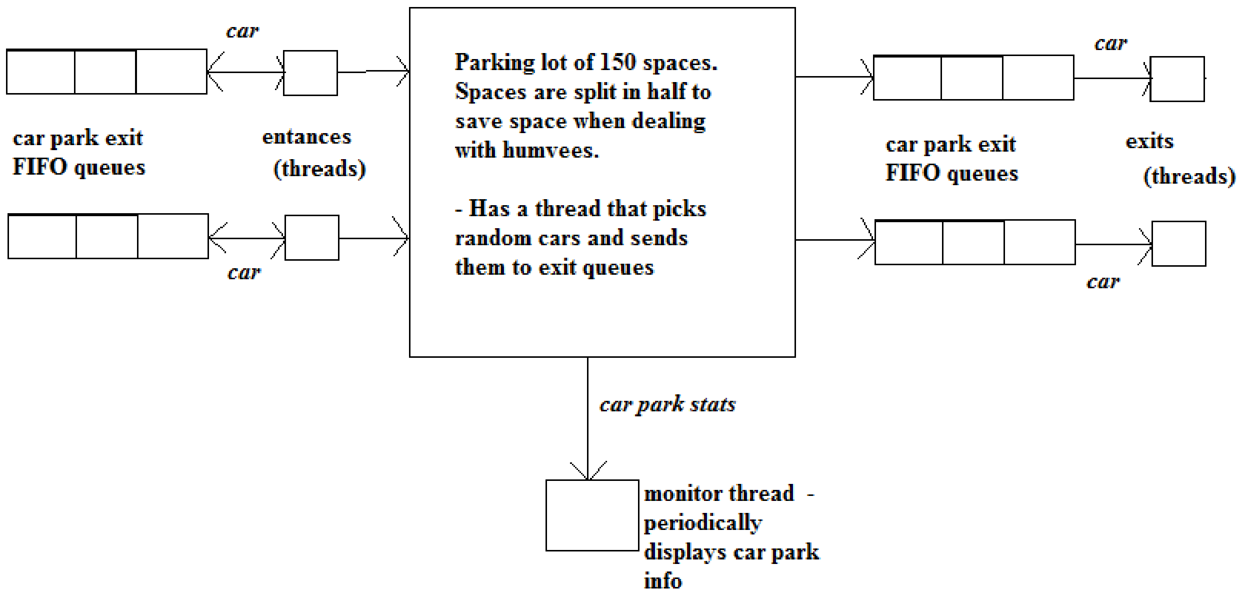
Overview

The car park is made up of 2 entrances, 2 exits and the main “car park” space. The entrances and exits are implemented using 4 threads - one for each entry point. There are two additional threads:

- the main program is a thread itself, it removes random cars from the car park and sends them to the exit queues
- the monitor thread periodically prints the car parks’ statistics.

See the diagram below for a visual representation of the system:


car
car park exit
FIFO queues
entances
(threads)
car
Parking lot of 150 spaces. Spaces are split in half to save space when dealing with humvees.
- Has a thread that picks random cars and sends them to exit queues
car
car park exit
FIFO queues
exits
(threads)
car
car park stats
monitor thread - periodically displays car park info

Physical constraints

When implementing the car park, some physical constraints had to be considered.

The entrance and exit queues cannot be sorted in any way

The car that is at the front of the queue has to be handled first since it’s blocking the entrance for the cars behind it, just like in a real car park.

The Humvees

The Humvees take up 1.5 spaces - this means that half-spaces will have to be considered. The program stores the newly arrived cars in an ordered fashion – imagine parking in space 1A, then another car parks in space 2A, and so on. In the real world, this would probably be more random, but for the sake of simplicity and speed the cars are put into ordered positions here.

Humvees provide an interesting problem – should the car park simulate them as “2-space” vehicles (as they would probably be treated in that way in the real-world – nobody likes to park in between the parking lines).

In my solution, I expanded the car park to have 300 half-spaces instead of the required 150 full-spaces. A Humvee that used to take 1.5 of a space now takes up 3 half-spaces and a standard-sized vehicle takes up 2. This solves the problem of leaving “gaps” in the car park and therefore wasting some space (0.5 space per every Humvee). The cars park next to each other regardless of the size of other cars or their positions – “if there is space, use it!”.

This approach creates a different problem – “fragmentation” in our data structure. When a Humvee is parked between two standard-sized cars and the Humvee leaves, it may be replaced by a standard-sized car. This leaves 1 space in our car park empty – remember, the smallest car requires 2 empty spaces to fit in.

Normally, in a digital environment these spaces could be removed by moving the data around, but since this program simulates a physical-world car park, it can not shuffle cars around – those gaps are simply ignored until they sort themselves out through normal operation (e.g. another Humvee parks in the old one’s place).

