



第六讲 运行时环境（基础版）

Runtime Environments

主要内容

- ◎ 运行时环境的作用
- ◎ 运行时环境的设计
- ◎ 运行时环境的实现

- ◎ 对应龙书章节：第 7 章

主要内容

- ◎ 运行时环境的作用
- ◎ 运行时环境的设计
- ◎ 运行时环境的实现

下面的三地址代码如何运行？

```
int fib(int n) {  
    int a; a = 1;  
    int b; b = 1;  
    while (!(n == 0)) {  
        int t;  
        t = a + b; a = b; b = t;  
        n = n - 1;  
    }  
    return a;  
}
```

```
int main() {  
    return fib(10);  
}
```

```
a = 1  
b = 1  
L1:  
    if n == 0 goto L0  
    goto L2  
L2:  
    t0 = a + b  
    t = t0  
    a = b  
    b = t  
    t1 = n - 1  
    n = t1  
    goto L1  
L0:  
    return a
```

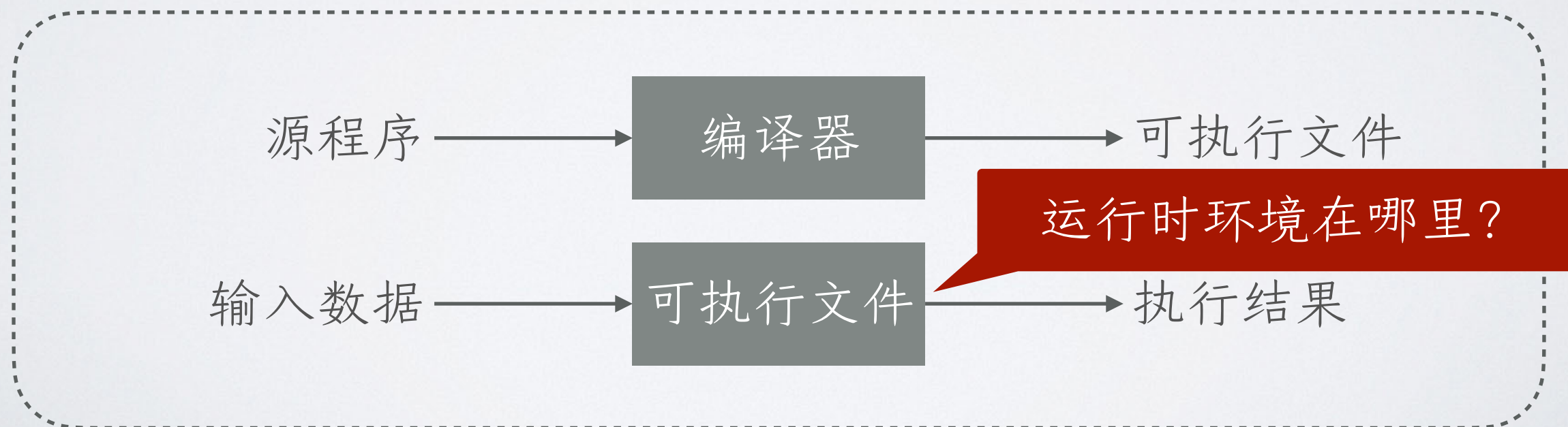
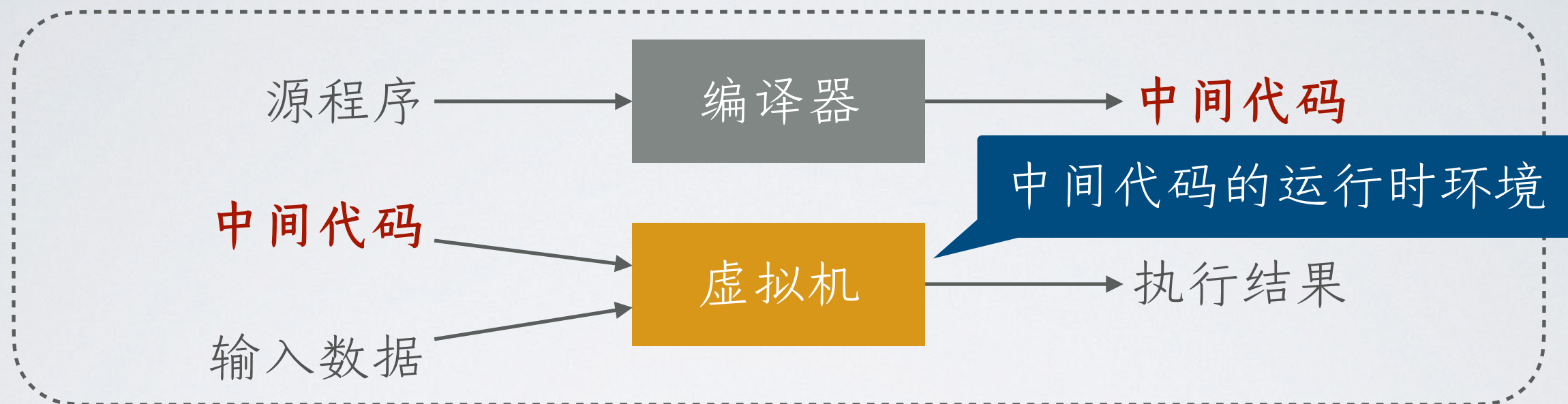
数据如何进行存储和访问？

过程/函数如何进行调用和返回？

```
param 10,1  
t0 = call fib,1  
return t0
```

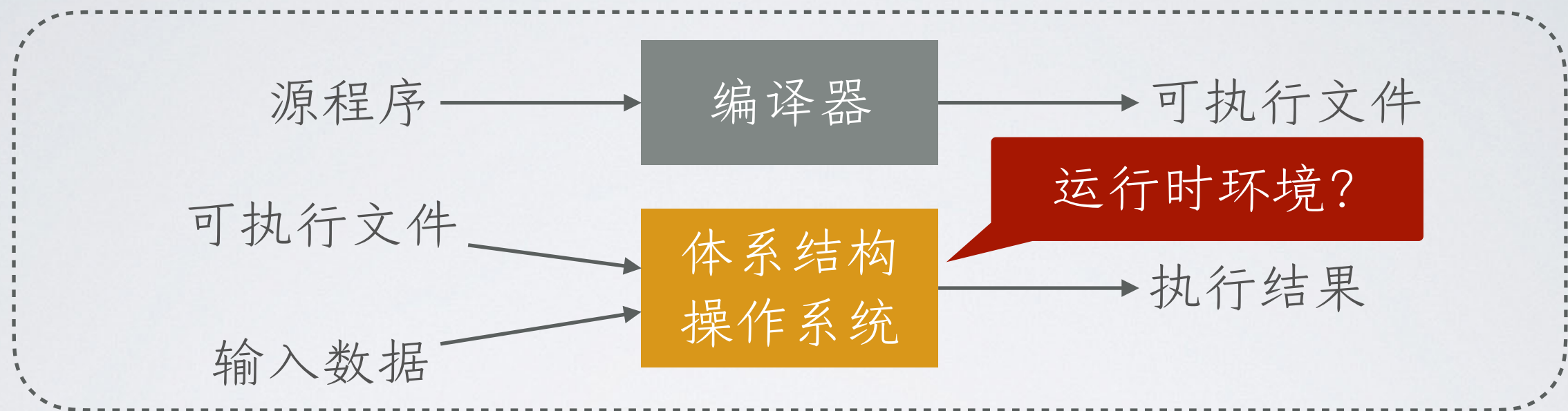
运行时环境

- 运行时环境的主要作用是实现**存储组织**和**过程抽象**



运行时环境

- 运行时环境的主要作用是实现**存储组织**和**过程抽象**



- 问题：运行时环境需要考虑源语言本身的特性**

❖ 比如：面向对象语言需要处理类之间的继承关系、虚函数表等

- 可执行文件 =**

源程序代表的计算 + 通过体系结构/操作系统接口实现的运行时环境

一个运行三地址代码的虚拟机

代码

→ (1) call main, 0
(2) halt
(3) param 10, 1
(4) call fib, 1
(5) retval t₀
(6) return t₀
(7) a = 1
(8) b = 1
(9) if n == 0 goto (18)
(10) goto (11)
(11) t₀ = a + b
(12) t = t₀
(13) a = b
(14) b = t
(15) t₁ = n - 1
(16) n = t₁
(17) goto (9)
(18) return a

虚拟机实现的接口：

- ◉ vm_get(name)
- ◉ vm_set(name, value)
- ◉ vm_param(value, idx)
- ◉ vm_call(name, nargs)
- ◉ vm_ret(value)

```
1.  vm_call("main", 0)
2.  vm_param(10, 1)
3.  vm_call("fib", 1)
4.  vm_set("n", 10)
5.  vm_set("a", 1)
6.  vm_set("b", 1)
7.  vm_get("n") // 10
8.  vm_get("a") // 1
9.  vm_get("b") // 1
10. vm_set("t0", 2)
11. vm_get("t0") // 2
12. vm_set("t", 2)
13. vm_get("b") // 1
```

```
14. vm_set("a", 1)
15. vm_get("t") // 2
16. vm_set("b", 2)
17. vm_get("n") // 10
18. vm_set("t1", 9)
19. vm_get("t1") // 9
20. vm_set("n", 9)
21. ...
22. vm_get("a") // 89
23. vm_ret(89)
24. vm_set("t0", 89)
25. vm_get("t0") // 89
26. vm_ret(89)
```

