The University of Queensland
School of Information Technology and Electrical Engineering

CSSE2310/CSSE7231 — Semester 1, 2023
Assignment 3 (version 1.0)

Marks: 75 (for CSSE2310), 85 (for CSSE7231)
Weighting: 15%
Due: 4:00pm Friday 5th May, 2023

Introduction

The goal of this assignment is to demonstrate your skills and ability in fundamental process management and communication concepts, and to further develop your C programming skills with a moderately complex program.

You are to create a program called `testuqwordiply` that creates and manages communicating collections of processes that test a `uqwordiply` program (from assignment one) according to a job specification file that lists tests to be run. For various test cases, your program will run both a test version of `uqwordiply` and the assignment one solution (available as `demo-uqwordiply`) and compare their standard outputs, standard errors and exit statuses and report the results. You will be provided with (and must use) a program named `uqcmp` that will compare the input arriving on two file descriptors. The assignment will also test your ability to code to a programming style guide and to use a revision control system appropriately.

~~CSSE7231 students will, in addition, write their own version of `uqcmp`.~~

Student Conduct

This is an individual assignment. You should feel free to discuss **general** aspects of C programming and the assignment specification with fellow students, including on the discussion forum. In general, questions like "How should the program behave if ⟨this happens⟩?" would be safe, if they are seeking clarification on the specification.

You must not actively help (or seek help from) other students or other people with the actual design, structure and/or coding of your assignment solution. It is **cheating to look at another student's assignment code** and it is **cheating to allow your code to be seen or shared in printed or electronic form by others**. All submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct actions will be initiated against you, and those you cheated with. That's right, if you share your code with a friend, even inadvertently, then **both of you are in trouble**. Do not post your code to a public place such as the course discussion forum or a public code repository. (Code in private posts to the discussion forum is permitted.) You must assume that some students in the course may have very long extensions so do not post your code to any public repository until at least three months after the result release date for the course (or check with the course coordinator if you wish to post it sooner). Do not allow others to access your computer – you must keep your code secure. Never leave your work unattended.

You must follow the following code referencing rules for **all code committed to your SVN repository** (not just the version that you submit):

Code Origin	**Usage/Referencing**
Code provided to you in writing **this semester** by CSSE2310/7231 teaching staff (e.g. code hosted on Blackboard, found in `/local/courses/csse2310/resources` on `moss`, posted on the discussion forum by teaching staff, provided in Ed Lessons, or shown in class).	May be used freely without reference. (You must be able to point to the source if queried about it – so you may find it easier to reference the code.)
Code you have personally written this semester for CSSE2310/7231 (e.g. code written for A1 reused in A3) – provided you have not shared or published it.	May be used freely without reference. (This assumes that no reference was required for the original use.)
Code examples found in man pages on `moss`.	May be used provided you understand the code AND the source of the code is referenced in a comment adjacent to that code (in the required format – see the style guide). If such code is used without appropriate referencing then this will be considered misconduct.
Code you have personally written in a previous enrolment in this course or in another ITEE course and where that code has not been shared or published.	
Code (in any programming language) that you have taken inspiration from but have not copied[1].	

[1] Code you have *taken inspiration from* must not be directly copied or just converted from one programming language to another.

Code Origin	Usage/Referencing
Code written by or obtained from, or based on code written by or obtained from, a code generation tool (including any artificial intelligence tool) that you personally have interacted with, without the assistance of another person.	May be used provided you understand that code AND the source of the code is referenced in a comment adjacent to that code (in the required format) AND an ASCII text file (named `toolHistory.txt`) is included in your repository and with your submission that describes in detail how the tool was used. If such code is used without appropriate referencing and without inclusion of the `toolHistory.txt` file then this will be considered misconduct.
Other code – includes (but may not be limited to): code provided by teaching staff only in a previous offering of this course (e.g. previous assignment one solution); code from public or private repositories; code from websites; code from textbooks; any code written or partially written or provided by or written with the assistance of someone else; and any code you have written that is available to other students.	May **not** be used. If the source of the code is referenced adjacent to the code then this will be considered code without academic merit (not misconduct) and will be removed from your assignment prior to marking (which may cause compilation to fail and zero marks to be awarded). Copied code without adjacent referencing will be considered misconduct and action will be taken.

Uploading or otherwise providing the assignment specification or part of it to a third party including online tutorial and contract cheating websites is considered misconduct. The university is aware of these sites and many cooperate with us in misconduct investigations.

The teaching staff will conduct interviews with a subset of students about their submissions, for the purposes of establishing genuine authorship. If you write your own code, you have nothing to fear from this process. If you legitimately use code from other sources (following the usage/referencing requirements in the table above) then you are expected to understand that code. If you are not able to adequately explain the design of your solution and/or adequately explain your submitted code (and/or earlier versions in your repository) and/or be able to make simple modifications to it as requested at the interview, then your assignment mark will be scaled down based on the level of understanding you are able to demonstrate and/or your submission may be subject to a misconduct investigation where your interview responses form part of the evidence. Failure to attend a scheduled interview will result in zero marks for the assignment. The use of referenced code in your submission (particularly from artificial intelligence tools) is likely to increase the probability of your selection for an interview.

In short - **Don't risk it!** If you're having trouble, seek help early from a member of the teaching staff. Don't be tempted to copy another student's code or to use an online cheating service. Don't help another CSSE2310/7231 student with their code no matter how desperate they may be and no matter how close your relationship. You should read and understand the statements on student misconduct in the course profile and on the school website: `https://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`.

