Operating systems – Assignment 2
Scheduling

Lennart Jern
CS: ens16ljn

Teacher
Ahmed Aley

December 12, 2016

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.

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 3 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

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

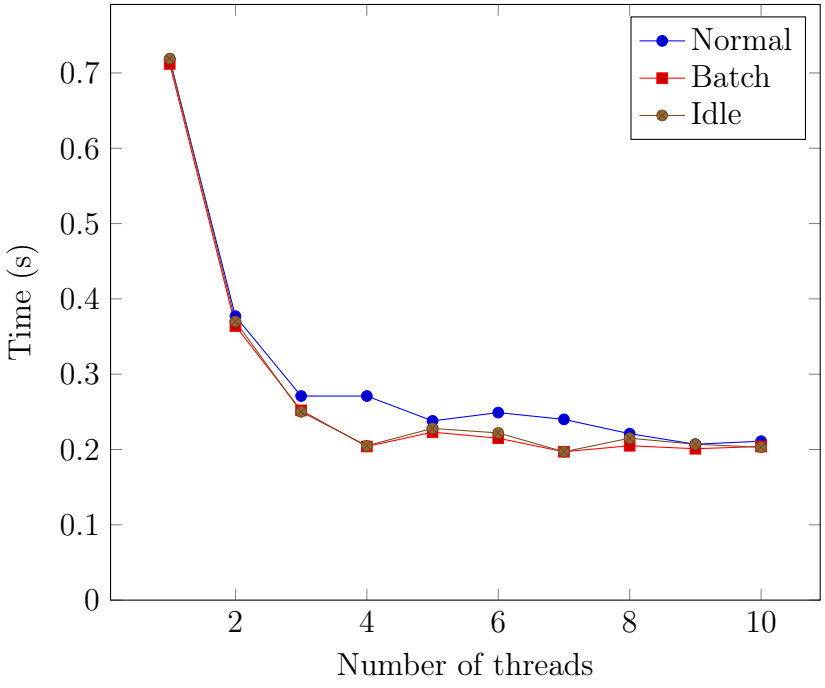
3 Results

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 1a for the normal schedulers and in figure 1b for the real-time schedulers. Similarly, the maximum and minimum run times for the normal, batch and idle schedulers can be seen in figures 2a and 3a respectively, while the real-time equivalents appears in figures 2b and 3b.

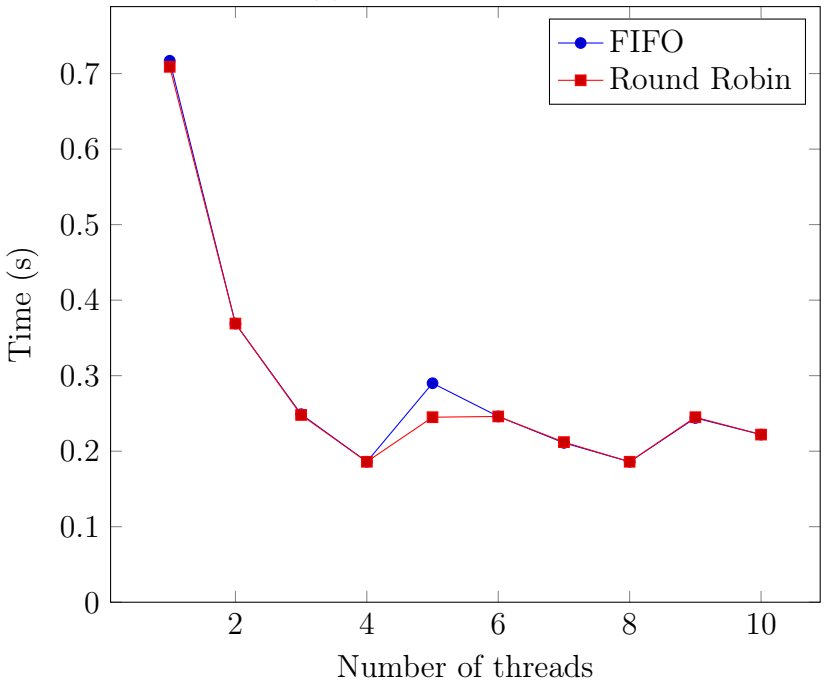
It is more interesting to compare the range of response times between the schedulers (fig. 4). This reveals a clear difference between the real-time and normal schedulers, where the real-time ones are clearly more predictable for two or more threads.

