RAMAIAH UNIVERSITY OF APPLIED SCIENCES

Assignment

Course Code	CSC302A
Course Name	Operating Systems
Programme	B.Tech
Department	CSE
Faculty	FET

Name of the Student	Satyajit Ghana
Reg. No.	17ETCS002159
Semester/Year	5/2019
Course Leader(s)	Ms. Naveeta

Declaration Sheet

Student Name	Satyajit Ghana		
Reg. No	17ETCS002159		
Programme	B.Tech	Semester/Year	05/2019
Course Code	CSC302A		
Course Title	Operating Systems		
Course Date		to	
Course Leader	Ms. Naveeta		

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Contents

List of Figures

1 Question 1

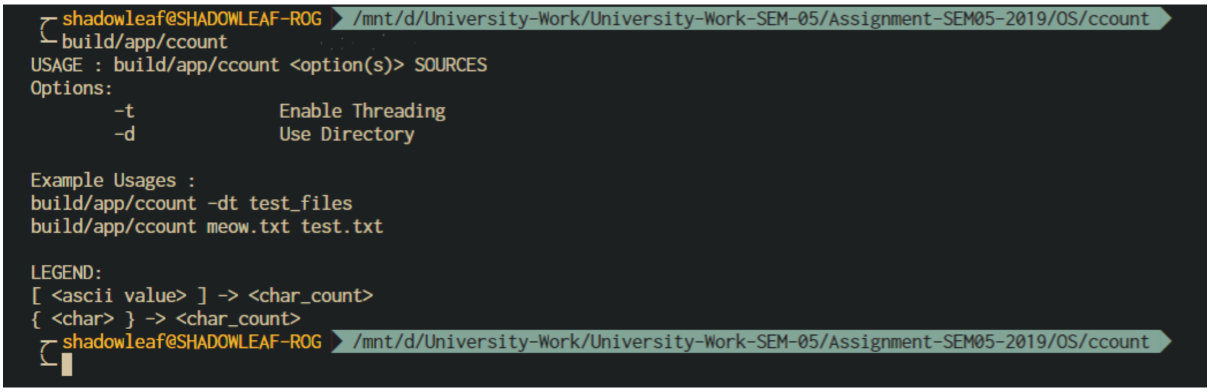
Solution to Question No. 1

1.1 Development of the Application

Complete Source Code of the Program is attached in Appendix A

The approach to the problem is to have two methods, sequential and multithreaded approach, the option is taken as a command line argument for the same. In both the methods the common task done by main thread is to combine the result obtained from the threads.

The structure of output of the program is as mentioned in the below figure, the help is displayed if no parameter is given to the program.

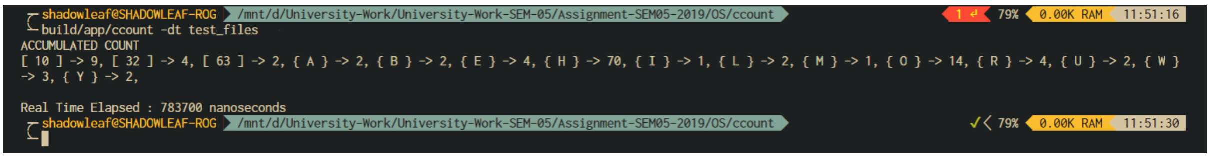


```
┌ shadowleaf@SHADOWLEAF-ROG  /mnt/d/University-Work/University-Work-SEM-05/Assignment-SEM05-2019/OS/ccount
└ build/app/ccount
USAGE : build/app/ccount <option(s)> SOURCES
Options:
        -t              Enable Threading
        -d              Use Directory

Example Usages :
build/app/ccount -dt test_files
build/app/ccount meow.txt test.txt

LEGEND:
[ <ascii value> ] -> <char_count>
{ <char> } -> <char_count>
┌ shadowleaf@SHADOWLEAF-ROG  /mnt/d/University-Work/University-Work-SEM-05/Assignment-SEM05-2019/OS/ccount
└
```

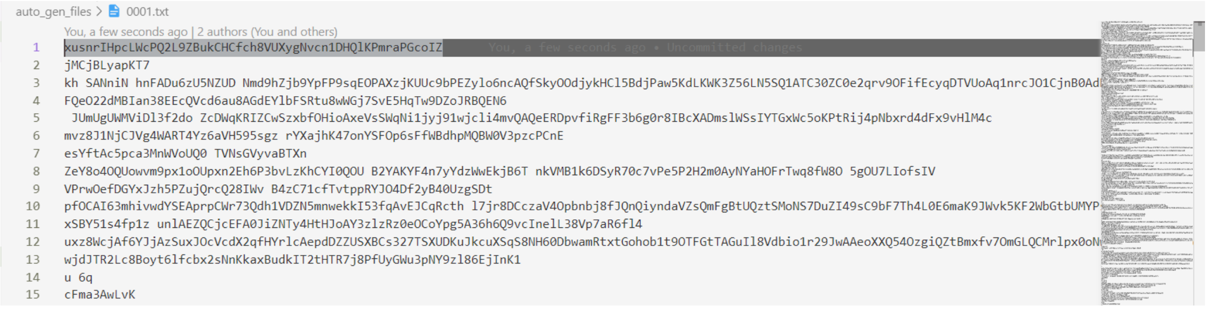
Figure 1-1 ccount help



```
┌ shadowleaf@SHADOWLEAF-ROG  /mnt/d/University-Work/University-Work-SEM-05/Assignment-SEM05-2019/OS/ccount      1 ↲  79%  0.00K RAM  11:51:16
└ build/app/ccount -dt test_files
ACCUMULATED COUNT
[ 10 ] -> 9, [ 32 ] -> 4, [ 63 ] -> 2, { A } -> 2, { B } -> 2, { E } -> 4, { H } -> 70, { I } -> 1, { L } -> 2, { M } -> 1, { O } -> 14, { R } -> 4, { U } -> 2, { W }
-> 3, { Y } -> 2,

Real Time Elapsed : 783700 nanoseconds
┌ shadowleaf@SHADOWLEAF-ROG  /mnt/d/University-Work/University-Work-SEM-05/Assignment-SEM05-2019/OS/ccount      ✓  79%  0.00K RAM  11:51:30
└
```

