OPERATING SYSTEMS

Threads and Concurrency 08

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

OPERATING SYSTEMS

Course Syllabus - Unit 2

UNIT 2: Threads and Concurrency

Introduction to Threads, types of threads, Multicore Programming, Multithreading Models, Thread creation, Thread Scheduling, PThreads and Windows Threads, Mutual Exclusion and Synchronization: software approaches, principles of concurrency, hardware support, Mutex Locks, Semaphores. Classic problems of Synchronization: Bounded-Buffer Problem, Readers -Writers problem, Dining Philosophers Problem concepts. Synchronization Examples - Synchronisation mechanisms provided by Linux/Windows/Pthreads. Deadlocks: principles of deadlock, tools for detection and Prevention.

OPERATING SYSTEMS

Course Outline - Unit 2

13	4.1,4.2	Introduction to Threads, types of threads, Multicore Programming.	4	42.8
14	4.3,5.4	Multithreading Models, Thread creation, Thread Scheduling	4	
15	4.4	Pthreads and Windows Threads	4	
16	6.1-6.3	Mutual Exclusion and Synchronization: software approaches	6	
17	6.3-6.4	principles of concurrency, hardware support	6	
18	6.5,6.6	Mutex Locks, Semaphores	6	
19	6.7.1-6.7.3	Classic problems of Synchronization: Bounded-Buffer Problem, Readers -Writers problem, Dining Philosophers Problem concepts	6	
20	6.9	Synchronization Examples - Synchronisation mechanisms provided by Linux/Windows/Pthreads.	6	
21	Handouts	Demonstration of programming examples on process synchronization		
22	7.1-7.3	Deadlocks: principles of deadlock, Deadlock Characterization.	7	
23	7.4	Deadlock Prevention, Deadlock example	7	
24	7.6	Deadlock Detection	7	

Topics Outline

- Mutex Locks - acquire(), release()
- Recursive Mutexes, Reader / Writer Mutexes, Spinlocks
- Pthread Mutexes

Synchronization: Mutex Locks

- **Mutex** is short for **MUT**ual **EX**clusion.
- Previous solutions are complicated and generally **inaccessible** to application programmers
- **OS designers** build software **tools** to solve critical section problem
- Simplest software tool is **mutex lock**
- It is used to protect a critical section by first calling **acquire**() a lock function, then **release**() the lock release function

Synchronization: Mutex Locks

- Boolean variable indicating if **lock** is available or not
- Calls to acquire() and release() must be **atomic**
- Usually **implemented** via **hardware atomic** instructions
- But this solution requires busy waiting
- This **lock** therefore are also called a **spinlock**

Synchronization: Mutex Locks

- **Mutex** is a mutual exclusion object that **synchronizes** access to a resource.
- It is **created** with a unique name at the **start** of a program.
- The **Mutex** is a locking mechanism that makes **sure only one** thread / process can **acquire** the **Mutex Variable** at a time and enter its **critical section**.

Synchronization: Mutex Locks

- This **thread / process** only releases the Mutex when it exits the critical section.

- Unless the word is qualified with additional terms such as shared mutex, recursive mutex or read/write mutex then it refers to a type of **lockable** object that can be **owned** by exactly **one thread** at a **time**.

- Only the **thread** that **acquired** the lock can **release** the lock on a **mutex**. When the mutex is **locked**, any **attempt** to **acquire** the lock will **fail** or block, even if that attempt is done by the **same thread**.

Synchronization: Mutex Locks

Recursive Mutexes

- A **recursive mutex** is similar to a plain mutex, but **one thread** may **own multiple** locks on it at the **same** time.
- If a lock on a recursive mutex has been acquired by thread A, then thread A can **acquire** further **locks** on the recursive mutex without releasing the locks already held.
- However, thread B **cannot** acquire any locks on the recursive mutex until **all** the locks held by **thread** A have been **released**.

Synchronization: Mutex Locks

Recursive Mutexes

- In most cases, a **recursive mutex** is **undesirable**, since the it makes it harder to reason correctly about the code. With a plain mutex, if you ensure that the invariants on the protected resource are valid before you release ownership then you know that when you acquire ownership those invariants will be valid.

- With a **recursive mutex** this is not the case, since being able to acquire the lock does not mean that the lock was not already held, by the current thread, and therefore does not imply that the invariants are valid.

Synchronization: Mutex Locks

Reader/Writer Mutexes

- Sometimes called **shared mutexes**, multiple-reader/single-writer mutexes or just read/write mutexes, these offer two distinct types of ownership: **shared ownership**, also called **read ownership**, or a **read lock**, and **exclusive ownership**, also called **write ownership**, or a **write lock**.

- Exclusive ownership **works** just like ownership of a **plain mutex**: only one thread may hold an exclusive lock on the mutex, only that thread can release the lock. No other thread may hold any type of lock on the mutex while that thread holds its lock.

Synchronization: Mutex Locks

Reader/Writer Mutexes

- **Shared ownership** is more **lax**. **Any** number of threads may **take shared ownership** of a **mutex** at the **same time**. No thread may take an exclusive lock on the mutex while any thread holds a shared lock.

