Lab 9: Buffered and non-buffered I/O

Overview

The code and data for a running application reside in main memory. When a running program reads or writes data to an external destination (e.g., a file on disk or over a network connection to another computer), the application has to invoke an operating system function. We use the term **system call** to refer to such a function. Although the code for a system call resembles a normal function call – for example, write(fd,buffer,length) invokes a system call – a system call requires the processor to change state and change the memory addressing map. Therefore, a system call is more time consuming, or expensive, than a normal function call. Because of the time expense of making a system call, most programs use buffering to reduce the number of system calls made and thereby increase I/O performance.

In the first part of this lab, you will create functions that implement buffered and non-buffered write operations and you will measure I/O performance using each of these operations. To measure unbuffered output, you will create a function mywritec() that takes a character as an argument and uses the system call write() to output the character to standard output (file descriptor 1). That is, on each call, you will write one character. To measure buffered I/O, you will write function myputc(), which takes a character as an argument. Function myputc() also writes characters to standard output, but uses buffering. That is, myputc() places outgoing characters into a buffer until the buffer is full. When placing a character into the buffer makes the buffer full, then myputc() calls write() to write the entire buffer contents with this one system call. You will also write mywritebufsetup() to initialize a buffer and mywriteflush() to force writing the buffer, again using write(), any time this is desired. To compare the non-buffered and buffered approaches, you will write a large number of bytes both ways, and measure the time required.

In the second part of the lab, you will implement an unbuffered input function, myreadc(), and a buffered input function, mygetc(). Each of them returns one byte from a file. You will also write myreadbufsetup() to initialize a buffer. You will compare the time required to read a large file using mygetc() with the time required to read the same file using myreadc().

We will supply a main.c that calls your functions to write a large number of bytes to standard output to test write operations and read a large file to test your read operations. Our main uses both buffered and non-buffered methods, and reports the time the functions take for input and output operations. Please copy this file https://drive.google.com/file/d/0BwL483kmcl5-MldfVXBGdzYxVEE/view to the directory where main.c exists. The file will be read by myreadc() and mygetc() functions.

Details

Below are descriptions of the C functions that you need to write. It is recommended to check out the resources at the end of this page before starting to write code.

1. Write without buffering

```
void mywritec(char ch)
{
   /* Use the system call write() to write char 'ch' to standard output (file descriptor 1) */
}
```

2. Write with buffering

```
/* Use the C preprocessor to define constant MAX_BUFFER to be 1048576. */
/* The constant will be the upper-bound on buffer size in this project. */
/* Declare a global array of MAX_BUFFER characters named "write_buf" that */
/*     will serve as an output buffer .*/
/* Declare a global character pointer, wp,  that will point to a location in the buffer */
/* Declare a global integer, write_buf_size, that stores the size of the output buffer */
/*     being used at the current time. */

void mywritebufsetup(int n)
{
   /* Verify that n is a positive integer less than or equal to MAX_BUFFER, and */
   /* store n in global variable write_buf_size. */
   /* Initialize wp to point to the first byte of buffer. */
}

void myputc(char ch)
{
   /* Place character ch in the location given by wp, and increment wp. */
   /* If the buffer is full (contains write_buf_size characters), write the entire buffer */
   /* to standard output using the write() system call and reset wp to the first location */
   /* of the buffer.  Note that myputc() will be called multiple times before the buffer */
   /* is completely written out. */
}

void mywriteflush(void)
{
   /* Note: this function will be called after all calls to myputc() have been made */
   /* if any characters remain in the write buffer, write them to standard output */
}
```

3. Read without buffering

```
int myreadc(void)
{
```

```
   /* Use read() system call to read a character from text file(file descriptor is 'fd_read') */
   /* if read() indicates end-of-file, return -1 to the caller. Otherwise, return the */
   /* character that was read in the low-order byte of the integer (be careful to avoid 8?
   /* sign extension). */
}
```

