COMP3231/9201/3891/9283 Operating Systems 2020/T1

UNSW

Tutorial Week 5

Questions and Answers

Kernel Entry and Exit

1. What is the `EPC` register? What is it used for?

   This is a 32-bit register containing the 32-bit address of the return point for the last exception. The instruction causing (or suffering) the exception is at `EPC`, unless `BD` is set in `Cause`, in which case `EPC` points to the previous (branch) instruction.

   It is used by the exception handler to restart execution at the at the point where execution was interrupted.

---

2. What happens to the `KUc` and `IEc` bits in the `STATUS` register when an exception occurs? Why? How are they restored?

   The 'c' (current) bits are shifted into the corresponding 'p' (previous) bits, after which `KUc = 0`, `IEc = 0` (kernel mode with interrupts disabled). They are shifted in order to preserve the current state at the point of the exception in order to restore that exact state when returning from the exception.

   They are restored via a `rfe` instruction (restore from exception).

---

3. What is the value of `ExcCode` in the `Cause` register immediately after a system call exception occurs?

   The value of `ExcCode` is 8.

---

4. Why must kernel programmers be especially careful when implementing system calls?

   System calls with poor argument checking or implementation can result in a malicious or buggy program crashing, or compromising the operating system.

---

5. The following questions are focused on the case study of the system call convention used by OS/161 on the MIPS R3000 from the lecture slides.
   1. How does the 'C' function calling convention relate to the system call interface between the application and the kernel?
   2. What does the most work to preserve the compiler calling convention, the system call wrapper, or the OS/161 kernel.
   3. At minimum, what additional information is required beyond that passed to the system-call wrapper function?

a. The 'C' function calling convention must always appear to be adhered to after any system-call wrapper function completes. This invovles saving and restoring of the *preserved* registers (e.g., s0-s8, ra).

   The system call convention also uses the calling convention of the C-compiler to pass arguments to OS/161. Having the same covention as the compiler means the system call wrapper can avoid moving arguments around and the compiler has already placed them where the OS expects to find them.

b. The OS/161 kernel code does the saving and restoring of preserved registers. The system call wrapper function does very little.

c. The interface between the system-call wrapper function and the kernel can be defined to provide additional information beyond that passed to the wrapper function. At minimum, the wrapper function must add the system call number to the arguments passed to the wrapper function. It's usually added by setting an agreed-to register to the value of the system call number.

---

6. In the example given in lectures, the library function *read* invoked the `read` system call. Is it essential that both have the same name? If not, which name is important?

   System calls do not really have names, other than in a documentation sense. When the library function `read()` traps in to the kernel, it puts the number of the system call into a register or on the stack. This number is used to index into a jump table (e.g. a C *switch* statement). There is really no name used anywhere. On the other hand, the name of the library function is very important, since that is what appears in the program.

   Note: The kernel programmer may can the code in the operating system that implements `read` a similar name, e.g., `sys_read`. This is purely a convention for the sake of code clarity.

---

7. To a programmer, a system call looks like any other call to a library function. Is it important that a programmer know which library function result in system calls? Under what circumstances and why?

   As far as program logic is concerned it does not matter whether a call to a library function results in a system call. But if performance is an issue, if a task can be accomplished without a system call the program will run faster. Every system call involves overhead in switching from the user context to the kernel context. Furthermore, on a multiuser system the operating system may schedule another process to run when a system call completes, further slowing the progress in real time of a calling process.

---

8. Describe a plausible sequence of activities that occur when a timer interrupt results in a context switch.

   Note: In the table below, almost everything that is not the timer device or CPU is actually just code executing within the kernel. The distinction of "who" is there to clarify which kernel subsystem is notionally responsible.

