# Project Summary

Process Scheduling is an essential part of any operating system. This becomes even more important when two or more processes make use of shared resources, they must be scheduled such that each process receives the most up-to-date version of that shared resource.

The model aims to use propositional logic and deduction to be able to identify different scheduling possibilities for a set of processes using a shared resource. This project specifically focuses on a sample set of processes and shared resources, but this can be extended for more processes and more resources

# Propositions

## Description

The project will focus on two processes and two shared resources. Each process will have two code segments where a shared resource is being accessed, this is called a critical section. A critical section can make use of one or both shared resources.

## Process 1 Critical Section Propositions

Process 1 has two critical sections, each critical section can either use resource 1, resource 2 or both. The following propositions cover every possibility for the critical sections:

a11 = Var('a11') *#true if p1\_crit\_sect1 uses r1*

a12 = Var('a12') *#true if p1\_crit\_sect1 uses r2*

a21 = Var('a21') *#true if p1\_crit\_sect2 uses r1*

a22 = Var('a22') *#true if p1\_crit\_sect2 uses r2*

## Process 2 Critical Section Propositions

Process 2 has two critical sections, each critical section can either use resource 1, resource 2 or both. The following propositions cover every possibility for the critical sections:

b11 = Var('b11') *#true if p1\_crit\_sect1 uses r1*

b12 = Var('b12') *#true if p1\_crit\_sect1 uses r2*

b21 = Var('b21') *#true if p1\_crit\_sect2 uses r1*

b22 = Var('b22') *#true if p1\_crit\_sect2 uses r2*

## Resource 1 Scheduling Propositions (In progress)

//the q propositions

## Resource 2 Scheduling Propositions (In progress)

// the p propositions

# Constraints

*List of constraint types used in the model and their (English) interpretation. You only need to provide one example for each constraint type: e.g., if you have constraints saying “cars have one colour assigned” in a car configuration setting, then you only need to show the constraints for a single car. Essentially, we want to see the pattern for all of the types of constraints, and not every constraint enumerated.*

//intend on filling this in after our feedback, current constraints are in newtestrun.py

# Model Exploration

*List all the ways that you have explored your model – not only the final version, but intermediate versions as well. See (C3) in the project description for ideas.*

*//draft will replace symbols with appropriate in final doc, just filling out document with content for feedback*

## Original Version

Version had only a few propositions, aim of model wasn’t really clear, it was the base version:

**Propositions**

Each process has a section where they access the shared resource. This is called the critical section.

p1 - true if process 1 is in their critical section  
p2 - true if process 2 is in their critical section  
w1 - true if process 1 is waiting to enter critical section  
w2 - true if process 2 is waiting to enter critical section

**Constraints and Models**

If process 1 is waiting, that means they are not accessing their critical section

w1 → !p1  
w2→!p2

Both processes cannot be in their critical section

(p1 ) → (!p2 )  
(p2) → (!p1)

If process1 is waiting and the shared resource is not being accessed then it should enter its critical section

(w1 \* !p2) → p1  
(w2 \* !p1) → p2

The worst case model, ie the model that shouldn’t happen is that all processes are waiting:

(w1 \* w2 \* !p1 \* !p2)

## Updated Complexity (testrun.py)

Using D1 Feedback, the model was updated to now model the scheduling of process code segments (critical sections on the cpu). Code segment propositions were generated (and are the same descriptions as those above). However, scheduling propositions are introduced:

|  |
| --- |
| *# ------------------------------RESOURCE 1 SCHEDULING PROPOSTIONS------------------------------------------*  *# So originally we had the following:*  *# q was used to represent resource scheduling for resource 1, and the number subscripts are for:*  *# (1) a11, (2) a21, (3) b11, (4) b21*  *# Since all the above propositions stand for critical sections that make use of the same resource r1,*  *# then we would have to schedule them (ie they cannot run in parallel)*  *# the qs represent the different ways/possibilities of scheduling:*  q12 = Var('q12') *#true if a11 is scheduled before a21,*  q13 = Var('q13') *#true if a11 is scheduled before b11*  q14 = Var('q14') *#true if a11 is scheduled before b21*  q21 = Var('q21') *#true if a21 is scheduled before a11*  q23 = Var('q23') *#true if a21 is scheduled before b11*  q24 = Var('q24') *#true if a21 is scheduled before b21*  q31 = Var('q31') *#true if b11 is scheduled before a11*  q32 = Var('q32') *#true if b11 is scheduled before a21*  q34 = Var('q34') *#true if b11 is scheduled before b21*  q41 = Var('q41') *#true if b21 is scheduled before a11*  q42 = Var('q42') *#true if b21 is scheduled before a21*  q43 = Var('q43') *#true if b21 is scheduled before b11*  *# ------------------------------RESOURCE 2 SCHEDULING PROPOSTIONS------------------------------------------*  *#similar thing for resource 2, now r2 is represented by p*  *# (1) a12, (2) a22, (3) b12, (4) b22*  p12 = Var('p12')  p13 = Var('p13')  p14 = Var('p14')  p21 = Var('p21')  p23 = Var('p23')  p24 = Var('p24')  p31 = Var('p31')  p32 = Var('p32')  p34 = Var('p34')  p41 = Var('p41')  p42 = Var('p42')  p43 = Var('p43') |

