CSE410 Operating Systems, Spring 2014

Laboratory 3: Multithreaded File Searching Program

Due: 23:59, Wednesday, April 23, 2014

1 Overview and Background

In this lab assignment you will learn about how to access and traverse the Unix/Linux file system from a program. You will do so using functions contained in `dirent.h` and `sys/stat.h`. You will also gain additional experience with multithreaded programming and the use of semaphores by implementing a multithreaded file searching program with a bounded buffer (which is used to implement a queue of tasks for threads). The program will be a simplified, but multithreaded, version of the `find` command in Linux. For a target directory and all its subdirectories, the program will list all files satifiying a set of search criteria (name and size constraints). Indeed, the program searches the entire directory tree under the target directory.

You may write your solution in either C or C++. The skeleton code provided to you is written in C++ [link] . Using the skeleton code is not mandatory, however.

IMPORTANT NOTE: You should develop and test your code on machines in 3353 EB, **NOT** on the department servers.

1.1 Accessing Directories using `dirent.h`

In this lab assignment, you will use the following three functions defined in `dirent.h` to access the Lunix file system.

- `opendir(const char *dirname)` opens the directory located at `dirname`. If it successfully opens `dirname`, a directory object (`DIR`) associated with the directory is created and a pointer to the object is returned. The directory object is used for the subsequent operations. If the open fails, `NULL` is returned and a corresponding `errno` is generated.
- `readdir(DIR dir_obj)` returns a pointer to the next directory entry of type (`struct dirent`) or `NULL` when there is no more entry. It returns `NULL` if there is an error and a corresponding `errno` is generated.
- `closedir(DIR dir_obj)` closes and frees the directory object. It returns 0 on success or -1 on failure. A corresponding `errno` is generated on failure.

The following is a simple example of directory access. You can save it as a filename.cpp file and use `g++ filename.cpp` to compile it. It performs the same functionality as the command `ls -a`: it opens the current directory and lists everything in the directory.

```
#include <iostream>
#include <string.h>
#include <dirent.h> // struct dirent is defined in this header
#include <errno.h>

using namespace std;

int main()
{
    string loc = ".";

    DIR *dir;
    struct dirent *entry;

    errno = 0;
    dir = opendir(loc.c_str()); // string to c string conversion
    if(errno != 0)
    {
        cout << "opendir error. " << strerror(errno) << endl;
    }
    else if(dir)
    {
        errno = 0;
        while((entry = readdir(dir)) != NULL)
        {
            cout << entry->d_name << "  ";
        }
        if(errno != 0)
        {
            cout << "readdir error. " << strerror(errno) << endl;
        }
        cout << endl;
    }
    closedir(dir);
}
```

Please refer to man pages for more information.
`opendir()`: http://man7.org/linux/man-pages/man3/opendir.3.html

`readdir()`: http://man7.org/linux/man-pages/man3/readdir.3.html
`closedir()`: http://man7.org/linux/man-pages/man3/closedir.3.html

1.1.1 `errno`

The variable `errno` is a global integer variable defined in `errno.h`. `errno` is set by libraries and system calls for indicating errors. Before invoking a function or system call, we need to reset `errno` to 0 and if the current value of `errno` should be preserved, it must be saved as shown below.

```
int errno1, errno2; // local errno for saving the value of errno
errno = 0;
system_call1();
errno1 = errno;

errno = 0;
system_call2()
errno2 = errno;
```

In the directory access example, the `errno` is handled (displayed) before next function call, so it does not need to be saved. The function `strerror` interprets the value of `errno` to a message as a string. Try this: change `loc` to a nonexistent directory or to a directory to which you do not have access, and see what happens.

Please refer to the `errno` man page for more information.
http://man7.org/linux/man-pages/man3/errno.3.html

1.1.2 `struct dirent`

In Linux, the type for directory entry `struct dirent` is defined as

```
struct dirent {
    ino_t          d_ino;       /* inode number */
    off_t          d_off;       /* offset to the next dirent */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file; not supported
                                   by all file system types */
    char           d_name[256]; /* filename */
};
```

We will discuss inodes, which contain metainformation about files, in class. For this lab assignment, we only need to be concerned with two fields: `d_name` and `d_type`. In the above example, we have used `entry->d_name` to display the filename. The `d_type` defines the *type* of the entry. If its value is 0x4, it is a directory (folder), which can contain other files. Otherwise, it is a type of file that does not contain other files. A macro `IS_DIRECTORY` is defined in the skeleton. You can check if an entry is a directory by using `if(entry->d_type == IS_DIRECTORY)`. In this lab, we only care if the entry is a directory or not; if it is a directory, it needs to be searched.

