Introduction to

Operating Systems

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.

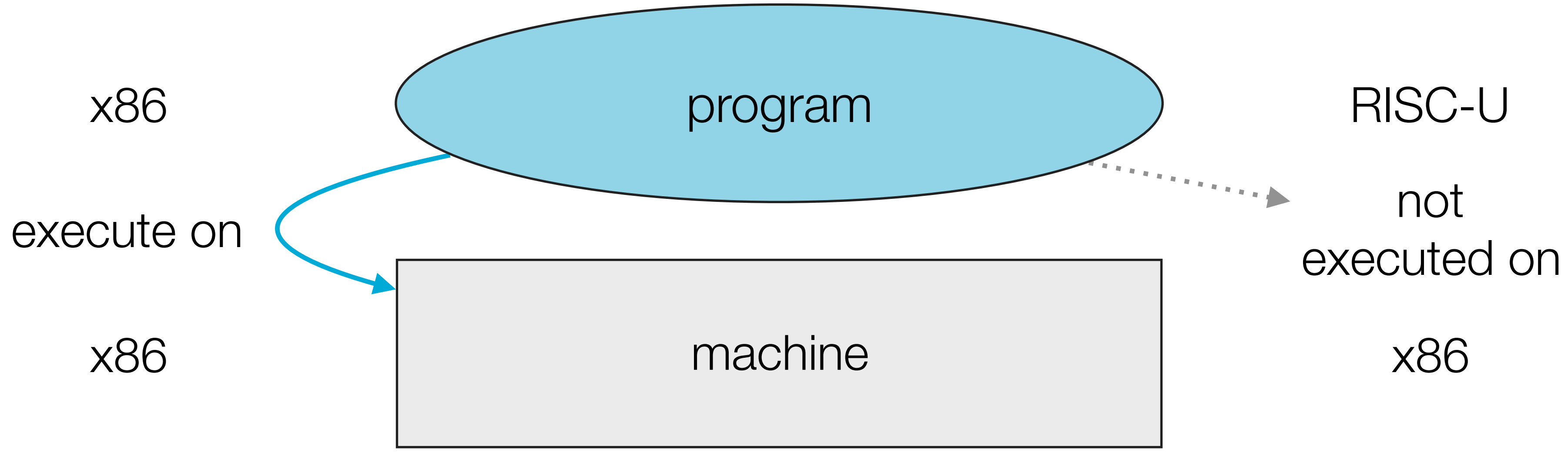
- Now we want to **execute** this compiled program.

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 instruction set architecture (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.



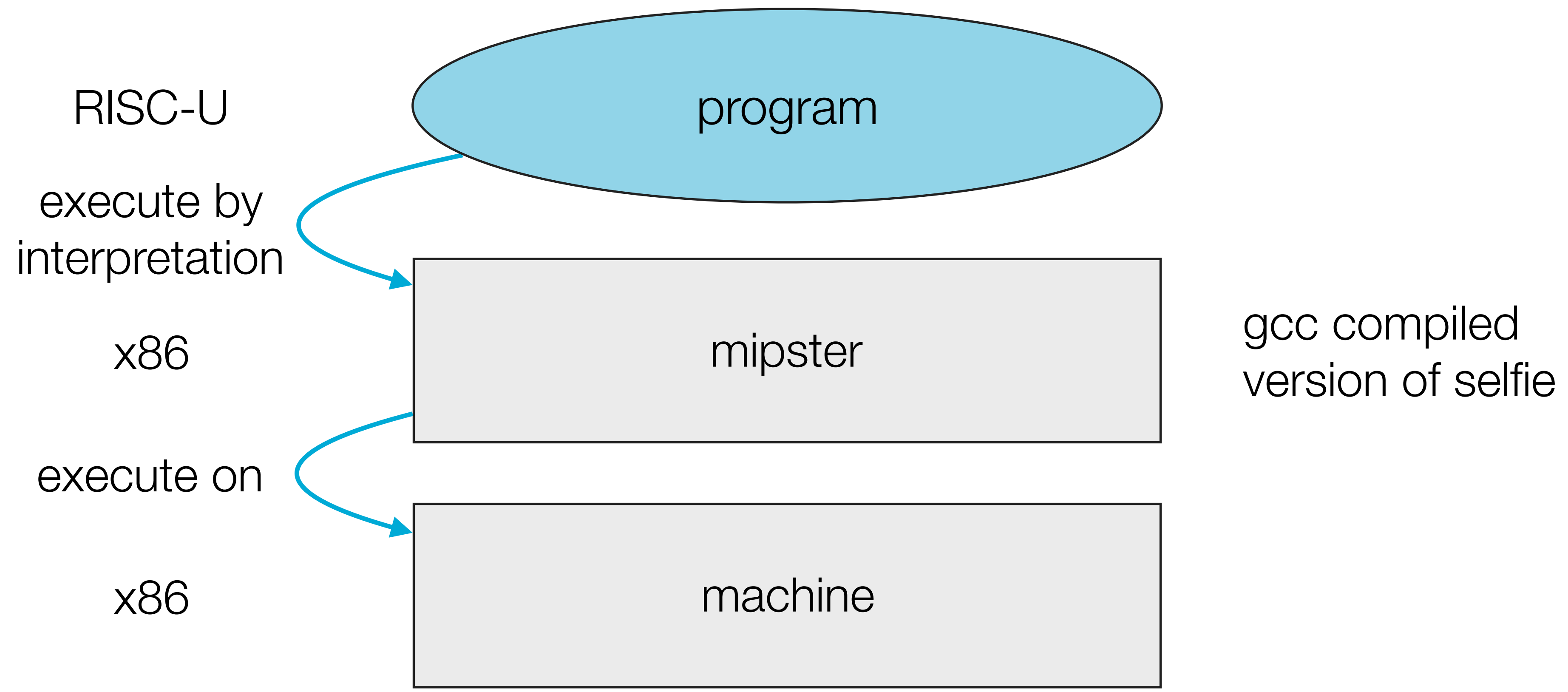
Mipster

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

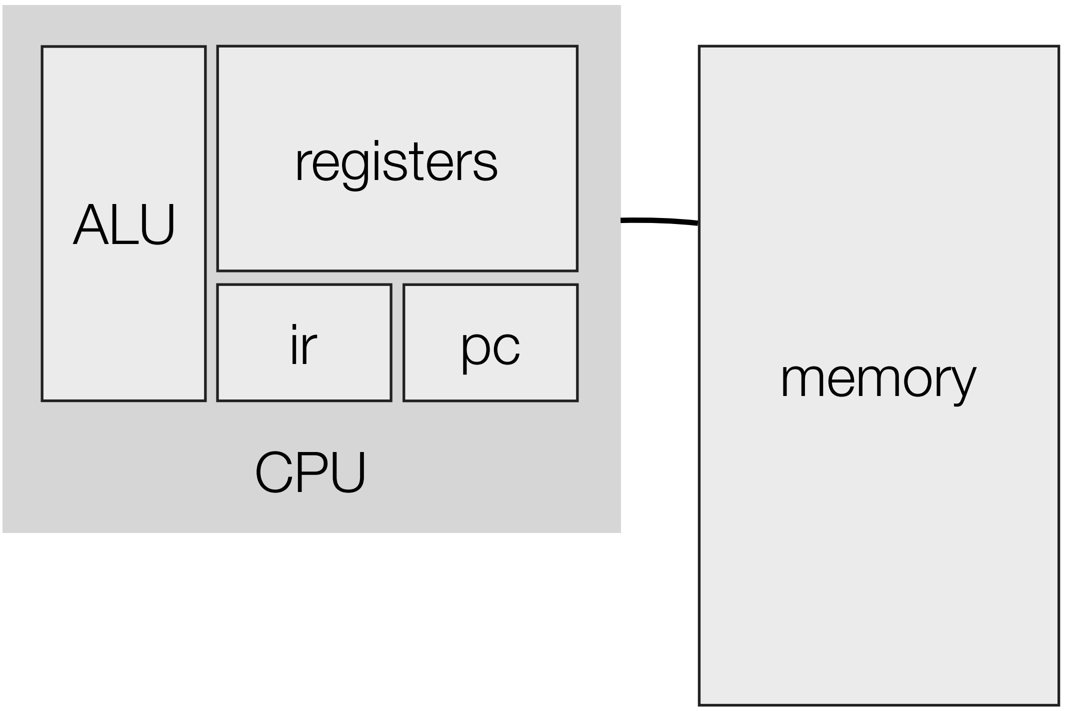
- To understand how mipster works we take a look at the machine it emulates.



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 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
```

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

- **flags a**

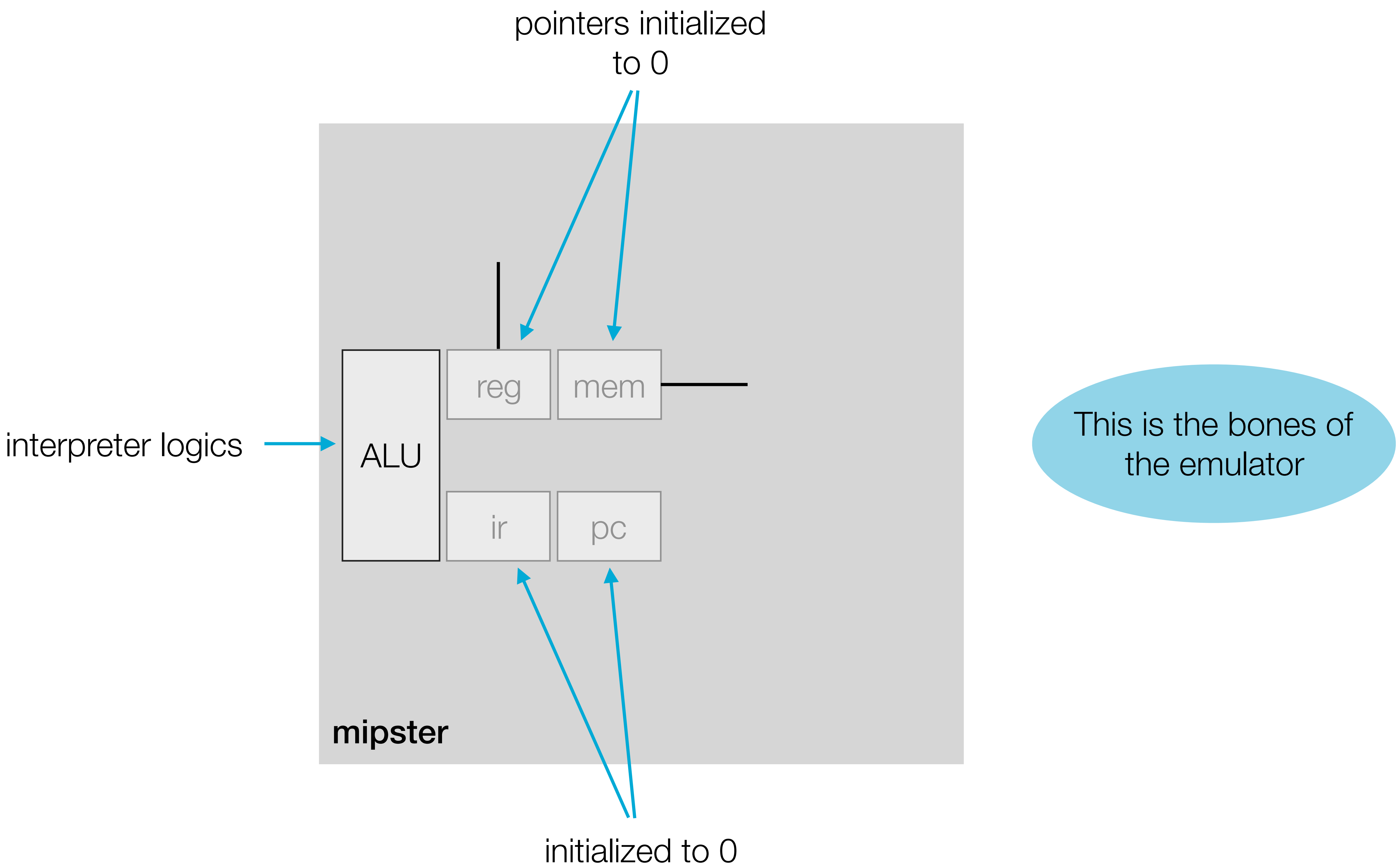
- **initialize memory size of machine** → global variable
  - `init_memory(atoi(peek_argument()))`

MIPSTER

- **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)

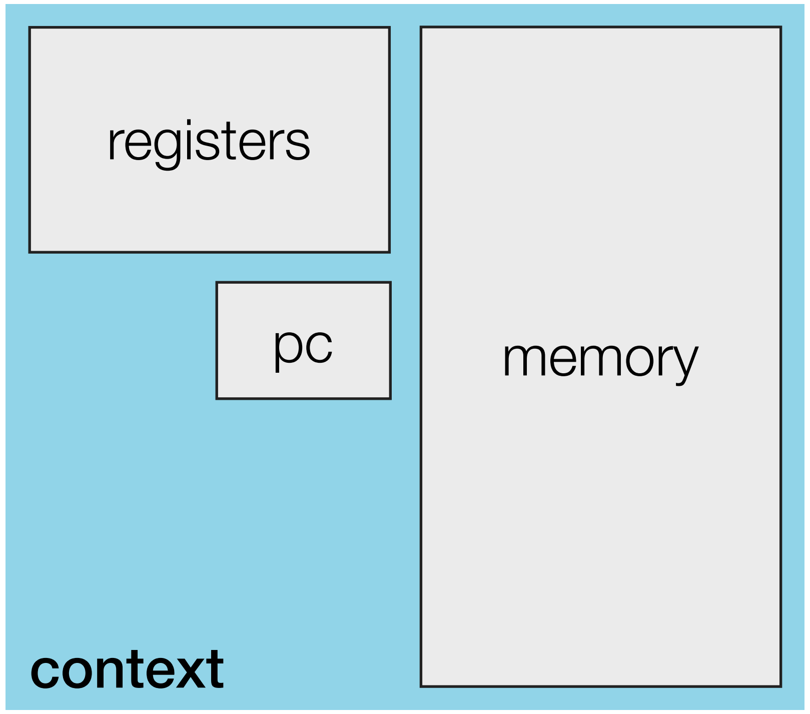
MIPSTER



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)

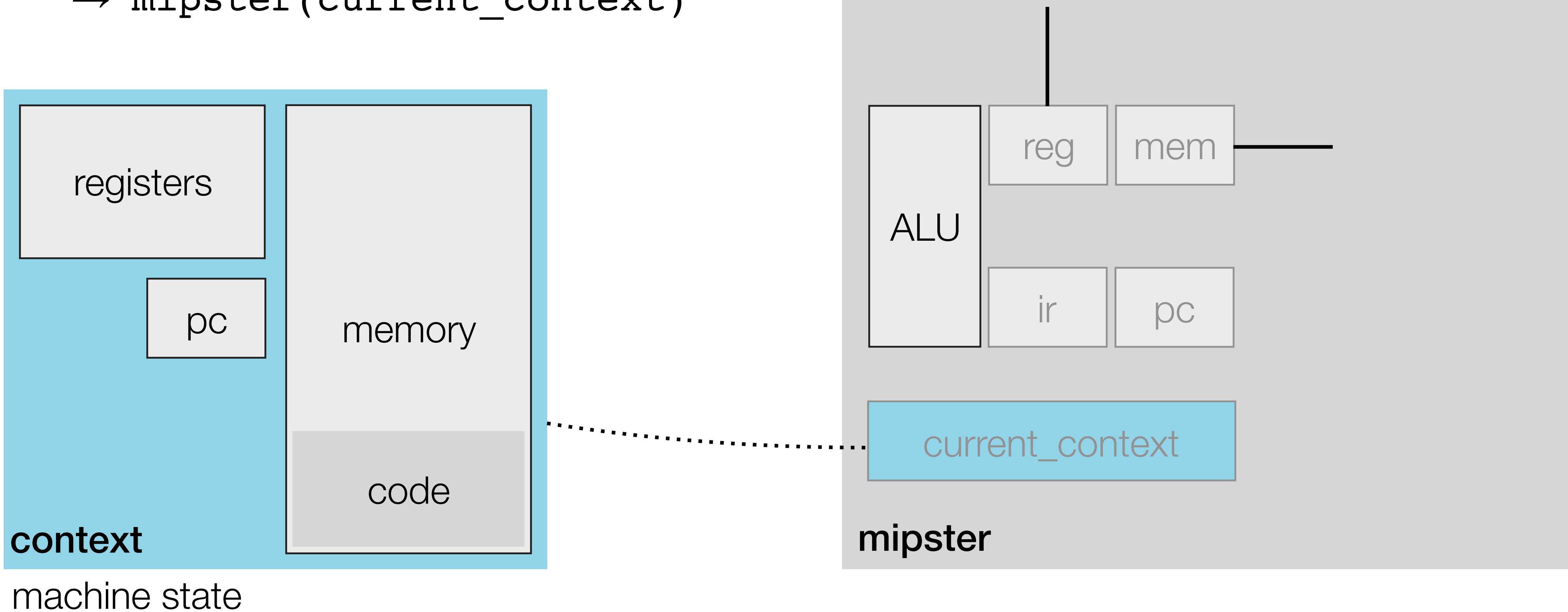
`./selfie -c`

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)

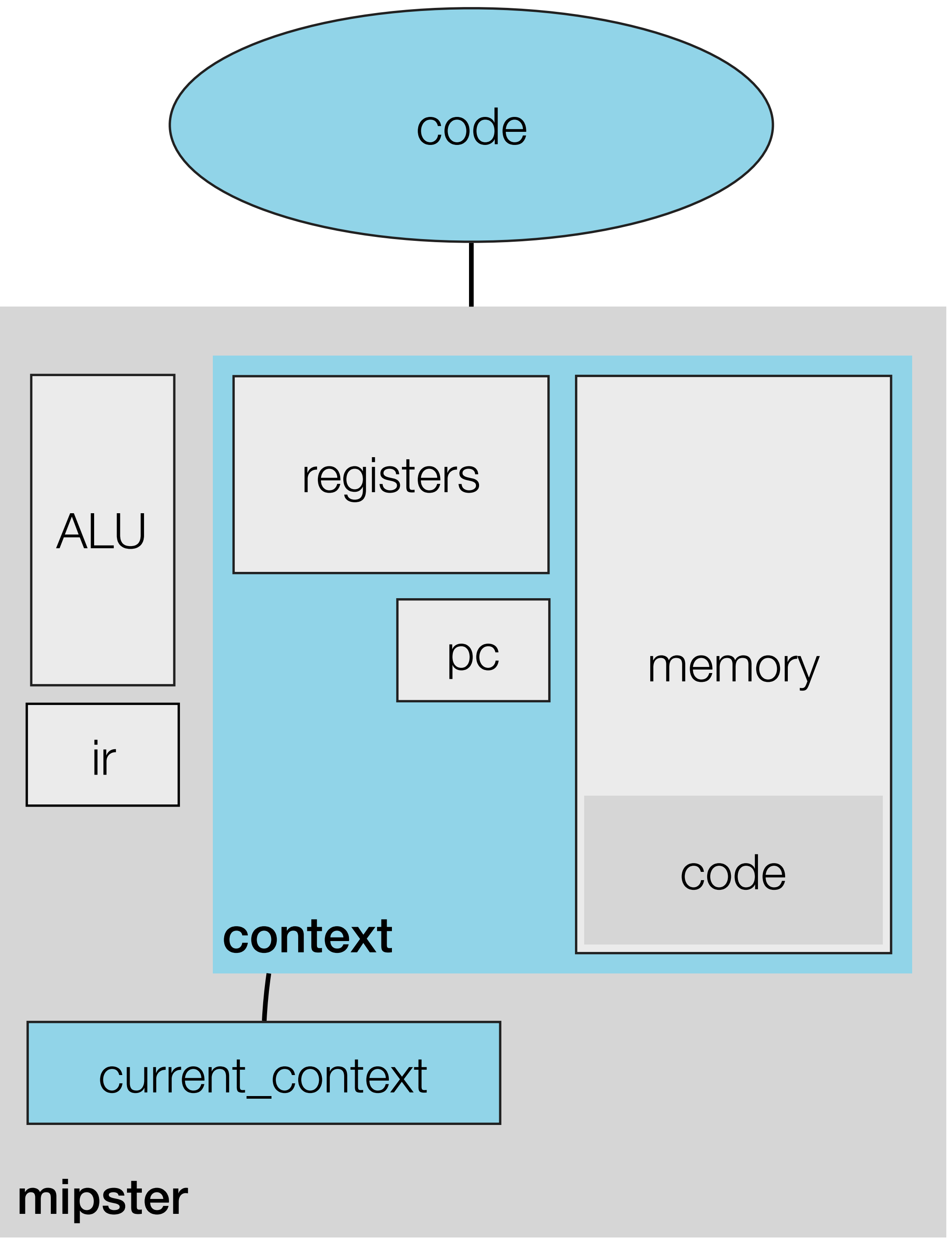
- so far we have
  - an new and "empty" **machine**
  - a **context** representing part of the machine state
- now we need to bring them together
  - let mipster “execute the context”
    → `mipster(current_context)`



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

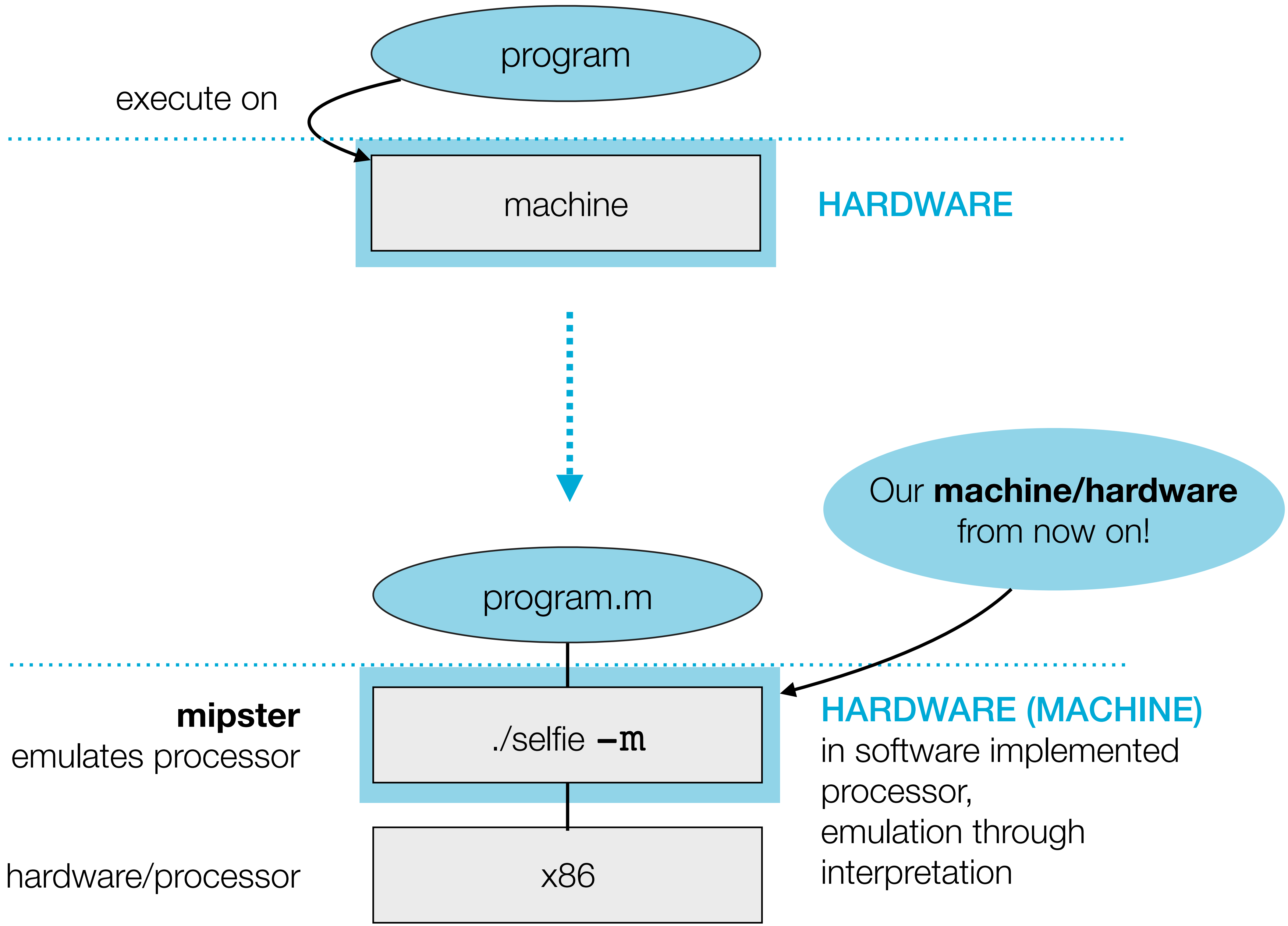
renew **TIMESLICE** and switch back to context

```
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



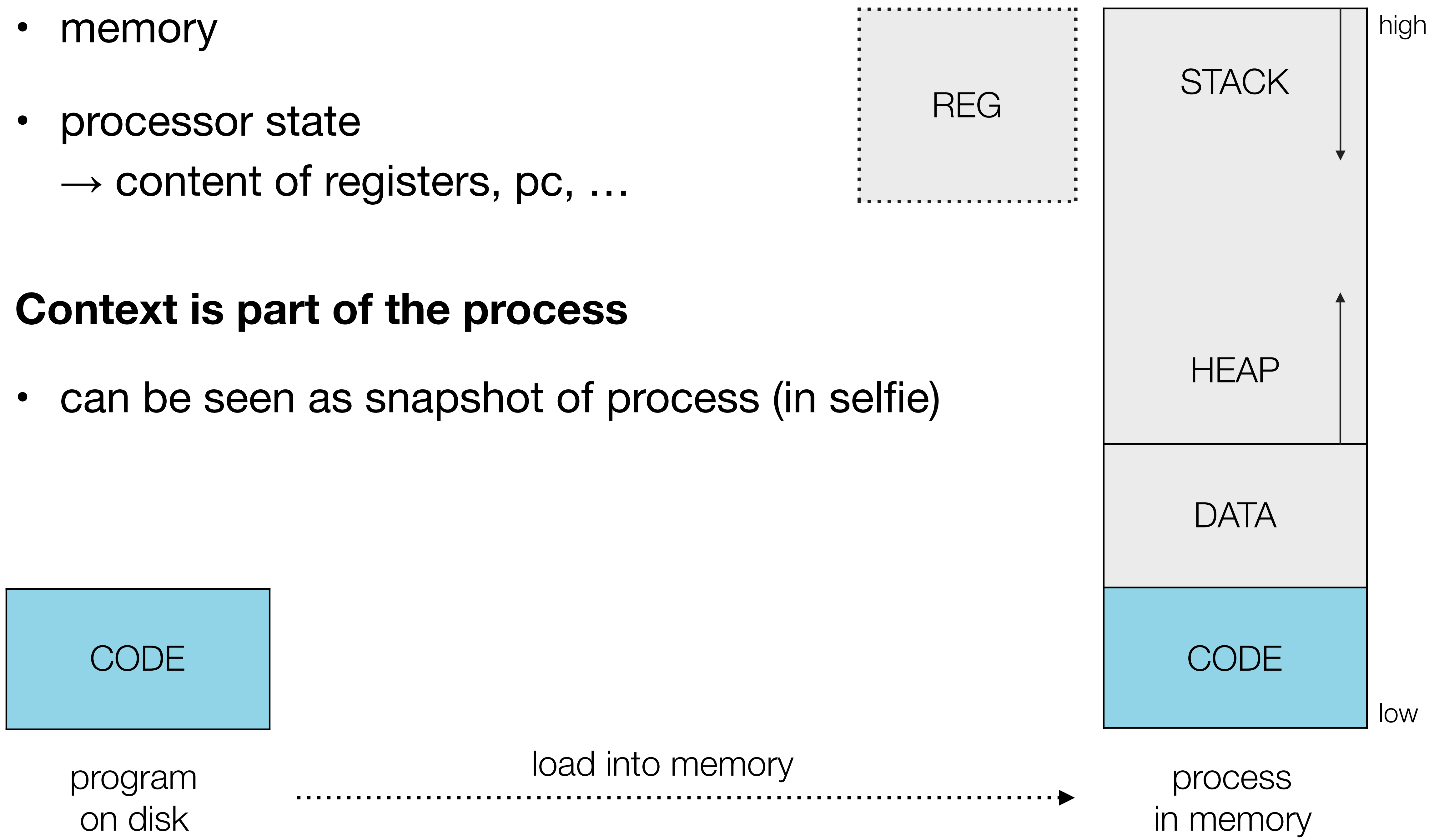
Process

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)



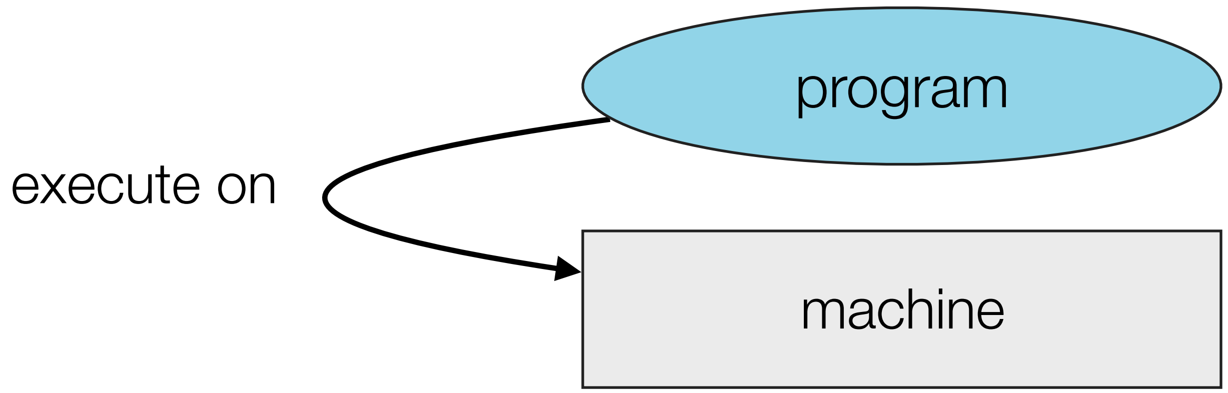
“Now we have enough insight into selfie and mipster to get started with operating systems.”

