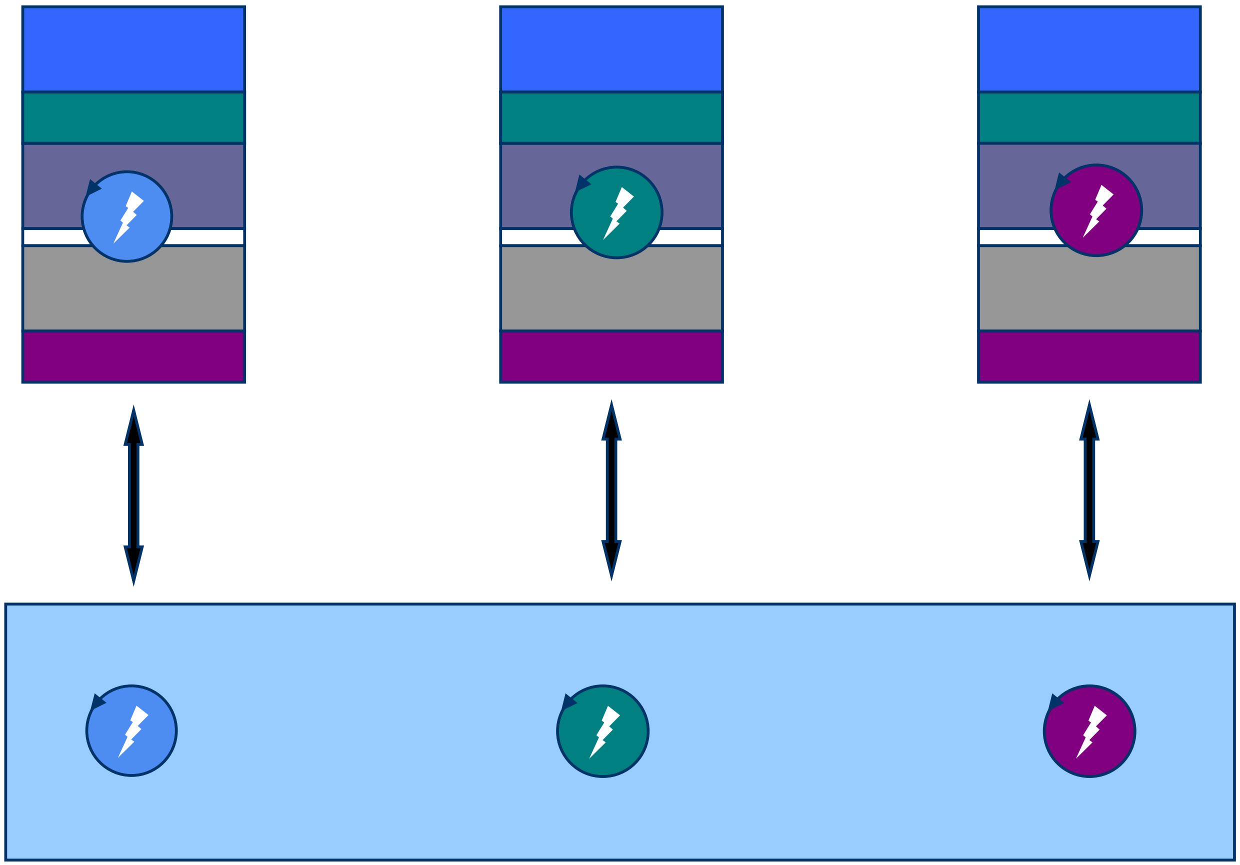
Duke Systems

CPS 310
Threads in the Kernel: See How they Block

Jeff Chase

Duke University

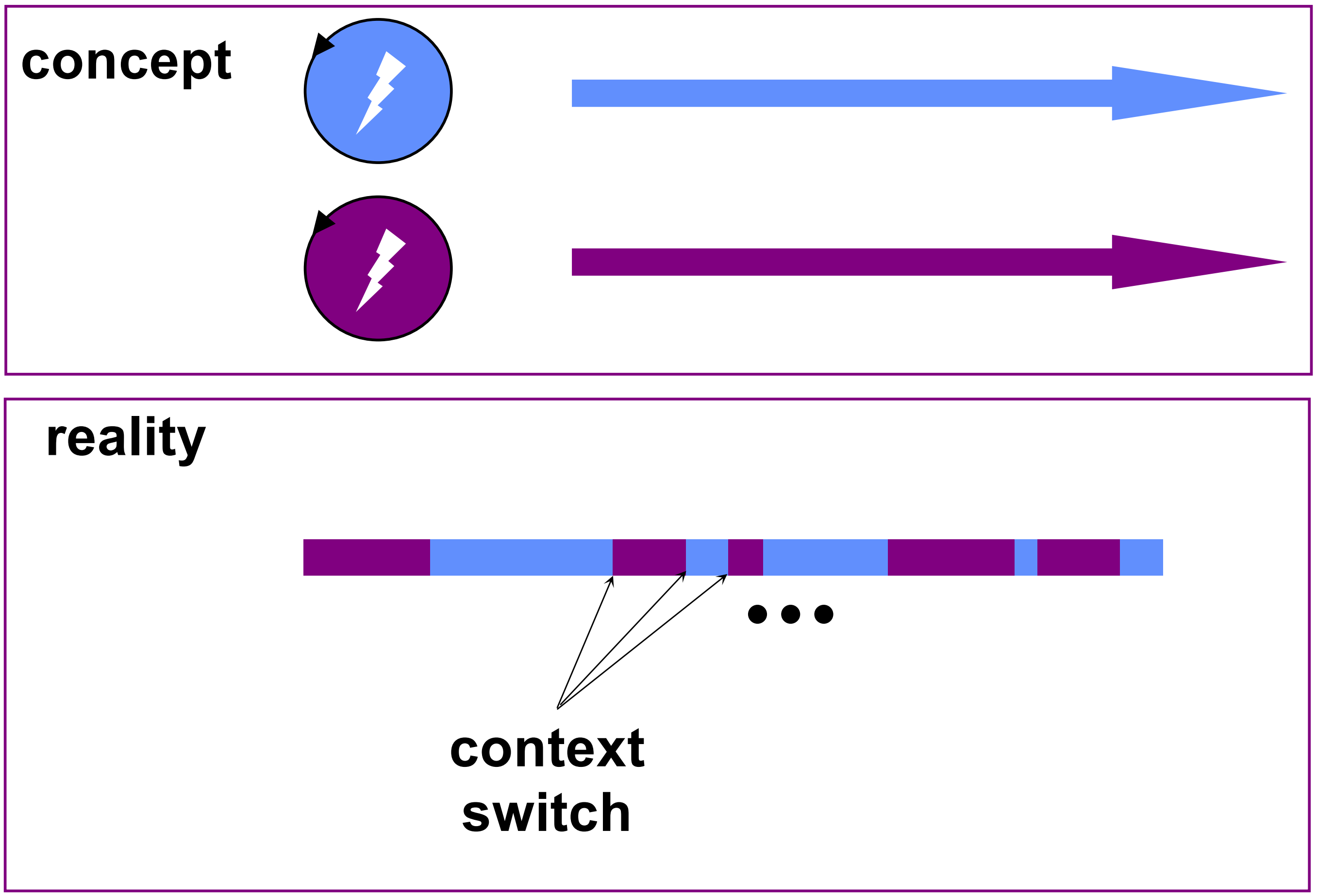
The story so far...



