Homework 1 - Virtual Machine Scheduling Analysis

Jared Klingenberger

February 8, 2016

Listings

1 Rosenblum Paper

To summarize, Rosenblum's paper on virtual machines is a good overview of the numerous motivations behind virtual machines. In the early days of virtual machine computing, a big motivation was for compatibility between different types of hardware by having a single target abstract machine to code to. However, a different style of virtual machine made a huge comeback in recent years with the arrival of VMWare, a hardware-level virtual machine that made it possible to emulate computer hardware devices. This is quite useful for many reasons; two such reasons are that these VMs provide superior isolation and that one can manage the execution of these VMs from a higher level with a Virtual Machine Manager.

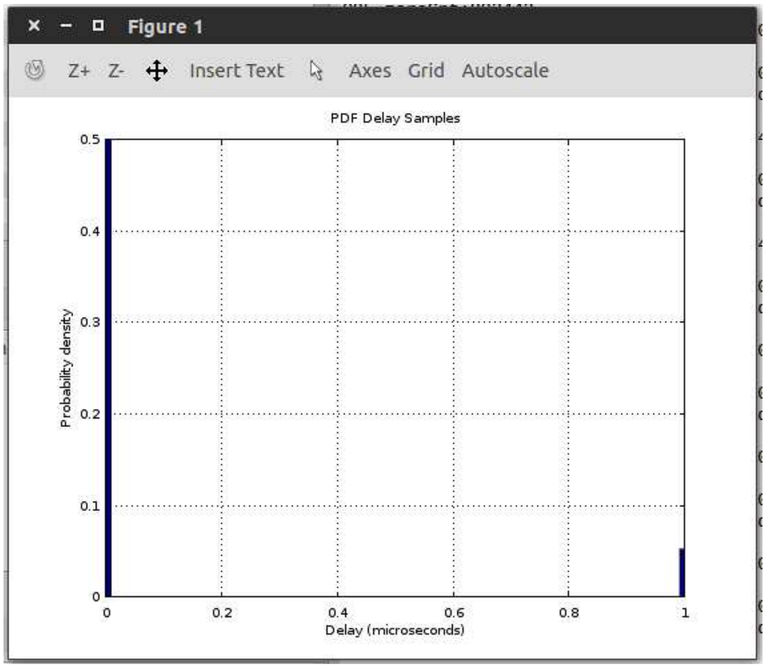
2 Manual Dataset Creation and Analysis

Overall for question 2, I noticed that as I increased the blocksize, there seemed to be noticeable periodic effects in the datasets. So most of the results would fall around the same value, but some delays were much longer and stacked around the same times. This likely has something to do with how the scheduler assigns clock time to different processes. The most noticeable effect can be seen with blocksize of 1 million.

2.1 Statistics

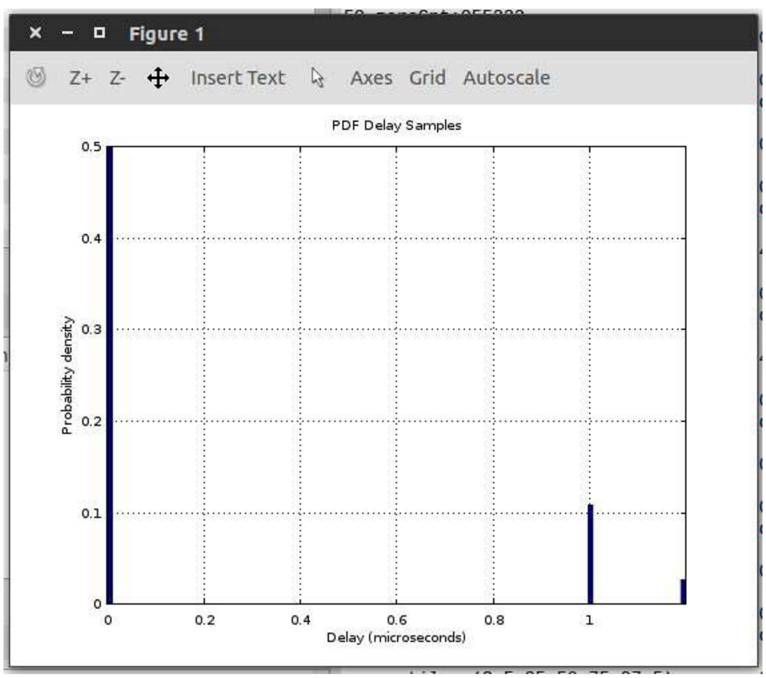
Here are histogram plots and numerical statistics for different blocksizes.

