The University of Queensland
School of Electrical Engineering and Computer Science

CSSE7231 – Semester 1, 2025
Assignment 3 (Version 1.1)

Marks: 85
Weighting: 15%
Due: 3:00pm Friday 9 May, 2025

This specification was created for the use of Andrew WILSON (s4828041) only.
Do not share this document. Sharing this document may result in a misconduct penalty.
Note that line numbers vary from those in the CSSE2310 specification.
Specification changes since version 1.0 are shown in red and are summarised at the end of the document.

Introduction

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

You are to create a program (called `uqparallel`) which implements a subset of the features of GNU `parallel` – which allows a series of tasks to be parallelised (up to some maximum number of parallel tasks).

The assignment will also test your ability to code to a particular programming style guide, and to use a revision control system appropriately.

Student Conduct

This section is unchanged from assignment one – but you should remind yourself of the referencing requirements.

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 person'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 usage and referencing rules for **all code committed to your SVN repository** (not just the version that you submit):

Code Origin	Usage/Referencing
Code provided by teaching staff this semester Code provided to you in writing **this semester** by CSSE7231 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).	**Permitted** 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 wrote this semester for this course Code you have personally written this semester for CSSE7231 (e.g. code written for A1 reused in A3) – provided you have not shared or published it.	**Permitted** May be used freely without reference. (This assumes that no reference was required for the original use.)

Code Origin	Usage/Referencing
Unpublished code you wrote earlier Code you have personally written in a previous enrolment in this course or in another UQ course or for other reasons and where that code has not been shared with any other person or published in any way.	**Conditions apply, references required** 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 from `man` pages on `moss` Code examples found in `man` pages on `moss`. (This does not apply to code from `man` pages found on other systems or websites unless that code is also in the `moss` `man` page.)	
Code and learning from AI tools Code written by, modified by, debugged by, explained by, obtained from, or based on the output of, an artificial intelligence tool or other code generation tool that you alone personally have interacted with, without the assistance of another person. This includes code you wrote yourself but then modified or debugged because of your interaction with such a tool. It also includes code you wrote where you learned about the concepts or library functions etc. because of your interaction with such a tool. It also includes where comments are written by such a tool – comments are part of your code.	**Conditions apply, references & documentation req'd** May be used provided you understand the code AND the source of the code or learning is referenced in a comment adjacent to that code (in the required format – see the style guide) 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. (All of your interactions with the tool must be captured.) The file must be committed to the repository at the same time as any code derived from such a tool. If such code is used without appropriate referencing and without inclusion of the `toolHistory.txt` file then this will be considered misconduct. See the detailed AI tool use documentation requirements on Blackboard – this tells you what must be in the `toolHistory.txt` file.
Code copied from sources not mentioned above Code, in any programming language: • copied from any website or forum (including StackOverflow and CSDN); • copied from any public or private repositories; • copied from textbooks, publications, videos, apps; • copied from code provided by teaching staff only in a previous offering of this course (e.g. previous assignment one solution); • written by or partially written by someone else or written with the assistance of someone else (other than a teaching staff member); • written by an AI tool that you did not personally and solely interact with; • written by you and available to other students; or • from any other source besides those mentioned in earlier table rows above.	**Prohibited** 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. This prohibition includes code written in other programming languages that has been converted to C.
Code that you have learned from Examples, websites, discussions, videos, code (in any programming language), etc. that you have learned from or that you have taken inspiration from or based any part of your code on but have not copied or just converted from another programming language. This includes learning about the existence of and behaviour of library functions and system calls that are not covered in class.	**Conditions apply, references required** May be used provided you do not directly copy code AND you understand the code AND the source of the code or inspiration or learning 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.

You must not share this assignment specification with any person (other than course staff), organisation, website, etc. 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 many of these sites and many cooperate with us in misconduct investigations. You are permitted to post small extracts of this document to the course Ed Discussion forum for the purposes of seeking or providing clarification on this specification.

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://eecs.uq.edu.au/current-students/guidelines-and-policies-students/student-conduct`.

Specification

The `uqparallel` program will allow a user to run a series of tasks in parallel – with the task commands and/or their arguments coming from the command line, a specified file or from `stdin`. `uqparallel` must also be able to form a pipeline of those commands (the output of one becomes the input of the next, etc.) and limit the number of processes running in parallel. The tasks may involve the same command (specified on the command line) with different arguments (coming from the command line, `stdin`, or a specified file); or the tasks may each involve different commands.

