# Operating systems – Assignment 2
# Scheduling

Lennart Jern
CS: ens16ljn

**Teacher**
Ahmed Aley

December 17, 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, task. 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. The source code for `work` can be found in listing 1. 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 Method

A Bash script (`timer.sh`, listing 2) was used to time the complete task 10 times for each scheduler, for thread counts ranging from 1 to 10. See code listing 2 for the code. Additionally, each thread keeps track of the time taken from the start of its execution until it is finished, and prints this information to `stdout`, which is forwarded to data files by the bash script.

All this data was then processed by a simple Python program, `stats`, in order to calculate the median, minimum and maximum run time for each scheduler and thread count; for both the total run times and the individual thread times. This program can be found in listing 3. In addition to the statistical calculations, `stats` also produces some figures to describe the data. The figures and calculations are mostly done using the python libraries Pandas[2] and Matplotlib[3].

It should be noted here that the processes running with real-time schedulers were run with maximum priority. Since the other schedulers does not accept a priority setting, other than the nice value, these were left untouched.

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

[1] `https://en.wikipedia.org/wiki/Grandi's_series`
[2] `http://pandas.pydata.org/`
[3] `http://matplotlib.org/`

| Component | Specification |
|---|---|
| OS: | Fedora 25 |
| Kernel: | Linux 4.8.13-300.fc25.x86_64 |
| CPU: | Intel Core i5-2500K CPU @ 3.7GHz |
| RAM: | 7965MiB |
| GCC: | 6.2.1 |
| Bash: | 4.3.43 |
| Python: | 3.5.2 |
| Pandas: | 0.18.1 |
| Matplotlib: | 1.5.3 |

Table 1: Test system specification.

# 3 Results

For the total run time of the process, there does not seem to be much of a difference between the five schedulers (see fig. 1). This is perhaps not too surprising considering that the system was practically idle, except for the work load process, so this process naturally got all resources the system could offer.

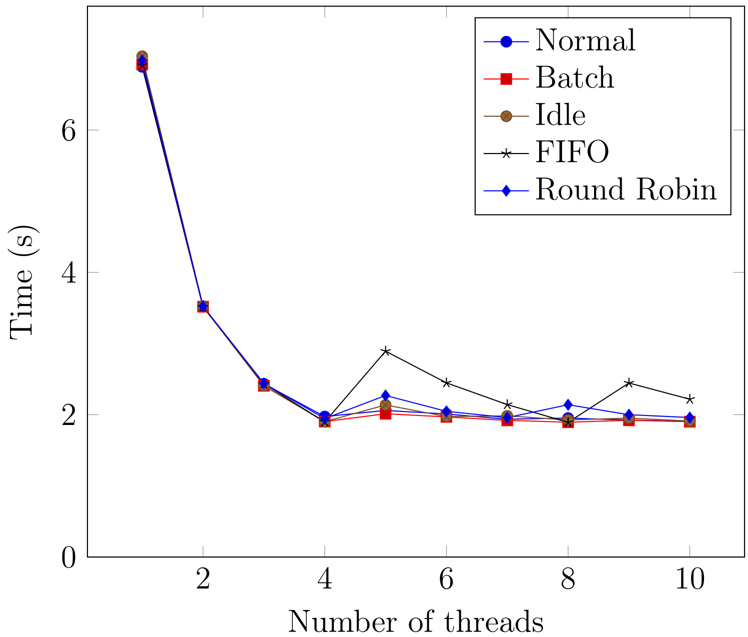
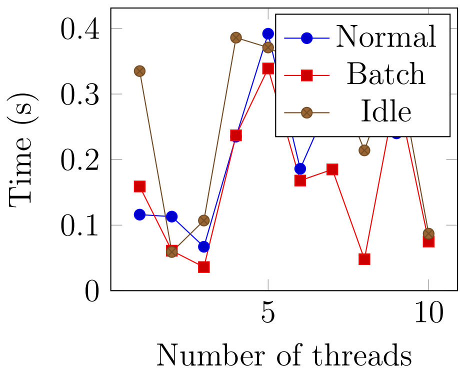
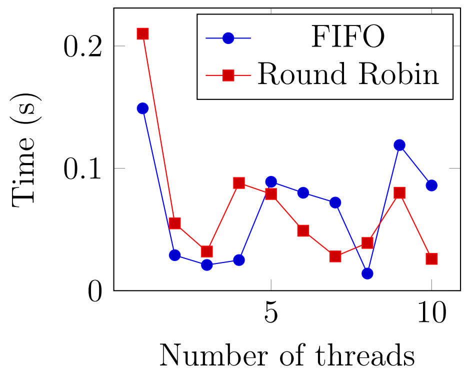



Figure 1: The median time required to finish the complete task.




