CS302 Spr'99 Lecture Notes
Lecture 8

Runtime Environments

• Data Representation (covered in CS301)

• Storage Organization

• Procedures (& Stacks)

- Activation Records

- Access to non-local names

- Parameter Passing

- Procedures as first-class values

• Storage Allocation

- Static/Stack/Heap

- Garbage Collection

Storage Organization

• Subdivide machine address space by function, access, allocation.

• Typical organization (Unix)

```
∞ ---------------
  |             |
  |   STACK     |
  |             |
  ---------------  "top" of stack
  |     |       |     (but stacks nearly always
  |     |       |          grow DOWN!)
  |     ↓       |
  |             |  (in VM systems, these pages
  |             |     don't actually exist)
  |     ↑       |
  |     |       |
  |     |       |
  ---------------  (controlled by  sbrk system call)
  |             |
  |    HEAP     |  Managed by allocator/collector.
  |             |
  ---------------------
  |             |  Uninitialized ("b[lock] s[tatic] s[torage]")
  | STATIC DATA |------
  |             |  Initialized (in object file)
  ---------------------
  |             |
  |   CODE      |  Read-only (in object-file)
  |             |
0 ---------------
```


Storage Classes

Static Data : Permanent Lifetimes

• Global variables and constants.

• Allows fixed address to be compiled into code.

• No runtime management costs.

• Original FORTRAN (no recursion) used static activation records.

Stack Data : Nested Lifetimes

• Allocation/deallocation is cheap (just adjust stack pointer).

• Most architectures support cheap `sp`-based addressing.

• Good **locality** for VM systems, caches.

• C, Algol family (including PCAT) use stack for activation records.

Heap Data : Arbitrary Lifetimes

• Requires explicit allocation and (dangerous) explicit deallocation or garbage collection.

• Lisp, ML, many interpreted languages need heap for activation records, which have non-nested lifetimes.

Procedures and Activations

• A procedure **definition** associates a **name** with a procedure `body` and associated **formal parameters**.

• A procedure **activation** is created during execution when the procedure is called (with **actual parameters**.

• Activations have **lifetimes**: the time between execution of the first and last statements in the procedure.

• Activations are either **nested** (e.g., a,b) or **non-overlapping** (e.g., b,c):

```
a() {                       a
  b();                     / \
  c();                    /   \
}                        b     c
                               |
c() {                          |
  d();                         d
}
```

• Procedure `f` is **recursive** if two or more activations of `f` are **nested**. (Note that `f` need not call itself directly.)

Activation Trees - Example

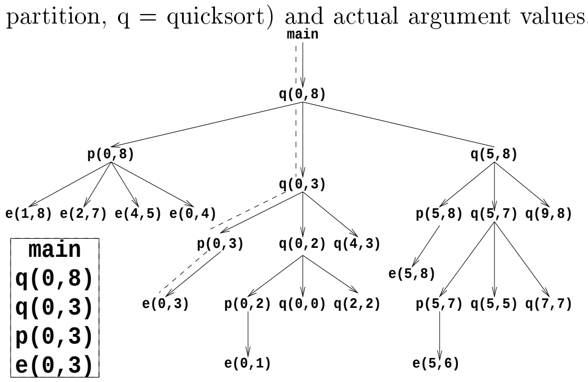
```
int a[9] = {10,32,567,-1,789,3,18,0,-51};

main() {
  quicksort(0,8);
}

void exchange(int i, int j) {
  int x = a[i]; a[i] = a[j]; a[j] = x;
}

int partition(int y, int z) {
  int i = y, j = z + 1;
  while (i < j) {
    i++; while (a[i] < a[y]) i++;
    j--; while (a[j] > a[y]) j--;
    if (i < j) exchange(i,j);
  };
  exchange(y,j);
  return j;
}

void quicksort(int m, int n) {
  if (n > m) {
    int i = partition(m,n);
    quicksort(m,i-1);
    quicksort(i+1,n);
  };
}
```


Activation Trees - Example (Cont.)

(A.k.a. "**call tree**")

Execution corresponds to depth-first traversal of tree.

Identify activations by name of function (e = exchange, p = partition, q = quicksort) and actual argument values.



Control stack keeps track of live activations; it contains activations along the path from root to "current" activation (see example above).

Activation Records (a.k.a. "Frames")

Contain **data** associated with a particular activation of a procedure:

• Actual parameters (maybe in registers).

• Return value (maybe in register).

• Local variables, including temporaries (perhaps containing saved registers).

• Access (or Static) link = pointer to **statically enclosing** activation record (to access non-local variables, if needed).

