Chapter 17
Parallel Algorithms and Concurrency: A High-Level View

Presented by
Paul Deitel, CEO, Deitel & Associates, Inc.

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17 Objectives

- Standard Library Parallel Algorithms
  - Profiling Sequential vs. Parallel
  - Execution Policies
- Multithreaded Programming
  - Thread States & Thread Life Cycle
  - Launching Tasks with `std::jthread`
- Producer–Consumer Relationship: A First Attempt
- Producer–Consumer: Synchronizing Access to Shared Mutable Data
  - Mutexes, Locks and Condition Variables

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17 Objectives

- Producer–Consumer: Minimizing Waits with a Circular Buffer
- Cooperatively Canceling `jthread`s
- Launching Tasks with `std::async`
- Thread-Safe, One-Time Initialization
- Brief Introduction to Atomics
- Coordinating Threads with C++20 Latches, Barriers & Semaphores

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

Outline

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.1 Introduction—Sequential, Concurrent, and Parallel Operation of Multiple Tasks

- **Sequential operation**: two tasks operate one after the other
- **Concurrent operation**: two tasks make progress, possibly in small increments and not necessarily simultaneously
- **Parallel operation**: two tasks execute truly simultaneously

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.1 Introduction—C++ Concurrency

- C++ concurrency via language and standard libraries
- Thread of execution
  - Single flow of control within a program
  - Programs can have multiple threads
  - Each has own function-call stack and program counter
- Enables tasks to execute in parallel
- Can enhance performance on single processor
  - Allow a thread to use processor while another is waiting (e.g., for I/O)

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.1 Introduction—A Concurrent Programming Use Case: Video Streaming

- Multiple threads download and play a video concurrently
- Download thread is referred to as the producer, and the play thread is referred to as the consumer
- Threads are synchronized to ensure smooth playback and avoid choppiness
- Synchronization ensures the player thread only begins when a sufficient portion of the video is in memory to keep the player proceeding smoothly
- Producer and consumer threads share data, and synchronization ensures correct execution

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.1 Introduction—Thread Safety

- When threads share mutable (modifiable) data, must ensure they do not corrupt it
- Known as making the code thread-safe

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.1 Introduction—Thread Safety Approaches

- Immutable (constant) data
  - Any number of threads can access a constant object at once
- Mutual exclusion
  - Allow only one thread to access mutable data at a time
- Atomic types
  - Automatically ensure their operations are atomic (i.e., not interruptible), so only one thread can access and possibly modify the data at a time
- Thread-local storage
  - Declare a `static` or global variable as `thread_local`
  - Each thread should have its own copy
  - Threads do not share `thread_local` variables

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.1 Introduction—C++11 and C++14: Providing Low-Level Concurrency Features

- C++11 introduced standardized multithreading
- C++11 and C++14 defined mostly low-level primitives
- C++11 added standard library features, including mutexes and locks
- C++14 added shared mutexes and shared locks
- These capabilities were used to build higher-level features in C++17, C++20

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.1 Introduction—C++17 and C++20: Providing Convenient Higher-Level Concurrency Features

- Simplify concurrent programming
- Avoid common errors
- Make programs easier to maintain and debug
- C++17: 69 parallel standard library algorithms
- C++20
    - Higher-level thread synchronization capabilities, including latches, barriers, and semaphores
    - Additional parallel algorithms
    - Coroutines

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.2 Standard Library Parallel Algorithms

- Multi-core processors for better performance
- Parallel standard library algorithms
    - Take advantage of multi-core and high-performance "vector mathematics“
    - Example: Sorting a large array by dividing it into smaller chunks and sorting each chunk on a different core simultaneously.
- Vector operations
    - Perform same task on many data items simultaneously using SIMD instructions provided by many CPUs and GPUs
    - Example: Adding two arrays element-wise using SIMD instructions, where multiple additions are performed in parallel within a single CPU cycle.

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.2.1 Example: Profiling Sequential and Parallel Sorting Algorithms

- Previously used `std::sort` to sort a `std::array` in one thread
- Let's compare with its parallel overload
  - Sort 100,000,000 randomly generated `int`s in `vector`s
- Use `<chrono>` to profile
- Results will vary based on hardware, OS, compiler, and system workload
- `std::ranges` algorithms are not parallelized

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.2.1 Example: Profiling Sequential and Parallel Sorting Algorithms

- Fig. 17.1
- Setting Up Random-Number Generation
  - Lines 13–15 set up random-number capabilities
  - Integers from 0 to `std::numeric_limits<int>::max()`
- Creating the `vector`s (lines 18-25)
  - Create a `vector` to hold 100,000,000 `int`s
  - `std::generate` to fill the `vector`
  - Copy the `vector`, so we can compare sequential and parallel sorting on `vector`s with identical contents

