编译原理/Compiler
Activation Records

周雅倩
复旦大学计算机科学技术学院
zhouyaqian@fudan.edu.cn
2018/11/2

References

- http://www.stanford.edu/class/cs143/
  - Partial contents are copied from the slides of Alex Aiken

复旦大学媒体计算研究所

2

Status

- We have covered the front-end phases
  - Lexical analysis
  - Parsing
  - Semantic analysis
- Next are the back-end phases
  - Optimization
  - Code generation
- We'll do code generation first . . .

复旦大学媒体计算研究所

3

Outline

- Run-time Environments
- Parameter Passing
- Alignment
- A Real Example
- Nested Procedures

复旦大学媒体计算研究所

4

Run-time Environments

5

Run-time environments

- Before discussing code generation, we need to understand what we are trying to generate
- There are a number of standard techniques for structuring executable code that are widely used

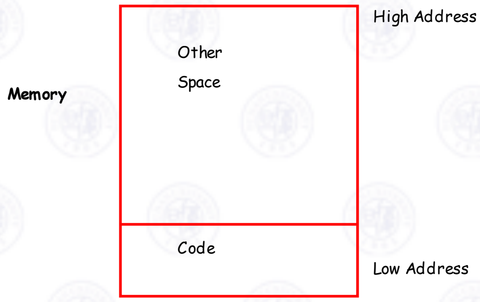
复旦大学媒体计算研究所

6

Run-time Resources

- Execution of a program is initially under the control of the operating system
- When a program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of the space
  - The OS jumps to the entry point (i.e., “main”)

Memory Layout



What is Other Space?

- Holds all data for the program
- Other Space = Data Space
- Compiler is responsible for:
  - Generating code
  - Orchestrating use of the data area

Code Generation Goals

- Two goals:
  - Correctness
  - Speed
- Most complications in code generation come from trying to be fast as well as correct

Assumptions about Execution

- Execution is sequential; control moves from one point in a program to another in a well-defined order
- When a procedure is called, control eventually returns to the point immediately after the call
- Do these assumptions always hold?

Activations

- An invocation of procedure P is an activation of P
- The lifetime of an activation of P is
  - All the steps to execute P
  - Including all the steps in procedures that P calls

复旦大学媒体计算研究所

Lifetimes of Variables

- The **lifetime** of a variable **x** is the portion of execution in which **x** is defined
- Note that
  - – **Lifetime** is a dynamic (run-time) concept
  - – **Scope** is a static concept

13

复旦大学媒体计算研究所

Activation Trees

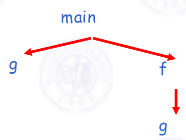
- Assumption (2) requires that **when P calls Q, then Q returns before P does**.
- Lifetimes of procedure activations are properly nested
- Activation lifetimes can be depicted as a tree

14

复旦大学媒体计算研究所

Example 1(in java)

```
class examaple1 {
    void g() { … }
    void f() { g(); }
    void static main(){{ g(); f(); }}
}
```



15

复旦大学媒体计算研究所

How to Record the activation?

```
class examaple1 {
    void g() { … }
    void f() { g(); }
    void static main(){{ g(); f(); }}
}
```

Main

Stack

Main

16

复旦大学媒体计算研究所

Example 1

```
class examaple1 {
    void g() { … }
    void f() { g(); }
    void static main(){{ g(); f(); }}
}
```

Stack

Main

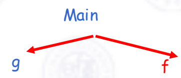
g

17

复旦大学媒体计算研究所

Example 1

```
class examaple1 {
    void g() { … }
    void f() { g(); }
    void static main(){{ g(); f(); }}
}
```



Stack

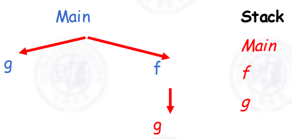
Main

f

18

Example 1

class examaple1 {
void g() { … }
void f() { g(); }
void static main(){{ g(); f(); }}
}



复旦大学媒体计算研究所

19

Notes

- The activation tree depends on run-time behavior
- The activation tree may be different for every program input
- Since activations are properly nested, a stack can track currently active procedures

复旦大学媒体计算研究所

20

Example 2 (in Java)

