Model Checking Safety Specifications in OS/161

Ankush Das, Jenny Lin, Sannan Tariq and Maria Khutoretsky

December 14, 2017

Abstract

OS/161 is a teaching operating system, that is, a simplified system used for teaching undergraduate operating systems classes. It provides low-level trap and interrupt code, device drivers, in-kernel threads, a baseline scheduler, and an extremely minimal virtual memory system and a virtual file system layer. Being a widely used instructional operating system, it is important to ensure that it is safe to execute in its current form. In this project, we intend to use CPAChecker, a standard tool for verification of C programs to prove the safety of OS/161 against null pointer dereferences and incorrect locking protocols. The two main modules we intend to verify are the device drivers and the virtual file system. A major obstacle here is that CPAChecker is unable to verify the entire OS/161 system due to state space explosion. Hence, we first create a simplified model of the source code relevant to our specification. This is done by removing function definitions and variables that don't impact the specification and simplifying the data structures. Next, we verify this abstract model using CPAChecker configured with predicate abstraction. CPAChecker either proves that the model satisfies the specification, or will return a counterexample identifying a bug in the program. We use this to prove assertions correct and identify the conditions under which runtime warnings are printed in OS/161. Creating this abstract model however is tedious and mostly manual at this point. It is also the source of error as abstract models may not correctly represent the specification. As our 125% goal, we design techniques to assist the programmer in creating abstract models. We parse the program to create a control flow graph of the program. We are currently working on annotating them with the variables modified by each function. Analyzing this graph will help a programmer eliminate irrelevant variables and function definitions. We report the results obtained while proving the specifications and our progress on designing assisting techniques.

1 Introduction

1.1 Fundamentals of Model Checking

We briefly introduce model checking and its limitations. Model checking refers to the following technique: Given a model of a system, exhaustively and automatically check whether the model meets a given specification. The specifications typically contain safety requirements such as absence of deadlocks, null pointer dereferences, buffer overflows and other critical states that can cause the system to crash. To solve such a problem algorithmically, the model of the system and specification are expressed in a precise mathematical language. Then, the task of the model satisfying the specification reduces to checking whether a given structure satisfies a given formula in the mathematical language. The mathematical languages used are generally mathematical logics, e.g. propositional logic, first-order logic, second-order logic, etc.

The fundamental problem with model checking is the exponential blow up of the state space, known as state explosion problem. Several approaches have been proposed to solve this problem. Binary Decision Diagrams (BDDs) are commonly used to abstract several states as one logical formula [5]. Bounded model checking algorithms unroll the state machine for a fixed number of steps k. Partial order reduction can be used (on explicitly represented graphs) to reduce the number of independent interleavings of concurrent processes that need to be considered. Counterexample guided abstraction refinement (CEGAR) [6] begins checking with a coarse (imprecise) abstraction and iteratively refines it. When a violation (counterexample) is found, the tool analyzes it for feasibility. If the violation is feasible, it is reported to the user; if it is not, the proof of infeasibility is used to refine the abstraction and checking begins again.

1.2 Basics of OS/161

OS/161 [8] is a simplified teaching operation system meant to be BSD-like in structure. It was developed at Harvard University to teach operating systems to undergraduates. Its base system provides low-level trap and interrupt code, device drivers, in-kernel threads, a baseline scheduler, and an extremely minimal virtual memory system. It also includes a simple skeleton file system and an emulator pass-through file system,