Instruction Set Architectures and Closures

CIS400 (Compiler Construction)
Kris Micinski, Fall 2021

- In the last lecture, we talked about closure conversion
- Today, I want to continue to motivate why closure conversion is a key step in compilation of functional languages

Key ideas:

- Closure conversion **eliminates lambdas** by turning them into "C-style" functions
  - ***Why*?**
    - Computers really good at calling C-style functions!
    - Processor has a native **call** instruction to invoke first-order procedures (C functions)
- Closure invocation then becomes *dynamic dispatch*

- In the last lecture, we talked about closure conversion
- Today, I want to continue to motivate why closure conversion is a key step in compilation of functional languages

In today's lecture, I hope to motivate this…

Key ideas:

- Closure conversion **eliminates lambdas** by turning them into "C-style" functions
  - ***Why*?**
    - Computers really good at calling C-style functions!
    - Processor has a native **call** instruction to invoke first-order procedures (C functions)
- Closure invocation then becomes *dynamic dispatch*

Course Detour — Assembly Crash Course

By which I mean x86-64 assembly...

Note to readers…

In this set of slides, I give a brief detour of how x86-64 assembly works. In this course we will not use x86-64 assembly, we will use LLVM assembly. LLVM is a higher-level assembly than x86-64, and while many of the concepts will be similar, they will not be identical.

ISA

- The **I**nstruction **S**et **A**rchitecture is the term used to refer to the native language executed by a (hardware) processor
    - vs. (say) a *bytecode* executed by a virtual machine (such as the JVM)
- All processors execute a set of very basic commands called *instructions*
- These instructions are kind of like ANF/CPS—they atomically execute one instruction at a time

Mov, sub, etc… are destination-last here, but there is no standard and in practice you should get used to figuring out if the instruction is destination-first or destination-last…

Opcode — instruction name

```
pushl     %ebp       ; save calling function's stack frame (ebp)
movl      %esp, %ebp  ; make a new stack frame on top of our caller's stack
subl      $4, %esp    ; allocate 4 bytes of stack space for local variables
```

Operands—atomic arguments to instructions

Operands must be atomic

- There are several kinds of atomic values in assembly:
  - Literal values
  - Registers (high-speed temporary variables close to process)
  - RAM (may be backed by cache) such as the stack

Registers

Originally, 8-bit registers: al, bl, cl, dl

Traditionally, x86 architectures only had **four** 16-bit general purpose registers: ax, bx, cx, dx

Also other registers: bp, sp, di, si

Base pointer

(Start of frame)

Stack pointer

(Top of stack)

Originally, 8-bit registers: al, bl, cl, dl

Traditionally, x86 architectures only had **four** 16-bit general purpose registers: ax, bx, cx, dx

Also other registers: bp, sp, di, si

Base pointer

(Start of frame)

Stack pointer

(Top of stack)

IP: instruction pointer

Points at current instruction, incremented after each instruction

FLAGS: holds flags

Set on subtraction, comparison, etc..

Traditionally, x86 architectures only had **four** 16-bit general purpose registers: ax, bx, cx, dx

Also other registers: bp, sp, di, si

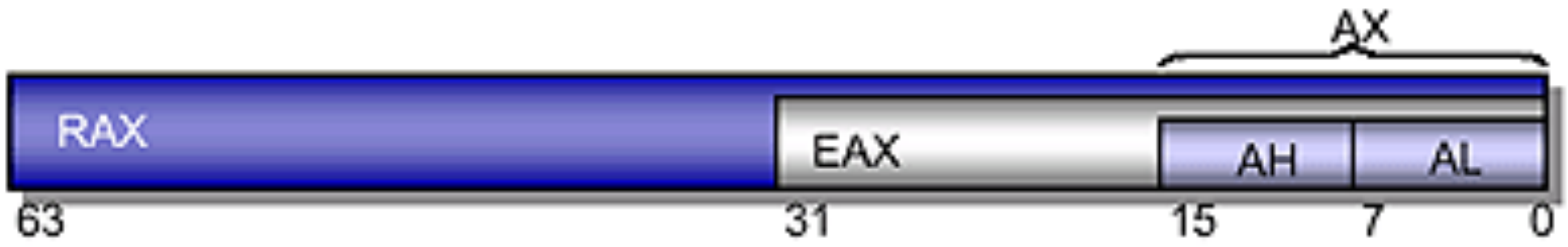
As time progressed, also added 32-bit registers: eax, ebx, ecx, edx

In past few years, 64-bit registers: rax, rbx, rcx, rdx

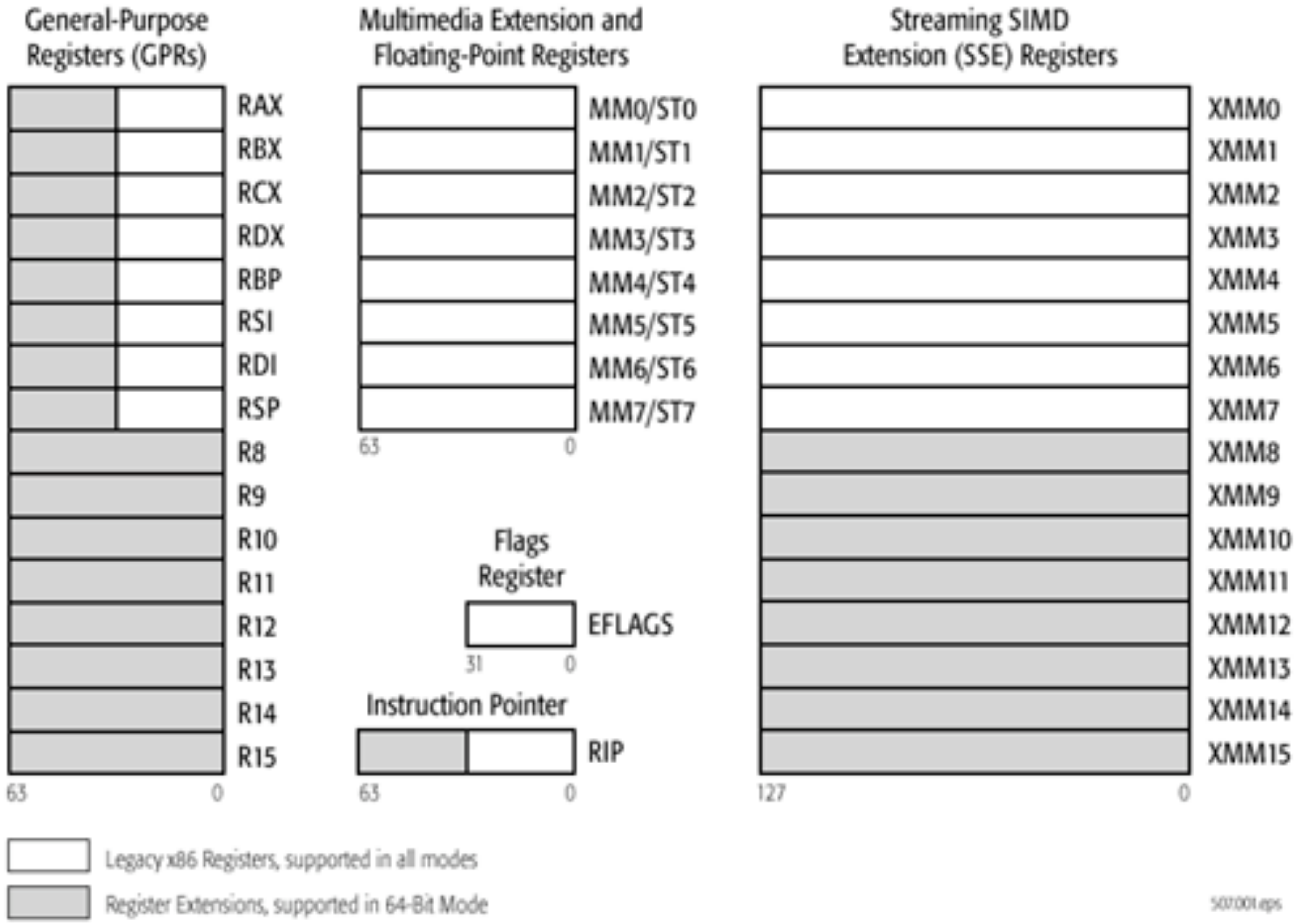
(Also 64-bit versions: rip, etc..)

We'll pretty much exclusively use 64-bit registers!

