Jackson Paull, Jenna May

JHP2539, <TODO>

10 April 2025

ECE 381L - RTOS

Graduate Project Report

1. **OBJECTIVES**

Our project was focused on the expansion of the basic OS implemented in class to include several additional features (with a focus on security). We implemented a separation of the MSP and the PSP, a more modular implementation of background tasks which run in thread mode rather than handler mode, a more robust filesystem based on the linux FS, and a integration of the MPU to protect OS code from loaded processes.

*Note: We have no hardware design or measured data for this project. All of the extensions we made were software-based extensions for a general RTOS rather than a task specific RTOS where measurements like overall CPU utilization would be an important optimization factor.*

1. **SOFTWARE DESIGN** - *Note: See GitHub, tag ‘final-release’ for source code files.*

In this section we provide software documentation for the features we developed and added to our OS.

*MSP And PSP Separation*

*Expanded Filesystem*

To complement our generalized OS, we included an implementation of an iNode based filesystem. The key benefit of this file system is that it does not need a file table, or any other resource that scales in size proportional to the disk size, to be loaded into RAM at all times.

Our implementation also includes nested directories which can be explored from the interpreter.

<TODO Add figure of disk format>

<TODO Add figure for Singly, doubly, and triply indirect pointers>

<TODO Add call graph>

The benefits of this addition were not directly related to security features, as are our other expansions, yet the expanded filesystem supports our goal of the creation of a more general RTOS, which is closer to something that would be commercially available than the simple RTOS developed in class.

*Expanded Background Threads*

Our implementation of background threads run in *thread mode* as opposed to the base OS running in *handler mode*. We accomplish this via a reserved priority level in the scheduler for background threads. These threads exist in a priority queue in the scheduler, rather than a circular linked list, therefore they are automatically unscheduled once they run. However, once these threads return, they are not expected to die forever – so they maintain control of their stack after death, in anticipation of the next scheduling event (i.e. timing or switch press).

<TODO Add in figure for background thread TCB>

<TODO Add in figure for background thread lifecycle>

The benefit of this implementation is two-fold. Firstly, our background threads have no restrictions on them (such as the inability to wait). This particular application would be most useful for background tasks triggered on switch input, which might then communicate with another thread. The second key benefit of this approach, and our primary motivation for implementing this, is that background threads will not exist in a privileged context. Otherwise, any user-added background threads would circumvent the protections implemented by the MPU.

*MPU Integration*

1. **ANALYSIS AND DISCUSSION**

Our current implementations could be expanded in a number of ways. We do not currently protect process space from other processes, under the assumption that the OS is a single user system and therefore any code or heap space shouldn’t be protected from other processes. However, our OS could be expanded to a multi-user environment where this protection becomes necessary.

Furthermore, our current implementation of the linux filesys is limited to an 8MB partition with 8MB filesizes. The extension to an arbitrary disk size, while non-trivial, would only require the modification of the bitmap that the filesystem maintains to allow for multiple sectors as opposed to just one and the introduction of a dynamic sector for the root directory dependent upon the disk size rather than being predefined. The extension to an arbitrary filesize is not as practical, but the introduction of triply, quadruply, and quintopoly indirect pointers would allow for file sizes up to ~15TB filesizes, which is so large for a single file that it is effectively an arbitrary file size.

Further extensions include an expansion of the supported interpreter commands, a rewrite of the ESP8266 driver to support multiple wireless connections simultaneously, and function level optimization to reduce overall CPU utilization. Lastly, there are a number of global pools which could easily be converted to utilize the heap, and the size of the heap could be tuned to use as much memory as possible.