6.004 Computation Structures
Spring 2009

## Operating system issues

---

Problem 1. The following code outlines a simple timesharing scheduler:

```
struct MState {
  int Regs[31];                 /* saved state of user's registers */
} User;

int N = 42;                     /* number of processes to schedule */
int Cur = 0;                    /* number of "active" process */

struct PCB {
  struct MState State;          /* processor state */
  Context PageMap;              /* VM map for process */
  int DPYNum;                   /* console/keyboard number */
} ProcTbl[N];                   /* one per process */

Scheduler() {
  ProcTbl[Cur].State = User;    /* save current user state */
  Cur = (Cur + 1)%N;            /* increment modulo N */
  User = ProcTbl[Cur].State;    /* make another process the current one */
}
```

Suppose that each time the user hits a key on the keyboard, an interrupt is generated and the interrupt handler copies the new character into a kernel-resident input buffer. The operating system includes a ReadKey service call (SVC) which can be invoked by the user to read the next character from the input buffer. If the input buffer is empty, the SVC should "hang" until a character is available.

The first draft of the ReadKey SVC handler is shown below. The SVC handler routine saves the user's state in the User structure and then call ReadKey_h(). When ReadKey_h() returns, the SVC handler restores the user's state and then does a JMP(XP) to restart the user's program.

```
ReadKey_h() {
  int kdbnum = ProcTbl[Cur].DPYNum;
  while (BufferEmpty(kdbnum)) {
    /* busy wait loop */
  }
  User.Regs[0] = ReadInputBuffer(kdbnum);
}
```

A. ★ After executing a ReadKey SVC, where will the user program find the next character from the input buffer?

The SVC stores the character read from the input buffer in the R0 slot of the user structures. This is loaded into the user's R0 just before the kernel returns to user mode.

B. ★ Explain what's wrong with this proposed implementation.

If the buffer is empty when the user makes the ReadCh supervisor call, the handler loops in the kernel with interrupts disabled. This prevents the buffer from being filled and so the code enters an endless loop.

C. ★ A second draft of the keyboard handler is shown below:

```
ReadKey_h() {
  int kdbnum = ProcTbl[Cur].DPYNum;
  if (BufferEmpty(kdbnum))
    User.Regs[XP] = User.Regs[XP] - 4;
  else
    User.Regs[0] = ReadInputBuffer(kdbnum);
}
```

Explain how the modifications fix the problems of the initial implementation.

When the buffer is empty, this code returns to user mode and re-executes the SVC. By returning to user mode, interrupts are enable (briefly!) so interrupts can happen and the buffer will eventually be filled when a key is typed.

D. ★ The designers notice that the process just wastes its time slice waiting for someone to hit a key. So they propose the following modifications:

```
ReadKey_h() {
  int kdbnum = ProcTbl[Cur].DPYNum;
  if (BufferEmpty(kdbnum)) {
    User.Regs[XP] = User.Regs[XP] - 4;
    Scheduler();
  } else
    User.Regs[0] = ReadInputBuffer(kdbnum);
}
```

What would be the most likely effect of removing the line of code where User.Regs[XP] is decremented by 4?

When next scheduled, the user program will *not* re-execute the SVC and will proceed assuming that the next character is available in R0. In short, it will process a "garbage" character, ie, use whatever happened to be in R0 at the time of the original SVC.

E. ★ Explain how this modification improves the overall performance of the system (assume the decrement by 4 has not been removed).

Calling scheduler() permits other programs to run while waiting for the next character to arrive.

F. ★ This version of the handler still doesn't prevent the process from being scheduled each quantum, even though it may just call Scheduler() once again if no character has appeared in the input buffer. Explain how the sleep(status) and wakeup(status) kernel routines described in lecture can be used to make sure that processes are only scheduled when there is something "useful" for them to do.

By calling sleep(status), the handler can notify the kernel when to next schedule the user's process for execution (ie, after a character has been typed). This avoids rescheduling the process for execution when all it will do is discover that the buffer is still empty and suspend itself.

If the keyboard handler calls wakeup(status), the kernel can locate all processes that went to sleep waiting for an event that's related to "status". It can then mark those processes as ready for execution once again. Thus processes are only run when there's something good reason for them to proceed.

