Selfie: Introduction to the Implementation of Programming Languages, Operating Systems, and Processor Architecture

Christoph Kirsch and Sara Seidl
Department of Computer Sciences
University of Salzburg
2020

selfie.cs.uni-salzburg.at/slides

v 01/31/20

What?

- This presentation is part of a 3-semester introduction to architecture, compilers, and operating systems using the selfie system.

- Selfie is a minimal, fully self-contained 64-bit implementation of a self-compiling C compiler, RISC-V emulator, and RISC-V hypervisor. Selfie can compile, execute, and virtualize itself any number of times!

- Everything needed is implemented in a single, 12-KLOC file of C code. There are no includes and no external libraries. For bootstrapping, a C compiler will do.

- The purpose of selfie is to teach the implementation of programming languages (C subset), operating systems (virtual memory, concurrent processes), and processor architecture (RISC-V subset), all using the same code!

Why?

- Selfie shows, from first principles, how a minimal but still realistic hardware and software stack works.

- Anyone understanding elementary arithmetic and Boolean logic may be able to follow!

- The goal is to see and understand how the semantics of programming languages and the concurrent execution of programs is constructed using nothing but bits.

- The self-referential nature of the construction is an important part of selfie. Seeing how to resolve it establishes a well-founded understanding of basic computer science principles.

How?

- We use the selfie code to explain everything. Students will need to read, understand, and modify the code to follow the course.
- Instead of copying code here we provide clickable links to actual selfie code snippets on github.com
- Standard terminology is introduced with clickable links to wikipedia.org

Course Material

- Website: selfie.cs.uni-salzburg.at
- Slides (draft): selfie.cs.uni-salzburg.at/slides
- Book (draft, outdated): leanpub.com/selfie
- Sources (code, slides, book): github.com/cksystemsteaching/selfie

- Install selfie on your machine or in the cloud using the instructions provided in the selfie repository on github.com

Who?

- Alireza S. Abyaneh
- Sebastian Arming
- Simon Bauer
- joanbm?
- Marcell Haritopoulos
- Christoph M. Kirsch
- Alexander Kollert
- Philipp Mayer
- Christian Mösl
- Hugo Platzer
- Clément Poncelet
- Manuel Rigger
- Sara Seidl
- Mario Strohmeier
- Sebastian Strumegger
- Ana Sokolova
- Tim Ungerhofer
- Dominique Watzl
- Manuel Widmoser

Syllabus

goto INDEX

1. **Programming in C***, the C subset in which selfie is written and compiles.

2. **Introduction to RISC-U**, the RISC-V subset targeted, emulated, and virtualized by selfie.

3. **Introduction to starc**, the selfie compiler (scanner, parser, type checker, register allocator, code generator).

Compilers

4. **Introduction to mipster**, the selfie emulator (virtual and physical memory, machine contexts).

5. **Introduction to hypster**, the selfie hypervisor (virtual memory, context switching).

Operating Systems

6. **Introduction to monster**, the selfie symbolic execution engine (planned).

Introduction to

Selfie and C*

Programming in C*

- C* is a tiny subset of the programming language C

- C* supports only 2 data types: unsigned integer, `uint64_t`, and pointer to unsigned integer, `uint64_t*`. There are no signed integers and no composite data types.

- C* features the unary * operator as the only means to access heap memory hence the name C*. There are no arrays and no structs in C*.

- C* features 5 statements (assignment, if-else, while loop, procedure call, return).

- C* has 3 types of literals (signed decimal number, character, string).

- C* supports 5 arithmetic operators (`+`, `-`, `*`, `/`, `%`) and 6 comparison operators (==, `!=` ,< ,<= ,> , >=). There are no bitwise operators and no Boolean operators.

atoi in C* (proper character and exception handling removed)

./selfie -c

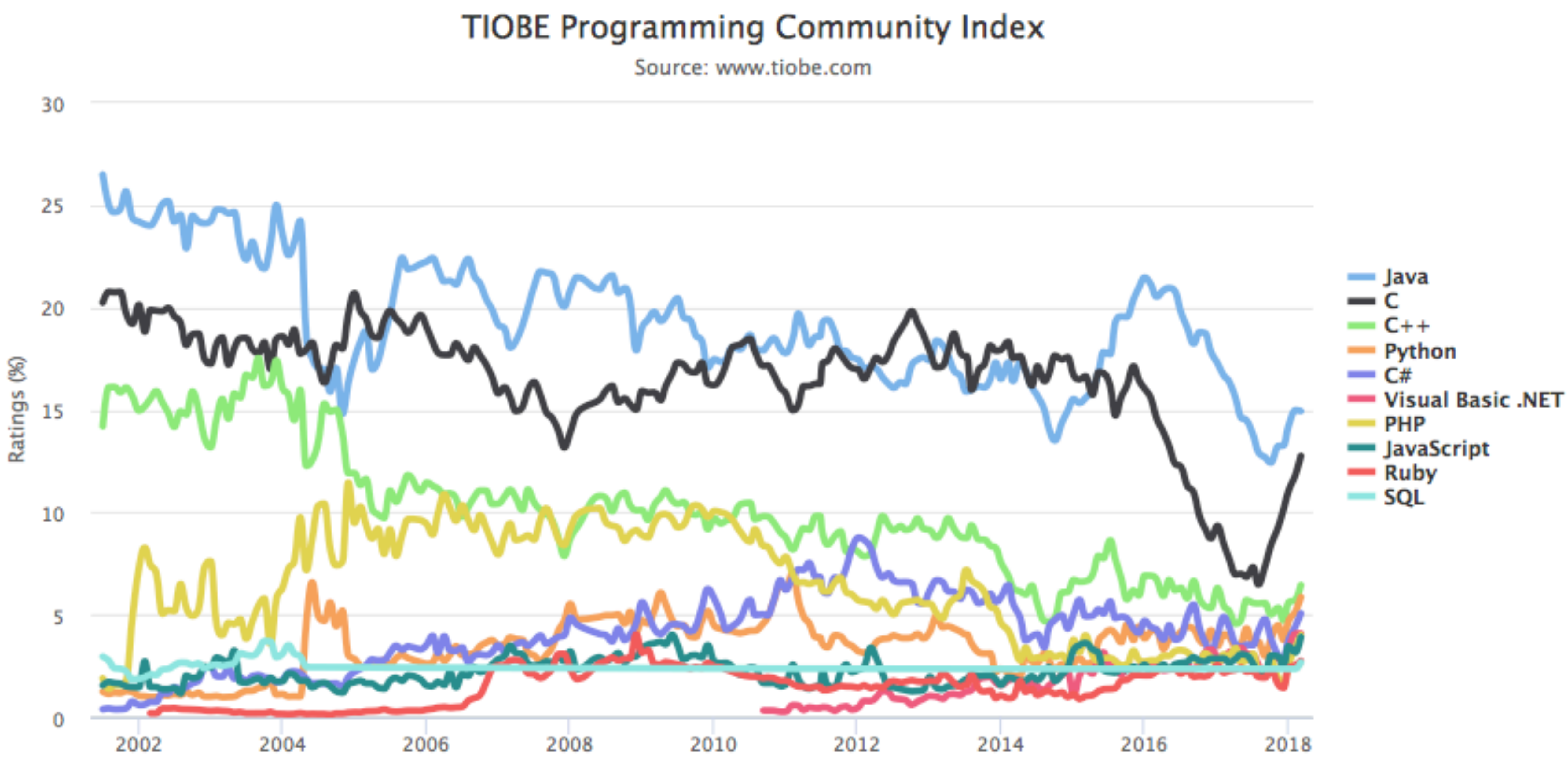
```
uint64_t atoi(uint64_t* s) {
  uint64_t n;

  n = 0;

  // loop until s is terminated
  while (*s != 0) {
    // use base 10, offset by '0'
    n = n * 10 + *s - '0';

    // go to next digit
    s = s + 1;
  }

  return n;
}
```

`atoi` stands for ASCII to integer

Given a string `s` of decimal digits, the `atoi` code computes the numerical value `n` represented by `s`

Click atoi to see the actual code in selfie.
We provide such links throughout the presentation.

Programming Language C



C with all its dialects is still the most popular programming language

https://www.tiobe.com/tiobe-index

Why C?

- This course aims at much more than just learning how to code.
- The choice of programming language is important but not more important than many other choices made here.
- C is relevant, C* is simple.
- With <u>imperative programming</u> the notion of program <u>state</u> maps one-to-one to the notion of machine state.
- Seeing the pitfalls of imperative programming enables students truly appreciate other paradigms such as <u>functional programming</u>.
- In fact, awareness of state makes students understand the value of functional programming. This is nevertheless left to others to teach.

C* by Example

```
uint64_t atoi(uint64_t* s) {
  uint64_t n;

  n = 0;

  // loop until s is terminated
  while (*s != 0) {
    // use base 10, offset by '0'
    n = n * 10 + *s - '0';

    // go to next digit
    s = s + 1;
  }

  return n;
}
```

procedure

pointer type

formal parameter

return type

integer type

local variable

integer literal

dereferencing

assignment

comparison

while loop

expression

integer arithmetic

character literal

pointer arithmetic

return statement

C* Integers and Pointers

- C* integers are unsigned 64-bit integers
- In C* there are five arithmetic operators: `+`, `-`, `*`, `/`, `%`
- And six comparison operators: `==`, `!=`, `<`, `<=`, `>`, `>=`
- C* pointers are 64-bit pointers to C* integers
- And pointer arithmetic: `+`, `-`

C* versus C Integer Literals

- C* integer literals are unsigned 64-bit
- C integer literals are signed 32-bit
- For example, `1/-1==0` in C* but `1/-1==-1` in C
- And, `1%-1==1` in C* but `1%-1==0` in C
- Also, `1<-1` and `1<=-1` hold in C* but `1>-1` and `1>=-1` do not whereas the opposite is true in C
- The semantics of `/` and `%` as well as <, <=, >, and >= is different for signed and unsigned integers!

C* Characters and Strings

- C* characters are <u>ASCII</u>-encoded.
- C* character literals are characters in code like `'c'`.
- C* strings are stored as null-terminated sequences of characters. *Alternatively the end of a string could be identified by storing the number of characters at its beginning.*
- C* string literals are strings in code like `"this"`.
- The difference between `'a'` and `"a"`.

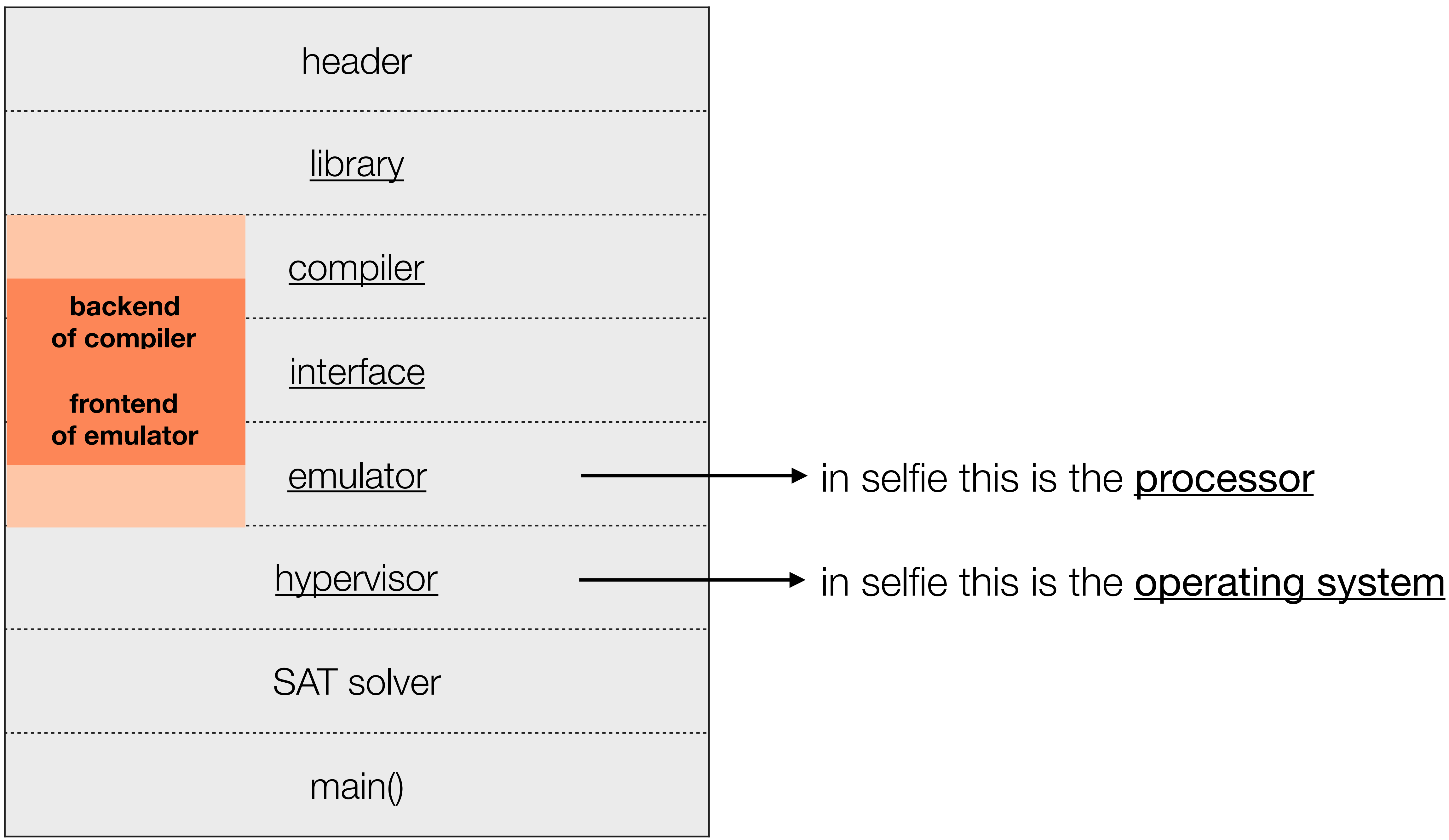
ascii representation (**numerical value**) in memory

pointer to first word of where the string is stored

C* versus C Strings

- C* strings are arrays of unsigned 64-bit integers.
- C strings are arrays of characters, that is, of type `char`.
- For example, `*"Hello World!"` is equal to `0x6F57206F6C6C6548` in C* but `0x48` in C.
- Note that `0x48` is ASCII for `H` while `0x65`, `0x6C`, `0x6F`, `0x20`, and `0x57` are ASCII for `e`, `l`, `o`, `space`, and `W`, respectively.

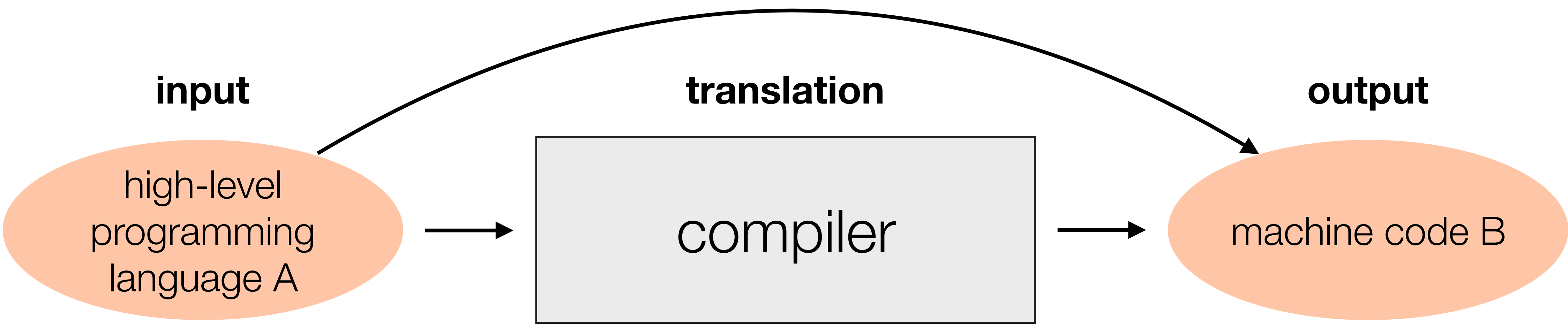
Selfie Code Structure



The Selfie Library

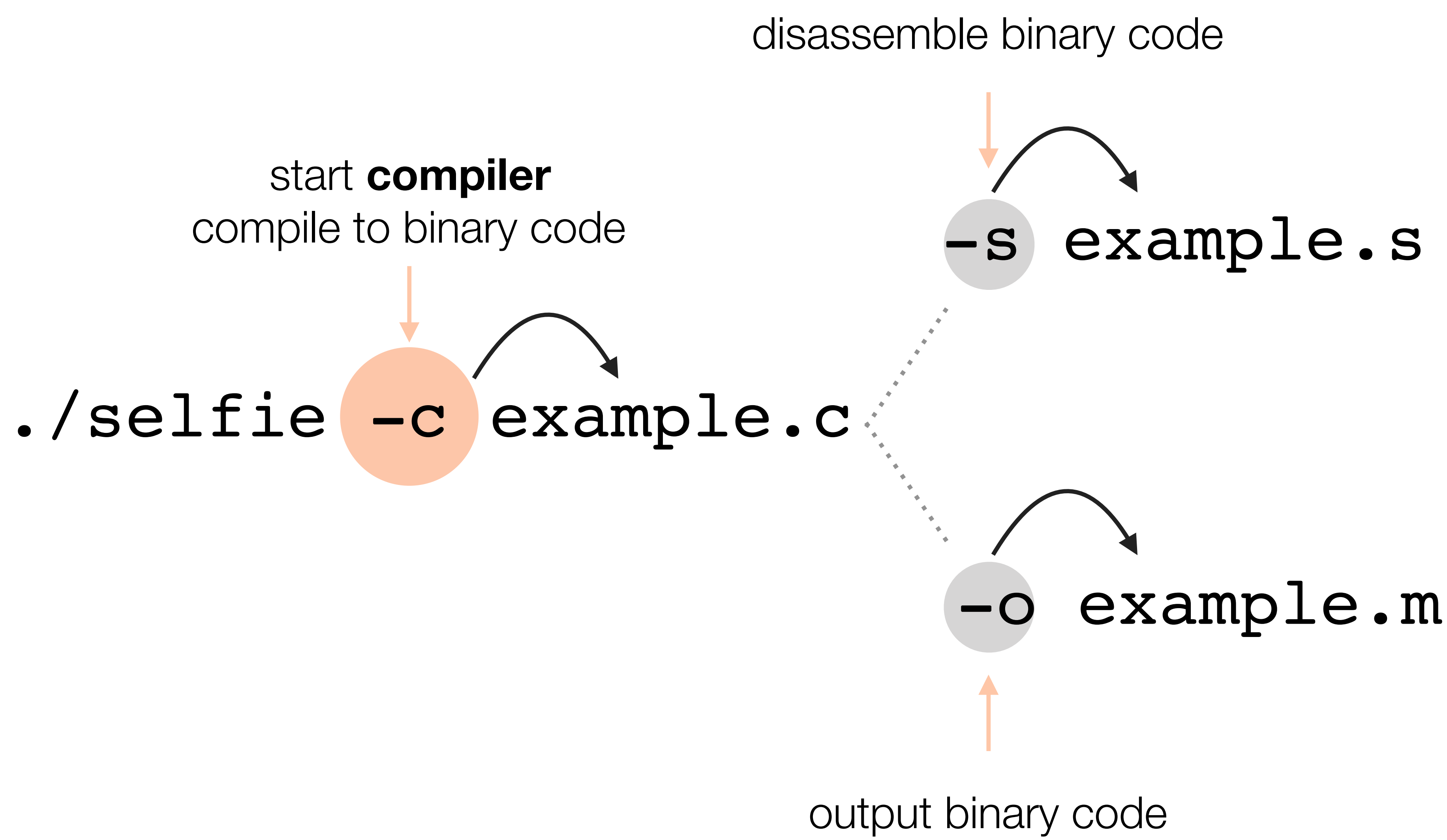
```
uint64_t leftShift(uint64_t n, uint64_t b);
uint64_t rightShift(uint64_t n, uint64_t b);

uint64_t getBits(uint64_t n, uint64_t i, uint64_t b);
uint64_t getLowWord(uint64_t n);
uint64_t getHighWord(uint64_t n);

uint64_t abs(uint64_t n);

uint64_t signedLessThan(uint64_t a, uint64_t b);
uint64_t signedDivision(uint64_t a, uint64_t b);

uint64_t isSignedInteger(uint64_t n, uint64_t b);
uint64_t signExtend(uint64_t n, uint64_t b);
uint64_t signShrink(uint64_t n, uint64_t b);

uint64_t  loadCharacter(uint64_t* s, uint64_t i);
uint64_t* storeCharacter(uint64_t* s, uint64_t i, uint64_t c);

uint64_t stringLength(uint64_t* s);
void     stringReverse(uint64_t* s);
uint64_t stringCompare(uint64_t* s, uint64_t* t);

uint64_t  atoi(uint64_t* s);
uint64_t* itoa(uint64_t n, uint64_t* s, uint64_t b, uint64_t a, uint64_t p);
```

A Compiler

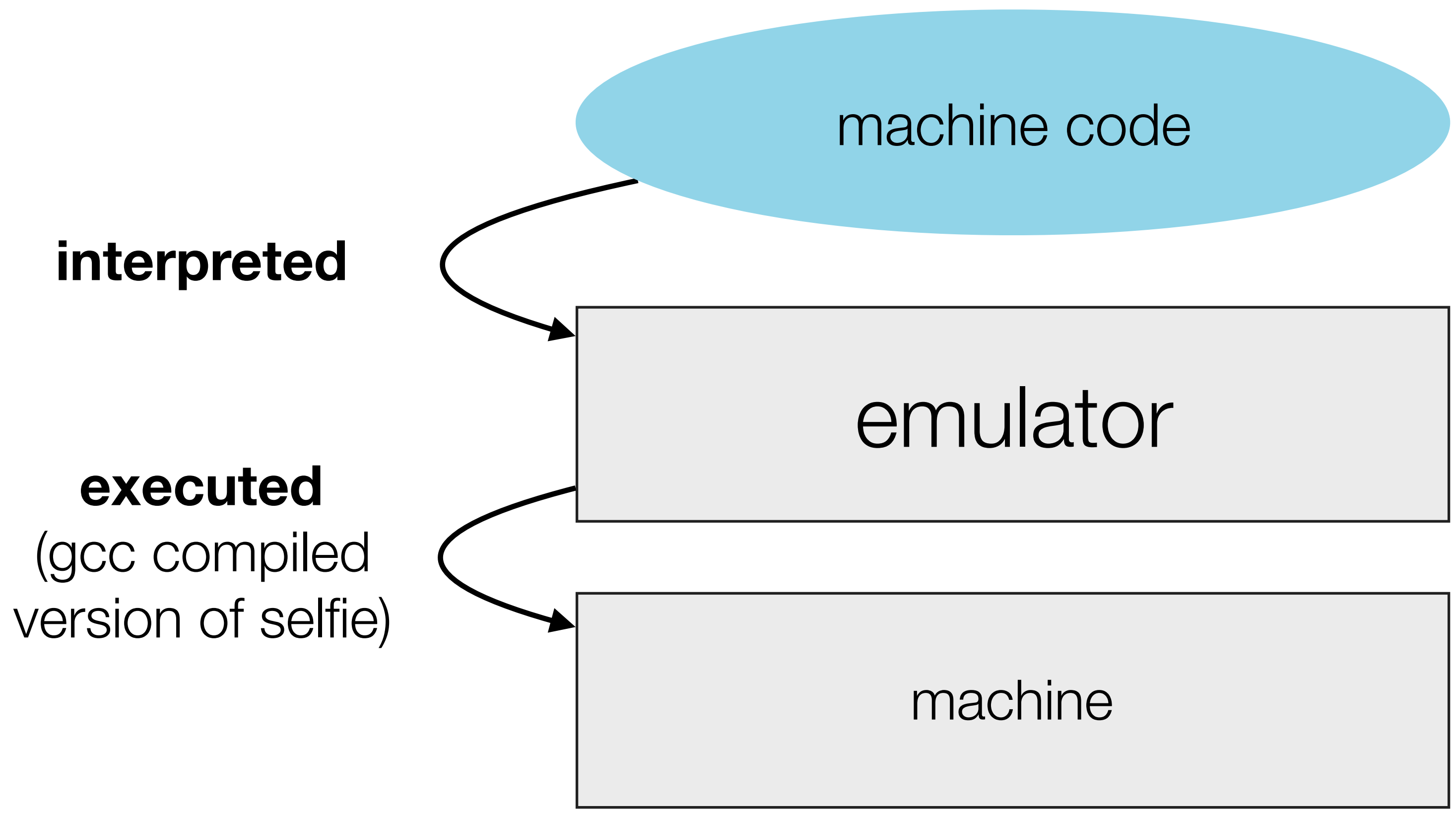


- A compiler **translates** code written in a high-level language into code that the machine (processor) can 'understand' and execute.
  - <u>High-level languages</u> have a structure that defines the control flow.
  - <u>Machine code</u> has no structure, it is just a sequence of instructions.
- The compiler reads an input program, which is a sequence of characters (ASCII, UTF-8-encoded), and writes machine code.
- The selfie compiler written in C* translates C* code (self-referential). (to RISC-U code!)

Using the Selfie Compiler



An Emulator



- The selfie emulator imitates a RISC-U processor. It **interprets** the machine code generated by the selfie compiler.

Using the Selfie Emulator

load binary code

```
./selfie -l example.m
```

start **mipster** and
emulate binary code with 1MB
of virtual memory

```
-m 1
```

```
./selfie -c example.c
```

start **starc** compiler and
compile to binary code

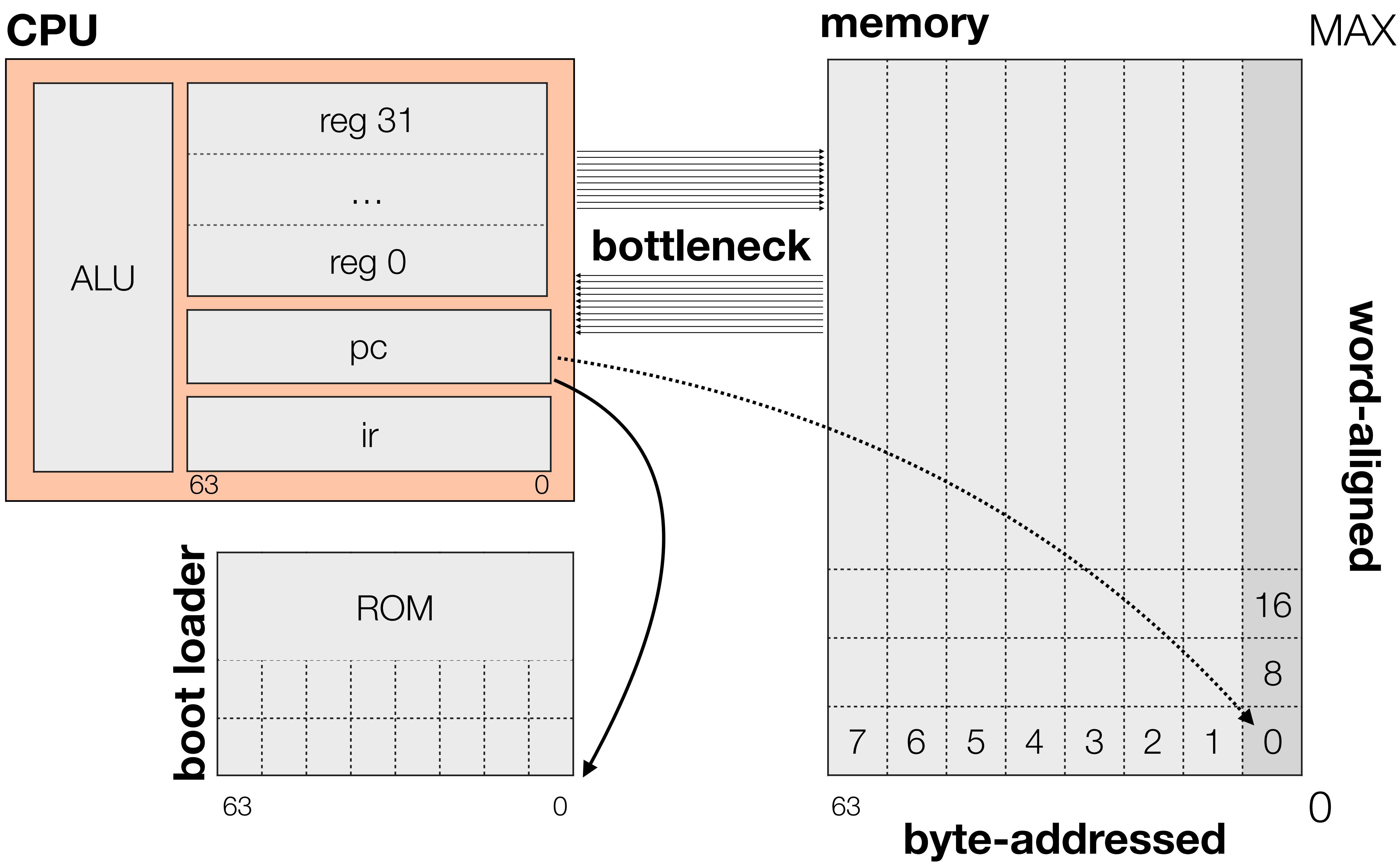
The Compiler

- We know C*, the language that the selfie compiler translates.
- Before we study the selfie compiler, we need to learn about the processor that the compiler targets, and the processors language.

Introduction to

Architecture and RISC-U

Von Neumann Machine



More about the [Von Neumann architecture]

Von Neumann Machine

- **Key idea** - code and data in same memory

- How do we know what code is and what data is?
  - The **program counter** points to an instruction in memory making it code. This instruction may instruct the processor to load bits from memory and modify them making these bits data.

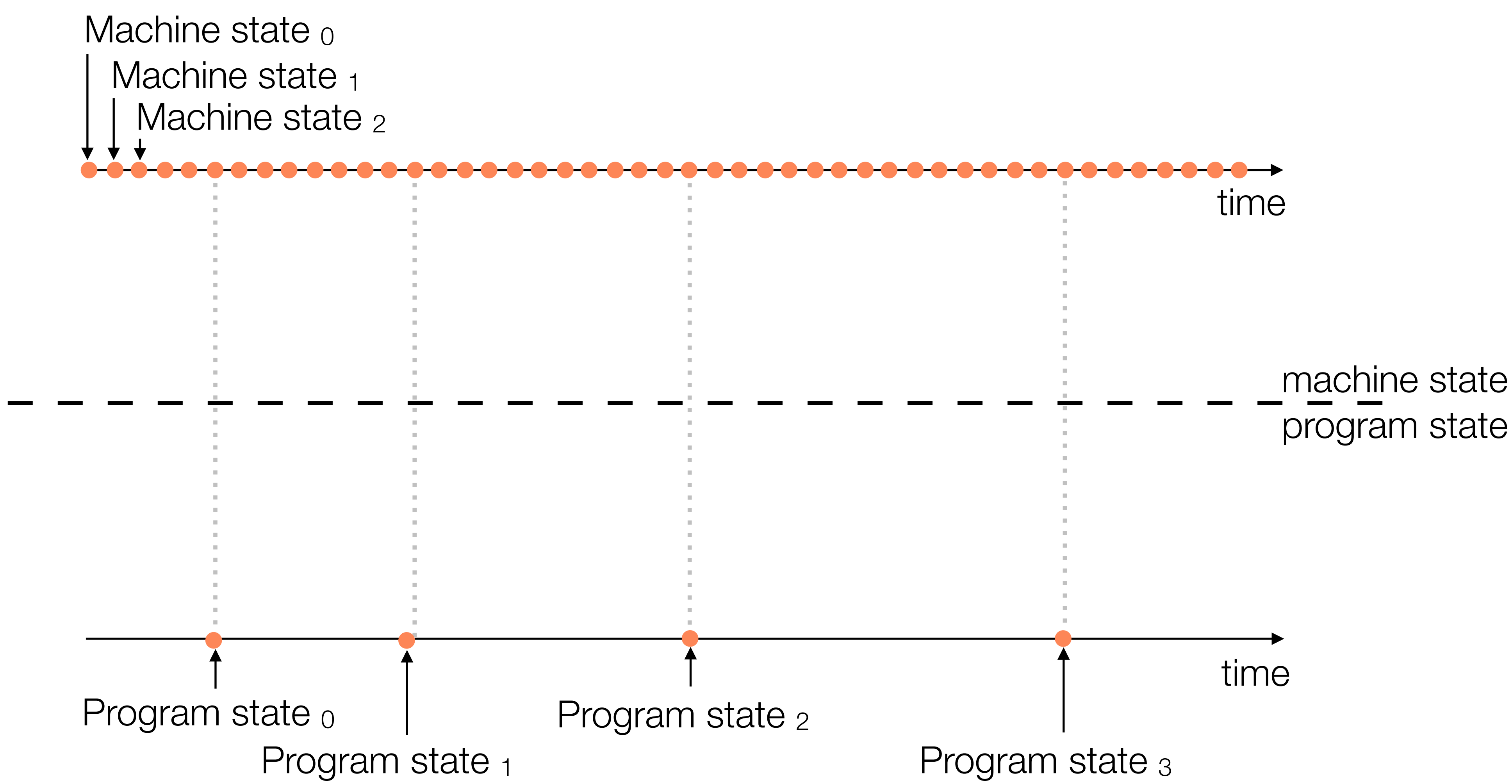
- **Bootstrapping** - Loading the first program
  - At the very beginning hardware support is needed to set the PC to address 0 of the memory where the boot-loader code is stored.
  - The boot-loader is code stored in non-volatile memory (ROM) that instructs the processor to load code.
  - Last instruction sets the PC to address 0 of main memory.

States Machine State and Program State

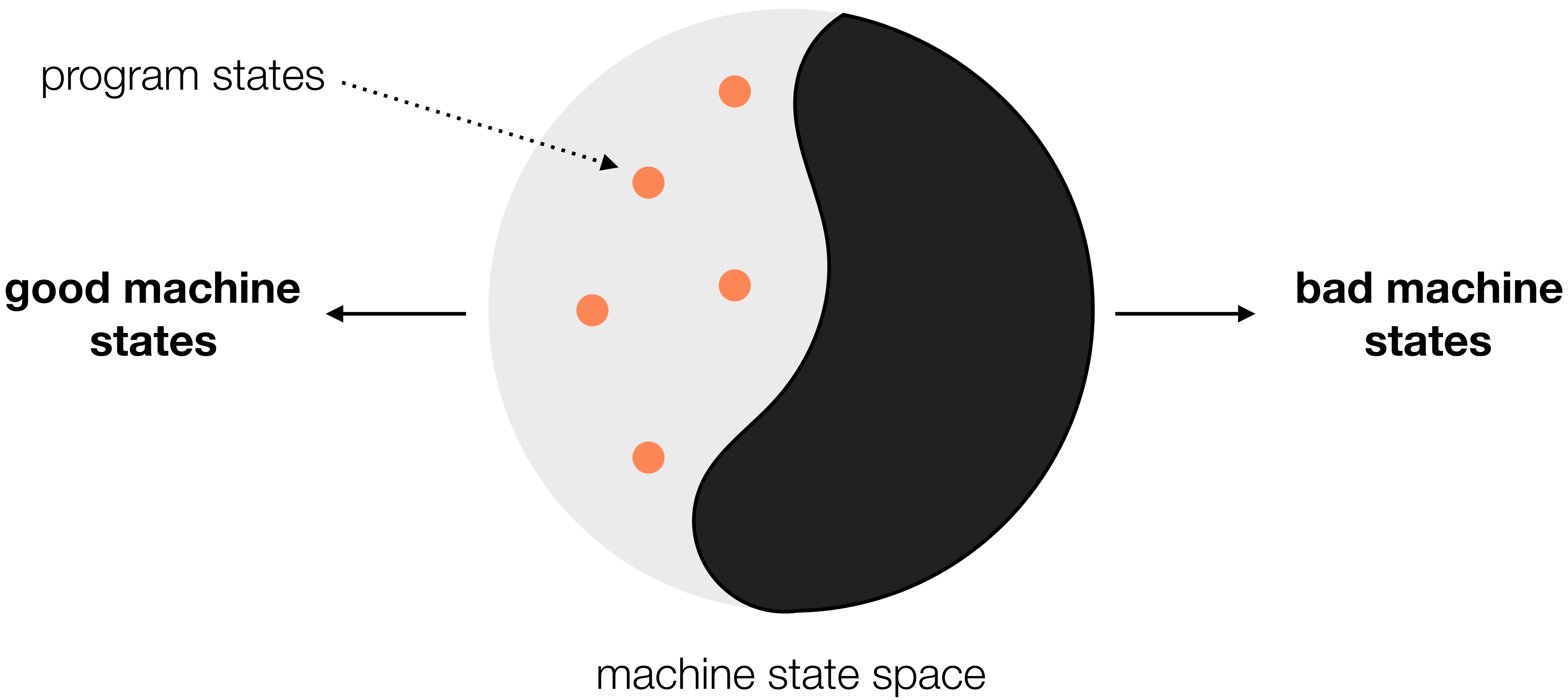
- **Machine State**
  - defined by the all the bits in the machine
  - #machine states = #bits ^{2}
- **Program State**
  - global variables
  - call stack - current statement
  - dynamically allocated memory (state space explodes)
  - #program states much smaller then #machine states

State Space High Level Programming Languages

- each program state corresponds to some machine state
- far less program states (high-level) than there are machine states



Correctness



- A correct program never takes the machine into the "bad state" space. (for any execution)

Introduction to RISC-U

- RISC-U is the RISC-V subset targeted, emulated, and virtualized by selfie.
- There are 14 instructions, each 32-bit wide, the processor (emulator) knows and the compiler generates code for.
- When talking about formal languages it is important to distinguish between the syntax and the semantics of that language.

Syntax of RISC-U

- At machine code level we distinguish further between the human readable <u>assembly</u> code and the <u>binary</u> code the CPU executes.

```
addi $t02 $t01 4

00000000010000011000000001110010011
```

- Selfie always compiles to/stores binary code which can then be <u>disassembled</u> to obtain the assembly code.

Syntax of RISC-U

- Different instructions have different binary encodings. Instructions are encoded using special **formats**.

- The format of an instruction specifies how the 32 bits are interpreted. It is designed in a way that allows fast decoding of the instructions.
  - the last 7 bit (LSB) of an instruction specify the format
  - ex. `addi` uses I-format

this part is interpreted
as a numerical value, namely 4

I-format

000000000100 0001100000011 10010011

(12 bit)

(7 bit)

- For other formats these 12 bit are interpreted differently.

Semantics of RISC-U

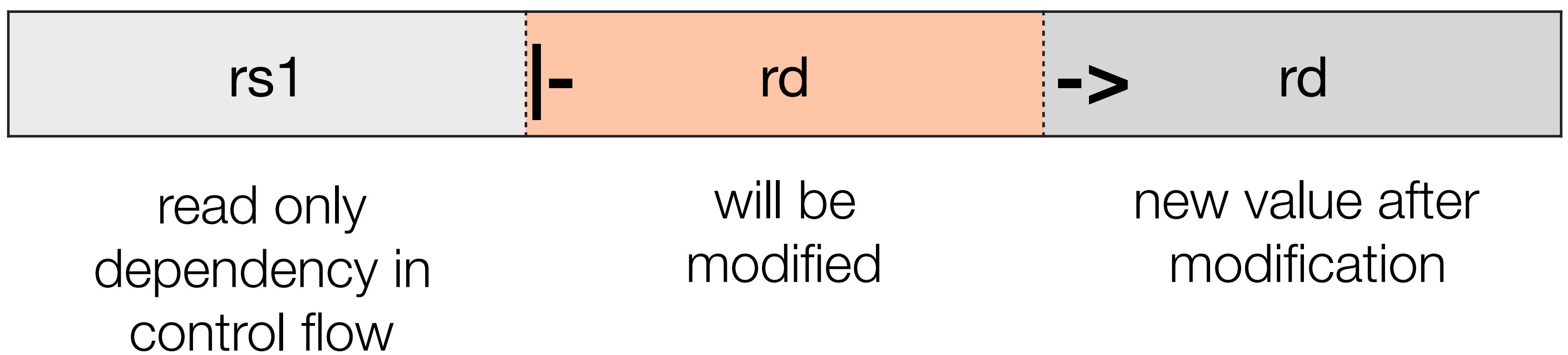
`./selfie -c`

- The semantics of an instruction is determined by how the processor implements it.
- Truly understanding each instruction is key to understanding the target language generated by the compiler and executed by the processor.
- In selfie this implementation is done in the `do_..` ’s, like `do_addi()` or `do_sltu()`.

Instructions and Machine State

./selfie -c

- Selfie uses special procedures (`_before()` & `_after()`) that show on which part of the machine state they depend, which part they modify and the modification itself.



- This information is enough to determine the machine state at any point of execution (completely deterministic).

- The only way to inject information from outside that is not known beforehand is through the read call.

Instructions and Formats

`./selfie -c`

RISC-U

lui	beq
addi	jal
add	jalr
sub	ld
mul	sd
divu	sltu
remu	ecall

special case of **`addi`**

nop

Formats

R-format
I-format
S-format
B-format
J-format
U-format

Immediate Arithmetic Instructions

`./selfie -c`

`lui`	`$rd`	`imm`	
`addi`	`$rd`	`$rs1`	`imm`

- **Load upper immediate** loads the immediate value `imm` shifted by 12 bits into register `$rd` and **add immediate** adds `imm` to the content of `$rs1` and stores the result in `$rd`.

- These two instructions are used to **initialize registers** (`$rs1 = $zero`) and to **load addresses** into a register in order to read values from memory.

- `lui` is used to load the upper and `addi` to load the lower bits - see `load_integer(uint64_t value)`.

- A special case of `addi` is `nop`, with `$zero = $zero + 0`.

Arithmetic Instructions

`add`	`$rd`	`$rs1`	`$rs2`
`sub`	`$rd`	`$rs1`	`$rs2`
`mul`	`$rd`	`$rs1`	`$rs2`
`divu`	`$rd`	`$rs1`	`$rs2`
`remu`	`$rd`	`$rs1`	`$rs2`

`$rd = $rs1 +,-,*,/,% $rs2`

- The processor executes these instructions using <u>unsigned integer arithmetic</u> with <u>wrap-around semantics</u>.

Wrap-around Semantics

- Cause of unbelievably expensive bugs, e.g. the Ariane 5 Flight 501.
- 2^{64} - 1 is the largest value that can be represented by 64 bits. In selfie this value is denoted `UINT64_MAX`.
- Adding `1` to `UINT64_MAX` leads to a wrap-around where only the 64 LSB are considered.

64 bit

2^{64} - 1 111111111…111111111

2^{64} - 1 + 1 1 000000000…000000000 wrap-around

```
UINT64_MAX + 1 = 0
UINT64_MAX + 2 = 1
UINT64_MAX + 3 = 2
…
```

Arithmetic Instructions

sltu	\$rd	\$rs1	\$rs2

- **Set Less Than Unsigned**
- Set `$rd` to `1` if `$rs1 < $rs2`.
- This is the only instruction needed to implement <, >, <=, >=, == and `!=`.
- How this is done:
  - `'=='` is implemented using `b - a < 1`.

In unsigned arithmetic only 0 satisfies this condition.

Memory Instructions

`ld`	`$rd`	`offset($rs1)`
`sd`	`$rs2`	`offset($rs1)`

- **Load** into `$rd` the value that is stored at the address that is obtained by adding the immediate offset to the content of `$rs1`.
- **Store** the content of `$rs2` at the address that is obtained by adding the immediate offset to the content of `$rs1`.
- The addressing mode used for these instructions is called register-relative addressing.

Control-Flow Instructions

- Control flow, at machine code level, is the order in which instructions are executed.
- All previous instructions feature implicit **trivial control flow**, that is, they simply set the program counter to the next instruction. Their main purpose is **data manipulation.**
- The following instructions have a more sophisticated effect on control flow.

Control-Flow Instructions

`beq`	`$rs1`	`$rs2`	`imm`
`jal`	`$rd`	`imm`	
`jalr`	`$rd`	`$rs1`	`imm`

- The first two instructions use a different addressing mode called pc-relative addressing at the resolution of 12 bit.

- **Branch on equal** sets the `pc` to `pc + imm` if the content of `$rs1` matches `$rs2`.

- **Jump and link** is used for procedure calls and stores the return address (address of next instruction) in `$rd`.

- **Jump and link register** is similar to `jal`, except that it uses register-relative addressing.

Link Register

```
uint64_t increment(uint64_t inc) {
   return inc + 1;
}

uint64_t main() {
   uint64_t a;

   a = 0
   a = increment(a);

   return a;
}
```

The next **instruction** that is linked is the assignment into a!

Operating System Support

`./selfie -c`

```
ecall
```

- This is not a standard machine instruction but rather a **programmable instruction** that allows you to request services from the operating system (it is also known as system call).

- The arguments that specify what action we need the OS to take are provided in `$a7`.

- Selfie supports these ecalls: `read`, `write`, `open`, `malloc`, and `exit`.

- The implementation of each of these ecalls is in `handle_system_call()`.

Bitwise Operations (Instructions)

- In selfie there are no bitwise operators and no native machine instructions for bitwise operations.
  - This section → introduce new machine instructions for bitwise operations
  - Next section → show how new language operators can be **systematically** implemented.

Language Operators

- The compiler has to recognize the operator and generate code for it.
- The processor has to understand the instructions encoded by the compiler and executes them.
- We first expand the processor by implementing a new machine instruction that can then be used by the compiler to generate code.

gets called by the **compiler**

encode

need to match

decode

gets called by the **processor**

Implementing Machine Instructions

- Extending the Processor → Implementing machine instructions for `SLL` and `SRL`.
- Before implementing an instruction the intended **semantics** must be clear.

Bitwise Operators

- Bitwise operators in C perform operations on bit level. There are:
  - bitwise operators: & (bitwise AND), | (bitwise OR) and ~ (bitwise NOT).
  - shift bitwise operators: << (bitwise left shift) and >> (bitwise right shift).

- Logical shifts (zeros are shifted in) or arithmetic shifts (sign bit is shifted in).

Bitwise Operators in Selfie

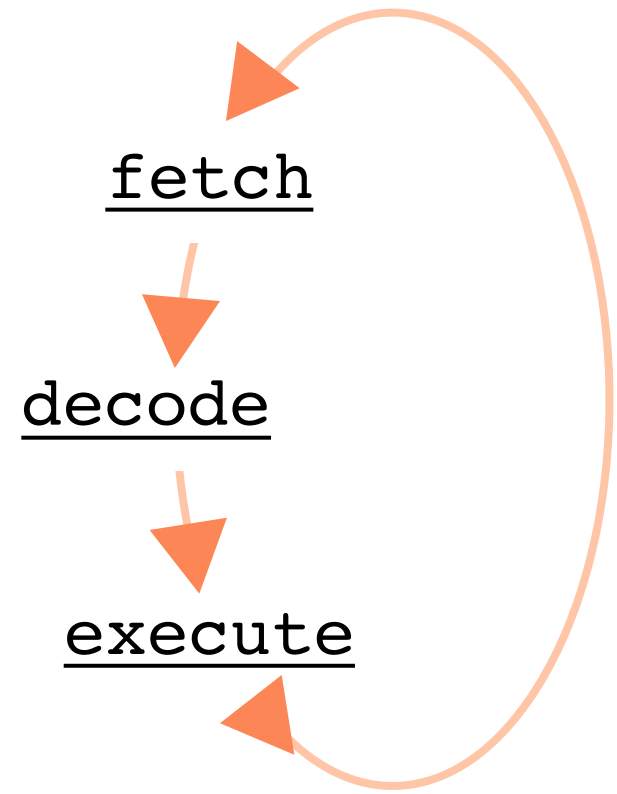
`./selfie -c`

- In selfie there are library functions that perform shift operations (see here):
  - `left_shift()`: $n << b = n * 2^b$
  - `right_shift()`: $n >> b = n / 2^b$
- The **semantics** of the bitwise operators << and >> on the `uint64_t` type is that of a *logical shift*.
- The semantic of instructions is determined by the processor.
  - next we learn how the processor operates

The Processor

`./selfie -c`

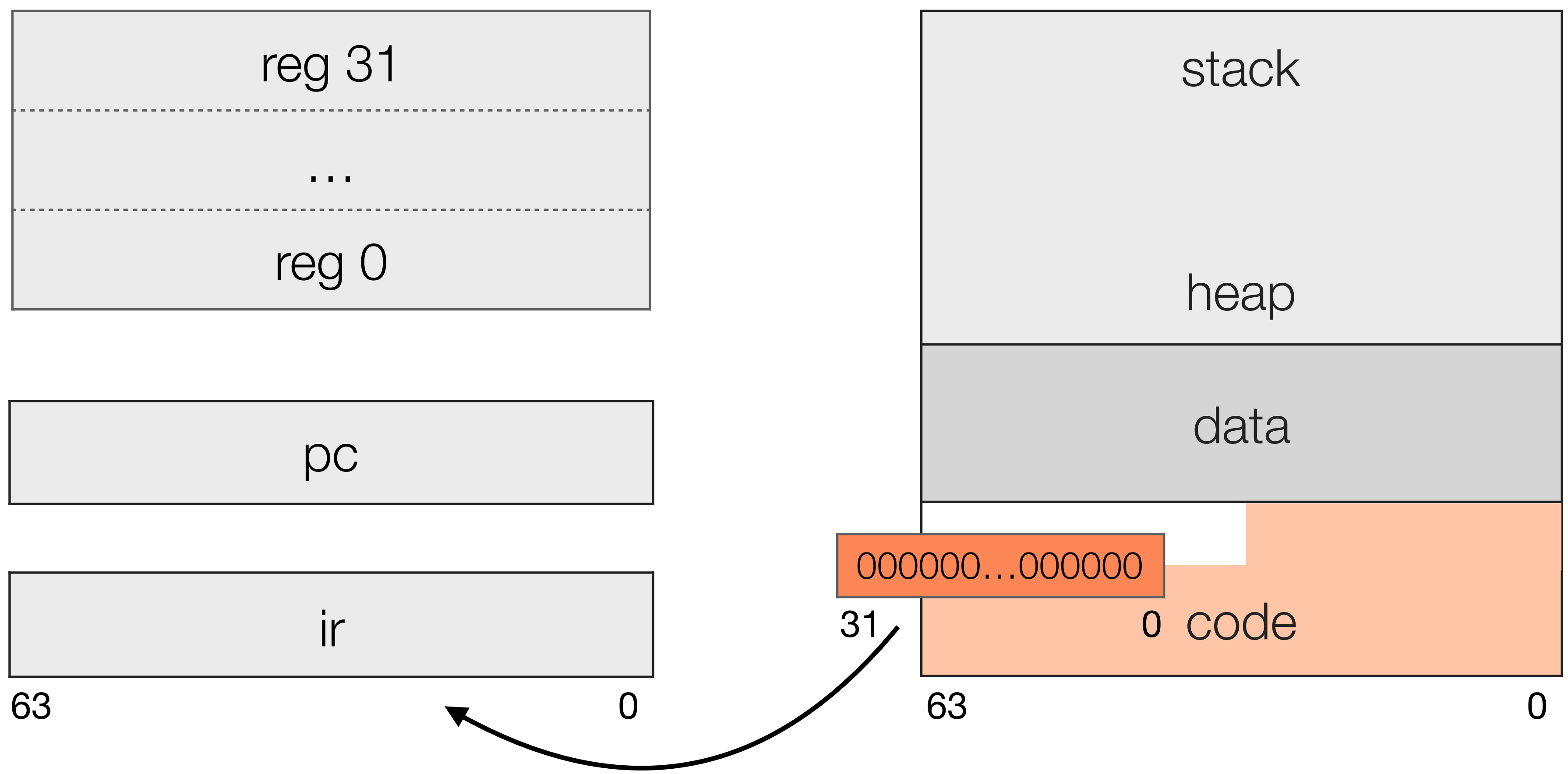
- The heart of the selfie emulator is the procedure
  run_until_exception()



Fetch Instructions

- A 32-bit instruction gets fetched from memory and is then stored in the instruction register (`ir`)

- 2 instructions stored in memory using hi/low word



Decode Instructions

`./selfie -c`

- Each instruction can be uniquely identified by certain parts of the 32 bits called **opcode**, **funct3** and **funct7** code.

- When the instruction is known, the meaning of the remaining bits of the instruction becomes clear.

Level 1

<u>opcode</u>

The opcode specifies the format in which the instruction is encoded.

The opcode might already uniquely identify the instruction that has to be decoded. If not, the funct3 code is looked at next.

looking further

Level 2

<u>funct3</u>

The opcode in combination with the funct3 code might already identify the instruction. Not part of all instruction formats.

looking further

Level 3

<u>funct7</u>

The instruction is definitely known at this point. Not part of all instruction formats.

Decode Instructions

`./selfie -c`

- The opcode specifies the instruction format in which the instruction is encoded.
- After decoding the binary code of the instruction, the processor knows what instruction and parameters it is dealing with.
- Example **ADDI**:
  - Immediate instructions contain bits that are interpreted as numerical value. This value is retrieved using shifting operations.
  - 5 bit are enough to encode 32 registers.

immediate	rs1	funct3	rd	opcode
12 bit	5 bit	3 bit	5 bit	7 bit

specifies format

Execute Instructions

- The execution of every RISC-U instruction has a well-defined effect. It changes the state of the machine only at a specific location involving little data.

- At most two registers or one register and one memory location are modified by an instruction.
  - Every instruction modifies the **`PC`**.
  - Most instructions which modify **data** (another register or memory location) have trivial control flow (`PC` to next instruction).
  - Control-flow instructions have a more sophisticated **control flow**, that is, they may change the `PC` using relative or absolute addressing.

Execute Instructions

`./selfie -c`

semantics: 64-bit unsigned addition with wrap-around semantics

- Example <u>**`ADDI`**</u>:
  - Bits in `$rd` are overwritten with `$rs1 + imm`.
  - `PC = PC + INSTRUCTIONSIZE`.
  - Used for initialization - loading constants into registers.

Realize…

- This is how virtually every general-purpose processor operates.
- Instructions only have a tiny effect on the overall machine state.
- What makes modern computation so powerful is:
  - the incredible **speed**.
  - the enormous size of **memory**.

Assignment 1: SLL, SRL

- In selfie there are no bitwise operators and no native machine instructions for bitwise operations.

- Students can extend the processor and implement the **RISC-U instructions** that perform bitwise left and right shift operations according to RISC-V standards.
  - check how it is done for other instructions (ex. `addi`)
  - encode, emit, decode and execute (`do_`…)

- **Hint**
  - do not break self-compilation! → for `do_`… take a look at `do_jalr()`

- Soon we learn how the compiler can use these instruction to implement shift **operators** ('<<', '>>')

Introduction to

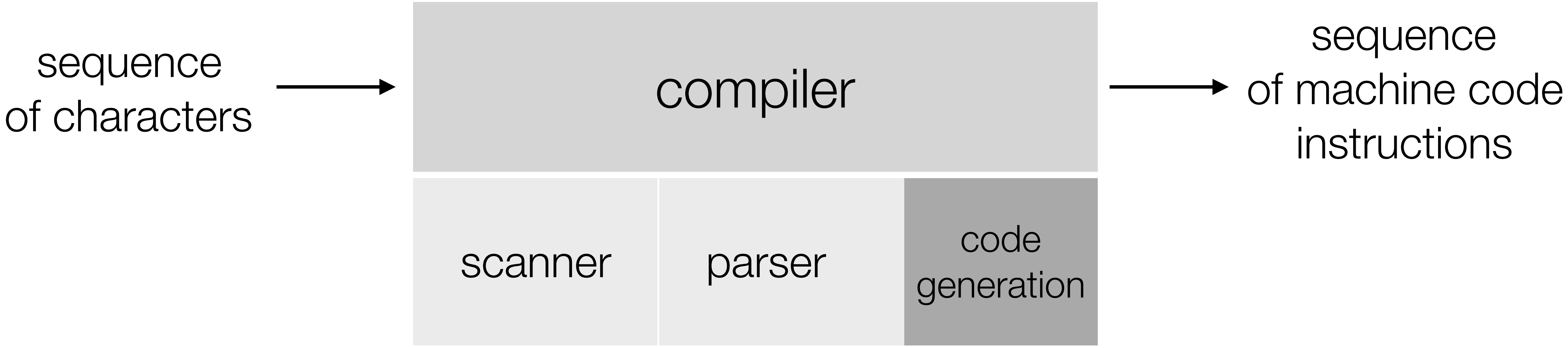
Compilers

Overview

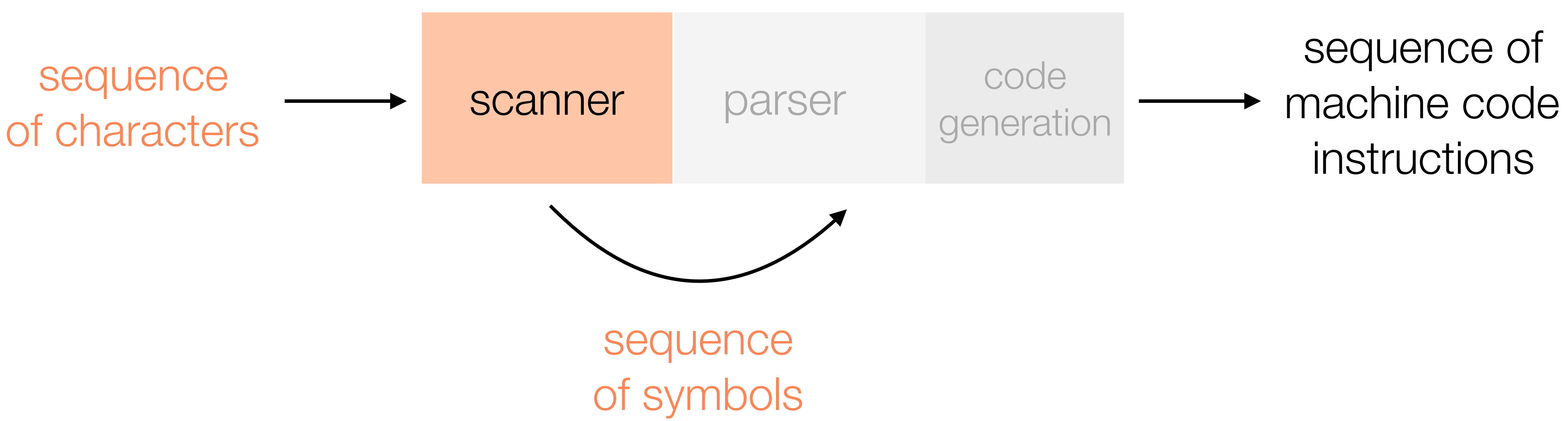
- **The Compiler - Stages**
  - Scanner → valid symbols
  - Parser → valid programs, parsing techniques
  - Code Generation → register allocation problem

- **The Compiler - Concepts**
  - Prerequisites → memory model, variables, symbol table
  - Data Structures and Data Types → array, struct
  - Memory Allocation → procedures, liveness
  - Control Flow → while loop, fixups
  - More Concepts → memory safety, garbage collection, bootstrapping

The Compiler



The Scanner


sequence
of characters
scanner
parser
code
generation
sequence of
machine code
instructions
sequence
of symbols

The Scanner

*The scanner performs a **membership test** on a sequence of characters. It checks whether the characters it sees form **valid symbols** of the programming language.*

- Reading only **one character** at a time the scanner transforms a sequence of characters into a sequence of symbols that has **no structure**.
- There are single-character symbols (`'<'`, `x`) and multi-character symbols (`'<='`,`'variable'`). All symbols are uniquely identified by a token (integer).
- The whole sequence of symbols is not stored. Only the last read symbol is remembered.

What are valid symbols?

`./selfie -c`

- **What is needed:**
  A specification of valid symbols → regular expression
  and a formalism to write such a specification.

- **In selfie…**
  the chosen formalism is EBNF (see in the grammar.md file).

Regular Expression

- Regular expression is a formalism that defines a regular language (= a set of symbols defined by regular expression).

- The production rules consists of terminal symbols, non-terminal symbols and operators (repetition `{}`, concatenation ␣, xor `|` , …).

- A regular expression can be reduced to **a single rule** by replacing every non-terminal symbol with its right-hand side until no non-terminal symbols are left.

non-terminal symbol

```
integer = digit {digit} .
digit   =  "0"| "1" | "2" | "3" | "4" | "5" |
           "6" |  "7" | "8" | "9" .
```

terminal symbol (literals of language)

EBNF Niklaus Wirth

```
ebnf = { production } .

production = identifier "=" expression "." .

expression = term { "|" term } .

term = factor { factor } .

factor = identifier | string | "(" expression ")" |
         "[" expression "]" | "{" expression "}" .

string = """ printableCharacter { printableCharacter }""" .

printableCharacters = "a" | … | "!" .

identifier =  "ebnf" | "production" | … | "identifier .
```

The Scanner

- **The problem statement:**
  Given a specification of valid symbols where symbols may consist of multiple characters…

  …Is a given input (source code) in this specification, or in other words do the characters in the source file form valid symbols according to the specification?

Abstraction

- A scanner for the English language accepts this sentence
- A sequence of characters gets assembled → symbols (words and punctuation).
    - alphabet (a,b,c,...,z) + special characters (!,?,...) accepted (valid)

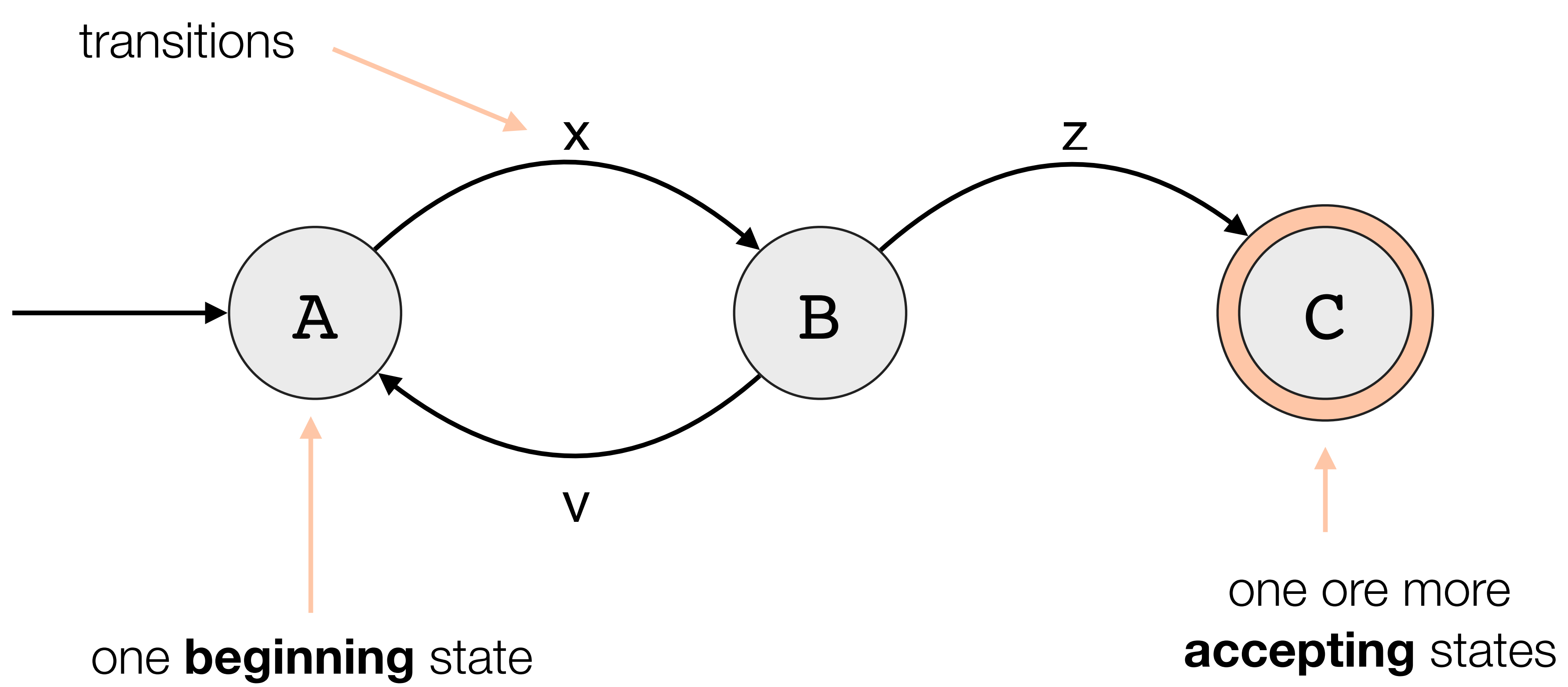
This␣sense␣makes␣!␣no␣sentence

Specification and Implementation

- **Specification**
  The specification of valid symbols is a regular expression written in EBNF. It is easier to reason about the correctness of regular expression and it is expressive enough. The number of valid symbols is infinite and their size is unbounded.

- **Implementation**
  The scanner implements this specification using a finite state machine (FSM), a computational model, that recognizes/accepts the set of regular expressions. The implementation is simple and can be done very efficiently.

FSM



- Transitions describe how to get from one state to another.
- E.x. When in state `B` the condition `z` (may be "reading z") is true, a move to state `C` happens.

The Scanner

`./selfie -c`

get_symbol()

calls

get_character()

find_next_character()

- The heart of the scanner is the procedure `get_symbol()`, which finds the next valid symbol in a sequence of characters and stores it in a global variable (`symbol`).
- It internally uses two procedures `get_character()` and `find_next_character()`.
- `get_character()` simply reads the next character of the input stream and stores it into a global variable (`character`).
- `find_next_character()` is the implementation of whitespace.

Sequence of Characters

- The procedure `get_character()` is used to read the next byte into a buffer.
- The return value is either the number of read bytes or 0, which indicates end of file (EOF).
- Any input file is nothing more than a **sequence of characters** to the scanner. It will even accept meaningless sequences as long as it consists of valid symbols in the specified language.

Whitespace

./selfie -c

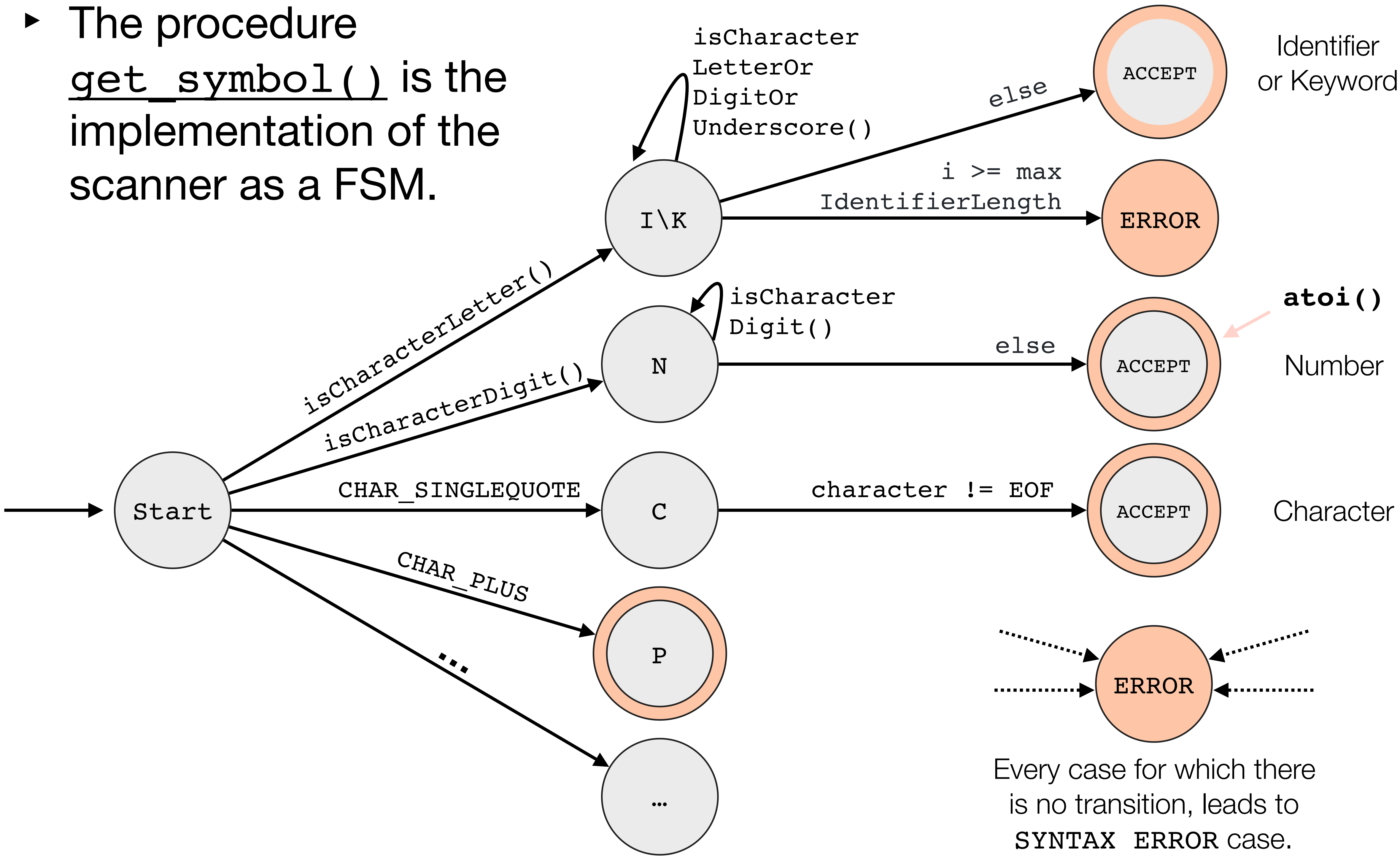
- Whitespace and comments do not change the semantics of code and could even be omitted (minification). Their purpose is to make the code more readable.

- The scanner implements whitespace and comments by simply ignoring them using the procedure `find_next_character()`.

The Scanner getSymbol()

./selfie -c

- The procedure `get_symbol()` is the implementation of the scanner as a FSM.



Properties

- **The scanner never crashes and always terminates:**
  In code the FSM is simply a huge `if/else if/else` statement that covers every possible case. Syntax errors fall through to the `else` case.

- **The scanner is good at forgetting:**
  As soon as the scanner reaches a state it has been in before, everything in between is forgotten.

- **Correctness:**
  The state space of a machine is usually huge or even infinite and therefore reasoning about it is hard. An FSM however has only very few states (finitely many) and reasoning about them is much easier.

Properties

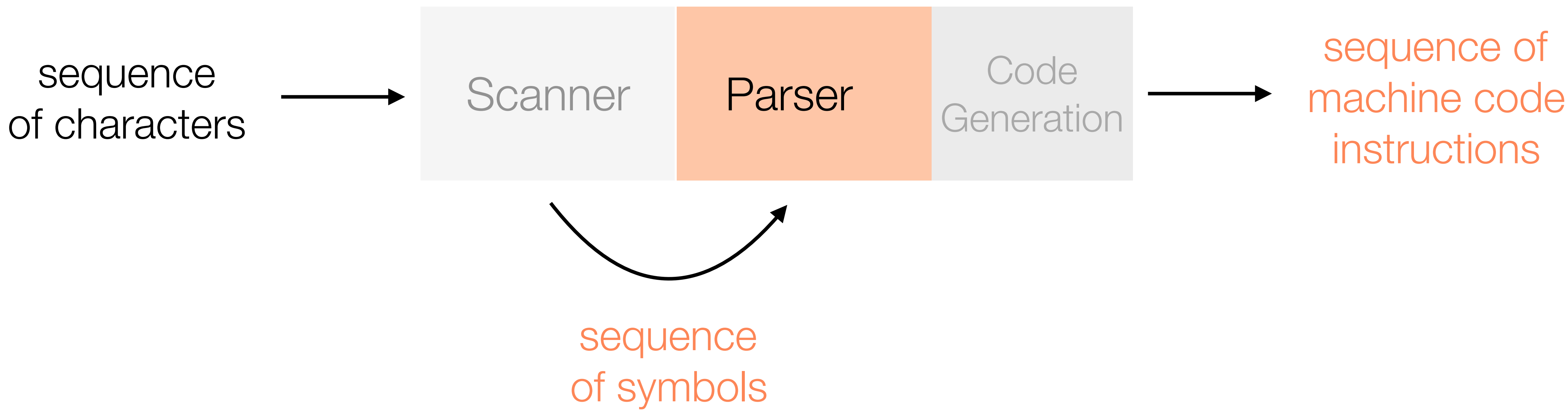
- **The scanner can not recognize structure:**
  It just operates on a sequence of characters and turns it into a sequence of symbols.
  To recognize structure, the scanner would need to be able to **count and remember** (e.g. the number of braces).

  Therefore we need something more powerful called a parser.

Assignment 2: << >>

- The scanner does not recognize the shift symbols << and >>
- Students can extend the compiler such that the scanner recognizes these **symbols**
    - check how it is done for '<='
- **Hint**
    - think of it as extending the FSM (add new states)

The Parser



sequence
of characters
Scanner
Parser
Code
Generation
sequence of
machine code
instructions
sequence
of symbols

The Parser

*The parser performs a **membership test** on a sequence of symbols. It checks whether the symbols form a **syntactically valid** program.*

- The set of syntactically valid programs is infinite.
- The parser builds an internal representation of the structure of the input.
- The technique used in selfie is single-pass, recursive-descent parsing.

What are syntactically valid programs?

- **What is needed:**
  A specification of syntactically valid programs → <u>Grammar</u>
  and a Formalism to write such a specification.

- **In selfie…**
  the specification is a <u>context-free grammar</u> (CNF) described using EBNF.

Abstraction

- A parser for the English language accepts this syntactically correct sentence
  - subject → verb → object → punctuation

This␣sense␣makes␣no␣sentence!

Context-Free Grammar

valid C* symbols → **Scanner** → Using **Regular Expression** that can be written as single rule.

valid C* programs → **Parser** → Using **Context Free Grammar** that cannot be written as single rule, because there is recursion over multiple production rules.

- A CFG is a set of <u>production rules</u>.
- There is only one <u>non-terminal symbol</u> on the left hand side. This symbol can be **substituted in any context** in which it appears on the right hand side.
- The parser checks if the input **matches** the production rules.

