Project 1 Report

w1049

Contents

1 Copy **1**
1.1 Description . . . 1
1.2 Usage . . . 1
1.3 Interesting code . . . 1
1.3.1 Error handling . . . 1
1.3.2 Timer . . . 2
1.4 Test and analysis . . . 3

2 shell **3**
2.1 Description . . . 3
2.2 Usage . . . 3
2.3 Interesting code . . . 3
2.3.1 Wrapped fork . . . 3
2.3.2 Multi-clients . . . 4
2.3.3 Parse commands . . . 4
2.3.4 Execute commands . . . 5

3 matrix **6**
3.1 Description . . . 6
3.2 Usage . . . 6
3.3 Interesting code . . . 6
3.3.1 Matrix partition . . . 6
3.3.2 Multi-thread . . . 6
3.4 Test and analysis . . . 6

1 Copy

1.1 Description

Copy file from one to another using two processes that communicate with `pipe` system call.

1.2 Usage

```
./Copy <InputFile> <OutputFile> [BufferSize]
```

This command will copy the InputFile to OutputFile, using a bufffer of BufferSize. If the BufferSize is not given, the buffer will use a default value of 131072.

1.3 Interesting code

1.3.1 Error handling

`err()` is a convenient function defined in `err.h` , which prints `errno` and exits. `ERROR` is a string macro [ERROR] used as prefix. Code below is using to open the src and dest files.

```c
    #define ERROR "\033[31m[Error]\033[0m "

    FILE *src, *dest;
    src = fopen(argv[1], "r");
    if (!src) err(1, ERROR "%s", argv[1]);

    dest = fopen(argv[2], "w");
    if (!dest) err(1, ERROR "%s", argv[2]);
```

`errx()` is another printing then exiting function, while it doesn't print the `errno`. Code below is using to set the buffer size.

```c
    int bufsize = argc >= 4 ? atoi(argv[3]) : 131072;
    if (bufsize <= 0) errx(1, ERROR "Invalid buffer size: %s -> %d", argv[3], bufsize);
    char *buf = malloc(bufsize);
    // ...
    free(buf);
```

Different from functions that return `-1` on error, `fread()` and `fwrite()` need `ferror()` to check if there is an error.

```c
    int readbytes;
    while ((readbytes = read(pipefd[0], buf, bufsize)) > 0) {
        fwrite(buf, readbytes, 1, dest);
        if (ferror(dest)) err(1, ERROR "Writing file");
    }
    if (readbytes == -1) err(1, ERROR "Reading pipe");
```

1.3.2 Timer

Three different functions can be used to get time.

- `clock()` returns the processor time used by the program.
- `gettimeofday()` returns system time accurate to microsecond.
- `clock_gettime()` with argument `CLOCK_MONOTONIC` returns system time accurate to nanosecond.

```c
#ifdef TIME1
    clock_t start, end;
    start = clock();
    // ...
    end = clock();
    elapsed = (end - start) * 1000.0 / CLOCKS_PER_SEC;
#elif TIME2
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC_COARSE, &start);
    // ...
    clock_gettime(CLOCK_MONOTONIC_COARSE, &end);
    elapsed = (end.tv_sec - start.tv_sec) * 1000 +
        (double)(end.tv_nsec - start.tv_nsec) / 1000000;
#else
    struct timeval start, end;
    gettimeofday(&start, NULL);
    // ...
    gettimeofday(&end, NULL);
    elapsed = (end.tv_sec - start.tv_sec) * 1000 +
        (double)(end.tv_usec - start.tv_usec) / 1000;
#endif
```

1.4 Test and analysis

Generate large files of different sizes and use different buffer sizes to test performance. The test will be run multiple times, and we calculate the average time consumption then the copying speed (ms/MiB). I ran the test on my VMware(AMD Ryzen 7 5800H) and Siyuan-1 cluster(Intel(R) Xeon(R) Platinum 8358). The result is shown in the Figure 1.