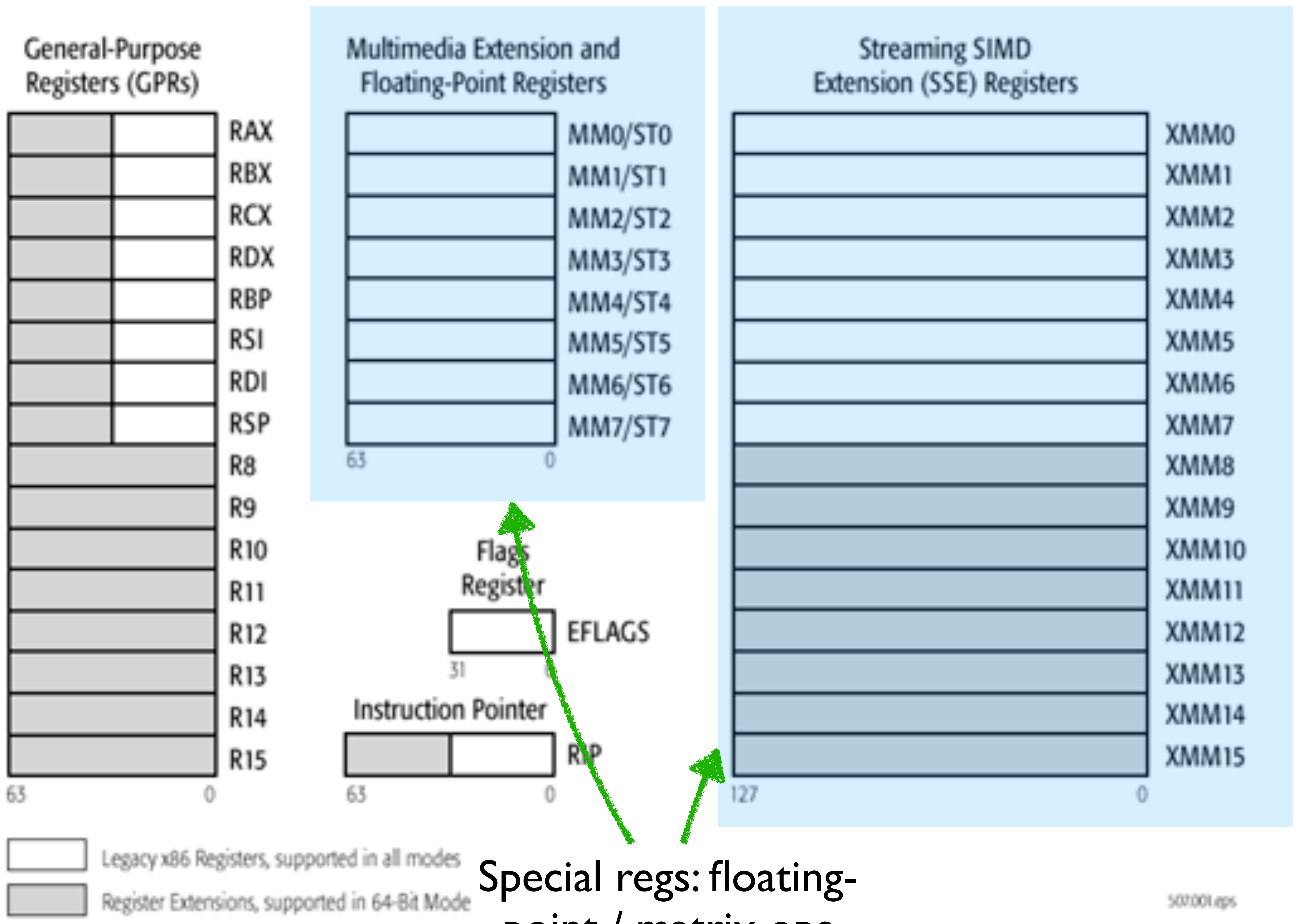
Note RAX is an **extension** of EAX



If you change EAX, you change lower 32 bits of RAX

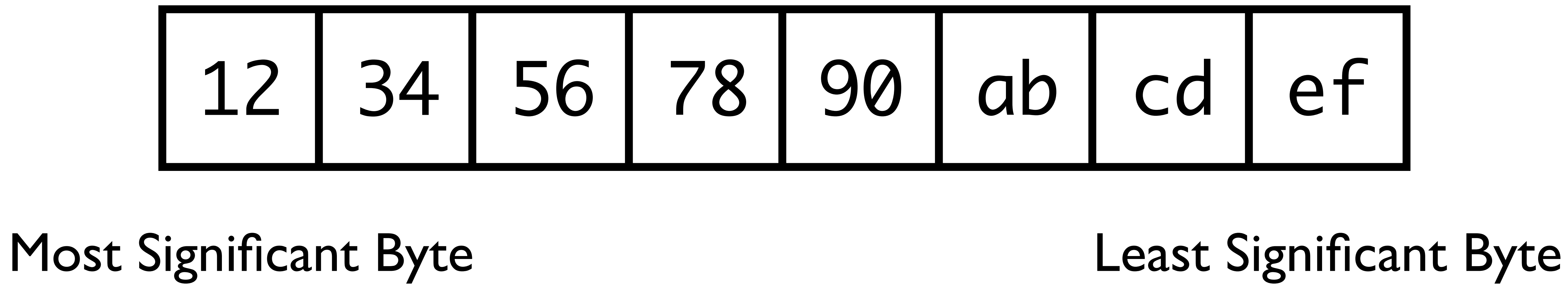


General-Purpose Registers (GPRs)
RAX
RBX
RCX
RDX
RBP
RSI
RDI
RSP
R8
R9
R10
R11
R12
R13
R14
R15
63
0
Multimedia Extension and Floating-Point Registers
MM0/ST0
MM1/ST1
MM2/ST2
MM3/ST3
MM4/ST4
MM5/ST5
MM6/ST6
MM7/ST7
63
0
Flags Register
EFLAGS
31
0
Instruction Pointer
RIP
63
0
Streaming SIMD Extension (SSE) Registers
XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7
XMM8
XMM9
XMM10
XMM11
XMM12
XMM13
XMM14
XMM15
127
0
Legacy x86 Registers, supported in all modes
Register Extensions, supported in 64-Bit Mode
507001.eps



Special regs: floating-point / matrix ops

To represent `0x1234567890abcdef`



x86 is a **little-endian** architecture

If an n-byte value is stored at addresses a to a+(n-1) in memory, byte a will hold the **least significant byte**

`0x1234567890abcdef`

Exercise with partner

Instructions

Binary code is made up of giant sequences of "instructions"

Modern Intel / AMD chip has hundreds of them, some very complex

Moving memory around　　Arithmetic　　Branch / If

Matrix operations　　Atomic-Instructions

Transactional memory instructions

Encoded as binary (as you may have seen from hardware-design course)

We (humans) write in a format named "assembly"

Confusingly: two types of assembly

AT&T	Intel
`movq 5, %rax`	`mov rax, 5`

Several **addressing modes**

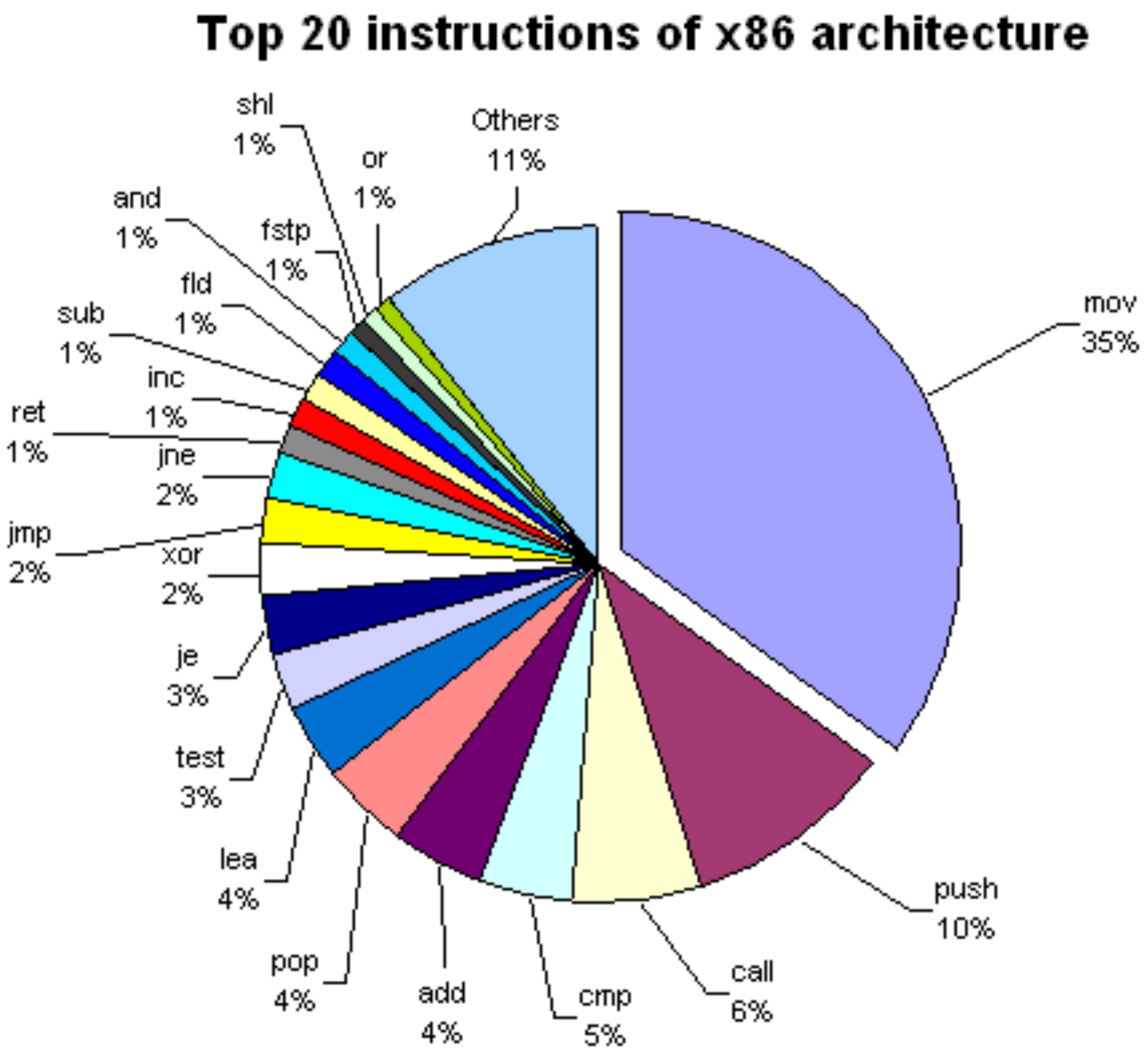
“Move the value from register rax into the register rbx”

Opcode name

Destination

```
mov  %rax, %rbx
```

Source



Plurality of instructions are **mov**s

Then **push**

Then **call**

Memory: a **giant chunk of bytes**

You can read from it and write to it in 1/2/4/8/16-byte increments

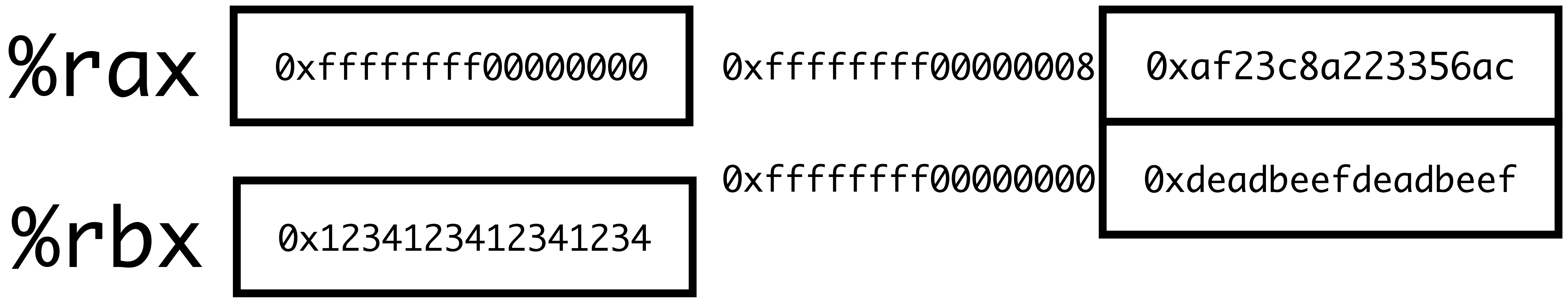
```
mov  (%rax), %rbx
```

“Move the value **at address** %rax into register %rbx”

Opcode name Destination

```
mov  (%rax), %rbx
```

Source



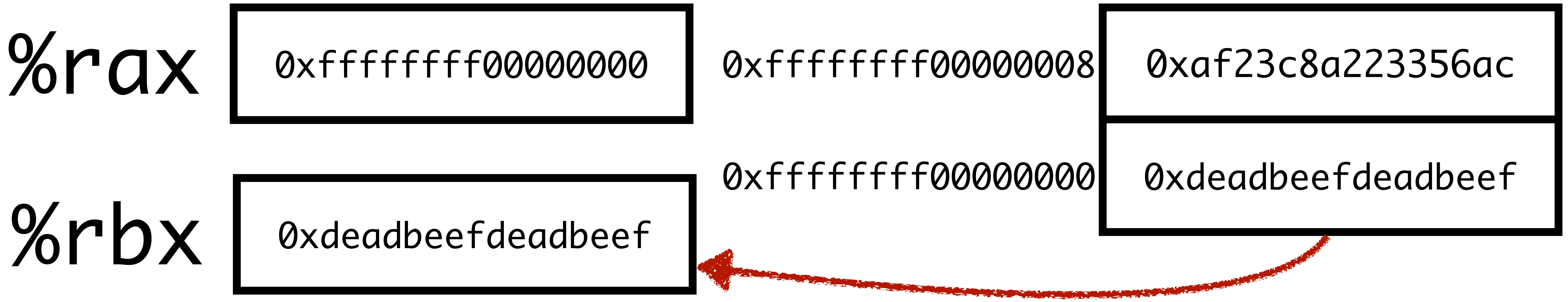
“Move the value **at address** %rax into register %rbx”

Opcode name

Destination

```
mov  (%rax), %rbx
```

Source



“Move the value **at address** %rax+8 into register %rbx”

