Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

CS Basics
7) Procedures

E. Benoist & C. Grothoff

Fall Term 2018-19

Computer Science Division

Procedures in Assembly

- Procedures
  - Syntax
  - Save Registers
  - Recursion
- Data
  - Local Data
  - Local Labels
- Example: hexdump2.asm
- Libraries
- Macros

Procedures

Need for Procedures

- **Large monolithic program**
  - overview impossible
  - impossible to test properly
  - tasks have to be done sequentially
- **Procedures**
  - Used for finer granularity in programming
  - Can be called once or more
  - Can be reused later
- **Examples of procedures**
  - Compute the logarithm
  - Read a string from the `stdin`
  - Transform a string into a number
  - ...

Procedure Call

- **Procedure:**
  - A piece of code
  - Intended to be called from anywhere in code
  - That returns to this code afterward
- **Difference with jumps**
  - Jumps should remain inside a procedure
  - Jumps are not intended to come back
  - Return: goes back where it was called (similar to interrupts)

Syntax

Syntax

- **Call the procedure**

```
call LoadBuff
```

- **Definition of the procedure**
  - This procedure loads the stdin into the buffer (and the number of bytes is in EBP)

```
LoadBuff:
 push eax      ; Save caller s EAX
 push ebx      ; Save caller s EBX
 push edx      ; Save caller s EDX
 mov eax,3     ; Specify sys_read call
 mov ebx,0     ; Specify File Descriptor 0: Standard Input
 mov ecx,Buff  ; Pass offset of the buffer to read to
 mov edx,BUFFLEN ; Pass nb of bytes to read at one pass
 int 80h     ; Call sys_read to fill the buffer
 mov ebp,eax ; Save # of bytes read from file for later
 xor ecx,ecx   ; Clear buffer pointer ECX to 0
 pop edx                    ; Restore caller s EDX
 pop ebx                    ; Restore caller s EBX
 pop eax                    ; Restore caller s EAX
 ret                ; And return to caller
```

Calling a procedure

- **CALL pushes the return address**
  - Then transfers the execution to the addess represented by the label
- **Procedure is terminated by the instruction RET**
  - Pops the address *off*
  - Transfers execution to this address
- **Similar to Interrupts**
  - But CALL does know the address
  - Whereas INT just knows the number of the interrupt

Save Registers

Values of Registers

- **Some registers may be used as input and output**
  - Parameters may be placed inside specific registers
  - Result may appear inside a given register
- **Most of registers need to be saved**
  - The procedure may need to use additional registers
  - But they could already be in use
- **Solution**
  - Store register values on the stack
  - Inside the procedure, each used register is saved on the stack
  - When the job is finished (before RET) the registers are restored with the saved values
  - Both should be done with all *caller-save* registers used inside the procedure

