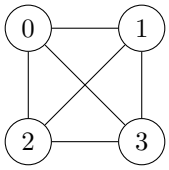
OPERATING SYSTEMS EXERCISE 3

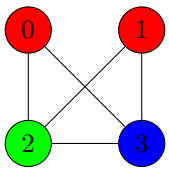
3-Coloring

Implement an algorithm which makes a graph 3-colorable by removing the least edges possible. A graph is 3-colorable if it is possible to assign one of 3 colors to each vertex, such that no two connected vertices share the same color.

For instance, consider the following graph:



This graph is not 3-colorable; however it can be made 3-colorable by removing the edge $(0, 1)$:



Thus $\{ (0, 1) \}$ is a minimal set of edges which must be removed to make this graph 3-colorable. Note that there are other such sets of equal length, for instance $\{ (0, 2) \}$, $\{ (0, 3) \}$ or $\{ (2, 3) \}$.

The problem of deciding whether a graph is 3-colorable is NP-complete and the same is true for the problem of finding a minimal set of edges to make a graph 3-colorable. Therefore an exact solution becomes infeasible for moderately large graph sizes. However, a randomized algorithm can be used to find a good approximation, i.e. to find a set of edges which is close to minimal. [1]

A simple randomized algorithm for this problem generates a set of edges which must be removed to make a given graph 3-colorable by executing following steps:

- Generate a random (not necessarily valid) 3-coloring of the graph, i.e. randomly assign one of 3 colors to each vertex of the graph.
- Select all edges (u, v) for which the color of u is identical to the color of v. These edges need to be removed to make the graph 3-colorable.

  Proof: Once these edges have been removed, the previously generated 3-coloring becomes valid. Thus removing these edges makes the graph 3-colorable.

These steps are executed repeatedly and the smallest set of edges so far is retained.

Although it might seem surprising given its simplicity, this algorithm has an expected runtime which is polynomial in the size of the graph and thus is on average much faster than an algorithm which tries to find an exact solution.

[1] You can read more about the complexity of the 3-colorability problem and about Monte Carlo randomized algorithms in general on Wikipedia:
`https://en.wikipedia.org/wiki/Graph_coloring`
`https://en.wikipedia.org/wiki/Monte_Carlo_algorithm`

Instructions

In order to further reduce the runtime of this algorithm, multiple processes generate the random sets of edges in parallel and report their results to a supervisor process, which remembers the set with the least edges.

Write two programs: a **generator program** and a **supervisor program**. **Multiple generator processes** generate random solutions to the problem and report their solutions to **one supervisor process**. The supervisor process remembers the best solution so far. The processes communicate with eachother by means of a circular buffer, which is implemented using shared semaphores and a shared memory.

Supervisor

The supervisor sets up the shared memory and the semaphores and initializes the circular buffer required for the communication with the generators. It then waits for the generators to write solutions to the circular buffer.

The supervisor program takes no arguments.

Once initialization is complete, the supervisor reads the solutions from the circular buffer and remembers the best solution so far, i.e. the solution with the least edges. Every time a better solution than the previous best solution is found, the supervisor writes the new solution to standard output. If a generator writes a solution with 0 edges to the circular buffer, then the graph is acyclic and the supervisor terminates. Otherwise the supervisor keeps reading results from the circular buffer until it receives a `SIGINT` or a `SIGTERM` signal.

Before terminating, the supervisor notifies all generators that they should terminate as well. This can be done by setting a variable in the shared memory, which is checked by the generator processes before writing to the buffer. The supervisor then unlinks all shared resources and exits.

Generator

The generator program takes a graph as input. The program repeatedly generates a random solution to the problem as described on the first page and writes its result to the circular buffer. It repeats this procedure until it is notified by the supervisor to terminate.

The generator program takes as arguments the set of edges of the graph:

```
SYNOPSIS
    generator EDGE1...
EXAMPLE
    generator 0-1 0-2 0-3 1-2 1-3 2-3
```

Each positional argument is one edge; at least one edge must be given. An edge is specified by a string, with the indices of the two nodes it connects separated by a `-`. Note that the number of nodes of the graph is implicitly provided through the indices in the edges. In the example above the generator program is called with the graph shown on the top of the first page.