Figure 1-2 ccount sample output


auto_gen_files > 0001.txt
You, a few seconds ago | 2 authors (You and others)
You, a few seconds ago • Uncommitted changes

```
xusnrIHpcLWcPQ2L9ZBukCHCfch8VUXygNvcn1DHQlKPmraPGcoIZ
jMCjBLyapKT7
kh SANniN hnFADu6zU5NZUD Nmd9hZjb9YpFP9sqEOPAXzjKUUc9bPEZylo6ncAQfSkyOOdjykHCl5BdjPaw5KdLKWK3Z56LN5SQ1ATC30ZC0e2qrv9OFifEcyqDTVUoAq1nrcJO1CjnB0Ad
FQeO22dMBIan38EEcQVcd6au8AGdEYlbFSRtu8wWGj7SvE5HqTw9DZoJRBQEN6
 JUmUgUWMViDl3f2do ZcDWqKRIZCwSzxbfOHioAxeVsSWqNi1jyj91wjcli4mvQAQeERDpvfiRgFF3b6g0r8IBcXADmslWSsIYTGxWc5oKPtRij4pNbxrd4dFx9vHlM4c
mvz8J1NjCJVg4WART4Yz6aVH595sgz rYXajhK47onYSFOp6sFfWBdhpMQBW0V3pzcPCnE
esYftAc5pca3MnWVoUQ0 TVNsGVyvaBTXn
ZeY8o4OQUowvm9px1oOUpxn2Eh6P3bvLzKhCYI0QOU B2YAKYF4n7yYdzWwEkjB6T nkVMB1k6DSyR70c7vPe5P2H2m0AyNYaHOFrTwq8fW8O 5gOU7LIofsIV
VPrwOefDGYxJzh5PZujQrcQ28IWv B4zC71cfTvtppRYJO4Df2yB40UzgSDt
pfOCAI63mhivwdYSEAprpCWr73Qdh1VDZN5mnwekkI53fqAvEJCqRcth l7jr8DCczaV4Opbnbj8fJQnQiyndaVZsQmFgBtUQztSMoNS7DuZI49sC9bF7Th4L0E6maK9JWvk5KF2WbGtbUMYP
xSBY51s4fp1z unlAEZQCjcEFA0JiZNTy4HtHJoAY3zlzRz0eOCoYpg5A36h6Q9vcInelL38Vp7aR6fl4
uxz8WcjAf6YJjAzSuxJOcVcdX2qfHYrlcAepdDZZUSXBCs327TSXUDKuJkcuXSqS8NH60DbwamRtxtGohob1t9OTFGtTAGuIl8Vdbio1r29JwAAeoXXQ54OzgiQZtBmxfv7OmGLQCMrlpx0oN
wjdJTR2Lc8Boyt6lfcbx2sNnKkaxBudkIT2tHTR7j8PfUyGWu3pNY9zl86EjInK1
u 6q
cFma3AwLvK
```

Figure 1-3 ccount - sample test file

The test files are generated using Bash Scripting in Linux, the source code for the script can be found in `generate_files.sh`, the goal is to generate alphanumeric characters with spaces and newlines randomly and dump it into a file with a proper name.

1.1.1 Using Sequential Approach

The sequential approach is pretty straight forward, the files are read from the disk one by one by the main thread and the result is pushed to a vector, here all the work is done by the main thread, i.e. disk I/O, counting the characters and also accumulating the results.

Algorithm:

```
1. Open the files using open syscall
2. Iterate over each of the file
3.      For each file get the character count and store in results
4. Iterate over the results and accumulate the character count
```

```
get_char_count():
1. if fd < 0 then print "File Error"
2. lseek(fd, 0, SEEK_SET) // go to start of the file
3. while (read(fd, &buffer, 1) == 1)
4.      ccount[buffer[0]]++
```

The sequential approach code is as simple as

```
for (auto& file : files) {
    std::map<char, int>* ccount = new std::map<char, int>();
    *ccount = file.get()->get_char_count();
    results.emplace_back(static_cast<void*>(ccount));
}
```

Here we iterate over all the files in the directory and then run `get_char_count()` for each of the file, which counts the characters and the results are stored in the results vector.

The function definition of `get_char_count` can be found in Appendix A. It's a simple iterate over all the contents of the file character-by-character and then add the count into a `std::map`.

1.1.2 Using Multithreaded Approach

In the Multithreaded approach the each of the file is assigned as a task to different thread, the main thread then waits for all these threads to complete their work and the result is pushed to the result vector, the main thread then does the accumulation of the results.

Algorithm:

```
1. Open the files using syscall
2. Generate threads using pthread_create
3. Call get_char_count() for each file in each thread
4. Get the results of threads using pthread_wait
5. Accumulate the results in the main thread
```

The threads are generated using the `gen_worker_threads` function,

```
// generate and run threads
        std::vector<pthread_t> threads = bromine::threader::gen_worker_threads(&bro
mine::file::threadable_ccount_fun, fargs);
```

this function generates `threads.size()` number of threads and runs them using `pthread_create`, the function arguments are taken as `fargs`, the function then returns the thread ids of the threads generated.

