Making a Real-Time Operating System for the Raspberry Pi 2B

Christian Cunningham

Physics Department, Arizona State University

Director: Aviral Shrivastava

Committee Member: Sarma Vrudhula

School of Computing and Augmented Intelligence, Arizona State University

April 9, 2022

Contents

I Introduction **6**

I.A Background 6

I.B What is a Real-Time Operating System 6

I.C About the System 7

I.C.1 ARM Execution Modes 8

I.D How to Evaluate a Real-Time Operating System 10

I.D.1 Overhead Test 10

I.D.2 Thread Tests 11

I.D.3 Mutex Tests 12

I.D.4 Semaphore Tests 12

I.D.5 Additional Tests 13

I.D.6 Priority Inversion 14

I.D.7 Deadlock 14

I.D.8 Stimulus Responsiveness Tests 15

I.D.9 Delayed Execution Tests 16

II Algorithmic Overviews of Jobbed **17**

II.A Memory Management - Static Objects 17

II.B Scheduler 17

II.B.1 Thread Switching 19

II.B.2 Thread Creation 20

II.B.3 Mutex Creating 20

II.B.4 Mutex Locking 21

II.B.5 Semaphore Waiting 21

II.B.6 Mutex Releasing 21

II.B.7 Semaphore Signaling 22

II.B.8 Priority Inversion Prevention 22

II.B.9 Deadlock Prevention - Eliminating Cyclic Wait 22

II.C Interrupts and Delayed Execution 23

III Evaluation against Linux **24**

III.A Quantitative Results 24
III.A.1 Thread Results 26
III.A.2 Mutex Results 27
III.A.3 Semaphore Results 27
III.A.4 Overall RTOS Test Results 28
III.A.5 GPIO Quantitative Test Results 29
III.A.6 Delayed Execution Test Results 30
III.B Priority Inversion Test 31

IV Qualitative Evaluation of Jobbed 32
IV.A Thread Prioritization Analysis 32
IV.B Deadlock Test 32
IV.C Semaphore Signaling Test 34
IV.D CPSR Preservation Test 34

V Future Work 36

VI Conclusion 37

VII Appendix 41
VII.A Useful Resources 41
VII.B Where to get the Latest Release 41
VII.C Using the Clock Cycle Counter to Benchmark *Jobbed* 42
VII.D Jobbed running in QEMU 43
VII.E Testing Setup w/ Arduino and Push Button 43

List of Figures

1 Priority Inversion Condition 14
2 The Deadlock Condition 15
3 Scheduling 19
4 Jobbed Execution Path 31
5 Deadlock Test Execution Path 33
6 Semaphore Test Execution Path 34

7 CPSR Test Timings 35
8 Jobbed in QEMU running the tests with the Cycle Counter . 43
9 Setup 43

List of Tables

1 ARMv7-A Execution Modes 9
2 RTOS Core Functionality Benchmarking Results 25
3 GPIO Benchmarking Results 29
4 Delayed Execution Benchmarking Results 30
5 RTOS Core Functionality Benchmarking Results with Cycle Counter 42

Abstract

Real-Time Operating Systems are used in a variety of applications ranging from autonomous vehicles, flight controllers, and energy management systems to pacemakers, satellite tracking systems, amateur robotics and much more. It turns out that while general-purpose computers can perform tasks quite quickly, the execution time for various processes varies noticeably between different executions. Execution time variation poses a big challenge for many computer-controlled systems that operate in the real-world such as robots, autonomous vehicles, drones, traffic signals, etc. The execution time variation matters in these systems since they must interact in the real world and perform actions at the proper times, and executing these tasks at other times can have varied effects ranging from a minor inconvenience to catastrophic failure. Many of these real-time systems are comprised of single board computers, such as a pacemaker. One single-board computer that is popular among hobbyists due to its form factor, cost, and performance is the Raspberry Pi, which uses an ARM-based processor. In order to provide a Real-Time Operating System for this single board computer this paper presents *Jobbed*, a single-core Real-Time Operating System which uses a fixed priority preemptive scheduler, targeted at the Raspberry Pi 2B. In this paper, we present the algorithmic structure behind this system and compare it to the *Raspbian* Operating System in a slew of performance and behavioral tests targeted towards proper Real-Time Operating Systems.

I Introduction

I.A Background

Across a variety of fields, the usage of single-board computers is an growing trend, most notably in industrial automation, aerospace, and among hobbyists[11]. In many amateur electronics circles, due to its small form factor and wide range of common peripherals, the Raspberry Pi is a highly used System-on-a-Chip (SoC) producing 7 million units in 2021[14].

Currently, the Default Operating System for the Raspberry Pi, *Raspbian*, only focuses on a user-driven experience which introduces additional overhead on process usage and nondeterministic process execution[8, 9]. Alternative Operating Systems advertised for the Raspberry Pi such as Lakka, Retro-Pie, and Ubuntu Core tout a similar goal–focusing on providing a strong user experience in a general-purpose operating system. Specifically, these general purpose operating systems focus on providing a system the maximizes its responsivity to the user, i.e. minimizes screen freezes or input delays. This goal means that the operating systems are typically non-deterministic and so execution times of various processes may vary considerably when executing the same task under the same conditions. Apart from the general purpose operating systems for the Rasbperry Pi, there are a few in-progress RTOS projects such as Trampoline OS, which mentions that a port for the Cortex-A7 (the Raspberry Pi 2B board) which is under development, or RODOS, which provides a middleware layer to be run atop of Linux.

