The University of Queensland
School of Information Technology and Electrical Engineering

CSSE2310/CSSE7231 — Semester 2, 2022
Assignment 3 (version 1.4)

Marks: 75 (for CSSE2310), 85 (for CSSE7231)
Weighting: 15%
Due: 6:00pm Friday 7th October, 2022

Specification changes are shown in red - version 1.0 to 1.1, blue - version 1.1 to 1.2, green – version 1.2 to 1.3, magenta – version 1.3 to 1.4. Changes 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, and to further develop your C programming skills with a moderately complex program.

You are to create a program called `jobthing`, which creates and manages processes according to a job specification file, and must monitor and maintain the status and input/output requirements of those processes. 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 write an additional program, `mytee` which emulates some basic functionality of the Unix `tee` command, as a further demonstration of your C programming, file and commandline handling capabilities.

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, and do not allow others to access your computer – you must keep your code secure.

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, posted on the discussion forum, or shown in class).	May be used freely without reference. (You must be able to point to the source if queried about it.)
Code you have personally written this semester for CSSE2310/7231 (e.g. code written for A1 reused in A3)	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 the source of the code is referenced in a comment adjacent to that code. (Code you have *taken inspiration from* must not be directly copied or just converted from one programming language to another.)
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.	
Other code – includes: code provided by teaching staff only in a previous offering of this course (e.g. previous A1 solution); code from websites; code from textbooks; any code written by someone else, or partially written by 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 course coordinator reserves the right to conduct interviews with 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 are not able to adequately explain your code or the design of your solution 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.

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. You should read and understand the statements on student misconduct in the course profile and on the school web-site: `https://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`

Specification – `jobthing`

`jobthing` reads a job configuration from a file whose name is provided as a command line argument. It creates processes and runs programs according to that specification, optionally connecting those processes' standard input and output either to files, or `jobthing` itself via pipes. `jobthing` then reads input from `stdin`, interpreting that input as commands that cause various actions to be taken, such as sending strings to the other processes, reporting on statistics and so on. `jobthing` must handle certain signals and take specific action upon receiving those signals.

Full details of the required behaviour are provided below.

Command Line Arguments

`jobthing` has one mandatory argument – the name of the job specification file, and also accepts several optional arguments. These arguments may appear in any order.

```
./jobthing [-v] [-i inputfile] jobfile
```

- The optional `-v` argument, if supplied, puts `jobthing` into verbose mode, causing it to emit additional debug and status information. Precise requirements are documented below.
- The optional `-i inputfile` argument is the name of a file which will be used to provide input to `jobthing` and its managed processes. If no inputfile is specified (i.e. the `-i` argument is not given), then `jobthing` is to take input from `stdin`. (Note that `-i` and `-v` are valid input file names – any string after `-i` is to be treated as an input file name.)
- The `jobfile` argument specifies the name of a file from which job information is to be read. This format is documented below. This argument is mandatory.

If the user provides invalid options, too few or too many command line arguments, including repeated arguments (e.g. `-v -v`), then `jobthing` shall emit the following usage information to `stderr`, and exit with return code 1:

```
Usage: jobthing [-v] [-i inputfile] jobfile
```

If the inputfile specified with the `-i` argument is unable to be opened for reading, then the following message should be emitted to `stderr` and `jobthing` should exit with return code 3:

```
Error: Unable to read input file
```

This should be checked prior to attempting to open the job file.

`jobthing` basic behaviour

`jobthing` reads the job specification file provided on the command line, spawning child processes and executing programs as required. In general, `jobthing` is required to maintain a constant process state, regardless of what happens to those child processes and programs. For example, if a child process is killed or terminates somehow, then, unless otherwise specified, `jobthing` is required to notice this, and re-spawn the job as required, up to the maximum number of retries specified for each job.

Depending on the contents of the jobfile, each job created by `jobthing` may have its `stdin` and `stdout` connected to a pipe (back to `jobthing`), or to a file on the filesystem.

Once `jobthing` has created the initial set of jobs, it is to take input either from `stdin`, or from the file specified with the `-i inputfile` commandline argument, one line at a time. By default each input line should be sent to each job to which `jobthing` has a pipe connection, however a line starting with the asterisk character '`*`' will be interpreted as a command.

After sending the input text to each job, `jobthing` will then attempt to read a line of input from each job (again, only those to which `jobthing` is connected by a pipe). Output received from jobs is emitted to `jobthing`'s standard output.

