Machine Program: Procedure

Jinyang Li

Some slides are based on Tiger Wang’s
What we’ve learnt (how hardware runs a program)

• Where are data and instructions stored?
  – memory: heap, stack, data, text
  – some local variables may reside in registers only

• Modes of execution:
  – Sequential:
    • PC (%rip) is changed to point to the next instruction
  – Control flow: jmp, conditional jmp
    • PC (%rip) 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?

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: ...
}
How to transfer control for procedure calls?

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

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

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

- Jump to f() (L1)
- Remember where to come back
- Jump to g() (L2)
- Remember where to come back
- Jump to h() (L3)
- 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 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 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(..)
L1:  ..
}

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

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

- 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
Memory

PC: 0x00...0028
IR: pushq %rdi
RAX: 
RBX: 
RCX: 
RDX: 
RSI: 
RDI: 0x5
RSP: 0x7f...0080
RBP: 
ZF: 0
SF: 0
CF: 0
OF: 0

TOP (Current %rsp Value)

BOTTOM (Initial %rsp value)
Stack – pop Instruction

`popq dest`
- Store the value at address `%rsp` to `dest`
- Increment `%rsp` by 8
Memory

0x7f...0098  0x1
0x7f...0090  0x2
0x7f...0088  0x3
0x7f...0080  0x4
0x7f...0078  0x5
0x7f...0070
0x7f...0068
0x7f...0060
0x7f...0058
0x7f...0050
...            TOP (Current %rsp value)
...
...
...
0x00...0030  popq %rsi
0x00...0028  pushq %rdi
0x00...0020
0x00...0018
0x00...0010            PC
...            BOTTOM (Initial %rsp value)

CPU

PC: 0x00...0030
IR: popq %rsi
RAX:
RBX:
RCX:
RDX:
RSI:
RDI: 0x5
RSP: 0x7f...0078
RBP:

ZF: 0  SF: 0
CF: 0  OF: 0
0x7f...0098 0x1
0x7f...0090 0x2
0x7f...0088 0x3
0x7f...0080 0x4
0x7f...0078 0x5
0x7f...0070
0x7f...0068
0x7f...0060
0x7f...0058
0x7f...0050
... 0x00
... 0x00
... 0x00
0x00...0030 0x00
0x00...0028 0x00
0x00...0020 0x00
0x00...0018 0x00
0x00...0010 0x00
... 0x00
---

BOTTOM (Initial ESP Value)

TOP

PC: 0x00...0030
IR: popq %rsi
RAX:
RBX:
RCX:
RDX:
RSI: 0x5
RDI: 0x5
RSP: 0x7f...0080
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
...
Call instruction: control transfer from caller to callee

**call label**
- Push return address on stack
  - “return address” points to instruction immediately after call
- Jump to the address of the label

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

```c
int main() {
    int c = add(0, 2);
    printf("%d\n", c);
    return 0;
}
```
Call instruction: control transfer from caller to callee

call label
  – Push return address on stack
    • “return address” points to instruction immediately after call
  – Jump to the address of the label

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

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

return address points to this instruction

`gcc -Og -S main.c`
Call instruction: control transfer from caller to callee

call label

gcc main.c
objdump –d a.out

00000000000400546 <add>:
  400546:  8d 04 37    lea (%rdi,%rsi,1),%eax
  400549:  c3        retq

0000000000040054a <main>:
  40054a:  48 83 ec 08    sub $0x8,%rsp
  40054e:  be 02 00 00 00    mov $0x2,%esi
  400553:  bf 00 00 00 00    mov $0x0,%edi
  400558:  e8 e9 ff ff ff    callq 400546 <add>
  40055d:  89 c2    mov %eax,%edx
Ret instruction: control transfer from callee back to caller

ret

- Pop 8 bytes from the stack to PC
  - pc = mem[\%rsp], \%rsp = \%rsp +8

0000000000400546 <add>:
  400546: 8d 04 37 lea (%rdi,%rsi,1),%eax
  400549: c3 ret

000000000040054a <main>:
  40054a: 48 83 ec 08 sub $0x8,%rsp
  40054e: be 02 00 00 00 mov $0x2,%esi
  400553: bf 00 00 00 00 mov $0x0,%edi
  400558: e8 e9 ff ff ff callq 400546 <add>
  40055d: 89 c2 mov %eax,%edx
0x00..40054e  
movl $2, %esi  
0x00..400549  
call add  
0x00..400548  
movl $0, %edi  
0x00..40054a  
movl %eax, %edx  
0x00..400546  
sub $0x8,%rsp  
0x00..40054d  
ret  
0x00..400553  
leal (%rdi,%rsi), %eax  
0x00..400555  
movl %eax, %edx  
0x00..400554  
movl %edi, %eax  
0x00..400552  
add %eax, %rax
Memory

0x7f...0098 ...
0x7f...0090 ...
0x7f...0088 0x00..40055d
0x7f...0080 ...
0x7f...0078 ...
0x7f...0070 ...
0x7f...0068 ...
0x7f...0060 ...
0x7f...0058 ...
...
...
0x00..40055d movl %eax, %edx
0x00..400558 call add
0x00..400553 movl $0, %edi
0x00..40054e movl $2, %esi
0x00..40054a sub $0x8,%rsp
0x00..400549 ret
0x00..400546 leal (%rdi,%rsi), %eax
...

