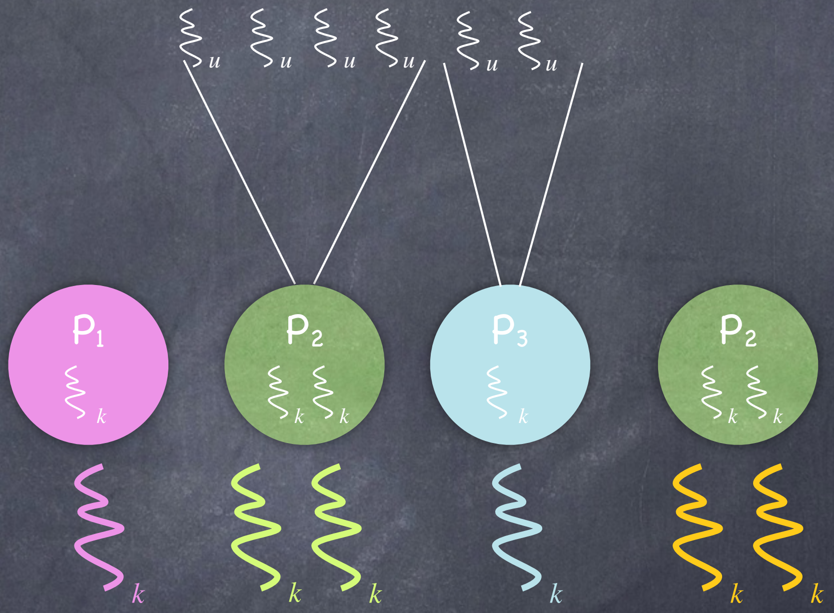
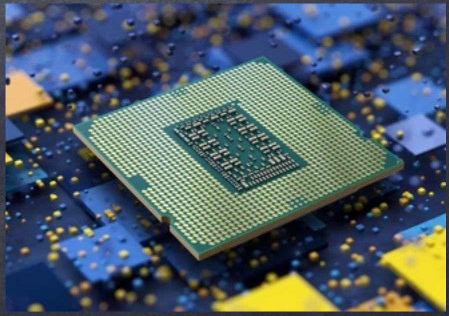
Where should threads be implemented?

