# Chapter 31 Semaphores

Single primitive for all things related to synchronization: can replace both

- Locks and
- Condition variables

Additional use of semaphores:

- Worker pool
  - Sometimes not desirable to create thousands or millions of goroutines
  - Instead we can limit the number of goroutines that can run at the same time
    - To some reasonable number, e.g., the number of CPU cores
  - Use a pool of “worker” goroutines/threads
    - Schedule work on these “workers"

**Def. Semaphore**: An object with an integer value that can be manipulated by

- sem.Wait() = P()
- sem.Post() = V()

Before using a semaphore:

- Initialize with some value
- sem := &Semaphore{value: 1}

Behavior of semaphore functions:

```
func (s *Semaphore) Wait() int {
	decrements the value of semaphore s by one
	wait if value of s is negative
}


func (s *Semaphore) Post() int {
	increments the value of semaphore s by one
	if there are one or more threads waiting, wake one
}
```

- Wait():
  - Returns right away if s.value > 1
  - Otherwise, suspends execution, waiting for a subsequent Post() call
    - Multiple calling threads: queued waiting to be woken up
- Post():
  - Increments s.value
  - If s has waiting threads, wake one up

Invariant of a semaphore object:

- s.value < 0 implies that s.value == #waiting threads

### 31.2 Binary Semaphore (Locks)

Use a semaphore as a lock:

```
sem := &Semaphore{value: X}  // initialize semaphore to X
sem.Wait();             // Lock()
// critical section
sem.Post();             // Unlock()
```

Q: What should X be?

X=1

Case 1: Thread 0 acquires and releases lock without Thread 1 interfering.

| Value of Semaphore | Thread 0 | Thread 1 |
|---|---|---|
| 1 | | |
| 1 | call `sem_wait()` | |
| 0 | `sem_wait()` returns | |
| 0 | `(crit sect)` | |
| 0 | call `sem_post()` | |
| 1 | `sem_post()` returns | |

Figure 31.4: **Thread Trace: Single Thread Using A Semaphore**

Case 2: Thread 0 holds the lock, when Thread 1 tries to enter critical section

| Value | Thread 0 | State | Thread 1 | State |
|---|---|---|---|---|
| 1 | | Running | | Ready |
| 1 | call `sem_wait()` | Running | | Ready |
| 0 | `sem_wait()` returns | Running | | Ready |
| 0 | `(crit sect: begin)` | Running | | Ready |
| 0 | *Interrupt; Switch→T1* | Ready | | Running |
| 0 | | Ready | call `sem_wait()` | Running |
| -1 | | Ready | `decrement sem` | Running |
| -1 | | Ready | `(sem<0)→sleep` | Sleeping |
| -1 | | Running | *Switch→T0* | Sleeping |
| -1 | `(crit sect: end)` | Running | | Sleeping |
| -1 | call `sem_post()` | Running | | Sleeping |
| 0 | `increment sem` | Running | | Sleeping |
| 0 | `wake(T1)` | Running | | Ready |
| 0 | `sem_post()` returns | Running | | Ready |
| 0 | *Interrupt; Switch→T1* | Ready | | Running |
| 0 | | Ready | `sem_wait()` returns | Running |
| 0 | | Ready | `(crit sect)` | Running |
| 0 | | Ready | call `sem_post()` | Running |
| 1 | | Ready | `sem_post()` returns | Running |

Figure 31.5: **Thread Trace: Two Threads Using A Semaphore**

**31.3 Semaphores for Ordering**

Semaphores can also be used to order events in a concurrent program

- Usage pattern
  - One thread waiting for something to happen
  - Another thread making that something happen
    - Signaling that it happened
    - Waking the waiting thread
- Similar to use of condition variables

Case 1: Parent thread continues to run and reach sem_wait() first

| Value | Parent | State | Child | State |
|---|---|---|---|---|
| 0 | `create(Child)` | Running | *(Child exists; is runnable)* | Ready |
| 0 | call `sem_wait()` | Running | | Ready |
| -1 | `decrement sem` | Running | | Ready |
| -1 | `(sem<0)`→`sleep` | Sleeping | | Ready |
| -1 | *Switch→Child* | Sleeping | `child runs` | Running |
| -1 | | Sleeping | call `sem_post()` | Running |
| 0 | | Sleeping | `increment sem` | Running |
| 0 | | Ready | `wake(Parent)` | Running |
| 0 | | Ready | `sem_post()` returns | Running |
| 0 | | Ready | *Interrupt; Switch→Parent* | Ready |
| 0 | `sem_wait()` returns | Running | | Ready |

