Assignment 1

Introduction

In Assignment 1, students will learn to use system calls.

System Calls

Manage Files

```
int open(const char *pathname, int flags, ...
                 /* mode_t mode */ );

ssize_t read(int fd, void buf[.count], size_t count);

ssize_t write(int fd, const void buf[.count], size_t count);
```

open()

The **open()** system call opens the file specified by pathname. If the specified file does not exist, it may optionally (if O_CREAT is specified in flags) be created by open().

The return value of open() is a file descriptor, a small, nonnegative integer that is an index to an entry in the process's table of open file descriptors. The file descriptor is used in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.) to refer to the open file. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

read()

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.
On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and **read**() returns zero.
If *count* is zero, **read**() *may* detect the errors described below. In the absence of any errors, or if **read**() does not check for errors, a **read**() with a *count* of 0 returns zero and has no other effects.

write()

write() writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.
The number of bytes written may be less than *count* if, for example, there is insufficient space on the underlying physical medium, or the **RLIMIT_FSIZE** resource limit is encountered (seesetrlimit(2)), or the call was interrupted by a signal handler after having written less than *count* bytes. (See also pipe(7).)

For a seekable file (i.e., one to which lseek(2) may be applied, for example, a regular file) writing takes place at the file offset, and the file offset is incremented by the number of bytes actually written. If the file was open(2)ed with **O_APPEND**, the file offset is first set to the end of the file before writing.
The adjustment of the file offset and the write operation are performed as an atomic step.

Map files

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in *addr*. The *length* argument specifies the length of the mapping (which must be greater than 0).

For more details, see mmap(2) - Linux manual page

```
void *mmap(void addr[.length], size_t length, int prot, int flags,int fd, off_t
 offset);
int munmap(void addr[.length], size_t length);
```