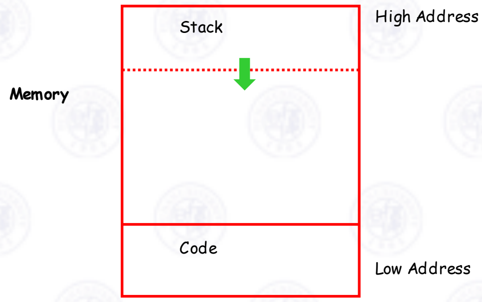
class examaple2{
int g() { … };
int f(int x) { if (x == 0) g(); else f(x - 1);}
void static main() {f(3); }
}

- What is the activation tree for this example?

复旦大学媒体计算研究所

21

Memory Layout



复旦大学媒体计算研究所

22

Activation Records

- On many machine the stack starts at high-addresses and grows towards lower addresses
- The information needed to manage one procedure activation is called an activation record (AR，活动记录) or frame
- If procedure F calls G, then G's activation record contains a mix of info about F and G.

复旦大学媒体计算研究所

23

What is in G's AR when F calls G?

- F is "suspended" until G completes, at which point F resumes.
- G's AR contains information needed to resume execution of F.
- G's AR may also contain:
  – Actual parameters to G (supplied by F)
  – G's return value (needed by F)
  – Space for G's local variables

复旦大学媒体计算研究所

24

The Contents of a Typical AR for G

- Space for G's return value
- Actual parameters
- Pointer to the previous activation record
  - The control link points to AR of caller of G
- Machine status prior to calling G
  - Contents of registers & program counter
  - Local variables
- Other temporary values

25

复旦大学媒体计算研究所

Example 2, Revisited

```
class examaple2{
    int g() { … };
    int f(int x) { if (x == 0) g(); else f(x - 1);**}
    void static main() {f(3);* }
}
```

- AR for f:

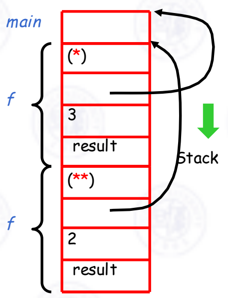
return address
control link
argument
result

- (*) and (**) are return addresses of the invocations of f
- The return address is where execution resumes after a procedure call finishes

26

复旦大学媒体计算研究所

Stack After Two Calls to f



Notes:
- main has no argument or local variables and its result is never used; its AR is uninteresting
- This is only one of many possible AR designs
  - Would also work for C, Pascal, FORTRAN, etc.

27

复旦大学媒体计算研究所

The Main Point

- The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record
- Thus, the AR layout and the code generator must be designed together!

28

复旦大学媒体计算研究所

Discussion

- There is nothing magic about this organization
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities differently
  - An organization is better if it improves execution speed or simplifies code generation
- For example:
  - The advantage of placing the return value 1st in a frame is that the caller can find it at a fixed offset from its own frame.

29

复旦大学媒体计算研究所

Discussion (Cont.)

- Real compilers hold as much of the frame as possible in registers
  - Especially the method result and arguments

30

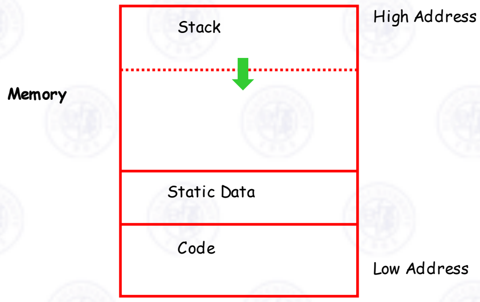
Globals

- All references to a global variable point to the same object
  - Can't store a global in an activation record
- Globals are assigned a fixed address once
  - Variables with fixed address are "statically allocated"
- Depending on the language, there may be other statically allocated values

31

复旦大学媒体计算研究所

Memory Layout with Static Data



32

复旦大学媒体计算研究所

Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR
- method foo() { new Bar }
  - The Bar value must survive deallocation of foo's AR
- Languages with dynamically allocated data use a heap to store dynamic data

33

复旦大学媒体计算研究所

Notes

- The code area contains object code
  - For most languages, fixed size and read only
- The static area contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
- The stack contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals
- Heap contains all other data
  - In C, heap is managed by malloc and free

34

复旦大学媒体计算研究所

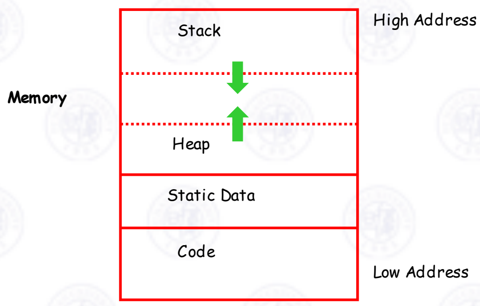
Notes (Cont.)

