

A decorative graphic featuring stylized green leaves and swirling vines, positioned horizontally across the upper and lower portions of the slide, framing the central title.

汇编作业讲解2

费翔 feix16@mails.tsinghua.edu.cn
2021.8

- 过程调用以及返回的顺序在一般情况下都是“过程返回的顺序恰好与调用顺序相反”，但是我们可以利用汇编以及对运行栈的理解来编写汇编过程打破这一惯例。有如下汇编代码（x86-32 架构），其中 **GET 过程唯一的输入参数是一个用于存储当前处理器以及栈信息的内存块地址**（假设该内存块的空间足够大），而 SET 过程则用于恢复被 GET 过程所保存的处理器及栈信息，**其唯一的输入参数也是该内存块地址**。在理解代码的基础上，回答下列问题

GET:	SET:
movl 4(%esp), %eax # (A)	movl 4(%esp), %eax
...	...
movl %edi, 20(%eax)	movl 20(%eax), %edi
movl %esi, 24(%eax)	movl 24(%eax), %esi
movl %ebp, 28(%eax)	movl 28(%eax), %ebp
movl %ebx, 36(%eax)	movl 36(%eax), %ebx
movl %edx, 40(%eax)	movl 40(%eax), %edx
movl %ecx, 44(%eax)	movl 44(%eax), %ecx
movl \$1, 48(%eax)	movl ____(%eax), %esp # (D)
movl (%esp), %ecx # (B)	pushl 60(%eax) # (E)
movl %ecx, 60(%eax)	movl 48(%eax), %eax
leal 4(%esp), %ecx # (C)	ret
movl %ecx, 72(%eax)	
movl 44(%eax), %ecx	
movl \$0, %eax	
ret	



GET:	SET:
movl 4(%esp), %eax #(A)	movl 4(%esp), %eax
...	...
movl %edi, 20(%eax)	movl 20(%eax), %edi
movl %esi, 24(%eax)	movl 24(%eax), %esi
movl %ebp, 28(%eax)	movl 28(%eax), %ebp
movl %ebx, 36(%eax)	movl 36(%eax), %ebx
movl %edx, 40(%eax)	movl 40(%eax), %edx
movl %ecx, 44(%eax)	movl 44(%eax), %ecx
movl \$1, 48(%eax)	movl (%eax), %esp #(D)
movl (%esp), %ecx #(B)	pushl 60(%eax) #(E)
movl %ecx, 60(%eax)	movl 48(%eax), %eax
leal 4(%esp), %ecx #(C)	ret
movl %ecx, 72(%eax)	
movl 44(%eax), %ecx	
movl \$0, %eax	
ret	

- SET 过程的返回地址是什么？
【rsp指向的内容就是返回地址（谁改变了rsp）。就是get的返回地址，本来保存在(%esp)中】
- 其返回值是多少？【1，保存在%eax中】



GET:	SET:
movl 4(%esp), %eax # (A)	movl 4(%esp), %eax
...	...
movl %edi, 20(%eax)	movl 20(%eax), %edi
movl %esi, 24(%eax)	movl 24(%eax), %esi
movl %ebp, 28(%eax)	movl 28(%eax), %ebp
movl %ebx, 36(%eax)	movl 36(%eax), %ebx
movl %edx, 40(%eax)	movl 40(%eax), %edx
movl %ecx, 44(%eax)	movl 44(%eax), %ecx
movl \$1, 48(%eax)	movl ____(%eax), %esp # (D)
movl (%esp), %ecx # (B)	pushl 60(%eax) # (E)
movl %ecx, 60(%eax)	movl 48(%eax), %eax
leal 4(%esp), %ecx # (C)	ret
movl %ecx, 72(%eax)	
movl 44(%eax), %ecx	
movl \$0, %eax	
ret	


```

movl 4(%esp), %eax # (A)
    eax=mem[esp+4]
leal 4(%esp), %ecx # (C)
    ecx=esp+4

```

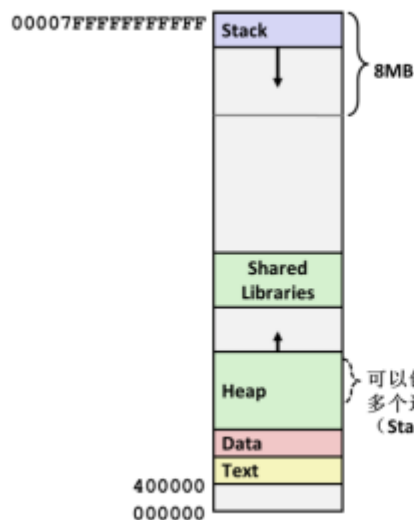
- 代码段中的 (A) 指令执行后，eax 中存放的是什么？【get 的唯一参数，内存块的首地址】
- (B) 指令执行后，ecx 中存放的是什么？【get 的返回地址】
- (C) 指令的作用是什么？【恢复 esp，或者说，保存调用 get 前的栈顶指针】（为什么 +4？后面有 push，隐含 -4）
- (E) 指令的作用是什么？【把 get 的返回地址压入栈，当做 set 的返回地址】
- 并将 (D) 指令补充完整【72】