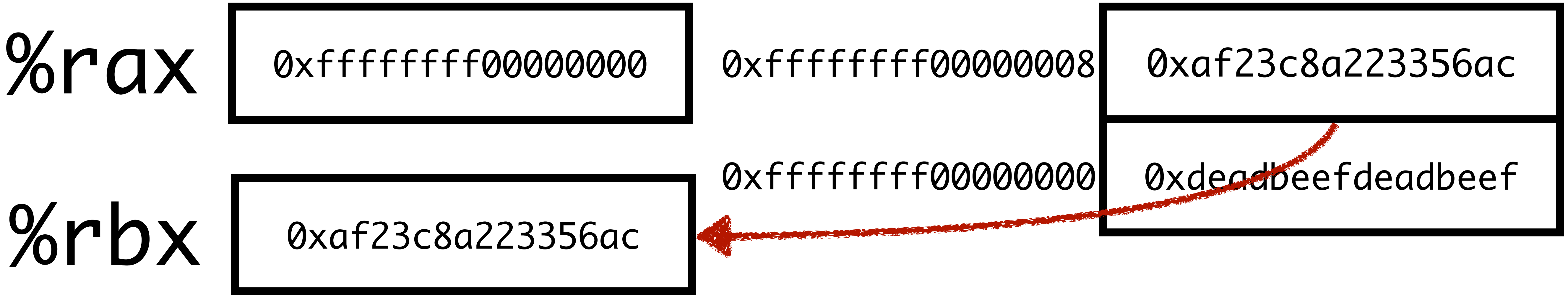
Opcode name

Destination

```
mov  8(%rax), %rbx
```

Source

```
(let ([%rbx (prim vector-ref %rax 1)])
…)
```



A few other more complicated ones that
allow you to add registers, offsets, etc…

Different instructions allow different addressing-modes

Caching

- Accessing RAM is slow, much slower than registers (~1-2 orders of magnitude, depending on how recently used)

- All memory access is backed by a **cache**, which mirrors recently-accessed pieces of RAM in an on-chip cache

Caching (contd…)

- When *cache miss* occurs (data not in cache), processor pulls the data's enclosing **cache line** (64-byte-aligned sequence of memory around the data) into the cache
  - Thus, accessing data within that cache line will be *fast*
- Designing for cache friendliness can be crucial
  - One **take-away** point for **all** students—if you touch data once, all data sharing the same cache line will be extremely fast to use
    - ***One course lesson — Situate data that will be used together on the same cache line!***

RISC vs. CISC

- x86_64 (i.e., Intel/AMD-style assembly) is a **C**omplex **I**nstruction **S**et **C**omputer—a huge mess of complex instructions to do tons of random things
    - This design is crummy for lots of reasons, and modern machines decode x86_64 into an internal RISC anyway, wasting lot of silicon area
- ARM is a competitor to x86_64, and it is a **R**educed **I**nstruction **S**et **C**omputer (RISC) instruction set
    - ARM has taken over the world of micro devices, phones, tablets, and now laptops and server chips
        - Much lower power, simpler instruction set, conventional wisdom says it will be less fast (will it? Likely not, server-class ARM chips now power world’s fastest supercomputer by Fujitsu)

Example ARM code…

```
    int total;
    int i;

    total = 0;
    for (i = 10; i > 0; i--) {
        total += i;
    }
```

```
        MOV   R0, #0              ; R0 accumulates total
        MOV   R1, #10             ; R1 counts from 10 down to 1
again   ADD   R0, R0, R1
        SUBS  R1, R1, #1
        BNE   again
halt    B     halt                ; infinite loop to stop computation
```

Carl Burch (Hendrix College) —
http://www.cburch.com/books/arm/

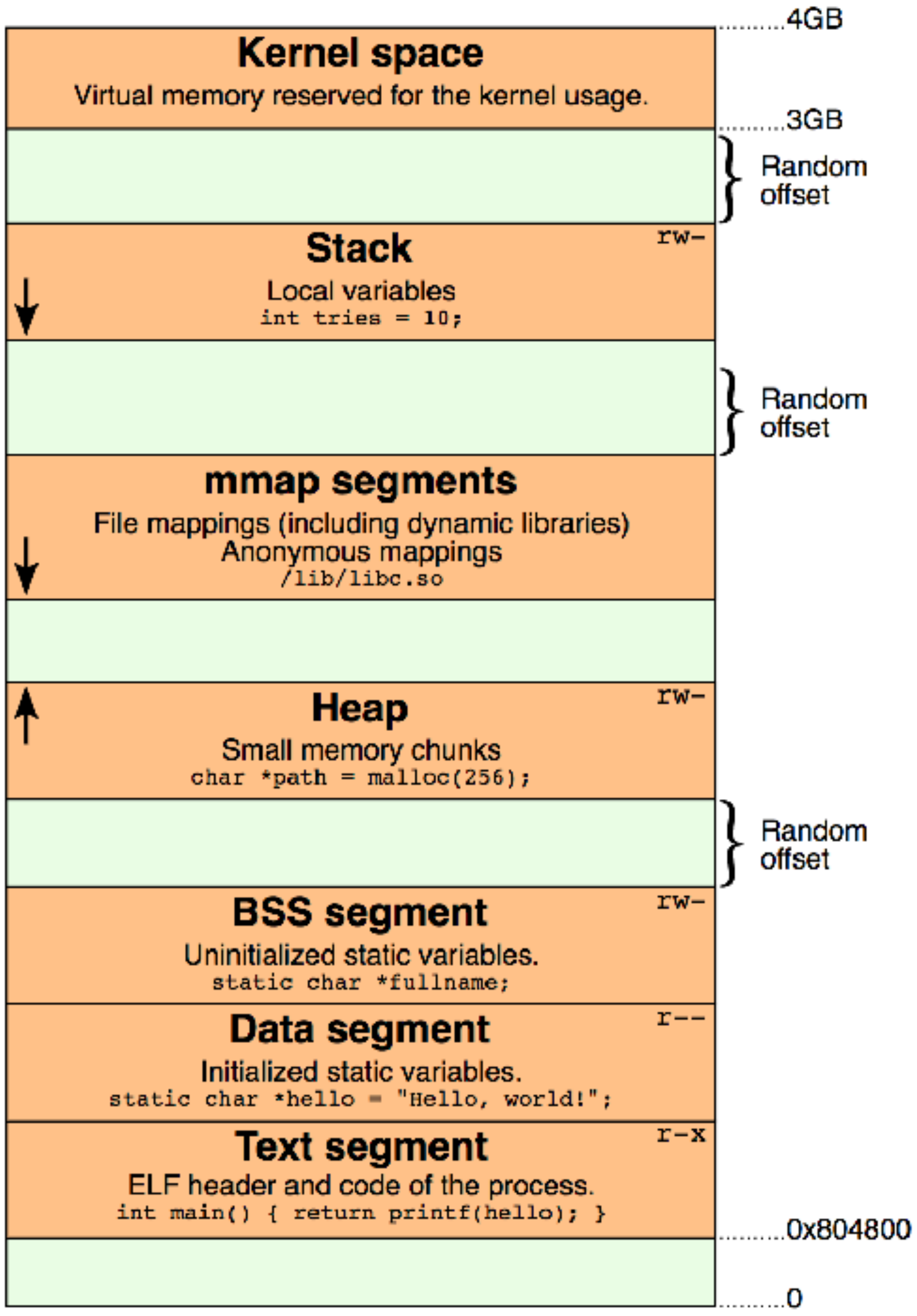
Memory is divided into different regions

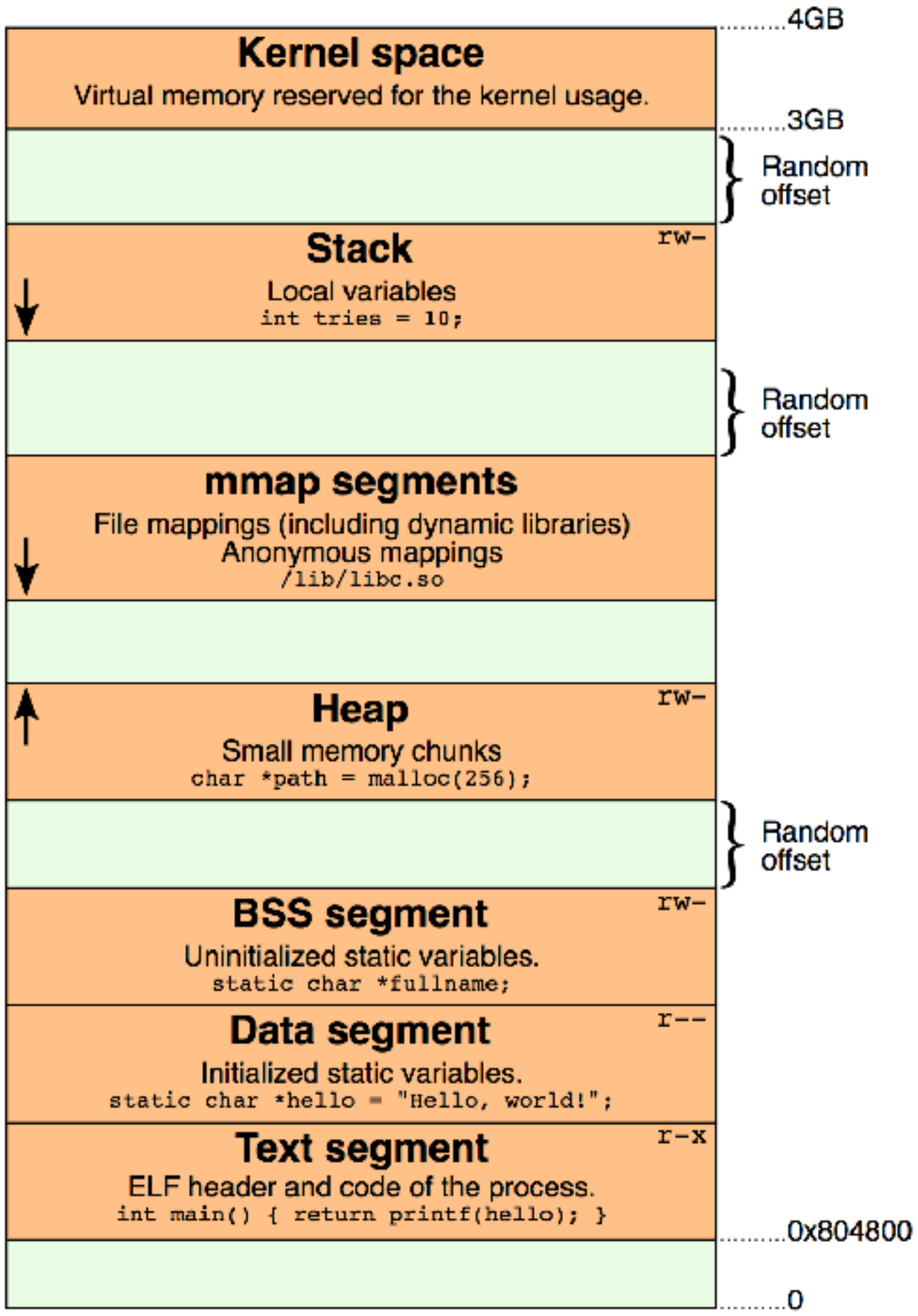
C stack (grows up and down)
Heap (`new/malloc'd` data)
Static variables
Code (text)
…

OS loads these into different **segments** for a variety of (mostly security-related) reasons

Kernel memory

Your OS uses it


4GB
Kernel space
Virtual memory reserved for the kernel usage.
3GB
Random offset
Stack
rw-
Local variables
int tries = 10;
Random offset
mmap segments
File mappings (including dynamic libraries)
Anonymous mappings
/lib/libc.so
Heap
rw-
Small memory chunks
char *path = malloc(256);
Random offset
BSS segment
rw-
Uninitialized static variables.
static char *fullname;
Data segment
r--
Initialized static variables.
static char *hello = "Hello, world!";
Text segment
r-x
ELF header and code of the process.
int main() { return printf(hello); }
0x804800
0

