# 4410 Lecture 8: monitors

- (finish semaphore impl)
- Monitor design pattern
- (quiz)

# Using semaphores for thread control (per-thread sema)

To sleep current thread:

- allocate new semaphore s
- put s in a data structure
- P(s) ← decrement

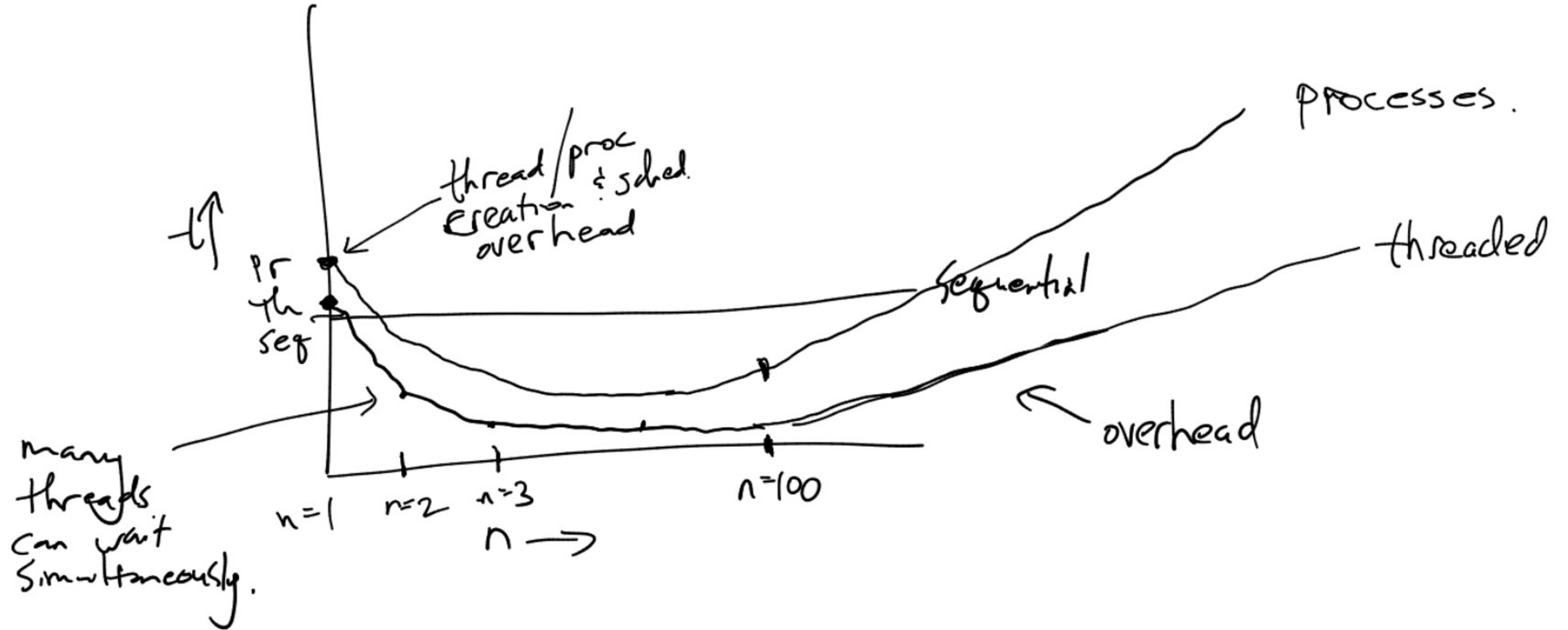
To wake up sleeping thread:

- get s from data structure
- V(s) ← increment
- delete s

To handle a syscall (monolithic kernel)

