Operating systems – Assignment 3
I/O Scheduling

Lennart Jern
CS: ens16ljn

Teacher
Ahmed Aley

January 7, 2017

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 **??**. 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 1) was used to time the complete task 10 times for each scheduler, for thread counts ranging from 1 to 10. See code listing 1 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 2. 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

	cfq	deadline	noop
count	1.709e+05	1.709e+05	1.709e+05
mean	4.502e-04	4.424e-04	4.408e-04
std	1.485e-03	1.364e-03	1.367e-03
min	2.300e-08	2.300e-08	2.300e-08
25%	8.600e-08	8.300e-08	8.400e-08
50%	2.150e-07	2.160e-07	2.170e-07
75%	1.879e-05	1.866e-05	1.849e-05
max	7.179e-02	2.453e-02	2.202e-02

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: timer.sh

```
#!/bin/bash

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

# Clean up old results
rm ../data/*.log

SCHEDULERS="cfq noop deadline"
DEVICE="/sys/block/sdc/queue/scheduler"
# Starting directory and expression to search for
START="/run/media/lennart/KINGSTON/test_files expression"
# START="/run/media/lennart/Verbatim/test_files expression"
MNT="/run/media/lennart/KINGSTON"

for S in $SCHEDULERS
do
    echo $S | sudo tee $DEVICE
    echo "Scheduler: `cat $DEVICE`"
    # LINE=""
    # Time the commands 10 times
    for i in $(seq 1 10)
    do
        # Unmount and mount to clear all cache
        sudo umount $MNT
        sudo rm -d /run/media/lennart/KINGSTON
        sudo mkdir /run/media/lennart/KINGSTON
        sudo mount /dev/sdc1 $MNT

        # We use 4 parallell comands that store their respective times in
        # separate log files
        COMMAND1="./mfind $START >> ../data/$S-1.log"
        COMMAND2="./mfind $START >> ../data/$S-2.log"
        COMMAND3="./mfind $START >> ../data/$S-3.log"
        COMMAND4="./mfind $START >> ../data/$S-4.log"

        eval $COMMAND1 &
        eval $COMMAND2 &
        eval $COMMAND3 &
        eval $COMMAND4 &
        # Wait for all commands to finish
        wait

        # A little progress report
        echo "Run $i done."
    done

done

# Restore cfq scheduler
echo "cfq" | sudo tee $DEVICE
```

Listing 2: stats.py

```
"""
stats.py

Process the data produced by timer.sh by calculating the
medians, max values and min values for each scheduler.
Alse plots the density curves.

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

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

def produce_stats():
    """
    Read the data from files, calculate statistical values and make a plot.
    """
    base = "../data/"
    ext = ".csv"
    header=("cfq", "deadline", "noop")

    # Get individual read times as a DataFrame
    df = get_read_times()

    stats = df.describe()
    # Escape per cent chars
    idx = ['count', 'mean', 'std', 'min', '25\%', '50\%', '75\%', 'max']
    stats = stats.set_index([idx])

    # Save stats as csv
    stats.to_csv(base+"stats.csv", header=header, float_format="%.3e")

    # Plot and save the density curves
    ax = df.plot.kde()
    ax.set_xlabel("Time␣(s)")
    ax.set_xlim([0, 0.006])
    fig = ax.get_figure()
    fig.savefig(base+"density.pdf")


def collect_read_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 get_read_times():
    """Collect timing information about all schedulers in a DataFrame."""
    schedulers = ["cfq", "deadline", "noop"]
    base = "../data/"
```

```
    ext = ".log"
    header=("cfq", "deadline", "noop")

    # Collect all times in one file
    times = {key: [] for key in schedulers}
    for s in schedulers:
        # We have 4 parallell log files for each scheduler
        for i in [1,2,3,4]:
            f = base+s+"-"+str(i)+ext
            times[s].extend(collect_read_times(f))
    # Return a DataFrame with all timing data
    df = pd.DataFrame(times)
    return df

# Collect statistical data
produce_stats()
```

