# 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 1) 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 2) 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 3) in order to obtain some statistical properties. The python library Pandas[1] proved very helpful in this regard.

**Remark 1.** *The program `mfind` was originally made for another assignment and was here simplified and modified to measure the time required for each I/O request. In addition to `mfind.c` the program also consists of a parser (`parser.c` and `parser.h` in listings 4 and 5) and the linked list that keeps track of what directories to search (`list.c` and `list.h` in listings 6 and 7). The complete program can be compiled using the make file in listing 8.*

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 9.

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

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

port.

| 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: Statistical properties for the Kingston flash drive. The time values are measured in seconds.

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: Statistical properties for the Verbatim HDD. The time values are measured in seconds.

A comparison of the two drives reveals that the scheduler should be chosen depending on the type of drive for best performance. On a traditional spinning drive, there is more to win by ordering the requests in an efficient way. However, the ordering does not matter much on a solid state drive, since there is no physical movement required. In that case is is better not to spend time finding the "perfect order" and thus delay some requests.

In order to improve the noop scheduler on traditional HDDs we would have to allow a higher maximum time. This would give the scheduler a better possibility to control the ordering of the requests.

Similarly, the cfq scheduler would probably be faster on flash drives if it just sent all requests in the order they came in, instead of reordering them. The deadline scheduler is similar in performance to the noop scheduler on both drives and could probably improve it's performance on HDDs in the same way. My guess is that the deadline and noop schedulers would show greater differences if the requests were longer.

# 4 Final thoughts and lessons learned

Caching of data gave some problems at first, but this could easily be solve by unmounting the drive between each run. However, the unmounting was quite problematic for the HDD since it would sometimes remain busy even after a `sync` call. To solve this, a one second sleep was added to the script.

The problem with `sync` and `umount` made me quite suspicious of the data gathered. I am uncertain about what really should be timed and how to make sure that the I/O requests are actually done on the disk and not just in some cache. Perhaps more fine tuned control would be needed to solve these problems, with calls to `syncfs` periodically. But that makes it hard to determine the actual time taken for individual requests.

One thing is clear: the I/O scheduler can definitely have a great impact on performance and it can be quite tricky to accurately measure the performance of these schedulers. Some theoretical background would have been interesting and helpful if time would have allowed for that during the lectures.

# A Code listings

Listing 1: mfind.c

```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

```

```
58      search_data->num_searchers = 0;
59
60      find_file(search_data);
61
62      // Check for errors
63      ret = search_data->error;
64      // Free allocated memory
65      SearchData_delete(search_data);
66      return ret;
67  }
68
69  /**
70   * find_file - search for files and directories
71   * @param  s_data -- search data, containing needle to look for and list of
72   *                   directories to look in
73   */
74  void *find_file(void *search_data) {
75      unsigned int reads = 0;
76      SearchData *data = search_data;
77      char *path = NULL;
78      int error = 0;
79
80      // Keep searching while there are dirs in the list.
81      while((path = (char *)List_get(data->directories)) != NULL) {
82
83          data->num_searchers++;
84
85          reads++;
86          if (search_path(data, path) != 0) {
87              perror(path);
88              // We don't consider permission denied or missing dir as errors
89              // error = 1;
90          }
91
92          delete_path(path);
93          data->num_searchers--;
94      } // End while. No more dirs to search and all threads done.
95        // Make sure caller knows if there were errors.
96      data->error = error;
97      printf("Reads:␣%d\n", reads);
98      return NULL;
99  }
100
101 /**
102  * search_path - open and search the directory given by path
103  * @param  data -- SearchData (what to search for)
104  * @param  path -- path to directory to search
105  * @return      0 on successful search, -1 if there were errors
106  */
107 int search_path(SearchData *data, char *path) {
108     // Open the directory. If it fails, clean up and continue with the next one.
109     DIR *dir = opendir(path);
110     int ret = 0;
111
112     if (dir == NULL) {
113         ret = -1;
114         return ret;
115     }
116
117     // Check for matches in the dir
118     if (search_directory(data, dir, path) != 0) {
119         ret = -1;
```

```
120     }
121
122     if (closedir(dir) != 0) {
123         perror("closedir");
124     }
125     return ret;
126 }
127
128 /**
129  * search_directory - check all files and folders in dir for matches and
130  * add folders to the list.
131  * @param search_data -- data regarding the search
132  * @param dir         -- dir to look in
133  * @param path        -- path to the dir (used for printing)
134  * @return              0 if everything went well, a poitive int otherwise.
135  */
136 int search_directory(SearchData *search_data, DIR *dir, char *path) {
137     struct dirent *priv_dirent;
138     struct stat f_stat;
139     char *file_path = NULL;
140     int at_end = 0;
141     int ret = 0;
142
143     while (at_end != 1) {
144         priv_dirent = malloc(sizeof(struct dirent));
145         if (priv_dirent == NULL) {
146             perror("malloc");
147             exit(EXIT_FAILURE);
148         }
149
150         at_end = get_dirent(priv_dirent, dir);
151         if (at_end != 0) {
152             // Either error or end of dir
153             if (at_end == -1) {
154                 ret++;
155             }
156             free(priv_dirent);
157             continue;
158         }
159
160         // Build file path string
161         if (asprintf(&file_path, "%s/%s", path, priv_dirent->d_name) == -1) {
162             fprintf(stderr, "Error:␣asprintf␣failed.␣Unable␣to␣set␣file_path.\n");
163             free(priv_dirent);
164             ret++;
165             continue;
166         }
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 /**
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 2: timer.sh

