University of Nottingham
UK | CHINA | MALAYSIA

Operating Systems and Concurrency

Lecture 5&6: Threads

University of Nottingham, Ningbo China

2024

Recap

Last Lecture

- **Types of schedulers**: preemptive/non-preemptive
- Performance **evaluation criteria**
- Scheduling **algorithms**: FCFS, SJF, Round Robin, Priority Queues

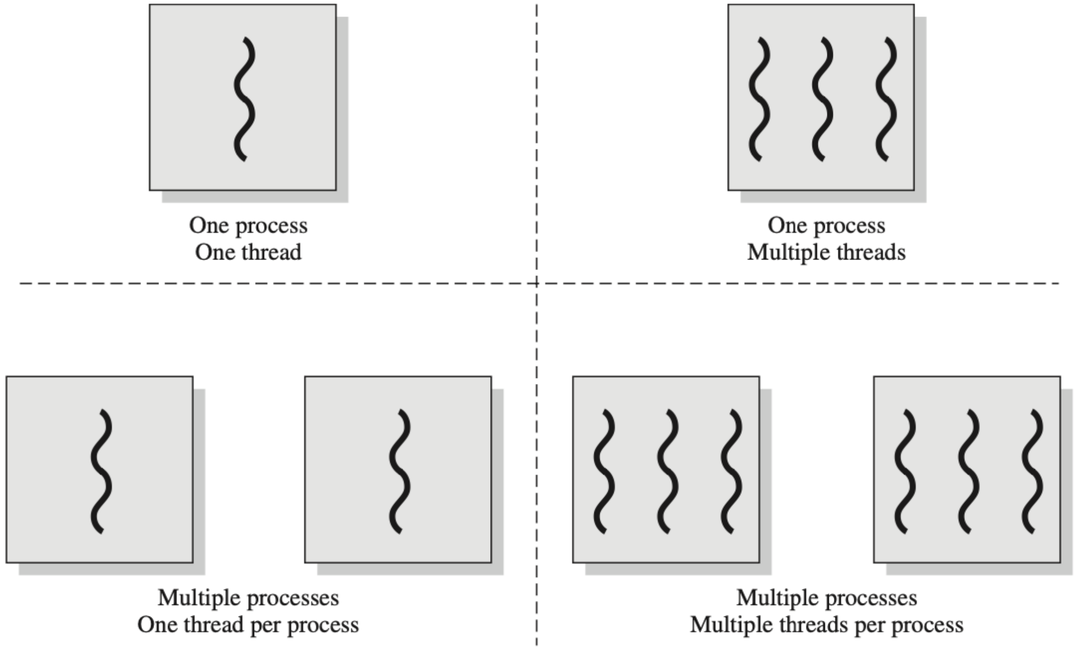
Goals for Today

Overview

- Thread definition
  - Key characteristics
- Threads vs. processes
- Thread usage
- When to use thread
- Hyperthreading (Simultaneous Multithreading or SMT)
- Different thread implementations

Thread Usage
What is thread?

- **Definition**:
  - A thread is the smallest unit of execution within a process.
  - It represents a single sequence of instructions that can be managed independently by a scheduler.

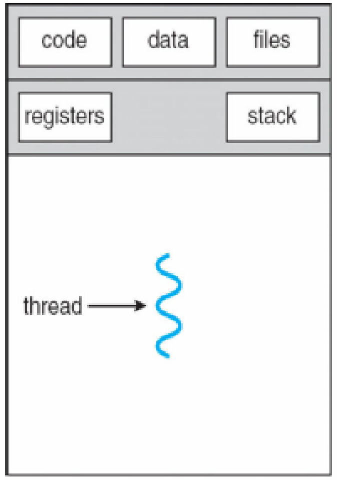


Thread Usage

What is thread: Key Characteristics

- Threads are sometimes referred to as "mini-processes" because they run within the context of a larger process and share its resources like memory, files, and data, but still execute independently.
- The heap is shared among all threads in a process, meaning threads can allocate, access, and modify data stored there.

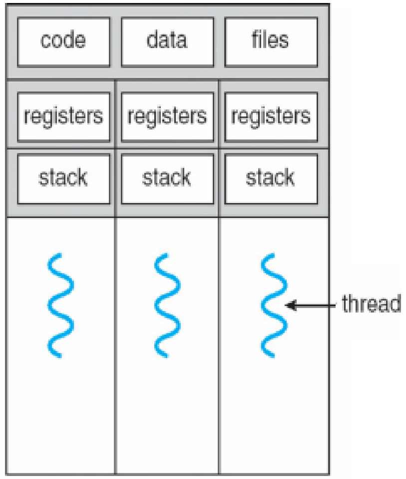
- Each thread has its own:
  - Program Counter (PC): Keeps track of the next instruction to execute.
  - Stack: Stores the local variables, function call information, and return addresses specific to that thread.
  - Registers: The temporary storage locations within the CPU for storing intermediate data.



single-threaded process

Thread Usage
What is thread: Key Characteristics

- **Multithreading** allows a process to perform multiple tasks concurrently by dividing the tasks among multiple threads.



multithreaded process

- Threads, like processes, go through various states:
  - New: When the thread is created.
  - Running: When the thread is executing on the CPU.
  - Blocked: When the thread is waiting for a resource (e.g., I/O or synchronization).
  - Ready: When the thread is ready to run but is waiting for CPU time.
  - Terminated: When the thread has finished execution.

