Machine Program: Procedure

Shuai MU

Slides from Jinyang Li and Tiger Wang
Roadmap: how does hardware execute a program?

• Where is data stored?
  – Instructions and (most) data are stored in memory
  – Temporary data (e.g. local variables, loop variables) stored in registers

• How does CPU execute a program?
  – Load an instruction from memory according to PC
  – Execute instruction (may access memory)
  – update PC
  – Repeat

• Modes of execution:
  – sequential:
    • PC is changed to point to the next instruction
  – control flow: jmp, conditional jmp
    • PC is changed to point to the jump target address
  – Today → procedure call
Requirements of procedure calls?

1. Passing control
Requirements of procedure calls?

1. Passing control
2. Passing Arguments & return value
Requirements of procedure calls?

1. Passing control
2. Passing Arguments & return value
3. Allocate / deallocate local variables
How to transfer control for procedure calls?

```c
void main(){
    ..
    f(..)
L1: ..
}

void f(){
    ..
    g(..)
L2: ..
}

void g(){
    ..
    h(..)
L3: ..
}
```
How to transfer control for procedure calls?

```c
void main()
    f(..)
L1: .. }  

void f()
    g(..)
L2: .. }  

void g()
    h(..)
L3: .. }  
```

Jump to f()
Remember where to come back
How to transfer control for procedure calls?

void main()
{
  ...
  f(..)
L1: ...
}

void f()
{
  ...
  g(..)
L2: ...
}

void g()
{
  ...
  h(..)
L3: ...
}

Jump to f()
Remember where to come back

Jump to g()
Remember where to come back
How to transfer control for procedure calls?

```c
void main(){
  ...
  f(..)
L1: ...
}

void f(){
  ...
  g(..)
L2: ...
}

void g(){
  ...
  h(..)
L3: ...
}
```

- Jump to f()
- Remember where to come back
- Jump to g()
- Remember where to come back
- Jump to h()
- Remember where to come back
How to transfer control for procedure calls?

void main()
{
   ..
   f(..)
L1: ..
}

void f()
{
   ..
   g(..)
L2: ..
}

void g()
{
   ..
   h(..)
L3: ..
}

Jump to f()
Remember where to come back

Jump to g()
Remember where to come back

Jump to L3
Forget L3
How to transfer control for procedure calls?

void main(){
  ...
  f(..)
L1:  ..
}

void f(){
  ...
  g(..)
L2:  ..
}

void g(){
  ...
  h(..)
L3:  ..
}

Jump to f()
Remember where to come back

Jump to L2
Forget L2

Jump to L3
Forget L3
How to transfer control for procedure calls?

```c
void main(){
  ..
  f(..)
L1: ..
}

void f(){
  ..
  g(..)
L2: ..
}

void g(){
  ..
  h(..)
L3: ..
}
```

- Jump to L1
- Forget L1
- Jump to L2
- Forget L2
- Jump to L3
- Forget L3
How to transfer control for procedure calls?

```c
void main()
{
    ..
    f(..)
}

void f()
{
    ..
    g(..)
}

void g()
{
    ..
    h(..)
}
```

- Jump to L1
- Forget L1
- Jump to L2
- Forget L2
- Jump to L3
- Forget L3

Stack
Stack – push Instruction

`pushq src`

- Decrement `%rsp` by 8
- Write operand at address given by `%rsp`
1. \%rsp = \%rsp – 8
1. %rsp = %rsp - 8
2. mem[%rsp] = %rdi

TOP

BOTTOM
(Initial ESP Value)
Stack – pop Instruction

popq dest

- Store the value at address %rsp to dest
- Increment %rsp by 8
1. \( rsi = \text{mem}[\text{rsp}] \)
1. %rsi = mem[%rsp]
2. %rsp = %rsp + 8
Control transfer from caller to callee

**call label(func name)**

- Push return address on stack
  - Current pc + 8
- Jump label
  - Change the pc to the address of the label

