Operating Systems Notes - Chapter 4

June 9, 2025

Contents

1 Threads & Concurrency

1.1 Overview

- A **thread** is a basic unit of CPU utilization, comprising a thread ID, program counter (PC), register set, and stack.
- Threads share with other threads of the same process: code section, data section, and OS resources (e.g., open files, signals).
- A traditional process has a single thread of control; a multithreaded process can perform multiple tasks concurrently.
- Modern operating systems provide features for processes to contain multiple threads of control.
- Identifying opportunities for parallelism using threads is crucial for modern multicore systems.

1.2 Motivation

- Most modern software applications are multithreaded (e.g., web browsers, word processors, image thumbnail generators).
- Multithreaded applications can leverage multicore systems for parallel CPU-intensive tasks.
- **Web Server Example**:
  - A busy web server may handle thousands of concurrent client requests.
  - Single-threaded server: services one client at a time, leading to long wait times.
  - Process-creation method: server creates a separate process for each request (time-consuming, resource-intensive).
  - Multithreaded server: creates a new thread for each request, more efficient.
- Most OS kernels are multithreaded (e.g., Linux kernel threads like `kthreadd` for device management, memory management, interrupt handling).
- Multithreading is beneficial for CPU-intensive problems in data mining, graphics, and AI.

1.3 Benefits

- **Responsiveness**:
  - Allows an application to continue running even if part is blocked or performing a lengthy operation.
  - Increases responsiveness to the user (e.g., UI remains active during a time-consuming background task).
- **Resource Sharing**:
  - Threads share memory and resources of their parent process by default.
  - More efficient than inter-process communication (shared memory, message passing).
  - Allows multiple threads of activity within the same address space.
- **Economy**:
  - More economical to create and context-switch threads than processes.
  - Thread creation consumes less time and memory than process creation.
  - Context switching is typically faster between threads.
- **Scalability**:
  - Greater benefits in multiprocessor architectures, where threads can run in parallel on different processing cores.
  - A single-threaded process runs on only one processor, regardless of available cores.

Section glossary

Term	Definition
single-threaded	A process or program that has only one thread of control (and so executes on only one core at a time).
multithreaded	A term describing a process or program with multiple threads of control, allowing multiple simultaneous execution points.
thread	A basic unit of CPU utilization; it comprises a thread ID, a program counter (PC), a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

2 Multicore Programming

2.1 Introduction to Multicore Systems

- Modern systems place multiple computing cores on a single processing chip, appearing as separate CPUs to the operating system. These are referred to as **multicore** systems.
- Multithreaded programming enables more efficient use of these multiple computing cores and improves concurrency.
- **Concurrency vs. Parallelism**:
  - **Concurrent system**: Supports more than one task by allowing all tasks to make progress (e.g., interleaving execution on a single-core system).
  - **Parallel system**: Can perform more than one task simultaneously (e.g., assigning separate threads to each core on a multicore system).
  - It is possible to have concurrency without parallelism.

2.2 Programming Challenges

- The trend towards multicore systems requires system designers and application programmers to better utilize multiple computing cores.
- Operating system designers must create scheduling algorithms for parallel execution.
- Application programmers must modify existing programs and design new ones to be multithreaded.
- Five key challenges in programming for multicore systems:
  1. **Identifying tasks**: Finding areas in applications that can be divided into separate, concurrent, and ideally independent tasks.
  2. **Balance**: Ensuring that parallel tasks perform equal work of equal value to justify the use of separate execution cores.
  3. **Data splitting**: Dividing the data accessed and manipulated by tasks to run on separate cores.
  4. **Data dependency**: Examining data for dependencies between tasks and ensuring synchronized execution to accommodate these dependencies.
  5. **Testing and debugging**: More difficult due to many possible execution paths in parallel programs.

