Semaphore Data Structure (with C++ interface)

```
class Semaphore
{
      int  value;                // counter
      Queue<Thread*> waitQ ;     // queue of threads blocked
                                 // this sema
      void Init(int v);          // initialization
      void P();                  // down(), wait()
      void V();                  // up(), signal ()
}
```

Dijkstra 1965

Initially developed as a “construct” to ease programming.

Alert → Dutch Lesson:

P = Probeer ('Try') and

V = Verhoog ('Increment', 'Increase by one’).

This version is based on multi-threaded capable OS or thread library
Older version used Process as the object , same principle

Semaphore implementation: Init()

```
void Semaphore::Init(int v)
{
     value = v;
     waitQ.init(); // empty queue
}
```

Semaphore implementations: P()

```
void Semaphore::P() //  or   wait()  or down()
{
     value = value - 1;
     if (value < 0)
     {
         waitQ.add(current_thread);

         current_thread->status = blocked;

         schedule();  // forces wait, thread blocked
     }
}
```

AKA “acquiring or grabbing the semaphore”

think of it as obtaining access rights

Semaphore implementations: V()

```
void Semaphore::V() //   or   signal()   or up()
{
    value = value + 1;
    if (value <= 0)
    {
       Thread *thd = waitQ.getNextThread();

       scheduler->add(thd); // make it scheduable
    }
}
```

AKA “releasing the semaphore”

Semaphore Solution

How do P() and V() avoid the race condition?

P() and V() must be atomic.

e.g.

- Using interrupts:
  - First line of P() & V() can disable interrupts.
  - Last line of P() & V() re-enables interrupts.

However: disabling interrupts only works on single CPU systems
Atomic lock variable an option on entry and exit,
but must release the lock as part of call to schedule() in P()
or must reacquire the lock as part of call to schedule() in V()

- Use lock variable with atomic operation

Semaphore Data Structure (with atomicity)

```
class Semaphore
{
     int  lockvar;                // to guarantee atomicity
     int  value;                  // counter
     Queue<Thread*> waitQ;        // queue of threads
                                  //  waiting on this sema
     void Init(int v);   // initialization
     void P();           // down(), wait()
     void V();           // up(), signal ()
}
```

Real Semaphore Implementation: P()

```
void Semaphore::P() //  or   wait()  or down()
{
      lock(&lockvar);
      value = value - 1;
      if (value < 0)
      {
           waitQ.add(current_thread);

           current_thread->status = blocked;

           unlock(&lockvar);

           schedule();  // forces wait, thread block

      } else {

           unlock(&lockvar);
      }
}
```

Real Semaphore Implementation: V()

```
void Semaphore::V() //   or   signal()   or up()
{
    lock(&lockvar);
    value = value + 1;
    if (value <= 0)
    {
       Thread *thd = waitQ.getNextThread();

       scheduler->add(thd); // make it scheduable
    }
    unlock(&lockvar);
}
```

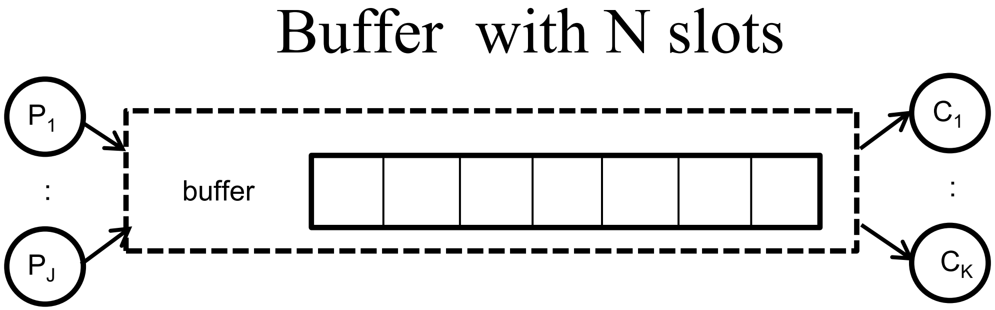
Two kinds of semaphores

- Mutex semaphores
  (or binary semaphores or LOCK):
  for mutual exclusion problems:
  value initialized to 1

- Counting semaphores:
  for synchronization problems.
  Value initialized to any value 0..N
  Value shows available tokens to enter or number or processes waiting when negative.

