CSCI2467: Systems Programming Concepts

Slideset 11: Machine Level III: Procedures

Source: CS:APP Chapter 3, Bryant & O'Hallaron

Course Instructors:

Matthew Toups
Caitlin Boyce

Course Assistants:

Saroj Duwal
David McDonald

Spring 2020

THE UNIVERSITY *of* NEW ORLEANS

Today

- Wrapping up processes

1. Procedures
   - Passing control
   - Passing data
   - Managing local data

- Since midterm, one big concept: **processes**

ECF/IO/MEM/VM

Chapters 6,8,9,10

- Since midterm, one big concept: **processes**
- Crucial idea: each process has its own **state**

ECF/IO/MEM/VM

Chapters 6,8,9,10

- Since midterm, one big concept: **processes**
- Crucial idea: each process has its own **state**
- . . . with many important implications:

ECF/IO/MEM/VM

Chapters 6,8,9,10

- Since midterm, one big concept: **processes**
- Crucial idea: each process has its own **state**
- . . . with many important implications:
- control flow (context switching, signals, system calls)

ECF/IO/MEM/VM

Chapters 6,8,9,10

- Since midterm, one big concept: **processes**
- Crucial idea: each process has its own **state**
- . . . with many important implications:
- control flow (context switching, signals, system calls)
- CPU state (registers, stack, instruction pointer)

ECF/IO/MEM/VM

Chapters 6,8,9,10

- Since midterm, one big concept: **processes**
- Crucial idea: each process has its own **state**
- . . . with many important implications:
  - control flow (context switching, signals, system calls)
  - CPU state (registers, stack, instruction pointer)
  - Memory state (address space, protection, caching, `fds`)

ECF/IO/MEM/VM

Chapters 6,8,9,10

- Since midterm, one big concept: **processes**
- Crucial idea: each process has its own **state**
- . . . with many important implications:
  - control flow (context switching, signals, system calls)
  - CPU state (registers, stack, instruction pointer)
  - Memory state (address space, protection, caching, `fds`)
- All *independent*, yet sharing one physical system

ECF/IO/MEM/VM

Chapters 6,8,9,10

- Since midterm, one big concept: **processes**
- Crucial idea: each process has its own **state**
- . . . with many important implications:
  - control flow (context switching, signals, system calls)
  - CPU state (registers, stack, instruction pointer)
  - Memory state (address space, protection, caching, `fds`)
- All *independent*, yet sharing one physical system
- Thanks to *abstractions*: processes and virtual memory (VM)

You have more insight into systems!

. . . and yes, it *will* be on the exam

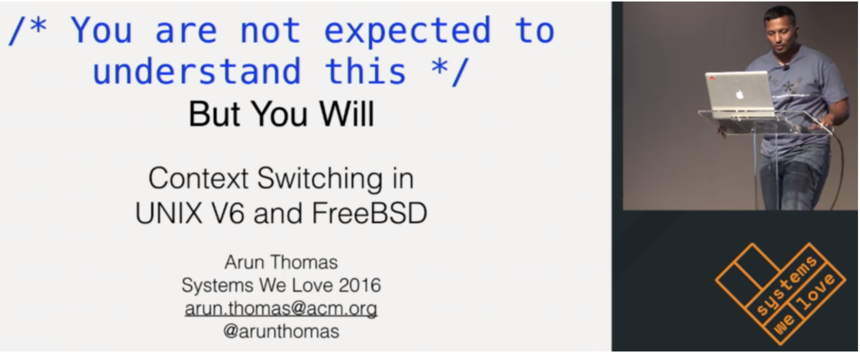
Processes (and associated abstractions) are the key to systems!

. . . also somewhat *notorious*

You have more insight into systems!

. . . and yes, it *will* be on the exam

Processes (and associated abstractions) are the key to systems!



/* You are not expected to understand this */

But You Will

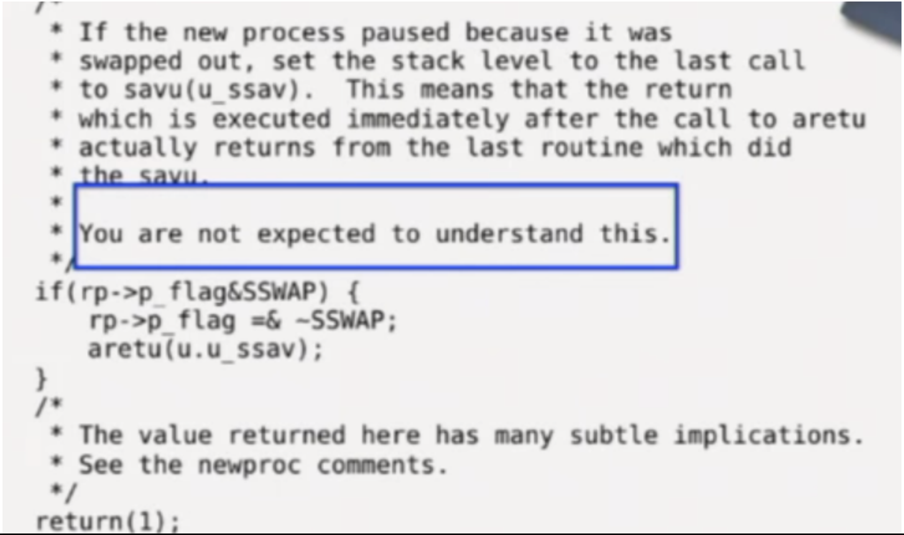
Context Switching in UNIX V6 and FreeBSD

Arun Thomas
Systems We Love 2016
arun.thomas@acm.org
@arunthomas

You have more insight into systems!

