UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Faculty of Engineering, Built Environment and Information Technology

Fakulteit Ingenieurswese, Bou-omgewing en Inligtingtegnologie / Lefapha la Boetšenere, Tikologo ya Kago le Theknolotši ya Tshedimošo

Department of Computer Science

COS 122 Operating Systems

Practical 1

Due: 09 September 2025 @ 23:00 PM

PLAGIARISM POLICY

UNIVERSITY OF PRETORIA

The Department of Computer Science considers plagiarism as a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent) and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to `https://www.library.up.ac.za/plagiarism` (from the main page of the University of Pretoria site, follow the *Library* quick link, and then *Services* and then click the *Plagiarism* link). If you have any form of question regarding this, please ask one of the lecturers, to avoid any misunderstanding. Also note that the OOP principle of code re-use does not mean that you should copy and adapt code to suit your solution. The use of generative AI such as ChatGPT is prohibited within the context of this practical and will result in plagiarism charges.

Objectives

This individual practical explores the concepts of process and threads and their differences using the C++ programming language. These concepts are covered in chapters 3, 4 and 5 of the prescribed textbook. This practical has 2 tasks for a total of 255 marks.

Upload Instructions

You need to write C++ programs for each of the tasks. You are then required to submit your code on Fitchfork where it will be marked automatically.

- Upload each of your tasks in a single zip or tar.gz file to the respective Practical 1 assignment slot on Fitchfork before 23:00 on 09-Sep-2025. **No late submissions will be accepted!**
- Make use of a Makefile for your programs.
- Your Makefile needs to use the `g++` compiler.
- Your Makefile **must** have at least these commands:
  - `# make`
  - `# make run`
- Only upload your source code (.h and .cpp files) and makefiles. Do not upload large, binary or OS files. Keep your uploaded archive small.
- **Failure to upload your work will result in 0 marks being awarded for your practical!**

General Instructions

For this practical, you will need to use some kind of C++ development environment. You can for example use the Linux machines in the Informatorium or you can set up a Linux/WSL environment on your own machine (if you don't already have one). See the additional documentation for various possible options.

- You are not allowed to use external libraries. Only standard C++ libraries may be imported.
- Every file you submit should contain your student number at the top of the file (source code and makefiles).
- If your program does not have a Makefile you will get 0 marks.
- If your program doesn't compile due to syntax errors you will get 0 marks.

- If your program receives a runtime error you will lose marks.
- We will be overriding your .h files, therefore do not add anything extra to them that is not in the spec.

C++11

C++ native threads were only introduced in the C++11 specification. Therefore, you will need to compile your code with the c++11 flag when using threads (`-std=c++11`). For all your other code, you should still stick to using the C++98 standard notation.

Useful Links

Linux man pages

`https://www.man7.org/linux/man-pages/man1/ps.1.html`
`https://www.man7.org/linux/man-pages/man2/fork.2.html`
`https://www.man7.org/linux/man-pages/man2/wait.2.html`
`https://www.man7.org/linux/man-pages/man3/sleep.3.html`

C++ reference

Threads: `https://cplusplus.com/reference/thread/thread/`
Files: `https://cplusplus.com/reference/fstream/ifstream/`
Strings: `https://cplusplus.com/reference/string/string/`
Mutex: `https://cplusplus.com/reference/mutex/`

Compiling

`https://courses.engr.illinois.edu/cs225/fa2018/resources/maketutorial/`

Task 1 - Processes and Threads [102 Marks]

In this task you are required to perform matrix multiplication using two different approaches: processes and threads. The objective is to see the practical implications of the differences between those two approaches.

Background Concepts

What is a matrix?

A matrix is a rectangular arrangement of numbers (or other mathematical objects) in rows and columns. It is usually denoted using uppercase letters, e.g., A, B, C.

Example:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Here:

- A has 3 rows and 3 columns (a 3×3 matrix).
- The element in the 1st row, 2nd column is $A_{1,2} = 2$.

Terminology:

- **Row:** Horizontal line of elements.
- **Column:** Vertical line of elements.
- **Dimension:** Described as $m \times n$ (m rows, n columns).

What is Matrix Multiplication?

Matrix multiplication is an operation that takes two matrices and produces a new matrix.

