



THE UNIVERSITY of EDINBURGH
informatics

Operating Systems (INFR10079) 2023/2024 Semester 2

Threads (User and kernel level threading)

abarbala@inf.ed.ac.uk

Chapter 4.3 (all), 4.4, 4.4.1, 4.5, 4.5.1, 4.5.2, 4.5.3,
4.6, 4.6.1, 4.6.2, 4.6.3, 4.6.4, 4.7, 4.7.2

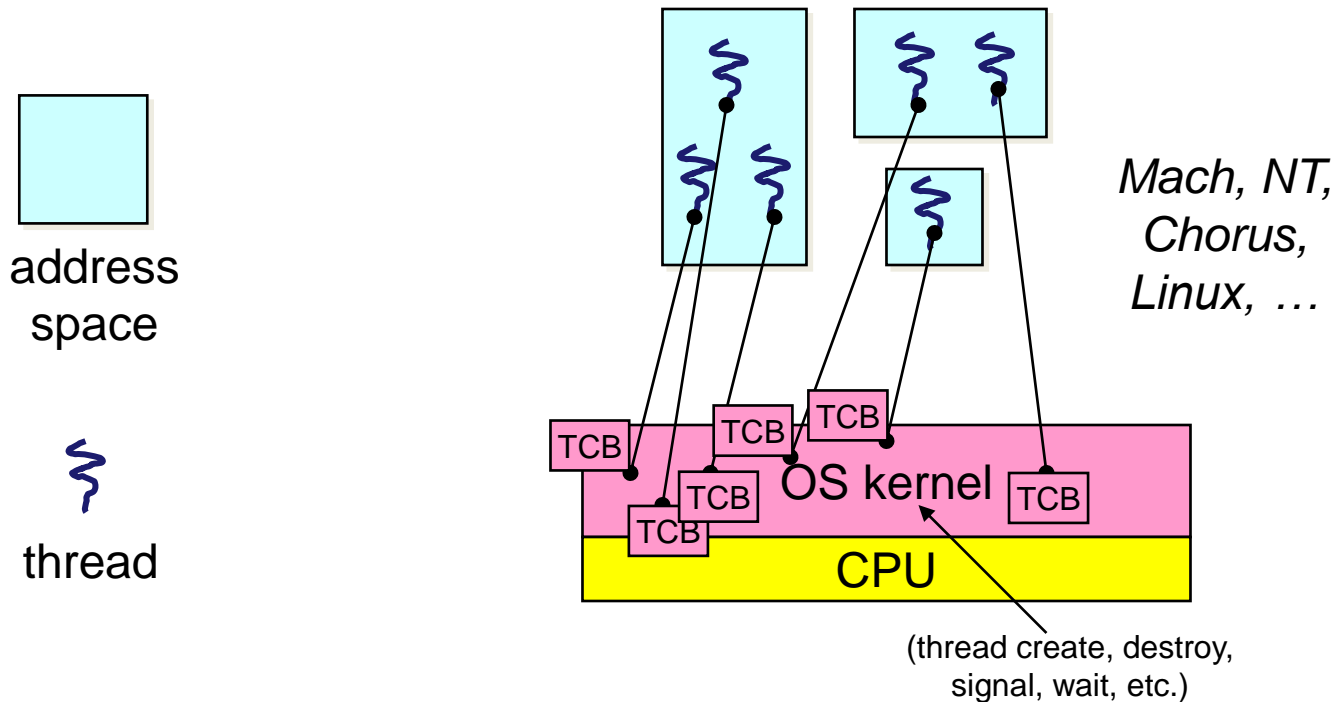
Overview

- Kernel-level Threading
- User-level Threading
- Explicit and Implicit Thread Interfaces

Who is Creating/Managing Threads?