1.2 Search Criteria

As noted above, your program will not only search all files in the target directory, but will also search the entire directory tree below the target. Your program will list all files that satisfy the search criteria. Here, we specify only three search criteria (the `find` command supports many more).

- *file name*: file name containing certain pattern.
- *max size*: file size is smaller than max size (in bytes).
- *min size*: file size is larger than max size (in bytes).

These criteria can be specified on the command line when invoking the program. There are corresponding arguments in `find` command. For example, the two invocations:

```
lab3 ~/ -name lab2 -min_size 100 -max_size 10000
find ~/ -name '*lab2*' -size +100c -size -10000c
```

should print the same set of files (although the order may differ).

Note that the `-name` argument for the `find` command is defined in terms of a regular expression. In this lab assignment, you can use the `string::find` method to determine whether the file name contains the `-name` pattern. The parsing of the search criteria is provided for you in the skeleton. Please refer to Section 3 for more information.

Please refer to the `find` man page for more information.
http://man7.org/linux/man-pages/man1/find.1.html

1.2.1 `sys/stat.h`

You will use functions defined in `sys/stat.h` to obtain file status information such as the file size. The function `stat(const char *pathname, struct stat *buf)` obtains information about a file located at `pathname` and returns a pointer to the buffer of type `struct stat`. The stat structure contains many fields, but in this lab assignment, we care about the size only. We can obtain the size of a file entry by using the following code.

```
string full_path;       // path to a file
struct stat filestatus; // the buffer
int ret = stat(full_path.c_str(), &filestatus);
if(ret == 0)
{
    cout << filestatus.st_size << endl;
}
```

Please refer to `sys/stat.h` man page for more information.
http://man7.org/linux/man-pages/man2/stat.2.html

1.3 Program Structure

For a single-threaded program, intuitively, we can create a function that opens a directory and processes the entries in the directory. The program starts with processing the target directory. For each entry in that directory, regardless of whether the entry is a file or a directory, the program output its path if it satisfies the search criteria. If the entry is a directory, the program *recursively* invokes the same function to process the subdirectory. The recursion terminates at directories that do not contain any subdirectories.

For a multithreaded search, although the functionality is similar, the approach needs to be slightly different. We suggest you use a queue of directory names, corresponding to search *tasks*. Initially, the target directory is placed into the queue. The threads then begin taking entries (tasks) from the queue and processing them. For each entry in the corresponding directory, the thread outputs its path if it satisfies the search criteria. If the entry is a directory, the thread places that directory's path into the queue. The threads keep processing directories (queue entries) until the queue is empty and all threads are idle. Pseudocode for this functionality is provided as follows. Note that shared data needs to be protected by mutexes (not shown), in order to prevent race conditions.

1.4 Using an Unbounded Queue (Required Functionality)

Please note the relationship between this problem and the producer-consumer scenarios described in the course notes (`2-4-concurrency.pdf`, especially slides 33 - 41). When a thread is retrieving items from the queue, it acts as a consumer. When a thread is placing items into the queue, the thread is a producer. Therefore, the access of the queue needs to be synchronized using semaphores.

The queue used in the pseudocode above is assumed to be unbounded, in which case the queue needs to be protected for mutual exclusion and there needs to be a semaphore indicating the number of items in the queue (so a consumer will block when the queue is empty). However, we do not need a semaphore indicating when the queue is full, since it can never be full.

```
Input: start_dir, Output: files that satisfy the search criteria;
queue = {start_dir}
Function search()
    while True do
        if queue is empty and all threads are idle then
            break; // We're done.
        else if queue is empty then
            continue; // Wait for other threads to complete and add to the queue
        else
            // queue is not empty
            curr_dir = get a directory from queue
        end
        open curr_dir;
        for each entry in curr_dir do
            if entry is a directory then
                if entry is accessible then
                    if entry satisfies the search criteria then
                        output this entry
                else
                    output a message indicating permission denied
                end
                put the entry into the queue
            else if entry is not a directory then
                if entry satisfies the search criteria then
                    output this entry
        end
    end
```

In this assignment, you can assume the queue is unbounded (actually, in the skeleton code we simply initialize it to a very large size). For extra credit (see below), you can handle the situation where the queue is bounded.