The raw data collected can be found in appendix B.

4 Final thoughts and lessons learned

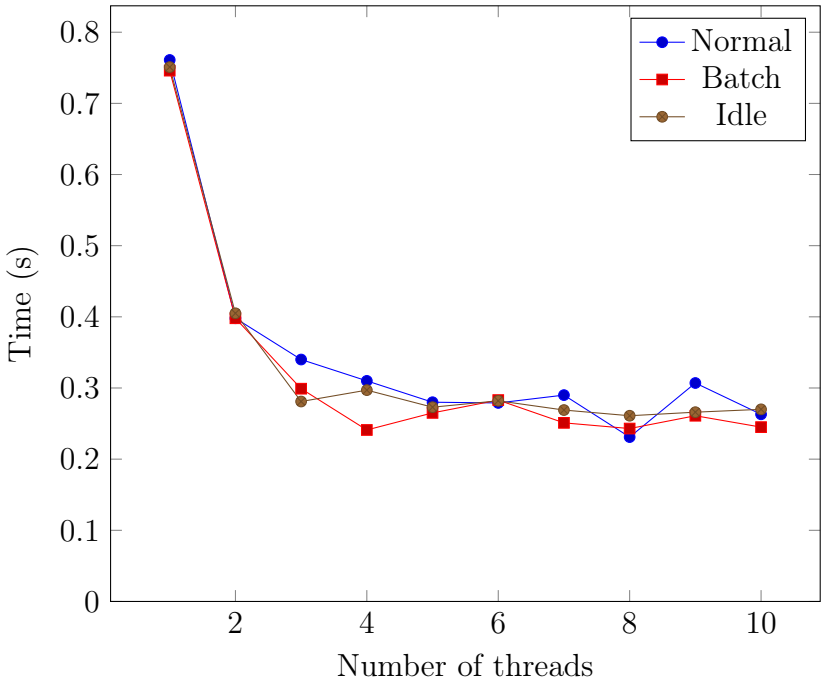


(a) Median time

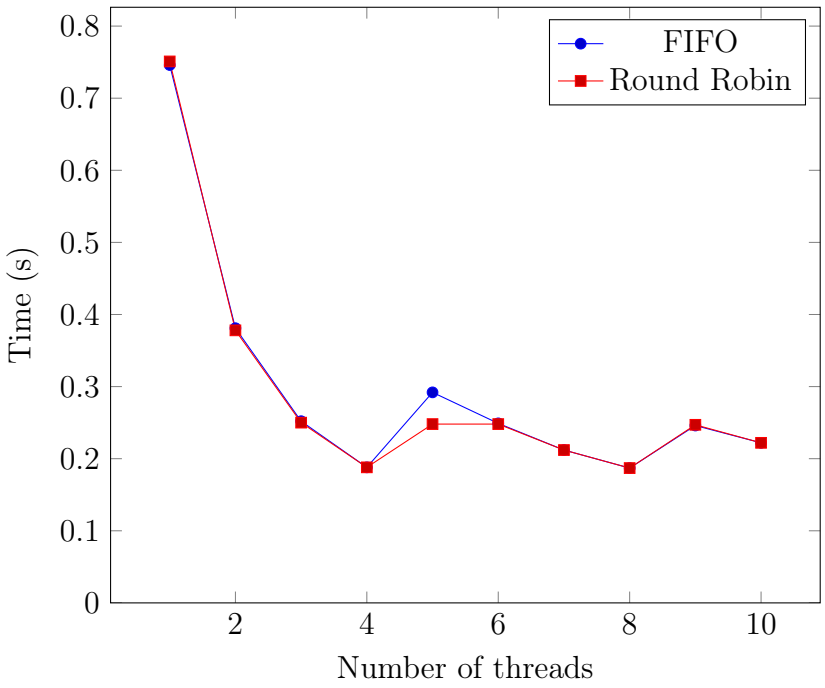


(b) Median time for real time schedulers.