Listing 3: generate_test_files.sh

```
#!/bin/bash

# File: generate_test_files.sh
# Author: Lennart Jern - ens16ljn@cs.umu.se
#
# Generate a file tree to do tests on.

# Starting directory
# START=/media/removable/KINGSTON
START=/run/media/lennart/Verbatim

# Move to correct directory/device
cd $START

# Remove tree if existent
rm -r test_files

# Create directory to hold all test files
mkdir test_files
cd test_files

# Split up the files between a few directories
for DIR in a b c d e; do
    mkdir $DIR
    cd $DIR
    # Create empy files
    for F in $(seq 1 20); do
        touch $F
    done

    for D in f g h i j; do
        mkdir $D
        cd $D
    done

    cd "$START/test_files"
done

# Big files
mkdir bigs
cd bigs

```

```
# Generate files with lots of zeros...
for F in $(seq 1 5); do
    head -c 50M < /dev/zero > "file$F"
done

cd "$START/test_files"
# Generate deep folders
for i in $(seq 1 10); do
    mkdir "deep$i"
    cd "deep$i"
    for D in $(seq 1 100); do
        mkdir "dir$D"
        cd "dir$D"
    done
    cd "$START/test_files"
done
```

Listing 4: mfind.c

```
/**
 * File: mfind.c
 * Author: Lennart Jern - ens16ljn@cs.umu.se
 *
 * Usage: ./mfind [-t {d|f|l}] [-p nrthr] start1 [start2 ...] name
 *
 * mfind can search after files, links and directories from given start paths.
 * The search can be done with more than one thread by specifying the flag
 * `-p#`, where # is the number of threads to use.
 *
 */
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <dirent.h>
#include <sys/stat.h>
#include <time.h>           // timing
#include "parser.h"         // Includes list.h

#define ONE_OVER_BILLION 1E-9

void *find_file(void *s_data);
int search_path(SearchData *data, char *path);
int search_directory(SearchData *search_data, DIR *dir, char *path);
int get_dirent(struct dirent *priv_dirent, DIR *dir);
void process_file(char *file_path, char *name,
                  struct stat f_stat, SearchData *data);
int add_dir(LinkedList *list, char *dir_path);
void check_starting_dirs(SearchData *search_data);
void print_path(void *path);
void delete_path(void *path);
void SearchData_delete(SearchData *s_data);

/**
 * main - parse arguments, do the search and then clean up.
 * @param  argc -- number of arguments
 * @param  argv -- array of arguments
 * @return      0 if everything went well, a positive int otherwise
 */
```

```c
int main(int argc, char *argv[]) {
    int ret = 0;

    SearchData *search_data = parse_arguments(argc, argv);
    check_starting_dirs(search_data);

  #ifdef DEBUG
    printf("# Search data\n==============\n");
    printf("# Threads: %d\n", search_data->num_threads);
    printf("# Type: %c\n", search_data->type);
    printf("# Needle: %s\n", search_data->needle);
    List_print(search_data->directories, print_path);
    printf("==============\n\n");
  #endif

    search_data->num_searchers = 0;

    find_file(search_data);

    // Check for errors
    ret = search_data->error;
    // Free allocated memory
    SearchData_delete(search_data);
    return ret;
}

/**
 * find_file - search for files and directories
 * @param  s_data -- search data, containing needle to look for and list of
 *                   directories to look in
 */
void *find_file(void *search_data) {
    unsigned int reads = 0;
    SearchData *data = search_data;
    char *path = NULL;
    int error = 0;

    // Keep searching while there are dirs in the list.
    while((path = (char *)List_get(data->directories)) != NULL) {

        data->num_searchers++;

        reads++;
        if (search_path(data, path) != 0) {
            perror(path);
            // We don't consider permission denied or missing dir as errors
            // error = 1;
        }

        delete_path(path);
        data->num_searchers--;
    } // End while. No more dirs to search and all threads done.
      // Make sure caller knows if there were errors.
    data->error = error;
    printf("Reads: %d\n", reads);
    return NULL;
}

/**
 * search_path - open and search the directory given by path
 * @param  data -- SearchData (what to search for)
 * @param  path -- path to directory to search
```

```
