Activation Records

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

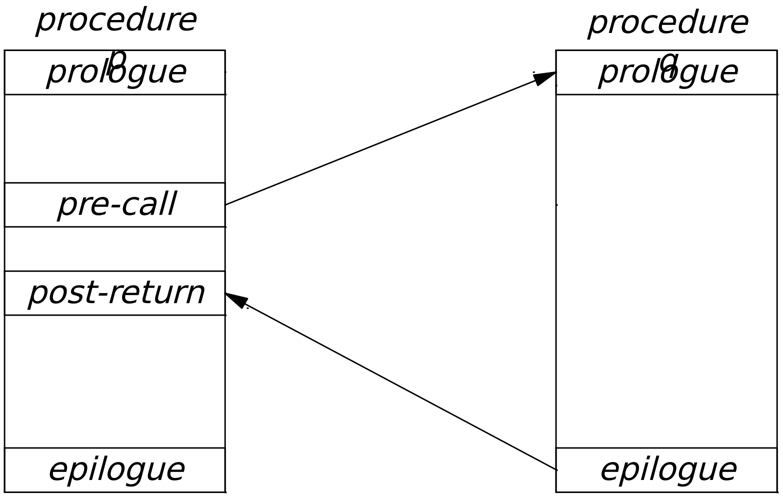
Linkages execute at *run time*

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



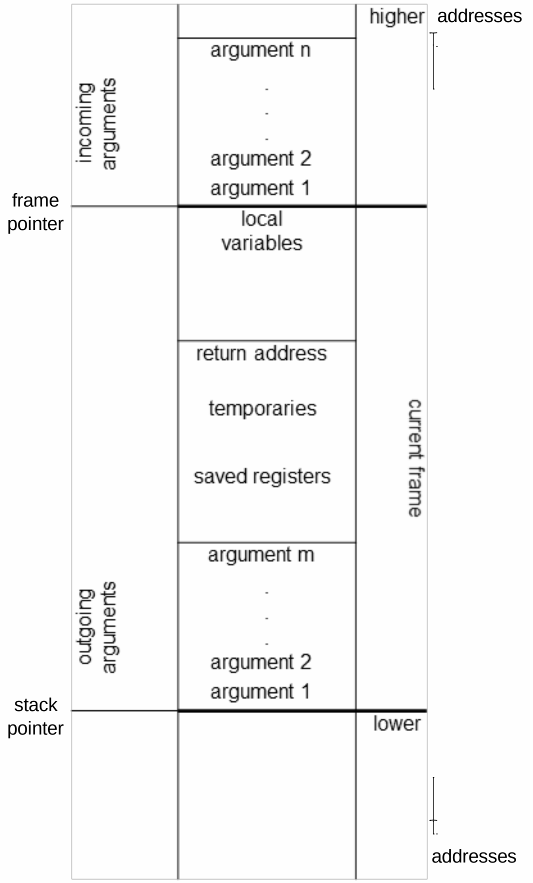
Each system has a *standard linkage*

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

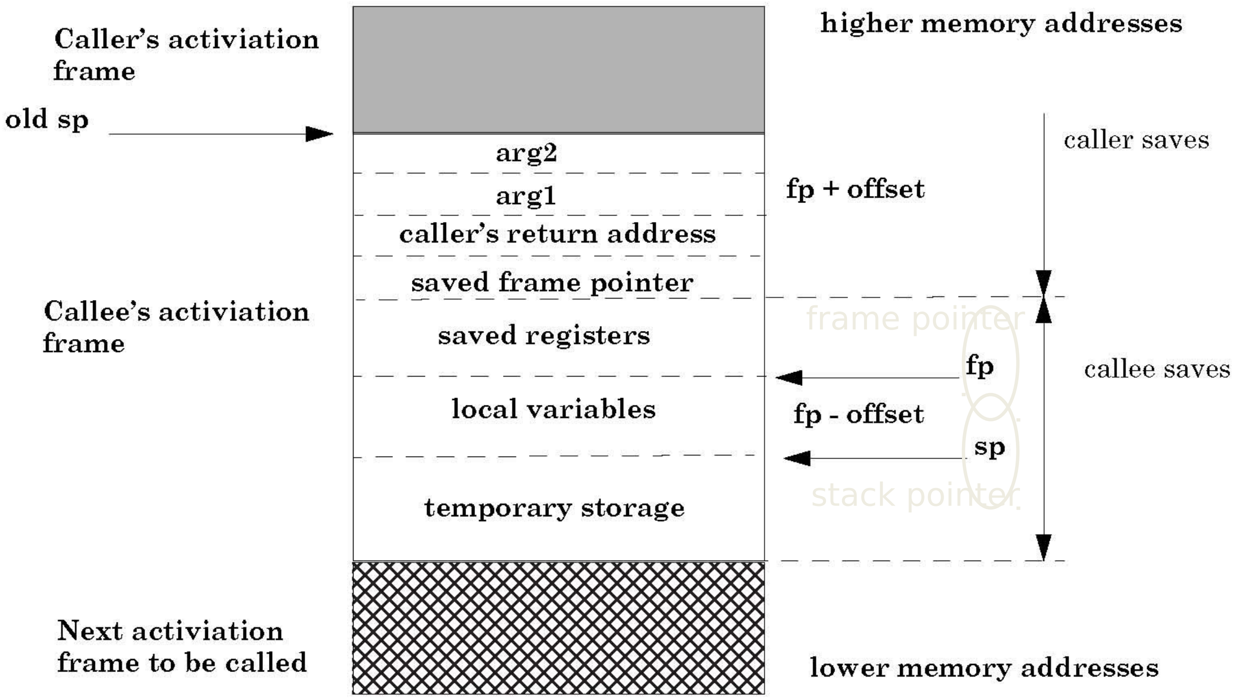
The linkage divides responsibility between *caller* and *callee*

	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

Typical x86 Activation Record



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

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

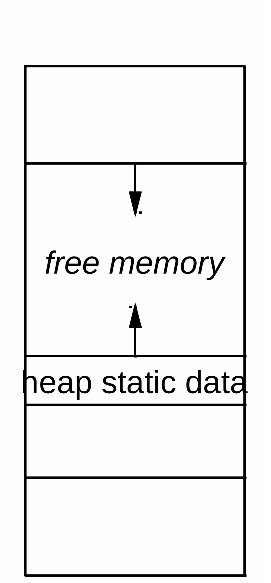
Control stack

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

Run-time storage organization

Typical memory layout

high address stack



code low address

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

> **Static variables**
- addresses compiled into code
- usually allocated at compile-time (fixed-size objects)
- naming scheme to control access

> **Global variables**
- similar to static variables
- layout may be important
- universal access

> **Procedure local variables**
- allocated on stack …
- if fixed size, limited lifetime, and values not preserved

> **Dynamically allocated variables**
- call-by-reference implies non-local lifetime
- usually explicit allocation
- de-allocation explicit or implicit

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*

- 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

Implementation of subprogram storage

Consider the following C subprogram:

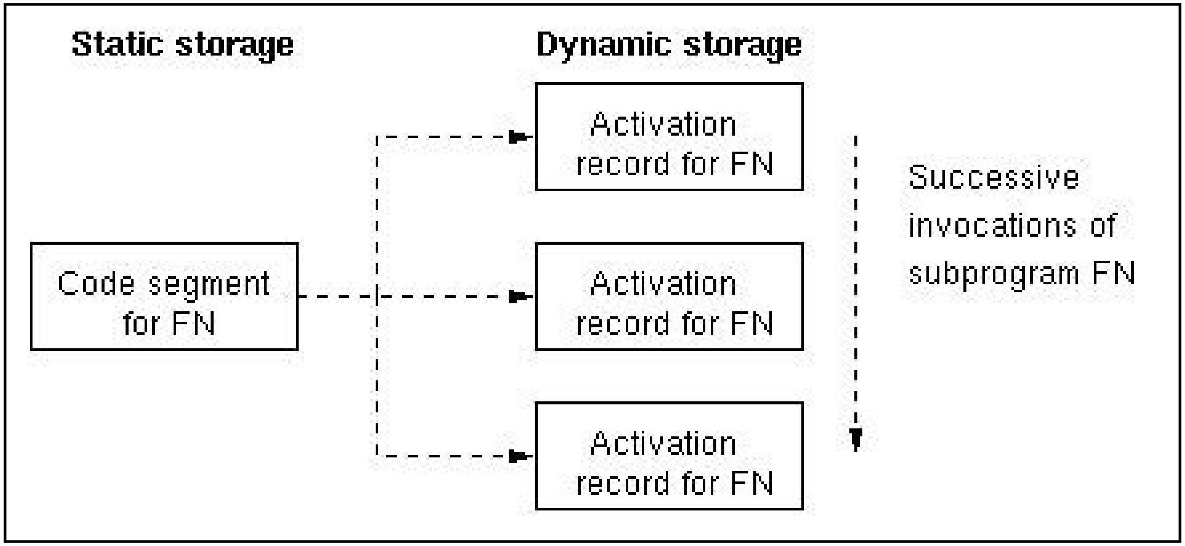
```
        float FN( float X, int Y)
                const initval=2;
                #define finalval  10
                float M[10]; int N;
                N = initval;
                if(N<finalval){ ... }
                return (20 * X + M[N]); }
```

- Information about procedure FN is contained in its activation record.

Dynamic nature of activation records

- Each invocation of FN causes a new activation record to be created.
- Thus the static code generated by the compiler for FN will be associated with a new activation record, each time FN is called.
- As we will see later, a stack structure is used for activation record storage.

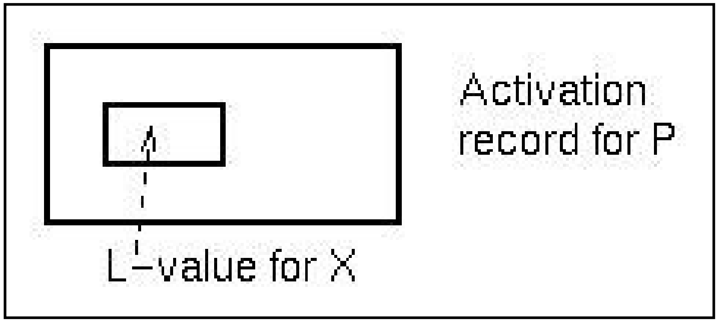
Dynamic nature of activation records



Subprogram control

- Remember that data storage for subprograms is in an activation record.

  var X: integer;
- X is of type integer.
- L-value of X is some specific offset in an activation record.



Subprogram control

- Goal is to look at locating activation record for P.

Given an expression: X = Y + Z

1. Locate activation record containing Y.
2. Get L-value of Y from fixed location in activation record.
3. Repeat process for Z and then X.