1.5 Using a Bounded Queue (Extra Credit)

For extra credit, you can solve the problem using a bounded (finite) buffer. Please note that doing so is not as simple as adding another semaphore to indicate when the buffer is full. Consider the following scenario: Directory A has 2 subdirectories B and C. Both B and C have 10 subdirectories. Suppose the queue size is only 2 and there are two threads. The first thread processes directory A and fills the queue with directories B and C. The first thread then retrieves B from the queue and processes it, while the second thread retrieves and processes C (or vice versa). Both B and C have 10 directories to be placed into the queue, but the queue size is full after only 2 insertions. Both B and C are still acting as producers, and there is no consumer to remove entries from the queue. The program will hang.

If you decide to do this extra credit part of the assignment, you should write a short report describing how you solved the problem and including performance results for different sized queues and numbers of threads. Your solution should work even if the queue size is 1.

2 Requirements, Deliverables and Grading

You are required to implement a multithreaded file searching program that list all files and directories satisfying the search criteria in a target directory and the directory tree below it. The user must specify a target directory to start searching. The user can set three search criteria: file name, max size and min size. The user can also specify the number of threads to be used in this program and the size of the queue, `queue_size` (by default it is unbounded).

Initially, the target directory should be placed into a queue. The main program creates the specified number of threads, each of which repeatedly retrieves directories (search tasks) from the queue and and processes them as described in Section 1.3. Specifically, for each entry in a directory, the thread should output its path if it satisfies the search criteria, regardless of whether the entry is a file or a directory. If the entry is a directory, the thread places the directory's pathname into the queue. The threads continue retrieving and processing directories until the queue is empty and all threads are idle.

Since the queue is shared by multiple threads, access to it needs to be protected and synchronized. Any other shared data must be protected by mutexes to avoid race condition.

Project Requirements:

- Implement a multithreaded file searching program that lists all files and all its subdirectories satisfying the search criteria in a target directory. The criteria includes name pattern, max size and min size. (The criteria are parsed by the skeleton code provided to you and is stored in the variable `pattern`, `max_size` and `min_size`.)

- The program should create **N** threads. Each thread gets a directory from the queue until all directories are processed. The number of threads **N** is parsed by the skeleton code and are stored in the variable `thread_count`.

- Use mutexes and semaphores to prevent race conditions for variables, including the queue, shared between threads.

- Conduct experiments with different combinations of target directory **D** and thread counts **N** and compare the performance of your program and the built-in `find` command. You can use `time find args` to measure `find`'s running time. You should run the tests for at least the following parameters: **D** = root directory (/), your home directory (˜); **N** = 1, 5, 10, 20. For statistical accuracy, it is probably a good idea to run each test at least 3 times. To automate the process, you might choose to write a simple shell script. An example shell script, runall.sh, is provided with the skeleton. To execute it simply type: "sh runall.sh" at the command line. (You may choose to make a more complex shell script, or use a different shell, in order to automate collection of data. You can find various tutorials on shell programming on the web.) Report the running time of each configuration and write a short paragraph explain/describe your results in the README file. In the skeleton code, you will find examples of the gettimeofday() system call. Use the following format to report your results. Two empty tables are provided in the README file.

Extra Credit:

- The queue should be bounded by **M**. As above, access of the queue should be synchronized using mutexes and semaphores. The queue size **M** is parsed by the skeleton code and are stored in the variable `queue_size`.

- The situation where the queue is full needs to be identified with a semaphore. Moreover, the logic of your program needs to handle this situation without hanging, while taking advantage of parallelism and concurrency among threads. Your program should work even if the size of the queue is 1.

- Write a short report (to be included in the README file) describing how you solved the problem and presenting results for different values of **M** and **N**. You should run the tests for at least the following parameters: **D** = root directory (/), your home directory (˜); **N** = 1, 5, 10, 20; **M** = 1, 100, 10000. BE SURE to indicate in the README file that you have addressed the extra credit part of the assignment.

2.1 Project Guidelines

Working alone or in pairs. You may work on this assignment individually or in pairs (not in groups of 3 or more, however). If you prefer to work in a pair, **both** students must submit a copy of the solution and identify their MSU NetID of both students in a README file. If you prefer to work individually, please clearly state that you are working individually and include your MSU NetID in the README file.

