**CS2106 Introduction to Operating Systems**
**Tutorial 1**

## Section 1. MIPS Assembly Revision

In this section we will revise MIPS assembly programming. You may assume the following:

i. Execution always begins at the label "main:"
ii. When a program ends, it hands control back to the operating system using the following two instructions (similar to what was done in the CS2100 labs):

```
li $v0, 10
syscall
```

iii. The availability of pseudo instructions like load immediate (li), load address (la) and move (mov).

Example code:

```
b = a + 10;

main:  la $t0, a
       lw $t1, 0($t0)
       addi $t1, $t1, 10;
       la $t0, b
       sw $t1, 0($t0)
       li $v0, 10
       syscall
```

Function calls in MIPS are performed using jal and jr. We consider an example where we call a function "func" to add 10 to $t0, using a "jump-and-link" or jal instruction. The address of the instruction immediately after the jal (in this case li $v0, 10) is placed into $ra, so that within the function we can just do jr $ra to return to the instruction just after the jal:

| | Address | Instruction | Comments |
|---|---|---|---|
| main: | 0x1000 | addi $t0, $zero, 5 | |
| | 0x1004 | jal func | ; Jump-and-link to func. Address 0x1008 put ; into $ra |
| | 0x1008 | li $v0, 10 | ; Exit to OS |
| | 0x100C | syscall | |
| func: | 0x1010 | addi $t0, $t0, 10 | |
| | 0x1014 | jr $ra | ; Jump to 0x1008 to exit function |

Question 1

Write the following C program in MIPS assembly. We will explore passing parameters using registers instead of stack frames.

Use $a0 and $a1 to pass parameters to the function f, and $v0 to pass results back. You will need to use the MIPS jal and jr instructions. All variables are initially in memory, and you can use the la pseudo instruction to load the address of a variable into a register.

```
int f(int x, y) {
        return 2 * (x + y);
}

int a = 3, b = 4, y;

int main() {
         y = f(a, b);
}
```

Question 2

In this question we explore how the stack and frame pointers on MIPS work. The MIPS stack pointer is called $sp (register $29) while the frame pointer is called $fp (register $30). Unlike many other processors like those made by ARM and Intel, the MIPS processor does not have "push" and "pop" instructions. Instead we manipulate $sp directly:

"Pushing" a value in $r0 onto the stack.

```
sw $r0, 0($sp)
addi $sp, $sp, 4
```

"Popping" a value from the stack to $s0:

```
addi $sp, $sp, -4
lw $s0, 0($sp)
```

Repeat Question 1 above using the stack to pass arguments and results instead of $a0, $a1 and $v0. You may find it easier if you save $sp to $fp, then used $fp to access the arguments.

**Section 2. Using Stack Frames**

Question 3

Can your approach to passing parameters and calling functions in Questions 1 and 2 above work for recursive or even nested function calls? Explain why or why not.

Question 4

We now explore making use of a proper stack frame to implement our function call from Question 1. Our stack frame looks like this when calling a function:

| Saved registers |
|---|
| $ra |
| parameter n |
| … |
| parameter 2 |
| parameter 1 |
| Saved $sp |
| Saved $fp |

We follow this convention (callee = function being called). Assume that initially $sp is pointing to the bottom of the stack.

| | |
|---|---|
| Caller: | 1. Push $fp and $sp to the stack. |
| | 2. Copy $sp to $fp |
| | 3. Reserve sufficient space on stack for parameters by adding appropriately to $sp |
| | 4. Write parameters to the stack using offsets from $fp |
| | 5. jal to callee |
| Callee: | 1. Push $ra to stack |
| | 2. Allocate enough space for local variables. |
| | 3. Push registers we intend to use onto the stack. |
| | 4. Use $fp to access parameters. |
| | 5. Compute result |
| | 6. Write result to to the stack. |
| | 7. Restore registers we saved from the stack |
| | 8. Get $ra from stack. |
| | 9. Return to caller by doing jr $ra |
| Caller: | 1. Get result from stack. |
| | 2. Restore $sp and$fp |

Rewrite Question 1 using this calling convention. Recall that in MIPS architectures an integer occupies 4 bytes.

Question 5

In Question 4 the callee saved registers it intends to use onto the stack and restores them after that. What would happen if the callee does not do that? Why don't we do the same thing for main?

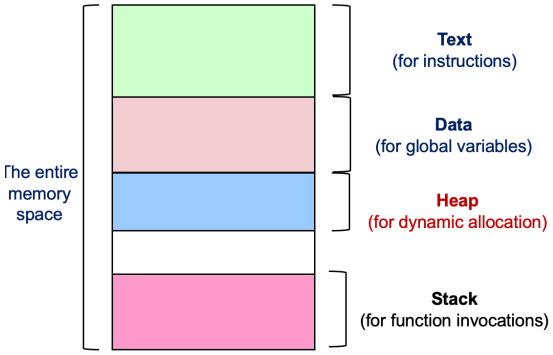
Question 6

Explain why, in step 7 of the callee, we retrieve $ra from the stack before doing jr $ra. Why can't we just do jr $ra directly?

**Section 3. Process Memory**

Question 7

The diagram below shows the memory allocated to a typical process:




We have the following program:

```
int fun1(int x, int y) {
        int z = x + y;
        return 2 * (z – 3);
}

int c;

int main() {
   int *a = NULL, b = 5;
   a = (int *) malloc(sizeof(int));
   *a = 3;
   c = fun1(*a, b);
}
```

Indicate in which part of a process is each of the following stored or created (Text, Data, Heap or Stack):

| Item | Where it is stored/created |
|---|---|
| a | |
| *a | |
| b | |
| c | |
| x | |
| y | |
| z | |
| fun1's return result | |
| main's code | |
| Code for f | |