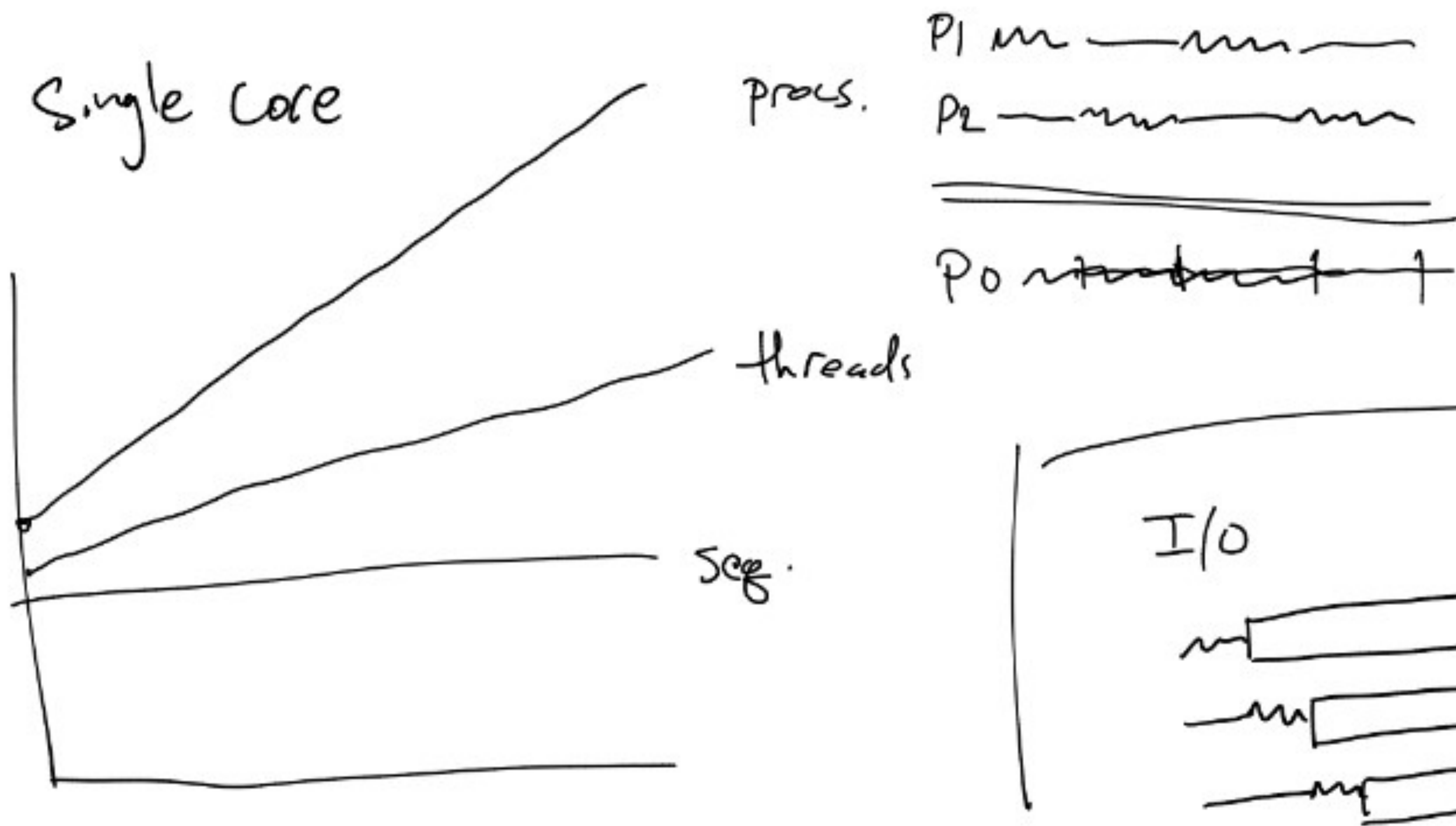
1. save state as if executing a normal function call
2. perform the operation (e.g. communicate w/ device)
   - branch (normal jump) to driver code to interact w/ device.
     (Note: if microkernel, schedule driver process)
3. return from syscall (restoring state)

