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. The compiler used were the x86 versions of Plan 9's C compiler, 8c. The compiler was run with optimizations disabled. 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 plus network overheads.

The use of actual hardware for this task should have less variance compared to measurements performed within a VM, although there were several points in time where it became clear that Plan 9 was not neccessarily able to handle the hardware interface as expected, such as during the high contention disk access task, which caused I/O errors to shoot off from the SSD.

The project was accomplished over the course of ten weeks in short five to twenty hour bursts, wherein team members took turns on implementation tasks according to their respective amounts of free time. In total, due to difficulties inherent with the particular operating system chosen for benchmarking, each team member spent around 30 to 40 hours on this project.

Benchmarking a forty year old bunny-focused OS maintained by less than 10 strangers including us and making it jump through rings of fire was admittedly a weird flex but okay.

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 (Sandy Bridge)
      cache size 3072 KB
      L1$ 128 KiB
        L1I$ 64 KiB 2x32 KiB 8-way set associative
        L1D$ 64 KiB 2x32 KiB 8-way set associative
write-back
      L2$ 512 KiB 2x256 KiB 8-way set associative
write-back
```

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

```
      L3$ 3 MiB   2x1.5 MiB 12-way set associative
write-back
      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
    Details at https://en.wikichip.org/wiki/
    intel/core_i5/i5-520m

    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 `cycles()` 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 (Reading Time)

The following section reports overheads of reading time. One trial involves looping over 16 `cycles` timing calls, 2^{16} times. We average out for each call of `cycles`, and we do 64 trials altogether.

Since we imagine getting the current cycles clock is a fast operation, so we estimate the hardware performance to be on the scale of nanoseconds, since one clock cycle takes approximately 0.5 nanoseconds. We think that the `cycles` call should do something similar to reading from the cycle counter, so it should be almost instantaneos, perhaps on the order of 2ns. [Page 547 of https://www.intel.com/content/dam/www/public/us/en/documents/manual ia-32-architectures-software-developer-vol-2b-manual.pdf]

The software cost of this should be low to non-existent, since that value is directly loaded into two registers, which might be directly examined under normal circumstances. It is the case that Plan 9's C compiler does not accept assembly directives, and thus we were forced to use the `cycles` call, which may be doing extra work.

This would put our predicted cost at around 2 to 10 nanoseconds, accounting for potential variations due to operating system tasks such as context switching, interrupts, and other architectural lags.

Our results are shown in table 1; it seems we may have underestimateed the cost of hardware and software in measurement of time. We feel that our methodology, however, is sound, given the incredibly low degree of variability in our results. Units are reported in nanoseconds.

3.2 Measurement Overhead (Loop Overhead)

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.

The cost of doing a single for loop based branch in hardware is also very low. Assuming a compare, jump, and register increment take 3 to 5 cycles, and adding additional overheads for branch misprediction, the loop overhead should be around 10 cycles. [Patterson, David A.; Hennessy, John L. Computer Organization and Design: The Hardware/Software Interface.]

Software, will slow this down due to random interruptions while executing, as mentioned before. Other system processes may need the processor, and thus the latency might be roughly twice this.

As a prediction, the cost of a loop will be on the order of 20 nanoseconds, maybe less, since our hardware estimate may be a bit too lenient.

The results for this experiment are also in table 1; we did underestimate the cost of a loop by a bit; we feel this is most likely due to higher hardware costs than expected, perhaps due to the ineffiencies of the processor's branch predictor or just high cycle count instructions. We have every reason to believe our methodology was sound.

	Reading (ns)	Loop (ns)
Hardware Guess	1	5
Software Multiplier	2-10	2
Prediction	2-10	10
Average	11	17.2
Std Dev.	0	0

Figure 1: Measurement Overheads

3.3 Procedure Call Overhead

In measuring the function overhead, we created functions of zero to seven integer arguments, we then proceeded to call eacah of these functions 20000 times, and found the average time taken over these 20000 calls. Overall, increment overhead was almost non-existent.