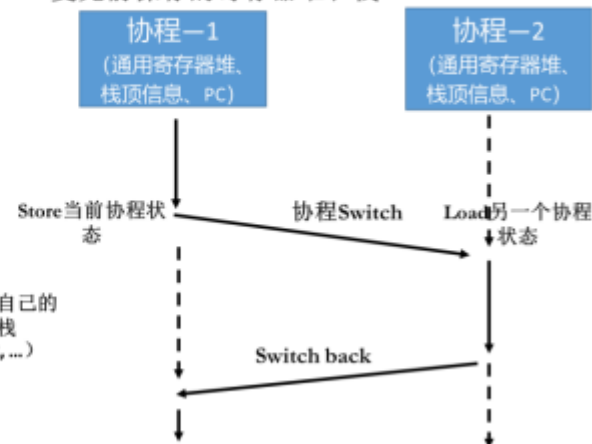
2. 上述的思路也可用于“协程”任务切换（上课讲过，PPT 加下图），

Linux进程的内存布局 (x86-64)



协程：轻量级任务（OS不可见）

- 拥有自己的寄存器堆和栈，协程调度切换时，将寄存器堆和栈保存起来，在切换回来时恢复先前保存的寄存器堆和栈



切换协程用的过程的代码如下：

```
ribs_swapcurcontext:
```

```
    movq    current_ctx, %rsi
```

```
    /* Save the preserved registers. */
```

```
    movq    %rsp, 0(%rsi)
```

```
    movq    %rbx, 8(%rsi)
```

```
    movq    %rbp, 16(%rsi)
```

```
    movq    %r12, 24(%rsi)
```

```
    movq    %r13, 32(%rsi)
```

```
    movq    %r14, 40(%rsi)
```

```
    movq    %r15, 48(%rsi)
```

```
    movq    %rdi, current_ctx
```

```
    /* Load the new stack pointer and the preserved registers.*/
```

```
    movq    0(%rdi), %rsp
```

```
    movq    8(%rdi), %rbx
```

```
    movq    16(%rdi), %rbp
```

```
    movq    24(%rdi), %r12
```

```
    movq    32(%rdi), %r13
```

```
    movq    40(%rdi), %r14
```

```
    movq    48(%rdi), %r15
```

```
    ret
```

将寄存器堆和栈保存起来

恢复先前保存的
寄存器堆和栈

- 2.1 请简要介绍其工作原理；
- rsi是当前协程（旧协程）对应的内存地址；rdi是切换的目标协程（新协程）对应的内存地址。首先保存7个寄存器的值，然后把current_ctx变成切换的目标协程，并从rdi中恢复目标协程的寄存器值，然后正常返回。
- 旧协程的寄存器数据被转移到内存中保存，新协程的数据从内存中取出保存到了寄存器上，完成协程的切换。



切换协程用的过程的代码如下：

ribs_swapcurcontext:

```
    movq    current_ctx, %rsi
    /* Save the preserved registers. */
```

```
    movq    %rsp, 0(%rsi)
```

```
    movq    %rbx, 8(%rsi)
```

```
    movq    %rbp, 16(%rsi)
```

```
    movq    %r12, 24(%rsi)
```

```
    movq    %r13, 32(%rsi)
```

```
    movq    %r14, 40(%rsi)
```

```
    movq    %r15, 48(%rsi)
```

```
    movq    %rdi, current_ctx
```

```
    /* Load the new stack pointer and the preserved registers. */
```

```
    movq    0(%rdi), %rsp
```

```
    movq    8(%rdi), %rbx
```

```
    movq    16(%rdi), %rbp
```

```
    movq    24(%rdi), %r12
```

```
    movq    32(%rdi), %r13
```

```
    movq    40(%rdi), %r14
```

```
    movq    48(%rdi), %r15
```

```
    ret
```

- 2.2 为何 save/load 的通用寄存器个数这么少（x86-64 有 16 个通用寄存器）？

如何使用寄存器作为程序的临时存储？

• 使用惯例——通用寄存器分为两类

- “调用者负责保存”

- Caller在调用子过程之前将这些寄存器内容存储在它的栈帧内

- “被调用者负责保存”