Multiplication Rule: If A is an $m \times n$ matrix and B is a $p \times q$ matrix, then multiplication is valid only if $n = p$. The result will be an $m \times q$ matrix.

How it Works: If $C = A \times B$, then each element $C_{i,j}$ is calculated as:

$$C_{i,j} = \sum_{k=1}^{n} (A_{i,k} \times B_{k,j})$$

This means:

1. Take row i of A.
2. Take column j of B.
3. Multiply corresponding elements.
4. Add them together.

Example: Let:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Calculations:

$$C_{1,1} = (1 \times 5) + (2 \times 7) = 19$$
$$C_{1,2} = (1 \times 6) + (2 \times 8) = 22$$
$$C_{2,1} = (3 \times 5) + (4 \times 7) = 43$$
$$C_{2,2} = (3 \times 6) + (4 \times 8) = 50$$

Result:

$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Matrix Implementation

You have been provided with a partially implemented Matrix class defined in `Matrix.h`. In this section, the remaining functions that you are required to implement in `Matrix.cpp` are described in detail. For every function, you must follow the described behaviour exactly to ensure that your program works as expected.

Private Member Variables

`int size`

- **Description:** Stores the dimension of the square matrix. For an $n \times n$ matrix, this variable holds the value n.

`int** data`

- **Description:** A pointer to a dynamically allocated 2D array of integers that stores the matrix elements.
- **Structure:** Implemented as a pointer to an array of `int*`, where each `int*` points to an array representing one row of the matrix.

`int Matrix::getSize() const`

- Returns the size member variable.

`bool Matrix::loadFromFile(const char* filename)`

- **Purpose:** Reads a square matrix from a given text file and stores it in the `Matrix` object.
- **Parameters:**
  - `filename` – a C-style string containing the path to the file to be read.
- **Expected Behaviour:**
  1. Open the specified file.
  2. Read all rows and determine the number of columns.
  3. Ensure the matrix is square (rows = columns).
  4. Allocate memory for `data` accordingly.
  5. Store each value into the matrix's `data` array.
  6. Close the file and return `true`.
- **Error Handling:** If the file cannot be opened, or if the matrix is not square, display an error message and return `false`.
- **Notes:** The file format is expected to be space-separated integers, one row per line. Empty lines should be ignored. See the included text files.

`int Matrix::getElement(int row, int col) const`

- **Purpose:** Retrieves the value stored at the given `row` and `col` position in the matrix.
- **Parameters:**
  - `row` – the row index (0-based).
  - `col` – the column index (0-based).
- **Return Value:** The integer value stored at the specified position.
- **Error Handling:** If the indices are out of bounds or the matrix data is not allocated, return 0.
- **Notes:** This function is `const`, meaning it should not modify the matrix.

`void Matrix::setElement(int row, int col, int value)`

- **Purpose:** Updates the value stored at the given `row` and `col` position in the matrix.
- **Parameters:**
  - `row` – the row index (0-based).

  - `col` – the column index (0-based).
  - `value` – the new integer value to be stored.

- **Expected Behaviour:** Replace the existing value at the given position with the new value.
- **Error Handling:** If the indices are out of bounds or the matrix data is not allocated, the function should do nothing.

`void Matrix::matrixCalculator(const Matrix& matrix1, const Matrix& matrix2, int rowIndex)`

- **Purpose:** Calculates a single row of the result matrix for the multiplication `matrix1` × `matrix2`.
- **Parameters:**
  - `matrix1` – the left-hand side operand.
  - `matrix2` – the right-hand side operand.
  - `rowIndex` – the index of the row in the result matrix to compute.
- **Expected Behaviour:** For each column j in `matrix2`:

  $$C_{\texttt{rowIndex},j} = \sum_{k=0}^{n-1} (\texttt{matrix1.getElement(rowIndex, k)} \times \texttt{matrix2.getElement(k, j)})$$

  Store the result using `setElement`.
- **Notes:** This function only computes one row at a time, allowing it to be used in both thread-based and process-based parallel implementations.

