



CSE 451: Operating Systems, 26au

L9: RW Locks, Hardware Synchronization Primitives

Instructor: Naomi Alterman

TAs:

Jonathan Trinh (Head)

Mahir Emran

Ted Guan

Jai Rungrachakanont

Soham Raut

Sagar Saby

Andrew Wood

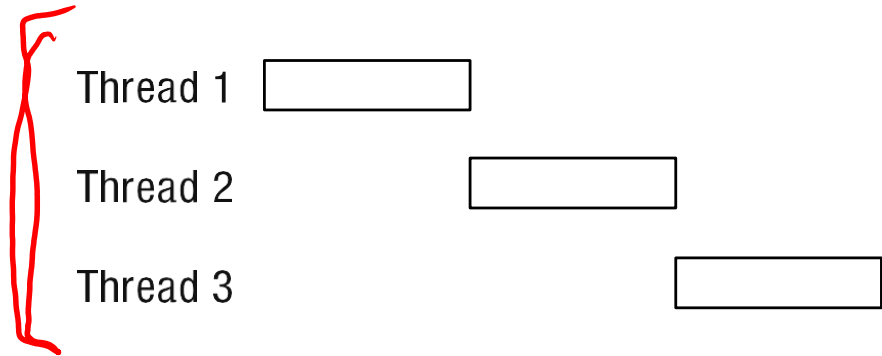
Adapted from slides by Tom Anderson

Administrivia

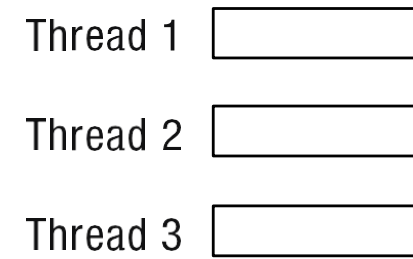
- Pset 2 out today, due 10/28

Concurrency

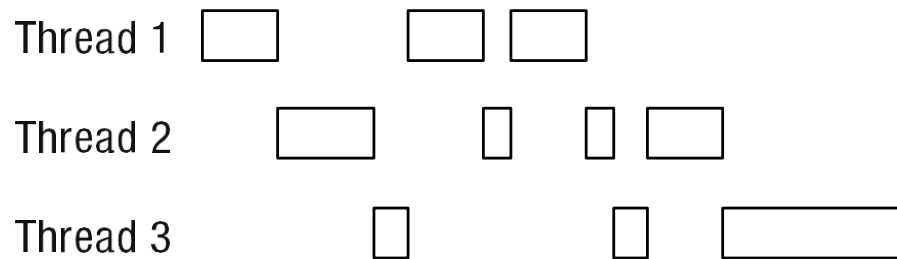
One Execution



Another Execution



Another Execution



R/W Locks

Readers/Writer Lock

- A more complicated lock to facilitate higher performance *where* ~~needed~~
- If a thread is only going to read shared data in its critical section, it's a **reader**
 - If a thread is going to modify shared data, it's a **writer**
 - Can have an **unlimited number of readers** at once
 - Can only have **one writer**, and the writer must **wait until all the readers are done**
- Interface:
 - ReaderAcquire(), ReaderRelease()
 - WriterAcquire(), WriterRelease()
 - ⚠ It's up to the programmer to distinguish between readers and writers! Be careful!! ⚠

R/W Lock Use Case

- **Hypothetical example:** working with the xk process list

 Acquire **read lock** to:

 Identify processes that have been asleep for a long time and prioritize switching to them

- Find the PID(s) of processes with a given name
 - Determine if a process has a given file open
 - ...
- Acquire **write lock** to:
 - Change the running state of a process (context switch)
 - ...

R/W Lock Use Case....?