2.1.1 Blocksize 4



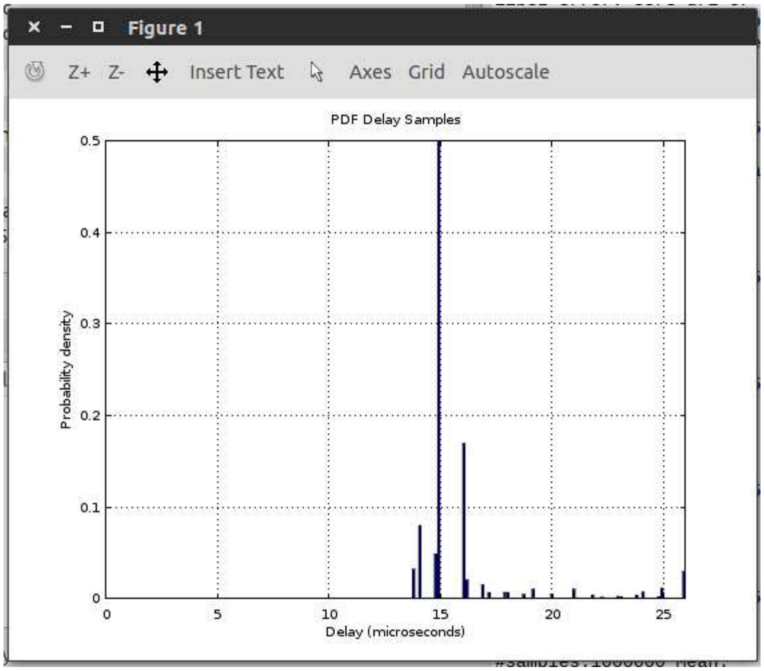

```
plotDataPDF (nargin:1):  sampleSize:1000000, maxValue:65.100000,  MAX_ALLOWED_VALUE:1000.000000
#samples:1000000 Mean:     0 microseconds,  median:      0,  std:      0, max:    65, min:      0, maxCount:10173
     zeroCnt:947490
percentiles (2.5 25 50 75 97.5): :     0      0      0      0      1
```

2.1.2 Blocksize 256



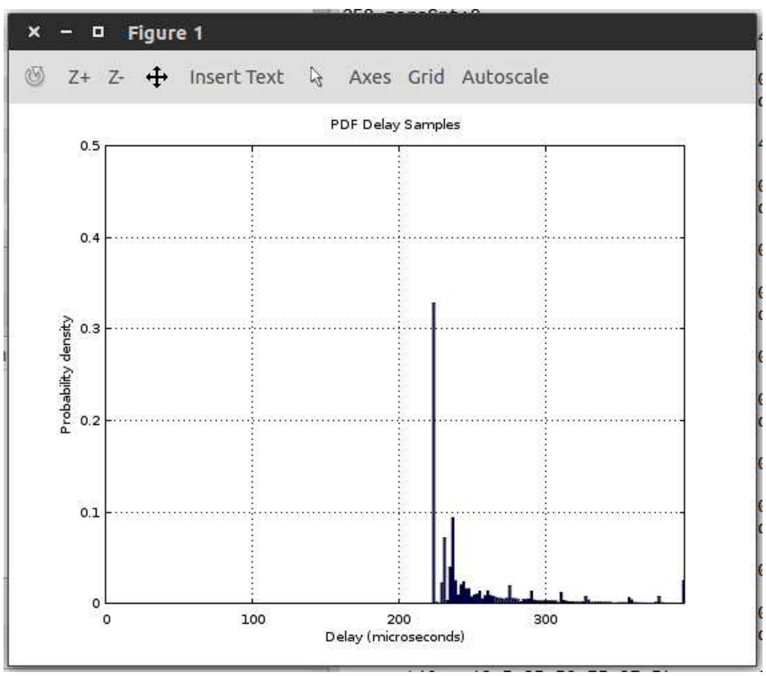

```
plotDataPDF (nargin:1):  sampleSize:1000000, maxValue:70.100000,  MAX_ALLOWED_VALUE:1000.000000
#samples:1000000 Mean:     0 microseconds,  median:      0,  std:      0, max:    70, min:      0, maxCount:235
    zeroCnt:865266
percentiles (2.5 25 50 75 97.5): :     0      0      0      0      1
```

2.1.3 Blocksize 65535




```
plotDataPDF (nargin:1):  sampleSize:1000000, maxValue:4635.100000,  MAX_ALLOWED_VALUE:1000.000000
#samples:1000000 Mean:    16 microseconds,  median:     15,  std:      7, max:  4635, min:    12, maxCount:22358
     zeroCnt:0
percentiles (2.5 25 50 75 97.5): :    14     15     15     16     26
```