Upon reading `EOF` from the input (`stdin` or the input file), `jobthing` shall terminate.

`jobthing` job parsing

The job file provided to `jobthing` is a text file, with one line per job.

If `jobthing` is unable to open the job file for reading, it is to emit the following message to `stderr`, and exit with return code 2:

```
Error: Unable to read job file
```

Lines beginning with the '`#`' (hash) character are comments, and are to be ignored by `jobthing`. 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 4 separate fields delimited by the colon ('`:`') character as follows

```
numrestarts:input:output:cmd [arg1 arg2 ...]
```

where each field has the following meaning or intepretation (where "empty" means a zero length string):

- `numrestarts` – specifies how many times `jobthing` shall start or restart this job if it terminates. 0 (zero) or empty implies that `jobthing` shall restart the job every time it terminates, 1 (one) means that `jobthing` should attempt to launch the job once only upon startup – if it terminates it is not restarted. Other integers are interpreted similarly.

- `input` – empty implies that this job shall receive its standard input from a pipe connected to `jobthing`. Otherwise, the named file is to be opened for reading and connected to this job's standard input stream.

- `output` – empty implies that this job shall send its standard output to a pipe connected to `jobthing`. Otherwise, the named file is to be opened for writing (with flags `O_CREAT | O_TRUNC` and permissions `S_IWUSR | S_IRUSR`) and connected to this job's standard output stream.

- `cmd [arg1 arg2 ...]` – the name of the program to be run for this job, and optional arguments to be provided on the commandline to that program. Arguments are separated by spaces. Program names and arguments containing spaces may be specified by enclosing them in double quotes. A helper function is provided to make this easer, see the `split_space_not_quote()` function described on page 11.

Note: Individual job specifications are independent, and you do not need to consider if jobs might interact with each other (e.g. sharing input or output files etc). We will only test job specifications that have predictable and deterministic behaviour.
Note: The colon character has special meaning and will only appear in job files as a separator. You do not need to consider, nor will we test for jobfiles that contain the colon character as part of a command name or argument.
Note: See the `split_line()` function described on page 11 for an easy way to split the colon-delimited job specifications.

Following are several sample jobfiles with explanatory comments:

```
# A job, running cat, stdin/stdout connected to jobthing.  Only start 'cat' once
1:::cat
```

```
# A job, running cat, stdin/stdout connected to jobthing.  Start 'cat' a maximum of 5 times
5:::cat
```

```
# A job, running cat, stdin/stdout connected to jobthing.
# retries = 0 -> re-launch cat every time it terminates, no limit
0:::cat
```

```
# A job, running cat, take stdin from /etc/services, send stdout to foo.txt.
# Only run  'cat' once
1:/etc/services:foo.out:cat
```

```
# A job, running cat, take stdin from /etc/services, stdout connected back to jobthing
# Only run  'cat' once
1:/etc/services::cat
```

```
# two jobs, both running cat, stdin and stdout connected to jobthing.
# The first job runs cat only once, the second will restart it forever as required
1:::cat
0:::cat
```

If verbose mode is specified, for each valid job line read from the jobfile, `jobthing` should emit the following output to its `stdout`

```
Registering worker N: cmd arg1 arg2 ...
```

where

- `N` is the replaced with job number, incrementing from 1 (one)
- `cmd` is the command to be run, and `arg1`, `arg2`, ... are any arguments provided to the command. Note that there should be a single space between `cmd` and any arguments, and there should be no trailing space on this line of output

The following is an example of such output:

```
Registering worker 1: cat
Registering worker 2: tee logfile.txt
```

A job line in the jobfile is invalid if any of the following conditions are met:

- There are not precisely 4 fields separated by colons
- The ~~integer value~~ first field (`numrestarts`) is not a proper, non-negative integer (if not empty)
- The `cmd` field in the job line is empty or starts with a space

Invalid job lines are to be ignored and are not given job numbers. Further, if verbose mode is specifed on the command line, then `jobthing` shall emit the following to its `stderr`:

```
Error: invalid job specification: <jobline>
```

where `<jobline>` is replaced by the offending job specification line, e.g.

```
Error: invalid job specification: -10:0:foobar baz
```

jobthing startup phase

Once the jobfile has been read, jobs are to be created in the order they were specified in the job file.

