COMP124

Computer Systems

07 | External I/O Calls | Stack Frames

Dr Stuart Thomason

External Subroutines

- Input and output are hard to do in pure assembly coding
  - Use specific registers to point to address of source or destination in memory
  - Trigger a CPU interrupt to pass control to the operating system
  - Device drivers perform the I/O by liaising with hardware

- The previous lecture explained how to pass parameters into subroutines
  - We can call external subroutines (C library code) in the same way
  - But we need to be sure to follow the C calling convention (cdecl)

- We will use the I/O routines from the C standard library
  - printf – Send formatted output to the console (DOS/CMD window)
  - scanf – Wait for input from the console

Program Output

- We use the external printf subroutine from the C standard library
- In a C program the call would use high level syntax to specify its parameter

```
printf("Hello World");
```

- We don't have that luxury in assembly code, so we need to set up a few things first
  - Define the string as a C variable outside the assembly code block (this is a bit of a cheat to make coding easier on this module)
  - Find the address of the string inside the assembly code block
- Following the cdecl calling convention…
  - We must push the parameter to the stack
  - We must use pass by reference (ie. memory address of string)
  - We must clean up the stack afterwards

Complete Example

- This is the full example code for Visual Studio

```
#include <stdio.h>
#include <stdlib.h>

int main (void) {

    char msg[] = "Hello World\n";

    _asm {
        lea eax, msg     ; Get address of string
        push eax         ; Set up parameter
        call printf      ; Call external routine
        pop eax          ; Clean up stack
    }

    return 0;
}
```

string declared outside assembly code (using C syntax)

Corrupted Registers

- Consider the assembly code to output the string 10 times

```
        mov ecx, 10
floop:  lea eax, msg
        push eax
        call printf
        pop eax
        loop floop
```

- We don't know exactly what happens inside any external subroutine
  - It will almost certainly make use of registers
  - Which would overwrite whatever is stored in them
  - In this case, the loop counter (ECX) would be corrupted
  - So we need to save any register values that are important to our code
- We already have a convenient place to store things we want to keep safe
  - Push them to the stack!

Using the Stack

- We need to save things on the stack and then restore after the external call returns

```
        mov ecx, 10
floop:  push ecx
        lea eax, msg
        push eax
        call printf
        pop eax
        pop ecx
        loop floop
```

- Now the code will behave as we expect
  - Loop counter is saved to the stack
  - Doesn't matter if external subroutine corrupts the register
  - We restore it from the stack before the loop instruction uses it

Outputting Values

- The external routine can take extra parameters that store values to be outputted

```
int age = 21;
char name[] = "Bob";
printf("I am %s and I am %d years old\n", name, age);
```

- In a C program this would output…

```
I am Bob and I am 21 years old
```

- The string contains format specifiers that are replaced by each extra parameter
  - %d – Display as a decimal integer
  - %s – Display as a string
  - %c – Display as a single character
  - %f – Display as a floating-point number
- The extra parameters to printf must match the specifiers used in the string (and they must be given in the correct order of insertion)

Example Snippets

- Output a numeric value as part of a string
- Declare the string and the value in the C part of the code

```
char msg[] = "The number is %d\n";
int num = 7;
```

- The string (containing the format specifiers) is always the first parameter
- Any extra parameters represent the data to be output within the string
- We must push these in reverse order, following the cdecl calling convention

```
push num
lea eax, msg
push eax
call printf
add esp, 8
```

- Adding to the stack pointer is a quicker way to clean up multiple parameters at once

Program Input

- We use the external scanf subroutine from the C standard library
- In a C program the call would use high level syntax to specify what type of input we want and where in memory to store it

```
scanf("%d", &num);
```

- The C syntax uses an ampersand (&) to represent the memory address of a variable (you don't need to remember this; it won't be on the exam)
- Following the cdecl calling convention…
  - We must push the parameters to the stack in reverse order
  - We must use pass by reference (ie. memory addresses of parameters)
  - We must clean up the stack afterwards
- We can input strings (text) from the user in a similar way
  - Use the %s format specifier
  - Declare a char array that is big enough to store what they enter (easy to overflow)

Example Snippets

- Input a numeric value (integer) and store it in the accumulator
- Declare the format string and a variable for the input in the C part of the code

```
char fmt[] = "%d";
int num;
```

- We must push these in reverse order, following the cdecl calling convention

```
lea eax, num
push eax
lea eax, fmt
push eax
call scanf
add esp, 8
```

- Remember to push the address of the variable, not its value
- The external subroutine will wait for the user to type something and press enter
- Then it will store what they entered at the address passed to the routine

Stacking Local Variables

- In high level languages, subroutines can declare their own internal (local) variables

```
int subber (int n) {
    int x;
    int y;
    int z;
    …
}
```

- Local variables only exist while the subroutine is active, so it makes sense to store them on the stack

- When high level code is compiled, it will generate machine code instructions to reserve memory on the stack for each variable

Stack Frames

- Each time a subroutine is called, a new stack frame is created on the stack
- This holds data that is needed by the subroutine
  - Parameters
  - Return address
  - Local variables
- With nested calls, several (or many) stack frames will be present on the stack
- We have already looked at the ESP register (stack pointer)
  - Always points to the top of the stack
  - Will change as space is reserved for local variables
- The CPU has another register called EBP (stack base pointer)
  - This always points to the start of the current stack frame
  - Can be used to access parameters and local variables using an offset (eg. EBP−4 is the address of the second parameter that was pushed)

Building the Stack Frame

- ESP always points to the top of the stack
- EBP initially points to the base of the stack (remember the stack grows downwards)
- When a subroutine is called…
  - Parameters are pushed to the stack first
  - Then the return address is pushed
  - Then value of EBP is pushed
  - Local variables are reserved on the stack (causing ESP to change)
  - Current value of ESP is put into EBP (to begin a new stack frame)
- When a subroutine is ready to return…
  - Remove any local variables from the stack
  - Pop top value into EBP (to restore previous stack frame)
  - Pop top value into EIP (to move execution back to caller)
- Finally, the caller is responsible for cleaning any parameters from the stack

Nested Calls and Stack Frames

- Consider a main program that calls a subroutine
- The subroutine calls another nested subroutine
- The stack grows as a stack frame is built up
  - Parameters
  - Return address
  - Old base pointer
  - Local variables
- The values of EBP and ESP change as the calls happen
  - ESP always points to the top of the stack (remember it grows downwards)
  - EBP changes with each subroutine call
- Stack is cleaned up (shrinks) as each subroutine returns