Guide to Running and Checking Fork Processes in C

1 Running C Code

To compile and run C code, use the following commands:

1. Compile: `gcc nameOfFile.c -o nameExecutable`
2. Execute: `./nameExecutable`

2 Checking Solutions

There are three methods to verify your fork process implementations.

2.1 Method 1: Using Python Script with Graphviz

2.1.1 Step 1: Setup Files

DISCLAIMER
The following code, even though it shows correctly the processes and its children, it can swap some branches between them For example, instead of: 1 – 2 — — 3 – 4 it may show: 1 – 3 – 4 — — 2 For the exact tree, use next method, even though it looks a bit weirder. BUT YOU CAN STILL USE IT, IT WORKS!

Create the following files in a folder:

template.c :

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>

void clear_log() {
    FILE *log_file = fopen("forks.log", "w");
    if (log_file != NULL) {
        fclose(log_file);
    }
}

void log_fork(pid_t parent, pid_t child) {
    FILE *log_file = fopen("forks.log", "a");
    if (log_file != NULL) {
        fprintf(log_file, "Parent %d Me %d\n", parent, child);
        fclose(log_file);
    }
}

```

```c
int main() {
    clear_log();

    // insert fork code here

    log_fork(getppid(), getpid());
    sleep(10);
    wait(NULL);

    return 0;
}
```

script.py :

```python
import os
import time
import graphviz

log_file_path = 'forks.log'


def update_graph():
    global dot
    dot = graphviz.Digraph(comment='Forks')

    with open(log_file_path, 'r') as file:
        lines = file.readlines()

    for line in lines:
        if line.strip():
            parts = line.split()
            parent = parts[1]
            child = parts[3]
            dot.node(parent, f'Parent {parent}')
            dot.node(child, f'Me {child}')
            dot.edge(parent, child)

    output_filename = 'forks_graph'
    dot.render(output_filename, format='png')
    print(f"Graph updated and saved as {output_filename}.png")


def monitor_log():
    if not os.path.exists(log_file_path):
        print(f"No log file found at {log_file_path}")
        return
    last_size = os.path.getsize(log_file_path)
    while True:
        update_graph()
        time.sleep(3)
if __name__ == "__main__":
    monitor_log()
```

Also create an empty file called `forks.log`.

2.1.2 Step 2: Run script.py

Ensure Python is installed (if not, install it with pip):

```
python3 script.py
```

2.1.3 Step 3: Run template.c in a SEPARATE Terminal

```
gcc template.c -o template
./template
```

2.1.4 Step 4: View the Output

Wait a few seconds and open `forks_graph.png`. It should look like this:

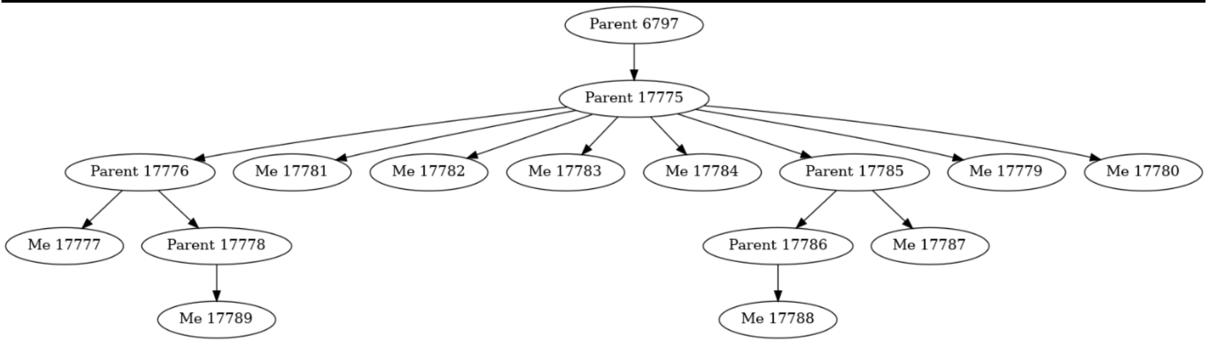


Figure 1: Example fork process graph

Note: The actual process tree starts from the second process. In this example, the graph starts from parent 17775. We ignore the first parent (6797) as that is the process which called our template.c.

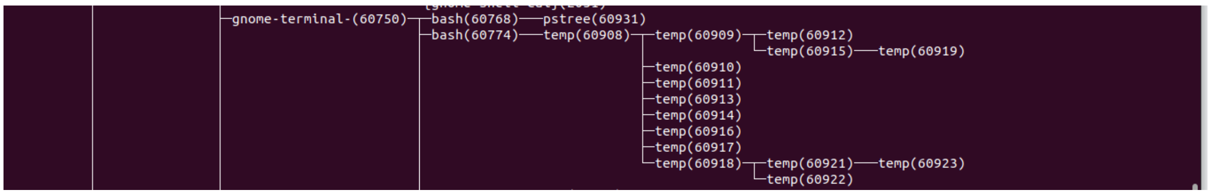
2.2 Method 2: Using pstree

1. Create the same `template.c` file as shown above
2. Run the code from template.c
3. In a **SEPARATE terminal**, run the command:

   ```
   pstree -p

   ```

4. It should print something like this:



```
├─gnome-terminal-(60750)─┬─bash(60768)───pstree(60931)
│                        └─bash(60774)───temp(60908)─┬─temp(60909)─┬─temp(60912)
│                                                    │             └─temp(60915)───temp(60919)
│                                                    ├─temp(60910)
│                                                    ├─temp(60911)
│                                                    ├─temp(60913)
│                                                    ├─temp(60914)
│                                                    ├─temp(60916)
│                                                    ├─temp(60917)
│                                                    └─temp(60918)─┬─temp(60921)───temp(60923)
│                                                                  └─temp(60922)
```

Figure 2: Example pstree output

We have to scroll until we see the branch `gnome-terminal`. Our tree will be shown under one of the bash processes, with the name of our executable file (in this case, the commands `gcc template.c -o temp` and `./temp` were run).

NOTE: We have to run the `pstree -p` command right after running the template executable so that no process remains orphan. The line `sleep(10);` makes sure that the tree isn't destroyed that fast.

3 Important Warning: Shall We Use AI?

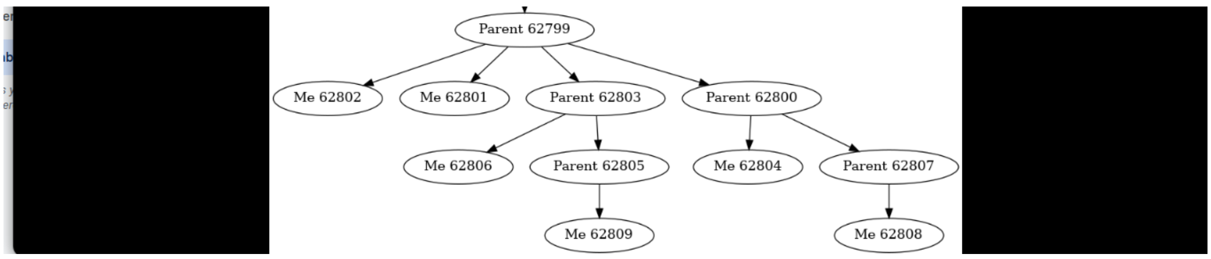
NO!!!! NEVER!!!!!
Current AI models don't know how concurrency and forks work, so they can really mess up trying to make a process tree.

3.1 Example of AI Failure

For the following code, I asked claude.ai to make the process tree:

```
if (!fork()){
    if (fork())
        if (!fork())
            fork();
} else
    if (fork())
        if (fork())
            if (!fork())
                if (!fork())
                    fork();
                else
                    fork();
```

3.1.1 Actual Tree:



[redacted]
[redacted]

Figure 3: The actual process tree

3.1.2 Claude's Tree:

?????????

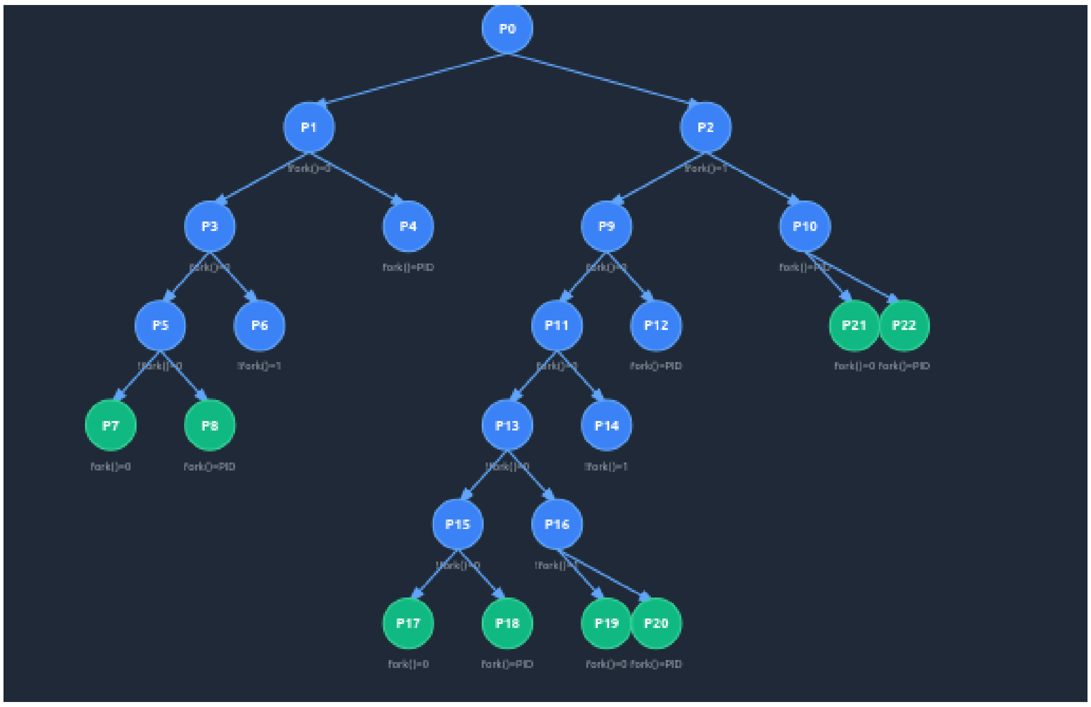


Figure 4: Claude's incorrect process tree

4 Creating Fork Code from Process Trees

If you want to create the fork code from a process tree, you can draw a random tree yourself or use a web tool that generates random trees, then try making the code yourself! Use the verification methods described above to check your solution.