(a) The range for normal schedulers




(b) The range for real time schedulers.

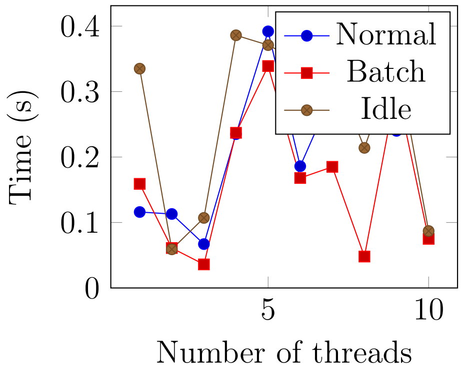
Figure 2: A comparison of response times for the complete task, between the different schedulers.

More interesting is the range (difference between maximum and minimum) of response times for the total run time of the process (fig. 2). 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.

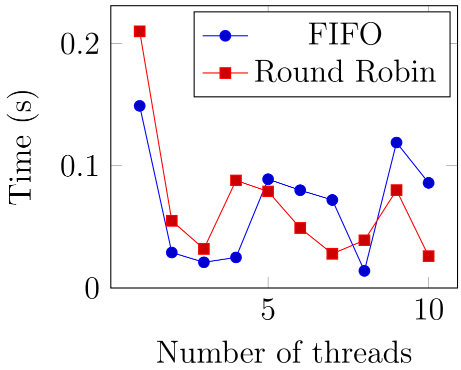
The same trend can also be seen for the thread response times in fig. 3. Furthermore, there is a small difference in median run time for the schedulers, when looking at the individual thread times, for a thread count between 6 and 8. This can be seen in fig. 4, where the Batch scheduler have a lower median time than the others. However, the difference seem to be somewhat periodic and vanishes almost completely with 9 threads. The Round Robin scheduler also follows this pattern but with a shorter period. With the exception of Batch, Round Robin beats the other schedulers for thread counts of 5, 6, 9 and 10.

The differences are even clearer when looking at the box charts for the individual thread times in figures 5 and 6 where the range between the maximum and minimum run times can also be seen.

To further analyze the behavior of the schedulers, we can take a look at the density plots for the thread times in figures 7 and 8. There are only small differences when running with two threads. But with four threads instead, we can see that the Batch scheduler have two distinct "batches" while RR is much closer to have all threads finish in one lump. I am quite surprised to see the FIFO curve so flat and spread out at four threads, since they should all be able to run together and finish simultaneously.




(a) The range in response times for normal schedulers




(b) The range in response times for real time schedulers.

Figure 3: A comparison of response times per thread between the different schedulers.

