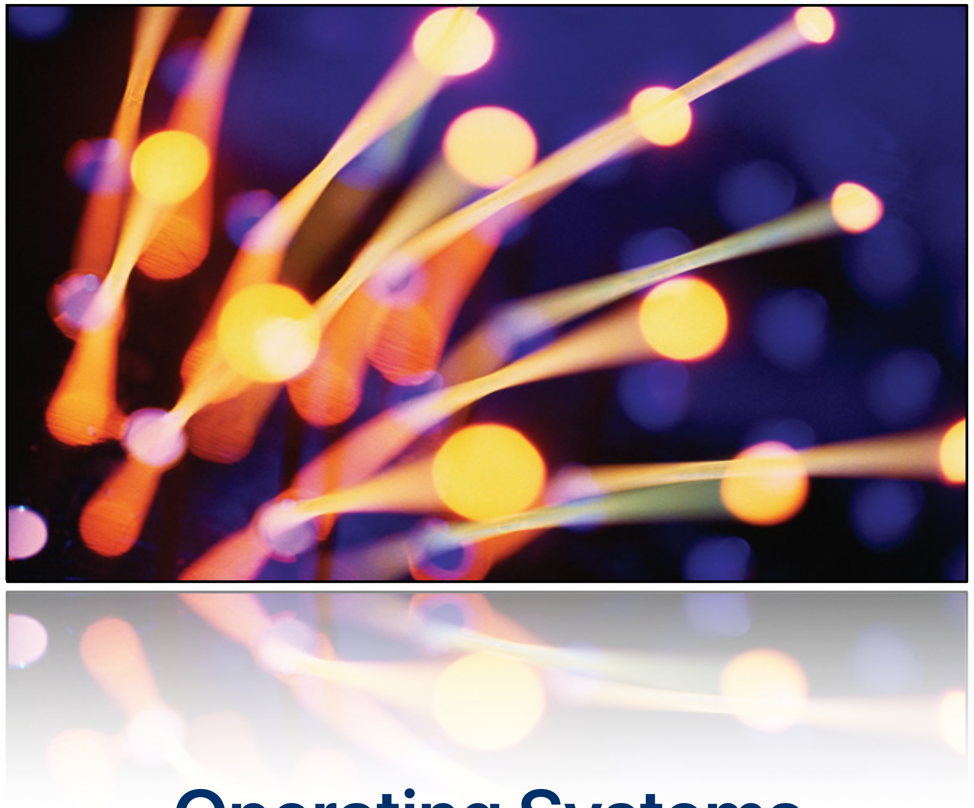
Threads: A Deep Dive

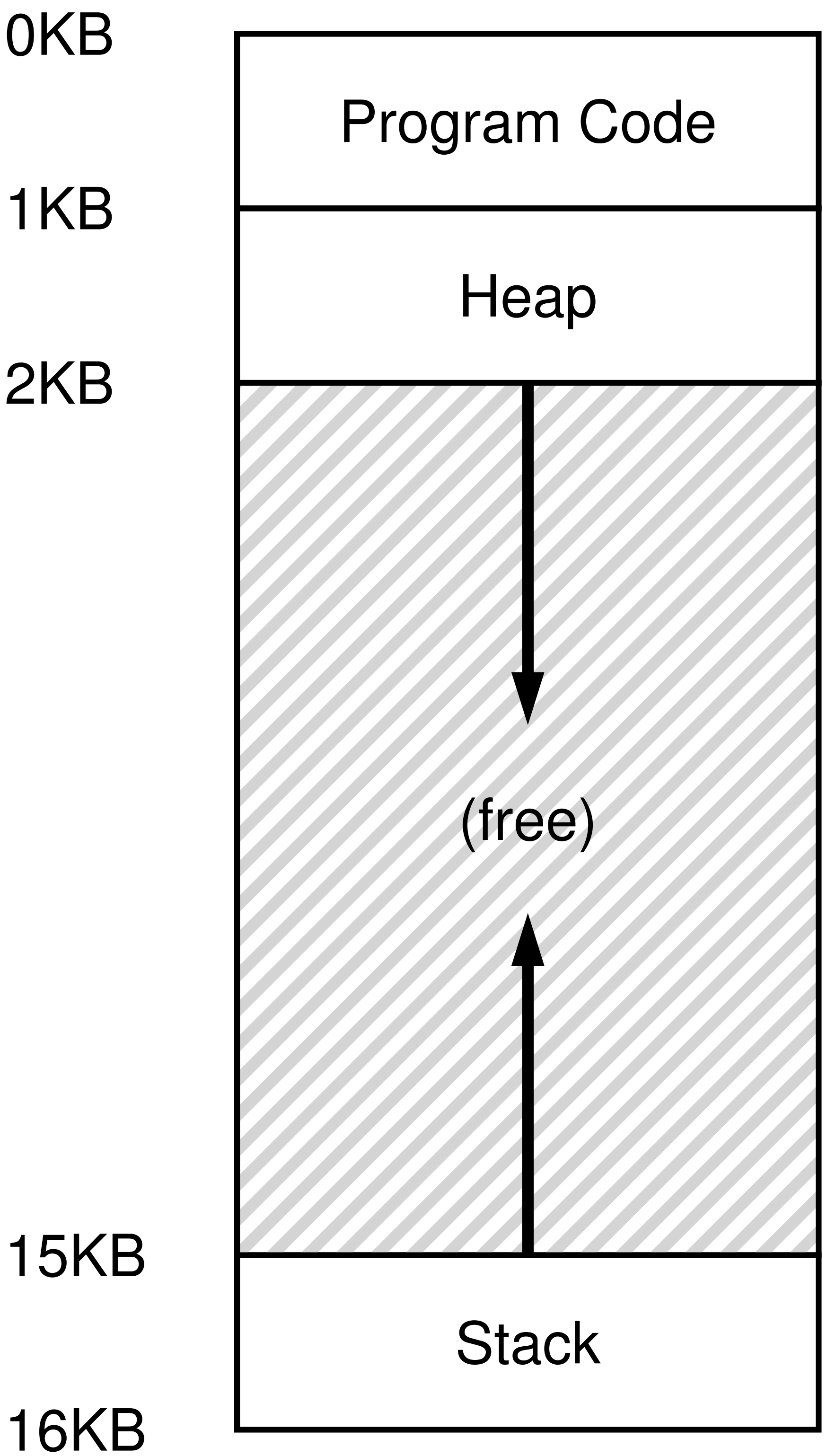


Operating Systems

Baochun Li

University of Toronto

Threads in a virtual address space

