Jackson Paull, Jenna May

JHP2539, JTM4565

29 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 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 file system based on the Linux FS, and an 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[[1]](#footnote-1), 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*

For our focus on security, we first separated user and supervisor code using the MSP and PSP. When starting execution of our first thread, we switch to unprivileged thread mode and use the PSP for all user code. User threads will use the PSP and will utilize SVC calls to access OS code, which will be run in privileged handler mode. Exception handlers, such as the PendSV handler, will utilize the MSP and will run in handler mode. These handlers are edited from our original OS to return to the appropriate PSP value upon exit. Discussed later, our work running background threads in thread mode will ensure complete separation between user code, which uses the PSP, and OS code, which uses the MSP.

In addition to decoupling thread and OS stacks, this separation restricts user access to certain areas of memory, including the system timer, NVIC, and system control block. User threads cannot write to protected system registers and therefore malicious threads are unable to tamper with system configurations.

*MPU Integration*

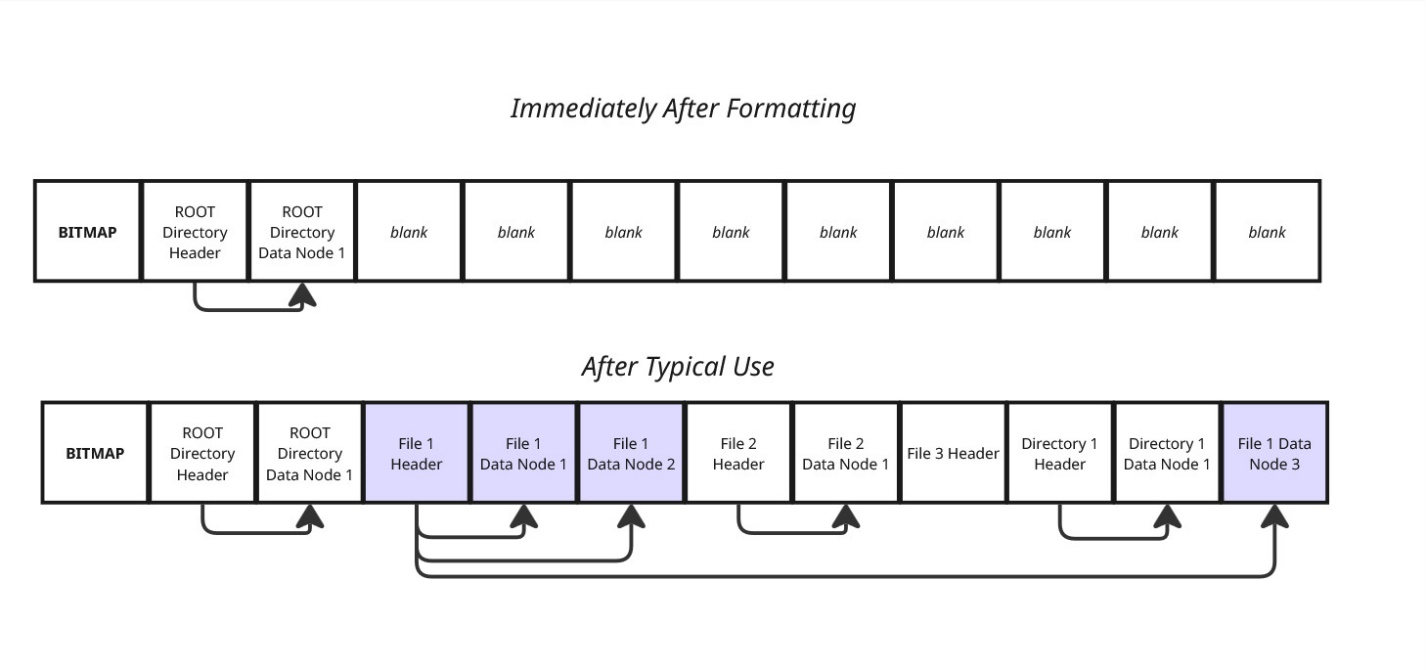
MSP and PSP separation alone leaves the OS vulnerable to code injection by a malicious thread. User threads have full privileges to read and write from OS code locations, so malicious code could be inserted in OS functions. This injected code would be run in privileged mode and could result in unauthorized system accesses by the user. To prevent this possibility, we utilize the MPU to prevent access to OS code locations. The MPU is dynamically configured during initialization to prevent any access to OS code locations by unprivileged code. Because the MPU is only able to configure memory regions aligned with the configured power-of-two region size, this initialization requires multiple configured regions. An average of 4 regions are required to successfully protect all OS code, and this initialization will not take up all 7 MPU regions.

Access to a protected location by unprivileged code will generate a memory fault. Our memory fault handler will kill the offending thread and return to regular operation. This protection prevents code injection as well as simple unprivileged calls to OS functions.

