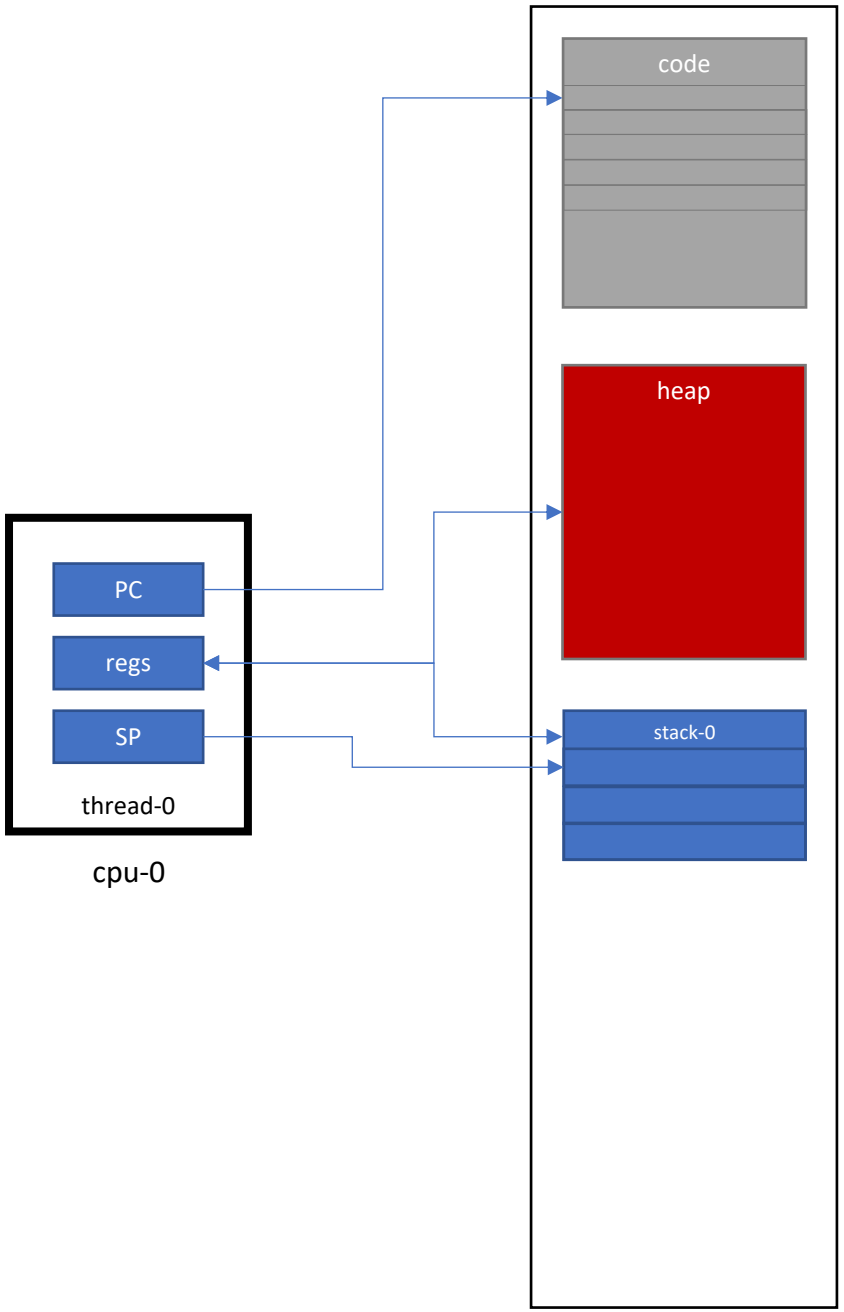
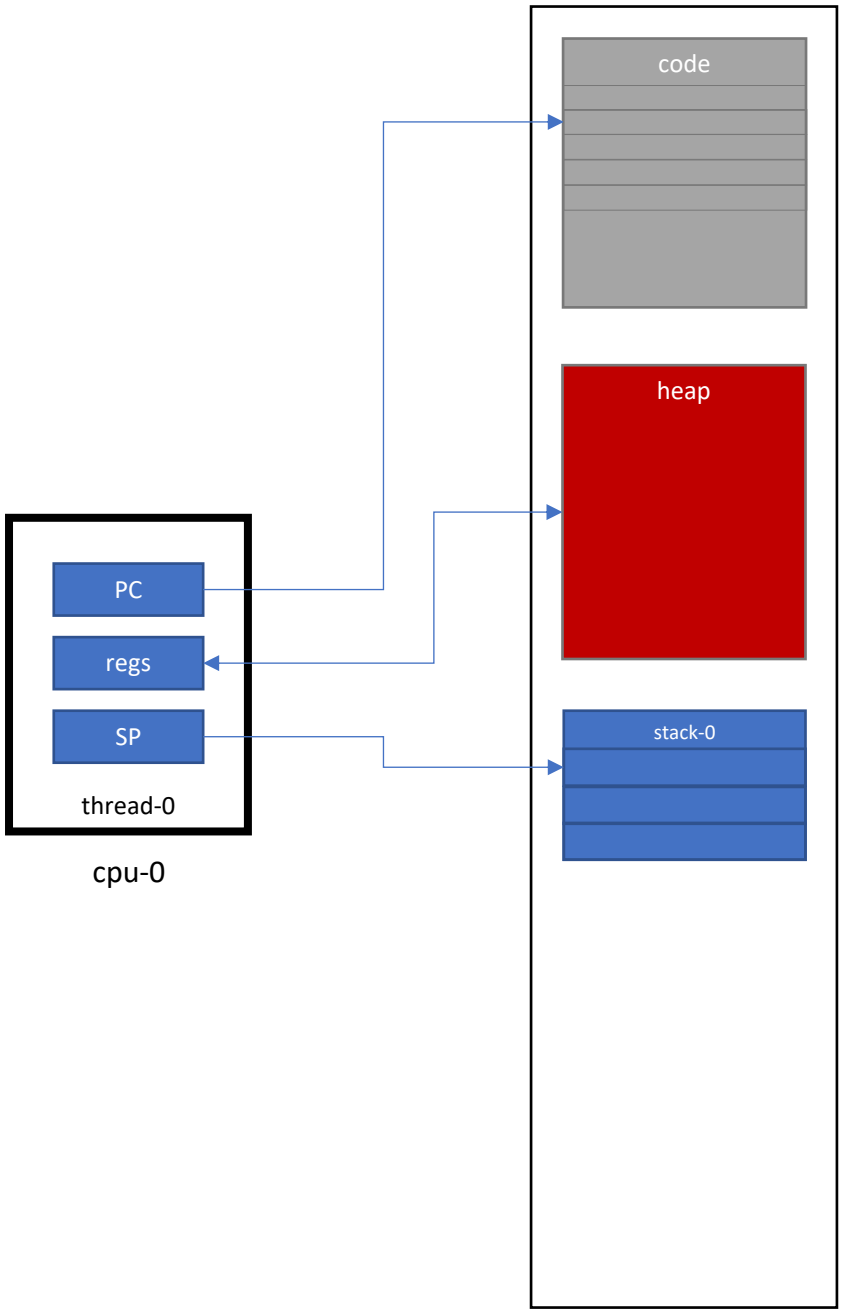
Threads