I.B What is a Real-Time Operating System

Real-Time Operating Systems are characterized by deterministic, bounded, and low latency processing with a scheduling system that is able to meet deadlines[6, 9, 21]. Two common scheduler strategies are event driven scheduling, which implements a preemptive scheduler using fixed prioritization of tasks, or round-robin scheduling, where tasks are switched at regular time-intervals. In this paper, we will be presenting a system that conforms to the event-driven

system paradigm, with fixed priorities assigned to tasks and the scheduler has the ability to preempt lower priority tasks in favor of executing a new higher priority task. Preemption is important so that the system is always running the highest priority task at any given time, one of the foundational pillars of an RTOS. The RTOS may temporarily violate this condition to prevent *priority inversion* from occurring, by running a lower-priority task, in a deterministic manner, until it releases a critical resource required by a higher priority task. The bounded execution times, strict prioritization, and deterministic behavior makes RTOSs well suited for real-time system requirements.

I.C About the System

The target system of this paper is the Raspberry Pi 2B (v1.1) which contains an ARMv7-A processor, the ARM Cortex-A7 CPU, and 1 gigabyte of RAM. Additionally, the system provides numerous peripherals such as 4 USB ports, 1 ethernet port, 1 audio jack, 1 HDMI port, and 40 GPIO pins that can facilitate UART and two SPI[5]. The wide range of peripherals allows flexibility for accessing different types of resources from the system and enables the system to interact with the real-world.

ARM generally divides their processors into A, M, and R-series. The A series targets rich operating systems which contain MMUs, the M series targets low-power IoT applications, and the R series focuses on high performance and hard real-time applications. The Raspberry Pi falls into this A series, which is why this RTOS is primarily oriented towards hobbyists[4].

In order to boot Raspberry Pi 2B, a micro SD is required[18, 19]. The SD's first partition must be formatted to a FAT filesystem, containing the proprietary GPU boot blobs in the root directory: **bootcode.bin**, **start.elf**. Including a **config.txt** allows the system to be further configured at boot from a simple text file with values that are discussed in the Raspberry Pi documentation[1]. In order to load an operating system/ kernel to the Raspberry Pi, the SD card must also contain a **kernel7.img**[1] in the root directory. Then,

[1]For some other Raspberry Pi models, they load kernel8.img, kernel7l.img, or kernel.img

after inserting the SD-card and applying power, the Raspberry Pi firmware will:

1. Turn its GPU on
2. Load the first stage bootloader from the ROM
3. Load the second stage bootloader from **bootcode.bin**
4. Load the GPU firmware from **start.elf**
5. Configure any supported aspect of the system from the specifications in the **config.txt**
6. Load the kernel from **kernel7.img**
7. Begin executing at address 0x8000 in Hypervisor mode and park the 3 other cores

The **kernel7.img** and the **config.txt** are the two files that are changed when loading different operating systems. In our case, we only use the **config.txt** to ensure that the UART clock is set to 3MHz so that we can provide a fixed 19200 baud for the UART. *Jobbed* is then compiled to and loaded in through the **kernel7.img**.

In order to build the **kernel7.img** the ARM cross compiler I used was the GNU Compiler Collection with no Embedded-Application Binary Interface: `gcc-arm-none-eabi` to compile the C and ARM-Assembly into ARM instructions while working on x86-based machines. I also used QEMU, an emulator that provides a Raspberry Pi 2B emulator, to quickly test the source code while I was on my x86 machines. While working on QEMU, I noted that the emulator fails to benchmark timings correctly and so all of the benchmarking in sections IV-V was done on a physical Raspberry Pi 2B for both *Raspbian* and *Jobbed* to ensure that the timings were accurate and comparable.

I.C.1 ARM Execution Modes

The Raspberry Pi 2B contains a Cortex-A7 Processor. This Cortex-A7 processor has 8 different execution modes which control what registers the CPU uses, the privilege level, and status registers[3].

Mode	Use
User	Run Threads in an Unprivileged Mode
FIQ	Lower Latency Interrupt Handler
IRQ	Handle an Interrupt
SVC	Handle Systemcalls, i.e. Supervisor Calls
ABT/UND	System Error State Modes
HVC	High Privilege mode, used in Virtualization
SYS	Identical to USR mode but with system privileges

Table 1: ARMv7-A Execution Modes

These modes are useful so that when system interruptions occur while a thread is running, such as a signal from the GPIO pins or a delayed execution timer, the current execution state can be preserved and then restored at a later time. This is especially useful if the interrupt creates a higher priority thread because the new thread can be run in a clean state while the preserved execution state can be stored in memory so that the new thread execution does not tarnish the state of the previously running thread.

Additionally, the user mode is useful because threads are not able to directly access the coprocessor states, which can change things like the timer frequencies and thus alter the timing of the tasks. Since the user mode has no privileges, the scheduler can use the SYS mode to setup the registers, stack, and linking register of the user state and then be able to use its privilege to switch back to the intermediate SVC state utilized by the scheduler.

Another benefit of these modes is that the USR mode has restricted access to different memory regions when the MMU is initialized. In RTOSs, this memory privilege separation is not as emphasized unless the system is going to be put into an untrusted network. *Jobbed* initializes the MMU for one primary reason – to use the Cortex-A7 exclusive monitors for concurrency purposes. The exclusive monitors in the Cortex-A7 allows a process to read and mark and address as an exclusive access, so that when it goes to store data back into that address, it can tell if the original data was changed by another process. This is important for shared memory, such as mutexes and

semaphores, since they may be locked or waited on by different processes, and so the data could have been modified between the load and the store instructions. These exclusive access monitors are enabled when the MMU has been initialized, and so *Jobbed* initializes the MMU, setting the permissions on all the pages of memory that it statically loads (text, data, bss) and the stack spaces reserved for the system's threads, to take advantage of them. This simplistic approach to memory management makes it easy to tighten control on critical memory regions if the application running on the RTOS requires it since the initialization flag can be changed to `PRIVILEGED_ACCESS_ONLY` (`0b01`) in the `kernel/lib/mmu` source.

