Operating systems – Assignment 2
Scheduling

Lennart Jern
CS: ens16ljn

Teacher
Ahmed Aley

December 14, 2016

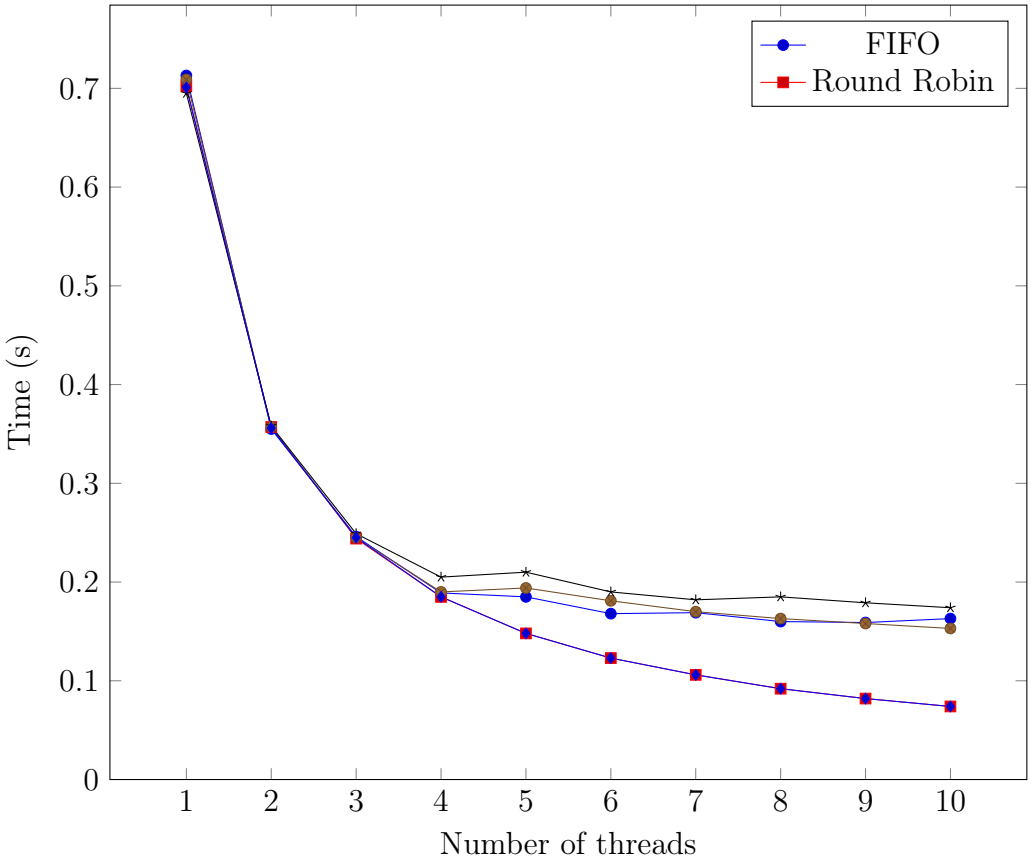


Figure 1: Threads.

1 Introduction

The Linux kernel provides a number of different scheduling policies that can be used to fine tune the performance of certain applications. In this report, five different schedulers are evaluated using an artificial, CPU intensive, work load. Three of the tested schedulers are "normal", while the last two are "real-time" schedulers, meaning that they provide higher priority for their processes than the normal ones do.

The work load consists of a simple program, called `work`, that sums over a part of Grandi's series[1] $(1 - 1 + 1 - 1 + \dots)$, using a specified number of threads. Since the task is easy to parallelize, only require minimal memory access and no disk access, it should be comparable to CPU intense tasks like compression and matrix calculations.

