City University of Hong Kong

CS3103 Project Report

by

Name:

*Wan chi yan 56696743 EE/CDE

Fung King Shun 56659723 CS

Zhao Zhi 56742322 EE/CDE

24/11/2021

CONTECT

1.Design

1.1. How to access the input file efficiently

The concept of input file will be using (int argc, char** argv) in the main function. For the (int argc) is to store how many files we have entered. And (char** argv) is the array to store the path of each file. E.g.

Input: $ ./pcount 2-0.in 2-1.in

argv[0]: 2-0.in
argv[1]: 2-1.in

argc: 2

Next, we will use a function to check the file if it exists. But this way to read files is not a good design. Because this method is to read 2-0.in first and then read 2-1.in, it will waste time and inefficiency. When reading files, most of the CPU time is spent waiting for the disk data to be read. During this period, the CPU is almost idle. Therefore we need multiple threads to help us. The solution is we try to use the value of argc to determine how many threads need to be created. For each file create a new thread job, complete the read file action by yourself. In this way, the performance of the CPU can be used more efficiently and the waiting time can be reduced. The 2-0.in file and 2-1.in can run it at the same time. For the coding part, we will use pthread_create to create a new thread and call the function to check the file location. If the location exists, then it will count the word. The utilization rate of both the CPU and the disk is increased. At the same time, this design will make each thread only care about its own files, and it also makes the logic simpler.

E.g.
Reading 1 file needs 1s. Now, we have 2 files.

Single thread: 1s + 1s = 2s

Multiple thread: 1s

This is the difference and the result. Also the requirements are all test cases within 2s process time. We must use multiple threads to meet the requirements.

And in the mian (argc, **argv) based variable use we find that this is not recognized in the face of the input case is a directory, so we need to deal with the input address.

First, to save time, we define a large-capacity pointer array to store file addresses, and then, through a custom function, first perform the first coarse screening, non-directory addresses

are stuffed directly into the pointer array, after which, if it is a directory, we recursively enter the directory and iterate through the files in the directory until there is no directory.

After the completion of the file address array filling, we will use the dictionary order to sort, here with a simple bubble sort to deal with, although the time complexity is not as good as some advanced algorithms, but this number is also small, so there is no time wasted in this development, but later optimization can be used to merge sort to deal with, if the fast row, the data is the worst case for the order itself, so it is not recommended here with fast row

1.2 How to parallelize the character counting

Earlier we talked about how we get all the file pointers, and below we will discuss how to work with file pointers.
In this regard, we originally intended to use the multi-process model, i.e. we don't need to bother with the thread calls of the program itself, all of which can use IO and CPU efficiently. The design idea is.

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

figure 1.2.1 Multi process operation

In the development, we can use the normal development mode. At the same time, we can set the file pointer to share and make shared calls through shared memory or message passing. However, due to the project requirements, we have to give up this design.
On multithreading, the general practice is consumer producer mode, that is, the producer's cache size is 1 ~ n, the producer locks the buffer, reads characters from the file, fills the buffer, releases lock, waits for the buffer to be 0, and is controlled by mutually exclusive semaphores.
In terms of thread operation, we can read the file content in segments through multiple threads, which can become a many to many relationship model.
The specific design is as follows:

