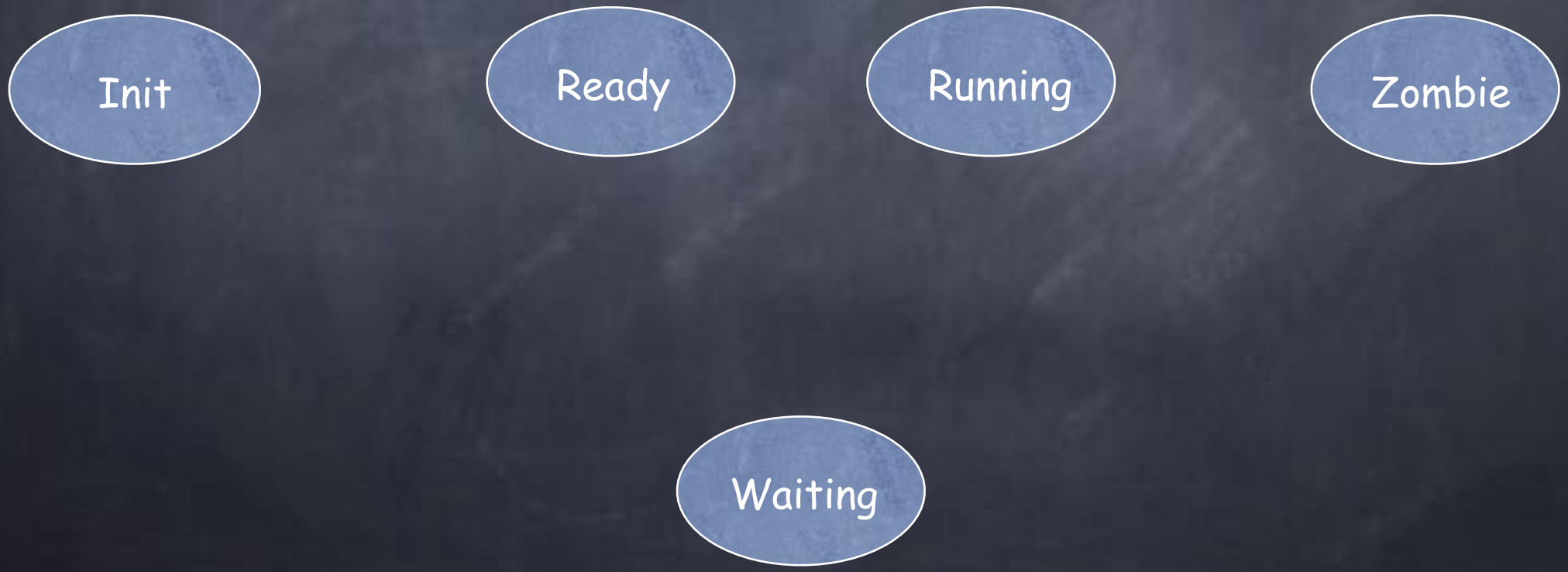
A simple API

Syscall	Description
void thread_create (thread, func, arg)	Creates a new thread in thread, which will execute function func with arguments arg.
void thread_yield()	Calling thread gives up processor. Scheduler can resume running this thread at any time
int thread_join (thread)	Wait for thread to finish, then return the value thread passed to thread_exit.
void thread_exit (ret)	Finish caller. Store return value on TCB. If another thread is waiting on thread_join, resume it.