[1] `https://en.wikipedia.org/wiki/Grandi's_series`

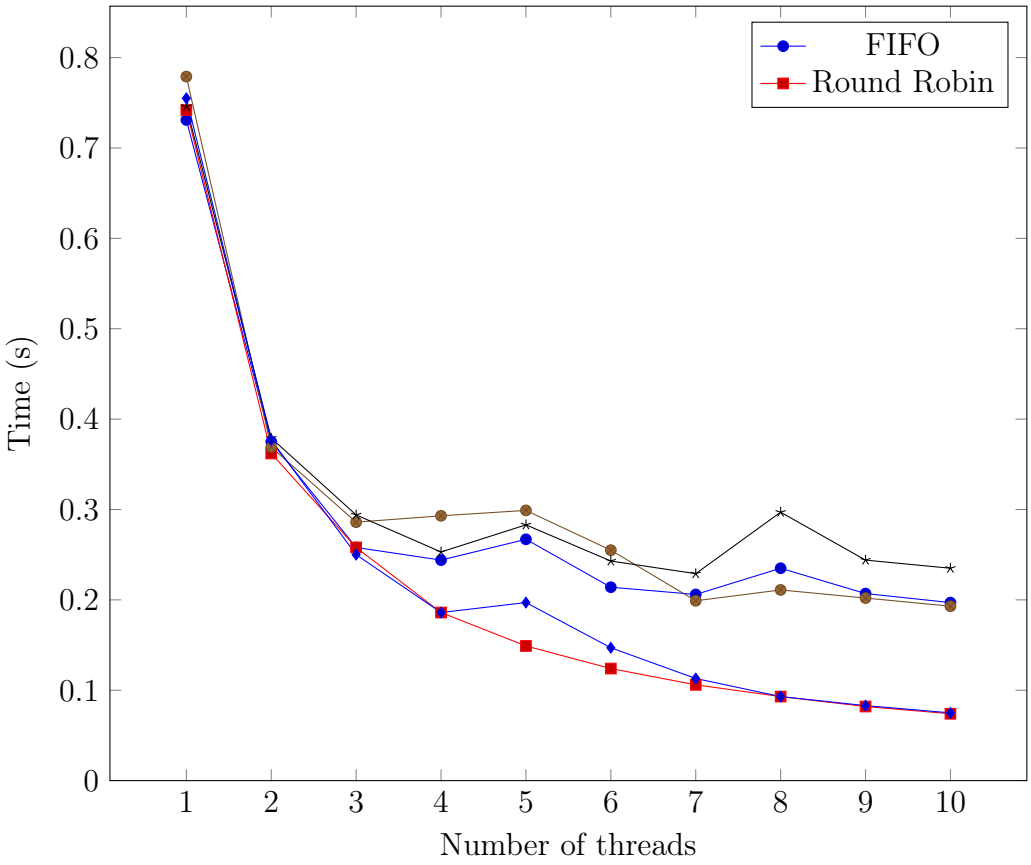


Figure 2: Threads max.