Parsing Techniques recursive-descent

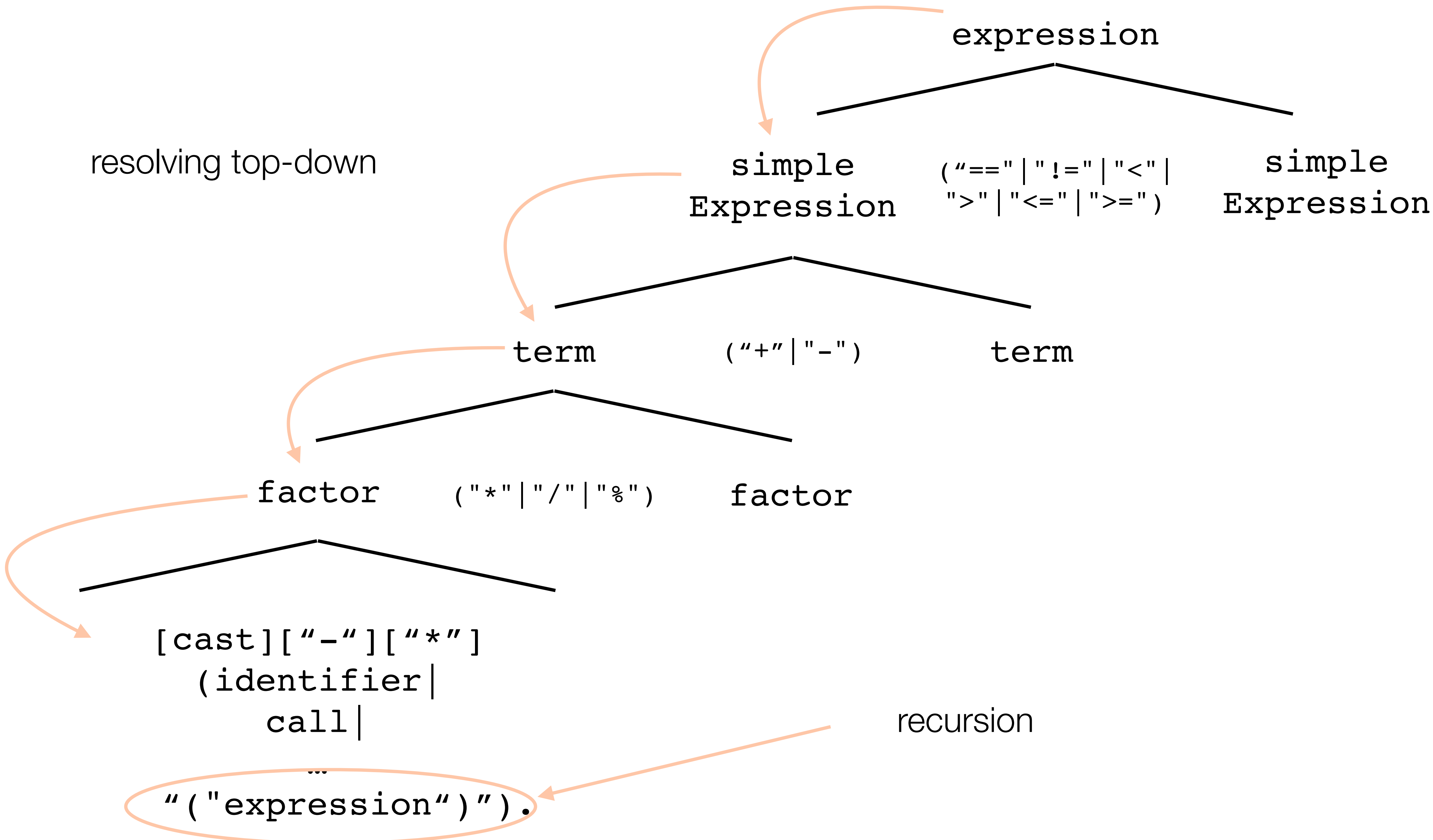
- A recursive descent parser uses recursion and a top-down approach.
- The top-down approach treats the input as a **single start symbol**.
  - start symbol is the left-hand side of the production rule for 'valid program'
- Every symbol that appears on the right-hand side of a production rule is (recursively) **resolved** and **matched** to the input.
- The recursive nature of the parsing rules allows that a symbol that has been resolved in a previous step can appear again on the right-hand side of a production rule.

Parsing Techniques single-pass & multi-pass

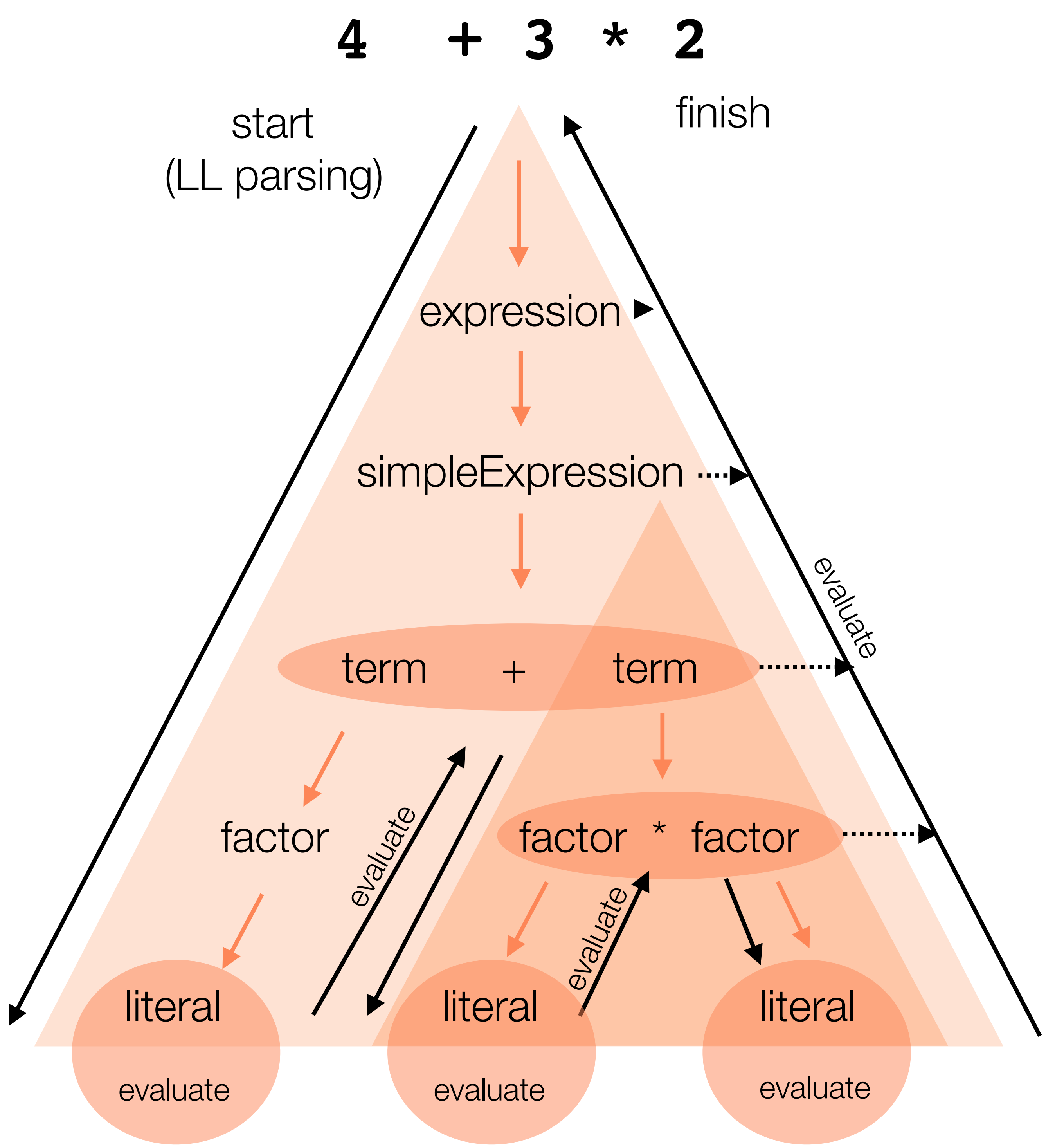
- The internal representation build by the parser is called a syntax tree.
- A <u>single-pass</u> parser parses the input once and never looks back. At any point it only represents a single branch of the syntax tree and never the whole context. This only allows local performance optimization.
- A <u>multi-pass</u> parser builds an internal representation of the whole syntax tree and walks over it multiple times, trying to optimize the performance on a global level.

Example: expression()

```
expression = simpleExpression [("=="|"!="|"<"|">"|"<="|">=") simpleExpression]
```



Parsing by Example



- To evaluate an expression we keep **resolving all non-terminal** symbols until only a **terminal** symbols is left (e.g `literal, '+'`). This terminal symbol is then **matched** against the input (`4,+`).
- So each **branch/subtree** is completely evaluated **before** the result of a parent node can be calculated.

Parsing by Example

- So each **branch/subtree** is completely evaluated **before** the result of a parent node can be calculated.

- In the next slides we will see the implications of this statement (on a single-parse parser).

Parsing Technique Implications

- **The selfie parser is a single-pass parser:**
  The input is read only once, without ever looking back. This means that code must be already generated during the parsing process.

  → The code generator is part of the parser.

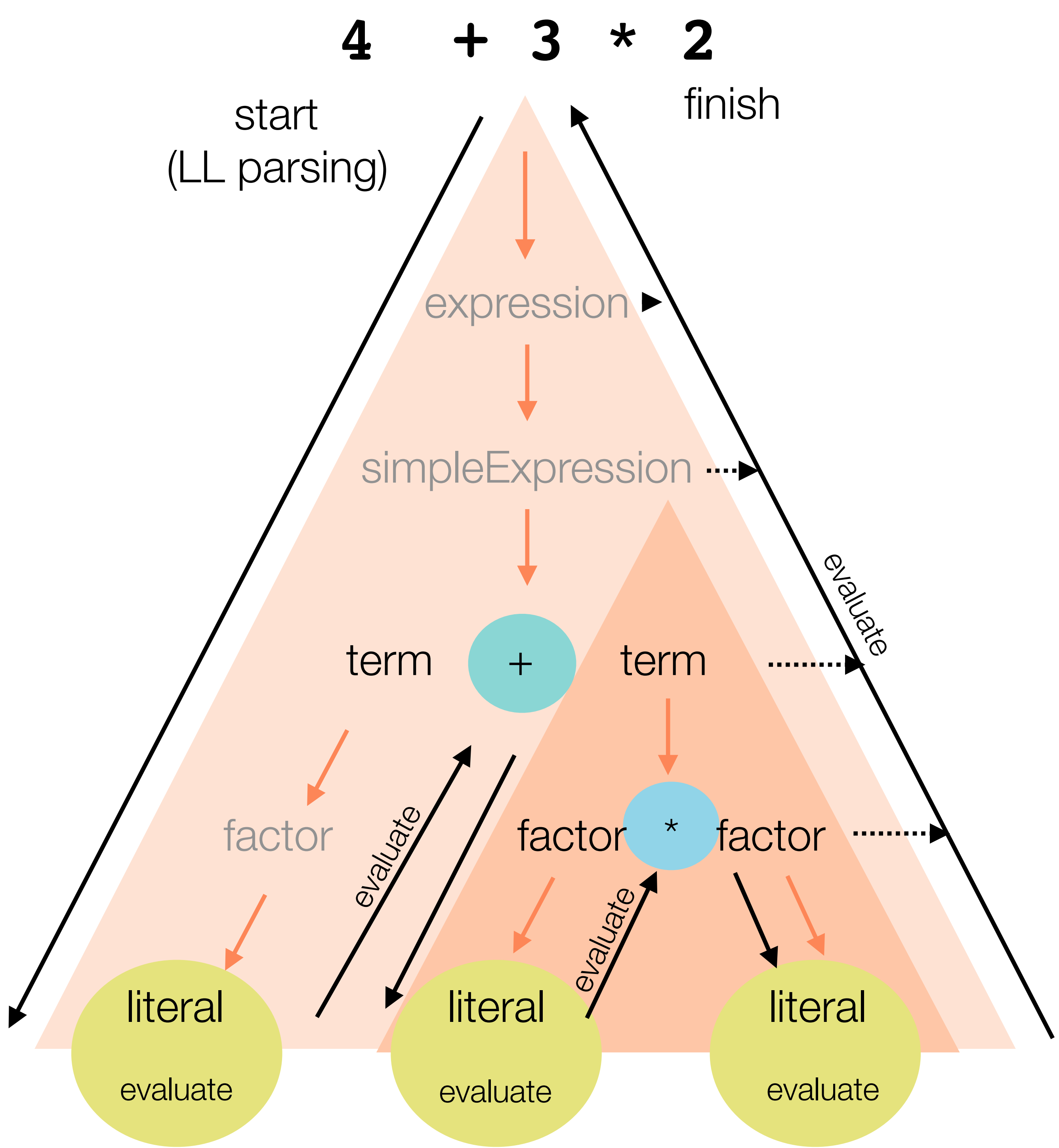
- **The selfie parser is a top-down parser:**
  Compiling an expression using a top-down approach may require to store intermediate computations temporary in registers. But we need to keep in mind that there is only a very limited amount of those (7 in selfie).

  → We have to deal with the register allocation problem

Code Generation

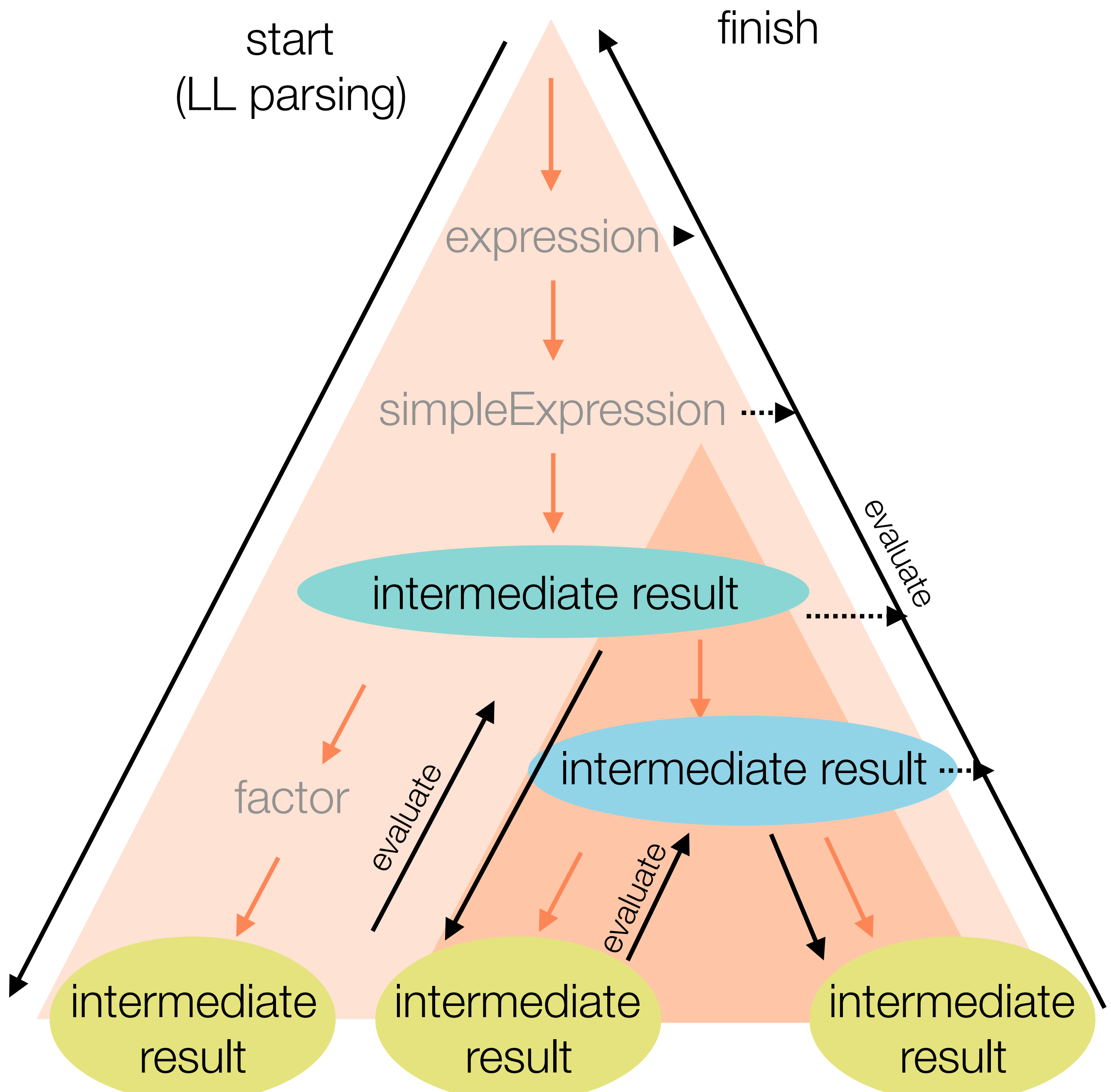
- The following example will show…
  - …(again) how an expression is evaluated by **resolving** all non-terminal symbols and **matching** the terminal symbols against the input.
  - that the compiler generates code as soon as possible.

Code Generation




```
addi $t0, $zero, 4
addi $t1, $zero, 3
addi $t2, $zero, 2
mul  $t1, $t1, $t2
add  $t0, $t0, $t1
```

Register Allocation Problem




```
addi $t0, $zero, 4
addi $t1, $zero, 3
addi $t2, $zero, 2
mul  $t1, $t1, $t2
add  $t0, $t0, $t1
```

Register Allocation Problem

- Register allocation is a **compile time problem**.

> "*How do we know which registers are **used** and which are **free** at runtime?*"

- When compiling we need to think of the **2 time axis'**
    - At **compile time** we compile and generate code…
    - …but we have to keep in mind that at **runtime** there is only a very limited amount of registers available for actually executing the generated code and computing expressions.

Register Allocation Problem

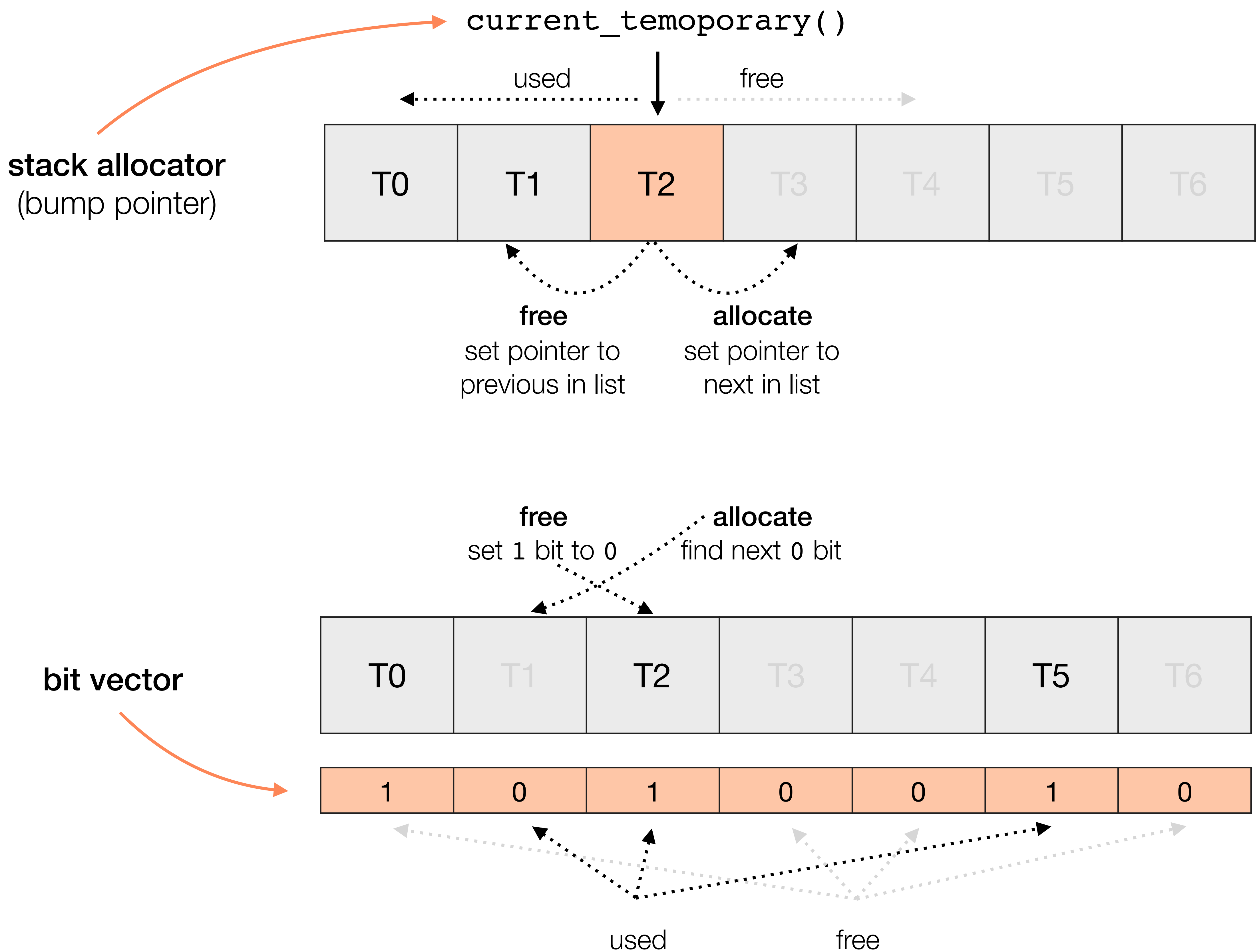
`./selfie -c`

- **stack allocator** (in selfie using `talloc()`, `tfree(...)`)
  - limitation: freeing in reverse order.
    in selfie, this is sufficiently enough and works because of the recursive structure of the parser and code generator.
  - complexity: free, allocate in constant time

- **bit vector** - marking used registers
  - note on complexity: free in constant time, allocate linear in the #registers.

- **advanced approach**
  - this is a question of liveness of variables (in registers) and can be reduced to the graph coloring problem.
    Find the most efficient mapping from registers to variables.

Register Allocation Problem



Assignment 3: << >>

- The parser does not recognize shift expressions like `4 << 2`
- Students can extend the parser such that it can recognize **shift expressions** and generates code accordingly (use SLL and SRL)
  - check how `compile_term` is implemented
  - introduce `compile_shift_expression` with right precedence (parse tree 'level')
- Hint: start with specifying the production rule the `grammar.md` file

Something to think about…

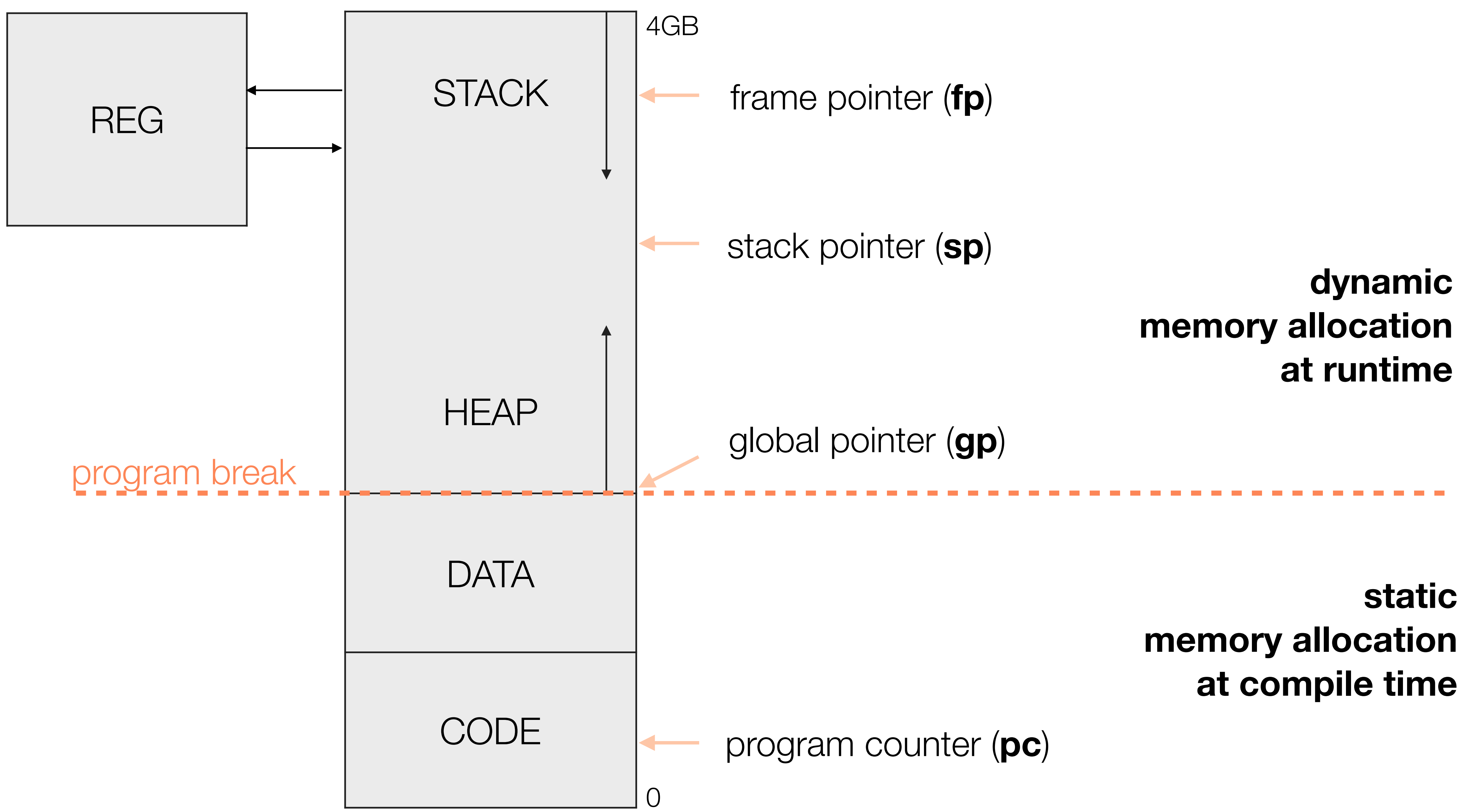
- After finishing Assignments 1-3 one can go back to Assignment 1 (shift instructions) and make a small change do `do_`…’s
  - instead of using left_shift() or right_shift we now can use '<<', '>>'
- **Why?**
  - the selfie compiler now knows '<<', '>>' → self-compilation works
  - gcc knows '<<', '>>' (**bootstrapping**)
    (think about it, it is the same for +, -,…)

Prerequisites

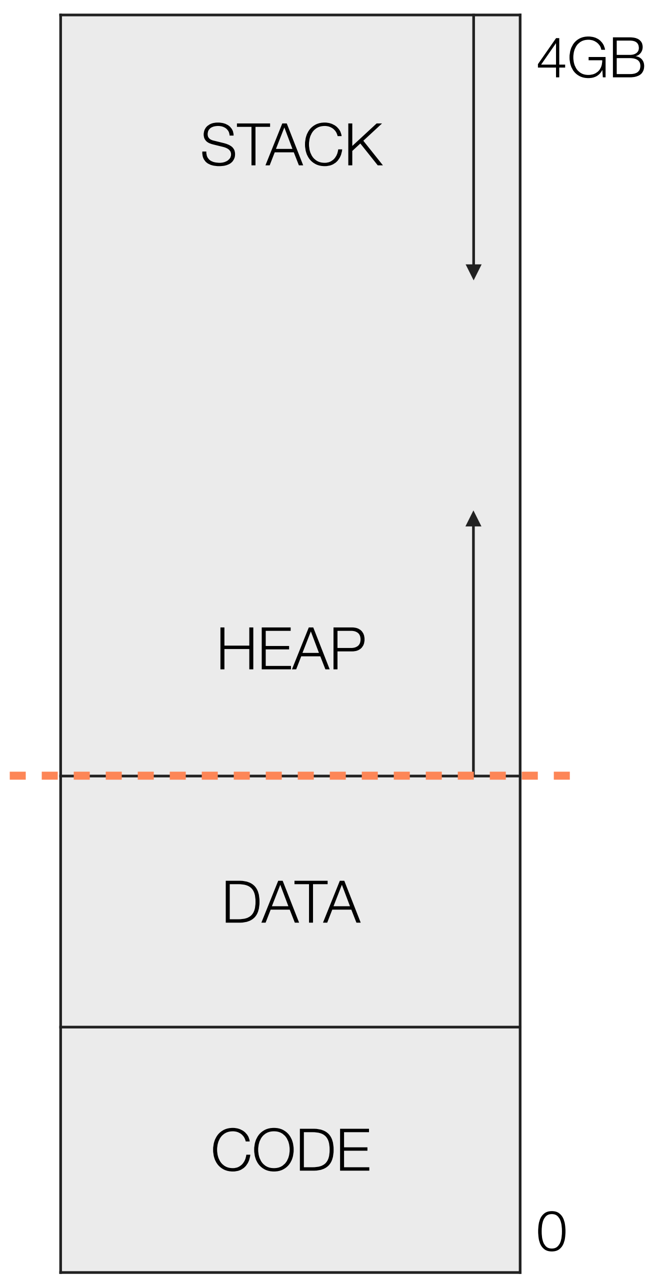
(Data Structures, Data Types)

Memory Model,
Variables

Selfie Memory Model

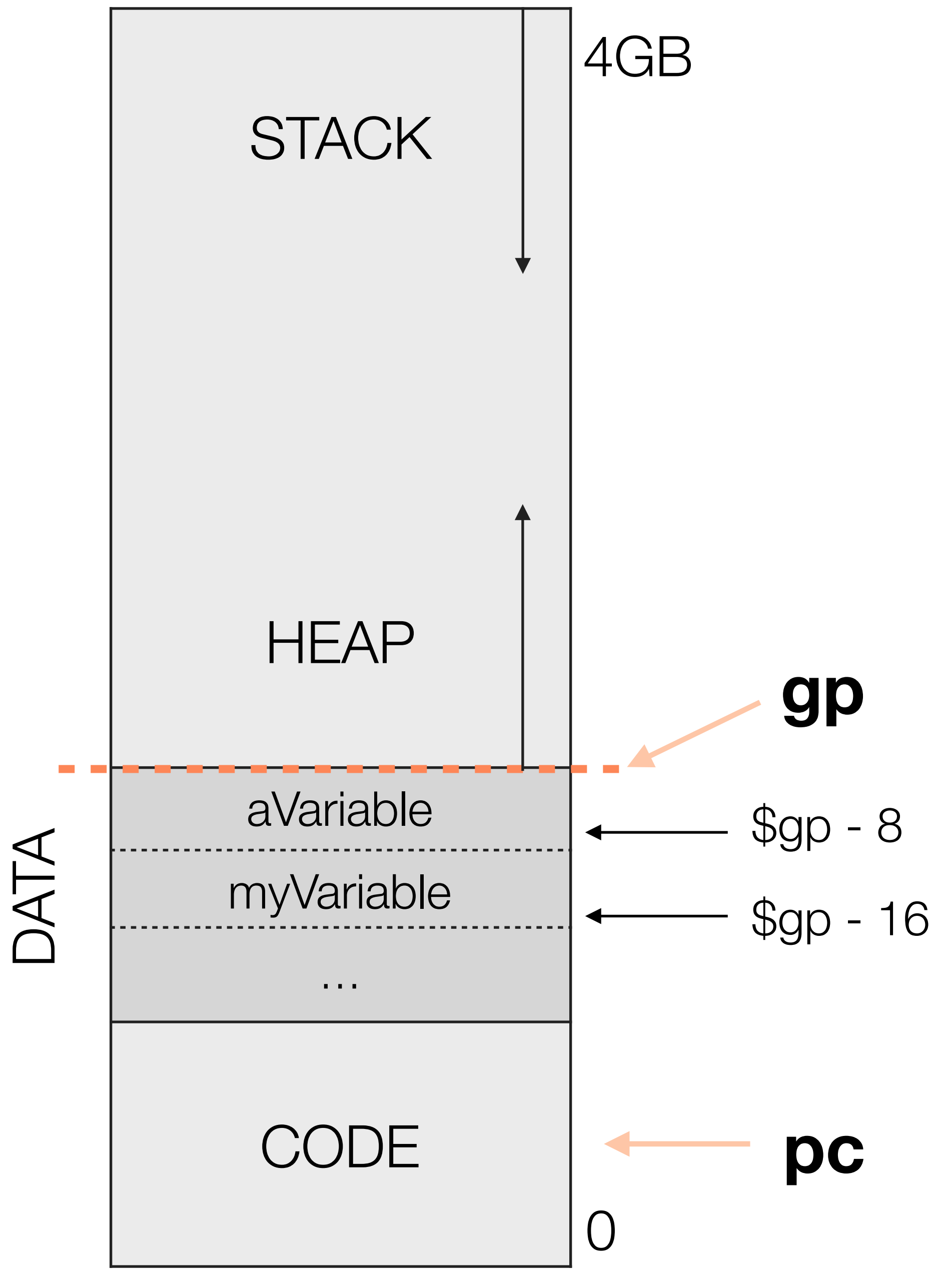


The Memory Layout



- The stack is used for procedure calls. It hosts local variables and procedure parameters.
- Memory allocated by `malloc()` is on the heap.
- The stack grows downwards from high to lower addresses whereas the heap grows upwards.
- The data segment is where global variables, strings and big integers are stored.
- Machine code instructions are stored in the code segment of our memory model.

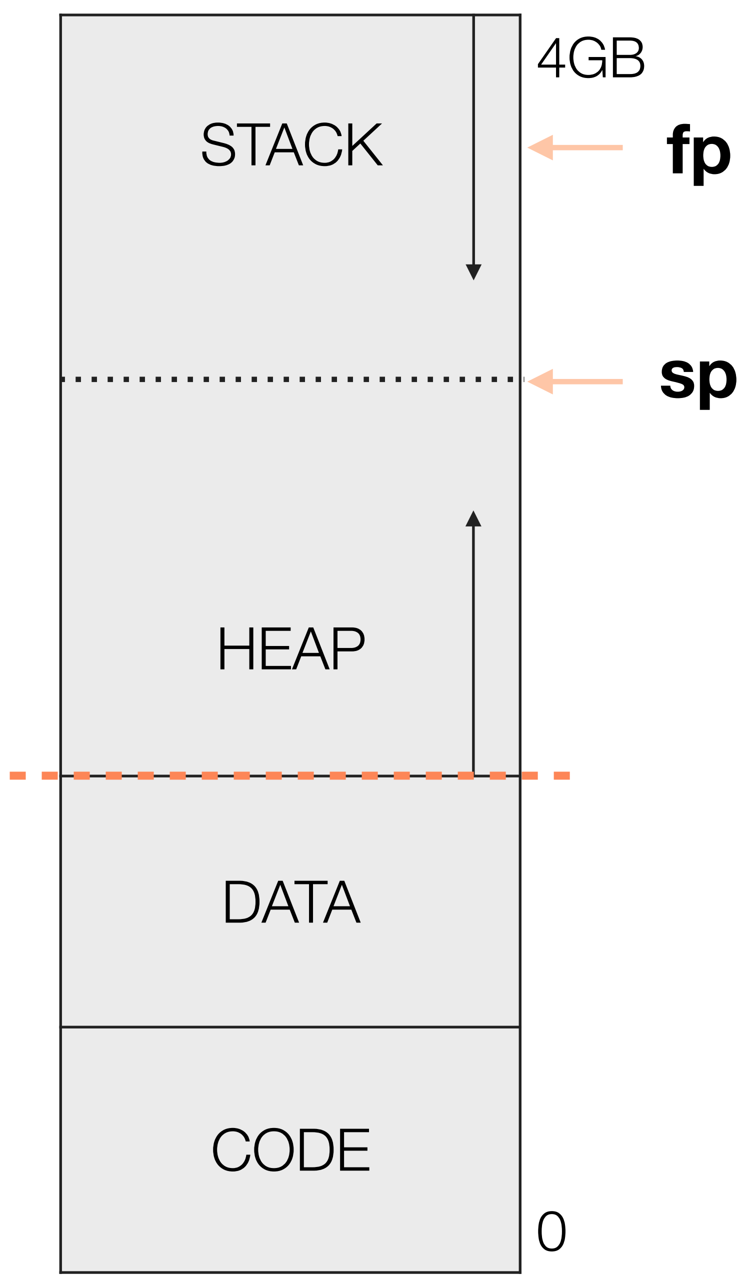
The Pointers



- Global variables, strings and big integers are accessed using the **global pointer** (**gp**) and the negative offset into the data segment that is stored in the symbol table. The address of the first memory cell is `$gp - 8`.

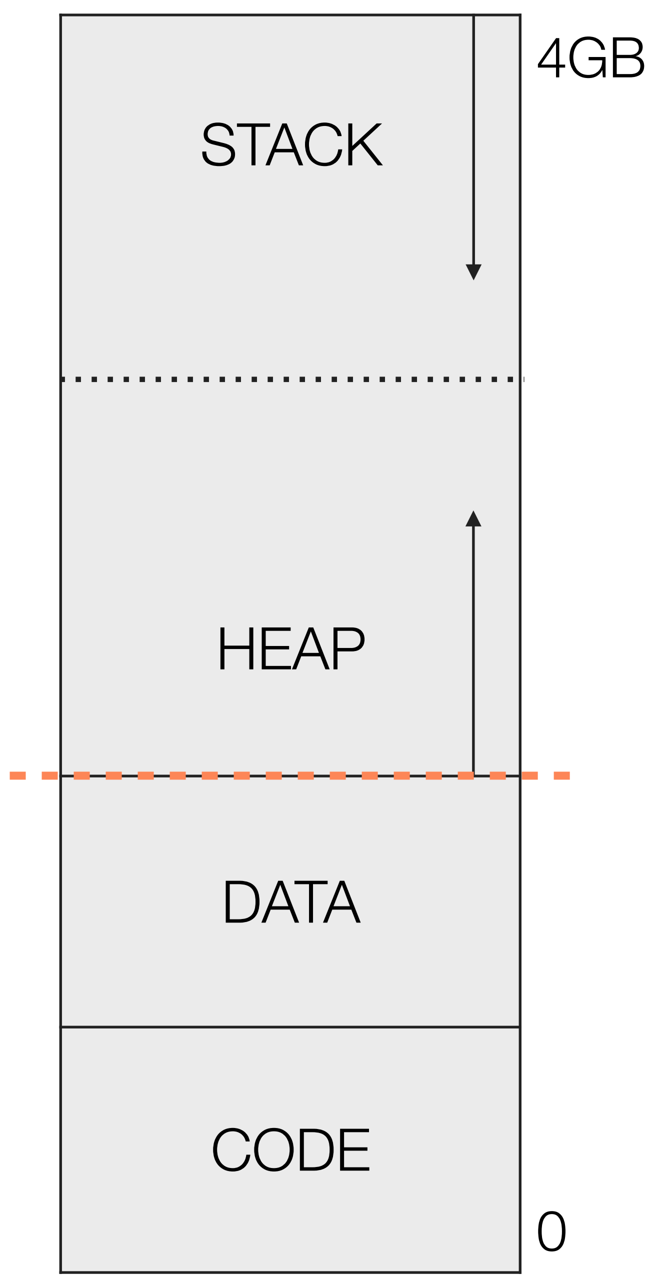
- The **program counter** (**pc**) is set to the next instruction that will be loaded into the instruction register.

The Pointers



- In our model the stack **grows** downwards from high memory addresses to lower ones whereas the heap grows upwards.
- The **stack pointer** always points to the top of the stack (lowest address of the stack) and is used for memory allocation and deallocation on the stack.
- The **frame pointer** provides a convenient way to access local variables and procedure parameters. This will be explained in more detail when procedures are discussed.

Static and Dynamic Memory



- The **program break** splits the memory logically into two parts.
- Memory below the program break is **static**
  - the layout is fully determined **at compile time** and is fixed during execution.
- Memory above the program break is **dynamic**
  - the layout changes **during execution.**

Memory Allocation

- Different concepts depending on when and where memory is allocated.
- The **When**
  - <u>static memory allocation</u> - compile-time concept.
  - <u>dynamic memory allocation</u> - runtime concept.
- The **Where**
  - using `malloc()` - on the heap.
  - procedure calls - in-order on the stack
    = simplest way to dynamically allocate memory.
  - at compile time by the compiler - in the data segment.

Variables

*Variables are **mappings** from names (identifiers) to memory locations (memory addresses) that hold information (value).*

- a variable is defined within a <u>scope</u> that defines its visibility and when/where space is allocated.
- in <u>statically typed languages</u> a variable has a type that gives information about the value associated with that variable (size of the type,…) and how the programmer intends to use the variable.
- in selfie the only types are `uint64_t` and `uint64_t*` (both 8 byte = `REGISTERSIZE`).

Parsing Variables

```
uint64_t aVariable;

uint64_t myVariable;

uint64_t main() {
  myVariable = 2;
  aVariable = myVariable
}
```

Declaration

Access

Global Variables

```
uint64_t myVariable;
```

- **Declaration**
  - declared outside any procedure - global scope.
  - visible (accessible) throughout program execution.

- **Allocation**
  - done once at compile time in the data segment.
  - space for variables is allocated in the order in which they appear.
  - `allocated_memory` holds the size (in byte) of the currently allocated memory in the data segment.

Local Variables

```
uint64_t func() {
    uint64_t myLocal;
}
```

- **Declaration**
  - declared inside a procedure - **local scope**.
  - visible (accessible) within the procedure it is declared in.
- **Allocation**
  - space is allocated upon each procedure call at runtime on the stack.

Parsing Variables

- Parsing a **declaration** in selfie
  - allocate space for 1 value of given type

    global: increase `allocated_memory` by `1*REGISTERSIZE`
    (variables in selfie are of same size)
    = reverse bump pointer.

    local: decreasing the value of stack pointer
    (this is part of the procedure prologue which is discussed later).
  - introduce mapping from name to memory location.

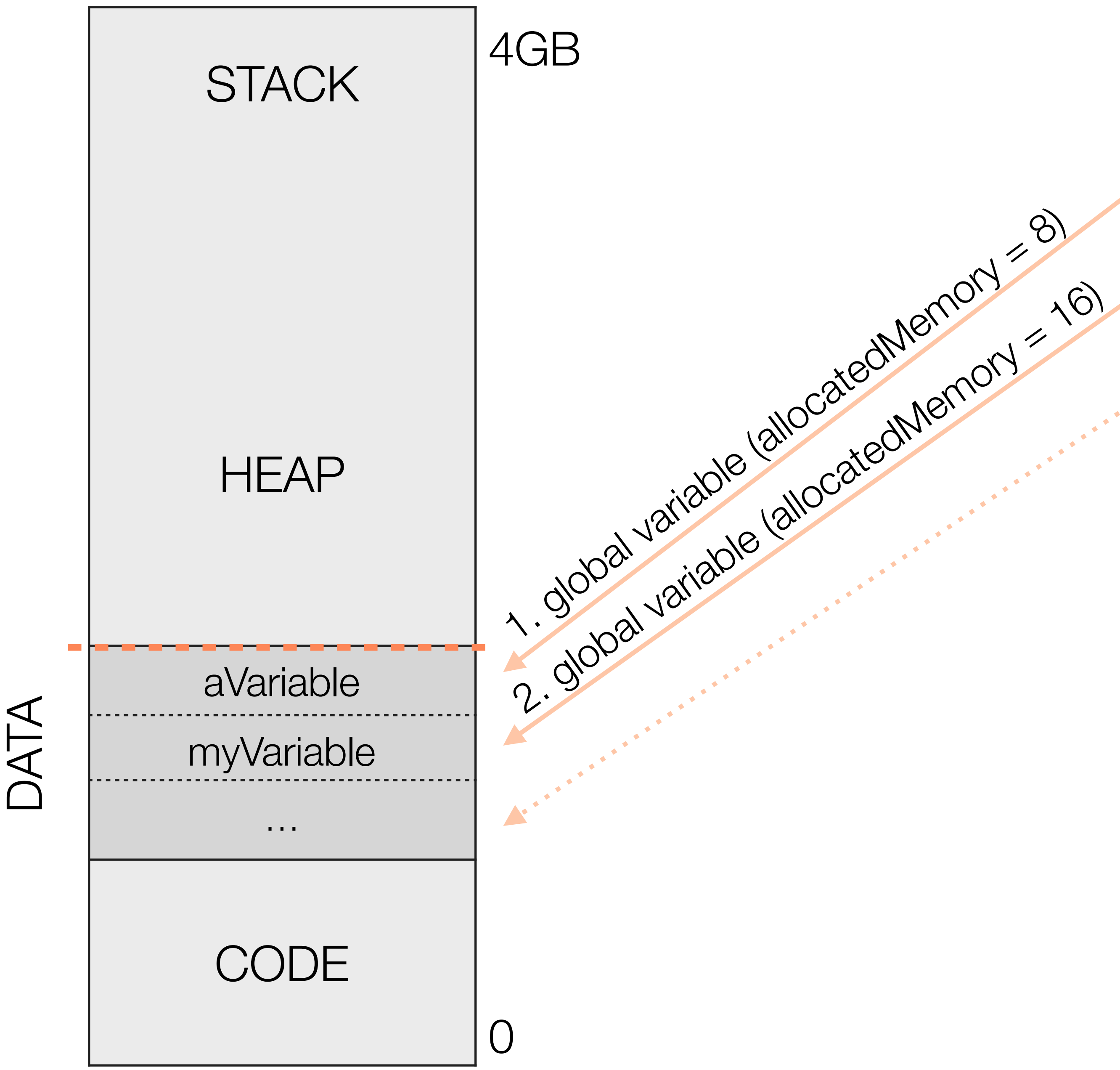
Parsing Variables

```
uint64_t aVariable;

uint64_t myVariable;

uint64_t main() {

  myVariable = 2;

  aVariable = myVariable

}
```

Declaration
allocate memory and
introduce mapping
(address know)

Access

Parsing Variables




```
uint64_t aVariable;
uint64_t myVariable;
...

uint64_t main() {

  myVariable = 2;

}
```

Parsing Variables

- Parsing an **access** in selfie
  - to access a variable (i.e. to get the value associated with it) the memory address of that variable is needed.
  - therefore we need a way to remember the previously introduced mapping from name to address.
  - a data structure called symbol table is used to remember these mappings.
- A variable that appears on the left-hand side of an assignment represents a **storage location** whereas a variable that appears on the right-hand side refers to the actual **value** associated with it.

lvalue and rvalue

lvalue

```
myVariable = myVariable;
```

rvalue

- **lvalue**
  - left-hand side of assignment.
  - a storage location (address).
  - when parsing the address is loaded into a register.

- **rvalue**
  - right-hand side of assignment.
  - value in traditional sense.
  - when parsing the address is loaded and dereferenced to retrieve the actual value from memory.

Parsing Variables

```
uint64_t aVariable;

uint64_t myVariable;


uint64_t main() {

   myVariable = 2;

   aVariable = myVariable

}
```

Declaration
allocate memory and
introduce mapping
(address know)

Access
address not known
therefore we need to
remember the mapping in
the symbol table

Symbol Table

- <u>Symbol Table</u>
- Provides 2 main functionalities
  - **mapping**
    associates a name with an entities so we can refer to it by its name.
  - **scope**
    enables us to have names with different scope (global, local).
- data structures used in selfie - hash table and singly linked list.
- stored on the heap (using `malloc()`).

Symbol Table

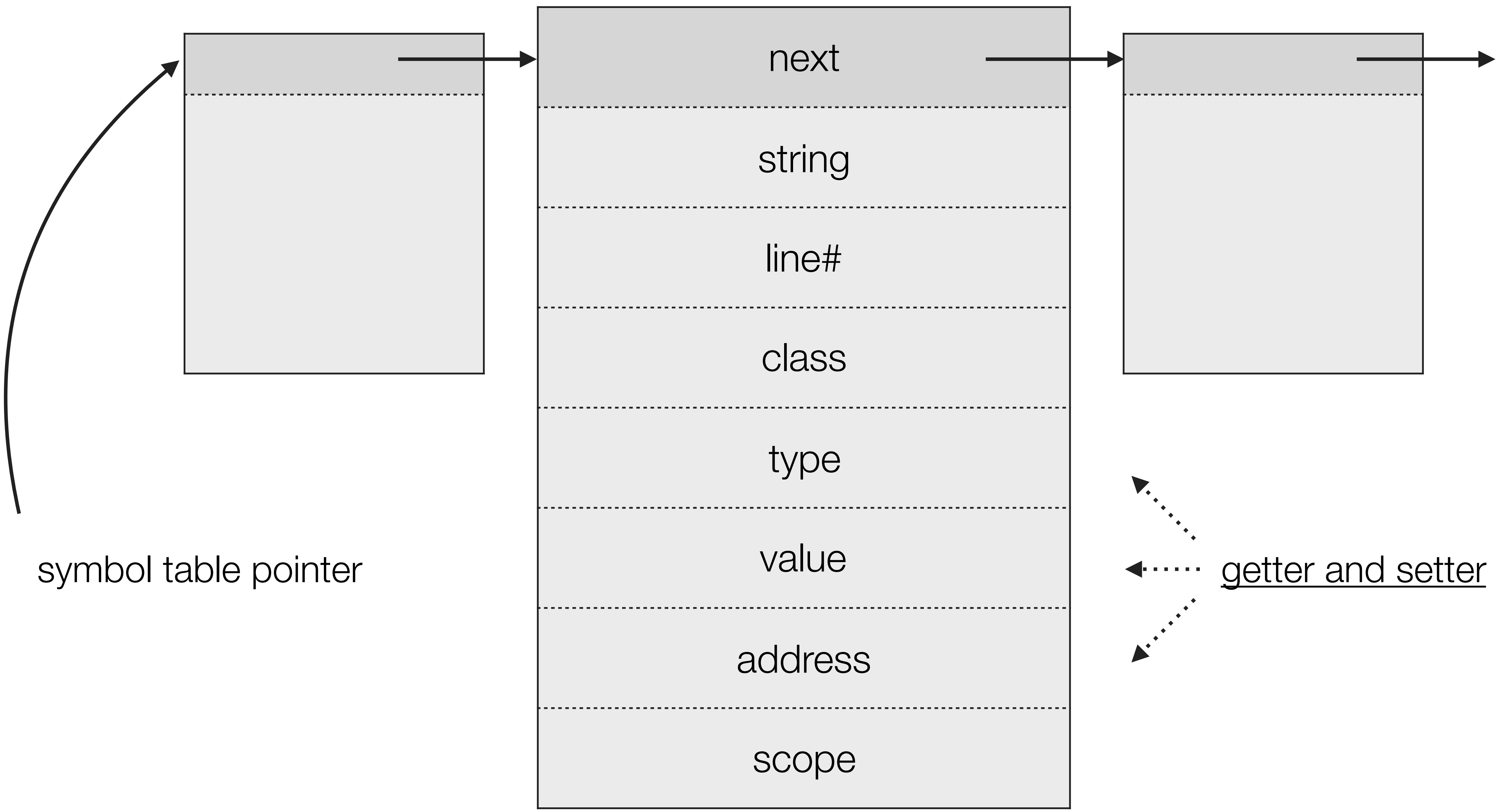
Why on the heap?

- the symbol table does not conform to the semantics of the **stack**
  - lifetime is not stack like - no in-order (de)allocation
- a symbol table of dynamic size cannot be stored in the **data segment** (static size)
  - a feature of our symbol table is unbound size (idea)
    (a fixed-size symbol table could be stored in the data segment)
  - the only other choice is to store the symbol table on the **heap**

Symbol Table Entry (as a list element)

`./selfie -c`

- symbol table in selfie



Hash Table

- a hash table is a data structure similar to an **array…**
  - it can be seen as array of buckets, each holding data
  - each bucket is uniquely identified by an index
- … but it uses a **different access method**
  - each data value is associated with a key value
  - a technique called hashing is applied to convert a key value into an index using a hash function
- **searching** values in a hash table (computing the index) can be done in **constant time** (iff collision free) whereas the search complexity for arrays is linear in the number of entries

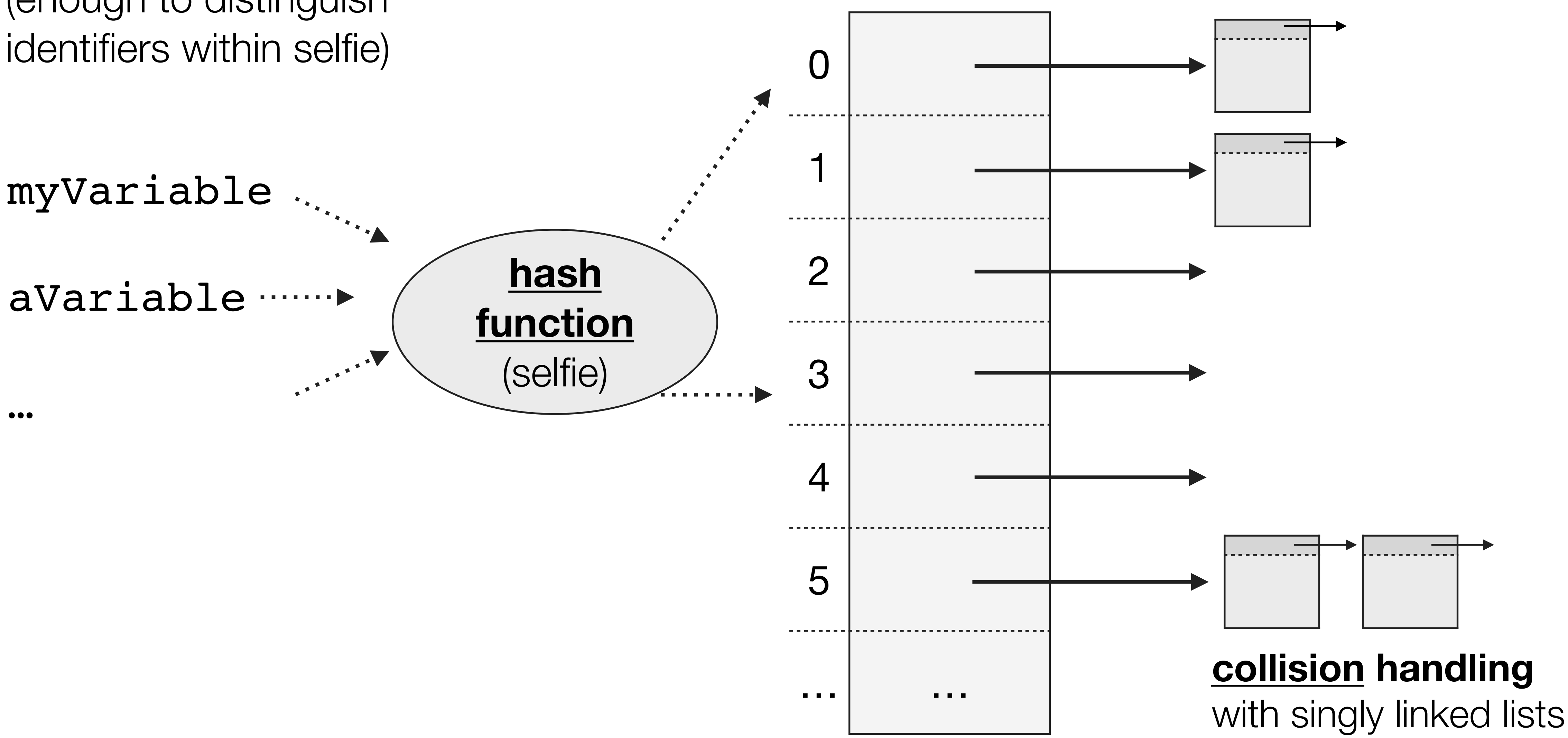
Hash Table

`./selfie -c`

keys
8 character prefix of variables' names
(enough to distinguish identifiers within selfie)

hash table
array of pointer to lists

symbol table entries
now many small list instead of one long one.



Symbol Table API

- insert and search
- getter/setter for adding an element
- **singly linked list:**
    - insert complexity - constant time
    - search complexity - linear in the number of entries
    - **BUT** below 1000 list elements linear is OK
- **hash table**
    - insert complexity - constant time
    - search complexity - constant time iff collision free (worst-case still linear in the number of entries)

Symbol Table Scope

`./selfie -c`

- names can have different scope (visibility) and therefore they do not need to be unique and can be shadowed

- In selfie we simply use 3 symbol tables - `get_scoped_symbol_table_entry`
  - **local symbol table** (singly linked list)
    first search local table (local variables shadow global variables
  - **library symbol table** (singly linked list)
    library procedures override defined/declared procedures in global symbol table. This table is used for bootstrapping syscalls using wrapper code.
  - **global symbol table** (hash map)
    search last if entry not already found

- Due to its enormous size the global symbol table is implemented as a hash table (speed up for `make` target of 30%)

Back to Variables

```
uint64_t aVariable;
uint64_t myVariable;

uint64_t main() {
  myVariable = 2;
}
```

next	next
string	myVariable
line#	2
class	VARIABLE
type	uint64_t
value	0
address	-16
scope	**GLOBAL_TABLE** ($gp)

- remember **negated offset** = the value of `allocated_memory`
- enough to compute the address from global pointer and offset

Back to Variables

```
uint64_t aVariable;

uint64_t main() {
   uint64_t myVariable;

   myVariable = 2;

}
```

next	next
string	myVariable
line#	4
class	VARIABLE
type	uint64_t
value	0
address	-8
scope	**LOCAL_TABLE** ($fp)

- compute address relative to frame pointer using offset
  - parameters have **positive**, variables have **negative** offsets
  - more on local variables when we discuss procedures

Variables Conclusion

- The examples so far show variables that hold a single value (`uint64_t my_variable`, `uint64_t* my_pointer`)

- A variable can also refer to multiple values
  - array
  - struct

Data Structures and Data Types

Arrays and Structs

Array and Struct

- Selfie does not support arrays or structs.
- However, this section should give you a good understanding of arrays and structs and the modifications necessary to implement support for global and local arrays and structs in selfie.
- We will therefore look at basic concepts, but also discuss details concerning the implementation of global arrays and structs in selfie.
- This should help you understand how data is stored in memory.

Data Structure

A data structure is a way to ***organize a collection of data*** *in memory that simplifies access and modification of that data.*

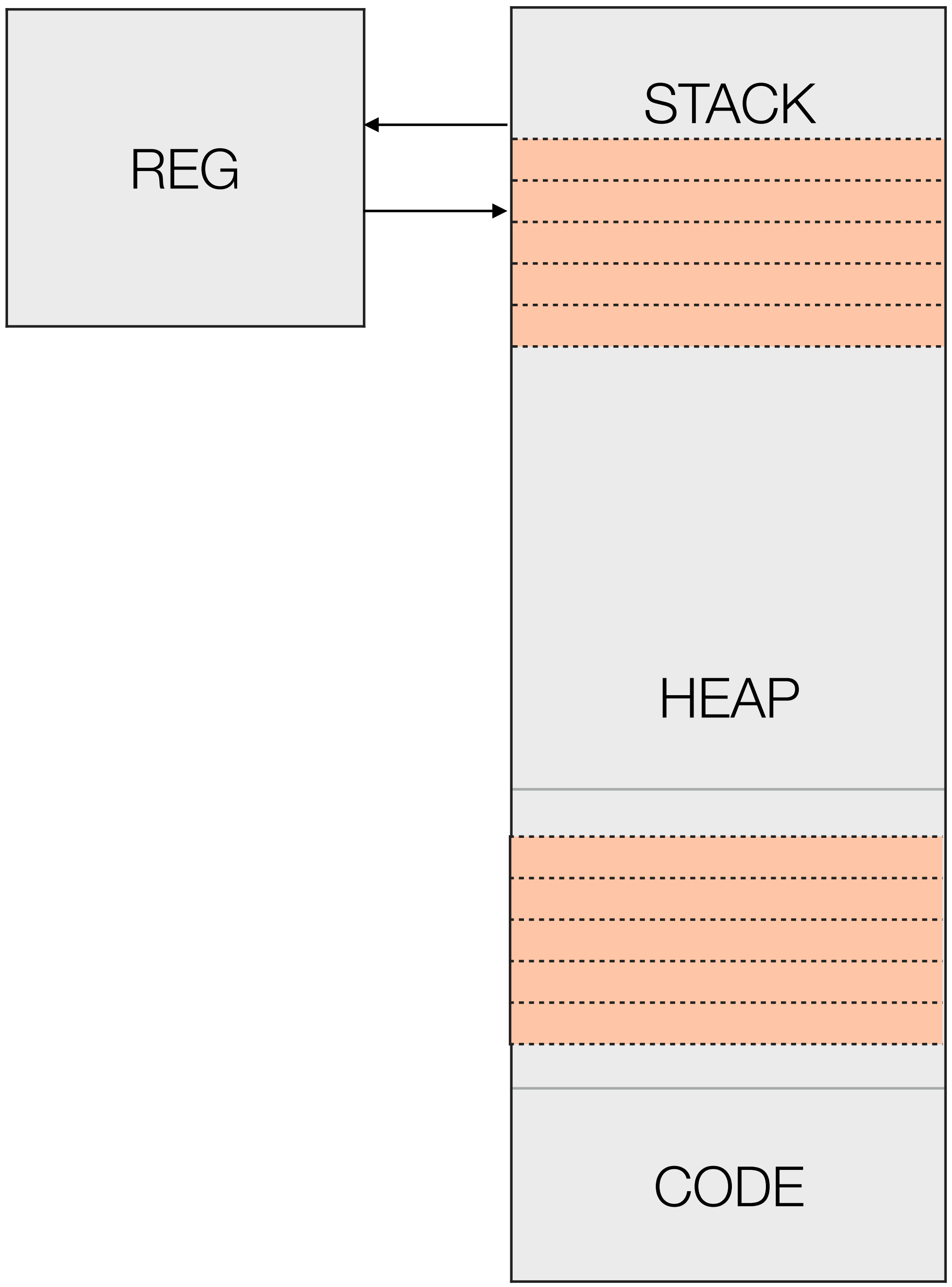
- defines relationship between data values
- defines functions/operations on the data structure
- some data structures: static arrays, dynamic arrays, linked lists, stack, …
- we will only consider **static arrays**
  (referred to as **array**)

Array

*An **array** is a **variable** that is associated with a fixed number of values of the same data type. An element of the array is selected using an index.*

- container for multiple values of the same data type
- the length is fixed when the array is created
- the index to access an element an array is computed at runtime
- arrays can be implemented by using contiguous space or a pointer structure

Array



- Arrays declared within procedures are **local** to that procedure.
  → Space is allocated on the stack at runtime (no pointers to arrays)

- Arrays declared outside any procedure are **global**.
  → Space is allocated once in the data segment at compile time

- From now on we will only consider **global arrays** implemented as **contiguous space** of addresses.

“Why do we want to support arrays in a programming language?”

“To answer this question we will take a look at the first two of three* metrics a we as computer scientists are generally interested in.”

time

space

energy

* there are others, but for us these are the most important

Time

- **The big advantage is constant time access** when the array is implemented as contiguous space of addresses.

```
uint64_t x;
uint64_t a[10]

i = 10;
x = a[i];
```

value at (address of a + i * size of element type)

- **The access is constant time because**
  … we assume that the operations to compute the address can be done in constant time.

Space

- **The downside is the potential for <u>fragmentation</u>** when the array is implemented as contiguous space of addresses.

- **The problem with fragmented memory…**
  - We request space of size `n`.
  - Take `n = 6` for the example below.
  - There is enough free space (> `n`), but there is **not enough contiguous free space** to accommodate a block of size `n`, so the request fails.

Space

- **Conditions that create fragmentation are…**
  - allow allocating space of different size together with deallocating out of order
    → no fragmentation on stack
- Using **lists** would solve the problem of fragmentation, since the nodes have the same size, but
  - we would have drawback in **size** (pointer structure)
  - and we would loose constant **time** access.

Array

```
uint64_t myVariable;
uint64_t myArray[5];

uint64_t main() {

  myArray[1] = 2;

}
```

Declaration

Access

Array Declaration

base address

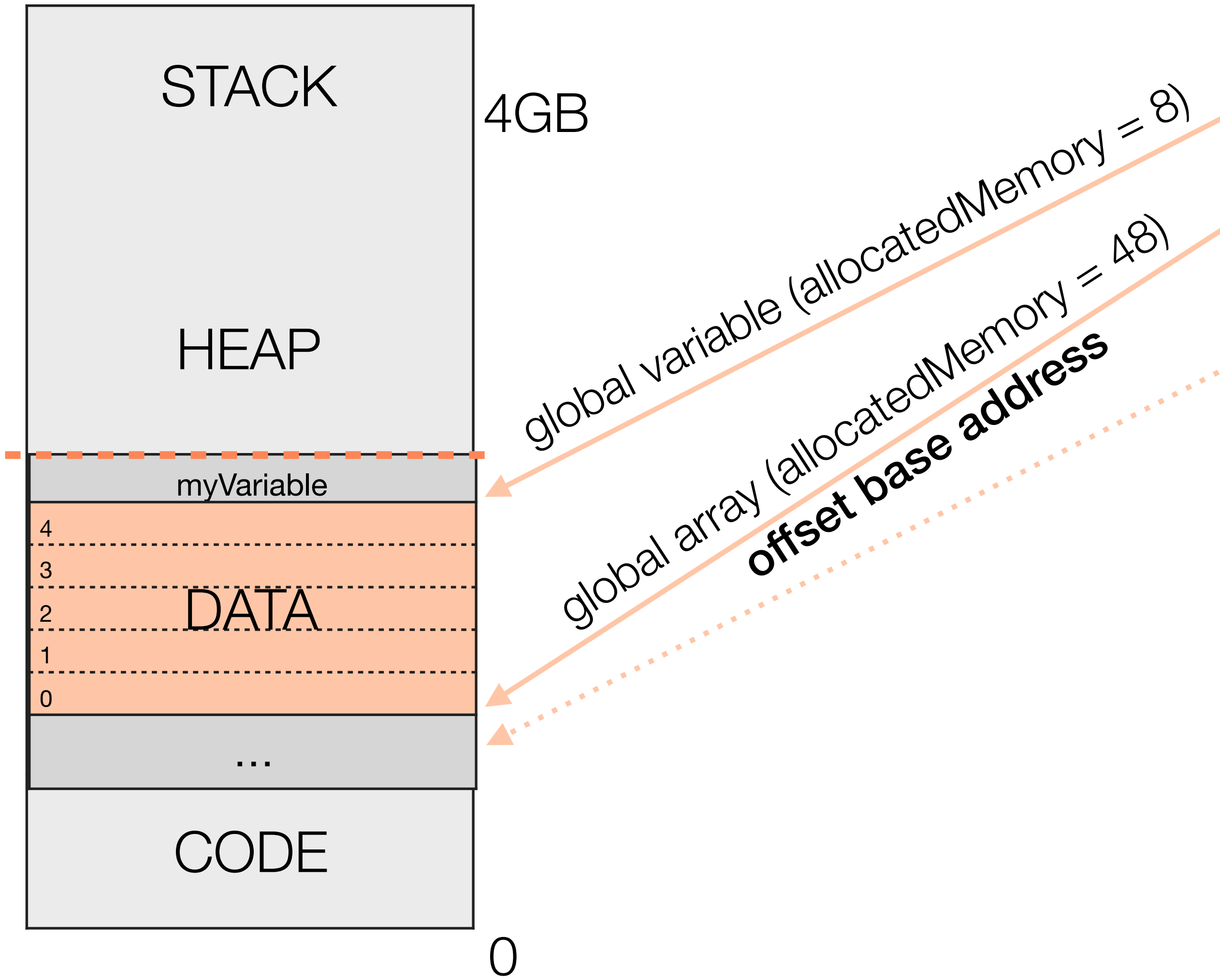
```
uint64_t myArray[5];
```

size

- **declare a variable and allocate space**
  - instead of allocating space for 1 value (as before) we allocate **contiguous** space for 5 values of given type
- **calculating size of array** - `myArray[2]…[4]`
  - size must be given using constant values
  - **formula**: 2 * … * 4 * size of element type

Array Declaration

- Arrays are organized **upside-down** (conform to C standard)




```
uint64_t myVariable;
uint64_t myArray[5];
...

uint64_t main() {

   myArray[1] = 2;

}
```

Selfie Declaration

```
uint64_t myArray[5];
```

What is new?

- **scanner** does not know `'['`, `']'`
  → new characters
  → new symbols
- **parser** does not recognize the selector syntax
  → new parsing procedure
- **symbol table** can not handle variables of size other then `REGISTERSIZE`
  → modify symbol table, store additional information
- **no code generation** for declarations

Array

```
uint64_t myVariable;
uint64_t myArray[10];

uint64_t main() {

  myArray[1] = 2;

}
```

Declaration
compiler allocates memory

Access

Array Access

base address

```
myArray[1] = 2;
yourArray[x] = 3;
```

positive offset into the array

- **access a value within the array**
  - using integer constants or integer variables in between brackets
  - address needs to be calculated from the base address and the offset into the array at runtime
  - like dereferencing using the star operator
- **calculating the address** - `myArray[2]…[4]`
  - the offset into the array has to be calculated at runtime
  - the code to do so is generated at compile time

Selfie Access

```
myArray[1] = 2;
yourArray[x] = 3;
```

What is new?

- **scanner** does not know `'['`, `']'`
  → new characters
  → new symbols

- **parser** does not recognize the selector syntax
  → new parsing procedure

- **code generation** to implement behavior according to specification
  → calculate address
  → load address (lvalue) or value (rvalue)

Array

```
uint64_t myVariable;
uint64_t myArray[10];

uint64_t main() {

  myArray[1] = 2;

}
```

Declaration
compiler allocates memory

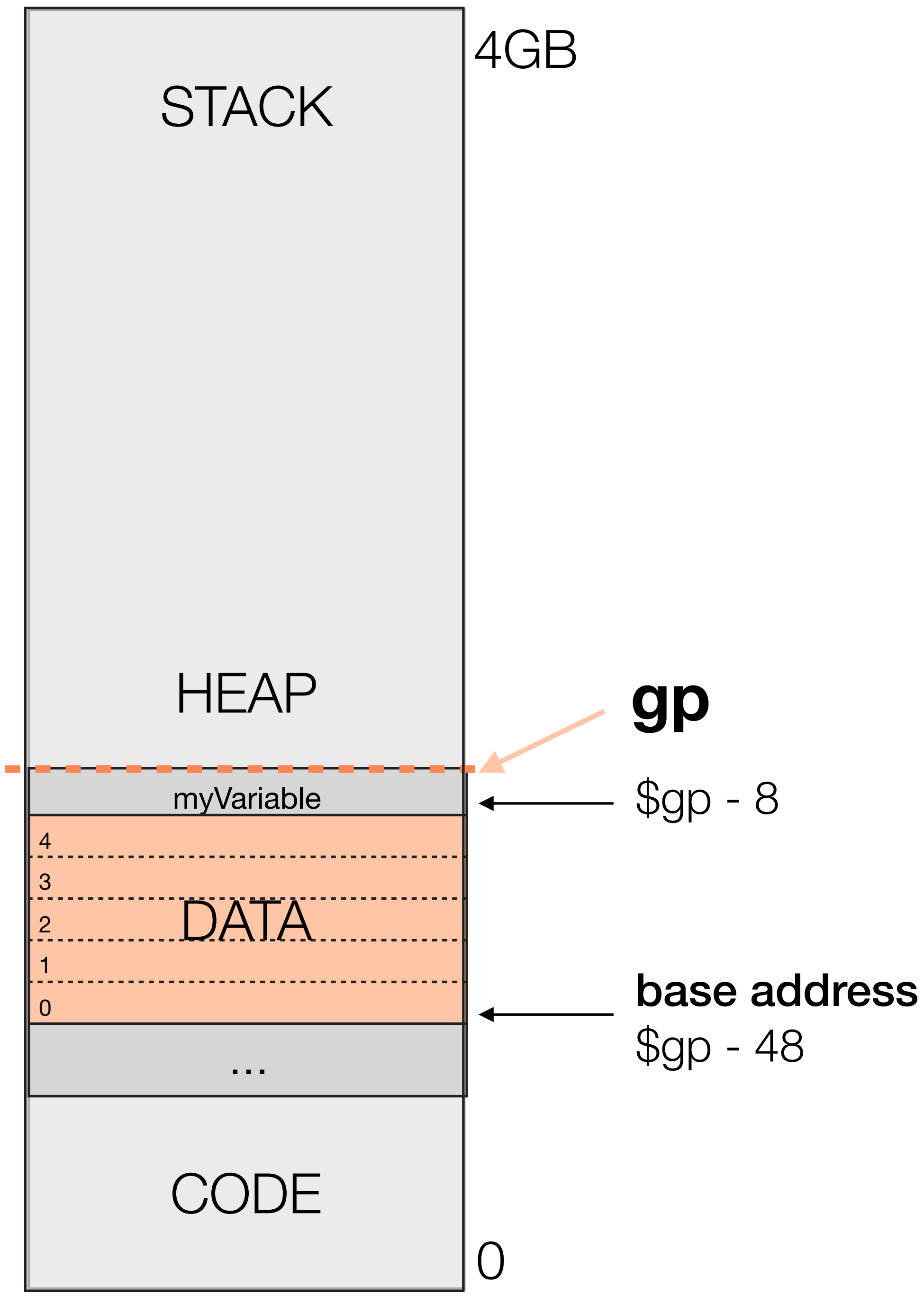
Access
calculate address at runtime

Array Access and Pointer Arithmetic

```
myArray[1] = 2;
*(myArray + 1) = 2;
```

- **equivalent semantic**
  - load address of myArray then load `1`
  - array semantics/pointer arithmetic - add `1 * REGISTERSIZE` → address
  - store 2 at that address
- depending the implementation even the emitted code might be equivalent

Calculate Address



base address + offset into array

gp - offset relative to gp

gp - 48 + 1*REGISTERSIZE

Calculate Address

- the **offset into the array** has to be calculated and added to the base address

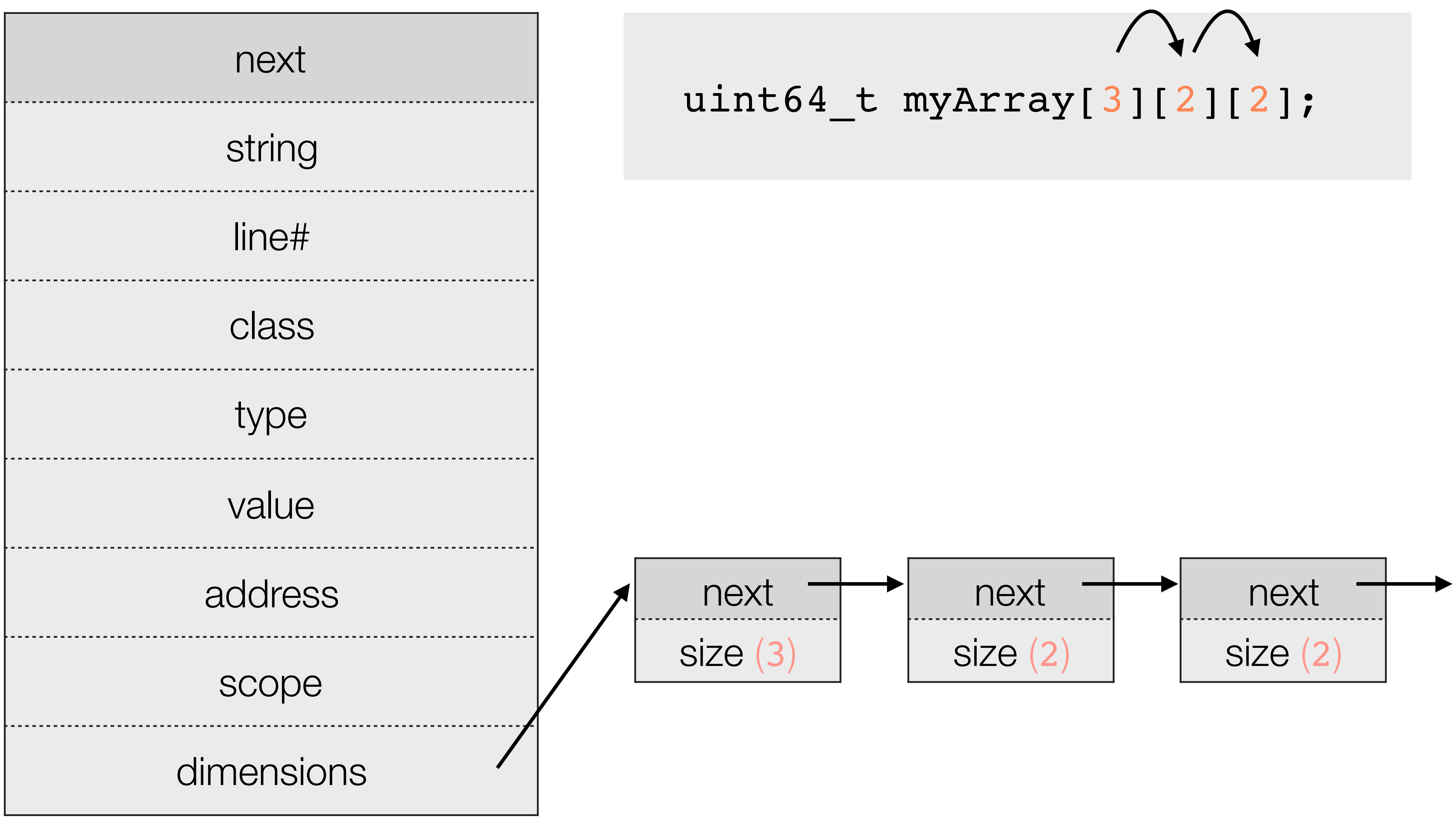
- therefore we need to **remember the structure** of the array
  - number and size of dimensions
  - in the symbol table

- storing multidimensional arrays in one-dimensional memory we can either use <u>row-major or column-major order</u>

- **IMPORTANT:**
  address calculation is done at **runtime**, because variables might be used to access the array.
  (access involving only constant values could be optimized)

Symbol Table Entry

- store the information **while parsing** the selector
- this can be done in multiple different ways (order, …)



Row-Major

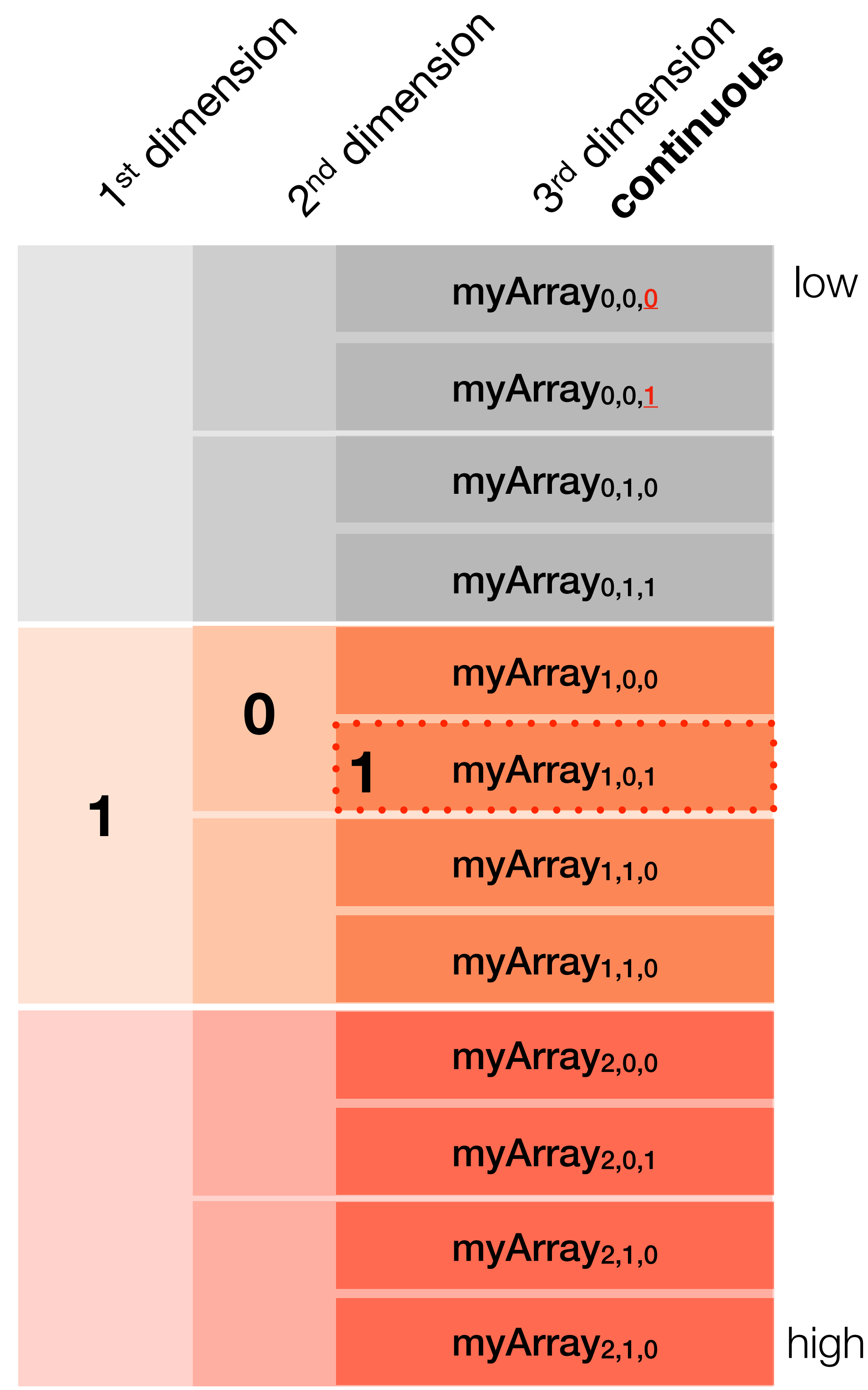
- row-major

```
uint64_t myArray[3][2][2];
```

```
myArray[1][0][1];
```

Address Calculation

$$1 * (\underbrace{2 * 2) + 0 * 2}_{\text{size of lower dimensions}} + 1$$



Column-Major

- <u>column-major</u>

```
uint64_t myArray[3][2][2];
```

```
myArray[1][0][1];
```

Address Calculation

$$1 + 0 * 3 + 1 * (3 * 2)$$

size of higher dimensions

low

myArray$_{0,0,0}$

myArray$_{1,0,0}$

myArray$_{2,0,0}$

myArray$_{0,1,0}$

myArray$_{1,1,0}$

myArray$_{2,1,0}$

myArray$_{0,0,1}$

myArray$_{1,0,1}$

myArray$_{2,0,1}$

myArray$_{0,1,1}$

myArray$_{1,1,1}$

myArray$_{2,1,1}$

high

1st dimension **continuous**

2nd dimension

3rd dimension

Assignment 4: [ ]

- Students can extend the language C* and implement (multi-dimensional) arrays.