- If an input file is specified for the job, jobthing shall attempt to open that file in read mode and the job is to have its stdin redirected from that file. Otherwise, jobthing shall create a pipe, connecting that job's stdin to the reading end of the pipe, with jobthing holding the write end of the pipe through which it will later send information.
- Similarly, if an output file is specified for the job, jobthing shall attempt to open that file in for writing and the job is to have its stdout redirected to that file. Otherwise, jobthing shall create a pipe, connecting that job's stdout to the writing end of the pipe, with jobthing holding the read end of the pipe through which it will later receive information.

If a job has an input file specified, and that file cannot be opened, jobthing shall emit the following message to stderr. The job shall be considered ~~invalid~~ unrunnable and no further handling or respawn attempts shall be made for that job. (If an output file is also specified, no attempt shall be made to open it.)

```
Error: unable to open "<filename>" for reading
```

where <filename> is the name of the file from the job specification.

Similarly, if an output file is specified and cannot be opened for writing, the job is considered ~~invalid~~ unrunnable, no further processing or spawn attempts are made, and jobthing shall emit to stderr:

```
Error: unable to open "<filename>" for writing
```

Once any input and output files and pipes are opened/created, jobthing shall spawn a new process for each runnable job, connect the stdin and stdout of the new job as required, and then exec the command line specified for the job.

If verbose mode is specified, then jobthing should emit the following to its stdout:

```
Spawning worker N
```

where N is replaced by the job number.

Important: jobthing shall ensure that all un-used file handles are closed before executing the job process. That is, the job shall have only its standard input, output and error file handles open. (Standard error is just to be inherited from jobthing.) Test scripts will check to ensure that no other file handles leak from jobthing to individual jobs.

If the child's exec call fails, the child process is to call _exit() with the return code 99. jobthing is not expected to detect a failure to exec (e.g. such jobs don't become unrunnable) – this is treated exactly the same as a successful child execution where the child immediately returned exit status 99.

jobthing operation and command format

Once jobthing has started the jobs, it should sleep for one second and then enter an infinite loop (terminated only by reading EOF on its input stream (stdin or the supplied input file) or by running out of viable workers – see descriptions below).

Each time through the loop jobthing shall perform the following operations, in the exact order specified below.

Note: Any jobs that were marked ~~invalid~~ unrunnable during the startup phase (i.e. because their input or output files could not be opened) are excluded from all handling during this main loop.

1. Check on the status of each job (in the order they were specified in the job file), report on any jobs that have terminated, and restart those which need to be restarted.
   - for any jobs that have terminated since the last check, jobthing shall generate a line of output to stdout in the following format:

     ```
     Job N has terminated with exit code M
     ```

     or

     ```
     Job N has terminated due to signal S
     ```

     depending on the reason for the job terminating. N, M and S should be substituted by the job number, exit code or signal number as appropriate.

   - close and clean up any pipes and file descriptors associated with communication to the terminated job
   - for each job that has terminated, if the total number of times it has been (re-)started is less than the maximum number specified in the job file, then the job shall be restarted in exactly the same way it was started (including input/output file redirection/pipes as required). Note that it is possible that a respawning might not be possible due to the inability to open an input or output file for reading or writing (even though that succeeded for an earlier run). In this case, `jobthing` should print the same error message as specified above (see "`jobthing` startup phase") and make no further attempts to respawn that job – i.e. the job becomes unrunnable. If the job is restarted and verbose mode is specified, then `jobthing` shall emit the following to its `stdout`:

     `Restarting worker N`

     where `N` is replaced by the job number.
   - If a job has already been restarted the maximum number of times, then it should not be restarted – the job is now unrunnable.

2. If `jobthing` determines that there are no jobs running and no further possible jobs left to (re)launch (all jobs are unrunnable), then it shall emit the following message to `stderr`, and exit with exit code 0 (zero):

   ```
   No more viable workers, exiting
   ```

   ~~Reasons for this are~~

   - ~~No jobs remain with a non-zero restart count remaining~~
   - ~~No jobs remain that have valid input or output redirection files~~

3. Read a line of input from `stdin` or the specific `jobthing` input file, and process it as a command according to the following requirements:

   - If EOF is detected, then `jobthing` shall exit with exit status 0 (zero).
   - Lines beginning with the asterisk '`*`' character are treated as commands – see below for details. After processing the command, `jobthing` shall sleep for 1 second and then return to the top of the main loop, checking job status again etc. (The same sleep/return applies to invalid commands also – in this case the "processing" of the command means outputting an error message as described below.)
   - Any other lines are sent as-is to each job to which `jobthing` has a connection (i.e. a pipe exists between `jobthing` and the job's `stdin`).
   - Data sent to each job is to be echoed to `jobthing`'s `stdout` in the following format:

     `ID<-'text'`

     where `ID` is the job ID (starting from one), and `'text'` is the line of input sent to the job, surrounded by single quotes.