2.1.4 Blocksize 1000000




```
plotDataPDF (nargin:1):  sampleSize:1000000, maxValue:5667.200000,  MAX_ALLOWED_VALUE:1000.000000
#samples:1000000 Mean:   255 microseconds,  median:    236,  std:     51, max:  5667, min:   210, maxCount:24825
     zeroCnt:0
percentiles (2.5 25 50 75 97.5): :   223    224    236    263    395
```

2.2 Parallel execution

I think the programs were completing too quickly to affect mpstat, but I was able to see a spike in CPU usage through using the ‘top‘ command.

```
11:30:35 PM  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
11:30:35 PM  all    0.38    0.00    0.03    0.00    0.00    0.00    0.00    0.00    0.00   99.58
11:30:35 PM    0    0.46    0.00    0.05    0.00    0.00    0.00    0.00    0.00    0.00   99.48
11:30:35 PM    1    0.34    0.01    0.03    0.00    0.00    0.00    0.00    0.00    0.00   99.62
11:30:35 PM    2    0.35    0.00    0.03    0.00    0.00    0.00    0.00    0.00    0.00   99.62
11:30:35 PM    3    0.36    0.00    0.02    0.01    0.00    0.00    0.00    0.00    0.00   99.61
```

2.2.1 Statistics - shell 1

```
plotDataPDF (nargin:1):  sampleSize:1000000, maxValue:683.100000,  MAX_ALLOWED_VALUE:1000.000000
#samples:1000000 Mean:     0 microseconds,  median:      0,  std:      1, max:   683, min:     0, maxCount:259
    zeroCnt:835032
percentiles (2.5 25 50 75 97.5): :     0      0      0      0      1
```

2.2.2 Statistics - shell 2

```
plotDataPDF (nargin:1):  sampleSize:1000000, maxValue:90.100000,  MAX_ALLOWED_VALUE:1000.000000
#samples:1000000 Mean:     0 microseconds,  median:      0,  std:      1, max:    90, min:     0, maxCount:284
    zeroCnt:835508
percentiles (2.5 25 50 75 97.5): :     0      0      0      0      1
```

3 Experiments in Parallel VM Scheduling

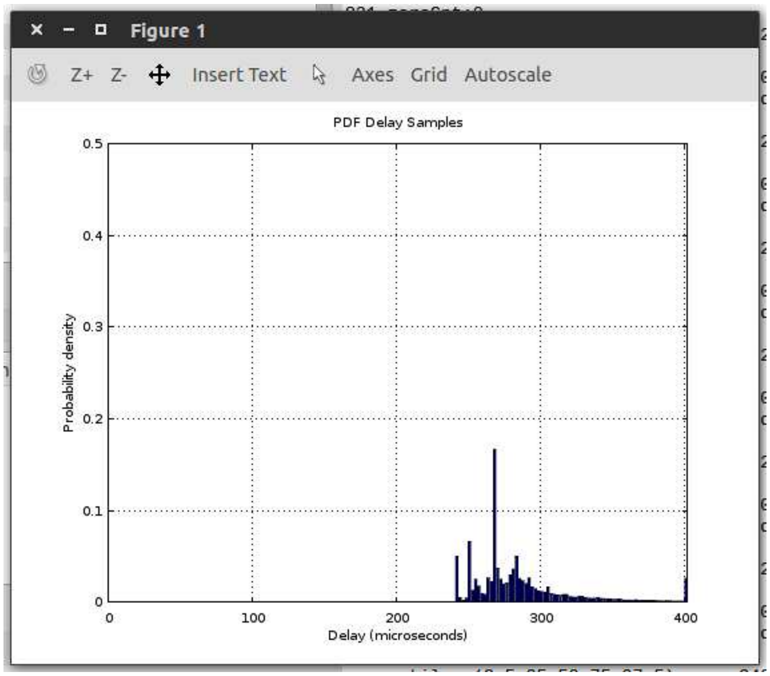
3.1 Experiment 1

This first experiment was simply an exercise in automating the testing process. With a blocksize that varies by $2^i, i \in \{1, 2, ..., 20\}$, the runExp.sh program automatically performs the experiment all at once.