- **OS kernel** is responsible for creating/managing threads
 - The kernel call to create a new thread would
 1. Allocate an execution stack within the process address space
 2. Create and initialize a Thread Control Block (TCB)
 - Stack pointer, program counter, register values
 3. Stick it on the ready queue
- This is **kernel-level threading**, or **1:1 threading**
 - There is a “thread name space”
 - Thread’s identifier (TID)
 - TIDs are integers, similar to PIDs
 - For each thread, a TCB, similar to PCB

Kernel-level Threading #1



There are still PCBs to describe each address space and their OS resources

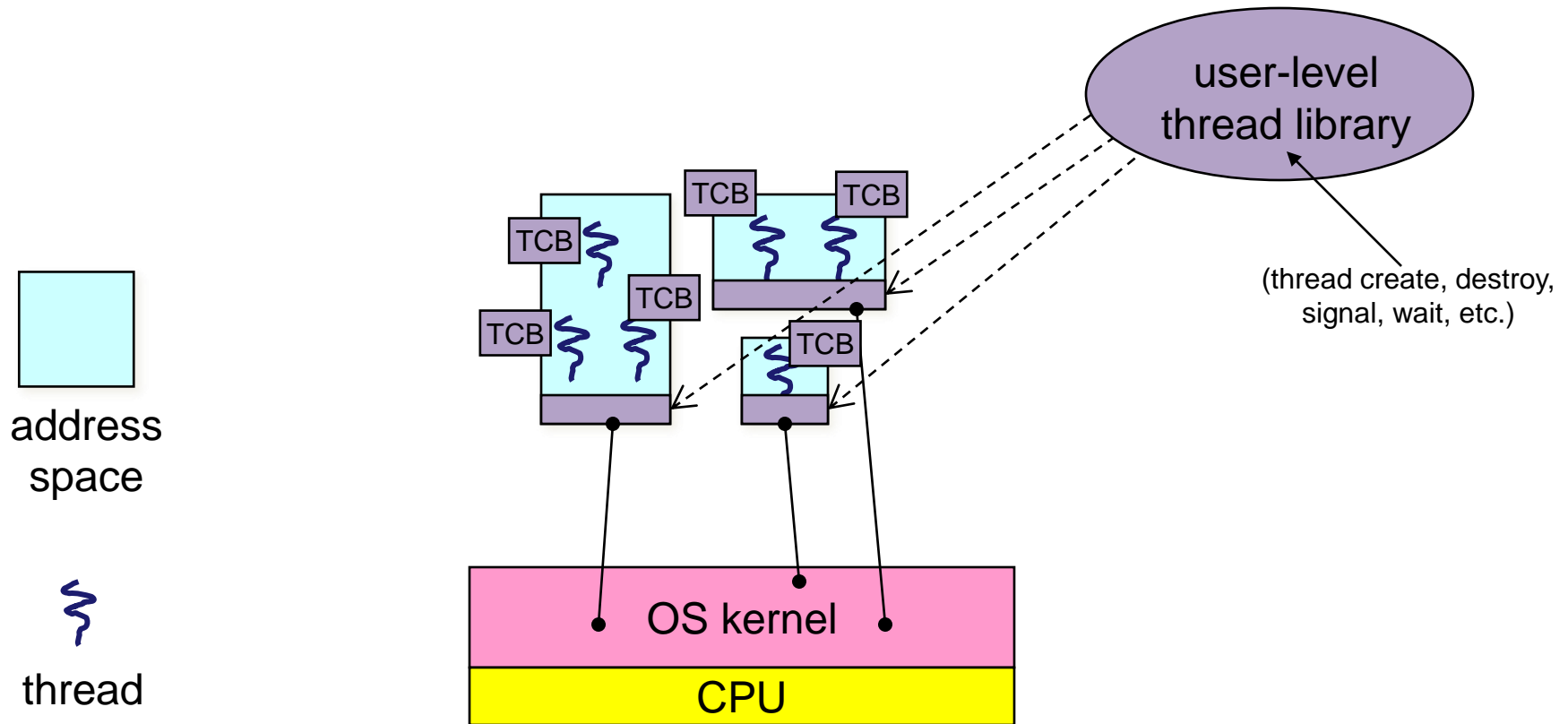
Kernel-level Threading #2

- OS manages **threads and processes**
 - All thread operations implemented in the kernel
 - OS schedules all threads in a system
 - If one thread in a process blocks (e.g., on I/O)
 - the OS knows about it, can run other threads from that process
 - Possible to **overlap I/O** and computation **within a process**
- (Kernel-managed) **Threads are cheaper than processes**
 - Less state to allocate and initialize
- But, **pretty expensive** for fine-grained use
 - Orders of magnitude more expensive than a procedure call
 - Thread operations are **system calls**
 - Context switch
 - Argument checks
 - Must maintain kernel state for each thread