4. `jobthing` shall sleep for 1 second

5. `jobthing` shall attempt to read exactly one line of input from each job to which it has a pipe connected to that job's `stdout`. Each line of output received from each job shall be emitted to `jobthing`'s `stdout` as follows:

   `ID->'text'`

   where `ID` is the job ID (starting from one), and `'text'` is the line of output received from the job, surrounded by single quotes. It is expected that jobs will have a line of output available. If a job fails to send such a line it is acceptable for `jobthing` to block until such time as a line is returned (or EOF is detected). If EOF is detected, no message is output unless verbose mode is enabled. If verbose mode is enabled then `jobthing` should output the following to `stderr`:

   `Received EOF from job N`

   where N is replaced by the job number.

6. Otherwise, `jobthing` shall repeat the loop starting back at Step 1 above.

NOTE: it is critical that you do these actions in this order, otherwise your program will behave differently and fail many tests.

`jobthing` signal handling requirements

Upon receiving `SIGHUP`, `jobthing` is to emit to its `stderr` statistics on the history and status of each valid job that was specified in the jobfile. Each line of the statistics report is of the format

`jobnum:numstarts:linesto`

where each field has the following interpretation:

- `jobnum` – the number of the job, starting from one which is the first valid job in the provided jobfile
- `numstarts` – how many times `jobthing` has attempted to start or restart the job, including the initial process creation upon startup
- `linesto` – how many lines of input `jobthing` has sent to the job. Jobs whose `stdin` is read from a file (rather than a pipe from `jobthing`) will report 0 (zero).

Note that the statistics for any given job accumulate over multiple restarts – if a job is terminated and respawned multiple times, the total number of lines sent to it over the lifetime of `jobthing` is reported. `jobthing` shall block or otherwise ignore `SIGINT` (Control-C).

`jobthing` must gracefully handle the possibility that it attempts to write information down a pipe to a job that has terminated. If this occurs, `jobthing` shall silently ignore this fact, and the terminated job and associated pipes should be cleaned up as per normal processing, with the job being restarted if appropriate.

We will not test whether system calls are restarted following a signal – e.g. a sleep resulting from a `*sleep` command can be shorter than expected if interrupted by a signal or can be resumed – either is acceptable.

`jobthing` command handling

The following table describes the commands that must be implemented by `jobthing`, and their syntax. Additional notes on each command will follow.

Command	Usage	Comments
`*signal`	`*signal jobID signum`	Send the signal (`signum` – an integer) to the given job (`jobID`).
`*sleep`	`*sleep millisec`	Sleep for `millisec` (an integer) milliseconds.

- Any invalid commands provided to `jobthing` (i.e. a command word starting with '`*`' is invalid), shall result in the following message to `stdout`:

  ```
  Error: Bad command 'cmd'
  ```

  where '`cmd`' is the offending command enclosed in single quotes. (`cmd` is the text after the '`*`' up until the first space or newline – and may be an empty string.)

- if the command is not provided the correct number of arguments, the following is emitted to `stdout`

  ```
  Error: Incorrect number of arguments
  ```

- All numerical arguments, if present, must be complete and valid numbers. e.g. "`15`" is a valid integer, but "`15a`" is not. Your program must correctly identify and report invalid numerical arguments (see details below for each command). Leading whitespace characters are permitted, e.g. " 10" is a valid number – these whitespace characters are automatically skipped over by functions like `strtol()` and `strtod()`.

- ~~Any text arguments, including strings and program names, may contain spaces if the argument is surrounded by double quotation marks, e.g. `"text with spaces"`. A line with an odd number of double quotes will be treated as though there is an additional double quote at the end of the line[1]. A helper function is provided to assist you with quote-delimited parsing, see the "Provided Library" section on page 11 for usage details.~~

[1] ~~This will not be tested~~

*signal

The `*signal` command shall cause a signal to be sent to a job. Exactly two integer arguments must be specified – the target job ID, and the signal number.