- These mutexes are typically used for protecting shared data that is seldom updated, but cannot be safely updated if any thread is reading it. The reading threads thus take shared ownership while they are reading the data. When the data needs to be modified, the modifying thread first takes exclusive ownership of the mutex, thus ensuring that no other thread is reading it, then releases the exclusive lock after the modification has been done.

Synchronization: Mutex Locks

Spinlocks

- A **spinlock** is a special type of **mutex** that does **not** use **OS synchronization** functions when a lock operation has to wait. Instead, it just keeps trying to update the mutex data structure to take the lock in a loop.
- If the lock is not held very often, and/or is only held for very short periods, then this can be more efficient than calling heavyweight thread synchronization functions.
- However, if the processor has to loop too many times then it is just wasting time doing nothing, and the system would do better if the OS scheduled another thread with active work to do instead of the thread failing to acquire the spinlock.

Programming with Threads - Andrew Birrell

An Introduction to Programming with Threads

Andrew D. Birrell

This paper provides an introduction to writing concurrent programs with "threads". A threads facility allows you to write programs with multiple simultaneous points of execution, synchronizing through shared memory. The paper describes the basic thread and synchronization primitives, then for each primitive provides a tutorial on how to use it. The tutorial sections provide advice on the best ways to use the primitives, give warnings about what can go wrong and offer hints about how to avoid these pitfalls. The paper is aimed at experienced programmers who want to acquire practical expertise in writing concurrent programs.

CR categories and Subject Descriptors: D.1.4 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs—*Concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management

General Terms: Design, Languages, Performance

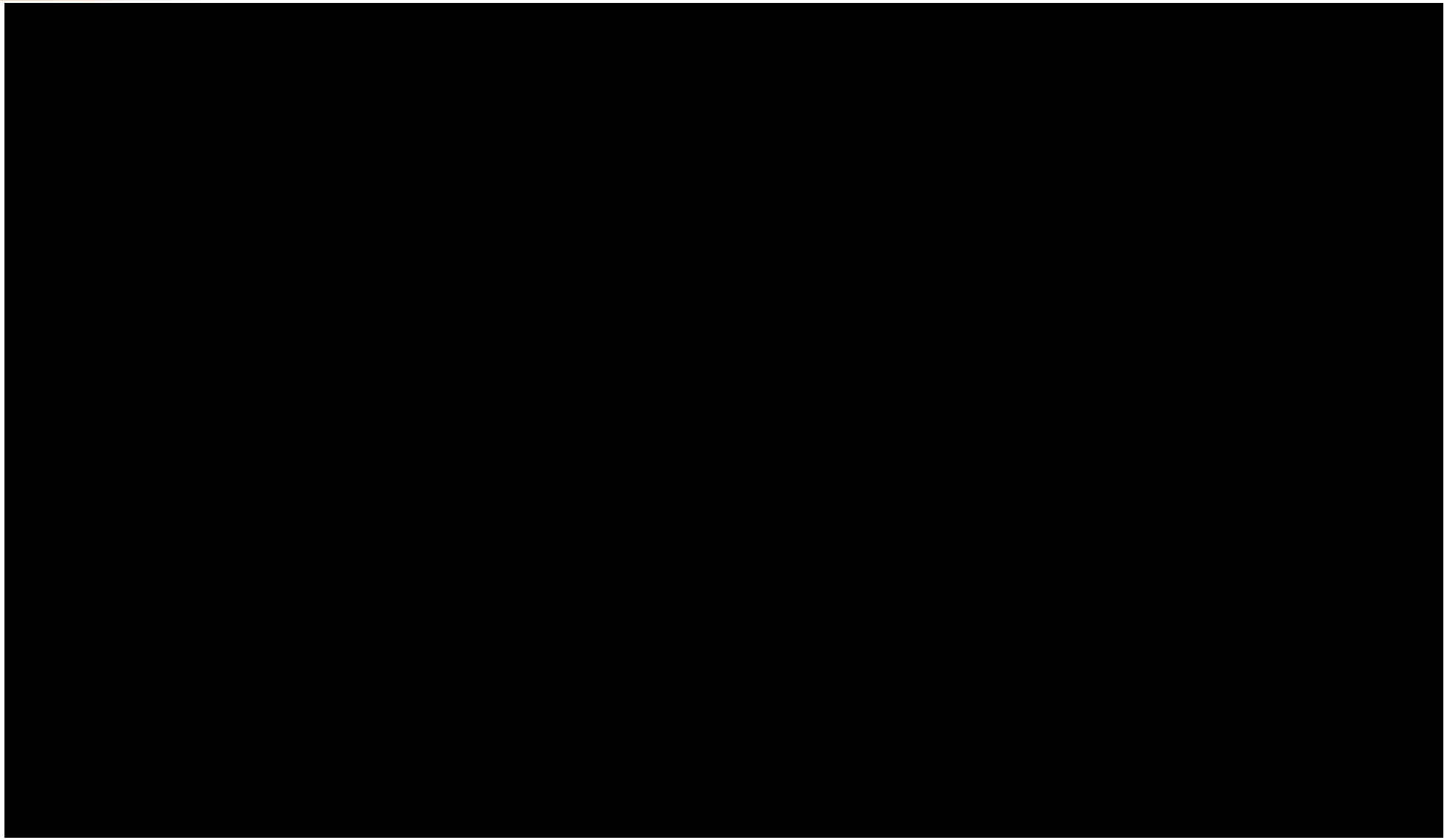
Additional Key Words and Phrases: Threads, Concurrency, Multi-processing, Synchronization