3.1.1 Overall stats

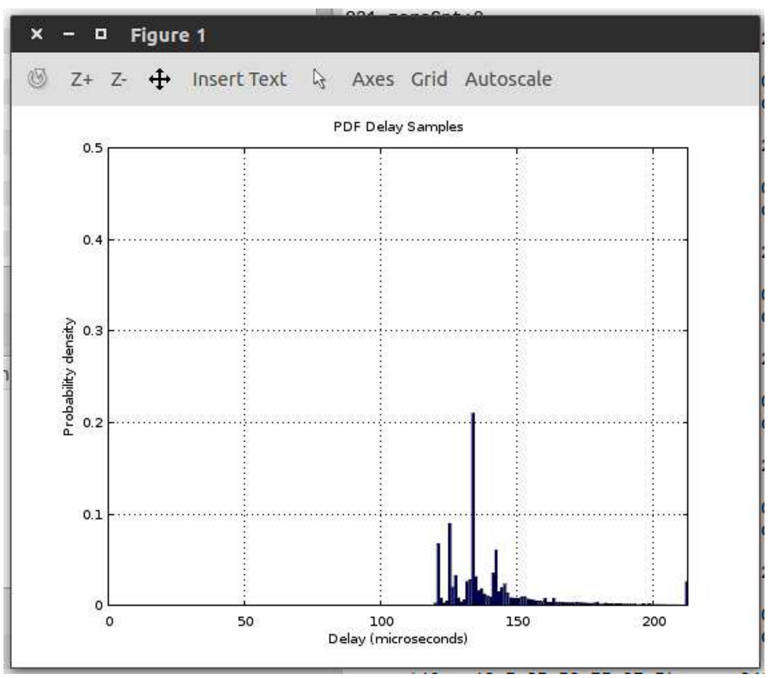
```
Experiment 1
1000000 1024 0.400 0.000 2.408
1000000 1048576 287.784 275.900 53.599
1000000 128 0.153 0.000 0.569
1000000 131072 36.570 33.900 17.870
1000000 16 0.082 0.000 1.135
1000000 16384 4.572 4.100 2.826
1000000 2 0.092 0.000 0.493
1000000 2048 0.782 1.000 3.638
1000000 256 0.278 0.000 2.141
1000000 262144 73.582 67.900 18.960
1000000 32 0.061 0.000 0.422
1000000 32768 9.244 8.100 6.677
1000000 4 0.095 0.000 0.630
1000000 4096 1.190 1.000 1.365
1000000 512 0.211 0.000 3.652
1000000 524288 143.082 134.000 32.334
1000000 64 0.084 0.000 1.034
1000000 65536 18.039 16.900 6.872
1000000 8 0.093 0.000 0.495
1000000 8192 2.286 2.100 1.999
```

3.1.2 Statistics - blocksize 1048576




```
plotDataPDF (nargin:1):  sampleSize:1000000, maxValue:5829.100000,  MAX_ALLOWED_VALUE:1000.000000
#samples:1000000 Mean:   288 microseconds,  median:    276,  std:     54, max:  5829, min:   184, maxCount:24693
     zeroCnt:0
percentiles (2.5 25 50 75 97.5): :   242    267    276    296    402
```

3.1.3 Statistics - blocksize 524288




```
plotDataPDF (nargin:1):  sampleSize:1000000, maxValue:9854.100000,  MAX_ALLOWED_VALUE:1000.000000
#samples:1000000 Mean:   143 microseconds,  median:    134,  std:     32, max:  9854, min:   107, maxCount:24795
     zeroCnt:0
percentiles (2.5 25 50 75 97.5): :   121    132    134    145    213
```

3.1.4 Interpretation

The results of this experiment are pretty clear. The larger blocksize definitely has a higher variability, which can be seen visually in the plot by how the distribution is more spread. This is probably because with more bits to checksum, there is a higher chance of scheduling affecting the timing with more granularity.

3.2 Experiment 2

Experiment 2 was more interesting because it was meant to do the same thing as experiment 1, but with the program competing for the CPU by running two instances of hw1 in parallel. This is what runExp.sh does – simply runs experiment 1 twice simultaneously.