Thread Usage
What is thread: Key Characteristics

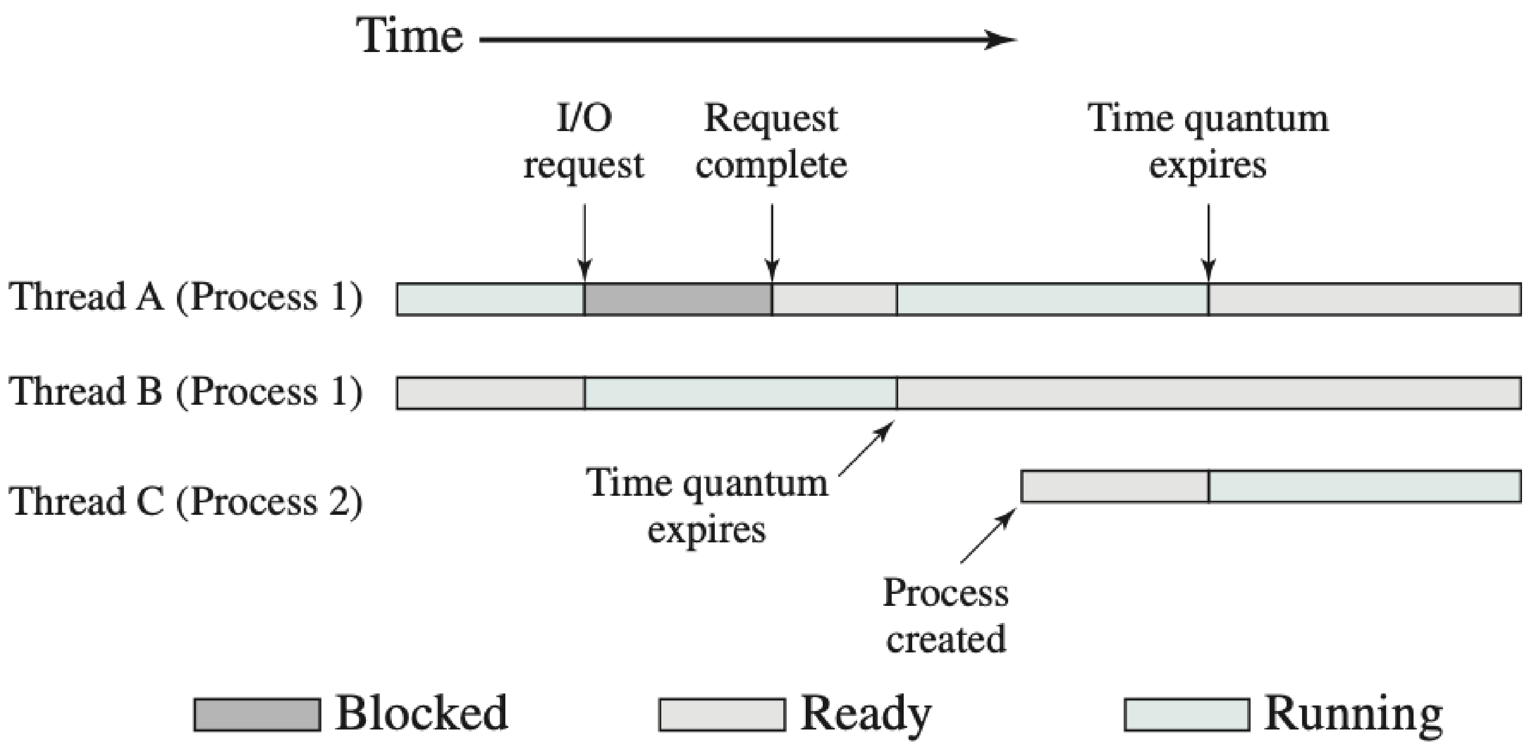


Figure. Multithreading example on a uniprocessor

Thread Usage
What is thread: Key Characteristics

- Since threads share the same memory space, there is a risk of race conditions or data corruption when multiple threads attempt to access or modify the same resource concurrently.
  - To prevent this, synchronization mechanisms are used (will discuss next week)

- Thread Control Block (TCB):Just as processes have a Process Control Block (PCB), threads have a Thread Control Block (TCB) to store the state information specific to a thread, such as its program counter, stack pointer, and register states.

Threads

Threads vs. Process - Definition

Aspect	Process	Thread
Definition	Independent unit of execution with its own memory space.	Lightweight execution unit within a process.
Memory	Has its own memory space.	Shares memory with other threads in the same process.
Resource sharing	Separate resources for each process.	Shares resources with other threads in the process.
Creation overhead	High, as it involves memory allocation and setup.	Low, since it shares resources.
Context switching	Slower due to switching the entire memory space.	Faster, as threads share memory.
Communication	Complex, usually through IPC mechanisms.	Easier, as threads share the same memory.
Isolation	Processes are fully isolated from each other.	Threads are not isolated; bugs can affect the entire process.

Threads

Threads vs. Process

- Example Scenarios
  - **Processes:**
    - If you're running two separate applications like a **web browser** and a **word processor**, they will be running as separate processes.
    - Each has its **own isolated memory space**, ensuring that a crash in one doesn't affect the other.
  - **Threads:**
    - Within a web browser, each tab may run as a separate thread.

Threads

Why Use Thread

- Lower Overhead:
  - Creating, terminating, and switching between threads is more efficient than between processes.
    - This is because threads within the same process share the same address space, which avoids the need to duplicate or manage separate memory spaces.
    - When you create a new process, the operating system needs to allocate and manage a new address space, which is more resource-intensive.
      - In contrast, threads share the same address space, reducing the overhead associated with memory management.

Threads

Why Use Thread

- **Lower Overhead:**
  - **User-Level Threads** have the least overhead for the "null fork" operation because the thread management is done entirely in user space, without requiring kernel intervention.
  - **Kernel-Level Threads** involve more overhead than user-level threads because they require interaction with the kernel to manage the threads, which adds to the processing time.
  - **Processes** incur the highest overhead since creating a new process involves much more work, such as duplicating the entire address space and setting up separate resources, which takes significantly longer.

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Figure: Comparison, in μs (Stallings)

Null fork: the overhead in creating, scheduling, running and terminating a null process/thread
Signal wait: overhead in synchronising threads

Threads

Why Use Thread

- Faster and Easier Inter-Thread Communication:
  - Threads within the same process share the same memory space by default, making communication between threads much faster and simpler compared to inter-process communication (IPC).
  - In IPC, processes usually need to communicate via mechanisms like pipes, sockets, or shared memory segments, which involve more complex setup and slower data transfer due to the need for synchronization and protection between different address spaces.
- Improved performance and responsiveness
  - Multithread helps perform multiple tasks simultaneously within a single application, leading to improved performance and responsiveness.

Threads

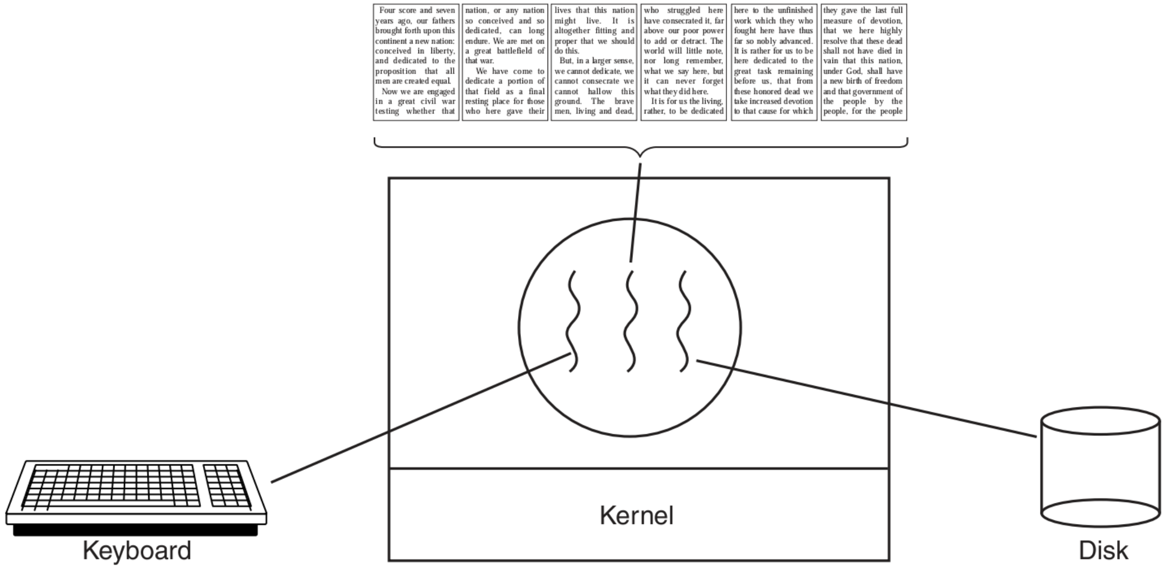
Why Use Thread

- No Protection Boundaries:
  - Threads are designed to work together towards **a common goal** within a single process.
  - Since they belong to the same process and user, they do not require the same level of protection boundaries as separate processes.
  - This lack of protection boundaries within a process reduces the need for system calls that manage these boundaries, resulting in fewer context switches and lower overhead.