- Hint: This is a big assignment so think before you code.
  - look at the next two slides for **tips**
  - only do one-dimensional arrays (at first)
  - everything you need to know/do should be in the previous slides

Assignment 4 Implementation Tips

- **Divide and conquer**

1. **Syntax**
   modify compiler to support **parsing** declaration and access without generating code

   - new parsing procedure(s) for selector syntax `'[..]'`

   - test on examples

2. **Information** (could be part of 1.)
   modify symbol table and parser and store the necessary **information**

   - store dimensions and size

3. **Semantics**
   modify compiler to **generate code** for access computation

   - use information from symbol table to emit instructions in the right places to enable address calculation at runtime

   - test on examples

Assignment 4 Implementation Tips

- keep in mind the **different semantics** of the selector for
- **declaration**
  - no code generation
  - only symbol table entry
  - calculate size of array
- **access**
  - code generation
  - calculate address

Compile Time and Runtime

- **declaration** is a **compile time** concept
  - create symbol table entry and allocate memory
- **access** is a **runtime** concept
  - the address calculation can only happen at runtime (variables)
  - at runtime there is code that computes the address
  - this code is emitted at **compile time** using the selector currently parsed and symbol table information

Array Conclusion

`./selfie -c`

- Looking at the implementation of the `power_of_two_table`, `SYMBOLS`,... now, you should realize and understand
  - practically a **one-dimensional arrays**
  - implemented by a pointer to a **block of addresses** (malloced) and accessed using **pointer arithmetic**
- Using arrays instead does not change the semantics but the **syntax.**

Array Conclusion

- **Advantages**
  - With arrays we have a **convenient way** to allocate space for multiple values and access them using indexes
    (no external pointer arithmetic)
- **Disadvantages**
  - An array cannot hold values of **different type** without type casting
  - Access to an array is **not “safe”**. Without checking bound, illegal access is possible. (access outside arrays)

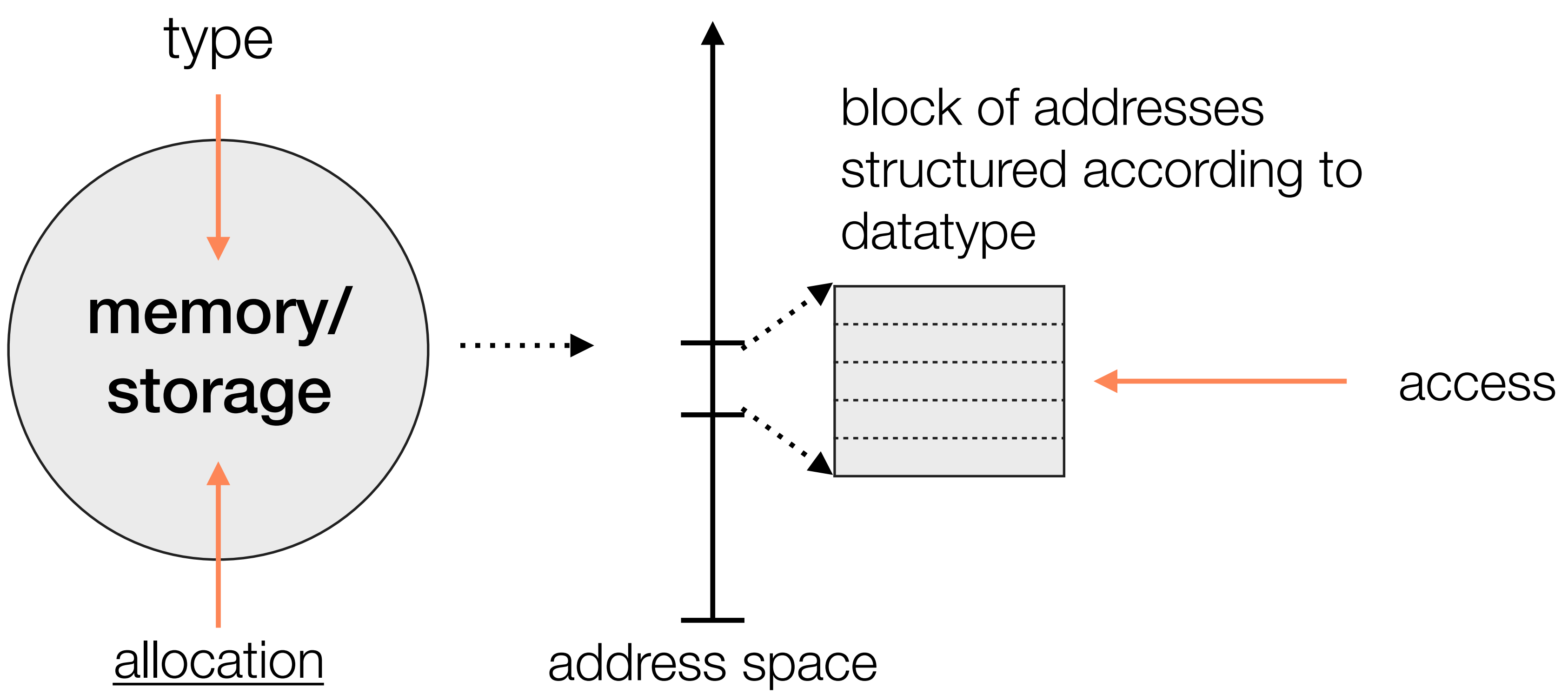
Data Type

*A data type defines the type and structure of a data item and tells the compiler (at **compile time**) how the programmer intends to use.*

- defines operations on a variable
- defines the domain and size of a variable
- primitive data types: uint64_t, uint64_t*
- composite data types: struct

compile time information

Data Type



- The **data type** of a variable defines the structure of data in memory and tells the compiler how the programmer intends to use that variable.

- Different data types use different **access** strategies that have different properties.

Struct also Record, Object in Java

*A **struct type** is a user defined composite **data type** that allows to combine data elements (variables) of different types under one name/identifier.*

*A **struct** is an instance of such a struct type. It is a ordered collection of variables called fields or members that store data.*

- a field is identified by a name/identifier
- a field can be of any data type - composite or primitive
- a struct instance is implemented as contiguous block of memory

We could ask the same Question again:

“Why do we want to support structs in a programming language?”

“To answer would be the same as it was for arrays

>time and space<

But here another important concept comes into play

>*compiler awareness**< ”

* no need to understand this now

Using Structs

order not optional

1. declare struct type (optional)

```
struct record_t;
```

2. define struct type

```
struct record_t {
    struct myStruct* field1;
    uint64_t field2;
};
```

3. declare instance of struct type
   we only discuss **pointers to structs**

```
struct record_t* record;
```

4. allocate space for struct

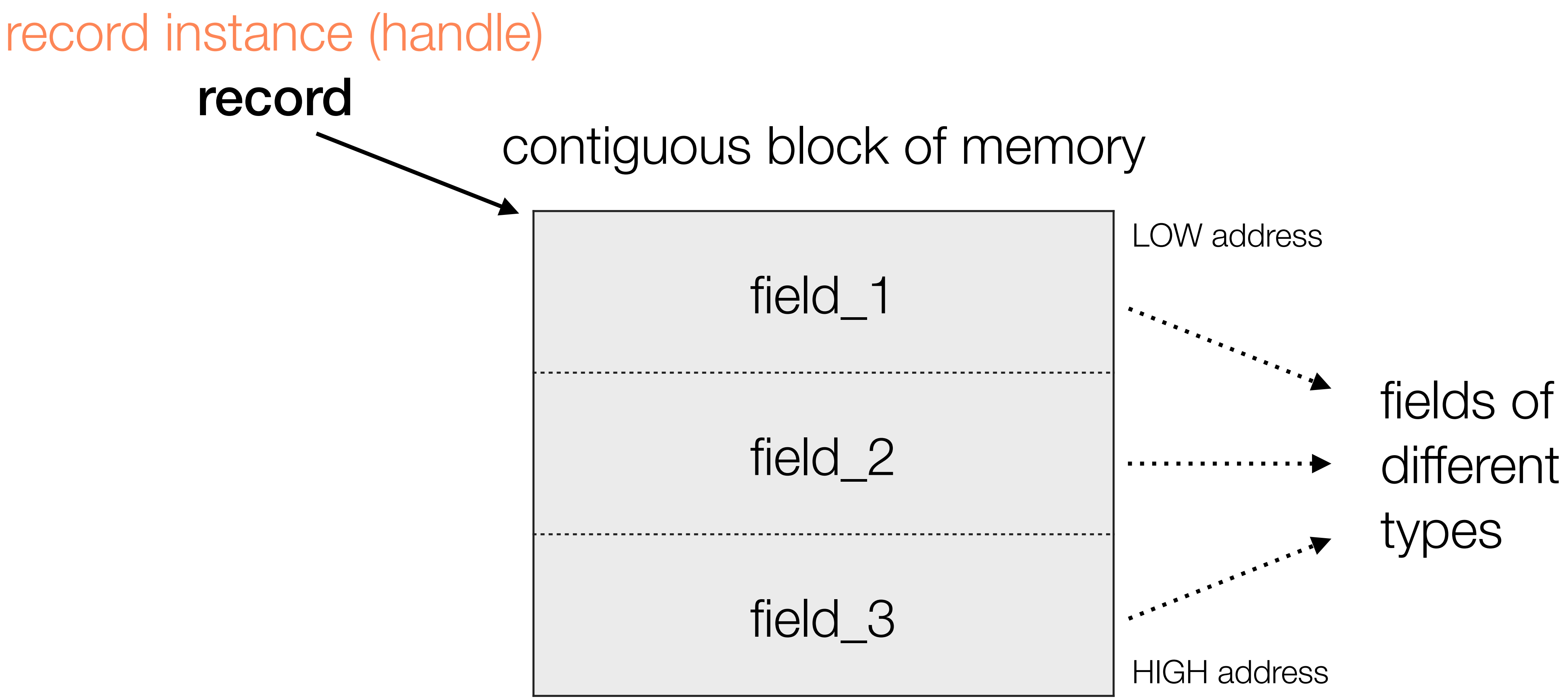
```
record = malloc(16);
```

5. access struct

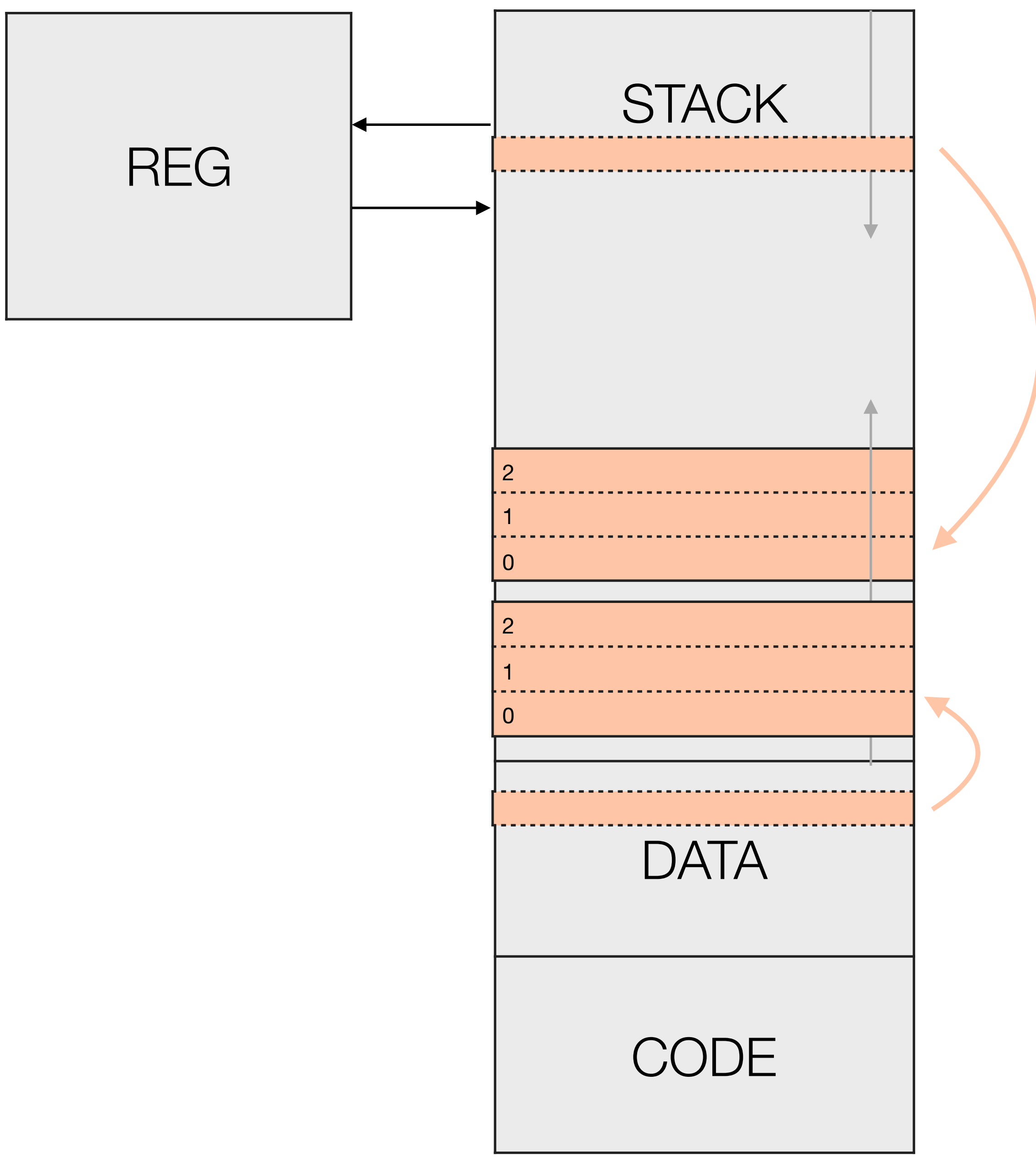
```
record -> field2
```

Struct Instance (abstract)

- we only discuss **pointers to structs**



Struct



- Pointers to structs can be declared **locally** and **globally.**

- The pointers will be put on the **stack** (local) or in the **data** segment (global).

- Before the struct can be accessed memory has to be allocated (malloc) by the programmer on the heap.

  → **Every struct is put on the heap and is organized up-side down**

Struct Definition Type

new type name

struct keyword

```
struct record_t {
    struct myStruct* field1;
    uint64_t field2;
};
```

type and name of fields

- **Defining a new struct type**
  - name/identifier for new type
  - name/identifier and type of each field
  - no code generation

- To declare a variable of this type later, we need to store and manage this <u>type information</u>

Struct Definition Type Information

new type name

struct keyword

```
struct record_t {
    struct myStruct* field1;
    uint64_t field2;
};
```

type and name of fields

- we need a data structure that is globally visible to store type information → use symbol table to store:
    - name/identifier of new type
    - associate new type with the **types structure**
- type structure is necessary to access fields of the struct
    - field name/identifier
    - field type
    - field <u>offset</u> within the struct

Offset

- The offset of a field is the accumulated size of the fields that come before it
  - offset of first field is always `0`
  - here offset of second field is size of `field1`

offset 0

offset 8

```
struct record_t {
    struct myStruct* field1;
    uint64_t field2;
};
```

Struct Definition Symbol Table Entry

`./selfie -c`

- Until now we only stored "simple" variables and array variables in the symbol table.

- Storing type information requires us to be able to distinguish between variables and types

    - this can achieved by **labeling** the entry
      (it actually is already done using the `class` field of the entry)

- **Classes**

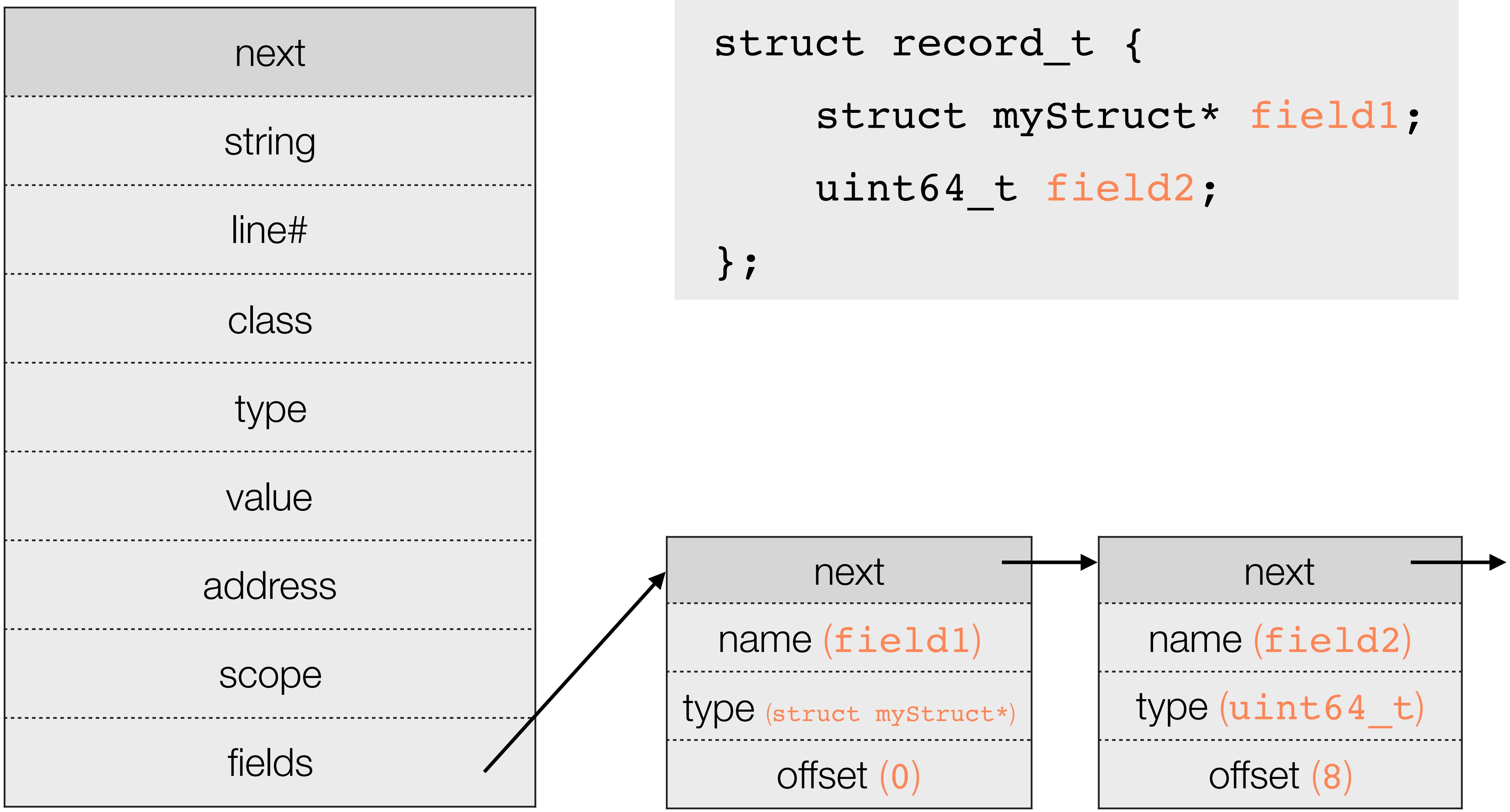
    - besides variable information the symbol table is used to store bigints, strings and procedure information
      (classes: `VARIABLE`, `BIGINT`, `STRING`, `PROCEDURE`)

    - we can introduce a new class `TYPE`

Struct Definition Symbol Table Entry

- store the information **while parsing** the definition
- one possibility to do so

```
struct record_t {
    struct myStruct* field1;
    uint64_t field2;
};
```



Selfie Definition

```
struct record_t {
    struct myStruct* field1;
    uint64_t field2;
};
```

What is new?

- **scanner** does not know "`struct`" keyword
  → new keyword
- **parser** does not recognize this syntax
  → new parsing procedure for struct definition
- **symbol table** can not store type information
  → modify symbol table, store type information
- **no code generation** for definition

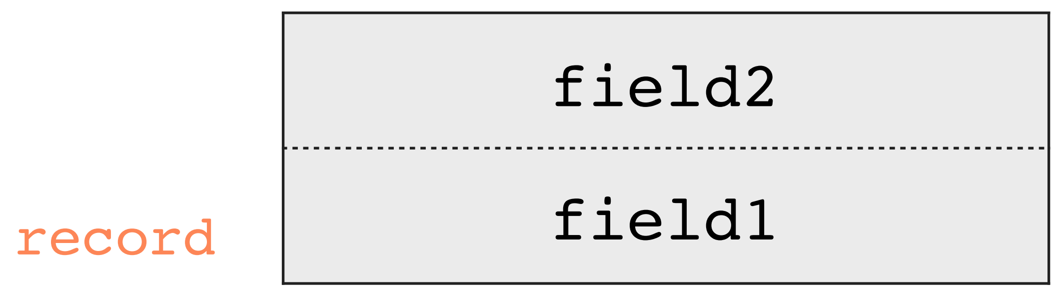
Struct Declaration in C

declaration

```
struct record_t record;
```

access

```
record.field2
```



address of record equals address of field1

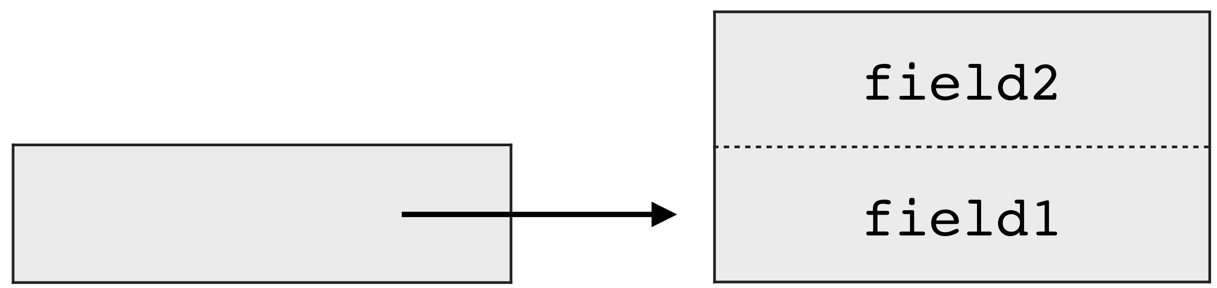
- **Declaring a variable of type** `struct record_t`
  - the variable **is** this type
  - the size of this variable is the accumulated size of its fields
  - accessed using the "`.`" notation

Struct Declaration in C

declaration
```
struct record_t* record;
```

access
```
record -> field2
```

record
(pointer)



the **value** of record equals the address of field1

- **Declaring a variable of type** `struct record_t*`
  - variable is **a pointer** to an instance of type `struct record_t`
  - the size of this variable (a pointer) is one word (`REGISTERSIZE`)
  - accessed using `'->'` notation
- We only consider **pointers to structs**

Selfie Declaration

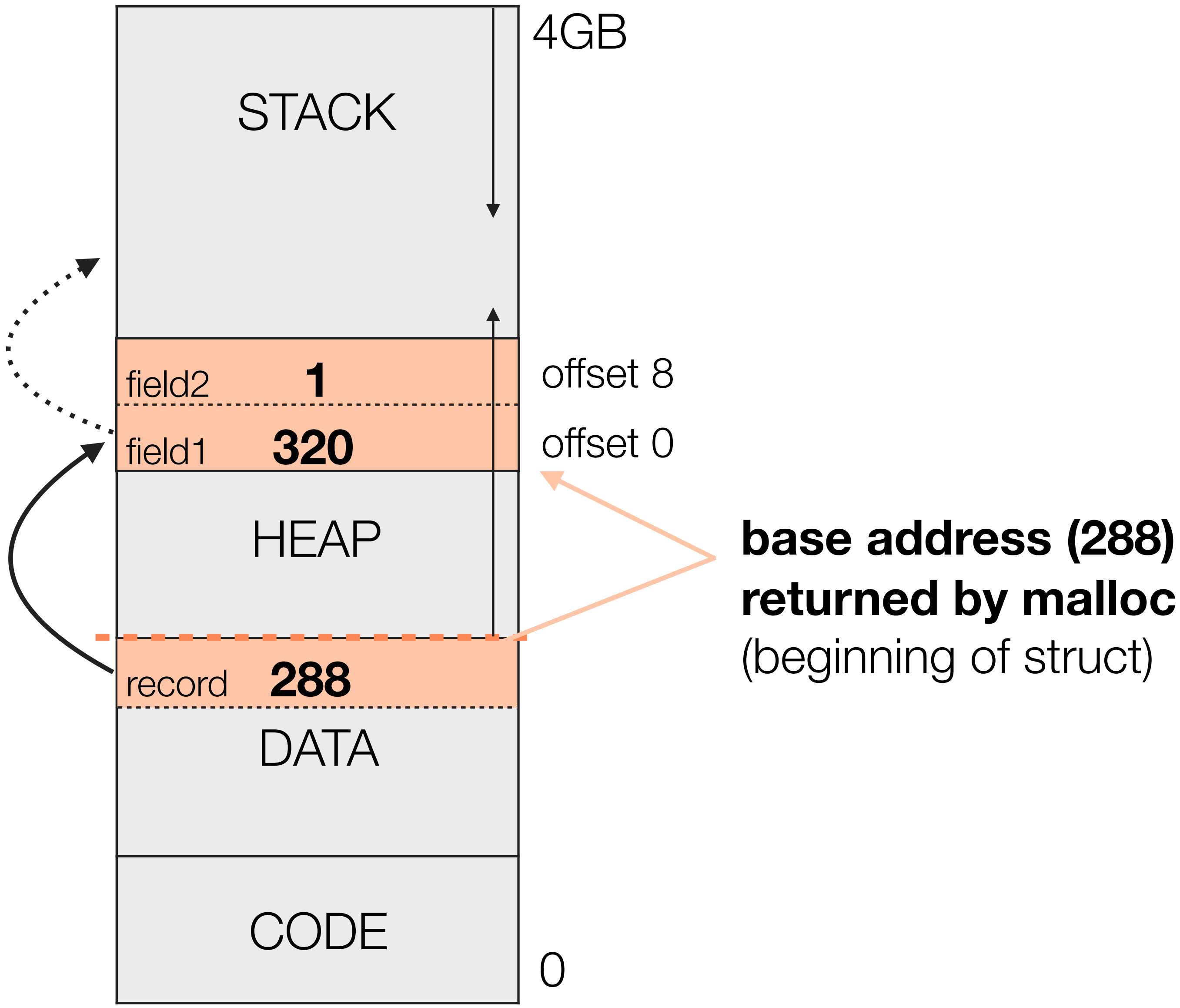
`./selfie -c`

```
struct record_t* record;
```

What is new?

- **scanner** does not know '`struct`' keyword
  → new keyword
- **parser** does not recognize the struct types
  → parse new type
- **symbol table** can not define variables of type "struct type"
  (struct types are represented by symbol table entries)
  → modify the symbol tables <u>type</u> field so it can refer to "struct type"
- **no code generation** for declarations

Allocate Space for Struct




```
record = malloc(16);
```

- allocate space for a struct on the **heap**
- size of the struct is the accumulated **size of its fields**

Struct Access

runtime information
base address

```
record -> field2
```

compile time information
offset

- **access a value within the struct**
  - use field name/identifier
  - address is calculated from the base address and the offset of the field inside the struct at runtime
- **calculating the address**
  - the offset of a field from the beginning of the struct is known at compile time
  - the base address of the struct returned by malloc is not known before runtime
  - code is generated to add the offset to the base address at compile time

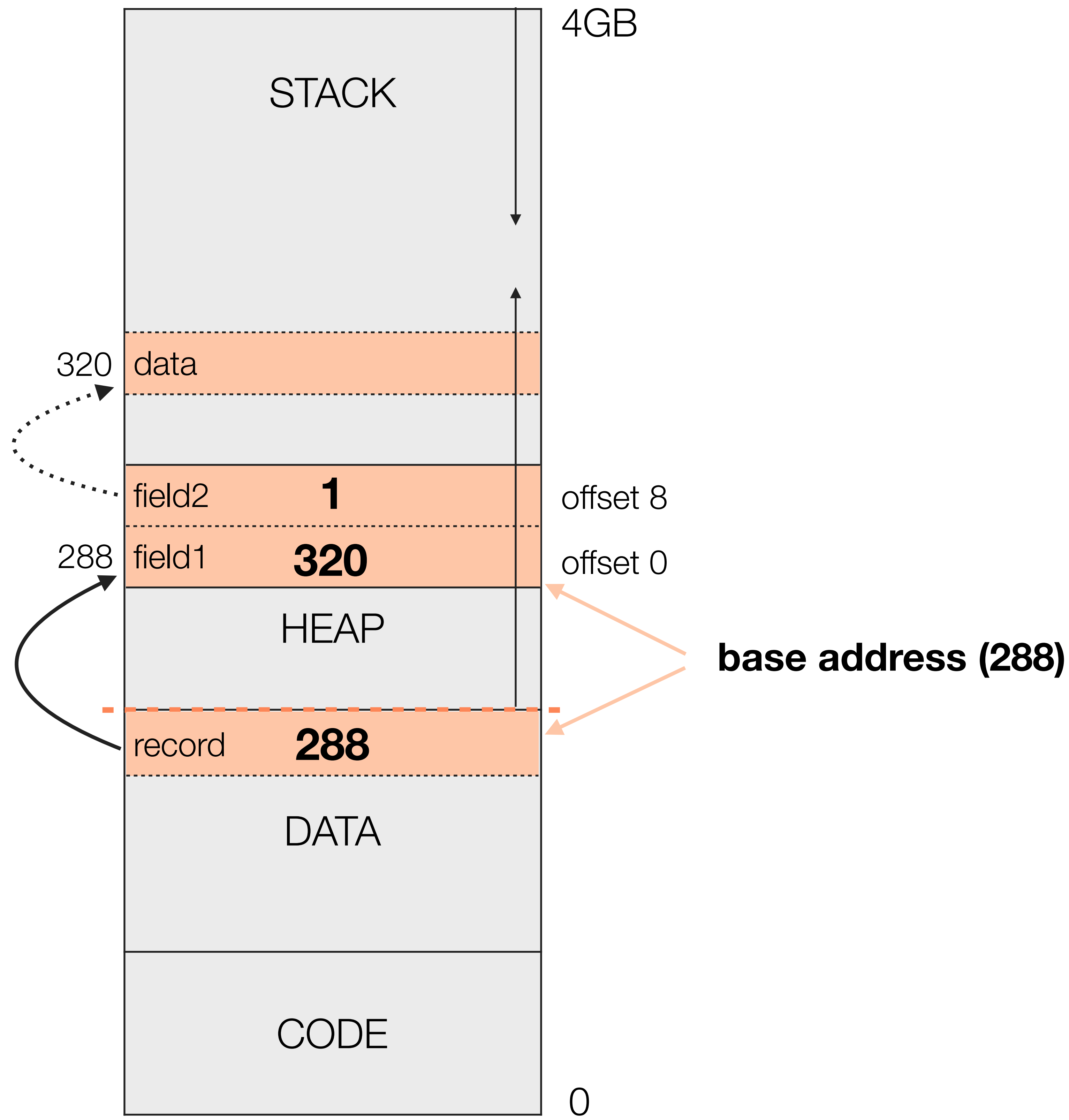
Selfie Access

```
record -> field2;
```

What is new?

- **scanner** does not know '`->`'
  → new symbol
- **parser** does not recognize the selector syntax
  → new parsing procedure
- **code generation** to implement behavior according to specification
  → calculate address
  → load address (lvalue) or value (rvalue)

Struct Access Example



Structs Code Generation

1. 2.

```
record -> field2
```

1. **load base address of struct** (`record`)

```
ld $t0 -8($gp)
```

2. **add offset of field** (`field2`) to the beginning of the struct

```
addi $t0 $t0 8
```

→ now `$t0` holds address (not value) of field `(field2)`

3. optional: **dereference** to get value

```
ld $t0 0($t0)
```

Structs within Structs Code Generation

1. 2. 3. 4.

```
record -> field1 -> data
```

Assumption:
Before we see an arrow the register contains the address (not the value) of the field.

1. **load base address** of struct (`record`)

```
ld $t0 -8($gp)
```

2. **add offset** of field (`field2`) to the beginning of the struct

```
addi $t0 $t0 8
```

→ now `$t0` holds address (not value) of field `(field2)`

see arrow
→ dereference

and repeat

3. **dereference** to get the base address of `myStruct` (320)

```
ld $t0 0($t0)
```

4. **add offset** of field (`data`) to the beginning of the struct

```
addi $t0 $t0 8
```

→ now `$t0` holds address (not value) of field `(data)`

see arrow
→ dereference

and repeat

5. optional: **dereference** to get value of data

```
ld $t0 0($t0)
```

…

Assignment 5: struct

- Students can extend the language C* and implement structs.
- Hint: This is another big assignment so think before you code.
  - look at the next two slides for **tips**
  - everything you need to know/do should be in the previous slides

Implementation Tips

- **Divide and conquer**

1. **Syntax**
   modify compiler to support **parsing** definition, declaration and access without generating code

   - `'struct'` keyword
   - new parsing procedure for struct definition and declaration
   - new parsing procedures for selector syntax `'->'`
   - test on examples

2. **Information** (could be part of 1.)
   modify symbol table and parser and store the necessary **information**

   - new class for `TYPE`
   - store type information
     → field names, types and offset relative to beginning of struct
   - store variables with a struct type

Implementation Tips

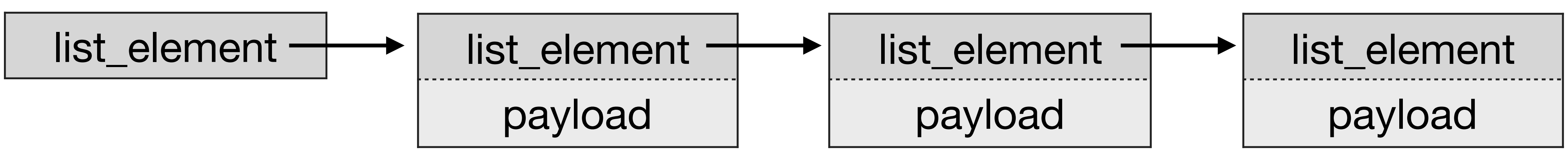
3. **Semantics**
   modify compiler to **generate code** for access computation

   - use information from symbol table to emit instructions in the right places to enable address calculation at runtime
   - test on examples

Potential for Recursion

- with structs we can define recursive data types

- recursive data types are used to build recursive data structures that are dynamic in size - lists, trees,…



Potential for Recursion

```
struct list_element_t {
    list_element_t* list_element;
    uint64_t payload;
};
```

- <u>forward declaration</u> of data type unavoidable
- It is important that a **pointer** is used in the above case
  - computing the size of this type would not be possible otherwise
    → dynamic size

Array vs Struct

- arrays and structs **share** some characteristics
  - contiguous space
  - constant time access
  - automated address calculation
  - size fixed at compile time (as we discussed)
  - the concept of local and global structs

Array vs Struct

- We will now look again at the **key differences** between arrays and structs
- **Data**
  - struct: data elements of **different type** allowed
  - array: data elements have to be of **same type**
- **Access**
  - struct: accessed using **field name/identifier**
  - array: accessed using **index values** (integers) using pointer arithmetic

Array vs Struct

- **Address computation**
  - struct: offset to beginning of struct always known at **compile time**
  - array: necessary information might not be available before **runtime**

`myArray[i]`

information only
available at runtime

`record -> field1`

information always
available at compile time
→ offset is always known

Array Access

- The key disadvantage of array access is the use of **pointer arithmetic**
  - illegal access outside the is easily possible
  - accidentally use wrong index
  - **compiler does not understand** the intentions of the programmer and therefore cannot prevent the above problems

Struct Access

- The key advantage of struct access is the use of **field names/ identifiers**
  - only fields of that array can be accessed
    (no illegal access possible when using correct syntax)
  - using the wrong name is not possible
  - **compiler does understand** the intentions of the programmer and therefore has the capability of giving feedback on correctness

Structs Conclusion

- By implementing/supporting structs we **teach** the compiler.
- We make the compiler become aware of the programmers **intention** and give it the capability to give **feedback** on correctness.
- With structs we now have an **unbound type space**.

Memory Allocation

Procedures,
Garbage Collection

Procedures

`./selfie -c`

Procedure Declaration

```
uint64_t doMagic(uint64_t power, uint64_t wand);
```

formal parameters

Procedure Definition

```
uint64_t doMagic(uint64_t power, uint64_t wand) {
    uint64_t trick;
```

no code generation - just symbol table

code generation

```
    trick = power * wand;
    return trick;
}
```

Procedure Call

```
doMagic(7, 1)
```

actual parameters

Formal/Actual Parameters & Local Variables

- Local variables represent types and names. They are a special case of formal parameters.
- Formal parameter represent types and names. They are more general and more expressive than local variables, because they enable us to access and modify information outside the procedure.
- Actual parameters represent the actual type of the parameters when the procedure is called.

Procedure Declaration

```
uint64_t doMagic(uint64_t power, uint64_t wand);
```

- **Symbol table entry**
  - introduce a name/identifier for procedure (**global**)
  - introduce name/identifier for formal parameters (**local**)
- **No code generation**

Procedure Definition

```
uint64_t doMagic(uint64_t power, uint64_t wand) {
    uint64_t trick;
    … body …
}
```

- **Symbol table entry**
  - introduce a name/identifier for procedure (**global**)
  - introduce formal parameters (**local**)
  - introduce local variables (**local**)
- **Code generation**
  - generate code to **prepare the machine** for the execution of the procedure (prologue)
  - for **body** of the procedure (its implementation)

Procedure Call

```
doMagic(7, 1);
```

- The calling procedure is also known as **caller**, whereas the called procedure we consider to be the **callee**.
- **Code generation**
    - generate code to **prepare** the machine for jump (parameters and saving certain registers)
    - for actual **jump** to the code implementing the procedure

Parameters

- The caller hands the parameters over to the procedure
- There are two strategies how this can be done:
  - using special **registers**
    → fewer parameters possible
  - placing them somewhere the **callee can access** them
    → more general
    → more parameters possible

    This is a convention that must be agreed on by the caller and the callee.
- In selfie the later one is implemented - pushing parameters on the **stack**

Parsing Parameters

- **Matching** between actual parameters and formal parameters based on the order in which they appear

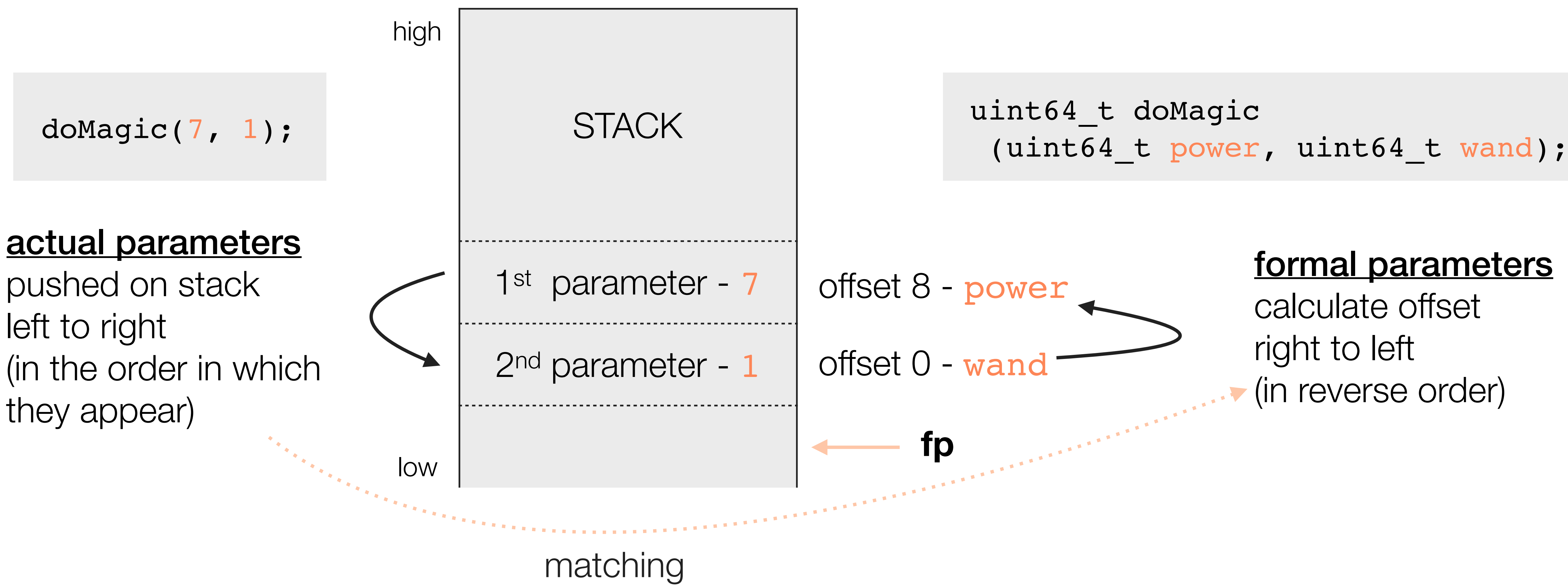
- Parsing a **declaration**
  - formal parameters parsed from left to right
  - offset calculated from right to left

matching

- Parsing a **call**
  - actual parameters parsed from left to right
  - pushed on stack from left to right

Matching Parameters

./selfie -c



- Why not parse declaration **the other way around**?
  - tedious to match for **every** call (pushing parameters)
  - Trade-off: either calls are easy **OR** declaration is easy

Caller vs Callee Save and Restore

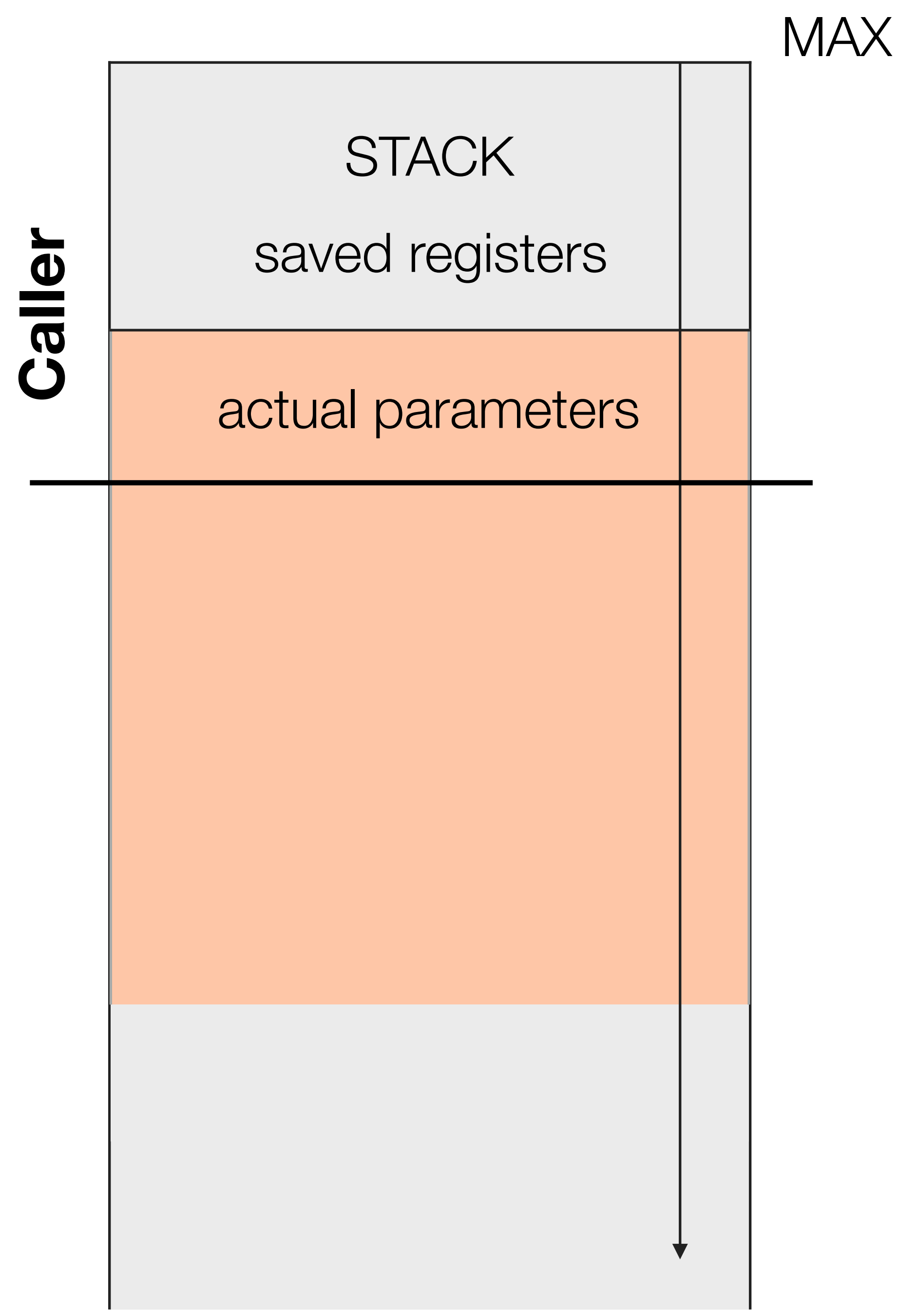
`./selfie -c`

- Saving and restoring registers in case of a procedure call can be done by either the caller or the callee. There are <u>conventions</u> about how to do this. More important are the **assumptions** that can be made in both cases.

  - **<u>Caller-saved registers</u>** are saved by the caller right before a procedure call and restored immediately after. It is only necessary to save the registers that contain values that are used by the caller after the call.
    Advantage - **callee** can assume that it can use all registers.

  - **<u>Callee-saved registers</u>** are saved by the callee right before they are modified and get restored before returning to the caller. Those registers can be used to pass information to the callee.
    Advantage - **caller** can assume that none of the register values have changed.

Procedure Frame

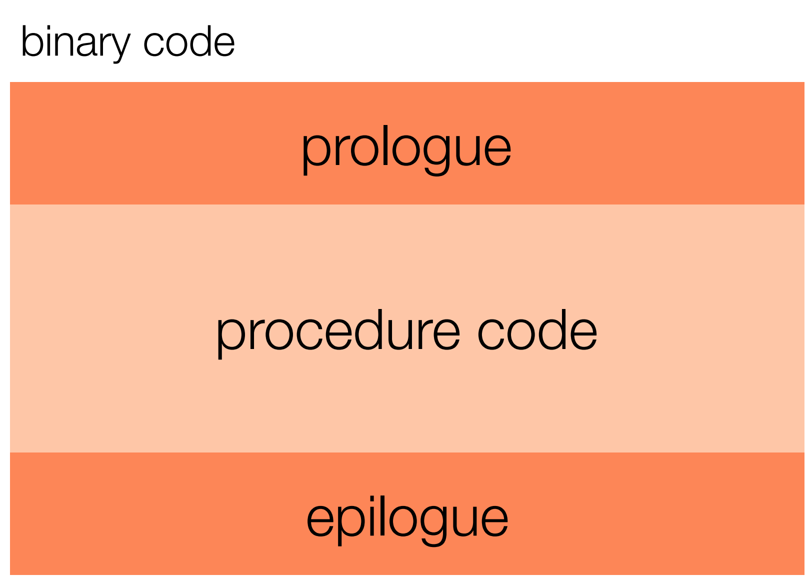


- part of the stack accessible to the procedure

Prologue & Epilogue

`./selfie -c`

- Every procedure is **encapsulated** by
  - a **prologue** that prepares the stack and registers for the execution of the procedure
  - an **epilogue** that prepares the stack and registers to return after the execution of the procedure.
- They are compiled as part of the procedure definition (`compile_procedure()`) and hence they are part of the callee.



Prologue

- The prologue of a procedure **prepares** the stack and registers for the **execution** of the procedure.
    - setting the **frame** for the callee
    - **memory allocation** for local variables
    - **saving** callers frame pointer and return address.
- The callers frame pointer needs to be saved because it will be overwritten when the callees frame is set.
- The `$ra` register contains the address of the next instruction to be executed after the procedure. This register will be overwritten if another procedure is called within the current procedure. Therefore it needs to be saved too.

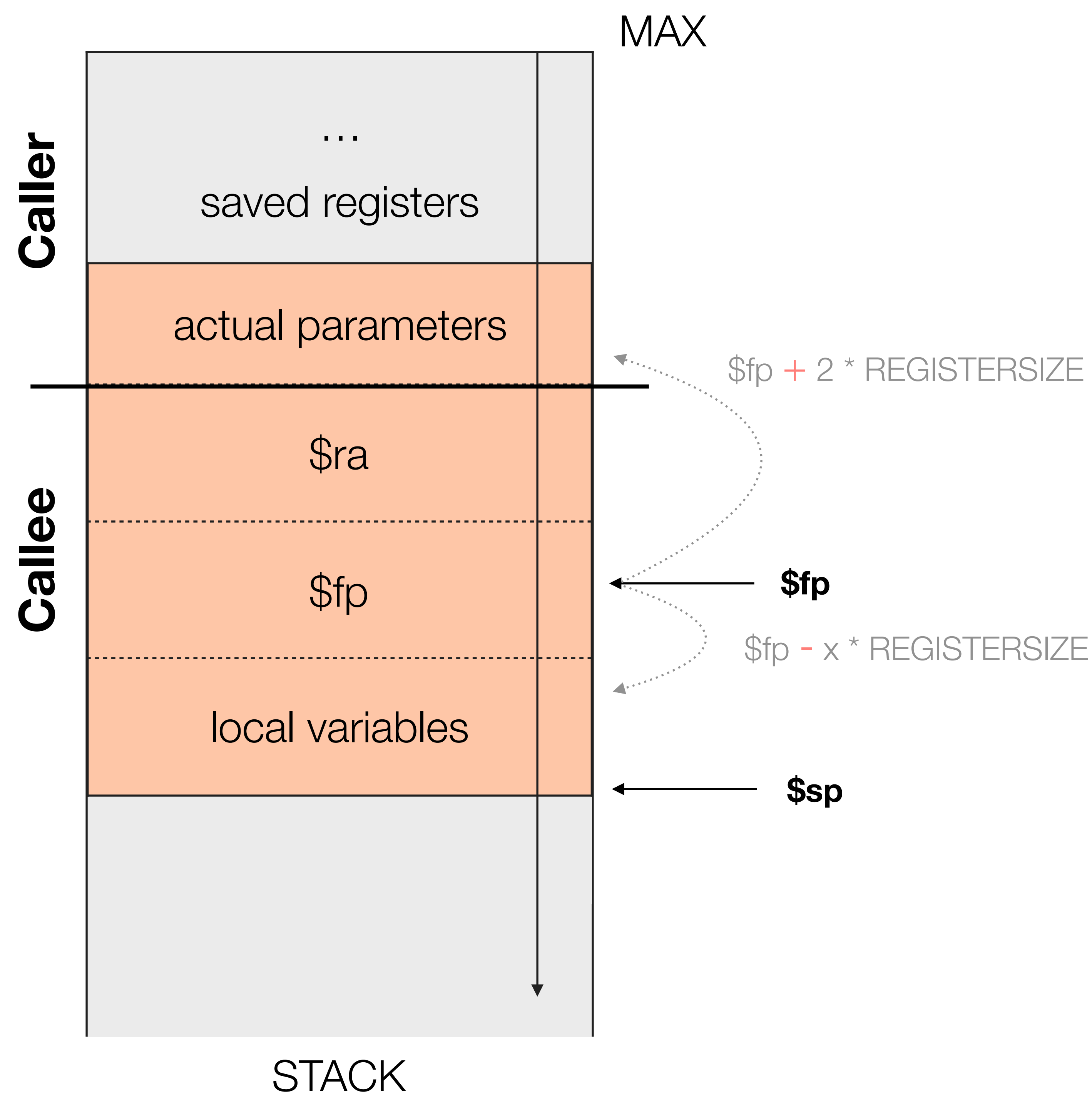
Prologue

./selfie -c

```
void help_procedure_prologue(uint64_t bytesOfLocalVariables) {

   // allocate memory for return address
  emitADDI(REG_SP, REG_SP, -REGISTERSIZE);

  // save return address
  emitSD(REG_SP, 0, REG_RA);

  // allocate memory for caller's frame pointer
  emitADDI(REG_SP, REG_SP, -REGISTERSIZE);

  // save caller's frame pointer
  emitSD(REG_SP, 0, REG_FP);

  // set callee's frame pointer
  emitADDI(REG_FP, REG_SP, 0);

  // allocate memory for callee's local variables
  if (number_of_local_variable_bytes > 0) {
    …
  }
}
```

Procedure Frame after Prologue

- Part of the stack allocated/ accessible to the procedure
- Access with
  - frame pointer (parameters, local variables)
  - stack pointer (top of stack)



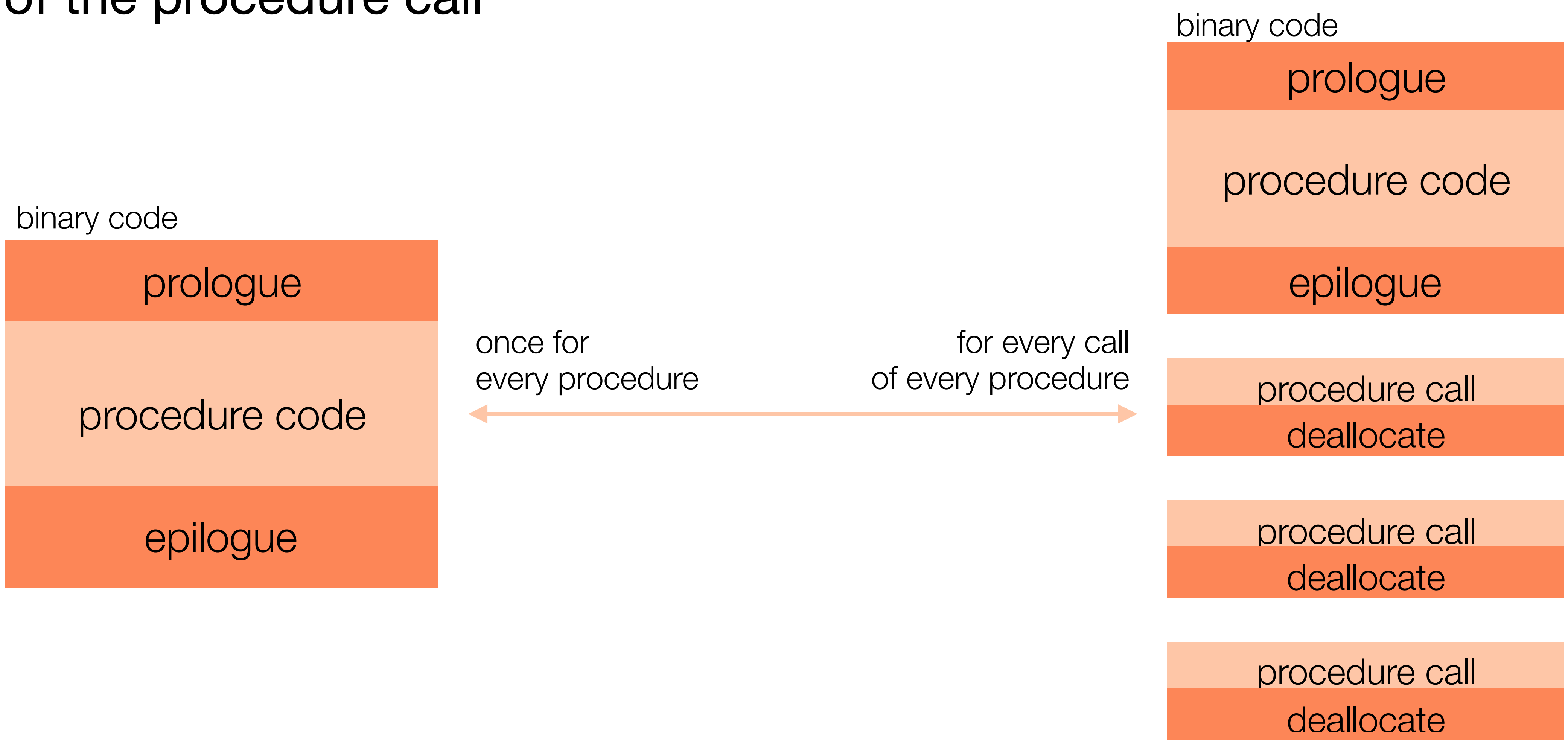
Epilogue

- The epilogue of a procedure **prepares** the stack and registers for the return of the procedure.
  - setting the **frame** (back) for the caller
  - **deallocate memory** of local variables
  - **restore** callers frame pointer and return address.
  - **deallocate memory** for parameters
- To deallocate memory for local variables the stack pointer is reset to the beginning of the frame.
- Notice: The memory for parameters is deallocated by the callee even though it was allocated by the caller. Why?

Epilogue

- Deallocating memory for parameters **in the epilogue (callee):** code for doing so is emitted just once as part of the procedure definition

- Deallocating memory for parameters **by the caller:** code for doing so is emitted each time the procedure is called as part of the procedure call



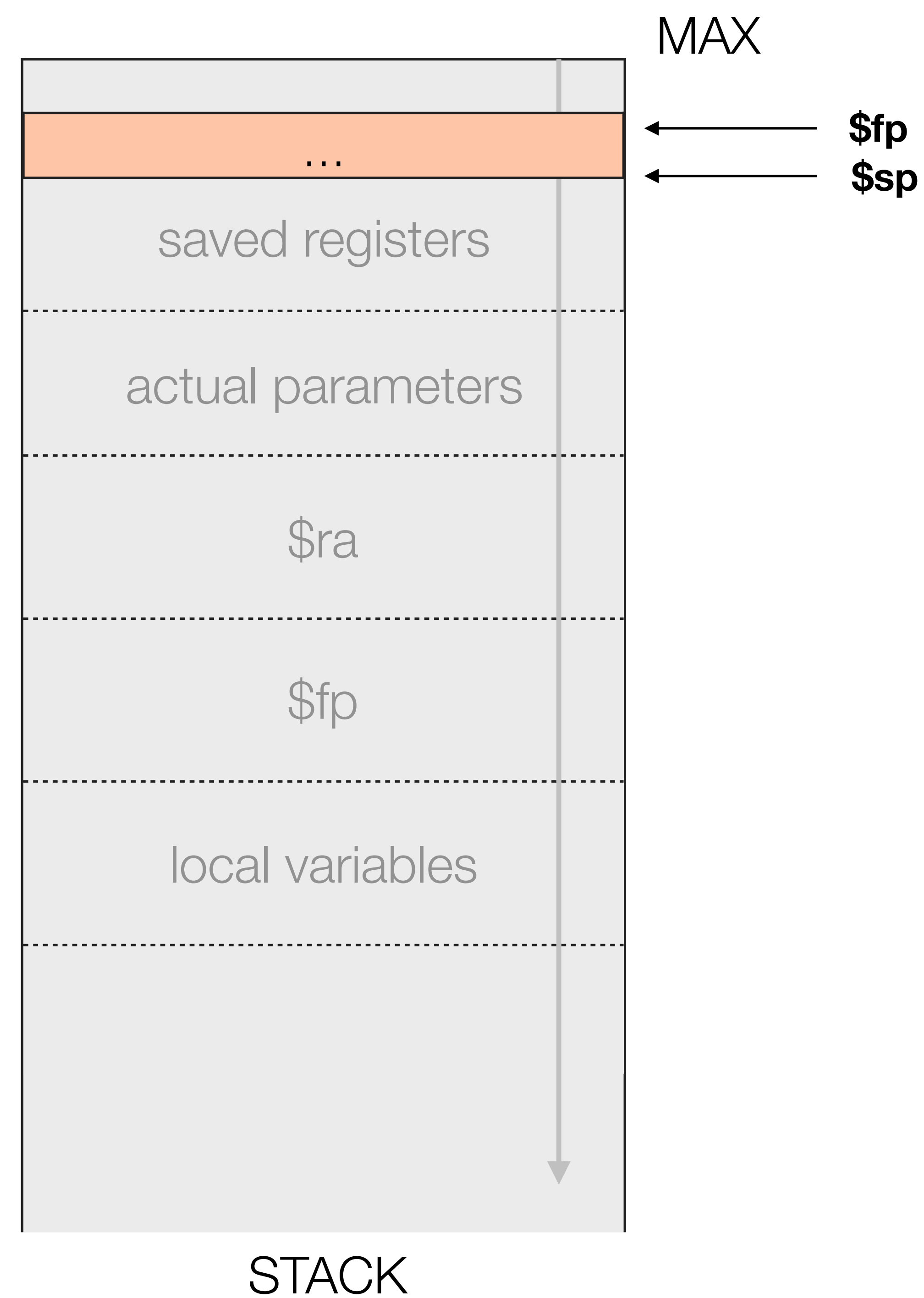
Epilogue

./selfie -c

```
void help_procedure_epilogue(uint64_t parameters) {
  // deallocate memory for callee's frame pointer and local
     variables
  emitADDI(REG_SP, REG_FP, 0);

  // restore caller's frame pointer
  emitLD(REG_FP, REG_SP, 0);

  // deallocate memory for caller's frame pointer
  emitADDI(REG_SP, REG_SP, REGISTERSIZE);

  // restore return address
  emitLD(REG_RA, REG_SP, 0);

  // deallocate memory for return address and parameters
  emitADDI(REG_SP, REG_SP, REGISTERSIZE + parameters *
  REGISTERSIZE);

  // return
  emitJALR(REG_ZR, REG_RA, 0);
}
```

Procedure Frame after Epilogue

- part of the stack allocated/ accessible to the procedure
- callers frame restored and memory deallocated by resetting `$sp`



Understanding Recursion

- **Now we know how…**
  - the prologue prepares the stack for execution of a procedure,
  - the epilogue prepares the stack for return of a procedure,
  - the procedure frame is laid out (and what it is)…
- **…we can look (from a new perspective) at recursion** and be able to really understand how recursive procedures work.
  - we show the stack during execution of a recursive procedure
- First we look at the iterative implementation of the `add(…)` procedure and then we will show the recursive implementation of this procedure.

Iteration

```
uint64_t add(uint64_t x, uint64_t y) {
    while (y > 0) {
        x = x + 1;
        y = y - 1;
    }
    return x;
}
```

- This procedure defines <u>iterative</u> addition of `x + y`
- The same procedure can be defined **recursively.**

Recursion

```
uint64_t add(uint64_t x, uint64_t y) {
    uint64_t sum;
    if (y > 0)
        sum = add(x, y - 1) + 1;
    else
        sum = x;
    return sum;
}
```

recursive call

base case

- The definition of a recursive procedure contains a call to itself.
- The recursive call usually simplifies the problem and leads towards a base case which is used to break out of the recursion.
- Taking a look at the **call stack** helps to really understand what happens during a recursive procedure call.

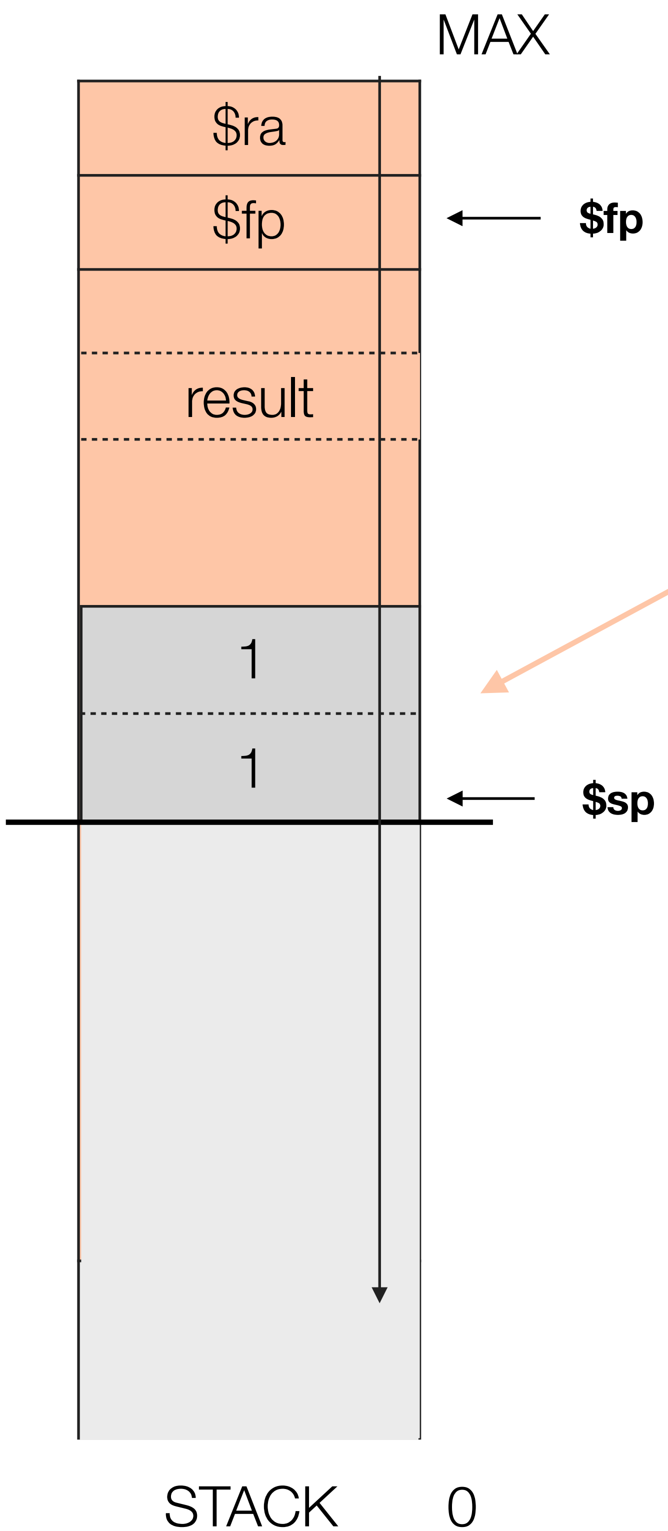
Recursion Call Recursive Procedure

```
result = add(1,1);
```

to solve	`add(1,1);`	simplify with recursive call
solve	`add(1,0) + 1;`	base case
solving the base case is simple	`add(1,0)`	

- We will look at the call stack during this call.

