Procedures & Activation Records

SOME SLIDES ARE MODIFICATIONS OF THE ONES PROVIDED BY THE AUTHORS OF THE FOLLOWING BOOK.

Copyright 2010, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

SLIDES ABOUT FAC FUNCTION CALLS ARE FROM VITALY SHMATIKOV

Overview

Procedures are probably the most significant advance in compilers after the development of FORTRAN

Procedures enable structured programming

They enable programmers to develop and test parts of a program in isolation.

Procedures help define interfaces between system components; cross-component interactions are typically structured through procedure calls.

Overview

Procedures create a controlled execution environment.

Each procedure has its own private named storage.

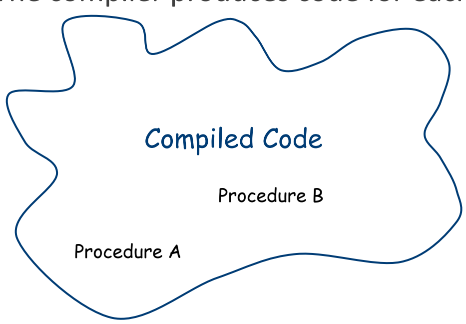
Statements executed inside the procedure can access the private, or local, variables in that private storage.

The procedure may return a value to its caller, in which case the procedure is termed a *function.*

Procedures

void function

The compiler produces code for each procedure



The individual code bodies must fit together to form a working program

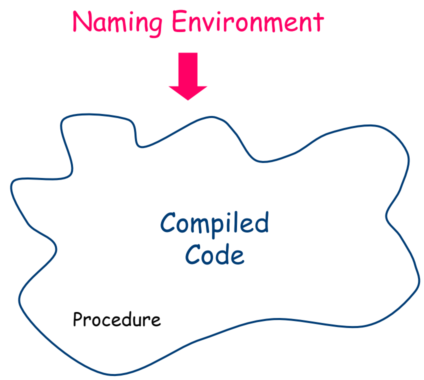
Procedures

Each procedure inherits a set of names
- Variables, values, procedures, objects, locations, …

Clean slate for new names

Local names may obscure identical, non-local names

Local names cannot be seen outside



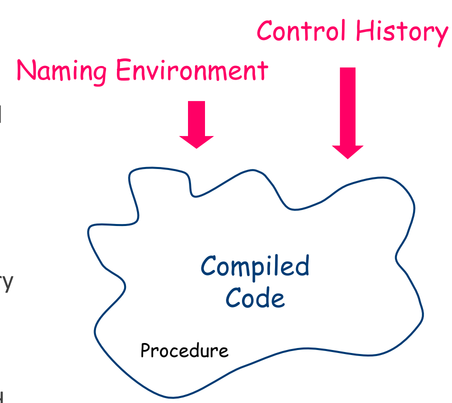
Procedures

Procedures have well defined entries and exits

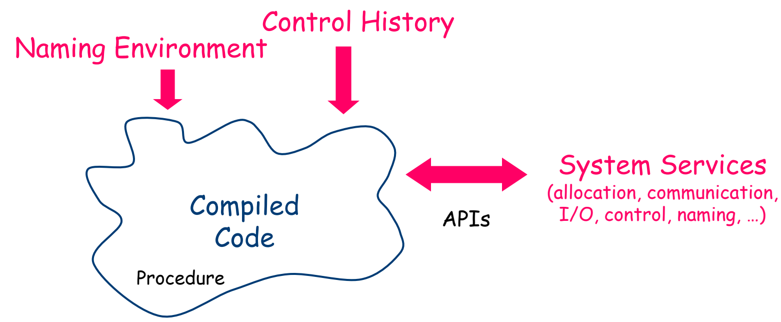
Each procedure inherits a control history
- Chain of calls that led to its invocation
- Mechanism to return control to calling procedure

In some languages, control history is a simple stack of activation records.

In Scheme and some other languages, it is more complicated due to closures and continuations.