Let’s solve some interesting problems

- Producer-Consumer problem

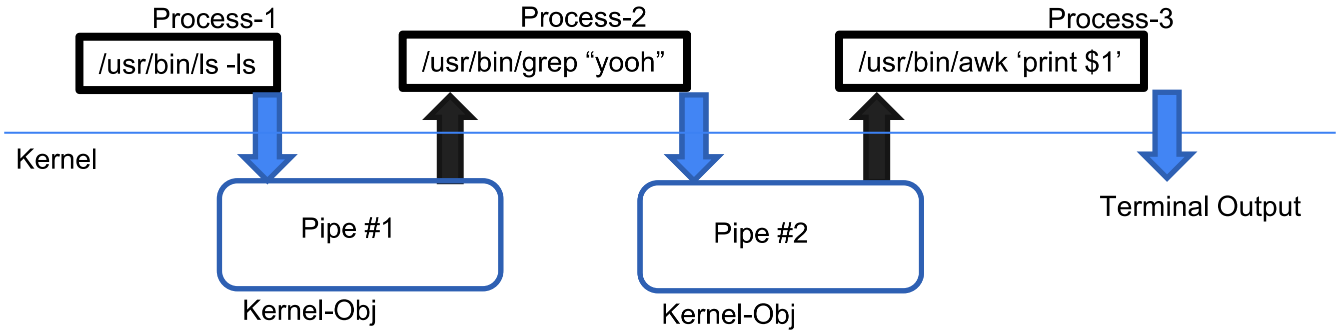


- Producer: (1 .. j)
  - How do I know a slot is free ?
  - How do I know a slot became free ?

- Consumer: (1 .. k)
  - How do I know nothing is available?
  - How do I know something became available ?

Where are N-buffer solutions required in an OSs?

- Example:
  - UNIX>   `ls -ls  |  grep "yooh" | awk '{ print $1 }'`



- Responsibility of a Pipe
  - Provide Buffer to store data from stdout of Producer and release it to stdin of Consumer
  - Block Producer when the buffer is full (because consumer has not consumed data)
  - Block Consumer if no data in buffer when the consumer wants to read(stdin)
  - Unblock Producer when buffer space becomes free
  - Unblock Consumer when buffer data becomes available

Semaphore Solution to the Producer Consumer Problem: 3 sem

LOCK

signaling

```
#define N <somenumber>

Semaphore empty = N;
Semaphore full  = 0;
Mutex mutex = 1;
```

3 Semaphores required

```
T buffer[N];
int widx = 0, ridx = 0;
```

```
Producer(T item)
{
     P(&empty);

     P(&mutex);     // Lock

     buffer[widx] = item;
     widx = (widx + 1 ) % N;

     V(&mutex);     // Unlock

     V(&full);
}
```

```
Consumer(T &item)
{
     P(&full);

     P(&mutex);     // Lock

     item = buffer[ridx];
     ridx = (ridx + 1 ) % N;

     V(&mutex);     // Unlock

     V(&empty);
}
```

Semaphore Solution to the Producer Consumer Problem: 3 sem

```
#define N <somenumber>

Semaphore empty = N;
Semaphore full  = 0;
Mutex mutex_w = 1;
Mutex mutex_r = 1;

T buffer[N];
int widx = 0, ridx = 0;
```

4 Semaphores required

Less contention by separating readers and writers

LOCK

signaling

```
Producer(T item)
{
     P(&empty);

     P(&mutex_w);     // Lock

     buffer[widx] = item;
     widx = (widx + 1 ) % N;

     V(&mutex_w);     // Unlock

     V(&full);
}
```

```
Consumer(T &item)
{
     P(&full);

     P(&mutex_r);     // Lock

     item = buffer[ridx];
     ridx = (ridx + 1 ) % N;

     V(&mutex_r);     // Unlock

     V(&empty);
}
```