Programming Language. You may implement this project using `C` or `C++`. The skeleton code is written in `C++`. Please clearly state the command to compile your project submission in your README file.

Testing your code. Each Linux distribution might be slightly different. It is the students' responsibility to make sure that the lab submissions compile on *at least one* of the following machines in 3353 EB: `carl`, `ned`, `marge` or `skinner`. A statement must be provided in the README file's header describing on which manchines the code has been tested. You will **not** be awarded any credit if your lab submission does not compile on any of those machines.

2.2 Deadline and Deliverables

This lab (including the Extra Credit component) is due no later than 23:59 (11:59 PM) on Wednesday, April 23. No late submission will be accepted. You must submit your lab using the *handin* utility. (`https://secure.cse.msu.edu/handin/`) Your submission should include:

All source files. Submit all source files in your project directory. If you use the skeleton code, submit all the files, even if some files are not modified.

A makefile. Include a makefile to compile your code. A makefile is provided in the skeleton code, but you may choose to modify it.

A README file. The README file should include a header, sample output and any relevant comments. Please provide the following items in header of the README: whether you are working individually or in a pair, the MSU NetIDs of the submitting students, a list of machines on which you have compiled your code, the command used to compile your code. Two sample README file headers are as follows:

```
Student NetID: alice999, I am working with bob99999.
Compilation tested on: ned, skinner, marge ...
Command for compile: gcc proj1.c -o proj1

Student NetID: doejohnQ, I worked on this project individually.
Compilation tested on: ned
Command for compile: make
```

Your README file should also include performance results as described above and example output from your program, which will aid the TA in debugging if he cannot reproduce your results. You are also encouraged to include any relevant comments in the README file. A sample README file is also included in the skeleton code.

As noted above, if you do the extra credit component, your README file should contain the report and performance results for that part of the assignment, as well.

2.3 Grading

This base project is worth 100 points. The extra credit component is worth 30 points. You will not be awarded any points if your submission does not compile. The grading rubric is as follows:

```
General requirements: 5 points
_____  2 pts: Coding standard, comments ... etc
_____  1 pts: README file
_____  2 pts: Descriptive messages/outputs

Multithreading and Synchronization: 45 points
_____ 10 pts: Thread creation, join and exit
_____ 25 pts: Terminate the program properly.
              (Queue empty, all threads idle)
_____ 10 pts: The access to the queue is synchronized by semaphores

Directory Searching: 35 points
_____ 10 pts: All search criteria work correctly
          -3 pt: -name does not work
          -1 pt: -max_size does not work
          -1 pt: -max_size does not work
          (gets 5 pt for listing the paths, but not filtering using
           any criteria)
_____ 25 pts: Searching the files correctly
          -0 pt: if (99\%, 100] results are the same as find
          -5 pt: if (75\%, 99\] results are the same as find
          -10 pt: if (50\%, 75\%] results are the same as find
          -25 pt: if [0\%, 50\%] results are the same as find

Error checking: 5 points
_____  5 pts: Check the return values and errno for the functions
         -1 pt: for each items not checked.
```

```
Experiments: 10 points:
_____ 10 pts: Conduct the experiments with different parameters and
              compare with find and explain the performance difference

Extra Credit: 30 points:
_____ 30 pts: Up to 30 points depending on the robustness of your solution
              (handling different queue sizes wile maintaining thread
               parallelism)
```

3 Skeleton Code and Test File

To assist you in this assignment, skeleton code comprising five files is provided. The first is an example README file, as described earlier. The second is a makefile; executing the command `make` will generate the executable `lab3`. The main function for this lab assignment is in `lab3.cpp`. You will develop your threaded search function in this file.

Several supporting functions and variables that will be of use to you are provided in `utilities.cpp`. `parse_argv` parses the arguments and construct several variables for you.

- `thread_count`: The number of threads specified by the user, default is 5
- `start_dir`: The starting directory for the searching. Must be provided.
- `queue_size`: Size of the bounded queue, default is unbounded.
- `pattern`: The pattern to be searched in the file name.
- `max_size`: file size is smaller than max size (in bytes).
- `min_size`: file size is larger than min size (in bytes).

Please refer to `utilities.cpp` for more information. A simple example shell script runall.sh is also included to help you conduct your experiments. Please note that you are not required to use the skeleton code.

4 Examples

Follows are examples of output from the program. Your output may differ. When there is double or confusion of the result, you may refer to the output of Linux built-in `find` command. The following examples are generated on carl in 3353 EB.

