

Lecture 9: Procedures

David Hovemeyer

September 21, 2026

601.229 Computer Systems Fundamentals



Control flow (part 2)

- ▶ Procedures
- ▶ Stacks:
 - ▶ Procedure calls and returns
 - ▶ Storage for local variables and temporary values
- ▶ Today's example programs are linked as `control2.zip` on the course website

Procedures

Procedures, call stack

- ▶ Procedures (a.k.a. functions, subroutines), the most important abstraction in programming
 - ▶ Can you imagine trying to write programs without them?
- ▶ *Call stack*: hardware-supported, runtime data structure
 - ▶ Stores *return addresses* so procedures know where to return to
 - ▶ Used to allocate *stack frames*: per-procedure-call storage area for local variables, temporary values, and (sometimes) argument values
 - ▶ As name suggests, is a stack, LIFO discipline (push and pop)

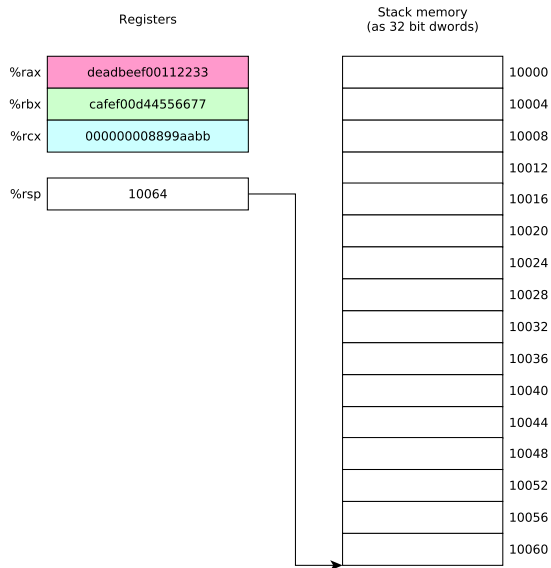
Stack pointer, instruction pointer

- ▶ *Stack pointer* register `%rsp`: contains address of current “top” of stack
 - ▶ Important: stack grows towards lower addresses, so top of stack is at lower address than bottom of stack
- ▶ *Instruction pointer* register `%rip`: contains code address of next instruction to be updated
 - ▶ Control flow changes the value of `%rip`
- ▶ Other architectures use the name “program counter” rather than “instruction pointer”, but they’re the same thing

push and pop

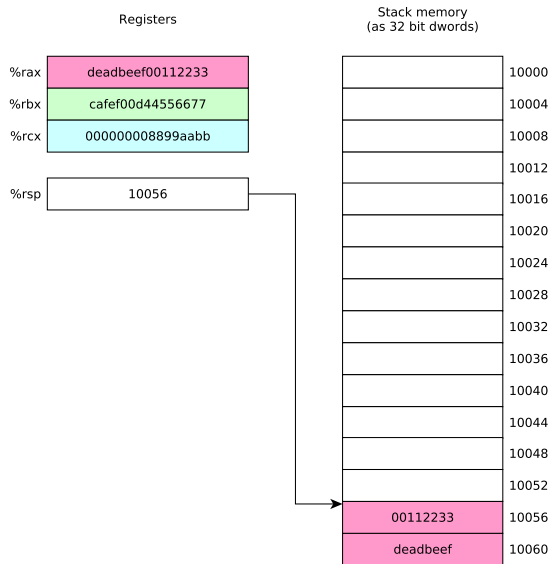
- ▶ `push`: push a data value onto the call stack
 - ▶ E.g., `pushq %rax`
 - ▶ Decrement `%rsp` by 8
 - ▶ Store value in `%rax` at memory location pointed-to by `%rsp`
- ▶ `pop`: pop a data value from the call stack
 - ▶ E.g., `popq %rax`
 - ▶ Load value at memory location pointed-to by `%rsp` into `%rax`
 - ▶ Increment `%rsp` by 8
- ▶ `push` and `pop` are amazingly useful for saving and restoring register values
- ▶ Various size operands (1, 2, 4, 8 bytes) can be pushed and popped; need to consider alignment

push and pop



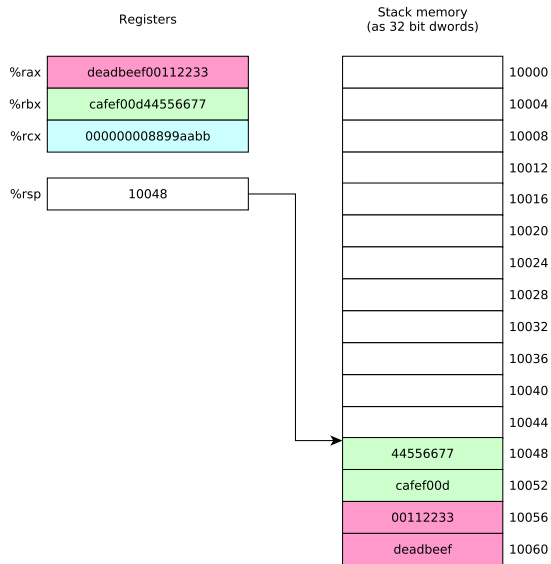
```
pushq %rax  
pushq %rbx  
pushq %rcx  
popq %rbx  
popq %rax  
popq %rcx
```

