CS313 Operating Systems

Lecture 13: Thread Implementation: User Level Threads, Kernel Managed Threads

Ravi Mittal
IIT Goa

Acknowledgements !

- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - CS162, Operating System and Systems Programming, Profs. Natacha Crooks and Anthony D. Joseph, University of California, Berkeley
  - CS240 Computer Systems, Univ of Illinois, Prof. Wade Fagen-Ulmschneider
  - Operating Systems: Three Easy Pieces, by Remzi and Andrea Arpaci-Dusseau,
    available for free online
  - Book: Modern Operating Systems, Andrew Tenenbaum, and Herbert Bos, 4th Edition, Pearson

References

- Book: Operating Systems: Principles and Practice: Thomas Anderson and Michael Dahlin, Part 2
- CS162, Operating Systems and Systems Programming, University of California, Berkeley
- CS4410, Operating Systems, Course, Cornell University, Spring 2019, Lecture on Threads
- Operating Systems: Three Easy Pieces, by Remzi and Andrea Arpaci-Dusseau,
  available for free online

Read the following:

- Book: Operating Systems: Principles and Practice (2nd Edition) Anderson and Dahlin
  - Volume 2, Concurrency
    - Chapter 4: Concurrency and Threads
- Book: Modern Operating Systems: Tenenbaum and Bos
  - Chapter 2: Processes and Threads

We will study..

- User Space Thread and Kernel Space Threads
- Thread Library - Introduction

Recall: Thread

A thread is a single execution stream in a process which can be independently scheduled by the kernel and shares the same addressable space with any other thread.

Recall: Thread and Concurrency

- Operations are concurrent if they may be arbitrarily interleaved so that they make progress independently
- Threads can be scheduled to use the CPU in any order with any other thread
- A thread may be removed from the CPU at any point of execution and replaced by another thread
- Operations which cause one thread to suspend execution do not cause the execution of other threads to be suspended
- If the order that threads execute matter, or a thread must fully complete a task before other threads can execute, then the thread execution must be synchronized to coordinate action

Recall: Thread's addressable Memory

- Any data whose address may be determined by a thread are accessible to all threads in the process
- This includes static variables, automatic variables, and storage obtained via malloc
- This does not mean that a thread cannot have private data
  - POSIX includes a way for a thread to have data which only it can access

Problems of Multithreaded Programs

- Computational overhead of thread synchronization and scheduling overhead, which may be intolerable on programs with little parallelism
  - So, poorly programmed multithreading may take more time
- Greater programming discipline to plan and coordinate different execution sequences
- Bad code breaks in ways that can be very hard to locate and repair

User Space Thread and Kernel Space Thread

User Threads, Kernel Threads

- There are two main places to implement threads
  - User Space
  - Kernel Space

User thread - Concept

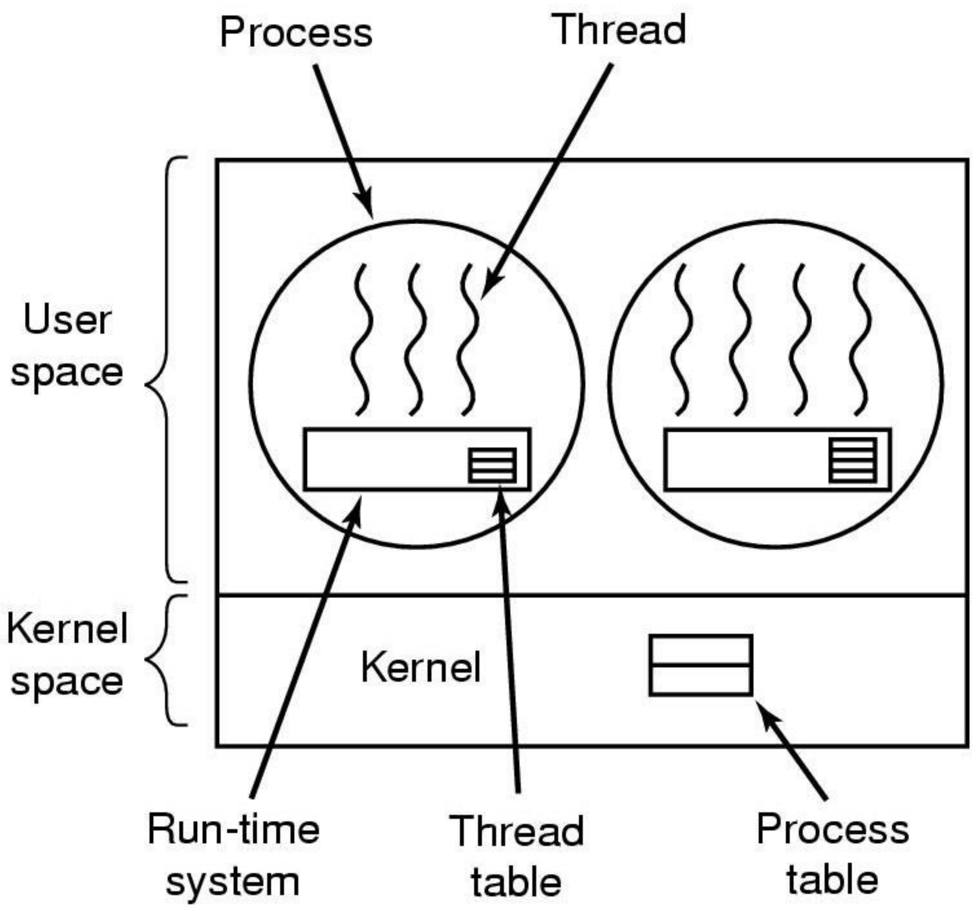
- Implement thread package entirely in user space
- All thread management is done by the application