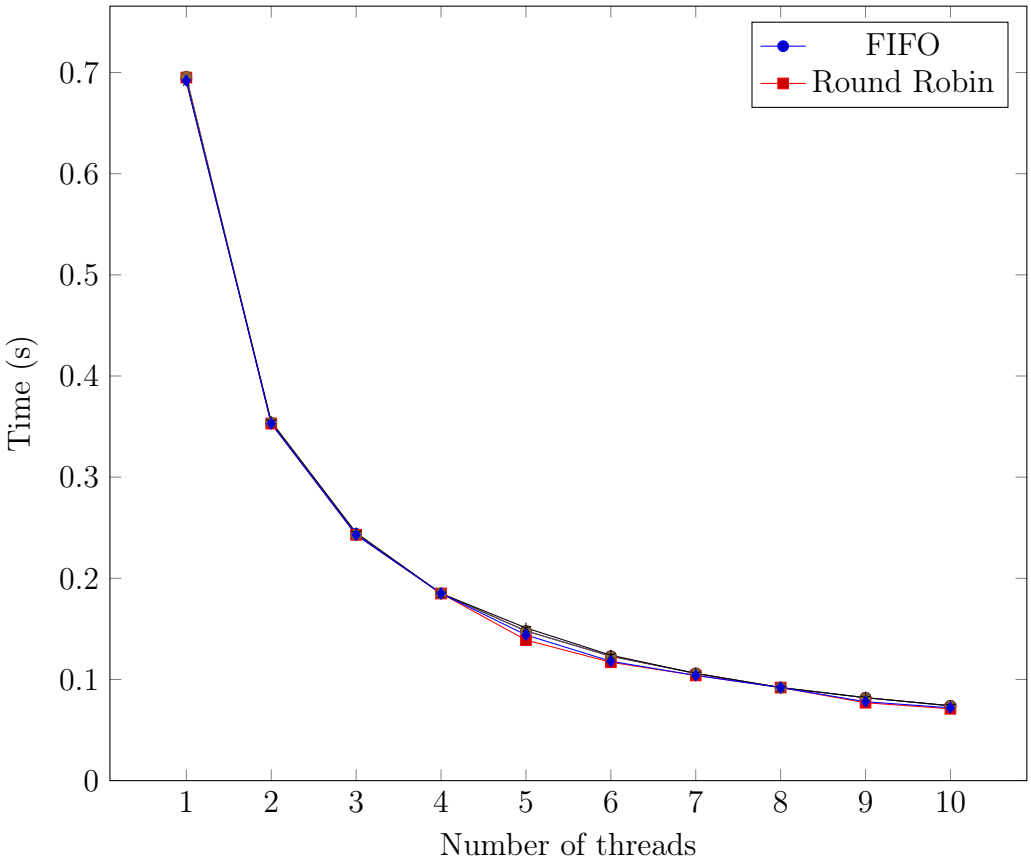


Figure 3: Threads min.

2 Implementation (Method?)

A Bash script (`timer.sh`) was used to collect data by timing the work load 10 times for each scheduler, for thread counts ranging from 1 to 10. See code listing 2 for the code. The data was then processed by a simple Python program in order to calculate the median, minimum and maximum run time for each scheduler and thread count. It should be noted here that the real-time schedulers were run with maximum priority. The other schedulers does not accept any priority settings.

All tests were run on my personal computer with the specifications seen in table 1.

Component	Specification
OS:	Fedora 25
Kernel:	Linux 4.8.12-300.fc25.x86_64
CPU:	Intel Core i5-2500K CPU @ 3.7GHz
RAM:	7965MiB

