Adding Persistence to Java

Mario Wolczko
Architect
Oracle Labs

PIRL
July 22, 2019

Overview

- One possible approach to adding persistence to the Java platform:
  - Rationale
  - Model
  - Implementation
  - Open questions

- Widely used

Goal: minimize disruption of current users' knowledge and practices

- Lots of existing code

Goal: maximize ability to reuse existing code

How is persistent state expressed and accessed?

How is persistent state expressed and accessed?

Could use:

- Special constructor for persistent objects (`pnew`)
- Inherit from a special `PersistentObject` class
- Add `@Persistent` annotation on a class

How is persistent state expressed and accessed?

Could use:

- Special constructor for persistent objects (`pnew`)
- Inherit from a special `PersistentObject` class
- Add `@Persistent` annotation on a class

All rejected as being *non-orthogonal* — prevents the use of a volatile class/object for persistence and *vice versa.*

What is the model for consistency and recovery?

What is the model for consistency and recovery?

Could add transactions at the object level;
nested transactions with compensation;
transaction-based concurrency control

What is the model for consistency and recovery?

Could add transactions at the object level;
nested transactions with compensation;
transaction-based concurrency control

Rejected as too invasive and disruptive to existing code and practices

Persistence by reachability

- Designate some variables as persistence *roots*
- All objects reachable from a persistent root are persistent
- Lots of research on this from ~1980—2000, including an implementation for Java at Sun Labs (known initially as OPJ, later as PJama).

However…

Hard state vs soft state

Hard state is information that cannot be lost

Soft state is information that can be reconstructed from existing hard state

From *Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service,* Saito, Bershad, and Levy, 1999

What is this man about to say?

https://www.youtube.com/watch?v=nn2FB1P_Mn8

What is this man about to say?

"Have you tried turning it off and on again?"

https://www.youtube.com/watch?v=nn2FB1P_Mn8

Hard & soft state in NVRAM

Soft state:

- *Could* discard at uncontrolled restarts
- Or, discard *only* when corruption is suspected

Hard state:

- **Cannot** discard. *Must* provide backup, recovery, etc., perhaps replication (for availability)

Our model

- Partitioned heap (following NVM Direct, see Bill Bridge's presentation tomorrow)
    - Allows enforced separation of state
- Partition checkpoints