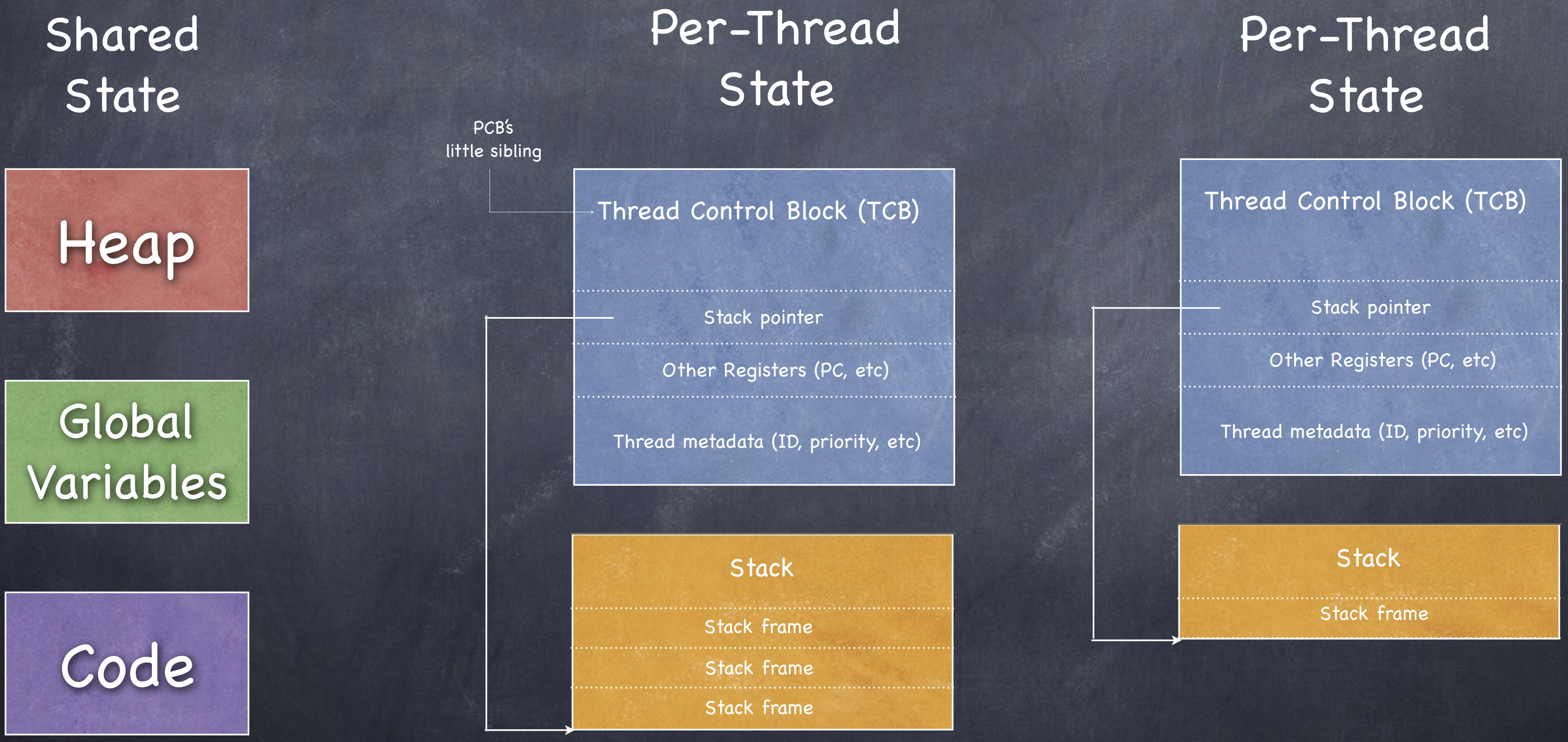
- In both!
    - Kernel multiplexes each physical CPU across multiple threads
    - Kernel can assign one or more threads to a process
    - Scheduler schedules threads
    - User level library multiplexes the process' single kernel thread across multiple user level threads





Hardware

How does a multi-threaded process look like?



Shared State

Heap

Global Variables

Code

Per-Thread State

PCB's little sibling

Thread Control Block (TCB)

Stack pointer

Other Registers (PC, etc)

Thread metadata (ID, priority, etc)

Stack

Stack frame

Stack frame

Stack frame

Per-Thread State

Thread Control Block (TCB)

Stack pointer

Other Registers (PC, etc)

Thread metadata (ID, priority, etc)

