CSC 501 PA0 - Extra Questions

Name: Jao-Jin Chang
Unity ID: jchang25

1. What is the difference in stack top address before and after calling printtos()? Draw a diagram to illustrate what are the contents of the items pushed into the stack between these two time points.

```
C/C++
Output:
void printtos()
Before[0x00ffef90]: 0x00ffefae
After [0x00ffef60]: 0x0001cacd
    element 1[0x00ffef64]: 0x00ffef60
    element 2[0x00ffef68]: 0x00ffef64
    element 3[0x00ffef6c]: 0x00ffef64
    element 4[0x00ffef70]: 0x00ffef84
```

a. Before:

Stack Register	Relatively Memory Location	Stack Value
%esp	0	12345 (Just a random value for demonstration purpose)
	+4	…
	+8	…

b. After:

Stack Register	Relatively Memory Location	Stack Value
%esp / %ebp	0	%ebp of the upper calling function
	+4	return address to the calling function
	+8	12345 (Previous %esp before calling printtos())
	+12	…

c. **Explanation**:

i. Before calling printtos():

1. At this point, printtos() has not been called at the high level. Therefore, no parameters or return address have been pushed onto the stack, and the stack remains unchanged.

Stack Register	**Stack**
%esp	12345 (random value, for demonstration purpose only)
…	…
…	…

ii. After preparing for printtos() but before executing the CALL instruction:

1. At this point, if the function being called has any parameters, they would be pushed onto the stack by the caller. However, since printtos() does not take any parameters, no additional values are pushed onto the stack in this scenario.

Stack Register	**Stack**
%esp	12345 (random value, for demonstration purpose only)
…	…
…	…

iii. After calling printtos() and executing the CALL instruction:

1. The return address of the calling function is pushed onto the stack.
2. The CALL instruction transfers control to the printtos() function by jumping to its entry point.

Stack Register	Relatively Memory Location	Stack
%esp	0	return address to the calling function
…	+4	12345 (Previous %esp before calling printtos())
…	+8	…

d. After entering printtos():

i. Upon entering printtos(), the program executes a procedure called the "Prologue," which typically includes the following instructions:

```
push %ebp     (Save the caller's base pointer)
mov  %esp, %ebp  (Set the base pointer)
```

ii. The purpose of the Prologue is to save the caller's base pointer (%ebp) so it can be restored after the function completes. It also sets a new base pointer (%esp -> %ebp) for the current function, ensuring a stable reference point for accessing arguments and local variables.

Stack Register	Relatively Memory Location	Stack Value
%esp / %ebp	0	%ebp of the upper calling function
	+4	return address to the calling function
	+8	12345 (Previous %esp before calling printtos())
	+12	…

2. Which byte order is adopted in the host machine that we are using? How did you find out?
    a. The hosting system (not XINU) is running using little endian.
    b. By using command:

```
C/C++
$ lscpu | grep "Byte Order"
Byte Order:            Little Endian
```

    c. Additionally, by writing a small C program with a variable and using GDB to inspect the value of the variable, we can confirm that the system uses little endian. This is evident because 0x12, the most significant byte, appears at the highest address, while 0x78, the least significant byte, appears at the lowest address.

```
C/C++
unsigned int num = 0x12345678
```

```
C/C++
(gdb) p/x num
$1 = 0x12345678
(gdb) x/4xb &num
0x7fffffffddbc: 0x78    0x56    0x34    0x12
```

3. Briefly describe the mov, push, pusha, pop, and popa instructions in the x86.
    a. **mov:**
        i. Copies data from the source operand to the destination operand.
        ii. Syntax:
            1. AT&T: movl <source>, <destination>
                a. In AT&T syntax, the size of the data is specified with a suffix
                    i. movl for 32-bit
                    ii. movw for 16-bit,
                    iii. movb for 8-bit operations.
            2. Intel X86: mov <destination>, <source>
        iii. Example:
            1. “movl %eax, %ebx”: moves a 32-bit value from %eax to %ebx.
    b. **push:**
        i. Decreases the stack pointer (%esp) and stores the value of the source operand at the new top of the stack.
        ii. Syntax:
            1. push <source>
        iii. Example:
            1. “push %eax” pushes the value of %eax onto the stack.
    c. **pusha:**
        i. Pushes the values of all general-purpose registers onto the stack with the following order:
            1. EAX
            2. ECX
            3. EDX
            4. EBX
            5. the initial value of ESP before EAX was pushed
            6. EBP
            7. ESI
            8. EDI
        ii. Syntax:
            1. pusha
        iii. It saves all registers in one instruction, unlike push, which works on a single register.
        iv. The value of %esp pushed is its initial value before the operation, unaffected by changes during the push operations.
    d. **pop:**
        i. Removes the top value from the stack and places it in the destination operand.
        ii. Syntax:
            1. pop <destination>

iii. Increases the stack pointer (%esp).
iv. Example: pop %eax retrieves the value from the top of the stack and stores it in %eax.
v.

e. popa:

i. The POPA instruction reverses the effect of the PUSHA instruction by popping the top eight words or doublewords from the stack into general-purpose registers, except for ESP
   1. The order is: EDI, ESI, EBP, skip (ESP), EBX, EDX, ECX, and EAX

ii. Syntax:
   1. popa

iii. The slot for %esp is skipped during the operation, but %esp is implicitly adjusted by the normal stack pointer update.