Specification – `testuqwordiply`

`testuqwordiply` reads a set of test jobs from a file whose name is provided as a command line argument. (The format of this file is specified below.) For each of these test jobs, `testuqwordiply` will run both a program to be tested (whose name is given on the command line, e.g. your solution to assignment one) and `demo-uqwordiply` and compare their outputs (`stdout` and `stderr`) and their exit statuses. If the `stdout`, `stderr` and exit statuses match, then the test passes, otherwise it fails. The comparison of the programs' outputs will require the use of a provided comparison program called `uqcmp` which reads input on file descriptors 3 and 4 (e.g. piped from `stdout` from each of the programs) and reports whether that data is the same or different.

Figure 1 illustrates the processes and pipes to be created by `testuqwordiply` for each test job. Four processes will be created (labelled A, B, C and D in Figure 1): an instance of the program being tested (A), an instance of `demo-uqwordiply` (B) and two instances of `uqcmp` – one for comparing the standard outputs of the two programs (C) and one for comparing the standard errors (D). This will require the creation of four pipes (labelled 1, 2, 3 and 4 in Figure 1). The standard input for the two programs being tested will come from a file whose name is specified in the job file – this contains sample user input for that test.

Full details of the required behaviour are provided below.

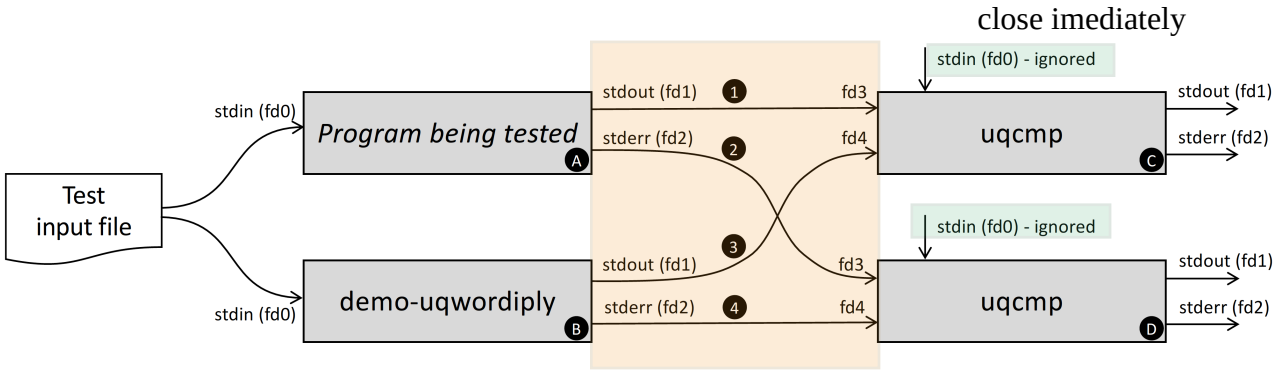

Figure 1: Processes (A to D) and pipes (1 to 4) to be created by `testuqwordiply` for each test job.

Command Line Arguments

`testuqwordiply` has two mandatory arguments – the name of the program being tested, and the name of a job specification file that specifies the tests to be run. There are two optional arguments which may appear in either order if both are present (but must appear before the mandatory arguments).

Usage of the program and the meaning of the arguments are as follows:

`./testuqwordiply [--quiet] [--parallel]` ***`testprogram jobfile`***

- `--quiet` – if present, the output (`stdout` and `stderr`) of the `uqcmp` processes is to be suppressed (i.e. not shown), otherwise it should be output to `testuqwordiply`'s `stdout` and `stderr` respectively. This option does not change `testuqwordiply`'s output – it will still send output to `stdout` and `stderr` as described in this specification. This behaviour is described in more detail below.
- `--parallel` – if present, the test jobs are to be run in parallel, otherwise they should be run sequentially (one after the other). This behaviour is explained in more detail below.
- ***`testprogram`*** – must be present and is the name of the program being tested against `demo-uqwordiply`. The name can be a relative or absolute pathname or otherwise (if it doesn't contain a slash '/') is expected to be found on the user's PATH. The argument may not begin with `--`. (If a program's name begins with `--` then the path to the program must be given, e.g. `./--prog`.)
- ***`jobfile`*** – must be present (after the program name) and is the name of the file containing details of the tests to be run (referred to in the remainder of this document as the *job specification file*). More details on the format of this file are given below. The argument may not begin with `--`. (If a file's name begins with `--` then the path to the file must be given, e.g. `./--jobfile`.)

Prior to doing anything else, your `testuqwordiply` program must check the validity of the command line arguments. If the program receives an invalid command line then it must print the message:

```
Usage: testuqwordiply [--quiet] [--parallel] testprogram jobfile
```

to standard error (with a following newline) and exit with an exit status of 2.

Invalid command lines include (but may not be limited to) any of the following:

- arguments that begin with `--` that are not either `--quiet` or `--parallel`[2]
- either the `--quiet` or `--parallel` argument is specified more than once
- the program name and/or the job file name are not specified on the command line
- an unexpected argument is present

Checking whether the program name and/or the file name are valid is not part of usage checking.

[2]To allow for `getopt()` implementations of argument parsing (not required) we will not run any tests using the argument `--`

what are the conditions of program name and files name?!

cases that # in the middle of the read line?

Provided Library: libcsse2310a3

A library has been provided to you with the following functions which your program may use. See the man pages on moss for more details on these library functions.

```
char* read_line(FILE *stream);
```

The function attempts to read a line of text from the specified stream, allocating memory for it, and returning the buffer.

```
char **split_line(char* line, char delimiter);
```

This function will split a line into substrings based on a given delimiter character.

```
char** split_space_not_quote(char *input, int *numTokens);
```

This function takes an input string and tokenises it according to spaces, but will treat text within double quotes as a single token.