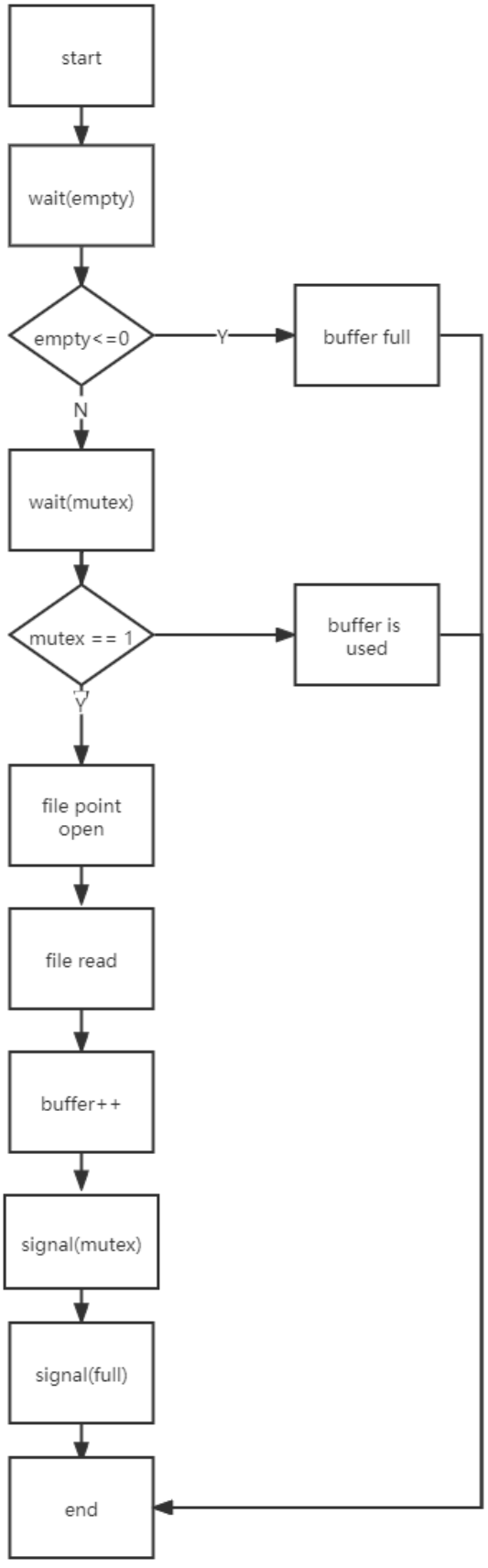

Figure 1.2.2 Producer flow chart

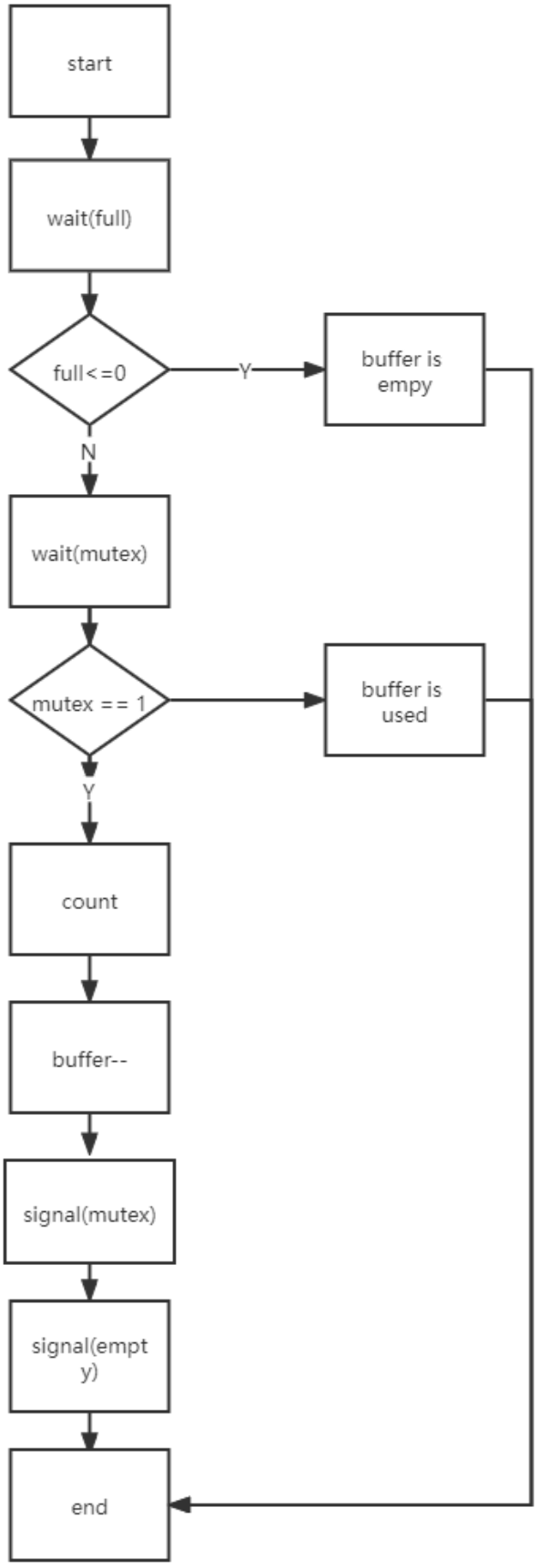

Figure 1.2.3 Consumer flow chart

However, in the end, we found that due to our limited technical ability and time, especially as an EE / CDE student, I have never used linux or written C before. This project is very difficult for me, so it is difficult to achieve this complex control.

Therefore, I abandoned the complex and error prone semaphore control mode and changed it to file based multithreading mode, but it can also be upgraded during optimization in the future, because in theory, the upgrade can be completed as long as another layer of shell is wrapped.

This mode may also be optimized. For example, when reading a large file, we can read in segments, especially after memory mapping (MMAP), we can read the characters of the file directly. Different from the traditional read call, the operation flow of read is that the operating system reads the disk file to the page cache, and then copies the data from the page cache to the buffer passed by read. If MMAP is used, the operating system only needs to read the disk to the page cache, and then the user can directly operate the memory mapped by MMAP through a pointer, The data copy from kernel state to user state is reduced, which can theoretically increase the reading and writing speed and effectively reduce the memory space consumption.

However, I do not recommend that I use multiple threads to access a file when looking for such methods, because in the disk system, when there are multiple different physical addresses, the seek time of the disk will be increased, but I think this part can still be optimized. For example, when the CPU and IO are relatively idle and only a few large files are left, the seek time can be increased accordingly, In exchange for more threads to complete file reading, which is also an optimization direction.

After the program is completed and tested, in reality, the increase of seek time does not seem to be too serious for the increase of file reading time. Therefore, we will discuss the feasibility plan of splitting large files by multiple threads.