Another way that concurrency could have been established is by using the Cortex-A7 atomic swap `SWP`, which allows values to be swapped atomically. In the mutex case, this would allow the process to immediately tell if it was successful in locking the resource. The disadvantage to the ARM atomic swap instruction is that it locks out a bus in order to ensure the atomicity of the operation, which is non-ideal for a responsive system[10]. This is the primary reason that ARM had deprecated this method in favor of the exclusive monitoring in ARMv6[2].

I.D How to Evaluate a Real-Time Operating System

In their paper, *Which is the best RTOS for Drones? Evaluation of Real-Time Characteristics of Nuttx and ChibiOS*, Zhang and Timmerman propose 4 broad categories of tests for an RTOS encompassing overheads, threads, mutexes, and semaphores[22].

I.D.1 Overhead Test

The overhead tests, also mentioned in David's paper *Context switch overheads for Linux on ARM platforms*, seek to answer the question[7]:

- How long does it take to write down the current clock counter?

This question is important in assessing a source of overhead, also called tracing, that might be present in the testing. In other words, if one system takes 20μs

longer to write down the current clock time and then in a later test, that system tool 20μs longer than the other to perform a task, then it highly probable that both systems completed that task in the same amount of time.

I.D.2 Thread Tests

The thread tests focusing on scheduling behavior, i.e. the qualitative tests, seek to answer the following:

- If a lower priority thread is created, is it deactivated until the creating thread completes?
- If a thread is created with the same priority, is it placed at the head or tail of the queue?
- If the creating thread yields after creating a thread with the same priority, does the created thread run?
- If the created thread has higher priority, does it immediately run?

These qualitative tests are used to ensure that the RTOS is not violating the expectation that the highest priority task is running at any given time, and also to note the behavior of where new tasks of the same priority go. The benchmarking tests, i.e. the quantitative tests, for thread management timings seek answers to:

- How long does it take to create a low priority thread?
- How long does it take to create a high priority thread?
- How long does it take to switch between threads?

These quantitative tests are important to assess the boundedness of the system tasks so that Real-Time application designers can know how long the different processes will take. Methods such as Compositional Performance Analysis can then be used with these system task bounds to establish bounds on entire processes by summing up the bounds of the constituent components/ functions used in the task chain[6]. Understanding how long thread switching and creation takes also informs the system designer how much of a performance impact modularizing a process would cause. For instance, if it takes 1 millisecond

to create a new task, then it would be important to minimize the creation of tasks, and thus creating much larger single-task processes. Further, if it takes 1 millisecond to swap between tasks then it would have a similar effect–reduce the number of tasks to minimize the need to switch between tasks.

Apart from the thread behavior tests, they also propose a priority inversion test to ensure that the scheduler is robust enough to prevent violating the "highest priority is always running at a given time" principle when resources cause higher priority threads to get blocked. See Section I.D.6 for more information on *Priority Inversion*.

I.D.3 Mutex Tests

Zhang and Timmerman also propose a slew of mutex object benchmarking tests which involves answering:

- How long does it take to create a mutex?
- How long does it take to destroy a mutex?
- How long does it take to acquire a mutex that is not contended?
- How long does it take to acquire a mutex that is contended?
- How long does it take to release a mutex?

Similarly for the thread tests, these are important to ensure the boundedness and time scale for these processes so that applications can be thoroughly analyzed.

I.D.4 Semaphore Tests

The qualitative test they propose for the semaphore consists of ensuring that threads are only blocked from executing if semaphore counters are zero. This test consists of first trying to wait on a semaphore that is zero, blocking if the system is running properly. Then the thread signals the semaphore (for this part it has been reset back to zero) N times, in our case 3, and then waits on the semaphore $N+1$ times, 4 times in our case. The thread should only block

at the last signal since the counter should only have returned back to zero at that point.

Apart from these qualitative tests of the semaphore, the quantitative tests for the semaphore are:

- How long does it take to wait on a semaphore?
- How long does it take to signal a semaphore?
- How long does it take to signal a semaphore that has a non-zero value?

The last two are distinct since non-zero semaphore counters imply that no threads are waiting on the resource, since if they were they would be executing. Thus, a scheduler can take advantage of this assumption and sidestep needing to reschedule tasks after increasing the semaphore's value beyond zero.

Another two quantitative tests that are mentioned but are dismissed as unimportant for semaphores are the times to create or destroy a semaphore. This is dismissed since semaphores are typically only created statically upon startup rather than dynamically over the runtime[22]. In *Jobbed*, there is no dynamic memory allocation[2], thus semaphores are not dynamically created or destroyed.

I.D.5 Additional Tests

These slew of tests are important in assessing the bounds upon the different scheduling/ concurrency processes that are crucial for the operating of an RTOS. They are also important to assess that the behavior of these critical components follow the behavior expected of an RTOS rather than a general purpose operating system. The priority inversion test is an example of a crucial behavior that is not always upheld in a general purpose operating system but is crucial for ensuring the determinism of an RTOS. The Mars Pathfinder is a notable example where Priority Inversion caused the system to reset errantly[15].