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

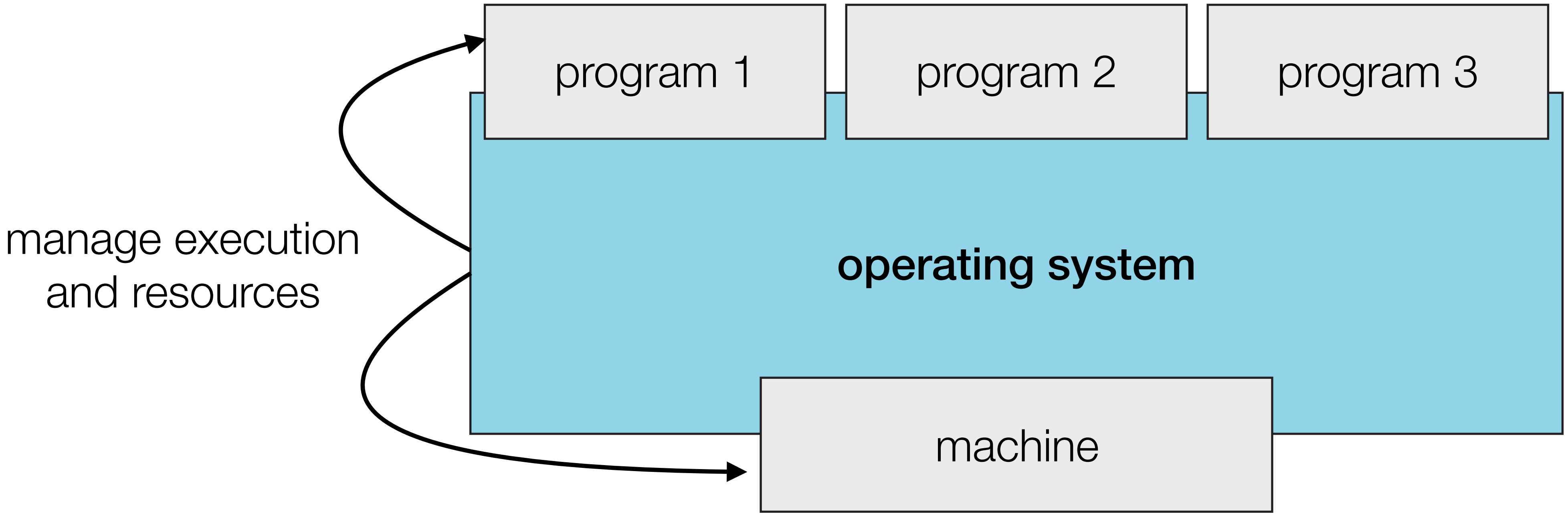
- **concurrency - a property of software**
  - illusion of running many processes at a time
  - can be achieved by:
    - time-sharing (single core)
    - execute in parallel (multi-core)
- **parallel - a property of hardware**
  - truly in parallel, simultaneously - hardware support essential
  - divide 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 want 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 components and we are primarily talking about the OS kernel.



Operating System

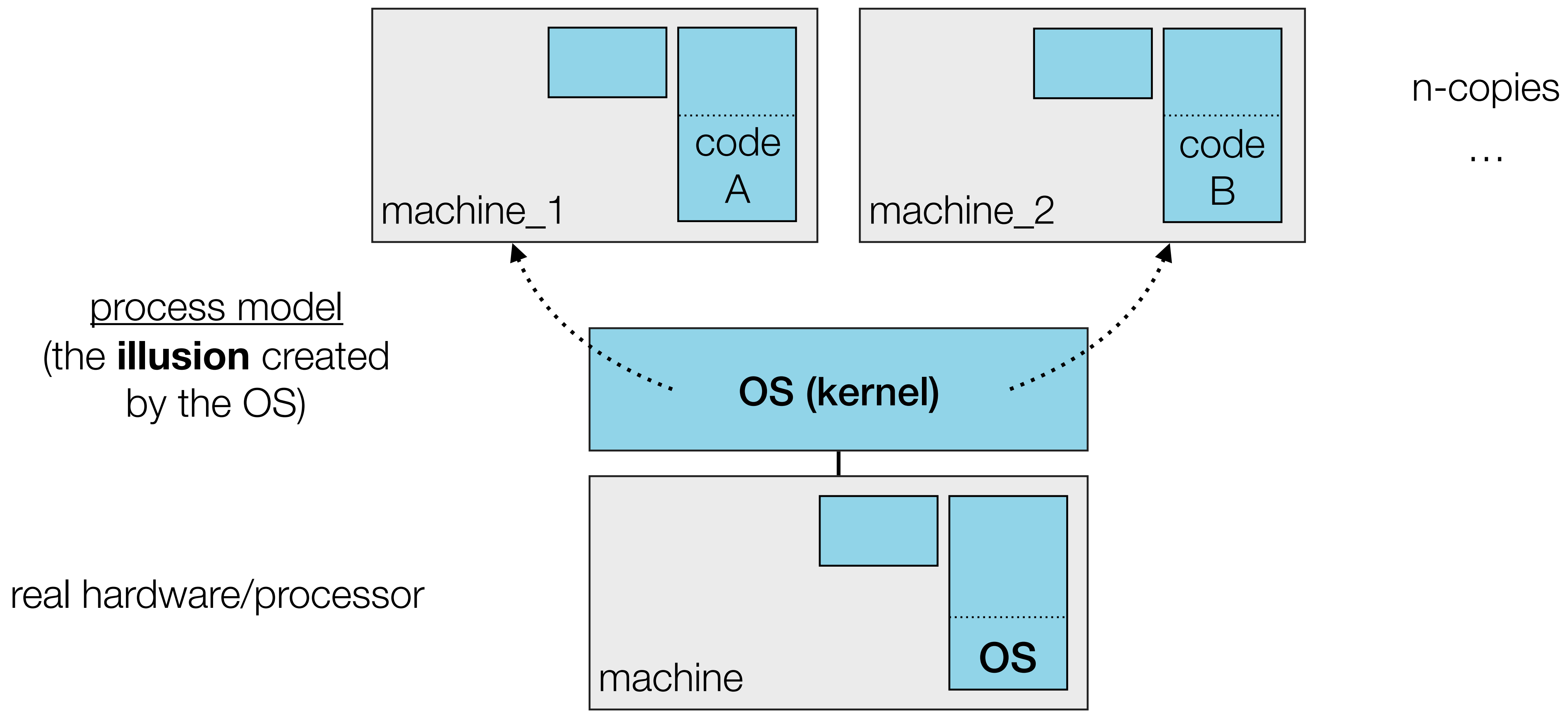
*An operating system manages resources and controls the **execution of many processes** on a machine.*

- 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
  - <u>hardware virtualization</u> (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

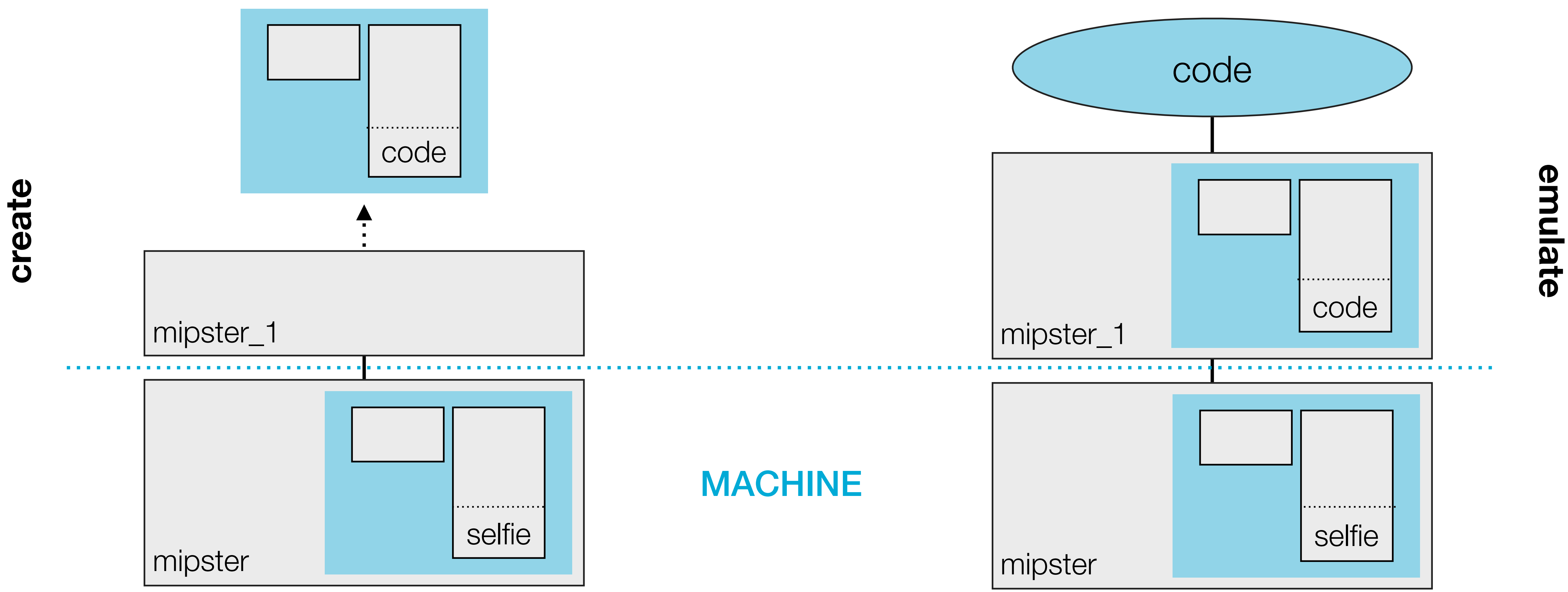
"The OS could interpret codeA for a while, then stop, save codeA's context and interpret codeB for a while, then stop, save codeB's context, restore codeA's context and interpret codeA for a while, then stop, save…"

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

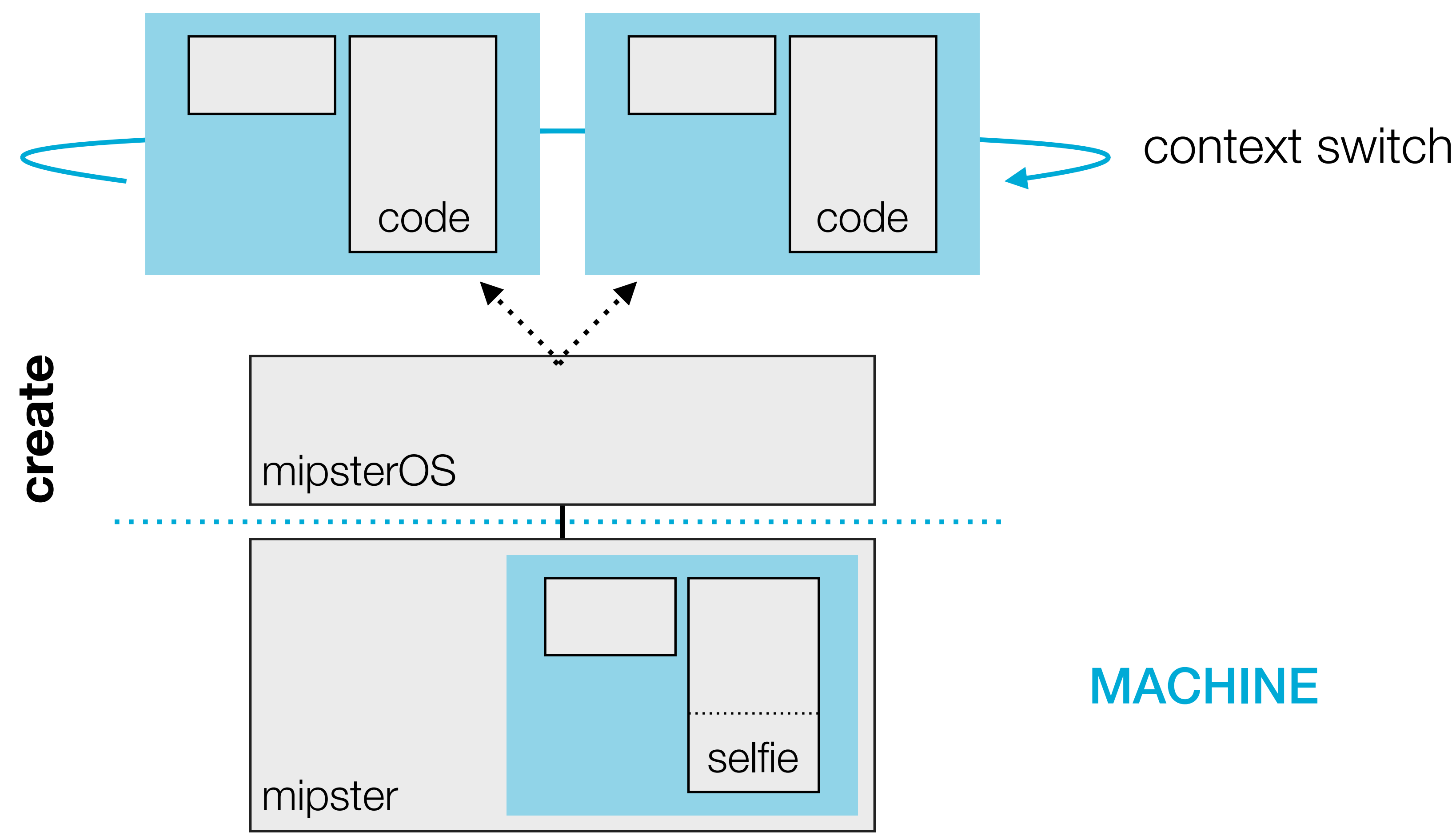
- **Consider this situation:**
  - Let our machine (mipster) execute selfie and start mipster, say mipster_1
  - mipster_1 creates and emulates **one instance** of the machine it is run on!
    - we want create and emulate **two instances** of that machine → two contexts



MipsterOS

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

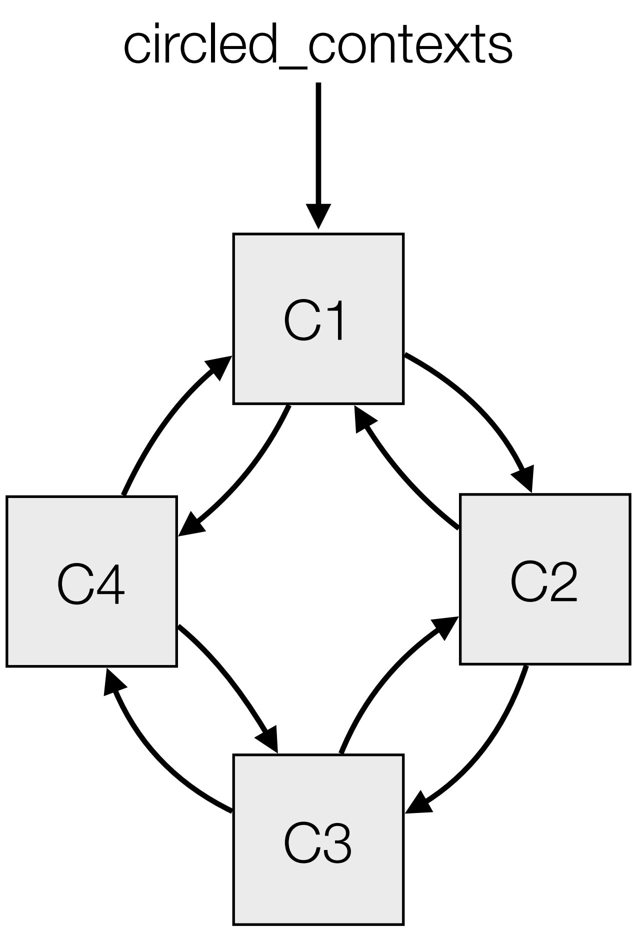
- Instead of creating and emulating one context, this system **creates and emulates 2** contexts concurrently by switching between them **after every instruction**.



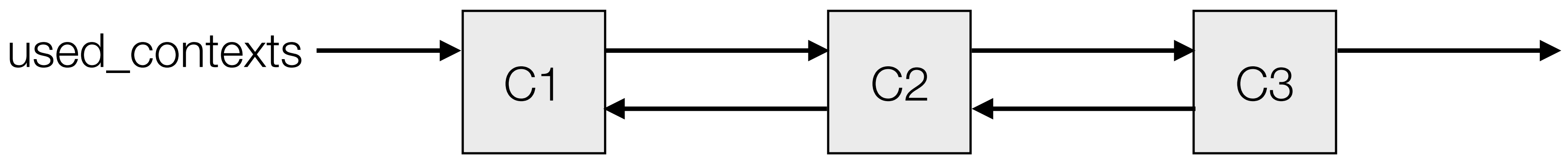
Context

`./selfie -c`

- 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



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 `hello-world.c` → 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 <u>`write`</u> 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** → <u>`handle_exception`</u>, <u>`handle_system_call`</u>
  - mipsterOS <u>implements</u> 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

- Modes 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**

```
#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*

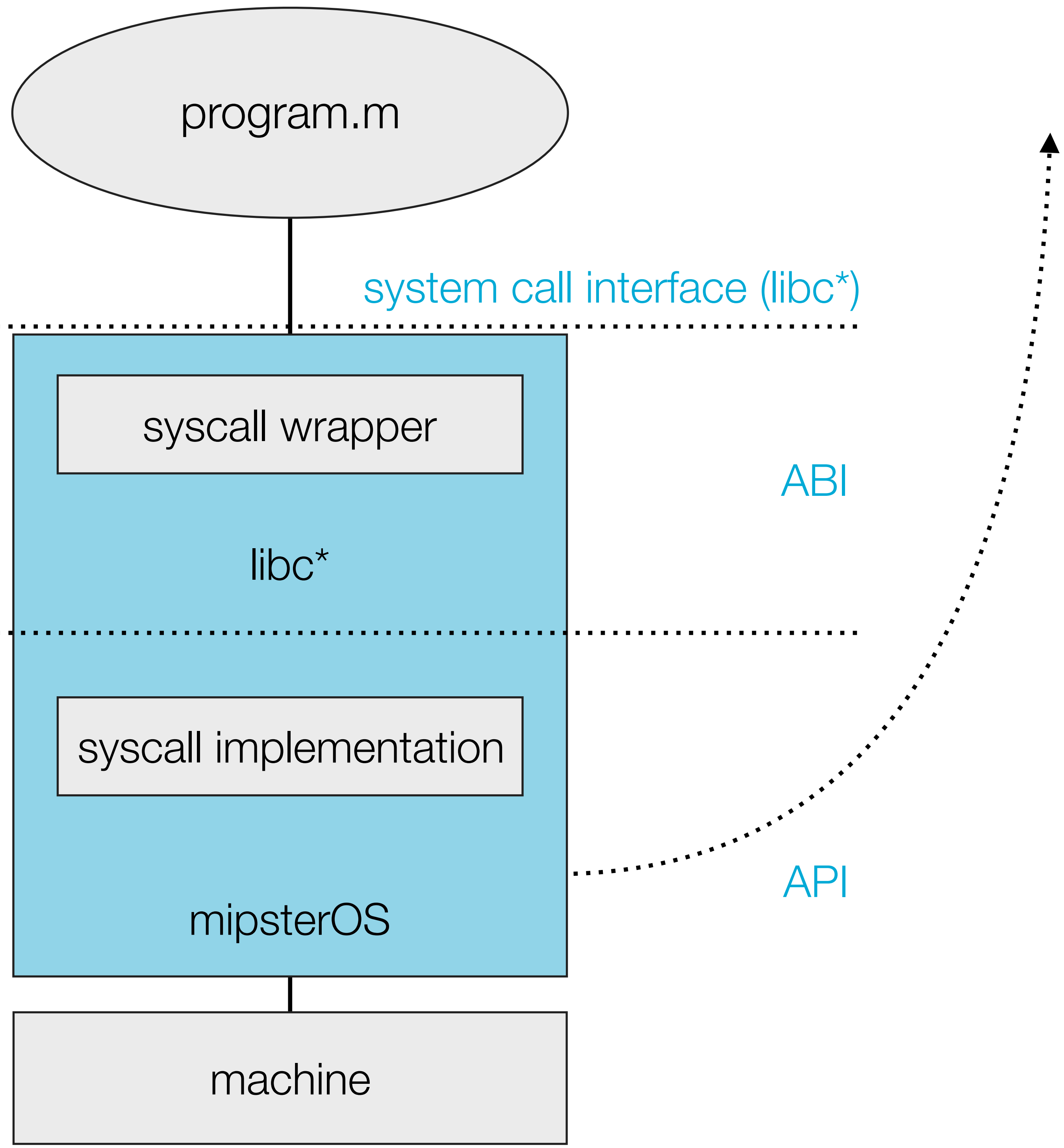
`write(…)`

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.

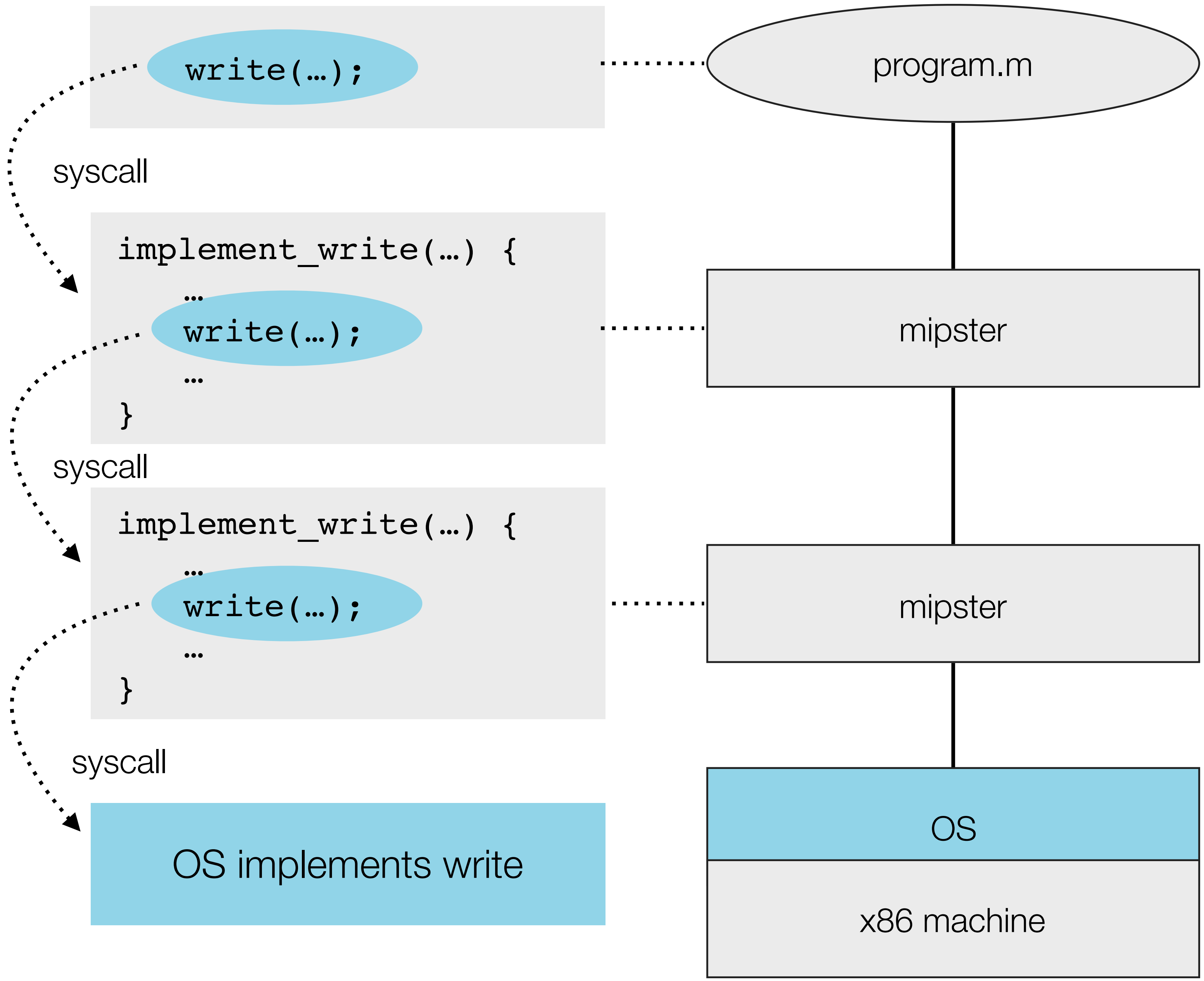
The other, as you might have noticed, is that the actual implementation of a syscall in mipster/mipsterOS contains that same syscall. (implement_write contains write)”.

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 synchronized (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**

A Write Lock Intended Semantics

a variable indicating lock owner

