## Runtime Environment

**Topics we will cover**

- The procedure abstraction and linkage conventions
- Runtime storage convention
- Non-local data access (brief)

*These issues are critical to high-performance code generation*

**Topics not covered** :

- Garbage collection support
- Heap management
- Exception handling support
- Optimization for cache and TLB: covered next semester

*Nevertheless, these issues are important for performance.*





## The Procedure Abstraction and Separate Compilation

**Requires system-wide compact**

- must involve architecture, OS, and compiler
- broad agreement on memory layout, protection requirements, calling sequence, error handling and reporting

**Separate compilation**

- *Compilation strategy that allows subsets (files, modules, or directories) of a single program to be compiled separately and then linked together*
- essential for building large systems
- keeps compile times reasonable
- requires independent procedures

**Establishes the need for private context**

- create a run-time “record” *for each procedure invocation*
- encapsulate run-time control and data information





## Storage Management Conventions

**1. Storage layout convention**

- defines memory layout of *code* and *data* (static, stack, heap)
- mostly specified by operating system

**2. Linkage convention**

- protocol for passing values and program control at procedure call and return
- all code in a single program must follow common convention
- partly specified by processor and OS; rest is left to compiler

**Related Tools**

- *linker*:
  - resolve name references across procedures
  - static linking: absolute virtual addresses chosen at link-time
  - dynamic linking: some addresses chosen at load time
- *loader*: loads code and static data into memory

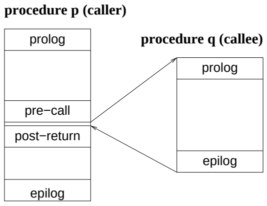




## *The linkage convention*

*The* linkage convention *ensures that procedures inherit a valid run-time environment* and *that they restore one for their parents.*

- specifies steps in calling sequence and return sequence
- division of responsibility between caller and callee




*At compile-time*: generate code to implement linkage convention
*At run-time*: that code manipulates the stack frame and data areas

## Activation Record or Stack Frame

At *run-time*, each procedure invocation has some associated storage. We call this its *activation record*. For most languages, activation records can live on the stack, and then they are also called *stack frames*.

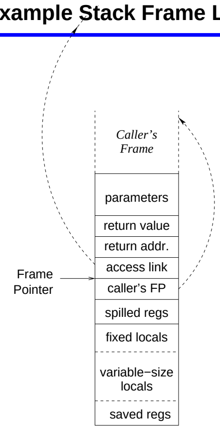
**Why is this useful?**

**Common components of a stack frame?**





## Example Stack Frame Layout




**Assumptions**

- Stack grows downwards in memory
- Dedicated frame pointer (FP) register
- Stack pointer (SP), if any, is separate

*Also see Sparc V9 Stack Frame handout.*





## Procedure linkages

**At a call**

**Caller:**

- allocate basic frame
- store parameters
- store return address
- save *caller-saved* regs
- store self's FP
- set FP for child
- jump to child

**Callee:**

- save *callee-saved regs*, state
- extend frame (for locals)
- initialize locals
- fall through to code

**At a return**

**Caller:**

- copy return value
- deallocate basic frame
- restore caller-saved regs.

**Callee:**

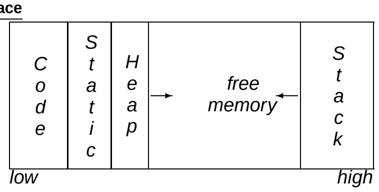
- store return value
- restore callee-saved regs, state
- unextend frame
- restore parent's FP
- jump to return address





## Run-time Storage Organization

**Logical address-space**




**Code**

- *functions*: fixed size, statically allocated

**Control stack**

- holds frames at run-time: dynamic slice of the *activation tree*

**Data**

- initialized global data
- uninitialized global data
- statically allocated local data
- variable-size or dynamic local data
- dynamic non-local data

## Mapping variables into memory

*Binding* ≡ mapping of a name to an attribute (e.g., storage location)

**Compile time**

→ assign each name a class (*base address*) & offset (*location*)
→ *Code generation*: generate instruction sequence for each access

**Link time**

→ decide static memory layout
→ resolve references across procedures (global variables and functions)