To use the library, you will need to add `#include <csse2310a3.h>` to your code and use the compiler flag `-I/local/courses/csse2310/include` when compiling your code so that the compiler can find the include file. You will also need to link with the library containing this function. To do this, use the compiler arguments `-L/local/courses/csse2310/lib -lcsse2310a3`.

Job Specification File

If the command line arguments are valid then `testuqwordiply` reads the job specification file listed on the command line. The whole file must be read and checked prior to any test jobs being run.

If `testuqwordiply` is unable to open the job specification file for reading then it must print the following message to `stderr` (with a following newline) and exit with an exit status of 3:

`testuqwordiply: Unable to open job file "`*`filename`*`"`

ex: testuqwordiply: Unable to open job file "test2"

where *`filename`* is replaced by the name of the file (as given on the command line). (The double quotes around the filename must be present in the output message.)

File Format

how about format of file is not correct?

The job specification file provided to `testuqwordiply` is a text file, with one line per job. The file may contain any number of lines.

If `testuqwordiply` is able to open the job specification file for reading, then it should read and check all of the test jobs in the file before starting any tests.

Lines in the job file beginning with the '`#`' (hash) character are comments[3], and are to be ignored by `testuqwordiply`. Similarly, empty lines (i.e. with no characters before the newline) are to be ignored.

All other lines are to be interpreted as job specifications, split over 2 separate fields delimited by the comma ('`,`') character as follows:

separator!

`input-file-name,[arg1 arg2 ...]`

space!

The `input-file-name` field is the name of a file to be used as the standard input to the the test program and to `demo-uqwordiply` for that test. It may be an absolute pathname or relative to the current directory. It must not be an empty field.

The second field is a space separated list of command line arguments to be provided to the test program and to `demo-uqwordiply` when the test is run. This field may be empty. Arguments that contain spaces may be enclosed by double quotes. Leading spaces and extra separating spaces are ignored. (A helper function is provided to make processing this field easier, see the `split_space_not_quote()` function described on page 10.)

Note that individual job specifications are independent – a given test job will always be run in the same way independent of any other test jobs described in the file.

The comma character has special meaning in job specification files and will only appear in job lines as a separator. You do not need to consider, nor will we test for, job specification files that contain the comma character as part of a file name or an argument. See the `split_line()` function described on page 10 for an easy way to split the comma-delimited job specifications.

Checking the Jobs

Each job listed in the job specification file must be checked in turn. If a job line is not syntactically valid then `testuqwordiply` must print the following message to `stderr` (with a following newline) and exit with an exit status of 4:

`testuqwordiply: syntax error on line `*`linenum`*` of "`*`filename`*`"`

perror()

where *`linenum`* is replaced by the line number in the job specification file (where lines are counted from 1), and *`filename`* is replaced by the name of the job specification file (as given on the command line). (The double quotes around the filename must be present in the output message.) No further job lines are checked.

A syntactically valid line will contain exactly one comma with at least one character before the comma (i.e. a non-empty input filename).

If the line is syntactically valid but the input file specified on the job line is unable to be opened for reading, then `testuqwordiply` must print the following message to `stderr` (with a following newline) and exit with an exit status of 5:

`testuqwordiply: unable to open file "`*`inputfile`*`" specified on line `*`linenum`*` of "`*`filename`*`"`

where *`inputfile`* is replaced by the name of the file specified on the job line (all characters before the comma), *`linenum`* is replaced by the line number in the job specification file (where lines are counted from 1), and

[3] "beginning with" means the `#` character is in the leftmost position on the line – there are no preceding spaces or other characters

filename is replaced by the name of the job specification file (as given on the command line). (The double quotes around both filenames must be present in the output message.)

If these checks are passed (the line is syntactically valid and the input file can be opened for reading) then `testuqwordiply` can move on to the next line in the job file.

If `testuqwordiply` reaches the end of the file and doesn't find any job specifications (i.e. the file is empty or only contains blank lines and/or comments) then `testuqwordiply` must print the following message to `stderr` (with a following newline) and exit with an exit status of 6:

`testuqwordiply: no jobs found in "`*filename*`"`

where *filename* is replaced by the name of the job specification file (as given on the command line). (The double quotes around the filename must be present in the output message.)

The following are examples of valid jobfiles[4] containing explanatory comments:

```
# Run a single test with no command line arguments and an immediate EOF on stdin
/dev/null,
```

```
# Run two tests - specifying the starter word in both cases
testfiles/cab,--start cab
input-file,--start PRO
```

```
# Run four tests - with various combinations of starter word and dictionary
# Also illustrates use of double quotes and additional spaces in arguments
input.1,--start art --dictionary testfiles/common-words
2.in,--dictionary "testfiles/simple dictionary" "--start" ion
3.in,  --start   xyz  --dictionary /dev/null

# Blank line on the line above will be ignored
2.in,--dictionary /usr/share/dict/words --start def
```

Running Test Jobs and Reporting Results

If all of the test jobs in the job specification file are valid then `testuqwordiply` must run the jobs, either in sequence (one after the other) or in parallel (if the `--parallel` argument is provided on the command line). The behaviour of these two different modes is described below.