Scope rules

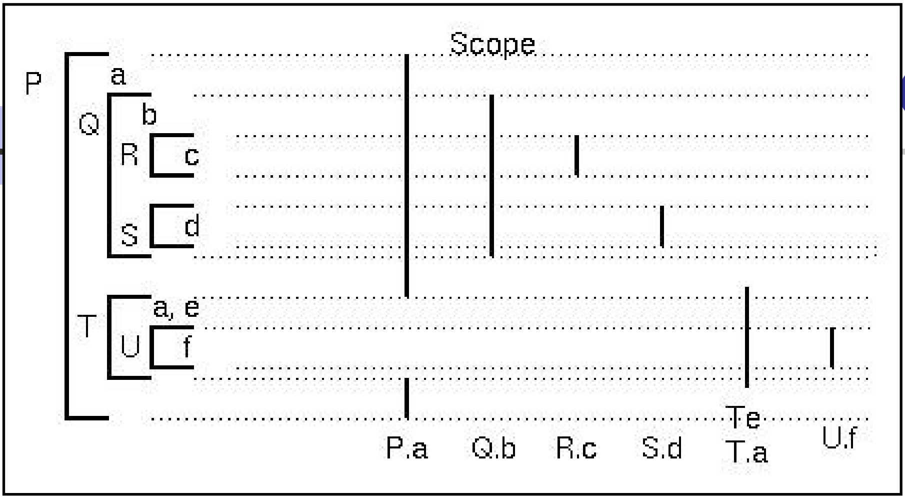
- Scope rules: The scope of a variable are the set of statements where the variable may be accessed (i.e., named) in a program.
- Static scope: Scope is dependent on the syntax of the program.
- Dynamic scope: Scope is determined by the execution of the program.

Scope rules

- Static nested scope: A variable is accessible in the procedure it is declared in, and all procedures internal to that procedure, except a new declaration of that variable name in an internal procedure will eliminate the new variable's scope from the scope of the outer variable.

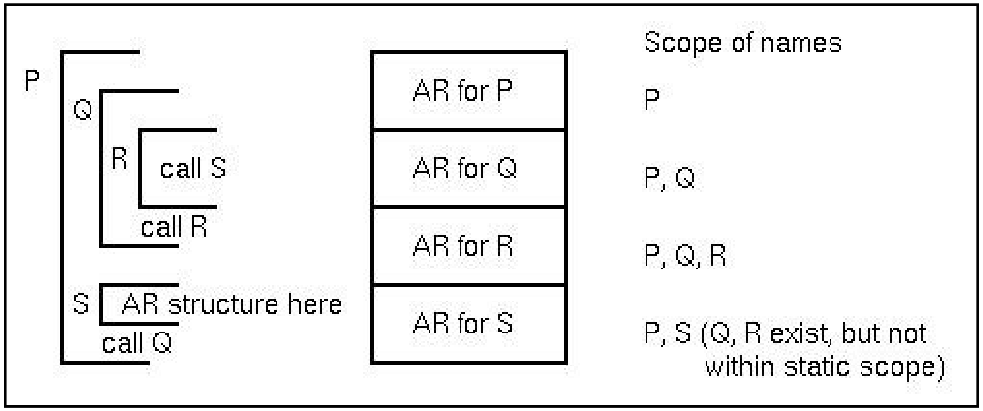
- A variable declared in a procedure is local in that procedure; otherwise it is global.

es



- Q and T are declarations of procedures within P, so scope of names Q and T is same as scope of declaration a.
- R and S are declarations of procedures in Q.
- U is a declaration of a procedure in T.
- Storage managed by adding activation records, when procedure invoked, on a stack.