Figure 31.7: **Thread Trace: Parent Waiting For Child (Case 1)**

Case 2: Child thread runs first reach sem_post() before parent runs sem_wait()

| Value | Parent | State | Child | State |
|---|---|---|---|---|
| 0 | `create(Child)` | Running | *(Child exists; is runnable)* | Ready |
| 0 | *Interrupt; Switch→Child* | Ready | `child runs` | Running |
| 0 | | Ready | call `sem_post()` | Running |
| 1 | | Ready | `increment sem` | Running |
| 1 | | Ready | `wake(nobody)` | Running |
| 1 | | Ready | `sem_post()` returns | Running |
| 1 | `parent runs` | Running | *Interrupt; Switch→Parent* | Ready |
| 1 | call `sem_wait()` | Running | | Ready |
| 0 | `decrement sem` | Running | | Ready |
| 0 | `(sem≥0)→awake` | Running | | Ready |
| 0 | `sem_wait()` returns | Running | | Ready |

Figure 31.8: **Thread Trace: Parent Waiting For Child (Case 2)**

**31.4 The Producer/Consumer Problem (Bounded Buffer)**

First Attempt: Use two semaphores

- Empty
- Full

MAX=1: This works even with multiple producers and multiple consumers

Next, let's assume multiple producers and multiple consumers when

MAX=10

Q: What can happen if MAX=10?

A: There is nothing that protects the critical section around get() and put()
so multiple threads can enter CS.

That is, sem.Wait() only waits when negative, and does not prevent another thread
from entering the CS.

**Example:** T1 and T2, both calling put() concurrently.

- Assume T1 runs first, and fill buffer (fillIndex = 0 on Line F1 in the Go code)
- T1 descheduled before Line F2
- T2 runs, and puts its data at the 0th element
  - Overwrite the data just written by T1
  - We have data loss!

**Solution: Add mutual exclusion!**

- Guard calls to put() and get() with locks (binary semaphore)
- However, our solution will cause a deadlock… Why?

**Example: Illustrate when our code can deadlock (Fig. 31.11)**

- C runs first
- Gets lock: mutex.Wait() ; Line c0
- Calls full.Wait() ; Line c1 — but there is nothing to consume
  - Yield the CPU, but still holds the lock
- Next, P runs to produce some data
  - Tries to get lock: mutex.Wait() ; Line p0
  - Lock already held; blocks!
- Simple cycle:
  - C holds *mutex*, waiting for someone to **signal** on the *full* semaphore
  - P could **signal** on *full* semaphore, but is **waiting** for the *mutex.*
- Both C and P are stuck waiting for each other!
  - Classic deadlock!

**31.5 Reader-Writer Locks**

Distinguish between operations that

- Read  (often much more frequent)
- Write

to a data structure.

Example: Concurrent list operations:

- Insert into list — update the list structure (WRITE)
- Lookup in the list — doesn't change the list structures (READ)

Since lookups don't change the list structure

- Can have many lookups run concurrently without causing problems since they are just reading that data
- Works if we can ensure that no writers run concurrently

*This is the job of a reader-writer lock.*

Pseudo-code for ReadWrite Lock:

```
func (rw *RWLock) AcquireReadLock() {
	wait for rw.lock: lock to access internal rwlock data structure
	readers++
	if readers == 1
		// first reader
		wait for the write lock
	release the rw.lock

}


func (rw *RWLock) ReleaseReadLock() {
	behavior is similar to acquire read lock
	except it releases the write lock when all readers are done

}
```

Discussion:

- First reader to get the writelock, essentially allows any reader to get the readlock too.
- A writer must thus wait for all readers to finish!

Problem with this implementation: Fairness

- Easy for readers to starve writers

A possible fix:

- Prevent more readers from entering lock once a writer is waiting

**31.6 The Dining Philosophers**

Intellectually stimulating / interesting:

- But practical utility is low
- Included because everyone should know about the problem

Setup:

- Five philosophers sitting around a table
- Single chopstick between each philosopher

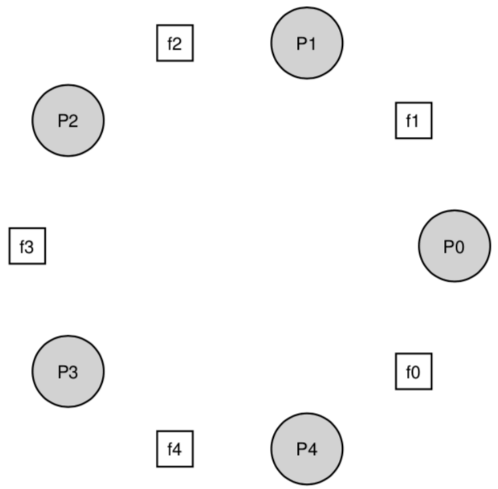


Figure 31.14: **The Dining Philosophers**

The philosophers alternate between

- Eating: need two chopsticks
  - One on its right and one on its left
- Thinking: don't need any chopsticks

```
while true {
        think()
        getforks()
        eat()
        putforks()
}
```

