Investigating the Linux Scheduler

Operating Systems - CSCI 3753

Samuel Volin, Alexander Tsankov, and Cristobol Salazar

Computer Science, University of Colorado Boulder

November 3, 2014

Abstract

A computer will run hundreds of processes every second, and the algorithm responsible for managing the run order of these processes is known as the scheduler. This paper examines different scheduler policies (non-preemptive first-in-first-out, round robin, completely-fair scheduling), how they respond to certain conditions, such as number of cores, I/O bound versus CPU bound processes, and measures the workload actively being managed by the Linux Scheduler.

1 Introduction

In our experiment, we operate our scheduler with three different scheduling policies: The Completely Fair Scheduler (SCHED_OTHER), a non-preemptive First-In-First-Out (SCHED_FIFO), and a Round-Robin scheduling algorithm (SCHED_RR). This experiment measured the performance of these scheduler policies in conjunction with the other independent variables, such as the number of spawned processes, and the level of io or cpu bounds of the processes.

Three different types of programs were used in this experiment. The first program (CPU) performs a CPU-bound task by calculating the value of pi a certain amount of times. The second program (IO) performs an I/O bound task by reading bytes from an input file (102400 bytes) and outputting those bytes to a different duplicate file, imitating the process of reading and writing from and to a disk. The third program (MIX) does a combination of both in a sequential order.

Each program takes in a number of arguments that set up the independent variables for the test. These include the scheduling policy to be used, and the number of simultaneous processes to be spawned per test. A fixed number of child processes are spawned via the fork system call (fork()) to run at the same time. The different workloads range from Singular (1), Low (8), Medium (64), to High (128), each denoting the number of simultaneous process instances spawned at run-time. The parent process waits for all child processes to complete and the results are measured.

Finally, we performed this experiment on a virtual machine running Lubuntu 13.10 with access to four operating cores, and then performed a second time on the same machine with only one activated core. More information about the device can be found in Appendix C.

To reiterate, the independent variables for this experiment are the scheduling policy used, the program type, the number of processes spawned, and the number of active cores used to measure run these tests.

We measured a number of dependant variables with the following tests. These variables are listed below:

1. (Wall): total measure of seconds for every forked program to conclude.
2. (User): total cpu seconds spent processing this program in user-mode
3. (System): total cpu seconds spent processing this program in kernel-mode
4. (CPU): percentage of the CPU that this job got, computed as (User + System) / Wall
5. (Wait-Time): total cpu seconds not scheduled in, computed as Wall - (System + User)
6. (V-Switched): number of times the program context switched voluntarily (went IO bound)
7. (I-Switched): number of times the program context switched involuntarily (was swapped out)

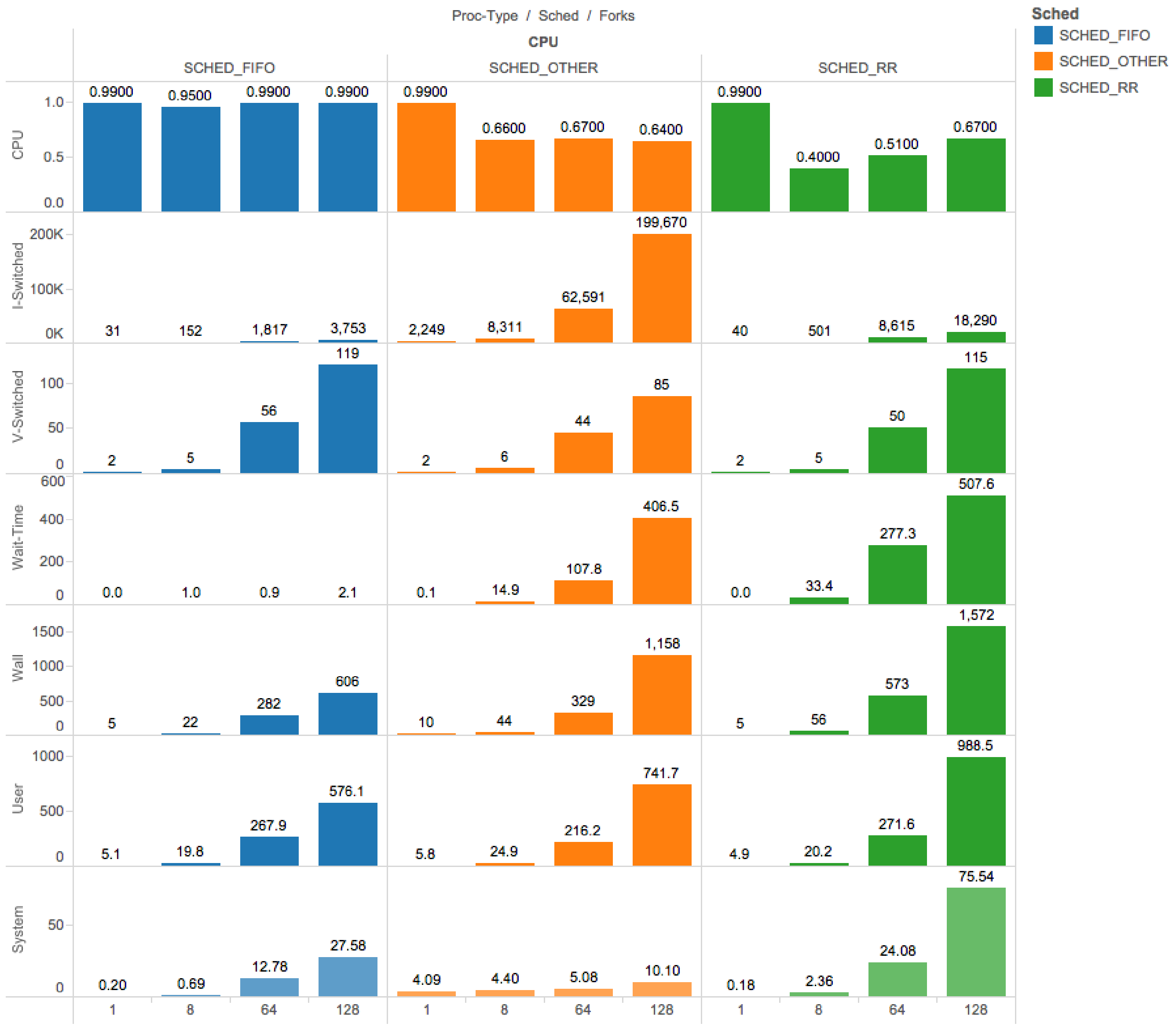
A bash script was created to automatically run all of the tests and output the values of the measured dependant variables. The time benchmarks look at the aforementioned characteristics. The script is listed below in Appendix B.