Thread Usage

Example 1: Word process

- word processor,
  - One thread might handle user input
  - While another handles background spell-checking,
  - Both running simultaneously within the same application.



- Keyboard -> Thread 1 -> Text Update
- Thread 2 -> Text Formatting
- Thread 3 -> Save to Disk

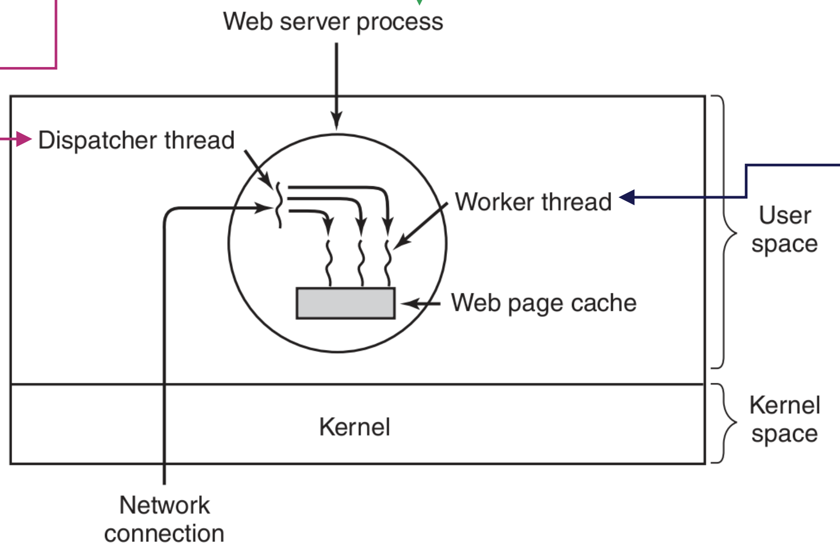
Benefit: Smooth user experience with continuous background processing.

Thread Usage
Example 2: Web Server

The main process running the web server. Consist of multiple threads.

Listens for incoming client requests from the network connection and distributes those requests to available worker threads.

Responsible for processing the client requests that the dispatcher sends them. Handle tasks like fetching a web page from the cache, performing computations, or accessing a database, and then returning the result to the client.



Benefit: Efficient handling of multiple requests, improved server performance.

Thread Usage

Example 2: Web Server(cont.)

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

Dispatcher Thread (a): Continuously listens for new requests and assigns them to worker threads.

Worker Thread (b): Handles the actual work of finding the requested page, first checking the cache and then possibly reading from disk, before returning the response to the client.

Threads

When to Use Thread

- Multiple Related Activities Sharing Resources:
  - When multiple activities or tasks need to access and manipulate the same set of resources (e.g., files, global variables, memory), using threads is advantageous.
    - Example: word processor,

- Handling Multiple Blocking Tasks:
  - In processes where there are multiple tasks that may block (e.g., waiting for I/O, waiting for network data, etc.), using threads allows these tasks to be handled in parallel or concurrently.
    - This improves the responsiveness and efficiency of the application.
      - Example: I/O operations (thread blocks): Other tasks can continue running while one thread waits for the I/O operation to complete.

Threads

When to Use Thread

- Application Examples:
  - **Web Servers:** Web servers can use threads to handle multiple client connections simultaneously, improving scalability and responsiveness.
  - **Make Program:** Build systems like ‘make’ can use threads to compile multiple files concurrently, reducing the overall build time.
  - **Word Processors:** As mentioned earlier, word processors can use threads to perform background tasks (e.g., spell check, autosave) without interrupting the user's typing.
  - **Processing Large Data Volumes:** Applications that process large amounts of data (e.g., data analysis tools, machine learning pipelines) can use threads to parallelize computations, making the processing more efficient and faster.

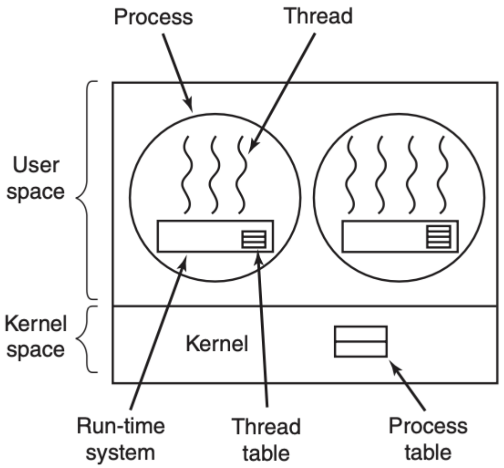
Hyperthreading (Simultaneous Multithreading or SMT)

- Is a hardware-level feature introduced by Intel to improve CPU efficiency.
- Is Intel's proprietary implementation of Simultaneous Multithreading (SMT), which allows each physical CPU core to run two threads (logical processors) concurrently.
- How It Works:
  - A single physical core is treated as two logical cores by the operating system.
  - Each logical core can execute its own thread, but they share the underlying physical resources (e.g., execution units, caches).
  - When one thread on a physical core is stalled (e.g., waiting for data from memory), the other thread can make use of the idle resources, increasing overall CPU throughput.

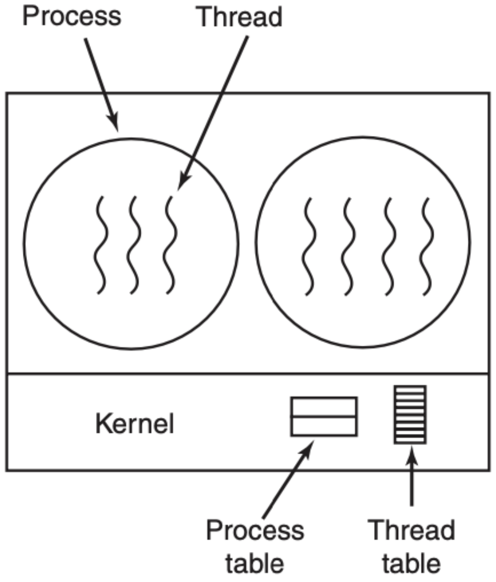
Threads

OS Implementations of Threads

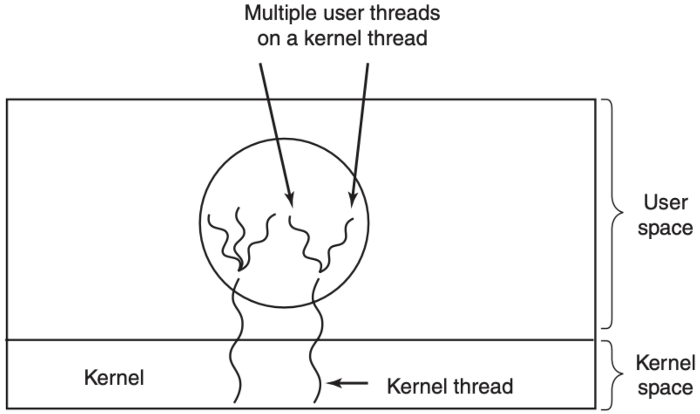
- Thread implementation can occur in three primary ways in modern operating systems: at the user level, the kernel level, or a combination of both in a hybrid model



