Fork

When running a C program, you may wish to spin off another process to execute in the background, to do this, you'll want to use the `fork` command. When `fork` is called, the program calling it is referred to as the `parent`, and the new process that starts running is referred to as the `child`. Once the `fork` process is called, both parent and child simultaneously from the exact same point in the code with the exact same values up to that point.

The `fork` command returns the child process's PID so that the parent process can signal to the child process. In the child process, `fork` returns a value, but it returns 0. It is possible for `fork` to fail, and if it does so, it returns a -1.

Let's create a file that will fork off a child, and then have both parent and child write to a file:

`forking.c`

```
1  #include<unistd.h>
2  #include<sys/types.h>
3  #include<stdio.h>
4  #include<stdlib.h>
5
6  int main(int argc, char** argv) {
7
8    pid_t childPid = fork();  // This is where the child process splits from
   the parent
9
10   if (childPid == 0) {
11     printf("I am a child! My parent's PID is %d, and my PID is %d\n",
   getppid(), getpid());
12   } else {
13     printf("I'm a parent! My pid is %d, and my child's pid is %d \n",
   getpid(), childPid);
14   }
15
16   return EXIT_SUCCESS;
17 }
```

In the above code, we have a parent forking off a child at line 8. From there, a new child process is created! We then access the child process's PID through using what was returned from `fork`. We can also see that the processes can access their own PIDs by using the `getpid` command, and they can even look upwards to see their parents' PIDs by using `getppid`.

Suppose

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

void printArray(int * arr, int size, FILE * outputFilePointer, char* executionContext) {
  fprintf(outputFilePointer, "%s's array says: ", executionContext);

  int i;
  for (i = 0; i < size; i++) {
    fprintf(outputFilePointer, "%d ", arr[i]);
  }
  fprintf(outputFilePointer, "\n");
}

void mutateArray(int *arr, int size, int multiplier) {
  int i;
  for (i = 0; i < size; i++) {
    arr[i] = arr[i] * multiplier;
  }
}

int main(int argc, char** argv) {
  char* filename = "logfile.txt";
  char* executionContext = "parent";
  int SIZE = 10;
  int someArray[SIZE] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

  pid_t childPid = fork();  // This is where the child process splits from the parent

  if (childPid == 0) {
    printf("I am a child! My parent's PID is %d, and my PID is %d\n", getppid(), getpid());

    executionContext = "child";

    mutateArray(someArray, SIZE, 3);
  } else {
    printf("I'm a parent! My pid is %d, and my child's pid is %d \n", getpid(), childPid);

    sleep(1);
```

```
    mutateArray(someArray, SIZE, 1);
  }


  FILE * outputFilePointer;

  outputFilePointer = fopen(filename, "a");

  if (outputFilePointer == NULL) {
    printf("Bad file! %s cannot be opened", filename);

    return EXIT_FAILURE;
  }

  printArray(someArray, SIZE, outputFilePointer, executionContext);

  fclose(outputFilePointer);

  return EXIT_SUCCESS;
}
```

What's happening in the above code!? We `fork`, and then immediately start accessing an array to mutate it. Both parent and child mutate their arrays in different ways. We then choose to write to a file! What's curious is that, depending on the speed of the system, it's possible for this output to be intertwined (however, for this specific example it's relatively unlikely since these processes are so short). Entirely new address spaces are used, so we don't ever have to worry about overwriting!