一个运行三地址代码的虚拟机

代码

```
(1) ...
(2) ...
(3) ...
(4) ...
```

数据

变量	值
a	0
b	0
n	0
t	0
t ₀	0
t ₁	0

虚拟机实现的接口：

- ◉ vm_get(name)
- ◉ vm_set(name, value)
- ◉ vm_param(value, idx)
- ◉ vm_call(name, nargs)
- ◉ vm_ret(value)

状态

当前指令下标

返回地址栈

函数返回值

函数参数栈

pc	(1)
ra	[]
a0	0
st	[]

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

如何实现接口？

一个运行三地址代码的虚拟机

代码

```
(1) ...
(2) ...
(3) ...
(4) ...
```

数据

变量	值
a	0
b	0
n	0
t	0
t ₀	0
t ₁	0

状态

pc	(1)
ra	[]
a0	0
st	[]

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

vm_get(name):

- 查变量表读取变量值

vm_set(name,value):

- 更改变量表中的值

vm_param(value,idx):

- 把参数值压入参数栈

vm_call(name,nargs):

- 把函数返回后的下一条指令的下标压入返回地址栈
- 查函数表找到函数入口的第一条指令
- 根据函数表中的形参信息从参数栈弹出相应的参数值并设置变量表

vm_ret(value):

- 设置 a0 为返回值
- 从 ra 弹出返回后的指令位置

一个运行三地址代码的虚拟机

代码

```

(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
    
```

数据

变量	值
a	0
b	0
n	0
t	0
t ₀	0
t ₁	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(1)
ra	[]
a0	0
st	[]

```
1.  vm_call("main",0)
```

一个运行三地址代码的虚拟机

代码

```

(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
    
```

数据

变量	值
a	0
b	0
n	0
t	0
t ₀	0
t ₁	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(3)
ra	[(2)]
a0	0
st	[]

```

1.  vm_call("main",0)
2.  vm_param(10,1)
    
```

一个运行三地址代码的虚拟机

代码

```

(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
    
```

数据

变量	值
a	0
b	0
n	0
t	0
t ₀	0
t ₁	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(4)
ra	[(2)]
a0	0
st	[10]

```

1.  vm_call("main",0)
2.  vm_param(10,1)
3.  vm_call("fib",1)
4.  vm_set("n",10)
    
```

一个运行三地址代码的虚拟机

代码

```

(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
    
```

数据

变量	值
a	0
b	0
n	10
t	0
t ₀	0
t ₁	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(7)
ra	[(2),(5)]
a0	0
st	[]

```

1.  vm_call("main",0)
2.  vm_param(10,1)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
    
```

一个运行三地址代码的虚拟机

代码

```

(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
    
```

数据

变量	值
a	1
b	0
n	10
t	0
t ₀	0
t ₁	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(8)
ra	[(2),(5)]
a0	0
st	[]

```

1.  vm_call("main",0)
2.  vm_param(10,1)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
    
```

一个运行三地址代码的虚拟机

代码

```

(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
    
```

数据

变量	值
a	1
b	1
n	10
t	0
t ₀	0
t ₁	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(9)
ra	[(2),(5)]
a0	0
st	[]

```

1.  vm_call("main",0)
2.  vm_param(10,1)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n") // 10
    
```

一个运行三地址代码的虚拟机

代码

```

(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
    
```

数据

变量	值
a	1
b	1
n	10
t	0
t ₀	0
t ₁	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(10)
ra	[(2),(5)]
a0	0
st	[]

```

1.  vm_call("main",0)
2.  vm_param(10,1)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n") // 10
    
```

一个运行三地址代码的虚拟机

代码

```

(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
    
```

数据

变量	值
a	1
b	1
n	10
t	0
t ₀	0
t ₁	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(11)
ra	[(2),(5)]
a ₀	0
st	[]

```

1.  vm_call("main",0)
2.  vm_param(10,1)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")    // 10
8.  vm_get("a")    // 1
9.  vm_get("b")    // 1
10. vm_set("t0",2)
    
```

一个运行三地址代码的虚拟机

代码

```

(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
    
```

数据

变量	值
a	1
b	1
n	10
t	0
t ₀	2
t ₁	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(12)
ra	[(2),(5)]
a0	0
st	[]

```

1.  vm_call("main",0)
2.  vm_param(10,1)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")    // 10
8.  vm_get("a")    // 1
9.  vm_get("b")    // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
    
```

一个运行三地址代码的虚拟机

代码

```
(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
```

数据

变量	值
a	1
b	1
n	10
t	2
t ₀	2
t ₁	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(13)
ra	[(2),(5)]
a ₀	0
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10,1)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")    // 10
8.  vm_get("a")    // 1
9.  vm_get("b")    // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b")    // 1
```

```
14. vm_set("a",1)
```

一个运行三地址代码的虚拟机

代码

```
(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
```

数据

变量	值
a	1
b	1
n	10
t	2
t ₀	2
t ₁	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(14)
ra	[(2),(5)]
a ₀	0
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10,1)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n")    // 10
8.  vm_get("a")    // 1
9.  vm_get("b")    // 1
10. vm_set("t0",2)
11. vm_get("t0")  // 2
12. vm_set("t",2)
13. vm_get("b")    // 1
```

```
14. vm_set("a",1)
15. vm_get("t")    // 2
16. vm_set("b",2)
```

一个运行三地址代码的虚拟机

代码

```

(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
    
```

数据

变量	值
a	1
b	2
n	10
t	2
t ₀	2
t ₁	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(15)
ra	[(2),(5)]
a ₀	0
st	[]

```

1.  vm_call("main",0)
2.  vm_param(10,1)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n") // 10
8.  vm_get("a") // 1
9.  vm_get("b") // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b") // 1
    
```

```

14. vm_set("a",1)
15. vm_get("t") // 2
16. vm_set("b",2)
17. vm_get("n") // 10
18. vm_set("t1",9)
    
```

一个运行三地址代码的虚拟机

代码

```

(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
    
```

数据

变量	值
a	1
b	2
n	10
t	2
t ₀	2
t ₁	9

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(16)
ra	[(2),(5)]
a ₀	0
st	[]

```

1.  vm_call("main",0)
2.  vm_param(10,1)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n") // 10
8.  vm_get("a") // 1
9.  vm_get("b") // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b") // 1
    
```

```

14. vm_set("a",1)
15. vm_get("t") // 2
16. vm_set("b",2)
17. vm_get("n") // 10
18. vm_set("t1",9)
19. vm_get("t1") // 9
20. vm_set("n",9)
    
```

一个运行三地址代码的虚拟机

代码

```

(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
    
```

数据

变量	值
a	1
b	2
n	9
t	2
t ₀	2
t ₁	9

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(17)
ra	[(2),(5)]
a ₀	0
st	[]

```

1.  vm_call("main",0)
2.  vm_param(10,1)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n") // 10
8.  vm_get("a") // 1
9.  vm_get("b") // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b") // 1
    
```

```

14. vm_set("a",1)
15. vm_get("t") // 2
16. vm_set("b",2)
17. vm_get("n") // 10
18. vm_set("t1",9)
19. vm_get("t1") // 9
20. vm_set("n",9)
    
```



一个运行三地址代码的虚拟机

若干轮循环后

一个运行三地址代码的虚拟机

代码

```

(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a

```

数据

变量	值
a	89
b	144
n	0
t	144
t ₀	144
t ₁	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(18)
ra	[(2),(5)]
a ₀	0
st	[]

```

1.  vm_call("main",0)
2.  vm_param(10,1)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n") // 10
8.  vm_get("a") // 1
9.  vm_get("b") // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b") // 1

```

```

14. vm_set("a",1)
15. vm_get("t") // 2
16. vm_set("b",2)
17. vm_get("n") // 10
18. vm_set("t1",9)
19. vm_get("t1") // 9
20. vm_set("n",9)
21. ...
22. vm_get("a") // 89
23. vm_ret(89)

```

一个运行三地址代码的虚拟机

代码

```

(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
    
```

数据

变量	值
a	89
b	144
n	0
t	144
t ₀	144
t ₁	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(5)
ra	[(2)]
a ₀	89
st	[]

```

1.  vm_call("main",0)
2.  vm_param(10,1)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n") // 10
8.  vm_get("a") // 1
9.  vm_get("b") // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b") // 1
    
```

```

14. vm_set("a",1)
15. vm_get("t") // 2
16. vm_set("b",2)
17. vm_get("n") // 10
18. vm_set("t1",9)
19. vm_get("t1") // 9
20. vm_set("n",9)
21. ...
22. vm_get("a") // 89
23. vm_ret(89)
24. vm_set("t0",89)
    
```

