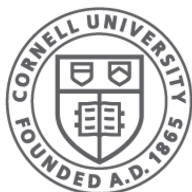
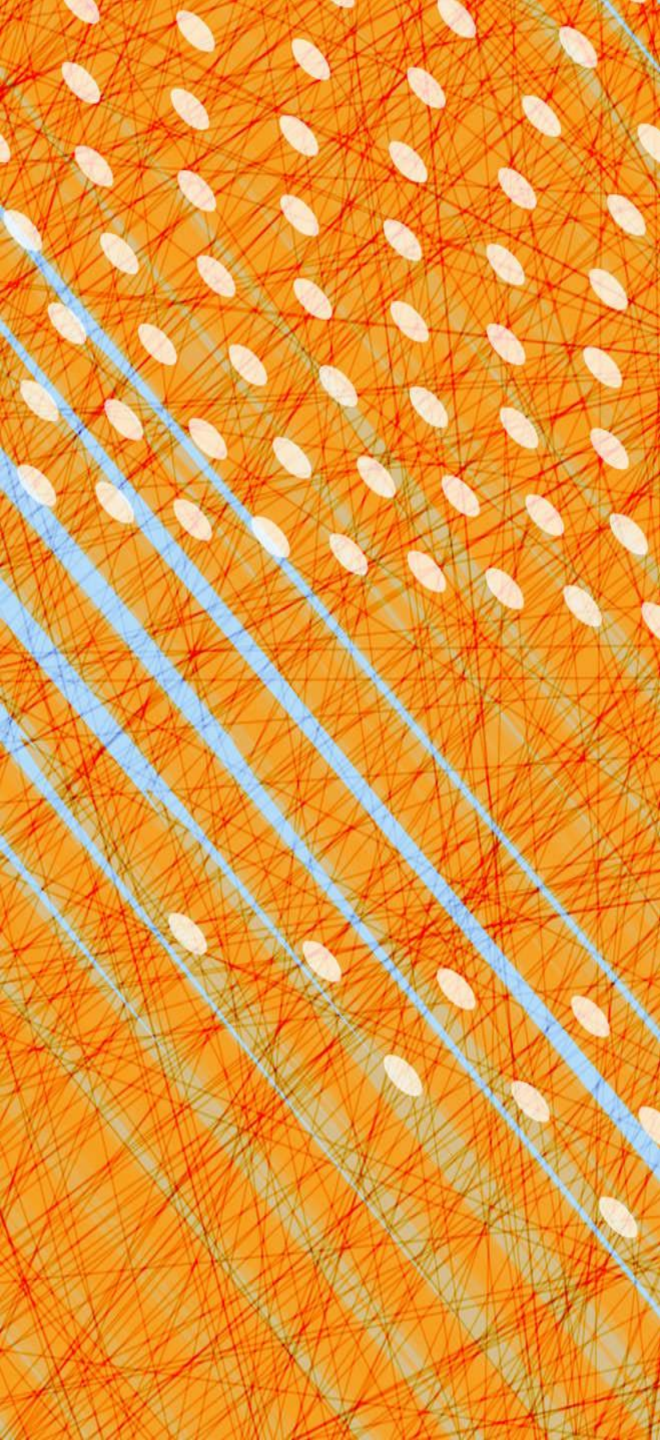
Synchronization

CS 4410
Operating Systems



Cornell CIS
COMPUTING AND INFORMATION SCIENCE

[R. Agarwal, L. Alvisi, A. Bracy, M. George, E. Sirer, R. Van Renesse]



- Foundations
- **Semaphores**
- Monitors & Condition Variables

Semaphores

- **Definition**
- Binary Semaphores
- Counting Semaphores
- Classic Sync. Problems (w/Semaphores)
  - Producer-Consumer (w/ a bounded buffer)
  - Readers/Writers Problem
- Classic Mistakes with Semaphores

What is a Semaphore?

[Dijkstra 1962]

Dijkstra introduced in the THE Operating System

Stateful:

- a **value** (incremented/decremented atomically)
- a queue
- a lock

Interface:

- Init(starting value)
- **P (procure)**: decrement, “consume” or “start using”
- **V (vacate)**: increment, “produce” or “stop using”

No operation to read the value!

Dutch 4410: P = Probeer (‘Try'), V = Verhoog ('Increment', 'Increase by one')

Semantics of P and V

P():

- wait until **value** >0
- when so, decrement **value** by 1

V():

- increment **value** by 1

```
P() {
    while(n <= 0)
        ;
    n -= 1;
}
```

```
V() {
    n += 1;
}
```

*These are the **semantics**,*

but how can we make this efficient?