```
1  #!/bin/bash
2
3  # timer.sh
4  #
5  # A timer script to measure the differences between i/o schedulers
6  #
7  # Author: Lennart Jern (ens16ljn@cs.umu.se)
8
9  # Clean up old results
10 rm ../data/*.log
```

```
11
12 SCHEDULERS="cfq␣noop␣deadline"
13 DEVICE="/sys/block/sdc/queue/scheduler"
14 # Starting directory and expression to search for
15 # START="/run/media/lennart/KINGSTON/test_files expression"
16 START="/run/media/lennart/Verbatim/test_files␣expression"
17 # MNT="/run/media/lennart/KINGSTON"
18 MNT="/run/media/lennart/Verbatim"
19
20 for S in $SCHEDULERS
21 do
22     echo $S | sudo tee $DEVICE
23     echo "Scheduler:␣‘cat␣$DEVICE‘"
24     # LINE=""
25     # Time the commands 10 times
26     for i in $(seq 1 10)
27     do
28         # Unmount and mount to clear all cache
29         sync
30         sleep 1
31         sudo umount -f $MNT
32         sudo rm -d $MNT
33         sudo mkdir $MNT
34         sudo mount /dev/sdc1 $MNT
35
36         # We use 4 parallell comands that store their respective times in
37         # separate log files
38         COMMAND1="./mfind␣$START␣>>␣../data/$S-1.log"
39         COMMAND2="./mfind␣$START␣>>␣../data/$S-2.log"
40         COMMAND3="./mfind␣$START␣>>␣../data/$S-3.log"
41         COMMAND4="./mfind␣$START␣>>␣../data/$S-4.log"
42
43         eval $COMMAND1 &
44         eval $COMMAND2 &
45         eval $COMMAND3 &
46         eval $COMMAND4 &
47         # Wait for all commands to finish
48         wait
49
50         # A little progress report
51         echo "Run␣$i␣done."
52     done
53
54 done
55
56 # Restore cfq scheduler
57 echo "cfq" | sudo tee $DEVICE
```

Listing 3: stats.py

```
1  """
2  stats.py
3
4  Process the data produced by timer.sh by calculating the
5  medians, max values and min values for each scheduler.
6
7  Author: Lennart Jern (ens16ljn@cs.umu.se)
8  """
9
10 import pandas as pd
11 import re
```

```

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")


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 4: 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"

/**
 * parse_arguments - parse command line arguments into SearchData.
 * The SearchData should be freed by calling SearchData_delete when
 * you are done.
 * @param  argc -- number of arguments
 * @param  argv -- array of arguments
 * @return      SearchData containing: dirs to look in, needle to look
 *                    for, file type and number of threads.
 */
SearchData *parse_arguments(int argc, char *argv[]) {
    SearchData *s_data;
    char *usage = "Usage: ./mfind [-t {d|f|l}] [-p nrthr] start1 [start2 ...] name";

    // No point to continue if there are less than 3 args
    if (argc < 3) {
        fprintf(stderr, "%s\n", usage);
        exit(EXIT_FAILURE);
    }

    s_data = malloc(sizeof(SearchData));
    if (s_data == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    // Set num_threads and type from given arguments
    if (set_flags(s_data, argc, argv) != 0) {
        fprintf(stderr, "%s\n", usage);
        exit(EXIT_FAILURE);
    }
    // Make sure the flags are safe
    check_flags(s_data);

    // Check that there is at least one dir to look in and a name to look for.
    if (optind >= argc -1) {
        fprintf(stderr, "%s\n", usage);
        free(s_data);
        exit(EXIT_FAILURE);
    }

    add_dirs_and_needle(s_data, argv, optind, argc-1);

    return s_data;
}

/**
 * set_flags - parse the arguments and add corresponding search data
 * @param s_data -- SearchData to add info to
```

```
 * @param argc   -- number of arguments
 * @param argv   -- array of arguments
 * @return          0 if everything went well, -1 otherwise.
 */
int set_flags(SearchData *s_data, int argc, char *argv[]) {
    char *optstr = "t:p:";
    int opt;
    char type = '\0';
    int num_threads = 1;