Figure 1: The median time required to finish the task.

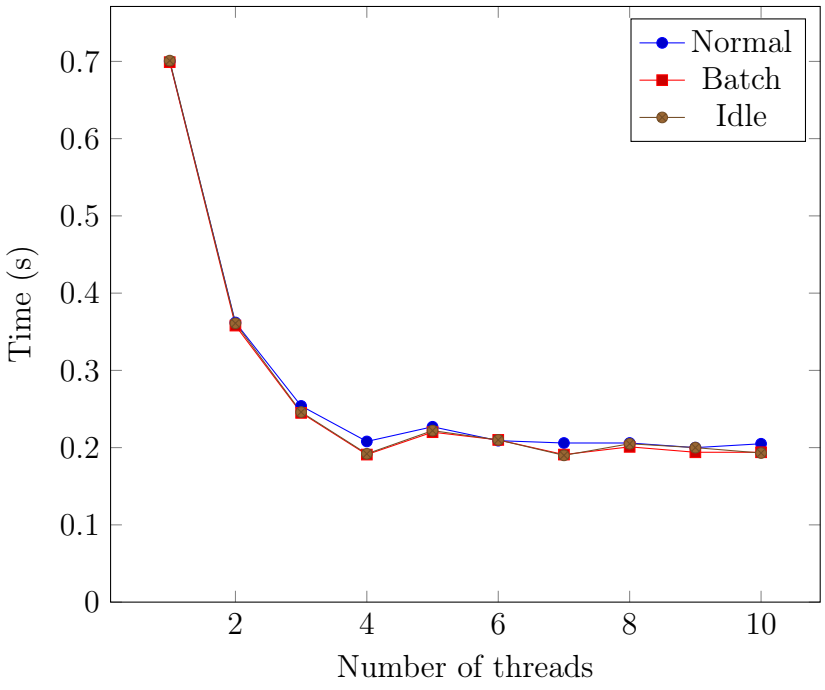


(a) Maximum time

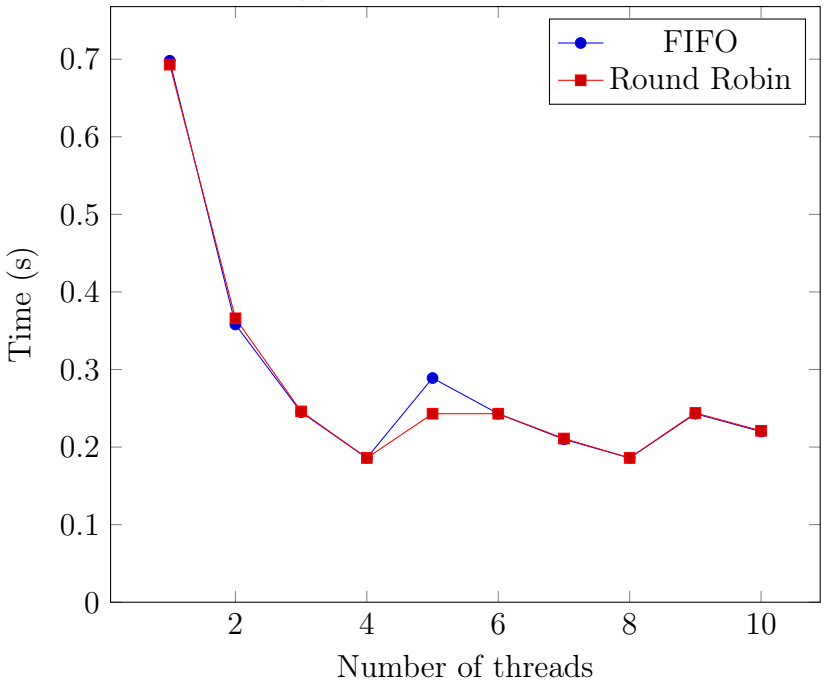


(b) Maximum time for real time schedulers.

Figure 2: The maximum run time required to complete the task.