- Both the heap and the stack grow
- Must take care that they don't grow into each other
- Solution: start heap and stack at opposite ends of memory and let the grow towards each other

35

复旦大学媒体计算研究所

Memory Layout with Heap



36

复旦大学媒体计算研究所

PARAMETER PASSING

复旦大学媒体计算研究所

Parameter Passing

- Call-by-Value
- Call-by-Reference
- Call-by-Restore
- Call-by-Name

复旦大学媒体计算研究所

Call-by-Value

- A formal parameter is treated just like a local name
- The storage for the formals is in the activation record of the called procedure
- The caller evaluates the actual parameters and places their r-values in the storage for the formals

复旦大学媒体计算研究所

Call-by-Reference

- Call-by-address
- If an actual parameter is a name or an expression having an l-value, then that l-value itself is passed
- If the actual parameter is an expression that has no l-value, then the expression is evaluated in a new location and the address of that location is passed

```
int& fun(int& a) //call by reference
{
    a += 5;
    return a;
}
```


复旦大学媒体计算研究所

Call-by-Restore

- A hybrid between call-by-value and call-by-reference
- Copy-restore linkage, copy-in copy-out, value-result

复旦大学媒体计算研究所

Call-by-Restore

- Before control flows to the called procedure
  - The actual parameters are evaluated
  - The r-values of the actuals are passed to the called procedure
  - l-values of those actual parameters having l-values are determined before the call
- When control returns, the current r-values of the formal parameters are copied back into the l-values of the actuals, using the l-values computed before the call

Call-by-Name

- The procedure is treated as if it were a macro
  - Macro-expansion
  - In-line expansion
- The local names of the called procedure are kept distinct from the names of the calling procedure
- The actual parameters are surrounded by parentheses if necessary to preserve their integrity

43

复旦大学媒体计算研究所

ALIGNMENT

44

复旦大学媒体计算研究所

Data layout

- Data layout is done on the symbolic table
- There are two variables
  - Global
  - Local
- Data layout is done before generating code instructions
- When generating code alongside the AST, the variables can be replaced by their data layout representation

45

复旦大学媒体计算研究所

Global data layout

```
int i;

main()
{
   i = 1 ;
}
```

46

复旦大学媒体计算研究所

Global data layout

```
movl      $1, _i

.comm   _i, 16   #4
```

- A module is generated, global variable is given a new name
- It is very easy to be done by scanning the global symbolic table

47

复旦大学媒体计算研究所

Local data layout

```
main()
{
  int i ;
  i = 1 ;
}

movl   $1, -4(%ebp)
```

- The local variable is placed in the AR
- All the references to i, is replaced by -4(%ebp) that is a memory address calculated by the offset and FR
- The offset is computed by the compiler when scanning the local symbolic tables

48

复旦大学媒体计算研究所

Computing the data size

- The size of most data type is very easy to calculate
- The structure data type is difficult in some extent, however it is definite

49

复旦大学媒体计算研究所

Data Layout

- Low-level details of machine architecture are important in laying out data for correct code and maximum performance
- Chief among these concerns is alignment

50

复旦大学媒体计算研究所

Alignment restrictions

- The address for some type of object must be a multiple of some value k (typically 2, 4, or 8)
- Simplify the hardware design of the interface between the processor and the memory system
- In IA32
  - hardware will work correctly regardless of the alignment of data
  - Aligned data can improve memory system performance

