The University of Queensland
School of Information Technology and Electrical Engineering

CSSE2310/CSSE7231 — Semester 2, 2021
Assignment 3 (version 1.2 - 4 October 2021)

Marks: 75 (for CSSE2310), 85 (for CSSE7231)
Weighting: 20%
Due: 3:59pm 8 October, 2021

Specification changes since version 1.1 are shown in red and are summarised at the end of the document.

Introduction

The goal of this assignment is to further develop your C programming skills, and to demonstrate your understanding of the system calls for process creation, management and IO using `fork()`, `pipe()` etc. You are to create a program (called `jobrunner`) which reads a job specification file, creating processes, setting up pipes and input/output redirections as required. It must then monitor those processes and report accurately on their terminating conditions. Advanced functionality such as job timeouts (aborting jobs that don't complete within a certain time limit), and `jobrunner` signal handling is also required for full marks.

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

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 proceedings 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.

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 they cooperate with us in misconduct investigations.

You may use code provided to you by the CSSE2310/CSSE7231 teaching staff **in this current semester** and you may use code examples that are found in man pages on moss. If you do so, you **must** add a comment in your code (adjacent to that code) that references the source of the code. If you use code from **other** sources then this is either misconduct (if you don't reference the code) or code without academic merit (if you do reference the code). Code without academic merit will be removed from your assignment prior to marking (which may cause compilation to fail) but this will not be considered misconduct.

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

Command Line Arguments

Your program (`jobrunner`) is to accept command line arguments as follows:

```
./jobrunner [-v] jobfile [jobfile ...]
```

In other words, your program should accept one optional argument (`[-v]`), and at least one mandatory argument (a job specification file). Any number of job specification files can be provided. Their syntax and handling is described below.[1]

The `-v` option indicates that the `jobrunner` program is to operate in verbose mode, outputting extra information about its operation. The exact details of verbose mode will be described below.

The jobfile arguments are the path name(s) of plain text files containing the details of the jobs to be run – one per line. If multiple jobfiles are specified then the effect is as if they were all combined into a single file – the jobs from each jobfile are added to the `jobrunner`'s list in the order they are read.

Jobfile Syntax

Lines beginning with the '`#`' character[2] are treated as comments and are to be ignored. Empty lines, including lines with only space and/or tab characters, are to be ignored. All other lines in a jobfile must specify a job, in the following comma-separated syntax:

```
program,stdin,stdout,timeout,arg1,arg2,...
```

The meaning of these fields is described below. Fields may be optional or mandatory – if optional then the field may be empty in the job spec file.

- `program` – the name of the program to run (mandatory field). The `$PATH` should be searched for this program, or a full path may be specified, e.g. `cat` or `/bin/cat`.
- `stdin` – the specifier for where the job should receive its standard input from (mandatory field). See below for more details.
- `stdout` – the specifier for where the job standard output should be sent (mandatory field). See below for more details.
- `timeout` – the (approximate) number of seconds (as a non-negative integer without any leading or trailing spaces) that this job should be allowed to run (optional field). If the job is still running after the timeout, the job is to be sent the `SIGABRT` signal. A missing value, or value of 0 (zero) means no timeout – the process will be permitted to run for as long as necessary. See the sample jobfiles in listings 1 to 4 below for several examples of how a zero timeout can be specified. If a job does not die after receiving a `SIGABRT` signal then it should be sent a `SIGKILL` signal. See the "Advanced Functionality" section for more details.
- `arg1`,`arg2` ... – optional arguments to be passed on the commandline to the `program`. Note that an argument might be the empty string (nothing between commas, or nothing between the last comma and the end of the line).

The comma character will never appear in any of the fields in a job specification line, including filenames and arguments. Note that you should not strip any trailing or leading spaces from any fields – it is perfectly valid for program names, file names, pipe names and arguments to be strings that include spaces.