- Callee 在使用这些寄存器之前将其原有内容存储在它的栈帧内
 - 退出前恢复

■ x86-64 寄存器使用惯例


%rax	Return Value	%r8	Argument #5
%rbx	Callee Saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller Saved
%rdx	Argument #3	%r11	Caller Saved
%rsi	Argument #2	%r12	Callee Saved
%rdi	Argument #1	%r13	Callee Saved
%rsp	Stack Pointer	%r14	Callee Saved
%rbp	Callee Saved	%r15	Callee Saved

栈帧指针(可选)

3. 请对照下列的 C 代码与对应的汇编代码, 解释下 C 函数返回 struct 类型是如何实现的? 可以通过画出 call return_struct 时栈的 layout 以及传参情况, 并辅以说明来解释。

```
typedef struct{
int age; int bye; int coo; int ddd; int eee;
} TEST_Struct;
int i = 2;
TEST_Struct __cdecl return_struct(int n)
{
    TEST_Struct local_struct;
    local_struct.age = n;
    local_struct.bye = n;
    local_struct.coo = 2*n;
    local_struct.ddd = n;
    local_struct.eee = n;
    i = local_struct.eee + local_struct.age *2 ;
    return local_struct;
}
int function1()
{
    TEST_Struct main_struct = return_struct(i);
    return 0;
}
```

```
return_struct:
    movq    %rdi, %rax
    movl    %esi, (%rdi)
    movl    %esi, 4(%rdi)
    leal    (%rsi,%rsi), %edx
    movl    %edx, 8(%rdi)
    movl    %esi, 12(%rdi)
    movl    %esi, 16(%rdi)
    addl    %edx, %esi
    movl    %esi, i(%rip)
    ret
function1:
    subq    $32, %rsp
    movl    i(%rip), %esi
    movq    %rsp, %rdi
    call    return_struct
    movl    $0, %eax
    addq    $32, %rsp
    ret
```

return_struct:

```
    movq    %rdi, %rax
    movl    %esi, (%rdi)
    movl    %esi, 4(%rdi)
    leal    (%rsi,%rsi), %edx
    movl    %edx, 8(%rdi)
    movl    %esi, 12(%rdi)
    movl    %esi, 16(%rdi)
    addl    %edx, %esi
    movl    %esi, i(%rip)
    ret
```

function1:

```
    subq    $32, %rsp
    movl    i(%rip), %esi
    movq    %rsp, %rdi
    call    return_struct
    movl    $0, %eax
    addq    $32, %rsp
    ret
```

Func1的返回地址

高地址

rsp

低地址



return_struct:

```
movq    %rdi, %rax
movl    %esi, (%rdi)
movl    %esi, 4(%rdi)
leal    (%rsi,%rsi), %edx
movl    %edx, 8(%rdi)
movl    %esi, 12(%rdi)
movl    %esi, 16(%rdi)
addl    %edx, %esi
movl    %esi, i(%rip)
ret
```

function1:

```
subq    $32, %rsp
movl    i(%rip), %esi
movq    %rsp, %rdi
call    return_struct
movl    $0, %eax
addq    $32, %rsp
ret
```

Func1的返回地址

高地址

rsp

低地址

return_struct:

```
movq    %rdi, %rax
movl    %esi, (%rdi)
movl    %esi, 4(%rdi)
leal    (%rsi,%rsi), %edx
movl    %edx, 8(%rdi)
movl    %esi, 12(%rdi)
movl    %esi, 16(%rdi)
addl    %edx, %esi
movl    %esi, i(%rip)
ret
```

function1:

```
subq    $32, %rsp
movl    i(%rip), %esi
movq    %rsp, %rdi
call    return_struct
movl    $0, %eax
addq    $32, %rsp
ret
```

把i的值放入esi

Func1的返回地址

高地址

rsp

低地址

return_struct:

```
movq    %rdi, %rax
movl    %esi, (%rdi)
movl    %esi, 4(%rdi)
leal    (%rsi,%rsi), %edx
movl    %edx, 8(%rdi)
movl    %esi, 12(%rdi)
movl    %esi, 16(%rdi)
addl    %edx, %esi
movl    %esi, i(%rip)
ret
```

function1:

```
subq    $32, %rsp
movl    i(%rip), %esi
movq    %rsp, %rdi
call    return_struct
movl    $0, %eax
addq    $32, %rsp
ret
```