`void threadWorker(const Matrix& matrix1, const Matrix& matrix2, Matrix& result, int rowIndex, std::mutex* printMutex)`

- **Purpose:** A helper function used when implementing the multithreaded matrix multiplication.
- **Parameters:**
  - `matrix1` – the left-hand side operand.
  - `matrix2` – the right-hand side operand.
  - `result` – the matrix where the computed values should be stored.
  - `rowIndex` – the index of the row to compute.
  - `printMutex` – a pointer to a mutex used to protect console output.
- **Expected Behaviour:**
  1. Lock the mutex and print a message indicating that the thread has started.

2. Unlock the mutex.
3. Call `result.matrixCalculator(matrix1, matrix2, rowIndex)` to compute the row.
4. Lock the mutex again and print a message indicating that the thread has completed.
5. Unlock the mutex.

- **Notes:** Only printing operations should be protected by the mutex to avoid unnecessary locking during computation.

Task 1.1 [51 Marks]

Implement matrix multiplication using multiple processes (via `fork`) in a file called `task1_processes.cpp`. Your implementation should load two matrices from individual text files and check that they are square and their dimensions allow multiplication. The parent process then needs to fork child processes such that each child process computes one row of the result matrix. Since processes do not share memory, they each need to write their results to a temporary file on secondary storage (a shared resource). The parent process waits for all child processes to complete and then reads all the temporary files to assemble the final result. It then prints the resulting matrix with it's size. The temporary files should be deleted after use by the parent process.

Each child process should print a message when it begins and when it completes its computation. The message should include the process id (`PID`) and the row number being calculated, as shown in the sample output:

```
Process 579465 calculating row 0
Process 579466 calculating row 1
Process 579465 completed row 0
Process 579467 calculating row 2
Process 579466 completed row 1
Process 579467 completed row 2
```

Figure 1: Example output during process execution

A function called `process_printStatus(pid, rowIndex, complete)` for printing this process message has been provided in the file called `task1_printstatus.cpp`.

Implementation

The following steps show how your function should work:

1. Load two input matrices from the given text files.

2. Verify that both matrices are square and of the same size. If they are not compatible for multiplication, print an appropriate error message and terminate.
3. For each row of the first matrix:
   (a) Create a new child process using `fork()`.
   (b) Each child process is responsible for computing the corresponding row of the result matrix.
   (c) Each child process then:
      - Prints a message indicating the process id and row number when it starts computing its row.
      - Computes all entries in the corresponding row of the result matrix.
      - Prints another similar message when it has completing its computation.
   (d) The child process writes the computed row to a temporary file and then terminates.
4. The parent process waits for all child processes to complete using `wait()`.
5. The parent process then:
   (a) Reads all the temporary files and assembles the final result matrix.
   (b) Deletes the temporary files after reading their contents.
   (c) Prints the resulting matrix size (e.g. `N x N`).
   (d) Prints the full result matrix values (using the provided `print()` function).

Function to Implement

You are required to implement the following function in `task1_processes.cpp`:

```
void task1_processes(const char* matrixFile1, const char* matrixFile2);
```

Expected Console Output

When running your implementation, the output should clearly show:

- The start of process-based matrix multiplication.
- For each process: its unique `PID` and assigned row number when it starts calculating, and when it finishes.
- The resulting matrix size.
- The final calculated matrix.

An example of what the program should display (using parameter 1 for this subtask):

```
./matrix_calculator 1
Matrix Multiplication Calculator
================================

=== TASK 1.1: PROCESS-BASED MATRIX MULTIPLICATION ===

Starting process-based matrix multiplication:
Process 606539 calculating row 0
Process 606540 calculating row 1
Process 606539 completed row 0
Process 606540 completed row 1
Process 606541 calculating row 2
Process 606541 completed row 2

Resulting Matrix (Process-based) size: 3x3
129 62 68
79 32 19
88 78 55
```

Figure 2: Example run using processes

Task 1.2 [51 Marks]

Implement matrix multiplication using multiple threads (via `std::thread`). Your implementation should load two matrices from individual text files and check that they are square and their dimensions allow multiplication. The main thread then starts worker threads such that each thread computes one row of the result matrix. The main thread then waits for all worker threads to complete, and then assembles the final result. It then prints the resulting matrix with it's size.

