Run Time Storage Organization

ASU Textbook Chapter 7.1–7.4, and 7.7–7.8

Tsan-sheng Hsu

tshsu@iis.sinica.edu.tw

http://www.iis.sinica.edu.tw/~tshsu
During the execution of a program, the same name in the source can denote different data objects in the computer. The allocation and deallocation of data objects is managed by the run-time support package.

Terminologies:
- name → storage space: the mapping of names to storage spaces is called an environment.
- storage space → value: the current value of a storage space is called its state.
- The association of a name to a storage location is called a binding.

Each execution of a procedure is called an activation.
- If it is a recursive procedure, then several of its activations may exist at the same time.
- Life time: the time between the first and last steps in a procedure.
- A recursive procedure needs not to call itself directly.
Activation record

- **Activation record**: data about an execution of a procedure.
  - **Parameters**:
    - **Formal parameters**: the declaration of parameters.
    - **Actual parameters**: the values of parameters for this activation.
  - **Links**:
    - **Control (or dynamic) link**: a pointer to the activation record of the caller.
    - **Access (or static) link**: a pointer to places of non-local data,

<table>
<thead>
<tr>
<th>returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual parameters</td>
</tr>
<tr>
<td>optional control link</td>
</tr>
<tr>
<td>optional access link</td>
</tr>
<tr>
<td>saved machine status</td>
</tr>
<tr>
<td>local data</td>
</tr>
<tr>
<td>temporaries</td>
</tr>
</tbody>
</table>
**General run time storage layout**

- **lower memory address**
  - **code**
  - **static data**
    - storage space for data that will not be changed during the execution: e.g., global data and constant, ...
  - **stack**
    - for activation records: local data, parameters, control info, ...
  - **heap**
    - for dynamic memory allocated by the program
- **higher memory address**
Issues in storage allocation

- There are two different approaches for run time storage allocation.
  - Static allocation.
  - Dynamic allocation.
- Need to worry about how variables are stored.
  - That is the management of activation records.
- Need to worry about how variables are accessed.
  - Global variables.
  - Locally declared variables, that is the ones allocated within the current activation record.
  - Non-local variables, that is the ones declared and allocated in other activation records and can be accesses here.

\[ \text{Non-local variables are different from global variables.} \]
Static storage allocation (1/3)

- **Static allocation:** uses no stack and heap.
  - **Strategies:**
    - For each procedure in the program, allocate a space for its activation record.
    - A.R.’s can be allocated in the static data area.
    - Names bound to locations at compiler time.
    - Every time a procedure is called, a name always refer to the same pre-assigned location.
  - Used by simple or early programming languages.

- **Disadvantages:**
  - No recursion.
  - Waste lots of space when inactive.
  - No dynamic allocation.

- **Advantages:**
  - No stack manipulation or indirect access to names, i.e., faster in accessing variables.
  - Values are retained from one procedure call to the next if block structure is not allowed.
    - For example: static variables in C.
On procedure calls,

- the calling procedure:
  - First evaluate arguments.
  - Copies arguments into parameter space in the A.R. of called procedure.
    Convention: call that which is passed to a procedure arguments from
    the calling side, and parameters from the called side.
  - May save some registers in its own A.R.
  - Jump and link: jump to the first instruction of called procedure and
    put address of next instruction (return address) into register RA (the
    return address register).

- the called procedure:
  - Copies return address from RA into its A.R.’s return address field.
  - control link := address of the previous A.R.
  - May save some registers.
  - May initialize local data.
On procedure returns,
- the called procedure:
  - Restores values of saved registers.
  - Jump to address in the return address field.
- the calling procedure:
  - May restore some registers.
  - If the called procedure is actually a function, that is the one that returns values, put the return value in the appropriate place.
Dynamic storage allocation for STACK (1/3)

- **Stack allocation:**
  - Each time a procedure is called, a new A.R. is pushed onto the stack.
  - A.R. is popped when procedure returns.
  - A register (SP for stack pointer) points to top of stack.
  - A register (FP for frame pointer) points to start of current A.R.
Dynamic storage allocation for STACK (2/3)

- On procedure calls,
  - the calling procedure:
    - May save some registers (in its own A.R.).
    - May set optional access link (push it onto stack).
    - Pushes parameters onto stack.
    - Jump and Link: jump to the first instruction of called procedure and put address of next instruction into register RA.
  - the called procedure:
    - Pushes return address in RA.
    - Pushes old FP (control link).
    - Sets new FP to old SP.
    - Sets new SP to be old SP + (size of parameters) + (size of RA) + (size of FP). (These sizes are computed at compile time.)
    - May save some registers.
    - Push local data (maybe push actual data if initialized or maybe just their sizes from SP)
On procedure returns,

- the called procedure:
  - Restore values of saved registers if needed.
  - Loads return address into special register RA.
  - Restores SP (SP := FP).
  - restore FP (FP := saved FP).
  - return.