把栈顶指针rsp
放入rdi, 当做
ret函数的参数

Func1的返回地址

高地址

rsp rdi
低地址

return_struct:

```
movq    %rdi, %rax
movl    %esi, (%rdi)
movl    %esi, 4(%rdi)
leal    (%rsi,%rsi), %edx
movl    %edx, 8(%rdi)
movl    %esi, 12(%rdi)
movl    %esi, 16(%rdi)
addl    %edx, %esi
movl    %esi, i(%rip)
ret
```

function1:

```
subq    $32, %rsp
movl    i(%rip), %esi
movq    %rsp, %rdi
call    return_struct
movl    $0, %eax
addq    $32, %rsp
ret
```

把下一条指令的地址压入栈

Func1的返回地址

ret_str函数的返回地址

高地址

rdi

rsp

低地址

return_struct:

```
    movq    %rdi, %rax
    movl    %esi, (%rdi)
    movl    %esi, 4(%rdi)
    leal    (%rsi,%rsi), %edx
    movl    %edx, 8(%rdi)
    movl    %esi, 12(%rdi)
    movl    %esi, 16(%rdi)
    addl    %edx, %esi
    movl    %esi, i(%rip)
    ret

function1:
    subq    $32, %rsp
    movl    i(%rip), %esi
    movq    %rsp, %rdi
    call    return_struct
    movl    $0, %eax
    addq    $32, %rsp
    ret
```

返回值是结构的首地址，用rax保存。虽然后面没用到

→ 结构体赋值

Func1的返回地址

eee

ddd

coo

bye

age

ret_str函数的返回地址

高地址

```
local_struct.age = n;
local_struct.bye = n;
local_struct.coo = 2*n;
local_struct.ddd = n;
local_struct.eee = n;
i = local_struct.eee
  + local_struct.age *2 ;
```

rdi

rsp

低地址

return_struct:

```
movq    %rdi, %rax
movl    %esi, (%rdi)
movl    %esi, 4(%rdi)
leal    (%rsi,%rsi), %edx
movl    %edx, 8(%rdi)
movl    %esi, 12(%rdi)
movl    %esi, 16(%rdi)
addl    %edx, %esi
movl    %esi, i(%rip)
ret
```

function1:

```
subq    $32, %rsp
movl    i(%rip), %esi
movq    %rsp, %rdi
call    return_struct
movl    $0, %eax
addq    $32, %rsp
ret
```

→ 计算i并保存到全局数据空间

Func1的返回地址

eee

ddd

coo

bye

age

ret_str函数的返回地址


高地址

```
local_struct.age = n;
local_struct.bye = n;
local_struct.coo = 2*n;
local_struct.ddd = n;
local_struct.eee = n;
i = local_struct.eee
  + local_struct.age *2 ;
```

rdi

rsp

低地址



```
return_struct:
```

```
    movq    %rdi, %rax
    movl    %esi, (%rdi)
    movl    %esi, 4(%rdi)
    leal    (%rsi,%rsi), %edx
    movl    %edx, 8(%rdi)
    movl    %esi, 12(%rdi)
    movl    %esi, 16(%rdi)
    addl    %edx, %esi
    movl    %esi, i(%rip)
```

```
ret
```

```
function1:
```

```
    subq    $32, %rsp
    movl    i(%rip), %esi
    movq    %rsp, %rdi
    call    return_struct
    movl    $0, %eax
    addq    $32, %rsp
    ret
```

Func1的返回地址

eee

ddd

coo

bye

age

ret_str函数的返回地址

高地址

rdi

rsp

低地址



return_struct:

```
movq    %rdi, %rax
movl    %esi, (%rdi)
movl    %esi, 4(%rdi)
leal    (%rsi,%rsi), %edx
movl    %edx, 8(%rdi)
movl    %esi, 12(%rdi)
movl    %esi, 16(%rdi)
addl    %edx, %esi
movl    %esi, i(%rip)
ret
```

function1:

```
subq    $32, %rsp
movl    i(%rip), %esi
movq    %rsp, %rdi
call    return_struct
movl    $0, %eax
addq    $32, %rsp
ret
```

Func1的返回地址

eee

ddd

coo

bye

age

ret_str函数的返回地址

高地址

rsp rdi

低地址

return_struct:

```
movq    %rdi, %rax
movl    %esi, (%rdi)
movl    %esi, 4(%rdi)
leal    (%rsi,%rsi), %edx
movl    %edx, 8(%rdi)
movl    %esi, 12(%rdi)
movl    %esi, 16(%rdi)
addl    %edx, %esi
movl    %esi, i(%rip)
ret
```

function1:

```
subq    $32, %rsp
movl    i(%rip), %esi
movq    %rsp, %rdi
call    return_struct
movl    $0, %eax
addq    $32, %rsp
ret
```