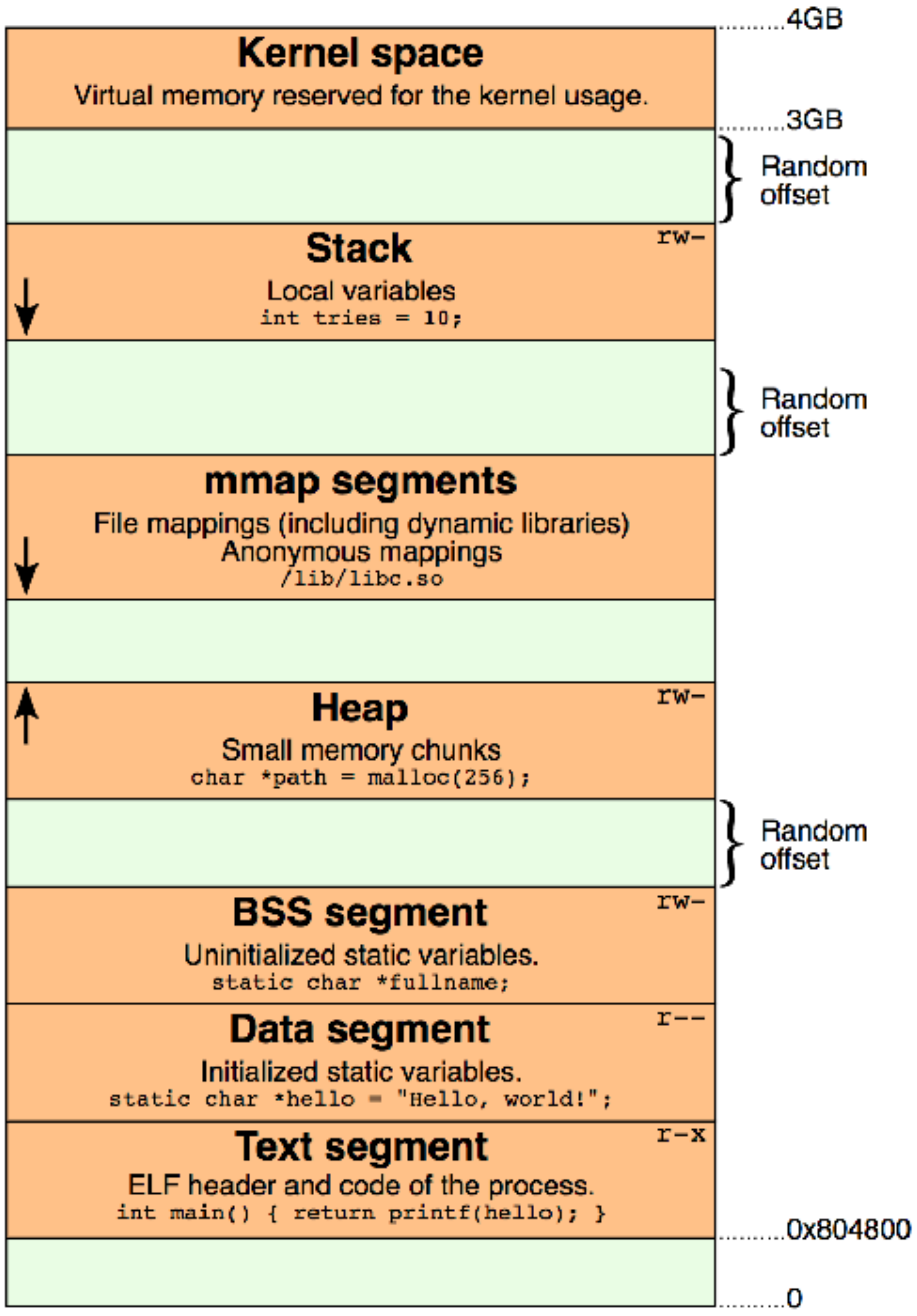


4GB
Kernel space
Virtual memory reserved for the kernel usage.
3GB
Random offset
Stack
rw-
Local variables
int tries = 10;
Random offset
mmap segments
File mappings (including dynamic libraries)
Anonymous mappings
/lib/libc.so
Heap
rw-
Small memory chunks
char *path = malloc(256);
Random offset
BSS segment
rw-
Uninitialized static variables.
static char *fullname;
Data segment
r--
Initialized static variables.
static char *hello = "Hello, world!";
Text segment
r-x
ELF header and code of the process.
int main() { return printf(hello); }
0x804800
0

Stack: push / pop

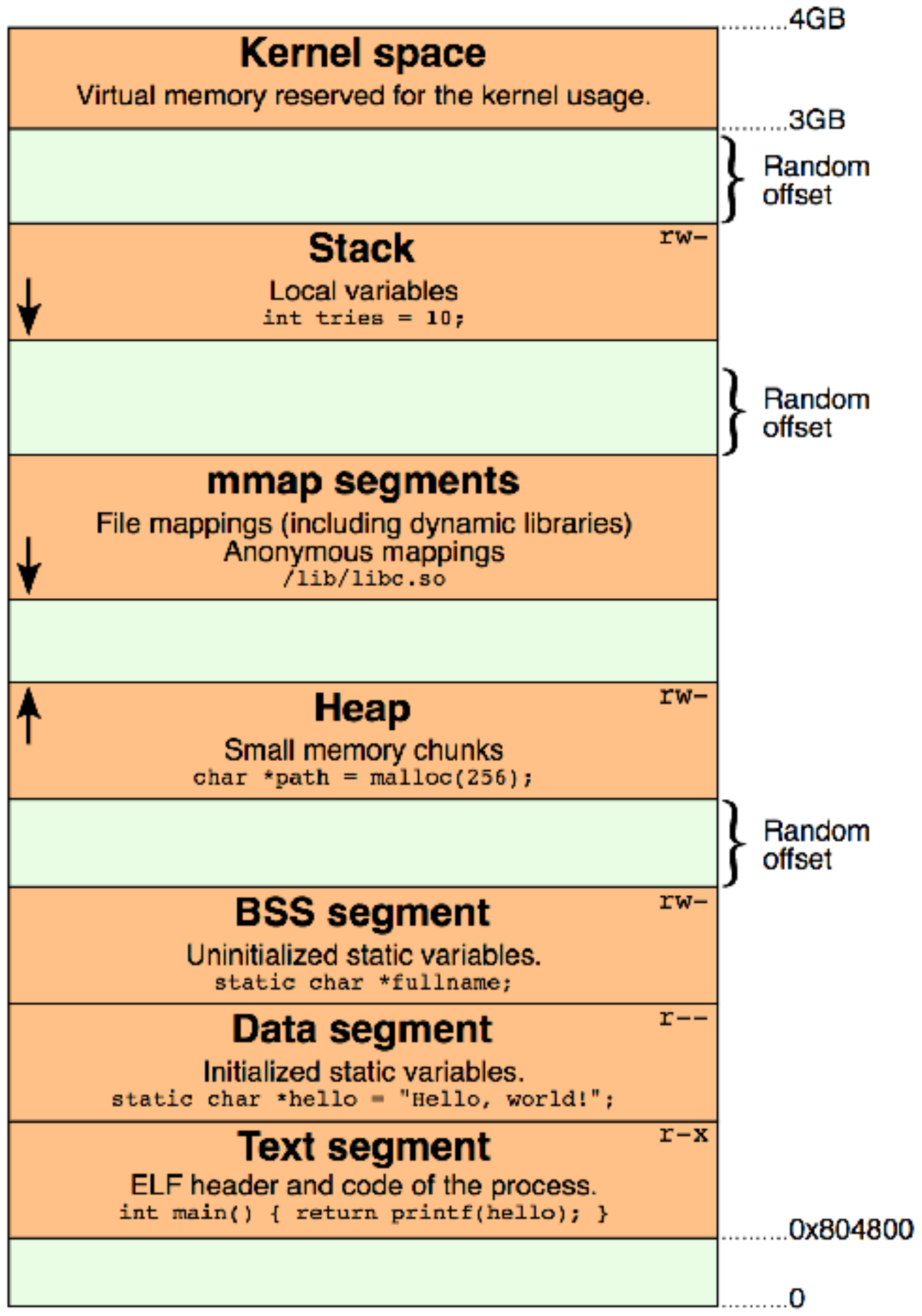
Very important:

The stack grows **down**


4GB
Kernel space
Virtual memory reserved for the kernel usage.
3GB
Random offset
Stack
rw-
Local variables
int tries = 10;
Random offset
mmap segments
File mappings (including dynamic libraries)
Anonymous mappings
/lib/libc.so
Heap
rw-
Small memory chunks
char *path = malloc(256);
Random offset
BSS segment
rw-
Uninitialized static variables.
static char *fullname;
Data segment
r--
Initialized static variables.
static char *hello = "Hello, world!";
Text segment
r-x
ELF header and code of the process.
int main() { return printf(hello); }
0x804800
0