The `gen_worker_threads` function is more elaborated as

```
std::vector<pthread_t> bromine::threader::gen_worker_threads(void* (*thread_fun)(void*)
, std::vector<void*> fargs) {
    // create a vector of worker threads
    std::vector<pthread_t> worker_threads(fargs.size());

    for (int i = 0; i < (int)worker_threads.size(); i++) {
        pthread_create(&worker_threads[i], NULL, thread_fun, fargs[i]);
    }
    return worker_threads;
}
```

The results are accumulated using `pthread_join` function, that waits for all the threads that were spawned earlier to complete and stores the data in a vector.

```
std::vector<void*> bromine::threader::get_threads_results(std::vector<pthread_t> thread
s) {
    auto results = std::vector<void*>(threads.size());

    for (int i = 0; i < (int)threads.size(); i++) {
        pthread_join(threads[i], &results[i]);
    }

    return results;
}
```

1.2 Comparison of Execution time and Analysis

For comparing the results, mostly CPU usage is affected, since in the single threaded application, only one the available cores are used, to do the testing **`perf`**, a testing utility for Linux was used that provided satisfiable results, the testing utility was created in bash scripting for automation, the source code for which can be found in `testing.sh`, which is attached in Appendix A. Since a single input character will be too small test case to test the application, the input character was chosen to be all the ASCII characters available, this will ensure that the results obtained to comparable.

Also just 3 files are not sufficient to compare the execution times, hence 400 files were taken, if we take 3 files, the results are fluctuating, and the execution time measurement needs very high precision, which is not possible, this is same as the Heisenberg's uncertainty principle that the instrument that we use to measure the time itself can add some extra time to the program, so it becomes difficult to compare execution time, when we test on 400 files, this becomes negligible and we get similar test results for various number of runs.

400 Files with 40000 characters each were used for testing and benchmarking the programs.

Testing Bench specifications is as follows

```
CPU: Intel Core i3-6006U @2.00Ghz 3M Cache

[ 2 Cores 4 Threads(HyperThreading ON) ]

RAM: 4GB LPDDR4

DISK: Seagate 500GB Barracuda 5400rpm

OS: Manjaro (Arch Linux) [ Linux Kernel 4.9 ]

COMPILER: GCC 9.1 with POSIX Thread
```

Single Threaded – perf stat

```
ACCUMULATED COUNT
[ 10 ] -> 251225, [ 32 ] -> 249547, { 0 } -> 249984, { 1 } -> 249856, { 2 } -> 249470, { 3
} -> 250628, { 4 } -> 250043, { 5 } -> 250199, { 6 } -> 250399, { 7 } -> 249785, { 8 } ->
250736, { 9 } -> 250580, { A } -> 249349, { B } -> 249697, { C } -> 250086, { D } -> 250345,
{ E } -> 250988, { F } -> 249419, { G } -> 250584, { H } -> 250114, { I } -> 250605, { J }
-> 249629, { K } -> 250470, { L } -> 249376, { M } -> 250031, { N } -> 249913, { O } ->
249845, { P } -> 249611, { Q } -> 249478, { R } -> 249601, { S } -> 249083, { T } -> 250680,
{ U } -> 250305, { V } -> 249700, { W } -> 249404, { X } -> 249967, { Y } -> 250037, { Z }
-> 250192, { a } -> 249858, { b } -> 249909, { c } -> 250924, { d } -> 249329, { e } ->
250550, { f } -> 249603, { g } -> 250269, { h } -> 250333, { i } -> 249218, { j } -> 249101,
{ k } -> 249405, { l } -> 250815, { m } -> 250157, { n } -> 250132, { o } -> 250876, { p }
-> 249498, { q } -> 250057, { r } -> 249850, { s } -> 249597, { t } -> 250008, { u } ->
249903, { v } -> 249462, { w } -> 250003, { x } -> 248996, { y } -> 250862, { z } -> 250324,

Real Time Elapsed : 16722432861 nanoseconds

 Performance counter stats for 'build/app/ccount -d auto_gen_files':

         16,726.46 msec task-clock                #    1.000 CPUs utilized
                27      context-switches          #    0.002 K/sec
                 0      cpu-migrations            #    0.000 K/sec
               460      page-faults               #    0.028 K/sec
    33,374,743,105      cycles                    #    1.995 GHz
    25,169,226,542      instructions              #    0.75  insn per cycle
     4,870,832,127      branches                  #  291.205 M/sec
        99,689,709      branch-misses             #    2.05% of all branches

      16.728176104 seconds time elapsed

       8.150538000 seconds user
       8.535908000 seconds sys
```

