Classic Synchronization Problems

- Bounded-buffer problem
- Readers-writers problem
- Dining Philosophers problem
- ….

- We will compose solutions using semaphores

Mutex + Counting

- A lot of concurrency problems require tracking counts as conditions
  - Can we design a primitive that implements mutual exclusion while tracking counts?
- Yes – they are called semaphores – 旗語
  - After the visual signal method called "semaphores"

Semaphores

- You are given a data-type Semaphore_t.
- On a variable of this type, you are allowed
  - P(Semaphore_t) -- wait
  - V(Semaphore_t) – signal
- Intuitive Functionality:
  - Logically one could visualize the semaphore as having a counter initially set to 0.
  - When you do a P(), you decrement the count, and need to block if the count becomes negative.
  - When you do a V(), you increment the count and you wake up 1 process from its blocked queue if not null.

PENNSTATE

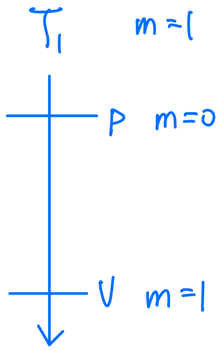
Semaphore Implementation

```
typedef struct {
  int value;
  struct TCB *L;
} semaphore_t;

void P(semaphore_t S) {
  Disable Interrupts/Use spinlock
  S.value--;
  if (S.value < 0) {
    add this thread to S.L and
      remove from ready queue
    context switch to another
  }
  Enable Interrupts/Use spinlock
}

void V(semaphore_t S) {
  Disable Interrupts/Use Spinlock
  S.value++;
  if (S.value <= 0) {
    remove a thread from S.L
    put it in ready queue
  }
  Enable Interrupts/Use Spinlock
}
```

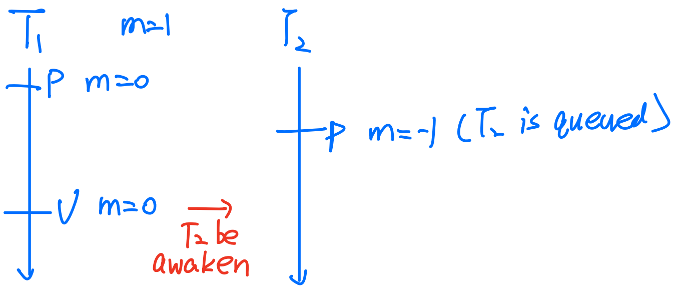
struct TCB.
a queue

Semaphores can implement mutex

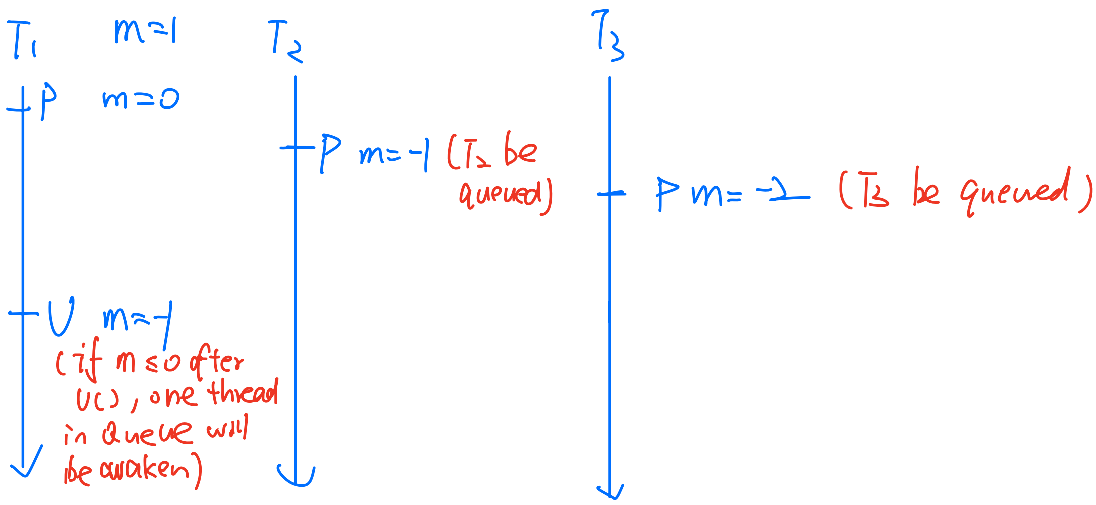
```
Semaphore_t m;    // Initialize its count/value to 1

Mutex_lock() {
  P(m);
}

Mutex_unlock() {
  V(m);
}
```



T_1
m=1
P m=0
V m=1



T_1
m=1
P m=0
V m=0
T_2 be awaken
T_2
P m=-1 (T_2 is queued)



T_1
m=1
P m=0
V m=-1
(if m ≤ 0 after V(), one thread in Queue will be awaken)
T_2
P m=-1 (T_2 be queued)
T_3
P m=-2 (T_3 be queued)

Bounded Buffer problem

- A queue of finite size implemented as an array.
- You need mutual exclusion when adding/removing from the buffer to avoid race conditions
- Also, you need to wait when appending to buffer when it is full or when removing from buffer when it is empty.