mmap segments

Allows you to **map** a file to memory

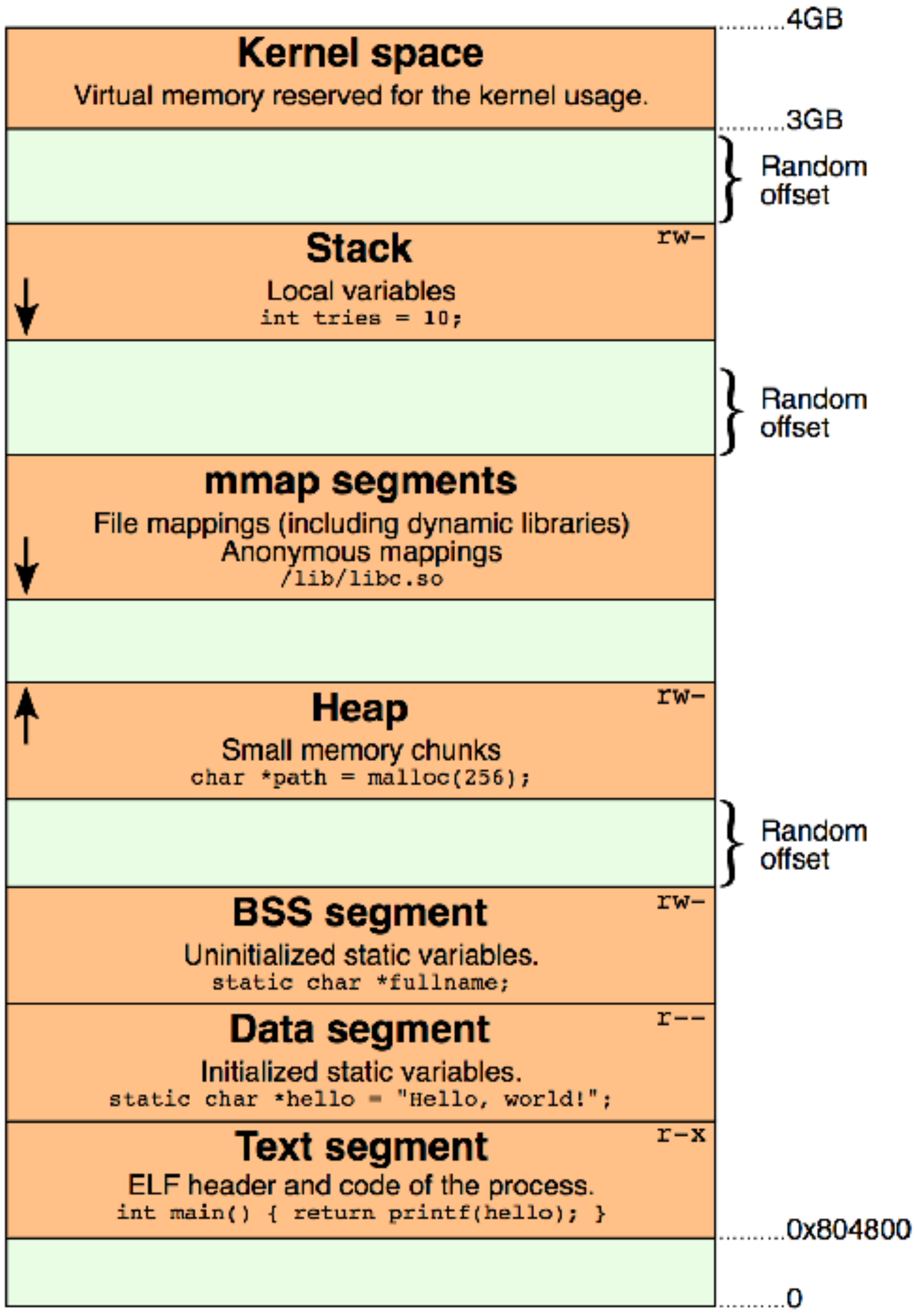


4GB
Kernel space
Virtual memory reserved for the kernel usage.
3GB
Random offset
Stack
rw-
Local variables
int tries = 10;
Random offset
mmap segments
File mappings (including dynamic libraries)
Anonymous mappings
/lib/libc.so
Heap
rw-
Small memory chunks
char *path = malloc(256);
Random offset
BSS segment
rw-
Uninitialized static variables.
static char *fullname;
Data segment
r--
Initialized static variables.
static char *hello = "Hello, world!";
Text segment
r-x
ELF header and code of the process.
int main() { return printf(hello); }
0x804800
0

Heap: dynamic allocation

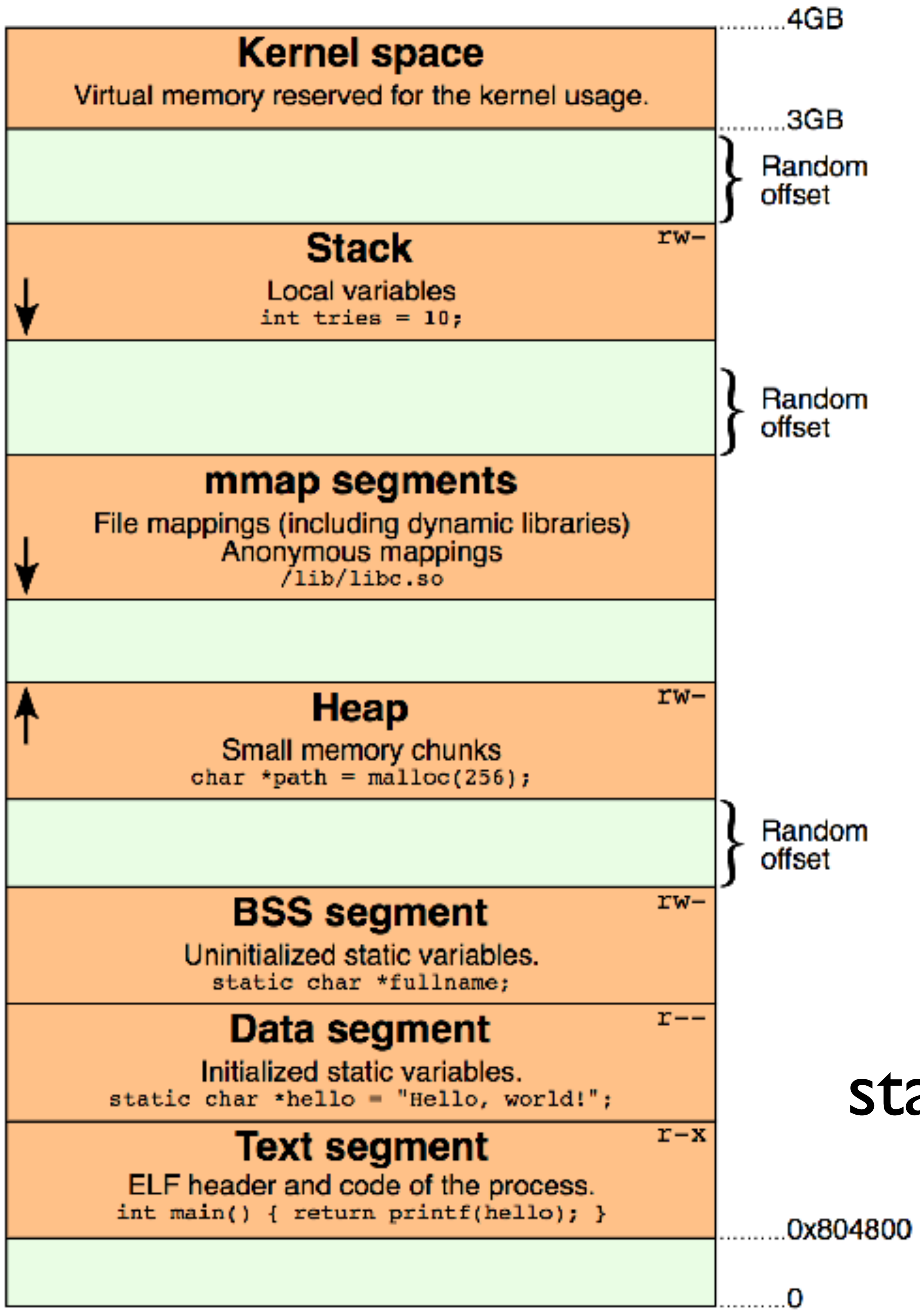
C++: New / delete

C: Malloc / free

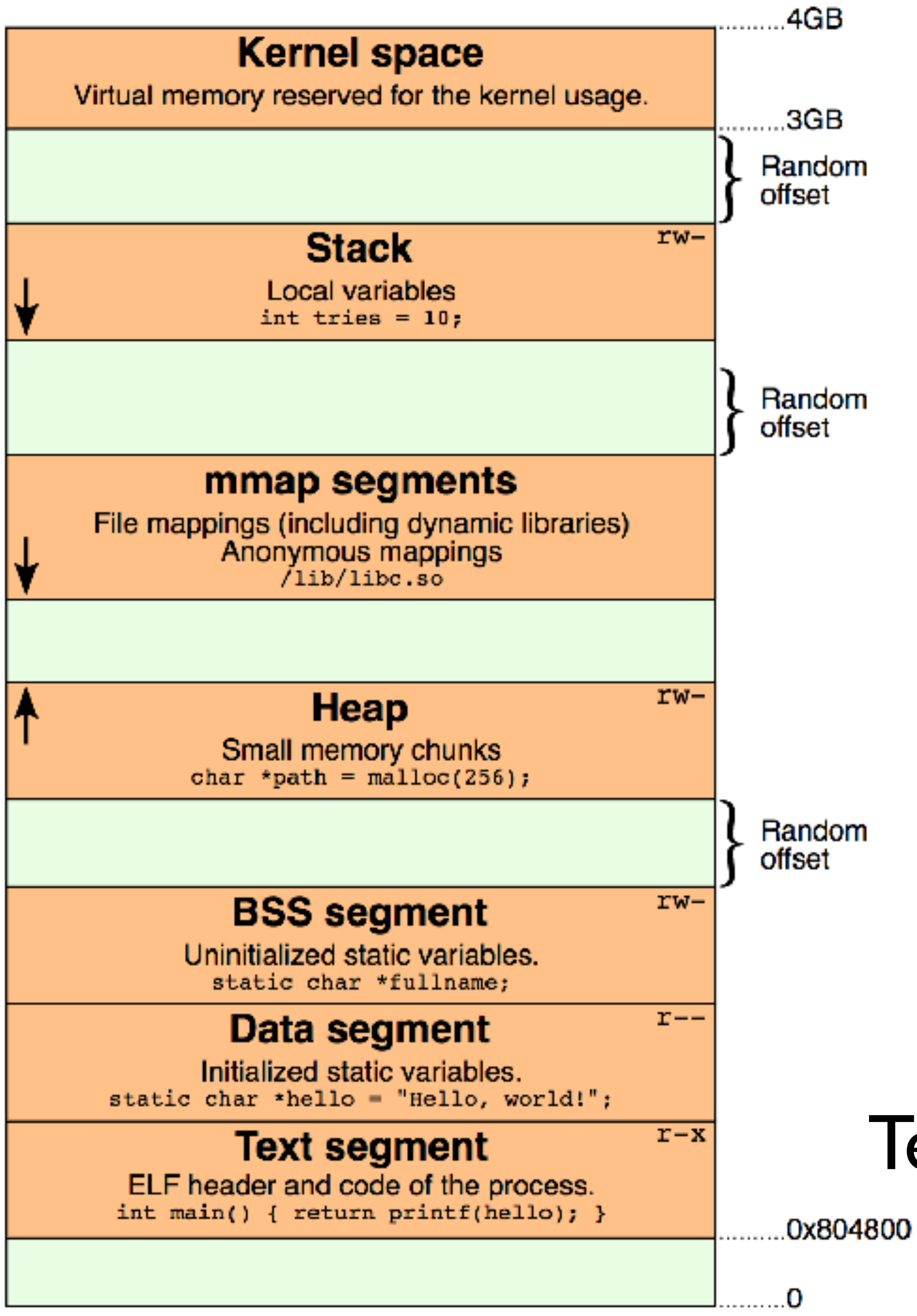


4GB
Kernel space
Virtual memory reserved for the kernel usage.
3GB
Random offset
Stack
rw-
Local variables
int tries = 10;
Random offset
mmap segments
File mappings (including dynamic libraries)
Anonymous mappings
/lib/libc.so
Heap
rw-
Small memory chunks
char *path = malloc(256);
Random offset
BSS segment
rw-
Uninitialized static variables.
static char *fullname;
Data segment
r--
Initialized static variables.
static char *hello = "Hello, world!";
Text segment
r-x
ELF header and code of the process.
int main() { return printf(hello); }
0x804800
0

