Generational redux

Schedule

	M	W
Sept 14	Intro/Background	Basics/ideas
Sept 21	Allocation/layout	GGGGC
Sept 28	Mark/Sweep	Copying GC
Octo 5	Details	Ref C
Octo 12	Thanksgiving	Mark/Compact
Octo 19	Partitioning/Gen	Generational
Octo 26	Other part	Runtime
Nove 2	Final/weak	Conservative
Nove 9	Ownership	Regions etc
Nove 16	Adv topics	Adv topics
Nove 23	Presentations	Presentations
Nove 30	Presentations	Presentations

The big problem

- Young collection requires old collection, old collection leaves young generation in inconsistent state
- How to collect with young in inconsistent state?

Two solutions

- Option one: Collect during old, but don’t promote, no inconsistency (depends on young strategy)
- Option two: Don’t move, retry young collection after old
- “Retry” is ill-defined

The big problem

- Young collection: Treat old as root, promote until old is full
- Old collection: Cannot promote young, so young don’t move
- Second young collection: Some things already moved, some not

The solution

- Standard young collection is copying-only
- Secondary young collection uses marks
- Need two sweeps to clear out marks
  - Unless you switch meaning of mark bit every collection

```
traceSecond(loc):
  obj := *loc
  if obj->header.forward:
    obj := obj->header.forward
    *loc := obj
  if !obj->header.mark:
    if obj in fromspace:
      (copy obj to newObj in tospace)
      obj->header.forward := newObj
      obj := newObj
      *loc := newObj
    (scan obj)
    obj->header.mark := 1
```

Project 3

- Because I didn't explain this detail, project 3 deadline extended to Wednesday the 11th
- Please also talk to me about final projects and presentations!

Finalization

Finalization

- When an object dies, reclaim resources other than memory
- Ideally: Get all resources into memory, problem solved
- Reality: e.g. open files, sockets, OS handles need special reclamation

Finalization

- Usual design: User-defined finalizer per object or type

```
class File {
  constructor() { this.fd = open(…); }
  finalizer() { close(this.fd); }
}
```

When to finalize

- Problems:
  - User-defined finalizers may allocate objects: Can’t do it during GC
  - User-defined finalizer has access to finalized object: Can’t actually free the object

When to finalize

```
class File {
  // pointless global variable
  static File lastFreedFile;
  constructor() { this.fd = open(…); }
  finalizer() {
    close(this.fd);
    File.lastFreedFile = this;
  }
}
```

When to finalize

```
class File {
  // pointless global variable
  static File lastFreedFile;
  constructor() { this.fd = open(…); }
  finalizer() {
    close(this.fd);
    File.lastFreedFile = this;
  }
}
```

Object resurrection!

Real finalization

- In e.g. Java, finalized objects are kept alive by GC, and may survive finalization
- After finalization, next GC cycle can free object
- Essentially: The pending finalizer acts like a reference

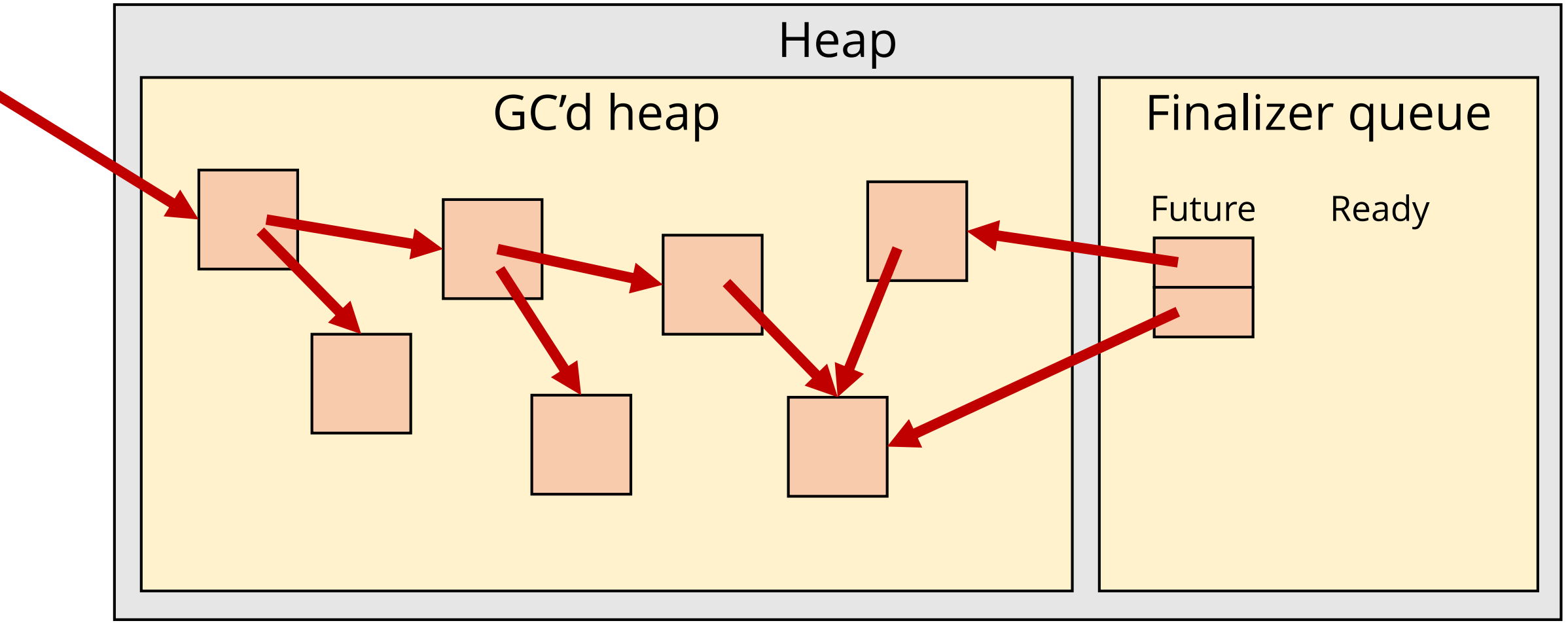
Handling finalization

- If an object is unreachable, but has a finalizer
- Mark it as reachable, enqueue finalizer
- Run finalizers at end of collector

