OS Project - Fribourgeois Bakery

Group 1: Gabriel, Odermatt, Wey & Yang

May 27, 2022

Contents

1 Introduction

The present documentation reports on the solution to the *Fribourgeois Bakery* project elaborated by Group 1.

The discussion of the solution will be structured as follows: Chapter one will present our solution to the task "Learning To Make Bread". Chapter two will focus on "Too Good To Go" and chapter four will present our additional feature. Our implementation decisions as well as the scenarios will be discussed in chapter one.

With our implementation we provide a commandline interface that allows the user to test all functionalities, provide parameters and run the testing scripts.

2 Learning To Making Bread

In summary, our implementation of "Learning To Make Bread" works as follows: (i) The user inputs the parameters such as the bread types and the ingredients they require, the number of apprentices, the scheduling algorithm, etc. (ii) Then the threads representing the baker, the apprentices and the so-called 'shopper' are spawned and start their respective task: The apprentices pick a recipe at random, access the inventory to get the required ingredients and bake the bread. The baker ensures that only one apprentice at a time accesses the inventory and the shopper restocks the inventory when it an ingredient runs out.

Let us look at some implementation decisions in a bit more detail:

2.1 Inventory

The inventory is implemented as a binary tree where each tree-node represents an ingredient type. Each of those tree-nodes holds a linked list where each list-node represents one unit of that ingredient type. This datastructure can grow in two ways: With respect to the number of ingredients (the tree grows) and with respect to the number of units of a specific ingredient (the respective linked list grows). The structure is defined in the file 'Inventory_BinTree.c' and exposes some useful functions such as 'registerIngredient', 'addIngredient', 'takeIngredient' and 'restock'.

Building and searching of tree: The tree is sorted lexographically – meaning that the left child of each parent node contains an ingredient type whose name is smaller and the right child contains an

ingredient type whose name is bigger than the parents name. "Smaller" and "bigger" in this context refer to the ordering of characters proposed by the ASCII standard. Due to this strict ordering of ingredients, the tree can be traversed and searched very efficiently.[1]

Traversal of the tree: The tree can be traversed both iteratively and recursively. The function 'getInvNode' – which searches the tree for an ingredient and returns one unit of that ingredient – works iteratively, i.e. the tree is traversed from the root to the leaves using a while-loop. Conversely, the function restockRecursion – which traverses the tree and restocks every ingredient – works recursively, i.e. the tree is traversed from the left-most leave to the right-most leave.

Mutual exclusion: Our datastructure does not enforce mutual exclusion. That is: It is up to the client using the datastructure to ensure that it is accessed only by one thread at a time. Our client-side approach to mutual exclusion will be discussed in the next section.

2.2 Mutual Exclusion and Scheduling

The inventory may only be accessed by one apprentice at a time. Since the resources are limited, every apprentices tries to access the inventory as soon as possible. Therefore, access to the inventory has to be made (i) mutually exclusive and (ii) coordinated in such a way that the apprentices may access in a specific order.

Scheduling: When an apprentice wants to access the inventory, they have to announce their interest by modifying the 'interested_array' at the index that corresponds to their thread-id. Depending on the scheduling algorithm used, they modify the array differently, e.g. if the algorithm 'ARRIVALORDER' is used, the apprentices add their time of arrival to the 'interested_array'. Once the apprentice has announced their interested they sleep on a semaphore. The baker analyses the interested_array and chooses the apprentice according to the order of the scheduling algorithm. The baker then wakes that apprentice by posting their semaphore and then sleeps on its own semaphore. The chosen apprentice wakes up and enters the inventory. When they've finished they leave the inventory, wake the baker and start baking their bread. This process repeats until the the bakery has reached the maximal number of breads for the day.

Our group has implemented a total of four scheduling algorithms: (i) ARRIVALORDER (analogy to FCFS) sorts the apprentices by their arrival time, favoring apprentices that have arrived earlier. (ii) FASTLEARNERS (analogy to "Shortest Process Next" or "Priority Scheduling") sorts the apprentices according to the number of breads they have already produced. The more bread an apprentice has baked the higher they're ranked. (iii) FAIRLEARNERS (analogy to "Guaranteed Scheduling") is the counterpart to FASTLEARNERS and favors the apprentice with the least amount of baked breads. (iv) Finally, PRELEARNERS (analogy to "Round Robin") enforces a predefined order that corresponds to the apprentice's thread-id.

Our scheduling algorithms are mere analogies to the real scheduling algorithms running inside an OS. However, in our implementation there is no analogous concept to I/O-bound and CPU-bound processes since the apprentices will spend the same amount of time in the inventory. The chosen scheduling algorithm does therefore not influence the running time of the "Making Bread" script. Comparing our algorithms regarding turnaround-time and throughput would therefore not make much sense. The following comparison of the scheduling algorithms will therefore focus on the real scheduling algorithms and compare them regarding throughput, turnaround-time and fairness (see Table 1).

Mutual Exclusion: Thanks to the scheduling algorithms it is practically impossible for two apprentices to access the inventory at the same time. However, just to be sure, we still made access to the inventory mutually exclusive using the mutex 'inventory_mutex'.