```
uint64_t LOCK;
…
```

lock call saves caller as lock owner (acquire 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

unlock call removes caller as lock owner (release lock)

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 operating 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'
  - 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,…)

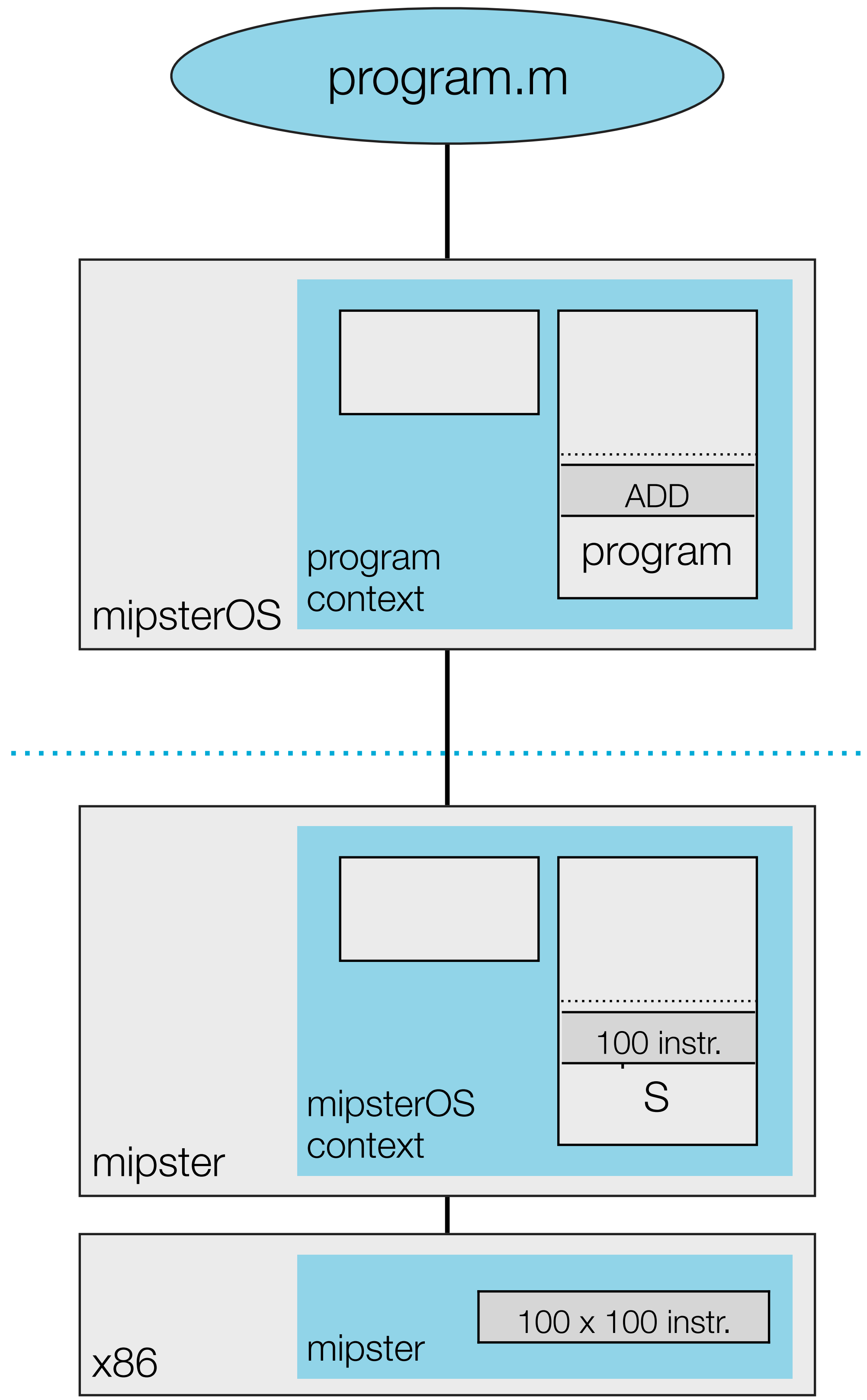
x100

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

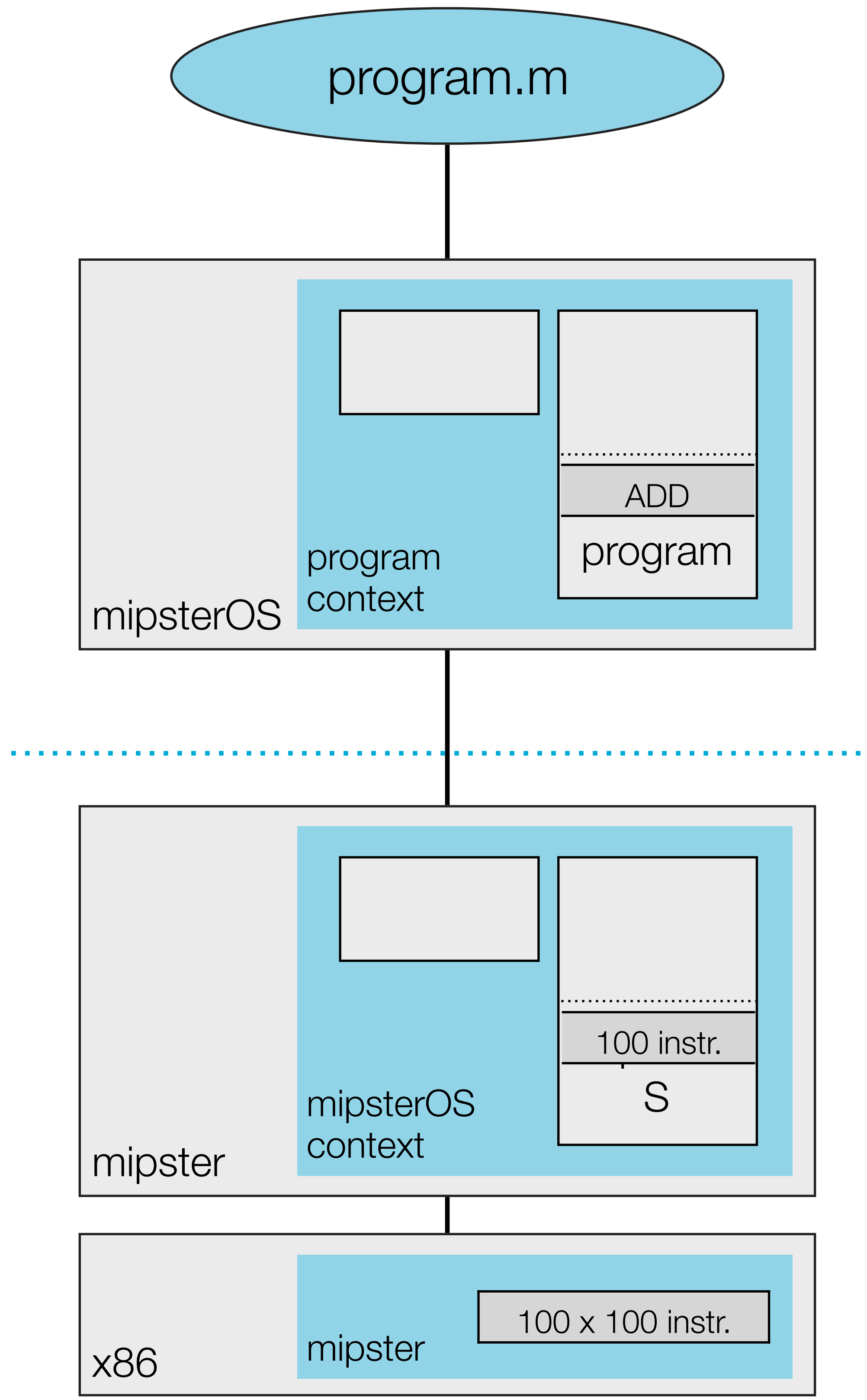
100 KHz emulator
(emulate 10^5 instr./s)*

:100

10 MHz emulator
(emulate 10^7 instr./s)*

:100

1 GHz processor
(execute 10^9 instr./s)*



USER PROCESS

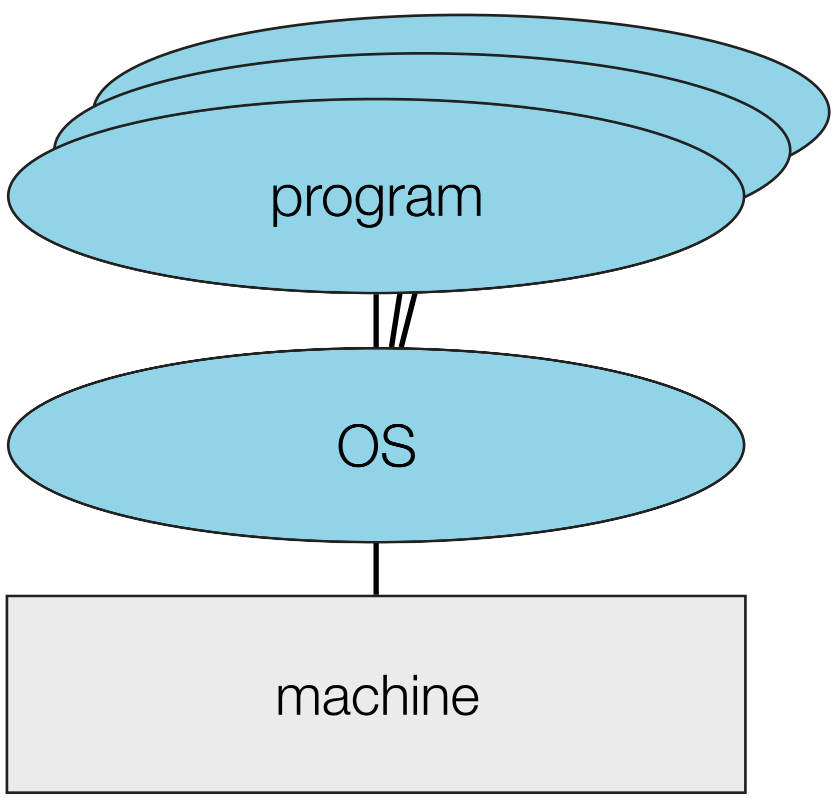
OS
emulation through interpretation

MACHINE
in software implemented processor,
emulation through interpretation

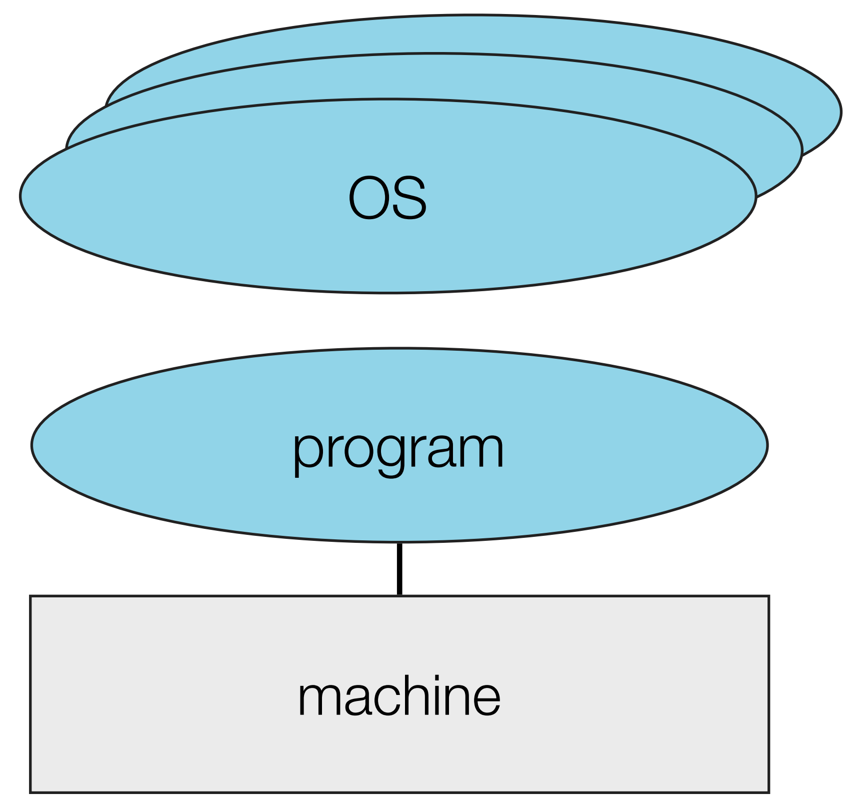
*assumption: 100 instructions to emulate 1, 1 instructions per cycle

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?**



interpretation is slow



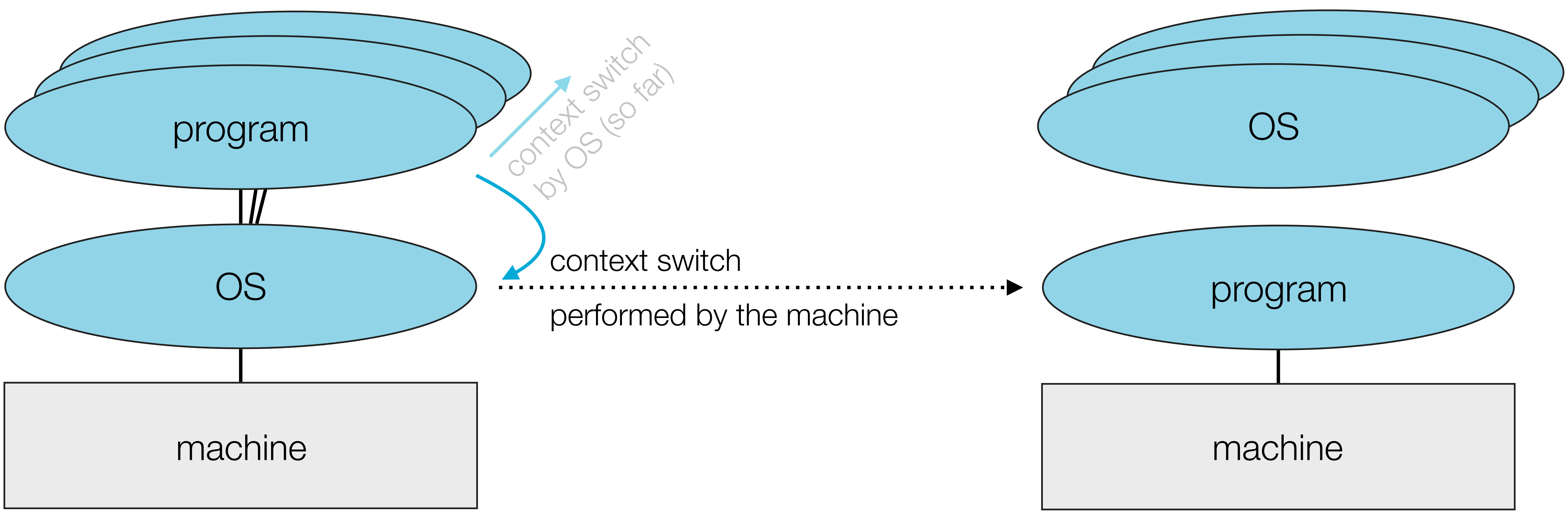
virtualization is fast

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 so 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



interpretation is slow

virtualization is fast

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

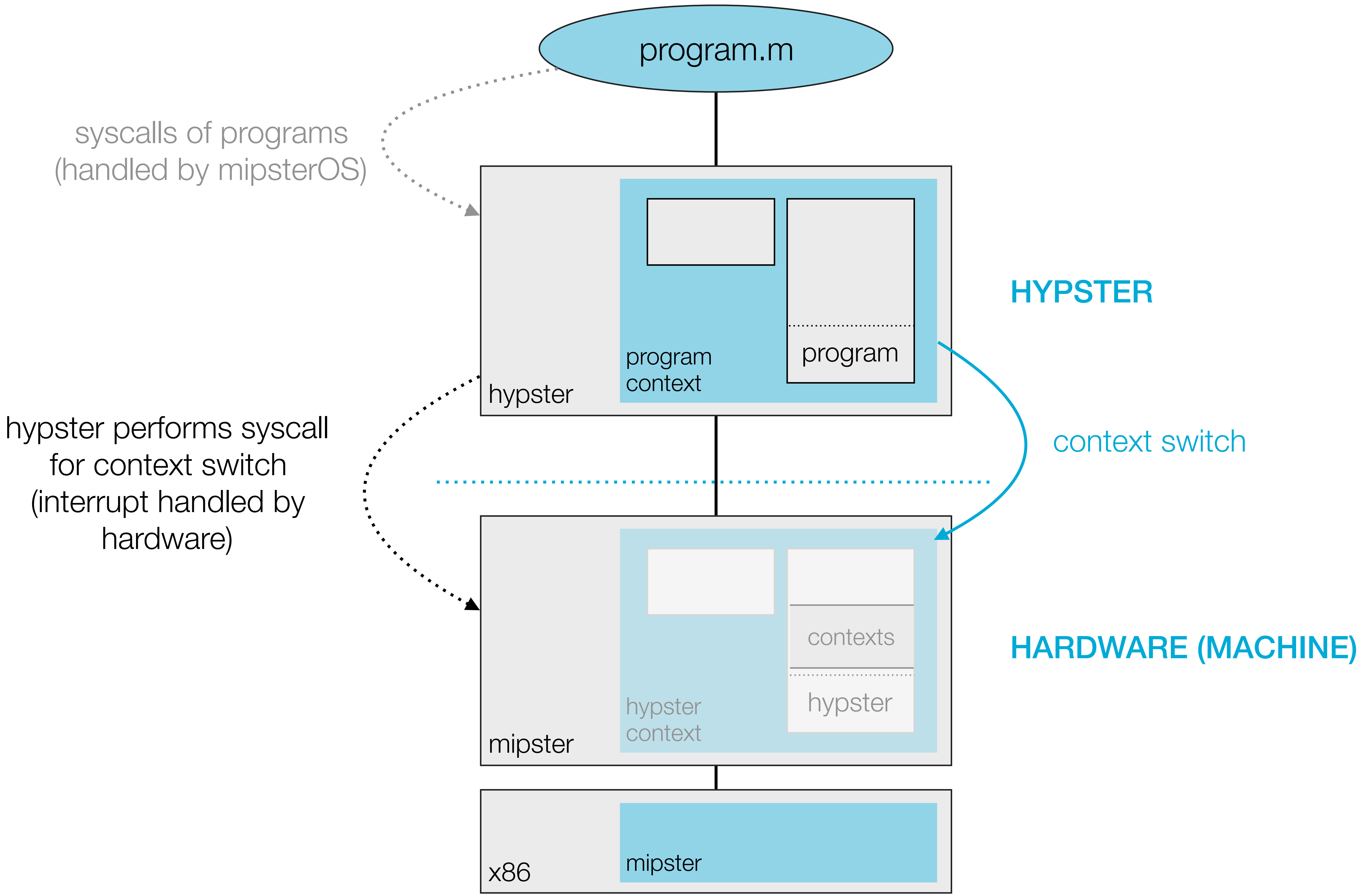
Hypster Hypervisor

- 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

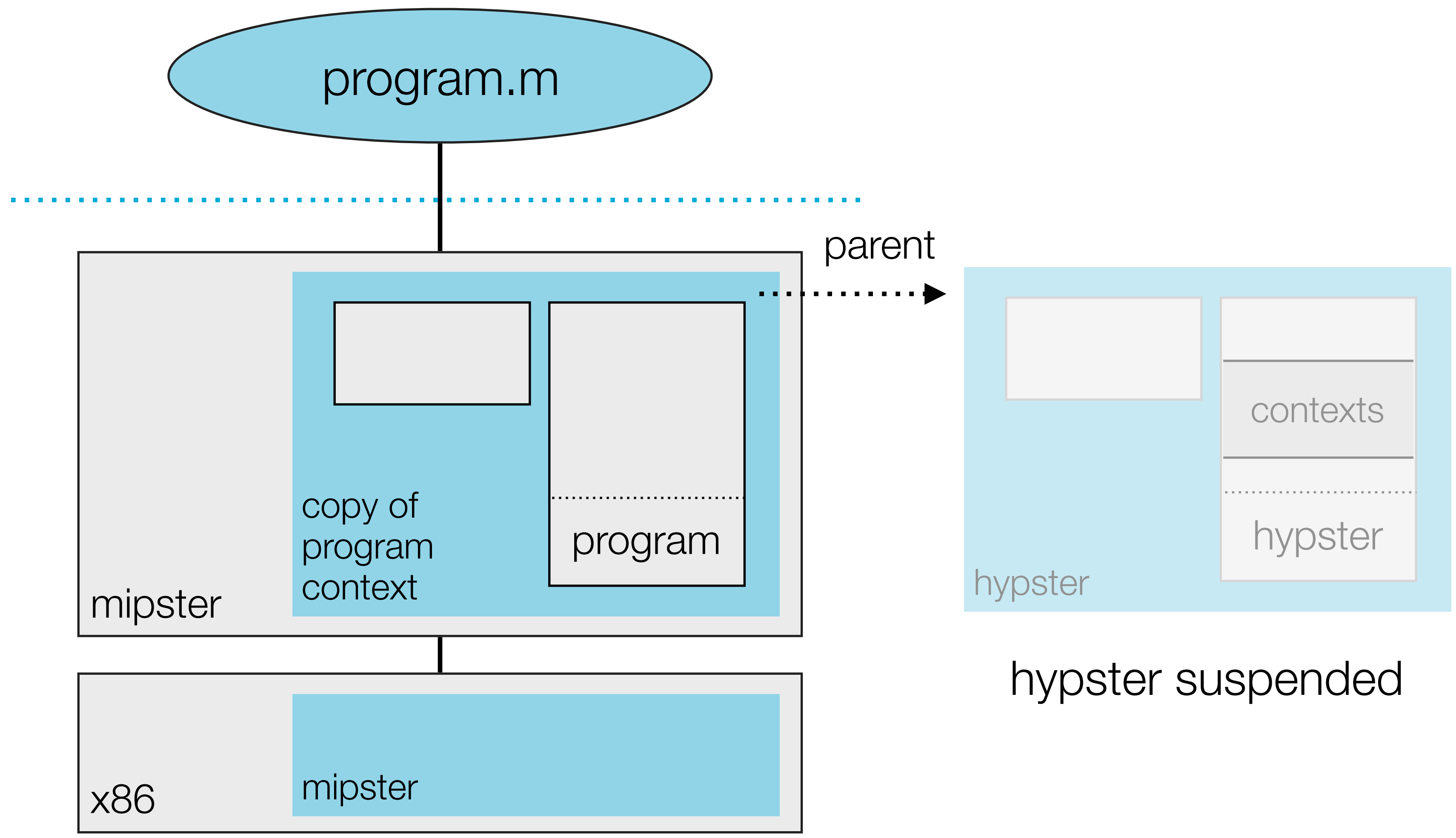
- **mipster switch** - mipster and mipsterOS
  - context switch performed by mipster/mipsterOS
  - switching implemented in software
- **hypster switch** - hypster
  - syscall by hypster
  - context switch performed by the machine hypster is running on
  - switching implemented in hardware

Context Switch Hypster → Program



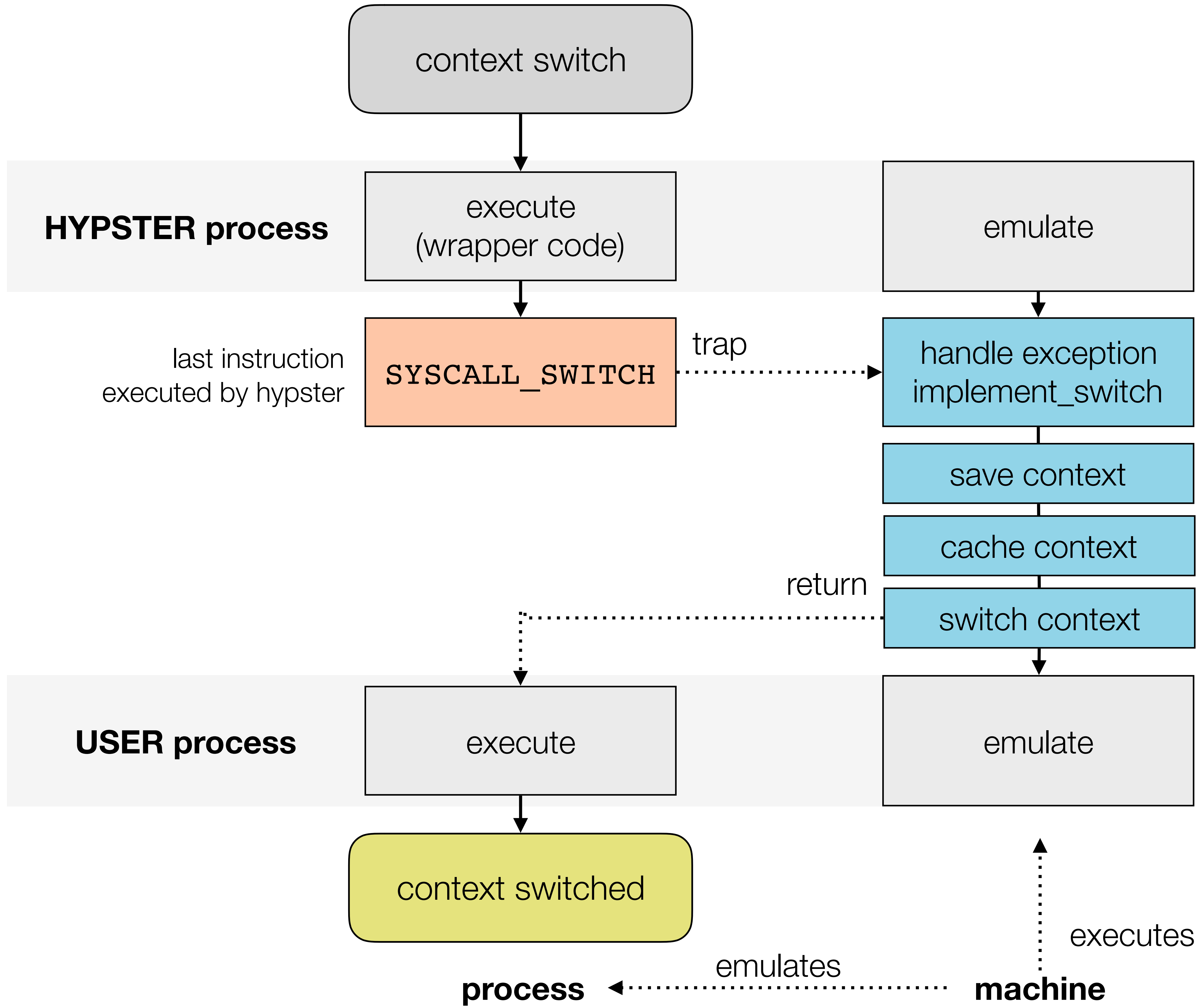
Context Switch Hypster → Program

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 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:**

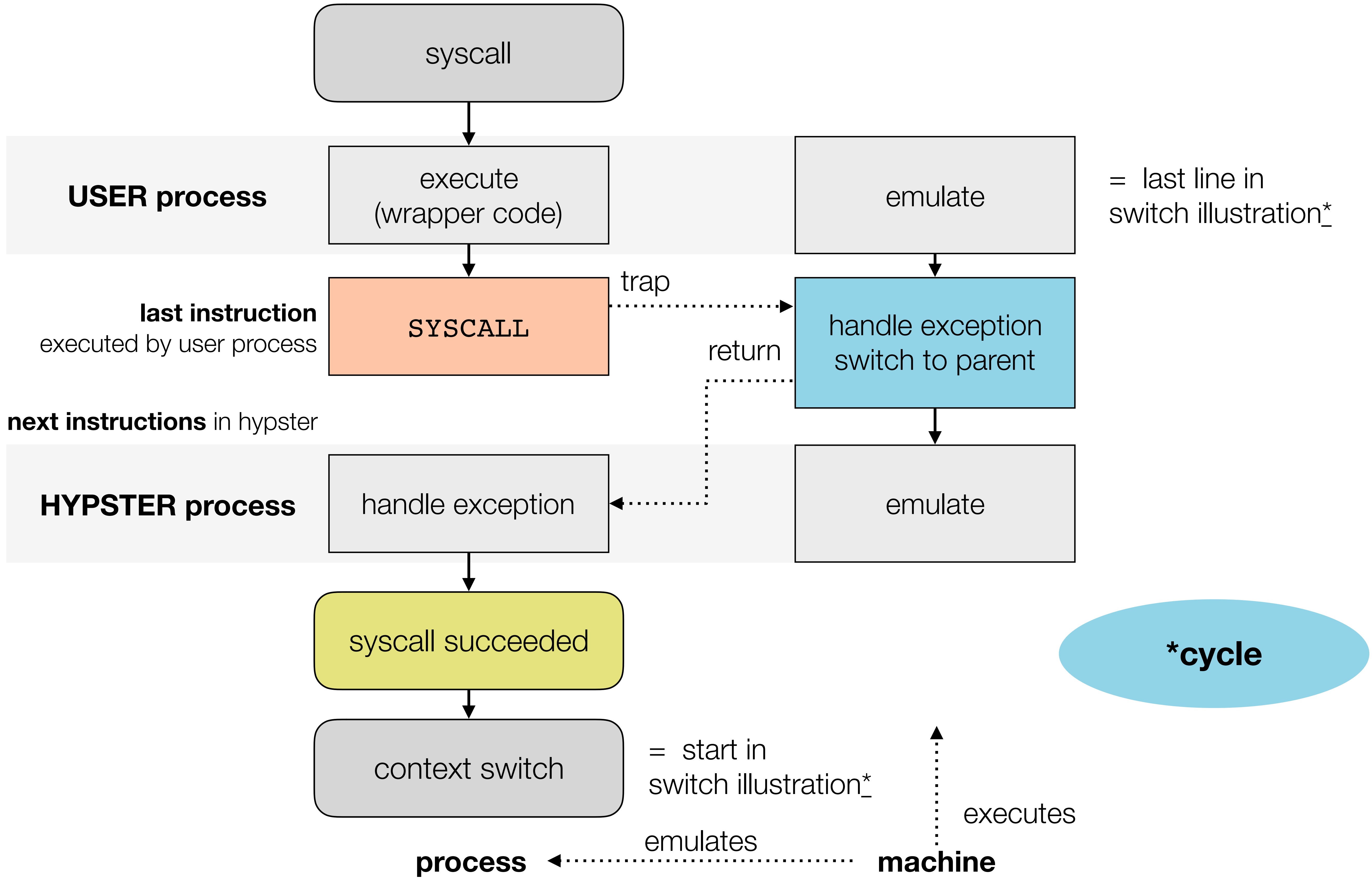
1. program runs on machine and makes syscall → trap

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.*

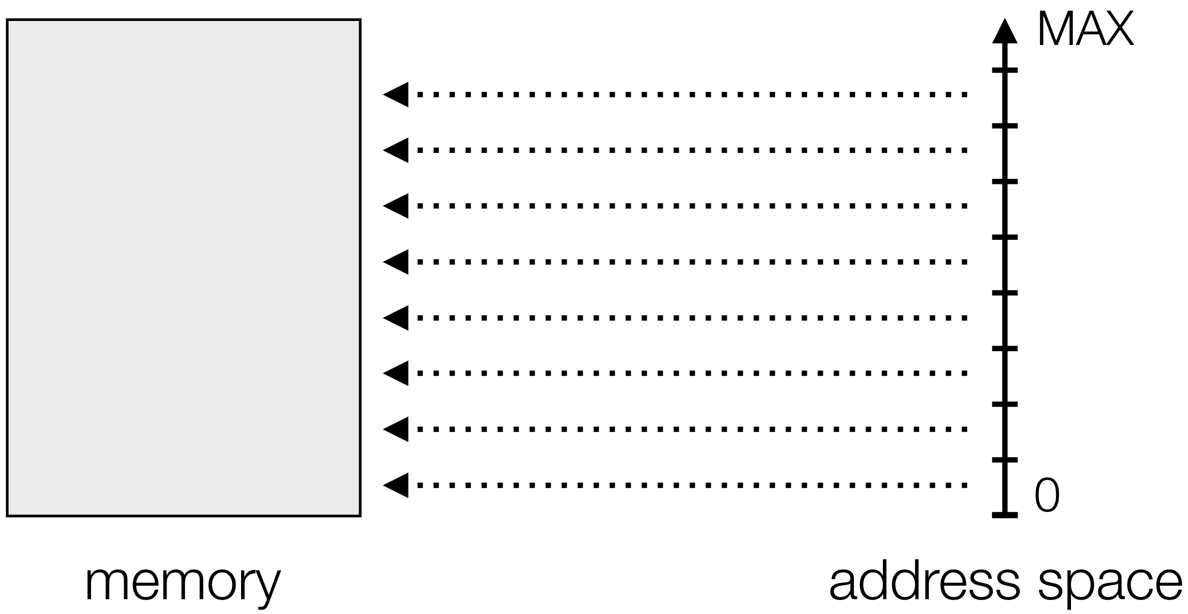
- Selfie can now execute two or more programs at the same time efficiently