- Write getforks() and putforks()

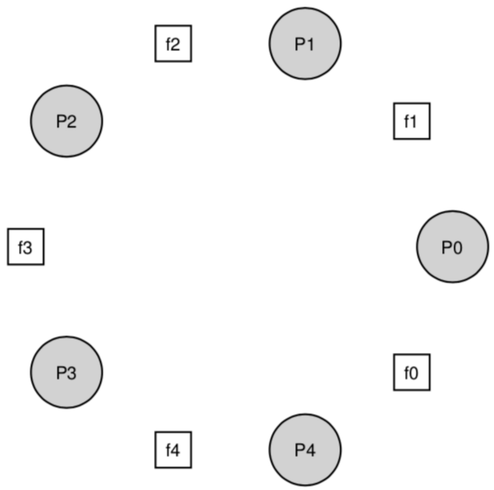


Figure 31.14: **The Dining Philosophers**

Requirements:

- No deadlock
- No philosopher starves (never gets to eat)
- Concurrency is high (as many philosophers can eat at the same time as possible)

Helper functions:

```
int left(int p)  {         return p;                        }
int right(int p) {         return (p+1) % 5;      }
```

Explain:

- left(): refers to the chopstick to the left of the philosopher *p*.
- right(): similar. Modulo operator allow last philosopher *p*=4 to get its right chopstick, which is 0.

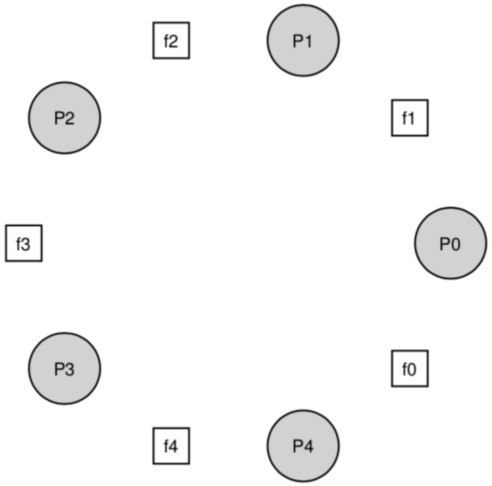


Figure 31.14: **The Dining Philosophers**

We will use five semaphores; one for each chopstick

```
func getforks() {
        sem.Wait(forks[left(p)]);
        sem.Wait(forks[right(p)]);
}

func putforks() {
        sem.Post(forks[left(p)]);
        sem.Post(forks[right(p)]);
}
```

First: pick up the left chopstick and the pick up the right chopstick.

**Deadlock:** *If each philosopher grab the fork on their left before any philosopher can grab the fork on their right, each will be stuck with one fork, waiting for another, forever.*

p0, f0, p1, f1, p2, f2, p3, f3, p4, f4 … deadlock.

All forks acquired.

All philosophers stuck waiting for another fork.

### A Solution: Breaking the Dependency

Make one of the philosophers pick up the chopsticks in a different order.

- Right, then left instead of
- Left the right.

Breaks the cycle of waiting.

```
void getforks() {
        if p == 4 {
                sem.Wait(forks[right(p)]);
                sem.Wait(forks[left(p)]);
        } else {
                sem.Wait(forks[left(p)]);
                sem.Wait(forks[right(p)]);
        }
}
```

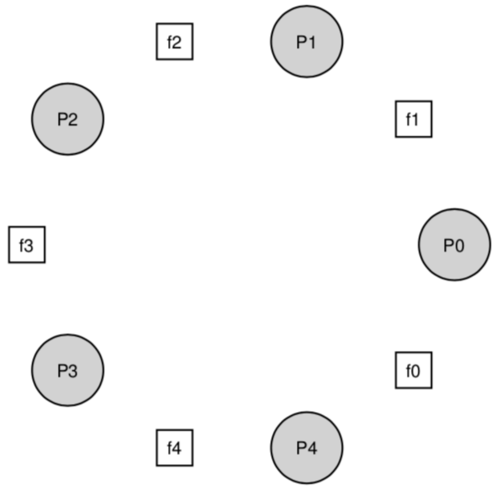


Figure 31.14: **The Dining Philosophers**

p0, f0, p1, f1, p2, f2, p3, f3, p4, f0 (must wait), p3, f4, … not deadlock

### 31.8 Summary

Semaphores can be viewed as a generalization of locks and condition variables

- Turns out building CVs using semaphores is difficult!
- Yet, some programmers use semaphores exclusively, because of their “simplicity” and utility.
- Arguably locks are simpler!!
- Condition variables are still a bit complex!
- Go channels are often much easier than CVs.
- Semaphores are useful for work load management!