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() decrement "probe"
- block until semaphore is at least 1 before decrementing.

in python: acquire()

V() . increment "release, signal"
(can go above initial value)

in python: release()

other non-standard operations to avoid:

- bounded semaphore:
  - specify an upper bound,
  - block if V() would go above

- read value of semaphore: dangerous!

```
if sema.getValue() > 0:
    P(sema)   # know this won't block, since sema is available
```

context switch

Semaphore design patterns:

call it lock.
or better yet: lock_resource.

① Mutual exclusion:
init semaphore to 1
before a crit. section: P(sema)
after: V(sema)
(only 1 thread in C.S. at a time)

think of semaphore as a lock
only one lock available.

call it num_resources.
num_printers

② counting semaphore:
init semaphore to # of available resources
to acquire resource: P
release resource: V

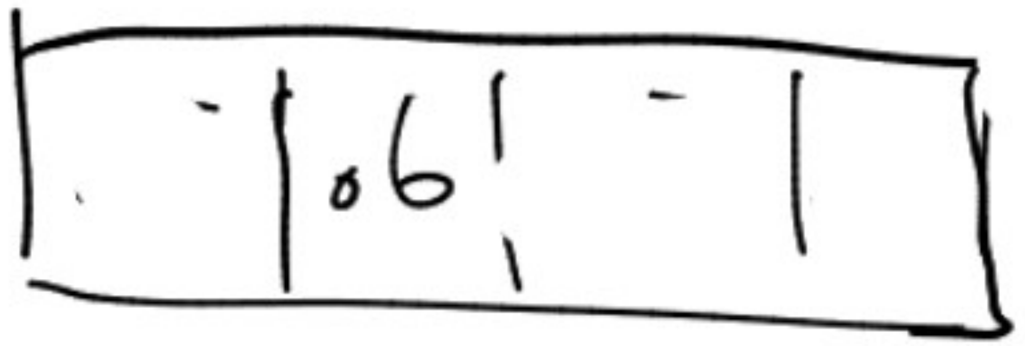
call it signal_description
signal_space_is_avail

③ signalling:
initialize semaphore to 0
to wait for a signal: P(sema)
to send signal: V(sema)

block until signal arrives

- testing is very partial because scheduler is non-deterministic.
- verification by other means is important
- need good code hygeine

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)

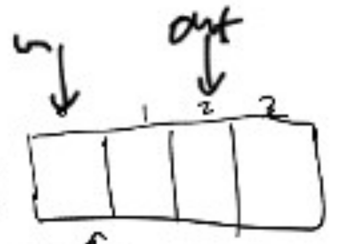
get() → o1
get() → o2
get() → o3
→ o4
→ o5
get() → o6

Obj get()

"block until an object
is in the buffer,
remove & return object"

- what code needs mutual exclusion?
- what resources need to be counted?
- what signals do you want to send?

init:

buffer = new Object[n]  n = 4

num_empty = new Sema(N)

num_objects = new Sema(0)

in = 0 # location of next object to put in the buffer, protected by lock

lock = new Sema(1)

~~signal_obj_avail = new Sema(0) # V'ed when num_objects is > 0~~

put(Object o)

switch?

P(num_empty)

P(lock)

buffer[in ++] = o — mod n, so loop around

V(lock)

V(num_objects)

~~V(signal_object_available)~~

get:

P(num_objects)

P(lock)

result = buffer[in-- (mod n)]

buffer[in] = null

V(lock)

V(num_empty)

alternately:

V(num_empty)

V(lock)

doesn't matter

holds / wait

P(lock)

P(number_empty)

:

V(number_obj)

get

1 → P(lock)

2 → P(number_objects)

Deadlock situation!

get can't proceed until put happens, but put can't happen while get is holding lock.

Using signalling 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_sig)

ready ⇝

running ⇝

waiting ⇝ dev1 → $\boxed{\text{TCB}}$ → TCB.

⇝ dev2

Implementing a semaphore

(Full impl next lec)

```
class Semaphore:
    def __init__(self, value)
        self.value = value   # protected by lock
        self.lock = 0        # TAS spin lock
        self.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 into waiting
                   state, (i.e. put it on a queue
                   of waiting procs for this
                   sema.)
            deschedule this thread
            context switch.
```