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

UNSW

Tutorial Week 4

Questions

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;
```

Staff
- Kevin Elphinstone (LiC)
- TBD (Admin)

Grievances
- Student Reps


```
}

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?

---

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
```

---

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

Threads

5. Compare cooperative versus preemptive multithreading?

---

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

---

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?

---

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?

---

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?

---

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

---

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?

---

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?

---

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?

---

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

Page last modified: 4:49pm on Saturday, 23rd of May, 2020

Print Version