Simplify for today

- For today, and for the thread labs, let’s assume classic processes, with a single thread.
- Then we can use the terms “process” and “thread” interchangeably.
  - (As Linux people would do anyway.)
- Let’s also assume a uniprocessor: one core, one slot.
  - Then we can ignore physical concurrency. There is only logical concurrency.
  - Which comes from….**what**?

Two threads sharing a CPU/core



The core-and-driver analogy

zipcar®

wheels when you want them

Core #1

Core #2

The machine has a bank of CPU cores for threads to run on.

The OS allocates cores to threads (the “drivers”).

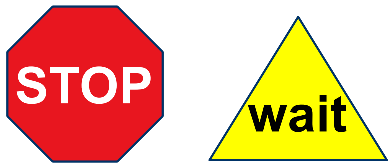
Cores are hardware. They go where the driver tells them.

OS can force a switch of drivers any time.

What causes a context switch?

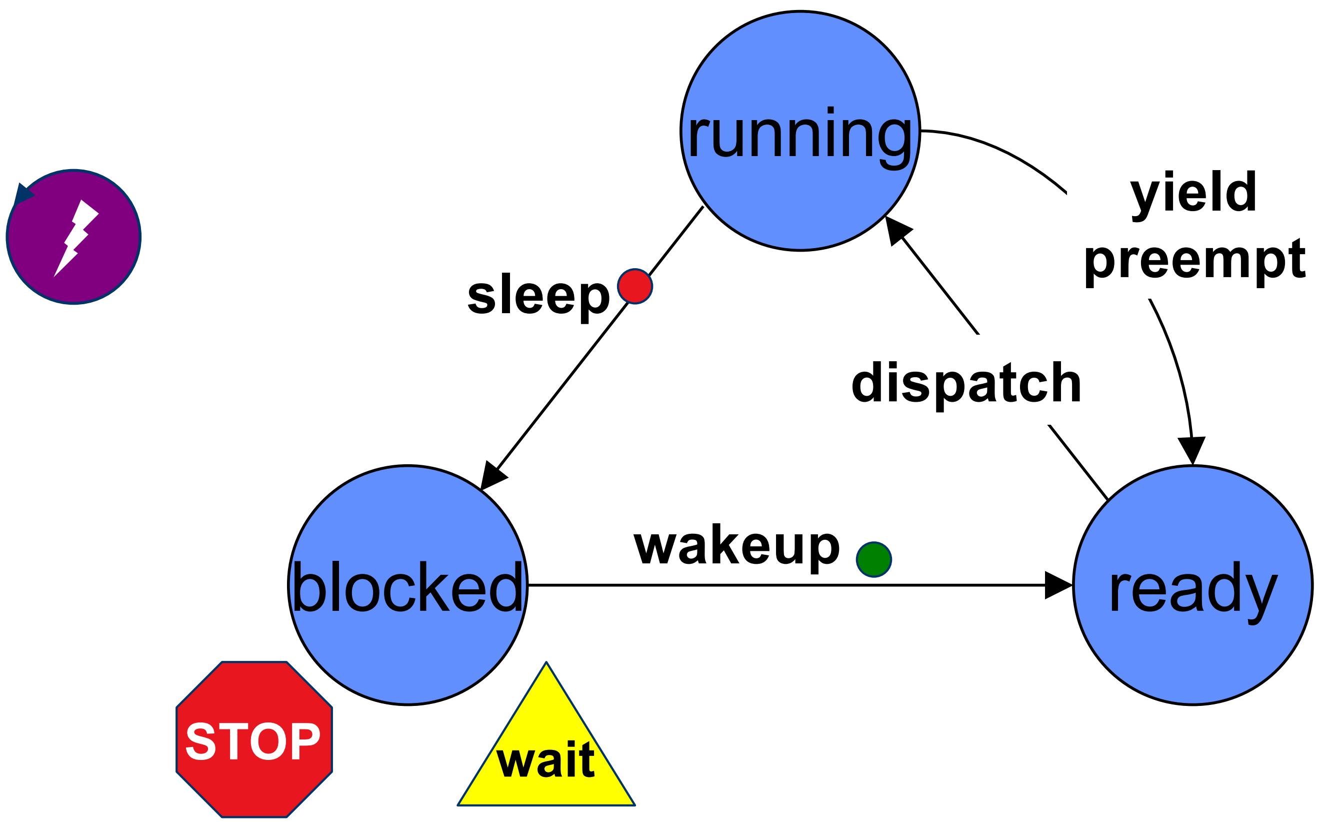
There are three possible causes:

1. Preempt (yield). The thread has had full use of the core for long enough. It has more to do, but it's time to let some other thread "drive the core".
   - E.g., timer interrupt, quantum expired → OS forces yield
   - Thread enters Ready state, goes into pool of runnable threads.
2. Exit. Thread is finished: "park the core" and die.
3. Block/sleep/wait. The thread cannot make forward progress until some specific occurrence takes place.
   - Thread enters Blocked state, and just lies there until the event occurs. (Think "stop sign" or "red light".)



