Mutual Exclusion Algorithms in Distributed Networks

Dan Vekhter
Niles North High School
8641 National Avenue
Niles, IL 60714

Alex Rasin
Deering High School
33 Oakland Street
Portland, ME 04103

Yuriy Brun
Holliston High School
142 Clinton Street
Hopkinton, MA 01748
Research conducted at Brandeis University Summer Odyssey Directed Research Project, 1996

ABSTRACT:
The problem of mutual exclusion is ever-present in distributed networks. Various algorithms have been suggested as a means to solve the problem. This paper discusses the efficiency of three such algorithms -- Test-and-Set bit, Dijkstra's, and Peterson's. The performance of each of the algorithms is dependent on the number of competing processes and the length of the remainder section. Our experimental results show which of the three algorithms are the most efficient to use depending on the size of the network and the number of remainder steps.

INTRODUCTION:

A distributed network is a system where one processor operates on several competing processes, or tasks, which require the use of the main processor's software and hardware resources. The tasks may be sent to the main processor from several different users of the same processor or from different computers.

When a task is waiting to use the shared resources, it is said to be in its "trying section." When the processor starts executing a task using the shared resources of the system, the process is said to enter its "critical section". It is in this section that competing processes can interfere with each other. When a job has finished its critical section, it enters its "remainder section," during which it does not use any of the shared resources. If the process is not finished yet and needs to use the main processor's shared resources again, it then goes back to the trying section and waits for another turn.

To obtain optimal performance, the shared resources must be used fairly. One of the fairness conditions says that all tasks get into their critical section after some time. "Starvation" occurs when one process never gets a turn in the critical section, often because it is much slower than the other tasks. The more destructive problem, however, is one of the violation of mutual exclusion which occurs when more than one task is in the critical section. The processor can't handle more than one job at a time and it usually crashes if this occurs.

To solve the mutual exclusion problem, many algorithms have been created to handle the tasks, preventing violation of mutual exclusion. The three mutual exclusion algorithms which we studied in the Brandeis Summer Odyssey Directed Research Project in Computer Science were Test-and-Set Bit algorithm (TSB) (1964), Dijkstra's algorithm (1965), and Peterson's algorithm (1981).

Test-and-Set Bit is the only algorithm which requires a special piece of hardware equipment (a section of the main processor) to operate properly. The hardware piece enables the processor to execute the TSB algorithm in one step, like an add or subtract instruction. When a process enters the trying section, the main processor accesses a variable which shows if any other process is in the critical section. If the critical section is empty, the hardware piece sets the variable to indicate a busy critical section, and the process enters the critical section. When the process exits critical section, the variable is changed again.

Dijkstra's algorithm resists starvation better than the Test-and-Set Bit algorithm, but sacrifices efficiency and time to become more fair. The algorithm ensures that the process trying to get into the critical section won't get into the critical section as soon as it is empty, but would have to wait until its turn has come. This makes the algorithm more fair, since every process which enters the trying section will definitely get into the critical section at least once. The same can't be said for the Test-and-Set bit algorithm, because a process may have to wait forever in the trying section while other, faster processes take turns in the critical section. Dijkstra's algorithm does not protect against starvation completely, however.

Peterson's algorithm simulates a sort of a line, or a queue, that keeps track of all the processes trying to enter the critical section. If process A enters trying section before process B, process A will be the first to enter critical section, even though process B may be faster. This guarantees that no starvation will occur.

METHOLOGY:

During our research, we have implemented a simulation of a distributed network using the Java Programming Language to test the performance and efficiency of the three mutual exclusion algorithms under different conditions. We coded each of the algorithms, transforming them from a high-level language to a very low-level string of steps that each process would take.

Splitting the algorithms into several steps was necessary to facilitate the switching of execution between processes. The procedure containing the algorithm is therefore called several times by different processes, which need to execute a different steps. A process enters the procedure, executes a step, and then exits it. The main processor must "remember" which process has executed which step. Since each loop in the algorithms is actually several recurring statements, it was necessary to break each loop up into a series of steps, controlled by *switch* statements.

