Offloading Stack Unwinding in a Rust OS

Sebastian (Alex) Schott, Advisor: Lin Zhong

Fall 2024

1 Introduction

It is often taken for granted that an operating system runs on the computer it manages. However, this need not be the case. For embedded systems which need to run a lightweight OS, there are many functions which are infeasible to implement as they would require too many resources. By offloading certain computations to a separate and more powerful computer, we open the door to many interesting optimizations. Hopter [2] is a lightweight port of the experimental Rust operating system Theseus [1] to microcontroller-based embedded systems developed by Zhiyao Ma at Yale. Because of Theseus's ability to recover from transient faults and update itself without rebooting it is ideal for running mission critical embedded systems. However, certain systems required far too much memory (SRAM) and storage (FLASH) to be run on a microcontroller and are prime candidates to be offloaded.

2 Project

This project will work on offloading functions for the Hopter operating system by moving error handing to a connected PC. Unlike most OS's, Theseus handles kernel panics through stack unwinding, and requires extra sections in application object files called the $.eh\_frame$ and $.gcc\_except\_table$. $.eh\_frame$ is the unwinding table used to recover callee-saved registers, and $.gcc\_except\_table$ contains the location of function's landing pads which is used to invoke destructors. Despite these sections only being used during a panic, they account for more than 8% of the entire binary size. With the

addition of the unwinding code in the *.text* section, this can account for up to 35% of the binary size. The existing Hopter code base supports local stack unwinding on a microcontroller, and the goal of this project will be to shift as much code and data as possible to the connected PC. By offloading computation we introduce a delay as we send data back and forth between the microcontroller and the PC. However, this is acceptable because panics should be rare and stack unwinding need not occur often. In the case a panic does occur, the alternative would be a complete system failure. Therefore, even if stack unwinding comes with a significant delay, it will be an improvement on the existing OS. This process can be split into the local component (which runs on the micro controller) and the server component (which runs on the connected PC) which will exchange data using a message passing system which implements error checking to ensure data is not lost during transmission.

3 Deliverables

All of the code will be made publicly available on GitHub, and will include examples and documentation that showcase the usefulness and performance of offloading stack unwinding. The two main sections of this work will be the local and server components. The local component will be a modification of the Hopter source code which replaces the local stack unwinding with a request to the server. The server will be responsible for receiving the and responding with the correct data or code segment. The server will also need to perform runtime linking, as the local component will not reserve memory for unwinding code until a panic occurs. In addition, I will submit a final report which showcases the full functionality of this project.

4 Mathematical Content

The mathematical foundation of this project is centered around several key areas: error detection and correction in communication systems, optimization techniques for reducing binary size, and the analysis of latency in offloaded computational processes. Each of these components plays a crucial role in ensuring the success and efficiency of offloading stack unwinding from an embedded system to a connected PC. Communication between local and

server components will be over a serial UART connection which can often be unreliable. To ensure data is not lost or corrupted during communication, this project will investigate and implement error detection and correction algorithms. In addition, a significant portion of this project will be spent on optimizations, to both reduce the latency of executing offloaded functions, and minimize memory usage on the microcontroller. This will be require an analysis of the trade offs between time and space complexity, as well as simple markov chains or state machines to ensure the local and server component are always in sync.

4.1 Timetable

- Sep 20 : Proposal In-class Presentation
- Oct 20 : Functional Code for Local and Server Components
- Oct 25 : Midterm Progress Report Presentation
- Nov 20 : Finalized Code
- Dec 2 : Final Report Draft
- Dec 3 : Poster Submission
- Dec 12 : Final Report Submission

References

[1] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. *In Proc. USENIX OSDI*, 2020.

[2] Zhiyao Ma, Samantha Detor, and Lin Zhong. Offloading operating system functions to the cloud. In *Proceedings of the 25th International Workshop on Mobile Computing Systems and Applications*, pages 40–46, 2024.