2.3 Amdahl's Law

- A formula that identifies potential performance gains from adding additional computing cores.
- If S is the portion of the application that must be performed serially on a system with N processing cores, the formula is:

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- **Example**: An application that is 75% parallel and 25% serial ($S = 0.25$):
  - On a 2-core system ($N = 2$): speedup ≈ 1.6 times.
  - On a 4-core system ($N = 4$): speedup ≈ 2.28 times.
- As N approaches infinity, the speedup converges to $1/S$.
- The serial portion of an application can disproportionately affect the performance gained by adding computing cores.

2.4 Types of Parallelism

- **Data parallelism**:
  - Focuses on distributing subsets of the same data across multiple computing cores.
  - Performs the same operation on each core.
  - **Example**: Summing elements of an array by dividing the array into subsets for different threads.
- **Task parallelism**:
  - Distributes tasks (threads) across multiple computing cores.
  - Each thread performs a unique operation.
  - Threads may operate on the same or different data.
  - **Example**: Two threads performing unique statistical operations on the same array.
- Data and task parallelism are not mutually exclusive; applications may use a hybrid approach.

Section glossary

Term	Definition
multicore	Multiple processing cores within the same CPU chip or within a single system.
data parallelism	A computing method that distributes subsets of the same data across multiple cores and performs the same operation on each core.
task parallelism	A computing method that distributes tasks (threads) across multiple computing cores, with each task performing a unique operation.

3 Multithreading Models

3.1 User Threads vs. Kernel Threads

- Support for threads can be provided at the user level (**user threads**) or by the kernel (**kernel threads**).
- **User threads**: Supported above the kernel and managed without kernel support.
- **Kernel threads**: Supported and managed directly by the operating system.
- Most contemporary operating systems (Windows, Linux, macOS) support kernel threads.
- A relationship must exist between user threads and kernel threads.

3.2 Many-to-One Model

- Maps many user-level threads to one kernel thread.
- Thread management is done by the thread library in user space, making it efficient.
- **Drawbacks**:
  - The entire process blocks if a user thread makes a blocking system call.
  - Multiple user threads cannot run in parallel on multicore systems because only one kernel thread can access the kernel at a time.
- **Example**: Green threads (Solaris, early Java).
- Rarely used now due to inability to leverage multicore systems.

3.3 One-to-One Model

- Maps each user thread to a kernel thread.
- Provides more concurrency than the many-to-one model:
  - Another thread can run when a thread makes a blocking system call.
  - Allows multiple threads to run in parallel on multiprocessors.
- **Drawback**: Creating a user thread requires creating a corresponding kernel thread, which can burden system performance if too many kernel threads are created.
- **Implementations**: Linux, Windows operating systems.
- Most operating systems currently use this model.

3.4 Many-to-Many Model

- Multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The number of kernel threads can be specific to the application or machine (e.g., more kernel threads on a system with more cores).
- **Advantages**:
  - Developers can create as many user threads as needed.
  - Corresponding kernel threads can run in parallel on a multiprocessor.
  - When a thread performs a blocking system call, the kernel can schedule another thread.
- **Two-level model**: A variation where many user-level threads are multiplexed to a smaller or equal number of kernel threads, but also allows a user-level thread to be bound to a kernel thread.
- Difficult to implement in practice.
- Less common now as limiting kernel threads is less important with increasing core counts, but some contemporary concurrency libraries still use this model.

Section glossary

Term	Definition
user thread	A thread running in user mode, managed without kernel support.
kernel threads	Threads running in kernel mode, supported and managed directly by the operating system.
many-to-one model	A multithreading model that maps many user-level threads to one kernel thread.
one-to-one model	A multithreading model that maps each user thread to a kernel thread.
many-to-many model	A multithreading model that multiplexes many user-level threads to a smaller or equal number of kernel threads.
two-level model	A variation of the many-to-many model that also allows a user-level thread to be bound to a kernel thread.

