HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 10: Runtime Systems

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Announcements

- Project 2 due today
- Project 3 out
  - Due Tuesday Oct 9 (5 days)
- Project 4 out
  - Due Thursday Oct 25 (21 days)

Today

- Garbage collection
  - Key idea
  - Mark and sweep
  - Stop and copy
  - Generational collection
  - Reference counting
  - Incremental collection, concurrent collection
  - Boehm collector
- Work stealing
- Virtual machines

Runtime System

- Runtime system: all the stuff that the language implicitly assumes and that is not described in the program
  - Handling of POSIX signals
    - POSIX = Portable Operating System Interface
    - IEEE Computer Society standards for OS compatibility
  - Automated memory management (garbage collection)
  - Automated core management (work stealing)
  - Virtual machine execution (just-in-time compilation)
  - Class loading
  - …
- Also known as “language runtime” or just “runtime”

Automated Memory Management

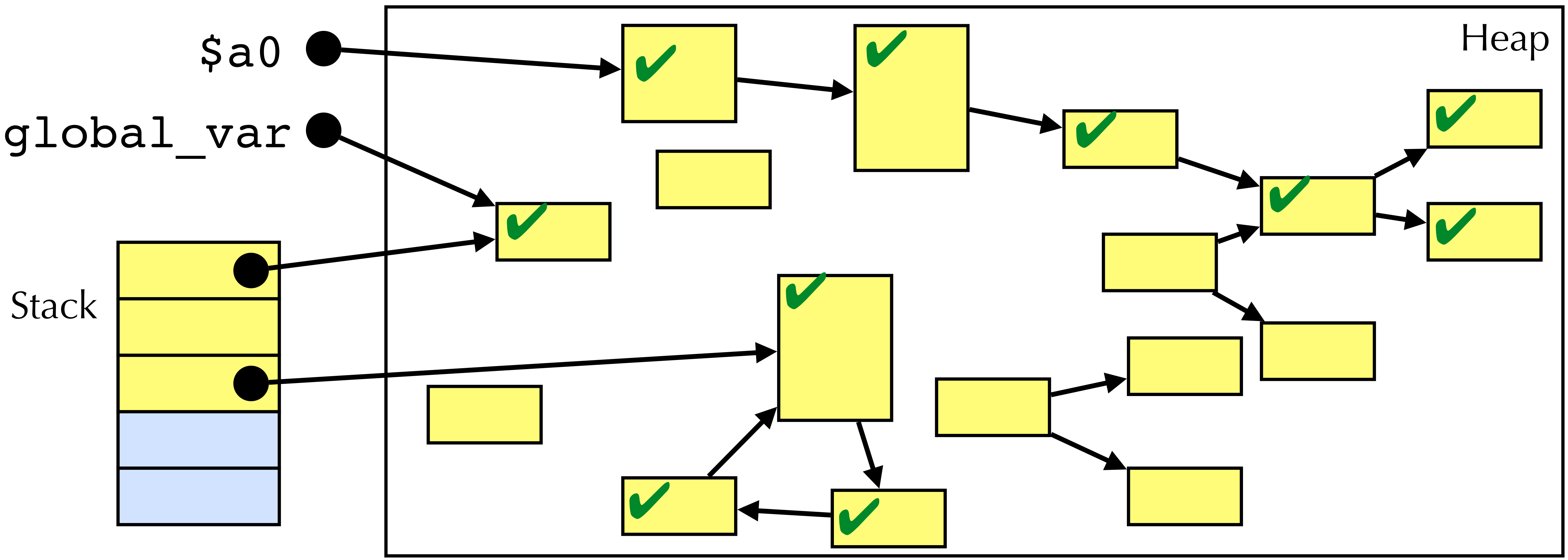
- Last lecture we talked about memory management
  - Manual memory management: programmers explicitly call `malloc()` and `free()`
  - Automatic memory management: runtime system looks after allocation and **garbage collection**
- Garbage collection: free memory that is no longer in use

Garbage Collection

- Runtime frees heap memory that is no longer in use
- How do we determine what is no longer in use?
- Ideally: any piece of memory that will not be used in the future of the computation
- In practice: any piece of memory that is not **reachable**
  - Reachable = can be accessed through some chain of pointers starting from program variables
  - This is a subset of the memory that will not be used in the future

Garbage Collection: Basic Idea

- Start from stack, registers, & globals (roots) and follow pointers to determine which objects in heap are reachable
- Reclaim any object that isn't reachable



- Problem: How do we know which values are pointers and which are non-pointers (e.g., ints)?