2 Results

The tests were executed from the bash script and run on a Linux virtual machine running Lubuntu (Ubuntu-derivitve) 13.10 set to (1) core and (4) cores, respectively. The computer had a SATA solid-state hard drive and a quad-core Intel i7 processor. The data produced by the bash script was saved as a .csv, and imported into a graphing software package called Tableau. With that, we produced the following results.

2.1 1 core, CPU-bound

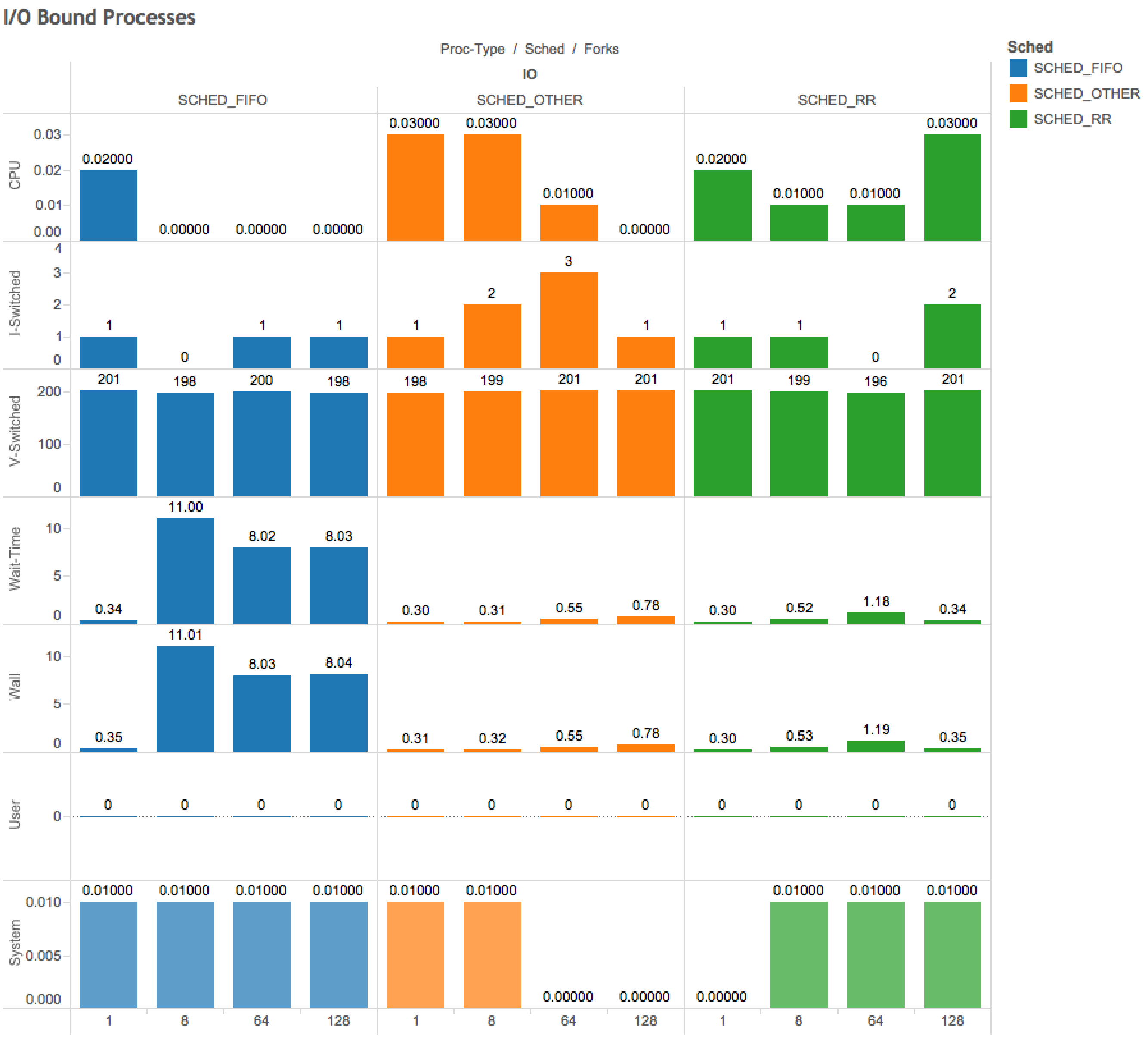
CPU Bound Processes



Sum of CPU, sum of I-Switched, sum of V-Switched, sum of Wait-Time, sum of Wall, sum of User and sum of System for each Forks broken down by Proc-Type and Sched. Color shows details about Sched. The view is filtered on Proc-Type, which keeps CPU.

