Operating systems – Assignment 3
I/O Scheduling

Lennart Jern
CS: ens16ljn

Teacher
Ahmed Aley

January 7, 2017

1 Introduction

Disk access is significantly slower than any CPU operation and is often a bottle neck when it comes to performance. In this report I compare the performance of three I/O schedulers: “cfq”, “noop” and “deadline”. Additionally, all tests are done on two different types of disks: a USB flash drive and a hard disk drive (also connected via USB).

The benchmark used consists of a search for files in a hierarchy of directories. In other words, this report is focused on seek times since nothing is written to disk and just meta data (such as file name and directory content) is read.

2 Method

The benchmark consists of a program (`mfind`, see listing 4) that searches for files in a hierarchy of directories and prints out the time taken for each call to `readdir`. A script (`timer.sh`, see listing 1) is used to run `mfind` in four parallel processes, ten times for each of the three schedulers, and collects all timing data in log files. These logs are then processed by `stats.py` (see listing 2) in order to obtain some statistical properties of the data. The python library Pandas[1] proved very helpful in this regard.

All tests were run on my personal computer with the specifications seen in table 1. The drives used was a Kingston DataTraveler 1 GB and a Verbatim 500 GB portable 2.5” HDD. Both drives were connected to a USB 2.0 port.

3 Results

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.

[1] `http://pandas.pydata.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.

	cfq	deadline	noop
count	$1.709 \cdot 10^{5}$	$1.709 \cdot 10^{5}$	$4.399 \cdot 10^{-4}$
mean	$4.546 \cdot 10^{-4}$	$4.478 \cdot 10^{-4}$	$1.395 \cdot 10^{-3}$
std	$1.465 \cdot 10^{-3}$	$1.396 \cdot 10^{-3}$	$2.100 \cdot 10^{-8}$
min	$2.600 \cdot 10^{-8}$	$2.600 \cdot 10^{-8}$	$8.400 \cdot 10^{-8}$
25%	$8.400 \cdot 10^{-8}$	$8.500 \cdot 10^{-8}$	$2.150 \cdot 10^{-7}$
50%	$2.160 \cdot 10^{-7}$	$2.140 \cdot 10^{-7}$	$1.845 \cdot 10^{-5}$
75%	$1.914 \cdot 10^{-5}$	$1.888 \cdot 10^{-5}$	$4.208 \cdot 10^{-2}$
max	$6.358 \cdot 10^{-2}$	$3.325 \cdot 10^{-2}$	$4.208 \cdot 10^{-2}$

Table 2: Kingston stats

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.

	cfq	deadline	noop
count	$1.709 \cdot 10^{5}$	$1.709 \cdot 10^{5}$	$4.413 \cdot 10^{-4}$
mean	$2.892 \cdot 10^{-4}$	$4.626 \cdot 10^{-4}$	$3.030 \cdot 10^{-3}$
std	$2.018 \cdot 10^{-3}$	$3.082 \cdot 10^{-3}$	$2.400 \cdot 10^{-8}$
min	$2.300 \cdot 10^{-8}$	$2.600 \cdot 10^{-8}$	$1.230 \cdot 10^{-7}$
25%	$1.170 \cdot 10^{-7}$	$1.210 \cdot 10^{-7}$	$2.590 \cdot 10^{-7}$
50%	$2.490 \cdot 10^{-7}$	$2.610 \cdot 10^{-7}$	$2.366 \cdot 10^{-6}$
75%	$2.036 \cdot 10^{-6}$	$2.320 \cdot 10^{-6}$	$8.346 \cdot 10^{-2}$
max	$1.259 \cdot 10^{-1}$	$8.426 \cdot 10^{-2}$	$8.346 \cdot 10^{-2}$

Table 3: Verbatim stats

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"
MNT="/run/media/lennart/Verbatim"

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
        sync
        sleep 1
        sudo umount -f $MNT
        sudo rm -d $MNT
        sudo mkdir $MNT
        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

```python
"""
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.
```

```c
 * @param  argc -- number of arguments
 * @param  argv -- array of arguments
 * @return      0 if everything went well, a positive int otherwise
 */
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;
}