Process

```
pid_t fork(void);

pid_t wait(int *_Nullable wstatus);
pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

fork()

fork() creates a new process by duplicating the calling process.
The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.
The child process and the parent process run in separate memory spaces. At the time of **fork**() both memory spaces have the same content.

Memory writes, file mappings (mmap(2)), and unmappings (munmap(2)) performed by one of the processes do not affect the other.
The child process is an exact duplicate of the parent process except for the following points:

- The child has its own unique process ID, and this PID does not match the ID of any existing process group (setpgid(2)) or session.
- The child's parent process ID is the same as the parent's process ID.
- The child does not inherit its parent's memory locks (mlock(2), mlockall(2)).

For more details, see fork(2) - Linux manual page

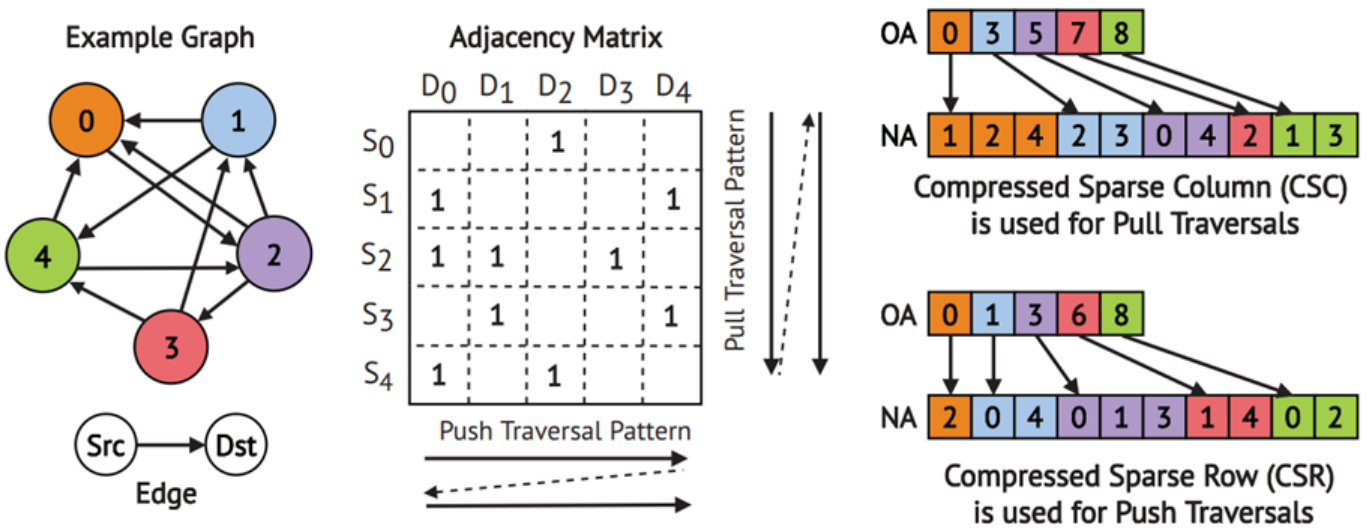
wait()

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below).

If a child has already changed state, then these calls return immediately. Otherwise, they block until either a child changes state or a signal handler interrupts the call (assuming that system calls are not automatically restarted using the **SA_RESTART** flag of sigaction(2)). In the remainder of this page, a child whose state has changed and which has not yet been waited upon by one of these system calls is termed *waitable*.

Graph Preleminary

Graph Layout



`OA` : Offset Array.

`NA` : Neighbor Array.

In our implementation, the graph is in CSR format. Which means that, if you want to know the neighbor of `node i` , get `OA[i]` and `OA[i+1]` , the index range of `NA` that stores neighbors.

In the `graph.size` file, it has 8 bytes, which are the number of nodes and number of edges. (`|nodes|` , `|edges|`)

In the `graph` file, the first `|nodes|+1` *4 Bytes are values of `OA` . The following `|edges|` *4 Bytes are values of `NA` . The last `|edges|` *4 Bytes are values of edge weight for corresponding edge.

Dataset Download

You can get dataset, and sample output of Assignment 1 from the following link:

CSC3150_23-24Term2_A1_data - OneDrive

Your dataset should be placed at the same directory of code