The hardware cost of a procedure call using the standard c calling convention should not be that high, as it is almost all done in registers. The caller pushes the arguments to the stack (very little cost), saves the return address, the frame pointer, and calls the function. There is also all the take-down cost for the function call, and the jumping of the instruction pointer, which will clear the incoming buffer of instructions. The hardware cost of this should be significant but not insane; we are thinking on the order of 30 cycles, so around 15 or less nsec. [https://9p.io/sys/doc/compiler.html]

The software cost of a function call in plan 9 is remarkably small, as the compiler stores all argument positions relative to a stack pointer, and thus needs to consider only one piece of metadata while making a call. There is still the threat of context switches, so the software cost may be a multiplier between 1 and 2.

This puts our prediction for the cost to be somewhere around 15 to 20 nanoseconds for a function call, independent of the number of parameters passed.

Our results are recorded in table 2; we were roughly in line with our estimates, though there is a significant degree of variance in the results, most likely due to needing to make a reference back into DRAM or context switches. This was a straightforward quantity to measure and most likely our results are accurate.

As a final note, there is little overhead for adding arguments to a function call in plan nine, most likely due to the stack pointer relative addressing and our lack of tests for access times to the arguments vs other operating systems.

3.4 System Call Overhead

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.

Hardware Guess	15 nsec	
Software Multiplier Guess	1-2	
Prediction	17 nsec	
Num Args	Average (nsec)	Std Dev. (nsec)
0	14.487840	8.702724 1
14.892560	8.991838 2	15.215200
9.187626 3	14.844640	8.996905 4
14.883760	8.986072 5	15.223920
9.204537 6	14.851200	8.963965 7
15.226480	9.198042 8	15.264640
9.227006		

Figure 2: Procedure Call Overheads

	System Call Overhead
Hardware Guess	100 nsec
Software Multiplier	10x
Prediction	1000 nsec
Average	861 nsec
Std Dev.	70.8 nsec

Figure 3: Syscall Overhead

Data from IBM puts the cost of making a system call on the Pentium processor in 1997/1995 to be 223 cycles, or roughly 1.68 usec for that processor. Our processor is much faster, putting the timing at around 100 nsec.

It is likely that the overhead of a syscall beyond this is pretty high due to the actual operations and implementation of the syscall operation. Thus, the cost could be 10x the hardware cost.

That puts our estimate at around 1000 nsec or 1usec, which still seems reasonable for a system call.

Results are shown in table three and reveal a slight overestimate on our part, though perhaps not an unreasonable one. Notably, this cost is much, much higher than a empty-body procedure call. We do not have any reason to believe that Plan 9 would be caching the results of this system call so that subsequent calls do not need to trap to the OS. This may be a potential source of optimizations.

Our methodology for evaluating this experiment was straight forward enough and is likely correct.

3.5 Task Creation Time

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. This methodology results from the fact that Plan 9 does not have kernel threads.

The hardware costs of a light fork should be much

	Light Fork	Heavy Fork
Hardware Guess	30 usec	30 usec
Software Multiplier	1x	1x
Prediction	30 usec	30 usec
Average	27.250 usec	28.674 usec
Std Dev	1.83 usec	58.068 usec

Figure 4: Fork Overheads

Hardware Cost	1600 nsec
Software Multiplier	10x
Prediction	16 usec
Average	13.458 usec
Std Dev.	9.569 usec
Min	7.862 usec

Figure 5: Context Switch Overheads

less than a heavy fork, since resources do not need to be shared, but because our processes are minimal, it is likely this difference is negligible. Plan 9 adopts a very similar memory management scheme to Unix, and so fork semantics are roughly equivalent, including copy-on-write. (The following citation contains a breakdown of almost every line in the Plan 9 kernel source code). [http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.75.5409rep=rep1type=pdf] The number of cycles involved in forking a process is likely huge, however, and in the tens of thousands of cycles, since all sorts of meta data (such as the page table) needs to be copied over. There aren't many good resources online which have benchmarked the number of cycles involved in a fork.

Since the software overhead of a fork is the fork itself, the hardware and software costs are pretty much interlinked in this particular experiment. Thus, we ignore a software multiplier, setting it equal to 1.