1.
```
>./lab3 ./
Task queue size:   unbounded
Number of threads: 5
Searching for:     anything
Max size:          2147483647
Min size:          0

./
Thread 0 started.
Thread 1 started.
Thread 4 started.
Thread 3 started.
Thread 2 started.
./Makefile
./README
./lab3.o
./lab3.cpp
./utilities.cpp
./runall.sh
./lab3
Thread 0 terminated.
Thread 2 terminated.
Thread 3 terminated.
Thread 1 terminated.
Thread 4 terminated.
Searching  with 5 threads and unbounded queue took 0.001281s.
```

2.
```
>./lab3 ./ -name README
Task queue size:   unbounded
Number of threads: 5
Searching for file names containing: README
Max size:          2147483647
Min size:          0

Thread 0 started.
Thread 1 started.
Thread 3 started.
Thread 2 started.
Thread 4 started.
./README
Thread 1 terminated.
Thread 2 terminated.
Thread 0 terminated.
Thread 4 terminated.
Thread 3 terminated.
Searching README with 5 threads and unbounded queue took 0.002062s.
```

3.
```
>./lab3 /bin
Task queue size:   unbounded
Number of threads: 5
Searching for:     anything
Max size:          2147483647
Min size:          0

/bin/
Thread 0 started.
Thread 1 started.
Thread 3 started.
Thread 2 started.
Thread 4 started.
/bin/bzgrep
/bin/ntfsls
/bin/bzexe
......
......
......
......
......
......
/bin/ntfscat
/bin/lsblk
/bin/zsh4
Thread 2 terminated.
Thread 4 terminated.
Thread 1 terminated.
Thread 0 terminated.
Thread 3 terminated.
Searching  with 5 threads and unbounded queue took 0.003877s.
```

4.
```
>./lab3 /123123123
Task queue size:   unbounded
Number of threads: 5
Searching for:     anything
Max size:          2147483647
Min size:          0
```

```
/123123123/
Thread 0 started.
Thread 1 started.
No such file or directory
Thread 1 terminated.
Thread 3 started.
Thread 3 terminated.
Thread 0 terminated.
Thread 2 started.
Thread 2 terminated.
Thread 4 started.
Thread 4 terminated.
Searching  with 5 threads and unbounded queue took 0.000584s.
```

5.
```
>./lab3 ~ -max_size 100
Task queue size:   unbounded
Number of threads: 5
Searching for:     anything
Max size:          100
Min size:          0

Thread 0 started.
Thread 1 started.
Thread 3 started.
Thread 4 started.
Thread 2 started.
/user/liuchinj/.dmrc
/user/liuchinj/.exrc
/user/liuchinj/.forward
/user/liuchinj/.imaprc
/user/liuchinj/.mailboxlist
/user/liuchinj/.mcoprc
/user/liuchinj/.msgsrc
/user/liuchinj/.snapshot
/user/liuchinj/.xmodmaprc
/user/liuchinj/dead.letter
/user/liuchinj/.CFUserTextEncoding
/user/liuchinj/.h
/user/liuchinj/.esd_auth
/user/liuchinj/.gksu.lock
/user/liuchinj/.results.txt.crc
```

```
/user/liuchinj/.lesshst
Thread 3 terminated.
Thread 0 terminated.
Thread 2 terminated.
Thread 1 terminated.
Thread 4 terminated.
Searching  with 5 threads and unbounded queue took 0.030986s.
```