The `stdin` and `stdout` fields in a job specification can take the following values and meanings:

- '`-`' – the job shall use the same `stdin` or `stdout` as the `jobrunner` program itself
- '`@pipename`' – `jobrunner` shall create a pipe, internally referred to by the label `pipename`[3], and give either the read or write end of the pipe to this job. (If used as the `stdin` argument, the read end is given to the job; if used as the `stdout` argument, the write end is given to the job.) Pipes are named, so they

[1] The square brackets (`[]`) indicate optional arguments.

[2] i.e. the `#` character is the very first character on the line. `#` characters in any other position do not have any special meaning.

[3] Note that pipenames (after the '`@`' character) can be made up of zero or more ASCII characters – other than the null character, the comma character or the newline character. Names might include spaces, the '#' character, etc. In practice, you do not need to check the validity of a pipename - it is just the string (possibly empty) between the '`@`' character and the next comma character.

can be referred to by other jobs [4]. See the example below to clarify. Note that '`@`' must be the very first character in the field (i.e. immediately after a comma) for this field to be considered a pipename rather than a filename.

- Any other name for this field refers to a standard file on the file system, which must either be opened read-only (for `stdin`) or for writing (create, truncate – with at least read permissions for the current user) (for `stdout`). Filenames may use relative or absolute paths. The only way to specify a filename beginning with '`@`' in the current directory is to refer to it with an absolute pathname or with `./` prefixed, e.g. `./@filename`.

The `stderr` of all jobs must be suppressed (hint: what happens if you redirect `stderr` to `/dev/null`?)

Jobfile Examples

Here is a very simple jobfile, it will create a single process `ls`, its standard input and output will be the same as the `jobrunner` program, and there is no timeout:

Listing 1: Simple job specification file, no stdin/stdout redirection

```
# Spawn a process to run ls, stdin and stdout shared with jobrunner, no timeout, no args
# This is equivalent to the shell command "ls"
ls,-,-,0
```

Listing 2: Simple input and output redirection

```
# run cat, take stdin from /etc/fstab, send stdout to out.txt, no timeout, no args
# this is equivalent to the shell command "cat < /etc/fstab > out.txt"
cat,/etc/fstab,out.txt
```

Listing 3: Program arguments and commandline redirection

```
# run cat, share stdin with jobrunner, send stdout to out.txt, no timeout, one argument
# this is equivalent to the shell command "cat /etc/mtab > out.txt"
cat,-,out.txt,,/etc/mtab
```

Each line in the jobfile creates a new job – they are created independently (except for pipes, see later).

Listing 4: Multiple jobs in a jobfile

```
# create two jobs, one sleeping for 10 seconds and the other
# running "ls" and directing output to ls.out
ls,-,ls.out,
sleep,-,-,0,10
```

Next we see a simple use of piping. Pipes are indicated with `@label`, either in the `stdin` or `stdout` position in the job specification.

Listing 5: Simple piping

```
# run ls, pipe its input into sort, redirect that to ls.out.  No timeouts
# this is equivalent to the shell command "ls | sort > ls.out"
ls,-,@pipe1,0
sort,@pipe1,ls.out,0
```

Any number of pipes may be created, for example to create a chain of pipes:

Listing 6: A chain of pipes

```
# run ls, pipe its output into cat, pipe that to sort, redirect that to ls.out.  No timeouts
# this is equivalent to the shell command "ls | /usr/bin/cat | sort > ls.out"
ls,-,@alice
sort,@bob,ls.out,
/usr/bin/cat,@alice,@bob,0
```

[4] These are NOT named pipes (fifo special files) in the operating system sense – the names are only known to `jobrunner` and not to the operating system.

Note that a pipe must have exactly one reader and exactly one writer. If a pipe is specified that has either zero or more than one readers or writers, that pipe is invalid, and any jobs referring to that pipe must not be run. For example:

Listing 7: Piping errors

```
# run ls, pipe its output to... where?  This is a broken jobspec and must be ignored
ls,-,@pipe1,0
# This part of the jobfile is valid and must still be executed (cat < /etc/mtab > cat.out)
cat,/etc/mtab,cat.out,0
```

If a non-zero timeout is specified, then `jobrunner` must terminate the job if it is still running after that many seconds.

