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   DW    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(char a, int * b) {
        int c = *b;
}
```

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. 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 DX:AX.
  For pointer: BX, DS:BX
- (Motorola 680X0) - D0 for data,
  A0 for pointer
- (VAX) - r0

Calling function cleanup after return - add value to SP

Calling function code:

```
PUSH    b
PUSH    a     ; Push 16 bit word!
PUSH    2     ; Number of arguments
CALL    NEAR FOO
ADD     SP, 6
MOV     X, AX
```

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  PROC    NEAR
     PUSH    BP
     MOVE    BP, SP
     SUB     SP, lo        ; lo=size of locals
     PUSH    BX            ; Push some registers

     MOV     BX, [BP+8] ; Get second arg
     MOV     BX, [BX]   ; Dereference
     MOV     [BP-2], BX ; Initialize c local
     ...                ; Function code

     POP     BX            ; Pop registers
     ADD     SP, lo
     POP     BP
     MOV     AX, ...       ; Return value
     RET
FOO  ENDP
```

Other High Level Languages

Common example - PASCAL

Different argument categories:
var arguments

Can be implemented by passing pointers

Push starting with right-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, SP
MOV    SP, 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

Assuming tiny model (i.e. CS=SS=DS):

```
NUMCO   DW   3 ; number of coroutines
SPT     DW   0 ; Temporary SP copy
CORS    DW   OFFSET CO1
JNK1    DW   OFFSET CO2
JNK2    DW   OFFSET CO3

STKSZ   EQU  1024 ; or something
CODEP   EQU  0    ; constant offsets
FLAGSP  EQU  2
SPP     EQU  4
CO1     DW   OFFSET CO1CODE
FLAGS1  DW   0
SP1     DW   OFFSET STK1 + STKSZ
STK1    DW   STKSZ DUP (0)
```

Code for CO-INIT

Assuming BX has offset of COn:

```
COINIT  PROC  NEAR
  PUSHA
  BTS    [BX+FLAGSP], 0
  JC     IDONE
  MOV    AX, [BX+CODEP] ; Initial PC
  MOV    SPT, SP
  MOV    SP, [BX+SPP]   ; Initial SP
  MOV    BP, SP
  PUSH   AX ; Push COn PC
  PUSHF       ; and flags
  PUSHA       ; and other regs
  MOV    [BX+SPP], SP
  MOV    SP, SPT
IDONE:
  POPA
  RET
COINIT  ENDP
```

Code for CO-CALL (RESUME)

Assuming BX has offset of COn to resume:

```
RESUME   PROC  NEAR
  PUSHF  ; Save state of caller
  PUSHA
  BT     [BX+FLAGSP], 0
  JC     NOINIT
  CALL   NEAR  COINIT
NOINIT:
  MOV    SP, [BX+SPP] ; COn SP
  POPA   ; restore COn state
  POPF
  RET    ; returns to COn code
RESUME   ENDP
```

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 - 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**

```
PUSHAXBX  MACRO
          PUSH   AX
          PUSH   BX
          ENDM
```

The code: PUSHABBX
becomes after expansion:

```
          PUSH   AX
          PUSH   BX
```

Macros with parameters

```
PROMPT MACRO MESSGE
      MOVE  AH, 09H
      LEA   DX, MESSGE
      INT   21H
      ENDM
```

To use the macro in a program:

```
   .DATA
MSG1  DB  'Some message', '$'
   .CODE

   PROMPT MSG1
```

Also possible to use IF, REPT, INCLUDE...

Macro variables, unique labels.

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, 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 symbol use addresses into temp, close files. (fixup table)

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.