synchronization
  why?
  In OS: physical resources are shared and need coordination, there are also abstract resources
  nowadays: synchronization is important for user-level multi-thread or multi-process programs too.

different forms of synchronization
  non-preemption based synchronization(yield)
    limitations: (single core, not working w/ i/o vm, make inter procedural programming difficult, …)
  preemption based synchronization:
  (mutual exclusion + ordering)
  public+private semaphore
  pthread lock+cond_variable
  monitor
  (java is like monitor)

Monitor:
synchronization+data+operation

```
single-resource
  bool busy;
//  cond available;
  acquire();
  release();

v1: if(!busy) busy=true;
     busy=false;
```

What is the problem?
    what if two threads try to acquire at the same time?
    what if an acquire and a release execute in parallel?

v2: correct version

1. mutual exclusion: two acquires cannot execute in parallel, acquire-release cannot execute in parallel because procedures of one monitor are mutually exclusive
2. is lock released at wait? yes
3. what if there are multiple waiters? one of the multiple will be woken up
4.

semantics:
  procedures of the same monitor object are mutually exclusive
  wait: what happened here?
      unlock, enqueue
  signal: what happened here (if no waiter, what happened? nothing, which is different from semaphore)
      dequeue, control immediately transfers to the dequeued/woken-up thread/process

proof rules (how to reason)
- invariants
- invariants established right before procedure exit; right before procedure entrance
- invariant established right at signal and right after wait’s wake-up (only applies to Hoare)

example: bounded buffer
- index, count, empty, full
- producer, consumer

other features:
- timed wait (alarm clock, disk head scheduler)
- reader writer …

=====

Mesa
- pilot, personal computer OS, mesa language, static checking for memory safety, procedure as process
- shared memory vs message passing

reality issues:
- wait in nested monitors
- deadlocks
  - M calls N; N calls M
  - M1-M2-wait; M1-signal
- condition variable
  - notify is just a hint!
  - (1) ease OS scheduling (2) ease notify logic
  - have to use “while”, instead of “if”
  - while(condition){wait;}
  - add new types of signal: + broadcast + timeout
- naked notify
- priority inversion

example:
- buffer allocation (malloc)
- mesa logic is more suitable here than Hoare logic