一个运行三地址代码的虚拟机

代码

```
(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
```

数据

变量	值
a	89
b	144
n	0
t	144
t ₀	89
t ₁	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(6)
ra	[(2)]
a ₀	89
st	[]

```
1.  vm_call("main",0)
2.  vm_param(10,1)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n") // 10
8.  vm_get("a") // 1
9.  vm_get("b") // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b") // 1
```

```
14. vm_set("a",1)
15. vm_get("t") // 2
16. vm_set("b",2)
17. vm_get("n") // 10
18. vm_set("t1",9)
19. vm_get("t1") // 9
20. vm_set("n",9)
21. ...
22. vm_get("a") // 89
23. vm_ret(89)
24. vm_set("t0",89)
25. vm_get("t0") // 89
26. vm_ret(89)
```

一个运行三地址代码的虚拟机

代码

```

(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
  
```

数据

变量	值
a	89
b	144
n	0
t	144
t ₀	89
t ₁	0

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

状态

pc	(2)
ra	[]
a ₀	89
st	[]

```

1.  vm_call("main",0)
2.  vm_param(10,1)
3.  vm_call("fib",1)
4.  vm_set("n",10)
5.  vm_set("a",1)
6.  vm_set("b",1)
7.  vm_get("n") // 10
8.  vm_get("a") // 1
9.  vm_get("b") // 1
10. vm_set("t0",2)
11. vm_get("t0") // 2
12. vm_set("t",2)
13. vm_get("b") // 1
  
```

```

14. vm_set("a",1)
15. vm_get("t") // 2
16. vm_set("b",2)
17. vm_get("n") // 10
18. vm_set("t1",9)
19. vm_get("t1") // 9
20. vm_set("n",9)
21. ...
22. vm_get("a") // 89
23. vm_ret(89)
24. vm_set("t0",89)
25. vm_get("t0") // 89
26. vm_ret(89)
  
```

一个运行三地址代码的虚拟机

代码

(1) ...
(2) ...
(3) ...
(4) ...

数据

变量	值
a	0
b	0
n	0
t	0
t ₀	0
t ₁	0

这个设计有什么问题？

所有变量都处于同一个全局作用域中！

- 可能与源语言的作用域规则不一致
- 也不支持递归过程/函数的实现

状态

当前指令下标

返回地址栈

函数返回值

函数参数栈

pc	(1)
ra	[]
a0	0
st	[]

函数	位置	形参
main	(3)	[]
fib	(7)	[n]

为支持存储组织和过程抽象，需要对运行时环境进行细致的设计

从三地址代码到目标代码

● 可执行文件 =

源程序代表的计算 + 通过体系结构/操作系统接口实现的运行时环境

```
(1)  call main,0
(2)  halt
(3)  param 10,1
(4)  call fib,1
(5)  retval t0
(6)  return t0
(7)  a = 1
(8)  b = 1
(9)  if n == 0 goto (18)
(10) goto (11)
(11) t0 = a + b
(12) t = t0
(13) a = b
(14) b = t
(15) t1 = n - 1
(16) n = t1
(17) goto (9)
(18) return a
```

RISC-V 汇编

```
.data
a:
.zero 4
b:
.zero 4
n:
.zero 4
...

.text
main:
li a0, 10
call fib
ret
fib:
la t6, n
sw a0, 0(t6)
...
```

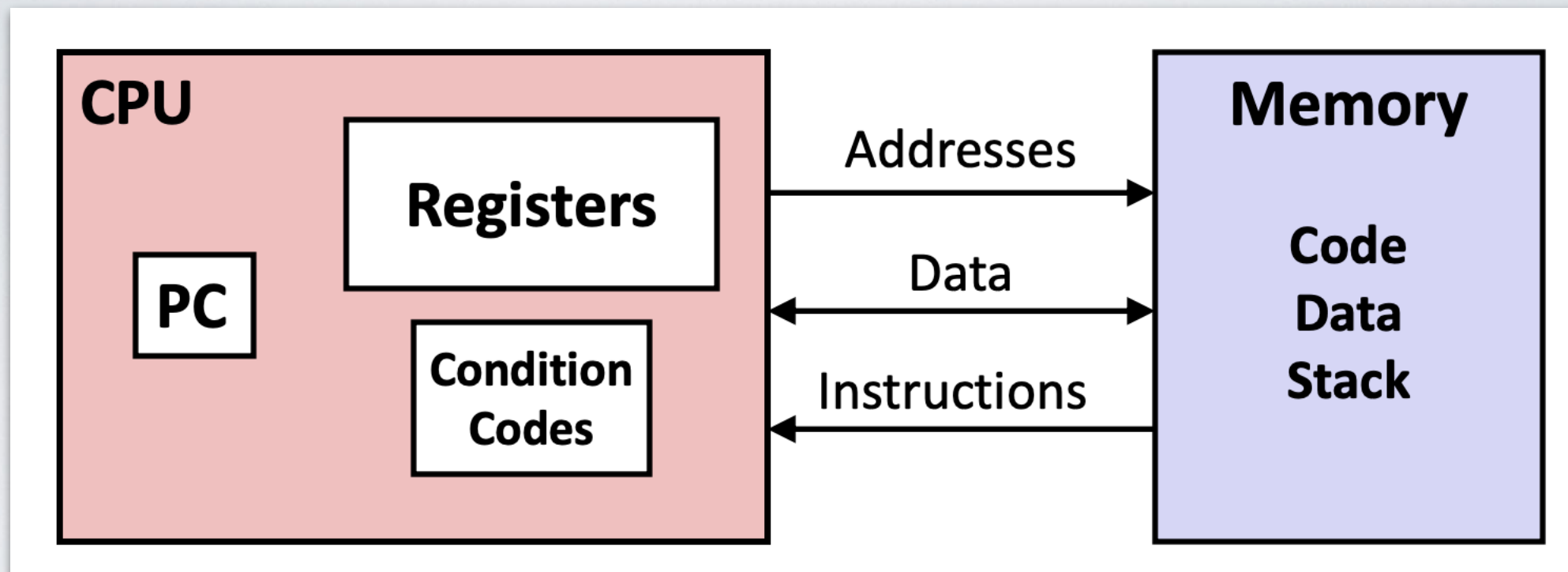
数据段：存储
全局变量

代码段：存储
程序代码

RISC-V 调用约定：寄存器 a0 被
用来传递第一个参数以及返回值

RISC-V 体系结构：寄存器 ra 被
用来记录函数的返回地址

从三地址代码到目标代码



- ◎ 体系结构和操作系统提供了**非常底层**的操作
- ◎ 运行时环境用这些操作来实现数据存储和过程调用
- ◎ 需要在生成的目标代码中插入这些底层操作
 - ❖ 本地(native)运行 $\approx$ 用机器指令「模拟」虚拟机

「机器无关」的运行时环境

- ◎ 本章内容**不限定**本地运行或虚拟机运行,也**不限定**具体的体系结构或操作系统
- ◎ 本章内容着重介绍设计和实现运行时环境的**通用技术**
- ◎ **重点: 存储组织和过程抽象**

主要内容

- ◎ 运行时环境的作用
- ◎ **运行时环境的设计**
- ◎ 运行时环境的实现

存储组织

- ◎ 在代码生成前，编译器需要进行**目标运行环境的设计**和**数据空间的分配**
- ◎ 编译器在操作系统/虚拟机规定的区域中存储生成的**目标代码**与代码运行时的**数据空间**
 - ❖ 比如 RISC-V 中，.text 段存储代码，.data 段存储全局变量等
- ◎ 需要存储空间的对象：
 - ❖ 源代码声明的各种类型的数据对象
 - ❖ 用来保留中间结果和传递参数的临时数据对象
 - ❖ 过程调用时需要记录的上下文信息
 - ❖

静态和动态存储分配

- ◎ **难点：**区分程序的**编译**时刻和**运行**时刻
- ◎ 编译时刻对应**静态分配**
 - ❖ 编译器通过程序文本即可做出分配决定
 - ❖ 例如：常量、全局变量、静态变量(C 中的 static 变量)
- ◎ 运行时刻对应**动态分配**
 - ❖ 程序在运行过程中才能做出分配决定
 - ❖ 例如：局部变量、动态变量(C 中的 malloc 函数分配的数据)
- ◎ **注意：**静态确定的存储空间大小**并不意味**静态分配
 - ❖ 很多时候空间大小可以由类型信息得出

纯静态存储分配

- 所有分配决定都在编译时得到
- 优点：不需要运行时的支持
- 缺点：不支持递归调用过程，不能动态建立数据结构
- 案例：标准 Fortran 语言使用纯静态存储分配