---

Problem 2. The following questions are about the simple timesharing kernel used in Lab 8. Click here to view the code in a separate window.

A. ★ What happens right after reset?

When the hardware finishes reset, it starts execution at location 0. Looking at the code, there is a BR (I_Reset) at 0 which directs us to the following code fragment:

```
CMOVE(P0Stack,SP)
CMOVE(P0Start,XP)
JMP(XP)
```

These instructions initialize the stack pointer for user process #0 and then jump to the first instruction for that process. Since we're just starting, the program will make no assumptions about the contents of the other registers, so we'll just let them have whatever value they happen to have. Note that this means buggy programs (eg, those that test register values before setting them) might behave in different ways after separate resets. To provide some guarantee of reproducibility, many OSes initialize a processes' registers to known values before starting the process.

B. ★ How does the processor get to execute the other user processes?

Clock interrupts are issued periodically by the hardware (a common interval between interrupts in 1/60 sec = 16.6ms); each interrupt is called a "tick" of the clock. Our interrupt handler (I_CLK) backs up XP to point to the interrupted instruction and saves the contents of register 0. It then checks to see how long the current user process has been running. If it's less than two ticks, the contents of register 0 is restored and execution of the interrupted program is resumed.

If the user process has finished its 2-tick quanta, all of the current process' registers are saved in a kernel data structure (UserMState) and we make a call to Scheduler() to determine which process to run next.

Scheduler() is pretty simple: it just cycles between the user processes in a round-robin fashion. Changing the current processes involves copying UserMState into the appropriate structure for the current process, and then copying the data for the next process into UserMState.

Finally, the kernel restores the user-mode state (see I_Rtn) and resumes execution of the user-mode process. Since we swapped which process state was "current", we actually resume a different process than we interrupted with the clock tick.

C. ★ How do supervisor calls (SVCs) work?

Supervisor calls are kernel-mode routines that provide specific services for user-mode programs. They happen in the kernel because we want to virtualize some resource or because the operation requires some privilege we don't want to grant to a user-mode process to use willy-nilly.

From a programmer's point of view, SVCs look like individual instructions whose operands appear in specific registers, on the stack, etc. (the exact convention depends on the particular SVC). At the hardware level the SVCs are just particular illegal instructions (in this example, they are instructions with an opcode of 1). When executed, they cause the expected illegal instruction trap which saves the PC+4 of the illegal instruction in XP, enters kernel mode, and branches to location 4.

The illegal instruction handler (I_IllOp) dispatches to an appropriate handler based on the opcode of the illegal instruction. Instructions with an opcode of 1 end up at SVC_UUO, all other instructions end up at UUOError.

The supervisor call handler (SVC_UUO) performs another level of dispatch based on the low-order bits of the illegal instruction, transferring control to one of eight possible supervisor call handlers. When the handler completes, the illegal instruction handler returns, resuming execution of the user-mode program with the instruction following the SVC.

D. ★ How do characters typed on the keyboard find their way to user-mode programs?

The process involves several steps:

1. *Keyboard interrupt*. Each time a character is typed, the hardware generates an interrupt, setting the PC to 0xC. The kernel-mode handler (I_Kbd) retrieves the character using the RDCHAR() instruction -- a priveledged instruction that can only be executed by kernel-mode programs. It saves the character in a kernel data buffer and resumes execution of the interrupted user-mode program.

2. *GetKey() supervisor call*. If a user-mode program wants to read a character from the keyboard it executes the GetKey() supervisor call. This allows the OS to implement various policies -- for example: how the keyboard is shared among user-mode programs (input focus), buffering of keyboard input for compute-intensive programs, etc.

If there's a character available in the kernel buffer, it's loaded into the user's register 0 and the program that issued the service call resumes execution just after the GetKey() SVC instruction (with the new character loaded into R0).

If there isn't a character available, the GetKey() handler wants to "hang" (ie, not return to the user-mode program) until a character is typed on the keyboard. But as discussed in the previous problem, it won't work to simply loop in the kernel (interrupts are disabled and we would waste CPU cycles that might be usefully employed by other user-mode programs). So the user-mode state is modified so that the GetKey() SVC will be re-executed when that program resumes execution, and the scheduler is invoked to run a *different* user-mode program. Every so many clock ticks the original program will resume execution, re-execute the GetKey() SVC, etc. This process continues until a character is typed and can be returned to the user.

