Subroutines and mixed ASM/C programming

Stack management – POP, PUSH

- POP,
  - syntax: *pop dest*
  - Dest must be a 32/64 bit register in a 32/64 bit architecture
- PUSH,
  - syntax: *push var/reg*
  - Dest must be a 32/64 bit register in a 32/64 bit architecture

Move to/from the stack and updates ESP/RSP

Functions – CALL, RET *(http://pages.cs.wisc.edu/~remzi/Classes/354/Fall2012/Handouts/Handout-CallReturn.pdf)*

CALL, syntax: CALL _function

- similar to a JUMP but stores EIP on the stack (for the return)
  1. EIP -> stack ; This is done by the CALL instruction
  2. EBP -> stack ;
  3. EBP <- ESP ; actually a “calling convention” abstraction
  4. ESP is decremented to, among several things, contain the local variables of _function
  5. EIP=OFFSET _function

RET, syntax: RET/RET num

1. EBP <- ESP ; restore the saved EBP
2. EIP <- ESP ; restore RETURN address

Stack Evolution during x86 Call/Return

1. Save caller registers onto the stack

   ```
   push ECX
   ```

2. Push arguments in reverse order (arg N to 0)

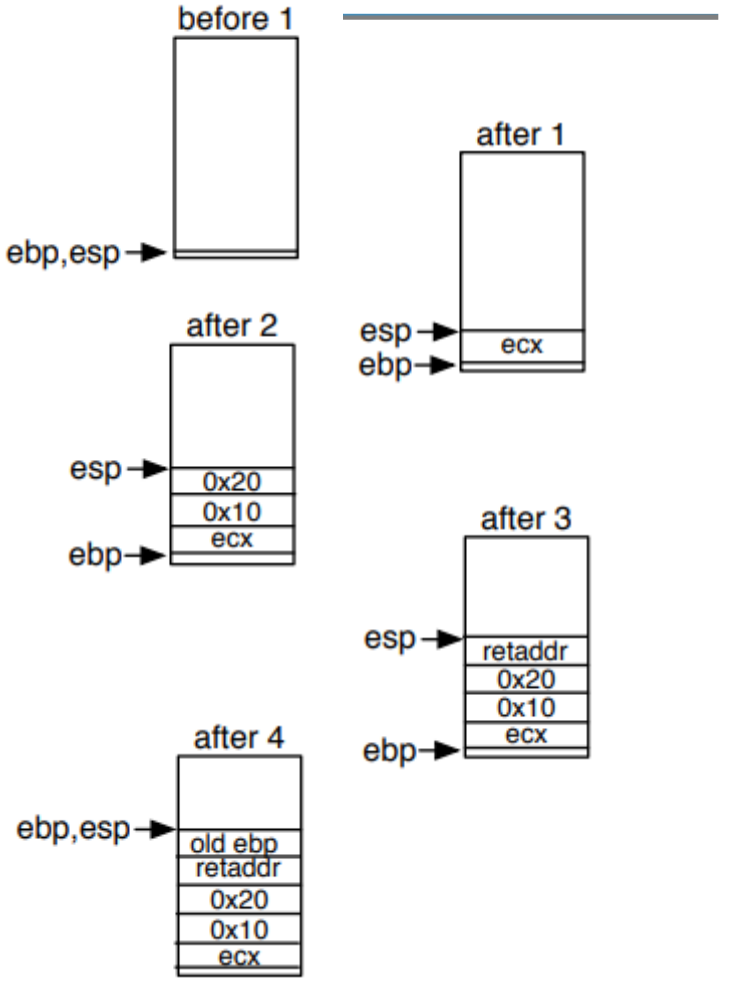
   ```
   push 0x10 // argument 2
   push 0x20 // argument 1
   ```