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.2.1 Example: Profiling Sequential and Parallel Sorting Algorithms

Timing Operations with `std::chrono`

- Namespace `std::chrono`'s `steady_clock`, `duration_cast` and `milliseconds`
- `steady_clock`—Recommended for timing operations
  - Get time before a sorting operation starts and after it completes
  - For time of day, `system_clock` recommended
- `duration_cast`
  - Converts a duration into another measurement
  - We'll convert duration to milliseconds
- `std::chrono::duration` type `milliseconds`
  - We'll convert duration to milliseconds

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.2.1 Example: Profiling Sequential and Parallel Sorting Algorithms

Sequential Sorting

- Lines 33–40 test sequential sorting
- Line 39 calculates **`duration`** and converts to **`milliseconds`**
- **`duration`**'s **`count`** member function returns milliseconds
  - We divide by 1000.0 to display the result in seconds

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.2.1 Example: Profiling Sequential and Parallel Sorting Algorithms

Parallel Sorting with `std::execution::par`

- Lines 43-50 test parallel sorting and time the result
- Each parallel algorithm overload requires an execution policy
  - Indicate whether to parallelize a task and how to do it
- `std::execution::par` executes portions of its work simultaneously on multiple cores
- Parallel sorting large volumes of data on multiple cores provides a clear performance advantage

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.2.1 Example: Profiling Sequential and Parallel Sorting Algorithms

Caution: Parallel Is Not Always Faster

- Processing small numbers of elements
- Using non-random-access iterators
- Microsoft defaults some parallel algorithms to sequential
  - Benchmarking showed slower performance for hardware targeted by Visual C++
  - `copy`, `copy_n`, `fill`, `fill_n`, `move`, `reverse`, `reverse_copy`, `rotate`, `rotate_copy`, `swap_ranges`
- Billy O'Neal from Microsoft: Use parallel algorithms for
  - Tasks that process at least 2,000 items
  - Tasks that require more than O(n) time, such as sorting

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.2.2 When to Use Parallel Algorithms

- Parallel execution might increase sort times for small datasets
- Ran preceding program with 100 to 100,000,000 `int` elements
  - four-core Windows 10, 64-bit system
  - eight-core Windows 10, 64-bit system
- On each, parallel execution started to (barely) outperform sequential execution at 10,000 elements

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.2.2 When to Use Parallel Algorithms

Number of elements	4-core sequential execution (in ns)	4-core parallel execution (in ns)	8-core sequential execution (in ns)	8-core parallel execution (in ns)
100	3,200	81,900	2,800	63,400
1,000	42,500	161,900	33,300	136,400
10,000	880,400	711,400	433,900	431,600
100,000	10,205,300	6,308,200	5,888,300	1,289,500
1,000,000	98,959,700	27,816,100	75,358,800	12,486,400
10,000,000	1,065,163,900	415,386,000	814,166,200	126,300,900
100,000,000	12,361,988,600	3,444,056,800	8,473,599,400	1,230,407,100

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.2.3 Execution Policies

- Four standard execution policies:
  - **`std::execution::seq`** – execute in a single thread
  - **`std::execution::par`** – algorithm can be parallelized
  - **`std::execution::par_unseq`** – algorithm can be parallelized and vectorized
  - **`std::execution::unseq`** (C++20) – algorithm can be vectorized
- C++ is used on a wide range of devices and OSs
- Some do not support parallelism
- Execution policies are suggestions

