

The procedure abstraction

Separate compilation:

- allows us to build large programs
- keeps compile times reasonable
- requires independent procedures

The linkage convention:

- a social contract
- machine dependent
- division of responsibility

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

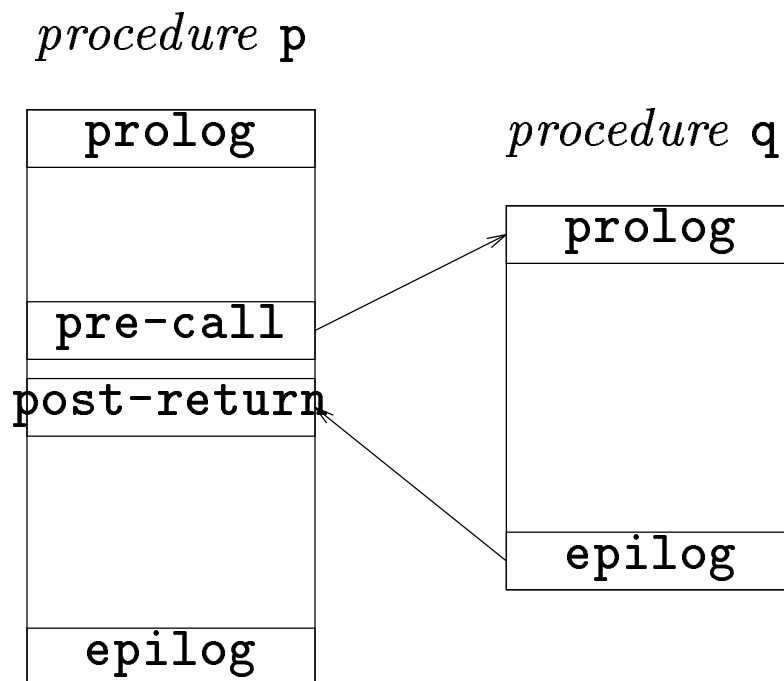
Linkages execute at *run time*

Code to make the linkage is generated at *compile time*

The procedure abstraction

The essentials:

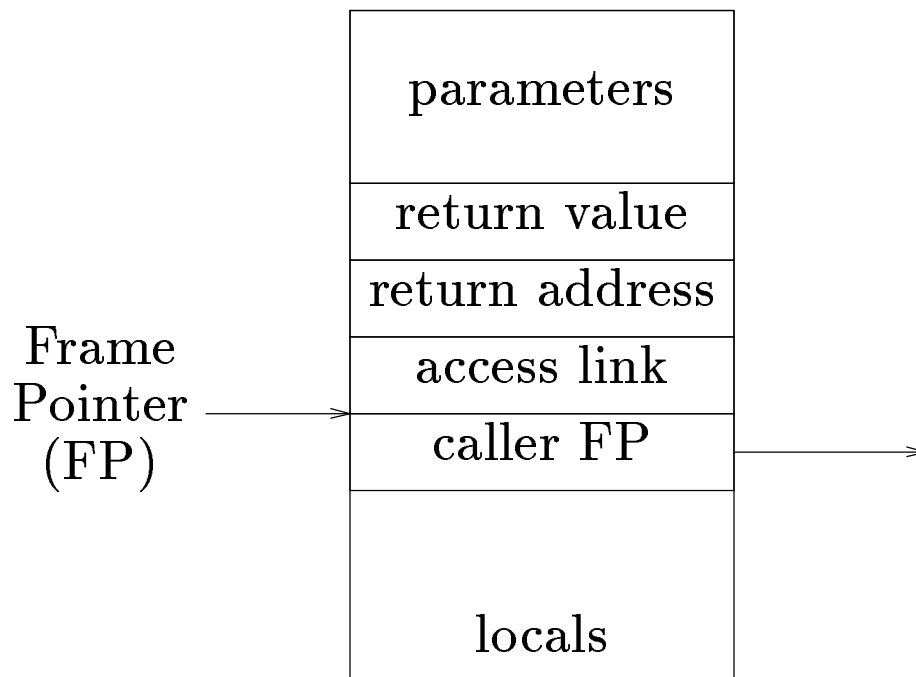
- *on entry*, establish *p*'s environment
- *at a call*, preserve *p*'s environment
- *on exit*, tear down *p*'s environment
- *in between*, addressability and proper lifetimes



Each system has a *standard linkage*

Procedure linkages

Assume that each procedure activation has an associated *activation record* or *frame* (at run time)



Assumptions:

- call by reference parameter passing
- RISC architecture
- can always expand an allocated block
- locals stored in frame

Procedure linkages

The linkage divides responsibility between *caller* and *callee*

	Caller	Callee
Call	<i>call sequence</i>	<i>prolog code</i>
	allocate basic frame evaluate & store params. store return address store FP set FP for child jump to child	save registers, state extend basic frame (for local data) find static data area initialize locals fall through to code
Return	<i>return sequence</i>	<i>epilog code</i>
	copy return value deallocate basic frame restore params. (?)	store return value restore state unextend basic frame restore parent's FP jump to return address

At compile time, we generate the code to do this

At run time, that code manipulates the frame & data areas

Run-time storage organization

To maintain the illusion of procedures, the compiler must adopt some conventions to govern memory use.

