Topic 6: Synchronization via Semaphores

Reading: 5.1-5.5
Next reading: 5.8

- The too-much-milk solution is much too complicated and too limiting. Turning off interrupts isn't possible at user-level. The mutual exclusion mechanism is too simple-minded: it uses only atomic reads and writes. This is sufficient, but unpleasant. It would be unbearable to extend that mechanism to many processes. Let's look at more powerful, higher-level mechanisms.

- *Required properties* of a mutual exclusion mechanism:

    - Must allow only one process into a critical section at a time.

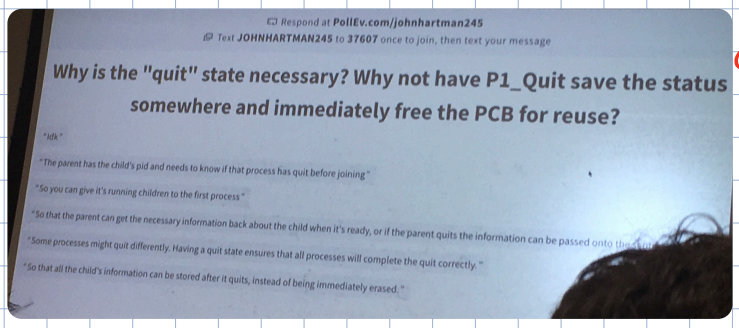
    - Must handle an arbitrary number of processes.

    - If several processes arrive at once, must allow one and only one process to proceed.

    - If one or more processes are waiting to enter the critical section one must do so when the process leaves the critical section (liveness).

- *Desirable properties* of a mutual exclusion mechanism:

    - **Fair**: if several processes waiting, each should get in eventually.



① freeing PCB (status) still in use

② PCB I → Quit → reuse

PCB II can be reused before completely cleared

PCB II function might be confused / unexpected result.

    - **Efficient**: should not use up substantial amounts of resources when waiting. E.g. no busy waiting.
    - **Simple**: should be easy to use (e.g. just bracket the critical sections, lock abstraction).

- Requirements of processes using the mechanism:
    - Always lock before manipulating shared data.
    - Always unlock after manipulating shared data.
- Desirable properties of processes using the mechanism:
    - Do not lock again if already locked.
        - Some mechanisms will cause the process to *deadlock* if this happens, others allow it.
    - Do not unlock if not locked by you
        - There are a few exceptions to this.
    - Do not spend lots of time in critical section.
- *Semaphore*: A synchronization variable that takes on positive integer values. Invented by E. Dijkstra in the mid 60's.
    - **P**(semaphore): an atomic operation that **waits** for semaphore to become positive, then decrements it by 1 ('Proberen' in Dutch).
    - **V**(semaphore): an atomic operation that increments semaphore by 1. It does **not** wait. ('Verhoog' in Dutch)

- Semaphores are simple and elegant and allow the solution of many interesting problems. They do a lot more than just mutual exclusion.
- Book doesn't like them, prefers higher-level constructs based on locks and condition variables.

- Too much milk problem with semaphores:

Process A & B

wait until positive, -1 then

```
P(OKToBuyMilk);
if (Milk == 0) {
    Milk++;
}
V(OKToBuyMilk);
```

```
P(OKToBuyMilk);
if (Milk == 0) {
    Milk++;
}
V(OKToBuyMilk);
```

  - Note: `OKToBuyMilk` must initially be set to 1. What happens if it isn't?
  - Why can there can never be more than one process buying milk at once?

- Semaphores aren't provided by hardware, (I'll describe implementation later) but they have several attractive properties:
  - Machine independent.
  - Simple.
  - Work with many processes.