Ideally, it is a dynamically planned allocation thread. When the producer needs more threads, it allocates more threads. When it does not need more threads, it releases those redundant threads, and so does the consumer.

However, due to our limited technical capacity, it may be initialization setting. The producer accounts for half of the total threads and the consumer also accounts for half. Then the producer reads and the consumer calculates.

But I think we can optimize it again. Instead of using the producer, we directly map each file to the virtual address, split the file into the total number of threads, and then each consumer thread directly counts the file offset allocated to it. For example, for a file of 20000000 byte, suppose our total number of threads is 100, We set the offset of 200000 for each thread, so that we can read one file by multiple threads, and multiple files are set as loops to complete the whole program design.
Unfortunately, we didn't have enough time in the end. There were too many homework and projects. I regret that we all took six courses. In addition, we are EE / CDE students, and our

programming ability is about nil, and C has not been written, Even C + + is only learned this year. Compared with CS students, it is very difficult to complete it.

So to sum up, our project now uses MMAP for memory mapping, converts the file space into virtual memory, then each thread reads a file, and finally subtracts "a" of ASCII characters from each character to obtain the alphabetic sequence number of each character, which is saved in the array for easy output later.

0	0x41	A	16	0x51	Q	32	0x67	g	48	0x77	w
1	0x42	B	17	0x52	R	33	0x68	h	49	0x78	x
2	0x43	C	18	0x53	S	34	0x69	i	50	0x79	y
3	0x44	D	19	0x54	T	35	0x6a	j	51	0x7a	z
4	0x45	E	20	0x55	U	36	0x6b	k	52	0x30	0
5	0x46	F	21	0x56	V	37	0x6c	l	53	0x31	1
6	0x47	G	22	0x57	W	38	0x6d	m	54	0x32	2
7	0x48	H	23	0x58	X	39	0x6e	n	55	0x33	3
8	0x49	I	24	0x59	Y	40	0x6f	o	56	0x34	4
9	0x4a	J	25	0x5a	Z	41	0x70	p	57	0x35	5
10	0x4b	K	26	0x61	a	42	0x71	q	58	0x36	6
11	0x4c	L	27	0x62	b	43	0x72	r	59	0x37	7
12	0x4d	M	28	0x63	c	44	0x73	s	60	0x38	8
13	0x4e	N	29	0x64	d	45	0x74	t	61	0x39	9
14	0x4f	O	30	0x65	e	46	0x75	u	62	0x2b	+
15	0x50	P	31	0x66	f	47	0x76	v	63	0x2f	/