4 Thread Libraries

4.1 Introduction to Thread Libraries

- A **thread library** provides an API for creating and managing threads.
- **Implementation Approaches**:
  - **User-space library**: Entirely in user space, no kernel support. Function calls are local, not system calls.
  - **Kernel-level library**: Supported directly by the operating system. Function calls typically result in system calls to the kernel.
- **Main Thread Libraries**: POSIX Pthreads, Windows, and Java.
- **Relationship to Host OS**:
  - Pthreads: May be user-level or kernel-level; common on UNIX-type systems (Linux, macOS).
  - Windows thread library: Kernel-level library on Windows systems.
  - Java thread API: Implemented using the host system's thread library (e.g., Windows API on Windows, Pthreads on UNIX/Linux/macOS).
- **Data Sharing**:
  - POSIX and Windows: Global data (declared outside functions) are shared among all threads in the same process.
  - Java: No equivalent of global data; shared data access must be explicitly arranged.
- **Summation Example**: Illustrative example for thread creation, calculating the summation of a non-negative integer N:

$$sum = \sum_{t=1}^{N} i$$

- **Threading Strategies**:
  - **Asynchronous threading**: Parent creates child thread and resumes execution independently. Little data sharing. Used in multithreaded servers and responsive UIs.
  - **Synchronous threading**: Parent creates child threads and waits for all children to terminate before resuming. Involves significant data sharing (e.g., parent combining results from children). All examples in this section use synchronous threading.

4.2 Pthreads

- **Pthreads**: POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. It is a specification, not an implementation.
- Implemented on most UNIX-type systems (Linux, macOS).
- **Basic API (C program example)**:
  - Include `pthread.h`.
  - Declare thread identifier (`pthread_t tid`) and attributes (`pthread_attr_t attr`).
  - Initialize attributes: `pthread_attr_init(&attr)`.
  - Create thread: `pthread_create(&tid, &attr, runner, argv[1])`.
    - `tid`: thread identifier.
    - `attr`: thread attributes.
    - `runner`: function where new thread begins execution.
    - `argv[1]`: parameter passed to the thread function.
  - Parent waits for child thread to terminate: `pthread_join(tid, NULL)`.
  - Child thread terminates: `pthread_exit(0)`.
  - Shared data (e.g., `sum`) is declared globally.
- Waiting for multiple threads: Use a `for` loop with `pthread_join()`.

4.3 Windows Threads

- Similar to Pthreads in technique.
- Include `windows.h`.
- Shared data (e.g., `Sum`) is declared globally (e.g., `DWORD` for unsigned 32-bit integer).
- Thread function (e.g., `Summation`) is passed a pointer to `void` (LPVOID).
- **Thread Creation**: Use `CreateThread()` function.
  - Parameters include security attributes, stack size, thread function, parameter to thread function, creation flags, and thread identifier.
  - Default attributes typically make the thread eligible to run immediately.
- **Waiting for Thread Completion**:

    - For a single thread: `WaitForSingleObject(ThreadHandle, INFINITE)`.
    - For multiple threads: `WaitForMultipleObjects(N, THandles, TRUE, INFINITE)`.

4.4 Java Threads

- Fundamental model of program execution in Java; available on any system with a JVM.
- All Java programs have at least one thread (the `main()` method).
- **Techniques for explicit thread creation**:
    1. Create a class derived from `Thread` and override its `run()` method.
    2. Define a class that implements the `Runnable` interface (more common).
        - `Runnable` defines `public void run()`. Code in this method executes in a separate thread.
- **Lambda Expressions (Java 1.8+)**: Provide cleaner syntax for creating threads by using a lambda expression instead of a separate class implementing `Runnable`.
- **Thread Execution**:
    - Create a `Thread` object, passing an instance of a class that implements `Runnable`.
    - Invoke the `start()` method on the `Thread` object.
        - Allocates memory and initializes a new thread in the JVM.
        - Calls the `run()` method (do not call `run()` directly).
- **Waiting for Thread Completion**: Use the `join()` method (can throw `InterruptedException`).
- **Java Executor Framework** (`java.util.concurrent`):
    - Provides greater control over thread creation and communication.
    - Based on the `Executor` interface, which defines `void execute(Runnable command)`.
    - Separates thread creation from execution.
    - **Returning Results**:
        - Java threads cannot return results directly from `run()`.
        - The `Callable` interface (similar to `Runnable`) allows returning a result.
        - Results are returned as `Future` objects, retrieved using the `get()` method.
    - This framework offers benefits like returning results and separating thread creation from result retrieval, and can be combined with other features for robust thread management.