 * @return      0 on successful search, -1 if there were errors
 */
int search_path(SearchData *data, char *path) {
    // Open the directory. If it fails, clean up and continue with the next one.
    DIR *dir = opendir(path);
    int ret = 0;

    if (dir == NULL) {
        ret = -1;
        return ret;
    }

    // Check for matches in the dir
    if (search_directory(data, dir, path) != 0) {
        ret = -1;
    }

    if (closedir(dir) != 0) {
        perror("closedir");
    }
    return ret;
}

/**
 * search_directory - check all files and folders in dir for matches and
 * add folders to the list.
 * @param search_data -- data regarding the search
 * @param dir         -- dir to look in
 * @param path        -- path to the dir (used for printing)
 * @return              0 if everything went well, a poitive int otherwise.
 */
int search_directory(SearchData *search_data, DIR *dir, char *path) {
    struct dirent *priv_dirent;
    struct stat f_stat;
    char *file_path = NULL;
    int at_end = 0;
    int ret = 0;

    while (at_end != 1) {
        priv_dirent = malloc(sizeof(struct dirent));
        if (priv_dirent == NULL) {
            perror("malloc");
            exit(EXIT_FAILURE);
        }

        at_end = get_dirent(priv_dirent, dir);
        if (at_end != 0) {
            // Either error or end of dir
            if (at_end == -1) {
                ret++;
            }
            free(priv_dirent);
            continue;
        }

        // Build file path string
        if (asprintf(&file_path, "%s/%s", path, priv_dirent->d_name) == -1) {
            fprintf(stderr, "Error:␣asprintf␣failed.␣Unable␣to␣set␣file_path.\n");
            free(priv_dirent);
            ret++;
            continue;
        }
```

```
167         // Get stats (file type)
168         if (lstat(file_path, &f_stat) != 0) {
169             perror(file_path);
170             free(priv_dirent);
171             free(file_path);
172             ret++;
173             continue;
174         }
175 
176         // Print matches.
177         process_file(file_path, priv_dirent->d_name, f_stat, search_data);
178 
179         // Add directories to the list (not . and ..)
180         if (S_ISDIR(f_stat.st_mode)
181             && strcmp(priv_dirent->d_name, ".") != 0
182             && strcmp(priv_dirent->d_name, "..") != 0) {
183 
184             if (add_dir(search_data->directories, file_path) != 1) {
185                 fprintf(stderr, "Failed to add directory to list.\n");
186                 return -1;
187             }
188         }
189 
190         free(file_path);
191         file_path = NULL;
192         free(priv_dirent);
193         priv_dirent = NULL;
194     }
195     return ret;
196 }
197 
198 /**
199  * get_dirent - copy the next dirent in dir to priv_dirent in a thread safe way.
200  * This private dirent is safe to use in a multi thread environment.
201  * @param  priv_dirent -- pointer to dirent where the dirent will be saved.
202  * @param  dir         -- dir to read from
203  * @return             -1 on error, 1 when the last element was read and
204  *                        0 otherwise
205  */
206 int get_dirent(struct dirent *priv_dirent, DIR *dir) {
207     struct dirent *dirent;
208     errno = 0;
209 
210     // Starting time
211     struct timespec start;
212     // Time when finished
213     struct timespec end;
214     clock_gettime(CLOCK_REALTIME, &start);
215 
216     dirent = readdir(dir);
217 
218     // Get the time when finished
219     clock_gettime(CLOCK_REALTIME, &end);
220     // Calculate time it took
221     double time_taken = (end.tv_sec - start.tv_sec)
222                         + (end.tv_nsec - start.tv_nsec)
223                         * ONE_OVER_BILLION;
224     printf("%.12lf\n", time_taken);
225 
226     if (errno != 0) {
227         perror("readdir");
228         return -1;
```

```
229     } else if (dirent == NULL) {
230         // No more files to read
231         return 1;
232     }
233     // Copy dirent to private memory
234     memcpy(priv_dirent, dirent, sizeof(struct dirent));
235     return 0;
236 }
237 
238 /**
239  * process_file - print out matching file.