Figure 1.2.4 ASCII Table

1.3. How to determine the number of threads to create.

The easiest and most convenient way is to use the array method for processing. First, we create an array of length 26 the index from 0 - 25 and initialize all the counts to be 0. It should be like this:

```
count[0] = 0;
count[1] = 0;
count[2] = 0;
……
count[25] = 0;
```

Now we need to think about how to not affect the work of other threads. Therefore we need to have created each thread for each path when we read the file. So every thread will have an array from 0-25. Like:

Thread 1	Thread 2	Therad 3
count[0] = 0;	count[0] = 0;	…….
count[1] = 0;	count[1] = 0;	…….
count[2] = 0;	count[2] = 0;	…….
……	…..	…..
count[25] = 0;	count[25] = 0;	…….

In this way, it can be ensured that the work of each thread is to calculate the word for its own file path. If there are too many threads, the biggest impact should be that the number of context switches becomes too large, which will affect performance. The system runs slowly and the CPU is 100% used.So it is very important to choose how many threads. In summary, determining the number how many threads the program will create are based on how many files we need to test.

If we change to the consumer producer mode or other modes, we can determine the number of threads through testing, because when there are few threads, the speed will naturally slow down due to fewer processors, but when there are many threads, the performance will decline after reaching a critical value due to thread switching loss, But generally speaking, we will query the CPU cores of the server through the API, so as to define a linear model to test which variable is multiplied by the number of cores, which will make the program most efficient.

1.4 How to write the counting results for each input file concurrently

Firstly, we initialize a large-capacity two-dimensional array with a length of 26 bits of global variables to store the results of file statistics. Since our program is a single thread single file, we don't need to consider the occupation of the array. If it is a multi thread single file, we need to consider the situation that the data address is occupied by different threads, At this time, we should use semaphores or lock shared resources.
The general idea of locking shared resources is as follows:

```
i = filenum, num = Alphabetic number
if(trylock(savestr[i]))
{
        savestr[i][num -'a']=+1;
        freelock(savestr[i])
}
```

Finally, through traversal, we output file statistics according to the order of file array (after dictionary order)

1.5 How to efficiently count different characters in multiple files in parallel

We mentioned how to process a single file based on multithreading, so next I will discuss the overall design of the program in detail.

First, we will read the number of input files and the file pointer array in the mian function, and then put it into the directory judgment function mentioned earlier for traversal.

In the directory determination function, we will first roughly screen the file address. If the encountered file address belongs to the file type, we use opendir() to read it. If it is empty, it means that the address is a file. If it is a return pointer, we will output the next step. First, replace the default address slash symbol and change '/' to '\ 0'

After that, the loop is carried out. When the auxiliary pointer under the read directory is not empty, the address is continuously recursive for the directory judgment function.
After this step, we will get all the file addresses and the total number of files.

Then we need to sort the file addresses in dictionary order. Here we use bubble sort to reduce the workload.

After that, we need to generate threads. Here, we need to call the CPU parameter to find the number of cores, get_ Nprocs (), and then we find that the number of school server cores is 96. Generally speaking, the number of threads is set to 96 * 1.5.

We then traverse according to the number of files, divide the file by 144 (96 * 1.5) according to the size in the file structure, and then perform secondary traversal, generate the pointer from No. 0 to 143, and fill in the countfile (No. * cutting size, No. * (cutting size + 1)) in the function generated by the pointer.

After that is the statistical function and output function. I won't talk about it in detail because I mentioned it in detail before.