Multithreaded – perf stat

```
ACCUMULATED COUNT
[ 10 ] -> 251225, [ 32 ] -> 249547, { 0 } -> 249984, { 1 } -> 249856, { 2 } -> 249470, { 3
} -> 250628, { 4 } -> 250043, { 5 } -> 250199, { 6 } -> 250399, { 7 } -> 249785, { 8 } ->
250736, { 9 } -> 250580, { A } -> 249349, { B } -> 249697, { C } -> 250086, { D } -> 250345,
{ E } -> 250988, { F } -> 249419, { G } -> 250584, { H } -> 250114, { I } -> 250605, { J }
-> 249629, { K } -> 250470, { L } -> 249376, { M } -> 250031, { N } -> 249913, { O } ->
249845, { P } -> 249611, { Q } -> 249478, { R } -> 249601, { S } -> 249083, { T } -> 250680,
{ U } -> 250305, { V } -> 249700, { W } -> 249404, { X } -> 249967, { Y } -> 250037, { Z }
-> 250192, { a } -> 249858, { b } -> 249909, { c } -> 250924, { d } -> 249329, { e } ->
250550, { f } -> 249603, { g } -> 250269, { h } -> 250333, { i } -> 249218, { j } -> 249101,
{ k } -> 249405, { l } -> 250815, { m } -> 250157, { n } -> 250132, { o } -> 250876, { p }
-> 249498, { q } -> 250057, { r } -> 249850, { s } -> 249597, { t } -> 250008, { u } ->
249903, { v } -> 249462, { w } -> 250003, { x } -> 248996, { y } -> 250862, { z } -> 250324,
Real Time Elapsed : 6572352826 nanoseconds

 Performance counter stats for 'build/app/ccount -dt auto_gen_files':

         25,699.24 msec task-clock                #    3.907 CPUs utilized
            10,448      context-switches          #    0.407 K/sec
               189      cpu-migrations            #    0.007 K/sec
             1,446      page-faults               #    0.056 K/sec
    51,249,351,391      cycles                    #    1.994 GHz
    27,129,213,412      instructions              #    0.53  insn per cycle
     5,412,543,554      branches                  #  210.611 M/sec
       102,972,422      branch-misses             #    1.90% of all branches

       6.576939478 seconds time elapsed

      10.477496000 seconds user
      14.992854000 seconds sys
```

Single Threaded – perf report

```
# Total Lost Samples: 0
#
# Samples: 71K of event 'cycles:u'
# Event count (approx.): 3241922147
#
# Overhead  Command  Shared Object       Symbol
    72.42%  ccount   ccount              [.] bromine::file::get_char_count
    19.09%  ccount   [unknown]           [k] 0xffffffffa260015f
     6.60%  ccount   libpthread-2.29.so  [.] __libc_read
     1.24%  ccount   ccount              [.] read@plt
     0.12%  ccount   ccount              [.] main
     0.08%  ccount   [unknown]           [k] 0xffffffffa2600b07
     0.08%  ccount   libstdc++.so.6.0.26 [.] std::_Rb_tree_insert_and_rebalance
     0.07%  ccount   libc-2.29.so        [.] malloc
     0.07%  ccount   libc-2.29.so        [.] _int_malloc
     0.04%  ccount   ld-2.29.so          [.] _dl_lookup_symbol_x
     0.03%  ccount   ccount              [.] operator delete@plt
     0.03%  ccount   libstdc++.so.6.0.26 [.] std::local_Rb_tree_decrement
     0.02%  ccount   libc-2.29.so        [.] _int_free
     0.02%  ccount   libstdc++.so.6.0.26 [.] operator new
```

Multithreaded – perf report

```
# Total Lost Samples: 0
#
# Samples: 90K of event 'cycles:u'
# Event count (approx.): 26392820082450
#
# Overhead  Command  Shared Object        Symbol
    40.00%  ccount   ccount               [.] bromine::file::get_char_count
    30.00%  ccount   libpthread-2.29.so   [.] __libc_read
    10.00%  ccount   [unknown]            [k] 0xffffffffa260015f
    10.00%  ccount   libpthread-2.29.so   [.] __pthread_disable_asynccancel
    10.00%  ccount   libpthread-2.29.so   [.] __pthread_enable_asynccancel
     0.00%  ccount   ccount               [.] read@plt
     0.00%  ccount   libc-2.29.so         [.] malloc
     0.00%  ccount   [unknown]            [k] 0xffffffffa2600b07
     0.00%  ccount   libc-2.29.so         [.] _int_malloc
     0.00%  ccount   ccount               [.] main
     0.00%  ccount   libc-2.29.so         [.] _int_free
     0.00%  ccount   libstdc++.so.6.0.26  [.] std::_Rb_tree_insert_and_rebalance
     0.00%  ccount   ld-2.29.so           [.] _dl_lookup_symbol_x
```

1.2.1 Analysis of the performance statistics

An average of 5 execution runs were taken

```
Single Threaded = [16.18496 17.27138 16.72817 17.38249 16.07385]

Average: 16.72817 secs

Multi Threaded = [7.12014 6.03372 6.57693 7.23125 5.92261]

Average: 6.57693 secs
```

Table 1 Performance Analysis

Performance Parameter	**Single-Threaded**	**Multi-Threaded**
Time Elapsed	16.72817 secs	6.57693 secs
IPC	0.75 ins/cycle	0.53 ins/cycle
Context Switches	27	10,448
CPU’s Utilized	1.000	3.907

First thing to observe is that the output for both the programs are same, i.e. multi-threaded and single-threaded, this means that our program is correct and there were no raceconditions or dead-locks in the multi-threaded approach, further testing might be required to test the programs for possible dead-locks, or performance enhancements.

From the above performance parameters, we can clearly observe that the multi-threaded program is approximately 3X faster than the single-threaded. The processor used here had 2 cores and 4 hyperthreaded cores, hence the CPU's utilized in multi-threaded is approximately 4, or all of them, in single threaded only one CPU is utilized. Another thing to note is that, since we had 10000 files and hence 10000 threads that are created, the number of context switches is relatively very high compared to the single threaded program.