For this set of tests, optimal performance was observed with the SCHED_FIFO policy, and the incurred wait-time and context-switches was minimized by the FIFO policy. Of course, a finite CPU-bound process is optimized when left to run in it's entirety. However, the CPU bound process monopolized the system and would show noticeable degradation if left to run alongside other programs. Of note, the completely fair scheduler minimized time spent in kernel mode. Both SCHED_FIFO and SCHED_OTHER performed better than the SCHED_RR counterpart.

2.2 1 core, IO-bound



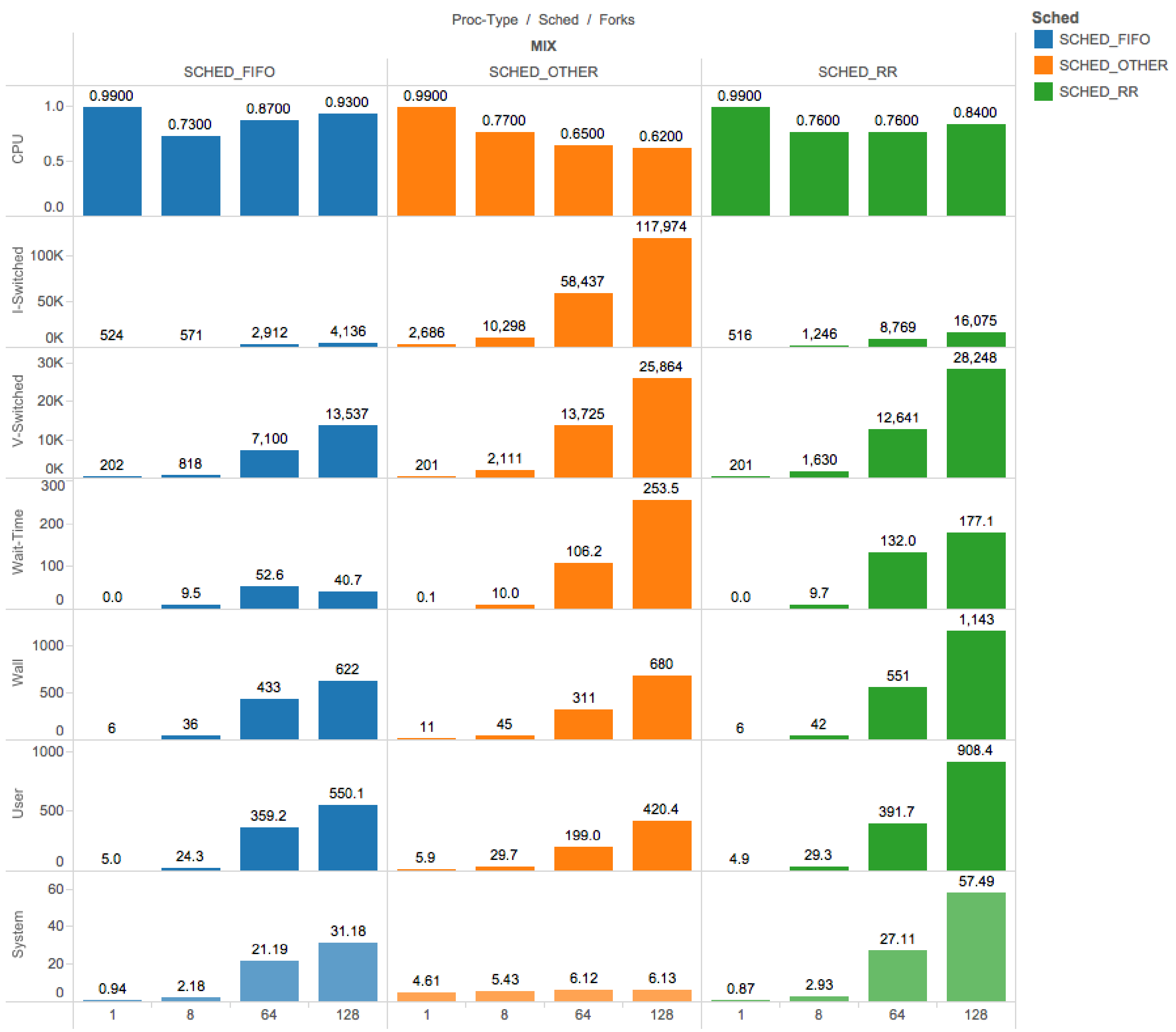
Metric	FIFO 1	FIFO 8	FIFO 64	FIFO 128	OTHER 1	OTHER 8	OTHER 64	OTHER 128	RR 1	RR 8	RR 64	RR 128
CPU	0.02000	0.00000	0.00000	0.00000	0.03000	0.03000	0.01000	0.00000	0.02000	0.01000	0.01000	0.03000
I-Switched	1	0	1	1	1	2	3	1	1	1	0	2
V-Switched	201	198	200	198	198	199	201	201	201	199	196	201
Wait-Time	0.34	11.00	8.02	8.03	0.30	0.31	0.55	0.78	0.30	0.52	1.18	0.34
Wall	0.35	11.01	8.03	8.04	0.31	0.32	0.55	0.78	0.30	0.53	1.19	0.35
User	0	0	0	0	0	0	0	0	0	0	0	0
System	0.01000	0.01000	0.01000	0.01000	0.01000	0.01000	0.00000	0.00000	0.00000	0.01000	0.01000	0.01000

Sum of CPU, sum of I-Switched, sum of V-Switched, sum of Wait-Time, sum of Wall, sum of User and sum of System for each Forks broken down by Proc-Type and Sched. Color shows details about Sched. The view is filtered on Proc-Type, which keeps IO.