[2]The closest process is giving a thread a reference to a statically allocated mutex that is managed by the system

I.D.6 Priority Inversion

Priority inversion is the condition when a scheduler causes a lower priority thread to finish before a higher priority thread. For example, let L be a low-priority task that has access to a resource R and H be a new high-priority task that wants access to R. In this case, H gets blocked by L until L releases R. If there is a medium-priority task M, M is executed before execution returns to L to release this signal. This provides a source of non-determinism for the execution of H and also means that M gets executed before H. To solve this issue, a scheduler must let L run until it releases R so that H may finish executing before M finishes executing[3].

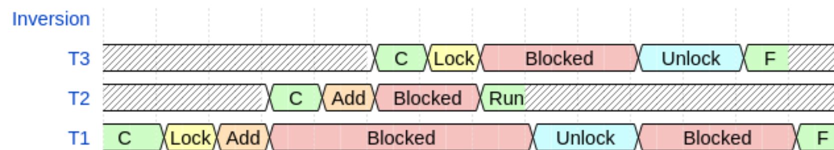


Figure 1: Priority Inversion Condition

Figure 1 showcases the priority inversion condition. In the figure, **C** refers to the thread being created and ran, the 'Add' blocks are when T_1 creates T_2 and T_2 creates T_3, and the number associated with the thread implies its priority (higher number means a higher priority in the figure). Thus, we see that T_2 completes before T_3, which is characteristic of priority inversion.

The reason why this is important is that there can be any number of threads with priority M, which means that there is a source of non-determinism for the wait time of T_3 since they would all need to complete in a naïve scheduling scheme.

I.D.7 Deadlock

Another test that is important in evaluating an RTOS is how does it deal with a deadlock condition. Deadlocks occur when tasks try to lock the same resources which then causes both of the threads to block, which then removes

[3]Or even starts executing if M is added after H starts

the condition for them to continue executing. For example, let T_1 initially access R_1 and T_2 initially access R_2. Then, a deadlock would occur if T_2 tries to access R_1 and gets blocked, allowing T_1 to run which then tries to access R_2, and is then blocked by T_2. In this situation, no thread can relinquish control of their resource to allow the other thread to complete or other threads to access these resources.

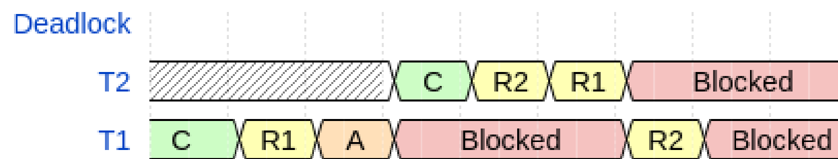


Figure 2: The Deadlock Condition

Figure 2 showcases one way a naïve scheduler may be susceptible to a deadlock. In the figure, **C** refers to the thread being created and started, the yellow background refers to the labeled resource trying to be locked by the process, and the orange **A** refers to thread one (A)dding T_2, which has higher priority than T_1. Since both threads have a resource required by the other, they both cannot continue to execute–i.e. they cannot release the resource to allow the other one to continue.

Thus, it is important to test *Jobbed* and ensure that sequences that are topologically similar to this do not cause threads to become indefinitely locked.

I.D.8 Stimulus Responsiveness Tests

Real-Time Operating Systems are used in environments where the system needs to respond to real-world conditions, or stimuli. These stimuli can come from a variety of sources, from temperature sensors, collision detectors, and so much more. In order to test the responsiveness of the systems, we will send GPIO pulses with periods ranging from 20μs to 2ms. The wide variability in pulse periods is to determine each system's behavior to slow and fast pulses, as well as show the threshold for predictable execution times of each system, if there is a threshold behavior expressed in either system. Using the difference between the real pulse length and the system's observed pulse length, we will be

able to bound the accuracy of the system for different pulse speeds. This pulse jitter is useful for RTOS Application Designers to determine if an operating system can accommodate their application. In other words, if their system is sensitive to jitter above 20μs and one of the systems has bounds that go above this level for the necessary speed, then it would not make sense to use that OS for the prospective system. On the other hand, if the OS can provide jitter bounds that fall below this threshold, then the system may be a viable candidate to use for the real-time system.

I.D.9 Delayed Execution Tests

Real-Time Systems also interact with a wide variety of processes, some of which may depend on executing tasks after a certain time period has passed. One example would be if the system sends a command for a substation to increase its output, it may add a 10s delayed call to then request the current production to ensure that the substation is increasing its production, and may also request additional state information like the temperature and if it is in the increase production state in the same call. In order to test the system's ability to delay executing a requested function, a user space application will request a specially tailored callback function to run n μs later, where n is set to a value between 10μs to 10ms and with each test at a constant n. The callback being run at the end of the 10μs to 10ms delay will be set to write down the current time, thus the time it starts executing, and then will compute how much time has elapsed between the request and execution of the callback. Using the difference between the computed delay and the requested delay across multiple tests will give bounds to the jitter of delaying a process's execution, similarly to the stimulus responsiveness tests. The RTOS Application Designer can then use the jitter bounds of each system as a test of viability of the operating system for the particular task.

II Algorithmic Overviews of Jobbed

II.A Memory Management - Static Objects

Jobbed employs a static memory layout rather than a randomly assigned memory layout, i.e. it does not implement the Kernel Address Space Layout Randomization that is present in many general purpose and Real-Time operating systems. This security feature is one of the features that may be implemented in a future release (see V Future Work). The lack of this randomness allows some Operating System procedures to be marginally quicker since the address space is statically assigned.