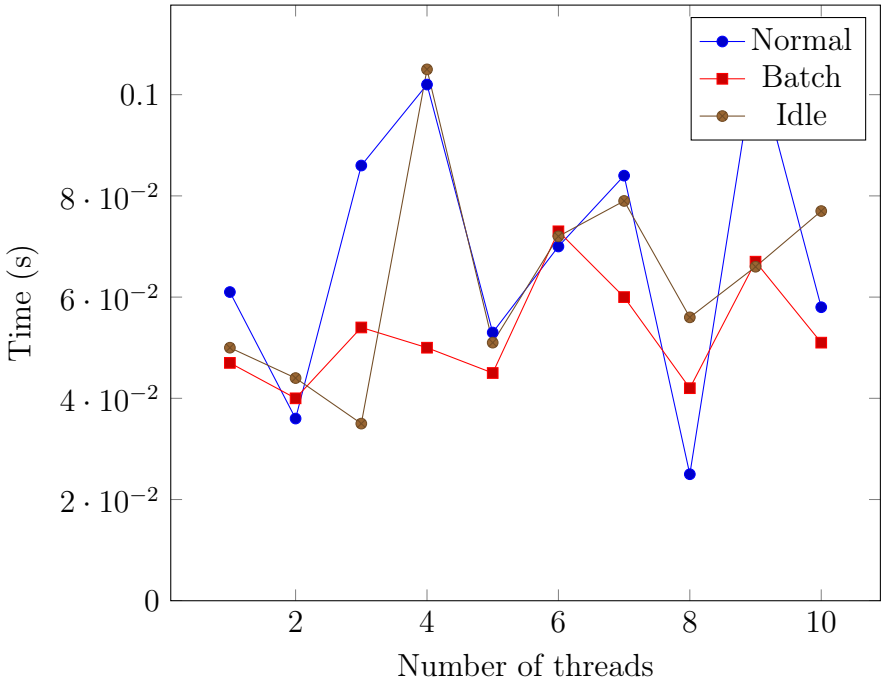


(a) Minimum time

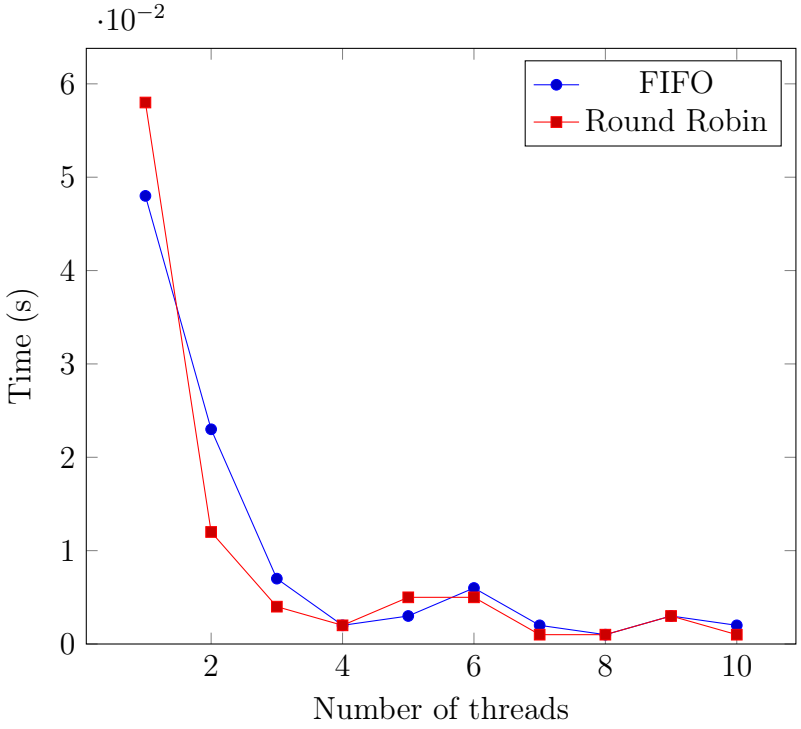


(b) Minimum time for real time schedulers.

Figure 3: The minimum run time required to complete the task.



(a) Spread



(b) Spread for real time schedulers.