Note that the values for System and User (time spent cpu-bound in user-mode and kernel mode) is close to zero, and the number of voluntary switches are equal between all three scheduling policies. Note also the number of involuntary switches is effectively zero, in comparison to the involuntary switches of cpu and mix processes. Almost no time was measured to be CPU-bound for any scheduling policy. Significant time spent swapped out and waiting was counted with the FIFO scheduler, suggesting that for multiple IO-bound processes, SCHED_RR is the most effective scheduling process for I/O bound processes.

2.3 1 core, MIX-bound

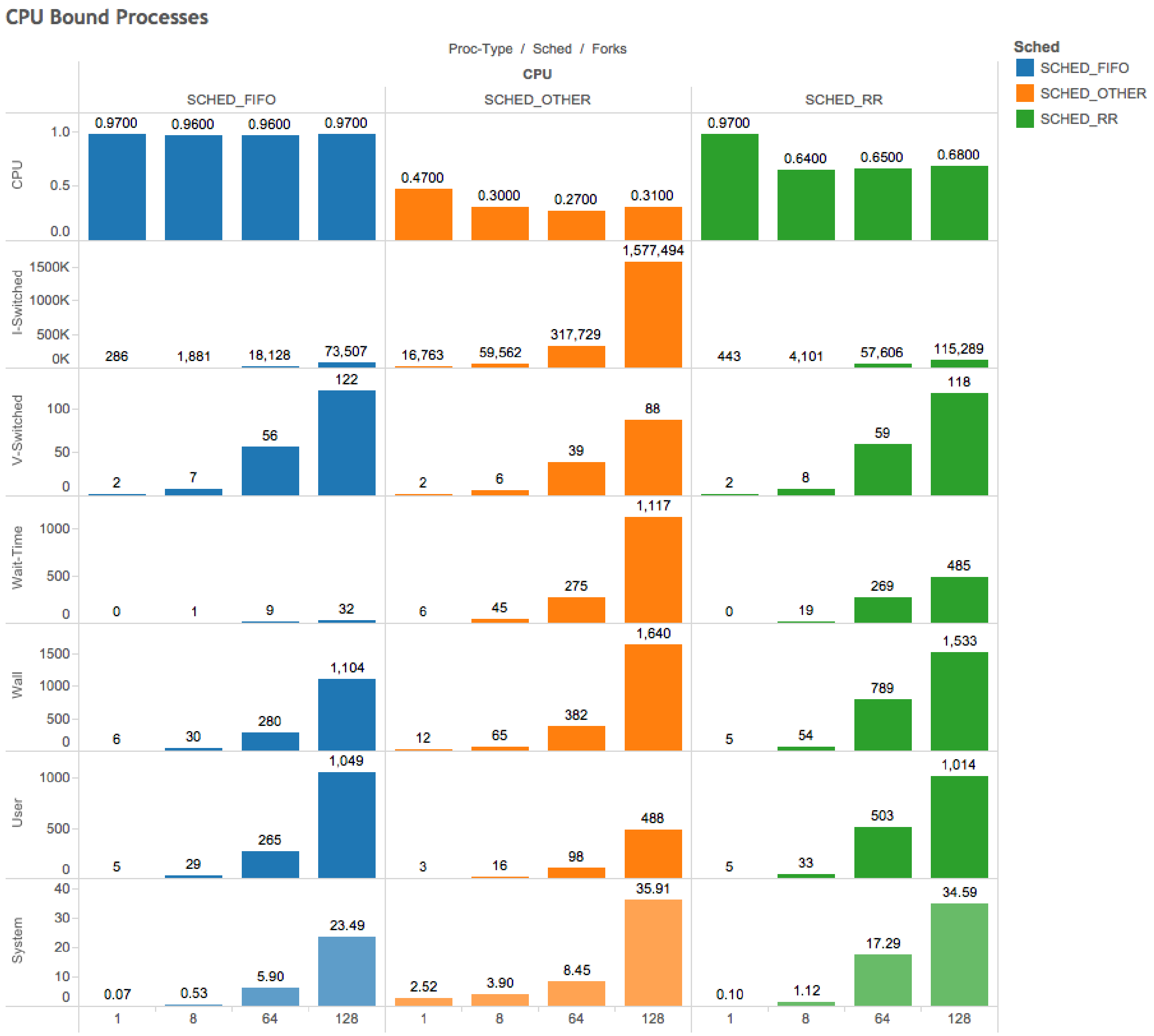
IO/CPU Mix



Sum of CPU, sum of I-Switched, sum of V-Switched, sum of Wait-Time, sum of Wall, sum of User and sum of System for each Forks broken down by Proc-Type and Sched. Color shows details about Sched. The view is filtered on Proc-Type, which keeps MIX.

Interesting results were found in pairs of scheduling policies and dependant variables. The SCHED_OTHER policy appears to yield more involuntary context switches, and by association, a greater wait-time than the other policies. The Round-Robin scheduler took the longest to run the programs, with the highest Wall, User and System times attributed to it. The SCHED_OTHER policy had near uniform time spent in kernel mode for all values of fork. This further reinforces the notion that SCHED_OTHER is a good medium when going between hybrid I/O and CPU-bound processes.