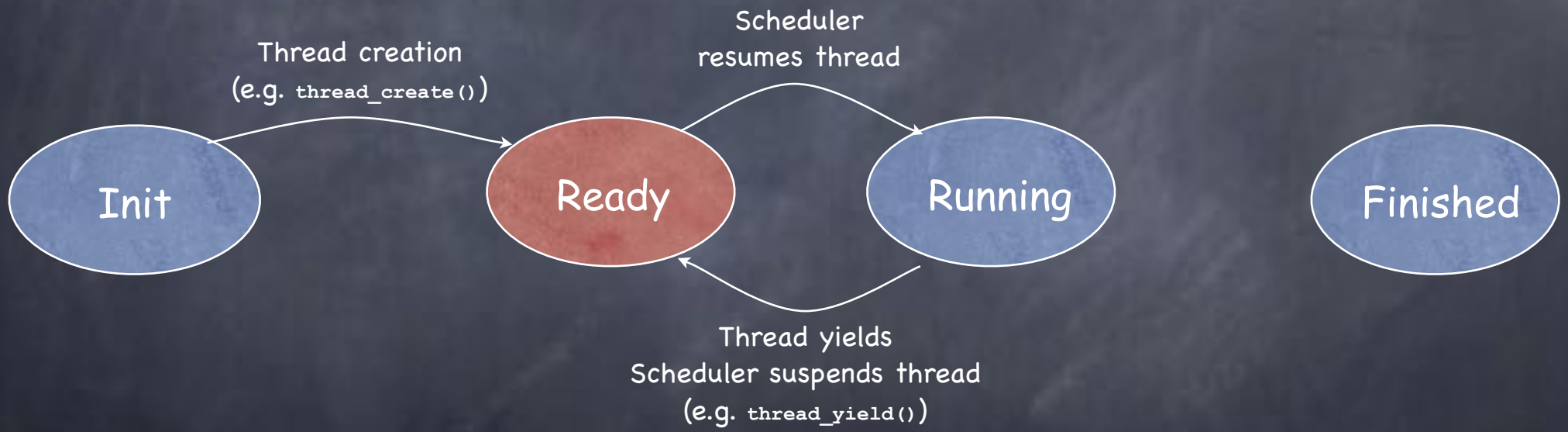
Process Life Cycle

- Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states



Threads Life Cycle

- Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states



TCB: Ready list
Registers: in TCB (or pushed on thread's stack) SP in TCB

Waiting

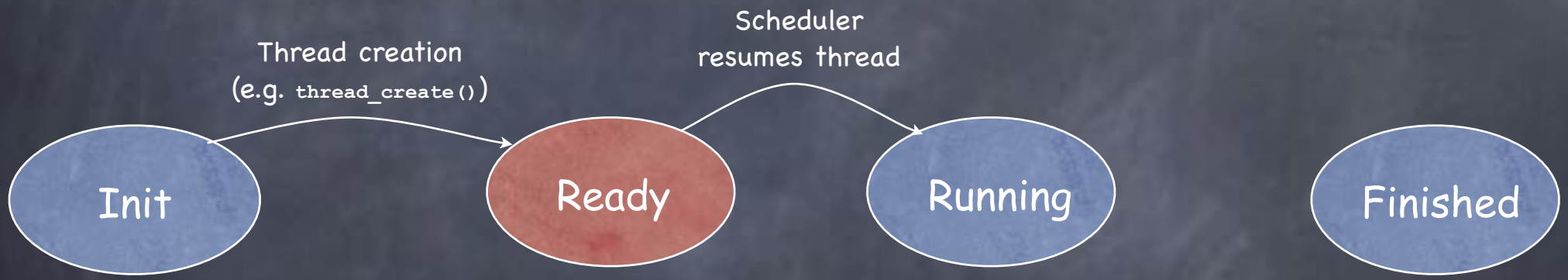
Threads Life Cycle

- Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states

TCB: being created
Registers: in TCB

Threads Life Cycle