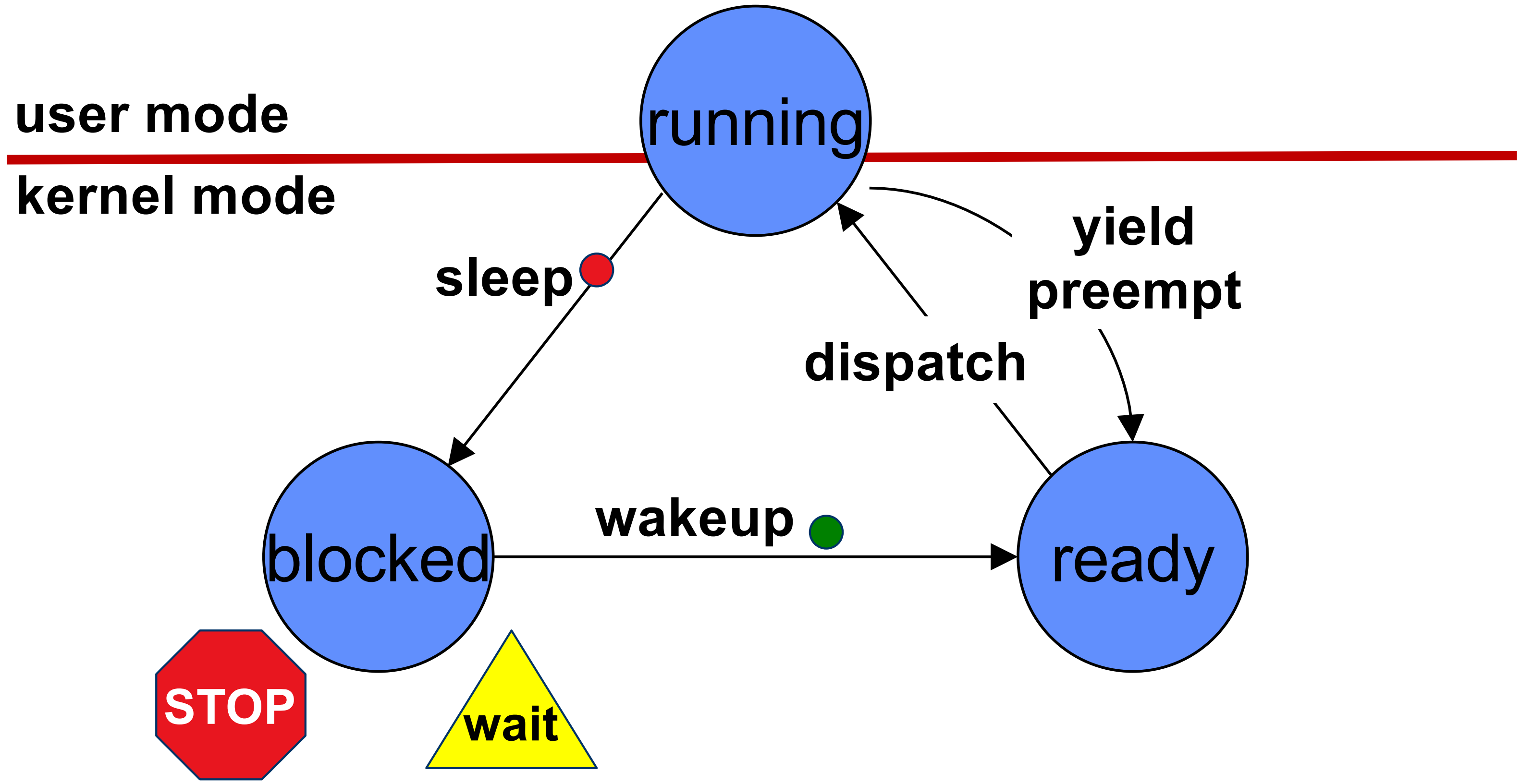
Thread states

Concept: block/sleep/wait



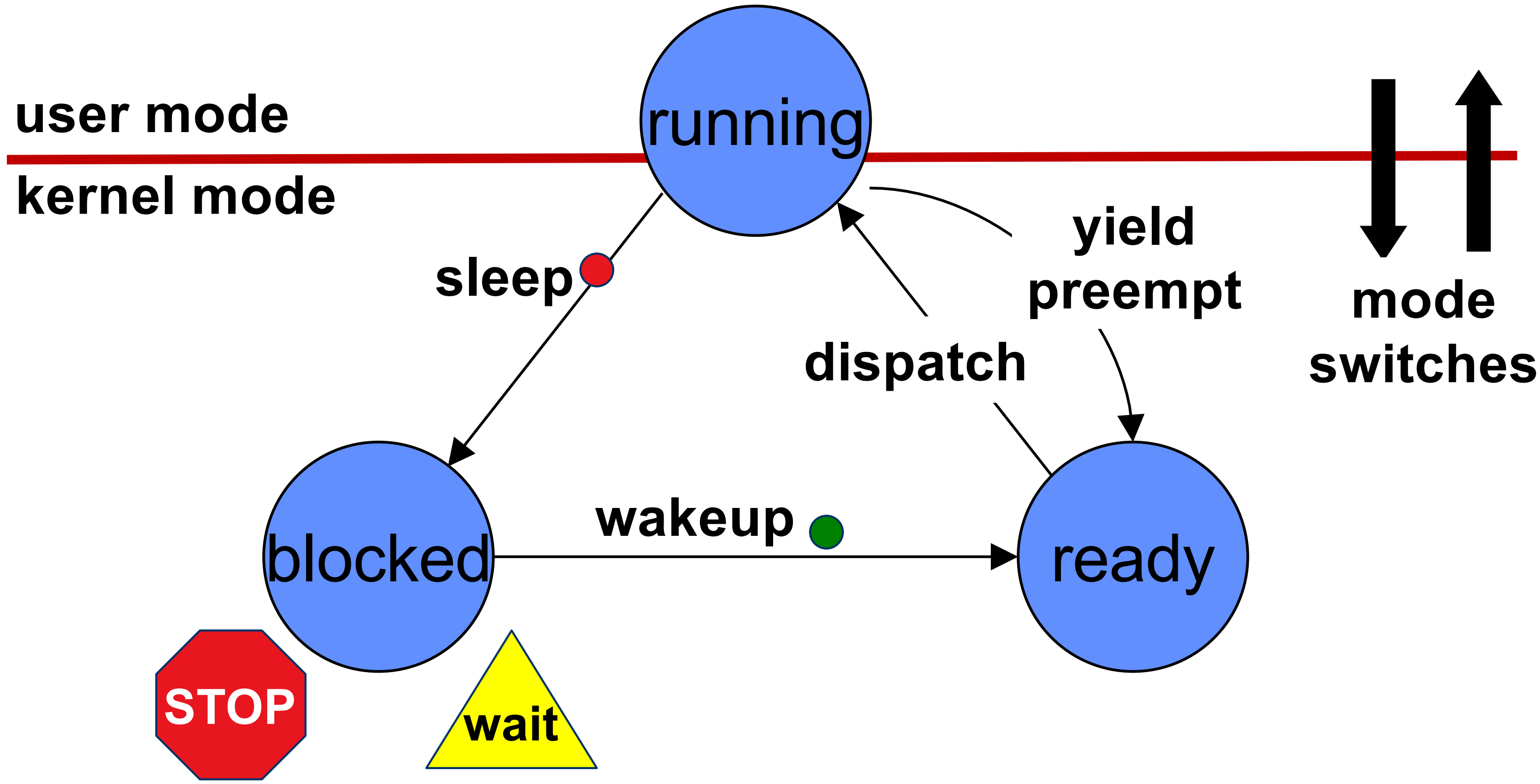
Often a thread's execution is blocked: it cannot continue until a designated event occurs. The thread waits for the event. We say that the thread **blocks** or **sleeps** and that the event **wakes** it up.

Thread states: user mode vs. kernel



In a "real" OS, threads/processes change states only in kernel mode.
Let us assume for today that our threads are in kernel mode

