Umeå University
Department of Computing Science

C Programming and Unix 7.5 p
5DV088

mfind

Submitted 2016-10-17
Author: Lorenz Gerber (dv15lgr@cs.umu.se lozger03@student.umu.se)
Instructor: Mikael Ränner / Filip Åberg / Jonathan Westin / Mattias Åsander

Contents

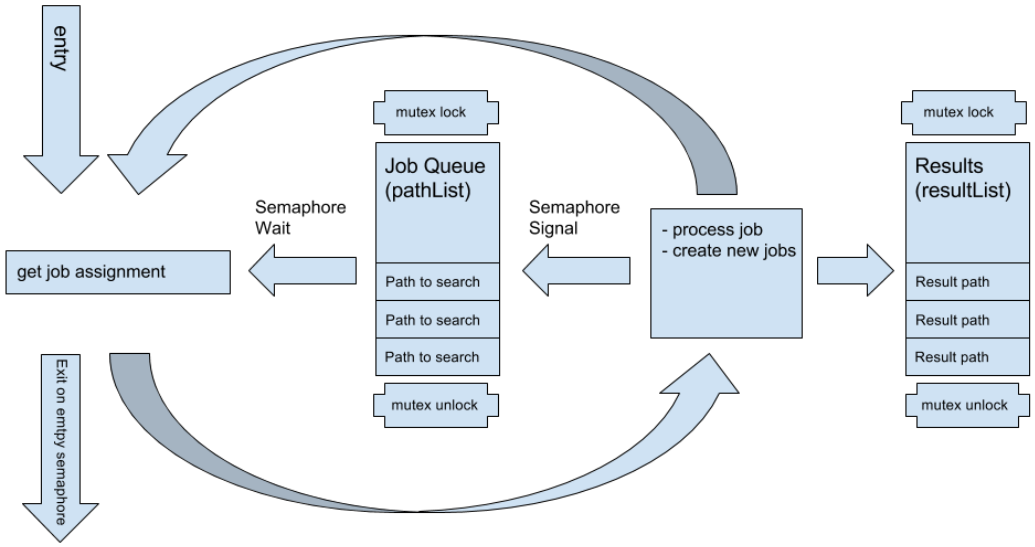

Figure 1: *This figure shows a schematic view of the mfind problem.*

1 *mfind* and Thread Safety

The scheme in figure 1 shows the most important aspects of the mfind system and how it was implemented here. It can be basically seen as a producer-consumer system, however, each thread can be both producer and consumer. The most important source of synchronization is a semaphore [2, chapter 15.8] which distributes the jobs to the threads. When there are no more jobs available, the thread subsequently terminates. The 'do while' loop looks futile on the first view, however, new jobs can be added after passing the 'no wait' semaphore until checking the value of the semaphore in the 'while' expression which will result in another round to catch a job.

The current construction is thread safe: The last thread in the loop can not leave until all jobs, also freshly self created are processed. In certain cases, this could lead to an uneven workload as threads can leave while it's still possible that new jobs are generated. This could be addressed by further synchronizing the threads, forexample with a mutex. That would however also affect performance for the most general cases. During testing the current code, it happend rarely that the distribution of tasks was not even between the different threads.

2 Performance Assessment

To assess the performance of the presented implementation, it was batched with the 'time' function on itchy.cs.umu.se. As instructed, 2 to 10 threads were run each 10 times on */pkg/comsol*. The results were written into a textfile, that was then imported to R [1], parsed and plotted as shown in figure 2.

