System Programming

PC Assembly Language

H. Turgut Uyar   Şima Uyar

2001-2010

Topics

PC Assembly
- Introduction
- System Calls

Assembly and C
- Subroutines
- Calling Conventions
- C from Assembly
- Assembly from C

Linkers and Loaders
- Introduction
- Address Binding
- Two-Pass Linking

Directives

- needed by the assembler
- not part of the instruction set

- labels
  - mark points in code and data
  - entry labels have to marked **global**
- segments
- data definition
- named constants: **equ**
  - no memory allocated

Segments

Template

```
segment .data
; initialized data definitions

segment .bss
; uninitialized data definitions

segment .text
global _start
...
_start:
  ; entry point
  ...
```

Data Definition

type	*initialized*	*uninitialized*
byte	**db**	**resb**
word	**dw**	**resw**
dword	**dd**	**resd**
qword	**dq**	**resq**
tword	**dt**	**rest**

Addressing Issues

- plain label:
  address of data

Example

```
mov eax,L1
```

- label in brackets:
  data at address

Example

```
mov eax,[L1]
mov ebx,[eax]
```

Addressing Issues

- not allowed to have both operands in memory
- operands must be of the same size

Example

- the following instructions are incorrect:

```
mov [L8] ,[L1]

mov ax , bl
```

Software Interrupt

- system calls are implemented using software interrupt `80h`
- to make a system call:
  - `eax` ← number of system call
  - `ebx` ← first argument
  - `ecx` ← second argument
  - `edx` ← third argument
  - **int** 80h

exit System Call

- system call number: 1
- first argument: return status
  - 0: success
  - 1: failure

read System Call

- system call number: 3
- first argument: input descriptor
- second argument: start of input buffer
- third argument: length of input

write System Call

- system call number: 4
- first argument: output descriptor
- second argument: start of output buffer
- third argument: length of output

Descriptors

- 0: standard input
- 1: standard output
- 2: standard error

System Call Example

Example (Hello world)

```
segment .data
msg db   "Hello, world!",10
len equ $ - msg

segment .text
global _start


 _start:
     mov   eax,4
     mov   ebx,1
     mov   ecx,msg                    mov   eax,1
     mov   edx,len                    mov   ebx,0
     int   80h                        int   80h
```

References

Required: Carter

- Chapter 1: Introduction
    - 1.2. Computer Organization
    - 1.3. Assembly Language

Stack

- accessed in 4-byte units

push operand

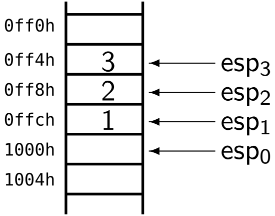
- subtract 4 from esp
- store operand to address [esp]

pop register

- store operand at address [esp] to register
- add 4 to esp

Stack Example

Example



```
push  dword  1
push  dword  2
push  dword  3
pop   eax
pop   ebx
pop   ecx
```

Subroutine Call

call target

- push address of next instruction
- jump to target

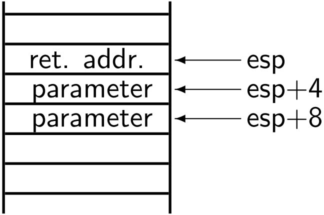
ret

- pop return address
- jump to return address

Stack Parameters

- called subroutine does not pop parameters
  - accesses parameters on the stack

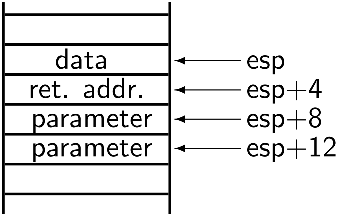
stack layout



Accessing Parameters

- offsets from esp may change

Example (after a push)



data ← esp

ret. addr. ← esp+4

parameter ← esp+8

parameter ← esp+12

Accessing Parameters

- use ebp

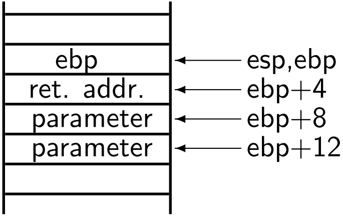
subroutine template

```
push  ebp
mov   ebp, esp

...

pop   ebp
ret
```

stack layout



Subroutine Example

Example (Factorial)

```
segment .bss
f       resd 1

segment .text

fact:
        push ebp
        mov  ebp,esp

        mov  dword [f],1
        mov  ecx,[ebp+8]
back:
        mov  eax,[f]
        mul  ecx
        mov  [f],eax
        dec  ecx
        cmp  ecx,1
        jne  back

        pop  ebp
        ret
```

Subroutine Example

Example (Calling Factorial)

```
segment .data
k   dd   5

segment .bss
f   resd 1

segment .text
global _start

fact:
    ...
```

```
_start:
    push ebp
    mov  ebp,esp

    push dword [k]
    call fact
    add  esp,4

    pop  ebp
    ret
```

Calling Conventions

- how will parameters be passed?
- if using stack:
  - in what order will the parameters be pushed?
  - who will remove parameters from the stack?
- how will the result be returned?
- which registers should remain unchanged?