Running Test Jobs Sequentially

If `testuqwordiply` is running jobs sequentially, then it must perform the following sequence of operations:

For each job (in the order specified in the job specification file):

- Start the job (as described below in *Running One Test Job*)
- Sleep for 2 seconds
- Send a SIGKILL signal to all processes that make up that job. (It is likely they are already dead, but `testuqwordiply` does not need to check this prior to attempting to send a signal.)
- Report the result of the job (as described below in *Reporting the Result of One Test Job*).

When one job is complete, the next job can be started.

Running Test Jobs In Parallel

If `testuqwordiply` is running jobs in parallel, then it must perform the following sequence of operations:

- Start all of the jobs specified in the job specification file (each individual job should be started as described below in *Running One Test Job*)
- Sleep for 2 seconds

[4] This assumes that all of the listed input files are valid readable files in the current directory.

- Send a SIGKILL signal to all processes that make up all jobs. (It is likely they are already dead, but `testuqwordiply` does not need to check this prior to attempting to send a signal.)
- Iterate over each job (in the order specified in the job specification file) and report the result of the job (as described below in *Reporting the Result of One Test Job*).

Running One Test Job

Just before running a test job, `testuqwordiply` must output the following to standard output (followed by a newline):

`Starting job` *`T`*

where *`T`* is replaced by the job number (1 to N, where N is the number of jobs in the job specification file).

For each individual test job, `testuqwordiply` must create four pipes and four processes as shown in Figure 1. The four processes are:

- an instance of the program being tested (as specified on the command line). Standard input for this process must come from the input file specified in the job. Standard output and standard error must be sent to file descriptor 3 of two instances of `uqcmp` (see description of `uqcmp` instances below).
- an instance of `demo-uqwordiply` – which will be found in the user's PATH. Standard input for this process must come from the input file specified in the job. Standard output and standard error must be sent to file descriptor 4 of the two instances of `uqcmp`. This is the expected output from the test program.
- an instance of `uqcmp` which compares standard outputs – which will receive the standard output of the program being tested on file descriptor 3 and the standard output of `demo-uqwordiply` on file descriptor 4. `uqcmp` must be started with one command line argument: "`Job` *`T`* `stdout`" where *`T`* is replaced by the job number.
- an instance of `uqcmp` which compares standard errors – which will receive the standard error of the program being tested on file descriptor 3 and the standard error of `demo-uqwordiply` on file descriptor 4. `uqcmp` must be started with one command line argument: "`Job` *`T`* `stderr`" where *`T`* is replaced by the job number.

If the `--quiet` option is provided on the `testuqwordiply` command line then the standard output and standard error of the `uqcmp` instances must be redirected to `/dev/null`. If `--quiet` is not specified, then the `uqcmp` instances must inherit standard output and standard error from `testuqwordiply` – i.e. send standard output and standard error to wherever `testuqwordiply`'s standard output and standard error are being sent.

Reporting the Result of One Test Job

`uqcmp` will exit with an exit status of 0 if the data received on file descriptor 3 matches that received on file descriptor 4, otherwise it will exit with a non-zero exit status. Full details on the behaviour of the `uqcmp` program are given below.

If any of the programs were unable to be executed (`demo-uqwordiply`, `uqcmp` or the program under test) then `testuqwordiply` must print the following to standard output (followed by a newline):

`Job` *`T`*`: Unable to execute test`

where *`T`* is replaced by the job number. This counts as a test failure. (If one test job fails then all test jobs are likely to fail. A possible scenario is where one of the programs is not found in the user's PATH.)

If the four programs run, then for one individual test (say job number T), `testuqwordiply` must report the following – in this given order. (In all of the messages below, *`T`* is replaced by the job number. All messages are sent to `testuqwordiply`'s standard output and are terminated by a single newline.)

- If the standard outputs of the two programs match (as determined by the exit status of that `uqcmp` instance), then `testuqwordiply` must print the following:
  `Job` *`T`*`: Stdout matches`
  If the standard outputs do not match, then `testuqwordiply` must print the following:
  `Job` *`T`*`: Stdout differs`

- If the standard errors of the two programs match (as determined by the exit status of that `uqcmp` instance), then `testuqwordiply` must print the following:
  `Job T: Stderr matches`
  If the standard errors do not match, then `testuqwordiply` must print the following:
  `Job T: Stderr differs`

- If both processes exit normally with the same exit status, then `testuqwordiply` must print the following:
  `Job T: Exit status matches`
  otherwise it must print:
  `Job T: Exit status differs`

Reporting the Overall Result

A test passes if the standard output, standard error and exit statuses of the program being tested and `demo-uqwordiply` all match each other.

For both sequential and parallel modes of operation, when all test jobs have been run and finished, then `testuqwordiply` must output the following message to `stdout` (followed by a newline):

`testuqwordiply: M out of N tests passed`

where *M* is replaced by the number of tests that passed, and *N* is replaced by the number of tests that have been run (which may be fewer than the number of the tests in the job specification file if the tests are interrupted – see below).

If all tests that have been run passed then `testuqwordiply` must exit with exit status 0 (indicating success), otherwise it must exit with exit status 1 (indicating failure).

Interrupting the Tests

If `testuqwordiply` receives a `SIGINT` (as usually sent by pressing Ctrl-C) then it should complete the test job(s) in progress (including any sleeps), not commence any more test jobs, and report the overall result (as described above) based on the tests that have been run.

In practical terms, this will only make a difference in sequential mode – tests that haven't been started will not be run. In parallel mode, all tests are started immediately, so all tests should be run to completion if a SIGINT is received.

IGNORE!