Recursive Procedure First Call



main():

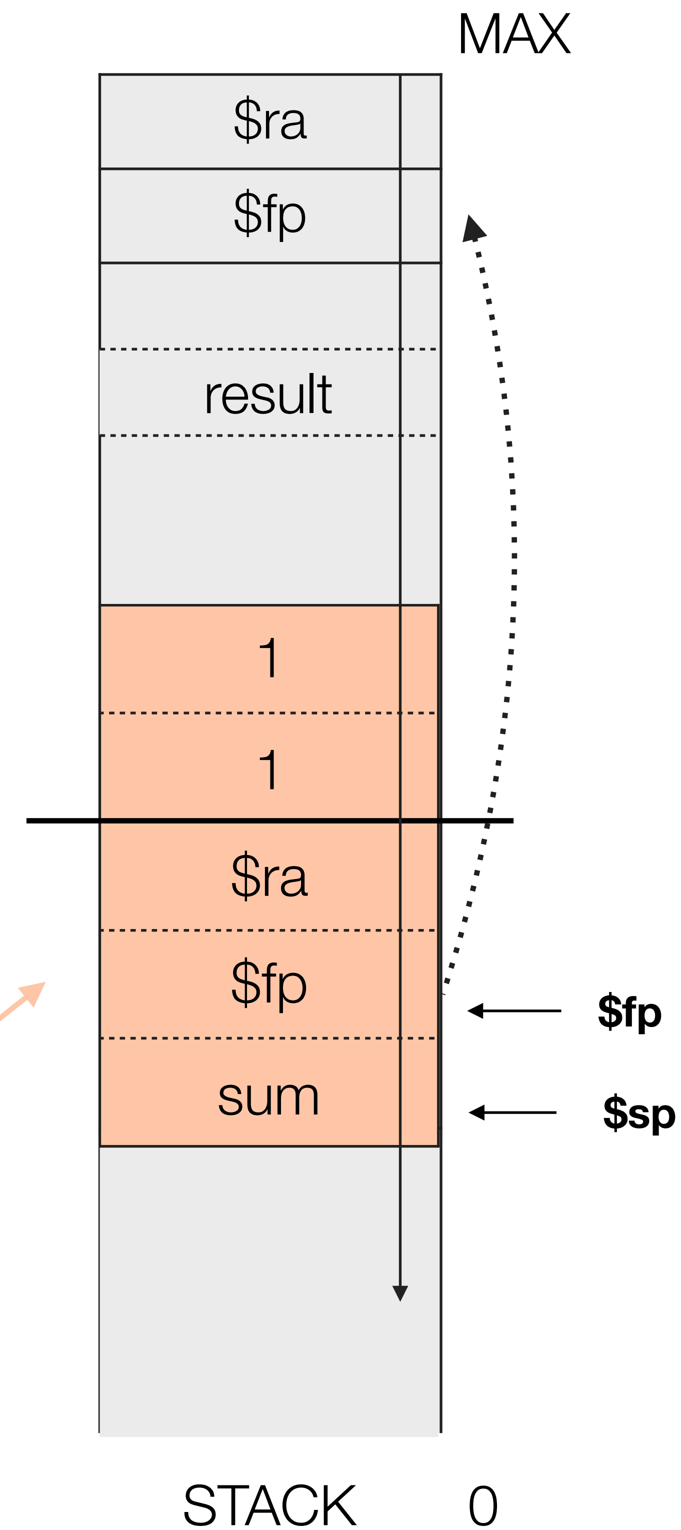
```
result = add(1,1);
```

before call:

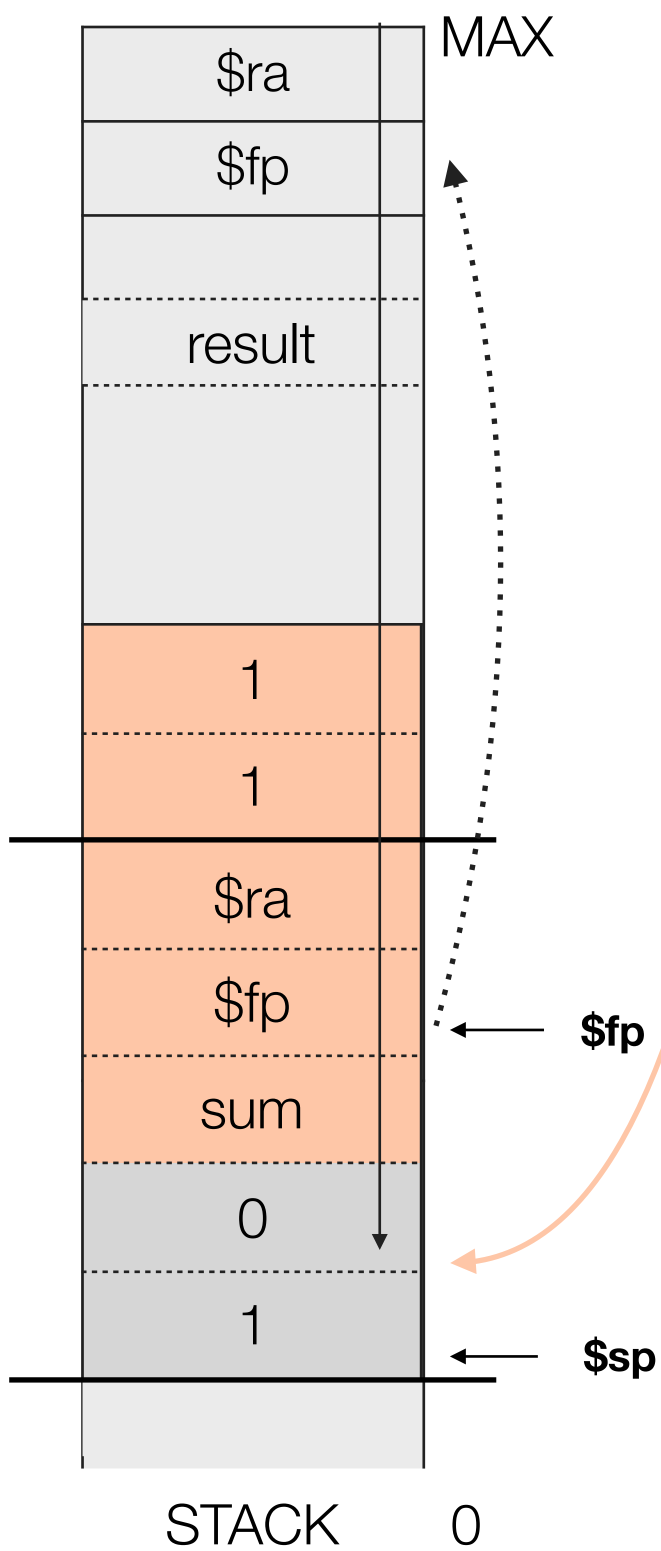
- arguments for add`(1,1)` pushed on stack
- **jump (`JAL`)** to beginning of procedure
- **link(`$ra`)** to next instruction (code that does the assignment into `result`)

after jump:

- save `$ra`, `$fp` and allocate space for local variable `sum`
- procedure frame changed



Recursive Procedure Recursive Call



add(1,1):

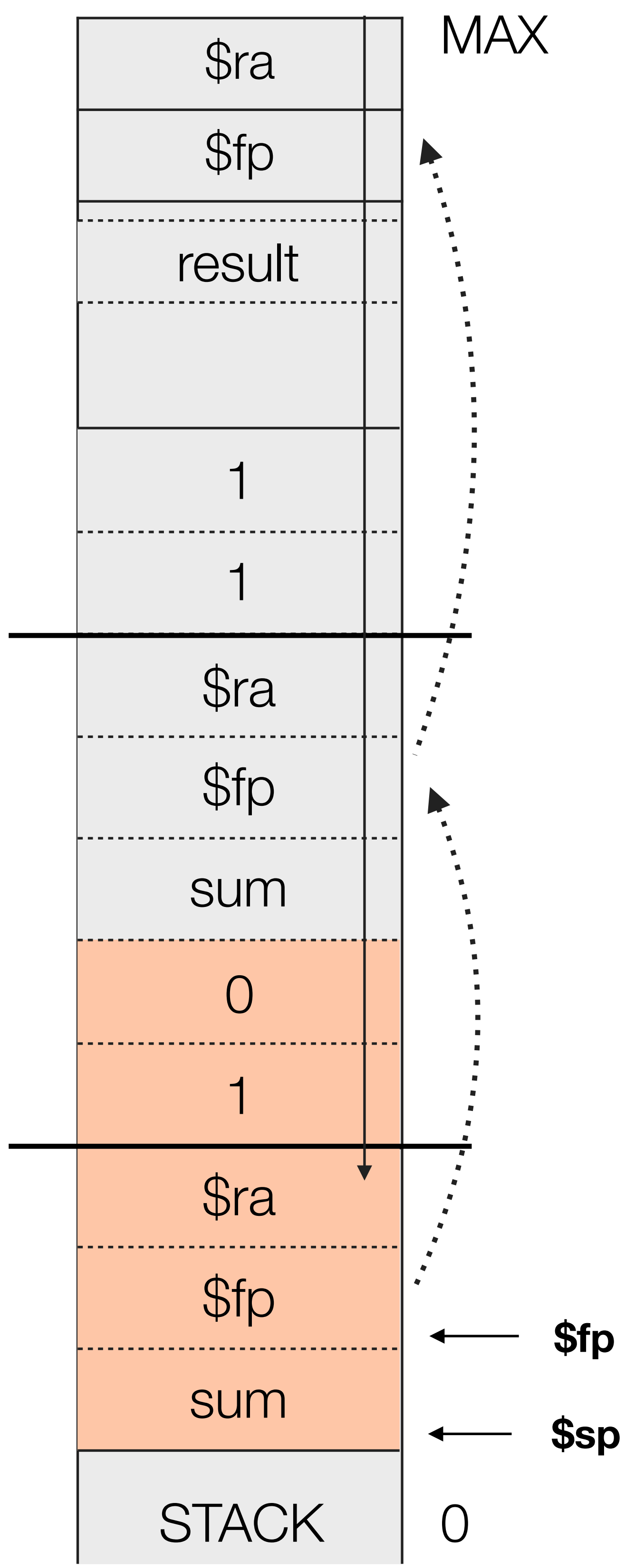
```
sum = add(0,1) + 1;
```

before call:

- arguments for `add(0,1)` pushed on stack
- **jump (`JAL`)** to beginning of procedure
- **link(`$ra`)** to next instruction (code that adds `1`)

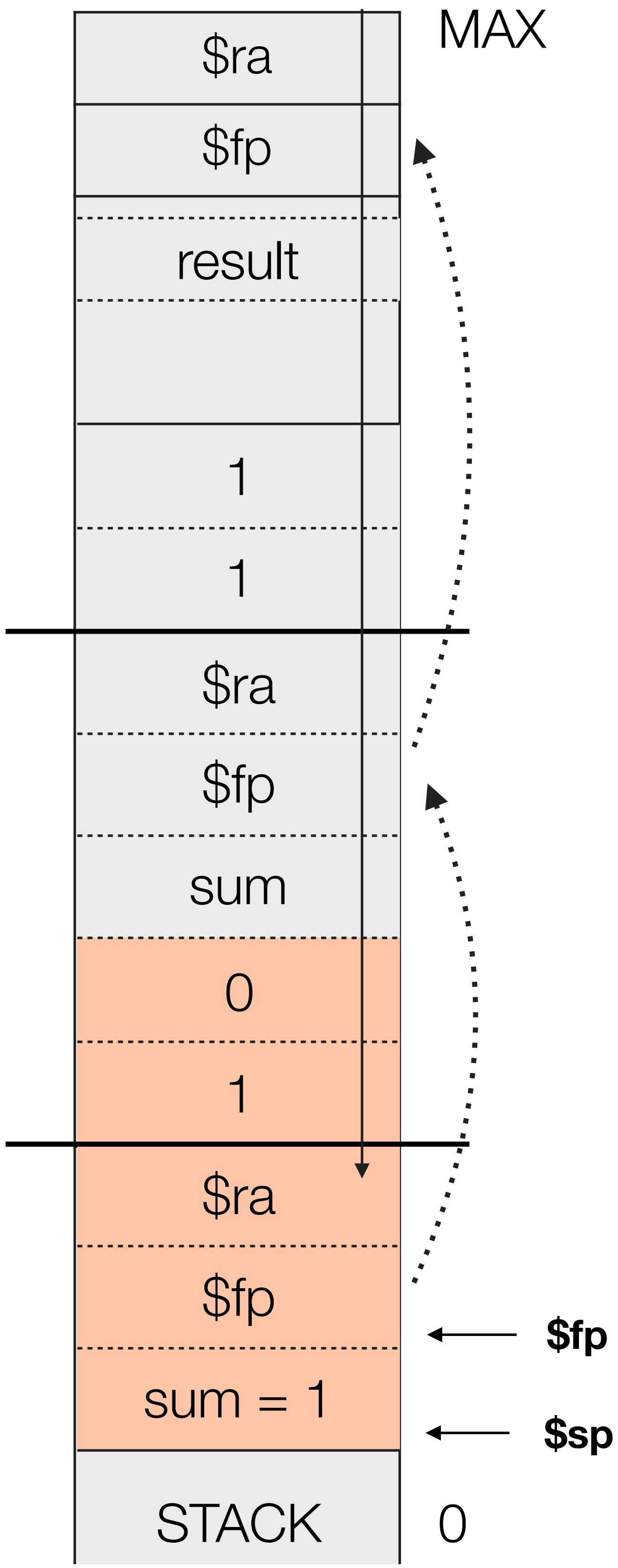
after jump:

- save `$ra`, `$fp` and allocate space for local variable `sum`
- procedure frame changed



different frame = different local variable

Recursive Procedure Base Case and Return



add(0,1):

```
sum = x;
…
return sum;
```

before return:

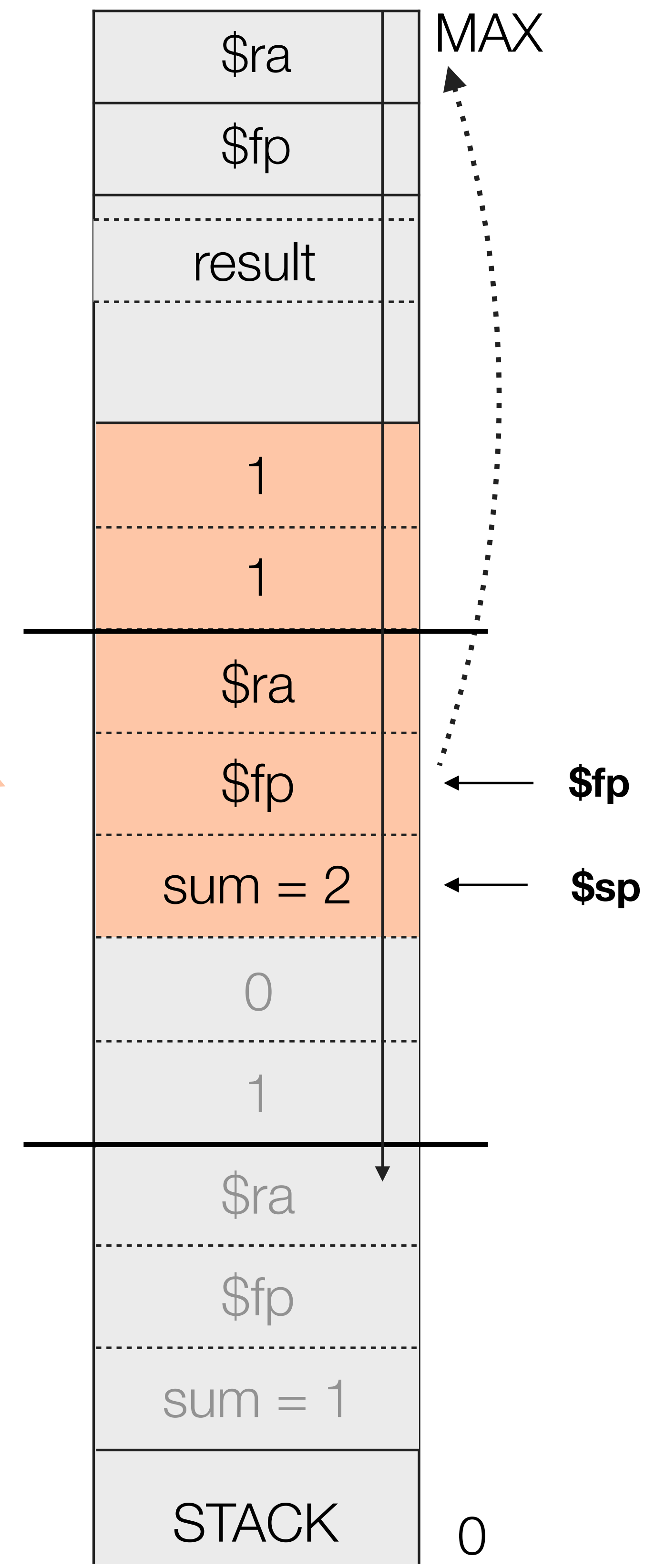
- store return value `sum` in `$a0` (register for return values)
- **deallocate** memory
- **restore `$fp`**
- **restore `$ra`**

after return jump:

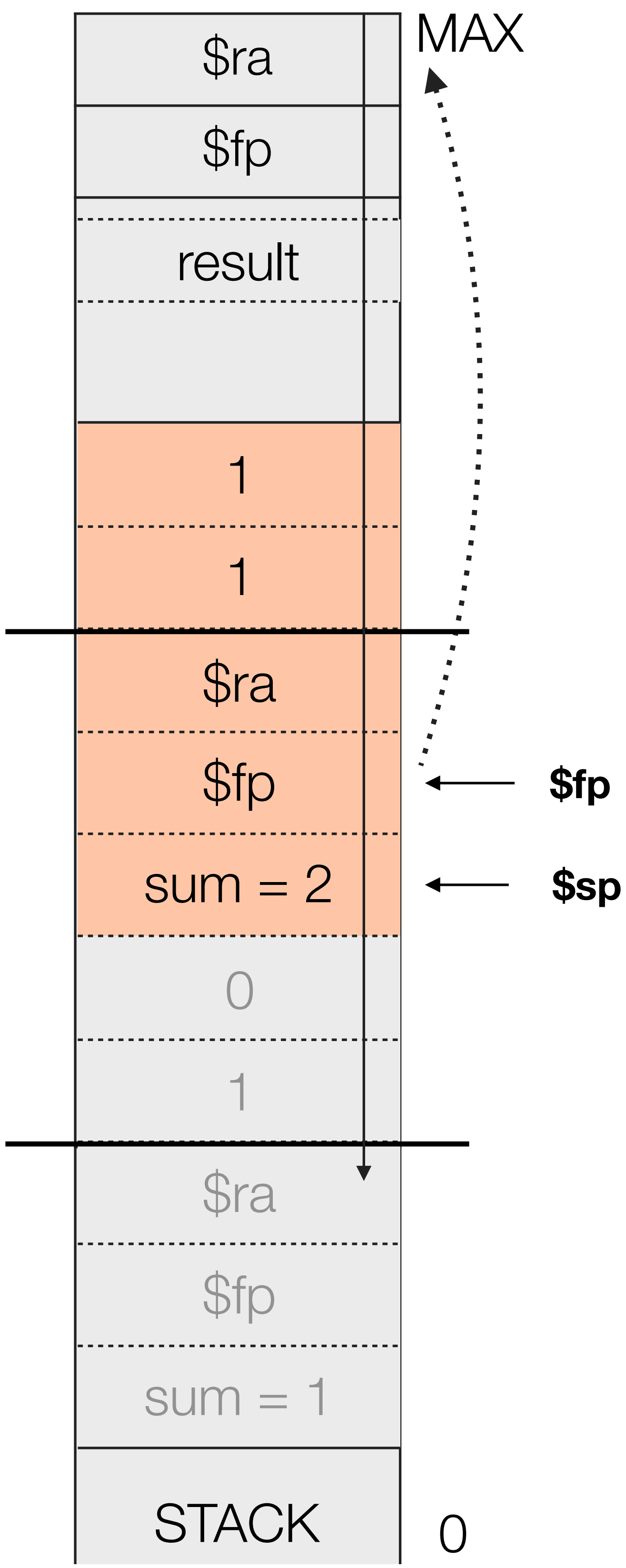
- procedure frame changed
- code that adds `1` to return value (`0 + 1`)

add(1,1):

```
sum = add(0,1) + 1;
```



Recursive Procedure Return



add(1,1):

```
sum = add(0,1) + 1;
...
return sum;
```

before return:

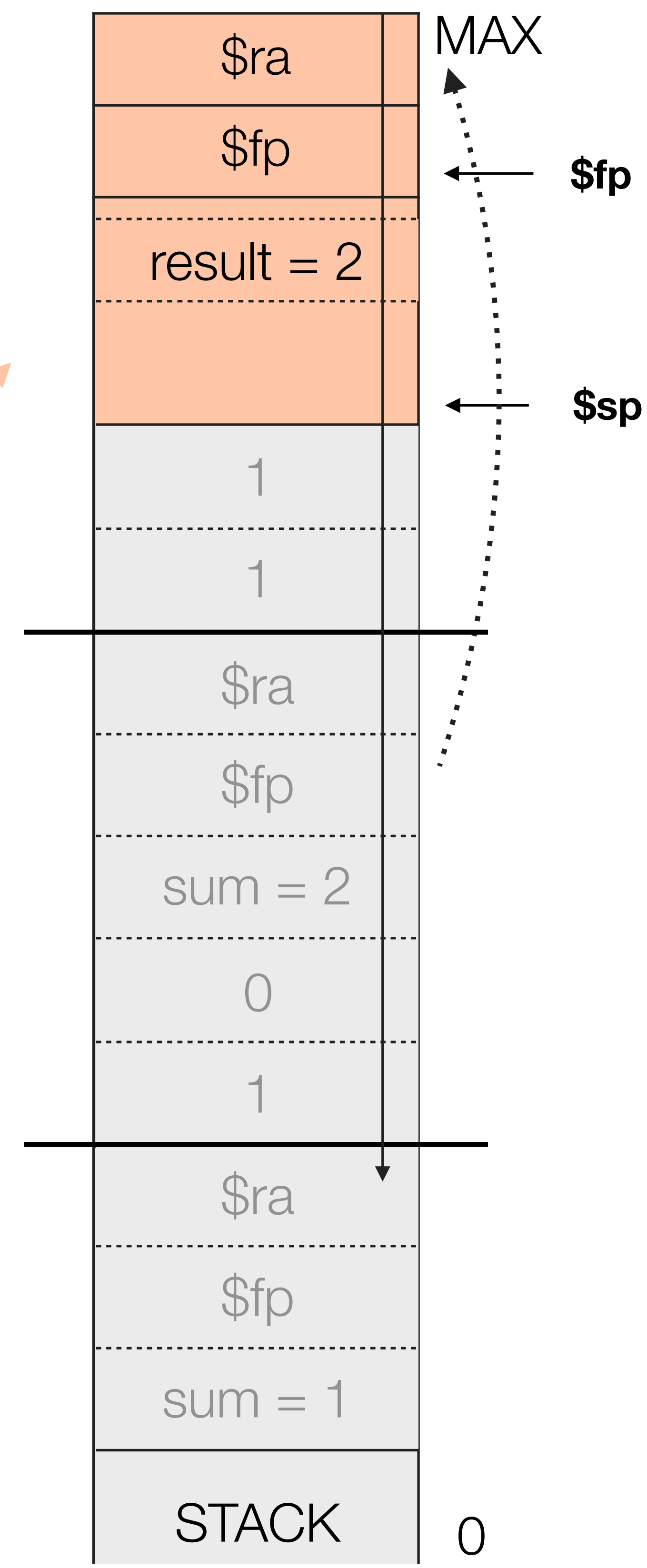
- store return value `sum` in `$a0` (register for return values)
- **deallocate** memory
- **restore `$fp`**
- **restore `$ra`**

after return jump:

- procedure frame changed
- code that assigns return value to `result`

main():

```
result = add(1,1);
```



“Why talk about procedures in the ‘Memory Allocation’ Section?”

“Procedures are the most efficient way to allocate memory”

Memory Allocation Procedures

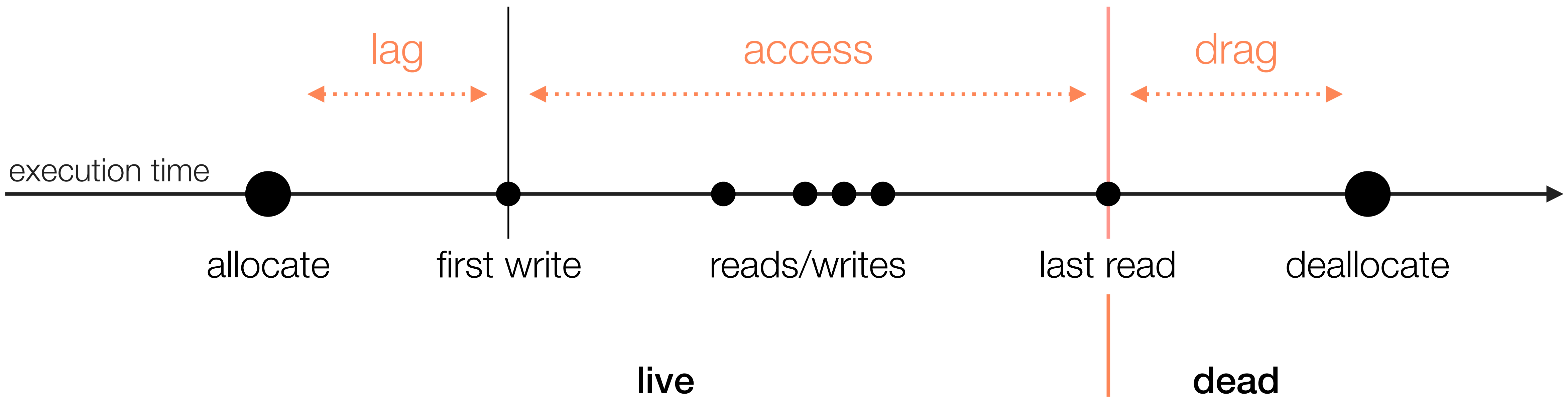
- Memory for procedures is allocated on the stack (stack allocator).
- **Efficiency**
  - using procedures there is no need for explicitly mallocing or deallocating memory
    → prologue and epilogue
  - memory management
    → deciding if memory is live or dead

Liveness

- An address can either be **live** or **dead**
  - a live address will be read at some point in the future, whereas a dead address will not be read anymore
  - a live address can not be reused

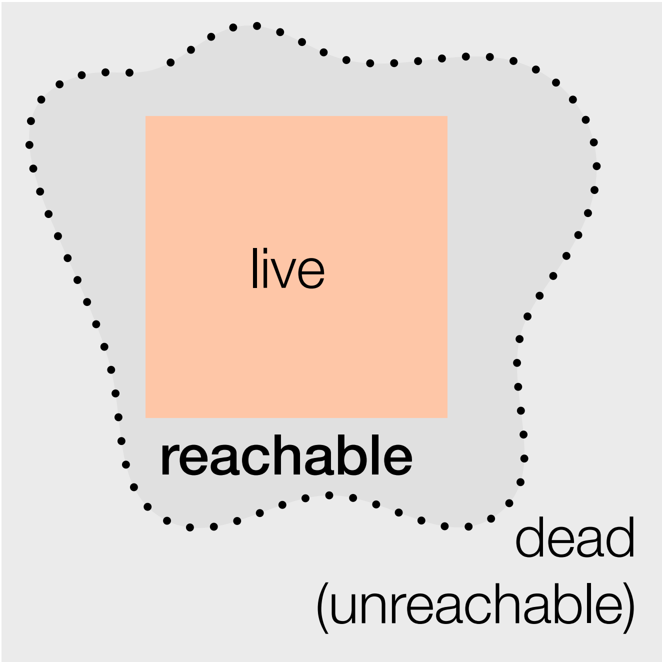
Liveness

- **Lag** - time between allocation and first write access
- Period of **accesses** - reads and writes
- **Drag** - time between last read and deallocation
- Long lag or drag periods → inefficient memory consumption



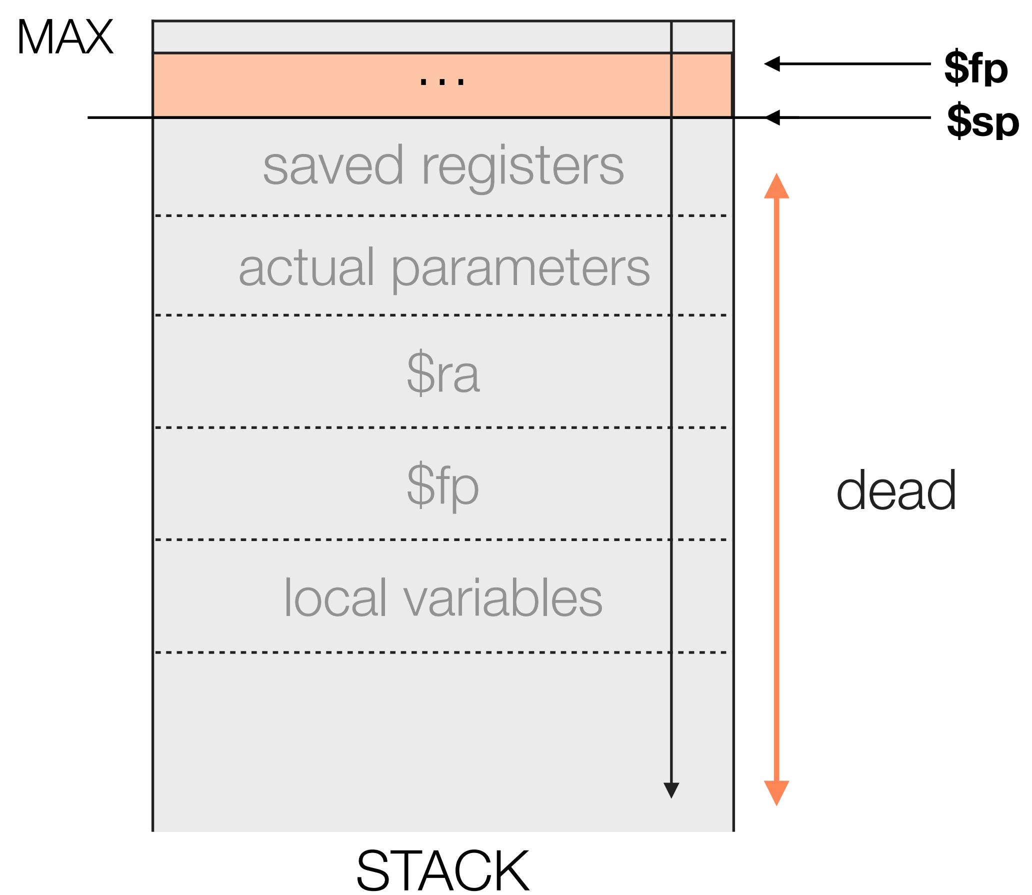
Computing Liveness

- It is **undecidable** whether an address is live or dead
  - neither the set of dead addresses nor the set of live addresses is computable (lines are blurred)
- We can however compute an approximation of liveness - **reachability**
  - compute superset of live addresses
  - guarantee that everything outside is dead
  - garbage collector uses this principle



Liveness Procedures

- It is **undecidable** whether an address is live or dead but for procedures we can at least make a clear statement about dead memory
  - jumping out of a procedure leaves the addresses below the stack pointer and above the heap dead



Control Flow

Control Flow Statements and Fixup

Control Flow

`./selfie -c`

*<u>Control Flow</u> is the **order** in which instructions/statements are executed. It can be described by a **sequence of program counter values**.*

- Selfie features 3 **control flow statements**:
  - <u>`while`</u>
  - <u>`if-then-else`</u>
  - <u>`return`</u>

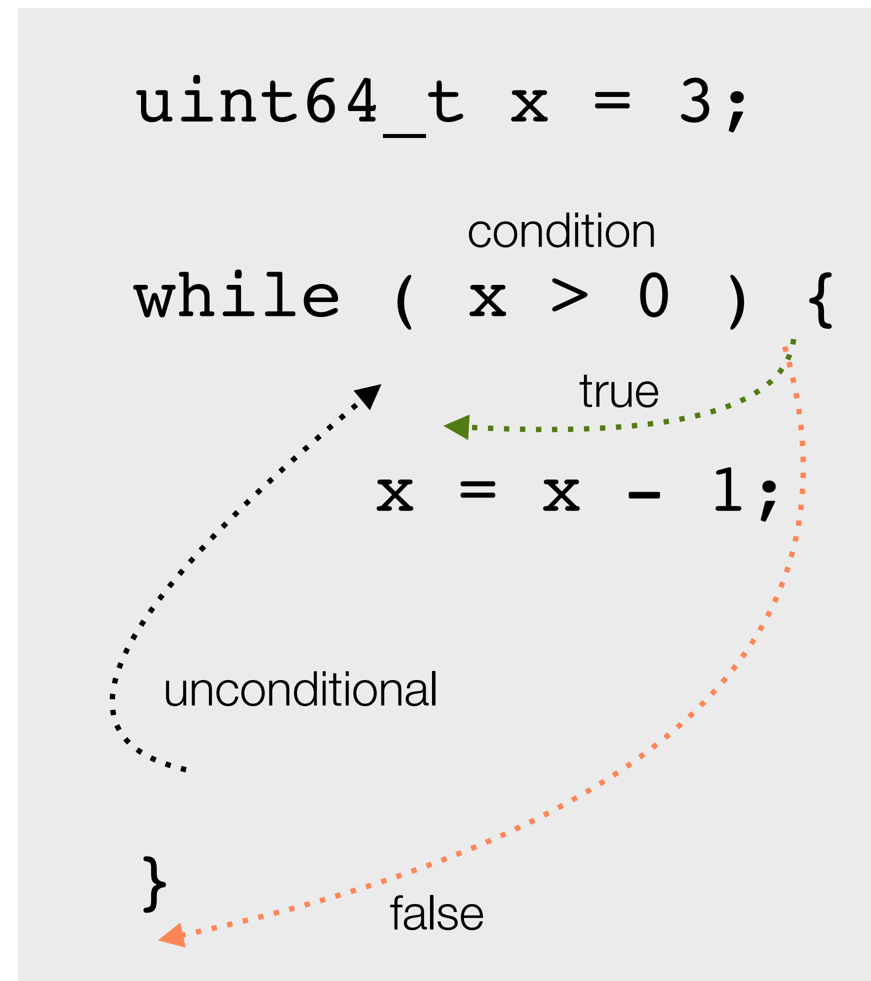
Expressivity of Control Flow Statements

- **while and for**
  - same expressivity - one can be rewritten into the other
- **while/for and if-then-else**
  - while/for more expressive than if-then-else
  - if-then-else can be rewritten to while/for but not the other way around
- **rewriting**
  - same algorithmic complexity - difference in size only a constant factor

While Loop Semantics

- If the condition evaluates to true the body of the while loop is executed. Otherwise the execution continues after the while loop - <u>**conditional**</u> **branch**

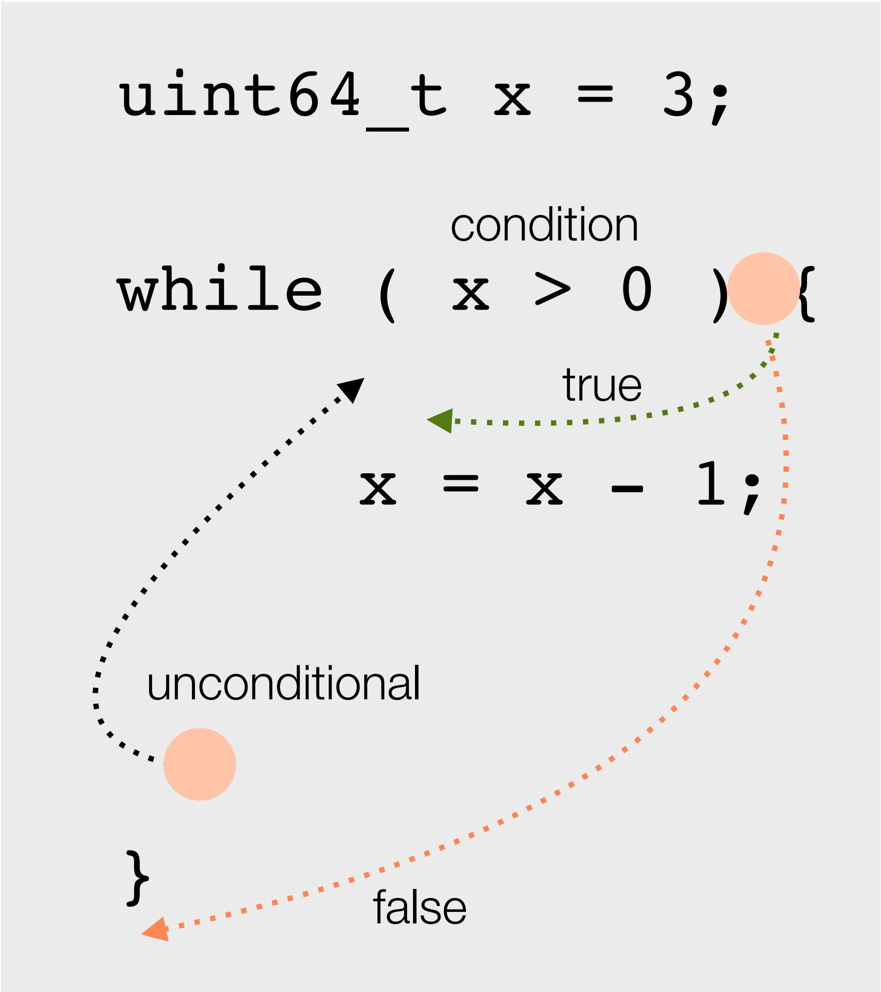
- After each execution of the body the expression is evaluated again - **unconditional branch**



While Loop Runtime vs Compile Time

- At **compile time** this code is parsed once from top to bottom and the instructions are emitted in sequence. Control flow instructions are written into the binary at the right points.

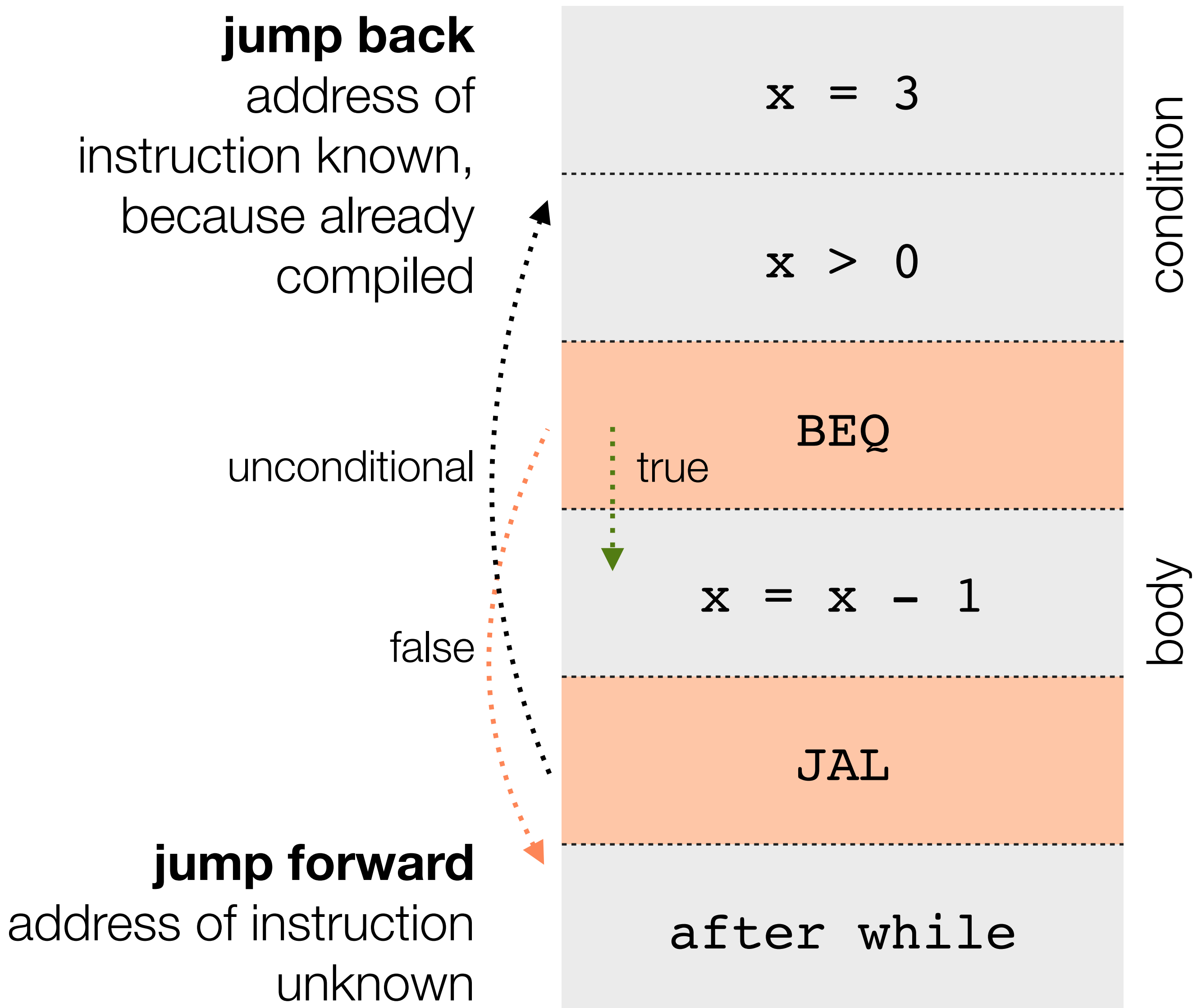
- At **runtime** those control flow instructions provide the mechanism that allows execution to follow different paths.



While Loop Runtime vs Compile Time

- Code is parsed from top to bottom and instructions are emitted in sequence.

- Control flow instructions are inserted to enable branching at runtime.



Parsing While-Loop

./selfie -c

1. Condition executed before each iteration:

   TODO: **remember** address of first instruction (`jump_back_to_while`)

2. Decide where to continue execution:

   TODO: emit **BEQ** instruction to unknown address and remember address of this “wrong” instruction (`jump_forward_to_end`)

3. Beginning of body:

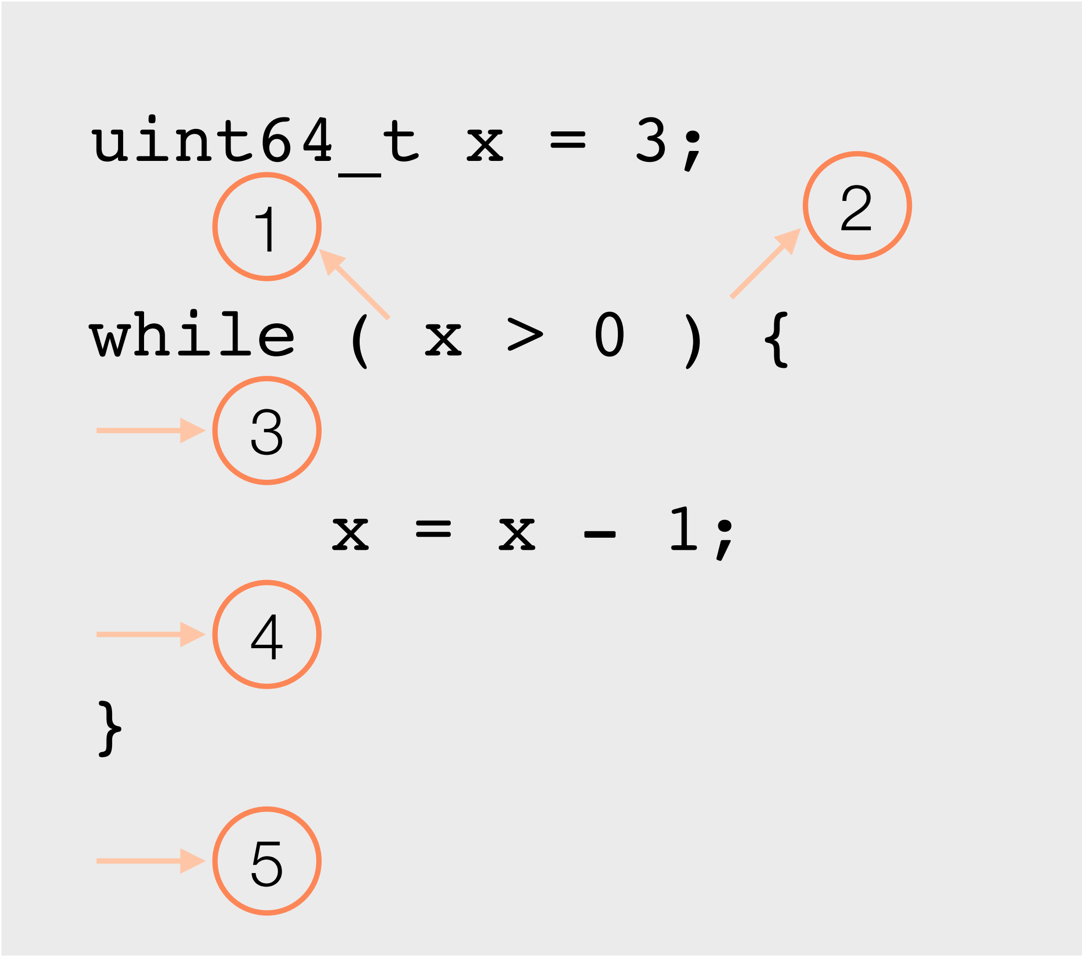
   TODO: parse **body**

4. End of body and back to 1:

   TODO: emit **JAL** instruction to remembered address (`jump_back_to_while`)

5. End of while (target address of BEQ known):

   TODO: fixup **BEQ** instruction



Parsing While-Loop

./selfie -c

1. Condition executed before each iteration:

   TODO: **remember** address of first instruction (`jump_back_to_while`)

2. Decide where to continue execution:

   TODO: emit **BEQ** instruction to unknown address and remember address of this "wrong" instruction (`jump_forward_to_end`)

3. Beginning of body:

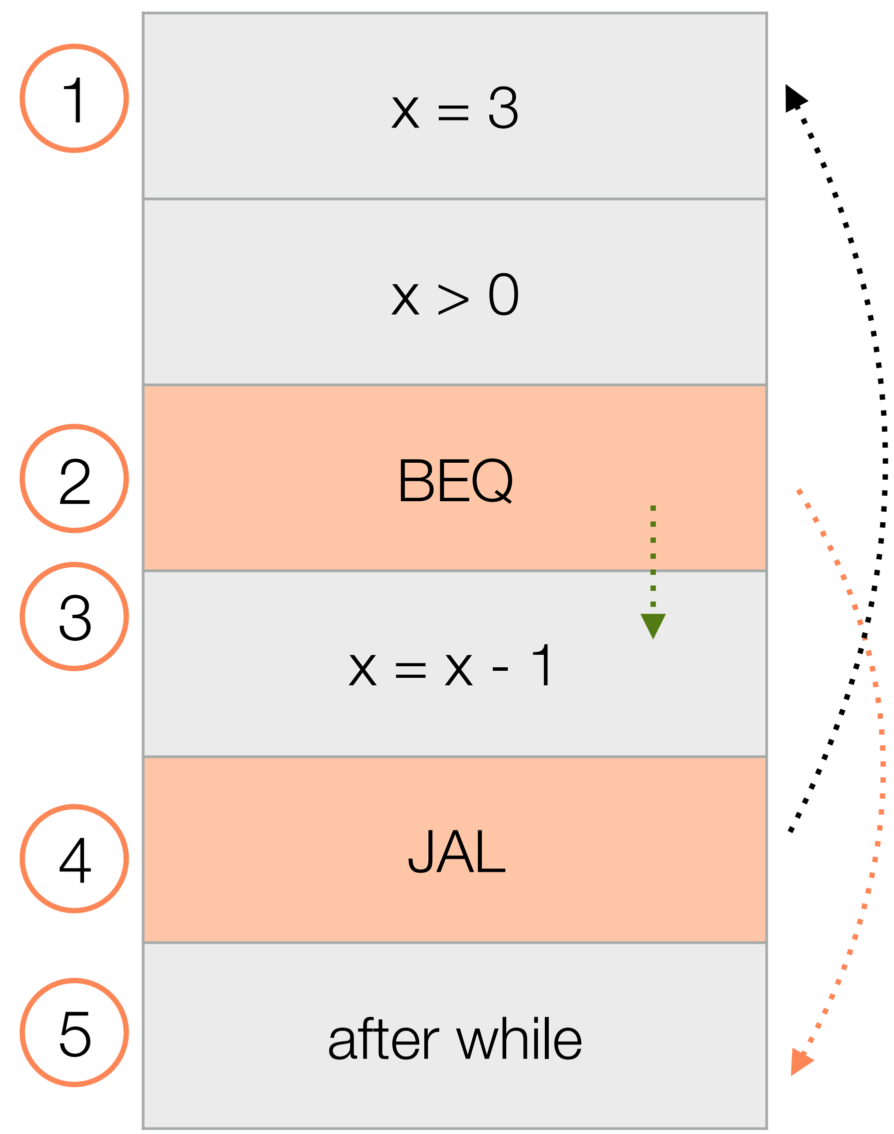
   TODO: parse **body**

4. End of body and back to 1:

   TODO: emit **JAL** instruction to remembered address (`jump_back_to_while`)

5. End of while (target address of BEQ known):

   TODO: fixup **BEQ** instruction



Jump and Branch

- Jump
  - already know absolute address at runtime (**strong assumption**)
  - 20 bit for address
- Branch
  - addressing relative to program counter
  - relocating code becomes simple
  - limited branch distance (12 bit for address)

Fixup

*A fixup is the process of changing (correcting) the destination **address** of a control flow instruction after it has been emitted.*

- Fixups solve a problem that is present in all single-pass compilers for languages featuring procedures or control flow statements.
- Necessary when **destination address** of instruction is **unknown** when the instruction has to be emitted.
    - instruction is emitted with wrong address
    - address needs to be fixed up later when the address is known

Fixup

- Fix up a single instruction
  - control flow statements (while, for, if-then-else)
- Fix up multiple instructions
  - control flow statements (if-elseif)
  - return statement
  - procedure calls

Fixup Single Instruction

- JAL or BEQ instructions emitted when compiling while, for or if-then-else

- **Fixed number** of instructions to fixup

- **How to fix up an instruction**

  1. emit instruction with wrong address and remember address of this wrong instruction (local variable to enable nesting)

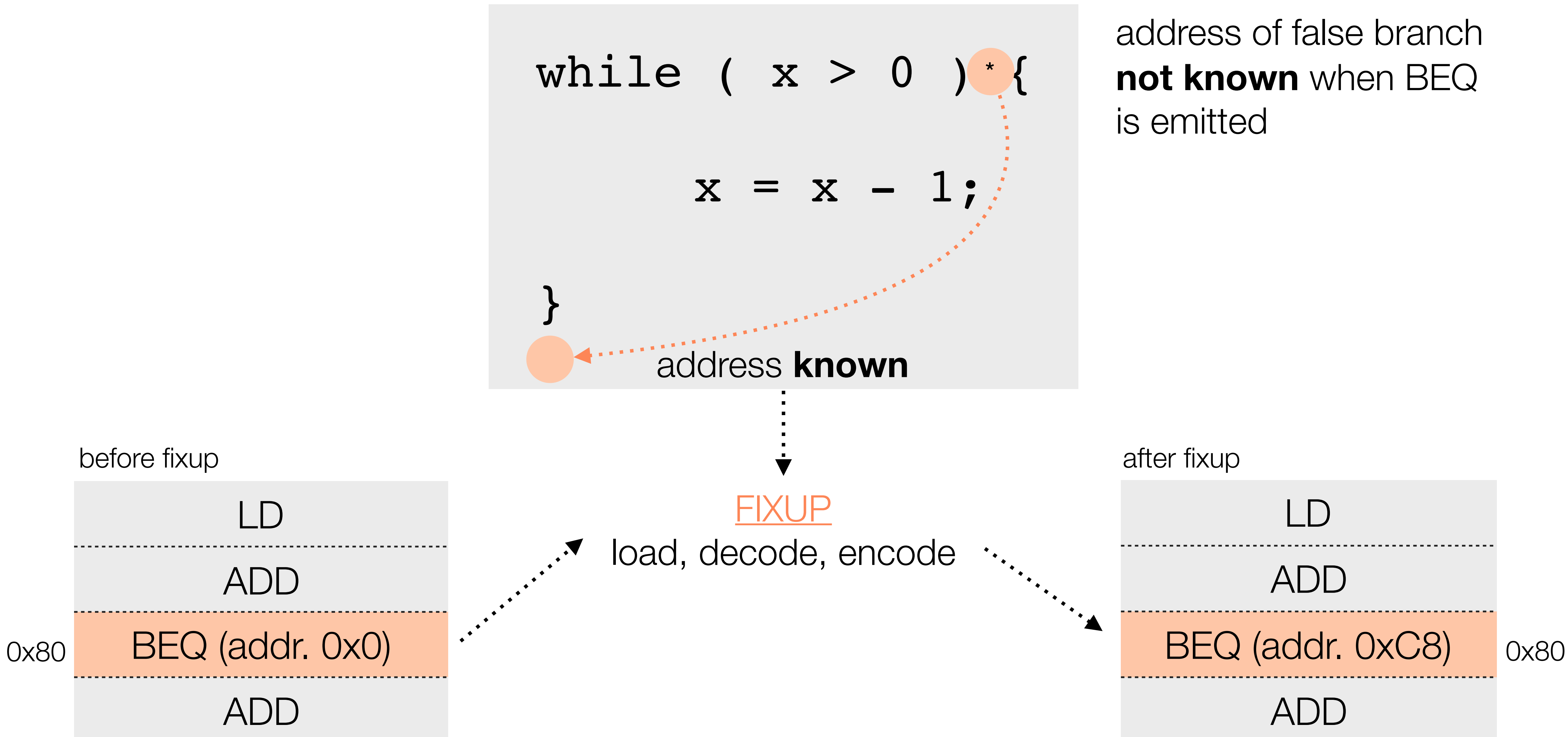
  2. fix up instruction as soon as address is known

     - load and decode instruction

     - encode and emit instruction with correct address (at old address)

Fixup Single Instruction

./selfie -c



address of false branch **not known** when BEQ is emitted

Remember address of wrong instruction:

```
branch_forward_to_end = 0x80
```

Fixup Multiple Instructions - Return

`./selfie -c`

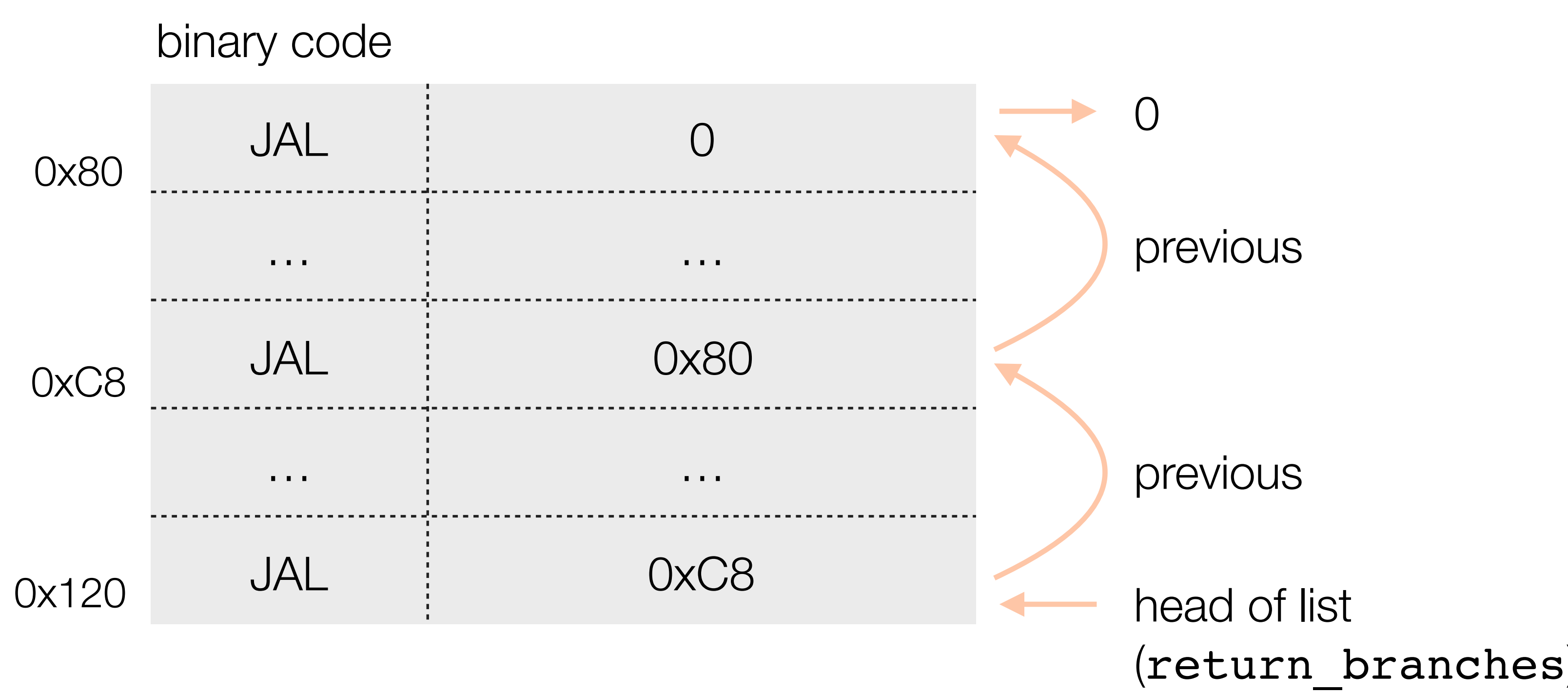
- <u>return statement</u>
  - compiling return → **JAL** to first instruction of epilogue that is not known at this point
  - a procedure might have **any number** of return statements

- Keep a **linked list of JAL instructions** that will be fixed up later by going through the list and fixing up each instruction
  (this also deletes the list)

Fixup Linked List of JAL Instructions

`./selfie -c`

- **Where is this list stored?**
  - in the **binary code**, using the **instructions** to store the list (elegant solution by N. Wirth)
  - the bits designated for the destination address of an instruction are used to store the address of the **previous** instruction
  - the **head** of the list is stored globally (`return_branches`) (no nesting of procedure definitions)



Fixup Multiple Instructions - Procedure Calls

`./selfie -c`

- Procedure call
  - compiling call → **JAL** to first instruction of prologue that might not be known at this point (definition not parsed)
  - a procedure might be called **any number of times**
- Using same principle as for return statements - **linked list** in code
- **Nesting** of procedure calls is possible, therefore storing the head globally is not sufficient
  - use symbol table instead
  - associate the procedures name with the head of the list (procedure address)

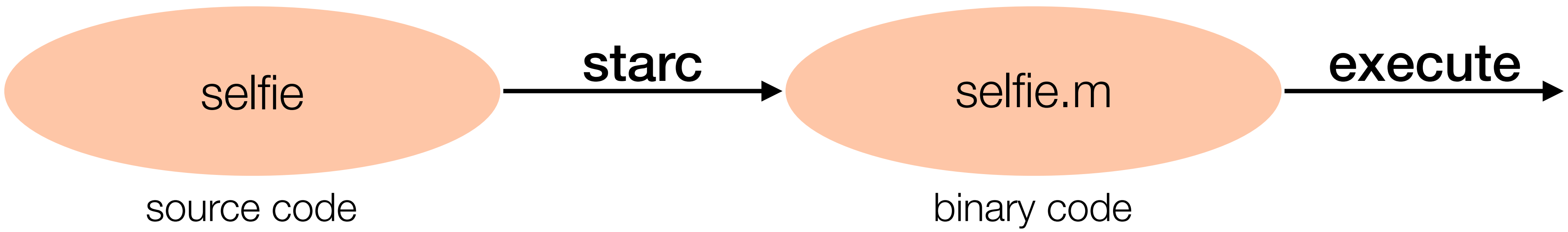
Assignment 6: for-loop

- Students can extend the language C* and introduce for-loops
  `for (i = 1; i < 10; i=i+1)`
  - very similar to while
  - think about when the parts in the parenthesis are executed (how to lay out in binary file)
- Hint: the tricky part is `i=i+1`
  - code must be emitted when the expression is parsed
  - requires a little jumping an fixing

Further Concepts & Selfie

Compilation,
Object-Oriented Programming,
Memory Safety,
Garbage Collection,
Bootstrapping Selfie

Compile Time and Runtime



- Compile time - program is executed to **generate code**
  - compilation done once
  - figure out as much as possible before execution
  - infinitely many programs can be created
- Runtime - **execution** of compiled program
  - heap and stack built for/by the program

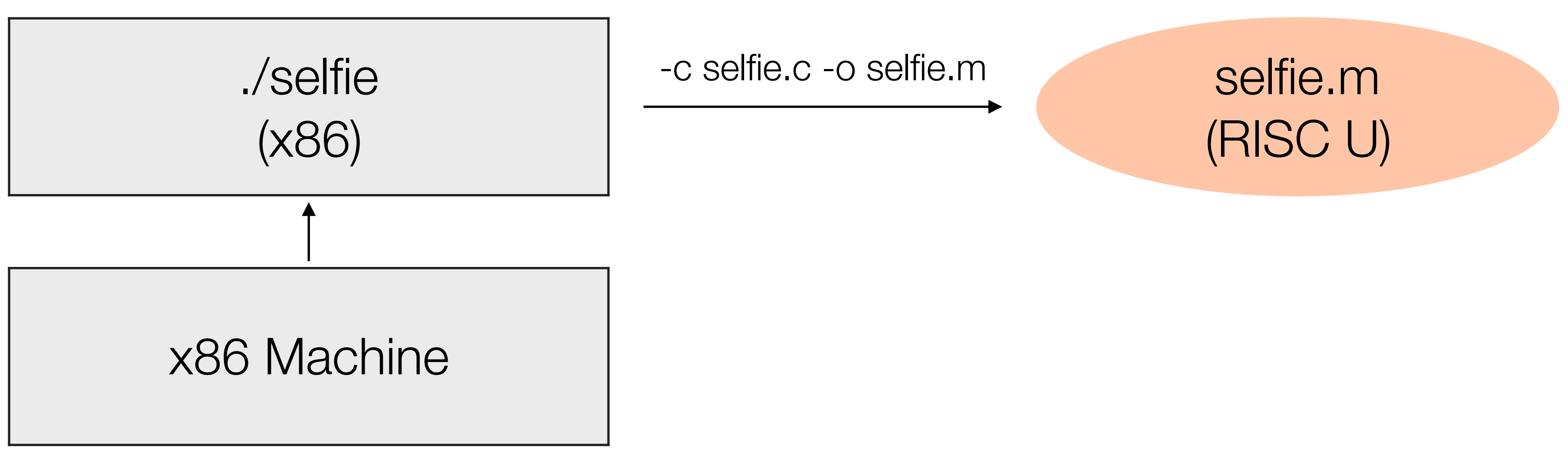
Compilation

Terminology

Cross Compilation

- A <u>cross compiler</u> outputs binary code that targets a different architecture than its own binary code does.

- For embedded systems, iOS and Android,…



Compiling Source Code

- So far we considered our code to be in a single source file.

```
…
uint64_t main() {
   return proc();
}

uint64_t proc() {
   return 1;
}
…
```

file.c

file.exe

Symbolic reference:
reference associated with a name/identifier (ex. `proc()`).

The address is not known at this point and the symbol table entry for `proc2()` contains the last `jal` instruction that needs to be fixed up.

Resolved into **direct address** as soon as address is known.

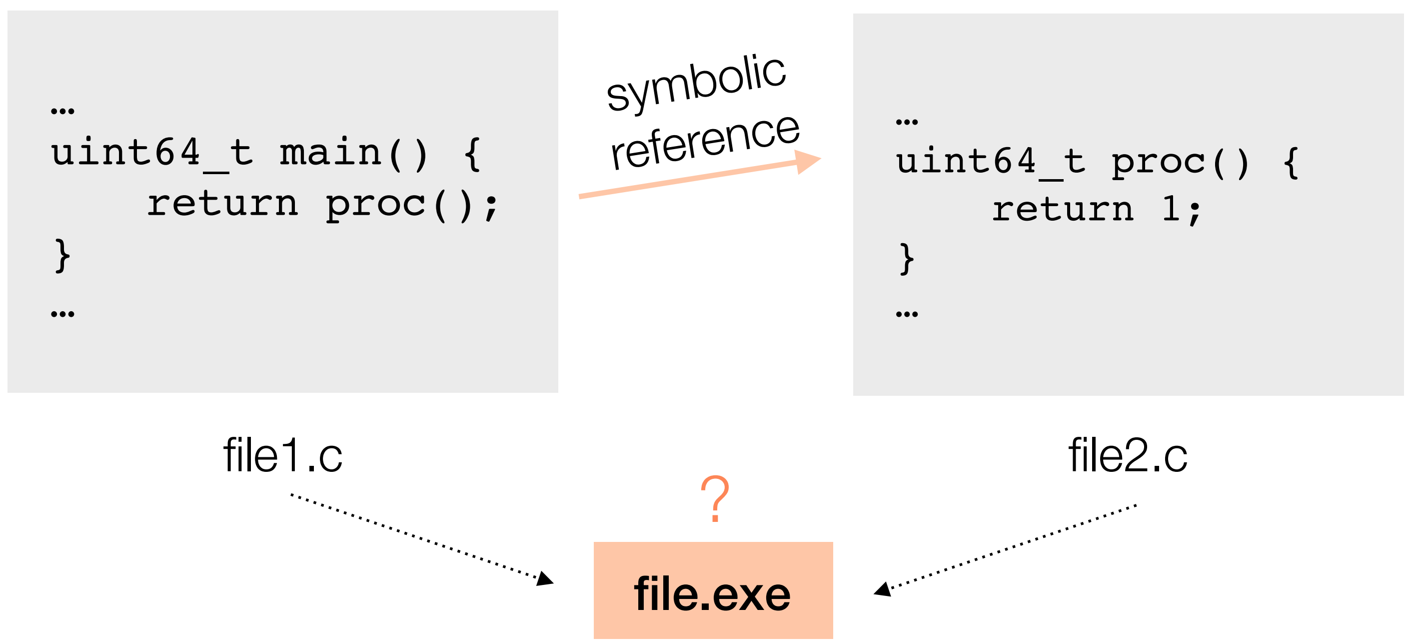
Direct address: different namespace, the one of machine instructions

Compiling Source Code

- To manage code better it is convenient to have **multiple source files** contain different parts of the code. However, we still want to use (reference) procedures and variables from those other source files.

- Forward declarations are unavoidable when we reference code from other files.

- But the necessary fixup cannot be performed until the file containing the definition of the referenced procedure/variable is compiled. Therefore a file can still contain symbolic references after it has been compiled.



Independent vs Separate Compilation

- **(Independent) Compilation:**
  compiling two separate source files that do not reference each other.

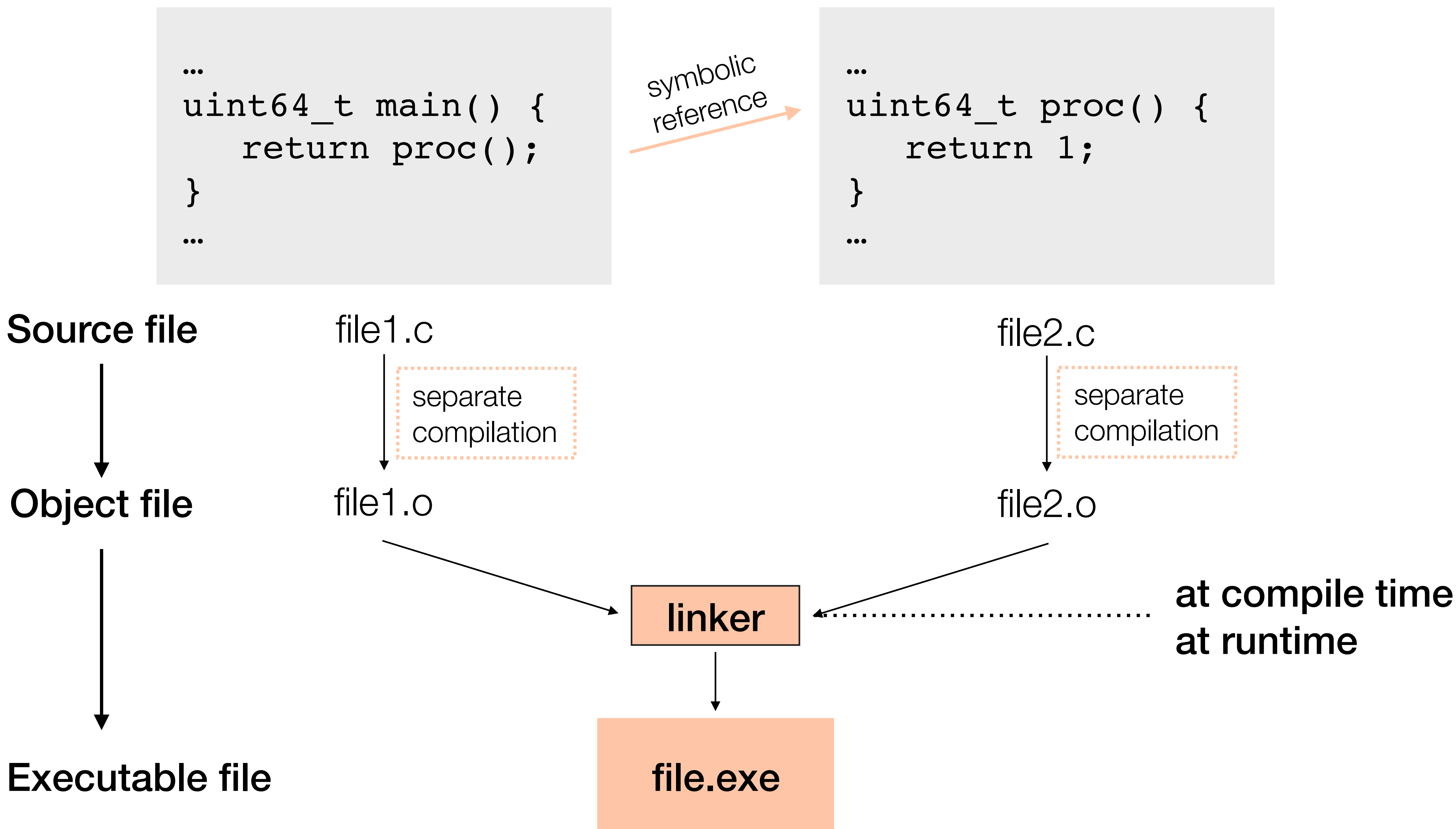
- **Separate compilation:**
  compiling two separate source files that reference each other.

Linking

*The fixup process that resolves **symbolic references** into **direct addresses** across **object files.***

- Compiling source code yields object code that might hold symbolic references.
- Symbolic references are part of unresolved fixup chains that are stored in the symbol table.
- The linker resolves all symbolic references and generates executable binary code.
- Linking can be done at compile time (static) but also at runtime (dynamic loading and linking)

Linking



…
uint64_t main() {
return proc();
}
…
symbolic reference
…
uint64_t proc() {
return 1;
}
…
Source file
file1.c
file2.c
separate compilation
separate compilation
Object file
file1.o
file2.o
linker
at compile time
at runtime
Executable file
file.exe

Object- and Executable Files

- **Object files**
  - might contain symbolic references (associated with name)
  - more generic than executable files
  - to resolve the references, the information from the symbol table is needed (fixup chains).

object file = `binary code` + representation of symbol table

in some format

- **Executable File**
  - no symbolic references
  - special case of object file

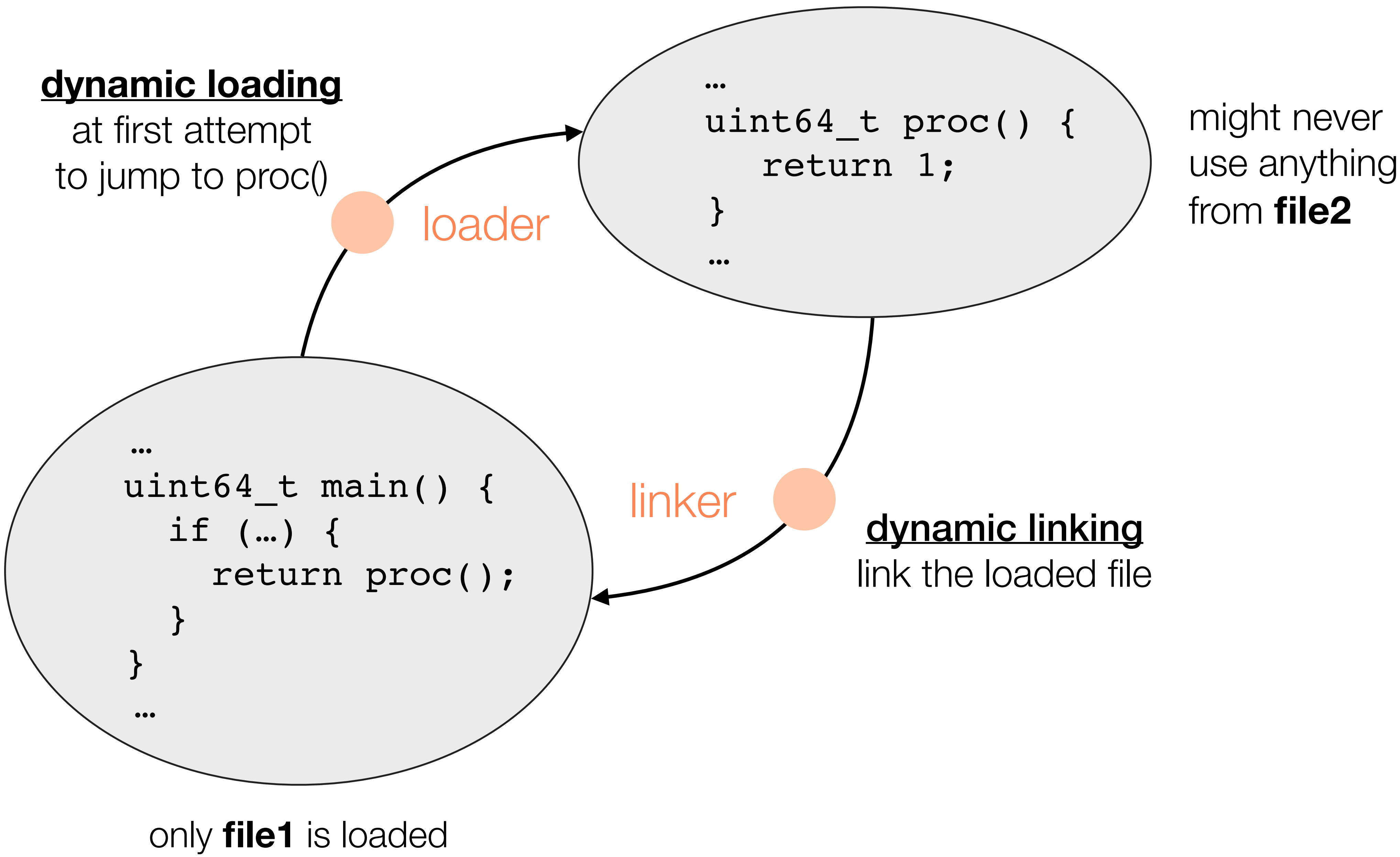
Dynamic Loading and Linking Java

- The file that gets executed does not have to be completely resolved. Under some circumstances the execution of an object file makes more sense.

- **Why?**
  - the referenced code is never used at runtime, so why link it? (this is specific to each run)

- **How?**
  - Code is loaded and linked at runtime, as soon as the executed program tries to jump to an unresolved address.

Dynamic Loading and Linking Java



Dynamic Loading and Linking Java

- Using this concept we could start from an **empty file** combined with the information from the **symbol table**.

1. Attempt to load the main() procedure
   - find the code in memory
   - load the code into main memory
     and execute it

2. Attempt to use another unresolved reference
   - find code in memory
   - load code into main memory
   - link code and continue execution

Just-In-Time (JIT) Compilation

A JIT compiler performs ***binary translation*** *of* ***hotspots*** *in* ***bytecode****.*

- It **is not** used with high-level languages.
- It is part of a runtime system such as the JVM.
- Used to increases runtime performance of code.

