Details of Semaphore Operation

- 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

Two Versions of Semaphores

- Semaphores from last time (simplified):

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

- "Classical" version of semaphores:

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

- Do both work? What is the difference??

Semaphores in Nachos

- The class Semaphore is defined in **threads/synch.h** and **synch.cc**
  - The classes Lock and Condition are also defined , but their member functions are empty (implementation left as exercise)
- Interesting functions:
  - Semaphores:
    - Semaphore::Semaphore( ) — creates a semaphore with specified name & value
    - Semaphore::P( ) — semaphore wait
    - Semaphore::V( ) — semaphore signal
  - Locks:
    - Lock::Acquire( )
    - Lock::Release( )
  - Condition variables:
    - Condition::Wait( )
    - Condition::Signal( )

Semaphores in Nachos

```
void
Semaphore::P()
{
   IntStatus oldLevel = interrupt->
      SetLevel(IntOff);    // disable interrupts

   while (value == 0) {        // sema not avail
      queue->                  // so go to sleep
          Append((void *)currentThread);
      currentThread->Sleep();
   }

   value--;             // semaphore available,
                        // consume its value

   (void) interrupt->    // re-enable interrupts
      SetLevel(oldLevel);
}
```

Semaphores in Nachos (cont.)

```
void
Semaphore::V()
{
   Thread *thread;

   IntStatus oldLevel = interrupt->
      SetLevel(IntOff);

   thread = (Thread *)queue->Remove();
   if (thread != NULL) // make thread ready,
              // consuming the V immediately
      scheduler->ReadyToRun(thread);

   value++;

   (void) interrupt->SetLevel(oldLevel);
}
```


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! */
}
```


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

Locks

- *Locks* provide mutually exclusive access to shared data:
  - A lock can be "locked" or "unlocked" (sometimes called "busy" and "free")
- Operations on locks (Nachos syntax):
  - Lock(*name) — create a new (initially unlocked) Lock with the specified name
  - Lock::Acquire( ) — wait (block) until the lock is unlocked; then lock it
  - Lock::Release( ) — unlock the lock; then wake up (signal) any threads waiting on it in Lock::Acquire( )
- Can be implemented:
  - Trivially by binary semaphores (create a private lock semaphore, use P and V)
  - By lower-level constructs, much like semaphores are implemented

Locks (cont.)

- Conventions:
  - Before accessing shared data, call Lock::Acquire( ) on a specific lock
    - Complain (via ASSERT) if a thread tries to Acquire a lock it already has
  - After accessing shared data, call Lock:: Release( ) on that same lock
    - Complain if a thread besides the one that Acquired a lock tries to Release it
- Example of using locks for mutual exclusion (here, "milk" is a lock):

Thread A	Thread B

milk–>Acquire( );
if (noMilk)
    buy milk;
milk–>Release( );
	milk–>Acquire( );
	if (noMilk)
	    buy milk;
	milk–>Release( );

  - The test in threads/threadtest.cc should work exactly the same if locks are used instead of semaphores

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 <u>inside</u> a critical section, by releasing the lock while the thread sleeps

Condition Variables

- *Condition variables* coordinate events
- Operations on condition variables (Nachos syntax):
  - Condition(*name) — create a new instance of class Condition (a condition variable) with the specified name
    - After creating a new condition, the <u>programmer</u> must call Lock::Lock( ) to 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 <u>one</u> of those threads and put it on the ready list; otherwise do nothing

Condition Variables (cont.)

- Operations (cont.):
  - Condition::Broadcast(conditionLock) — if threads are waiting on the lock, wake up <u>all</u> of those threads and put them on the ready list; otherwise do nothing
- **Important**: a thread **<u>must</u>** hold the lock before calling Wait, Signal, or Broadcast
- Can be implemented:
  - Carefully by higher-level constructs (create and queue threads, sleep and wake up threads as appropriate)
  - Carefully by binary semaphores (create and queue semaphores as appropriate, use P and V to synchronize)
    - Does this work? More on this in a few minutes…
  - Carefully by 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;                  int avail = 0;
Condition *c;
                           /* consumer */
/* producer */             while (1) {
while (1) {                    lk-> Acquire( );
    lk->Acquire( );            if (avail==0)
    produce next item              c->Wait(lk);
    avail++;                   consume next item
    c->Signal(lk)              avail--;
    lk->Release( );            lk->Release( );
}                          }
```


Comparing Semaphores and Condition Variables

- Semaphores and condition variables are pretty similar — perhaps we can build condition variables out of semaphores
- Does this work?

```
Condition::Wait( ) {       Condition::Signal( ) {
    sema->P( );                sema->V( );
}                          }
```

  - No, we're going to use these condition operations inside a lock. What happens if we use semaphores inside a lock?
- How about this?

```
Condition::Wait( ) {       Condition::Signal( ) {
    lock->Release( );          sema->V( );
    sema->P( );            }
    lock->Acquire( );
}
```

  - How do semaphores and condition variables differ with respect to keeping track of history?

Comparing Semaphores and Condition Variables (cont.)

```
Condition::Wait( ) {       Condition::Signal( ) {
    lock->Release( );          sema->V( );
    sema->P( );            }
    lock->Acquire( );
}
```

- Semaphores have a value, CVs do not!
- On a **<u>semaphore</u>** signal (a V), the value of the semaphore is always incremented, even if no one is waiting
  - Later on, if a thread does a semaphore wait (a P), the value of the semaphore is decremented and the thread **<u>continues</u>**
- On a **<u>condition variable</u>** signal, if no one is waiting, the signal has no effect
  - Later on, if a thread does a condition variable wait, it **<u>waits</u>** (it **always** waits!)
  - It doesn't matter how many signals have been made beforehand

Two Kinds of Condition Variables

- Hoare-style (named after C.A.R. Hoare, used in most textbooks including *OSC*):
  - When a thread performs a Signal( ), it gives up the lock (and the CPU)
    - The waiting thread <u>is picked as the next thread that gets to run</u>
  - Previous example uses Hoare-style CVs
- Mesa-style (used in Mesa, Nachos, and most real operating systems):
  - When a thread performs a Signal( ), it keeps the lock (and the CPU)
    - The waiting thread <u>gets put on the ready queue</u> with no special priority
      - There is **<u>no guarantee</u>** that it will be picked as the next thread that gets to run
      - Wore yet, another thread may even run and acquire the lock before it does!
  - When using Mesa-style CVs, **<u>always</u>** surround the Wait( ) with a "while" loop