THEORY OF COMPILATION

LECTURE 10

ACTIVATION RECORDS
You are here

Source text → Lexical Analysis → Syntax Analysis → Semantic Analysis → IR Optimization → Code Generation → Executable code

+ Runtime
Supporting Procedures

- How is that done?
- What do we need from the compiler?

\[
n = f(a[i]);
\]

\[
t1 = i * 4
\]
\[
t2 = a[t1] \ // \text{could have expanded this as well}
\]
\[
\text{param} \ t2
\]
\[
t3 = \text{call} \ f, 1
\]
\[
n = t3
\]
Supporting Procedures

• Extending our computing environment
  ‣ (at least) enough 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

• Memory layout
  ‣ Allocating space for local variables

• Caller/callee conventions
  ‣ who saves and restores register values?
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);
}
```

a name refers to its (closest) enclosing scope

known at compile time

<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 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.
Example

```cpp
int x = 42;

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

- What values are output by `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 Addressing for Static Scoping (first attempt)

```c
int x = 42;

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

<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>
Variable Addressing for Static Scoping (first attempt)

```c

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) ;
}
```

<table>
<thead>
<tr>
<th>identifier</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a (global)</td>
<td>0x42</td>
</tr>
<tr>
<td>i</td>
<td>...</td>
</tr>
</tbody>
</table>

(inside quicksort)

**what is the address of the variable “i” in the procedure quicksort?**
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
    - procedure calls are frequent
  - variable size
    - different procedures may require different memory sizes
Memory Layout

- Stack grows down (towards lower addresses)
- Heap grows up (towards higher addresses)

- Higher addresses
- Lower addresses
- Stack
- Heap
- Static data
- Code

0x0000
Memory Layout

- Stack grows down (towards lower addresses)
- Heap grows up (towards higher addresses)

- Higher addresses
- Lower addresses
- Stack
- Heap
- Static data
- Code

Address: 0x0000
Activation Record (frame)

- parameter k
- parameter 1
- lexical pointer
- return information
- dynamic link (= prev. fp)
- registers & misc
- local variables
- temporaries
- next frame would be here

Higher addresses

Administrative part

Lower addresses

Incoming parameters

Stack grows down

Frame pointer

Stack pointer
Runtime Stack

• Stack of activation records
  ‣ Call = push new activation record
  ‣ Return = pop activation record
• Only one “active” record at a time (per thread)
  ‣ top of stack
• How do we handle recursion?
Runtime Stack

- SP – stack pointer
  - end of current frame
- FP – frame pointer
  - beginning of current frame
    - Sometimes called BP (base pointer)
x86 Runtime Stack

**Stack registers**

<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>Frame pointer</td>
</tr>
</tbody>
</table>

**Stack and subroutine instructions**

<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>pop from runtime stack</td>
</tr>
<tr>
<td>call</td>
<td>transfer control to called routine</td>
</tr>
<tr>
<td>ret</td>
<td>transfer control back to caller</td>
</tr>
</tbody>
</table>
Call Sequences

• The processor does not save the content of registers on “call” instruction

• So who will?
  ‣ Caller saves and restores registers
  ‣ Callee saves and restores registers
  ‣ But can also have both save/restore some registers
Call Sequences

FP

SP

Previous frame
Call Sequences

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

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

Caller push code
Call Sequences

Caller push code

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

FP
Previous frame
Reg 1
...
Reg n
Param n
...
Param 1

SP
Call Sequences

Caller push code

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

Push return address
Jump to call address

Param n
Param 1
Reg n
Reg 1
Previous frame

FP
SP
Call Sequences

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

Call Sequences:
Call Sequences

**Call**

- **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

- **Callee push code** *(epilogue)*

- **Return address**

- **SP**

- **FP**
  - Previous frame
  - Reg 1
  - ... Reg n
  - Param n
  - ... Param 1
  - Return address
Call Sequences

caller

Caller push code

Callee push code

(prologue)

call

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

FP

Previous frame

Reg 1

...

Reg n

Param n

...

Param 1

Return address

Previous FP

SP
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

- Return address
- Param 1
- Param n
- Reg n
- Reg 1

- Previous frame
- Previous FP

- SP
- FP
Call Sequences

Call

Caller push code

Callee push code (prologue)

Call

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

Previous frame

Reg 1
...  
Reg n

Param n
...  
Param 1

Return address

Previous FP

Local 1
Local 2
...  
...  
Local n

SP

FP
Call Sequences

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

**Call**

**Callee push code** (prologue)
- Push current base-pointer
- BP = SP
- Push local variables
- Push callee-save registers

**Callee pop code** (epilogue)
- Pop callee-save registers
- Pop callee activation record
- Pop old base-pointer

**Return address**

**Previous frame**
- Reg 1
  - ...
  - Reg n

**Previous FP**
- Local 1
  - Local 2
  - ...
  - Local n

**FP**

**SP**
Call Sequences

<table>
<thead>
<tr>
<th>caller</th>
<th>callee</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Caller push code</strong></td>
<td><strong>Callee push code (prologue)</strong></td>
