CS2106 Operating Systems

Semester 2 2020/2021

Week of 25th January 2021

Tutorial 1 Suggested Solution

Process Abstraction

1. [Memory Layout] Consider the following code fragment:

C

```
int x;
int *p;
int f = 0;
int y = 0;

int main()
{
   int g=7;

   p = malloc(100);
   f = 6;
   sub(f, g);
   f=8;

   return 0;
}

void sub(int a, int b)
{
  int c;

  c = a;
  a = x;
  b = y;
  x = y;
  *p = b;
}
```

For each variable, indicate 1) where it will be located in the memory, 2) what is the scope of the variable, and 3) what is the lifetime of the variable, 4) what component is responsible for its allocation.

ANS:

- **x**, pointer **p**, **f** and **y** are in the **data segment**. They are global variables accessible from the entire program. Their lifetime is from the beginning until the end of the program execution. These variables are allocated by the compiler and they are part of the executable.
- **g** and **c** are local variables, allocated on the stack by the compiler. Their scope is within the function where they were declared. Their lifetime is from the beginning until the end of the function execution.
- The array of 100 elements is dynamically allocated on the heap. The scope is global, and the lifetime is from the moment of allocation (with malloc) until the moment of explicit deallocation (free) or the program termination, whichever is earlier. Their allocation is explicitly requested by the programmer, but it is the operating system that satisfies that request at runtime through an underlying system call.
- **a** and **b** are function parameters passed from main to sub() through the stack. Their scope is local to sub() and they exist during the duration of function sub(). They are allocated by the compiler.

Exploration (Interesting tidbits not formally covered in CS2106):

An executable (e.g. "a.out", "hello.exe") contains minimally the compiled instructions (machine code) and the global variables as laid out by the compiler. When we execute / run the executable, the machine code are loaded by an OS component known as **program loader**. A heavily simplified overview: the machine code will be loaded into the *text segment*, global variable into the *data segment*. The stack frame for the main function is then setup in the *stack segment* before execution begins.

In the old days when disk capacity was more limited, there was a compiler optimization to reduce the size of the executable. For the given code example, we can see that global variable **f** and **y** are initialized to 0, while **x** and pointer **p** are not. So, the compiler can place the **x** and pointer **p** in another special segment known as **BSS segment**, with other uninitialized global variables. BSS variables are often not allocated by the compile time (i.e. not in the executable), and instead only the size of the entire segment is specified in the executable. The OS *program loader* will allocate the BSS segment at runtime and fill it with zeros. This optimization was historically important, as it reduced the size of executable, which saved space on the disk, in memory, and resulted in faster loading of certain programs. Today, this optimization is fairly irrelevant.

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

   Given below is an **iterative** factorial function in 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 page 3. You should find most of it (vaguely) familiar from your basic assembly programming course. The remaining missing pieces are all related to 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, 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"). Recall that the second parameter of **`lw`**/**`sw`** has the form of **`offset(base address stored in register)`**, e.g. **`-12($fp)`** == content of register **`$fp - 12`**. You can assume that the **`$SP`** and **`$FP`** registers are initialized properly.

   b. [Stack Frame – Caller prepare 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 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 jump to the specified "function".

Complete the relevant portions tagged with "Part b".

c. [Stack Frame – Callee Enter Function] Now, let us fill in the instructions for callee to setup the stack frame upon entering the function. Tasks required:
   - Save the registers used in the function (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 Exit Function] At the end of the function, we need to:
   - Place the return result onto stack frame.
   - Restores the saved registers, FP, SP.
   - Return to caller with the saved PC. For simplicity, we assume there is a "**`return offset(register)`**" instruction, which overwrite 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 frame, calling convention and the usage of stack / frame pointers.

e. [Extra Challenge – Not discussed) Draft a solution for **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, figure 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 convention on different platforms, e.g. intel x86 (see exploration question 5), 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. ☺

MIP-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
      #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                      #Where should the "10" go?
    call iFact,           #start executing the function
```

ANS:

MIP-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
```

3. [Function Parameter - Midterm AY1819S1] 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()`			For `change()`		
		← **$SP**			← **$SP**
…	…		Saved SP	-8	← **$FP**
myInt	-12		Saved FP	-4	
Saved SP	-8	← **$FP**	Saved PC	0	
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.

- `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. [For your own exploration] 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:

[Further discussion on rationale and thinking process behind the question can be found in the sample solution for AY1819S1 Midterm.]

a. $R1 ← Load 4($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.

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

b. **Key Point:** If the reference is passed to another function as referece, 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 actually works 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 schme is not even applicable.

c. 2: Place the actual address of the value. Change the instruction to load/store from that address