Today

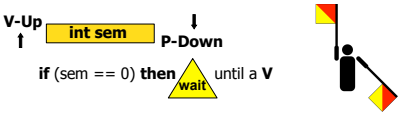
- Synchronization Mechanisms
  - Semaphores

Another synchronization mechanism

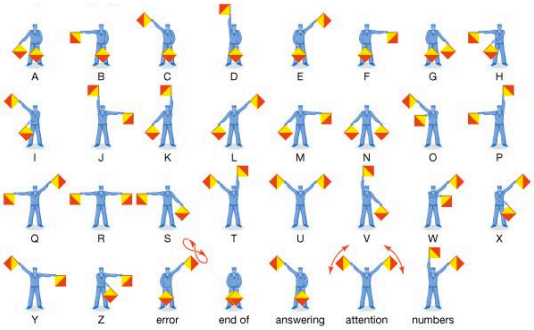
SEMAPHORE

Semaphore

- A ***semaphore*** is a hidden atomic integer counter with only increment/up (V) and decrement/down (P) operations.
  - Book calls V `signal` and P `wait`
- Decrement blocks iff the count is zero.
- Semaphores handle all of your synchronization needs with one elegant but confusing abstraction.

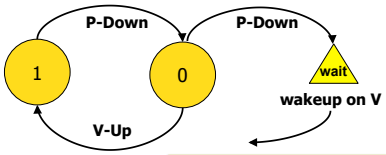


Semaphore – flag signals



Example: binary semaphore

- A binary semaphore takes only values 0 and 1.
- Requires a usage constraint:
  the set of threads using the semaphore call P and V in strict alternation.
  - Never two V in a row.

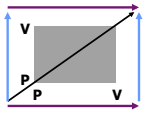


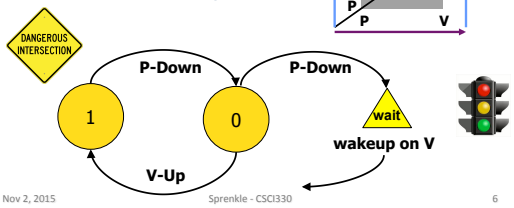
Typical initialization:
`Semaphore s = new Semaphore(1);`

A mutex is a binary semaphore

A ***mutex*** is a binary semaphore with an initial value of 1, for which each thread calls **P-V** in strict pairs.

Once a thread **A** completes its **P**, no other thread can **P** until **A** does a matching **V**.





Ping-pong with semaphores

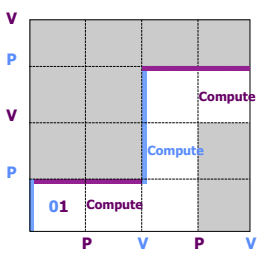
Ping-pong with semaphores

```
blue = Sempahore(0);
purple = Semaphore(1);
```

```
void
PingPong() {
   while(not done) {
      blue.P();
      Compute();
      purple.V();
   }
}
```

```
void
PingPong() {
   while(not done) {
      purple.P();
      Compute();
      blue.V();
   }
}
```


Ping-pong with semaphores



The threads compute in strict alternation.

Ping-pong with semaphores

```
blue = Sempahore(0);
purple = Semaphore(1);
```

```
void
PingPong() {
   while(not done) {
      blue.P();
      Compute();
      purple.V();
   }
}
```

```
void
PingPong() {
   while(not done) {
      purple.P();
      Compute();
      blue.V();
   }
}
```


Producer/Consumer with Semaphores?

- To start:
  - Just one resource produced/consumed

Basic producer/consumer

```
empty = Semaphore(1);
full = Semaphore(0);
int buf;
```

```
void Produce(int m) {
   empty.P();
   buf = m;
   full.V();
}
```

```
int Consume() {
   int m;
   full.P();
   m = buf;
   empty.V();
   return m;
}
```

This use of a semaphore pair is called a **split binary semaphore**: the sum of the values is always {0,1} – never more

Basic producer/consumer is called **rendezvous**:
one producer, one consumer, and one item at a time.
It is the same as ping-pong:
producer and consumer access the buffer in strict alternation.

Prod.-cons. with semaphores

- This time: more than one resource can be stored
- Same before-after constraints
  - If buffer empty, consumer waits for producer
  - If buffer full, producer waits for consumer
- What data structures/synchronization mechanisms should we use?