Command Line Arguments

Your `uqparallel` program is to accept command line arguments as follows:

`./uqparallel [--pipe] [--exit-on-error] [--joblimit` *`n`*`] [--dry-run] [--argsfile` *`argument-file`*`] [`*`cmd`* `[`*`fixed-args`* `...]] [::: `*`per-task-args`* `...]`

The square brackets (`[]`) indicate optional arguments or groups of arguments. The *italics* indicate placeholders for user-supplied arguments. An ellipsis (...) indicates the previous argument can be repeated. Note that the option arguments (those beginning with "`--`" and their associated value argument, if any) can be in any order. The other arguments must follow the option arguments.

The meaning of the arguments is as follows. Further details of the expected behaviour are provided below.

- `--exit-on-error` – if this option argument is present then `uqparallel` will stop executing tasks if any previous task fails (i.e. doesn't exit normally with status 0)
- `--pipe` – if this option argument is present then the tasks must execute as a pipeline, i.e. the output of the first task is piped into the input of the second and so on.
- `--argsfile` *`argument-file`* – if this option argument and value are present then the given *`filename`* is read to obtain the arguments for each task (or the commands and arguments if no command is specified on the command line)
- `--joblimit` *`n`* – if this option argument and value are present then the number of tasks executed in parallel must not exceed *`n`* – where *`n`* must be an integer in the range 1 to 120 inclusive. (If this option is not specified then the maximum is 120.)
- `--dry-run` – if this option argument is present then the tasks are not executed, but the commands and arguments for each task will be printed to `stdout`.
- *`cmd`* `[`*`fixed-args`* `...]` – if the *`cmd`* is present (immediately after any option arguments) then this command (and the arguments with it) will form the start of the command line for every task to be executed. The given command cannot begin with `--`. (Such an argument would be assumed to be an option argument.)
- `:::` *`per-task-args`* `...` – if `:::` is present on the command line (after any option arguments and any command and fixed arguments) then the arguments following `:::` will determine the number of tasks to be executed and will be provided as the last command line argument to each task in turn. This element may not be present if `--argsfile` is present.

Some examples of how the program might be run include the following[1]. Assume that the file `./one` has 3 lines (containing `-c`, `-l`, and `-w`); that the file `./two` has 3 lines (containing `cat /etc/services`, `grep tcp`, and `wc -l`) and that the file `./three` has 3 lines (containing `ps -Us4828041`, `ps -Uuser2`, `ps -Uuser3`)

1. `./uqparallel ls -a ::: /etc /usr`
   - will execute 2 tasks (in parallel): `ls -a /etc` and `ls -a /usr`
2. `./uqparallel --argsfile ./one wc /etc/motd`
   - will execute 3 tasks (in parallel): `wc /etc/motd -c`, `wc /etc/motd -l`, `wc /etc/motd -w`
3. `./uqparallel --argsfile ./two --pipe`

[1] This is not an exhaustive list and does not show all possible combinations of arguments.

- will execute the pipeline: `cat /etc/services | grep tcp | wc -l`

4. `./uqparallel --joblimit 1 ::: whoami uname uptime`
   - will execute 3 tasks (sequentially): `whoami`, `uname`, and `uptime`
5. `./uqparallel --argsfile ./three`
   - will execute 3 tasks (in parallel): `ps -Us4828041`, `ps -Uuser2`, `ps -Uuser3`
6. `./uqparallel --argsfile ./three --dry-run`
   - will print 3 tasks (over 3 lines, each prefixed by the job number – counting from 1):
     ```
     1: ps -Us4828041
     2: ps -Uuser2
     3: ps -Uuser3
     ```

More details on the expected behaviour of the program are provided later in this document.

Prior to doing anything else, your program must check the command line arguments for validity. If the program receives an invalid command line then it must print the following (as a single line)

```
Usage: ./uqparallel [--pipe] [--exit-on-error] [--joblimit n] [--dry-run]
[--argsfile argument-file] [cmd [fixed-args ...]]  [::: per-task-args ...]
```

to standard error (terminated with a newline), and exit with an exit status of 10.