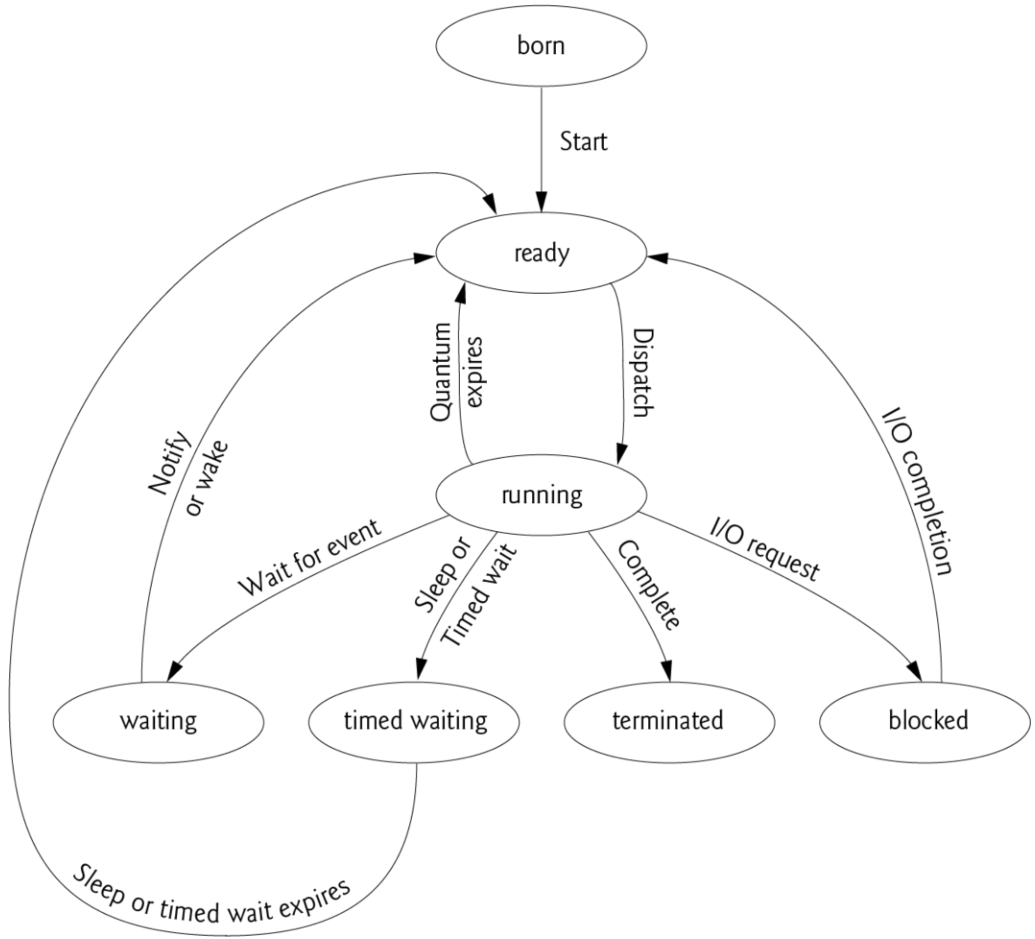
DEITEL®
©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.3 Multithreaded Programming

- Separate threads enable parallel execution of program tasks when sufficient cores are available
- Programs compete with OS, other applications, and background tasks for processor attention
- Completion time for tasks can vary based on processor speed, number of cores, and other activities running on the system

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.3.1 Thread States and the Thread Life Cycle



©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.3.1 Thread States and the Thread Life Cycle

- Born and Ready States
  - New thread begins in **born state**
  - When program starts, thread moves to **ready state**
  - In C++, constructing a thread object with a function as an argument creates a thread and immediately starts it

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.3.1 Thread States and the Thread Life Cycle

- Running State
  - Ready thread enters the running state (begins executing) when the OS assigns it to a processor
    - Known as dispatching the thread
  - OS gives each thread a small amount of processor time—called a quantum or timeslice—to make progress on its task
  - When its quantum expires, thread returns to the ready state, and OS assigns another thread to the processor
  - OS's thread scheduling decisions must be made carefully to ensure good performance and avoid problems like indefinite postponement of waiting threads

DEITEL®
©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.3.1 Thread States and the Thread Life Cycle

- Waiting State
  - Running thread transitions to waiting state to wait for another thread to perform a task
  - Waiting thread transitions back to the ready state when another thread notifies it to continue executing

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.3.1 Thread States and the Thread Life Cycle

- Timed Waiting State
  - Running thread enters timed waiting state for a specified time interval
  - Transitions back to ready state when time interval expires or the event it's waiting for occurs
  - Putting a running thread to sleep also transitions the thread to the timed waiting state
  - Sleeping thread remains in that state for a period of time (called a sleep interval), after which it returns to the ready state
  - Threads sleep when they momentarily do not have work to perform

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.3.1 Thread States and the Thread Life Cycle

- Blocked State
  - Running thread transitions to blocked state when it attempts to perform a task that cannot be completed immediately
  - For example, when a thread issues an I/O request, OS blocks the thread from executing until the I/O completes
  - Blocked thread then transitions to the ready state to resume execution
- Terminated State
  - Running thread enters terminated state when it completes its task

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.3.1 Thread States and the Thread Life Cycle

- Thread Scheduling
  - Timeslicing enables threads to share a processor
  - Even if a thread has not finished executing when its quantum expires, OS takes the processor away and gives it to the next thread if one is available
  - OS thread scheduler determines which thread runs next
  - Thread scheduling is platform-dependent

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.3.2 Deadlock and Indefinite Postponement

- When a higher-priority thread enters the ready state, OS generally preempts the running thread
  - preemptive scheduling
- Steady influx of higher-priority threads could indefinitely postpone execution of lower-priority threads.
  - starvation
- OS can use aging to prevent starvation
  - As thread waits in ready state, OS gradually increases the thread’s priority to ensure that it will eventually run

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.3.2 Deadlock and Indefinite Postponement

- A thread is deadlocked if waiting for an event that will not occur
- When resources are shared among a set of threads, with each thread maintaining exclusive control over particular resources allocated to it, deadlocks can develop in which some threads will never be able to complete execution
- Result can be reduced system throughput and even system failure