后来func1函数
没有用到结构
体，自顾自返
回了

Func1的返回地址

eee

ddd

coo

bye


age

ret_str函数的返回地址

高地址

rsp rdi

低地址



3. 请对照下列的 C 代码与对应的汇编代码, 解释下 C 函数返回 struct 类型是如何实现的? 可以通过画出 `call return_struct` 时栈的 layout 以及传参情况, 并辅以说明来解释。

- 父函数在自己的栈中开辟用于保存结构体的空间, 把结构体的首地址传给子函数, 子函数把返回值保存在该地址中



4. 请分别对照下列的 C 代码与对应的汇编代码, 解释下 C 函数是如何传入 struct 类型参数的? 可以通过画出 call input_struct 时栈的 layout, 并辅以说明来解释。

4.1 gcc -Og ...

```
typedef struct{
int age; int bye; int coo; int ddd; int eee;
} TEST_Struct;
int i = 2;
int input_struct(TEST_Struct in_struct)
{
    return in_struct.eee + in_struct.age*2 ;
}
int function2()
{
    TEST_Struct main_struct;
    main_struct.age = i;
    main_struct.bye = i;
    main_struct.coo = 2*i;
    main_struct.ddd = i;
    main_struct.eee = i;
    return input_struct(main_struct);
}
```

input_struct:

```
movl    8(%rsp), %eax    #age
addl    %eax, %eax
addl    24(%rsp), %eax   #eee
ret
```

function2:

```
subq    $56, %rsp
movl    i(%rip), %eax
movl    %eax, 24(%rsp)   #age
movl    %eax, 28(%rsp)   #bye
leal    (%rax,%rax), %edx
movl    %edx, 32(%rsp)   #coo
movl    %eax, 36(%rsp)   #ddd
movq    24(%rsp), %rdx
movq    %rdx, (%rsp)     #age/bye
movq    32(%rsp), %rdx
movq    %rdx, 8(%rsp)    #coo/ddd
movl    %eax, 16(%rsp)   #eee
call    input_struct
addq    $56, %rsp
ret
```



input_struct:

```
movl    8(%rsp), %eax    #age
addl    %eax, %eax
addl    24(%rsp), %eax    #eee
ret
```

function2:

```
subq    $56, %rsp
movl    i(%rip), %eax

movl    %eax, 24(%rsp)    #age
movl    %eax, 28(%rsp)    #bye
leal    (%rax,%rax), %edx
movl    %edx, 32(%rsp)    #coo
movl    %eax, 36(%rsp)    #ddd

movq    24(%rsp), %rdx
movq    %rdx, (%rsp)      #age/bye

movq    32(%rsp), %rdx
movq    %rdx, 8(%rsp)     #coo/ddd

movl    %eax, 16(%rsp)    #eee
call    input_struct
addq    $56, %rsp
ret
```

func自己的

func复制后给
子函数的

Func1的返回地址

~~eee~~

ddd

coo

bye

age

...

eee

coo/ddd

age/bye

高地址

rsp+24

rsp+16

rsp+8

rsp

低地址



input_struct:

```
movl    8(%rsp), %eax    #age
addl    %eax, %eax
addl    24(%rsp), %eax    #eee
ret
```

function2:

```
subq    $56, %rsp
movl    i(%rip), %eax
movl    %eax, 24(%rsp)    #age
movl    %eax, 28(%rsp)    #bye
leal    (%rax,%rax), %edx
movl    %edx, 32(%rsp)    #coo
movl    %eax, 36(%rsp)    #ddd
movq    24(%rsp), %rdx
movq    %rdx, (%rsp)      #age/bye
movq    32(%rsp), %rdx
movq    %rdx, 8(%rsp)     #coo/ddd
movl    %eax, 16(%rsp)    #eee
call    input_struct
addq    $56, %rsp
ret
```

func自己的

func复制后给
子函数的

Func1的返回地址

~~eee~~

ddd

coo

bye

age

...

eee

coo/ddd

age/bye

高地址

rsp+24

rsp+16

rsp+8

rsp



input_struct:

```
movl    8(%rsp), %eax    #age
addl    %eax, %eax
addl    24(%rsp), %eax    #eee
ret
```

```
return in_struct.eee + in_struct.age*2 ;
```

function2:

```
subq    $56, %rsp
movl    i(%rip), %eax
movl    %eax, 24(%rsp)    #age
movl    %eax, 28(%rsp)    #bye
leal    (%rax,%rax), %edx
movl    %edx, 32(%rsp)    #coo
movl    %eax, 36(%rsp)    #ddd
movq    24(%rsp), %rdx
movq    %rdx, (%rsp)      #age/bye
movq    32(%rsp), %rdx
movq    %rdx, 8(%rsp)     #coo/ddd
movl    %eax, 16(%rsp)    #eee
call    input_struct
addq    $56, %rsp
ret
```

func自己的

func复制后给
予函数的

Func1的返回地址

~~eee~~

ddd

coo

bye

age

...

eee

coo/ddd

age/bye

input函数的返回地址

高地址

新值 旧值

rsp+32 rsp+24

rsp+24 rsp+16

rsp+16 rsp+8

rsp+8 rsp




4. 请分别对照下列的 C 代码与对应的汇编代码，解释下 C 函数是如何传入 struct 类型参数的？可以通过画出 call input_struct 时栈的 layout，并辅以说明来解释。

4.1 gcc -Og ...