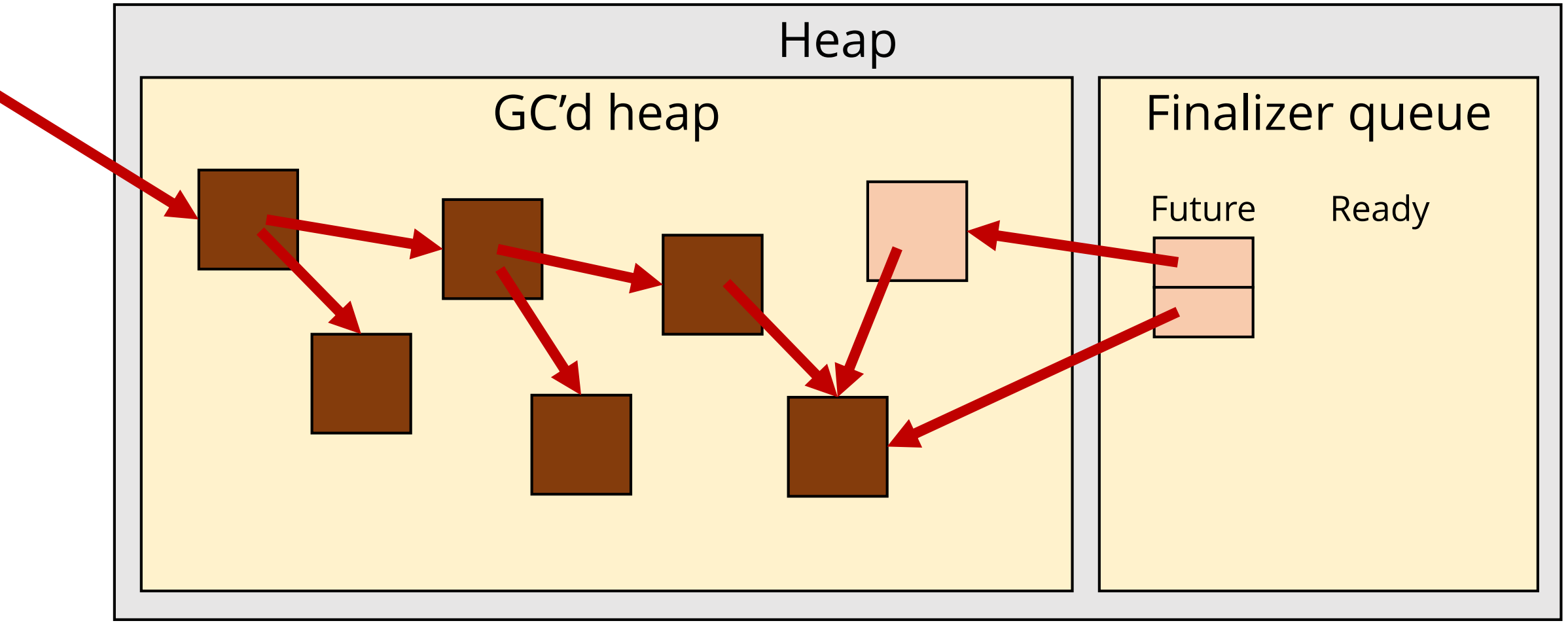
- Runtime interface: Tell GC “this object has this function as its finalizer”

```
collect():
  (initialize worklist)
  (handle worklist items)
  foreach qi in finalizerQueue:
    if !qi.obj->header.mark:
      (move qi to ready queue)
      (add qi.obj to worklist)
  (handle worklist items)
  (sweep)
  foreach qi in finalizerReadyQueue:
    qi.function(obj)
```