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

- Both the `--argsfile` option argument and `:::` are present on the command line. (Arguments can only be taken from one source: the command line, the file specified after `--argsfile`, or (if neither are specified) then `stdin`.)
- `--pipe` is specified on the command line but neither `--argsfile` nor `:::` appears. (Arguments can not be taken from `stdin` if a pipeline is being run – the `stdin` of `uqparallel` will be used as the `stdin` of the first task in the pipeline.)
- Any value argument or command is the empty string. (It is permissible for any of the *`fixed-args`* or *`per-task-args`* to be the empty string.)
- The maximum number of jobs is not in the expected range.

Checking whether files exist or can be opened is not part of the usage checking (other than checking that filename values are not empty). This is checked after the command line is known to be valid.

File Checking

If `--argsfile` is specified on the command line with an associated filename then `uqparallel` must check that the file can be opened for reading. If not, then your program must print a defined message to standard error and exit with a defined non-zero exit status. (For this, and other incompletely specified behaviour, run the demo program to determine the expected behaviour.)

Program Behaviour

`uqparallel` will form a series of command "lines" (one for each task) and then execute them. Command lines are formed from

- the *`cmd`* and any *`fixed-args`* specified on the command line, followed by one of the following:
  - a single *`per-task-arg`* from the command line (if `:::` is given on the command line);
  - arguments extracted from a single line of the file whose name is given after `--argsfile`; or
  - arguments extracted from a single line of `stdin` (if neither `:::` nor `--argsfile` is present on the command line).

It is permissible for no *`cmd`* and *`fixed-args`* to be specified on the command line. (See the example on line 87 above.)

The number of tasks to be run (assuming all goes well) will be one of

- the number of arguments following `:::` on the command line, or
- if no command is given on the command line, the number of non-empty lines in the file whose name is given after `--argsfile`, or the number of non-empty lines read from `stdin` prior to EOF, or
- if a command is given on the command line, then the number of lines in the file whose name is given after `--argsfile`, or the number of lines read from `stdin` prior to EOF

Parsing command lines from files or `stdin`

Command lines specified in files (named after the `--argsfile` argument) or on `stdin` may contain multiple arguments. (Each line provides the arguments for one task.) Each line should be parsed in a manner similar to a shell. Specifically, any number of space characters can be used to separate individual arguments, and if it is desired to include one or more spaces in an argument, then double quotes must be used to surround the argument. A helper function is provided to do this parsing for you – see the Provided Library section below.

Restricting the number of tasks run in parallel

`uqparallel` must try to maximise the number of tasks running parallel, i.e. start the execution of each task as soon as possible. If the `--joblimit` option argument is specified on the command line then `uqparallel` must never execute more than the specified number of tasks in parallel, i.e. never have more than this many child processes, but must still try to maximise the number of tasks running in parallel. As soon as one task finishes, `uqparallel` must immediately start the next task. (Tasks must be executed in order.)

Executing Commands

If a command is the empty string with no arguments then it is to be skipped (as if it wasn't present – this is how a blank line in the input file or on `stdin` will be treated if no *`cmd`* is given on the command line). The rest of this section does not apply.

If a command is the empty string with arguments then `uqparallel` must not attempt execution (i.e. must not fork) and must print the following message to `stderr` (terminated by a newline):

```
uqparallel: unable to execute empty command
```

If a command is not the empty string then an attempt must be made to execute it (with its associated arguments). Programs are to be found on the user's `PATH`. If a (non empty string) command is unable to be executed (i.e. the `exec*()` call returns) then the child process must terminate itself with the `SIGUSR1` signal and `uqparallel` must print the following message to `stderr` (terminated with a newline):

```
uqparallel: cannot execute "cmd"
```

where *`cmd`* is replaced by the command that could not be executed. The double quotes must be present. The arguments to the command are not printed.

If an empty or non-executable command is part of a pipeline (i.e. `--pipe` is specified on the command line) then `uqparallel` must behave as if `--exit-on-error` was specified on the command line, even if it was not. See the Exiting on Error – Advanced Functionality 1 section below.

Normal Exit

If `--exit-on-error` was not specified on `uqparallel`'s command line, then `uqparallel` must exit with the exit status of the last task run. If the last task run exited due to a signal (including `SIGUSR1`), then `uqparallel` must exit with status 78. If the last task was unable to be run due to an empty command (with arguments), then `uqparallel` must exit with status 94.