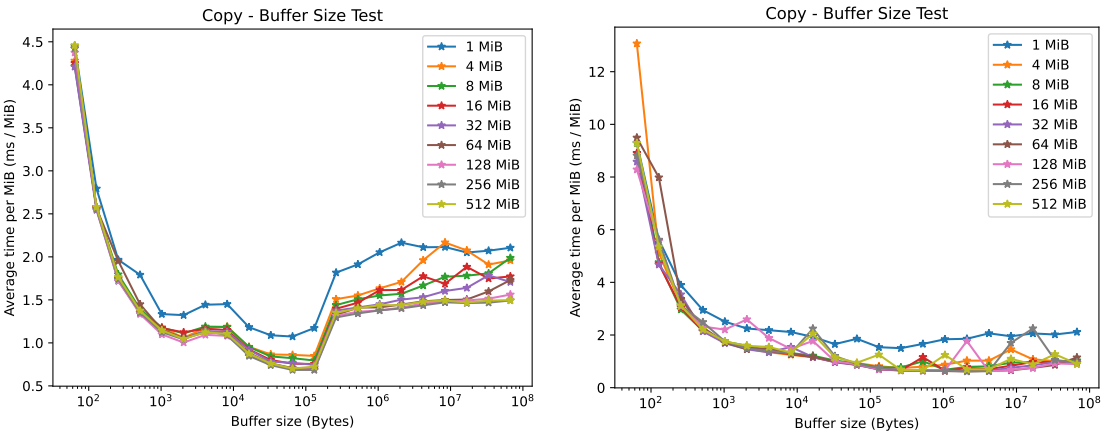

Figure 1: First test on VMware(left) and Siyuan-1 cluster(right)

On my VMware, we can see that as the buffer size increases, the time consumption generally decreases, but increases from 131072 Bytes (128 KiB). However, on Siyuan-1 cluster, the time consumption is almost always decreasing.

Obviously, the performance of the two machines is different. What causes this result? It could be cache, because CPU L1 cache on my VMware is 256 KiB. When the buffer size is set to 128 KiB or less, the cache can hold a buffer, plus `fread()` / `fwrite()` buffer and other things; when the buffer size is larger than 256 KiB, the cache can only hold (or even cannot hold) the buffer itself at most, which is not enough and will cause many cache misses. But I don't know why the curve on Siyuan-1 cluster is so smooth, probably because its cache and memory are very fast.

Finally, I choose 131072 Bytes (128KiB) as the buffer size.

2 shell

2.1 Description

A shell-like server, which can be connected by many clients, handling the commands with arguments and the commands connected by pipes.

2.2 Usage

```
./shell <Port>
```

This command will start a server, then you can use `telnet` to coneect to it and send commands.

2.3 Interesting code

Some codes are inspired by xv6 shell and *Computer System: A Programmer's Perspective*.

2.3.1 Wrapped fork

Checking the return value of `fork()` every time is a pain, so a wrapper function is handy.

```c
pid_t fork1() {
    pid_t pid = fork();
    if (pid == -1) err(1, ERROR "Failed to fork");
    return pid;
}
```

2.3.2 Multi-clients

There are many ways to handle multi-clients (concurrent programming), such as using I/O multiplexing based `select()` or `epoll()`, or just using processes based `fork()` to create new process for each client. I choose the last one, because it's easy to implement and the performance is good enough for our lab. The code is shown below, in which `serve()` is the entry function to provide the service, and it returns only when the client disconnects.

```c
    while (1) {
        // accept()
        if (fork1() == 0) {
            close(sockfd);
            serve(client_sockfd);
            close(client_sockfd);
            exit(0);
        }
        close(client_sockfd);
    }
    while (wait(NULL));
```

Notice that when a client use `exit` command, the child process will exit. However, the parent process is in the `while` loop, so it cannot use `wait` and the child process will become a zombie process. To avoid this, we need to add a signal handler to handle the `SIGCHLD` signal, which is sent when a child process exits.

```c
void sigchlc_handler(int sig) {
    while (waitpid(-1, NULL, WNOHANG) > 0);
}
int main() {
    // ...
    signal(SIGCHLD, sigchlc_handler);
    // ...
}
```

2.3.3 Parse commands

Our client is based on `telnet`, which will send the command line by line (on Linux), with CRLF at the end. That's different from pure Linux, which has only LF.

```c
while ((n = read(clientfd, buf, sizeof(buf))) > 0) {
      buf[n] = 0;
      while (n > 0 && (buf[n - 1] == '\r' || buf[n - 1] == '\n'))
          buf[--n] = 0;
      int argc = parse(buf, commands); // use strtok()
      // ...
}
```