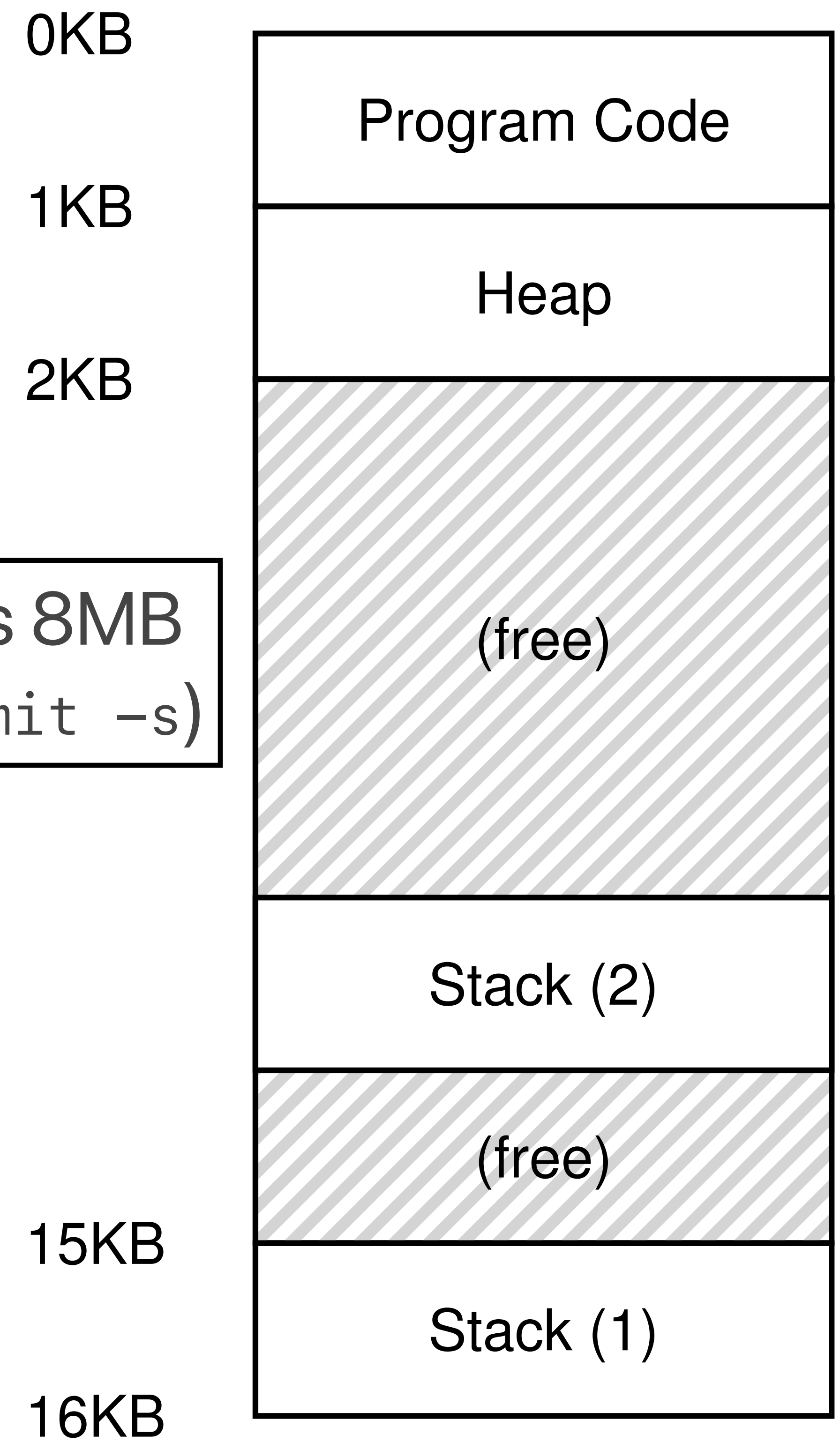


the code segment:
where instructions live

the heap segment:
contains malloc'd data
dynamic data structures
(it grows downward)