Code space

- fixed size
- statically allocated *(link time)*

Data space

- fixed size data may be statically allocated
- variable size data must be dynamically allocated
- some data is dynamically allocated in code

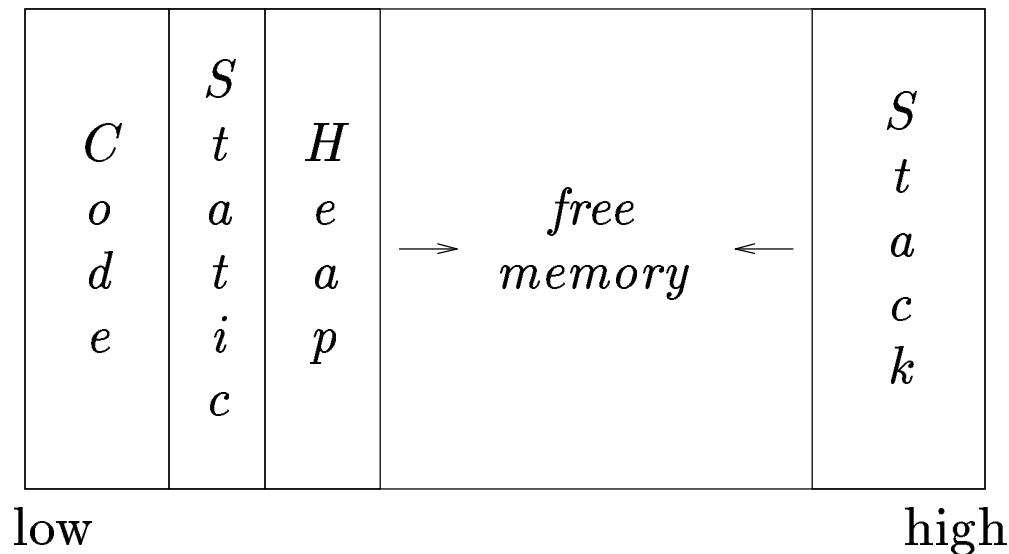
Control stack

- dynamic slice of activation tree
- return addresses
- usually implemented in hardware

Run-time storage organization

Typical memory layout

Logical Address Space



The classical scheme

- allows both stack and heap maximal freedom
- code and static may be separate or intermingled

Run-time storage organization

Where do local variables go?

When can we allocate them on a stack?

Key issue is lifetime of local names

Downward exposure:

- called procedures may reference my variables
- dynamic scoping
- lexical scoping

Upward exposure:

- can I return a reference to my variables?
- functions that return functions
- continuation-passing style

With only *downward exposure*, the compiler can allocate the frames on the run-time stack

Run-time storage organization

Each variable must be assigned a storage class
(*base address*)

Static variables:

- addresses compiled into code (*relocatable*)
- (*usually*) allocated at compile-time
- limited to fixed size objects
- control access with naming scheme

Global variables:

- almost identical to static variables
- layout may be important (*exposed*)
- naming scheme ensures universal access

Linkage editor must handle duplicate definitions

Run-time storage organization

Storage classes (*con't*):

Procedure local variables

Put them on the stack —

- *if* sizes are fixed
- *if* lifetimes are limited
- *if* values are not preserved

Dynamically allocated variables

Must be treated differently —

- call-by-reference, pointers, lead to non-local lifetimes
- (*usually*) an explicit allocation
- explicit or implicit deallocation

Access to non-local data

How does the code find non-local data at *run-time*?

Real globals

- visible *everywhere*
- naming convention gives an address
- initialization requires cooperation

Lexical nesting

- view variables as $(level, offset)$ pairs
(*compile-time*)
- chain of non-local access links
- more expensive to find (*at run-time*)

Access to non-local data

Two important problems arise

1. *How do we map a name into a (level,offset) pair?*

We use a block structured symbol table

- when we look up a name, we want to get the most recent declaration for the name
- the declaration may be found in the current procedure or in any nested procedure

2. *Given a (level,offset) pair, what's the address?*

Two classic approaches

⇒ access links

(*static links*)

⇒ displays

Access to non-local data

To find the value specified by (l, o)

- need current procedure level, k
- if $k = l$, is a local value
- if $k > l$, must find l 's activation record
- $k < l$ 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

To improve run-time access costs, use a *display*.

- table of access links for lower levels
- lookup is index from known offset
- takes slight amount of time at call
- a single display or one per frame
- for level k procedure, need $k - 1$ slots

Access with the display

assume a value described by (l, o)

- find slot as $DP + 4 \times l$
- add offset to pointer from slot

“setting up the base frame” now includes display manipulation.

Display management

Single global display: *simple method*

Key insight – overallocate the display by 1 slot

on entry to a procedure at level l

save the level l display value

push FP into level l display slot

on return

restore the level l display value

Quick, simple, and foolproof!