Section glossary

Term	Definition
thread library	A programming library that provides programmers with an API for creating and managing threads.
asynchronous threading	Threading in which a parent creates a child thread and then resumes execution, so that the parent and child execute concurrently and independently of one another.
synchronous threading	Threading in which a parent thread creating one or more child threads waits for them to terminate before it resumes.
Pthreads	The POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization (a specification for thread behavior, not an implementation).
closure	In functional programming languages, a construct to provide a simple syntax for parallel applications (also known as Lambda expressions in Java).

5 Implicit Threading

5.1 Introduction to Implicit Threading

- Designing applications with many threads (hundreds or thousands) is complex, involving challenges related to correctness (e.g., synchronization, deadlocks).
- **Implicit threading**: A strategy to address these difficulties by transferring thread creation and management from application developers to compilers and run-time libraries.
- Developers identify **tasks** (functions) that can run in parallel, and the run-time library maps these tasks to separate threads, typically using the many-to-many model.
- **Advantage**: Developers focus on identifying parallelizable tasks, while libraries handle the details of thread creation and management.

5.2 Thread Pools

- A **thread pool** involves creating a number of threads at startup and placing them into a pool, where they wait for work.
- **Problem with creating a new thread per request (e.g., web server)**:
  - Time and overhead to create and discard threads for each request.
  - Unbounded number of concurrent threads can exhaust system resources (CPU, memory).
- **How thread pools work**:
  - Server receives request, submits it to the thread pool.
  - An available thread from the pool is awakened to service the request immediately.
  - If no thread is available, the task is queued.
  - Once a thread completes its work, it returns to the pool.
  - Works well for asynchronously executed tasks.
- **Benefits of thread pools**:
  1. Faster request servicing with existing threads.
  2. Limits the number of concurrent threads, preventing resource exhaustion.
  3. Separates task definition from thread creation mechanics, allowing flexible scheduling (e.g., delayed or periodic execution).
- Number of threads in the pool can be set heuristically or dynamically adjusted based on system load.
- **Windows Thread Pool API**:
  - Functions like `QueueUserWorkItem()` submit a function to be executed by a thread from the pool.
  - Parameters include a pointer to the function, a parameter for the function, and flags for thread management.
- **Java Thread Pools (`java.util.concurrent`)**:
  - Provides various thread pool architectures:
    - `newSingleThreadExecutor()`: Pool of size 1.
    - `newFixedThreadPool(int size)`: Pool with a specified number of threads.
    - `newCachedThreadPool()`: Unbounded pool that reuses threads.
  - Created using factory methods in the `Executors` class, returning an `ExecutorService` object.
  - `ExecutorService` extends `Executor` (with `execute()` method) and provides methods for managing pool termination (`shutdown()`).

5.3 Fork-Join

- The **fork-join** model is a synchronous threading strategy where a parent thread creates (forks) child threads and waits for them to terminate and join with it to combine results.
- An excellent candidate for implicit threading: parallel tasks are designated, and a library manages thread creation and task assignment.
- Similar to a synchronous version of thread pools, where the library determines the number of threads.
- **Java Fork-Join Library (Java 1.7+)**:
  - Designed for recursive divide-and-conquer algorithms (e.g., Quicksort, Mergesort).
  - Tasks are forked during the divide step, assigned smaller subsets of the problem, and execute concurrently.
  - Problem is solved directly when small enough (below a `THRESHOLD`).
  - Uses `ForkJoinPool` to manage worker threads.
  - Tasks inherit from `ForkJoinTask` (abstract base class), which is extended by:
    - `RecursiveTask`: Returns a result via its `compute()` method.
    - `RecursiveAction`: Does not return a result (void `compute()`).
  - `fork()` method creates new tasks; `join()` method blocks until a task completes and returns its result.

  - **Work Stealing**: Each thread in a `ForkJoinPool` maintains a queue of forked tasks; if a queue is empty, it can "steal" a task from another thread's queue to balance workload.