Semaphore Solution to the Producer Consumer Problem: 4 sem

```
#define N <somenumber>

Semaphore empty = N;
Semaphore full  = 0;
Mutex mutex_w = 1;
Mutex mutex_r = 1;

T buffer[N];
int widx = 0, ridx = 0;
```

4 Semaphores required

LOCK

signaling

This example doesn't work; too aggressive !!!!!

```
Producer(T item)
{
     P(&empty);

     P(&mutex_w);     // Lock

     int wi = widx;
     widx = (widx + 1 ) % N;

     V(&mutex_w);     // Unlock
     buffer[wi] = item;

     V(&full);
}
```

Nasty race condition on wi

```
Consumer(T &item)
{
     P(&full);

     P(&mutex_r);     // Lock

     int ri = ridx;
     ridx = (ridx + 1 ) % N;

     V(&mutex_r);     // Unlock
     item = buffer[ri];

     V(&empty);
}
```

Barrier using Semaphores

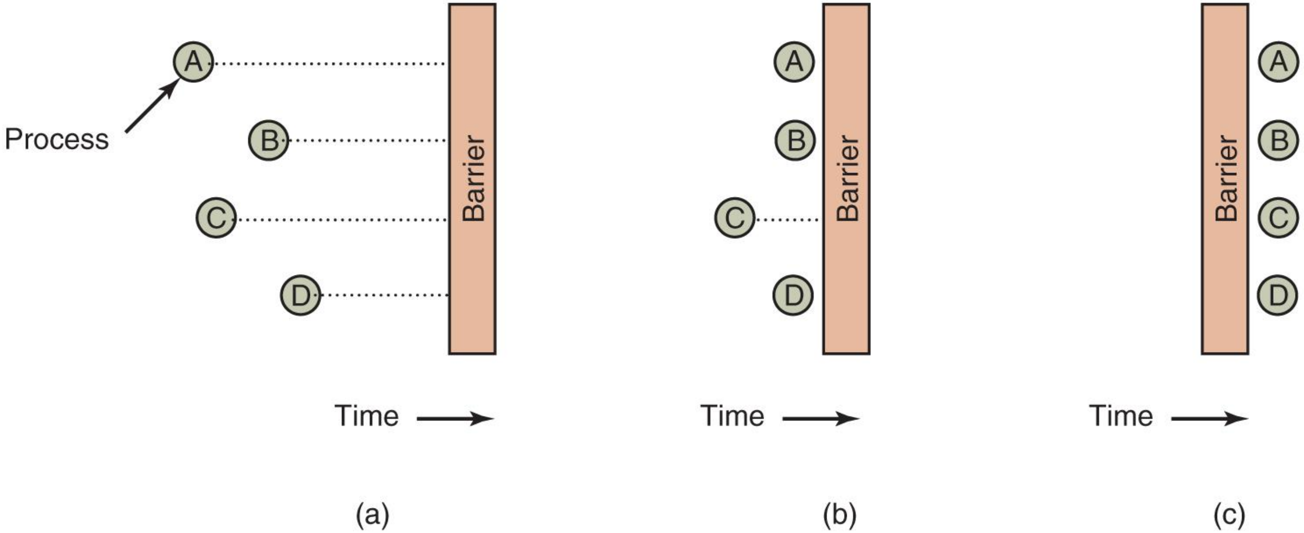
```
rendezvous
critical point
```

The synchronization requirement is that no thread executes **`critical point`** until after all threads have executed **`rendezvous`**.

You can assume that there are n threads and that this value is stored in a variable, n, that is accessible from all threads.

When the first n − 1 threads arrive they should block until the nth thread arrives, at which point all the threads may proceed.

Barriers : a graphical view



Use of a barrier.

(a) Processes approaching a barrier.

(b) All processes but one blocked at the barrier.

(c) When the last process arrives at the barrier, all of them are let through.