Identifying pointers

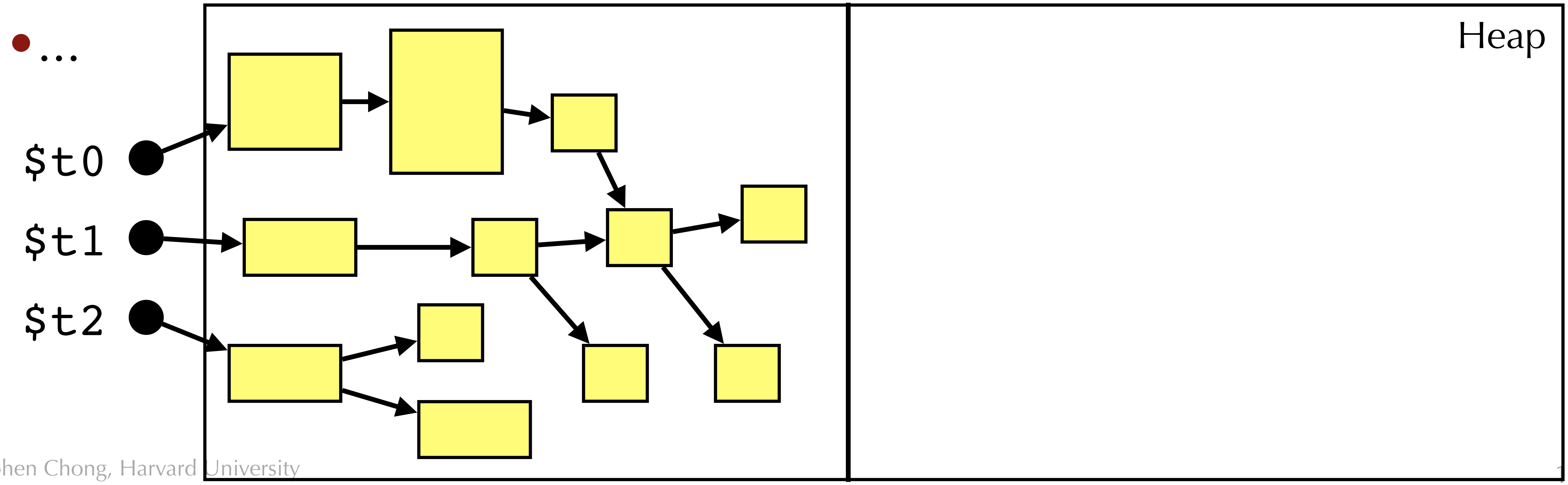
- OCaml uses the low bit: 1 it's a scalar, 0 it's a pointer
  - Why the low bit? Why not the high bit?
- In Java, we put tag bits in the meta-data
- In C (e.g., Boehm collector), typically use heuristics
  - If value doesn’t point into an allocated object, it’s not a pointer

Mark and Sweep Collector

- Reserve a mark-bit for each object.
- Mark phase
  - Starting from roots, mark all accessible objects.
  - Stick accessible objects into a queue or stack.
    - queue: breadth-first traversal
    - stack: depth-first traversal
  - Loop until queue/stack is empty:
    - remove marked object (say *x*) from queue/stack
    - if *x* points to an (unmarked) object *y*, then mark y and put it in the queue
- Sweep phase
  - Consider each object:
    - If it is not marked, put on the free list (i.e., deallocate it)
    - If it is marked, clear the mark bit

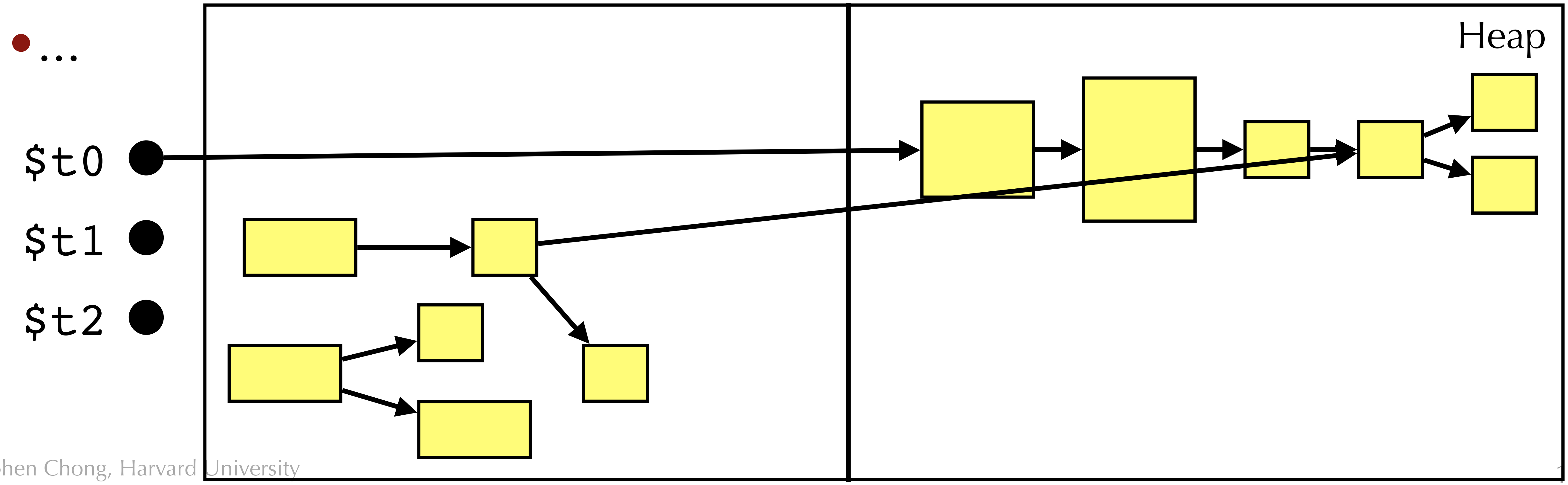
Stop and Copy Collector

- Split the heap into two pieces.
- Allocate in 1st piece until it fills up.
- Copy the reachable data into the 2nd area, compressing out the holes corresponding to garbage objects.
- Can now reclaim all of the 1st piece!
- Allocate in 2nd piece until it fills up
- ...