5.4 OpenMP

- A set of compiler directives and an API for C, C++, or FORTRAN, supporting parallel programming in shared-memory environments.
- Identifies **parallel regions** as blocks of code that may run in parallel.
- Developers insert compiler directives (e.g., `#pragma omp parallel`) to instruct the OpenMP run-time library to execute regions in parallel.
- When `#pragma omp parallel` is encountered, OpenMP creates as many threads as there are processing cores in the system, and all threads execute the parallel region simultaneously. Threads are terminated upon exiting the region.
- Provides directives for parallelizing loops (e.g., `#pragma omp parallel for`) to divide work among threads.
- Allows manual control over the number of threads and identification of shared vs. private data.
- Available on various compilers for Linux, Windows, and macOS.

5.5 Grand Central Dispatch (GCD)

- Developed by Apple for macOS and iOS.
- A combination of a run-time library, API, and language extensions for identifying code sections (tasks) to run in parallel.
- Manages most threading details.
- Schedules tasks by placing them on a **dispatch queue**. Tasks are assigned to available threads from a managed pool.
- **Types of Dispatch Queues**:
  - **Serial queues**: Tasks removed in FIFO order; one task completes before the next is removed. Each process has a **main queue** (private dispatch queue); developers can create additional local serial queues. Useful for sequential execution.
  - **Concurrent queues**: Tasks removed in FIFO order, but multiple tasks can execute in parallel. System-wide concurrent queues (global dispatch queues) are divided by quality-of-service classes:
    - **QOS_CLASS_USER_INTERACTIVE**: User interface and event handling (small amount of work, responsive UI).
    - **QOS_CLASS_USER_INITIATED**: User-associated tasks requiring longer processing (e.g., opening files/URLs); must complete for user interaction to continue.
    - **QOS_CLASS_UTILITY**: Longer-running tasks not requiring immediate results (e.g., importing data).
    - **QOS_CLASS_BACKGROUND**: Non-visible, non-time-sensitive tasks (e.g., indexing, backups).
- **Task Submission**:
  - C, C++, Objective-C: Use a **block** (self-contained unit of work, specified by `^{}`).
  - Swift: Use a **closure** (similar to a block, self-contained unit of functionality).
  - Tasks submitted using functions like `dispatch_async()`.
- Internally, GCD's thread pool uses POSIX threads and dynamically adjusts thread count. Implemented by `libdispatch`.

5.6 Intel Threading Building Blocks (TBB)

- A template library for designing parallel C++ applications.
- Requires no special compiler or language support.
- Developers specify tasks, and the TBB task scheduler maps them to underlying threads.
- Provides load balancing and is cache-aware (prioritizes tasks with data in cache).
- Offers templates for parallel loop structures (`parallel_for`), atomic operations, mutual exclusion locking, and concurrent data structures.
- **Parallel For Loops**:
  - `parallel_for (range, body)`: `range` defines the iteration space, `body` specifies an operation on a subrange.
  - Example: `parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});`
  - TBB divides loop iterations into "chunks" and assigns tasks to available threads.
  - Developers identify parallelizable operations, and the library manages work division and parallel execution.
- Available in commercial and open-source versions for Windows, Linux, and macOS.

Section glossary

Term	Definition
implicit threading	A programming model that transfers the creation and management of threading from application developers to compilers and run-time libraries.
thread pool	A number of threads created at process startup and placed in a pool, where they sit and wait for work.
fork-join	A strategy for thread creation in which the main parent thread creates (forks) one or more child threads and then waits for the children to terminate and join with it.
parallel regions	Blocks of code that may run in parallel, identified by OpenMP.
dispatch queue	An Apple OS feature for parallelizing code; blocks are placed in the queue by Grand Central Dispatch (GCD) and removed to be run by an available thread.
main queue	Apple OS per-process serial dispatch queue.
user-interactive	In the Grand Central Dispatch Apple OS scheduler, the scheduling class representing tasks that interact with the user.
user-initiated	In the Grand Central Dispatch Apple OS scheduler, the scheduling class representing tasks that interact with the user but need longer processing times than user-interactive tasks.
utility	In the Grand Central Dispatch Apple OS scheduler, the scheduling class representing tasks that require a longer time to complete but do not demand immediate results.
background	In the Grand Central Dispatch Apple OS scheduler, the scheduling class representing tasks that are not time sensitive and are not visible to the user.
block	In the Grand Central Dispatch Apple OS scheduler, a language extension (for C, C++, Objective-C) that allows designation of a self-contained unit of work to be submitted to dispatch queues.
iteration space	In Intel Threading Building Blocks, the range of elements that will be iterated.

