

The University of Queensland  
School of Information Technology and Electrical  
Engineering  
Semester One, 2011  
COMP2303 / COMP7306 - Assignment 3  
Due: 11pm 13th May, 2011  
Marks: 50  
Weighting: 25% of your overall assignment mark  
(COMP2303)

## Introduction

In this assignment, you will learn to execute, control and interact with other programs from within a C program. You will write a program called **octopus** which plays a number of simultaneous games of **flip**. This will require some modifications to the version of **flip** from Assignment one, apart from these though, the behaviour should be the same as specified for Assignment one.

Your assignment must comply with the C style guide available on the course website.

This is an *individual assignment*. You should feel free to discuss aspects of C programming and the assignment specification with fellow students. You should not actively help (or seek help from) other students with the actual

coding of your assignment solution. It is cheating to look at another student's code and it is cheating to allow your code to be seen or shared in printed or electronic form. You should note that all submitted code may be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct proceedings will be initiated against you. A likely penalty for a first offence would be a mark of 0 for the assignment. Don't risk it! If you're having trouble, seek help from a member of the teaching staff — don't be tempted to copy another student's code. You should read and understand the statements on student misconduct in the course profile and on the school website: <http://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism>

In this course we will use the **subversion** (svn) system to deal with assignment submissions. Do not commit any code to your repository unless it is your own work or it was given to you by teaching staff. *If you have questions about this, please ask.*

## **Octopus**

Octopus will take in actions from a file. These actions will be things like running a program or sending input to an already running program. Your main tests will have the

octopus playing a number of games of flip simultaneously but it should work for other programs as well. The programs run by the octopus should send their output to files on disk.

The **octopus** program will take a file which lists the actions it should perform. For example:

```
r:ls
```

Would run the **ls** command. A file containing:

```
r:./flip new 4 2 2
```

```
r:./flip new 6 1 2
```

```
coffeebreak:2
```

Would run two instances of **flip** and then pause for 2 seconds. Inputs (players) will be numbered starting from 0, so Player O and Player X of the first game will be 0 and 1 respectively. Player O of the second game will be 2 and so on. The outputs from players will be numbered using the same sequence. Output is sent to a file (by default **out.n** where **n** is the player number). Note that these files (inputs and outputs) should be opened even if the process won't use them.

To run **octopus** use:

```
./octopus script [prefix]
```

where *prefix* is an optional string used to name output files. So if the the prefix was "run" the output files would

be named `run.n`.

To send a line of text from a player use their number, a colon then the text. For example to send as Player 2:

```
2:your text here
```

A complete game might look like this:

```
r:./flip new 4
0:0 2
1:2 3
0:3 0
1:0 0
0:0 1
1:2 0
0:1 0
1:0 3
0:1 3
0:3 1
1:3 3
0:3 2
coffeebreak:2
```

When `octopus` completes its input file, it should terminate and reap any subprocesses which are still running. It should try `SIGINT` before moving on to `SIGKILL`.

## Octopus commands

The types of lines in the `input_file` are given in the following table.

<b>Command</b>	<b>Purpose</b>
	Blank lines - ignore
<i>#text</i>	Comment - this line should be ignored
<i>r:program args</i>	Execute <i>program</i>
<i>n:text</i>	Send <i>text</i> to player number <i>n</i>
<i>!:n</i>	End of input for player number <i>n</i>
<i>coffeebreak:n</i>	Wait for <i>n</i> seconds. Any child processes which have terminated should be reaped by the end of the pause.

Attempting to send to an input after using `!` on it should produce an error. All commands must appear at the beginning of the line (no leading spaces).

## Octopus errors

The following error messages should be output to **standard error**.

<b>Situation</b>	<b>Exit status</b>	<b>Message</b>
Wrong number of args	1	Usage: octopus input_file [prefix]
Can't read from input file	2	Unable to open input_file for reading.
Invalid line in input	3	Line %d is invalid.
Cannot open output files	4	Unable to create output file.
Can't send to an input	5	Unable to send to Player %d.
Can't create pipes / sub-processes / dup fails ....	6	System error.

## Octopus outputs

If a task fails to execute the send:

- Task %d failed to execute.

If a task exits cleanly with status 0, then **octopus** will not print anything. Otherwise, print one of the following messages (with placeholders filled in) to **standard out** as appropriate.

- Task %d exited due to a signal.
- Task %d exited with status %d.

Note that an abnormal/non-zero exit should not cause **octopus** to close.

#### File descriptors used by children of octopus

We want to be able to communicate with players independently so, whenever **octopus** executes a program it should set up file descriptors in the child process as follows:

<b>PID</b>	<b>Purpose</b>
0	Input for PlayerO
1	Output from PlayerO
2	Redirected to <code>/dev/null</code>
3	Input for PlayerX
4	Output from PlayerX

Remember, these files should be set up regardless of the command to be run. For example, `/bin/ls` will not use file descriptors 3 or 4 but you should set them up anyway.