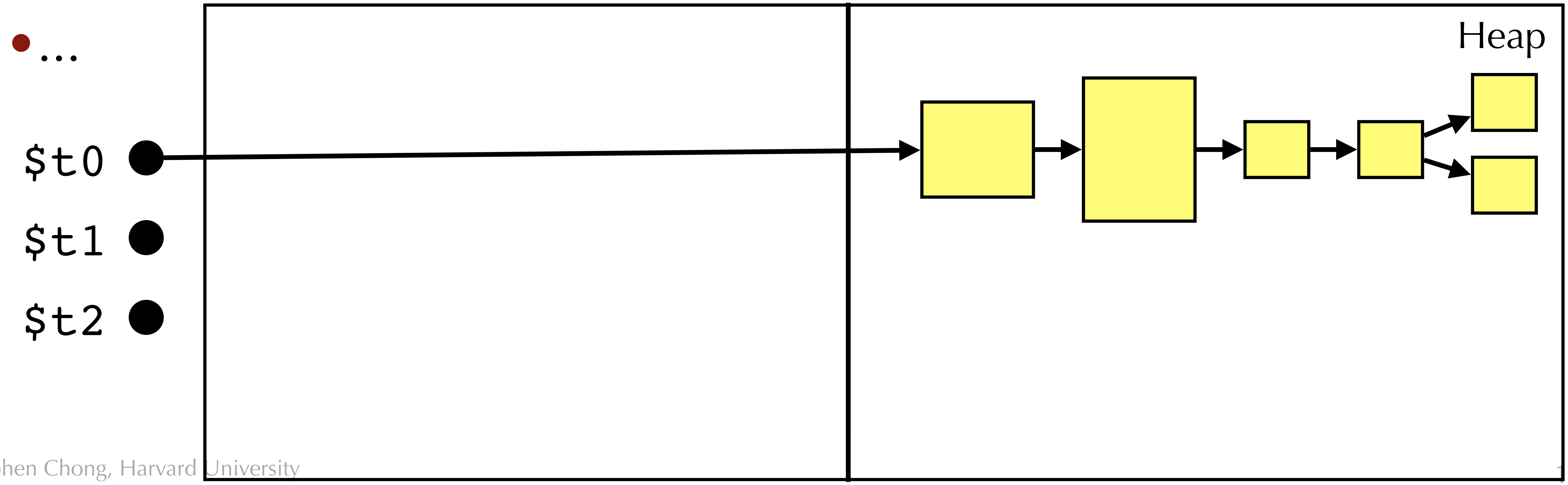
Stop and Copy Collector

- Split the heap into two pieces.
- Allocate in 1st piece until it fills up.
- Copy the reachable data into the 2nd area, compressing out the holes corresponding to garbage objects.
- Can now reclaim all of the 1st piece!
- Allocate in 2nd piece until it fills up
- ...



Stop and Copy Collector

- Split the heap into two pieces.
- Allocate in 1st piece until it fills up.
- Copy the reachable data into the 2nd area, compressing out the holes corresponding to garbage objects.
- Can now reclaim all of the 1st piece!
- Allocate in 2nd piece until it fills up
- ...

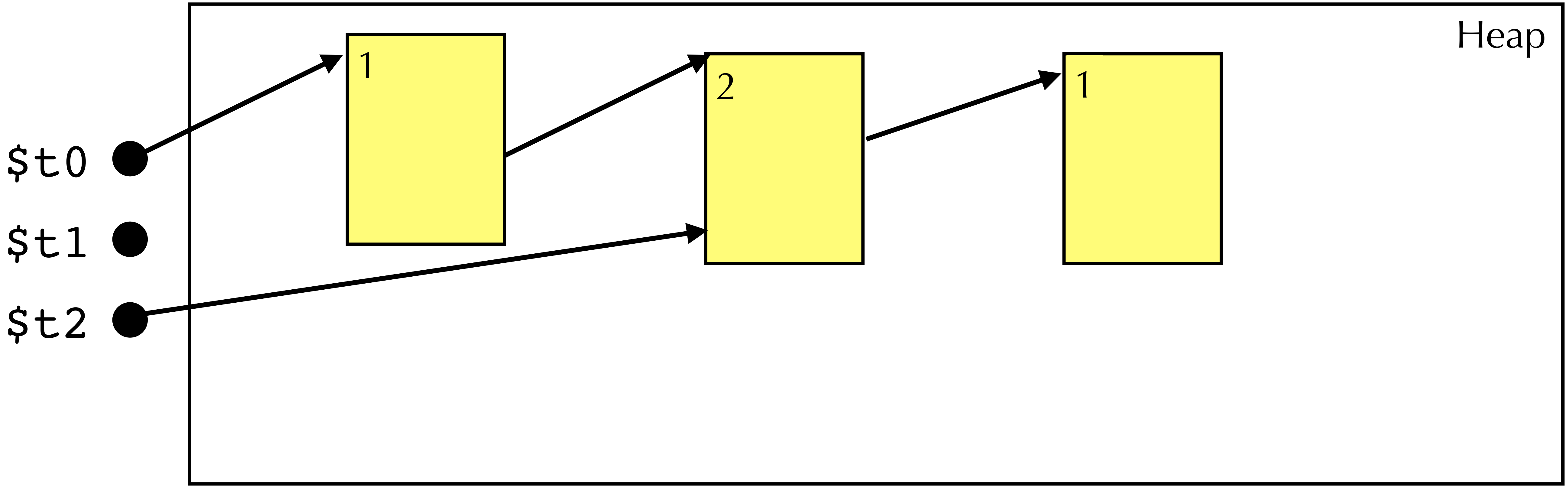


Generational Collection

- In many programs, newly created objects are likely to die soon
- Conversely, objects that survive many collections will probably survive many more collections
- So: collector should concentrate effort on "young" data (where there is higher proportion of garbage)
- Key idea: Divide heap into **generations**
  - Allocate new objects into generation G_0
  - Collect G_0 frequently, G_1 less frequently, G_2 even less so, ...
  - If object survives 2-3 collections in G_i, copy it into G_{i+1}
- Roots now include pointers from older generations to younger ones
  - Relatively rare
  - But need mechanism to remember them