6 Threading Issues

6.1 The `fork()` and `exec()` System Calls

- The semantics of `fork()` and `exec()` change in multithreaded programs.
- `fork()` considerations:
  - Some UNIX systems offer two versions of `fork()`:
    - Duplicates all threads in the new process.
    - Duplicates only the calling thread in the new process.
  - Choice depends on application:
    - If `exec()` is called immediately after forking, duplicating only the calling thread is appropriate.
    - If the new process does not call `exec()`, duplicating all threads is appropriate.
- `exec()` behavior:
  - If a thread invokes `exec()`, the specified program replaces the entire process, including all threads.

6.2 Signal Handling

- A **signal** in UNIX systems notifies a process of an event.
- Signals can be synchronous or asynchronous.
- **Signal Pattern**: Generated → Delivered → Handled.
- **Synchronous Signals**: Generated by an event within the running process (e.g., illegal memory access, division by zero); delivered to the same process that caused the signal.
- **Asynchronous Signals**: Generated by an event external to the running process (e.g., `<control><C>`, timer expiration); typically sent to another process.
- **Signal Handlers**:
  - **Default signal handler**: Kernel-provided handler for each signal.
  - **User-defined signal handler**: Overrides the default action to handle the signal.
- **Signal Delivery in Multithreaded Programs**: More complex than single-threaded. Options include:
  1. Deliver to the thread to which the signal applies.
  2. Deliver to every thread in the process.
  3. Deliver to certain threads in the process.
  4. Assign a specific thread to receive all signals for the process.
- Synchronous signals are delivered to the causing thread. Asynchronous signals may be sent to all threads (e.g., process termination) or to the first non-blocking thread.
- **UNIX Functions**:
  - `kill(pid_t pid, int signal)`: Delivers a signal to a specified process.
  - `pthread_kill(pthread_t tid, int signal)`: Delivers a signal to a specified Pthread.
- **Windows Emulation**: Windows does not explicitly support signals but emulates them using **asynchronous procedure calls** (APCs). An APC is delivered to a particular thread.

6.3 Thread Cancellation

- **Thread cancellation**: Terminating a **target thread** before it has completed.
- **Scenarios**:
  - Multiple threads searching a database; one finds result, others are canceled.
  - User stops a web page from loading.
- **Cancellation Scenarios**:
  - **Asynchronous cancellation**: One thread immediately terminates the target thread.
  - **Deferred cancellation**: The target thread periodically checks whether it should terminate, allowing orderly termination.
- **Difficulties with Cancellation**:
  - Resources allocated to canceled thread may not be fully reclaimed.
  - Data shared with other threads may be left in an inconsistent state (especially with asynchronous cancellation).
- **Pthreads Cancellation**:
  - Initiated with `pthread_cancel(tid)`. This is a request; actual cancellation depends on target thread's setup.
  - Supports three cancellation modes (state and type):
    - **State**: Enabled or Disabled.
    - **Type**: Deferred or Asynchronous.

  - Default type is deferred cancellation.
  - Cancellation occurs at a **cancellation point** (e.g., blocking system calls like `read()`).
  - `pthread_testcancel()`: Function to explicitly establish a cancellation point.
  - **Cleanup handler**: A function invoked if a thread is canceled, allowing resource release before termination.
  - Asynchronous cancellation is generally not recommended in Pthreads.
- **Java Thread Cancellation**:
  - Similar to deferred cancellation.
  - Invoke `interrupt()` method on target thread to set its interruption status to true.
  - Thread checks its interruption status using `isInterrupted()` method.