From the `perf report` we can determine the sub-routines in the program that cause the major overhead, this tells us how the work is distributed among the threads. The major overhead in our program is `bromine::file::get_char_count`, in single threaded this has 72.42% overhead, while in multithreaded its brought down to 40.00%, since now we have opened multiple files all at once, the `get_char_count` overhead is not there anymore, suppose a thread is currently executing `ccount`, the other thread might be opening a file, hence the overhead is distributed among reading and processing the file, as we can see in the perf report of multithreaded, the `__libc_read` function takes up about 30.00% overhead, in single threaded, the file opening and `ccount` is sequential, i.e. one single thread can open the file and then process the file, majority of the time is spent on processing the file, and only one single CPU is utilized.

1.2.2 Comments

A big disadvantage of this program is that it is not scalable, the number of threads is proportional to the number of files, if the number of files is really large, then the number of context switches are going to increase drastically, this will cause more overhead, making it very inefficient, instead each thread can be assigned some aggregated number of files it can work with, this will even out the workload among the threads.

Another improvement that can be done is to use a good multi programming library like OpenMP or MPI, since they are well optimized and the compiler is well equipped to optimize it even futher.

2 Question 2

Solution to Question 2

2.1 Number of page faults that occur when FIFO, LRU, and Optimal page replacement algorithms are used respectively

Legend:

P - Page Fault

* - Page Hit

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in.

Page Fault – A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

1. FIFO

In FIFO (First-In-First-Out) is one of the simplest page replacement algorithm, in this the Operating System keeps track of the pages in the memory in form of a queue. When the page needs to be replaced, page in the front of the queue is removed and the new page is replaced.

16 Page Faults

FRAME	0	1	2	3	2	3	0	4	5	2	3	1	4	3	2	6	3	2	1	2
	0	1	2	3	3	3	0	4	5	2	3	1	4	4	2	6	3	3	1	2
		0	1	2	2	2	3	0	4	5	2	3	1	1	4	2	6	6	3	1
			0	1	1	1	2	3	0	4	5	2	3	3	1	4	2	2	6	3
	P	P	P	P	*	*	P	P	P	P	P	P	P	*	P	P	P	*	P	P

Given the string, the first three characters are page fault, since they are not in the memory, then 3 comes, which is not in the memory, hence the oldest item in the queue which is 2 is replaced with 3, now 2, 3 comes, which are already present so it's a page hit, when 0 comes it's a page fault, so the oldest item, which is 1 is removed and 0 is added to the stack, this process then continues until the end of the string, the algorithm being, if the item is not in the memory, then the oldest item in the memory is replaced with this new item.

2. LRU

In LRU (Least Recently Used) algorithm is a greedy algorithm, the idea is based on locality of reference, when there is a page fault, then the least recently used block is replaced with the new memory, hence here we have to keep track of the time at which the memory blocks were accessed.

14 Page Faults

FRAME	0	1	2	3	2	3	0	4	5	2	3	1	4	3	2	6	3	2	1	2
	0	1	2	3	**2**	**3**	0	4	5	2	3	1	4	**3**	2	6	**3**	**2**	1	**2**
		0	1	2	3	2	3	0	4	5	2	3	1	4	3	2	6	3	2	1
			0	1	1	1	2	3	0	4	5	2	3	1	4	3	2	6	3	3
	P	P	P	P	*	*	P	P	P	P	P	P	P	*	P	P	*	*	P	*

The solution here uses the stack algorithm of LRU. Initially the first three characters of the string give a page fault, every time a page fault happens the whole stack is pushed down and the new item is added to the top. When 3 comes, it is a page fault, so the stack is pushed down and 3 is added to the top, now 2 comes, which is a page hit, so 3 is pushed-up the stack, i.e. on every page fault, the stack is pushed down, and on every page-hit, the item that is hit is pushed-up. Now 3 enters, which is also a page hit, so the 3 in the stack is pushed-up by a unit. When 0 enters, it is a page-fault so the entire stack is pushed-down and the item 0 is added to the top. The process continues until the end of the string.

3. Optimal Page Replacement

In Optimal Page Replacement pages are replaced which would not be used for the longest duration of time in the future.

10 Page Faults

FRAME	0	1	2	3	2	3	0	4	5	2	3	1	4	3	2	6	3	2	1	2
	0	0	0	0	0	0	**0**	4	5	5	5	1	4	4	4	6	6	6	1	1
		1	1	3	3	**3**	3	3	3	3	**3**	3	3	**3**	3	3	**3**	3	3	3
			2	2	**2**	2	2	2	2	**2**	2	2	2	2	**2**	2	2	**2**	2	**2**
	P	P	P	P	*	*	*	P	P	*	*	P	P	*	*	P	*	*	P	*

Initially the first three items, 0, 1, 2 are page faults since they are not in the memory, now 3 comes, which is a page fault, we observe that 1 which is in the memory is least used in the future, so it is replaced by 3, now 2, 3, 0 are in the page so there is no page fault, when 4 comes it is a page fault, we see that 0 is least used in the future, so 4 will now replace 0 in the memory. This process continues until the end of the string. One thing to observe is that on every page fault we have to look through the entire future characters for the number of occurrences of them, so that we know which page to replace. Since 2 and 3 are most used throughout the string `[2 occurs 6 times; 3 occurs 5 times]` they are not replaced most of the time.

Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

2.2 Diagram of the probability density function of distance strings based on LRU

Assuming LRU algorithm is used.

