My Notes

In my attempt to model check the Xv6 scheduler, I faced several challenges. The challenges can be put into three distinct categories (ordered from most enjoyable to least enjoyable): challenges arising from attempting to mathematically model the scheduler's properties (learning to translate a piece of code to mathematics, and learning the TLA+ language), challenges that came up as decision choices (choosing the level of detail and atomicity), and challenges that the tools (TLA+ toolbox and TLC) presented.

This document was written up after the project was completed, and heavily borrows material from the original notes I kept as I worked on the project.

Mathematical Modeling Challenges

The most significant downside of trying to use TLA+ as a beginner is how unintuitive the language is. To be fair, the problem isn't that TLA+ is unintuitive; rather, the problem is that I, as a programmer, expected TLA+ to resemble some sort of pseudo-code. TLA+ is essentially a syntax for discrete mathematics and the temporal logic of actions (TLA).

Temporal Logic of Actions

The configuration of all variables of the system at a given moment represent the state of the system (in fact, the state is even broader than that - the state is the configuration of the entire universe - but we'll use this simplification as it makes explaining easy). Depending on the system's properties, only certain transitions are possible from a given state. To illustrate this, take the example of a system that consists of one variable, `x`, that counts upwards in steps of `1` starting from `0` and loops around back to `0` when `x > 3`.

Let us denote the current state of the system as `[x = k]`, when `x` has the value `k`. There are many possible states that the system could have, though some may seem nonsensical: `[x = "blah"]`, `[x = 3]`, `[x = sqrt(-1)]` are all possible states of our system. However, only a few of these states are *reachable* (in the states listed, `[x = 3]` is the only reachable state). There could be an infinite number of reachable states (this would be the case if `x` didn't loop back to `0`, and kept counting upwards instead), and in specifying our system, we don't explicitly specify which states are reachable, but implicitly do so by specifying the possible transitions the system can make.

The transitions are written as relations between the next state and current state. The variable in the next state is denoted with an apostrophe: `x'` is `x` in the next state. So the relation `x' = (x + 1) % 4` represents the possible transition of our system. That transition is not an assignment statement. It is a boolean expression, true when `x' = 3` and `x = 2`, for example, and false when `x' = 9` and `x = "hello"`.

A transition between two states is said to be enabled when the transition is `true` for the pair of states.

Discrete Math

TLA+ is the temporal logic of actions and discrete math. Therefore, operations such as picking the maximum value from a list need to be written in the form of math. For example, the process of choosing the maximum

from a set S could be written as:

```
Choose x from S such that for all y in S, x >= y
```

The Transitions of the System

The following transitions are specified: * Preemption: A RUNNING process is made RUNNABLE, and the scheduler is invoked (pageTableLock acquired). * Sleep: A RUNNING process is made NONRUNNABLE, and the scheduler is invoked (pageTableLock acquired) * Schedule: A RUNNABLE process is selected, and is run on this CPU. If one doesn't exist, the pageTableLock is released. * MagicRunnable: A NONRUNNABLE process is magically made RUNNABLE. This is what would happen if a new process was spawned or a sleeping process was woken up. * MagicSchedule: A CPU can come alive magically (via timer interrupts in real life) to run the scheduler.

The Invariants Tested

These are the invariants tested (must be true at every state traversed).
* Deadlock: The system doesn't come to a halt (a transition is always possible). * TLBValid: The transition look-aside buffer contains the right address translations. * SchedCPUIsFree: The CPU running the scheduler doesn't have a process associated with it. * SchedulerHasLock: When the scheduler is active, the pageTableLock must be held by it. * NotSameProc: No two CPUs must be running the same process. * TypeInfo: Ensures integrity of data structures.

Level of Abstraction and Atomicity Challenges

A few decisions were made in order to limit the state space of the search. * The process table size was limited to five entries. * The number of CPUs is assumed to be 2 for this proof. * The number of states a process table entry could have was limited to RUNNING, RUNNABLE and NONRUNNABLE. The original Xv6 scheduler has the states UNUSED, SLEEPING, EMBRYO and ZOMBIE, in addition to RUNNING and RUNNABLE. Although in bunching these four together as NONRUNNABLE, a hole has been introduced into my proof, I think (and hope) it's a trivial hole. * The procedure of acquiring the pageTableLock and then finding a process to run is modeled as two separate steps. Treating these procedures as a single, atomic step would trivialize the proof of correctness. * On the other hand, the procedure of finding a process to run (once the pageTableLock has been acquired) is done in an atomic step. The reasons are that firstly, the system cannot progress until the pageTableLock has been released (a RUNNING process cannot stop running, it requires the pageTableLock, and likewise, a NONRUNNABLE process can't come alive without an external force but that force also needs the pageTableLock), and secondly, the state space searched would increase (each step in the search would be a state). * I have tried to generalize as much as I can, without trivializing the proof.

The Challenges of using TLA+ and TLC

This portion serves to list out the challenges faced in attempting to model the scheduler with TLA+ and then model check the resulting specification with TLC.

Note 1 (on record definitions):

There is a differences between "|->" and "->". The difference is basically that the latter is used in function definitions (type definitions) when one wants to specify that a function 'f' maps domain 'D' to range 'R'. Essentially, "->" results in a Set of records, whereas "|->" is used to map single values to the corresponding image.

Note 2 (on temporally reassigning records):

Unfortunately, it looks like the best way to do so is the `i \in Domain` syntax. Looks pretty ugly, but directly saying `[var |-> value]` ends up mapping the variable to some value. I was hoping to use the `@` and `!` syntax of reassigning records.

Note 3 (deadlock):

If you encounter a deadlock, examine the state that TLC complains about. Evaluate the "Next" step condition, whatever it is, at that state (it should fail - that's why we're getting a deadlock). I guess peer into why it's failing. I pray that the `Next` step you're examining isn't massive. In the Missionaries-Cannibals file, I screwed up by typing an `>` instead of `>=` (under the IsSideSafe(side)). Luckily it was easy enough to find.

Note 4 (Indexing):

TLA+ indexes stuff from 1 .. N rather than 0 .. N - 1. Quite sad making that shift as a programmer. The error shown is ridiculous:

```
 The exception was a java.lang.RuntimeException
: Attempted to apply tuple
<<1, 1>>
```

Note 5 (Variables unspecified by action):

You need to say `UNCHANGED` or otherwise specify what happens to every variable on performing an action. TLC is complaining for your own benefit. Otherwise, the variables could take on some senseless values during the transition from one state to another.

Note 6 (TLC takes forever to find initial states)

TLC tries to generate all the states that match the initial condition specified. In TLC, if the initial condition is written as `TypeInformation /\ Init`, this could kill you: TLC first generates all the states fitting `TypeInformation` (and there could be many, many, many such states), and then filters these states through `Init`. The better way to generate states is to say `Init /\ TypeInformation`. This will work as you desire.

Future Ideas

Things that could be done in the future, to prove the correctness of the Xv6 scheduler in a more fine-grain manner.

Units of Work:

Keep track of the total number of units of work that the CPUs have performed. Keep track of the units of work performed on each process. Then, complain if there is a process that has not received its fair share of units. Is there a way to configure the number of CPUS / maybe their shifts so that we can starve a process?

It may also be possible set a periodicity in UoW per process. That way, premption happens exactly then, otherwise, we can specify a timer that preempts periodically. We preempt whichever comes faster