Calling a procedure and returning

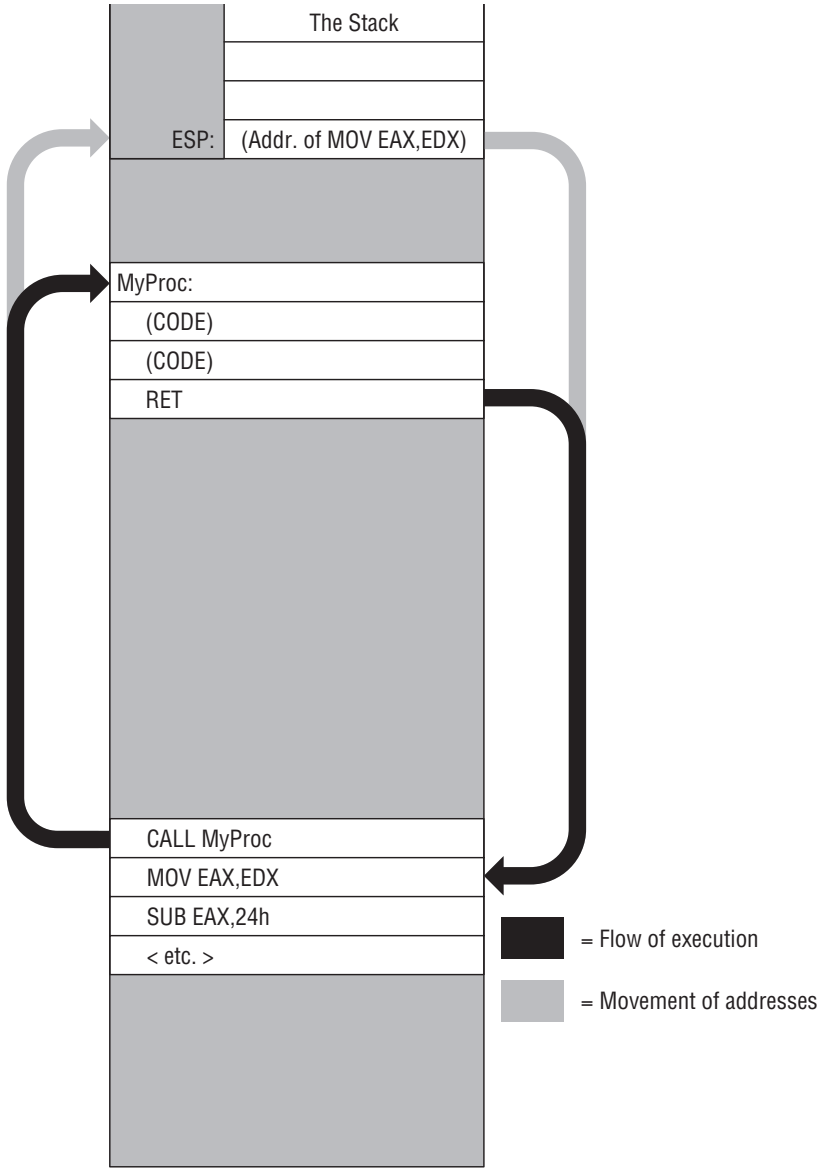

Figure 10-1: Calling a procedure and returning

Recursion

Calls within Calls

- **Within a procedure you can do anything**
  - Includes: calling a procedure

```
ClearLine:
push rdx   ; Save caller's registers
  push rax
  mov edx,15   ; We are going to go 16 pokes,↘
  → counting from 0
.poke:
  mov eax,0   ; Tell DumpChar to poke a '0'
  call DumpChar   ; Insert the '0' into the ↘
  →hex dump string
  sub edx,1   ; DEC does not affect CF!
  jae .poke   ; Loop back if EDX >= 0
  pop  rax      ; Restore all caller s GP ↘
  →registers
  pop rdx
  ret           ; Go home
```

Recursion

- **Some functions call themselves: Recursion**
- **Example: exponential**
  - Function computing x^y
  - To compute x^y we test if $y = 0$ then return 1
  - Otherwise, we compute $z = x^{y-1}$ and multiply z with x to obtain the result
- **Danger with recursion**
  - The stack is used each time to store variables
  - The stack may overflow if recursion is not correctly used
  - Stack collides with other memory: undefined behavior[1]
  - In Java: Stack overflow

Further reading: **tail calls**.

[1] On modern Linux: usually crash due to page fault in guard page.

Data

Procedures and data

- **Procedures need data**
  - As input
  - Produce data as output
  - Two types of data: `global` and `local`
- **Global data**
  - Is accessible to any code anywhere in the program
  - Is defined in `.data` or `.bss` sections
  - CPU registers are also global and can be accessed from anywhere
- **Simple program**
  - Use registers to send parameters
  - Example: interrupt 80h, inputs are put in RAX, RBX, ...
  - Tables and buffers are accessed like in any part of program: with memory address "between the brackets"

Saving registers

- **You will never have enougth registers**
  - You can not create variables like in Java
  - Programs are limited by the registers
  - You can not know what is in a register
- **Need to protect the values of the caller program**
  - If a register is used in the program as a counter
  - Should not clobber it for another purpose
- **Solution**
  - Save the registers before to change them
  - Store values on the stack
  - In the end of the procedure: restore all values from the stack

Save and restore registers

- **Example of code**
  Store the registers you will use

```
push rbx
push rsi
push rdi
```

  In the end of the procedure restore them

```
pop rdi
pop rsi
pop rbx
```

- **Important**
  - Values must be POPed in reverse order!

Caller save and callee save registers

Modern systems distinguish

- caller save registers: procedure must prevent clobbering
- callee save registers: caller must "save" **before** call

Which registers are of what type is defined in the **calling conventions** of the CPU architecture and programming language.

This only matters if you interact with compilers!

