6 Synchronization with Semaphores

The too-much-milk 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 that mechanism to many processes. Let's look at more powerful, higher-level mechanisms.

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: don't 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 (usually: there are a few exceptions to this).
- Do not spend large amounts of time in critical section.

Semaphore: A synchronization variable that takes on positive integer values. Invented by Edsger Dijkstra in the mid 60's.

- P(semaphore): an atomic operation that waits for semaphore to become greater than zero, then decrements it by 1 ("proberen" in Dutch).
- V(semaphore): an atomic operation that increments semaphore by 1 ("verhogen" in Dutch).

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
P(OKToBuyMilk);
if (NoMilk) {
      Buy Milk;
}
V(OKToBuyMilk);
```

Note: OKToBuyMilk must initially be set to 1. What happens if it isn't?

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

Binary semaphores are those that can only take on two values, 0 and 1.

Semaphores aren't provided by hardware. (I'll describe implementation later.) But they have several attractive properties:

- Machine independent.
- Simple.
- 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.

Desirability of layering: picking powerful and flexible intermediate solutions to problems. A synchronization kernel is appropriate for one layer.

Semaphores are used in two different ways:

- Mutual exclusion : to ensure that only one process is accessing shared information at a time. If there are separate groups of data that can be accessed independently, there may be separate semaphores, one for each group of data. These semaphores are always binary semaphores.
- Condition synchronization : to permit processes to wait for certain things to happen. If there are different groups of processes waiting for different things to happen, there will usually be a different semaphore for each group of processes. These semaphores aren't necessarily binary semaphores.

Semaphore Example: Producer & Consumer. Suppose one process is creating information that is going to be used by another process, e.g. suppose one process is reading information from the disk, and another process will compile that information from soure code to binary. Processes shouldn't have to operate in perfect lock-step: producer should be able to get ahead of consumer.

- Producer: creates copies of a resource.
- Consumer: uses up (destroys) copies of a resource. (may produce something else)
- Buffers: used to hold information after producer has created it but before consumer has used it.
- Synchronization: keeping producer and consumer in step.
- Define constraints (definition of what is "correct"). Note importance of doing this before coding.
  - Consumer must wait for producer to fill buffers. (condition synchronization)

  - Producer must wait for consumer to empty buffers, if all buffer space is in use. (condition synchronization)
  - Only one process must manipulate buffer pool at once. (mutual exclusion)

- A separate semaphore is used for each constraint. Explain the three semaphores, what they mean, who P's and who V's.

- Initialization:
  - Put all buffers in pool of empties.
  - Initialize semaphores: empties = numBuffers, fulls= 0, mutex = 1;

- Producer process:

```
P(empties);
P(mutex);
get empty buffer from pool of empties;
V(mutex);
produce data in buffer;
P(mutex);
add full buffer to pool of fulls;
V(mutex);
V(fulls);
```

- Consumer process:

```
P(fulls);
P(mutex);
get full buffer from pool of fulls;
V(mutex);
consume data in buffer;
P(mutex);
add empty buffer to pool of empties;
V(mutex);
V(empties);
```

- Important questions:
  - Why does producer P(empties) but V(fulls)? Explain in terms of creating and destroying resources.
  - Why is order of P's important? Deadlock (deadly embrace).
  - Is order of V's important?
  - Could we have separate mutex semaphores for each pool?
  - How would this be extended to have 2 consumers?

Producers and consumers produces something much like Unix pipes.

THIS IS AN IMPORTANT EXAMPLE! Go over the two classes of semaphore usage again: mutual exclusion and scheduling.

Another example of semaphore usage: a shared database with readers and writers. It is safe for any number of readers to access the database simultaneously, but each writer must have exclusive access. Must use semaphores to enforce these policies. Example: checking account (statement-generators are readers, tellers are writers).

- Writers are actually readers too.
- Constraints:
  - Writers can only proceed if there are no active readers or writers (use semaphore OKToWrite).
  - Readers can only proceed if there are no active or waiting writers (use semaphore OKToRead).
  - To keep track of who's reading and writing, need some shared variables. These are called *state variables*. However, must make sure that only one process manipulates state variables at once (use semaphore Mutex).
- State variables:
  - AR = number of active readers.
  - WR = number of waiting readers.

- AW = number of active writers.
- WW = number of waiting writers.

AW is always 0 or 1.

AR and AW may not both be non-zero.

- Initialization:
  - OKToRead = 0; OKToWrite = 0; Mutex = 1;
  - AR = WR = AW = WW = 0;
- Scheduling: writers get preference.

Reader Process:

```
P(Mutex);
if ((AW+WW) == 0) {
    V(OKToRead);
    AR = AR+1;
} else {
    WR = WR+1;
}
V(Mutex);
P(OKToRead);
– read the necessary data;
P(Mutex);
AR = AR-1;
if (AR==0 && WW>0) {
    V(OKToWrite);
    AW = AW+1;
    WW = WW-1;
}
V(Mutex);
```

Writer Process:

```
P(Mutex);
if ((AW+AR+WW) == 0) {
    V(OKToWrite);
    AW = AW+1;
} else {
    WW = WW+1;
}
V(Mutex);
P(OKToWrite);
– write the necessary data;
P(Mutex);
AW = AW-1;
if (WW>0) {
    V(OKToWrite);
    AW = AW+1;
    WW = WW-1;
} else while (WR>0) {
    V(OKToRead);
    AR = AR+1;
    WR = WR-1;
}
V(Mutex);
```

Go through several examples: (tell what happens)

- Reader enters and leaves system.
- Writer enters and leaves system.
- Two readers enter system.
- Writer enters system and waits.
- Reader enters system and waits.
- Readers leave system, writer continues.

- Writer leaves system, last reader continues and leaves.

Questions:

- In case of conflict between readers and writers, who gets priority? Readers can get locked out.
- Is the WW necessary in the writer's first if? No: if there is a waiting writer, then there must be an active reader or an active writer.
- Can OKToRead ever get greater than 1? What about OKToWrite?
- Is the first writer to execute P(Mutex) guaranteed to be the first writer to access the data?