User threads



Kernel threads



Hybrid implementations

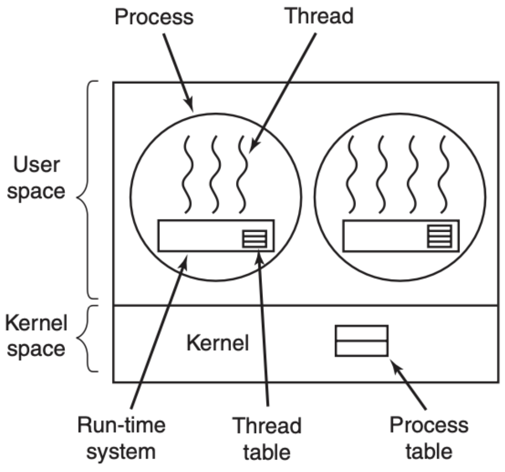
Threads

User-level Threads

- The thread management (creation, scheduling, synchronization) is handled by a thread library in the user space.

- The process maintains a thread table managed by the run-time system without the kernel’s knowledge.

- Example: Many early threading libraries, such as POSIX Pthreads in user-level mode, used this approach.



Threads

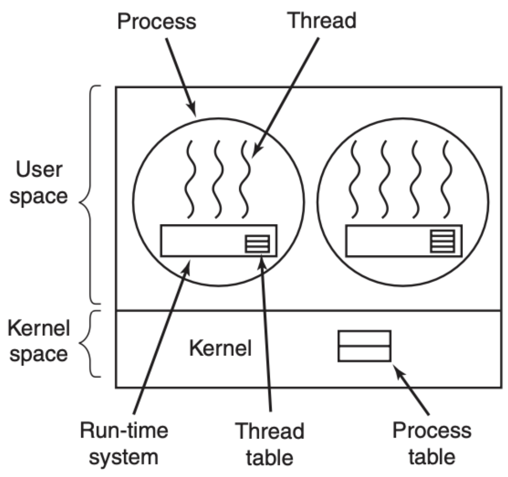
User-level Threads: Advantages

- Efficiency: Context switches between user-level threads are extremely fast because they do not involve the kernel, avoiding the overhead of kernel mode transitions.

- Flexibility: Since the threads are managed in user space, they can be implemented and scheduled in ways that are more suited to the application's needs.
  - Full control over the thread scheduler (e.g. website server)

- Threads are in user space (i.e., no mode switches required)

- Portability: User-level threads can be implemented on operating systems that do not natively support threads.



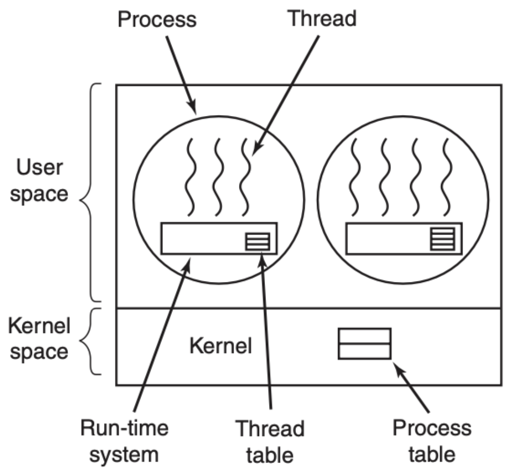
Threads

User-level Threads: Disadvantages

- Blocking: If a user-level thread performs a blocking system call (e.g., reading from a file), the entire process is blocked because the operating system cannot switch to another thread in the process.

- Clock interrupts are non-existent (i.e. user threads are non-preemptive, can not be scheduled round-robin fashion)

- No true parallelism: On multiprocessor systems, user-level threads cannot take advantage of multiple CPUs since the operating system is only aware of one process, not the multiple threads within it.



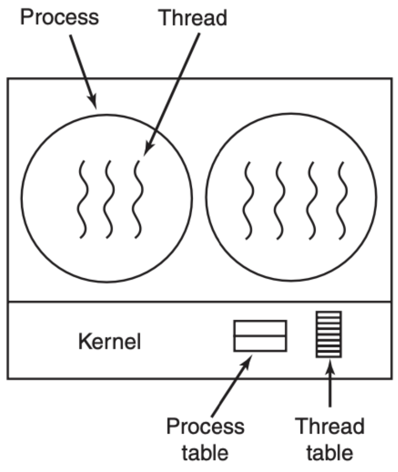
Threads

Kernel-Level Threads

- Managed directly by the operating system kernel.

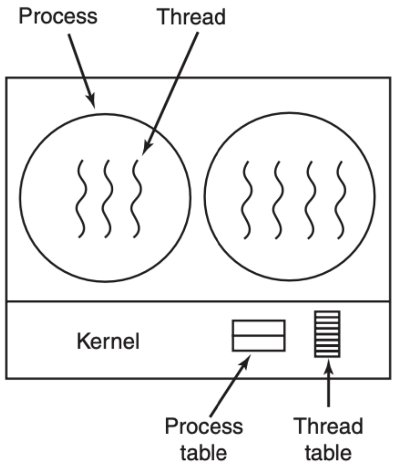
- The kernel is aware of each thread and is responsible for scheduling them and managing their resources.

- User applications interact with threads through an API provided by the operating system, typically through system calls.
  - These system calls allow the application to create, manage, and synchronize threads, with the kernel doing the heavy lifting behind the scenes.
- E.g. Windows, Linux



Threads

Kernel-Level Threads: Advantages



- **True Parallelism:** Kernel-level threads can be scheduled on different processors in a multiprocessor system, allowing true parallel execution.

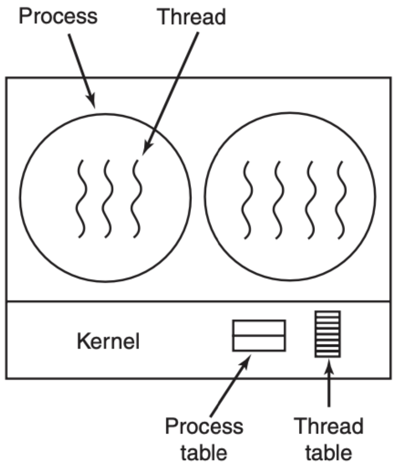
- **Support for Advanced Hardware**:
  - Some CPUs, especially those with **hyperthreading**, have direct hardware support for multithreading. These CPUs can manage multiple hardware threads per core, allowing even greater parallelism and performance.

