Empirical Analysis of Algorithms: Banker's Algorithm

Jessica Nguy Eric Pereira

December 4, 2017

Contents

1 Problem **4**

2 Algorithm **4**
2.1 Example 5
2.2 Safety Algorithm 5
2.3 Resource-Request Algorithm 7

3 Implementation **8**
3.1 Code 8

4 Test **9**

5 Profile **10**

6 Analyze **10**

7 Evaluate **10**

8 Conclusions **11**

Abstract

This paper details the process of programming, profiling, analyzing, and writing of the group Empirical Analysis of Algorithms project. The project topic is Banker's Algorithm. The code itself is written with Visual Studio Code, the report is written with LaTeX , and the profiler used is the GNU GProf.

1 Problem

Compute best time complexity and worst time complexity of the Banker's Algorithm.

Let n be the number of processes and m be the number of resources.
The *Claim Vector* is a data structure of size m that determines the amount of available resources for each process. It is often referred to as *Work*. *Allocated Resources* is a data structure of size m*n that determines the amount of resources assigned to each process. *Maximum Claim* is an array of size m*n that defines the maximum allowed resources that each process can use. [1]
The *Need Matrix* is calculated from *Allocated Matrix* and *Maximum Claim*. It is calculated by $N[i][j] = M[i][j] - A[i][j]$.

2 Algorithm

The Safety Algorithm determines whether or not a system is in a $safe$ state. The system is considered $safe$ if all processes terminate. If processes do not terminate, then the system is in a deadlock and is considered $unsafe$.
Safety Algorithm

1.) Let Work and Finish be vectors of length m and n respectively.
Initialize: $Work = Available$
Finish$[i] = false$; for $i = 1, 2, ..., n$

2.) Find an i such that both
a.) Finish$[i] = false$
b.) $Need_i \leq Work$
$If no such i exists go to step (4)$

3.)$Work = Work$+Allocation$_i$
Finish$[i] = true$
Go to step (2)

4.) If Finish$[i] = true$ for all i, then the system is in $safe$ state

[2] The Resource-Request Algorithm allows for a process to 'request' for more resources so that it is able to terminate.

Resource-Request Algorithm
1.) If $Request_i < Need_i$
Go to step (2); otherwise raise an error condition, since the process

[1] http://www.geeksforgeeks.org/operating-system-bankers-algorithm/
[2] http://www.geeksforgeeks.org/operating-system-bankers-algorithm/

has exceeded its maximum claim.

2.) If $Request_i < Available$
Go to step(3); otherwise P_i must wait, since the resources are not available.

3.) Have the system pretend to have allocated the requested resources to process P_i by modifying hte state as follows:
$Available = Available - Request_i$
$Allocation_i = Allocation_i + Request_i$
$Need_i = Need_i - Request_i$

[3]

2.1 Example

Let $n = 3$ and $m = 4$.

The Claim Vector *(W)* is:

A	B	C	D
3	1	1	2

The Allocated Resources *(A)* is

A	B	C	D
1	2	2	1
1	0	3	3
1	2	1	0

The Maximum Claim *(M)* is

A	B	C	D
3	3	2	2
1	2	3	4
1	3	5	0

2.2 Safety Algorithm

The program determines whether or not the inputs are *safe* or *unsafe*. A program is considered $safe$ if all processes terminate. If the processes do not terminate and W is smaller than all P_n , then the operating system enters a deadlock and is considered as $unsafe$.
To calculate the Need Matrix (N), we calculate $M[i][j] - A[i][j]$ for each index of the matrix.

Thus, the Need Matrix N is

A	B	C	D
2	1	1	1
0	2	0	1
0	1	4	0

[3] http://www.geeksforgeeks.org/operating-system-bankers-algorithm/

Iterate through the Need Matrix. Given $i = 0$ we check if $N_0 < W$

$\mathrm{N}_0 =< 2|1|1|1 >$
$W =< 3|1|1|2 >$
$N_0 \leq \mathrm{W}(1)$
Therefore, N_0 is less than W. Process 0 then terminates. Add the amount allocated to P_0 to the Claim Matrix.

$\mathrm{W} = \mathrm{W} + \mathrm{A}_0$
$< 2|1|1|1 > + < 3|1|1|2 >=< 5|2|2|3 > (2)$

Increase i by 1 and continue until i is equal to n.

$\mathrm{N}_1 =< 0|2|0|1 >$
$W =< 5|2|2|3 >$
$N_1 \leq \mathrm{W}(3)$
N_1 is less than W. Process 1 then terminates. Add the amount allocated to P_1 to the Claim Matrix.