Procedures



Each procedure has access to external interfaces

- Access by name, with parameters *(may include dynamic link & load)*

The Procedure as a Control Abstraction

A procedure is invoked at a call site, with some set of *actual parameters*

Control returns to call site, immediately after invocation

Most languages allow recursion

```
…
s = p(10,t,u);
…
```

```
int p(a,b,c)
  int a, b, c;
{
  int  d;
  d = q(c,b);
  …
}
```

```
int q(x,y)
  int x,y;
{
  if ( … )
    x = q(x-1,y);
  return x + y;
}
```

The Procedure as a Control Abstraction

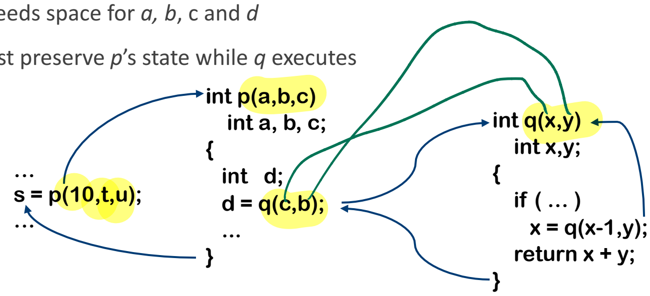
Need to save and restore a return address

Map actual parameters to formal parameters *(10→a, t→b, u→c)*

Must create storage for local variables

p needs space for *a, b*, c and *d*

Must preserve *p's* state while *q* executes



Activation Records

An *activation record* (AR) is a private block of memory associated with an invocation of a procedure.

It is a runtime structure used to manage a procedure call.

An AR is used to map a set of arguments, or parameters, from the caller's name space to the callee's name space.

An AR includes a mechanism to return control to the caller and continue execution at the point immediately after the call.

Most languages allow a procedure to return one or more values to the caller.

Activation Records

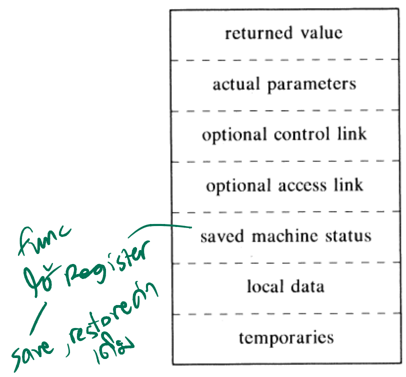
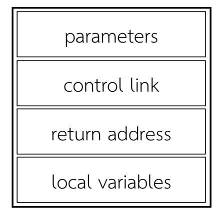
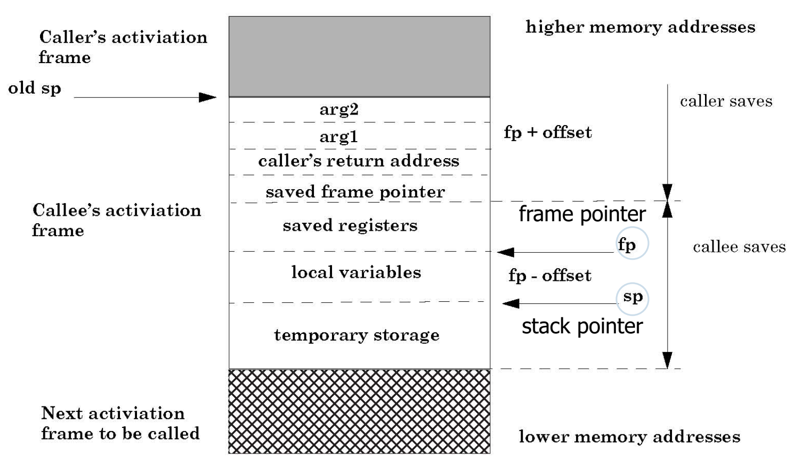


Fig. 7.8. A general activation record.



ระเบียนการทำงาน (activation record)

Typical x86 Activation Record



Activation Records

Caller View	Callee View	Contents	Frame
8(%rbp)		return address	
0(%rbp)		old rbp	
-8(%rbp)		local 1	Caller
...		...	
$-8k$(%rbp)		local k	
$8n-8$(%rsp)	$8n+8$(%rbp)	argument n	
	...	...	
0(%rsp)	16(%rbp)	argument 1	
	8(%rbp)	return address	
	0(%rbp)	old rbp	
	-8(%rbp)	local 1	Callee
	...	...	
	$-8m$(%rsp)	local m	

Figure 6.4: Memory layout of caller and callee frames.