The generator uses the algorithm described on the first page to generate random sets of edges which must be removed to make the given graph 3-colorable. It writes these edge sets to the circular buffer, one at a time; therefore a set of edges is a single element of the circular buffer. The generator may produce debug output, describing the edge sets which it writes to the circular buffer.

Circular Buffer

The generators report their solutions to the supervisor by means of a circular buffer. The generators write their solutions to the write end of the circular buffer and the supervisor reads them from the read end.

A circular buffer is a common data structure which uses a single fixed-size buffer to implement a queue (i.e. a FIFO buffer). A circular buffer is essentially **an array of the elements** you want to pass through. Elements are written successively to the array and once the end of the array is reached, writing restarts from the beginning. Similarly, elements are also read successively from the array, restarting from the beginning upon reaching the end, thus reading the elements in the exact same order they have been written. [2]

Implement your circular buffer such that **reading and writing** to the buffer can happen **simultaneously**. Some synchronization is required in order to avoid overwriting data which has not been read yet and also to avoid trying to read from locations which have not been written yet. This is achieved using two semaphores: **one semaphore tracking the free space** in the circular buffer and **one semaphore tracking the used space**. The value of the free space semaphore corresponds to the amount of free space in the buffer array; it is initialized to the size of the buffer, since initially the circular buffer is empty. The value of the used space semaphore corresponds to the amount of used space in the buffer array; it is initialized to 0.

Upon **writing to the circular buffer**, the free space semaphore is decremented; if the buffer is currently full and there is no free space to write to, this intentionally blocks the write until space becomes available. After the write is complete, the used space semaphore is incremented, since the buffer now holds one more element which can be read.

Upon **reading from the circular buffer**, the used space semaphore is decremented; if the buffer is currently empty and there is no data to be read, this intentionally blocks the read until data becomes available. After the read is complete, the free space semaphore is incremented, since the position occupied by the element which has been read just has been freed up.

Since multiple generators write to the circular buffer, an additional semaphore will be required to guarantee a **mutually exclusive access to the write end** of the circular buffer.

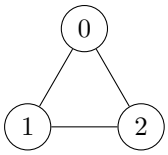
Hints

- Store the graph in a way which is suitable for the implementation of the randomized algorithm.
- Think of a suitable structure to store a set of edges. Choose a limit on the maximum number of edges which it can contain. Since we are looking for small solutions, the generator can discard any solutions which contain too much edges instead of writing them to the circular buffer. You may choose a limit as low as 8 edges.
- Select a reasonable size for the circular buffer. If the size is too small, then the generators will spend a lot of time waiting for free space to become available. If the size is too large, the circular buffer is wasting resources. The size of the shared memory should not exceed 4 KiB.

[2] Visualizations of circular buffers can be found on Wikipedia:
`https://en.wikipedia.org/wiki/Circular_buffer`

Examples

Simple 3-colorable graph



This graph is already 3-colorable.

Invocation of the supervisor:

```
$ ./supervisor
[./supervisor] Solution with 1 edges: 0-2
[./supervisor] The graph is 3-colorable!
```

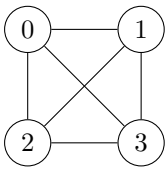
Invocation of one generator:

```
$ ./generator 0-1 0-2 1-2
```

Invocation of 10 generators in parallel:

```
$ for i in {1..10}; do (./generator 0-1 0-2 1-2 &); done
```

Graph from the first page



Removing one edge is sufficient to make this graph 3-colorable, as seen on the first page.

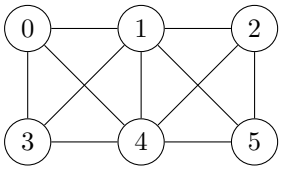
Invocation of the supervisor:

```
$ ./supervisor
[./supervisor] Solution with 2 edges: 0-3 1-2
[./supervisor] Solution with 1 edges: 0-1
```

Invocation of the generator:

```
$ ./generator 0-1 0-2 0-3 1-2 1-3 2-3
```

More complex graph



This graph can again be made 3-colorable by removing a single edge.

