Assembly

תרגול 6

פונקציות והתקפת buffer

Modified by Eyal Cohen

Procedures (Functions)

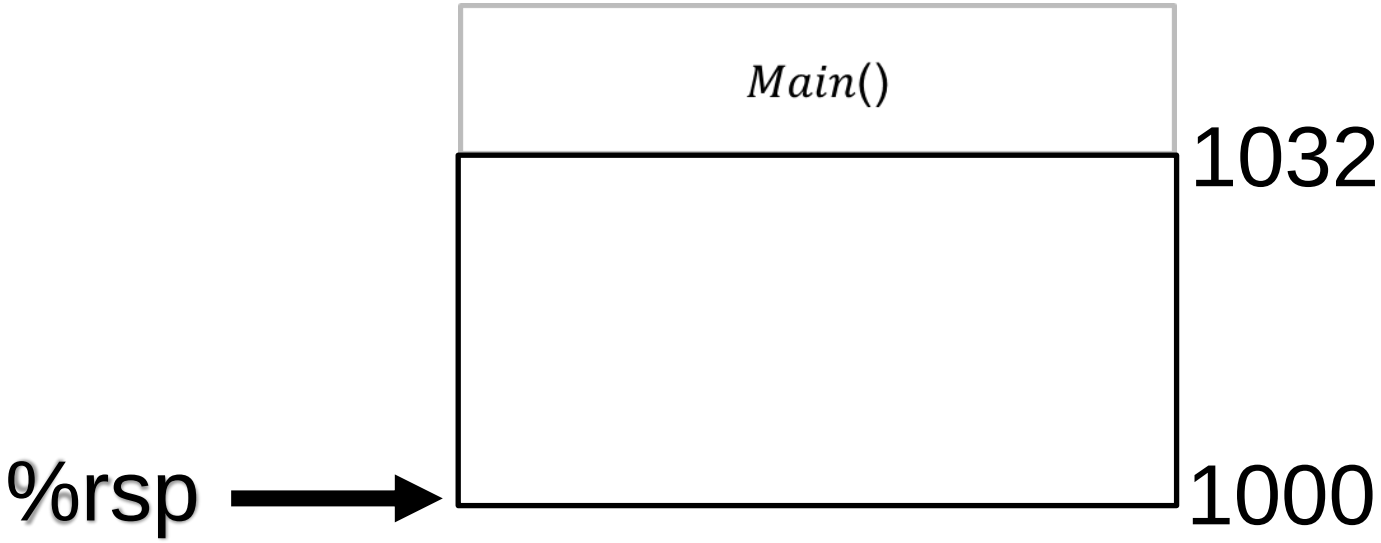
- A procedure call involves passing both data and control from one part of the code to another.
  - Data = procedure parameters and return values.
- When passing control to another procedure, the procedure might allocate space for the local variables of the procedure on entry and deallocate them on exit.
- X86-64 provides only simple instructions for transferring control to and from procedures. The passing of data and the allocation and deallocation of local variables is handled by manipulating the registers and program stack.

Using the Stack

- X86-84 programs make use of the program stack to support procedure calls.
- The stack is used to:
  - pass (some) procedure arguments.
  - store return information.
  - save registers for later restoration
  - local storage.
- The portion of the stack allocated for a single procedure call is called a *stack frame*.

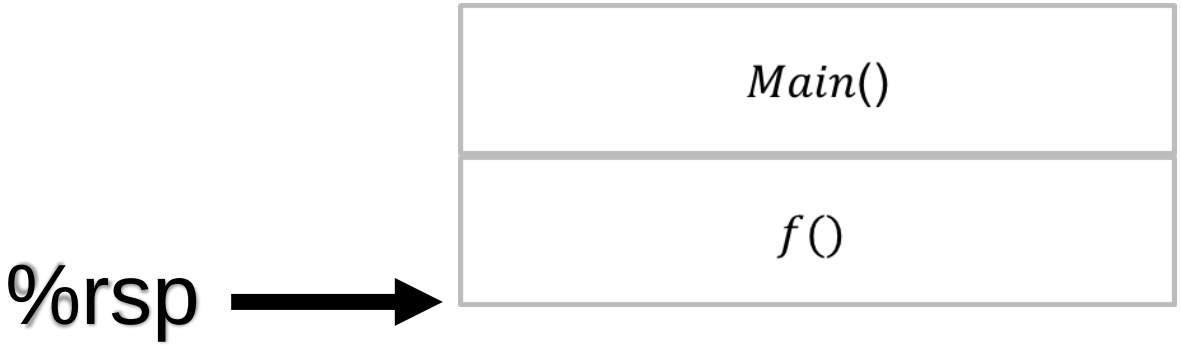
The Stack Segment

The Stack Segment

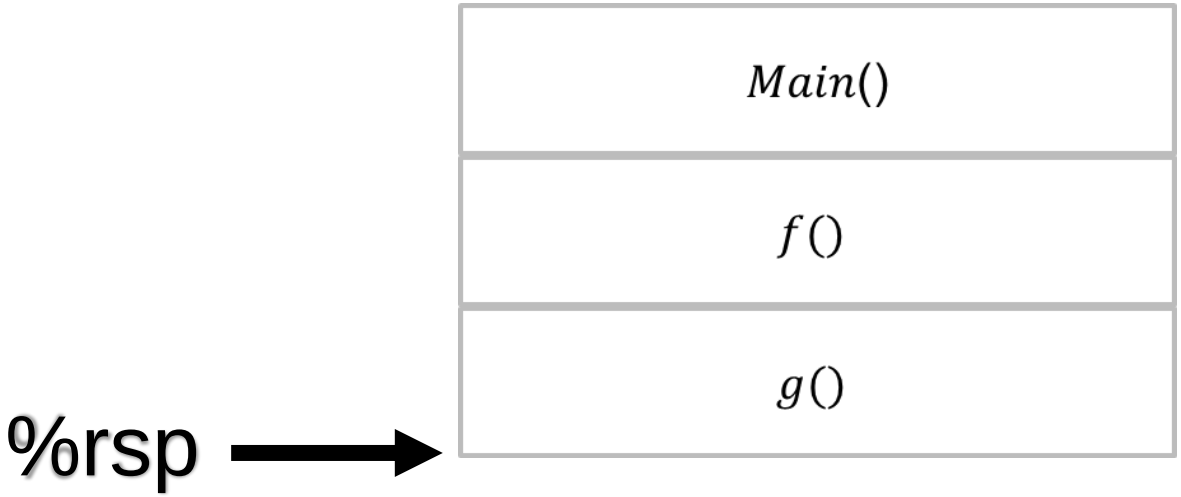


The Stack Segment

The Stack Segment



The Stack Segment



Transferring Control Instructions

Instruction		Description
`call`	*Label*	Procedure Call
`call`	**Operand*	Procedure Call
`leave`		Prepare stack for return
`ret`		Return from call

← Optional

`call` Instruction

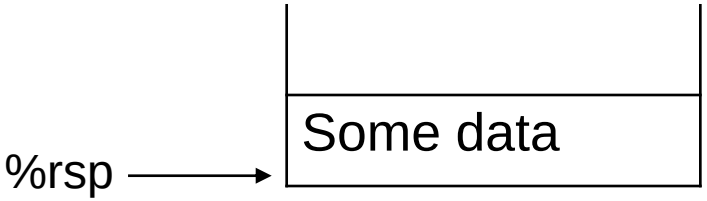
- Similar to the Jump (`jmp`) instruction.
  - Has a target indicating the address of the instruction where the called procedure starts.
  - Can either be direct (with label) or indirect (*Operand).
- The effect of a call instruction is to:
  - Push a return address on the stack.
  - Jump to the start of the called procedure.
- The return address is the address of the instruction immediately following the call in the program, so that execution will resume at this location when the called procedure returns.

