The following solutions to the concurrent programming problems show one way of solving the problem. There may be multiple correct solutions. These solutions use semaphores, although similar solutions can be created using Java synchronized methods.

**Limited producer / consumer** – Consider the classical producer / consumer problem with a limited queue. Many producer threads create DoDads and put them on a queue while many consumer threads get DoDads from the queue. Write the methods putDoDad and getDoDad. If a thread calls getDoDad and the queue is empty, it should be suspended until a DoDad is available. If a producer thread call putDoDad and there are already $N$ DoDads on the queue, it will have to wait.

```java
semaphore mutex = 1, dodadSem = 0, qSpace = N;

DoDad getDoDad() {
    p(dodadSem);
    p(mutex);
    get DoDad from queue;
    v(mutex);
    v(qSpace);
    return DoDad;
}

void putDoDad(DoDad thing) {
    p(qSpace);
    p(mutex);
    put DoDad on queue;
    v(mutex);
    v(dodadSem);
}
```
Santa Claus – Consider a program with one Santa thread, nine reindeer threads and many elf threads. The Santa thread should rest unless all nine reindeer arrive or three or more elves have problems. Once all nine reindeer call the arrive method, Santa should call the sleigh method. If three or more elves call the helpNeeded method, Santa should call the workshop method. The elves and reindeer do not need to wait for Santa.

Semaphore santaSleep = 0; mutex = 1;  
int numReindeer = 0; numElves = 0;  
Santa  
   p(santaSleep);  
   p(mutex);  
   if (numReindeer == 9) sleigh();  
   if (numElves >= 3) helpNeeded();  
   v(mutex);  
} 
reindeer  
   p(mutex);  
   numReindeer++;  
   if(numReindeer == 9) v(santaSleep);  
   v(mutex);  
} 
elf  
   p(mutex);  
   numElves++;  
   if(numElves == 3) v(santaSleep);  
   v(mutex);  
}
**Check for prime** – Check if a given number is a prime number. This can be done by attempting to divide it by smaller numbers. If it does not evenly divide by any smaller number, then it is prime. Use K threads to make the program run faster.

To test if a number is a prime, you can try to divide it by all of the prime numbers smaller than the square root of the number. We will assume we have a list of prime numbers. Each of the K threads will try to divide the number by every Kth prime in the list.

```java
int numThreads = K; 
semaphore mutex = 1; 
boolean done = false;       // true when answer found  
start K-1 threads (main thread continues as one of the worker threads)  
while (NOT done || more divisions to do) {
   if (it divides evenly) done = true;
}
mutex.p();
numThreads--;  
if( numThreads > 0) {
   mutex.v();
   terminate thread;
}
mutex.v();  
Print solution;
```
Crossing the river – Somewhere near Redmond, Washington there is a rowboat that is used by both Linux hackers and Microsoft employees (serfs) to cross a river. The ferry can hold exactly four people; it won’t leave the shore with more or fewer. To guarantee the safety of the passengers, it is not permissible to put one hacker in the boat with three serfs, or to put one serf with three hackers. Any other combination is safe. As each thread boards the boat it should invoke a method called board. You must guarantee that all four threads from each boatload invoke board before any of the threads from the next boatload do. After all four threads have invoked board, exactly one of them should call a method named rowBoat, indicating that that thread will take the oars. It doesn’t matter which thread calls the function, as long as one does.

```c
semaphore mutex = 1, linux = 0, ms = 0;
int numHackers = 0, numSerfs = 0;
hacker{
    p(mutex);
    numHackers++;
    if( numHackers == 4) {
        v(linux); v(linux); v(linux);
        numHackers -= 4;
        v(mutex);
        getOnBoard ();
    } else if(numHackers >= 2 && numSerfs >= 2) {
        v(linux); v(ms); v(ms);
        numHackers -= 2;
        numSerfs -= 2;
        v(mutex);
        getOnBoard ();
    } else {
        v(mutex);
        p(linux);
        getOnBoard ();
    }
}

Semaphore boardMutex = 1;
int numBoarded = 0;
getOnBoard () {
    p(boardMutex);
    numBoarded++;
    board();
    if (numBoarded == 4) {
        rowboat ();
        numBoarded = 0;
    }
    v(boardMutex);
}
```
**List update** – Three kinds of threads share access to a singly-linked list: searchers, inserters, and deleters. Searchers merely examine the list; hence they can execute concurrently with each other. Inserters add new items to the end of the list. Insertions must be mutually exclusive to preclude two inserters from inserting new items at about the same time. However, one insert can proceed in parallel with any number of searches. Finally, deleters remove items from anywhere in the list. At most one deleter process can access the list at a time, and deletion must also be mutually exclusive with searches and insertions. Write code for searchers, inserters, and deleters.

```c
Semaphore notSearching = 1, notInserting = 1, insertSem = 1,
searchMutex = 1, insertMutex = 1;
int numSearcher = 0, numInserter = 0;

searcher {
  p(searchMutex);               // lock shared variables
  numSearcher++;               // lock right to search
  if( numSearcher == 1) p(notSearching);      // lock right to search
  v(searchMutex);              // unlock shared variables
  // search “Therefore search and see if there is not some place where you may invest your humanity.”
  p(searchMutex);              // lock shared variables
  numSearcher--;              // no more searchers
  if( numSearcher == 0) v(notSearching);     // no more searchers
  v(searchMutex);
}

inserter {
  p(insertMutex);              // lock shared variables
  numInserter++;              // lock right to insert
  if( numInserter == 1) p(notInserting); // lock right to insert
  v(insertMutex);              // limit to one inserter
  p(insertSem);               // if debugging is the art of removing bugs, then programming must be the art of inserting them.
  v(insertSem);               // allow another inserter
  numInserter--;             // last inserter done
  if( numInserter == 0) v(notInserting);     // last inserter done
  v(insertMutex);
}

deleter {
  p(notSearching);           // wait for no searchers
  p(notInserting);           // wait for no inserters
  // delete - “Delete the negative; Accentuate the positive!”
  v(notInserting);
  v(notSearching);
}
```
Train bridge – Consider a deep river gorge crossed by a bridge that holds a single set of railroad tracks. Trains can go across the bridge either direction, east to west or west to east, but not at the same time. If two trains are going in opposite directions, they will crash. The bridge is only strong enough to hold four trains at a time. Write the methods east2west and west2east that allow trains to use the bridge safely.