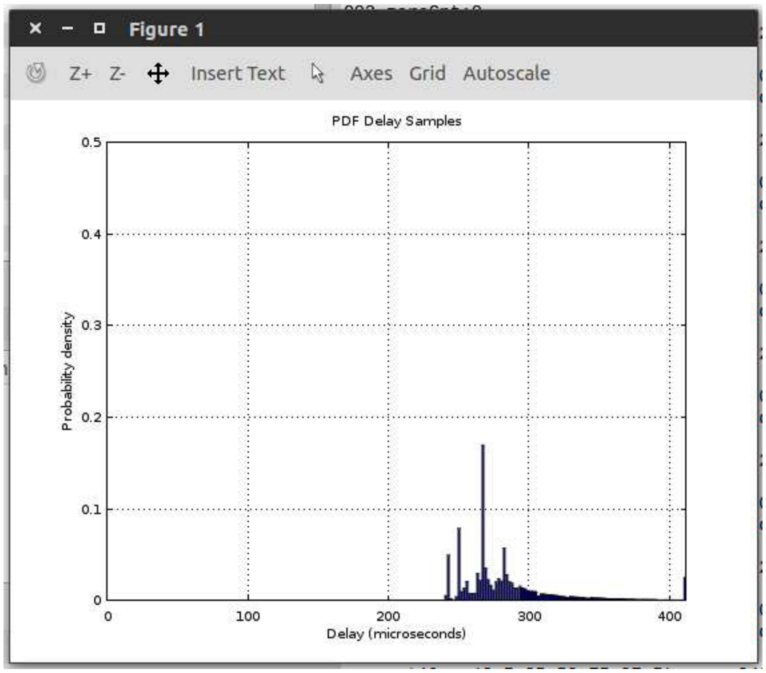
3.2.1 Overall stats - core 1

```
Experiment 2
1000000 1024 0.396 0.000 7.013
1000000 1048576 287.971 273.900 67.686
1000000 128 0.118 0.000 0.550
1000000 131072 35.803 33.100 14.463
1000000 16 0.078 0.000 1.587
1000000 16384 4.431 4.100 11.559
1000000 2 0.092 0.000 0.430
1000000 2048 0.828 1.000 1.000
1000000 256 0.299 0.000 2.810
1000000 262144 70.562 67.000 19.655
1000000 32 0.049 0.000 0.384
1000000 32768 8.706 8.100 4.817
1000000 4 0.091 0.000 1.428
1000000 4096 1.196 1.000 4.301
1000000 512 0.176 0.000 1.366
1000000 524288 144.228 134.000 36.245
1000000 64 0.080 0.000 1.815
1000000 65536 17.792 16.900 10.709
1000000 8 0.088 0.000 0.429
1000000 8192 2.240 1.900 5.022
```

3.2.2 Overall stats - core 2

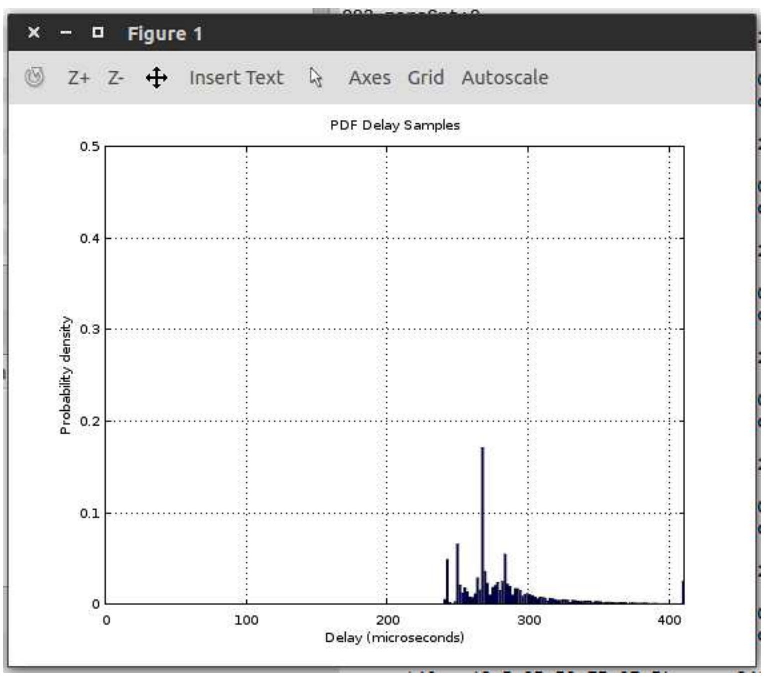
```
Experiment 2
1000000 1024 0.395 0.000 3.535
1000000 1048576 287.680 273.000 65.885
1000000 128 0.118 0.000 0.492
1000000 131072 35.738 33.100 15.173
1000000 16 0.075 0.000 1.518
1000000 16384 4.416 4.100 6.982
1000000 2 0.091 0.000 0.523
1000000 2048 0.831 1.000 3.241
1000000 256 0.284 0.000 2.709
1000000 262144 70.584 67.000 21.541
1000000 32 0.051 0.000 0.405
1000000 32768 8.735 8.100 4.718
1000000 4 0.081 0.000 1.261
1000000 4096 1.203 1.000 7.994
1000000 512 0.174 0.000 0.694
1000000 524288 143.792 134.000 40.463
1000000 64 0.084 0.000 4.540
1000000 65536 17.807 16.900 9.195
1000000 8 0.087 0.000 0.446
1000000 8192 2.237 1.900 5.171
```