2.3 Scenarios

Lets have a look how our implementation manages the following three scenarios:

[1] Strictly speaking this does not always have to be the case. If the ingredients happen to be added in ASCII order (or in reverse) then the tree would be a normal linked list and searching the tree would perform accordingly.

Table 1: Comparison of Scheduling Algorithms

Scheduling Algorithm	+	-
FCFS	Optimal usage of CPU time	Sub-optimal throughput and turnaround time (Because a CPU-bound processes block the CPU)
Shortest Process Next	Optimal throughput and turnaround time	CPU-bound processes might starve
Priority Scheduling	Allows to enforce a metric	Processes with low priority might starve
Guaranteed Scheduling	Fair	Sub-optimal throughput and turnaround time
Round Robin	Fair	Sub-optimal throughput and turnaround time

Apprentices add the same item to the inventory: In our implementation a dedicated thread called 'shopper' takes care of restocking the inventory. Since there is only one such thread it is impossible for two threads to add the same item at the same time. Furthermore, the inventory is mutexed, ensuring that only one thread can access it at a time. As an additional precaution, the function registerIngredient ensures that when an ingredient is registered, this ingredient has not been registered before. If it is not the case, the function simply adds units to the already existing ingredient.

Apprentices access the same item in the inventory: This is impossible since access to the entire inventory is mutually exclusive. This is a results of our decision to implement mutual exclusion client-side and not inside the datastructure. We chose this approach since it is conceptually cleaner to leave the decision to enforce mutual exclusion (or not) to the client. However, it would have been more performant to ensure mutual exclusion inside the datastructure because we could have differentiated between the different ingredient types making access to every ingredient individually mutually exclusive. This would have allowed several apprentices to access the inventory at the same time for as long as the do not touch the same ingredient.

An apprentice is adding an item and another is retrieving it: Again, this scenario is not possible since the entire datastructure is mutexed. Therefore, if the shopper tried to restock the inventory while an apprentice was retrieving an ingredient, they would sleep on the mutex until the apprentice leaves the inventory.

3 Too Good To Go

On a high level, our implementation of "Too Good To Go" works as follows: (i) The user inputs the parameters such as the name of the bread types, the number of units of each type, the paging algorithm to be used, the number of ticks between TGTG-decisions, etc. (ii) The bakery produces an initial amount of breads. (iii) The bakery sells the bread and bakes new breads as they run out. (iv) Every X number of ticks, the baker applies the paging algorithm to decide which breads to donate. (v) As soon as the selling target for the day is reached, the bakery closes.

Let's have a look at the actual implementation of some of those steps:

Datastructure: The bread types are represented as a dynamic array. Each bread type occupies an index in the array. Every index contains a linked list in which each node represents one unit of that type of bread. Each node is timestamped when it is added to the list. Furthermore, the linked list contains some additional information such as the timestamp of the oldest bread present in the list, whether this bread type was recently requested by a customer, etc.

Time of TGTG: The point in time when the baker has to decide which breads to donate is user-defined and enforced by a thread called 'tgtg_coordinator'. Every X seconds, the 'tgtg_coordinator' sets the flag 'tgtg_flag' to TRUE. After every time the baker sold a bread, they check the flag. If its set to TRUE we call the paging algorithm to decide which breads to donate. Nota bene: Since the flag

'tgtg_flag' is a shared variable we ensure mutual exclusion using the mutex 'mutTGTGFlag'.

Paging Algorithms: We have implemented all three strategies: "donate the old bread" (analogy to FIFO), "second chance" (analogy to SECOND_CHANCE) and "not recently sold" (analogy to NRU). The implementation of FIFO works as follows: The baker looks at the timestamp of every bread and checks whether the bread was made in the last X seconds where X is user-defined via the variable 'grace_period'. If the bread is older than the current time minus the grace period, it is donated. The implementation of NRU checks for each bread type whether that bread type was sold to at least one customer in the last Y seconds. Y is user-define via the variable 'ticks'. If not, all breads of that type are donated. The last algorithm, SECOND_CHANCE, is a hybrid of FIFO and NRU: Like FIFO, it checks for every bread whether it is older than a certain number of seconds. If so, it also checks whether that bread type was purchased by at least one customer during the last Y seconds. If so, the bread gets a second chance and is not donated.

4 Additional Feature

Group 1 chose to implement the second additional feature which we will call "Sleeping Baker". Conceptually, our implementation works as follows: (i) The user inputs the parameters such as the number of customers that will try to enter the baker, the number of chairs, etc. (ii) Then the baker-thread is created. For as long as the bakery is empty, the baker will sleep. As soon as a customer enters the bakery, the baker will wake up and serve them. (iii) From then on, some customer-threads are created at certain intervals and will enter the bakery. If a chair is free they will take a seat and wait to be served. If no chair is free they will leave again. (iv) The script stops when the maximal number of customers was created.

Let's have a look at the actual implementation:

Synchronization: The interactions between the customers and the baker is synchronized using semaphores. When a customer takes a seat on a free chair they up the baker's semaphore and start sleeping on their own semaphore. They baker's semaphore stores these "ups" and ensures that the baker does not fall asleep for as long as there are still customers in the bakery.

Datastructure: The customers that have been able to take a chair are stored in a queue. Every time the baker has finished serving a customer he dequeues the next semaphore from the queue.