Also convenient to include **control** information about the calling procedure:

• Return address in caller.

• Control (or Dynamic) Link = pointer to **caller**'s activation record (if needed).

Use **fixed layout** (as far as possible) for activation records, so contents can be referenced as:

```
(frame pointer) + (statically-known offset)
```

Most architectures perform such references efficiently.

Activation Record Lifetimes

The **lifetime** of an activation record corresponds to the longest lifetime of anything contained in it.

The lifetimes of all contents begin when the activation begins (i.e., when the procedure is called).

The lifetime of **control** information ends when the activation's lifetime ends (i.e., when the procedure returns).

For most conventional languages, including C, Pascal, PCAT, etc., the lifetimes of local **data** are also contained within the **activation**'s lifetime.

Thus, since activation lifetimes behave in a stack-like manner, we can allocate and deallocate activation records on a **stack**.

(For some languages, like Lisp, ML, etc., data lifetimes don't obey these rules; such data cannot be stack-allocated. More later.)

We don't "push" and "pop" whole activation records; instead, we build and destroy them in pieces.

Calling Sequence

= Sequence of steps that, taken together, build and destroy activation records.

Typically divide into:

- Call sequence – performed by caller.
- Entry sequence – performed by callee.
- Exit sequence – performed by either/both.

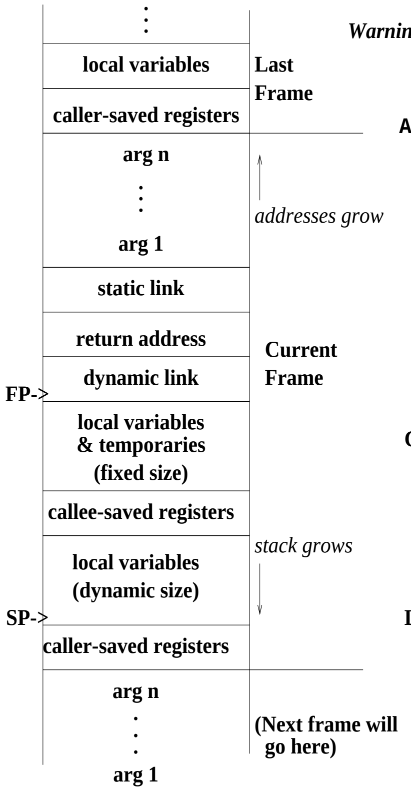
Key problem: caller and callee know very little about each other.

- Single callee may have many callers.
- Caller may not callee statically (e.g., C function pointers).
- Caller shouldn't need to know details of callee's implementation (may be compiled separately).

Thus, caller and callee must blindly cooperate via a set of **conventions**.

Typical Activation Records (e.g., X86)

Warning: NOT like SPARC!



Addressing

```
arg_i = M[fp+2+i]
local_j = M[fp-j]
old fp = M[fp]
old pc = M[fp+1]
```

Common variations:

- **order of info**
- **defn. of "frame"**
- **use of dynamic sizing**

Depends on:

- **hardware**
- **convention**

Typical Calling Sequence (e.g., X86)

1. Caller pushes caller-save registers.

2. Caller pushes arguments (in reverse order) and static link (if any).

3. Caller executes `call` instruction, which pushes `pc` (details vary according to machine architecture).

4. Callee pushes `fp` as dynamic link and sets `fp = sp`.

5. Callee adjusts `sp` to make room for fixed-size locals.

6. Callee pushes callee-save registers.

7. Callee can adjust `sp` dynamically during procedure execution to allocate dynamically-sized data on the stack.

Typical Return Sequence (e.g., X86)

1. Callee restores callee-save registers.

2. Callee resets `sp = fp`, thereby popping locals and any dynamically-sized data.

3. Callee pops dynamic link into `fp`.

4. Callee does a `return`, which pops return address into `pc`.

5. Caller pops static link and args.

6. Caller restores caller-save registers.

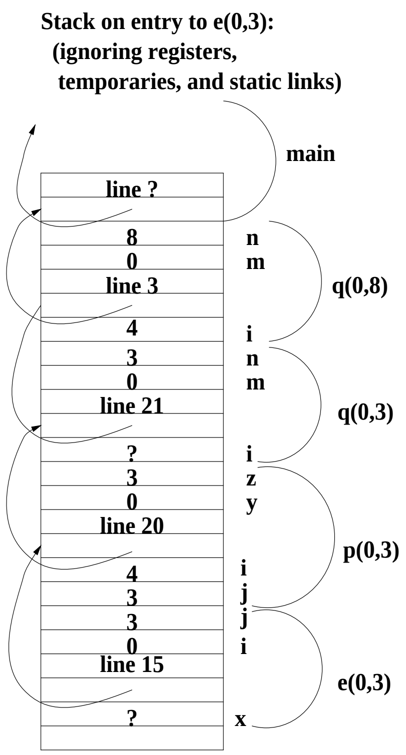
Common variations

• Hardware instructions do more or less.