C Calling Conventions

- parameters are passed via the stack
  - caller pushes parameters in reverse order
  - caller removes parameters from the stack
- result is returned over eax
- ebx,esi,edi,ebp,cs,ds,ss,es should remain unchanged

Calling C from Assembly

- to call a C function from Assembly:
  - declare function as **extern**
  - push arguments in reverse order
  - call function
  - adjust esp

C from Assembly Example

Example (Printing Factorial)

```
segment .data
k       dd    5
intf    db    "%d",10,0

segment .bss
f       resd  1

segment .text
global  main
extern  printf

fact:
        ...
```

```
main:
        ...

        push  dword [k]
        call  fact
        add   esp,4

        push  dword [f]
        push  intf
        call  printf
        add   esp,8

        ...
```

C Variables

- global: in fixed memory locations
- static: same as global, only scope is different
- automatic: on stack
- register: in a register (if possible)
- volatile: do not optimize

Automatic Variables

- allocation is done by subtracting from esp

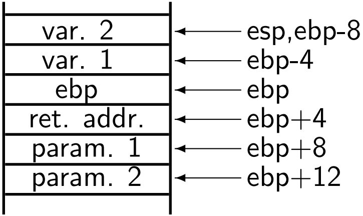
subroutine template

```
push ebp
mov  ebp,esp
sub  esp,BYTES

...

mov  esp,ebp
pop  ebp
ret
```

stack layout



var. 2	esp,ebp-8
var. 1	ebp-4
ebp	ebp
ret. addr.	ebp+4
param. 1	ebp+8
param. 2	ebp+12

Function Example

Example (Factorial (C))

```
int y;

void fact(int k)
{
    register int i;

    y = 1;
    for (i = k; i > 1; i--)
        y = y * i;
}
```

Function Example

Example (Factorial (C))

```
int fact(int k)
{
    int y;
    register int i;

    y = 1;
    for (i = k; i > 1; i--)
        y = y * i;
    return y;
}
```

Function Example

Example (Factorial)

```
segment .text
global  fact

fact:
   push ebp
   mov  ebp,esp
   sub  esp,4

   mov  dword [ebp-4],1
   mov  ecx,[ebp+8]
```

```
back:
   mov  eax,[ebp-4]
   mul  ecx
   mov  [ebp-4],eax
   dec  ecx
   cmp  ecx,1
   jne  back

   mov  eax,[ebp-4]
   mov  esp,ebp
   pop  ebp
   ret
```

Function Example

Example (Recursive Factorial (C))

```c
int fact(int k)
{
    if (k == 1)
        return 1;
    else
        return k * fact(k - 1);
}
```

Function Example

Example (Recursive Factorial)

```
fact:
    push  ebp
    mov   ebp,esp

    mov   eax,1
    mov   ecx,[ebp+8]
    cmp   ecx,1
    je    end_rec

    push  ecx
    dec   ecx
    push  ecx
    call  fact
    add   esp,4

    pop   ecx
    mul   ecx

end_rec:
    pop   ebp
    ret
```

Calling Assembly from C

- to call an Assembly function from C:
  - in Assembly file: declare function as **global**
  - in C file: declare the prototype

Function Example

Example (Calling Factorial)

```
int fact(int k);

int main(void)
{
    int x, y;

    ...
    y = fact(x);
    ...
}
```

References

Primary Text: Carter

- Chapter 4: Subprograms

Basic Functions

- binding abstract names to concrete names
    - easier to write code using abstract names
- related but conceptually different actions:
    - symbol resolution
    - relocation
    - program loading

Symbol Resolution

- references between subprograms are made using *symbols*
- linker
  - notes the location assigned to the called subprogram
  - patches the caller's object code

Example (`main` calls `sqrt`)

- linker finds location assigned to `sqrt` in the math library
- patches the object code of `main` so the call refers to that location

Relocation

- compiler generated object code starts at address 0
  - subprograms have to be loaded at non-overlapping addresses
- linker creates output starting at address 0
  - subprograms relocated within the big program
- loader picks the actual load address
  - linked program relocated as a whole

Program Loading

- loader copies program from secondary storage to memory
  - copy data from disk to memory
  - allocate storage
  - set protection bits
  - arrange for virtual memory

Address Binding

- early computers were programmed in machine language
  - write code on paper
  - assemble by hand
- symbols were bound to addresses:
  - by the programmer
  - at the time of translation

Address Binding

- if an instruction had to be inserted or deleted:
  - inspect the whole program
  - change affected addresses
- names bound to addresses too early

