

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

Algorithm 3 (cont.)

- Code:


```

t1 () {
    while (true) {
        t1_in_crit = true;
        turn = 2;
        while (t2_in_crit == true && tum != 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:

Thread A

```

milk.P( );
if (noMilk)
    buy milk;
milk.V( );

```

Thread B

```

milk.P( );
if (noMilk)
    buy milk;
milk.V( );

```

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

Using Semaphores for Mutual Exclusion

■ Too much milk:

<u>Thread A</u>	<u>Thread B</u>
milk.P();	milk.P();
if (!haveMilk)	if (!haveMilk)
buy milk;	buy milk;
haveMilk=true;	haveMilk=true;
milk.V();	milk.V();

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

■ Execution:

<u>After:</u>	<u>milk</u>	<u>queue</u>	<u>A</u>	<u>B</u>
	1			
A: milk.P();	0		in CS	
B: milk.P();	-1	B	in CS	waiting
A: milk.V();	0		finish	ready, in CS
B: milk.V();	1			finish

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:

```

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):

<u>wait (s):</u>	<u>signal (s):</u>
s = s - 1	s = s + 1
if (s < 0)	if (s ≤ 0)
block the thread that called wait(s)	wake up & run one of 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! */  
}
```