- 父函数在自己的栈中复制了一份结构体，保存在 `rsp` 堆栈的栈顶，子函数从 `rsp` 堆栈中读取数据





4.2 gcc -O1/2 ...

C 代码不变。汇编如下：

```
input_struct:
    movl    24(%rsp), %eax
    movl    8(%rsp), %edx
    leal    (%rax,%rdx,2), %eax
    ret
function2:
    movl    i(%rip), %eax
    leal    (%rax,%rax,2), %eax
    ret
```

请分析这段代码，编译器做了什么优化工作。

- 父函数不调用input子函数，编译器理解子函数语义后在父函数本地实现了子函数的功能，类似于inline内联函数的实现



4.3 如果在上面的C代码的 `int input_struct (...)` 声明前加上 `static`, `gcc`

-O1/2 ... 编译后的代码如下:

```
function2:
    movl    i(%rip), %eax
    leal    (%rax,%rax,2), %eax
    ret
```

- 请分析这段代码，编译器做了什么优化工作？
- 在函数 `input_struct` 声明前加上 `static`，即此函数只在该源码文件中可见，不会有其他地方调用它。编译器调用 `gcc -O1/2` 时 `input_struct` 相当于内联函数，其逻辑直接在 `function2` 中实现，所以子函数没有出现在汇编代码中。



5. 请分别对照下列的C代码与对应的汇编代码(编译开关: -S -O2 -fno-stack-protector), 解释下 C99 标准中引入的 variable-length array (简称 VLA, 即允许使用变量定义数组各维) 在这一块代码中是如何实现的? 可以画出函数运行时的栈 layout, 并辅以说明来解释。

read_and_process:

```
pushq    %rbp
movslq   %edi, %rax
leaq     22(,%rax,8), %rax
movq     %rsp, %rbp
pushq    %r14
pushq    %r13
pushq    %r12
pushq    %rbx
andq     $-16, %rax
leal     -1(%rdi), %r13d
subq     %rax, %rsp
testl    %edi, %edi
movq     %rsp, %r14
jle      .L4
leal     -1(%rdi), %eax
movq     %rsp, %rbx
leaq     8(%rsp,%rax,8), %r12
movq     %rax, %r13
```

.L3:

```
xorl     %eax, %eax
addq     $8, %rbx
call     read_val
cltq     # sign-extend eax to all of rax
movq     %rax, -8(%rbx)
cmpq     %r12, %rbx
jne      .L3
```

.L4:

```
movslq   %r13d, %r13
movq     (%r14,%r13,8), %rax
leaq     -32(%rbp), %rsp
popq     %rbx
popq     %r12
popq     %r13
popq     %r14
popq     %rbp
ret
```

```
long read_and_process(int n)
{
    long vals[n];

    for (int i = 0; i < n; i++)
        vals[i] = read_val();
    return vals[n-1];
}
```

rsp



高地址

低地址



5. 请分别对照下列的C代码与对应的汇编代码(编译开关: -S -O2 -fno-stack-protector), 解释下 C99 标准中引入的 variable-length array (简称 VLA, 即允许使用变量定义数组各维) 在这一块代码中是如何实现的? 可以画出函数运行时的栈 layout, 并辅以说明来解释。

read_and_process:

```
pushq    %rbp
movslq   %edi, %rax
leaq     22(,%rax,8), %rax
movq     %rsp, %rbp
pushq    %r14
pushq    %r13
pushq    %r12
pushq    %rbx
andq     $-16, %rax
leal     -1(%rdi), %r13d
subq     %rax, %rsp
testl    %edi, %edi
movq     %rsp, %r14
jle      .L4
leal     -1(%rdi), %eax
movq     %rsp, %rbx
leaq     8(%rsp,%rax,8), %r12
movq     %rax, %r13
```