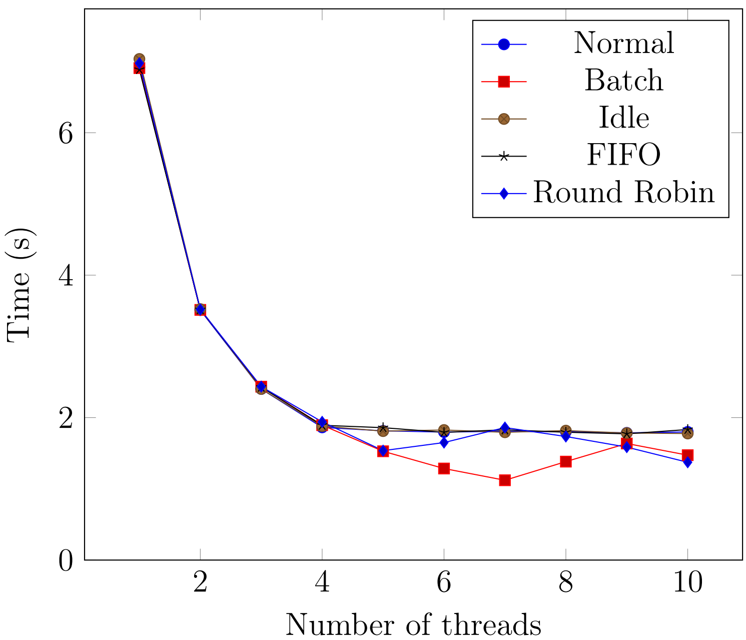
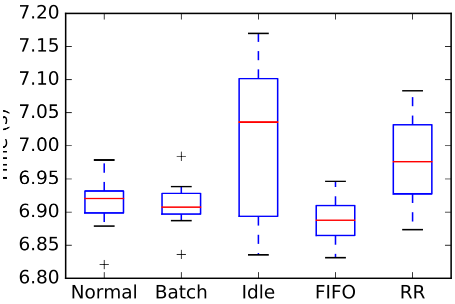
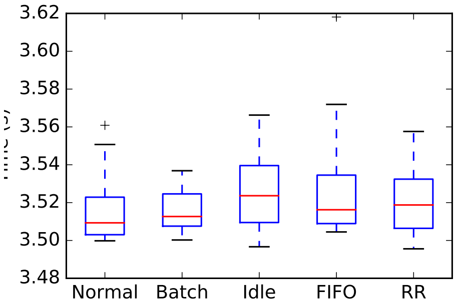


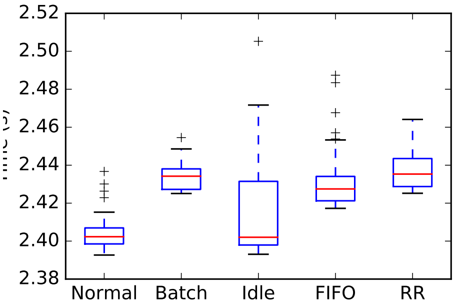

Figure 4: The median time per thread.