. . . and yes, it *will* be on the exam

Processes (and associated abstractions) are the key to systems!

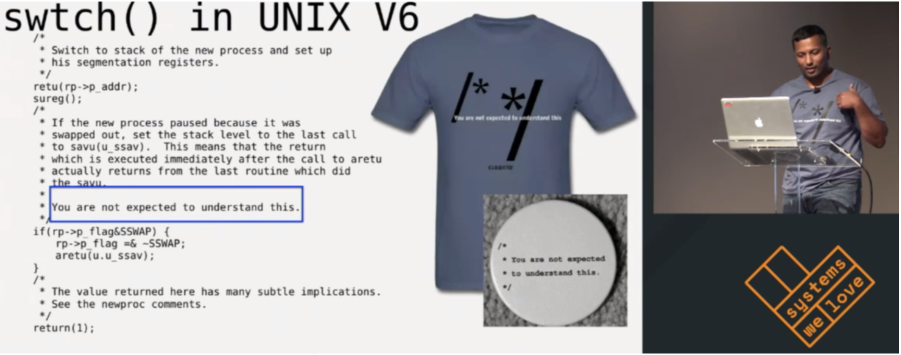


```
/*
 * If the new process paused because it was
 * swapped out, set the stack level to the last call
 * to savu(u_ssav).  This means that the return
 * which is executed immediately after the call to aretu
 * actually returns from the last routine which did
 * the savu.
 *
 * You are not expected to understand this.
 */
if(rp->p_flag&SSWAP) {
    rp->p_flag =& ~SSWAP;
    aretu(u.u_ssav);
}
/*
 * The value returned here has many subtle implications.
 * See the newproc comments.
 */
return(1);
```

You have more insight into systems!

. . . and yes, it *will* be on the exam

Processes (and associated abstractions) are the key to systems!

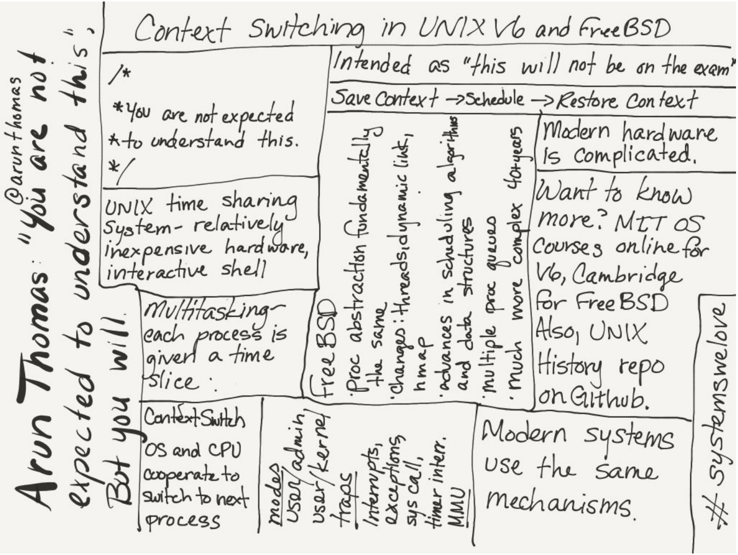


swtch() in UNIX V6

```
/*
 * Switch to stack of the new process and set up
 * his segmentation registers.
 */
retu(rp->p_addr);
sureg();
/*
 * If the new process paused because it was
 * swapped out, set the stack level to the last call
 * to savu(u_ssav).  This means that the return
 * which is executed immediately after the call to aretu
 * actually returns from the last routine which did
 * the savu.
 *
 * You are not expected to understand this.
 */
if(rp->p_flag&SSWAP) {
    rp->p_flag =& ~SSWAP;
    aretu(u.u_ssav);
}
/*
 * The value returned here has many subtle implications.
 * See the newproc comments.
 */
return(1);
```

You have more insight into systems!

. . . and yes, it *will* be on the exam



@arunthomas

Arun Thomas: "You are not expected to understand this", But you will.

Context Switching in UNIX V6 and FreeBSD

/*
*You are not expected
*to understand this.
*/

Intended as "this will not be on the exam"

Save Context -> Schedule -> Restore Context

UNIX time sharing System - relatively inexpensive hardware, interactive shell

Multitasking - each process is given a time slice.