$\mathrm{W} = \mathrm{W} + \mathrm{A}_1$
$< 5|2|2|3 > + < 1|0|3|3 >=< 6|2|5|6 > (4)$

Increase i by 1 and continue until i is equal to n.

$\mathrm{N}_2 =< 1|3|5|0 >$
$W =< 6|2|5|6 >$
$N_2 \leq \mathrm{W}(5)$
N_2 is less than W. Process 2 then terminates. Add the amount allocated to P_2 to the Claim Matrix.

$\mathrm{W} = \mathrm{W} + \mathrm{A}_2$
$< 6|2|5|6 > + < 1|3|5|0 >=< 7|5|10|6 >$
(6)

End loop.

Therefore, since all processes terminated, the system is in a safe state. The order of processes that terminated is P_0, P_1, P_2. If the *Claim Matrix* was less than the *Need Matrix*, skip the current process and return to it once the code has iterated through the loop once. If the process never terminates due to a lack of resources, then the system is in an unsafe state.

2.3 Resource-Request Algorithm

Process 2 requests resources of $1B$ and $1C$. $R_2 =< 0|1|1|0 >$. Check if $R_2 < N_2$.

$R_2 \leq N_2$
<0 | 1 | 1 | 0> < <0 | 1 | 4 | 0> (7)

R_2 is less than N_2. Proceed to step (2) of the Resource-Request Algorithm.
Check if R_2 is less than the Available Resources (W).

$R_2 \leq W$
$< 0|1|1|0 > \leq < 3|1|1|2 >$ (8)R_2 is less than W. Proceed to step (3) of the Resource-Request Algorithm.
Calculate the new W, A_2, and N_2 line of the matrix.

W = W - R_2
= <3 | 1 | 1 | 2> - <0 | 1 | 1 | 0>
= <3 | 0 | 0 | 2>

$A_2 = A_2 + R_2$
= <1 | 2 | 1 | 0> + <0 | 1 | 1 | 0>
= <1 3 1 0>

$N_2 = N_2 - R_2$
= <0 1 4 0> - <0 1 1 0>
= <0 0 3 0> (9)

The Resource-Request Algorithm works for P_2 requesting $1B$ and $1C$. However, check with the Safety Algorithm to determine if this new request is safe for the operating system.

$$i = 0$$

N_0 = <2 | 1 | 1 | 1>
N_0 ≤W
< 2111 >>< 3002 > (10)Since N_0 is not less than W, skip P_0 and increment i by 1.

$$i = 1$$

N_1 = <0 2 0 1>
N_1 ≤W
< 0201 >>< 3002 > (11)Since N_1 is not less than W, skip P_1 and increment i by 1.

$$i = 2$$

N_2 = <0 0 3 0>
N_2 ≤W
< 0030 >>< 3002 > (12)Since N_2 is not less than W, and that no processes have been terminated, the operating system enters into a deadlock. The process is an $unsafe$ state.

3 Implementation

algorithm is coded in C++ and prompts user input for n, m, *Claim Vector*, *Allocated Matrix*, and *Maximum Claim*.

The code mostly features while- and for-loops to calculate the result of the algorithm.

3.1 Code

[4] Code is based off of a C variant created by Neeraj Mishra. *count* is equal to n, so that the while loop has to run through at least n loops (to go through each array size m). *running* array determines whether that specific array has already been reviewed, *running* only holds 1 or 0 to determine true or false boolean var. *max_claim* subtracting the *curr* ints at specific points in an array is equal to a specific point in the need matrix without creating the need matrix (this helped relieve some memory issues encountered when previously trying to store need matrix variables.)

```
while (count != 0) {
 safe = false;
  for (i = 0; i < p; i++) {
   if (running[i]) {
    exec = 1;\\
    for (j = 0; j < r; j++) {
     if (max_claim[i][j] - curr[i][j] > avl[j]) {
      testNum = test(testNum);
      exec = 0;
```

[4] https://www.thecrazyprogrammer.com/2016/07/bankers-algorithm-in-c.html

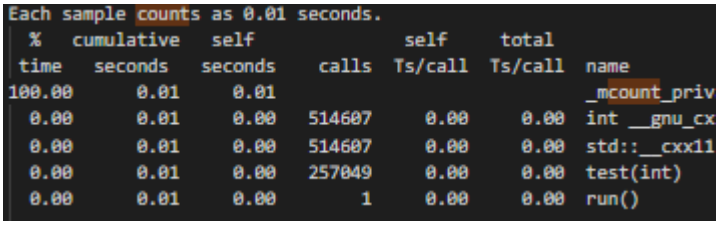
```
      break;
     }
    }
    if (exec) {
     printf("\nProcess%d is executing.\n", i + 1);
      testNum = test(testNum);
      running[i] = 0;
      count--;
      safe = true;
      for (j = 0; j < r; j++)
       avl[j] += curr[i][j];
      break;
    }
   }
  }
  if (!safe) {
   printf("\nThe processes are in unsafe state.");
   break;
  }
  if (safe)
   printf("\nThe process is in safe state.");
   printf("\nAvailable vector: ");
   for (i = 0; i < r; i++)
    printf("%d ", avl[i]);
}
```

4 Test

Was tested for best-case scenario and worst-case scenario. For best-case scenario, we had the *Claim Vector* set to a large number, so that when the loop iterated through the *Need Matrix*, the process would terminate, ensuring that the code iterates n times.

