CS-UY 2214 — Homework 8

Jeff Epstein

Introduction

Unless otherwise specified, put your answers in a plain text file named `hw8.txt`. Number each answer. Submit your work on Gradescope.

You may consult the E20 manual, which is available on Brightspace.

Problems

1. Consider the following data structure in E20 assembly language:

```
array_size:
    .fill 10
array:
    .fill 5
    .fill 1
    .fill 4
    .fill 2
    .fill 9
    .fill 6
    .fill 5
    .fill 8
    .fill 0
    .fill 7
```

The data structure describes an array of unsigned numbers, which begins at address `array`. In addition, the value at address `array_size` gives the length of the array.

Write an E20 assembly language program implementing the bubble sort algorithm. Your program will sort in place an array of arbitrary size. That is, after running your program, the values stored in sequence in memory, starting at address `array`, will be as follows:

```
0000 0001 0002 0004 0005 0005 0006 0007 0008 0009
```

The algorithm in pseudocode (via Wikipedia):

```
procedure bubbleSort(A : list of sortable items)
    n := length(A)
    repeat
        swapped := false
        for i := 1 to n-1 inclusive do
            /* if this pair is out of order */
            if A[i-1] > A[i] then
                /* swap them and remember something changed */
                swap(A[i-1], A[i])
```

```
                    swapped := true
                end if
            end for
        until not swapped
    end procedure
```

Your program must work not only with the above array, but with any array of numbers, of any size (> 0), that can be stored in E20 memory.

Your solution must be thoroughly commented, or else it will not be graded. Make sure that the comments help the grader understand the intent of your code. Test your solution in your simulator.

Put your answer (including the provided data structure) in a file named `bubble.s`.

2. Consider a virtual memory system with 40-bit virtual addresses, 16KB pages, and 36-bit physical addresses. Each memory cell is one byte.

   What is the total size (in bytes) of the page table for each process? Include the protect, dirty, and valid bits for each page table entry. Assume that all virtual pages are used.

3. Consider a virtual memory system with 32-byte pages, 8-bit virtual addresses, and 8-bit physical addresses. The TLB is fully-associative, has 4 blocks, and uses LRU replacement. Each memory cell is one byte.

   Below, we show the initial state of the page table and TLB. Note that the Valid bit in the TLB refers to the validity of that TLB entry, not to the validity of the underlying page table entry: invalid pages are not cached.

Page table

Virtual page	V	D	P	Physical page
0	1	0	1	0
1	1	0	0	4
2	0	0	0	3
3	1	1	0	4
4	1	1	0	1
5	1	0	0	1
6	1	0	0	2
7	0	0	0	5

TLB

V	P	Tag	Value
1	0	3	4
0			
0			
0			

Starting from this initial state, determine what will happen for each of the following memory operations. For each memory operation, indicate if it causes a hit or a miss on the TLB. Then, indicate if it causes a page fault. If it doesn't cause a page fault, indicate which physical address will be accessed. If the access causes a protection fault, say so.

```
WRITE to virtual address 53
READ from virtual address 32
```

```
WRITE to virtual address 5
READ from virtual address 98
READ from virtual address 64
READ from virtual address 80
WRITE to virtual address 100
WRITE to virtual address 35
```

4. Recall the principle of *spatial locality*, which states that memory locations that are nearby to each other are likely to be cached together.

   Consider the following C++ code:

   ```
   int a[2048] = {...};
   int b[2048] = {...};
   int c[2048];

   for (i=0; i<2048; i++)       /* Make this
       c[i] = a[i] + b[i];         part faster */
   ```

   Re-write the code to take advantage of cache effects relating to spatial locality. The goal of your revised code is to optimize the performance of the marked lines. The final effect of your revised code must be identical to the original. You may assume that any data structure that you create is initialized appropriately, as long as you state those assumptions.

   Hint: you will have to restructure the data. You may want to use a `struct`.

   I should point out that the following is NOT a valid solution, as it does not improve the spatial locality of the addends:

   ```
   struct array {
         int a[2048] = {...};
         int b[2048] = {...};
         int c[2048];
   };
   ```