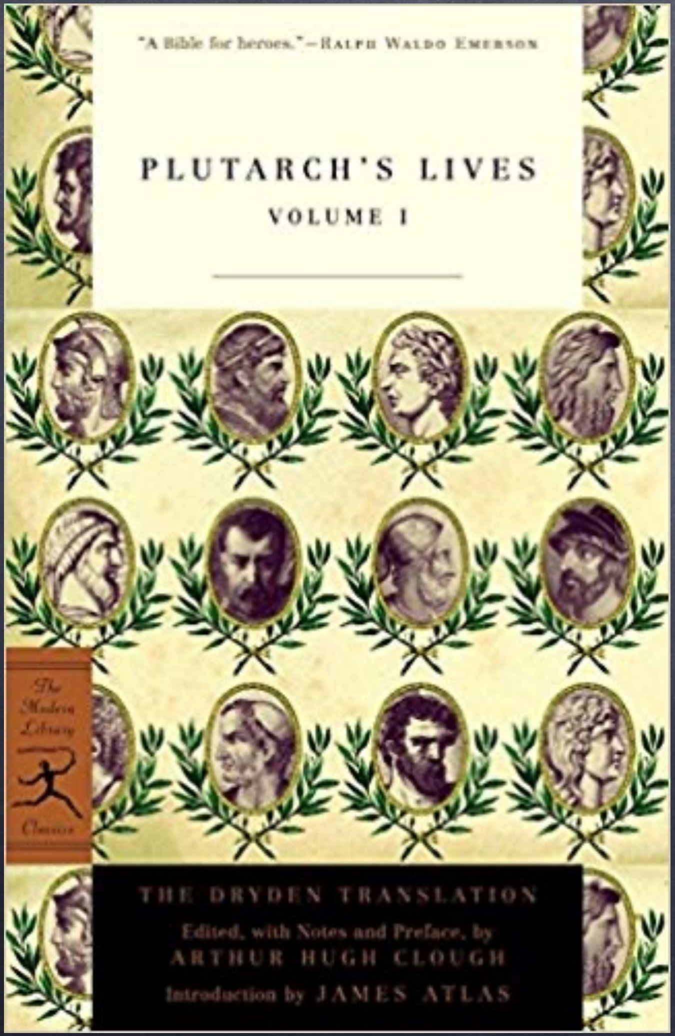
Stack

Stack frame

Note: No protection enforced at the thread level!

Processes vs. Threads:
Parallel lives

More
books!

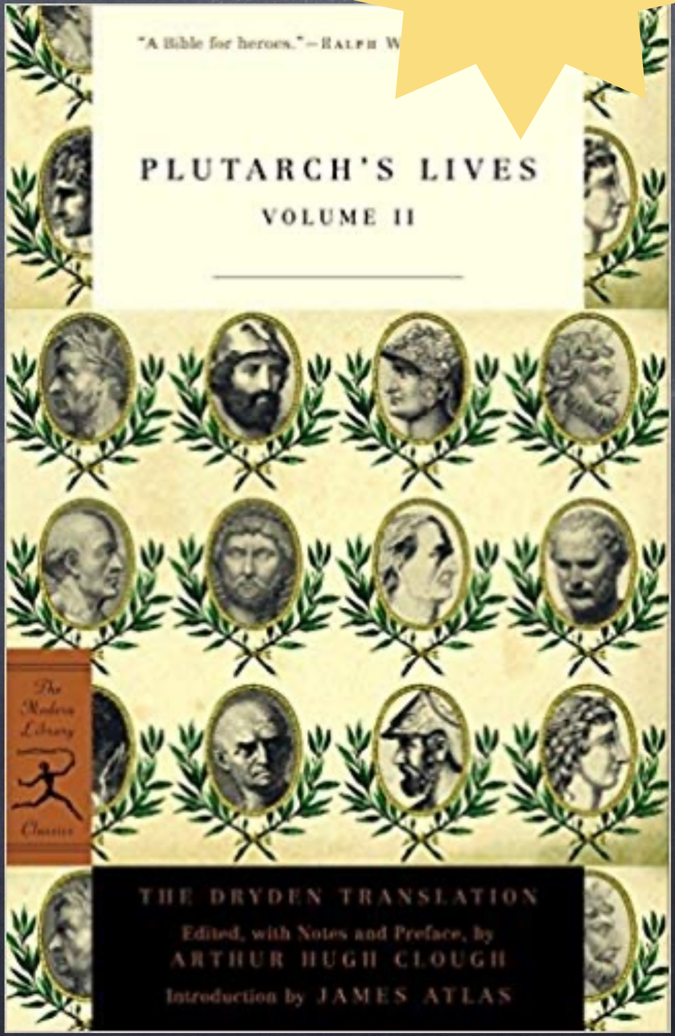


"A Bible for heroes."—Ralph Waldo Emerson

PLUTARCH'S LIVES
VOLUME I

The Modern Library Classics

THE DRYDEN TRANSLATION
Edited, with Notes and Preface, by
ARTHUR HUGH CLOUGH
Introduction by JAMES ATLAS



"A Bible for heroes."—Ralph W

PLUTARCH'S LIVES
VOLUME II

The Modern Library Classics

THE DRYDEN TRANSLATION
Edited, with Notes and Preface, by
ARTHUR HUGH CLOUGH
Introduction by JAMES ATLAS

