• Condition Variables
  o Introduction
    ▪ Mutexes are good for controlling access to a critical region
    ▪ Semaphores are good for situations where counting is needed
    ▪ Sometimes we want to block until some condition is met
      • Try to use the sleep/wakeup paradigm again

• Producer-consumer problem solution

```c
#define N 100
int count = 0;
mutex the_mutex;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item(); /* repeat forever */
        if (count == N) sleep(); /* if buffer is full, go to sleep */
        insert_item(item); /* put item in buffer */
        count = count + 1; /* increment count of items in buffer */
        if (count == 1) wakeup(consumer); /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        sleep(); /* repeat forever */
        if (count == 0) sleep(); /* if buffer is empty, go to sleep */
        item = remove_item(); /* take item out of buffer */
        count = count - 1; /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item); /* print item */
    }
}
```

• Race condition exists
  ▪ Buffer is empty and consumer reads count as zero
  ▪ Producer is then given the CPU before sleep() is called
  ▪ Producer adds an item, increments count, and calls wakeup() on the consumer
  ▪ Because the consumer has not called sleep(), the wakeup() call is lost
  ▪ When consumer runs next, it will immediately sleep.
  ▪ Producer will never call wakeup again, will fill up the buffer, then sleep itself
  ▪ Both sleep forever

• Using mutual exclusion on count doesn’t help
void producer(void)
{
    int item;
    while(1){
        item = produce_item();
        mutex_lock(&the_mutex);
        if( count == N ){
            mutex_unlock(&the_mutex);
            sleep();
            mutex_lock(&the_mutex);
        }
        insert_item(item);
        count = count + 1;
        if( count == 1 ) wakeup(consumer);
        mutex_unlock(&the_mutex);
    }
}

void consumer(void)
{
    int item;
    while(1){
        mutex_lock(&the_mutex);
        if( count == 0 ){
            mutex_unlock(&the_mutex);
            sleep();
            mutex_lock(&the_mutex);
        }
        item = remove_item();
        count = count - 1;
        if( count == N - 1 ) wakeup(producer);
        mutex_unlock(&the_mutex);
        consume_item(item);
    }
}

- Race condition now
  - Same situation as before, but now consumer is preempted right after calling `mutex_unlock()` and before `sleep()`.
  - Producer still calls `wakeup()` while consumer is not logically asleep.
  - Both eventually sleep forever
  - We can’t remain in the critical region while sleeping or producer will never wake up consumer

- Solution – condition variables
  - Method
    - Provides atomicity for the unlock and sleep operations.
    - Requires that the mutex be locked prior to `wait()` call
    - Automatically and atomically unlocks mutex and then calls `sleep()`
    - When `wait()` returns, the mutex is automatically relocked
  - Code
```c
#define N 100
int count = 0;
mutex the_mutex;
cond_var condc, condp;

void producer(void)
{
    int item;
    while(1){
        item = produce_item();
        mutex_lock(&the_mutex);
        while( count == N ) cond_wait(&condp, &the_mutex);
        insert_item(item);
        count = count + 1;
        if( count == 1 ) cond_signal(&condc);
        mutex_unlock(&the_mutex);
    }
}

void consumer(void)
{
    int item;
    while(1){
        mutex_lock(&the_mutex);
        while( count == 0 ) cond_wait(&condc, &the_mutex);
        item = remove_item();
        count = count - 1;
        if( count == N - 1 ) cond_signal(&condp);
        mutex_unlock(&the_mutex);
        consume_item(item);
    }
}
```

- **Implications**
  - Because unlocking the mutex and sleeping are atomic, a proper signal cannot be lost
  - The while loop is necessary to avoid problems due to spurious wakeups
  - Condition variables can be used for more arbitrary blocking conditions (not necessarily counting like with semaphores)
  - Multiple conditions can also be used in the **while** statement if desired

- **Barriers – Another synchronization mechanism**
  - **Introduction**
    - Example: Consider a set of files, where each must be read into memory and preprocessed
      - After **all** have been preprocessed, post-processing can start
      - No thread/process may start post-processing until all have completed pre-processing
    - Barriers can be used to achieve this behavior
All threads are blocked at the barrier until the configurable number of threads has reached the barrier.

Common uses
- Number crunching on matrices
  - Parallel operations can occur on a single matrix, but the matrix must be complete for the next operation to begin
- Allow multiple threads to do preprocessing before starting their actual work
  - Example: scanner threads
    - Sender thread – creates packet buffers
    - Receiver thread – opens various files for writing
    - IP generation thread – reads large blacklist from a file, creates a custom data structure for fast lookups
  - To avoid backlog, they must all be ready to start their work before any packet is sent

- Linux System Calls
  - pthread_barrier_init(pthread_barrier_t *barrier, attrs, unsigned int count)
    - Initializes barrier as a barrier, with count being the threshold when threads will be released
    - count must be greater than zero
  - pthread_barrier_destroy(pthread_barrier_t *barrier)
    - Undefined result if called when threads are blocking on barrier
  - pthread_barrier_wait(pthread_barrier_t *barrier)
    - Calling thread blocks until required number of threads have called the same function
    - When the threshold is reached, all threads waiting at the barrier are awoken
      - One receives return value of PTHREAD_BARRIER_SERIAL_THREAD
- The rest receive a return value of zero
- State of the barrier is returned to that of the most recent pthread_barrier_init() call

- PTHREAD_BARRIER_SERIAL_THREAD might be used by the programmer to ensure that only one thread attempts to reset other shared variables