Bytecode

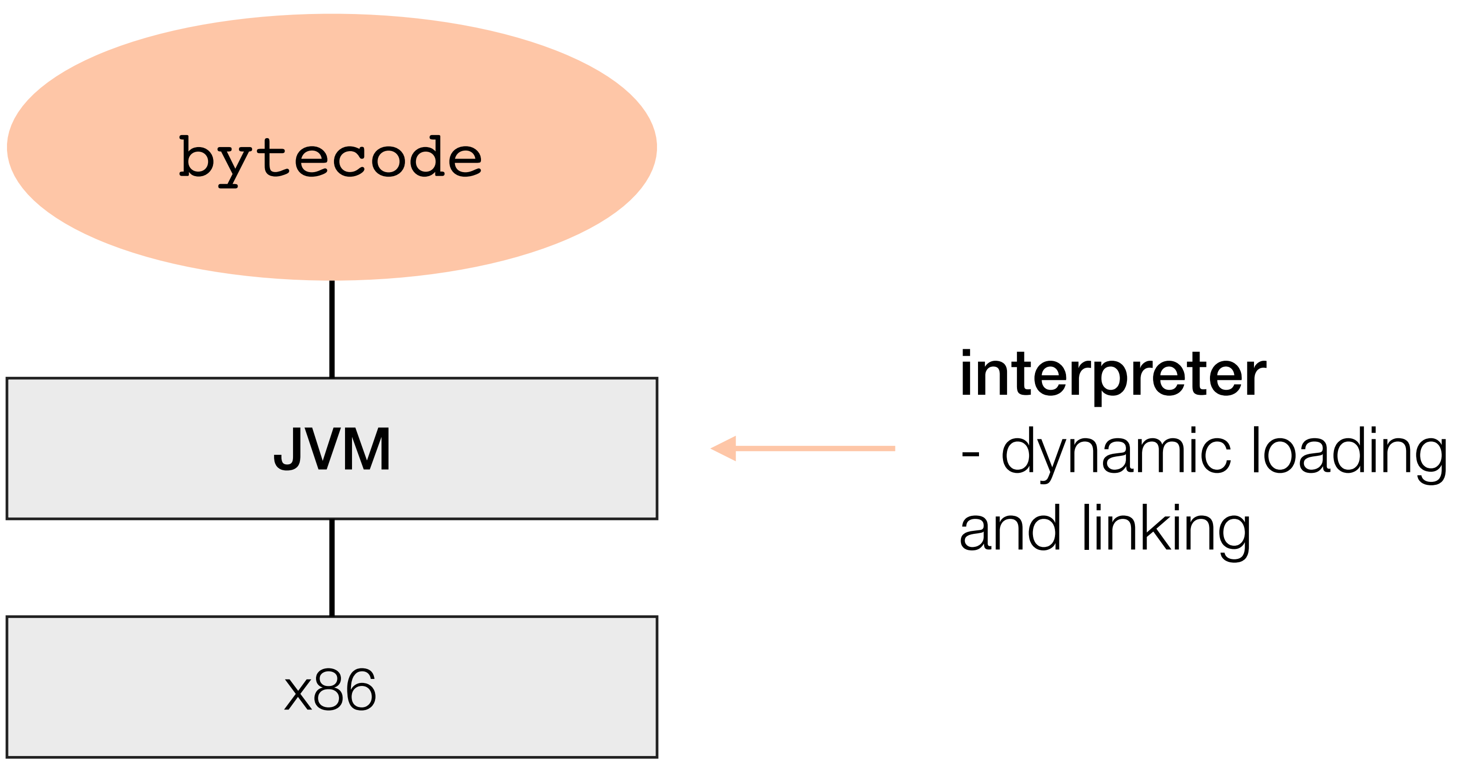
java source code		java compiler		bytecode
`file.java`	→	javac	→	`file.class`

- Bytecode is code for which there exists no processor that can execute it.
    - a (virtual) machine is needed that can load and execute byte code.

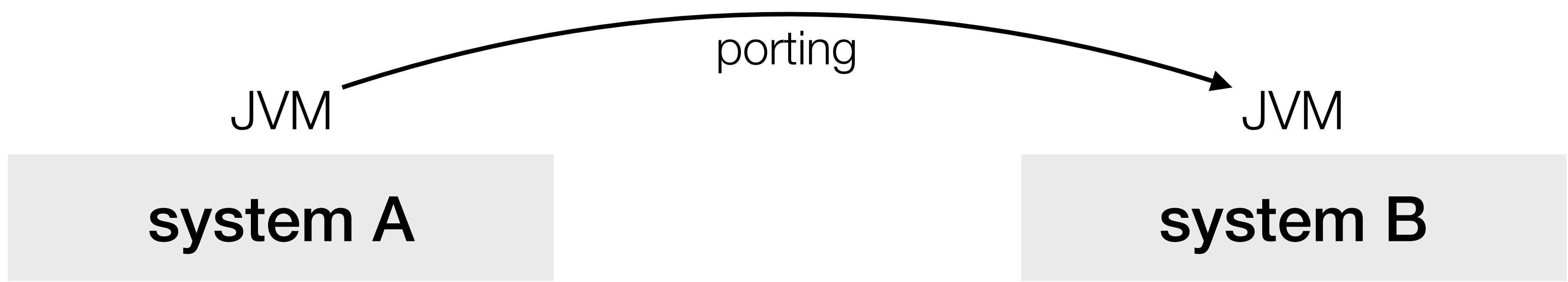
- It is **independent** of the architecture it runs on.

Java Virtual Machine JVM

- The JVM is an emulator of a byte code processor.
  - it executes bytecode through interpretation
- It is a runtime system more complex than an OS.



Bytecode



- **C**
    - change backend of compiler so it produces binary code for a different architecture

- **Java**
    - compiler remains unchanged
    - port JVM to run on different architectures (portability)

- **Porting the JVM**
    - change backend of the compiler that compiles the JVM for system A so it compiles the JVM for system B

The Problem Interpretation is slow, Execution is fast

- Ex.: Interpreting a while loop in bytecode hundreds of times is slow.
- **Make it faster**
    - instead of interpreting the bytecode feed it a compiler that translates it to binary code that can be **executed** by the machine it is running on (just-in-time).
    - It is <u>hotspots</u> in bytecode that get translated to binary code
- A **hotspot** is binary code that gets executed often (enough) within a certain amount of time.

Object Orientation

and Polymorphism

Object-Oriented Programming OOP

- Object-oriented programming is a programming paradigm
- Crucial factors of OOP
  - classes - abstract data types
  - structure - hierarchy
  - and most important - polymorphism

Polymorphism of Procedures

*The actual procedure implementation is **selected at runtime according to the data types** of the parameters the procedure is called with.*

- The actual type of the parameters is not known at compile time

  → compiler does not know what procedure to call

- The runtime system needs to decide which procedure to call

  → determining the data types and choosing the implementation (<u>dynamic binding</u>)

- On the level of procedures we are talking about **polymorphism.** But in object-oriented languages this concept is implemented using **dynamic binding.**

Memory Safety

Memory Safety

Memory safety is a property of program execution that is concerned with memory ***access violations.***

- We distinguish between:
  - spatial safety - memory is legally allocated
  - temporal safety - no access to undefined memory

Memory Safety

- This problem does not exist in Java since you cannot…
  - … access objects you deleted
  - … perform illegal access into arrays
- Therefore Java is a memory-safe language
  - You cannot write a program that accesses memory outside its own space (defined and allocated)

Memory Safety

What does it need?

- no pointer arithmetic or operators that access memory directly (only internally)
- array range checking at runtime
- using references
- a garbage collector when memory is deallocated

Memory Safety and Garbage Collection

- Deallocating memory
  - **using free** → memory safety would be lost because dangling pointers might exist
  - a **garbage collector** ensures memory safety when dealing with memory deallocation

Garbage Collection

and Memory Deallocation

Memory Deallocation

- Many but **finitely many** addresses
- Memory deallocation only necessary if addresses need to be reused
  - only dead memory can be reduced
  - not necessary for short running programs
  - **tip**: only use free in long running programs (server,…)
- <u>Manual memory management</u>
  → programmer explicitly deallocates memory (free)
- <u>Automatic memory management</u>
  → garbage collector reclaims dead memory

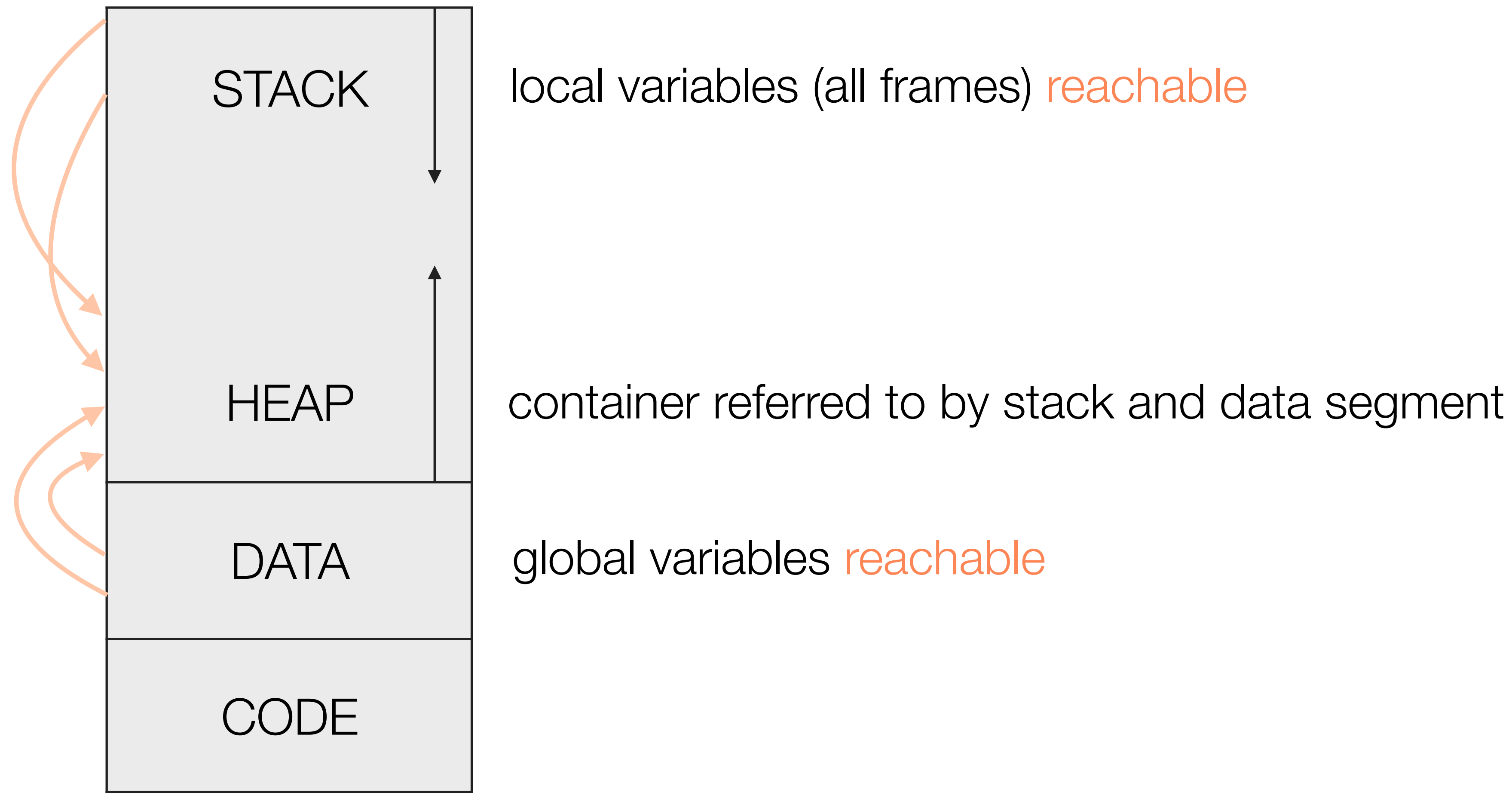
Garbage Collection

> *A garbage collector automatically* ***reclaims dead memory*** *so it can be reused.*

- Garbage collection is not restricted to object-oriented languages
- A garbage collector tries to find **proof** that memory is **dead**
- A garbage collector computes reachability
  (liveness is undecidable)
- A garbage collector computes the **set of unreachable addresses**
  - check if an address is still reachable
  - are there any direct or indirect references?

Garbage Collection Check References

- Look for references on the **stack**, **heap** and in the **data** segment
- And the transitive closure - all possible references (tracing)



Garbage Collection in C

- In C we are able to talk about the actual value of pointer

- **Conservative garbage collector (tracing)**
  - an approach to implement a garbage collector for C and C++
  - see everything on stack and in data segment as pointer
    (+ transitive closure)

Garbage Collection

- **Advantages**
  - garbage collection rules out <u>dangling pointers</u>
- **Disadvantages**
  - Garbage collection does not protect against <u>reachable memory leaks</u>
  - garbage collections consumes additional resources
    → might have a negative impact on performance

Reachable Memory Leaks

- Memory leaks are caused by incorrect memory management
  - not releasing memory that is not needed or accessible any more
- Reachable memory leaks occur when memory is still reachable but not accessed any more
  - hold on to references (i.e. hoard in hash table)

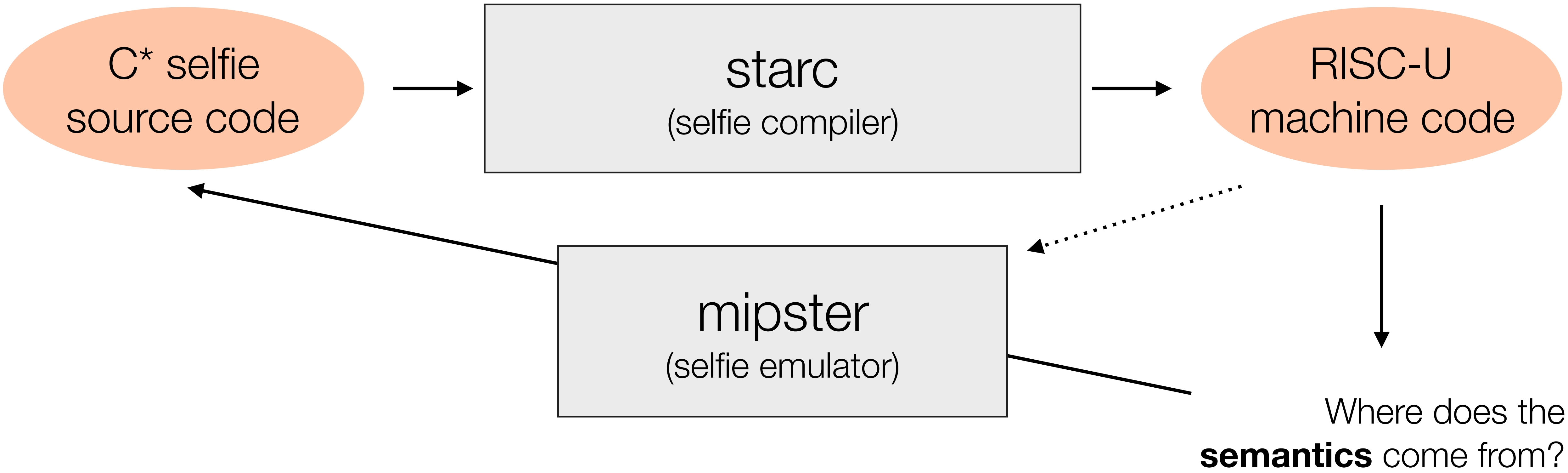
Bootstrapping Selfie

Semantics of Selfie?

Where does the **semantics** come from?

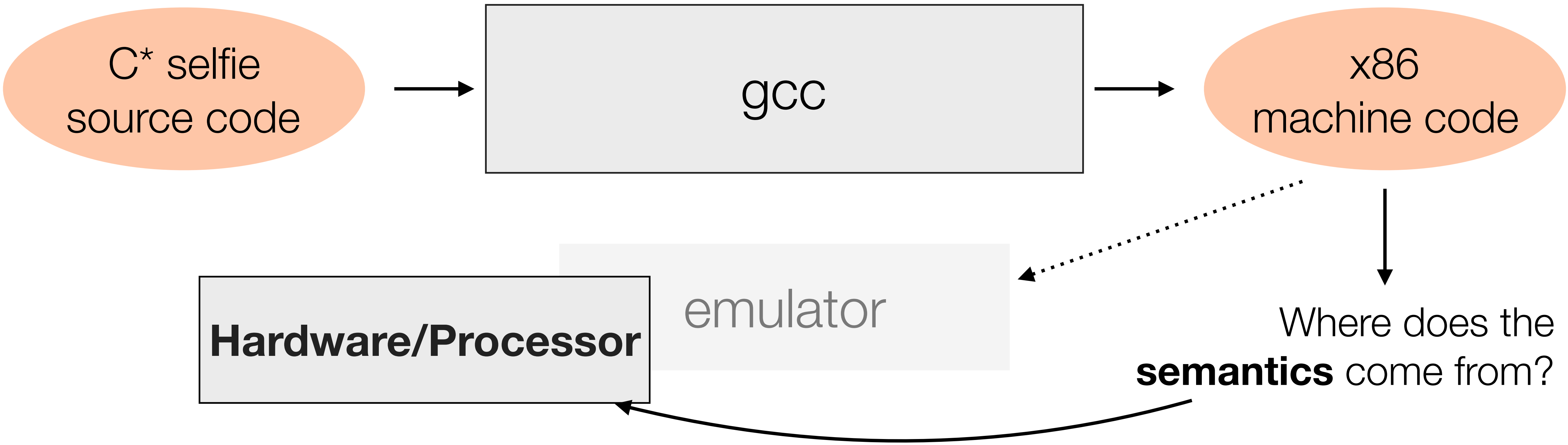
- "The semantics of an instruction is determined by how the processor implements it" - Semantics of RISC-U
- Machine code is a sequence of instruction, therefore the processor that executes it determines its semantics.
- Execute RISC-U code when only x86 processor available?
- Fortunately, selfie also features an emulator that can interpret RISC-U instructions.
  - selfie defines the semantics of selfie → **cycle**
  - we need executable of selfie to compile and interpret selfie.
    Where does this executable come from? → **cycle**

Semantics of selfie? Cycle



- We define the semantics in selfie:
  - how code is generated and **interpreted** by selfies RISC-U emulator
- The semantics of `selfie.c` is in `selfie.c` - **cycle**

Breaking the Cycle



- We break the cycle when we **compile** the first selfie.c **with gcc** to get our first executable of selfie.
- x86 code does not need an emulator. It is executed directly on hardware.
- Therefore the semantics is ultimately defined by the system/hardware it runs on.
- Processor errors might affect the semantics of code.

Bootstrapping

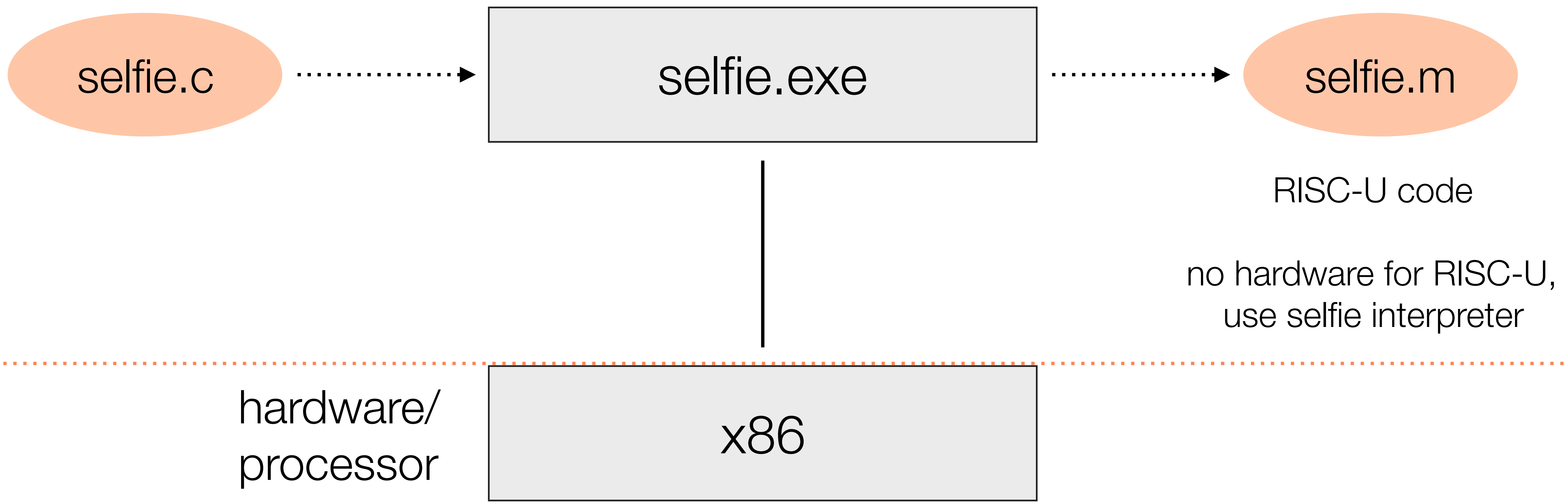
We only have the **selfie.c source file**

1. Execute **gcc**
   compile selfie.c to get an x86 executable version of selfie

Bootstrapping

2. Execute **selfie.exe**
   compile selfie.c to get an RISC-U executable version of selfie

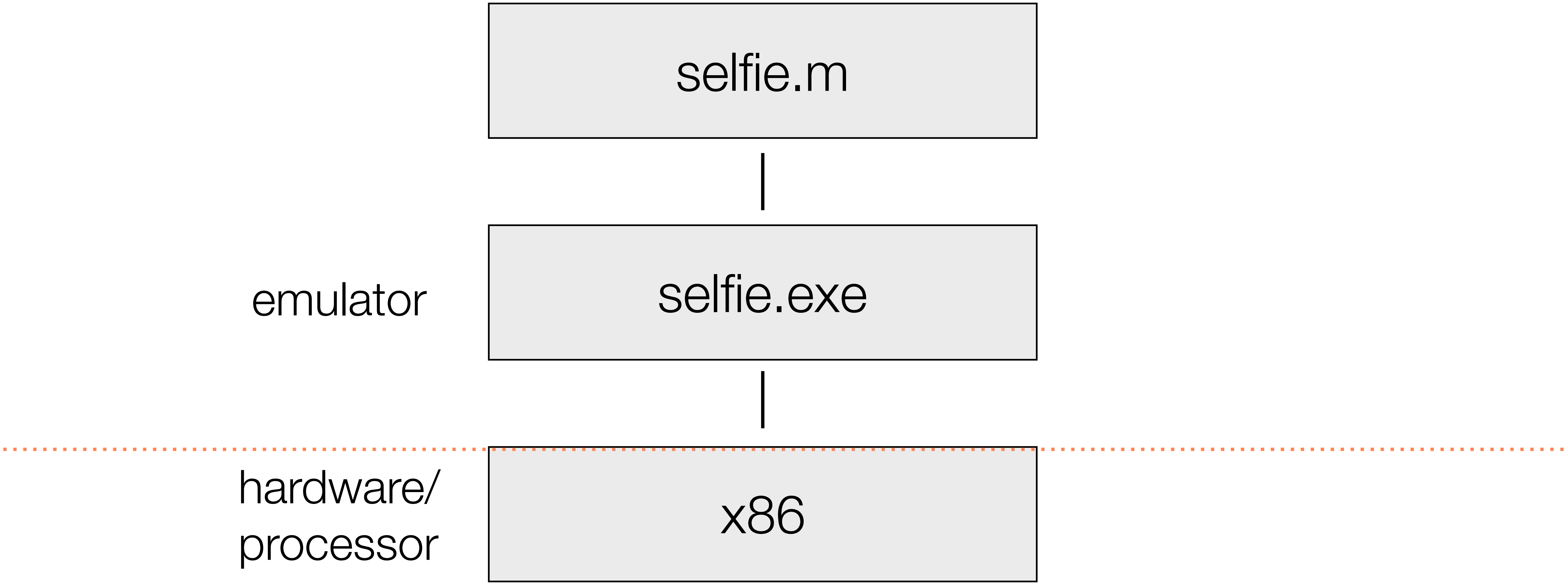
   → **self-compilation**



Bootstrapping

3. Execute **selfie.exe**
   start the selfie emulator and have it **interpret** selfie.m

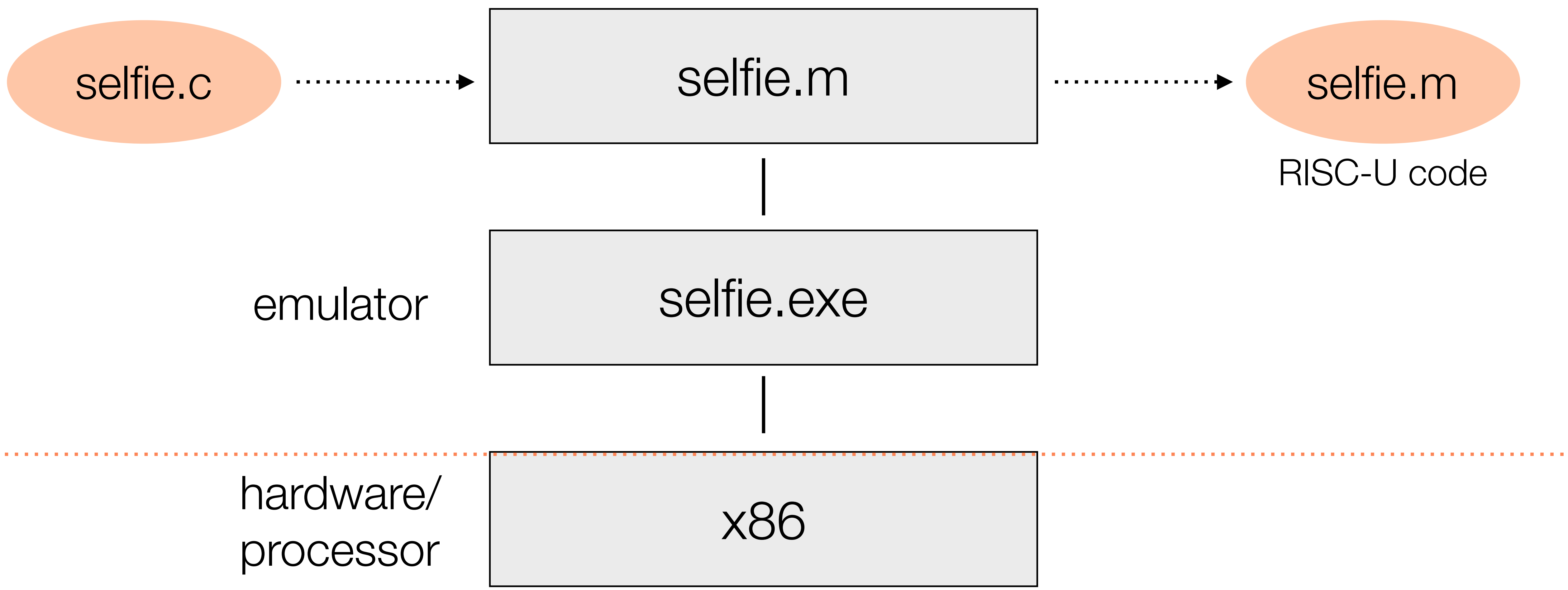
   → **self-execution**



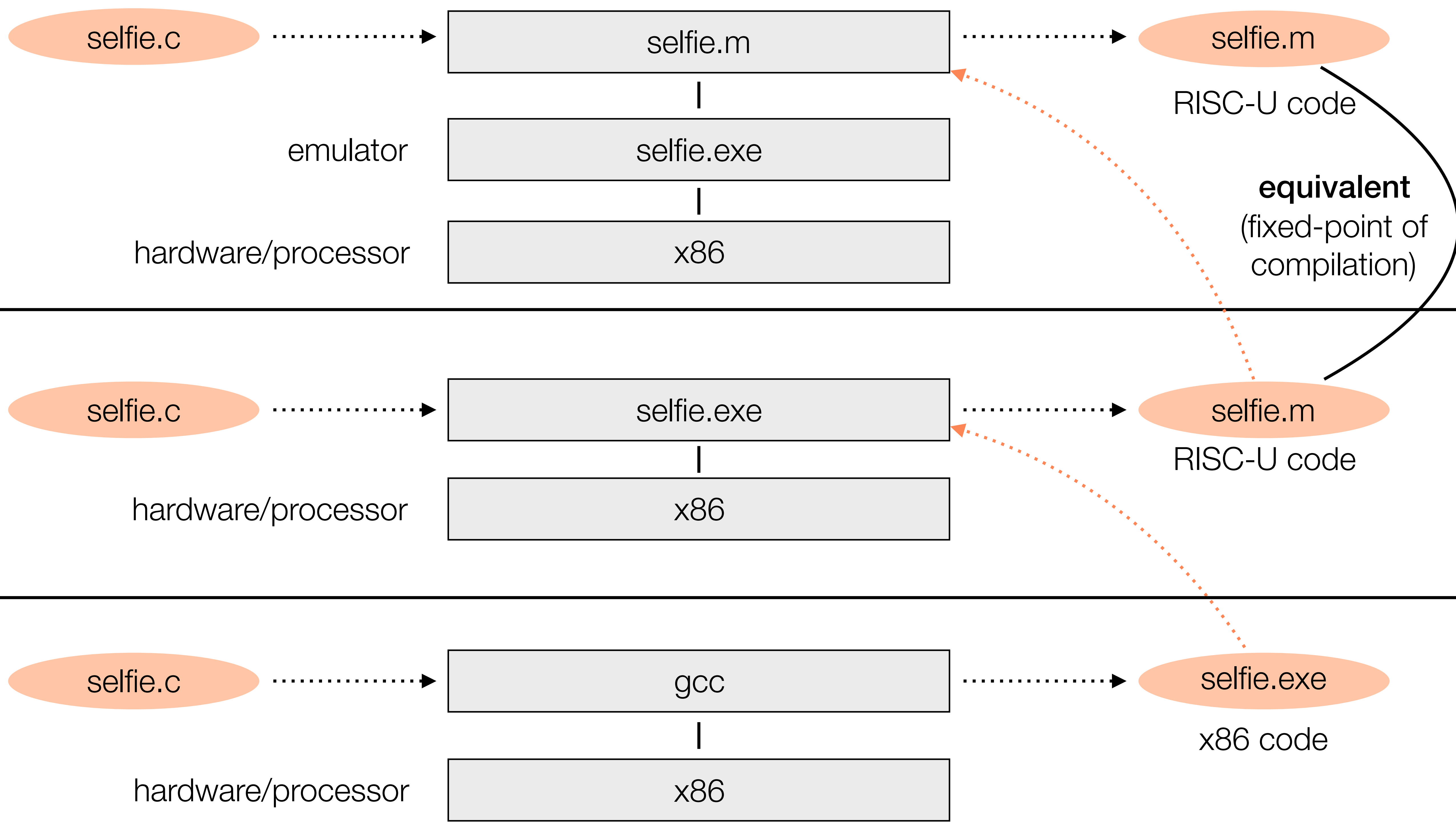
Bootstrapping

4. **Interpret** selfie.m
   compile selfie.c to get an RISC-U executable version of selfie

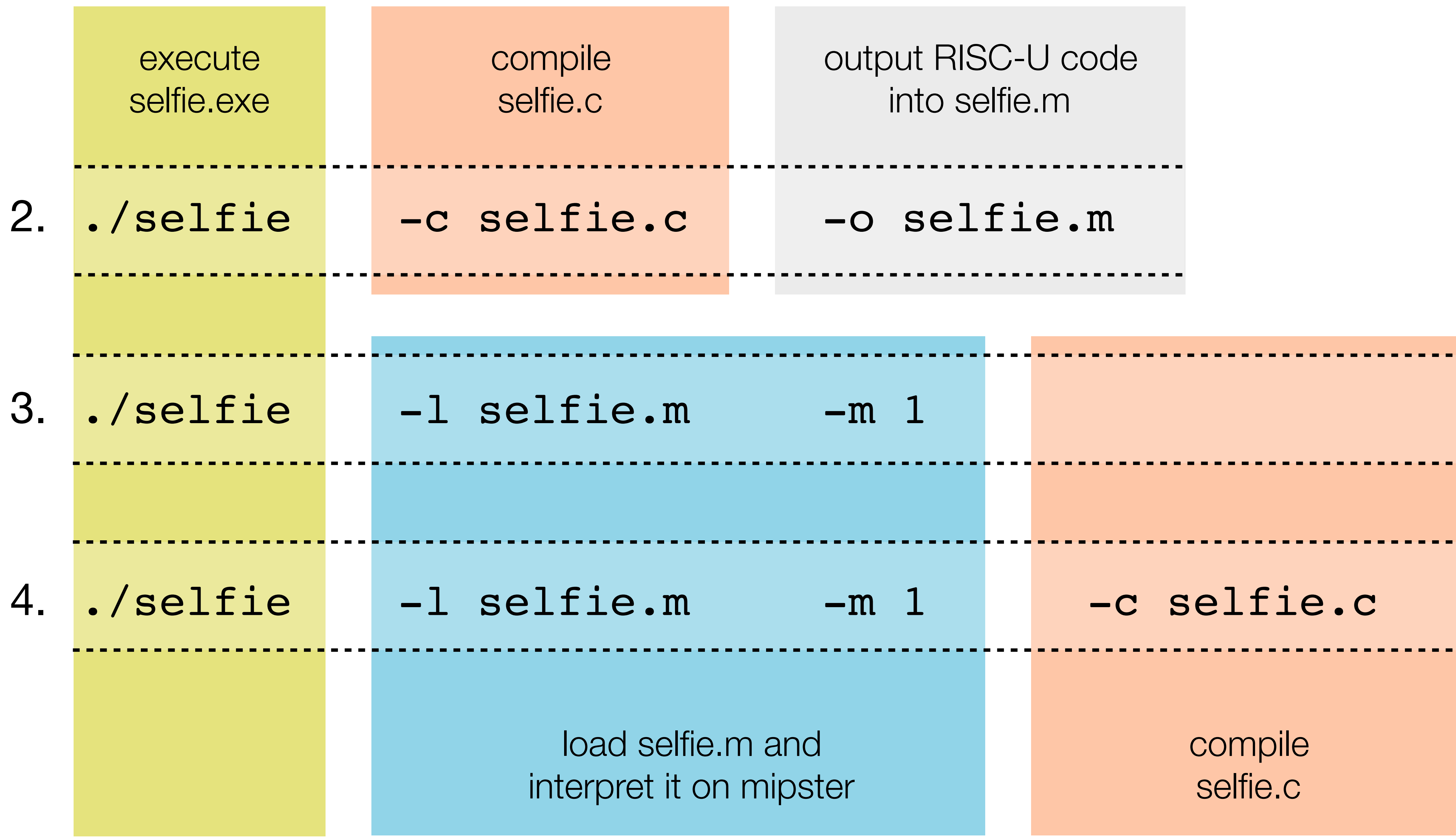
   → **again self-compilation**



The Whole Picture



Command Line



- Compiled code can be executed directly:

```
./selfie    -c selfie.m    -m 1    -c selfie.c
```

Introduction to

Operating Systems

Overview

- **Mipster** → emulation, context, process
- **Operating Systems** (concurrency)
  - System Calls → protection, API/ABI, wrapper function, bootstrapping
  - Emulation vs Virtualization → interpretation is slow
  - Hypster → context switch, cpu virtualization
  - Memory Management → address space/translation, memory virtualization, paging
  - Process Management → process lifecycle, threads (race condition, mutual exclusion, concurrent data structures, false sharing)

Operating Systems

An operating system manages the ***execution*** *of a program on a machine.*

- We will extend and refine this definition and build an understanding of operating systems as we extend the minimal operating system support already implemented in selfie.
- However, before we address operating systems, we will learn how a program is executed by selfie.

Previously on…

- Introduction to compilers explained how source code written in C* is **translated** into an executable RISC-U binary.

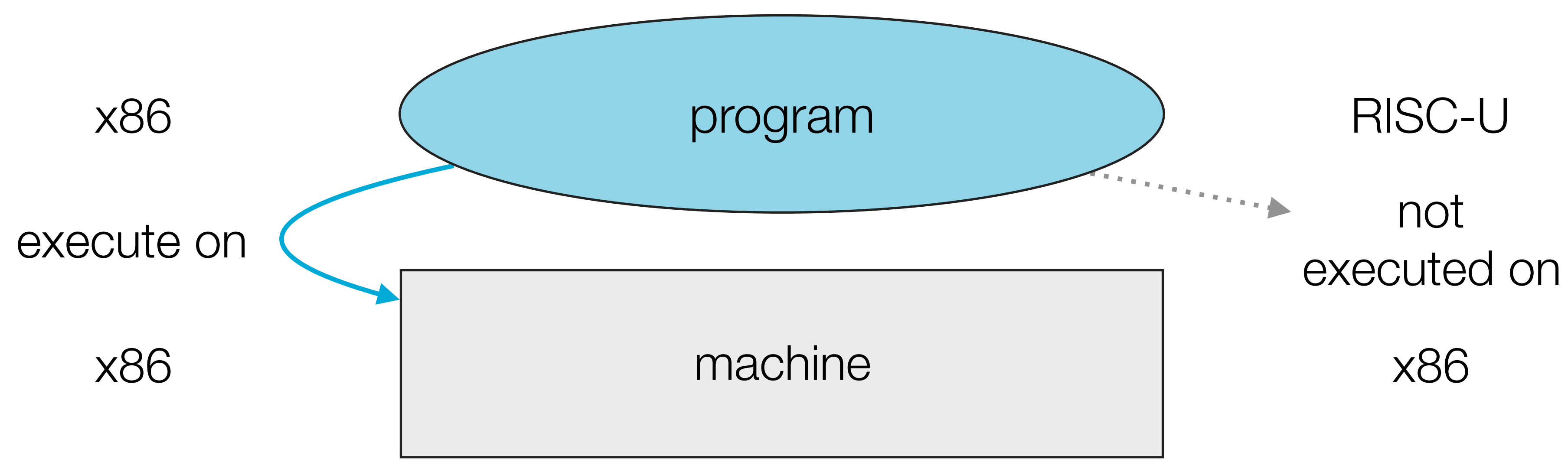
- Now we want to **execute** this compiled program.

Executing a Program

- The machine provides **resources** for the execution of a program (CPU, memory, I/O devices).

- The machine can only execute machine code that is written in its machine language, which is defined by the machine's <u>instruction set architecture</u> (x86, ARM, MIPS, RISC-V,…).

- We cannot execute RISC-U code directly on an x86 processor. Therefore, we will use **mipster**, a simple emulator for RISC-U code, as we did when explaining bootstrapping.



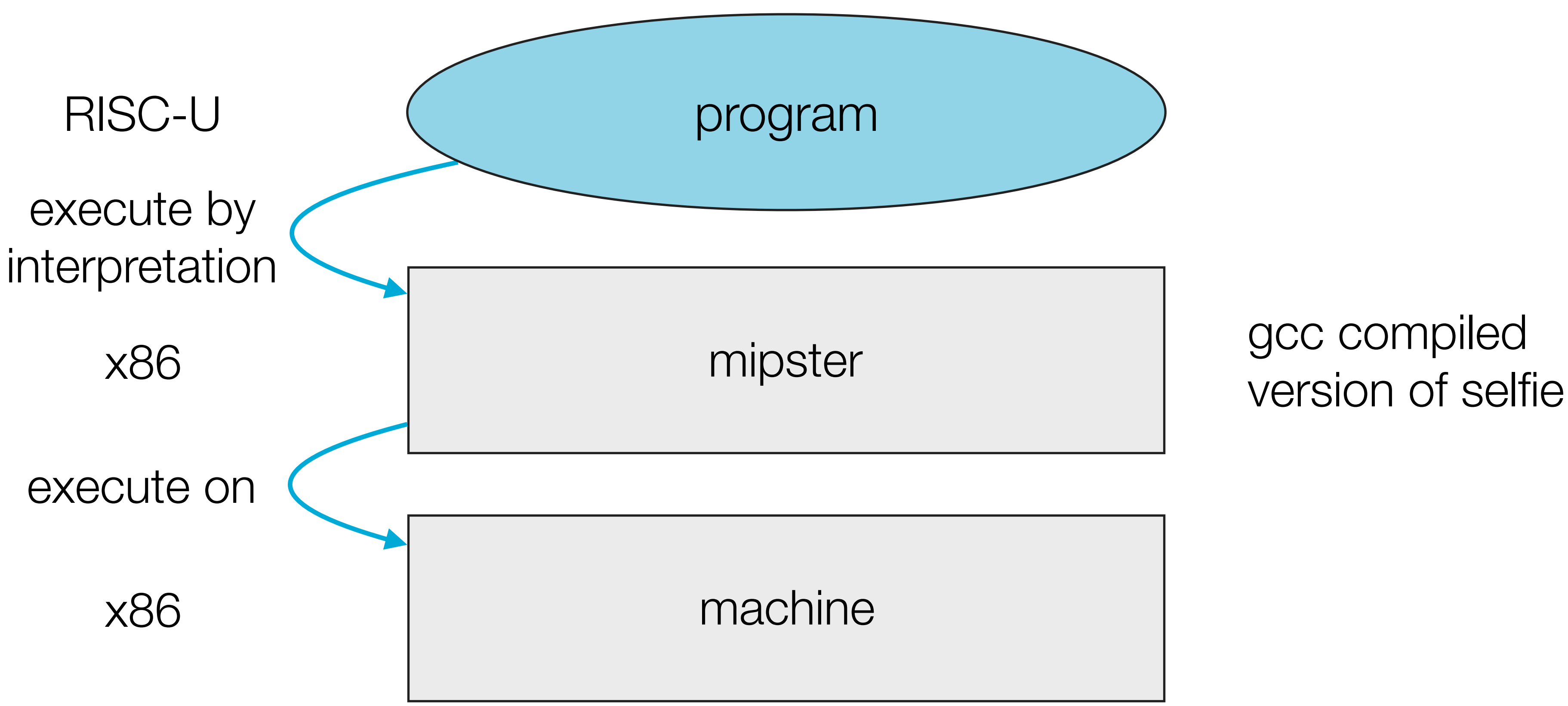
Mipster

Emulation,
Context,
Process

Mipster A RISC-U Emulator

- Mipster is an <u>emulator</u> that
  - emulates a RISC-U processor in software
  - can execute RISC-U code by interpreting it

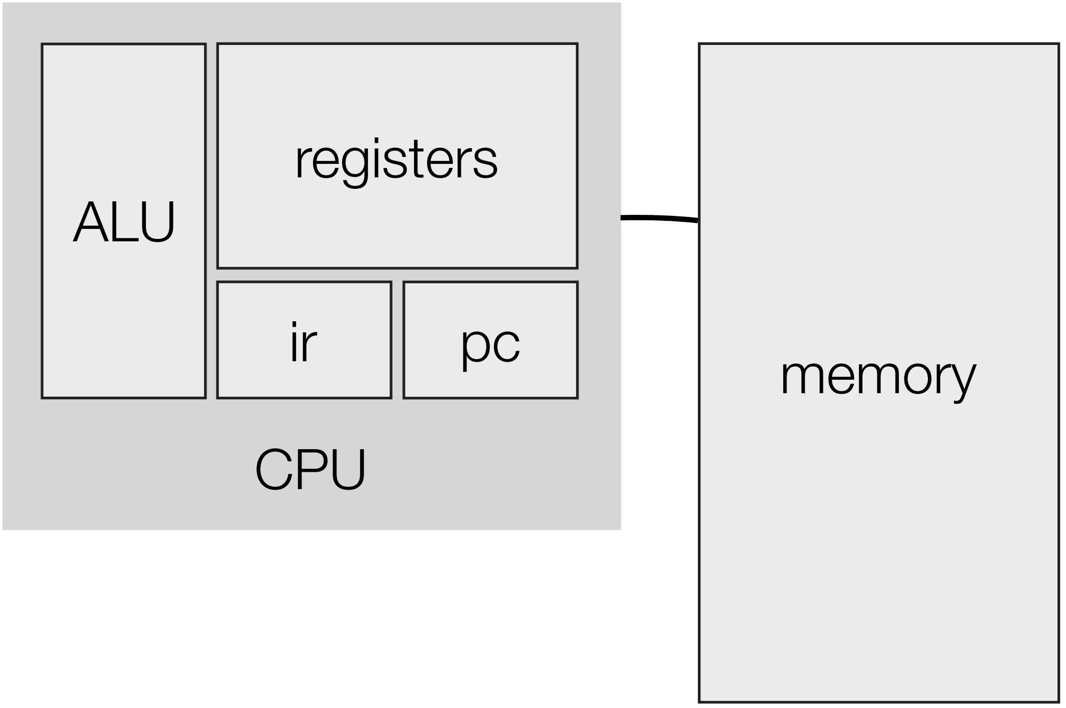
- To understand how mipster works we take a look at the machine it emulates.



Mipster A RISC-U Emulator

- Emulates "Von Neumann" machine
  - mipster creates a machine instance with 32 registers and 0-64 MB of memory

- **How does emulation in selfie work?**
  1. selfie executed with `-m` option starts mipster
  2. mipster creates a machine instance in software
  3. mipster starts emulation by implementing fetch-decode-execute cycle (fetch instruction, decode instruction, execute instruction by interpreting it)



Mipster A RISC-U Emulator

- We will use the following example to look at each step that is necessary to start mipster and have it emulate a program.

```
./selfie -c program.c -m 1
```

- The following slides explain the design and implementation of mipster and are best studied with the actual code next to them. However, it is not necessary to understand every little detail at this point.

1. Execute Selfie main()

`./selfie -c`

```
./selfie -c program.c -m 1
```

- Executing `./selfie` starts the `main()` procedure where:
  - the selfie system is initialized → `init_selfie(…)`, `init_library()`
  - selfie is run → `selfie()`

1. Execute Selfie selfie()

`./selfie -c`

```
./selfie -c program.c -m 1
```

- `selfie()` - This is selfie and all it can do.

- The arguments provided to `./selfie` determine what part of selfie is executed.
  - `-c` option starts the compiler → `selfie_compile()`
  - `-s` option disassembles binary code → `selfie_disassemble()`
  - `-l` option loads a binary → `selfie_load()`
  - options to **execute code on different machines** → `selfie_run(machine)` (`-m` option starts mipster)
  - invalid options print help → `print_usage()`

2. Create Machine Instance selfie_run(machine)

`./selfie -c`

- `selfie_run(...)` initializes mipster and creates a machine instance
- Machine state is represented by a set of global variables
- Two main steps
  - setting up **mipster**
  - creating something called a context

2. Create Machine Instance selfie_run(machine)

./selfie -c

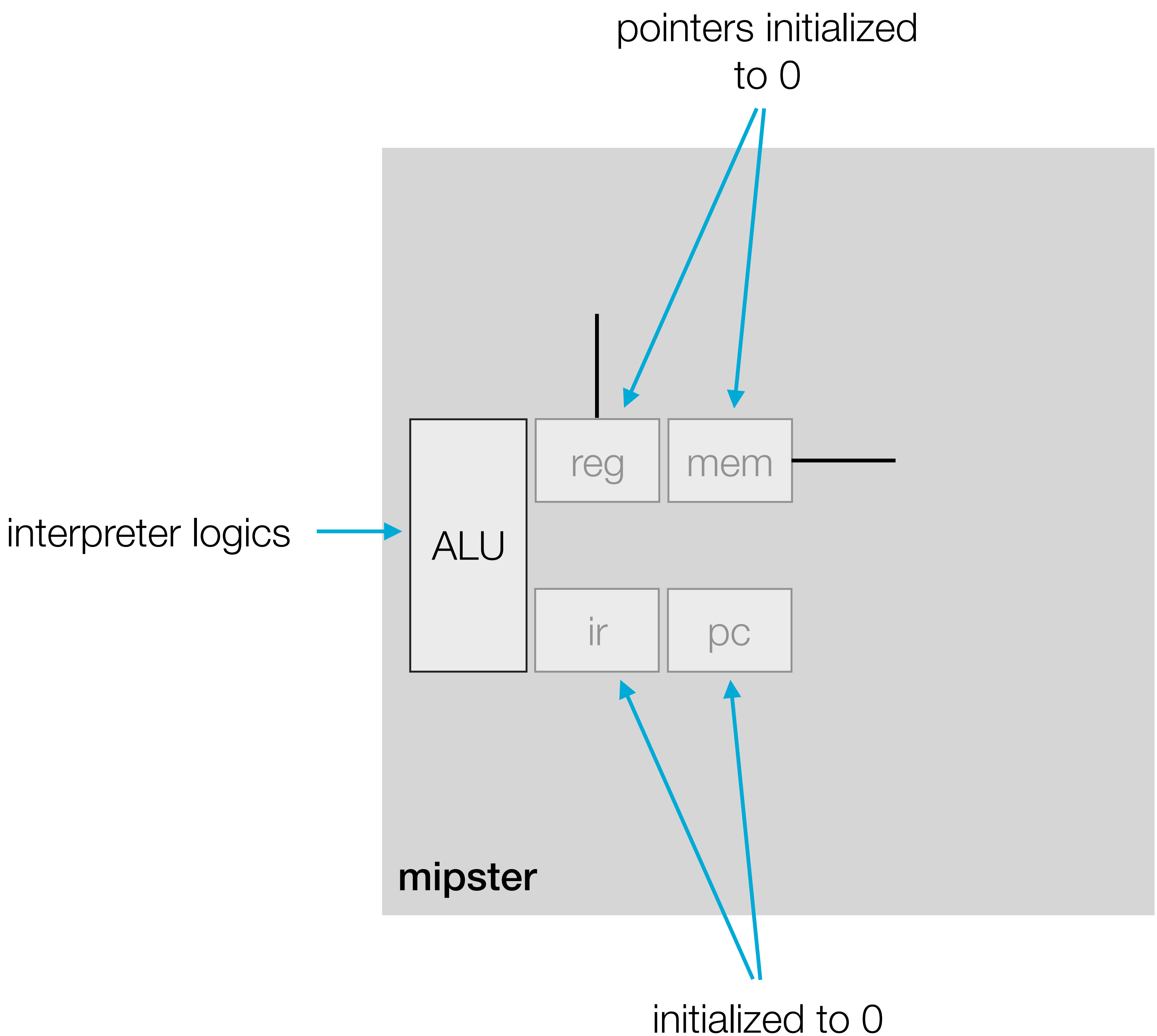
MIPSTER

```
./selfie -c program.c -m 1
```

- **Flags** are set depending on the machine that is run
- **Initialize memory size of machine** → global variable
  - `init_memory(atoi(peek_argument()))`
- **Reset/Initialize mipster** → global variables
  program counter(pc), instruction register(ir), pointer to memory(pt), registers
  - `reset_interpreter()`
  - `reset_microkernel()`

2. Create Machine Instance selfie_run(machine)

MIPSTER

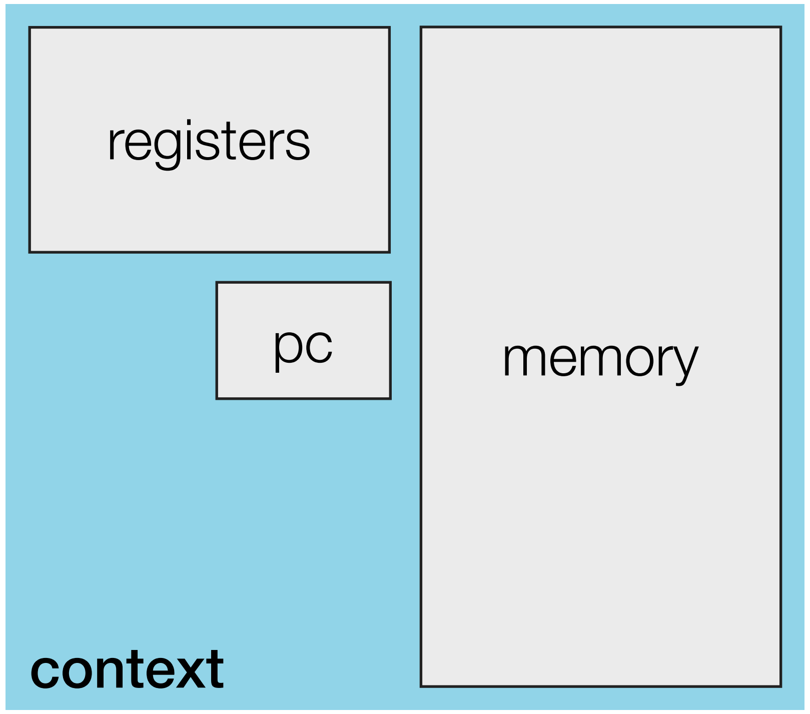


This is the bones of the emulator

Context For Now

For now - A container that holds part of the machine state.

- Selfies context is explained in more detail later
- Stores
  - program counter
  - registers
  - memory (page table)
  - ,…
- A context is uniquely identified by its **address**



machine state

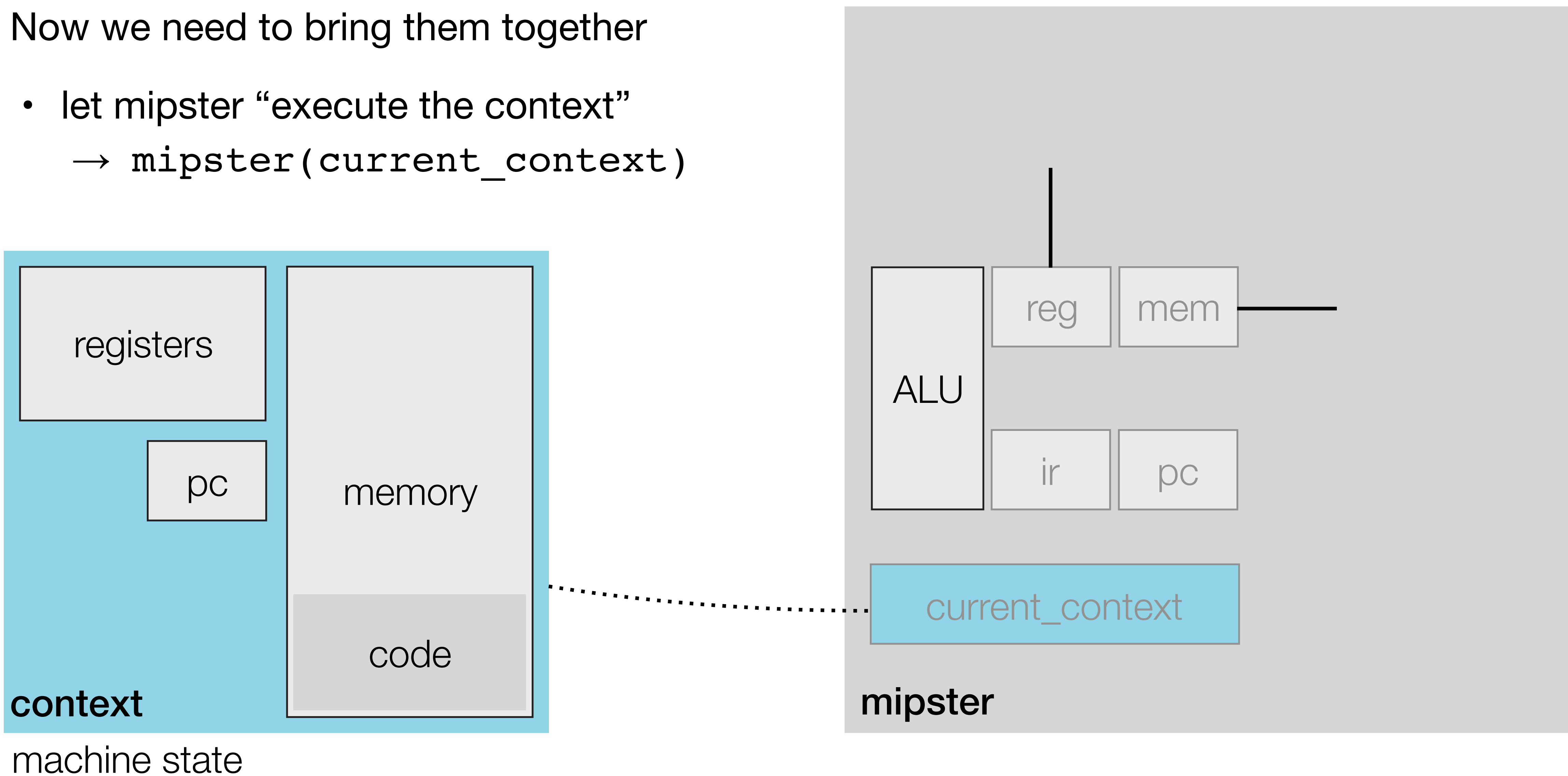
2. Create Machine Instance selfie_run(machine) ./selfie -c

CONTEXT

- **Create a context**
  - allocate space for the context and its components (memory, registers) → `allocate_context()`
  - the machine for which the context is created is the **parent** of that context
    - `MY_CONTEXT` to the machine
- **Upload binary** into the current context → `up_load_binary()`
- **Set binary name** as first argument that will be passed to context → `set_argument()`
- Pass **name** and remaining **arguments** to context
  - arguments for the emulated program → `up_load_arguments()`

2. Create Machine Instance selfie_run(machine)

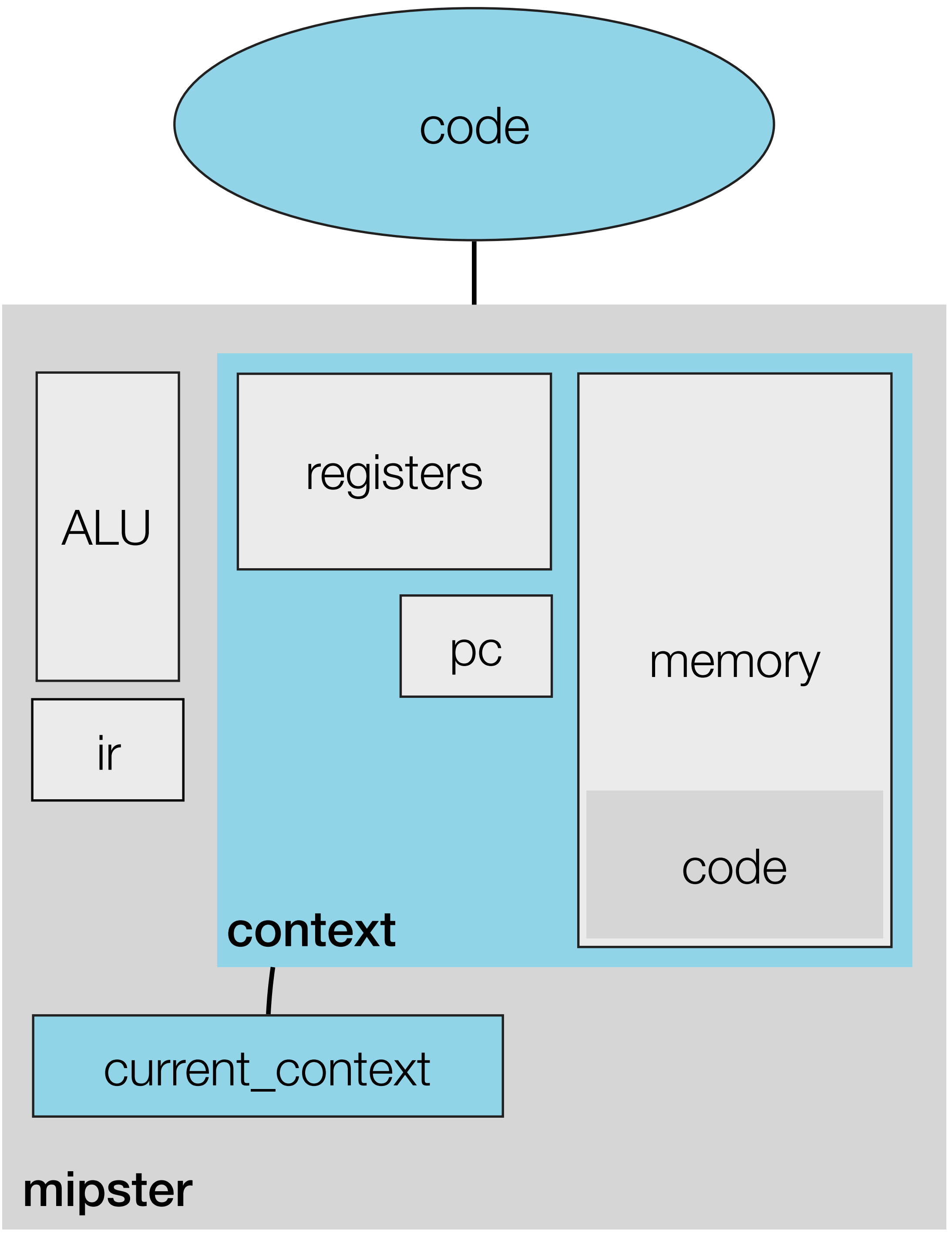
- So far we have
  - an new and "empty" **machine**
  - a **context** representing part of the machine state
- Now we need to bring them together
  - let mipster “execute the context”
    → `mipster(current_context)`



3. Start Emulation

`./selfie -c`

- Mipster is provided with the context it is supposed to execute
    - `mipster(to_context)`
- A context switch is performed
    - load the context into mipster and execute it
        - `mipster_switch(to_context)`



“Why create a context in the first place and not set up mipster right away?”

“The concept of a context is part of the operating system functionality already provided by selfie. It will make what comes next a lot easier.”

3. Start Emulation mipster_switch

`./selfie -c`

- The heart of mipster is the procedure `mipster_switch(to_context,…)`
- It is composed of 3 parts
  - the actual **context switch** where the contents of the context are loaded into the machine.
    → `do_switch(…)`
  - the **execution** of the context until the occurrence of an exception (system call, timer interrupt,…). The implementation of the von Neuman cycle.
    → `run_until_exception()`
  - **saving** the context before returning to `mipster()` after an exception (storing machine state back into context).
    → `save_context(…)`

3. Start Emulation mipster(to_context)

./selfie -c

After **TIMESLICE** many instruction execution will be interrupted

Mipster **switching to and executing** the context

Not important yet (the created context is always **MY_CONTEXT**)

Exception handling - only an exception that yields an **EXIT** breaks the loop and exits the emulation

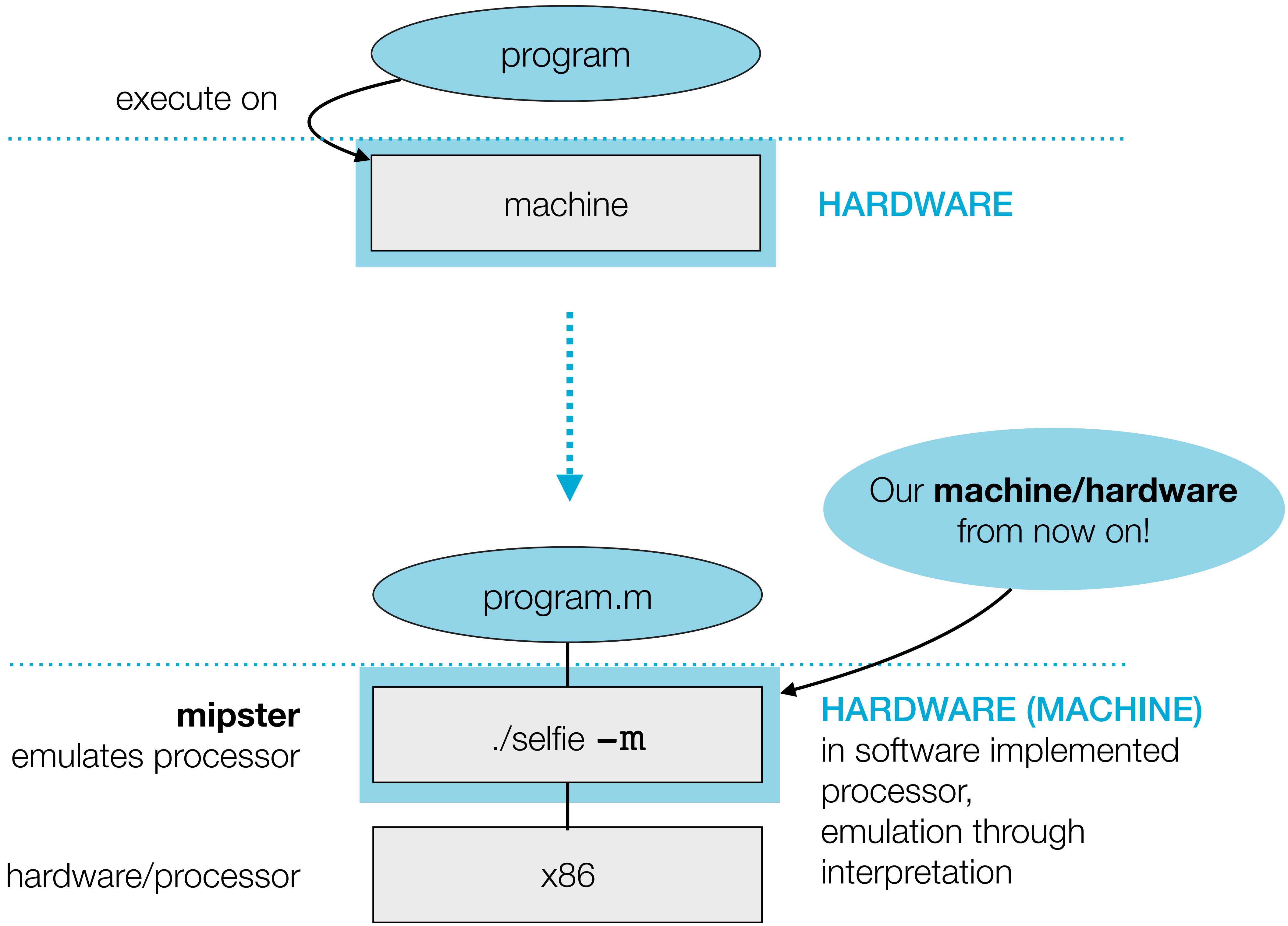
Renew **TIMESLICE** and switch back to context

```
timeout = TIMESLICE;

while (1) {

    from_context = mipster_switch(to_context, timeout);

    if (get_parent(from_context) != MY_CONTEXT) {

        …

    }

    else if (handle_exception(from_context) == EXIT)

        return get_exit_code(from_context);

    else {

        to_context = from_context;

        timeout = TIMESLICE;

    }

}
```

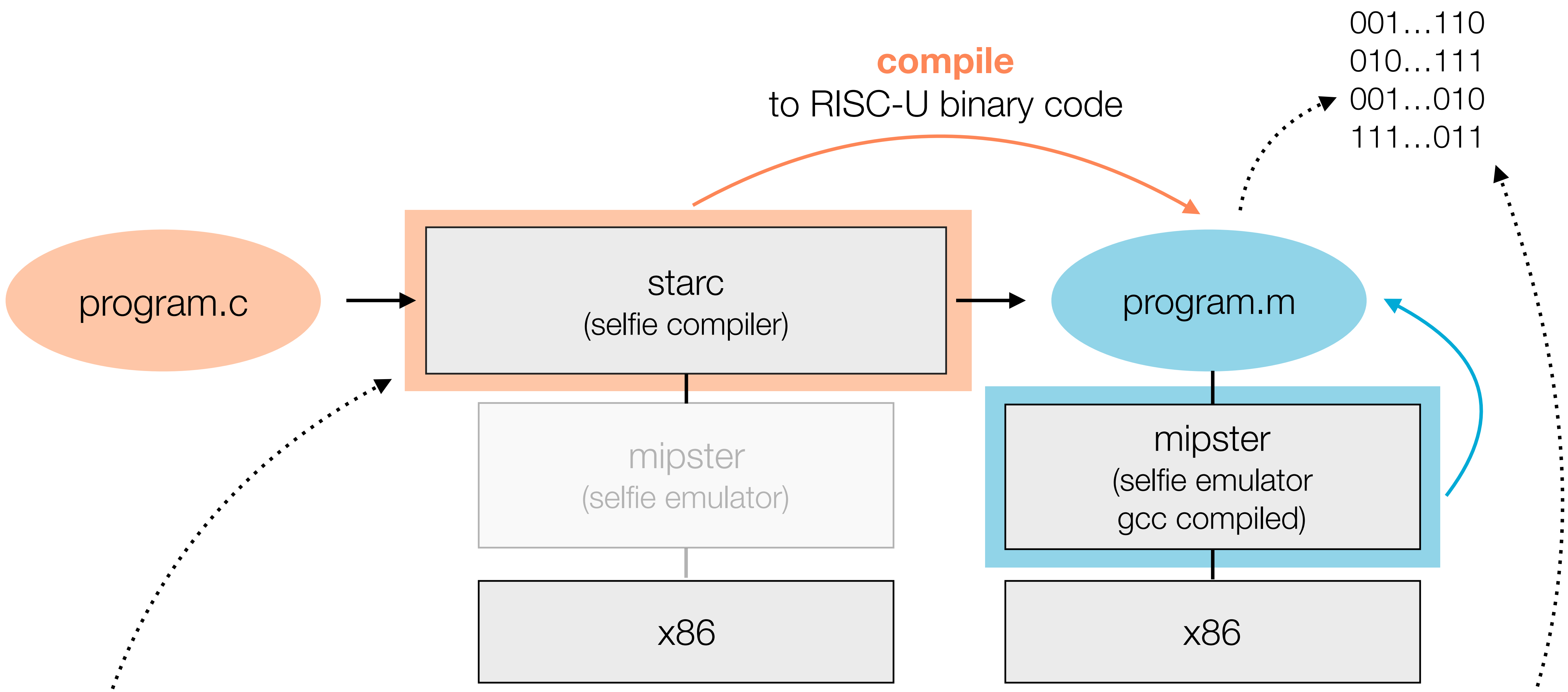
Summary Mipster

- RISC-U code can not be executed on an x86 processor.
- We use mipster, an emulator for a RISC-U processor in software to execute RISC-U code → create a **process**
- From now on we consider this first mipster (x86 version) to be our machine(hardware).

Summary Mipster



Summary Compilation and Interpretation



use compiler (starc) of
gcc compiled selfie
(execute compiler on x86 machine)

starc compiled selfie
(emulate compiler on mipster)

interpret
starc compiled
RISC-U code
(**fetch-decode-execute**
each instruction)

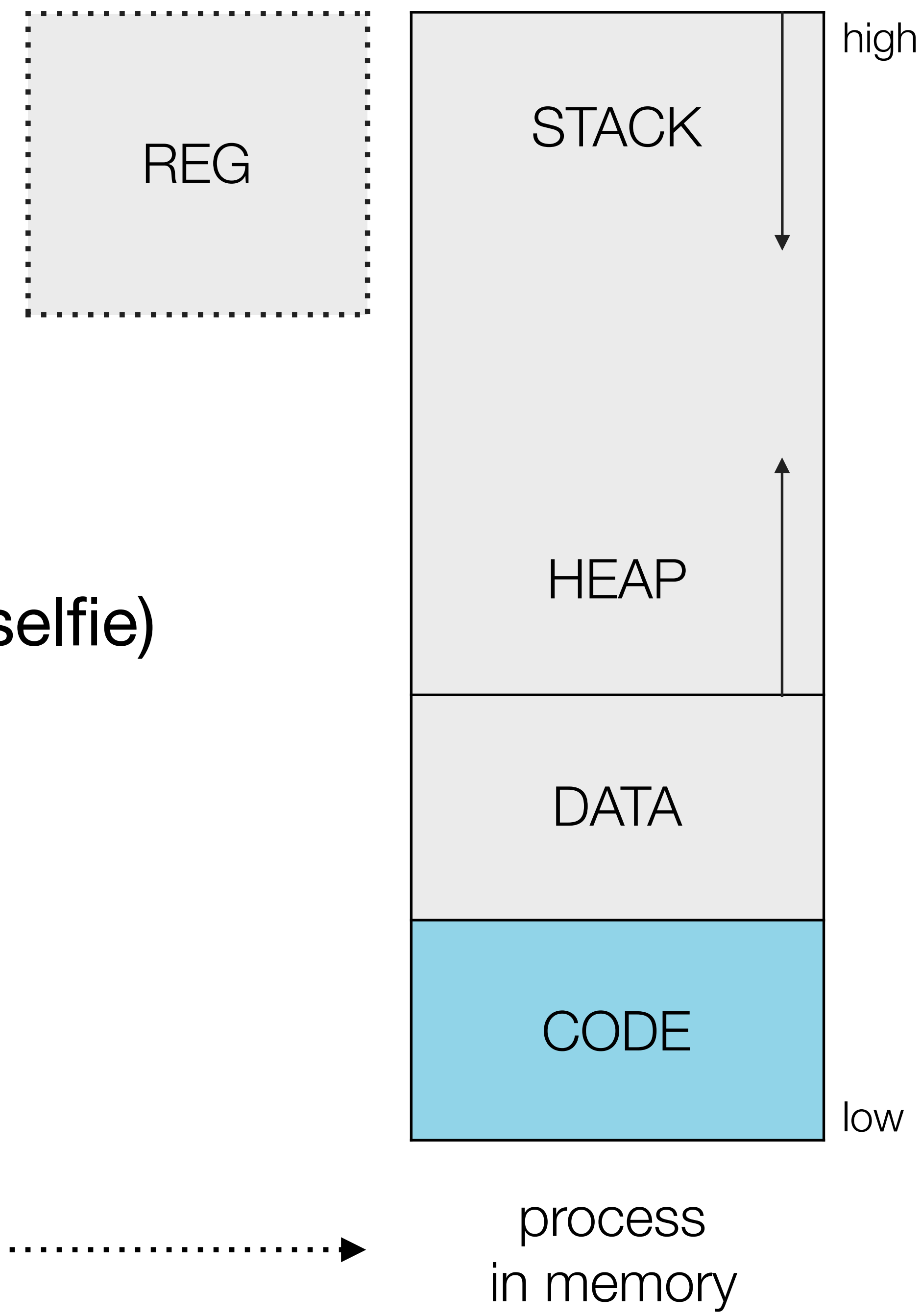
Process

*A <u>process</u> is a program **in execution**. More precise, it is an **abstraction** of a running program that is created by the OS.*

- **Program → passive**
  - executable binary
  - sequence of machine instructions somewhere on disk
  - “a program is executed”
- **Process → active**
  - execution of binary code, **pc** pointing to the next instruction to be executed
  - **code** in memory and a set of **resources** available to the process
  - processes are isolated, no communication between processes
  - “a process executes"

Process Looks familiar?

- **Resources owned by the process**
  - memory
  - processor state
    → content of registers, pc, …
- **Context is part of the process**
  - can be seen as snapshot of process (in selfie)



“Now we have enough insight into selfie and mipster to get started with operating systems.”

Operating System

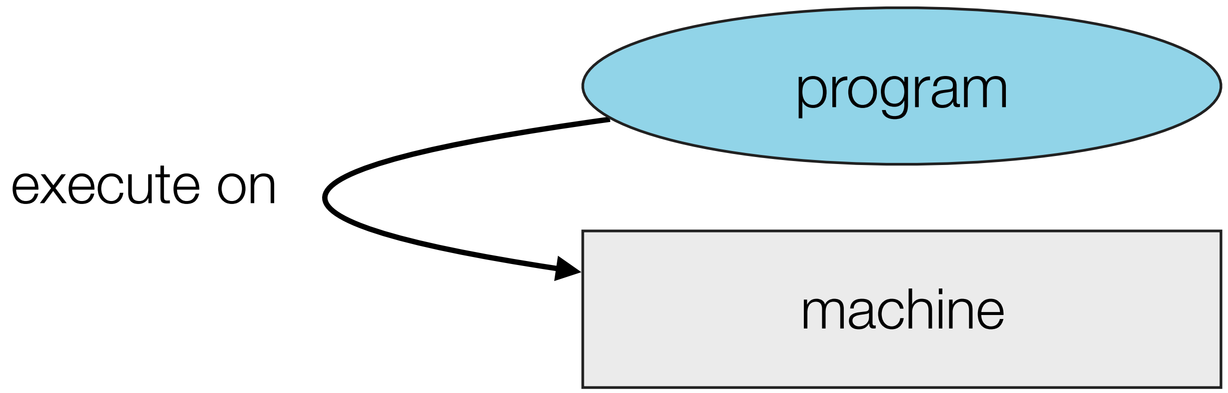
Concurrency,
Shared Resources,
Process Model,
Context

Operating System Introduction

- The machine provides **resources/hardware** for the execution of a program (CPU, memory, I/O devices).