Concurrent Programming


code
heap
PC
regs
SP
thread-0
cpu-0
stack-0


code
heap
stack-0
PC
regs
SP
thread-0
cpu-0

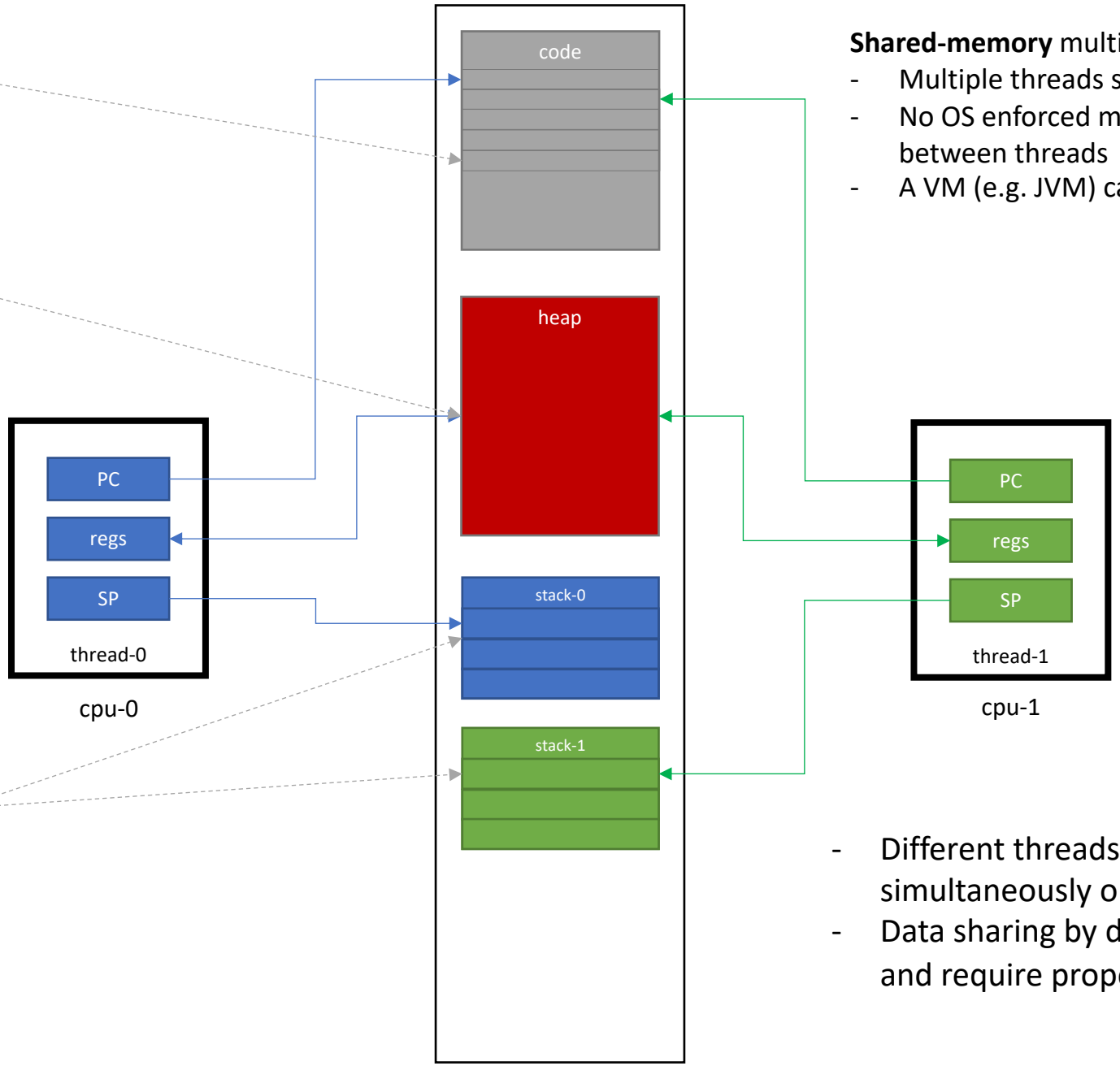
Same **code** block

Same **heap** block

Distinct **stack** blocks

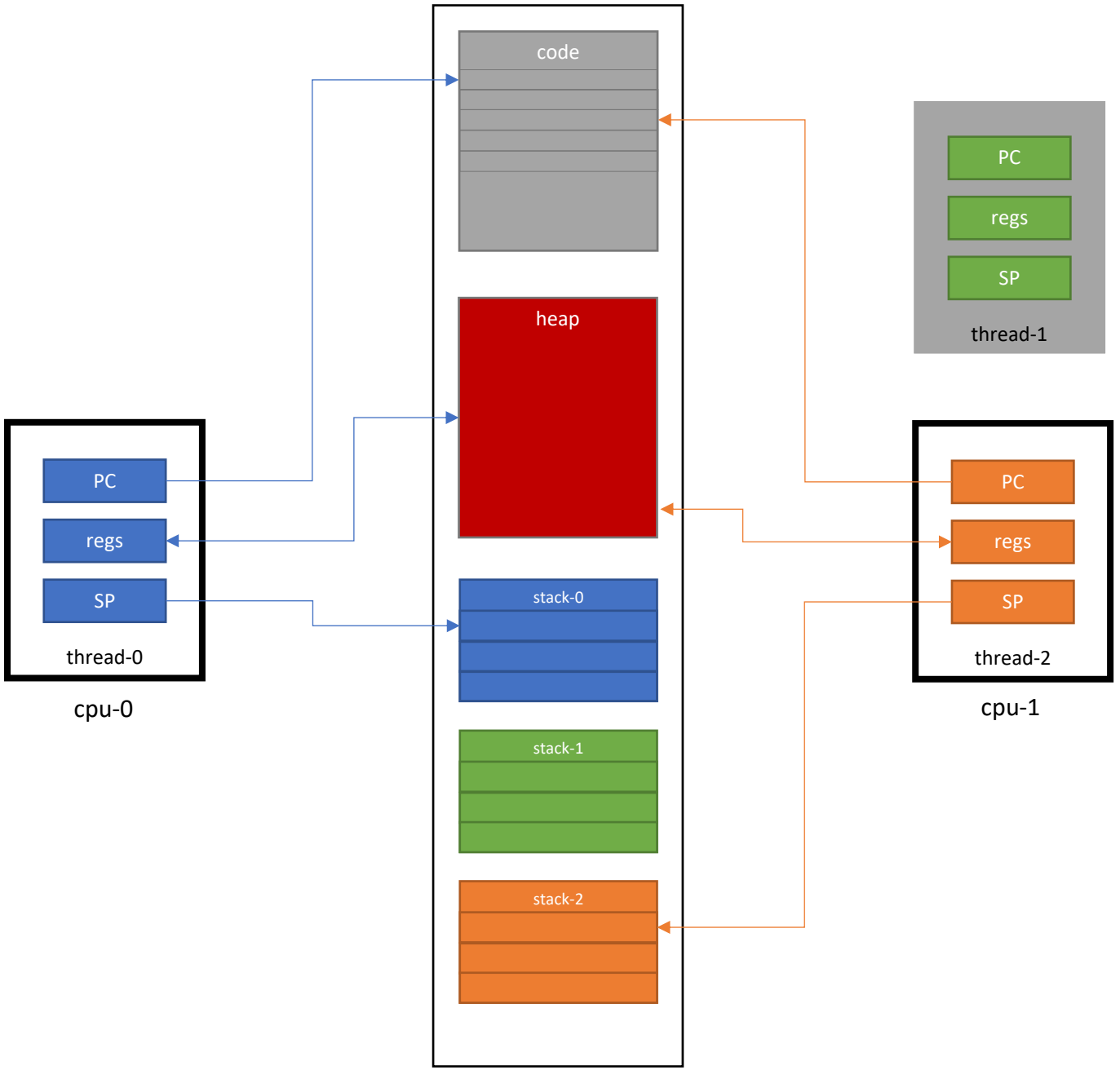
stack contains:
- return addresses
- local variables
- function parameters

specific to each thread



