Operating systems – Assignment 3
I/O Scheduling

Lennart Jern
CS: ens16ljn

Teacher
Ahmed Aley

January 8, 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. The partition is unmounted between each run to make sure that no files are cached.

Processing of the data is done by `stats.py` (see listing 2) in order to obtain some statistical properties. The python library Pandas[1] proved very helpful in this regard.

Remark 1. *TODO: The program `mfind` is repurposed...*

A script was also used to create the tree of searched directories in order to make the benchmark easier to reproduce. This script can be found in listing 3.

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.

[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.

3 Results

The calculated statistical properties of the timing data can be seen in tables 2 and 3. Only slight differences can be seen between the schedulers for the flash drive. Most noticeably, the maximum time required is considerably greater for cfq than for the other schedulers. This also reflects on a greater standard deviation.

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

Table 2: Kingston stats

Looking at the means and medians (50 % quantile), we see that noop has the lowest mean, but is beaten by deadline when it comes to the median. Noop also has the lowest minimum time required.

The picture is quite different when looking at table 3. This is for a traditional spinning hard drive, so the cfq scheduler get to shine. The cfq scheduler has the best mean, minimum, median and 75 % quantile. All this comes

at a cost, of course, the maximum time is far greater than for the other schedulers. Despite this, however, the cfq still manages to have the lowest standard deviation among the three.

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

Table 3: Verbatim stats

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"
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

```
"""
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
 */
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);
```

```
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 /**
291  * check_starting_dirs - check if the starting dirs match the search criterias
292  * @param search_data -- data egarding the search
293  */
294 void check_starting_dirs(SearchData *search_data) {
295     char *path;
296     struct stat f_stat;
297     LinkedList *checked_dirs = List_init();
298 
299     // Check all starting dirs for matches
300     while((path = (char *)List_get(search_data->directories)) != NULL) {
301         if (lstat(path, &f_stat) != 0) {
302             perror(path);
303             continue;
304         }
305         // Print if there is a match
306         process_file(path, basename(path), f_stat, search_data);
307 
308         // Add the checked dir to the new list
309         char *new_dir = malloc(strlen(path)+1);
310         if (new_dir == NULL) {
311             perror("malloc");
312             exit(EXIT_FAILURE);
313         }
314         strcpy(new_dir, path);
315 
316         if (List_append(checked_dirs, (void *)new_dir) == 0) {
317             fprintf(stderr, "Could␣not␣add␣path␣to␣list.\n");
318             search_data->error++;
319         }
320         free(path);
321     }
322     // Delete the old list
323     List_delete(search_data->directories, delete_path);
324     // Add the checked dirs
325     search_data->directories = checked_dirs;
326 }
327 
328 /**
329  * print_path - print out a path
330  * @param path -- a void pointer to a path string
331  */
332 void print_path(void *path) {
333     char *str = (char *)path;
334     printf("%s\n", str);
335 }
336 
337 /**
338  * delete_path - delete and free any memory occupied by path
339  * @param path -- path to be freed
340  */
341 void delete_path(void *path) {
342     free(path);
343 }
344 
345 /**
346  * SearchData_delete - free all memory allocated for s_data
347  * @param s_data -- SearchData to free
348  */
```

```
349 void SearchData_delete(SearchData *s_data) {
350     free(s_data->needle);
351     List_delete(s_data->directories, delete_path);
352     free(s_data);
353 }
```