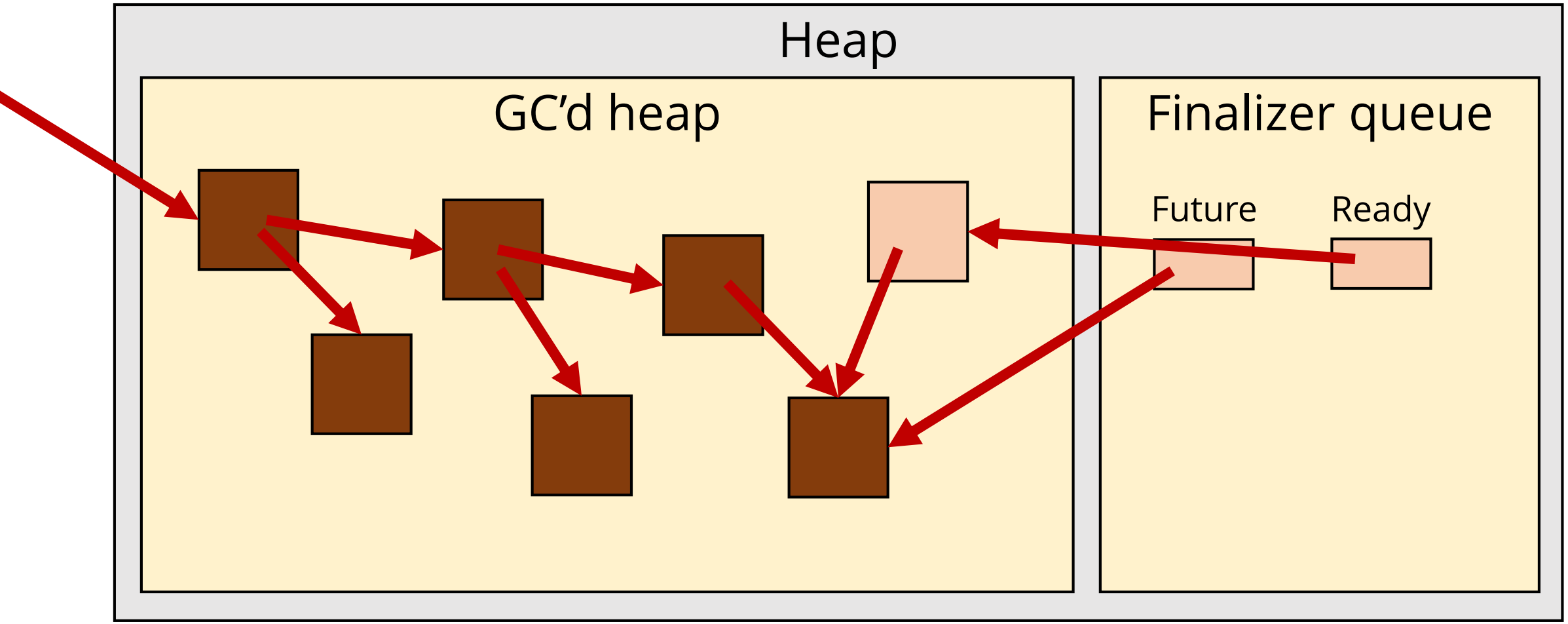
Finalization queue



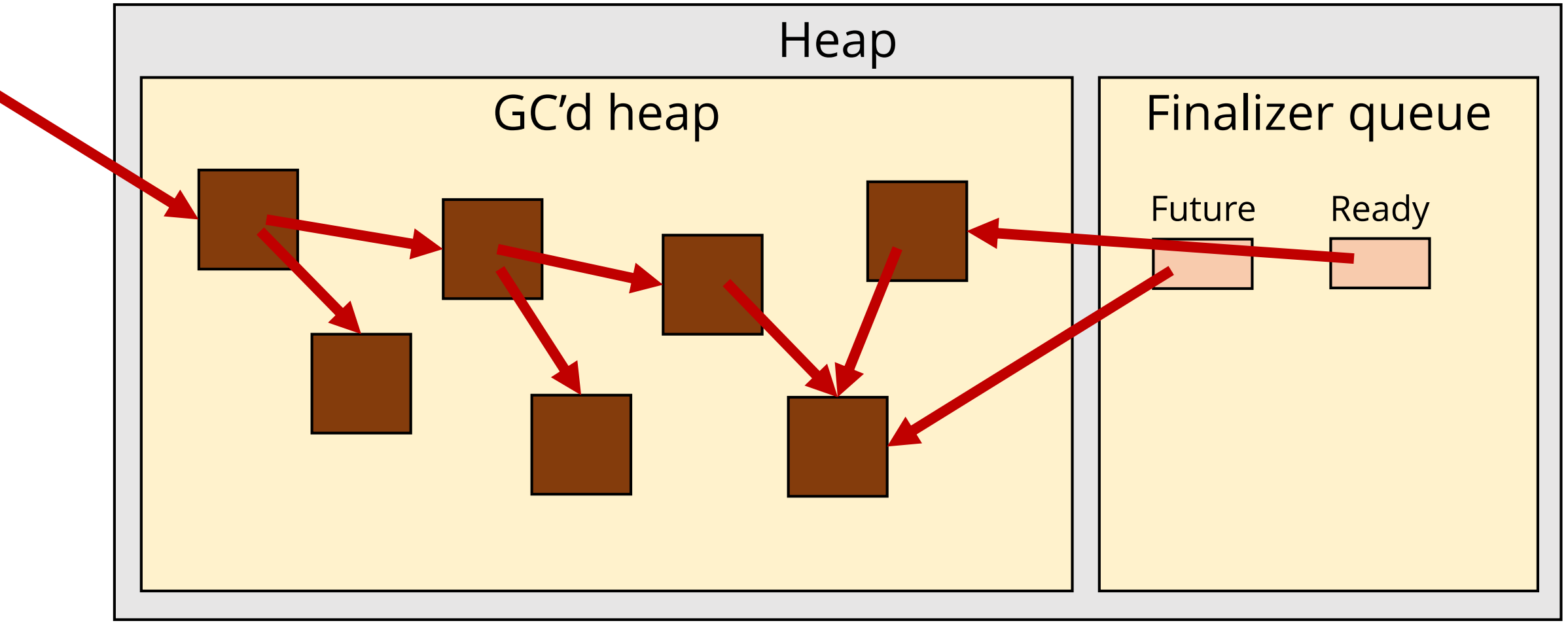
Finalization queue



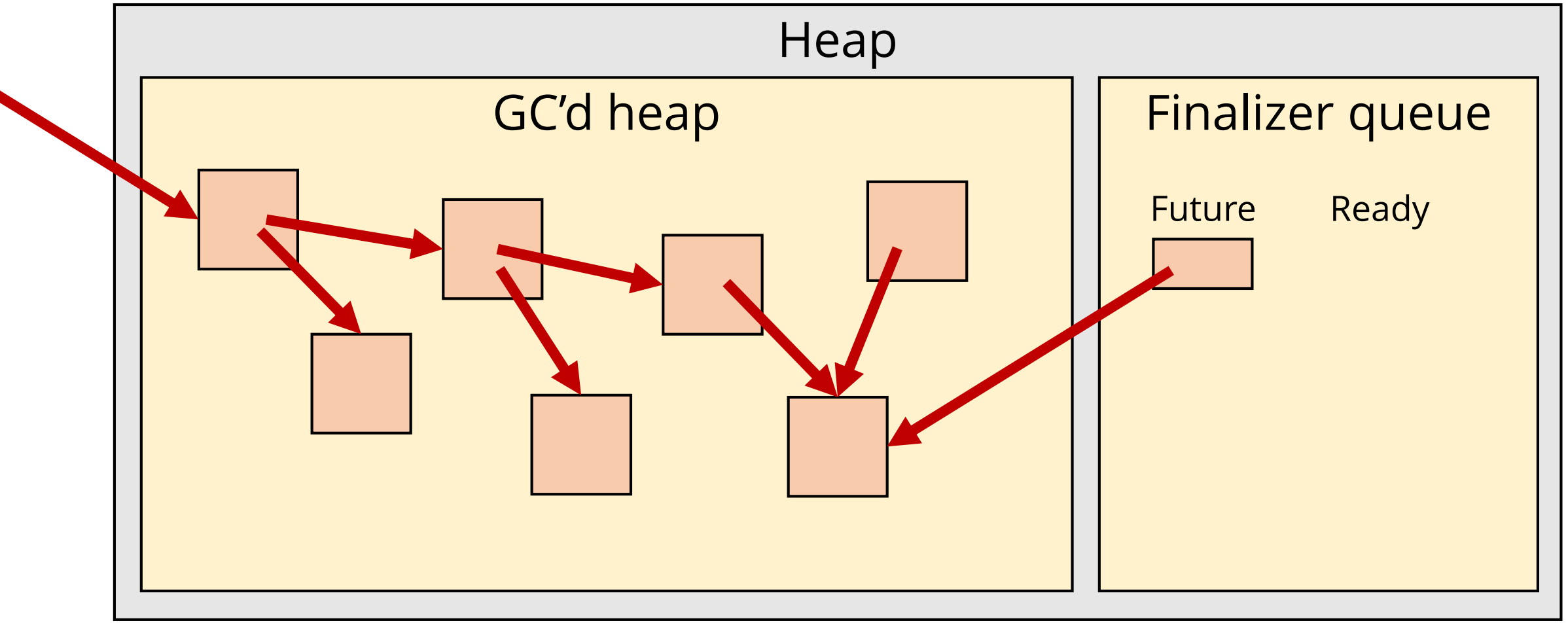
Finalization queue



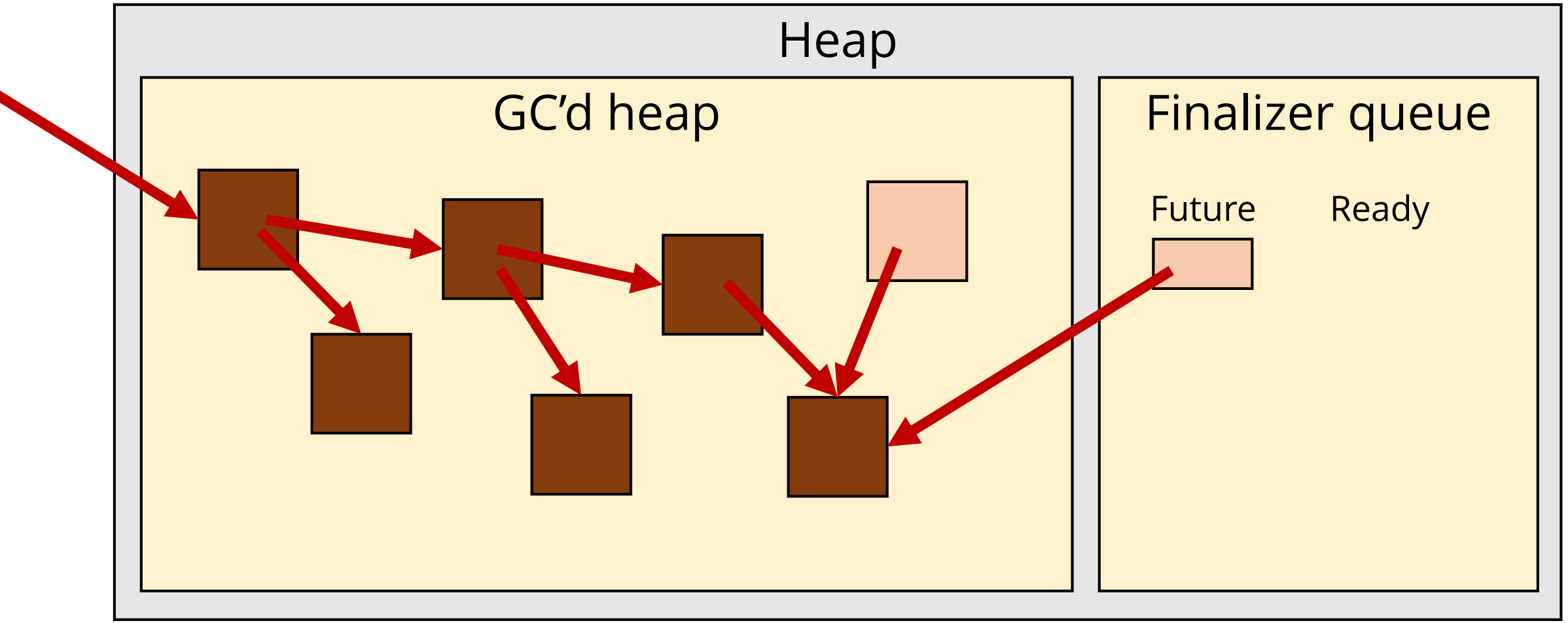
Finalization queue



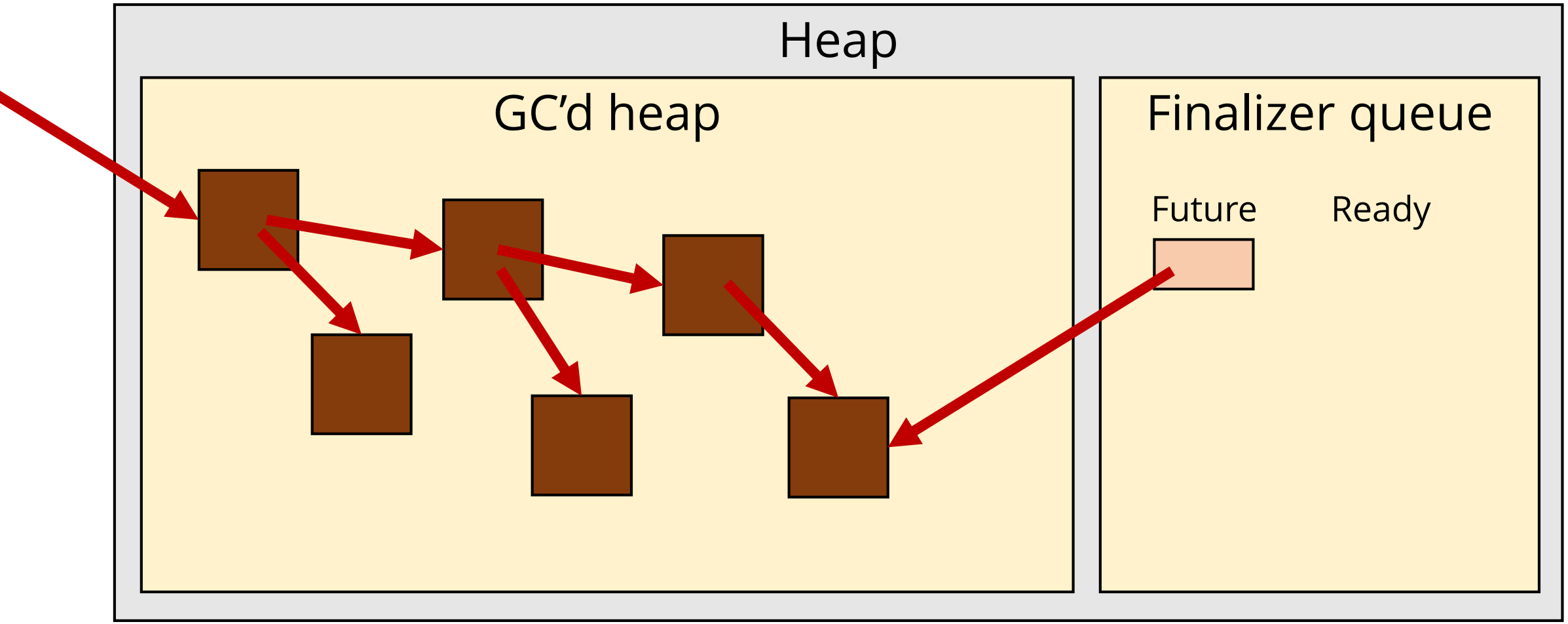
Finalization queue



Finalization queue



Finalization queue



Other finalizer thoughts

- Order of finalization might matter
