COMP30023 Project 1
Lock, Lock, ..., Deadlock

Out date: March 14, 2022
Due date: No later than 14:59 March 29, 2022 AEDT

Weight: 15% of the final mark

Background

In this project you will familiarise yourself with detection of deadlocks in a multi-process environment. Please note that you do not need to lock any files to implement this project.

We strongly advise you to start working on the project as soon as possible and read the whole spec before beginning implementation.

1 Detecting and breaking locks

You will be given a description of running processes and the files they need to execute, where each process is associated with two files. The first one is the file that the process has locked and the second one is the file that it is waiting to lock. The goal of your program is to determine whether given the current resource allocation and requests there is a deadlock or not. Please see Week 2 Lecture 2 on deadlocks for more information.

If there is no deadlock, you are asked to compute the minimum execution time required for all processes to finish given the following specification. If a process locked a file, it takes 1 time unit to operate on this file and release it. If a process requested a file, after a process is given this file, it takes 1 time unit to operate on this file and release it. Hence, any process takes at least 2 time units to finish. If k processes have requested the same file, it will take k time units for the processes to finish, where $k \geq 1$. Processes that have locked or requested distinct files can operate in parallel. That is, if two processes have requested two different files then it will take a total of 1 time unit for the processes to operate on these files if both files are not locked by other processes.

If there is a deadlock, you are asked to return process(es) that have to be terminated in order to resolve the deadlock.

2 Program Specification

Your program will be invoked via the command line. It must be called `detect` and take the following command line arguments. The arguments can be passed **in any order** but you can assume that they will be passed exactly once.

`-f` ***filename*** specifies the path to the file describing requests.
`-e` is an optional parameter, which when provided requires your code to compute execution time. If this option is not given, your code should identify presence or absence of deadlocks and which processes need to be terminated to resolve deadlock(s).
`-c` is an optional parameter, which when provided invokes your file per process allocation from Section 4.

The file specified by *filename* contains the processes and their locked and requested resources in the following format.

Each line of the file corresponds to a process and consists of a space-separated tuple:

process-id space *file-id* space *file-id* where the first *file-id* is the file that the process has locked and the second *file-id* is the file requested by the process. You can assume that all *process-id*s will be distinct integers in the domain of $[0, 2^{32})$. You can assume that all *file-id*s will be integers in the domain of $[0, 2^{32})$.

Example: `./detect -f resources.txt`

The detection program is required to determine if there is a deadlock in the current resource allocation.

For example `resources.txt` with the following information:

```
0 1 3
1 2 7
```

describes a system where process 0 has locked file 1 and is waiting for file 3 while process 1 has locked file 2 and waiting for file 7.

If `resources.txt` has the following information:

```
0 1 2
1 2 1
```

then the system has processes 0 and 1, with locks on files 1 and 2, respectively. In addition, process 0 requested file 2 while process 1 requested file 1. Hence, there is a deadlock.

Each line (including the last) will be terminated with a LF (ASCII `0x0a`) control character.

We will not give malformed input (e.g., negative ids for processes and files, non-space delimiters, situations where more than one process locked the same file).

3 Expected Output

In order for us to verify that your code meets the specification, it should print to standard output (`stderr` will be ignored) the following information. You should print each line of the expected output followed by new line.

To get full marks you will be required to print the whole transcript as described below. However, you can get marks for each task individually as long as your output for that specific task is correct. For example, you can get full marks for Task 1 and 2 without having implemented later tasks.

Task 1. Systems statistics You are asked to print the total number of processes and files in the system (both locked and requested).

Task 2. Execution time (No deadlock) You can assume that there is no deadlock in the file allocation for this task **only**. You are asked to print minimum execution time required for all processes to finish given the specification in Section 1.

Task 3. Deadlock detection Your program should print `No deadlocks` if there are no deadlocks, or `Deadlock detected` if there is at least one deadlock.

Tasks 4,5. Breaking deadlocks If your program returns `Deadlock detected`, you are also asked to return a list of processes that need to be terminated to break the deadlock. You should print them on a separate line, separated by spaces. If there are many processes which can be terminated to break a particular deadlock, choose the process with the smallest ID.

You will need to print the minimum number of processes that need to be terminated to remove *all* deadlocks. If there is more than one process that need to be terminated, separate them by space and print them in ascending order, sorted by process ID.