push and pop



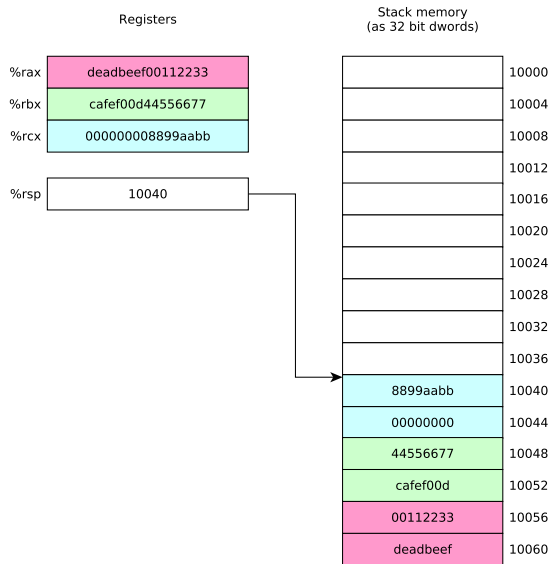
```
pushq %rax
pushq %rbx
pushq %rcx
popq %rbx
popq %rax
popq %rcx
```

push and pop



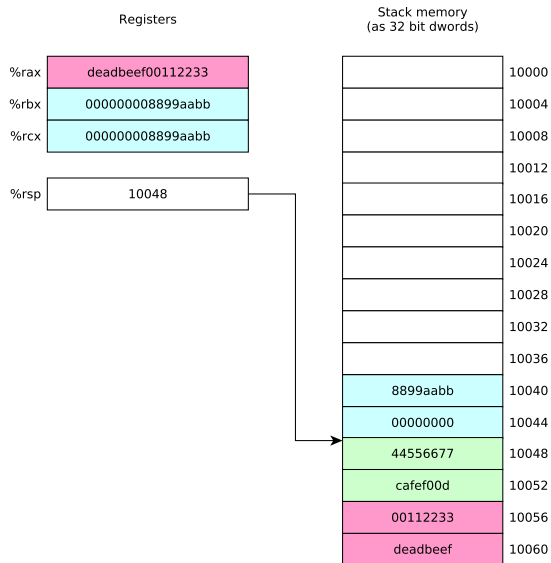
```
pushq %rax
pushq %rbx
pushq %rcx
popq %rbx
popq %rax
popq %rcx
```

push and pop



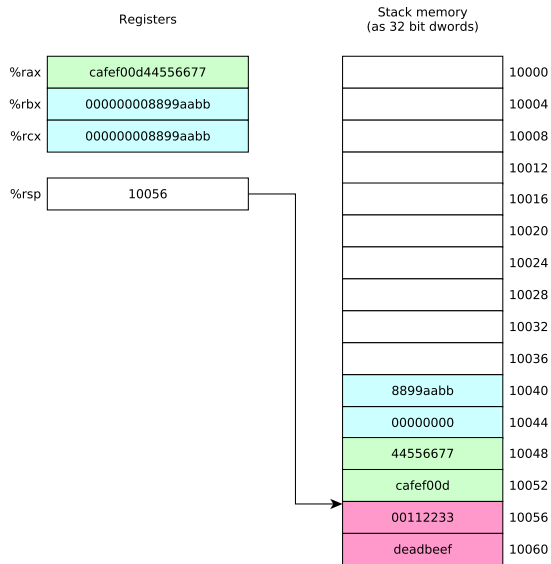
```
pushq %rax  
pushq %rbx  
pushq %rcx  
popq %rbx  
popq %rax  
popq %rcx
```

push and pop



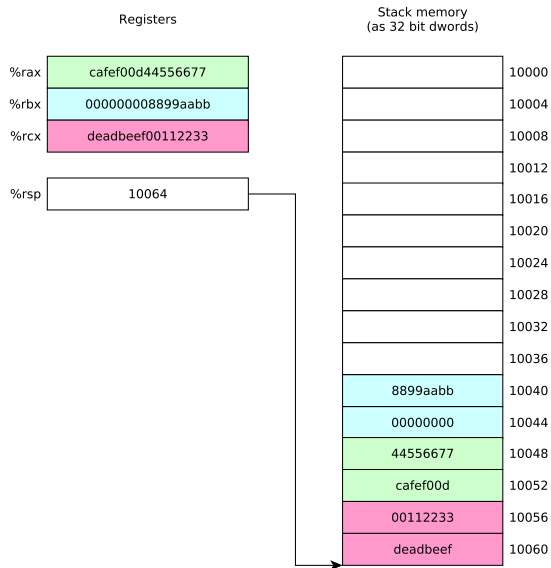
```
pushq %rax
pushq %rbx
pushq %rcx
popq %rbx
popq %rax
popq %rcx
```

push and pop



```
pushq %rax
pushq %rbx
pushq %rcx
popq %rbx
popq %rax
popq %rcx
```

push and pop



```
pushq %rax
pushq %rbx
pushq %rcx
popq %rbx
popq %rax
popq %rcx
```

call and ret

- ▶ `call` instruction: calls procedure
 - ▶ `%rip` contains address of instruction following `call` instruction
 - ▶ Push `%rip` onto stack (as though `pushq %rip` was executed): this is the *return address*
 - ▶ Change `%rip` to address of first instruction of called procedure
 - ▶ Called procedure starts executing
- ▶ `ret` instruction: return from procedure
 - ▶ Pop saved return address from stack into `%rip` (as though `popq %rip` was executed)
 - ▶ Execution continues at return address

Stack alignment

- ▶ Recall that storage for multibyte values should be allocated in memory using *natural* alignment

Stack alignment

- ▶ Recall that storage for multibyte values should be allocated in memory using *natural* alignment
- ▶ E.g., storage for an 8 byte value should be stored at an address which is a multiple of 8

Stack alignment

- ▶ Recall that storage for multibyte values should be allocated in memory using *natural* alignment
 - ▶ E.g., storage for an 8 byte value should be stored at an address which is a multiple of 8
- ▶ This is true of stack-allocated values!