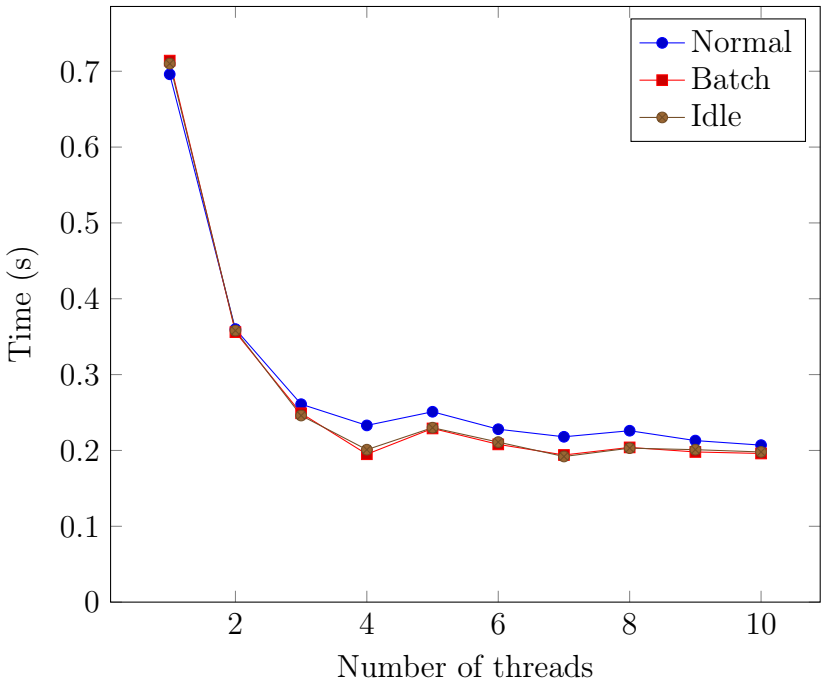
Table 1: Test system specification

3 Results

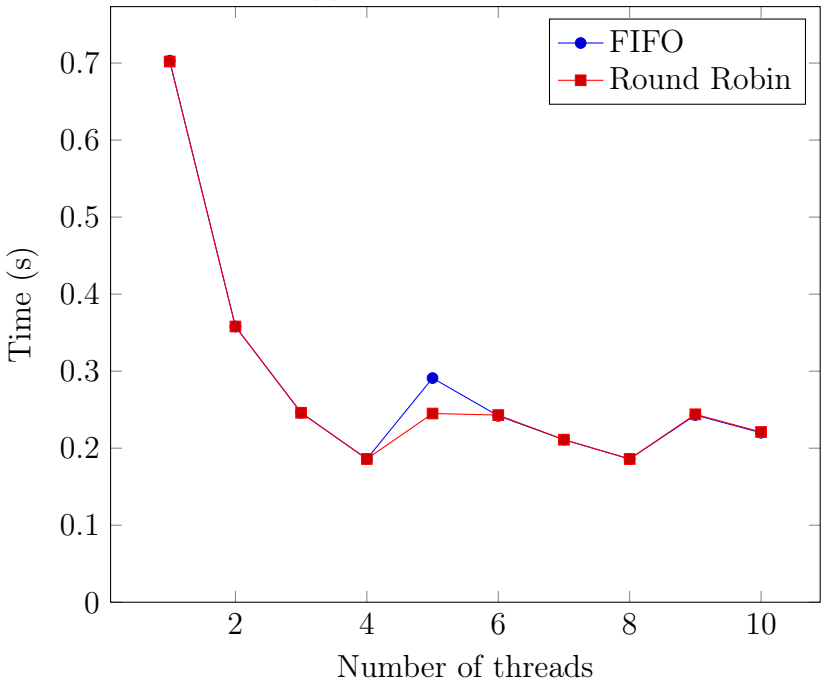
An immediate inspection of the timing data does not reveal any significant differences between the schedulers, not even between the normal and real-time ones. The median run times can be seen in figure 4a for the normal schedulers and in figure 4b for the real-time schedulers. Similarly, the maximum and minimum run times for the normal, batch and idle schedulers can be seen in figures **??** and **??** respectively, while the real-time equivalents appears in figures **??** and **??**.

It is more interesting to compare the range of response times between the schedulers (fig. 5). This reveals a clear difference between the real-time and normal schedulers, where the real-time ones are clearly more predictable for two or more threads.

The raw data collected can be found in appendix B.

