Synchronization Primitives

- Learning Objectives (over the next couple of weeks):
  - Explain what each of the following synchronization primitives are and use them to solve synchronization problems
    - Locks
    - Binary semaphores
    - Counting semaphores
    - Condition variables
  - Select the right primitive to solve the problem
- Learning Objectives (immediately):
  - Understand these well enough that the interfaces to particular implementations we use make sense.

Locks

- Obtain a resource for exclusive use.
  - Sometimes referred to as a mutex (for mutual exclusion).
- Interface:
  - 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, wait for the resource.
- Semantics
  - **Same agent acquires/releases the lock.**

Lock Example

LOCK(B)

You	Your Banking Buddy

`B = get_balance();`	LOCK
	`B = get_balance();`
	BLOCK
	`// Withdraw $100`
	`B = B - 100;`

`// Withdraw $100` `B = B - 100;`	
`set_balance(B);`	`set_balance(B);`

UNLOCK(B)

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 (never blocks)
  - P: Wait for counter to go positive and decrement
- Requirements:
  - P and V are themselves critical sections

Semaphore Usage (1)

- Binary semaphore (similar, but not identical to, a lock)
- How to use a binary semaphore like a lock:
  - Initialize the semaphore to 1.
  - After initialization, the semaphore is always acquired and released in pairs, with acquisition (P) happening before a release (V).
  - P: locks resource
  - V: releases resource
- How to use a binary semphore for scheduling:
  - Initialize semaphore to 0.
  - Issue a P to block self.
  - Someone else comes along and does V to unblock.
- The only values the semaphore has are 0 and 1
  - Behaves very much like a lock, but
  - Can be acquired/released by different parties.

Binary Semaphore: Scheduling Example

You

```
S = create_semaphore(0);

// Block until work to do
S.P(); //blocks


// Wake after
// someone does a V
Do_Work();
```

Work Generator

```
// Work to do!  Better
// wake up server
S.V();
```

0 → 1

Semaphore Usage (2)

- Counting semaphore: somewhat unique (unlike other primitives):
  - Schedule N fungible (interchangeable) resources.
  - Initialize the semaphore to N.
  - P: uses resource
  - V: frees reource
  - Allows up to N simultaneous users

Counting Semaphore: Example

Your gym has 10 ellipticals; use a counting semaphore to grant access.

```
Init_elliptical(int N) { esem = create_semaphore(N); }
```

```
Get_elliptical() {

      esem.P();
}
```

```
Release_elliptical() {
      esem.V();
}
```

Condition Variables (CV)

- A construct designed to let you atomically check a condition and wait if the condition is not true.
- Paired with a mutex that protects the state that the condition checks.
- Interface
  - `cv_create (cv_destroy)`: Create (Destroy) a condition variable
  - `cv_wait`: block until the condition becomes true
  - `cv_broadcast`: wake eveyone waiting on this condition variable
  - `cv_signal`: wake one entity waiting on this condition variable
- Use case:
  - Want to run when a condition is true
  - Condition is typically simple
  - Need to check condition and wait atomically

CV Usage Pattern

- Usage:
  1. Acquire mutex
  2. Check condition
  3. If you need to wait on condition, call `cv_wait`.
  4. Once condition is true, decide if you want to `cv_signal` or `cv_broadcast` information to others.
  5. Release mutex
- 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.
  - pthreads uses Mesa semantics; you must code accordingly.
    - Typically, this means that condition checks appear in a while loop.

CV Example

- How might we do the, “Check if there is work on a work queue, and if so, let the server processes know.”

```
work_cv = create_cv();
work_mutex = create_mutex();
lock(work_mutex);
while (work queue is empty)
      cv_wait(work_cv,work_mutex);
// Now we can signal workers
cv_broadcast(work_cv);
unlock(work_mutex);
```

Wrapping Up

- Don’t stress if these don’t make perfect sense right now.
- We’ll get lots of practice with them over the next couple of weeks.
- Key points:
  - Concurrent code is typically incorrect if it is not synchronized.
  - There are multiple different tools you can use to perform synchronization.
  - Good design is all about picking the right primitive.