Assemblers

- programmers use symbolic names
    - assemblers bind names to addresses
- if program changes → reassemble
- the work of assigning addresses is pushed from the programmer to the assembler

Operating Systems

- before operating systems:
    - every process can access the entire memory
    - assemble and link for fixed memory addresses
- after operating systems:
    - processes share memory
    - actual addresses aren't known until program is loaded
    - final address binding deferred past link time to load time

Linker-Loader Separation

- linker does part of address binding
  - assigns relative addresses within each program
- loader does a final relocation
  - assigns actual addresses

Multitasking

- multiple programs run at the same time
- frequently multiple copies of the same program
  - some parts of the program are the same among all instances
  - other parts are unique to each instance
- separate changing parts from unchanging parts
  - use single copy of unchanging parts

Multitasking

- compilers were modified to generate object code in multiple sections
  - one section for read-only code
  - another for writable data
- linkers had to combine sections of each type
  - combine code sections to produce a code section
  - combine data sections to produce a data section

Libraries

- even different programs share common code
  - library functions
- modern systems provide shared libraries
  - all programs that use a library can share a single copy
  - better performance, less resources

Static Shared Libraries

- addresses are bound when the library is built
  - linker binds references to these addresses
- very inflexible
  - if any part of library changes → relink all programs

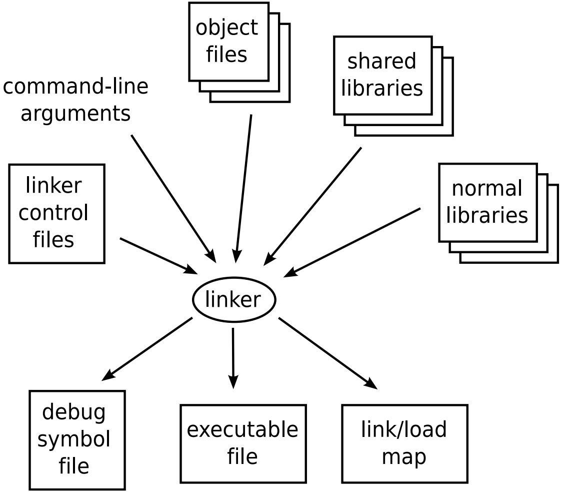
Dynamic Shared Libraries

- library symbols are bound when program starts running
    - linker binds references to these addresses
- can be delayed even farther:
    - at the time of the first call
- programs can bind to libraries at runtime
    - load libraries at runtime

Two-Pass Linking

- input: a set of object files and libraries
  - each input file contains segments
- output: executable or object code
  - load map, debugger symbols, ...

Two-Pass Linking



Symbol Table

- each input file contains a symbol table
- exported symbols
  - defined within the file for use in other files
  - names of subprograms within the file that can be called from elsewhere
- imported symbols
  - used in the file but defined elsewhere
  - names of subprograms called but not present in the file

First Pass

- scan input files:
  - find sizes of segments
  - collect references and definitions of all symbols
- create:
  - *segment table*: all segments defined in input files
  - *symbol table*: all imported and exported symbols

Second Pass

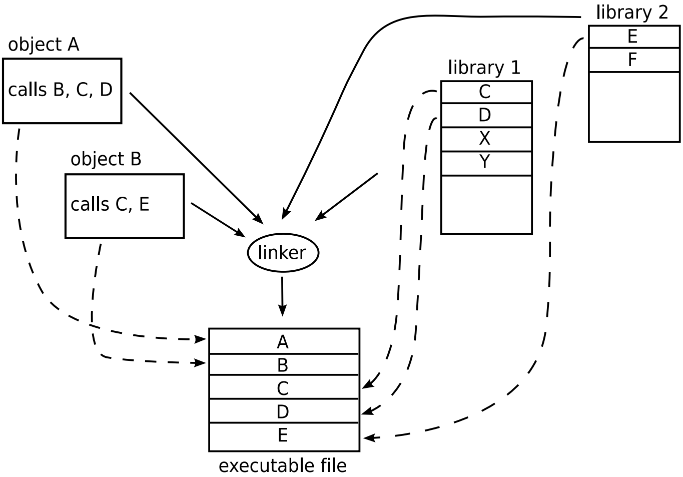
- assign numeric locations to symbols
- determine size and location of segments in output
- substitute numeric addresses for symbol references
    - adjust memory addresses in code and data to reflect relocated addresses

Linking Libraries

- library: collection of object code
- when resolving symbols:
  - process all regular input files
  - if any imported symbols are still missing:
    link in any library that exports the symbol

Linking Libraries

Example



Linking Shared Libraries

- linker identifies the shared libraries that resolve the undefined names
- rather than linking, it notes the libraries
- shared library is bound when program is loaded

References

Primary Text: Levine

- Chapter 1: Linking and Loading
- Chapter 3: Object Files