Operating systems – Assignment 2
Scheduling

Lennart Jern
CS: ens16ljn

Teacher
Ahmed Aley

December 15, 2016

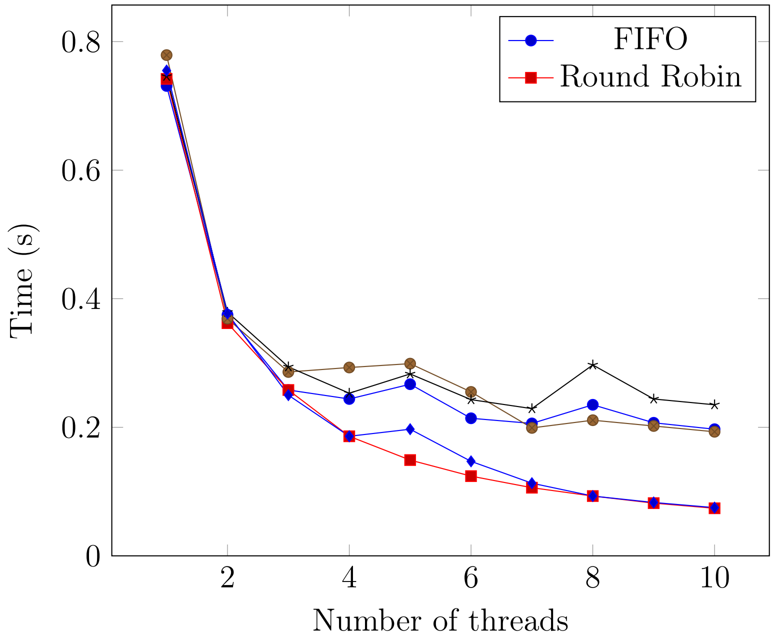


Figure 1: Threads max.

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`

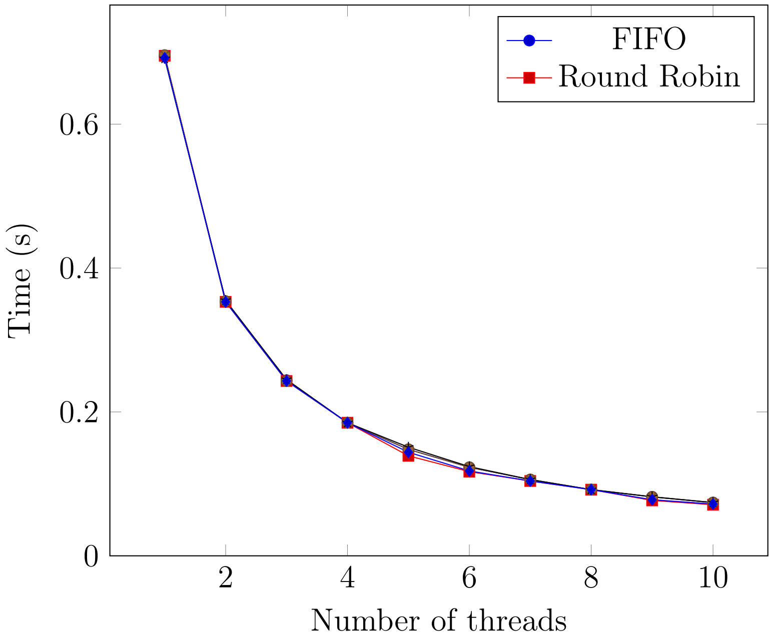


Figure 2: Threads min.