---

Problem 3. Real Virtuality, Inc markets three different computers, each with its own operating system. The systems are:

**Model A**: A timeshared,multi-user Beta system whose OS kernel is uninterruptable.

**Model B**: A timeshared Beta system which enables device interrupts during handling of SVC traps.

**Model C**: A single-process (not timeshared) system which runs dedicated application code.

Each system runs an operating system that supports concurrent I/O on several devices, including an operator's console including a keyboard. Les N. Dowd, RVI's newly-hired OS expert, is in a jam: he has dropped the shoebox containing the master copies of OS source for all three systems. Unfortunately, three disks containing handlers for the ReadKey SVC trap, which reads and returns the ASCII code for the next key struck on the keyboard, have gotten confused. Of course, they are unlabeled, and Les isn't sure which handler goes into the OS for which machine. The handler sources are

```
ReadCh_h() {                              /* VERSION R1 */
  if (BufferEmpty(0))                     /* Has a key been typed? */
    User->Regs[XP] = User->Regs[XP]-4;    /* Nope, wait.  */
  else
    User->Regs[0] = ReadInputBuffer(0); /* Yup, return it. */
}

ReadCh_h() {                              /* VERSION R2 */
  int kbdnum=ProcTbl[Cur].DpyNum;
  while (BufferEmpty(kbdnum)) ;           /* Wait for a key to be hit*/
  User->Regs[0] = ReadInputBuffer(kbdnum);  /*...then return it. */
}

ReadCh_h() {                              /* VERSION R3 */
  int kbdnum=ProcTbl[Cur].DpyNum;
  if (BufferEmpty(kbdnum)) {              /* Has a key been typed? */
    User->Regs[XP] = User->Regs[XP]-4;    /* Nope, wait. */
    Scheduler();
  } else
    User->Regs[0] = ReadInputBuffer(kbdnum);    /* Yup, return it. /
}
```

A. Show that you're cleverer than Les by figuring out which handler goes with each OS, i.e., for each operating system (A, B and C) indicate the proper handler (R1, R2 or R3). Briefly explain your choices.

R1 goes with C. This handler hardwires the keyboard to port 0, so it must be Model C.

R2 goes with B. This handler is for a timeshared system, but does not yield when waiting so the SVC must be interruptible.

R3 goes with A. This handler is also for a timeshared system, but calls the scheduler before waiting so the kernel may not interrupt the handler on its own.

B. But Les isn't that clever. In order to figure out which handler code goes with each OS version, Les makes copies of each disk and distributes them as "updates" to beta-test teams for each OS. Les figures that if each handler version is tried by some beta tester in each OS, the comments of the testers will allow him to determine the proper OS for each handler.

Les sends out the alleged source code updates, routing each handler source to testers for each OS. In response, he gets a barrage of complaints from many of the testers. Of course, he's forgotten which disk he sent to each tester. He asks your help to figure out which combination of system and hander causes each of the complaints. For each complaint below, explain which handler and which OS the complainer is trying to use.

Complaint: "I get linkage errors; Scheduler and ProcTbl are undefined!"

Handler R3 and OS C. R3 is the only handler using Scheduler and C is the only kernel without it

C. Complaint: "I can link up the system using the new handler, but the system hangs when my application tries to read a key."

Handler R2 and OS A. R2 is the only handler that won't yield if no key is available and A is the only kernel that timeshares but has interrupts disabled when running in kernel mode (so the loop in the R2 handler that waits for a key to be hit can never be interrupted).

D. Complaint: "Hey, now the system always reads everybody's input from keyboard 0. Besides that, it seems to waste a lot more CPU cycles than it used to."

Handler R1 and OS A. R1 is the only handler that assumes keyboard 0. The model A user was using a handler (R3) which called scheduler when the buffer was empty which in theory wastes fewer CPU cycles than busy looping. So when the user substitutes R1, the program busy loops when the buffer is empty. The model B user wouldn't have the same complaint since the handler she was using (R2) already loops.

E. Complaint: "Neat, the new system seems to work fine. It even seems to waste less CPU time than it used to!"

Handler R3 and OS B. The model B user is happy because we've replaced her kernel busy loop in her old handler (R2) with a call to Scheduler().