`call` Instruction – Example

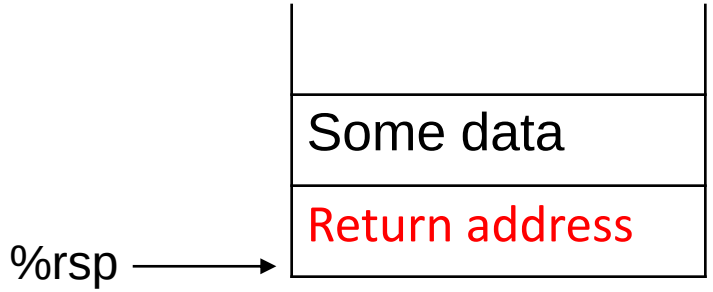
```
1 #
2 # some commands
3 #
4 call    g
5 movq    %rax,%rdx
```

The return address is the address of this instruction (in the code segment)

Stack before "call":



Stack after "call":



Return address =

movq instruction address(5)

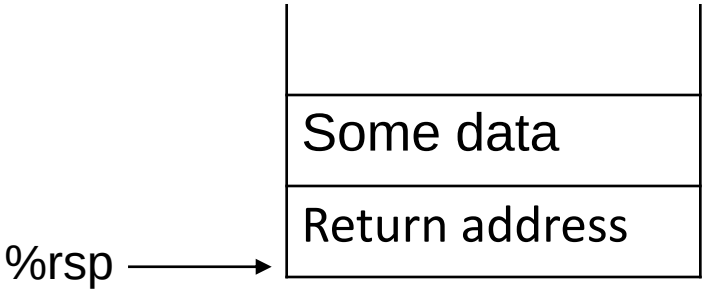
`ret` Instruction

- It pops an address off the stack and jumps to this location.
- The proper use of this instruction is to have prepared the stack so that the stack pointer points to the place where the preceding call instruction stored its return address.

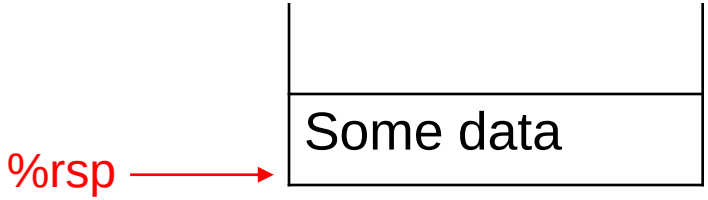
`ret` Instruction – Example

```
1 #
2 # some commands
3 #
4 ret
```

Stack before “ret”:



Stack after “ret”:



- Next instruction to execute is the one stored in the return address

Passing Data

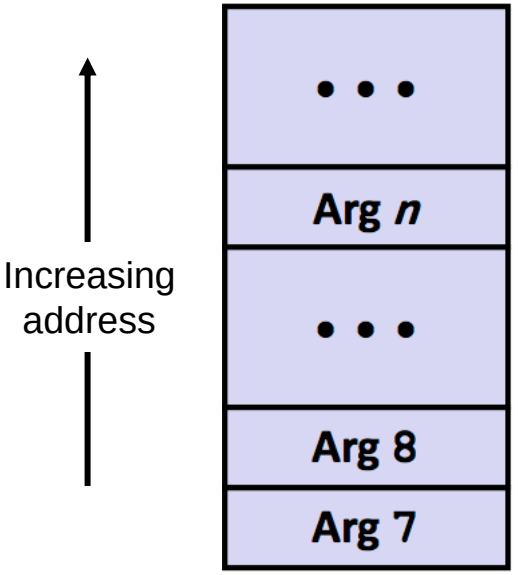
- The first 6 arguments are passed using the registers:

Argument Number	Operand Size (bits)			
	64	32	16	8
1	%rdi	%edi	%di	%dil
2	%rsi	%esi	%si	%sil
3	%rdx	%edx	%dx	%dl
4	%rcx	%ecx	%cx	%cl
5	%r8	%r8d	%r8w	%r8b
6	%r9	%r9d	%r9w	%r9b

- Return value is in %rax

Passing Data (2)

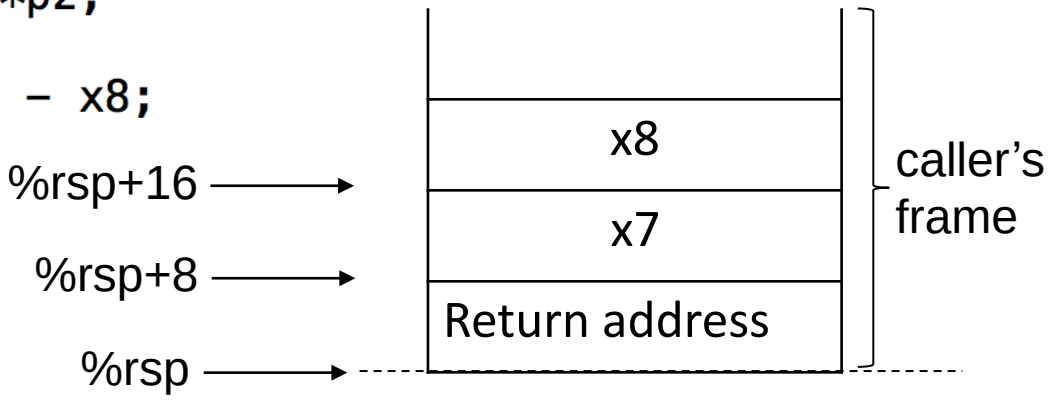
- 7^{th} – n^{th} arguments and so on, are passed on the stack:
  - All data sizes should be rounded up to multiples of 8