- Kernel knows nothing about it – it continues to manage ordinary single threaded process
- Advantage:
  - A user-level thread package can be implemented on an OS that doesn’t support threads
- Most old generation Operating systems didn’t support threads
- Hence, threads were implemented by a library

User Thread - Concept

- Implement thread creation/scheduling using procedure calls to a user-level library rather than system calls
  - User-level library implementations of `thread_create()`, `thread_yield()`, etc.
- User level library performs same set of actions as corresponding system calls, but thread management is controlled by user-level library
- The entity created by the library to implement the POSIX specification is called a user thread
- Examples: Mach C-threads, BSD, Solaris UI-threads and DCE-threads.

Implementing Thread in User Space (1)



- Threads run on top of run-time system
  - Collection of procedures to manage threads
- Each process needs own private thread table (Thread control Block)

Implementing Thread in User Space (2)

- Thread Table (or Thread Control Block)
  - Keeps track of the threads in the process
  - Per thread properties: PC, SP, registers, state, etc
- Thread table is managed by runtime system
  - When a thread moves to ready state or blocked state, the information required to restart it is stored in Thread table (or TCB)
  - If a running thread may become locally blocked
    - Eg. Waiting for another thread (in the same process) to complete some work
  - It calls for run-time system procedure
  - The procedure checks if the thread must be put into blocked state. If so, it stores threads state into the table (TCB)
  - It re-starts a ready thread by loading it's registers from the thread's table or TCB

Advantages of User space threads

- In machines, processor supports
  - an instruction to store all registers
  - Another instruction to load them all
  - □ Thread switch can be done with a few instructions
- Thread switching is very fast compared to use of kernel trappings for switching
- A thread may run `thread_yield`
  - The code for `thread_yield` saves thread state into the table/TCB and calls scheduler to pick up another thread
    - Local procedures – more efficient than making a kernel call
    - No trap is required – no context switch (by Kernel) is required
  - Makes scheduling very fast

Advantages of User space threads

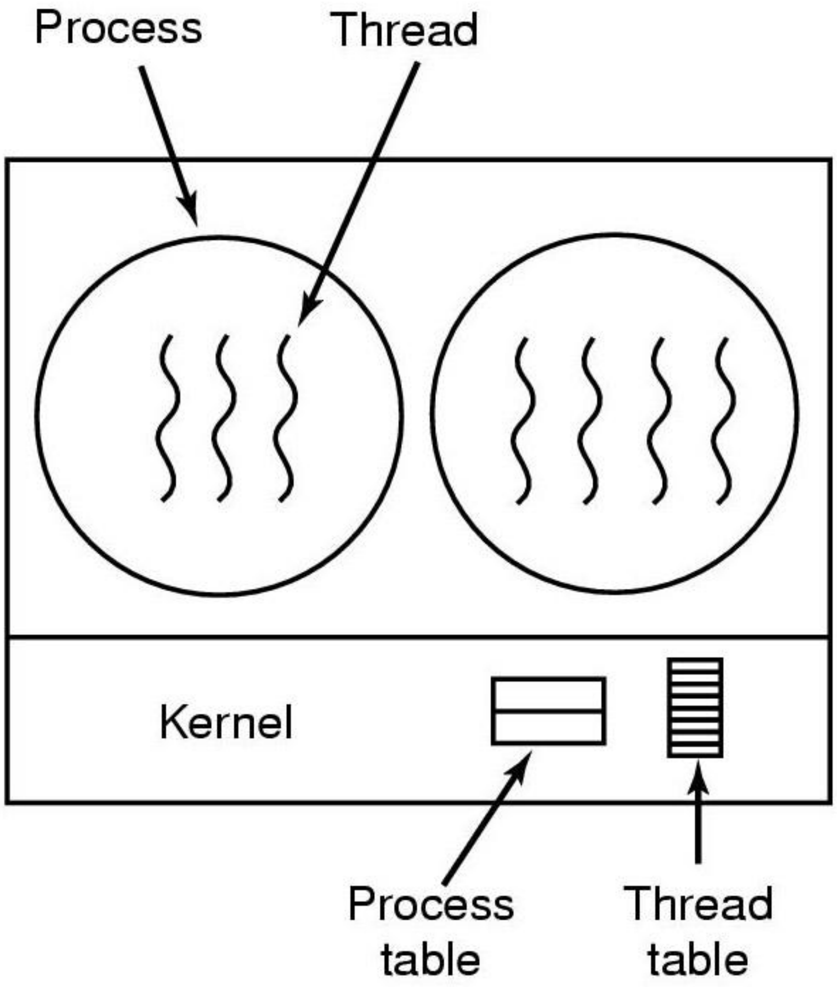
- Each process can have its own customized scheduling algorithm
- Easy garbage collection by garbage collection threads
- Overall better scaling – large number of threads