That puts our estimate at around 60,000 cycles or 30,000 nsec for both fork operations. We do not believe that Plan 9's lightweight fork operation will help much, and that most of the resources and state will still need to be copied to the child.

Our results are recorded in table 4; we were roughly in line with our estimates, but there is something quite interesting going on with Plan 9's heavy fork, the equivalent of running a process vs. a kernel thread. There is no difference in average fork time, but there is a huge variation in the two. Heavy forks have a long tail; in future work, it may be interesting to plot this long tail, and analyze the impact it has on end-user applications.

In general, programmers should opt for light forks, if possible. Just don't bend them!

3.6 Context Switch Time

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.

We estimated the base hardware cost of context switching while pinned to a single CPU to be on the order of around 1600 ns based upon benchmarks done on the Intel E5-2620, another Sandy Bridge processor model. [https://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html]

Our prediction for the software cost of this is going to be quite large, most likely around 10x, since there are so many factors involved and our method of triggering a context switch is not the most scientific method due to limitations in low-level threading primitives for Plan 9.

This puts our prediction right around 16 usec; the cost of the context switch itself may be less than this in Plan 9, but in order to measure this, we will need to develop modifications to the kernel. These modifications are reserved for future work.

The results are included in table 5. Our estimates were a bit too pessimistic, however, they were close to the truth, as the results were skewed to values even greater than 16 microseconds.

In general, our methodology could be improved; a lack of familiarity with such a niche system reduced our ability to benchmark effectively.

3.7 RAM Access Time

To measure ram access time we allocate a 1.6 GB array on the heap, and iterated through it with changing stride size to hit L1, L2 cache and finally memory. The stride size gets increased by factor 1.01 for each iteration. Predictions are presented in fig. 6, while the measurements are presented in fig. 7. Figure 8 presents the plot of the mean in orange and standard deviation in blue with the last index accessed in the iteration on the *x* axis, and memory access latency is displayed on a logarithmic scale *y* axis. The measurements are not completely free of overhead, but still a transition from L1 to L2, and from L2 to memory is visible in the graph.

We estimated the base hardware cost of memory accesses based upon well documented knowledge of the timings of such memory accesses. According to work done on breaking AES, L1 cach is available after 3 cycles, L2 after 11, and memory after this takes hundreds of

prediction L1 latency	2-3 cycles	0.8-1.2 nsec
prediction L2 latency	10 cycles	4 nsec
prediction memory latency	100 cycles	40 nsec

Figure 6: RAM access time prediction

L1 latency	6 nsec
L2 latency	10 nsec
Memory	250 nsec

Figure 7: RAM access time measurements

cycles. [http://palms.ee.princeton.edu/system/files/Cache-timing+attacks+on+AES.pdf]

For measurements like the one outlined above, there is very little software cost involved; thus our software overhead estimate is a multiplier of 1.

As a result, our prediction is that the memory access times under Plan 9 will correspond well with other operating systems and personal computers, particularly those running x86 processors.

And our results, shown in tables 6, 7, and figure 8, reveal that our prediction was, indeed, correct. Standard deviations were high for some portions, due to natural variations in experimental conditions. This is an important result from a security point of view, as it reveals that Plan 9 may be vulnerable to significant timing side channel attacks, something which should be addressed by the community in due time, especially considering the distributed nature of the Plan 9 system.

Our methodology has revealed a common expectation and typical empirical feature of x86 processors, and thus we are confident in our methodology.

3.8 RAM Bandwidth

We measured RAM bandwidth by allocating an array of $5 \cdot 10^6$ 4-byte integers, and iterated through it to measure read and write bandwidth. For the write bandwidth, we use `memset` to write to the entire array. In order to measure the read bandwidth, we looped through the entire array, referencing each integer element one by one, and compensated for the loop overhead after averaging over all the times.