Listing 5: list.c

```
1  /**
2   * File: list.c
3   * Author: Lennart Jern - ens16ljn@cs.umu.se
4   *
5   * A simple implementation of a linked list.
6   *
7   */
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12 #include "list.h"
13
14 /**
15  * List_init
16  * Create and initialize a LinkedList.
17  *
18  * @return  pointer to list
19  */
20 LinkedList *List_init(void) {
21     LinkedList *lst = calloc(1, sizeof(LinkedList));
22     if (lst == NULL) {
23         fprintf(stderr, "Allocation␣of␣memory␣for␣linked␣list␣failed\n");
24         exit(EXIT_FAILURE);
25     }
26     return lst;
27 }
28
29 /**
30  * Appends an element with the specified value to the list
31  * @param  l:     List to append to
32  * @param  value: Pointer to value to be added
33  * @return       1 on success, otherwise 0.
34  */
35 int List_append(LinkedList *l, void *value) {
36
37     Node *new_node = calloc(1, sizeof(Node));
38     if (new_node == NULL) {
39         fprintf(stderr, "Failed␣to␣allocate␣memory␣for␣new␣node.\n");
40         return 0;
41     }
42     new_node->value = value;
43     new_node->next = NULL;
44
45     if (l->first == NULL) {
46         l->first = new_node;
47         return 1;
48     }
49     Node *node_ptr = l->first;
50
51     while (node_ptr->next != NULL) {
52         node_ptr = node_ptr->next;
53     }
```

```
54      node_ptr->next = new_node;
55      return 1;
56  }
57
58  /**
59   * List_get - get the value of the first node in the list and delete the node
60   * from the list.
61   * @param  l -- the list
62   * @return   -- pointer to value, remember to free it later.
63   */
64  void *List_get(LinkedList *l) {
65      if (l->first == NULL) {
66          return NULL;
67      }
68      Node *node = l->first;
69      l->first = l->first->next;
70      void *value = node->value;
71      free(node);
72      node = NULL;
73      return value;
74  }
75
76  /**
77   * List_sort
78   * Sort the list lst by selection sort, using the comparison function comp.
79   * @param lst:  LinkedList to sort
80   * @param comp: function used to compare the values of two nodes to determine
81   *              what order they should be placed in.
82   */
83  void List_sort(LinkedList *lst, int (*comp)(void *value1, void *value2)) {
84      if (lst->first == NULL) {
85          // Empty list, nothing to do
86          return;
87      }
88      Node *boundary = lst->first; // ordered nodes before this
89      Node *smallest = lst->first; // should be placed next in order
90      Node *last_sorted = NULL; // add smallest after this one
91      Node *current = lst->first;
92      Node *prev = NULL;
93      Node *before_smallest = NULL;
94
95      // run untill the whole list is sorted
96      while (boundary->next != NULL) {
97          smallest = boundary;
98          current = boundary;
99          prev = NULL;
100         before_smallest = NULL;
101         // loop through the unordered part of the list and pick out the node
102         // with the "smallest" value
103         while (current->next != NULL) {
104             prev = current;
105             current = current->next;
106             if (comp(smallest->value, current->value) < 0) {
107                 smallest = current;
108                 before_smallest = prev;
109             }
110         }
111
112         // Do we have to move smallest?
113         if (before_smallest != NULL) {
114             // cut out smallest
115             before_smallest->next = smallest->next;
```

```
116             if (last_sorted == NULL) {
117                 // place it first
118                 smallest->next = lst->first;
119                 lst->first = smallest;
120             } else {
121                 // place it after last_sorted
122                 smallest->next = last_sorted->next;
123                 last_sorted->next = smallest;
124             }
125         }
126         // update last_sorted and boundary
127         last_sorted = smallest;
128         boundary = smallest->next;
129     }
130 }
131
132 /**
133  * List_remove
134  * Removes an element from the bottom of the list and frees the allocated memory
135  * by calling delete_value(value).
136  *
137  * @param  lst: the list to remove from
138  * @return     1 on successful removal, 0 if no node was removed.
139  */
140 int List_remove(LinkedList *lst, void (*delete_value)(void *value)) {
141     Node *node = lst->first;
142     Node *prev = node;
143     if (node == NULL) {
144         return 0;
145     }
146     if (node->next == NULL) { // Only one element in list
147         delete_value(node->value);
148         free(node);
149         lst->first = NULL;
150         return 1;
151     }
152     while (node->next != NULL) {
153         prev = node;
154         node = node->next;
155     }
156     prev->next = NULL;
157     delete_value(node->value);
158     free(node);
159     return 1;
160 }
161
162 /**
163  * List_delete
164  * Frees all memory allocated by the nodes in the LinkedList
165  * using delete_value(value) and frees the list itself after that.
166  *
167  * @param lst:          LinkedList to free.
168  * @param delete_value: function used to free the memory allocated by the value
169  *                      of a node.
170  */
171 void List_delete(LinkedList *lst, void (*delete_value)(void *value)) {
172     if (lst) {
173         while(List_remove(lst, delete_value)) ;
174         free(lst);
175     }
176 }
177
```

```
178 /**
179  * List_print
180  * Prints the LinkedList using the function provided.
181  *
182  * @param lst:   LinkedList to print.
183  * @param print: function used to print the node values.
184  */
185 void List_print(LinkedList *lst, void (*print)(void *value)) {
186     Node *current = lst->first;
187 
188     while (current != NULL) {
189         print(current->value);
190         current = current->next;
191     }
192 }
```