3.2.3 Statistics - core 1 (blocksize 1048576)




```
plotDataPDF (nargin:1):  sampleSize:1000000, maxValue:22301.900000,  MAX_ALLOWED_VALUE:1000.000000
#samples:1000000 Mean:   288 microseconds,  median:    274,  std:     68, max: 22302, min:   223, maxCount:24821
     zeroCnt:0
percentiles (2.5 25 50 75 97.5): :   242    265    274    294    412
```

3.2.4 Statistics - core 2 (blocksize 1048576)




```
plotDataPDF (nargin:1):  sampleSize:1000000, maxValue:25430.000000,  MAX_ALLOWED_VALUE:1000.000000
#samples:1000000 Mean:   288 microseconds,  median:    273,  std:     66, max: 25430, min:   233, maxCount:24657
     zeroCnt:0
percentiles (2.5 25 50 75 97.5): :   242    265    273    294    411
```

3.2.5 Interpretation

In this experiment, the differences were really hard to discern. I didn't notice any striking changes when scaling up the blocksize, nor when comparing the statistics of each core. I suspect that this is because my VM had four cores to work with, thanks to how I configured the VM. In the future, I may want to increase the number of parallel processes at work.

3.3 Experiment 3

This experiment is largly a repeat of experiment 2, but the difference comes in to how each process is prioritized. I assign one core a nice value of 0 and the other core a nice value of 19. (Normal users cannot set negative nice values by default in Ubuntu.)

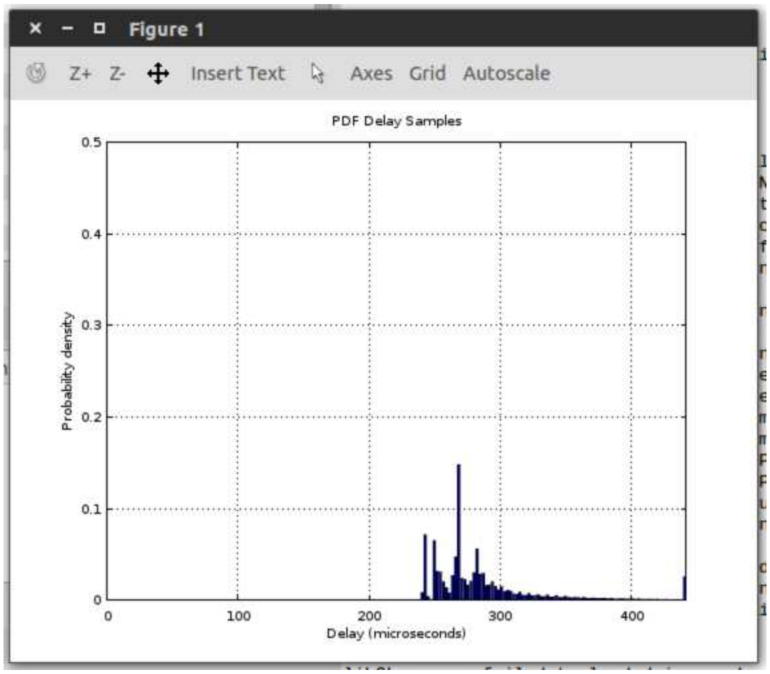
3.3.1 Overall stats - core 1

```
Experiment 3
1000000 1024 0.386 0.000 5.003
1000000 1048576 289.974 272.000 73.226
1000000 128 0.116 0.000 2.633
1000000 131072 37.451 33.900 21.779
1000000 16 0.080 0.000 1.124
1000000 16384 4.463 4.100 4.246
1000000 2 0.096 0.000 0.546
1000000 2048 0.881 1.000 5.817
1000000 256 0.273 0.000 0.611
1000000 262144 72.662 67.000 27.157
1000000 32 0.047 0.000 1.658
1000000 32768 8.811 8.100 6.075
1000000 4 0.092 0.000 2.417
1000000 4096 1.193 1.000 2.703
1000000 512 0.155 0.000 2.092
1000000 524288 141.603 134.000 33.523
1000000 64 0.132 0.000 7.266
1000000 65536 17.985 16.900 12.531
1000000 8 0.090 0.000 1.283
1000000 8192 2.354 1.900 8.776
```