User-level Threading

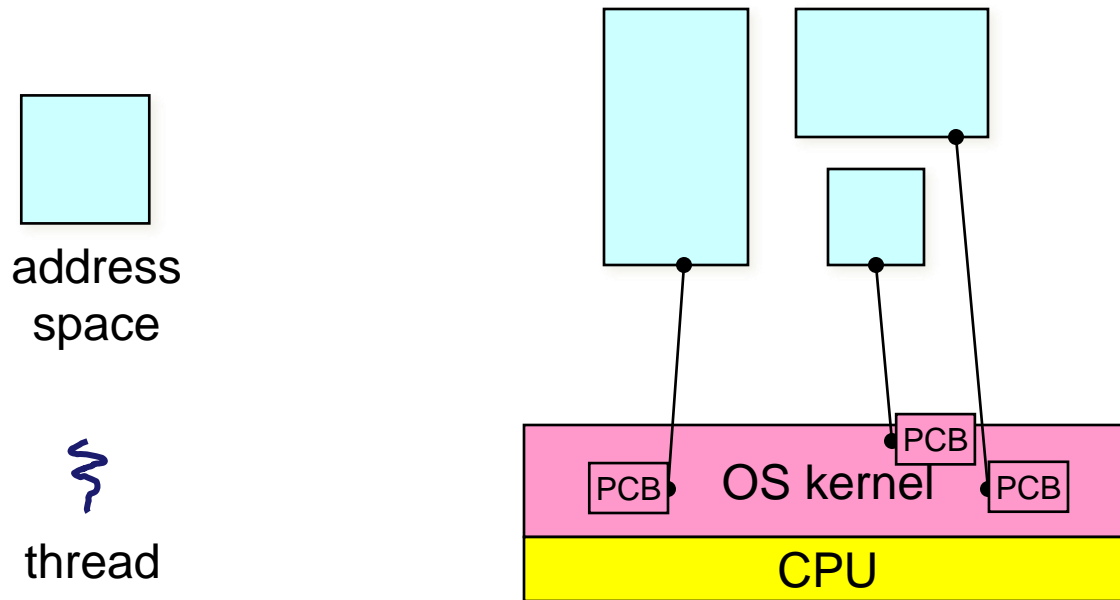
- **Alternative** to kernel-level threading
- Threads managed at the user level, **within the process**
 - **A library** into the program manages the threads
 - The thread manager **doesn't** need to manipulate **address spaces** (which only the kernel can do)
 - Threads differ (roughly) only in hardware contexts (PC, SP, registers), which can be manipulated by user-level code
 - The **thread package** multiplexes user-level threads in a process
- This is **user-level threading**, or **1:N threading**
 - Kernel is **unaware of threads existence**
 - Thread control blocks (TCBs) at user level only

User-level Threading



Now thread id is unique within the context of a process, not unique system-wide

User-level Threading: What the Kernel Sees



Why User-level Threading?

- User-level threading is **lightweight** and **fast**
 - Managed entirely by user-level library
 - Each thread is represented simply by
 - PC, registers, a stack
 - Small **thread control block**
 - Creating a thread, switching between threads, and synchronizing threads are done via **procedure calls**
 - **No kernel involvement** is necessary
- User-level threading **operations** can be 10-100x faster than kernel threads

(Old) Performance Example

- On a 700MHz Intel Pentium, running Linux 2.2.16 (only the relative numbers matter; ignore the ancient CPU and kernel)

- Processes

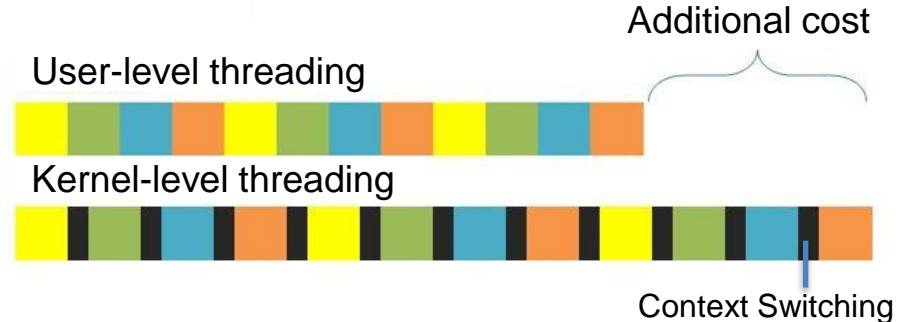
- `fork/exit`: 251 μ s

- Kernel-level threading

- `pthread_create()/pthread_join()`: 94 μ s (2.5x faster)