- Execution of a program should be easy, stable, fast,…

- **Managing resources** is not a big issue if only a single <u>sequential</u> program would be executed at any time (as mipster does).
→ but we want to execute many, possibly <u>concurrent</u>, programs simultaneously



Concurrency and Operating System

- **Goal**
  - execute many programs seemingly at the same time
- **Problem** is limited resources
  - one physical CPU
    - machine can only execute 1 instruction per core
    - fixed number of cores, but different number of applications
  - one physical memory
    - fixed-sized memory
- **Our Goal**
  - learn understand how these problems are overcome

Concurrency and Operating System

- Where does the **need for concurrency** come from?
  - machine interacts with **real world**, where things happen in parallel
  - machine needs to reflect and deal with that parallel mindset of users
  - a concurrent programming model is intuitive

Concurrent or Parallel

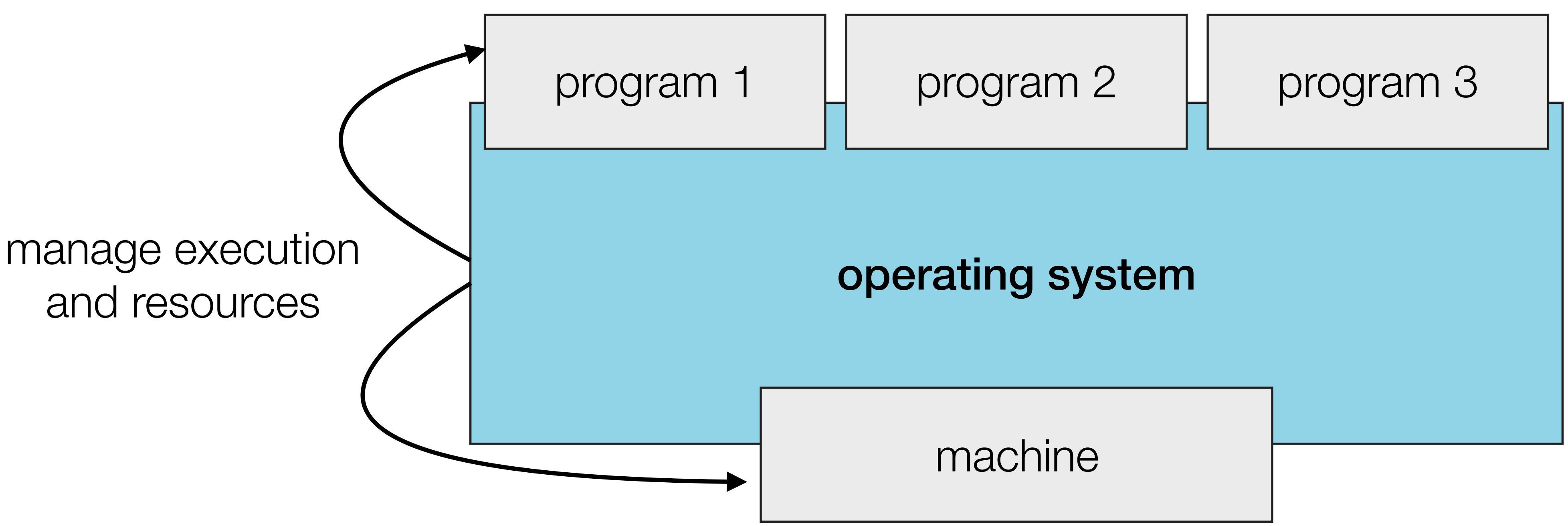
- **Concurrency - a property of software**
  - illusion of running many processes at a time
  - can be achieved by:
    - time-sharing (single core)
    - execute in parallel (multi-core)
- **Parallel - a property of hardware**
  - truly in parallel, simultaneously - hardware support essential
  - divide workload to increase performance
  - program itself does not have to be concurrent

Operating System Introduction

- As we know, we can execute many programs on a single machine at the same time, all of which get their fair share in resources → **thanks to the OS**

- The OS acts as an intermediary between processes and machine,
  - it **manages** resources.
  - it **manages** and **controls** the execution → protection, error management,… .

- The OS consists of several <u>components</u> and we are primarily talking about the OS <u>kernel</u>.



Operating System

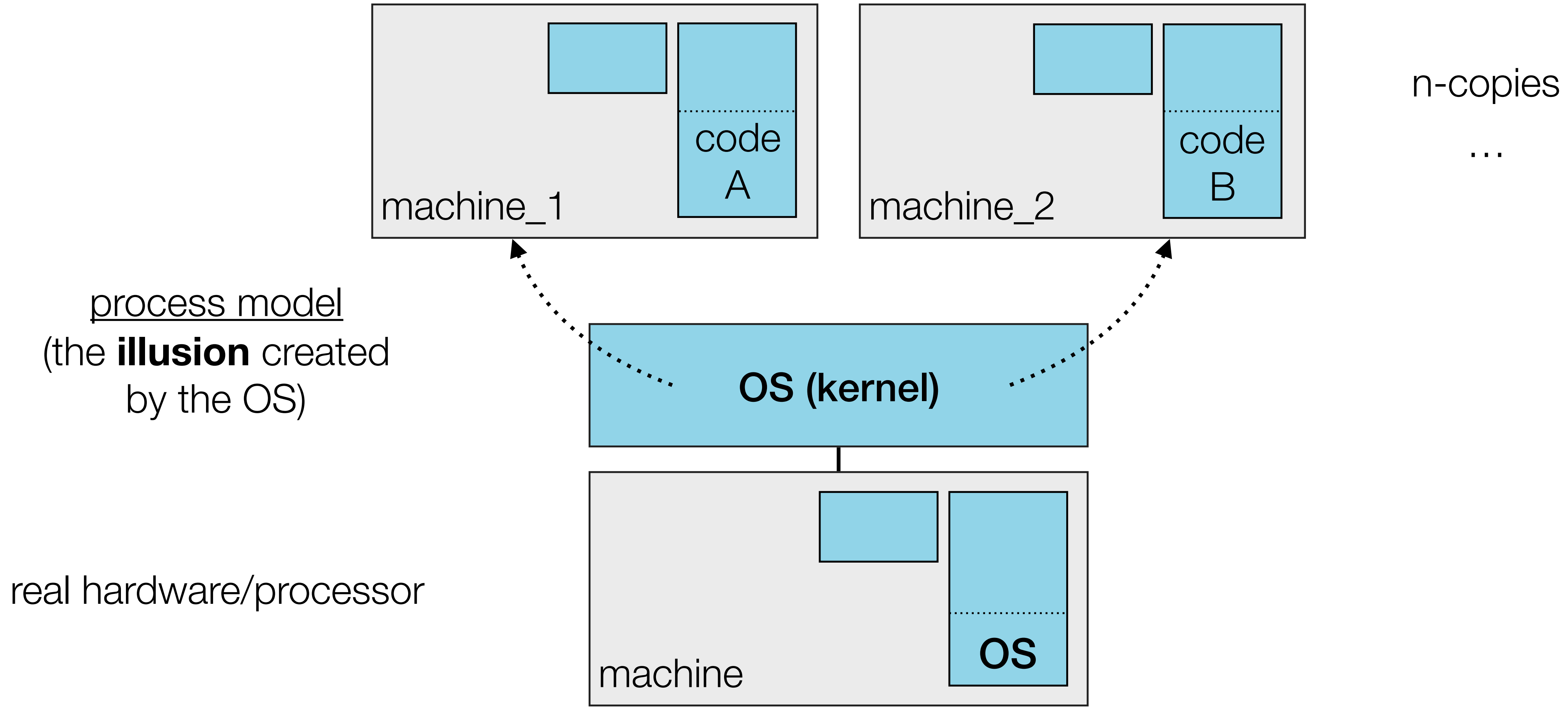
An operating system manages resources and controls the ***execution of many processes*** *on a machine.*

- Selfie (mipster) can not execute more than one program at a time. However it already provides some basic operating system functionality we will built upon.
- Step by step we will extend selfie to enable concurrent execution of programs.
- **First step:**
  - Selfie can execute two copies of the same binary concurrently.
    - we will solve the problem of **one physical CPU**
  - To figure out how to implement this feature in selfie, we look at how an OS achieves this goal.

Concurrency in an OS

- The OS kernel creates `n` instances of the machine it runs on to enable concurrent execution of `n` processes
  - hardware virtualization (CPU and memory)

- Process is not aware that it runs in such a container .



Process Model

*A process model describes the **illusion** that the operating system creates.*

- Several process models, which differ
  - in how close the illusion created by the OS is to the real machine
  - in the level of temporal and spatial isolation they provide

1. **system virtualization** → an exact copy that is absolutely indistinguishable from the machine
2. **UNIX process** → a subset of the machine
3. **threads** → an even smaller subset of the machine

1 CPU

“How could the OS actually execute two processes on a single CPU concurrently?”

“It could behave like mipster, it could interpret code.

The OS could interpret codeA for a while, then stop and interpret codeB for a while, then stop and interpret codeA for a while, then stop and…”

Concurrency in an OS

- **Interrupt and continue execution:**
    - It is necessary to **save enough of the machines state/process** at the point it is interrupted…
    - … so it can be **restored** when execution is continued some time later.

- **The actual purpose of a context:**
    - the minimal set of data saved so execution can be interrupted and continued

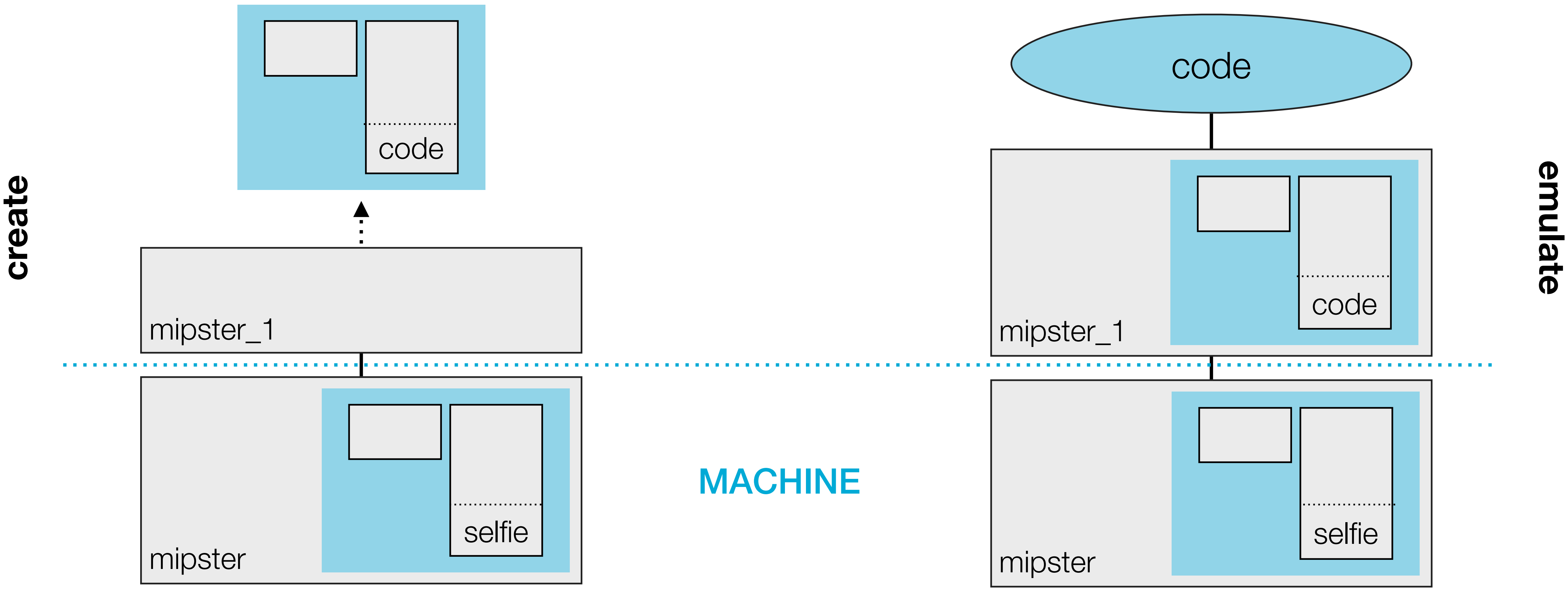
"The OS could interpret codeA for a while, then stop, save codeA's context and interpret codeB for a while, then stop, save codeB's context, restore codeA's context and interpret codeA for a while, then stop, save…"

Concurrency in Selfie

- The machine our OS will run on is the machine instance created by mipster.

  → this means mipster can create an **instance** of that machine

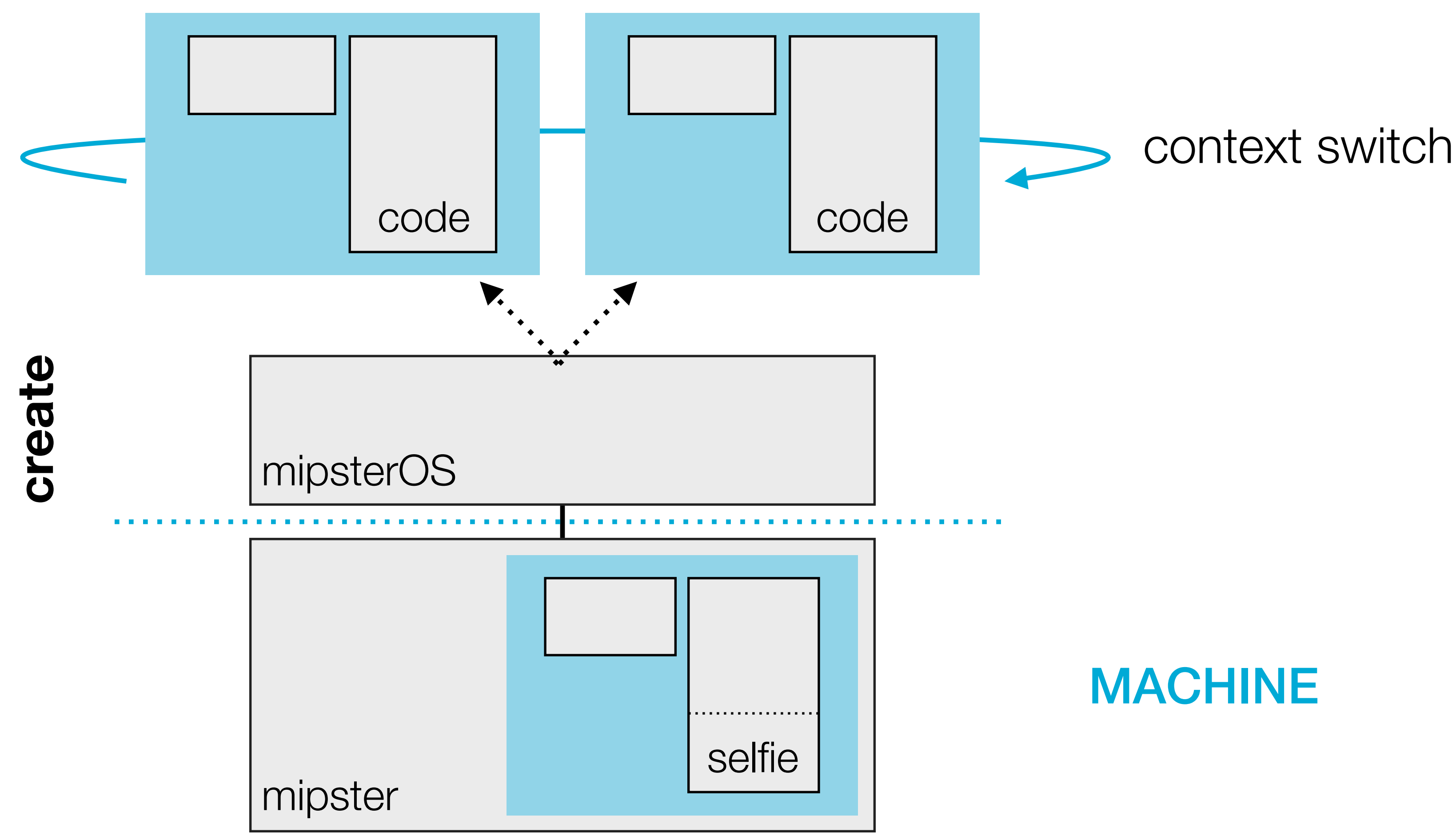
- **Consider this situation:**
  - Let our machine (mipster) execute selfie and start mipster, say mipster_1
  - mipster_1 creates and emulates **one instance** of the machine it is run on!
    - we want create and emulate **two instances** of that machine → two contexts



MipsterOS

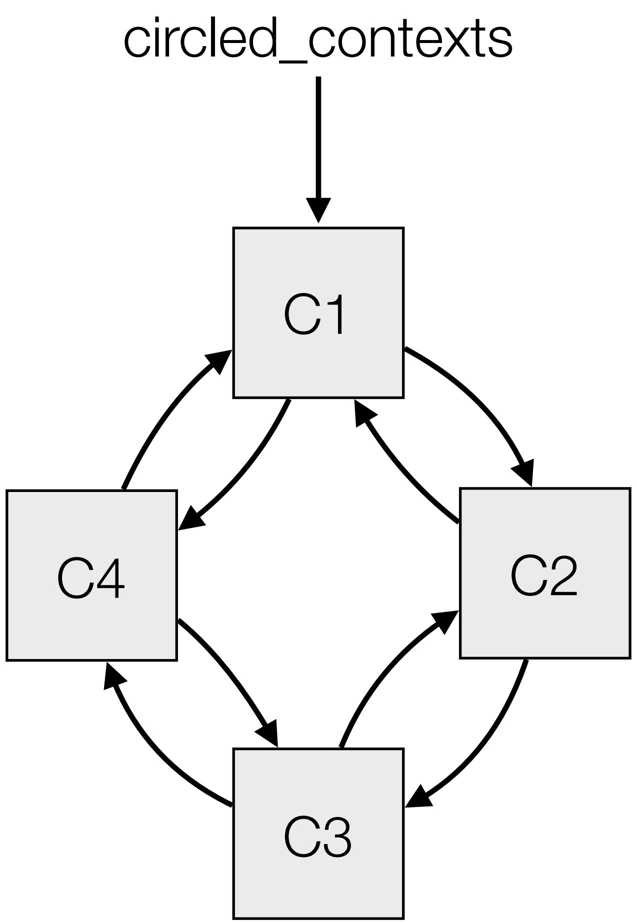
- First approach towards building an OS will be a **system similar to mipster.**
  - implement `-x` option that creates a <u>multi-tasking</u> system (mipsterOS) that runs two copies of the same binary → two processes

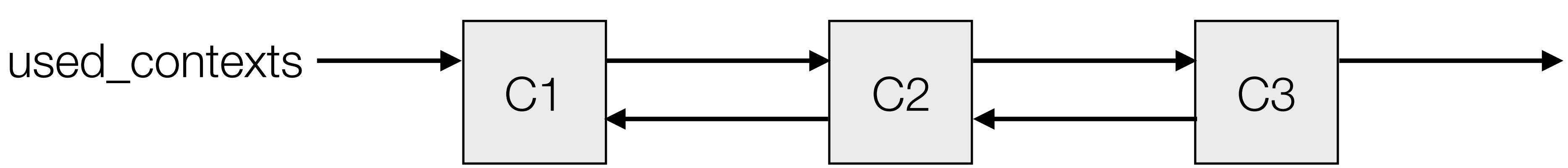
- Instead of creating and emulating one context, this system **creates and emulates 2** contexts concurrently by switching between them **after every instruction**.



Context

- You can make yourself familiar with the context structure and procedures for managing multiple contexts in selfie
  - creating and allocating a new context
  - searching for a context
  - deleting and freeing a context
  - saving, restoring and caching a context (later)
- Selfie maintains two **lists** to manage contexts
  - `used_contexts` → doubly-linked list of used contexts,
  - `free_contexts` → singly-linked of free contexts
- Lists are just one possible way to organize/structure contexts
  - any structure that can be built from parent, previous and next is possible





Assignment 1 MipsterOS `-x`

- Students can implement a simple **multi-tasking system**, mipsterOS, that can emulate two copies of the same binary concurrently (as described in the previous slides).
- Hint: similar to mipster
  - study how mipster sets up a context
  - look at the next two slides for **tips**

MipsterOS Implementation Tips

- Recognize `-x` option as argument and set up mipsterOS (similar to mipster)

- **Copy or modify** `selfie_run(…)`
  - implement `selfie_run_mipsterOS(…)` that creates two contexts using `selfie_run(…)` as blueprint
  - modify the existing `selfie_run(…)`, such that a second context is created when mipsterOS is run

- **Individual or linked** contexts
  - no linking, just passing both contexts as arguments to the mipsterOS procedure
  - link the contexts (each being the others next) and pass the first context is mipsterOS

- **Extra challenges**
  - make sure that both contexts finish execution with an exit call
  - enable loading two or more different binaries and execute them concurrently

MipsterOS Testing

`./selfie -c`

- Have mipsterOS execute <u>`hello-world.c`</u> → a program that prints "Hello World!" to the console

- The correct, yet not so nice looking output should be

```
> Hello Wo Hello World!    rld!
```

- **But why?** - look for an answer in `hello-world.c`
  - `write(…)` 8 bytes at a time
  - switch after one instruction

“Wait…It is rather unlikely that `write()` corresponds to a single machine instruction. Why is the output not 'HHeelloo…' ”

“This is another example of the OS support already provided by mipster and therefore also mipsterOS. Let us look for the answer in the code.

Try to find the definition of the write procedure in selfie”

Write in Selfie

`./selfie -c`

- There is no procedure definition for write in the traditional sense
  - only `emit_write`, `implement_write` and a declaration of `write`
- A closer look at `emit_write` shows that a library table entry for the `write` procedure is created, followed by its implementation (machine instructions - we come back to this soon)
  - part of that implementation is `SYSCALL_WRITE`
- A syscall causes mipsterOS to stop interpreting code and return from `mipster_switch` (as described when mipster was introduced)
- A syscall is handled **by mipsterOS** → `handle_exception`, `handle_system_call`
  - mipsterOS implements the syscall, **not the process**

Write in Selfie

- The process does not write to the console directly
- Instead, the process uses a mechanism to have mipsterOS execute write on its behalf
- An OS uses this mechanism to solve a problem that we created because we wanted to execute programs concurrently

The Problem

*We created two or more processes that are **unaware***

- *that they run in a container*
- *of each other*

*and that **use machine resources**.*

- Processes that write to the console or open a file at the same time would cause undefined behavior.
- Therefore, processes can not be allowed to perform certain operations. Instead they ask the OS to perform these critical operations for them.
- **Solution** - the OS controls execution and manage resources
    - the OS provides services
    - the process may request those services via **system calls**
    - process and OS run in different **modes**

System Calls

Modes and Protection,
API and ABI,
Wrapper Function and Trap,
Bootstrapping

Modes Protection

- <u>Modes</u> with different privileges provide a means of **protection** and **isolation**
- CPU has a **bit** (or more) that is set to its current privilege level (mode). Code executed on this CPU is running in this mode.
- An OS is run in **kernel mode**, it has all the privileges → trusted
    - unrestricted access to hardware and memory
    - allowed to execute any instruction
- A program is run in **user mode**, it has the least privileges → untrusted
    - not allowed to access hardware and memory directly
    - not allowed to execute certain instructions
    - not allowed to access OS code

Modes Protection

- If a program executes an instruction it is not allowed to (program's privileges level too low), an exception is raised.
- This exception prompts the processor to switch into kernel-mode and execute the OS code that handles the exception.
- Selfie **does not implement** actual mode switching as described.
    - nothing that indicates the current mode

System Call

*System calls are **requests** made by processes **for services** provided by the operating system.*

- Operating system performs the requested services and returns control to the program
- A switch from user mode into kernel mode takes place

System Call

- The OS creates an **abstraction** of the services by providing an **API** that specifies available functions and their parameters and return values.
    - no direct access to actual system call
    - portability and ease of use - same API across different systems
- The API is not accessed directly but via a library provided by the OS
    - "not directly": the program does not jump to the function within the OS directly
    - library is the interface to system calls
    - library provides wrapper functions for calls to API
    - wrapper functions conform to **ABI**

```c
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

UNIX write system call (API)

API and ABI

- **Application Programming Interface -** API
  - communicate between two pieces of software on source code level
    - relatively hardware-independent
    - exposed parts of software that can be accessed from outside
  - abstraction of underlying implementation → provide building blocks to **programmer**
  - API is specification (behavior), library its implementation
  - for OS: API describes interface between application and OS (ex. POSIX)
- **Application Binary Interface -** ABI
  - communicate between two binary programs
    - hardware-dependent
  - conforming to ABI is mostly done by **compiler, the OS, a library author**,…
  - describes calling convention (passing parameters), syscall interface,…

System Call in Selfie

- Selfie already supports 5 system calls (API)
  - `exit`, `read`, `write`, `open` and `malloc`
  - mipsterOS has a implementation/definition for each of them
- A program that calls `write` does **not jump** to the implementation of `write` in mipsterOS.
  - it is not allowed to do so → potential for malicious behavior
- Instead selfie provides an interface as part of the library `libc*`
  - contains **wrapper functions** for all system calls (ABI)
  - program jumps to these wrapper functions when executing a system call

System Call in Selfie

using the API via selfies library libc*

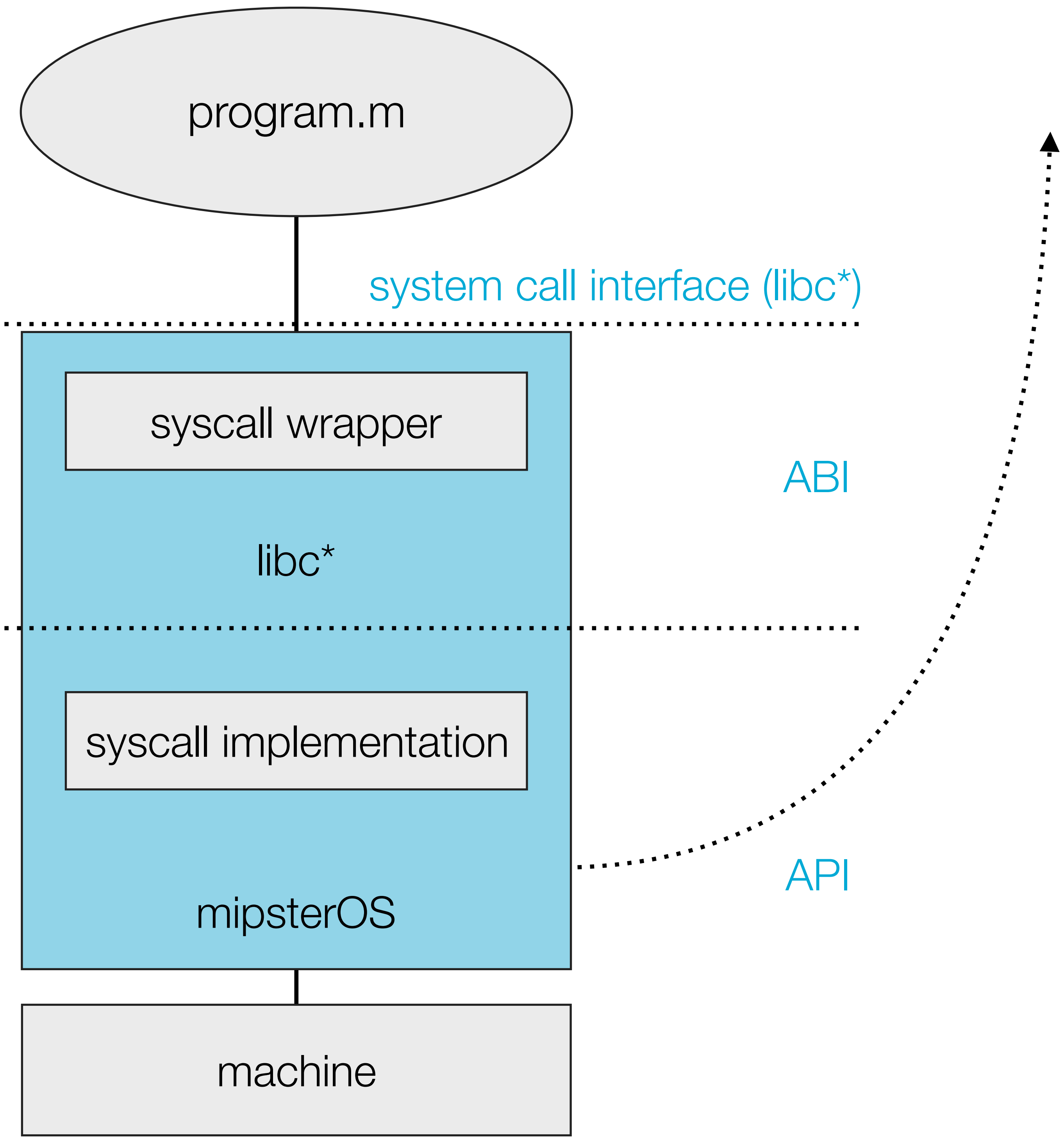
`write(…)`

libc* contains wrapper functions for all syscalls

`write(…)`

mipsterOS implement the syscall

`implement_write(…)`



Wrapper Function

`./selfie -c`

> *A wrapper function "wraps" the call to another function and usually performs little additional computation. They are often used to* ***hide details*** *and create an extra layer of* ***abstraction****.*

- Wrapper functions are put into the binary at compile time by selfie → `emit_write`
- **Prepare** actual syscall - conform to syscall ABI
  - calling convention
    - copy arguments from stack into registers in which mipsterOS expects them to be
    - put unique syscall number into register
- **Generate a** trap, a software generated interrupt to **transfer control** from process to OS
- Interface for syscalls and provide **protection** to the OS kernel

Trapping Mechanism in Selfie

`./selfie -c`

- Syscalls and errors throw an exception → `throw_exception`
  - set the exception code
  - set the **trap** (flag)
- The trap **signals** the emulator to stop interpreting code and handle the exception
  - transfer control from program to emulator (switch from user to kernel mode)
- After handling the trap, control is **returned** to the process

“So wrapper functions are put into the binary at compile time by selfie… . What about the gcc compiled version of selfie?”

“That is one missing piece we haven't talked about yet.

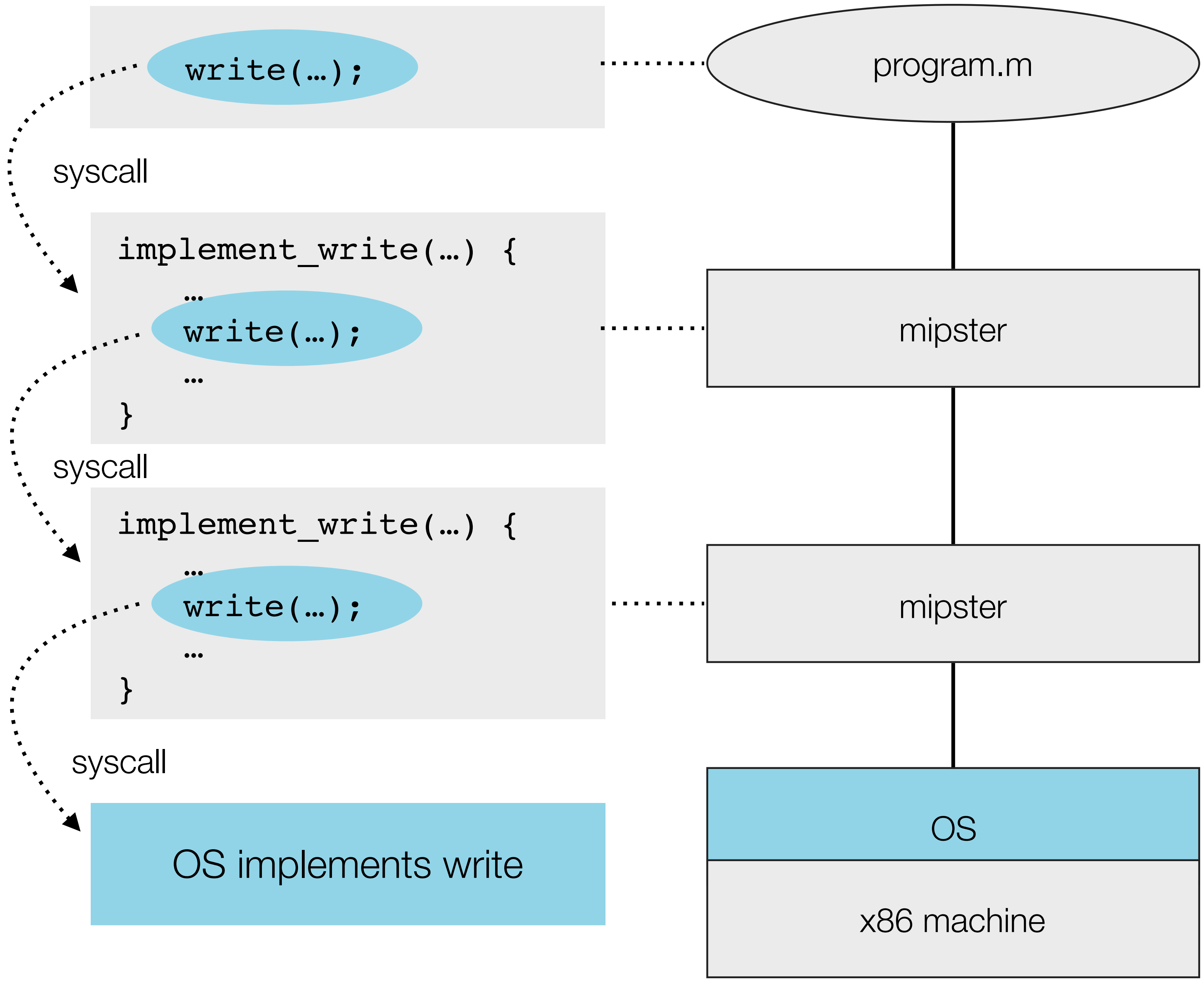
The other, as you might have noticed, is that the actual implementation of a syscall in mipster/mipsterOS contains that same syscall. (implement_write contains write)”.

Bootstrapping Syscalls on Bootlevel 0

- syscall wrapper - the same principle applies
  - the compiler (gcc) puts the wrapper code into the selfie binary
- How?
  - the compiler sees **undefined procedures**
    (the declaration as mentioned before)
  - therefore, the compiler provides the implementation
- Therefore, syscalls on bootlevel 0 are **handled by the actual OS** running on your machine.

Bootstrapping Syscalls on Bootlevel 0

- Notice how syscalls are passed down?
- Therefore, **all syscalls** reach bootlevel 0 and are **handled by the actual OS** running on your machine.
- The same way syscalls are passed down, return values are passed up to the program.



Summary OS Introduction

- Being able to run several processes at the same time is a huge gain.
- Unfortunately, there are **problems** that have to be solved to enable concurrent execution.
- The operating system **provides solutions** to these problems.

Summary OS Introduction

- We addressed the first **problem**
  - several processes using the same CPU (and console)
- The OS **solves** this problem by managing CPU and controlling execution
  - CPU time is shared among processes → time-sharing
  - OS provides services to the process → system call
- Concepts and mechanisms used
  - **process** → program in execution
  - **abstraction/illusion** → creating machine instances (containers), processes, system call API (hide hardware details), wrapper functions (conform to ABI)
  - **different modes** → kernel mode for OS and user mode for processes
  - **trap** → software interrupt to switch between mods

MipsterOS Testing Revisited

- Remember this “problem”?
  - `Hello Wo Hello World!    rld!`
- Writing has to be <u>synchronized</u> (coordinated)
- BUT processes are **unaware** of each other and they have **no way of communicating** directly with each other
  - the process can not be responsible for checking if it is safe to write
  - OS support is needed
- We can solve this problem with help of the mipsterOS by implementing a mechanism called **locking** (system call!)

A Write Lock Intended Semantics

a variable indicating
lock owner

lock call
saves caller as
lock owner
(acquire lock)

unlock call
removes caller as
lock owner
(release lock)

