CS 537 Notes, Section #6: Semaphores and Producer/Consumer Problem

---

The refrigerator example solution is much too complicated. The problem is that the mutual exclusion mechanism was too simple-minded: it used only atomic reads and writes. This is sufficient, but unpleasant. It would be unbearable to extend the fridge mechanism to many processes. Let's look for a more powerful, higher-level mechanism.

Requirements for a mutual exclusion mechanism:

- Must allow only one process into a critical section at a time.
- If several requests at once, must allow one process to proceed.
- Processes must be able to go on vacation outside critical section.

Desirable properties for a mutual exclusion mechanism:

- Fair: if several processes waiting, let each in eventually.
- Efficient: do 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).

Desirable properties of processes using the mechanism:

- Always lock before manipulating shared data.
- Always unlock after manipulating shared data.
- Do not lock again if already locked.
- Do not unlock if not locked by you.
- Do not spend large amounts of time in critical section. E.g. do not go on vacation.

Semaphore: A synchronization variable that takes on positive integer values. Invented by Dijkstra.

- P(semaphore): an atomic operation that waits for semaphore to become positive, then decrements it by 1.
- V(semaphore): an atomic operation that increments semaphore by 1.

  The names come from the Dutch, *proberen* (test) and *verhogen* (increment). Semaphores are simple and elegant and allow the solution of many interesting problems. They do a lot more than just mutual exclusion.

Too much milk problem with semaphores:

Processes A & B

```
OKToBuyMilk.P();
if (NoMilk) BuyMilk();
OKToBuyMilk.V();
```

Note: OKToBuyMilk must initially be set to 1. What happens if it is not?

Show why there can never be more than one process buying milk at once.

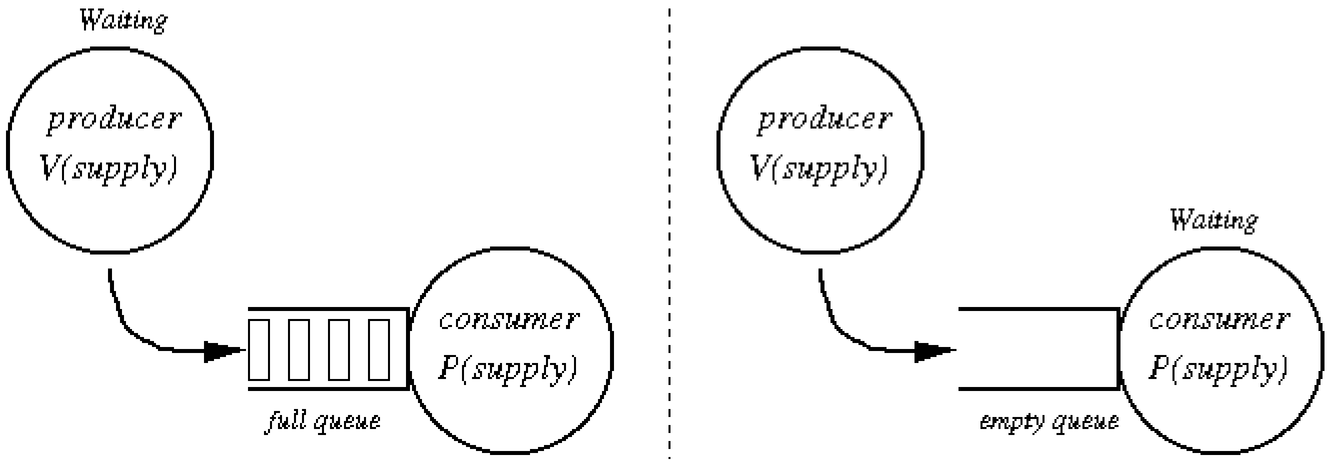
Binary semaphores are those that have only two values, 0 and 1. They are implemented in the same way as regular semaphores except multiple V's will not increase the semaphore to anything greater than one.

Semaphores are not provided by hardware (we will discuss implementation later). But they have several attractive properties:

- Machine independent.
- Simple.
- Powerful. Embody both exclusion and waiting.
- Correctness is easy to determine.
- Work with many processes.
- Can have many different critical sections with different semaphores.
- Can acquire many resources simultaneously (multiple P's).
- Can permit multiple processes into the critical section at once, if that is desirable.

Semaphores can be used for things other than simple mutual exclusion. For example, resource allocation: P = allocate resource, V = return resource. More on this later.

Semaphore Example: Producer & Consumer: Consider the operation of an assembly line or pipeline. Processes do not have to operate in perfect lock-step, but a certain order must be maintained. For example, must put wheels on before the hub caps. It is OK for wheel mounter to get ahead, but hub-capper must wait if it gets ahead. Another example: compiler & disk reader.



Producer and Consumer

- Producer: creates copies of a resource.
- Consumer: uses up (destroys) copies of a resource.
- Synchronization: keeping producer and consumer in step.
- Define constraints (definition of what is "correct").
    - B must wait for A to fill buffers. (resource management)
    - A must wait for B to empty buffers. (resource management)
    - Only one process must fiddle with buffer list at once. (mutual exclusion)
- A separate semaphore is used for each constraint.

Declare and initialize variables:

```
        semaphore empty(N),
                  full(0),
                  list(1);

        List empties (N),
             fulls (0);
```

Producer Process:	**Consumer Process:**
`P01:  empty.P();`	`C01:  full.P();`
`P02:  list.P();`	`C02:  list.P();`
`P03:  b = empties.remove();`	`C03:  b = fulls.remove();`
`P04:  list.V();`	`C04:  list.V();`
`P05:  `*`Put data into B`*	`C05:  `*`Remove data from B`*
`P06:  list.P();`	`C06:  list.P();`
`P07:  fulls.add(b);`	`C07:  empties.add(b);`
`P08:  list.V();`	`C08:  list.V();`
`P09:  full.V();`	`C09:  empty.V();`

- Important questions:
    - Why does A do an `empty.P()` but `full.V()`?
    - Why is order of P's important?
    - Is order of V's important?

    - Could we have separate semaphores for each list?
    - How would this be extended to have 2 consumers?

Producers and consumers are much like Unix pipes.

THIS IS A VERY IMPORTANT EXAMPLE!