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

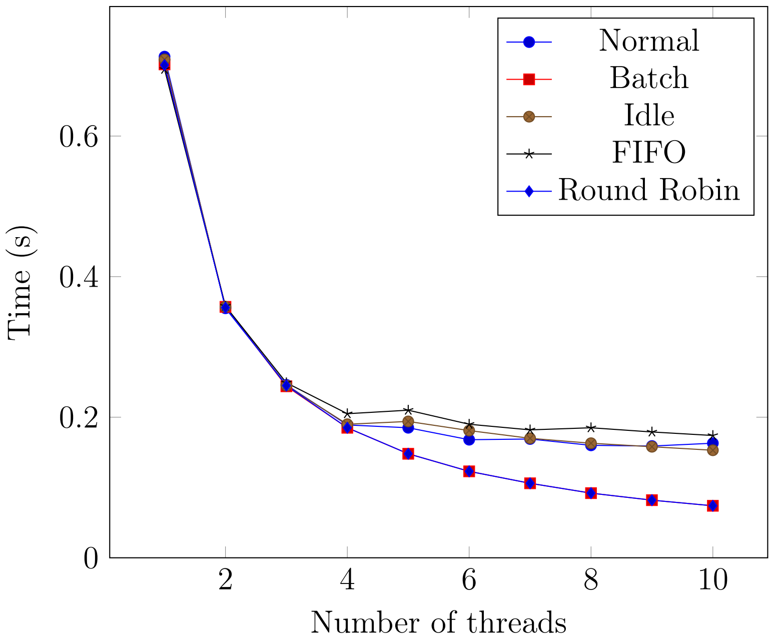


Figure 3: The median time for a single thread.

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 4 for the normal schedulers and in figure **??** 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.

4 Final thoughts and lessons learned

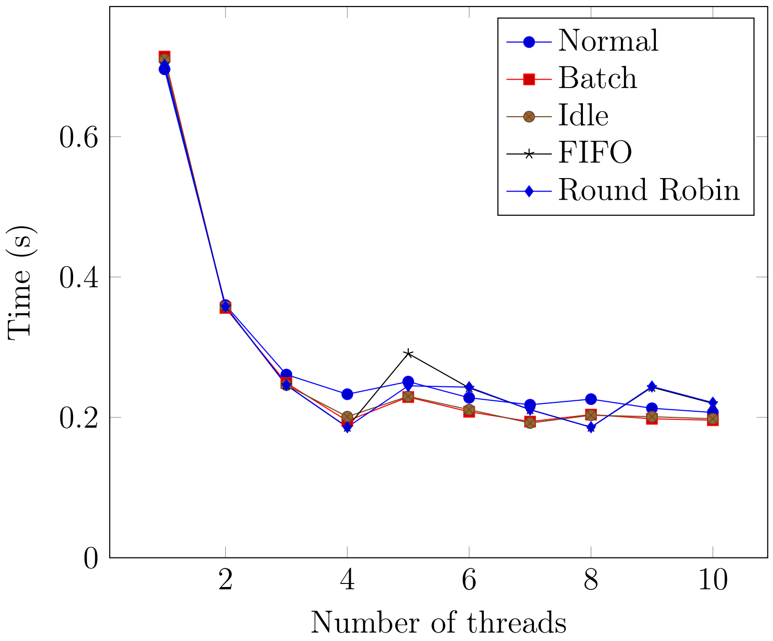


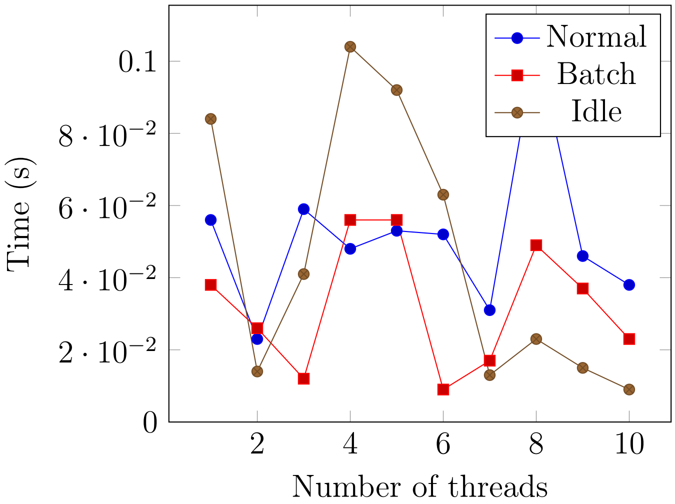
Figure 4: The median time required to finish the complete task.