6.4 Thread-Local Storage (TLS)

- **Thread-local storage (TLS)**: Data unique to each thread, even though threads typically share process data.
- **Purpose**: When each thread needs its own copy of certain data (e.g., unique transaction ID for each transaction-processing thread).
- **Distinction from Local Variables**: TLS data are visible across function invocations, unlike local variables.
- Useful when thread creation is not controlled by the developer (e.g., thread pools).
- Similar to `static` data, but unique per thread (often declared as `static`).
- **Support**: Most thread libraries and compilers provide TLS support.
  - Java: `ThreadLocal<T>` class with `set()` and `get()` methods.
  - Pthreads: `pthread_key_t` for thread-specific keys to access TLS data.
  - C#: `[ThreadStatic]` attribute.
  - `gcc` compiler: `_thread` storage class keyword.

6.5 Scheduler Activations

- Concerns communication between the kernel and the thread library, especially for many-to-many and two-level models.
- Aims to dynamically adjust the number of kernel threads for optimal performance.
- **Lightweight Process (LWP)**: An intermediate data structure between user and kernel threads.
  - Appears as a virtual processor to the user-thread library.
  - User threads are scheduled onto LWPs.
  - Each LWP is attached to a kernel thread, which the OS schedules on physical processors.
  - If a kernel thread (and thus its LWP) blocks, the user-level thread also blocks.
  - Applications may require multiple LWPs for I/O-intensive tasks.
- **Scheduler Activation**: A communication scheme between user-thread library and kernel.
  - Kernel provides LWPs to the application.
  - Kernel informs application about events via an **upcall**.
  - **Upcall handler**: Function in the thread library that handles upcalls, running on a virtual processor.
  - **Example Upcall**: When an application thread is about to block, the kernel makes an upcall, allocates a new virtual processor, and the upcall handler saves the blocking thread's state and schedules another thread. When the blocking event completes, another upcall informs the library, and the unblocked thread becomes eligible to run.

Section glossary

Term	Definition
signal	In UNIX and other operating systems, a means used to notify a process that an event has occurred.
default signal handler	The signal handler that receives signals unless a user-defined signal handler is provided by a process.
user-defined signal handler	The signal handler created by a process to provide non-default signal handling.
asynchronous procedure call (APC)	A facility that enables a user thread to specify a function that is to be called when the user thread receives notification of a particular event.
thread cancellation	Termination of a thread before it has completed.
cancellation point	With deferred thread cancellation, a point in the code at which it is safe to terminate the thread.
clean-up handler	A function that allows any resources a thread has acquired to be released before the thread is terminated.
thread-local storage (TLS)	Data available only to a given thread.
lightweight process (LWP)	A virtual processor-like data structure allowing a user thread to map to a kernel thread.
scheduler activation	A threading method in which the kernel provides an application with a set of LWPs, and the application can schedule user threads onto an available virtual processor and receive upcalls from the kernel to be informed of certain events.
upcall	A threading method in which the kernel sends a signal to a process thread to communicate an event.
upcall handler	A function in a process that handles upcalls.

7 Operating-System Examples

7.1 Windows Threads

- A Windows application runs as a separate process, which can contain one or more threads.
- Windows uses the one-to-one mapping model, where each user-level thread maps to an associated kernel thread.
- **General Components of a Thread**:
  - Thread ID (unique identifier).
  - Register set (processor status).
  - Program counter.
  - User stack (for user mode) and kernel stack (for kernel mode).
  - Private storage area (for run-time libraries and DLLs).
- The register set, stacks, and private storage area constitute the **context** of the thread.
- **Primary Data Structures of a Thread**:
  - **ETHREAD** (executive thread block): Contains a pointer to the owning process and the thread's starting routine address; also points to the corresponding KTHREAD.
  - **KTHREAD** (kernel thread block): Includes scheduling and synchronization information, the kernel stack, and a pointer to the TEB.
  - **TEB** (thread environment block): A user-space data structure containing the thread identifier, user-mode stack, and an array for thread-local storage.
- ETHREAD and KTHREAD exist in kernel space (only accessible by the kernel); TEB is in user space.

