Chapter 11 -- Procedures

```
All about Procedures
--------------------

an introduction to procedures

why have procedures?
  -- reuse of code simplifies program writing
  -- modular code facilitates modification
  -- allows different programmers to write different parts of the
     same program
  -- etc.

Assembly languages typically provide little or no support for
  procedure implementation.

So, we get to build a mechanism for implementing procedures out
of what we already know.




First, some terms and what we need.

In Pascal:

      begin
        .
        .
        .
       x := larger(a, b);      CALL
        .
        .
        .
      end.
                  HEADER     PARAMETERS
      function larger (one, two: integer): integer;
      begin
         if ( one > two ) then
           larger := one             BODY
         else
           larger := two
      end;

In C:

      {
        .
        .
        .
       x = larger(a, b);      CALL
        .
        .
        .
      }
               HEADER     PARAMETERS
      int larger (int one, int two)
      {
         if ( one > two )
           larger = one;             BODY
         else
           larger = two;
      }
```

```
Steps in the execution of the procedure:
    1. save return address
    2. procedure call
    3. execute procedure
    4. return


   what is return address?        instruction following call

   what is procedure call?        jump or branch to first instruction
                                  in the procedure

   what is return?                jump or branch to return address


A possible Pentium implementation of procedure call:

             ;  one call
             lea  EAX, rtn_point
             jmp  proc1
   rtn_point:


             ;  another call
             lea  EAX, rtn_point2
             jmp  proc1
   rtn_point2:

   proc1:    ; 1st instruction of procedure here
             .
             .
             .
             jmp  (EAX)




This really doesn't work well if there are nested calls,
since we need a location to store the return address.
We CANNOT just use register EAX, as in the following
BAD code:

             ;  one call
             lea  EAX, rtn_point
             jmp  proc1
   rtn_point:


   proc1:    ; 1st instruction of procedure here
             .
             .
             .
             lea  EAX, rtn_point_nested
             jmp  proc2
   rtn_point_nested:
             .
             .
             .
             jmp  [EAX]    ; EAX will have been overwritten by the
                           ;   lea instruction in proc1.

   proc2:    ; 1st instruction of procedure here
             .
```

```
            .
            .
            jmp  [EAX]
```

What is needed to handle this problem is to have a way to save return addresses as they are generated. For a recursive subroutine, it is not known ahead of time how many times the subroutine will be called. This data is generated dynamically; while the program is running.

These return addresses will need to be used in the reverse order that they are saved.

The best way to save dynamically generated data is on a STACK.

Here is the code rewritten to use a stack:

```
   .data
   addr_stack  dd  100 dup(0)  ; hope that 100 addresses is enough!
   stack_ptr   dd  ?


            ;  stack initialization code
            lea  EDX, addr_stack
            mov  stack_ptr, EDX

            ;  one call
            lea  EAX, rtn_point
            jmp  proc1
   rtn_point:

   proc1:   ; 1st instruction of procedure here
            ; push return address on stack
            mov  [EDX], EAX
            add  EDX, 4
            .
            .
            .
            lea  EAX, rtn_point_nested
            jmp  proc2
   rtn_point_nested:
            .
            .
            .
            ; pop retn address off stack
            sub  EDX, 4
            mov  EAX, [EDX]
            jmp  [EAX]

   proc2:   ; 1st instruction of procedure here
            .
            .
            .
            jmp  [EAX]
```

SYSTEM STACK

A stack is so frequently used in implementing procedure call/return, that many computer systems predefine a stack, the SYSTEM STACK.