Disadvantages of User level threads

- Blocking System call implementation
  - A thread making a syscall can block all threads
  - One solution is to make many syscalls non-blocking
- A page fault by a thread – entire process (all threads including) will be blocked
- If a thread starts running, there is no way to stop it
  - Unless OS moves the process out
  - Or thread voluntarily exits
- Cannot take advantage of multiprocessor architecture
  - Why?

Thread Management in Kernel (1)

- Kernel knows how to manage threads in a user process
  - No run-time system required
  - No thread table in each process
- When a thread wants to create a new thread or destroy an existing thread, makes a system call
  - Kernel creates or destroys thread as per request.
  - Updates kernel thread table



(PCBs) (TCBs)

Thread Management in Kernel (2)

- All calls that might block the thread are implemented as system call
- When a thread blocks, the kernel can run another thread
  - From the same process or
  - different process
- Advantages:
  - Can take full advantage of multiprocessor architecture within a single process
- Disadvantages:
  - Poor scaling when many threads are used, as each thread takes kernel resources from the system
  - Generally, slow thread context switches since these require the kernel to intervene.

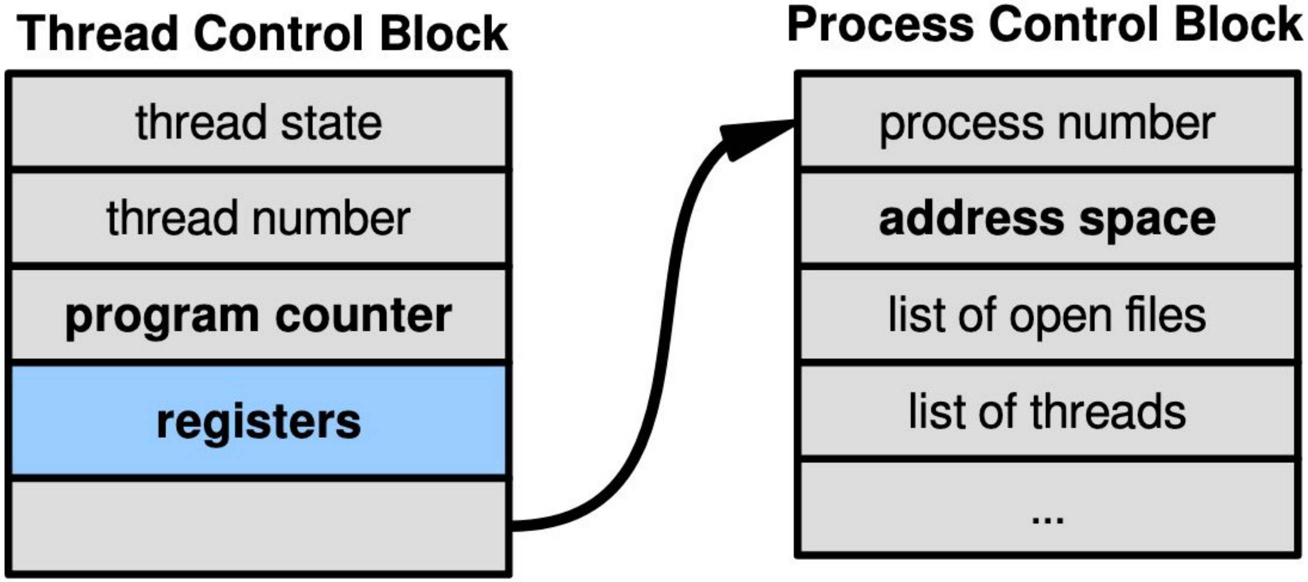
Kernel Managed Threads

- Examples:
  - Windows operating system and Linux Threads

- Kernel managed threads: Some authors call these as Kernel threads (we use the term: Kernel managed threads)

Thread Management in Kernel (3)