The simulation read a schedule of steps for each process from a file, and then executed the schedule on each of the algorithms. The program kept track of the average number of steps for each process, the average number of times each process entered critical sections, and the average number of steps in the trying section.

To test the algorithms under different conditions, we used different, randomly-created, schedules. We tested the algorithms on remainder sections with different lengths -- 1, 10, and 30 steps and with 3, 10, and 15 processes competing for shared resources. We also tested several unbalanced schedules, where one of the processes was either twice as fast or twice as slow as the other processes. All of the schedules had the same number of steps -- 25,000. That allowed us to determine how each of the three variables -- the number of processes, the number of remainder steps, and the speed of the processes -- affect the performance of each of the three algorithms. We measured efficiency by comparing the average number of steps spent by each process in the trying section -- the lower the number of steps, the more efficient the algorithm.

RESULTS:

As the graphs show, the efficiency of each of the algorithms increases with the length of the remainder section (graphs 1, 2, and 3). The longer the remainder section is, the less time the processes spend in the trying section. That can best be explained by the fact that if the processes spent a lot of time in the remainder section, fewer "collisions" occur, maximizing efficiency. "Collisions" occur when a process is denied entry into the critical section because another processor is in the critical section. With a long remainder section, at any time, most of the processes will be in the remainder section, so a process trying to enter the critical section can do so without any difficulties. With a very short remainder section, at any time, most of the processes are in the trying section. That makes it more difficult for any particular process to enter the critical section because there is a much larger chance that it will be occupied.

The number of competing processes also adversely affects the efficiency of all of the algorithms. Since the number of collisions increases with more processes, the efficiency decreases (see graph 4).

Graphs 1, 2, and 3, show the average time each process spent in the trying section with differing numbers of competing processes and differing lengths of remainder sections under each of the three algorithms. Graph 4 shows the total number of steps the processes spent in the trying section with differing number of processes under the Test-and-Set bit algorithm and Dijkstra's algorithm.

From the graphs, it is obvious that Peterson's and Test-and-Set algorithms usually get better results than Dijkstra's algorithm. With a large number of processes, both Peterson's and Dijkstra's take a lot of steps to get into the critical section even if all but one of the processes is in the remainder section.

CONCLUSION:

Our research helped us find the optimal operating conditions for each of the algorithms. When deciding on which of the mutual algorithms to use in a network, one needs to consider the size of the network, the type of main processor and shared resources, the length of the remainder section, and other factors as well.

We have found that the Test-and-Set Bit algorithm works best with a small number of processes and a large remainder section. Its performance under these conditions exceeds that of both Peterson's and Dijkstra's algorithms, but its expensive hardware requirements decrease its overall usefulness. The fact that it does not resist any fairness violations make it less efficient than the other two algorithms in some cases. During our research, we have found a schedule which caused a starvation of one of three processes under Test-and-Set Bit algorithm. The process, although having many steps, never entered the critical section.

Peterson's algorithm is able to handle a lot of processes much better than the other two algorithms. It works much better than the other algorithms with a very small remainder section, and is only slightly exceeded in performance by Test-and-Set Bit with large remainder sections.

Dijkstra's algorithm's efficiency never exceeds that of Test-and-Set Bit algorithm and is almost always less efficient than Peterson's algorithm. It performs best, compared to other algorithms, when there are few processes and a large remainder section.

Overall, there is no clear "winner" algorithm. To use the algorithms efficiently, the network must be analyzed and, based on the analysis, the proper algorithm must be chosen.

ACKNOWLEDGMENTS:

The authors would like to thank David Wittenberg, the teacher of the Directed Research Project, and Max Saltykov, the teaching assistant, for their help with our research at Brandeis University. The authors would also like to acknowledge the financial support of the National Science Foundation and the Brandeis Summer Odyssey Program for making the research possible.

