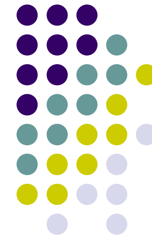
CSc 453
Runtime Environments

Saumya Debray
The University of Arizona
Tucson



Issues

- Managing the relationship between source program names and runtime data objects.
- Managing allocation/deallocation of, and access to, data objects at runtime
- Managing different activations of a procedure.

  In general, several different activations of a procedure may be "alive" at the same time.

Flow of Control Assumptions

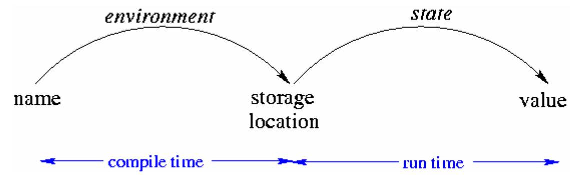
- Sequential control flow:

  At each step during execution, control is at some specific point in the program.
  - *no program level parallelism.*
- Procedure execution:
  - Each execution of a procedure starts at the beginning of the procedure body.
  - After the procedure has executed, control returns to the point immediately after the call site.
  - *no coroutining (Icon) or backtracking (Icon, Prolog).*

Procedure Activation Characteristics

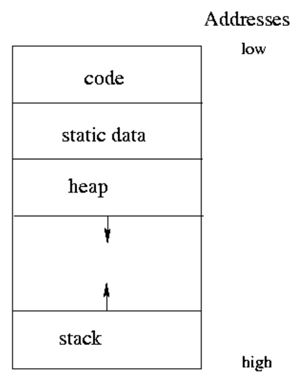
- Our procedure execution assumptions imply that the lifetimes of any two procedure activations are either nested or disjoint.
- This implies that procedure activations can be managed using a *control stack*:
  - *push* a node for an activation at entry to the procedure;
  - *pop* the node when returning from the procedure.

Bindings of Names



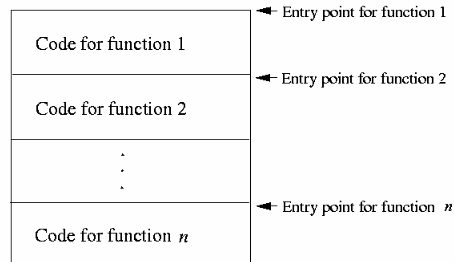
Runtime Memory Organization

Runtime memory is organized to hold the components of an executing program, e.g.:



Organization of Code Area

- Usually, code is generated a function at a time.



- Within a function, the compiler has freedom to organize the code in any way.

  Careful code layout can improve cache performance and increase speed.

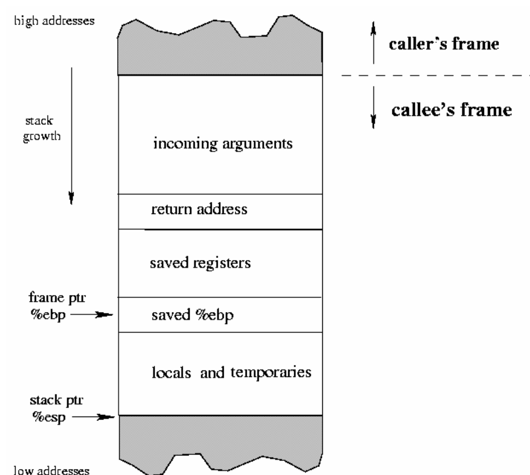
Activation Records

- An ***activation record*** contains information needed to manage a single activation of a procedure, e.g.:
  - saved machine state (PC, registers, return address);
  - actual parameter values;
  - local and temporary variables.
- The contents of an activation record may be spread across the stack frame and registers.

Activation Records: Layout

- Some aspects of activation record layout, e.g., location of actual parameters and some machine state info, are specified by the *calling convention*.
- The compiler decides the layout for local variables and temporaries:
  - the amount of storage needed for an object is determined by its type;
  - storage layout must conform to any *alignment restrictions* of the underlying architecture.

