School of Computing Technologies

COSC1114 Operating Systems Principles

Assignment 1

	Assessment Type: Individual assignment; no group work. Submit online via Canvas → Assignments → Assignment 1. Clarifications/updates may be made via announcements and relevant discussion forums.
	Due Date: Week 5, Friday 18th August 2023, 11:59pm.
	Weighting: 100 marks that contributes 30% of the total assessment.

1. Overview

This assignment focuses on the concept of **multithreading**.

It is common in modern programming to have a program running multiple threads to solve a problem. In this assignment you will compare the performance of a single-threaded program and multi-threaded program to copy a text file (a text file is not just a file that ends in .txt but any file that contains text, such as source code).

Please note that it is expected that your program will compile and run cleanly on the provided servers:

```
titan.csit.rmit.edu.au
jupiter.csit.rmit.edu.au
saturn.csit.rmit.edu.au
```

Please make sure that your program will suffer no memory leaks or other invalid memory operations.

2. Learning outcomes

This assessment is relevant to the Course Learning Outcomes CLOs 2, 5, and 6.

3. Assessment details

This assessment will determine your ability to

1. Understand the concepts taught over the first 4 weeks of the course.
2. Work independently in self-directed study to research the identified issues.

4. Submission

Your assignment should follow the requirement below and submit via Canvas → Assignments → Assignment 1. You may resubmit the assignment if you need to, only the most recent version will be marked. Failure to follow the requirements incurs up to 10 marks penalty for this assessment.

You will submit two things:

1. Your C++ source code in a zip file.
   If your student ID is s1234567, then please create a zip file named s1234567_OS_A1.zip including the code you have developed and a README file, in which please specify in detail how to run your code on the provided servers.
2. Your report in pdf format.
   Use at least 12-point font size. If your student ID is s1234567, then please create a pdf file named s1234567_OS_A1.pdf.
   Turnitin will not accept PDF image files, forms, or portfolios, files that do not contain highlightable text (e.g., a scanned file - usually an image), documents containing multiple files or files created with software other than Adobe Acrobat. To determine if a document contains actual text, copy, and paste a section or all the text into a plain-text editor such as Microsoft Notepad or Apple TextEdit. If no text is copied over, the selection is not actual text.

It is your responsibility to correctly submit your files. Please verify that your submission is correctly submitted by downloading what you have submitted to see if your submitted file includes the correct content.

Never leave submission to the last minute – you may have difficulty uploading files.

You can submit multiple times – a new submission will override any earlier submissions. **However, if your final submission is after the due time, late penalties will apply.**

5. Academic integrity and plagiarism

It is understood by us that many of the algorithms used in this course have common implementations. You are welcome to look at online code examples to understand possible solutions to the set problems. However, what you submit must be your own work and your submission will be checked and compared with other solutions. **The penalties for submitting code that is not your own can be severe. Do not simply copy other people's work, it is not difficult for us to detect copied work and we will pursue such cases.**

6. Late submission policy

A penalty of 10% per day of the total available marks will apply for each day being late. **After 10 days, you will receive zero marks for the assignment.**

If you want to seek an extension of time for assignment submission, you must have a substantial reason for that, such as unexpected circumstances. Reasons such as, unable to cope with study load, is not substantial. Also, you must apply for an extension as soon as possible. Last minute extensions cannot be granted unless it attracts special consideration.

Please find out how to apply for special consideration online at https://www.rmit.edu.au/students/student-essentials/assessment-and-results/special-consideration/eligibility-and-how-to-apply.

Any student wishing an extension must go through the official procedure for applying for extensions and must apply at least a week before the due date. Do not wait till the submission due date to apply for an extension.

7. Rubric and marking guidelines

The rubric can also be found in **Canvas → Assignments → Assignment 1.**

For each assessment requirement, the rubric is designed to be general and the details of what you can improve on will be outlined by your marker in the comments. Nevertheless, this should be treated as a guide to what to aim for. Please note that code with segmentation faults in the respective part of the assignment we are assessing cannot get above 50%, and code with problems highlighted by `valgrind` cannot get over 75% in that part of the assignment.