Mode switches

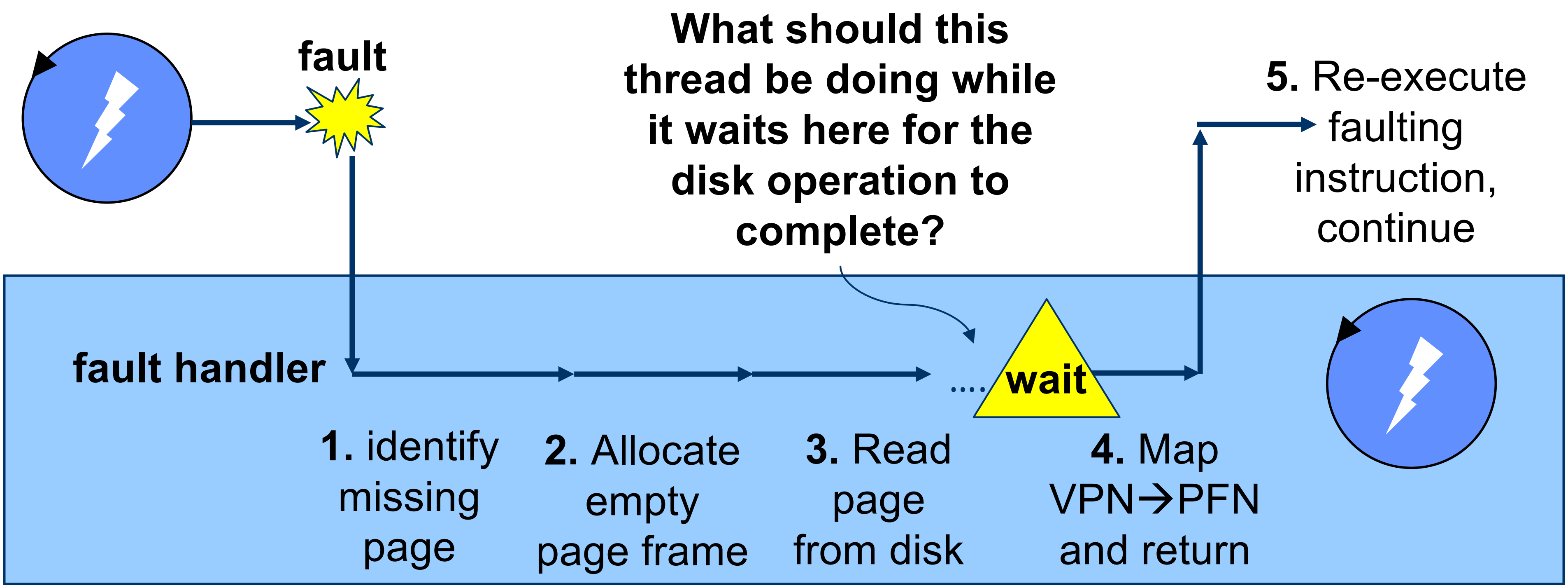


<u>Mode switches</u>

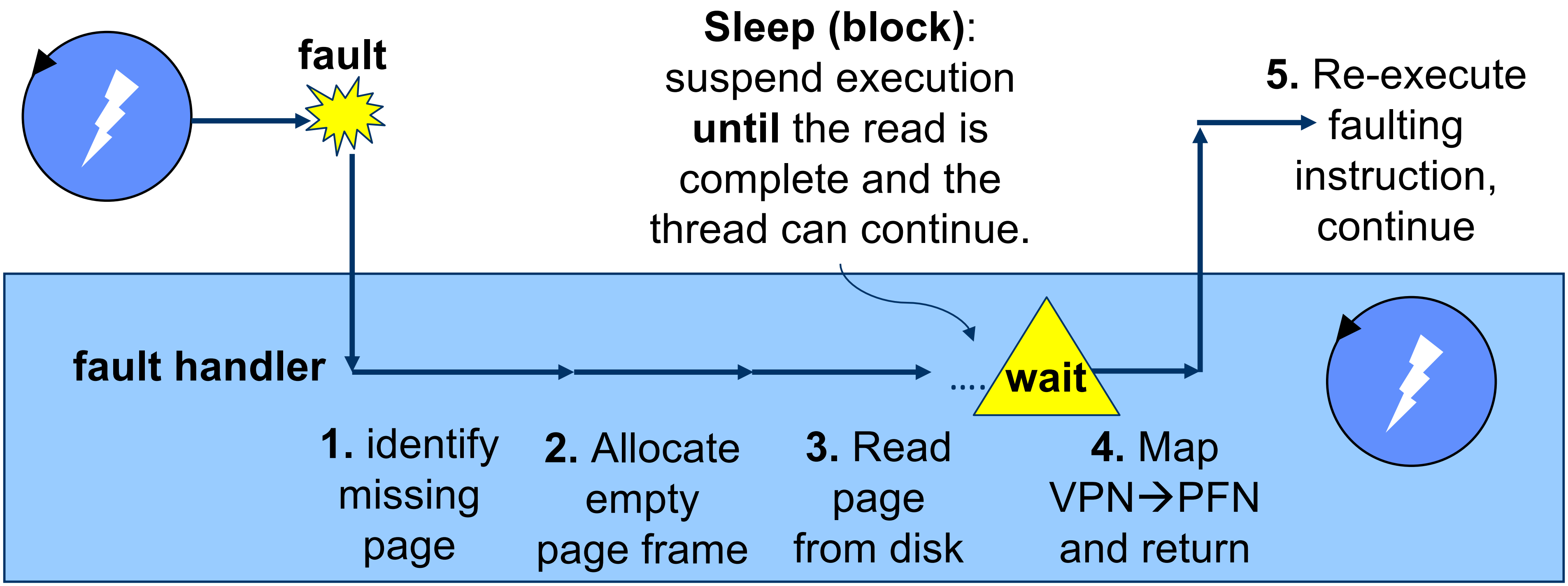
User mode to kernel mode: trap, fault, or interrupt.

Kernel mode to user mode: return from trap, fault, or interrupt.

Handling a page fault



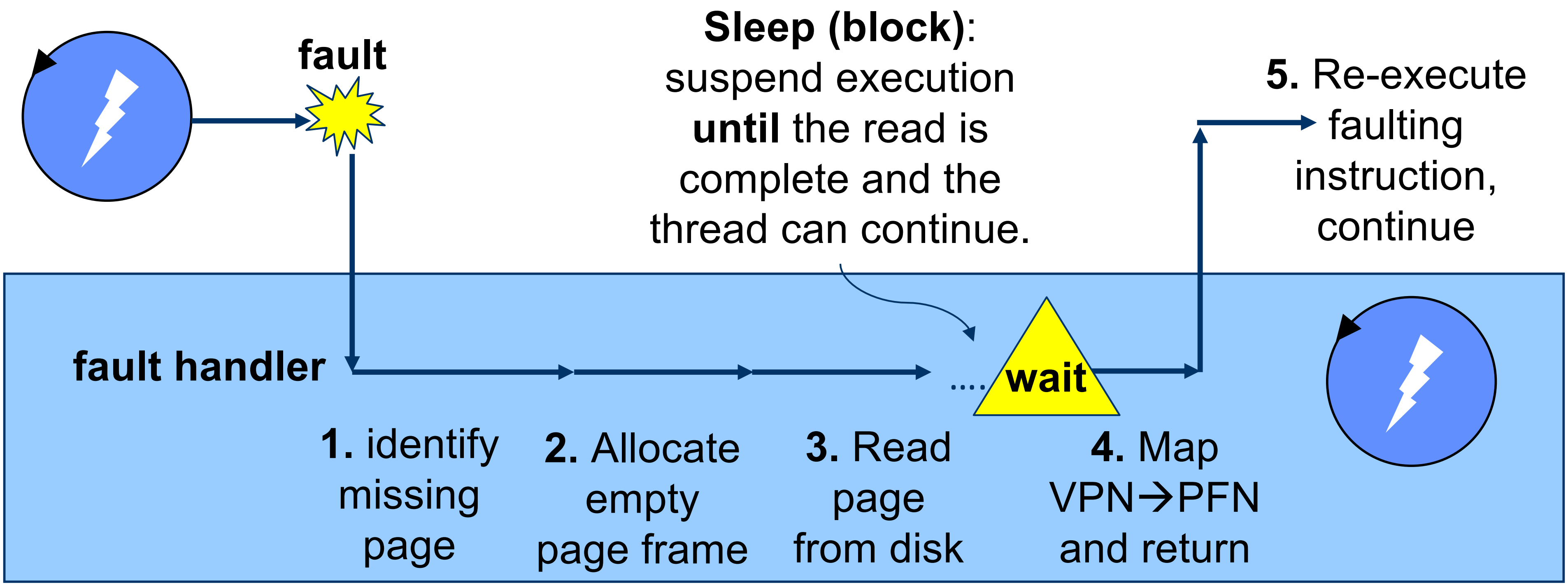
Blocking/sleeping in a page fault



When a thread **blocks**, it stops running, leaves the core for use by other threads, and **sleeps**: it wakes/resumes only after some specified event or condition occurs.