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.3.2 Deadlock and Indefinite Postponement

- Four Necessary Conditions for Deadlock
  - A resource may be acquired for the exclusive use of only one thread at a time (mutual exclusion condition)
  - A thread that has acquired an exclusive resource may hold that resource while the thread waits to obtain other resources (wait-for condition, also called the hold-and-wait condition)
  - Once a thread has obtained a resource, the system cannot remove it from the thread’s control until the thread has finished using the resource (no-preemption condition)
  - Two or more threads are locked in a “circular chain” in which each thread is waiting for one or more resources that the next thread in the chain is holding (circular-wait condition)
- Disallowing any of these prevents deadlock from occurring

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.3.2 Deadlock and Indefinite Postponement

- Indefinite Postponement
  - A thread that is not deadlocked could wait for an event that might never occur or might occur unpredictably far in the future due to biases in the system's resource-scheduling policies

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.3.2 Deadlock and Indefinite Postponement

- There are various deadlock-prevention techniques
- Most practical approach is for each thread to request all its required resources at once and not proceed until all have been granted
- If a thread requests and gets all its needed resources at once, runs to completion, then releases them, there cannot be a circular wait, and the thread cannot deadlock
- If the thread cannot get all its resources at once, it should cancel the request and try again later, allowing other threads to proceed
- Not a perfect solution—can lead to indefinite postponement and might result in poor system-resource utilization

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.3.2 Deadlock and Indefinite Postponement

- Deadlock and indefinite postponement each involve some form of waiting
- In concurrent programming, these waiting scenarios often develop in subtle ways that are not easily detectable, especially as the number of active concurrent tasks grows
- A crucial key to building reliable business-critical and mission-critical systems is the trend toward developing and employing higher-level concurrency primitives

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.4 Launching Tasks with `std::jthread`

- `std::thread` and `std::jthread` (C++20) for launching concurrent tasks (`<thread>` header)
- `std::jthread` fixes several `std::thread` problems
- Creating a task to execute concurrently
  - Create a function, lambda or function object
  - Initialize a `std::jthread` object with it
- Task's return value is ignored
  - Other mechanisms are used to communicate data between threads.
- Exceptions in `std::jthread`s
  - If a `std::jthread`'s task exits via an exception, the program terminates by calling `std::terminate`

DEITEL®

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.4.1 Defining a Task to Perform in a Thread

- Fig. 17.3, lines 11–15 (printtask)
  - Every thread has a unique ID number
  - `std::this_thread::get_id()`
  - ID returned as a `std::thread::id` object
  - Can use its `operator<<` to convert to `std::string`

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.4.1 Defining a Task to Perform in a Thread

- Function `printTask` (lines 18–39) parameters
  - name we use to identify each task in program's output
  - `sleepTime` in milliseconds
- When a thread calls `printTask`
  - Lines 26–27 display message with current thread's name, unique ID and `sleepTime`
  - Line 29 gets time before thread goes to sleep
  - Line 32 calls `std::this_thread::sleep_for`
    - Thread loses the processor
    - Our examples often make threads sleep to simulate performing work
    - `std::this_thread::sleep_until` sleeps until a specified time

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.4.1 Defining a Task to Perform in a Thread

- When a thread calls `printTask` (continued...)
  - When sleep time expires, thread reenters ready state but does not necessarily execute immediately
  - When OS assigns a processor
    - Line 34 gets the time
    - Line 35 calculates total time thread was not executing
    - Line 36 calculates difference between total time and `sleepTime`
    - Lines 37–38 display the times
  - When `printTask` terminates, its `jthread` enters the terminated state

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.4.2 Executing a Task in a jthread

- Function **`main`** launches two concurrent threads that execute **`printTask`**
- Line 18 creates a **`vector`** to store **`std::jthread`**s
  - We use this to enable **`main`** to wait for the threads to complete their tasks before the program terminates
- Lines 23–32 create two **`std::jthread`**s
  - **`jthread`** constructor receives the function to execute (**`printTask`**) and arguments to pass to it
  - We create **`jthread`** as a temporary object, so **`vector`**’s **`push_back`** function moves the new **`jthread`** into the new **`vector`** element
- Constructing a **`jthread`** with a function to execute starts the **`jthread`**

DEITEL®
©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.4.2 Executing a Task in a jthread

- Waiting for Previously Scheduled Tasks to Terminate
  - After scheduling tasks to execute, you typically want to wait for them to complete—for example, to use their results
  - "join the thread" with a call to its `join` function explicitly or implicitly via `jthread`'s destructor
    - One of `jthread`'s benefits over `std::thread`
- Thread with shortest sleep time typically awakens first
- Cannot predict the order in which tasks will start executing, even if we know order in which they were created and started