Stack size per thread is 8MB by default in UNIX (`ulimit -s`)

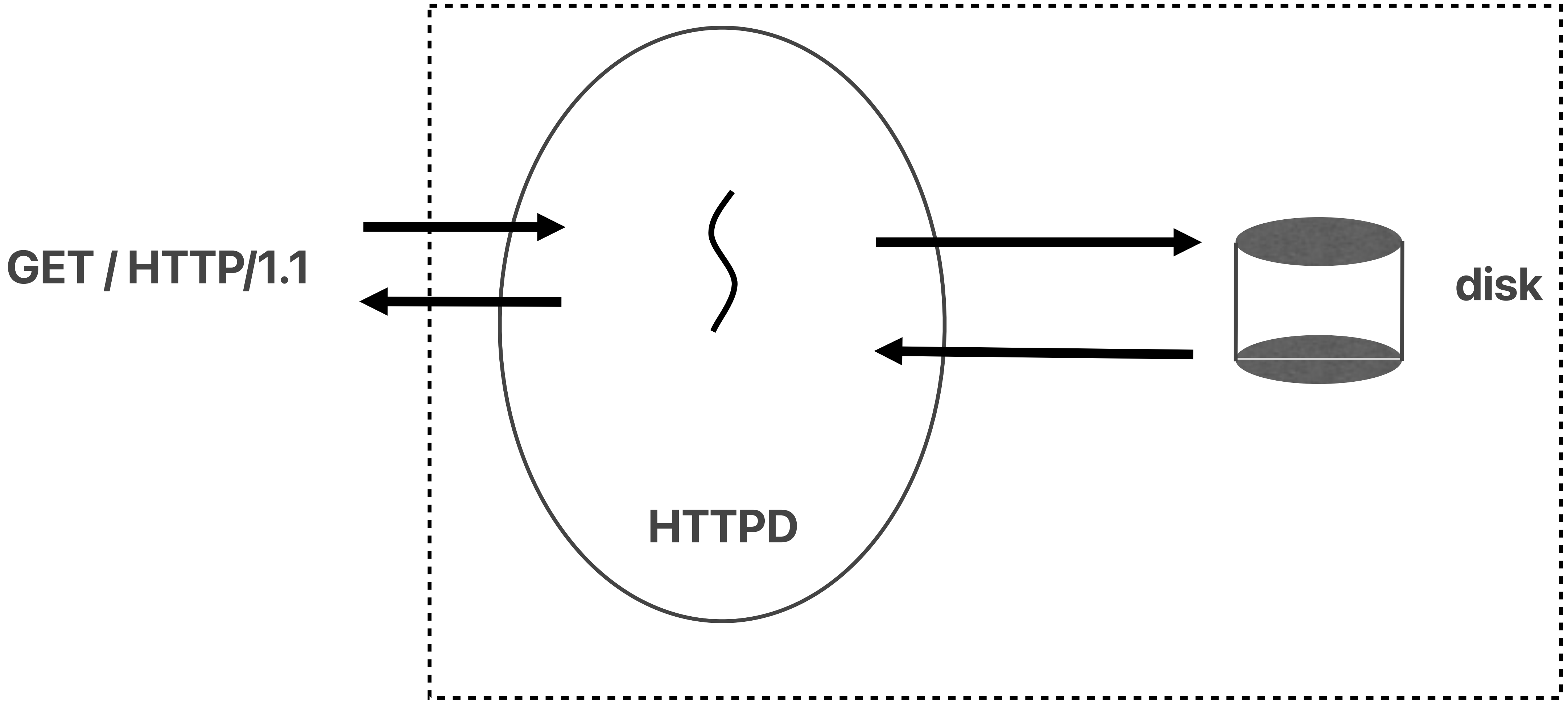
(it grows upward)
the stack segment:
contains local variables
arguments to routines,
return values, etc.