Stack alignment

- ▶ Recall that storage for multibyte values should be allocated in memory using *natural* alignment
 - ▶ E.g., storage for an 8 byte value should be stored at an address which is a multiple of 8
- ▶ This is true of stack-allocated values!
- ▶ The Linux x86-64 calling conventions require `%rsp` to be a multiple of 16 at the point of a procedure call (to ensure that 16 byte values can be accessed on the stack if necessary)

Stack alignment

- ▶ Recall that storage for multibyte values should be allocated in memory using *natural* alignment
 - ▶ E.g., storage for an 8 byte value should be stored at an address which is a multiple of 8
- ▶ This is true of stack-allocated values!
- ▶ The Linux x86-64 calling conventions require `%rsp` to be a multiple of 16 at the point of a procedure call (to ensure that 16 byte values can be accessed on the stack if necessary)
- ▶ **Issue:** on entry to a procedure, $\text{\%rsp} \bmod 16 = 8$ because the `call` instruction (which called the procedure) pushed `%rip` (the program counter) onto the stack

Ensuring correct stack alignment

- ▶ To ensure correct stack alignment:

Ensuring correct stack alignment

- ▶ To ensure correct stack alignment:
 - ▶ On procedure entry: `subq $8, %rsp`

Ensuring correct stack alignment

- ▶ To ensure correct stack alignment:
 - ▶ On procedure entry: `subq $8, %rsp`
 - ▶ Prior to procedure return: `addq $8, %rsp`

Ensuring correct stack alignment

- ▶ To ensure correct stack alignment:
 - ▶ On procedure entry: `subq $8, %rsp`
 - ▶ Prior to procedure return: `addq $8, %rsp`
- ▶ You've seen these in previous code examples, now you know why they're used

Ensuring correct stack alignment

- ▶ To ensure correct stack alignment:
 - ▶ On procedure entry: `subq $8, %rsp`
 - ▶ Prior to procedure return: `addq $8, %rsp`
- ▶ You've seen these in previous code examples, now you know why they're used
- ▶ The Linux `printf` function will segfault if the stack is misaligned

Register use conventions

- ▶ Very important issue:

Register use conventions

- ▶ Very important issue:
 - ▶ There is only one set of registers

Register use conventions

- ▶ Very important issue:
 - ▶ There is only one set of registers
 - ▶ Procedures must share them

Register use conventions

- ▶ Very important issue:
 - ▶ There is only one set of registers
 - ▶ Procedures must share them
 - ▶ *Register use conventions* are rules that all procedures use to avoid conflicts

Register use conventions

- ▶ Very important issue:
 - ▶ There is only one set of registers
 - ▶ Procedures must share them
 - ▶ *Register use conventions* are rules that all procedures use to avoid conflicts
- ▶ Another important issue:

Register use conventions

- ▶ Very important issue:
 - ▶ There is only one set of registers
 - ▶ Procedures must share them
 - ▶ *Register use conventions* are rules that all procedures use to avoid conflicts
- ▶ Another important issue:
 - ▶ How are argument values passed to called procedures?

Register use conventions

- ▶ Very important issue:
 - ▶ There is only one set of registers
 - ▶ Procedures must share them
 - ▶ *Register use conventions* are rules that all procedures use to avoid conflicts
- ▶ Another important issue:
 - ▶ How are argument values passed to called procedures?
 - ▶ Calling conventions typically designate that some argument values are passed in specific registers

Register use conventions

- ▶ Very important issue:
 - ▶ There is only one set of registers
 - ▶ Procedures must share them
 - ▶ *Register use conventions* are rules that all procedures use to avoid conflicts
- ▶ Another important issue:
 - ▶ How are argument values passed to called procedures?
 - ▶ Calling conventions typically designate that some argument values are passed in specific registers
 - ▶ Procedure return value is typically returned in a specific register

Do I really need to follow register use conventions?

- ▶ Register use conventions are *conventions*

Do I really need to follow register use conventions?

- ▶ Register use conventions are *conventions*
- ▶ You might (sometimes) be able to violate them and get away with it

Do I really need to follow register use conventions?

- ▶ Register use conventions are *conventions*
- ▶ You might (sometimes) be able to violate them and get away with it
- ▶ Here's why you should always follow them:

Do I really need to follow register use conventions?

- ▶ Register use conventions are *conventions*
- ▶ You might (sometimes) be able to violate them and get away with it
- ▶ Here's why you should always follow them:
 - ▶ They help you modularize your own code (because they set groundrules to allow procedures to interact with each other safely)

Do I really need to follow register use conventions?

- ▶ Register use conventions are *conventions*
- ▶ You might (sometimes) be able to violate them and get away with it
- ▶ Here's why you should always follow them:
 - ▶ They help you modularize your own code (because they set groundrules to allow procedures to interact with each other safely)
 - ▶ They allow your code to interoperate with other code, including library routines and (OS) system calls

Do I really need to follow register use conventions?