Creating and Destroying Activation Records

A procedure call must allocate and initialize an AR to preserve it's own state.

Upon returning from a procedure, it must dismantle it's own environment and restore the caller's state.

Caller and called procedure must collaborate on the problem

- Caller knows some of the necessary state:
  - Return address, parameter values, access to other scopes
- Called procedure knows the rest:
  - Size of local data area (with spills), registers it will use

Division of Tasks between Caller and Callee

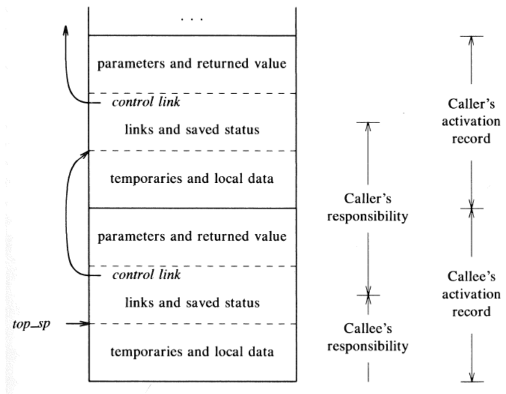
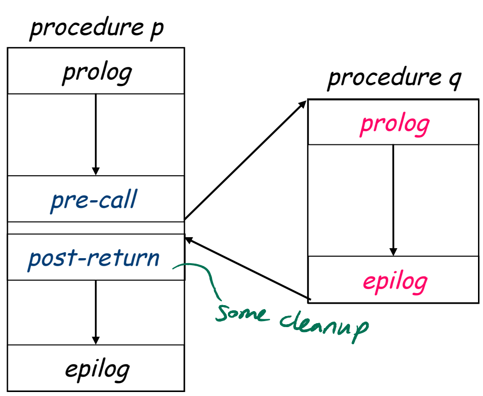


Fig. 7.14. Division of tasks between caller and callee.

Procedure Linkages

Standard Procedure Linkage



Procedure has
- standard prolog
- standard epilog

Each call involves a
- pre-call sequence
- post-return sequence

These are completely predictable from the call site ⇒ depend on the number & type of the actual parameters

Procedure Linkages: Pre-call

Purpose:

Sets up the called procedure's basic activation record

It helps preserve its own environment — save & restore ค่า var ของ caller

Details:

Allocate space for the called procedure's activation record

- except space for local variables

Store values of arguments in the parameters section.

Save return address

If access links are used

- Find appropriate lexical ancestor and copy into AR

Save any caller-save registers — caller → callee

Jump to address of called procedure's prolog code

Procedure Linkages: Prolog

Purpose:

Finish setting up called procedure's environment

Preserve parts of caller's environment that will be disturbed

Details:

Preserve any called procedure-save registers

Allocate space for local data

- Easiest scenario is to extend the AR

Find any static data areas referenced in the called procedure.

Handle any local variable initializations — declare ไว้โดยบอกค่า init ให้ default val

Saving Registers

Who saves the registers? Caller or called procedure?

- Caller knows which values are LIVE across the call
- Called procedure knows which registers it will use

Conventional wisdom: divide registers into three sets

- Caller saves registers
  - Caller targets values that are not LIVE across the call
- Called proc. saves registers
  - Called proc. only uses these AFTER filling caller saves registers
- Registers reserved for the linkage convention
  - ARP, return address (if in a register), …

Procedure Linkages: Epiloge

Purpose:

Start restoring the caller's environment restore registers

Details:

Store return value.

Restore called procedure-save registers

Free space for local data, if necessary (on the heap)

Load return address from AR

Restore caller's ARP

Jump to the return address

Procedure Linkages: Post-return

อาจเป็น epilog ทำ ½ แล้ว caller ทำอีก ½

Purpose:

Finish restoring caller's environment

Place any value back where it belongs

Details:

Copy return value from called procedure's AR, if necessary

Free the called procedure's AR

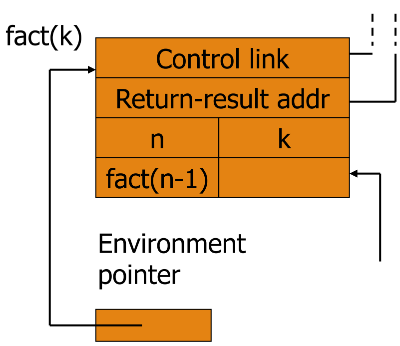
Restore any caller-save registers

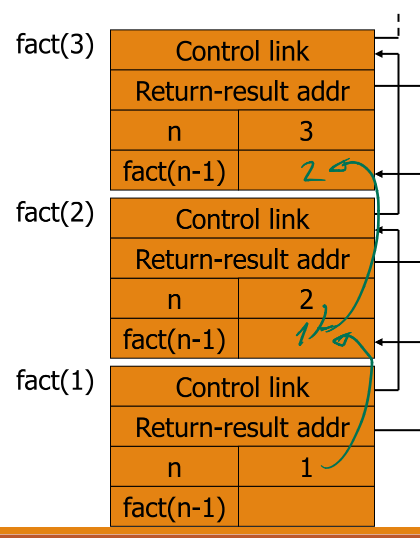
Restore any call-by-reference parameters to registers, if needed

- Also copy back call-by-value/result parameters

Continue execution after the call

Function Call

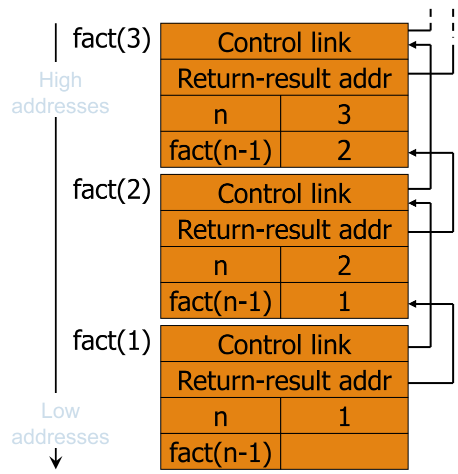




fact(n) = if n<= 1 then 1
else n * fact(n-1)

Return address omitted; would be a pointer into code segment

Function Return



Placing Run-time Data Structures

- A virtual address space
- Code, static and global data have known size
- Heap and stack both grow and shrink over time
- Better utilization if stack and heap grow toward each other

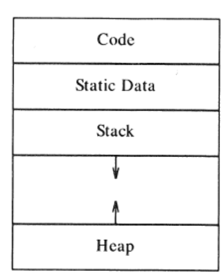


Fig. 7.7. Typical subdivision of run-time memory into code and data areas.

24

Activation Record Details

Where do activation records live?

If lifetime of AR matches lifetime of invocation and

If code normally executes a "return"

⇒Keep ARs on a stack

If a procedure can outlive its caller or

If it can return an object that can reference its execution state

⇒ARs must be kept in the heap

If a procedure makes no calls

⇒AR can be allocated statically

เก็บ global ได้หมด

Where to put Variables?

Where do variables live?

Locals and parameters ⇒ in procedure's activation record (AR)

Static (at any scope) ⇒ in a named static data area

- Procedure scope ⇒ name a storage area for the procedure
  - &_p.x for variable x in procedure p
- Class scope ⇒ name a storage area for class name

Dynamic (at any scope) ⇒ on the heap

Global

- One or more named global data areas
- One per variable, or per file, or per program, …

If lifetime does not match procedure's lifetime, then allocate it on the heap

Variable length items?

Put a descriptor in the "natural" location

Allocate actual item at end of AR or in the heap

Storage for Blocks within a Single Procedure

```
B0: {
        int a, b, c
B1:     {
            int v, b, x, w
B2:         {
                int x, y, z
                …
            }
B3:         {
                int x, a, v
                …
            }
            …
        }
        …
    }
```

Fixed length data can always be at a constant offset from the beginning of a procedure's data area

- In our example, the *a* declared at level 0 will always be the first data element, stored at byte 0 in the fixed-length data area
- The *x* declared at level 1 will always be the sixth data item, stored at byte 20 in the fixed data area
- The *x* declared at level 2 will always be the eighth data item, stored at byte 28 in the fixed data area
- But what about the a declared in block B3, the second block at level 2?