Listing 8: Using timeouts

```
# Sleep for 10 seconds, but with a 5 second timeout.
# This process should receive SIGABRT from jobrunner
sleep,-,-,5,10
```

Program Operation

The program is to operate as follows:

Command Line Processing

- If the program receives an invalid command line then it must print the message:
  `Usage: jobrunner [-v] jobfile [jobfile ...]`
  to standard error, and exit with an exit status of 1.

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

  - no command line arguments are specified
  - `-v` is the only argument given (i.e. no jobfiles are specified)
  - `-v` appears in the command line in other than the first position[5]

  Usage errors must be checked before attempting to open any jobfiles.

- If a given jobfile filename does not exist or can not be opened for reading, your program should print the message:
  `jobrunner: file "`*`filename`*`" can not be opened`
  to standard error, and exit with an exit status of 2. (The italicised *`filename`* is replaced by the actual filename i.e. the full argument given on the command line. The double quotes must be present.) Your program must exit immediately on finding the first named jobfile that cannot be opened – you do not need to check subsequent jobfiles specified on the command line. No jobs should be run.

Jobfile Parsing

- If a jobfile contains an invalid (i.e. syntactically incorrect) line, your program must print the message:
  `jobrunner: invalid job specification on line `*`linenum`*` of "`*`filename`*`"`
  to standard error, and exit with an exit status of 3. This must be the first invalid line found when processing the jobfiles in the order given on the command line. The italicised *`linenum`* is replaced by the actual line number (counting from one) from the jobfile that contained the syntax error, and the italicised *`filename`* is replaced by the actual filename i.e. the full argument given on the command line. (The double quotes must be present.) No jobs must be run if any invalid line is detected in any jobfile. An invalid line is one that is missing a mandatory field (e.g. any of the first three fields on a line are empty or missing) or one where the timeout field is not blank and contains a value other than a non-negative integer. Invalid pipe references and unopenable files do not make a line invalid – see below for the messages to be output in those cases.

[5] If you wish to refer to a jobfile named `-v` then you should pass it as `./-v`.

Checking for Job Runnability

If the jobfile(s) can be parsed, i.e. there are no invalid lines, then the proposed jobs must be checked for runnability.

- If `jobrunner` is unable to open a specified stdin file for reading then `jobrunner` should emit the following message to **standard error** and not attempt to run that job:
  `Unable to open "`*`filename`*`" for reading`
  including the quotation marks. The italicised *`filename`* must be replaced by the filename specified in the jobfile.

- If `jobrunner` is unable to open a specified stdout file for writing then `jobrunner` should emit the following message to **standard error** and not attempt to run that job:
  `Unable to open "`*`filename`*`" for writing`
  including the quotation marks. The italicised *`filename`* must be replaced by the filename specified in the jobfile.

- If there are multiple unopenable filenames then multiple messages must be output (one per line – in the order in which the filenames are found in the jobfile(s)[6]. Other jobs in the jobfile(s) that are valid should still be run (see next point also).[7]

- If the jobfile(s) contain an invalid pipe usage, i.e. either end of a `@pipe` does not have exactly 1 reference, then `jobrunner` should emit the following message to **standard error**, and ignore (i.e. not run) any jobs that reference the invalid pipe:
  `Invalid pipe usage "`*`pipename`*`"`
  where the italicised *`pipename`* text should be replaced with the name of the offending pipe (including the quotation marks but not including the `@` character pipe specifier). If there are multiple invalid pipes in the jobfile(s) then multiple messages must be output (one per line) – in the order in which the offending pipenames are first mentioned in the jobfile(s). There is only one message per invalid pipe no matter how many times that pipe is mentioned in the jobfile(s). Other jobs in the jobfile(s) that are valid should still be run. If there are no runnable jobs remaining then `jobrunner` should exit as described above (exit code 4). Note: pipe usage and other error checking (e.g. ability to open files) must be completed before creating any processes. Note that checking for cascading job invalidity is an advanced feature. For example, if a job that references an invalid `@pipe` or an unopenable filename also contains a reference to another valid pipe (e.g. `@pipe2`) then that other job becomes unrunnable because the process at the other end of the pipe is unrunnable (and this may cascade to cause another job to become unrunnable etc.) Pipes like this are not reported as invalid (they have two valid ends) – the relevant jobs just become unrunnable. There are only a very small number of marks associated with implementing this cascading pipe validity checking functionality.

