9Back : Making Our Plans Great Again

Maxwell Bland Leon Cheung Kilian Pfeiffer

University of California, San Diego

Abstract

The following paper describes a set of benchmarks run on the Plan 9 operating system. In particular, it uses the only actively developed branch, 9FRONT ”RUN FROM ZONE!” (2018.09.09.0 Release). The goal of this project is to provide researchers and enthusiasts with greater insight into the current state of the operating system's performance and capabilities of interaction with hardware. Through tests of CPU, Memory, Network, and File System operations, we will gain insight on bottlenecks in the system's performance and the interactions between low-level (hardware) and high-level (OS) system components. These performance statistics will be contrasted with subjective experiences of "responsiveness".

1 Introduction

Plan 9 from Bell labs is a distributed operating system which emerged around the 1980's. It built upon the ideas of UNIX, but adopted an ideology that "everything is a file". Although the system was marketed in the 90's, it did not catch on, as prior operating systems had already gained enough of a foothold. Eventually, during the 2000's, Bell Labs ceased development on Plan 9, meaning official development halted. Unofficial[1] development continues on the 9front fork of the codebase, with new support for Wi-Fi, Audio, and everything anyone could ever want or need.

The experiments were performed as a group using a shared codebase and a single machine described in the following section. The measurements were performed via programs written in Plan 9's *special* version of C 99 and Alef, both described in the original Plan 9 paper todocite. The compilers used were the x86 versions of Plan 9's C compiler and Alef compiler, 8c and 8al respectively. The compilers were run with no special optimization settings. Version numbers are not available. Measurements were performed on a single machine running Plan 9 directly from hardware; given the nature of the Plan 9 system, additional metrics could be established for networked file systems and CPU servers; these measurements were not done for sake of simplicity, and because results under these conditions should be inferrable from the results cataloged within this paper.

[1]This is debatable. If you adopt an orphan, are they not your official child?

2 Machine Description

We ran this beautiful operating system of the gods on a Thinkpad T420, the machine of true developers.

```
Processor: model, cycle time, cache sizes (L1, L2,
  instruction, data, etc.)
  Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz
  cache size 3072 KB
  cpu family 6
  model 42
  stepping 7
  siblings 4
  cores 2
  fpu yes
  fpu_exception yes
  fpu_exception yes
  bogomips 4986.98
  clflush size 64
  cache_alignment 64

Memory bus
  DDR3-1333
  i/o-bus-frequency: 666MHz
  bus-bandwith:  10656 MB/s
  memory-clock:  166MHz
  Column Access Strobe (CAS) latency:

I/O bus
  SataIII-speed: 600MB/s

RAM size
  8 GB
```

```
Disk: capacity, RPM, controller cache size
  Samsung SSD 860 EVO 500GB
  Capacity: 500GiB
  RPM: 550MB/s read, 520 MB/s write
   and 98,000 IOPS (Read QD32)
  Controller Cache Size: 512MB
Network card speed:
 Intel 82579 LM Gigabyte: 1Gb/s
 intel Centrino Ultimate-N 6300: 450 Mbps

Operating system (including version/release)
  9FRONT "RUN FROM ZONE!" (2018.09.09.0 Release)
```

3 Experiments

For each section, we report the base hardware performance, make a guess as to how much overhead software will add to base hardware performance, combine these two into an overall prediction of performance, and then implment and perform the measurement. In all cases, we run the experiment multiple times, and calculate the standard deviation across measurements. We use the `nsec()` syscall to record the timestamp. Dynamic CPU frequency scaling was disabled for all trials, and all trials were restricted to a single core.

3.1 Measurement Overhead

The following section reports overheads of reading time, and the overhead of using a loop to measure meany iterations of an operation. One trial involves looping over 16 `nsec` timing calls, 2^{16} times. We average over a single trial, and we do 64 trials.

In order to measure the loop overhead, we took the time at the end of a loop iteration and at the start of a loop iteration, and find the difference between those to measure the loop overhead time. We repreat this 16384 times within each trial, and perform 64 trials. We average over all of these trials.

In measuring the function overhead, we created empty functions of zero, two, and eight parameters. We then proceeded to call eacah of these functions 20000 times, and found the average time taken over these 20000 calls.

To measure the syscall overhead, we used the `errstr(2)` syscall, in particular, `rerrstr(char*, vlong)`. `reerrstr` reads the error string for the error of a previous syscall, and does not clear the errstr buffer. Within each trial, we performed the syscall 16384 times, and averaged over 64 trials.

We measured process creation overhead in two different ways, one in which we `rfork`'ed a new process without copying the parent state to the child, and, `fork`, where the child will copy all of the parent's file descriptors, and share the other state as normal. We do these 20000 times for each type of `fork` and we average over all of these calls.

	mean	standard dev.	min
measurement overhead	1ns	1ns	-
loop overhead	1ns	1ns	-
procedure call overhead	1ns	1ns	-
syscall overhead			-
fork time			-
context switch time			1ns

Figure 1: Measurement Table

Finally, to measure context switching, we created a pipe in a parent process, and forked off a new child process. We forced a context switch by writing to the pipe in the parent and then waiting for the child to read from the pipe and terminate. We take the time between when the parent goes to sleep and when the parent wakes up again after the child's termination. Then, we subtract the overhead of reading from the pipe and closing the pipe (which happens in the child), and divide the time in half to account for the two context switches. We repeat this measurement 1000 times and use a new pipe each time. To take the time of a context switch, we take the minimum time over all of these trials.

For the function overhead, we found that the number of function parameters does not significantly change the overhead of calling the function.

Compare the measured performance with the predicted performance. If they are wildly different, speculate on reasons why. What may be contributing to the overhead? Evaluate the success of your methodology. How accurate do you think your results are? For graphs, explain any interesting features of the curves. Answer any questions specifically mentioned with the operation.

References

Notes

[1]Remember to use endnotes, not footnotes!