**Load time**

→ load code and static data into memory
→ resolve relocatable labels

**Run time**

→ execute linkage code at each procedure invocation
→ execute generated instruction sequences





## Run-time Storage Organization

Each variable must be assigned a storage class

**Global and "static" variables**

- symbolic addresses compiled into code (*relocatable*)
- limited to fixed-size objects
- layout may be important for performance
- compiler enforces access rights (private vs. global)

**Dynamically Allocated (non-local) Variables**

- returned pointers lead to non-local lifetimes
- explicit allocation
- explicit or implicit deallocation

**Procedure-local variables**

- Put in the stack frame *if* lifetime is limited (*see next slide*)
- Otherwise allocate on the heap
- *On stack:* May be allocated statically or dynamically
- *On stack*: May be fixed-size or variable size
- *On stack*: Implicit deallocation at procedure return





## Run-time Storage Organization

Do local variables go on stack?
⟸ Lifetime of memory is key

**Downward exposure**

- called procedure may reference my variables
- lexical *or* dynamic scoping

**Upward exposure**

- return a reference to my variables
- return a function that may reference my variables
- special case: <u>*later instance*</u> of current function or callee may reference my varibles
  - e.g., `static` (C) or `save` (FORTRAN)
  - problem: value must be preserved across calls





## Run-time Storage Organization

**How should these variables be allocated? Assume all are local/formal.**

C
```
   QueueItem* q;                         /* q?  */
   static int n;                         /* n?  */
```

Java
```
 QueueItem q = new QueueItem;           // q?  new object?
```

COOL
```
 let m: Int <- a+b in self.n <- m       -- m? self?
```

OCaml
```
 # let findFunc = fun x ->              (* x?       *)
     let p = makePair (x+1) (x+2) in    (* p?       *)
       (fun y -> (car p > y));;         (* y?  fun? *)
```

## Run-time Storage Organization

**Conditions for Stack Allocation**

*Allocate procedure-local data values in the stack frame if the locations are downwards-exposed only*
$\Longrightarrow$ values are not preserved across calls

**Optimization**

- *Escape analysis:* Decides whether a local variable or object can be used after procedure returns.
- Important for languages like Java, C# where all objects are heap-allocated by default.





## Access to non-local data

*Global & static variables*

**Compiler**

- assign variable an offset ($k$) from some symbolic address (*label*)
- generate unique label for each name (*name mangling*)
- code to emit:

```
loadI   ⟨label⟩   → r1
loadAI  r1, k,    → r2
```

**Static linking**

- Resolve all labels at link-time *Relocation*
- Assigns each label an absolute virtual address

**Dynamic linking**

- Relocate some labels at run-time
- *Indirection table for external references*
- *Compiler*: generates extra load to get base address from table
- *Runtime linker*: fills addresses of labels into indirection table





## Access to non-local data

*Local Variables with Lexical Scoping*

**Languages without nested procedures** :

- C, C++, Java, etc.
- Non-local names must be global or static
- Note: Heap-allocated data has no name: need some non-heap variable to access it (e.g., C pointer or Java reference)

**Languages with nested procedures (e.g., Pascal)** :

- view variables as ⟨*level,offset*⟩ pairs (*compile-time*)
- find pointer to appropriate activation record for *level*
- add *offset* to *level*'s pointer
- more expensive to access than locals

The ⟨*level,offset*⟩ pair is called a *Static Distance Coordinate*





## Access to non-local data

**Two important problems arise**

- How do we map a name into a ⟨*level,offset*⟩ pair?
- Given a ⟨*level,offset*⟩ pair, how to compute the address?