Listing 6: list.h

```
1  /**
2   * File: list.h
3   * Author: Lennart Jern - ens16ljn@cs.umu.se
4   *
5   * This is a header file for my own implementation of a liked list.
6   *
7   */
8 
9  typedef struct linked_list LinkedList;
10 typedef struct node Node;
11 
12 struct node {
13         void *value;
14         struct node *next;
15 };
16 
17 struct linked_list {
18         Node *first;
19 };
20 
21 struct user_info {
22         unsigned int uid;
23         char *uname;
24 };
25 
26 /**
27  * List_init
28  * Create and initialize a LinkedList.
29  *
30  * @return  pointer to list
31  */
32 LinkedList * List_init(void);
33 
34 /**
35  * Appends an element with the specified value to the list
36  * @param  l:     List to append to
37  * @param  value: Pointer to value to be added
38  * @return       1 on success, otherwise 0.
39  */
40 int List_append(LinkedList *l, void *value);
41 
42 /**
43  * List_get - get the first element from the list. The element is removed
```

```
 * from the list so remember to free it when you are done.
 * @param  l -- the list
 * @return   -- a pointer to the first value in the list
 */
void *List_get(LinkedList *l);

/**
 * List_sort
 * Sort the list lst by selection sort, using the comparison function comp.
 * @param lst:  LinkedList to sort
 * @param comp: function used to compare the values of two nodes to determine
 *              what order they should be placed in.
 */
void List_sort(LinkedList *lst, int (*comp)(void *value1, void *value2));

/**
 * List_remove
 * Removes an element from the bottom of the list and frees the allocated memory
 * by calling delete_value(value).
 *
 * @param  lst: the list to remove from
 * @return     1 on successful removal, 0 if no node was removed.
 */
int List_remove(LinkedList *lst, void (*delete_value)(void *value));

/**
 * List_delete
 * Frees all memory allocated by the nodes in the LinkedList
 * using delete_value(value) and frees the list itself after that.
 *
 * @param lst:          LinkedList to free.
 * @param delete_value: function used to free the memory allocated by the value
 *                      of a node.
 */
void List_delete(LinkedList *lst, void (*delete_value)(void *value));

/**
 * List_print
 * Prints the LinkedList using the function provided.
 *
 * @param lst:   LinkedList to print.
 * @param print: function used to print the node values.
 */
void List_print(LinkedList *lst, void (*print)(void *value));
```