Threads

Kernel-Level Threads: Advantages

- **No Blocking Issues:** Since the kernel is aware of each thread, if one thread blocks (e.g., due to an I/O operation), the kernel can schedule another thread from the same process to run.

- No Run-Time System Needed:
  - Since the kernel itself manages all aspects of threading, there is no need for a separate run-time system in user space to handle threads.
    - This simplifies application development because developers can rely on the kernel's threading capabilities without needing to implement their own thread management.

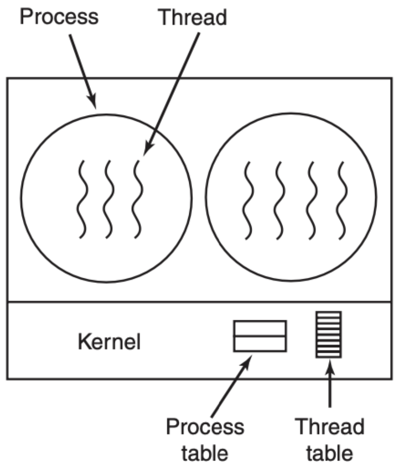


Threads

Kernel-Level Threads: Disadvantages

- **Overhead**: Managing threads in the kernel adds overhead due to the need for mode switching between user space and kernel space, and the complexity of managing each thread.

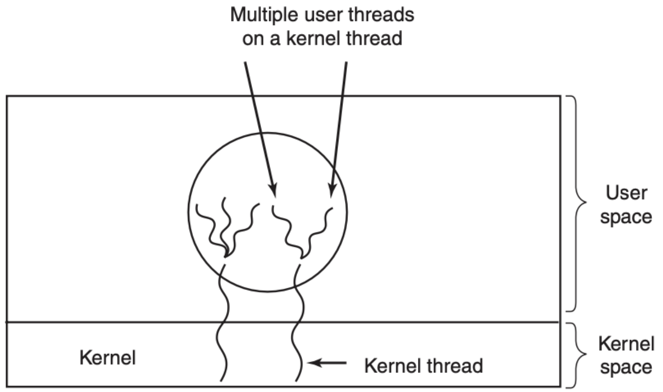
- **Resource-Intensive:** Kernel-level threads require more memory and processing power to manage, as each thread is a full-fledged entity in the kernel's eyes.



Threads

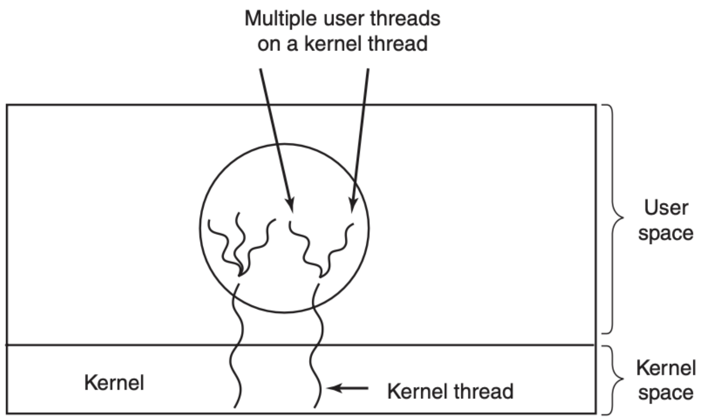
Hybrid Implementations

- Combine aspects of both user-level and kernel-level threading models.
- Typically, the hybrid model maps multiple user-level threads to a smaller or equal number of kernel threads.
- These systems aim to balance the efficiency of user-level threads with the robustness and parallelism capabilities of kernel-level threads.
  - Example: Solaris is an example of an operating system that used this approach, although modern Solaris has shifted to kernel-level threads exclusively.



Threads
Hybrid Implementations: Advantages

- True Parallelism with Flexibility:
  - Allows true parallelism since kernel threads are scheduled by the OS and can run on multiple processors simultaneously.
  - At the same time, the user-level thread library provides flexibility in managing user threads.

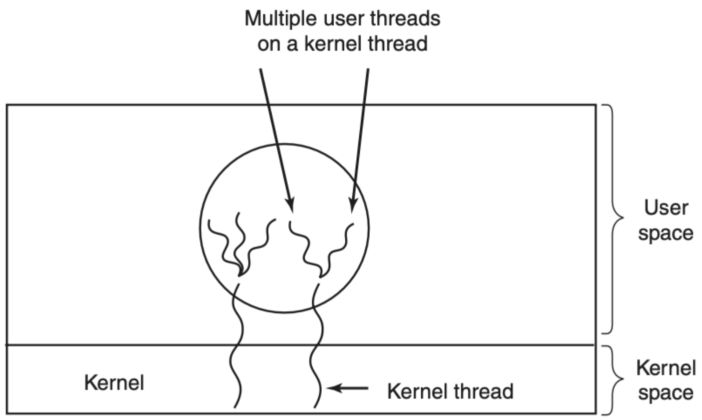


Example: In a multiprocessor system, a web server might have many user threads handling client requests. These user threads are mapped onto fewer kernel threads, which can run in parallel on different processors.

Threads

Hybrid Implementations: Advantages

- Efficient Resource Utilization:
  - Reduce the overhead associated with frequent context switches and system calls in pure kernel-level threading by managing many user threads within fewer kernel threads.
  - This allows for more efficient use of system resources.

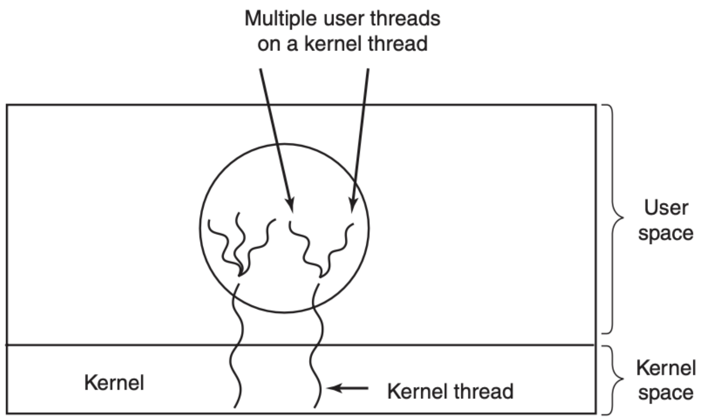


Example: A data processing application can create thousands of user threads for handling different parts of a dataset, but these threads are efficiently multiplexed onto a smaller number of kernel threads, reducing the load on the kernel.

Threads

Hybrid Implementations: Advantages

- Non-Blocking System Calls:
  - If a user-level thread within a kernel thread blocks (e.g., due to an I/O operation), the run-time system can switch to another user thread within the same kernel thread. This allows the process to continue making progress without the entire process being blocked.