Shared-memory multi-threading:
- Multiple threads share the same code and heap
- No OS enforced memory protection mechanism between threads
- A VM (e.g. JVM) can impose some restrictions

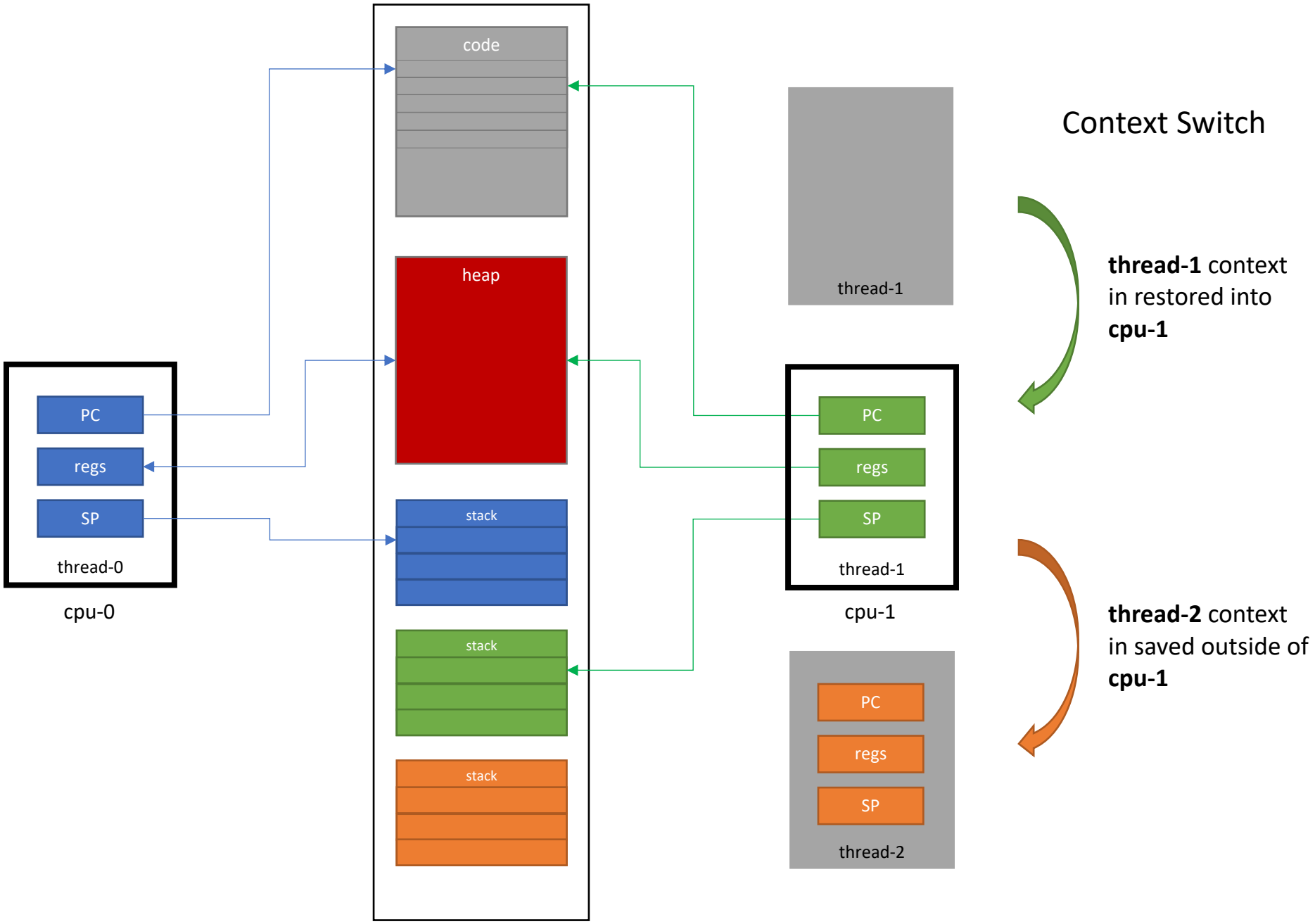
- Different threads can be *running* simultaneously on different CPUs
- Data sharing by different threads is *hazardous* and require proper *synchronization* techniques



