CS 33

Machine Programming (3)
IA32 Stack

- Region of memory managed “last-in, first-out”
- Grows toward lower addresses
- Register `%esp` contains lowest stack address – address of “top” element

Stack pointer: `%esp`
IA32 Stack: Push

- **pushl src**
  - fetch operand at src
    - immediate, register, or memory location
  - decrement %esp by 4
  - store operand at address given by %esp
IA32 Stack: Pop

- `popl dest`
  - fetch operand from address given by `%esp`
  - put operand in `dest`
  » register or memory location
  - increment `%esp` by 4

Stack pointer: `%esp`
Procedure Control Flow

- Use stack to support procedure call and return

  **Procedure call:** `call sub`
  - push return address on stack
  - jump to `sub`

  **Return address:**
  - address of the next instruction after call
  - example from disassembly

    | Address | Instruction     |
    |---------|----------------|
    | 804854e | call 8048b90 <sub> |
    | 8048553 | pushl %eax      |

  - return address = 0x8048553

- **Procedure return:** `ret`
  - pop address from stack
  - jump to address
Procedure Call

804854e: e8 3d 06 00 00 call 8048b90 <sub>
8048553: 50 pushl %eax

804854e:
  e8 3d 06 00 00 call 8048b90 <sub>
8048553:
  50 pushl %eax

%esp 0x108
%esp 0x104
%eip 0x804854e
%eip 0x8048b90

%eip: program counter
Procedure Return

8048591:  c3  ret

%esp  0x104
%eip  0x8048591

ret

%esp  0x104
%eip  0x8048591

%esp  0x108
%eip  0x8048553

0x110
0x10c
0x108
0x104

0x110
0x10c
0x108
0x104

123

0x8048553

123

0x8048553

%eip: program counter
The IA32 Stack Frame

- arg n
- arg 1
- return address
- saved frame pointer: ebp
- saved registers
- local variables: esp
Passing Arguments

```c
int x;
int res;
int main() {
    ...
    res = subr(3, x);
    ...
}
```

```
main:
    ...
    pushl x
    pushl $3
    call subr
    movl %eax, res
    ...
```
Retrieving Arguments

```c
int subr(int a, int b) {
    return a + b;
}
```

subr:
- `pushl %ebp`
- `movl %esp, %ebp`
- `movl 12(%ebp), %eax`
- `addl 8(%ebp), %eax`
- `popl %ebp`
- `ret`
Space for Local Variables

```c
int subr(int a, int b) {
    int array[20];
    ...
}
```

```
subr:
    pushl %ebp
    movl %esp, %ebp
    subl $80, %esp
    ...
    addl $80, %esp
    popl %ebp
    ret
```
Quick Exit ...

```c
int subr(int a, int b) {
    int array[20];
    ...
}
```

```
subr:
    pushl %ebp
    movl %esp, %ebp
    subl $80, %esp
    ...
    leave
    ret
```
Register-Saving Conventions

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

- Can registers be used for temporary storage?

  - contents of register `%edx` overwritten by `who`
  - this could be trouble: something should be done!

    » need some coordination

`yoo`:
```
  . . .
  movl $33, %edx
  call who
  addl %edx, %eax
  . . .
  ret
```

`who`:
```
  . . .
  movl 8(%ebp), %edx
  addl $32, %edx
  . . .
  ret
```
Register-Saving Conventions

- When procedure \texttt{yoo} calls \texttt{who}:
  - \texttt{yoo} is the \texttt{caller}
  - \texttt{who} is the \texttt{callee}

- Can registers be used for temporary storage?

- Conventions
  - "\texttt{caller save}"
    » caller saves registers containing temporary values on stack before the call
    » restores them after call
  - "\texttt{callee save}"
    » callee saves registers on stack before using
    » restores them before returning
IA32/Linux+Windows Register Usage

- `%eax`, `%edx`, `%ecx`  
  - caller saves prior to call if values are used later

- `%eax`  
  - also used to return integer value

- `%ebx`, `%esi`, `%edi`  
  - callee saves if wants to use them

- `%esp`, `%ebp`  
  - special form of callee-save  
  - restored to original values upon exit from procedure
Register-Saving Example

yoo:

... 
movl $33, %edx
pushl %edx
call who
popl %edx
addl %edx, %eax
...
ret

who:

... 
pushl %ebx
...
movl 4(%ebp), %ebx
addl %53, %ebx
movl 8(%ebp), %edx
addl $32, %edx
...
popl %ebx
...
ret
Quiz 1

- The leave instruction copies the current value of %ebp into %esp. It’s followed by a ret instruction. Does this approach for returning from a procedure work if there are saved registers in the stack frame?
  