If the job ID is invalid, your program should emit the following to `stdout`:

```
Error: Invalid job
```

Reasons for a job number being invalid are:

- an invalid integer, e.g. "23a"
- an invalid job number (less than 1, greater than the total number of jobs)
- the specified job is unrunnable ~~has terminated or is otherwise invalid~~ (reached maximum number of restarts, input or output files could not be opened)

If the signal number is invalid (non-numeric, less than 1 or greater than 31) then your program should emit the following to `stdout`:

```
Error: Invalid signal
```

If all arguments are valid, the signal shall be sent to the targetted job. (There is no need to check whether the job is still running when the signal is actually sent.)

*sleep

The `*sleep` command shall cause `jobthing` to sleep for the specified number of milliseconds. Exactly one non-negative integer argument must be provided.

If the sleep duration value is invalid (not a properly formed integer, or a negative value), your program should emit the following to `stdout`:

```
Error: Invalid duration
```

If the arguments are valid, `jobthing` shall sleep for the required duration (in milliseconds).

Example `jobthing` Sessions

In this section we walk through a couple of increasingly more complex examples of `jobthing`'s behaviour. Note however 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.

Consider a config file `once_cat.txt` with following contents:

```
1:::cat
```

This defines a single job, running `cat`, to be launched once only, and `stdin` and `stdout` connected via pipes to `jobthing`. We launch `jobthing` in verbose mode and interact with the job in some simple ways. Note that text formatted in **bold** is entered and echoed on the terminal, it is not output of `jobthing` itself.

```
$./jobthing -v once_cat.txt
Registering worker ~~1:cat~~1: cat
Spawning worker 1
hello there
1<-'hello there'
1->'hello there'
this is some text
1<-'this is some text'
1->'this is some text'
```

Thus we see simple job startup, and text passing to and from the job. Next we can explore the `*signal` command (continuing the same session):

```
*signal 0 11
Error: Invalid job
*signal 1 45
Error: Invalid signal
*signal 23a 45
Error: Invalid job
*signal 1 9
WorkerJob 1 has terminated due to signal 9
No more viable workers, exiting
```

Here after several invalid `*signal` command attempts, we finally send signal 9 to worker 1, which causes its termination. `jobthing` then determined that this job should only be launched once, and that with no more ~~viable~~ runnable jobs to run it terminates.

We next consider job input and output redirection, with the file `cat_once_in_out.txt`:

```
1:/etc/services:./foo.out:cat
```

This job file runs a single job, `cat`, but it takes its standard input from `/etc/services`, and redirects its output to `./foo.out`. It runs only once.

Launching this job, in verbose mode, we see the following:

```
Registering worker 1: cat
Spawning worker 1
WorkerJob 1 has terminated with exit code 0
No more viable workers, exiting
```

In this example, we see that the process was spawned, but because its input was redirected from a file, the `cat` process ran to completion almost immediately. This was detected by `jobthing`, which identified that no further restarts should be attempted, and that no runnable jobs remained, so the program exits without pausing for any user input.

Let's now consider a more complex example, with multiple processes and different run counts. The example configuration file is `mixed_multicat.txt`:

```
0:::cat
1:::cat
2:::cat
```

Here we have three jobs, all to run `cat` with their `stdin` and `stdout` connected via pipes to `jobthing`, however each has a different number of restarts: zero (i.e. re-start endlessly), 1, and 2.

```
$./jobthing -v mixed_multicat.txt
Registering worker 1: cat
Registering worker 2: cat
Registering worker 3: cat
Spawning worker 1
Spawning worker 2
Spawning worker 3
Hello
1<-'Hello'
2<-'Hello'
3<-'Hello'
1->'Hello'
2->'Hello'
3->'Hello'
```

Entering a single line of input (`'Hello'`) has it sent to each job in turn, then that same string is returned from each job by the `cat` process.