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

int main() {
    int a = 0;
    int b = 2;
    int c = add(a, b);
    printf("%d\n", c);
    return 0;
}
```
Control transfer – call Instruction

call label(func name)
  – Push the return address on stack
    • Return address points to the next instruction after call
  – Jump label
    • Change the pc to label’s value

add:
  leal (%rdi, %rsi), %eax
  ret

main:
  movl $2, %esi
  movl $0, %edi
  call add
  movl %eax, %edx

GCC -Og *.c

return address points to this instruction
Control transfer from callee back to caller

ret
  – Pop 8 bytes from the stack to PC
    • pc = mem[%rsp]

add:
  leal (%rdi,%rsi), %eax
  ret

main:
  movl $2, %esi
  movl $0, %edi
  call add
  movl %eax, %edx

...
<table>
<thead>
<tr>
<th>Memory Offset</th>
<th>Value</th>
<th>Memory Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00...0098</td>
<td>...</td>
<td>0x00...0088</td>
<td>...</td>
</tr>
<tr>
<td>0x00...0090</td>
<td>...</td>
<td>0x00...0080</td>
<td>...</td>
</tr>
<tr>
<td>0x00...0088</td>
<td>...</td>
<td>0x00...0078</td>
<td>...</td>
</tr>
<tr>
<td>0x00...0080</td>
<td>...</td>
<td>0x00...0070</td>
<td>...</td>
</tr>
<tr>
<td>0x00...0078</td>
<td>...</td>
<td>0x00...0068</td>
<td>...</td>
</tr>
<tr>
<td>0x00...0070</td>
<td>...</td>
<td>0x00...0060</td>
<td>...</td>
</tr>
<tr>
<td>0x00...0068</td>
<td>...</td>
<td>0x00...0058</td>
<td>...</td>
</tr>
<tr>
<td>0x00...0060</td>
<td>...</td>
<td>0x00...0050</td>
<td>...</td>
</tr>
<tr>
<td>0x00...0048</td>
<td>...</td>
<td>0x00...0040</td>
<td>movl %eax, %edx</td>
</tr>
<tr>
<td>0x00...0040</td>
<td>...</td>
<td>0x00...0038</td>
<td>call add</td>
</tr>
<tr>
<td>0x00...0038</td>
<td>...</td>
<td>0x00...0030</td>
<td>movl $0, %edi</td>
</tr>
<tr>
<td>0x00...0030</td>
<td>...</td>
<td>0x00...0028</td>
<td>movl $2, %esi</td>
</tr>
<tr>
<td>0x00...0028</td>
<td>...</td>
<td>0x00...0020</td>
<td>ret</td>
</tr>
<tr>
<td>0x00...0020</td>
<td>...</td>
<td>0x00...0018</td>
<td></td>
</tr>
<tr>
<td>0x00...0018</td>
<td>...</td>
<td>0x00...0010</td>
<td>leal (%rdi,%rsi), %eax</td>
</tr>
<tr>
<td>0x00...0010</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CPU**

- **PC:** 0x00...0028
- **IR:** movl $2, %esi
- **RAX:**
- **RBX:**
- **RCX:**
- **RDX:**
- **RSI:**
- **RDI:**
- **RBP:**
- **RSP:** 0x00...0090
- **ZF:** 0
- **SF:** 0
- **OF:** 0
### Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00...0098</td>
<td>...</td>
</tr>
<tr>
<td>0x00...0090</td>
<td>...</td>
</tr>
<tr>
<td>0x00...0088</td>
<td>...</td>
</tr>
<tr>
<td>0x00...0080</td>
<td></td>
</tr>
<tr>
<td>0x00...0078</td>
<td></td>
</tr>
<tr>
<td>0x00...0070</td>
<td></td>
</tr>
<tr>
<td>0x00...0068</td>
<td></td>
</tr>
<tr>
<td>0x00...0060</td>
<td></td>
</tr>
<tr>
<td>0x00...0058</td>
<td></td>
</tr>
<tr>
<td>0x00...0050</td>
<td></td>
</tr>
<tr>
<td>0x00...0048</td>
<td></td>
</tr>
<tr>