- If there are both unopenable files and invalid pipes in the jobfile(s) then all "Unable to open ..." messages must be output prior to all "Invalid pipe usage ..." messages.

- Pipe references across multiple jobspec files are valid. That is, the first jobspec file could have a process writing to `@pipe1`, and the second jobspec file could have a process reading from `@pipe1`.

- If after parsing the provided jobfiles and checking for job runnability, there are no runnable jobs, then `jobrunner` should emit the following message to `standard error`:
  `jobrunner: no runnable jobs`
  and terminate with an exit status of 4. This message will be output after any applicable "Unable to open ..." or "Invalid pipe usage ..." messages. A "runnable" job in this case means a job where an attempt will be made to execute the job, e.g. it hasn't been deemed unrunnable due to a reference to an invalid pipe or an unopenable file. It does not mean the `exec()` will be successful.

Running the jobs

- All remaining (i.e. runnable) jobs must be executed after setting up their standard inputs and standard outputs as appropriate. If `jobrunner` attempts to `exec()` a program and the call fails, then that child process should exit with an exit status of 255, which should be reported by `jobrunner` (see below).

[6]If an unopenable file is mentioned more than once in the jobfile(s) then it will be listed more than once in the error messages
[7]Note that the behaviour of `jobrunner` can be undefined if a file specified as a stdin file is created as the output of another job – we will not test this.

- `jobrunner` must assign a unique incrementing job number starting from 1 to each job specified in the jobfiles. Unrunnable jobs, such as those with invalid pipe or file references, keep their number but are simply not launched.
- The ordering of jobs in the jobfile is irrelevant other than the job number that will be assigned. In particular, for jobs referring to the read and write ends of the same pipe, the relative order of the lines in the jobfile should not change the overall behaviour and interactions between the two processes.

Handling errors

- Note – if there is not a usage error (i.e. the program is not exiting with exit status 1) and more than one of the other errors applies (i.e. those for exit status 2, 3 or 4), then any one (and only one) of the applicable error messages can be output – subject to the contraints already stated above[8].
- Note also that all error messages must be terminated by a single newline character.

Program output

The operation of `jobrunner` can be considered in three phases:

- Initialisation – loading, parsing and checking the jobspec files
- Running – while the processes and jobs are running
- Finalisation – when all jobs have terminated.

The required output in each phase of operation is described below, for normal and verbose (`[-v]` command line option). All `jobrunner` output is to `stderr`.

Initialisation

Normal mode: No output other than error checking described above.

Verbose mode: If `-v` is specified on the command line and there is at least one runnable job then the complete job table should be output one line per job to `stderr` in the following format:

`jobNum:command:stdin:stdout:timeout[:arg1[:arg2...]]`

where `arg1, arg2` and their preceding colon are only emitted if present. If a job is unrunnable for any reason, e.g. due to incorrect pipe usage, it should not be listed.

Listing 9: Verbose job table output to stderr

```
1:ls:-:@pipe1:0:/etc
2:sort:@pipe1:ls.out:0
4:cat:-:cat.out:0:/etc/fstab
```

In this example, job number 3 was unrunnable for some reason and has not been output. Note that the timeout must be specified even if it was not present in the original jobfile. This job table output must be flushed to stderr before any jobs are run and after all runnabiliity error messages (if any) are output.

Running

Any jobs which specified '-' as their standard out, will share that file descriptor with `jobrunner` and as such their output will appear on `jobrunner`'s standard output.