• Arguments may be passed in registers. (Return value almost always is.)

• If everything is fixed-size, there's no real need for a frame pointer, or a dynamic link – but still handy.

Typical Sequence - Example (Quicksort)



Access to Non-local Variables

• In Pascal, Ada, PCAT, etc., we can nest procedure declarations **inside** other procedure declarations. (Cannot do this in C!)

• Parameters and local variables of outer procedures are visible within inner procedures.

• More precisely, the variables associated with the **most recent** still-live activation of the outer procedure are visible within inner procedures.

• References to these variables must be compiled to code that can locate the corresponding values at runtime.

• Any variables that are referenced non-locally must be stored in activation records in memory; they cannot be held just in registers.

• Can analyze program to tell which variables are so referenced (or, as we'll do for PCAT, just assume the worst).

• In most languages, if procedure `f` is declared inside `g`, then `f` can only appear as descendent of `g` in the activation tree. This allows us to stack-allocate activation records, and still guarantee that non-local variables will still exist when they are needed.

Example: Quicksort Revisited (in PCAT)

```
1  PROGRAM (* 1 *) IS
2    TYPE IARRAY IS ARRAY OF INTEGER;
3    VAR a := IARRAY[<9 OF 0>];b := IARRAY[<99 OF 0>];
4    PROCEDURE sort (* 2 *) (n:INTEGER;  a: IARRAY) IS
5      PROCEDURE exchange (* 3 *) (i,j: INTEGER) IS
6        BEGIN
7          x :=  a[i];  a[i] :=  a[j];  a[j] := x;
8        END;
9      PROCEDURE quicksort (* 3 *) (m,n: INTEGER) IS
10       VAR i : INTEGER := 0;
11       PROCEDURE partition(*4*)(y,z:INTEGER):INTEGER IS
12         VAR i,j: INTEGER := 0;
13         BEGIN
14           ...
15           WHILE ( a[i] <  a[y]) DO i := i + 1 END;
16           ...
17           exchange(y,j);
18           RETURN j;
19         END;
20       BEGIN
21         IF n > m THEN
22           i := partition(m,n);
23           quicksort(m,i-1);
24           quicksort(i+1,n);
25         END;
26       END;
27     BEGIN
28       quicksort(0,n-1);
29     END;
30  BEGIN
31     sort(9,a);
32     sort(99,b);
33  END;
```


Access Links for Non-local Variables

• Each activation record can include an **access link** (a.k.a. **static link**) pointing to the statically enclosing activation record.

• If `p` is nested immediately inside `q`, then the access link in `p`'s activation record points to the activation record for the most recent live activation of `q`.

• Non-local `v` is found by following one or more access links to the activation record that contains `v`, and then taking the appropriate offset within that record.

• If `v` is declared locally at depth n_v, and accessed in `p` at depth n_p, then the number of access links to follow is just $(n_p - n_v)$.

• In general, variable locations can be described as a pair:

(number of access links to follow,
offset within resulting activation record)

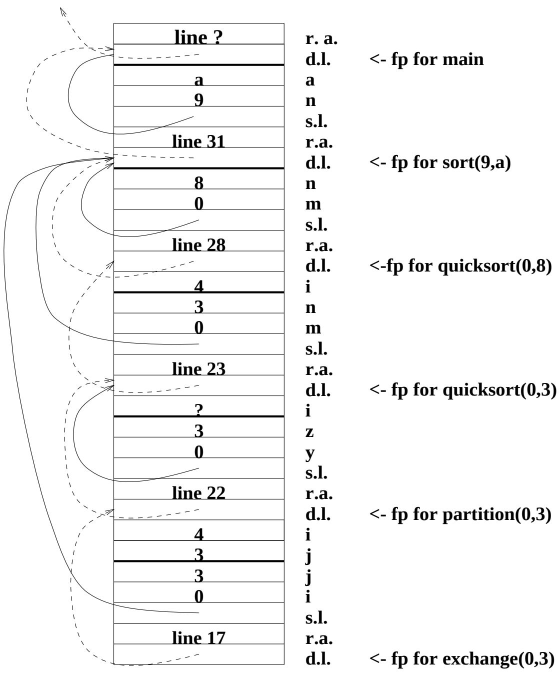
Local variable at offset z is represented as $(0, z)$.