Local Data

Local Data

- **Only accessible to a particular procedure**
  - Data that is placed on stack when a procedure is called
  - Data is PUSHed on the stack before the CALL
  - The caller sends data to the procedure
- **In the procedure**
  - Can not pop the data (remember the return address)
  - Anything PUSHed on the stack before is under the return address in the stack
  - Memory needs to be accessed manually
  - Takes a lot of care and discipline

Constant data in code definition

- **Possiblity to define data within the .text section**
  - After the RET instruction one can define data
  - Data and program are just data
  - Need a label

```
Newlines:
        push ecx    ; Save the status of the ↘
        →registers into the stack
        push eax
        ...
        ret                 ; Return to the ↘
        →calling program
MyStr:    db "Hello␣World",10
```

Local Labels

Local Labels

- **Programs get larger**
  - You need more and more labels (for loops, jumps, ...)
  - You will use twice the same label - Big problem
  - Names of local labels start with a period (.)
- **Example**

```
Scan:
    xor rax, rax    ; errase value in RAX
   ....
.loop
      mul rcx           ; multiply rax by rcx
      sub rbx, 1     ; decrement rbx
      jnz .loop        ; loop to the .loop label
```

- Local labels can not be referenced outside their global label (here `Scan`)
- i.e. the global label before their position
- **Force access to a local label**
  - To access a local label from outside: concatenate the global label and the local label
  - `Scan.loop` can be accessed from anywhere

Local and global labels

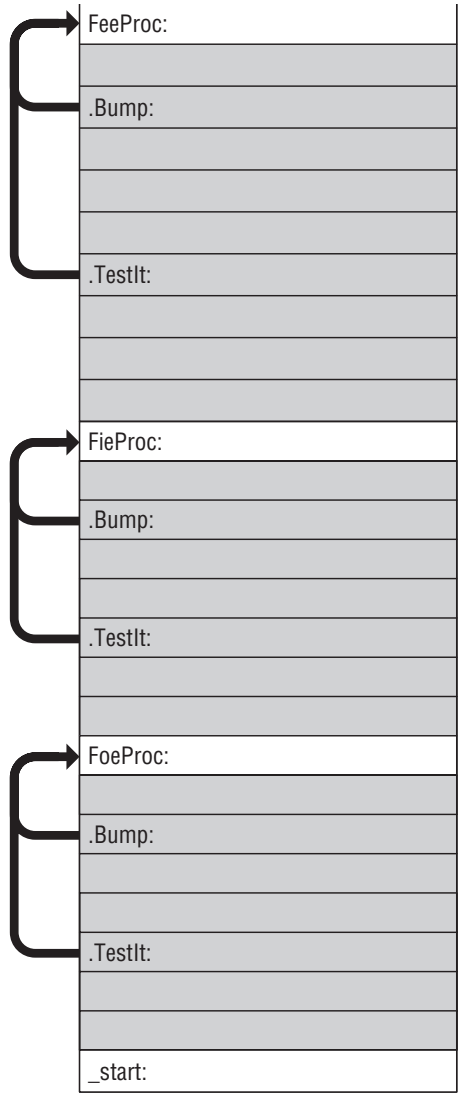

Figure 10-2: Local labels and the globals that own them

Short, Near and Far jumps

- **Jumps can be of three types**
  - `Short` within 127 bytes in code
  - `Near` Inside the same code segment
  - `Far` anywhere inside the code
  - Default: `Short`
- **Syntax**

```
jne Scan    ; Short jump, within 127 bytes↘
→ in either directions
jne near Scan   ; Near jump anywhere in ↘
→the current code segment
```

- `Strategy: insert NEAR each time you receive an error ‘‘Short jump is out of range’’.`

Example: hexdump2.asm

Example: hexdump2.asm

```
;   Executable name : hexdump2                                  ;   ↘
→Version             : 1.0
;   Created date    : 4/15/2009 ↘
→                                ;   Last update      : ↘
→4/20/2009
;   Author          : Jeff Duntemann
;   Description     : A simple hex dump utility ↘
→demonstrating the use of
;                          assembly language procedures
;
;   Build using these commands:
;     nasm -f elf -g -F stabs hexdump2.asm
;     ld -o hexdump2 hexdump2.o
;
SECTION .bss                          ; Section containing ↘
→uninitialized data
        BUFFLEN EQU 10
        Buff    resb BUFFLEN
SECTION .data                         ; Section containing ↘
→initialised data
; Here we have two parts of data structure. The first part↘
→ displays 16 bytes in
; hex separated by spaces. Immediately following is a 16-↘
→character line
```