3. Call function (ret addr. onto the stack)

   ```
   call function_name
   ```

4. Update base pointer

   ```
   push EBP
   movl EBP, ESP
   ```



Stack Evolution during x86 Call/Return

5. Save callee register

   ```
   push  EBX
   ```

6. Make room for local variables

   ```
   sub  ESP,0x08
   ```

7. Execute body of routine (using BP to access arguments)

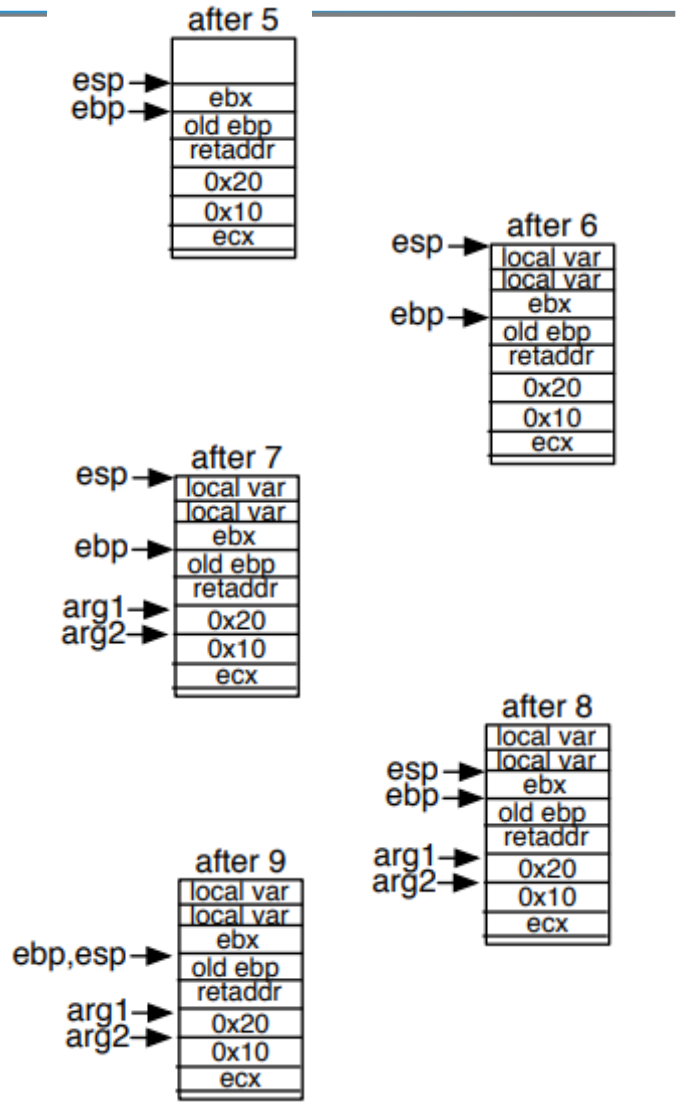
   ```
   movl  ECX,0x08(EBP)
   movl  EDX,0x0C(EBP)
   ```

8. Deallocattion: Fee local stack space

   ```
   add  ESP,0x08
   ```

9. Restore callee registers

   ```
   pop  EBX
   ```



Stack Evolution during x86 Call/Return

10. Restore old base pointer

    `pop EBP`

11. Return from function (pop return addr)

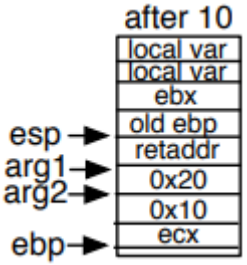
    `ret`

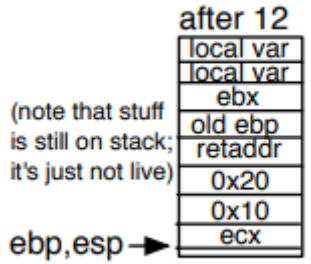
    and free input parameters space

    `add ESP,0x08`

12. Restore caller registers

    `pop ECX`





CALLing conventions

- On 32-bit x86 on Linux, the calling convention is named cdecl
- caller (parent) pushes the arguments from right to left onto the stack, calls the target function (callee/child), receives the return value in eax, and pops the arguments

Platform	Return Value	Parameter Registers	Additional Parameters	Stack Alignment
System V i386	eax, edx	none	stack (right to left)[1]	
System V X86_64[2]	rax, rdx	rdi, rsi, rdx, rcx, r8, r9	stack (right to left)[1]	16-byte at call[3]

GCC disassembly (Intel Style & AT&T style)