Specification – `uqcmp`

This section details the expected operation of `uqcmp`. **Only CSSE7231 students are required to implement `uqcmp`**, but this section is useful to all students so as to understand how `uqcmp` behaves. The UQ provided version of `uqcmp` will be used in testing `testuqwordiply`. `uqcmp` programs implemented by CSSE7231 students will be tested independently of `testuqwordiply`.

`uqcmp` will read from file descriptors 3 and 4, compare the data read from both file descriptors, and determine whether it is the same. The data read from file descriptor 4 is considered to be the "expected" (i.e. "correct") data, and is being compared with that received from file descriptor 3.

`uqcmp` Command Line Arguments

`uqcmp` takes one optional command line argument – a prefix that is to be emitted with messages that are output (see below).

If more than one command line argument is provided, then `uqcmp` is to output the following message to standard error (with a following newline) and then exit with exit status 1:

```
Usage: uqcmp [prefix]
```

If no prefix is specified on the command line then the prefix "`uqcmp`" is to be used (without the quotes).

`uqcmp` Operation

If file descriptor 3 is not open, then `uqcmp` must output the following message to standard error (with a following newline) and then exit with status 2:

`uqcmp: fd 3 is not open`

If file descriptor 3 is open and file descriptor 4 is not open, then `uqcmp` must output the following message to standard error (with a following newline) and then exit with status 2:

`uqcmp: fd 4 is not open`

If both file descriptors are open, then `uqcmp` will print a message (with a following newline) and exit as described below. The text *`PREFIX`* is to be replaced by the prefix specified on the command line (or `uqcmp` if no prefix is specified on the command line).

- If identical data is received from both file descriptors 3 and 4 (from commencement to EOF) then `uqcmp` must print the following to `stdout`:
  `PREFIX``: OK`
  and exit with status 0.
- If no data is received on file descriptor 3 but data is received on file descriptor 4 then `uqcmp` must print the following to `stderr`:
  `PREFIX``: Expected data but received nothing`
  and exit with status 3.
- If data is received on file descriptor 3 but no data is received on file descriptor 4 then `uqcmp` must print the following to `stderr`:
  `PREFIX``: Received data but expected nothing`
  and exit with status 4.
- If data is received on both file descriptors 3 and 4 but this data does not match then `uqcmp` must print the following to `stderr`:
  `PREFIX``: Received data does not match expected data`
  and exit with status 5.

`uqcmp` may exit at any point after it is able to make this determination – i.e. it does not have to wait for EOF on both file descriptors in order to exit if it has identified differences – though it may wait until then if desired.

`uqcmp` ignores its standard input.

Testing `uqcmp`

You can test `uqcmp` at the command line by getting the shell to redirect files or standard input to file descriptors 3 and/or 4, e.g.
`./uqcmp 3<&1 4</dev/null`
will redirect file descriptor 3 from standard input and file descriptor 4 from /dev/null.

IGNORE!

Other Functionality – `uqcmp` and `testuqwordiply`

It is not expected that `uqcmp` has any special signal handling capability.

It is expected that `testuqwordiply` frees all dynamically allocated memory before exiting. (This requirement does not apply to child processes of `testuqwordiply`.)

It is expected that child processes created by `testuqwordiply` do not inherit unnecessary open file descriptors. close everything prior to exec it

Example `testuqwordiply` Output

In this section we give examples of expected `testuqwordiply` output for some given job specification files and command lines. Note that two programs that call `get_wordiply_starter_word()` at about the same time will receive the same starter word – the random number generator in this program is seeded by the current time (in seconds).

Note that these examples, like provided test-cases, are not exhaustive. You need to implement the program specification as it is written, and not just code for these few examples.

Example One

Consider a job specification file as follows (named `job1`):

```
# One job - no command line args, immediate EOF on stdin
/dev/null,
```

In the following runs of `testuqwordiply`, assume that `./uqwordiply` is a fully functional program.

```
$ ./testuqwordiply ./uqwordiply job1
Starting job 1
Job 1 stdout: OK
Job 1 stderr: OK
Job 1: Stdout matches
Job 1: Stderr matches
Job 1: Exit status matches
testuqwordiply: 1 out of 1 tests passed
$ ./testuqwordiply --quiet ./uqwordiply job1
Starting job 1
Job 1: Stdout matches
Job 1: Stderr matches
Job 1: Exit status matches
testuqwordiply: 1 out of 1 tests passed
```

Note that in the first run of `testuqwordiply`, the first two lines of output (lines 3 and 4 in the listing) are the standard output from the two instances of `uqcmp`. (These could be printed in either order.) The remaining lines are printed about two seconds later – and will always be in the order shown. In the second run of `testuqwordiply`, the output of `uqcmp` is suppressed by using the `--quiet` command line argument.

Example Two

In the following example, assume that `./buggy-uqwordiply` works correctly except that it exits with a segmentation fault when given an empty dictionary. The job specification file `job2` has the following contents:

```
./7.era.in, --start ERA
7.lay.in,--start LAY --dictionary /dev/null
6.1.in, --dictionary /usr/share/dict/words --start top
```

