## Previous lecture overview

- Semaphores provide the first high-level synchronization abstraction that is possible to implement efficiently in OS.
  - This allows avoid using ad hoc Kernel synchronization techniques like non-preemptive kernel
  - allows to implement in multiprocessors
- problems
  - programming with semaphores is error prone - the code is often cryptic:
    - a semaphore combines the counting mechanism and synchronization mechanism



## Lecture 13: locks and condition variables

- problems with semaphores
- locks and CVs
  - definition and usage
  - solutions to synchronization problems
  - implementation



## What's wrong with semaphores?

- Besides other shortcomings programming with semaphores is deadlock - prone

```
P(fridge);                    V(fridge);
if (noMilk){                  if (noMilk){
        buy milk;                 buy milk;
        noMilk=false;             noMilk=false;
}                             }
P(fridge);                    P(fridge);
```

  - are these programs correct? what's wrong with them?
- solution — new language constructs
  - (Conditional) Critical region
    - **region** v **when** B **do** S; variable v is a shared variable that can only be accessed inside the critical region
    - Boolean expression B governs access
    - Statement S ( critical region) is executed only if B is true; otherwise it blocks until B becomes true
    - can prevent some simple programming errors but still problematic
  - monitors (somewhat similar to locks)



## Semaphore=Lock+Condition Variable

- semaphore serves two purposes:
  - Mutual exclusion — protect shared data
    - milk - in too much milk
    - buffer in producer/consumer
    - shared resource in readers/writers
    - forks in Dining philosophers
  - temporal coordination of events (one thread waits for something, other thread signals when it's available)
    - stop the roommate from going to the store while you are out to get milk
    - suspend producer when buffer is full, consumer - when empty
    - what is the coordination in readers/writers and dining philosophers?
- idea — two separate constructs:
  - *Locks* — provide mutual exclusion
  - *Condition variables* — provide conditional 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") initially it is unlocked
  - a thread is said to have (*own*) the lock if it successfully executed **`acquire`** statement.
  - If other threads attempt to acquire a lock - they are suspended
  - to achieve mutually exclusive access to variables threads should access them only inside acquire/release statements

```
Thread A               Thread B
acquire(milk );        acquire(milk );
if (noMilk)            if (noMilk)
    buy milk;              buy milk;
release(milk);         release(milk );
```



## Locks, why do we need anything else?

- 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

```
Queue::Add(int *item){
   lock->Acquire();
   /* add item to queue
*/
   lock->Release();
}
Queue::Remove( )  {
   int *item;
   lock->Acquire( );

   if (!queue-
>empty()){
      /* remove item
         from queue */
   }
   lock->Release();
   return(item);
}
```

## Condition variables

- condition variable (CV) coordinates events
- three basic operations on CVs:
  - **`wait -`** blocks the thread and releases the associated lock
  - **`signal`** - if threads are waiting on the lock, wake up one of those threads and put it on the ready list; otherwise do nothing
  - **`broadcast`** — if threads are waiting on the lock, wake up all of those threads and put them on the ready list; otherwise do nothing
- problem: when a thread P wakes up another Q they are technically both inside the protected area. Which thread is allowed to proceed?
  - P (Hoare style) – seems "logical" but the awakened thread may miss the condition
  - Q (Hansen style) – what to do with signaling thread?
  - P proceeds but immediately releases the lock – can wake up only one thread
- all these techniques are implemented and equivalent in power



## Using locks and CVs for producer /consumer problem

```
conditionvar *cv;
lock *lk;
int avail=0;

/* producer */
while(1){
  acquire(lk);
  /* produce next */
  avail++
  signal(cv,lk);
  release(lk);
}

/* consumer */
while(1){
   acquire(lk);
   if(avail==0)
      wait(cv,lk);
   /* consume next */
   avail--;
   release(lk);
}
```

- Unbounded producer/consumer with locks and CVs
- 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



## Using locks and CVs for readers/writers problem

```
conditionvar wrt, rdr;
int nr=0, nw=0;
lock lk;

writer() {
      acquire(lk);
      while(nr>0 || nw>0)
          wait(wrt,lk);
      nw++;
      release(lk);
      /* perform write */
      acquire(lk);
      nw--;
      signal(wrt);
      broadcast(rdr);
      release(lk);
}
```

```
reader() {
      acquire(lk);
      while(nw>0)
          wait(rdr,lk);
      nr++;
      release(lk);
      /* perform read */
      acquire(lk);
      nr--;
      if(nr==0)
         signal(wrt);
      release(lk);
   }
```

- notice the use of **`broadcast`** to wake up all readers
- is this a readers or writers preference solution?



## Locks/CVs implementation

- the issues related to implementation of semaphores and locks/CVs are similar
- spinlock - a locked process does not release CPU but rather "spins" constantly checking the lock until it opens
- sleeplock - a locked process blocks and is put back on the ready queue only when the lock is open



## Spinlock implementation

- Simplest implementation of locks - set up a boolean variable (**`*s`**) is by busy waiting and constantly checking on it's value with atomic RMW instruction like test&set (**`testnset`**)
- problem - test&set monopolizes memory access and degrades system performance
- solution - have two **`while`** loops check by test&set once - if locked - check with regular read until unlocked
- what's the problem with both of these solutions?
- Unfair!

variant 1

```
void spin_lock (bool *s) {
      while (testnset(*s))
       ;
}
void spin_unlock (bool *s) {
      *s=FALSE;
}
```

variant 2

```
void spin_lock (bool *s) {
   while (testnset(*s))
      while (*s)
       ;
}
void spin_unlock (bool *s) {
      *s=FALSE;
}
```



## Implementing CV using spinlocks

```
/* condition consists of:
  list - waiting threads
  listlock - lock protecting
       operation on list*/
void wait(condition *c,
          lock *s){
   spinlock(c->listlock);
   /* add self to list */
   spinunlock(c->listlock);
   unlock(s);
   /* block current thread */
   lock(s);
   return;
}
void signal(condition *c){
   spinlock(c->listlock);
   /* remove a thread from
     list if list not empty */
   spinunlock(c->listlock);
   /* make removed thread
     runnable */
}
```

- the CV contains a list that holds the waiting threads, the operations on this list are protected by a spinlock
- note the difference between this spinlock - the internal CV lock and **`s -`** the external lock that is used in association with the CV