# Project Summary

A process is an instance of a program running on a computer. There can be several processes running on a computer but only a few can utilize the CPU at a given time. Therefore, the operating system must be able to schedule them to optimize CPU usage.

The goal of this project is to use propositional logic to generate possible process schedules for a set of processes using a set of resources on a set of processors. The number of processes, resources and processors can be changed to match the needs of different cases.

# Propositions

## Description

A process may contain sections of code where it accesses different shared resources. These are called critical sections and are represented by code blocks. Each process can have one or more code blocks that make use of zero or more shared resources. Furthermore, processes do not need to have the same number of code blocks.

Time slots are used to break down sections of time for process code blocks. The number of time slots for a given simulation is equal to the total number of code blocks in the simulation. This is to encapsulate the worst possible scheduling case where all code blocks are scheduled on one processor.

There are also several processors where process code blocks can be scheduled to execute simultaneously. This is overlayed on top of the time slots, meaning that if there are time slots and processors, then there will be schedule positions i.e. positions where a process code block can be scheduled.

The number of processors, processes, code blocks for each process and resources can be specified to fit the requirements of the user. Furthermore, which shared resources are used in the different process code blocks can also be specified.

## Schedule Propositions

The schedule propositions are used to cover all scheduling possibilities a simulation can have. They are of the format:

Where represents the index of the time slot, represents the index of the processor, represents the index of the process and represents the index of the code block for the given process. This proposition is true if code block of process is scheduled in time slot on processor .

Each code block has a proposition for every schedule position in the simulation. Therefore, for a simulation with total code blocks and processors will have schedule propositions. This is because there total possible spots in the schedule and each code block will have a proposition for each spot, therefore with total code blocks, there are total propositions.

These truth values of these propositions are the result of the model. They are used to construct the schedule of processes on the different processors. For a simulation with 2 processors, 2 processes each with one code block, Table 1 below shows the schedule propositions generated.

Table 1 - Schedule Propositions for Sample Simulation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Time Slot | Processor | Process | Code Block | Proposition |
| T0 | P0 | Process 0 | Code Block 0 | schedule\_0\_0\_0\_0 |
| Process 1 | Code Block 0 | schedule\_0\_0\_1\_0 |
| P1 | Process 0 | Code Block 0 | schedule\_0\_1\_0\_0 |
| Process 1 | Code Block 0 | schedule\_0\_1\_1\_0 |
| T1 | P0 | Process 0 | Code Block 0 | schedule\_1\_0\_0\_0 |
| Process 1 | Code Block 0 | schedule\_1\_0\_1\_0 |
| P1 | Process 0 | Code Block 0 | schedule\_1\_1\_0\_0 |
| Process 1 | Code Block 0 | schedule\_1\_1\_1\_0 |

## Process Requirement Propositions

These are propositions used to specify the resources being used by a code block of a process. They are of the format:

Where is the index of the process, is the index of code block and is the index of the shared resource. This proposition is true if code block of process uses/accesses shared resource .

For total code blocks and resources, there will be process requirement propositions. Their truth values can be modified to match the requirements of the simulation.

The truth values of the propositions are added to the model as constraints based on the user specification. The user specifies which resources a code block uses in the *cb\_resc\_use\_array­*, as shown in Figure **1** below.

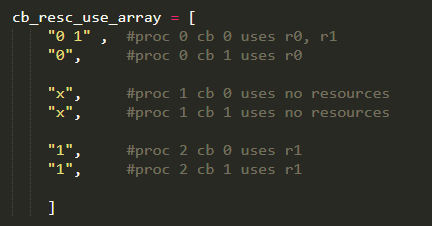


Figure 1 - Resource Usage Specification Array

The user specifies the index of the resources used by process code blocks or an ‘x’ if the code block uses no resources. This array is then used to convert these resource specifications to truth values of the resource propositions and then add them as constraints to the model. Table 2 below shows the truth values of some resource propositions for this case (assuming only two shared resources).