DEITEL®
©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.4.3 How `jthread` Fixes `thread`

- Prefer `std::jthread` over `std::thread`
- `thread` has various problems
  - If you do not `join` a thread or `detach` it before it's destroyed, its destructor calls `std::terminate`
  - Must explicitly join each thread
    - ```
      for (auto& t : threads) {
         t.join();
      }
      ```
  - If an uncaught exception occurs before joining each `thread`, `thread`s will be destroyed, and first `thread` destructor to execute will call `std::terminate`
  - `std::jthread` fixes this by auto-joining in its destructor
- `jthread` also
  - supports cooperative cancellation (Section 17.9)
  - supports proper move semantics
  - is an RAII type (discussed in Section 11.5) that correctly cleans up its resources

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com
      ```

RAII: Resource Allocation Is Initializatoin

- aka CADRe (Constructor Acquires, Destructor Releases)
- Resource allocation is done during object creation
- Resource deallocation (release) is done during object destruction
- Special case:
  - Allocation of dynamic memory, release in destructor
  - Use of automatic variables: destroyed automatically when variable goes out of scope → use smart pointers

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

11.5 Modern C++ Dynamic Memory Management—RAII and Smart Pointers (1 of 2)

- Common design pattern
  - Allocate dynamic memory
  - Assign the address of that memory to a pointer
  - Use the pointer to manipulate the memory
  - Deallocate the memory when it's no longer needed
- If an exception occurs before deallocation → **memory leak**

DEITEL®
©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

11.5 Modern C++ Dynamic Memory Management—RAII and Smart Pointers (2 of 2)

- C++ Core Guidelines recommend **RAII—Resource Acquisition Is Initialization**
  - Create local object and **acquire the resource during construction**
  - Use the object
  - When the object goes out of scope, **destructor called automatically** to release the resource

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

RAII: exception safe file allocation

```
#include <fstream>
#include <iostream>
#include <mutex>
#include <stdexcept>
#include <string>

void WriteToFile(const std::string& message) {
  // |mutex| is to protect access to |file| (which is shared across threads).
  static std::mutex mutex;

  // Lock |mutex| before accessing |file|.
  std::lock_guard<std::mutex> lock(mutex);

  // Try to open file.
  std::ofstream file("example.txt");
  if (!file.is_open()) {
    throw std::runtime_error("unable to open file");
  }

  // Write |message| to |file|.
  file << message << std::endl;

  // |file| will be closed first when leaving scope (regardless of exception)
  // |mutex| will be unlocked second (from |lock| destructor) when leaving scope
  // (regardless of exception).
}
```

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.5 Producer–Consumer Relationship: A First Attempt

- producer–consumer relationship
  - a producer thread generates data and stores it in a shared object
  - a consumer thread reads data from that shared object
- When concurrent threads share mutable data and that data is modified indeterminate results may occur
- Program's behavior cannot be trusted
- Could appear to run correctly on one run and incorrectly on the next

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.5 Producer–Consumer Relationship: A First Attempt

- Correctness requires synchronization
  - **Operations on shared mutable data accessed by concurrent threads must be guarded with a lock to prevent corruption**
- Operations on the shared buffer are state-dependent
  - If buffer is not full, producer may produce
  - If buffer is not empty, consumer may consume
  - If buffer is full when producer wants to write a new value, it must wait until there's space in the buffer
  - If buffer is empty when consumer wants to read a value, it must wait for new data to become available

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.5 Producer–Consumer Relationship: A First Attempt

Logic Errors from Lack of Synchronization

- Each value the producer thread writes to the shared buffer must be consumed exactly once by the consumer thread
- Without synchronization
  - data can be lost or garbled if the producer places new data into the shared buffer before the consumer reads the previous data
  - data can be incorrectly duplicated if the consumer consumes the same data again before the producer produces the next value
- Our consumer thread totals the values it reads
  - Producer produces values from 1 through 10
  - If consumer correctly reads each value only once, the total will be 55
  - Could still incorrectly get the correct total of 55

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.5 Producer–Consumer Relationship: A First Attempt

- `UnsynchronizedBuffer` does not synchronize access to its data
  - **not thread-safe**
- `m_buffer` set to `–1` to show when consumer attempts to consume a value before the producer ever places a value in `m_buffer`

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.5 Producer–Consumer Relationship: A First Attempt

- Fig. 17.6
  - Line 12 creates `UnsynchronizedBuffer` object `buffer` shared by the concurrent producer and consumer threads
  - Producer thread will invoke `produce` (lines 15–36)
  - Consumer thread will invoke `consume` (lines 39–60)
  - Both lambdas capture `buffer` by reference
  - Lines 65–66 create two `jthread`s
    - `producer` executes the `produce` lambda, and `consumer` executes the `consume` lambda