Sample output 1

Given `resources.txt` with the following information:

```
0 1 2
1 2 1
```

On `./detect -f resources.txt` your program should print:

```
Processes 2\n
Files 2\n
Deadlock detected\n
Terminate 0\n
```

Sample output 2

Given `resources.txt` with the following information:

```
    0 1 3
    1 2 4
```

On `./detect -e -f resources.txt` your program should print:

```
Processes 2\n
Files 4\n
Execution time 2\n
```

since processes can proceed in parallel with each other, with each process taking 2 time units as each is operating on 2 files sequentially.

4 Challenge task

If a deadlock was detected, you are asked to come up with a transcript that allocates files to processes in such a way that a deadlock is avoided. That is, you can assume that all resources are available and no files have been locked yet. Instead, the two *file-id*s associated with a process in the input file are two files required for the process to execute. A process can only execute if it obtains *both* files at the same time. Once it obtains the files, it locks them and releases at the next time unit. Hence, a file cannot be allocated to more than one process at any given time. A naive way of allocating files to processes would be to allocate both files to a process, then, once this process releases them (i.e., at the next time unit), proceed to allocating files to the next process. This would however lead to a slow execution that does not have any parallelism. Your task is instead to allocate files to separate processes at the same time while avoiding deadlocks.

In order for us to evaluate whether your allocation avoids deadlocks, you are asked to print an allocation transcript in the following format. If a process is allocated its two files, print on a separate line

time space *process-id* space *file-id* comma *file-id*

The order in which the *file-id*s appear above is up to you. You can assume that the process will automatically release the files allocated to it at the next time unit, so you do not have to print anything to release the files. You are also asked to print the total simulation time, i.e., the time when all processes have been given all the files and released them. Time should start at 0, that is the first line of your transcript will have 0 as *time*.

You will be required to explain how your algorithm is more efficient than naive sequential allocation in a short report.

Sample output 3

Given `resources.txt` with the following information:

```
    0 1 2
    1 2 3
    2 3 4
    3 4 1
```

`./detect -f resources.txt -c` could print:

```
0 0 1,2\n
0 2 3,4\n
1 1 2,3\n
1 3 4,1\n
Simulation time 2\n
```

Note that with `-c` your program should not print the output required in Section 3.

5 Marking Criteria

The marks are broken down as follows:

Task # and description	**Marks**
1. System statistics	1
2. Execution time	3
3. Deadlock detection	2
4. Breaking deadlocks (1 deadlock)	3
5. Breaking deadlocks (> 1 deadlock)	2
6. Challenge task (Section 4)	2
7. Quality of software practices	1
8. Build quality	1

Tasks 1-5. We will run all baseline algorithms against our test cases. You will be given access to half of these test cases and their expected outputs. Half of your mark for the first 5 tasks will be based on these known test cases. Hence, we strongly recommend you to test your code against them.

Task 6. The challenge task, as described in Section 4, will be evaluated based on a set of 3 test cases provided to you and your report. We will use these test cases to compare the `Simulation time` of your algorithm (with `-c` option) with the sequential execution. A scheduling algorithm that results in a shorter `Simulation time` over the sequential algorithm on all 3 test cases will be awarded 1.5 marks, 1 mark for improving on 2/3 test cases and 0.5 marks for improving on 1/3 test cases. We may use a verification algorithm on your transcript to verify that your allocation is sound (e.g., that same file is not given to two different processes at the same time and that your simulation time is computed correctly). You will not be provided with expected outputs for test cases in this task.

Please submit a *report* explaining your scheduling algorithm and why it performs better than the sequential one. Reports longer than 50 words will automatically get 0 marks. Hence, ensure that you get 50 or less words when you run `wc -w report.txt`. The report should be in ASCII format. The report will be marked only if your algorithm improves simulation time on at least one test case.

Task 7. Quality of software practices Factors considered include **quality of code**, based on the choice of variable names, comments, indentation, abstraction, modularity, and **proper use of version control**, based on the regularity of commit and push events, their content and associated commit messages (e.g., repositories with a single commit and push, non-informative commit messages will lose 0.5 mark).

Task 8. Build quality Running `make clean && make -B && ./detect <command line arguments>` should execute the submission. If this fails for any reason, you will be told the reason, and be allowed to resubmit (with the usual late penalty).

A 0.5 mark penalty will be applied if compiling using "`-Wall`" yields a warning or if your final commit contains `detect`, any other executable or `.o` files (see Practical 2). Executable scripts (with `.sh` extension) are excepted.