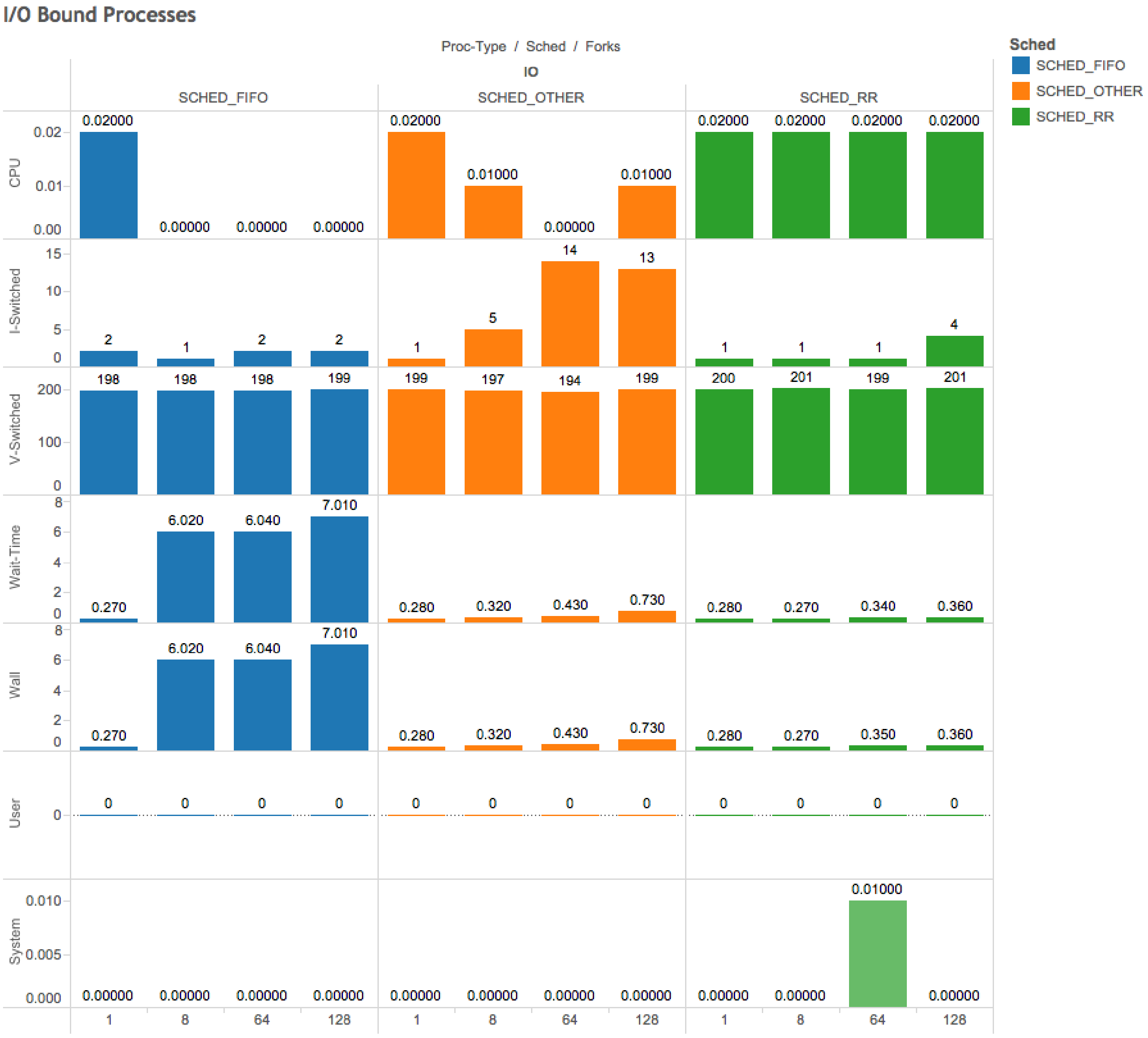
2.4 4 core, CPU-bound



Sum of CPU, sum of I-Switched, sum of V-Switched, sum of Wait-Time, sum of Wall, sum of User and sum of System for each Forks broken down by Proc-Type and Sched. Color shows details about Sched. The view is filtered on Proc-Type, which keeps CPU.

The quad-core test showed only marginal performance loss in regards to more time spent in kernel mode and user mode. Best performance with minimal context switching was observed with the FIFO scheduler. Overall, more cores seems to have increased Wall and wait-time as opposed to the single-core counterpart. We suspect this was due to an unclean testing environment with other processes running.