(doesn't this look like a spinlock?!?)

Implementation of P and V

P():

- block (**sit on Q**) til n > `0`
- when so, decrement **value** by `1`

V():

- increment **value** by `1`
- **resume a thread waiting on Q (if any)**

```
P() {
    while(n <= 0)
        ;
    n -= 1;
}
```

```
V() {
    n += 1;
}
```

Okay this looks efficient, but how is this safe?

Implementation of P and V

P():

- block (**sit on Q**) til n > `0`
- when so, decrement **value** by 1

V():

- increment **value** by 1
- **resume a thread waiting on Q (if any)**

This is what TAS locks are good for!

```
P() {
    acquire(&guard);
    while(n <= 0) {
        waiting.enq(self);
        release(&guard);
        sleep();
        acquire(&guard);
    }
    n -= 1;
    release(&guard);
}
```

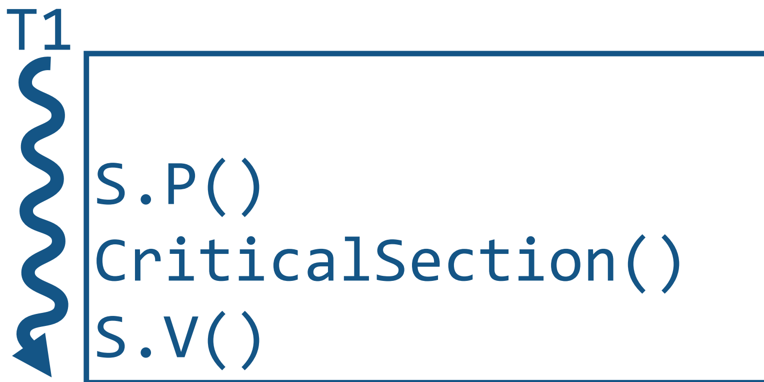
```
V() {
    acquire(&guard);
    n += 1;
    if(!waiting.empty()) {
       wake(waiting.deq());
    }
    release(&guard);
}
```

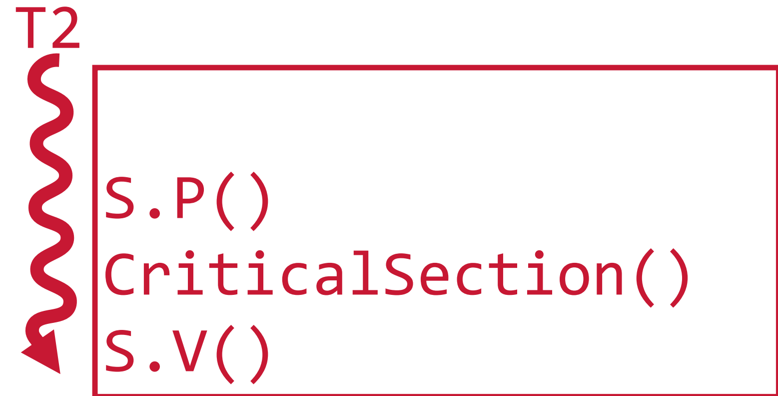
Binary Semaphore

Semaphore value is either 0 or 1

- Used for **mutual exclusion**
  (semaphore as a more efficient lock)
- Initially 1 in that case

```
Semaphore S
S.init(1)
```





Example: A simple mutex

```
Semaphore Lock
Lock.init(1)
```

```
Lock.P()
CriticalSection()
Lock.V()
```

```
P() {
    while(n <= 0)
        ;
    n -= 1;
}
```

```
V() {
    n += 1;
}
```

T1

```
Lock.P()
CriticalSection()
Lock.V()
```

T2

```
Lock.P()
CriticalSection()
Lock.V()
```

Counting Semaphores

Sema count can be any integer

- Used for signaling or counting resources
- Typically:
  - one thread performs P() to await an event
  - another thread performs V() to alert waiting thread that event has occurred

```
Semaphore packetarrived
packetarrived.init(0)
```

T1 ReceivingThread:

```
pkt = get_packet()
enqueue(packetq, pkt);
packetarrived.V();
```

T2 PrintingThread:

```
packetarrived.P();
pkt = dequeue(packetq);
print(pkt);
```

Semaphore's count:

- must be initialized!
- keeps state
  - reflects the sequence of past operations
  - `>0` reflects number of future P operations that will succeed