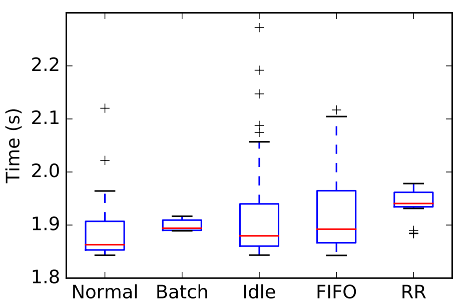

(a) Thread count: 1



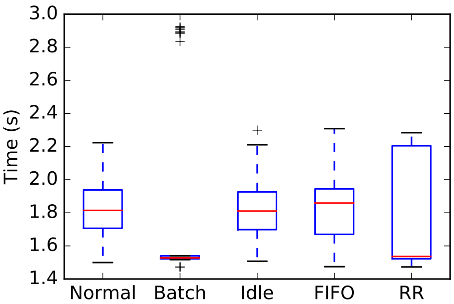

(b) Thread count: 2



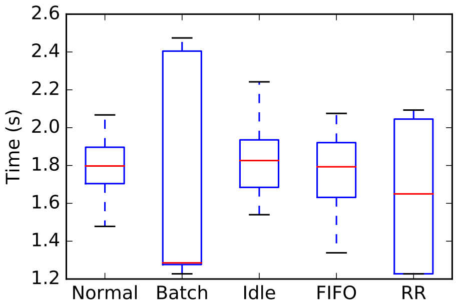

(c) Thread count: 3




(d) Thread count: 4

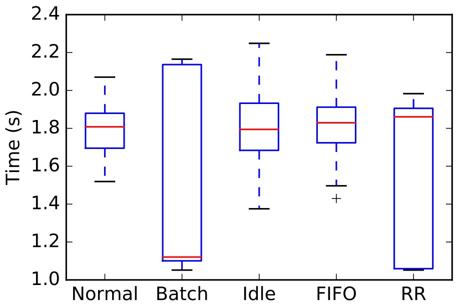



(e) Thread count: 5

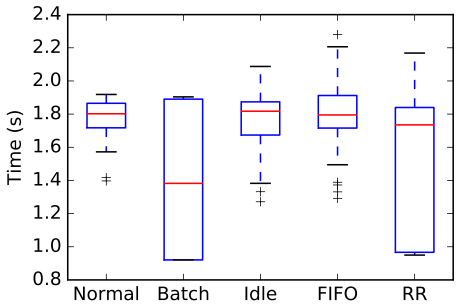



(f) Thread count: 6

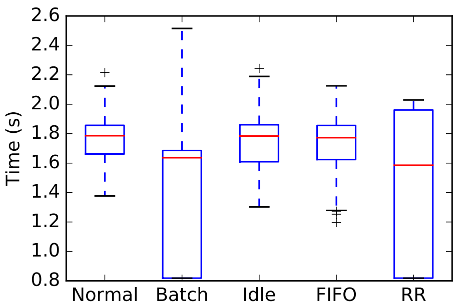
Figure 5: Box charts for thread counts 1 to 6.



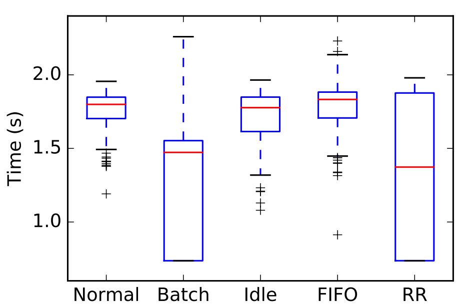

(a) Thread count: 7




(b) Thread count: 8

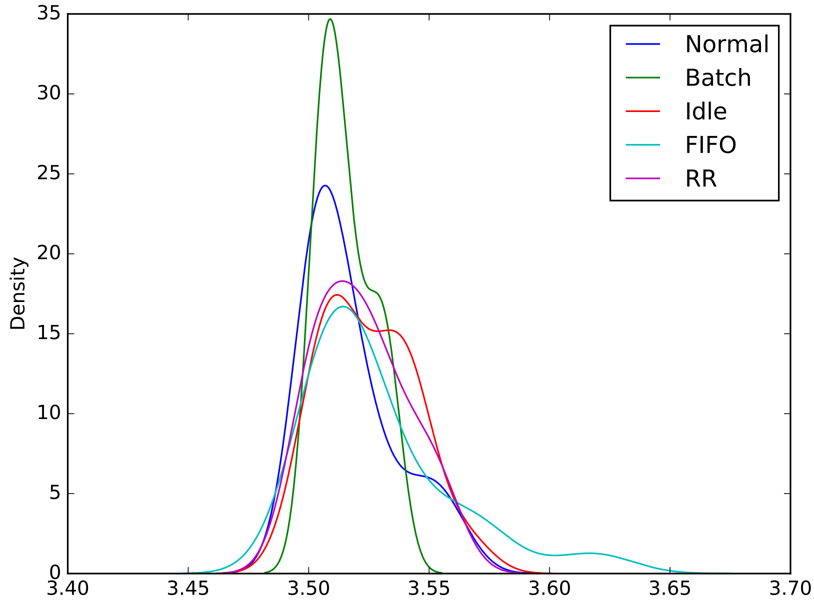



(c) Thread count: 9

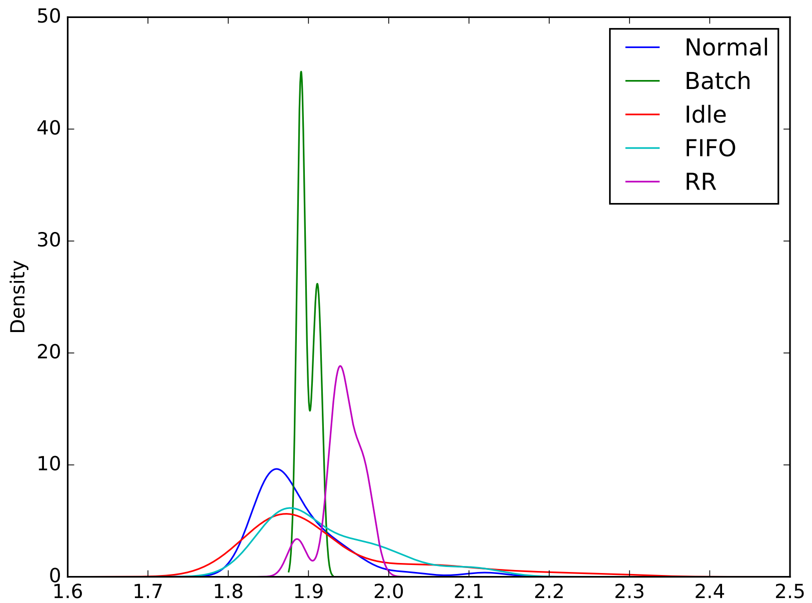



(d) Thread count: 10

Figure 6: Box charts for thread counts 7 to 10.