```
main@ubuntu:~/Desktop$ ls
a.out  grader  g1  g1.size  g2  g2.size  graph.cpp  graph.h  sample_output
```

Important

In this assignment, all datasets to read and output files you need to write are in binary format. Please take a look at `std::ios::binary` .

sample_output is used for self-checking if the answers to task1 are correct.

Task1

In task1, you are provided with data sets `graph1` , `graph2` correspondings to `Task1_1` and `Task1_2` . In such graphs, the nodes can be represented as cities, edges between cities can be represented as roads. We provide a program traversing nodes through BFS algorithm.

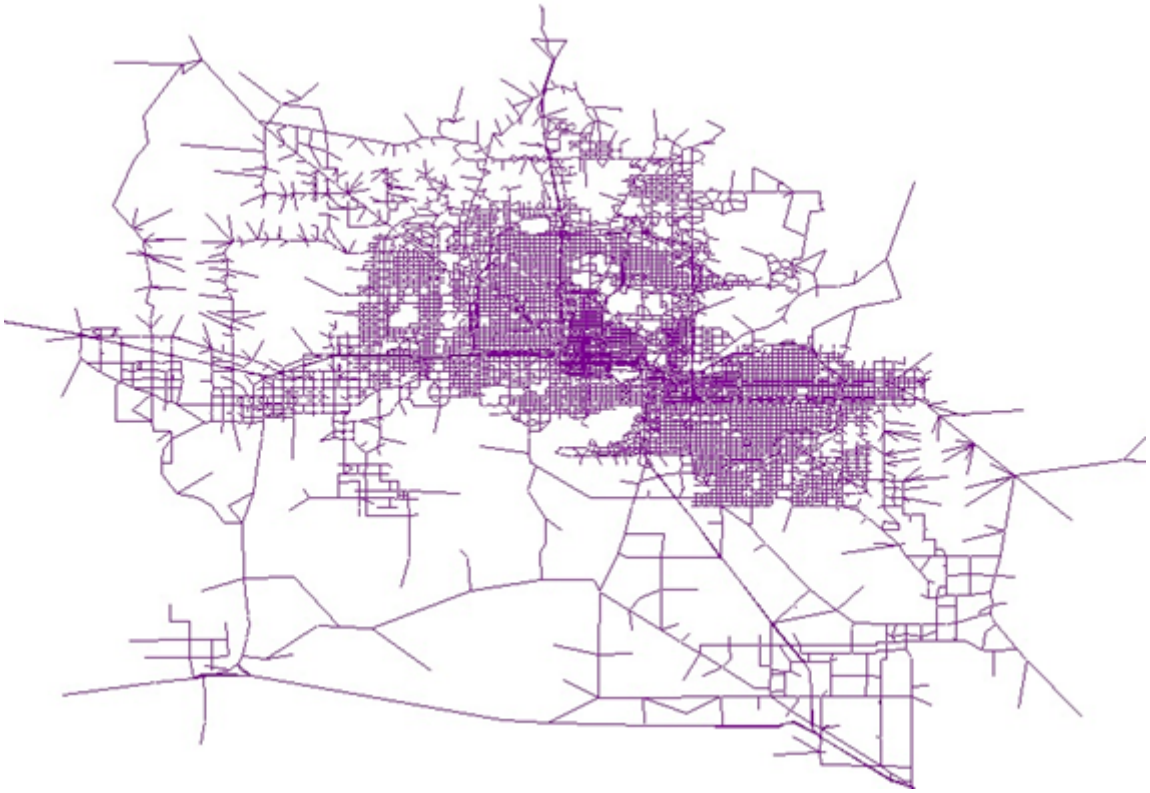
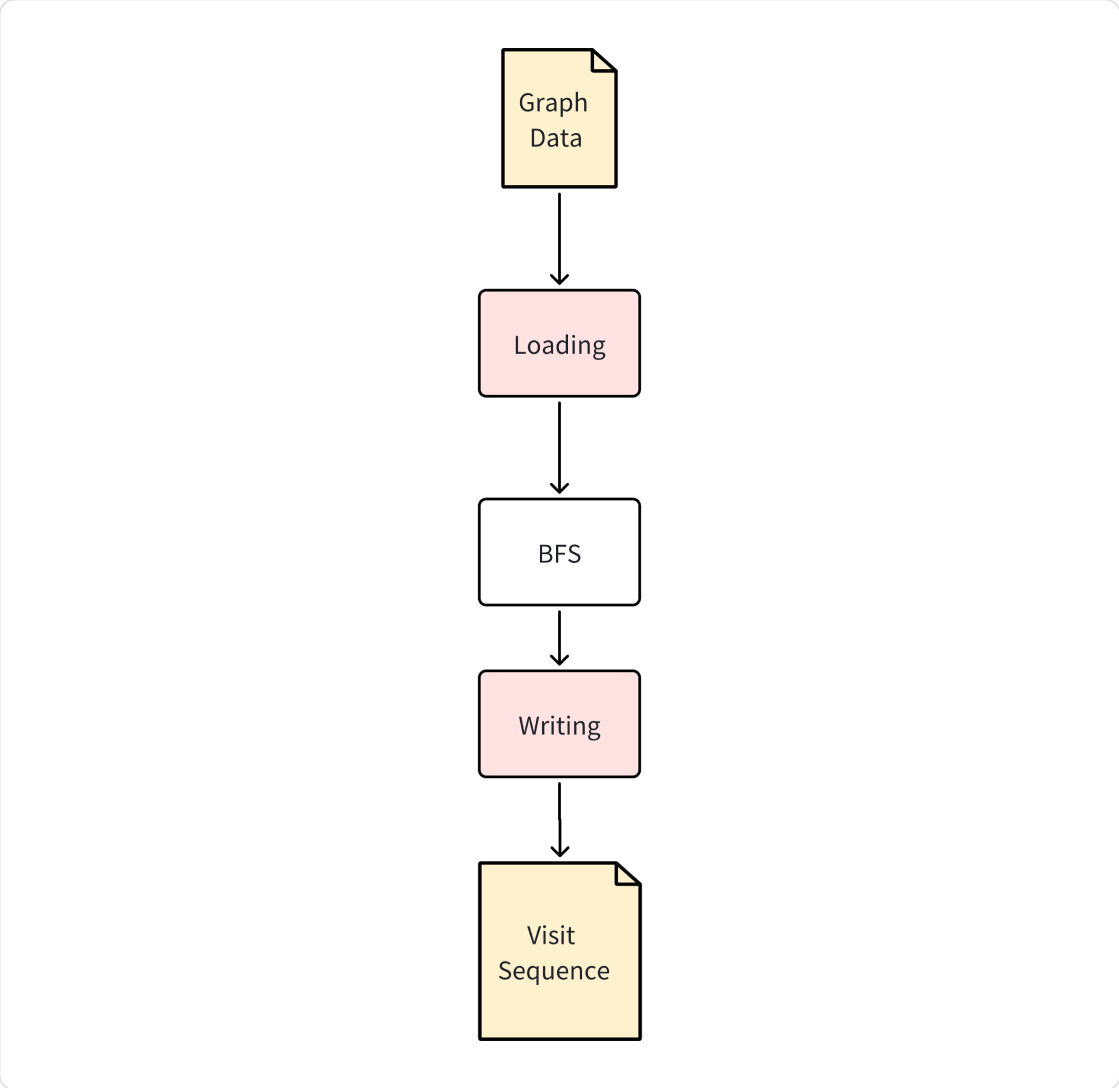


Figure: Transportation Graph

