# Miscellaneous Programming Issues

This section contains:

- High-level language support
- Co-routines and introduction to processes
- The process of assembly and linking
- Monitors and debuggers

## High-level language support

In a high level language, we have storage types:

- Global variables
- Local variables
- Function/procedure arguments/parameters
- Returned values
- Dynamically allocated data

Global variables: each variable can have a constant address.

```
int x = 40;
```

In assembly language:

```
X   DD    40
```

Address of X in memory - determined by the location counter at the time the line is processed.

Local values and parameters: need special names to avoid collision.

```
x=foo(x,&y);
foo(int a, int * b){
        int c = *b;
        return (c);
}
```

Can use names: foo_a, foo_b, foo_c

Like return address at constant address - recursion impossible

Most languages have **activation frames**

Activation frame allocated per function activation.

Activation frame contains:

- Return address
- Other machine state (flags)
- Function arguments
- Local variables
- In some languages, nested scope

Simple solution - activation frame on stack

## Calling conventions

Scheme for activating a procedure is called a **calling convention**.

For assembly language can use:

- Arguments in agreed registers
- Returned values in registers (or even flags!)

Advantage: in many cases optimal speed.

Disadvantages:

- Hard to generalize
- Convention non-portable

Intermediate solution: arguments on stack, returned value in registers.

Advantages:

- Reasonably fast
- Reasonably portable
- Works for HLL with single return value

In fact, used by most C compilers

## Calling convention for C

Simplest general HLL convention - only call by value.

Allows for variable number of arguments.

Push arguments in reverse order.

Result: first argument always nearest TOS

Function expecting only k arguments but called with n greater than k, works OK, without even being aware of the extra
arguments!

To support nargs, push number of arguments after left-most argument.

Returned value in registers:

- (Intel 80X86) - AL, AX, or EAX.
- (Motorola 680X0) - D0 for data, A0 for pointer
- (VAX) - r0

Calling function cleanup after return - add value to SP

Code for calling the function:

```
sub     esp, 8    ; stack align
push    dword y
push    dword [x]
call    foo
add     esp, 16
mov     [x], eax
```

## Called function in C

Saves registers used in function

Local variables in registers + stack

If available, BP register simplifies access

Intermediate variables also on stack.

To return, transfer value to result registers, restore used registers, use RET.

Local variables “disappear” (though still on stack)

# Called function code for C

```
foo:
   push    ebp
   move    ebp, esp
   sub     esp, lo          ; lo=size of locals
   push    ebx              ; push some registers

   mov     ebx, [ebp+12]  ; get second arg
   mov     ebx, [ebx]      ; dereference
   mov     [ebp-4], ebx   ; initialize c local
     ...                   ; function code

   mov     eax, ...         ; return value
   pop     ebx              ; pop registers
   mov     esp, ebp
   pop     ebp              ; "leave" instr.
   ret
```

# Other High Level Languages

Common example - PASCAL

Different argument categories:
var arguments

Can be implemented by passing pointers

Push starting with left-most

Called procedure can clean up:
(Intel 80X86): RET n

Other parameter passing:

- Optional arguments
- Keyword arguments

# HLL support instructions

## Motorola 680X0

LINK An, #d ; Push An, move SP to An, subtract d from SP.

UNLK ; Move An to SP, pop An.

## Intel 80486

BOUND reg, addr
Compare reg against bounds
INT 5 if out of bounds
addr and addr+2 (or 4) contain bounds

ENTER framesize, level

LEAVE

# Co-routines and "Processes"

```
Coroutine1::
   DoSomeWork();
   Resume(Coroutine2);
   DoSomeMoreWork();
   Resume(Coroutine2);
   exit();

Coroutine2::
   DoSomeWork();
   Resume(Coroutine1);
   DoSomeMoreWork2();
   Resume(Coroutine1);
   exit();
```

## Using several stacks

Some processors have multiple SPs

Motorola 680X0: USP, ISP, MSP
Also, any An can be a SP

In general case can save SP, then re-load SP (Intel 80X86):

```
mov     [spsave1], esp
mov     esp, [spsave2]
```

## State of computation (process)

For an executing program, state is:

- All registers (including IP, SP, PSW)
- Local variables and arguments
- Other variables (global)
- Other state (files, devices)

If all state is saved, program can be suspended and then resumed without adverse effect.

We ignore, for now, global variables and IO state

State = all registers + the stack

Code that can run independent of other code (including copies of itself) is called **re-entrant**

## Reentrant code

Does not change global variables and IO state

Local variables and other local state **separate** for each activation

Using stack for activation frame, code that changes only local variables is **reentrant**

As a special case, reentrant code supports recursion

# Implementing co-routines

Each co-routine has its own stack.

Co-routines are initialized, then can be SUSPENDED and RESUMED at any point. (Synonyms: co-init and co-call)

Co-routines can call procedures normally.