Context Switch OS and CPU cooperate to switch to next process

modes user/admin, user/kernel, traps

Interrupts, exceptions, sys call, timer interr. MMU

FreeBSD
- Proc abstraction fundamentally the same
- changes: threads, dynamic link, mmap
- advances in scheduling algorithms and data structures
- multiple proc queues
- much more complex 40+ years

Modern hardware is complicated.

Want to know more? MIT OS Courses online for V6, Cambridge for FreeBSD Also, UNIX History repo on Github.

Modern systems use the same mechanisms.

systemswelove

The infamous source code comment

Dr. Summa's link:

An explainer on Unix's most notorious code comment

Also, System Performance

- CPU/Memory/disk/network speed gap is central to performance questions

Also, System Performance

- CPU/Memory/disk/network speed gap is central to performance questions
- *Huge* differences between each, orders of magnitude

Also, System Performance

- CPU/Memory/disk/network speed gap is central to performance questions
- *Huge* differences between each, orders of magnitude
- Gaps have been growing for a long time, driving hardware and software design

Also, System Performance

- CPU/Memory/disk/network speed gap is central to performance questions
- *Huge* differences between each, orders of magnitude
- Gaps have been growing for a long time, driving hardware and software design
- **Caching** is how we've addressed this gap

Also, System Performance

- CPU/Memory/disk/network speed gap is central to performance questions
- *Huge* differences between each, orders of magnitude
- Gaps have been growing for a long time, driving hardware and software design
- **Caching** is how we've addressed this gap
- ... which depends on the idea of **Locality**

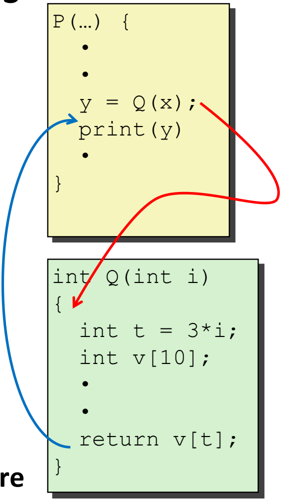
Today

- Wrapping up processes

1. Procedures
   - Mechanisms
   - Stack structure
   - Calling conventions
     - Passing control
     - Passing data
     - Managing local data
   - Illustration of recursion

Mechanisms in Procedures

- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**

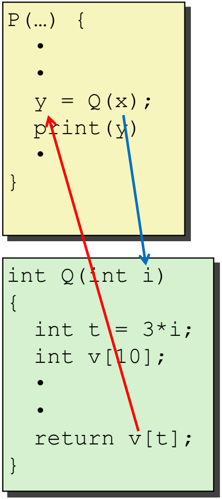


```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

Mechanisms in Procedures

- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**



```
P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

```
int Q(int i)
{
   int t = 3*i;
   int v[10];
   •
   •
   return v[t];
}
```

Mechanisms in Procedures

- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

Today

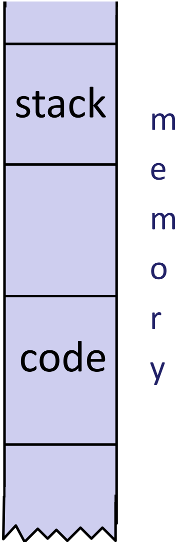
- Wrapping up processes

1. Procedures
   - Mechanisms
   - Stack structure
   - Calling conventions
     - Passing control
     - Passing data
     - Managing local data
   - Illustration of recursion

x86-64 stack

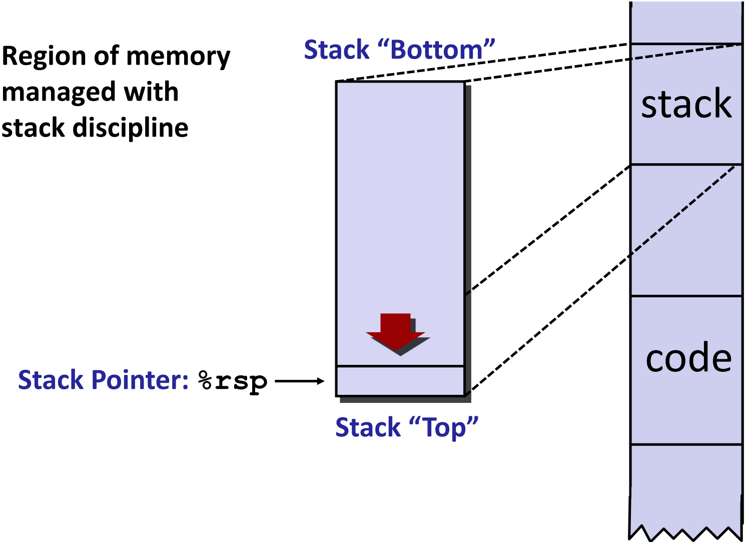
Region of memory managed with stack discipline