```
uint64_t LOCK;
…

lock();
    while(*foo != 0) {
        write(1, foo, 8);
        foo = foo + 1;
    }
unlock();
```

iff the lock is held
by a process,
only the lock owner
is allowed to **write**
to the console

Assignment 2 System Calls

- Students can implement the `lock()` and `unlock()` **system calls** within the mipsterOS.
- Hint: look at the next slide for **tips**

A Write Lock Implementation Tips

- `uint64_t LOCK` is a global variable **within the mipsterOS**
- A process is not allowed to set this variable itself
- The OS has to provide this as a **service**
  - a lock and unlock system call → API: `lock(), unlock()`, ABI: syscall number
  - a libc* wrapper function → syscall interface (ABI)
  - handling the syscalls → implementation in mipsterOS
- **Possible pitfalls**
  - unlike the other syscalls, lock is not 'passed down' (mipsterOS implements/emulates it)
  - a lock can only be acquired when it is not held (syscall successful)
    - otherwise the process has to **wait** (syscall failed)
    - the OS sets the program counter back so the process makes the syscall again later
  - only the lock owner can unlock

Emulation vs. Virtualization

Operating System

*An operating system **solves problems** that arise from concurrent execution of processes. It manages resources and **controls the execution** of many programs on a machine and **abstracts** that machine.*

- Already a good definition
- BUT mipsterOS does not control execution like a 'real' OS kernel
  - controlling execution by interpretation is **highly insufficient** (exponential in #emulators)
  - more efficient if the program would run **directly on the machine**

Interpretation is Slow

1 instruction
(ADD,…)

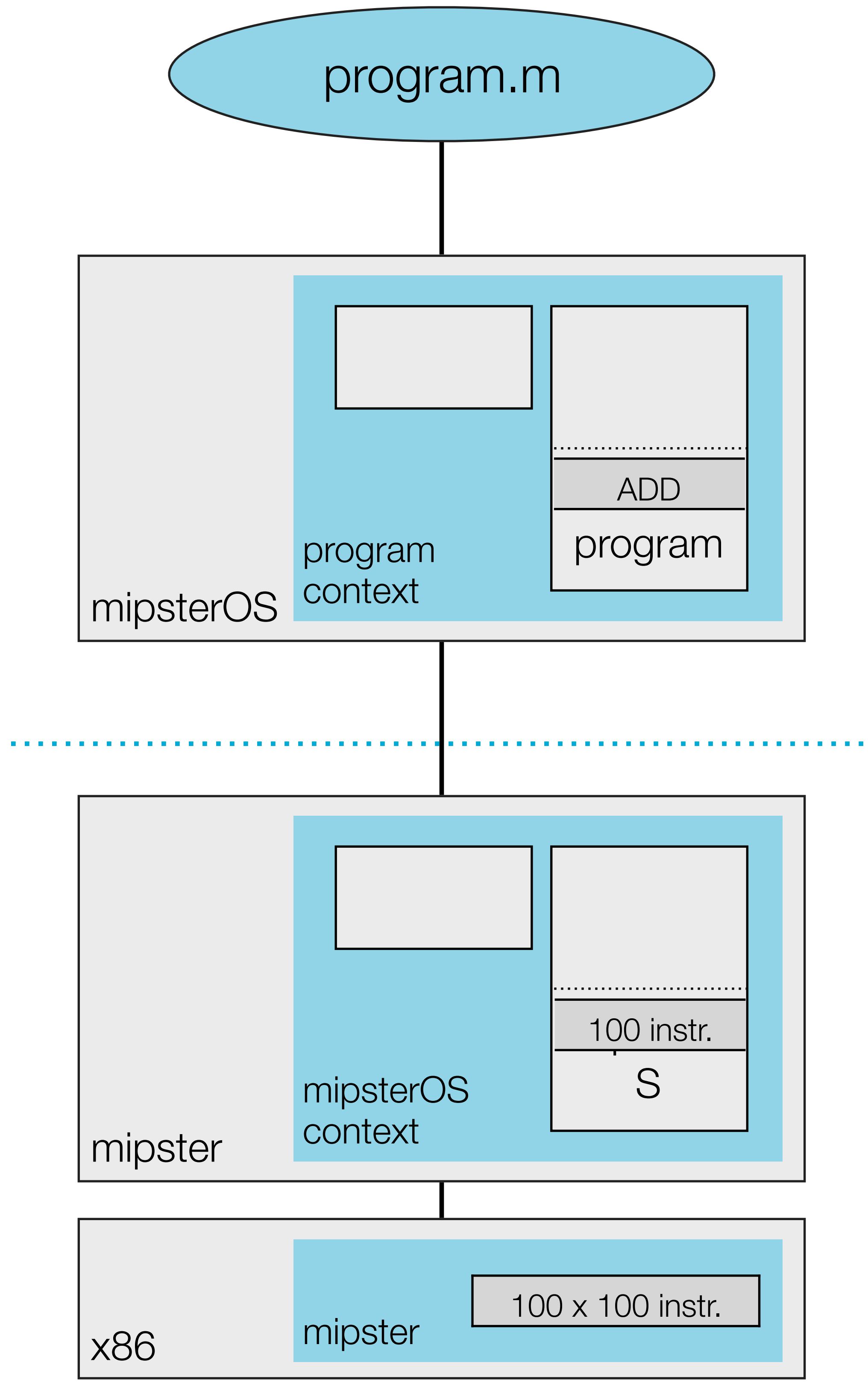
x100

100 instructions* in mipsterOS
to emulate
1 instruction of program.m

x100

100 instructions* in mipster
to emulate
1 instruction of mipsterOS

10.000 instructions* to emulate
program on top of OS



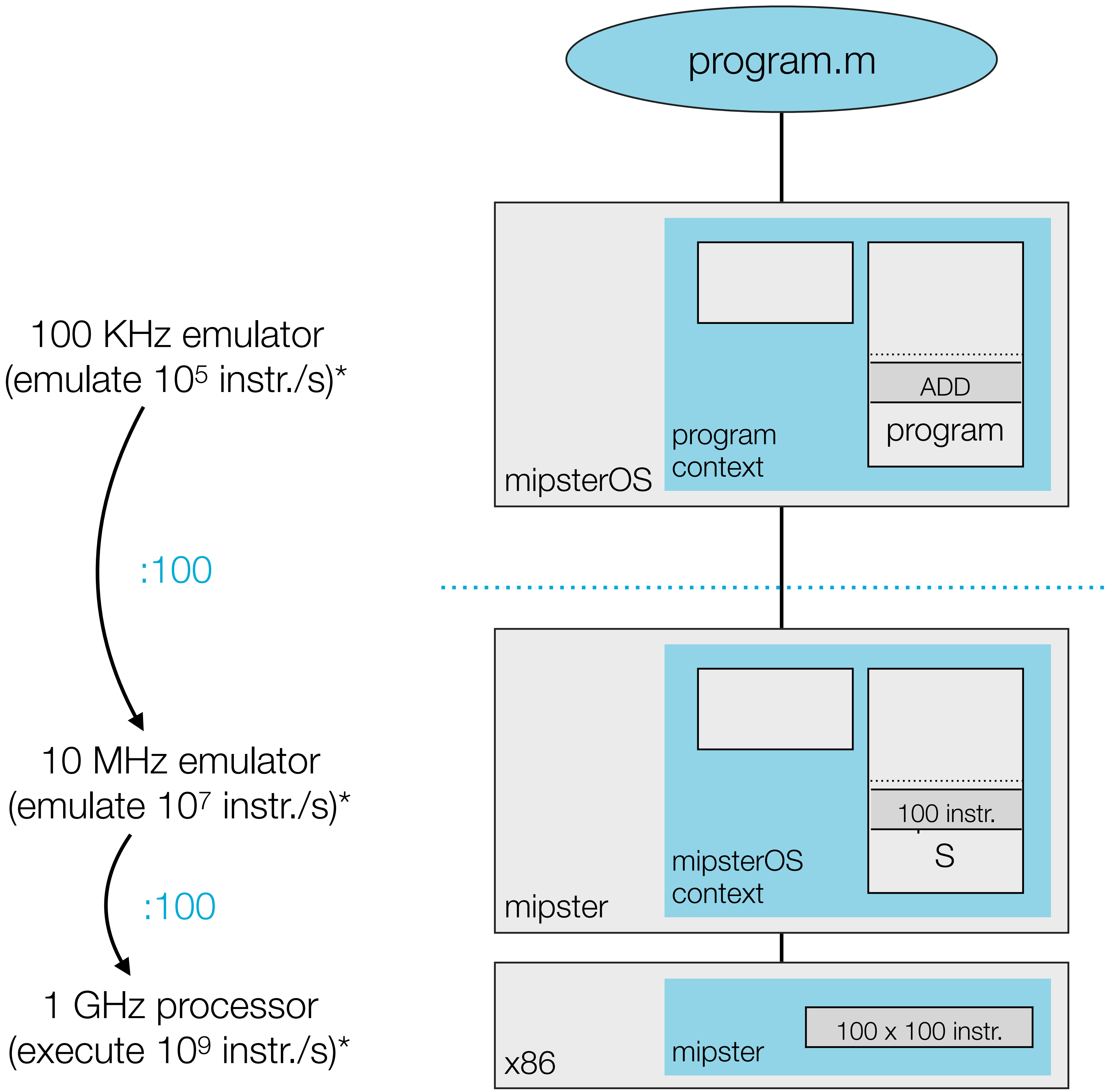
USER PROCESS

OS
emulation through
interpretation

MACHINE
in software implemented
processor,
emulation through
interpretation

*assumption: 100 instructions to emulate 1

Interpretation is Slow



USER PROCESS

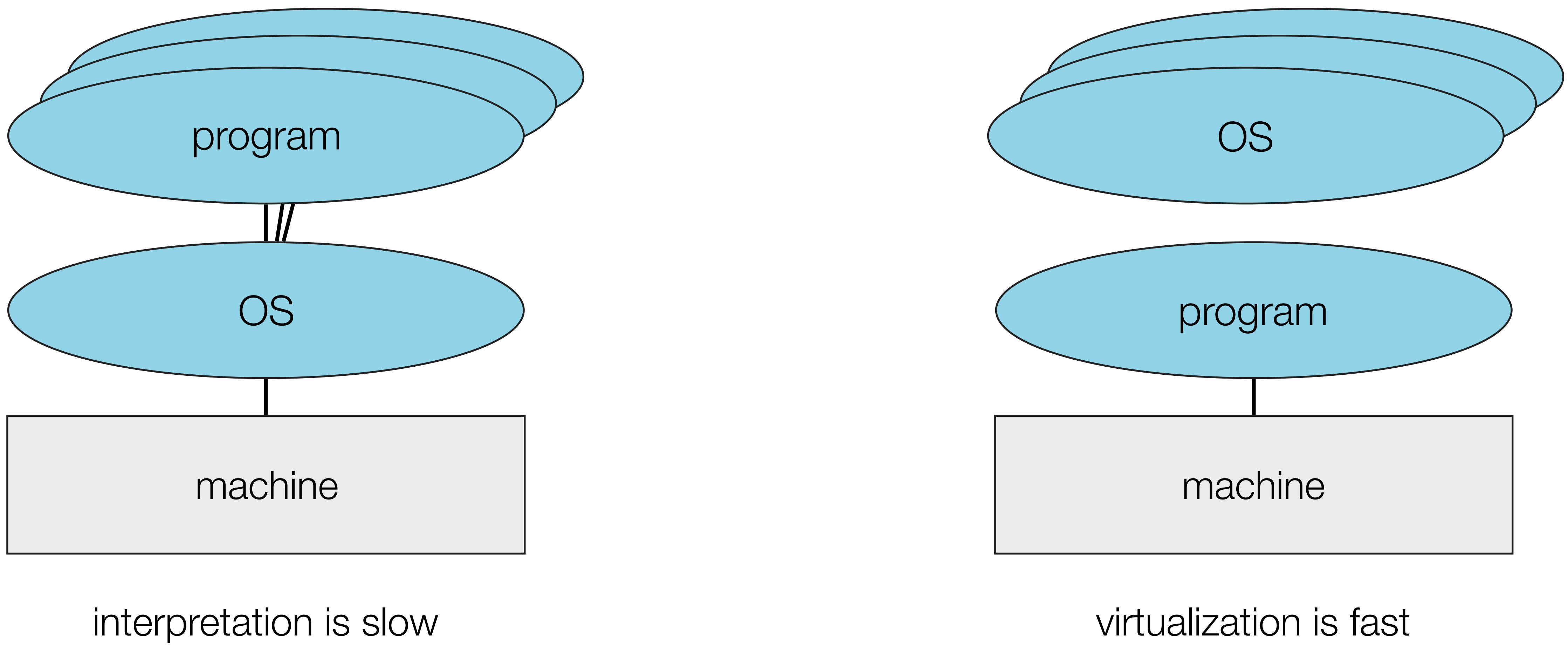
OS
emulation through interpretation

MACHINE
in software implemented processor,
emulation through interpretation

*assumption: 100 instructions to emulate 1, 1 instructions per cycle

Interpretation is Slow Solution

- An OS uses a mechanism to allow user processes to execute directly on the machine the OS itself runs on
  - the OS <u>virtualizes</u> the CPU, it 'asks' the machine(CPU) to execute the program on its behalf
- To achieve this, the OS has to give up the machine - **give up control?**

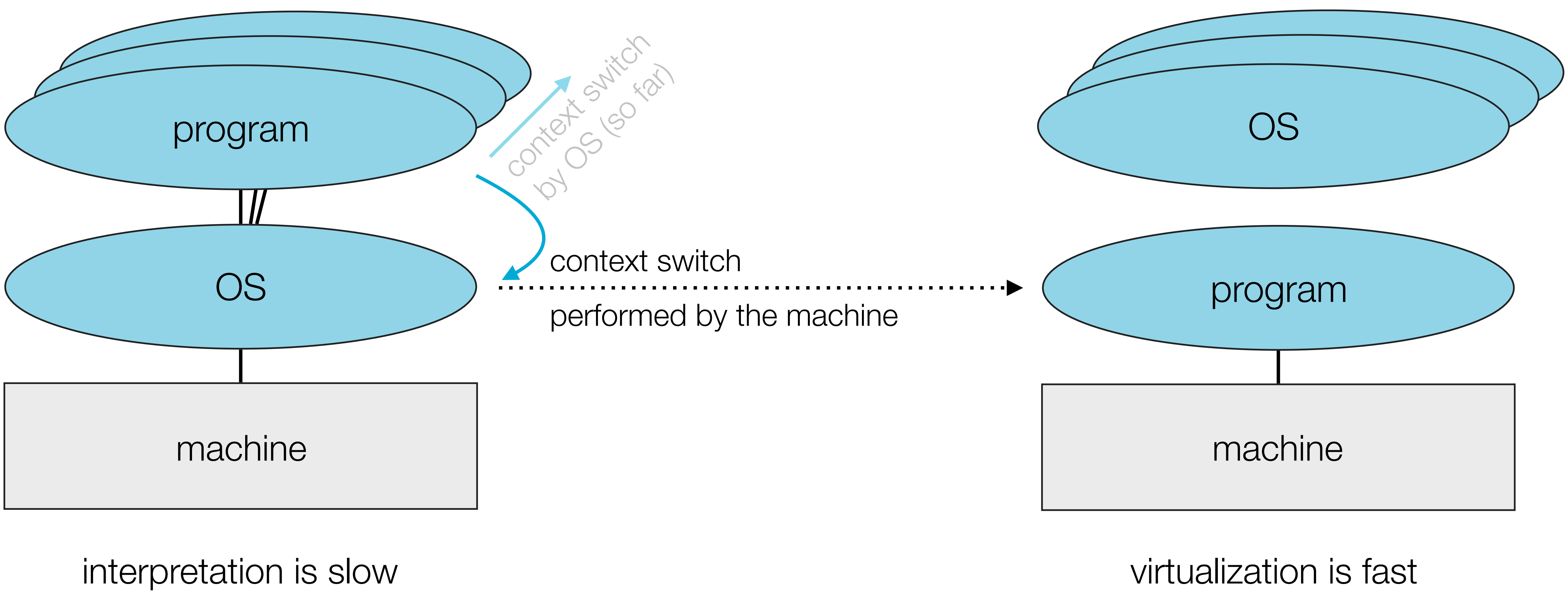


CPU Virtualization

- With this concept some questions arise
  - **control**
    - How does the OS give up the machine (and stay in control)?
  - **system calls**
    - Are they different now, how does it work?
  - **switching between processes**
    - Is this different now, how does it work?
- What we discussed so far still holds. We only left out some details.
- Whenever the OS is needed it regains control of the machine
  - with support of the machine

CPU Virtualization Pass-Over Control

- The mechanism used to virtualize the CPU is a **context switch**
  - initiated by OS
  - for actual switch **hardware support** necessary



CPU Virtualization Regain Control

- Two methods possible
  - **cooperative**
    - process gives up CPU voluntarily via syscall (cooperative)
    - syscall → switch to OS
  - **preemptive**
    - process is forced to give up CPU
    - before giving up the CPU, the OS sets a timer
    - timer causes interrupt → switch to OS

CPU Virtualization Syscall

- **Emulation → emulator(OS) interprets** syscall instruction of program
  - trap into OS
  - trap signals the OS to stop interpreting and handle the syscall
- **Virtualization → machine executes** syscall instruction of program
  - trap into machine
  - trap causes the machine to perform a context switch to the OS
    - machine cannot handle syscalls
  - the OS then handles the syscall
    (and afterwards switches back to the process)

CPU Virtualization Switching

- Whenever the OS regains control it decides which process to execute next (performing context switch)
  - Which next? → we talk about process management soon
  - this switch is performed by the OS (software)

“I understand the general idea but implementing this seems rather complicated and like a lot of work…”

“Yes…getting this to work requires quite a lot of thinking. There are many details that have to be done right.

Fortunately, selfie already implements this mechanism. It is exactly how selfies hypervisor hypster executes a single program.

We can study this mechanism without having to implement it.”

Hypster

Hosting,
Context Switch

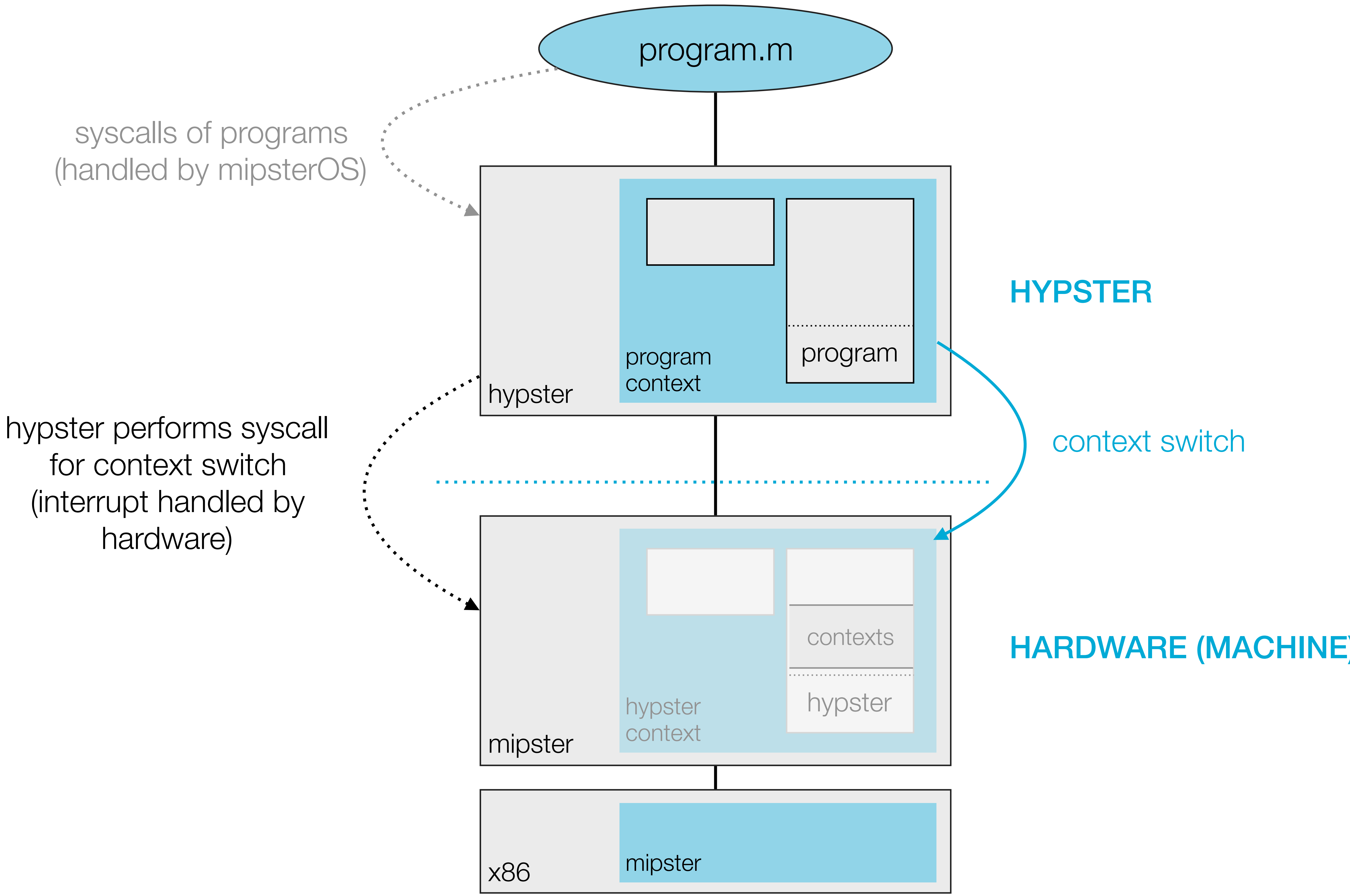
Hypster Hypervisor

- A hypervisor or virtual machine monitor (VMM) creates a virtual machine
- **Hosts** code
  - does not execute code itself
  - uses the interpreter it runs on (has to run on a mipster)
- Comparing hypster and mipster
  - `hypster_switch` instead of `mipster_switch`
  - no get_parent(from_context) != MY_CONTEXT
    (soon you will understand why)

Context Switch Hypster vs Mipster

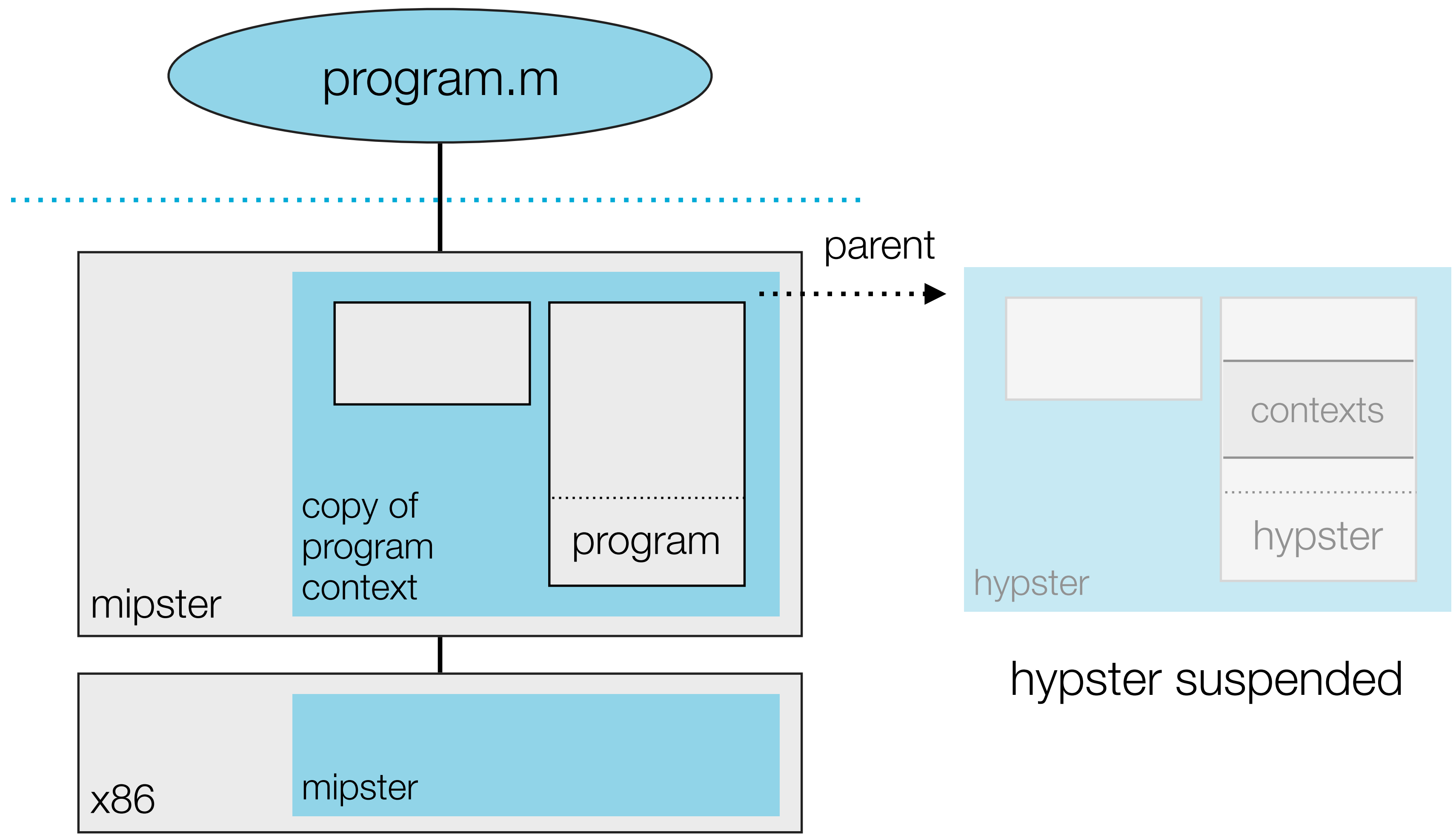
- **Mipster switch** - mipster and mipsterOS
  - context switch performed by mipster/mipsterOS
  - switching implemented in software
- **Hypster switch** - hypster
  - syscall by hypster
  - context switch performed by the machine hypster is running on
  - switching implemented in hardware

Context Switch Hypster → Program



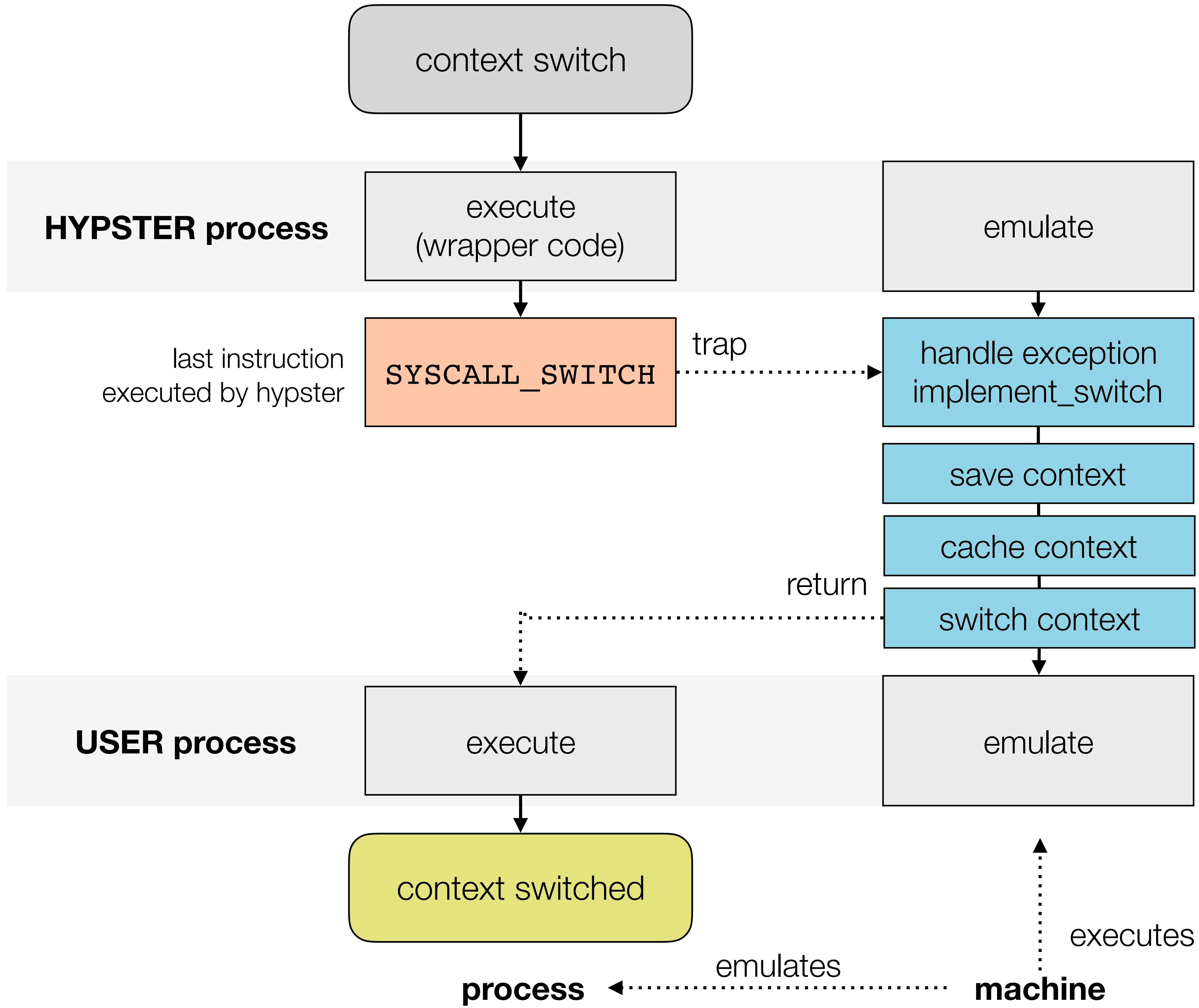
Context Switch Hypster → Program

program running
on hardware natively



hypster suspended

CPU Virtualization Pass-Over Control in Selfie



CPU Virtualization Pass-Over Control in Selfie

- We have mipster emulating hypster and hypster hosting a program

- **Hypster** runs on machine
  - hypster_switch jumps to the code of **emit_switch**(syscall wrapper)
    - because we search for definition in library table first (compile time)
    - no definition in library table on bootlevel 0 → search global table (fall back to mipster_switch)

- **Machine** implementing/handling the `SYSCALL_SWITCH`
  - saves the current context (hypster)
  - switches to the **cached context** (program)
  - continues with emulation (now program)

Cached Context Abstraction

./selfie -c

- Cached context is simply a **deep copy** of a context that the **machine** creates
  - hypster manages the original context
  - the machine uses the copy for execution
- Only cached contexts are used for direct execution on machine
- Original and copy are synchronized before and after a context switch
  - machine updates the **copy before** the switch → `restore_context`
    - hypster modified context (ex. by handling a syscall for the process)
  - machine updates the **original after** the switch → `save_context`
    - machine executed and thereby modified context

Cached Context Parent

- the parent of a context is the machine the context was created for
  - the **original contexts parent** is the hypster context (`MY_CONTEXT(0)` to hypster)
  - the cached contexts parent is also the hypster context (**cannot be** `MY_CONTEXT(0)` to the machine)
- So how does the machine know **who the cached contexts parent is**?
  - the parent initiates the switch to execute its context
  - machine knows which context executed `syscall_switch`
    - the context it was executing (`current_context`)

CPU Virtualization Syscall in Selfie

- Only a contexts **parent can handle syscalls**

- **We add a little detail to the** trapping mechanism **in selfie:**

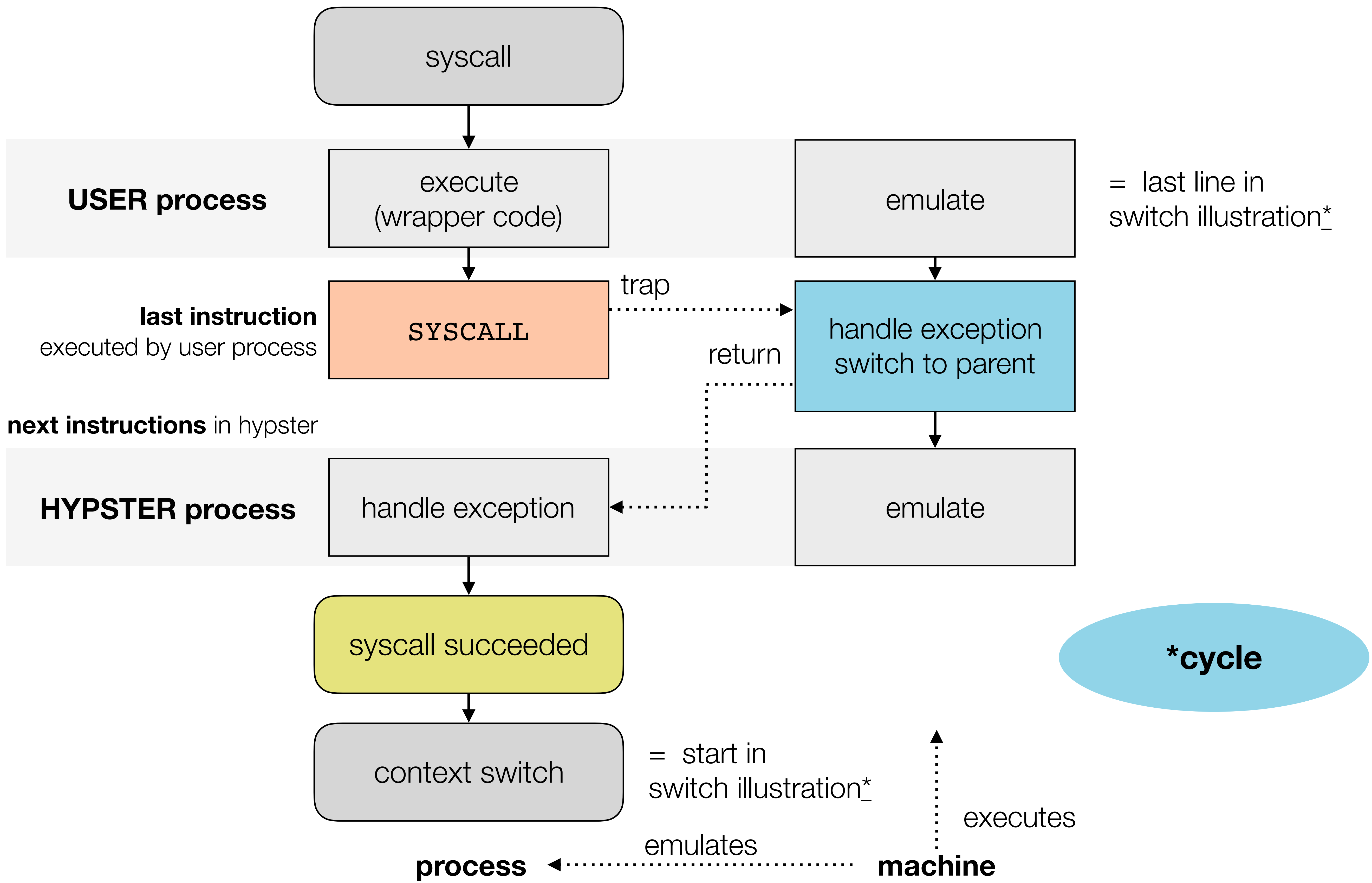
1. Program runs on machine and makes syscall → trap

2. Trap causes the machine to perform context switch to hypster
   - machine cannot handle syscall → machine is **not** the contexts **parent**
   - switch to parent → `get_parent(from_context != MY_CONTEXT)`
   - syscall/exception information remains in context

3. Hypster runs on the machine again
   - machine executes next instructions of hypster code
     - the instructions after the switch syscall in wrapper function
       (return from hypster_switch and handle exception)

4. After handling the exception hypster switches back to process

CPU Virtualization Syscall in Selfie



get_parent(from_context != MY_CONTEXT)

- Only a contexts parent can handle that contexts syscalls

- **Machine** can execute a context that is **not its own**
  - from_context could be a cached context executed on behalf of hypster
  - therefore the machine **checks** the contexts parent

- **Hypster** does not execute **any context**
  - hypster only handles the syscalls of its own context
  - from_context can never be a context that is not hypsters context
  - therefore hypster **does not check** the contexts parent

Summary Emulation vs. Virtualization

- **Problem** → interpreting code is slow, hosting code is fast
  - take out the middle man and execute directly on hardware
- The OS virtualizes the CPU
  - OS gives up control → context switch
  - has mechanisms to regain control → cooperative vs. preemptive
- Mipster switch and hypster switch
  - switch implemented in software
  - switch with hardware support

Memory Management

Address Space,
Memory Virtualization

Operating Systems

*An operating system **solves problems** that arise from concurrent execution of processes. It **manages resources** and controls the execution of many programs on a machine and **abstracts** that machine.*

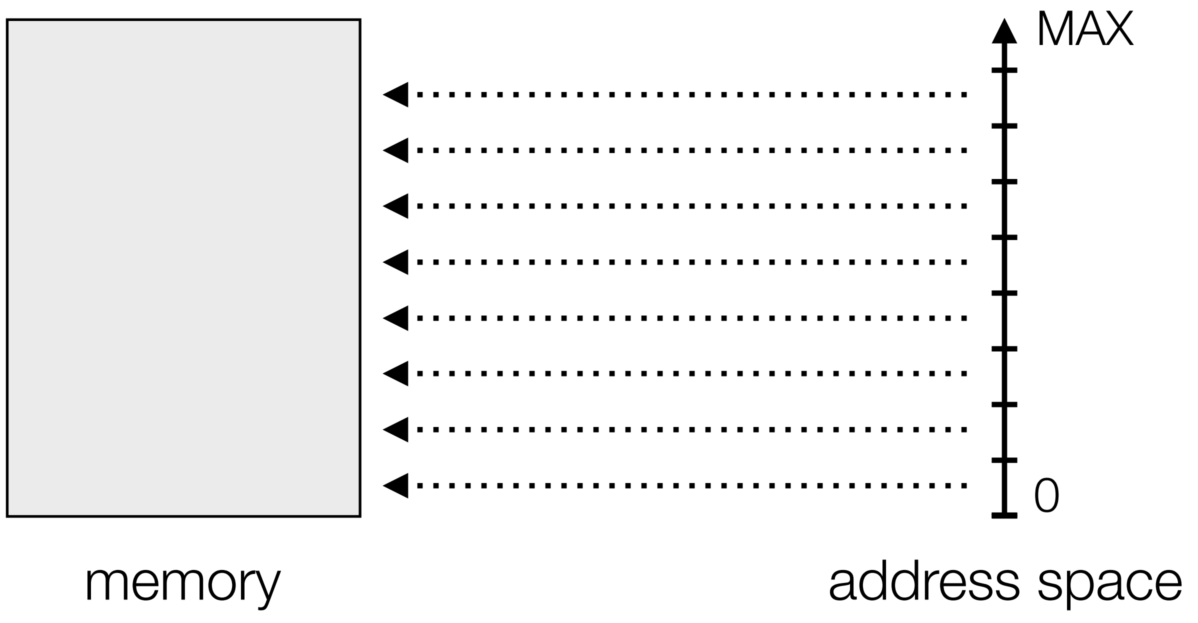
- Selfie can now execute two or more programs at the same time efficiently
  - the problem of having only **one physical CPU** was solved by virtualizing the CPU using a time-sharing approach
  - the challenge of sharing **one physical memory** was already solved by the operating system support implemented in selfie
- Nevertheless, we will discuss how an OS **manages one physical memory** and see how this is implemented in selfie.

1 Memory

Address Space

*An address space is a range of **memory addresses.** It is an **abstraction** of physical memory created by the OS.*

- A contiguous block of numbers (addresses) ordered from low to high
- Addresses are necessary to locate pieces of data in memory
  - each address uniquely identifies a piece of data



Memory Management Problem?

- The challenge with one physical memory
  - memory is fixed-sized
  - memory has to be shared among processes

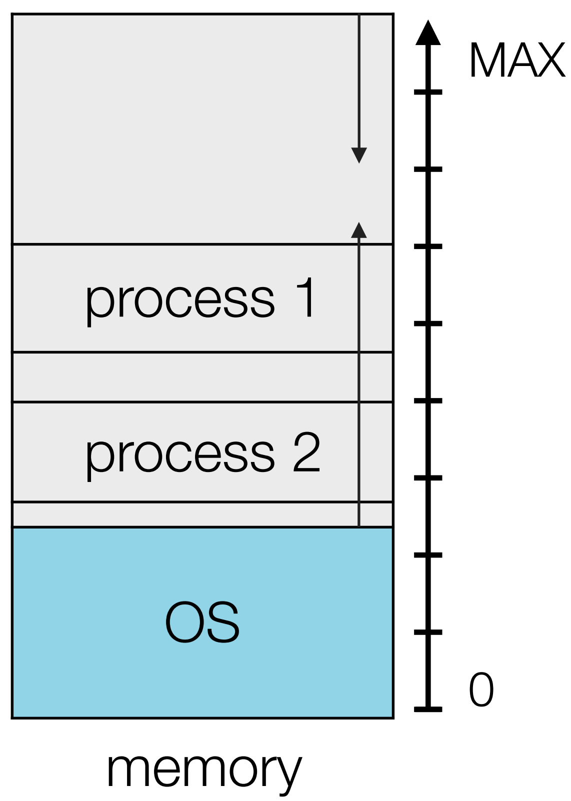
“How could the OS actually execute two processes with one memory concurrently?”

“We said earlier that memory is part of a processes context. This means that the memory gets switched when a context switch occurs.

Does this mean that processes (contexts) are stored somewhere on disk when they are not executing?”

Memory Management Shared Memory

- Possible solution
  - only the running process is in main memory, others reside on disk
  - **huge drawback** → context switches are slow (memory to disk)
- Better solution
  - keep processes in memory and switch between them



Memory Management Shared Memory

- Keeping several programs in memory creates new problems
  - **portability**
    → a process should run independent of where it is loaded into memory
  - **protection**
    → a process should not access memory outside its 'own' memory
    → processes should be spatially isolated
  - **transparency**
    → a process should be unaware of the fact that memory is shared memory
- All the above boil down to one issue - addressing

Memory Management Addressing

- **Portability**
  - addresses are needed at compile time, but are unknown before program is loaded into memory
- **Protection**
  - no access to addresses outside processes 'own' memory
- **Transparency**
  - process believes it can access every address from 0 to MAX

- Creating a layer of **abstraction** solves these problems
  → **virtualizing memory**

Memory Virtualization

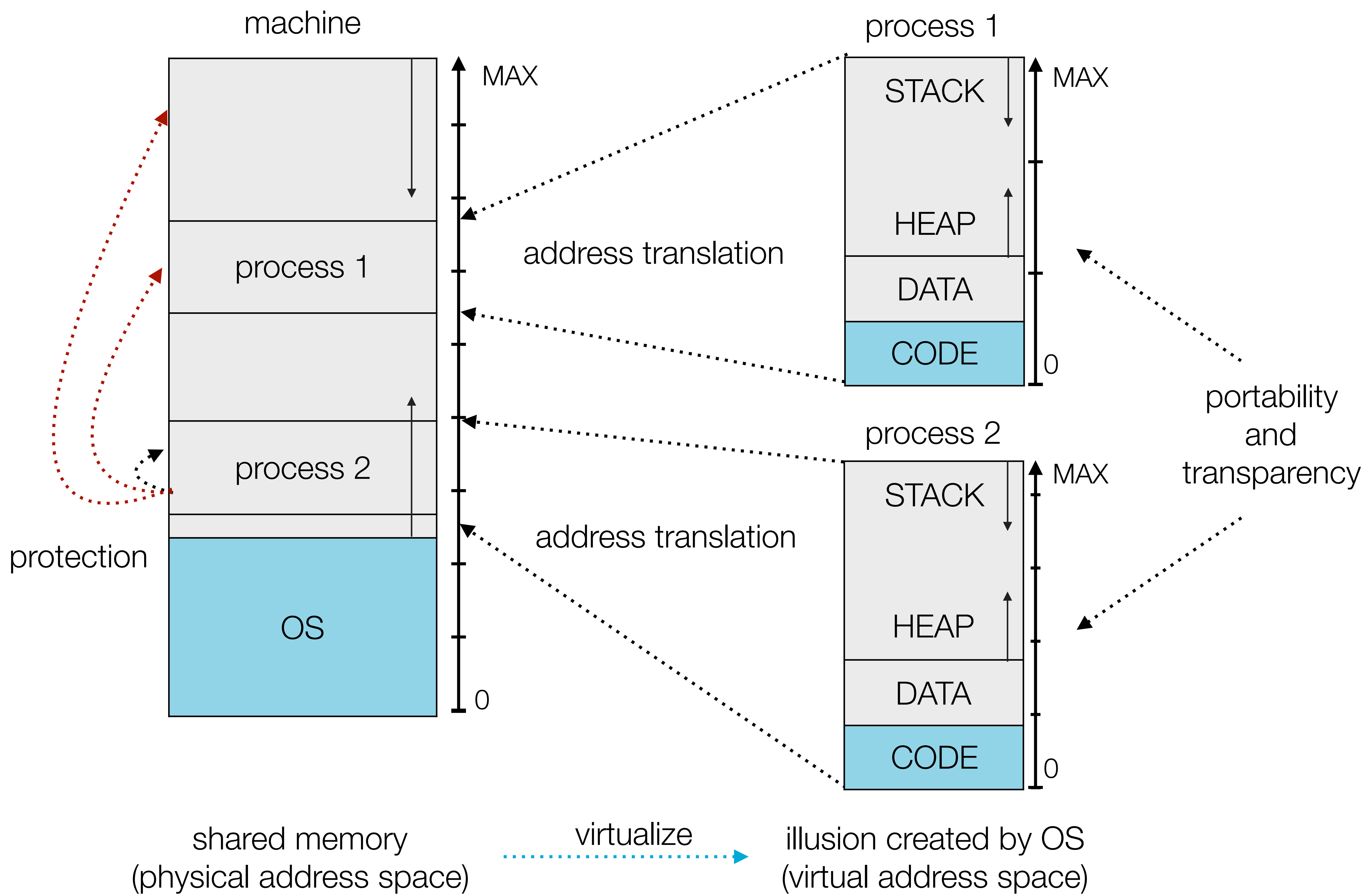
- OS abstracts physical address space using a technique called **address translation**
- OS provides each process with **virtual memory**, an **illusion of private memory**
  - process uses <u>**virtual addresses**</u>
    (code compiles with virtual addresses)
  - OS assigns physical memory to virtual memory
  - hardware with OS support **translates** virtual addresses to physical addresses where data is located
- **Portability and transparency**
  - each process is provided the same illusion
    (same memory layout and virtual address space 0 to MAX)
- **Protection**
  - depends on how OS virtualizes memory

Address Translation

*<u>Address Translation</u> is the process of **finding the physical address** for a given virtual address.*

- **Efficient** address translation performed by **hardware**
  - by part CPU often referred to as memory managing unit(MMU)
  - raises exception/trap when process attempts illegal access
- Supported by **OS**
  - manages memory (assignment, free?)
  - set up hardware so addresses are mapped correctly
  - handles exception raised by hardware (illegal access)
- How addresses are translated depends on memory virtualization technique used by OS

Memory Virtualization



Memory Virtualization Techniques

- The OS can use different techniques to virtualize memory
  - base and bound, segmentation
  - paging

Memory Virtualization Base and Bound

PROTECTION

- The OS assigns each process a **contiguous block**(chunk) of physical addresses
  - #assigned addresses = #virtual addresses
- It remembers for each process the lowest and highest address of this block → a **base** and **bound**
  - the range of addresses the process can access

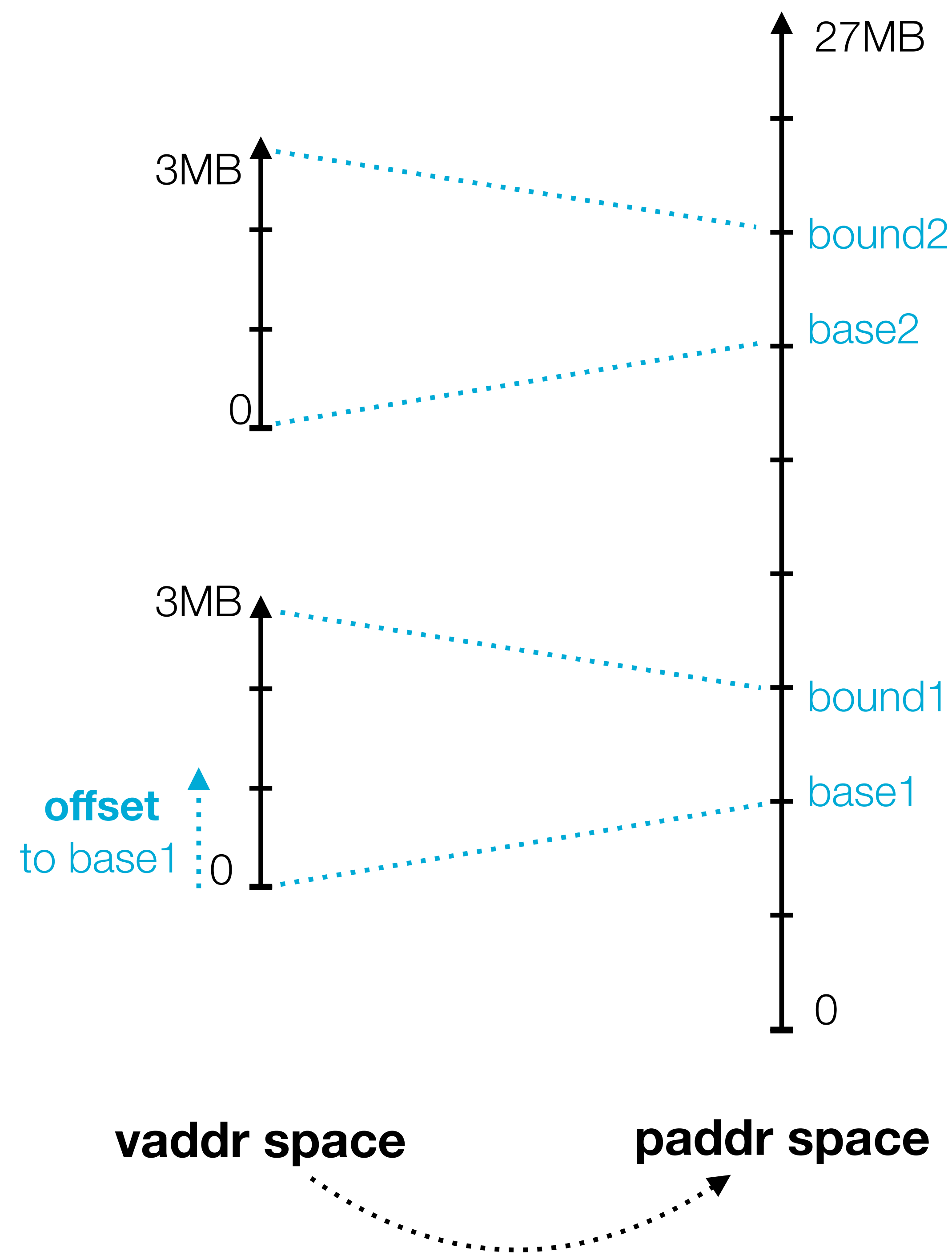
TRANSPARENCY

- Address translation
  - virtual address is the **offset** to the **base** in physical address space

Memory Virtualization Base and Bound

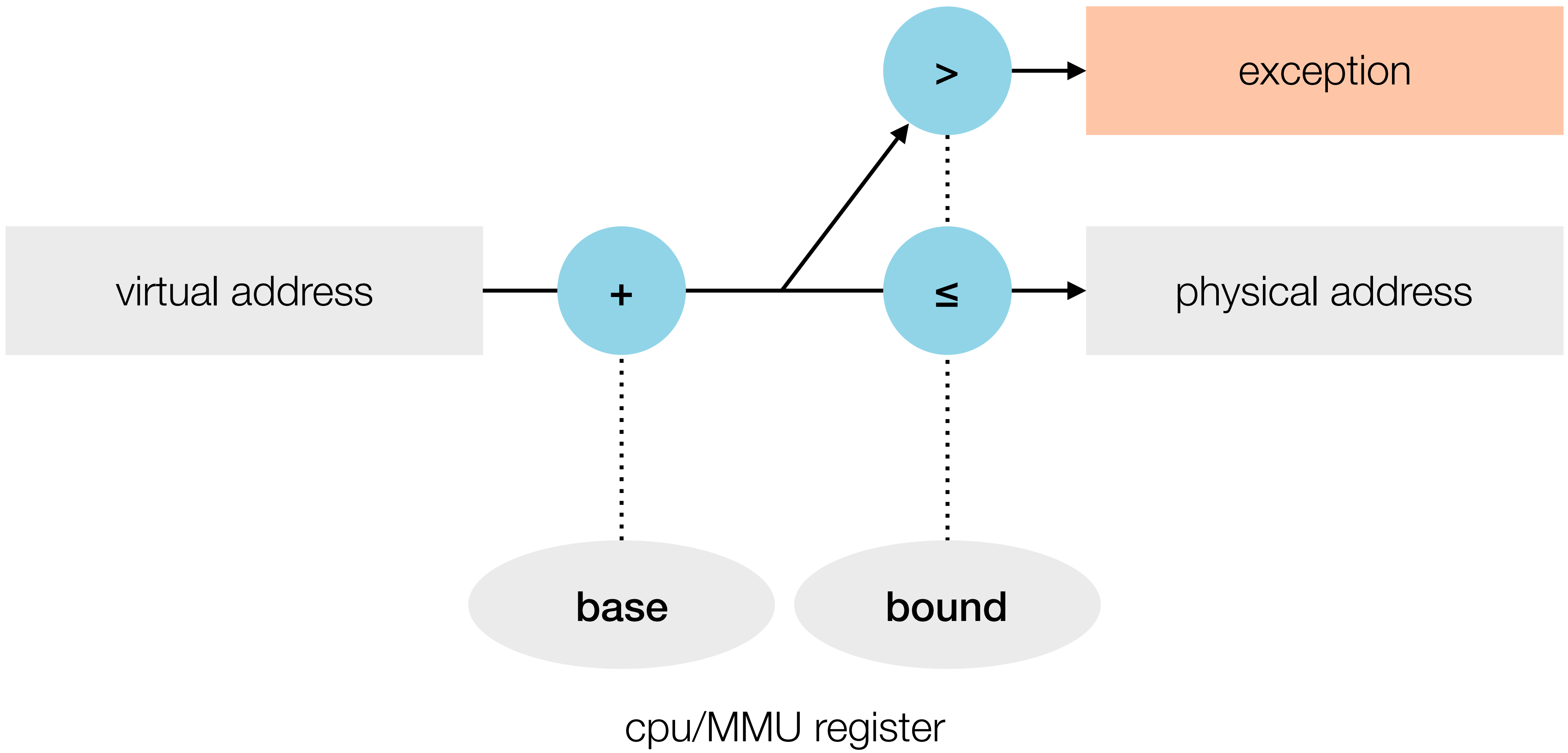
process 1	base1	bound1
process 2	base2	bound2

data structure to store
base and bound information



address translation
uses vaddr as offset to base

Memory Virtualization Base and Bound



Memory Virtualization Base and Bound

- **Advantages**
  - address translation is simple and fast
- **Limitations**
  - size and number of processes in memory is hardware specific
  - potentially a lot of unused memory
    - internal fragmentation - gap between heap and stack (improved by segmentation)
    - external fragmentation - find contiguous chunk

Memory Virtualization Segmentation

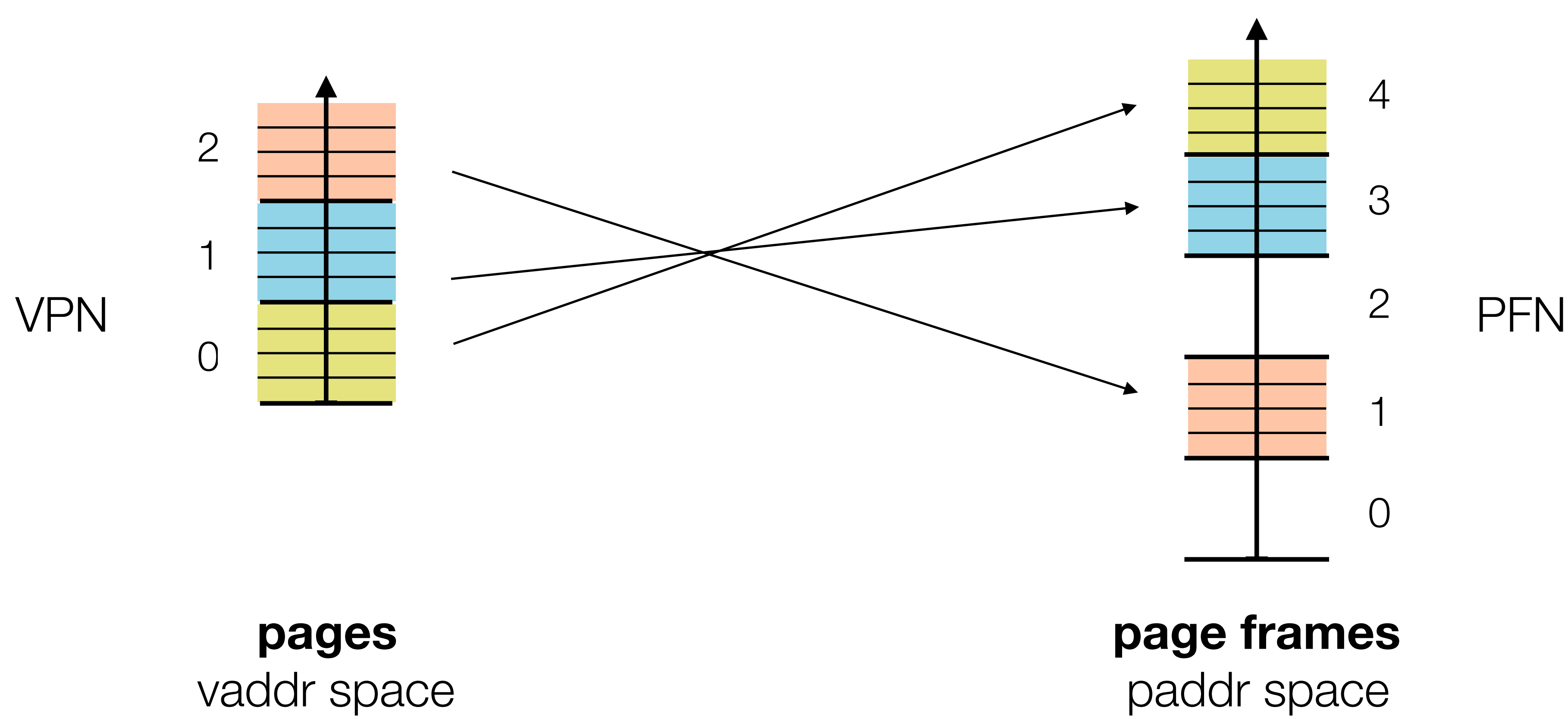
- **Similar** to base and bound
- Virtual memory divided into **variable-sized logical segments**
  - code, data, heap, stack
  - base and bound for each segment
  - variable-sized → external fragmentation
- Less internal fragmentation

“Why not chop virtual memory in fixed-sized pieces, wouldn't that help with external fragmentation?”

“Yes, it would. This is exactly the idea behind paging”

Memory Virtualization Paging

- Partition address space into fixed-sized chunks
  - virtual address space into **pages** → identified by virtual page number VPN
  - physical address space into **page frames** → identified by physical frame number PFN
- Pages and page frame have same size → 4KB worth of addresses
- Each virtual page maps to a physical page frame
  - to translate addresses these mappings are simply remembered in **page table**



Paging Page Table

A page table is the data structure used for ***address translation****.*
It stores the mapping from ***VPN to PFN*** *for each process.*

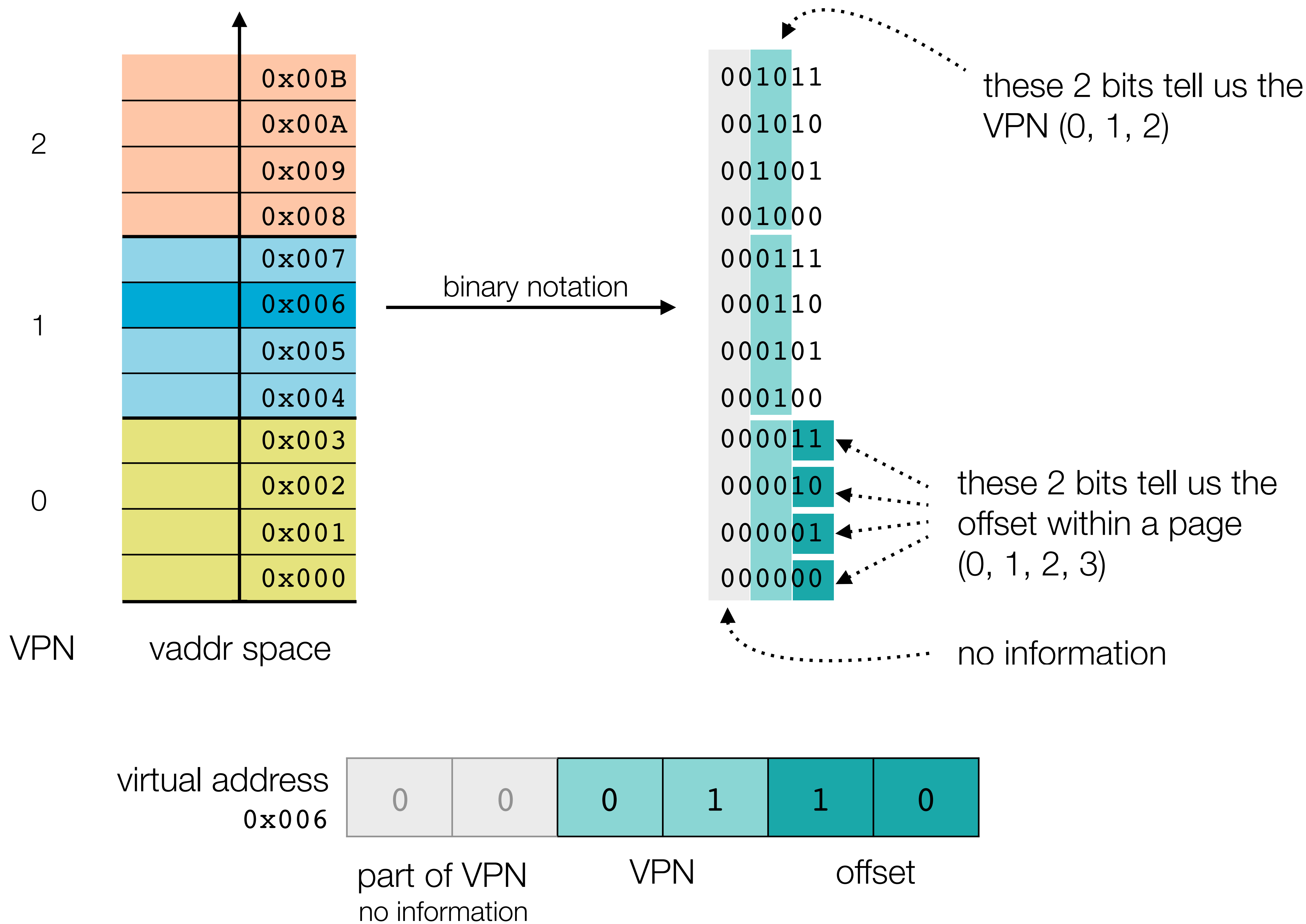
- **OS** maintains one page table per process
- Simple implementation
  - array lookup
  - indexed with VPN to find PFN

VPN	PFN
0	4
1	3
2	1

\

Virtual Address Logical Point of View

- there is **more to an vaddr** than you might think → take a close look at its binary notation*



*simplified example with byte aligned memory

Virtual Address Do the Math

- To uniquely identify ***x*** elements it needs $\lceil \log_2(x) \rceil$ bits
  - 4 bytes → 2 bit, 3 pages → 2 bit
- Therefore address size (#bits) puts an upper bound on memory size
  - 32-bit cannot address more than 2^{32} byte of memory
- Using 32-bit virtual address and 4KB pages and page frames (selfie)
  - 12 bit are needed for the offset
  - 20 bit remain for VPN (max. of 2^{20} pages possible)

Paging Page Table

- Size of one **page table entry**
  - necessary/minimum = #bits needed to identify PFN
    (depends on size of physical memory)
  - allocate more and use remaining bits to store additional information
    (dirty bit, present bit,…)

- Size of **page table** (selfie) - 4GB of virtual memory
  - #virtual pages * sizeof(page table entry)
  - 8MB (2^{20} * 8 byte)
  - huge therefore stored in memory
    - requires additional access for every translation

Paging Address Translation

`./selfie -c`

- Performed by hardware (MMU) with support of OS
- Only VPN gets translated, **offset is the same** for page and page frame

1. **Get VPN** from virtual address
   - get_page_of_virtual_address
2. **Lookup PFN** in page table using VPN as index
   - get_frame_for_page
3. **Build** physical address
   - calculate using PFN and offset of virtual address

“Let’s consider hypster and cached contexts for a moment…

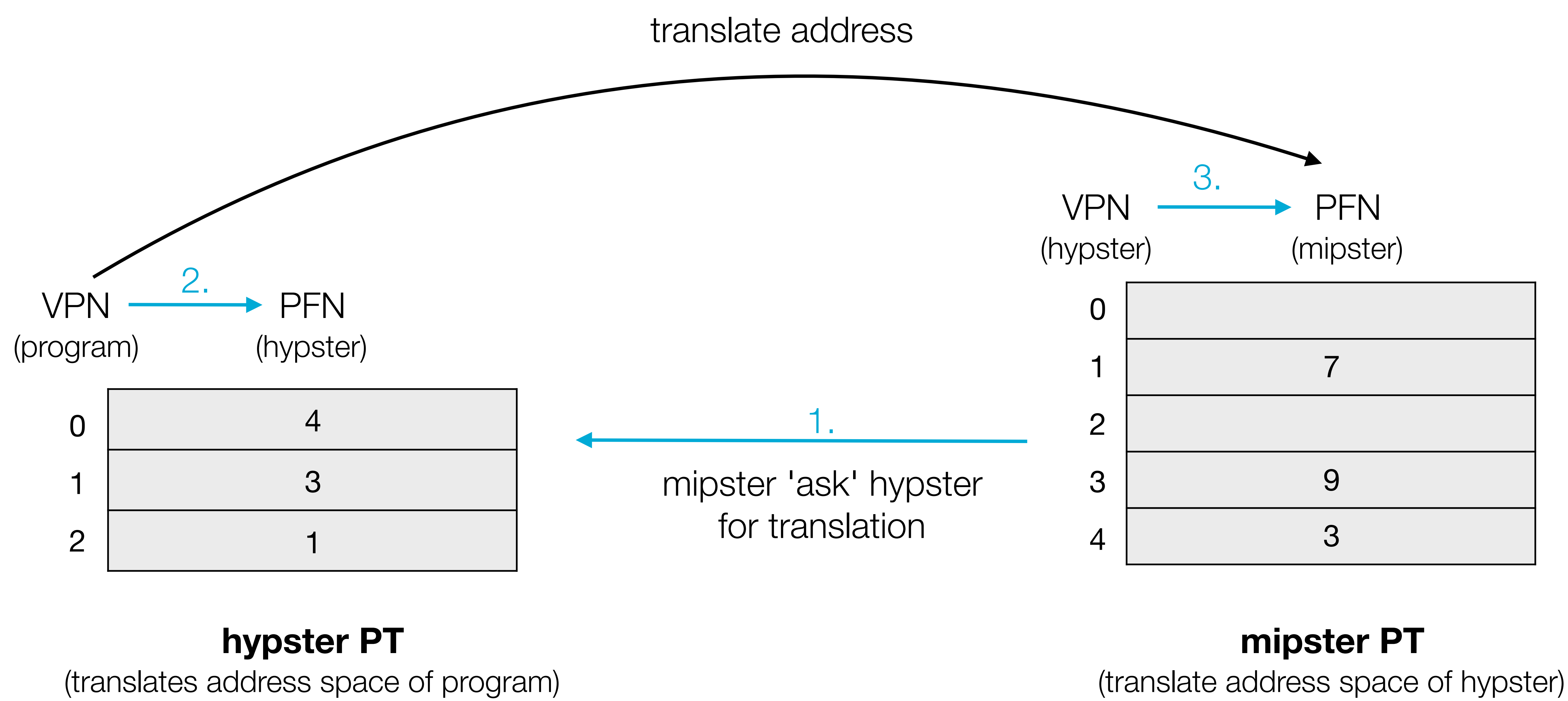
Address Translation is the last missing piece necessary to understand why we cache a hosted context.”

or skip next 3 slides

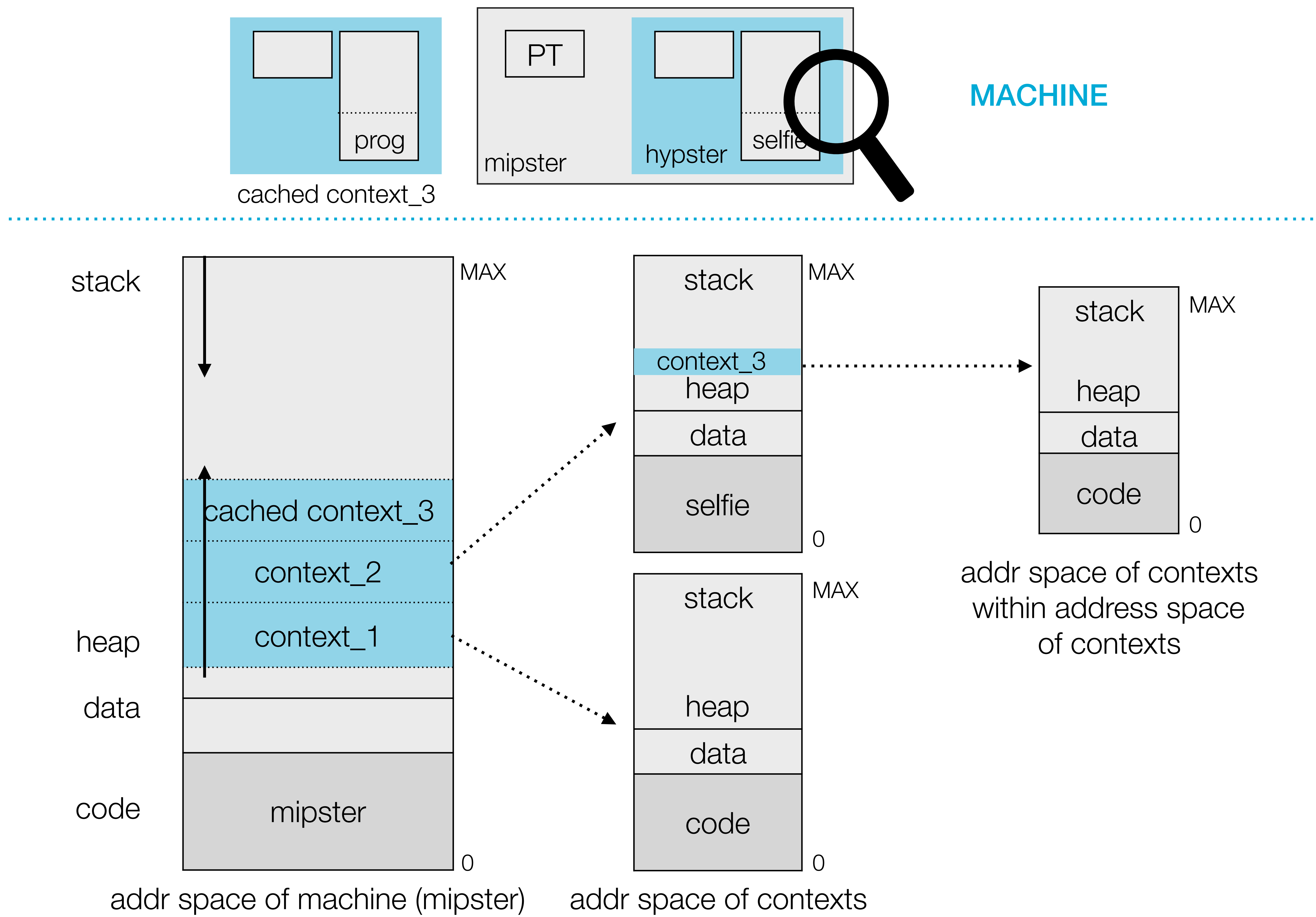
Cache Context Why?

- We are dealing with two **different address spaces**
  - the virtual address space of hypster on the machine (mipster or another hypster)
  - the virtual address space of the program hosted by hypster
- Caching is one way to **translate addresses** used by the hosted program
  - the machine creates a copies of the context in its own address space
  - the machine can then translate the addresses used by the context itself
- Another way would require some sort of 'multi-level' translation
  - mipster cannot translate addresses of program directly
    (ex. VPN of program to PFN of mipster on next slide)

Cache Context Go through PTs



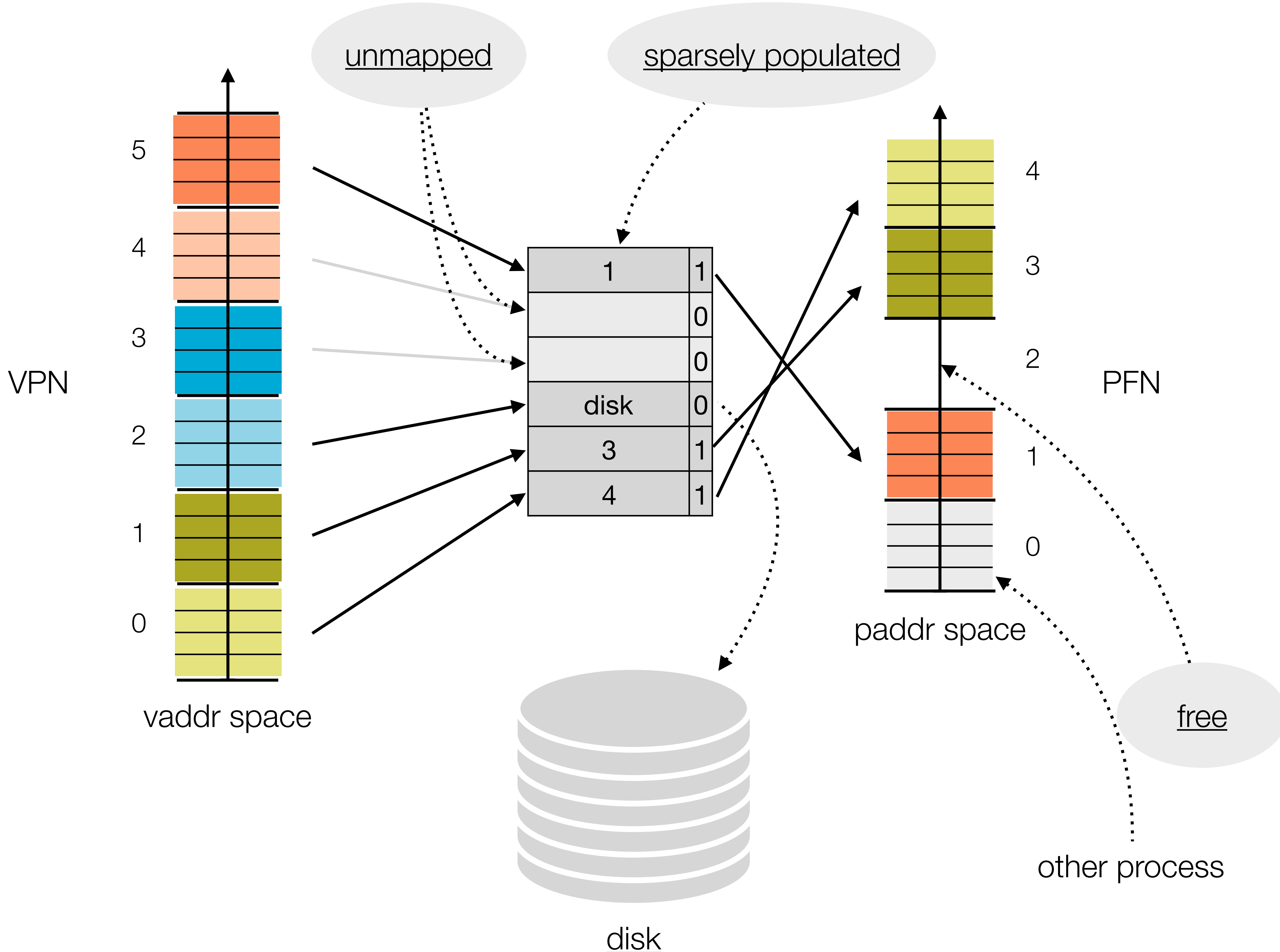
Cache Context Cache Context



Memory Virtualization Paging

- **Limitations**
  - additional memory to store page table
  - additional memory access for lookup
- **Advantages**
  - managing free memory is easy, external fragmentation is not a problem
  - paging is a **powerful** virtualization technique
    - completely abstracts physical memory (layout and size)
    - virtual memory is not limited by physical memory size (temporarily move pages to disk when not needed)

Memory Virtualization Paging



“A large virtual address space and a huge page table…isn’t that a huge waste of memory and disk space?”

“Yes, it most likely is.
(unless the process would really need all that memory)

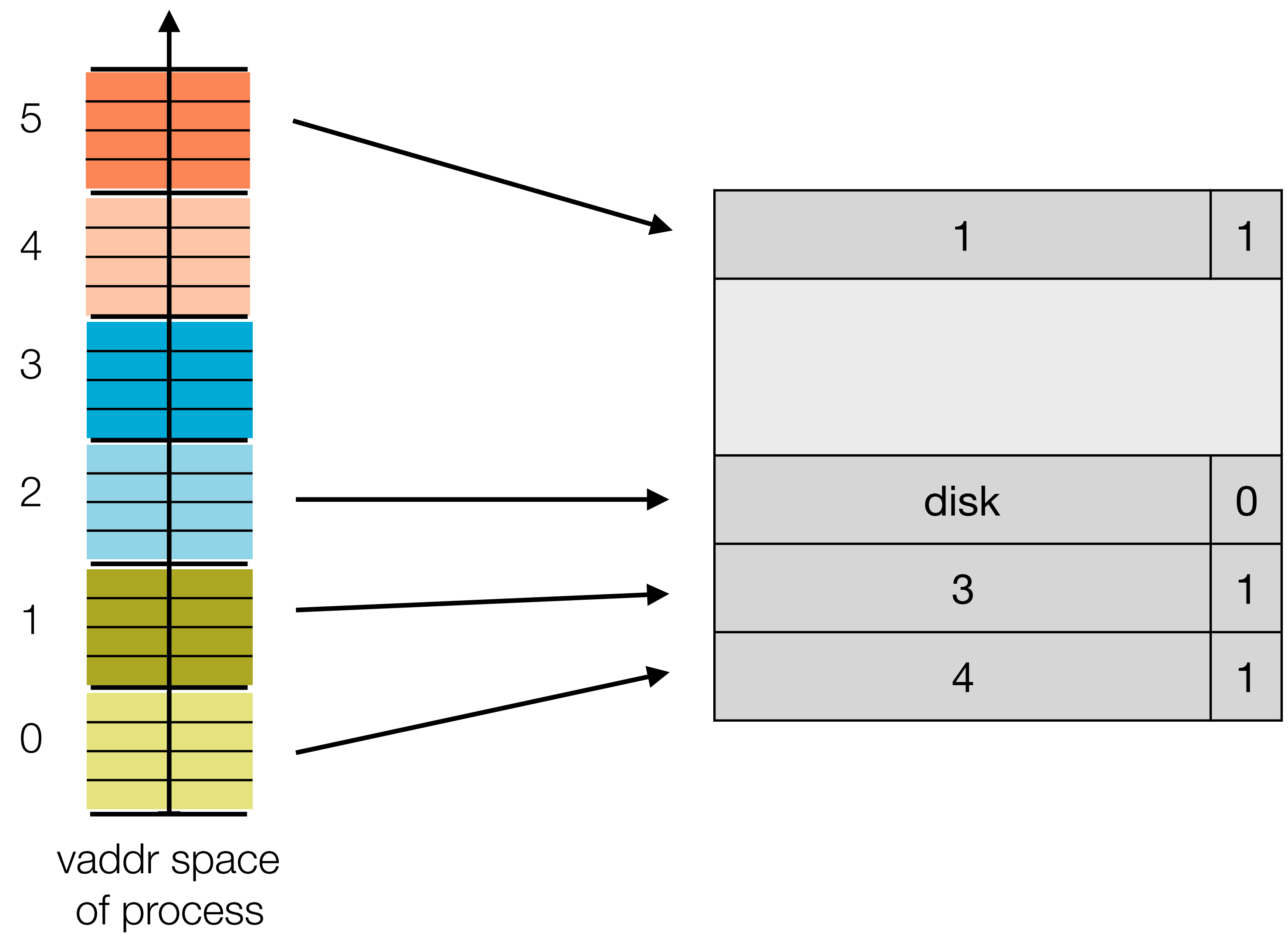
We can get around this easily by not mapping the entire virtual address space. Demand paging is one possibly way of doing so.

Paging on Demand

- Not the entire virtual address space is mapped when a process is created
- Only pages the process accesses are mapped
  - at process creation different options
    - maybe code segment, data segment, first page of stack (arguments)
    - or nothing is mapped
  - memory the process allocates is not mapped until accessed (lazy loading)
    - might be never → no consumption of physical memory
    - this also applies to the page table → **sparsely populated**
- When a process attempts to access an unmapped page an exception is raised
  - OS intervenes → allocates a new/free page frame and stores mapping in page table
  - **problem** → which frames can be used?

Paging on Demand

- page table is allocated in address space of OS
- page table is **sparsely populated**
  - gap between stack and heap
  - because of how virtual memory is laid out
- therefore page table itself is not entirely mapped



Paging on Demand in Selfie

`./selfie -c`

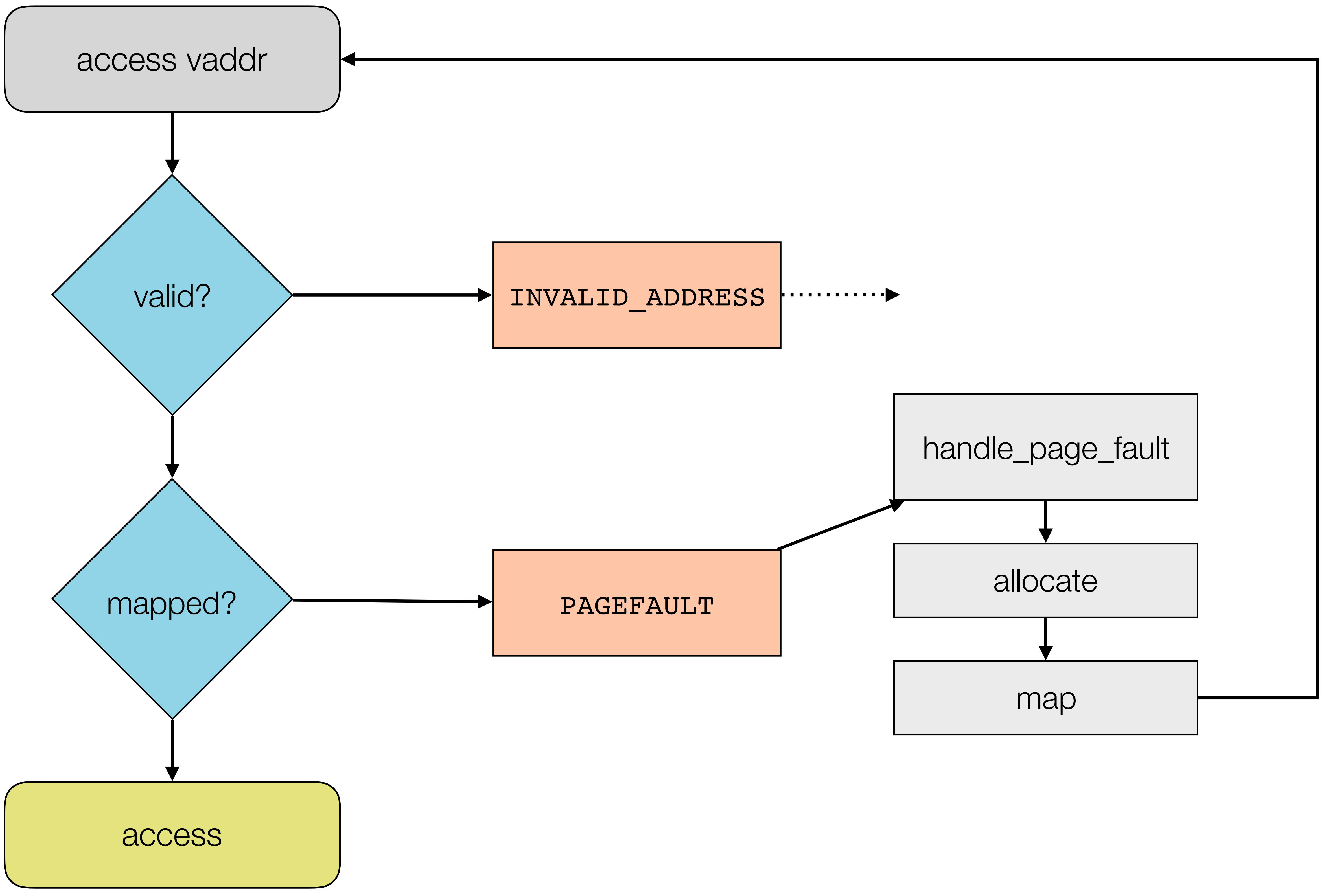
- **Process accesses an virtual address** (ex. `do_ld`)
  - check if virtual address is **valid**
    - within vaddr space, word-addressed
    - `is_valid_virtual_address`
  - check if page for virtual address is **mapped**
    - `is_virtal_address_mapped`, `is_page_mapped`
  - if both checks **pass**, translate address and load from physical memory
    - `tlb`, `load_virtual_memory`, `load_physical_memory`
  - otherwise an **exception** is thrown
    - `EXCEPTION_PAGEFAULT, EXCEPTION_INVALID_ADDRESS`

Paging Page Fault

`./selfie -c`

- Page faults are handled by the OS
  - page frame is **allocated** iff available → `palloc`
  - virtual page is **mapped** to this page frame → `map_page`
  - OS returns control to process
    - `pc` not increased
    - process executes same instruction again - successfully
  - `handle_page_fault`

Paging On-Demand in Selfie



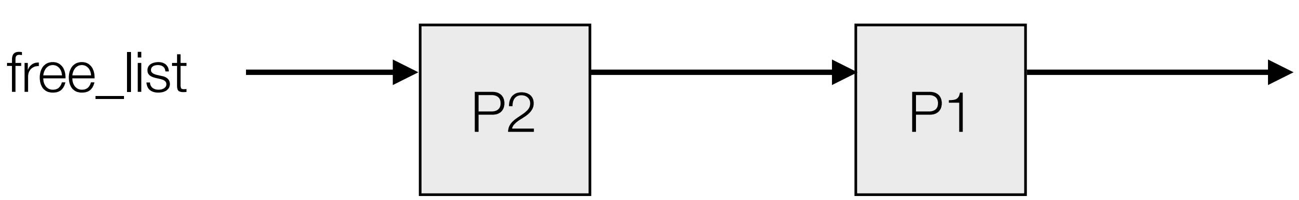
Paging Page Frame Allocation

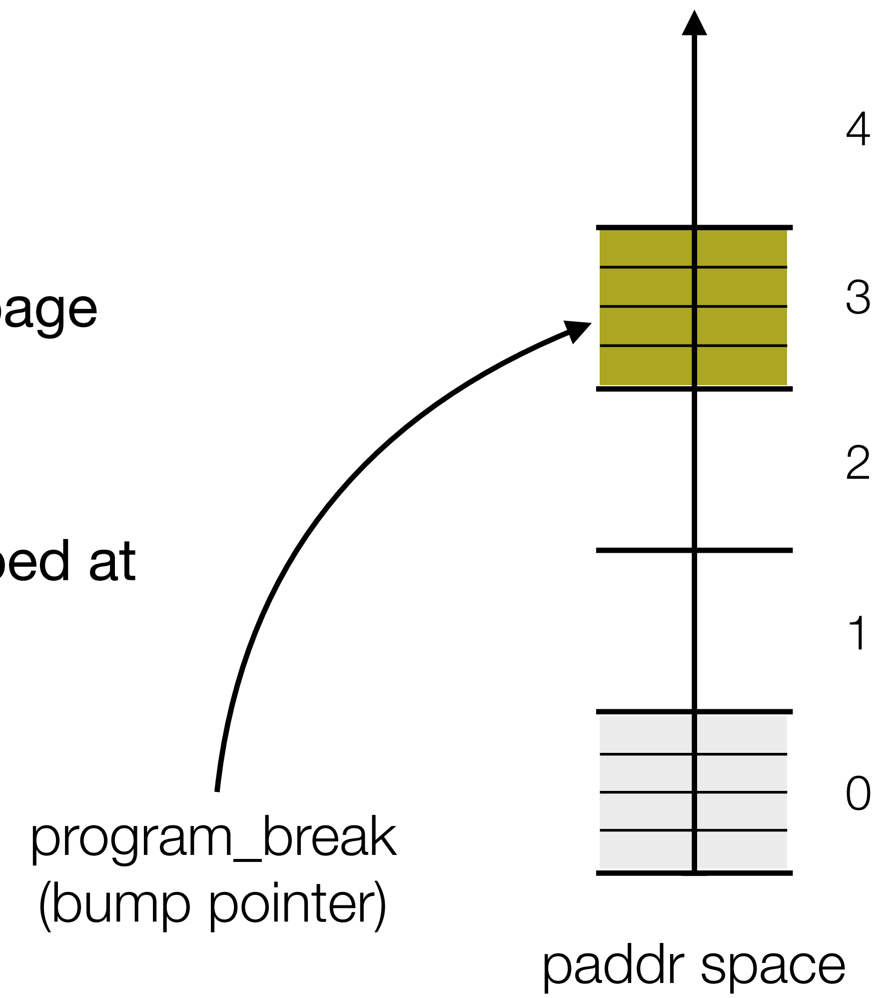
Problem *→ Which frames can be used?*

- The set of available frames can be partitioned int **used and free** frames
  - used_page_frame_memory, free_page_frame_memory
- Page frame allocator
  - keeps track of **free** pages
    - used memory is known by the application that uses the allocator (information in page table)
    - using a <u>free list</u>
- In selfie there are only 2 pointers to manage free memory

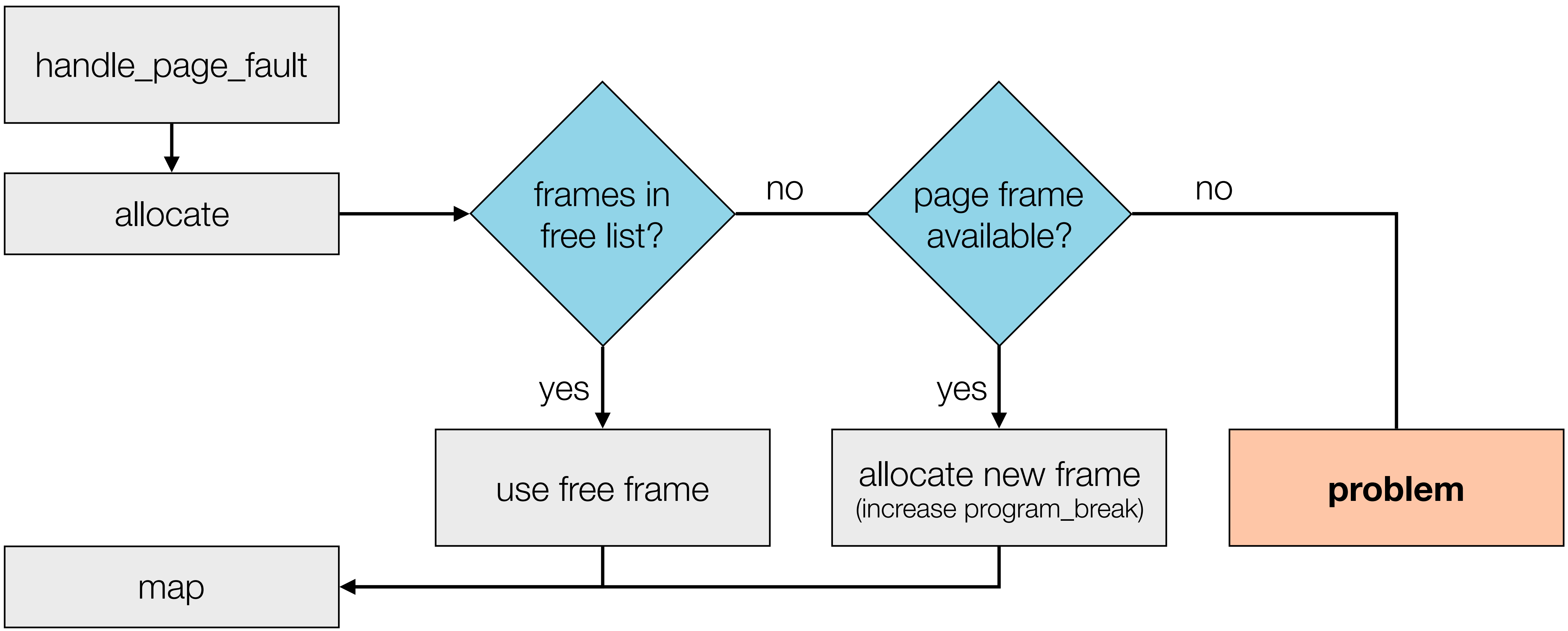
Paging Page Frame Allocation in Selfie

- Selfie uses a simple memory allocation concept
  - bump pointer allocation
- Selfie maintains **2 pointers** to keep track of free memory
  - initialized to 0
  - `free_list` → pointing to a simply linked list
    - containing pages that **became free**
  - `program_break` → pointing to the last allocated page
    - everything above is free
    - tells you max. #pages that had ever been mapped at the same time





Paging Page Frame Allocation in Selfie



Paging Swapping

Problem → *All frames are taken but we need a frame!*

- Ideal → find a frame that will **never be used/accessed again** and free it
  - unfortunately we cannot make such a statement
- We can however take the data from a frame and **store it somewhere else**
  - frame can be freed and data can be brought back if needed
- We split the problem:
  - What frame do we free?
  - How and where store data?

Paging Swapping

Page Replacement Problem → *What frame do we free?*

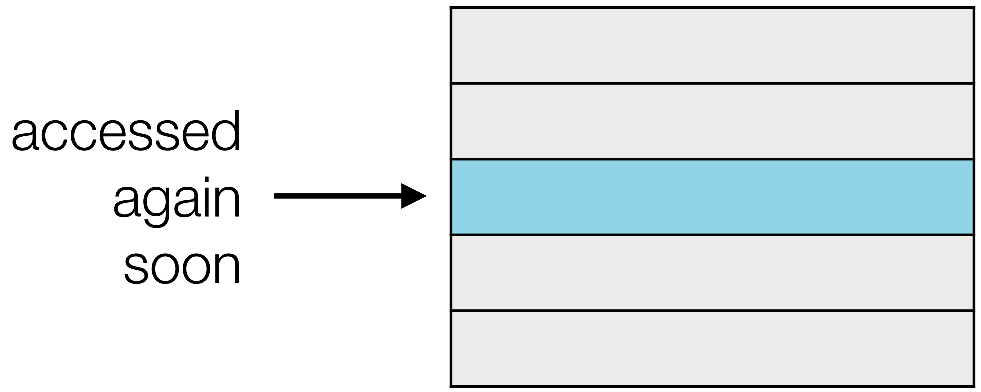
- Best frame would be the one that will be accessed furthest in the future
- Best we can do is find an **approximation** to that frame
- Different **eviction strategies**, depending on assumption we make
  - **most recently used** (MRU)
    - free the frame that was touched most recently
    - assumption → it probably will not be touched in a long time
  - **least recently used** (LRU)
    - free the frame that has not been touched the longest
    - assumption → bet on **locality**

Paging Bet on Locality

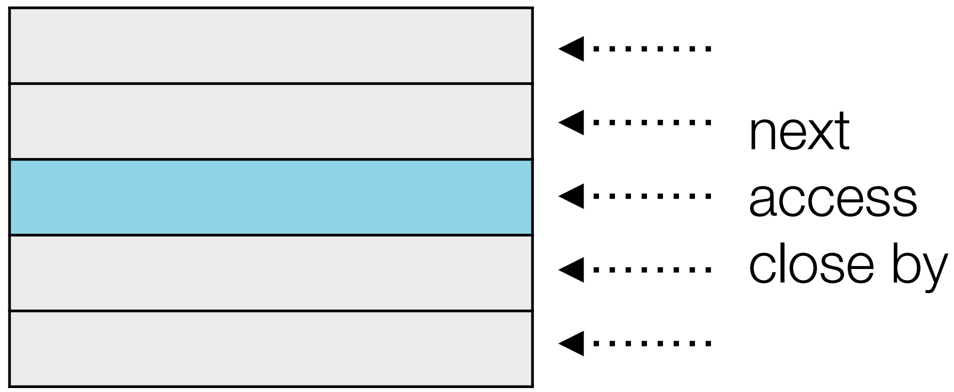
- Choosing LRU we bet on a property of code - locality

- **Temporal locality**
  - when a memory location (address) is accessed, it is likely that the **same memory location** is accessed again in the near future

- **Spacial locality**
  - when a memory location is accessed, it is likely that the **next** memory access happens in the '**neighborhood**'
  - most code shows spatial locality → not much jumping around



temporal locality



spatial locality

Paging Swapping

<u>Swapping</u> *→ How and where store data so we can get it back?*

- **'page out'**
  - move data (frame content) from main memory to swap space on disk
  - mark page table entry as 'not available' and remember address of data in swap space
- **'page in'**
  - access a swapped out page → **page fault**
  - move data from swap space back into main memory → page frame allocation (probably causing a 'page out')

Paging Page Fault

*A <u>Page Fault</u> is an **exception** that is raised when a process tries to access a page that is **not mapped**.*

- Page faults decrease performance - do 'administrative work'
  - fetch from disk or allocate, map,…
- When virtual memory is **overused** the number of page faults increases dramatically (page-in, page-out)
- <u>Thrashing</u>
  - more time spent on 'administrative work' than on process progress/execution
  - performance of machine degrades or collapses

Summary OS Introduction Continued

- We addressed the second **problem**
  - several processes using the same fixed size **memory**
- The OS **solves** this problem by managing memory
  - memory is shared among processes
  - OS virtualizes memory → illusion of private memory
- Concepts and mechanisms used
  - **abstraction/illusion** → physical address space, virtual address space (illusion of private memory), address translation
  - **virtualization techniques** → base & bound, segmentation, paging
  - **paging** → on demand, page faults, page frame allocation, swapping (page replacement strategies),

Process Management

Process Lifecycle:
create
execute
terminate

Threads

Operating System

*An operating system **solves problems** that arise from concurrent execution of processes. It **manages resources** and **controls the execution** of many programs on a machine and **abstracts** that machine.*

- By now we have a basic understanding of operating systems
  - Why do we need it?
  - What are its capabilities?
  - How does it work (techniques, mechanisms, concepts,…)?
- **Processes** are a fundamental concept
- The OS is responsible for **process management**
  - creating a process,
  - manage execution of processes,
  - terminating a process

Create

Process Management Creating a Process

- **mipsterOS/hypsterOS**
  - creates a fixed number of processes in advance*
  - execute these processes concurrently (time-sharing CPU)
  - this is not how real operating systems create processes
- **Real OS**
  - creates one process called the init-process
  - this init-process can create more processes, which again can create processes
  - the OS provides the service of creating processes through the **fork syscall**

* two or more, depending on your implementation of mipsterOS

Goodby mipsterOS

- mipsterOS was our first approach towards building an OS that can execute processes concurrently.
    - sufficient to demonstrate a multi-tasking system
    - however, not like a real OS (process management)
- We will continue with **mipster** as our OS and extend its operating system functionality

Back to mipster

- mipster creates one process → our init-process
- We will add
  - the **syscall fork()** to create more processes
  - a simple **scheduling algorithm**
  - modify exit handling to **exit** a process correctly

Syscall fork() Semantics

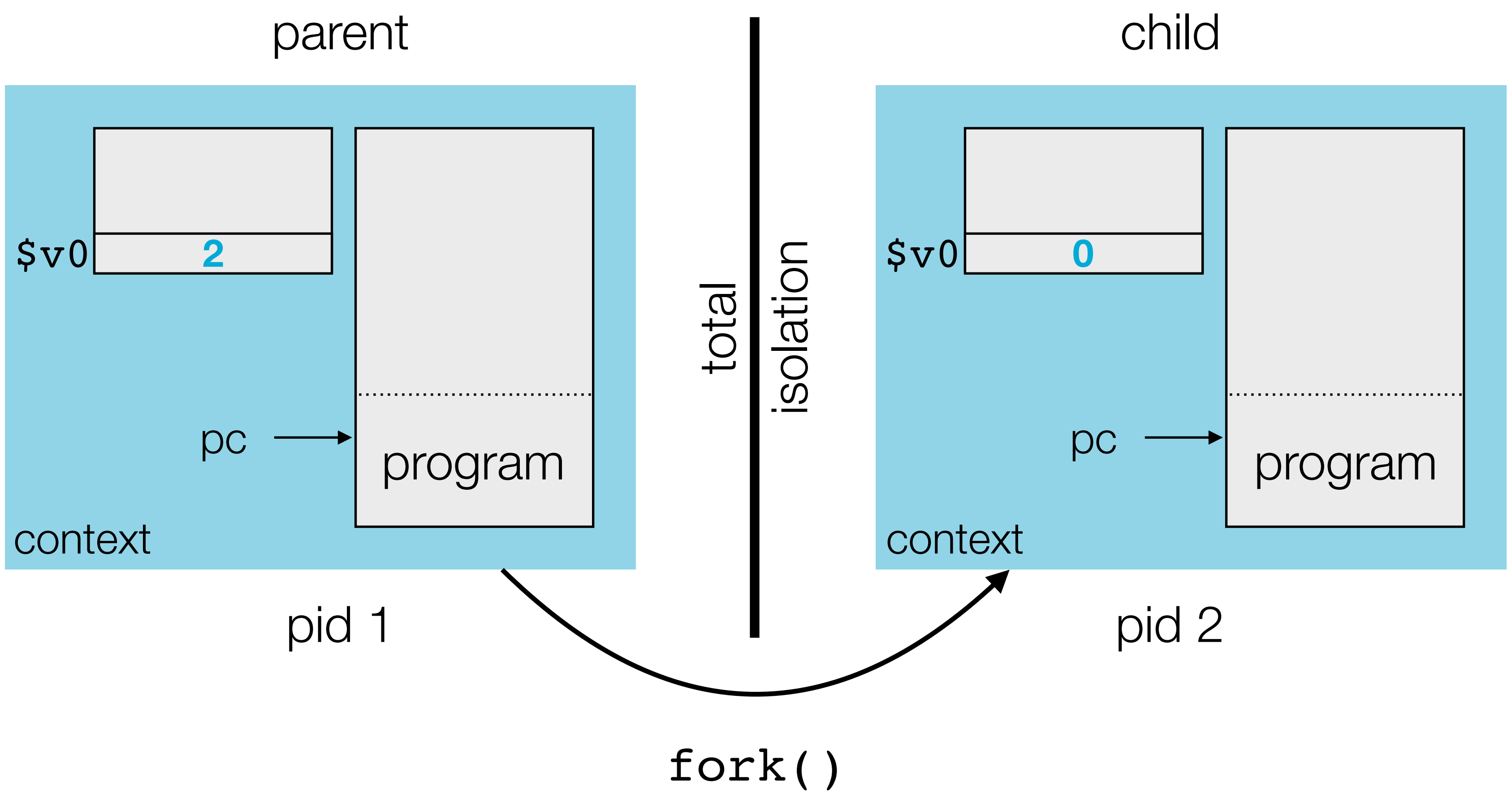
return **process identifier** of the process that was created (>= 0 if successful)

```
uint64_t main() {
   uint64_t child_pid;

   child_ pid = fork();

   …
}
```

simplest way: create a copy of itself that will run concurrently

- Terminology
  - **parent** → the process calling fork
  - **child** → the forked(created) process
  - **process identifier** → a **unique** number identifying a process (**known** by the process)
- fork() creates a copy of the process calling fork
- fork() returns the child's pid to the parent and 0 to the child
- parent and child process are completely <u>isolated</u>

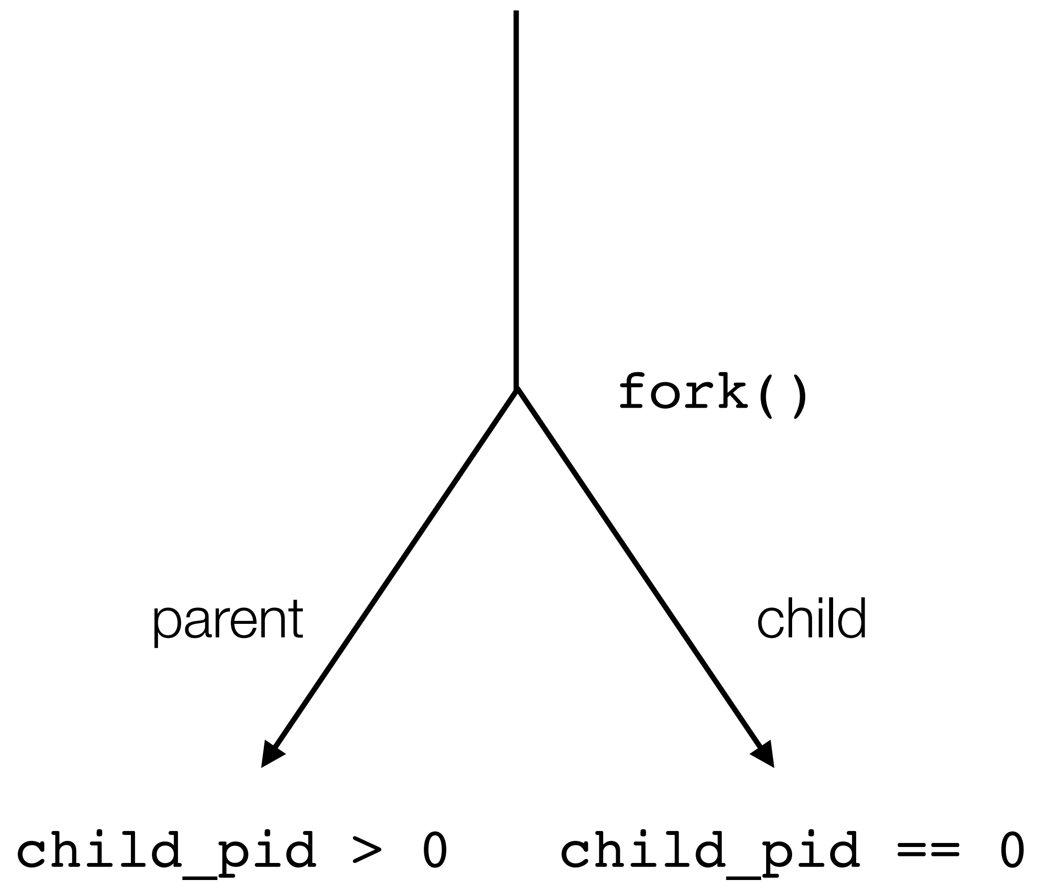
Syscall fork() Semantics



- The child context is almost an exact copy
  - execution state is the same
  - both processes will execute the instruction after syscall next
  - BUT the **return value (child_pid)** in register **`$v0`** is different
- Return value can be used to **distinguish** between parent and child process
  - different paths of execution

Syscall fork() Semantics

```
uint64_t main() {
   uint64_t child_pid;

   child_pid = fork();

   if (child_pid > 0)
     //parent
   else if (child_pid == 0)
     //child
   else
     //error occurred


}
```



Syscall fork() Essentials

- Generating **unique identifiers**
  - simple solution → bump pointer (global variable) maintained by OS