```

```c
/**
 * 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;
        }
        // Get stats (file type)
        if (lstat(file_path, &f_stat) != 0) {
            perror(file_path);
            free(priv_dirent);
            free(file_path);
            ret++;
            continue;
        }

        // Print matches.
        process_file(file_path, priv_dirent->d_name, f_stat, search_data);

        // Add directories to the list (not . and ..)
        if (S_ISDIR(f_stat.st_mode)
            && strcmp(priv_dirent->d_name, ".") != 0
            && strcmp(priv_dirent->d_name, "..") != 0) {

            if (add_dir(search_data->directories, file_path) != 1) {
                fprintf(stderr, "Failed␣to␣add␣directory␣to␣list.\n");
                return -1;
            }
        }

        free(file_path);
        file_path = NULL;
        free(priv_dirent);
        priv_dirent = NULL;
    }
    return ret;
}

/**
 * get_dirent - copy the next dirent in dir to priv_dirent in a thread safe way.
 * This private dirent is safe to use in a multi thread environment.
 * @param  priv_dirent -- pointer to dirent where the dirent will be saved.
 * @param  dir         -- dir to read from
 * @return             -1 on error, 1 when the last element was read and
 *                        0 otherwise
 */
int get_dirent(struct dirent *priv_dirent, DIR *dir) {
    struct dirent *dirent;
    errno = 0;

    // Starting time
    struct timespec start;
    // Time when finished
    struct timespec end;
    clock_gettime(CLOCK_REALTIME, &start);

    dirent = readdir(dir);

    // 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("%.12lf\n", time_taken);
```

```c

    if (errno != 0) {
        perror("readdir");
        return -1;
    } else if (dirent == NULL) {
        // No more files to read
        return 1;
    }
    // Copy dirent to private memory
    memcpy(priv_dirent, dirent, sizeof(struct dirent));
    return 0;
}

/**
 * process_file - print out matching file.
 * @param file_path -- the path to the file
 * @param name      -- name of the file
 * @param f_stat    -- file stats
 * @param data      -- SearchData (what type and name are we looking for?)
 */
void process_file(char *file_path, char *name,
                  struct stat f_stat, SearchData *data) {
    int match = 0;
    char type = data->type;
    // Is the name matching?
    if (strcmp(name, data->needle) == 0) {
        match = 1;
    }

    // Check type, print if we have a match
    if (S_ISDIR(f_stat.st_mode)) {
        if (match == 1 && (type == 'd' || type == '\0') ) {
            printf("%s\n", file_path);
        }
    } else if (S_ISREG(f_stat.st_mode)) {
        if (match == 1 && (type == 'f' || type == '\0') ) {
            printf("%s\n", file_path);
        }
    } else if (S_ISLNK(f_stat.st_mode)) {
        if (match == 1 && (type == 'l' || type == '\0') ) {
            printf("%s\n", file_path);
        }
    }
}

/**
 * add_dir - add a directory to the list in a thread safe manner
 * @param  list     -- list to add to
 * @param  dir_path -- path to the directory
 * @return          1 if the dir was added, 0 if addition failed.
 */
int add_dir(LinkedList *list, char *dir_path) {
    char *new_dir = malloc(strlen(dir_path)+1);
    if (new_dir == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    strcpy(new_dir, dir_path);

    if (List_append(list, (void *)new_dir) != 1) {
        return 0;
    }
    return 1;
}

/**
 * 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
 */
```

```c
void SearchData_delete(SearchData *s_data) {
    free(s_data->needle);
    List_delete(s_data->directories, delete_path);
    free(s_data);
}
```

Listing 5: Makefile

```makefile
SOURCE=mfind.c list.c
OBJECTS=mfind.o list.o parser.o
FLAGS=-std=c11 -Wall -pedantic -Werror -pthread

all: $(OBJECTS)
	gcc $(OBJECTS) -pthread -o mfind
	gcc $(FLAGS) crazy_search.c -o crazy_search

mfind.o: mfind.c
	gcc $(FLAGS) -c mfind.c

list.o: list.c list.h
	gcc $(FLAGS) -c list.c

parser.o: parser.c parser.h list.h
	gcc $(FLAGS) -c parser.c

debug: FLAGS+=-DDEBUG -g
debug: all

test: all
	./mfind -td -p2 . .. fail mfind

time: FLAGS+=-DTIME
time: all

memtest: all
	valgrind ./mfind -td -p2 . .. mfind

clean:
	rm -f mfind *.o
```