240  * @param file_path -- the path to the file
241  * @param name      -- name of the file
242  * @param f_stat    -- file stats
243  * @param data      -- SearchData (what type and name are we looking for?)
244  */
245 void process_file(char *file_path, char *name,
246                   struct stat f_stat, SearchData *data) {
247     int match = 0;
248     char type = data->type;
249     // Is the name matching?
250     if (strcmp(name, data->needle) == 0) {
251         match = 1;
252     }
253 
254     // Check type, print if we have a match
255     if (S_ISDIR(f_stat.st_mode)) {
256         if (match == 1 && (type == 'd' || type == '\0') ) {
257             printf("%s\n", file_path);
258         }
259     } else if (S_ISREG(f_stat.st_mode)) {
260         if (match == 1 && (type == 'f' || type == '\0') ) {
261             printf("%s\n", file_path);
262         }
263     } else if (S_ISLNK(f_stat.st_mode)) {
264         if (match == 1 && (type == 'l' || type == '\0') ) {
265             printf("%s\n", file_path);
266         }
267     }
268 }
269 
270 /**
271  * add_dir - add a directory to the list in a thread safe manner
272  * @param  list     -- list to add to
273  * @param  dir_path -- path to the directory
274  * @return          1 if the dir was added, 0 if addition failed.
275  */
276 int add_dir(LinkedList *list, char *dir_path) {
277     char *new_dir = malloc(strlen(dir_path)+1);
278     if (new_dir == NULL) {
279         perror("malloc");
280         exit(EXIT_FAILURE);
281     }
282     strcpy(new_dir, dir_path);
283 
284     if (List_append(list, (void *)new_dir) != 1) {
285         return 0;
286     }
287     return 1;
288 }
289 
290 /**
```

```
 * check_starting_dirs - check if the starting dirs match the search criterias
 * @param search_data -- data egarding the search
 */
void check_starting_dirs(SearchData *search_data) {
    char *path;
    struct stat f_stat;
    LinkedList *checked_dirs = List_init();

    // Check all starting dirs for matches
    while((path = (char *)List_get(search_data->directories)) != NULL) {
        if (lstat(path, &f_stat) != 0) {
            perror(path);
            continue;
        }
        // Print if there is a match
        process_file(path, basename(path), f_stat, search_data);

        // Add the checked dir to the new list
        char *new_dir = malloc(strlen(path)+1);
        if (new_dir == NULL) {
            perror("malloc");
            exit(EXIT_FAILURE);
        }
        strcpy(new_dir, path);

        if (List_append(checked_dirs, (void *)new_dir) == 0) {
            fprintf(stderr, "Could␣not␣add␣path␣to␣list.\n");
            search_data->error++;
        }
        free(path);
    }
    // Delete the old list
    List_delete(search_data->directories, delete_path);
    // Add the checked dirs
    search_data->directories = checked_dirs;
}

/**
 * print_path - print out a path
 * @param path -- a void pointer to a path string
 */
void print_path(void *path) {
    char *str = (char *)path;
    printf("%s\n", str);
}

/**
 * delete_path - delete and free any memory occupied by path
 * @param path -- path to be freed
 */
void delete_path(void *path) {
    free(path);
}

/**
 * SearchData_delete - free all memory allocated for s_data
 * @param s_data -- SearchData to free
 */
void SearchData_delete(SearchData *s_data) {
    free(s_data->needle);
    List_delete(s_data->directories, delete_path);
    free(s_data);
```

```
353 }
```

Listing 5: Makefile

```
1  SOURCE=mfind.c list.c
2  OBJECTS=mfind.o list.o parser.o
3  FLAGS=-std=c11 -Wall -pedantic -Werror -pthread
4
5  all: $(OBJECTS)
6          gcc $(OBJECTS) -pthread -o mfind
7          gcc $(FLAGS) crazy_search.c -o crazy_search
8
9  mfind.o: mfind.c
10         gcc $(FLAGS) -c mfind.c
11
12 list.o: list.c list.h
13         gcc $(FLAGS) -c list.c
14
15 parser.o: parser.c parser.h list.h
16         gcc $(FLAGS) -c parser.c
17
18 debug: FLAGS+=-DDEBUG -g
19 debug: all
20
21 test: all
22         ./mfind -td -p2 . .. fail mfind
23
24 time: FLAGS+=-DTIME
25 time: all
26
27 memtest: all
28         valgrind ./mfind -td -p2 . .. mfind
29
30 clean:
31         rm -f mfind *.o
```