Table 2 - Resource Proposition Truth Values

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Process | Code Block | Resource | Proposition | Truth Value |
| Process 0 | Code Block 0 | Resource 0 | req\_0\_0\_0 | T |
| Resource 1 | req\_0\_0\_1 | T |
| Code Block 1 | Resource 0 | req\_0\_1\_0 | T |
| Resource 1 | req\_0\_1\_1 | F |
| Process 1 | Code Block 0 | Resource 0 | req\_1\_0\_0 | F |
| Resource 1 | req\_1\_0\_1 | F |
| Code Block 1 | Resource 0 | req\_1\_1\_0 | F |
| Resource 1 | req\_1\_1\_1 | F |
| Process 2 | Code Block 0 | Resource 0 | req\_2\_0\_0 | F |
| Resource 1 | req\_2\_0\_1 | T |
| Code Block 1 | Resource 0 | req\_2\_1\_0 | F |
| Resource 1 | req\_2\_1\_1 | T |

# Constraints

There are 10 distinct lines of code that add constraints to the encoding. Each line adds a different type of constraint, and each line is surrounded by loop(s) to create all of the constraints of that type.

For simplicity, let

Furthermore, all ranges discrete integer values differing by one.

## Constraint 1: Each Process Code Block Must Run at Least Once.

For a given code block of process , at least of one of the schedule propositions should be true.

where

This is added for and .

## Constraint 2: Each Process Code Block Must Run Only Once

For a code block of process , and a processor on timeslot .

If is true, then all other schedule propositions for code block of process should be false.

where

This constraint is added for , , and .

## Constraint 3: No Two Code Blocks Can Run at Same Time on Same Processor

For code block of process , code block or process , timeslot and processor :

Therefore:

This constraint is added for , , , , with iterations for and .

## Constraint 4: No Two Code Blocks Can Use Same Resource at Same Time

For code block of process , code block or process , on timeslot , processor and for resource :

where

where

It cannot be the case where both are true:

Therefore:

This constraint is added for: , , , , with iterations for  
 , , .

## Constraint 5: Code Blocks of a Process Must Run in Order

### Part 1:

For a given code block of process . If the code block is scheduled on timeslot , then code blocks to of process must NOT be scheduled on timeslots to . Consider code block of process where is always less than .

where

where

In NNF:

This constraint is added for , , , with iterations for .

### Part 2:

For code block of process scheduled on timeslot , then every code block after it must run in timeslots

where

where

In NNF:

## Constraint 6: There Can Be No Timeslot Where All Processes Are Idle

### Part 1:

A timeslot is considered empty when there is at least one code block scheduled on timeslot and timeslot . This should not happen in the schedule.

where

where

where

Therefore:

This constraint added for all in .

### Part 2:

This just implements the case where the first time slot cannot be empty.

where

Therefore:

### Code References

All lines referenced are in *profldea\_implemented\_feedback.py.*

|  |  |
| --- | --- |
| **Constraint** | **Lines** |
| Constraint 1 | 185-193 |
| Constraint 2 | 196-218 |
| Constraint 3 | 220-240 |
| Constraint 4 | 244-276 |
| Constraint 5: Part 1 | 283-319 |
| Constraint 5: Part 2 | 327-347 |
| Constraint 6: Part 1 | 351-390 |
| Constraint 6: Part 2 | 393-401 |

# Jape Proofs

In the case where proc0\_cb0 and proc2\_cb0 use the same resource r0, they can’t be scheduled on the same time step

So:

Input: req\_2\_0\_0, req\_0\_0\_0,

Constraint: ((req\_2\_0\_0 & req\_0\_0\_0) -> (~(schedule\_0\_0\_2\_0 & schedule\_0\_1\_0\_0) & ~(schedule\_0\_1\_2\_0 & schedule\_0\_0\_0\_0)))

result(based on output from program): schedule\_0\_0\_2\_0 & ~schedule\_0\_1\_0\_0

Let:

A = req\_2\_0\_0

B = req\_0\_0\_0

C = schedule\_0\_0\_2\_0

D = schedule\_0\_1\_0\_0

E = schedule\_0\_1\_2\_0

F = schedule\_0\_0\_0\_0

so it would be (I reduced to nnf)

input: A, B,

constraint: ((A & B) -> ((~C | ~D) & (~E | ~F)))

result(based on output from program): C & ~D

In jape format:

A, B, (A ∧ B ) → ((¬C ∨ ¬D)∧(¬E ∨ ¬F)) ⊢ C∧¬D

# 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.*

## Model History

### Initial Version

The original version of the model aimed to capture resource sharing between only two processes. It featured propositions to represent when a process was using a resource and waiting for a resource to be used. It also had constraints that imposed the rules of resource sharing between processes ie constraints for Mutual Exclusion, Bounded Wait and Progress.