51

复旦大学媒体计算研究所

Linux alignment restriction

- 1-byte data types are able to have any address
- 2-byte data types must have an address that is multiple of 2
- Any larger data types must have an address that is multiple of 4
- Example:

```
struct test 1
{
    char x1;
    short x2;
    float x3;
    char x4;
};
```

total 12 bytes

```
struct test2
{
 double x1;
 char x2;
 int x3;
};
```

total 16 bytes

52

复旦大学媒体计算研究所

Alignment

- Alignment is enforced by
  - Making sure that every data type is organized and allocated in such a way that every object within the type satisfies its alignment restrictions.
- malloc()
  - Returns a generic pointer that is void *
  - Its alignment requirement is 4

53

复旦大学媒体计算研究所

Alignment

- Structure data type
  - may need to insert gaps in the field allocation
  - may need to add padding to the end of the structure

54

Use of Registers

- To avoid memory traffic, modern compilers often pass arguments, return results, and allocate local variables in machine registers.
- Typical parameter-passing convention on modern machines: the first k arguments (k = 4 or 6) of a function are passed in registers $R_p, \ldots, R_{p+k-1}$, the rest are passed on the stack.

55

Use of Registers(cont.)

- Problem : extra memory traffic caused by passing args. in registers
  int g(int x, int y, int z){ return x*y*z;}
  int f(int x, int y, int z) {int a= g(z+3, y+3, x+4) ;
  return a*x+y+z;}
- Suppose function f and g pass their arguments in R1, R2, R3; then f must save R1, R2, and R3 to the stack frame before calling g

56

Frame Resident Variables

- Certain values must be allocated in stack frames because
  – the value is too big to fit in a single register
  – the variable is passed by reference --- must have a memory address
  – the variable is an array -- need address arithmetic to extract components
  – the register that the variable stays needs to be used for other purpose!
  – just too many local variables and arguments — there are not enough registers !!! -------- SPILLING

57

Frame Resident Variables(cont.)

- Open research problem: When to allocate local variables or passing arguments in registers ?
- Needs good heuristics

58

Linux in IA32

A REAL EXAMPLE

59

Procedure

- In high level programming languages
  – Parameter passing (actual vs. formal)
  – Return value
  – Control transfer
  – Allocate space for local variables on entry
  – Deallocate space for local variables on exit

60

Procedure Implementation

- Stack frame
- A fixed register is used to hold the return value
- Simple instructions for control transferring
- Register usage
- Calling convention

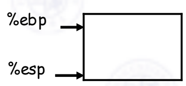
复旦大学媒体计算研究所

- Few registers:
  - General purpose 32bit: %eax, %ebx, %ecx, %edx, %edi, %esi, %ebp, %esp
- Many instructions:
  - Arithmetic: addl, subl, incl, modl, idivl, imull, etc.
  - Logic: and, orl, notl, xorl
  - Comparison: cmp, test
  - Control flow: jmp, jcc, jecz
  - Function calls: call, leave
  - Data movement: movb, movw, movl
  - Stack manipulations: pushl, popl
  - Other: leal

复旦大学媒体计算研究所

Frame Structure

- A stack frame is delimited by
  - The frame pointer %ebp
  - The stack pointer %esp

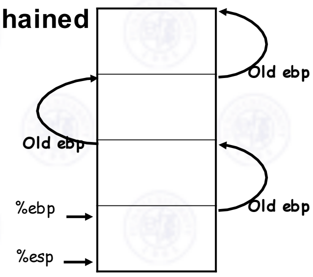


- The stack pointer can move when the procedure is executing
- The frame pointer relative accessing

复旦大学媒体计算研究所

Frame Structure

- Frames are chained



复旦大学媒体计算研究所

Calling Convention

- Caller and Callee

```
Caller()
{
    call callee (argument1, … , argumentn);
}
```

- Two ways:
  - Caller saved registers
  - Callee saved registers

复旦大学媒体计算研究所

Register Usage(1)

- Caller saved registers
  - %eax, %edx, %ecx
  - Saved by caller
  - Callee can use these registers freely
  - The contents in these registers may be changed after return
  - Caller must restore them if it tries to use them after calling

Register Usage(2)

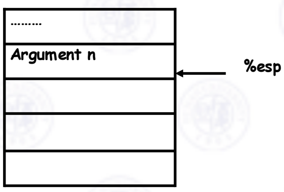
- Callee saved registers
  - %ebx, %esi, %edi
  - Callee must save them before using
  - Callee must restore them before return