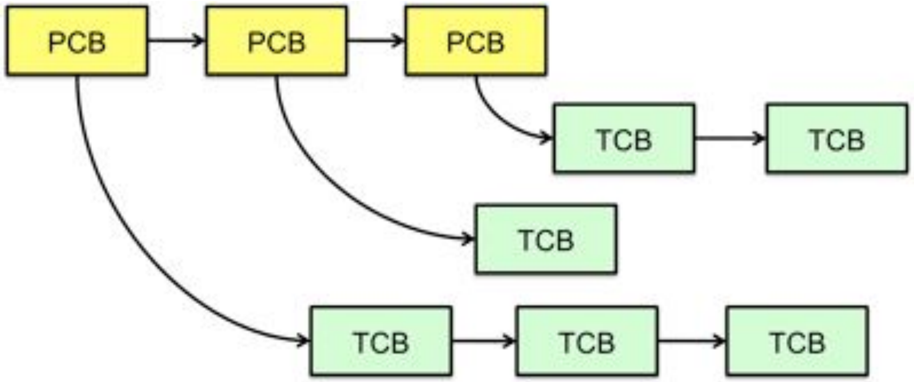
- Each thread has a thread control block
  - CPU registers, including PC, pointer to stack
  - Scheduling info: priority, etc.
  - Pointer to Process control block
- OS scheduler uses TCBs, not PCBs



Thread Management in Kernel (4)

- Kernel's Thread table/TCB implementation is kernel dependent
- Thread Table/TCB holds register values, PC, etc
  - Kept in kernel space
- Information about each process is kept in Process Table/PCB

Kernel Space



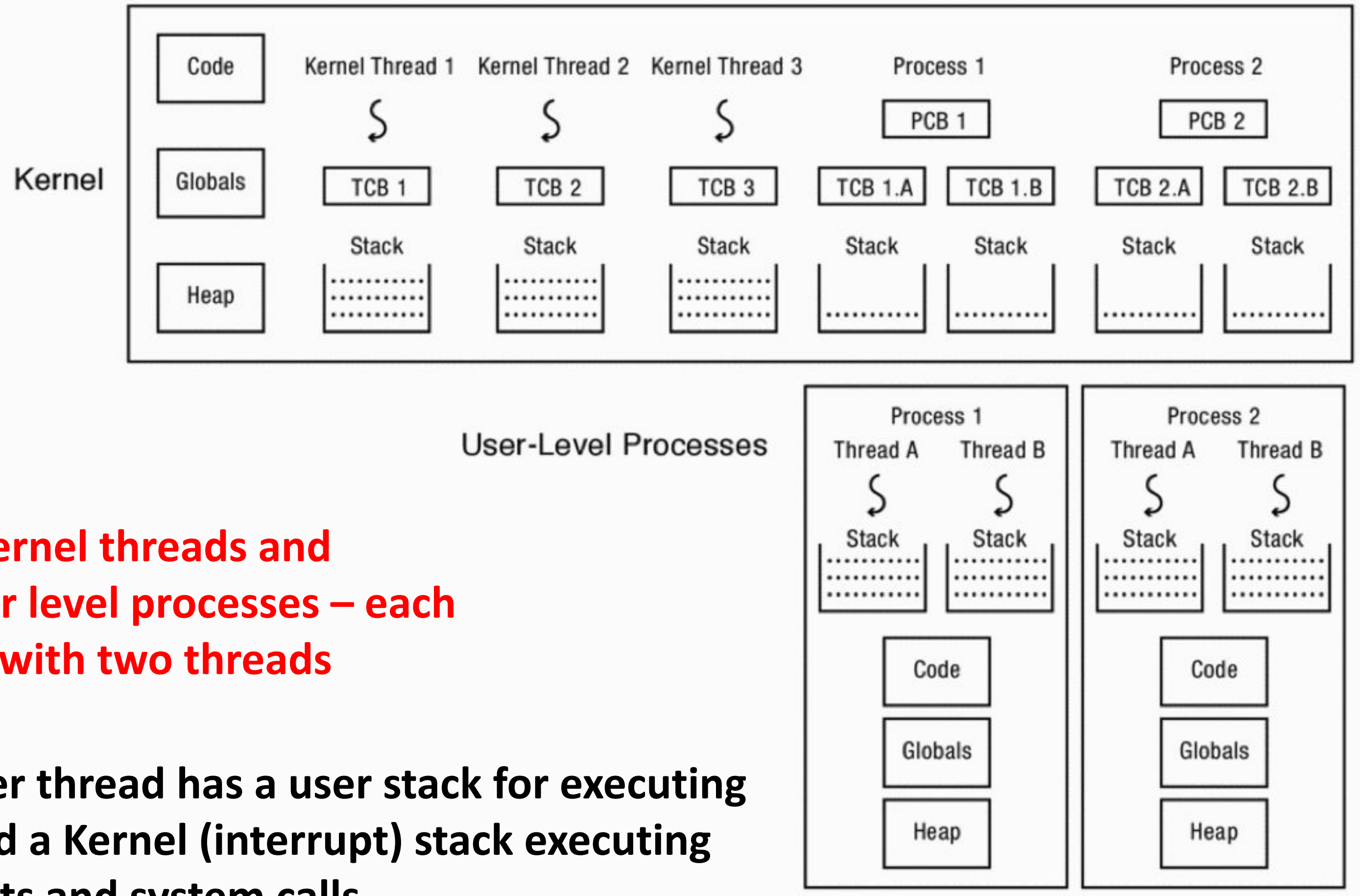
Other implementations are also possible

Kernel Threads

- Threads created by kernel for Kernel Processes (Using Kernel code)