    // Parse flags
    while ((opt = getopt(argc, argv, optstr)) != -1) {
        char *end;
        switch (opt) {
        case 't':
            type = *optarg;
            break;
        case 'p':
            errno = 0;
            num_threads = strtol(optarg, &end, 10);
            if (errno != 0) {
                perror("strtol");
            }
            break;
        default:
            return -1;
        }
    }

    s_data->num_threads = num_threads;
    s_data->type = type;
    return 0;
}

/**
 * check_flags - make sure the flags (num_threads and type) are correct and safe
 * @param s_data -- SearchData to check.
 */
void check_flags(SearchData *s_data) {
    int num_threads = s_data->num_threads;
    char type = s_data->type;

    if (num_threads < 1) {
        fprintf(stderr, "Number␣of␣threads␣must␣be␣more␣than␣0!\n");
        exit(EXIT_FAILURE);
    } else if (num_threads > MAXTHREADS) {
        fprintf(stderr, "Too␣many␣threads!␣Resetting␣to␣1.\n");
        num_threads = 1;
    }
    if (type != 'd' && type != 'f' && type != 'l' && type != '\0') {
        fprintf(stderr, "Type␣must␣be␣d|f|l.\n");
        exit(EXIT_FAILURE);
    }

    s_data->num_threads = num_threads;
    s_data->type = type;
}

/**
 * add_dirs_and_needle - parse starting dirs and needle from arguments and
 * add them to the search data.
 * @param s_data -- search data to add to
```

```
 * @param argv   -- array of arguments
 * @param first  -- index of first directory in argv
 * @param last   -- index of last element in argv. This should be the needle.
 */
void add_dirs_and_needle(SearchData *s_data, char *argv[], int first, int last) {
    LinkedList *dirs = List_init();
    char *needle;
    int num_dirs = last - first;

    // Add all starting directories to the list of dirs.
    for (int i = 0; i < num_dirs; i++) {
        char *new_dir = malloc(strlen(argv[first+i])+1);
        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 5: 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 6: list.c

```
/**
 * File: list.c
 * Author: Lennart Jern - ens16ljn@cs.umu.se
 *
 * A simple implementation of a linked list.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "list.h"

/**
 * List_init
 * Create and initialize a LinkedList.
 *
 * @return  pointer to list
 */
LinkedList *List_init(void) {
    LinkedList *lst = calloc(1, sizeof(LinkedList));
    if (lst == NULL) {
        fprintf(stderr, "Allocation␣of␣memory␣for␣linked␣list␣failed\n");
        exit(EXIT_FAILURE);
    }
    return lst;
}

/**
 * Appends an element with the specified value to the list
 * @param  l:     List to append to
 * @param  value: Pointer to value to be added
 * @return       1 on success, otherwise 0.
 */
int List_append(LinkedList *l, void *value) {

    Node *new_node = calloc(1, sizeof(Node));
    if (new_node == NULL) {
        fprintf(stderr, "Failed␣to␣allocate␣memory␣for␣new␣node.\n");
        return 0;
    }
    new_node->value = value;
    new_node->next = NULL;

    if (l->first == NULL) {
        l->first = new_node;
        return 1;
    }
    Node *node_ptr = l->first;

    while (node_ptr->next != NULL) {
        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
```

```
            before_smallest->next = smallest->next;
            if (last_sorted == NULL) {
                // place it first
                smallest->next = lst->first;
                lst->first = smallest;
            } else {
                // place it after last_sorted
                smallest->next = last_sorted->next;
                last_sorted->next = smallest;
            }
        }
        // update last_sorted and boundary
        last_sorted = smallest;
        boundary = smallest->next;
    }
}

/**
 * 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)) {
    Node *node = lst->first;
    Node *prev = node;
    if (node == NULL) {
        return 0;
    }
    if (node->next == NULL) { // Only one element in list
        delete_value(node->value);
        free(node);
        lst->first = NULL;
        return 1;
    }
    while (node->next != NULL) {
        prev = node;
        node = node->next;
    }
    prev->next = NULL;
    delete_value(node->value);
    free(node);
    return 1;
}

/**
 * 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)) {
    if (lst) {
        while(List_remove(lst, delete_value)) ;
        free(lst);
    }
}
```

```
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 7: 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 /**
```

```
 * 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 8: 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
```

```
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
```

Listing 9: generate_test_files.sh

```
1  #!/bin/bash
2
3  # File: generate_test_files.sh
4  # Author: Lennart Jern - ens16ljn@cs.umu.se
5  #
6  # Generate a file tree to do tests on.
7
8  # Starting directory
9  # START=/run/media/lennart/KINGSTON
10 START=/run/media/lennart/Verbatim
11
12 # Move to correct directory/device
13 cd $START
14
15 # Remove tree if existent
16 rm -r test_files
17
18 # Create directory to hold all test files
19 mkdir test_files
20 cd test_files
21
22 # Split up the files between a few directories
23 for DIR in a b c d e; do
24     mkdir $DIR
25     cd $DIR
26     # Create empy files
27     for F in $(seq 1 20); do
28         touch $F
29     done
30
31     for D in f g h i j; do
32         mkdir $D
33         cd $D
34     done
35
36     cd "$START/test_files"
37 done
38
39 # Big files
40 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
```