Neither `jobrunner` nor any jobs it launches should emit anything to `stderr` during this phase.

[8]For example, the "no runnable jobs" message can not be output if a jobfile can not be opened or contains invalid lines; and invalid line messages must be about the first invalid line found. However, if the first jobfile on the command line contains an invalid line and the second jobfile can not be opened then you can exit with either exit status 2 or 3 and the appropriate message.

Finalisation

After all jobs have been launched, `jobrunner` shall monitor and report on child job's termination as follows.

Approximately once per second, `jobrunner` should check on its jobs (using `waitpid()`) – in job number order. Any jobs that have terminated will generate and flush a line of output to `stderr` in either one of the following formats, depending on the terminating condition:

Listing 10: `jobrunner` output for exiting jobs

```
Job N exited with status Y
```

where `N` is the job number (from 1 up to the total number of jobs), and `Y` is the exit code of the job process.

Alternatively, for jobs which terminate with a signal, the output should be as follows:

Listing 11: `jobrunner` output for signal-terminated jobs

```
Job N terminated with signal S
```

where again `N` is the job number, and `S` is the signal number (not name) that caused the job to terminate.

After emitting the status of any jobs that have terminated, `jobrunner` should `sleep()` for 1 second, and then repeat the `waitpid()` scan on any remaining jobs. When (or if) the last job finishes, then `jobrunner` should exit with exit status 0.

If you are implementing the timeout feature, then the `SIGABRT` signal generation to timed out jobs should also be done in this loop.

A full example of `stderr` output at the end of a job run is shown below, followed by explanatory commentary.

Given the following job specification file:

Listing 12: Sample `jobrunner` job specification

```
# Job 1 - this program blocks SIGABRT and sleeps. jobrunner should end up sending it SIGKILL
sigabort_blocker,-,-,~~5~~3
# Job 2 - there is no such program - exec() should fail and the child exit(255)
nosuchprogram,-,-
# Jobs 3 and 4 should run and complete very quickly
cat,/etc/mtab,out.txt
cat,/etc/fstab,fstab.txt
# Job 5 will sleep for 10 seconds
sleep,-,-,,10
# Job 6 is an example of an unrunnable job
cat,@badpipe,cat.out
# Job ~~6~~7 will sleep for 10 secs but has a 5 sec timeout - should terminate with SIGABRT
sleep,-,-,5,10
```

Listing 13: Sample `jobrunner` finalisation output

```
Job 2 exited with status 255
Job 3 exited with status 0
Job 4 exited with status 0
Job 1 terminated with signal 9
Job ~~6~~7 terminated with signal 6
Job 5 exited with status 0
```

In this example:

- job 6 will not be reported in the finalisation stage because the job was not runnable – an earlier message about invalid pipe usage would have been output
- job 2 terminated immediately with 255, because the `jobrunner exec()` call failed
- jobs 3 and 4 exited normally with zero error codes
- some time later (about 3 seconds after all the jobs were started), job 1 timed out and `jobrunner` sent it `SIGABRT`, however that was blocked by the program so a second later `jobrunner` then sent it `SIGKILL` (signal number 9), which caused the program to exit

- job ~~6~~7 received `SIGABRT` (signal number 6) when it timed out after 5 seconds, and
- finally job 5 exited last with an exit code of zero after the `sleep 10` completed

Advanced Functionality

If `jobrunner` receives `SIGHUP`, it should immediately send the `SIGKILL` signal to all processes still running. `jobrunner` should then still notice and report the termination of these job processes as per normal operation.