</tr>
<tr>
<td><strong>call</strong></td>
<td><strong>Callee push code</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Callee pop code (epilogue)</strong></td>
</tr>
</tbody>
</table>

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

- **Callee push code (prologue)**
  - Push current base-pointer
  - BP = SP
  - Push local variables
  - Push callee-save registers

- **Callee pop code (epilogue)**
  - Pop callee-save registers
  - Pop callee activation record
  - Pop old base-pointer

- **Call Sequences**
  - Push return address
  - Jump to call address

- **Return address**
  - Previous frame
  - Reg 1
    - ... Reg n
  - Param n
    - ... Param 1
  - Previous FP
  - Local 1
    - Local 2
    - ... Local n
Call Sequences

**caller**

- **Caller push code**

**call**

- **Callee push code** (prologue)
- **Callee pop code** (epilogue)

**callee**

**return**

- **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**

---

**Frame Stack**

- **SP**
- **Previous FP**
- **Local 1**
- **Local 2**
- **...**
- **Local n**

- **FP**
- **Previous frame**
- **Reg 1**
- **...**
- **Reg n**
- **Param n**
- **...**
- **Param 1**
- **Return address**

---

**Diagram Notes**

- **Call Sequences**
- **FP**
- **SP**

---

**Additional Notes**

- **Call Sequences**
- **FP**
- **SP**
Call Sequences

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

**Call**
- **prologue**
  - Push current base-pointer
  - BP = SP
  - Push local variables
  - Push callee-save registers

**Callee**
- **push code**
  - Push callee-save registers
  - Pop callee-save registers
  - Pop callee activation record
  - Pop old base-pointer

**return**
- **pop code**
  - Pop return address
  - Jump to address
Call Sequences

**caller**
- Caller push code

**call**
- Call
  - Callee push code (prologue)
  - Callee pop code (epilogue)

**callee**
- 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

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

**caller**
- Caller pop code
  - Pop parameters
  - Pop caller-save registers
Call Sequences

**caller**
- **Caller push code**

**call**
- **Callee push code**
  - (prologue)
  - **Callee pop code**
    - (epilogue)

**callee**
- 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

**return**
- Pop return address
- Jump to address
- Pop parameters
- Pop caller-save registers

**caller**
- **Caller pop code**

**variable areas**
- Previous frame
  - Previous FP
  - Reg 1
    - ... Reg n
  - Param n
    - ... Param 1
  - Return address
  - Previous FP
  - Local 1
    - ... Local n
### Call Sequences

#### Call
- `push %ecx`
- `push $21`
- `push $42`
- `call _foo`

#### Callee
- `push %ebp`
- `mov %esp, %ebp`
- `sub $8, %esp`
- `push %ebx`
- `pop %ebx`
- `mov %ebp, %esp`
- `pop %ebp`
- `ret`

#### Caller
- `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 parameters**
- Pop caller-save registers

---

**Foo(42, 21)**
“To Callee-save or to Caller-save?”

• That is indeed the question
  ‣ Callee-saved registers need only be saved when callee modifies their value
  ‣ Caller-saved registers need only be saved if the caller needs their value after the call returns

• Some conventions and heuristics 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(int n)}

\begin{verbatim}
.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      # return 1
    .lreturn:
    movl -4(%ebp),%ebx # restore ebx
    movl %ebp,%esp    # restore esp
    popl %ebp         # restore ebp
\end{verbatim}
Nested Procedures

• For example – in Pascal, JavaScript

• 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
  – “non-local” variables
Example: Nested Procedures

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

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

what is the address of variable “y” in procedure d?
Nested Procedures

- A routine can call a sibling or an ancestor
- When "c" uses (non-local) variables from "a", which instance of "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 hit the same routine more than once in a sequence of calls
  - routine of level $k$ uses variables of the same nesting level $\Rightarrow$ it uses its own variables
  - if it uses variables of nesting level $j < k$ $\Rightarrow$ 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

Possible call sequence: $p \rightarrow a \rightarrow a \rightarrow c \rightarrow b \rightarrow c \rightarrow d$
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.
    - According to what we just said, this is always enough.
- Lexical pointers are created at runtime.
- Number of links to be traversed is always known at compile time.
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.
begin...
  c...
  b...
  a...
... end;
possible call sequence:
p→a→a→c→b→c→d
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.
begin...
c...end;
begin...
b... end;
begin...
d... end;
possible call sequence:
p~a~a~c~b~c~d
void foo (char *x) {
    char buf[2];
    strcpy(buf, x);
}

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

% ./a.out abracadabra
Segmentation fault

(YMMV)
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

(YMMV)
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");
        printf("      Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
    } else {
        printf("\nAccess Denied.\n");
    }
}

(source: "hacking – the art of exploitation, 2nd Ed")
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");
    }
}
Activation Records: Summary

✓ Compile time memory management for procedure data
  – Used to pass parameters, store local variables, and restore registers
✓ Works well for data with well-scoped lifetime
  – deallocation when procedure returns
  – no need to call free() for stack data
Coming Up

MEMORY and ERROR Mgmt