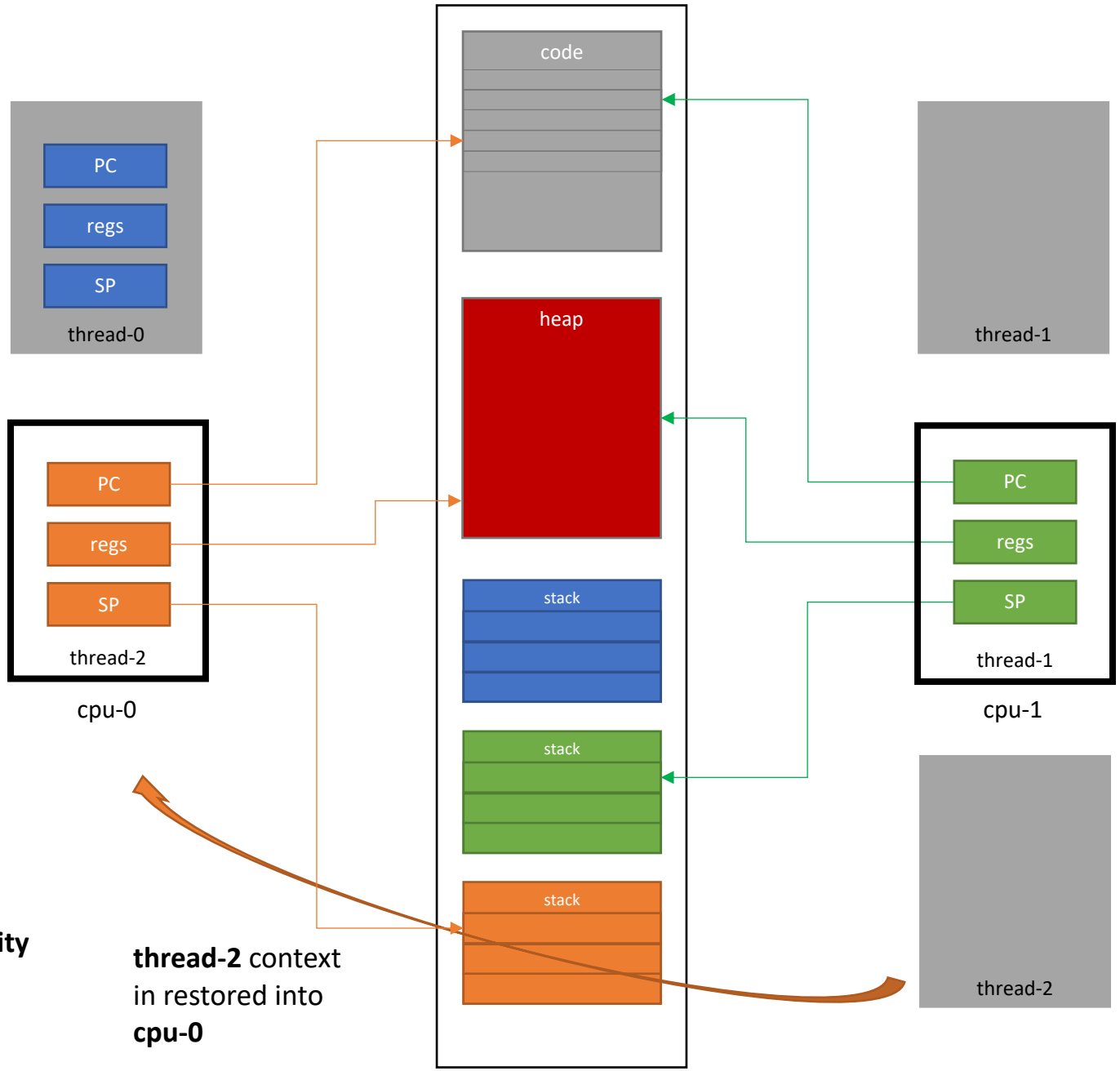
There can be more threads than processors

- **thread-1 context** is saved outside of **cpu-1**
- **thread-2** starts executing on **cpu-1**

thread-2 has its own stack, separate from **thread-1** stack


code
heap
stack
stack
stack
thread-1
PC
regs
SP
thread-0
cpu-0
PC
regs
SP
thread-1
cpu-1
PC
regs
SP
thread-2
Context Switch
thread-1 context in restored into cpu-1
thread-2 context in saved outside of cpu-1