Each thread should print a message when it begins and when it completes its computation. The message should include the thread id (`thread::id`) and the row number being calculated, as shown in the sample output:

```
Thread 140466678986304 calculating row 0
Thread 140466678986304 completed row 0
Thread 140466670593600 calculating row 1
Thread 140466670593600 completed row 1
Thread 140466662200896 calculating row 2
Thread 140466662200896 completed row 2
```

Figure 3: Example output during thread execution

A function called `thread_printStatus(tid, rowIndex, complete)` for printing this thread message has been provided in the file called `task1_printstatus.cpp`.

Implementation

The following steps show how your function should work:

1. Load two input matrices from the given input text files.
2. Verify that both matrices are square and of the same size. If they are not compatible for multiplication, print an appropriate error message and terminate.
3. For each row of the first matrix:
   (a) Create a new worker thread using `std::thread()`
   (b) Each worker thread is responsible for computing the corresponding row of the result matrix.
   (c) Each worker thread then:
      - Prints a message indicating the thread id and row number when it starts computing its row.
      - Computes all entries in the corresponding row of the result matrix.
      - Prints another similar message when it has completed its computation.
      - Use a `std::mutex` to ensure synchronised console output.
4. The main thread waits for all worker threads to complete using `join()`.
5. The main thread then:
   (a) Prints the resulting matrix size (e.g. `N x N`).
   (b) Prints the full result matrix values (using the provided `print()` function).

Function to Implement

You are required to implement the following function in `task1_threads.cpp`:

```
void task1_threads(const char* matrixFile1, const char* matrixFile2);
```

Expected Console Output

When running your implementation, the output should clearly show:

- The start of thread-based matrix multiplication.
- For each thread: its unique `thread::id` and assigned row number when it starts calculating, and when it finishes.
- The resulting matrix size.
- The final calculated matrix.

An example of what the program should display (using parameter 2 for this subtask):

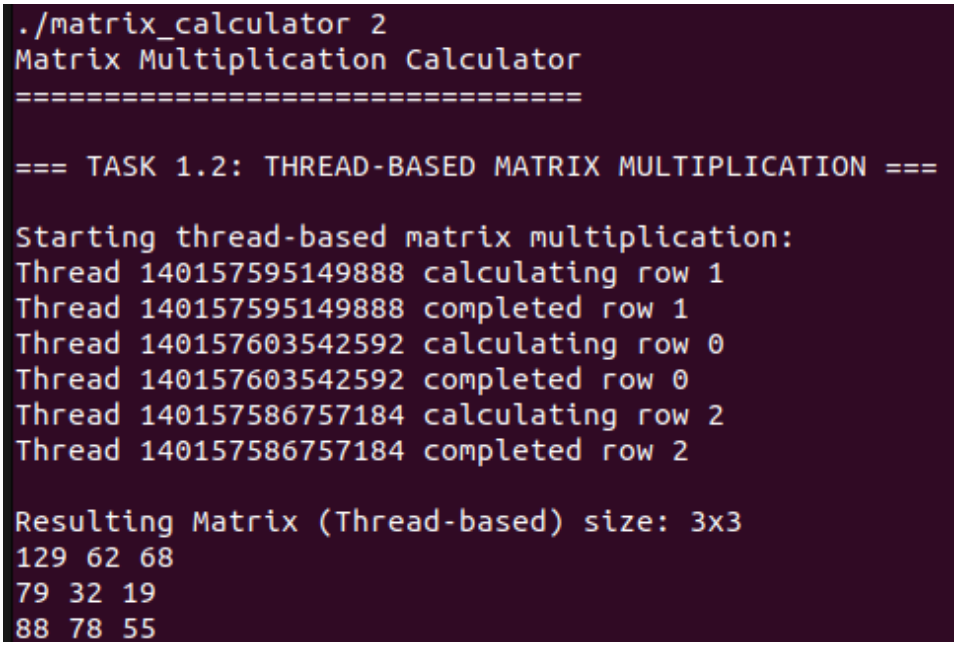

Figure 4: Example run using threads

Important!

- You should use the specified C++ program and function names.
- A main file and matrix input text files have been provided for you to test your output. **Make use of these.**
- Make sure you test your program for various different sized square matrices.
- You have to create and use a makefile.
- Make sure that your output strictly matches the example output. There should be no extra spaces or new lines.
- `main.cpp`, `Matrix.h` and `task1_printstatus.cpp` will be overwritten, do not include implementation specific code in it.