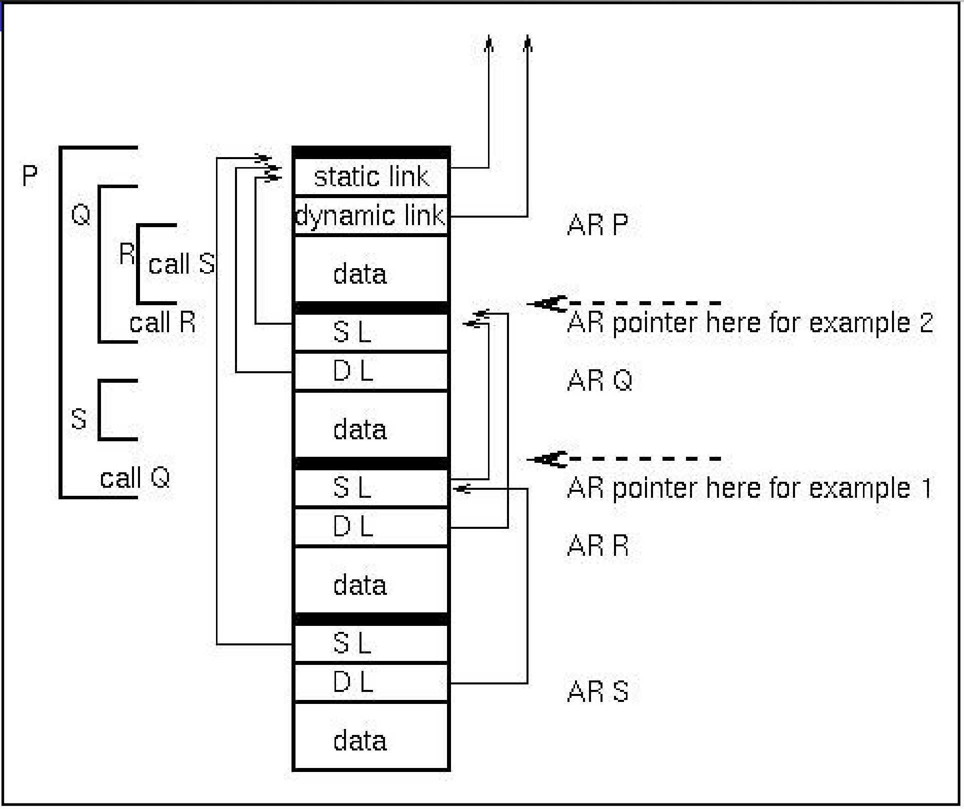
Activation record stack



Activation record stack

- Problem is: How to manage this execution stack?
- Two pointers perform this function:
- 1. Dynamic link pointer points to activation record that called (invoked) the new activation record. It is used for returning from the procedure to the calling procedure. And to catch exceptions in dynamic scope EH.
- 2. Static link pointer points to the activation record that is global to the current activation record (i.e., points to the activation record of the procedure containing the declaration of this procedure).

Example of act. record stack



Declarations

Var A in P

B in Q

C in R

Activation record example 1

- Ex 1. In R: C := B+A; ⇒ C local, A,B global
- For each variable, get pointer to proper activation record.
- Assume AR is current act.record pointer (R).
- 1. B is one level back:
- Follow AR.SL to get AR containing B.
- Get R-value of B from fixed offset L-value
- 2. A is two levels back:
- Follow (AR.SL).SL to get activation record containing A.
- Add R-value of A from fixed offset L-value
- 3. C is local. AR points to correct act record.
- Store sum of B+A into L-value of C

Activation record example 2

- Example 2. Execution in procedure Q: A := B
- ⇒ B is local, A global
- Assume AR is current activation record pointer (Q)

- 1. B is now local. AR points to activation record
  - Get R-value from local activation record

Activation record example 2

- 2. A is now one level back
- AR.SL is activation record of P
    - Store R-value of B into L-value of A

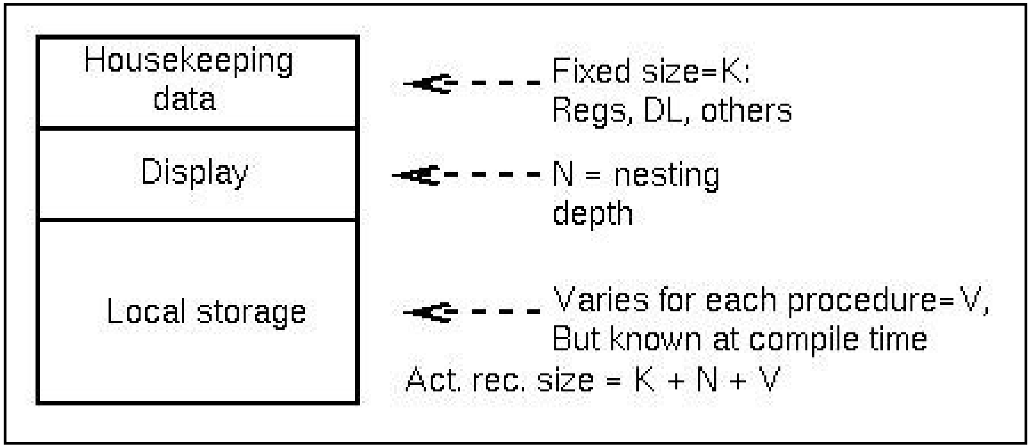
- Compiler knows static structure, so it can generate the number of static link chains it has to access in order to access the correct activation record containing the data object. This is a compile time, not a runtime calculation.