  - the problem of having only **one physical CPU** was solved by virtualizing the CPU using a time-sharing approach
  - the challenge of sharing **one physical memory** was already solved by the operating system support implemented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

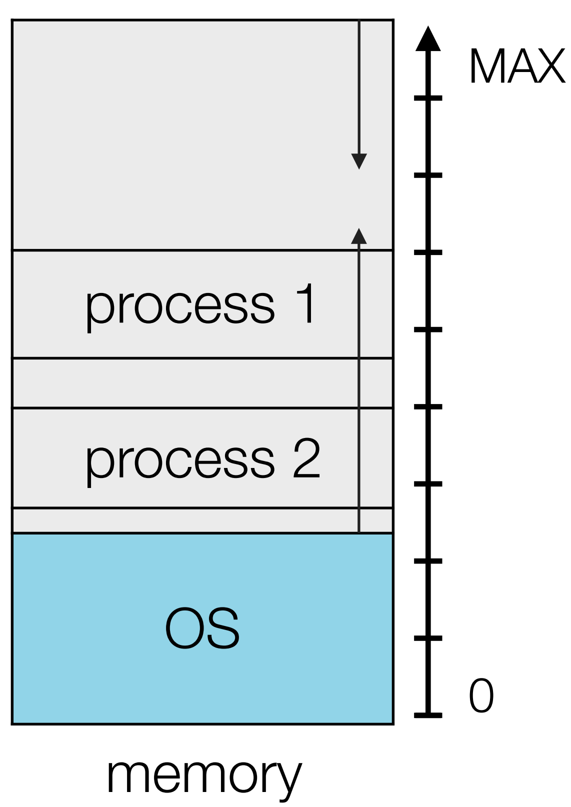
“How could the OS actually execute two processes with one memory concurrently?”

“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 problems 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
  - **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

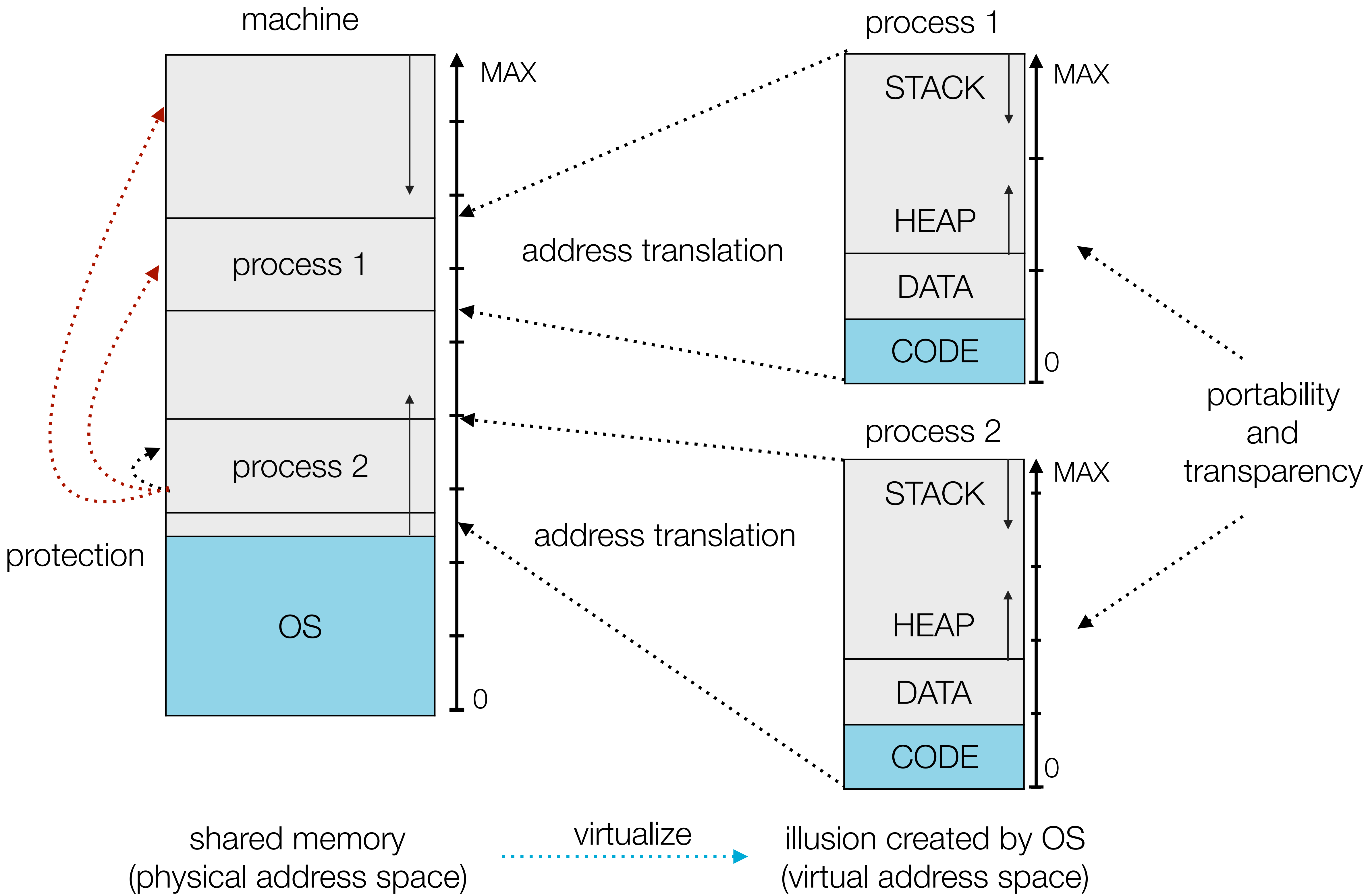
- OS abstracts physical address space using a technique called **address translation**
- OS provides each process with **virtual memory**, an **illusion of private memory**
  - process uses <u>**virtual addresses**</u>
    (code compiles with virtual addresses)
  - OS assigns physical memory to virtual 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

Address Translation 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

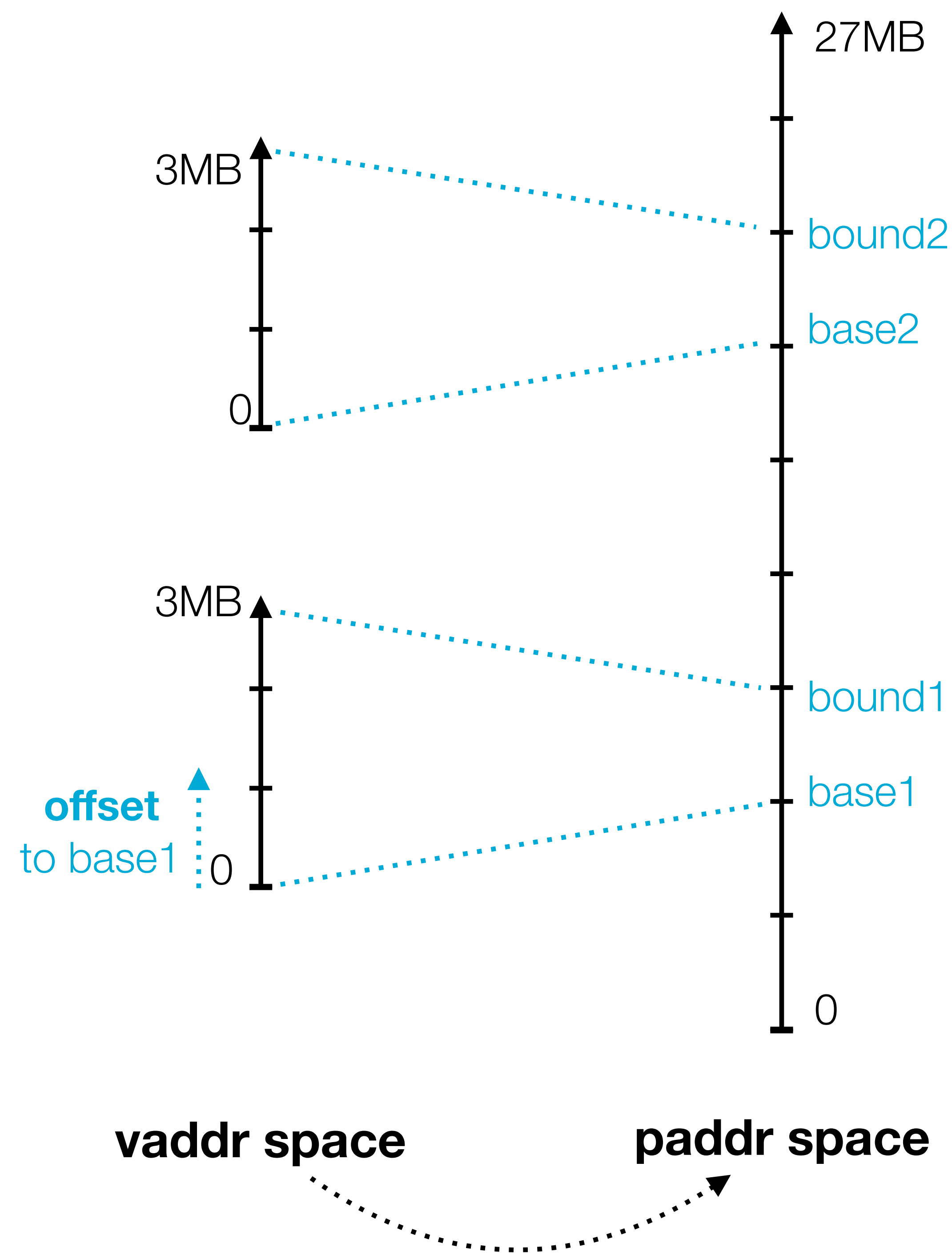
TRANSPARENCY

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

Memory Virtualization Base and Bound

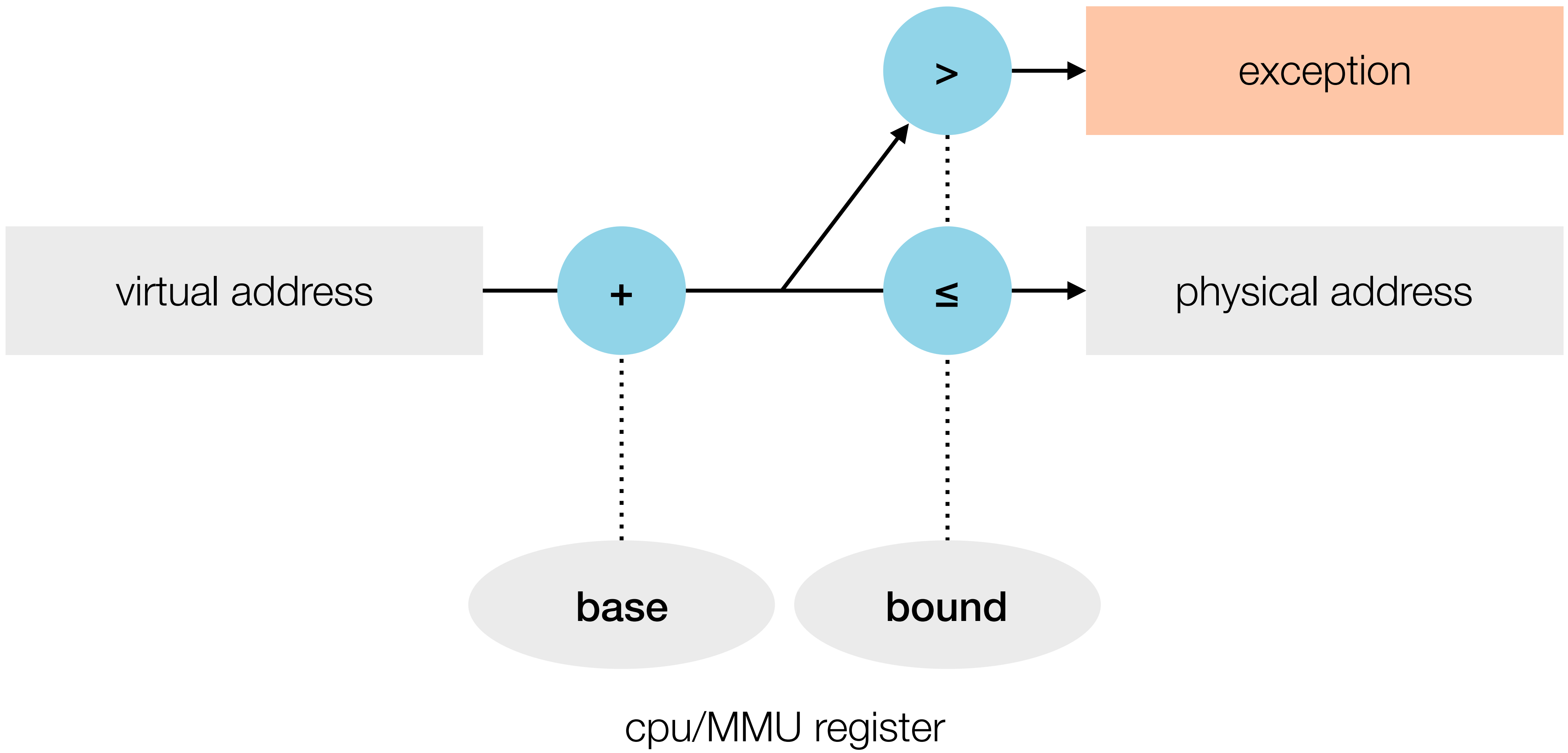
process 1	base1	bound1
process 2	base2	bound2

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

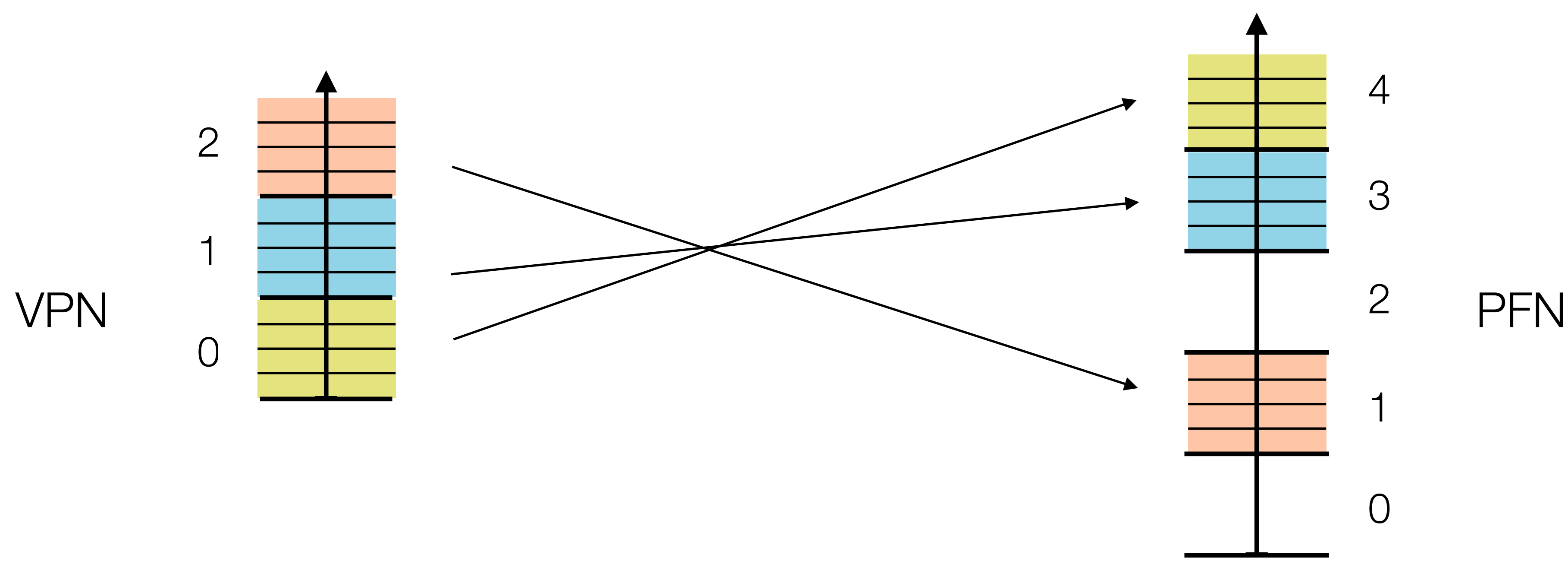
- **similar** to base and bound
- virtual memory divided into **variable-sized logical segments**
  - code, data, heap, stack
  - base and bound for each segment
  - variable-sized → external fragmentation
- less internal fragmentation

“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**



pages
vaddr space

page frames
paddr space

Paging Page Table

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

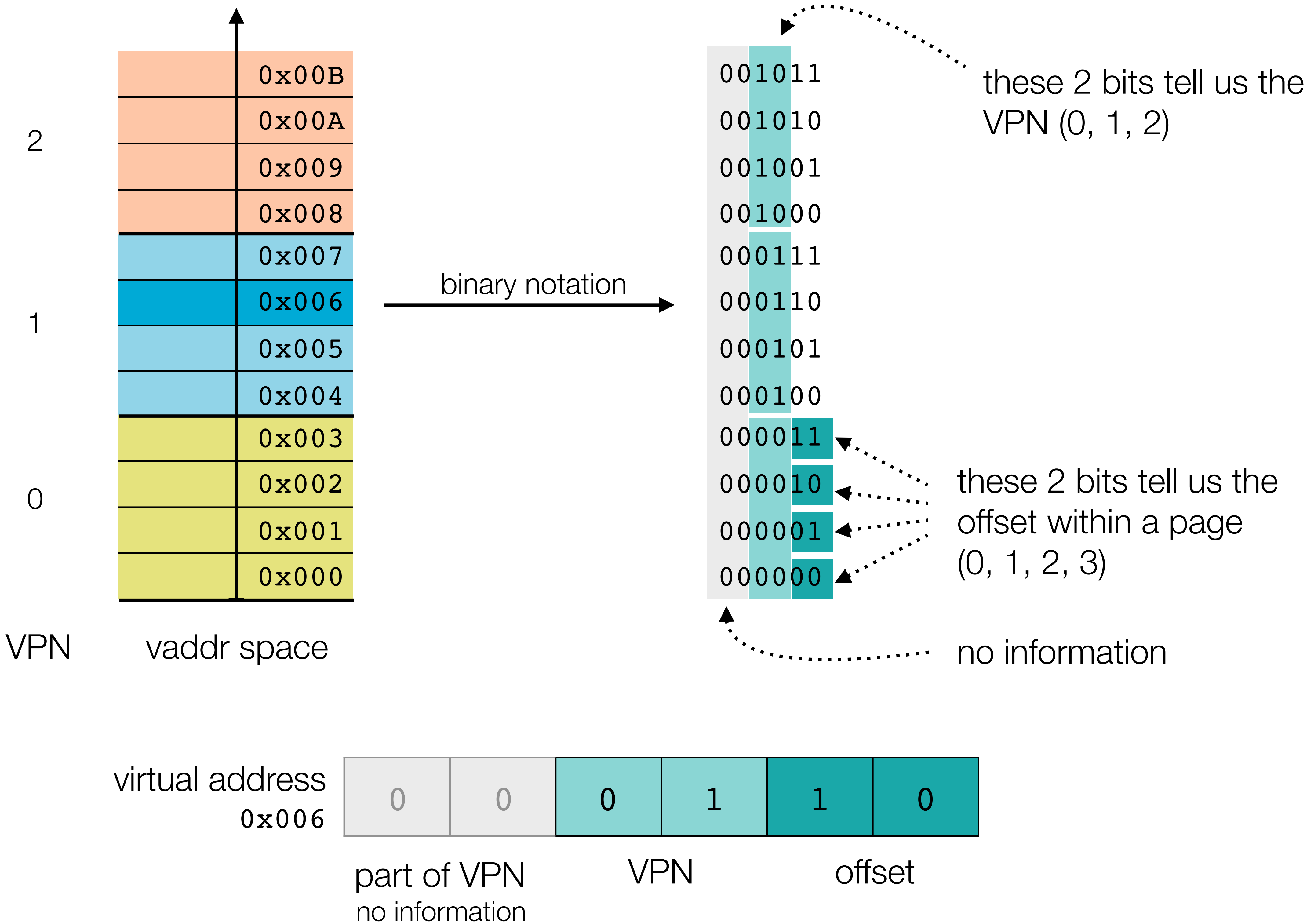
- **OS** maintains one page table per process
- simple implementation
  - array lookup
  - indexed with VPN to find PFN

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…

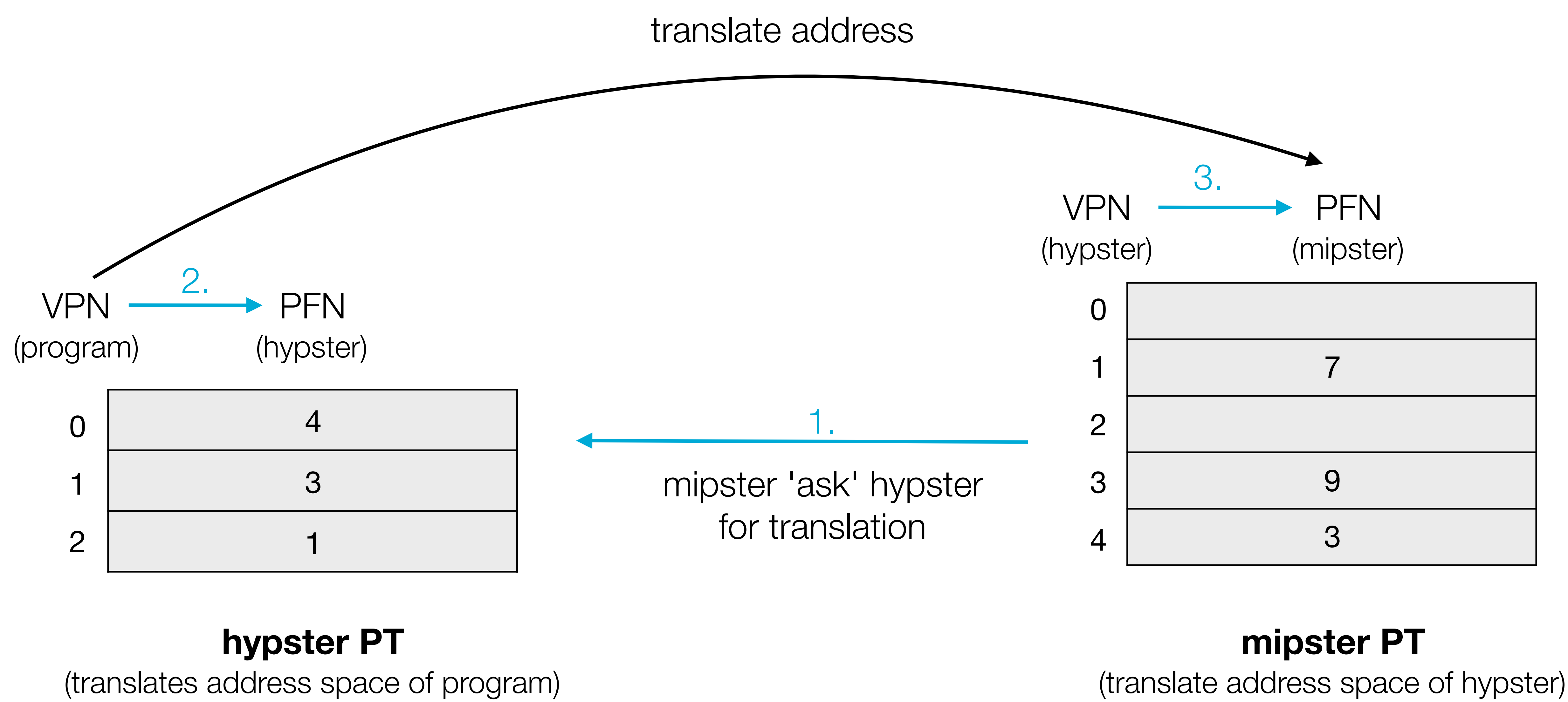
Address Translation is the last missing piece necessary to understand why we cache a hosted context.”

or skip next 3 slides

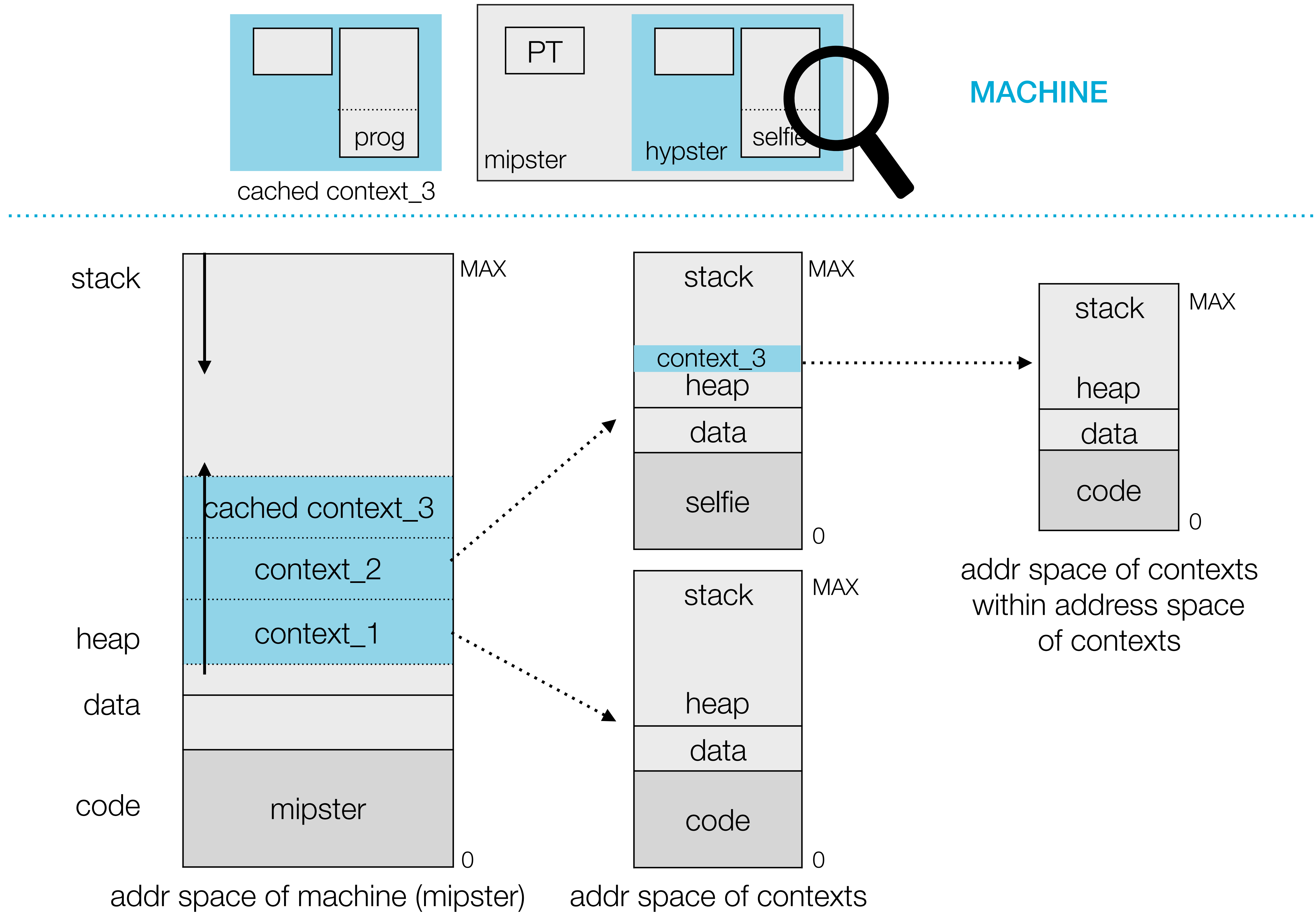