  - e.g. FileBuffer relies on File, FileBuffer finalizes by writing last data, File finalizes by closing file
- Finalization may have thread issues and surprising races

Finalizers and meta-GC

- Finalizers are to collect non-memory resources
- GC triggered only because of memory
- Could e.g. fail to open files because dead objects keeping remainder alive
- Reason to allow manual request for GC

Weak references

Weak references

```
Weak<Object> foo = new Weak<Object>(obj);

foo.get() == obj; // true

obj = null; // and lose all other references

foo.get(); // might be null or obj
// GC occurs
foo.get() == null; // true
```

Typical use

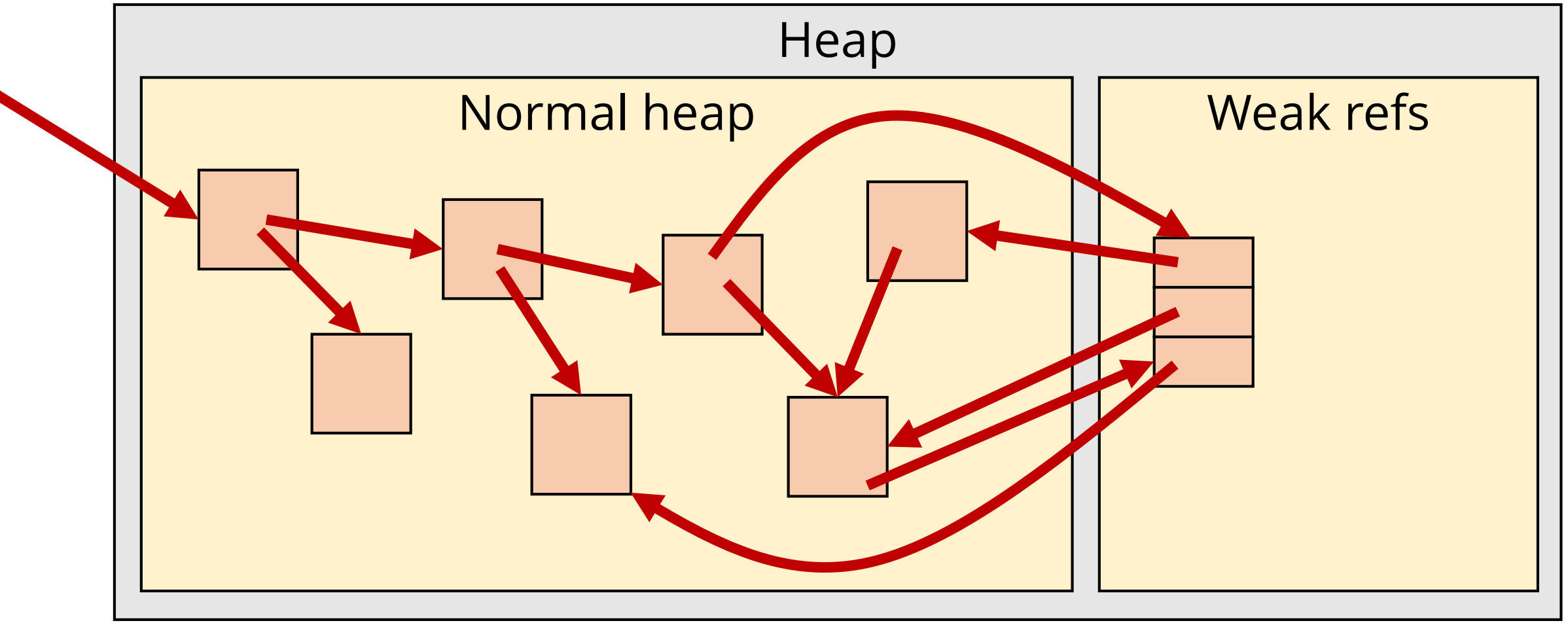
- Keep extra data for certain objects
- If objects die, no longer need extra data
- But, need reference to object to remember extra data!
- So, keep weak reference for extra data