- A kernel thread executes kernel code and modifies kernel data structures

Thread Management in Kernel (4)



Three Kernel threads and
Two user level processes – each
process with two threads

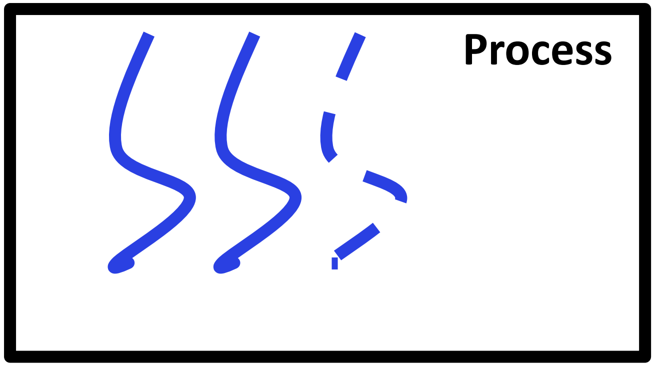
Each user thread has a user stack for executing code and a Kernel (interrupt) stack executing interrupts and system calls

Thread management is done in Kernel

Thread Management: User level /Kernel level

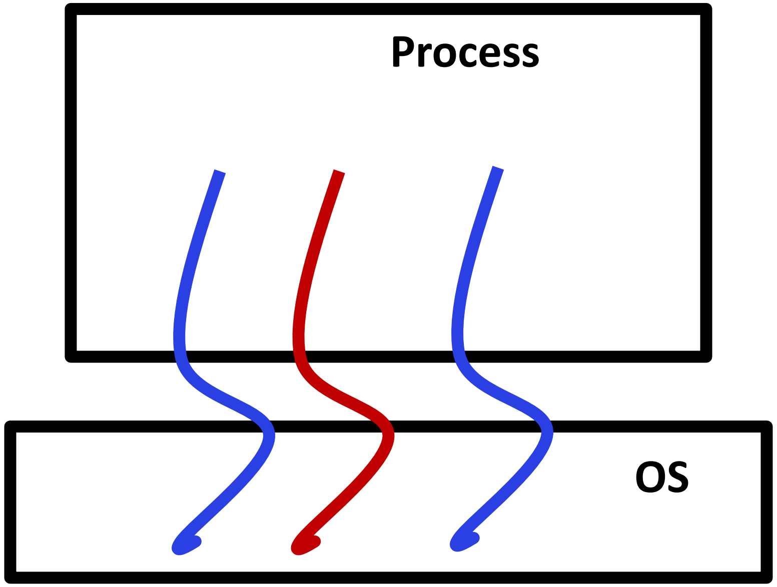
A Thread can be scheduled separately by a scheduling algorithm; Eg running only third thread

Management at user level



A Thread be blocked without blocking other threads; ex: Only second thread is blocked

Management at Kernel level



Threads can run on different CPUs
Good from perf perspective
Bad from management perspective

Thread Execution

Correctness with Concurrent Threads

- Non-determinism:
  - Scheduler can run threads in **any order**
  - Scheduler can switch threads **at any time**
  - This can make testing very difficult
- Independent Threads:
  - No state shared with other threads
  - Deterministic, reproducible conditions
- Cooperating Threads:
  - Shared state between multiple threads

- **Goal: Correctness by Design**

Race Conditions

- Initially `x == 0 and y == 0`

```
Thread A      Thread B
x = 1;        y = 2;
```

- What are the possible values of x below after all threads finish?

- Must be **1**. Thread B does not interfere

Race Conditions

- Initially `x == 0 and y == 0`

```
Thread A        Thread B
x = y + 1;      y = 2;
                y = y * 2;
```

- What are the possible values of x below?
- 1 or 3 or 5 (non-deterministically)
- Race Condition: Thread A races against Thread B!

Example 2: Parallelization (self reading)

- Consider the following code segment:

```
for (k = 0; k < n; k++)
    a[k] = b[k] × c[k] + d[k] × e[k]
```

- Is there a missed opportunity here?

```
thread_create(T1, fn, 0, n/2)
thread_create(T2, fn, n/2, n)

fn(l,m) {
  for (k = l; k < m; k++)
     a[k] = b[k] × c[k] + d[k] × e[k]
}
```

Example : Web Server (self reading)

- Consider a Web server
  - get network message from client
  - get URL data from disk
  - compose response
  - send response

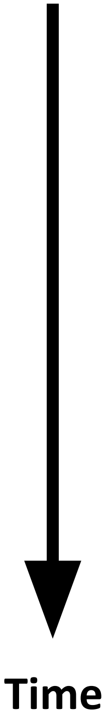
Example 3: Web Server (self reading)

- Consider a Web server

  Create a number of threads, and for each do

  - get network message from client
  - get URL data from disk
  - compose response
  - send response
- What did we gain?

