CS2106 Introduction to Operating Systems

Semester 1 2021/2022

Solutions

Week 3 (23-29 August 2021)

Tutorial 1: **Process Abstraction**

1. [Function invocation – the gory details] Let's use a "simple" function to really understand the idea of stack frames and calling conventions. Note that the stack frame layout in this question is slightly different from the one covered in lecture 2. So, ensure you have a good understanding of the basics before attempting this question.

Given below is an **iterative** factorial function in C.

C

```c
int iFact( int N )
{
	int result = 1, i;

	for (i = 2; i <= N; i++){
		result = result * i;
	}

	return result;
}
```

a. [Code translation] Take a look at the partial assembly code translation on pages 4 & 5. You should find most of it (vaguely) familiar from your basic assembly programming course. The remaining missing pieces are all related to the function call (setup/tear down of stack frame).

Suppose the following stack frame is used on this platform. For simplicity, all integers and registers are assumed to occupy 4 bytes, and the stack region is "growing" towards lower address.

Unused Stack Memory Space			-- Stack Pointer ($**sp**)
Local Variable	-36	[result]	
	-32	[i]	
Parameter	-28	[N]	
Return Result	-24		
Saved Registers	-20	[$11]	
	-16	[$12]	
	-12	[$13]	-- Frame Pointer ($**fp**)
Saved SP	-8		
Saved FP	-4		
Saved PC	0		

Complete the memory offsets in the **`lw`**/**`sw`** instructions (they are tagged with "Part a"). You can assume that the **`$sp`** and **`$fp`** registers are initialized properly.

b. [Stack frame – caller prepares to call a function] Refer to the calling convention sample given in lecture 2. Assume we make the following function call **`iterativeFactorial( 10 )`** from the **`main()`** function.

Essentially, you need to:

- Pass the parameter ("10") onto the stack.
- Save the PC to return to on the stack. For simplicity, we assume there is a "`call` ***`function offset(register)`***" instruction. This instruction **saves the next PC** to the memory address specified by "`offset(register)`", then jumps to the specified "function".

Complete the relevant portions tagged with "Part b".

c. [Stack frame – callee enters function] Now, let us fill in the instructions for the callee to setup the stack frame upon entering the function. Tasks required:

- Save the registers used in the caller (i.e. $11-$13).
- Save the current FP, SP registers on the stack.
- Allocate space for local variables, i.e. "Result", "i".
- Adjust the special registers FP, SP.

d. [Stack frame – callee exits function] At the end of the function, we need to:

- Place the return result onto stack frame.
- Restores the saved registers, FP, and SP.
- Return to the caller with the saved PC. For simplicity, we assume there is a "**`return offset(register)`**" instruction, which overwrites the PC with the value stored at the memory location "**`offset(register)`**".

With (d), we now have a complete demonstration of the idea of stack frames, calling conventions and the usage of stack / frame pointers.

e. [Extra challenge – not discussed] Draft a solution for a **recursive version of** factorial. The bits and pieces from (a), (c) and (d) are very similar, the only tricky part is that the recursive factorial function is both a caller and a callee… So, figuring out where (b) should be placed is the main challenge.

Notes: To illustrate a generic calling convention, we have to provide several made-up instructions, e.g. **`call`** and **`return`**. Feel free to explore other real calling conventions on different platforms, e.g. Intel x86 (as shown in Q1), MIPS, etc, for different languages, e.g. C/C++, Python and Java on JVM. You should be able to see the same ideas echoed across different environments.

MIPS-like Assembly Code

```
iFact:
      #Part (c) - Callee enter function
                              #save registers

                              #save $fp, $sp

                              #move $fp, $sp to
                              #     new position
      #Part (c) - Callee enter function ends

      addi $11, $0, 1         #init "result"
      sw   $11, ____($fp)     ##Part (a) result = 1

      addi $12, $0, 2         #init "i"
      sw   $12, ____($fp)     ##Part (a) i = 2

      lw   $13, ____($fp)     ##Part (a) Get N
loop: bgt  $12, $13, end

      mul  $11, $11, $12      #assume no overflow
      sw   $11, ___($fp)      ##Part (a) update result

      addi $12, $12, 1
      sw   $12, ___($fp)      ##Part (a) i++
      j loop

end:
      #Part (d) - Callee exit function
                              #save return result

                              #restore registers


                              #restore $sp, $fp



      return         #resume execution of the caller


 #Continues on the next page
```

```
### Main Function
main:

    ......                    #irrelevant code omitted

    #Part (b) - Caller prepare to call function
    addi  $13, $0, 10         #Use $13 to store 10
    sw                        #Where should the "10" go?
    call iFact,            #start executing the function
```

