**Concurrent Systems Modelling in JCSP**

**An Automated Car-Parking System**

Technical Report by

Glenn Healy

SR017409

[**g.healy@student.reading.ac.uk**](mailto:g.healy@student.reading.ac.uk)

**Introduction**

CSP (Communicating sequential processes) is a way of formally describing the patters of interactions between processes in Concurrent systems. Or less formally, it is a language for modelling concurrent systems. The idea being that you can execute and finish various parts or units of a program/system in any order without affecting the outcome.

This allows you to run longer portions of the program first, and more usefully, develop Parallel systems to execute multiple portions of the system at once. Cutting down compute time.

With this in mind, the goal of this work was to develop an understanding of CSP systems and concepts, while developing a basic system. Allowing us to model the behaviours and patters of interactions between the processes of a test case, this being an automated car parking system. For this, we made use of a CSP library for one of the OOP/concurrent capable languages (in this case JCSP for Java) and developed the basic behaviour of the system making use of the tools in the JCSP library to pass the relevant messages through the various modules/units of the system via channels.

**CSP Modelling/Formalism**

To get going with this, I spent some time working out the basic behaviours expected, and thus, the basic class/unit breakdown:

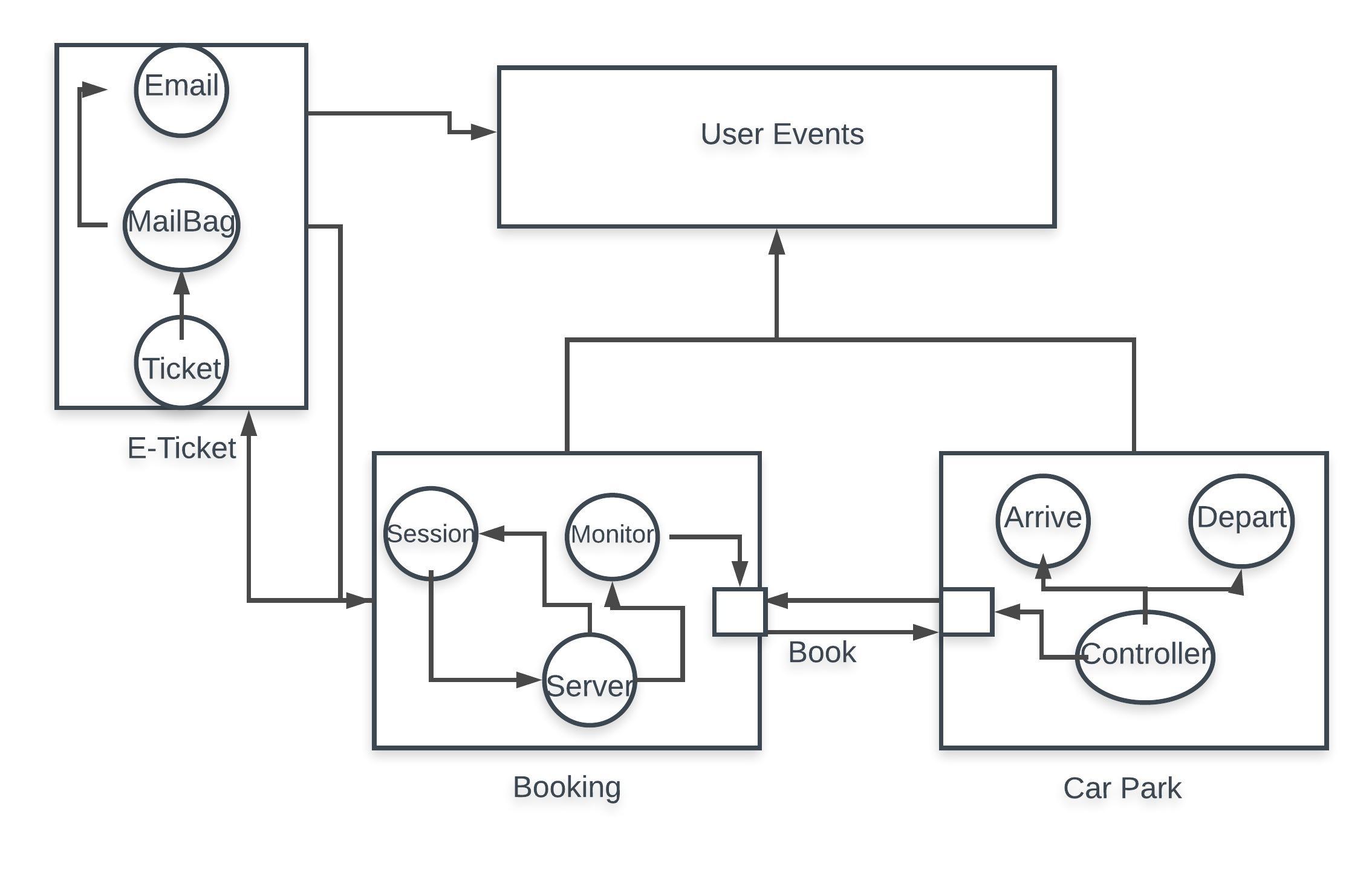


Figure 1 - Basic Process flow diagram

From here, came working out the basic channels and concurrency behaviour of the system, which is modelled using the standard Formalism of CSP, for this I had to further refine my processes:

*Booking:*

*Book*

*e-Ticket*

*Carpark:*

*Arrive*

*Depart*

*Controller*

*e-Ticket:*

*Ticket*

*Email*

From here, came breaking down the basic model of the behaviour/flow of events around the system:

*P = a -> b -> c | c | a -> c*

Booking = Book -> e-Ticket -> Carpark

e-Ticket = Book -> Ticket -> Email |

Carpark = Book -> e-Ticker -> Controller {Ticker -> Arrive -> Depart}

Controller = Ticket - > Arrive -> Depart

Class breakdown:

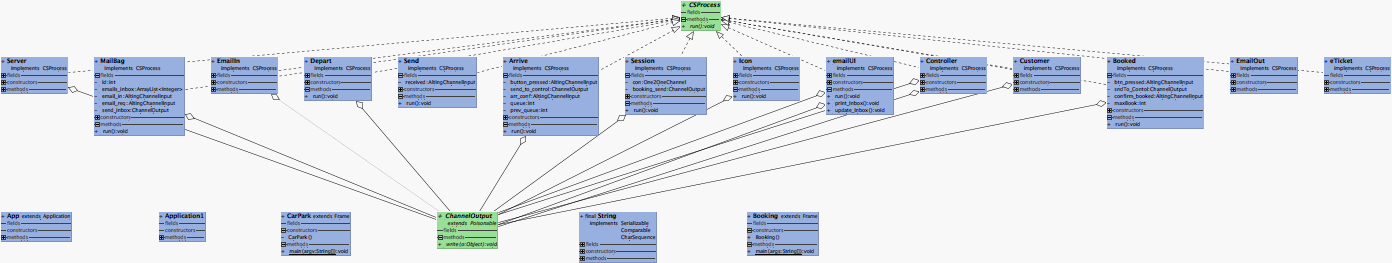
****

Figure 2 - UML Diagram (class diagram)

**Technical Implementation & Testing**

The technical implementation is the part of this where I struggled most. While I understood the basic theory pretty well, I found it hard to really grasp the practical end of this and found it hard to really understand JCSP and its syntax.

However, after several hours of Googling and a lot of trying and error I was able to get something together that vaguely meets the definition of CSP and Concurrent systems:

As per my class diagram, the project is broken down into three main sections:

*Booking*

*e-Ticket*

*CarPark*

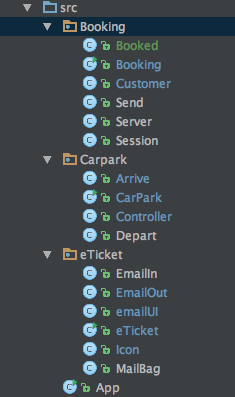
As per the following image:

Figure 3 - Class breakdown

This meant that I was able to break down each portion of the program into separate units and work on each independently from one another and have each working one by one. Meaning that each module works individually and is not entirely reliant on the next.

Each class is in a folder relating to the module is work/belongs to. This is to help keep the process of

Developing the program clean and tidy.