If you implement the timeout functionality, keep the following in mind:

- A timeout value of zero means that no timeout will be applied to the job
- The timeout values are expressed in seconds
- The timeouts only need to be approximate (i.e. within a second or so), e.g. for testing we may specify a job running "sleep 10" with a timeout of 5 seconds, to check that it is working properly.
- When a job times out, `jobrunner` must send it the SIGABRT signal.
- The termination of jobs that time out and are sent SIGABRT must be reported in the same way that standard job exit/signal termination is reported (see previous section).
- If a job keeps running (i.e. is still alive when `jobrunner` next checks) after being sent `SIGABRT` (i.e. because the process ignores or handles that signal), then `jobrunner` must then send it `SIGKILL`. (This happens about a second after sending the `SIGABRT` – when `jobrunner` next checks the status of the job.) The subsequent termination of the job must be reported in the usual manner which may be about a second after the `SIGKILL` is sent – when `jobrunner` next checks the status of the job.

Provided Library: `libcsse2310a3`

Two library functions have been provided to you to aid parsing of `jobrunner` jobfiles. These are

- `char* read_line(FILE* stream);`
- `char** split_by_commas(char* line);`

These functions are declared and described in `/local/courses/csse2310/include/csse2310a3.h` on moss. To use them, 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 these functions. To do this, use the compiler arguments `-L/local/courses/csse2310/lib -lcsse2310a3`

Style

Your program must follow version 2.1.1 of the CSSE2310/CSSE7231 C programming style guide available on the course BlackBoard site.

Hints

1. You can check the open file descriptors of any process by doing
   `$ ls -al /proc/<pid>/fd`
   where `<pid>` is the process ID of the process you want to examine. This will help you debug issues around ensuring that necessary file handles are closed in parent and child processes.
2. If using valgrind to debug memory issues, you can use the `--child-silent-after-fork=yes` option to prevent valgrind tracing child process, and thus only report on `jobrunner`'s memory usage. We will use this option for the valgrind marking criteria.
3. Use the provided library functions (see above).
4. All checking of jobs and creation of file descriptors for runnable jobs (e.g. setting up pipes and opening input and output files) should be done before any child processes are created. Each child can then set up its own stdin/stdout as required using `dup2()`.

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()`
- `mkfifo()` or `mkfifoat()`
- POSIX regex functions

Submission

Your submission must include all source and any other required files (in particular you must submit a `Makefile`). Do not submit compiled files (eg `.o`, compiled programs) or test (jobspec) files.

Your program (named `jobrunner`) must build on `moss.labs.eait.uq.edu.au` with:
`make`

Your program must be compiled with gcc with at least the following switches (plus applicable `-I` options etc. – see *Provided Library: `libcsse2310a3`* above):
`-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.

If any errors result from the `make` command (i.e. the `jobrunner` 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 (besides the one we have provided for you to use).

Your assignment submission must be committed to your subversion repository under

`https://source.eait.uq.edu.au/svn/csse2310-sem2-sXXXXXXX/trunk/a3`

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

You must ensure that all files needed to compile and use your assignment (including a Makefile) are committed and within the `trunk/a3` directory in your repository (and not within a subdirectory) 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 – the `reptesta3.sh` script will do this for you.

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)[9]. 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.

[9]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.

If you make a submission prior to the deadline[10] then your final submission prior to the deadline will be the submission that we mark – i.e. your last "on-time" submission will be marked. If your first submission is after the deadline (i.e. your submission is late) then that first submission will be the submission that we mark. A late penalty will apply in this case – 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 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. If your program takes longer than 20 seconds to run any test, then it will be terminated and you will earn no marks for the functionality associated with that test.

Exact text matching of files and output (standard out and standard error) is used for functionality marking. Strict adherance to the output format in this specification is 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	(3 marks)
2.	Program correctly handles comments and blank lines in job spec files	(3 marks)
3.	Program correctly handles invalid lines in job spec files	(5 marks)
4.	Program correctly handles multiple job spec files	(4 marks)
5.	Program currectly handles job standard in/out shared with `jobrunner`	(10 marks)
6.	Program correctly handles job standard in/out from regular files (including error handling)	(10 marks)
7.	Program correctly handles job standard in/out from pipes (including error handling)	(10 marks)
8.	Program correctly handles jobs with timeouts	(5 marks)
9.	Program correctly handles `SIGHUP` signal	(4 marks)
10.	Program correctly `free()`s all allocated memory (as reported by valgrind)	(4 marks)
11.	Program correctly cascades job invalidation for bad pipes/unopenable files	(2 marks)

Marks can only be awarded for criteria 10 if your `jobrunner` is capable of running child processes in at least some circumstances, i.e. your program earns some marks for criteria 5 to 7.