ANS:

MIPS-like Assembly Code

```
iFact:
        #Part (c) - Callee enter function
        sw   $11, -20($sp)          #save registers
        sw   $12, -16($sp)
        sw   $13, -12($sp)

        sw   $sp, -8($sp)           #save $fp, $sp
        sw   $fp, -4($sp)

        addi $sp, $sp, -40         #move $fp, $sp to
        addi $fp, $sp, 28          #      new position
        #Part (c) - Callee enter function ends

        addi $11, $0, 1           #init "result"
        sw   $11, -24($fp)        ##Part (a) result = 1

        addi $12, $0, 2           #init "i"
        sw   $12, -20($fp)           ##Part (a) i = 2
        lw   $13, -16($fp)           ##Part (a) N
loop: bgt  $12, $13, end

        mul  $11, $11, $12        #assume no overflow
        sw   $11, -24($fp)           ##Part (a) result = 1

        addi $12, $12, 1
        sw   $12, -20($fp)           ##Part (a) i++
        j loop
end:
        #Part (d) - Callee exit function
        sw   $11, -12($fp)          #save return result
```

```
        lw   $11, -8($fp)          #restore registers
        lw   $12, -4($fp)
        lw   $13, 0($fp)

        lw   $sp, 4($fp)           #restore $sp, $fp
        lw   $fp, 8($fp)

        return 0($sp)    #resume execution of the caller
        #Part (d) - Callee exit function

  ### Main Function
  main:
      ......                    #irrelevant code omitted
      #Part (b) - Caller prepare to call function

      addi  $13, $0, 10       #Use $13 to store 10
      sw    $13, -28($sp)     #Where should the "10" go?
      call iFact, 0($sp)   #start executing the function
```

2. [Functions and stack – AY1819S1 Midterm] In many programming languages, function parameter can be **passed by reference**. Consider this fictional C-like language example:

```
void change( int<Ref> i ) { //i is a pass-by-reference parameter
   i = 1234;   //this changes main's variable myInt in this case
}

int main() {
   int myInt = 0;
   change( myInt );     //myInt become 1234 after the function call
   ......  //other variable declarations and code
}
```

Mr. Holdabeer feels that he has the perfect solution **that works for this example** by relying on **stack pointer and frame pointer.** The key idea is to load main's local variable "`myInt`" whenever the variable "`i`" is used in the change() function.

Given that the stack frame arrangement shown **independently** as follows:

For `Main()`

		← **$SP**
…	…	
myInt	-12	
Saved SP	-8	← **$FP**
Saved FP	-4	
Saved PC	0	

For `change()`

		← **$SP**
Saved SP	-8	← **$FP**
Saved FP	-4	
Saved PC	0	

a. Suppose the `main()`'s and change()'s stack frame has been properly setup, and `change()` is now executing, show how to store the value "1234" into the right location. You only need pseudo-instructions like below. **[3 marks]**

- `Register_D ← Load Offset( Register_S )`
  Load the value at memory location `[Register_S] + Offset` and put into `Register_D`
  e.g. `$R1 ← Load -4($FP)`
- `Offset(Register_S) ← Store Value`
  Put the `value` into memory location `[Register_S] + Offset`,
  e.g. `-4($FP) ← Store 1234`

b. Briefly describe another usage scenario for pass-by-reference parameter that **will not work with** this approach.

c. Briefly describe a better, universal approach to handle pass-by-reference parameter on stack frame. Sketch the stack frame for the `change()` function to illustrate your idea.