1. **Register Naming**: prefixed with % => registers are %eax, %cl etc (not eax, cl, …)
2. **Ordering of operands**: source(s) first, and destination last.
   Intel syntax "mov eax, edx"
   AT&T assembly "mov %edx, %eax"
3. **Operand Size**: In AT&T syntax, the size of memory= suffix l
   b for (8-bit) byte, w for (16-bit) word, and l for (32-bit) long
   "movl %edx, %eax".
4. **Immediate Operand**: marked with a $ prefix
   "addl $5, %eax"
5. **Memory Operands**: Missing operand => memory-address;
   "movl $bar, %ebx" puts the address of variable bar into register %ebx,
   "movl bar, %ebx" puts the contents of variable bar into register %ebx.
6. **Indexing**: Indexing or indirection is done by enclosing the index register or indirection memory cell address in parentheses.
   "movl 8(%ebp), %eax" (moves the contents at offset 8 from the cell pointed to by %ebp into register %eax).

Exercise

- Create and compile (try also –S) a simple program that sums up two integer in a function
- Disassemble it:

  `objdump -d` *`filename`*
- Compile again with the –g option (include debug info)
- Now disassemble with `objdump -S` *`filename`*
- Debug the program using gdb

Inline ASM code (inline1.c)

```
#include <stdio.h>

int main() {
  /* Add 10 and 20 and store result into register
       %eax */
     __asm__ ( "movl $10, %eax;"
               "movl $20, %ebx;"
               "addl %ebx, %eax;"
     );

  /* Subtract 20 from 10 and store result into
       register %eax */
  __asm__ ( "movl $10, %eax;"
               "movl $20, %ebx;"
               "subl %ebx, %eax;"
  );

  /* Multiply 10 and 20 and store result into
       register %eax */
     __asm__ ( "movl $10, %eax;"
               "movl $20, %ebx;"
               "imull %ebx, %eax;"
     );

  return 0 ;
}
```

Operands from/to variables

```
asm ( assembler template
    : output operands                (optional)
    : input operands                 (optional)
    : clobbered registers list       (optional)
    );
```

- Output operands are output variables
- Input operands are input variables

- In “assembler templates” output and input operands are referenced as %0 %1 …

Example: inline2.c

```
int no = 100, val ;
__asm__ ("movl %1, %%ebx;"
      "movl %%ebx, %0;"
      : "=r" ( val )   /* output «=r» param %0 */
      : "r" ( no )     /* input param %1 */
      : "%ebx"         /* clobbered register (GCC do
                          not use*/
  );
```

- Constraints:

  r = a register to be used

  g = whatever the compiler prefers (memory, register, literal)

  a = eax b=ebx c=ecx d=edx S=ESI D=EDI

Lab 2

Mixed C/ASM programming

Debugging

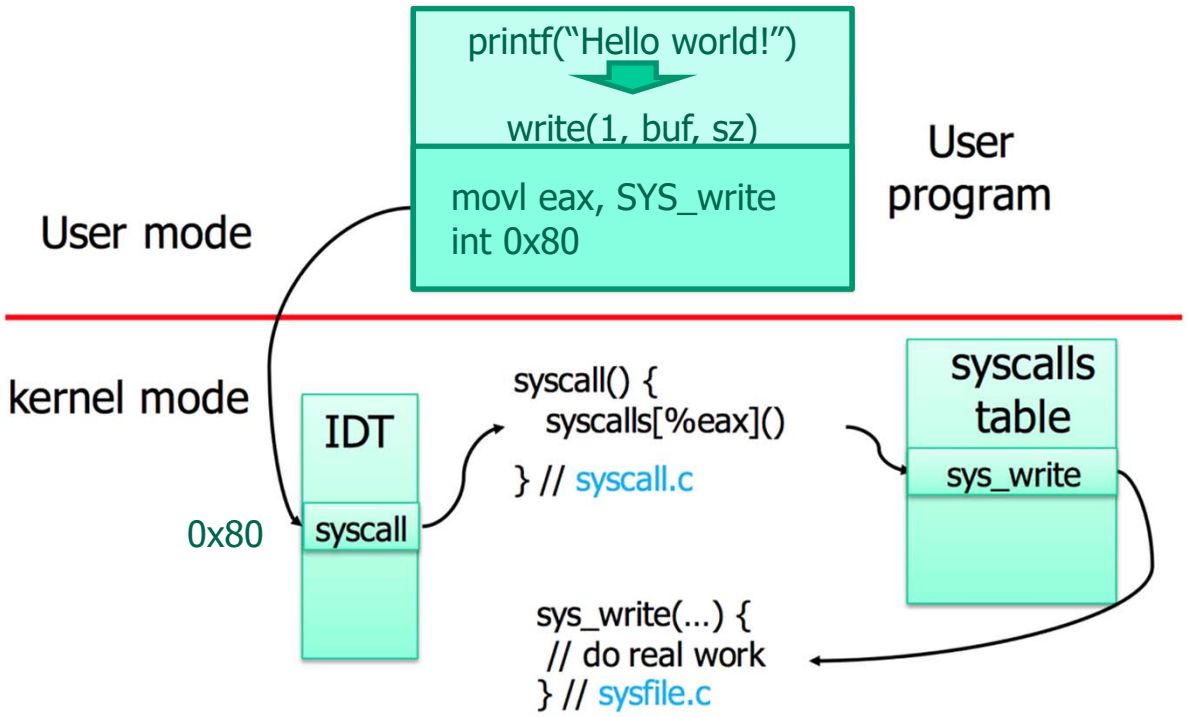
- Considers the program inline3_error.c (next slide)
- Compile and execute it…you will get an error…
- Debug by using gdb until the error
- When you find the error line, checks registers to figure out why there is an error…
- Revise the semantic of the instruction resulting in the mistake
- Fix the error…
- Finally try to reorganize the program in subroutines

