CS220 - Computer System II
Lab 9

Due: 10/26/2017, 11:59pm

1 Introduction

In this lab, you will examine the I/O latencies between hard disk (HDD) and main memory. You will also examine the performance impact of sequential versus non-sequential memory access. Because you are in the latter half of the semester, we assume that you know how to fill in the gaps (e.g., filling missing declarations for variables that are used in the code fragments below). If you are unsure about anything, ask the TA.

2 Getting Started

Create a folder **Lab9** and a file **mem_access.c** within Lab9. Copy the functions `timeval_subtract` and `timeval_print` from previous Lab 7 along with corresponding header file inclusions into **mem_access.c**.

3 Implementation

1. In order to examine file I/O, we will first create a large file with random data. For the purpose of this lab, a 64 MB file will suffice.

```
1  $ head -c 64M < /dev/urandom > file64M
```

   This command will read 64MB of random data from /dev/urandom and store it into a file called file64M. **IMPORTANT: Do not forget to delete file64M AFTER completion of the lab.**

2. Now, within mem_access.c, create a `main` function and a `test_hdd` function. File I/O will go in `test_hdd`. In the main function, you will call `test_hdd`, which will open file64M and read 64MB from the file. So far, you have used `fopen()`, `fclose()`, `fread()`, `fscanf()`, etc. functions to perform file I/O. This time, you will use a set of low-level functions to perform I/O. Specifically, you will use `open()`, `read()` and `close()`. These functions are similar in functionality to fopen(), fclose(), etc., but with key differences (`http://stackoverflow.com/questions/1658476/c-fopen-vs-open`). Because we intend to measure the hardware performance, we will use low-level functions.

```
#include ...

#define STEP_SIZE  ....
#define READ_SIZE  ...

void test_hdd() {
...
  fd = open("file64M", O_RDONLY); /* O_RDONLY implies read-only mode */
  /* Start timer here */
  for(i = 0; i < READ_SIZE; i+=STEP_SIZE)
    read(fd, &c[0], STEP_SIZE);
  /* Stop timer here */
  /* Calculate time taken here */
  timeval_print("HDD ACCESS: ", &tvDiff);
...
}

int main() {
  test_hdd();
  return 0;
}
```

Integer `fd` is an file descriptor that represents the open file. Refer to the man page for more information on open. The function read is a corresponding function to read from a file descriptor. In the above code, `read` reads from the file whose descriptor is `fd`. It reads `STEP_SIZE` bytes into `c`. You are to declare `fd` as an integer and `c` as an array of at least `STEP_SIZE` bytes. Thus, you should also make a preprocessor definition `STEP_SIZE` to be a fixed numeric value. You will start with a value of 4 and increase in subsequent runs. Also, make a preprocessor definition `READ_SIZE` to be 64 MB (i.e., 64*1024*1024 bytes). `O_RDONLY` is defined in **fcntl.h**. You will observe that the time taken significantly reduces as the `STEP_SIZE` increases. This is due to fewer number of calls to read in order to read `READ_SIZE`. Each call to read must make a system call that transfers control to the kernel and back, which is expensive.

3. Compile and run **mem_access.c** to generate an executable **mem_access**. Create a file **Lab9.txt** in Lab9 and record the disk access times for `STEP_SIZE` values of 4, 32, 128 and 1024.

4. Next, you will create a function called `test_mem()` to test main memory. You are

working on a 64bit system (remote.cs.binghamton.edu). So, you will create a large amount of space (greater than the addressable 4GB on 32 bit systems) on the heap and perform sequential byte-wise I/O on the memory. Memory is divided into contiguous blocks called pages. Each page in the virtual memory maps to a page in the physical memory. First, let us identify the size of each page on the system you are working on.

```
$ getconf PAGESIZE
```

Record the output. You will use it in the next step. Next, implement the `test_mem` function as shown below.

```
...
#define PAGE_SIZE       /* Output of $ getconf PAGESIZE */
#define NG               /* Number of Gigabytes */
#define PAGES_IN_NG      (NG*1024*1024)/PAGE_SIZE
...

void test_mem()
{
  ...
  unsigned char *mem_arr[PAGES_IN_NG];
  ...
  /* Allocate N GB on the heap */
  for(i = 0; i < PAGES_IN_NG; i++) {
    if((mem_arr[i] = memalign(PAGE_SIZE, PAGE_SIZE)) == NULL) {
      printf("Malloc failed...\n");
    }
  }

  /* Start timer */
  for(i = 0; i < PAGES_IN_NG; i++)
    for(j = 0; j < PAGE_SIZE; j++)
      mem_arr[i][j]++;
  /* End timer */
  /* Calculate time taken */
  timeval_print("MEM SERIAL ACCESS: ", &tvDiff);
  ...
}
```

A couple of points to note:

- The function `memalign` is similar to `malloc` except that it returns memory that is guaranteed to start from a particular alignment. In this case, it returns a page that is aligned to a page boundary (i.e., address of the created memory is a multiple of `PAGE_SIZE`). Therefore, each pointer in `mem_arr` points to a page. You will need to include `malloc.h` for `memalign`.
- The statement `mem_arr[i][j]++` reads and writes to each byte in each page *sequentially*. That is, it starts from byte 0 of the first page (page pointed to by `mem_arr[0]`), updates each byte up to the end of the page, then moves on to the second page (page pointed to by `mem_arr[1]`), and so on.

5. Next, we will access the pages pointed to by `mem_arr`, but in steps of `PAGE_SIZE`.

```
void test_mem()
{
...
/*** CODE TO PERFORM SEQUENTIAL I/O ****/
...
/* Start timer */
for(j = 0; j < PAGE_SIZE; j++)
  for(i = 0; i < PAGES_IN_NG; i++)
    mem_arr[i][j]++;
/* End timer */
/* Calculate time taken */
timeval_print("MEM PAGE STEP ACCESS: ", &tvDiff);
...
}
```

Notice how the loop variables have been switched between the inner and outer for loops. Consequently, the memory accesses will be non-sequential steps of `PAGE_SIZE`: (page0,byte0), (page1,byte0), (page2,byte0),...,(page4096,byte0), (page0,byte1)...

6. Within **Lab9.txt**, record serial and page step access times for NG values of 1, 4, 32 and 128. Also, for each value of `NG`, use the following command to run the program mem_access.

```
$ perf stat ./mem_access
```

The perf program is a profiler program that prints interesting statistics about a program's execution. We are particularly interested in page faults. This tells us the number of times a page was not found in the memory. Therefore the OS had to (potentially) flush something out and bring the requested page into the memory from the hard disk. Within **Lab9.txt**, record the number of page faults for NG values of 1, 4, 32 and 128.

4 Submitting the result

Remove binaries and intermediate files from Lab9. Create a tar.gz of Lab9 folder with only **mem_access.c** and **Lab9.txt**.:

```
$ tar -cvzf lab9\_submission.tar.gz ./Lab9
```

Submit lab9_submission.tar.gz to Blackboard.

Finally, DO NOT FORGET to delete file64M.

```
$ rm -f file64M
```