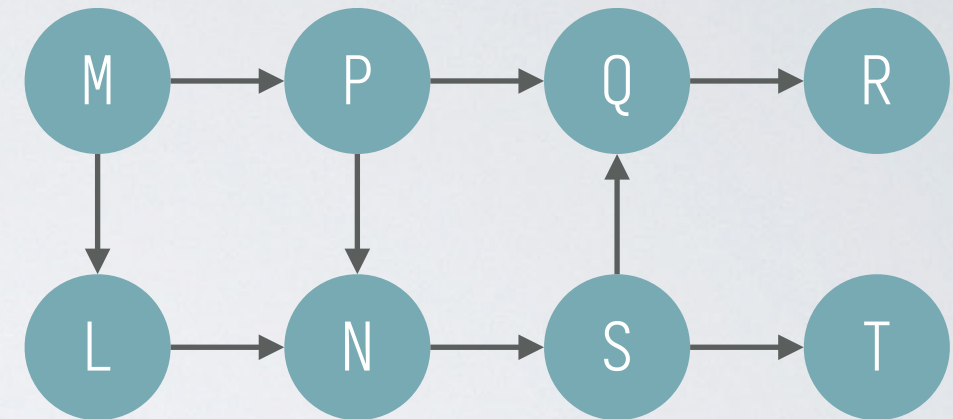
Fortran 语言的存储组织

过程列表

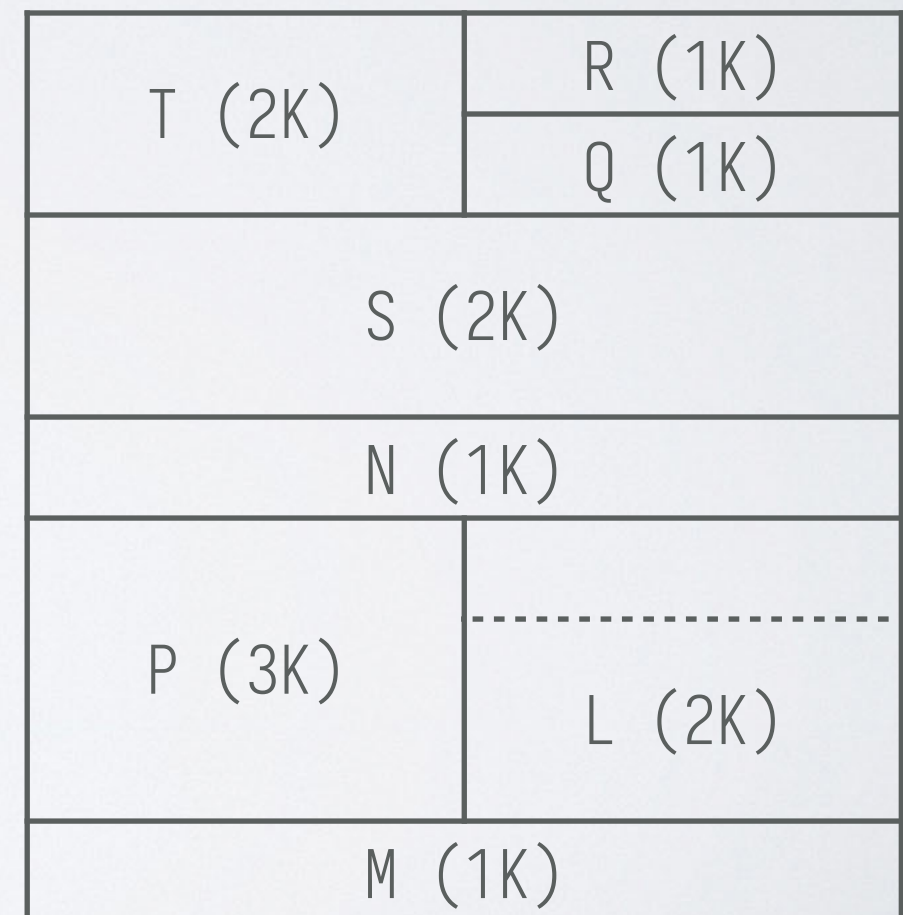
过程名	空间大小
M	1K
P	3K
Q	1K
R	1K
L	2K
N	1K
S	2K
T	2K

M 为主过程

调用关系



存储分配



分时复用!

如何支持递归调用过程？

- ◎ 过程调用的次数一般不能静态确定，所以需要**动态分配**
- ◎ 过程调用在时间上总是嵌套的
 - ❖ 后调用的先返回 (last-in-first-out, LIFO)
 - ❖ 适合使用**栈(stack)**来管理过程活动的存储空间
 - ❖ 例如：过程的局部变量、形式参数等，这些值**与过程的生命周期相同**
- ◎ **栈式存储管理**

如何支持动态建立数据结构？

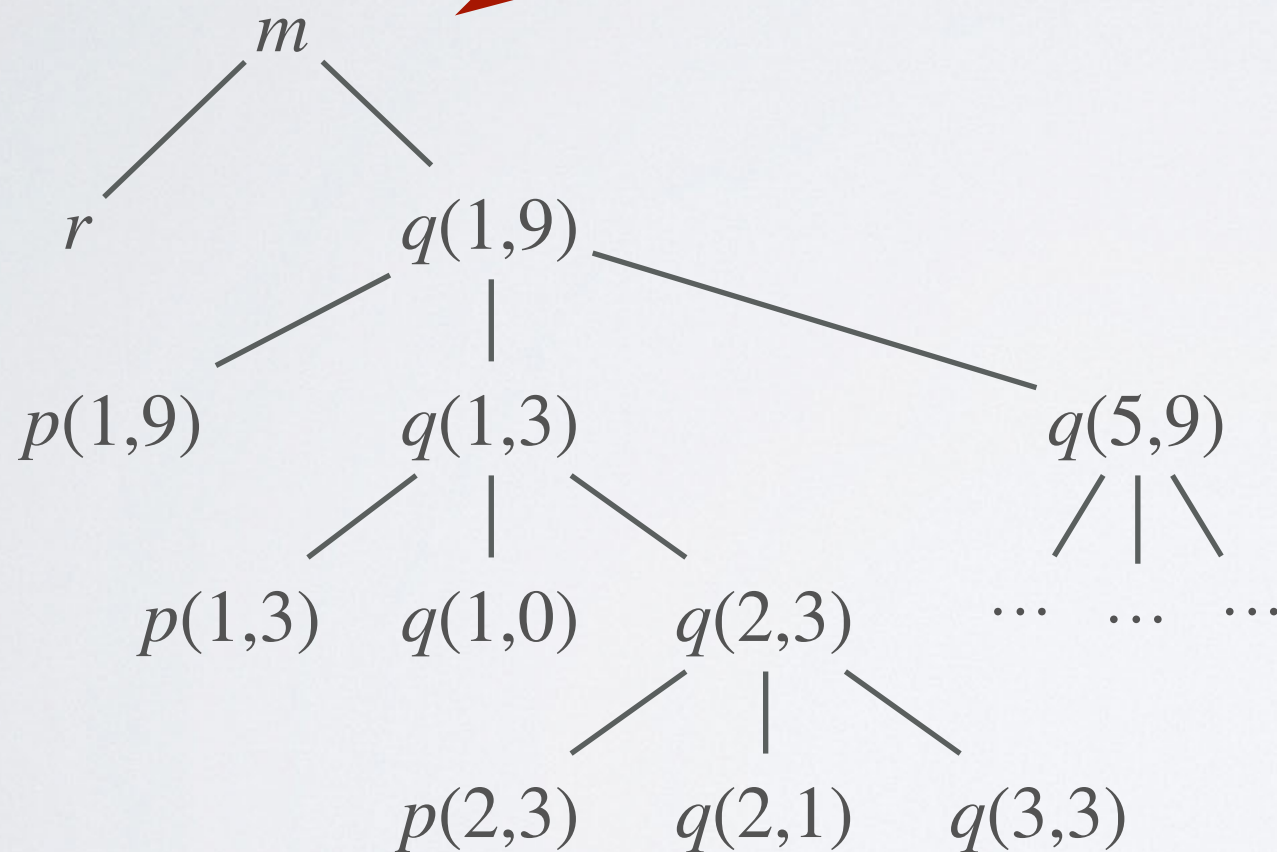
- ◎ 动态数据结构的大小一般不能静态确定，所以需要**动态分配**
- ◎ 其生命周期可能并不和某个过程的调用/返回相匹配
 - ❖ 数据对象可能比创建它的过程调用更长寿
 - ❖ **非词法作用域(non-lexical scope)**
- ◎ **堆(heap)式存储管理**
 - ❖ 不是数据结构中的堆
 - ❖ 可以理解为「**可用内存池**」

过程抽象

- ◎ 所谓「抽象」, 不过是隐藏一定的「细节」
- ◎ 多数现代编程语言都提供了过程抽象
 - ❖ 通过定义过程/函数的形式隐藏其代表的计算的**细节**
 - ❖ 每个过程/函数对外部提供的**抽象**则是其名字、参数、类型声明等
- ◎ **活动树**(activation tree): 程序运行的过程活动可以用树表示
 - ❖ 每个结点对应于一个过程活动
 - ❖ 根结点对应于主过程(或入口过程)的活动
 - ❖ 过程 p 的某次活动对应的结点的所有子结点: 此次活动所调用的各个过程活动(从左到右, 表示调用的先后顺序)