## Changes to flip

Instead of reading moves for all human players from *stdin* and sending all output to *stdout*, each player should draw their input from the file descriptors given in the table above (remember you can use `fdopen()`). These changes rely on these file descriptors being open when `flip` is run. You will need to think about how to test your code under these conditions.

The output sent to each player will change slightly. Before a player is prompted for their move, the current board should be sent to them. Once the player has made a move (apart from saving) the board should be sent to them. If a player is forced to pass, then the board should not be sent but the “passes” message should be sent to both players. The game over and end of input messages should be sent to both players.

The load and computer players should work as before. The command line args, error messages and exit codes remain unchanged. As before, error messages should be sent to *stdout*.

If file descriptors 3, 4 cannot be used `flip` should print **File descriptor 3 or 4 could not be used.** exit with status 7.

## Cleanup

Individual users are only allowed to have limited numbers of processes running and files open simultaneously. For this reason it is important that subprocesses are reaped and files are closed when they are no longer needed.

When **octopus** exits (normally or in response to SIGINT or SIGTERM), any subprocesses should also be terminated.

## Compilation

Your code must compile with command:

**make**

The **Makefile** must pass the following flags to gcc:  
**-Wall -pedantic --std=gnu99**

You must not use flags or pragmas to try to disable or hide warnings.

If any errors result from the compile command (ie the executable cannot be created), then you will receive 0 marks for functionality. 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). If your code produces warnings (as opposed to errors), then you will lose style marks (see later).

Your solution must not invoke other programs (except where explicitly listed in the input file) or use non-standard headers/libraries. Your solution may consist of multiple `.c` and `.h` files, a **Makefile** but nothing else.

Your solution must not employ the `goto` instruction. To do so will result in a mark of **zero**.

### Style

You must follow *version 1.5* of the COMP2303/COMP7306 C programming style guide found at [http://www.itee.uq.edu.au/~comp2303/resources/c\\_resources.html](http://www.itee.uq.edu.au/~comp2303/resources/c_resources.html).

### Submission

Submission must be made electronically by committing using subversion. In order to mark your assignment the markers will check out `/ass3/trunk` from your repository on `svn.eait.uq.edu.au`. Code checked in to any other part of your repository will not be marked.

The due date for this assignment is 13th May at 11pm.

Test scripts will be provided to test the code on the trunk. Students are *strongly advised* to make use of this facility after committing.

## Marks

Marks will be awarded for both functionality and style.

### Functionality (42 marks)

Provided that your code compiles (see above), 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 open a file, we can not determine if it would correctly execute any commands. The markers will make no alterations to your code (other than to remove code without academic merit). Your programs should not crash or lock up/loop indefinitely.

- Command line errors
  - Usage instructions (2 marks)
  - Prefix support (2 marks)
- inputs and outputs connect correctly (2 marks)
- Coffeekbreak

- waits correctly (2 marks)
- reaps processes (2 marks)
- closes files (2 marks)
- Input handling
  - missing input file (2 marks)
  - invalid input lines (4 marks)
- Other octopus errors
  - only send to valid players (2 marks)
  - errors opening output files (2 marks)
- Octopus shutdown
  - terminate subprocesses (4 marks)
  - sending correct signals (2 marks)
- Single game (6 marks)
- Multiple tasks (8 marks)

**Style (8 marks)**

If  $g$  is the number of style guide violations and  $w$  is the number of compilation warnings, your style mark will be the minimum of your functionality mark and:

$$8 \times 0.9^{g+w}$$

The number of compilation warnings will be the total number of distinct warning lines reported during the compilation process described above. The number of style guide violations refers to the number of violations of version 1.5 of the COMP2303/COMP7306 C Programming Style Guide. A maximum of 5 violations will be penalised for each broad guideline area. The broad guideline areas are Naming, Comments, Braces, Whitespace, Indentation, Line Length and Overall. For naming violations, the penalty will be one violation per offending name (not per use of the name) up to the maximum of five. 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 mark 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.

### **Late Penalties**

Late penalties will apply as outlined in the course profile.

## Specification Updates

It is possible that this specification contains errors or inconsistencies or missing information. It is possible that clarifications will be issued via the course website. Any such clarifications posted 5 days (120 hours) or more before the due date will form part of the assignment specification. If you find any inconsistencies or omissions, please notify the teaching staff.

## Test Data

Test data and scripts for this assignment will be made available. The purpose of these are to help clarify some areas of the specification and to provide a basic sanity check of code which you have committed. *These are not guaranteed to check all possible problems nor are they guaranteed to resemble the tests which will be used to mark your assignments.* Testing that your assignment complies with this specification is still *your* responsibility.

## Notes

1. You must not use the following functions:
  - `system()`

- `popen()`

- `vfork()`

2. Some (but not all) of the tests for this assignment rely on a reasonably correct implementation of Assignment one. If you are not confident in your implementation please consider using the sample solution.
3. There are a number of ways to detect execution failure (ask in tutes). For this assignment you can assume that no program you will try to execute ever returns 99. You can use this value to indicate exec failure to the parent.