**How do we map a name into a level & offset?**

*Use a scoped symbol table* (compile-time)

- look up a name, want its most recent declaration
- declaration may be at current level or any lower level
- offsets directly determined by stack frame layout

**Given a level & offset, what's the address?**

*Two classic approaches* (run-time)

- access links or static links
- displays

*See slides and Cooper-Torczon text if interested.* (run-time)

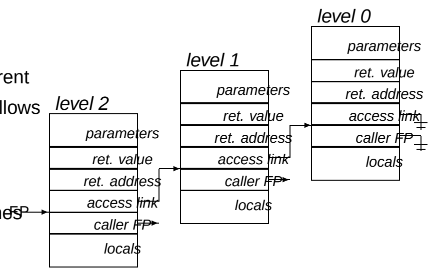
## Access links: run-time addressability

**The idea**

- each frame contains a pointer to its lexical parent
- chain of indirection follows lexical nesting

**The implementation**

- creates a chain of frames by lexical ancestry
- can follow chain to find level $k$ frame
- must maintain chain at call & return




*Note: caller FP is not same as access link*





## Access links: run-time addressability

**To find the value specified by $\langle l, o\rangle$**
(*assume $k$ = current procedure level = 2*)

| $\langle l, o\rangle$ | *Generated code* |
|---|---|
| $\langle 2, 8\rangle$ | `loadAI FP, 8` $\Rightarrow$ `r2` |
| $\langle 1, 12\rangle$ | `loadAI FP, -4` $\Rightarrow$ `r1`<br>`loadAI r1, 12` $\Rightarrow$ `r2` |
| $\langle 0, 16\rangle$ | `loadAI FP, -4` $\Rightarrow$ `r1`<br>`loadAI r1, -4` $\Rightarrow$ `r1`<br>`loadAI r1, 16` $\Rightarrow$ `r2` |

*Access cost for non-locals varies with $k - l$ Note: $k < l$ cannot occur*

**Maintaining access links:** (*static links* )

- calling level $k + 1$ procedure
  1. pass my FP as access link
  2. my backward chain will work for lower levels
- calling procedure at level $l < k$
  1. find my link to level $l - 1$ and pass it
  2. its access link will work for lower levels





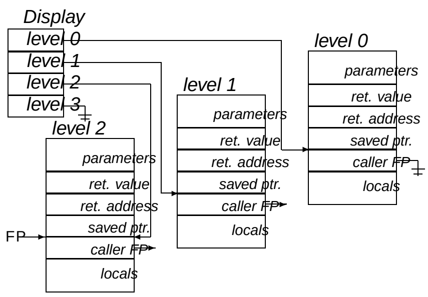
## The display: run-time addressability

**The idea**

- replace linked list with global array indexed by level
- global array: holds currently active frame pointers

**The implementation**

- single, global table
- each call updates the table appropriately
- reference is one indirection through table








## The display: run-time addressability

*To find the value specified by $\langle l, o\rangle$*
(*assume $k = 2$*)

| $\langle l, o\rangle$ | *Generated code* |
|---|---|
| $\langle 2, 8\rangle$ | `loadAI FP, 8` $\Rightarrow$ `r2` |
| $\langle 1, 12\rangle$ | `loadI DISPLAY_BASE` $\Rightarrow$ `r1`<br>`loadAI r1, 4` $\Rightarrow$ `r1`<br>`loadAI r1, 12` $\Rightarrow$ `r2` |
| $\langle 0, 16\rangle$ | `loadI DISPLAY_BASE` $\Rightarrow$ `r1`<br>`loadAI r1, 8` $\Rightarrow$ `r1`<br>`loadAI r1, 16` $\Rightarrow$ `r2` |

*Desired* FP *is at* DB $+ 4 \times l$
*Access cost is constant & independent of $k - l$*

*Maintaining a global display*
(*overallocate by 1 slot*)

1. *on entry to procedure at level $l$*
   $\rightarrow$ save level $l$ display ptr in frame
   $\rightarrow$ push FP into level $l$ display slot
2. *on return*
   $\rightarrow$ restore the level $l$ display slot from frame

*Quick, simple, & foolproof*

## Minor issues (Displays & access links)

**Improvements**

- *Leaf procedures* (*contain no calls*)
  - → with display, don't update it
  - → statically allocate frame
- *Other procedures*
  - → 1 call in loop ⇒ move frame manipulation out of loop
  - → keep accessed FP elements in temporaries

**Cost comparison**

- *Display* *Major problem: threads*
  - → `load` + `store` on CALL & RETURN
  - → `loadI` + 2 `loadAI`'s per access
- *Access links*
  - → `store` on CALL, nothing on RETURN
  - → $(k - l + 1)$ `loadAI`'s per access