- Arguments are pushed in reversed order

Passing Data (3) - Example

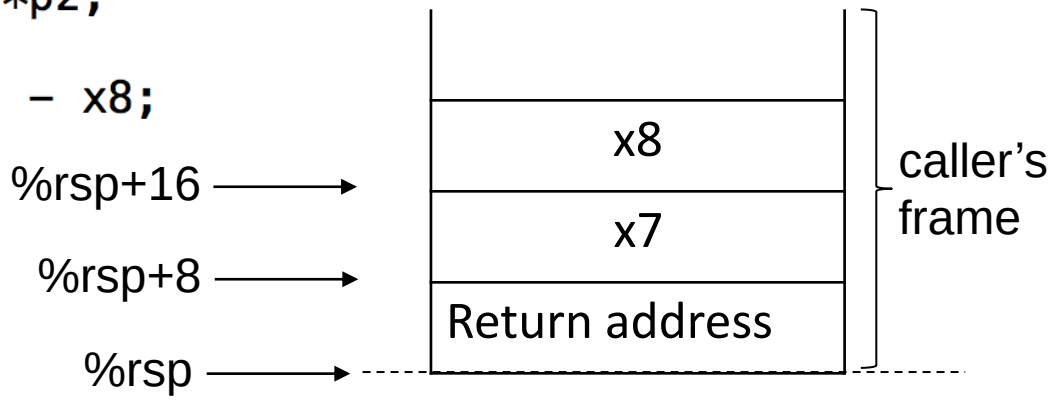
```
1 long f(int* p1, int* p2, int* p3, int* p4,
2        long x5, long x6, long x7, long x8) {
3
4     long result = *p1 + *p2;
5     result += *p3 - *p4;
6     result += x5*x6 + x7 - x8;
7
8     return result;
9 }
```




```
1  f:
2    movl    (%rdi),%eax      # get *p1
3    addl    (%rsi),%eax      # result = *p1+*p2
4    addl    (%rdx),%eax      # result += *p3
5    subl    (%rcx),%eax      # result -= *p4
6    imulq   %r8,%r9          # %r9 = x5*x6
7    addq    %r9,%rax         # result += x5*x6
8    addq    8(%rsp),%rax     # result += x7
9    subq    16(%rsp),%rax    # result -= x8
10   ret
```

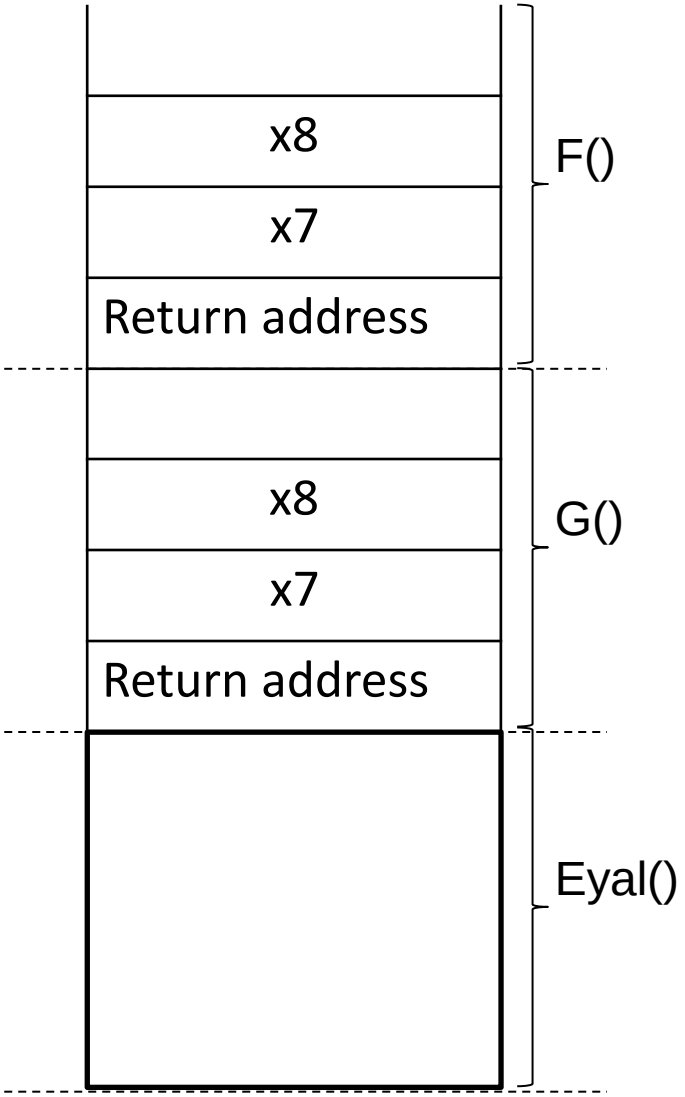
Passing Data (3) - Example

```
1  long f(int* p1, int* p2, int* p3, int* p4,
2         long x5, long x6, long x7, long x8) {
3
4      long result = *p1 + *p2;
5      result += *p3 - *p4;
6      result += x5*x6 + x7 - x8;
7
8      return result;
9  }
```




```
1  f:
2    movl    (%rdi),%eax      # get *p1
3    addl    (%rsi),%eax      # result = *p1+*p2
4    addl    (%rdx),%eax      # result += *p3
5    subl    (%rcx),%eax      # result -= *p4
6    imulq   %r8,%r9          # %r9 = x5*x6
7    addq    %r9,%rax         # result += x5*x6
8    addq    8(%rsp),%rax     # result += x7
9    subq    16(%rsp),%rax    # result -= x8
10   ret
```

Passing Data (4) - Example



Example

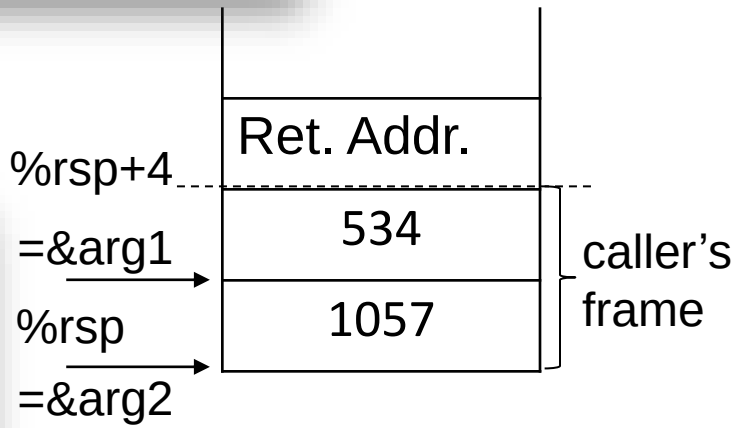
- The C Code:

```
1  int swap_add(int *xp, int *yp)
2  {
3      int x = *xp;
4      int y = *yp;
5
6      *xp = y;
7      *yp = x;
8      return x + y;
9  }
10
11 int caller()
12 {
13     int arg1 = 534;
14     int arg2 = 1057;
15     int sum = swap_add(&arg1,&arg2);
16     int diff = arg1 - arg2;
17
18     return sum * diff;
19 }
```

Example (2)

