Lecture 15： Thread Library

1. **Threading Models（线程模型）**
    - **“Almost all current implementations” → one-to-one model**
        - 当前几乎所有系统（如 Linux、macOS、Windows）都采用 **一对一模型**（每个用户线程对应一个内核线程），因为它支持多核并行。
    - **many-to-one / many-to-many** 模型缺点：
        - many-to-one：不支持多核并行。
        - many-to-many：复杂、管理成本高。
2. **Concurrent programming challenges**
    - **“Placing a significant burden on developers to ensure implementation is free of race conditions and other bugs”**
        - 并发程序中容易出现 **race condition（竞争条件）**，编程调试复杂。
3. **OpenMP**
    - **编译器级隐式线程化（implicit threading）**，减少程序员手动管理线程的负担。
    - 指令如 `#pragma omp parallel` 自动生成多个线程。
4. **Pthreads (POSIX Threads)**
    - 核心函数：`pthread_create()`，`pthread_join()`，`pthread_exit()`，`pthread_cancel()`
    - Linux 实现：使用 **clone()** 系统调用创建线程，使用 **futex()** 实现同步。
    - **线程共享全局变量。**
5. **Thread cancellation**
    - **Deferred cancellation** 是默认且安全的方式。
    - 异步取消可能导致资源泄露。
6. **Thread pool & fork-join model**
    - **Thread pool**：控制线程数量，减少频繁创建/销毁的开销。
    - **Fork-join model**：同步版本的线程池，主线程等待子线程全部完成。

Lecture 16： Synchronization Intro

1. **Race Condition**
    - **当程序结果取决于线程执行顺序时** 就出现了竞争条件。
    - 核心原因：共享数据 + 非原子操作 + 不确定执行顺序。
2. **Critical Section（临界区）**
    - 定义：不能被多个线程同时执行的代码段。
    - 要求：
        - **Mutual Exclusion（互斥）**
        - **Liveness（活性）**
        - **Bounded Waiting（有界等待）**
3. **Too Much Milk Problem**
    - 通过三个尝试展示并发同步设计的困难。
    - 第三次尝试虽然正确，但：
        - 太复杂、非对称（代码不通用）、依赖 busy waiting。

Lecture 17： Hardware Support（硬件支持）

1. **Dekker’s Algorithm & Peterson’s Algorithm**
    - **Peterson’s Algorithm**：
        - 简化 Dekker 算法。
        - 只需两变量：`lock[i]` 与 `turn`。
        - **互斥成立的条件：**线程仅在 `(¬lock[j] ∨ turn == i)` 时进入临界区。
        - 具备 **安全性、活性、有界等待**。
2. **Correctness Conditions**
    - **Mutual Exclusion, Progress, Bounded Waiting** 是三大条件，必须会定义。
3. **Software-only Limitations**
    - **现代系统中不再可靠**：
        - 因为 CPU/编译器可能会 **指令重排**。
    - 必须引入硬件级支持（memory barrier 或原子指令）。

4. **Hardware Support**
    - **Memory barrier**：防止指令重排。
    - **Atomic instructions**：
        - `test_and_set`
        - `compare_and_swap`（CAS）
    - **Atomic variables**（C++ `std::atomic`）的底层原理基于这些原子操作。

Lecture 18： Synchronization Primitives

红字重点与考点

1. **Mutex Lock**
    - 定义：只能由一个线程持有，用于进入临界区。
    - 实现方式：
        - **Spinlock（忙等锁）**
        - **Blocking lock（阻塞锁）**
2. **实现机制**
    - **uniprocessor**：可用 `intr_disable()` / `intr_enable()`。
    - **multiprocessor**：必须使用 `test_and_set` 或 `compare_and_swap`。
3. **Busy Waiting 优化**
    - 使用 `guard` 变量减少自旋时间。
    - 让线程在锁被占用时进入休眠（`thread_block()`）。
4. **Condition Variable（条件变量）**
    - 操作：
        - `Wait(lock)`：释放锁并阻塞。
        - `Notify()`：唤醒一个等待线程。
        - `NotifyAll()`：唤醒所有等待线程。
    - 必须在持锁的情况下调用。
    - 常与 **bounded buffer / producer-consumer** 问题配合使用。

Lecture 19： Synchronization Primitives 2

1. **Semaphore（信号量）**
    - **Binary semaphore**：互斥锁。
    - **Counting semaphore**：条件同步。
    - 原子操作：
        - `Wait()`（P 操作）：值减 1，若 <0 则阻塞。
        - `Signal()`（V 操作）：值加 1，若 ≤0 则唤醒等待线程。
2. **Semaphore vs Condition Variable**
    - **信号量有记忆（memoryful）**；条件变量无记忆（memoryless）。
    - `signal()` 顺序问题不同步时会导致逻辑差异。
3. **Monitor（管程）**
    - 高级同步结构（Per Brinch Hansen 提出）。
    - 将 **数据 + 同步方法** 封装在类中。
    - 内部自带锁和条件变量。
    - **两种语义：**
        - **Mesa-style（常用）**：`signal` 后线程继续执行。
        - **Hoare-style**：`signal` 立即切换 CPU 给等待线程。
4. **Producer–Consumer Example**
    - 基于 Monitor 的信号量实现：`mutex`，`empty`，`full` 三个信号量。

模块	高频考点	关键术语
Thread Library	pthread API、fork-join、thread pool	pthread_create、join、cancel
Synchronization Intro	race condition、critical section	mutual exclusion、liveness
Hardware Support	Peterson、CAS、test_and_set	memory barrier、atomic
Sync Primitives I	Mutex、Condition Variable	wait/signal、bounded buffer
Sync Primitives II	Semaphore、Monitor	Mesa vs Hoare、producer-consumer