Upload Archive

Make sure you submit an archive (zip or tar.gz) to **Fitchfork** containing the following files:

- main.cpp
- Matrix.cpp
- Matrix.h
- task1_processes.cpp

- task1_threads.cpp
- task1_printstatus.cpp
- makefile

Task 2 - Concurrent Sorting [135 marks]

For this task you will be required to sort an array in **ascending** order of any given size in a multi-threaded environment without the use of any locks. The key here is to make sure that different threads do not access the same index at the same time to ensure mutual exclusion. The steps to achieve this are illustrated below:

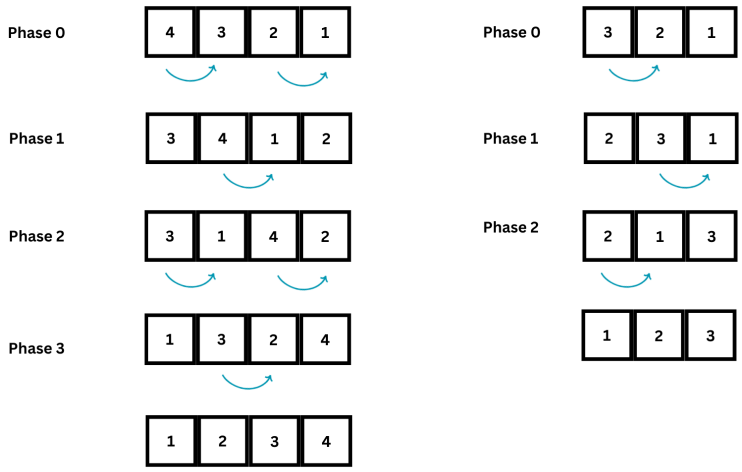

Figure 5: 2 cases (even and odd length) of the process of concurrent sorting by continuously swapping indices

Each blue arrow represents a single thread, pay attention to the distinct indices being accessed by each thread. On the left is an array of even length and on the right is an array of odd length.

Concurrent Sorting Implementation

The file `ConcurrentSorting.h` declares all the necessary functions and global variables needed to implement this task. **Note:** This file will be overwritten. So do not declare any additional helper functions or variables in it. Rather do that in your implementation file.

For simplicity and ease of use we will use a vector, part of the std namespace and standard C++ library, in place of a traditional array, since vectors have many useful built-in functions and most importantly allow for dynamic resizing (which will be happening a lot during the sorting process).

The following is given and **must not** be changed:

- `std::vector<int> buffer;`
  Description: This is the shared array that will store the integers being sorted.

- `std::vector<std::thread> threadPool;`
  Description: This array acts as a pool and will store all current threads that are in need.

- `std::vector<int> indices;`
  Description: This array stores the indices that are to be swapped in a phase. (Hint: refer to Figure 1).

- `int currentPhase;`
  Description: Stores the number of the current phase. eg 0, 1, 2.

- `int totalSwaps;`
  Description: Keeps track of the total amount of swaps taken place in the whole sorting process.

- `int totalThreadsUsed;`
  Description: Keeps track of the total amount of threads used in the whole sorting process.

- `std::string buffertoString();`
  Purpose: Converts the array contents into specific string format. **Returns:** `std::string`

Your task

You will need to implement the following functions in `ConcurrentSorting.cpp`:

- `void initBuffer(std::vector<int> &vec);`
  Parameters: `std::vector<int> vec` - the passed in vector to which buffer will be set equal to.
  Purpose: Sets buffer equal to the passed in vector vec.

- `bool isSorted();`
  Purpose: Returns `true` if the buffer is sorted and `false` if it is not sorted.

- `int maxPhasesExpected();`
  Purpose: Determine and return the **maximum** amount of phases expected for a given array/buffer size.
  Return: int

- `int maxSwapsExpected();`
  Purpose: Determine and return the **maximum** amount of swaps expected for a given array/buffer size.
  Return: int

- `void setIndices();`
  Purpose: Determines the indices needed to be swapped in a single phase, and stores them in `std::vector<int> indices`.