Weak references

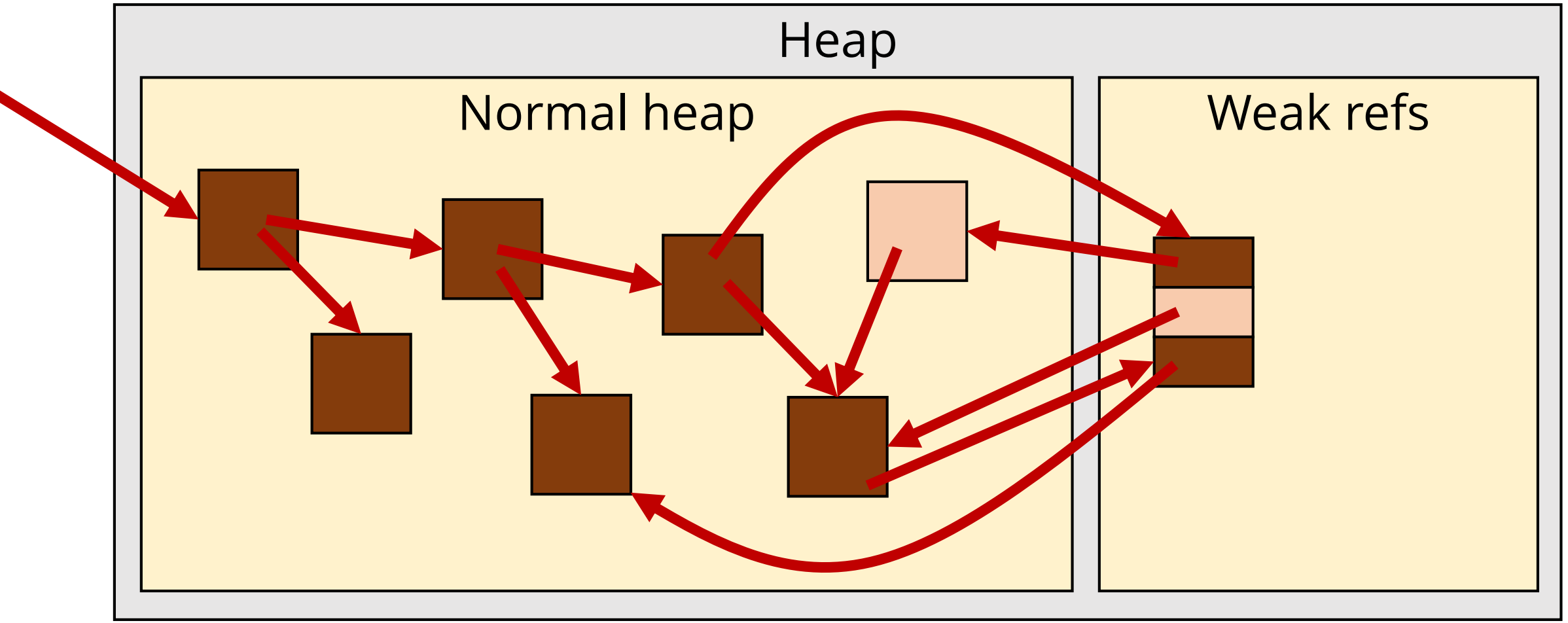
- Usually opaque: Special object with "`get`" function to return real reference
- Special partition for these objects
- Don't trace objects in weak partition
- All weak reference objects are same size: No fragmentation possible in weak partition

```
collect():
  (initialize worklist)
  (handle worklist items, don't scan weak ref objects)
  foreach wrObj in weak ref partition:
    if wrObj->header.mark:
      obj := wrObj->referant
      if !obj->header.mark:
        wrObj->referant := NULL
    else:
      free(wrObj)
  (sweep)
```