Cache Context Why?

- 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** an **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



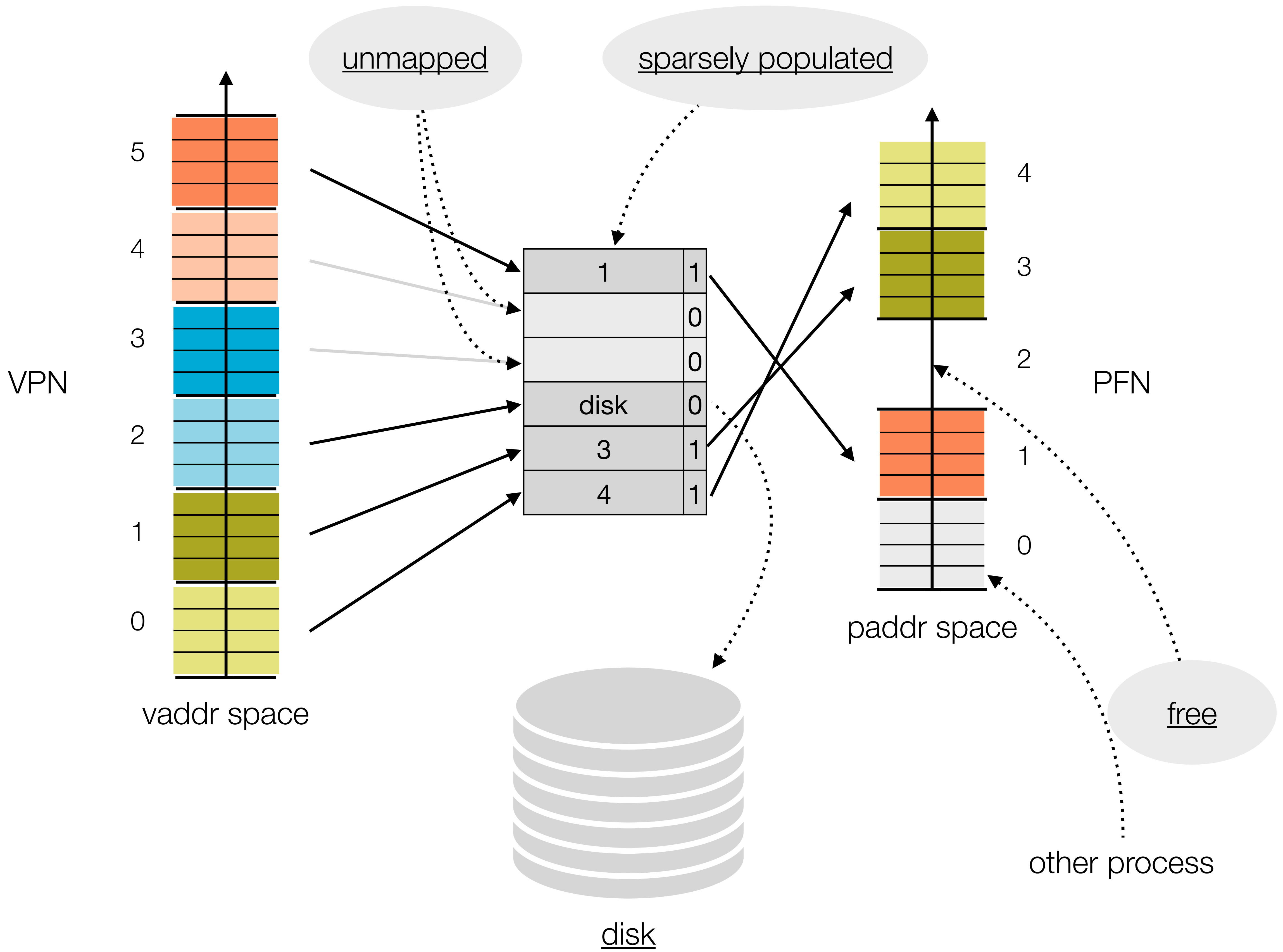
Cache Context Cache Context



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)

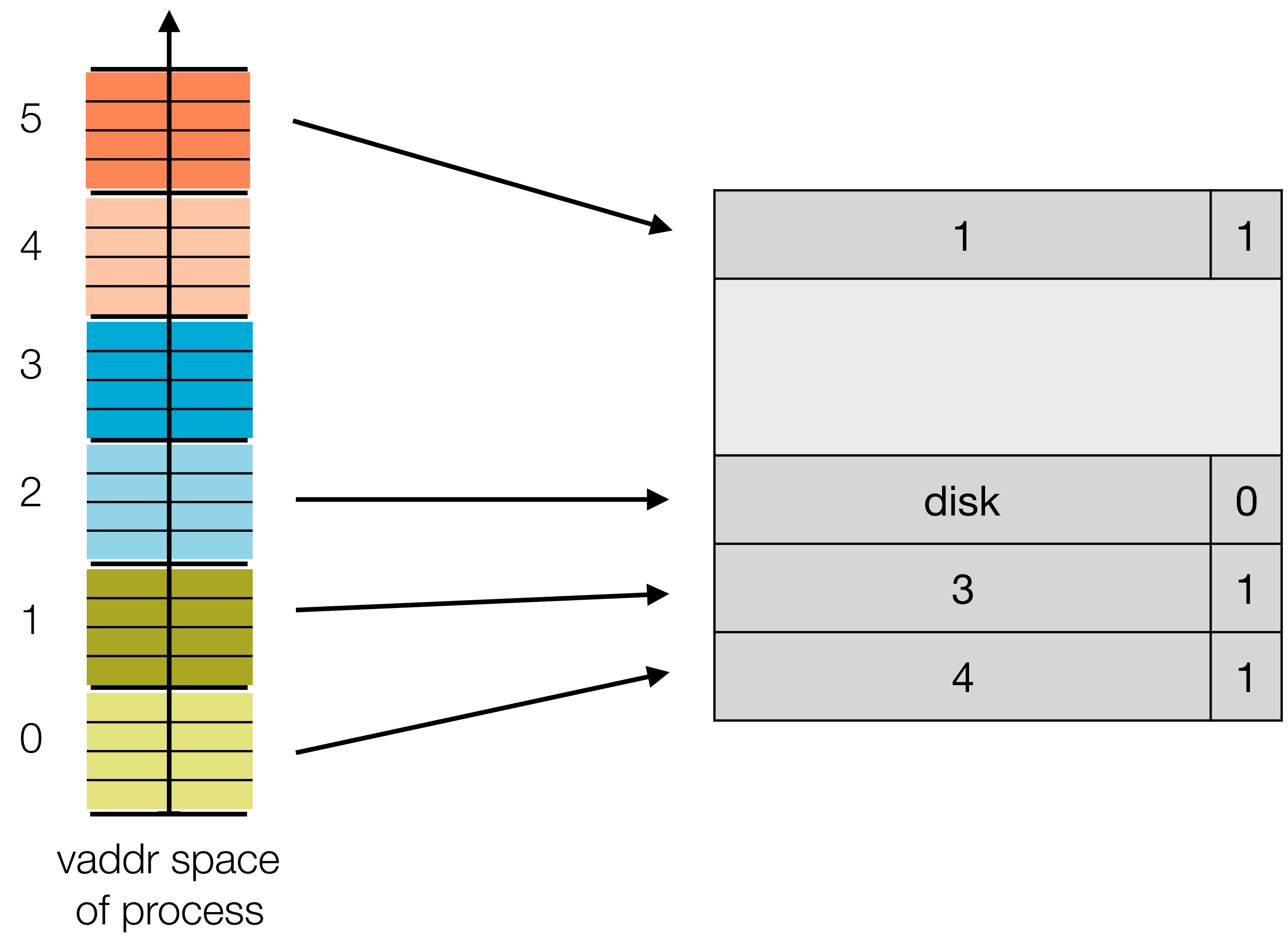
We can get around this easily by not mapping the entire virtual address space. Demand paging is one possibly way of doing so.

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`

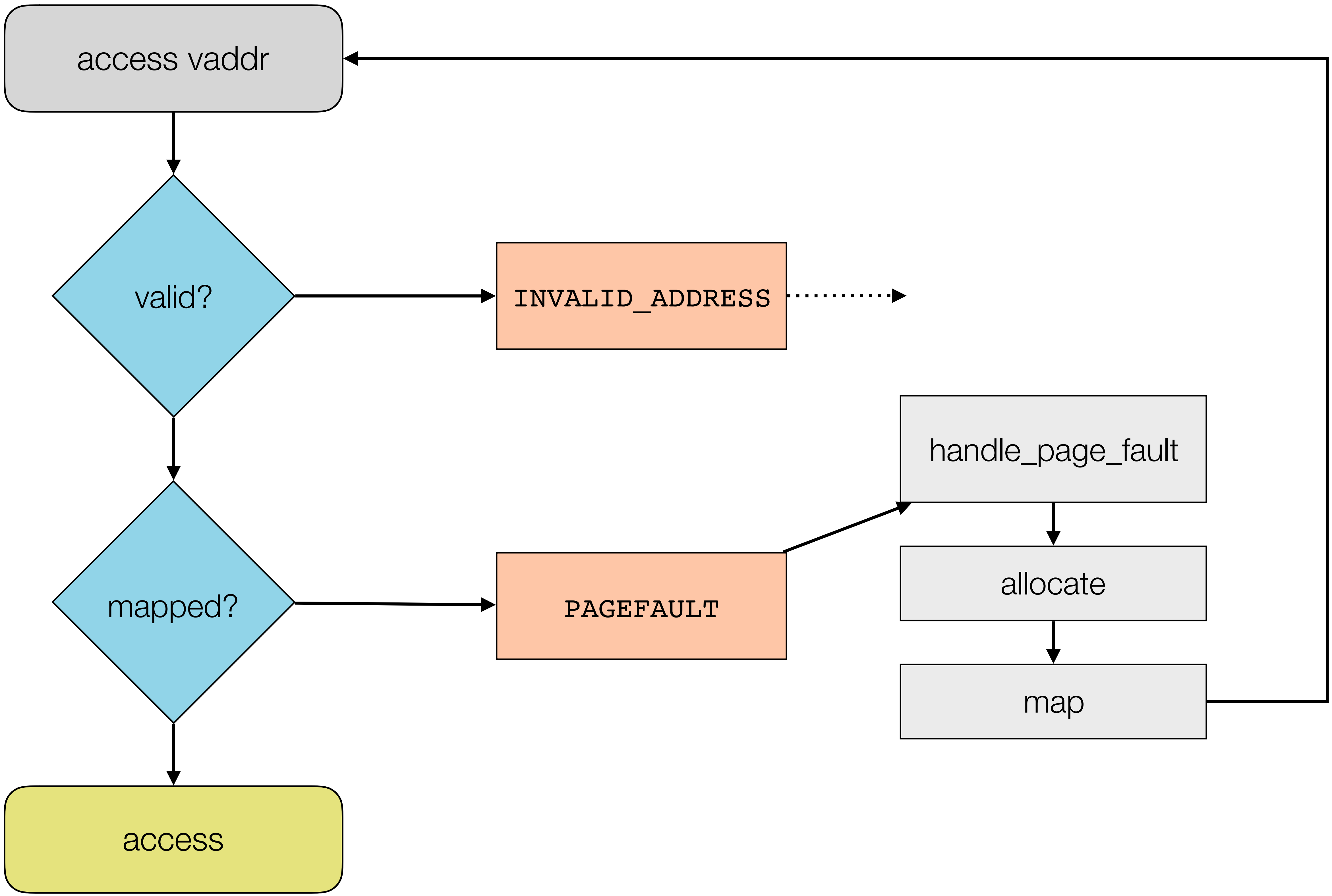
- **process accesses an virtual address** (ex. `do_ld`)
  - check if virtual address is **valid**
    - within vaddr space, word-addressed
    - `is_valid_virtual_address`
  - check if page for virtual address is **mapped**
    - `is_virtal_address_mapped`, `is_page_mapped`
  - if both checks **pass**, 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

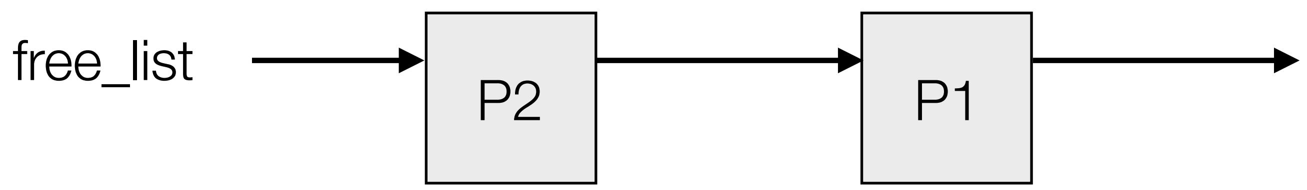
Problem *→ Which frames can be used?*

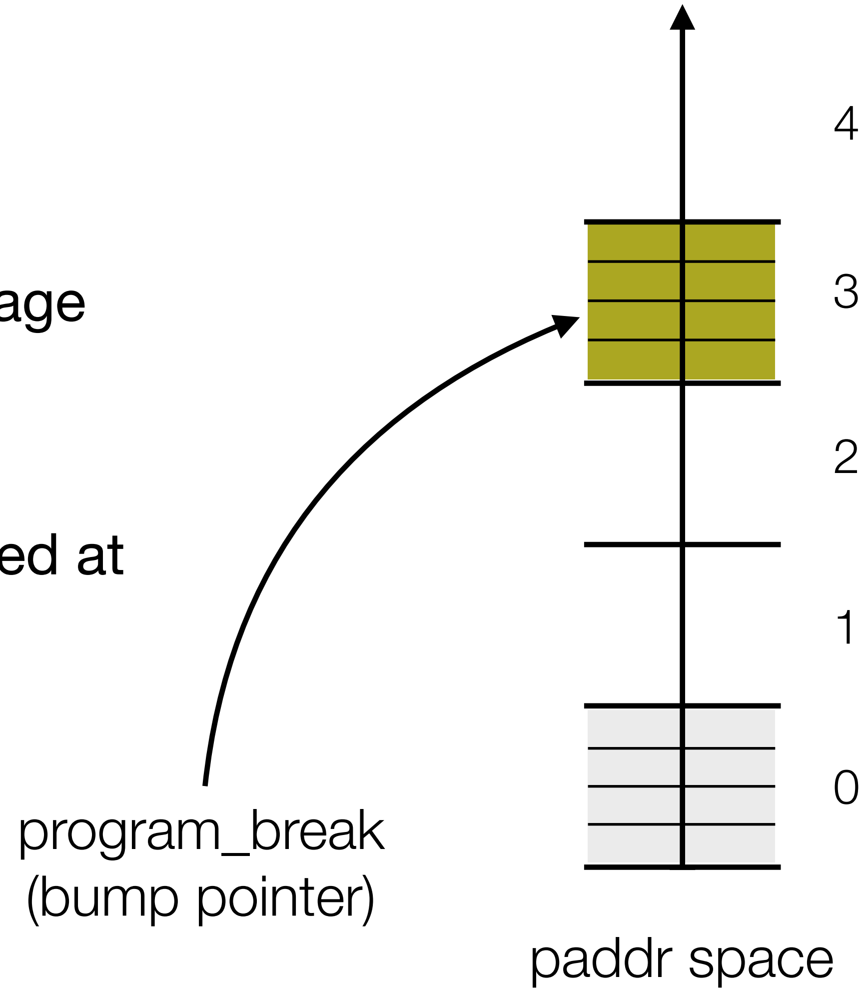
- the set of available frames can be partitioned int **used and free** frames
  - used_page_frame_memory, free_page_frame_memory
- page frame allocator
  - keeps track of **free** pages
    - used memory is known by the application that uses the allocator (information in page table)
    - using a <u>free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

Paging Page Frame Allocation in Selfie

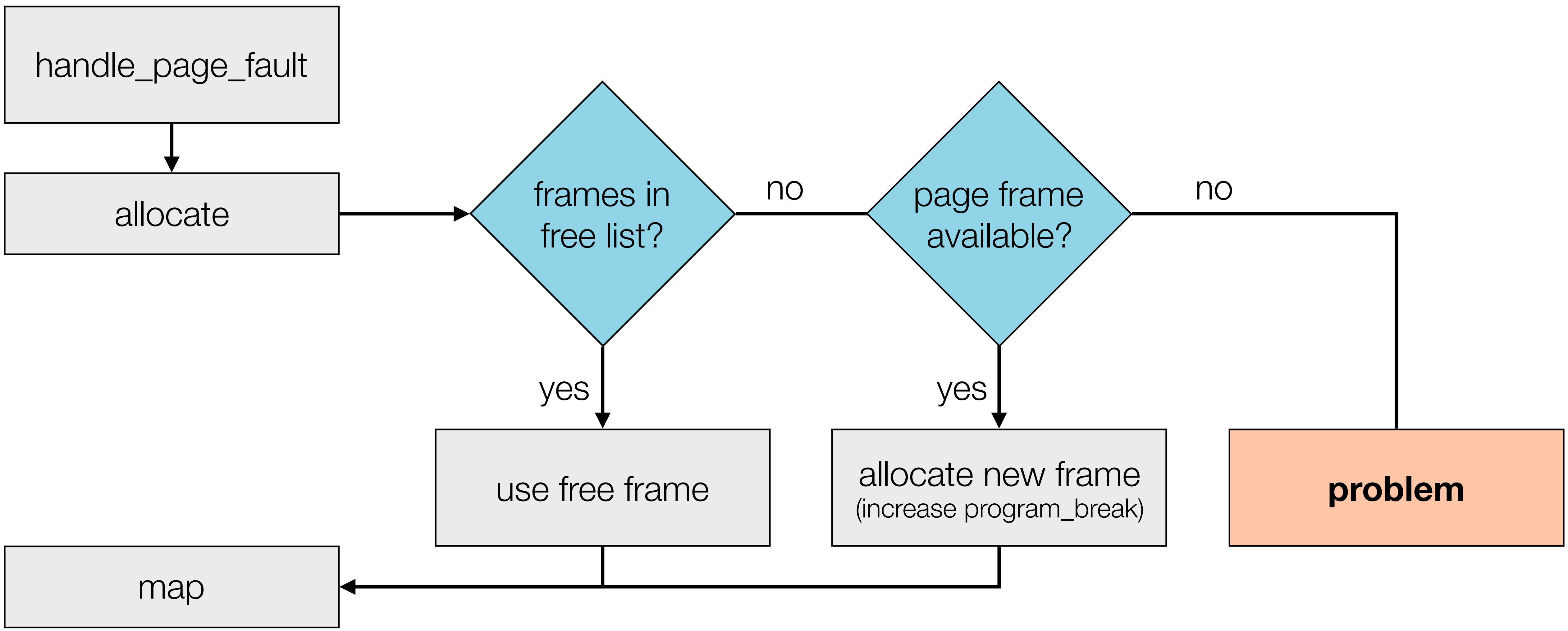
- selfie uses a simple memory allocation concept
  - bump pointer allocation

- 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?

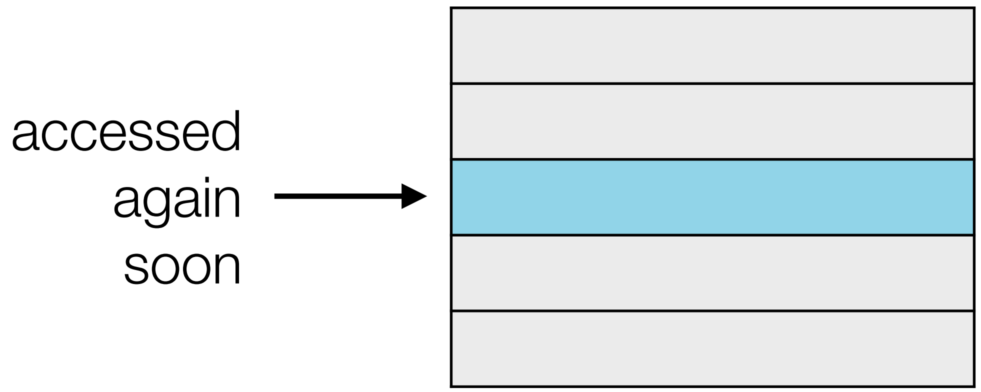
Paging Swapping

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

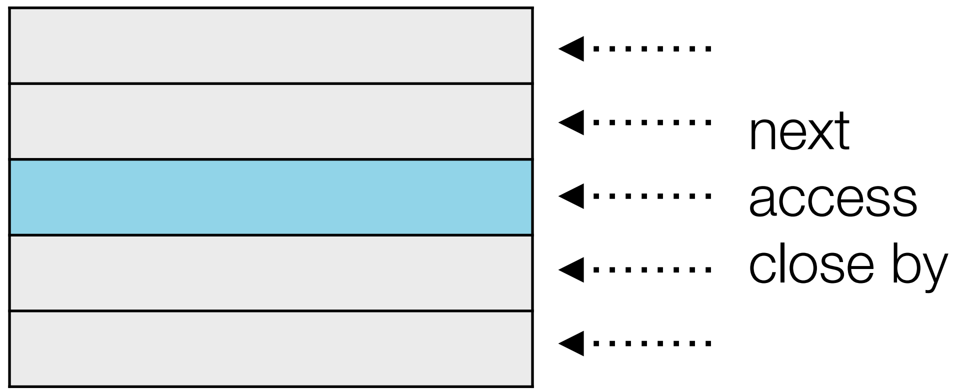
- best frame would be the one that will be accessed furthest in the future
- best we can do is find an **approximation** to that frame
- different **eviction strategies**, depending on assumption we make
  - **most recently used** (MRU)
    - free the frame that was touched most recently
    - assumption → 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

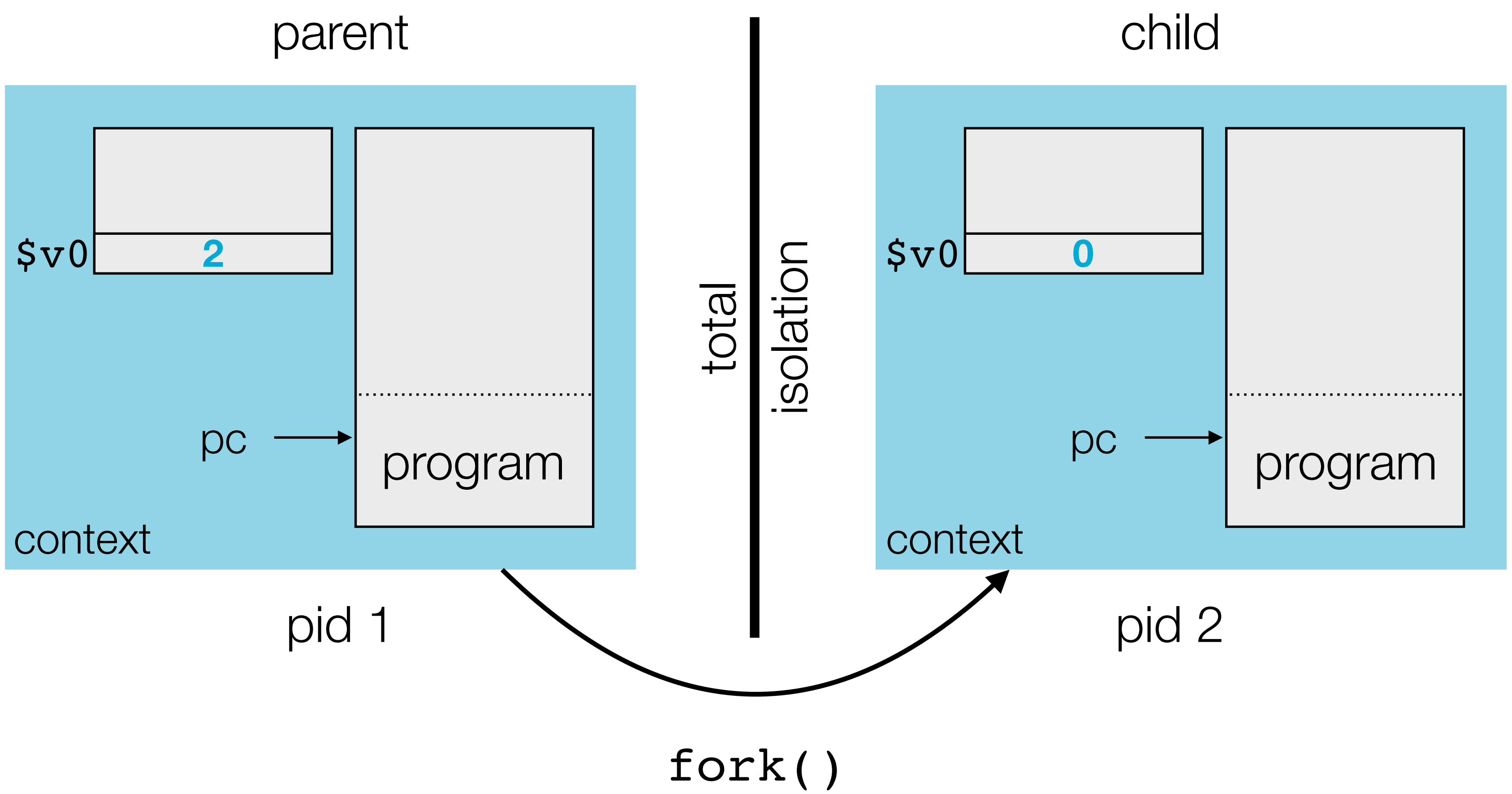
return **process identifier** of the process that was created (>= 0 if successful)