6.
```
>./lab3 ~ -min_size 1000
Task queue size:   unbounded
Number of threads: 5
Searching for:     anything
Max size:          2147483647
Min size:          1000

Thread 0 started.
Thread 1 started.
Thread 3 started.
Thread 4 started.
Thread 2 started.
/user/liuchinj/mail
/user/liuchinj/.ICEauthority
/user/liuchinj/.OWdefaults
/user/liuchinj/.xsession-errors
/user/liuchinj/.Xdefaults
/user/liuchinj/.dtprofile
/user/liuchinj/.emacs
/user/liuchinj/.fvwm2rc
/user/liuchinj/.fvwmrc
/user/liuchinj/.history
/user/liuchinj/.login-personal
/user/liuchinj/.mailrc
/user/liuchinj/.mwmrc
/user/liuchinj/.openwin-init
/user/liuchinj/.openwin-menu-cps
/user/liuchinj/.openwin-menu-programs
/user/liuchinj/.viminfo
/user/liuchinj/.twmrc
/user/liuchinj/.snapshot
/user/liuchinj/.xinitrc
/user/liuchinj/._Library
/user/liuchinj/.DS_Store
```

```
/user/liuchinj/._.TemporaryItems
/user/liuchinj/._.cups
/user/liuchinj/.vimrc
/user/liuchinj/.edgar.py.swp
/user/liuchinj/.pig_history
/user/liuchinj/.xsession-errors.old
/user/liuchinj/lab2_file
/user/liuchinj/.mysql_history
/user/liuchinj/results.txt
/user/liuchinj/.hivehistory
/user/liuchinj/a_million_and_a_hundred_As
/user/liuchinj/pig_1396087673920.log
/user/liuchinj/.results.swp
Thread 2 terminated.
Thread 0 terminated.
Thread 3 terminated.
Thread 4 terminated.
Thread 1 terminated.
Searching  with 5 threads and unbounded queue took 0.001801s.
```

7.
```
>./lab3 ~ -min_size 10000
Task queue size:   unbounded
Number of threads: 5
Searching for:     anything
Max size:          2147483647
Min size:          10000

Thread 0 started.
Thread 1 started.
Thread 2 started.
Thread 4 started.
Thread 3 started.
/user/liuchinj/.ICEauthority
/user/liuchinj/.xsession-errors
/user/liuchinj/.fvwm2rc
/user/liuchinj/.fvwmrc
/user/liuchinj/.viminfo
/user/liuchinj/.snapshot
/user/liuchinj/.DS_Store
/user/liuchinj/.edgar.py.swp
/user/liuchinj/.pig_history
/user/liuchinj/.xsession-errors.old
```

```
/user/liuchinj/lab2_file
/user/liuchinj/.mysql_history
/user/liuchinj/.hivehistory
/user/liuchinj/a_million_and_a_hundred_As
/user/liuchinj/.results.swp
Thread 0 terminated.
Thread 2 terminated.
Thread 1 terminated.
Thread 3 terminated.
Thread 4 terminated.
```

8.
```
>./lab3 ~ -max_size 10000 -min_size 1000
Task queue size:   unbounded
Number of threads: 5
Searching for:     anything
Max size:          10000
Min size:          1000

Thread 0 started.
Thread 2 started.
Thread 1 started.
Thread 3 started.
Thread 4 started.
/user/liuchinj/mail
/user/liuchinj/.OWdefaults
/user/liuchinj/.Xdefaults
/user/liuchinj/.dtprofile
/user/liuchinj/.emacs
/user/liuchinj/.history
/user/liuchinj/.login-personal
/user/liuchinj/.mailrc
/user/liuchinj/.mwmrc
/user/liuchinj/.openwin-init
/user/liuchinj/.openwin-menu-cps
/user/liuchinj/.openwin-menu-programs
/user/liuchinj/.twmrc
/user/liuchinj/.snapshot
/user/liuchinj/.xinitrc
/user/liuchinj/._Library
/user/liuchinj/._.TemporaryItems
/user/liuchinj/._.cups
/user/liuchinj/.vimrc
```

```
/user/liuchinj/results.txt
/user/liuchinj/pig_1396087673920.log
Thread 2 terminated.
Thread 3 terminated.
Thread 0 terminated.
Thread 4 terminated.
Thread 1 terminated.
Searching  with 5 threads and unbounded queue took 0.019558s.
```

9.
```
>./lab3 ~ -max_size 10000 -min_size 1000 -name mail
Task queue size:   unbounded
Number of threads: 5
Searching for file names containing: mail
Max size:          10000
Min size:          1000

Thread 0 started.
Thread 1 started.
Thread 4 started.
Thread 3 started.
Thread 2 started.
/user/liuchinj/mail
/user/liuchinj/.mailrc
Thread 2 terminated.
Thread 1 terminated.
Thread 3 terminated.
Thread 0 terminated.
Thread 4 terminated.
Searching mail with 5 threads and unbounded queue took 0.001885s.
```

10.
```
D = {/user/liuchinj/}
N        1              5               10              20              find
time     7.740          0.2471          0.1856          0.1813          1.57

D = {/usr}
N        1              5               10              20              find
time     61.32          29.75           18.92           10.29           45.68
// The time might vary significantly depend on the machine. The result might
// be different at different time. Performance will not be graded.
```