In the following solution there is one semaphore to lock access to the bridge. The first train going in either direction locks the bridge preventing trains from traveling in the other direction. If the first train in a direction gets suspended on the bridge semaphore, this will block all other trains in that direction since the mutual exclusion semaphore is still locked. Once the first train has access to the bridge, all trains in that direction will be allowed through the mutex area. The limit semaphores limit the number of trains on the bridge. Note that this solution can starve trains in one direction if there are always trains in the opposite direction.

Semaphore bridge = 1, eMutex = 1, wMutex = 1, eLimit = 4, wLimit = 4;
int goingE = 0, goingW = 0;

east2west() { //west2east is similar
    p(eMutex);    // lock access to counter
    goingE++;     // count trains in this direction
    if( goingE == 1) p(bridge); // lock bridge if first train
    v(eMutex);    // allow other trains in this direction
    p(eLimit);    // Limit to 4 trains on the bridge
    // cross the bridge
    v(eLimit);    // allow another train on the bridge
    p(eMutex);    // lock access to counter
    goingE--;
    if( goingE == 0) v(bridge); // if last train, unlock bridge
    v(eMutex);    // unlock counter
}
**Dynamic resources** – Consider a system with a large number of threads. Periodically the threads will need to acquire specific resources. We can assume there are 10 possible resources. A thread will request exclusive access to resources which the threads will specify. The threads will want 2 – 4 of these resources. The threads will behave as:

```java
boolean resourcesWanted[10];
randomly set 2 to 4 of resourcesWanted to true and the rest to false;
```

Call `getResources(resourcesWanted)` which returns the resources requested or waits until they are available.

Write the `getResources` and `releaseResources` methods.

```java
boolean resources[10]; // true if resource available else false
boolean wanted[10][N]; // vector of desired resources
Semaphore waiting[N] = 0; // semaphore for each waiting thread
Semaphore mutex = 1;

getResources(resourcesWanted) {
    do forever {
        // repeat until return
        boolean allAvail = true;
        p(mutex);
        for (i = 0; i < 10; i++)
            if (resourcesWanted[i])
                allAvail &= resources[i]; // check if all avail
        if (allAvail) {
            // if all available
            for (i = 0; i < 10; i++)
                if (resourcesWanted[i])
                    resources[i] = false; // mark resources used
            v(mutex);
            return;
        } else {
            // if not available
            copy resourcesWanted into next slot in wanted
            v(mutex);
            p(waiting[next slot]);
        }
    }
}

goalResources(resourcesWanted) {
    for (i = 0; i < 10; i++)
        if (resourcesWanted[i])
            resources[i] = true; // mark resources available
    // Check if a waiting thread now has all resources need
    // by comparing resources to each row in wanted
    if (a thread has all resources) v(waiting[row]);
}