Fall 2014 CS 33
Lab 3. Cache Simulator

A cache is a memory bank sitting in between two others, one with faster speed and another with slower speed. When reading or writing from/to the slower memory, the cache retrieves data from memory addresses near the one desired so that if these nearby values are read or written, it can be done at the speed of the cache as opposed to the slower memory.

In this Lab, we will implement a cache simulator so that we can directly measure cache hit and miss performance based on spatial locality and other factors. We will measure the number of hits and misses (review pp 592-595 in the text) in a cache under different scenarios.

Recall the parameters of a E-way set associative cache: (E lines per set).

a certain number of sets 2^s = **S**,
a certain number of lines per set = **E** (1 < E < C/B where C = E*S*B)
a certain number of blocks per set 2^b = **B**.
a memory size 2^m = **M**.
a certain number of tag bits: **T** = m-(s+b)

But, when E = 1, it is a direct mapped cache.

when you build a cache, you decide how many sets, how many lines per block and how many blocks per line. Then the size of the cache, in relative terms is

S*B*E

so it covers M/(E*S*B) of the memory, which means that M/(E*S*B) memory addresses map to the same cache location.

In order to implement a read/write cache, we need to introduce a “dirty” bit to indicate that the cache contains a value which has not yet been written to memory. That is, we will implement a write-back cache (see page 612 in the text).

Our line replacement method shall be least recently used: LRU. We will also need an integer storing the last time the line was used.

The picture of a cache (direct mapped cache with # lines per set =1, n-way associative when E > 1)

```
Set 0:
  Line 1:      valid bit      dirty bit      tag value(T bits)      last used      blocks[B]
  Line 2:      valid bit      dirty bit      tag value(T bits)      last used      blocks[B]
   .
   .
  Line E:      valid bit      dirty bit      tag value(T bits)      last used      blocks[B]

Set 1:
  Line 1:      valid bit      dirty bit      tag value(T bits)      last used      blocks[B]
  Line 2:      valid bit      dirty bit      tag value(T bits)      last used      blocks[B]
   .
   .
  Line E:      valid bit      dirty bit      tag value(T bits)      last used      blocks[B]
.
.
Set s-1
```

or

```
struct cache_t
  {
  char  valid ;  // only values 0,1
  char  dirty ;  // only values 0,1
  int             tag  ;  // largest value 2^T-1
  int             last  ;  // call # last used
  int   *block ;          // data from memory
  } cache[S][E] ;
```

We have integers here but the hardware uses bit and usually *block is char.

our memory is

```
  int memory[M] ; or int *memory when you use malloc.
```

In our simulator, we will assume that each memory location holds an int. For testing, you should initialize memory[i] = i so that you can tell in debugs whether you are retrieving the correct data from memory. x and y will be “pointers” not in the usual sense but the integer location in memory[] where arrays x, y start. You can leave x at zero throughout. That is x, y and z is just an integer “addresses” pointing to a location in “memory”.

Ni and nj simulate two dimensional arrays x[ni][nj]: x[i][j] will be at location x+i*nj+j. When you vary ni and nj, you should make sure that the arrays do not overlap. Normally, x will start at location 0 so, y should start at least at location ni*nj. For our tests, we will have ni = nj.

There is a cachelab.c framework uploaded. Find the places where you should add code.

1) Fill in the initcache() function to initialize the cache

2) Fill in the readwritecache(int readwrite, int a, int *value, int *hitmiss, int voice) function. I have provided pseudo code.

You have one gratis function included in the framework: locationexample(). You could use that to test your simulator. I would suggest using the "voice" parameter to add debug information inside of readwritecache() to show values of readwrite, a, si, ta, bo, where a hit or miss is happening and then you will see what is going on. The sample just copies one linear array to another.

We will be dealing with transposing a two dimensional matrix: setting x[i][j] = y[j][i] in 2 nested loops. We will do it row-wise and col-wise (change j to the outer loop). We assume that x and y are distinct.

3) After you have the simulator working, add two transposition test cases duplicating the for(y loop in the locationexample():

3.1) one to transpose the y array into the x array row-wise and

3.2) transpose the y array into the x array column-wise (reverse the i and j loops).

Call stats(t) at the end for each test. You should see that the cache runs smoothly, even though y is in a bad location for copying from y to x.

Working set size examples. See wsexample() in the framework.

Set y at 20000.

4) Create a loop for ni from 88 to 120 in steps of 8, setting nj = ni. This will put the working set size over the limit of the cache size. Transpose y into x row-wise. Show stats. You should see problems with write hits.

5) Add 8x blocking to 4). remember?

```
for( ii=0; ii<ni; ii=ii+8 )
```

```
for( jj=0; jj<nj; jj=jj+8 )
  for( i=ii; i<ii+8; i++ )
    for( j=jj; j<jj+8; j++ )
```

6) Create a loop for ni from 88 to 120, setting nj = ni. This will put the working set size over the limit of the cache size. Transpose y into x col-wise. Show stats. You should see problems with read hits.

7) Add 8x blocking to 6.

When you run your program, save the output to a .txt file. This will be one of your submissions.

8) Change the #define for E to 1, making it a direct mapped cache and run the program, saving the output to a .txt file.

The Lab is due on Sunday, November 30 at 6PM. This program must be a .c program. .cpp is not acceptable. And it must compile correctly on SEAS Lnux. Deliverables are: your updated cachelab.c (with E=1 or 4, noproblem) and the two .txt files.

REMEMBER! Your program must compile correctly on SEAS Linux.