Likewise to test the worst-case scenario, we set the *Claim Vector* to a small number, so that the last process P_n would terminate, then $P_n - 1$ would terminate, and so on until P_0 terminates, ensuring that the code iterates $m * n$ times.

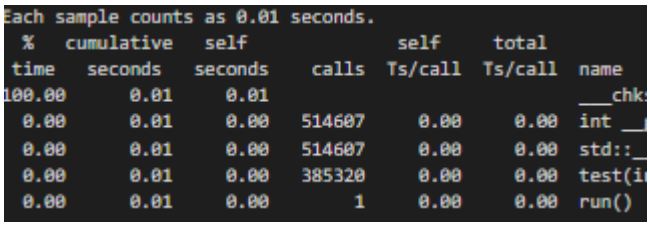
5 Profile



Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
100.00	0.01	0.01				_mcount_priv
0.00	0.01	0.00	514607	0.00	0.00	int __gnu_cx
0.00	0.01	0.00	514607	0.00	0.00	std::__cxx11
0.00	0.01	0.00	257049	0.00	0.00	test(int)
0.00	0.01	0.00	1	0.00	0.00	run()

Figure 1: Best time complexity Gprof chart for safe state



Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
100.00	0.01	0.01				___chk
0.00	0.01	0.00	514607	0.00	0.00	int __
0.00	0.01	0.00	514607	0.00	0.00	std::_
0.00	0.01	0.00	385320	0.00	0.00	test(i
0.00	0.01	0.00	1	0.00	0.00	run()

Figure 2: Worst time complexity Gprof chart for safe state

6 Analyze

The *test* function is the function used the count calls. Gprof is a profiler that does not trace calls in a loop. In order to count how many times the algorithm was called upon the *test* function was put in place as a dummy variable to check exactly how often it was called upon.
The chart produced in Figure 1 shows the best time complexity for a safe state algorithm that holds an n value of 507, and an m value of 507. Figure 1 shows that in best time case scenario it runs in nm time as it runs 257,409 times. This produces a Ω(n*m) time complexity.

The chart produced by Gprof in figure 2 shows the worst time complexity for a safe state algorithm that holds the same n and m value as the best time complexity example. In this chart the test is called on 385,320 times, which is equal to:

$$\sum_{k=0}^{n-1}(m+k) = 385,320 \tag{13}$$

This was the equation that was expected, This produces an O(n*n*m) time complexity.

7 Evaluate

The results that were expected were the results attained in our tests. According to the Algorithm, and predictions stated by the algorithm, there should be an

$n*m$ calls to the test function in the best case scenario of a safe state run. This time complexity was expected as it would review n processes and in each process do m comparisons. This is true according to the profiler, which shows that the algorithm is called on exactly $n*m$ times.

According to our predictions as well worst case scenario would be $n*n*m$ calls to the test function. This is due to the *Claim Vector* being called on and compared to every process array in the *Need* matrix. That means going through n arrays n times, each array holding m thus doing m comparisons. This leaves with the equation number 13 specified in the Analyze portion.

8 Conclusions

Banker's algorithm, although a good idea in theory, is not optimal in the use of operating systems, as it was initially built for. It is impossible for an operating system to always know how to make a possible *need* matrix available for a program; however this deadlock avoidance system was found useful in banks as it prevented issues with bankruptcy and could be run to check, with absolute certainty, in the case that the *need* matrix is absolutely certain and resources from the *max* matrix is malleable, as in there are resources that could eventually be moved out of it and it will not remain static.

References

[1] Banker's Algorithm *Geeks for Geeks*, http://www.geeksforgeeks.org/operating-system-bankers-algorithm/

[2] Banker's Algorithm in C *The Crazy Programer*, https://www.thecrazyprogrammer.com/2016/07/bankers-algorithm-in-c.html

[3] EWD 623: The mathematics behind the Banker's Algorithm, *Edsger Dijkstra*, http://www.cs.utexas.edu/users/EWD/ewd06xx/EWD623.PDF