- ▶ Register use conventions are *conventions*
- ▶ You might (sometimes) be able to violate them and get away with it
- ▶ Here's why you should always follow them:
 - ▶ They help you modularize your own code (because they set groundrules to allow procedures to interact with each other safely)
 - ▶ They allow your code to interoperate with other code, including library routines and (OS) system calls
- ▶ **Always follow the appropriate register use conventions**

x86-64 Linux register use conventions

- ▶ Arguments 1–6 passed in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`

x86-64 Linux register use conventions

- ▶ Arguments 1–6 passed in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
- ▶ Argument 7 and beyond, and “large” arguments such as pass-by-value struct data, passed on stack

x86-64 Linux register use conventions

- ▶ Arguments 1–6 passed in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
 - ▶ Argument 7 and beyond, and “large” arguments such as pass-by-value struct data, passed on stack
- ▶ Integer or pointer return value returned in `%rax`

x86-64 Linux register use conventions

- ▶ Arguments 1–6 passed in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
 - ▶ Argument 7 and beyond, and “large” arguments such as pass-by-value struct data, passed on stack
- ▶ Integer or pointer return value returned in `%rax`
- ▶ Caller-saved registers: `%r10`, `%r11` (and also the argument registers)

x86-64 Linux register use conventions

- ▶ Arguments 1–6 passed in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
 - ▶ Argument 7 and beyond, and “large” arguments such as pass-by-value struct data, passed on stack
- ▶ Integer or pointer return value returned in `%rax`
- ▶ Caller-saved registers: `%r10`, `%r11` (and also the argument registers)
- ▶ Callee-saved registers: `%rbx`, `%rbp`, `%r12`, `%r13`, `%r14`, `%r15`

Caller-saved vs. callee-saved

- ▶ What happens to register contents when a procedure is called?

Caller-saved vs. callee-saved

- ▶ What happens to register contents when a procedure is called?
- ▶ *Callee-saved* registers: caller may assume that the procedure call will preserve their value

Caller-saved vs. callee-saved

- ▶ What happens to register contents when a procedure is called?
- ▶ *Callee-saved* registers: caller may assume that the procedure call will preserve their value
 - ▶ In general, all procedures must save their values to memory before modifying them, and restore them before returning

Caller-saved vs. callee-saved

- ▶ What happens to register contents when a procedure is called?
- ▶ *Callee-saved* registers: caller may assume that the procedure call will preserve their value
 - ▶ In general, all procedures must save their values to memory before modifying them, and restore them before returning
- ▶ *Caller-saved* registers: caller must *not* assume that the procedure call will preserve their value

Caller-saved vs. callee-saved

- ▶ What happens to register contents when a procedure is called?
- ▶ *Callee-saved* registers: caller may assume that the procedure call will preserve their value
 - ▶ In general, all procedures must save their values to memory before modifying them, and restore them before returning
- ▶ *Caller-saved* registers: caller must *not* assume that the procedure call will preserve their value
 - ▶ In general any procedure can freely modify them

Caller-saved vs. callee-saved

- ▶ What happens to register contents when a procedure is called?
- ▶ *Callee-saved* registers: caller may assume that the procedure call will preserve their value
 - ▶ In general, all procedures must save their values to memory before modifying them, and restore them before returning
- ▶ *Caller-saved* registers: caller must *not* assume that the procedure call will preserve their value
 - ▶ In general any procedure can freely modify them
 - ▶ A caller might need to save their contents to memory prior to calling a procedure and restore the value afterwards

Using registers

- ▶ Using registers correctly and effectively is one of the main challenges of assembly language programming

Using registers

- ▶ Using registers correctly and effectively is one of the main challenges of assembly language programming
- ▶ Some advice:

Using registers

- ▶ Using registers correctly and effectively is one of the main challenges of assembly language programming
- ▶ Some advice:
 - ▶ Use caller-saved registers (`%r10`, `%r11`, etc.) for very short-term temporary values or computations

Using registers

- ▶ Using registers correctly and effectively is one of the main challenges of assembly language programming
- ▶ Some advice:
 - ▶ Use caller-saved registers (%r10, %r11, etc.) for very short-term temporary values or computations
 - ▶ You can use the argument registers as (caller-saved) temporary registers

Using registers

- ▶ Using registers correctly and effectively is one of the main challenges of assembly language programming
- ▶ Some advice:
 - ▶ Use caller-saved registers (%r10, %r11, etc.) for very short-term temporary values or computations
 - ▶ You can use the argument registers as (caller-saved) temporary registers
 - ▶ Understand that called procedures could modify them!

Using registers

- ▶ Using registers correctly and effectively is one of the main challenges of assembly language programming
- ▶ Some advice:
 - ▶ Use caller-saved registers (`%r10`, `%r11`, etc.) for very short-term temporary values or computations
 - ▶ You can use the argument registers as (caller-saved) temporary registers
 - ▶ Understand that called procedures could modify them!
 - ▶ Use callee-saved registers for longer term values that need to persist across procedure calls

Using registers

- ▶ Using registers correctly and effectively is one of the main challenges of assembly language programming
- ▶ Some advice:
 - ▶ Use caller-saved registers (`%r10`, `%r11`, etc.) for very short-term temporary values or computations
 - ▶ You can use the argument registers as (caller-saved) temporary registers
 - ▶ Understand that called procedures could modify them!
 - ▶ Use callee-saved registers for longer term values that need to persist across procedure calls
 - ▶ Use `pushq/popq` to save and restore their values on procedure entry and exit

Recursive Fibonacci computation

Compute n th Fibonacci number recursively (warning: exponential-time algorithm!)

