Lecture 10 – Activation Records

Eran Yahav

www.cs.technion.ac.il/~yahave/tocs2011/compilers-lec10.pptx

Reference: Dragon 7.1,7.2. MCD 6.3,6.4.2
You are here
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
Design Decisions

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

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

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Scopes</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=0</td>
<td>B0,B1,B3</td>
</tr>
<tr>
<td>b=0</td>
<td>B0</td>
</tr>
<tr>
<td>b=1</td>
<td>B1,B2</td>
</tr>
<tr>
<td>a=2</td>
<td>B2</td>
</tr>
<tr>
<td>b=3</td>
<td>B3</td>
</tr>
</tbody>
</table>
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 existing scope where identifier is declared
  - pop identifier stack
- evaluating the identifier in any context binds to the current top of stack
- determined at runtime
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?
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
Variable addresses for static scoping: first attempt

<table>
<thead>
<tr>
<th>identifier</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>x (global)</td>
<td>0x42</td>
</tr>
<tr>
<td>x (inside g)</td>
<td>0x73</td>
</tr>
</tbody>
</table>

```c
int x = 42;

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

- code
- static data
- stack
- heap

Stack grows down (towards lower addresses)
Heap grows up (towards higher addresses)
Activation Record (frame)

- parameter k
- parameter 1
- lexical pointer
- return information
- dynamic link
- registers & misc
- local variables
- temporaries
- next frame would be here

High addresses

Incoming parameters

Administrative part

Stack grows down

Low addresses

Frame pointer

Stack pointer
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?
Runtime Stack

- SP – stack pointer
  - top of current frame
- FP – frame pointer
  - base of current frame
  - Sometimes called BP (base pointer)

(stack grows down)
Pentium Runtime Stack

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>EBP</td>
<td>Base pointer</td>
</tr>
</tbody>
</table>

Pentium stack registers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>push, pusha, ...</td>
<td>push on runtime stack</td>
</tr>
<tr>
<td>pop, popa, ...</td>
<td>Base pointer</td>
</tr>
<tr>
<td>call</td>
<td>transfer control to called routine</td>
</tr>
<tr>
<td>return</td>
<td>transfer control back to caller</td>
</tr>
</tbody>
</table>

Pentium stack and call/ret instructions
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
Call Sequences

**Caller push code**
- Push caller-save registers
- Push actual parameters (in reverse order)
- Push return address
- Jump to call address

**Callee push code**
- (prologue)
  - Push current base-pointer
  - bp = sp
  - Push local variables
  - Push callee-save registers
- (epilogue)
  - Pop callee-save registers
  - Pop callee activation record
  - Pop old base-pointer

**return**
- Pop return address
- Jump to address

**Caller pop code**
- Pop caller-save registers
- Pop parameters
Call Sequences – Foo(42,21)

**Caller**
- push %ecx
- push $21
- push $42
- call _foo

**Call**
- push %ebp
- mov %esp, %ebp
- sub %8, %esp
- push %ebx

**Callee**
- pop %ebx
- mov %ebp, %esp
- pop %ebp
- pop %ecx
- ret

**Return**
- add $8, %esp
- pop %ecx

**Push caller-save registers**
Push actual parameters (in reverse order)

**Push return address**
Jump to call address

**Push current base-pointer**
bp = sp
Push local variables
Push callee-save registers

**Pop callee-save registers**
Pop callee activation record
Pop old base-pointer

**Pop return address**
Jump to address

**Pop caller-save registers**
Pop parameters
“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
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
Factorial - \texttt{fact}(\texttt{int } n)\\

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

\text{(disclaimer: real compiler can do better than that)}
Windows Exploit(s)  
Buffer Overflow

```c
void foo (char *x) {
    char buf[2];
    strcpy(buf, x);
}
int main (int argc, char *argv[]) {
    foo(argv[1]);
}
./a.out abracadabra
Segmentation fault
```

Memory addresses

```
br
da
dc
ra
ab
```

Stack grows this way
Buffer overflow

```c
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\textsuperscript{nd} Ed”)

Buffer overflow

```c
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, 2nd Ed”)

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>
0x08048554 <+107>: movl   $0x8048696,(%esp)
0x08048556 <+114>: call   0x8048394 <puts@plt>
```
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
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.
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?

possible call sequence: 
P → a → a → c → b → c → d
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
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
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.

possible call sequence:
p→a→a→c→b→c→d
Activation Records: Summary

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