- User-level threading

- `pthread_create()/pthread_join`: 4.5 μ s (another 20x faster)



User-level Threading Implementation

1. **OS schedules** the process
2. Process executes user code (at user-level)
 - Including the thread support library and its thread scheduler
3. **Thread scheduler** determines when a user-level thread runs
 - Uses queues to keep track of what threads do (run, ready, wait, ...)
 - Like the OS, but in user-space as a library
4. **Context switch at the user-level**
 1. Save context of currently running thread
 - push CPU state onto thread stack
 2. Restore context of the next thread
 - pop CPU state from next thread's stack
 3. Return as the new thread
 - execution resumes at PC of next thread
 - It works at the level of the **procedure calling convention**
 - **No changes** to memory mapping required

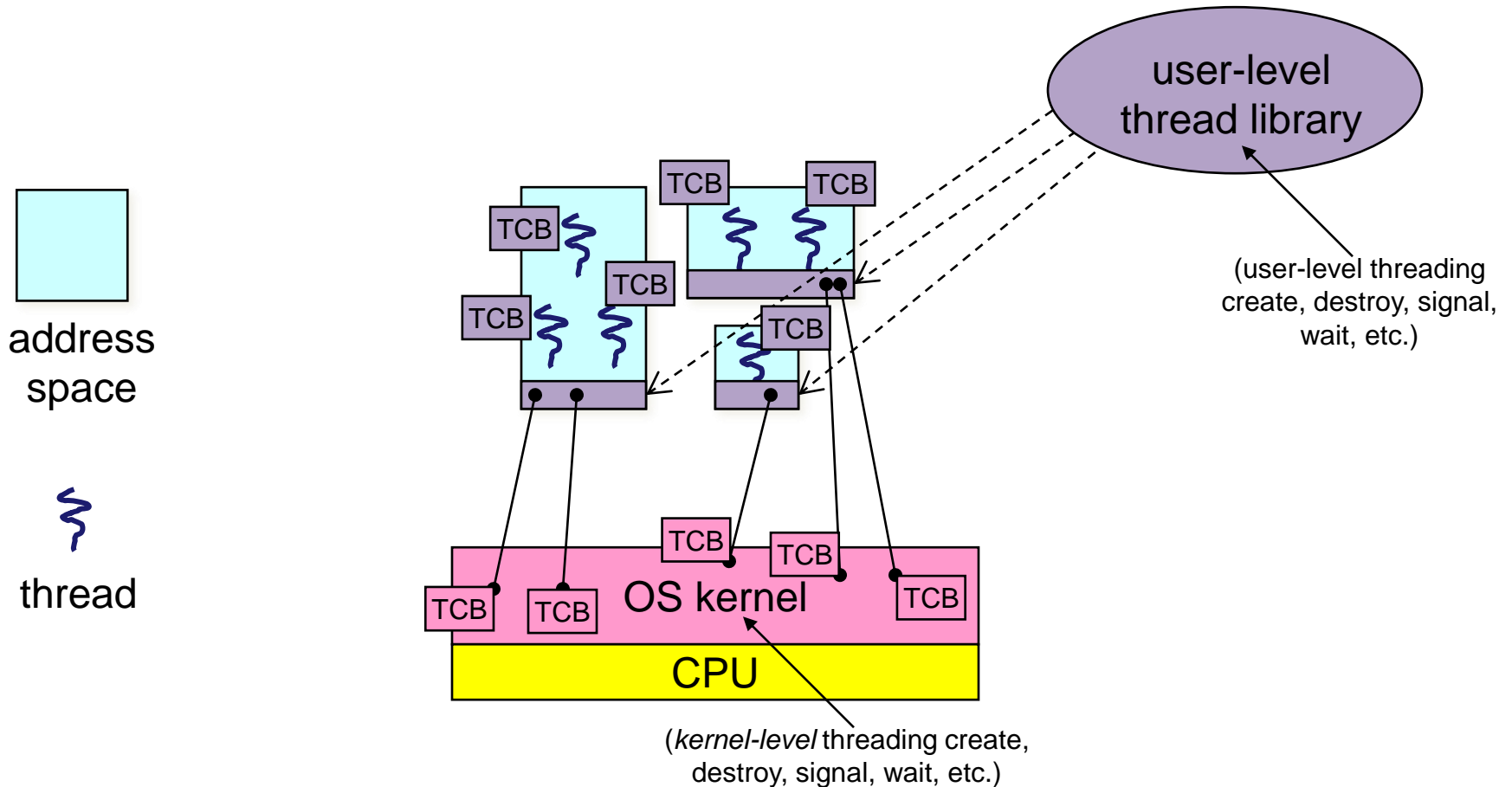
How to Keep a User-level Thread from Hogging the CPU?