Pipelines

If the `--pipe` option argument is specified on the command line then `uqparallel` must establish a pipe between the `stdout` of each task and the `stdin` of the next task. The `stdin` of the first task will come from `uqparallel`'s `stdin`. The `stdout` of the last task will be sent to `uqparallel`'s `stdout`. If there is only one task then the `--pipe` option has no effect – other then ensuring that `stdin` can't be used to specify arguments – the task's `stdin` and `stdout` must be those of `uqparallel`.

If both `--joblimit` and `--pipe` are specified on the command line then it is possible (and permitted) that the pipeline may hang if the number of tasks is greater than that specified after `--joblimit`. This could happen if pipe buffers fill up so that a program is blocked from writing into a pipe (and so won't terminate) and it is not possible to start a task that will consume from the pipeline due to the limit on the number of tasks.

Printing commands – the `--dry-run` option

If the `--dry-run` option argument is specified on `uqparallel`'s command line, then the command line for each task must be printed to `stdout` – one per line, with each command preceded by the job number (counted

from 1). See the example on line 89. Arguments must be separated by a single space character. Arguments containing one or more space characters must be surrounded by double quotes.

If the `--pipe` option is also given on the command line then each command line printed (except the last) should be followed by a single space and the pipe(|) symbol.

Exiting on Error – Advanced Functionality 1

If `--exit-on-error` was specified on `uqparallel`'s command line, then if `uqparallel` detects execution failure, i.e. that a task did not terminate normally with execution status zero, then it (a) must not run any further tasks, (b) if any tasks are still running (i.e. have not been reaped yet – whether they are running or not) then it must immediately send a `SIGTERM` signal to those tasks, or if no tasks are running then it must exit, and (c) if any `SIGTERM` signals were sent, then it must wait for those tasks to terminate and exit as soon as all have terminated, or, if any tasks are still running one second after sending the `SIGTERM` signal(s) or after the last task termination (whichever is later), then it must send a `SIGKILL` signal to those remaining tasks and exit.

If `uqparallel` is exiting due to the detection of execution failure and tasks remain unrun[2] and/or running tasks had to be terminated (i.e. at least one `SIGTERM` was sent), then `uqparallel` must print the following message to `stderr` (terminated with a newline):

```
uqparallel: aborting because of execution failure
```

(If execution failure is detected but no tasks were terminated and none are still to be run then this message is not printed.)

If execution failure was detected due to the normal exit of a task with a non-zero exit status, then `uqparallel` must exit with that status. If execution failure was detected due to a task being terminated by a signal, then `uqparallel` must exit with status 78. If execution failure was detected due to an empty command being part of a pipeline, then `uqparallel` must exit with status 94.

Note that `uqparallel` should check if any previous tasks have finished (and reap them) before starting each task, however, this does not mean that `uqparallel` will detect execution failure immediately. For example, all tasks may have been started (or attempted to be started) prior to `uqparallel` becoming aware that a task has exited/finished. Similarly, if `uqparallel` is reading commands/arguments from `stdin`, it's possible that `uqparallel` won't detect an execution failure on a previous command until after it has read the next command from `stdin` some time later.

Interrupting `uqparallel`– Advanced Functionality 2

If `uqparallel` receives a `SIGINT` signal (as usually sent by pressing Ctrl-C[3]) then it (a) must not run any further tasks, (b) if any tasks are still running then it must immediately send a `SIGINT` signal to those tasks, or if no tasks are running then it must exit, and (c) if any `SIGINT` signals were sent, then it must wait for those tasks to terminate and exit as soon as all have terminated, or, if any tasks are still running one second after sending the `SIGINT` signal(s) or after the last task termination (whichever is later), then it must send a `SIGKILL` signal to those remaining tasks and exit.

If `uqparallel` is exiting with tasks remaining unrun and/or running tasks had to be terminated (i.e. at least one `SIGINT` was sent), then `uqparallel` must print the following message to `stderr` (terminated with a newline):

```
uqparallel: aborting because of interruption
```

(If no tasks were terminated and none are still to be run then this message is not printed.)
`uqparallel` must then exit with status 15.

Redirecting `stdout` and `stderr` – CSSE7231 functionality

If arguments are being read from a file specified after `--argsfile` or from `stdin` (i.e. not from the command line after `:::`), then file redirection arguments are to be supported. Specifically, if an argument (anywhere after the command name) of the form

`>filename`

is present, then the `stdout` of that task is to be saved to the named file. If the given *`filename`* is the empty string or can not be opened for writing then `uqparallel` must print the following message to `stderr` (terminated with a newline):

[2] If arguments are being read from `stdin` and EOF has not been detected then it can be assumed that tasks remain unrun.