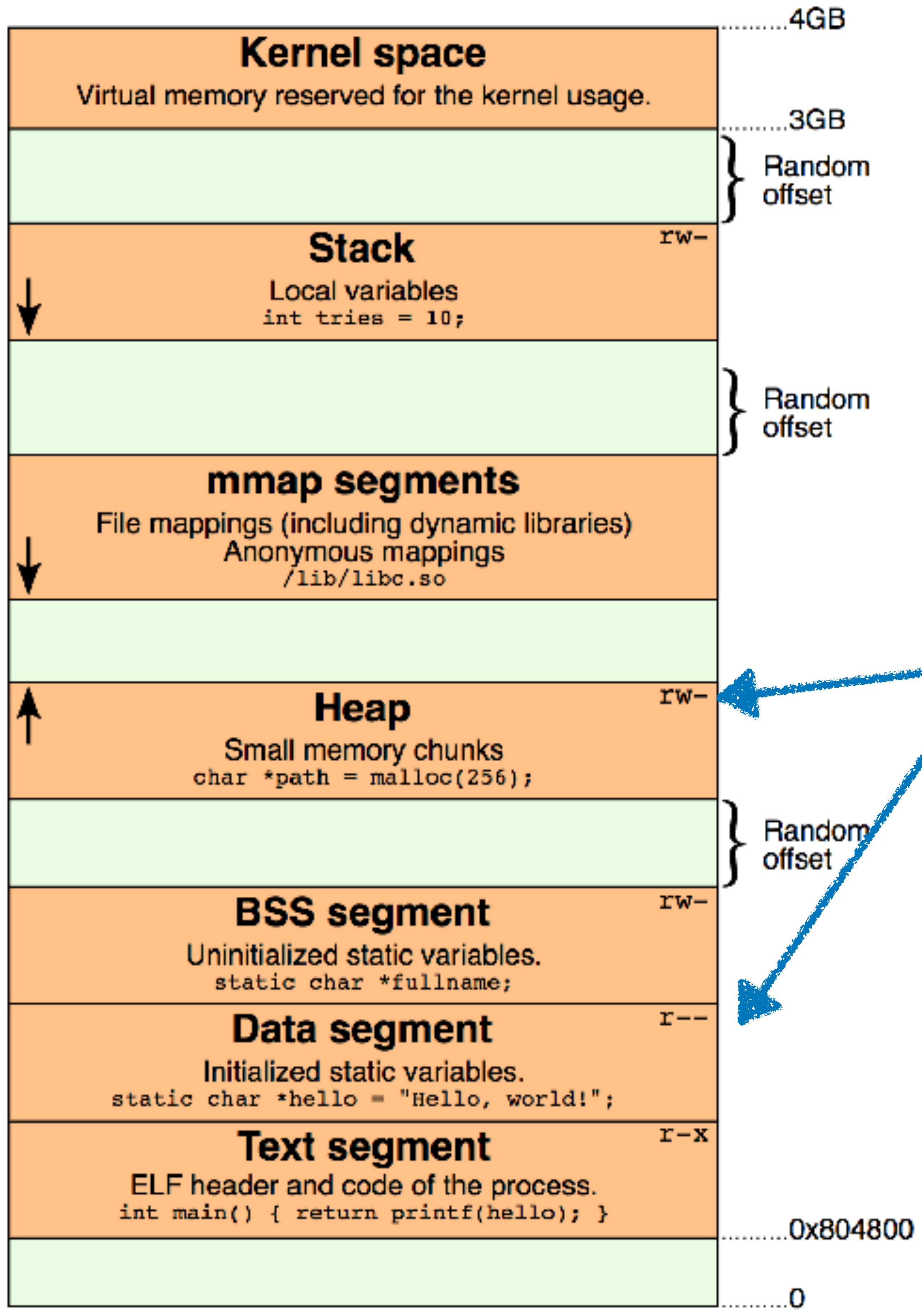
BSS: Uninitialized static vars (globals)


4GB
Kernel space
Virtual memory reserved for the kernel usage.
3GB
Random offset
Stack
rw-
Local variables
int tries = 10;
Random offset
mmap segments
File mappings (including dynamic libraries)
Anonymous mappings
/lib/libc.so
Heap
rw-
Small memory chunks
char *path = malloc(256);
Random offset
BSS segment
rw-
Uninitialized static variables.
static char *fullname;
Data segment
r--
Initialized static variables.
static char *hello = "Hello, world!";
Text segment
r-x
ELF header and code of the process.
int main() { return printf(hello); }
0x804800
0

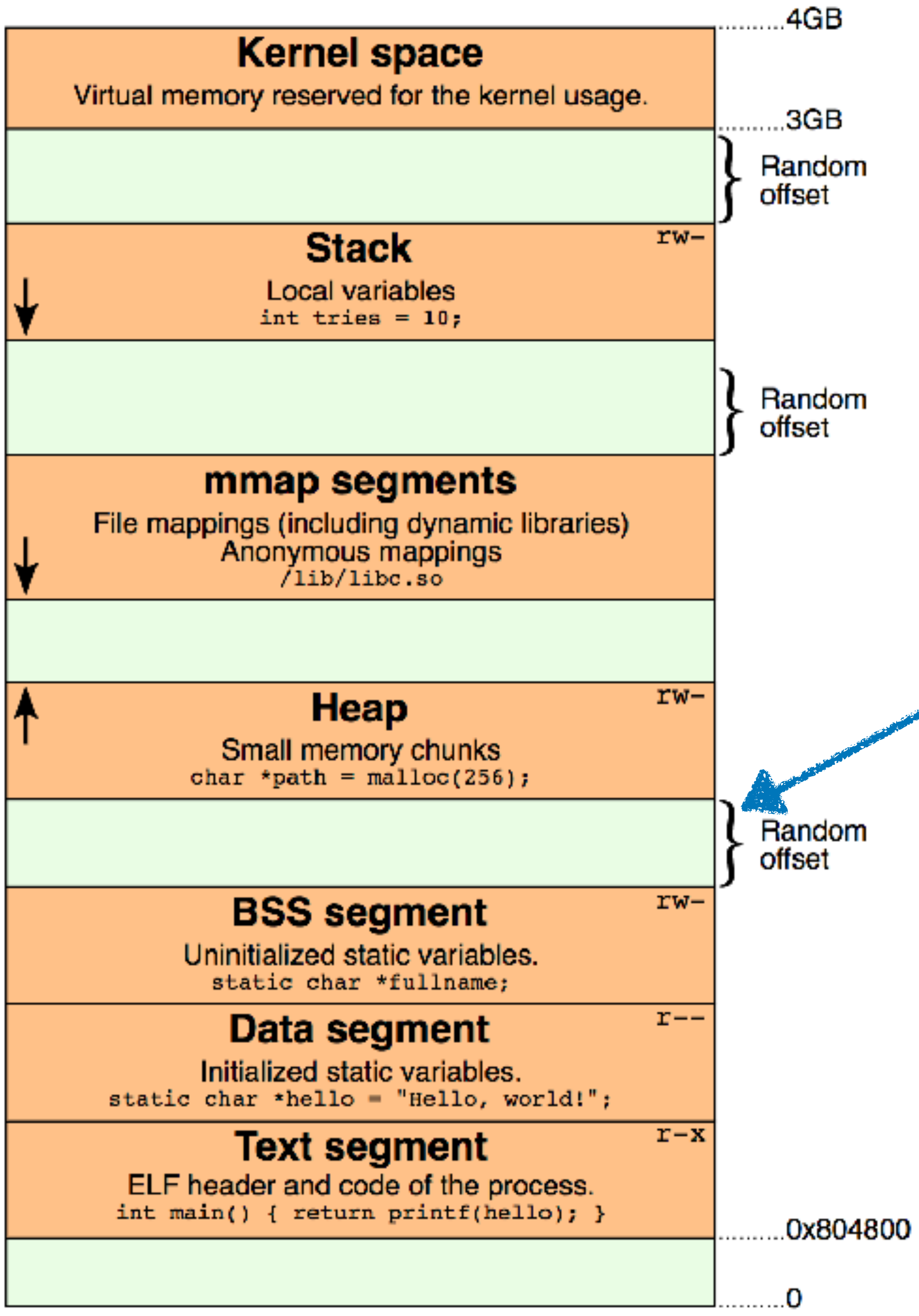
Data segment: initialized statics—e.g., constant strings


4GB
Kernel space
Virtual memory reserved for the kernel usage.
3GB
Random offset
Stack
rw-
Local variables
int tries = 10;
Random offset
mmap segments
File mappings (including dynamic libraries)
Anonymous mappings
/lib/libc.so
Heap
rw-
Small memory chunks
char *path = malloc(256);
Random offset
BSS segment
rw-
Uninitialized static variables.
static char *fullname;
Data segment
r--
Initialized static variables.
static char *hello = "Hello, world!";
Text segment
r-x
ELF header and code of the process.
int main() { return printf(hello); }
0x804800
0

Text segment: program code


4GB
Kernel space
Virtual memory reserved for the kernel usage.
3GB
Random offset
Stack
rw-
Local variables
int tries = 10;
Random offset
mmap segments
File mappings (including dynamic libraries)
Anonymous mappings
/lib/libc.so
Heap
rw-
Small memory chunks
char *path = malloc(256);
Random offset
BSS segment
rw-
Uninitialized static variables.
static char *fullname;
Data segment
r--
Initialized static variables.
static char *hello = "Hello, world!";
Text segment
r-x
ELF header and code of the process.
int main() { return printf(hello); }
0x804800
0

Note the **permissions**


4GB
Kernel space
Virtual memory reserved for the kernel usage.
3GB
Random offset
Stack
rw-
Local variables
int tries = 10;
Random offset
mmap segments
File mappings (including dynamic libraries)
Anonymous mappings
/lib/libc.so
Heap
rw-
Small memory chunks
char *path = malloc(256);
Random offset
BSS segment
rw-
Uninitialized static variables.
static char *fullname;
Data segment
r--
Initialized static variables.
static char *hello = "Hello, world!";
Text segment
r-x
ELF header and code of the process.
int main() { return printf(hello); }
0x804800
0

This **random offset** really security feature

Calling conventions

Touch-tone phones, send an acoustic wave over the wire

If Alice wants to call Bob, her phone needs to send the right sounds over the wire in the right order

Calling conventions

When function A wants to call function B, it has to do the same

- Where do arguments go?
- How to store return address?
- Who saves registers?
- Where is result stored?

Calling conventions

Modern computers use a few **different** calling conventions

De-facto standard (Linux / MacOS / etc..) : **x86-64 System V ABI**

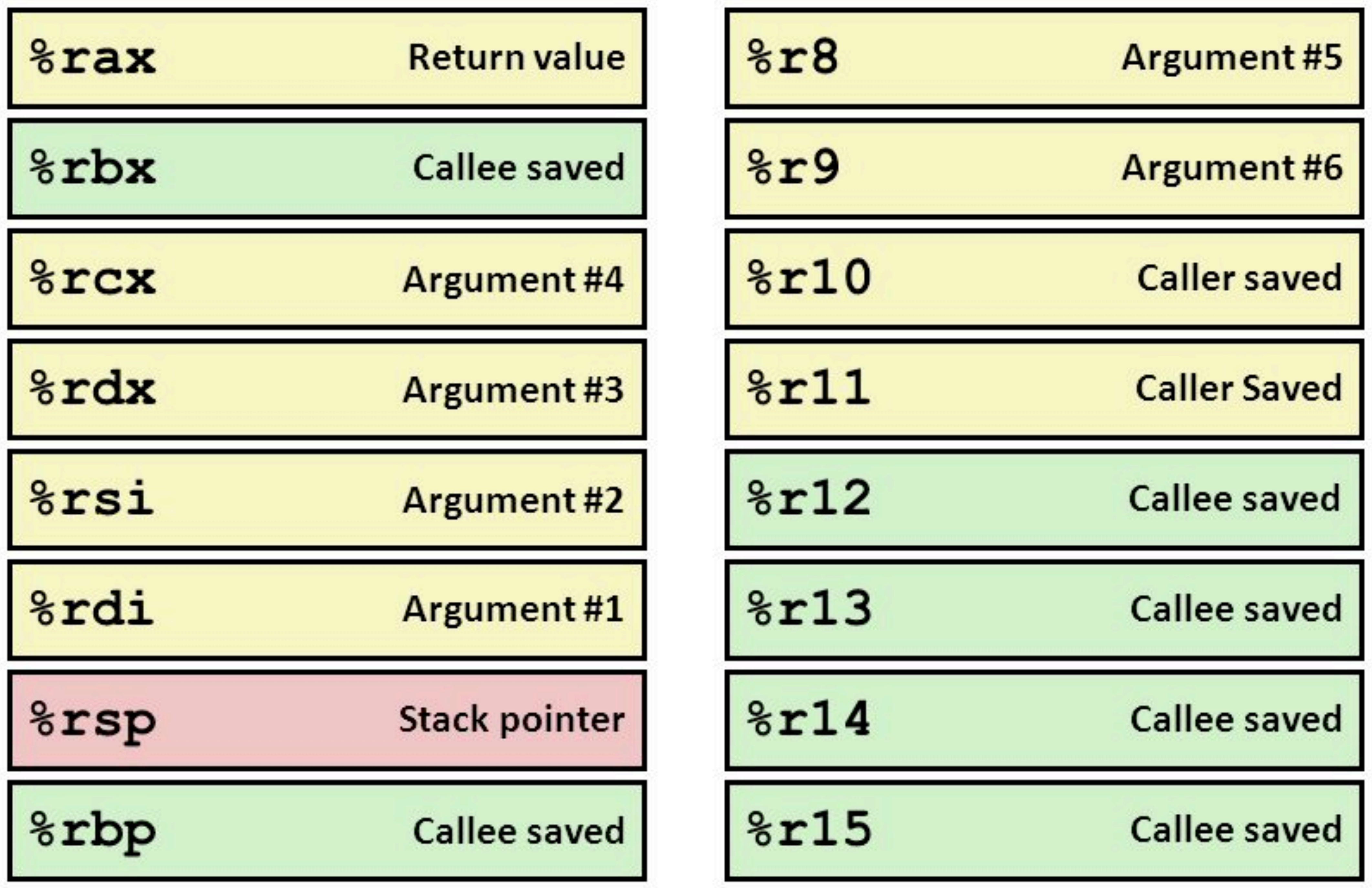
- Where do arguments go?
- How to store return address?
- Who saves registers?
- Where is result stored?