Example 3: Web Server - Overlapping I/O and Computation (self reading)



Request 1

Thread 1

get network message (URL) from client

get URL from disk

(disk access latency)

send data over network

Request 2

Thread 2

get network message (URL) from client

get URL from disk

(disk access latency)

send data over network

Total time is less than Request1 + Request 2 done separately

Multithreaded Programs

- You know how to compile a C program and run the executable
  - This creates a process that is executing that program

- Initially, this new process has *one thread* in its own address space
  - With code, globals, etc. as specified in the executable

- Q: How can we make a multithreaded process?
- A: Once the process starts, it issues *system calls* to create new threads
  - These new threads are part of the process: they share its address space

Thread Management

- Threads are identified by a process unique thread ID
- When threads are created they begin executing a function, whose parameter is passed during creation
  - Compare process creation with fork and exec, when an argument list is passed
- Threads can exit or be terminated by other threads
- A thread can wait for another thread and collect its return value
- Threads have modifiable attributes like priority

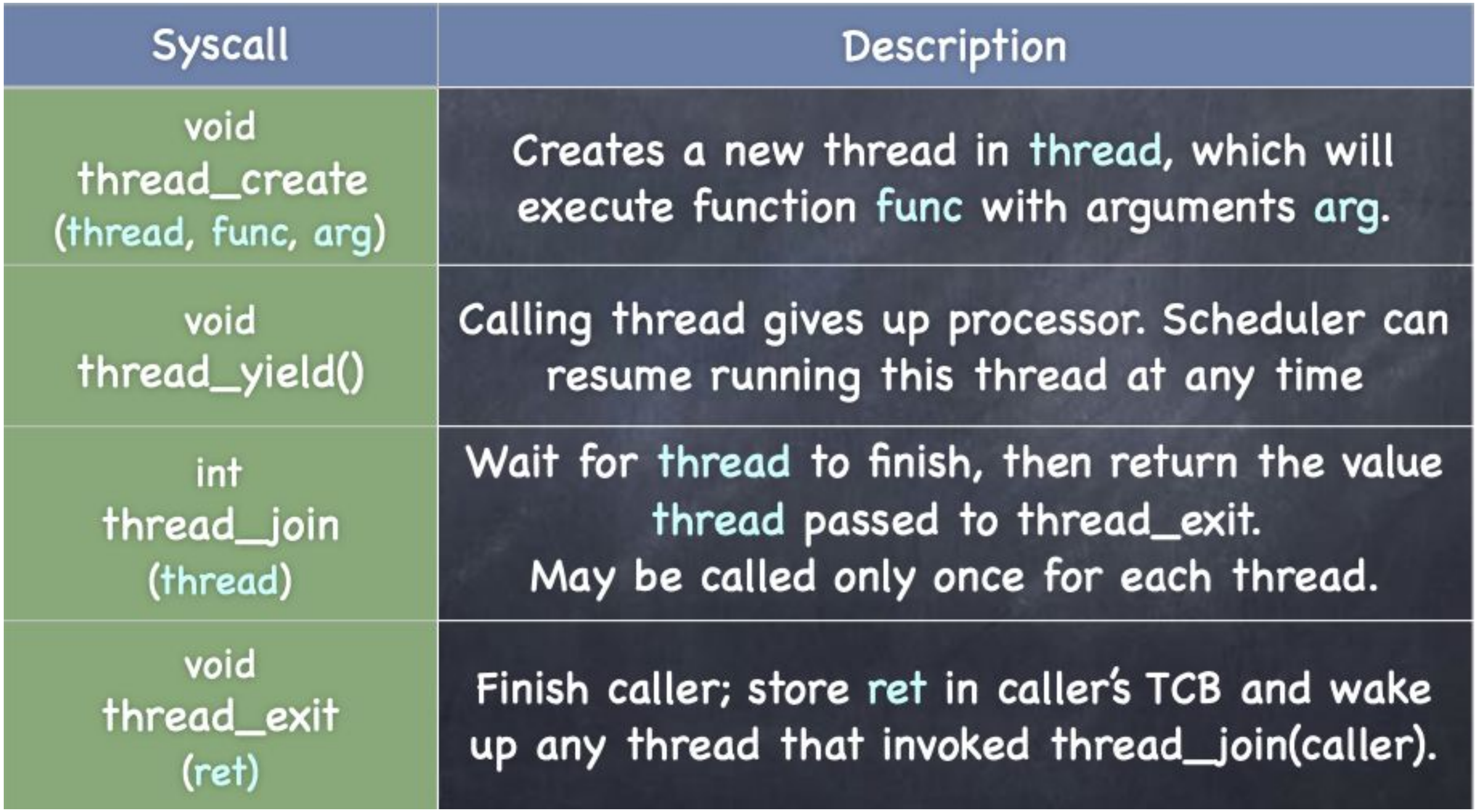
Thread Libraries

- Three main Thread Libraries in use today
  - POSIX Pthread
    - User or Kernel level
  - Windows
    - Kernel-level
  - Java
    - Using Windows APIs
- Linux, Unix, MacOS
  - **Pthreads**