ANS:

```
$R1 ← Load 4($FP)  //get saved FP, i.e. main's FP
-4($R1) ← 1234     //don’t forget the offset
```

Rationale: In this case, FP is the correct use as we are not sure about the offset from SP. Secondly, you need to access the main()'s FP via the "saved FP" location. Note that we are not particular about the syntax, you can even write: 4(-4($FP)) ← 1234 as answer and still get full marks.

Common mistakes:

```
Use "saved FP" directly
No offset
Use FP/SP as the intermediates, e.g.
$SP←load 4($FP) ….
```

a. If the reference is passed to another function as reference, then the scheme breaks down. (As saved FP only point back to the caller)

Rationale: As (a) only works with direct caller, it'll break down once you go beyond that. Common mistakes:

- Say something like "use change() repeatedly", which is actually fine under this scheme. Need to mention "use change() in a chain" or "if change() is a recursive function" etc.
- Use non-local variable as argument, e.g. heap data. This is not correct as heap data are pointed by a _local variable_ (a pointer), if you pass that pointer into change(), then this scheme still work.
- Use global variable as argument. Since global variable access will be compiled differently (they are accessed via direct address as laid out by compiler). So, this scheme is not even applicable.

b. Place the actual address of the value. Change the instruction to load/store from that address

For Main()			For change()		
		← **$SP**			← **$SP**
…	…		address of myInt	-12	
myInt	-12				
Saved SP	-8	← **$FP**	Saved SP	-8	← **$FP**
Saved FP	-4		Saved FP	-4	
Saved PC	0		Saved PC	0	

3. [Process state] Suppose we compile and execute the following C code. Using the 5-state process model, trace the corresponding process state transitions for the executable ("**a.exe**") produced from the code below.

```
int main()
{
Before main() even runs: New -> Ready -> Running
    int input, result;

    // will either cause a state transition?
    printf("Give input below:\n");
Depends on buffering and speed of the consumer on the
other side of the stdout pipe.
Likely: Running -> Blocked -> Ready -> Running

    scanf("%d", &input);
Running -> Blocked -> [user input] -> Ready -> Running

    // takes a long time - math operations only
    result = ComplexFunc( input );
Might take multiple time quanta (though students will
not have learnt this yet). Possibly many iterations
of Running -> Ready -> Running -> Ready ->...

    // write result to disk
    saveToDisk( result );
Running -> Blocked -> Ready -> Running, since the
disk is also an IO resource that is very likely to
block. Advanced students might argue about non-
blocking I/O, in which case this may not even cause
a state transition.

    printf("Result is %d\n", result);
Same argument as earlier printf.

    return 0;
}
```

Discussion question if time permits:

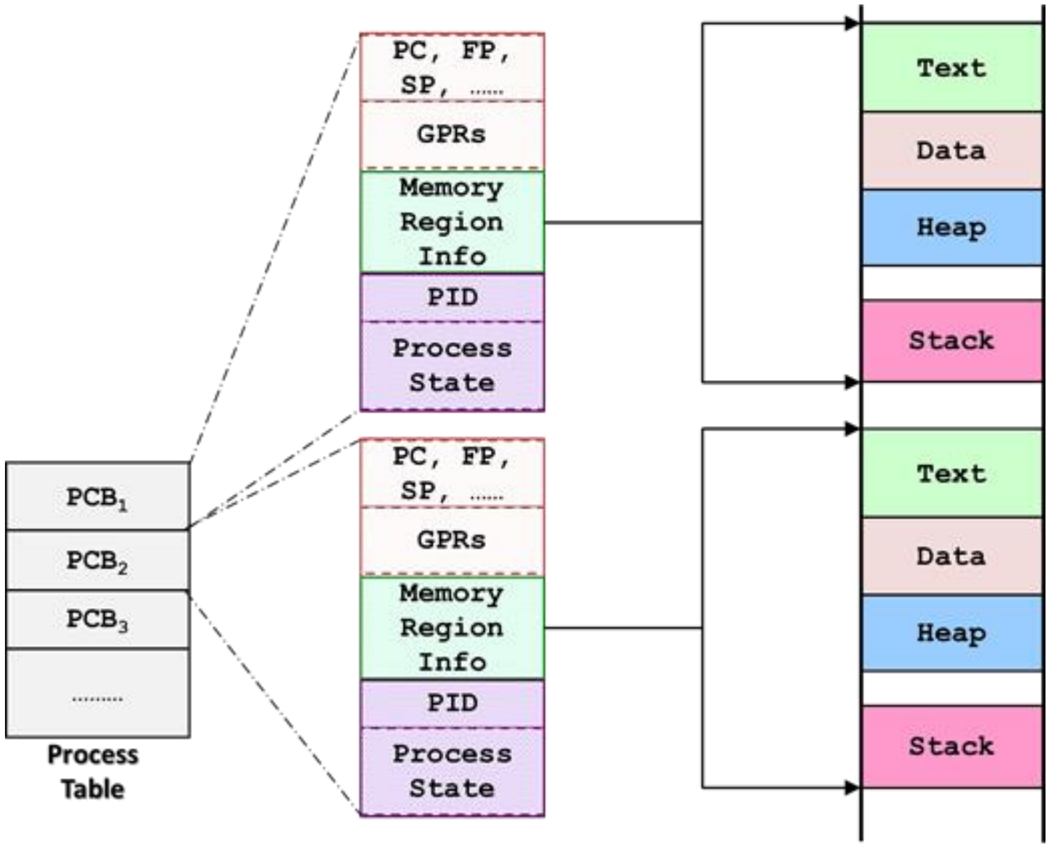
4. [Process control block] Question 1 focuses on the use of **stack memory** within a single program. Let us take a step back and look at **multiple executing programs.** Suppose we execute the program containing the factorial function **twice** and both processes run in parallel (i.e. exist at the same time). Draw the PCBs of the two processes (similar to lecture 2's slide on PCBs), indicate clearly what is laid out in the physical memory and how the different memory regions of a process fit together.