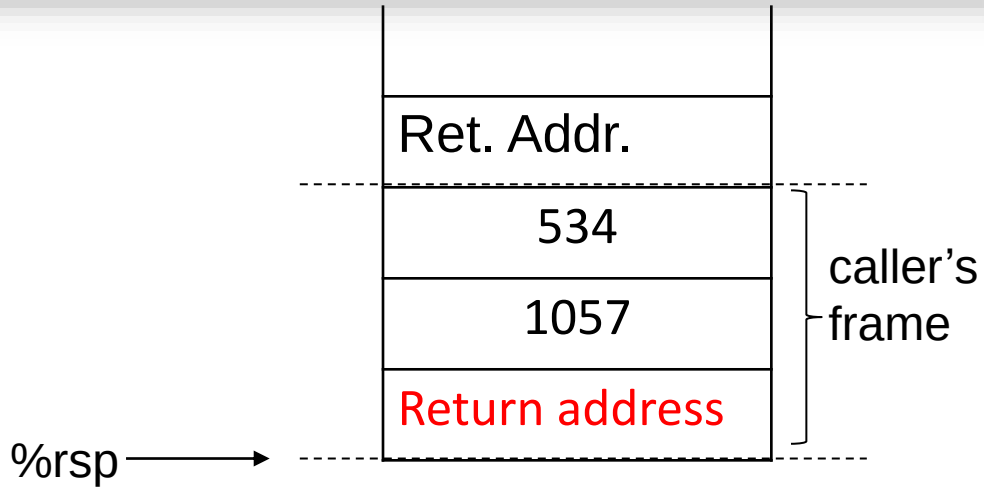
```
1  # Calling code in caller
2  leaq    -8(%rsp),%rsp  # allocate 2 ints on stack
3  movl    $534,4(%rsp)
4  movl    $1057,(%rsp)
5  movq    %rsp,%rsi      # pass &arg2 (2nd argument)
6  leaq    4(%rsp),%rdi   # pass &arg1 (1st argument)
7  call    swap_add
```

```
11 int caller()
12 {
13     int arg1 = 534;
14     int arg2 = 1057;
15     int sum = swap_add(&arg1,&arg2);
16     int diff = arg1 - arg2;
17
18     return sum * diff;
19 }
```



Example (2)

```
1 swap_add:
2   movl    (%rdi),%eax # get x
3   movl    (%rsi),%ecx # get y
4   movl    %ecx,(%rdi) # store y at *xp
5   movl    %eax,(%rsi) # store x at *yp
6   addl    %ecx,%eax   # set return value x+y
7   ret
```



Registers

- The set of program registers acts as a single resource shared by all of the procedures.
- We must make sure that when one procedure (the *caller*) calls another (the *callee*), the callee does not overwrite some register value that the caller planned to use later.
- For this reason, x86-64 adopts a uniform set of conventions for register usage that must be respected by all procedures, including those in program libraries.

Registers conventions

- Registers %rax,%rdi,%rsi,%rdx,%rcx,%r8,%r9,%r10,%r11 are classified as *caller save* registers.
  - When procedure Q is called by P, it can overwrite these registers without destroying any data required by P.
- Registers %rbx,%r12,%r13,%r14,%r15,%rbp are classified as *callee save* registers.
  - This means that Q must save the values of any of these registers on the stack before overwriting them, and restore them before returning, because P (or some higher-level procedure) may need these values for its future computations.

Registers conventions (2)

- Register %rsp (and sometimes %rbp) must be maintained according to the conventions learnt.
  - Will be used only for maintaining the stack and frame.
- Register %rax is used for returning the value of any function that returns an integer or pointer.

Caller vs. Callee

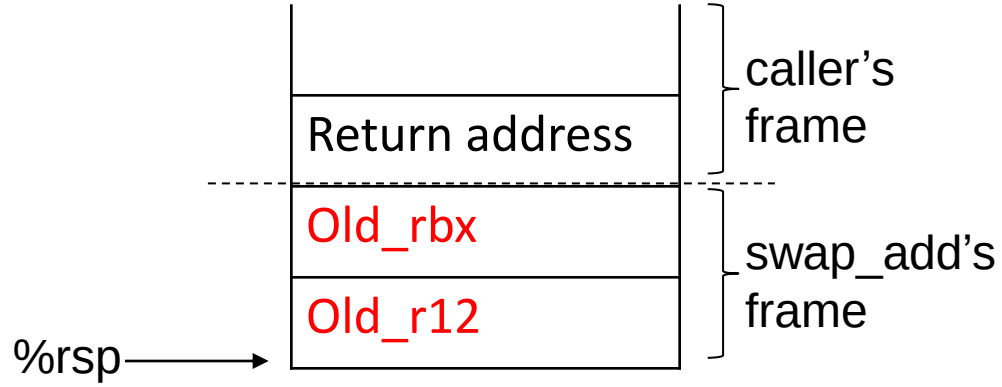
```
1 int P(int x)
2 {
3       int y = x*x;
4       int z = Q(y);
5
6       return y + z;
7 }
```

- Procedure P computes y before calling Q, but it must also ensure that the value of y is available after Q returns. It can do this by one of two means:
  - Store the value of y in its own stack frame before calling Q. When Q returns, it can then retrieve the value of y from the stack.
  - Store the value of y in a callee save register. If Q, or any procedure called by Q, wants to use this register, it must save the register value in its stack frame and restore the value before it returns. Thus, when Q returns to P, the value of y will be in the callee save register, either because the register was never altered or because it was saved and restored.

Example

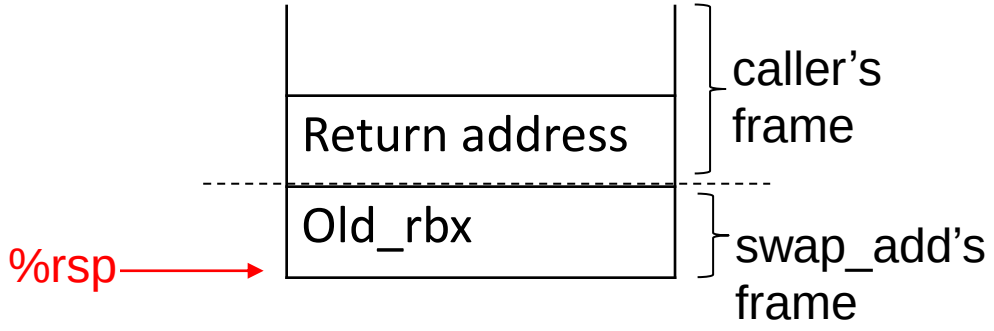
- Let’s take another look at swap_add
- This time, use callee-save registers
- Note that previous version is better
  - No extra memory references (push, pop)

Example (2)



