CS221 – Lab11

Paul Haskell

INTRODUCTION

This lab will have you perform an important systems programming task: benchmarking the performance of key systems components. You will measure the speed of your computer memory and the performance of your computer's system clock. These are tasks that a professional system programmer would perform to evaluate a new hardware platform.

This lab also will give you experience writing C in an object-oriented style:

- define one or more structs that store important related data (the "class variables")
- write methods that take a pointer to a struct object as the first argument (the "class methods")

You will write code for a "first-in, first-out queue" that stores `unsigned long long` numbers. You will need this queue in a later lab.

But first, a quick in-class lab…

Endianness

For the in-class portion of today's lab, please write a program called **swapends.c** . The program repeatedly reads integers from the keyboard, swaps the byte order of the bytes in each integer, and prints the result. The integers are entered and printed in decimal, but to make debugging easier, you can debug first by printing results in hex. (**Undo that** before pushing to GitHub!) For example, if a test input is

168496141

then the output should be

218893066

(In hex, 168496141 is 0x0a0b0c0d and 218893066 is 0x0d0c0b0a.)

Your program should run until the keyboard input ends (ctrl-D or ctrl-Z).

Oh yeah, how to do the swapping? Use your bit operations! |, &, ^, <<, >>.

Memory benchmarking

Your next program for this lab, **memspeed.c** , will measure the memory access speed on your computer.

- Step #1 is to figure out approximately how much memory your computer program can allocate successfully. Start by allocating a small buffer, and if successful, freeing it. Then double the size

of the buffer you attempt to allocate. Keep repeating this process until your `malloc()` fails. Now you should be able to figure out approximately the largest buffer a program can allocate.

- Now, allocate a buffer approximately ¾ of the maximum possible size.
- Make an array of 1,000,000 `char*`'s. The values of the `char*`'s should be randomly and evenly distributed across your buffer.
- Use the `clock()` method to see how much time in microseconds is required to set each of your 1,000,000 random locations to `(char) 1`.
- Fun! You made a measure of memory performance for your computer.

Now do the whole thing again, but instead of allocating a really big buffer, allocate a buffer whose size is only 1000 entries. Your array of 1,000,000 random `char*`s will have a lot of entries that point to the same location in your first buffer. How long does it now take to set each value to 1?

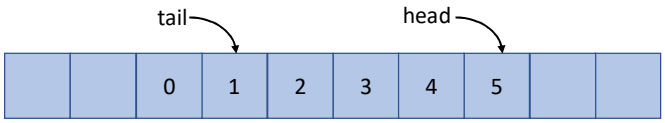
Write your results to a text file called **memspeed.txt** , with a format like below (but using your actual numbers):

```
big: 12
small: 6
```

Why are the numbers so different? Not all memory is created equal inside a computer. Inside the CPU itself, there is a small amount of memory built in called the "cache". This memory is very fast, but there is not very much of it. Typically, on the same chip package as the CPU but maybe not inside the CPU itself there will be a "secondary cache" which is larger than the "primary cache" but not quite as fast. The bulk of the computer's memory is in separate chip packages, and this memory is even slower. Computers automatically manage what information is stored in the caches, to maximize speed.

First-in, first-out queue

This queue will be an object into which you `push()` numerical values and later on, you can `pop()` them out. You may push several values in a row, then pop a few, rather than always having `push()` and `pop()` perfectly alternating. Your queue will store a number of values to account for this.



This diagram shows a queue where the user has pushed the values 0, 1, 2, …, 5 and has popped the value 0.

You can build your queue on top of an ordinary array. When your `head` and `tail` indices reach the end of the array, they should "wrap around" back to the beginning.

Here is (mostly) the code you will need for your structure:

```
#define LEN (64)
struct FifoUll {
    unsigned long long data[LEN]; // the data you store

    int head; // point to just-pushed data element
    int tail; // point to about-to-be-popped element
    int fullness; // how many have been pushed but not yet popped
};
```

It is your job to write the methods! In **fifo.h** and **fifo.c** please write at least the following:

- `init(&FifoULL); // initialize variables in the Fifo`
- `push(&FifoULL, unsigned long long); // add a value to the Fifo`
- `pop(&FifoULL); // remove and return a value from the Fifo`
- `isEmpty(&FifoULL); // is the Fifo empty?`
- `isFull(&FifoULL); // is the Fifo full?`

You can add any other variables and methods you desire.

`push()` should check `isFull()` to report an error and exit if anyone tries to `push()` to a full Fifo.

`pop()` should check `isEmpty()` to report an error and exit if anyone tries to `pop()` an empty Fifo.

Please write a **runfifo.c** program that does the following:

- create and initialize a Fifo
- in a loop, 20000 times, at random, either `push()` a number or `pop()` a number. (Don't push if the fifo is full and don't pop if the fifo is empty.)
- The numbers pushed to the file should count up by 1 every time
- Write to a file all the numbers you pop. (Should be about 20000/2 = 10000 numbers.)

Finally, you should write a **testfifo.c** program that reads a file whose name is given as the first command line argument (use the file from **runfifo.c)** and makes sure that all the numbers you pushed to the Fifo were output in the correct order. **testfifo.c** should print either "PASS" or "FAIL". (Well, keep debugging until it prints "PASS"!)

CONCLUSION:

Please push the programs and text files to GitHub before the lab deadline. The first program is due at the end of class. The deadline for completion of the rest of Lab09 is 11:59pm Thursday January 25.

Task	Score, points
swapends.c compiles, runs, and gives correct results for 5 test inputs	10
memspeed.c compiles, runs, and gives reasonable output values	10
memspeed.txt is present, has the correct format, and contains reasonable values	5

fifo.h and **fifo.c** compile and work correctly	20
runfifo.c compiles and does what is requested	15
testfifo.c compiles and does what is requested	10
Software and design quality as judged by the grader	10