- Memory viewed as array of bytes.
- Different regions have different purposes.
- (Like ABI, a policy decision)



x86-64 stack

Region of memory managed with stack discipline



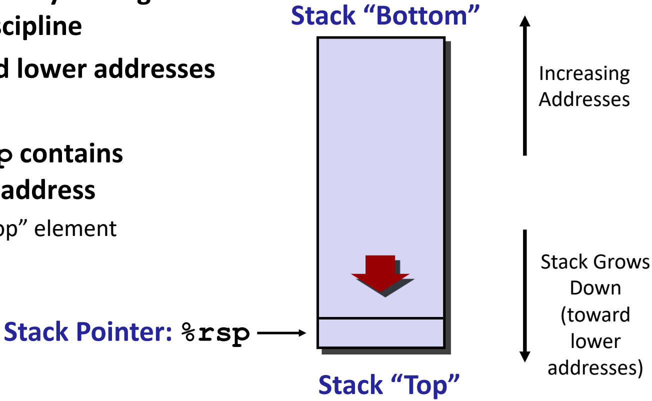
x86-64 stack

Region of memory managed with stack discipline

Grows toward lower addresses

Register `%rsp` contains lowest stack address

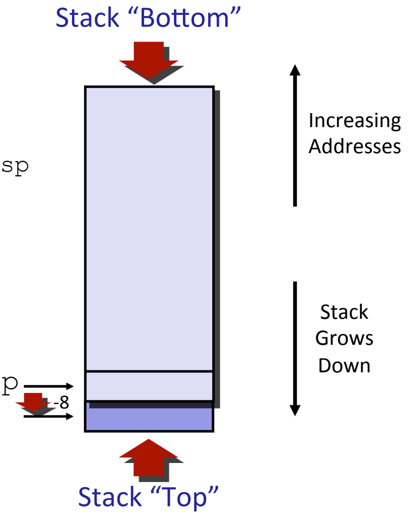
- address of “top” element



x86-64 stack: push

`pushq` Src

- Fetch operand at Src
- Decrement `%rsp` by 8
- Write operand at address given by `%rsp`

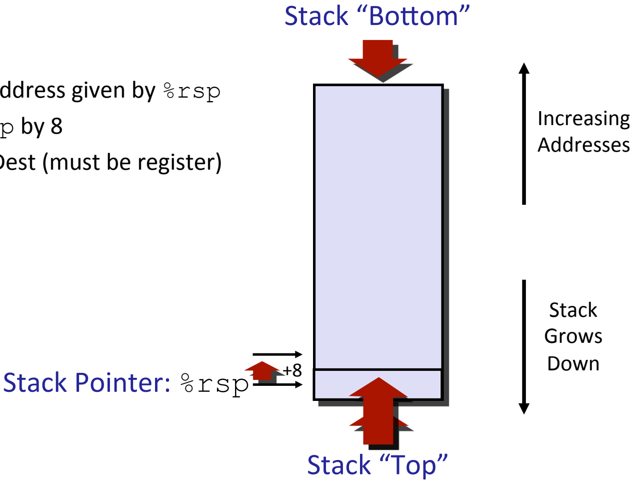


Stack Pointer: `%rsp`

x86-64 stack: pop

`popq` Dest

- Read value at address given by `%rsp`
- Increment `%rsp` by 8
- Store value at Dest (must be register)



- Wrapping up processes

1. Procedures
    - Mechanisms
    - Stack structure
    - Calling conventions
        - Passing control
        - Passing data
        - Managing local data
    - Illustration of recursion

Code Examples

```
void multstore(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400540: push   %rbx                # Save %rbx
  400541: mov    %rdx,%rbx           # Save dest
  400544: callq  400550 <mult2>      # mult2(x,y)
  400549: mov    %rax,(%rbx)         # Save at dest
  40054c: pop    %rbx                # Restore %rbx
  40054d: retq                       # Return
```

```
long mult2(long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax       # a
  400553:  imul   %rsi,%rax       # a * b
  400557:  retq                   # Return
```

Procedure Control Flow

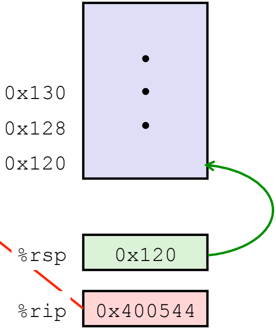
- Use stack to support procedure call and return
- Procedure call: `call label`
  - push return address on stack
  - jump to label
- Return address:
  - address of the next instruction right after call

    (example from disassembly)
- Procedure return: `ret`
  - pop address from stack
  - jump to address

Control Flow Example #1

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

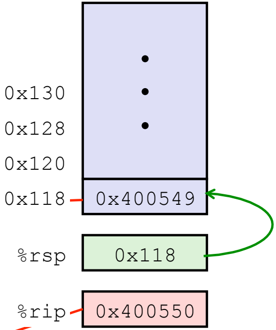
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

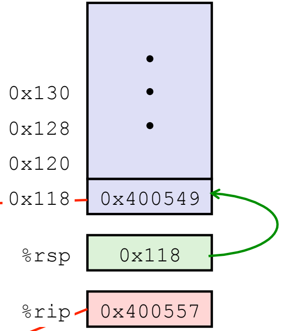
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #3