The call stack inherently allows recursion: there is nothing special we need to do to make it work

Recall that

$$\text{fib}(0) = 0$$

$$\text{fib}(1) = 1$$

$$\text{For } n > 1, \text{fib}(n) = \text{fib}(n - 2) + \text{fib}(n - 1)$$

Recursive Fibonacci function (see fibRec.S for full program)

```
fib:
    cmpl $2, %edi          /* check base case */
    jae .LrecursiveCase   /* if n>=2, do recursive case */
    movl %edi, %eax        /* base case, just return n */
    ret

.LrecursiveCase:
    /* recursive case */
    pushq %r12             /* preserve value of %r12 */
    movl %edi, %r12d        /* save n in %r12 */
    subl $2, %edi           /* compute n-2 */
    call fib               /* compute fib(n-2) */
    movl %r12d, %edi        /* put saved n in %edi */
    subl $1, %edi           /* compute n-1 */
    movl %eax, %r12d        /* save fib(n-2) in %r12 */
    call fib               /* compute fib(n-1) */
    addl %r12d, %eax        /* return fib(n-2)+fib(n-1) */
    popq %r12              /* restore value of %r12 */
    ret                    /* done */
```

Running the program (with $N=9$)

```
$ gcc -c -g -no-pie -o fibRec.o fibRec.S
$ gcc -no-pie -o fibRec fibRec.o
$ ./fibRec
fib(9) = 34
```

Clicker quiz!

Clicker quiz omitted from public slides

Stack memory allocation

Allocating space for local variables

- ▶ Sometimes, registers aren't sufficient to store the data used in a procedure

Allocating space for local variables

- ▶ Sometimes, registers aren't sufficient to store the data used in a procedure
- ▶ So, storage for variables must be allocated in memory

Allocating space for local variables

- ▶ Sometimes, registers aren't sufficient to store the data used in a procedure
- ▶ So, storage for variables must be allocated in memory
- ▶ Could use global variables (in `.data` or `.bss` segments)

Allocating space for local variables

- ▶ Sometimes, registers aren't sufficient to store the data used in a procedure
- ▶ So, storage for variables must be allocated in memory
- ▶ Could use global variables (in `.data` or `.bss` segments)
 - ▶ Can make program behavior difficult to understand

Allocating space for local variables

- ▶ Sometimes, registers aren't sufficient to store the data used in a procedure
- ▶ So, storage for variables must be allocated in memory
- ▶ Could use global variables (in `.data` or `.bss` segments)
 - ▶ Can make program behavior difficult to understand
 - ▶ Not useful for recursive or reentrant functions

Allocating space for local variables

- ▶ Sometimes, registers aren't sufficient to store the data used in a procedure
- ▶ So, storage for variables must be allocated in memory
- ▶ Could use global variables (in `.data` or `.bss` segments)
 - ▶ Can make program behavior difficult to understand
 - ▶ Not useful for recursive or reentrant functions
 - ▶ In general, wasteful of memory

Allocating space for local variables

- ▶ Sometimes, registers aren't sufficient to store the data used in a procedure
- ▶ So, storage for variables must be allocated in memory
- ▶ Could use global variables (in `.data` or `.bss` segments)
 - ▶ Can make program behavior difficult to understand
 - ▶ Not useful for recursive or reentrant functions
 - ▶ In general, wasteful of memory
- ▶ Could use heap allocation (i.e., `malloc`, `free`)

Allocating space for local variables

- ▶ Sometimes, registers aren't sufficient to store the data used in a procedure
- ▶ So, storage for variables must be allocated in memory
- ▶ Could use global variables (in `.data` or `.bss` segments)
 - ▶ Can make program behavior difficult to understand
 - ▶ Not useful for recursive or reentrant functions
 - ▶ In general, wasteful of memory
- ▶ Could use heap allocation (i.e., `malloc`, `free`)
 - ▶ Has overhead due to bookkeeping, locking

Allocating space for local variables

- ▶ Sometimes, registers aren't sufficient to store the data used in a procedure
- ▶ So, storage for variables must be allocated in memory
- ▶ Could use global variables (in `.data` or `.bss` segments)
 - ▶ Can make program behavior difficult to understand
 - ▶ Not useful for recursive or reentrant functions
 - ▶ In general, wasteful of memory
- ▶ Could use heap allocation (i.e., `malloc`, `free`)
 - ▶ Has overhead due to bookkeeping, locking
- ▶ The call stack is an ideal place to allocate storage for local variables

Stack allocation

- ▶ Stack allocation of storage is simple:
 - ▶ To allocate n bytes, subtract n from `%rsp`
 - ▶ Updated `%rsp` is a pointer to the beginning of the allocated memory
 - ▶ To deallocate n bytes, add n to `%rsp`
- ▶ Complication: instructions such as `push` and `pop` change `%rsp`
- ▶ Solution: use the *frame pointer* register `%rbp` to keep track of allocated memory area

Using the frame pointer

On entry to procedure:

```
pushq %rbp
movq %rsp, %rbp
subq $N, %rsp
```

Before returning from procedure:

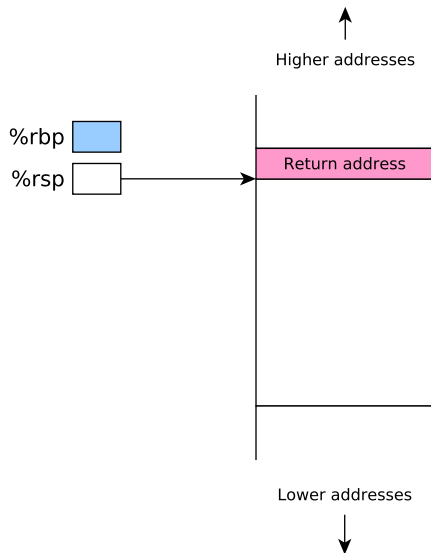
```
addq $N, %rsp
popq %rbp
```

`%rbp` points to a memory location *just above* a block of N bytes allocated in the current stack frame. Note that