# I/O bound



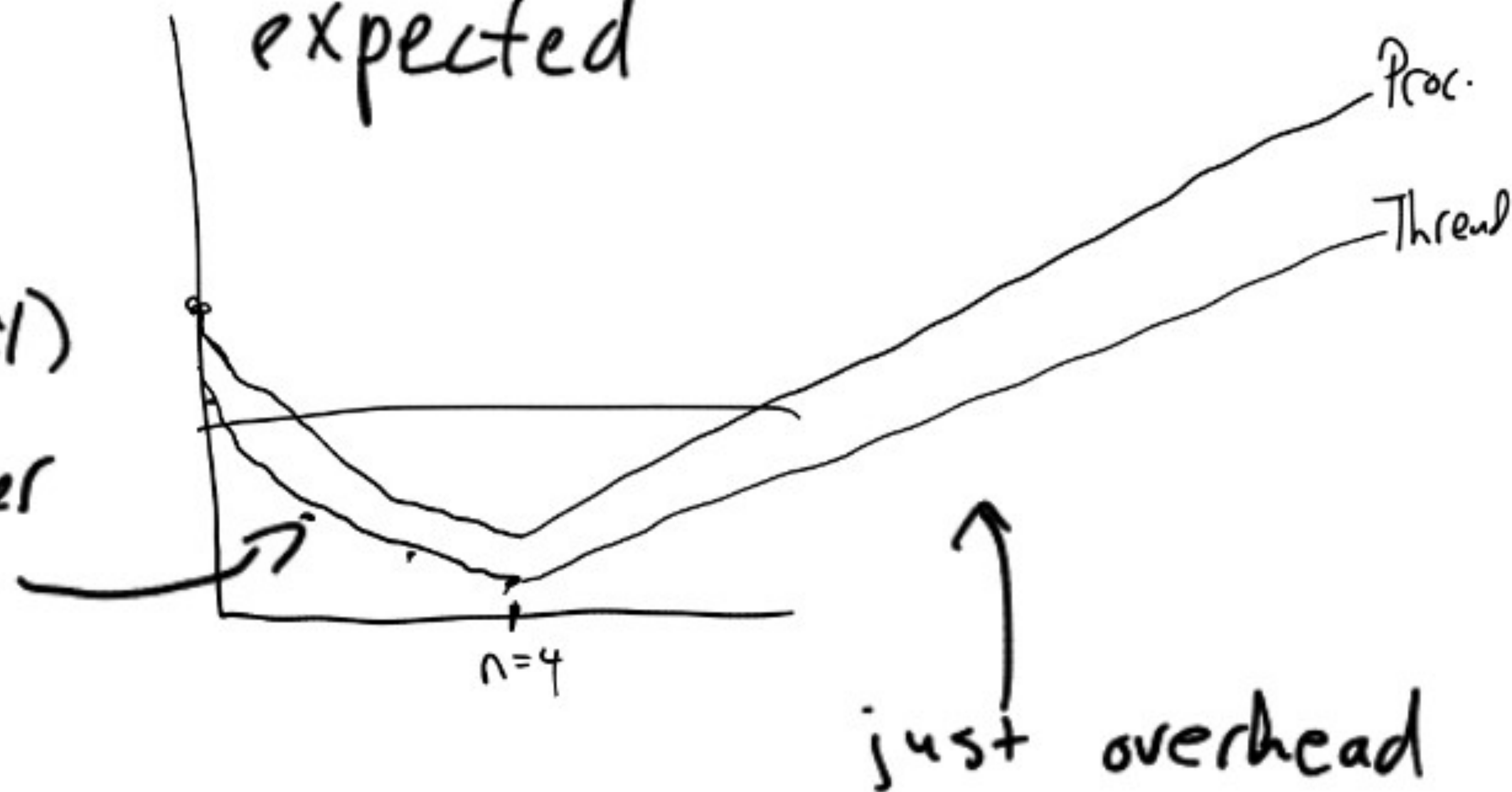
# CPU bound threads/procs

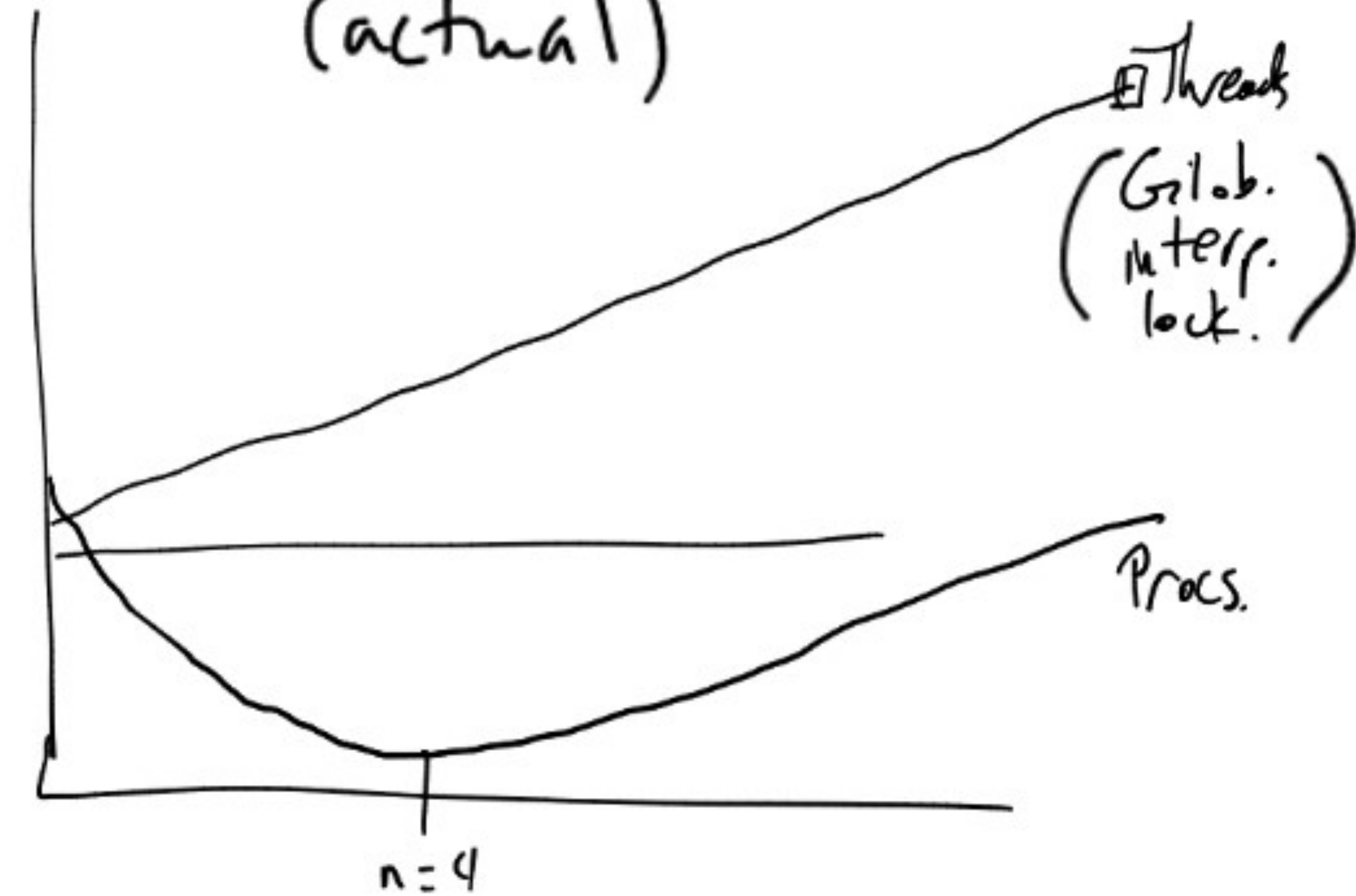
Only one CPU, so just overhead →




more threads ⇒ more CPUs (if avail) ⇒ faster

multicore (4)



```
class Semaphore (object):
    def __init__(self, val):
        self.val = val   # protected by self.lock
        Self.lock = 0  # TAS lock
        Self.queue = [] # protected by self.lock
          (waiting)
        # invariant: val >= 0, val = 0 if q is nonempty
    def P(self):
        while TAS(self.lock):
            yield()  # or pass
        if self.val > 0 :
            self.val --
            self.lock = 0
            return
        else:
            self.queue.enqueue(running_thread_TCB)
            release lock & deschedule ()
                current thread
            return
                            (scheduler)

    def V(self):
        while TAS(lock):
            yield() # or pass
        if queue.empty():
            self.val ++
            lock = 0
            return
        else:
            dequeue a thread t,
            mark t as ready
            lock = 0
            return.
```

## Monitor design pattern

- Invented by sir Tony Hoare.
  - closely related to Hoare logic
- Get support from PL.

(semaphores invented by Dijkstra)

Monitor object is like a normal OOP object,

- state (member vars)
- methods
- at any time, only one thread running inside a given monitor.
- able to wait for certain conditions to hold (e.x. wait for count > 0)

## Semaphore impl. as a Monitor (Hoare style)

```
class Semaphore(Monitor):
    def __init__(value):
        self.val = value   # invariant: value >= 0

    def P(self):
        wait until self.value > 0
        self.val --
    def V(self):
        self.val ++
```

Hoare-style.

In practice, need a little more effort from programmer.

# Bounded buffer monitor (Hoare style)

```
class Buffer(Monitor):
    def _init_(self):
        self.in = 0            # next empty slot
        self.out = 0           # next full slot (if ≠ in)
        self.buf = Object[N]
        # invariant: 0 ≤ out,in < N
        # in = out represents empty queue,
        # in+1 = out represents full queue

    def put(self, obj):
        wait until in+1 ≠ out (mod N)
        buf[in++ (mod N)] = obj

    def get(self, obj):
        wait until in ≠ out
        return buf[++out (mod N)]
```