Barrier using Semaphores

```
rendezvous

sem_wait(mutex)
    count = count + 1
sem_post(mutex)

if count == n: sem_post(barrier)

sem_wait(barrier)

critical point
```

```
n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

Problem?

Barrier using Semaphores

```
rendezvous

sem_wait(mutex)
    count = count + 1
sem_post(mutex)

if count == n: sem_post(barrier)

sem_wait(barrier)
sem_post(barrier)

critical point
```

```
n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

Reusable Barrier using Semaphores

```
rendezvous

sem_wait(mutex)
    count = count + 1
sem_post(mutex)

if count == n: sem_post(barrier)

sem_wait(barrier)
sem_post(barrier)

critical point
```

```
n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

Problem?

Reusable Barrier using Semaphores

```
rendezvous

sem_wait(mutex)
    count = count + 1
    if count == n: sem_post(barrier)
sem_post(mutex)

sem_wait(barrier)
sem_post(barrier)

critical point
```

```
n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

Problem?

Reusable Barrier using Semaphores

```
rendezvous

sem_wait(mutex)
count = count + 1
if count == n: sem_post(barrier)
sem_post(mutex)

sem_wait(barrier)
sem_post(barrier)

critical point

sem_wait(mutex)
    count = count - 1
    if count == 0: sem_wait(barrier)
sem_post(mutex)
```

```
n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

Problem?

Reusable Barrier using Semaphores

```
rendezvous

sem_wait(mutex)
count = count + 1
if count == n:
     sem_wait(barrier2)
     sem_post(barrier)
sem_post(mutex)

sem_wait(barrier)
sem_post(barrier)

critical point

sem_wait(mutex)
     count = count - 1
     if count == 0:
          sem_wait(barrier)
          sem_post(barrier2)
sem_post(mutex)

sem_wait(barrier2)
sem_post(barrier2)
```

```
n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
barrier2 = Semaphore(1)
```

So which lock should I use ?

- Busy Lock vs. Mutexes vs Semaphores vs. …..

- If lock hold time is short and/or code is uninterruptible, then “lock variables with busy waiting” is OK (linux kernel uses it all the time)

- Otherwise use semaphores

  Note the two types of semaphores again
  - Mutex (Lock) and Counting (signaling/sync)

Lock Contention

- Lock Contention arises when a process/thread attempts to acquire a lock and the lock is not available.

- This is a function of
  - frequency of attempts to acquire the lock
  - Lock hold time (time between acquisition and release)
  - Number of threads/processes acquiring a lock

- Lock contention is a function of lock hold time and lock acquisition frequency.

- This can influence the lock type you are planning to use.

Monitors   (lets do it more automatic)

- Concurrency meets Object-Oriented Programming
- Handling/Programming Concurrency / Locking can be buggy
- Monitor
  - Object with a set of **monitor procedures** (i.e., methods)
  - Only one **active thread** at a time (i.e., monitor procedures are not concurrent)

How to implement monitor?

Compiler automatically inserts lock and unlock operations upon entry and exit of monitor procedures to create critical section

```
class account {
     int balance;

     public synchronized void deposit()
     {                                       ------>   lock(this.m);
        ++balance;                                     --balance;
     }                                       ------>   unlock(this.m);


     public synchronized void withdraw()
     {                                       ------>   lock(this.m);
        --balance;                                     --balance;
     }                                       ------>   unlock(this.m);
};
```

Condition Variables Revisited

- A monitor is 1 mutex + N cond var in a class object
  - In Java, it's 1 mutex + 1 condition variable
- Java Object methods for condition variable
  - wait(), notify(), notifyAll()
- Condition variables vs. Semaphores
  - Semaphores are **sticky,** but condition variables are not
  - But one can be implemented using the other

RCU: Lock-free Synchronozation

- Reader-writer lock still too slow, even for reading
  - Counter variable access needs expensive atomic instructions and memory barriers
  - Does not scale with large number of CPUs
- Can we just get rid of locks?
  - Sometimes we get lucky when we forget to lock
  - Can we just replicate the luck all the time?
- Read-Copy-Update (RCU):
  - Many readers + one writer can run simultaneously
  - Readers may read old, but consistent data
  - No lock!

RCU in a Nutshell: Add Spatial Dimension

```
struct foo {
    int a;
    int b;
} *global_foo;

// global_foo initialized elsewhere

DEFINE_SPINLOCK(foo_lock);
```

```
void get(int *p, int *q) {
    struct foo *copy_foo;