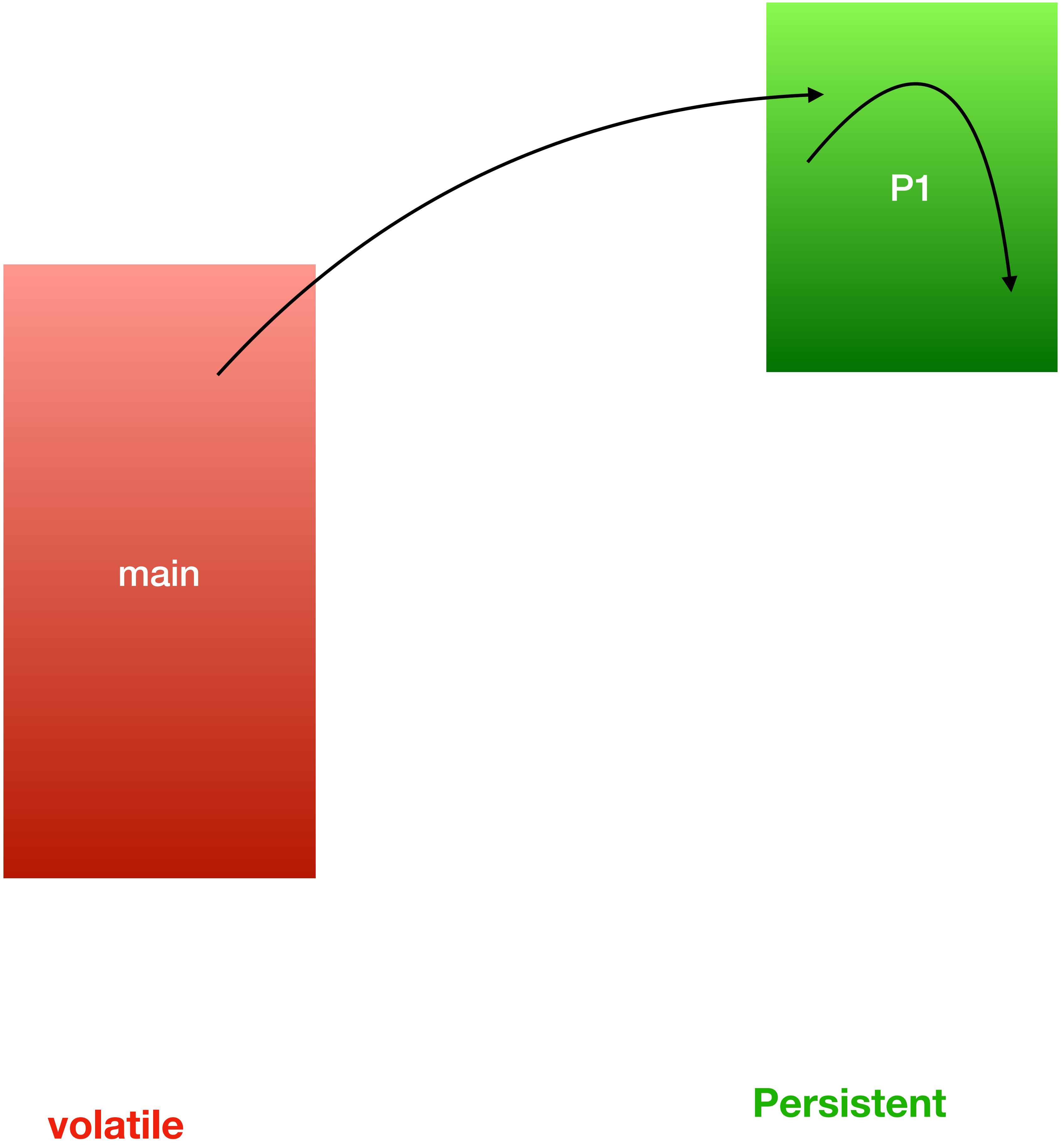
Partitioned heaps

- Partition the heap into volatile and persistent *regions*, visible to the programmer.
- Persistent regions are mapped from DAX files.
- Each object is allocated in a specific region.
- Disallow cross-region references, except from volatile regions.
- When `main` begins, there is a single volatile region.