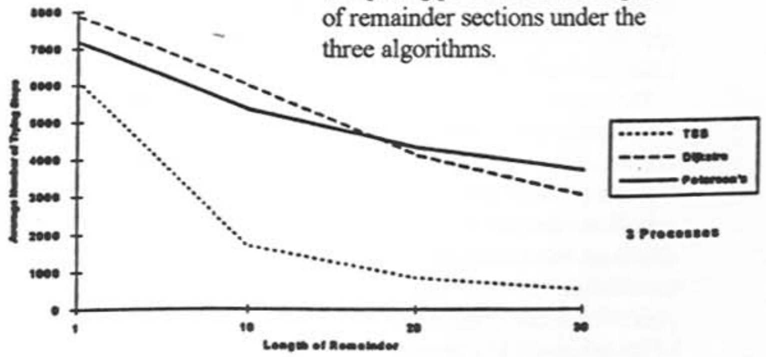
REFERENCES:

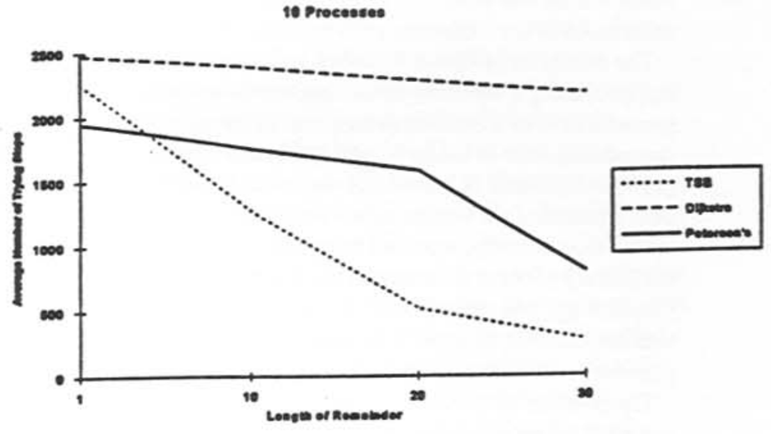
Algorithms for Mutual Exclusion, by M. Raynal; The MIT Press, 1986.

Distributed Algorithms, by Nancy Lynch; Morgan Kaufmann Publishers, 1996. "Simulating Distributed Computation for High School Students", by David Wittenberg, 1996.

The Java Programming Language, by Ken Arnold and James Gosling; Addison-Wesley Publishing Company, Inc., 1996.

Figures 1, 2, and 3. Shows the average time each process spent in the trying section with differing numbers of competing processes and lengths of remainder sections under the three algorithms.





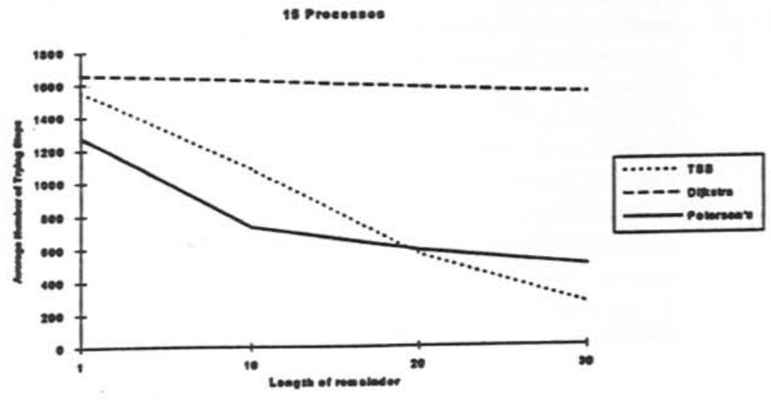
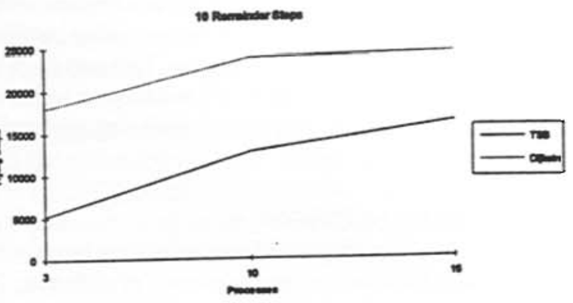


Figure 4. Shows the total number of steps the processes spent in the trying section with differing number of processes under the Test-and-Set bit algorithm and Dijkstra's algorithm.


10 Remainder Steps
Trying Steps
Processes
TSB
Dijkstra