Processes vs. Threads: Parallel lives

Processes

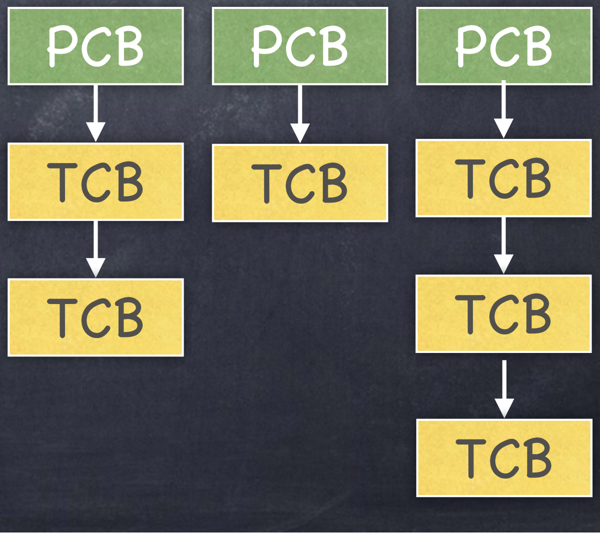
- Have data/code/heap and other segments
- Include at least one thread
- If a process dies, its resources are reclaimed and its threads die
- Interprocess communication via OS and data copying
- Have own address space, isolated from other processes'
- Each process can run on a different processor
- Expensive creation and context switch

Threads

- No data segment or heap
- Needs to live in a process
- More than one can be in a process. First calls main.
- If a thread dies, its stack is reclaimed
- Inter-thread communication via memory
- Have own stack and registers, but no isolation from other threads in the same process
- Each thread can run on a different processor
- Inexpensive creation and context switch

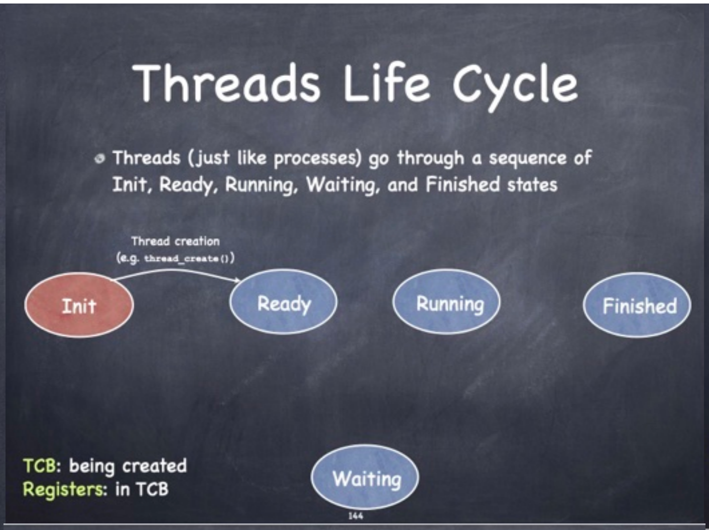
PCB vs TCB

- Several fields are in common
  - Respective ID, State, Priority, Register values
- PCB contains information about the resources shared by all that process' threads
  - memory allocation, file descriptors, signal handlers
- In multi-threaded processes, each PCB contains a pointer to a list of TCBs
- TCB has a back pointer to the PCB it belongs to



A simple API

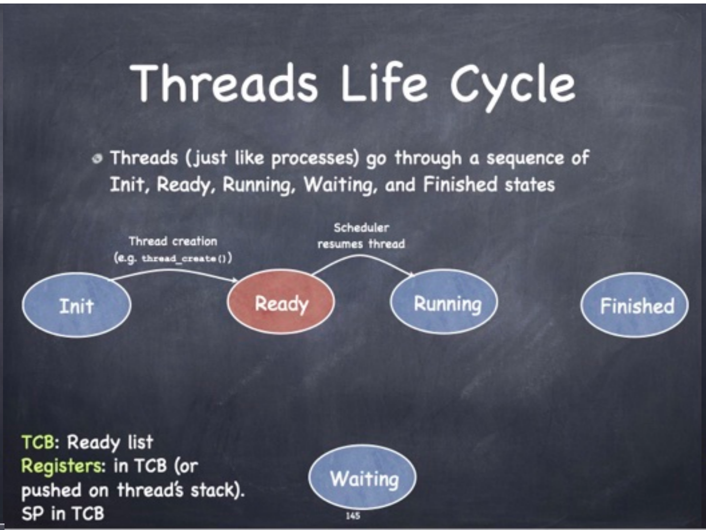
void thread_create (thread, func, arg)	Creates a new thread in thread, which will execute function func with arguments arg.
void thread_yield()	Calling thread gives up processor. Scheduler can resume running this thread at any time
int thread_join (thread)	Wait for thread to finish, then return the value 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.



