CS2106 Operating Systems

Semester 2 2020/2021

Week of 25th January 2021

Tutorial 1

Process Abstraction

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

C
```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.

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
```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 4. 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
```

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()`

		← $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.

- `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.

For your own exploration:

4. [Process Control Block] Q2 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 Process Control Blocks (PCB) of the two processes (similar to lecture 2 on PCB), indicate clearly what is laid out in the physical memory and how the different memory regions of a process fit together.

5. [Understanding Function Call Convention] In the lecture, we gave an example (and rather complicated) function call convention. Let us now see an **actual** calling convention in action. Below is a simple C program, purposely written to utilize:
    a. Global variables
    b. Function with parameters and local variables
    c. Function calls

```c
#include <stdio.h>

int global = 0;

int function( int parameter )
{
    int local;

    local = parameter + 123;
    global = local + 456;

    return local;
}

int main()
{
    int result;

    result = function( 999 );

    return 0;
}
```

We generated the assembly code for the above C program on two different platforms:

i. **SPARC** (used by sunfire / suna server), assembly code in **`Sample.sparc.s`**.
ii. **Intel x86 64-bit**, assembly code in **`Sample.x86.s`**.

For each of the platforms, **make educated guess** for the following questions:

a. Is there a stack pointer and frame pointer?
b. Who (caller or callee) adjust the stack pointer in function entry?
c. How does the caller pass function parameter to the callee? (e.g. in stack frame? Register? Etc)

d. How does the callee pass back the return result to the caller?

Note: Before you complain on <insert latest social network here>, note that you **don't need** to understand the assembly code completely to attempt this question. As long as you have basic understanding of assembly code in general (instruction type, register, addressing mode etc), you should be able to **deduce the answer** in an educated fashion. You can also ignore any assembler directive ("instruction" starts with a ".").

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

```
int main()
{
     int input, result;

     printf("Give input below:\n");
     scanf("%d", &input);

     // takes a long time to compute
     result = ComplexFunc( input );

     printf("Result is %d\n", result);

     return 0;
}
```