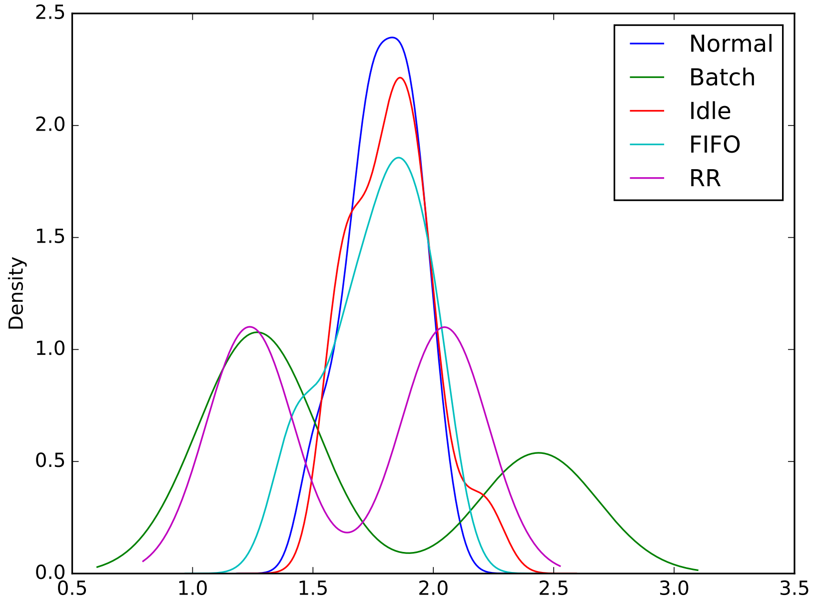



(a) Thread count: 2

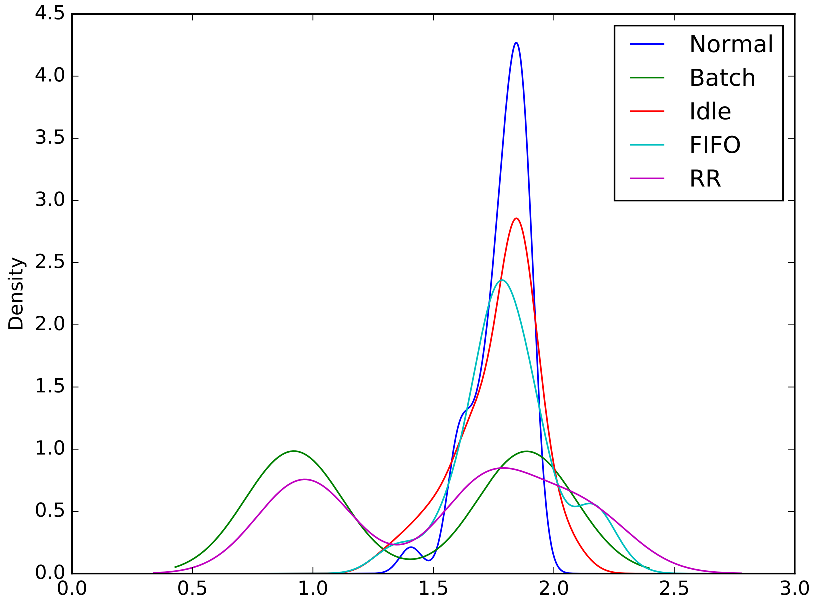



(b) Thread count: 4

Figure 7: Density plots for the different schedulers when running with 2 and 4 thread




(a) Thread count: 6




(b) Thread count: 8

Figure 8: Density plots for the different schedulers when running with 6 and 8 thread

At six and eight threads, a pattern emerges where the Normal, Idle and FIFO schedulers finishes all their threads close together, while the Batch and RR schedulers both have two groups of threads finishing together. With a higher resolution we would probably se two peaks for FIFO also, a small bump is already visible in the plots for six and eight threads.