Dynamical allocation of memory is generally non-deterministic for shared memory multiprocessing, and so memory for tasks must be defined at compile-time[20]. This prevents both the nondeterministic implementations of memory allocation as well as also sidesteps any memory leakage that can occur in dynamic allocation schemes.

Due to the static allocation of objects and threads, the maximum number of threads and system-managed mutexes can be determined at compile time. Given that the Raspberry Pi 2B has 1GB of ram, the absolute number of threads and mutexes are constrained since the threads take up the determined stack size (4K in the default setting) and both take up $N_T \cdot 9$ bytes and $N_M \cdot 2$ bytes for the base structure plus $(N_T + N_M) \cdot 3$ bytes for the queue entries. Therefore, the total space taken up by the Threads and Mutexes are $N_T \cdot 4107 + N_M \cdot 5$ bytes. This calculation can then be used to determine if *Jobbed* can accommodate a prospective Real-Time application's memory footprint.

II.B Scheduler

Jobbed's scheduler consists of 5 key components:

1. A pointer to the currently running thread
2. Each priority's queue of threads that are ready to run
3. Each priority's queue of threads that are waiting on a mutex

4. Each priority's queue of threads that are waiting on a semaphore
5. A queue of all of the unused, i.e. free, threads

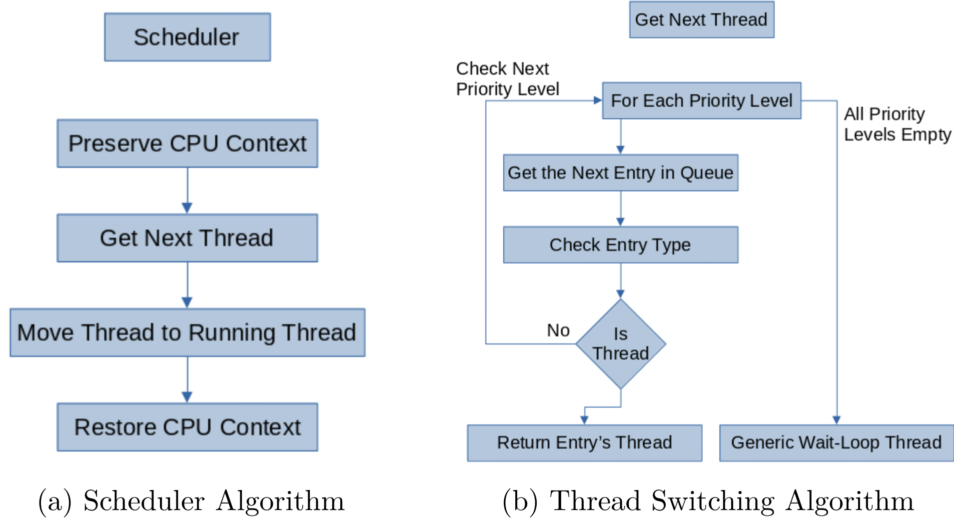
The access to the currently running thread provides faster context switching times since the context switcher can load the address and context information much faster.

The usage of a queue structure allows new threads to be easily obtained, added, and then removed when finished. Another approach to finding unused threads would be to have a fixed region of memory of threads and an array or bitfield to determine if that thread is being used. While this ensures an upper bound to finding a new thread, this approach would provide increasingly long execution times as more threads are removed from the list. On the other hand, the queue approach utilizes structure so that a new thread can be obtained from removing the entry at the beginning of the queue and can be returned to the end of the free queue when finished.

Additionally, the queue structure allows the next running task to be found in $\mathcal{O}(p)$-time at worst case, where p is the number of priorities assigned at compile time, since the first element can be queried in each queue to determine if the queue is empty.

It is also important to note that in *Jobbed*, a lower priority number in the scheduler implies a higher priority, since it will be indexed first from the queues. In the rest of this paper, I will be using a higher number to imply a higher priority but the implementation is the reverse case.

II.B.1 Thread Switching



(a) Scheduler Algorithm

(b) Thread Switching Algorithm

Figure 3: Scheduling

The thread switching algorithm, i.e. the scheduling algorithm, is fairly basic. It preserves the current context, gets the next thread, and then restores the new threads context. For the Raspberry Pi 2B, i.e. a ARMv7-A system, a threads context consists of five main components: the current value of the program counter, the current stack pointer, the current linking register (i.e. the return address of the current function), the current program status register, and the general-purpose registers r0-r12. Preserving these contexts involves adding the general-purpose register values to the stack and storing the special components in the Thread's structure managed by the scheduler.

Getting the next thread involves peeking at the top of each queue to see if the queue of that priority is full. At the worst case, all of the queues are empty and so a special thread, an infinite loop waiting task, will be added as the running thread while the scheduler waits for a new thread to run. Otherwise, it will have found an non-empty queue and return the reference to the queue's ready thread. Further, if the current thread is the highest priority of the ready threads, then this search will return the currently running thread and so it will continue executing.

Moving the next thread to the running thread involves overwriting the currently running thread pointer to the new thread's pointer. Since the old thread's context has been preserved and it still remains in the running list for its priority, it will be executed after the higher priority threads are finished as if it had not been preempted.