*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 supports nested directories which can be explored from the interpreter or opened and manipulated via any foreground threads.

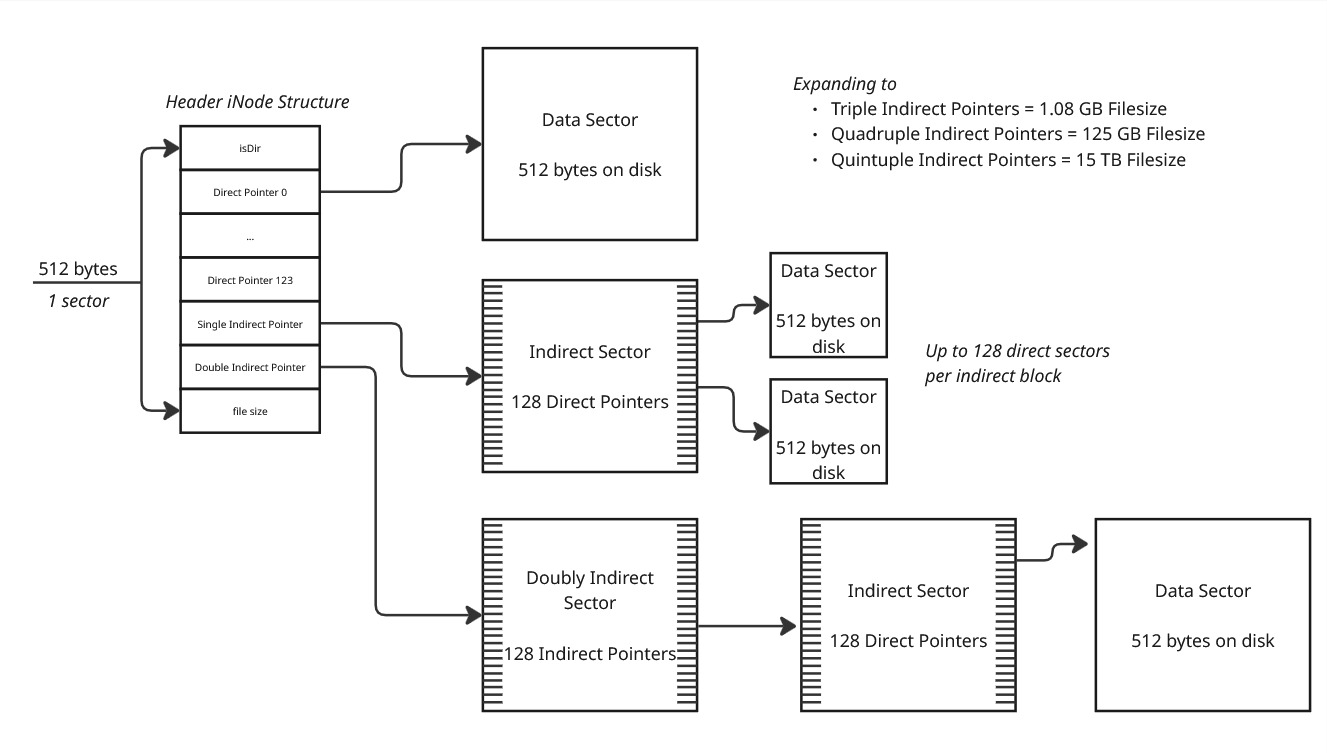


*Figure 1. Disk Format*

After formatting (see figure 1), the first sector is reserved for the **bitmap**. The bitmap uses one bit per sector to track whether a given sector is currently allocated or free. Our current implementation uses only one sector for this bitmap; hence we can track up to 4096 sectors on disk, which corresponds to an 8MB partition. Expanding this implementation such that the bitmap takes up multiple sectors allows for an unlimited[[2]](#footnote-2) disk partition size (allocating 0.024% of disk space for the bitmap itself).