Example: hexdump2.asm

```
; The HexDigits table is used to convert numeric values to their hex
; equivalents. Index by nybble without a scale: [HexDigits+eax]
HexDigits:      db "0123456789ABCDEF"

; This table is used for ASCII character translation, into the ASCII
; portion of the hex dump line, via XLAT or ordinary memory lookup.
; All printable characters "play through" as themselves. The high 128
; characters are translated to ASCII period (2Eh). The non-printable
; characters in the low 128 are also translated to ASCII period, as is
; char 127.
DotXlat:
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 20h,21h,22h,23h,24h,25h,26h,27h,28h,29h,2Ah,2Bh,2Ch,2Dh,2Eh,2Fh
        db 30h,31h,32h,33h,34h,35h,36h,37h,38h,39h,3Ah,3Bh,3Ch,3Dh,3Eh,3Fh
        db 40h,41h,42h,43h,44h,45h,46h,47h,48h,49h,4Ah,4Bh,4Ch,4Dh,4Eh,4Fh
        db 50h,51h,52h,53h,54h,55h,56h,57h,58h,59h,5Ah,5Bh,5Ch,5Dh,5Eh,5Fh
        db 60h,61h,62h,63h,64h,65h,66h,67h,68h,69h,6Ah,6Bh,6Ch,6Dh,6Eh,6Fh
        db 70h,71h,72h,73h,74h,75h,76h,77h,78h,79h,7Ah,7Bh,7Ch,7Dh,7Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
```

Example: hexdump2.asm

```
SECTION .text                    ; Section containing code
;-----------------------------------------------------------
; ClearLine:    Clear a hex dump line string to 16 0 values
; UPDATED:      4/15/2009
; IN:           Nothing
; RETURNS:      Nothing
; MODIFIES:     Nothing
; CALLS:        DumpChar
; DESCRIPTION:  The hex dump line string is cleared to binary 0 by
;               calling DumpChar 16 times, passing it 0 each time.

ClearLine:
        pushad                   ; Save all caller s GP registers
        mov edx,15               ; We are going to go 16 pokes, counting from ↘
        →0
.poke:  mov eax,0          ; Tell DumpChar to poke a '0'
        call DumpChar      ; Insert the '0' into the hex dump string
        sub edx,1                ; DEC does not affect CF!
        jae .poke                        ; Loop back if EDX >= 0
        popad                    ; Restore all caller s GP registers
        ret                              ; Go home
```

Example: hexdump2.asm

```
; DumpChar:       "Poke" a value into the hex dump line string.
; UPDATED:        4/15/2009
; IN:             Pass the 8-bit value to be poked in EAX. Pass the value s ↘
→position in the line (0-15) in EDX
; RETURNS:        Nothing
; MODIFIES:       EAX, ASCLin, DumpLin
; CALLS:          Nothing
; DESCRIPTION:    The value passed in EAX will be put in both the hex dump
;                 portion and in the ASCII portion, at the position passed
;                 in EDX, represented by a space where it is not a printable ↘
→character.
DumpChar: ; First we insert the input char into the ASCII portion of the dump ↘
→line
        push ebx                    ; Save caller s EBX
        push edi                    ; Save caller s EDI
        mov bl,byte [DotXlat+eax]       ; Translate nonprintables to '.'
        mov byte [ASCLin+edx+1],bl      ; Write to ASCII portion
; Next we insert the hex equivalent of the input char in the hex portion of ↘
→the hex dump line:
        mov ebx,eax                 ; Save a second copy of the input char
        lea edi,[edx*2+edx]         ; Calc offset into line string (ECX X 3)
; Look up low nybble character and insert it into the string:
        and eax,0000000Fh               ; Mask out all but the low nybble
        mov al,byte [HexDigits+eax]   ; Look up the char equiv. of nybble
        mov byte [DumpLin+edi+2],al   ; Write the char equiv. to line string
; Look up high nybble character and insert it into the string:
        and ebx,000000F0h           ; Mask out all the but second-lowest nybble
        shr ebx,4                   ; Shift high 4 bits of byte into low 4 bits
        mov bl,byte [HexDigits+ebx] ; Look up char equiv. of nybble
        mov byte [DumpLin+edi+1],bl ; Write the char equiv. to line string
;Done! Let s go home:
        pop edi                     ; Restore caller s EDI
        pop ebx                     ; Restore caller s EBX
```