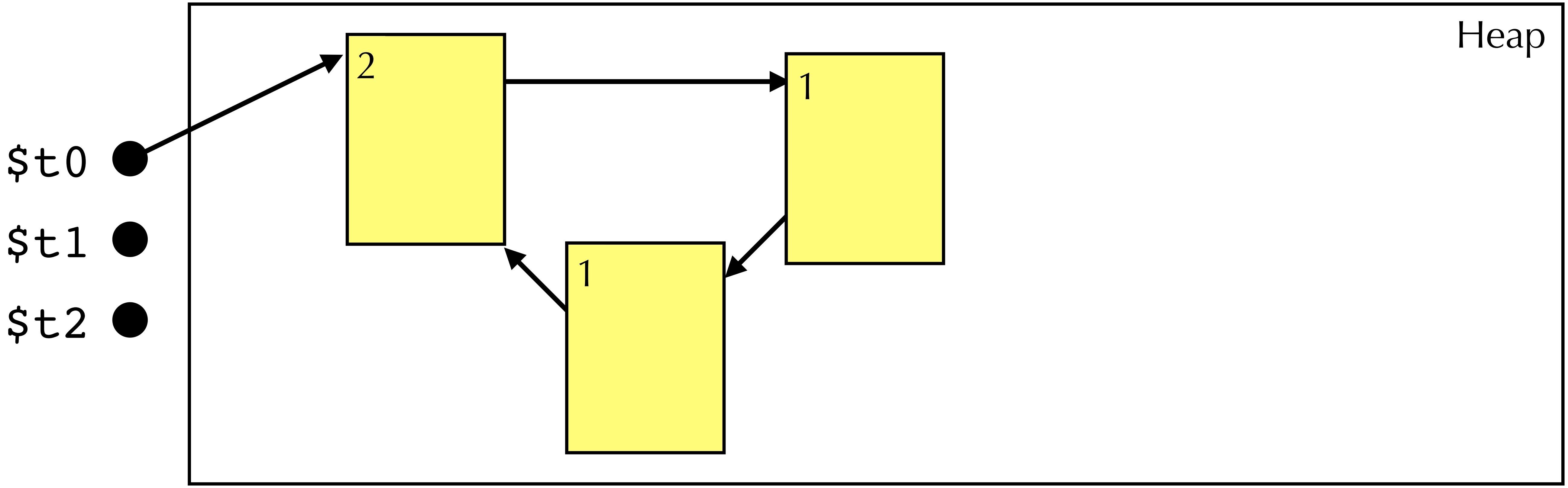
Reference Counting

- Key idea: track how many pointers point to each object
  - The **reference count** of the object, stored with object
  - Compiler modifies stores to increment/decrement reference counts
  - If reference count reaches 0, free object!



Reference Counting

- Any problems?
- What about cycles of garbage?
  - Require programmer to break cycles
  - Or do occasional mark-sweep collection



Incremental Collection
Concurrent Collection

- Collector will occasionally interrupt program for long periods of time for garbage collection
  - Problem for interaction or realtime programs!
- **Incremental collection** performs some work on garbage collection when the program requests it
- **Concurrent collection** performs garbage collection concurrently with program
- Can greatly reduce latency!

Reality

- Large objects (e.g., arrays) can be copied “virtually" without a physical copy.
- Some systems use mix of copying collection and mark/sweep with compaction.
- A real challenge is scaling to server-scale systems with terabytes of memory…
- Interactions with OS matter a lot: cheaper to do GC than to start paging…
- Java has a variety of GCs available with different tradeoffs
  - Default is generational collector that uses multiple threads when it runs
- OCaml uses a generational/incremental collector, invoked only in allocation

Conservative Collectors

- Work without help from the compiler.
  - e.g., legacy C/C++ code.
- Cannot accurately determine which values are pointers.
  - But can rule out some values (e.g., if they don't point into the data segment.)
- So they must conservatively treat anything that looks like a pointer as such.
- What happens if we have a value we aren’t sure is a pointer or not?
  - Two bad things: leaks, can't move the object!

The Boehm Collector

- Based on mark/sweep.
  - Performs sweep lazily
- Organizes free lists as we saw earlier.
  - Different lists for different sized objects.
  - Relatively fast (single-threaded) allocation.
- Most of the cleverness is in finding roots:
  - global variables, stack, registers, etc.
- And determining values aren't pointers:
  - e.g., blacklisting (recording values that aren’t pointers but are in vicinity of heap)

Are We Done with Runtimes?

- Garbage collection takes care of managing an important resource: memory
- **Work-stealing** takes care of managing cores/processors