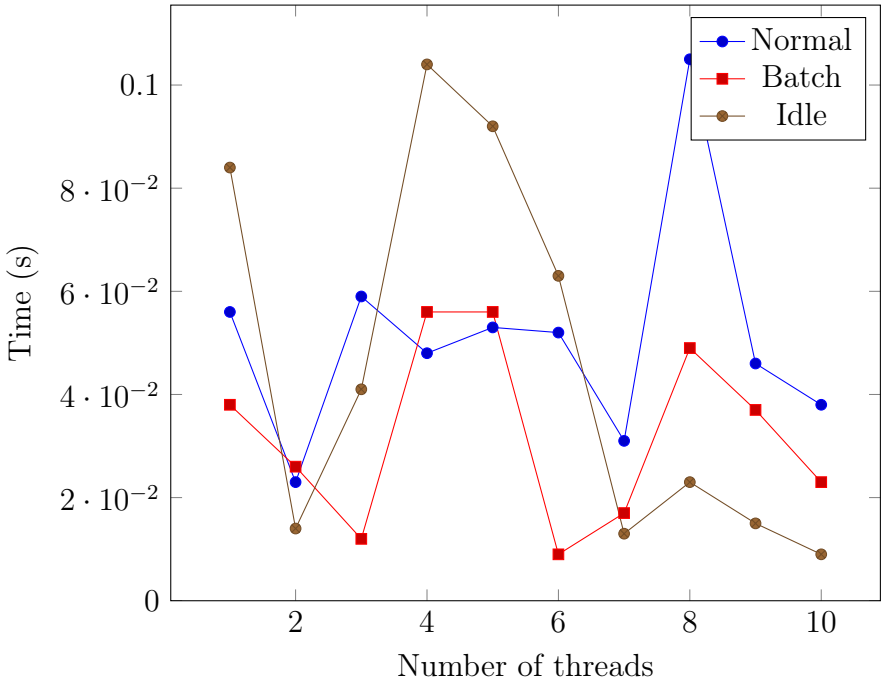


(a) Median time

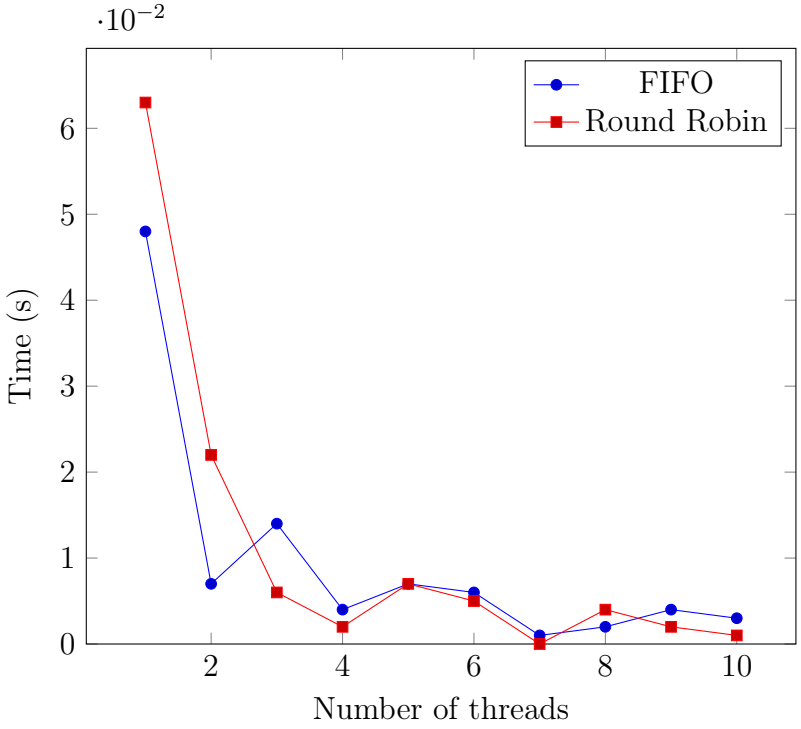


(b) Median time for real time schedulers.

Figure 4: The median time required to finish the task.



(a) Spread



(b) Spread for real time schedulers.

Figure 5: Range

4 Final thoughts and lessons learned

A Code listings

Listing 1: work.c

```
/**
 * work.c
 *
 * Just a silly "do something that takes time" program.
 * It tries to calculate the sum of Grandi's series (1-1+1-1+1-1...).
 * As long as the length it is summing over is even the sum should always be 0.
 *
 * Author: Lennart Jern (ens16ljn)
 *
 * Call sequence: work [-p <policy>] [-j <number of jobs>]
 * The policy is given by a single char according to this:
 * n - Normal
 * b - Batch
 * i - Idle
 * f - FIFO
 * r - RR
 * d - Deadline
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>           // timing
#include <errno.h>
#include <pthread.h>        // threading
#include <sys/types.h>      // pid
#include <unistd.h>         // pid, getopt
#include <linux/sched.h>    // schduling policies

// Length of sequence to sum
//#define LENGTH 2147483400
#define LENGTH 214748340
//#define LENGTH 2048
#define ONE_OVER_BILLION 1E-9

typedef struct work_load {
    int nworkers;
    long data_length;
    char scheduler;
} WorkLoad;

typedef struct work_packet {
    long index;
    long length;
    long result;
} Packet;

WorkLoad *get_work_load();
void *work(void *data);
int get_grandi(int index);
long calculate_sum(long index, long length);
void run_workers(WorkLoad *wl);
void print_schduler();
void set_scheduler(WorkLoad *wl);
void set_settings(WorkLoad *wl, int argc, char *argv[]);

int num_policies = 6;
char c_policies[] = {'n', 'b', 'i', 'f', 'r', 'd'};
```

```
58  char *str_policies[] = {"Normal", "Batch", "Idle", "FIFO", "RR", "Deadline"};
59  int policies[] = {SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE, SCHED_FIFO, SCHED_RR,
        SCHED_DEADLINE};
60
61  int main(int argc, char *argv[]) {
62
63      WorkLoad *wl = get_work_load();
64
65      set_settings(wl, argc, argv);
66
67      // Set scheduler
68      set_scheduler(wl);
69
70      // Print scheduler to make sure it is set coorectly
71      print_schduler();
72
73      run_workers(wl);
74
75      free(wl);
76      printf("Done\n");
77  }
78
79  /**
80   * get_work_load - initialize the work load and return a pointer to it
81   * @return  pointer to allocated memory
82   */
83  WorkLoad *get_work_load() {
84      WorkLoad *wl;
85      // Allocate memory for work load
86      wl = malloc(sizeof(WorkLoad));
87      // Initialize work load
88      wl->nworkers = 1;
89      // wl->data_length = 1073741824; // 2^30
90      wl->data_length = LENGTH;
91      return wl;
92  }
93
94  /**
95  * work - a silly attempt to calculate the limit of Grandi's series
96  * @param packet   the part of the work load to work on
97  * @return         nothing
98  */
99  void *work(void *packet) {
100     Packet *pkt = (Packet *)packet;
101     long sum = 0;
102
103     // Calculate time taken by a request
104     struct timespec requestStart, requestEnd;
105     clock_gettime(CLOCK_REALTIME, &requestStart);
106
107     sum = sum + calculate_sum(pkt->index, pkt->length);
108
109     clock_gettime(CLOCK_REALTIME, &requestEnd);
110
111     // Calculate time it took
112     double accum = ( requestEnd.tv_sec - requestStart.tv_sec )
113       + ( requestEnd.tv_nsec - requestStart.tv_nsec )
114       * ONE_OVER_BILLION;
115     printf( "%lf\n", accum );
116 }
117
118 /**
```

```
 * get_grandi - calculate the i:th number of Grandi's series
 * @param  index    index of the number you want to know
 * @return          1 if index is even, -1 otherwise
 */
int get_grandi(int index) {
    if (index % 2 == 0) {
        return 1;
    } else {
        return -1;
    }
}

/**
 * calculate_sum - sum Grandi's series from index over a given length
 * @param  index    index to start from
 * @param  length   how many numbers to sum over
 * @return          the sum
 */
long calculate_sum(long index, long length) {
    long sum = 0;

    for (int i = index; i < index+length; i++) {
        sum = sum + get_grandi(i);
    }
    return sum;
}

/**
* run_workers - start work threads and wait for them to finish
*/
void run_workers(WorkLoad *wl) {
    int num = wl->nworkers;
    long total_sum = 0;

    Packet *pkt[num];
    for (int i = 0; i < num; i++) {
        pkt[i] = malloc(sizeof(Packet));
        if (!pkt[i]) {
            perror("malloc");
        }
        pkt[i]->result = 0;
    }

    // create threads
    pthread_t threads[num];
    long len = wl->data_length;
    long p_len = len / num;
    int i;
    for (i = 0; i < num-1; i++) {
        pkt[i]->index = i * len / num;
        pkt[i]->length = p_len;
        if (pthread_create(&threads[i], NULL, work, (void *)pkt[i]) != 0) {
            perror("Could not create thread");
        }
    }
    pkt[i]->index = i * len / num;
    pkt[i]->length = p_len;
    work((void *)pkt[i]);

    total_sum += pkt[i]->result;
    free(pkt[i]);