Assume that `7.lay.in`, `7.era.in` and `6.1.in` are valid input files in the current directory (matching those provided in the assignment one test files found in `/local/courses/csse2310/resources/a1/testfiles.public` on moss).

```
$ time ./testuqwordiply --quiet ./buggy-uqwordiply job2
Starting job 1
Job 1: Stdout matches
Job 1: Stderr matches
Job 1: Exit status matches
Starting job 2
Job 2: Stdout differs
Job 2: Stderr matches
Job 2: Exit status differs
Starting job 3
Job 3: Stdout matches
Job 3: Stderr matches
Job 3: Exit status matches
testuqwordiply: 2 out of 3 tests passed

real    0m6.004s
user    0m0.165s
sys     0m0.014s
$ time ./testuqwordiply --parallel ./buggy-uqwordiply job2
Starting job 1
```

```
21 Starting job 2
22 Starting job 3
23 Job 2 stdout: Expected data but received nothing
24 Job 2 stderr: OK
25 Job 1 stderr: OK
26 Job 1 stdout: OK
27 Job 3 stderr: OK
28 Job 3 stdout: OK
29 Job 1: Stdout matches
30 Job 1: Stderr matches
31 Job 1: Exit status matches
32 Job 2: Stdout differs
33 Job 2: Stderr matches
34 Job 2: Exit status differs
35 Job 3: Stdout matches
36 Job 3: Stderr matches
37 Job 3: Exit status matches
38 testuqwordiply: 2 out of 3 tests passed
39
40 real    0m2.003s
41 user    0m0.165s
42 sys     0m0.012s
```

The lines shown in green (and the blank line prior to these) are the output of the `time` utility which reports how much time is taken to run the program. You can see in the first run (sequential mode) that the three tests take about six seconds total to run. In the second run (parallel mode), the three tests take about two seconds total to run.

In this second run, the output lines from `uqcmp` (lines 23 to 28 inclusive) could have been printed in any order. Note that test job 2 failed due to the exit status and standard output contents not matching.

Provided Library: libcsse2310a3

A library has been provided to you with the following functions which your program may use. See the man pages on moss for more details on these library functions.

```
char* read_line(FILE *stream);
```

The function attempts to read a line of text from the specified stream, allocating memory for it, and returning the buffer.

```
char **split_line(char* line, char delimiter);
```

This function will split a line into substrings based on a given delimiter character.

```
char** split_space_not_quote(char *input, int *numTokens);
```

This function takes an input string and tokenises it according to spaces, but will treat text within double quotes as a single token.

To use the library, you will need to add `#include <csse2310a3.h>` to your code and use the compiler flag `-I/local/courses/csse2310/include` when compiling your code so that the compiler can find the include file. You will also need to link with the library containing this function. To do this, use the compiler arguments `-L/local/courses/csse2310/lib -lcsse2310a3`.

Style

Your program must follow version 2.3.0 of the CSSE2310/CSSE7231 C programming style guide available on the course Blackboard site.

Hints

1. Make sure you review the lecture/contact content and examples and the Ed Lessons exercises from weeks 6 and 7. These provide the necessary background for this assignment.

2. While not mandatory, the provided library functions will make your life a lot easier – use them!
3. You can examine the file descriptors associated with a process by running `ls -l /proc/PID/fd` where `PID` is the process ID of the process.
4. If you implement the test interruption functionality, consider the use of `nanosleep()` to allow resumption of interrupted sleeps.
5. `uqcmp` (CSSE7231): You can determine if a file descriptor is open by trying to use it in a function that will report an error if it is not open, e.g. `fdopen()`, `dup()`, `fcntl()`, etc.

Suggested Approach

It is suggested that you write your program using the following steps. Test your program at each stage and commit to your SVN repository frequently. Note that the specification text above is the definitive description of the expected program behaviour. The list below does not cover all required functionality but will get you started.

1. Write a program to parse the expected command line arguments and handle usage errors.
2. Write code to read and parse the job specification file and save it to a data structure (and output error messages if problems are found).
3. Write code that correctly starts the four processes associated with one job (with the necessary command line arguments).
4. Add code to redirect the standard input of the program under test and `testuqwordiply`.
5. Add code to create the four pipes and connect them as required to the four processes.

Forbidden Functions

You must not use any of the following C functions/statements. If you do so, you will get zero (0) marks for the assignment.

- `goto`
- `longjmp()` and equivalent functions
- `system()` or `popen()`
- `mkfifo()` or `mkfifoat()`

Submission

Your submission must include all source and any other required files (in particular you must submit a `Makefile`). Do not submit compiled files (e.g. `.o` files and compiled programs).

Your programs `testuqwordiply` and `uqcmp` (CSSE7231 only) must build on `moss.labs.eait.uq.edu.au` with the command:

```
make
```

Your program must be compiled with `gcc` with at least the following options:

```
-pedantic -Wall -std=gnu99
```

You are not permitted to disable warnings or use pragmas to hide them. You may not use source files other than `.c` and `.h` files as part of the build process – such files will be removed before building your program.

CSSE7231 only – the default target of your Makefile must cause both programs to be built[5].

If any errors result from the `make` command (i.e. no executable is created) then you will receive 0 marks for functionality (see below). Any code without academic merit will be removed from your program before compilation is attempted (and if compilation fails, you will receive 0 marks for functionality).

[5] If you only submit an attempt at one program then it is acceptable for just that single program to be built when running `make`.

Your program must not invoke other programs or use non-standard headers/libraries other than those explicity described in this specification.

Your assignment submission must be committed to your subversion repository under

`https://source.eait.uq.edu.au/svn/csse2310-sem1-sXXXXXXX/trunk/a3`

where sXXXXXXX is your moss/UQ login ID. Only files at this top level will be marked so **do not put source files in subdirectories**. You may create subdirectories for other purposes (e.g. your own test files) but these will not be considered in marking – they will not be checked out of your repository.

You must ensure that all files needed to compile and use your assignment (including a Makefile) are committed and within the `trunk/a3` directory in your repository (and not within a subdirectory or some other part of your repository) and not just sitting in your working directory. Do not commit compiled files or binaries. You are strongly encouraged to check out a clean copy for testing purposes.

