# CSCB09 2025 Summer Assignment 3

Due: July 27 2025, 11:59PM

In this assignment, you will practice making child processes, exec'ing other programs, and setting up I/O redirections and/or pipes.

As usual, you should aim for reasonably efficient algorithms and reasonably organized, comprehensible code.

## Process Substitution

bash has a convenient extra feature "process substitution" not in sh. In the simplest form it goes like

```
command1 <(command2)
```

command1 will see a command line argument. We users don't see it, but command1 does. Let's call it F because I need to refer to it later. command1 can treat F as a filename and open it for reading (only); command2 can write data to stdout. Here is the magical part: Whenever command1 reads from "file" F, it sees command2's output!

But it can be more general!

```
command1 <(command2a) <(command2b)
```

The above means: There are secret filenames F2a and F2b. When command1 reads from F2a, that's command2a's output; when command1 reads from F2b, that's command2b's output.

```
command1 <(command2 <(command3))
```

The above means: There are secret filenames F and G. When commmand1 reads from F, that's command2's output; when command2 reads from G, that's command3's output.

Many students used this feature to evade learning a required topic. The prof is not pleased. The prof's revenge is to make you implement this feature so you have a harder topic to learn!

(On the bright side, haven't you always wondered how bash pulls off this trick?)

### Discovery (optional, not graded)

Write a shell script or C program that prints the command line arguments it sees. It doesn't actually need to read any file. Use it as command1. This is to discover the secret F to help find out what's going on. Example bash command:

```
./yourprog <(echo hello)
```

What do you find? What do you think it means?

### Theory

/dev/fd is a special directory (not on real disk) made up by the kernel. Whenever a process P looks into it, P finds numeric filenames, and the numbers are exactly P's current open file descriptors. And if P reads from, say, /dev/fd/42, it's equivalent to reading from file descriptor 42.

(Naturally, different processes see different content, since the kernel is making it up on the fly, tailor-made to the process that asks.)

With this friendly service from the kernel, bash can pull off its trick. (It is still a pipe.) We assume that commands are executable programs with arguments in this assignment, even though bash is much more flexible than that.

For example, here is how to do `command1 <(command2)`:

- Create a pipe. Let's say the read-end FD is 42, write-end FD is 43, for the sake of concreteness, but this is just an example.
- Create a child process for command1 with argument /dev/fd/42 (the secret F above). So if command1 opens that "file" and read, it effectively reads from the pipe. Note that this is **not** stdin redirection.
- Create a child process for command2 with stdout redirected as FD 43. So if command2 writes to stdout, the data goes to the pipe.
- All 3 processes should also close certain unneeded file descriptors, as discussed in the lecture concerning pipe hygiene. It is your job to figure out which and when.

### Implementation

The function you will implement (create and hand in procsub.c) is:

```
int run(struct command *cmd, int *wstatus);
```

`struct command` specifies a command; see procsub.h, but informally it has: program name, number of arguments, and (pointer to) the arguments.

`struct argument` specifies an argument, see procsub.h; informally it has two cases: a string, or a command (meaning that process substitution is to be done). We will only implement the `<(cmd)` kind of substitutions.

Further clarifications can be found as examples in sample-main.c with comments stating the bash equivalents. In particular, if the bash equivalent is of the form `progname foo`, then the `numargs` field is 1, and the `args` array has just one element (for `foo`); so this is different from what `exec*()` expects, and you have to bridge the gap.

`run()` should run the given command in a child process, wait for it to terminate, and store its wait status at the address given by `wstatus` (you may assume that it is non-NULL). Needless to say, extra children are also necessary for arguments of the `SUBST` case; for simplicity, you do not need to call `wait()` or `waitpid()` on those other children.

`run()` should return 0 except under the conditions in the next section.

### Error handling

Any process: If `exec*()` fails, exit with exit code 127. Error message is up to you, but send any to stderr.

Other errors: You may ignore or handle, your choice. If you choose to handle: 1. send any error message to stderr; 2. if the error occurs in the parent, `run()` returns -1; 3. if the error occurs in a child, exit with exit code 127.