Thread Operations in general

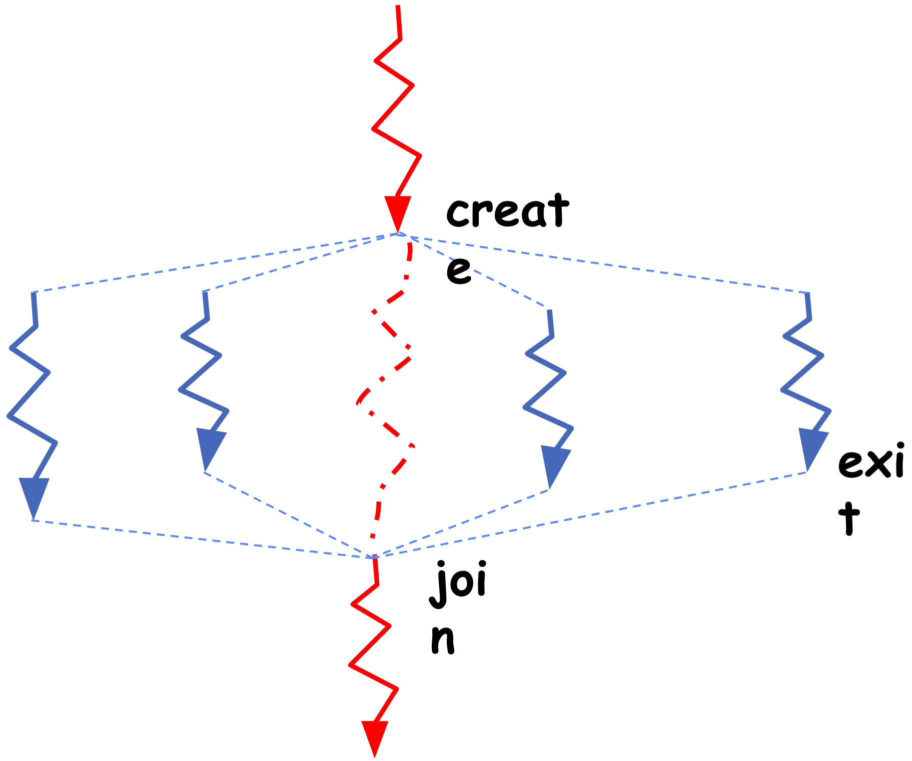
- `thread_create(thread, func, args)`
  - Create a new thread to run func(args)
- In some OS: `thread_fork`
- `thread_yield()`
  - Relinquish processor voluntarily
- `thread_join(thread)`
  - In parent, wait for forked thread to exit, then return
- `thread_exit`
  - Quit thread and clean up, wake up joiner if any

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. May be called only once for each thread.
void thread_exit (ret)	Finish caller; store ret in caller's TCB and wake up any thread that invoked thread_join(caller).

Fork-Join Pattern



- Main thread *creates* (forks) collection of sub-threads passing them args to work on, *joins* with them, collecting results.

Lecture Summary

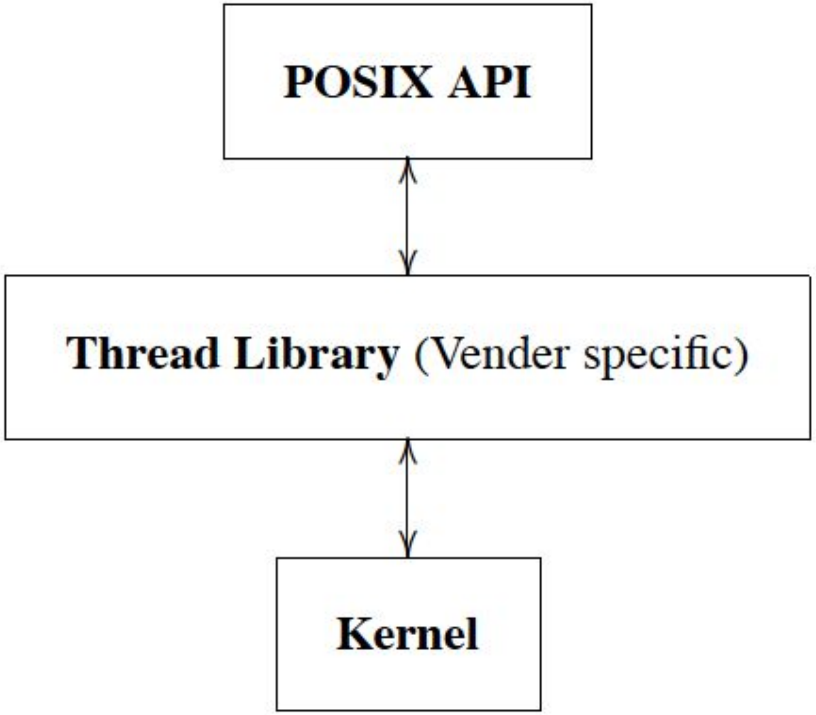
- Threads abstraction gives illusion of infinite number of processors
- User level threads can be managed at user level by run-time system
- User level threads can also be managed at Kernel level
  - Popular
- Concurrent execution of thread (of a process) may lead to
  - Non-determinism
- Thread operations include
  - Thread_create, thread_yield, thread_join, thread_exit
- Multithreading model defines
  - User level threads
  - Kernel level threads

