Avoiding Register Overflow in the Bakery Algorithm

Amirhossein Sayyadabdi
University of Isfahan
ahsa@eng.ui.ac.ir

Mohsen Sharifi
University of Science and Technology
msharifi@iust.ac.ir

SRMPDS '20, Edmonton, AB, Canada

Agenda

- Background on mutual exclusion and the Bakery algorithm
- Problem statement
- Bakery++
- Performance, practicality and correctness of Bakery++
- Discussion and future work
- Conclusions

Mutual Exclusion

Started the field of concurrency.

Mutual Exclusion

Started the field of concurrency.

Originally proposed and solved by Dijkstra in 1965.

Mutual Exclusion

Started the field of concurrency.

Originally proposed and solved by Dijkstra in 1965.

A group of independent sequential processes that have access to shared memory.

Mutual Exclusion

Started the field of concurrency.

Originally proposed and solved by Dijkstra in 1965.

A group of independent sequential processes that have access to shared memory.

Prevent the processes from executing a specific region of code called the "critical section" simultaneously.

Mutual Exclusion (cont.)

Correctness conditions as specified by Knuth:

1. No two processes are allowed to execute their critical sections simultaneously.

Mutual Exclusion (cont.)

Correctness conditions as specified by Knuth:

1. No two processes are allowed to execute their critical sections simultaneously.
2. A reliable process should be allowed to enter its critical section eventually.

Mutual Exclusion (cont.)

Correctness conditions as specified by Knuth:

1. No two processes are allowed to execute their critical sections simultaneously.
2. A reliable process should be allowed to enter its critical section eventually.
3. Crashing of a process should not block others from accessing the critical section.

Mutual Exclusion (cont.)

Correctness conditions as specified by Knuth:

1. No two processes are allowed to execute their critical sections simultaneously.
2. A reliable process should be allowed to enter its critical section eventually.
3. Crashing of a process should not block others from accessing the critical section.
4. Processes may fail at any time and then restart outside of the critical section.

Mutual Exclusion (cont.)

Correctness conditions as specified by Knuth:

1. No two processes are allowed to execute their critical sections simultaneously.
2. A reliable process should be allowed to enter its critical section eventually.
3. Crashing of a process should not block others from accessing the critical section.
4. Processes may fail at any time and then restart outside of the critical section.
5. No assumptions are made about the execution speeds of processes.

Resource Management

Resource management concerns the coordination and collaboration of users.

Resource Management

Resource management concerns the coordination and collaboration of users.

It is usually based on making a decision.

Resource Management

Resource management concerns the coordination and collaboration of users.

It is usually based on making a decision.

In the case of mutual exclusion, that decision is about granting access to a resource.

Resource Management

Resource management concerns the coordination and collaboration of users.

It is usually based on making a decision.

In the case of mutual exclusion, that decision is about granting access to a resource.

Mutual exclusion is useful for supporting resource access management.

The Bakery Algorithm

1. Processes can enter their critical sections in first-come-first-served order.

The Bakery Algorithm

1. Processes can enter their critical sections in first-come-first-served order.
2. The failure of individual system components will not cause the entire system to halt.

The Bakery Algorithm

1. Processes can enter their critical sections in first-come-first-served order.
2. The failure of individual system components will not cause the entire system to halt.
3. No process writes into the memory of other processes.

The Bakery Algorithm

1. Processes can enter their critical sections in first-come-first-served order.
2. The failure of individual system components will not cause the entire system to halt.
3. No process writes into the memory of other processes.
4. If a read and a write occur simultaneously at a memory location, then the value obtained by the read operation may have any arbitrary value.

The Bakery Algorithm (cont.)

```
integer array choosing [1..N], number [1..N];
begin integer j;
L1: choosing[i] := 1;
number[i] := 1 + maximum (number[1], … , number[N]);
choosing[i] := 0;
for j = 1 step 1 until N do
begin
    L2: if choosing[j] ≠ 0 then goto L2;
    L3: if number[j] ≠ 0 and (number[j], j < number[i], i) then goto L3;
end;
critical section; number[i] := 0; noncritical section; goto L1;
end
```

Trouble with Bakery

The Bakery algorithm assumes unbounded registers.