The results for Round Robin follow the theory nicely, as it becomes more and more flat as the thread count increases (compare the figures for six and eight threads), since it tries to distribute the CPU time as evenly as possible. The Batch scheduler avoids switching between threads, and should thus get one batch of four and one of two threads finishing together when running with six threads total (since the processor has four cores). With eight threads, there should be two equal batches instead, and this is exactly what we can se in figures 8a and 8b. FIFO should in theory get two distinct peaks, but the resolution in the data does not seem to be high enough for this to appear.

# 4 Final thoughts and lessons learned

It is quite clear that it is possible to spend a considerable amount of time just analyzing schedulers. The results are intriguing as they show clear differences in the behavior between the schedulers, and with several different tasks to compare, even more patterns would surely emerge.

A lesson learned is that one should think carefully about when and where to start and stop the timers. If measuring just the whole task, this is quite easy, but for the individual threads it gets more tricky. The thread that starts the other threads must also start the timers since the working thread may not get to start immediately. Similarly, the working threads must stop their own timers, since there might be a pause between the threads finishing and actually getting joined.

# 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
 */

#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 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;
    struct timespec start;
} 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 = 5;
char c_policies[] = {'n', 'b', 'i', 'f', 'r'};
char *str_policies[] = {"Normal", "Batch", "Idle", "FIFO", "RR"};
int policies[] = {SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE,
```

```c
                   SCHED_FIFO, SCHED_RR};

int main(int argc, char *argv[]) {

    WorkLoad *wl = get_work_load();

    set_settings(wl, argc, argv);

    // Set scheduler
    set_scheduler(wl);

    // Print scheduler to make sure it is set coorectly
    print_schduler();

    run_workers(wl);

    free(wl);
    printf("Done\n");
}

/**
 * get_work_load - initialize the work load and return a pointer to it
 * @return  pointer to allocated memory
 */
WorkLoad *get_work_load() {
    WorkLoad *wl;
    // Allocate memory for work load
    wl = malloc(sizeof(WorkLoad));
    // Initialize work load
    wl->nworkers = 1;
    // wl->data_length = 1073741824; // 2^30
    wl->data_length = LENGTH;
    return wl;
}

/**
* work - a silly attempt to calculate the limit of Grandi's series
* @param packet   the part of the work load to work on
* @return         nothing
*/
void *work(void *packet) {
    Packet *pkt = (Packet *)packet;
    long sum = 0;
    // Starting time
    struct timespec start = pkt->start;
    // Time when finished
    struct timespec end;

    // Do the actual work
    sum = sum + calculate_sum(pkt->index, pkt->length);

    // Get the time when finished
    clock_gettime(CLOCK_REALTIME, &end);
    // Calculate time it took
    double time_taken = (end.tv_sec - start.tv_sec)
                        + (end.tv_nsec - start.tv_nsec)
                        * ONE_OVER_BILLION;
    printf("%lf\n", time_taken);
}

/**
 * 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;

    // Allocate memoy for all packets
    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;
        // Set the starting time
        clock_gettime(CLOCK_REALTIME, &pkt[i]->start);
        if (pthread_create(&threads[i], NULL, work, (void *)pkt[i]) != 0) {
            perror("Could␣not␣create␣thread");
        }
    }
    // Run this thread instead of just waiting for the others
    pkt[i]->index = i * len / num;
    pkt[i]->length = p_len;
    clock_gettime(CLOCK_REALTIME, &pkt[i]->start);
    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;
        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
```

```c
    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
 */
void set_settings(WorkLoad *wl, int argc, char *argv[]) {
    // Two possible options: j(obs) and p(olicy)
    char *optstr = "j:p:";
    int opt;
    char policy = 'n';
    int num_threads = 1;
    int policy_ok = 0;
    int threads_ok = 0;

    // Parse flags
    while ((opt = getopt(argc, argv, optstr)) != -1) {
        char *end;
        switch (opt) {
            case 'p':
            policy = *optarg;
            break;
            case 'j':
            errno = 0;
            num_threads = strtol(optarg, &end, 10);
            if (errno != 0) {
                perror("strtol");
            }
            break;
            default:
            printf("Option␣%c␣not␣supported\n", opt);
        }
    }

