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.

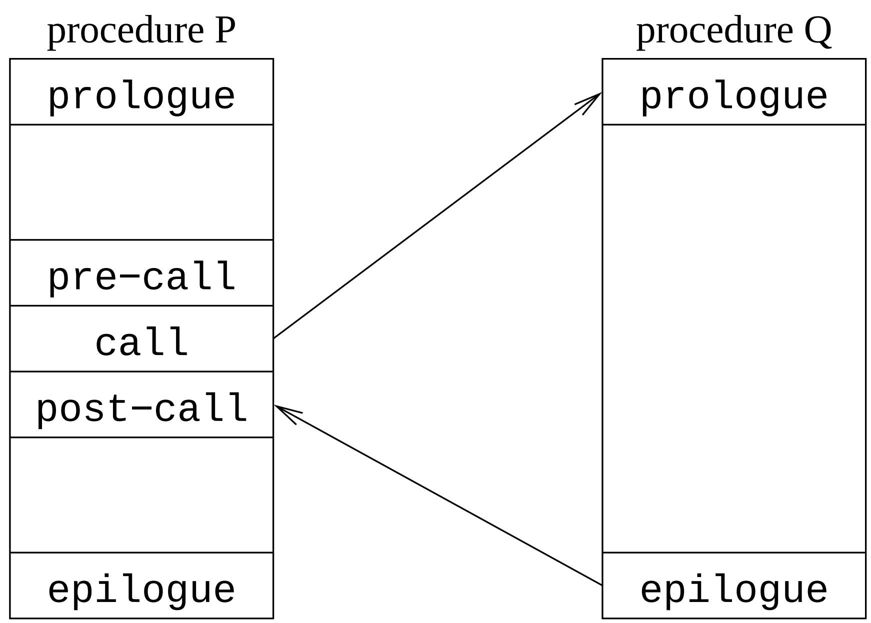
Linkages execute at *run time*

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



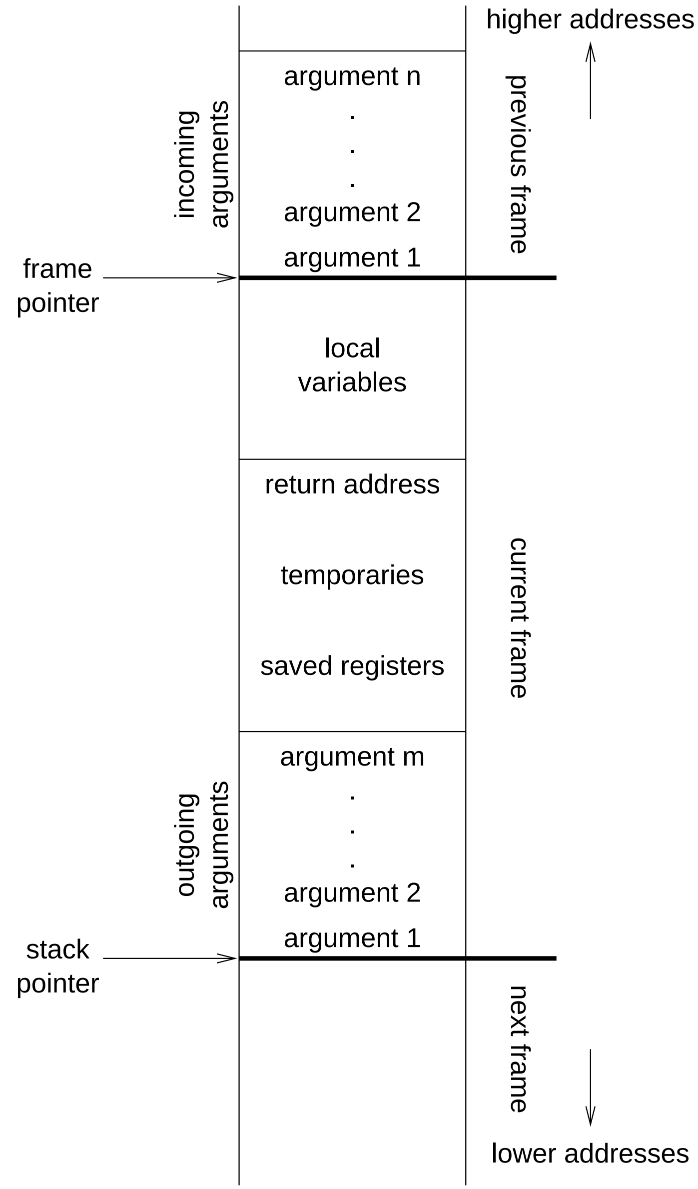
Each system has a *standard linkage*

Procedure linkages

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

Assumptions:

- 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	*pre-call*	*prologue*
	1. allocate basic frame 2. evaluate & store params. 3. store return address 4. jump to child	1. save registers, state 2. store FP (dynamic link) 3. set new FP 4. store static link 5. extend basic frame (for local data) 6. initialize locals 7. fall through to code
Return	*post-call*	*epilogue*
	1. copy return value 2. deallocate basic frame 3. restore parameters (if copy out)	1. store return value 2. restore state 3. cut back to basic frame 4. restore parent's FP 5. jump to return address

At compile time, 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 can adopt some conventions to govern memory use.

Code space

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

Data space

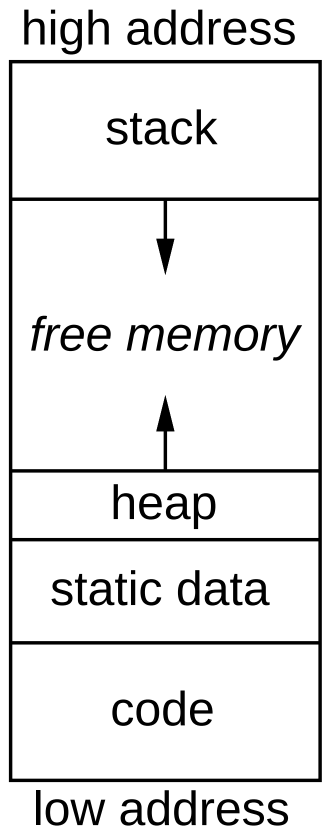
- fixed-sized data may be statically allocated
- variable-sized data must be dynamically allocated
- some data is dynamically allocated in code

Control stack

- dynamic slice of activation tree
- return addresses
- may be implemented in hardware

Run-time storage organization

Typical memory layout



The classical scheme

- allows both stack and heap maximal freedom
- code and static data 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 call stack

Storage classes

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

Link editor must handle duplicate definitions

Storage classes (*cont.*)

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