Note: this is **new** for the 64 bit ABI. You might see stuff online for the 32-bit ABI that is **different**

Calling conventions: x86-64 System V ABI

- Where do arguments go?
  - First six: rdi,rsi,rdx,rcx,r8,r9
- How to store return address?
  - `call` instruction puts on top of stack
- Who saves registers?
  - Caller saves caller-save registers
    - R10,R11, any ones used for args
- Where is result stored?
  - Result stored in %rax

x86-64 Integer Registers: Usage Conventions



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

http://slideplayer.com/slide/9679824/

x86-64 System V ABI

Rules for **caller**:

- Save caller-save registers
- First six args in registers, after that put on stack
- Execute `call`—pushes ret addr

Afterwards:

- Pop saved registers
- Result now in %rax

x86-64 System V ABI

Rules for **callee**:

- First six args available in registers
- Push %rbp—caller's base pointer
- Move %rsp to %rbp—Setup new frame
- Subtract necessary stack space
- Push callee-save registers
- Before exit: restore rbp/callee-saved regs
  - `leave` instruction restores rbp
- When function done, put result in %rax
- Use `ret` instruction to pop return rip

These rules are cumbersome: I frequently look them up, they change depending on the kind of function you're calling, etc…

Upshot: don't feel you have to memorize, just get the gist / know how to recognize them

Small examples: interactive demo of x86-64 ABI

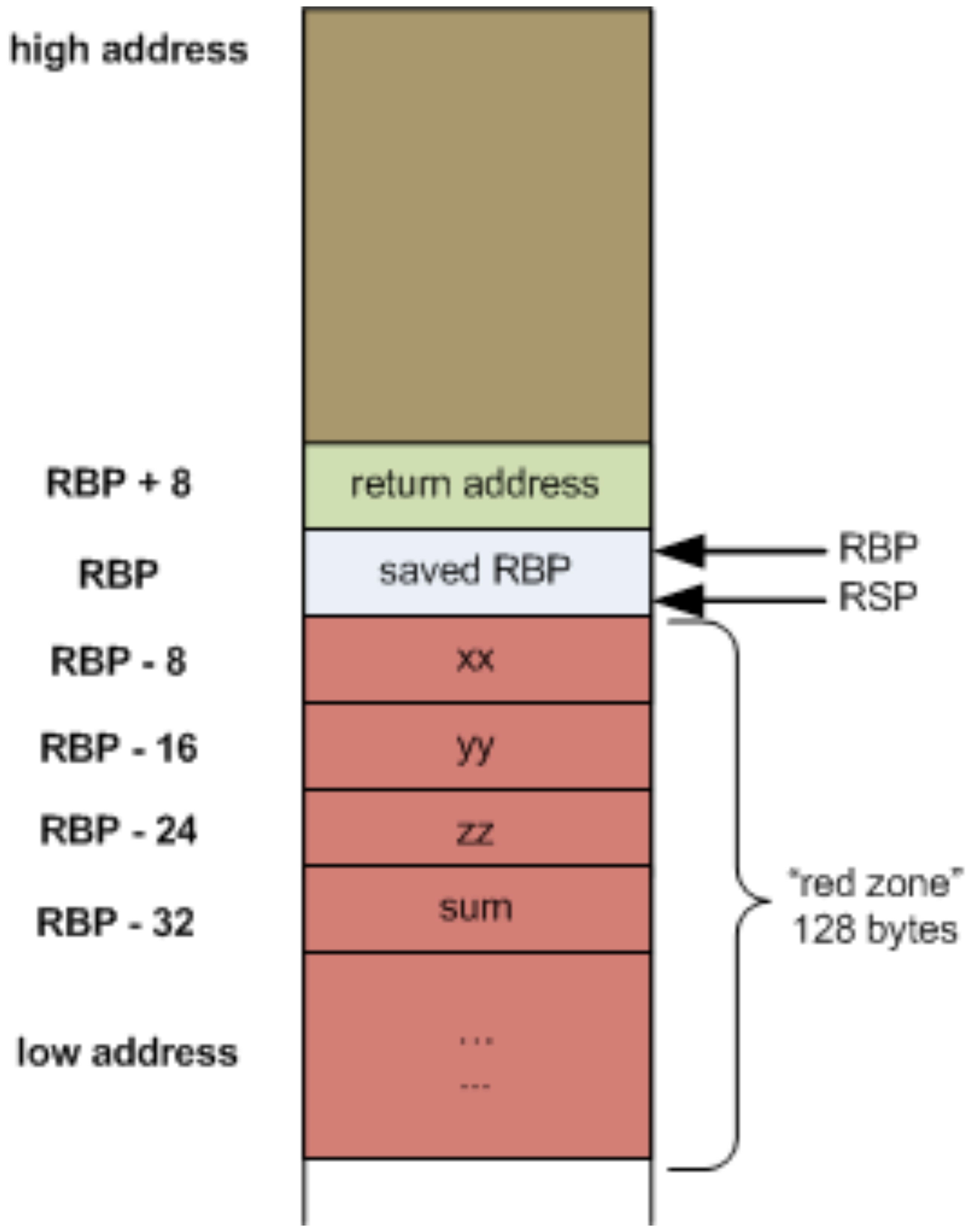
Trivia: the red zone

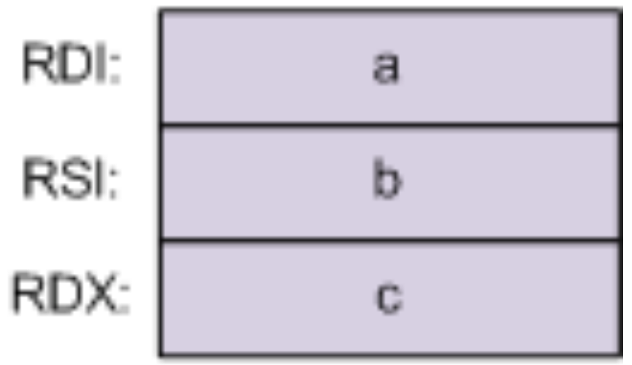
```
int bar(int a, int b) {
  return a + b;
}
```

```
bar:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    ret
```

Weird! This code using -4(%rbp) before decrementing the stack pointer!!

Turns out: x86-64 **guarantees** there are always 128 bytes below %rsp





Upshot: if a function uses at most 128 bytes below RSP, doesn't have to subtract anything from RSP

This is an optimization for "small" functions: so they never have to subtract from RSP

```
int foo(int x) {
    return bar(x+2);
}

int bar(int y) {
  return y*3;
}

int main(int argc, char **argv) {
    return foo(stoi(argv[0]));
}
```

```
foo:  # @foo
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     dword ptr [rbp - 4], edi
        mov     eax, dword ptr [rbp - 4]
        add     eax, 2
        mov     edi, eax
        call    bar
        add     rsp, 16
        pop     rbp
        ret
bar:   # @bar
        push    rbp
        mov     rbp, rsp
        mov     dword ptr [rbp - 4], edi
        imul    eax, dword ptr [rbp - 4], 3
        pop     rbp
        ret
```

```
main:   # @main
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     dword ptr [rbp - 4], 0
        mov     dword ptr [rbp - 8], edi
        mov     qword ptr [rbp - 16], rsi
        mov     rax, qword ptr [rbp - 16]
        mov     rdi, qword ptr [rax]
        mov     al, 0
        call    stoi
        mov     edi, eax
        call    foo
        add     rsp, 16
        pop     rbp
        ret
```

Note: this assembly is destination-first

```
int foo(int x) {
    return bar(x+2);
}

int bar(int y) {
  return y*3;
}

int main(int argc, char **argv) {
    return foo(stoi(argv[0]));
}
```

If we turn on compiler flag -O1 (optimization-level-1) we get *much better* code

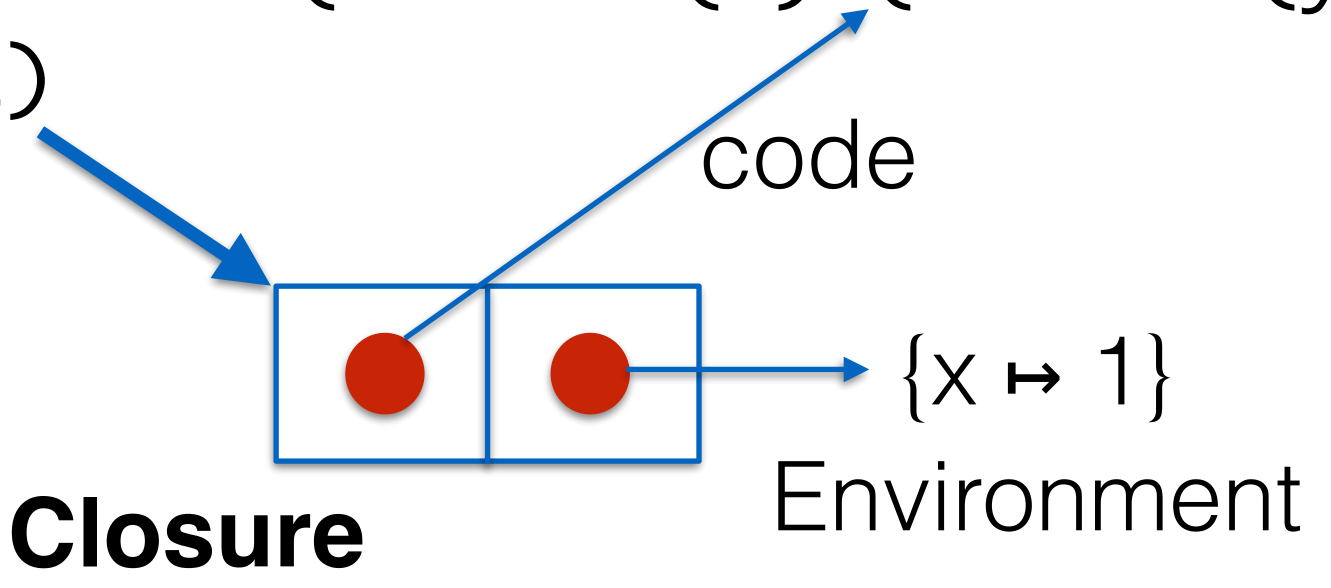
- calls to foo **and** bar are tail calls, so just turn into jumps!

```
foo:                                    # @foo
        add     edi, 2
        jmp     bar                     # TAILCALL
bar:                                    # @bar
        lea     eax, [rdi + 2*rdi]
        ret
main:                                   # @main
        push    rax
        mov     rdi, qword ptr [rsi]
        xor     eax, eax
        call    stoi
        mov     edi, eax
        pop     rax
        jmp     foo                     # TAILCALL
```