[3] Remember from class that Ctrl-C will send `SIGINT` to the whole foreground process group. You should test this functionality by using `kill` to send a `SIGINT` just to `uqparallel`.

`uqparallel: cannot write to "`*`filename`*`"`

where *`filename`* is replaced by the characters after `>` in the argument. `uqparallel` must not execute the task and treat this as a failure to execute the command.

Similarly, if an argument (anywhere after the command name) of the form

`2>`*`filename`*

is present, then the `stderr` of that task is to be saved to the named file. If the given *`filename`* is the empty string or can not be opened for writing then `uqparallel` must print the following message to `stderr` (terminated with a newline):

`uqparallel: cannot write to "`*`filename`*`"`

where *`filename`* is replaced by the characters after `>` in the argument. `uqparallel` must not execute the task and treat this as a failure to execute the command.

Both types of redirection can be present in a command line. These arguments do not need to appear at the end of the command line. If more than one redirection is present in the arguments then at most one error message will be printed even if more than one file can't be opened for writing. This will be for the first unopenable file in the redirection arguments. If more than one redirection of the same output stream appears in the command line (to openable files) then the last redirection will take effect. If more than one task redirects its `stdout` and/or `stderr` to the same file then the results may be unpredictable as to which task's output ends up in the file. (This is fine). Files must be created if they don't exist (with at least read and write permissions for the owner) and truncated if they don't exist.

Note that if the `--pipe` command line argument is specified then `stdout` redirection in an argument is to be ignored (and no attempt is made to open the file). The argument should be deleted (not passed to the command) and the task's `stdout` should be redirected to the next task's `stdin` as per the expected pipeline behaviour. `stderr` redirection is still to take place for pipelined commands (if specified in the arguments).

Other Requirements

Your program must also meet all of the following requirements:

- `uqparallel` must free all dynamically allocated memory before exiting. This requirement to free memory does not apply to child processes of `uqparallel`, only to the original process.
- Programs exec'd in child processes of `uqparallel` must not inherit any unnecessary open file descriptors opened by `uqparallel`. (Open file descriptors that `uqparallel` inherits from its parent and that are passed to a child must remain open in the child.)
- `uqparallel` is not to leave behind any orphan processes (i.e. when `uqparallel` exits normally then none of its children must still be running). `uqparallel` is also not to leave behind any zombie processes – dead children should be reaped quickly.
- `uqparallel` must not busy wait, i.e. it should not repeatedly check for something in a loop using a non-blocking call or a short sleep. (Reaping dead children in a loop is fine provided your program doesn't keep trying to reap dead children once it knows there are currently no more dead children.)
- All tasks run by `uqparallel` must be direct children of `uqparallel`, i.e., the use of grandchild processes is not permitted.
- When `--pipe` is not specified on the command line, the `stdout` of each task must inherited from `uqparallel`.
- No tasks run by `uqparallel` must ever output anything to `uqparallel`'s `stderr`. If `stderr` redirection is implemented then a task must save that task's stderr to the nominated file as described above, otherwise a task's `stderr` must be discarded.
- `uqparallel` must never create any temporary files.
- This specification is incomplete. For those aspects not completely specified here, your program must exactly match the behaviour of `demo-uqparallel` on `moss`.

We will not test for unexpected system call or library function failures in an otherwise correctly-implemented program including those calls that allocate resources (`fork()`, `malloc()` etc.). Your program can behave however it likes in these cases, including crashing.

Provided Library

A library has been provided to you with the following function which your program may use:

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

See the `man` page on `moss` for details.

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 3 of the CSSE2310/CSSE7231 C programming style guide available on the course Blackboard site. Your submission must also comply with the *Documentation required for the use of AI tools* if applicable.

Note that a single global variable of type `bool` may be used in your assignment – for the implementation of signal handling. Any other use of global variables will be heavily penalised – see style marking details on page 12.

