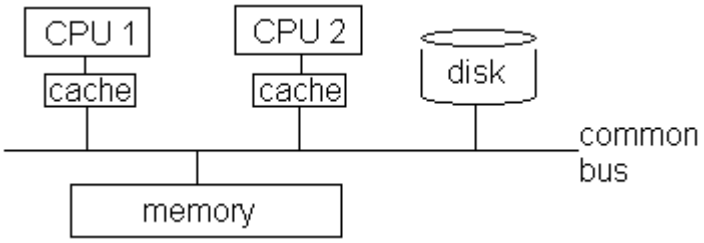
COMP 3000
Notes for November 2, 2004 taken by Melissa Lombard

Concurrency # 1 – Principles, Mutexes & Semaphores

Chapter 5, Sections 1 – 3, pp. 200 – 227

Consider the following arrangment:



Most of the time you want each CPU to access it's own cache memory (faster).
However, what if both CPUs cache the same area of memory and then try to write it out?
There has to be some kind of mechanism for ensuring that:

- one CPU's cache is not out of date
- they do not overwrite each other's changes

Test & Set Instruction

```
1    int testSet (int i) {
2        if( i == 0 )
3            i = 1;
4            return true;
5        else
6            return false;
7    }
```

Before entering a **critical section** of code, you must get a **true** return value from the testSet function. If it fails, you **busy wait** – run the test over and over again until you succeed (not good use of resources, might deadlock).

What if you get interrupted between lines 3 and 4? -> You have reached a true condition, but the integer was not set – another process may also get a true return value from testSet – Now two processes are in their critical sections which may interfere with each other!

Solution - This code is implemented as a single, atomic special instruction so that it cannot be interrupted.

Semaphores

2 Types:

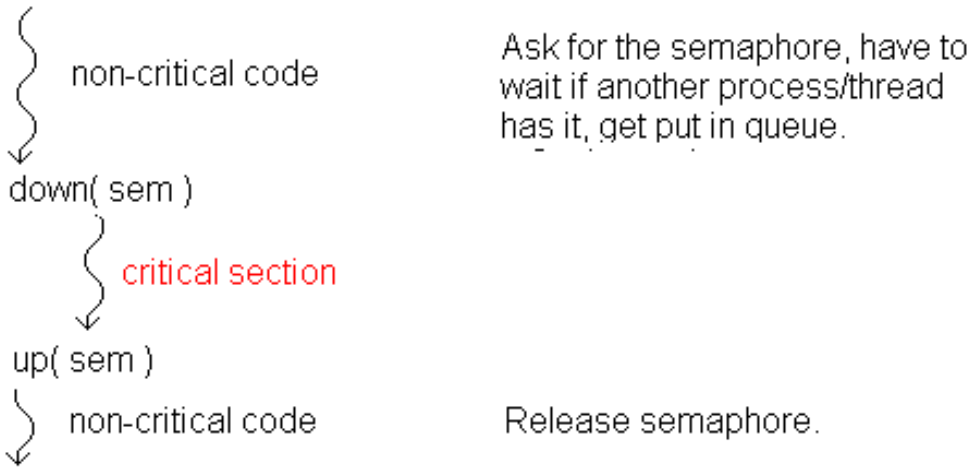
- Counting / general semaphores
- Binary semaphore / Mutex

Semaphore parts:

- State variable (binary int)
- Queue of waiting threads or processes

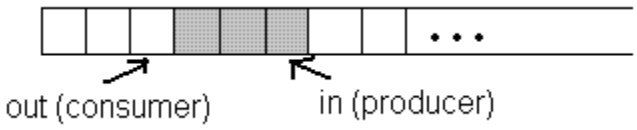
Operations:

- Wait (or “down” in linux) - check if it’s safe to enter
- Signal (or “up” in linux) - say “ok I’m done!”



Producer / Consumer Problem

Buffer



To enforce integrity of the data structure, only 1 of the producer or consumer should access it at a time.

```
int n = 0
semaphore s = 1, delay = 0
```

producer

```
1    while(true) {
2         produce()
3         down(s)
4         append()
5         n++
6         if (n == 1)
7              up(delay)
8         up(s)
9    }
```

consumer

```
1    down(delay)
2    while(true) {
3         down(s)
4         take()
5         n--
6         up(s)
7         consume()
8         if (n == 0)
9              down(delay)
10   }
```

Consumer notes:

Line 1 – Wait until there is something produced.

Lines 4, 5 – Should actually make a copy of the semaphor **`s`** …

Line 8 - … and check if $\mathbf{n_{copy} == 0}$ instead, because the original **`s`** could have been changed after **`up(s)`** in Line 7

Downside – Semaphore code has to appear everywhere that you access the data structure

Monitors

Monitors are objects that only allow one thread to access them at a time. They are a wrapper for the critical resource. They have:

- an initialization sequence
- state variables (local)
- procedures for accessing the critical resource

```
monitor boundedBuffer              // monitor has cwait() and csignal() methods
buffer[N]
nextin, nextout                    // pointers
count
notfull, not empty

append(X) {
     if (count == N)
          cwait(notfull)           // if the buffer is full, wait
     buffer[nextin] = X;           // put something in the buffer
     nextin = (nextin + 1) % N     // increment pointer (wrap around if at end)
     count++                       // increment counter
     csignal(notempty)                   // signal completion
}

take(X) {
     if (count == 0)
          cwait(notempty)          // if the buffer is empty, then wait
     X = buffer[nextout]           // get next item
     nextout = (nextout + 1) % N   // increment pointer
     count--                       // increment counter
     csignal(notfull)              // signal completion
}
```

now, producer and consumer code is much cleaner:

```
producer {
     while(true)
          produce()
          append()       // access critical resource using monitor function
}
```

** See also the monitor implementation using **notify**() and **broadcast**() instead of cwait() and csignal(), on page 232.