Bounded Buffer using Semaphores

```
int BB[N];
int count, head, tail = 0;
Semaphore_t m; // value initialized to 1
Semaphore_t notfull; // value initialized to N
Semaphore_t notempty; // value initialized to 0
```

```
Append(int elem) {
  P(notfull);
  P(m);

  BB[tail] = elem;
  tail = (tail + 1)%N;
  count = count + 1;

  V(m);
  V(notempty);
}
```

```
int Remove () {
  P(notempty);
  P(m);

  int temp = BB[head];
  head = (head + 1)%N;
  count = count - 1;

  V(m);
  V(notfull);
  return(temp);
}
```

PENNSTATE

Readers-Writers Problem

- There is a database to which there are several readers and writers.
- The constraints to be enforced are:
  - **When there is a reader accessing the database, there could be other readers concurrently accessing it.**
  - **However, when there is a writer accessing it, there cannot be any other reader or writer.**

Readers-writers using Semaphores

```
Database db;
int nreaders = 0;
Semaphore_t m; // value initialized to 1
Semaphore_t wrt; // value initialized to 1

Reader() {
  P(m);
  nreaders++;
  if (nreaders == 1) P(wrt);
  V(m);

  …. Read db here …

 P(m);
 nreaders--;
 if (nreaders == 0) V(wrt);
 V(m);
}

Writer() {
    P(wrt);

    … Write db here …

    V(wrt);
}
```

P(m)
nreader ++
if (nreader == 1) P(wrt)
V(m)

writers : wrt = 1

n readers : m = 1

(mutex)

P(m)

nreaders--

if (nreaders==0) P(wrt)

V(m)

① only 1 writer, no reader

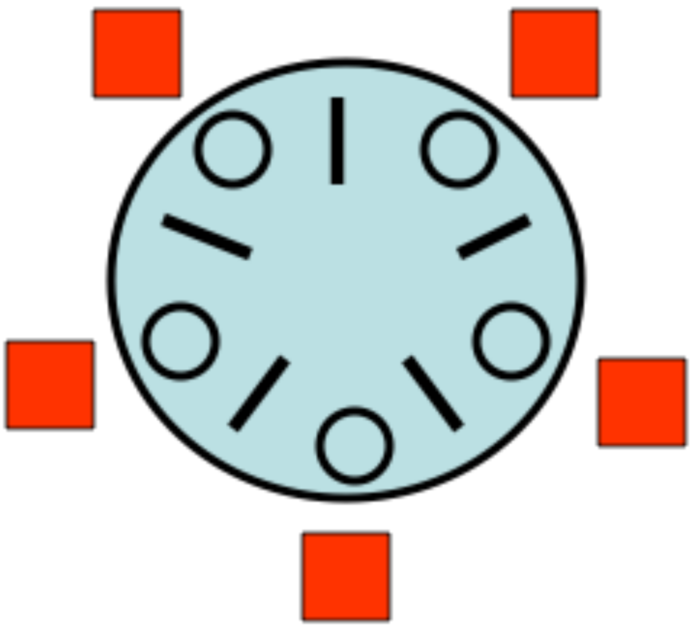
writer() go

② only 1 reader, can go reader(), no problem

$m: 0 \rightarrow 1 \rightarrow 0 \rightarrow 1$

② only 1 writer, 1 reader

Dining Philosophers Problem



Philosophers alternate between thinking and eating.

When eating, they need both (left and right) chopsticks.

A philosopher can pick up only 1 chopstick at a time.

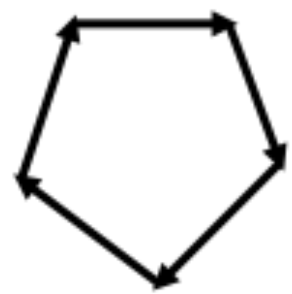
After eating, the philosopher puts down both chopsticks.

```
Semaphore_t chopstick[5];

Philosopher(i) {
  while () {
      P(chopstick[i]);
      P(chopstick[(i+1)%5];

      … eat …

      V(chopstick[i]);
      V(chopstick[(i+1)%5];

      … think …
  }
}
```

This is NOT correct!

Though no 2 philosophers use the same chopstick at any time, it can so happen that they all pick up 1 chopstick and wait indefinitely for another.

This is called a deadlock,



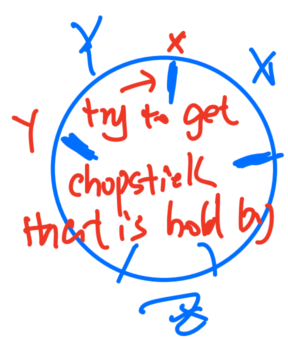
- Note that putting mutex

  P(chopstick[i]);

  P(chopstick[(i+1)%5];

  within a critical section (using say P(mutex)/V(mutex)) can avoid the deadlock.

- But then,
  - only 1 philosopher can pick up at a time (even if you don't depend on philosophers across the table).
  - Can still deadlock if you hold the mutex and block for a chopstick



try to get chopstick that is hold by X (in mutex)

mutex_lock(m)
P( )
P( ) ← Y
mutex_unlock(m)
mutex_lock() ← X
V( )

```
int state[N];
Semaphore_t s[N]; // init. to 0
Semaphore_t mutex; // init. to 1

#define LEFT    (i-1)%N
#define RIGHT   (i+1)%N

philosopher(i) {
  while () {
   take_chopsticks(i);
   eat();
   put_chopsticks(i);
          think();
  }
}
```

```
take_chopsticks(i) {
   P(mutex);
   state[i] = HUNGRY;
   test(i);
   V(mutex);
   P(s[i]);
}

put_chopsticks(i) {
   P(mutex);
   state[i] = THINKING;
   test(LEFT);
   test(RIGHT);
   V(mutex);
}
```

```
test(i) { /* can phil i eat? if so, signal that philosopher */
  if (state[i] == HUNGRY &&
     state[LEFT] != EATING && state[RIGHT] != EATING) {
     state[i] = EATING;
     V(s[i]);
  }
}
```