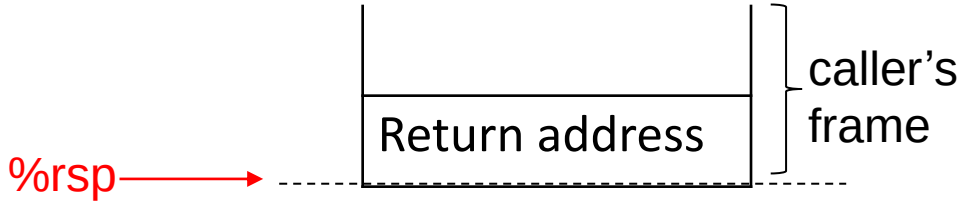

```
1  swap_add:
2    pushq   %rbx            # backup callee-save register
3    pushq   %r12            # backup callee-save register
4    movl    (%rdi),%ebx     # get x
5    movl    (%rsi),%r12d    # get y
6    movl    %r12d,(%rdi)    # store y at *xp
7    movl    %ebx,(%rsi)     # store x at *yp
8    addl    %r12d,%ebx      # compute x+y
9    movl    %ebx,%eax       # set return value
10   popq    %r12            # restore %r12
11   popq    %rbx            # restore %rbx
12   ret
```

Example (3)




```
1  swap_add:
2    pushq   %rbx          # backup callee-save register
3    pushq   %r12          # backup callee-save register
4    movl    (%rdi),%ebx   # get x
5    movl    (%rsi),%r12d  # get y
6    movl    %r12d,(%rdi)  # store y at *xp
7    movl    %ebx,(%rsi)   # store x at *yp
8    addl    %r12d,%ebx    # compute x+y
9    movl    %ebx,%eax     # set return value
10   popq    %r12          # restore %r12
11   popq    %rbx          # restore %rbx
12   ret
```

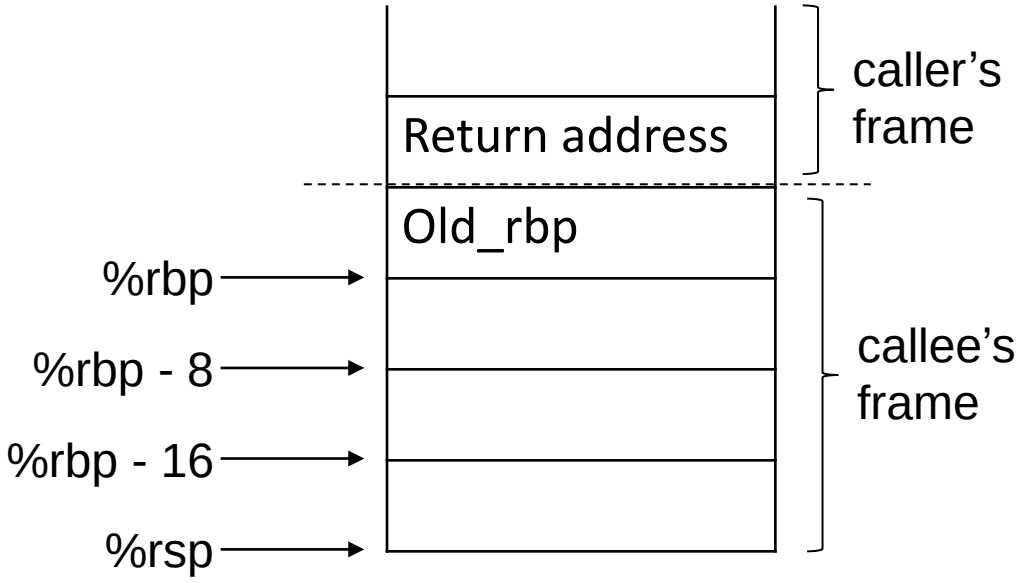
Example (4)




```
1  swap_add:
2    pushq   %rbx          # backup callee-save register
3    pushq   %r12          # backup callee-save register
4    movl    (%rdi),%ebx   # get x
5    movl    (%rsi),%r12d  # get y
6    movl    %r12d,(%rdi)  # store y at *xp
7    movl    %ebx,(%rsi)   # store x at *yp
8    addl    %r12d,%ebx    # compute x+y
9    movl    %ebx,%eax     # set return value
10   popq    %r12          # restore %r12
11   popq    %rbx          # restore %rbx
12   ret
```

Variable-Size Frame

- Assume frame size is not known at compile time
- %rbp is used as a pointer to frame's start

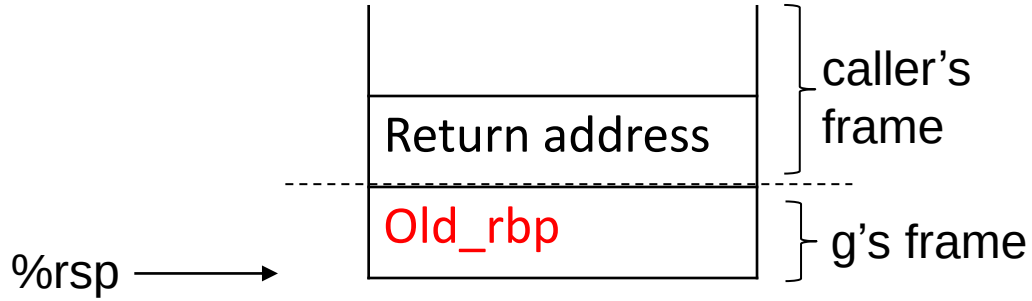


Variable-Size Frame (2)

- Always at function's start:

```
1 g:
2     pushq    %rbp
3     movq     %rsp,%rbp
```

1. Backup %rbp at function start

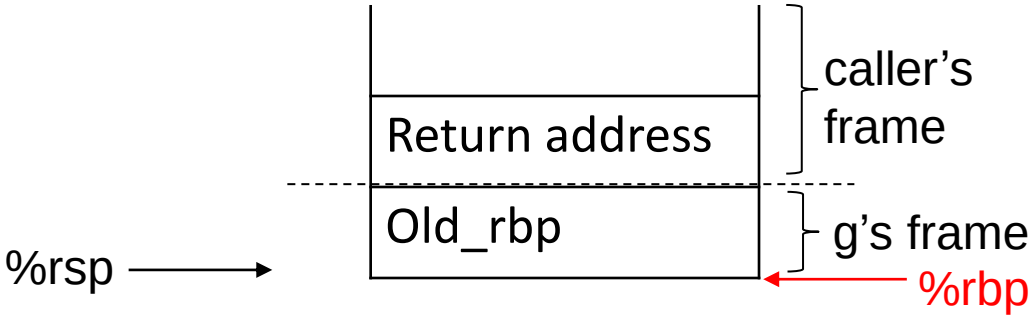


Variable-Size Frame (3)

- Always at function's start:

```
1 g:
2     pushq   %rbp
3     movq    %rsp,%rbp
```