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

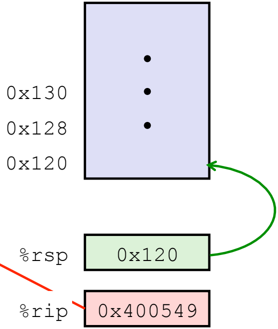
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #4

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Procedure Data Flow

Registers

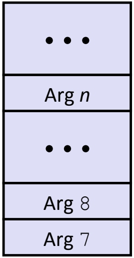
- **First 6 arguments**

`%rdi`
`%rsi`
`%rdx`
`%rcx`
`%r8`
`%r9`

- **Return value**

`%rax`

Stack



- **Only allocate stack space when needed**

Data Flow Examples

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  • • •
  400541: mov    %rdx,%rbx        # Save dest
  400544: callq  400550 <mult2>   # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)      # Save at dest
  • • •
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov    %rdi,%rax       # a
  400553:  imul   %rsi,%rax       # a * b
  # s in %rax
  400557:  retq                   # Return
```

Stack used for procedures

- Stack used in languages which support recursion
  - examples: C, Java
  - code must be “re-entrant”

    (multiple simultaneous instantiations of single procedure)

- Why stack?

  We need some place to store *state* of each instantiation:

  - arguments
  - local variables
  - return pointer

Stack used for procedures

- Stack discipline:
  - state for given procedure needed for limited time
    - from when called to when return
  - callee returns before caller does
- Stack allocated in *frames*:
  - frame holds state for a single procedure instantiation

Call Chain Example

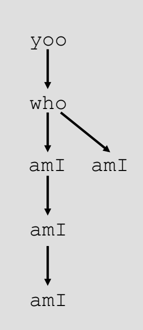
```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

```
who(…)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}
```

```
amI(…)
{
  •
  •
  amI();
  •
  •
}
```

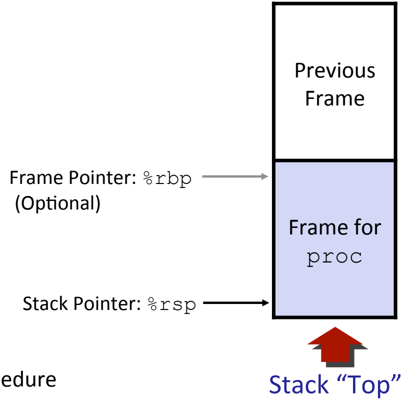
Procedure `amI()` is recursive

Example
Call Chain



Stack Frames

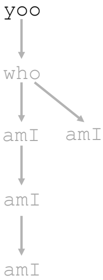
- **Contents**
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)

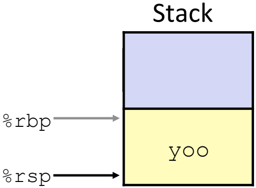


- **Management**
  - Space allocated when enter procedure
    - "Set-up" code
    - Includes push by `call` instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by `ret` instruction

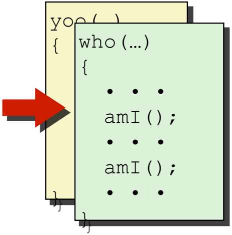
Example

```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```



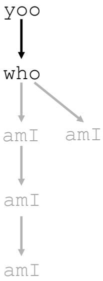


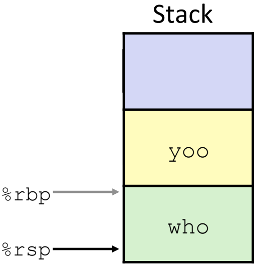
Example



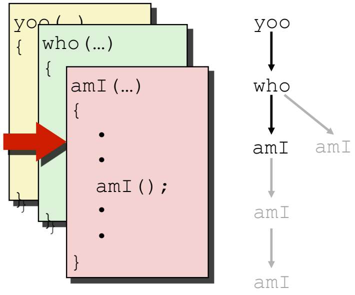
```
yoo(…)
{

}
```

```
who(…)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}
```





Example