- Working with the xk process list
- Acquire **read lock** to go through the process list and update a static global variable counting the number of open files

R/W Lock Implementation (1/3)

- State:

```
struct RwLock {  
    struct mutex lock;  
    struct condvar okToRead, okToWrite;  
    int waiting_readers, waiting_writers;  
    int active_readers, active_writers;  
};
```

R/W Lock Implementation (2/3)

```
void ReaderAcquire(struct RwLock *rw) {  
    rw->lock.Acquire();  
    while (rw->active_writers > 0 ||  
           rw->waiting_readers > 0)  
    {  
        rw->waiting_readers++;  
        rw->okToRead.wait(&rw->lock);  
        rw->waiting_readers--;  
    }  
    rw->active_readers++;  
    rw->lock.Release();  
}  
  
void WriterAcquire(struct RwLock *rw) {  
    rw->lock.Acquire();  
    while (rw->active_writers > 0 ||  
           rw->active_readers > 0)  
    {  
        rw->waiting_writers++;  
        rw->okToWrite.wait(&rw->lock);  
        rw->waiting_writers--;  
    }  
    rw->active_writers++;  
    rw->lock.Release();  
}
```

any writers at all

u_u

z z z

☺

☺

R/W Lock Implementation (3/3)

```
void ReaderRelease(struct RwLock *rw;) {  
    rw->lock.Acquire();  
    rw->active_readers--;  
    if (rw->active_readers == 0 &&  
        rw->waiting_writers > 0)  
    {  
        rw->okToWrite.Signal();  
    }  
    rw->lock.Release();  
}
```

```
void WriterRelease(struct RwLock *rw;) {  
    rw->lock.Acquire();  
    rw->active_writers--;  
    if (rw->waiting_writers > 0) {  
        rw->okToWrite.Signal();  
    } else if (rw->waiting_readers > 0) {  
        rw->okToRead.Signal();  
    }  
    rw->lock.Release();  
}
```

Going back to our proc example...

```
top - 19:33:10 up 19 days, 1:53, 837 users, load average: 1.88, 2.03, 2.31
Tasks: 1974 total, 2 running, 1970 sleeping, 0 stopped, 2 zombie
%Cpu(s): 0.7 us, 1.4 sy, 0.0 ni, 97.9 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 192048.8 total, 55781.1 free, 28759.7 used, 110638.8 buff/cache
MiB Swap: 196608.0 total, 194628.3 free, 1979.7 used. 163289.1 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2593767	root	20	0	53524	17000	6392	R	99.0	0.0	0:10.40	(fwupdmgr)
2318672		20	0	36.5g	107068	18556	S	6.1	0.1	13:19.86	zabbix_agent2
2593692	naomila	20	0	242448	8520	4540	R	3.2	0.0	0:01.91	top
1794	root	20	0	7531724	2.0g	1.5g	S	1.6	1.1	39,04	falcon-sensor-b
2445192		20	0	50.9g	526016	82952	S	1.0	0.3	6:52.06	MainThread
2547854		20	0	1551020	73824	50116	S	0.6	0.0	0:03.07	MainThread
106	root	20	0	0	0	0	S	0.3	0.0	0:08.85	rcu_exp_par_gp_kthread_worker/2
506540		20	0	29.4g	103284	32836	S	0.3	0.1	11:55.39	MainThread
1210312		20	0	154680	15772	11852	S	0.3	0.0	0:02.29	code-07f806f999
2445123		20	0	1742168	115316	55708	S	0.3	0.1	1:33.20	MainThread
2496841		20	0	1779876	133940	55096	S	0.3	0.1	0:06.08	MainThread
2496842		20	0	1561136	76448	50028	S	0.3	0.0	0:08.72	MainThread
2506752	root	20	0	0	0	0	I	0.3	0.0	0:00.03	kworker/38:0-events
2507631		20	0	9.8g	302124	59300	S	0.3	0.2	0:13.55	MainThread
2517743		20	0	1814692	176908	53572	S	0.3	0.1	0:06.98	MainThread
2518637		20	0	1777688	137880	53600	S	0.3	0.1	0:04.24	MainThread

top::getProcStats

A →
procLock.ReaderAcquire();
scrapeProcStatistics()
procLock.ReaderRelease();

bash::fork

procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();

A procLock

Lock	FREE
Ready list	A
active_writers	0
active_readers	0
waiting_readers	0
waiting_writers	0
okToRead Queue	/
okToWrite Queue	
Lock queue	

top::getProcStats

A→