DEITEL®
©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.5 Producer–Consumer Relationship: A First Attempt

- **Random-number generation is not thread-safe**
- To ensure each thread can safely produce random numbers, we defined separate random-number generators in the lambdas `produce` (lines 18–20) and `consume` (lines 42–44) rather than sharing one random-number generator between them
  - See C++ Standard, "16.5.5.10 Data Race Avoidance." `https://timsong-cpp.github.io/cppwp/n4861/res.on.data.races`

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.5 Producer–Consumer Relationship: A First Attempt

`std::this_thread::sleep_for` used for demo purposes

- **To emphasize that you cannot predict the relative speeds of asynchronous concurrent threads, we call function `std::this_thread::sleep_for`**
- **It's generally unpredictable when and for how long a thread will perform its task when it has a processor**
- Without the `sleep_for` calls
  - If the producer were to execute first, the producer would likely complete its task before the consumer got a chance to execute
  - If the consumer were to execute first, it would likely consume the same garbage data ten times, then terminate before the producer could produce the first real value

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.5 Producer–Consumer Relationship: A First Attempt

- Spotting errors is challenging in multithreaded programs
  - May occur so infrequently and unpredictably that a broken program does not produce incorrect results during testing, creating the illusion that it’s correct
- **Prefer using predefined containers and higher-level primitives that handle the synchronization for you**

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.6 Producer–Consumer: Synchronizing Access to Shared Mutable Data

- Data races (race conditions)
  - The thread that “wins the race” by getting there first performs its task, even if it should not
  - If producer wins, it overwrites previously written values before they’re consumed, causing lost data
  - If consumer wins, it reads invalid data (`-1`) before the producer has produced its first legitimate value or reads stale values it read previously

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.6 Producer–Consumer: Synchronizing Access to Shared Mutable Data

Thread Synchronization, Mutual Exclusion and Critical Sections

- Give only one thread at a time exclusive access to code that manipulates shared mutable data
- During that time, the other thread must wait
- When thread with exclusive access finishes accessing the shared mutable data, the waiting thread can proceed
- **mutual exclusion** – each thread accessing a shared object excludes the other thread from doing so simultaneously
- **critical sections** – code sections protected using mutual exclusion

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.6 Producer–Consumer: Synchronizing Access to Shared Mutable Data

Executing a Set of Operations As If They Are One

- To make our shared buffer thread safe
  - Ensure that only one thread at a time can store a value in the buffer or read a value from the buffer
  - Ensure that these operations cannot be divided into smaller suboperations—known as making the operations atomic

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.6 Producer–Consumer: Synchronizing Access to Shared Mutable Data

- C++ Core Guidelines: "You can't have a race condition on a constant"
  - "use `const` to define objects with values that do not change after construction" (Con: Constants and Immutability, https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-const)
  - "use `constexpr` for values that can be computed at compile time" (CP.3: Minimize Explicit Sharing of Writable Data. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-data)

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.6.1 Class `SynchronizedBuffer`: Mutexes, Locks and Condition Variables

- Figures 17.7–17.8 demonstrate correctly accessing a synchronized shared mutable buffer
  - producer always produces a value first
  - consumer correctly consumes only after producer produces
  - producer correctly produces next value only after consumer consumes previous (or first) value
- **Note: We output messages from synchronized operations for demo purposes only**
  - I/O is slow and should not be performed in critical sections
  - Crucial to minimize amount of time an object is "locked"

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.6.1 Class **SynchronizedBuffer**: Mutexes, Locks and Condition Variables

SynchronizedBuffer's **m_buffer** and **m_occupied** Data Members

- `m_buffer` – `int` in which a producer thread will write data and from which a consumer thread will read data.
- `m_occupied` – `bool` indicating whether `m_buffer` contains data; tracks the shared buffer's state for synchronization
- Both part of a `SynchronizedBuffer`'s state
  - Must synchronize access to ensure the buffer is thread-safe

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.6.1 Class `SynchronizedBuffer`: Mutexes, Locks and Condition Variables

`SynchronizedBuffer`’s `std::condition_variable`

- Producer/consumer proceed only if buffer in correct state
    - `m_occupied` is `false`—buffer empty, producer can produce
    - `m_occupied` is `true`—buffer full, consumer can consume
- Must tell waiting thread when condition changes
    - After writing, producer notifies waiting consumer (if there is one)
    - After reading, consumer notifies waiting producer (if there is one)
- Wait/notify via `std::condition_variable`
    - header `<condition_variable>`

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.6.1 Class **SynchronizedBuffer**: Mutexes, Locks and Condition Variables