- the calling procedure:
  - May restore some registers.
  - If it is in fact a function that was called, put the return value into the appropriate place.
Activation tree

- Use a tree structure to record the changing of the activation records.
- Example:

```c
main{
    r();
    q(1);
}

r{
    ...
}

q(int i)
{
    if(i>0) then q(i-1);
}
```

```
main
stack
main
stack
main
stack
main
stack
main
stack
main
```

Compiler notes #6, Tsan-sheng Hsu, IIS
Dynamic storage allocation for HEAP

- **Storages requested from programmers during execution:**
  - Example:
    - **PASCAL:** `new` and `free`.
    - **C:** `malloc` and `free`.
  - Issues:
    - *Garbage collection.*
    - *Segmentation.*
    - *Dangling reference.*

- **More or less O.S. issues.**
Run time variable accesses

- Global variables:
  - Access by using names.
  - Addresses known at compile time.

- Local variables:
  - Stored in the activation record of declaring procedure.
  - Access a local variable \( v \) in a procedure \( P \) by \( \text{offset}(v) \) from the frame pointer (FP).

  ▶ Let \( \text{local}\_\text{start}(P) \) be the amount of spaces used by data in the activation record of procedure \( P \) that are allocated before the local data area.
  ▶ The value \( \text{local}\_\text{start}(P) \) can be computed at compile time.
  ▶ The value \( \text{offset}(v) \) is the amount of spaces allocated to local variables declared before \( v \).
  ▶ The address of \( v \) is \( \text{FP} + \text{local}\_\text{start}(P) + \text{offset}(v) \).
  ▶ The actual address is only known at run time, depending on the value of FP.
int P()
{
    int I, J, K;
    ...
}

• **Address of** \( J \) **is** \( \text{FP} + \text{local\_start}(P) + \text{offset}(v) \).

• **offset**(\( v \)) **is** \( 1 \times \text{sizeof}(\text{int}) \) **and** **is known at compile time.**

• **local\_start**(\( P \)) **is known at compile time.**

• **Actual address is only known at run time, i.e., depends on the value of FP.**
Two scoping rules for accessing non-local data.

- Lexical or static scoping.
  - **PASCAL, C and FORTRAN.**
  - The correct address of a non-local name can be determined at compile time by checking the syntax.
  - Can be with or without block structures.
  - Can be with or without nested procedures.

- Dynamic scoping.
  - **LISP.**
  - A use of a non-local variable corresponds to the declaration in the “most recently called, still active” procedure.
  - The question of which non-local variable to use cannot be determined at compile time. It can only be determined at run-time.
Lexical scoping with block structures

- **Block**: a statement containing its own local data declaration.
- **Scoping** is given by the following so-called most closely nested rule.

  - The scope of a declaration in a block \( B \) includes \( B \) itself.
  - If \( x \) is used in \( B \), but not declared in \( B \), then we refer to \( x \) in a block \( B' \), where
    - \( B' \) has a declaration \( x \), and
    - \( B' \) is more closely nested around \( B \) than any other block with a declaration of \( x \).
Lexical scoping without nested procedures

- **Nested procedure**: a procedure that can be declared within another procedure.

- If a language does not allow nested procedures, then
  - a variable is either global, or is local to the procedure containing it;
  - at runtime, all the variables declared (including those in blocks) in a procedure are stored in its A.R., with possible overlapping;
  - during compiling, proper offset for each local data is calculated using information known from the block structure.
Lexical scoping with nested procedures (1/3)

- In a program with lexical scoping and nested procedures, what are the procedures that can be called in a given procedure \( Q_0 \)?
  - The procedure \( Q_1 \) who declares \( Q_0 \).
  - The procedure \( Q_i \) who declares \( Q_{i-1} \), \( i > 0 \).
  - The procedure \( P_i \) whom is declared together with, but before, \( Q_i \), \( i > 0 \).

- In a procedure declaration tree, \( Q_0 \) can call any procedure that is its direct ancestor or the older siblings of its direct ancestor.