- ▶ N should be a multiple of 16 to ensure correct stack alignment
- ▶ The function will access memory locations in the allocated block using *negative* offsets from `%rbp`

Before allocating space in stack frame

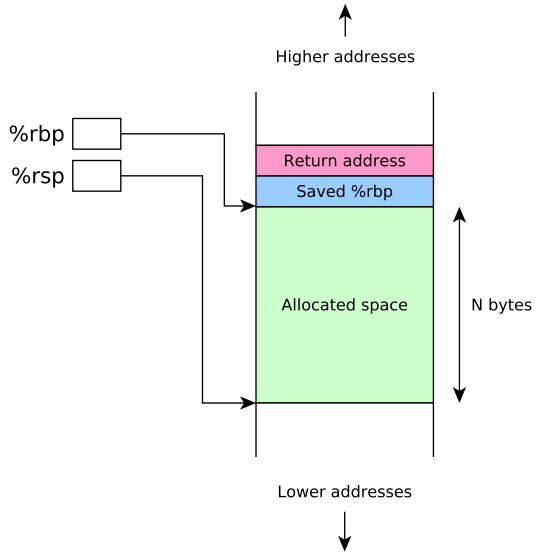
```
--> pushq %rbp  
    movq %rsp, %rbp  
    subq $N, %rsp
```



After allocating space in stack frame

```
pushq %rbp  
movq %rsp, %rbp  
subq $N, %rsp
```

-->



Putting it all together

- ▶ Let's examine a simple program which
 - ▶ Reads two 64 bit integer values from user
 - ▶ Computes their sum using a function
 - ▶ Prints out the sum
- ▶ Calling `scanf` to read input requires variables in which to store input values: we'll allocate them on the stack

addLongs, C version

```
#include <stdio.h>

long addLongs(long a, long b);

int main(void) {
    long x, y, sum;
    printf("Enter two integers: ");
    scanf("%ld %ld", &x, &y);
    sum = addLongs(x, y);
    printf("Sum is %ld\n", sum);
}

long addLongs(long a, long b) {
    return a + b;
}
```

addLongs, assembly version

```
.section .rodata
sPromptMsg: .string "Enter two integers: "
sInputFmt: .string "%ld %ld"
sResultMsg: .string "Sum is %ld\n"
```

```
.section .text
.globl main
.align 16
```

main:

```
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
```

```
movl $0, %eax
movq $sPromptMsg, %rdi
call printf
```

```
movl $0, %eax
movq $sInputFmt, %rdi
leaq -16(%rbp), %rsi
leaq -8(%rbp), %rdx
call scanf
```

```
movq -16(%rbp), %rdi
movq -8(%rbp), %rsi
call addLongs
```

```
movq $sResultMsg, %rdi
movq %rax, %rsi
call printf
```

```
addq $16, %rsp
popq %rbp
ret
```

```
.align 16
addLongs:
movq %rdi, %rax
addq %rsi, %rax
ret
```

addLongs, assembly version

```
.section .rodata
sPromptMsg: .string "Enter two integers: "
sInputFmt: .string "%ld %ld"
sResultMsg: .string "Sum is %ld\n"

.section .text
.globl main
.align 16

main:
    pushq %rbp          <-- save orig value of %rbp
    movq %rsp, %rbp
    subq $16, %rsp

    movl $0, %eax
    movq $sPromptMsg, %rdi
    call printf

    movl $0, %eax
    movq $sInputFmt, %rdi
    leaq -16(%rbp), %rsi
    leaq -8(%rbp), %rdx
    call scanf

    movq -16(%rbp), %rdi
    movq -8(%rbp), %rsi
    call addLongs

    movq $sResultMsg, %rdi
    movq %rax, %rsi
    call printf

    addq $16, %rsp
    popq %rbp
    ret

    .align 16
addLongs:
    movq %rdi, %rax
    addq %rsi, %rax
    ret
```

addLongs, assembly version

```
.section .rodata
sPromptMsg: .string "Enter two integers: "
sInputFmt:  .string "%ld %ld"
sResultMsg: .string "Sum is %ld\n"

.section .text
.globl main
.align 16

main:
    pushq %rbp
    movq %rsp, %rbp    <-- %rbp points to top
                        of alloc'ed area
    subq $16, %rsp

    movl $0, %eax
    movq $sPromptMsg, %rdi
    call printf

    movl $0, %eax
    movq $sInputFmt, %rdi
    leaq -16(%rbp), %rsi
    leaq -8(%rbp), %rdx
    call scanf
```

```
    movq -16(%rbp), %rdi
    movq -8(%rbp), %rsi
    call addLongs

    movq $sResultMsg, %rdi
    movq %rax, %rsi
    call printf

    addq $16, %rsp
    popq %rbp
    ret

    .align 16
addLongs:
    movq %rdi, %rax
    addq %rsi, %rax
    ret
```