```
procLock.ReaderAcquire();  
  
scrapeProcStatistics()  
  
procLock.ReaderRelease();
```

bash::fork

```
procLock.WriterAcquire();  
  
allocateNewPCB();  
  
procLock.WriterRelease();
```

procLock

Lock	HELD (A)
Ready list	A
active_writers	...
active_readers	...
waiting_readers	...
waiting_writers	...
okToRead Queue	...
okToWrite Queue	...
Lock queue	...

top::getProcStats

A

```
procLock.ReaderAcquire();
scrapeProcStatistics()
procLock.ReaderRelease();
```

bash::fork

```
procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();
```

procLock

Lock	<u>FREE</u>
Ready list	A
active_writers	0
active_readers	1
waiting_readers	0
waiting_writers	0
okToRead Queue	
okToWrite Queue	
Lock queue	

top::getProcStats

A

```
procLock.ReaderAcquire();
scrapeProcStatistics()
procLock.ReaderRelease();
```

bash::fork

```
procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();
```

procLock

Lock	FREE
Ready list	A
active_writers	0
active_readers	1
waiting_readers	0
waiting_writers	0
okToRead Queue	
okToWrite Queue	
Lock queue	

top::getProcStats

A

```
procLock.ReaderAcquire();
scrapeProcStatistics();
procLock.ReaderRelease();
```

bash::fork

B

```
procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();
```

procLock

Lock	FREE
Ready list	A, B
active_writers	0
active_readers	1
waiting_readers	0
waiting_writers	0
okToRead Queue	
okToWrite Queue	
Lock queue	

top::getProcStats

A

```
procLock.ReaderAcquire();
scrapeProcStatistics()
procLock.ReaderRelease();
```

bash::fork

B

```
procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();
```

procLock

Lock	HELD (B)
Ready list	A, B
active_writers	...
active_readers	...
waiting_readers	...
waiting_writers	...
okToRead Queue	
okToWrite Queue	
Lock queue	

top::getProcStats

A

```
procLock.ReaderAcquire();
scrapeProcStatistics()
procLock.ReaderRelease();
```

bash::fork

B

```
procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();
```

procLock

Lock	FREE
Ready list	A
active_writers	0
active_readers	1
waiting_readers	0
waiting_writers	1
okToRead Queue	
okToWrite Queue	B
Lock queue	

top::getProcStats

A

```
procLock.ReaderAcquire();
scrapeProcStatistics()
procLock.ReaderRelease();
```

bash::fork

B

```
procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();
```

procLock

Lock	FREE
Ready list	A
active_writers	0
active_readers	1
waiting_readers	0
waiting_writers	1
okToRead Queue	
okToWrite Queue	B
Lock queue	

top::getProcStats

```
procLock.ReaderAcquire();
scrapeProcStatistics()
procLock.ReaderRelease();
```

bash::fork

```
procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();
```

procLock

Lock	HELD (A) 
Ready list	A
active_writers	...
active_readers	...
waiting_readers	...
waiting_writers	...
okToRead Queue	
okToWrite Queue	B
Lock queue	

top::getProcStats

```
procLock.ReaderAcquire();  
scrapeProcStatistics()  
procLock.ReaderRelease();
```

A →

bash::fork