Requirement	Little to no Attempt (0%)	Poor (25%)	OK (50%)	Good (75%)	Excellent (100%)
1. File copier without threading (20%)	No attempt	A reasonable attempt but it does not compile.	Code compiles but is an incomplete implementation.	Mostly complete implementation but there are minor problems.	Excellent job. Well done.
2. Multithreaded Program (20%)	No attempt.	Some attempt but did not get the code working or does not compile.	Code compiles but threading is not consistently implemented	Threading works but is not configured based on command line arguments or a similar issue.	Excellent job. Well done.
3. Add locks (mutexes) (20%)	No attempt.	Some attempt to add locks but did not get the code working.	Code compiles with mutexes but there are still major synchronization issues such as deadlocks and race conditions.	Overall good implementation but there are minor issues like forgetting to release the locks before the program exits.	Excellent job. Well done.
4. Avoid busy waiting (20%)	No attempt.	Some attempt but most of the busy waiting is still occurring.	A good attempt but only using the sleep() function.	A good attempt using condition variables but there are minor problems such as some busy waiting still occurring.	Excellent job. Well done.
5. Report including timings (20%)	No attempt at timing code	You attempted timing code but could not get it to work.	Timing code is working but there's not much analysis, or the analysis is incorrect.	Poorly formatted or minor problems with analysis.	Excellent job. Well done.

8. Assignment tasks

Build Environment

You are expected to compile and run your code on `titan`, `jupiter` or `saturn` servers provided by the school. As such some revision of navigating the UNIX environment is advisable.

You may use any `c++` features up to and including `c++20`.

A reminder that you can enable `c++20` features on the servers using the command:

```
$scl enable devtoolset-11 bash
```

You can then let the compiler know you wish to use `c++20` features with the flags:

```
-Wall -Werror -std=c++20
```

And these are the flags we want you to use in your assignment.

Code Quality

Please note that while code quality is not specifically marked for, it is part of what is assessed as this course teaches operating systems principles partly through the software you develop. As such lack of comments, lack of error checking, good variable naming, avoidance of magic numbers, etc., may be taken into consideration by your marker in assigning marks for the components of this assignment, although operating systems principles do take priority.

C Libraries

For this assignment and other assignments in this course we will introduce you to libraries written in the C language. Typically, operating systems provide a set of C functions that allow you to request services. These are what's called system call libraries. We will use the `pthread` library in this assignment which is a system call library that provides threading functionality.

One issue you will face is that `pthread` is **separately** linked which means the executable code for the functions is stored in a different place to the standard library functions.

Hint: You will need to let the compiler know in the linking phase of compilation to link in the `pthread` library by proving the linker flag: `-lpthread`.

Another thing you need to be aware of (you may not have come across before although it is part of the `c++` language) is void pointers (`void *`). A void pointer is a special kind of pointer that has no type associated with it.

Hint: You can assign any other kind of pointer to it, but you must cast that pointer type to another pointer type before you can use it.

Memory Correctness

All accesses to memory are expected to be correct (no reading from uninitialized memory, out of bounds access, not freeing memory that was allocated) and we will check for these things with `valgrind` on the provided UNIX servers. Please check your code prior to submission with `valgrind` and if you are unsure how to do this, ask the teaching staff. Memory incorrect code will attract a deduction especially as this is an operating systems course. You will need to compile each source file with the `-g` flag to get sensible output such as line numbers and variable names.

The command to test your code with `valgrind` is:

```
$valgrind --track-origins=yes --leak-check=full --show-leak-kinds=all
[executable name] [args]
```

For example (all on one line):

```
$valgrind --track-origins=yes --leak-check=full --show-leak-kinds=all
./mtcopier 100 ~/e70949/shared/osp2023/data.512m /tmp/e70949output
```

Note: you should replace `e70949` with your student number.

Starter Code

Starter code for this assignment will be available from this URL:

https://github.com/muaddib1971/osp2023-startup

You are expected to start from this code to implement the required solutions for this assignment.