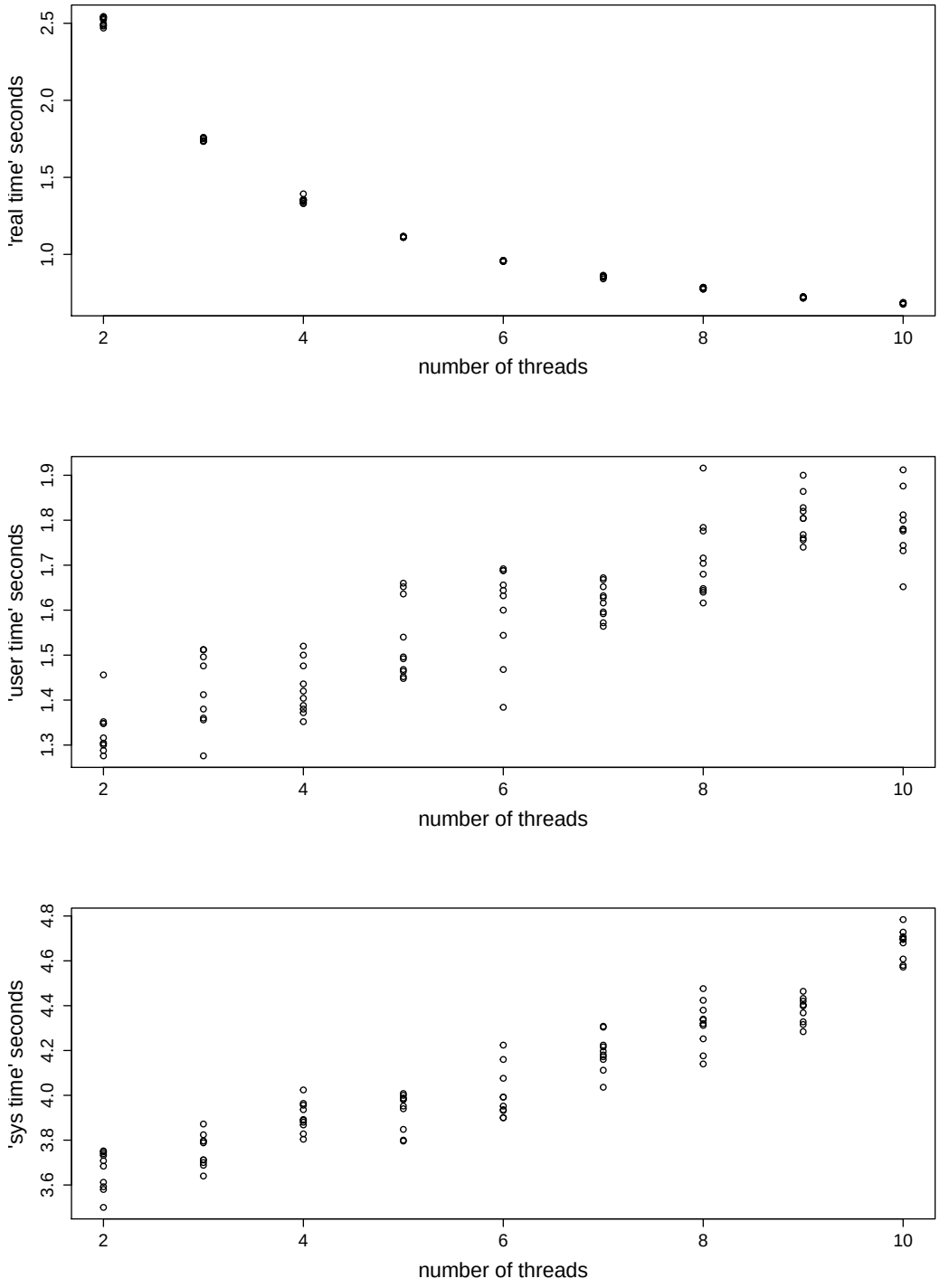

Figure 2: *This figure shows the execution time in relation to number of threads.*

It was chosen to plot real, user and system time separate. As it can be seen in uppermost graph of figure 2, the 'real' time decreases for increased number of cores. However, it is also obvious from the plot, that the line flattens out and would eventually start to increase again when using a too high number of threads. In single process experiments, it was estimated that the lowest point, hence the shortest run time is achieved at around 14 threads (not visible in the graph).

The middle graph shows 'user' time which increases steadily, the same is true for the lowermost graph that shows 'system' time. While real time shows the time which actually passes, 'system' and 'user' time account time for each thread used and sum it up. The time increase correlated with the higher number of threads for 'system' and 'user' time shows that more 'overhead' time is spend to coordinate the threads.

References

[1] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[2] W. Richard Stevens and Stephen Rago. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Boston, USA, 2013.