2.3.4 Execute commands

The shell (or exacly the `serve()` function) will fork a child process to execute the command, while the parent process waiting for next command. However, there are some special cases, such as `cd` and `exit`, which should be handled by the parent process. If the child process changes the current directory, the parent process cannot see the change. This is obviously not what we want.

```
    if (argc == 0) { /* do nothing */ }
    else if (argc == -1) { /* error: too many arguments */ }
    else if (strcmp(commands[0], "cd") == 0) { /* handle cd */ }
    else if (strcmp(commands[0], "exit") == 0) { return; }
    else { // other command
        if (fork1() == 0) {
            // redirect all stdio to clientfd
            close(0), close(1), close(2);
            dup(clientfd), dup(clientfd), dup(clientfd);
            close(clientfd);

            run(commands);
        }
        wait(NULL);
    }
```

The `run()` function is used to execute the command, and it never returns. It use recursion to handle the commands connected by pipes. So every time it is called, it will just find the first pipe "|", and divide the commmands into two parts.

```
void run(char *commands[]) {
    for (int i = 0; commands[i]; i++)
        if (strcmp(commands[i], "|") == 0) {
            // if a pipe is found
            if (i == 0) // error: first |
                errx(1, "Pipe at the beginning");
            else if (commands[i + 1] == NULL)  // error: last |
                errx(1, "Pipe at the end");

            int pipefd[2];
            if (pipe(pipefd) == -1) err(1, ERROR "Failed to pipe");

            if (fork1() == 0) {
                close(1);
                dup(pipefd[1]);
                close(pipefd[0]), close(pipefd[1]);
                commands[i] = NULL;
                run(commands);
            }
            if (fork1() == 0) {
                close(0);
                dup(pipefd[0]);
                close(pipefd[0]), close(pipefd[1]);
                run(commands + i + 1);
            }
            close(pipefd[0]), close(pipefd[1]);
            wait(NULL), wait(NULL);
            exit(0);
        }
    if (execvp(commands[0], commands) == -1) // commands without pipe
        err(1, "Failed to execvp '%s'", commands[0]);
}
```

3 matrix

3.1 Description

Do matrix multiplication using matrix partition based algorithm with time complexity $O(n^3)$.

3.2 Usage

```
./single
./multi # When no command line argument given, the program automatically read input
        # from “data.in”, and output to “data.out”.
./multi <Size>
```

3.3 Interesting code

Some codes are inspired by *Computer Organization and Design: The Hardware/Software Interface*.

3.3.1 Matrix partition

The input matrix is partitioned into 32×32 (or smaller when matrix size is too small) sub-matrices, to make use of cache.

```
void do_block(int n, int *A, int *B, int *C, int si, int sj, int sk) {
    for (int i = si; i < si + bsize; i++)
        for (int j = sj; j < sj + bsize; j++) {
            int cij = C[i * n + j];
            for (int k = sk; k < sk + bsize; k++)
                cij += A[i * n + k] * B[k * n + j];
            C[i * n + j] = cij;
        }
}
```

3.3.2 Multi-thread

Once the partition is done, we can divide the sub-matrices roughly equally among each thread and let them compute in parallel.

```
void *worker(void *arg_v) {
    struct data *arg = (struct data *)arg_v;
    for (int sj = arg->begin; sj < arg->end; sj++) {
        for (int si = 0; si < n; si += bsize)
            for (int sk = 0; sk < n; sk += bsize)
                do_block(n, A, B, C, si, sj * bsize, sk);
    }
    pthread_exit(0);
}
```

3.4 Test and analysis

It's easy to test `multi` because it can generate random matrix. I use a macro to decide whether to time it or not. My VMware has 8 cores, and the Siyuan-1 cluster has only 1 core (I choose it because 1 core queuing is faster). I divided the consumption time by n^3, so that I can roughly calculate the coefficient in the time complexity. The result is shown in Figure 2.