Example: hexdump2.asm

```
;-----------------------------------------------------
; PrintLine:     Displays DumpLin to stdout
; UPDATED:       4/15/2009
; IN:            Nothing
; RETURNS:       Nothing
; MODIFIES:      Nothing
; CALLS:         Kernel sys_write
; DESCRIPTION:   The hex dump line string DumpLin is ↘
→displayed to stdout
;                using INT 80h sys_write. All GP ↘
→registers are preserved.

PrintLine:
        pushad                  ; Save all caller s GP ↘
        →registers
        mov eax,4               ; Specify sys_write call
        mov ebx,1               ; Specify File Descriptor 1:↘
        → Standard output
        mov ecx,DumpLin         ; Pass offset of line string
        mov edx,FULLLEN         ; Pass size of the line ↘
        →string
```

Example: hexdump2.asm

```
;-----------------------------------------------------------------
; LoadBuff:       Fills a buffer with data from stdin via INT 80h sys_read
; UPDATED:        4/15/2009
; IN:             Nothing
; RETURNS:        # of bytes read in EBP
; MODIFIES:       ECX, EBP, Buff
; CALLS:          Kernel sys_write
; DESCRIPTION:    Loads a buffer full of data (BUFFLEN bytes) from stdin
;                 using INT 80h sys_read and places it in Buff. Buffer
;                 offset counter ECX is zeroed, because we are starting in
;                 on a new buffer full of data. Caller must test value in
;                 EBP: If EBP contains zero on return, we hit EOF on stdin.
;                 Less than 0 in EBP on return indicates some kind of error.

LoadBuff:
        push eax            ; Save caller s EAX
        push ebx            ; Save caller s EBX
        push edx            ; Save caller s EDX
        mov eax,3           ; Specify sys_read call
        mov ebx,0           ; Specify File Descriptor 0: Standard Input
        mov ecx,Buff        ; Pass offset of the buffer to read to
        mov edx,BUFFLEN     ; Pass number of bytes to read at one pass
        int 80h             ; Call sys_read to fill the buffer
        mov ebp,eax         ; Save # of bytes read from file for later
        xor ecx,ecx         ; Clear buffer pointer ECX to 0
        pop edx             ; Restore caller s EDX
        pop ebx             ; Restore caller s EBX
        pop eax             ; Restore caller s EAX
        ret                 ; And return to caller
```

Example: hexdump2.asm

```
GLOBAL _start
;-----------------------------------------------------------
; MAIN PROGRAM BEGINS HERE
;-----------------------------------------------------------
_start:
        nop                     ; No-ops for GDB
; Whatever initialization needs doing before the loop scan starts is here:
        xor esi,esi             ; Clear total byte counter to 0
        call LoadBuff           ; Read first buffer of data from stdin
        cmp ebp,0               ; If ebp=0, sys_read reached EOF on stdin
        jbe Exit

; Go through the buffer and convert binary byte values to hex digits:
Scan:
        xor eax,eax             ; Clear EAX to 0
        mov al,byte[Buff+ecx]   ; Get a byte from the buffer into AL
        mov edx,esi             ; Copy total counter into EDX
        and edx,0000000Fh       ; Mask out lowest 4 bits of char counter
        call DumpChar           ; Call the char poke procedure

; Bump the buffer pointer to the next character and see if buffer s done:
        inc esi                 ; Increment total chars processed counter
        inc ecx                 ; Increment buffer pointer
        cmp ecx,ebp             ; Compare with # of chars in buffer
        jb .modTest             ; If we ve processed all chars in buffer...
        call LoadBuff           ; ...go fill the buffer again
        cmp ebp,0               ; If ebp=0, sys_read reached EOF on stdin
        jbe Done                ; If we got EOF, we're done
```

Example: hexdump2.asm

```
; See if we are at the end of a block of 16 and need ↘
→to display a line:
.modTest:
        test esi,0000000Fh      ; Test 4 lowest bits ↘
        →in counter for 0
        jnz Scan                                    ; If ↘
        →counter is *not* modulo 16, loop back
        call PrintLine                          ; ...otherwise↘
        → print the line
        call ClearLine                          ; Clear hex ↘
        →dump line to 0 s
        jmp Scan                                    ; ↘
        →Continue scanning the buffer

; All done! Let s end this party:
Done:
        call PrintLine          ; Print the "leftovers↘
        →" line
Exit:   mov eax,1               ; Code for Exit ↘
        →Syscall
```

