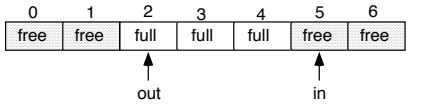
The Producer-Consumer Problem (Review from Chapter 03)

- 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                                                  n = 7
```




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

Critical Section & Mutual Exclusion

- *Critical section* (*region*) — code that only one thread can execute at a time (e.g., code that modifies shared data)
- *Mutual exclusion* — ensures that only one thread does a particular activity at a time — all other threads are *excluded* from doing that activity
  - More formally, if process P_i is executing in its critical section, then no other processes can be executing in their critical sections
- *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 — support for mutual exclusion
  - Up to hardware —architectural support
- Solution must make *progress* — if no process is executing in its critical section, and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
  - 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

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 …
    }
}
```


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 && 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:

Thread A	Thread B

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

Thread A	Thread B

milk.P( );
if (!haveMilk)
  buy milk;
  haveMilk=true;
milk.V( );
	milk.P( );
	if (!haveMilk)
	  buy milk;
	  haveMilk=true;
	milk.V( );

  - "haveMilk" is a Boolean variable
  - "milk" is a semaphore initialized to 1
- Execution:

After:	milk	queue	A	B
	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):

wait (s):	signal (s):
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! */
}
```


Two Versions of Semaphores

- Semaphores from last time (simplified):

wait (s):	signal (s):
s = s – 1	s = s + 1
if (s < 0)	if (s ≤ 0)
block the thread that called wait(s)	wake up one of the waiting threads
otherwise	
continue into CS	

- "Classical" version of semaphores:

wait (s):	signal (s):
if (s ≤ 0)	if (a thread is waiting)
block the thread that called wait(s)	wake up one of the waiting threads
s = s – 1	s = s + 1
continue into CS	

- Do both work? What is the difference??

Implementing Semaphores

- Implementing semaphores using *busy-waiting*:

wait (s):	signal (s):
while (s ≤ 0) do nothing; s = s – 1	s = s + 1

- Evaluation:
  - ✘ Waiting threads wastes time *busy-waiting* (doing nothing useful, wasting CPU time)
  - ✘ The code inside wait(s) and signal(s) is a critical section also, and it's not protected
  - ✘ Doesn't support a queue of multiple blocked threads waiting on the semaphore (why is this bad?)

Implementing Semaphores (cont.)

- Implementing semaphores (not fully) by *disabling interrupts*:

wait (s):	signal (s):
disable interrupts while (s ≤ 0) do nothing; s = s – 1 enable interrupts	disable interrupts s = s + 1 enable interrupts

- Evaluation:
  - ✔ Protects code inside wait(s) and signal(s) (but does this add any problems?)
  - ✘ Waiting threads wastes time *busy-waiting*
  - ✘ Doesn't support queue of blocked threads
  - ✘ Users can't disable interrupts
  - ✘ Can interfere with timer, which might be needed by other applications
  - ✘ Doesn't work on multiprocessors

Implementing Semaphores (cont.)

- Implementing semaphores (not fully) using a *test&set instruction*:

wait (s):	signal (s):

while (test&set(lk)!=0)
  do nothing;
while (s ≤ 0)
  do nothing;
s = s – 1
lk = 0
	while (test&set(lk)!=0)
	  do nothing;
	s = s + 1
	
	lk = 0

- Operation:
  - Lock "lk" has an initial value of 0 (free)
  - If "lk" is free (lk=0), test&set atomically:
    - reads 0, sets value to 1, and returns 0
    - since lock was free when tested, exit loop, but at the same time set the lock to busy
  - If "lk" is busy (lk=1), test&set atomically:
    - reads 1, sets value to 1, and returns 1
    - lock is busy, so keep looping until free

Implementing Semaphores (cont.)

- Test&set is an example of an atomic *read-modify-write* (RMW) instruction
  - RMW instructions atomically read a value from memory, modify it, and write the new value to memory
    - Test&set — on most CPUs
    - Exchange — Intel x86 — swaps values between register and memory
    - Compare&swap — Motorola 68xxx — read value, if value matches value in register r1, exchange register r1 and value
- Evaluation:
  - ✔ Can be made to work, even on multiprocessors (although there may be some cache consistency problems)
  - ✘ Waiting threads wastes time *busy-waiting*
  - ✘ Doesn't support queue of blocked threads waiting on the semaphore

The Dining Philosophers

- 5 philosophers live together, and spend most of their lives thinking and eating (primarily spaghetti)
  - They all eat together at a large table, which is set with 5 plates and 5 forks
  - To eat, a philosopher goes to his or her assigned place, and uses the two forks on either side of the plate to eat spaghetti
  - When a philosopher isn't eating, he or she is thinking
- Problem: devise a ritual (an algorithm) to allow the philosophers to eat
  - Must satisfy *mutual exclusion* (i.e., only one philosopher uses a fork at a time)
  - Avoids *deadlock* (e.g., everyone holding the left fork, and waiting for the right one)
  - Avoids *starvation* (i.e., everyone eventually gets a chance to eat)

The Dining Philosophers (Using Semaphores)

- First solution — doesn't work: (why not?)