Task 1 – Single-threaded file copier (20 marks)

Write a single threaded C++ program that reads data from one file and writes it to another. The two files need to be identical at the end of this program. The program should read data one line at a time. You should consider extensibility in your design as we will modify this program in the next step to make it multithreaded. At the very least consider two separate classes for reading and writing.

Your program will be invocable as: `$./copier infile outfile`

where `infile` is the file to be copied and `outfile` is the destination and these may be any valid filesystem path.

Keep a copy of this program as you will compare the performance of this program with the modified version and submit both.

As part of creating this initial program, write a `makefile` that compiles each source file and links together into an executable file.

Your `makefile` should build Task 1 program when I type "`make copier`" from the command line.

Task 2 – Multi-threaded file copier (20 marks)

Modify the program you created in Task 1 to be multithreaded. You will have a 'team' of readers and a 'team' of writer threads. The reader threads will take turns reading lines from the file and storing them in a shared queue. The writers likewise will take turns removing lines from the queue and writing them to the target file.

Update your `makefile` to have a separate target for this new program.

Your `makefile` should build Task 2 program when I type "`make mtcopier`" from the command line.

Task 3 – Insert locks (20 marks)

You will need to insert locks into the appropriate places using the `pthread_mutex` API to lock other threads out of accessing important resources (such as file reading, the queue, etc.) so that there are no race conditions or deadlocks.

Task 4 – Avoid busy waiting (20 marks)

You are to avoid busy waiting. Busy waiting is when a thread does nothing but check if some condition is met repeatedly. Instead, when you are waiting for a condition to be met you should use a `sleep()` function call or if you are aiming for a high distinction, a condition variable (see The API section for details). What this means is that if you use the `sleep()` solution you can get half the marks here but only a solution using the `pthread_cond_*` API can attract full marks.

Your program will be invoked as: `$./mtcopier numthreads infile outfile`

where `numthreads` will be an integer between ~~`10 and 10000`~~ **`25 and 100`**, `infile` and `outfile` are the same as in Task 1, for example:

```
$./mtcopier 100 ~e70949/shared/osp2023/data.1g /tmp/output
```

The experiment

Insert timing code into the code you have developed to measure the duration of the distinct parts of the code such as reading from file, time to gain a lock, etc. You will output the timing information to standard output. Add an optional flag `-t` to the end of the command line arguments to allow this to be turned on. Your aim here is to reflect the relative time of the distinct parts of your code. Add this timing code to both the single-threaded and multi-threaded versions.

For this you should use the C function `clock()`.

`clock()` returns CPU time (number of clock cycles) since the program started. However, we normally want that time in seconds that some tasks took and if there are a series of tasks, we want to capture the duration of each event and only convert to seconds at the end. You will need to include the `<time.h>` header file to use the `clock()` function.

Hint: please note that `clock()` returns `((clock_t)-1)` on error and so you should check for this value when you call the `clock()` function.

To time a single event we would do the following:

```
clock_t start = clock();
... task code goes here ...
clock_t end = clock();
clock_t duration = end - start;
```

This will give us the number of elapsed clock cycles. We can convert this to seconds with a line such as: `int seconds = (int)duration / (double)CLOCKS_PER_SEC;`

We have generated some random data files so you can compare the time it takes for distinct parts of the code you have written.

- If you look in the directory `~e70949/shared/osp2023` on `titan`, `jupiter` or `saturn` servers (this will only work when logged into the terminal, programs like WinSCP and FileZilla do not understand this kind of path) you will see data files of sizes 512 megabytes (`data.512m`), 1 gigabyte (`data.1g`), 2 gigabytes (`data.2g`), and 4 gigabytes (`data.4g`).
- Use these files as input to test the time for parts of your program. When running your experiments, you will need to output to the `/tmp` as there is more space available there. **To avoid conflicts, prepend all output files with your student number and delete the output files after you are done.**

~~You should test each of these four files with 10 threads, 100 threads, 1000 threads, and 10000 threads.~~

You should test each of these four files with 25 threads, 50 threads, 75 threads, and 100 threads.