    // 1) begin reading (no lock)

    // 2) copy pointer once
    copy_foo = global_foo;

    // 3) access data using copy_foo
    *p = copy_foo->a;
    *q = copy_foo->b;

    // 4) end reading (no unlock)
}
```

```
void set(int x, int y){
    struct foo *new_foo = kmalloc(...);
    struct foo *old_foo;

    // 1) synchronize multiple writers
    spin_lock(&foo_lock);

    // 2) copy old pointer once
    old_foo = global_foo;

    // 3) update data
    new_foo->a = old_foo->a + x;
    new_foo->b = old_foo->b + y;

    // 4) switch pointer
    global_foo = new_foo;

    spin_unlock(&foo_lock);

    // 5) wait a bit for old readers

    // 6) free old struct
    kfree(old_foo);
}
```

RCU Core API

```
struct foo {
    int a;
    int b;
} *global_foo;

// global_foo initialized elsewhere

DEFINE_SPINLOCK(foo_lock);
```

```
void get(int *p, int *q) {
    struct foo *copy_foo;

    // 1) begin reading (no lock)
    rcu_read_lock();
    // 2) copy pointer once
    copy_foo =
        rcu_dereference(global_foo);
     // 3) access data using copy_foo
    *p = copy_foo->a;
    *q = copy_foo->b;

    // 4) end reading (no unlock)
    rcu_read_unlock();
}
```

```
void set(int x, int y){
    struct foo *new_foo = kmalloc(...);
    struct foo *old_foo;

    // 1) synchronize multiple writers
    spin_lock(&foo_lock);

    // 2) copy old pointer once
    old_foo = rcu_dereference_protected(
        global_foo, lockdep_is_held(&foo_lock));
    // 3) update data
    new_foo->a = old_foo->a + x;
    new_foo->b = old_foo->b + y;

    // 4) switch pointer
    rcu_assign_pointer(global_foo, new_foo);

    spin_unlock(&foo_lock);

    // 5) wait a bit for old readers
    synchronize_rcu();
    // 6) free old struct
    kfree(old_foo);
}
```

RCU (Toy) Implementations

#1: Using RW Lock

```
DEFINE_RWLOCK(global_rw_lock);

void rcu_read_lock(void) {
    read_lock(&global_rw_lock);
}

void rcu_read_unlock(void) {
    read_unlock(&global_rw_lock);
}

void synchronize_rcu(void) {
    write_lock(&global_rw_lock);
    write_unlock(&global_rw_lock);
}
```

#2: “Classic” RCU

```
void rcu_read_lock(void) {
    prermpt_disable[cpu_id()]++;
}

void rcu_read_unlock(void) {
    prermpt_disable[cpu_id()]--;
}

void synchronize_rcu(void)
{
    int cpu;

    for_each_possible_cpu(cpu)
        run_on(cpu);
}
```

RCU Today

- Linux kernel
  - TREE_RCU: high perf, super complex implementation of grace period handling
    - https://twitter.com/joel_linux/status/1175700053056512000/photo/1
  - Rich set of API
    - https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html#full-list-of-rcu-apis
- User-space implementations
  - Ex) C++ standard library
- Use cases beyond reader-writer paradigm
- References:
  - Paul McKenney's RCU home page: http://www2.rdrop.com/users/paulmck/RCU/
  - Kernel RCU doc: https://www.kernel.org/doc/html/latest/RCU/index.html
  - Linux RCU API as of 2019: https://lwn.net/Articles/777036/

Let’s attempt a dive into Linux Kernel Browsing

- **Spinlocks:** Small, fast, non-sleeping locks used when holding time is short, such as in interrupt handlers.
- **Mutexes:** Sleeping locks for mutual exclusion, offering better performance for longer critical sections where sleeping is allowed.
- **Semaphores:** Traditional locks that can allow multiple holders, used for complex synchronization.
- **Read/Write Locks (rwlock / rwsem):** Allow multiple simultaneous readers but only one exclusive writer.
- **Read-Copy-Update (RCU):** A mechanism for high-read-frequency scenarios, allowing readers to run without locks.

https://elixir.bootlin.com/linux/v6.19.3/source