Tutorial Week 4

Questions

It is the pipeline structure of MIPS CPU meaning that when a jump instruction is performed and a new program counter is generated, the instruction after the jump will be decoded, rather than discarded, the architecture rules state that the instruction after a branch is always executed before the instruction at the target of the branch

R3000 and assembly

1. What is a *branch delay*?

---

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
```

because there is no local variable inside the function, so the compiler does not need space on the stack to store them

a. `arg1` (and functions in general) returns its return value in what register? v0
b. Why is there no stack references in `arg2`?
c. What does `jr ra` do? It jumps to the address in the ra register
d. Which register contains the first argument to the function? a0
e. Why is the `move` instruction in `arg1` after the `jr` instruction. because move is in the branch delay, so it is executed before arg1 returns
f. Why does `arg5` and `arg6` reference the stack? because we only have a0 - a3 in our register, if we need to store more arguments, we need to use 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?

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.

```
004000f0 <reverse_print>:
  4000f0:       27bdffe8        addiu   sp,sp,-24       reverse 24 bytes for the stack, 16 for a0-a3, and 8 for ra and s0
  4000f4:       afbf0014        sw      ra,20(sp)       save the return address for the function on the stack
  4000f8:       afb00010        sw      s0,16(sp)       save s0 on the stack so we can use the register in this function,
  4000fc:       80820000        lb      v0,0(a0)        load the byte at a0 to register v0 as the first argument
  400100:       00000000        nop                     v0 won't be ready here
  400104:       10400007        beqz    v0,400124 <reverse_print+0x34>  check the value of v0 (the character) is zero, if so jump to 400124
  400108:       00808021        move    s0,a0           this is on the branch delay slot, save the pointer in s0 to a0
  40010c:       0c10003c        jal     4000f0 <reverse_print>  call reverse_print
  400110:       24840001        addiu   a0,a0,1         this is at the branch delay, so the value of a0++, move onto next char
  400114:       24040001        li      a0,1            load the file descriptor for write(1) ???
  400118:       02002821        move    a1,s0           pass the pointer in s0 to write
  40011c:       0c1000af        jal     4002bc <write>  jump and link to 4002bc, call write function
  400120:       24060001        li      a2,1            this is at the branch delay, load the number of bytes write outputs
  400124:       8fbf0014        lw      ra,20(sp)       restore the return address of this function in prep for return
  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
```

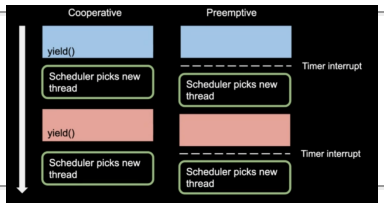
---

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

---

because we have limited size of kernel stack and it can grow very fast, we don't want stack overflow crashes the entire machine

Threads



in the cooperative case you have to manually yield, so you don't take any resources from other parts of the system.
but for preemptive case, we don't have to worry about it because the OS handles everything for us

5. Compare cooperative versus preemptive multithreading?

---

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

they run in user space, kernel level threads are managed by the kernel but not ran in the kernel

user-level threads:
can be run on an os that doesn't support kernel threads;
faster to create, destroy etc
usually cooperative, some user level threads use alarms or timeouts to provide a tick for preemption
blocking syscalls freeze up the entire process

kernel-level threads
take advantage of multi-core cpus
usually preemptive
slow and inefficient, requires kernel entry and exit, more overhead

---

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?

kernel level threads are being used because if the server is constructed using user-level threads, when someone is performing a read, other people will be blocked from the process until the user has finished reading, or syscall returns

---

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?

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 point where execution was interrupted.

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).

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

11. What is the value of `ExcCode` in the `Cause` register immediately after a system call exception occurs?
the value will be corresponding to the number of the syscall exception - 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.
v0 register contains what type of syscalls. a0-a3 are the arguments

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?
    The C function calling convention relate to the system call interface between the application and the kernel. This involves 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.
    2. What does the most work to preserve the compiler calling convention, the system call wrapper, or the OS/161 kernel.
    The OS/161 kernel code does the saving and restoring of preserved registers. The system call wrapper function does very little.
    3. At minimum, what additional information is required beyond that passed to the system-call wrapper function?
    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 don't really have names, the name is there just for us to understand what system calls are invoked. The library function read() traps in to the kernel, it puts the number of tye system call into a register or on the stack. This number is used to index into a jump table. There is really no name used anywhere. On the other hand, the name of the library function is very important as it is what appears in the program

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?
in performance critical, syscalls are not ideal because we need context switch between user space and kernel space, which is a performance overhead
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.