

Lecture 10 – Activation Records

# THEORY OF COMPILATION

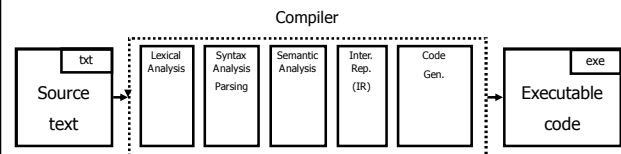
Eran Yahav

[www.cs.technion.ac.il/~yahave/tocs2011/compilers-lec10.pptx](http://www.cs.technion.ac.il/~yahave/tocs2011/compilers-lec10.pptx)

Reference: Dragon 7.1,7.2.MCD 6.3,6.4.2

1

You are here



2

## Supporting Procedures

- new computing environment
  - at least temporary memory for local variables
- passing information into the new environment
  - parameters
- transfer of control to/from procedure
- handling return values

3

## Design Decisions

- scoping rules
  - static scoping vs. dynamic scoping
- caller/callee conventions
  - parameters
  - who saves register values?
- allocating space for local variables

4

## Static (lexical) Scoping

```
main ()
{
  int a = 0;
  int b = 0;
  {
    int b = 1;
    {
      B2 int a = 2;
      printf ("%d %d\n", a, b)
    }
    B1 {
      int b = 3;
      B3 {
        printf ("%d %d\n", a, b);
      }
      printf ("%d %d\n", a, b);
    }
    printf ("%d %d\n", a, b);
  }
}
```

a name refers to its (closest) enclosing scope  
known at compiletime

Declaration	Scopes
a=0	B0,B1,B3
b=0	B0
b=1	B1,B2
a=2	B2
b=3	B3

5

## Dynamic Scoping

- each identifier is associated with a global stack of bindings
- when entering scope where identifier is declared
  - push declaration on identifier stack
- when exiting scope where identifier is declared
  - pop identifier stack
- evaluating the identifier in any context binds to the current top of stack
- determined at runtime

6

## Example

```
int x = 42;
int f() { return x; }
int g() { int x = 1; return f(); }
int main() { return g(); }
```

- what value is returned from main?
- static scoping?
- dynamic scoping?

7

## Why do we care?

- we need to generate code to access variables
- static scoping
  - identifier binding is known at compile time
  - address of the variable is known at compile time
  - assigning addresses to variables is part of code generation
  - no runtime errors of "access to undefined variable"
  - can check types of variables

8

### Variable addresses for static scoping: first attempt

```
int x = 42;

int f() { return x; }
int g() { int x = 1; return f(); }
int main() { return g(); }
```

identifier	address
x (global)	0x42
x (inside g)	0x73

9

### Variable addresses for static scoping: first attempt

```
int a [11];

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

  main() {
    ...
    quicksort (1, 9);
  }
}
```

what is the address of the variable "i" in the procedure quicksort?

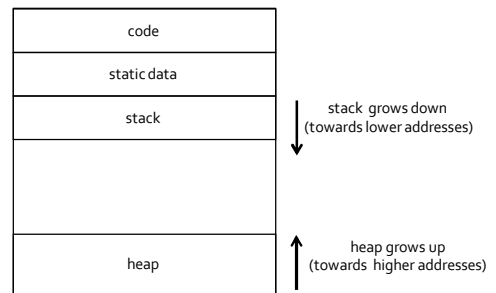
10

### Activation Record (frame)

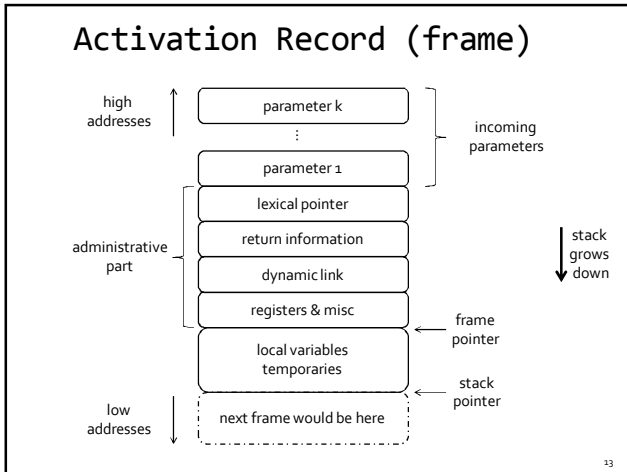
- separate space for each procedure invocation
- managed at runtime
  - code for managing it generated by the compiler
- desired properties
  - efficient allocation and deallocation
    - procedures are called frequently
  - variable size
    - different procedures may require different memory sizes