```
uint64_t main() {
   uint64_t child_pid;

   child_ pid = fork();

   …
}
```

simples 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>

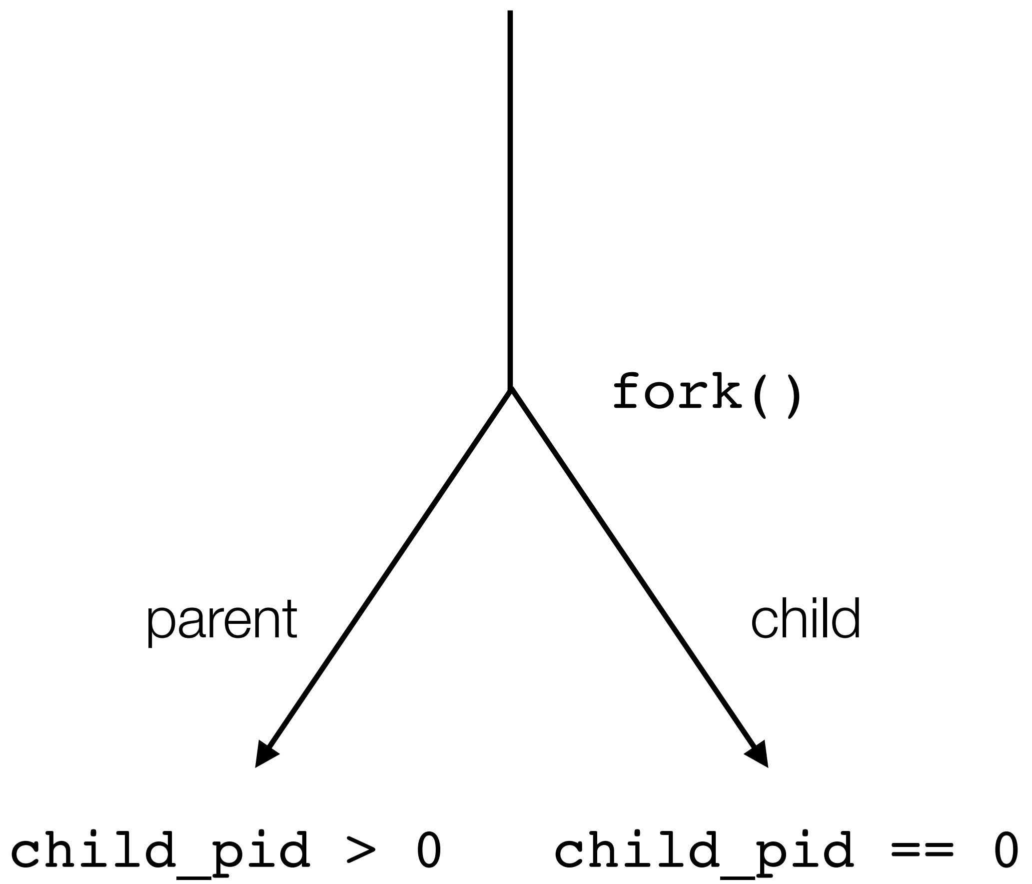
Syscall fork() Semantics



- 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

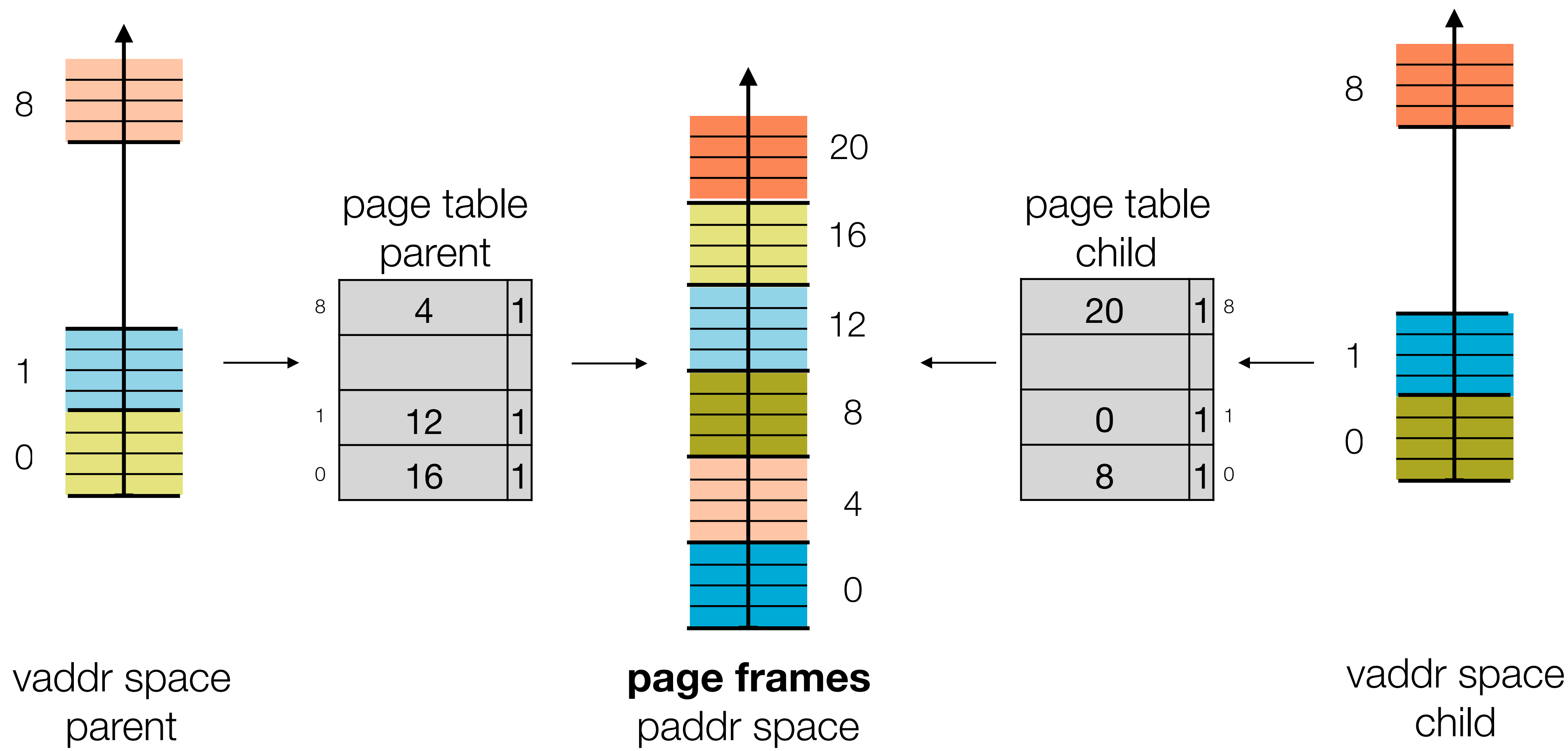
```
copy_memory() {
    allocate_new_child_page_table();


    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!

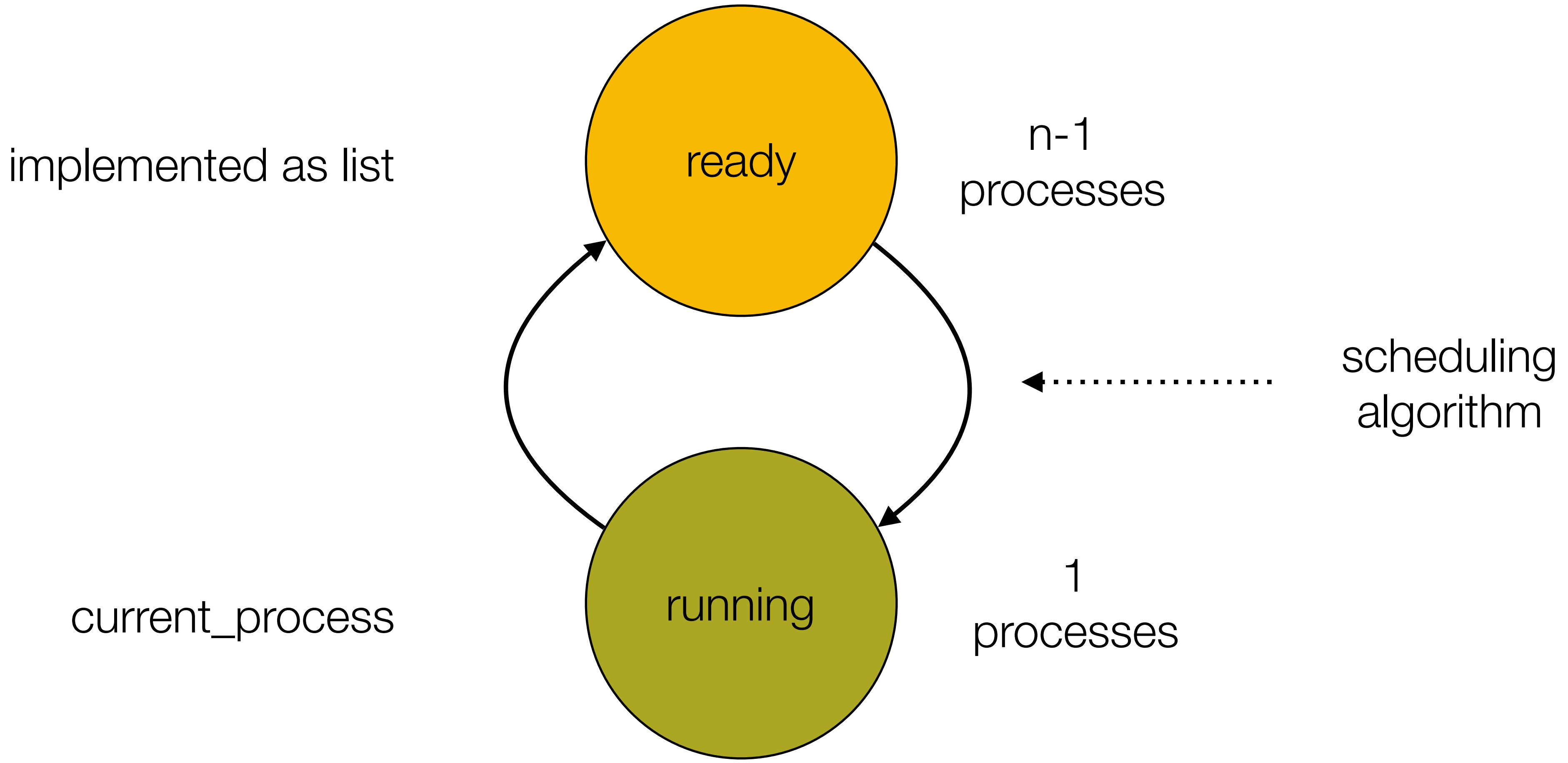
Manage

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!

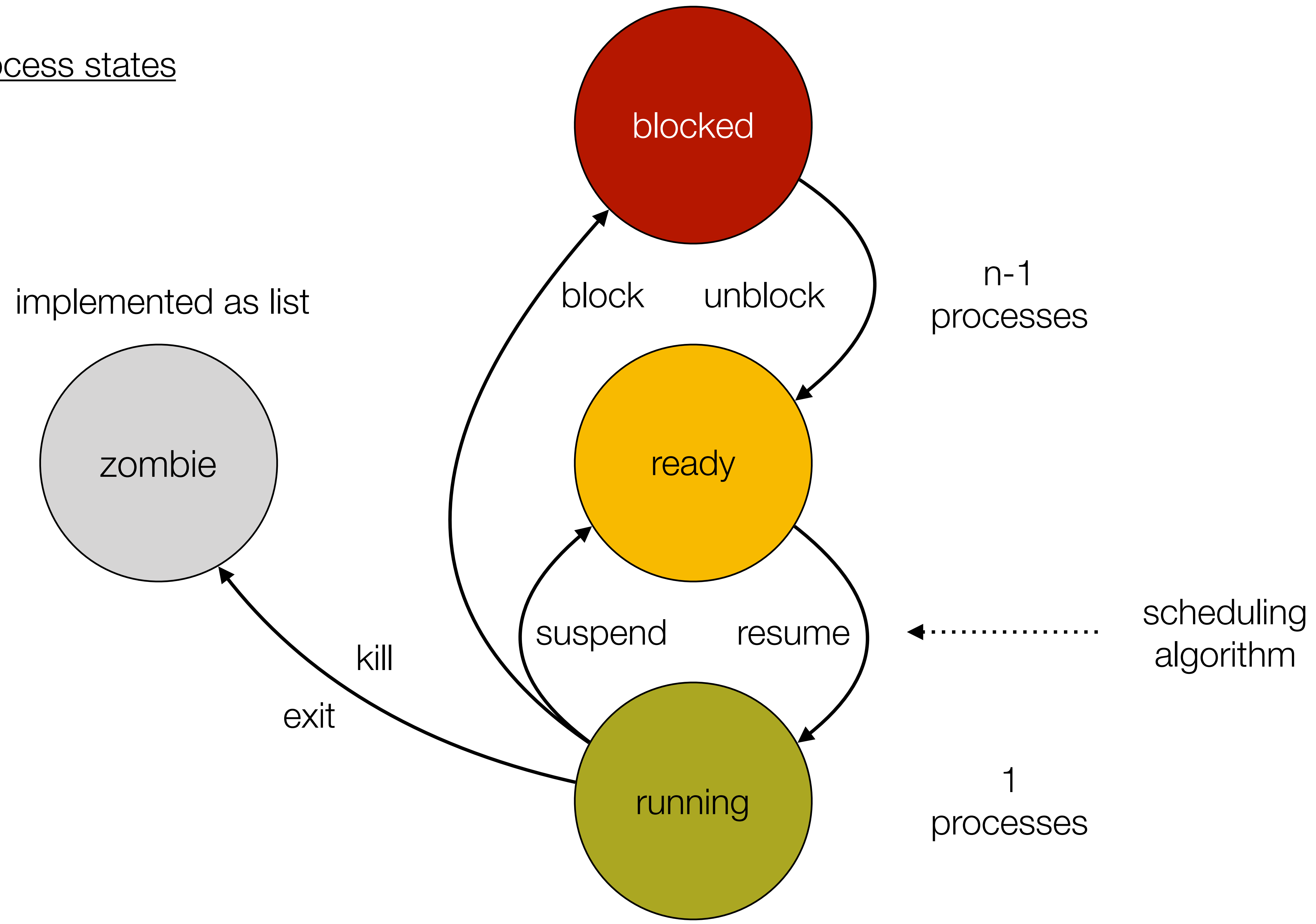
…the children could be terminated too or
they could become orphans”

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

process 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.*

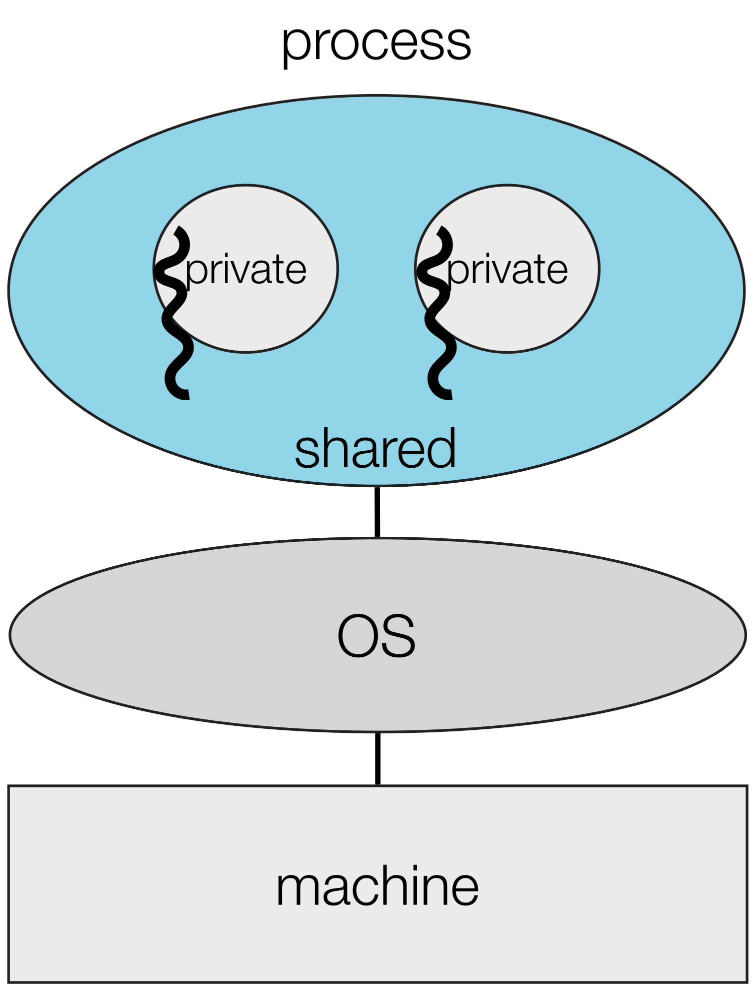
- can be seen light-weight processes
  - threads are generally part of a process → a process can have multiple threads of execution
  - carry less state information as they share part of their state
- threads **share** process state
  - parts of the memory (address space) and resources
- allow efficient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