活动树示例

问：前序遍历和后序遍历
分别对应什么？



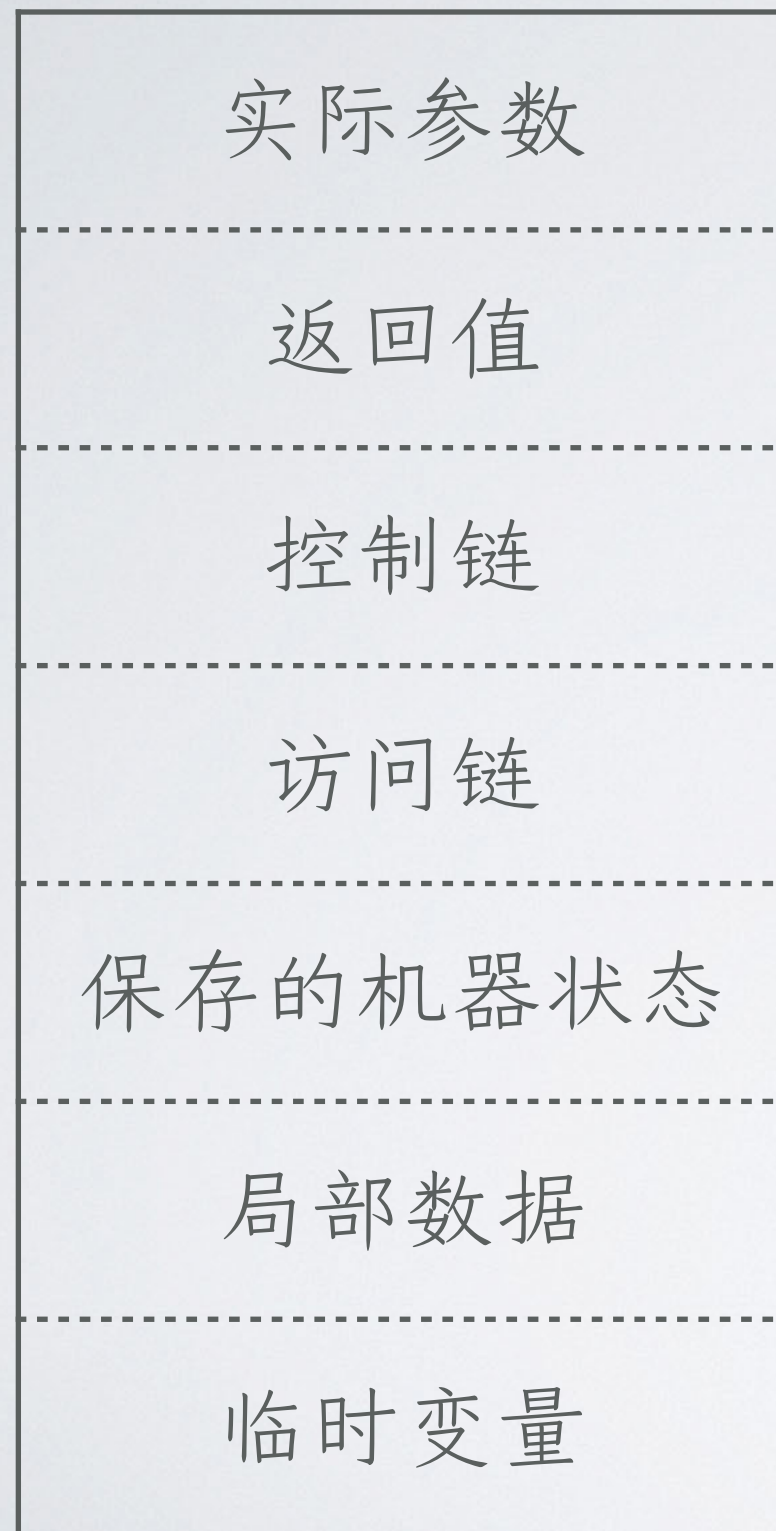
```

enter main()
  enter readArray()
  leave readArray()
  enter quickSort(1,9)
    enter partition(1,9)
    leave partition(1,9)
    enter quickSort(1,3)
      ...
    leave quickSort(1,3)
    enter quickSort(5,9)
      ...
    leave quickSort(5,9)
  leave quickSort(1,9)
leave main()
  
```

活动记录

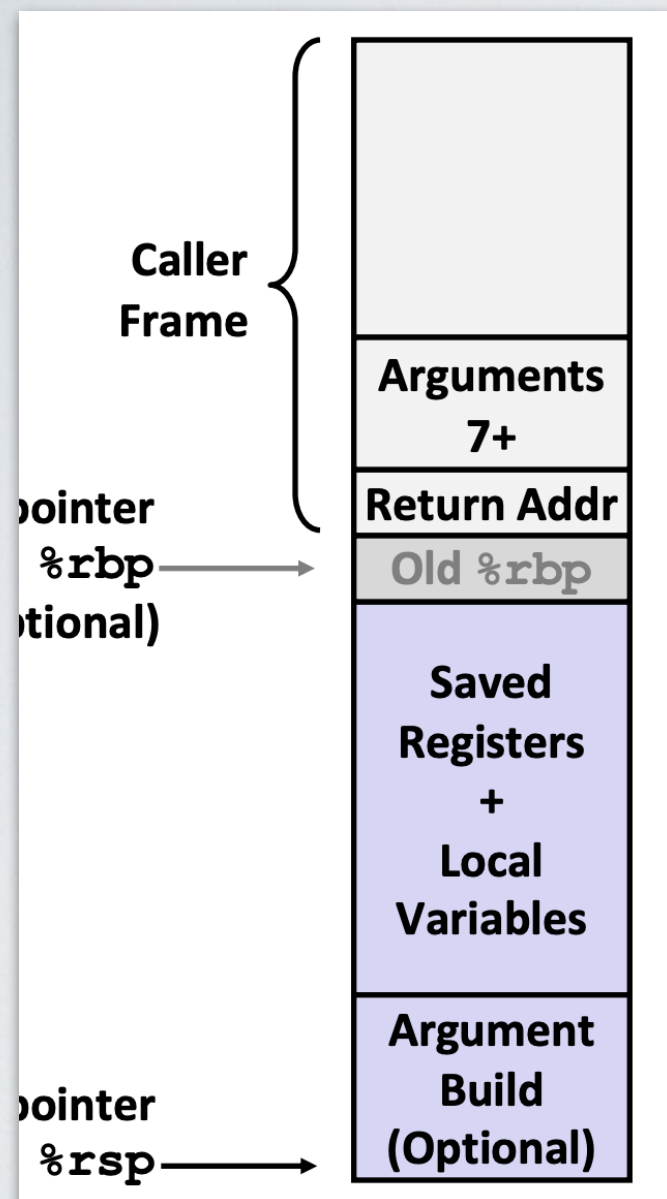
- ◎ 子程序(过程或函数)的一次运行(由于被调用而引发),称为子程序的一次**活动(activity)**
- ◎ 子程序运行时所用到的局部信息(比如局部变量)存储在一个地址连续的存储块内,这个存储块称为**活动记录(activation record)**,又称**帧(frame)**
- ◎ 本地运行:
 - ❖ 现代体系结构通常提供**栈式**活动记录管理,比如 x86、RISC-V 等
 - ❖ **栈帧(stack frame)**
- ◎ 虚拟机运行:
 - ❖ 虚拟机自身通过数据结构进行**栈式或堆式**管理,比如 Java、Lua 等

活动记录的结构



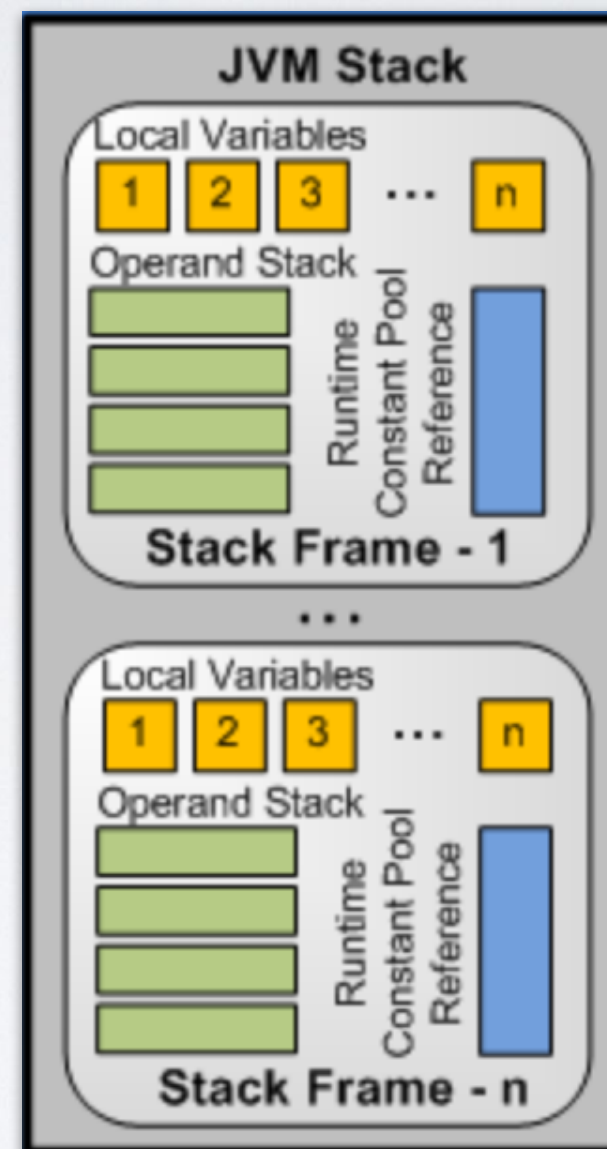
- **保存的机器状态**: 此次调用之前的机器状态信息, 包括**返回地址**
 - **局部数据**: 过程中声明和使用的局部变量
 - **临时变量**: 中间代码或目标代码生成时产生的临时值
 - **控制链**: 指向调用者的活动记录
 - **访问链**: 指向过程中要访问的**非局部数据**所在的活动记录
- ❖ 后续考虑在专题中讲