```
philosopher-i ( )
    while (true)
        think;
        P(fork[i]);
        P(fork[i+1 mod 5]);
        eat;            /* critical section */
        V(fork[i]);
        V(fork[i+1 mod 5]);
```

- Second solution — only 4 eat at a time:

```
philosopher-i ( )
    while (true)
        think;
        P(room_at_table);
        P(fork[i]);
        P(fork[i+1 mod 5]);
        eat;            /* critical section */
        V(fork[i]);
        V(fork[i+1 mod 5]);
        V(room_at_table);
```


From Semaphores to Locks and Condition Variables

- A semaphore serves two purposes:
  - Mutual exclusion — protect shared data
    - mutex in Coke machine
    - milk in Too Much Milk
    - Always a binary semaphore
  - Synchronization — temporally coordinate events (one thread waits for something, other thread signals when it's available)
    - fullSlot and emptySlot in Coke machine
    - Either a binary or counting semaphore
- Idea — two separate constructs:
  - *Locks* — provide mutually exclusion
  - *Condition variables* — provide synchronization
  - Like semaphores, locks and condition variables are language-independent, and are available in many programming environments and OSs

Locks

- *Locks* provide mutually exclusive access to shared data:
  - A lock can be "locked" or "unlocked" (sometimes called "busy" and "free")
  - Before accessing shared data, call Lock::Acquire( ) on a specific lock
  - After accessing shared data, call Lock:: Release( ) on that same lock
- Example (here,"milk" is a lock):

Thread A	Thread B
milk->Acquire( );	milk->Acquire( );
if (noMilk)	if (noMilk)
buy milk;	buy milk;
milk->Release( );	milk->Release( );

- Can be implemented:
  - Trivially using binary semaphores
  - Using lower-level constructs

Locks vs. Condition Variables

- Consider the following code:

```
Queue::Add( ) {
    lock->Acquire( );
    add item
    lock->Release( );
}
```

```
Queue::Remove( ) {
    lock->Acquire( );
    if item on queue
        remove item
    lock->Release( );
    return item;
}
```

  - Queue::Remove will only return an item if there's already one in the queue
- If the queue is empty, it might be more desirable for Queue::Remove to wait until there is something to remove
  - Can't just go to sleep — if it sleeps while holding the lock, no other thread can access the shared queue, add an item to it, and wake up the sleeping thread
  - Solution: **condition variables** will let a thread sleep inside a critical section, by releasing the lock while the thread sleeps

Condition Variables

- *Condition variables* coordinate events
  - After creating a new condition, the programmer must create a lock that will be associated with that condition variable
  - Condition::Wait(conditionLock) — release the lock and wait (sleep); when the thread wakes up, immediately try to re-acquire the lock; return when it has the lock
  - Condition::Signal(conditionLock) — if threads are waiting on the lock, wake up one of those threads and put it on the ready list; otherwise do nothing
- Can be implemented:
  - By higher-level constructs
  - Using binary semaphores
  - Using lower-level constructs, much like semaphores are implemented

Using Locks and Condition Variables

- Associated with a data structure is both a lock and a condition variable
  - Before the program performs an operation on the data structure, it acquires the lock
  - If it needs to wait until another operation puts the data structure into an appropriate state, it uses the condition variable to wait
- Unbounded-buffer producer-consumer:

```
Lock *lk;
Condition *c;

/* producer */
while (1) {
    lk->Acquire( );
    produce next item
    avail++;
    c->Signal(lk)
    lk->Release( );
}
```

```
int avail = 0;

/* consumer */
while (1) {
    lk-> Acquire( );
    if (avail==0)
        c->Wait(lk);
    consume next item
    avail--;
    lk->Release( );
}
```


Monitors

- A *monitor* is a programming-language abstraction that automatically associates locks and condition variables with data
  - A monitor includes private data and a set of atomic operations (member functions)
    - Only one thread can execute (any function in) monitor code at a time
    - Monitor functions access monitor data only
    - Monitor data cannot be accessed outside
  - A monitor also has a lock, and (optionally) one or more condition variables
    - Compiler automatically inserts an acquire operation at the beginning of each function, and a release at the end
- Special languages that supported monitors were popular with some OS people in the 1980s, but no longer
  - Now, most OSs (OS/2, Windows NT, Solaris) just provide locks and CVs

The Dining Philosophers (Using Locks and CVs)

```
#define N 5
enum philosopher-state (thinking,hungry,eating);
Lock mutex;
Condition self[N];
philosopher-state state[N];

void pickup (int i) {
    mutex.Acquire();
    state[i] = hungry;
    test(i);
    if (state[i] != eat)
        self[i].Wait(mutex);
    mutex.Release();
}

void putdown (int i) {
mutex.Acquire();
state[i] = thinking;
      test((i+N–1) % N);
test((i+1) % N);
mutex.Release();
}

Void test (int k) {
    if ((state([k+N–1) % N] != eat) &&
        (state[k] == hungry) &&
        state[(k+1) % N] != eat)) {
           state[k] = eat;
           self[k].Signal(mutex);
     }
}
```