001 - Theoretical and practical aspects in the construction of concurrent systems

Question 1:

The interference concept in concurrent systems is when a shared resource is used by 2 threads or process at the wrong order (Lecture 2 Dr Daniele Scrimieri 2026). This resource is then called a critical resource, and its access need to be restricted at only one thread or process for one operation. In case of interference, the program can behave unpredictably or produce incorrect results. These incorrect results could be from a thread that write a value before another thread read this value without taking in account the wanted order.

For the Betty's cafe problem's case of interference can be about the buffet.

The buffet could have 1 tea, and a client wants 2 teas, at the same time a staff wants to bring 1 tea if the client takes the tea he wants before the staff have bring his tea there will be -1 tea in the buffet. Or having a negative number of teas is impossible.

Question 2:

A deadlock is a situation where several threads need each other to complete their tasks. In such situation the threads that are in this situation are blocked and can't finish their execution (Lecture 4 Dr Daniele Scrimieri 2026).

A deadlock can happen when four necessary and sufficient conditions are achieved.

The first condition is the "mutual exclusion" (Lecture 4 Dr Daniele Scrimieri 2026: 4); it means that a shared resource cannot be used by several threads at the same time. To avoid this the resource should be shared without lock if possible (Lecture 4 Dr Daniele Scrimieri 2026).

The second condition is "the incremental acquisition" (Lecture 4 Dr Daniele Scrimieri 2026: 4); it means that a thread keep the lock on a resource while waiting for another one. If several threads hold on their resource all the shared resource could be locked and the program would be blocked (Lecture 4 Dr Daniele Scrimieri 2026).

The third condition is the lack of preemption; it means that the system cannot release lock on resource the unlock mechanism needs to be voluntary. We can avoid the lack of pre-emption if the system could release locked resource (Lecture 4 Dr Daniele Scrimieri 2026).

The fourth condition is the "circular wait" (Lecture 4 Dr Daniele Scrimieri 2026: 4); it means that a chain of process exists where each process hold the necessary resource for its successor (Lecture 4 Dr Daniele Scrimieri 2026). To avoid a circular chain, we need to break this chain by introducing some irregularities.

To avoid a deadlock, we can either break a circular chain, adding some preemption, avoid the incremental acquisition or avoid using lock to use shared resource.

Question 3:

In the current situation one way to cause a deadlock is to make the client holds the buffet while wanting to play the piano and then releasing the buffet and vice versa. If the piano is already occupied by 2 client, one client wants to play after eating and one of the clients playing wants to eat the program will be blocked.

To avoid this deadlock a client should release the lock on the finished task (playing the piano or eating at the buffet) before requesting another resource. It corresponds to avoid the incremental acquisition and is a way to avoid deadlock.

Question 4:

To describe this solution, I will first talk about the architecture that I choose then I will talk about the main class of this program and finally I will talk about the concurrency concept and the synchronization mechanism I used to attain a program with mutual exclusion, no deadlock , no livelock and fairness .

Firstly I made several folders in those folders all the files have the same package name.

In the asset's folder there is all the object I will use, for example there is a client a staff and the buffet. No class in this folder use any other classes written by me.

As for the buffet it uses a reentrantLock with fairness and synchronized method to protect its content. There is 2 method tryUse() to try acquiring the lock by returning a boolean and release() to unlock the object.

In the Event's folder there is all the event I used. I used event to characterises the interactions with the console, like a command, a message to write or the end of the program (Event-driven programming). None of this event have access to the console they mainly serve to implement method like "write()" in the MessageEvent classes to format the message in the output in function of his category and argument.

In the Util's folder there is some useful classes like a class to generate random number or probability. This class is useful in case I want to modify how I generate random number because I just need to modify 1 class and not all the class in the project that use random number. As for the other class in this folder, it is useful to generate execution time and to modify the speed of the simulation. In addition, this class is useful because I can balance the speed of the program by only modifying this class.

As for the Tread's folder I created 4 classes.
The first class "InputRunnable.java" is here to capture when the user write something in the console and to transform this input into either a QuitEvent or an InputEvent. This class use a volatile variable named running linked to the Boolean variable responsible for running the main program and is common for all the thread. When the user types a quit command a new QuitEvent is created and the program stopped by modifying the running variable.

The second class "WriterRunnable.java" is responsible to write message on the console to do that it uses the write method of the messageEvent and write it in the corresponding space.