In this case, the scheduling propositions addressed all possible ways two code segments can be scheduled. So for example, q12 specifies that a11 is put before a21, q21 specifies that a21 is put before a11.

This also resulted in different constraints. The first constraint is applied to the scheduling proposition possibilities. In the format of q21 >> ~q12, q12 >> ~q21. This constraint is introduced to limit the scheduling propositions. That is, if a11 and a21 are scheduled, then it can either be a11 before a21 or a21 before a11, it can never be both. Therefore, if one case is true, then it must mean that the other case is false:

*# if a11 is scheduled before a21 then it cant be the case*

*#that a21 is scheduled before a11, not possible*

E.add\_constraint(~q12 | ~q21)

E.add\_constraint(~q13 | ~q31)

E.add\_constraint(~q14 | ~q41)

E.add\_constraint(~q23 | ~q32)

E.add\_constraint(~q24 | ~q42)

E.add\_constraint(~q34 | ~q43)

*#same for resource 2*

E.add\_constraint(~p12 | ~p21)

E.add\_constraint(~p13 | ~p31)

E.add\_constraint(~p14 | ~p41)

E.add\_constraint(~p23 | ~p32)

E.add\_constraint(~p24 | ~p42)

E.add\_constraint(~p34 | ~p43)

These are reduced to nnf form, therefore the biimplication is preserved in the commutative property of the OR function.

We also have the constraint that ties the segment propositions to the scheduling propositons. If two segments make use of the same resource, then they must be scheduled and they have two different scheduling possibilities. For example, if process 1 critical section 1 and process 1 critical section 2 make use of resource 1, then they must be scheduled. Process 1 critical section 1 using r1 means a11 is true and process 1 critical section 1 means a21 is true. Therefore if a11 and a21 are true, then they must be scheduled. Two possible ways: a11 before a21 (q12) or a21 before a11 (q21), hence we have the full constraint: (a11 & a21) >> (q12 | q21). Extending to the other segment propositions:

E.add\_constraint( (~a11 | ~a21) | (q12 | q21) )

E.add\_constraint( (~a11 | ~b11) | (q13 | q31) )

E.add\_constraint( (~a11 | ~b21) | (q14 | q41) )

E.add\_constraint( (~a21 | ~b11) | (q23 | q32) )

E.add\_constraint( (~a21 | ~b21) | (q24 | q42) )

E.add\_constraint( (~b11 | ~b21) | (q34 | q43) )

E.add\_constraint( (~a12 | ~a22) | (p12 | p21) )

E.add\_constraint( (~a12 | ~b12) | (p13 | p31) )

E.add\_constraint( (~a12 | ~b22) | (p14 | p41) )

E.add\_constraint( (~a22 | ~b12) | (p23 | p32) )

E.add\_constraint( (~a22 | ~b22) | (p24 | p42) )

E.add\_constraint( (~b12 | ~b22) | (p34 | p43) )

These have also been reduced to nnf form.

## Alternate Scheduling Propositions (Newtestrun.py)

This is a different version of the scheduling propositions. Rather than the scheduling propositions addressing every scheduling possibility between two code segments that use the same resource, it simply is used to determine if the two segments need to be scheduled.