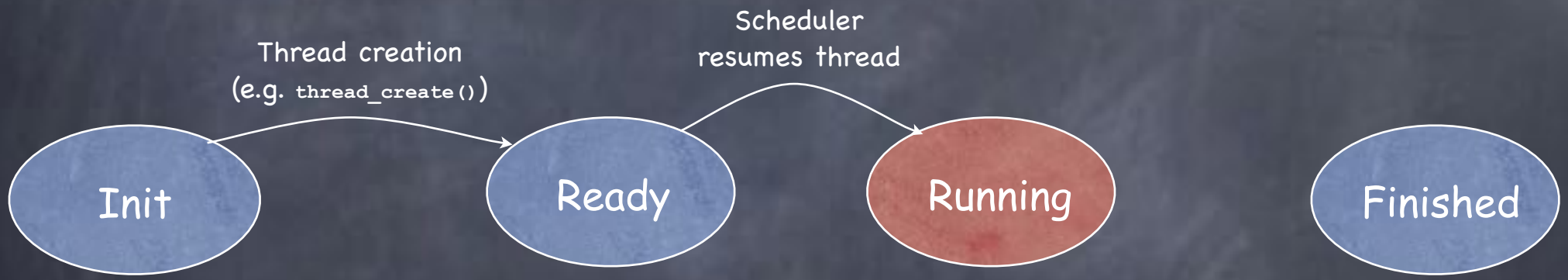
- Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states



TCB: Ready list
Registers: in TCB (or pushed on thread's stack). SP in TCB

Threads Life Cycle

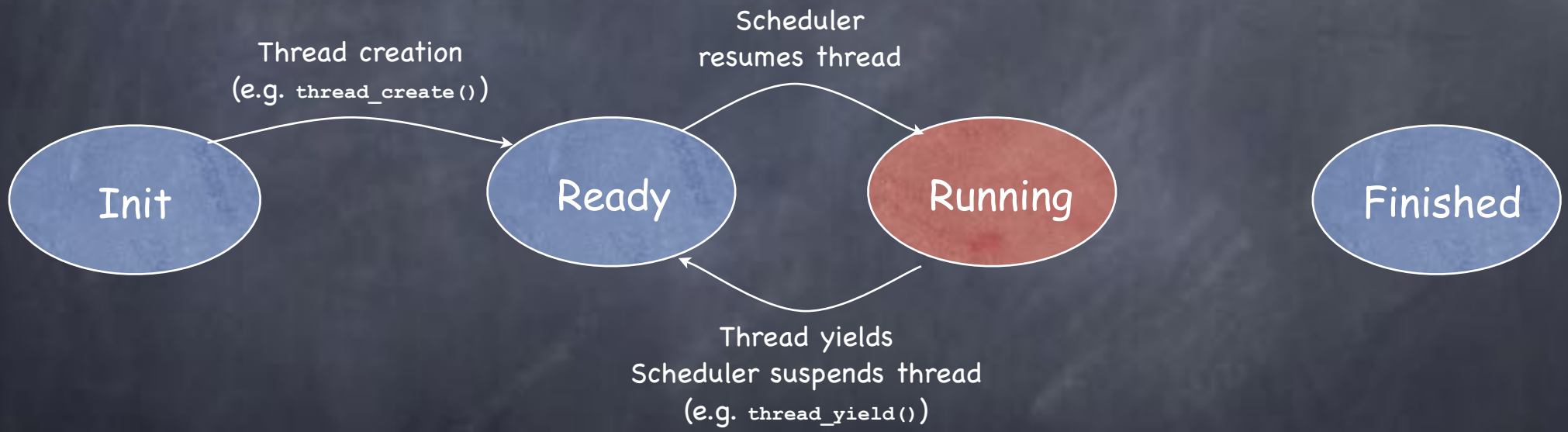
- Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states



TCB: On no list
Registers: Restored from TCB or thread's stack into CPU

Threads Life Cycle

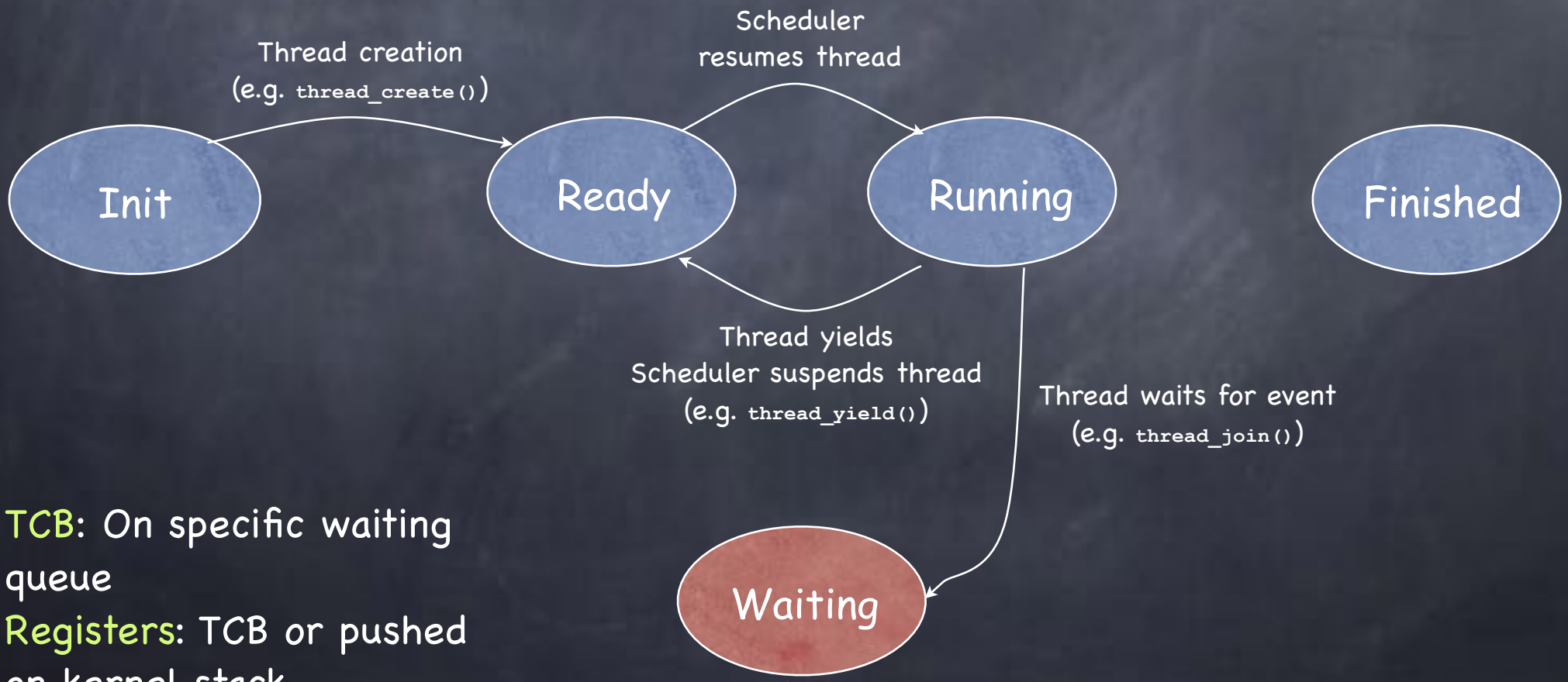
- Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states



