Machine-Level Programming III: Procedures

CS351: Systems Programming
Day 8: Sep. 15, 2022

Instructor:
Nik Sultana

Slides adapted from Bryant and O’Hallaron
Next time: recorded lecture

- **LEC 9** and **LEC 10** will be pre-recorded and circulated on Blackboard.
  - Do not come to SB104 those days – there will not be an in-person lecture.
  - My away-at-a-conference days are marked on the course calendar.
Textbook survey

- On the use and access to textbooks.
- The form is anonymous. (It checks that you’re at Illinois Tech to filter abuse, but I don’t see who submitted any of the forms.)
- [https://forms.gle/tv1A36JdjCBPkcdn9](https://forms.gle/tv1A36JdjCBPkcdn9)
Mechanisms in Procedures

- **Passing control**
  - To beginning of procedure code
  - Back to return point

- **Passing data**
  - Procedure arguments
  - Return value

- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return

- **Mechanisms all implemented with machine instructions**

- **x86-64 implementation of a procedure uses only those mechanisms required**

```c
int Q(int i) {
    int t = 3*i;
    int v[10];
    •
    •
    return v[t];
}
```

```c
P(...) {
    •
    •
    y = Q(x);
    print(y)
    •
}
```
Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
**x86-64 Stack**

- Region of memory managed with stack discipline
- Grows toward lower addresses

- Register `%rsp` contains lowest stack address
  - address of “top” element

---

Stack Pointer: `%rsp`

Stack “Top”

Stack “Bottom”

Increasing Addresses

Stack Grows Down
x86-64 Stack: Push

- `pushq Src`
  - Fetch operand at `Src`
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`

Stack Pointer: `%rsp`

Stack “Bottom”

Increasing Addresses

Stack Grows Down

Stack “Top”
x86-64 Stack: Pop

- `popq Dest`
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at `Dest` (must be register)
Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
Code Examples

```c
void multstore
  (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```assembly
0000000000400540 <multstore>:
  400540: push %rbx        # Save %rbx
  400541: mov %rdx,%rbx    # Save dest
  400544: callq 400550 <mult2> # mult2(x,y)
  400549: mov %rax,(%rbx)  # Save at dest
  40054c: pop %rbx         # Restore %rbx
  40054d: retq            # Return

long mult2
  (long a, long b)
{
    long s = a * b;
    return s;
}
```

```assembly
0000000000400550 <mult2>:
  400550: mov %rdi,%rax   # a
  400553: imul %rsi,%rax  # a * b
  400557: retq           # Return
```
Procedure Control Flow

- Use stack to support procedure call and return

- **Procedure call:** `call label`
  - Push return address on stack
  - Jump to label

- **Return address:**
  - Address of the next instruction right after call
  - Example from disassembly

- **Procedure return:** `ret`
  - Pop address from stack
  - Jump to address
Control Flow Example #1

00000000000400540 <multstore>:
•
•
400544: callq 400550 <mult2>
400549: mov %rax, (%rbx)
•
•

00000000000400550 <mult2>:
400550: mov %rdi, %rax
•
•
400557: retq
Control Flow Example #2

**multstore**:  
400544: callq 400550 <mult2>  
400549: mov %rax, (%rbx)  

**mult2**:  
400550: mov %rdi, %rax  
400557: retq
Control Flow Example #3

0000000000400540 <multstore>:
- 
- 400544: callq 400550 <mult2>
400549: mov %rax,(%rbx)
- 
- 

0000000000400550 <mult2>:
  400550: mov %rdi,%rax
  - 
  - 
  400557: retq
Control Flow Example #4

00000000000400540 <multstore>:

400544: callq 400550 <mult2>
400549: mov %rax,(%rbx)

00000000000400550 <mult2>:

400550: mov %rdi,%rax
400557: retq
Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustrations of Recursion & Pointers
Procedure Data Flow

Registers

- First 6 arguments
  - %rdi
  - %rsi
  - %rdx
  - %rcx
  - %r8
  - %r9

- Return value
  - %rax

Stack

- Only allocate stack space when needed
  - ... 
  - Arg n
  - ... 
  - Arg 8
  - Arg 7
void multstore
  (long x, long y, long *dest)
{
  long t = mult2(x, y);
  *dest = t;
}

long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
Stack-Based Languages

- **Languages that support recursion**
  - e.g., C, Pascal, Java
  - Code must be “Reentrant”
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer

- **Stack discipline**
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does

- **Stack allocated in Frames**
  - state for single procedure instantiation
Call Chain Example

```java
yoo(...) {
    ...
    who();
    ...
}
```

```java
who(...) {
    ...
    amI();
    ...
}
```

```java
amI(...) {
    ...
    amI();
    ...
}
```

Procedure `amI()` is recursive
Stack Frames

Contents
- Return information
- Local storage (if needed)
- Temporary space (if needed)