A stack handles dynamic data well.

```
 STATIC -- can be defined when program is written (compile time)
 DYNAMIC -- is defined when a program is executed (run time)

     In this case, it is the amount of storage that cannot be
     determined until run time.

 The size of the system stack is very large.  In theory, it should
 be infinitely large.  In practice, it must have a size limit.



 In memory, we have:

address  |         |
   0     | your    |
         | program |
         | here    |
         |         |
         |         |
         |         |
         |         |
         |         |
         | system  |  / \
 very    | stack   |   |  grows towards smaller addresses
large    | here    |   |
addresses


   terminology:

      Some people say that this stack grows DOWN in memory.
      This means that the stack grows towards smaller memory
      addresses.  Their picture would show address 0 at the
      bottom (unlike my picture).

      DOWN and UP are vague terms, unless you know what the
      picture looks like.

The Pentium stack is defined to grow towards smaller
addresses, and the stack pointer points to the full location
at the top of the stack.  The stack pointer is register ESP,
and it is defined before program execution begins (by the OS).



push, in Pentium:

   sub   esp, 4
   mov   [esp], ?

      OR

   mov   [esp - 4], ?  ; not a good implementation, since it
   sub   esp, 4        ; uses the space before allocation

pop, in Pentium:

   mov   ?, [esp]
   add   esp, 4


   NOTE:  If you use register esp for any use other than for
          a stack pointer, then the location of the stack is lost.
```

```
The use of the stack is SO common, that there are explicit
instruction to do the push and pop operations:

   push   r/m
          reg
          immed

   pop    reg

So, we would never use the 2-instruction sequence above,
only the push and pop instructions!


An example of using the system stack to save return addresses:

     lea  EAX, rtn1
     jmp  proc1
rtn1
     .
     .
     .
     lea  EAX, rtn2
     jmp  proc1
rtn2:
     .
     .
     .


proc1:  push  EAX            ; save return address

       .
       .
       .
       lea  EAX, rtn3       ; this would overwrite the return
       jmp  proc2           ; address if it had not been saved.
rtn3:
       .
       .
       .

        pop EAX              ; restore return address
        jmp [EAX]


proc2:
       .
       .
       .
       jmp  [EAX]



It is presumed that the code to call/return from procedures
will be well-used.  And, it is.

To make the compiler or assembly language programmer's job
easier, there are 2 instructions that do procedure call and
return.

   call   r/m       ; Push the return address onto the stack
                    ;  and jump to effective address given by
                    ;  the operand.
                    ; The return address is the address of the
                    ;  instruction following the call.

   ret    immed     ; Pop the return address off the stack
```

```
                    ;  and jump to that address.  Stack pointer
                    ;  (esp) is adjusted by the number of bytes
                    ;  given in the immediate operand.
                    ; This instruction can also be used with no
                    ;  operands.  It then defaults to only popping
                    ;  the return address off the stack.


The example again, using call and return instead of push/pop/lea/jmp.

     call  proc1
     .
     .
     .
     call  proc1
     .
     .
     .


proc1:
      .
      .
      .
      call  proc2
      .
      .
      .
      ret


proc2:
      .
      .
      .
      ret



about STACK FRAMES (ACTIVATION RECORDS)
---------------------------------------


From a compiler's point of view, there are a bunch of things
that should go on the stack relating to procedure call/return.
They include:
     return address
     parameters
     other various registers

Each procedure has different requirements for numbers of
  parameters, their size, and how many registers (which ones)
  will need to be saved on the stack.  So, we compose a
  STACK FRAME or ACTIVATION RECORD that is specific to a
  procedure.


Space for a stack frame gets placed on the stack each time
a procedure is called, and taken off the stack each time a
return occurs.  These stack frames are pushed/popped
DYNAMICALLY (while the program is running).


example:
  call  A
  call  B
  .
```

```
   .

A:  call C
    call D
    ret

B:  call D
    ret

C:  call E
    ret

D:  ret

E:  ret
```

show stack for a trace through this calling sequence

The code (skeleton) for one of these procedures:

```
A:  call C
    call D
    ret
```

becomes . . .

```
A:  sub esp, 20          ; allocate frame for A
                         ; A's return address is at [esp+20]
    call C
    call D

    add esp, 20          ; remove A's frame from stack
    ret
```

Some notes on this:
- the allocation and removal of a frame should be done within the body of the procedure. That way, the compiler does not need to know the size of a procedure's frame.
- Accesses to A's frame can be done via offsets from esp.

about frame pointers

The stack gets used for more than just pushing/popping stack frames. During the execution of a procedure, there may be a need for temporary storage of variables. The common example of this is in expression evaluation.