```

```
181     // Join the threads
182     for (int i = 0; i < num-1; i++) {
183         pthread_join(threads[i], NULL);
184         total_sum += pkt[i]->result;
185         free(pkt[i]);
186     }
187 
188     printf("Sum is %d\n", total_sum);
189 }
190 
191 /**
192 * print_schduler - print the current scheduler
193 * @param  pid  the pid of the process
194 */
195 void print_schduler() {
196     pid_t pid = getpid();
197     int schedlr = sched_getscheduler(pid);
198 
199     char *schedlr_name;
200     switch (schedlr) {
201         case SCHED_NORMAL:
202         schedlr_name = "Normal/Other";
203         break;
204         case SCHED_BATCH:
205         schedlr_name = "Batch";
206         break;
207         case SCHED_IDLE:
208         schedlr_name = "Idle";
209         break;
210         case SCHED_FIFO:
211         schedlr_name = "FIFO";
212         break;
213         case SCHED_RR:
214         schedlr_name = "RR";
215         break;
216         case SCHED_DEADLINE:
217         schedlr_name = "Deadline";
218         break;
219         default:
220         schedlr_name = "Unknown";
221     }
222     printf("Scheduler: %s\n", schedlr_name);
223 }
224 
225 /**
226  * set_scheduler - update scheduler to reflect the given WorkLoad
227  * @param wl    the work load
228  */
229 void set_scheduler(WorkLoad *wl) {
230     struct sched_param param;
231     pid_t pid = getpid();
232     int policy = SCHED_NORMAL;
233 
234     for (int i = 0; i < num_policies; i++) {
235         if (wl->scheduler == c_policies[i]) {
236             policy = policies[i];
237             break;
238         }
239     }
240 
241     // Set the priority
242     param.sched_priority = sched_get_priority_max(policy);
```

```
243
244     if (sched_setscheduler(pid, policy, &param) != 0) {
245         perror("Set␣scheduler");
246     }
247 }
248
249 /**
250  * set_settings - parse arguments and set the settings for the work load
251  * @param wl    the work load to update
252  * @param argc  argument cound
253  * @param argv  array of arguments
254  */
255 void set_settings(WorkLoad *wl, int argc, char *argv[]) {
256     // Two possible options: j(obs) and p(olicy)
257     char *optstr = "j:p:";
258     int opt;
259     char policy = 'n';
260     int num_threads = 1;
261     int policy_ok = 0;
262     int threads_ok = 0;
263
264     // Parse flags
265     while ((opt = getopt(argc, argv, optstr)) != -1) {
266         char *end;
267         switch (opt) {
268             case 'p':
269             policy = *optarg;
270             break;
271             case 'j':
272             errno = 0;
273             num_threads = strtol(optarg, &end, 10);
274             if (errno != 0) {
275                 perror("strtol");
276             }
277             break;
278             default:
279             printf("Option␣%c␣not␣supported\n", opt);
280         }
281     }
282
283     // Check the parsed options
284     for (int i = 0; i < num_policies; i++) {
285         if (policy == c_policies[i]) {
286             policy_ok = 1;
287             break;
288         }
289     }
290
291     if (num_threads <= 100 && num_threads > 0) {
292         threads_ok = 1;
293     }
294
295     // Set values if they are safe, or set defaults
296     if (policy_ok) {
297         wl->scheduler = policy;
298     } else {
299         wl->scheduler = 'n';
300     }
301
302     if (threads_ok) {
303         wl->nworkers = num_threads;
304     } else {
```

```
305         wl->nworkers = 1;
306     }
307 }
```