thread-0 context
in saved outside of
cpu-0



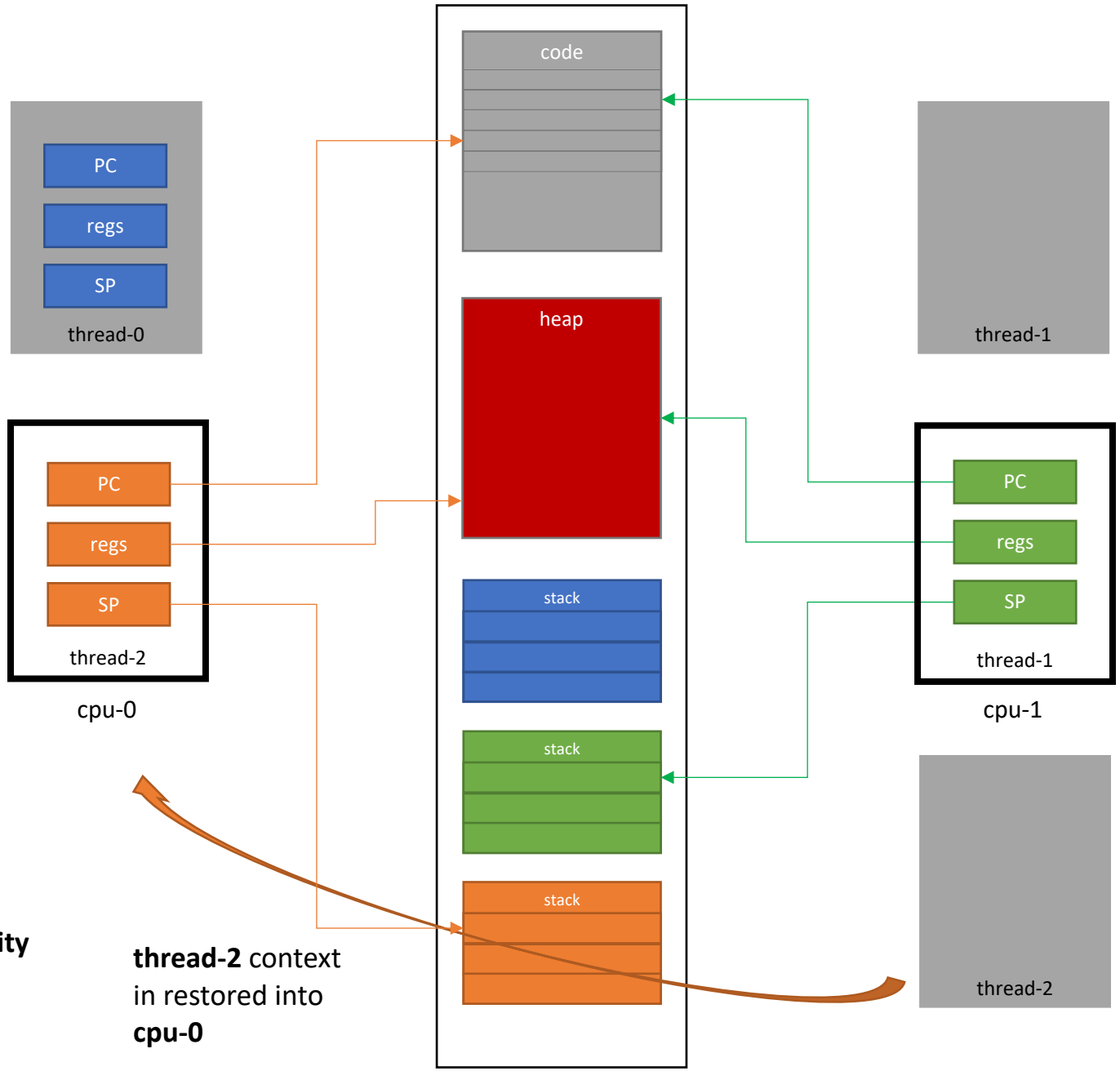
Context Switch

Threads may run in different CPUs,
i.e., threads may not have **CPU affinity**

thread-2 context
in restored into
cpu-0

thread-0 status is *not running*
- ready – able to run when a CPU is available
- not-ready/blocked/waiting – not able to run, waiting for *something* to happen

thread-0 context in saved outside of **cpu-0**



thread-1 status is **running** (on cpu-1)

Context Switch

Threads may run in different CPUs, i.e., threads may not have **CPU affinity**

thread-2 context in restored into **cpu-0**

thread-2 status is **running** (on cpu-0)

Scheduling

- Conceptually, there is a list with all the **ready** threads
- Scheduling is the process
  - Deciding which **running** threads should transition to **ready** (i.e. *loose* the CPU)
  - Deciding which **ready** threads should transition to **running** (i.e. *gain* the CPU)
  - Performing the associated context switches
- When is scheduling performed?
  - When a thread calls the scheduling code (e.g. indirectly via a function call)
  - When an interrupt (system timer) occurs and calls the scheduler
    - This means that code running on a thread cannot control the points where the scheduling occurs and the threads is switched out of the CPU.
    - I.e. a thread can be switched out at any assembly instruction boundary.

Scheduling

- Scheduling is based on
  - Thread **status**: only **running** and **ready** threads are eligible to go/keep **running**
  - Thread **priority**: a way of sort to prefer some threads in relation to others
    - Because some threads may be more “important” than others
  - Thread **execution time**: how long has a thread been in the running state
    - To do time multiplexing between ready and running threads.

OS threads vs. VM threads

- VM-based programming environments, such as the JVM and .NET, provide a way to create threads associated with managed code.
- Typically, these *managed threads* are implemented using OS threads
  - 1-1 model: one managed thread uses one OS thread.
  - Scheduling is performed by the OS, which controls the OS threads.

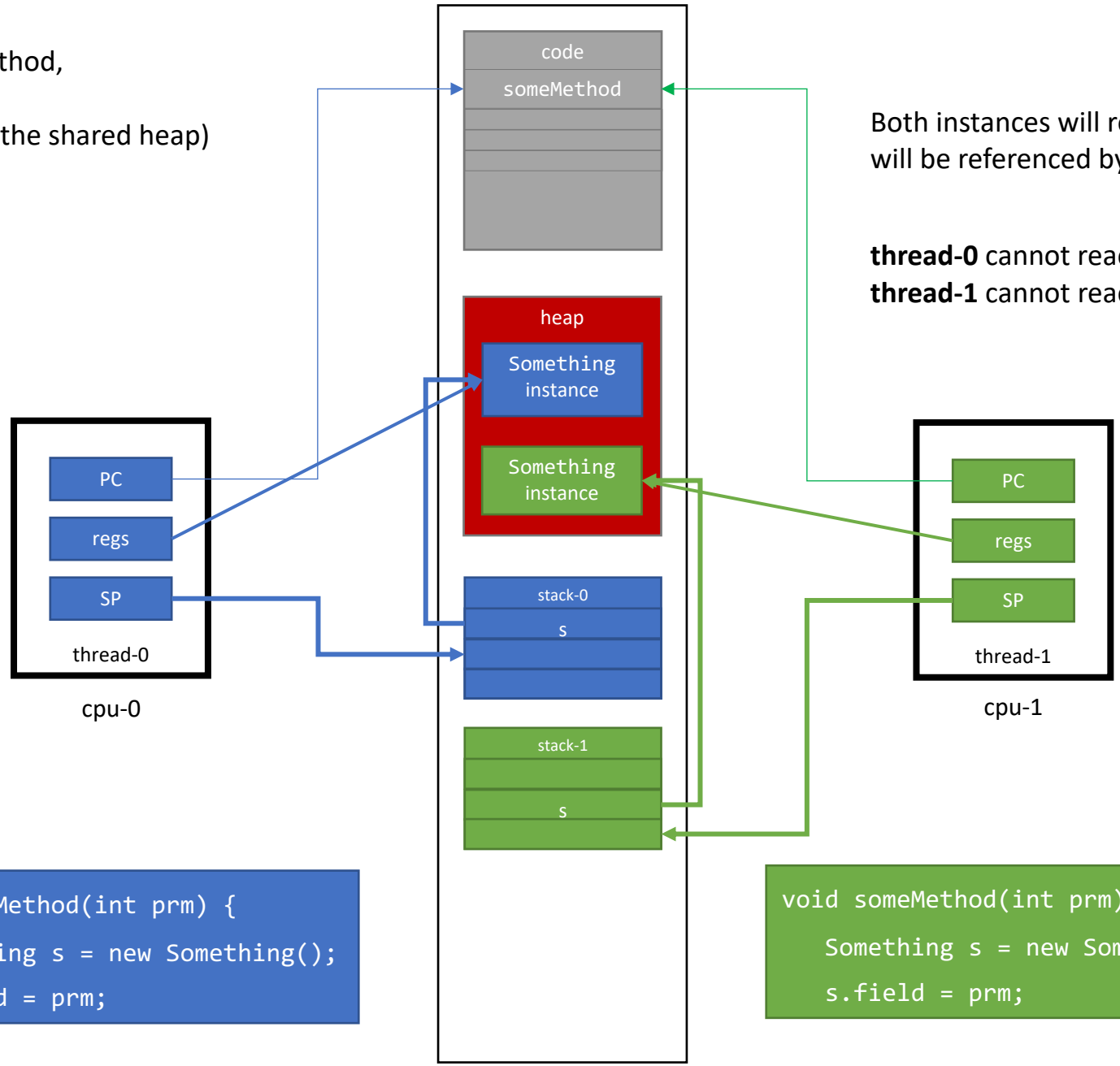
- In the JVM, this model may change with Project Loom - https://openjdk.java.net/projects/loom/

