# Report for System Programming Hw3

I stated my method to fulfill this homework into three parts, one is the main body of my code, another one for `fork()` and the last one is for `vfork()`.

In my code, I first check the arguments inside the `argv[]`, and then initialize the form of the data I stored, which is a square table with side length of 1, 4, 9, 16, or 25. Besides filling the form, I will record some more information of every row, column and block with status compression. After that, I will begin `DFS()` with the position that I am caring about in the form and the number of process that I still can use. I will enumerate all the valid possibilities at the certain position, then determine whether I can use a child process to `DFS()` or just `DFS()` by the current process itself. And no matter which way it `DFS()`, the current process will `wait()` until all the child processes return back. Furthermore, if one process get a solution, it will create a file and then print the answer. O_EXCL will guarantee an unique existence of the answer.

Now, I stated the method I use `fork()`. I will first determine the number of child process that I can use in this child process. After that, I will `fork()` first , and then the child process will change the form into the one of the valid possibilities and `DFS()` . In the meantime, this current process will keep enumerating the next possibility.

Finally, I stated the method I use `vfork()`. I will first determine the number of child process that I can use in this child process., too I will change the form into one of the valid possibilities and then `vfork()`. The child process then will copy the current data and some compulsory information into a temp array and then `execv()` with `argv[]`, which is copied from the temp array. In the meantime, the current process will have to first restore the status because of the share of the data with child process before `execv()`. After that, the current process will keep enumerating the next possibilities.

Testing results:

| (arguments) | ( PC: real T / user T / system T) | (On 217 real T / user T / system T) |
|---|---|---|
| `-m fork -p 1` | 14.572/ 4.308/ 24.814 | 31.72/ 5.68/ 57.49 |
| `-m vfork -p 1` | 14.508/ 4.260/ 24.730 | 33.20/ 6.22/ 59.89 |
| `-m fork -p 16` | 19.577/ 9.221/ 56.544 | 47.59/ 9.71/ 157.59 |
| `-m fork -p 32` | 17.996/ 8.557/ 53.195 | 47.77/ 10.17/ 163.68 |
| `-m fork -p 128` | 19.601/ 9.181/ 57.252 | 49.04/ 9.91/ 160.96 |

Analysis:

First, comparing the first and second data, we can find out that the `vfork()` runs faster than `fork()` when executing on the 217. Because that `vfork()` do the same things as `fork()`, but it does a lot of work on copy the arguments into `execv()`. And, I believe that the results on my computer are just because that the data is too small and some deviation occurred.

Second, comparing the first and the last data, we can find out that the time doesn't increase stably. The reason is that the numbers of available processes in the recursion function may lead to different cost of time. However, when the number of process was compared between the first and last data, it is quite obvious that the system is quite a lot because of the check of available processes.

Another interesting point is that the ratio of system time to user time is quite large. It is the result of my checking system. When I get an answer, I will create a file in order to preserve the unique answer. However, it made a lot of system cost on using `stat()`.