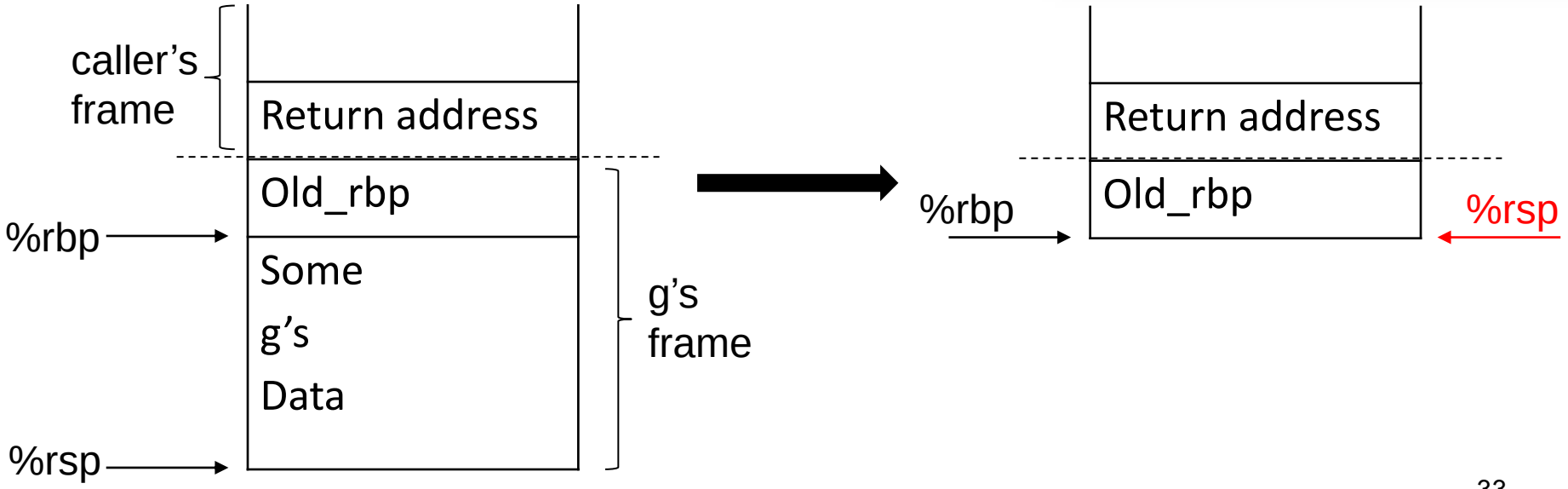
2. Set %rbp with new frame's start address



Variable-Size Frame (4)

- Always at function's end:
  - Prepare stack for returning:
  1. Free stack frame's memory

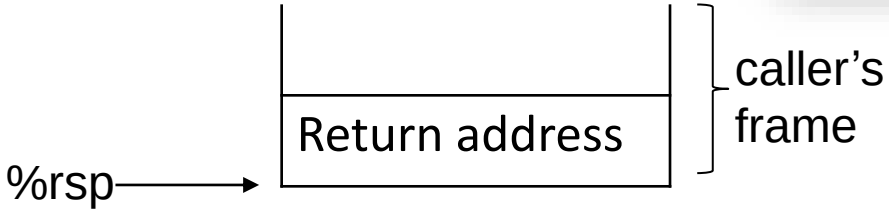
```
1  g:
2    #
3    # some code
4    #
5    movq     %rbp,%rsp
6    popq     %rbp
7    ret
```



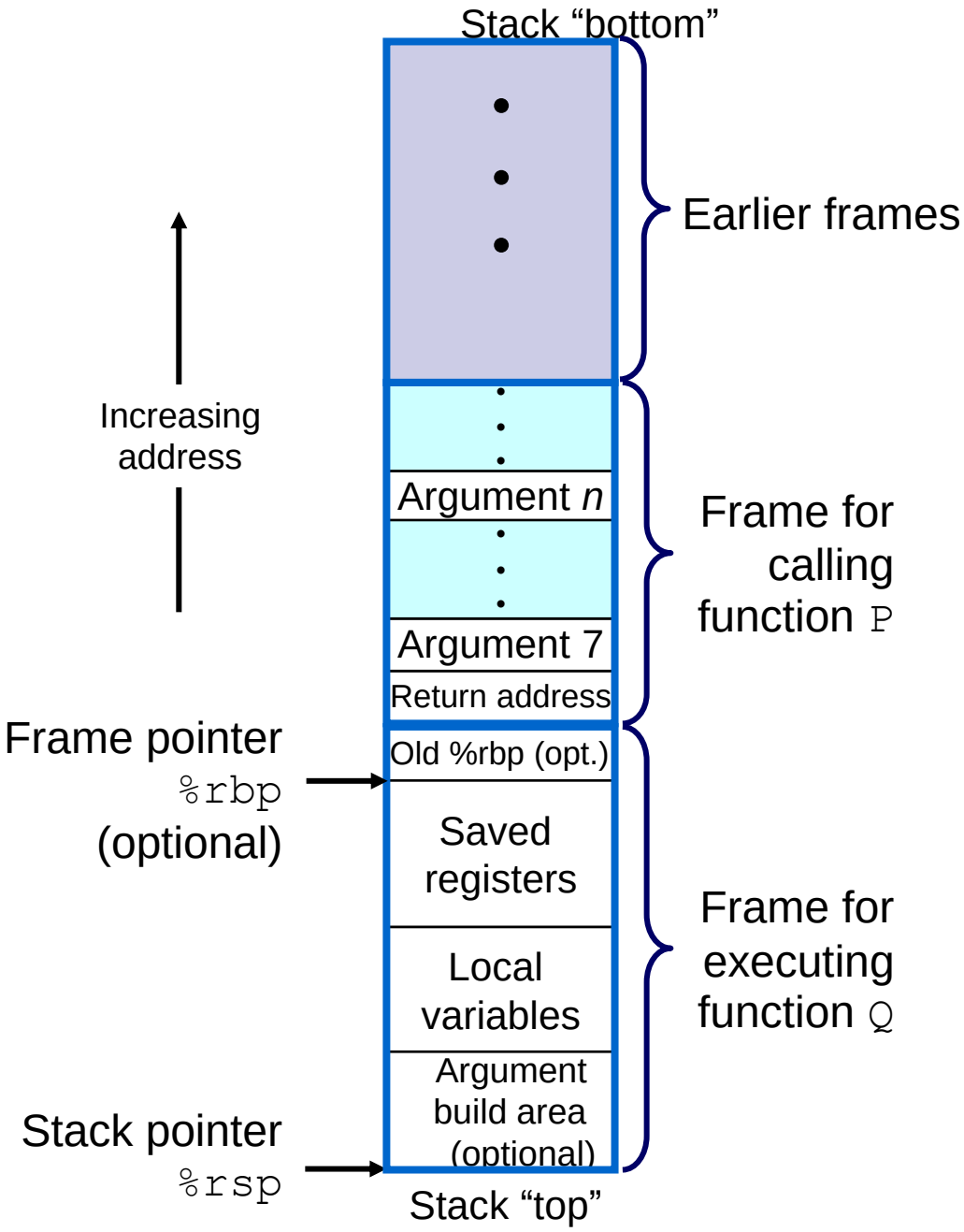
Variable-Size Frame (5)