Listing 7: parser.c

```
/**
 * File: parser.c
 * Author: Lennart Jern - ens16ljn@cs.umu.se
 *
 * This parser extracts search data from command line arguments.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <errno.h>
#include <string.h>
#include "parser.h"
```

```
15
16 /**
17  * parse_arguments - parse command line arguments into SearchData.
18  * The SearchData should be freed by calling SearchData_delete when
19  * you are done.
20  * @param  argc -- number of arguments
21  * @param  argv -- array of arguments
22  * @return      SearchData containing: dirs to look in, needle to look
23  *                    for, file type and number of threads.
24  */
25 SearchData *parse_arguments(int argc, char *argv[]) {
26     SearchData *s_data;
27     char *usage = "Usage:␣./mfind␣[-t␣{d|f|l}]␣[-p␣nrthr]␣start1␣[start2␣...]␣name";
28
29     // No point to continue if there are less than 3 args
30     if (argc < 3) {
31         fprintf(stderr, "%s\n", usage);
32         exit(EXIT_FAILURE);
33     }
34
35     s_data = malloc(sizeof(SearchData));
36     if (s_data == NULL) {
37         perror("malloc");
38         exit(EXIT_FAILURE);
39     }
40
41     // Set num_threads and type from given arguments
42     if (set_flags(s_data, argc, argv) != 0) {
43         fprintf(stderr, "%s\n", usage);
44         exit(EXIT_FAILURE);
45     }
46     // Make sure the flags are safe
47     check_flags(s_data);
48
49     // Check that there is at least one dir to look in and a name to look for.
50     if (optind >= argc -1) {
51         fprintf(stderr, "%s\n", usage);
52         free(s_data);
53         exit(EXIT_FAILURE);
54     }
55
56     add_dirs_and_needle(s_data, argv, optind, argc-1);
57
58     return s_data;
59 }
60
61 /**
62  * set_flags - parse the arguments and add corresponding search data
63  * @param s_data -- SearchData to add info to
64  * @param argc   -- number of arguments
65  * @param argv   -- array of arguments
66  * @return          0 if everything went well, -1 otherwise.
67  */
68 int set_flags(SearchData *s_data, int argc, char *argv[]) {
69     char *optstr = "t:p:";
70     int opt;
71     char type = '\0';
72     int num_threads = 1;
73
74     // Parse flags
75     while ((opt = getopt(argc, argv, optstr)) != -1) {
76         char *end;
```

```
77          switch (opt) {
78          case 't':
79              type = *optarg;
80              break;
81          case 'p':
82              errno = 0;
83              num_threads = strtol(optarg, &end, 10);
84              if (errno != 0) {
85                  perror("strtol");
86              }
87              break;
88          default:
89              return -1;
90          }
91      }
92
93      s_data->num_threads = num_threads;
94      s_data->type = type;
95      return 0;
96  }
97
98  /**
99   * check_flags - make sure the flags (num_threads and type) are correct and safe
100  * @param s_data -- SearchData to check.
101  */
102 void check_flags(SearchData *s_data) {
103     int num_threads = s_data->num_threads;
104     char type = s_data->type;
105
106     if (num_threads < 1) {
107         fprintf(stderr, "Number␣of␣threads␣must␣be␣more␣than␣0!\n");
108         exit(EXIT_FAILURE);
109     } else if (num_threads > MAXTHREADS) {
110         fprintf(stderr, "Too␣many␣threads!␣Resetting␣to␣1.\n");
111         num_threads = 1;
112     }
113     if (type != 'd' && type != 'f' && type != 'l' && type != '\0') {
114         fprintf(stderr, "Type␣must␣be␣d|f|l.\n");
115         exit(EXIT_FAILURE);
116     }
117
118     s_data->num_threads = num_threads;
119     s_data->type = type;
120 }
121
122 /**
123  * add_dirs_and_needle - parse starting dirs and needle from arguments and
124  * add them to the search data.
125  * @param s_data -- search data to add to
126  * @param argv   -- array of arguments
127  * @param first  -- index of first directory in argv
128  * @param last   -- index of last element in argv. This should be the needle.
129  */
130 void add_dirs_and_needle(SearchData *s_data, char *argv[], int first, int last) {
131     LinkedList *dirs = List_init();
132     char *needle;
133     int num_dirs = last - first;
134
135     // Add all starting directories to the list of dirs.
136     for (int i = 0; i < num_dirs; i++) {
137         char *new_dir = malloc(strlen(argv[first+i])+1);
138         if (new_dir == NULL) {
```

```
            perror("malloc");
            exit(EXIT_FAILURE);
        }
        strcpy(new_dir, argv[first+i]);

        if (List_append(dirs, (void *)new_dir) == 0) {
            fprintf(stderr, "Could not add path to list.\n");
            s_data->error++;
        }
    }

    // The last arg is the needle/name to search for
    needle = argv[last];
    s_data->needle = malloc(strlen(needle)+1);
    if (s_data->needle == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    // Add everything to s_data.
    strcpy(s_data->needle, needle);
    s_data->directories = dirs;
}
```

Listing 8: parser.h

```
/**
 * File: parser.h
 * Author: Lennart Jern - ens16ljn@cs.umu.se
 *
 * This is a header file for a parser that extracts search data from command
 * line arguments.
 *
 */

#include "list.h"

#define MAXTHREADS (1)

typedef struct search_data SearchData;

struct search_data {
        char *needle;
        LinkedList *directories;
        int num_threads;
        char type;
        int num_searchers;
        unsigned int error;
};

SearchData *parse_arguments(int argc, char *argv[]);
int set_flags(SearchData *s_data, int argc, char *argv[]);
void check_flags(SearchData *s_data);
void add_dirs_and_needle(SearchData *s_data, char *argv[], int first, int last);
```

Listing 9: Makefile

```
SOURCE=mfind.c list.c
OBJECTS=mfind.o list.o parser.o
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
```