Data sharing

- A shared-memory multi-threading model doesn't mean all data is effectively shared.
- Identifying and avoid unnecessary sharing is a very important skill

- Let's look at some examples…

Two threads running the same method,
which creates a local variable
of a reference type (i.e. hosted on the shared heap)



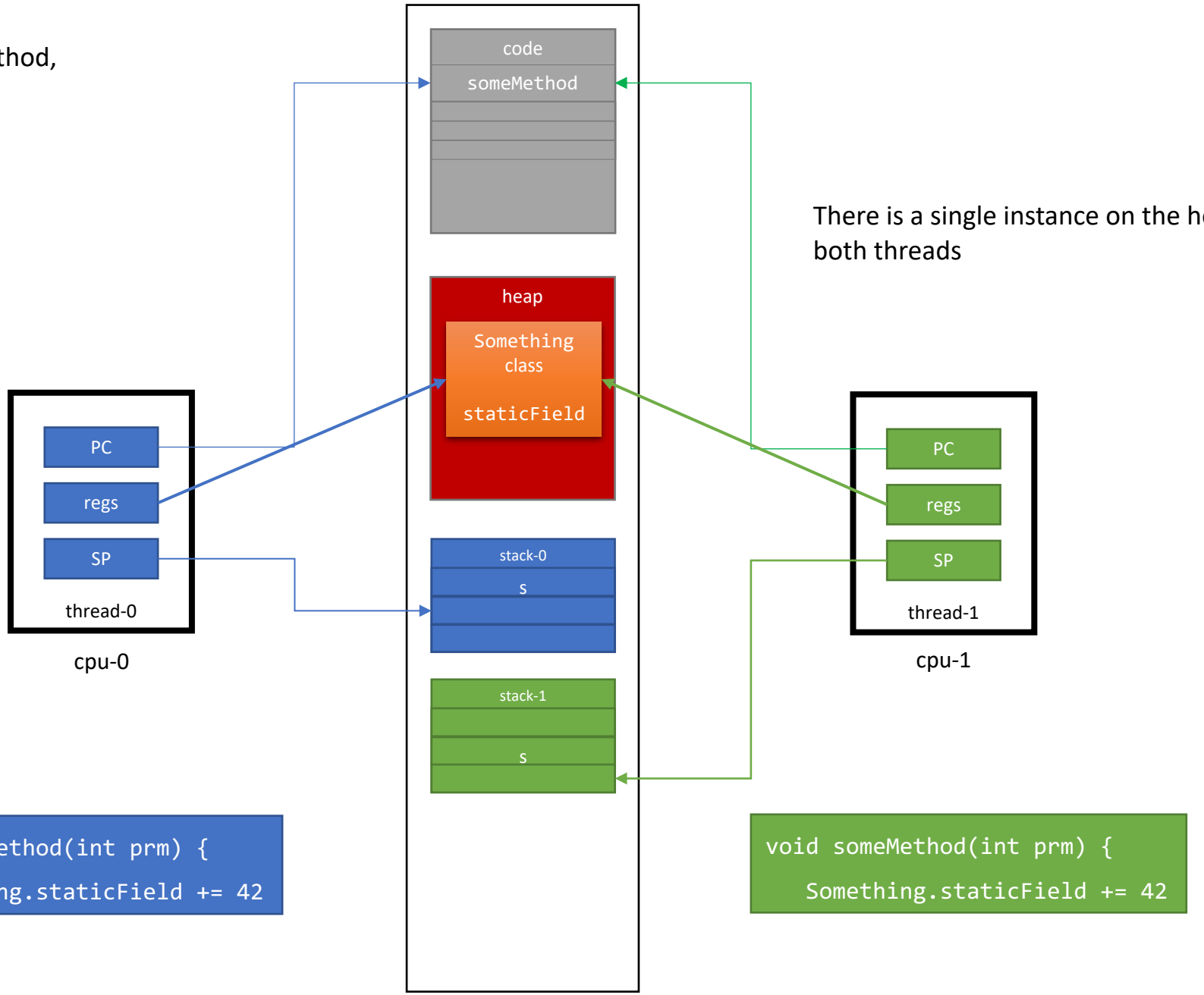
Both instances will reside in the shared heap, however will be referenced by references in distinct stacks.

thread-0 cannot reach the green instance
thread-1 cannot reach the blue instance

```
void someMethod(int prm) {
   Something s = new Something();
   s.field = prm;
```