```
yoo(…)
{

}
```

```
who(…)
{

}
```

```
amI(…)
{
  •
  •
  amI();
  •
  •
}
```

yoo

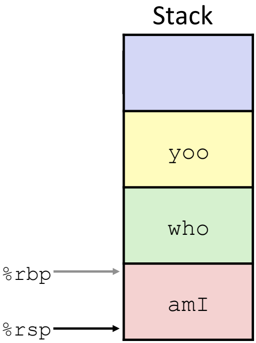
who

amI

amI

amI

amI



Stack

yoo

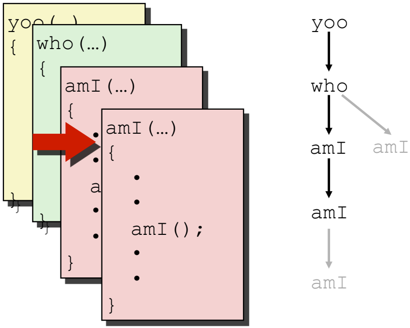
who

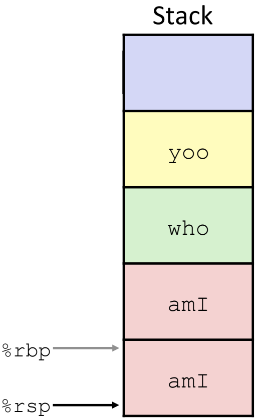
amI

%rbp

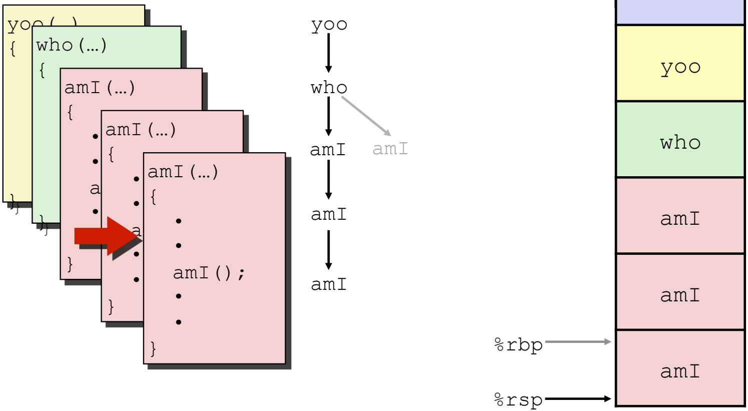
%rsp

Example

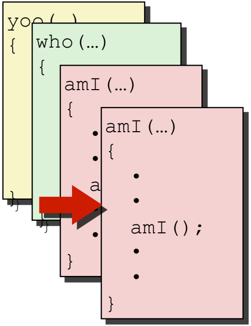

yoo(…)
{
}
who(…)
{
}
amI(…)
{
a
}
amI(…)
{
amI();
}
yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

Example


yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
a
•
}
amI(…)
{
•
•
a
•
•
}
amI(…)
{
•
•
amI();
•
•
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
amI
amI
%rbp
%rsp

Example

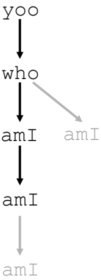


```
yoo(…)
{
```

```
who(…)
{
```

```
amI(…)
{
```

```
amI(…)
{
  •
  •
  amI();
  •
  •
}
```



yoo

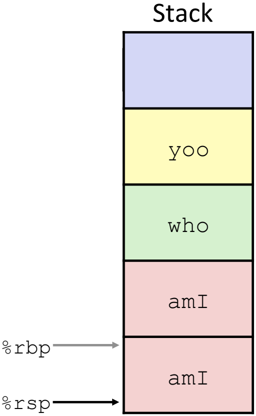
who

amI

amI

amI

amI



Stack

yoo

who

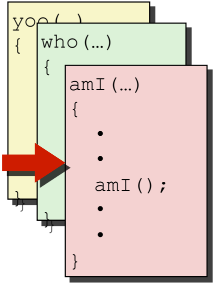
amI

amI

%rbp

%rsp

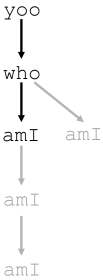
Example

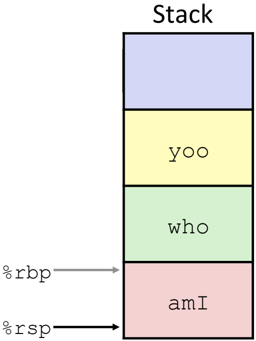


```
yoo(…)
{
```

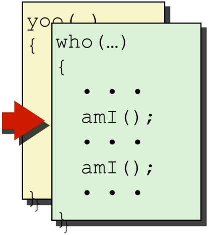
```
who(…)
{
```

```
amI(…)
{
  •
  •
  amI();
  •
  •
}
```



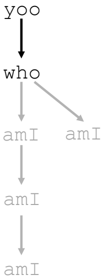


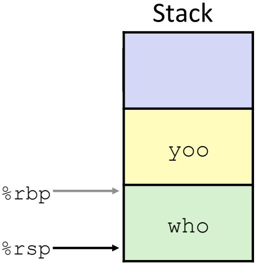
Example



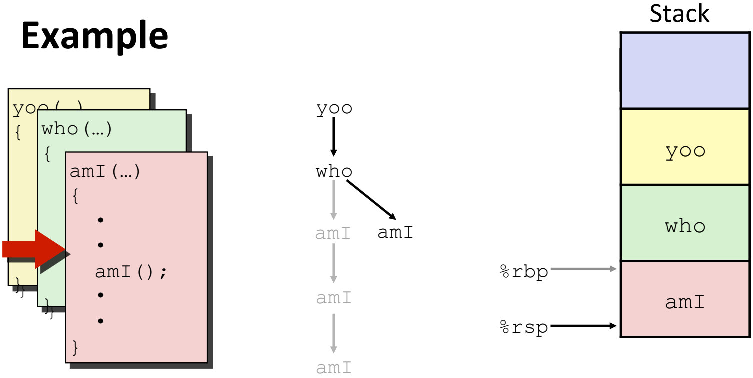
```
yoo(…)
{
```