- Your first task is to complete the remaining part of program so that it can correctly load data into memory and do BFS traverse execution.
- After BFS traversal is done, the program needs to store the sequence of visits to the result file.
- In the following progress charts, blocks in **yellow** means it's a file, in **red** means it's a **TODO** part for you.
- In this task, you need to complete `Task1_1(), Task1_2()`. In `Task1_2()` we only have 2GB memory but `g2` has 1.8GB data. Traditional read from disk to memory does not work.



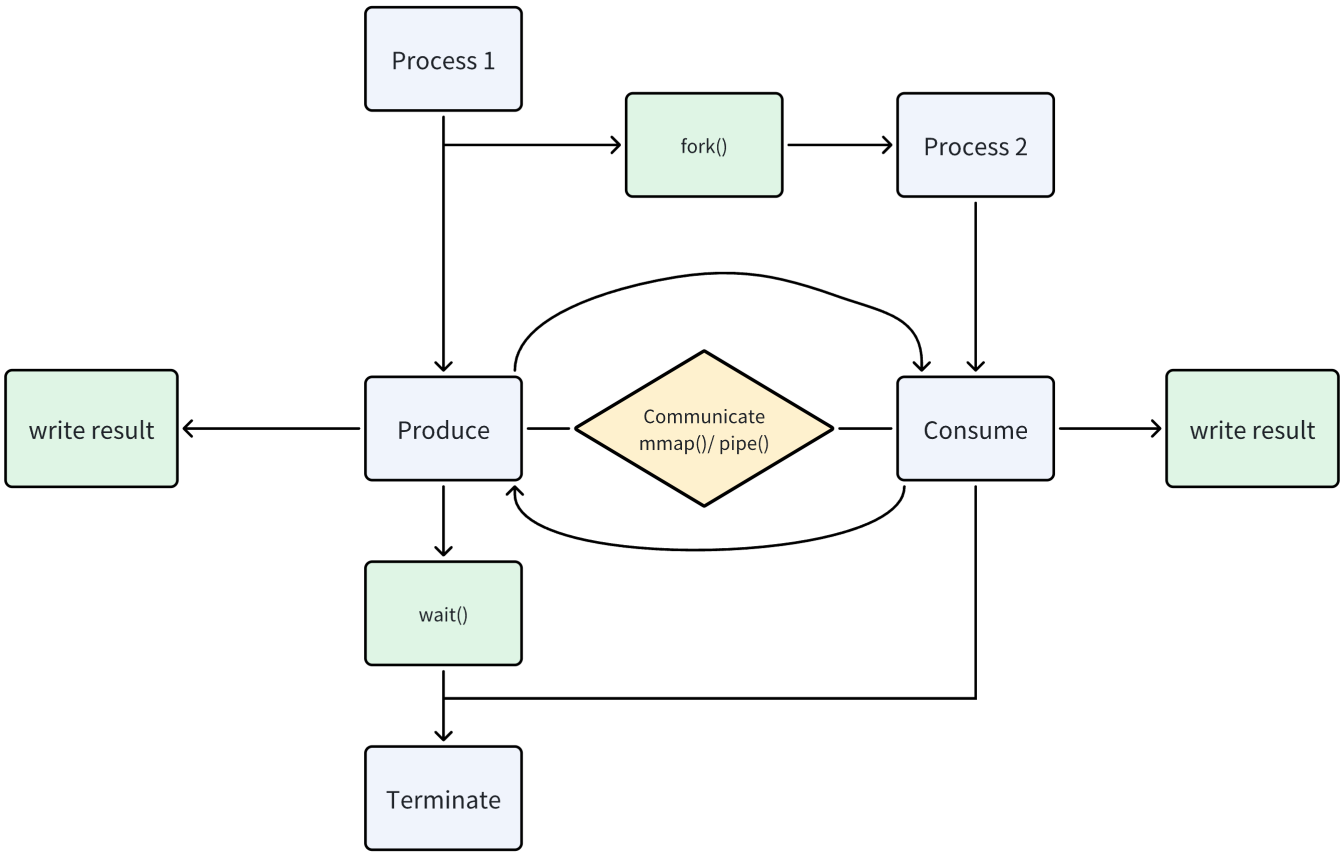
Hint

- When loading the graph, please note that `OA` has length `|nodes|+1` instead of `|nodes|`. This is because we miantain one more integer for some boundary problems. What you read from the `.size` file is `|nodes|` and `|edges|`.

Task2

In task 2, the program goes into a simple simulation. In the daytime, a node gains weights from its edges with corresponding edge weights, implemented by `produce()`. In the night, a node's weight will self-increased by 1, implemented by `consume()`.

The global view of process is shown in the figure below. We highlight the components you needs to do in green and yellow blocks:



You are required to use `fork()` to handle two processes. For the parent process, we will do calculation of `produce()`. For the child process, we will do `consume()`. Both of these functions are provided, you do not need to do any modification to it.