edi是输入参数n
8n是数组vals的字节长度
多加22个可能用于其他用途

直接跳过整个循环

进入循环前的准备

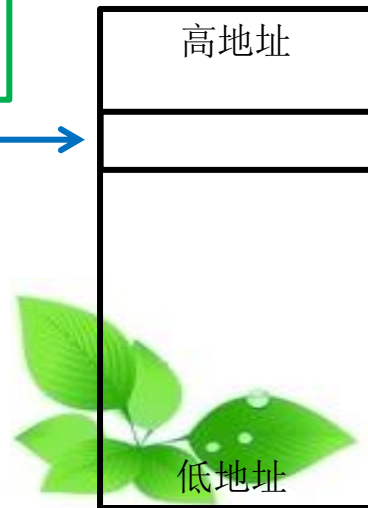
```
.L3:
xorl     %eax, %eax
addq     $8, %rbx
call     read_val
cltq     # sign-extend eax to all of rax
movq     %rax, -8(%rbx)
cmpq     %r12, %rbx
jne      .L3

.L4:
movslq   %r13d, %r13
movq     (%r14,%r13,8), %rax
leaq     -32(%rbp), %rsp
popq     %rbx
popq     %r12
popq     %r13
popq     %r14
popq     %rbp
ret
```

```
long read_and_process(int n)
{
    long vals[n];

    for (int i = 0; i < n; i++)
        vals[i] = read_val();
    return vals[n-1];
}
```

明显是循环体



5. 请分别对照下列的C代码与对应的汇编代码(编译开关: `-S -O2 -fno-stack-protector`), 解释下 C99 标准中引入的 variable-length array (简称 VLA, 即允许使用变量定义数组各维) 在这一块代码中是如何实现的? 可以画出函数运行时的栈 layout, 并辅以说明来解释。

read_and_process:

```
pushq    %rbp                                .L3:
movslq   %edi, %rax                          edi是输入参数n
leaq     22(,%rax,8), %rax                   8n是数组vals的字节长度
                                                多加22个可能用于其他用途
movq     %rsp, %rbp
pushq    %r14
pushq    %r13
pushq    %r12
pushq    %rbx
andq     $-16, %rax
leal     -1(%rdi), %r13d
→ subq    %rax, %rsp                         在栈上开辟一块8n的空间
testl    %edi, %edi
movq     %rsp, %r14
jle      .L4
leal     -1(%rdi), %eax
movq     %rsp, %rbx                         把vals的起始地址给rbx
leaq     8(%rsp,%rax,8), %r12
movq     %rax, %r13
```

```
xorl     %eax, %eax
addq     $8, %rbx
call     read_val
cltq     # sign-extend
movq     %rax, -8(%rbx)
cmpq     %r12, %rbx                         退出for循环的条件
jne      .L3                               是当前计算的地址到
                                                达vals[n], 而非
                                                i<n, 两者等价
```

```
movslq   %r13d, %r13
movq     (%r14,%r13,8), %rax
leaq     -32(%rbp), %rsp
popq     %rbx
popq     %r12
popq     %r13
popq     %r14
popq     %rbp
ret
```

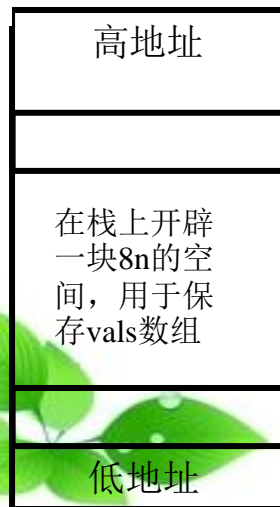
```
long read_and_process(int n)
{
    long vals[n];

    for (int i = 0; i < n; i++)
        vals[i] = read_val();
    return vals[n-1];
}
```

退出for循环的条件
是当前计算的地址到
达vals[n], 而非
i<n, 两者等价

old rsp →

new rsp=rsp-rax →



5. 请分别对照下列的C代码与对应的汇编代码(编译开关: `-S -O2 -fno-stack-protector`), 解释下 C99 标准中引入的 variable-length array (简称 VLA, 即允许使用变量定义数组各维) 在这一块代码中是如何实现的? 可以画出函数运行时的栈 layout, 并辅以说明来解释。

read_and_process:

```
pushq    %rbp
movslq   %edi, %rax
leaq     22(%rax,8), %rax
movq     %rsp, %rbp
pushq    %r14
pushq    %r13
pushq    %r12
pushq    %rbx
andq     $-16, %rax
leal     -1(%rdi), %r13d
subq     %rax, %rsp
testl    %edi, %edi
movq     %rsp, %r14
jle      .L4
leal     -1(%rdi), %eax
movq     %rsp, %rbx
leaq     8(%rsp,%rax,8), %r12
movq     %rax, %r13
```

edi是输入参数n
8n是数组vals的字节长度
多加22个可能用于其他用途

在栈上开辟一块8n的空间

.L3:

```
xorl     %eax, %eax
addq     $8, %rbx
call     read_val
cltq     # sign-extend eax to all of rax
movq     %rax, -8(%rbx)
cmpq     %r12, %rbx
jne      .L3
```