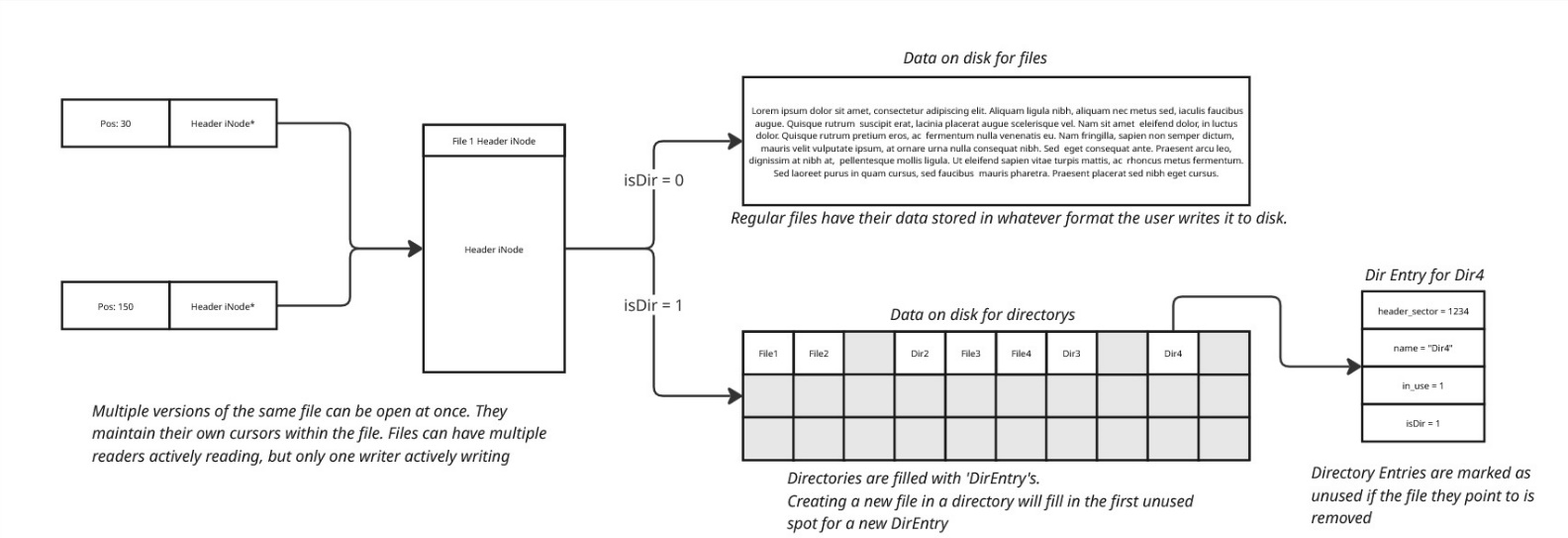
Immediately following the bitmap is the root directory. Directories are treated as files in that they share the same low-level structure as typical files. However, we do not recommend reading to or writing from directories in the same way that one might read to or write to a file, as this will likely corrupt the structure of the directory’s data nodes. See Figure 3 for more information.

After typical use, files will have multiple data sectors allocated, and there is no external fragmentation as there is no need for these data sectors to be continuous. Internal fragmentation is common and is upper bounded by 3 nearly empty sectors (1527 bytes). This worst-case scenario is representative of a case where a file has a single data sector allocated within the triply indirect sector. Meaning that only 1 of the 128 singly indirect sector pointers are used within the doubly indirect sector, only 1 of the 128 direct pointers within that singly indirect sector are used, and that only 1 byte within the data sector is being used.



*Fig 2. Structure of data stored on disk via header iNodes*

Note that the iNode structure is shared between files and directories (see figure 2). However, it is not recommended to use filesys functions intended for files to manipulate directories or vice versa because the data structure on the disk is distinct for the two. See Fig 3 below.



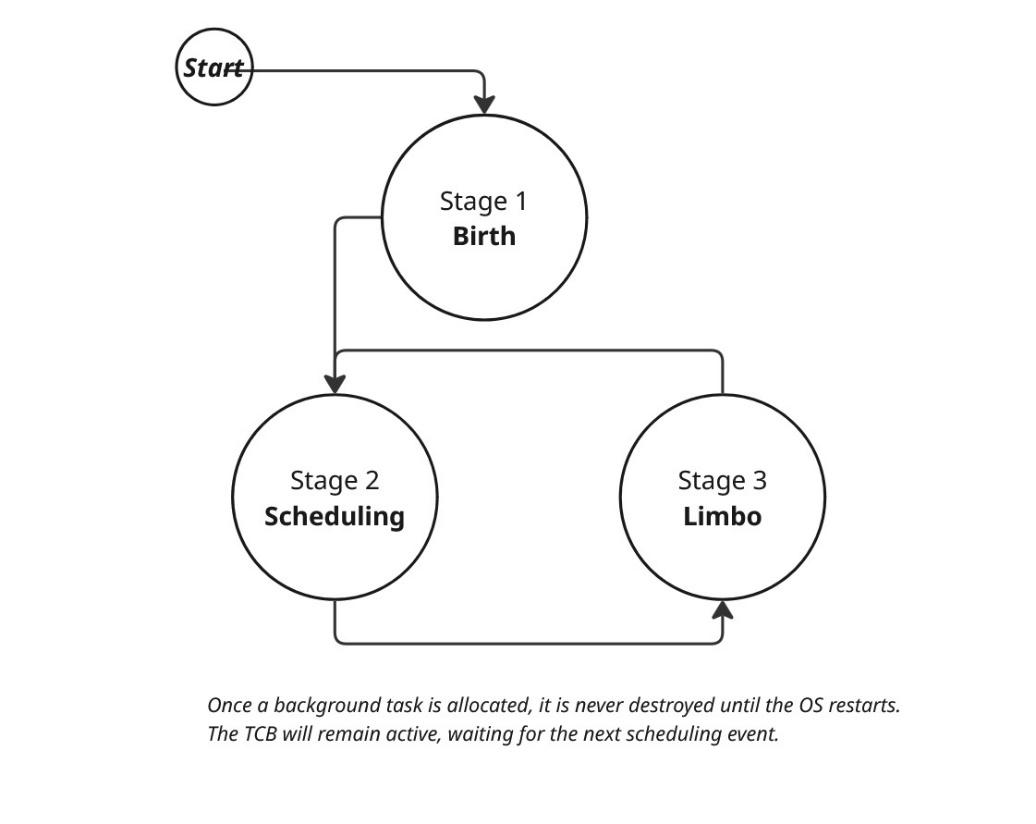
*Fig 3. Differences between directories and files on disk*