However, at present, in order to take into account the development time, we can only sacrifice a little speed to turn multi-threaded reading of a single file into single threaded reading of a single file, which greatly reduces the difficulty of development and leaves room for subsequent upgrading. As long as we put on a layer of shell and add a little internal change, we can complete the program upgrade, but unfortunately, No more time and only one developer.

2.Implementation

```
Output function, which is used to output file statistics
Enter file serial number for int.
It can output statistics about the file
*/
void PrintCount(int idx){
    int i;
    char c;
    printf("%s\n",filenames[idx]);//Output file path
    for(i=0;i<26;i++){//Output statistics of 26 letters
        if(cnt[idx][i] > 0){
            c = 'a' + i;//Because the letter 'a' was subtracted before,
                        //it is now added back to become a normal ASCII
character code
            printf("%c %d\n",c,cnt[idx][i]);
        }
    }
}
/*
void *CountFile(void *args)
Counting function, used to count the number of different letters in the
file.
The input is a typeless pointer,
but the general input is an int pointer,
which points to the file serial number.

If there is no return, the data will be stored in the global variable
**cnt
*/
void *CountFile(void *args){
    int *idx = (int*)args;
    int i;
    char *buf;
    struct stat st;

    FILE *fp = fopen(filenames[*idx],"r");//read file
    int fd = fileno(fp);//Gets the file descriptor used by the file
stream
    fstat(fd, &st);//Get file status from file descriptor
    buf = mmap(NULL, st.st_size, PROT_READ, MAP_SHARED, fd, 0);//Memory
mapping
```

```
    for(i=0;i<st.st_size;i++){
        if(buf[i]<'a' || buf[i]>'z') continue;//Because I found capital
letters
        cnt[*idx][buf[i]-'a']++;//letter num ++
    }
    fclose(fp);
    return NULL;
}

/*
void get_all_files(char *path)
Directory judgment function
Enter the file address of char *
No return
Add the read file to the file address array.
If a directory is encountered,
add all the files in the directory to the file address array
until the directory root is reached
*/
void get_all_files(char *path){
    DIR *dir;
    struct dirent *ptr;
    int path_len = strlen(path);

    // If the current path is not a directory, it defaults to file
    if((dir=opendir(path)) == NULL){
        filenames[file_num++] = path;
        return;
    }

    // At this time, you can conclude that the path is a folder path,
    //and you need to remove the trailing slash
    if(path[path_len-1]=='/'){
        path[path_len-1]='\0';
        path_len--;
    }

    while((ptr=readdir(dir)) != NULL){
        if(strcmp(ptr->d_name,".")==0 || strcmp(ptr->d_name,"..")==0){
            //Not the current directory or parent directory
            continue;
        }
        int subfilepath_len = strlen(ptr->d_name);//Get address length
```

```
        char *subpath = (char*)malloc(path_len+subfilepath_len+2);
        //Create new address pointer
        strcpy(subpath, path);//copy
        strcpy(subpath+path_len, "/");
        strcpy(subpath+path_len+1, ptr->d_name);//to new dir
        get_all_files(subpath);//Enter secondary directory
    }
    closedir(dir);
}

// Swap two strings
void swap(char **a, char **b){
    char *t = *b;
    *b = *a;
    *a = t;
}

// Using bubble sorting method to sort files
//Allows files to be output in dictionary order
void bubble_sort(char *a[],int len){
    int i,j;
    for(i=0;i<len;i++){
        for(j=len-1;j>=i+1;j--){
            if(strcmp(a[j], a[j-1])<0){
                swap(&a[j], &a[j-1]);
            }
        }
    }
}

int main(int argc, char** argv){
    //sem_t semLock;
    int i;
    pthread_t *pThreads;

    //Null parameter
    if(argc==1){
        puts("pzip: file1 [file2 ...]");
        return 1;
    }

    // sem_init(&semLock,0,1);
```

```
    for(i=1;i<argc;i++){
        get_all_files((char*)argv[i]);
    }

    //Bubble dictionary sort
    bubble_sort(filenames, file_num);
    //Thread array
    pThreads = (pthread_t *)malloc(file_num*sizeof(pthread_t));
    //Based on single file, single thread traversal
    for(i=0;i<file_num;i++){
        int *indices = (int*)malloc(sizeof(int));
        *indices = i;
        pthread_create((pThreads+i), NULL, CountFile, indices);
    }
    //Wait for all threads to end
    for(i=0;i<file_num;i++){
        pthread_join(*(pThreads+i),NULL);
    }
    //free
    free(pThreads);
    //print
    for(i=0;i<file_num;i++){
        PrintCount(i);
    }

    return 0;
}
```