For example, the Booking module can run without the CarPark or E-Ticket modules:

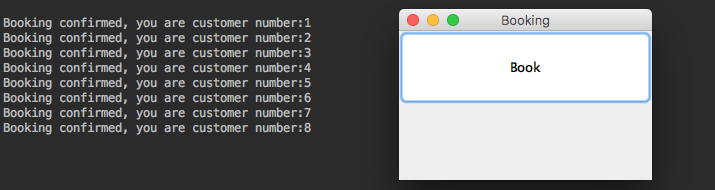


Figure 4 - Booking

This allows the user to book a space in the carpark, and will not allow more than the max number of parking spaces:

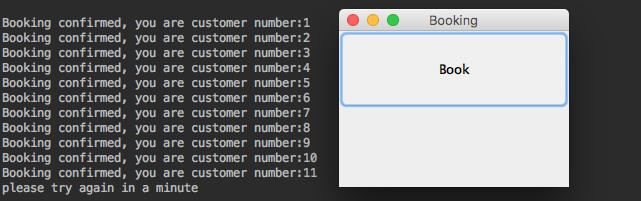


Figure 5 - Booking - max hit

It will simply break from the loop and await the customer to re-connect. The next section that I worked on was the Carpark and the arriving and departing events/processes.

This meant that I had to have it talk to the booking module and ensure that the basic behaviour met the design I had come up with, this breaks down into the following sections:

CarPark:

Carpark

Controller

Arrive

Depart

The Carpark module, initiates the controller module, which handles the flow of information/messages via channels between arrive, depart and booking.

When the Carpark module runs, it initiates a **new CSProcess** (parallel process) and carries out a channel2channel message to the Arrive and depart processes via the controller:

// Processes  
 Arrive arrive = new Arrive(arrive\_event.in(), arrival.out(), arrival\_confirm.in());  
 Controller control = new Controller(arrival.in(), departure.in(), arrival\_confirm.out(), departure\_confirm.out());  
 Depart depart = new Depart(depart\_event.in(), departure.out(), departure\_confirm.in());  
// Book book = new Book(book\_event.in(), booking.out(), book\_confirm.in());  
  
 new Parallel(new CSProcess[] { activeClosingFrame, arrive\_active, depart\_active, arrive, control, depart }).run();

This is started by the carparks main() function:

public static void main(String argv[]) {  
 CarPark cp = new CarPark();  
 Booking bk = new Booking();  
 }  
}

The booking module is broken down into two. The fist is the booking class, that initiates the channel between the Carpark and the booked class.

It also contains the information for the GUI (book button). This simply creates the process for booking the space by calling the booked class which initiates several instances of the Customer class and placing them into a new parallel process. And calls the send method, which sends the notification of the booking to the controller:

One2OneChannel internet = Channel.*one2one*(new OverWriteOldestBuffer(65645));  
 One2OneChannel booking\_send = Channel.*one2one*(new OverWriteOldestBuffer(64645));  
  
 Server server = new Server(internet.in(), booking\_send.out());  
 Send send = new Send(booking\_send.in());  
  
  
 Customer customer = new Customer(internet.out());  
 Customer customer2 = new Customer(internet.out());  
 Customer customer3 = new Customer(internet.out());  
 Customer customer4 = new Customer(internet.out());  
 Customer customer5 = new Customer(internet.out());  
  
 //starting the process  
 new Parallel(new CSProcess[] {server, send, customer, customer2, customer3, customer4, customer5}).run();  
  
}

Where by the Run method creates a new JCSP Guard and prints out the traces of the booking confirmation and the customers booking number.

This is also how the booking method can/will get called upon system starting. From here, the system can handle all incoming and outgoing (arrive and depart) cars (messages via channels). It will allow for cars to come in and go, but will then develop a queue (buffer) when the number of messages (cars) exceeds its max (10):

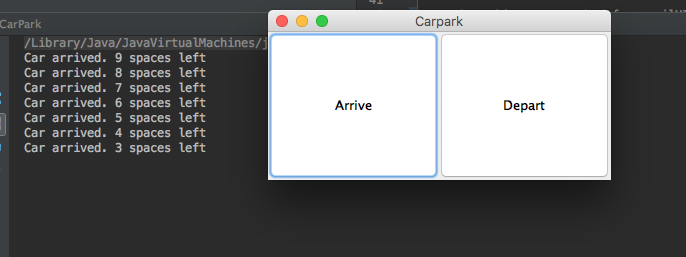


Figure 6 - Arrive/Depart

You can simply press the Depart button to leave or the arrive button to add cars. When 10 cars are hit it will prevent more arriving, but will add to the queue:

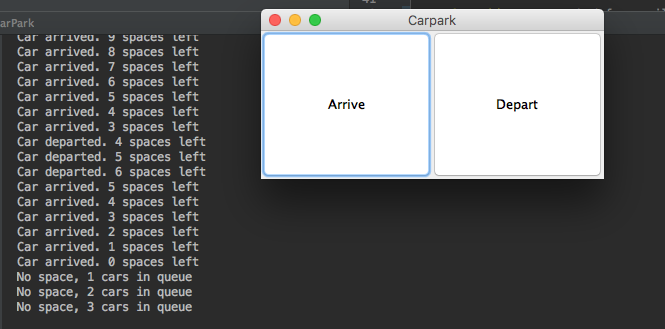


Figure 7 - Arrive - Queue

When No cars are in the carpark, it will trace out a message indicating that no cars are in he carpark:

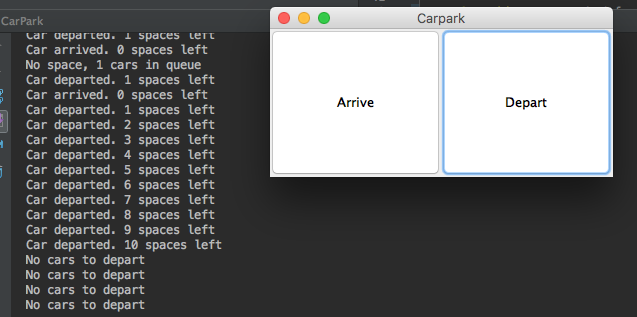


Figure 8 - Depart - CP empty

The next and final portion to develop was the e-Ticket system which would take the booking and generate messages to send out a ticket and an “email” which was basically an acknowledgment of the booking.

This was modelled on the “Mail Tool” design provided in the lecture slides. This comprises of the following sections:

Mail Tool:

Mail Bag

Unit

Dispatch

The above is a rough outline of how the system ended up. The mail tool works by creating a UI for the email client which dictates the behaviour of the email client and sets up the channels for the messages to and from the booking link and the customer (prints traces):

//constructor  
public emailUI(One2OneChannel[] connection\_inf, One2OneChannel next\_event2, ChannelOutput dispatch\_email2) {  
 Check\_email = connection\_inf[0].out();  
 get\_Email = connection\_inf[1].in();  
  
 next\_event = next\_event2;  
  
 this.send\_Email = dispatch\_email2;  
 buffer = new ArrayList<Integer>();  
 Index = 0;  
  
}

This is also where the two methods for controlling the inbox (buffer) are created:

public void print\_Inbox() {  
 StringBuilder output = new StringBuilder();  
  
 for (int i = 0; i < buffer.size(); i++) {  
 if (Index == i)  
 {  
 output.append(buffer.get(i));  
 } else {  
 output.append(buffer.get(i));  
 }  
 }  
 System.*out*.println(output);  
}  
  
public void update\_Inbox() {  
 Check\_email.write("get-email");  
 buffer = (ArrayList<Integer>) get\_Email.read();  
 System.*out*.println("Got" + buffer.size() + "emails");  
}

This is handled by the MailBag class which declares a new instance of “internet” and simply runs through a loop which constantly checks for new emails in the inbox and buffer and handles them accordingly.

For this to run, the e-Ticker class has three options:

Arrive

Get

Send

These buttons initiate a new instance of the processes they are tied to and call their respective methods:

// Processes  
EmailIn arrive = new EmailIn(arrive\_event.in(), arrival.out());  
MailBag mailbag = new MailBag(arrival.in(), internet);  
GetMail get = new GetMail(getmail\_event.in(), internet, send\_event, dispatch\_email.out());  
EmailOut dispatch = new EmailOut(dispatch\_email.in());

new Parallel(new CSProcess[] { activeClosingFrame, btn\_arrive, btn\_getmail, btn\_send, arrive, mailbag, get, dispatch }).run();

Where:

Emailin = Arrive

EmailOut = Send

GetMail = Getmail

Upon getting mail, the system will print this trace:

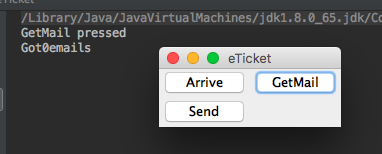


Figure 9 - get mail - e-Ticket

However, the behaviour for the other two buttons does not work due to some issues I was having generating the e-Tickets. Thus, the tickets are only partially implemented. As I was unable to get the behaviour working as desired. And when Arrive or Send are pressed the system shall crash. Requiring a restart of the module.