II.B.2 Thread Creation

Jobbed allows threads to be created with one argument and a given priority. The algorithm to add a thread to the priority p's ready list is:

1. Get a thread from the free-list
2. If the thread is NULL, exit with an error
3. Otherwise
   (a) Initialize the given thread's entry
   (b) Set the priority, program counter, and argument
   (c) Add it to the priority p's queue

II.B.3 Mutex Creating

The Mutex Locking algorithm that *Jobbed* implements does the following:

1. Gets the next available Mutex object, removing it from the free queue
2. If the next available Mutex object is null, return an error
3. Otherwise
   (a) Clear the lock value
   (b) Set the Mutex's resource reference to the given address
   (c) Add it to the in-use mutex queue
   (d) Return the mutex's handle to the caller

Since the kernel's Mutexes are created at compile time, it is important to setup the system with enough Mutexes for the workload so that the error condition does not occur.

II.B.4 Mutex Locking

The Mutex Locking algorithm *Jobbed* implements does the following:

1. Checks the Mutex's lock value[4]
2. If the lock value is non-zero, i.e. a thread is accessing it, block the calling thread and then call the scheduler
3. Otherwise, store the currently running process's pid to the lock

II.B.5 Semaphore Waiting

The Semaphore Waiting algorithm that *Jobbed* implements is similar to the Mutex lock and does the following:

1. Checks the Semaphore's value[5]
2. If the Semaphore's value is zero, block the calling thread and then call the scheduler
3. Otherwise, decrement the value and store it back into the semaphore

II.B.6 Mutex Releasing

The Mutex Releasing algorithm is:

1. Unlocks the mutex
2. Searches through the mutex wait queues for the first thread that is waiting on this mutex
3. If there is one, moves it to the ready queue and immediately calls the scheduler
4. If there is not one, it returns to the running task

[4]Using ARM's Exclusive Monitor to ensure that the value hasn't changed since accessing it

[5]Using ARM's Exclusive Monitor to ensure that the value hasn't changed since accessing it

II.B.7 Semaphore Signaling

The Semaphore Signaling algorithm is:

1. Increments the semaphore's value
2. If the semaphore was 0, it calls the scheduler
3. If the semaphore was non-zero, it returns to the running task

The non-zero case means that the resource already had a resource that could be consumed, and so no threads will be waiting on it.

Jobbed also implements a second Semaphore Signaling algorithm, allowing a process to increase the semaphores value by an arbitrary amount, N. This is so that processes can signal that they have 'produced' many consumables without having to call the increment N times manually. In this case, the signaling algorithm is:

1. Adds N the semaphore's value
2. If the semaphore was 0, it wakes up to N tasks that are waiting on that semaphore and then it calls the scheduler
3. If the semaphore was non-zero, it returns to the running task

II.B.8 Priority Inversion Prevention

In order to combat priority inversion, *Jobbed*'s scheduler promotes the lower priority blocking thread to the priority of the high priority thread temporarily, and then returns it to the original priority once the resource has been released. This process is called *Priority Inheritance* and is one way to solve the Priority Inversion problem[16].

II.B.9 Deadlock Prevention - Eliminating Cyclic Wait

There are four conditions required for a deadlock: exclusive access to the resource, resources can be held and waited upon, processes do not give up the resource until they are finished, and each waiting process forms a cyclic dependency graph[17]. Since deadlocks only occur when two threads have one

resource and request the resource of the other, they establish a cycle. In order to prevent this from occurring, *Jobbed* gives each resource an ordering, and so it gives it an implicit numbering from the index in the queue, and ensures that resources can only in decreasing number. This means that when one of the threads tries to access a resource, it must first temporarily release the previous resources so that any other higher priority threads that want that resource can grab the resource as it gets released. Therefore, when accessing resources it is important to declare all of the resources needed for the critical section, the sections where exclusive access to a resource is necessary, before proceeding so that the scheduler can assure the exclusive access[13].

II.C Interrupts and Delayed Execution

Jobbed provides a `subscribe_irq` function that takes the requested Interrupt(IRQ), a callback function, and a pointer to the IRQ-specific parameters, such as priority level of the IRQ handler or the time to delay before executing the callback. Currently, *Jobbed* implements the following IRQ handlers[6]:

- UART - for handling data received on the UART pins
- GPIO - for rising edge triggers on requested pins
- Local Timer - For the system's count-down timer
- System Timer (x4) - One for each of the System Timer's compare registers, which can be used to delay the execution of the requested callback to n μs later

These interrupts allow the system to collect incoming stimuli from its environment as well as delay the execution of requested procedures to a later time.

Jobbed's interrupt handler does the following for each supported interrupt, continuing to the next interrupt type if any listed condition is not met:

1. Checks if the interrupt has a callback
2. If it has a callback, checks if the interrupt is active

[6]See V Future Work for future IRQ handlers being explored such as GPIO falling edge, Ethernet packets, etc.

3. If the interrupt is active, adds the callback to the requested priority queue with any data

Additionally, if the interrupt was one of the system timer interrupts, it checks to see if the callback is a one-shot timer or a periodic timer. If it is periodic, it uses the original delay specified when the user subscribed the timer to determine the next compare value. Because of this behavior, periodic and one-time delayed executions have the same jitter timings.

III Evaluation against Linux

Now that the RTOS tests have been established and *Jobbed*'s core functionality has been discussed, we will see how it compares to *Raspbian*, the Linux Operating System for the Raspberry Pi. Both systems were tested on the same Raspberry Pi 2B (v1.1) and in an unloaded state so that their core functionalities can be accurately bounded. In each of the tests, the thread policies for *Raspbian* were set to Real-Time FIFO policies to ensure that it is using the Real-Time Scheduling Mechanisms, and so the analysis can be accurately compared for prospective Real-Time Applications.

III.A Quantitative Results

Each test results comes from 4096 benchmarking samples.