<td>0x00...0040</td>
<td></td>
</tr>
<tr>
<td>0x00...0038</td>
<td></td>
</tr>
<tr>
<td>0x00...0030</td>
<td></td>
</tr>
<tr>
<td>0x00...0028</td>
<td></td>
</tr>
<tr>
<td>0x00...0020</td>
<td></td>
</tr>
<tr>
<td>0x00...0018</td>
<td></td>
</tr>
<tr>
<td>0x00...0010</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

### CPU

- **PC:** 0x00...0028
- **IR:** `movl $2, %esi`
- **RAX:**
- **RBX:**
- **RCX:**
- **RDX:**
- **RSI:** 0x2
- **RDI:**
- **RSP:** 0x00...0090
- **RBP:**
- **ZF:** 0
- **SF:** 0
- **CF:** 0
- **OF:** 0
0x00...0098  ...  
0x00...0090  ...  
0x00...0088  
0x00...0080  
0x00...0078  
0x00...0070  
0x00...0068  
0x00...0060  
0x00...0058  
0x00...0050  
0x00...0048  ...  
0x00...0040  movl %eax, %edx  
0x00...0038  call add  
0x00...0030  movl $0, %edi  
0x00...0028  movl $2, %esi  
0x00...0020  ret  
0x00...0018  
0x00...0010  leal (%rdi,%rsi), %eax  
...  

Memory

TOP

BOTTOM

CPU

PC: 0x00...0038
IR: call add
RAX: 
RBX: 
RCX: 
RDX: 
RSI: 0x2
RDI: 0x0
RSP: 0x00...0090
RBP: 
ZF: 0  SF: 0
CF: 0  OF: 0
movl %eax, %edx
movl $0, %edi
movl $2, %esi
ret

leal (%rdi,%rsi), %eax

1. push return address on stack.
1. push return address on stack.

CPU
- PC: 0x00...0038
- IR: call add
- RAX: 
- RBX: 
- RCX: 
- RDX: 
- RSI: 0x2
- RDI: 0x0
- RSP: 0x00...0088
- RBP: 
- ZF: 0
- SF: 0
- CF: 0
- OF: 0

Memory
- 0x00...0098
- 0x00...0090
- 0x00...0088: 0x00...0040
- 0x00...0080
- 0x00...0078
- 0x00...0070
- 0x00...0068
- 0x00...0060
- 0x00...0058
- 0x00...0050
- 0x00...0048
- 0x00...0040
  - movl %eax, %edx
- 0x00...0038
  - call add
- 0x00...0030
  - movl $0, %edi
- 0x00...0028
  - movl $2, %esi
- 0x00...0020
- 0x00...0018
  - ret
- 0x00...0010
  - leal (%rdi,%rsi), %eax
- 0x00...0010
0x00...0098  ...
0x00...0090  ...
0x00...0088  0x00...0040
0x00...0080  ...
0x00...0078  ...
0x00...0070  ...
0x00...0068  ...
0x00...0060  ...
0x00...0058  ...
0x00...0050  ...
0x00...0048  ...
0x00...0040  movl %eax, %edx
0x00...0038  call add
0x00...0030  movl $0, %edi
0x00...0028  movl $2, %esi
0x00...0020  ...
0x00...0018  ret
0x00...0010  leal (%rdi,%rsi), %eax
...

CPU
PC: 0x00...0038
IR: call add
RAX:
RBX:
RCX:
RDX:
RSI: 0x2
RDI: 0x0
RSP: 0x00...0088
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
...

Memory
BOTTOM

TOP

1. push return address on stack.
2. Jump to add
### CPU States

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>0x00...0010</td>
</tr>
<tr>
<td>IR</td>
<td>call add</td>
</tr>
<tr>
<td>RAX</td>
<td></td>
</tr>
<tr>
<td>RBX</td>
<td></td>
</tr>
<tr>
<td>RCX</td>
<td></td>
</tr>
<tr>
<td>RDX</td>
<td></td>
</tr>
<tr>
<td>RDI</td>
<td>0x0</td>
</tr>
<tr>
<td>RSI</td>
<td>0x2</td>
</tr>
<tr>
<td>RSP</td>
<td>0x00...0088</td>
</tr>
<tr>
<td>RBP</td>
<td></td>
</tr>
<tr>
<td>ZF</td>
<td>0</td>
</tr>
<tr>
<td>SF</td>
<td>0</td>
</tr>
<tr>
<td>CF</td>
<td>0</td>
</tr>
<tr>
<td>OF</td>
<td>0</td>
</tr>
</tbody>
</table>

### Memory

