Subprograms II

Ned Nedialkov

McMaster University
Canada

SE 3F03
February 2014

Outline

Interfacing assembly and C

Return values

Calling C from assembly

Mechanism

C variables

Interfacing assembly and C

C assumes that a call to a subroutine

- does not change
  - **ebx**, **esi**, **edi**, **ebp**, **es**, **ds**, **ss**, **es**
  - a subroutine must save and restore any of them if changed
- can change
  - **eax**, **ecx**, **edx**

Most C compilers append _ before a name of function or global or static variable
A Fortran compiler appends _ after a function name

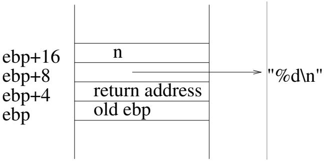
Return values

- **char**, **int**, **enum** are returned in **eax**
  If smaller than 32 bits: extended to 32 bits
- 64 bit values are returned in **edx**:**eax**
- Pointers are returned in **eax**

Calling C from assembly

Example: calling `printf("%d\n", n);`

- Parameters are pushed onto the stack from right to left



- `printf` knows that
  - the first parameter is at **ebp**+8
  - from the format string there is one parameter that is an integer
  - this integer is at **ebp**+16

```
segment .data
format db "%d\n", 0
segment .text
;;
push eax         ;push n
push dword format;push string address
call _printf
add esp, 8       ;clear parameters
;;
```

Example: computing array sum

Adapted from http://www.drpaulcarter.com/pcasm/

```
; subroutine calc_sum
; finds the sum of the integers 1 through n
; Parameters:
;   n    - what to sum up to (at [ebp + 8])
;   sump - pointer to int to store sum into (at [ebp + 12])
; pseudo C code:
; void calc_sum( int n, int * sump )
; {
;   int i, sum = 0;
;   for( i=1; i <= n; i++ )
;     sum += i;
;   *sump = sum;
; }
segment .text
        global  calc_sum
calc_sum:
        enter   4,0               ; allocate room for sum on stack
        push    ebx               ; should be preserved
        mov     dword [ebp-4],0   ; sum = 0
        mov     ecx, 1            ; ecx is i in pseudocode
```

```
for_loop:
        cmp     ecx, [ebp+8]        ; cmp i and n
        jnle    end_for             ; if not i <= n, quit
        add     [ebp-4], ecx        ; sum += i
        inc     ecx
        jmp     short for_loop
end_for:
        mov     ebx, [ebp+12]       ; ebx = sump
        mov     eax, [ebp-4]        ; eax = sum
        mov     [ebx], eax
        pop     ebx                 ; restore ebx
        leave
        ret
```

```
#include <stdio.h>
int main( void ){
  int n, sum;
  printf("Sum␣integers␣up␣to:␣");
  scanf("%d", &n);
  calc_sum(n, &sum);
  printf("Sum␣is␣%d\n", sum);
  return 0;
}
```

Addresses of local variables

- Local variables are at **ebp**`-n`, `n` is a multiple of 4
- In `scanf` we need to pass the address of `n`
- `n` is at **ebp**`-4`
- **mov eax**, **ebp**`-4` does not work
  - the value **mov** stores in **eax** must be computed by the assembler
  - it does not know the value of **ebp**`-4`
- **lea eax**, `[`**ebp**`-4]`
  - load effective address
  - calculates the address of `[`**ebp**`-4]`
  - we can push it onto the stack before calling `scanf`

Mechanism

- In the caller
  - Push parameters onto the stack from right to left
    Caller must keep track how many are pushed
  - Call the function
  - The processor pushes `EIP` onto the stack
  - `EIP` contains the address of the first byte after the **call** instruction

- In the callee
  - save and update **ebp** (**ebp** is associated with the caller)

    ```
    push ebp
    mov  ebp,esp
    ```
  - arguments are accessed at **ebp**`+8`, `+12`, ...
  - allocate space for local variables by subtracting from **esp**
  - save registers used for temporaries
  - execute the body of the function
  - restore saved registers
  - release local storage; e.g. add to **esp**
  - restore old **ebp**
  - return

    ret pops `EIP`
- In the caller
  - Clean up pushed parameters

C variables

- global
  - can be accessed everywhere in a file
  - if not static, can be accessed from any other file
  - in `.bss` or `.data` segments
- static
- local
  - can be accessed only in the block where they are declared
- register
  - hint to the compiler to put it in a register
- volatile
  - its value can be changed at any time
  - the compiler cannot optimize using this variable

Consider

```
void foo()
{
  int *addr;
  addr = 100;
  *addr = 0;
  while (*addr!=255)
    ;
}
```

A compiler woud optimize to

```
void foo()
{
  while (1)
    ;
}
```

To prevent the compiler from optimizing, use **volatile**:

```
void foo()
{
  volatile int *addr;
  addr = 100;
  *addr = 0;
  while (*addr!=255)
    ;
}
```