We then kill job number 2, and send some more input:

```
*signal 2 9
WorkerJob 2 has terminated due to signal 9
```

```
Hello again
1<-'Hello again'
3<-'Hello again'
1->'Hello again'
3->'Hello again'
```

Here we see that job 2 was terminated and not restarted (its restart count was only 1), and that the subsequent input (`Hello again`) was then only sent to remaining live jobs (1 and 3). Let's send a signal to job number 3, and send some more input:

```
*signal 3 11
~~Worker~~Job 3 has terminated due to signal 11
Restarting worker 3
foobar
1<-'foobar'
3<-'foobar'
1->'foobar'
3->'foobar'
```

And again:

```
*signal 3 11
~~Worker~~Job 3 has terminated due to signal 11
baz
1<-'baz'
1->'baz'
```

After the second signal, worker 3 has finally terminated and not restarted, and the input entered at the console is sent only to worker 1. Since worker 1 has a restart count of zero (restart forever), we can send it as many signals as we like, it will continue being restarted:

```
*signal 1 5
~~Worker~~Job 1 has terminated due to signal 5
Restarting worker 1
*signal 1 6
~~Worker~~Job 1 has terminated due to signal 6
Restarting worker 1
still there?
1<-'still there?'
1->'still there?'
```

Finally, let's send `SIGHUP` to `jobthing`, and see the statistics reporting:

```
~~1:cat:2:4~~ 1:3:5
~~2:cat:1:1~~ 2:1:1
~~3:cat:2:3~~ 3:2:3
```

Here we can see how many times each job was restarted (including expired ones like jobs 2 and 3), and how many lines of input were sent to each.

Specification - `mytee` (CSSE7231 students only)

CSSE7231 students are to write an additional program, called `mytee`. `mytee` reads lines of input from `stdin`, and writes them back out to `stdout`, and also to another file whose name is provided on the command line. By default, `mytee` creates a new output file every time it is run, however this can be overridden by providing the `-a` command line option.

Command Line Arguments

`mytee` requires one mandatory argument – the name of the output file, and accepts one optional argument. These arguments may appear in any order.

```
./mytee [-a] outfile
```

- The optional `-a` argument puts `mytee` into append mode. In this mode, if the specified output file already exists, then `mytee` shall append content to it (add it at the end). Otherwise, and by default, `mytee` shall overwrite the `outfile`.
- The `outfile` argument specifies the name of the file to which the input received from `stdin` should be written.

Operation and errors

Once the required output file has been opened for writing (and possibly appending, depending on the presence or absence of the `-a` option), `mytee` shall sit in an endless loop reading input one line at a time from `stdin`. Each line of input should then be written to the output file, and also written to `stdout`. All output should be flushed immediately to ensure predictable operation.

Upon receiving `EOF` on `stdin`, `mytee` shall terminate immediately with exit code 0.

If `mytee` is run with an invalid command line (incorrect, missing or additional arguments), it shall emit the following usage message to `stderr`, and exit with return code 1:

```
Usage: mytee [-a] outfile
```

If `mytee` is unable to open the specified `outfile` for writing, it shall emit the follow to `stderr`, and return exit code 2:

```
Error: unable to open <filename> for writing
```

where the string `<filename>` is replaced with the offending filename, for example

```
Error: unable to open /etc/services for writing
```

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 2.2.0 of the CSSE2310/CSSE7231 C programming style guide available on the course Blackboard site.

Hints

1. You **may** wish to consider the use of the standard library functions `strtol()`, and `usleep()` or `nanosleep()`.
2. While not mandatory, the provided library functions will make your life a lot easier – use them!
3. The standard Unix `tee` command behaves like `cat`, but also writes whatever it receives on `stdin` to a file. This, combined with `watch -n 1 cat <filename>` in another terminal window, may be very helpful when trying to figure out if you are setting up and using your pipes correctly.

4. 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. This may be helpful to ensure you are closing all required file descriptors before executing jobs.

5. Review the lectures/contacts from weeks 6 and 7. These cover the basic concepts needed for this assignment and the code samples may be useful. Similarly, the Ed Lessons exercises for weeks 6 and 7 may be useful.

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.

1. Write small test programs to figure out the correct usage of the system calls required for each `jobthing` command – i.e. how to connect both `stdin` and `stdout` of a child process to pipes and manage access to them from the parent. (This is essentially what the week 6 and 7 Ed Lessons exercises ask you to do.)

2. Write the initial job spawning capability of `jobthing`.

3. Add the required input/output setup for each job.

4. Add the main loop functionality – implement the basic input/output functionality of `jobthing` first, and make sure you can talk to the jobs. Then extend that to add command processing as a special case.

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 `jobthing` and `mytee` (CSSE7231 only) must build on `moss.labs.eait.uq.edu.au` with:
`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[2].

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).

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-sem2-sXXXXXXX/trunk/a3`

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

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)[3]. 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 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[4] will incur a late penalty – see the CSSE2310/7231 course profile for details.

