

CS 414 : Operating Systems

UNIVERSITY OF VIRGINIA
Department of Computer Science

Spring 2008

Topic 4: Synchronization with Semaphores

- Readings for this topic: Sections 6.1-6.6
- The too-much-milk solution is much too complicated. Why?
- Requirements for a mutual exclusion mechanism:
- Desirable properties for a mutual exclusion mechanism:
 - Fair:
 - Efficient:
 - Simple:
- 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 (there are a few exceptions to this).
 - Do not spend large amounts of time in critical section.
- *Semaphore*: An integer variable that can be accessed only through two atomic operations. In-

vented by Edsger Dijkstra in 1965.

- P(semaphore): an atomic operation that waits for semaphore to become positive, then decrements it by 1 (also called *wait*)
- V(semaphore): an atomic operation that increments semaphore by 1 (also called *signal*)

Do names mean anything? Proberen and Verhogen

Semaphores are simple and elegant and allow the solution of many interesting problems.

- Can semaphores be accessed by reads and writes?

- Too much milk problem with semaphores:

Processes A & B

```
1  P(OKToBuyMilk);
2  if (NoMilk) {
3      BuyMilk;
4  }
5  V(OKToBuyMilk);
```

- What must be the initial value of OKToBuyMilk?
- Show why there can never be more than 1 process buying milk at once.
- *Binary semaphores*
- Are semaphores provided by hardware?
- Attractive properties of semaphores:
- 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. Are these semaphores always binary semaphores?

- Scheduling: 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. Do these semaphores necessarily be 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 source code to binary. Processes shouldn't have to operate in perfect lock-step: producer should be able to get ahead of consumer.
 - Buffers: hold item after producer created it but before consumer use it.
 - Synchronization: keeping producer and consumer in step.
 - Define constraints (definition of what is "correct").
 - can't underflow: consumer must wait for producer to fill buffers. (scheduling)
 - can't overflow: producer must wait if all buffer space is in use. (scheduling)
 - only one process must manipulate buffer pool at once. (mutual exclusion)
 - A separate semaphore is used for each constraint.

- Initialization:
- Producer process:

```
repeat
    produce an item;
    P(empty);
    P(mutex);
    add the item to buffer;
    V(mutex);
    V(full);
until false;
```

- Consumer process:
repeat
 P(full);
 P(mutex);
 remove item from buffer;
 P(mutex);
 V(empty);
 consume the item removed;
until false;

- Important questions:

- Why does producer P(empty) but V(full)?
- Is order of P's important?
- Is order of V's important?
- How would this be extended to have 2 consumers?

- Readers and writers problem. Example: airline reservations, bank accounts

- Constraints:
 - Readers can proceed only if there are no active or waiting writers (use semaphore OKToRead).
 - Writers can proceed only if there are no active readers or writers (use semaphore OKToWrite).
 - To keep track of the number of readers and writers, 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.

What can be the value of AW? What about AR and WW?

- Initialization:
 - OKToRead = 0; OKToWrite = 0; Mutex = 1; AR = WR = AW = WW = 0;

- 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:

- 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?
 - Is the WW necessary in the writer's first if?
 - 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?
- Dining Philosophers Problem
 - What do the philosophers do?
 - What is their favorite food?
 - N: number of philosophers; fork[N]: semaphores
 - Philosopher (i):


```

while (true)
{
    P(fork[i]);
    P(fork[(i+1)%N]);
    eat;
    V(fork[(i+1)%N]);
    V(fork[i]);
    think;
}
      
```
 - Does it work?

- A solution using state variables
 - State[N]: keep everyone's state: Thinking, Hungry, Eating
 - semaphore mutex, sem[N]
 - Left=(i-1)%N; Right=(i+1)%N
- Philosopher (i):


```

while (true)
{
    think;
    take_fork(i);
    eat;
    put_fork(i);
}
      
```
- take_fork(i)


```

P(mutex);
State[i]=Hungry;
test(i);
V(mutex);
P(sem[i]);
      
```
- test(i)


```

if (State[i]==Hungry && State[Left]!=Eating && State[Right]!=Eating)
    {State[i]=Eating;
     V(sem[i]);}
      
```
- put_fork(i)


```

P(mutex);
State[i]=Thinking;
test(Left);
test(Right);
V(mutex);
      
```