案例：本地运行和虚拟机运行的帧



x86-64/Linux

- 前 6 个参数通过寄存器传递, 更多的则放在调用者的栈帧里
- 返回地址也放在调用者的栈帧里
- 返回值通过寄存器传递
- 当前栈帧可能存放的 Old %rbp 可认为是实现了控制链

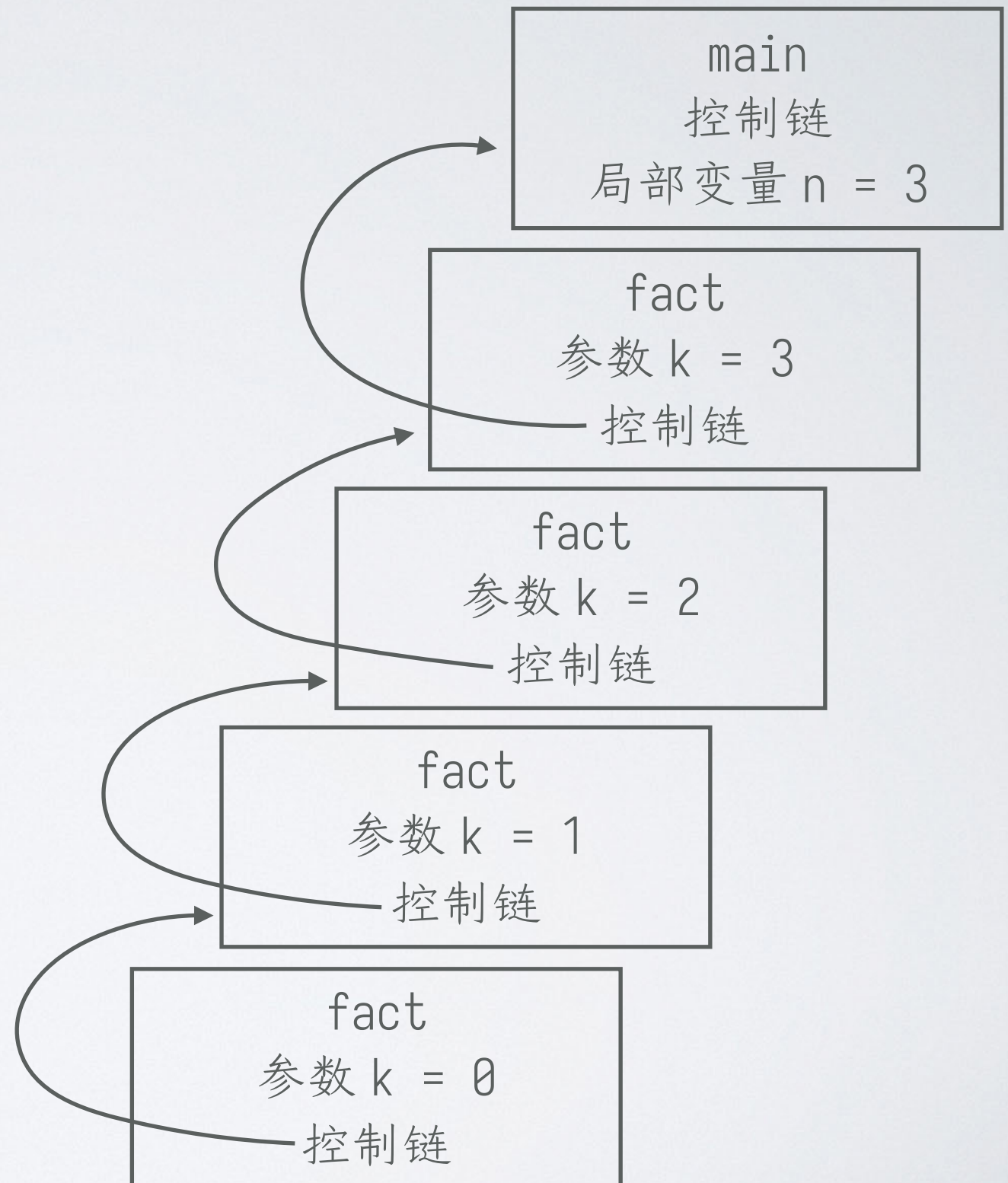


Java VM

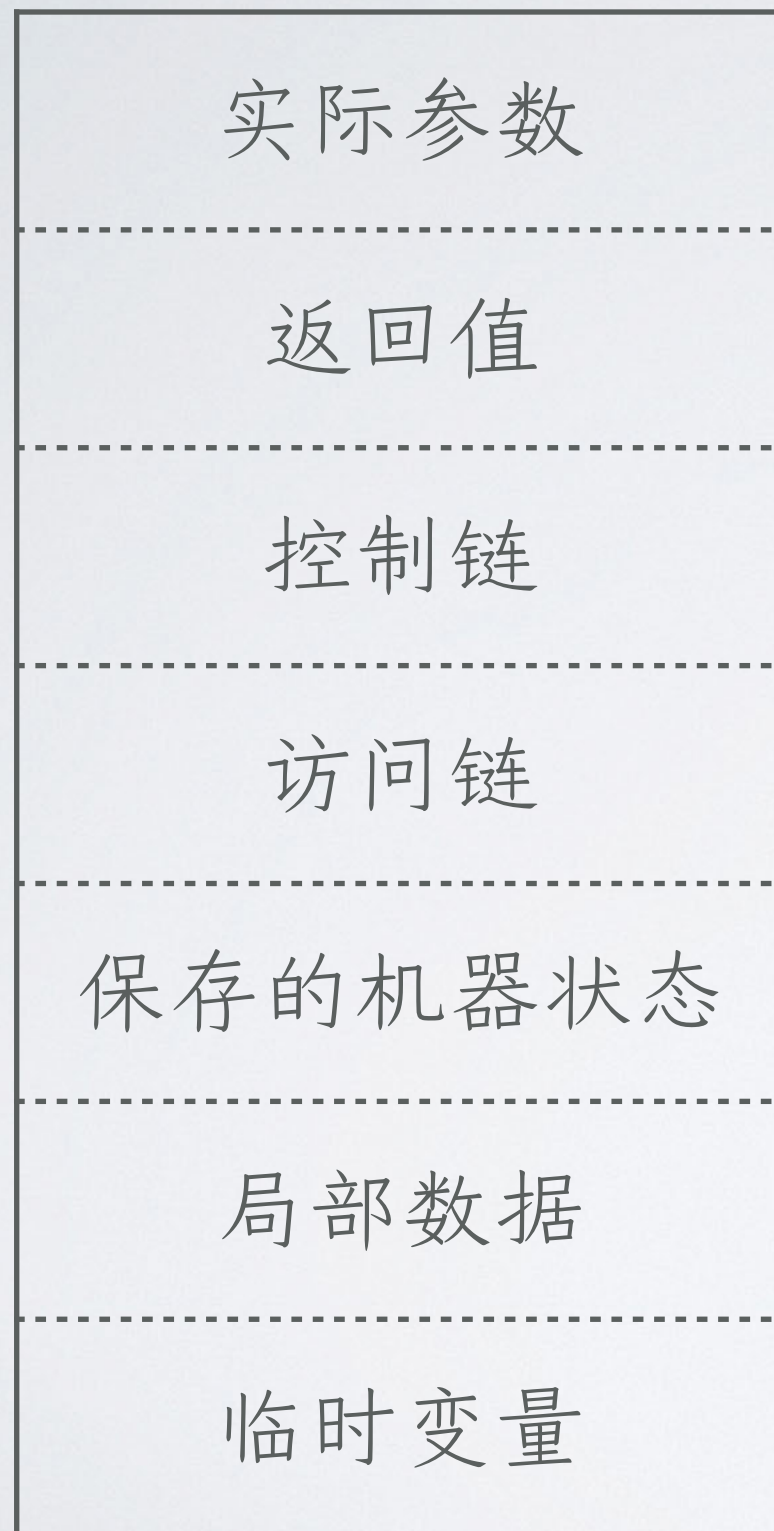
- JVM 没有寄存器, 运算通过栈帧中的操作数栈 (operand stack) 进行
- 返回值会在返回后压入调用者的操作数栈
- 栈帧中还存储了一些跟动态链接相关的数据结构

