Performance experiment of Simpsh, Bash, Execline

Report of CS111 Project 1C

By Chunnan Yao, Jiaqi Huo

1. Benchmark

We use the following command line to generate arbitrarily size of data.

```
base64 /dev/urandom | head -c [# BYTES OF DATA] > [FILE NAME]
```

In this report, we use two datasets, one 10MB and the other 10KB, named “dataset1” and “dataset2”.

2. Test cases

We show our 4 test cases here. For each case, we have versions of Simpsh, Bash, Execline listed respectively.

1) case 1

case 1 tests performance of basic command execution, file reading, file outputting, command execution, just using a “sort” operation.

Simpsh:

```
./simpsh \
--rdonly dataset1 \
--creat --trunc --wronly b \
--creat --append --wronly c \
--command 0 1 2 sort
```

Bash:

```
(sort < dataset1 > b) 2>>c
```

Execline:

```
redirfd -r 0 dataset1
redirfd -w 1 b
redirfd -a 2 c
sort
```

2) case 2

case 2 tests performance more comprehensively, adding pipe and multi-command.

Simpsh:

```
./simpsh \
--rdonly dataset1 \
--pipe \
--pipe \
--creat --trunc --wronly b \
--creat --append --wronly c \
--command 0 2 6 sort \
--command 1 4 6 cat dataset2 - \
--command 3 5 6 tr A-Z a-z
```

Bash:

```
(sort < dataset1 | cat dataset2 - | tr A-Z a-z > b) 2>>c
```

Execline:

```
redirfd -r 0 dataset1
redirfd -w 1 b
redirfd -a 2 c
pipeline {
        sort
} pipeline {
        cat dataset2 -
}
tr A-Z a-z
```

3) case 3

case 3 inspects the influence of –wait on command execution.

Simpsh:

```
./simpsh \
--rdonly dataset1 \
--pipe \
--pipe \
--creat --trunc --wronly b \
--creat --append --wronly c \
--command 0 2 6 sort \
--command 1 4 6 cat dataset2 - \
```

```
--command 3 5 6 tr A-Z a-z \
--wait
```

Bash:

```
(sort < dataset1 | cat dataset2 - | tr A-Z a-z > b) 2>>c
wait
wait
```

Execline:

```
redirfd -r 0 dataset1
redirfd -w 1 b
redirfd -a 2 c
foreground {
pipeline {
        sort
} pipeline {
        cat dataset2 -
}
tr A-Z a-z
}
wait
```

4) case 4

case 4 inspects performance of signal mechanism.

Simpsh:

```
./simpsh \
--catch 11 \
--abort
```

Bash:

```
trap 'echo catch SIGSEGV' SIGSEGV
kill -s SIGSEGV $$
```

Execline:
My teammate and I spent more than 2 hours trying to learn how to use “trap” “getpid” in execline but failed. Execline is too finicky and there is no online resources and even examples in its document. We tried our best and decided to give up. The time spent is meaningless and we will never do that again.

3. Experiments

We use linux “time” command to measure execution time for each shell and simpsh’s –profile option to get the time of its subcommands. However, we should make sure the source of the statistics that “time” gets. So we need do some pre-experimenets.

We run each experiment 5 times and calculate average value as result.

Pay attention, for simpsh, we are able to get better accuracy in time using getrusage() from GNU C library. But we can’t get same accuracy using linux command “ time”.

Please refer to http://unix.stackexchange.com/questions/70653/increase-e-precision-with-usr-bin-time-shell-command) for explanation
.

3.1Pre-experiments

3.1.1 Time of Child Process in Pipeline For Execline

we execute:

```
#! /bin/execlineb
pipeline {
   sort dataset1
}
sort dataset2
```

and:

```
#! /bin/execlineb
pipeline {
   sort dataset2
}
sort dataset1
```

The first script runs very fast, 0.000s user time and 0.003s system time. The second script runs slow, 0.397s user time and 0.007s system time. However, they both execute sort on dataset1 and dataset2, the execution time should be similar. Thus we infer that “time” on execline script will only count its parent process’ user time and system time. Note that for pipeline usage, the reading program resides in parent process. So we can utilize this feature to measure sub-commands’ running time without worrying about the time spent in reading files.