Trouble with Bakery

The Bakery algorithm assumes unbounded registers.

The value of number[i] for process i may tend to infinity!

Trouble with Bakery

The Bakery algorithm assumes unbounded registers.

The value of number[i] for process i may tend to infinity!

Two competing processes may keep incrementing this value forever:

number[i] := 1 + maximum (number[1], … , number[N]);

Trouble with Bakery

The Bakery algorithm assumes unbounded registers.

The value of number[i] for process i may tend to infinity!

Two competing processes may keep incrementing this value forever:

number[i] := 1 + maximum (number[1], … , number[N]);

This causes register overflow in real systems.

Our Purpose

Avoid register overflows in the Bakery algorithm without making compromises.

Previous approaches to achieve the same goal:

- Introduce new shared variables.
- Redefine certain operations or functions in the algorithm.

Notable Solutions

The majority of approaches to avoid overflows in the Bakery algorithm include:

1. Changing the definitions of "<" operator and "maximum" function.

Notable Solutions

The majority of approaches to avoid overflows in the Bakery algorithm include:

1. Changing the definitions of "<" operator and "maximum" function.
2. Using modulo arithmetic instead of integer arithmetic.

Notable Solutions

The majority of approaches to avoid overflows in the Bakery algorithm include:

1. Changing the definitions of "<" operator and "maximum" function.
2. Using modulo arithmetic instead of integer arithmetic.
3. Introducing new shared variables or using extra memory.

Notable Solutions

The majority of approaches to avoid overflows in the Bakery algorithm include:

1. Changing the definitions of "<" operator and "maximum" function.
2. Using modulo arithmetic instead of integer arithmetic.
3. Introducing new shared variables or using extra memory.
4. Resetting the values of registers before an overflow occurs.

The Bakery++ Algorithm

There is an important theoretical question in the paper that introduced Bakery:

"Can one find an algorithm for finite processors such that processors enter their critical sections on a first-come-first-served basis, and no processor may write into another processor's memory?"

The Bakery++ Algorithm

There is an important theoretical question in the paper that introduced Bakery:

"Can one find an algorithm for finite processors such that processors enter their critical sections on a first-come-first-served basis, and no processor may write into another processor's memory?"

To our knowledge, all of the previous works on bounding the Bakery algorithm have failed to answer this question.

Bakery++

Bakery++ is a slightly modified version of the Bakery algorithm.

Bakery++

Bakery++ is a slightly modified version of the Bakery algorithm.

We call it "Bakery++" because it is almost identical to Bakery.

Bakery++

Bakery++ is a slightly modified version of the Bakery algorithm.

We call it "Bakery++" because it is almost identical to Bakery.

It does not use any additional variables.

Bakery++

Bakery++ is a slightly modified version of the Bakery algorithm.

We call it "Bakery++" because it is almost identical to Bakery.

It does not use any additional variables.

It does not redefine the operators or functions used in Bakery.

Bakery++ (cont.)

```
constant M;
integer array choosing [1..N], number [1..N];
begin integer j;
L1: if ∃ q ∈ {1, … , N} such that number[q] ≥ M then goto L1;
choosing[i] := 1;
number[i] := maximum (number[1], … , number[N]);
if number[i] ≥ M then begin
                          number[i] := 0; choosing[i] := 0; goto L1;
                          end
else number[i] := number[i] + 1;
choosing[i] := 0;
for j = 1 step 1 until N do
begin
      L2: if choosing[j] ≠ 0 then goto L2;
      L3: if number[j] ≠ 0 and (number[j], j < number[i], i) then goto L3;
end;
critical section; number[i] := 0; noncritical section; goto L1;
end
```

Performance and Practicality

Bakery++ does not introduce new variables, so the spatial complexities of both algorithms are identical.