活动记录示例

```
int fact(int k) {  
    if (k == 0)  
        return 1;  
    else  
        return k * fact(k - 1);  
}  
  
void main() {  
    int n;  
    printf("Enter a number:");  
    scanf("%d", &n);  
    printf("Results = %d\n",  
        fact(n));  
}
```

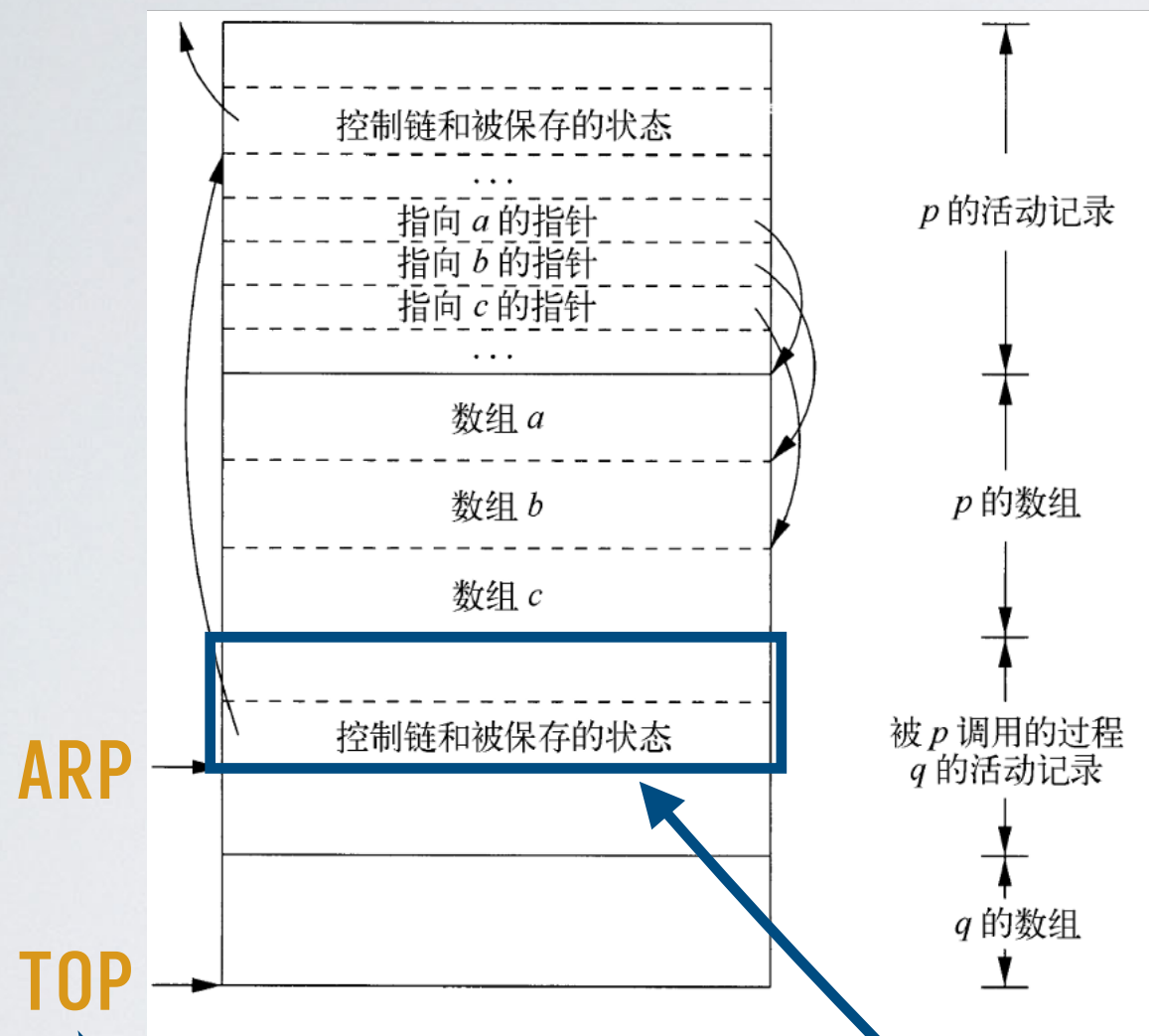


活动记录指针



- 运行时环境中通过**活动记录指针 (activation-record pointer)** 指向正在运行的当前过程的活动记录
- 可以设计 **ARP** 为指向活动记录中**固定长度**数据的末端
 - ❖ 编译器可静态计算得出长度
 - ❖ 生成的代码可以根据相对 **ARP** 的偏移来获取这些数据(比如参数、控制链等)
- 这样的设计也支持运行时**动态确定局部数据的长度**

栈帧中的变长局部数据



栈式活动记录管理：
运行时环境通常会维护一个栈顶指针 TOP

- 如果数据对象的生命周期局限于过程活动的生命周期，就可以分配在运行时刻栈中
- 变长数组也可以放在栈中
- 问：被调用过程 *q* 返回时，如何恢复调用过程 *p* 对应的 ARP 和 TOP 指针？
 - ❖ ARP: 读取控制链存放的指针
 - ❖ TOP: 把(未恢复的) ARP 与被调用过程 *q* 的固定长度字段的长度相加

数据的存储与访问

◎ 常量、全局变量的存储与访问

- ❖ 运行时环境中开辟一块存储空间来存取这些数据
- ❖ 虚拟机可以维护常量表, 并设计读取常量表的指令
- ❖ RISC-V 支持 .data 段放全局变量, 并配合 `la/lw/sw` 等指令进行存取

◎ 过程活动中局部变量的存储与访问

- ❖ 活动记录中的局部数据块可用于存储局部变量的数据
- ❖ 虚拟机可以在活动记录中维护变量名到值的映射表, 并设计读取局部变量表的指令
- ❖ RISC-V、x86 等采取栈式活动记录管理, 在栈上存储局部变量, 访问可以通过相对 **ARP** 或 **TOP** 的偏移来进行

主要内容

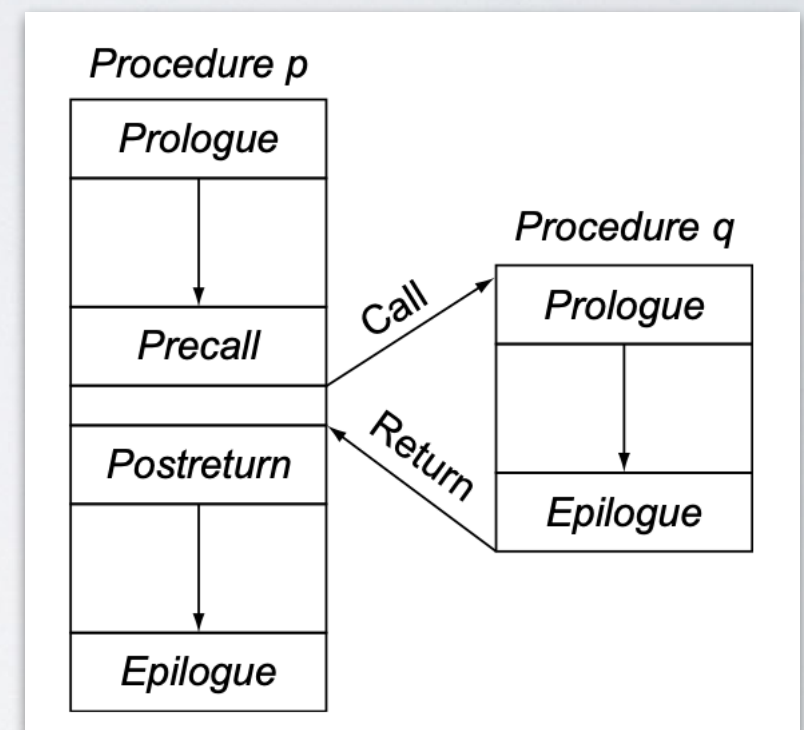
- ◎ 运行时环境的作用
- ◎ 运行时环境的设计
- ◎ 运行时环境的实现

过程抽象的实现

- ◎ 问题：如何创建和维护活动记录？
- ◎ 如果是实现虚拟机……
 - ❖ 你想怎么实现就怎么实现！
- ◎ 如果是需要生成目标代码？
 - ❖ 需要和操作系统和体系结构达成一致
 - ❖ 在目标代码中插入指令实现运行时环境
- ◎ 总体策略：
 - ❖ 调用代码序列：调用者中的 precall 和被调用者的 prologue
 - ❖ 返回代码序列：被调用者的 epilogue 和调用者中的 postreturn

过程链接

- **调用代码序列**: 给被调用过程的活动记录分配空间, 填写记录中的信息
 - ❖ 分割为 precall 和 prologue
- **返回代码序列**: 释放被调用过程的活动记录, 恢复机器状态, 返回到调用过程中继续执行
 - ❖ 分割为 epilogue 和 postreturn
- 根据源语言、体系结构、操作系统的规定, 可以有不同的分割方案
 - ❖ **如果调用者做得多**: 生成的代码会比较长
 - ❖ 每处调用都需要重复生成
 - ❖ **如果被调用者做得多**: 可能会有冗余的存储操作
 - ❖ 考虑被调用者保存 (callee-saved) 寄存器



一种调用代码序列的设计

- ◎ 调用者中的 precall:
 - ❖ 计算实际参数的值, 并存入被调用者的活动记录
 - ❖ 保存机器状态信息(比如 caller-saved 寄存器)
 - ❖ 将返回地址和原来的 **ARP** 指针存入被调用者的活动记录, 并更新 **ARP** 指向被调用者的活动记录
- ◎ 被调用者的 prologue:
 - ❖ 保存机器状态信息(比如 callee-saved 寄存器)
 - ❖ 初始化其局部数据并开始执行
- ◎ 问: 上述过程与 x86-64/Linux 的规定有什么异同?