Threads block for page faults, and for many other reasons as well.
It happens all the time.

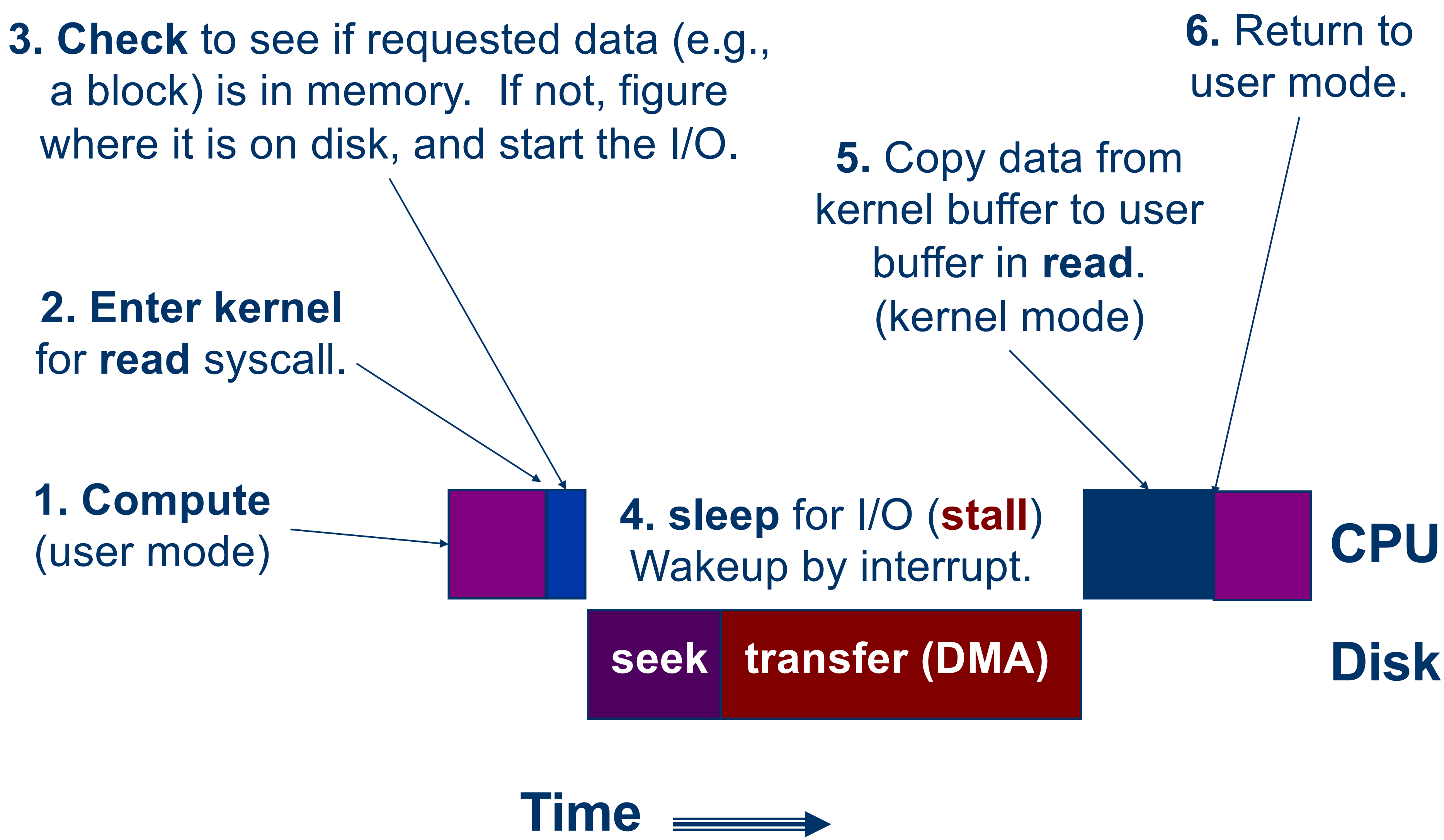
Blocking/sleeping in a page fault



When a thread **blocks**, it stops running, leaves the core for use by other threads, and **sleeps**: it wakes/resumes only after some specified event or condition occurs.

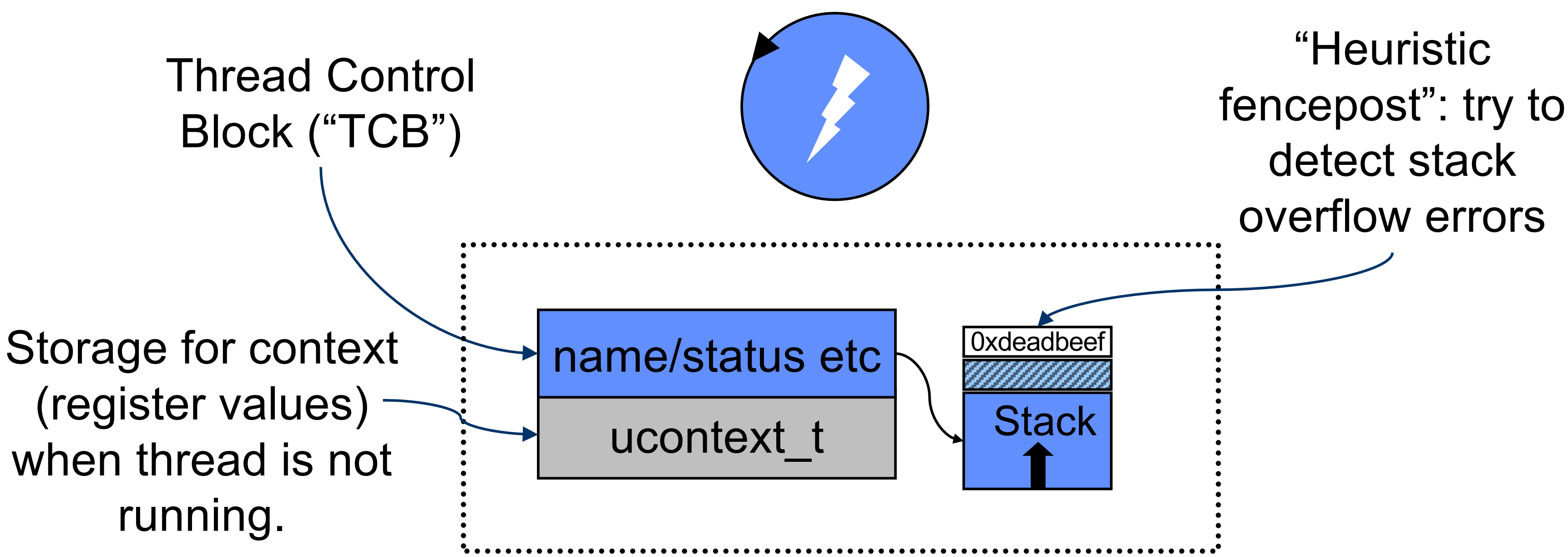
Threads block for page faults, and for many other reasons as well.
It happens all the time.