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

  Use a *block-structured symbol table* (remember last lecture?)

  - look up a name, want its most recent declaration
  - declaration may be at current level or any lower level

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

  Two classic approaches

  - access links (or *static links*)
  - displays

Access to non-local data

To find the value specified by (l, o)

- need current procedure level, k
- $k = l \Rightarrow$ local value
- $k > l \Rightarrow$ 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 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 `display`$[l]$
- add offset to pointer from slot (`display`$[l][o]$)

"Setting up the basic frame" now includes display manipulation

Display management

Single global display: *complex, obsolete method*

bogus idea, do not use

Call from level k to level l

if $l = k+1$
add a new display entry for level k

if $l = k$
no change to display is required

if $l < k$
preserve entries for levels l through $k-1$ in the local frame

On return (*back in calling procedure*)

if $l < k$
restore preserved display entries

A single display ties up another register

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 $l > k$ $\quad (l = k+1)$
copy $k-1$ entries into child's frame
copy own FP into k^{th} slot in child's frame

No work required on return

- 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	$\Rightarrow$	use access links
lots of registers	$\Rightarrow$	use global display
shallow average nesting	$\Rightarrow$	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-reference

- pass address
- access to formal is indirect reference to actual

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

Calls: Saving and restoring registers

	caller's registers	callee's registers	all registers
callee saves	1	3	5
caller saves	2	4	6