```
procLock.WriterAcquire();  
allocateNewPCB();  
procLock.WriterRelease();
```

B →

procLock

Lock	FREE
Ready list	A, B
active_writers	0
active_readers	0
waiting_readers	0
waiting_writers	1
okToRead Queue	
okToWrite Queue	
Lock queue	

top::getProcStats

procLock.ReaderAcquire();
scrapeProcStatistics()
A → procLock.ReaderRelease();

bash::fork

B → procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();

network::recv_msg

C → procLock.WriterAcquire();
wakeUpFromSleep(discord);
procLock.WriterRelease();

procLock

Lock	FREE
Ready list	A, B, C
active_writers	0
active_readers	0
waiting_readers	0
waiting_writers	1
okToRead Queue	
okToWrite Queue	
Lock queue	

top::getProcStats

procLock.ReaderAcquire();
scrapeProcStatistics()
A → procLock.ReaderRelease();

bash::fork

B → procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();

network::recv_msg

C → procLock.WriterAcquire();
wakeUpFromSleep(discord);
procLock.WriterRelease();

procLock

Lock	HELD (C)
Ready list	A, B, C
active_writers	...
active_readers	...
waiting_readers	...
waiting_writers	...
okToRead Queue	
okToWrite Queue	
Lock queue	

top::getProcStats

procLock.ReaderAcquire();
scrapeProcStatistics()
A → procLock.ReaderRelease();

bash::fork

B → procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();

network::recv_msg

C → procLock.WriterAcquire();
wakeUpFromSleep(discord);
procLock.WriterRelease();

procLock

Lock	FREE
Ready list	A, B, C
active_writers	0
active_readers	0
waiting_readers	0
waiting_writers	1
okToRead Queue	
okToWrite Queue	
Lock queue	

top::getProcStats

procLock.ReaderAcquire();
scrapeProcStatistics()
A → procLock.ReaderRelease();

bash::fork

B → procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();

network::recv_msg

C → procLock.WriterAcquire();
wakeUpFromSleep(discord);
procLock.WriterRelease();

procLock

Lock	HELD (B)
Ready list	A, C
active_writers	...
active_readers	...
waiting_readers	...
waiting_writers	...
okToRead Queue	
okToWrite Queue	B
Lock queue	

top::getProcStats

procLock.ReaderAcquire();
scrapeProcStatistics()
procLock.ReaderRelease();

bash::fork

B → procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();

network::recv_msg

C → procLock.WriterAcquire();
wakeUpFromSleep(discord);
procLock.WriterRelease();

procLock

Lock	FREE
Ready list	A, C
active_writers	1
active_readers	0
waiting_readers	0
waiting_writers	1 ☹️
okToRead Queue	
okToWrite Queue	B
Lock queue	

top::getProcStats

procLock.ReaderAcquire();
scrapeProcStatistics()
A → procLock.ReaderRelease();

bash::fork

B → procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();

network::recv_msg

C → procLock.WriterAcquire();
wakeUpFromSleep(discord);
procLock.WriterRelease();

procLock

Lock	FREE
Ready list	A, C
active_writers	1
active_readers	0
waiting_readers	0
waiting_writers	1
okToRead Queue	
okToWrite Queue	B
Lock queue	

top::getProcStats

```
procLock.ReaderAcquire();  
scrapeProcStatistics()  
A → procLock.ReaderRelease();
```

bash::fork

```
B → procLock.WriterAcquire();  
allocateNewPCB();  
procLock.WriterRelease();
```

network::recv_msg

```
procLock.WriterAcquire();  
wakeUpFromSleep(discord);  
C → procLock.WriterRelease();
```

procLock

Lock	HELD (C)
Ready list	A, C
active_writers	...
active_readers	...
waiting_readers	...
waiting_writers	...
okToRead Queue	
okToWrite Queue	B
Lock queue	

top::getProcStats

procLock.ReaderAcquire();
scrapeProcStatistics()
procLock.ReaderRelease();

bash::fork

procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();

network::recv_msg

procLock.WriterAcquire();
wakeUpFromSleep(discord);
procLock.WriterRelease();

procLock

Lock	FREE
Ready list	A, C, B
active_writers	0
active_readers	0
waiting_readers	0
waiting_writers	0
okToRead Queue	
okToWrite Queue	
Lock queue	

top::getProcStats

bash::fork

procLock

A→

```
procLock.ReaderAcquire();
scrapeProcStatistics()
procLock.ReaderRelease();
```

B→

```
procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();
```

network::recv_msg

C→