Marks

Marks will be awarded for functionality and style and documentation. Marks may be reduced if you are asked to attend an interview about your assignment and you 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 your communication with that job, or your ability to send it signals. 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 exist also. If your program takes longer than 15 seconds to run any test, 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 `jobthing` in the following categories (CSSE2310 and CSSE7231):

1. `jobthing` correctly rejects invalid command lines and handles inability to open files (3 marks)
2. `jobthing` correctly starts jobs and sets up their input and output correctly (including pipes) (includes handling comments in job files and invalid job specifications) (12 marks)

[3]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.

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

3. jobthing correctly handles jobthing standard input (or inputfile) text (non commands and invalid commands) (9 marks)
4. jobthing correctly handles and identifies the termination of child jobs and their causes (including restarting jobs when appropriate) (8 marks)
5. jobthing correctly closes all unnecessary file handles in child processes (4 marks)
6. jobthing correctly implements *signal command and argument error checking (7 marks)
7. jobthing correctly implements *sleep command and argument error checking (5 marks)
8. jobthing correctly handles SIGHUP and emits job statistics (5 marks)
9. jobthing correctly handles SIGPIPE and SIGINT as appropriate (2 marks)
10. jobthing frees all allocated memory prior to exit (when exiting under normal circumstances, i.e. EOF received on jobthing's stdin, or no viable jobs remain) (5 marks)

Note that verbose mode functionality is covered in multiple categories.

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

11. mytee correctly rejects invalid command lines (3 marks)
12. mytee correctly handles errors with output files (2 marks)
13. mytee correctly duplicates input onto stdout and the required output file (3 marks)
14. mytee correctly handles appending to output files (2 marks)

Some functionality may be assessed in multiple categories, e.g. the ability to launch jobs must be working to test more advanced functionality. Your programs must not create any files other than those possibly specified as output redirection for jobs, or files created by the jobs themselves. Doing otherwise may cause tests to fail.

Style Marking

Style marking is based on the number of style guide violations, i.e. the number of violations of version 2.2 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[5].

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)

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

- 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[6].

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`.

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

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 assessment (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:

[6] 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.

- 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 – single commit OR all commit messages are meaningless.
1	Some progressive development evident (more than one commit) OR at least one commit message is meaningful.
2	Some progressive development evident (more than one commit) AND at least one commit message is meaningful.
3	Progressive development is evident (multiple commits) AND at least half the commit messages are meaningful.
4	Multiple commits that show progressive development of all functionality AND meaningful messages for most 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, or 70 for CSSE7231).
- 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)

Your total mark for the assignment will be:

$$M = F + \min\{F, S + H\} + \min\{F, C\} + \min\{F, D\}$$

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

In other words, you can't get more marks for style or SVN commit history or documentation 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`.

Version 1.1

- Added details of what should happen when inputfile (after `-i` argument) can not be opened (and where inability to open files is covered in the marking criteria).
- Clarify what is meant by "no inputfile is specified".

Version 1.2

- Added details of spawning/restarting messages to be printed in verbose mode (to make the spec. consistent with the examples).
- Added note that respawning may not be possible because it may not be possible to open required files.
- Clarified that mark category 2 includes handling comments in job files and invalid job specifications.
- Described another type of invalid job line.
- Clarified which jobs get "registered"/numbered (valid job lines).
- Use the terms "runnable" and "unrunnable" for jobs that resulted from valid job lines but are still able to be run or not able to be run (replace previous overloaded use of "valid"/"invalid").
- Made it clear that `jobthing` does not need to handle child execution failure in any special way.
- Removed redundant text about invalid text arguments for `jobthing` commands.
- Added clarification about sending signals to jobs that may have died.
- Added note that verbose mode functionality is covered in multiple categories.
- Clarified that handling of invalid commands is covered in category 3.

Version 1.3

- Added requirement to sleep for 1 second after executing a * command
- Clarified what "empty" means in a job line and made it clear that this is a valid value for the first field

Version 1.4

- Fixed examples to match specified behaviour.
- Clarified message to be output on bad commands.
- Clarified that statistics are only reported for valid jobs.
- Clarified that invalid commands have the same one second delay after them as valid commands.
- Clarified that we won't test whether system calls are restarted following a signal to `jobthing`.
- Clarified that repeated arguments are usage errors and that `-i` and `-v` are valid input file names.