Style Marking

Style marking is based on the number of style guide violations, i.e. the number of violations of version 2.1.1 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.

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

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[11].

Automated Style Marking (5 marks)

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

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

- W be the total number of distinct compilation warnings recorded when your `.c` files are individually built (using the correct compiler arguments)
- A be the total number of style violations detected by `style.sh` when it is run over each of your `.c` and `.h` files individually[12].

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)

- V will be the number of **additional** style violations detected by human markers. Violations will not be counted twice (e.g. if a poorly named variable is used multiple times it will count as a single violation).

Your human style mark H will be

$$H = 5 - V$$

If $V \geq 5$ then H will be zero (0) – no negative marks will be awarded. If your code has many style violations, human style marking may be incomplete, e.g. the marker may stop after reaching 5 violations.

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:

- 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

Documentation (10 marks) – for CSSE7231 students only

CSSE7231 students must submit a PDF document containing a written overview of the architecture and design of your program. This must be submitted via the Turnitin submission link on Blackboard.

Please refer to the grading criteria available on BlackBoard under "Assessment" for a detailed breakdown of how these submissions will be marked. Note that your submission time for the whole assignment will be

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

[12] 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.

considered to be the later of your submission times for your zip file and your PDF design document. Any late penalty will be based on this submission time and apply to your whole assignment mark.
This document should describe, at a general level, the functional decomposition of the program, the key design decisions you made and why you made them. It must meet the following formatting requirements:

- Maximum two A4 pages in 12 point font
- Diagrams are permitted up to 25% of the page area. The diagram(s) must be discussed in the text, it is not ok to just include a figure without explanatory discussion.

Don't overthink this! The purpose is to demonstrate that you can communicate important design decisions, and write in a meaningful way about your code. To be clear, this document is not a restatement of the program specification – it is a discussion of your design and your code.

If your documentation obviously does not match your code, you will get zero for this component, and will be asked to explain why.

Total mark

Let

- F be the functionality mark for your assignment (out of 60).
- 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).
- D be the documentation mark for your assignment (out of 10 for CSSE7231 students) – or 0 for CSSE2310 students.

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@helpdesk.eait.uq.edu.au`.

Version 1.1

- Clarified what a valid timeout field is (non-negative integer with no leading or trailing spaces).
- Clarified that trailing and leading spaces should not be stripped from any fields.
- Clarified what a valid pipename is (essentially any characters you can put in a jobfile between the '`@`' and a comma.
- Added detail to the Program Operation section to make the sequence of operations clearer, including
  - clarified that usage errors must be checked before attempting to open any jobfiles;
  - clarified what is meant be an invalid job line;
  - specified the need to check whether stdin/stdout files can be opened;
  - clarified the ordering of messages when files can't be opened and when pipes are invalid (and both);

  - clarified that cascading job validity checking may be needed (as an advanced feature); and
  - clarified what is meant by a runnable job.

- Made it clear that `jobrunner` must flush stderr after the job table output (verbose mode) and when reporting each terminated process.
- Specified the exit status for when `jobrunner` exits after running jobs.
- Renamed the library functions to comply with the CSSE2310 style guide (deprecation warnings will be added for use of the old names before the functions are removed in the week of 20 September).
- Added another hint about the use of `dup2()`.
- Modified the marking criteria to allow marks for checking for cascading job invalidation.

Version 1.2

- Clarified minimum permissions for output files
- Made it clear that argument `-v` by itself is an invalid command line
- Clarified pipe error message ordering
- Clarified when the verbose job table is output
- Fixed mistake in listing 12/13 and added some explanatory text and an example of an unrunnable job
- Clarified timing of `SIGABRT`/`SIGKILL`