4. Read with buffer

```
/* Declare a global array of MAX_BUFFER characters named "read_buf" that will */
/*     serve as an input buffer. */
/* Declare a global character pointer, rp,  that will point to a location in the buffer. */
/* Declare a global integer, read_buf_size, that stores the size of the input buffer */
/*     being used at the current time. */
/* Declare a global integer, read_count, that tells how many bytes were read. */

void myreadbufsetup(int n)
{
   /* Verify that n is a positive integer less than or equal to MAX_BUFFER, */
   /*     and store n in global variable read_buf_size. */
   /* Set read_count to zero. */
}

int mygetc()
{
   /* If read_count is less than or equal to zero, call read() to read up to */
   /* read_buf_size bytes into read_buf from text file (file descriptor is 'fd_read'),
   /* and set read_count to the number of bytes actually read. Set rp to the */
   /* first location in the buffer.  If the read_count is zero (the read call returned */
   /* end-of-file), return -1 to the caller to indicate end-of-file. Extract the next  */
   /* character from the buffer, increment rp, and decrement read_count. */
   /* Return the character extracted from the buffer in the low-order byte of an */
   /* integer (be careful to avoid sign extension). */
   /* Note that this function will be called multiple times before the buffer is emptied. */
}
```

Compiling and Running the program

Copy and paste the above global declarations and functions into the file main.c
Note: we supply function prototypes, which means your functions may be added to the end of the file.
Use these commands to compile and run the code:

```
gcc -o main main.c'
./main
```

Part 1 (15 pts)

Write without buffer. Show the code to the TA. Execute the program and make a note of the time to write:

1 - Write without buffering
2 - Write with buffering
3 - Read without buffering
4 - Read with buffering
5 - Exit
Enter the option: 1
Writing byte by byte
Writing byte by byte
Writing byte by byte
.
.
Writing byte by byte

Time to write without buffering: 5.340000 secs

Part 2 (30 pts)

Write with buffer. Show the code to the TA. Execute the program and make a note of the time to write:

1 - Write without buffering
2 - Write with buffering
3 - Read without buffering
4 - Read with buffering
5 - Exit
Enter the option: 2

Enter the buffer size in bytes: 1024
Writing using buffers
Writing using buffers
Writing using buffers
.
.
Writing using buffers

Time to write with buffering: 0.290000 secs

Part 3 (15 pts)

Read without buffer. Show the code to the TA. Insert print statement in main function to print out the characters read. Execute the program to verify that read operation was performed correctly. Re-run again without the print statement to measure just the time for read:

1 - Write without buffering
2 - Write with buffering
3 - Read without buffering
4 - Read with buffering
5 - Exit
Enter the option: 3

Time to read without buffering: 1.980000 secs

Part 4 (30 pts)

Read with buffer. Show the code to the TA. Insert print statement in main function to print out the characters read. Execute the program to verify that read operation was performed correctly. Re-run again without the print statement to measure just the time for reading:

1 - Write without buffering
2 - Write with buffering
3 - Read without buffering
4 - Read with buffering
5 - Exit
Enter the option: 4

Enter the buffer size in bytes: 1024

Time to read with buffering: 0.110000 secs

Part 5 (10 pts)

Fill in the below table, plot graphs of Write Throughput vs Buffer size and Read Throughput vs Buffer Size (Use MS Excel or any graphing/plotting tool) and explain the reason for time differences:

Buffer size (bytes)	Time to write (sec)	Write throughput (bytes/sec)	Time to read (sec)	Read throughput (bytes/sec)
1				
32				
256				
1024 (1 KB)				
8192 (8 KB)				
65536 (64 KB)				

Turnin

Note: You are responsible for getting your lab graded by your instructor during lab and also for using the turnin command specified below; before the deadline.

Follow these instructions to turnin lab9:

Make sure that your program is built using gcc.

Create a folder lab9-src which contains main.c

Navigate to the parent directory of lab9-src.

Then type:

```
$ turnin -c cs250 -p lab9 lab9-src
$ turnin -c cs250 -p lab9 -v
```

The second command verifies that you have successfully submitted your files. If you forget to add -v it will erase your previous submission.

Resources

http://gd.tuwien.ac.at/languages/c/programming-bbrown/c_075.htm
http://en.wikipedia.org/wiki/File_descriptor