Thread Library: pthread

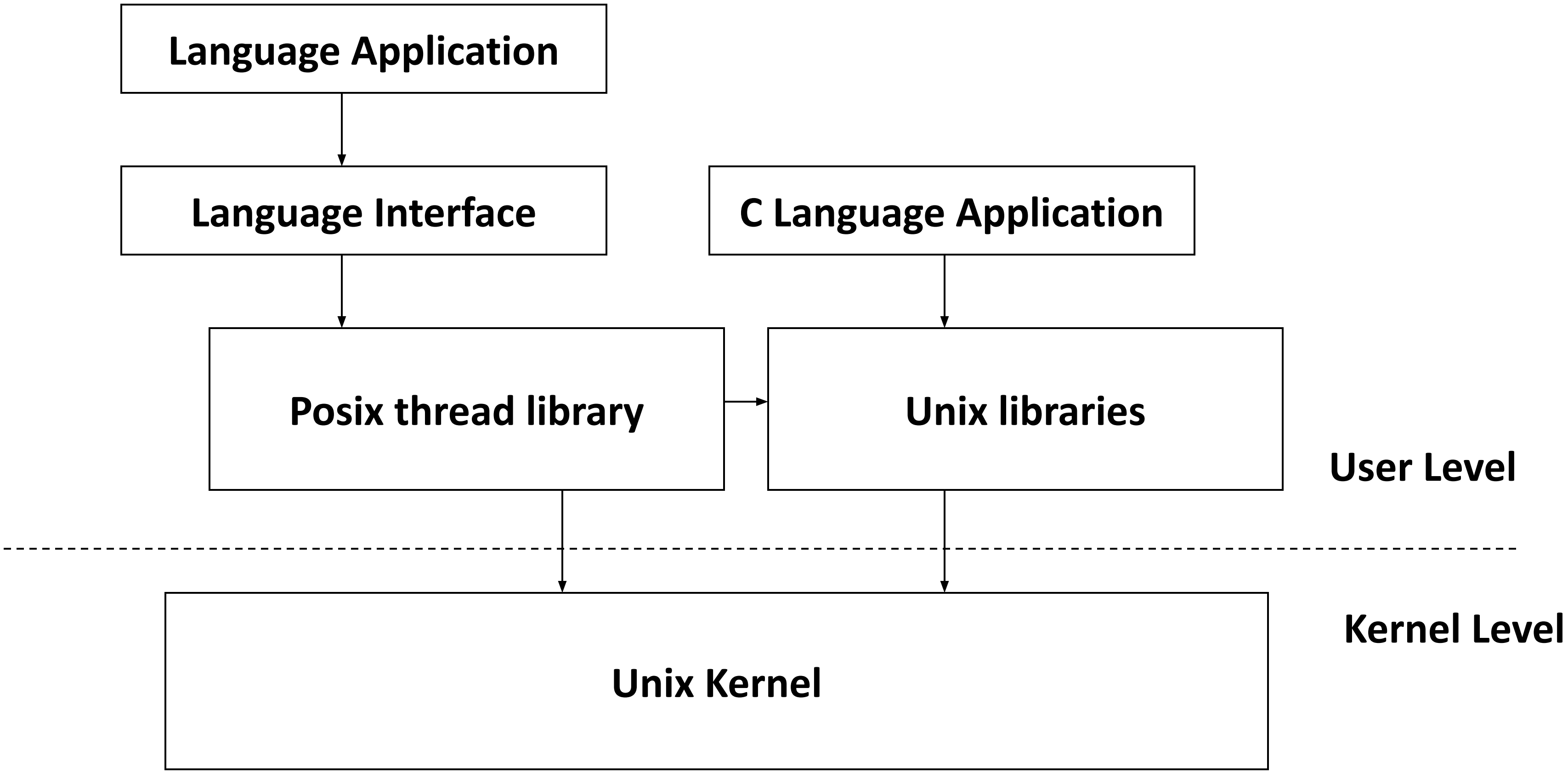
What are pthreads?

- POSIX standard IEEE 1003.1c defines a thread interface
  - `pthreads`
- Unix/Linux provides `pthread` library
  - APIs to create and manage threads
- Implementation is up to development of the library
- Simply a collection of C functions
- Standard interface for ~60 functions that manipulate threads from C programs
- Primary way of doing threading in Linux is `pthread`
- Since 2003 (Kernel 2.6), Linux implements POSIX threads as kernel-scheduled threads

Thread Implementation: Layers of Abstraction



POSIX Library Implementation (F. Mueller's Paper)


Language Application
Language Interface
C Language Application
Posix thread library
Unix libraries
User Level
Kernel Level
Unix Kernel