Using Locks and Condition Variables (Review)

- 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 **semaphore** 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 **continues**
- On a **condition variable** signal, if no one is waiting, the signal has no effect
  - Later on, if a thread does a condition variable wait, it **waits** (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 is picked as the next thread that gets to run
  - 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 gets put on the ready queue with no special priority
      - There is **no guarantee** 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, **always** surround the Wait( ) with a "while" loop

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

- 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);
```


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);
    }
}
```