```
who(…)
{
   •  •  •
   amI();
   •  •  •
   amI();
   •  •  •
}
```



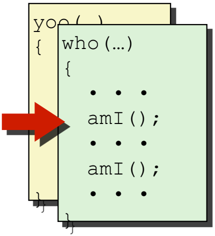


Example

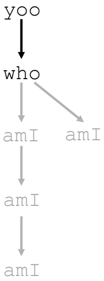


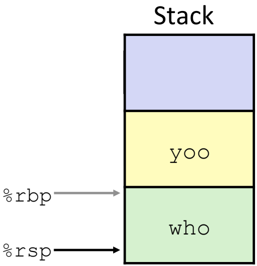
```
yoo(…)
{
who(…)
{
amI(…)
{
  •
  •
  amI();
  •
  •
}
}
}
```

Example



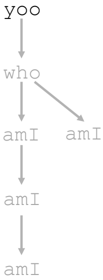
```
yoo(…)
{
  who(…)
  {
    • • •
    amI();
    • • •
    amI();
    • • •
  }
}
```

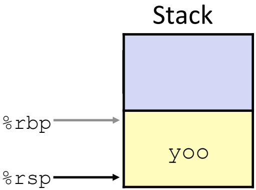




Example

```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

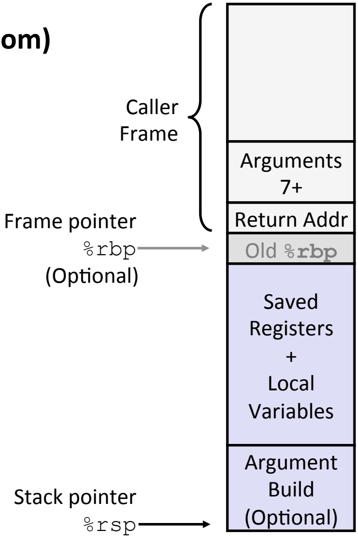




x86-64/Linux stack frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build:"
    Parameters for function about to call
  - Local variables
    If can't keep in registers
  - Saved register context
  - Old frame pointer (optional)

- **Caller Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call


Caller Frame
Arguments 7+
Return Addr
Frame pointer %rbp (Optional)
Old %rbp
Saved Registers + Local Variables
Argument Build (Optional)
Stack pointer %rsp

Example: incr

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

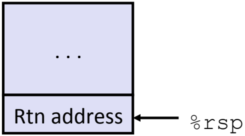
```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

Register	Use(s)
`%rdi`	Argument **`p`**
`%rsi`	Argument **`val`**, **`y`**
`%rax`	**`x`**, Return value

Example: Calling incr

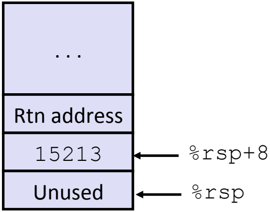
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Initial Stack Structure



```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Resulting Stack Structure

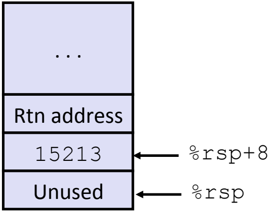


Example: Calling incr

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq     $16, %rsp
  movq     $15213, 8(%rsp)
  movl     $3000, %esi
  leaq     8(%rsp), %rdi
  call     incr
  addq     8(%rsp), %rax
  addq     $16, %rsp
  ret
```

Stack Structure



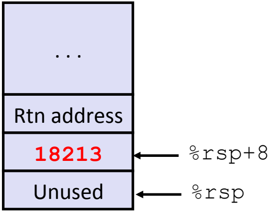
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling incr

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



. . .

Rtn address

18213

Unused

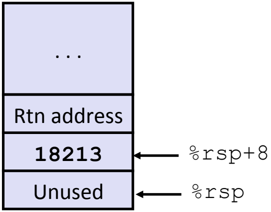
%rsp+8

%rsp

Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling incr

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```



. . .

Rtn address

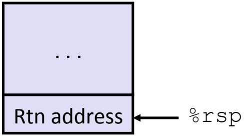
18213 ← %rsp+8

Unused ← %rsp

```
call_incr:
  subq     $16, %rsp
  movq     $15213, 8(%rsp)
  movl     $3000, %esi
  leaq     8(%rsp), %rdi
  call     incr
  addq     8(%rsp), %rax
  addq     $16, %rsp
  ret
```

Register	Use(s)
%rax	Return value

Updated Stack Structure



. . .

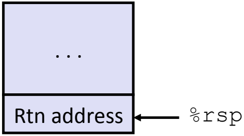
Rtn address ← %rsp

Example: Calling incr

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

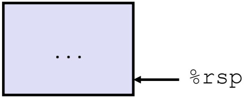
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Updated Stack Structure



Register	Use(s)
%rax	Return value

Final Stack Structure



Register saving conventions

- **When procedure yoo calls who:**
  - yoo is the caller
  - who is the callee
- **Can register be used for temporary storage?**