```
// Executed in parent process
void produce(int *weights, int len, Graph &g) {
    for (int i=0; i<len; i++) {
        int l,r;
        g.get_edge_index(i,l,r);
        for (int j=l;j<r;j++) {
            weights[i+1]+=g.edge_weights[j];
        }
    }
}


// Executed in child process
void consume(int *weights, int len, Graph &g) {
    for (int i=0; i<len; i++) {
        weights[i+1] += 1;
    }
}
```

TODO

- Your task is to complete the following functions with correct system call and control logic so that program runs correctly as shown in the above figrue.
- In implementation of communication, it actually shares the data stored in `weights`
  - Format of array `weights` : the first integer should be the value of current iteration (start from 0). The following `|V|` integers are corresponding weights of vertex
- For each iteration generated by child process, child process should write result to file

  e.g. For 4th and 5th iterations, write sequence of nodes weight to `/home/csc3150/A1/Task2_parent.output_2` and `/home/csc3150/A1/Task2_child.output_2`.
- Program terminates at 10th iteration

```
// Task2: Process Inter-Communicationvoid Task2() {
    Graph g;
    int fd;
    fd = g.map_global_graph("g2");
    std::string ipc_path("ipc_file");
    // Creating inter process communication file if there is not.
    int ipc_fd = open(ipc_path.c_str(),O_RDWR| O_CREAT,0777);
    lseek (ipc_fd, (g.v_cnt+1)*sizeof(int)-1, SEEK_SET);
    write (ipc_fd, "", 1);
    close(ipc_fd);
    std::string output_parent_path("Task2_parent.output");
    std::string output_child_path("Task2_child.output");
    // process control
}

// Task2: Process Inter-Communication with control
void parent_process(const std::string &path) {
    int pid = getpid();
    printf("parent proc %d is producing\n",pid);

    // produce()
    return;
}

// Task2: Process Inter-Communication with control
void child_process(const std::string &path) {
    int pid = getpid();
```

```
        printf("child proc %d is consuming\n",pid);