```
new_pid = old_pid + 1

//use new_pid

old_pid = new_pid
```

  - advanced solution → reused pids of processes that already exited

Syscall fork() Essentials

- New syscall
  - emit → wrapper function in the compiler
  - implement → as part of the OS
- **Deep copy** of parent context
  - create new context
  - copy context structure, registers, memory (isolation)

“Copying context structure and registers sounds simple: just allocate and copy…

But selfie uses paging…
How do we copy memory?”

“Without paging, when each context has a block of memory, we could do just the same - allocate and copy…

With paging, we need to dig a little deeper.

Start by looking at what you have:
virtual addresses and a
page table to translate them”

Syscall fork() Deep Copy

- Remember how selfie manages memory → paging
  - the context uses virtual addresses that are mapped to physical addresses using the page table
- The forked process has the same virtual addresses but they are mapped to different physical addresses → deep copy of page table
- **Deep copy simple**
  - copy page table → **new page table** and
  - copy entries → **new page frames** and copy content
  - new page table = different mapping same content
- **Deep copy advanced (copy on write)**
  - use the **same** page table (same pointer) **until** the first memory update
  - even then it is possible to use the same mapping except for the changed frames

Syscall fork() Deep Copy

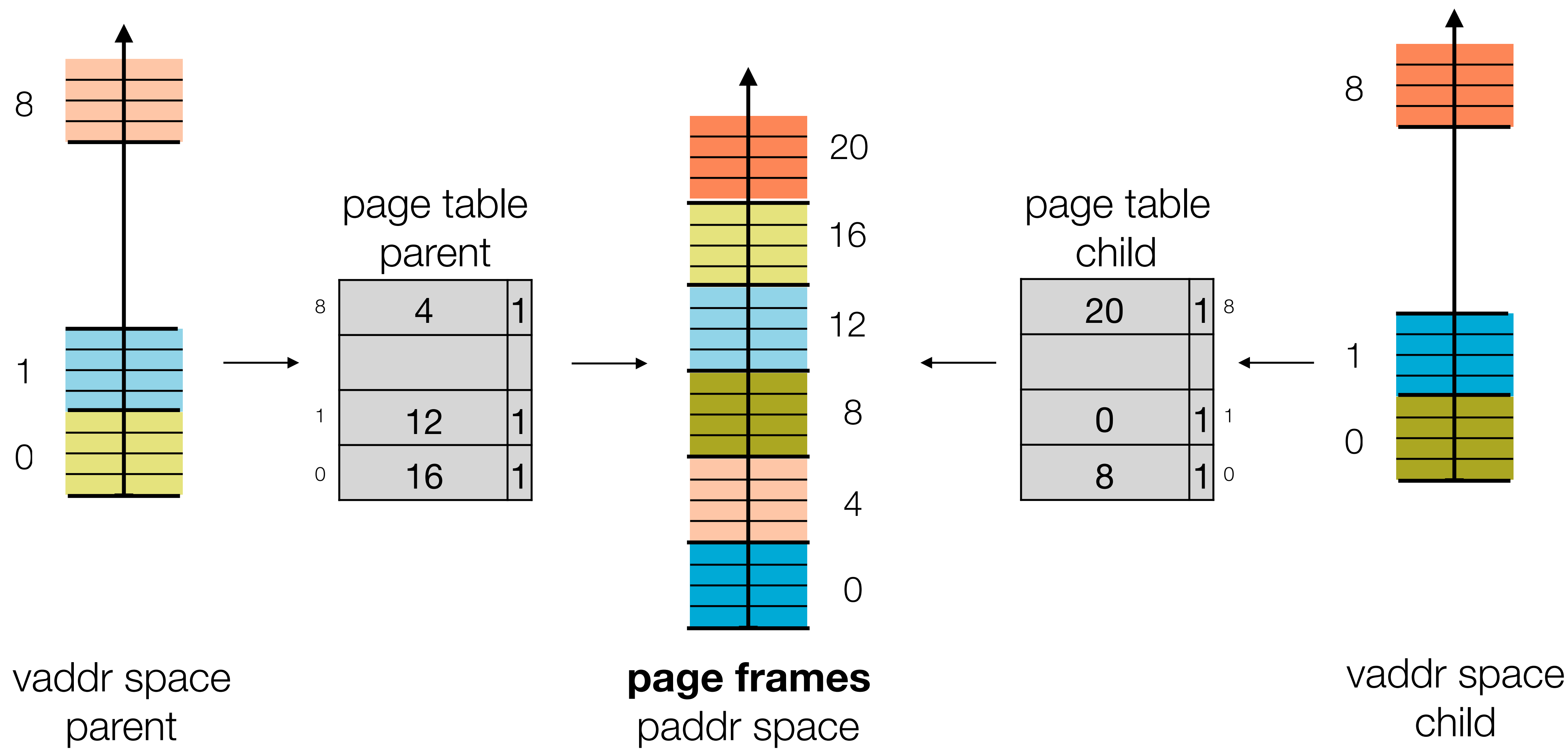
```
copy_memory() {
    allocate_new_child_page_table();

    for page in parent_page_table {
        new_page_frame = copy_page_frame(page);
        map_in_child_pt(page, new_page_frame);
    }
}
```

remember that the page table is **sparsely populated**

`palloc()` to allocate new page frame

- Page → index of page table entry
- Page table entry → address physical page frame

Syscall fork() Deep Copy



- **Sparsely populated** no need to copy everything
  - `lo_page` → lowest mapped page - page 0
  - `mi_page` → highest mapped page of lower part (code, data, heap) - page 1
  - `hi_page` → highest unmapped page of upper part (stack) - page 7

Syscall fork() Implementation Tips

- Use the `malloc` as reference
  - no parameters, one return value
- The operating system provides this service
  - `fork()` syscall → API: `uint64_t fork()`, ABI: syscall number
  - a libc* wrapper function → syscall interface (ABI)
  - handling the syscall → implementation in mipster
- Careful with virtual and physical addresses!

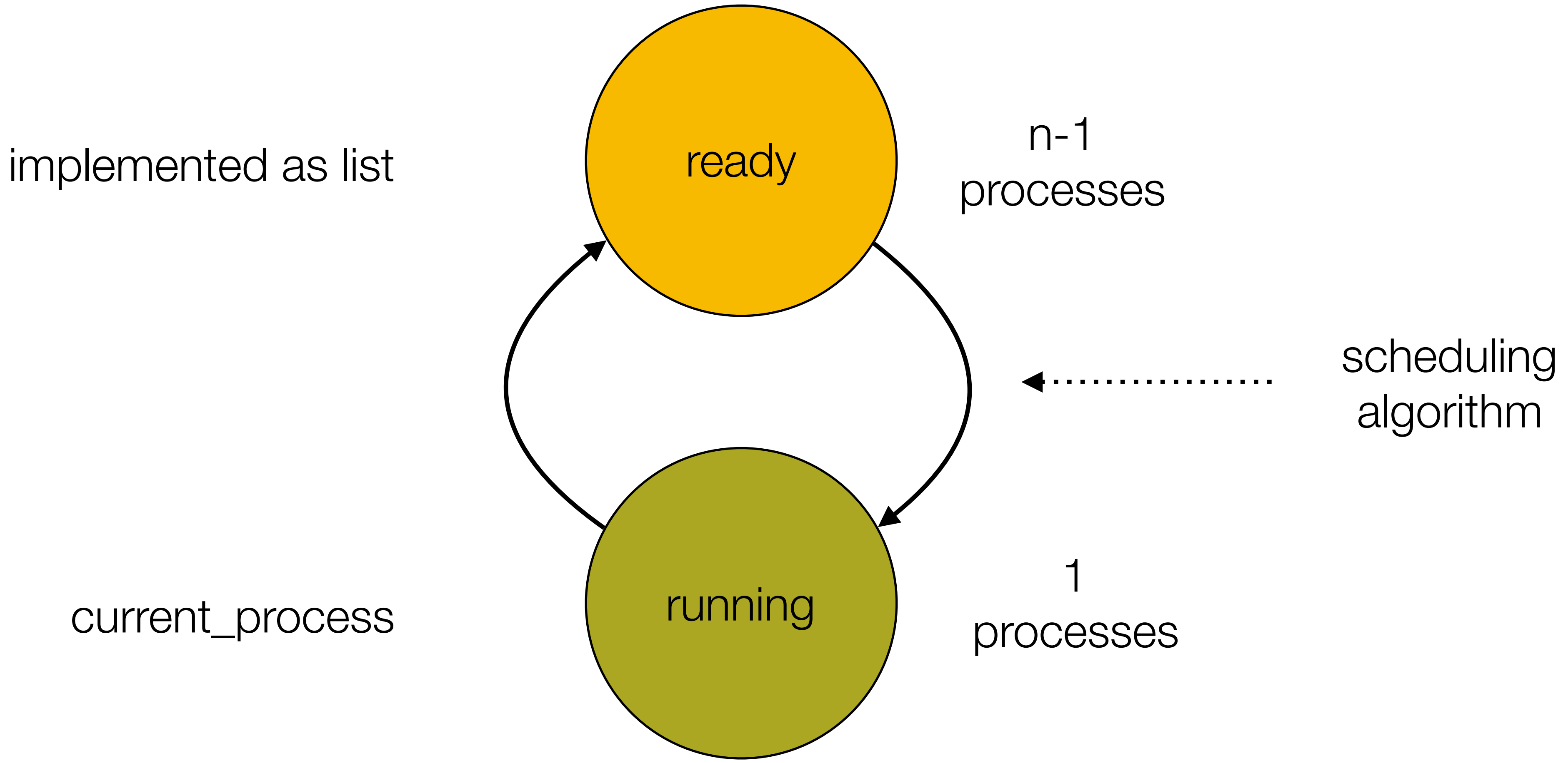
Manage

Process Management Manage Execution of a process

- At any given time a process can be in one of two **states**
  - running on the machine
  - ready and waiting to be executed
- **Where to put ready processes?**
  - in selfie there is a list of processes (used_contexts)
- **Which process to execute?**
  - scheduling problem
    - ex. which context execute after syscall or timer interrupt
  - solved by scheduling algorithms
    - a simple algorithm is Round-robin

Manage Execution Process State

- So far we know two states a process can be in
  - running
  - ready
- A scheduling algorithm decides how process change state



Terminate

Process Management Terminate a process

- A process is terminated with an exit syscall
- So far the exit syscall of a process tears down the emulator*
  - return from mipster
  - reasonable if only one process is executed
- Different behavior now that processes are executed concurrently
  - do not tear down emulator if other processes are left (ready)
  - **delete** the context that exited

* unless you already implemented this differently in mipsterOS, hypsterOS

Manage and Terminate Implementation Tips

- mipster manages the process list
  - you can use `used_context` as is
  - insert new processes → `fork()`
- mipster schedules the processes
  - `to_context != from_context`
  - implement Round Robin to choose next process
- mipster handles the exit of a context
  - delete to context (removing it from the list)
  - continue with next ready processes
  - exit if no ready process is left

**“What happens when the parent exits before its children…
What is the intended behavior?”**

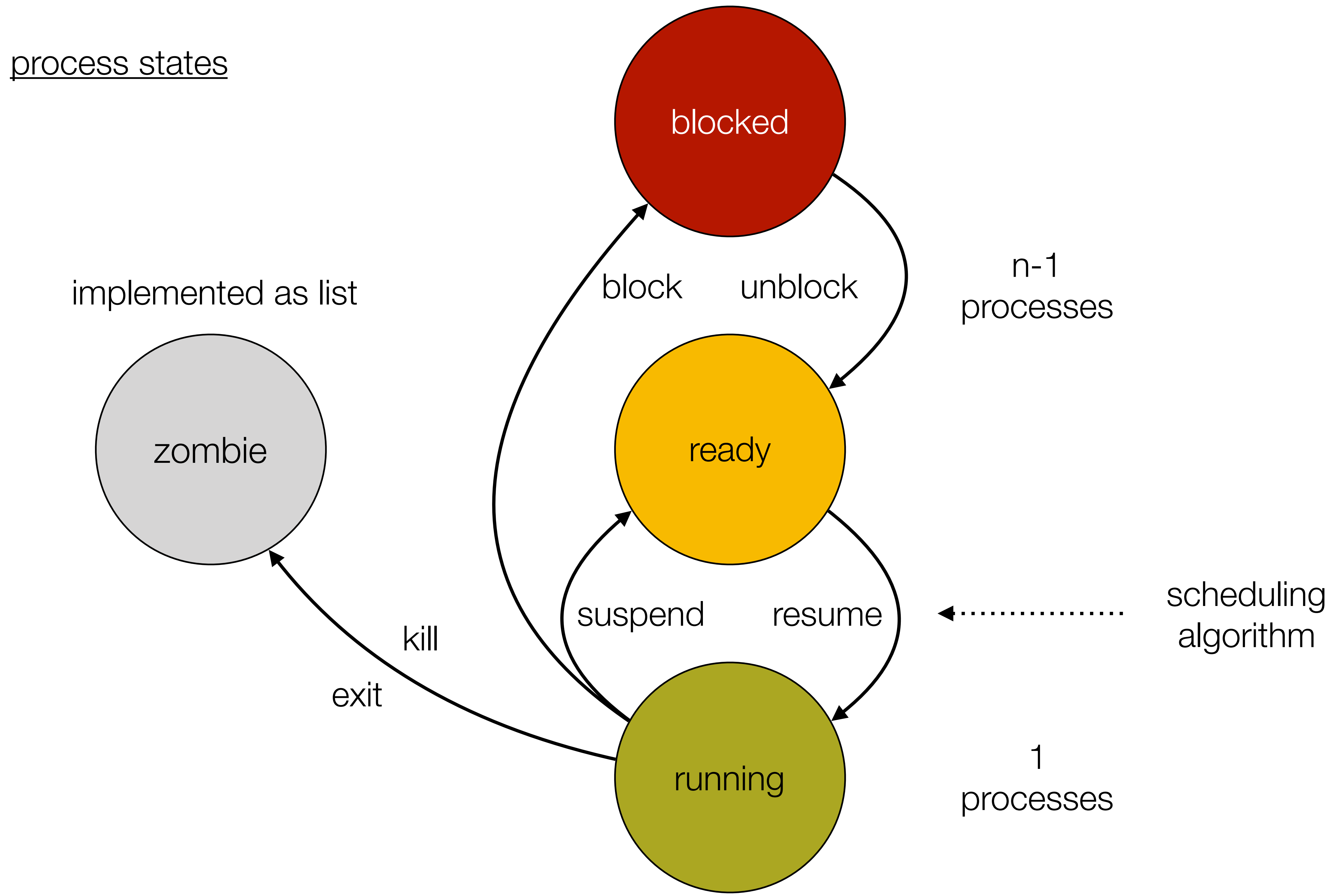
“Whatever you implement!

…the children could be terminated too or
they could become orphans”

Process Management Terminate a process

- A process knows its children
- The parent might still need information from a child that exited (like exit code)
  - the child process cannot be deleted before the parent exits
- We introduce new states
  - **blocked** → not ready for execution
  - **zombie** → not running, ready or blocked but not deleted
- Introduce new syscalls
  - `wait()` → suspend execution at this point until one child terminates
  - `kill(child_pid)` → terminate a child forcefully

Process Management States



"traffic light model"

Summary Process Management

- The OS is responsible for process management
  - creating a process,
  - manage execution of processes,
  - terminating a process

- Concepts and mechanisms used
  - **process model** → isolated processes
  - **process state** → ”traffic light model” (running, ready, blocked and zombie)
  - **unique identifiers** → create them using bump pointer
  - **scheduling** → change state, round robin

Recap Concurrency

- Remember were we started
- **Goal ACHIEVED**
  - execute many processes seemingly at the same time
- **Problem** of limited resources **SOLVED**
  - one physical CPU
  - one physical memory
- **HOW**
  - isolated processes managed by the OS

“I guess isolation also means there is no communication between processes?”

“Primarily yes, but…

…the OS also provides a mechanism, namely inter process communication (IPC), that allows process to communicate with each other.

We can, however, get communication easier when we use a different process model - threads”

Threads

Process vs. Thread

- **Process** → '*start private*' - approach
  - inter process communication (IPC) to communicate
  - ex. give processes access to same memory → mapping (virtual address space to same physical address space)
- **Thread** → '*start shared*' - approach
  - share almost everything from the start
    - communicate through shared memory

Threads

Threads describe ***another process model***, *a different abstraction created by the OS.*

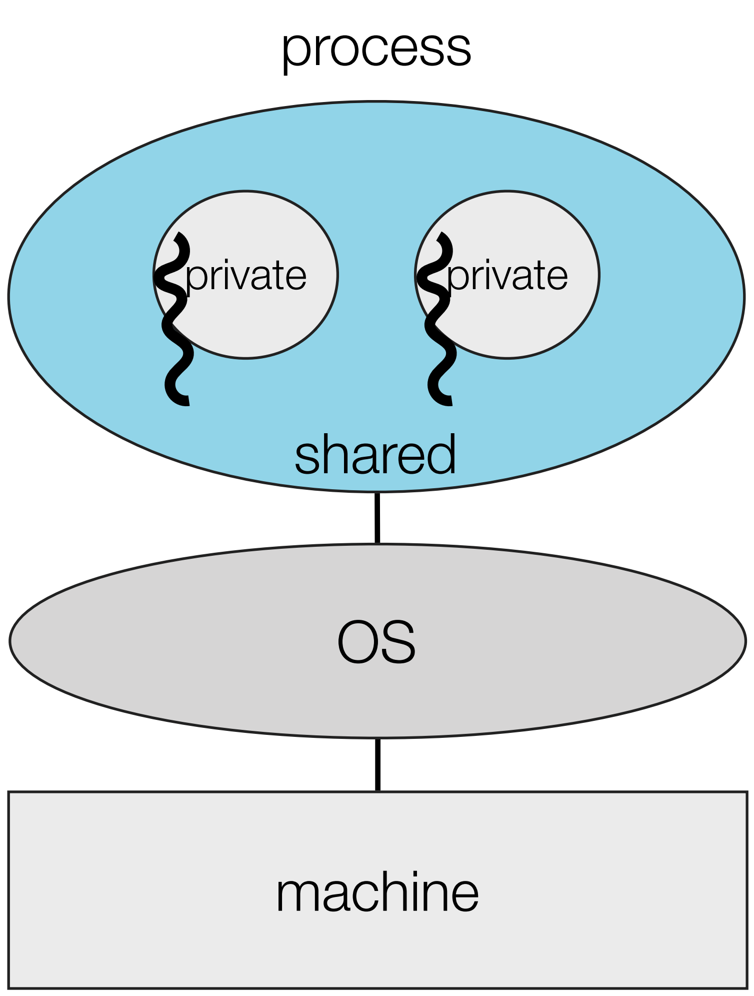
- Can be seen light-weight processes
  - threads are generally part of a process → a process can have multiple threads of execution
  - carry less state information as they share part of their state
- Threads **share** process state
  - parts of the memory (address space) and resources
- Allow efficient resource sharing and easy communication

Threads

- One process with **two threads of execution**

Word-Process

one thread takes keyboard input while another thread shows it on the screen



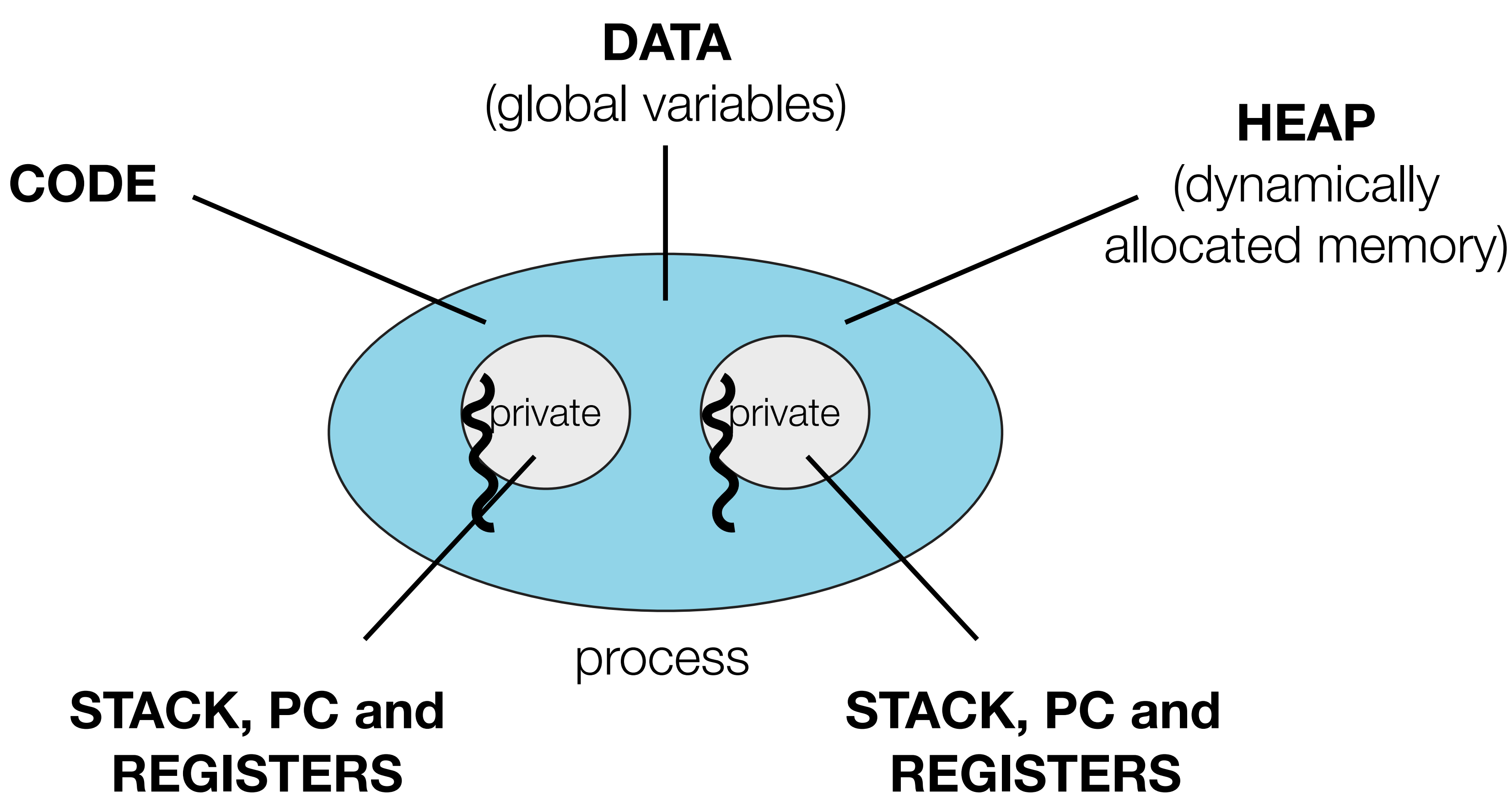
Process

threads can be used to **divide workload** → requires communication and sharing information

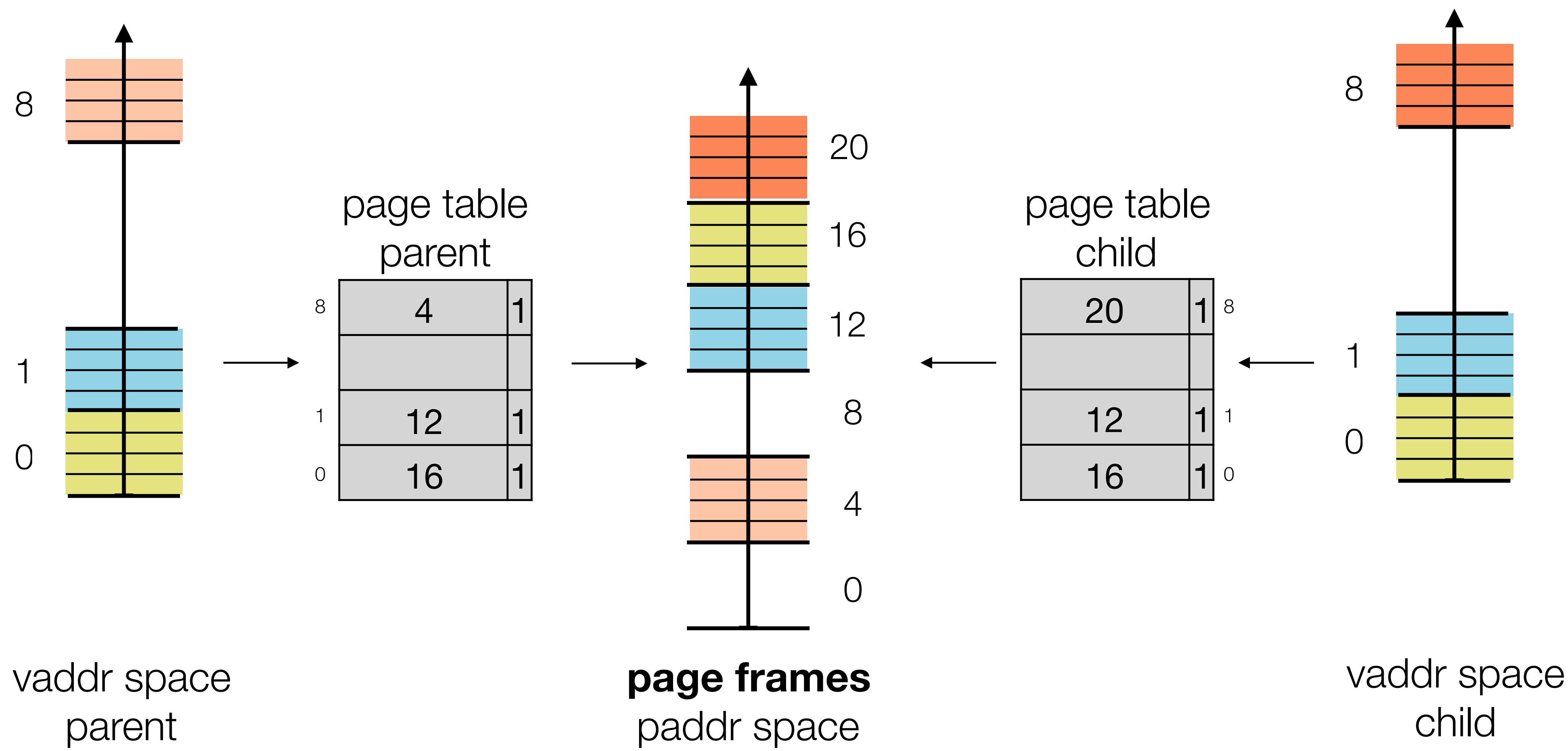
Threads What is shared / private?

- **Shared**
  - code, data segment and the heap

- **Private** → necessary to be at two **different points of execution** at the same time
  - stack → procedure calls (different procedures + arguments)
  - registers → computation
  - program counter → point of execution (next instruction)



Syscall thread() Shallow Copy



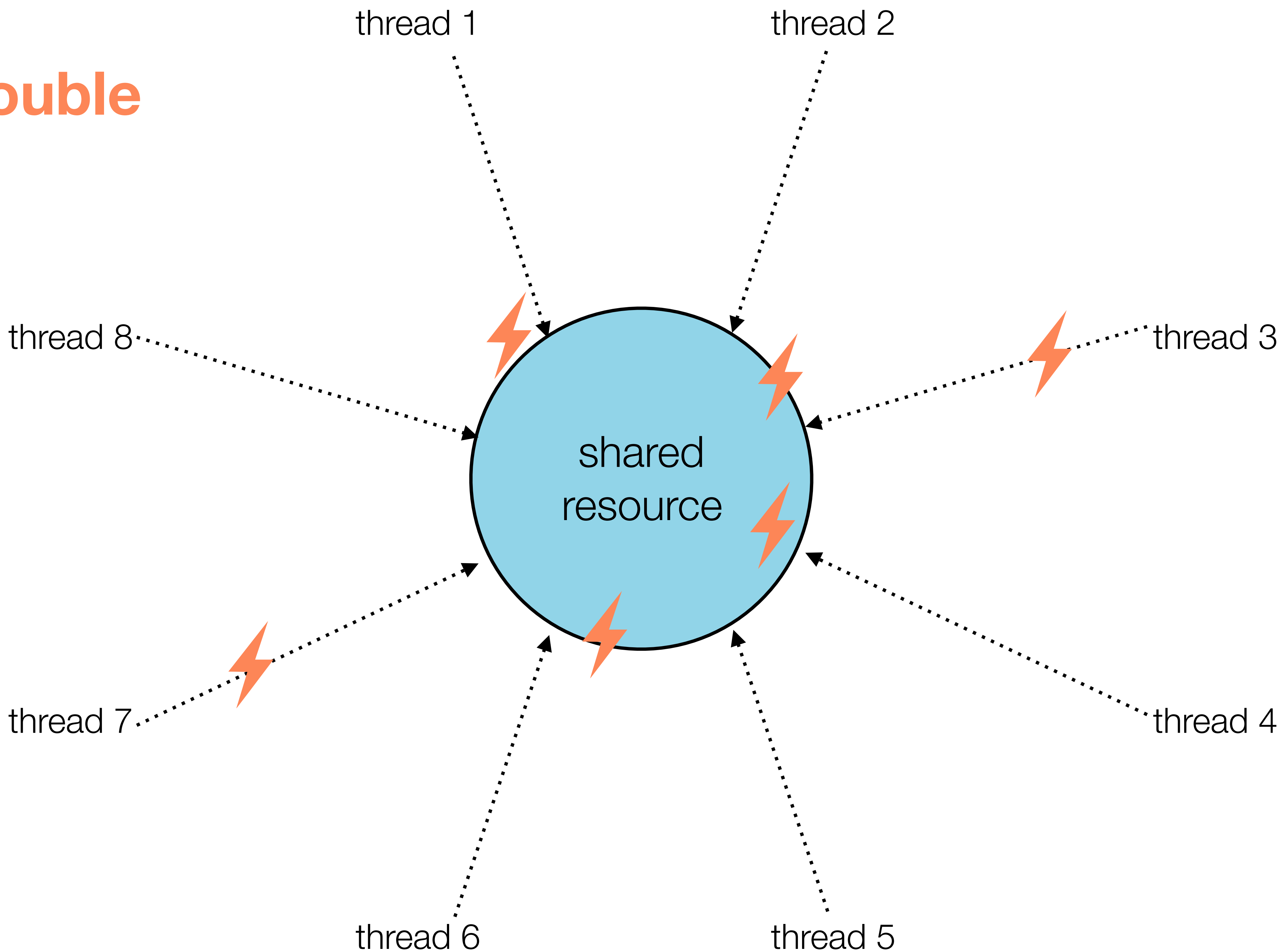
- **Shared memory** → code, data and heap segment
- **Private memory** → stack
  - local variables provide <u>thread local</u> storage

Syscall thread() Implementation Tips

- Use the `fork` as reference
  - shallow copy of lower part memory
  - deep copy of stack
- The operating system provides this service
  - `thread()` syscall → API: `uint64_t thread()`, ABI: syscall number
  - a libc* wrapper function → syscall interface (ABI)
  - handling the syscall → implementation in mipster

Problem Sharing

Trouble



“The OS will not come to rescue this time…”

“Dealing with this problem, is the programmers responsibility…
So let’s see what we are dealing with.”

Race Condition

*Code contains a <u>race condition,</u> if the semantics of that code depends on the **speed** of execution and/or **timing** between threads (**interleaving**).*

- A **race** for CPU time (#instructions a thread executes)
  - behavior of software depends on *how* threads run concurrently
  - nondeterministic
- A race condition is
  - <u>critical</u> iff it determines the final machine state
  - non-critical iff its occurrence has no impact on the final machine state

Race Condition Shared State

- Race conditions can become a problem when shared state is involved
  - threads depend on shared data → critical race condition might occur
  - threads operate on shared date → potential for inconsistent data

- A <u>critical section</u> is a piece of code that accesses a shared resource

```
uint64_t x;

uint64_main() {
   x = 0;

   thread();

   x = x + 1;

   return x;
}
```

critical section

`x` is shared
and 2 threads try to
access it concurrently

`x` = ?

Race Condition Shared State and Interleaving

thread 1 thread 2

critical section

```
x = x + 1;

return x;
```

switch after **one** instruction

LD	$t0, -8($gp)	x
LD	$t0, -8($gp)	x
ADDI	$t1, $zr, 1	1
ADDI	$t1, $zr, 1	1
ADD	$t0, $t0, $t1	+
ADD	$t0, $t0, $t1	+
SD	$t0, -8($gp)	=
SD	$t0, -8($gp)	=
LD	$t0, -8($gp)	x+1
LD	$t0, -8($gp)	x+1

both return 1

?

return 1 and 2

switch after **four** instruction

LD	$t0, -8($gp)	x
ADDI	$t1, $zr, 1	1
ADD	$t0, $t0, $t1	+
SD	$t0, -8($gp)	=
LD	$t0, -8($gp)	x+1
ADDI	$t1, $zr, 1	1
ADD	$t0, $t0, $t1	+
SD	$t0, -8($gp)	=
LD	$t0, -8($gp)	x+1+1
LD	$t0, -8($gp)	x+1+1

both return 2

“So which of the 2
…or actually any other option…
is correct?”

“Not a trivial question - what should the semantics be?
We absolutely don’t want:

'IT DEPENDS’

Both executions should count - sounds reasonable..!?”

2, Please!

- **Desired semantics**
  - both executions should count
  - execution should always produce the same result
- **Ensuring it**
  - different options, but all based on same principle → All-Or-Nothing (Atomicity)
- Execute a **critical section** all at once or not at all
  - critical operations must be fully completed
  - hence only one thread can be in the critical section → **Mutual Exclusion**

Mutual Exclusion

*<u>Mutual Exclusion</u> is the **requirement, policy** that at most one thread can enter and be in a critical section.*

- Prevents concurrent access to shared resources
- Enforced by hardware and/or software
  - disable interrupts (HW)
  - atomic instructions (HW)
  - locking (SW)

Mutual Exclusion Hardware Support

- **Disable interrupts** (context switching) while in critical section
  - not fault tolerant
  - not a solution on multicore machines
- **Atomic instructions** - make it one operation
  - an atomic operation
    - completes in one step
    - can not be interrupted
    - done by one thread
  - simultaneously read and change/write
- **Selfie**
  - syscalls are mutually exclusive by design
    --> syscall handling disables the timer interrupt → `TIMESLICE`

Atomic Instructions Test-and-Set(TS)

atomic in hardware

```
test_and_set(addr) {

   old_value = *addr;
   *addr = 1;

   return old_value;
}
```

- Can be used to implement locks 0 → 1
  - return value **0** indicates **success**

Atomic Instructions Compare-and-Swap(CAS)

atomic in hardware

```
cas(addr, old_value, new_value) {

   if(*addr == old_value) {
      *addr = new_value;

      return 1;
   }

   return 0;
}
```

- More general than TS
- Upon **success 1** is returned
  - other implementations may define this differently

Locking

- Spinlock → memory location
  - free (0) or held (1)
  - **loop** until lock is free and can be acquired (0 → 1)
  - efficient if held for short timespan
- Blocking lock → queue
  - threads wait in queue
  - 1 OS thread keeps checking the look on behalf of queued threads
  - better when lock held long (>1000 instr.)
- Disadvantage → not fault tolerant
  - deadlock, lock contention,…

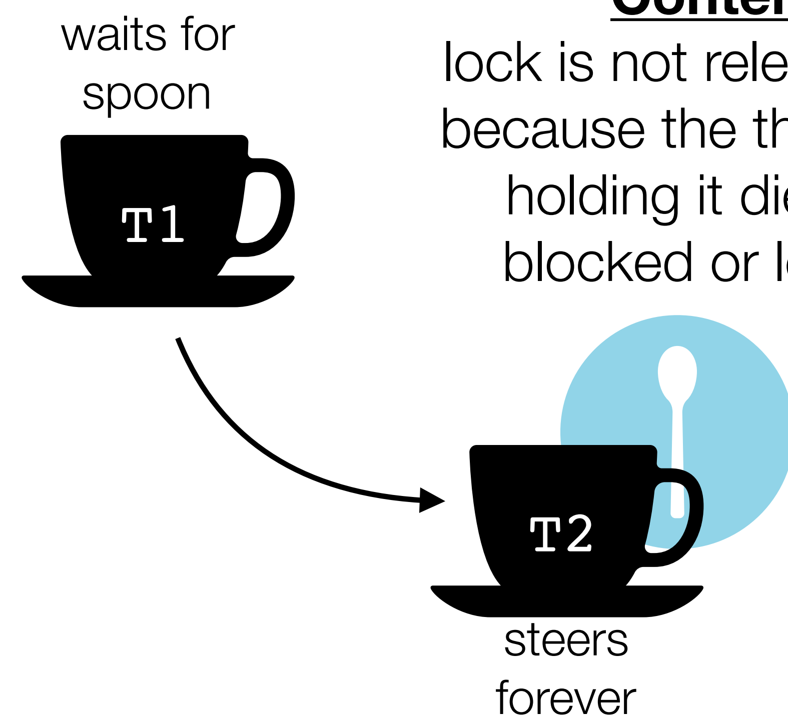
Locking

Locking must be **atomic**
(race condition)

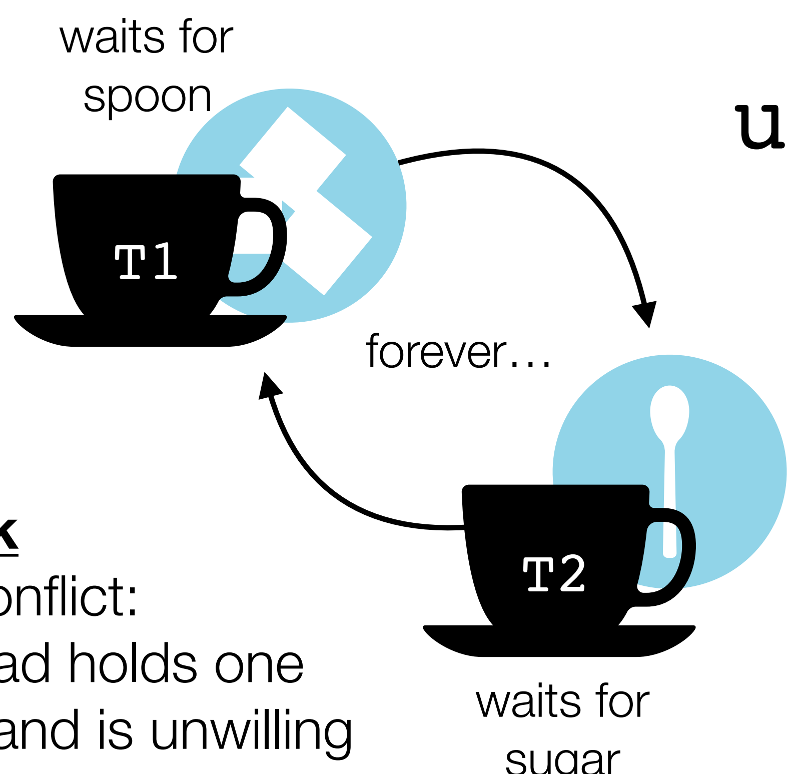
```
lock();
```

critical section

```
unlock();
```



<u>Contention</u>
lock is not released because the thread holding it dies, is blocked or loops



<u>Deadlock</u>
circular conflict:
each thread holds one resource and is unwilling to release it.

Famous example
<u>dining philosophers</u>

Mutual Exclusion Implementation Tips

- See lock implementation tips
- Implement `lock()`, `test_and_sat()`, `compare_and_swap()` as syscalls
  - syscalls in selfie are atomic (no timer interrupt `TIMESLICE`)
- The operating system provides this service
  - syscall → API and ABI
  - a libc* wrapper function → syscall interface (ABI)
  - handling the syscall → implementation in mipster

Summary Threads Part 1

- Additional effort is necessary to allow communication between **processes**
- The OS provides **another process model**, that make communication easier
  - **threads** are light-weight processes that share parts of memory through which they communicate more easily
  - BUT new **problems** arise when shared resources come into play
- Concepts and mechanisms used
  - **race condition** → critical or non-critical
  - **mutual exclusion/atomicity** → disable interrupts, atomic instructions, locking

Concurrent Data Structures

> *Threads communicate through shared memory. A <u>concurrent data structure</u> is a way to **organize data in shared memory** for access by multiple threads.*

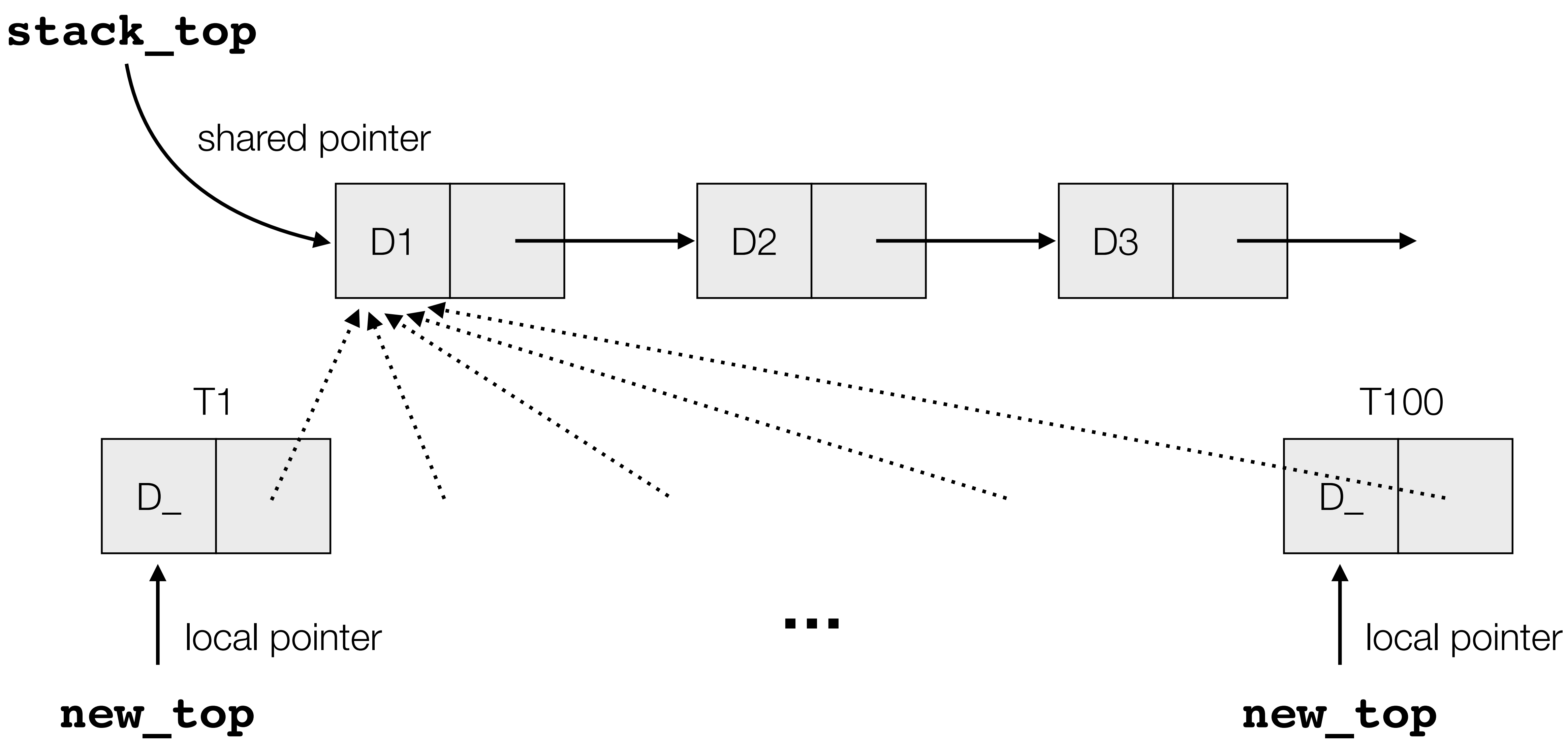
- Concurrent access + interleaving = potentially unexpected outcome
  - mechanisms to ensure 'correct' behavior are required
- Designing and verifying effective concurrent data structures is difficult
  - safety properties → nothing bad happens
  - liveness properties → something good keeps happening
- Blocking or non-blocking implementation
  - non-blocking guarantees → lock free, wait free
- Performance measure is <u>scalability</u> - <u>speedup</u>

Concurrent Data Structures

- Stack
  - singly linked list stack allowing push and pop
  - Treiber Stack
  - Time-Stamped Stack
- Queue
  - Michael-Scott-Queue
- Pools
- …

Concurrent Stack Problem

- 100 threads try to push at the same time → interleaving



Concurrent Stack Treiber Stack

- Shared pointer to top of stack
- Operations **push** and **pop**
- **Push**
  1. create new element
  2. set next-pointer to top element - thread local
  3. set shared top pointer to new element

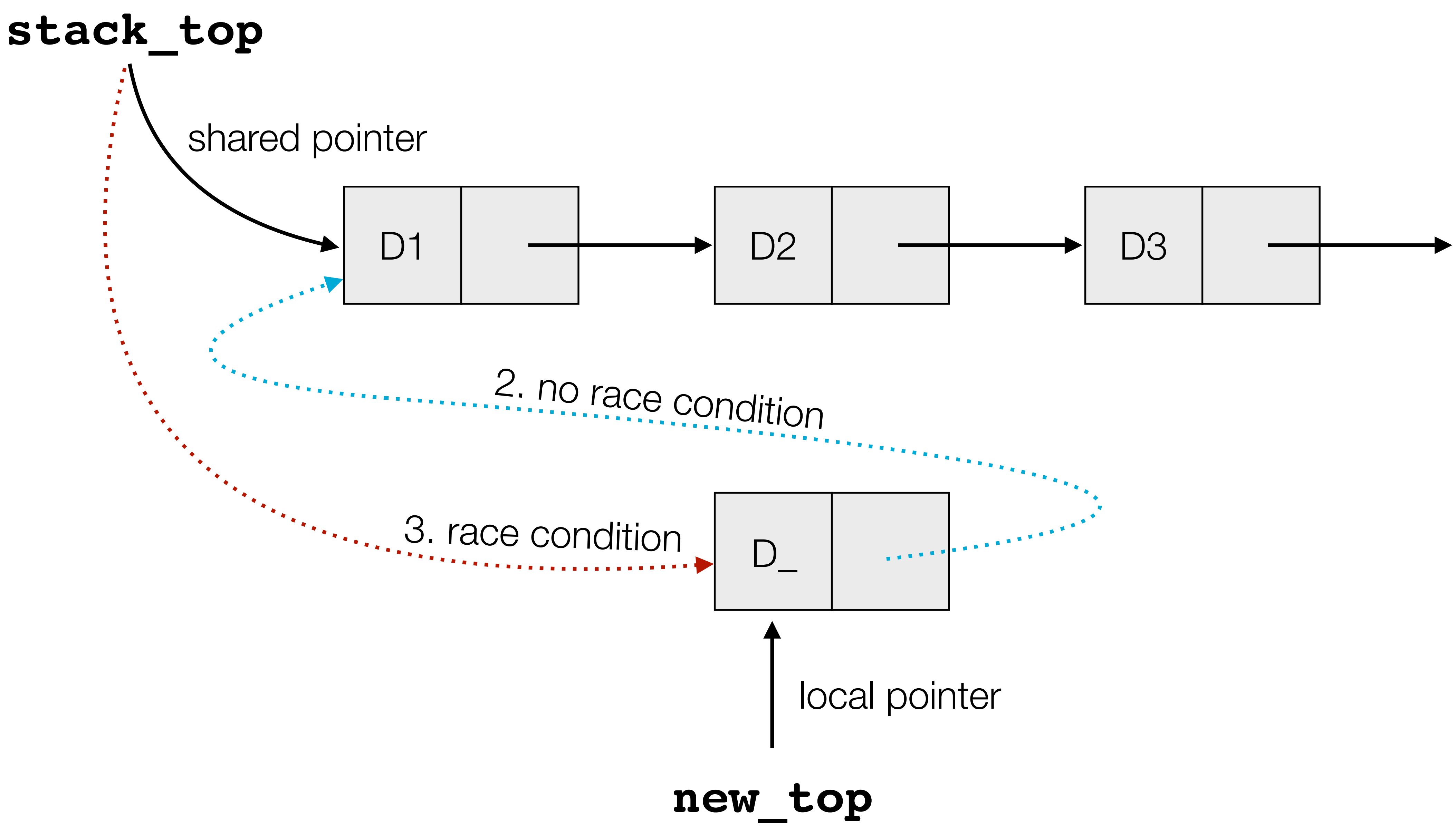
  DO NOT SPLIT

  critical section
- **Pop**
  1. grab top and second to top element - thread local
  2. set shared top pointer to second to top element
  3. return top element

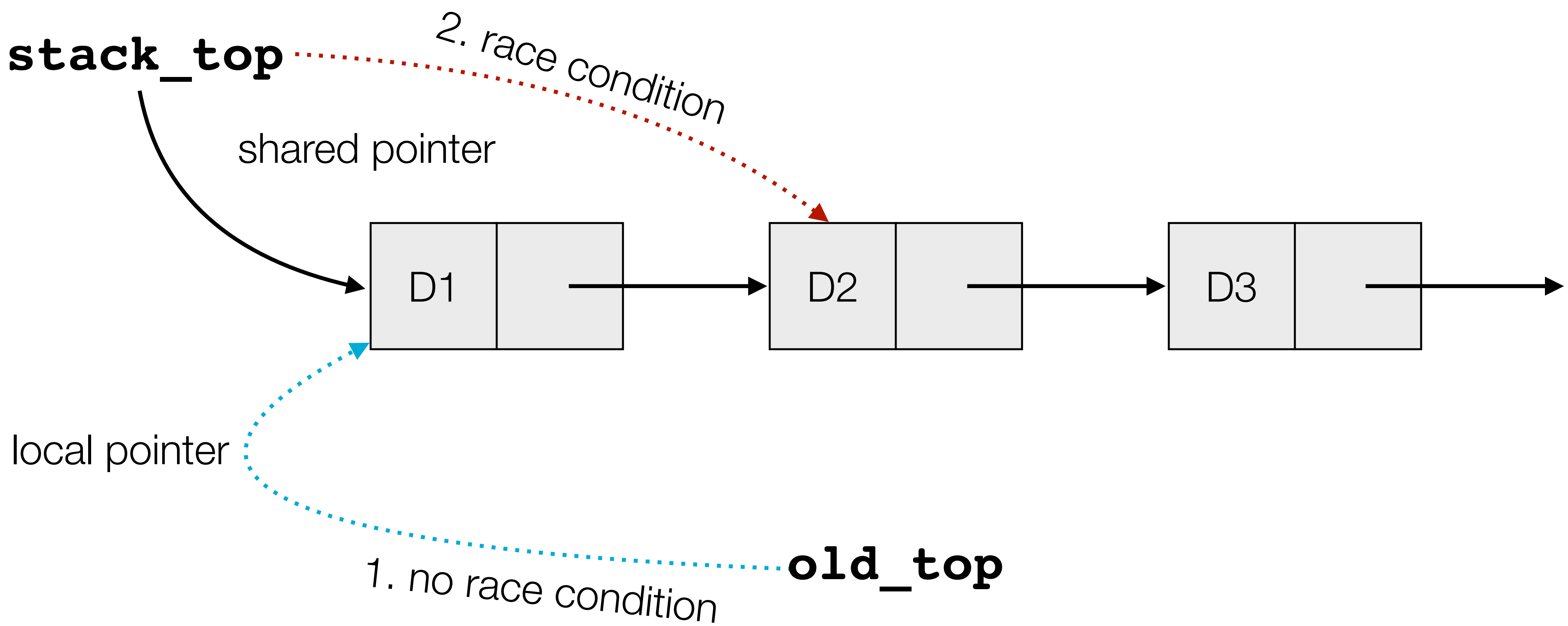
  DO NOT SPLIT

  critical section
- We need atomicity → one single moment in which push/pop takes effect (<u>linearizability</u>)

Concurrent Stack Push



Concurrent Stack Pop



“We can always just lock push and pop and risk getting stuck in an infinite loop…mmh

…maybe we can do better and somehow use atomic instructions?”

“Yes, there is another way…you are talking about non-blocking implementations.

They guarantee lock-freedom.”

Lock-Free != Lock Free

- NOT the absence of locks
  - it is not a property of code
- Look-freedom is the **guarantee of overall progress** (system-wide)
  - at least one thread succeeds in finitely many steps (progresses)
  - no guarantees for other threads
  - ex. 100 threads push/pop → 1 will succeed, 99 may loose
- A stronger guarantee is wait-freedom
  - every thread is able to make progress in finitely many steps (per-thread progress)

Lock-Free Implementation Push

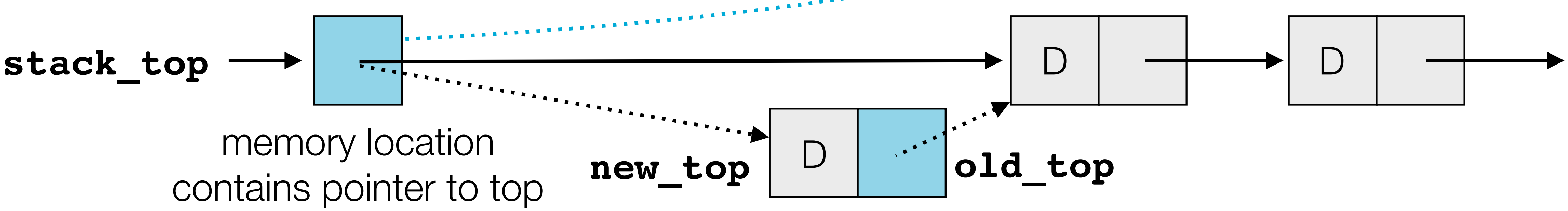
```
void push(uint64_t number) {

   uint64_t* new_top;
   uint64_t* old_top;

   new_top = new_node(number);
   old_top = *stack_top;
   new_top.next = old_top;

   while (cas(stack_top, old_top, new_top) == 0) {

      old_top = *stack_top;
      new_top.next = old_top;
   }
}
```

prepare new node (thread local)

compare **pointer**

ABA?!

try until successful



memory location contains pointer to top

Lock-Free Implementation Pop

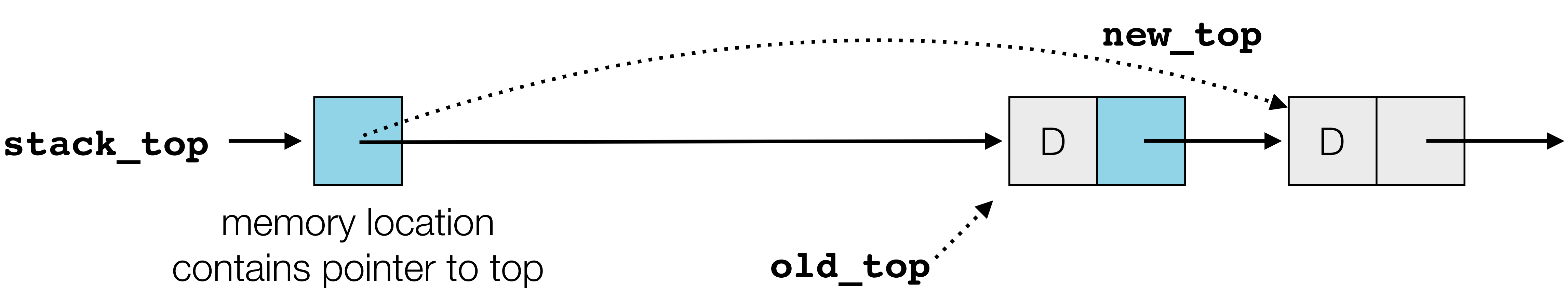
```
uint64_t pop() {

   uint64_t* new_top;
   uint64_t* old_top;

   old_top = *stack_top;
   new_top = old_top.next;

   while (cas(stack_top, old_top, new_top) == 0) {

      old_top = *stack_top;
      new_top = old_top.next;
   }

   return old_top.number;
}
```

grab pointer
(thread local)

compare
pointer

ABA?!

try until successful



“How can we show or argue that this implementation is lock-free?”

“Consider many threads trying to push data onto the stack…

If one thread fails, it’s because the top of the stack changed.
Hence, one thread must have made progress.

A thread failing to make progress is proof of overall progress."

ABA Problem

The problem is this misconception:

is the same ⇒ nothing has changed

Every Monday:
"What's the first
letter of the alphabet?"

*spends weekend on
lonely island*
*cut off from the
outside world…*

On Monday:
"What's the first
letter of the alphabet?"

—— A —— B —— A ——→

time

On Saturday:
It has been decided to make ‘B’
the first letter for today.

is the same $\not\Rightarrow$ ***nothing has changed***

ABA Problem and Pointer

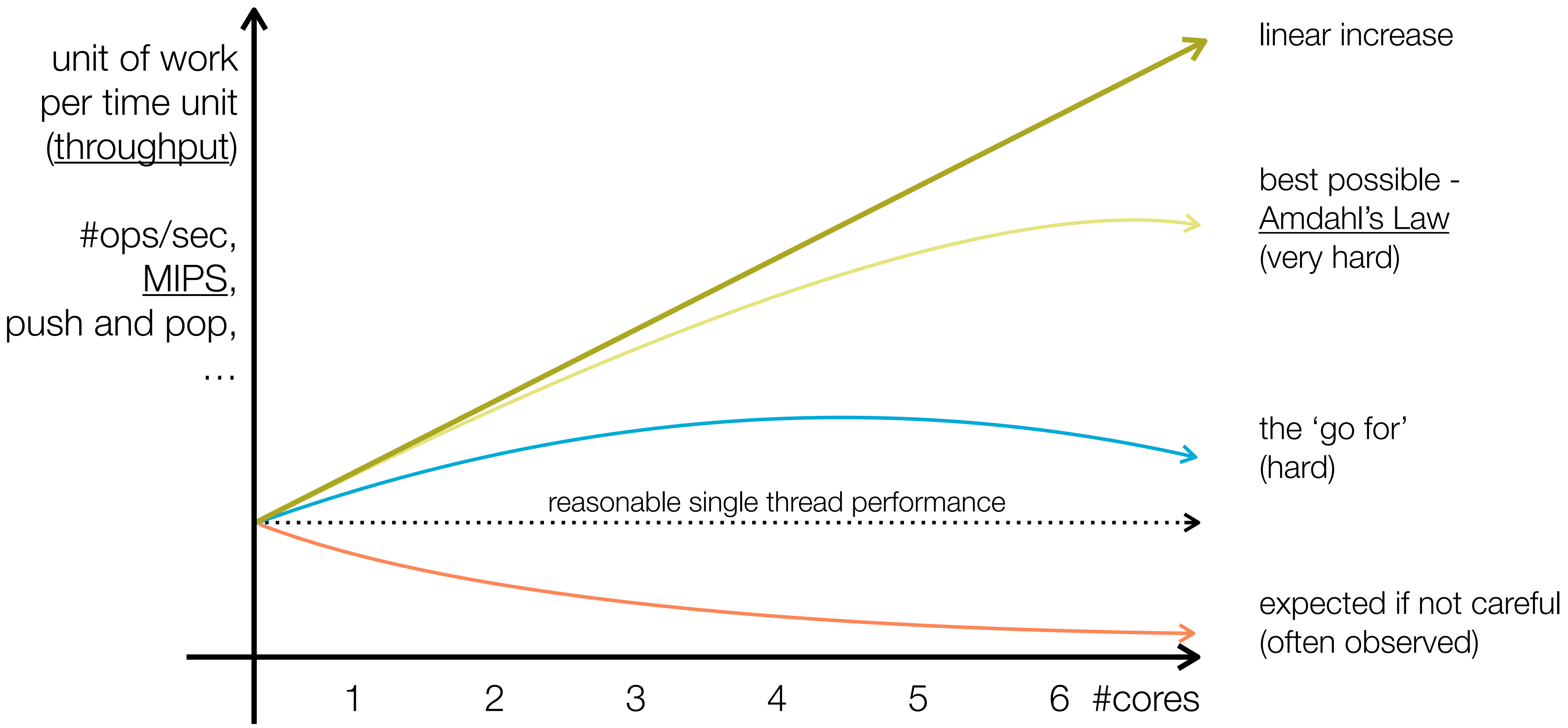
- Careful with deallocating/reusing popped nodes
  - dangling references → other nodes might still have the reference (in `old_top`)
  - first free, then malloc → might return the same pointer again
- **ABA**
  - the pointer is the same ⇒ nothing has changed behind it
- **Solution**
  - 100%: do not reuse pointer as long as other threads have a reference or
  - <100%: tag with version number
    - same pointer different version number (be aware of wrap-around)

“So far we were talking about concurrent execution on a single-core machine.

For this last part we are looking at multi-core machines.”

“To get some performance advantages…?”

Performance Bottleneck

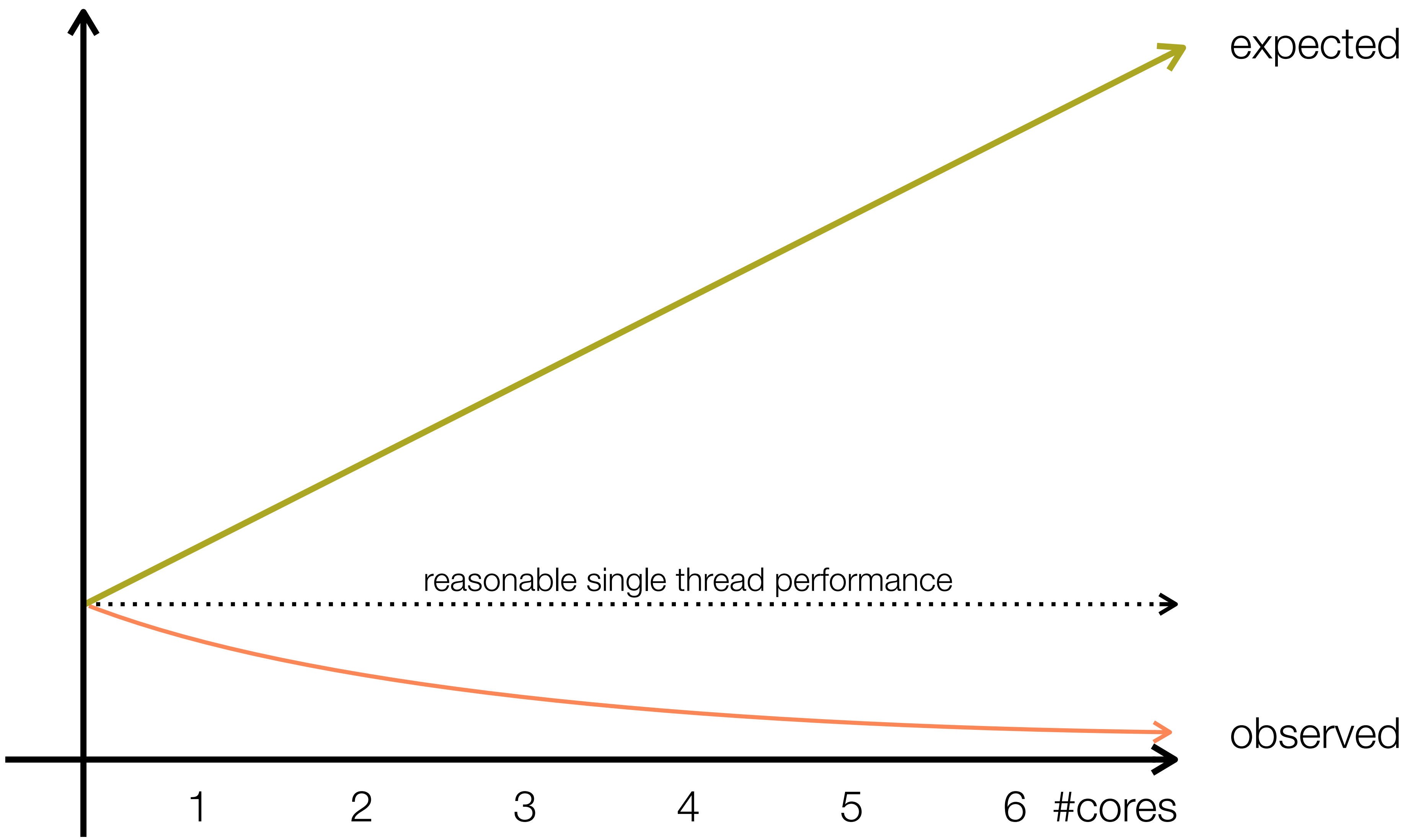


Amdahl’s Law

- Amdahl’s Law gives upper bound for **theoretical** speedup
  - always limited by parts of program that cannot be parallelized
- Distinguishes two parts of code
  - code that can execute potentially in parallel (< 100%)
  - code that **must** execute sequentially (> 0%)
- Parallel vs Sequential
  - property of code **as it executes**, not in the code
  - **indirectly** sequential due to memory layout (false sharing,…)

Performance Bottleneck

- Assume many threads executing on its own → **nothing is shared**
  - we **expect** linear speedup
  - we **observe** negative scalability
- To explain this we have to look closely at the machine

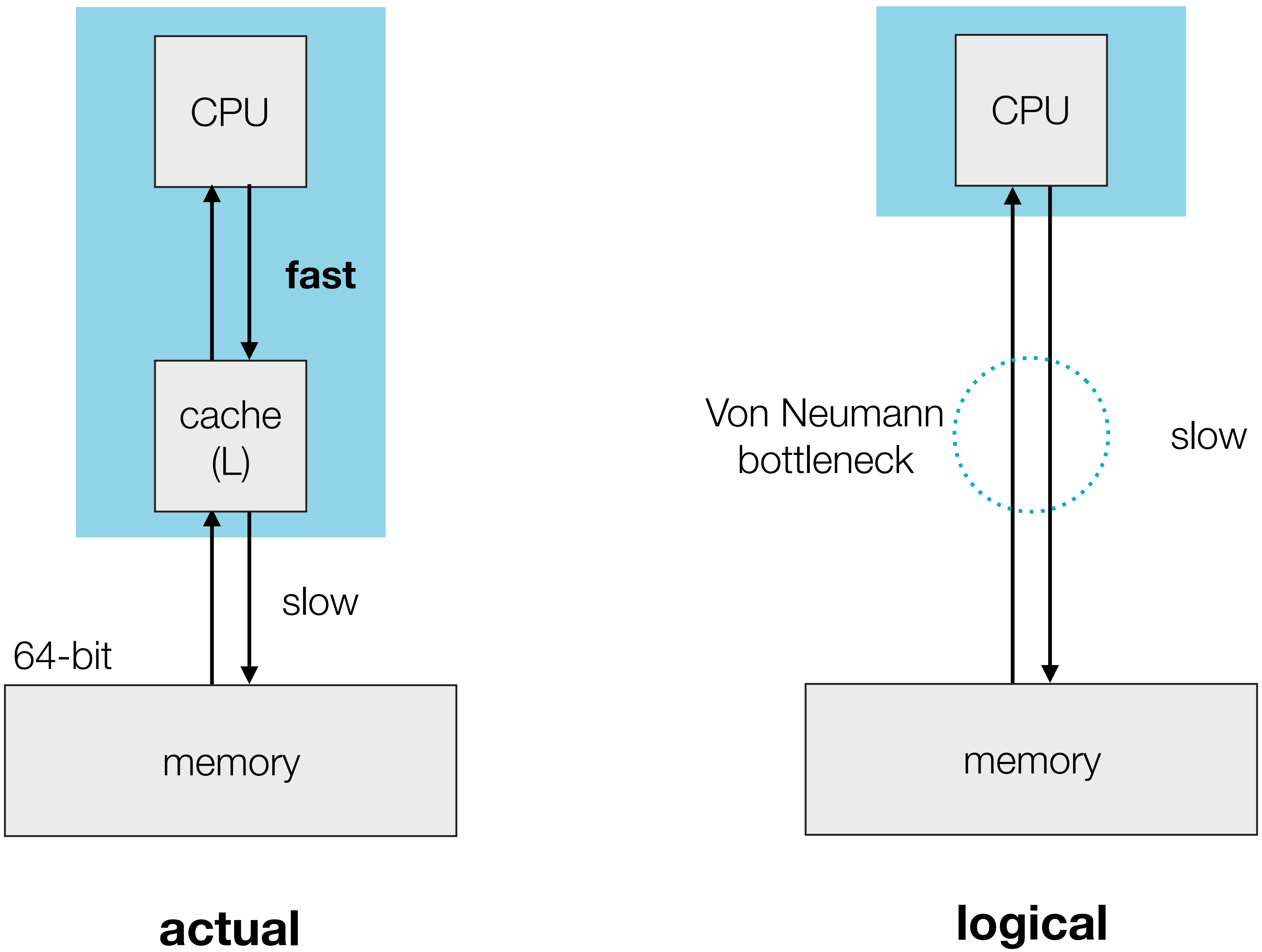


Cache

- **Von Neumann architecture**
  - data and instructions go from memory to CPU
  - Von Neumann bottleneck = bus
- **We know**
  - more reads than writes/stores
  - code shows temporal and spacial locality
- **Exploit this behavior**
  - put another layer of memory between main memory and CPU which has a **fast connection** → cache

Cache

- Cache should be **logically invisible**



Cache Read

- Search the cache for value → is it caching the requested address? (still faster than main memory access)

- **Cache miss**

    - value not currently in the cache

    - push down-read to main memory → expensive

- **Cache hit**

    - value found

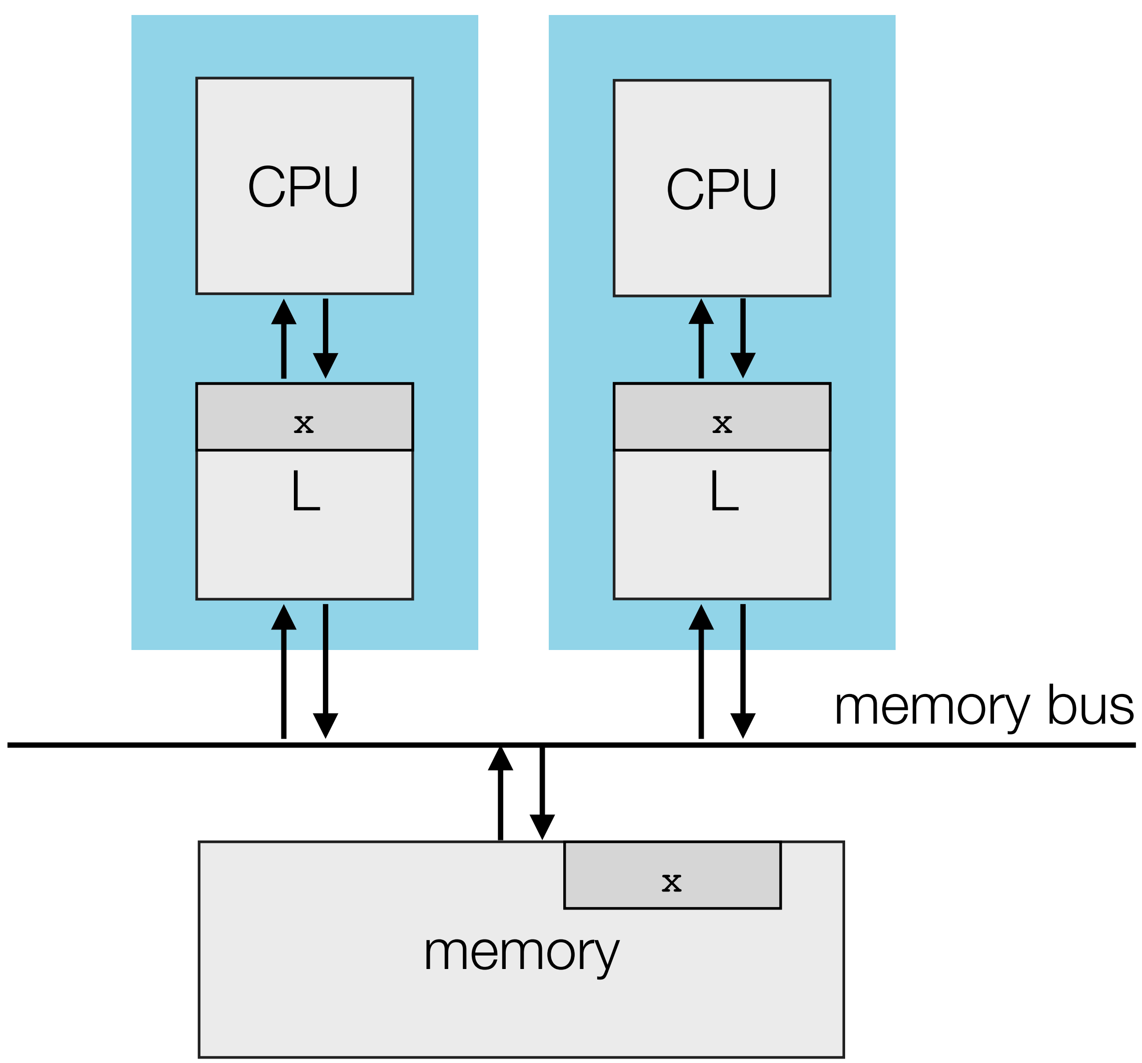
    - no activity on slow memory line → cheap

Cache Write

- Write the value into cache
- It might never get written into main memory
- **Write-down**
  - when address is to be read or written but cache is full
  - cache eviction → find a slot that can be made available (as with swapping)
  - write evicted value back iff it changed
  - the goal is to **avoid writing** to memory as long as possible

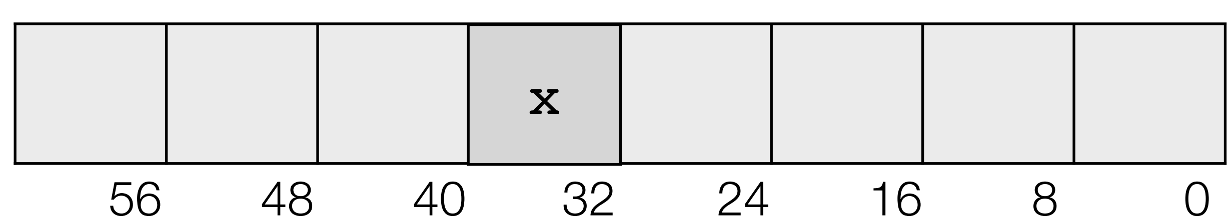
Problem with Multiple Cores

- One thread keeps updating the `x` in its cache
- Another thread wants to read `x`
  - sees `x` is 'old'
  - force write-down (cache invalidation)



Problem with Multiple Cores

- Reads are expensive → if necessary why not make the most of it?
- Therefore cache **takes more** than the 8 byte that are requested
  - takes the whole neighborhood
  - 64 byte (8 machine words) = <u>cache line</u>
- It exploits spacial locality next requested address is in the neighborhood



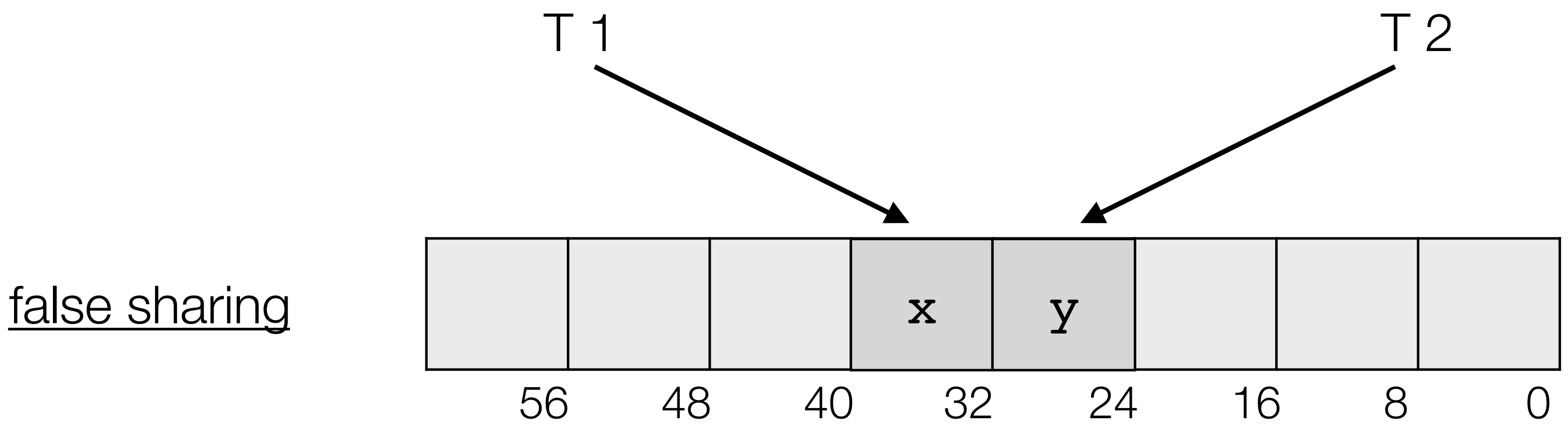
cache line

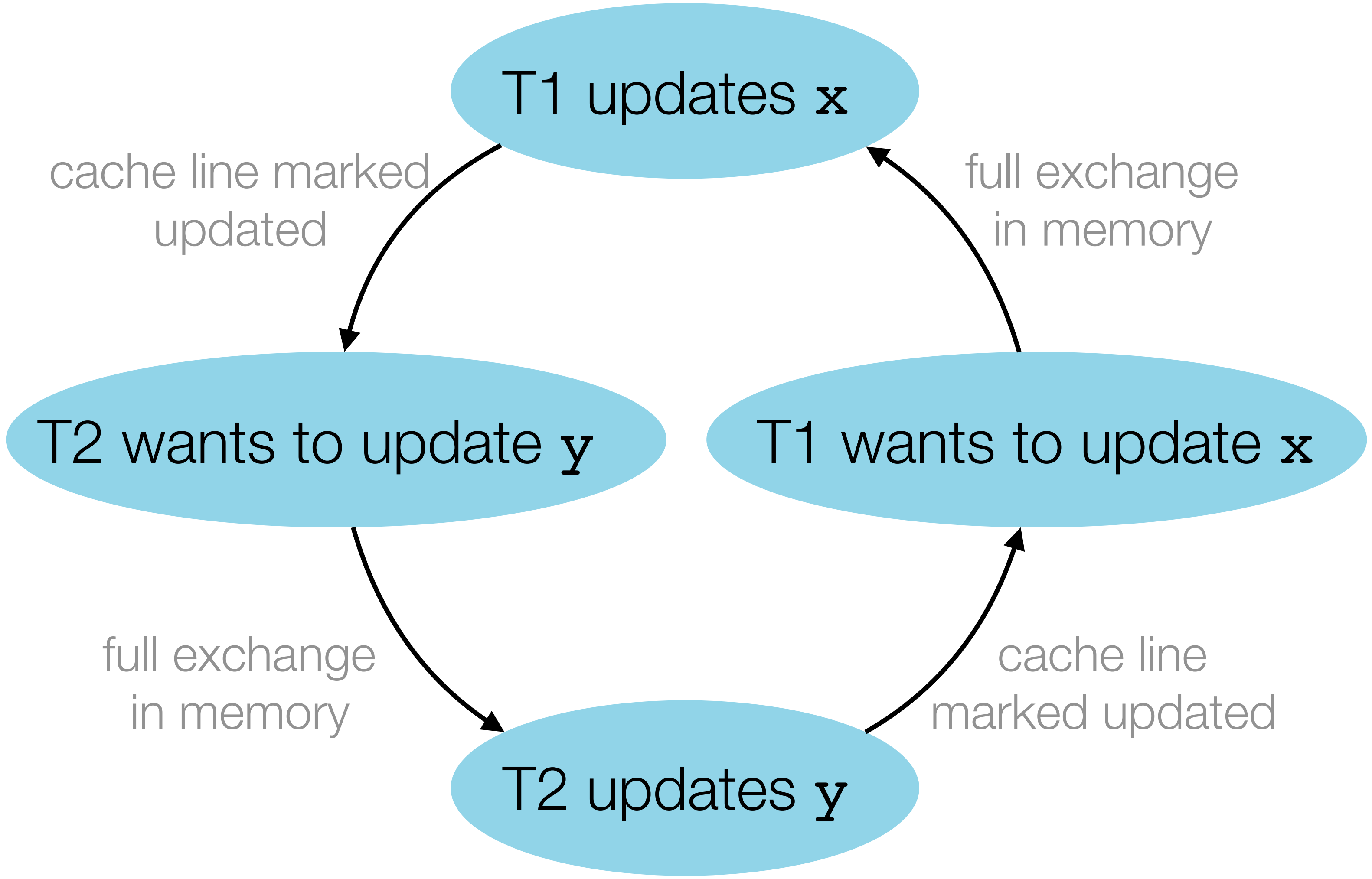
“I think I know what’s happening…

The threads don’t share anything, but they actually share a cache line?”

“That’s exactly what happens, the problem we observe here is false sharing.”

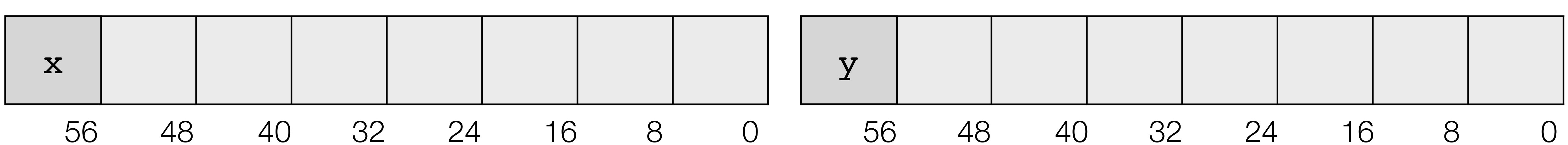
False Sharing





False Sharing

- **Logically**
  - threads do not share any resources
  - ex. each uses its own global counter
- **On the hardware level**
  - threads access data that **physically** lies belongs the same cache line
- **Solution**
  - layout → making sure `x` and `y` are sufficiently far apart in memory
  - in direct competition to spacial locality



Summary Threads Part 2

- Threads are a process model that make communication easier
  - using threads introduces new problems the **programmer** has to be aware of and has to take care of

- **Concurrent data structures** can be used to organize data in shared memory
  - accessed by multiple threads (efficient, save and correct)

- Linear speedup is impossible - not 100% of the program can be parallelized

- Concepts and mechanisms used
  - **blocking and non-blocking** implementation of concurrent data structures
  - **progress guarantees** → lock-free, wait-free
  - **ABA problem** → reuse of pointers
  - **sequential / parallel code** → property of code as it executes, indirectly sequential
  - **false sharing**→ memory layout effects performance (requires sequential execution)

Index Compiler

Index Compiler

Index Compiler

Index Operating Systems

Index Operating Systems

Index Operating Systems