Calling Convention

- Caller save registers (%eax, %edx, %ecx)
- Push actual arguments from right to left
- Call instruction
  - Save return address
  - Transfer control to callee

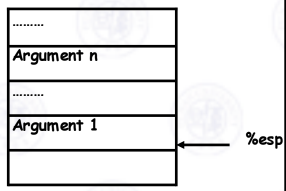
Passing Parameters

- pushl argument n



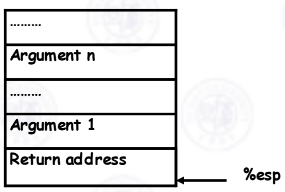
Passing Parameters

- pushl argument n
- ....
- pushl argument 1



Passing Parameters

- pushl argument n
- ....
- pushl argument 1
- call callee



Call Instruction

- call label (direct)
- call *operand (indirect)
- Save return address in the stack
- Jump to the entry to callee

Initialize Stack frame

- pushl %ebp
- movl %esp, %ebp
- save callee saved register
- adjust %esp to allocate space for local variables and temporaries

复旦大学媒体计算研究所

Destruct a Frame

restore callee saved registers
movl %ebp, %esp
popl %ebp
ret
Integeral return value is in the %eax

restore caller saved registers
leave
ret

复旦大学媒体计算研究所

Example

```
int swap_add(int *xp, int *yp)
{
        int x = *xp;
        int y = *yp;
        *xp = y;
        *yp = x;
        return x + y;
}
```


复旦大学媒体计算研究所

Example

```
int caller()
{
    int arg1 = 534;
    int arg2 = 1057;
    int sum = swap_add(&arg1, &arg2);
    int diff = arg1 - arg2;
    return sum * diff;
}
```


复旦大学媒体计算研究所

Example

Before
int sum = swap_add(&arg1, &arg2);

Stack frame for caller

0	Saved %ebp	←%ebp
-4	arg2(1057)	
-8	arg1(534)	←%esp
-12		

复旦大学媒体计算研究所

Parameter Passing

1 leal -4(%ebp), %eax
Compute &arg2
2 pushl %eax
Push &arg2

Stack frame for caller

0	Saved %ebp	←%ebp
-4	arg2(1057)	
-8	arg1(534)	
-12	&arg2	←%esp

Parameter Passing

复旦大学媒体计算研究所

1 leal -4(%ebp), %eax
Compute &arg2
2 pushl %eax
Push &arg2
3 leal -8(%ebp), %eax
Compute &arg1
4 pushl %eax
Push &arg1

Stack frame for caller

0	Saved %ebp	%ebp
-4	arg2(1057)	
-8	arg1(534)	
-12	&arg2	
-16	&arg1	%esp

79

Call Instruction

复旦大学媒体计算研究所

1 leal -4(%ebp), %eax
Compute &arg2
2 pushl %eax
Push &arg2
3 leal -8(%ebp), %eax
Compute &arg1
4 pushl %eax
Push &arg1
5 call swap_add
Call the swap_add function

Stack frame for caller

0	Saved %ebp	%ebp
-4	arg2(1057)	
-8	arg1(534)	
-12	&arg2	
-16	&arg1	
-20	Return Addr	%esp

80

Setup code in swap_add

复旦大学媒体计算研究所

swap_add:
1 pushl %ebp
Save old %ebp

Stack frame for caller

24	Saved %ebp	%ebp
20	arg2(1057)	
16	arg1(534)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	%esp

81

Setup code in swap_add

复旦大学媒体计算研究所

swap_add:
1 pushl %ebp
Save old %ebp
2 movl %esp, %ebp
Set %ebp as frame pointer

Stack frame for caller

24	Saved %ebp	%ebp
20	arg2(1057)	
16	arg1(534)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	%esp

Stack frame for swap_add

82

Setup code in swap_add

复旦大学媒体计算研究所

swap_add:
1 pushl %ebp
Save old %ebp
2 movl %esp, %ebp
Set %ebp as frame pointer
3 pushl %ebx
Save %ebx