Listing 2: timer.sh

```
1  #!/bin/bash
2
3  # A timer script to measure the differences between schedulers/policies
4  #
5  # Author: Lennart Jern (ens16ljn@cs.umu.se)
6
7  for THREADS in $(seq 1 10)
8  do
9      DATA="Normal,Batch,Idle,FIFO,Round␣Robin"
10     echo "Running␣with␣$THREADS␣threads"
11     # Time the commands 10 times
12     for i in $(seq 1 10)
13     do
14         LINE=""
15         # For the polices n(ormal) b(atch) and i(dle)
16         for POLICY in n b i f r
17         do
18             # Set policy and number of threads
19             FLAGS="-p$POLICY␣-j$THREADS"
20             COMMAND="./work␣$FLAGS␣>>␣../data/threads$THREADS$POLICY.log"
21             # Run the command and store the time
22             t="$(sh␣-c␣"TIMEFORMAT='%5R'; time $COMMAND"␣2>&1)"
23             # Build the line
24             if [ "$POLICY" = "n" ]; then
25                 LINE="$t"
26             else
27                 LINE="$LINE,$t"
28             fi
29         done
30         DATA=$DATA$'\n'$LINE
31         # A little progress report
32         echo "Run␣$i␣done."
33     done
34
35     # Write data to a file
36     echo "$DATA" > "../data/data$THREADS.csv"
37     chown lennart ../data/threads*
38
39 done
```