Threads Life Cycle

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

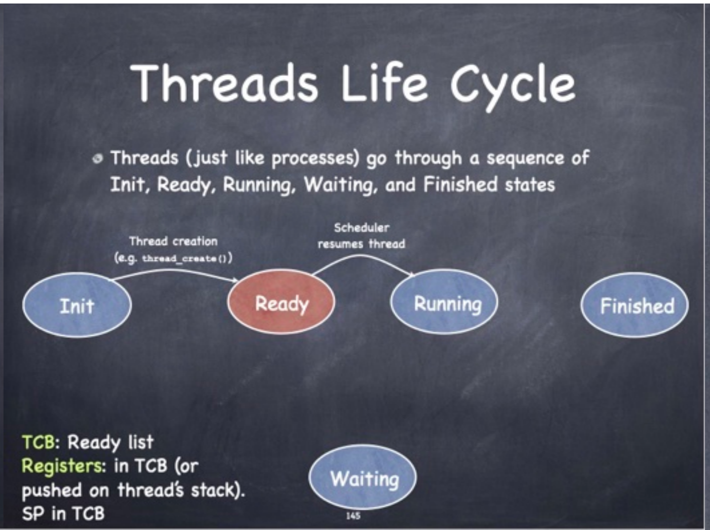
TCB: being created
Registers: in TCB



Threads Life Cycle

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

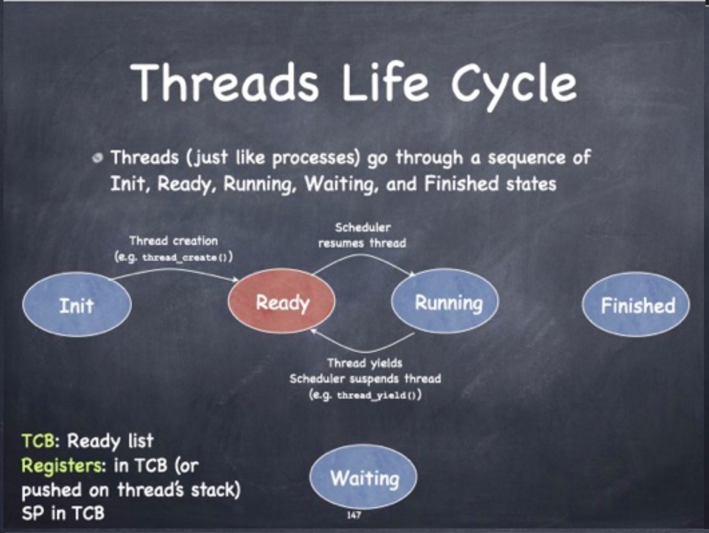
TCB: Ready list
Registers: in TCB (or pushed on thread's stack). SP in TCB



Threads Life Cycle

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

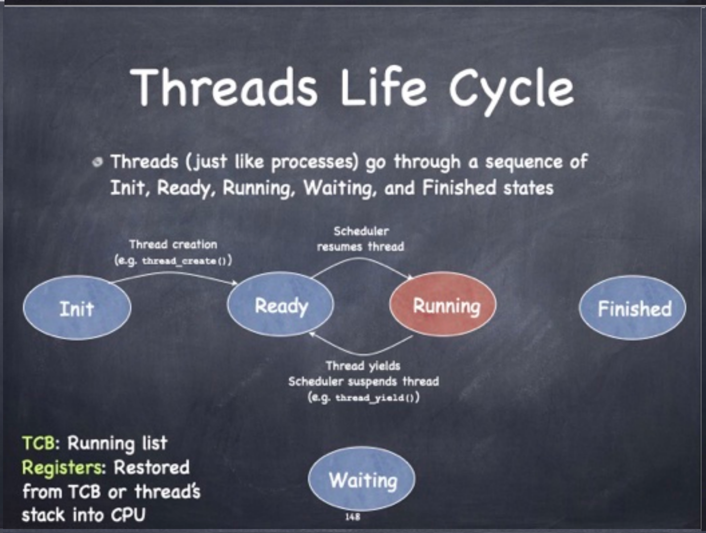
TCB: Ready list
Registers: in TCB (or pushed on thread's stack). SP in TCB