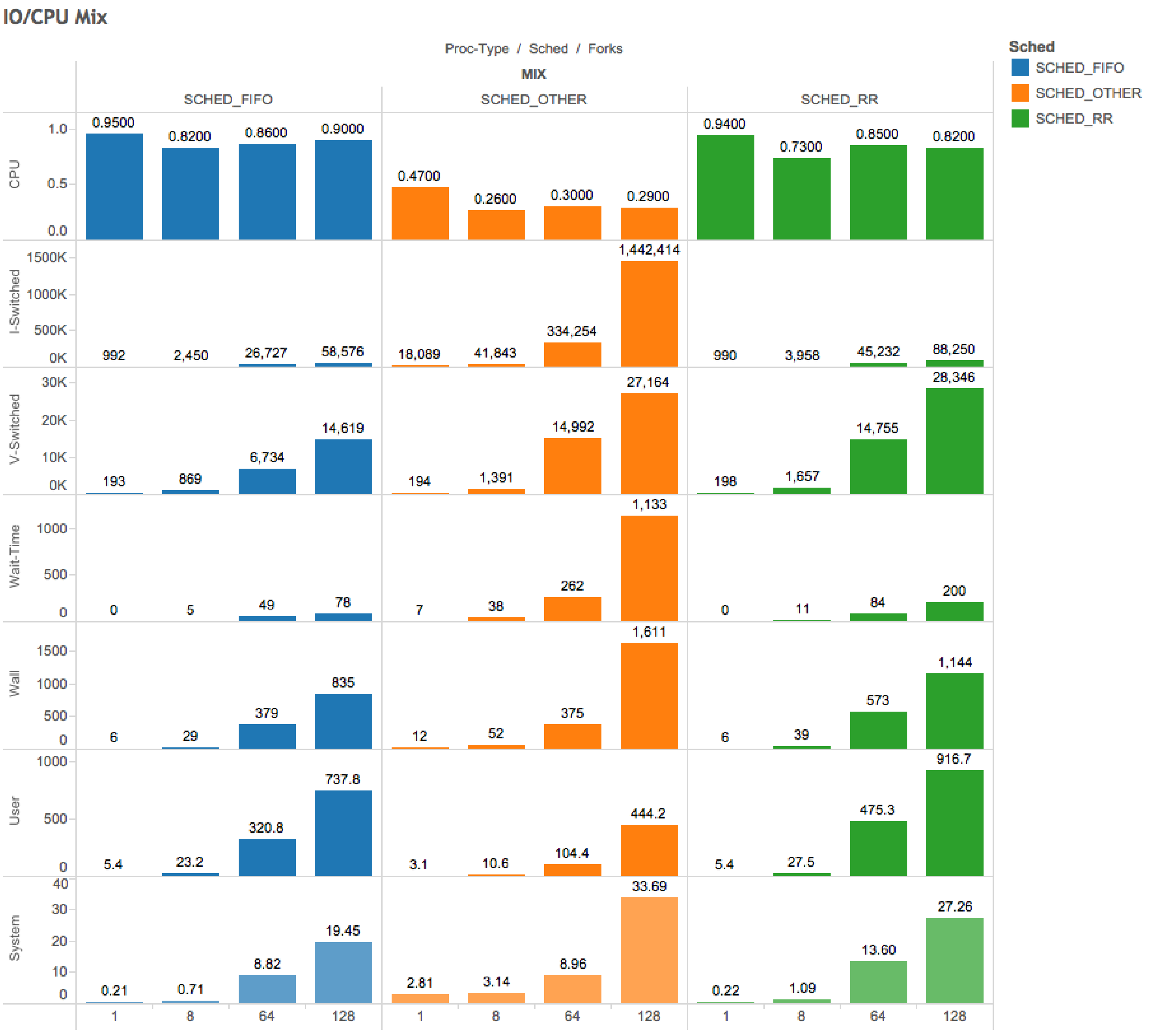
2.5 4 core, IO-bound



Sum of CPU, sum of I-Switched, sum of V-Switched, sum of Wait-Time, sum of Wall, sum of User and sum of System for each Forks broken down by Proc-Type and Sched. Color shows details about Sched. The view is filtered on Proc-Type, which keeps IO.

The single-core system had similar results to the quad-core test, with similar conclusions: wait-time is compounded with the fifo policy, voluntary switches are identical across all policies and processes, and virtually no time is spent in cpu-bound in user mode or kernel mode. If the number of cores used highlighted one difference, it seems the SCHED_OTHER policy exacerbated the number of involuntary switches that occur.

2.6 4 core, MIX-bound



Metric	FIFO 1	FIFO 8	FIFO 64	FIFO 128	OTHER 1	OTHER 8	OTHER 64	OTHER 128	RR 1	RR 8	RR 64	RR 128
CPU	0.9500	0.8200	0.8600	0.9000	0.4700	0.2600	0.3000	0.2900	0.9400	0.7300	0.8500	0.8200
I-Switched	992	2,450	26,727	58,576	18,089	41,843	334,254	1,442,414	990	3,958	45,232	88,250
V-Switched	193	869	6,734	14,619	194	1,391	14,992	27,164	198	1,657	14,755	28,346
Wait-Time	0	5	49	78	7	38	262	1,133	0	11	84	200
Wall	6	29	379	835	12	52	375	1,611	6	39	573	1,144
User	5.4	23.2	320.8	737.8	3.1	10.6	104.4	444.2	5.4	27.5	475.3	916.7
System	0.21	0.71	8.82	19.45	2.81	3.14	8.96	33.69	0.22	1.09	13.60	27.26

Sum of CPU, sum of I-Switched, sum of V-Switched, sum of Wait-Time, sum of Wall, sum of User and sum of System for each Forks broken down by Proc-Type and Sched. Color shows details about Sched. The view is filtered on Proc-Type, which keeps MIX.

Comparing the single core mix-bound process to the multi-core mix-bound process will note one big interesting result: on average, it took longer for the multicore machine to run these tasks For the FIFO and OTHER schedulers, it took ten times as many involuntary context switches and four times longer processing time. Overall, less time was spent in kernel mode in the multi-core test, but otherwise worse performance was observed.

3 Analysis

The results from the first experiment, in section 2.1, show that the clear optimal scheduling process for CPU-bound processes on one core is SCHED_FIFO. This can be expected because SCHED_FIFO runs processes until a process yields or completes and all of the processes are finite. SCHED_FIFO can run all of the processes to completion before switching a process out. SCHED_RR will switch out a process after its quanta has elapsed, thus incurring lengthier overhead spent swapping out processes. SCHED_OTHER has a significantly large amount of involuntary switches. This can be a good thing because it allows for other programs to step in and decreases the risk of starvation of processes. The high number of involuntary switches, again, incurs overhead and time penalties.

In Section 2.2, displaying IO-bound processes on a single core, there is a large increase in voluntary switches across all schedulers. This is expected because as the processes is blocking on IO, it will voluntarily yield thus allowing another process to step in. The results also show very few involuntary switches. We hypothesize two possible reasons for this. One, the processes are writing so few bytes to the solid-state hard disk (Sata 128 GB) that the processes are already done before there is a need to be swapped out. The other possible reason is that due to the increased voluntary switches, every time a process blocks on IO, the quanta gets reset. This would suggest the processes never take the full quanta before voluntarily switching.