Figure 4: Range

A Code listings

Listing 1: work.c

```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 <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

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'};
char *str_policies[] = {"Normal", "Batch", "Idle", "FIFO", "RR", "Deadline"};
int policies[] = {SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE, SCHED_FIFO, SCHED_RR,
    SCHED_DEADLINE};
```

```
57
58  int main(int argc, char *argv[]) {
59
60      WorkLoad *wl = get_work_load();
61
62      set_settings(wl, argc, argv);
63
64      // Set scheduler
65      set_scheduler(wl);
66
67      // Print scheduler to make sure it is set coorectly
68      print_schduler();
69
70      printf("Starting␣workers...\n");
71      run_workers(wl);
72
73      free(wl);
74      printf("Done\n");
75  }
76
77  /**
78   * get_work_load - initialize the work load and return a pointer to it
79   * @return  pointer to allocated memory
80   */
81  WorkLoad *get_work_load() {
82      WorkLoad *wl;
83      // Allocate memory for work load
84      wl = malloc(sizeof(WorkLoad));
85      // Initialize work load
86      wl->nworkers = 1;
87      // wl->data_length = 1073741824; // 2^30
88      wl->data_length = LENGTH;
89      return wl;
90  }
91
92  /**
93  * work - a silly attempt to calculate the limit of Grandi's series
94  * @param packet   the part of the work load to work on
95  * @return         nothing
96  */
97  void *work(void *packet) {
98      Packet *pkt = (Packet *)packet;
99
100     long sum = 0;
101
102     sum = sum + calculate_sum(pkt->index, pkt->length);
103 }
104
105 /**
106  * get_grandi - calculate the i:th number of Grandi's series
107  * @param  index    index of the number you want to know
108  * @return          1 if index is even, -1 otherwise
109  */
110 int get_grandi(int index) {
111     if (index % 2 == 0) {
112         return 1;
113     } else {
114         return -1;
115     }
116 }
117
118 /**
```

```
 * 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");
        }
    }
    printf("Working...\n");
    pkt[i]->index = i * len / num;
    pkt[i]->length = p_len;
    work((void *)pkt[i]);

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

    // Join the threads
    for (int i = 0; i < num-1; i++) {
        pthread_join(threads[i], NULL);
        total_sum += pkt[i]->result;
        free(pkt[i]);
    }

    printf("Sum is %d\n", total_sum);
}

/**
* print_schduler - print the current scheduler
```

```
 * @param  pid  the pid of the process
 */
void print_schduler() {
    pid_t pid = getpid();
    int schedlr = sched_getscheduler(pid);

    char *schedlr_name;
    switch (schedlr) {
        case SCHED_NORMAL:
        schedlr_name = "Normal/Other";
        break;
        case SCHED_BATCH:
        schedlr_name = "Batch";
        break;
        case SCHED_IDLE:
        schedlr_name = "Idle";
        break;
        case SCHED_FIFO:
        schedlr_name = "FIFO";
        break;
        case SCHED_RR:
        schedlr_name = "RR";
        break;
        case SCHED_DEADLINE:
        schedlr_name = "Deadline";
        break;
        default:
        schedlr_name = "Unknown";
    }
    printf("Scheduler:␣%s\n", schedlr_name);
}

/**
 * set_scheduler - update scheduler to reflect the given WorkLoad
 * @param wl    the work load
 */
void set_scheduler(WorkLoad *wl) {
    struct sched_param param;
    pid_t pid = getpid();
    int policy = SCHED_NORMAL;

    for (int i = 0; i < num_policies; i++) {
        if (wl->scheduler == c_policies[i]) {
            policy = policies[i];
            break;
        }
    }

    // Set the priority
    param.sched_priority = sched_get_priority_max(policy);

    if (sched_setscheduler(pid, policy, &param) != 0) {
        perror("Set␣scheduler");
    }
}

/**
 * set_settings - parse arguments and set the settings for the work load
 * @param wl    the work load to update
 * @param argc  argument cound
 * @param argv  array of arguments
 */
```

