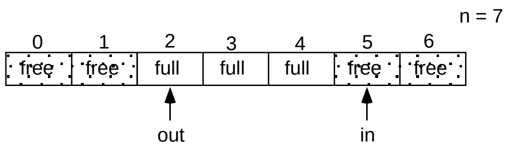
The Producer-Consumer Problem (Review from Lecture 07)

- One thread is a producer of information; another is a consumer of that information
  - They share a bounded circular buffer
  - Processes — OS must support shared memory between processes
  - Threads — all memory is shared

```
var buffer:  array[0..n-1] of items;     /* circular array */
in = 0
out = 0
```




```
/* producer */
repeat forever
      …
      produce item nextp
      …
      while (in+1 mod n == out)
            do nothing
      buffer[in] = nextp
      in = in+1 mod n
end repeat
```

```
/* consumer */
repeat forever
      while (in == out)
            do nothing
      nextc = buffer[out]
      out = out+1 mod n
      …
      consume item nextc
      …
end repeat
```


Too Much Milk!

Time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge, no milk
3:25	Buy milk, leave	Leave for grocery
3:30		
3:35	Arrive home	Arrive at grocery
3:36	Put milk in fridge	
3:40		Buy milk, leave
3:45		
3:50		Arrive home
3:51		Put milk in fridge

3:51 Oh, no! ***Too much milk!!***

- The problem here is that the lines:

  "Look in fridge, no milk"

  through

  "Put milk in fridge"

  are not an **atomic** operation

Another Example

```
Thread A                  Thread B

i = 0                     i = 0
while (i < 10)            while (i > –10)
   i = i + 1                  i = i – 1
print "A wins"            print "B wins"
```

- Assumptions:
  - Memory load and store are atomic
  - Increment and decrement are not atomic
- Questions:
  - Who wins?
  - Is it guaranteed that someone wins?
  - What if both threads have their own CPU, running concurrently at exactly the same speed? Is it guaranteed that it goes on forever?
  - What if they are sharing a CPU?

Synchronization Terminology

- *Synchronization* — using *atomic* (indivisible) operations to ensure cooperation between threads
- *Mutual exclusion* — ensures that only one thread does a particular activity at a time — all other threads are *excluded* from doing that activity
- *Critical section* (*region*) — code that only one thread can execute at a time (e.g., code that modifies shared data)
- *Lock* — mechanism that prevents another thread from doing something:
  - *Lock* before entering a critical section
  - *Unlock* when leaving a critical section
  - Thread wanting to enter a locked critical section must **wait** until it's unlocked

Enforcing Mutual Exclusion

- Methods to enforce mutual exclusion
  - Up to user — threads have to explicitly coordinate with each other
  - Up to OS — OS provides support for mutual exclusion
  - Up to hardware — hardware provides architectural support for mutual exclusion
- Solution must:
  - Avoid *starvation* — if a thread starts trying to gain access to the critical section, then it should eventually succeed
  - Avoid *deadlock* — if **some** threads are trying to enter their critical sections, then **one** of them must eventually succeed
- We will assume that a thread may halt in its non-critical-section, but not in its critical section

Algorithm 1

- Informal description:
  - Igloo with blackboard inside
    - Only one person (thread) can fit in the igloo at a time
    - In the igloo is a blackboard, which is large enough to hold only one value
  - A thread that wants to execute the critical section enters the igloo, and examines the blackboard
    - If its number is not on the blackboard, it leaves the igloo, goes outside, and runs laps around the igloo
      - After a while, it goes back inside, and checks the blackboard again
      - This "busy waiting" continues until eventually its number is on the blackboard
    - If its number is on the blackboard, it leaves the igloo and goes on to the critical section
    - When it returns from the critical section, it enters the igloo, and writes the other thread's number on the blackboard

Algorithm 1 (cont.)

- Code:

```
t1 ( )   {
    while (true)   {
        while (turn != 1)
            ;     /* do nothing */
        … critical section of code …
        turn = 2;
        … other non-critical code …
    }
}

t2 ( )   {
    while (true)   {
        while (turn != 2)
            ;     /* do nothing */
        … critical section of code …
        turn = 1;
        … other non-critical code …
    }
}
```


Algorithm 2a

- Informal description:
  - Each thread has its own igloo
    - A thread can examine and alter its own blackboard
    - A thread can examine, but not alter, the other thread's blackboard
    - "true" on blackboard = that thread is in the critical section
  - A thread that wants to execute the critical section enters the other thread's igloo, and examines the blackboard
    - It looks for "false" on that blackboard, indicating that the other thread is not in the critical section
      - When that happens, it goes back to its own igloo, and writes "true" on its own blackboard, and then goes on to the critical section
    - When it returns from the critical section, it enters the igloo, and writes "false" on the blackboard

Algorithm 2a (cont.)

- Code:

```
t1 ( )   {
    while (true)   {
        while (t2_in_crit == true)
            ;     /* do nothing */
        t1_in_crit = true;
        … critical section of code …
        t1_in_crit = false;
        … other non-critical code …
    }
}

t2 ( )   {
    while (true)   {
        while (t1_in_crit == true)
            ;     /* do nothing */
        t2_in_crit = true;
        … critical section of code …
        t2_in_crit = false;
        … other non-critical code …
    }
}
```


Algorithm 2b

- Code:

```
t1 ( )   {
    while (true)   {
        t1_in_crit = true;
        while (t2_in_crit == true)
            ;     /* do nothing */
        … critical section of code …
        t1_in_crit = false;
        … other non-critical code …
    }
}

t2 ( )   {
    while (true)   {
        t2_in_crit = true;
        while (t1_in_crit == true)
            ;     /* do nothing */
        … critical section of code …
        t2_in_crit = false;
        … other non-critical code …
    }
}
```