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
  - for signal and wait to be atomic on multiprocessor architecture - a low level locking primitives (like test&set instruction) need to be available
  - blocking and unblocking require context switch - performance degradation
  - no means of finding out whether the thread will block on semaphore
  - convoys

Lecture 13: locks and condition variables

- Problems with semaphores
- locks
  - definitions and usage
  - implementation
    - spinlocks
    - sleeplocks
- condition variables
  - definition and usage
    - unbouded producer/consumer problem
    - dining philosophers problem
  - implementing CVs

What's wrong with semaphores?

- Besides other shortcomings programming with semaphores is deadlock - prone

```
milk–>V( );            milk–>P( );
if (noMilk)            if (noMilk)
     buy milk;              buy milk;
milk–>P( );            milk–>P( );
```

  - 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 - convoluted and seldom used

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 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") initially it is unlocked
  - a thread is said to have (*own*) the lock if it successfully executed `lock` statement.
  - If other threads attempt to execute a lock - they are suspended
  - to achieve mutually exclusive access to variables threads should access them only inside lock/unlock statements

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


Spinlocks and sleeplocks

- locks can be implemented differently depending on its use:
- spinlock - a locked process does not release CPU but rather "spins" constantly checking the lock until it opens
  - advantages
    - fast - the process proceeds as soon as the lock is open
    - may save time for locks that are held for short time - no context switching
  - disadvantages
    - wasteful for locks that are held long - the process wastes CPU cycles spinning
    - cannot be used on uniprocessor systems. Why?
- sleeplock - a locked process blocks and is put back on the ready queue only when the lock is open
  - advantages
    - can be used on uniprocessor
    - saves CPU time on locks held long

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!

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

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


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
- CV is associated with a *predicate* (an expression that evaluates to either true of false) and a lock;
- three basic operations on CVs:
  - **wait -** blocks the thread and releases the associated lock
  - **signal** - 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
  - **broadcast** — 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
- the predicate is tested <u>outside</u> of the CV primitives
- the lock is closed and (sometimes) released <u>outside</u> of CV

Using locks and CVs for producer /consumer problem

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

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

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

- 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 dining philosophers problem

```
mutex:  lock;
self:  array [0..N–1] of condition;
state:  array [0..N–1] of
  (think,hungry,eat)
             initially all thinking

test (int k) {
    if ((state[k+N–1 mod N] != eat) &&
        (state[k] == hungry) &&
         state[k+1 mod N] != eat)) {
      state[k] = eat;
      signal(self[k]);
    }
}
```

```
pickup (int i) {
    acquire(mutex);
    state[i] = hungry;
    test(i);
    if (state[i] != eat)
      wait(self[i],mutex);
    release(mutex);
}

putdown (int i) {
    acquire(mutex);
    state[i] = thinking;
    test(i+N–1 mod N);
    test(i+1 mod N);
    release(mutex);
}
```

- The philosophers try to acquire the forks until they succeed
- does this solution ensure MX? Fairness?
- does a process need to know about non-neighbors?

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
- <u>note</u> the difference between this spinlock - the internal CV lock and **s -** the external lock that is used in association with the CV