```
procLock.WriterAcquire();
wakeUpFromSleep(discord);
procLock.WriterRelease();
```

Lock	FREE
Ready list	A, C, B
active_writers	0
active_readers	0
waiting_readers	0
waiting_writers	0
okToRead Queue	
okToWrite Queue	
Lock queue	

top::getProcStats

bash::fork

procLock

A →
procLock.ReaderAcquire();
scrapeProcStatistics();
procLock.ReaderRelease();

B →
procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();

network::recv_msg

C →
procLock.WriterAcquire();
wakeUpFromSleep(discord);
procLock.WriterRelease();

Lock	HELD (A)
Ready list	A, C, B
active_writers	...
active_readers	...
waiting_readers	...
waiting_writers	...
okToRead Queue	
okToWrite Queue	
Lock queue	

top::getProcStats

bash::fork

procLock

A →
procLock.ReaderAcquire();
scrapeProcStatistics();
procLock.ReaderRelease();

B →
procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();

network::recv_msg

C →
procLock.WriterAcquire();
wakeUpFromSleep(discord);
procLock.WriterRelease();

Lock	FREE
Ready list	A, C, B
active_writers	0
active_readers	1
waiting_readers	0
waiting_writers	0
okToRead Queue	
okToWrite Queue	
Lock queue	

top::getProcStats

A →
procLock.ReaderAcquire();
scrapeProcStatistics();
procLock.ReaderRelease();

bash::fork

B →
procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();

network::recv_msg

C →
procLock.WriterAcquire();
wakeUpFromSleep(discord);
procLock.WriterRelease();

procLock

Lock	HELD (B)
Ready list	A, C
active_writers	...
active_readers	...
waiting_readers	...
waiting_writers	...
okToRead Queue	
okToWrite Queue	B
Lock queue	

top::getProcStats

bash::fork

procLock

A →
procLock.ReaderAcquire();
scrapeProcStatistics();
procLock.ReaderRelease();

B →
procLock.WriterAcquire();
allocateNewPCB();
procLock.WriterRelease();

network::recv_msg

C →
procLock.WriterAcquire();
wakeUpFromSleep(discord);
procLock.WriterRelease();

Lock	FREE
Ready list	A, C
active_writers	0
active_readers	1
waiting_readers	0
waiting_writers	1
okToRead Queue	
okToWrite Queue	B
Lock queue	

Concurrency Tragedy: **Starvation**



Concurrency Tragedy #1: Starvation

- There's nothing immediately wrong with our R/W Lock, but its implementation (combined with our scheduler) is **naïve**
 - Unlucky processes may get stuck continually waiting while other processes hop in line
 - In the extreme case this is called **resource starvation**: a thread never getting unblocked because The Algorithm always favors someone else
- Part of our job as OS developers is to understand *where* starvation can happen and try to mitigate it

R/W Lock, as-is

- Can readers starve?
 - Yes: writers take priority
- Can writers starve?
 - Yes: a waiting writer may not be able to proceed, if another writer slips in between signal and wakeup

R/W Starvation Mitigation, Try #1



```
void WriterAcquire(struct RwLock *rw) {  
    rw->lock.Acquire();  
    while (rw->active_writers > 0 || rw->active_readers > 0 ||  
           rw->waiting_writers > 0)  
    {  
        rw->waiting_writers++;  
        rw->okToWrite.wait(&rw->lock);  
        rw->waiting_writers--;  
    }  
    rw->active_writers++;  
    rw->lock.Release();  
}
```

NOPE

R/W Starvation Mitigation, Try #2

```
void WriterAcquire(struct RwLock *rw) {
    rw->lock.Acquire();
    int myPos = rw->acquireNum++;
    while (rw->active_writers > 0 ||
           rw->active_readers > 0 ||
           myPos > rw->nextToGo)
    {
        rw->waiting_writers++;
        rw->okToWrite.wait(&rw->lock);
        rw->waiting_writers--;
    }
    rw->active_writers++;
    rw->lock.Release();
}
```