Stack frame for caller

24	Saved %ebp	
20	arg2(1057)	
16	arg1(534)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	%ebp
-4	Saved %ebx	%esp

Stack frame for swap_add

83

Body code in swap_add

复旦大学媒体计算研究所

5 movl 8(%ebp), %edx
Get xp

%edx xp(=&arg1=%ebp+16)

Stack frame for caller

24	Saved %ebp	
20	arg2(1057)	
16	arg1(534)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	%ebp
-4	Saved %ebx	%esp

Stack frame for swap_add

84

复旦大学媒体计算研究所

Body code in swap_add

Stack frame for caller

5 movl 8(%ebp), %edx
Get xp
6 movl 12(%ebp), %ecx
Get yp

%edx xp(=&arg1=%ebp+16)
%ecx yp(=&arg2=%ebp+20)

24	Saved %ebp	
20	arg2(1057)	
16	arg1(534)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	←%ebp
-4	Saved %ebx	←%esp

Stack frame for swap_add

85

复旦大学媒体计算研究所

Body code in swap_add

Stack frame for caller

5 movl 8(%ebp), %edx
Get xp
6 movl 12(%ebp), %ecx
Get yp
7 movl (%edx), %ebx
Get x
8 movl (%ecx), %eax
Get y

%edx xp(=&arg1=%ebp+16)
%ecx yp(=&arg2=%ebp+20)
%ebx 534
%eax 1057

24	Saved %ebp	
20	arg2(1057)	
16	arg1(534)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	←%ebp
-4	Saved %ebx	←%esp

Stack frame for swap_add

86

复旦大学媒体计算研究所

Body code in swap_add

Stack frame for caller

9 movl %eax, (%edx)
Store y at *xp

%edx xp(=&arg1=%ebp+16)
%ecx yp(=&arg2=%ebp+20)
%ebx 534
%eax 1057

24	Saved %ebp	
20	arg2(1057)	
16	arg1(1057)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	←%ebp
-4	Saved %ebx	←%esp

Stack frame for swap_add

87

复旦大学媒体计算研究所

Body code in swap_add

Stack frame for caller

9 movl %eax, (%edx)
Store y at *xp
10 movl %ebx, (%ecx)
Store x at *yp

%edx xp(=&arg1=%ebp+16)
%ecx yp(=&arg2=%ebp+20)
%ebx 534
%eax 1057

24	Saved %ebp	
20	arg2(534)	
16	arg1(1057)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	←%ebp
-4	Saved %ebx	←%esp

Stack frame for swap_add

88

复旦大学媒体计算研究所

Body code in swap_add

Stack frame for caller

9 movl %eax, (%edx)
Store y at *xp
10 movl %ebx, (%ecx)
Store x at *yp
11 addl %ebx,%eax
Set return value = x+y

%edx xp(=&arg1=%ebp+16)
%ecx yp(=&arg2=%ebp+20)
%ebx 534
%eax 1591

24	Saved %ebp	
20	arg2(534)	
16	arg1(1057)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	←%ebp
-4	Saved %ebx	←%esp

Stack frame for swap_add

89

复旦大学媒体计算研究所

Finishing code in swap_add

Stack frame for caller

12 popl %ebx Restore %ebx

%edx xp(=&arg1=%ebp+16)
%ecx yp(=&arg2=%ebp+20)
%ebx original value
%eax 1591

24	Saved %ebp	
20	arg2(534)	
16	arg1(1057)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	←%ebp ←%esp
-4	Saved %ebx	

Stack frame for swap_add

90

复旦大学媒体计算研究所

Finishing code in swap_add

12 popl %ebx Restore %ebx
13 movl %ebp, %esp Restore %esp

%edx xp(=&arg1=%ebp+16)

%ecx yp(=&arg2=%ebp+20)

%ebx original value

%eax 1591

Stack frame for caller

24	Saved %ebp	
20	arg2(534)	
16	arg1(1057)	
12	yp(&arg2)	
8	xp(&arg1)	
4	Return Addr	
0	Saved %ebp	← %ebp
-4	Saved %ebx	← %esp

Stack frame for swap_add

复旦大学媒体计算研究所

Finishing code in swap_add

12 popl %ebx Restore %ebx
13 movl %ebp, %esp Restore %esp
14 popl %ebp Restore %ebp

%edx xp(=&arg1=%ebp+16)

