# COMP3231/9201/3891/9283 Operating Systems 2021/T1

UNSW

## Tutorial Week 4

# Questions and Answers

## R3000 and assembly

1. What is a *branch delay*?

   The pipeline structure of the MIPS CPU means that when a jump instruction reaches the "execute" phase and a new program counter is generated, the instruction after the jump will already have been decoded. Rather than discard this potentially useful work, the architecture rules state that the instruction after a branch is always executed before the instruction at the target of the branch.

---

2. The goal of this question is to have you reverse engineer some of the C compiler function calling convention (instead of reading it from a manual). The following code contains 6 functions that take 1 to 6 integer arguments. Each function sums its arguments and returns the sum as a the result.

```
#include <stdio.h>

/* function protoypes, would normally be in header files */
int arg1(int a);
int arg2(int a, int b);
int arg3(int a, int b, int c);
int arg4(int a, int b, int c, int d);
int arg5(int a, int b, int c, int d, int e );
int arg6(int a, int b, int c, int d, int e, int f);

/* implementations */
int arg1(int a)
{
  return a;
}

int arg2(int a, int b)
{
  return a + b;
}

int arg3(int a, int b, int c)
{
  return a + b + c;
}

int arg4(int a, int b, int c, int d)
{
  return a + b + c + d;
}

int arg5(int a, int b, int c, int d, int e )
{
```

```
  return a + b + c + d + e;
}

int arg6(int a, int b, int c, int d, int e, int f)
{
  return a + b + c + d + e + f;
}

/* do nothing main, so we can compile it */
int main()
{
}
```

The following code is the disassembled code that is generated by the C compiler (with certain optimisations turned of for the sake of clarity).

```
004000f0 <arg1>:
  4000f0:       03e00008        jr      ra
  4000f4:       00801021        move    v0,a0

004000f8 <arg2>:
  4000f8:       03e00008        jr      ra
  4000fc:       00851021        addu    v0,a0,a1

00400100 <arg3>:
  400100:       00851021        addu    v0,a0,a1
  400104:       03e00008        jr      ra
  400108:       00461021        addu    v0,v0,a2

0040010c <arg4>:
  40010c:       00852021        addu    a0,a0,a1
  400110:       00861021        addu    v0,a0,a2
  400114:       03e00008        jr      ra
  400118:       00471021        addu    v0,v0,a3

0040011c <arg5>:
  40011c:       00852021        addu    a0,a0,a1
  400120:       00863021        addu    a2,a0,a2
  400124:       00c73821        addu    a3,a2,a3
  400128:       8fa20010        lw      v0,16(sp)
  40012c:       03e00008        jr      ra
  400130:       00e21021        addu    v0,a3,v0

00400134 <arg6>:
  400134:       00852021        addu    a0,a0,a1
  400138:       00863021        addu    a2,a0,a2
  40013c:       00c73821        addu    a3,a2,a3
  400140:       8fa20010        lw      v0,16(sp)
  400144:       00000000        nop
  400148:       00e22021        addu    a0,a3,v0
  40014c:       8fa20014        lw      v0,20(sp)
  400150:       03e00008        jr      ra
  400154:       00821021        addu    v0,a0,v0

00400158 <main>:
  400158:       03e00008        jr      ra
  40015c:       00001021        move    v0,zero
```

a. `arg1` (and functions in general) returns its return value in what register?
b. Why is there no stack references in `arg2`?
c. What does `jr ra` do?
d. Which register contains the first argument to the function?
e. Why is the `move` instruction in `arg1` after the `jr` instruction.
f. Why does `arg5` and `arg6` reference the stack?

a. `v0`
b. There are no local variables inside the function, so the compiler does not need space on the stack to store them.
c. It jumps (changes the program counter) to the address in the `ra` register. The `ra` register is set by a `jal` instruction to the address of the instruction after `jal`. Thus function calls can be implemented with `jal` and `jr ra` instructions.
d. `a0`
e. The instruction after a jump (i.e. the instruction in the *branch delay slot* is executed prior to arriving at the destination of the jump. Thus, logically, the `move` instruction is executed before `arg1` returns.
f. Up to 4 arguments can be passed to a function in registers. Arguments beyond the fourth are passed on the stack?

---

3. The following code provides an example to illustrate stack management by the C compiler. Firstly, examine the C code in the provided example to understand how the recursive function works.

```
#include <stdio.h>
#include <unistd.h>

char teststr[] = "\nThe quick brown fox jumps of the lazy dog.\n";

void reverse_print(char *s)
{
  if (*s != '\0') {
    reverse_print(s+1);
    write(STDOUT_FILENO,s,1);
  }
}

 int main()
{
  reverse_print(teststr);
}
```

The following code is the disassembled code that is generated by the C compiler (with certain optimisations turned off for the sake of clarity).

a. Describe what each line in the code is doing.
b. What is the maximum depth the stack can grow to when this function is called?

```
004000f0 <reverse_print>:
  4000f0:       27bdffe8        addiu   sp,sp,-24
  4000f4:       afbf0014        sw      ra,20(sp)
  4000f8:       afb00010        sw      s0,16(sp)
  4000fc:       80820000        lb      v0,0(a0)
  400100:       00000000        nop
  400104:       10400007        beqz    v0,400124 <reverse_print+0x34>
  400108:       00808021        move    s0,a0
  40010c:       0c10003c        jal     4000f0 <reverse_print>
  400110:       24840001        addiu   a0,a0,1
  400114:       24040001        li      a0,1
  400118:       02002821        move    a1,s0
  40011c:       0c1000af        jal     4002bc <write>
  400120:       24060001        li      a2,1
  400124:       8fbf0014        lw      ra,20(sp)
  400128:       8fb00010        lw      s0,16(sp)
  40012c:       03e00008        jr      ra
  400130:       27bd0018        addiu   sp,sp,24
```