Keep a struct for each co-routine, with:

- Initial entry point
- Stack pointer
- (Optionally) base pointer
- (Optionally) initialization flag
- Actual stack

# Data structure for coroutines

```
numco:   dd      3
CORS:    dd      CO1
         dd      CO2
         dd      CO3

 STKSZ    equu     16*1024
 CODEP    equ      0  ; constant offsets
 FLAGSP   equ      4
 SPP      equ      8

; Structure for first co-routine
CO1:     dd      CO1code
Flags1:  dd      0
SP1:     dd      STK1+STKSZ

STK1:    resb    STKSZ
```

# Code for CO-INIT

; Assuming EBX is pointer to COn

```
co_init:
  pusha
  bts     dword [EBX+FLAGSP],0 ; initialized?
  jc      init_done
  mov     EAX,[EBX+CODEP] ; Get initial IP
  mov     [SPT], ESP
  mov     ESP,[EBX+SPP]   ; Get initial SP
  mov     EBP, ESP        ; Also use as EBP
  push    EAX ; Push initial "return" address
  pushf       ; and flags
  pusha       ; and all other regs
  mov     [EBX+SPP],ESP   ; Save new SP
  mov     ESP, [SPT]      ; Restore old SP
init_done:
  popa
  ret
```

## Code for CO-CALL (RESUME)

EBX: pointer to co-init struct of co-routine to be resumed.

CURR: pointer to co-init structure of the curent co-routine.

```
resume:
  pushf     ; Save state of caller
  pusha
  mov   EDX, [CURR]
  mov   [EDX+SPP],ESP  ; Save current SP
do_resume:
  mov   ESP, [EBX+SPP] ; Load SP (resumed co)
  mov   [CURR], EBX
  popa   ; Restore resumed co-routine state
  popf
  ret    ; "return" to resumed co-routine!
```

## Process of Assembly and Linking

This section covers the following issues:

1. Macros (outline)
2. Assembly: pass I
3. Assembly: pass II
4. Address fixup tables,
   relocatable object files
5. The linking process and executable files
6. Libraries, dynamic linking

# Macros (outline)

A macro is a re-write rule

A user makes **definitions** and then **uses** the definitions

Macro processor processes data, substitutes data based on definitions

Macro processors exist in:

- Editors, word processors, and other user interfaces
- Script languages,
  stand-alone macro processors (m4)
- High level languages
  (example - C pre-processor)
- Macro-assemblers (example - NASM, MASM)

## Macro definition and expansion

A macro is like a **compile-time** function!

It is first **defined**, then can be used.

When a defined macro name is seen by macro processor, it is **expanded**

```
%define STK_UNIT        4

%macro  Iamalive        0
        push    Str1
        call    printf
        add     esp, STK_UNIT
%endmacro

section .rodata
Str1:   db      'I am alive', 10, 0
```

The code: Iamalive
becomes after expansion:

```
push    Str1
call    printf
add     esp, 4
```

# Macros with parameters

```
%macro  my_printf       2
section .rodata
Str2: db %2 , 10, 0
section .text
        push    %1
        push    Str2
        call    printf
        add     esp, STK_UNIT*2
%endmacro
```

To use the macro in a program:

```
my_printf       35, "The number is %ld"
```

Expanded into:

```
section .rodata
Str2: db "The number is %ld", 10, 0
section .text
        push    35
        push    Str2
        call    printf
        add     esp, 8
```

**Problem**: next use of macro will cause multiple definition of label - need a **local** label.

```
%macro  my_printf       2
section .rodata
%%Str2: db %2 , 10, 0
section .text
        push    %1
        push    %%Str2
        call    printf
        add     esp, STK_UNIT*2
%endmacro
```

# Assembler - Pass I

1. Open input file and temporary file
2. Initialize symbol table, location counter
3. Scan input file while:
   (a) Perform macro processing
   (b) Obey directives: org, db, section, etc.
   (c) Add symbols to symbol table
   (d) Translate instructions
   (e) Continually update location counter
   (f) Save locations where labels used
   (g) Write translated code into temp file
   (h) Find and list errors
4. Write final symbol table and relocation table (fixup table) into temp, close files.

## Assembler - Pass II

1. Open temporary, object, and listing files
2. Read symbol and fixup tables
3. Scan tranlated code, fix up addresses by using symbol table
4. Find and list errors (e.g. jump too far)
5. Generate listing
6. Write fixed code into relocatable
7. Write symbol table (publics) and fix-up table (externs) into relocatable

## Linking and Loading

**Linking** - a 2 pass operation. In **pass I**:
Open all relocatable objects and libraries
Resolve all externs - with other objects and libraries.
If no errors (unresolved externs, duplicates), create final symbol table (MAP).

In **pass II**, use fixup tables to fix all references in code to extern symbols.
Merge all fixed code (inluding some of library code) into a single executable file.
**Executable** contains:

- Code and initialized data
- Program entry point
- Size and location of code and data
- Optionally, symbol table (for symbolic debug)

**Loading**: read executable file, place code and data in appropriate memory locations.

# Monitors and Debuggers

Monitors allow for:

- Loading a program
- Viewing and modifying registers and memory data
- Running, single-stepping

Full debuggers allow, in addition:

- Disassembly
- Breakpoints
- Trace and watch
- Other source code viewing (symbolic debugger)

## Architecture support for debuggers

Some machines allow for:

- Single step mode
- Breakpoint interrupts (80486: INT 3)
- Trace and watch registers

Definition and use of breakpoints:

1. List code address of breakpoint
2. Save instruction of breakpoint location
3. Place a breakpoint interrupt instruction at breakpoint location.
4. When breakpoint interrupt occurs, restore original instruction, re-activate debugger user interface.

Trace and watch registers (80486): DR0-DR3 contain linear address.

Interrupt on execute, or write, or access, to address in any of DR0 to DR3.