COL331 - Operating Systems
Assignment 2

Nilaksh Agarwal, 2015PH10813

March 23, 2019

Objective

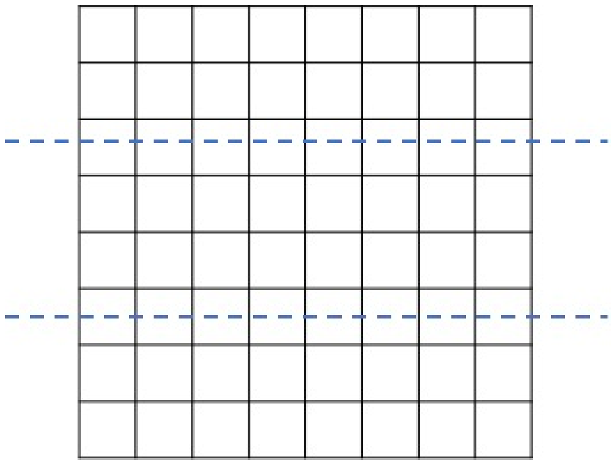
To write user programs to achieve the following functionalities:

1. Jacob Algorithm
   (a) To implement the steady state heat distribution algorithm using the Jacobi algorithm in xv6
   (b) To parallelise this algorithm
2. Maekawa Algorithm
   (a) Implement the Maekawa algorithm for mutual exclusion. This is used by parallel processes to get an exclusive lock to access a shared resource.
3. Linux Implementation
   (a) Implement both the previous parts in Linux (since xv6 doesn't support multiple cores)

Jacob

In this part, we parallelised the jacob.c code given to us. For this, we split the matrix into P smaller sections (P being the number of parallel processes), and calculating the values of the cells independently.

We split the matrix along only one axis, into multiple smaller rectangular matrices as shown:



To coordinate between processes, we define some pipes:

```
        int from_parent[P][2];   //From coodinator process to child
        int to_parent[P][2];     //From child process to parent
        int child_front[P-1][2];//From child to neighbour child
        int child_back[P-1][2]; //From child to neighbour child
```

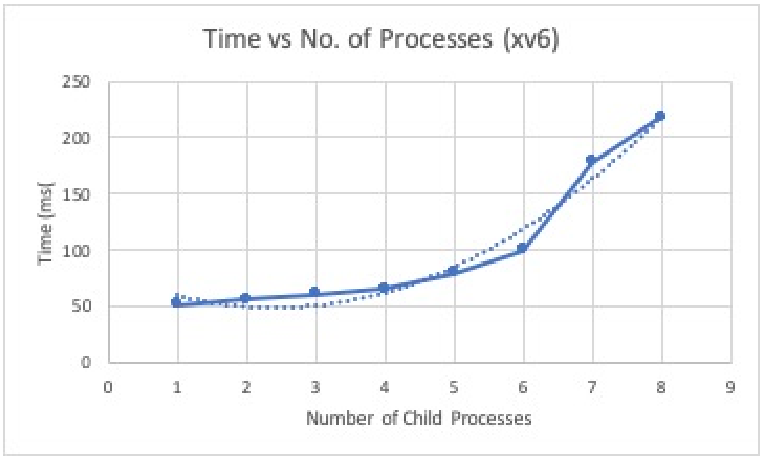
Each child process (P total) is given one subsection of the matrix to compute. It then iteratively calculates the values of each cell element and the "diff" or change in the values per iteration.

The child process calculates it's elements and passes the last and first row to the neighbouring processes. (Since they need it to compute their values). It also gets the same from it's neighbours. After calculating the new values, it computes the max change (diff) between an old and new value among it's matrix, and passes this value to the parent.

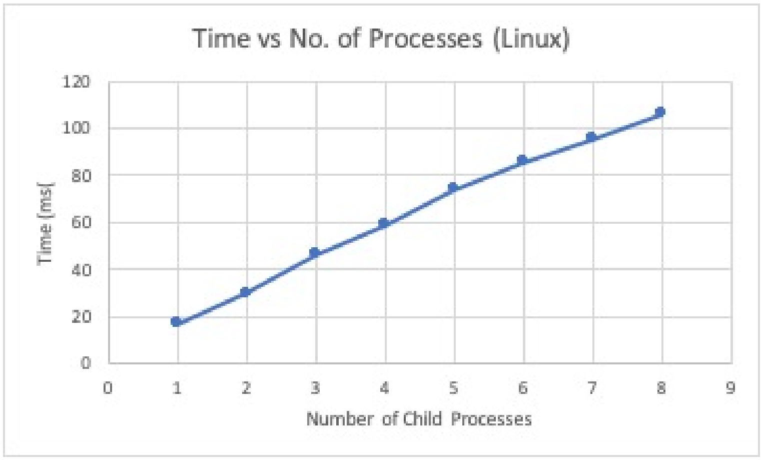
The parent collects all these "diff"s and finds the max among them. If the max(diff) is below a threshold epsilon or if the counter is greater than a N, it passes a "finished" flag to all it's children, who stop their computations.

Otherwise, the process repeats until these conditions are met.

But we see, since xv6 runs on a single processor, increasing the number of processes worsens the performance, especially when we add additional time for scheduling overheads. So, the following trends are observed for xv6 and linux respectively.



This shows a polynomial (Fitting a quadratic trendline) relation with number of processes. This is due to the overheads of simulating multiple cores on a single core CPU.



In Linux however, the result is linear, still showing an increase, but much more gradual change. The increase however, might still be due to overheads of pipes, of blocking message receives etc.