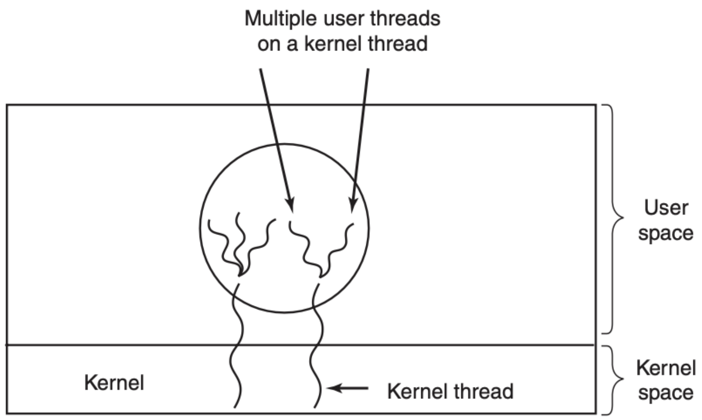


Example: In a file I/O operation, if one user thread blocks while waiting for data, another user thread can be scheduled on the same kernel thread to continue processing, improving overall responsiveness.

Threads

Hybrid Implementations: Disadvantages

- Resource Overhead: While the hybrid model aims to reduce overhead, there is still some resource consumption associated with maintaining both user-level and kernel-level threads.
  - Managing the mapping between the two can also add overhead.

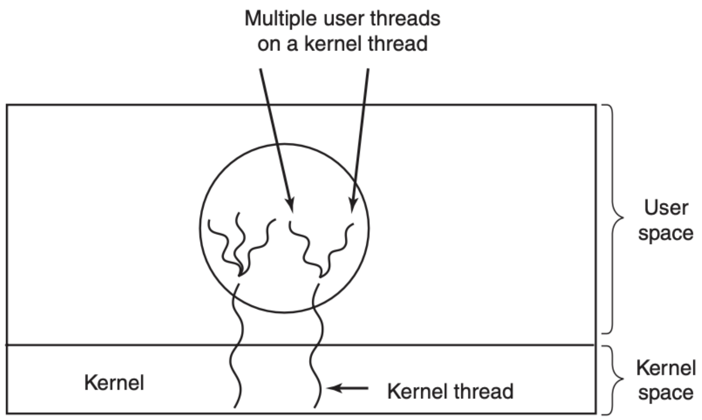


Example: If the mapping between user threads and kernel threads is not optimized, it can lead to inefficient resource usage, where some kernel threads are underutilized while others are overloaded.

Threads

Hybrid Implementations: Disadvantages

- Suboptimal Performance in Certain Scenarios: In some cases, the overhead of multiplexing user threads onto kernel threads can lead to suboptimal performance, especially if the workload is not evenly distributed among the kernel threads.



Example: If an application has many short-lived user threads, the overhead of constantly mapping and unmapping these threads to kernel threads might reduce overall efficiency

- Thread Models
- Thread scheduling
- POSIX Threads (Pthreads) Overview

Thread Models (Implementation Strategies)

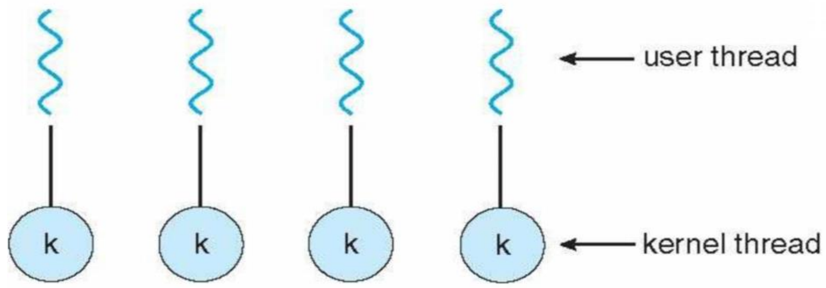
- User-level threads often need to be mapped to kernel-level threads for better performance, scalability, and functionality.
- However, how this mapping is done depends on the threading model.
- There are several models of how user-level threads can be mapped to kernel threads:
  - One to one
  - Many to one
  - Many to Many

Thread Models
One-to-One Model

- In a 1:1 threading model, each user thread maps directly to a single kernel thread.

- This means that when a user thread is created, a corresponding kernel thread is created, and the operating system's kernel is responsible for managing and scheduling these threads.

- This strategy simplifies many aspects of concurrency from the programmer's perspective, as the kernel provides scheduling and resource management.

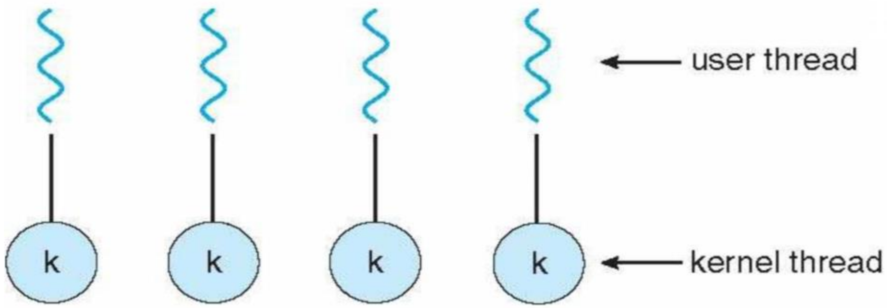


Thread Models
One-to-One Model: Key Features

- **Direct Mapping**: Each user thread corresponds to exactly one kernel thread.

- **Kernel Scheduling**: The operating system kernel takes responsibility for scheduling and managing the threads.

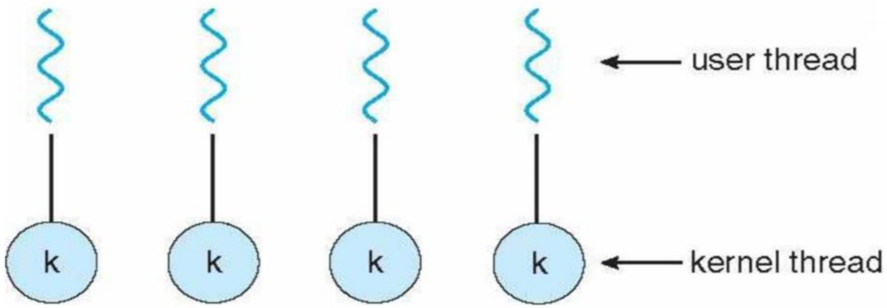
- **Concurrency**: True parallelism can be achieved on multiprocessor systems, as kernel threads can be scheduled on different processors or cores.



Thread Models
One-to-One Model: Key Features

- Blocking: If a thread blocks (e.g., on I/O), only that particular thread is blocked, not the entire process. Other threads can continue to execute.

- Lightweight Process (LWP): In some systems like Solaris, each kernel thread that corresponds to a user thread is called an LWP (Lightweight Process).