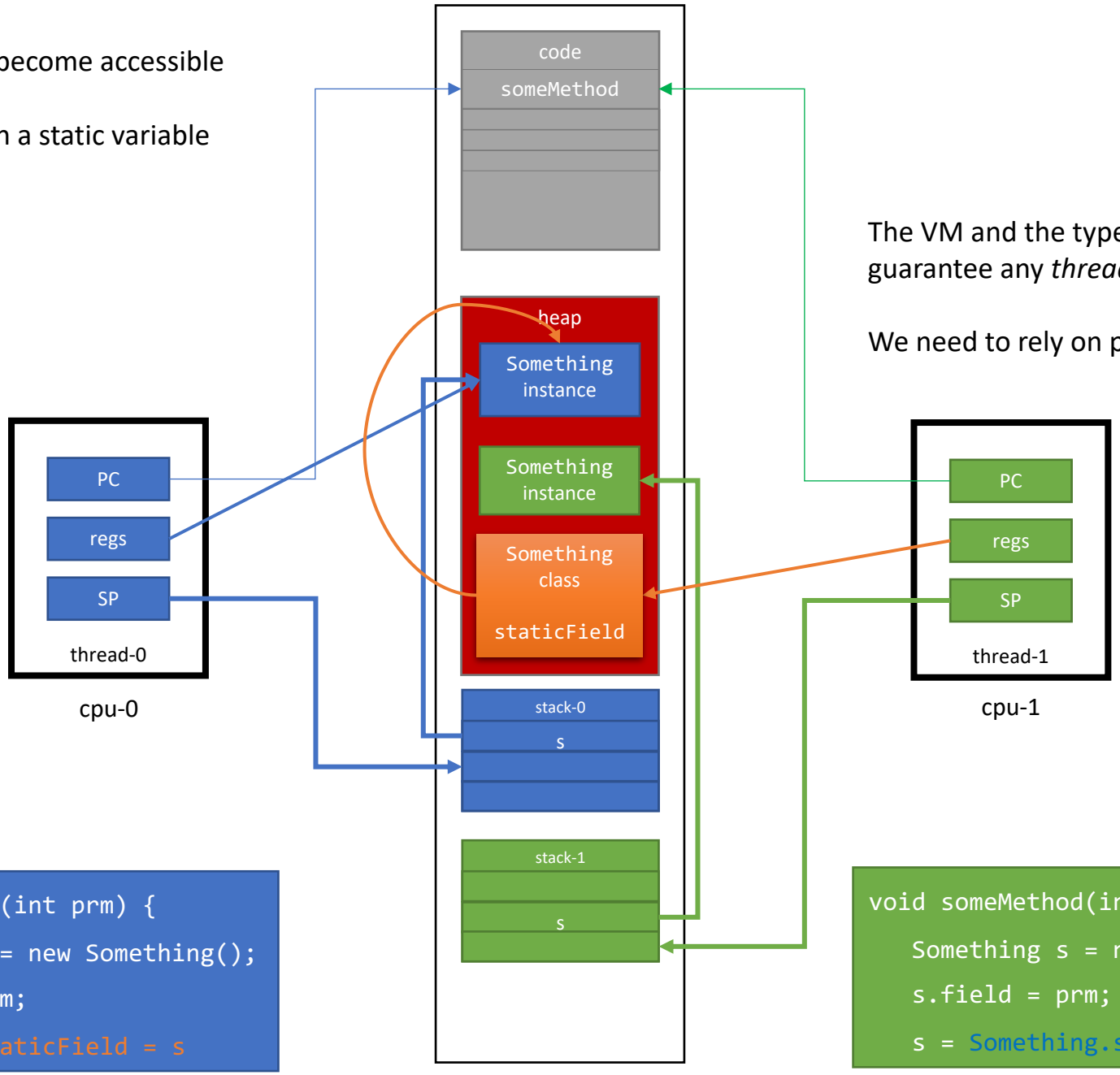
```
void someMethod(int prm) {
   Something s = new Something();
   s.field = prm;
```

Two threads running the same method,
which accesses a static variable
(i.e. hosted on the shared heap)



There is a single instance on the heap, accessible by both threads

References to object can *leak* and become accessible by other threads
E.g. by a thread publishing them on a static variable



The VM and the type system will not enforce or guarantee any *thread confinement*

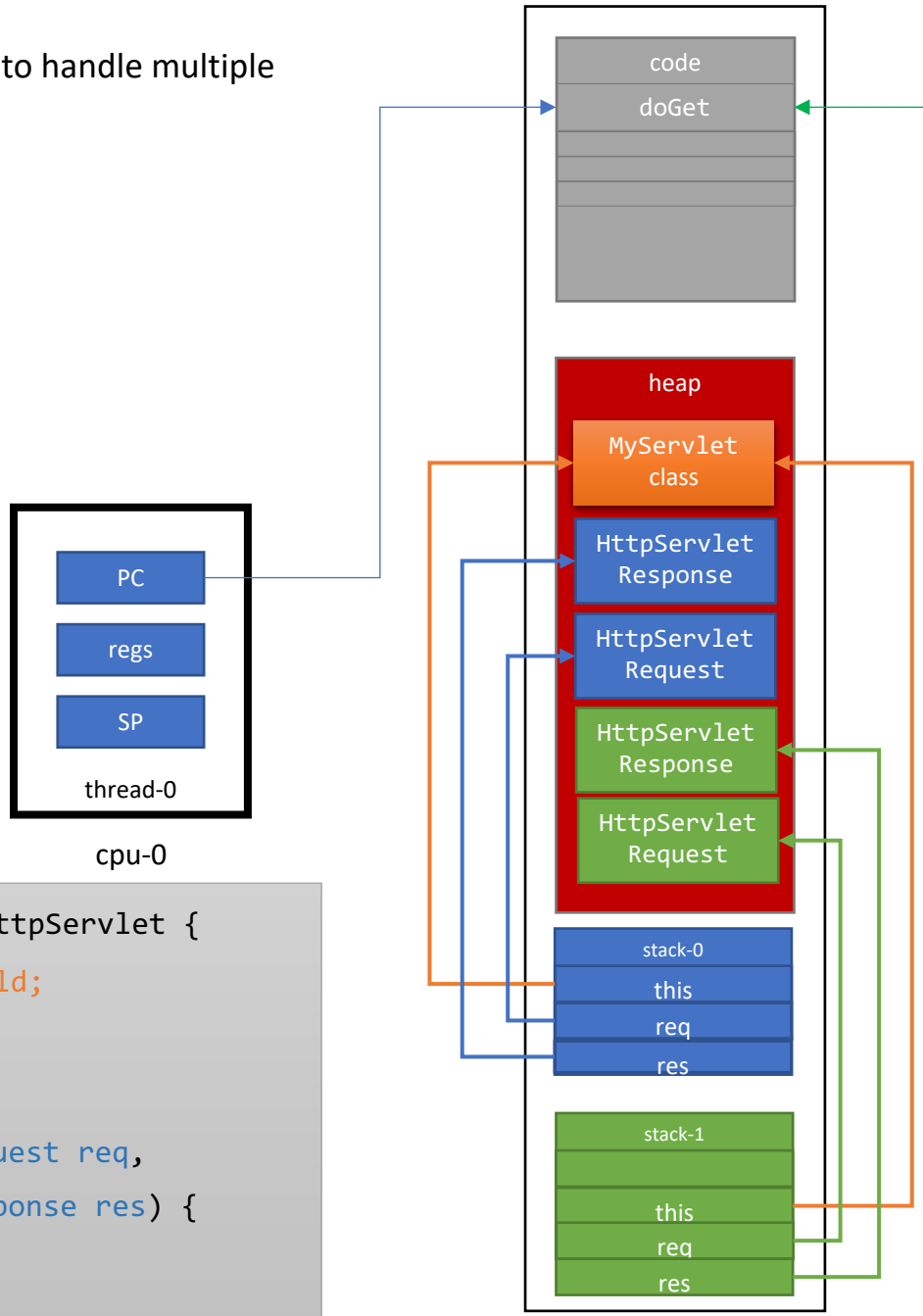
We need to rely on proper code organization

```
void someMethod(int prm) {
   Something s = new Something();
   s.field = prm;
   Something.staticField = s
```

```
void someMethod(int prm) {
   Something s = new Something();
   s.field = prm;
   s = Something.staticField
```

