Week 4: Low and High Level Synchronization Primitives

MPCS 52060: Parallel Programming

University of Chicago

Low Level Primitives

Practical Implementation of Spin Locks

We are going finish our discussion on lock implementations

- TAS Lock
- TTAS Lock
- Exponential Backoff Lock
- Queue Locks: Anderson, CLH, MCS, CLH Abort(Time-Out) Lock
- See the Lecture 3 Companion Slides 54-242

- A semaphore is generalization of a mutual exclusion locks.
- Each Semaphore has a capacity (c), which allows for having at most (c) threads in a critical section. Unlike with locks, where only one thread can be a critical section at a time.
- The capacity (c) is determined when the Semaphore is Initialized.

Motivation for Semaphores

- Resource Handling - a semaphore can be used to control access to a resource that has multiple instances.
  - Known as a *counting semaphore* - Initialize the semaphore to be equal to the number of available resources.
    - Example - login queue. A system can only handle a certain number of users concurrently signed on. After the maximum number of logged in users is reached, then others must wait until others logout.

Motivation for Semaphores (cont.)

- Producer-consumer problems - One thread generates tasks (the producer) and another thread receives and uses them (the consumer).
  - There are can be multiple producers and consumers.
  - They communicate using a queue of maximum size N and must adhere to the following conditions:
    - Consumers must wait for a producer to produce a task if the queue is empty.
    - Producer must wait for the consumer to consume a task if the queue is full.

Challenge - Think about how you might implement a problem like this using a semaphore.

Semaphore Pseudo-Implementation

The capacity variable of a Semaphore is an integer value that cannot be directly accessed.

- Go does not have a semaphore construct so the below examples are psudeo-code similar to the implementations of Semaphores in other languages:

- Creation: Must initialize it to some capacity integer value

```
var sema Semaphore
//semInit (s *Semaphore, value int)
semaInit(&sema, 0)
```


- It has two main operations that modify this integer value
  - *semaDown(s *Semaphore)*: Decrements semaphore
  - *semaUp(s *Semaphore)*: Increment semaphore

Semaphore Pseudo-Implementation

Implementation


```
func semaDown(s *Semaphore) {
    //Wait until value of semaphore s is greater than 0
    //Decrement the value of semaphore s by one
}
```



```
func semaUp(s *Semaphore) {
    //Increment the value of semaphore s by 1
    //If there are 1 or more threads waiting, wake one up
}
```

Semaphore Pseudo-Implementation & Mutual Exclusion

You can use a semaphore like a mutex

- Binary semaphore: initial value of 1

```
var mutex_sema Semaphore
semaInit(&mutex_sema, 1)
....
semaDown(&mutex_sema)
//critical section
semaUp(&mutex_sema)
```

You can also have mutual exclusion with more than one resource using a counting semaphore:

- The initial integer value is greater than one
- Initialize the semaphore to be equal to the number of available resources

Monitors

A monitor is an object with a set of monitor methods and only one thread may be active (i.e., accessing the monitor methods at a time):

- Provide in many object-oriented languages (e.g. Java)
- Can think of a monitor as one big lock for a set of operations/methods
- Typically the compiler automatically inserts lock and unlock operations upon entry and exit of monitor methods.


```
class Account {
    int balance;
    // synchronized is indicating this a monitor procedure
    // The compiler would insert a lock() at beginning and a
    // unlock() at the end of the method
    public synchronized void deposit() {
        ++balance;
    }
    public synchronized void withdraw() {
        --balance;
    }
};
```

Barriers

- Synchronizing threads to make sure that they all are at the same point in a program is called a barrier.
- No thread can cross the barrier until all the threads have reached it.

For example, you can use a barrier for debugging (Pseudocode):

```
point in program we want to reach;
barrier;
if goFD == 0 {
  fmt.Printf("All threads reached this point\n")
}
```

How could we implement a barrier? condition variables

- A condition variable is a data object that allows a thread to suspend execution until a certain event or condition occurs.
- When the event or condition occurs another thread can signal the thread to “wake up.”
- A condition variable is always associated with a mutex.

Condition Variables (Pseudocode)


```
lock mutex;
if condition has occurred
    signal thread(s);
else {
    // Wait until another signals to wake you up
    unlock the mutex and wait;
    /* After waking up, thread has acquired the lock. */
}
unlock mutex;
```

Condition Variables in Go

`sync.Cond` represents conditional variables in Go:

- Creation: (`NewCond(l Locker) *Cond`)

```
var mutex sync.Mutex
condVar := sync.NewCond(&mutex)
```

- Operations on condition variables:
  - (`func (c *Cond) Wait()`): suspends the calling thread and releases the monitor lock. When it resumes, reacquire the lock. Called when condition is not true
  - (`func (c *Cond) Signal()`): resumes one thread waiting in wait() if any. Called when condition becomes true and wants to wake up one waiting thread.
  - (`func (c *Cond) Broadcast()`): resumes all threads waiting in wait(). Called when condition becomes true and wants to wake up all waiting threads.

Barrier Example: Barrier

Live Demo: Simple Barrier implementation (see upstream repository week4/barrier/barrier.go)

High-level Primitives

See Companion Slides

- Linked-Lists: lec4_concurrent_linkedlists.pdf
- Queues & Stacks: lec4_concurrnet_queues_stacks.pdf