The automated test script expects `detect` to exit/return with status code of 0 (i.e., it successfully runs and terminates).

We will not give partial marks or allow code edits for either known or hidden cases without applying late penalty (calculated from the deadline).

6 Submission

All code must be written in C (e.g., it should not be a C-wrapper over non C-code) and cannot use any external libraries. Your code will likely rely on data structures for managing processes and memory. You will be expected to write your own versions of these data structures. You can reuse the code that *you wrote* for your other *individual* projects if you clearly specify when and for what purpose you have written it (e.g., the code and the name of the subject, project description and the date, that can be verified if needed). You may use standard libraries (e.g., to print, read files, sort, parse command line arguments[1] etc.). Your code must compile and run on the provided VMs and produce deterministic output.

The repository must contain a Makefile which produces an executable named "`detect`", along with all source files required to compile the executable. Place the Makefile at the root of your repository, and ensure that running `make` places the executable there too. Make sure that all source code is committed and pushed. **Executable files** (that is, all files with the executable bit which are in your repository) **will be removed** before marking. Hence, ensure that none of your source files have the executable bit.

For your own protection, it is advisable to commit your code to git at least once per day. Be sure to `push` after you `commit`. The git history may be considered for matters such as special consideration, extensions and potential plagiarism. Your commit messages should be a short-hand chronicle of your implementation progress and will be used for evaluation in the Quality of Software Practices criterion.

You must **submit the full 40-digit SHA1 hash** of your chosen commit to the **Project 1 Assignment** on LMS. You must **also push your submission** to the repository named `comp30023-2022-project-1` in the subgroup with your username of the group `comp30023-2022-projects` on `gitlab.eng.unimelb.edu.au`. You will be allowed to update your chosen commit. However, only the last commit hash submitted to LMS before the deadline will be marked without late penalty.

You should ensure that the commit which you submitted is accessible from a fresh clone of your repository. For example (below ... are added for aesthetic purposes to break the line):

```
git clone https://gitlab.eng.unimelb.edu.au/comp30023-2022-projects/<username>/ ...
... comp30023-2022-project-1
cd comp30023-2022-project-1
git checkout <commit-hash-submitted-to-lms>
```

Late submissions will incur a deduction of 2 mark per day (or part thereof).

Extension policy: If you believe you have a valid reason to require an extension, please fill in the form accessible on Project 1 Assignment on LMS. Extensions **will not be** considered otherwise. Requests for extensions are not automatic and are considered on a case by case basis.

7 Testing

You have access to several test cases and their expected outputs. However, these test cases are not exhaustive and will not cover all edge cases. Hence, you are also encouraged to write more tests to further test your own implementation.

Testing Locally: You can clone the sample test cases to test locally, from:
`comp30023-2022-projects/project-1`.

Continuous Integration Testing: To provide you with feedback on your progress before the deadline, we have set up a Continuous Integration (CI) pipeline on Gitlab. Though you are strongly encouraged to use this service, the usage of CI is not assessed, i.e., we do not require CI tasks to complete for a submission to be considered for marking.

Note that the test cases which are available on Gitlab are the same as the set described above.

The requisite `.gitlab-ci.yml` file has been provisioned and placed in your repository, but is also available from the `project-1` repository linked above. Please clone, commit and push to trigger.

[1] `https://www.gnu.org/software/libc/manual/html_node/Getopt.html`

8 Getting help

Please see Project 1 Help module on LMS.

9 Collaboration and Plagiarism

You may discuss this project abstractly with your classmates but what gets typed into your program must be individual work, not copied from anyone else. Do **not** share your code and do **not** ask others to give you their programs. Do **not** post your code on subject's discussion board Ed. The best way to help your friends in this regard is to say a very firm "**no**" if they ask to see your program, pointing out that your "**no**", and their acceptance of that decision, are the only way to preserve your friendship. See `https://academicintegrity.unimelb.edu.au` for more information.

Note also that solicitation of solutions via posts to online forums, whether or not there is payment involved, is also Academic Misconduct. You should not post your code to any public location (e.g., `github`) until final subject marks are released.

If you use any code not written by you, you must attribute that code to the source you got it from (e.g., a book or Stack Exchange).

Plagiarism policy: You are reminded that all submitted project work in this subject is to be your own individual work. Automated similarity checking software may be used to compare submissions. It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the student concerned.

Using git is an important step in the verification of authorship.