Work-Stealing

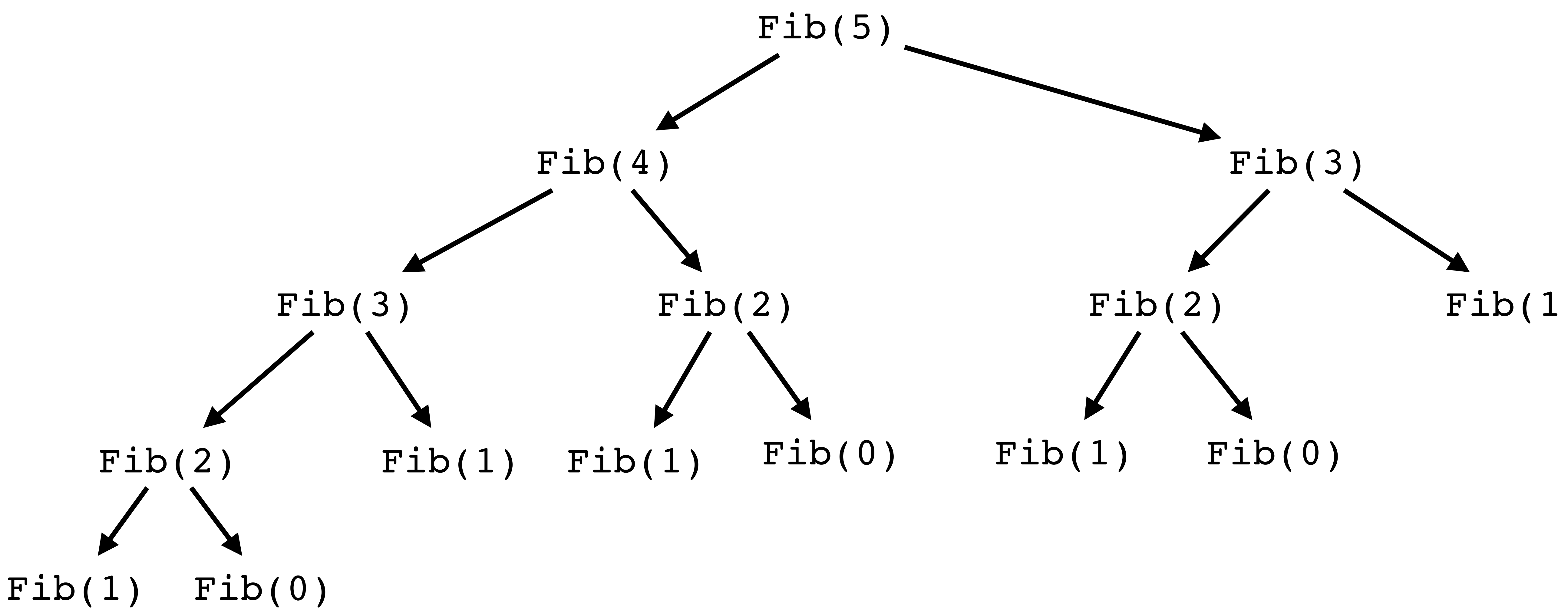
- Number of worker threads
  - Each thread has a work dequeue (double-ended queue)
  - Typically one or more threads per core
- A thread pushes and pops work from front of its dequeue
- When out of work, a thread steals work from back of dequeue of randomly selected “victim” thread

Work Stealing: Example

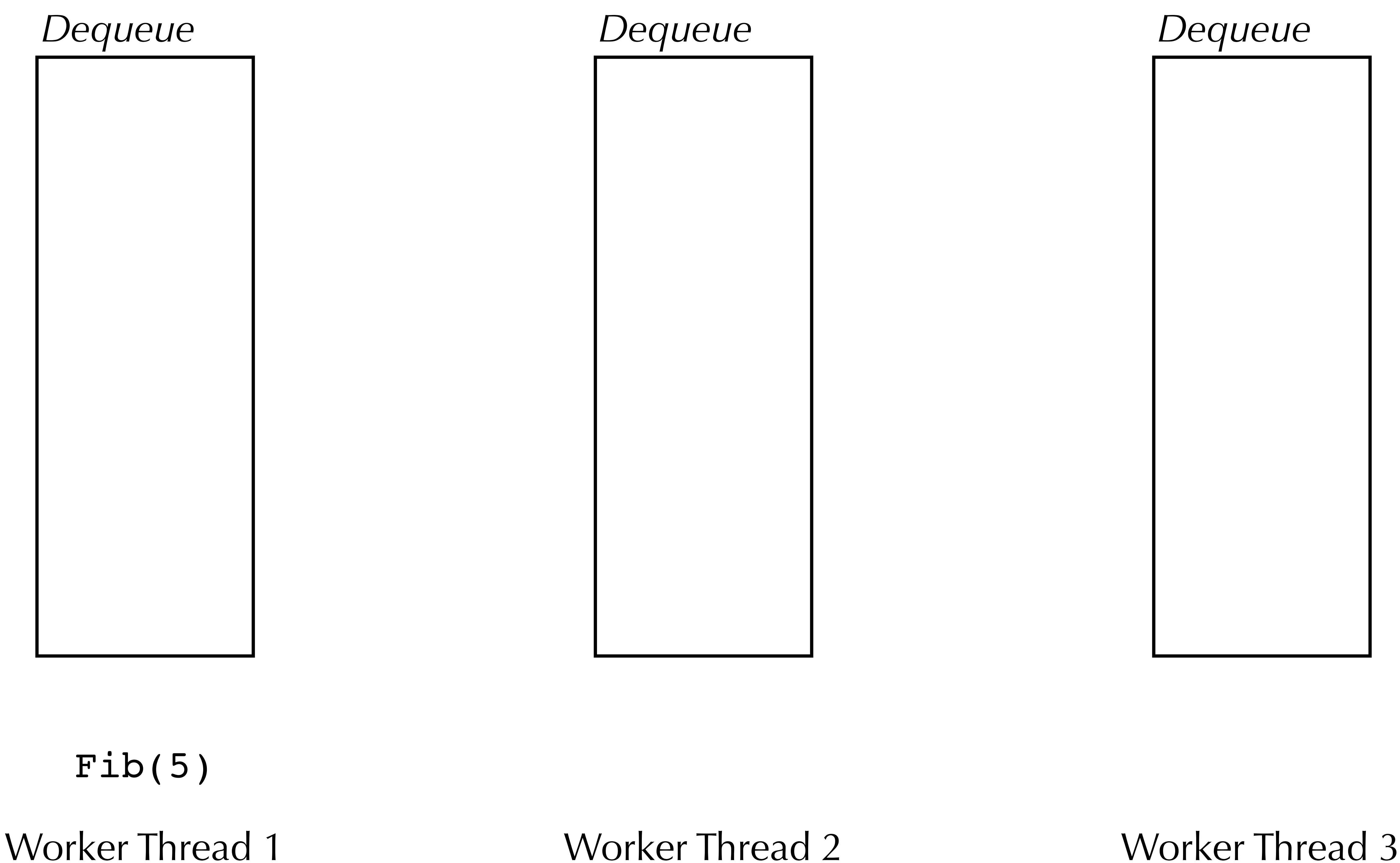
```
class Fibonacci extends RecursiveTask<Integer> {
  final int n;
  Fibonacci(int n) { this.n = n; }
  Integer compute() {
    if (n <= 1) return n;
    Fibonacci f1 = new Fibonacci(n - 1);
    Fibonacci f2 = new Fibonacci(n - 2);
    f1.fork();
    f2.fork();
    return f1.join() + f2.join();
  }
}
```

