -1;
111     if(argint(1, &b) < 0)
112         return -1;
113     return a + b;
114 }

```

For the `sys_add` system call implementation, a function that takes two arguments and returns the sum is used. The function checks for two arguments and if not present returns `-1` as an error code. Corresponding additions of definitions have been done in files - `user.h`, `syscall.h`, `usys.S`, `syscall.c`.

Part 3 - sys_ps System Call

Listing 5: `sys_ps` function implementation - `proc.c`

```
270 // list all processes
271 int
272 sys_ps(void)
273 {
274     struct proc *p;
275
276     acquire(&ptable.lock);
277
278     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
279         if(p->pid != 0 && p->state != ZOMBIE){
280             cprintf("pid:%d name:%s\n",p->pid,p->name);
281         }
282     }
283
284     release(&ptable.lock);
285     return 0;
286 }
```

For the `sys_ps` system call implementation, a function that loops over the `ptable` has been used. The function locks the `ptable` during the loop and prints the pid and name of process with non-zero pid. The zombie processes are ignored in this implementation. Corresponding additions of definitions have been done in files - `user.h`, `syscall.h`, `usys.S`, `syscall.c`. Also an `extern` definition has been added to `sysproc.c`.