        // consume()
        return;
    }
```

Hint

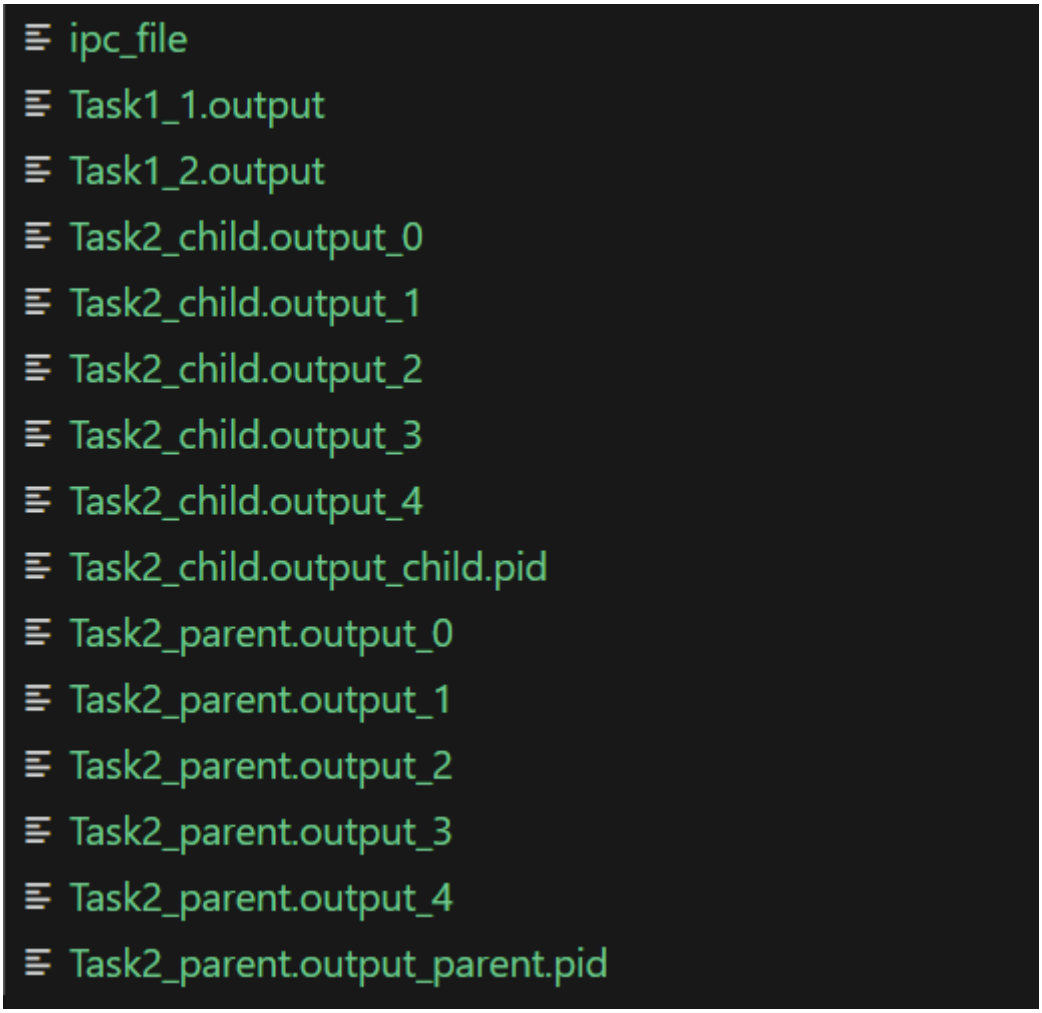
- The status of vertices need to be communicated inter-process at each iteration because both processes need to process/ fetch data.
- You might need to define `index 0` of mapped file in mmap() as control `int`
    - Because of Data Dependency, execution order matters
    - By reading control byte, each process will know whether they should wait, process, or terminate.
- You can use mmap() to share reading the data graph if both processes need information of graph.
    - Try to figure out how memory is consumed when multi-process using mmap map to one file.

Output files

You are expected to generate the following files when you complete all the tasks

All these files are in the same directory as ***graph.h, graph.c***

(wrong file name will not be correctly graded):



Report 10'

You shall strictly follow **the provided latex template** for the report, where we have emphasized important parts and respective grading details. **Reports based on other templates will not be graded**.

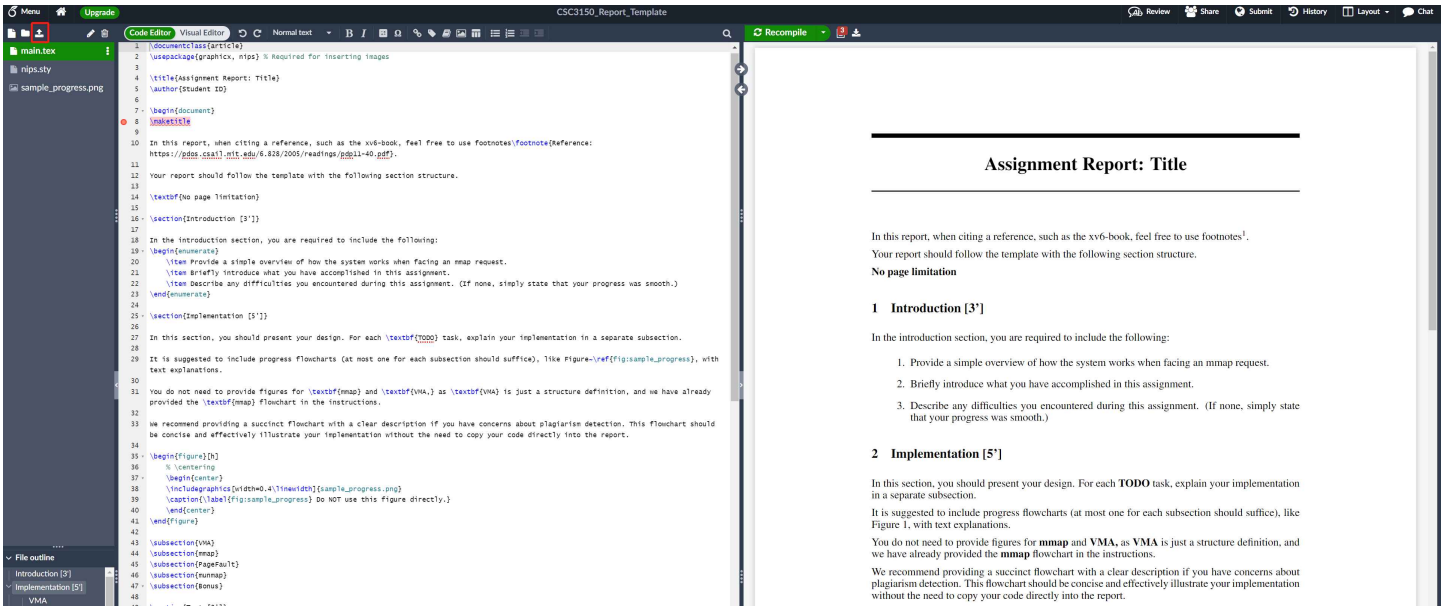
Report Template

- You can find latex template in the following link
  - https://www.overleaf.com/read/ybgnjwnyvjpx#dd17ed
- We also provide template at BB.

LaTex Editor

For your convenience, you might use Overleaf, an online LaTex Editor.

1. Create a new blank project.
2. Click the following highlight bottom and upload the template we provide.
3. Click **Recompile** and you will see your report in PDF format.



Extra Credit

Instead of using shared memory space for inter-process communication, you are required to implement Task2 with using message passing.

Using any profiling tools to compare performance of communication via shared memory (mmap) and message passing.

Report your result and your discovery.

Hint

- You can use pipe() to pass updated messages from parent process to child process/ or from child process to parent process.
- You can learn to use dtrace for profiling in this task.

Execution