An Example of Processes vs. Threads

Consider a web server with a single-threaded process

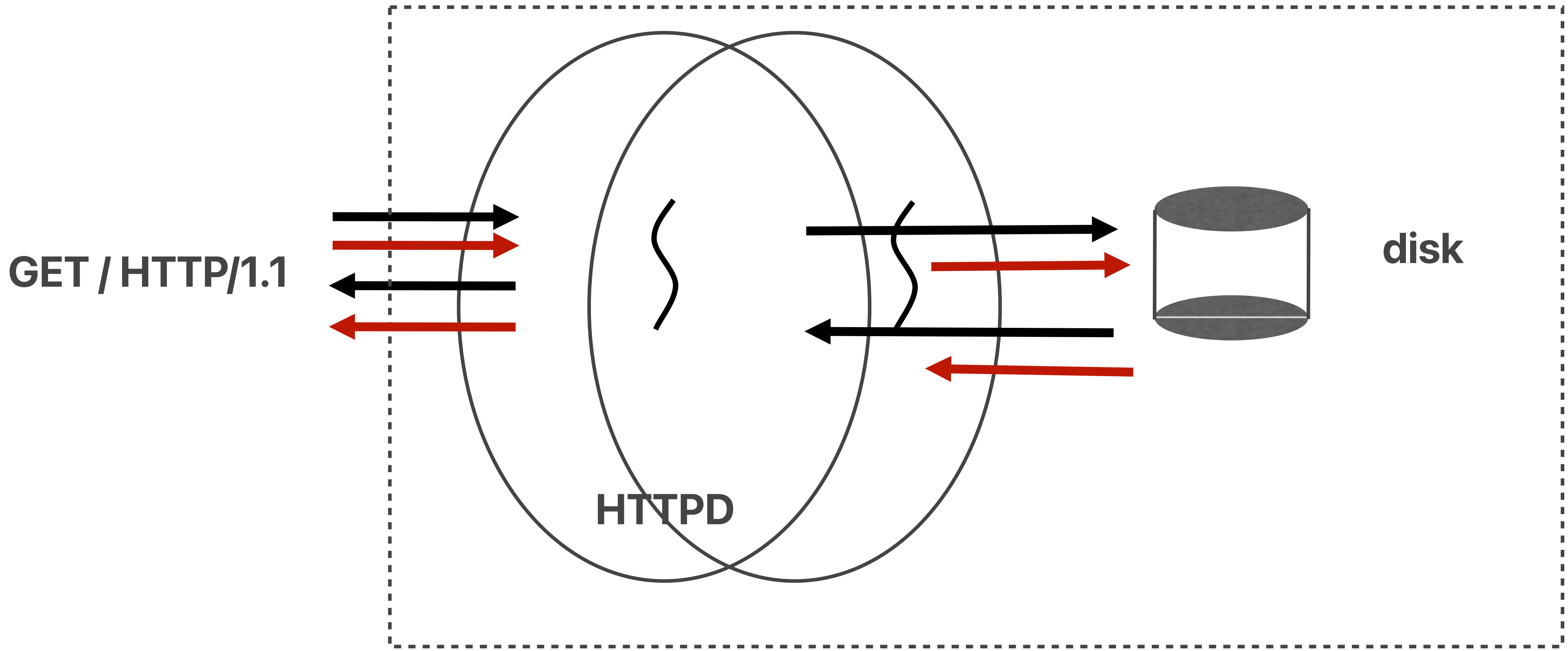
Why is this not a good web server design?