- `void swap(int index1, int index2);`
  Parameters:
  - index1 - index in buffer that is being considered for swapping.
  - index2 - index in the buffer that is being considered for swapping (hint : index2 = index1 + 1).

  Purpose: Swaps the values in buffer at index1 with the value in buffer at index2 (hint : we are sorting in ascending order). You will also need to print to screen in this format (you are required to research how to obtain a thread's ID in c++):

  `Thread <thread id> is using indices <index1> and <index2>`

  If there is going to be swap you will also need to print:

  `Thread <thread id> is swapping <buffer[index1]> and <buffer[index2]>`

  An example is given below:

```
1   Thread 140254824617536 is using indices 1 and 2
2   Thread 140254833010240 is using indices 3 and 4
3   Thread 140254833010240 is swapping -9 and -46
4   Thread 140254841402944 is using indices 5 and 6
```

Figure 6: Example console output from swap function

Note : it is only necessary to print line 3 if there is going to be a swap

- `void clearThreadPool();`
  Purpose: This will clear the vector called threadPool.

- `void SetAndRunThreads();`
  Purpose: This will assign each thread its task and then run each thread. You will also need to print:
  `<currentAmountOfThreads> threads about to execute`
  before actually running the threads, eg:
  `3 threads about to execute`

- `void concurrentSort();`

  Purpose: This is where the sorting process will take place. You will need to call the necessary functions in the correct order to make the necessary swaps per phase.

Algorithm 1 General Algorithm

1: **while** buffer is not sorted **do**
2: Determine and set indices
3: Print "Current phase : `<currentPhase>`"
4: Setup and run threads
5: Print "Phase results = `<buffertoString>`"
6: Print "Is sorted = `<Yes> or <No>`"
7: **end while**

```
Current phase : 1
3 threads about to execute
Thread 138565105206848 is using indices 1 and 2
Thread 138565105206848 is swapping 8 and 5
Thread 138565113599552 is using indices 3 and 4
Thread 138565113599552 is swapping 6 and 3
Thread 138565121992256 is using indices 5 and 6
Thread 138565121992256 is swapping 4 and 1
Phase results = | 7 | 5 | 8 | 3 | 6 | 1 | 4 | 2 |
Is sorted = No
```

Figure 7: Example output from single phase execution

- `void printResults();`

  Purpose: This must print the final results of the sorting process in this format:

  ```
  Post sort results :
  Total threads used = <totalThreadsUsed>
  Maximum swaps expected = <maxSwapsExpected>
  Total swaps = <totalSwaps>
  Maximum phases expected = <maxPhasesExpected>
  Amount of phases entered = <currentPhase>
  After = <buffertoString>
  ```

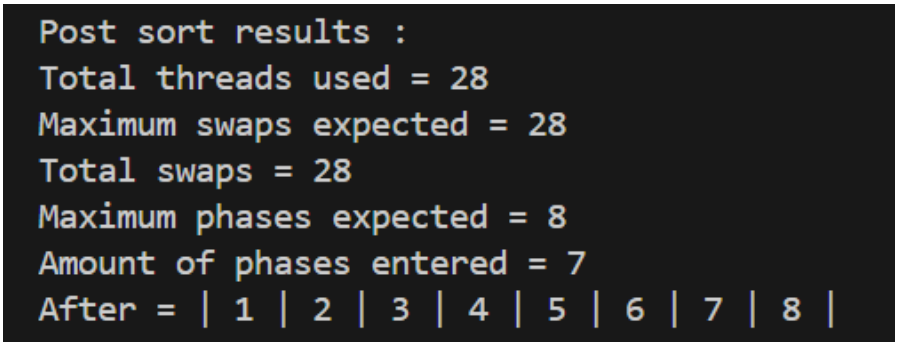

Figure 8: Example of printResults()

Important!

- You should use the specified C++ program file names.
- You must ensure thread safety where shared resources are used.
- Make sure that your output strictly matches the example output. There should be no extra spaces or new lines.
- `main.cpp` and `ConcurrentSorting.h` will be overwritten, do not include implementation specific code in it.

Upload Archive

You will need to submit the following files as an archive (zip or tar.gz) to **FitchFork**.

- main.cpp
- ConcurrentSorting.h
- ConcurrentSorting.cpp
- makefile