3. Evaluation

3.1. Correctness

We tested all the test cases and the results are all passed within the 2s running times required.

```
kingsfung8@ubt18a:~/project$ make test
TEST 0 - clean build (program should compile without errors or warnings)
Test finished in 0.161 seconds
RESULT passed

TEST 1 - single file test, a small file of 10 MB (2 sec timeout)
Test finished in 0.044 seconds
RESULT passed

TEST 2 - multiple files test, twelve small files of 10 MB, 20 MB, 30 MB, 10 MB, 20 M
Test finished in 0.133 seconds
RESULT passed

TEST 3 - empty file test (2 sec timeout)
Test finished in 0.008 seconds
RESULT passed

TEST 4 - no file test (2 sec timeout)
Test finished in 0.006 seconds
RESULT passed

TEST 5 - single large file test, a large file of 100 MB (2 sec timeout)
Test finished in 0.301 seconds
RESULT passed

TEST 6 - multiple large files test, six large files of 100 MB, 200 MB, 300 MB, 100 M
Test finished in 0.966 seconds
RESULT passed

TEST 7 - directory test, a directory that contains twelve small files of 10 MB, 20 M
 timeout)
Test finished in 0.172 seconds
RESULT passed

TEST 8 - mixed test 1, a directory that contains six small files of 10 MB, 20 MB, 10
MB, 300 MB, 100 MB, 200 MB, 300 MB (2 sec timeout)
Test finished in 1.008 seconds
RESULT passed
```

```
TEST 9 - mixed test 2, a directory that contains six large fi
, 40 MB, 30 MB, 40 MB, 30 MB, 40 MB (2 sec timeout)
Test finished in 0.706 seconds
RESULT passed

TEST 10 - mixed test 3, two directories that contain three sm
 outside directory of 10 MB, 20 MB, 10 MB, 20 MB, 10 MB, 20 M
Test finished in 0.639 seconds
RESULT passed
```

3.2. Performance

Although, the program can run successfully and correctly. But the run time is not very ideal. Because part of the test case takes about 1s to complete the entire inspection. For example: test case 6 needs 0.966s and test case 8 needs 1.008s.

Problem:

1. Files Size

One of the most influential elements of running time must be the size of the file. For test 6 contains 6 large files and each file over 100MB. For test 8 also contains 6 large files and 6 small files. Since, the operating criteria in our program is based on a file to create a thread processing, it is a weakness for our program to deal with these large files. Relatively, when dealing with small size files, the program can have the best performance and also is our program's strengths. In summary, our program is most suitable for processing large numbers of small files.

2. Output Order

Another issue that affects performance is that the program needs to wait for all threads to complete work before outputting results. Because the output requirement is ordered, it is necessary to wait for all files to have passed, and then output all the results one by one according to the position of the file name sorted by the bubble sorting method. In this way, the small file must be the fastest to complete the calculation, but it cannot be output immediately. It can be output only after the large file also is completed, which wastes part of the time.

Improve:

1. Create more threads

If we want to improve the existing program, the most effective way is to create more threads. The current program is one thread that controls one file to calculate the number of a-z. However, if we can create 26 threads for each file, each is responsible for calculating a-z. Increasing the speed of calculation may reduce a lot of time. But, in terms of execution conditions, the required technology is a very powerful task that can be completed. Also, it will take a long time to understand and try. So we finally chose a simple method to implement.

If we have more time, we can change the program to the design mentioned in the above design