CPU

PC: 0x00..400558
IR: call add
RAX: 
RBX: 
RCX: 
RDX: 
RSI: 0x2
RDI: 0x0
RBP: 
RSP: 0x00...0088
ZF: 0
SF: 0
CF: 0
OF: 0
...

TOP

BOTTOM

1.push return address on stack.
0x7f...0098 ...  
0x7f...0090 ...  
0x7f...0088 0x00..40055d  
0x7f...0080  
0x7f...0078  
0x7f...0070  
0x7f...0068  
0x7f...0060  
0x7f...0058  
...  
... ...  
0x00..40055d movl %eax, %edx  
0x00..400558 call add  
0x00..400553 movl $0, %edi  
0x00..40054e movl $2, %esi  
0x00..40054a sub $0x8,%rsp  
0x00..400549 ret  
0x00..400546 leal (%rdi,%rsi), %eax  
...  

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

BOTTOM  
TOP  
PC  

CPU  
PC: 0x00..400546  
IR: call add  
RAX:  
RBX:  
RCX:  
RDX:  
RSI: 0x2  
RDI: 0x0  
RSP: 0x00...0088  
RBP:  
ZF: 0 SF: 0  
CF: 0 OF: 0  

Memory
Memory

CPU

PC: 0x00..400546
IR: leal (%di,%rsi), %eax
RAX: 0x2
RBX: 
RCX: 
RDX: 
RSI: 0x2
RDI: 0x0
RSP: 0x00...0088
RBP: 
ZF: 0
SF: 0
CF: 0
OF: 0
Memory

CPU

PC: 0x00..40055d

IR: ret

RAX: 0x2

RBX: 

RCX: 

RDX: 

RSI: 0x2

RDI: 0x0

RSP: 0x00...0090

RBP: 

ZF: 0  SF: 0  CF: 0  OF: 0

TOP

BOTTOM

0x7f...0098 ...
0x7f...0090 ...
0x7f...0088 0x00..40055d
0x7f...0080
0x7f...0078
0x7f...0070
0x7f...0068
0x7f...0060
0x7f...0058 ...
...
0x00..40055d movl %eax, %edx
0x00..400558 call add
0x00..400553 movl $0, %edi
0x00..40054e movl $2, %esi
0x00..40054a sub $0x8,%rsp
0x00..400549 ret
0x00..400546 leal (%rdi,%rsi), %eax

1. Pop 8 bytes from stack to PC
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 and return value?
  – Arguments and return value 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 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

Registers are much faster than memory but there are only a few of them.
C/UNIX’s calling convention

**Registers**
- First 6 arguments:
  - %rdi
  - %rsi
  - %rdx
  - %rcx
  - %r8
  - %r9

- Return value:
  - %rax

**Stack**
- Only allocate stack space when needed
  - Arg 7
  - Arg 8
  - Arg n
  - ...
Calling convention: args, return vals

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

```
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\b", c);
    return 0;
}
```
Calling convention: args, return vals

```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\b", c);
    return 0;
}
```

**main:**
- `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`

8(%rsp) stores g
16(%rsp) stores h
what does (%rsp) store?
How to allocate/deallocate local vars?

• For primitive data types, use registers whenever possible
• Allocate local array/struct variables on the stack

```c
int main() {
  int a[10];
  clear_array(a, 10);
  return 0;
}
```
Calling convention: Caller vs. callee-save registers

What can the caller assume about the content of a register across function calls?

```c
int foo() {
    int a;    // suppose a is stored in %r12
    a = ....  // compute result of a

    int r = bar();

    int result = r + a;  // does %r12 still store the value of a?
    return result;
}
```
Calling convention: register saving

- **Caller-save**
  - If caller is going to need X’s value after the call, it saves X on stack before the call and restores X after the call
- **Callee-save**
  - If callee is going to use Y, it saves Y on stack before using and restores Y before returning to caller
## Calling convention: Register saving

### Return value

<table>
<thead>
<tr>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
</tr>
<tr>
<td>%rdi</td>
</tr>
<tr>
<td>%rsi</td>
</tr>
<tr>
<td>%rdx</td>
</tr>
</tbody>
</table>

### Arguments

<table>
<thead>
<tr>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rcx</td>
</tr>
<tr>
<td>%r8</td>
</tr>
<tr>
<td>%r9</td>
</tr>
<tr>
<td>%r10</td>
</tr>
<tr>
<td>%r11</td>
</tr>
</tbody>
</table>

### Caller-save

<table>
<thead>
<tr>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
</tr>
<tr>
<td>%rsp</td>
</tr>
</tbody>
</table>

### Callee-save

<table>
<thead>
<tr>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r12</td>
</tr>
<tr>
<td>%r13</td>
</tr>
<tr>
<td>%r14</td>
</tr>
<tr>
<td>%rbp</td>
</tr>
<tr>
<td>%rsp</td>
</tr>
</tbody>
</table>

Callee can directly use these registers.

Caller can assume these registers are unchanged.
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;
}
```

```assembly
add2:
    leal (%rdi,%rsi), %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, %r9
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;
}
```

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

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

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

- `%rdx` (contains `c`) is caller save, i.e. may be changed by `add2`
  - save `%rbx` (callee-save) before overwriting it
  - `%rdx` (contains `c`) is caller save, which is callee save
  - restore `%rbx` before `ret`