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

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)
```

```

for THREADS in $(seq 1 10)
do
    DATA="Normal,Batch,Idle,FIFO,Round␣Robin"
    echo "Running␣with␣$THREADS␣threads"
    # Time the commands 10 times
    for i in $(seq 1 10)
    do
        LINE=""
        # For the polices n(ormal) b(atch) and i(dle)
        for POLICY in n b i f r
        do
            # Set policy and number of threads
            FLAGS="-p$POLICY␣-j$THREADS"
            COMMAND="./work␣$FLAGS␣>␣/dev/null"
            # Run the command and store the time
            t="$(sh␣-c␣"TIMEFORMAT='%5R'; time $COMMAND"␣2>&1)"
            # Build the line
            if [ "$POLICY" = "n" ]; then
                LINE="$t"
            else
                LINE="$LINE,$t"
            fi
        done
        DATA=$DATA$'\n'$LINE
        # A little progress report
        echo "Run␣$i␣done."
    done

    # Write data to a file
    echo "$DATA" > "data$THREADS.csv"

done
```

Listing 3: stats.py

```
"""
stats.py

Process the data produced by timer.sh by calculating the
medians, max values and min values for each scheduler and thread count

Author: Lennart Jern (ens16ljn@cs.umu.se)
"""

import pandas as pd

# The data file names are of the form data<thread count>.csv
name = "data"
ext = ".csv"
# Data frames to store the results in
medians = pd.DataFrame(columns=("Normal", "Batch", "Idle", "FIFO", "Round␣Robin"))
mx = pd.DataFrame(columns=("Normal", "Batch", "Idle", "FIFO", "Round␣Robin"))
mn = pd.DataFrame(columns=("Normal", "Batch", "Idle", "FIFO", "Round␣Robin"))

# For each number of threads
for i in range(1,11):
    # Build the file name
    f = name + str(i) + ext
    # Read the time data
    df = pd.read_csv(f)
```

```
26
27      # Add data to results
28      medians.loc[i] = df.median()
29      mx.loc[i] = df.max()
30      mn.loc[i] = df.min()
31
32  # Write everything to files
33  medians.to_csv("medians" + ext, index_label="Threads", float_format="%.3f")
34  mx.to_csv("max" + ext, index_label="Threads", float_format="%.3f")
35  mn.to_csv("min" + ext, index_label="Threads", float_format="%.3f")
36
37  spread = mx-mn
38  spread.to_csv("spread" + ext, index_label="Threads", float_format="%.3f")
```

Listing 4: Makefile

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

B Raw data

Threads	Normal	Batch	Idle	FIFO	Round Robin
1	0.718	0.712	0.719	0.717	0.709
2	0.377	0.364	0.370	0.369	0.369
3	0.271	0.252	0.250	0.249	0.248
4	0.271	0.204	0.205	0.186	0.186
5	0.238	0.223	0.228	0.290	0.245
6	0.249	0.215	0.222	0.246	0.246
7	0.240	0.197	0.197	0.211	0.212
8	0.221	0.205	0.215	0.186	0.186
9	0.207	0.201	0.207	0.244	0.245
10	0.211	0.204	0.203	0.222	0.222

Threads	Normal	Batch	Idle	FIFO	Round Robin
1	0.761	0.746	0.751	0.746	0.751
2	0.398	0.398	0.405	0.381	0.378
3	0.340	0.299	0.281	0.252	0.250
4	0.310	0.241	0.297	0.188	0.188
5	0.280	0.265	0.273	0.292	0.248
6	0.279	0.283	0.282	0.249	0.248
7	0.290	0.251	0.269	0.212	0.212
8	0.231	0.243	0.261	0.187	0.187
9	0.307	0.261	0.266	0.246	0.247
10	0.263	0.245	0.270	0.222	0.222

Threads	Normal	Batch	Idle	FIFO	Round Robin
1	0.700	0.699	0.701	0.698	0.693
2	0.362	0.358	0.361	0.358	0.366
3	0.254	0.245	0.246	0.245	0.246
4	0.208	0.191	0.192	0.186	0.186
5	0.227	0.220	0.222	0.289	0.243
6	0.209	0.210	0.210	0.243	0.243
7	0.206	0.191	0.190	0.210	0.211
8	0.206	0.201	0.205	0.186	0.186
9	0.200	0.194	0.200	0.243	0.244
10	0.205	0.194	0.193	0.220	0.221