Recalling the LS project…

Example:
- A single **MyServlet** instance, used to handle multiple simultaneous HTTP requests



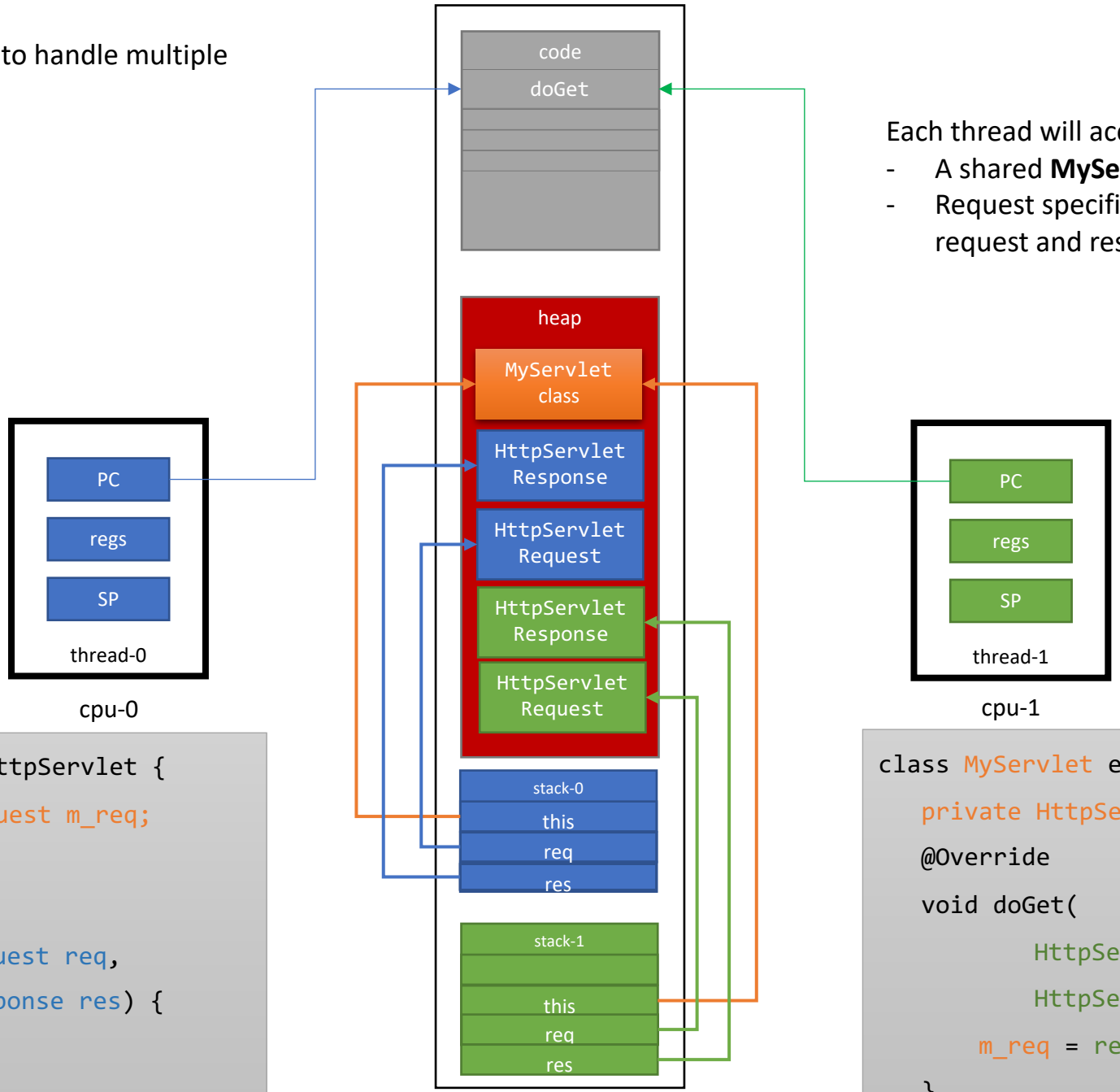
Each thread will access
- A shared **MyServlet** instance fields
- Request specific (and therefore thread specific) request and response objects

```
class MyServlet extends HttpServlet {
   private Something aField;
   @Override
   void doGet(
          HttpServletRequest req,
          HttpServletResponse res) {
   }
}
```

```
class MyServlet extends HttpServlet {
   private Something aField;
   @Override
   void doGet(
          HttpServletRequest req,
          HttpServletResponse res) {
   }
}
```

Example:
- A single **MyServlet** instance, used to handle multiple simultaneous HTTP requests

The type system
or the VM
will **not** warn us of this




```
class MyServlet extends HttpServlet {
   private HttpServletRequest m_req;
   @Override
   void doGet(
           HttpServletRequest req,
           HttpServletResponse res) {
      m_req = req
   }
```

Each thread will access
- A shared **MyServlet** instance fields
- Request specific (and therefore thread specific) request and response objects

```
class MyServlet extends HttpServlet {
   private HttpServletRequest m_req;
   @Override
   void doGet(
           HttpServletRequest req,
           HttpServletResponse res) {
      m_req = req
   }
```

Three types of data

- Mutable thread-bound data
  - E.g. the servlet request and response objects
  - No thread sharing issues (because no sharing)
- Immutable shared data
  - E.g. the servlet, the router, the handlers
  - No thread sharing issues (if *some* requirements are fulfilled)
- Mutable shared data
  - E.g. the data source
  - Prone to concurrency hazards
  - Proper synchronization is required