Libraries

Libraries

- **Problem with big files**
  - Difficult to manage
  - Copy paste of the same functionalities
  - Need to reuse the same code many times
  - Not efficient / error prone
- **Libraries**
  - Assembly language files compiled on their own
  - Contain useful the methods
  - Can compiled by themeselves
  - Can be tested and debugged once and reused everywhere

Libraries

- **Definition of a library of procedures**
  - Separate file
  - Contains the definition of the procedure
  - Needs to define procedures
  - Public procedures are declared as `GLOBAL`
- **Use of the procedure**
  - In the file where we use the procedure
  - Declare a label as `EXTERN`
  - Use the procedure exactly as if it were in the same file
- **Joined at linking**
  - References are left open by compiling, object file contains unsolved references.
  - Solved at linking
- **Variables can also be EXTERNal**

Connecting globals and externals

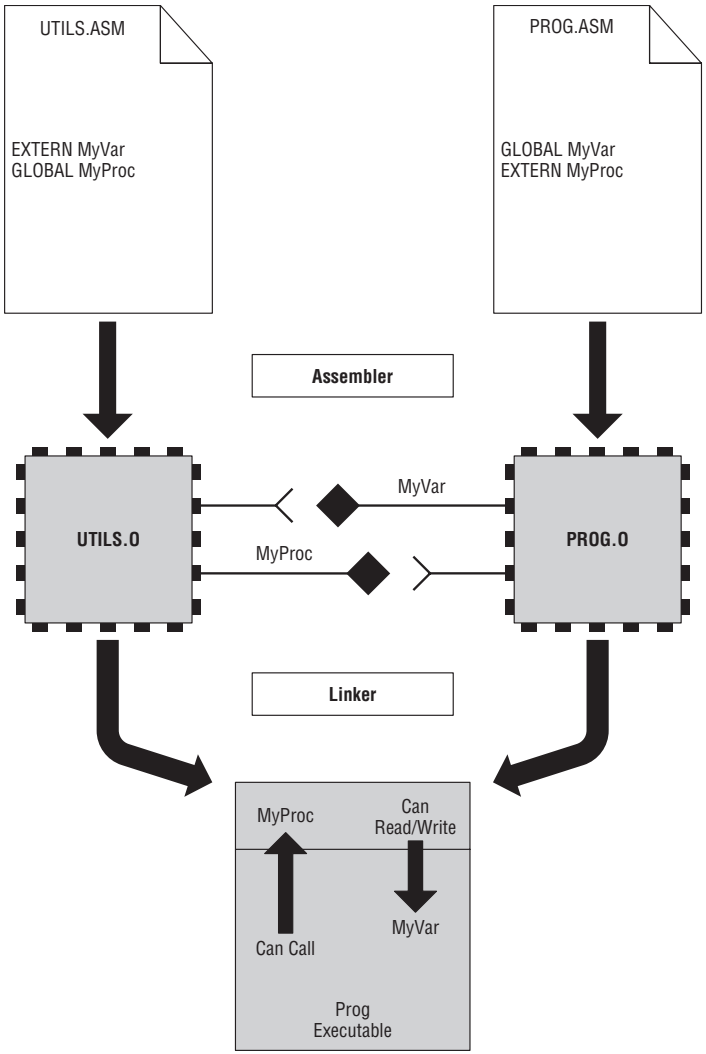

Figure 10-3: Connecting globals and externals

Create a Library

- **Definition of a library**
  - File .asm containing all sections (`.bss`, `.data`, `.text`)
  - Do not contain any `_start` (only one start per executable)
  - All public procedures or variables must be declared ``GLOBAL''
- **Example**

```
SECTION .bss    ; Section containing uninitialized data
        BUFFLEN EQU 10
        Buff    resb BUFFLEN

SECTION .data       ; Section containing initialised data

GLOBAL  DumpLin, HexDigits, BinDigits  ;Data items
GLOBAL  DumpLength, ASCLength, FullLength, DUMPLEN  ;Equate exports

DumpLin:    db " 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "
DUMPLEN     EQU $-DumpLin
.....                   ; Definition and initialization of all variables

SECTION .text    ; Section containing code

GLOBAL  ClearLine, DumpChar, Newlines, PrintLine  ;Procedures

ClearLine:
        ....    ; We do the job for this procedure
        ret     ; Go home
...
```

