**Algorithm 3**

- Think of this algorithm as using a referee who keeps track of whose “turn” it is
  - Anytime the two disagree about whose turn it is, they ask the referee, who keeps track of whose turn it is to have priority
  - This is called Peterson’s algorithm (1981)
    - The original (but more complicated) solution to this problem is Dekker’s algorithm (1965)

- For n processes, we can use Lamport’s Bakery algorithm (1974)
  - When a thread tries to enter the critical section, it get assigned a number higher than anyone else’s number
  - Thread with lowest number gets in
  - If two threads get the same number, the one with the lowest process id gets in

```java
Algorithm 3 (cont.)

Code:
```
t1 ( ) {
    while (true) {
        t1_in_crit = true;
        turn = 2;
        while (t2_in_crit == true && turn != 1) {
            /* do nothing */
            … critical section of code …
        }
        t1_in_crit = false;
        … other non-critical code …
    }
}
```
t2 ( ) {
    while (true) {
        similar…
    }
}
```

---

**Semaphores — OS Support for Mutual Exclusion**

- Semaphores were invented by Dijkstra in 1965, and can be thought of as a generalized locking mechanism
  - A semaphore supports two *atomic* operations, `P / wait` and `V / signal`
    - The semaphore initialized to 1
    - Before entering the critical section, a thread calls “P(semaphore)”, or sometimes “wait(semaphore)”
    - After leaving the critical section, a thread calls “V(semaphore)”, or sometimes “signal(semaphore)”

- Too much milk:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>milk.P();</td>
<td>milk.P();</td>
</tr>
<tr>
<td>if (noMilk) buy milk;</td>
<td>if (noMilk) buy milk;</td>
</tr>
<tr>
<td>milk.V();</td>
<td>milk.V();</td>
</tr>
</tbody>
</table>

**What Does a Semaphore Do?**

- Semaphore “s” is initially 1

- Before entering the critical section, a thread calls “P(s)” or “wait(s)”
  - wait (s):
    - s = s – 1
    - if (s < 0)
      - block the thread that called wait(s) on a queue associated with semaphore s
    - otherwise
      - let the thread that called wait(s) continue into the critical section

- After leaving the critical section, a thread calls “V(s)” or “signal(s)”
  - signal (s):
    - s = s + 1
    - if (s ≤ 0), then
      - wake up one of the threads that called wait(s), and run it so that it can continue into the critical section
Semaphores for Mutual Exclusion

- **Too much milk:**

  Thread A
  - `milk.P();`
  - `if (!haveMilk) { buy milk; haveMilk=true; milk.V(); }`

  Thread B
  - `milk.P();`
  - `if (!haveMilk) { buy milk; haveMilk=true; milk.V(); }`

  - “haveMilk” is a Boolean variable
  - “milk” is a semaphore initialized to 1

- **Execution:**

<table>
<thead>
<tr>
<th>After:</th>
<th>milk</th>
<th>queue</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A: milk.P();</td>
<td>0</td>
<td>in CS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B: milk.P();</td>
<td>-1</td>
<td>B</td>
<td>waiting</td>
<td></td>
</tr>
<tr>
<td>A: milk.V();</td>
<td>0</td>
<td>finish</td>
<td>ready, in CS</td>
<td></td>
</tr>
<tr>
<td>B: milk.V();</td>
<td>1</td>
<td>finish</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Semaphore Operation

- **Informal description:**
  - Single igloo, containing a blackboard and a very large freezer
  - Wait — thread enters the igloo, checks the blackboard, and decrements the value shown there
    - If new value is 0, thread goes on to the critical section
    - If new value is negative, thread crawls in the freezer and hibernates (making room for others to enter the igloo)
  - Signal — thread enters igloo, checks blackboard, and increments the value there
    - If new value is 0 or negative, there’s a thread waiting in the freezer, so it thaws out a frozen thread, which then goes on to the critical section

Using Semaphores

- **Code using semaphores:**

  ```java
t1 () {
    while (true) {
      wait (s);
      … critical section of code …
      signal (s);
      … other non-critical code …
    }
  }

t2 () {
    while (true) {
      wait (s);
      … critical section of code …
      signal (s);
      … other non-critical code …
    }
  }
```

Semaphore Operation & Values

- **Semaphores (simplified slightly):**

  ```java
  wait (s): signal (s):
  s = s - 1          s = s + 1
  if (s < 0)          if (s <= 0)
                     block the thread
                     wake up & run one of
                     that called wait(s)
                     the waiting threads
  otherwise
  continue into CS
  ```

- **Semaphore values:**

  - *Binary semaphore* has an initial value of 1 and is used for mutual exclusion
    - Positive semaphore = number of (additional) threads that can be allowed into the critical section (usually max of 1)
    - Negative semaphore = number of threads blocked (note — there’s also one in CS)
  - *Counting semaphore* has an initial value greater than 1, and is used for synchronization between threads
The Coke Machine
(Bounded-Buffer Producer-Consumer)

/* number of full slots (Cokes) in machine */
semaphore fullSlot = 0;
/* number of empty slots in machine */
semaphore emptySlot = 100;
/* only one person accesses machine at a time */
semaphore mutex = 1;

DeliveryPerson()
{
    emptySlot.P( ); /* empty slot avail? */
    mutex.P( ); /* exclusive access */
    put 1 Coke in machine
    mutex.V( );
    fullSlot.V( ); /* another full slot! */
}

ThirstyPerson()
{
    fullSlot.P( ); /* full slot (Coke)? */
    mutex.P( ); /* exclusive access */
    get 1 Coke from machine
    mutex.V( );
    emptySlot.V( ); /* another empty slot! */
}