Invocation of the supervisor:

```
$ ./supervisor
[./supervisor] Solution with 4 edges: 0-3 0-4 1-2 3-4
[./supervisor] Solution with 3 edges: 0-4 1-2 4-5
[./supervisor] Solution with 2 edges: 1-3 1-5
[./supervisor] Solution with 1 edges: 1-4
```

Invocation of the generator:

```
$ ./generator 0-1 0-3 0-4 1-2 1-3 1-4 1-5 2-4 2-5 3-4 4-5
```

If you want to test your implementation with a challenging graph, try this one [3]:

```
0-1 0-2 0-3 1-28 1-29 2-30 2-31 3-32 3-33 4-6 4-14 4-16 5-7 5-15 5-17 6-7 6-18 7-19
8-9 8-12 8-23 9-13 9-22 10-15 10-19 10-25 11-14 11-18 11-24 12-17 12-27 13-16 13-26
14-23 15-22 16-21 17-20 18-21 19-20 20-31 21-30 22-33 23-32 24-27 24-29 25-26 25-29
26-28 27-28 30-33 31-32
```

Note: This graph is 3-colorable, therefore one of your generators should eventually come up with a solution with 0 edges.

[3] Source: Graph #3349 from the House of Graphs: `https://hog.grinvin.org/ViewGraphInfo.action?id=3349`

Coding Rules and Guidelines

Your score depends upon the compliance of your submission to the presented guidelines and rules. Violations result in deductions of points. Hence, before submitting your solution, go through the following list and check if your program complies.

Rules

Compliance with these rules is essential to get any points for your submission.
In other words, a violation of any of the following rules results in 0 points for your submission.

1. The program must compile via

   ```
   $ gcc -std=c99 -pedantic -Wall -D_DEFAULT_SOURCE -D_BSD_SOURCE -D_SVID_SOURCE
                                  -D_POSIX_C_SOURCE=200809L -g -c filename.c
   ```

   without *errors*. These flags must be used in the Makefile, of course. The feature test macros must not be bypassed (i.e., by undefining these macros or adding some in the C source code).

2. The functionality of the program must conform to the assignment. The program shall operate according to the specification/assignment given the test cases in the respective assignment.

General Guidelines

Violation of following guidelines leads to a deduction of points.

1. The program must compile with

   ```
   $ gcc -std=c99 -pedantic -Wall -D_DEFAULT_SOURCE -D_BSD_SOURCE -D_SVID_SOURCE
                                  -D_POSIX_C_SOURCE=200809L -g -c filename.c
   ```

   without *warnings and info messages*.

2. There must be a Makefile for the program implementing the targets: `all` to build the program from the sources (this must be the first target in the Makefile); `clean` to delete all files that can be built from your sources with the Makefile.

3. The program shall operate according to the specification/assignment without major issues (e.g., segmentation fault, memory corruption).

4. Arguments have to be parsed according to UNIX conventions (we strongly encourage the use of `getopt(3)`). The program has to conform to the given synopsis/usage in the assignment. If the synopsis is violated (e.g., unspecified options or too many arguments), the program has to terminate with the usage message containing the program name and the correct calling syntax. Argument handling should also be implemented for programs without arguments.

5. Correct (=normal) termination, including a cleanup of resources.

6. Upon success the program has to terminate with exit code 0, in case of errors with an exit code greater than 0. We recommend to use the macros `EXIT_SUCCESS` and `EXIT_FAILURE` (defined in `stdlib.h`) to enable portability of the program.

7. If a function indicates an error with its return value, it *should* be checked in general. If the subsequent code depends on the successful execution of a function (e.g. resource allocation), then the return value *must* be checked.

8. Functions that do not take any parameters have to be declared with `void` in the signature, e.g., int get_random_int(void);.

9. Procedures (i.e., functions that do not return a value) have to be declared as `void`.

10. Error messages shall be written to `stderr` and should contain the program name `argv[0]`.

11. It is forbidden to use the functions: `gets`, `scanf`, `fscanf`, `atoi` and `atol` to avoid crashes due to invalid inputs.

FORBIDDEN	USE INSTEAD
gets	fgets
scanf	fgets, sscanf
fscanf	fgets, sscanf
atoi	strtol
atol	strtol

12. Documenation is mandatory. Format the documentation in Doxygen style (see Wiki and Doxygen's intro).

13. Write meaningful comments. For example, meaningful comments describe the algorithm, or why a particular solution has been chosen, if there seems to be an easier solution at a first glance. Avoid comments that just repeat the code itself
(e.g., i = i + 1; /* i is incremented by one */).

14. The documentation of a module must include: name of the module, name and student id of the author (`@author` tag), purpose of the module (`@brief`, `@details` tags) and creation date of the module (`@date` tag).