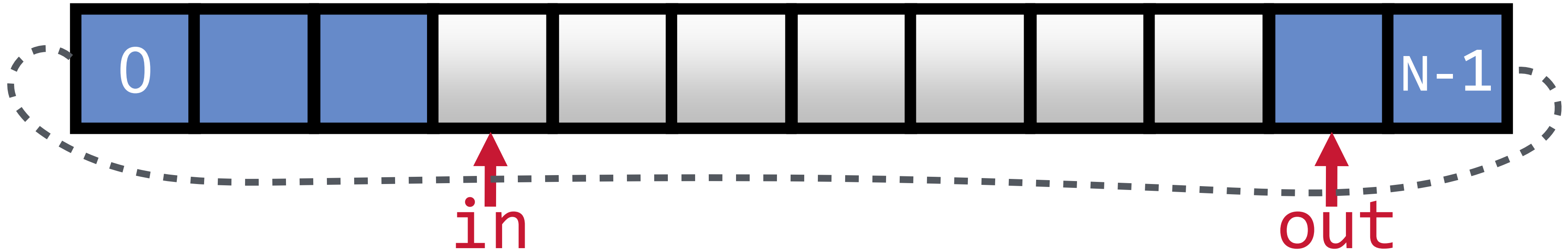
Not possible to:

- read the count
- grab multiple semaphores at same time
- decrement/increment by more than 1!

Producer-Consumer Problem

2+ threads communicate:

some threads **produce** data that others **consume**



Bounded buffer: size —**N entries**—

Producer process writes data to buffer

- Writes to **`in`** and moves rightwards

Consumer process reads data from buffer

- Reads from **`out`** and moves rightwards

Producer-Consumer Applications

- Pre-processor produces source file for compiler's parser
- Data from bar-code reader consumed by device driver
- File data: computer → printer spooler → line printer device driver
- Web server produces data consumed by client's web browser
- "pipe" (|) in Unix `>cat file | sort | more`

Starter Code: No Protection

```
Shared:
int buf[N];
int in, out;
```

```
// add item to buffer
void produce(int item) {
  buf[in] = item;
  in = (in+1)%N;
}
```

```
// remove item
int consume() {
  int item = buf[out];
  out = (out+1)%N;
  return item;
}
```

Problems:

1. Unprotected shared state (multiple producers/consumers)
2. Inventory:
   - Consumer could consume when nothing is there!
   - Producer could overwrite not-yet-consumed data!

Part 1: Guard Shared Resources

```
Shared:
int buf[N];
int in = 0, out = 0;
Semaphore mutex_in(1), mutex_out(1);
```

```
// add item to buffer
void produce(int item)
{
  mutex_in.P();
  buf[in] = item;
  in = (in+1)%N;
  mutex_in.V();

}
```

now atomic

```
// remove item
int consume()
{
  mutex_out.P();
  int item = buf[out];
  out = (out+1)%N;
  mutex_out.V();
  return item;
}
```

Part 2: Manage the Inventory

```
Shared:
int buf[N];
int in = 0, out = 0;
Semaphore mutex_in(1), mutex_out(1);
Semaphore empty(N), filled(0);
```

```
void produce(int item)
{
   empty.P(); //need space
   mutex_in.P();
   buf[in] = item;
   in = (in+1)%N;
   mutex_in.V();
   filled.V(); //new item!
}
```

```
int consume()
{
   filled.P(); //need item
   mutex_out.P();
   int item = buf[out];
   out = (out+1)%N;
   mutex_out.V();
   empty.V(); //more space!
   return item;
}
```

Sanity checks

1. **Is there a V for every P?**
2. **Mutex initialized to 1?**
3. **Mutex P&V in same thread?**

```
Shared:
int buf[N];
int in = 0, out = 0;
Semaphore mutex_in(1), mutex_out(1);
Semaphore empty(N), filled(0);
```

```
void produce(int item)
{
   empty.P(); //need space
   mutex_in.P();
   buf[in] = item;
   in = (in+1)%N;
   mutex_in.V();
   filled.V(); //new item!
}
```

```
int consume()
{
   filled.P(); //need item
   mutex_out.P();
   int item = buf[out];
   out = (out+1)%N;
   mutex_out.V();
   empty.V(); //more space!
   return item;
}
```

Producer-consumer: How did we do?