FRAME	0	1	2	3	2	3	0	4	5	2	3	1	4	3	2	6	3	2	1	2
	0	1	2	3	2	3	0	4	5	2	3	1	4	3	2	6	3	2	1	2
		0	1	2	3	2	3	0	4	5	2	3	1	4	3	2	6	3	2	1
			0	1	1	1	2	3	0	4	5	2	3	1	4	3	2	6	3	3
				0	0	0	1	2	3	0	4	5	2	2	1	4	4	4	6	6
								1	2	3	0	4	5	5	5	1	1	1	4	4
									1	1	1	0	0	0	0	5	5	5	5	5
															0	0	0	0	0	
String Distance	x	x	x	x	2	2	4	x	x	5	5	6	5	3	4	x	3	3	5	2

`x indicates the infinity distance.`

The first 4 items 0, 1, 2, 3 are page fault, since they are not in the page, then when 2 comes, its already there in the page, at a distance of 2, then 3 comes, which is at a distance of 2, 0 comes, which is at a distance of 4. Now when 4 comes, it is a page-fault, so the entire stack is pushed down, since the number of virtual pages is 8, there is no loss, the string distance in case of faults is considered to be infinity. This process is repeated for the rest of the string.

The probability density function is calculation of the number of occurrences of the page frame, for example the distance 1 has occurred 0 times out of 20 times, hence P(1) = 0, and 2 has occurred 3 times out of 20, so P(2) = 3/20, similarly rest of the values of probability is computed. This describes the probability of the string distance from 0 to 1, which will help us determine the optimal number of page frames.

```
P(1) = 0/20 = 0.0
P(2) = 3/20 = 0.15
P(3) = 3/20 = 0.15
P(4) = 2/20 = 0.1
P(5) = 4/20 = 0.2
P(6) = 1/20 = 0.05
P(infinity) = 7/20 = 0.35
```

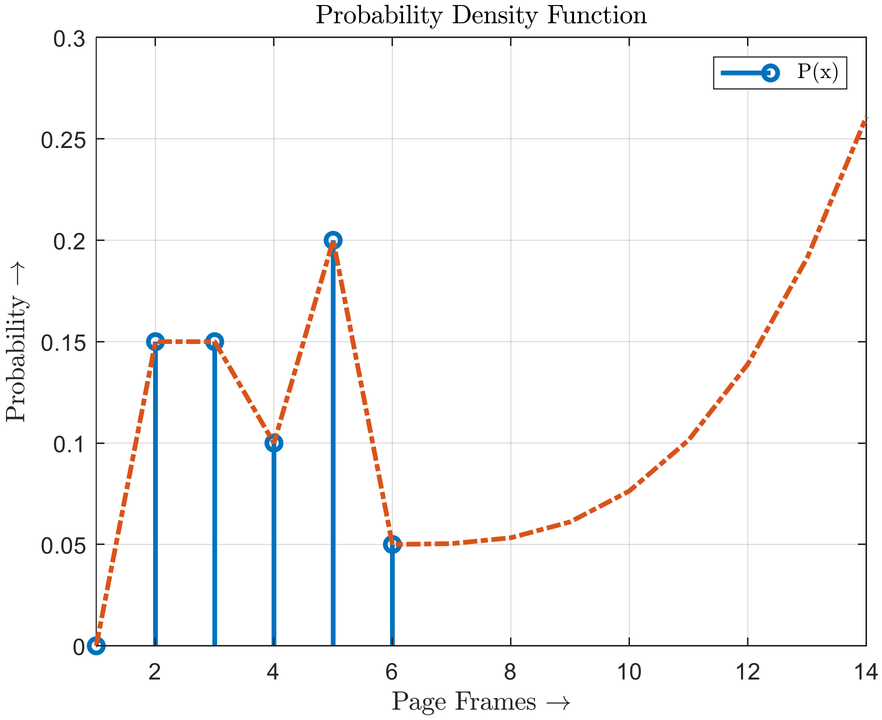


Figure 2-1 Probability Density Function

```
MATLAB Code

clear all;
x = 1:1:6;
y = [0 0.15 0.15 0.1 0.2 0.05];
x = [x, 15];
y = [y, 0.35];
stem(x, y, 'LineWidth', 2);
hold on;
xq = 1:1:10000;
vq = interp1(x, y, xq, 'pchip');
plot(xq, vq, '-.', 'LineWidth', 2);
```

```
xlim([1 14]);
grid on;
hold off;
title('Probability Density Function', 'Interpreter', 'latex')
legend({'P(x)'}, 'Interpreter', 'latex')
xlabel('Page Frames $\rightarrow$', 'Interpreter', 'latex')
ylabel('Probability $\rightarrow$', 'Interpreter', 'latex')
```

2.3 Recommendation of an optimal number of physical page frames appropriate for the given string of accesses

Since, the probability is highest at frame 5 of 0.2 the optimal number of physical page frames appropriate for the given string of accesses is 5, and also because based on the diagram 5 frames would be a good choice.

Theoretically the highest probability is of infinite number of physical frames of 0.35, albeit it is not practical.