Test	Linux			Jobbed		
In nanoseconds:	Avg	σ	Max	Avg	σ	Max
Tracing	238	160	2920	287	452	1000
Thread Creation (L)	96600	26100	825000	1880	967	5000
Thread Creation (H)	221000	56500	1470000	2660	887	7000
Thread Switch	15400	625000	40000000	1420	742	4000
Mutex Create	317	1040	41400	561	595	3000
Mutex Destroy	321	1300	45100	576	585	3000
Mutex Lock (C)	23700	6950	206000	4910	1020	9000
Mutex Lock (NC)	365	309	8440	798	597	3000
Mutex Unlock	400	782	50100	1370	572	4000
Semaphore P	391	1550	95800	427	495	1000
Semaphore V	2550	125000	7860000	768	421	1000
Semaphore V (NZ)	313	223	11700	439	496	1000

Table 2: RTOS Core Functionality Benchmarking Results

As a first note, the tracing results, i.e. the amount of time it takes to write down the time, were very similar amongst both machines. The most accurate timer on the Raspberry Pi 2B is the system timer, which returns μs values[5, 12]. Other timers available on the system are the ARM timer and the cycle counter. The ARM timer is derived from the system clock and so it can change its timing dynamically[5]. The system cycle counter can be used to infer the amount of time that has passed but is not completely accurate below 1μs[5]. Linux utilizes these other timers to infer below the 1μs resolution provided by the most accurate timer, and thus allow it to report sub microsecond times but these measurements are only holistically accurate within a microsecond. Despite this slight difference in timing method, the average tracing time was roughly the same and had deviations below 1μs and so the tracing time will not be a significant factor to differences in the results.

III.A.1 Thread Results

In all of the Thread results, Linux had higher average runtimes, wider deviations in runtimes, and much larger maximum runtimes than *Jobbed*. There are many contributing factors to these much larger latencies, most notably are: ASLR, parent-child relationships, multiple scheduling policies, dynamical allocation of the thread's stack space, copy the parent thread's attributes, and setting given attributes. The ASLR was mentioned earlier, which means that the Linux system has its address space randomized which may contribute a small part to the discrepancy. Linux also allows threads to spawn child threads, cloning the attributes of the parent thread. *Jobbed* on the other hand treats every thread as a standalone process and does not utilize child processes in order to ensure a more deterministic system. Linux also contains many more attributes than *Jobbed*, such as the scheduling policy for that particular task. These extra attributes are derived from POSIX standards, which are highly general standards for general operating systems. This generality means that the standard implements attributes that are not employed in real-time systems, where the requirements are much stricter and specific. Linux also implements a multitude of different scheduling policies are also not present in *Jobbed*. *Jobbed* does not support the multitude of different scheduling policies since the system is Real-Time and so it must employ deterministic scheduling behaviors. The dynamical allocation of the thread's stack is another large component of this latency since the Linux kernel must find a stack space for the new thread, whereas that is statically determined in *Jobbed*. During this dynamical allocation of the stack, the MMU must also be updated to ensure that the process will have privileges to this stack and also the Linux system initializes stack guards for the process to determine if the stack has been overrun, which contributes to larger overhead.

The maximum runtimes for *Jobbed* were much smaller, being on the order of microseconds whereas the Linux benchmarks were around the order of milliseconds. This large bound indicates that the execution times do not have a bound, and thus are nondeterministic on Linux.

III.A.2 Mutex Results

For the Mutex creation and deletion tasks, the average runtime and standard deviation is comparable between the operating systems, being less than a μs, but maximum runtime for Linux was much larger than in *Jobbed*. In the case of the average runtime, *Jobbed* was marginally slower which is most likely due to the fact that it moves a system managed entry to a list whereas Linux's Mutex object was created at compile time of the process. Because of this static allocation, Linux's mutex creation/deletion, or *initialization/destruction* since the mutex already exists, is marginally faster but still within 1μs. The deviations are also within a microsecond, in favor of *Jobbed*, and so the discrepancy may be due to the slight ambiguity in these sub-microsecond measurements. The larger maximum time hints at the non-determinism of the Mutex creation/ destruction on Linux.

The non-contended lock and unlock tests also have Linux as marginally faster (within a microsecond) on average. The deviations were comparable, with *Jobbed* being less than a microsecond larger for the locking and Linux being less than a microsecond larger for the unlocking. The largest differences were in the maximum times, where Linux took longer than 3μs longer to lock a non-contended mutex and 46μs longer to unlock a mutex when comparing the longest execution times. This major discrepancy is another indication of the non-determinism of the Linux system.

The contented lock on Linux was roughly 19μs slower on average than *Jobbed*, had a wider standard deviation, and the longest execution time scenario was roughly 196μs slower. This large upper bound is also indicative of the unboundedness to the execution of the process, hence implies that the *Raspbian* Operating System contains non-deterministic processes.

III.A.3 Semaphore Results

The semaphore wait had comparable execution times, with Linux being less than a microsecond faster on average, with a wider deviation (roughly 1μs wider), and 94μs slower than *Jobbed* in the longest execution case. Since the

average execution times are below a microsecond in their difference, the average case is not significantly different but the larger maximum further highlights the wider bounds, if any, on Linux.

The semaphore signal when the semaphore is non-zero (NZ) to start are roughly comparable as well, with Linux being less than a microsecond faster, having less than a microsecond narrower deviation, but having a 10 microsecond slower longest execution case. As with the before test, the average and deviations in the times are less than a millisecond and so they are not significantly different. The main difference is that Linux's longest execution case has higher bound than *Jobbed*.

