CS2106 Introduction to Operating Systems
25/26 Semester 1
Tutorial 1

SUGGESTED SOLUTIONS

Section 1. MIPS Assembly Revision

NOTE TO TAs: This is a long tutorial so try not to spend too much time on Questions 1 and 2.

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);
        }
```

```
f:      add $t1, $a0, $a1       ; x + y
        sll $v0, $t1, 1         ; 2 * (x + y)
        jr $ra                  ; Return to caller

main:   la $t0, a               ; Load a
        lw $a0, 0($t0)          ;
        la $t0, b               ; Load b
        lw $a1, 0($t0)          ;
        jal f                   ; Call f. Note this does a call by value
        la $t0, y               ; Store results
        sw $v0, 0($t0)          ;
```

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.

```
f:      lw $t0, 0($fp)          ; Get first parameter
        lw $t1, 4($fp)          ; Get second parameter
        add $v0, $t0, $t1       ; $v0 = $t1 + $t2
        sll $v0, $v0, 1         ; $v0 = 2($t1 + $t2)
        sw $v0, 0($fp)          ; Store result
        jr $ra                  ; Return to caller

main:   addi $fp, $sp, 0                ; Save $sp. Note you can also do mov $fp, $sp
        addi $sp, $sp, 8                ; Reserve 2 integers on the stack for stack frame
        la $t0, a                       ; Load a
        lw $t0, 0($t0)                  ;
        sw $t0, 0($fp)                  ; Write to stack frame
        la $t0, b                       ; Load b
        lw $t0, 0($t0)
        sw $t0, 4($fp)                  ; Write to stack frame
        jal f                           ; call f
        lw $t1, 0($fp)                  ; Get result from stack frame
        la $t0, y                       ; Store into y
        sw $t1, 0($t0)
        addi $sp, $sp, -8               ; Pop off stack frame
        li $v0, 10                      ; Exit to OS
        syscall
```

Note to TA: Notice how we increment $sp by 8 to make space for the two arguments, then store using offsets from $fp, rather than doing this (which students are likely to do).

```
sw $t0, 0($sp)
addi $sp, $sp, 4
sw $t0, 0($sp)
addi $sp, $sp, 4
```

“Reserving” space on the stack by incrementing $sp by 8 first, then “pushing” to the stack using offsets from $fp is more efficient as it saves on the number of adds to $sp.

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.

Note to TA: Some students may store $ra on the stack and correctly retrieve it before doing jr $ra, in which case it's likely their solution will support nesting and recursion.

The solution as provided above does not support nesting as it does not save $ra. If f calls g using jal for example, $ra would be overwritten by the instruction after the jal instruction in f. In addition old values of $fp and $sp are not saved, and will be overwritten by function calls.

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. Push registers we intend to use onto the stack.
	3. Use $fp to access parameters.
	4. Compute result
	5. Write result to the stack.
	6. Restore registers we saved from the stack
	7. Get $ra from stack.
	8. 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.

This question requires some planning:

The caller needs to reserve:

- 8 bytes to save $sp and $fp
- 8 bytes to pass a and b
- 4 bytes for callee to save $ra

Thus the caller must save 20 bytes and gives us the following map:

Offset from $fp	Contents
0	$fp
4	$sp
8	a
12	b
16	$ra

```
f:      sw $ra, 16($fp)         ; Save $ra
        addi $sp, $sp, 8        ; Reserve 8 bytes on stack to store registers
                                ; we want to use
        sw $t0, 20($fp)         ; Save $t0 and $t1 which we intend to use
        sw $t1, 24($fp)
        lw $t0, 8($fp)          ; Load value of a
        lw $t1, 12($fp)         ; Load value of b
        addi $t1, $t0, $t1      ; a + b
        sll $t1, $t1, 1         ; 2(a + b)
        sw $t1, 8($fp)          ; Write result back to stack frame
        lw $t0, 20($fp)         ; Restore $t0 and $t1
        lw $t1, 24($fp)
        addi $sp, $sp, -8       ; Deallocate space on stack for $t0, $t1
        lw $ra, 16($fp)         ; Restore $ra
        jr $ra                  ; Return to caller

main:   sw $fp, 0($sp)          ; Save $fp
        mov $fp, $sp            ; Copy $sp to $fp. Can also use addi $fp, $sp, 0
        sw $sp, 4($sp)          ; Save $sp to stack frame
        addi $sp, $sp, 20       ; Reserve 20 bytes for stack frame
        la $t0, a               ; Load a
        lw $t0, 0($t0)
        sw $t0, 8($fp)          ; Write a to stack
        la $t0, b               ; Load b
        lw $t0, 0($t0)
        sw $t0, 12($fp)         ; Write to stack
        jal f                   ; Call f
        lw $t0, 8($fp)          ; Retrieve result
        la $t1, y               ; Store to y
```

```
sw $t0, 0($t1)
lw $sp, 4($fp)          ; Restore $sp. We don’t need to explicitly subtract 20 from sp.
lw $fp, 0($fp)          ; Restore $fp
li $v0, 10              ; Return to OS
syscall
```

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?

The callee does not know what registers the caller is using, and thus may accidentally change the contents of a register that the caller was using, resulting in incorrect execution. By saving and restoring registers it intends to use, it prevents this error from happening.

We can do this for main, but it’s not necessarily since main is likely to be invoked by the OS, and the OS would have saved registers it (or other processes) needed during context switching.

Question 6

Explain why, in step 8 of the callee, we retrieve $ra from the stack before doing jr $ra. Why can’t we just do jr $ra directly?

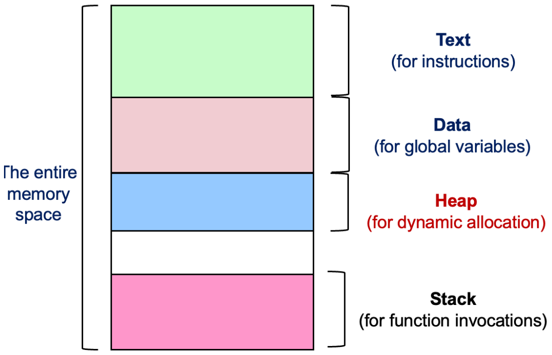
This step lets us support nesting/recursion by explicitly saving the $ra for f, in case we call another function (e.g. g) which would overwrite $ra. Directly doing jr $ra would then result in incorrect execution,.

In this case by using the saved value of $ra we are guaranteed a return to the correct caller.

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	Stack
*a	Heap
b	Stack
c	Data memory
x	Stack
y	Stack
z	Stack
fun1's return result	Stack
main's code	Text
Code for f	Text