a.

```
004000f0 <reverse_print>:
  4000f0:       27bdffe8        addiu   sp,sp,-24
  Allocate 24 bytes on the stack, 16 for a0-a3 (unused) and 8 for ra
  and s0

  4000f4:       afbf0014        sw      ra,20(sp)
  Save the return address for the function on the stack. This
  function calls other functions, which means the ra register will be
  overwritten.

  4000f8:       afb00010        sw      s0,16(sp)
  Recall the 's' registers must be preserved when we return from this
  function. We only use s0, so save it on the stack so we can use the
  register in this function, but restore it before returning.

  4000fc:       80820000        lb      v0,0(a0)
  Load a character from the pointer passed as the first argument.

  400100:       00000000        nop
  nop

  400104:       10400007        beqz    v0,400124 <reverse_print+0x34>
  Test is the character is zero, if so, jump forward to 400124

  400108:       00808021        move    s0,a0
  This is on the delay slot, save the pointer in s0

  40010c:       0c10003c        jal     4000f0 <reverse_print>
  Call reverse print

  400110:       24840001        addiu   a0,a0,1
  This is in the delay slot, add 1 to the pointer to have reverse
  print start on the next character in the string.

  400114:       24040001        li      a0,1
  Load the file descriptor for write (1).

  400118:       02002821        move    a1,s0
  Remember s0 is preserved across function calls above, so s0 still
  contains the original pointer passed into the function. Pass the
  pointer to write.

  40011c:       0c1000af        jal     4002bc <write>
  Call write function

  400120:       24060001        li      a2,1
  Another delay slot, load the number of bytes write should output (1
byte).

  400124:       8fbf0014        lw      ra,20(sp)
  Restore the return address of this function in prep for return from
  function

  400128:       8fb00010        lw      s0,16(sp)
  Restore s0 to whatever it was before this function was called.

  40012c:       03e00008        jr      ra
  Return to the caller.

  400130:       27bd0018        addiu   sp,sp,24
  In the branch delay slot, deallocate the stack.
```

b. The stack of each invocation of `reverse_print` is 24 bytes, but the function is recursive. The allocation is 24 bytes times the length of the string, and thus if the string is unbounded, so is the recursion, and thus stack growth is also unbounded.

---

4. Why is recursion or large arrays of local variables avoided by kernel programmers?

The kernel stack is usually a limited resource. A stack overflow crashes the entire machine.

---

## Threads

5. Compare cooperative versus preemptive multithreading?

   Cooperative multithreading is where the running thread must explicitly yield() the CPU so that the dispatcher can select a ready thread to run next. Preemptive multithreading is where an external event (e.g. a regular timer interrupt) causes the dispatcher to be invoked and thus *preempt* the running thread, and select a ready thread to run next.

   Cooperative multithreading relies on the cooperation of the threads to ensure each thread receives regular CPU time. Preemptive multithreading enforces a regular (at least systematic) allocation of CPU time to each thread, even when a thread is uncooperative or malicious.

---

6. Describe *user-level threads* and *kernel-level threads*. What are the advantages or disadvantages of each approach?

   User-level threads are implemented in the application (usually in a "thread library"). The thread management structures (Thread Control Blocks) and scheduler are contained withing the application. The kernel has no knowledge of the user-level threads.

   Kernel threads are implemented in the kernel. The TCBs are managed by the kernel, the thread scheduler is the normal in-kernel scheduler.

   - User threads are generally faster to create, destroy, manage, block and activate (no kernel entry and exit required).
   - If a single user-level thread blocks in the kernel, the whole process is blocked. However, some libraries (e.g., most UNIX pthreads libraries) avoid blocking in the kernel by using non-blocking system call variants to emulate the blocking calls.
   - User threads don't take advantage of parallelism available on multi-CPU machines.
   - Kernel threads are usually preemptive, user-level threads are usually cooperative (Note: some user-level threads use alarms or timeouts to provide a tick for preemption).
   - User-level threads can be implemented on OSes without support for kernel threads.

---

7. A web server is constructed such that it is multithreaded. If the only way to read from a file is a normal blocking `read` system call, do you think user-level threads or kernel-level threads are being used for the web server? Why?

   A worker thread within the web server will block when it has to read a Web page from the disk. If user-level threads are being used, this action will block the entire process, destroying the value of multithreading. Thus it is essential that kernel threads are used to permit some threads to block without affecting the others.

---

8. Assume a multi-process operating system with single-threaded applications. The OS manages the concurrent application requests by having a *thread* of control within the kernel for each process. Such a OS would have an in-kernel stack assocaited with each process.

   Switching between each process (in-kernel thread) is performed by the function `switch_thread(cur_tcb,dst_tcb)`. What does this function do?

The function saves the registers required to preserve the compiler calling convention (and registers to return to the caller) onto the current stack.

The function saves the resulting stack pointer into the thread control block associated with `cur_tcb`, and sets the stack pointer to the stack pointer stored in destination tcb.

The function then restores the registers that were stored previously on the destination (now current) stack, and returns to the destination thread to continue where is left off.

---

## Kernel Entry and Exit

9. 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.

---

10. 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).

---

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

    The value of `ExcCode` is 8.

---

12. 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.

---

13. 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.

---

14. 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.

---

15. 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.

---

16. 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 |

---

*Page last modified: 9:13am on Tuesday, 9th of February, 2021*

Screen Version

CRICOS Provider Number: 00098G