11

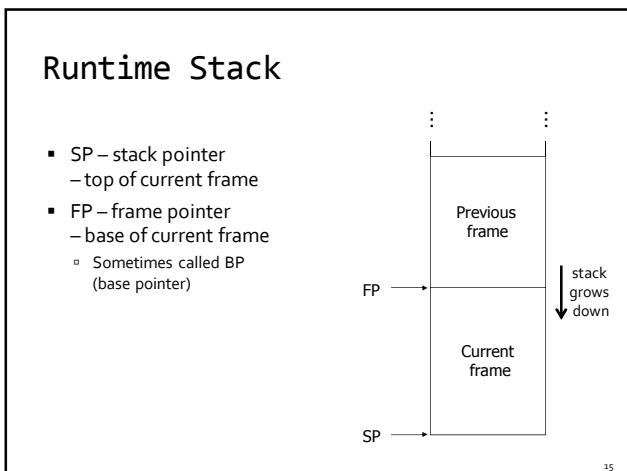
### Memory Layout



12



- ### Runtime Stack
- Stack of activation records
  - Call = push new activation record
  - Return = pop activation record
  - Only one “active” activation record – top of stack
  - How do we handle recursion?
- 14



### Pentium Runtime Stack

Register	Usage
ESP	Stack pointer
EBP	Base pointer

Pentium stack registers

Instruction	Usage
push, pusha, ...	push on runtime stack
pop, popa, ...	Base pointer
call	transfer control to called routine
return	transfer control back to caller

Pentium stack and call/ret instructions

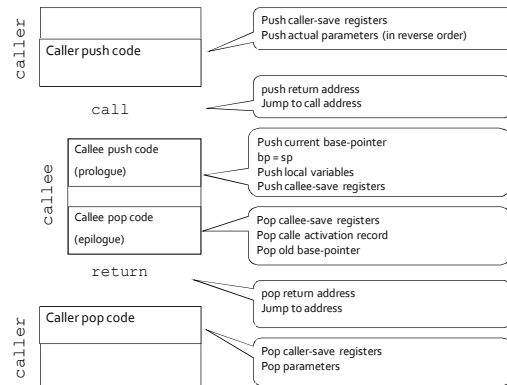
16

## Call Sequences

- The processor does not save the content of registers on procedure calls
- So who will?
  - Caller saves and restores registers
  - Callee saves and restores registers
  - But can also have both save/restore some registers

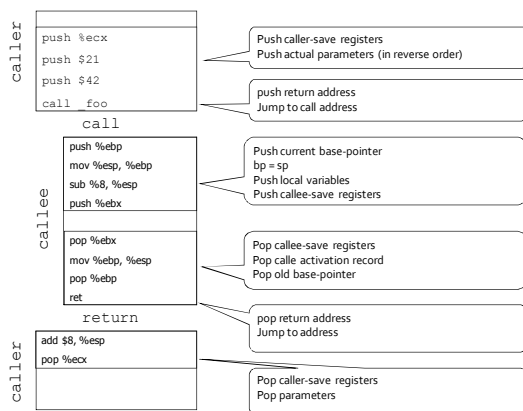
17

## Call Sequences



18

## Call Sequences - Foo(42,21)



19

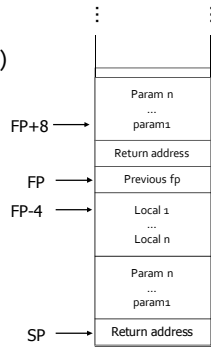
## “To Callee-save or to Caller-save?”

- That is indeed The question
- Callee-saved registers need only be saved when callee modifies their value
- some heuristics and conventions are followed

20

## Accessing Stack Variables

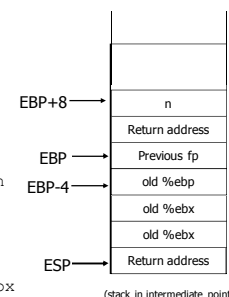
- Use offset from FP (%ebp)
- Remember – stack grows downwards
- Above FP = parameters
- Below FP = locals
- Examples
  - %ebp + 4 = return address
  - %ebp + 8 = first parameter
  - %ebp - 4 = first local



21

## Factorial - fact(int n)

```
fact:
pushl %ebp           # save ebp
movl %esp,%ebp      # ebp=esp
pushl %ebx          # save ebx
movl 8(%ebp),%ebx   # ebx = n
cmpl $1,%ebx        # n = 1 ?
jle .lresult        # then done
leal -1(%ebx),%eax  # eax = n-1
pushl %eax          #
call fact           # fact(n-1)
imull %ebx,%eax     # eax=retv*n
jmp .lreturn        #
.lresult:
movl $1,%eax        # retv
.lreturn:
movl -4(%ebp),%ebx  # restore ebx
movl %ebp,%esp     # restore esp
popl %ebp           # restore ebp
```



(disclaimer: real compiler can do better than that)

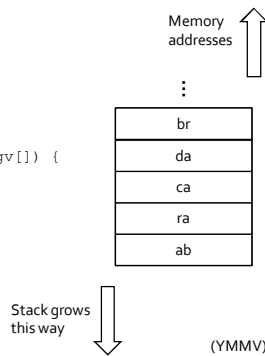
22

## Windows Exploit(s) Buffer Overflow

```
void foo (char *x) {
    char buf[2];
    strcpy(buf, x);
}

int main (int argc, char *argv[]) {
    foo(argv[1]);
}

./a.out abracadabra
Segmentation fault
```