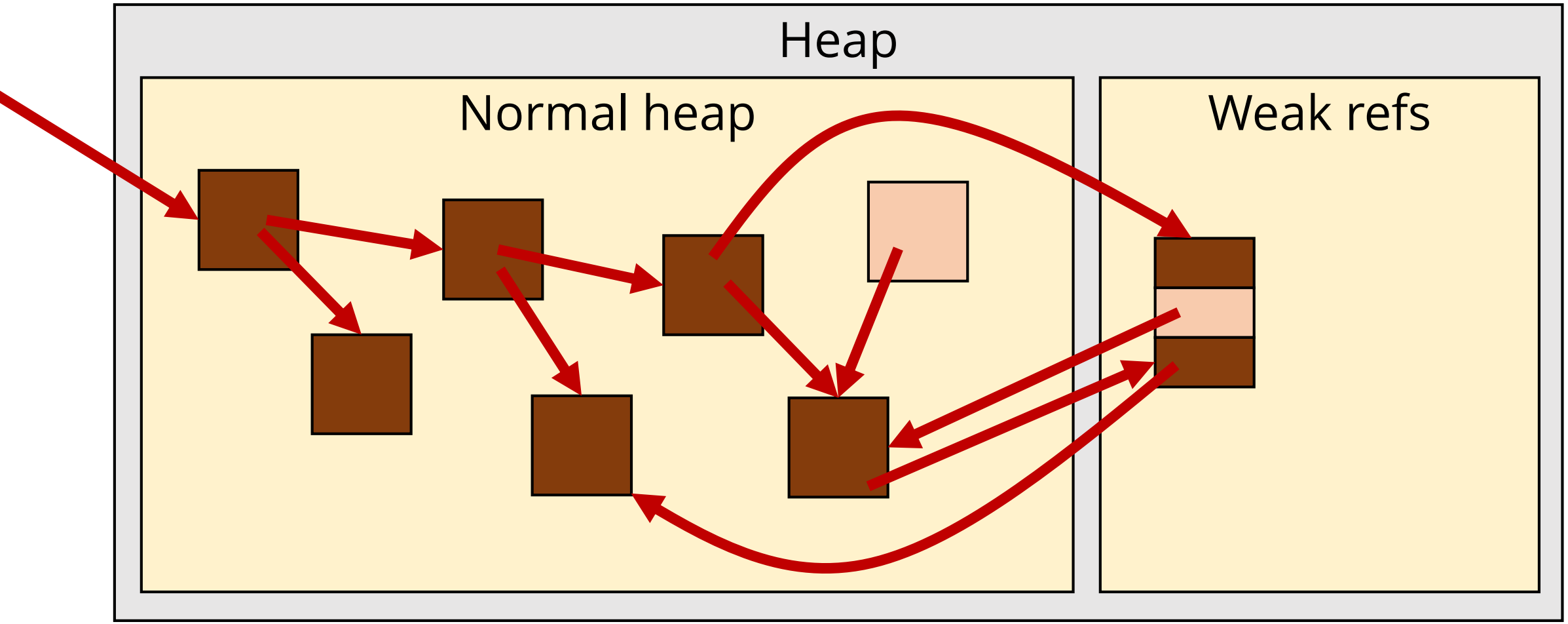
Weak ref partition



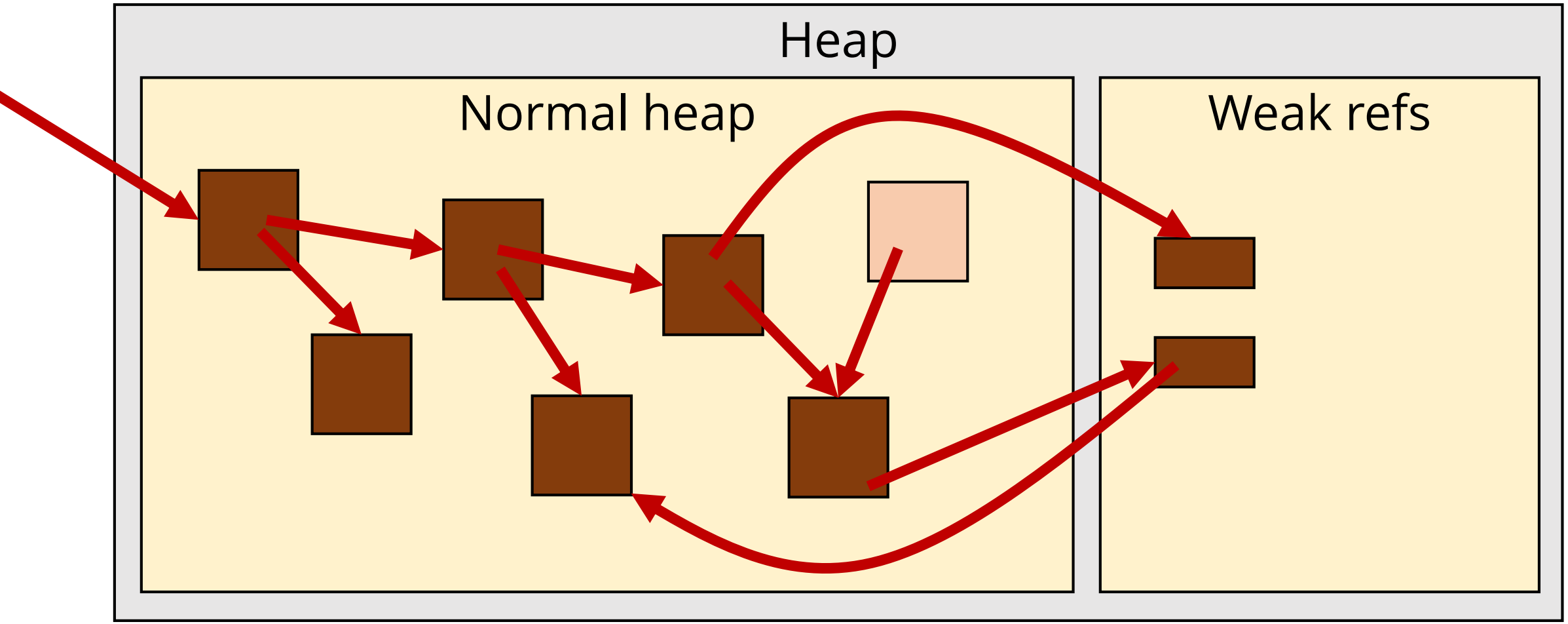
Weak ref partition



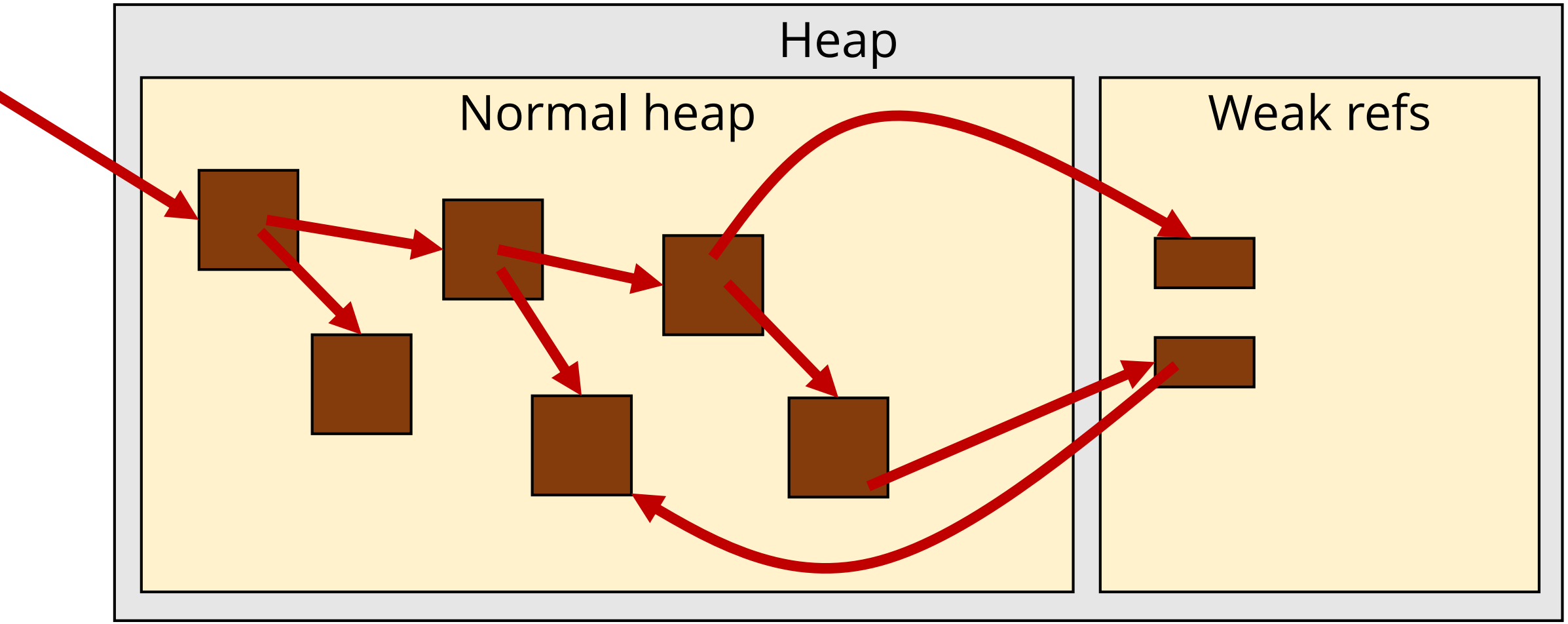
Weak ref partition



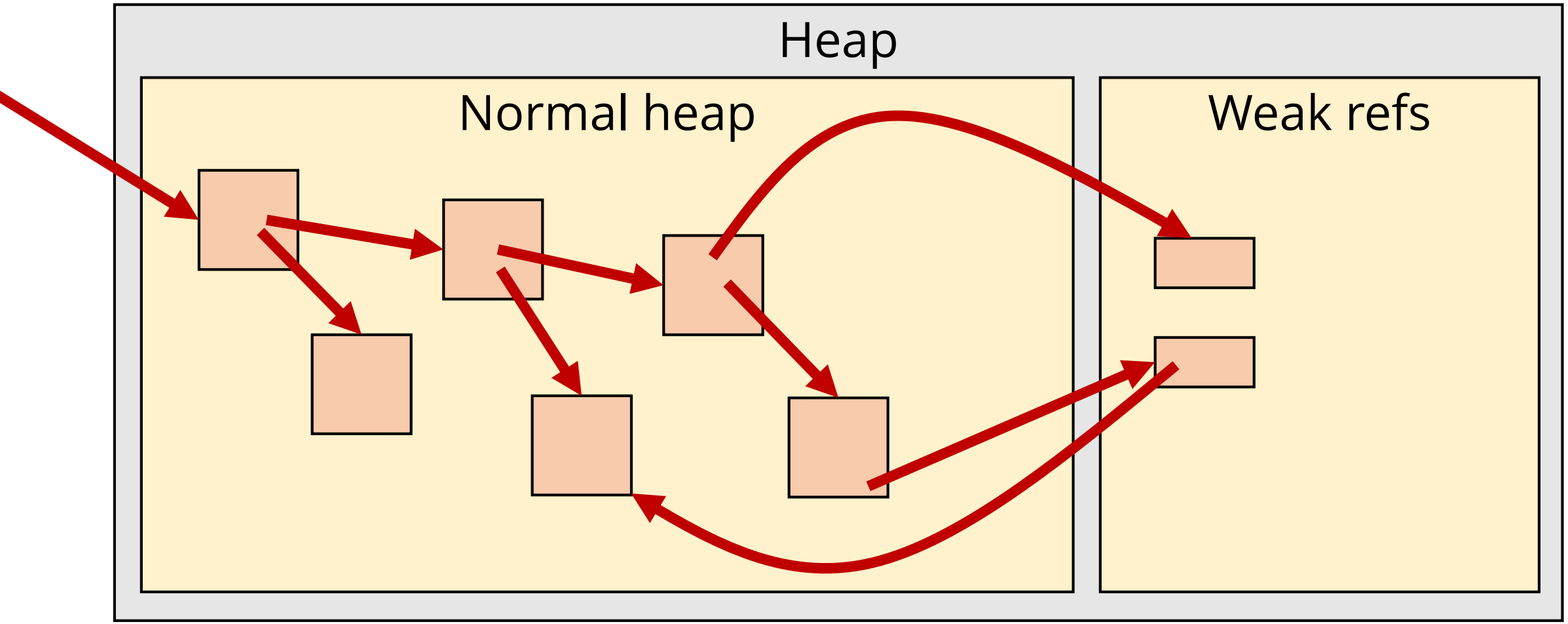
Weak ref partition



Weak ref partition



Weak ref partition



A complete garbage collector

Java partitioning scheme

- Young generation
  - Bucket 0: partitioned by thread
  - Bucket 1: semispace
- Old generation: mark-and-compact
- Weak ref partition: mark-and-sweep
- Immobile partition for JNI: mark-and-sweep
- Finalizer queue
- Executable partition: manual, freed by finalizers

```
collect():
  b1tospace, b1fromspace := b1fromspace, b1tospace
  (initialize worklist from roots)
  while loc := worklist.pop():
    obj := *loc
    if obj in young gen:
      if !obj->header.forward:
        if obj in bucket 0:
          (copy obj to newObj in b1tospace)
          obj->header.forward := newObj
        else if obj in b1fromspace:
          (try to promote obj to newObj)
          if newObj == NULL:
            (perform old collection)
            (retry young collection)
          obj->header.forward := newObj
        (scan newObj)
      *loc := obj->header.forward
    else if !obj->header.mark:
      if obj !in weak ref partition: (scan obj)
      obj->header.mark := 1
  (sweep finalizer partition, continue above loop)
  (sweep weak ref partition, nullify dead refs)
  (execute finalizers)
```