Use of displays

- Problem with static links:
- Many links to follow if deeply nested. (But how common is that?)
- Use of displays reduces access always to 2 instructions:
- 1. Access Display[I] = Act. Rec. pointer at level I
- 2. Get L-value of variable (fixed offset in act. rec.)

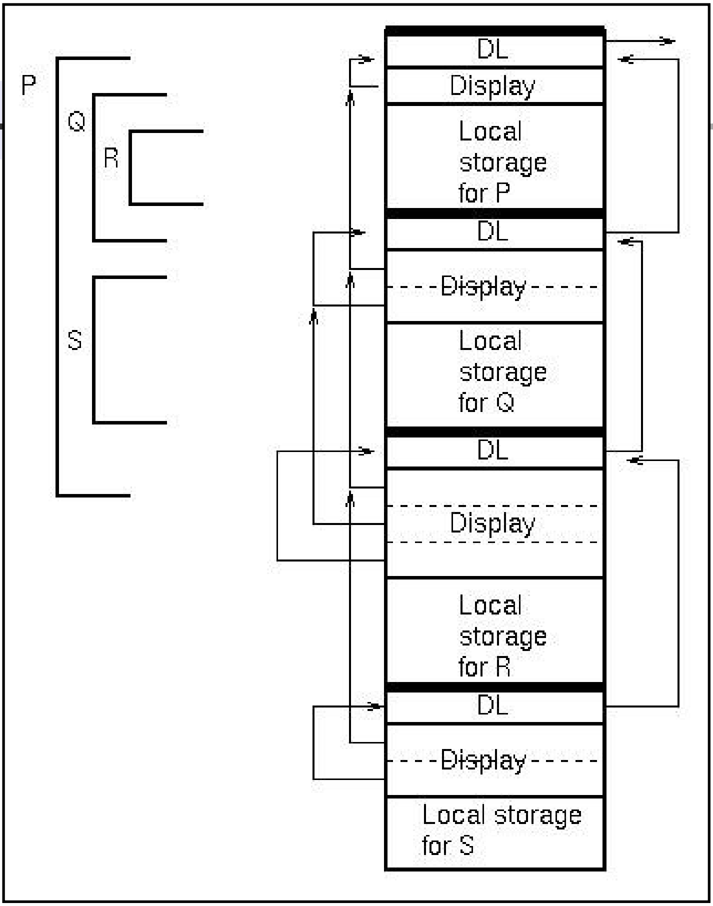
Use of displays

- Activation record structure:


Housekeeping data
Display
Local storage
Fixed size=K: Regs, DL, others
N = nesting depth
Varies for each procedure=V, But known at compile time
Act. rec. size = K + N + V

Static chain copied into display vector on subprogram entry.