- A procedure can only access the variables that is global in a procedure that is its direct ancestor.
  - When you call a procedure, a variable name follows the lexical scoping rule.
  - Use the access link to link to the procedure that is lexically enclosing the called procedure.
  - Need to set up the access link properly to access the right storage space.
Lexical scoping with nested procedures (2/3)

procedure main
  procedure a1
    procedure s1
  procedure a2
    procedure b1
      procedure q1
    procedure b2
  procedure c1
    procedure d1
    procedure d2
  procedure d3

Compiler notes #6, Tsan-sheng Hsu, IIS
Lexical scoping with nested procedures (3/3)

- **Nesting depth:**
  - depth of main program = 1.
  - add 1 to depth each time entering a nested procedure.
  - substrait 1 from depth each time existing from a nested procedure.
  - Each variable is associated with a nesting depth.
  - Assume in a depth-h procedure, we access a variable at depth k, then
    - h ≥ k.
    - follow the access (static) link h – k times, and then use the offset information to find the address.
Algorithm for setting the links

- The control link is set to point to the A.R. of the calling procedure.

- How to properly set the access link at compile time.
  - Procedure $p$ at depth $n_p$ calls procedure $x$ at depth $n_x$:
    - If $n_p < n_x$, then $x$ is enclosed in $p$ and $n_p = n_x - 1$.
      - Same with setting the control link.
    - If $n_p \geq n_x$, then it is either a recursive call or calling a previously declared procedure.
      - Observation: go up the access link once, then the depth is decreased by 1.
      - Hence, the access link of $x$ is the access link of $p$ going up $n_p - n_x + 1$ times.
Program sort
  var a: array[0..10] of int;
  x: int;
  procedure r
  var i: int;
  begin ... r
  end

  procedure e(i,j)
  begin ... e
    a[i] <-> a[j]
  end

  procedure q
  var k,v: int;
  procedure p
  var i,j;
  begin ... p
    call e
  end
  begin ... q
    call q or p
  end

  begin ... sort
    call q
  end
Accessing non-local data using DISPLAY

- **Idea:**
  - Maintain a global array called DISPLAY.
    - *Using registers if available.*
    - *Otherwise, stored in the static data area.*
  - When procedure $P$ at nesting depth $k$ is called,
    - DISPLAY[1], ..., DISPLAY[k-1] hold pointers to the A.R.’s of the most recent activation of the $k - 1$ procedures that lexically enclose $P$.
    - DISPLAY[k] holds pointer to $P$’s A.R.
    - To access a variable with declaration at depth $x$, use DISPLAY[$x$] to get to the A.R. that holds $x$, then use the usual offset to get $x$ itself.
    - Size of DISPLAY equals maximum nesting depth of procedures.
  - Bad for languages allow recursions.

- **To maintain the DISPLAY**
  - When a procedure at nesting depth $k$ is called
    - *Save the current value of DISPLAY[$k$] in the save-display field of the new A.R.*
    - *Set DISPLAY[$k$] to point to the new A.R., i.e., to its save-display field.*
  - When the procedure returns, restore DISPLAY[$k$] using the value saved in the save-display field.
Access links v.s. DISPLAY

- **Time and space trade-off.**
  - Access links require more time (at run time) to access non-local data. Especially when non-local data are many nesting levels away.
  - DISPLAY probably require more space (at run time).
  - Code generated using DISPLAY is simpler.
Dynamic scoping

- Dynamic scoping: a use of a non-local variable refers to the one declared in the “most recently called, still active” procedure.
- The question of which non-local variable to use cannot be determined at compile time.
- It can only be determined at run time.
- May need symbol tables at run time.
- Two ways to implement non-local accessing under dynamic scoping.
  - Deep access.
  - Shallow access.
Dynamic scoping – Example

program main
  procedure test
  var x : int;
  begin
    x := 30;
    call DeclaresX;
    call UsesX;
  end
  procedure DeclaresX
    var x: int;
    begin
      x := 100;
      call UsesX;
    end
  procedure UsesX
  begin
    write(x);
  end
begin
  call test;
end

- Which $x$ is it in the procedure UsesX?
- If we were to use static scoping, this is not a legal statement; No enclosing scope declares $x$. 
- **Def:** given a use of a non-local variable, use control links to search back in the stack for the most recent A.R. that contains space for that variable.
  - Note: this requires that to be possible to tell the set of variables stored in each A.R.
  - Need to use the symbol tables at run time.
Shallow access

- **Idea:**
  - Maintain a current list of variables.
  - Space is allocated (in registers or in the static data area) for every possible variable name that is in the program (i.e., one space for variable $x$ even if there are several declarations of $x$ in different procedures).
  - For every reference to $x$, the generated code refers to the same location.

- **When a procedure is called,**
  - it saves, in its own A.R., the current values of all of the variables that it declares itself (i.e., if it declares $x$ and $y$, then it saves the values of $x$ and $y$ that are currently in the space for $x$ and $y$).
  - It restores those values when it finishes.

- **Comparisons:**
  - Shallow access allows fast access to non-locals, but there is overhead on procedure entry and exit proportional to the number of local variables.
  - Deep access needs to use a symbol table at run time.