TCB: Running list
Registers: Restored from TCB or thread's stack into CPU

Threads Life Cycle

- Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states

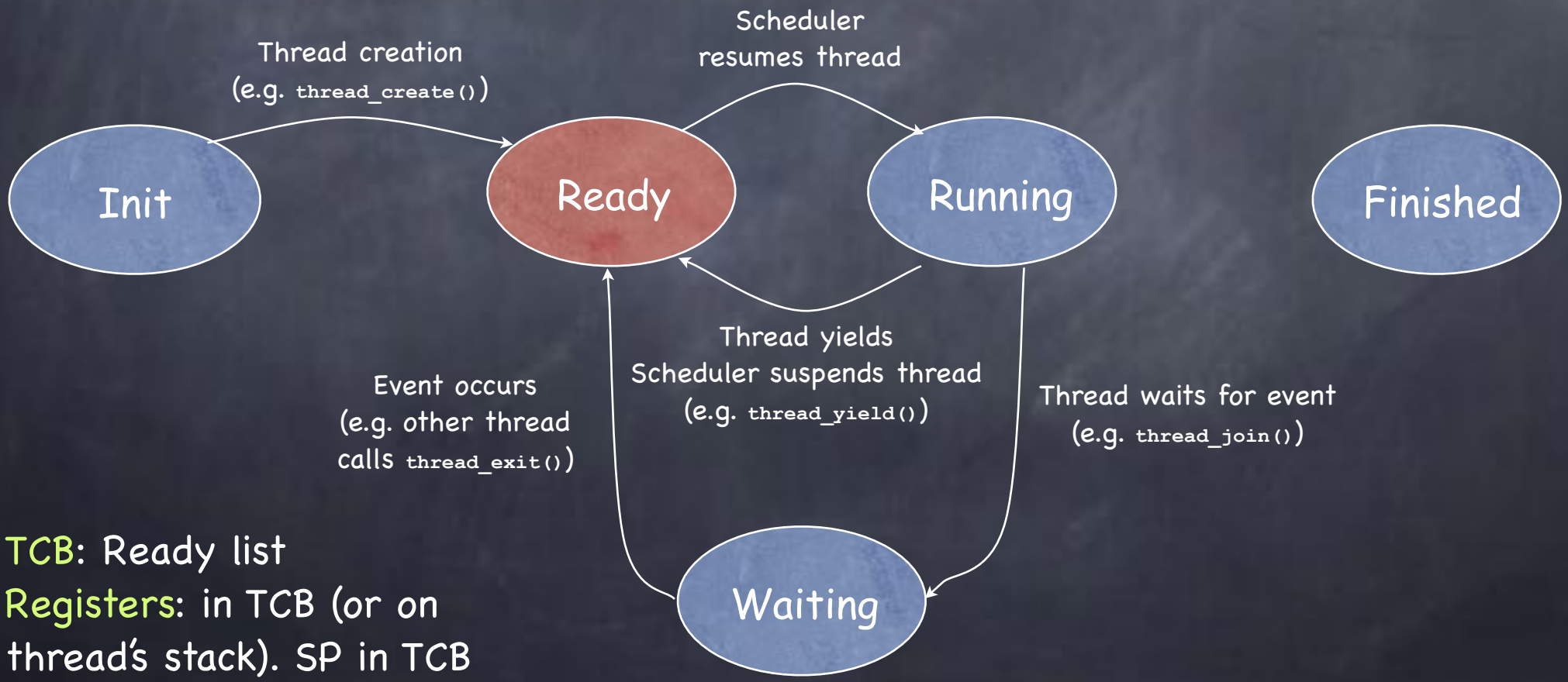


TCB: On specific waiting queue

Registers: TCB or pushed on kernel stack

Threads Life Cycle

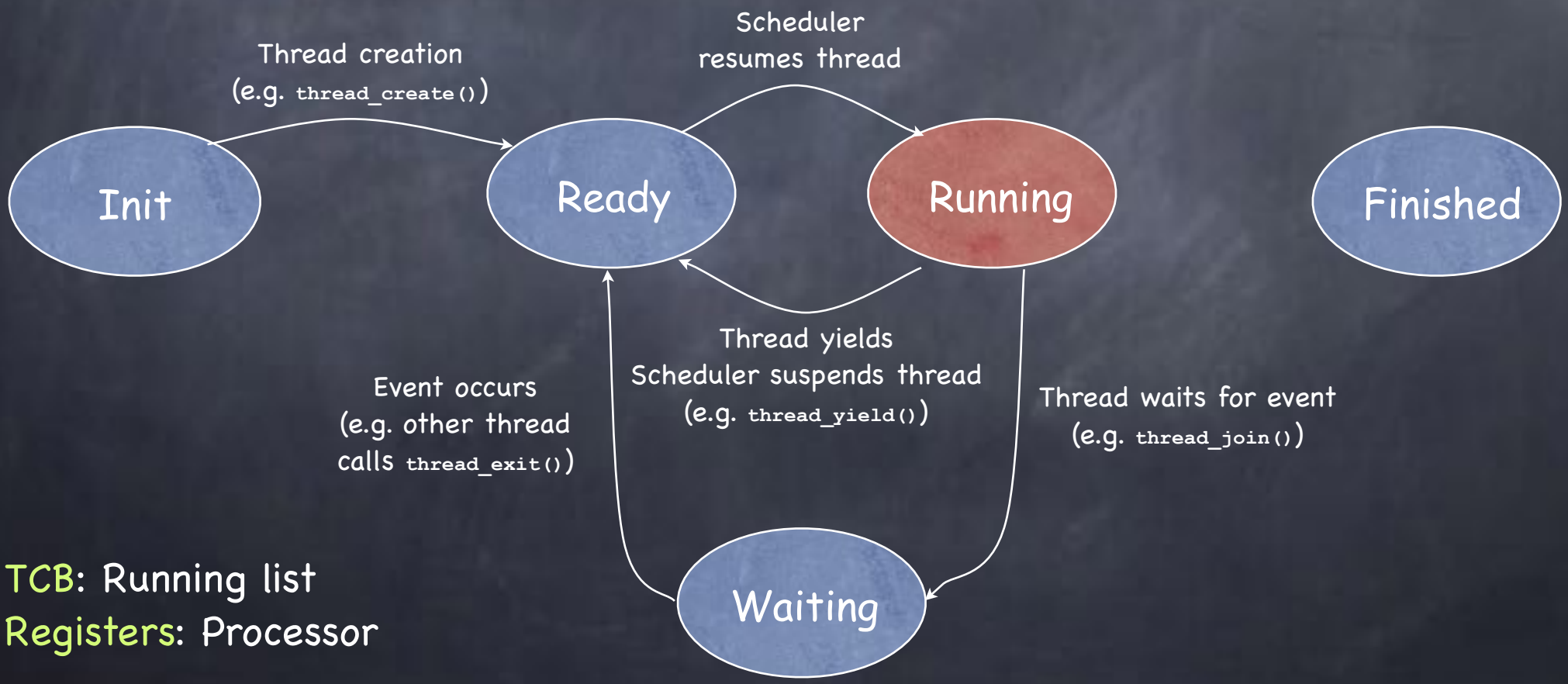
- Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states



TCB: Ready list
Registers: in TCB (or on thread's stack). SP in TCB

Threads Life Cycle

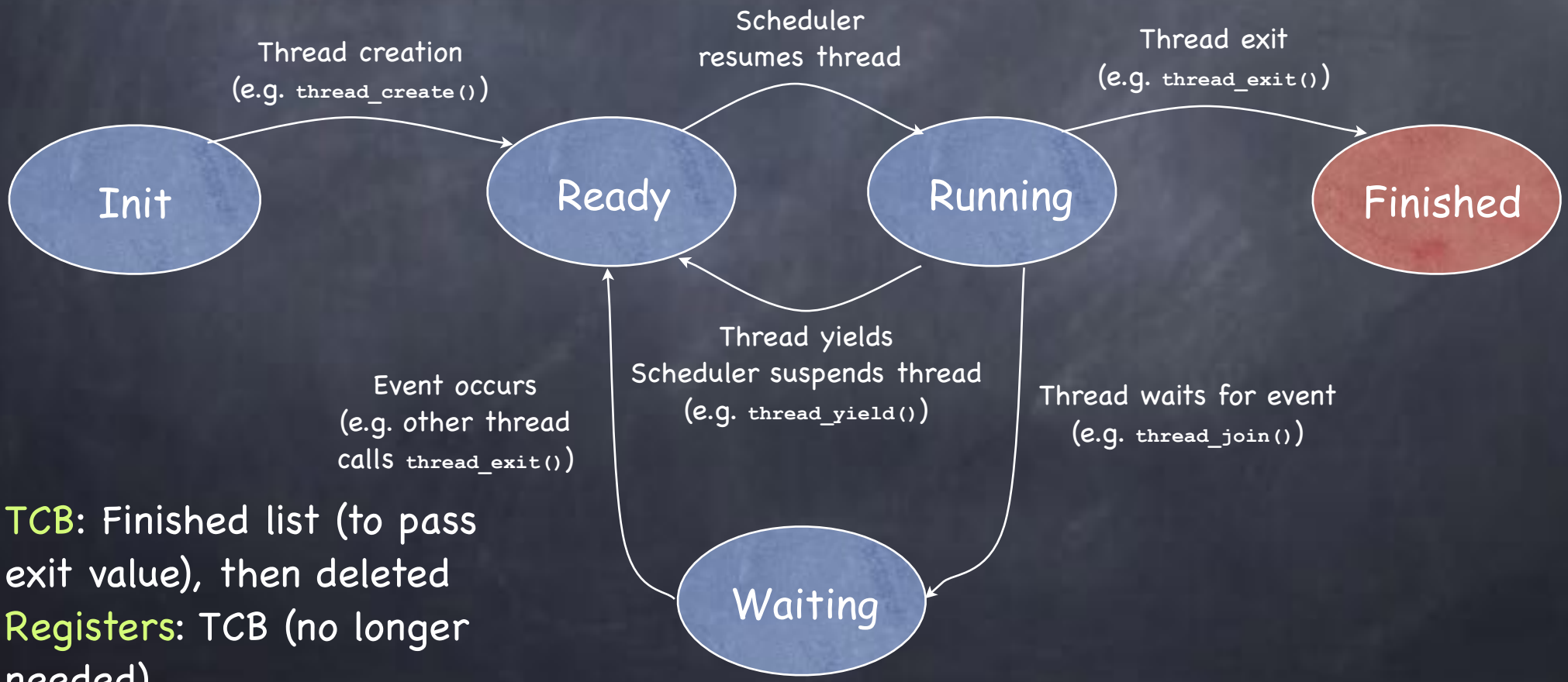
- Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states