23

## Buffer overflow

```
int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[16];

    strcpy(password_buffer, password);
    if(strcmp(password_buffer, "brillig") == 0)
        auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(argc < 2) {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }
    if(check_authentication(argv[1])) {
        printf("\n===== \n");
        printf(" Access Granted.\n");
        printf("===== \n");
    }
    else {
        printf("\nAccess Denied.\n");
    }
}
```

(source: "hacking - the art of exploitation, 2<sup>nd</sup> Ed")

24

## Buffer overflow

```
int check_authentication(char *password){
char password_buffer[16];
int auth_flag = 0;

strcpy(password_buffer, password);
if(strcmp(password_buffer, "brillig") == 0)
    auth_flag = 1;
if(strcmp(password_buffer, "outgrabe") == 0)
    auth_flag = 1;
return auth_flag;
}

int main(int argc, char *argv[]) { if(argc < 2) {
printf("Usage: %s <password>\n", argv[0]); exit(0); }
if(check_authentication(argv[1])) {
printf("\n===== \n");
printf(" Access Granted.\n");
printf("===== \n"); }
else {
printf("\nAccess Denied.\n");
}
}
```

(source: "hacking – the art of exploitation, 2<sup>nd</sup> Ed")

25

## Buffer overflow

```
0x08048529 <+69>: movl $0x8048647, (%esp)
0x08048530 <+76>: call 0x8048394 <puts@plt>
0x08048535 <+81>: movl $0x8048664, (%esp)
0x0804853c <+88>: call 0x8048394 <puts@plt>
0x08048541 <+93>: movl $0x804867a, (%esp)
0x08048548 <+100>: call 0x8048394 <puts@plt>
0x0804854d <+105>: jmp 0x804855b <main+119>
0x0804854f <+107>: movl $0x8048696, (%esp)
0x08048556 <+114>: call 0x8048394 <puts@plt>
```

26

## Nested Procedures

- For example – Pascal
- any routine can have sub-routines
- any sub-routine can access anything that is defined in its containing scope or inside the sub-routine itself

27

## Example: Nested Procedures

```
program p;
var x: Integer;
procedure a
    var y: Integer;
    procedure b begin...b... end;
    function c
        var z: Integer;
        procedure d begin...d... end;
        begin...C...end;
    begin...a... end;
begin...p... end.
```

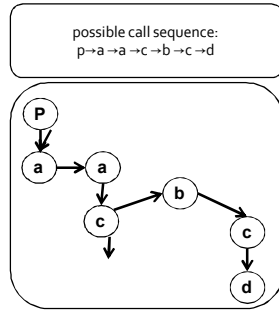
possible call sequence:  
p → a → a → c → b → c → d

what is the address of  
variable "y" in procedure d?

28

### nested procedures

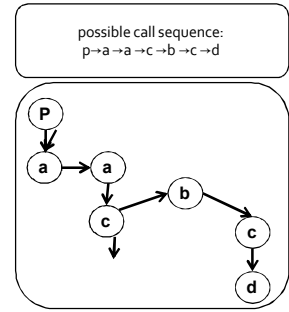
- can call a sibling, ancestor
- when "c" uses variables from "a", which "a" is it?
- how do you find the right activation record at runtime?



29

### nested procedures

- goal: find the closest routine in the stack from a given nesting level
- if we reached the same routine in a sequence of calls
  - routine of level k uses variables of the same level, it uses its own variables
  - if it uses variables of level j < k then it must be the last routine called at level j
- If a procedure is last at level j on the stack, then it must be ancestor of the current routine



30

### nested procedures

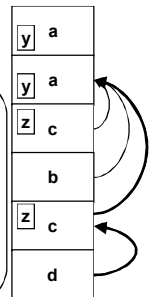
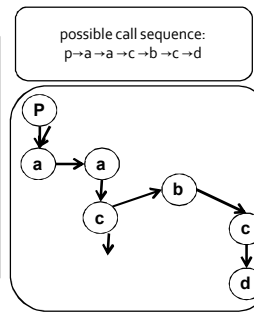
- problem: a routine may need to access variables of another routine that contains it statically
- solution: lexical pointer (a.k.a. access link) in the activation record
- lexical pointer points to the last activation record of the nesting level above it
  - in our example, lexical pointer of d points to activation records of c
- lexical pointers created at runtime
- number of links to be traversed is known at compile time

31

### lexical pointers

```

program p;
var x: Integer;
procedure a
  var y: Integer;
  procedure b begin...b... end;
  function c
    var z: Integer;
    procedure d begin...d... end;
  begin...c...end;
begin...a...end;
begin...p...end.
    
```



32



## Activation Records: Summary

- compile time memory management for procedure data
- works well for data with well-scoped lifetime
  - deallocation when procedure returns

33

The End

34