The MIX sample program, displayed in Section 2.3, shows SCHED_OTHER having a significantly large number of involuntary switches. In this test, the relatively higher involuntary switches compared to the IO-bound test shows that when CPU-bound elements are introduced into a process, SCHED_OTHER makes significantly more involuntary switches. Another data point to acknowledge is that SHED_RR had a large Wall, User, and System time. Comparing back to I/O- and CPU-bound tests, the data supports that SCHED_RR performs very poorly, regarding time, when CPU-bound elements are in the running process. This provides more evidence that the over-head from switching out processes more frequently can severely decrease time performance. This suggests that large processes can benefit from an increased time quanta, but that might incur starvation in other processes

The experiments run with 4 cores gave similar results as the 1 core experiments. There is a surprising decrease in time performance from the 1 core to the 4 core tests. It would be expected that the 4 core would be faster; this may be a source of error. One potential reason for this error could be that background processes were running during the tests. A virtual machine was used for the experiment, and when it was given 4 cores the VM must share the cores with the native operating system. So if applications were running on the native OS, they could have affected the results because the 4 cores need to be shared.

4 Conclusion

From the results we can conclude that different properties and conditions suit different schedulers. The SCHED_FIFO is optimal for finite CPU-bound processes. The reason it performs so well is that it allows processes to run until the process yields or ends. This can allow for higher time performance because the scheduler is not paying the over-head costs of switching processes frequently. SCHED_FIFO is not optimal though when dealing with infinite or very long processes. This will lead to starvation of other processes in the FIFO if an infinite process is using all of the CPU.

SCHED_RR is rarely the correct choice of scheduler. While it decreased time performance, SCHED_RR will switch out processes to ensure no process starves. Even though it has this advantage over

SCHED_FIFO, SCHED_OTHER performs the same round-robin scheduling, but SCHED_OTHER is faster and has better performance across the board than SCHED_RR. For IO heavy processes, it works effectively as most switches are voluntary and the overhead incurred negligible.

SCHED_OTHER is the best of the three schedulers because it can be used to achieve a balance between speed and fairness. SCHED_OTHER is not as fast as SCHED_FIFO on CPU-bound processes, but it is faster than SCHED_RR. SCHED_OTHER also voluntarily switches processes after a certain quanta, thus allowing resources to the shared with all processes.

5 Appendix A: Raw Data

1 core Raw Data:

Proc-Type	Sched	Forks	wall	user	system	CPU	i-switched	v-switched	wait-time
CPU	OTHER	1	9.94	5.8	4.09	0.99	2249	2	0.05
IO	OTHER	1	0.31	0	0.01	0.03	1	198	0.3
MIX	OTHER	1	10.56	5.89	4.61	0.99	2686	201	0.06
CPU	FIFO	1	5.27	5.06	0.2	0.99	31	2	0.01
IO	FIFO	1	0.35	0	0.01	0.02	1	201	0.34
MIX	FIFO	1	5.97	5	0.94	0.99	524	202	0.03
CPU	RR	1	5.12	4.92	0.18	0.99	40	2	0.02
IO	RR	1	0.3	0	0	0.02	1	201	0.3
MIX	RR	1	5.81	4.92	0.87	0.99	516	201	0.02
CPU	OTHER	8	44.22	24.93	4.4	0.66	8311	6	14.89
IO	OTHER	8	0.32	0	0.01	0.03	2	199	0.31
MIX	OTHER	8	45.12	29.71	5.43	0.77	10298	2111	9.98
CPU	FIFO	8	21.55	19.84	0.69	0.95	152	5	1.02
IO	FIFO	8	11.01	0	0.01	0	0	198	11
MIX	FIFO	8	35.98	24.31	2.18	0.73	571	818	9.49
CPU	RR	8	55.99	20.23	2.36	0.4	501	5	33.4
IO	RR	8	0.53	0	0.01	0.01	1	199	0.52
MIX	RR	8	41.92	29.31	2.93	0.76	1246	1630	9.68
CPU	OTHER	64	329.06	216.23	5.08	0.67	62591	44	107.75
IO	OTHER	64	0.55	0	0	0.01	3	201	0.55
MIX	OTHER	64	311.36	199.02	6.12	0.65	58437	13725	106.22
CPU	FIFO	64	281.52	267.87	12.78	0.99	1817	56	0.87
IO	FIFO	64	8.03	0	0.01	0	1	200	8.02
MIX	FIFO	64	432.95	359.17	21.19	0.87	2912	7100	52.59
CPU	RR	64	572.99	271.59	24.08	0.51	8615	50	277.32
IO	RR	64	1.19	0	0.01	0.01	0	196	1.18
MIX	RR	64	550.8	391.69	27.11	0.76	8769	12641	132
CPU	OTHER	128	1158.33	741.7	10.1	0.64	199670	85	406.53
IO	OTHER	128	0.78	0	0	0	1	201	0.78
MIX	OTHER	128	679.99	420.41	6.13	0.62	117974	25864	253.45
CPU	FIFO	128	605.77	576.12	27.58	0.99	3753	119	2.070
IO	FIFO	128	8.04	0	0.01	0	1	198	8.03
MIX	FIFO	128	621.95	550.06	31.18	0.93	4136	13537	40.710
CPU	RR	128	1571.64	988.49	75.54	0.67	18290	115	507.61
IO	RR	128	0.35	0	0.01	0.03	2	201	0.34
MIX	RR	128	1142.99	908.44	57.49	0.84	16075	28248	177.06

4 core Raw Data:

Proc-Type	Sched	Forks	wall	user	system	CPU	i-switched	v-switched	wait-time
CPU	OTHER	1	11.81	3.11	2.52	47%	16763	2	6.18
IO	OTHER	1	0.28	0	0	2%	1	199	0.28
MIX	OTHER	1	12.42	3.11	2.81	47%	18089	194	6.5
CPU	FIFO	1	5.57	5.38	0.07	97%	286	2	0.12
IO	FIFO	1	0.27	0	0	2%	2	198	0.27
MIX	FIFO	1	5.94	5.43	0.21	95%	992	193	0.3
CPU	RR	1	5.4	5.15	0.1	97%	443	2	0.15
IO	RR	1	0.28	0	0	2%	1	200	0.28
MIX	RR	1	5.98	5.43	0.22	94%	990	198	0.33
CPU	OTHER	8	65.1	15.85	3.9	30%	59562	6	45.35
IO	OTHER	8	0.32	0	0	1%	5	197	0.32
MIX	OTHER	8	51.83	10.59	3.14	26%	41843	1391	38.1
CPU	FIFO	8	30.01	28.53	0.53	96%	1881	7	0.95
IO	FIFO	8	6.02	0	0	0%	1	198	6.02
MIX	FIFO	8	28.97	23.16	0.71	82%	2450	869	5.1
CPU	RR	8	53.57	33.16	1.12	64%	4101	8	19.29
IO	RR	8	0.27	0	0	2%	1	201	0.27
MIX	RR	8	39.14	27.5	1.09	73%	3958	1657	10.55
CPU	OTHER	64	382.06	98.37	8.45	27%	317729	39	275.24
IO	OTHER	64	0.43	0	0	0%	14	194	0.43
MIX	OTHER	64	375.29	104.43	8.96	30%	334254	14992	261.9
CPU	FIFO	64	280.08	265.04	5.9	96%	18128	56	9.14
IO	FIFO	64	6.04	0	0	0%	2	198	6.04
MIX	FIFO	64	378.95	320.79	8.82	86%	26727	6734	49.34
CPU	RR	64	788.97	502.9	17.29	65%	57606	59	268.78
IO	RR	64	0.35	0	0.01	2%	1	199	0.34
MIX	RR	64	572.66	475.26	13.6	85%	45232	14755	83.8
CPU	OTHER	128	1640.09	487.5	35.91	31%	1577494	88	1116.68
IO	OTHER	128	0.73	0	0	1%	13	199	0.73
MIX	OTHER	128	1610.53	444.16	33.69	29%	1442414	27164	1132.68
CPU	FIFO	128	1104.29	1049.27	23.49	97%	73507	122	31.53
IO	FIFO	128	7.01	0	0	0%	2	199	7.01
MIX	FIFO	128	834.97	737.81	19.45	90%	58576	14619	77.71
CPU	RR	128	1533.29	1013.97	34.59	68%	115289	118	484.73
IO	RR	128	0.36	0	0	2%	4	201	0.36
MIX	RR	128	1144.33	916.73	27.26	82%	88250	28346	200.34

6 Appendix B: Code

```
#!/bin/bash
ITERATIONS=100000000
BYTESTOCOPY=102400
BLOCKSIZE=1024
# e: elapsed real time in seconds
# u: Total number of CPU-seconds that the process spent in user mode.
# S: seconds spent in kernel mode,
# P: Percentage of the CPU that this job got, computed as (%U + %S) /E,
# c: number of times the program context switched voluntarily,
# w: Number of waits: times that the program was context-switched voluntarily.
#waitTime: Realtime - (System time + user time) -> e - (u+s)

#TIMEFORMAT="wall=%e,user=%U,system=%S,CPU=%P,i-switched=%c,v-switched=%w"
TIMEFORMAT="%e,%U,%S,%P,%c,%w"
MAKE="make -s"

echo Building code...
$MAKE clean
$MAKE

COUNT=0
SCHED=SCHED_RR

ARR=(cpu io mix)

#loop through fork COUNT
for i in `seq 1 4`;
do
case $i in
1) #single
COUNT=1
;;
2) #low
COUNT=8
;;
3) #medium
COUNT=64
;;
*) #high
COUNT=128
;;
esac
#loop through SCHEDuler type
for j in `seq 1 3`;
do
case $j in
1) #other
```

```
SCHED=SCHED_OTHER
;;
2) #fifo
SCHED=SCHED_FIFO
;;
*) #round robin
SCHED=SCHED_RR
;;
esac
#loop through process type

#cpu bound
echo CPU,$SCHED,$COUNT
        /usr/bin/time -f "$TIMEFORMAT" ./pi_fork $ITERATIONS $SCHED $COUNT > /dev/null

        #io bound
        echo IO,$SCHED,$COUNT
        /usr/bin/time -f "$TIMEFORMAT" ./rw_fork $BYTESTOCOPY $BLOCKSIZE rwinput rwoutput

        #mix of both
        echo MIX,$SCHED,$COUNT
        /usr/bin/time -f "$TIMEFORMAT" ./mix_fork $BYTESTOCOPY $BLOCKSIZE rwinput rwoutput
done
done

rm -rf rwoutput-*
rm -rf rwinput*
```

7 Appendix C: proc_info

```
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 69
model name : Intel(R) Core(TM) i7-4650U CPU @ 1.70GHz
stepping : 1
cpu MHz : 2344.303
cache size : 6144 KB
physical id : 0
siblings : 4
core id : 0
cpu cores : 4
apicid : 0
initial apicid : 0
fpu : yes
fpu_exception : yes
cpuid level : 5
wp : yes
bogomips : 4688.60
clflush size : 64
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual
power management:
```