Who	What
Timer Device	Raises interrupt line
CPU	Generates an interrupt exception
CPU	Switches from user to kernel mode
CPU	Begins executing the *Kernel Interrupt Handler*

Kernel Interrupt Handler	Changes the sp to the kernel stack for the interrupted process
Kernel Interrupt Handler	Saves the user registers for the interrupted process onto the stack
Kernel Interrupt Handler	Determines the source of the interrupt to be the timer device
Kernel Interrupt Handler	Calls the *Timer Interrupt Handler*
Timer Interrupt Handler	Acknowledges the interrupt to the timer device
Timer Interrupt Handler	Calls the *Scheduler* (dispatcher)
Scheduler	Chooses a new process to run
Scheduler	Calls the *Kernel* to switch to the new process
Kernel	Saves the current process's in-kernel context to the stack
Kernel	Switches to the new process's kernel stack by changing sp
Kernel	Reads the new process's in-kernel context off the stack
Kernel	Restores the user registers from the stack
Kernel	Sets the processor back to user mode
Kernel	Jumps to the new user process's PC

Memory Hierarchy and Caching

9. Describe the memory hierarchy. What types of memory appear in it? What are the characteristics of the memory as one moves through the hierarchy? How can do memory hierarchies provide both fast access times and large capacity?

   The memory hierarchy is a hierarchy of memory types composed such that if data is not accessible at the top of the hierarchy, lower levels of the hierarchy are accessed until the data is found, upon which a copy (usually) of the data is moved up the hierarchy for access.

   Registers, cache, main memory, magnetic disk, CDROM, tape are all types of memory that can be composed to form a memory hierarchy.

   In going from the top of the hierarchy to the bottom, the memory types feature decreasing cost per bit, increasing capacity, but also increasing access time.

   As we move down the hierarchy, data is accessed less frequently, i.e. frequently accessed data is at the top of the hierarchy. The phenomenon is called "locality" of access, most accesses are to a small subset of all data.

---

10. Given that disks can stream data quite fast (1 block in tens of microseconds), why are average access times for a block in milliseconds?

    Seek times are in milliseconds (e.g. .5 millisecond track to track, 8 millisecond inside to outside), and rotational latency (1/2 rotation) is in milliseconds (e.g. 2 milliseconds for 15,000rpm disk).

---

11. You have a choice of buying a 3 Ghz processor with 512K cache, and a 2 GHz processor (of the same type) with a 3 MB cache for the same price. Assuming memory is the same speed in both machines and is much less than 2GHz (say 400MHz). Which would you purchase and why? Hint: You should consider what applications you expect to run on the machine.

If you are only running an small application (or a large one, that accesses only a small subset), then the 3GHz processor will be much faster. If you are running a large application access a larger amount of memory than 512K but generally less than 3MB, the 2GHz processor should be faster as the 3 GHz processor will be limited by memory speed.

Files and file systems

12. What permissions would you have on the following files:

```
om:[/tmp]% ls -ld t* .
drwxrwxrwt    6 root     root         4096 May 21 12:19 .
-rw-rw----    1 nash     stud          216 May 18 18:59 t1
-rw--w----    1 nash     stud          260 May 18 18:59 t2
-rw-------    1 nash     stud          458 May 18 18:59 t3
-rwsrwsr-x    1 nash     stud          138 May 21 12:19 t4
-rwsrwxr-x    1 nash     stud          285 May 21 12:19 t5
```

The following are the 'interesting' things to note about each entry in the listing.

- '.': Everybody can create, list and access files (depending on individual file permissions) in the directory. Note the "sticky" bit 't' is set on the directory.

  From man 2 stat on linux: *When the sticky bit is set on a directory, files in that directory may be unlinked or renamed only by root or their owner. Without the sticky bit, anyone able to write to the directory can delete or rename files. The sticky bit is commonly found on directories, such as /tmp, that are world-writable.*

- t1: nash and any member of the group stud can read and write this file.
- t2: nash can read/write, group stud can only write.
- t3: Only nash can read/write the file.
- t4: Everybody can read and execute the file, only nash and group stud can modify it. Both the setuid and setgid bits ('s') are set. The executable t4 when executed while change the effective user id to nash and the effective group id to stud for the duration of execution.
- t5: As above, except the effect group id will be unchanged and default to the executor's default group.

Page last modified: 11:02am on Monday, 18th of March, 2019

Screen Version