Performance and Practicality

Bakery++ does not introduce new variables, so the spatial complexities of both algorithms are identical.

The temporal complexity of Bakery++ depends on the number of executions of the goto statement exactly after label L1.

Performance and Practicality

Bakery++ does not introduce new variables, so the spatial complexities of both algorithms are identical.

The temporal complexity of Bakery++ depends on the number of executions of the goto statement exactly after label L1.

The reason Bakery ++ is useful in practice is that it is almost as simple as the original Bakery, without new variables and fancy operations or functions.

Performance and Practicality

Bakery++ does not introduce new variables, so the spatial complexities of both algorithms are identical.

The temporal complexity of Bakery++ depends on the number of executions of the goto statement exactly after label L1.

The reason Bakery ++ is useful in practice is that it is almost as simple as the original Bakery, without new variables and fancy operations or functions.

There are no practical limitations for implementing the Bakery++ algorithm.

Correctness Argument

Bakery++ does not introduce new variables and its control flow is almost identical to Bakery. The changes that have been made to Bakery include:

- Introducing a constant that represents the maximum value storable.

Correctness Argument

Bakery++ does not introduce new variables and its control flow is almost identical to Bakery. The changes that have been made to Bakery include:

- Introducing a constant that represents the maximum value storable.
- Adding a conditional statement and a goto after label L1 that does not manipulate the values of Bakery's data objects.

Correctness Argument

Bakery++ does not introduce new variables and its control flow is almost identical to Bakery. The changes that have been made to Bakery include:

- Introducing a constant that represents the maximum value storable.
- Adding a conditional statement and a goto after label L1 that does not manipulate the values of Bakery's data objects.
- Adding a conditional statement before incrementing the maximum value obtained from reading all processes' variable number.

Correctness Argument

Bakery++ does not introduce new variables and its control flow is almost identical to Bakery. The changes that have been made to Bakery include:

- Introducing a constant that represents the maximum value storable.
- Adding a conditional statement and a goto after label L1 that does not manipulate the values of Bakery's data objects.
- Adding a conditional statement before incrementing the maximum value obtained from reading all processes' variable number.
- If there is a possibility of overflow in process i, then we simply set number[i] = choosing[i] = 0 and then we jump to label L1. Otherwise, we will continue by incrementing the maximum value and the original Bakery algorithm.

Discussion and Future Work

There are two questions:

1. What happens if there are more customers in the bakery than the maximum number that can be stored in a register?
2. What is the definition of the exact moment when a process is considered to have taken its turn for entering its critical section?

Conclusions

We revisited the problem of mutual exclusion and the Bakery algorithm, the first true mutual exclusion algorithm, to solve the problem of integer overflow.

Conclusions

We revisited the problem of mutual exclusion and the Bakery algorithm, the first true mutual exclusion algorithm, to solve the problem of integer overflow.

Previous approaches to solving the problem of register overflow concentrated on making use of new variables and redefining the operations or functions used in the original Bakery, and they were complicated solutions.

Conclusions

We revisited the problem of mutual exclusion and the Bakery algorithm, the first true mutual exclusion algorithm, to solve the problem of integer overflow.

Previous approaches to solving the problem of register overflow concentrated on making use of new variables and redefining the operations or functions used in the original Bakery, and they were complicated solutions.

Bakery ++ is quite simple and it differs from Bakery in just a few instructions.

Conclusions

We revisited the problem of mutual exclusion and the Bakery algorithm, the first true mutual exclusion algorithm, to solve the problem of integer overflow.

Previous approaches to solving the problem of register overflow concentrated on making use of new variables and redefining the operations or functions used in the original Bakery, and they were complicated solutions.

Bakery ++ is quite simple and it differs from Bakery in just a few instructions.

We have specified Bakery++ in the PlusCal language and performed model checking.

Thank you!