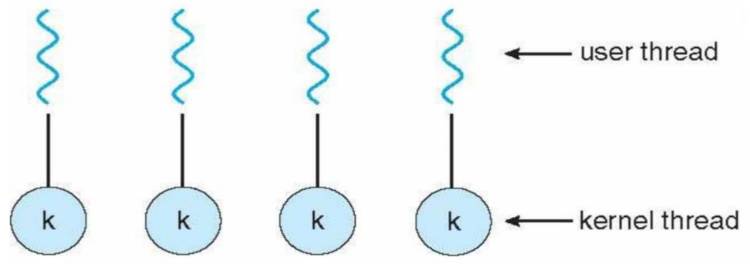
Thread Models
One-to-One Model: Advantages

- **Parallelism**: Kernel threads are managed by the OS and can be run in parallel on multiple processors.

- **Scalability**: Each thread can be scheduled independently by the kernel, allowing for better scalability on systems with multiple processors.

- **Blocking I/O**: When a user thread blocks, only that thread is blocked, not the entire process, which improves responsiveness.

- **Preemption**: Since the kernel schedules threads, preemption between threads is handled efficiently, and the OS can stop and resume threads as needed.

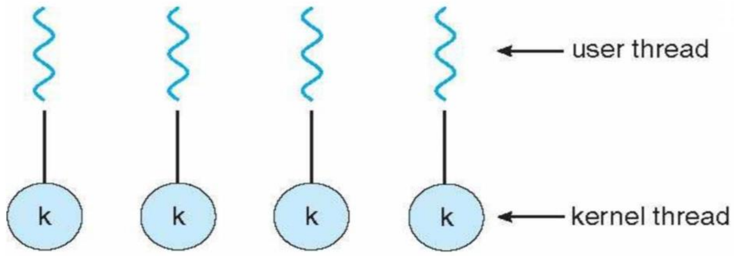


Thread Models
One-to-One Model: Disadvantages

- **Overhead**: Creating and managing a kernel thread has more overhead compared to user-level threads. Each thread requires memory for kernel data structures and incurs context-switching costs.

- **Resource Limits**: Since each user thread maps directly to a kernel thread, there are limits on the number of threads that can be created, as the OS imposes resource constraints (e.g., thread count, memory usage).

- **Complexity in Programming**: While the kernel handles scheduling, the programmer still needs to consider concurrency-related issues like locking, synchronization, and data sharing between threads.



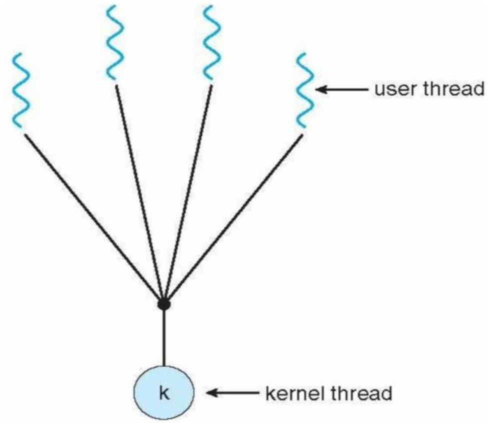
Thread Models
Many-to-One Model:

- All user-level threads are mapped to a single kernel thread.

- The operating system only manages one kernel thread per process, while all user threads are handled by a user-level thread library in user space.

- No parallelism: Since there is only one kernel thread, only one user-level thread can be executed at any time, even on multi-core systems.

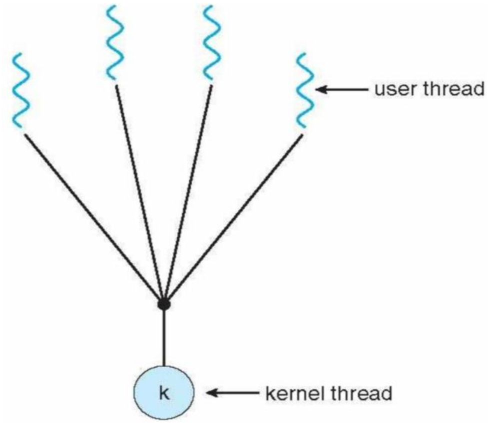
- User-space scheduling: The user-level thread library handles the scheduling and switching between threads, without kernel involvement.
- Example:
    - GNU Portable Threads (Pth) library.



Thread Models
Many-to-One Model: Advantages

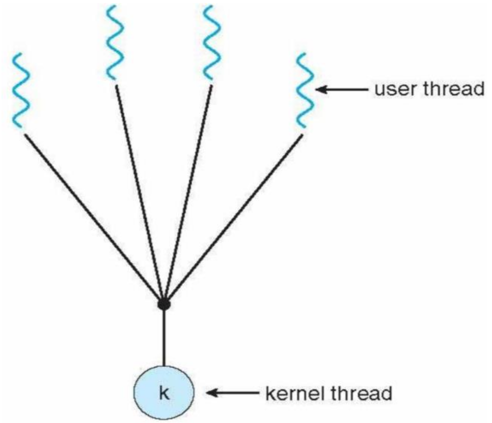
- Low overhead: Since all thread operations (creation, switching, scheduling) happen in user space, there are no system calls required, making the operations fast.

- Custom scheduling: Applications can implement custom thread scheduling algorithms.



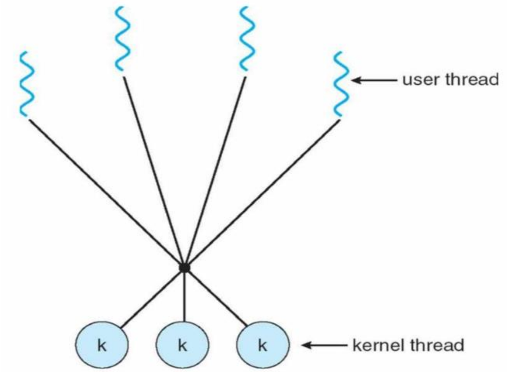
Thread Models
Many-to-One Model: Disadvantages

- Blocking problem: If a user-level thread makes a blocking system call (e.g., I/O), the entire process is blocked.
- No parallelism: Even on multi-core processors, only one user-level thread can execute at a time, severely limiting performance in CPU-bound applications.
- Cooperative scheduling: Threads need to yield control voluntarily, which can lead to problems if one thread monopolizes the CPU.



Thread Models
Many-to-Many Model:

- Multiple user threads are mapped to a smaller or equal number of kernel threads.
- Combines the advantages of both previous models and avoids their drawbacks.
- Characteristics:
  - M user threads share N kernel threads, where M ≥ N.
  - User-space scheduling and kernel support: User threads are scheduled by the user-level thread library, but kernel threads are scheduled by the operating system.
  - Parallelism: Multiple user threads can execute in parallel on different processors, but the number of kernel threads limits how many can run simultaneously.
  - Non-blocking: If a user thread makes a blocking system call, the kernel can schedule another user thread on the same kernel thread.
- Example:
  - Earlier versions of Solaris:
  - GNU Portable Threads (Pth) can be extended to support Many to many threading in some contexts.