```
main@ubuntu:~/Desktop$ g++ graph.cpp
main@ubuntu:~/Desktop$ ./a.out
```

Submission

- **Due on: 23:59, 28 Feb 2023**
- **Plagiarism is strictly forbidden.** Please note that TAs may ask you to explain the meaning of your program to ensure that the codes are indeed written by yourself. Please also note that we would check whether your program is too similar to your fellow students' code and solutions available on the internet using plagiarism detectors.

- Late submission: A late submission **within 15 minutes** will not induce any penalty on your grades. But **00:16 am-1:00 am: Reduced by 10%; 1:01 am-2:00 am: Reduced by 20%; 2:01 am-3:00 am: Reduced by 30% and so on.** (e.g., Li Hua submitted a perfect attempt at 2:10 a.m. He will get (100+10 (bonus))*0.7=77p
- Any incorrect execution results by the following reasons are not arguable (Please double check before submit)
  - Compile error (Ensure no modification on header file)
  - Incorrect file name/ path (Please strictly follow naming and path provided in instruction and template)
  - Successfully running program with incorrect result

Format guide

The project structure is illustrated below. You can also use `ls` command to check if your structure is fine. Structure mismatch would cause grade deduction.

For this assignment, you don't need a specific folder for the bonus part. The source folder should contain four files: ***graph.c, report.pdf .*** Please note that ***graph.h*** is ***not*** need to submit.

We limit your implementation within ***graph.c*** file, starting with "Task:" comments.

```
main@ubuntu:~/Desktop/Assignment_3_120010001$ ls
report.pdf  graph.c

(One directory and one pdf.)
```

Please compress all files in the file structure root folder into a single zip file and **name it using your student ID as the code shown below and above, for example, Assignment_3_120010001.zip.** The report should be submitted in the format of **pdf**, together with your source code. Format mismatch would cause grade deduction. Here is the sample step for compressing your code.

```
main@ubuntu:~/Desktop$
 zip -q -r Assignment_1_120010001.zip Assignment_3_120010001

main@ubuntu:~/Desktop$ ls
Assignment_1_120010001                Assignment_1_120010001.zip
```

Grading Rules

Program part 90'

You can test the correctness of your output using the following commands under directory.

- Execution with argument 0 will simply return grade
- Execution with argument 1 will return the test result of each case (1 for correct, 0 for wrong).

```
main@ubuntu:~/Desktop$ ./grader 1
Test result 1 0 0
main@ubuntu:~/Desktop$ ./grader 0
20
```

Task1_1	20p
Task1_2	30p
Task2	40p
Report	10p