main

volatile

Persistent

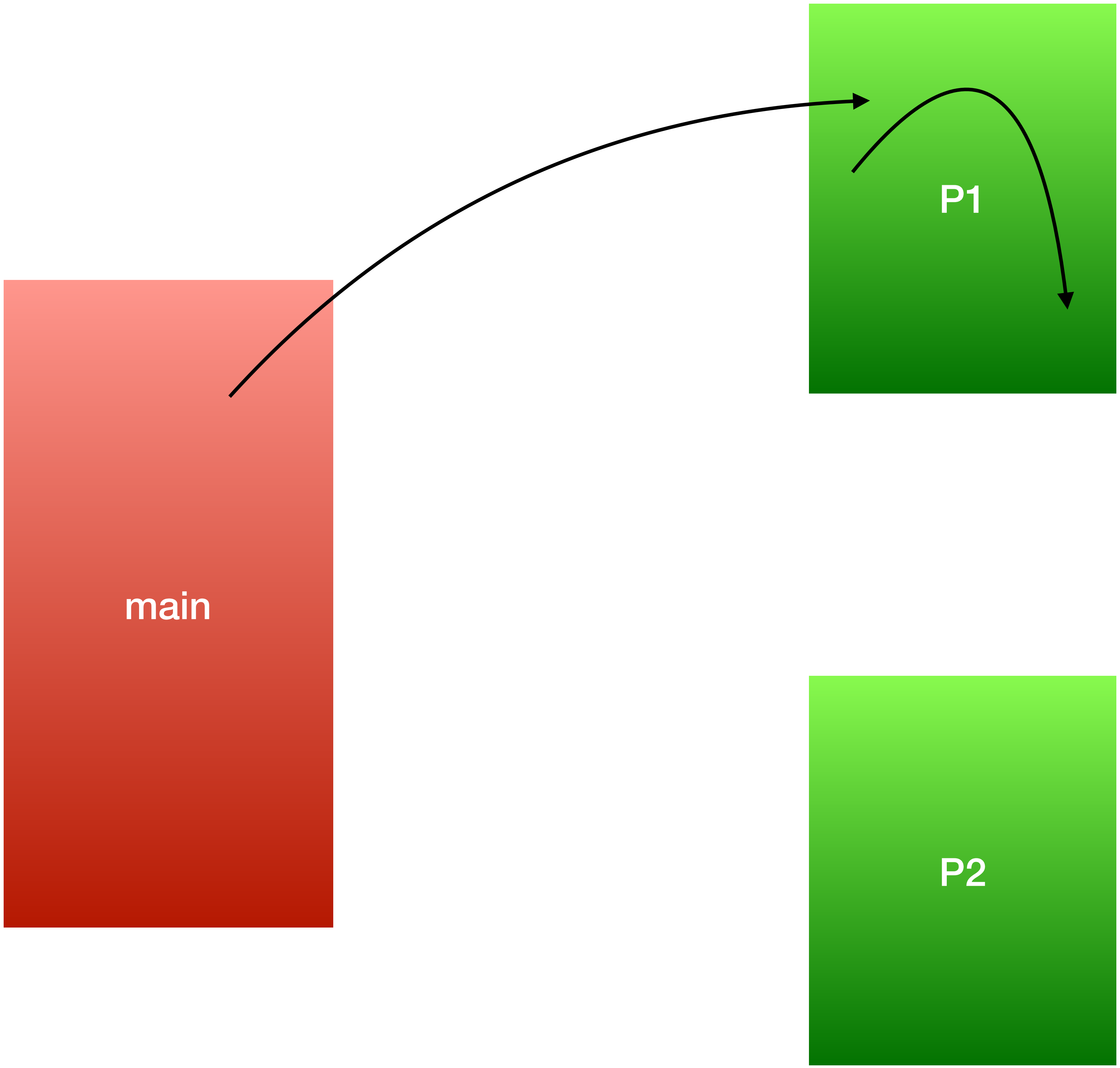


main

P1

volatile

Persistent



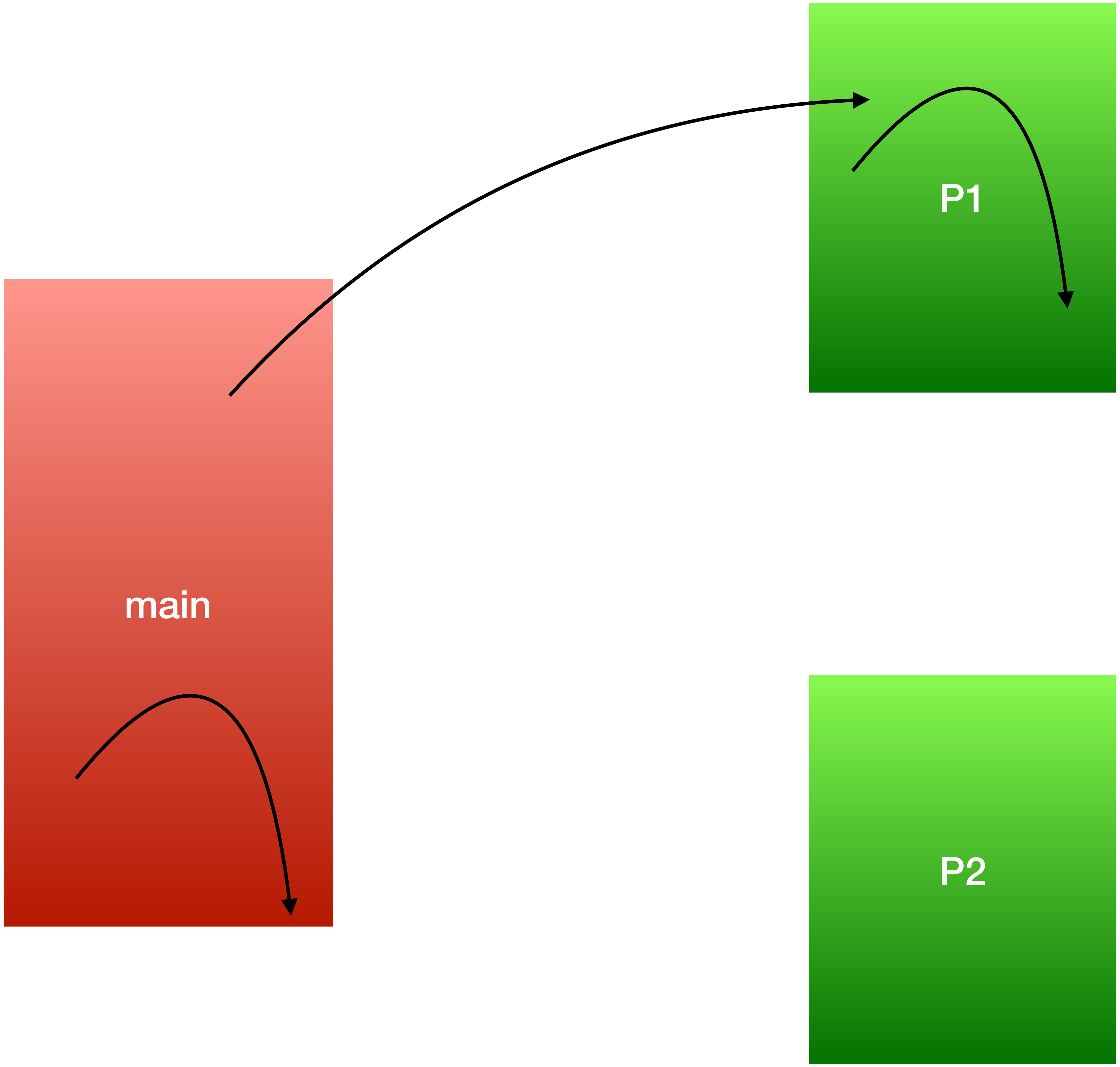
main

P1

P2

volatile

Persistent



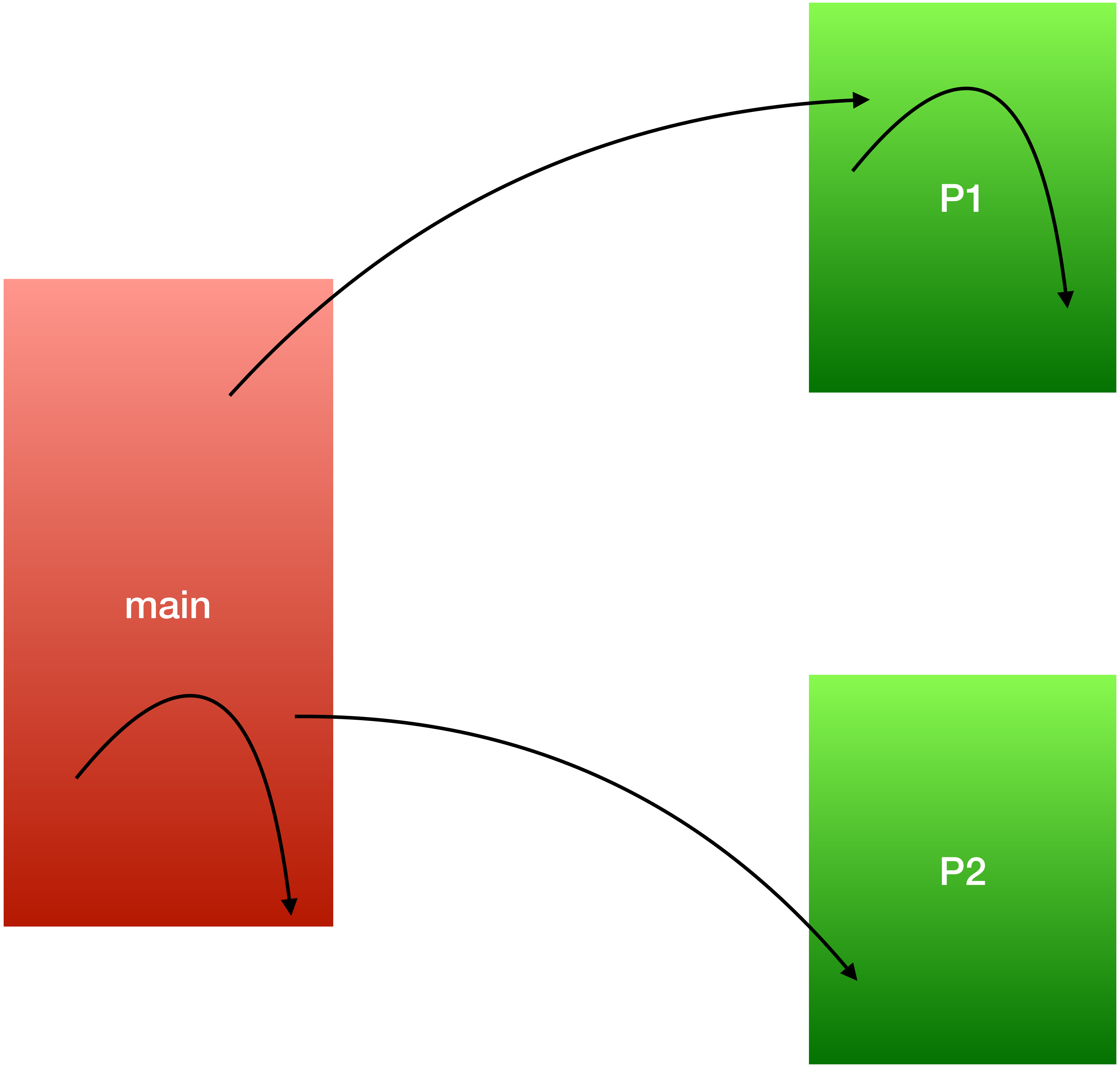
main

P1

P2

volatile

Persistent



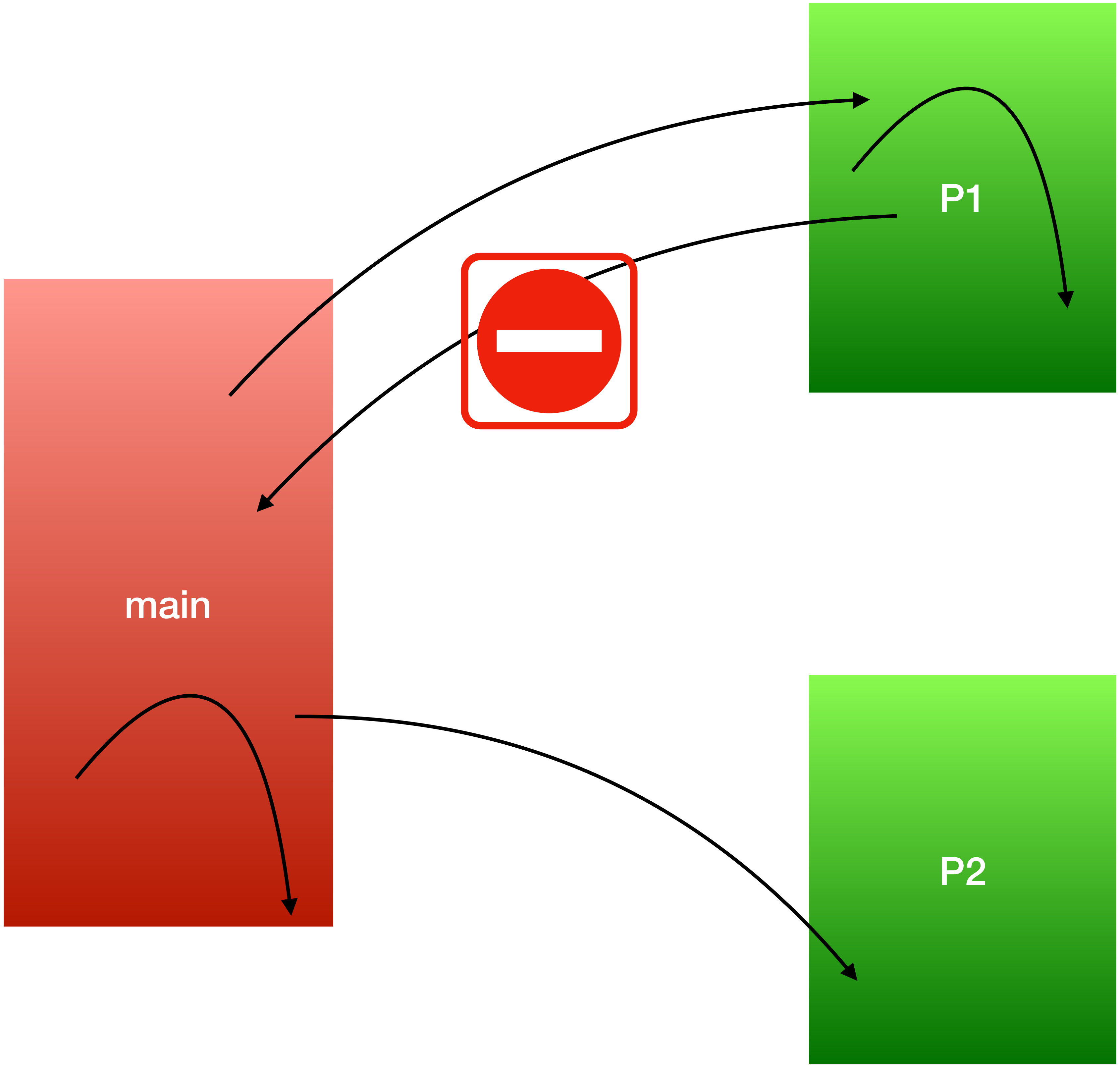
P1

main

P2

volatile

Persistent



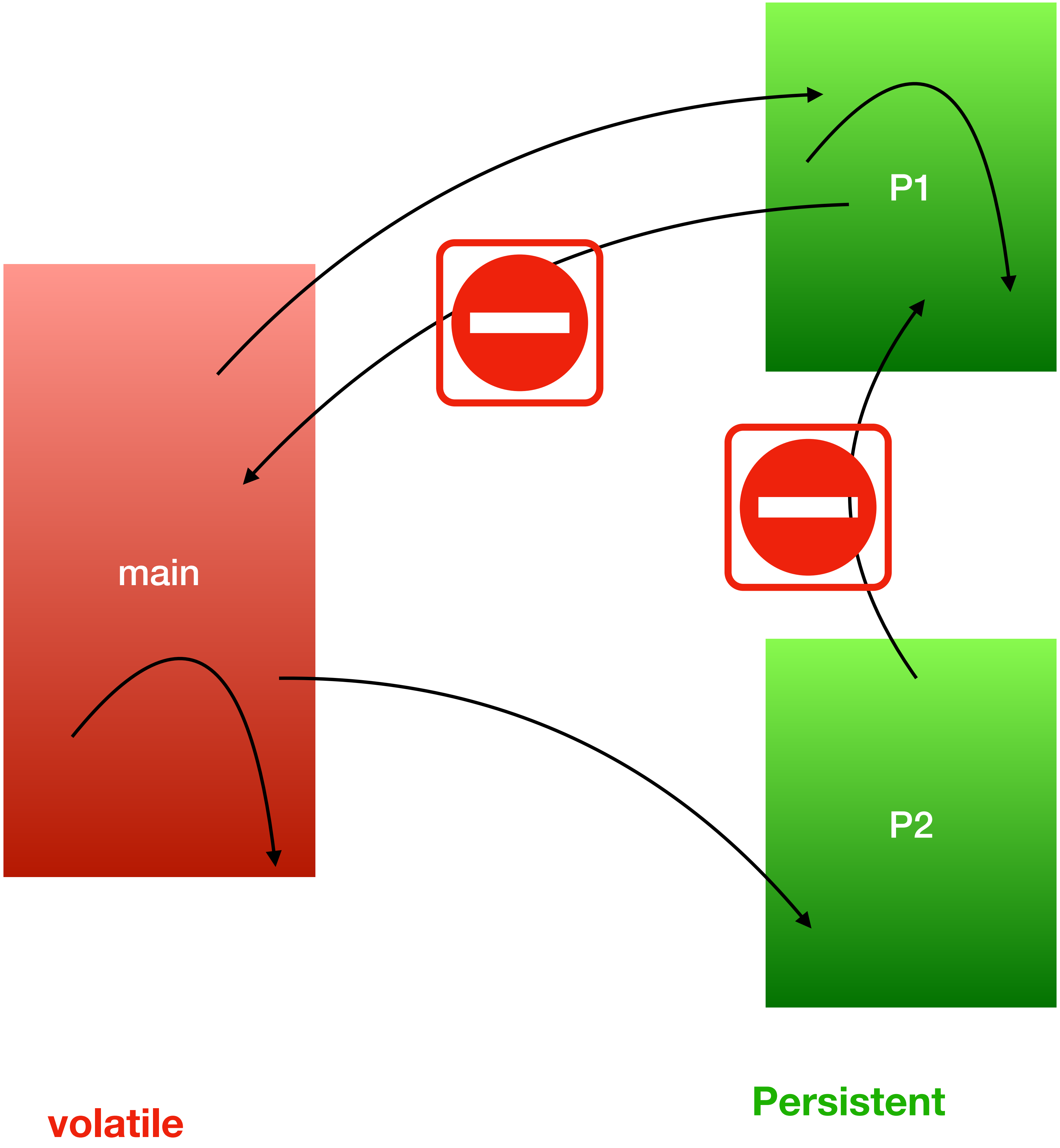
P1

main

P2

volatile

Persistent


P1
main
P2
volatile
Persistent

The current region

- Each thread has a notion of *the current region* — all allocations by the thread are in the current region.