The exit queue

The exit queue only serves to clean up the program – the cars that are on the exit queue are no longer considered when the car park is looking for free spaces. At this stage, the cars that are on the exit queue have already left their parking spots and are lined up to leave the car park.

Detailed Implementation

The program utilizes reentrant locks and conditions to synchronize between the threads.

There are four different thread groups:

1. Entrance threads
2. Exit threads
3. Parking lot thread
4. Monitor thread.

To accommodate synchronization between these threads three locks are used:

1. Parking lock
2. Entrance lock
3. Exit lock

To accommodate communication between these threads three conditions are used:

1. Parking lot not full
2. Parking lot not empty
3. Exit queue not empty

Entrance threads require the **entrance lock** to check their own entrance queues and the **parking lock** to move cars from the entrance queue to the parking lot. The **entrance lock** is acquired first, then the thread attempts to acquire the **parking lock** as well and move the cars from the entrance to the car park. If the car park is full, the thread relinquishes the two locks and waits on the **parking lot not full** condition. If the thread succeeds in moving a car into the parking lot it signals all threads that may be waiting on the **parking lot not empty** condition.

Exit threads need the **exit lock** to remove cars from the exit queue.

The Parking lot thread needs the **parking lock** to check the state of the parking lot and the **exit lock** to move cars from the car park to the exit queues. The **parking lock** is acquired first, and if the car park is empty the lock is relinquished and the thread waits on the **parking lot not empty** condition. If the car park is successful in moving a car to one of the two exits (the actual exit is chosen randomly), it signals threads that may be waiting on the **parking lot not full** and **exit queue not empty** conditions.

The monitor thread needs the **parking lock** to access the parking lot's current stats.

Issues & Solutions

Fairness

Fairness is ensured in two ways:

- Similar threads wait on the same conditions – whenever it's relevant and makes sense, similar threads are made to wait on the same conditions.
- FIFO Fairness - the "fairness parameter" is used when constructing instances of ReentrantLock. According to the Java documentation for ReentrantLock, "when set true, under contention, locks favor granting access to the longest-waiting thread."

Starvation

Threads that cannot continue are forced to relinquish their locks and wait. In addition, threads are forced to relinquish the lock every time they complete one cycle of their execution and sleep for a period of time, giving other threads a chance to acquire the lock.

In addition to that, fairness (explained above) ensures that no thread will wait indefinitely while others keep getting the lock ahead of it.

Deadlock

Deadlock will not occur because no two threads are reliant on the same shared data in this program – if one thread is a producer, the other acts as the consumer in every relationship.

- Entrance relies on the car park being "not full", car park relies on the entrance supplying new cars. If the car park is full, entrance waits while the car park sends cars to the exit. If the car park is empty, it waits while the entrance sends it new cars.
- The exits rely on the car park to send them data, the car park does not rely on the exits for new data – if it can acquire the lock it can send cars to the exit queue.
- The monitor relies on the car park for statistics, the car park does not rely on the monitor.

Mutual Exclusion

All critical sections are protected by the 3 locks described before – no two threads can access the common shared variables at the same time.

The second problem

Whilst the second potion of the assignment is fairly similar to the first one, it does introduce some specific problems that require new solutions. These issues will be addressed here.

The second portion of the assignment introduces precedence – lecturers now have precedence for 70% of the spaces, with equal precedence for the remaining 30%.

Solution

The car park has been split into two sections – a section for "lecturers" and a common section.

The lecturers that arrive at the entrances are sent to the lecturers' section if that section isn't full, otherwise they are sent to the common section. If both sections are full they are made to wait until space frees up in either one.

The students' case is more complicated than that. If a student car arrives and the common section is empty, that car is sent to the common section. If the common section is full, the student is held back until enough space frees up in it. There is a third case – if two students are blocking both entrances (one student at each entrance) while waiting for the common section to free up, one of the students will be allowed into the lecturers' section (but only if the lecturers' section is not full).

This is another example of a physical constraint that dictates the behavior of the simulated system – cars waiting at the entrance (queue) cannot be moved around and if a student car isn't admitted into the lecturers' section it will be blocking other (lecturers') cars that are behind it.

Note: as long as the student section is full and a student is waiting at one entrance, with 10 lecturers waiting at the other entrance the 10 lecturers will be admitted into the lecturers' car park ahead of that one student.

Technical details

Note: the program has been split into two to differentiate between the two problems and preserve clarity in code – the majority of classes are the same in both cases.

The communication between the two entrance threads is handled using two public, static Boolean variables. Each of the entrances has its own public Boolean variable and if that entrance is blocked by a student (as described in the section above) its Boolean is set to true to signal that fact to the other entrance. If both entrances are signaling that they're blocked by students, the program will attempt to admit one of the students into the lecturers' section of the car park.

When a student is successfully admitted into any car park, the associated Boolean is set to false.