1. Call includes bitmap of caller's registers to save/restore (best with save/restore instructions to interpret bitmap)
2. Caller saves and restores its own registers
   Unstructured returns (e.g., non-local gotos, exceptions) create some problems, since code to restore must be located and executed
3. Backpatch code to save regs used in callee on entry, restore on exit
   e.g., VAX places bitmap in callee's stack frame for use on call/return/non-local goto/exception
   Non-local gotos and exceptions must unwind dynamic chain restoring callee-saved registers
4. Bitmap in callee's stack frame is used by caller to save/restore (best with save/restore instructions to interpret bitmap directly)
   Unwind dynamic chain as for 3
5. Easy: Non-local gotos and exceptions must restore all registers from "outermost callee"
6. Easy (use utility routine to keep calls compact)
   Non-local gotos and exceptions need only restore original registers from caller

Top-left is best: saves fewer registers, compact calling sequences

Call/return

Assuming callee saves:

1. caller pushes space for return value
2. caller pushes SP
3. caller pushes space for:
   return address, static chain, saved registers
4. caller evaluates and pushes actuals onto stack
5. caller sets return address, callee's static chain, performs call

6. callee saves registers in register-save area
7. callee copies by-value arrays/records using addresses passed as actuals
8. callee allocates dynamic arrays as needed

9. on return, callee restores saved registers
10. jumps to return address

Caller must allocate much of stack frame, because it computes the actual parameters

Alternative is to put actuals below callee's stack frame in caller's: common when hardware supports stack management (e.g., VAX)

MIPS procedure call convention

Registers:

Number	Name	Usage
0	`zero`	Constant 0
1	at	Reserved for assembler
2, 3	v0, v1	Expression evaluation, scalar function results
4–7	a0–a3	first 4 scalar arguments
8–15	t0–t7	Temporaries, caller-saved; caller must save to preserve across calls
16–23	s0–s7	Callee-saved; must be preserved across calls
24, 25	t8, t9	Temporaries, caller-saved; caller must save to preserve across calls
26, 27	k0, k1	Reserved for OS kernel
28	gp	Pointer to global area
29	sp	Stack pointer
30	s8 (fp)	Callee-saved; must be preserved across calls
31	ra	Expression evaluation, pass return address in calls

MIPS procedure call convention

Philosophy:

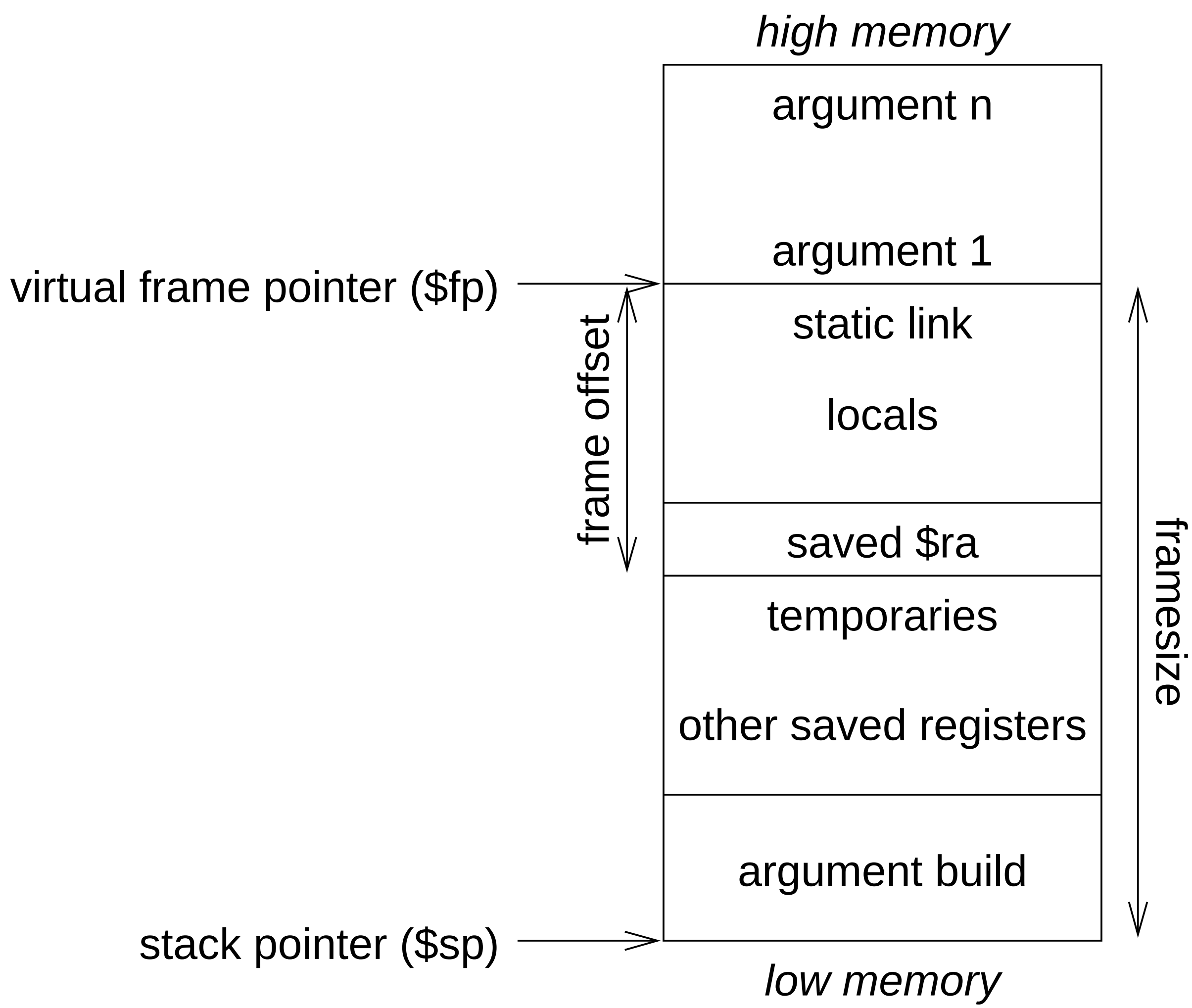
Use full, general calling sequence only when necessary; omit portions of it where possible (e.g., avoid using fp register whenever possible)

Classify routines as:

- non-leaf routines: routines that call other routines
- leaf routines: routines that do not themselves call other routines
  - leaf routines that require stack storage for locals
  - leaf routines that do not require stack storage for locals

MIPS procedure call convention

The stack frame



MIPS procedure call convention

Pre-call:

1. Pass arguments: use registers \$a0 . . . \$a3; remaining arguments are pushed on the stack along with save space for \$a0 . . . \$a3
2. Save caller-saved registers if necessary
3. Execute a `jal` instruction: jumps to target address (callee's first instruction), saves return address in register \$ra

MIPS procedure call convention

Prologue:

1. Leaf procedures that use the stack and non-leaf procedures:
   (a) Allocate all stack space needed by routine:
   - local variables
   - saved registers
   - sufficient space for arguments to routines called by this routine

   ```
   subu $sp,framesize
   ```

   (b) Save registers ($ra, etc.):

   ```
   sw $31,framesize+frameoffset($sp)
   sw $17,framesize+frameoffset-4($sp)
   sw $16,framesize+frameoffset-8($sp)
   ```

   where `framesize` and `frameoffset` (usually negative) are compile-time constants
2. Emit code for routine

MIPS procedure call convention

Epilogue:

1. Copy return values into result registers (if not already there)
2. Restore saved registers

   ```
   lw reg,framesize+frameoffset-N($sp)
   ```
3. Get return address

   ```
   lw $31,framesize+frameoffset($sp)
   ```
4. Clean up stack

   ```
   addu $sp,framesize
   ```
5. Return

   ```
   j $31
   ```