Note: each run might vary slightly in terms of time, so you want to run each combination multiple times and take the average.

Output the timing results based on these file sizes and collate this data into a table.

Task 5 – The report (20 marks including timing)

Set up your experimental runs to answer the following questions:

1. Why did you choose to measure these parts – provide reasons for each.
2. Where was the most time spent? How does that relate to the operations being done and to the overall time?
3. Did increasing and/or decreasing the number of threads change the performance of the program? How do you account for this?
4. Was it worthwhile to turn this program into a multithreaded program? Explain your reasoning.

Please include sensible experimental results (including timing results of distinct parts based on different file sizes) to support your discussion.

Please ensure your report (no more than 3 pages) is neat and professionally presented.

9. The API

Note: this is a basic API of the functions we want you to use for concurrency and thread safety. Please refer to the man pages for these functions on titan for further details. The man pages also have examples of how to use these functions.

Please note that all functions return result values which you are expected to check in your code.

```
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void*),
    void *arg);
```

The pthread_create() function creates a thread with the specified attributes. The new thread starts execution by invoking start_routine(); arg is passed as the sole argument of start_routine().

Hint: a void pointer is just a pointer that can point to any pointer type. attr can be ignored for this assignment (just pass in a nullptr, the default attributes shall be used).

Find out more on titan: `$man pthread_create`

Your program should wait for the threading function to complete by calling the function:

```
int pthread_join(pthread_t thread, void **retval);
```

The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately. The thread specified by thread must be joinable.

Find out more on titan: `$man pthread_join`

The following functions are used to manipulate mutexes which are variables that enforce mutual exclusion. A mutex can only be locked by one thread at a time and all other threads that try to lock that mutex will sleep until the mutex becomes available.

```
int pthread_mutex_init(
    pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
```

The pthread_mutex_init() function shall initialize the mutex referenced by mutex with attributes specified by attr.

Hint: you can pass in nullptr for attr and the default attributes are used.

Find out more on titan: `$man pthread_mutex_init`

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

The pthread_mutex_destroy() function shall destroy the mutex object referenced by mutex; the mutex object becomes, in effect, uninitialized.

Find out more on titan: `$man pthread_mutex_destroy`

```
int pthread_mutex_lock(pthread_mutext_t *mutex);
```

The mutex object referenced by mutex shall be locked by calling pthread_mutex_lock(). If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

Find out more on titan: `$man pthread_mutex_lock`

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

The pthread_mutex_unlock() function shall release the mutex object referenced by mutex.

Find out more on titan: `$man pthread_mutex_unlock`

Avoiding Busy Waiting

For the **basic marks** for this, you can use the `sleep` function which is in `<unistd.h>`:

```
int sleep (int seconds);
```

For **top marks** however, you will need to use condition variables. These block until some condition is met although they may wake up before then, so you need to use them in a loop. The functions for these are:

```
int pthread_cond_init(pthread_cond_t *cond,
    const ptread_condattr_t *attr);
```

The pthread_cond_init() function shall initialize the condition variable referenced by cond with attributes referenced by attr. If attr is NULL, the default condition variable attributes shall be used.

Find out more on titan: `$man pthread_cond_init`

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

The pthread_cond_destroy() function shall destroy the given condition variable specified by cond; the object becomes, in effect, uninitialized.

Find out more on titan: `$man pthread_cond_destroy`

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

blocks the current thread (it sits there waiting) until the function returns.

Upon return, either the condition has been satisfied (have a global or shared variable that represents this) or you would call this function again.

Hint: the mutex must be locked when passed in, it is a bug for this not to be the case.

Find out more on titan: `$man pthread_cond_wait`

```
int pthread_cond_signal(pthread_cond_t *cond);
```

sends a signal to one of the waiting threads that it can wake up and continue with the work as the condition waited upon has been satisfied.

Find out more on titan: `$man pthread_cond_signal`