COMSW4118-16-1

MIDTERM REVIEW

10/26/2017

- test_and_set() is typically implemented by locking the memory bus while accessing the variable
- system calls: program
  - instr 1
  - instr 2
  - ① software interrupt (from syscall) → ② jump to kernel in interrupt handler (entry.S) and dispatch to software ③ interrupt handler (i.e. the system call handler)
  - instr 3

  ④ syscall args are stored (registers, but depends on arch.)
  ⑤ determine syscall ~~tabl~~ number
  ⑥ Lookup number is system call table (number → function)
  ⑦ Call the respective function
  ⑧ Unwind / back to user space
  - check for signals
  - check if scheduler needs to be called
  - switch to user mode
  - Restore state (registers, etc.) to return to user program that was executing
- To create a system call (for ARM64): (Q4 on sample midterm):

  ① include/linux/syscalls.h - add func prototype

  asmlinkage long sys-pinfo (pid_t pid, struct procstruct __user *proc)

  ↑ int passed by value, so no __user necessary

  ② Modify the system call table:

  include/uapi/asm-generic/unistd.h ← ORDER MATTERS! you can't stick the 245 at any line, put it after the 244 syscall.

  ```
  # define __NR_pinfo 245
  __SYSCALL (__NR_pinfo, sys-pinfo)
  ```

  ③ Write the function for the new syscall.

```
~~DEFINE_SYSCALL~~ SYSCALL_DEFINE2( pinfo, pid_t, pid, struct proc_struct __user *,
                                   proc) {
        ...
}

if (pid < 0)
    return -EINVAL;

task_struct *p;
~~P= find_process~~ read_lock(&tasklist_lock);
P= find_process_by_pid(pid);
if (!p) {          read_unlock(&tasklist_lock);
   return -ES[redacted]RCH;
}
```

↓ cont...

COMSW4118-16-2 10/26/2017

NOTE: no need to lock the task, (task lock) as we do not access stuff that can change (ex. cmd). The tasklist_lock protects the task list, hence the parent/sibling nodes.

... cont.

```
struct proc_struct ps;
ps.pid = task_pid_nr(p);
ps.parent_pid = task_pid_nr(
    p->real_parent->pid);
ps.nice = task_nice(p);
read_unlock(tasklist_lock);
if (copy_to_user(proc, &ps, sizeof(ps))) {
    return -EFAULT;
return 0;
```

NOTE: task_struct.pid is the internal kernel PID, what you need is what the user will see, which is provided by task_pid_nr, which translates it to the appropriate architecture type, & namespace!

NOTE: task_struct.pid will be deprecated!

Basically, what task_pid_nr does is translate it to the user namespace (which probably is the same).

- To call the system call:

```
struct proc_struct ps;
syscall(__NRpinfo, 1, &ps);
```

- Timer interrupt handler (Q2 on sample midterm):

a) COUNT is being incremented by the hardware. You can set it atomically to 0 and read it atomically, but not both! If you update it, you will likely skip microsecond counts. The solution in a) won't work.

b) 64 bits will be overflown in 564 years or so if we just keep adding, so just increasing COUNT by 1 every microsecond is OK! Same for compare, so just set COMPARE += 1000. (also, increment TICK by 1, of course). Note that COUNT -= COMPARE is not atomic and won't work.

c)

```
while (COMPARE <= COUNT) {
    COMPARE += 1000;
    TICK++;
}
```

NOTE: It will work since it is sufficient for the COMPARE to be < COUNT for some point in time. This will guarantee that the HW will generate an interrupt when COUNT becomes == COMPARE and the handler will be eventually called again, even though a high priority interrupt keeps it from completion right after exiting the while loop (the interrupt flag is set → handler will be called ASAP again).

- TRUE/FALSE questions (Q1 on sample)

a) TRUE ← FALSE!

b) "monitor mode" = "kernel mode" = "supervisor mode"

(!) change to monitor mode is NOT protected!

c) FALSE (I/O is generally privileged)

d) TRUE e) TRUE f) FALSE g) TRUE