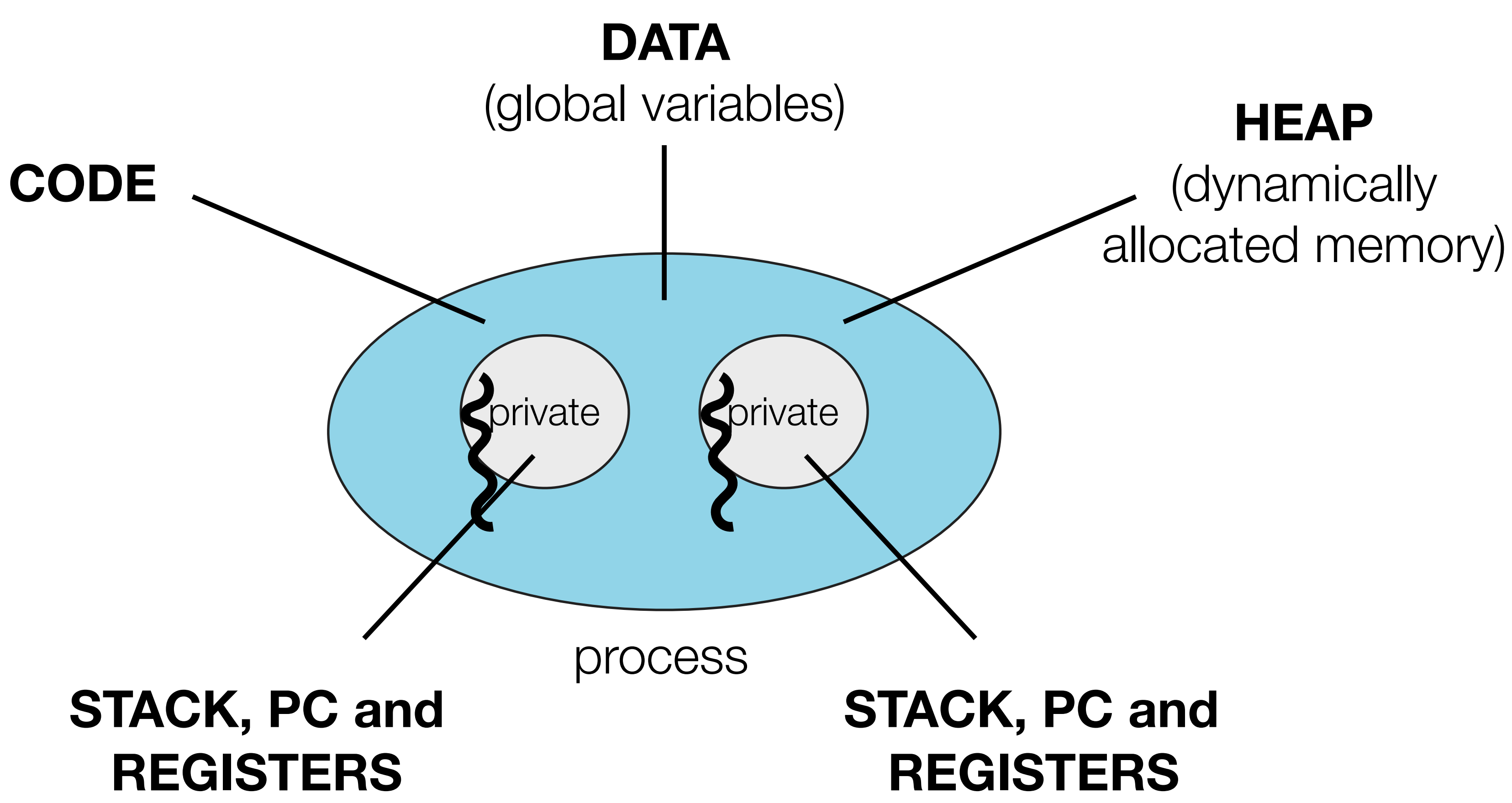
Process

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

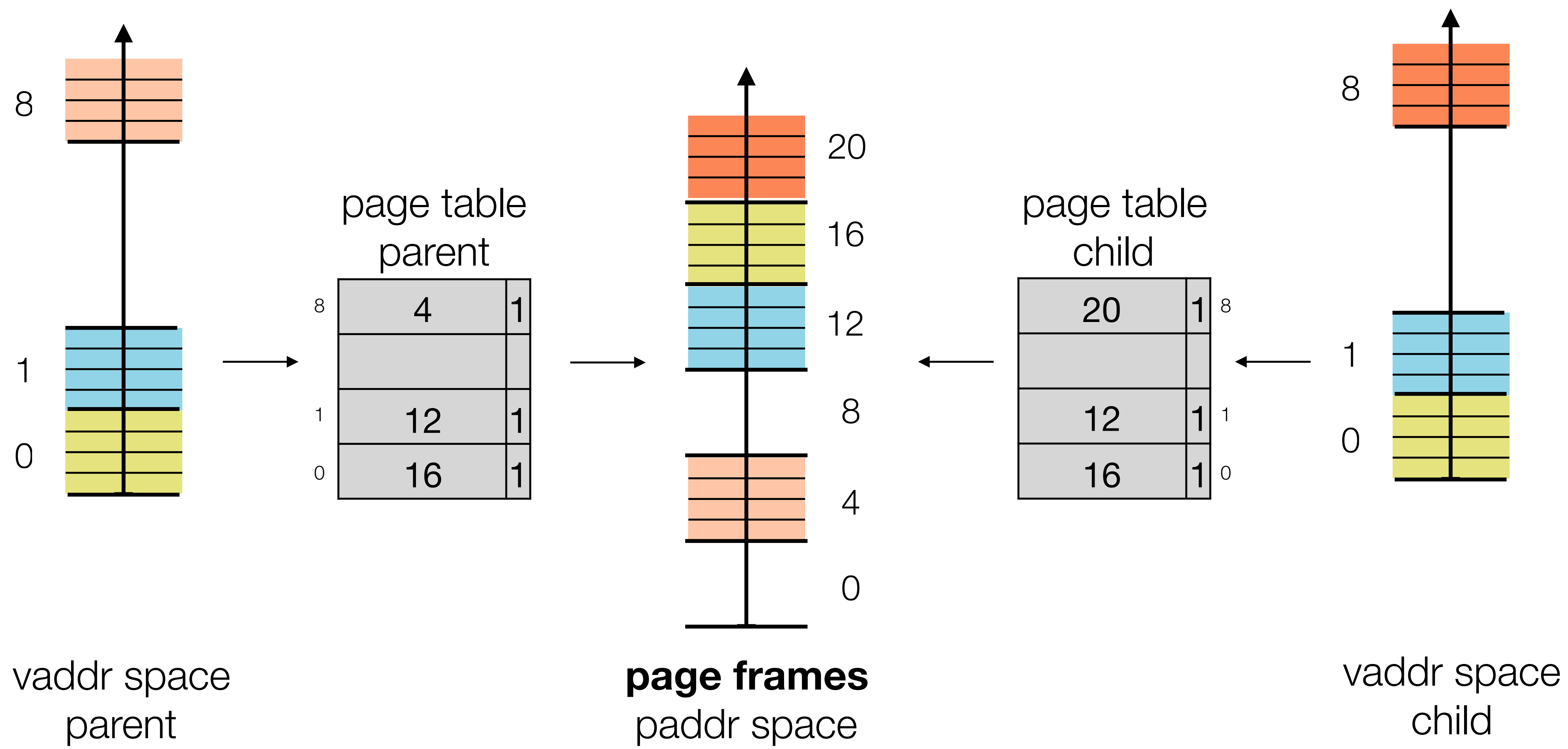
Threads What is shared / private?

- **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

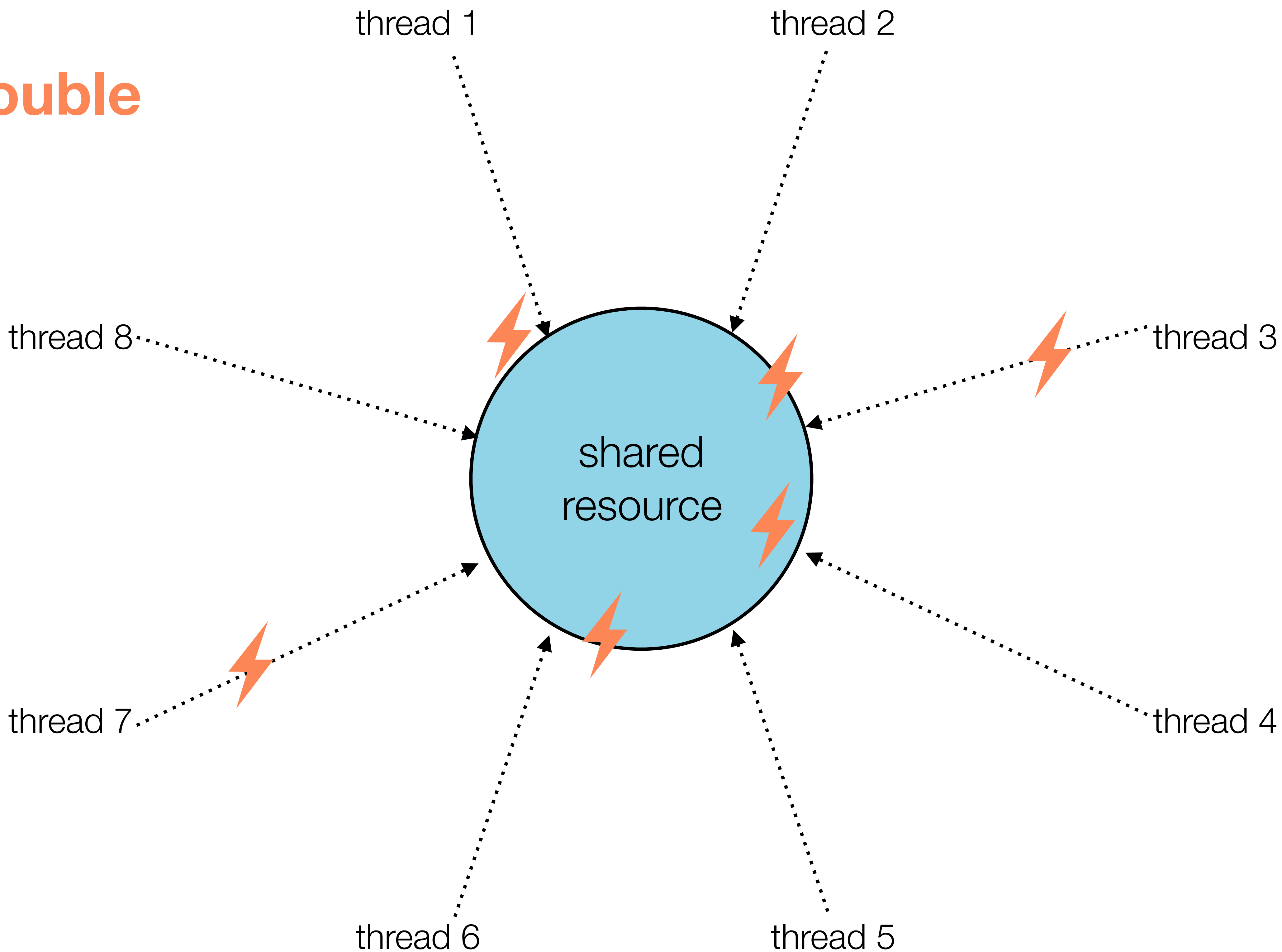
Syscall thread() Implementation Tips

- 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

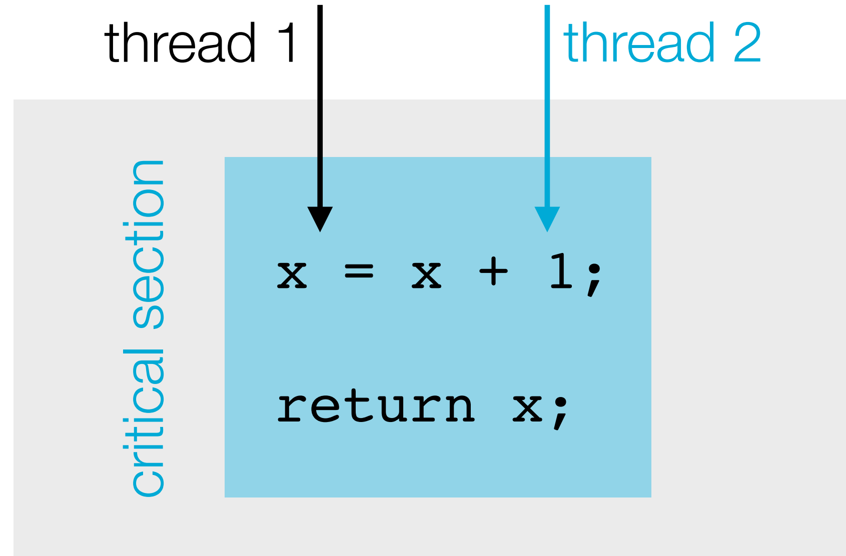
`x` is shared
and 2 threads try to
access it concurrently

critical section

```
uint64_t x;

uint64_main() {
   x = 0;

   thread();

   x = x + 1;

   return x;
}
```

$\mathtt{x} = ?$

Race Condition Shared State and Interleaving



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?
What we absolutely don’t want is:

'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)

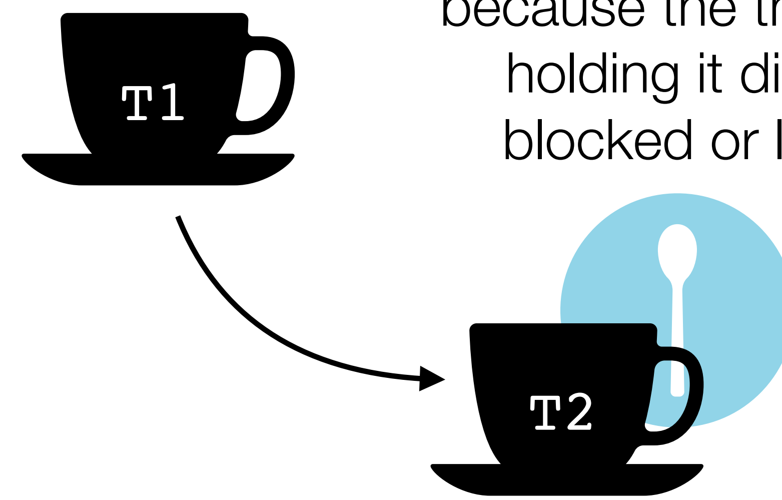
```
lock();
```

critical section

```
unlock();
```

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

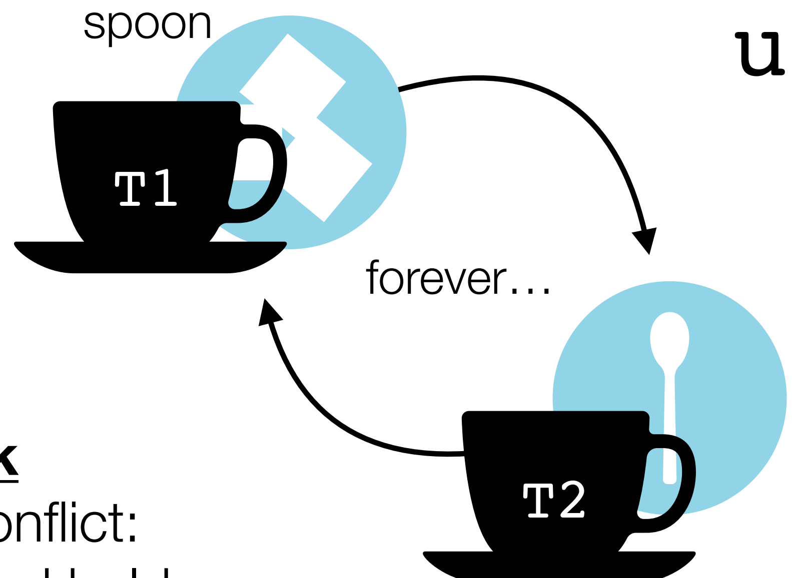
waits for spoon



steers forever

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

waits for spoon



waits for sugar

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 concurrent data structure is a way to ***organize data in shared memory*** *for access by multiple threads.*

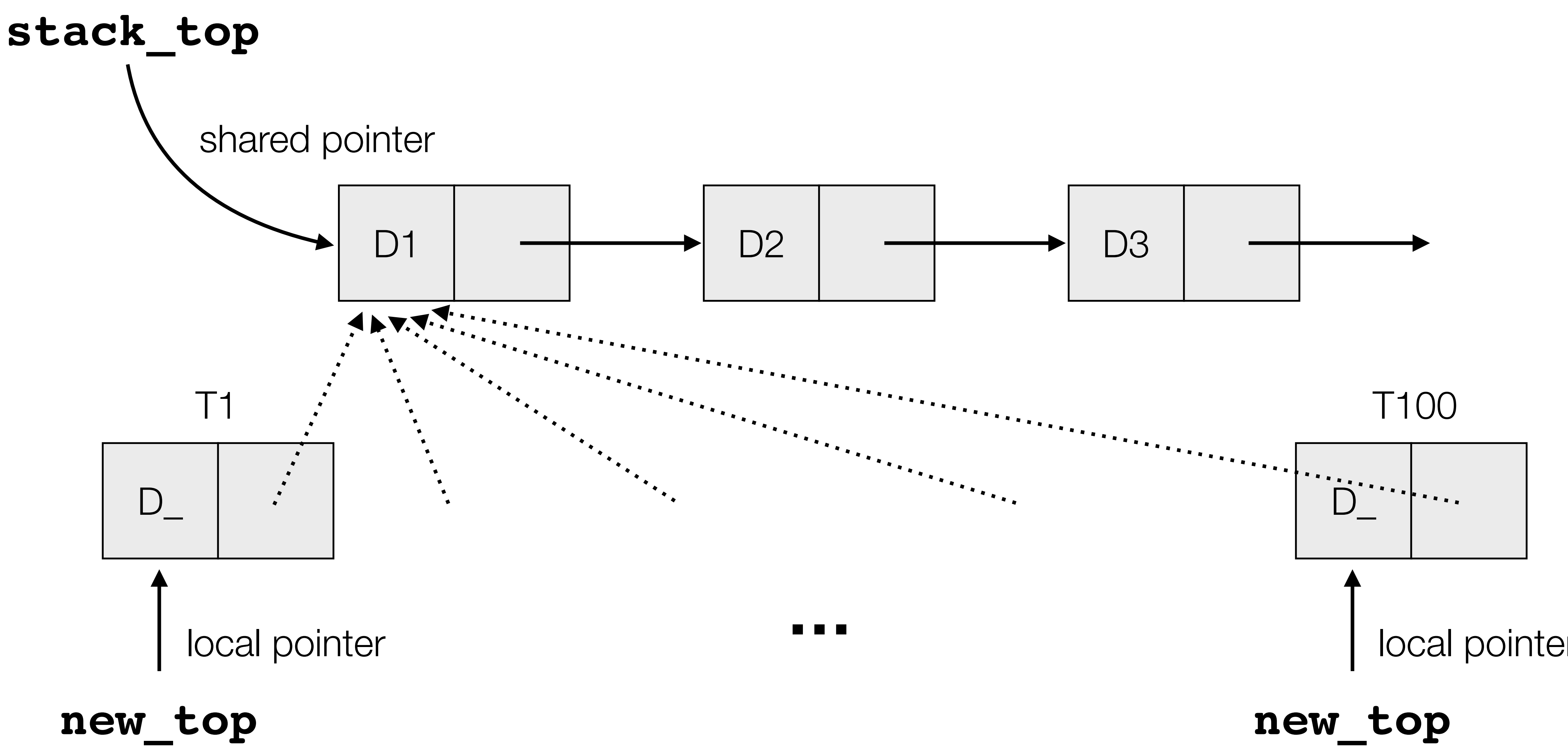
- concurrent access + interleaving = potentially unexpected outcome
    - mechanisms to ensure 'correct' behavior are required
- designing and verifying effective concurrent data structures is difficult
    - safety properties → nothing bad happens
    - liveness properties → something good keeps happening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

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**

critical section

1. create new element
2. set next-pointer to top element - thread local
3. set shared top pointer to new element

DO NOT SPLIT

- **pop**

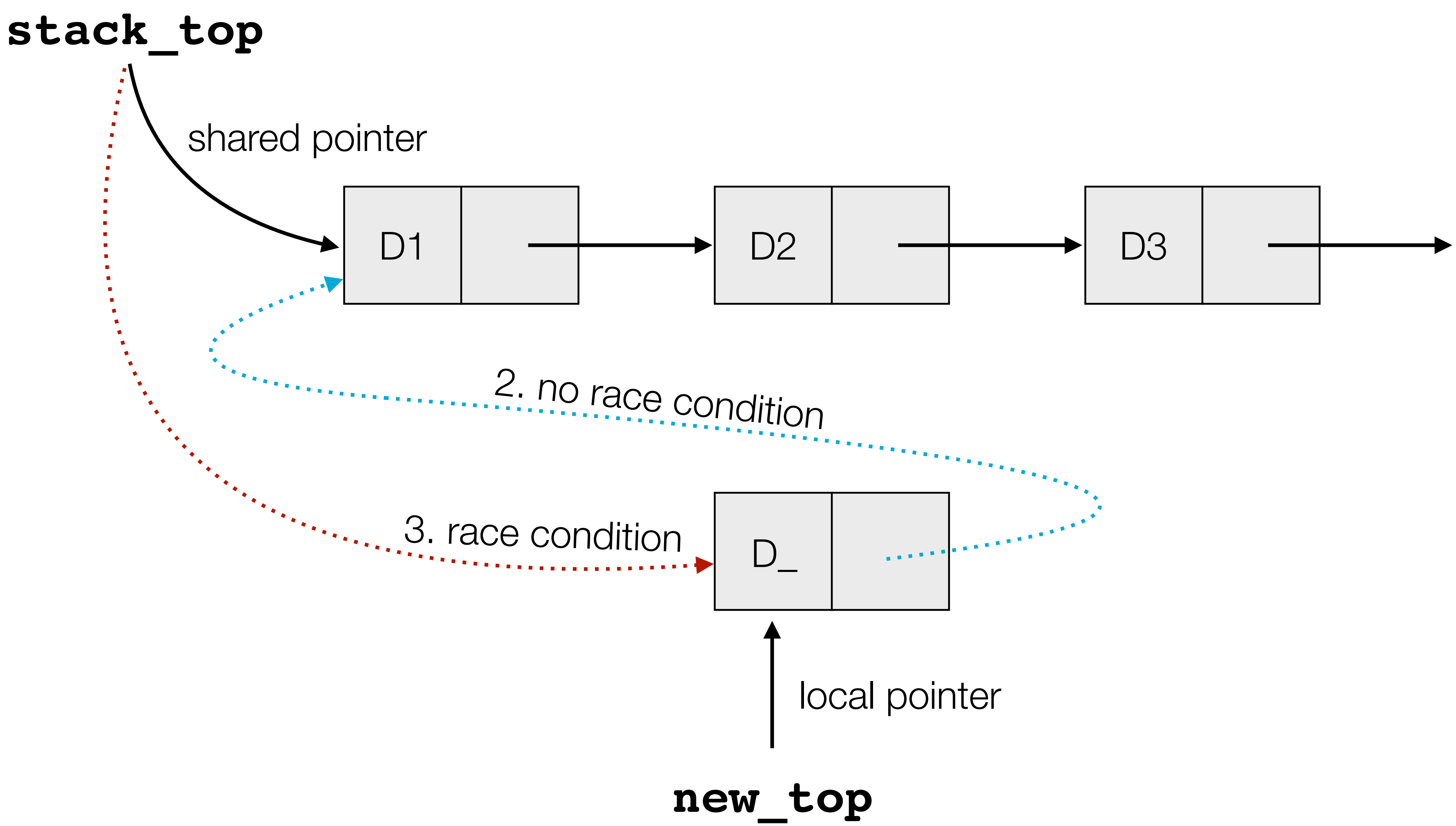
critical section

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

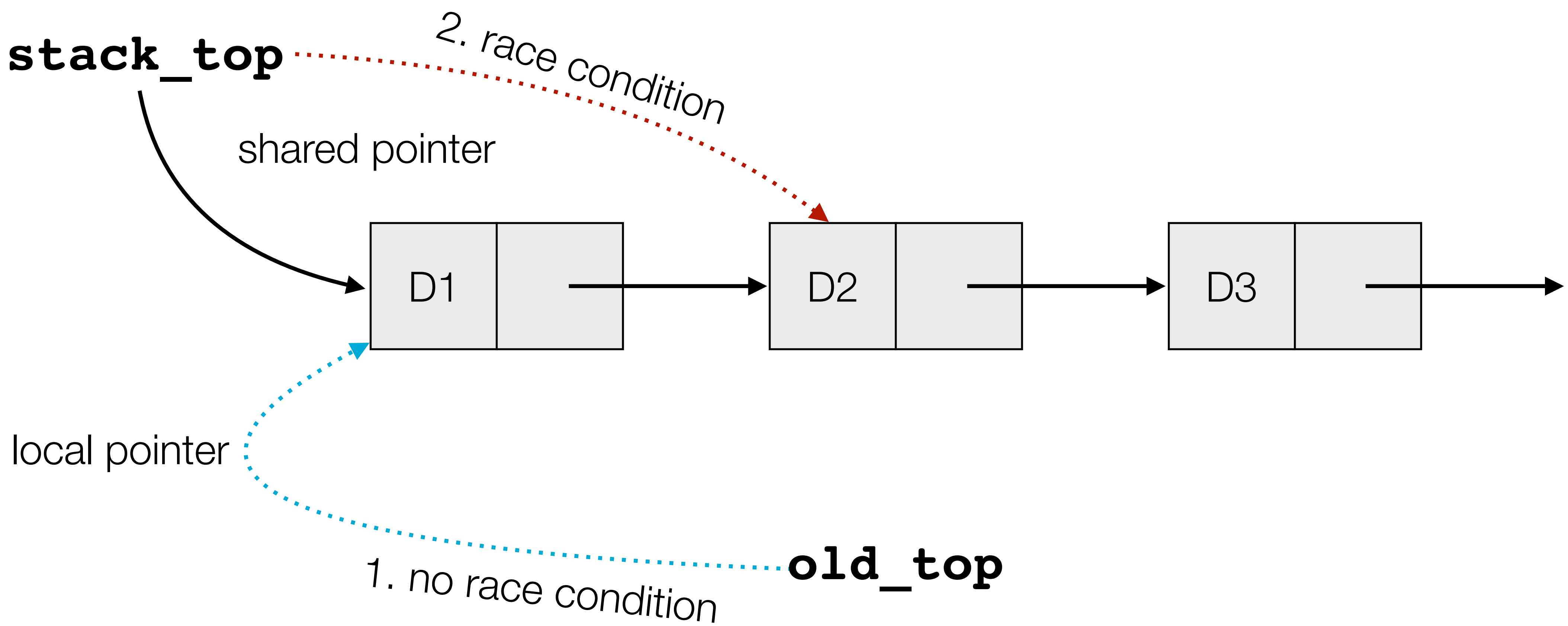
DO NOT SPLIT

- we need atomicity → one single moment in which push/pop takes effect (linearizability)

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

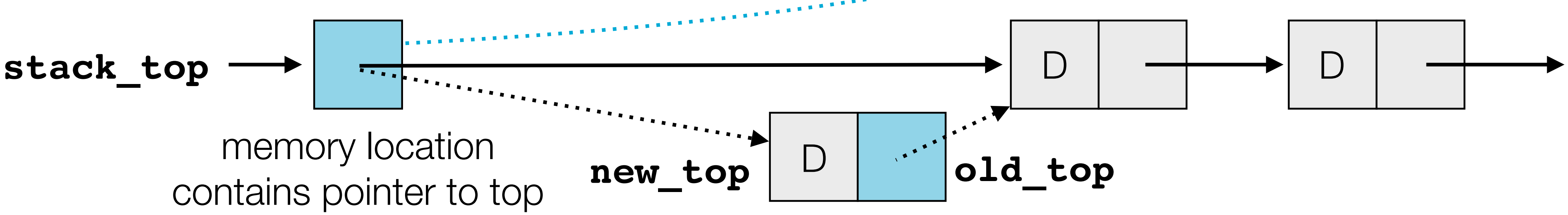
```
void push(uint64_t number) {

   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



Lock-Free Implementation Pop

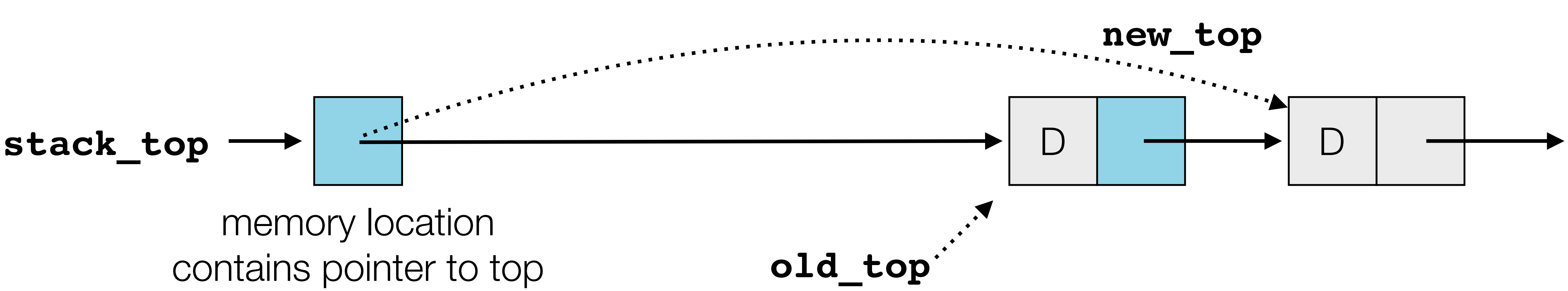
```
uint64_t pop() {

   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?”