An Example of Processes vs. Threads

Consider a web server with multiple processes

Is there a problem with this web server design?



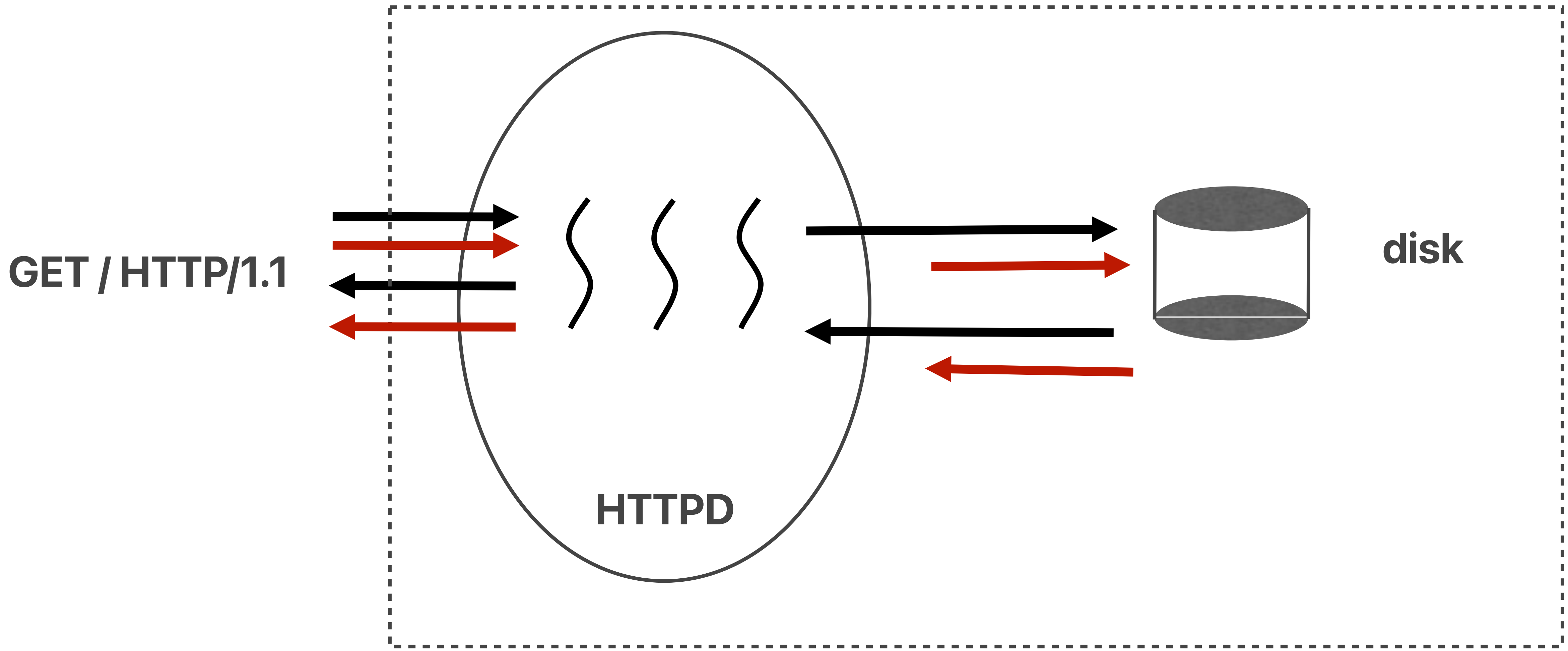
Why Do We Need Threads — Advantages

- **Low cost communication via shared address space**
- **Lightweight in thread creation, termination and switching**
  - Faster than processes by at least an order of magnitude
  - Context switching with processes is expensive!
- **Overlap computation and blocking (due to I/O) on a single CPU**
  - A simple model for handling asynchronous events
- **Meets the need of multi-processor (-core) systems well**

Back to our example

Consider a web server with multiple threads

Should we use one thread per client or per request?



A thread per client or a thread per request?

Thread-per-client: A new thread for each client

- What if a client has many concurrent requests?