- The current region can be changed. Code can be wrapped to use a specific region, e.g.:

```
myRegion.run(() -> {h = new HashMap();});
```

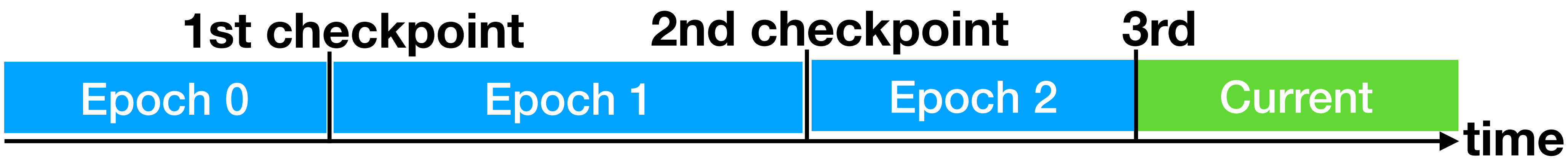
- The persistence of a data structure can be specified by context—enables reuse.

Checkpoints

- The program can invoke a checkpoint primitive which snapshots the current state of one or more regions.
- At re-attach, a region is recovered to the last checkpoint.
- Regions can be checkpointed independently or together. Example: checkpoint a DB infrequently, but the log after every addition.

Epochs

- Each region goes through a series of epochs. An epoch is the period between two checkpoints (or before the first).



1st checkpoint

2nd checkpoint

3rd

Epoch 0

Epoch 1

Epoch 2

Current

time

- A checkpoint freezes the state of the region within that epoch. The data within an old epoch are immutable.

- When an object is modified for the first time within the current epoch, a copy is made. All subsequent modifications within the current epoch are to this copy.

- The running program can only observe the most recent version of an object.

Recovery

- When a restart occurs, and the latest epoch is uncommitted, recovery takes place.
- Recovery discards the current, uncommitted epoch and reverts to the state at the end of the previous epoch.
- Locks held on objects in the region are re-initialized.

Managing checkpoints

- Each checkpoint persists until explicitly discarded.
- If the oldest checkpoint is deleted, objects in that checkpoint without modified copies in the next epoch are absorbed into the next epoch. If an object is superseded by a copy in the next epoch, it can be discarded.
- An intermediate-age checkpoint can also be deleted, merging the two epochs around it.
- We can also recover to any checkpoint, discarding all the later epochs.

