1) Exercise 2.15a
   Y will get the value of 5. When core 1 tries to read x from memory, it will first miss, then the dirty value of x=5 at core 0 will be written to the shared memory. Once complete, core 1 can correctly read the value for x.
2) Exercise 4.11
   a. If two successive elements in the list are deleted by two separate threads, the previous element's pointer will not correctly point to its new successive element.
   b. If the insert and delete are being performed at the same element location in the list, the previous element's pointer will either be pointing to the new element or the new element's successor.
   c. If the threads are accessing the same element and the deletion thread completes first, this causes problems for the access to that member.
   d. If two elements are inserted at the same position, they will both be pointing to the same successor, but the previous element will only be pointing to one of the new elements.
   e. Similarly to C, if the threads execute on the same position, then there may be issues if the insert completes first.
3) Exercise 4.16
   Assuming the vector y is evenly divided across the threads, thread 0 will span y[0] to y[1999] and thread 2 will span y[4000] to y[5999]. For false sharing to occur, we would see the cache line beginning at the element in thread 0 that is closest to the elements spanned by thread 2. With a cache line of 64 bytes, we would not see this happen for either thread 2 or thread 3.
4) Exercise 4.17
   a. The minimum number of cache lines would be 1 as we can fit 8 elements on a single cache line.
   b. The maximum number of cache lines would be 2 if the chunk of 8 elements does not begin at the first element of a cache line.
   c. There is only one single way, if the boundaries coincide.
   d. There would only be three ways to assign the four threads across two processors. 0 and 1, 0 and 2, or 0 and 3 on one processor, and the others on the second processor.
   e. This scenario would be achieved when assigning threads 0 and 1 to a processor and threads 2 and 3 to the other processor
   f. There are 8 ways to assign y to the cache line. With 3 different ways of assigning threads, that gives us 24 assignments of components to cache lines and threads to processors.
   g. Of all assignments, only when threads 0 and 1 are assigned the same processor is there a chance of false sharing. From this, the only single possibility is the scenario from question e

5) Programming assignment

```
//Code by Joey Black and Curtis Maher

#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define CACHE_LINE_SIZE 128

void matrixMultiply(const int, const double *, const double *, double *);
void blockedMatrixMultiply(const int M, const double *A, const double *B, double *C);
void subMatrixMultiply(const int M, const double *A, const double *B, double *C,
                const int rowStartA, const int columnStartB, const int columnStartA,
                const int nRowStartA, const int nColumnStartB, const int nColumnStartA);

int main()
{
   //Use clock for stats on algo and time for randomization
   clock_t start, end;
   time_t t;
   double cpu_time_used;

   const int size = 2000;
   double *A = (double *) malloc(size * size * sizeof(double));
   double *B = (double *) malloc(size * size * sizeof(double));
   double *C1 = (double *) malloc(size * size * sizeof(double));
   double *C2 = (double *) malloc(size * size * sizeof(double));

   //Initialize all the matrices
   srand((unsigned) time(&t));
   for(int i = 0; i < size; i++)
   {
      for(int j = 0; j < size; j++)
      {
         A[size * i + j] = rand() % 1000;
         B[size * i + j] = rand() % 1000;
         C1[size * i + j] = 0;
         C2[size * i + j] = 0;
      }
   }

   //Run naive algorithm and get the time
   start = clock();
   matrixMultiply(size, A, B, C1);
   end = clock();
   cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
   printf("%f\n", cpu_time_used);

   //Run the block algorithm and get the time
   start = clock();
   blockedMatrixMultiply(size, A, B, C2);
   end = clock();
```

```
   cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
   printf("%f\n", cpu_time_used);

   //Check for errors
   for(int i = 0; i < size * size; i++)
   {
      if(C1[i] != C2[i])
      {
         printf("fail\n");
         break;
      }
   }

   //Free all the memory
   free(A);
   free(B);
   free(C1);
   free(C2);
}

void matrixMultiply(const int M, const double *A, const double *B, double *C)
{
int i , j , k;
   for ( i = 0; i < M; ++i) {
      for ( j = 0; j < M; ++j) {
         for (k = 0; k < M; ++k)
            C[i*M + j] += A[i*M+k] * B[k*M+j];
      }
   }
}

void blockedMatrixMultiply(const int M, const double *A, const double *B, double *C)
{
   int i, j, k;
   int outsideValue = M % CACHE_LINE_SIZE;
   int cacheIndex = CACHE_LINE_SIZE;
   int rowASize, colBSize, colASize;

   //Make sure that the size is greater than the cache line size
   //If not, just use single block since it won't benefit from blocking
   if(M < cacheIndex)
   {
      cacheIndex = M;
      outsideValue = 0;
   }

   /*
      Goes through each block and does matrix multiplication
      A block is size CacheBlock x CacheBlock
      Handles matrices that aren't mutliples of block size
      This algo hovers around 15-20 times faster than Naive, something else is probably affected it
         but Im still willing to say its at least 10x faster
   */
```

```
   for(i = 0; i < M; i+= CACHE_LINE_SIZE)
   {
      for(k = 0; k < M; k+= CACHE_LINE_SIZE)
      {
         for(j = 0; j < M; j+= CACHE_LINE_SIZE)
         {
            rowASize = cacheIndex, colBSize = cacheIndex, colASize = cacheIndex;
            if(outsideValue != 0)
            {
               if(i + CACHE_LINE_SIZE >= M && outsideValue != 0)
                  rowASize = outsideValue;
               if(j + CACHE_LINE_SIZE >= M && outsideValue != 0)
                  colBSize = outsideValue;
               if(k + CACHE_LINE_SIZE >= M && outsideValue != 0)
                  colASize = outsideValue;
            }
            subMatrixMultiply(M, A, B, C, i, j, k, rowASize, colBSize, colASize);
         }
      }
   }
}

void subMatrixMultiply(const int M, const double *A, const double *B, double *C,
                  const int rowStartA, const int columnStartB, const int columnStartA,
                  const int nRowStartA, const int nColumnStartB, const int nColumnStartA)
{
   //Goes through a block and does matrix multiplication
   //Goes row by row through A and B and changes the write address everytime
   //This should make cache access on B easier since its going by row and not
   //By column
   int i, j, k;
   for(i = rowStartA; i < (nRowStartA + rowStartA); i++)
   {
      for(k = columnStartA; k < (nColumnStartA + columnStartA); k++)
      {
         for(j = columnStartB; j < (nColumnStartB + columnStartB); j++)
         {
            C[i*M + j] += A[i*M + k] * B[k*M + j];
         }
      }
   }
}
```