Threads Life Cycle

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

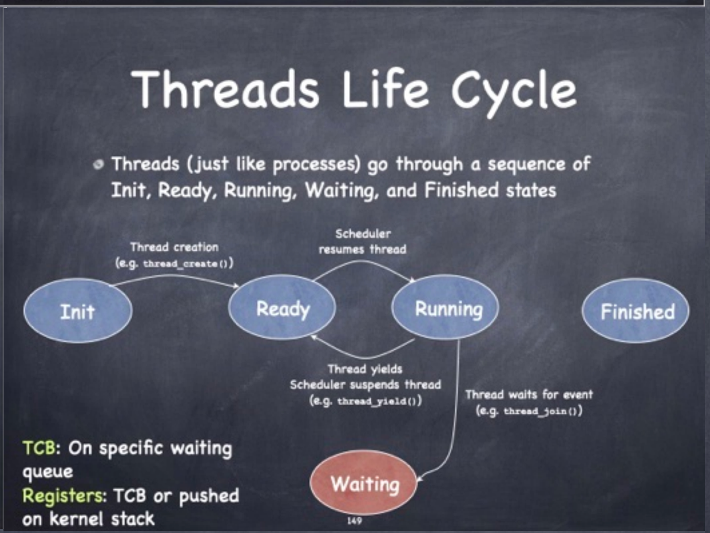
TCB: Ready list
Registers: in TCB (or pushed on thread's stack) SP in TCB



Threads Life Cycle

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

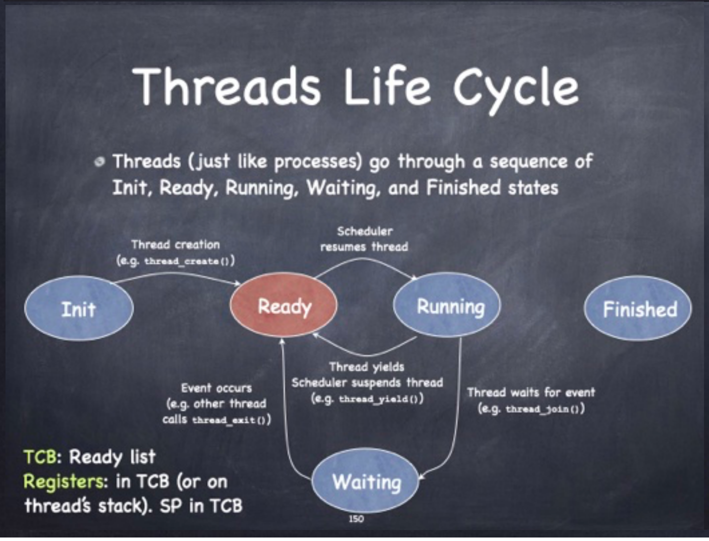
TCB: Running list
Registers: Restored from TCB or thread's stack into CPU



Threads Life Cycle

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

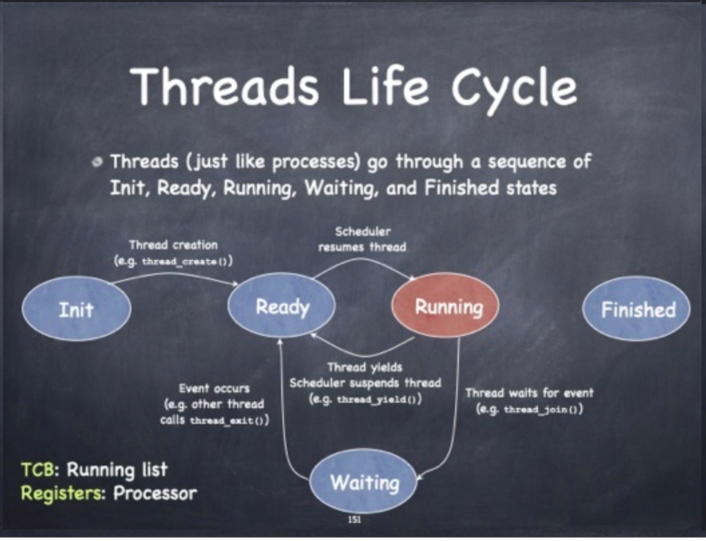
TCB: On specific waiting queue
Registers: TCB or pushed on kernel stack