```
0x00...0010 | 0x00...0040
0x00...0050 |
0x00...0058 |
0x00...0060 |
0x00...0068 |
0x00...0070 |
0x00...0078 |
0x00...0080 |
0x00...0088 | 0x00...0040
0x00...0090 |
0x00...0098 |
```

### Instructions

1. **push return address on stack.**
2. **Jump to add.**

```assembly
movl %eax, %edx
call add
movl $0, %edi
movl $2, %esi
ret
leal (%rdi,%rsi), %eax
```
1. Push return address on stack.
2. Jump to add
0x00...0098  ...
0x00...0090  ...
0x00...0088  0x00...0040
0x00...0080
0x00...0078
0x00...0070
0x00...0068
0x00...0060
0x00...0058
0x00...0050
0x00...0048  ...
0x00...0040  movl %eax, %edx
0x00...0038  call add
0x00...0030  movl $0, %edi
0x00...0028  movl $2, %esi
0x00...0020
0x00...0018  ret
0x00...0010  leal (%rdi,%rsi), %eax

...
Memory

0x00…0098 ....
0x00…0090 ....
0x00…0088 0x00…0040
0x00…0080 ....
0x00…0078 ....
0x00…0070 ....
0x00…0068 ....
0x00…0060 ....
0x00…0058 ....
0x00…0050 ....
0x00…0048 ....
0x00…0040 movl %eax, %edx
0x00…0038 call add
0x00…0030 movl $0, %edi
0x00…0028 movl $2, %esi
0x00…0020 ....
0x00…0018 ret
0x00…0010 leal (%rdi,%rsi), %eax

...
movl %eax, %edx
call add
movl $0, %edi
movl $2, %esi
ret
leal (%rdi,%rsi), %eax

pop 8 bytes to PC

Memory

CPU
PC: 0x00...0040
IR: ret
RAX: 0x2
RBX: 
RCX: 
RDX: 
RSI: 0x2
RDI: 0x0
RSP: 0x00...0090
RBP: 
ZF: 0
SF: 0
CF: 0
OF: 0
Where to store function arguments and return values?

• Hardware does not dictate where arguments and return value are stored
  – It’s up to the software (compilers).

• Where to put arguments/return value?
  – Arguments and return value are like local variables
  – They are allocated when function is called, de-allocated when function returns.
  – Must do such allocation/de-allocation very fast
Where to store function arguments and return values?

- Two possible designs:
  - Store everything on stack
  - Use registers

- The chosen design → the calling convention
  - All code on a computer system must obey the same convention
  - Otherwise, libraries won’t work
C/UNIX’s calling convention

Registers

First 6 arguments

- `%rdi`
- `%rsi`
- `%rdx`
- `%rcx`
- `%r8`
- `%r9`

Return value

- `%rax`

Stack

- Arg 7
- Arg 8
- Arg n

Only allocate stack space when needed
C’s calling convention: args/return values

Registers

- First 6 Arguments: %rdi, %rsi, %rdx, %rcx, %r8, %r9
- Return value: %rax