TCB: Running list
Registers: Processor

Threads Life Cycle

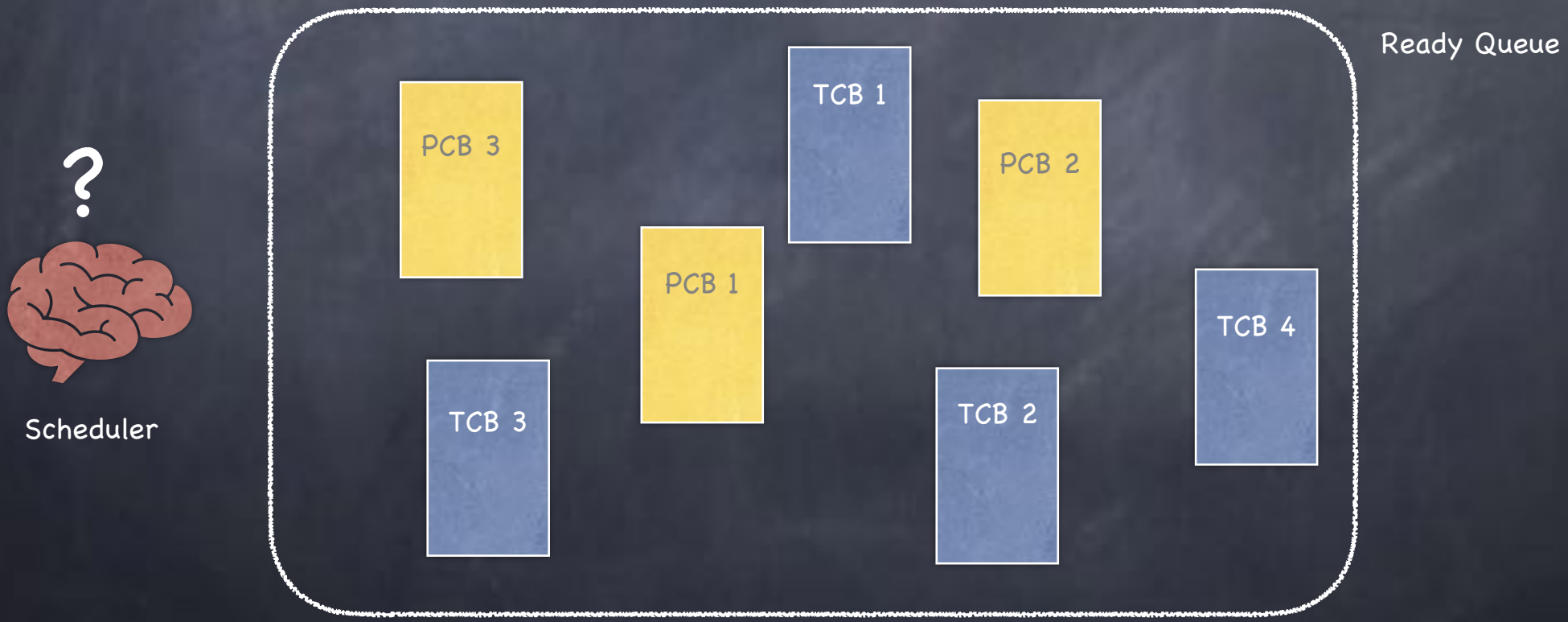
- Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states



TCB: Finished list (to pass exit value), then deleted
Registers: TCB (no longer needed)

How a Ready Queue may look like

It is an actually not a queue, but a set...



One Abstraction, Two Implementations

- User Threads
  - implemented entirely in user space; invisible to the kernel
  - one PCB for the process
  - each thread has its own Thread Control Block (TCB) [implemented in the host process' heap]
- Kernel Threads
  - visible (and schedulable) by kernel
  - each thread has own TCB and stack in the kernel (in addition to a stack in user space, if appropriate)

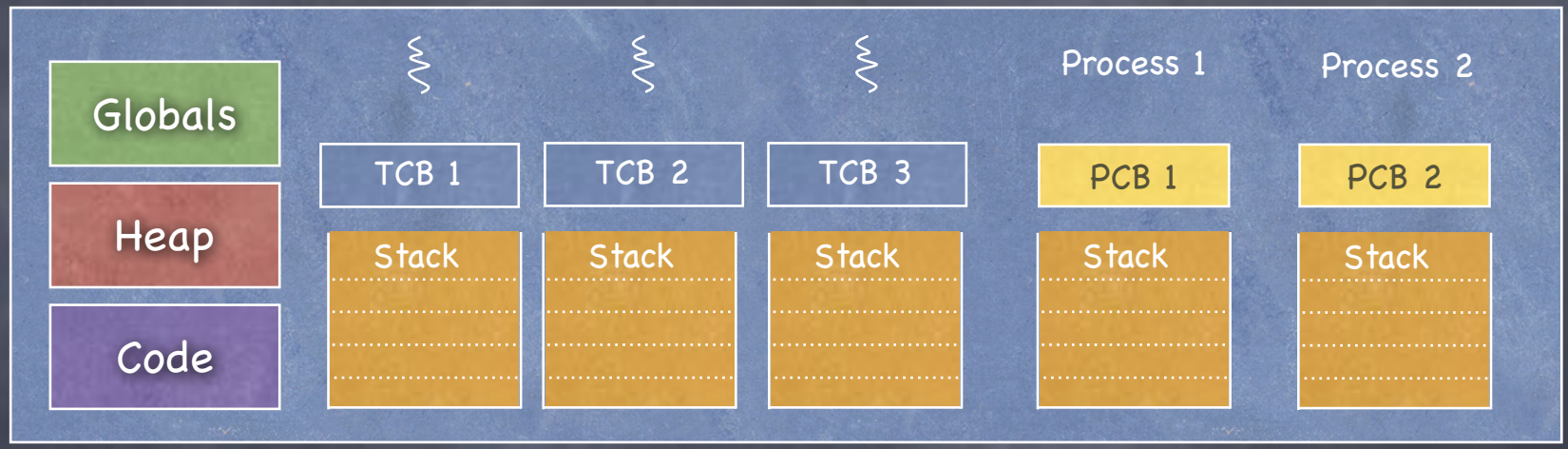
Binding user and kernel threads

- To associate a kernel thread to a user thread
  - invoke appropriate system call
  - kernel allocates a TCB & interrupt stack...
  - and sets it up so that, if scheduled, the thread will start executing on the user-level stack at the at the beginning of the procedure requested in thread_create()