%ecx yp(=&arg2=%ebp+20)

%ebx original value

%eax 1591

Stack frame for caller

0	Saved %ebp	← %ebp
-4	arg2(534)	
-8	arg1(1057)	
-12	yp(&arg2)	
-16	xp(&arg1)	
-20	Return Addr	← %esp
	Saved %ebp	
	Saved %ebx	

复旦大学媒体计算研究所

Finishing code in swap_add

12 popl %ebx Restore %ebx
13 movl %ebp, %esp Restore %esp
14 popl %ebp Restore %ebp
15 ret Return to caller

Call by value

%edx xp(=&arg1=%ebp+16)
%ecx yp(=&arg2=%ebp+20)
%ebx original value
%eax 1591

Stack frame for caller

0	Saved %ebp	← %ebp
-4	arg2(534)	
-8	arg1(1057)	
-12	yp(&arg2)	
-16	xp(&arg1)	← %esp
-20	Return Addr	
	Saved %ebp	
	Saved %ebx	

复旦大学媒体计算研究所

NESTED PROCEDURES

复旦大学媒体计算研究所

Scope Rule

- Lexical(static)-scope rule
  - Block(intro in chapter 8)
  - Nonlocal names in C
  - Nonlocal names in Pascal/ML
    - Static link & display

复旦大学媒体计算研究所

Lexical Scope without Nested Procedures

- A procedure definition cannot appear within another
- Name allocations
  - Stack allocation for local names
  - Static allocation for nonlocal names

Lexical Scope without Nested Procedures

- Declared procedures can freely be
  – passed as parameters
  – returned as results
- Any name nonlocal to one procedure is nonlocal to all procedures

Lexical Scope with Nested Procedures - Example (in Pascal)

(1) int a[11]
(2) readarray() { … a … }
(3) int partition(int y, int z) { … a … }
(4) quicksort(int m, int n) { … }
(5) main() { … a … }

Lexical Scope with Nested Procedures - Example (in Pascal)

```
program sort(input, output) ;
    var a: array[0..10] of integer ;
      x: integer ;
    procedure readarray ;
      var i : integer ;
      begin … a …  end { readarray } ;
    procedure exchange(i, j : integer) ;
      begin
          x: = a[i] ; a[i] := a[j] ; a[j] := x
      end { exchange } ;
```


Lexical Scope with Nested Procedures - Example (in Pascal)

```
  procedure quicksort(m,n  : integer) ;
      var k, v : integer ;
      function partition(y,  z: integer): integer ;
        var i, j : integer ;
        begin … a …
              … v …
              … exchange(i, j) ; …
        end { partition } ;
      begin  … end { quicksort }
  begin … end { sort }.
```


Lexical Scope with Nested Procedures - Example (in Pascal)

```
sort
    readarray
    exchange
    quicksort
        partition
```


Lexical Scope with Nested Procedures

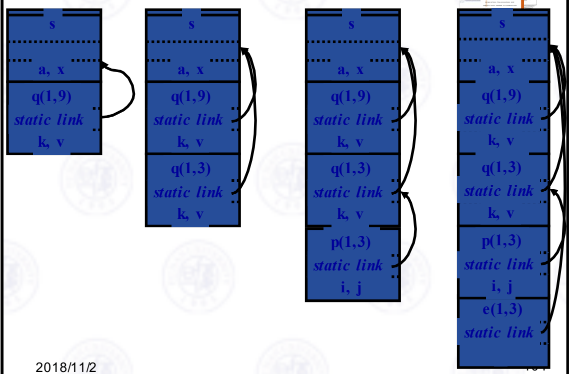
- Nesting depth
  – Main is at 1
  – Add 1 when going from an enclosing to an enclosed

复旦大学媒体计算研究所

Static Link

- Static link (also called access link) is used to implement lexical scoping.
- If function p is nested immediately within q in the source code, then the static link in activation of p is a point to the most recent activation of q.
- Non-local variable v is found by following static links to an activation (i.e, frame) that contains v
- If v is declared at depth nv and accessed in p declared at depth np, then we need follow np-nv static links