Hints

1. Review the provided assignment one sample solutions to see what you can learn from them. You may freely use code from these solutions without reference.
2. Review and understand the code examples covered in the week 6 and 7 lectures. You may freely use code from these without reference.
3. Complete the week 6 and 7 Ed Lessons[4]. These lessons cover important functionality and system calls that will be needed in this assignment. You can freely use code from these and other Ed Lessons.
4. A demo program is available on `moss` that implements the required functionality: `demo-uqparallel`. Make sure you try it out to see how your program is expected to behave.
5. A process can send a signal to another process with the `kill(2)` system call or it can send a signal to itself with the `raise(3)` function.
6. Remember that in pipeline mode (when implementing `--pipe` functionality), the number of pipes needed will be one less than the number of commands to be run (and that in pipeline mode, the number of jobs is known from the start).
7. Use of the `SA_RESTART` flag when establishing a signal handler will prevent a blocking system call (e.g. reading from `stdin` or `waitpid()`) from returning immediately after the signal handler runs, and prevent your program taking immediate action in response to the signal.
8. To implement a one second delay while waiting for a child (or children) to terminate, consider using the `sigtimedwait(2)` system call to wait for a `SIGCHLD` signal. Use of the `*sleep()` functions is forbidden.
9. For a given process, you can examine the file descriptors that it has open by running
   ```
   ls -l /proc/PID/fd
   ```
   where PID is replaced by the process ID.
10. You can use the `--trace-children=no` and `--child-silent-after-fork=yes` options to `valgrind` when checking for memory leaks. This will look only at `uqparallel` and ignore child processes.

Suggested Approach

It is suggested that you follow the following steps. Test your program at each stage and commit to your SVN repository frequently. Note that the specification text above and the behaviour of the demo program are the definitive descriptions of the expected program behaviour. The list below does not cover all required functionality.

1. Write a program that checks the command line arguments and exits with a usage error if they are invalid.

[4] These lessons require at least the week 2 lessons to be completed first.

2. Implement the `--dry-run` functionality next – just constructing and printing the command lines that will be executed. Do this for
   - arguments obtained from the command line;
   - arguments obtained from a given file;
   - arguments obtained from `stdin`.
3. Implement the execution functionality – running all of the commands in parallel.
4. Add reaping functionality to obtain the exit statuses of the commands that are run.
5. Add code to support the `--joblimit` functionality – limiting the number of tasks that can run in parallel.
6. Add support for the `--pipe` functionality.
7. Implement remaining functionality as required ...

Forbidden Functions, Statements etc.

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

- `goto`
- `#pragma`
- `gcc` attributes (other than the possible use of `__attribute__((unused))` as described in the style guide)

You must not use any of the following C functions. If you do so, you will get zero (0) marks for any test case that calls the function.

- `longjmp(3)` and equivalent functions
- `system(3)`
- `popen(3)`
- `mkfifo(3)` or `mkfifoat(3)`
- `signal(2)`, `sigpending(2)`, `sigqueue(3)`, `sigwaitinfo(2)`, `sigsuspend(2)`
- `sleep(3)`, `usleep(3)`, `nanosleep(2)`
- Any `pthread*` functions
- Any `sem*` semaphore functions
- Functions described in the man page as non standard, e.g. `strcasestr(3)`. Standard functions will conform to a POSIX standard – often listed in the "CONFORMING TO" section of a man page. Note that `getopt_long(3)` and `getopt_long_only(3)` are an exception to this – these functions are permitted if desired.

The use of comments to control the behaviour of `clang-format` and/or `clang-tidy` (e.g., to disable style checking) will result in zero marks for automatically marked style.

Testing

You are responsible for ensuring that your program operates according to the specification. You are encouraged to test your program on a variety of scenarios. A variety of programs will be provided to help you in testing:

- A demonstration program (called `demo-uqparallel`) that implements the correct behaviour is available on `moss`. You can test your program with a set of command line arguments and also run the same test (perhaps with a different output filename) with `demo-uqparallel` to check that you get the same output. You can also use `demo-uqparallel` to check the expected behaviour of the program if some part of this specification is unclear.
- A test script will be provided on `moss` that will test your program against a subset of the functionality requirements – approximately 50% of the available functionality marks. The script will be made available about 10 to 14 days before the assignment deadline and can be used to give you some confidence that you're on the right track. The "public tests" in this test script will not test all functionality and you should be sure to conduct your own tests based on this specification. The "public tests" will be used in marking, along with a set of "private tests" that you will not see.

- The Gradescope submission site will also be made available about 10 to 14 days prior to the assignment deadline. Gradescope will run the test suite immediately after you submit. When this is complete[5] you will be able to see the results of the "public tests". You should check these test results to make sure your program is working as expected. Behaviour differences between `moss` and Gradescope may be due to memory initialisation assumptions in your code, so you should allow enough time to check (and possibly fix) any issues after submission.