The third class “StaffRunnable.java” is responsible for modelling staffs. This class use the synchronized keywords to access the buffet and choose the number of products it will brings.

The fourth class “ClientRunnable.java” is responsible for modelling clients. This class overrides the run method into using 2 methods consume(...) and entertain(...). It has a reentrantLock to protect the piano with fairness.

The entertain method uses another entertain method that need a probability to decide the next activity the client want to do between playing piano or listening to music. When the client wants to listen to music, he listens to music for an execution time generated.

When the client wants to play piano the piano’s availability is checked if the piano is unavailable a messageEvent is send and the piano’s availability is checked until the client can use the piano. Then another message is sent if the client has waited and he plays the piano for a set execution time and release the lock once the client finished playing.

The consume method uses another consume method that needs a probability and some quantities the decide what he wants to eat and returns a string telling what the client want. Then if the buffet is not available the client must wait until he can use it. Several messages are sent if the client needed to wait. When the clients have the lock for the buffet the program calls another method to take items from the buffet and keep the lock until the client is served, if the buffet have not any product for the client a message is sent. After taking item from the buffet the client eats or drink for a set time and then release the lock.

One way to upgrade this class is to implement an orderQueue with fairness to ensure any client don’t wait too much because a previous client made an enormous demand.

In the main class I first initialize my number and time generator then I introduce 2 blocking Queue to store all the events. These 2 blocking Queue are BlockingQueue<> interface implemented by the type LinkedBlockingQueue<>. One of these queues is for the input and quit event and the other is for the message events (Event-driven programming). I have done 2 of these queues because the input event and the quit event haven’t the write method and I don’t want to add this method in the IEvent interface because they both don’t need this method. I also use a volatile Boolean variable “running” because his value will be updated to all the tread in the program and all the class that have this value in argument of their constructor.

I used a method Init() to initialize the simulation without using thread because to initialize the output thread I need to initialize the buffet.

To store the client and the staff I used two vectors of thread because the vector can access, delete and add new element in an efficient way and the vector synchronized access to its data for every single method so there is no need to synchronize these 2 vectors when adding new threads (Vector synchronized).

After that I initialize every vector and the buffet with the value the user wants I used my input event to control the GUI.

As for the quit event I interrupt every thread I am using (input thread, output thread, staff's thread, client's thread) and to ensure that every thread is interrupted at the end of the program I use the join method on all the thread. And finally, I turn the running variable to false to avoid staying in the simulation loop.

The shared resources are the buffet, the piano the command Queue and the output Queue.

To ensure the mutual exclusion on the buffet I used the keyword synchronized for the staff's thread and a reentrant lock with fairness for the client's thread. I used the keyword synchronized for the staff's thread because the staff just need to add some product to the buffet which is a very fast operation so implementing fairness will not be significant. I used a reentrant lock with fairness for the client's thread because the client might have the lock on the buffet a lot more time than the staff. And in a cafe the customer is king so it's best if a staff wait to bring more items in the buffet than a client who needs to wait for the buffet to be available while having enough items.

To ensure mutual exclusion on the piano I used a reentrant lock with fairness to avoid having a client who is just waiting till the end of the program.

To ensure mutual exclusion on the command queue I used a LinkedBlockingQueue<> because it implements the interface BlockinQueue<> which is made for producer-consumer queue and thread safe (Doc BlockingQueue).

To ensure mutual exclusion on the message queue I used the same object as the command queue for the same reason.

The program is guaranteed to be deadlock free because the only resources that are prone to cause a deadlock are the piano and the buffet. But a deadlock cannot happen because there is no incremental acquisition for any of the shared resources.

To avoid livelock in the program I put in place the fact that when the piano or the buffet is unavailable the client's thread keep the lock on the shared resource. So, a situation of livelock is very unlikely to happen

In the program fairness is guaranteed because of the two reentrant locks initialised with fairness as for the staff his access is not implemented with much fairness but like I said earlier the client is king.

Reference List:

Lecture 2 Dr Daniele Scrimieri (2026) Lecture 2 – Synchronisation: Canvas

Lecture 4 Dr Daniele Scrimieri (2026) Lecture 2 – Deadlock: Canvas

Event-driven programming https://en.wikipedia.org/wiki/Event-driven_programming

Vector synchronized https://stackoverflow.com/questions/43015213/why-vector-class-in-java-collection-has-poor-performance-with-multi-threading

Doc BlockingQueue
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html