However, the model was found to be insufficiently complex based on the proposal feedback that was received. The model did not capture a sufficiently complex real system where there are several processes using resources that would have to be scheduled on the processors.

### Intermediate Version

This version of the model extended the previous case, where processes now have code blocks that can make use of different resources. These code blocks are the critical sections of the process. The aim of this version was to determine a schedule for the process code blocks on the different processors.

This version featured new types of propositions. Resource propositions covered every possible resource use case for the different code blocks of the two processes. Schedule propositions covered every possible scheduling need (each schedule proposition covered the case where one code block needed to be scheduled before another).

The constraints of this version are imposed on the different schedule proposition to eliminate the case where two contradicting schedule propositions are true i.e., the case where code block k is scheduled before code block l but also code block l is scheduled before code block k. Furthermore, there were also constraints to assign a truth value to the schedule propositions based on the truth values set to the resource propositions. That is, constraints for the case that code block k and code block l would need to be scheduled if they made use of the same resource r2.

This version was inadequate because it only modelled a limited scenario. The model was for only two processes, each with two code blocks and two resources. Furthermore, the model only assumed one processor for the system which meant the need for scheduling based on resource use was not required, therefore invalidating the need for the resource constraints.

### Final Version

This version is an extension of the previous version, extending the model to be functional for any specified number of resources, processes, process code blocks and processors. A distinct addition to this version was the use of time slots to capture each part of the schedule.

This version also heavily used Python for-loops when creating variables and constraints to support the modification of the number of processes, processors, resources and code blocks.

## Model Output and Optimizations

As previously mentioned, the output of the model consisted of the truth values of the schedule propositions, which are printed to the console. The propositions can then be used to construct the schedule for the code blocks on the processors. For convenience, the program uses the propositions to construct the schedule and print the output onto the console. In addition, the program also prints the resource usage for each of the process code blocks.

In process/job scheduling there is no known method for finding the optimum schedule (the schedule which takes the minimum number of timesteps) apart from trial and error. This method of trial and error was applied to estimate an optimum solution.

As a starting point to the estimating the optimum solution, a minimum number of time slots is calculated. In perfect circumstances, all the code blocks are distributed evenly across each processor. Since the number of time slots is equal to the number of code blocks, the absolute optimum solution will have a maximum time slot equal to that case.

The calculated is assigned to *optimumMaxSlot* and is the starting point for the method. The program runs a number of trials, where in each trial the highest number of timeslots (*maxSlot*) is calculated and compared to *optimumMaxSlot.* If *maxSlot* is less than or equal to *optimumMaxSlot*, then the system has found an estimated optimum solution. This is because the current solution has less or equal used time slots than the optimum solution. If no solution is found after the trials, then ­*optimumMaxSlot* is incremented by one and the trials are done again. This is because, they may not be a solution that uses less or equal timeslots than the optimum solution.

It is important to note that the found optimum solution will not be closest to the real optimum solution. This is because the loop will break once it has found a solution with the correct number of slots or after the number of trials has expired. The number of trials does not always capture the number of possible solutions. (Docker was unavailable to count the solutions).

The number of trials executed is adaptive based on the complexity of the problem. The addition of more processors, processes, code blocks, etc. will add more constraints to the model therefore making it harder to solve and lowering the number of solutions. Furthermore, it will also take longer to be solved by the SAT solvers which results in a longer amount of time spent on each trial. Therefore, using the same number of trials for each problem will be insufficient.