Use a library

- **In the calling library**
  - Define each external label (procedures and variables) as external
  - Use them as if they were in the same file
  - Has to respect the interface published in the comments of the library
  - Parameters for input and / or output
  - ...
- **Example**

```
...
SECTION .text    ; Section containing code
EXTERN ClearLine, DumpChar, PrintLine, DUMPLEN
GLOBAL _start
_start:
        nop  ; This no-op keeps gdb happy...
....
        mov edx,esi ; Copy total counter into EDX
        and edx,0000000Fh ; Mask lowest 4 bits
        call DumpChar; Call the procedure
```

Linking libraries into your programs

- **Makefile for programs seen previousely**
  - Contain only one single .asm file

```
hexdump2: hexdump2.o
        ld -o hexdump2 hexdump2.o
hexdump2.o: hexdump2.asm
        nasm -f elf64 -g -F stabs ↘
        →hexdump2.asm
```

- **Makefile using a library**

```
hexdump3: hexdump3.o textlib.o
        ld -o hexdump3 hexdump3.o textlib.o
hexdump3.o: hexdump3.asm
        nasm -f elf64 -g -F stabs ↘
        →hexdump3.asm
textlib.o: textlib.asm
        nasm -f elf64 -g -F stabs ↘
        →textlib.asm
```

Macros

Macros

- **Procedures**
  - jump to where the procedures are in memory
- **Macro**
  - The code is copied INSIDE the calling code
  - Equivalent to a copy / paste
- **Advantages of Macros**
  - Can uses fewer instructions, no use of stack
  - Call and return without jump and manipulation of Instuction Pointer
- **Problems with Macros**
  - Possibly larger code size as code is replicated

Macros, Syntax

- **Definition of a macro**

```
%macro  ExitProg 0
   mov eax,1   ; Code for Exit Syscall
   mov ebx,0   ; Return a code of zero
   int 80H       ; Make kernel call
%endmacro
```

- **Use of the macro**

```
_start:
   nop    ; This no-op keeps gdb happy...
; First we clear the terminal display...
    ClrScr
; Then we post the ad message centered on the console:
   WriteCtr 12,AdMsg,ADLEN
; Position the cursor for the "Press Enter" prompt:
   GotoXY 1,23
; Display the "Press Enter" prompt:
   WriteStr Prompt,PROMPTLEN
; Wait for the user to press Enter:
   WaitEnter
; ...and we are done!
   ExitProg
```

Macros with parameters

- **Parameters will be replicated inside the code %1 for the first parameter, ...**
- **Usage: write the parameters separated with columns ","**

```
; Then we post the ad message centered on the 80-wide ↘
→console:
        WriteCtr 12,AdMsg,ADLEN
```

- **Definition**

```
%macro WriteCtr 3   ; %1 = row; %2 = String addr; %3 = ↘
→String length
   push ebx         ; Save caller s EBX
   push edx         ; Save caller s EDX
   mov edx,%3       ; Load string length into EDX
   xor ebx,ebx      ; Zero EBX
   mov bl,SCRWIDTH  ; Load the screen width value to BL
   sub bl,dl   ; diff. screen width and string length
   shr bl,1         ; Divide difference by two for X value
   GotoXY bl,%1     ; Position the cursor for display
   WriteStr %2,%3   ; Write the string to the console
   pop edx                  ; Restore caller s EDX
   pop ebx                  ; Restore caller s EBX
%endmacro
```

Labels inside a Macro

- **Labels inside a Macro can only be accessed from within the macro**
  - start with %%

```
%macro UpCase 2    ; %1 = Address of buffer, %2 = ↘
→Chars in buffer
        mov  edx, %1  ; Place the offset of the ↘
        →buffer into edx
        mov ecx, %2   ; Place the number of bytes ↘
        →in the buffer into ECX
%%IsLC: cmp byte [edx+ecx-1], 'a'    ; Below 'a'?
        jb %%Bump       ; Not lowercase, Skip
        cmp byte [edx+ecx-1], 'z'          ; Above '↘
        →z'?
       ja %%Bump       ; Not lowercase, Skip
       sub byte [edx+ecx-1], 20h    ; Force byte in↘
       → buffer to uppercase
%%Bump: dec ecx      ; decrement character count
         jnz %%IsLC
%endmacro
```

Macros vs. Procedures

- **Pro Macros**
  - CALL and RET take time, code size and stack space
- **Cons**
  - Increased the size of the code, increasing chance of cache misses which may take more time

Which one is faster depends on application and CPU model!