Example: high level language statement

```
Z = (X * Y) + (A/2) - 100
```

The intermediate values of X*Y and A/2 must be stored somewhere. On older machines, register space was at a premium. There just weren't enough registers to be used for this sort of thing. So, intermediate results (local variables) were stored on the stack.

They don't go in the stack frame of the executing procedure; they are pushed/popped onto the stack as needed.

So, at one point in a procedure, return address might be at [esp+16]

```
  |       |
  ---------
  |       |
  ---------
  |       | --- <- esp
  ---------    |
  |       |    |
  ---------    |
  |       |    |--- procedure's frame
  ---------    |
  |       |    |
  ---------    |
  |rtnaddr|    |
  --------- ---
  |       |
  ---------
```

and, at another point within the same procedure, it might be
at [esp+24]

```
  ---------
  |       |
  ---------
  | temp2 | <- esp
  ---------
  | temp1 |
  ---------
  |       | ---
  ---------    |
  |       |    |
  ---------    |
  |       |    |--- procedure's frame
  ---------    |
  |       |    |
  ---------    |
  |rtnaddr|    |
  --------- ---
  |       |
  ---------
```

All this is motivation for keeping an extra pointer around that does
not move with respect to the current stack frame.

Call it a FRAME POINTER. Make it point to the base of the current
frame:

```
  ---------
  |       |
  ---------
  | temp2 | <- esp
  ---------
  | temp1 |
  ---------
  |       | ---
  ---------    |
  |       |    |
  ---------    |
  |       |    |--- procedure's frame
  ---------    |
  |       |    |
```

```
  ---------   |
  |rtnaddr|   | <- frame pointer
  --------- ---
  |       |
  ---------
```

Now items within the frame can be accessed with offsets from the frame pointer, AND the offsets do not change within the procedure.

The return address will now be at frame pointer.

Pentium architecture has a register dedicated as a frame pointer. It is EBP. The last 2 letters stand for "base pointer".

Items within the the frame are accessed using negative (but fixed) offsets from EBP.

NOTE:
 -- EBP must be initialized at the start of every procedure,
 and restored at the end of every procedure.

The skeleton implementation of a procedure that uses a frame pointer:

```
A:  push ebp             ; save caller's frame pointer
    mov  ebp, esp        ; set up A's frame pointer
    sub esp, 16          ; allocate remainder of frame for A
                         ; A's return address is at [ebp+4]
    call C
    call D

    mov  esp, ebp        ; remove A's frame from stack
    pop  ebp             ; restore caller's frame pointer
    ret
```

parameter passing.
----

parameter = argument

Just as there is little/no support for implementing procedures in many assembly languages, there is little/no support for passing parameters to those procedures.

Remember, when it comes to the implementation,
 -- convention
 -- its up to the programmer to follow the conventions

Passing parameters means getting data into a place set aside for the parameters. Both the calling program and the procedure need to know where the parameters are.

The calling program places them there, and possibly uses values returned by the procedure. The procedure uses the parameters.