Anatomy of a read()



Portrait of a thread: the TCB

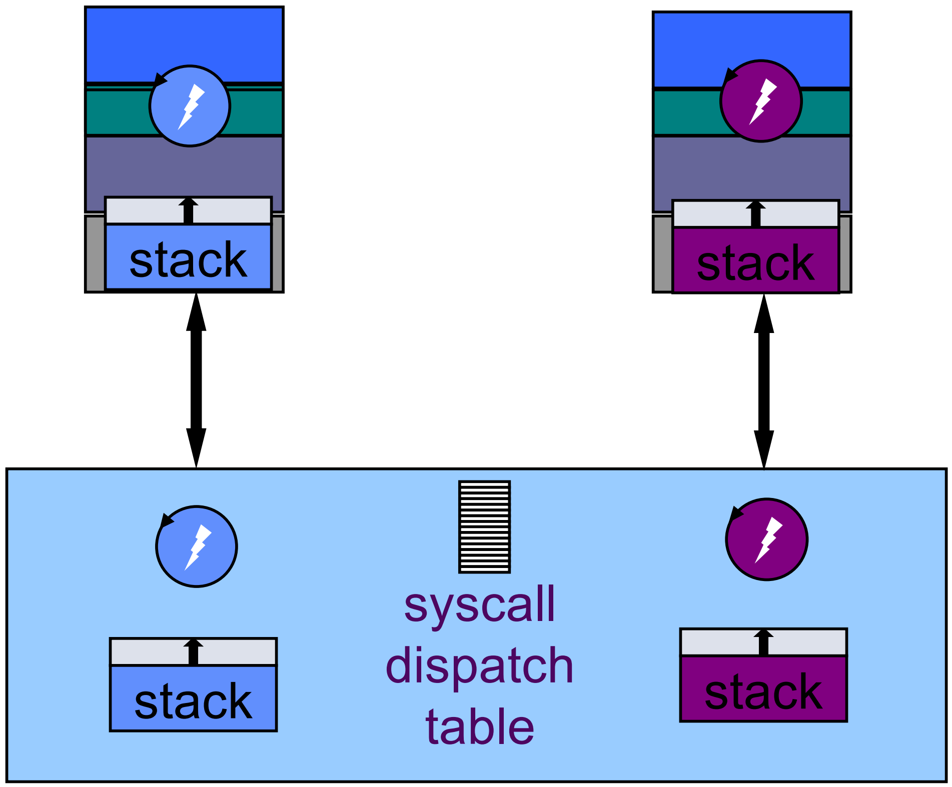
In an implementation, each thread is represented by a data structure. We call it a "**thread object**" or "**Thread Control Block**". It stores information about the thread, and may be linked into other system data structures.



Each thread also has a runtime stack for its own use. As a running thread calls procedures in the code, frames are pushed on its stack.

Kernel Stacks and Trap/Fault Handling

Threads execute user code on a **user stack** in user space (the process virtual address space).



System calls and faults run in kernel mode on a kernel stack for the current thread.

Each thread has a second **kernel stack** in **kernel space** (VM accessible only in kernel mode).

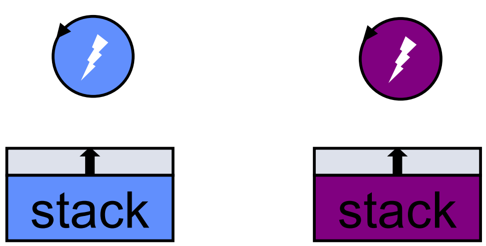
Kernel code running in P's process context has access to P's virtual memory.

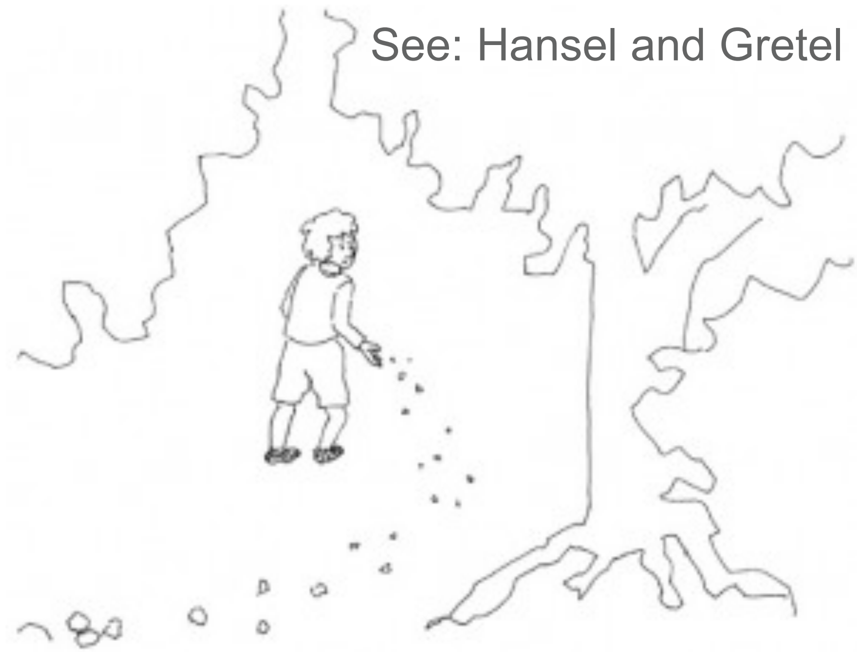
The syscall (trap) handler makes an indirect call through the system call dispatch table to the handler registered for the specific system call.

More analogies: threads and stacks

- Each thread chooses its own path. (By its program.)
- But they must leave some “bread crumbs” to find their way back on the return!
- Where does a thread leave its crumbs? **On its stack!**
  - Call frames with local variables
  - Return addresses

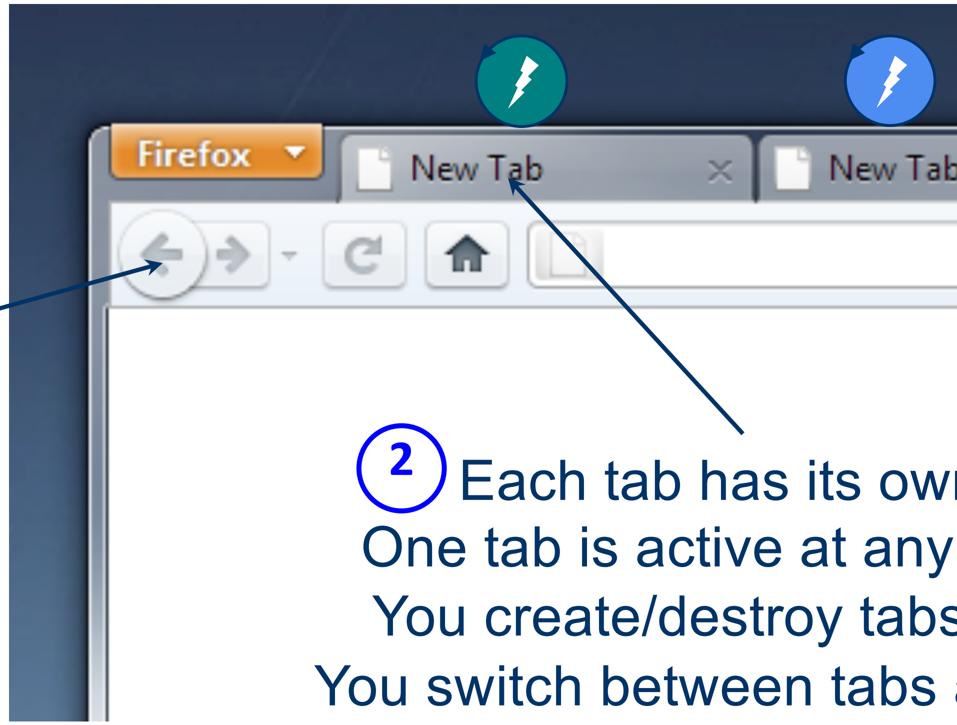
This means that **each thread must have its own stack** in memory, so that their crumbs aren’t all mixed together.