a) always  
b) usually  
c) never
Recursive Function

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

• Registers
  – %eax, %edx used without first saving
  – %ebx used, but saved at beginning & restored at end

pcount_r:
pushl %ebp
movl %esp, %ebp
pushl %ebx
subl $4, %esp
movl 8(%ebp), %ebx
movl $0, %eax
testl %ebx, %ebx
je .L3
movl %ebx, %eax
shrl $1, %eax
movl %eax, (%esp)
call pcount_r
movl %ebx, %edx
andl $1, %edx
leal (%edx,%eax), %eax
.L3:
addl $4, %esp
popl %ebx
popl %ebp
ret
Recursive Call #1

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

• Actions
  – save old value of %ebx on stack
  – allocate space for argument to recursive call
  – store x in %ebx

pcount_r:
pushl %ebp
movl %esp, %ebp
pushl %ebx
subl $4, %esp
movl 8(%ebp), %ebx

...
Recursive Call #2

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

• Actions
  – if x == 0, return
    » with %eax set to 0

...movl $0, %eax
testl %ebx, %ebx
je .L3
... .L3:
... ret
Recursive Call #3

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

• Actions
  – store x >> 1 on stack
  – make recursive call

• Effect
  – %eax set to function result
  – %ebx still has value of x

movl %ebx, %eax
shrl $1, %eax
movl %eax, (%esp)
call pcount_r

...
Recursive Call #4

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

• Assume
  – %eax holds value from recursive call
  – %ebx holds x
• Actions
  – compute (x & 1) + computed value
• Effect
  – %eax set to function result

movl  %ebx, %edx
andl  $1, %edx
leal (%edx,%eax), %eax

CS33 Intro to Computer Systems  XII–22
Recursive Call #5

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

- **Actions**
  - restore values of `%ebx` and `%ebp`
  - restore `%esp`

```
L3:
addl $4, %esp
popl %ebx
popl %ebp
ret
```

CS33 Intro to Computer Systems  XII–23
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
  – 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
Why Bother with a Frame Pointer?

• It points to the beginning of the stack frame
  – making it easy for people to figure out where things are in the frame
  – but people don’t execute the code ...

• The stack pointer always points somewhere within the stack frame
  – it moves about, but the compiler knows where it is pointing
    » a local variable might be at 8(%rsp) for one instruction, but at 16(%rsp) for a subsequent one
    » tough for people, but easy for the compiler

• Thus the frame pointer is superfluous
  – it can be used as a general-purpose register
### x86-64 General-Purpose Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r8</td>
<td>Argument #5</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%r10</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%r11</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%r12</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r13</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r14</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r15</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>
x86-64 Registers

- Arguments passed to functions via registers
  - if more than 6 integral parameters, then pass rest on stack
  - these registers can be used as caller-saved as well

- All references to stack frame via stack pointer
  - eliminates need to update %ebp/%rbp

- Other registers
  - 6 callee-saved
  - 2 caller-saved
  - 1 return value (also usable as caller-saved)
  - 1 special (stack pointer)
x86-64 Long Swap

```c
void swap_l(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

- Operands passed in registers
  - first (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers
- No stack operations required (except ret)
- Avoiding stack
  - can hold all local information in registers
x86-64 Locals in the Red Zone

/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}

• Avoiding stack-pointer change
  – can hold all information within small window beyond stack pointer
    » 128 bytes

swap_a:
movq (%rdi), %rax
movq %rax, -24(%rsp)
movq (%rsi), %rax
movq %rax, -16(%rsp)
movq -16(%rsp), %rax
movq %rax, (%rdi)
movq -24(%rsp), %rax
movq %rax, (%rsi)
ret
x86-64 NonLeaf without Stack Frame

/* Swap a[i] & a[i+1] */
void swap_ele(long a[], int i)
{
    swap(&a[i], &a[i+1]);
}

• No values held while swap being invoked
• No callee-save registers needed
• rep instruction inserted as no-op
  – based on recommendation from AMD
    » can’t handle transfer of control to ret

swap_ele:
    movslq %esi,%rsi          # Sign extend i
    leaq 8(%rdi,%rsi,8), %rax # &a[i+1]
    leaq (%rdi,%rsi,8), %rdi  # &a[i] (1st arg)
    movq %rax, %rsi           # (2nd arg)
    call swap
    rep                      # No-op
    ret
x86-64 Stack Frame Example

```c
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
  (long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += (a[i]*a[i+1]);
}
```

- Keeps values of &a[i] and &a[i+1] in callee-save registers
  - rbx and rbp
- Must set up stack frame to save these registers
  - else clobbered in swap

```asm
swap_ele_su:
  movq  %rbx, -16(%rsp)
  movq  %rbp, -8(%rsp)
  subq  $16, %rsp
  movslq %esi,%rax
  leaq  8(%rdi,%rax,8), %rbx
  leaq  (%rdi,%rax,8), %rbp
  movq  %rbx, %rsi
  movq  %rbp, %rdi
  call  swap
  movq  (%rbx), %rax
  imulq (%rbp), %rax
  addq  %rax, sum(%rip)
  movq  (%rsp), %rbx
  movq  8(%rsp), %rbp
  addq  $16, %rsp
  ret
```
Understanding x86-64 Stack Frame

swap_ele_su:

```assembly
movq  %rbx, -16(%rsp)       # Save %rbx
movq  %rbp, -8(%rsp)        # Save %rbp
subq  $16, %rsp             # Allocate stack frame
movslq %esi,%rax            # Extend i into quad word
leaq  8(%rdi,%rax,8), %rbx  # &a[i+1] (callee save)
leaq  (%rdi,%rax,8), %rbp   # &a[i]    (callee save)
movq  %rbx, %rsi            # 2nd argument
movq  %rbp, %rdi            # 1st argument
call  swap
movq  (%rbx), %rax          # Get a[i+1]
imulq (%rbp), %rax          # Multiply by a[i]
addq  %rax, sum(%rip)        # Add to sum
movq  (%rsp), %rbx          # Restore %rbx
movq  8(%rsp), %rbp         # Restore %rbp
addq  $16, %rsp             # Deallocate frame
ret
```

# Save %rbx
# Save %rbp
# Allocate stack frame
# Extend i into quad word
# &a[i+1] (callee save)
# &a[i]    (callee save)
# 2nd argument
# 1st argument
# Get a[i+1]
# Multiply by a[i]
# Add to sum
# Restore %rbx
# Restore %rbp
# Deallocate frame
Understanding x86-64 Stack Frame

movq %rbx, -16(%rsp)  # Save %rbx
movq %rbp, -8(%rsp)  # Save %rbp

subq $16, %rsp  # Allocate stack frame

movq (%rsp), %rbx  # Restore %rbx
movq 8(%rsp), %rbp  # Restore %rbp

addq $16, %rsp  # Deallocate frame
Quiz 2

swap_ele_su:

```assembly
movq %rbx, -16(%rsp)
movq %rbp, -8(%rsp)
subq $16, %rsp
movslq %esi,%rax
leaq 8(%rdi,%rax,8), %rbx
leaq (%rdi,%rax,8), %rbp
movq %rbx, %rsi
movq %rbp, %rdi
call swap
movq (%rbx), %rax
imulq (%rbp), %rax
addq %rax, sum(%rip)
movq (%rsp), %rbx
movq 8(%rsp), %rbp
addq $16, %rsp
ret
```

Since a 128-byte red zone is allowed, is it necessary to allocate the stack frame by subtracting 16 from %rsp?

a) yes
b) no

# Add to sum
# Restore %rbx
# Restore %rbp
# Deallocate frame
Tail Recursion

```c
int factorial(int x) {
    if (x == 1)
        return x;
    else
        return x*factorial(x-1);
}
```

```c
int factorial(int x) {
    return f2(x, 1);
}
```

```c
int f2(int a1, int a2) {
    if (a1 == 1)
        return a2;
    else
        return f2(a1-1, a1*a2);
}
```
No Tail Recursion (1)

```
x: 6  return addr
x: 5  return addr
x: 4  return addr
x: 3  return addr
x: 2  return addr
x: 1  return addr
```
## No Tail Recursion (2)

<table>
<thead>
<tr>
<th>x: 6</th>
<th>return addr</th>
<th>ret: 720</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 5</td>
<td>return addr</td>
<td>ret: 120</td>
</tr>
<tr>
<td>x: 4</td>
<td>return addr</td>
<td>ret: 24</td>
</tr>
<tr>
<td>x: 3</td>
<td>return addr</td>
<td>ret: 6</td>
</tr>
<tr>
<td>x: 2</td>
<td>return addr</td>
<td>ret: 2</td>
</tr>
<tr>
<td>x: 1</td>
<td>return addr</td>
<td>ret: 1</td>
</tr>
</tbody>
</table>
Tail Recursion

```
ret: 720

a1: 1, a2: 720
return addr
```

```
a1: 2, a2: 360
return addr
```

```
a1: 3, a2: 120
return addr
```

```
a1: 4, a2: 30
return addr
```

```
a1: 5, a2: 6
return addr
```

```
a1: 6, a2: 1
return addr
```
Code: gcc –O1

f2:

    movl %esi, %eax
    cmpl $1, %edi
    je .L5
    subq $8, %rsp
    movl %edi, %esi
    imull %eax, %esi
    subl $1, %edi
    call f2    # recursive call!
    addq $8, %rsp

.L5:

    rep
    ret
Code: gcc -O2

f2:
    cmpl $1, %edi
    movl %esi, %eax
    je .L8

.L12:
    imull %edi, %eax
    subl $1, %edi
    cmpl $1, %edi
    jne .L12

.L8:
    rep
    ret

loop!