    // Check the parsed options
    for (int i = 0; i < num_policies; i++) {
        if (policy == c_policies[i]) {
            policy_ok = 1;
            break;
        }
    }

    // Boundaries for the number of threads
    if (num_threads <= 100 && num_threads > 0) {
        threads_ok = 1;
    }

    // Set values if they are safe, or set defaults
    if (policy_ok) {
        wl->scheduler = policy;
    } else {
        wl->scheduler = 'n';
    }

    if (threads_ok) {
```

```
        wl->nworkers = num_threads;
    } else {
        wl->nworkers = 1;
    }
}
```

Listing 2: timer.sh

```
#!/bin/bash

# timer.sh
#
# A timer script to measure the differences between schedulers/policies
#
# Author: Lennart Jern (ens16ljn@cs.umu.se)

for THREADS in $(seq 1 10)
do
    DATA="Normal,Batch,Idle,FIFO,RR"
    echo "Running␣with␣$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␣>>␣../data/threads$THREADS$POLICY.log"
            # 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 "$DATA" > "../data/data$THREADS.csv"
    chown lennart ../data/threads*
    chown lennart ../data/data*

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.
Also collects the data about individual thread times, and produces similar
statistics for them.
```

```
Lastly, produces a set of plots describing the data.

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

import pandas as pd
import re
import matplotlib.pyplot as plt

def total_stats():
    """
    Read the data from files and calculate statistical values and make plots.
    """
    # The data file names are of the form data<thread count>.csv
    base = "../data/data"
    thread_base = "../data/threads"
    ext = ".csv"
    header = ("Normal", "Batch", "Idle", "FIFO", "RR")
    # Data frames to store the results in
    med = pd.DataFrame(columns=header)          # Medians (total runtime)
    mx = pd.DataFrame(columns=header)           # Max (total runtime)
    mn = pd.DataFrame(columns=header)           # Min (total runtime)
    thread_med = pd.DataFrame(columns=header)   # Medians (threads)
    thread_mx = pd.DataFrame(columns=header)    # Max (threads)
    thread_mn = pd.DataFrame(columns=header)    # Min (threads)

    # For each number of threads
    for i in range(1,11):
        # Build the file name
        f = base + str(i) + ext             # Total run times
        thr_f = thread_base + str(i) + ext  # Thread times
        # Read the time data
        df = pd.read_csv(f)
        thr_df = pd.read_csv(thr_f)

        # Calculate some statistical properties
        med.loc[i] = df.median()
        mx.loc[i] = df.max()
        mn.loc[i] = df.min()
        thread_med.loc[i] = thr_df.median()
        thread_mx.loc[i] = thr_df.max()
        thread_mn.loc[i] = thr_df.min()

        # Plot and save some nice figures
        # Density curves for thread count 2, 4, 6 and 8
        if (i == 2 or i == 4 or i == 6 or i == 8):
            ax = thr_df.plot.kde()
            ax.set_xlabel("Time␣(s)")
            fig = ax.get_figure()
            fig.savefig("density"+str(i)+".pdf")

        # Box plots for all thread counts
        ax = thr_df.plot.box(figsize=(4.5,3))
        ax.set_ylabel("Time␣(s)")
        fig = ax.get_figure()
        fig.savefig("box"+str(i)+".pdf")


    # Calculate ranges
    rng = mx-mn
    thr_rng = mx-mn

```

```python
    # Write everything to files
    data_frames = [med, mx, mn, thread_med, thread_mx, thread_mn, rng, thr_rng]
    base = "../data/"
    names = ["medians", "max", "min", "thread_medians", "thread_max",
             "thread_min", "range", "thread_range"]
    for frm, name in zip(data_frames, names):
        frm.to_csv(base+name + ext, index_label="Threads", float_format="%.5f")


def collect_thread_times(file_name):
    """Read thread times from a file."""
    f = open(file_name)
    times = []
    # Regular expression to find floats
    time = re.compile("(\d+\.\d+)")

    for line in f:
        match = time.match(line)

        if (match):
            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", "RR")

    # 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):
    """For a specific number of threads: Get name of file to write data to"""
    base = "../data/threads"
    ext = ".csv"
    return base + str(threads) + ext

# Collect thread timings
thread_stats()
# Get statistics and plots
total_stats()
```