`SynchronizedBuffer`'s `std::mutex` Data Member

- Critical sections
  - Synchronized blocks of code
  - Execute atomically using features from `<mutex>`
- `std::mutex` can be owned by only one thread at a time
- Thread requiring exclusive access to a resource must first acquire `mutex`'s lock — typically at the beginning of a block
- Other threads attempting to lock same `mutex` are blocked until lock is released—typically at the end of a block

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.6.1 Class `SynchronizedBuffer`: Mutexes, Locks and Condition Variables

`SynchronizedBuffer`'s `std::mutex` Data Member

- If multiple critical sections are synchronized with the same `std::mutex`, only one can execute at a time
- In multithreaded programs, place all accesses to shared mutable data in critical sections that synchronize on the same `std::mutex`
- All operations within a critical section represent an atomic operation
- Promptly release the lock when it's no longer needed

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.7 Producer–Consumer: Minimizing Waits with a Circular Buffer

- Problem: Asynchronous concurrent threads may wait excessively, causing less efficiency, less responsiveness, and long delays
- Solution: Use a circular buffer to minimize waiting among concurrent threads sharing resources and operating at the same average speeds

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.7 Producer–Consumer: Minimizing Waits with a Circular Buffer

Circular Buffer

- Fixed number of cells for producer to write values into and consumer to read values from
- Manages writes and reads in order, wrapping around when reaching the last element

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.7 Producer–Consumer: Minimizing Waits with a Circular Buffer

- Benefits
  - Producer can write additional values if temporarily operating faster than consumer
  - Consumer can read additional values if temporarily operating faster than producer
- Limitations
  - Producer may still need to wait if buffer is full
  - Consumer may still need to wait if buffer is empty
  - Inappropriate if producer and consumer operate at significantly different average speeds

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.7 Producer–Consumer: Minimizing Waits with a Circular Buffer

- Implementing a Circular Buffer
  - Create an array of an optimal size (depends on the application)
  - Manage access to shared mutable data with synchronization (e.g., `std::condition_variable`, `std::mutex`)
  - Key: Optimizing the buffer size to minimize thread wait times while not wasting memory

DEITEL®
©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.9 Cooperatively Canceling `jthreads`

- Multithreaded application termination
  - Shut down threads still performing tasks
  - Release resources
    - Close files
    - Close database connections
    - Close network connections

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.9 Cooperatively Canceling `jthread`s

- Prior to C++20, no graceful built-in mechanism
- C++20 `jthread`s support cooperative cancelation
  - Programs can notify tasks to terminate
  - Tasks can watch for notifications, complete critical work, release resources and terminate

DEITEL®
©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.9 Cooperatively Canceling jthreads

- jthread has a std::stop_source with associated std::stop_token
    - <stop_token> header
- jthread’s task function can optionally specify a stop_token as first parameter
- Task function periodically calls stop_token’s stop_requested member function
    - Returns true if task should stop executing

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.9 Cooperatively Canceling `jthread`s

- To tell a task to stop, another thread or the `jthread`'s destructor calls the `jthread`'s `request_stop`
- `stop_source` notifies the `stop_token`
  - Next call to `stop_token`'s `stop_requested` returns `true`
  - Task can gracefully shut down
- If task function never calls `stop_requested`, `jthread` continues executing
  - Hence **cooperative cancellation**

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.9 Cooperatively Canceling `jthread`s

Optional `stop_callback`

- Registers a function with a given `stop_token`
  - Constructor receives as arguments a `stop_token` and a function with no parameters
- When `stop_token` notified of stop request, it calls `stop_callback` on thread that requested the stop
- Any number of `stop_callback`s can be created
- Execution order is not specified

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.10 Launching Tasks with `std::async`

- `<future>` header
  - features to **execute asynchronous tasks**
  - receive tasks' results when they finish executing
- `std::async`—Higher-level way to launch a task in a separate thread
- Two versions
  - One receives a `std::launch` policy
  - The other chooses a `std::launch` policy for you

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.10 Launching Tasks with `std::async`

- Launch policy options
  - `std::launch::async`
  - `std::launch::deferred`
  - Both separated by a bitwise OR (`|`) operator
- Combining these values lets the system choose whether to execute asynchronously or synchronously

DEITEL®
©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.10 Launching Tasks with `std::async`

Inter-Thread Communication

- `std::async` returns a `std::future`
- Enables communication between thread that calls `async` and task `async` executes
- C++ Core Guidelines recommend using a `future` to return a result from an asynchronous task
- “Under the hood”
  - `async` uses a `std::promise` from which it obtains a `future`
  - When task completes, `async` stores task’s result in the `promise`
  - `async`’s caller uses the `future` to access result in the `promise`
  - Do not need to work with the `promise` directly

DEITEL®
©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.10 Launching Tasks with `std::async`

