Lecture 7: Semaphores

- Programming with semaphores
  - Mutual exclusion
  - Counting
  - Signalling

- Implementing semaphores
Semaphore interface:
contains a non-negative counter (counting available resources)
init (value)

\[ P() \text{ decrement } \] " pits "
\[ V() \text{ increment } \] " release, signal "
(can go above initial value)

other non-standard operations to avoid:
- bounded semaphore
  - specify an upper bound
  - block if \( V() \) would go above
- read value of semaphore: dangerous!

\begin{verbatim}
if sema.get_value() > 0:
  \( P(sema) \) # know this won't block,
  since sema is available
\end{verbatim}

\( m \text{ python: acquire()} \)
\( m \text{ python: release()} \)
Semaphore design patterns:

1. Mutual exclusion:
   - Initialize semaphore to 1
   - Before a critical section: P(semaphore)
   - After: V(semaphore)
   - (only 1 thread in CS at a time)

2. Counting semaphore:
   - Initialize semaphore to # of available resources
   - To acquire resource: P
   - To release resource: V

3. Signalling:
   - Initialize semaphore to 0
   - To wait for a signal: P(semaphore)
   - To send signal: V(semaphore)

- Testing is very partial because scheduler is non-deterministic.
- Verification by other means is important.
- Need good code hygiene.
Example: Bounded buffer

Interface:

put (object)
"block until space available to add o, add o to buffer, return"

put (o1)
put (o4)
put (o5)
put (o6)
Obj get ()
"block until an object is in the buffer, remove & return object"
- What code needs
  reserved exclusive?
- What resources need
  to be cleaned?
- What errors do you
  want to send?

```java
with
    get:
      buffer_wait = new Channel()
      num_objects = new Spin<>()
      m = 0
      loop:
        if buffer_wait is not
        lock
          lock = new Spin<>()
          m = m + 1
          lock = lock
          get
          buffer_wait
          m = m - 1
          lock
          loop
        # wait when
        num_objects > 0
        yield
```

```java
// with
    put
      object
      P(num-empty)
      V(num-objects)
      V(signal-object-available)
```
Using signal semaphores:

```
threads = pool()

# thread represented by an object:
- identifier
- TCB
- semaphore allocated for this thread: (input_signal)

inside: thread, to wait for input:
P(self, input_signal)

inside: scheduler or I/O handler:
to give input to the thread: V(thread, input)
```

States:
- ready
- running
- waiting
- dev1 → [TCB] → TCB
- dev2
Implementing a semaphore

```python
class Semaphore:
    def __init__(self, value):
        self.value = Value(value)  # protected by lock
        self.lock = Lock()  # TAS spin lock
    def waiting = Queue()  # protected by lock

    def P(self):
        while not TAS(lock):
            # wait: either
            - pass  # multi-processor
            - or put this process in ready queue.
            - yield() # single processor
        acquired lock
        check value:
        if >0, decrement, release lock, return.
        if =0, put current thread on waiting state, (i.e. put it on a queue
        or wait for process for this
        sema.)
        deschedule this thread
        context switch.
```