Management
- Space allocated when enter procedure
  - “Set-up” code
  - Includes push by call instruction
- Deallocated when return
  - “Finish” code
  - Includes pop by ret instruction
Example

```c
yoo (...) {
    •
    •
    who () ;
    •
    •
}
```

Stack

```
%rbp
%rsp
```

```
yoo
who
aml
aml
aml
```

Illinois Tech CS351 Fall 2022
Example

```c
yoo() {
    who(...) {
        • • •
        amI();
        • • •
        amI();
        • • •
    }
}
```

Stack

```
%rbp

%rsp
```

Illustration of function call stack and memory allocation.
Example

```plaintext
yoo() {
  who(...) {
    amI(...) {
      •
      •
      amI();
      •
    }
  }
}
```

Stack

- `yoo`
- `who`
- `amI`
- `%rbp`
- `%rsp`

Illinois Tech CS351 Fall 2022
Example

```c
yoo() {
  who(...) {
    amI(...) {
      amI(...) {
        amI();
        ...
      }
    }
  }
}
```

Stack

```
yoo
who
amI
amI
%rbp
%rsp
```
Example

```c
yoo(...) {
  who(...) {
    amI(...) {
      amI(...) {
        amI();
        ...
      }
    }
  }
}
```

Stack

```
%rbp

%rsp
```

```
yoo
who
amI
amI
```
Example

```c
yoo() {
  who(...) {
    amI(...) {
      •
      •
      amI();
      •
      •
    }
  }
}
```

Stack

```
Stack

yoo
who
amI
%rbp %rsp
```
Example

```c
void yoo() {
    who(...) {
        • • •
        amI();
        • • •
        amI();
        • • •
    }
}
```

Stack

```c
who
%rbp
%rsp
yoo
who
```
Example

```c
yoo() {
  who(...) {
    amI(...) {
      • • •
      amI();
      • • •
    }
    • • •
  }
  who();
  • • •
}
```
Example

```c
yoo() {
  who(...) {
    • • •
    amI();
    • • •
    amI();
  }
}
```

Stack

```
%rbp
```

```
%rsp
```

```
yoo
```
Example

```c
yoo (...) {
    •
    •
    •
    who ();
    •
    •
}
```

Stack

```
%rbp

%rsp
```

```c
yoo

who

aml

aml

aml
```
x86-64/Linux Stack Frame

**Current Stack Frame** ("Top" to Bottom)
- "Argument build:"
  Parameters for function about to call
- Local variables
  If can’t keep in registers
- Saved register context
- Old frame pointer (optional)

**Caller Stack Frame**
- Return address
  - Pushed by *call* instruction
- Arguments for this call
Example: incr

```c
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

incr:
```
movq (%rdi), %rax
addq %rax, %rsi
movq %rsi, (%rdi)
ret
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument p</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument val, y</td>
</tr>
<tr>
<td>%rax</td>
<td>x, Return value</td>
</tr>
</tbody>
</table>
Example: Calling incr #1

```c
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

call_incr:
```assembly
subq  $16, %rsp
movq  $15213, 8(%rsp)
movl  $3000, %esi
leaq  8(%rsp), %rdi
call  incr
addq  8(%rsp), %rax
addq  $16, %rsp
ret
```
Example: Calling incr #2

```c
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

call_incr:
- subq $16, %rsp
- movq $15213, 8(%rsp)
- movl $3000, %esi
- leaq 8(%rsp), %rdi
- call incr
- addq 8(%rsp), %rax
- addq $16, %rsp
- ret

Stack Structure:
- Rtn address
- 15213
- Unused
- %rsp+8
- %rsp

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>3000</td>
</tr>
</tbody>
</table>
Example: Calling `incr` #3

```c
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

call_incr:

```
subq $16, %rsp
movq $15213, 8(%rsp)
movl $3000, %esi
leaq 8(%rsp), %rdi
call incr
addq 8(%rsp), %rax
addq $16, %rsp
ret
```

Stack Structure:

<table>
<thead>
<tr>
<th>Rtn address</th>
<th>%rsp+8</th>
</tr>
</thead>
<tbody>
<tr>
<td>18213</td>
<td></td>
</tr>
<tr>
<td>Unused</td>
<td>%rsp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>3000</td>
</tr>
</tbody>
</table>
Example: Calling \texttt{incr} #4

```c
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

**Stack Structure**

<table>
<thead>
<tr>
<th>Address</th>
<th>Register/Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp+8</td>
<td>18213</td>
</tr>
<tr>
<td>%rsp</td>
<td>Unused</td>
</tr>
</tbody>
</table>

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

**Updated Stack Structure**
Example: Calling `incr` #5

```c
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

**Updated Stack Structure**

**Final Stack Structure**
Register Saving Conventions

- **When procedure yoo calls who:**
  - `yoo` is the **caller**
  - `who` is the **callee**

- **Can register be used for temporary storage?**