7.2 Linux Threads

- Linux provides the `fork()` system call for duplicating a process.
- Linux also provides the `clone()` system call for creating threads.
- Linux does not distinguish between processes and threads; it uses the term **task** to refer to a flow of control.
- `clone()` **System Call**:
  - Passed a set of flags that determine the level of sharing between parent and child tasks.
  - **Examples of flags**:
    - `CLONE_FS`: Shares file-system information.
    - `CLONE_VM`: Shares the same memory space.
    - `CLONE_SIGHAND`: Shares signal handlers.
    - `CLONE_FILES`: Shares the set of open files.
  - If these flags are set, `clone()` behaves like thread creation (parent shares most resources).
  - If no flags are set, `clone()` provides functionality similar to `fork()` (no sharing).
- **Task Representation in Linux Kernel**:
  - A unique kernel data structure (`struct task_struct`) exists for each task.
  - This structure contains pointers to other data structures (e.g., open files, signal handling, virtual memory) rather than storing the data directly.
  - When `fork()` is invoked, a new task is created with copies of all associated data structures.
  - When `clone()` is invoked, the new task points to the parent's data structures, depending on the flags, allowing varying levels of sharing.
- The flexibility of `clone()` extends to creating Linux containers, which are virtualized systems running in isolation under a single Linux kernel.

Section glossary

Term	**Definition**
context	When describing a thread, the state of its execution, including its register set, program counter, user stack, kernel stack, and private storage area.
ETHREAD	Executive thread block; a primary data structure for a Windows thread, existing in kernel space.
KTHREAD	Kernel thread block; a primary data structure for a Windows thread, existing in kernel space, containing scheduling and synchronization information.
TEB	Thread environment block; a primary data structure for a Windows thread, existing in user space, containing thread-specific information.
task	In Linux, a term referring to a flow of control within a program, encompassing both processes and threads.
clone()	The Linux system call used to create threads, allowing varying levels of resource sharing with the parent task via flags.

8 Summary

- A thread is a basic unit of CPU utilization; threads belonging to the same process share many process resources, including code and data.
- There are four primary benefits to multithreaded applications: (1) responsiveness, (2) resource sharing, (3) economy, and (4) scalability.
- **Concurrency** exists when multiple threads are making progress. **Parallelism** exists when multiple threads are making progress simultaneously. On a single-CPU system, only concurrency is possible; parallelism requires a multicore system with multiple CPUs.
- Designing multithreaded applications presents several challenges, including dividing and balancing work, dividing data between threads, identifying data dependencies, and the increased difficulty of testing and debugging.
- **Data parallelism** distributes subsets of the same data across different computing cores and performs the same operation on each core. **Task parallelism** distributes tasks (not data) across multiple cores, with each task running a unique operation.
- User applications create user-level threads, which must be mapped to kernel threads for execution on a CPU. Common mapping models include many-to-one, one-to-one, and many-to-many.
- A **thread library** provides an API for creating and managing threads. Key thread libraries include Windows, Pthreads, and Java threading. Windows is specific to Windows systems, Pthreads is for POSIX-compatible systems (UNIX, Linux, macOS), and Java threads run on any system supporting a Java Virtual Machine.
- **Implicit threading** involves identifying tasks (not threads) and allowing languages or API frameworks to create and manage threads. Approaches include thread pools, fork-join frameworks, and Grand Central Dispatch. Implicit threading is increasingly common for developing concurrent and parallel applications.
- Threads can be terminated using either **asynchronous cancellation** (immediate termination) or **deferred cancellation** (target thread periodically checks for termination, allowing orderly shutdown). Deferred cancellation is generally preferred due to issues with resource reclamation and data consistency in asynchronous cancellation.
- Unlike many other operating systems, Linux does not distinguish between processes and threads, referring to both as **tasks**. The Linux `clone()` system call can create tasks that behave more like processes or threads, depending on the flags passed for resource sharing.