Single-threaded processes & kernel threads

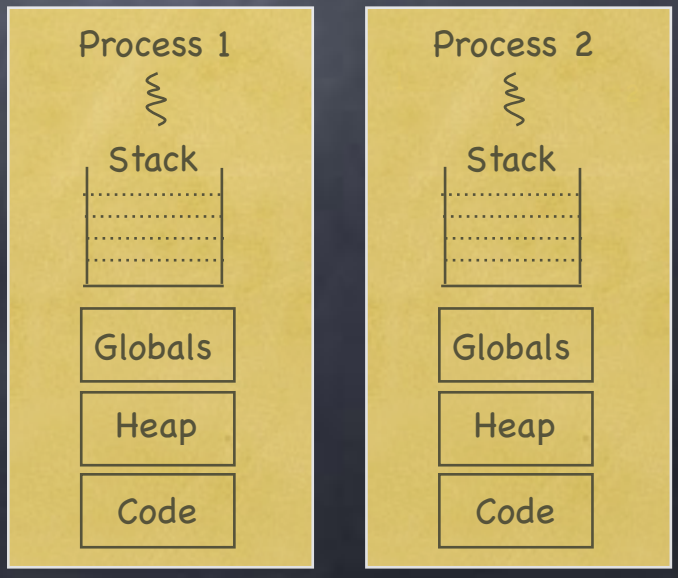
Kernel

Each kernel thread has its own TCB and its own stack. Each process has a PCB and a kernel interrupt stack



User-level processes

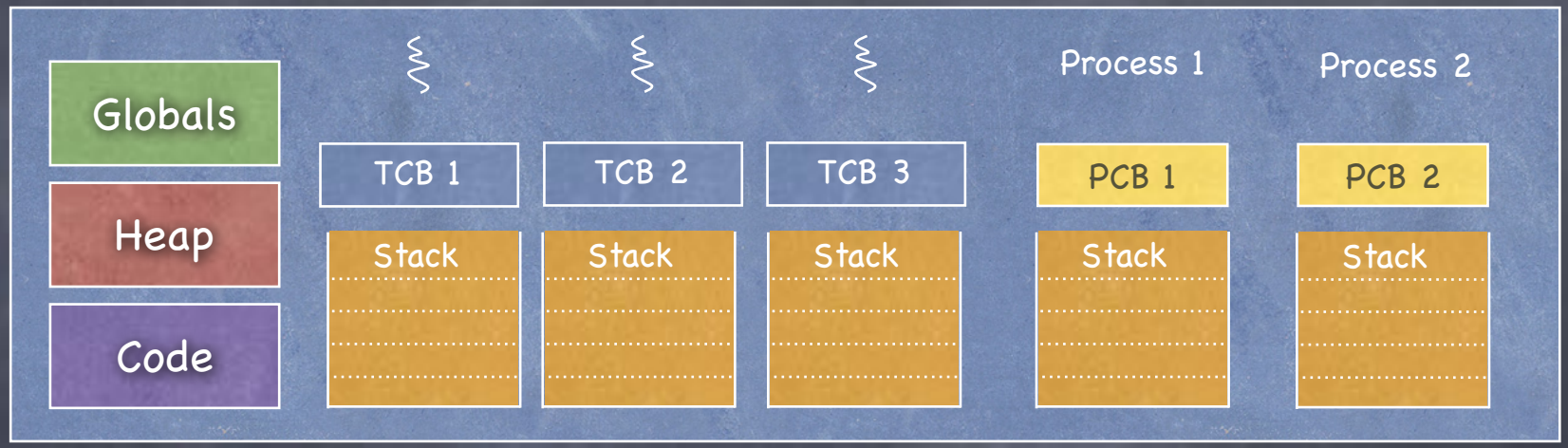
Each user process has a stack at user-level for executing user code and a kernel interrupt stack for executing interrupts and system calls.



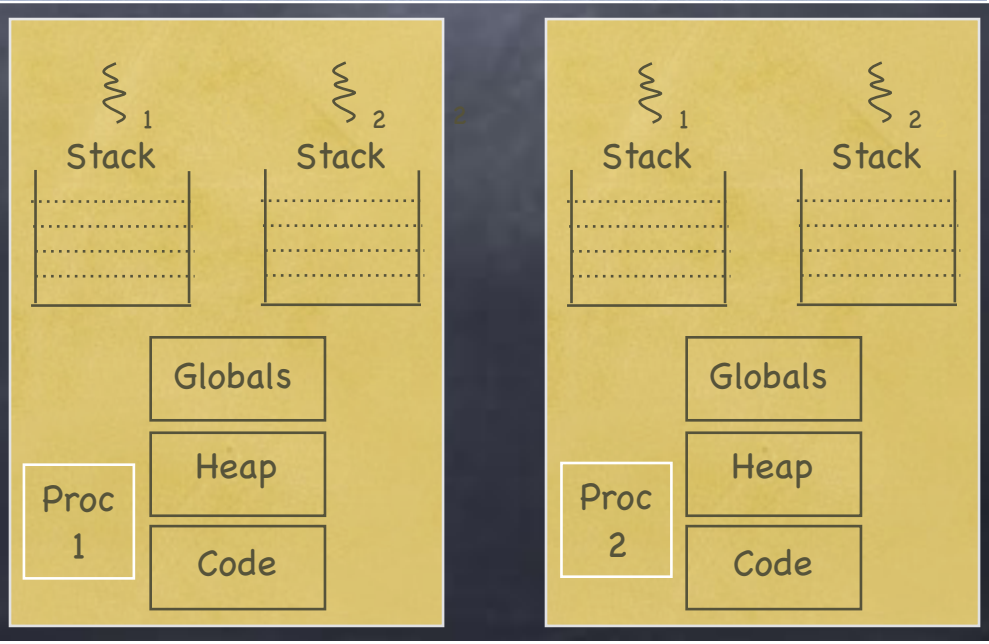
Multi-threaded processes: user-level threads

Kernel

Each kernel thread has its own TCB and its own stack. Each process has a PCB and a kernel interrupt stack