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`, compiled programs).

Your program (named `uqparallel`) must build on `moss.labs.eait.uq.edu.au` and in the Gradescope environment with:

```
make
```

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

```
-Wall -Wextra -pedantic -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.

If any errors result from the `make` command (i.e. the `uqparallel` executable can not be 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).

Your program must not invoke other programs or use non-standard headers/libraries.

Your assignment submission must be committed to your Subversion repository under

```
svn+ssh://source.eait.uq.edu.au/csse2310-2025-sem1/csse2310-s4828041/trunk/a3
```

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) 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

`s4828041_csse2310_a3_`*`timestamp`*`.zip`

where *`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 and also to confirm your use (or not) of AI tools to help you.

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.

Multiple submissions to Gradescope are permitted. We will mark whichever submission you choose to "activate" – which by default will be your last submission, even if that is after the deadline and you made

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

[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.

submissions before the deadline. Any submissions after the deadline[7] will incur a late penalty – see the CSSE7231 course profile for details.

Marks

Marks will be awarded for functionality and style and documentation. Marks may be reduced if you attend an interview about your assignment and you are unable to adequately respond to questions – see the CSSE7231 Student Interviews section below.

Functionality (70 marks for CSSE7231)

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. Not all features are of equal difficulty.

Partial marks will be awarded for partially meeting the functionality requirements. A number of tests will be run for each marking category listed below, testing a variety of scenarios. Your mark in each category will be proportional (or approximately proportional) to the number of tests passed in that category.

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 execute a program then we can not determine if your program can set up a pipeline correctly. Your tests must run in a reasonable time frame, which could be as short as a few seconds for usage checking to many tens of seconds when `valgrind` is used to test for memory leaks. If your program takes too long to respond, then it will be terminated and you will earn no marks for the functionality associated with that test.

Exact matching of output messages and output files are used for functionality marking. Strict adherence to this specification and matching behaviour of the demo program are critical to earn functionality marks.

The markers will make no alterations to your code (other than to remove code without academic merit).

Marks will be assigned in the following categories.

1. Program correctly handles invalid command lines (usage errors) (4 marks)
2. Program correctly handles inability to read argument file specified on command line (1 mark)
3. Program correctly handles no tasks to run (2 marks)
4. Program correctly prints tasks (i.e. implements `--dry-run` functionality) (4 marks)
5. Program correctly runs tasks specified only on the command line (no pipeline, no task limit, no errors) (4 marks)
6. Program correctly runs tasks with commands/arguments obtained from the file specified on the command line (no pipeline, no task limit, no errors) (4 marks)
7. Program correctly runs tasks with commands/arguments obtained from `stdin` (no pipeline, no task limit, no errors) (4 marks)
8. Program correctly runs a two task pipeline (no task limit, no errors) (5 marks)
9. Program correctly runs multi-task pipelines (includes handling errors) (5 marks)
10. Program correctly limits the number of tasks running, i.e. implements `--joblimit` functionality, including for pipelines) (5 marks)
11. Program correctly handles inability to execute a command (including empty commands) (excluding pipeline errors – these are covered in category 9) (5 marks)
12. Program correctly implements `--exit-on-error` functionality (including empty or non-executable commands as part of a pipeline) (5 marks)
13. Program correctly implements interruption (signal handling) for a variety of execution scenarios (5 marks)
14. Program correctly closes all unnecessary file descriptors in child processes (for a variety of execution scenarios) (3 marks)

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

15. Program operates correctly, does not busy wait, does not send signals to non-existent (reaped) processes, and frees all memory upon exit (for a variety of execution scenarios) (4 marks)
16. (CSSE7231 functionality) Program correctly handles stdout redirection (single redirection at end of command(s)) (3 marks)
17. (CSSE7231 functionality) Program correctly handles stdout redirection (multiple redirections) (2 marks)
18. (CSSE7231 functionality) Program correctly handles stderr redirection (includes multiple stderr and stdout redirections) (2 marks)
19. (CSSE7231 functionality) Program correctly handles failing stdout and/or stderr redirection (3 marks)

Some functionality may be assessed in multiple categories. For example, if your program can't correctly execute commands then it will fail most tests.

Style Marking

Style marking is based on the number of style guide violations, i.e. the number of violations of version 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. 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 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.

You are encouraged to use the `2310reformat.sh` and `2310stylecheck.sh` tools installed on `moss` to correct and/or check your code style before submission. The `2310stylecheck.sh` tool 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[8].