“Well, 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 $\Rightarrow$ ***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

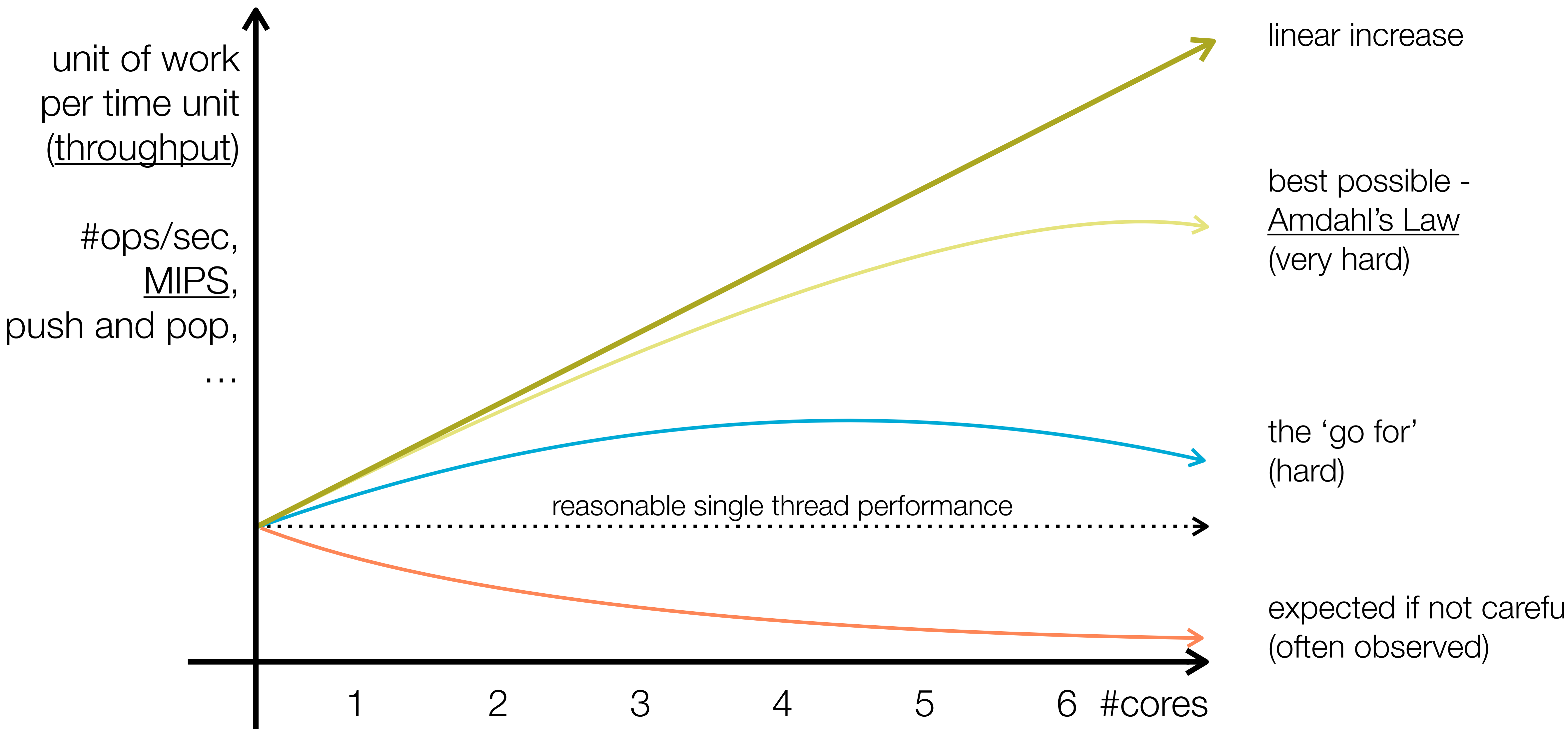
- careful with deallocating/reusing popped nodes
  - dangling references → other nodes might still have the reference (in `old_top`)
  - first free, then malloc → might return the same pointer again
- **ABA**
  - the pointer is the same ⇒ nothing has changed behind it
- **solution**
  - 100%: do not 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 multi-core machines.”

“To get some performance advantages…?”

Performance Bottleneck

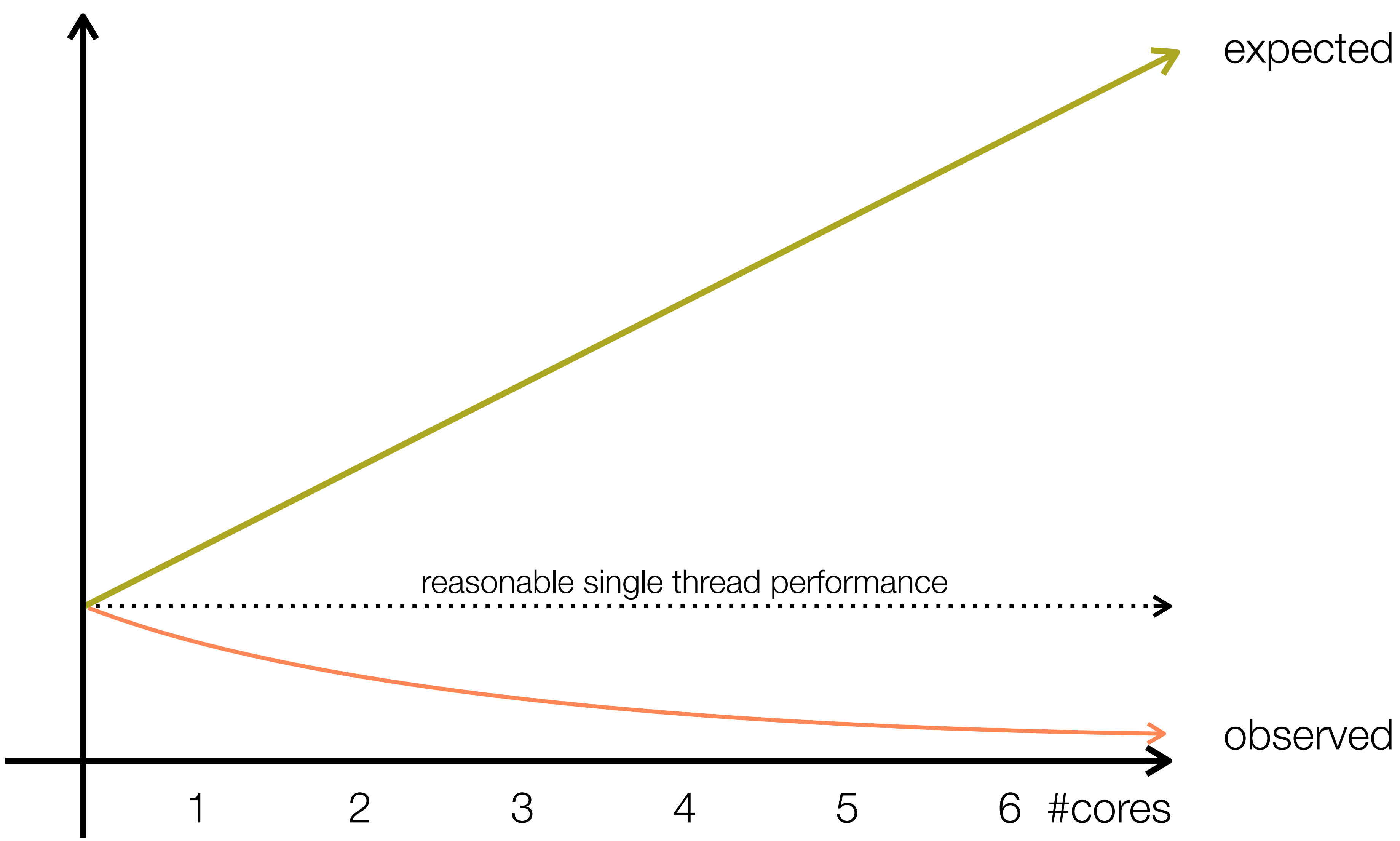


Amdahl’s Law

- 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 executes potentially in parallel (< 100%)
  - code that **must** executes 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

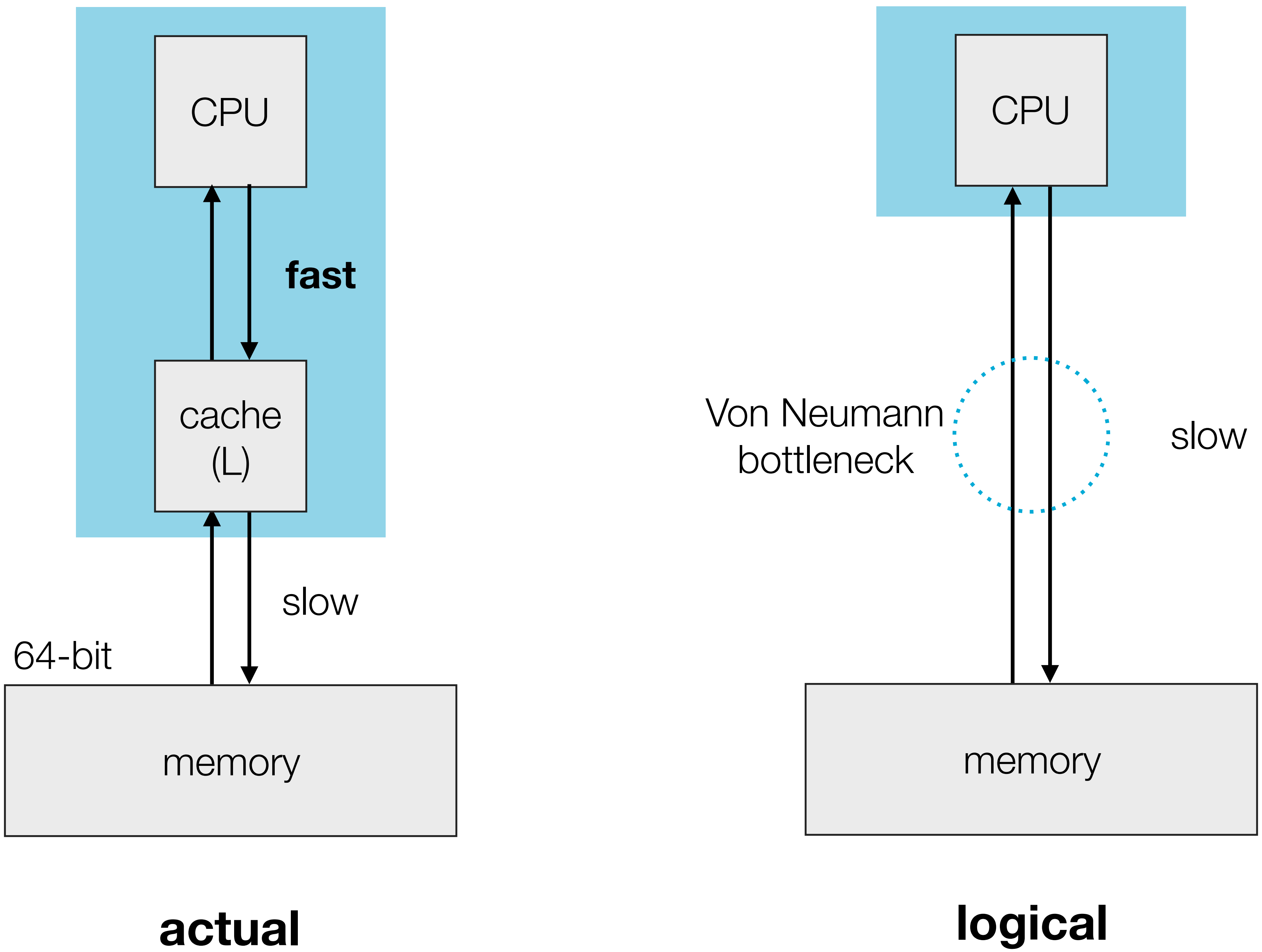


Cache

- **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

- <u>cache</u> should be **logically invisible**



Cache Read

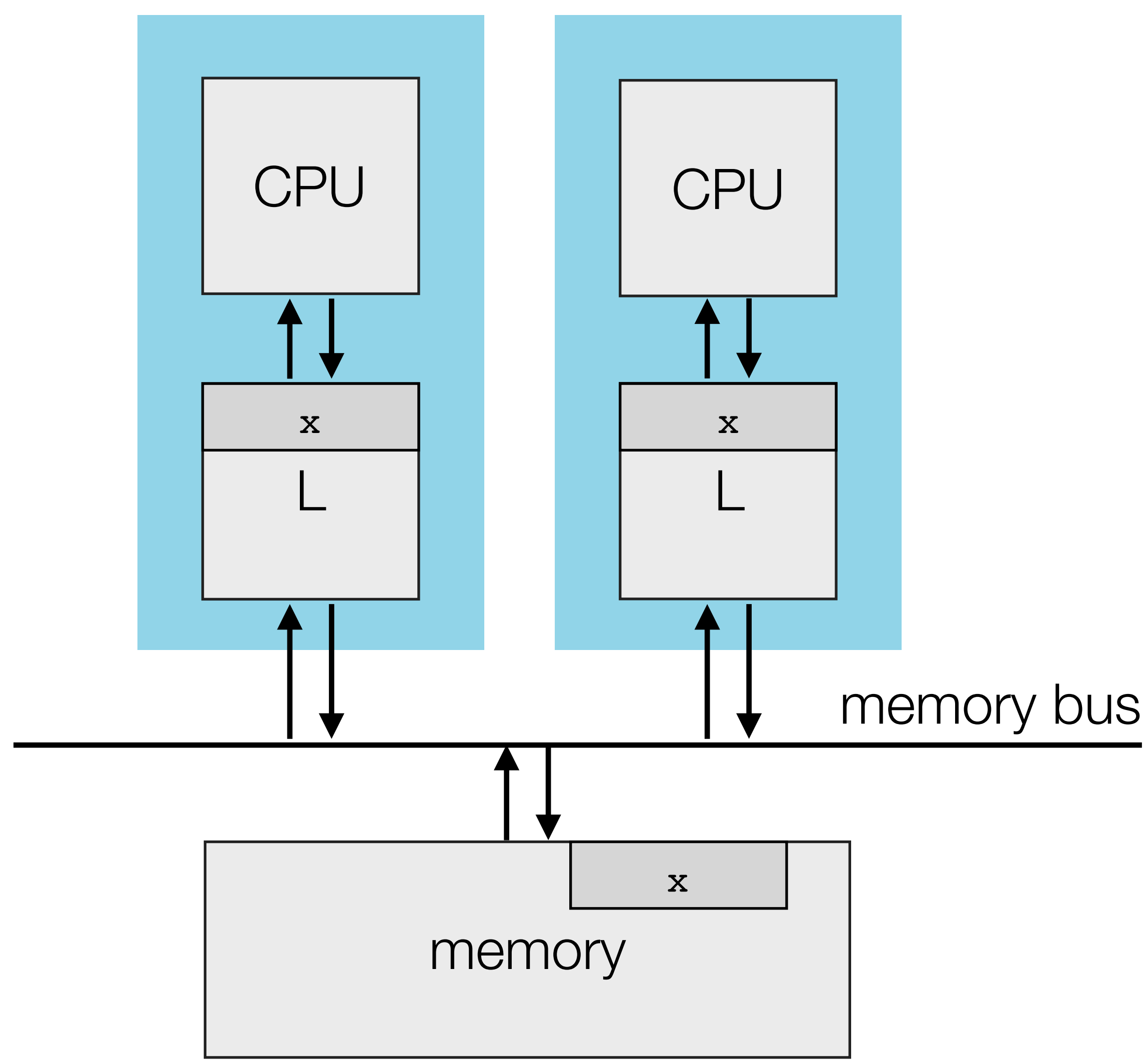
- search the cache for value → is it caching the requested address? (still faster than main memory access)
- **<u>cache miss</u>**
  - value not currently in the cache
  - push down-read to main memory → expensive
- **cache hit**
  - value found
  - no activity on slow memory line → cheap

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

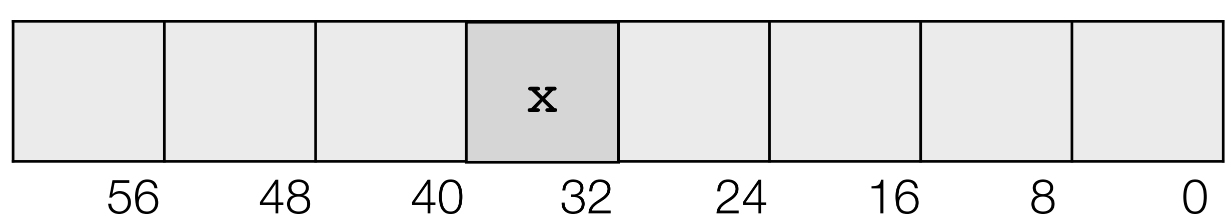
Problem with Multiple Cores

- one thread keeps updating the `x` in its cache
- another thread wants to read `x`
  - sees `x` is 'old'
  - force write-down (<u>cache invalidation</u>)



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



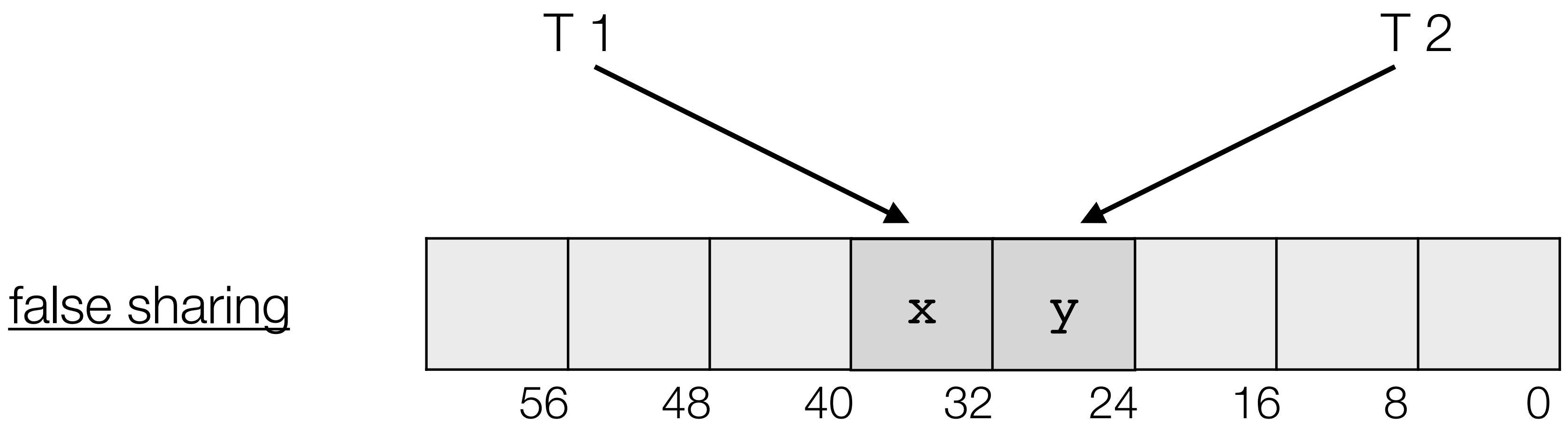
cache line

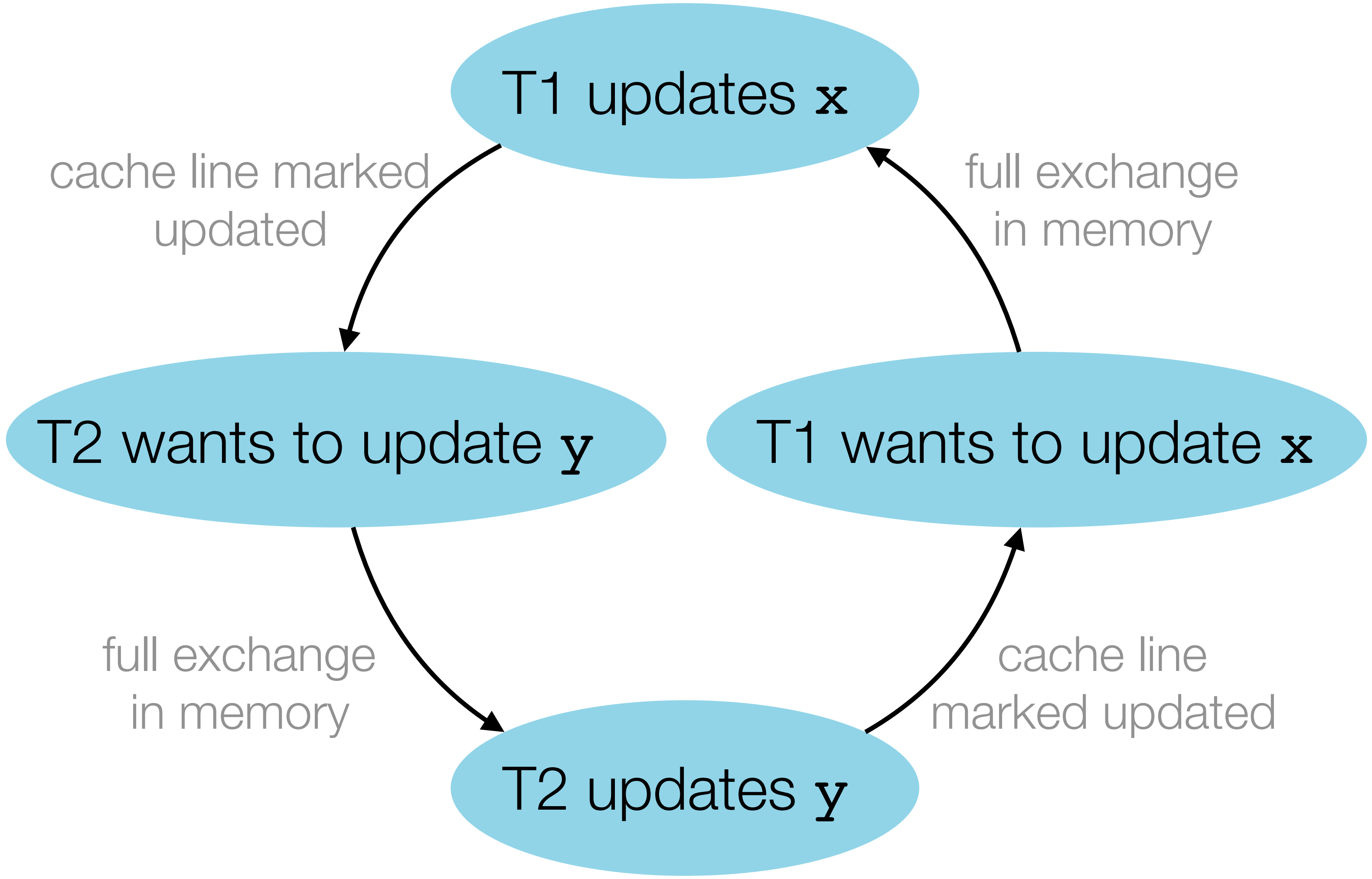
“I think I know what’s happening…

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.”

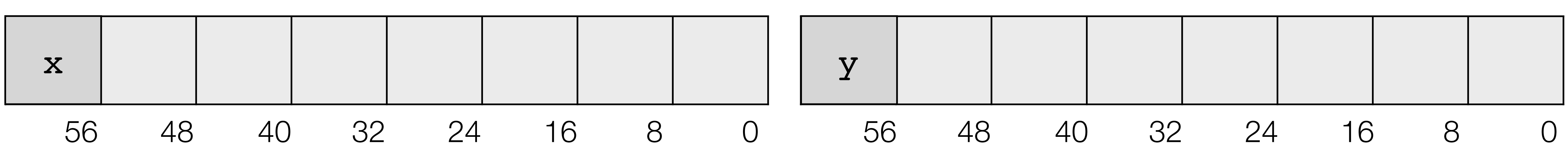
False Sharing





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)