Threads Life Cycle

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

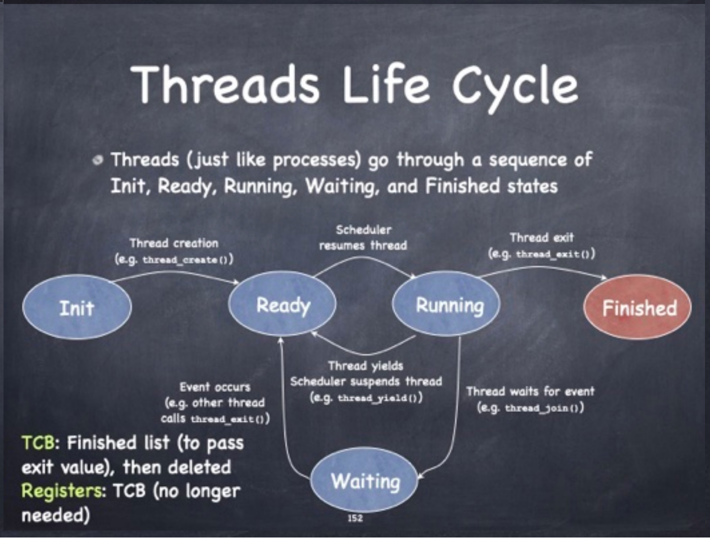
TCB: Ready list
Registers: in TCB (or on thread's stack). SP in TCB



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)

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)
    - kernel threads need not be associated with user threads

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

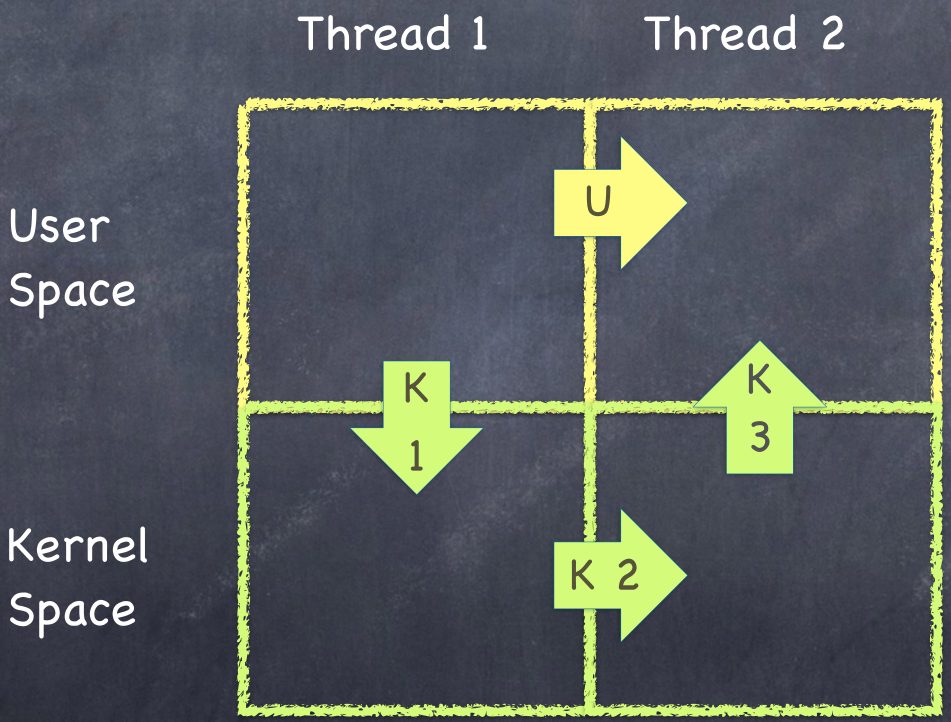
Preemption for U threads

- Use a timer signal (SIGALRM)
  - Use the alarm() or setitimer() system calls to generate a SIGALARM signal after a specified time
  - Define a signal handler for the SIGALRM signal, which must:
    - save the context of the current thread
    - select the next thread to run
    - restore its context
- User process must also maintain a ready queue to hold contexts of ready threads