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). If your code uses comments to control the behaviour of `clang-format` and/or `clang-tidy` then your automated style mark will be zero. If any of your source files contain C functions longer than 100 lines of code[9] then your automated and human style marks will both be zero. If you use any global variables (other than a single flag of `bool` type for signal handling) then your automated and human style marks will both be zero.

If your code does compile and does not contain any C functions longer than 100 lines and does not use any global variables (other than a single flag of `bool` type for signal handling) and does not interfere with the expected behaviour of `clang-format` and/or `clang-tidy` 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 `2310stylecheck.sh` when it is run over each of your `.c` and `.h` files individually[10].

Your automated style mark S will be

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

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

[9] Note that the style guide requires functions to be 50 lines of code or fewer. Code that contains functions whose length is 51 to 100 lines will be penalised somewhat – one style violation (i.e. one mark) per function. Code that contains functions longer than 100 lines will be penalised very heavily – no marks will be awarded for human style or automatically marked style.

[10] 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. You can check that a header file compiles by itself by running `gcc -c` *`filename.h`* – with any other `gcc` arguments as required.

If $W + A \geq 5$ then S will be zero (0) – no negative marks will be awarded. If you believe that `2310stylecheck.sh` is behaving incorrectly or inconsistently then please bring this to the attention of the course coordinator prior to submission, e.g., it is possible the style checker may report different issues on moss than it does in the Gradescope environment. Your automated style mark can be updated if this is deemed to be appropriate. You can check the result of Gradescope style marking soon after your Gradescope submission – when the 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 "modularity". Note that if your code contains any functions longer than 100 lines or uses a global variable (other than a single flag of `bool` type for signal handling) then your human style mark is zero and the criteria/standards below are not relevant.

The meanings of words like *appropriate* and *required* are determined by the requirements in the style guide.

Comments (3 marks)

Mark	Description
0	25% or more of the comments that are present are inappropriate AND/OR at least 50% of the required comments are missing
1	At least 50% of the required comments are present AND the vast majority (75%+) of comments present are appropriate AND the requirements for a higher mark are not met
2	All or almost all required comments are present AND all or almost all comments present are appropriate AND the requirements for a mark of 3 are not met
3	All required comments are present AND all comments present are appropriate AND additional comments are present as appropriate to ensure clarity

Naming (1 mark)

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

Modularity (1 mark)

Mark	Description
0	There are two or more instances of poor modularity (e.g. repeated code blocks)
0.5	There is one instance of poor modularity (e.g. a block of code repeated once)
1	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 zip file creation 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). Progressive development is expected, i.e., no large commits with multiple features in them.
- 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, not in terms of specific numbered test cases in the test suite) 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. The mark awarded will be the highest for which the relevant standard is met.

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 almost all or all functionality AND at least two-thirds of 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 OR Multiple commits that show progressive development of almost all functionality AND meaningful messages for ALL 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 70 for CSSE7231 students).
- S be the automated style mark for your assignment (out of 5).
- $A = F + \min\{F, S\}$ (the automatically determined mark for your assignment).
- H be the human style mark for your assignment (out of 5).
- C be the SVN commit history mark (out of 5).
- V be the scaling factor (0 to 1) determined after interview (see the CSSE7231 Student Interviews section below) – 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:

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

out of 85 (for CSSE7231 students).

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

Late Penalties

Late penalties will apply as outlined in the course profile.

CSSE7231 Student Interviews

This section is unchanged from assignment one.

The teaching staff will conduct interviews with all CSSE7231 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 outlined in this assignment, the style guide, and the AI tool use documentation requirements) 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 unless there are documented exceptional circumstances that prevent you from attending. Interviews will take place in your first prac session after the assignment deadline (or your extended deadline). You must attend the session that you have been allocated to.

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.

Version 1.1

- Clarified that task "still running" means that it has not been reaped yet (line 187)
- Corrected the description of the number of tasks that will be run (lines 129 to 132)
- Clarified that category 11 marks exclude pipeline execution errors (line 457)
- Clarified what is meant by child processes not inheriting unnecessary open file descriptors (line 256)
- Clarified that empty commands should not be executed (i.e. no fork) (line 150) and clarified empty command exit status (line 165)