To submit your assignment, you must run the command

```
2310createzip a3
```

on `moss` and then submit the resulting zip file on Blackboard (a GradeScope submission link will be made available in the Assessment area on the CSSE2310/7231 Blackboard site)[6]. The zip file will be named

`sXXXXXXX_csse2310_a3_`*`timestamp`*`.zip`

where `sXXXXXXX` is replaced by your moss/UQ login ID and *timestamp* is replaced by a timestamp indicating the time that the zip file was created.

The `2310createzip` tool will check out the latest version of your assignment from the Subversion repository, ensure it builds with the command '`make`', and if so, will create a zip file that contains those files and your Subversion commit history and a checksum of the zip file contents. You may be asked for your password as part of this process in order to check out your submission from your repository. You will be asked to confirm references in your code.

You must not create the zip file using some other mechanism and you must not modify the zip file prior to submission. If you do so, you will receive zero marks. Your submission time will be the time that the file is submitted via GradeScope on Blackboard, and **not** the time of your last repository commit nor the time of creation of your submission zip file.

We will mark your last submission, even if that is after the deadline and you made submissions before the deadline. Any submissions after the deadline[7] will incur a late penalty – see the CSSE2310/7231 course profile for details.

Note that Gradescope will run the test suite immediately after you submit. When complete[8] you will be able to see the results of the "public" tests.

Marks

Marks will be awarded for functionality, style and `svn` commit history messages. Marks may be reduced if you are asked to attend an interview about your assignment and you do not attend or are unable to adequately respond to questions – see the **Student conduct** section above.

Functionality (60 marks)

Provided your code compiles (see above) and does not use any prohibited statements/functions (see above), and your zip file has been generated correctly and has not been modified prior to submission, then you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Partial marks will be awarded for partially meeting the functionality requirements. Not all features are of equal difficulty. **If your program does not allow a feature to be tested then you will receive 0 marks for that feature**, even if you claim to have implemented it. For example, if your program can never create a child process then we can not test pipe communication between the processes of that job, or your ability to send it a termination signal. Memory-freeing tests require correct functionality also – a program that frees allocated memory but doesn't implement the required functionality can't earn marks for this criteria. This is not a complete list of all dependencies, other dependencies may also exist. If your program takes longer than 10

[6] You may need to use scp or a graphical equivalent such as WinSCP, Filezilla or Cyberduck in order to download the zip file to your local computer and then upload it to the submission site.

[7] or your extended deadline if you are granted an extension.

[8] Gradescope marking may take only a few minutes or more than 30 minutes depending on the functionality/efficiency of your code.

seconds to run any test[9], then it will be terminated and you will earn no marks for the functionality associated with that test. The markers will make no alterations to your code (other than to remove code without academic merit).

Functionality marks (out of 60) will be assigned for `testuqwordiply` in the following categories (CSSE2310 and CSSE7231):

1. `testuqwordiply` correctly handles invalid command lines and the inability to open job files (5 marks)
2. `testuqwordiply` correctly handles errors in job files and job files containing no jobs (5 marks)
3. `testuqwordiply` is able to run the required four processes for one test job (described in a single line job specification file). File descriptors need not be setup correctly. Multiple valid single-job files and command line arguments will be tested. (5 marks)
4. `testuqwordiply` constructs the necessary pipes and stdin redirections for the four processes in one test job (described in a single line job specification file). Multiple valid single-job files and command line arguments will be tested. (5 marks)
5. `testuqwordiply` correctly implements sequential testing (including reporting results and handling job files with comments and blank lines) (10 marks)
6. `testuqwordiply` correct implements parallel testing (including reporting results and handling job files with comments and blank lines) (10 marks)
7. `testuqwordiply` correctly implements quiet mode (for both sequential and parallel testing) (5 marks)
8. `testuqwordiply` correctly implements test interruption (signal handling) (5 marks)
9. `testuqwordiply` correctly closes unnecessary file descriptors in child processes (5 marks)
10. `testuqwordiply` frees all allocated memory prior to exit (original process, not children) (5 marks)

Some functionality may be assessed in multiple categories, e.g. the ability to launch multiple test jobs sequentially and in parallel must be working to fully test more advanced functionality such as checking for unnecessary file descriptors being closed and for memory to be freed.

Functionality marks (out of 10) will be assigned for `uqcmp` in the following categories (CSSE7231 only):

11. `uqcmp` correctly rejects invalid command lines (2 marks)
12. `uqcmp` correctly handles file descriptor 3 and/or 4 not being open (2 marks)
13. `uqcmp` correctly reports matching data (3 marks)
14. `uqcmp` correctly reports non-matching data (3 marks)

Style Marking

Style marking is based on the number of style guide violations, i.e. the number of violations of version 2.3 of the CSSE2310/CSSE7231 C Programming Style Guide (found on Blackboard). Style marks will be made up of two components – automated style marks and human style marks. These are detailed below.

You should pay particular attention to commenting so that others can understand your code. The marker's decision with respect to commenting violations is final – it is the marker who has to understand your code. To satisfy layout related guidelines, you may wish to consider the `indent(1)` tool. Your style marks can never be more than your functionality mark – this prevents the submission of well styled programs which don't meet at least a minimum level of required functionality.

You are encouraged to use the `style.sh` tool installed on `moss` to style check your code before submission. This does not check all style requirements, but it will determine your automated style mark (see below). Other elements of the style guide are checked by humans.

All `.c` and `.h` files in your submission will be subject to style marking. This applies whether they are compiled/linked into your executable or not[10].

