Synchronization Primitives

- Topics
  - Locks, spinlocks
  - Semaphores
  - Condition Variables
  - Monitors
- Learning Objectives:
  - Given a problem, select a suitable synchronization primitive
  - Identify poor choices of synchronization primitives
  - Explain how synchronizing applications is similar to/different from synchronizing inside an operating system.

Review: Locks

- Obtain a resource for exclusive use.
  - Acquire/Lock: Get the resource
  - Release/Unlock: Give up the resource
- Use case
  - Need to arbitrate exclusive access to a resource.
  - If resource is unavailable, want to wait for the resource.
  - **Same agent acquires/releases access to the resource**

Spinlock

- A very simple locking mechanism.
- Busy wait for resource to become available.
  - Atomically test if a resource is available and get it.
  - If it's not available, try again
- Requirements:
  - Requires some kind of hardware support (disabling interrupts or atomic instructions such as TAS, CAS)
- Assumptions:
  - True concurrency
  - Exclusive access
  - Short duration

Semaphore

- Counting and locking mechanism (shared counter).
  - A semaphore has a value that is always greater than or equal to 0.
  - You "acquire" a semaphore using an operation named P (for proberen which means "to test" in Dutch).
  - You "release" a semaphore using an operation named V (for verhogen, which means "increase" in Dutch).
- Semantics
  - V: Increment counter
  - P: Wait for counter to go positive and decrement
- Requirements:
  - P and V must be critical sections

Semaphore Usage

- Binary semaphore: similar to a lock:
  - Initialize the semaphore to 1.
  - The semaphore will only have the value 0 or 1.
  - Can be acquired/released by different parties.
  - P: locks resource
  - V: releases resource
  - Use when waiting is unlikely
- Counting semaphore: somewhat unique:
  - Schedule N fungible (interchangeable) resources.
  - Initialize the semaphore to N.
  - P: uses resource
  - V: frees reource
  - Allows up to N simultaneous users

Condition Variables (CV)

- A construct designed to let you run only when some condition about the world is true.
- Always paired with a lock (makes operations critical sections).
- API
  - `cv_create (cv_destroy)`: Create (Destroy) a condition variable
  - `cv_wait`: block until the condition becomes true
  - `cv_broadcast`: wake all threads waiting on this condition variable
  - `cv_signal`: wake a single thread waiting on this condition variable
- Use case:
  - Want to run when a condition is true
  - Not necessarily exclusive access
  - Condition is typically simple
  - Need to lock/wait atomically

CV Usage Pattern

- Usage:
  - Acquire lock
  - Check condition
  - If you need to wait on condition, call `cv_wait`.
  - Once condition is true, decide if you want to `cv_signal` or `cv_broadcast` information to others.
  - Release lock.
- Semantics:
  - Hoare semantics: If you wait on a condition, when you wake up you are **guaranteed that the condition is true**.
  - Mesa semantics: **No guarantees** when you wake; someone else may have beaten you to the punch.
  - OS161 uses Mesa semantics; you must code accordingly.

Monitors

- Higher order construct for synchronization.
- Provides API-level synchronization so programmers don't need to worry about them.
- Typically built into languages or libraries:
  - Java synchronized classes
  - C# classes that derive from Monitor
  - Ruby classes extended with MonitorMixin
- Use case:
  - Provide synchronized access to a data structure via its API.
  - Well defined API
  - Absolutely no manipulation or visibility outside of API

Kernel Synchronization Similarities

- Can use all the same primitives
- Same principles: critical sections, deadlocks, etc.
- Deadlocks are easier to debug than race conditions
- Same requirements:
  - Only one thread in a critical section.
  - Must make forward progress.
  - Activity outside a critical section cannot block the critical section.
  - Critical sections are short.
- Desirable properties:
  - Fair: if several processes are waiting, let each in eventually.
  - Efficient: don't use substantial amounts of resources when waiting. E.g., no busy waiting.
  - Simple: should be easy to use. E.g., just bracket the critical sections.

Kernel Synchronization Differences

- Somewhat similar to synchronizing with a server.
- Differences from synchronizing with normal user code:
  1. Must synchronize with hardware.
  2. Performing operations on behalf of someone else, so you don’t control what you do (e.g., you don’t necessarily know all the resources a particular thread is going to want).
  3. Must avoid deadlocks (finding deadlocks and killing things isn’t really OK).