Pros:

- Live & Safe (& Fair)
- No Busy Waiting! *(is this true?)*
- Scales nicely

Cons:

- Still seems complicated: is it correct?
- Not so readable
- Easy to introduce bugs

Invariant

$0 \le in - out \le N$

```
Shared:
int buf[N];
int in = 0, out = 0;
Semaphore mutex_in(1), mutex_out(1);
Semaphore empty(N), filled(0);
```

```
void produce(int item)
{
   empty.P(); //need space
   mutex_in.P();
   buf[in%N] = item;
   in += 1;
   mutex_in.V();
   filled.V(); //new item!
}
```

```
int consume()
{
   filled.P(); //need item
   mutex_out.P();
   int item = buf[out%N];
   out += 1;
   mutex_out.V();
   empty.V(); //more space!
   return item;
}
```

Readers-Writers Problem

[Courtois+ 1971]

Models access to a database: shared data that some threads **read** and other threads **write**

At any time, want to allow:

- multiple concurrent readers *—OR—(exclusive)*
- only a single writer

Example: making an airline reservation

- Browse flights: web site acts as a **reader**
- Reserve a seat: web site has to **write** into database (to make the reservation)

Readers-Writers Specifications

N threads share **1** object in memory

- Some write: **1** writer active at a time
- Some read: **n** readers active simultaneously

Insight: generalizes the critical section concept

Implementation Questions:

1. Writer is active. Combo of readers/writers arrive.
   Who should get in next?
2. Writer is waiting. Endless of # of readers come.
   Fair for them to become active?

For now: back-and-forth turn-taking:

- If a reader is waiting, readers get in next
- If a writer is waiting, one writer gets in next

Readers-Writers Solution

```
Shared:
int rcount = 0;
Semaphore count_mutex(1);
Semaphore rw_lock(1);
```

```
void write() {
    rw_lock.P();
    . . .
    /* perform write */
    . . .
    rw_lock.V();
}
```

```
int read()
{
    count_mutex.P();
    rcount++;
    if (rcount == 1)
        rw_lock.P();
    count_mutex.V();
    . . .
    /* perform read */
    . . .
    count_mutex.P();
    rcount--;
    if (rcount == 0)
        rw_lock.V();
    count_mutex.V();
}
```

Readers-Writers: Understanding the Solution

If there is a writer:

- First reader blocks on **`rw_lock`**
- Other readers block on **`mutex`**

Once a reader is active, all readers get to go through

- Which reader gets in first?

The last reader to exit signals a writer

- If no writer, then readers can continue

If readers and writers waiting on rw_lock & writer exits

- Who gets to go in first?

Readers-Writers: Assessing the Solution

When readers active no writer can enter ✓

- Writers wait @ **`rw_lock`**.P()

When writer is active nobody can enter ✓

- Any other reader or writer will wait (where?)

Back-and-forth isn't so fair:

- Any number of readers can enter in a row
- Readers can "starve" writers

Fair back-and-forth semaphore solution is tricky!

- Try it! (don't spend too much time…)

Semaphores

- Definition
- Binary Semaphores
- Counting Semaphores
- Classic Sync. Problems (w/Semaphores)
  - Producer-Consumer (w/ a bounded buffer)
  - Readers/Writers Problem
- **Classic Mistakes with Semaphores**

Classic Semaphore Mistakes

```
P(S)
CS
P(S)
```

I

Gets stuck on 2nd P(). Subsequent processes freeze up on 1st P().

```
V(S)
CS
V(S)
```

J

Undermines mutex:

- Doesn't get permission via P()
- "extra" V()s allow other processes into the CS inappropriately

```
P(S)
CS
```

K

Next call to P() will freeze up. Confusing because the *other* process could be correct but hangs when you use a debugger to look at its state!

```
P(S)
if(x) return;
CS
V(S)
```

L

Conditional code can change code flow in the CS. Caused by code updates (bug fixes, etc.) by someone other than original author of code.

Semaphores Considered Harmful

“During system conception … we used the semaphores in two completely different ways. The difference is so marked that, looking back, one wonders whether it was really fair to present the two ways as uses of the very same primitives. On the one hand, we have the semaphores used for mutual exclusion, on the other hand, the private semaphores.”

— Dijkstra “The structure of the ‘THE’-Multiprogramming System” Communications of the ACM v. 11 n. 5 May 1968.

Semaphores NOT to the rescue!

These are “low-level” primitives. Small errors:

- Easily bring system to grinding halt
- Very difficult to debug

Two usage models:

- **Mutual exclusion:** “real” abstraction is a critical section
- **Communication:** threads use semaphores to communicate (*e.g.*, bounded buffer example)

Simplification: Provide concurrency support in compiler

→ Enter Monitors