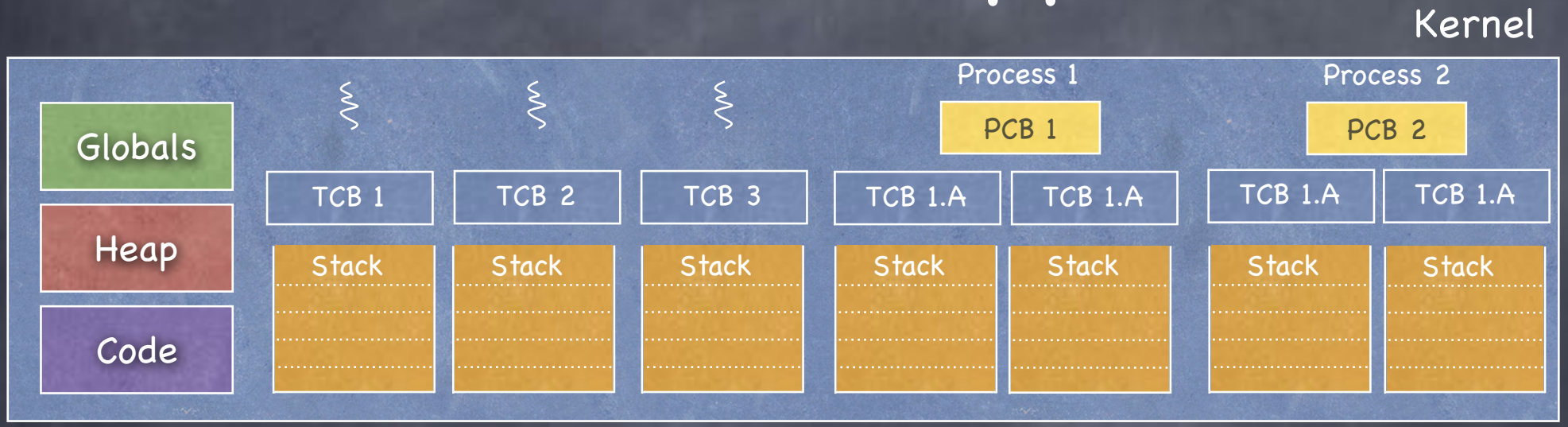


User-level processes

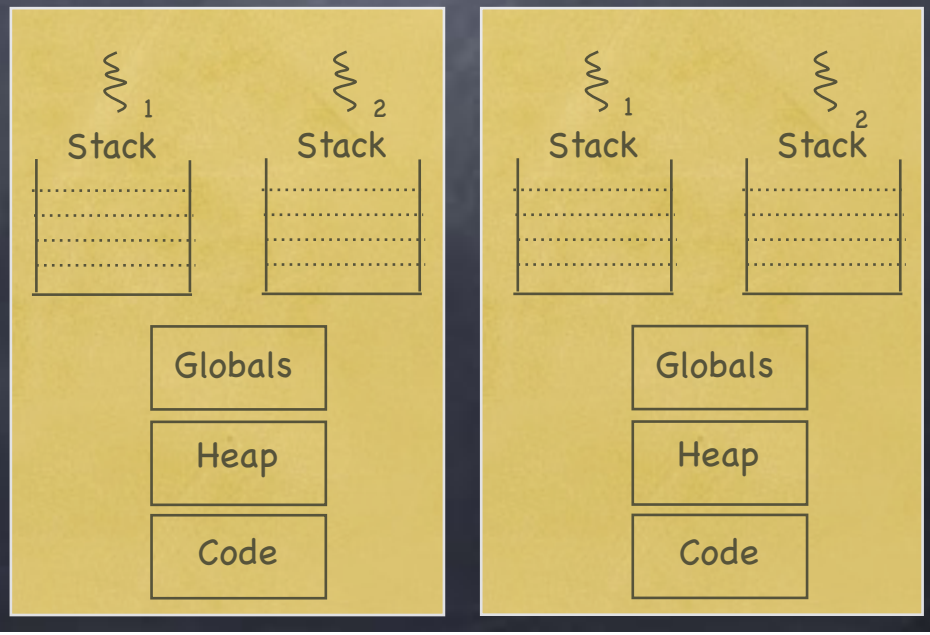


Each process has multiple user-level threads. Each thread has its own stack in user space, but these user-level threads are invisible to the kernel, which can only schedule what appears to it as a single-threaded process

Multi-threaded processes with kernel support



User-level processes



Each user-level thread has a user-level stack and a corresponding interrupt stack in the kernel for executing interrupts and system calls.
Each user-level thread can then be independently scheduled by the kernel

Preempt or Not Preempt?

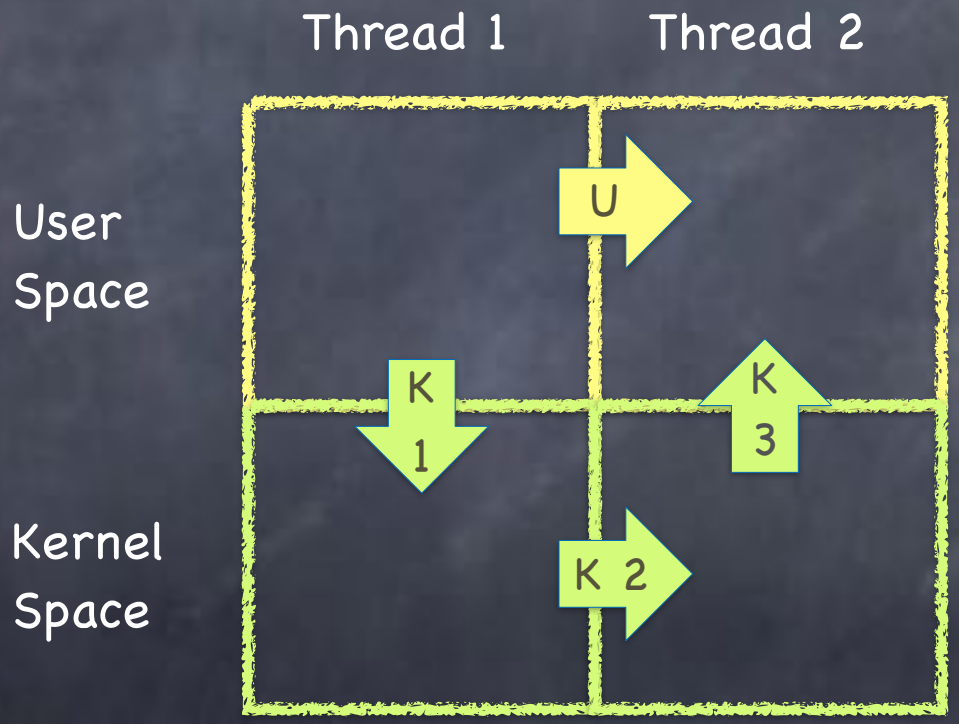
- Preemptive
  - yield automatically upon clock interrupts
  - true of most modern threading systems
- Non-preemptive
  - explicitly yield to pass control to other threads

- Most modern threading systems are preemptive
  - but not CS4411 P1 project

Kernel- vs. Only User-level Threads

	Kernel-level Threads	Only User-Level Threads
Ease of implementation	Easy to implement: just like process, but with shared address space	Requires implementing user-level schedule and context switches
Handling system calls	Thread can run blocking systems call concurrently	Blocking system call blocks all threads: needs OS support for non-blocking system calls (scheduler activations)
Cost of context switch	Thread switch requires three context switches	Thread switch efficiently implemented in user space

Kernel- vs. User-level Thread Switching




The shell
https://www.youtube.com/watch?v=ycm5IIZrpKs

What is a shell?