```
yoo:
    • • •
    movq $15213, %rdx
    call  who
    addq %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq $18213, %rdx
    • • •
     ret
```

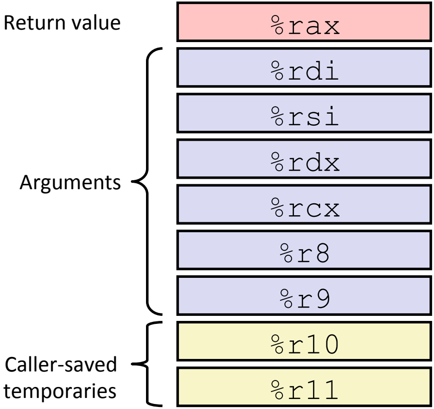
- Contents of register %rdx overwritten by who
- This could be trouble → something should be done!
  - Need some coordination

Register saving conventions

- **When procedure `yoo` calls `who`:**
  - `yoo` is the caller
  - `who` is the callee
- **Can register be used for temporary storage?**
- **Conventions**
  - "Caller Saved"
    - Caller saves temporary values in its frame before the call
  - "Callee Saved"
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

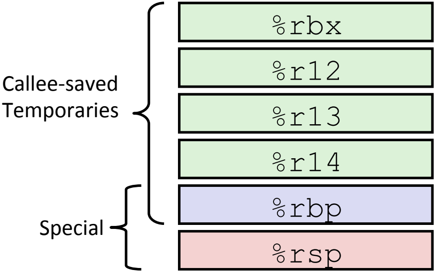
x86-64/linux register usage

- **`%rax`**
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- **`%rdi`**, ..., **`%r9`**
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- **`%r10`**, **`%r11`**
  - Caller-saved
  - Can be modified by procedure



x86-64/linux register usage

- **`%rbx`, `%r12`, `%r13`, `%r14`**
  - Callee-saved
  - Callee must save & restore
- **`%rbp`**
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match
- **`%rsp`**
  - Special form of callee save
  - Restored to original value upon exit from procedure

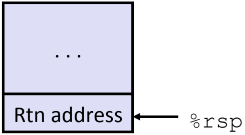


Callee-saved example

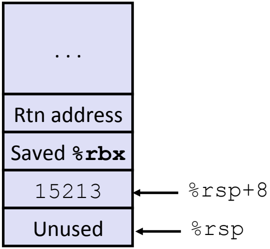
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



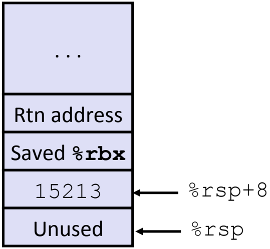
Resulting Stack Structure



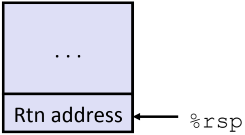
Callee-saved example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq    %rbx
  subq     $16, %rsp
  movq     %rdi, %rbx
  movq     $15213, 8(%rsp)
  movl     $3000, %esi
  leaq     8(%rsp), %rdi
  call     incr
  addq     %rbx, %rax
  addq     $16, %rsp
  popq     %rbx
  ret
```



Pre-return Stack Structure



- Wrapping up processes

1. Procedures
    - Mechanisms
    - Stack structure
    - Calling conventions
        - Passing control
        - Passing data
        - Managing local data
    - Illustration of recursion

Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Recursive Function Terminal Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

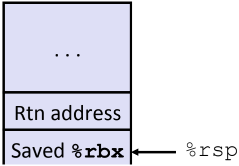
Register	Use(s)	Type
`%rdi`	`x`	Argument
`%rax`	Return value	Return value

Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```

Register	Use(s)	Type
%rdi	x	Argument



Recursive Function Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```

Register	Use(s)	Type
`%rdi`	`x >> 1`	Rec. argument
`%rbx`	`x & 1`	Callee-saved

Recursive Function Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
`%rbx`	`x & 1`	Callee-saved
`%rax`	Recursive call return value	

Recursive Function Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

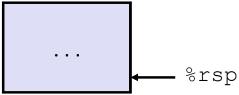
Register	Use(s)	Type
`%rbx`	`x & 1`	Callee-saved
`%rax`	Return value	

Recursive Function Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
`%rax`	Return value	Return value



...

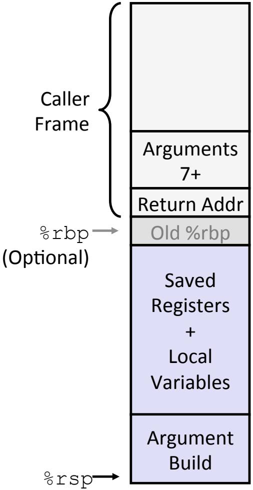
%rsp

Observations About Recursion

- **Handled Without Special Consideration**
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
- **Also works for mutual recursion**
  - P calls Q; Q calls P

x86-64 Procedure Summary

- **Important Points**
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- **Recursion (& mutual recursion) handled by normal calling conventions**
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in `%rax`
- Pointers are addresses of values
  - On stack or global



Caller Frame
Arguments 7+
Return Addr
%rbp
(Optional)
Old %rbp
Saved Registers + Local Variables
Argument Build
%rsp