Now, back to closure representation…

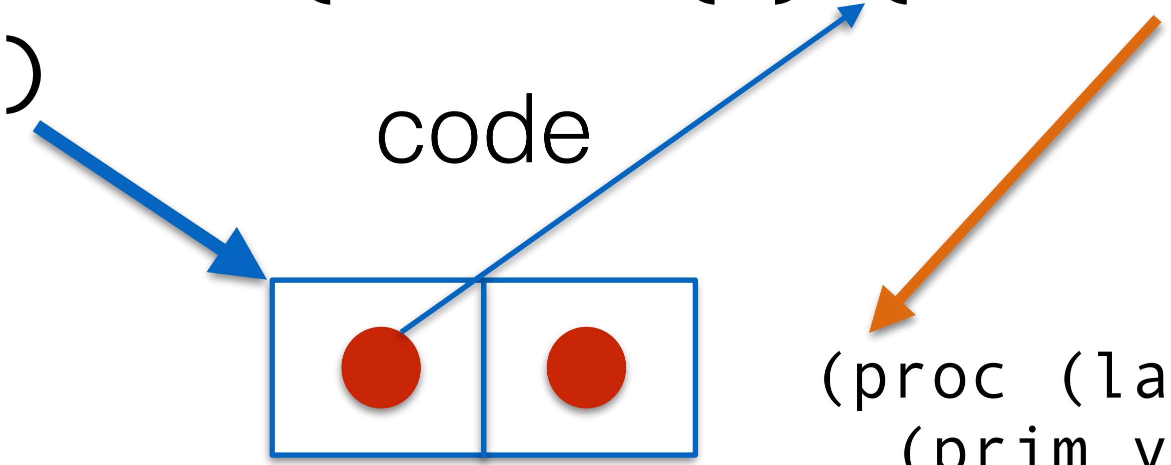
Of course closures are code + data

```
(define f (lambda (x) (lambda (y) x)))
(f 1)
```



In last lecture, we said we would use **vectors** to represent closures as a sequence of (a) the name of a lambda and (b) values of its free variables in a known canonical order

```
          lam12          lam14
(define f (lambda (x) (lambda (y) x)))
(f 1)
```



```
(proc (lam14 env0 y)
  (prim vector-ref env0 1))
```

At runtime…

```
(vector
      lam14 ;; compiled as proc lam14
      1)    ;; remember x is offset 1
```

Then, each of these top-level procs can be compiled **directly** to C (or a C-like function)

```
(proc (lam14 env0 y)
  (prim vector-ref env0 1))
```

```
typedef unsigned long long value;
typedef value* value_vec;
typedef value_vec closure;

value lam14(closure env0, value y) {
    return env0[1];
}
```

There is a direct translation from the set of procs to C-style functions, which then translate readily to assembly…

```
lam14:    # @lam14
        mov     rax, qword ptr [rdi + 8]
        ret
```

(Of course, as our IR uses CPS, all of our returns will simply be function calls)

Bottom-up closure conversion

- As the AST is traversed, free variables are computed.
- At each lambda: 1) the algorithm converts any lambdas under the lambda’s body first (and also computes a set of free variables); then 2) it emits code to allocate the lambda’s closure/environment and replaces free vars with env access.
- Converting the body of a lambda yields a set of free variables that can be *canonically ordered*.
- Closures are *flat* heap-allocated vectors containing a function pointer and then each free var *in order*.
- Accesses of free variables are turned into a `vector-ref` with the predetermined index.

```
…
  (λ (a)
      …

          (λ (b c)
              …

                   (λ (d)
                       …
free vars: a , c , f

                          (f a c)

 …) …) …) …
```

```
…
   (λ (a)
        …
```

```
          (λ (b c)
               …
```

```
(proc (lam14 env d)
  …
  (clo-app
   (prim vector-ref
         env ‘3)    ;f
   (prim vector-ref
         env ‘1)    ;a
   (prim vector-ref
         env ‘2))   ;c
    …)
```

adds first-order proc

```
(prim vector
      lam14
      a
      c
      f)
```

allocates flat closure

```
…) …) …) …
```

```
…
  (λ (a)
     …


         (λ (b c)
            …
```

free vars: $\underbrace{\textbf{a f}}$ **x y**

```
                (prim vector
                      lam14
                      a
                      c
                      f)
```

references at closure allocation can remain free

```
 …)  …)  …)  …
```

```
(clo-app
   (prim vector-ref
         env ‘3)    ;f
   (prim vector-ref
         env ‘1)    ;a
   (prim vector-ref
         env ‘2))   ;c
```

$$\downarrow$$

```
(let ([f-clo (prim vector-ref env ‘3)])
  (let ([f-ptr (prim vector-ref f-clo ‘0)])
    (let ([a (prim vector-ref env ‘1)])
      (let ([c (prim vector-ref env ‘2)])
        (C-style-call f-ptr f-clo a c)))))
```

application: 1) function pointer is accessed from closure
2) closure (`f-clo`) is passed to invoked function ptr

Let’s look at an example that uses invocation…

Consider the following example from the lambda calculus…

```
((lambda (a) (a a))
 (lambda (b) b))
```

Let’s look at an example that uses invocation…

Consider the following example from the lambda calculus…

```
((lambda (a) (a a))
 (lambda (b) b))
```

What would this look like post-CPS-conversion…?

Every lambda gets an extra current-continuation argument, and the (special) `halt` continuation is passed through (at runtime, `k0`/`k1` are `halt`)

The “final” return is within (lambda (k1) …) and returns b to the halt continuation (signaling the end).

Let’s look at an example that uses invocation…

Consider the following example from the lambda calculus…

```
((lambda (a) (a a))
 (lambda (b) b))
```

What would this look like post-CPS-conversion…?

```
((lambda (k0 a) (a k0 a))
 halt
 (lambda (k1 b) (k1 b ‘0)))
```

Every lambda gets an extra current-continuation argument, and the (special) `halt` continuation is passed through (at runtime, `k0`/`k1` are `halt`)

The “final” return is within (lambda (k1) …) and returns b to the halt continuation (signaling the end).

What will closure conversion then produce…?

```
      lam5
           ((lambda (k0 a) (a k0 a))
            halt
     lam16 (lambda (k1 b) (k1 b ‘0)))
```

```
(proc (lam5 env0 k0 a)  (proc (lam16 env1 k1 b)
  (clo-app a k0 a))       (clo-app k1 b ‘0))
```

Let’s live code bottom-up closure conversion.

```
; Input Language (simplified cps):
; e ::= (let ([x 'dat]) e)
;     | (let ([x (prim op x ...)]) e)
;     | (let ([x (lambda (x ...) e)]) e)
;     | (if x e e)
;     | (x x ...)
```

Our output language is a list of C-style procedures

```
; Output Language (procedural):
; p ::= ((proc (x x ...) e) ...)
; e ::= (let ([x 'dat]) e)
;     | (let ([x (prim op x ...)]) e)
;     | (let ([x (make-closure x x ...)]) e)
;     | (let ([x (env-ref x nat)]) e)
;     | (if x e e)
;     | (clo-app x x ...)
```

```
(define (closure-convert simple-cps)
 ;; Exp x List[Proc] -> Exp x Set[Var] x List[Proc]
 (define (bottom-up e procs)
    (match e
       …))
 ;; returns the top expression, set of free variables
 ;; and list of sub-procs
 (match-define `(,main-body ,free ,procs)
   (bottom-up simple-cps ‘()))
 ;; return the list of procs & main expression proc
 `((proc (main) ,main-body) . ,procs))
```

```
(define (closure-convert simple-cps)
 ;; Exp x List[Proc] -> Exp x Set[Var] x List[Proc]
 (define (bottom-up e procs)
    (match e
      [`(let ([,x ',dat]) ,e0)
       (match-define `(,e0+ ,free+ ,procs+)
                     (bottom-up e0 procs))
       `((let ([,x ',dat]) ,e0+)
         ,(set-remove free+ x)
         ,procs+)]
```

```
(define (closure-convert simple-cps)
 ;; Exp x List[Proc] -> Exp x Set[Var] x List[Proc]
 (define (bottom-up e procs)
    (match e
      …
      [`(let ([,x (prim ,op ,xs ...)]) ,e0)
       (match-define `(,e0+ ,free+ ,procs+)
                     (bottom-up e0 procs))
       `((let ([,x (prim ,op ,@xs)]) ,e0+)
         ,(set-remove (set-union free+ (list->set xs)) x)
         ,procs+)]
```

```
(define (closure-convert simple-cps)
 ;; Exp x List[Proc] -> Exp x Set[Var] x List[Proc]
 (define (bottom-up e procs)
    (match e
      …
      [`(let ([,x (lambda (,xs ...) ,body)]) ,e0)
       (match-define `(,e0+ ,free0+ ,procs0+)
                     (bottom-up e0 procs))
       (match-define `(,body+ ,freelam+ ,procs1+)
                     (bottom-up body procs0+))
       (define env-vars (foldl (lambda (x fr) (set-remove fr x))
                               freelam+
                               xs))
       (define ordered-env-vars (set->list env-vars))
       (define lamx (gensym 'lam))
       (define envx (gensym 'env))
       (define body++ (cdr (foldl (lambda (x count+body)
                                    (match-define (cons cnt bdy) count+body)
                                     (cons (+ 1 cnt)
                                           `(let ([,x (env-ref ,envx ,cnt)])
                                              ,bdy)))
                                  (cons 1 body+)
                                  ordered-env-vars)))
       `((let ([,x (make-closure ,lamx ,@ordered-env-vars)]) ,e0+)
         ,(set-remove (set-union free0+ env-vars) x)
         ((proc (,lamx ,envx ,@xs) ,body++) . ,procs1+))]
```

```
(define (closure-convert simple-cps)
 ;; Exp x List[Proc] -> Exp x Set[Var] x List[Proc]
 (define (bottom-up e procs)
    (match e
      …
      [`(if ,(? symbol? x) ,e0 ,e1)
       (match-define `(,e0+ ,free0+ ,procs0+)
                     (bottom-up e0 procs))
       (match-define `(,e1+ ,free1+ ,procs1+)
                     (bottom-up e1 procs0+))
       `((if ,x ,e0+ ,e1+)
         ,(set-union free1+ free0+ (set x))
         ,procs1+)]
```

```
(define (closure-convert simple-cps)
 ;; Exp x List[Proc] -> Exp x Set[Var] x List[Proc]
 (define (bottom-up e procs)
    (match e
      …
      [`(,(? symbol? xs) ...)
       `((clo-app ,@xs)
         ,(list->set xs)
         ,procs)])
```