January 6, 1989

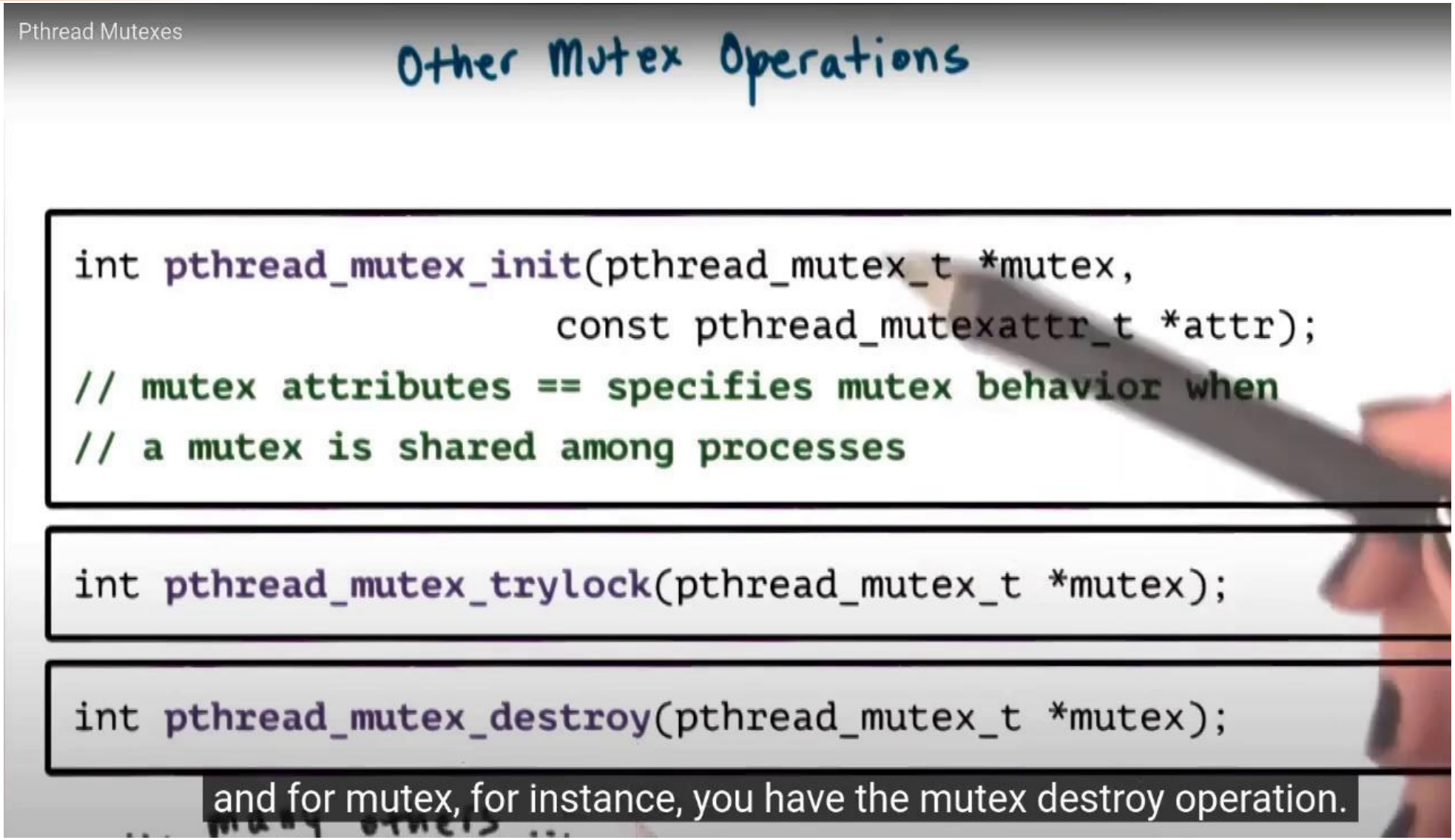
digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Mutex Locks



Pthread Mutexes



Topics Uncovered in this Session

- Mutex Locks - acquire(), release()
- Recursive Mutexes, Reader / Writer Mutexes, Spinlocks
- Pthread Mutexes

THANK YOU

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

nitin.pujari@pes.edu

For Course Deliverables by the Anchor Faculty click on www.pesuacademy.com