- Strategy 1: **force everyone** to cooperate
 - A thread willingly gives up the CPU by calling `yield()`
 - `yield()` calls into the scheduler, which context switches to another ready thread
 - What happens if a thread never calls `yield()`?
- Strategy 2: use **preemption**
 - Scheduler requests that a **timer interrupt** be delivered by the OS periodically
 - Usually delivered as a UNIX signal (`man signal`)
 - Signals are just like software interrupts, but delivered to user-level by the OS instead of delivered to OS by hardware
 - At each timer interrupt, scheduler gains control and context switches as appropriate

What if a Thread Tries to do I/O?

- The process “powering” it **“is lost”** for the duration of the (synchronous) I/O operation!
 - The process blocks in the OS
 - The OS is not aware of the threads, OS sees one thread/process
 - No process’ thread makes progress
 - Other processes can progress tho
- This is **not the case** with kernel-level threading
 - Kernel knows about each process’ thread
 - Another thread can be schedule
- *Can kernel-level threading and user-level threading be merged?*

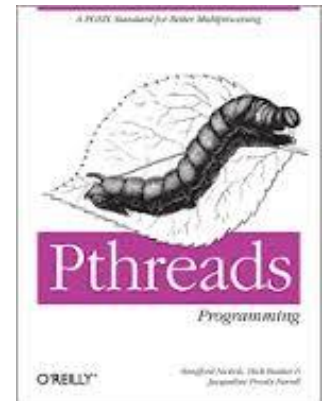
The N:M Threading Model (Merges 1:1 and 1:N Models)



Explicit Thread Interface (User- or Kernel- level)

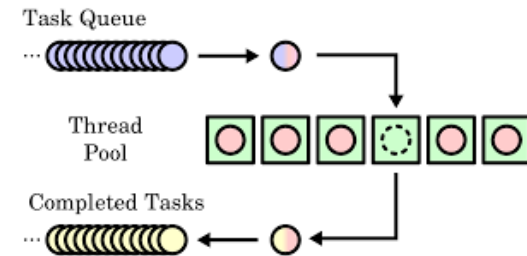
Try at
home!

- POSIX Thread (pthread) APIs
 - `ret = pthread_create(&t, attributes, start_procedure)`
 - Creates a new thread of control
 - New thread begins executing at `start_procedure`
 - `pthread_cond_wait(condition_variable, mutex)`
 - The calling thread blocks on a conditional variable
 - `pthread_signal(condition_variable)`
 - Starts a thread waiting on the condition variable
 - `pthread_exit()`
 - Terminates the calling thread
 - `pthread_join(t)`
 - Waits for the named thread to terminate



Implicit Thread Interface (User-level)

- Thread management to library/runtime
- Identify application's tasks, not threads
- Compiler-level support (in most cases)
 - Code annotation
 - Pragmas
 - Templates
- Examples
 - Thread pools
 - Fork-join
 - OpenMP
 - Grand Central Dispatch
 - Intel Thread building blocks



Summary

- Multiple threads per process (and address space)
 - **Real resource sharing** for multiple instruction flows
- **Kernel-level threading (1:1)** implemented in OS kernel
 - All operations require a kernel call and parameter validation
 - Enables concurrency and parallelism
- **User-level threading (1:N)** implemented in application
 - Cheaper and faster
 - Enables concurrency
 - Great for common-case operations
 - Creation, synchronization, destruction
 - May block all threads on the same process
 - Blocking IO
- **N:M threading**
 - Best of both the previous