Our memory bus bandwidth is about 10GB/s, so this serves as our predicted bandwidth for both reading and writing. We don't expect much software overhead because we can directly read from and write to memory, provided the pages are faulted in for us, and the cost of these faults should be amortized. We still will most likely not reach peak performance, however, due to various OS interruptions. Writing will likely be slower than reading, as bits must be set on disk, and there may be more guarantees of correctness involved with writing, as well as memory organization issues.

And our results, shown in table 9, reveal that our prediction was roughly correcy. Write bandwidths were around 70% of the total possible bandwidth, and writes were 90%. Cache line prefetching may also be speeding up the reads, and improving bandwidth where it may not be improved by raw bus speed. Disabling optimizations, read speeds dropped dramatically.

Our methodology was approximate and modified by software in a myriad of ways, but we feel it is reasonable enough to say that Plan 9 was effectively taking advantage of bandwidth. It may be worth looking into improving write speeds; either via improvment of the benchmark or the kernel.

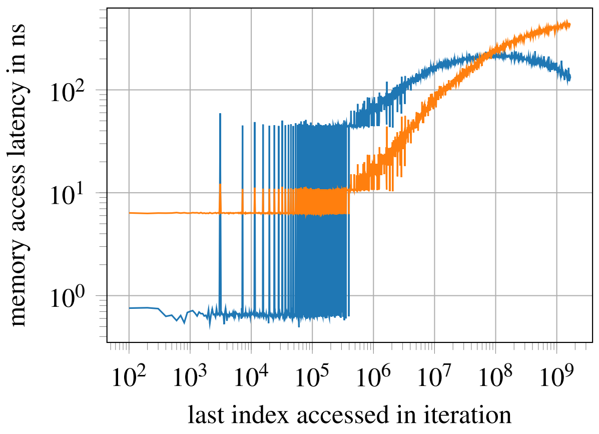

Figure 8: RAM access time with size of the memory region accessed on x axis in an logarithmic scale and logarithmic latency in seconds on the y axis. The orange graph displays the mean latency, while the blue graph displays the standard deviation. L1 cache accesses end around 10^4 and L2 cache accesses end between 10^5 and 10^6, where there are clear increases in the orange line.

	Bandwidth (GiB/s)	
Theoretical Maximum r/w	10	
Software Multiplier	0.8	
Prediction	8	
	Average	Std Dev.
Write	6.879842	1.9829
Read[2]	9.109494	1.2845

Figure 9: Memory Bandwidth

3.9 Page Fault Service Time

Since getting pages to reliably swap to disk in Plan 9 remained difficult due to imposed memory limits on processes, as well as random process deaths for no good reason, as well as failing calls to `brk`, and crashes when the swap file was overloaded. There was also no ability to mmap. In light

[2]This measurement was done using optimization, without it the read speed is 1.17

	Page fault time (ns)	
Hardware	3000	
Software Multiplier	1	
Prediction	3000	
	Average	Std Dev.
Actual Timing	2370.5	57.084

Figure 10: Page Fault Servicing

of the instability of high-memory demand approaches, we chose a different approach to page fault service time. In order to measure the Page Fault service time, we repeatedly ran executable files with a globally mapped 4 kilobyte array, read in a single byte, and then stopped execution. We timed the time it took to read that single byte; concurrent monitoring of page fault statistics on the machine informed us that this was causing 16 repeated page faults from disk. Thus, we had our measurement of the service time.

Articles were split on the timing for page faults: putting it around 3 microseconds for an SSD. Thus, we expected page fault timings to be somewhere around this. https://pdfs.semanticscholar.org/dd0a/dcde7d074a414a9df76fb20d52a0d8aa8c71.pdf

In the end, this prediction seemed to be a bit too conservative; the actual page fault service timing of the SSD was around 2.3 microseconds. Our results are summarised exactly in table 10. The SSD tested on is only a few months old, and it is possible that access speeds have gone up. It is also possible that we are missing some latency due to caching effects, though this is unlikely.

This methodology is quite rough, but it is the best which we were able to do given the constraints placed upon us by Plan 9 in terms of forcing page faults.