Display for previous example



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

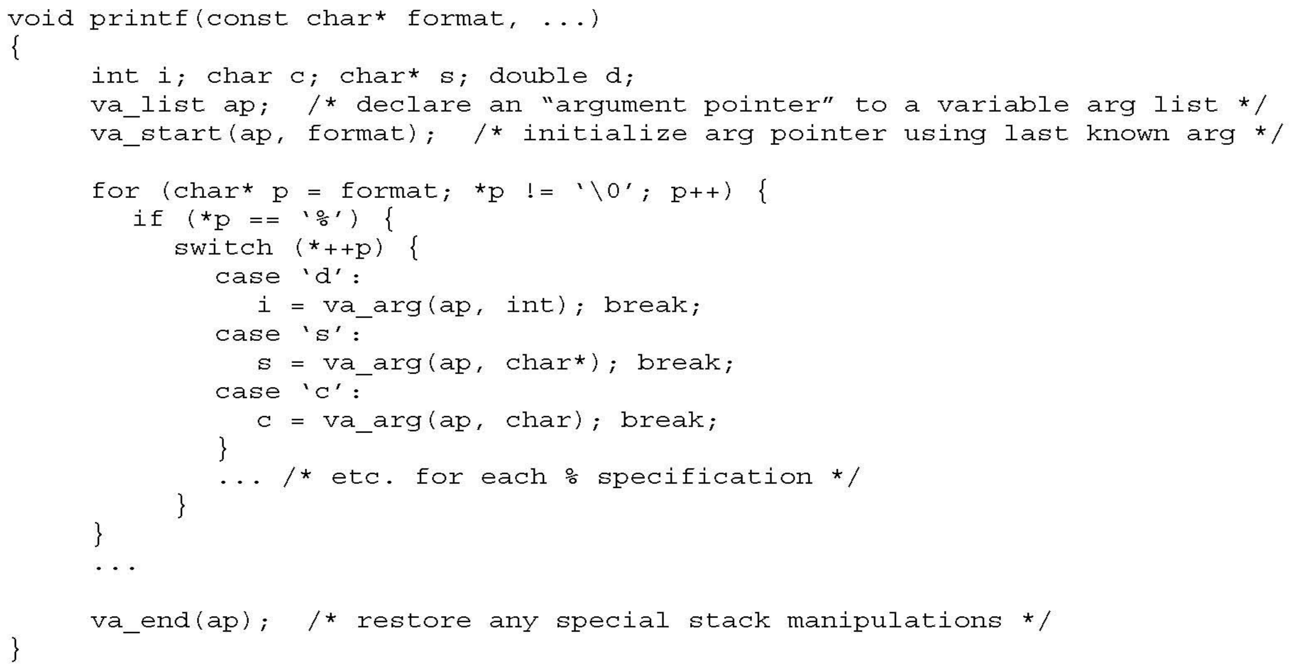
tight on registers	⇒	use access links
lots of registers	⇒	use global display
shallow average nesting	⇒	frame-based display

Your mileage will vary

Making the decision requires understanding reality

Variable Arguments (Redux)

- Special functions va_start, va_arg, va_end compute arguments at run-time (how?)