However, the base of the system is there, I was just unable to work out how to get the tracing for these to work in time due to some issues I was having generating the messages via the channels. I was also unable to get this working when the booking is done. Meaning that the bookings and the e-tickets are not linked in any way and the e-Ticket module is entirely separate from the system.

**Discussion on Correctness of Model/Implementation**

Comparing my implementation to the information delivered in the lectures and the information I was able to find online during my development, I feel the overall design meets the concepts of CSP and the Traces model well. When a module is called and the corresponding traces will reflect the process flow of the project, which seem to fit the logic of the traces model or less.

**Lock-Free/Wait=Free Concurrency in Java**

*(Note: I ran out of time to develop any actual prototypes for this)*

Lock-free algorithms/systems allow the CPU to continue to carry out useful work during the execution of the algorithm, however its own execution can/may be blocked to allow that to happen. This means that, should anything need to be carried out by the system, the process will be stalled while the CPU carries this out.

Whereas, wait-free prevents any process from blocking and/or stalling any other process. Meaning that each process will and can continue to be executed by the CPU without risking stalling something more important. Wait-free algorithms often have higher guarantees than lock-free algorithms and allow for high-throughput without sacrificing for high latency of any operation.

However, they are much harder to implement than Lock-free, and by thus are less common in none CPU urgent algorithms/processes. Making Lock-free far more common.

When looking at lock-free vs wait-free in Java (specifically java.util.concurrent) we can see that there are several options for lock-free algorithm implementations, most notably the java.util.concurrent.atomic package is an example of lock-free against single variables. Allowing for each process to complete against that single variable on the CPU.

Whereas as of Java 7 ConcurrentLinkedQueue is an example of a package that is wait-free. Which allows for an unbounded thread-safe queue on linked nodes to be executed.

This is a good option for a system where many of the threads will share access to a common set of variables. However, due to the asynchronous nature of the queues, may result in it not being the best decision for the carpark program.

The changing/variable nature of the queue size also presents issues for the potential use in the counter program. It also does not allow for null pointers/elements.

**Conclusions**

Looking through what I have done, I am proud of the work and parts of the implementation I was able to complete/get working given my newness to both multithreading applications, concurrent systems design and CSP

I feel that although the system does not work fully or exactly as desired, I think that it meets the basic definitions of correctness as defined in the lectures.

In hind site, my lack of experience with JCSP and CSP concepts made this assignment tricky, and resulted in the program being clunky and having several issues littered throughout. However, the design I still feel is adequate and meets the goals of what I set out to do. And given more time and the knowledge I have now, I feel I would be able to apply this concept far easier.

Regarding CSP and JCSP. I don't quite see the benefit of having to learn an entire new library to simply apply the logic of a non-locking design using the traces model. I feel that much of what has been implemented in this could have been done much easier in java.util.concurrent as it would have removed the need for quite so many channels. And allowed me to simply declare the threads I wanted as I wanted them. So, overall I feel the JCSP does not really help in the development of the concepts presented in the lectures.

**References**

Used for JCSP library:

<https://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp-1.1-rc4/jcsp-doc/>

<https://www.cs.kent.ac.uk/projects/ofa/jcsp/cpa2007-jcsp.pdf>

Overview of JCSP:

<https://www.ibm.com/developerworks/java/library/j-csp2/>

Similar concept using Java.util.concurrent – used for modelling program & comparing wait-free and lock-free:

<https://github.com/cerealb0x/ParkingLot/blob/master/src/parking/ParkingSystem.java>

Java AWT guide:

<https://www.javatpoint.com/java-awt>

Concurreny guide:

<http://concurrencyfreaks.blogspot.co.uk/2013/05/lock-free-and-wait-free-definition-and.html>

Wait-Free library for java.util.concurrent:

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html>

**Appendix**

Links to project:

GitHub link to project:

<https://github.com/glennismade/ConcurrentSystems>

**Entire project folder on blackboard under:**

Concurrent-systems.zip