The programmer’s burden

To convert an existing application to use NVM, the programmer:

- Partitions the heap into volatile and non-volatile regions
- Adds the region creation and loading logic,
- Wraps non-volatile object creations in `Region.run()`.
- Adds calls to `checkpoint()` at places where the region is consistent.

Example: simple phone directory

(following an example by Eliot Moss)

```
public class PhoneDirectory {

  HashMap<String, String> dir;

  public static void main(String[] args) {
    …
  }
  …
}
```

Example: simple phone directory

(following an example by Eliot Moss)

```
public class PhoneDirectory extends Region {

  HashMap<String, String> dir;

  public static void main(String[] args) {
    …
  }
  …
}
```

```
public class PhoneDirectory extends Region {

  HashMap<String, String> dir;

  public static void main(String[] args) {
    PhoneDirectory pd = null;
    try {
      pd = new PhoneDirectory("phonedir.region");
    } catch (InvalidRegionFileException | IOException ex) {
      System.err.println("Cannot connect/create region");
      System.exit(1);
    }
    …
  }

  …

}
```

```
public class PhoneDirectory extends Region {

  HashMap<String, String> dir;

  public static void main(String[] args) {
    PhoneDirectory pd = null;
    try {
      pd = new PhoneDirectory("phonedir.region");
    } catch (InvalidRegionFileException | IOException ex) {
      System.err.println("Cannot connect/create region");
      System.exit(1);
    }
    …
  }

  @Override
  public void initialize() { this.dir = new HashMap<>(); }

}
```

Further along in main():

```
// command received to add an entry for name and number
…
addEntry(name, number);
```

Without persistence

Further along in main():

```
// command received to add an entry for name and number
…
run(() -> addEntry(name, number));
```

Region management added

```
private void addEntry(String name, String number)
  throws PDException
{
  if (dir.containsKey(name)) {
    error(“name already in directory");
  }
  dir.put(name, number);

}
```

Without persistence

```
private void addEntry(String name, String number)
  throws PDException
{
  if (dir.containsKey(name)) {
    error(“name already in directory");
  }
  dir.put(name, number);
  checkpoint();
}
```

Region management added - but there’s a bug

```
private void addEntry(String name, String number)
  throws PDException
{
  String name2 = copyString(name);
  if (dir.containsKey(name2)) {
    error("name already in directory");
  }
  dir.put(name2, copyString(number));
  checkpoint();
}
```

```
private void addEntry(String name, String number)
  throws PDException
{
  String name2 = copyString(name);
  if (dir.containsKey(name2)) {
    error("name already in directory");
  }
  dir.put(name2, copyString(number));
  checkpoint();
}
```

- The String parameters must be copied to the region.

```
private void addEntry(String name, String number)
  throws PDException
{
  String name2 = copyString(name);
  if (dir.containsKey(name2)) {
    error("name already in directory");
  }
  dir.put(name2, copyString(number));
  checkpoint();
}
```

- The String parameters must be copied to the region.
- For immutable data, the JVM could do this automatically.

```
@CopyArgs
private void addEntry(String name, String number)
  throws PDException
{
  if (dir.containsKey(name)) {
    error("name already in directory");
  }
  dir.put(name, number);
  checkpoint();
}
```

- Using annotations to save boilerplate

Implementation

- A region can be mapped anywhere (although it may be possible to have it commonly mapped at the same address). Hence it must be position-independent, or cheaply (and preferably incrementally) relocatable.
- We'd like checkpoints to be fast, so they can be used relatively often
- We'd like checkpoints of small regions to be even faster
- Steady-state performance of long-running programs is paramount
- Crashes and recovery are infrequent
- NB: NVM is much cheaper (per byte) than DRAM, so we can trade space for time—but still be mindful of cache behaviors