```c
int add(int a, int b, int c, int d, int e, int f, int g, int h) {
    int r = a + b + c + d + e + f + g + h;
    return r;
}

int main() {
    int c = add(1, 2, 3, 4, 5, 6, 7, 8);
    printf("%d\n", c);
    return 0;
}
```
C’s calling convention: args/return values

```c
int add(int a, int b, int c, int d, int e, int f, int g, int h) {
    int r = a + b + c + d + e + f + g + h;
    return r;
}
```

```
main:
    subq $8, %rsp
    pushq $8
    pushq $7
    movl $6, %r9d
    movl $5, %r8d
    movl $4, %ecx
    movl $3, %edx
    movl $2, %esi
    movl $1, %edi
    call add
```

```
add:
    addl %esi, %edi
    addl %edi, %edx
    addl %edx, %ecx
    addl %r8d, %ecx
    addl %r9d, %ecx
    movl %ecx, %eax
    addl 8(%rsp), %eax
    addl 16(%rsp), %eax
    ret
```
How to allocate/deallocate local variables?

Use registers whenever possible

Allocate local variables on the stack
  - subq $0x8,%rsp  //allocate 8 bytes
  - movq $1, 8(%rsp)  //store 1 in the allocated 8 bytes
Calling convention:
Caller vs. callee-save registers

When procedure \( f \) calls \( g \):
- \( f \) is the caller, \( g \) is the callee

Can caller assume register values do not change when callee returns?
If not, caller must save register values (in memory) that it needs to use them later
Calling convention: register saving

Some registers are “caller saved”, others are “callee saved”

- Caller saved
  - Caller saves “caller saved” registers on stack before the call
- Callee saved
  - Callee saves “callee saved” registers on stack before using
  - Callee restores them before returning to caller
C’ calling convention: Register Usage

<table>
<thead>
<tr>
<th>Return value</th>
<th>Arguments</th>
<th>callee-saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%rdi</td>
<td>%rbx</td>
</tr>
<tr>
<td></td>
<td>%rsi</td>
<td>%r12</td>
</tr>
<tr>
<td></td>
<td>%rdx</td>
<td>%r13</td>
</tr>
<tr>
<td></td>
<td>%rcx</td>
<td>%r14</td>
</tr>
<tr>
<td></td>
<td>%r8</td>
<td>%rbp</td>
</tr>
<tr>
<td></td>
<td>%r9</td>
<td>%rsp</td>
</tr>
<tr>
<td></td>
<td>%r10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>%r11</td>
<td></td>
</tr>
</tbody>
</table>

Caller-saved

Callee can directly use these registers

Caller can assume these registers are unchanged.
Example

int add2(int a, int b)
{
    return a + b;
}

int add3(int a, int b, int c)
{
    int r = add2(a, b);
    r = r + c;
    return r;
}

add2:
    leal (%edi,%esi), %eax
    ret

add3:
    pushq %rbx
    movl %edx, %ebx
    movl $0, %eax
    call add2
    addl %ebx, %eax
    popq %rbx
    ret

Registers
First 6 Arguments: %rdi, %rsi, %rdx, %rcx, %r8, %9
Return value: %rax
Example

```c
int add2(int a, int b)
{
    return a + b;
}

int add3(int a, int b, int c)
{
    int r = add2(a, b);
    r = r + c;
    return r;
}
```

```
add2:
    leal (%rdi,%rsi), %eax
    ret

add3:
    pushq %rbx  # use %rbx to keep r
    movl %edx, %ebx
    movl $0, %eax
    call add2
    addl %ebx, %eax
    popq %rbx  # restore %rbx before ret
    ret
```

Registers
First 6 Arguments: %rdi, %rsi, %rdx, %rcx, %r8, %9
Return value: %rax