A Code listings

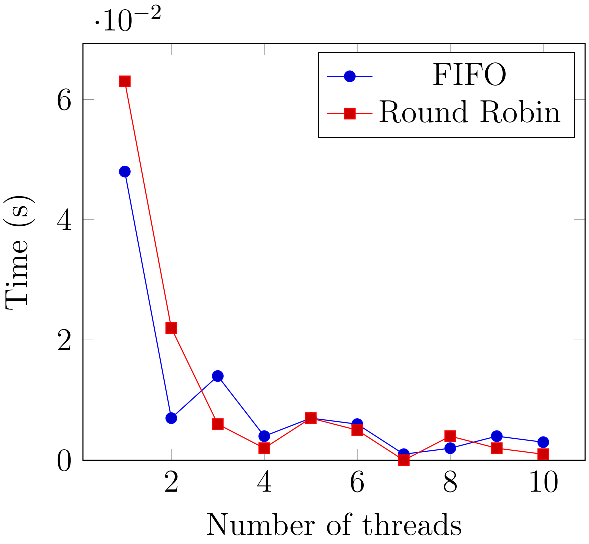
Listing 1: work.c

```
/**
 * 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
```



(a) Range for normal schedulers



(b) Range for real time schedulers.

Figure 5: Range

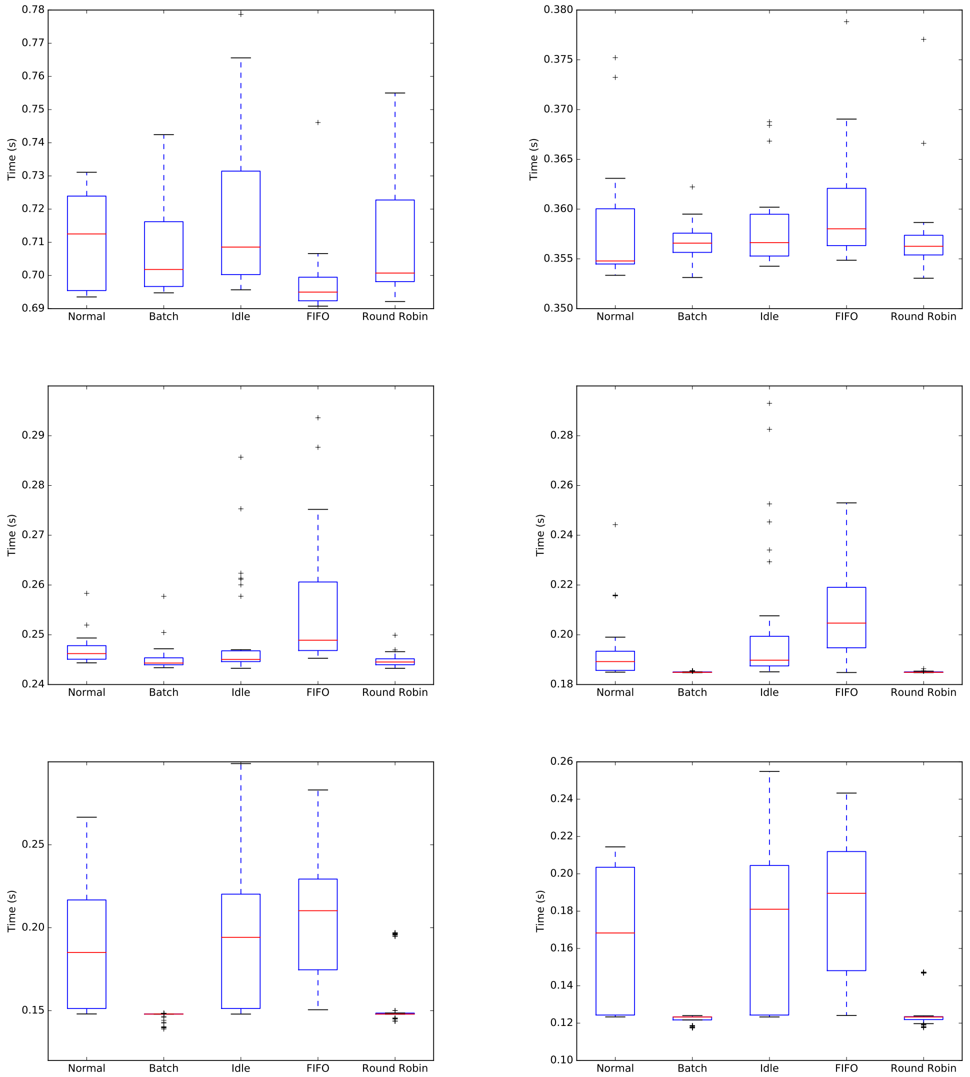


Figure 6: box1

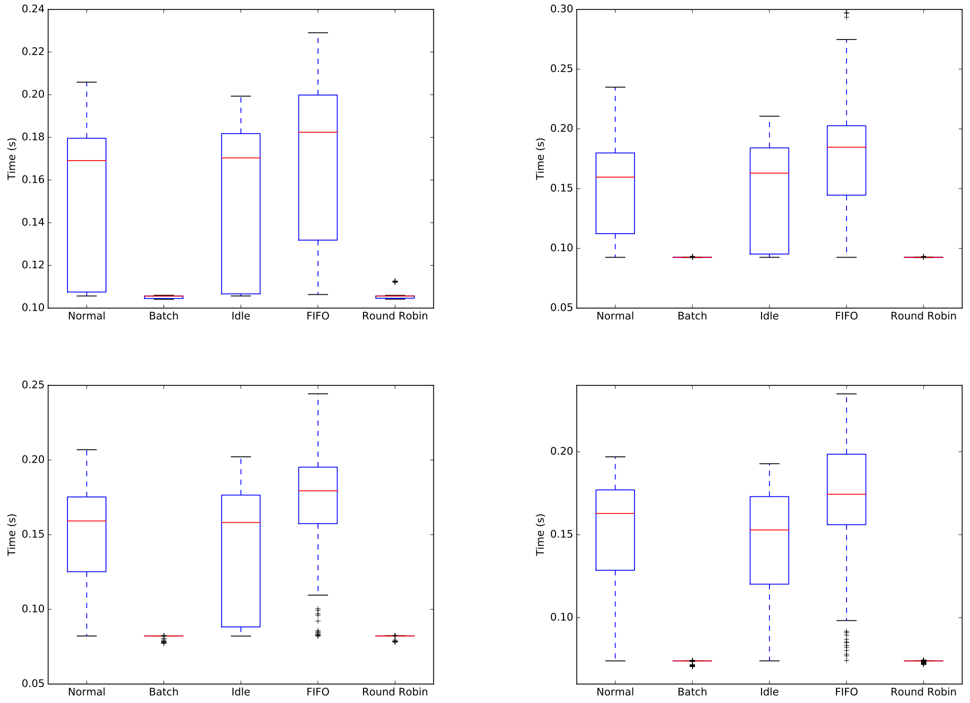


Figure 7: box2

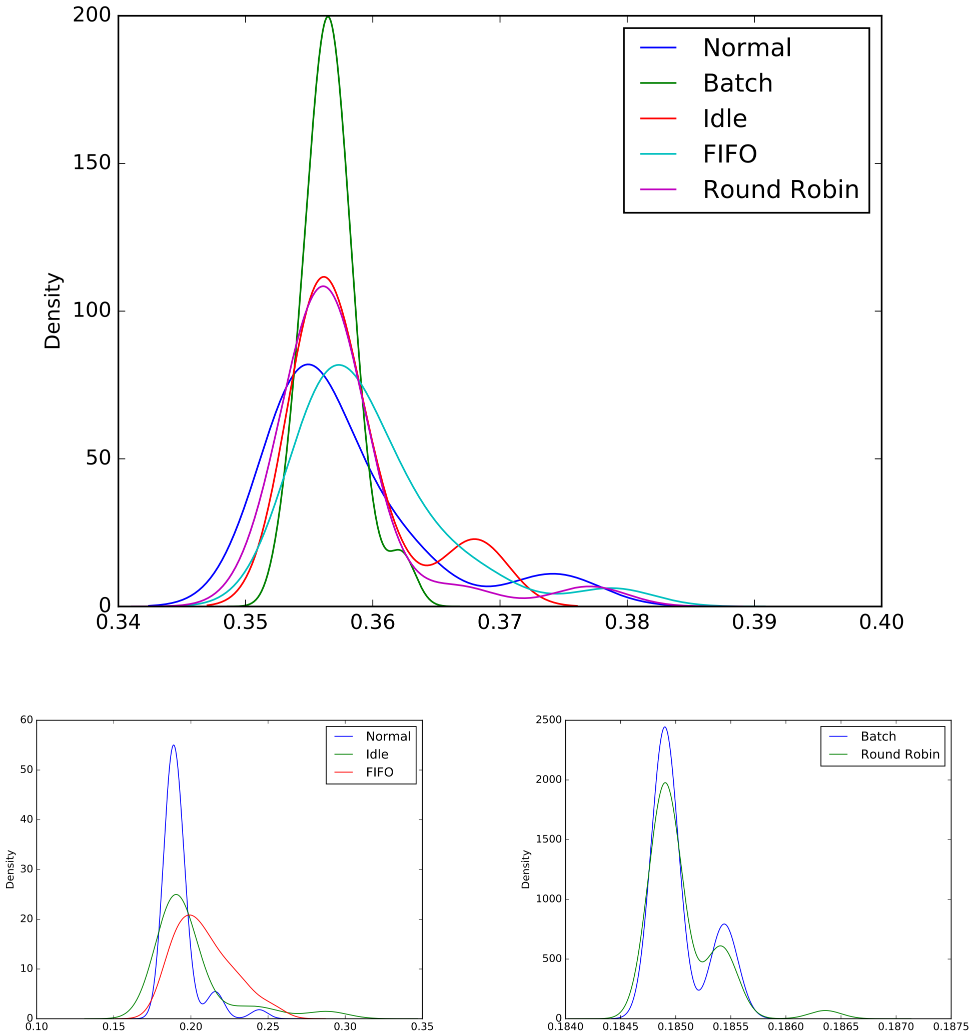


Figure 8: density

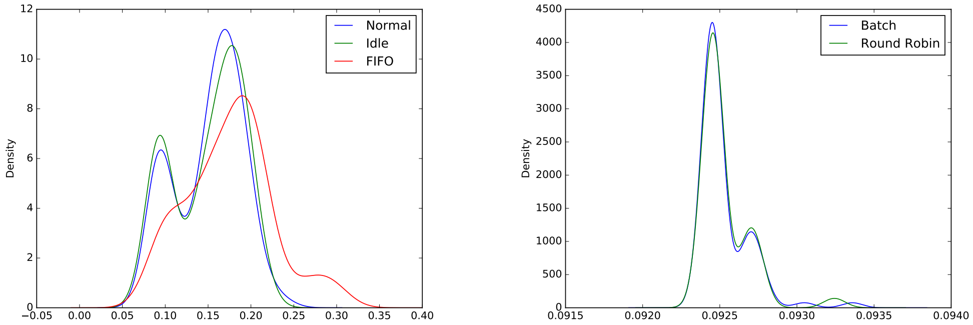


Figure 9: density 8

```

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

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;

    // Calculate time taken by a request
    struct timespec requestStart, requestEnd;
    clock_gettime(CLOCK_REALTIME, &requestStart);

    sum = sum + calculate_sum(pkt->index, pkt->length);

    clock_gettime(CLOCK_REALTIME, &requestEnd);

    // Calculate time it took
