Chapter 1

Bakery Algorithm

Abstract

Bakery algorithm is a classic solution to the critical section problem in concurrent programming, ensuring mutual exclusion among multiple processes accessing shared resources. It operates by assigning a unique "ticket" or number to each process wanting to enter the critical section. These tickets are generated using a lexicographical order, meaning processes with lower numbers have priority. Before entering the critical section, a process announces its intention and takes a number higher than any other process's current number. When checking for access, a process compares its ticket with others; if its number is the lowest, or if its number is equal to another process's but its process ID is lower (breaking ties), it can proceed. This ensures that only one process is in the critical section at any given time, preventing race conditions and data corruption.

1.1 Introduction

This Java implementation demonstrates the Bakery algorithm, a classic solution to the critical section problem in concurrent programming. It ensures mutual exclusion among multiple threads accessing a shared resource by assigning numbered "tickets" to each thread, granting access based on ticket order and thread ID to prevent race conditions.

1.2 Bakery Algorithm

Bakery algorithm guarantees mutual exclusion in concurrent systems by assigning numbered "tickets" to processes competing for a critical section.

1.2.1 Number Selection

Each process, before entering the critical section, chooses a number higher than any other process's current number, effectively "taking a ticket" in a bakery-like fashion. This number represents the process's priority.

1.2.2 Turn Checking

Processes then check the numbers of all other processes. A process can enter the critical section only if its number is the lowest or, in case of a tie, if its process ID is lower than the other process's ID.

1.2.3 Tie Breaking

If two or more processes choose the same number, the process with the lower process ID is given priority. This deterministic tie-breaking mechanism is crucial for ensuring mutual exclusion and preventing deadlock or starvation.

1.3 Code

```
1  package javaapp;
2  /* Peterson Algorithm
3   * @author Shahriar
4   * @date Nov 25, 2024
5   */
6  public class JavaApp {
7      public static void main(String[] args) {
8          int numProcesses = 5;
9          BakersAlgorithm bakers = new BakersAlgorithm(numProcesses);
10         Thread[] threads = new Thread[numProcesses];
11         for (int i = 0; i < numProcesses; i++) {
12             int processId = i;
13             threads[i] = new Thread(() -> {
14                 while (true) {
15                     bakers.enterRegion(processId);
```

```
16                      System.out.println("Process␣" + processId + "␣
                            is␣in␣critical␣region.");
17                      try {
18                          Thread.sleep(1000);
19                      } catch (InterruptedException e) {
20                          e.printStackTrace();
21                      }
22                      bakers.leaveRegion(processId);
23                      System.out.println("Process␣" + processId + "␣
                            is␣in␣non-critical␣reigon.");
24                  }
25              });
26              threads[i].start();
27          }
28      }
29  }
30  class BakersAlgorithm {
31      private int N;
32      private int[] number;
33      private boolean[] choosing;
34      public BakersAlgorithm(int N) {
35          this.N = N;
36          number = new int[N];
37          choosing = new boolean[N];
38      }
39      public void enterRegion(int process) {
40          int max = 0;
41          choosing[process] = true;
42          number[process] = 1 + maxFor(number);
43          choosing[process] = false;
44          for (int j = 0; j < N; j++) {
45              while (choosing[j]);
46              while (number[j] != 0 && (number[j] < number[process]
                    || (number[j] == number[process] && j < process)));
47          }
48      }
49      public void leaveRegion(int process) {
50          number[process] = 0;
51      }
52      private int maxFor(int[] array) {
53          int max = 0;
54          for (int i = 0; i < array.length; i++) {
55              max = Math.max(max, array[i]);
56          }
57          return max;
58      }
59  }
```

1.4 Explain code

This Java code implements the Bakery algorithm for mutual exclusion, allowing multiple threads to safely access a shared resource (the "critical region"). Let's break down the code step by step:

1.4.1 JavaApp Class (Main Execution)

- int numProcesses = 5;: This line defines the number of concurrent processes (represented by threads) that will compete for the critical region.
- BakersAlgorithm bakers = new BakersAlgorithm(numProcesses);: An instance of the BakersAlgorithm class is created, initializing the necessary data structures for the algorithm.
- Thread[] threads = new Thread[numProcesses];: An array of Thread objects is created to manage the concurrent threads.
- The for loop (from i = 0 to numProcesses - 1) creates and starts each thread:
- int processId = i;: Assigns a unique ID to each thread.
- threads[i] = new Thread(() -¿ ...);: Creates a new thread using a lambda expression to define its execution logic.
- while (true) ... : This infinite loop represents the continuous execution of each process.
- bakers.enterRegion(processId);: This is the crucial part where the Bakery algorithm's entry protocol is executed. The thread attempts to acquire access to the critical region.
- System.out.println("Process " + processId + " is in critical region.");: This line represents the critical section. Only one thread should be executing this line at any given time.
- try Thread.sleep(1000); catch (InterruptedException e) e.printStackTrace(); : This simulates some work being done within the critical region, pausing the thread for 1 second.
- bakers.leaveRegion(processId);: After finishing in the critical region, the thread calls this method to release its access.
- System.out.println("Process " + processId + " is in non-critical reigon.");: This indicates the thread is now outside the critical section.
- threads[i].start();: Starts the execution of the newly created thread.

1.4.2 BakersAlgorithm Class (Algorithm Implementation):

- private int N;: Stores the number of processes.
- private int[] number;: This array stores the "ticket" or number for each process. A value of 0 indicates the process is not trying to enter the critical region.

- private boolean[] choosing;: This array is used to handle the brief period when a process is choosing its number, preventing race conditions during number selection.
- BakersAlgorithm(int N) (Constructor): Initializes the N, number, and choosing arrays.
- enterRegion(int process): This is the core of the Bakery algorithm:
- choosing[process] = true;: Sets the choosing flag for the current process to true, indicating it's in the process of choosing a number.
- number[process] = 1 + maxFor(number);: The process chooses a number that is one greater than the maximum number currently held by any other process. This ensures that each process gets a unique, increasing number. The maxFor method (explained below) finds the maximum number.
- choosing[process] = false;: Sets the choosing flag back to false.
- The nested for and while loops implement the waiting and checking logic:
- while (choosing[j]);: This inner loop waits if process j is currently choosing a number. This prevents race conditions during number selection.
- while (number[j] != 0 (number[j] ¡ number[process] —— (number[j] == number[process] j ¡ process)));: This is the main waiting condition. A process waits if another process j has a non-zero number (meaning it wants to enter the critical region) and either:
- number[j] ¡ number[process]: Process j has a lower number (higher priority).
- number[j] == number[process] j ¡ process: Process j has the same number but a lower process ID (tie-breaker).
- leaveRegion(int process): Sets the process's number back to 0, indicating it has left the critical region.
- private int maxFor(int[] array): A helper method that finds the maximum value in an integer array.

1.4.3 In summary

This code uses the Bakery algorithm to manage concurrent access to a critical section. Each thread simulates a process, and the BakersAlgorithm class ensures that only one thread can be in the critical region at any given time, preventing data corruption and race conditions. The choosing array and the tie-breaking mechanism (using process IDs) are crucial for the algorithm's correctness.

References

Geeks for Geeks https://www.geeksforgeeks.org/bakery-algorithm-in-process-synchronization/
Scaler topics https://www.scaler.com/topics/operating-system/bakery-algorithm-in-os/
Wikipedia https://en.wikipedia.org/wiki/Lamport's_bakery_algorithm
Google Gemini for editing text https://gemini.google.com/