Dividing by a page size of 4096 bytes, we have roughly 2300 ns per 4096 bytes, or 0.578 ns per byte. This is much faster than accessing a single byte from main memory, since during a page fault we are able to stream in the data all at once, even though the time required to service the fault is high.

3.10 Size of File Cache

In this section we report our estimation for the size of file cache. In order to measure the file cache we find the time taken to perform reads over a calculated number of bytes. Our benchmark always reads 16KiB (4 page) blocks at a time. For file sizes below 16KiB, we simply read the file again and again until we reach enough bytes to make a comparison. In effect we find the amount of time to read a 4 page block of data for increasing file sizes. All this is done within one trial, and we average over 32 trials done for each file size, which ranged from 1B to 128MB, in increments of powers of two.

We imagine the in-memory file cache to be generously large (¿64MB). We estimate that reads from the file cache should be within a magnitude of our RAM bandwidth as measured in the previous section. The software overhead of a file cache should not be too large - we must perform some checks in order to see whether a file has been mapped to the cache, and we may feel some effects of initially mapping a file into cache. This overhead should be small, i.e. decrease the bandwidth by no more than 10%.

When reading from the file cache, we would then expect our read bandwidth to be at about 7GB/s.

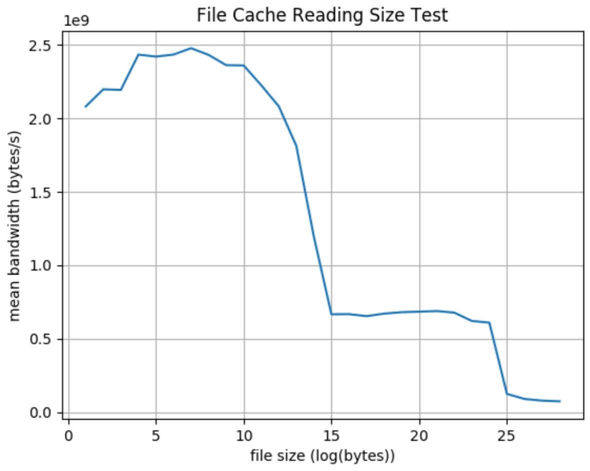

Figure 11: Graph of file read bandwidth vs. file size

3.11 File Read Time

To measure the file read time, we vary the file size we are reading over from 1B to 4MB. We then read from these files 16KB at a time, compensating in the files smaller than 16KB by reading more times. This comprised one trial, and we averaged over 4 trials for each given file size. We found this small amount of trials to be sufficient because of the large file sizes we operated over.

We tried to do measure sequential reads by reading sequentially through files we created earlier, but how sequential this is depends on the physical placement of data blocks by the file system, and how many levels of indirection a block can support. This also may not matter altogether because of the lack of a long and variable seek time in our SSD, which has no mechanically moving parts.

Random reads were done by seeking to random offsets within the file after every read.

We could not find an raw interface to access the disk itself in order to skip the effects of the file buffer cache. Instead we specified a hint to the file interface to disable the use of buffering reads and writes using `setvbuf`.