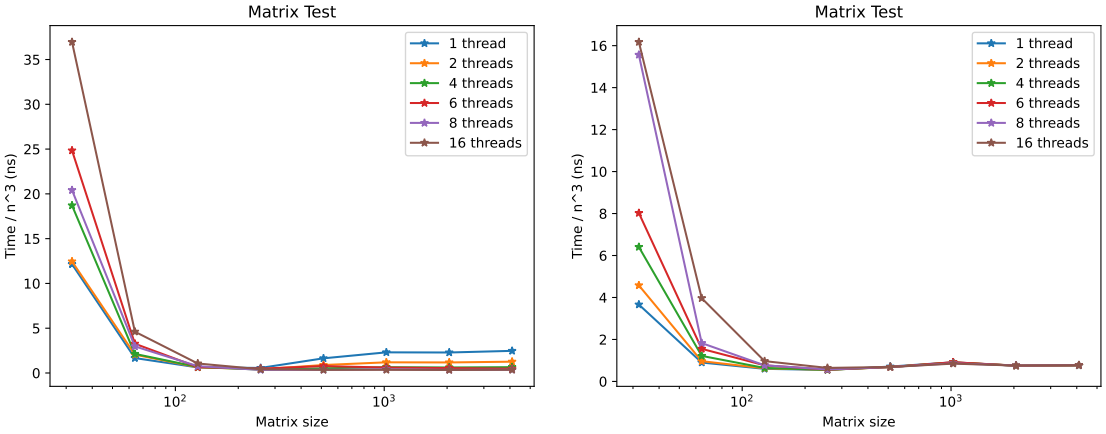

Figure 2: Matrix test on VMware(left) and Siyuan-1 cluster(right)

When the matrix size is small, we can see the performance is not good, and the more threads, the slower the program is. This is because when the matrix is small, the overhead of creating threads is larger than the benefit of parallel computing.

In fact, partition also has a cost, so if the matrix is too small, "1 thread" in the figure will also be slower than the simplest matrix multiplication.

In the left figure, there is a difference in performance between the number of threads when the matrix is relatively large, but it is not very clear. So I plotted the part of the size ≥ 128 separately, as in the figure 3.

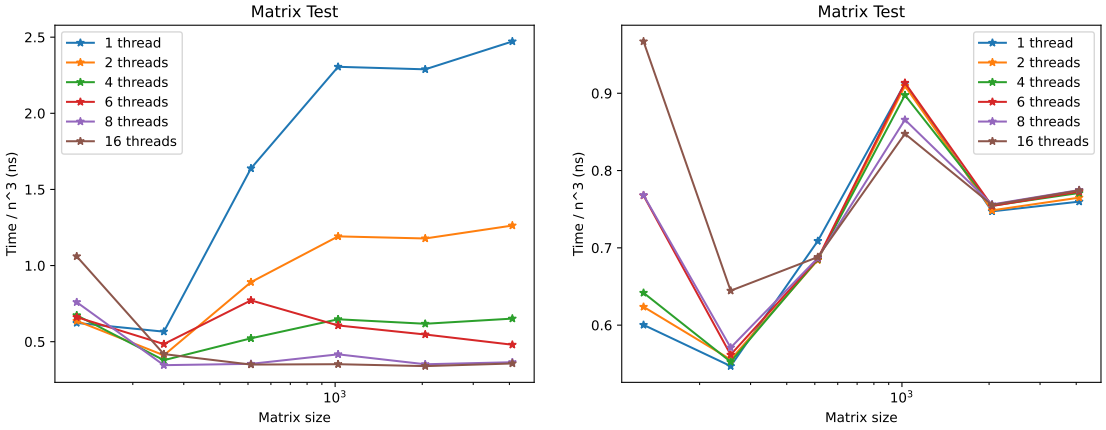

Figure 3: Matrix test (only size ≥ 128) on VMware(left) and Siyuan-1 cluster(right)

It is easy to see that on my VMware, when the number of threads is between 1 and 8, the more threads the faster it is. 2 threads are almost twice as fast as 1 thread, but 16 threads and 8 threads are almost the same. However on Siyuan-1 cluster, the more threads, the slower it is.

This is caused by the overhead of thread context switching. A 8-core (1 thread per core) CPU allows 8 threads to run in parallel, but if there are more than 8 threads, the CPU will have to switch between them, which will cause performance degradation. The same is true for a 1-core CPU, which can only allow one thread to run at a time. Multi-threading is not suitable for this situation.

In practice, it is better not to use more than the number of cores in a multi-threaded program. For example, on my VMWare, I should choose less than or equal to 8 threads.