```
void WriterRelease(struct RwLock *rw;) {
    rw->lock.Acquire();
    rw->active_writers--;
    rw->nextToGo++;
    if (rw->waiting_writers > 0) {
        rw->okToWrite.Broadcast();
    } else if (rw->waiting_readers > 0) {
        rw->okToRead.Broadcast();
    }
    rw->lock.Release();
}
```

.....but Naomi, what about reader starvation?

Low-level Synchronization Implementations

hhhhhow



OS as Glue

- We don't technically need an OS to provide synchronization primitives
- But:

“You should **never ever** think that you're clever enough to write your own locking routines.. Because the likelihood is that you aren't (and by that "you" I very much include myself - we've tweaked all the in-kernel locking over *decades*, and gone through the simple test-and-set to ticket locks to cacheline-efficient queuing locks, and even people who know what they are doing tend to get it wrong several times).

There's a reason why you can find *decades* of academic papers on locking. Really. It's hard.”

~Linus Torvalds, Semi-benevolent Linux Dictator [[source](#)]

Core Challenge

- Multi-processor systems are difficult to synchronize
- • Memory is duplicated across multiple levels and locations of cache
 - Instructions get reordered
 - Instructions aren't always atomic

- • Hardware needs to provide us some support
 - **atomic read-modify-write instructions**
 - Atomically read value from memory, operate, and then write result back to memory
 - Hardware prevents any other core from accessing that data
 - Hardware defers interrupts until instruction completes
 - **memory fence instructions**
 - Hardware forces all levels of cache to update
 - Hardware won't reorder instructions across fence

x86 Spinlocks

A spinlock waits in a loop for the lock to become free

- Assumes lock will be held for a short time

➔ Used to protect the CPU scheduler and to implement locks, CVs

- Implemented as a byte in memory that is 0 when free, 1 when held

#pointer to lock value in (%eax)

loop:

LOCK XCHGB (%eax), 1

JNZ loop

atomically swap args

Mem	Reg	Mem	Reg
1	0	0	1



Spinlocks and Interrupts

- Hey, suppose we're in an interrupt handler and need to access some shared data
 - To put a thread on the ready list (I/O completion)
 - To switch between threads (time slice)
- What happens if a thread gets interrupted while holding a spinlock?

Concurrency Tragedy #2: **Deadlock**

- Our interrupt handler is waiting on a lock another thread is holding
- That other thread is sleeping, waiting for our interrupt handler to complete
- Neither will ever wake up.
- AGAIN 🦴 💀 🪦 🪦 🌳
- xk mitigates this by disabling interrupts while spinlocks are held

(and we'll have much more to say about deadlock next class...)

xk's **kernel spinlock** implementation

- `kernel/spinlock.c`



Class concluded here on 10/19
and will resume on 10/21



Blocking Locks

- If a thread will be waiting on a lock for a while, we want the scheduler to put it to sleep until the lock is free again. This is a **blocking lock**/"sleeplock".
- Common implementation:
 - Use a spinlock to protect the blocking lock struct
 - Acquire spinlock, see if sleeplock is free
 - If not, go to sleep and tell scheduler to not wake us up until someone releases the sleeplock
 - Once we wake up, try to acquire the sleeplock again
- Note: the process list is also protected by its own spinlock
 - Need to manage hand-off carefully
 - Performance bottleneck

xk's **kernel blocking lock** implementation

- kernel/sleeplock.c
- kernel/proc.c
 - sleep()
 - wakeup()



Userspace locks

- Our implementation of spinlocks requires us to disable interrupts
 - Disabling interrupts is a  **PRIVILEGED INSTRUCTION** 
 - Hmm...
- Most real-life userspace locks are “blocking” and involve multiple kernel<->user context switches
 - Small optimization: spin for a little bit waiting for shared memory to be released, and then trip into the kernel if it takes too long
- Xk doesn't provide userspace synchronization

WDLD (what does Linux do 🐧 ?)

- <https://docs.kernel.org/locking/locktypes.html>

Semaphores

Semaphores

- A more general synchronization primitive (can implement both locks and condition variables with them)
 - Mathematical purity
 - Write once/cry once
- The data: **an atomically modifiable non-negative integer**
- The (atomic) operations:
 - **Up / “Signal” / “V”**: Increment the integer
 - **Down / “Wait” / “P”**: Wait until the integer is positive, and then decrement it

Semaphore varieties

- **Mutex / binary semaphore**
 - Used to represent single resource (eg, a critical section of code)
 - `up()` sets value to 1
- **Counting / general semaphore**
 - Used to represent a resource with many units, or a resource that allows concurrent access (e.g., reading)
 - Multiple threads can up/down the semaphore



Semaphore design patterns (1/2)

- Mutual exclusion:

```
semaphore sem(1); // initialize with value of 1
thread_func() {
    sem.down()
    // critical section
    sem.up()
}
```

Semaphore design patterns (2/2)

- Ordering constraints:
 - Example: thread A wants to wait for thread B to finish

```
semaphore sem(0); // initialize with value of 0
```

```
thread_A() {  
    // some work...  
    sem.down() // hold up!!  
    // more work...  
}
```

```
|  
:  
|  
:  
|  
:  
|  
:  
|
```

```
thread_B() {  
    // some work...  
    sem.up() // all good!!  
}
```