复旦大学媒体计算研究所



复旦大学媒体计算研究所

Access Nonlocal Variables

- Procedure p at nesting depth np
- Procedure p refers to a nonlocal v with nesting depth nv ≤ np
  - When control is in p,
    - An activation record for p is at the top of the stack
    - Follow np-nv static links from the record at the top of the stack.
  - After following np-nv links, we reach an activation record for the procedure that a is local to.
  - np-nv can be computed at compile time

复旦大学媒体计算研究所

Displays

- An array d of pointers to activation records
- Storage for a nonlocal a at nesting depth i is in the activation record pointed to by display element d[i]

复旦大学媒体计算研究所

Displays

- Suppose control is in an activation of a procedure p at nesting depth j
  - The first j-1 elements of the display point to the most recent activations of the procedures that lexically enclose procedure p
  - d[j] points to the activation of p
- Using a display is generally faster than following static links

复旦大学媒体计算研究所

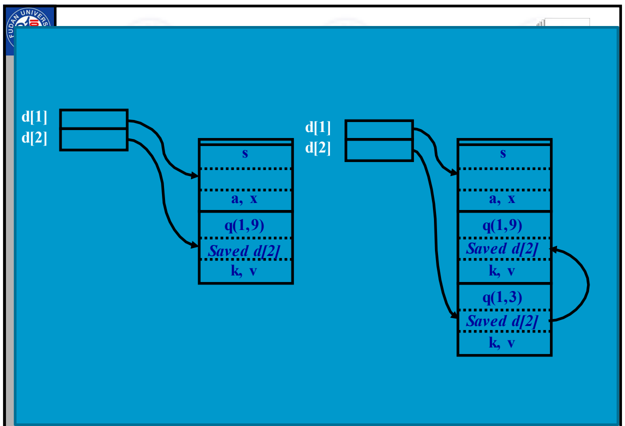
Displays

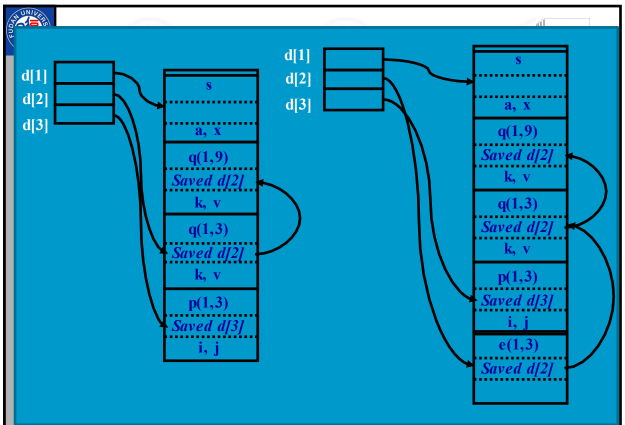
- The display changes when
  - A new activation occurs
  - Control returns from the new activation
- Simple arrangement
  - Uses static links
  - The display is updated by following the chain of static links

复旦大学媒体计算研究所

Displays

- A more efficient way
  - When a new activation record for a procedure at nesting depth i is set up
    - Save the value of d[i] in the new activation record
    - Set d[i] to point to the new activation record
  - Just before an activation ends
    - d[i] is reset to the saved value





复旦大学媒体计算研究所

Limitation of Stack Frames

- It does not support higher-order functions --- it cannot support "nested functions" and "procedure passed as arguments and results" at the same time.
  - C --- functions passed as args and results, but no nested functions;
  - PASCAL --- nested functions, but cannot be passed as args or res.
- Alternative to the standard stack allocation scheme ---
  - use a linked list of chunks to represent the stack
  - allocate the activation record on the heap --- no stack frame pop !
  - advantages: support higher-order functions and parallel programming well

复旦大学媒体计算研究所

Exercise

Describe the changes of stack frame during execution of the following program

```
int main()
{
  int arg1 = 2;
  int arg2 = 5;
  int sum = callee(arg1,
   &arg2);
  int diff = arg1 - arg2;
  return sum * diff;
}
```

```
int callee(intx, int *yp)
{
   int y = *yp;
   int count= 0;
   if(y>x) {
     int diff = y-x;
     *yp = diff;
    count= 1;
   }
   if(y>2*x){
     count+= callee(x,yp);
   }
   return count;
}
```