.L4:

```
movslq   %r13d, %r13
movq     (%r14,%r13,8), %rax
leaq     -32(%rbp), %rsp
popq     %rbx
popq     %r12
popq     %r13
popq     %r14
popq     %rbp
ret
```

为什么会有-32的
偏移量, 因为后面
有4个pop

```
long read_and_process(int n)
{
    long vals[n];

    for (int i = 0; i < n; i++)
        vals[i] = read_val();
    return vals[n-1];
}
```

old rsp

rsp=rsp-rax

高地址

在栈上开辟
一块8n的空间, 用于保
存vals数组

低地址

5. 请分别对照下列的C代码与对应的汇编代码(编译开关: -S -O2 -fno-stack-protector), 解释下 C99 标准中引入的 variable-length array (简称 VLA, 即允许使用变量定义数组各维) 在这一块代码中是如何实现的? 可以画出函数运行时的栈 layout, 并辅以说明来解释。

read_and_process:

```
pushq    %rbp
movslq   %edi, %rax
leaq     22(,%rax,8), %rax
movq     %rsp, %rbp
```

edi是输入参数n
8n是数组vals的字节长度
多加22个可能用于其他用途

```
pushq    %r14
pushq    %r13
pushq    %r12
pushq    %rbx
andq     $-16, %rax
leal     -1(%rdi), %r13d
subq     %rax, %rsp
```

在栈上开辟一块8n的空间

```
testl    %edi, %edi
movq     %rsp, %r14
jle      .L4
leal     -1(%rdi), %eax
movq     %rsp, %rbx
leaq     8(%rsp,%rax,8), %r12
movq     %rax, %r13
```

代码放这里, 则可以删除-32的偏移量, 和左侧的操作顺序相反

```
leaq     (%rbp), %rsp
```

.L3:

```
xorl     %eax, %eax
addq     $8, %rbx
call     read_val
cltq     # sign-extend eax to all of rax
movq     %rax, -8(%rbx)
cmpq     %r12, %rbx
jne      .L3
```

.L4:

```
movslq   %r13d, %r13
movq     (%r14,%r13,8), %rax
```

```
popq     %rbx
popq     %r12
popq     %r13
popq     %r14
popq     %rbp
ret
```

```
long read_and_process(int n)
{
    long vals[n];

    for (int i = 0; i < n; i++)
        vals[i] = read_val();
    return vals[n-1];
}
```

old rsp →

rsp=rsp-rax →

高地址

在栈上开辟一块8n的空间, 用于保存vals数组

低地址

5. 请分别对照下列的C代码与对应的汇编代码(编译开关: -S -O2 -fno-stack-protector), 解释下 C99 标准中引入的 variable-length array (简称 VLA, 即允许使用变量定义数组各维) 在这一块代码中是如何实现的? 可以画出函数运行时的栈 layout, 并辅以说明来解释。

read_and_process:

```
pushq    %rbp
movslq   %edi, %rax
leaq     22(,%rax,8), %rax
movq     %rsp, %rbp
pusnq    %r14
pushq    %r13
pushq    %r12
pushq    %rbx
andq     $-16, %rax
leal     -1(%rdi), %r13d
subq     %rax, %rsp
testl    %edi, %edi
movq     %rsp, %r14
jle      .L4
leal     -1(%rdi), %eax
movq     %rsp, %rbx
leaq     8(%rsp,%rax,8), %r12
movq     %rax, %r13
```

edi是输入参数n
8n是数组vals的字节长度
多加22个可能用于其他用途

在栈上开辟一块8n的空间

代码放这里, 则可以删除-32的偏移量, 和左侧的操作顺序相反

leaq (%rbp), %rsp

.L3:

.L4:

```
movslq   %r13d, %r13
movq     (%r14,%r13,8), %rax
popq     %rbx
popq     %r12
popq     %r13
popq     %r14
popq     %rbp
ret
```

程序在自己的栈中开辟8n长度的内存空间, 用于保存vals数组。当退出该程序时, 需要恢复栈顶指针rsp, 相当于释放vals数组对应的内存空间

问题: 开辟8n长度的内存空间时, 一定需要修改rsp吗?

```
long read_and_process(int n)
{
    long vals[n];

    for (int i = 0; i < n; i++)
        vals[i] = read_val();
    return vals[n-1];
}
```

old rsp →

rsp=rsp-rax →

高地址

在栈上开辟一块8n的空间, 用于保存vals数组

低地址