These locations are fully known at `compile` time.

• Access links are initialized during the calling sequence; usually calculated by the caller and passed as a "hidden" first argument.

Example with Static Links: Quicksort

Stack on entry to e(0,3):



Note: A static link always points into an activation record at the same place as the frame pointer for that activation.

Another Addressing Example

```
PROCEDURE foo IS
  VAR x : INTEGER;
  PROCEDURE f (a:INTEGER) IS
    VAR y : INTEGER:
    PROCEDURE g (b: INTEGER) IS
      VAR z : INTEGER;
      BEGIN
        f(x+y+z);
        g(a+b);
      END;
  BEGIN
    g(a+y);
    f(x);
  END;
BEGIN
  f(x);
END;
```

If the body of `foo` is at scope level n, then:

- Symbol `foo` is at scope level $n-1$
- Symbols `x,f` are at scope level n
- Symbols `a,y,g` are at scope level $n+1$
- Symbols `b,z` are at scope level $n+2$

Make sure you can calculate the addressing code for each mention of each symbol.

SPARC Runtime Environment

• Somewhat different from "generic" architecture.

• Procedure **frame** contains procedure's own locals **and** arguments passed to called routines.

• Frame is normally **fixed-size**.

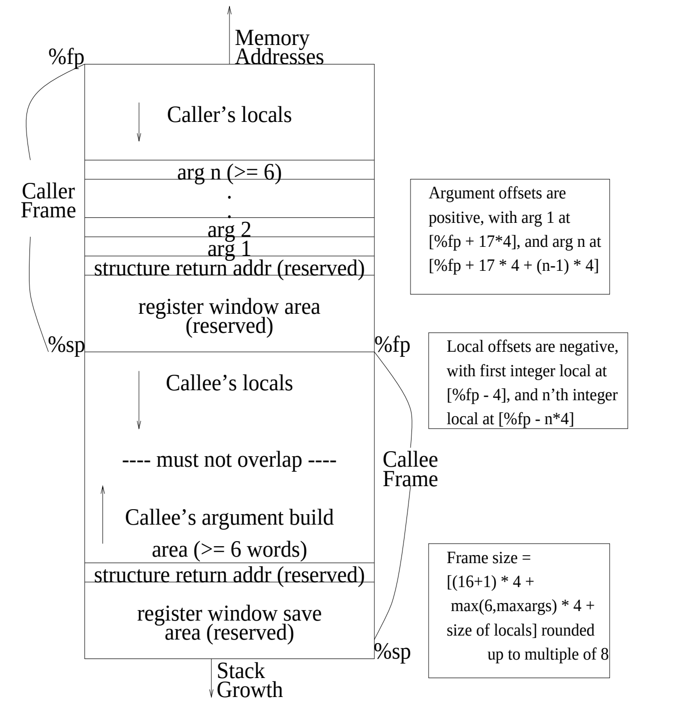
• Most register saving (including old `fp`,`sp`,`pc`) is taken care of by register windows.

• Arguments are normally passed in **registers** (up to 6); we won't do this for PCAT, however.

```
Register window shifts:

%o0     save -->        %i0
.                       .
.       <-- restore     .
.                       .
%o5                     %i5
%o6 = %sp         %fp = %i6
%o7 = saved pc caller = %i7
```

• None the less, **extra space** must be reserved in frame for registers and arguments (due to system and C conventions).

SPARC/C Language Frame Layout



SPARC/C Calling Sequence

1. Caller puts arguments in `%o0`, `%o1`, …, `%o5`. (Arguments beyond 6th word are put in argument build area of frame.)

2. Caller executes `call` instruction, which jumps to label and stores old `pc` in `%o7`.

3. Callee executes `save` instruction, which shifts register window and allocates frame. (Small leaf procedures can avoid this.)

4. Callee optionally stores arguments into argument build area of **caller's** frame.

Return Sequence

1. Callee puts (integer) return value in `%i0`.

2. Callee executes `restore` instruction, which resets register window and deallocates frame.

3. Callee executes `ret` instruction, which jumps to `%i7+8` (just past delay slot in caller).

Floating Point arguments are spread over pairs of registers; f.p. return values are in `%f0`; all f.p. registers are **caller-save**.

Structures (may be multiple words)

Arguments are spread over multiple registers; return values are written to space allocated by the **caller**, and pointed to by `%sp+64`.