ANS:

Sample illustration given. PCB_1 and PCB_2 represents two independent processes running the same executable.

Key points to highlight / discuss:

- Two distinct regions of memory are used by the processes.
  - Point out opportunity for saving (e.g. can code be shared?)
  - Point out why distinct region is needed for other (e.g. why couldn't we share data region?)
- Point out how we switch between the two processes. (high level understanding will do, e.g. point out the registers are stored in PCB, which can be used to restore the context among other info).
- Can illustrate the progression of the two processes by utilizing the stack region (e.g. show one is executing fac(2), while the other is executing fac(5))



Additional exercise question if time permits:

5. [MIPS and stack frames – a refresher] A snapshot of the current state of a program's stack (along with the current locations of the stack and frame pointer) is given below. In this example, the second column refers to the difference in memory address from the Saved PC, and the third column refers to the value currently at that position. For simplicity, all integers and registers are assumed to occupy 4 bytes, and the stack region is "growing" towards lower addresses.

From this state, we execute a few MIPS-like instructions. Assume we have a function named **`print`** which prints the value in register **`$1.`** We can call it with: **`call print.`** **What would be printed as a result of running each set of instructions?** If there is no clear answer, state precisely why.

Also, recall that the second parameter of the **`lw`**/**`sw`** instruction has the form of **`offset(register)`**, e.g.
`-12($fp)` == (**`content of register $fp) - 12`**.

(Each sub-part of this question is independent, e.g. b) behaves like a) never happened, and so on)

Unused Stack Memory Space			-- Stack Pointer ($**sp**)
Local Variable	-36	22	
	-32	8	
Parameter	-28	16	
Return Result	-24	7	
Saved Registers	-20	9	
	-16	1	
	-12	3	-- Frame Pointer ($**fp**)
Saved SP	-8	*not relevant*	
Saved FP	-4	*not relevant*	
Saved PC	0	*not relevant*	

a.

MIPS-like Assembly Code
```
        lw   $1, 0($sp)
        call print Not clear: 0($sp) is unused space
```

b.

MIPS-like Assembly Code
```
        lw   $2, -16($fp)  $2 = 16
        add  $sp, $sp, $2  $sp = $sp + 16 (points to value 7)
        lw   $1, 0($sp)    $1 = 7
        call print         prints the number 7
```

c.

MIPS-like Assembly Code
```
        lw   $sp, -16($fp)  $sp = 16 <- some random address 16
        lw   $1, 0($sp)     $1 = unknown value at addr 16
        call print          Not clear: what is at addr 16?
```