**UMEÅ UNIVERSITY**
**Department of Computing Science**
**Assignment 3 report**

**October 15, 2021**

5DV088 - Assignment 3, v1.0

# C programming and Unix, Autumn 2021, 7.5 Credits

du implementation with multi-threading

| | |
|---|---|
| **Name** | Elias Olofsson |
| **user@cs.umu.se** | tfy17eon@cs.umu.se |

**Teacher**
Mikael Rännar

**Graders**
Jonas Ernerstedt, Elias Häreskog, Oscar Kamf

# Contents

# 1 Introduction

In this lab we have created a bare-bones, multi-threaded version of the GNU du utility, used to recursively estimate disk usage of files and folders on a system. While such a program is useful to acquire information and get an overview on large files and folders with many sub-directories, this lab has mainly been an exercise in writing a multi-threaded program, while practicing how to navigate through directories and get information on files in C.

# 2 Thread safety

My final C-program `mdu.c` uses supplementary data types implemented in `list.c` and `list.h`, and can conveniently be compiled to an executable `mdu` using the included `Makefile`. While the program will run in a single thread if no options are specified at program execution, the power in this program comes when multi-threading is enabled, using the optional `-j` flag with the associated number of threads to use. This allows the program to have multiple streams of instructions executing concurrently, which will on a multi-core system typically make the program complete its tasks in less time compared to a typical single-threaded version of the same program.

However, since multiple things can be worked on at the same time, we need to make sure that the program behaviour is predictable and consistent when shared data structures are accessed and manipulated by multiple threads. This is what we mean by thread safety, i.e. that the program behaviour and progression is not affected by unintended interactions between the working threads. To achieve this and ensure a controlled program flow and access pattern of common resources, synchronization is key.

On shared data structures, it is important that only one thread are accessing the common resource at a time. Threads thus need to synchronize and communicate amongst each other such that they can "take turns" on accessing critical sections of the code. In my program `mdu.c`, the important shared data structures are a common task queue and an array of results. The task queue is a to-do list of things to do next, while the array contains the total number of 512B blocks allocated on disk for each of the specified files or folders given at program execution as arguments. Since any thread will potentially have to access any of these common resources, it is most effective to give each of these resources their own personal *mutex*, short for "mutual exclusion". Such a device will, if used properly, guarantee that a thread has exclusive access to a section of the code or some shared data structure. At

any time when any thread wants to access any of these shared resources, the corresponding mutex must first be locked before access and subsequently unlocked after access is completed. In our case, having separate mutexes for the queue and for each entry in the array will minimize the likelihood of threads having to wait on each other, compared to if we simply used a single common mutex for all of these shared resources together.

Furthermore, since the each of the threads will be able to create and add new tasks to the queue, we need a way to signal a waiting processes that more work has arrived. This is achieved using a *condition variable*. The forever repeating main loop of the thread worker algorithm is that first, the thread takes the mutex for the queue. Now when it has exclusive access to the queue, it removes the first item from the queue, releases the lock and carries out the task. However, if the task queue is empty and no work is available, instead of manually retrying and frequently checking the queue again for more tasks, we let the thread *wait* on a condition variable. This releases the mutex for the queue, such that other threads are able to access the queue, and puts the thread in a idle state were the execution is suspended. The thread will remain in this waiting state, without consuming CPU-time of the system, until the condition variable is signaled. If we let the threads signal the condition variable when a new task is added to the queue, we can conveniently wake up threads when more work is made available. Then, when the condition variable is signaled, the waiting thread wakes up and reacquires the mutex for the queue, dequeues the first task, releases the lock and carries out the task.

One last detail, how do we know when all work is done? Since more tasks can be created by each worker thread during the execution of a task, it is simply not enough to check if the list of tasks is empty to determine if all work is completed. The answer is to ensure that all tasks have been carried out and *no more tasks can be created*. The way I have chosen to implement this is to keep track of how many threads are currently waiting, using a simple integer, protected by the same mutex as for the queue. Before a thread starts to wait, I let this integer be incremented, and when the thread is awoken, I decrement it. Thus, when the task queue is empty and all threads but one are waiting, we know that all work is done and all of the threads can exit.

# 3 Performance analysis

How suitable a given program design or problem solution is to multi-threading can vary a lot. Some problems are inherently serial in nature and cannot

easily be split up into distinctly separate and parallel tasks. How scalable a program is to parallelism can be described by Ahmdahl's law, which gives the maximum parallel speed up compared to truly serial execution as

$$\text{maximum parallel speed up} = \frac{1}{f_s}, \tag{1}$$

where $f_s$ is the fraction of the execution time spent in code parts where the execution happens in ordinary serial fashion. Since any program will always have some parts which are not truly parallelizable, be it due to software design or inherent problem limitations, there will always be an upper limit on the maximum gain one can obtain by throwing more threads and CPU cores at the problem.