- Demo task to execute: **`getFactors`** function
  - Determines whether a number is prime
  - If not, determines its prime factors
- To ensure demo tasks run for a few seconds each, used two 19-digit numbers, including a prime value from the University of Tennessee Martin's **Prime Pages** website
  - https://primes.utm.edu/curios/index.ph

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.12 A Brief Intro to Atomics

- `<atomic>` header
- Atomic-type operations are indivisible
  - Can share mutable data without explicit synchronization
- Higher-level in that they don't require programmers to synchronize access
  → Generally considered a low-level feature
- Used under-the-hood of C++20 library features like `std::latch`, `std::barrier`, `std::semaphore`

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.12 A Brief Intro to Atomics

- Predefined `std::atomic` specializations
  - `bool`, integer, pointer, floating-point, trivially copyable types
- Atomic smart pointers
  - `std::atomic<shared_ptr<T>>`
  - `std::atomic<weak_ptr<T>>`
- `std::atomic_ref`
  - Can be initialized with reference to object of any trivially copyable type

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.12 A Brief Intro to Atomics

- Demo uses concurrent threads to increment
  - An `int`
  - A `std::atomic<int>`
  - A `std::atomic_ref<int>`
- `++` one of a limited set of arithmetic and bitwise operations
  - https://en.cppreference.com/w/cpp/atomic/atomic

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.13 Coordinating Threads with C++20 Latches and Barriers

- C++20 added `std::latch` and `std::barrier`
- Synchronization without explicit use of mutexes, condition variables and locks

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.13.1 C++20 `std::latch`

- From the `<latch>` header
- Single-use gateway
- Remains closed until a specified number of threads reach the latch
- Then remains open permanently
- Serves as a one-time synchronization point
  - Allows threads to wait until a specified number of threads reach that point

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.13.1 C++20 `std::latch`

- Consider a parallel sorting algorithm that
  - launches several worker threads to sort portions of a large array,
  - waits for the worker threads to complete, then
  - merges the sorted sub-arrays into the final sorted array
- Assume the algorithm uses two worker threads, each sorting half the array

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.13.1 C++20 `std::latch`

- Use `std::latch` to wait until the workers are done
- Create `std::latch` with a non-zero count
  - Each worker references the same latch
- After launching workers, algorithm waits on that latch
  - Blocks algorithm from continuing
- When a worker thread completes its task, it reduces latch's count by 1, known as signaling the latch
- When latch's count becomes 0, it opens permanently, unblocking thread(s) waiting on the latch
  - Once latch is open, any thread attempting to wait on it simply passes through the gateway and continues executing

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.13.2 C++20 `std::barrier`

- `<barrier>` header
- Like a reusable latch
- Typically, used for repetitive tasks in a loop
  - Each thread works, then reaches a barrier and waits for it to open
  - When the specified number of threads reaches the barrier, an optional completion function executes
  - The barrier resets its count, which unblocks the threads so they may continue executing and repeat this process

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.13.2 C++20 `std::barrier`

- Consider a simulation of the painting step in an automated automobile assembly line
  - Often several robots work together to perform a given step
- Assume separate threads control the cars moving along the assembly line and two robots' operations
- Once the work on one car finishes, we want to
  - reset everything
  - advance the assembly line
  - perform the work again for the next car
- Ideal for a `std::barrier`

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.14 C++20 Semaphores

- Another mutual-exclusion mechanism
- Semaphore contains an integer value representing maximum number of concurrent threads that can access a shared resource
  - such as shared mutable data
- Once initialized, that integer can be accessed and altered by only two operations
  - `acquire` when a thread wants to enter a critical section
  - `release` when a thread wants to exit a critical section
- Once the maximum number of threads are operating in the critical section, other threads trying to enter must wait
- Can support any number of cooperating threads

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.14 C++20 Semaphores

- `<semaphore>` header
  - Defines semaphore capabilities
  - Lower level than `std::latch` and `std::barrier`, but higher level than mutexes, locks, condition variables and atomics
- C++ standard says semaphores “are widely used to implement other synchronization primitives and, whenever both are applicable, can be more efficient than condition variables.”

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

17.14 C++20 Semaphores

- `std::counting_semaphore`
  - Allows multiple threads to access a shared resource
  - Maintains an internal integer counter
  - When a thread `acquire`s a semaphore, internal counter decrements
  - If counter is 0, a thread attempting to `acquire` will block until counter increases to indicate the shared resource is available
  - When a thread `release`s the semaphore, internal counter increments by one (by default) and threads waiting unblock
- `std::binary_semaphore`
  - A `counting_semaphore` with a count of 1
  - Used like a `std::mutex`

DEITEL®
©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com

Exercise

- Redesign the readings exercise in a multithreaded way:
  - Make 3 threads: reader, processor, writer.
  - Which techniques will you use to optimize throughput?

©Copyright 1992-2024 by Pearson Education, Inc. All Rights Reserved. https://deitel.com