For example, in the previous example: q12 and q21 stood for the different scheduling possibilities for a11 and a21. In this case, we would only have q12, and q12 represents that the process 1 critical section 1 and process 1 critical section 2 (where both use resource 1) would have to be scheduled:

*# Rather than having propositions for the specific ways that the segments can be scheduled*

*# we can just have propositions for if there is a scheduling required, this will cut the*

*# current propositions in half. So instead we have the following, using the same notation:*

*# (1) a11, (2) a21, (3) b11, (4) b21*

q12 = Var('q12') *#true if a11 and a21 need tscheduled*

q13 = Var('q13') *#true if a11 and b11 need to be scheduled*

q14 = Var('q14') *#true if a11 and b21 need to be scheduled*

q23 = Var('q23') *#true if a21 and b11 need to be scheduled*

q24 = Var('q24') *#true if a21 and b21 need to be scheduled*

q34 = Var('q34') *#true if b11 and b21 need to be scheduled*

*# This will generate a new version of constraints*

*# ------------------------------RESOURCE 2 SCHEDULING PROPOSTIONS------------------------------------------*

*# Applying the same change from resource 1. Now p represents resource 2 and the subscripts references are*

*# (1) a12, (2) a22, (3) b12, (4) b22*

p12 = Var('p12') *#true if a12 and a22 need to be scheduled*

p13 = Var('p13') *#true if a12 and b12 need to be scheduled*

p14 = Var('p14') *#true if a12 and b22 need to be scheduled*

p23 = Var('p23') *#true if a22 and b12 need to be scheduled*

p24 = Var('p24') *#true if a22 and b22 need to be scheduled*

p34 = Var('p34') *#true if b12 and b22 need to be scheduled*

This therefore makes changes to our constraints. Instead of having to address the two possible scheduling propositions if two critical sections use the same resource, we now simply just know that they would need to be scheduled, therefore: (a11 & a21) >> q12. Where q12 now means that a11 and a21 must be scheduled.

However, because of the nature of the implication, if the antecedent is false, then the implication will always be true despite the value of the consequent. This would mean that if a11 and a21 is false, then the implication will still hold even if q12 is true, which isn’t logical since the critical sections would not have to be scheduled. Therefore we would need a bimplication: (a11 & a21) <<>> q12. Implemented in code as two implications. Extending for the different scheduling propositions:

E.add\_constraint( (~a11 | ~a21) | q12 ) *#(a11 & a21) >> q12*

E.add\_constraint( ~q12 | (a11 & a21) ) *#q12 >> (a11 & a21)*

E.add\_constraint( (~a11 | ~b11) | q13 ) *#(a11 & b11) >> q13*

E.add\_constraint( ~q13 | (a11 & b11) ) *#q13 >> (a11 & b11)*

E.add\_constraint( (~a11 | ~b21) | q14 ) *#(a11 & b21) >> q14*

E.add\_constraint( ~q14 | (a11 & b21) ) *#q14 >> (a11 & b21)*

E.add\_constraint( (~a21 | ~b11) | q23 ) *#(a21 & b11) >> q23*

E.add\_constraint( ~q23 | (a21 & b11) ) *#q23 >> (a21 & b11)*

E.add\_constraint( (~a21 | ~b21) | q24 ) *#(a21 & b21) >> q24*

E.add\_constraint( ~q24 | (a21 & b21) ) *#q24 >> (a21 & b21)*

E.add\_constraint( (~b11 | ~b21) | q34 ) *#(b11 & b21) >> q34*

E.add\_constraint( ~q34 | (b11 & b21) ) *#q34 >> (b11 & b21)*

*#-------------------------RESOURCE 2 CONSTRAINTS------------------------------*

E.add\_constraint( (~a12 | ~a22) | p12 ) *#(a12 & a22) >> p12*

*#if p1\_crit\_sect1 uses r2 (a12) and p1\_crit\_sect2 uses r2 (a22)*

*#then they must be scheduled (p12)*

E.add\_constraint( ~p12 | (a12 & a22) ) *#p12 >> (a12 & a22)*

E.add\_constraint( (~a12 | ~b12) | p13 ) *#(a12 & b12) >> p13*

E.add\_constraint( ~p13 | (a12 & b12) ) *#p13 >> (a12 & b12)*

E.add\_constraint( (~a12 | ~b22) | p14 ) *#(a12 & b22) >> p14*

E.add\_constraint( ~p14 | (a12 & b22) ) *#p14 >> (a12 & b22)*

E.add\_constraint( (~a22 | ~b12) | p23 ) *#(a22 & b12) >> p23*

E.add\_constraint( ~p23 | (a22 & b12) ) *#p23 >> (a22 & b12)*

E.add\_constraint( (~a22 | ~b22) | p24 ) *#(a22 & b22) >> p24*

E.add\_constraint( ~p24 | (a22 & b22) ) *#p24 >> (a22 & b22)*

E.add\_constraint( (~b12 | ~b22) | p34 ) *#(b12 & b22) >> p34*

E.add\_constraint( ~p34 | (b12 & b22) ) *#p34 >> (b12 & b22)*

## Revised Constraints (In progress)

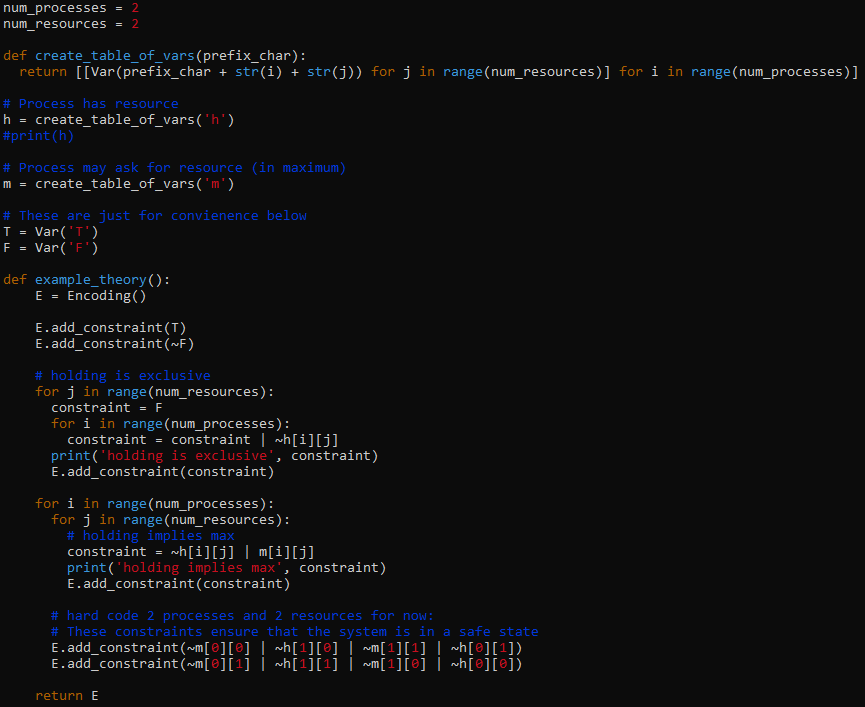
## Test Implications (In progress)

## Alternate problem with more complex constraints (deadlock.py)

Just in case the above problems are not complex enough, we’ve also looked into modeling another problem altogether: predicting deadlock. In ELEC 377, we learned a technique for avoiding deadlock. If each process declares ahead of time the maximum resources it may have throughout its execution, the operating system can systematically run processes in an order that cannot result in deadlock. We say that a state is allowed if there is an order in which the processes can be run such that there is no possibility of deadlock occurring.  
“h” is a table of variables indexed by process number and then resource number. A true value means that the process is currently holding the resource.  
“m”, similarly, contains variables that indicate that a process may request a resource.

The code below constrains the variables in “h” and “m” to only combinations that are in a safe state, using 2 processes and 2 resources. There are several constraints on the system. First, if one process is holding a resource, then no other process can be holding it. Next, if a process is holding a resource, then the resource must be in its maximum set of resources. Finally, we ensure that the system is in a safe state by preventing circular wait.

Extensions to this problem involve determining which processes are safe to run (rather than if the system is in a safe state) and writing code to generate the constraints given any number of processes and resources. (The later especially is very difficult.)



# First-Order Extension

*Describe how you might extend your model to a predicate logic setting, including how both the propositions and constraints would be updated.* ***There is no need to implement this extension!***

# Useful Notation

*Feel free to copy/paste the symbols here and remove this section before submitting.*

# Feedback Request

* How is our model complexity? We are planning to go with testrun.py as it seems more complex than the alternate version, but we are still wondering if it suffices in complexity
* If testrun.py is not complex enough, what do you think of the idea of modelling deadlock?
* Not sure what should be included in the Jape Proofs file. Are we trying to prove that our constraints are valid or that the solutions provided are valid? If we were trying to prove that how would we go about it?
* Is this all valid exploration for our model? I know there are examples provided and we are yet to test our partial assignments thoroughly, just wanted to know if this is all valid content
* How would we properly include code in the document, would you prefer a line reference to files on the repository e.g. testrun.py: line 10 - 24