See: Hansel and Gretel

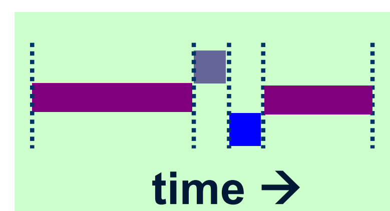
More analogies: context/switching



1
Page links and back button navigate a "stack" of pages in each tab.

2 Each tab has its own stack.
One tab is active at any given time.
You create/destroy tabs as needed.
You switch between tabs at your whim.

3 Similarly, each thread has a separate stack.
The OS switches between threads at its whim.
One thread is active per CPU core at any given time.

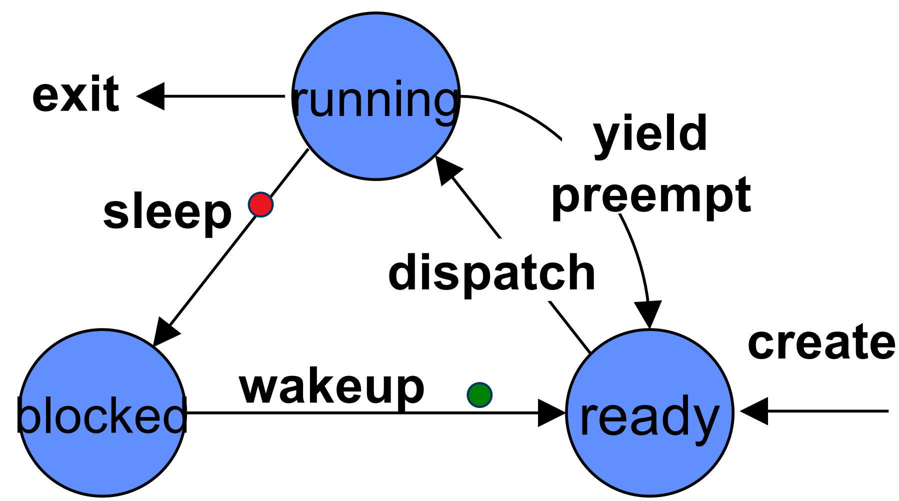


Thread states and queues

- At most on thread T is **running** (per core).
  - Current[core] points at T's TCB.
- Any other thread P is **blocked** or **ready**.
- Link P's TCB onto a **thread queue**.

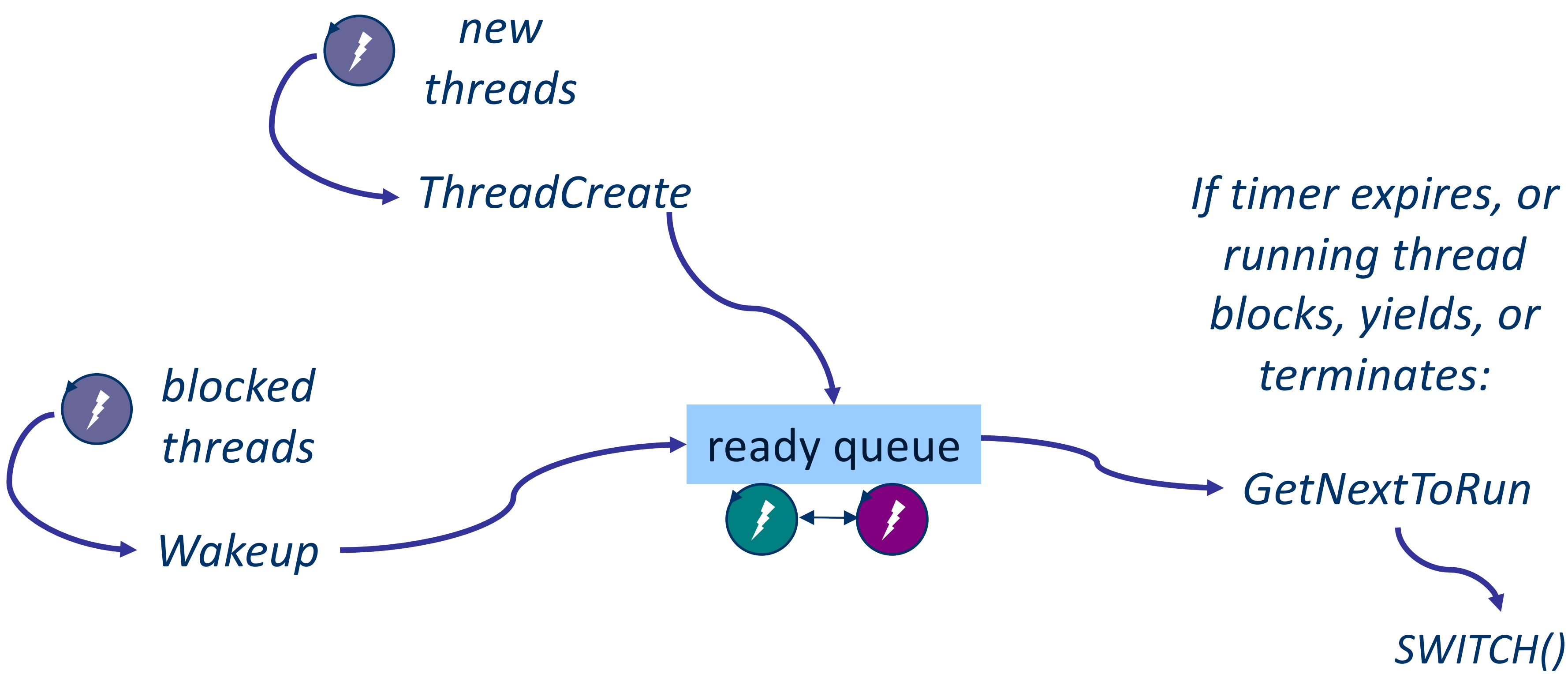
Internal routines manage thread state changes.
E.g.: **yield**, **sleep**, **wakeup**.

These routines slam thread TCBs on and off various internal thread queues.