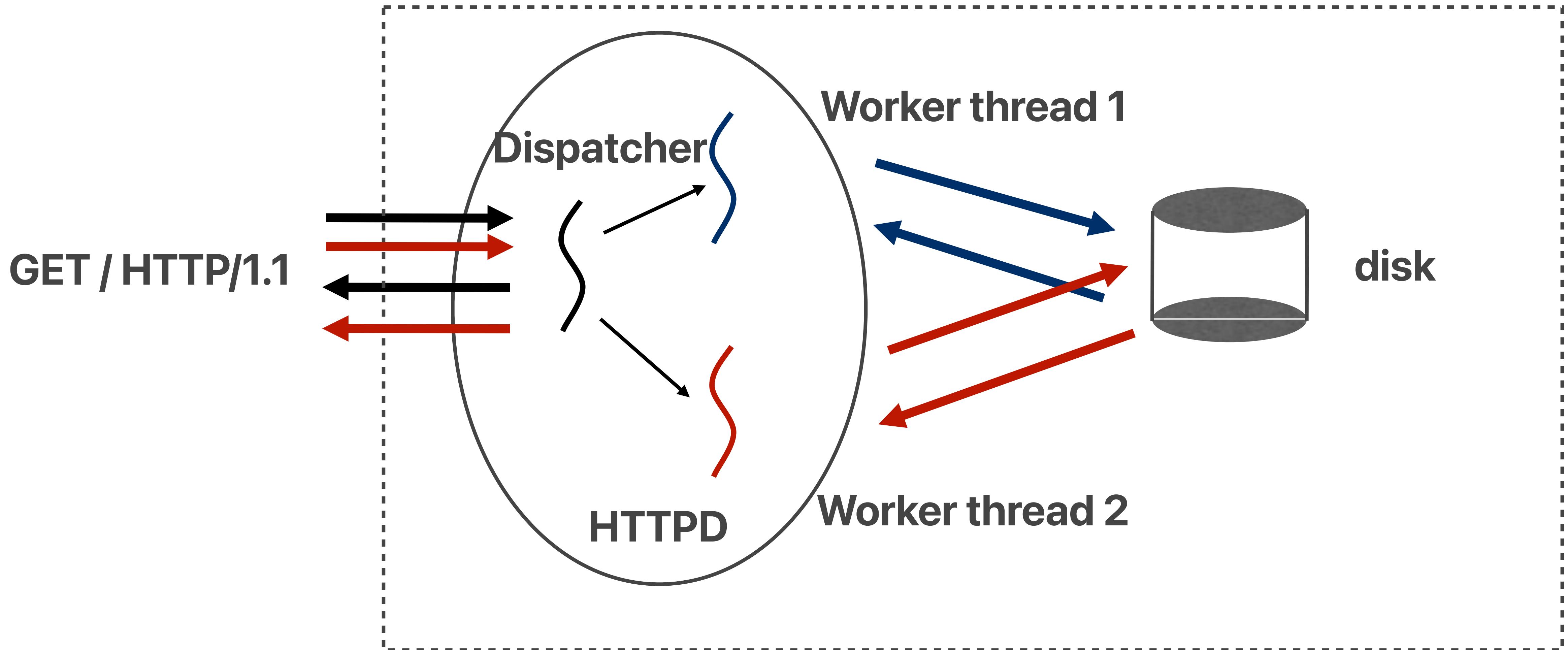
Thread-per-request: Create a new thread for each incoming connection request

- Drawback: When demand surges, too many threads lead to performance degradation

A Thread Pool Design of the Web Server

Now consider a multi-threaded web server using a thread pool

Is there a problem with this web server design?