    double accum = ( requestEnd.tv_sec - requestStart.tv_sec )
      + ( requestEnd.tv_nsec - requestStart.tv_nsec )
      * ONE_OVER_BILLION;
    printf( "%lf\n", accum );
}

/**
 * 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]);

    // 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
 */
```

```c
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;
        }
    }

    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

```bash
#!/bin/bash

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

```

for THREADS in $(seq 1 10)
do
    DATA="Normal,Batch,Idle,FIFO,Round␣Robin"
    echo "Running␣with␣$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␣>>␣../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*

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
import re
import matplotlib.pyplot as plt

def total_stats():
    # 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", "Round␣Robin")
    # 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 and 8
        if (i == 2):
            ax = thr_df.plot.kde()
            fig = ax.get_figure()
            fig.savefig(’density2.pdf’)

        if (i == 4 or i == 8):
            ax = thr_df[["Normal", "Idle", "FIFO"]].plot.kde()
            fig = ax.get_figure()
            fig.savefig("density"+str(i)+"_nif.pdf")
            ax2 = thr_df[["Batch", "Round␣Robin"]].plot.kde()
            fig2 = ax2.get_figure()
            fig2.savefig("density"+str(i)+"_br.pdf")

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


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

    # 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", "Round␣Robin")

    # 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

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