An interpreter

- Runs programs on behalf of the user
- Allows programmer to create/manage set of programs
  - sh — Original Unix shell (Bourne, 1977)
  - csh — BSD Unix C shell (tcsh enhances it)
  - bash — "Bourne again" shell
- Every command typed in the shell starts a child process of the shell
- Runs at user-level. Uses syscalls: fork, exec, etc.

The Unix shell (simplified)

```
while(! EOF)
read input
handle regular expressions
int pid = fork()    // create child
if (pid == 0) { // child here
  exec("program", argc, argv0,...);
}
else {  // parent here
...
}
```

Some important commands

- echo [args] # prints args
- pwd # prints working directory
- ls # lists current directory
- cd [dir] # change current directory
- ps # lists your running processes

Commands can be modified with flags

- ls -l # long list of current directory
- ps -a # lists all running processes

Foreground vs Background

- The shell is either
  - reading from standard input or
  - waiting for a process to finish
    - this is the foreground process
    - other processes are background processes
- To start a background process, add &
  - (sleep 5; echo hello) &
  - x & y # runs x in background and y in foreground

Pipes

- x | y
  - runs both x and y in foreground
  - output of x is input to y
  - finishes when both x and y are finished

echo Lorenzo | tr r b | tr n r | tr z t | tr L R

CPU Scheduling

(Chapters 7-11)

Mechanism and Policy

- Mechanism
  - enables a functionality
- Policy
  - determines how that funtionality should be used

Mechanisms should not determine policies!

The Problem

- You are the cook at the State Street Diner
  - Customers enter and place orders 24 hours a day
  - Dishes take varying amounts of time to prepare
- What are your goals?
  - Minimize average turnaround time?
  - Minimize maximum turnaround time?
- Which strategy achieves your goal?

Context matters!

- What if instead you are:
  - the owner of an expensive container ship, and have cargo across the world
  - the head nurse managing the waiting room of an emergency room
  - a student who has to do homework in various classes, hang out with other students, eat, and (occasionally) sleep

Schedulers in the OS

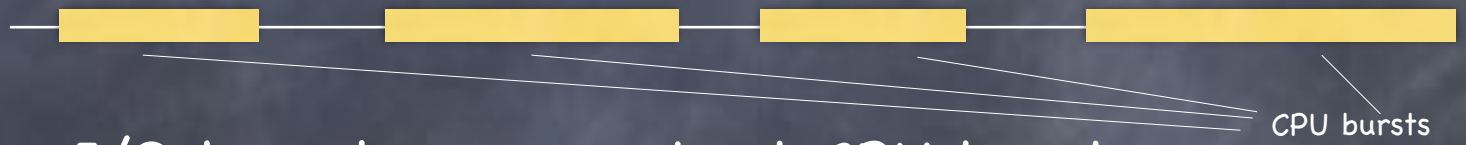
- CPU scheduler selects next process to run from the ready queue
- Disk scheduler selects next read/write operation
- Network scheduler selects next packet to send or process
- Page Replacement scheduler selects page to evict

Scheduling processes

- OS keeps PCBs on different queues
  - Ready processes are on ready queue - OS chooses one to dispatch
  - Processes waiting for I/O are on appropriate device queue
  - Processes waiting on a condition are on an appropriate condition variable queue
- OS regulates PCB migration during life cycle of corresponding process

Why scheduling is challenging

- Processes are not created equal!
  - CPU-bound process: long CPU bursts
    - mp3 encoding, compilation, scientific applications



  - I/O-bound process: short CPU bursts
    - index a file system, browse small web pages

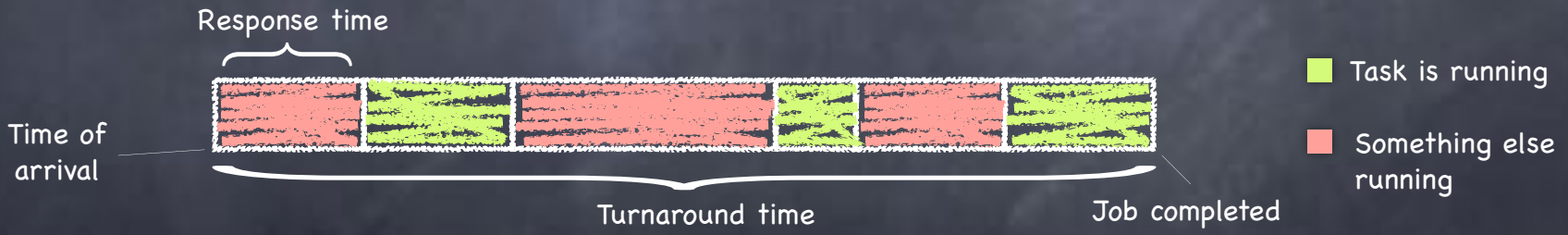
- Problem
  - don't know jobs type before running it
  - jobs behavior can change over time

Job Characteristics

- Job: A task that needs a period of CPU time
  - A user request: e.g., mouse click, web request, shell command...
- Defined by:
  - Arrival time
    - When the job was first submitted
  - Execution time
    - Time needed to run the task in isolation
  - Deadline
    - By when the task must have completed (e.g. for videos, car brakes...)

Metrics

- Response time
  - How long between job's arrival and first time job runs?
- Total waiting time
  - How much time on ready queue but not running?
    - sum of "red" intervals below
- Execution time: sum of "green" intervals



- Turnaround time: "red" + "green"
  - Time between a job's arrival and its completion
- Throughput: jobs completed/unit of time

Other Concerns

- Fairness: Who get the resources?
  - Equitable division of resources
- Starvation: How bad can it get?
  - Lack of progress by some job
- Overhead: How much useless work?
  - Time wasted switching between jobs
- Predictability: How consistent?
  - Low variance in response time for repeated requests

The Perfect Scheduler

- Minimizes response time and turnaround time for each job
- Maximizes overall throughput
- Maximizes resource utilization ("work conserving")
- Meets all deadlines
- Is fair: everyone makes progress, no one starves
- Is Envy-Free: no job wants to switch its schedule with another
- Has zero overhead

Alas, no such scheduler exists...

When does the Scheduler Run?

- Non-preemptive
  - job runs until it voluntarily yields the CPU
    - process blocks on an event (e.g., I/O or P(sem))
    - process explicitly yields
    - process terminates
- Preemptive
  - all of the above, plus timer and other interrupts
    - when processes can't be trusted
  - incurs some context switching overhead