```
```
yoo:
  movq $15213, %rdx
  call who
  addq %rdx, %rax
  ret
``` who:
  subq $18213, %rdx
  ret
```

- Contents of register `%rdx` overwritten by `who`
- This could be trouble → something should be done!
  - Need some coordination
Register Saving Conventions

- **When procedure** `yoo` **calls** `who`:
  - `yoo` **is the** caller
  - `who` **is the** callee

- **Can register be used for temporary storage?**

- **Conventions**
  - “**Caller Saved”**
    - Caller saves temporary values in its frame before the call
  - “**Callee Saved”**
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller
**x86-64 Linux Register Usage #1**

- **%rax**
  - Return value
  - Also caller-saved
  - Can be modified by procedure

- **%rdi, ..., %r9**
  - Arguments
  - Also caller-saved
  - Can be modified by procedure

- **%r10, %r11**
  - Caller-saved
  - Can be modified by procedure

---

**Return value**
- %rax
- %rdi
- %rsi
- %rdx
- %rcx
- %r8
- %r9

**Arguments**
- %r10
- %r11
x86-64 Linux Register Usage #2

- `%rbx`, `%r12`, `%r13`, `%r14
  - Callee-saved
  - Callee must save & restore

- `%rbp`
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match

- `%rsp`
  - Special form of callee save
  - Restored to original value upon exit from procedure
Callee-Saved Example #1

```c
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

call_incr2:

```
pushq  %rbx
subq  $16, %rsp
movq  %rdi, %rbx
movq  $15213, 8(%rsp)
movl  $3000, %esi
leaq  8(%rsp), %rdi
call  incr
addq  %rbx, %rax
addq  $16, %rsp
popq  %rbx
ret
```
Callee-Saved Example #2

```c
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

**call_incr2:**
- pushq %rbx
- subq $16, %rsp
- movq %rdi, %rbx
- movq $15213, 8(%rsp)
- movl $3000, %esi
- leaq 8(%rsp), %rdi
- call incr
- addq %rbx, %rax
- addq $16, %rsp
- popq %rbx
- ret

**Pre-return Stack Structure**
- ... Rtn address
- Saved %rbx
- 15213 %rsp+8
- Unused %rsp

**Resulting Stack Structure**
- ... Rtn address
- %rsp
Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
Recursive Function

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    jeq     .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}

pcount_r:
    movl  $0, %eax
    testq %rdi, %rdi
    je      .L6
    pushq %rbx
    movq  %rdi, %rbx
    andl  $1, %ebx
    shrq  %rdi
    call  pcount_r
    addq  %rbx, %rax
    popq  %rbx
.L6:
    rep; ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
<td>Argument</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
</tbody>
</table>
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}

Register Use(s) Type
%rdi x Argument

pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
    call pcount_r
    addq %rbx, %rax
    popq %rbx
.L6:
    rep; ret

...
Recursive Function Call Setup

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

Register Use(s)       Type
---                  ----
%rdi    x >> 1     Rec. argument
%rbx    x & 1      Callee-saved

pcount_r:
```assembly
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
    .L6:
    rep; ret
```
Recursive Function Call

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>x &amp; 1</td>
<td>Callee-saved</td>
</tr>
<tr>
<td>%rax</td>
<td>Recursive call return value</td>
<td></td>
</tr>
</tbody>
</table>
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}

Recursive Function Result

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
    call pcount_r
    addq %rbx, %rax
    popq %rbx
.L6:
    rep; ret
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>x &amp; 1</td>
<td>Callee-saved</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
<td></td>
</tr>
</tbody>
</table>
Recursive Function Completion

/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}

pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
    call pcount_r
    addq %rbx, %rax
    popq %rbx
    .L6:
    rep; ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Observations About Recursion

- **Handled Without Special Consideration**
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another’s data
    - Unless the C code explicitly does so (e.g., buffer overflow)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out

- **Also works for mutual recursion**
  - P calls Q; Q calls P
x86-64 Procedure Summary

- **Important Points**
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
  - Recursion (& mutual recursion) handled by normal calling conventions
    - Can safely store values in local stack frame and in callee-saved registers
    - Put function arguments at top of stack
    - Result return in $rax
  - Pointers are addresses of values
    - On stack or global

- **Callers**
  - Arguments
  - Return Addr
  - Old %rbp
  - Saved
    - Registers
    - Local
    - Variables
  - Argument
    - Build

Illinois Tech CS351 Fall 2022
Per-lecture feedback

- Better sooner rather than later!
- I can help with issues sooner.
- There is a per-lecture feedback form.
- **The form is anonymous.**
  (It checks that you’re at Illinois Tech to filter abuse, but I don’t see who submitted any of the forms.)
- https://forms.gle/qoeEbBuTYXo5FiU1A
- I’ll remind about this at each lecture.
Next time: recorded lecture

- **LEC 9 and LEC 10** will be pre-recorded and circulated on Blackboard.
  - **Do not come to SB104 those days** – there will not be an in-person lecture.
  - **My away-at-a-conference days** are marked on the course calendar.