We decided to just run the whole script once to get parent process’ time plus one subcommand’s time, and run each sub-command separately. And after that we should

do some data post-processing to get parent process's time and child process's time separately.

3.1.2 time of child process in bash

we execute:

sort dataset1 | sort dataset2

sort dataset2 | sort dataset1

sort dataset1

sort dataset2

The first script runs 0.375s user time and 0.007s system time. The second runs 0.416s user time and 0.010 system time. The second script always runs a little bit slower. The third script runs 0.390s user time and 0.010s system time. The forth script runs 0.000s user time and 0.001s system time.

We infer that when "time" bash scripts, it will count all the child process' time. So we don't have the same trouble as in execline.

3.2Performance test

1) Test case 1

Chart 3.2.1 case 1 test result

command(in bash) CPU time (s) / shell	(sort < dataset1 > b) 2>>c (user time/kernel time)
Simpsh	0.388/0.017
Bash	0.389/0.015
Execline	0.420/0.015

2) Test case 2

We measured subprocesses' CPU time as well as parent process' CPU time.

To get sub processes' running time of exeline, as we've pointed out in section 3.1.1, we run:

```
redirfd -r 0 data0
pipeline {
    sort
}
tr a-z A-Z
```

to get execution time of sub-process “tr a-z A-Z”.

We get the parent process execution time of simpsh by adding corresponding data reported by –profile.

We get child process execution time of bash by using GNU “time” to each sub commands in pipe. Our method is shown as follows.

```
alias time=/usr/bin/time
(/usr/bin/time -f "sort:\nreal %E\nuser %S\n" sort < dataset1 | /usr/bin/time -f "cat:\nreal %U\nuser %S\n" cat dataset2 - |/usr/bin/time -f "tr:\nreal %U\nuser %S\n" tr A-Z a-z > b) 2>>c
```

Chart 3.2.2 case 2 test result

CPU Time(s) / Shell	sort < dataset1	cat dataset2 dataset1	tr A-Z a-z <dataset0> b (dataset0 is concatenation of dataset1 and dataset2)	(sort < dataset1 | cat dataset2 - | tr A-Z a-z > b) 2>>c
Simpsh	0.402743/ 0.009994	0.001144/ 0.012590	0.009629/0.018294	0.001028/0.002014
Bash	0.39/0.00	0.00/0.01	0.01/0.01	0.009/0.044
Execline	0.398/0.012	0.001/0.003	0.013/0.009	0.003/0.011

3) Test case 3

Chart 3.2.3 case 3 test result

CPU time (s) / Shell	sort < dataset1	cat dataset2 dataset1	tr A-Z a-z wait; wait; wait <dataset0> b (dataset0 is concatenation	(sort < dataset1 | cat dataset2 - | tr A-Z a-z > b)

			of dataset1 and dataset2)		2>>c; wait; wait; wait
Simpsh	Same as case 2	Same as case 2	Same as case 2	0.000/0.000	Almost same as case 2
Bash	Same as case 2	Same as case 2	Same as case 2	0.000000/0.000158	Almost same as case 2
Execline	Same as case 2	Same as case 2	Same as case 2	0.000/0.000	Almost same as case 2

4) Test case 4

Case 3.1.4 case 4 test result

CPU time / Shell	trap 'echo catch SIGSEGV' SIGSEGV kill -s SIGSEGV $$
Simpsh	0.000/0.001
Bash	0.002/0.001
Execline	We don't know how to measure. The "trap" option is too hard to use.

4. Result Analysis

From the results above, we conclude that performance of child processes of all three shells are similar. For parent processes, simpsh is better in both user time and kernel time. That's because simpsh only need to deal with simple needs, so it is highly customized, while bash and execline have more general implementation, thus they introduced more complexity.

For signal handling, simpsh has better user time. It's because Bash "trap" command have to call "echo" to deal with signals, while Simpsh implement signal handler using printf() in C by itself.

To conclude, simpsh parent process performs better due to its customization using GNU C library.