a	b	c	v	b	x	w	x	y	z

L_0 L_1 L_2

Storage in block B2

Variable-length Data

vargs

```
B0: {   int a, b
        …
        assign value to a
        …
B1:     {  int v(a), b, x
           …
B2:        {  int x, y(8)
              …
           }
        }
    }
```

Arrays

→If size is fixed at compile time, store in fixed-length data area

→If size is variable, store descriptor in fixed length area, with pointer to variable length area

→Variable-length data area is assigned at the end of the fixed length area for the block in which it is allocated (including all contained blocks)

printf 32 params
ใช้หลายตัว

a	b	v	b	x	x	y(8)	v(a)

Includes fixed length data for all blocks in the procedure …

Variable-length data area

Establishing Addressability

Local variables
- Convert to static data coordinate and use ARP + offset

Global & static variables
- Construct a label by mangling names (*i.e.,* &_fee)

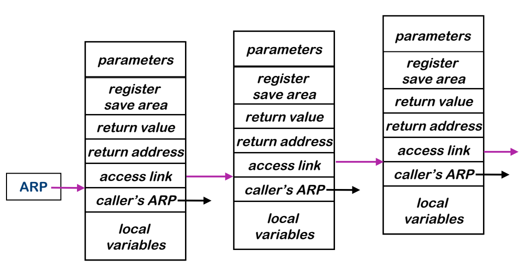
Local variables of other procedures
- Convert to static coordinates
- Find appropriate ARP
- Use that ARP + offset

Establishing Addressability

Each AR has a pointer to AR of lexical ancestor

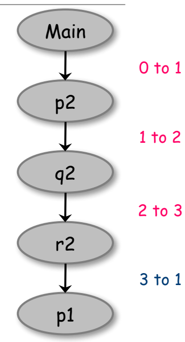
Lexical ancestor need not be the caller

Cost of access is proportional to lexical distance



Finding the "Right" ARP

```
procedure main {
   procedure p1 { … }
   procedure p2 {
      procedure q1 { … }
      procedure q2 {
         procedure r1 { … }
         procedure r2 {
            call p1; … // call from level 3 to level 1
         }
         call r2;        // call from level 2 to level 3
      }
      call q2;        // call from level 1 to level 2
   }
   call p2;           // call  from level 0 to level 1
}
```

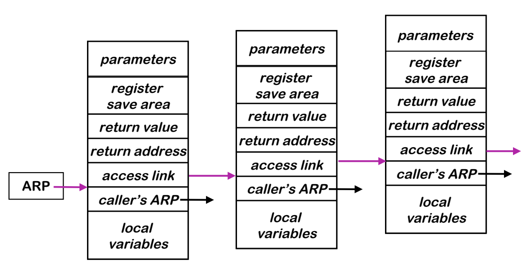


Call History

If the call is to a level greater than the current, then the called procedure must be nested within the calling procedure.

If the call is to a level smaller than the current, then the called procedure must be nested within the containing procedure (i.e. is a lexical ancestor)

Establishing Addressibility



- If the call is to level greater than the current: Use caller's ARP link
- If the call is to level smaller than the current: Use access link to lexical ancestor

Translating Local Names

How does the compiler represent a specific instance of *x* ?

Name is translated into a *static coordinate*

- < *level*, *offset* > pair
- *"level"* is lexical nesting level of the procedure
- *"offset"* is *unique* within that scope

Subsequent code will use the static coordinate to generate addresses and references

"level" is a function of the table in which *x* is found

- Stored in the entry for each *x*

"offset" must be assigned and stored in the symbol table

- Assigned at *compile time*
- Known at *compile time*
- Used to generate code that *executes* at *run-time*

Establishing Addressability

Access & maintenance cost varies with level

All accesses are relative to ARP (r_0)

Static Coordinate	*Generated Code*	
<2,8>	loadAI r_0,8	$\Rightarrow r_{10}$
<1,12>	loadAI r_0,-4	$\Rightarrow r_1$
	loadAI r_1,12	$\Rightarrow r_{10}$
<0,16>	loadAI r_0,-4	$\Rightarrow r_1$
	loadAI r_1,-4	$\Rightarrow r_1$
	loadAI r_1,16	$\Rightarrow r_{10}$

Assume
- Current lexical level is 2
- Access link is at ARP – 4
- ARP is in r_0