- NB: This is a dumb Fibonacci to illustrate work-stealing

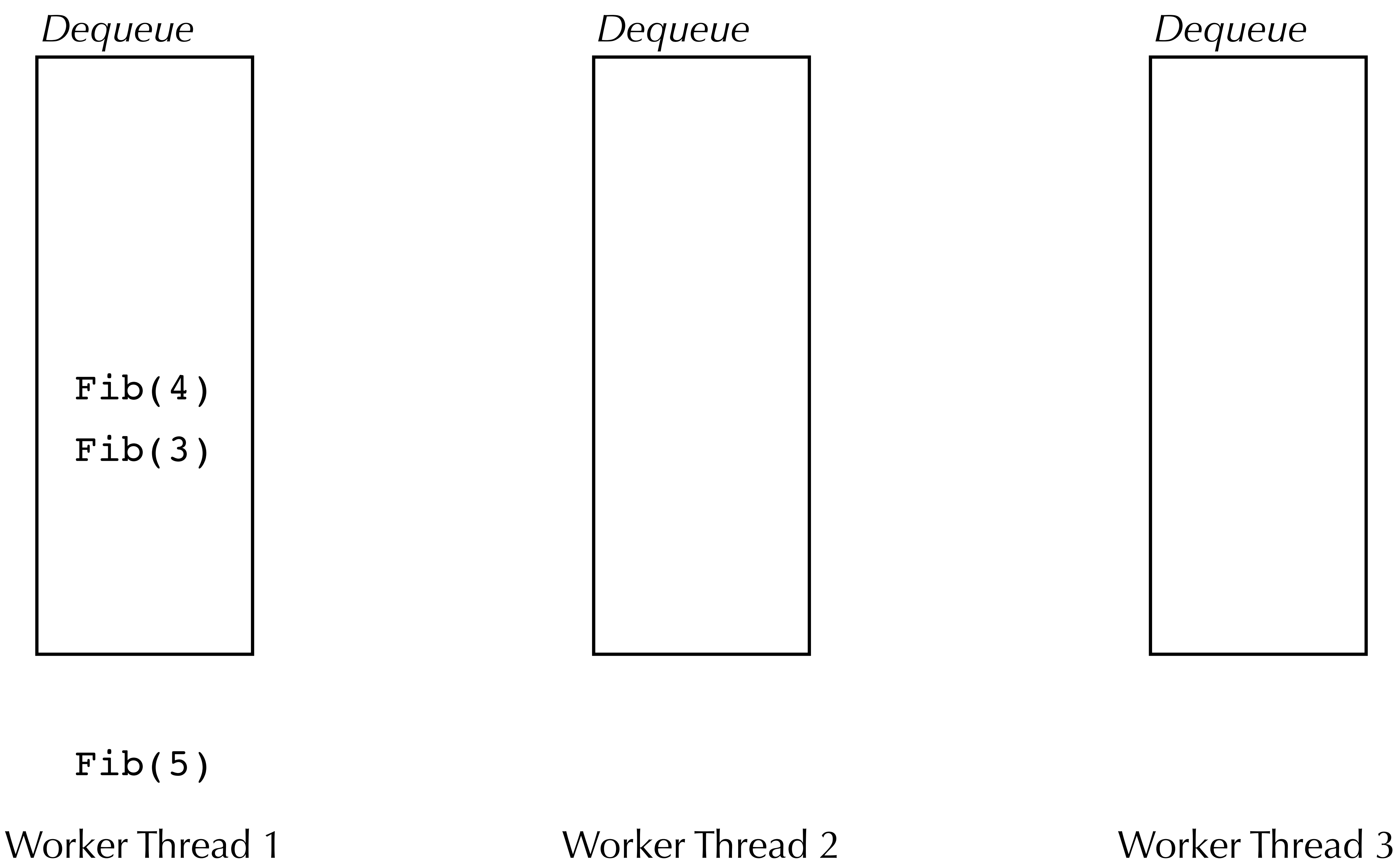
Computation Tree



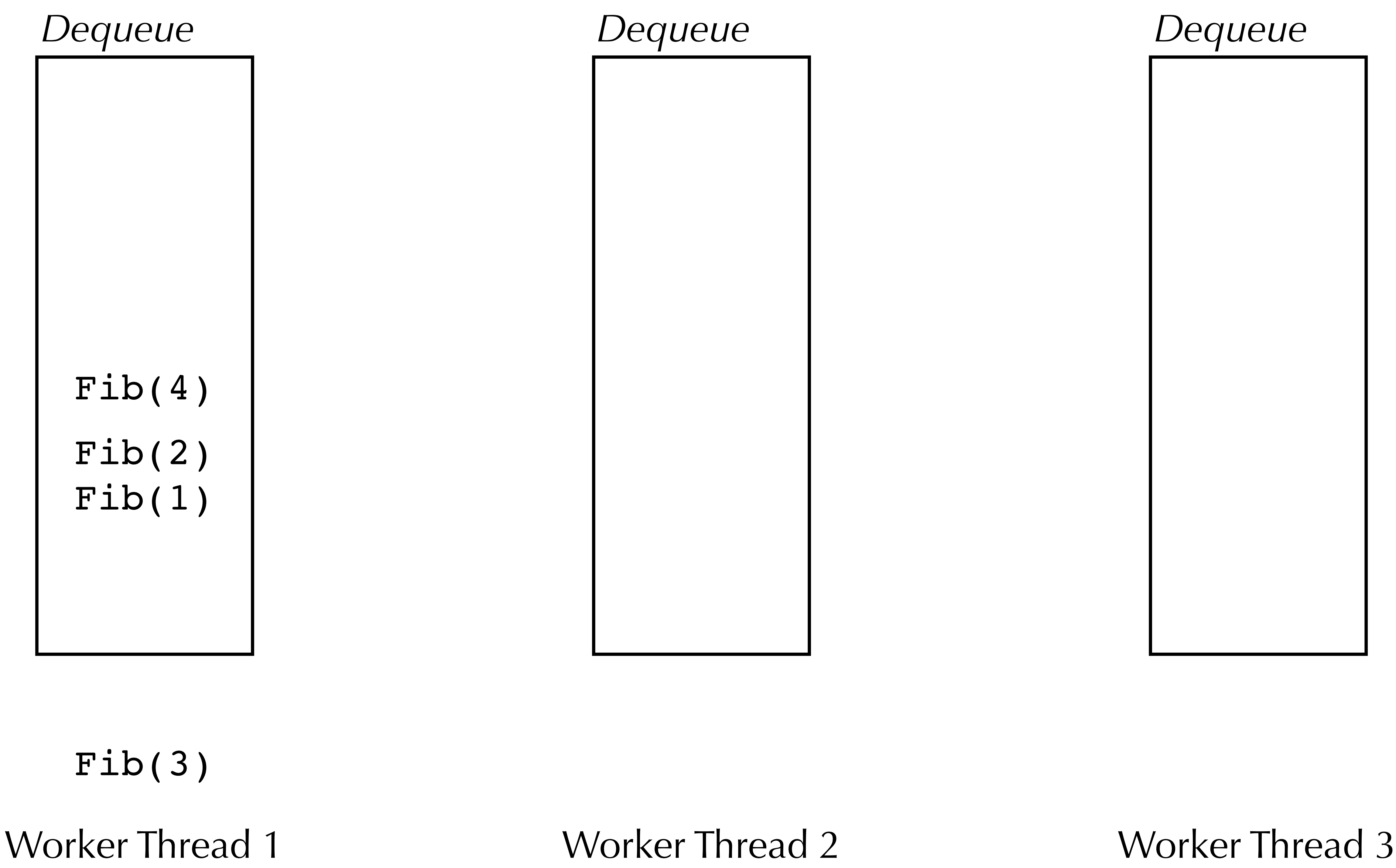
Work Stealing: Example



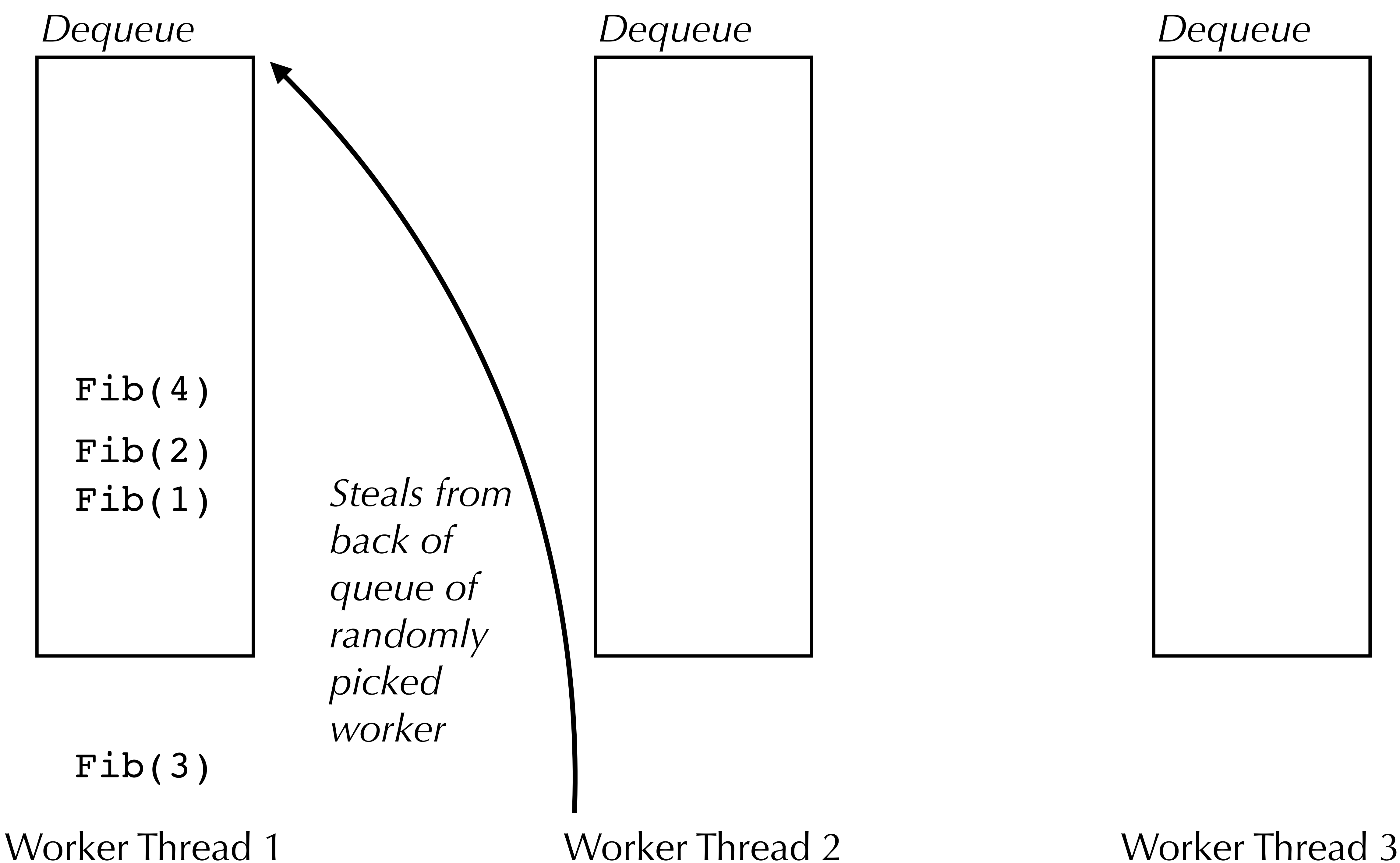
Work Stealing: Example



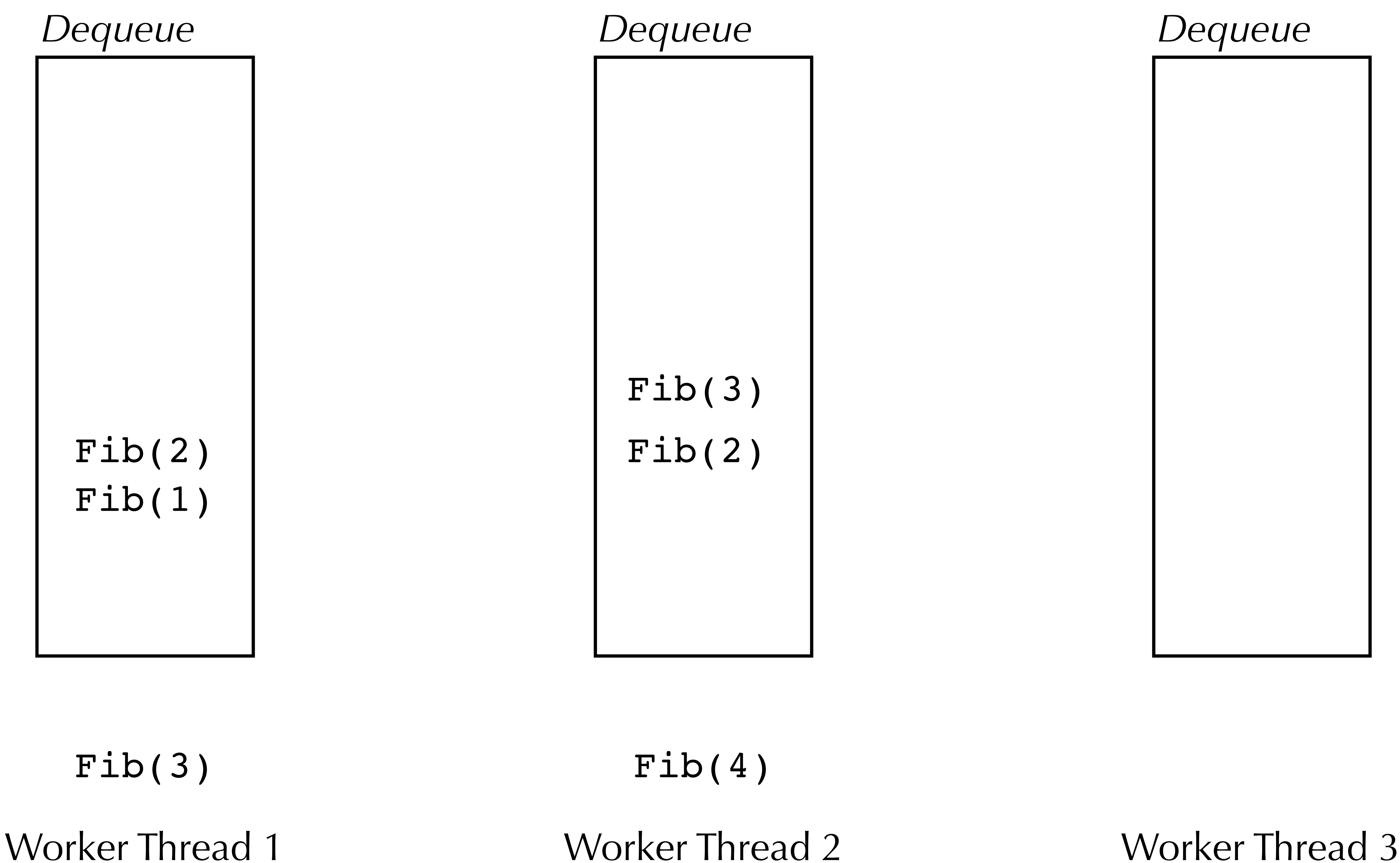
Work Stealing: Example



Work Stealing: Example



Work Stealing: Example



Virtual Machines

- Some languages are neither interpreted nor compiled to native code
- Instead the compiler generates code in a virtual assembly language (called **bytecode**)
- At runtime, the bytecode is interpreted by a virtual machine
- Sometimes, the runtime can compile important code further to native code on the fly. This is called **Just-In-Time compilation**
- Bytecode facilitates portability
  - Bytecode typically easier to implement than full language

Example: Java

```
public class Hi {

    public static void main(String[] args) {
        System.out.println("Hi");
    }

}
```

- Running `javac Hi.java` generates `Hi.class`

Example: Java

- Running `javap -cp . -p Hi` produces

```
Compiled from "Hi.java"
public class Hi {
  public Hi();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3                  // String Hi
       5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

- Try running `javap -cp . -v Hi` to see more details of the class file

Other Virtual Machines

- OCaml has a bytecode representation
- LLVM (a popular modern compiler) has a bytecode representation
  - Typically an intermediate representation en route to native code
- Many dynamically-typed languages (JavaScript, Python) have bytecode representations and use JIT