Proof of correctness

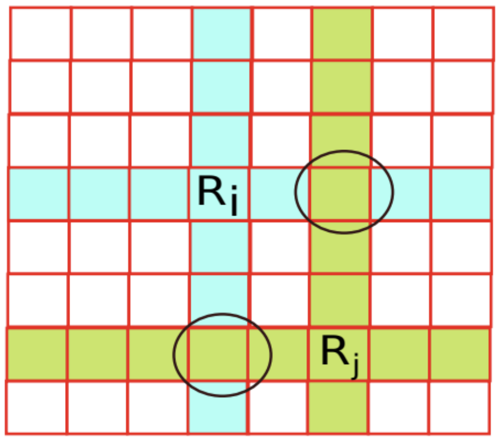
Here, we compared our outputs to the ones by the original jacob.c given in the problem statement. Since we are just dividing the sections for which to do the computation, the algorithm remains essentially the same.

Furthermore, by comparing outputs for different combinations of N,e,T,P and L, we're able to establish that the three programs (original jacob.c, xv6 jacob.c and jacob_linux.c) are the same.

Maekawa Algorithm

In this part, we implemented the Maekawa [1] algorithm for mutual exclusion to access a shared resource.

The mutual exclusion is brought about by a *quorum* for each process, which has to agree to grant access to the critical sections. This quorum is built in such a way that no two processes can get a complete acceptance from their individual quorums, hence providing mutual exclusion to the files.



This shows the quorum of two processes R_i and R_j
Image from slides on Mutual Exclusion by Prof. Smruti R. Sarangi, IIT Delhi.

This process works by implementing an inquiry based system, where each process sends various messages back and forth to each process in it's quorum. The various kinds of messages are:

- Request - Sent to all members of quorum to gain access to shared resource.
- Locked - Sent to a requesting process when the request is approved.
- Failed - Sent back to a requesting message when another higher priority process is waiting for access before it.
- Inquire - Sent to a process to check status of the lock given to the process.
- Relinquish - Sent in reply to an inquire if a process has recieved one or more failed messages from others in it's quorum.
- Release - Sent back to all processes in quorum once a process has comlpeted accessing the shared resource.

Here I have impliemnted another *END* message sent to the parent controller once access is done. This is to keep track of how many processes are yet to access the resource.

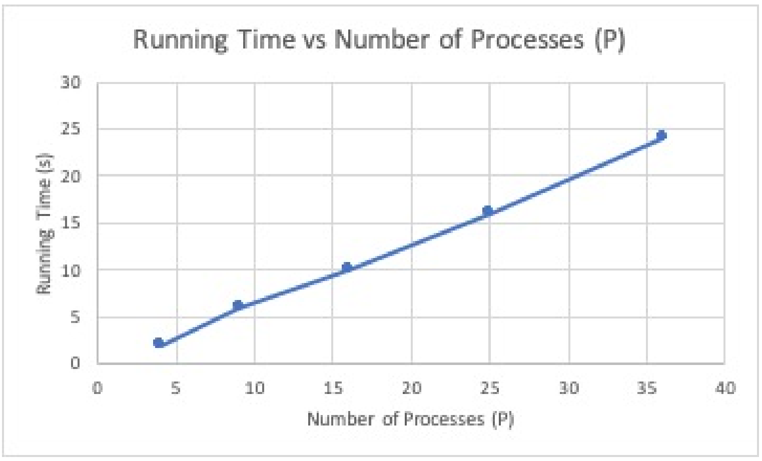
The messages contain the message as well as the PID of the sending process.

```
struct message {
        int data;
        int sender;
};
```

In this assignment we have 3 kinds of Processes :

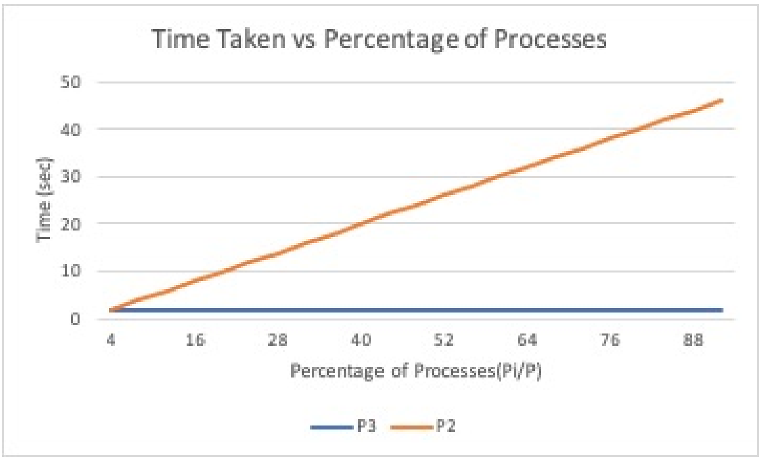
1. P1 : Don't access the shared resource
2. P2 : Access the shared resource for 2 secs.
3. P3 : Access the shared resourcee for an instant.

While varying these numbers, we see that the time taken varies almost linearly as we increase square of number of process (n^2)



This plot assumes that P1 = P2 = P3 (approx)

Similarly, we find that the time taken for a process P3 to acquire a lock is almost negligible and similar to if the process didn't ask for a lock at all (P1). And the processes P2 variation linearly increases the running time (since the process asks for 2secs every time it accesses the resource)



This plot assumes that all other processes are P1 and there is only a singular P2/P3 (if P2 is being varied, P3 = 1, P1 = P - (1+P2)

References

[1] Mamoru Maekawa. A &#8730;n algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, May 1985.