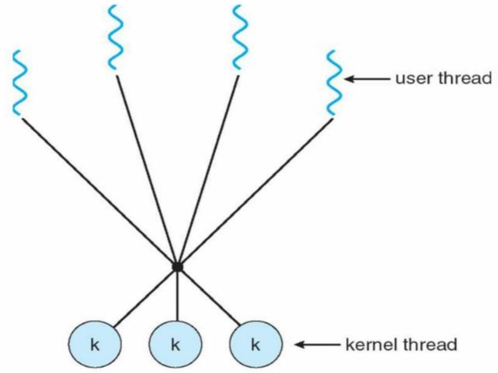


Thread Models
Many-to-Many Model: Advantages

- Better resource utilization: By using fewer kernel threads, this model reduces the overhead of creating kernel threads while still allowing user threads to be executed in parallel.

- Flexible scheduling: The user-level thread library can manage scheduling policies, and the operating system handles kernel thread scheduling.

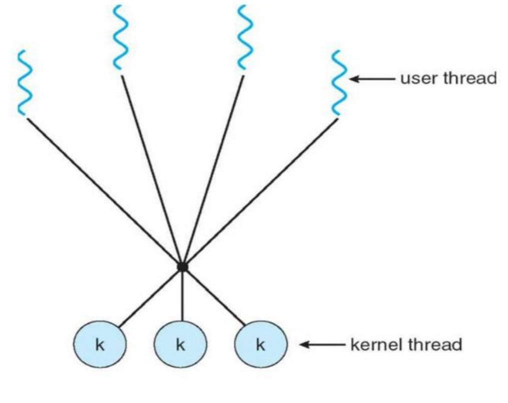
- Avoids blocking issues: Since multiple kernel threads exist, blocking system calls in one thread don't block the entire process.



Thread Models
Many-to-Many Model: Disadvantages

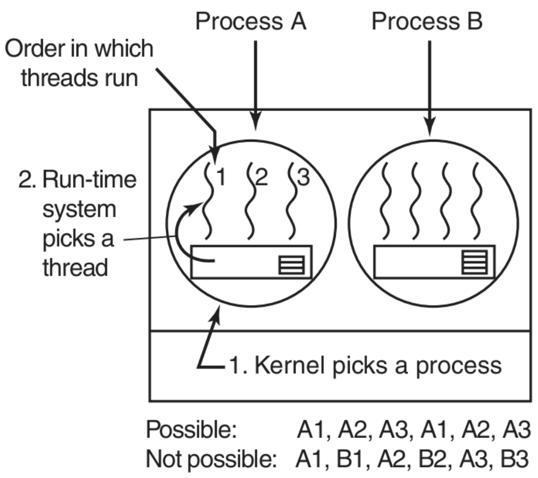
- Complexity: Managing the interaction between user-level threads and kernel threads adds complexity to the thread library.

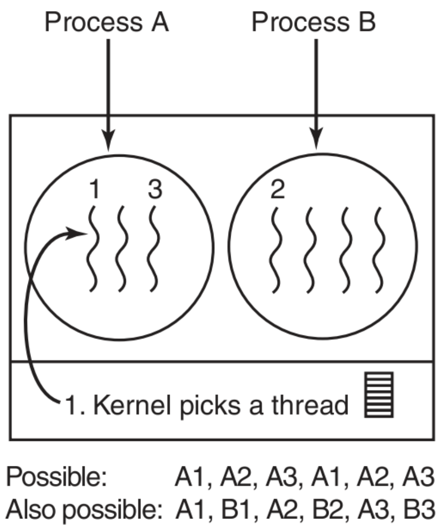
- Load balancing: The library needs to manage the distribution of user threads onto kernel threads efficiently. If not managed properly, this could result in bottlenecks or poor performance.



Thread scheduling

- Use the same algorithms as **process scheduling.**
- Occur at both the kernel level and the user level, depending on the threading model used.
- In kernel-level threading, the operating system scheduler directly handles thread management.
- User-level threading, a user-space library (run-time system) handles scheduling with limited or no direct involvement from the OS.





APIs across different operating systems

- Windows:
  - Windows threads are often managed via APIs like CreateThread() or higher-level thread management using ThreadPool.

- Solaris/Illumos (Lightweight Process):
  - In Solaris (also known as Illumos in later versions), the 1:1 threading model is implemented using Lightweight Processes (LWPs).

- POSIX Threads (PTHREAD_SCOPE_SYSTEM):
  - This model is widely used in UNIX-like systems, including Linux, where the pthread_create() function creates a new thread that is scheduled by the kernel.

POSIX Threads (Pthreads) Overview

- POSIX Threads (Pthreads): A standard threading API for C/C++ in POSIX-compliant systems.

- Functions in Pthreads:
  - pthread_create(): Create a new thread.
  - pthread_join(): Wait for a thread to finish execution.
  - pthread_exit(): Terminate a thread.
  - pthread_cancel(): Request the cancellation of a thread.

Creating Threads:
Example: Creating a thread

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

- thread: Pointer to store the thread ID.
- attr: Thread attributes such as Scheduling policy and priorityStack size (can be NULL for default attributes).
- start_routine: Function the thread will execute.
- arg: Argument passed to the thread function.

- Example: Creating a thread

```
pthread_t thread;
int arg = 10;
pthread_create(&thread, NULL, thread_function, &arg);
```

Joining Threads

- pthread_join():
  - Waits for a specific thread to finish its execution.
  - Used to ensure that the main thread waits for all child threads to complete.
- Function prototype:

```
int pthread_join(pthread_t thread, void **retval);
```

  - thread: The thread to wait for.
  - retval: Pointer to store the return value of the thread (can be NULL).
- Example

```
pthread_join(thread, NULL);
```

Thread Management Example
Simple Thread Creation Example

```c
#include <pthread.h>
#include <stdio.h>

void* thread_function(void* arg) {
    printf("Thread function received value: %d\n", *(int*)arg);
    return NULL;
}

int main() {
    pthread_t thread;
    int value = 5;

    // Create a new thread
    pthread_create(&thread, NULL, thread_function, &value);

    // Wait for the thread to finish
    pthread_join(thread, NULL);

    printf("Main thread finished\n");
    return 0;
}
```

```
Thread function received value: 5
Main thread finished
```

Thread Termination

- Threads can terminate in three ways:
    - Return from the start routine: Normal thread completion.
    - Calling pthread_exit(): Explicit thread termination.
    - Cancellation via pthread_cancel(): One thread can request cancellation of another.
    - pthread_exit():Terminates the calling thread and can pass a return value to pthread_join().

- pthread_exit():
    - Terminates the calling thread and can pass a return value to pthread_join().
    - Example:

```
pthread_exit(NULL);
```

Managing Multiple Threads

- You can create multiple threads in the same program.
- Each thread performs its own independent task, but may access shared resources.
- Example: Multiple thread creation

```c
pthread_t threads[5];
for (int i = 0; i < 5; i++) {
    pthread_create(&threads[i], NULL, thread_function, &i);
}
for (int i = 0; i < 5; i++) {
    pthread_join(threads[i], NULL);
}
```

Summary

Take Home Message

- What are **threads** and why are they useful
- Different **thread** implementations from an OS perspective
- The principle/idea behind **PThreads**