Example: Stack Frame for an x86



Activation Records: Allocation Strategies

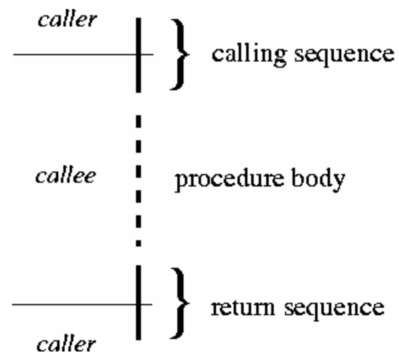
- *Static Allocation* (Fortran 77):
  - all storage allocated by the compiler;
  - no recursion, dynamic memory allocation;
- *Stack Allocation* (C, C++, Java):
  - activation records organized as a stack;
  - cannot be used if values of locals must be retained when an activation ends, or if a called invocation outlives the caller.
- *Heap Allocation* (Lisp, Scheme):
  - activation records allocated, deallocated in any order;
  - some form of garbage collection or compaction needed to reclaim space.

Procedure Calls and Returns

- *Calling sequence*: handles a call to a procedure:
  - loads actual parameters where callee can find them;
  - saves machine state (return address, …);
  - branches to callee;
  - allocates an activation record.
- *Return sequence*: handles the return from a procedure call:
  - loads the return value where the caller can find it;
  - deallocates the activation record;
  - restores machine state (saved registers, PC, etc.);
  - branches back to caller.

Procedure Calls and Returns: cont'd

Structure of code executed for a procedure call:



Calling Conventions

- A *calling convention* for an architecture and/or language specifies how values are communicated between procedures:
  - register usage (caller vs. callee saved registers, …);
  - argument and return value placement.

  E.g.: on the x86 [C calling convention]: an integer return value is placed in register **eax**.

- We can have multiple calling conventions, e.g.: __cdecl, __stdcall, __fastcall in MS Windows.

Caller vs. Callee Saved Registers

- A calling convention typically divides registers into two classes:
  - *caller-saved*: registers whose values will be overwritten by a function call;

    E.g.: On the x86 [C calling convention]: **ebx**, **ecx**, **edx** are caller-saved.
  - *callee-saved*: registers whose values will survive across a function call.

    E.g.: On the x86 [C calling convention]: **edi**, **esi**, **esp**, **ebp** are callee-saved.
- A function using a callee-saved register must save it on entry and restore it on exit.

Nested Functions

- Some languages allow function definitions to be *nested*.

  E.g.: GNU C allows definitions of the form

```
int foo(int x, int y)
{
  int bar(int x) { return x*y; }
  return bar(x) + bar(y);
}
```

- Nested functions are typically able to access variables in enclosing scopes.

  E.g.: the variable `y` above.

Accessing Non-Local Variables

Problem: In general, we may not know how far deep in the stack a variable in an enclosing scope may be.

E.g:

```
int p(int m)
{
  int x;    /* x is local to p, hence in p's activation record */
  int q(int n)
  {
    if (n > 0) return 2*q(n-1);
    else return x+1;
  }

  printf("%d\n", q(m+2));
}
```


Accessing Non-Local Variables

- Basic Idea: pass an access link at each call.

  A procedure p's *access link* is a pointer to the (most recent) activation record of the procedure q that encloses *p's definition*.

- The code to set up access links can be generated at compile time.
- Using access links, at runtime the program can "walk up" the static nesting structure to access non-local variables.

Accessing Non-Local Variables

- The *nesting depth* of a procedure:
  - The outermost scope (globals) has nesting depth 0.
  - Nesting depth increases by 1 when we enter a new scope, and decreases by 1 when we leave the scope.
  - Nesting depths are known at compile time.
- Suppose a procedure p at nesting depth n_p refers to a variable x at nesting depth n_x ($n_x \leq n_p$).

  The code generated is as follows:
  - follow access links $n_p - n_x$ times; (both n_p, n_x known at compile time);
  - access x within the stack frame reached. (x's offset known at compile time).