Furthermore, simply using more software threads on any given machine will also have diminishing returns beyond a certain point. This is due the fact that each thread takes up memory and has associated overheads for thread management and scheduling. Too many threads put a heavier burden on the system's performance than the gain achieved by using more threads. Also, one should remember that a given system can at any given time only execute as many instruction streams as there are hardware threads. When there are more software threads running on a machine than there are hardware threads/CPU cores, the scheduler is constantly switching between execution of the different software threads, which adds extra overhead and slows down the overall speed of program execution.

Additionally, I recorded the execution times of the program `mdu` using varying numbers of threads for the directory `/pkg/` on the `itchy` server at the Department of Computing Science, Umeå. The mean over 10 program runs at each thread count between 1 and 100 threads, in increments of 1 thread, can be seen in Fig.(1). Also in the same figure, one can see the mean parallel speed up gained by using more threads.

From this graph, one can discern the phenomena we were talking about previously. Since we reach a maximum speed up around $\sim 8.5$ while using 24 threads, even though `itchy` has 64 physical cores and 128 logical hardware threads, I would estimate using Ahmdahl's law that approximately $\sim 12\%$ of the execution time of this particular problem was effectively serial in nature. This is likely the main bottleneck which is holding the performance of this program back, even if more cores were available.

Also, one can note that after the peak speed up at 24 threads, there is a steady decline in the relative speedup with increasing number of threads. I would guess this is be due to the fact that we have already reached (or are close to) the limit of parallel speedup which Ahmdahl's law permits, but we take a performance hit each time we add more threads which adds overhead.

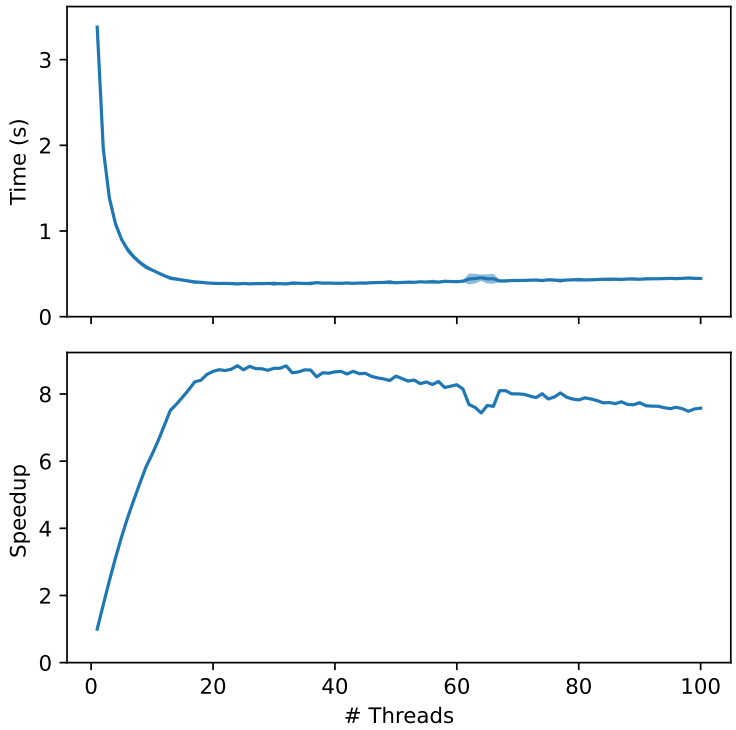


**Figure 1** – Top: Mean execution time recorded as a function of threads available to the program `mdu` with confidence interval of two standard deviations. Mean is calculated over 10 runs at each thread count between 1 and 100 threads, in increments of 1. Bottom: Mean relative parallel speed up compared to true serial execution, as a function of available threads to the program.

# 4 Discussion and reflection

This lab was quite a lot of fun to complete, but it also took me way much more time than I had anticipated. I was first struggling to figure out a good recursive algorithm which would also work together with the pre-implemented functions and methods existing to open and traverse directories. I believe I was possibly overthinking it, while also having misunderstood a bit how to the directory methods would behave and be used. However, once I figured out the main algorithm, the implementation of multi-threading came actually quite naturally. I was afraid that would take a lot more time and energy than it actually did.