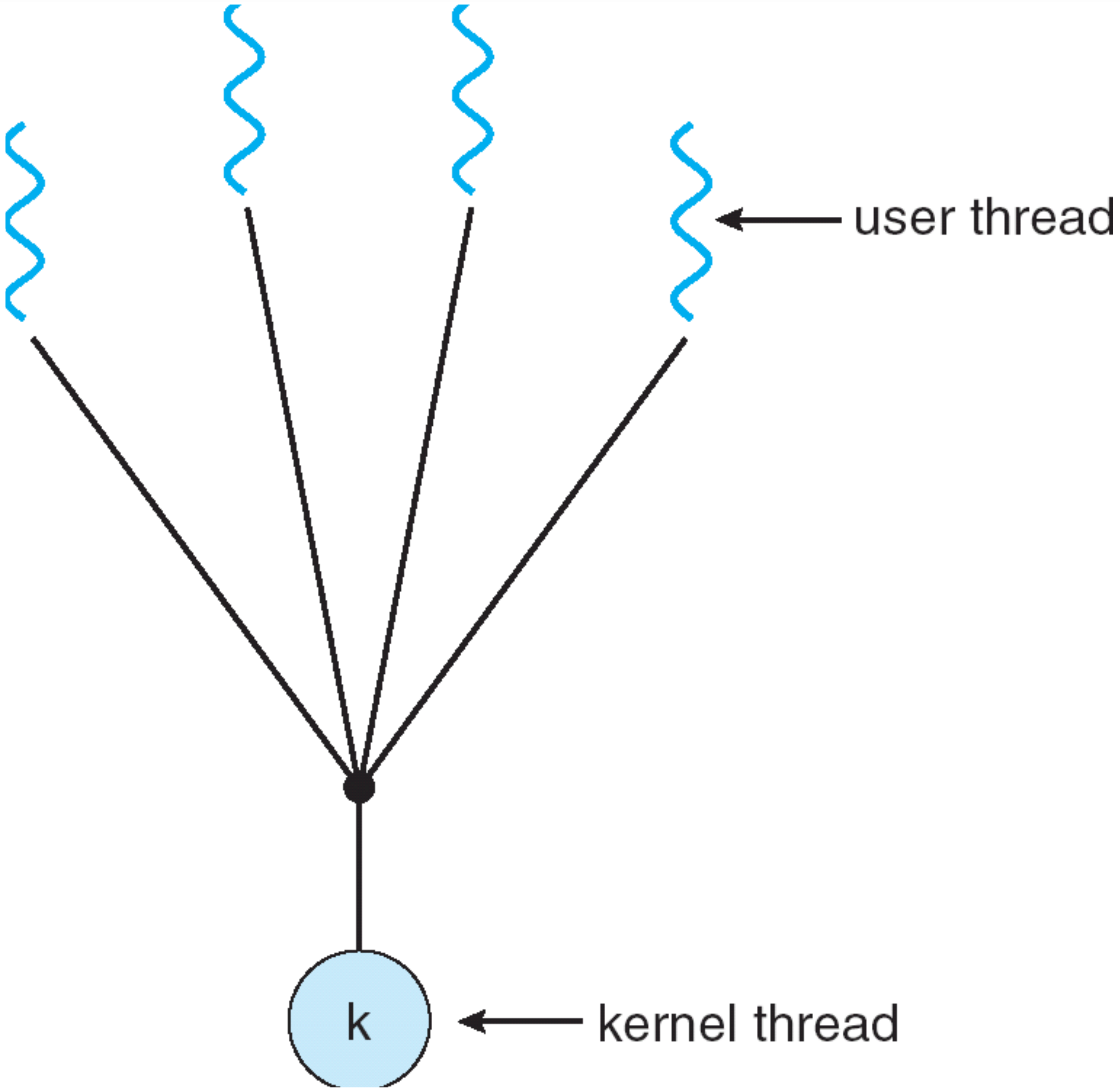


User Threads (N:1) — the Many-to-One Model

Thread scheduler manages thread contexts in the user space

- Each process needs its own private thread table to keep track of threads
  - keeps track of PC, SP, registers, state (ready, blocked, running)
- OS sees only a traditional process
- No need to modify OS kernels if they do not support threads initially

User Threads (N:1) — the Many-to-One Model



User Threads (N:1): Advantages

Context switching among threads in the same process is cheap

- No context switch to kernel and back to user level
- Can be done in time closer to procedure call
- Scales better to a very large number of threads

Allows each process to have its own customized scheduling algorithm

Example: GNU Portable Threads (Pth)

User Threads (N:1): Disadvantages

What happens with blocking system calls?

Letting one of the thread to block on the system call is not acceptable, as it will stop all the threads

All the system calls can be changed to non-blocking, but that requires changes to the OS, which defeats the purpose of using user threads

It may be possible to first check to see if blocking is necessary (using select()), before making the system calls