Appendix A

Source Code for Program in Question 1

Project Structure

- app
  - file_ops.cpp
  - file_ops.hpp
  - main.cpp
  - threader.cpp
  - threader.hpp

main.cpp

```
// C++ Includes
#include <algorithm>
#include <chrono>
#include <iostream>
#include <map>
#include <memory>
#include <numeric>
#include <vector>

// C Includes
#include <dirent.h>
#include <fcntl.h>
#include <pthread.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

// User defined Includes
#include "file_ops.hpp"
#include "threader.hpp"

void help(char name[]);

int main(int argc, char** argv) {
    // vector of files
    std::vector<std::shared_ptr<bromine::file>> files;

    bool is_directory = false;
    bool is_threaded = false;
    int opt;
    while ((opt = getopt(argc, argv, ":dt")) != -1) {
        switch (opt) {
            case 'd':
                is_directory = true;
                break;
            case ':':
                std::cout << "MISSING FOLDER NAME";
                help(argv[0]);
                exit(EXIT_FAILURE);
                break;
            case '?':
                std::cout << "UNKNOWN OPTION : " << argv[optind];
                break;
            case 't':
                is_threaded = true;
                break;
        }
    }

    if (is_directory) {
        // directory path used
        DIR* d;
        struct dirent* dir;

        d = opendir(argv[optind]);
        if (d) {
            while ((dir = readdir(d)) != nullptr) {
                if (dir->d_type == DT_REG) {
                    std::string rel_path(dir->d_name);
                    rel_path = std::string(argv[optind]) + "/" + rel_path;
                    std::shared_ptr<bromine::file> ptr(new bromine::file(rel_path));
                    files.emplace_back(ptr);
                }
            }
        } else {
            std::cerr << "ERROR OPENING DIRECTORY " << argv[2] << std::endl;
            exit(EXIT_FAILURE);
        }
    } else {
        if (argc == 1) {
            help(argv[0]);
            exit(EXIT_FAILURE);
        }

        for (; optind < argc; optind++) {
            // files are specified in argv
            for (int i = 1; i < argc; i++) {
                std::shared_ptr<bromine::file> ptr(new bromine::file(argv[optind]));
                files.emplace_back(ptr);
            }
        }
    }

    std::vector<void*> results;

    auto start = std::chrono::high_resolution_clock::now();

    if (is_threaded) {  // MULTITHREADED
        // transform the vector of file arguments into void* vector
        std::vector<void*> fargs(files.size());
        std::transform(files.begin(), files.end(), fargs.begin(), [](std::shared_ptr<bromine::file> p) {
            return static_cast<void*>(p.get());
        });

        // generate and run threads
        std::vector<pthread_t> threads = bromine::threader::gen_worker_threads(&bromine::file::threadable_ccount_fun, fargs);
        // join the threads and get the results
        results = bromine::threader::get_threads_results(threads);
    } else {  // SEQUENCIAL
        for (auto& file : files) {
            // please use shared_ptr or unique_ptr
            std::map<char, int>* ccount = new std::map<char, int>();
            *ccount = file.get()->get_char_count();
            results.emplace_back(static_cast<void*>(ccount));
        }
    }

    // variable to store the accumulated results
    std::map<char, int> accumulated_vals;

    // accumulate the results in the main thread
    for (auto& result : results) {
        std::map<char, int> ccount = *static_cast<std::map<char, int>*>(result);
        for (auto& elem : ccount) {
            if (accumulated_vals[elem.first]) {
                accumulated_vals[elem.first] += elem.second;
            } else {
                accumulated_vals[elem.first] = elem.second;
            }
        }
    }

    auto end = std::chrono::high_resolution_clock::now();
    std::cout << "ACCUMULATED COUNT" << std::endl;

    // print the character count
    bromine::file::print_ccount(accumulated_vals);

    auto time_taken = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start)
.count();
    // time_taken *= 1e-9;
    std::cout << "\nReal Time Elapsed : " << std::fixed << time_taken << " nanoseconds" <
< std::endl;
}

// for printing the help documentation
void help(char name[]) {
    std::cerr << "USAGE : " << name << " <option(s)> SOURCES\n"
              << "Options:\n"
              << "\t-t\t\tEnable Threading\n"
              << "\t-d\t\tUse Directory\n"
              << "\nExample Usages : \n"
              << name << " -dt test_files\n"
              << name << " meow.txt test.txt\n"
              << "\nLEGEND:"
              << "\n[ <ascii value> ] -> <char_count>"
              << "\n{ <char> } -> <char_count>"
              << std::endl;
}
```

file_ops.hpp

```
#pragma once

#include <map>
#include <string>

namespace bromine {

class file {
   private:
    // file name
    std::string file_name;
    // the file descriptor
    int fd;
    // to check if the file is closed
    bool isclosed = true;

   public:
    file() {
        file_name = "";
        fd = 0;
    };
    file(std::string file_name);
    ~file();

    // fetch the file descriptor
    int get_fd() { return this->fd; };

    /**
     * Returns the character count
     * @param none
     * @return character count as a map
    */
    std::map<char, int> get_char_count();

    /**
     * Opens the file
     * @param file name as a string
     * @return none
    */
    void open(std::string file_name);
    std::string get_file_name() { return this->file_name; };

    // char count function
    static void print_ccount(const std::map<char, int>& ccount);
    // threadable char count
    static void* threadable_ccount_fun(void*);
};

}  // namespace bromine
```