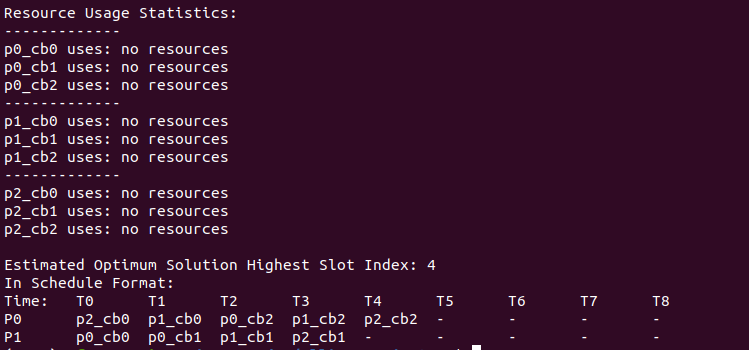
The complexity of the problem is measured based on the amount of time it takes the SAT solver to determine if there is a solution. More complex problems will take the SAT solver longer. The measured time is input to a basic function that converts the amount of solving time to a number of trials.

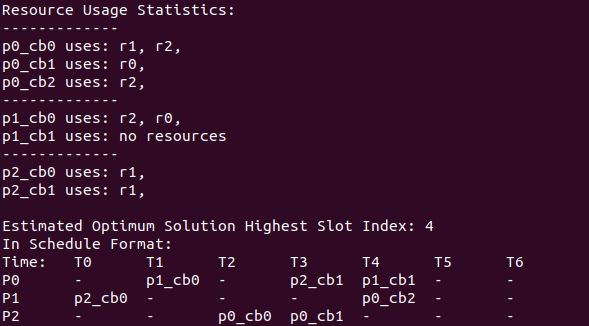
The exponential function was found to be an adequate function for determining trials. The function balanced the need for rigorous testing with the need for an average running time of the program.

The estimated number of trials as well as the highest timeslot index in the estimated optimum solution are both printed to the console.

## Model Testing and Verification

The implementation of the model in Python code was tested manually to ensure that it behaved correctly. Several techniques made this verification easier. First, the program was changed to print out its results in a more comprehensive way. Secondly, a debugger was used to unsure the the control flow of the code was as expected. Finally, the program was run with several different configurations and its outputs were observed.

The idea behind the first test setup was to ensure that the program could find models that scheduled each of the code blocks as soon as possible without resource sharing.

The second test setup was designed to test that code blocks that shared resources were not scheduled during the same time slot and the code blocks of each process were scheduled in the correct order.  


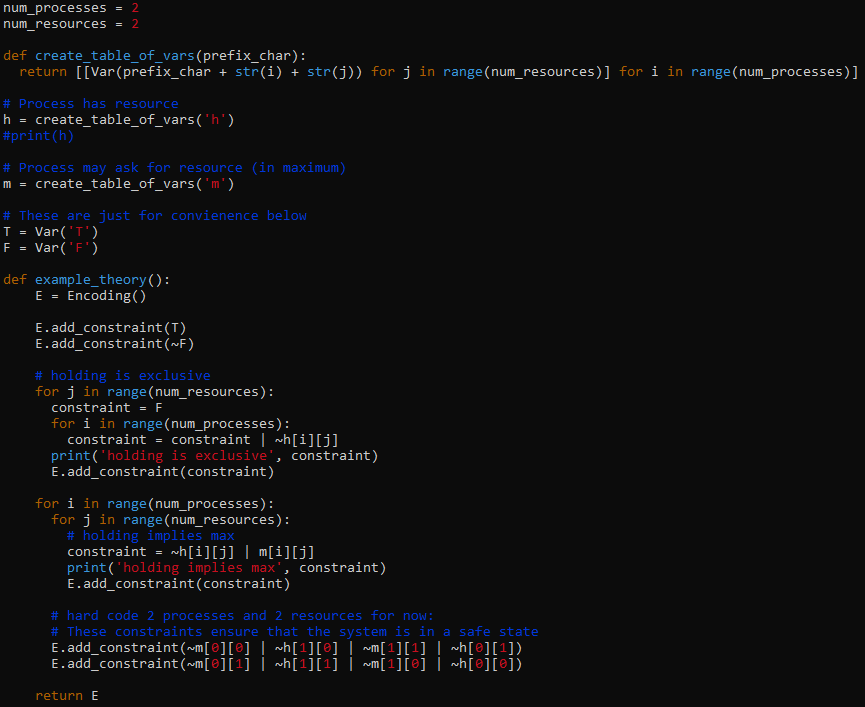
The third test setup aimed to ensure that the program did not break when the scale of the problem was increased.

## 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.)



# Useful Notation

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