Question 1 Answer:

Synchronization primitives:

1. `volatile struct lock **bowlLocks;`

   An array of locks for the bowls, to enforce one creature eating at one bowl at one time.
2. `volatile struct lock *foodcourt_lock;`

   Lock for the eating room `foodcourt,` to facilitate the mutual exclusion.
3. `volatile struct cv *foodcourt_accessible;`

   Condition variable for the eating room `foodcourt`, re-acquiring the `foodcourt_lock` when `foodcourt` is eligible for access (when `foodcourt` contains only cats or mice.)

Shared variables:

1. `volatile struct queue *waiting_creatures;`

   A queue for creatures waiting to enter the eating room `foodcourt.`
2. `volatile int numEaters;`

   Number of creatures currently eating in `foodcourt.`
3. `volatile char currentEater;`

   The type of creature currently eating in `foodcourt.`

Question 2 Answer:

First, I used `foodcourt_lock` and `foodcourt_accessible` as the lock and cv to enforce mutual exclusion – when some creature tried to enter the `foodcourt`, first it would check whether the lock is available or not. Then, according to the variables `currentEater` and `waiting_creatures`, if `currentEater` is either null or equivalent to this creature, and `waiting_creatures` is empty, this creature would still hold the lock and continue on (enter `foodcourt` and increment `numEaters`), without waiting on cv; otherwise, it would wait on `foodcourt_accessible` for the signal to wake it up, and at the same time being queued on `waiting_creatures`. For the creatures inside `foodcourt`, they would hold a `bowlLock` assigned for each bowl if the lock had not been held. Finally, after all current eating creatures had done eating, by that time `numEaters` would be 0, which triggered the `cv_signal()` to signal next N cats or mice to enter `foodcourt`.

Question 3 Answer:

In my implementation, each bowl is assigned a different lock. So there is no chance for other guys to eat from the bowl while one guy is eating from this bowl. If creature A is currently occupying a bowl, another creature B who wants to occupy that bowl will continuously trying to require that lock until that lock is released, so that creature B can hold that lock and eat.

Question 4 Answer:

Assuming that there are several cats eating now, when a mouse thread hold the `foodcourt_lock`, it will check whether `currentEater` is type cat or mouse. When it finds out that `currentEater` is cat, it will be queued in `waiting_creatures`, and wait on `foodcourt_accessible` cv, until all eating cats are out of `foodcourt.` After that, that waiting mouse (or a bunch of waiting mice) will be signaled.

Question 5 Answer:

It is impossible for cats or mice to starve, because in my implementation, a cat or mouse that wants to eat should eventually be able to eat. All creatures (threads) will be queued (if necessary) according to their time of arrivals.

Question 6 Answer:

Efficiency:

My implementation guaranteed high level of efficiency. For example, initially when the queue is empty and no bowls are occupied, if a mouse comes along, it will directly go for one bowl whose index is equal to that mouse's index. If another mouse comes along, it will still go for another vacant bowl even if that first mouse is still there, so on so forth, until a cat comes along.

Fairness:

My implementation guaranteed very high level of fairness. Since cats thread and mouse threads come one after another, they will be queued according to their time of arrivals. This implementation enforces FIFO, so fairness is guaranteed.

There is no bias against cats or mice in terms of fairness.

Tradeoff between efficiency and fairness:

This implementation guaranteed very high level of fairness (almost absolute fairness), so it sacrifices some efficiency. There are cases where there are still some vacant bowls beside eating cats, but cats queued after mice won't have a chance to immediately go for those bowls before those mice.