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: avoiding that requires OS support for non-blocking system calls (asynch call + callback, as in scheduler activations)
Cost of context switch	Thread switch requires three context switches	Thread switch efficiently implemented in user space
Portability	Require OS support	Can be implemented on any OS
Parallelism	Can leverage multiple cores	Cannot leverage multiple cores

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

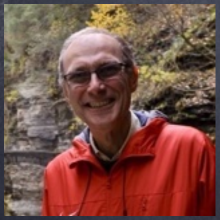
- ls -l # long list of current directory
- ps -a # lists all running processes

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 — e.g., the dispatcher
- Policy
  - determines how that functionality should be used — e.g., the scheduler

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

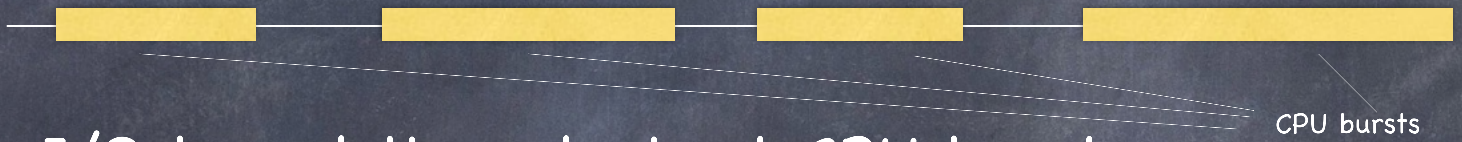
- CPU scheduler selects next process to run from the ready queue
- Disk scheduler selects next read/write operation
- Network scheduler selects next packet to send or process
- Page Replacement scheduler selects page to evict

Scheduling threads

- OS keeps TCBs on different queues
  - Ready threads are on ready queue - OS chooses one to pass to the dispatcher
  - Threads waiting for I/O are on appropriate device queue
  - Threads waiting on a condition are on an appropriate condition variable queue (we'll see about those)
- OS regulates TCB migration during life cycle of corresponding thread

Why scheduling is challenging

- Threads are not created equal!
  - CPU-bound thread long CPU bursts
    - mp3 encoding, compilation, scientific applications



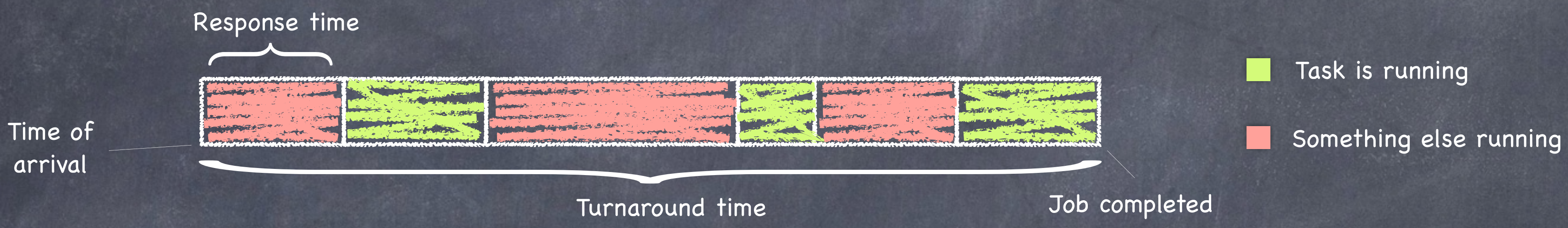
  - I/O-bound thread: short CPU bursts
    - index a file system, browse small web pages

- Problem
  - don't know type before running
  - 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 above
- **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 (e.g. 10 jobs/sec)