Complete example (find the error and fix it – inline3_error.c -> inline3.c)

```
#include <stdio.h>

int main() {

   int arg1, arg2, add, sub, mul, quo, rem ;

   printf( "Enter two integer numbers : " );
   scanf( "%d%d", &arg1, &arg2 );

 /* Perform Addition, Subtraction, Multiplication &
Division */
   __asm__ ( "addl %%ebx, %%eax;" : "=a" (add) :
        "a" (arg1) , "b" (arg2) );
   __asm__ ( "subl %%ebx, %%eax;" : "=a" (sub) :
        "a" (arg1) , "b" (arg2) );
   __asm__ ( "imull %%ebx, %%eax;" : "=a" (mul) :
        "a" (arg1) , "b" (arg2) );

   __asm__ ( "movl $0x0, %%edx;"
         "movl %2, %%eax;"
         "movl %3, %%ebx;"
          "idivl %%ebx;" : "=a" (quo), "=d" (rem) :
          "g" (arg1), "g" (arg2) );

   printf( "%d + %d = %d\n", arg1, arg2, add );
   printf( "%d - %d = %d\n", arg1, arg2, sub );
   printf( "%d * %d = %d\n", arg1, arg2, mul );
   printf( "%d / %d = %d\n", arg1, arg2, quo );
   printf( "%d %% %d = %d\n", arg1, arg2, rem );

   return 0 ;
}
```

System Calls

SYSTEM CALL



An interface between application and OS kernel

- Linux int 0x80 (syscall x86_64)
- Win int 0x2e

eax = # syscall in syscalls table (e.g. 1 = write)

Gp registers host parameters (syscall dependents)

helloWorld.asm

```
SECTION .data

Hello:      db "Hello
world!"
len_Hello:  equ $-Hello

SECTION .text
global _start
_start:
      mov rax,1
; write syscall (x86_64)
      mov rdi,1
; fd = stdout
      mov rsi,Hello
; *buf = Hello
      mov rdx,len_Hello
; count = len_Hello
      syscall

      mov rax,60
; exit syscall (x86_64)
      mov rdi,0
; status=0 (exit normally)
      syscall
```

References

- http://www.cs.umd.edu/~meesh/cmsc311/links/handouts/ia32.pdf
- https://sensepost.com/blogstatic/2014/01/SensePost_crash_course_in_x86_assembly-.pdf
- http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
- https://www.codeproject.com/Articles/15971/Using-Inline-Assembly-in-C-C
- https://www.youtube.com/watch?v=75gBFiFtAb8
- https://software.intel.com/en-us/articles/introduction-to-x64-assembly
- https://wiki.osdev.org/Inline_Assembly