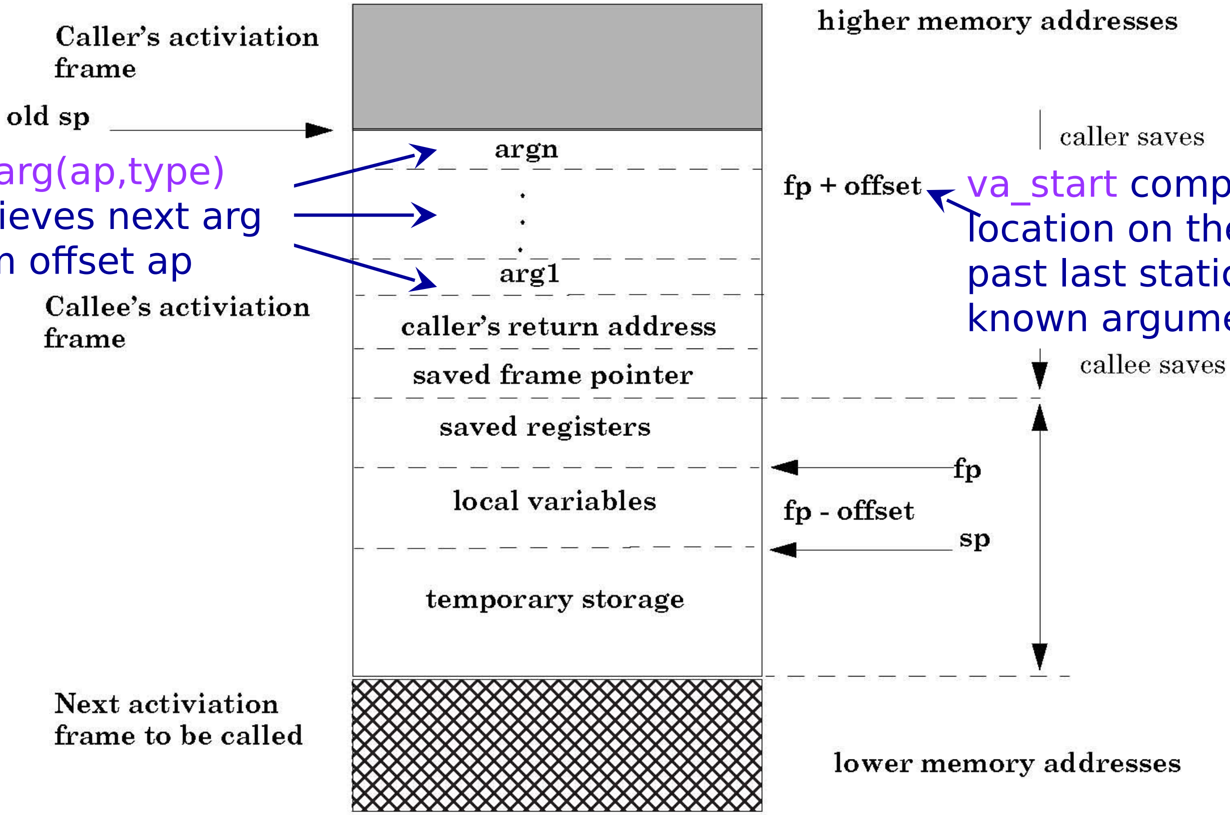


```
void printf(const char* format, ...)
{
      int i; char c; char* s; double d;
      va_list ap;  /* declare an "argument pointer" to a variable arg list */
      va_start(ap, format);  /* initialize arg pointer using last known arg */

      for (char* p = format; *p != '\0'; p++) {
         if (*p == '%') {
            switch (*++p) {
               case 'd':
                  i = va_arg(ap, int); break;
               case 's':
                  s = va_arg(ap, char*); break;
               case 'c':
                  c = va_arg(ap, char); break;
               }
               ... /* etc. for each % specification */
            }
      }
      ...

      va_end(ap);  /* restore any special stack manipulations */
}
```

Activation Record for Variable Args



va_arg(ap,type) retrieves next arg from offset ap

va_start computes location on the stack past last statically known argument

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 pre-serve across calls
16–23	s0–s7	Callee-saved; must be preserved across calls
24, 25	t8, t9	Temporaries, caller-saved; caller must save to pre-serve 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

Syllabus

Lecture Series (hours)	Topics
1-4	Introduction and Motivation, Paradigms
5-10	Syntax and Semantics, BNF, Compilation
11-18	Data Types, Constructs, Functions, Activation Records, Names and Bindings
19-28	Concurrency, Functional PLs, Logical PLs, Lambda Calculus, Event driven programming
29-36	Virtual Machines, Managed Languages, JIT, Case study

Continuation passing scheme

```
void buttonHandler() { // This is executing in the Swing UI thread.
// We can access UI widgets here to get query parameters.
            final int parameter = getField();
            new Thread(new Runnable() {
                        public void run() {
                        // This code runs in a separate thread.
                        // We can do things like access a database or a
                        // blocking resource like the network to get data.
                        final int result = lookup(parameter);
                        javax.swing.SwingUtilities.invokeLater(new Runnable() {
                        public void run() {
                        // This code runs in the UI thread and can use
                        // the fetched data to fill in UI widgets.
                        setField(result); } });
            } }).start();
}
```