Display management

Individual frame-based displays:

call from level k to level l

if $l \leq k$

copy $l - 1$ display entries into child's frame

if $l > k$ ($l = k + 1$)

copy $k - 1$ entries into child's frame

copy own FP into k^{th} slot in child's frame

no work required on return

\Rightarrow display is deallocated with frame

Display accessed by offset from FP.

\Rightarrow one less register required

Display versus access links

How to make the trade-off?

The cost differences are somewhat subtle

- frequency of non-local access
- average lexical nesting depth
- ratio of calls to non-local access

(*Sort of*) Conventional wisdom

tight on registers ⇒ use access links

lots of registers ⇒ use global display

shallow average nesting ⇒ frame-based display

Your mileage will vary

Making the decision requires understanding reality

Parameter passing

What about parameters?

Call-by-value

- store values, not addresses
- never restore on return
- arrays, structures, strings are a problem

Call-by-value-result

- store values, not addresses
- always restore on return
- arrays, structures, strings are a problem

Call-by-name

- build and pass *thunk*
- access to parameter invokes thunk
- all parameters are same size in frame!

Parameter passing

What about variable length argument lists?

1. if *caller* knows that *callee* expects a variable number
 - (a) *caller* can pass number as 0^{th} parameter
 - (b) *callee* can find the number directly
2. if *caller* doesn't know anything about it
 - (a) *callee* must be able to determine number
 - (b) first parameter must be closest to FP

Consider `printf` :

- number of parameters determined by the format string
- it assumes the numbers match

Heap management techniques

Functionality:

- `alloc(k)` — locates a block of *at least* *k* bytes in the free space pool, removes it from the pool, and returns its address
- `free(p)` — places the block pointed to by *p* back in the pool of free space for allocation

If unused storage is reclaimed, `free` is not needed.

Potential problems:

- wasted space — if `alloc` returns blocks that are larger than requested, the excess space is wasted
- fragmentation — after a series of `alloc` and `free` commands, the free space pool becomes fragmented, preventing allocation of large blocks
- speed — `alloc` and `free` should be inexpensive

A heap management scheme must balance these issues.

Knuth, Volume 1, §2.5

Heap management techniques

Scheme 1:

initialization

- start the free list with a single large block
- need two fields in each free block: *size* and *next*
- allocated blocks keep *size* field

alloc(*k*)

- find first block where $k + 4 \leq \textit{size}$
- no such block \Rightarrow report failure or get more space
- create block at $k + 4$ and put it on free list
- update free list
- return address of second word of allocated block

free(*p*)

- put *p* back on the free list

This is a *first-fit* scheme without *coalescing* .

Heap management techniques

Problems with scheme 1:

1. over time, start of free list becomes dominated by small blocks

solution: rover

2. blocks get too small to be useful

solution: place a minimum size on blocks

3. over time, large blocks get scarce

solution: coalesce on **free**

Heap mangement techniques

Scheme 2:

initialization

- start the free list with a single large block
- set rover to start of free list

alloc(k)

- starting at rover, find first block that fits
- no such block \Rightarrow report failure or get more space
- split block into $2^{\lceil \log_2(k+4) \rceil}$ bytes and the rest
- place second block on free list, *in address order*
- return address of first block + 4

free(p)

- insert p on the free list, *in address order*
- if block before p is contiguous, coalesce with p
- if block after p is contiguous, coalesce with p .

First fit with rover, coalescing, bounded block size

Heap mangement techniques

Scheme 3:

- instead of finding *first fit* , find the *best fit*
- only split a block if $size > k + threshold$

Comparison

- best fit wastes less space
- best fit requires more allocation time

first fit \Rightarrow average case $<$ size of *free pool*

best fit \Rightarrow average case is worst case

Heap mangement techniques

Modern allocators rely on several observations:

- large real and virtual memories are common
- speed is important
- sizes come in two flavors, common and unusual

Capitalizing on these facts, we can do better in practice.

Principles:

- separate pools for common sizes
- limit number of common sizes (2^i)
- use page size as upper bound on common size

Heap mangement techniques

A modern allocator:

- use separate free space pools for common sizes
 - 2^4 to 2^{12} (*or page size*)
- first fit within pool
 - **alloc** is $\mathbf{O}(1)$ + amortizing new page sbrk's
 - **free** is $\mathbf{O}(1)$, no coalescing, no order
- use a separate mechanism for larger regions
 - first fit over list of full pages
 - allocate only full size pages
 - coalesce on **free** to ensure contiguous blocks
 - $\mathbf{O}(n)$ **alloc** and **free**

Heap mangement techniques

A collecting allocator

- make `free()` automatic
- no pointer to a block → done with it
- deallocate only when we run out of space

Simplifies programmer's life (important)

Observations

- either recognize pointers or be *very* conservative
- eliminate all those calls to `free()`
- add time for collection

Techniques

- reference counting
- marking algorithms (Schorr-Waite)
- copying collectors

Relating lecture to the book

Chapter 7

— we have covered most of it

Remaining issues:

- alignment and layout of data
 - varies from machine to machine
 - may require padding (*deliberate waste*)
- procedure-valued parameters
 - pass both code address and lexical level
 - generate code to check cases at run-time