[9] Valgrind tests for memory leaks (category 10) will be allowed to run for longer, as will tests that run many jobs sequentially.

[10] Make sure you remove any unneeded files from your repository, or they will be subject to style marking.

Automated Style Marking (5 marks)

Automated style marks will be calculated over all of your `.c` and `.h` files as follows. If any of your submitted `.c` and/or `.h` files are unable to be compiled by themselves then your automated style mark will be zero (0). (Automated style marking can only be undertaken on code that compiles. The provided `style.sh` script checks this for you.)

If your code does compile then your automated style mark will be determined as follows: Let

- W be the total number of distinct compilation warnings recorded when your `.c` files are individually built (using the correct compiler arguments)
- A be the total number of style violations detected by `style.sh` when it is run over each of your `.c` and `.h` files individually[11].

Your automated style mark S will be

$$S = 5 - (W + A)$$

If $W + A \geq 5$ then S will be zero (0) – no negative marks will be awarded. Note that in some cases `style.sh` may erroneously report style violations when correct style has been followed. If you believe that you have been penalised incorrectly then please bring this to the attention of the course coordinator and your mark can be updated if this is the case. Note also that when `style.sh` is run for marking purposes it may detect style errors not picked up when you run `style.sh` on moss. This will not be considered a marking error – it is your responsibility to ensure that all of your code follows the style guide, even if styling errors are not detected in some runs of `style.sh`. You can check the result of Gradescope style marking soon after your Gradescope submission – when its test suite completes running.

Human Style Marking (5 marks)

The human style mark (out of 5 marks) will be based on the criteria/standards below for "comments", "naming" and "other". The meanings of words like *appropriate* and *required* are determined by the requirements in the style guide. Note that functions longer than 50 lines will be penalised in the automated style marking. Functions that are also longer than 100 lines will be further penalised here.

Comments (2.5 marks)

Mark	Description
0	The majority (50%+) of comments present are inappropriate OR there are many required comments missing
0.5	The majority of comments present are appropriate AND the majority of required comments are present
1.0	The vast majority (80%+) of comments present are appropriate AND there are at most a few missing comments
1.5	All or almost all comments present are appropriate AND there are at most a few missing comments
2.0	Almost all comments present are appropriate AND there are no missing comments
2.5	All comments present are appropriate AND there are no missing comments

Naming (1 mark)

Mark	Description
0	At least a few names used are inappropriate
0.5	Almost all names used are appropriate
1.0	All names used are appropriate

[11] Every `.h` file in your submission must make sense without reference to any other files, e.g., it must `#include` any `.h` files that contain declarations or definitions used in that `.h` file.

Other (1.5 marks)

Mark	Description
0	One or more functions is longer than 100 lines of code OR there is more than one global/static variable present inappropriately OR there is a global struct variable present inappropriately OR there are more than a few instances of poor modularity (e.g. repeated code)
0.5	All functions are 100 lines or shorter AND there is at most one inappropriate non-struct global/static variable AND there are at most a few instances of poor modularity
1.0	All functions are 100 lines or shorter AND there are no instances of inappropriate global/static variables AND there is no or very limited use of magic numbers AND there is at most one instance or poor modularity
1.5	All functions are 100 lines or shorter AND there are no instances of inappropriate global/static variables AND there is no use of magic numbers AND there are no instances of poor modularity

SVN Commit History Marking (5 marks)

Markers will review your SVN commit history for your assignment up to your submission time. This element will be graded according to the following principles:

- Appropriate use and frequency of commits (e.g. a single monolithic commit of your entire assignment will yield a score of zero for this section)
- Appropriate use of log messages to capture the changes represented by each commit. (Meaningful messages explain briefly what has changed in the commit (e.g. in terms of functionality) and/or why the change has been made and will be usually be more detailed for significant changes.)

The standards expected are outlined in the following rubric:

Mark (out of 5)	Description
0	Minimal commit history – only one or two commits OR all commit messages are meaningless.
1	Some progressive development evident (three or more commits) AND at least one commit message is meaningful.
2	Progressive development is evident (multiple commits) AND at least half the commit messages are meaningful.
3	Multiple commits that show progressive development of ALL functionality (e.g. no large commits with multiple features in them) AND at least half the commit messages are meaningful.
4	Multiple commits that show progressive development of ALL functionality AND meaningful messages for all but one or two of the commits.
5	Multiple commits that show progressive development of ALL functionality AND meaningful messages for ALL commits.

Total Mark

Let

- F be the functionality mark for your assignment (out of 60 for CSSE2310 students or out of 70 for CSSE7231 students).
- S be the automated style mark for your assignment (out of 5).
- H be the human style mark for your assignment (out of 5).
- C be the SVN commit history mark (out of 5).
- V is the scaling factor (0 to 1) determined after interview(s) (if applicable – see the Student Conduct section above) – or 0 if you fail to attend a scheduled interview without having evidence of exceptional circumstances impacting your ability to attend.

Your total mark for the assignment will be:

$$M = (F + \min\{F, S + H\} + \min\{F, C\}) \times V$$

out of 75 (for CSSE2310 students) or out of 85 (for CSSE7231 students).

In other words, you can't get more marks for style or SVN commit history than you do for functionality. Pretty code that doesn't work will not be rewarded!

Late Penalties

Late penalties will apply as outlined in the course profile.

Specification Updates

Any errors or omissions discovered in the assignment specification will be added here, and new versions released with adequate time for students to respond prior to due date. Potential specification errors or omissions can be discussed on the discussion forum or emailed to `csse2310@uq.edu.au`.