file_ops.cpp

```
// user includes
#include "file_ops.hpp"

// system includes
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <iostream>
#include <memory>
#include <utility>

/**
```

```
 * Constructor
 * @param string file_name
 * @return object of bromine::file
*/
bromine::file::file(std::string file_name) : file_name(file_name) {
    this->open(this->file_name);
}

/**
 * Destructor
 * close the file if it was open
*/
bromine::file::~file() {
    if (!isclosed) {
        // for debugging
        // std::cout << "CLOSING " << this->file_name << std::endl;
        close(this->fd);
    }
    this->isclosed = true;
}

/**
 * Opens the file, given the file_name
 * @param string file_name
 * @return none
*/
void bromine::file::open(std::string file_name) {
    this->file_name = file_name;

    if (this->fd = ::open(this->file_name.c_str(), O_RDONLY); this->fd != -1) {
        // std::cout << "SUCCESSFULLY OPENED " << file_name << std::endl;
        this->isclosed = false;
    } else {
        throw std::runtime_error("ERROR OPENING FILE : " + file_name);
    }
}

/**
 * Gets the character count from the file
 * @param none
 * @return map of char, int which is the character count
*/
std::map<char, int> bromine::file::get_char_count() {
    std::map<char, int> ccount;

    if (this->fd < 0) {
        std::cerr << "FILE ERROR" << std::endl;
        return ccount;
    }

    char buffer[2];

    // seek to start of the file
    lseek(this->fd, 0, SEEK_SET);

    while (read(this->fd, &buffer, 1) == 1) {
        // for debugging
        // std::cout << "#" << buffer[0] << "#";

        // increment the character count of the character
        ccount[buffer[0]]++;
    }

    return ccount;
}

/**
 * Threadable Character Cound function which can be passed
 * to gen_worker_threads function in threader library
*/
void* bromine::file::threadable_ccount_fun(void* file_obj) {
    bromine::file* file = static_cast<bromine::file*>(file_obj);

    // how stupid am i to use RAW pointer ?
    // never use RAW pointer, use unique pointer and Move Semantics
    // and make use of RVO (Return Value Optimization)
    auto ccount = new std::map<char, int>(file->get_char_count());

    // thread count needs to be type casted to void*
    return static_cast<void*>(ccount);
}

/**
 * Prints the Character Count
 * @param map<char, int> ccount
 * @return none
 *  ccount
 * @return none
*/
void bromine::file::print_ccount(const std::map<char, int>& ccount) {
    // elem.first is the character
    // elem.second is the character count
    for (auto& elem : ccount) {
        if (isalnum(elem.first)) {
            std::cout << "{ " << elem.first << " } -> " << elem.second << ", ";
        } else {
            std::cout << "[ " << static_cast<unsigned>(elem.first) << " ] -
> " << elem.second << ", ";
        }
    }
    std::cout << std::endl;
}
```

threader.hpp

```
#pragma once

#include <pthread.h>
#include <vector>

namespace bromine {
class threader {
   public:
    static std::vector<pthread_t> gen_worker_threads(void* (*thread_fun)(void*), std::ve
ctor<void*> fargs);
    static std::vector<void*> get_threads_results(std::vector<pthread_t>);
};
}  // namespace bromine
// namespace bromine
```

threader.cpp

```
#include "threader.hpp"

#include <pthread.h>
#include <iostream>
#include <map>
#include <memory>

#include "file_ops.hpp"

/**
 * generates worker threads
 * @param pointer to the thread function, arguments to the thread function
 * @return vector of threads
*/
std::vector<pthread_t> bromine::threader::gen_worker_threads(void* (*thread_fun)(void*)
, std::vector<void*> fargs) {
    // create a vector of worker threads
    std::vector<pthread_t> worker_threads(fargs.size());

    for (int i = 0; i < (int)worker_threads.size(); i++) {
        pthread_create(&worker_threads[i], NULL, thread_fun, fargs[i]);
    }
    return worker_threads;
}

/**
 * joins the generated threads and return the results
 * @param vector of threads
 * @return vector of void* which are the results from the threads
*/
std::vector<void*> bromine::threader::get_threads_results(std::vector<pthread_t> thread
s) {
    auto results = std::vector<void*>(threads.size());

    for (int i = 0; i < (int)threads.size(); i++) {
        pthread_join(threads[i], &results[i]);
    }

    return results;
}
```

generate_files.sh

```
#!/bin/bash
for n in $(seq 1 $1); do
    < /dev/urandom tr -dc "\n [:alnum:]" | head -c$2 > $( printf %04d "$n" ).txt
done
```

testing.sh

```
#!/bin/bash

COLS=$(tput cols)

function print_head {
    print_line
    echo -e "$1\n"
}

function print_line {
    printf '=%.0s' $(seq $COLS)
    printf "\n"
}

start=$SECONDS

print_head "Building"
rm -rf build 2> /dev/null
mkdir build
```

```
(cd build; cmake -DCMAKE_BUILD_TYPE=Release ..; make)

# Perf Report Directory
rm -r perf_report &> /dev/null
mkdir perf_report

print_head "TESTING PROGRAM TIMES"

print_head "Generating Files"
rm -rf auto_gen_files 2> /dev/null
mkdir auto_gen_files

(cd auto_gen_files; sh ../generate_files.sh 400 40000)

echo -e "Generated 400 files with 40000 characters each\n"

print_head "Single Threaded Run"

perf stat build/app/ccount -d auto_gen_files |& tee perf_report/perf_stat_st.txt

print_head "Multi Threaded Run"

perf stat build/app/ccount -dt auto_gen_files |& tee perf_report/perf_stat_mt.txt

# Run Perf Benchmarks and save to file
print_head "Running Perf . . . (could take a while) "

# single threaded run
perf record -s build/app/ccount -d auto_gen_files
mv perf.data perf_report/perf_st.data
perf report --stdio -i perf_report/perf_st.data |& tee perf_report/perf_report_st.txt

# multi threaded run
perf record -s build/app/ccount -dt auto_gen_files
mv perf.data perf_report/perf_mt.data
perf report --stdio -i perf_report/perf_mt.data |& tee perf_report/perf_report_mt.txt

echo -e "perf records and output saved to perf_report/\n"

duration=$(( SECONDS - start ))
echo -e "testing took $duration seconds\n"


# print_head "Cleaning Up"
# rm -r auto_gen_files 2> /dev/null
```

Bibliography