    Also the Makefile has to include a header, with author and program name at least.

15. Each function shall be documented either before the declaration or the implementation. It should include purpose (`@brief`, `@details` tags), description of parameters and return value (`@param`, `@return` tags) and description of global variables the function uses (`@details` tag).

    You should also document `static` functions (see `EXTRACT_STATIC` in the file `Doxyfile`). Document visible/exported functions in the header file and local (`static`) functions in the C file. Document variables, constants and types (especially `structs`) too.

16. Documentation, names of variables and constants shall be in English.

17. Internal functions shall be marked with the `static` qualifier and are not allowed to be exported (e.g., in a header file). Only functions that are used by other modules shall be declared in the header file.

18. All exercises shall be solved with functions of the C standard library. If a required function is not available in the standard library, you can use other (external) functions too. Avoid reinventing the wheel (e.g., re-implementation of `strcmp`).

19. Name of constants shall be written in upper case, names of variables in lower case (maybe with fist letter capital).

20. Use meaningful variable and constant names (e.g., also semaphores and shared memories).

21. Avoid using global variables as far as possible.

22. All boundaries shall be defined as constants (macros). Avoid arbitrary boundaries. If boundaries are necessary, treat its crossing.

23. Avoid side effects with `&&` and `||`, e.g., write if (b != 0) c = a/b; instead of if (b != 0 && c = a/b).

24. Each `switch` block must contain a `default` case. If the case is not reachable, write `assert(0)` to this case (defensive programming).

25. Logical values shall be treated with logical operators, numerical values with arithmetic operators (e.g., test 2 strings for equality by strcmp (...) == 0 instead of !strcmp (...)).

26. Indent your source code consistently (there are tools for that purpose, e.g., `indent`).

27. Avoid tricky arithmetic statements. Programs are written once, but read more times. Your program is not better if it is shorter!

28. For all I/O operations (read/write from/to `stdin`, `stdout`, files, sockets, pipes, etc.) use *either* standard I/O functions (`fdopen(3)`, `fopen(3)`, `fgets(3)`, etc.) *or* POSIX functions (`open(2)`, `read(2)`, `write(2)`, etc.). Remember, standard I/O functions are buffered. Mixing standard I/O functions and POSIX functions to access a common file descriptor can lead to undefined behaviour and is therefore forbidden.

29. If asked in the assignment, you must implement signal handling (`SIGINT`, `SIGTERM`). You must only use *async-signal-safe* functions in your signal handlers.

30. Close files, free dynamically allocated memory, and remove resources after usage.

31. Don't waste resources due to inconvenient programming. Header files shall not include implementation parts (exception: macros).

Exercise 3 Guidelines

Violation of following guidelines leads to a deduction of points in exercise 3.

1. Correct use of named sempahores (`sem_open(3)`, `sem_close(3)` `sem_unlink(3)`) and POSIX shared memory (`shm_overview(7)`) for inter-process communication of separated programs (e.g., server and client).

   Use your matriculation number as prefix in the names of all resources.

2. "Busy waiting" is forbidden. (Busy waiting is the repeated check of a condition in a loop for synchronization purposes.)

3. Synchronization with `sleep` is forbidden.

Hints

Below are several hints regarding this exercise which should help you to ensure the guidelines.

Client/server architecture. Derive from the exercise if a client/server architecture makes sense. In case of a client/server architecture (server creates the resources):

- The client shall terminate with an appropriate error message if the resources cannot be found.
- Assume that more than one client can start concurrently. You can assume only one server runs at the same time.
- The server should stay functional after error-free termination of a client.

If you don't use a client/server architecture, assume that each process can be started once in random order.

Cleanup resources. Resources shall be cleaned up also in case of errors.

Note, shared memory objects and semaphores won't be automatically deleted after the termination of a program. Hence, these resources have to be explicitly removed.

Shared memory and semaphores, that haven't been removed due to a program error or crash, can be listed and removed with the usual commands, `ls` and `rm` respectively, in the folder `/dev/shm/`. You can identify your resources by the chosen name (matriculation number) or ownership of the resource files.

Avoid wasting resources. Use a shared memory of fixed size. Use a minimum number of semaphores possible, but ensure correct synchronization (you may ask our tutors during lab hours).

Termination. After correct program execution (without errors) a synchronous termination shall be performed, i.e., take care of the synchronization between running processes when deleting the resources.