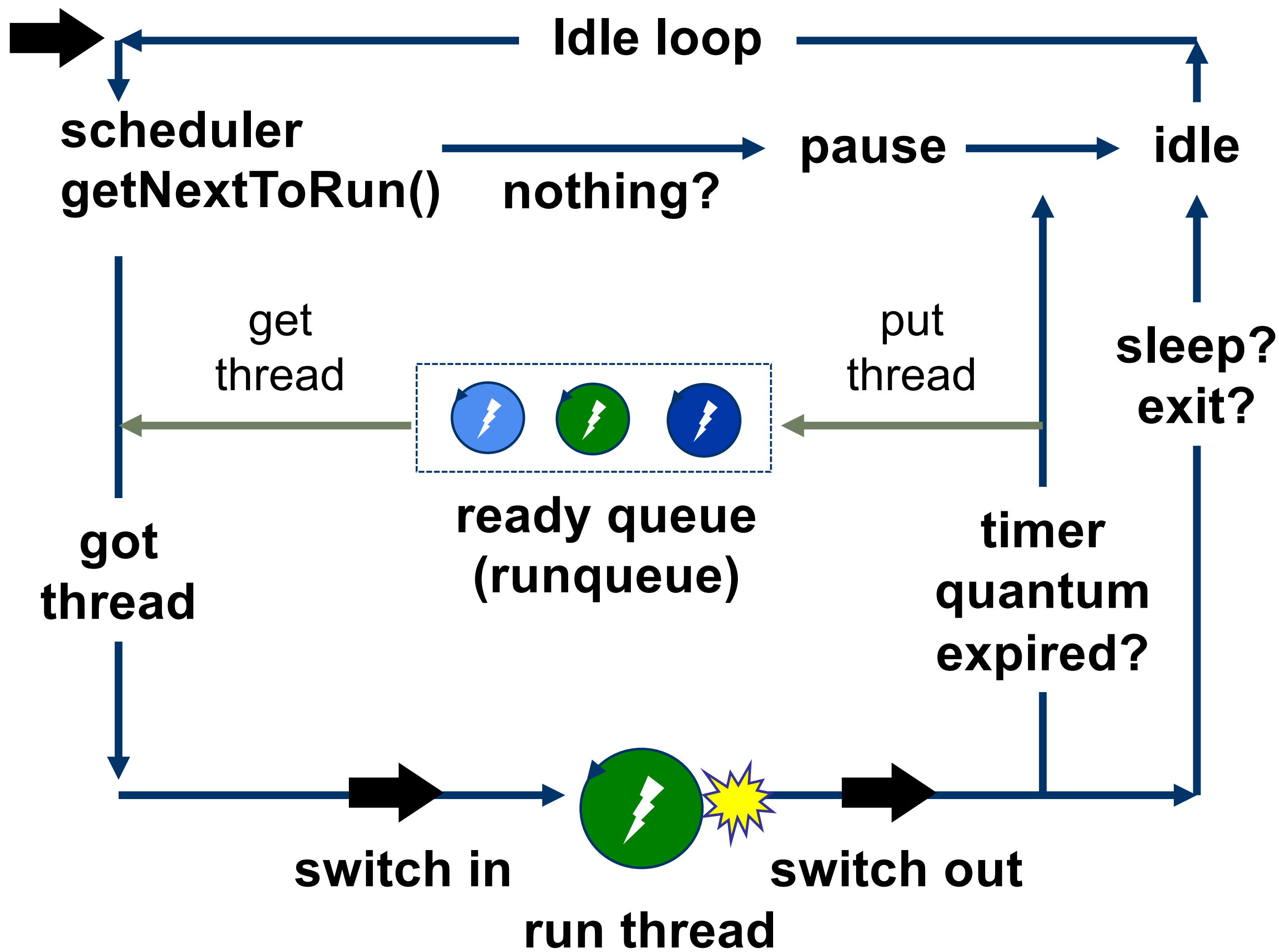
Ready→ready queue
Blocked→waiter queue for event/object

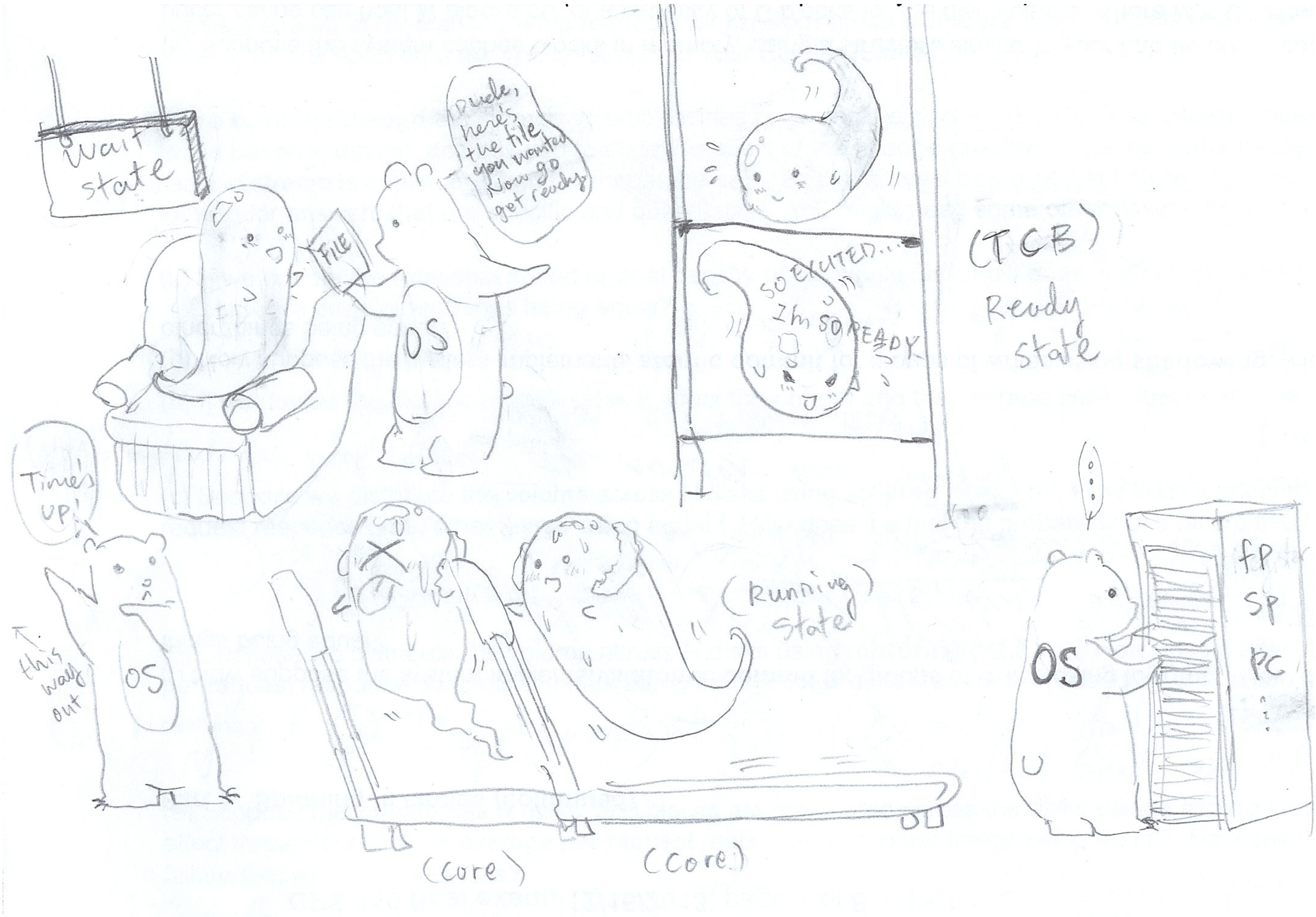
The ready queue



For p1t, the ready pool is a simple FIFO queue: the **ready list** or **ready queue** (**runqueue**). Scalable multi-core systems use multiple runqueues to reduce locking contention among cores. It is typical to implement a ready pool as an array of runqueues for different priority levels.

What cores do





A former student named Grace Chen drew this artwork freeform for extra credit on a final exam.