3.3.2 Overall stats - core 2

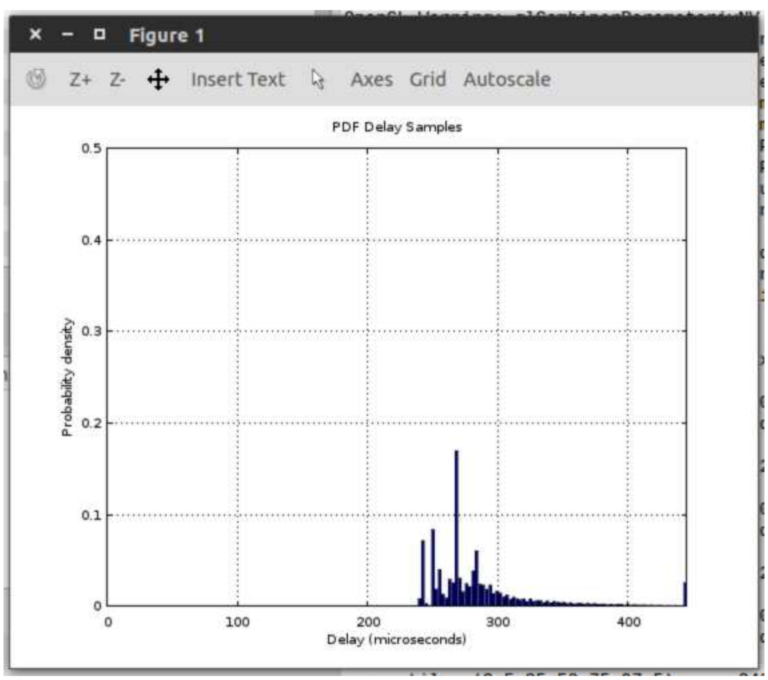
```
Experiment 3
1000000 1024 0.385 0.000 3.961
1000000 1048576 290.656 272.000 76.704
1000000 128 0.115 0.000 2.711
1000000 131072 37.443 33.900 27.228
1000000 16 0.060 0.000 1.825
1000000 16384 4.478 4.100 6.304
1000000 2 0.096 0.000 0.549
1000000 2048 0.865 1.000 5.666
1000000 256 0.286 0.000 0.668
1000000 262144 72.529 67.000 29.583
1000000 32 0.042 0.000 1.999
1000000 32768 8.815 8.100 4.721
1000000 4 0.096 0.000 1.277
1000000 4096 1.201 1.000 3.675
1000000 512 0.163 0.000 6.748
1000000 524288 141.794 134.000 40.413
1000000 64 0.071 0.000 1.391
1000000 65536 18.100 16.900 16.403
1000000 8 0.091 0.000 1.125
1000000 8192 2.346 1.900 7.503
```

3.3.3 Statistics - core 1 (blocksize 1048576)




```
plotDataPDF (nargin:1):  sampleSize:1000000, maxValue:7099.900000,  MAX_ALLOWED_VALUE:1000.000000
#samples:1000000 Mean:   290 microseconds,  median:    272,  std:     73, max:  7100, min:   209, maxCount:24881
     zeroCnt:0
percentiles (2.5 25 50 75 97.5): :   242    262    272    294    442
```

3.3.4 Statistics - core 2 (blocksize 1048576)




```
plotDataPDF (nargin:1):  sampleSize:1000000, maxValue:12529.900000,  MAX_ALLOWED_VALUE:1000.000000
#samples:1000000 Mean:   291 microseconds,  median:    272,  std:     77, max: 12530, min:   187, maxCount:24902
     zeroCnt:0
percentiles (2.5 25 50 75 97.5): :   242    262    272    295    445
```

3.3.5 Interpretation

In experiment 3, there was slightly more variability in the lower priority process. This makes intuitive sense, since the scheduler will have a higher chance of skipping over the core if other system processes need the processor more.