- Always at function's end:
  - Prepare stack for returning:
  2. Restore %rbp

```
1  g:
2    #
3    # some code
4    #
5    movq    %rbp,%rsp
6    popq    %rbp
7    ret
```



- Could replace lines 5 and 6 with `leave`
  - Instruction is equivalent


Stack “bottom”
Earlier frames
Increasing address
Argument n
Argument 7
Return address
Frame for calling function P
Frame pointer %rbp (optional)
Old %rbp (opt.)
Saved registers
Local variables
Argument build area (optional)
Frame for executing function Q
Stack pointer %rsp
Stack “top”

Another example

- Tirgul5b_func.s

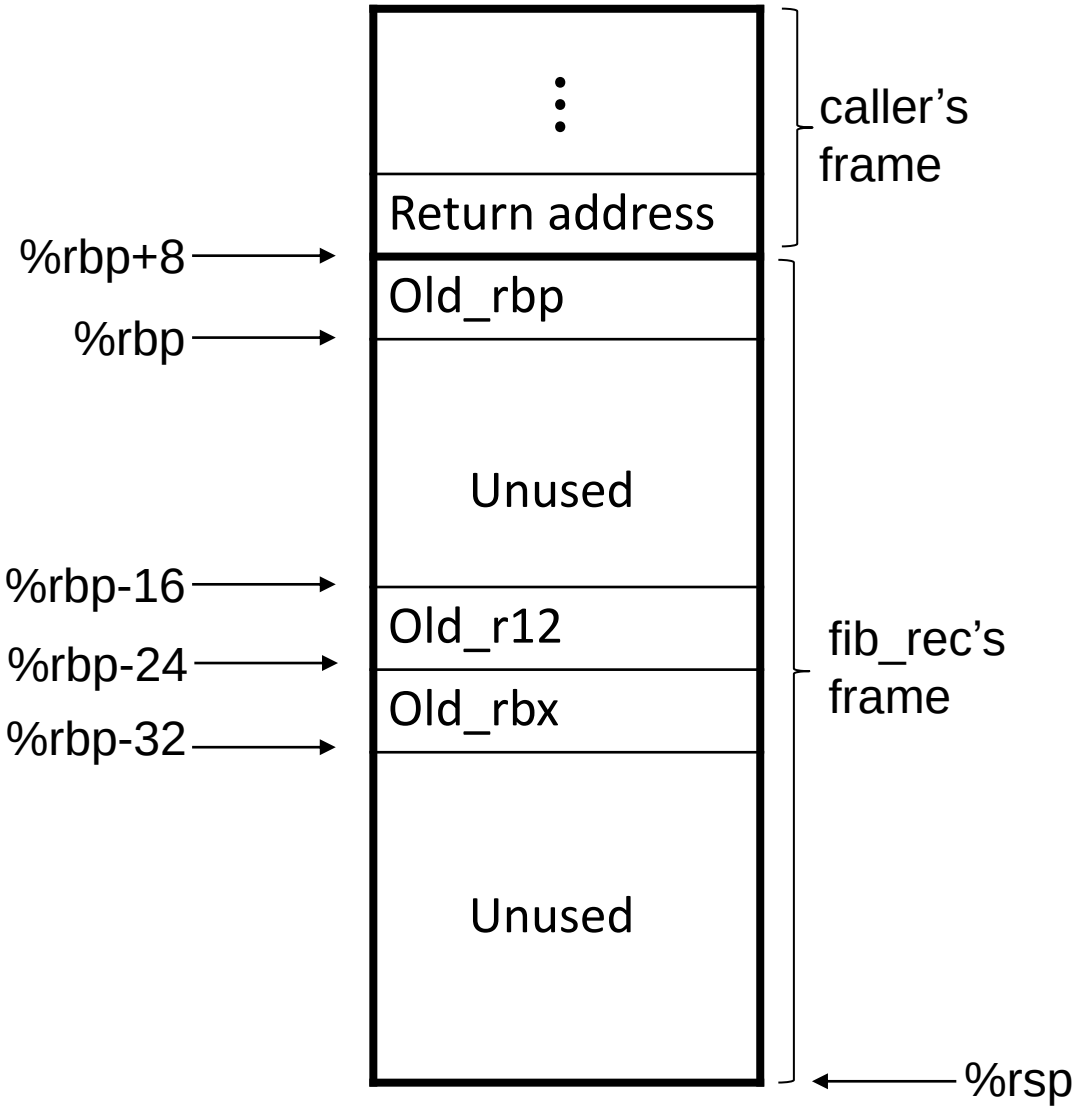
Recursive Procedures

- The stack and linkage conventions described in the previous section allow procedures to call themselves recursively.
- Since each call has its own private space on the stack, the local variables of the multiple outstanding calls do not interfere with one another.
- Furthermore, the stack discipline naturally provides the proper policy for allocating local storage when the procedure is called and deallocating it when it returns.

Fibonacci function:

```
1  fib_rec:
2    # setup code
3    pushq   %rbp              # save old %rbp
4    movq    %rsp,%rbp         # set %rbp as frame pointer
5    subq    $16,%rsp          # allocate 16 bytes on stack
6    pushq   %r12              # save %r12
7    pushq   %rbx              # save %rbx
8
9    # body code
10   movq    %rdi,%rbx         # %rbx = n
11   cmpq    $2,%rdi           # compare n:2
12   jl      .L24
13   addq    $-12,%rsp         # allocate 12 bytes on stack
14   leaq    -2(%rbx),%rdi     # pass n-2 as argument
15   call    fib_rec           # call fib_rec(n-2)
16   movq    %rax,%r12         # store result in %r12
17   addq    $-12,%rsp         # allocate 12 bytes on stack
18   leaq    -1(%rbx),%rdi     # pass n-1 as argument
19   call    fib_rec           # call fib_rec(n-1)
20   addq    %r12,%rax         # compute val + nval
21   jmp     .L25              # go to done
22
23   # terminal condition
24 .L24:
25   movq    $1,%rax           # return value 1
26   # finishing code
27 .L25:
28   leaq    -32(%rbp),%rsp    # set stack to offset -32
29   popq    %rbx              # restore %rbx
30   popq    %r12              # restore %r12
31   movq    %rbp,%rsp         # restore stack pointer
32   popq    %rbp              # restore %rbp
33   ret
```

Stack frame for Recursive Fibonacci:



A Note on IA32 Procedures

- All arguments are passes on stack
- %ebp always points to frame's start
- Don't you like x86-64 better? ☺

Out-of-Bounds Memory References and Buffer Overflow

Buffer Overflow

- C does not perform any bounds checking for array references.
- Local variables are stored on the stack along with state information such as register values and return pointers.
- This combination can lead to serious program errors, where the state stored on the stack gets corrupted by a write to an out-of-bounds array element.
- When the program then tries to reload the register or execute a `ret` instruction with this corrupted state, things can go seriously wrong.