addLongs, assembly version

```
.section .rodata
sPromptMsg: .string "Enter two integers: "
sInputFmt: .string "%ld %ld"
sResultMsg: .string "Sum is %ld\n"

.section .text
.globl main
.align 16

main:
    pushq %rbp
    movq %rsp, %rbp
    subq $16, %rsp    <-- allocate 16 byte area

    movl $0, %eax
    movq $sPromptMsg, %rdi
    call printf

    movl $0, %eax
    movq $sInputFmt, %rdi
    leaq -16(%rbp), %rsi
    leaq -8(%rbp), %rdx
    call scanf
```

```
    movq -16(%rbp), %rdi
    movq -8(%rbp), %rsi
    call addLongs

    movq $sResultMsg, %rdi
    movq %rax, %rsi
    call printf

    addq $16, %rsp
    popq %rbp
    ret

    .align 16
addLongs:
    movq %rdi, %rax
    addq %rsi, %rax
    ret
```

addLongs, assembly version

```
.section .rodata
sPromptMsg: .string "Enter two integers: "
sInputFmt:  .string "%ld %ld"
sResultMsg: .string "Sum is %ld\n"

.section .text
.globl main
.align 16

main:
    pushq %rbp
    movq %rsp, %rbp
    subq $16, %rsp

    movl $0, %eax
    movq $sPromptMsg, %rdi
    call printf

    movl $0, %eax
    movq $sInputFmt, %rdi
    leaq -16(%rbp), %rsi <-- pass address of 1st var
    leaq -8(%rbp), %rdx
    call scanf

    movq -16(%rbp), %rdi
    movq -8(%rbp), %rsi
    call addLongs

    movq $sResultMsg, %rdi
    movq %rax, %rsi
    call printf

    addq $16, %rsp
    popq %rbp
    ret

    .align 16
addLongs:
    movq %rdi, %rax
    addq %rsi, %rax
    ret
```

addLongs, assembly version

```
.section .rodata
sPromptMsg: .string "Enter two integers: "
sInputFmt: .string "%ld %ld"
sResultMsg: .string "Sum is %ld\n"

.section .text
.globl main
.align 16

main:
    pushq %rbp
    movq %rsp, %rbp
    subq $16, %rsp

    movl $0, %eax
    movq $sPromptMsg, %rdi
    call printf

    movl $0, %eax
    movq $sInputFmt, %rdi
    leaq -16(%rbp), %rsi
    leaq -8(%rbp), %rdx    <-- pass address of 2nd var
    call scanf

    movq -16(%rbp), %rdi
    movq -8(%rbp), %rsi
    call addLongs

    movq $sResultMsg, %rdi
    movq %rax, %rsi
    call printf

    addq $16, %rsp
    popq %rbp
    ret

    .align 16
addLongs:
    movq %rdi, %rax
    addq %rsi, %rax
    ret
```

addLongs, assembly version

```
.section .rodata
sPromptMsg: .string "Enter two integers: "
sInputFmt: .string "%ld %ld"
sResultMsg: .string "Sum is %ld\n"
```

```
.section .text
.globl main
.align 16
```

main:

```
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
```

```
movl $0, %eax
movq $sPromptMsg, %rdi
call printf
```

```
movl $0, %eax
movq $sInputFmt, %rdi
leaq -16(%rbp), %rsi
leaq -8(%rbp), %rdx
call scanf
```

```
movq -16(%rbp), %rdi <-- pass value of 1st var
movq -8(%rbp), %rsi
call addLongs
```

```
movq $sResultMsg, %rdi
movq %rax, %rsi
call printf
```

```
addq $16, %rsp
popq %rbp
ret
```

```
.align 16
addLongs:
movq %rdi, %rax
addq %rsi, %rax
ret
```

addLongs, assembly version

```
.section .rodata
sPromptMsg: .string "Enter two integers: "
sInputFmt: .string "%ld %ld"
sResultMsg: .string "Sum is %ld\n"
```

```
.section .text
.globl main
.align 16
```

main:

```
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
```

```
movl $0, %eax
movq $sPromptMsg, %rdi
call printf
```

```
movl $0, %eax
movq $sInputFmt, %rdi
leaq -16(%rbp), %rsi
leaq -8(%rbp), %rdx
call scanf
```

```
movq -16(%rbp), %rdi
movq -8(%rbp), %rsi  <-- pass value of 2nd var
call addLongs
```

```
movq $sResultMsg, %rdi
movq %rax, %rsi
call printf
```

```
addq $16, %rsp
popq %rbp
ret
```

```
.align 16
addLongs:
movq %rdi, %rax
addq %rsi, %rax
ret
```

addLongs, assembly version

```
.section .rodata
sPromptMsg: .string "Enter two integers: "
sInputFmt: .string "%ld %ld"
sResultMsg: .string "Sum is %ld\n"
```

```
.section .text
.globl main
.align 16
```

main:

```
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
```

```
movl $0, %eax
movq $sPromptMsg, %rdi
call printf
```

```
movl $0, %eax
movq $sInputFmt, %rdi
leaq -16(%rbp), %rsi
leaq -8(%rbp), %rdx
call scanf
```

```
movq -16(%rbp), %rdi
movq -8(%rbp), %rsi
call addLongs
```

```
movq $sResultMsg, %rdi
movq %rax, %rsi
call printf
```

```
addq $16, %rsp <-- deallocate alloc'ed area
popq %rbp
ret
```

```
.align 16
addLongs:
movq %rdi, %rax
addq %rsi, %rax
ret
```

addLongs, assembly version

```
.section .rodata
sPromptMsg: .string "Enter two integers: "
sInputFmt: .string "%ld %ld"
sResultMsg: .string "Sum is %ld\n"
```

```
.section .text
.globl main
.align 16
```