How Macros Work

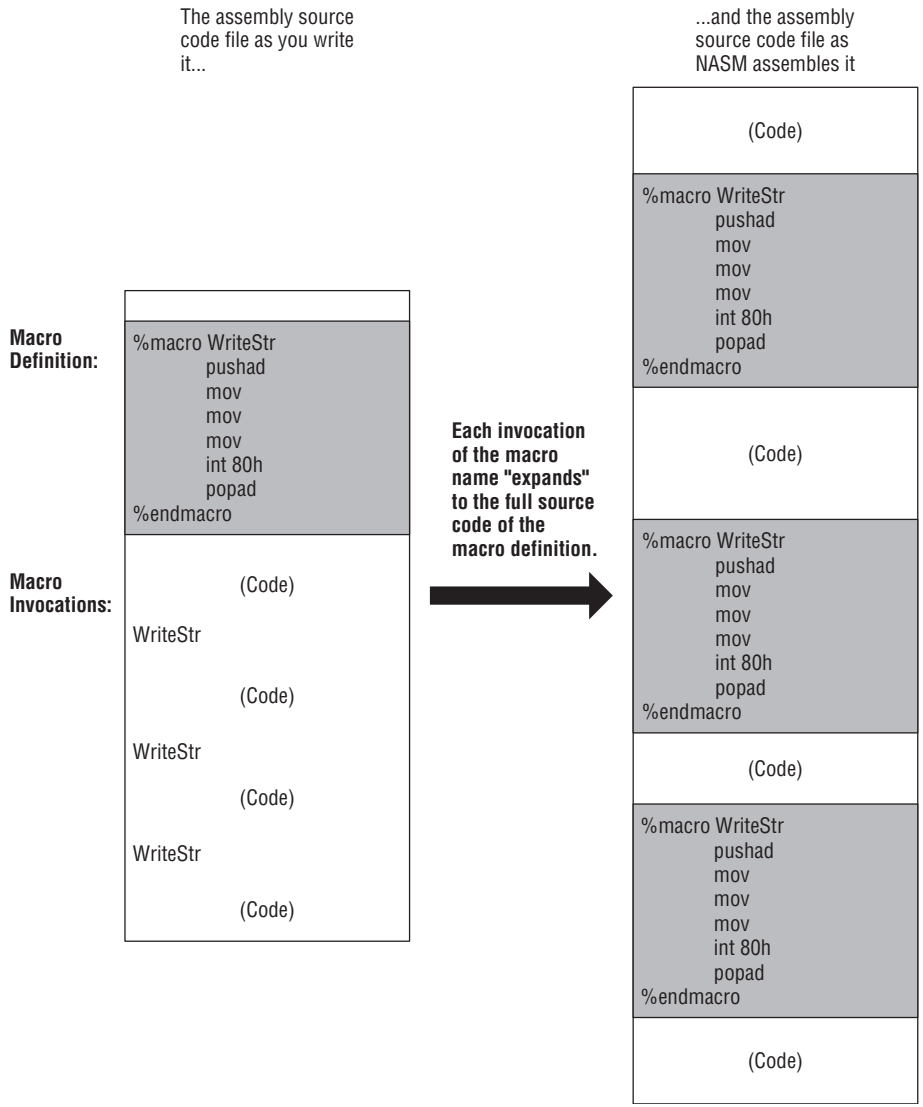

Figure 10-4: How macros work

Conclusion

- **Procedures**
  - Usefull to structure a program
  - Give possiblity to have some "private" labels (e.g. `.poke`)
  - Allow a sort of structured programming
  - Require the programmer to save registers before using them
- **Example: hexdump2**
  - Structured in procedures
  - Shows the content of a binary file
- **Libraries**
  - To store procedures you want to reuse
  - Test and debug once, reuse many times
- **Macros**
  - Are directly replaced inside the code (like copy and paste)
  - Efficient, but needs more place in executable

Bibliography

- This course corresponds to chapter *10* of the course book: **Assembly Language Step by Step (3rd Edition)**