Assignment 3
CS 3305b

Paul Vrbik

250389673

February 26, 2010

Part 1: Your first Minix Hacks

Minix source file `/user/src/servers/pm/exec.c`

Modified line 70 (`name_buf[]` was printed).

Printed

```
172.16.225.132 login:
/bin/gettyin/ssh
/sbin/gettyin/ssh

Password:
/usr/lib/pwdauth
/bin/sh/pwdauth
/usr/bin/stty
```

Part 2: A "Simple" System Call

This code was copied from the code for `top`.

```
1  #define PROCS (NR_PROCS+NR_TASKS)
2  PUBLIC int do_numberprocs(void) {
3  int p, alive;
4  static struct proc proc[PROCS];
5
6      sys_getproctab(proc);
7
8      alive = 0;
9
10     for(p = 0; p < PROCS; p++) {
11         if(p - NR_TASKS == IDLE)
12             continue;
13         if(proc[p].p_rts_flags & SLOT_FREE)
14             continue;
15         alive++;
16     }
17     return alive;
18 }
```

Part 3: Comparing Scheduling Algorithms

Original

Trial	Real	User	System
1	1:37.20	1.16	6.90
2	1:35.20	1.56	8.80
3	1:36.91	1.88	9.71

Proc1.c
(single user queue)

Trial	Real	User	System
1	1:41.33	1.48	3.98
2	1:33.78	1.53	8.25
3	1:33.76	1.35	6.41

Proc2.c
(single user queue, modified enqueue)

Trial	Real	User	System
1	6:41.43	1.08	5.41
2	6:38.56	1.03	6.08
3	1:33.76	0.88	5.46

Proc3.c
(least `p_user_time` first)

Trial	Real	User	System
1	0:46.95	1.16	9.48
2	0:35.11	2.33	11.93
3	0:42.36	2.11	10.71

Proc1.c The actual change was made in `proc.h` to the environment variable `USER_Q`. This variable was changed from 7 to 1 (which changes the number of user queues to one).

Timings here are basically unchanged because reducing the number of queues from seven to one—when there are only two user processes—won't have any significant affects. There is nothing to gain here from having higher priority queues. Two processes can't even occupy more than one queue in the first place (one will be running, the other queued)!

Proc2.c The single user queue was implemented as in `Proc1.c`. The new queue requirement was done by making a trivial change to the `enqueue()` procedure. Namely, the part of the "if" statement that would add to the front of the queue was removed.

Modifying the `enqueue()` procedure in the prescribed way has the unfortunate effect of never allowing i/o bound processes to preempt cpu bound processes. Here `scanfiles` will only get control of the cpu when `timewaste` blocks or is interrupted. The longer timings are a result of this.

Proc3.c The procedure `pick_proc.c` was changed to the following:

```
1  PRIVATE void pick_proc()
2  {
3    register struct proc *rp;                          /* process to run */
4    int q;                                             /* iterate over queues */
5    int MinTime, MinProc;
6
7    MinTime = -1;
8
9    for (q=0; q < NR_SCHED_QUEUES; q++) {
10     if ( (rp = rdy_head[q]) != NIL_PROC) {
11         if (MinTime == -1) {
12             MinTime = rp -> p_user_time;
13             MinProc = q;
14         } else if ( MinTime > rp -> p_user_time) {
15             MinTime = rp -> p_user_time;
16             MinProc = q;
17         }
18     }
19   }
20
21   rp = rdy_head[MinProc];
22   next_ptr = rp;
23
24   if (priv(rp)->s_flags & BILLABLE)
25     bill_ptr = rp;                          /* bill for system time */
26
27   return;
28 }
```

Here we have the opposite scenario as in `Proc2.c`. Now processes which have the least amount of cumulative time on the cpu (i.e. i/o bound processes) will *always* preempt those processes "hogging" the CPU. As a result we see that `scanfiles` finishes much quicker — which seems like a good thing here, but would result in much longer wait times for CPU bound processes.