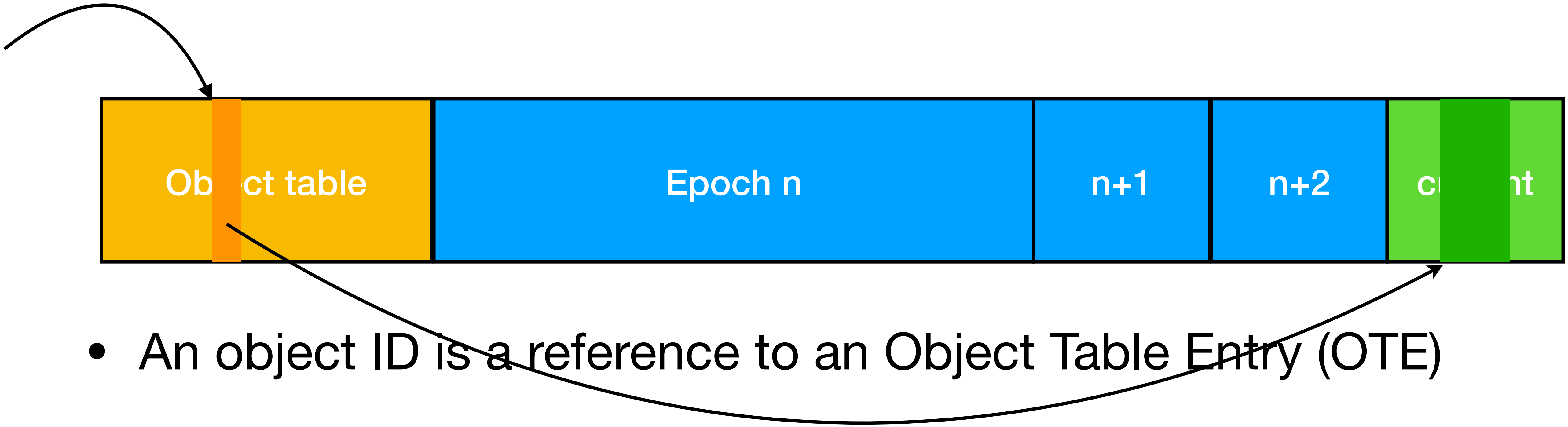
The heap structure of a region

- An object ID is a reference to an Object Table Entry (OTE)
  - Self-relative when in the heap
- The OT references the most recent version of an object.
- Each version references the next oldest from its header (reserved header space).

The heap structure of a region

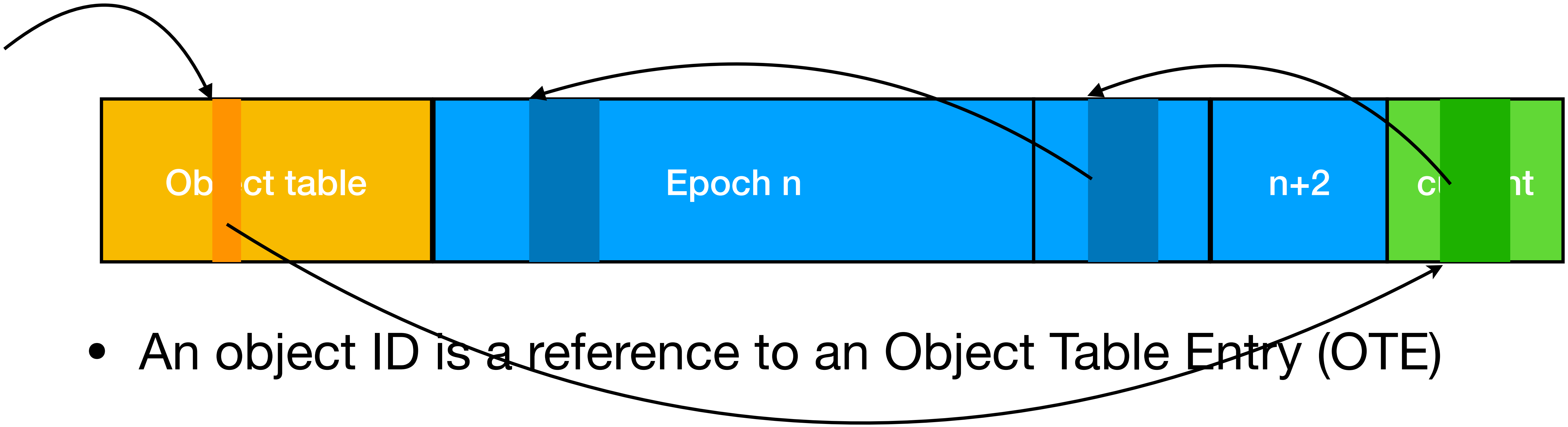
- An object ID is a reference to an Object Table Entry (OTE)
  - Self-relative when in the heap
- The OT references the most recent version of an object.
- Each version references the next oldest from its header (reserved header space).

The heap structure of a region



- An object ID is a reference to an Object Table Entry (OTE)
  - Self-relative when in the heap
- The OT references the most recent version of an object.
- Each version references the next oldest from its header (reserved header space).

The heap structure of a region



- An object ID is a reference to an Object Table Entry (OTE)
    - Self-relative when in the heap
- The OT references the most recent version of an object.
- Each version references the next oldest from its header (reserved header space).

Open questions

1. Is this a useful model? When is it not usable? What kinds of mistakes are made, and what is the mitigation?

2. Which libraries and apps need to be changed? How do we find them?

3. How to handle threads? Objects with external soft state?

4. Make it work; make it fast. How fast?

   - Peak performance, checkpoint latency

5. How to enable software evolution?

Questions?
Comments?

Integrated Cloud

Applications & Platform Services