main:

```
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
```

```
movl $0, %eax
movq $sPromptMsg, %rdi
call printf
```

```
movl $0, %eax
movq $sInputFmt, %rdi
leaq -16(%rbp), %rsi
leaq -8(%rbp), %rdx
call scanf
```

```
movq -16(%rbp), %rdi
movq -8(%rbp), %rsi
call addLongs
```

```
movq $sResultMsg, %rdi
movq %rax, %rsi
call printf
```

```
addq $16, %rsp
popq %rbp      <-- restore orig value of %rbp
ret
```

```
.align 16
addLongs:
movq %rdi, %rax
addq %rsi, %rax
ret
```

addLongs, assembly version

```
.section .rodata
sPromptMsg: .string "Enter two integers: "
sInputFmt: .string "%ld %ld"
sResultMsg: .string "Sum is %ld\n"
```

```
.section .text
.globl main
.align 16
```

main:

```
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
```

```
movl $0, %eax
movq $sPromptMsg, %rdi
call printf
```

```
movl $0, %eax
movq $sInputFmt, %rdi
leaq -16(%rbp), %rsi
leaq -8(%rbp), %rdx
call scanf
```

```
movq -16(%rbp), %rdi
movq -8(%rbp), %rsi
call addLongs
```

```
movq $sResultMsg, %rdi
movq %rax, %rsi
call printf
```

```
addq $16, %rsp
popq %rbp
ret
```

```
.align 16
```

addLongs: <-- does not use stack, ignore alignment :-P

```
movq %rdi, %rax
addq %rsi, %rax
ret
```

Running the program

```
$ gcc -c -no-pie -o addLongs.o addLongs.S
$ gcc -no-pie -o addLongs addLongs.o
$ ./addLongs
Enter two integers: 2 3
Sum is 5
```

Running the program in gdb

```
$ gdb addLongs
...output omitted...
(gdb) break addLongs.S:28
Breakpoint 1 at 0x401172: file addLongs.S, line 28.
(gdb) run
Starting program: /home/daveho/.../src/control2/addLongs
Enter two integers: 3 4

Breakpoint 1, main () at addLongs.S:28
28          movq -16(%rbp), %rdi          /* pass first value */
(gdb) print *(long *)($rbp-16)
$1 = 3
(gdb) print *(long *)($rbp-8)
$2 = 4
```

Running the program in gdb

```
$ gdb addLongs
...output omitted...
(gdb) break addLongs.S:28  <-- set breakpoint just after scanf returns
Breakpoint 1 at 0x401172: file addLongs.S, line 28.
(gdb) run
Starting program: /home/daveho/.../src/control2/addLongs
Enter two integers: 3 4

Breakpoint 1, main () at addLongs.S:28
28          movq -16(%rbp), %rdi      /* pass first value */
(gdb) print *(long *)($rbp-16)
$1 = 3
(gdb) print *(long *)($rbp-8)
$2 = 4
```

Running the program in gdb

```
$ gdb addLongs
...output omitted...
(gdb) break addLongs.S:28
Breakpoint 1 at 0x401172: file addLongs.S, line 28.
(gdb) run                                <-- start the program
Starting program: /home/daveho/.../src/control2/addLongs
Enter two integers: 3 4

Breakpoint 1, main () at addLongs.S:28
28          movq -16(%rbp), %rdi          /* pass first value */
(gdb) print *(long *)($rbp-16)
$1 = 3
(gdb) print *(long *)($rbp-8)
$2 = 4
```

Running the program in gdb

```
$ gdb addLongs
...output omitted...
(gdb) break addLongs.S:28
Breakpoint 1 at 0x401172: file addLongs.S, line 28.
(gdb) run
Starting program: /home/daveho/.../src/control2/addLongs
Enter two integers: 3 4      <-- enter input values

Breakpoint 1, main () at addLongs.S:28
28          movq -16(%rbp), %rdi      /* pass first value */
(gdb) print *(long *)($rbp-16)
$1 = 3
(gdb) print *(long *)($rbp-8)
$2 = 4
```

Running the program in gdb

```
$ gdb addLongs
...output omitted...
(gdb) break addLongs.S:28
Breakpoint 1 at 0x401172: file addLongs.S, line 28.
(gdb) run
Starting program: /home/daveho/.../src/control2/addLongs
Enter two integers: 3 4

Breakpoint 1, main () at addLongs.S:28
28          movq -16(%rbp), %rdi          /* pass first value */
(gdb) print *(long *)($rbp-16)  <-- print first input value at -16(%rbp)
$1 = 3
(gdb) print *(long *)($rbp-8)
$2 = 4
```

Running the program in gdb

```
$ gdb addLongs
...output omitted...
(gdb) break addLongs.S:28
Breakpoint 1 at 0x401172: file addLongs.S, line 28.
(gdb) run
Starting program: /home/daveho/.../src/control2/addLongs
Enter two integers: 3 4

Breakpoint 1, main () at addLongs.S:28
28          movq -16(%rbp), %rdi          /* pass first value */
(gdb) print *(long *)($rbp-16)
$1 = 3
(gdb) print *(long *)($rbp-8)    <-- print second input value at -8(%rbp)
$2 = 4
```

Running the program in gdb

```
$ gdb addLongs
...output omitted...
(gdb) break addLongs.S:28
Breakpoint 1 at 0x401172: file addLongs.S, line 28.
(gdb) run
Starting program: /home/daveho/.../src/control2/addLongs
Enter two integers: 3 4

Breakpoint 1, main () at addLongs.S:28
28          movq -16(%rbp), %rdi          /* pass first value */
(gdb) print *(long *)($rbp-16)
$1 = 3
(gdb) print *(long *)($rbp-8)
$2 = 4
```