Each thread has a “currently open” directory, which is inherited from the parent first and defaults to the root directory if the parent has nothing open. These directories are closed on kill. These directories allow any two threads to operate from two distinct working directories simultaneously.

The number of direct pointers is maximized, choosing to maintain only a single indirect and single doubly indirect pointer. This is motivated by the fact that most files are small in size, and therefore we want to avoid allocating a separate sector just for pointers if we don’t absolutely need to.

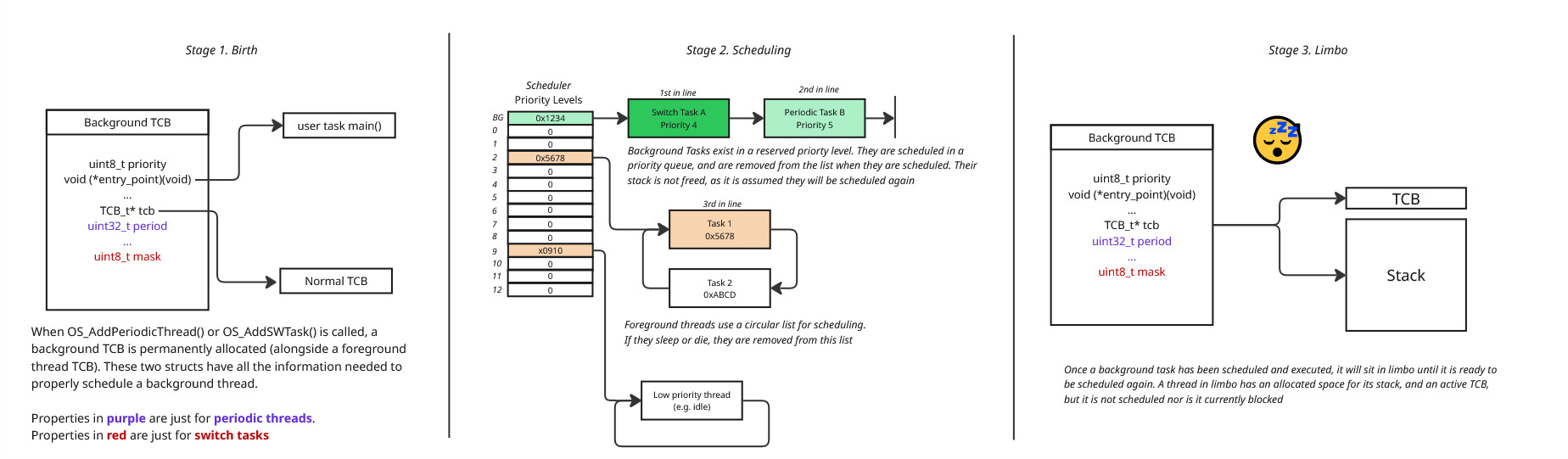
The benefits of the file system expansion are not directly related to security features, as was the goal for our other features, yet the expanded filesystem supports our goal of the creation of a more general RTOS, which is closer to something that would be commercially available than the simple RTOS 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).

*Figure 4. Background Thread Lifecycle*

A diagram of a computer

AI-generated content may be incorrect.

*Figure 5. Background Thread Lifecycle In-Depth*

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.

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 file sizes. 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 file size is not as practical, but the introduction of triply, quadruply, and quintopoly indirect pointers would allow for file sizes up to ~15TB file sizes, 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

Overall, we both learned a great deal about the practical implementation process for a more generalized Real-Time Operating System and, given more time, have a variety of paths from which our work could be further augmented.

1. https://github.com/JacksonPaull/RTOS [↑](#footnote-ref-1)
2. Unlimited disk space would require an infinite width integer to address the space. Using 32-bit unsigned integers yields a maximum of a 2TB hard-drive space. [↑](#footnote-ref-2)