Buffer overflow example

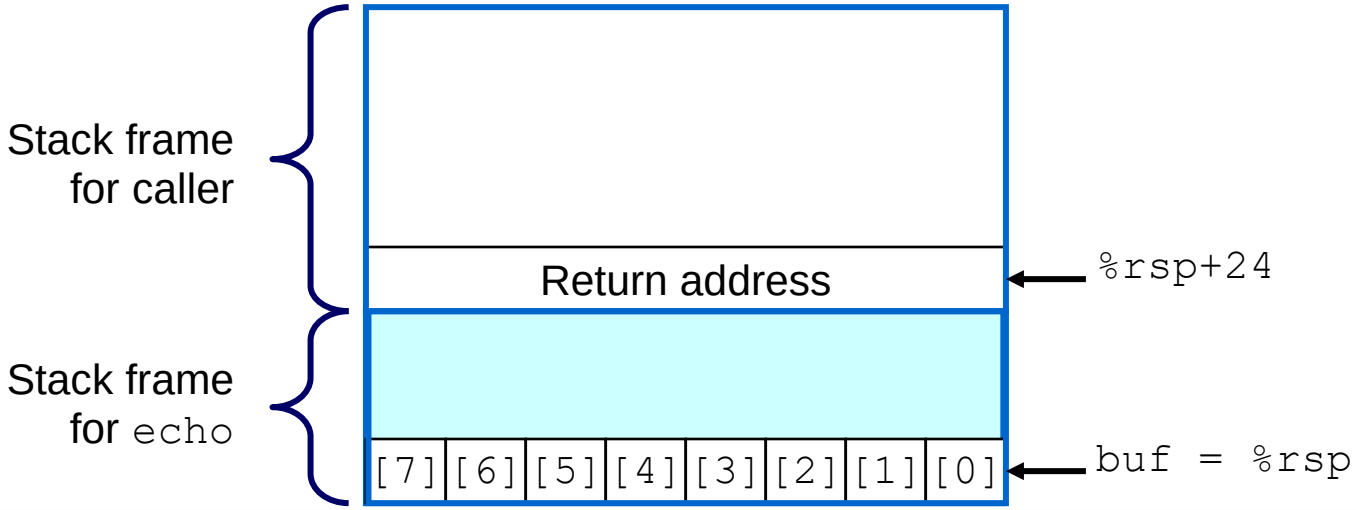
- The function `get`, assumes that `dest` is big enough!

```
1  /* Implementation of library function gets() */
2  char* gets(char* s)
3  {
4      int c;
5      char* dest = s;
6      while((c = getchar()) != '\n' && c != EOF)
7          *dest++ = c;
8      if (c == EOF && dst == s)
9          /* No characters read */
10         return NULL;
11     *dest++ = '\0'; /* Terminate string */
12     return s;
13 }
14
15 /* Read input line and write it back */
16 void echo()
17 {
18     char buf[8]; /* Way too small! */
19     gets(buf);
20     puts(buf);
21 }
```

Buffer overflow example (2)

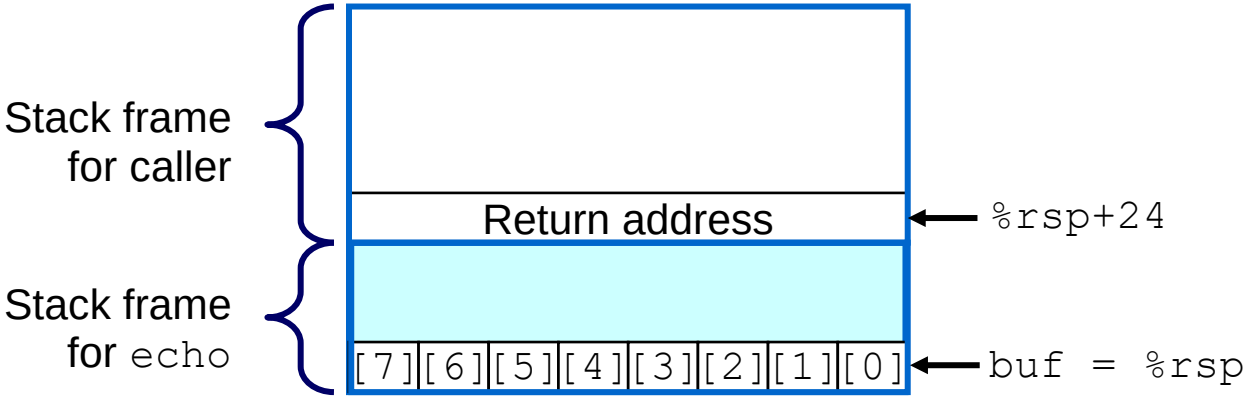
- The problem with gets is that it has no way to determine whether sufficient space has been allocated to hold the entire string.
- In the echo example, we have purposely made the buffer very small - just four characters long.
- Any string longer than three characters will cause an out-of-bounds write.

Buffer overflow example (3)




```
1 echo:
2   subq    $24,%rsp    # allocate 24 bytes on stack
3   movq    %rsp,%rdi   # compute buff as %rsp
4   call    gets        # call gets function
5   movq    %rsp,%rdi   # compute buff as %rsp
6   call    puts        # call puts function
7   addq    $24,%rsp    # deallocate stack space
8   ret
```

Buffer overflow example (4)



- Write to buf[8] through buf[23] – unused stack space is corrupted.
- Write to buf[24] through buf[31] – the return address will be corrupted.
- Write to buf[32]+ – saved state in caller is corrupted
- As this example illustrates, buffer overflow can cause a program to seriously misbehave.

Buffer attack

Illegal!

- The get function was sloppy and cause a buffer overflow by mistake.
- A more pernicious use of buffer overflow is to get a program to perform a function that it would otherwise be unwilling to do.
- This is one of the most common methods to attack the security of a system over a computer network.
- The program is fed with a string that contains the byte encoding of some executable code, called the *exploit code*, plus some extra bytes that overwrite the return pointer with a pointer to the code in the buffer.
- The effect of executing the `ret` instruction is then to jump to the exploit code.

Forms of attacks

- The exploit code uses a system call to start up a shell program, providing the attacker with a range of operating system functions.
- The exploit code performs some otherwise unauthorized task, repairs the damage to the stack, and then executes `ret` a second time, causing an (apparently) normal return to the caller (one of the tools of the “Great Worm” of November, 1988).
- etc.
- “Tirgul5c_switch_return_address.s” - an example how to “damage” the return address and then fixing it.