a note on parameter passing --

```
  a HLL specifies rules for passing parameters.  There are basically
  2 types of parameters.

  Note that a language can offer 1 or both types.


    call by value -- what C has.  In Pascal, these are parameters
        declared without the var in front of the variable name.
        Fortran doesn't have this type of parameter.

        The parameter passed may not be modified by the procedure.
        This can be implemented by passing a copy of the value.
        What call by value really implies is that the procedure can
        modify the value (copy) passed to it, but that the value
        is not changed outside the scope of the procedure.

    call by reference -- what Fortran has.  In Pascal, these are
        "var type" parameters.

        The parameter passed to the subroutine can be modified,
        and the modification is seen outside the scope of the
        subroutine.  It is sort of like having access to global
        variable.


There are many ways of implementing these 2 variable types.
If call by value is the only parameter type allowed, how
can we implement a reference type parameter?
   Pass the address of the variable as the parameter.  Then
   access to the variable is made through its address.  This
   is what is done in C.



Simplest parameter passing mechanism -- Use registers
-----------------------------------------------------

the calling program puts the parameter(s) into specific registers,
and the procedure uses them.

  example:

              .
              .
              .
        mov   EAX, [EDX]      ; put parameter in EAX
        call  decrement
        mov   [EDX], EAX      ; recopy parameter to its correct place.
              .
              .
              .
   decrement:
        sub  EAX, 1
        ret


Notes:   -- This is a trivial example, since the procedure is 1 line
            long.
         -- Why not just use [EDX] within the procedure?
                1. convention -- parameters are passed in specific registers.
                2. same procedure could be used to decrement the value
                   in other parameters -- just copy the value in before
                   the call, and copy it out afterwards.

         -- The Intel architectures suffer from not having enough
            registers.  With only a few to play around with (as
            general purpose registers), we VERY soon run out of
```

```
             registers to use.  So, this method of passing parameters
             is really not used for this architecture.  It IS used
             on all the more modern architectures.  They even go so
             far as to set aside a subset of their registers dedicated
             as a location for passing parameters.



historically more significant mechanism:  parameters on stack
                                          ---------------------
  (The method of choice for this architecture.)

place the parameters to a procedure (function) in the activation
record (AR) for the procedure.

       push  P1         ; place parameter 1 into AR of proc
       push  P2         ; place parameter 2 into AR of proc
       call  proc
       .
       .
       .
proc:  push ebp             ; save caller's frame pointer
       mov  ebp, esp        ; set up A's frame pointer
       sub esp, 16          ; allocate remainder of frame for A
                            ; A's return address is at [ebp+4]

       ; use parameters in procedure calculations
       ; P1 is at [ebp+12]
       ; P2 is at [ebp+8]

       mov  esp, ebp        ; remove A's frame from stack
       pop  ebp             ; restore caller's frame pointer
       ret  8               ; pop return address, return, and
                            ;  remove the parameters!



calling program:  pushes parameters into stack
                  calls procedure

procedure:        allocates AR (or remainder of AR)
                  deallocates AR of procedure



The activation record (frame) for a procedure so far:


                       ^ smaller addresses up here
  |-----------------|
  |                 |
  |-----------------|
  |                 |
  |-----------------|
  | caller's ebp    |<- ebp (AND possibly esp)
  |-----------------|
  | return address  |
  |-----------------|
  |       P2        |
  |-----------------|
  |       P1        |
  |-----------------|
  |                 |
  |-----------------|
  |                 |
```

```
New problem:
  What happens if a procedure has lots of variables, and it
  runs out of registers to put them in.  These are really the
  local variables of the procedure.

  Most common solution:  store local variables temporarily on the
                         stack in AR.


Two ways of implementing this:

CALLEE SAVED
   A procedure clears out some registers for its own use.

   The called procedure saves register values in its AR.


CALLER SAVED
   The calling program saves the registers and local variables
   that it does not want a called procedure to overwrite.




REVISITING PROCEDURES.


What needs to be done depends on HLL.
The order is fairly consistent.
What is done by caller/callee varies from implementation to implementation.


Needed:
 --> items to go in activation record.

   return address
   frame pointer (if used)
   parameters
   local variables  --|  may be some overlap here
   saved registers  --|


 --> mechanism

 before ANY procedure CALL
 1.  caller gets parameters into correct location
 then
 2.  control is transfered to procedure

 before procedure RETURN
 1.  put return values into correct location
 2.  restore anything that needs to be restored (return address, callee
     saved registers, frame pointer)
 3.  remove activation record
 then
 4.  jump to return location
```