一种返回代码序列的设计

- ◎ 被调用者的 epilogue:
 - ❖ 在自己的活动记录的返回值字段中放一个返回值
 - ❖ 利用保存的控制链和机器状态的信息, 恢复 **ARP** 和其它相关状态(比如 callee-saved 寄存器)
 - ❖ 按照返回地址转移到调用者的代码之中
- ◎ 调用者中的 postreturn:
 - ❖ 从被调用者的活动记录中获取返回值
 - ❖ 利用保存的机器状态的信息恢复相关状态(比如 caller-saved 寄存器)
- ◎ 问: 上述过程与 x86-64/Linux 的规定有什么异同?

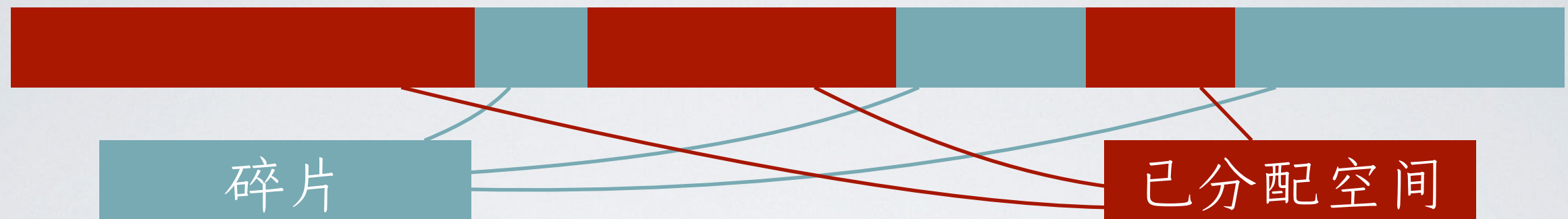
栈式存储管理的实现

- ◎ 如果是实现虚拟机……
 - ❖ 你想怎么实现就怎么实现!
- ◎ 如果是需要生成目标代码?
 - ❖ 现有的体系结构和操作系统基本都提供了方便的栈式存储管理
 - ❖ 比如: RISC-V 的栈指针 `sp`、帧指针 `fp`, 过程调用 `call`、返回 `ret`

堆式存储管理的实现

- ◎ 需要实现的基本功能：分配/回收堆区空间
 - ❖ **分配**：为每个内存请求分配一块连续的、适当大小的堆空间
 - ❖ 必要时请求操作系统增加堆空间
 - ❖ **回收**：把被回收的空间返回空闲空间缓冲池，以满足后续的内存请求
- ◎ 评价存储管理器的特性：
 - ❖ **空间效率**：使程序需要的堆空间最小，即减少**碎片**
 - ❖ **程序效率**：充分利用存储层次结构，提高效率
 - ❖ **额外开销**：使分配/回收内存的操作尽可能高效
- ◎ **问：程序效率和额外开销的区别是什么？**

堆空间的碎片问题



- 随着运行时分配/回收内存，堆区被逐渐割裂成为若干空闲存储块(「窗口」, hole)和已用存储块的交错
- 对于一个分配请求，通常是使用一个足够大的窗口的一部分，其余部分成为更小的窗口
- 对于一个回收请求，被释放的存储块被放回缓冲池，通常需要把连续的窗口**接合(coalesce)**成更大的窗口

堆存储分配策略

- ◎ 假设分配请求的大小为 N 字节
- ◎ **最佳适配 (best-fit)**: 把长度大于或等于 N 的空闲存储块中的长度最小者作为待分配块
- ◎ **首先适配 (first-fit)**: 把构成堆中的空闲存储块中第一个长度大于或等于 N 的作为待分配块
 - ❖ **下次适配 (next-fit)**: 从上次分配的位置开始寻找待分配块
- ◎ **最大适配 (largest-fit)**: 把长度大于或等于 N 的空闲存储块中的长度最大者作为待分配块
- ◎ **问: 什么场景适合最大适配?**

人工存储管理的问题

◎ 内存泄漏(memory leak)

- ❖ 一直未能删除不能被访问的数据

```
void f() {  
    int *a = malloc(sizeof(int));  
    return;  
}
```

◎ 悬空指针解引用(dangling-pointer dereference)

- ❖ 解引用已经被删除的数据

```
int *a = malloc(sizeof(int));  
free(a);  
...  
int b = *a;
```

现代存储管理机制

◎ 静态机制

- ❖ 静态推导对象的生命周期(lifetime)、对象的所有者(ownership)等
- ❖ **RUST: R**egions, **U**niqueness, Owner**s**hip & **T**ypes ([链接](#))

◎ 动态机制

- ❖ 智能指针(smart pointers)
- ❖ **垃圾回收**

◎ 后续考虑在专题中讲

主要内容

- ◎ 运行时环境的作用
- ◎ 运行时环境的设计
- ◎ 运行时环境的实现

- ◎ 实践演示

玩具语言

- ◎ C 语言, 但只有主函数, 只有 int 类型, 只有直线代码

```
int main() {  
    int x = 10;  
    int y = x + 1 * x;  
    int z = x * (y + 1);  
    return z % 66;  
}
```

```
int main() {  
    int y;  
    y = 1;  
    int main;  
    main = y + 1;  
    return main;  
}
```

```
int main() {  
    int a = 1;  
    {  
        a = a + 2;  
        int a = 3;  
        a = a + 4;  
    }  
    a = a + 5;  
    return a;  
}
```

```
int main() {  
    int a = 1;  
    int b = a + 2;  
    {  
        b = a + b;  
        {  
            int a = 3;  
            b = b + a;  
            { int b = 1; }  
            { return b; }  
        }  
        int b = 1;  
    }  
}
```

玩具语言的类三地址 IR

「纯」表达式

$R ::= \text{id}(x) \mid \text{num}(i) \mid \text{add}(R_1, R_2) \mid \text{sub}(R_1, R_2) \mid \text{mul}(R_1, R_2)$

「三」地址指令

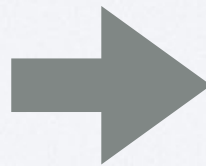
$I ::= \text{move}(x, R) \mid \text{div}(x, R_1, R_2) \mid \text{mod}(x, R_1, R_2) \mid \text{ret}(R) \mid \text{label}(\ell)$

函数体是
指令列表

$F ::= \text{func}(x, \vec{I})$

$P ::= \text{prog}(F)$

```
prog(
  func(main,
    decl(x,
      seq(assign(x, 10),
        decl(y,
          seq(assign(y, x + (1 * x)),
            decl(z,
              seq(assign(z, x * (y + 1)),
                ret(z % 66))))))))))
```



```
main:
// label("main")
x.1 <- 10
// move(x.1, num(10))
y.2 <- x.1 + (1 * x.1)
// move(y.2, add(id(x.1), mul(num(1), id(x.1))))
z.3 <- x.1 * (y.2 + 1)
// move(z.3, mul(id(x.1), add(id(y.2), num(1))))
%t.4 <- z.3 % 66
// mod(%t.4, id(z.3), num(66))
ret %t.4
// ret(id(%t.4))
```

本讲小结

◎ 运行时环境的作用

- ❖ 实现代码运行所需的存储组织和过程抽象

◎ 运行时环境的设计

- ❖ 存储组织: 纯静态存储管理、栈式存储管理、堆式存储管理
- ❖ 过程抽象: 活动记录、控制链

◎ 运行时环境的实现

- ❖ 过程抽象: 调用代码序列、返回代码序列、寄存器的保存和恢复
- ❖ 栈式存储管理、堆式存储管理

◎ 后续可能讲的专题:

- ❖ 过程嵌套、闭包、垃圾回收、基于区域的存储管理等

思考问题

- ◎ 为什么编译过程需要设计运行时环境？
- ◎ 运行时环境中需要保留和维护编程语言的哪些信息？
- ◎ 虚拟机考虑的代码形式通常都是线性的(比如各种字节码)，这是为什么呢？可以设计成图状的代码形式吗？
- ◎ 虚拟机实现难度相对低，但是直接生成的目标机器代码的效率相对高，有没有什么方法可以结合两者的优点？
- ◎ 面向对象语言、函数式语言的运行时环境需要考虑哪些设计问题？你能想象出什么实现难点？
- ◎ 人工智能时代，是否能够面向机器学习代码设计虚拟机？