Listing 3: stats.py

```
1  """
2  stats.py
3
4  Process the data produced by timer.sh by calculating the
5  medians, max values and min values for each scheduler and thread count
6
7  Author: Lennart Jern (ens16ljn@cs.umu.se)
8  """
9
10 import pandas as pd
11 import re
12 import matplotlib.pyplot as plt
```

```
13
14 def total_stats():
15     # The data file names are of the form data<thread count>.csv
16     base = "../data/data"
17     thread_base = "../data/threads"
18     ext = ".csv"
19     header = ("Normal", "Batch", "Idle", "FIFO", "Round␣Robin")
20     # Data frames to store the results in
21     med = pd.DataFrame(columns=header)          # Medians (total runtime)
22     mx = pd.DataFrame(columns=header)           # Max (total runtime)
23     mn = pd.DataFrame(columns=header)           # Min (total runtime)
24     thread_med = pd.DataFrame(columns=header)   # Medians (threads)
25     thread_mx = pd.DataFrame(columns=header)    # Max (threads)
26     thread_mn = pd.DataFrame(columns=header)    # Min (threads)
27
28     # For each number of threads
29     for i in range(1,11):
30         # Build the file name
31         f = base + str(i) + ext             # Total run times
32         thr_f = thread_base + str(i) + ext  # Thread times
33         # Read the time data
34         df = pd.read_csv(f)
35         thr_df = pd.read_csv(thr_f)
36
37         # Calculate some statistical properties
38         med.loc[i] = df.median()
39         mx.loc[i] = df.max()
40         mn.loc[i] = df.min()
41         thread_med.loc[i] = thr_df.median()
42         thread_mx.loc[i] = thr_df.max()
43         thread_mn.loc[i] = thr_df.min()
44
45     # Calculate ranges
46     rng = mx-mn
47     thr_rng = mx-mn
48
49     # Write everything to files
50     data_frames = [med, mx, mn, thread_med, thread_mx, thread_mn, rng, thr_rng]
51     names = ["medians", "max", "min", "thread_medians", "thread_max", "thread_min", "
           range", "thread_range"]
52     for frm, name in zip(data_frames, names):
53         frm.to_csv(name + ext, index_label="Threads", float_format="%.5f")
54
55     # Plot medians
56     ax = thr_df.plot.box()
57     fig = ax.get_figure()
58     fig.savefig('box.pdf')
59
60
61
62 def collect_thread_times(file_name):
63     """Read thread times from a file."""
64     f = open(file_name)
65     times = []
66     # Regular expression to find floats
67     time = re.compile("(\d+\.\d+)")
68
69     for line in f:
70         match = time.match(line)
71
72         if (match):
73             t = float(match.group(1))
```

```
            times.append(t)

    return times


def thread_stats():
    """Collect timing information about all threads and store in csv files"""
    threads = [i for i in range(1, 11)]
    schedulers = ["n", "b", "i", "f", "r"]
    base = "../data/threads"
    ext = ".log"
    header=("Normal", "Batch", "Idle", "FIFO", "Round␣Robin")
    # Files to write: thread{}.csv
    # csv_files = [base+str(i)+".csv" for i in threads]

    # Collect all times for one thread count in one file
    for t in threads:
        times = {key: [] for key in schedulers}
        for s in schedulers:
            f = get_file_name(t, s)
            times[s] = collect_thread_times(f)
        # Write to file
        df = pd.DataFrame(times)
        df.to_csv(get_csv_name(t), index=False, header=header)




def get_file_name(threads, scheduler):
    """Get the file name for the data regarding <scheduler> and <threads>"""
    base = "../data/threads"
    ext = ".log"
    return base + str(threads) + scheduler + ext

def get_csv_name(threads):
    """Get name of file to write data about thread count <threads> to"""
    base = "../data/threads"
    ext = ".csv"
    return base + str(threads) + ext

thread_stats()
total_stats()
```

Listing 4: Makefile

```
all: work.c
        gcc work.c -pthread -o work
```

B Raw data

Threads	Normal	Batch	Idle	FIFO	Round Robin
1	0.696	0.714	0.710	0.703	0.702
2	0.360	0.356	0.358	0.358	0.358
3	0.261	0.249	0.246	0.246	0.246
4	0.233	0.195	0.201	0.186	0.186
5	0.251	0.229	0.230	0.291	0.245
6	0.228	0.208	0.211	0.242	0.243
7	0.218	0.194	0.192	0.211	0.211
8	0.226	0.204	0.203	0.186	0.186
9	0.213	0.198	0.201	0.243	0.244
10	0.207	0.196	0.198	0.220	0.221

Threads	Normal	Batch	Idle	FIFO	Round Robin
1	0.747	0.732	0.780	0.744	0.756
2	0.379	0.380	0.370	0.363	0.378
3	0.309	0.259	0.286	0.258	0.251
4	0.254	0.246	0.294	0.190	0.188
5	0.284	0.272	0.313	0.295	0.251
6	0.260	0.215	0.256	0.248	0.247
7	0.234	0.207	0.202	0.212	0.211
8	0.307	0.241	0.213	0.188	0.190
9	0.251	0.227	0.210	0.246	0.245
10	0.241	0.212	0.202	0.222	0.222

Threads	Normal	Batch	Idle	FIFO	Round Robin
1	0.691	0.694	0.696	0.696	0.693
2	0.356	0.354	0.356	0.356	0.356
3	0.250	0.247	0.245	0.244	0.245
4	0.206	0.190	0.190	0.186	0.186
5	0.231	0.216	0.221	0.288	0.244
6	0.208	0.206	0.193	0.242	0.242
7	0.203	0.190	0.189	0.211	0.211
8	0.202	0.192	0.190	0.186	0.186
9	0.205	0.190	0.195	0.242	0.243
10	0.203	0.189	0.193	0.219	0.221