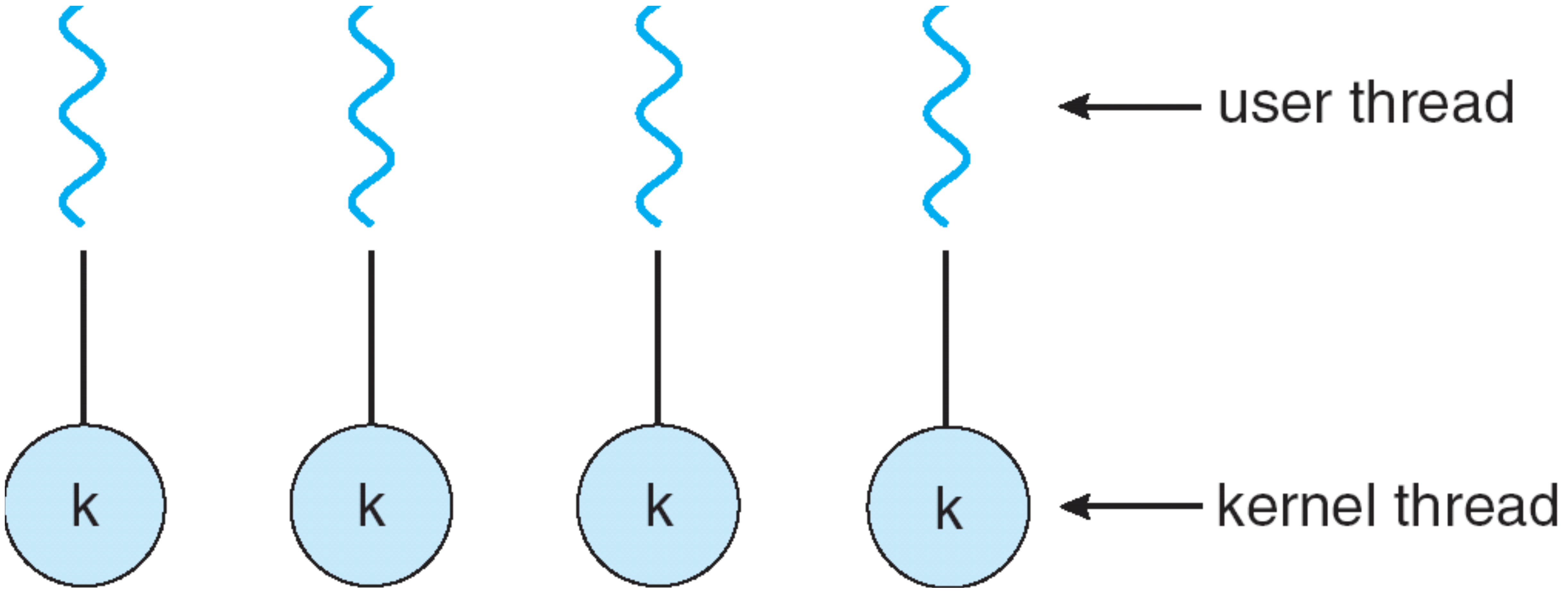
What if a thread does not give up the CPU?

the user thread scheduler cannot use timer interrupts — it is non-preemptive

What if we have multi-core CPUs?

User threads in the same process can only run one at a time

Kernel Threads (1:1) — The One-to-One Model



Kernel Threads (1:1) — The One-to-One Model

Advantage: Allows another thread to run when a thread makes a blocking system call

- No need to change blocking system calls to nonblocking

Disadvantage: The cost of a system call is substantial — much more overhead to create or switch across threads

All major operating systems: Windows, Linux (with the Native POSIX Thread Library), macOS

Hybrid Threads (M:N)

M user threads mapped onto N kernel entities (or virtual processors)

Advantage: Avoid expensive context switching among user threads that involve few system calls

Disadvantage: More complexity

Example: Windows user-mode scheduling

Disadvantage with the use of threads in general

The implementation must be thread-safe, and avoid all race conditions

If we use the N:1 or M:N model, we must implement a user-level thread scheduler (typically in a thread library)

It may be simpler to just use multiple processes! Remember the Apache web server?

An Alternative Design Without Threads?

If non-blocking system calls are available, we can design an asynchronous model in our example

- When a request comes in, the one and only thread (**per CPU core**) responds
- If needed, a nonblocking I/O operation is started
- The thread records the state of the current request, and then gets the next event
- The next event may either be a new request, or a reply from the I/O subsystem about the completion of a previous operation

The Asynchronous Model

The sequential nature in previous designs is lost

The state of computation must be saved at every switch from one request to another

It is a finite-state machine, as events trigger transitions across different states

It has an event-driven nature

Main advantage: no need to use more than one thread per CPU

The Asynchronous Model: Disadvantages

- **Requires event notification support from OS kernel**
  - Some kind of a "callback" mechanism
- **Used to design modern web servers: node.js**
  - Windows: **overlapped I/O with completion ports** (IOCP)
    - The earliest OS that supports this model
  - Linux (2.6 kernel): the **epoll** interface
  - macOS: the **kqueue** interface

What we’ve covered so far

Three Easy Pieces: Chapter 26.1 and 26.2 (Concurrency: An Introduction)