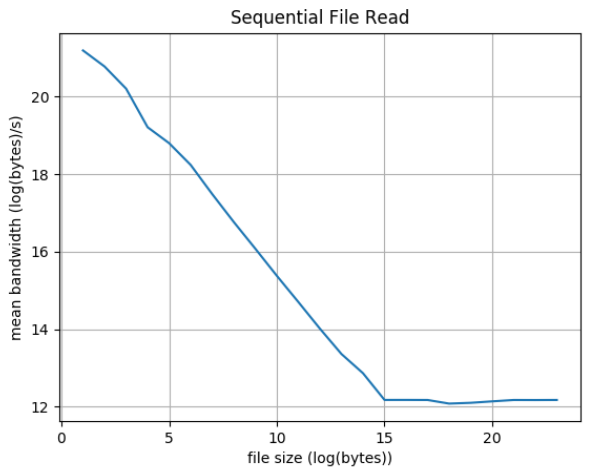

Figure 12: Graph of file sequential read bandwidth vs. file size

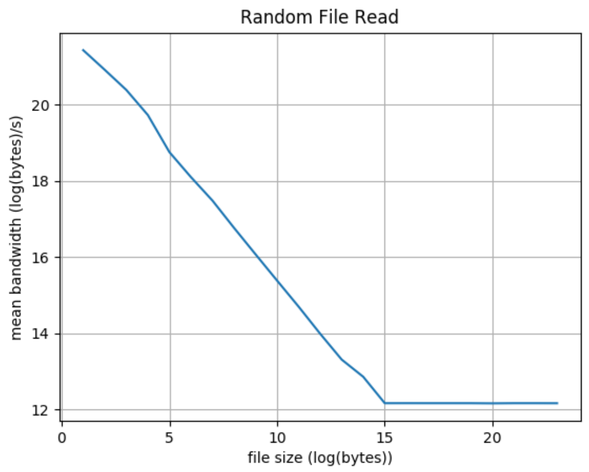

Figure 13: Graph of file random read bandwidth vs. file size

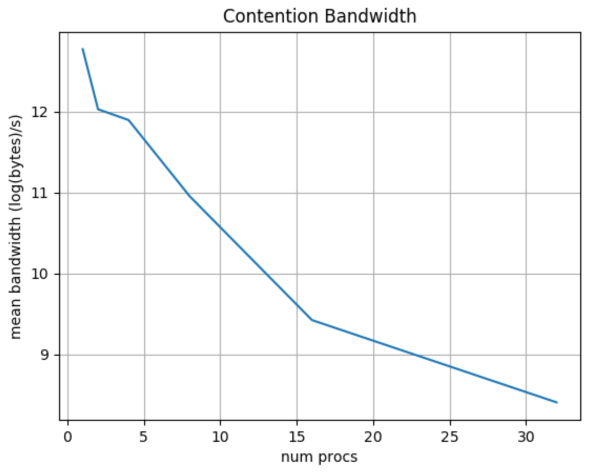

Figure 14: Graph of file read bandwidth vs. number of contending processors

3.12 Remote File Read Time

Finally, we are able to take advantage of one of Plan 9's most unique features: remote filesystem mounting. We were able to mount a public Plan 9 file server over the 9p protocol and test the remote read speed simply by measuring the time to read a remote file into main memory. Again we averaged only over two trials, since we found that we were already averaging the read speed over many bytes (a 5MB file) anyways.

3.13 Contention

To measure the effect of contending processes performing reads on different files in the filesystem, our benchmark spawns up to 32 processes in powers of two, and at each level of number of processes conntending for reads, we read from random 1MB long files (of which there are 32 to choose from) in 4K sized chunks. We then measure the bandwidth experienced by a constant process as more processes are forked in the background.

3.14 Discussion

Compare the measured performance with the predicted performance. If they are wildly different, speculate on reasons why. What may be contributing to the overhead?

This go-around our paper is probably less structured than previously, since we all worked on the various sections at the same time. It most likely needs some polishing and a read through before it is really ready. However, we have cleaned up the code for our prior benchmarks, so we should be getting pretty accurate results at this point.

Evaluate the success of your methodology. How accurate do you think your results are? Our results are now a bit faster than the Plan 9 paper, which is expected given that our processor is faster, although I don't think we've rigorously sanity checked them. It was hard to ensure we were taking the best approach, but in each case it seems that we measured what we were trying to measure, to good approximation.

4 Conclusion

Okay, so this time we had a bit less time to clean up the writing and make sure everything is clear, so we might need to go back through a bit of this in the coming two weeks and make it readable so that the final report is something we can be proud of. Other than that, it seems we have most of our code working as expected, and a nice development workflow inside Plan 9 itself; I'm writing this inside an emacs editor inside of an mrxvt window inside of *equis* graphical window multiplexed via a linux emulator running on top of an rc shell window inside of rio inside of Plan 9.

As opposed to earlier weeks, it should be smooth sailing from here. Once we clean up this paper a bit and present our results coherently, we should be ready to tackle the next part of the benchmarking. We probably need to get some feedback on our results so far, but that can wait till next monday. It has been a crazy ride!