4 Code

Listing 1: hw1.c

```c
/************************************************************
 * exampleUnixTime -  this program illustrates how to obtain
*    a Unix timestamp, how to compute a RTT, and how to print
*    a timestamp as a human readable string.
*
* Notes:
*
* Revisions
*   $A0: 1-21-2016  release v1.0
*
 ************************************************************/
#include "hw1.h"

void signal_handler();
double getTime();


#define TEST_SLEEP 1
// 1: test sleep  for TEST_SLEEP seconds
// 0: loop forever running instructions
#define LOOP_TEST  1


int main(int argc, char *argv[])
{
  double t = 0.0;
  struct sigaction sig;
  unsigned int k = 0;
  unsigned int iters = atoi(argv[1]); //Purposefully not validating input
  unsigned int bs = atoi(argv[2]);
  int sum;
  unsigned char *block = NULL;

  /* initialize RNG */
  srand(time(NULL));

  /* setup signals */
  memset(&sig,0,sizeof(sig));
  sig.sa_handler = signal_handler;
  sigaction (SIGINT,  &sig, NULL);
  sigaction (SIGALRM, &sig, NULL);

  /* allocate and initialize data block */
  block = (unsigned char*)malloc(sizeof(unsigned char) * bs);
  memset(block,rand()%256,sizeof(unsigned char)*bs);

  while (k < iters)
  {
    t = getTime();
    sum = csum((unsigned short*)block, bs);
    t = getTime() - t;
    printf("%3.7f 0X%xd \n",t,sum);
    k++;
  }

  exit(0);
}

double getTime()
{
  struct timeval curTime;
  (void) gettimeofday (&curTime, (struct timezone *) NULL);
  return (((((double) curTime.tv_sec) * 1000000.0)
             + (double) curTime.tv_usec) / 1000000.0);
}

void signal_handler(int sig) {
  switch(sig) {
    case SIGINT:
      printf("SIGINT:   Exit! \n");
      exit(0);
```

```
      break;
    case SIGALRM:
      printf("SIGALRM:   Exit! \n");
      exit(0);
      break;
  }
}
```

Listing 2: runHW1.sh

```
#!/bin/bash

if [[ $# -ne 5 ]]; then
  echo "Invalid parameters."
  echo "Syntax: $0 [iterationCountP] [firstScalingP] [numberRuns]
  [resultsDirectory] [baseDataFileName]"
  exit 1
fi

mkdir -p $4
rm -rf "$4/*"

for i in `seq $2 \`expr $2 + $3\` | awk '{print 2 ^ $1}'`; do
  echo "Running $1 iterations on blocksize $i..."
  ./hw1 $1 $i > "$4/$5.$i"
done
```

Listing 3: runExp.sh

```
#!/bin/bash

if [[ $# -ne 6 ]]; then
  echo "Invalid parameters."
  echo "Syntax: $0 [exp#] [parameters 1-5 for ./runHW1.sh]"
  exit 1
fi

ctrl_c() {
  for job in `jobs -p`; do
    kill -9 $job
  done
}

exp1() {
  ./runHW1.sh $1 $2 $3 $4 $5
}

exp2() {
  trap ctrl_c SIGINT
  ./runHW1.sh $1 $2 $3 $4 "$5.1" &
  ./runHW1.sh $1 $2 $3 $4 "$5.2" &
  # Wait until background tasks complete
  for job in `jobs -p`; do
    wait $job
  done
  trap - SIGINT
}

exp3() {
  trap ctrl_c SIGINT
  # Users can't set negative nice values by default
  nice -n 0 ./runHW1.sh $1 $2 $3 $4 "$5.1" &
  nice -n 20 ./runHW1.sh $1 $2 $3 $4 "$5.2" &
  # Wait until background tasks complete
  for job in `jobs -p`; do
    wait $job
  done
  trap - SIGINT
}

case $1 in
  1)
    echo "Experiment 1!"
    exp1 $2 $3 $4 $5 $6
    echo "Done."
```

```
    ;;
  2)
    echo "Experiment 2!"
    exp2 $2 $3 $4 $5 $6
    echo "Done."
    ;;
  3)
    echo "Experiment 3!"
    exp3 $2 $3 $4 $5 $6
    echo "Done."
    ;;
  *)
    echo "Invalid experiment"
    ;;
esac
```

Listing 4: analysisHW1.sh

```
#!/bin/bash

if [[ $# -ne 3 ]]; then
  echo "Invalid parameters."
  echo "Syntax: $0 [baseDataFilename] [resultsDirectory] [resultsFile]"
  exit 1
fi

base=$1
dir=$2
out=$3

if [ ! -d $dir ]; then
  echo "Directory $dir not found"
  exit 1
fi

rm -f ${dir}/${out}
echo "Enter experiment name:"
read name
echo "Experiment $name" >> ${dir}/${out}

for f in ${dir}/${base}.*; do
  sort -n $f | awk -f analyzeResults.awk scale=${f##*.} >> ${dir}/${out}
done
```

Listing 5: analyzeResults.awk

```
#!/usr/bin/awk
# assumes variable `scale' set from shell script
# assumes input is pre-sorted by $1

BEGIN {
  i = 0
  sum = 0
  sumsq = 0
}

{
  usec = $1 * 1000000
  a[i++] = usec
  sum += usec
  sumsq += usec^2
}

END {
  if (NR > 0) {
    printf("%d %d %.3f %.3f %.3f\n",
           NR, scale, sum/NR, a[NR/2], sqrt(sumsq/NR - (sum/NR)^2))
  }
}
```