The big difference comes from signaling a semaphore that has a zero counter. In this case, *Jobbed* runs faster by more than a microsecond on all three metrics. In the case of the longest execution case situations, *Jobbed* runs 7.86 milliseconds faster than Linux, which further highlights the non-determinism of Linux's semaphores. Apart from the longest execution case, the average execution time is roughly 1.7 microseconds slower on Linux but the standard deviation of the average is 124 microseconds wider. In this case, *Jobbed*'s semaphore signaling performs way better than Linux and has a much tighter bound.

III.A.4 Overall RTOS Test Results

From the quantitative results, we see that when *Jobbed* is slower on average it is within 1μs of *Raspbian*. On the other hand, there are 4 results for *Raspbian* that are slower than *Jobbed* and exceed 14μs difference in average execution time. Further, *Jobbed*'s deviation follows the same pattern, staying within 1μs of *Raspbian* when it is slower but the converse is not always true.

The most obvious discrepancies are for the maximum execution times, where *Jobbed* always has a smaller upper bound on the execution of the various processes, three of which are faster by at least 1 millisecond.

III.A.5 GPIO Quantitative Test Results

Other than the standard RTOS tests, we also tested both Linux and *Jobbed*'s ability to respond to fast and slow sequential GPIO pulses. To generate these 20μs to 2ms pulses, we attached pin 13 of an ARDUINO Uno to the 16th GPIO pin of the Raspberry Pi and had both systems create callbacks for rising edge events on that pin. We also include the minimum execution times to highlight Linux's lower bound behavior.

Test	Linux				Jobbed			
In microseconds:	Avg	σ	Min	Max	Avg	σ	Min	Max
20μs pulse	68	39.3	55.7	1230	25	1	22	32
40μs pulse	67.9	45.6	58.1	2240	45	1	43	52
160μs pulse	167	23.5	76.3	999	166	2	163	174
1ms pulse	1020	359	620	24000	1010	1	1000	1020
2ms pulse	2040	865	255	57100	2030	1	2010	2040

Table 3: GPIO Benchmarking Results

These tests were performed on systems that were unloaded and prioritized the requested GPIO pin rising edge callback. In all cases, *Jobbed*'s average, best, and longest execution case timings were better than Linux's and also had a significantly smaller deviation across the 4096 samples. Additionally, from Linux's minimum time on the 40μs pulse, we see that Linux incurs at least 18μs overhead for handling the pulse. We also see that the 20μs and the 40μs have similar results on the Linux system, indicating that we have started to hit a responsiveness threshold for GPIO pulses around the 68μs range. Further, Linux's responsiveness to the slower 2ms pulse has a minimum of 255μs, which is well below the period length of 2ms. This in conjunction with the fact that the maximum took 57ms indicates that Linux is missing some of the longer pulses, and so the elapsed time can widely vary, leading to abnormally fast and slow pulse length readings. On the other hand, *Jobbed*'s handling of the pulses has a range of around 10μs for short pulses and 37μs for longer pulses. The difference for the longer pulse is due to the system going into a reduced power

mode, due to the `wfe` idling, while waiting for an event to occur. *Jobbed*'s tighter bounds further emphasize the benefit of using *Jobbed* for real-time processes over Linux since it is able to recognize each pulse in the 20μs-2ms range, whereas Linux shows that it may miss pulses for long lengths and it has a minimum semi-consistent pulse length of around 68μs.

III.A.6 Delayed Execution Test Results

Test	Linux				Jobbed			
In microseconds:	Avg	σ	Min	Max	Avg	σ	Min	Max
10μs delay	690	89.2	610	1560	10.3	0.925	8	14
20μs delay	464	90.7	383	1410	20.4	0.975	18	24
50μs delay	729	79.3	642	1770	50.4	0.963	48	54
100μs delay	781	76.0	697	1600	100	0.853	98	104
200μs delay	897	160	800	9770	200	0.789	198	204
500μs delay	1330	98.5	1130	2150	500	0.935	498	504
1ms delay	1860	124	1640	2840	1000	0.896	998	1000
5ms delay	6010	178	5670	14800	5000	0.875	5000	5000
10ms delay	11000	122	10700	12300	10000	0.915	10000	10000

Table 4: Delayed Execution Benchmarking Results

These tests were performed on systems that were unloaded and prioritized the requested delayed callback. Much like the GPIO tests, we see that *Jobbed* performs better on all accounts in comparison to Linux. We especially see that Linux struggles to consistently hit delays below 500μs and at higher delays it incurs roughly an additional 1ms delay. On the other hand, *Jobbed*'s bounds range within 4μs from the requested time across the entire 10μs to 10ms range and has an average close to the expected value and with a standard deviation below 1μs.

III.B Priority Inversion Test

The other main behavior comparison test is to check both Operating Systems for the Priority Inversion execution condition laid out in Figure 1.

In *Jobbed* execution proceeded in the following manner:

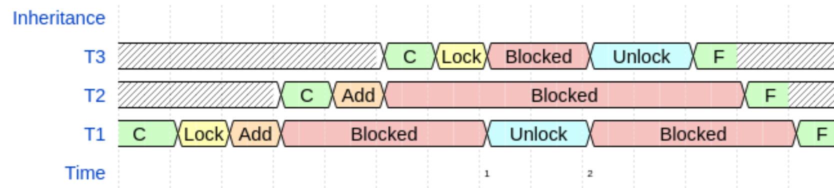


Figure 4: Jobbed Execution Path

As mentioned in section II.B.8, *Jobbed* employs priority inheritance, where the process that is responsible for blocking a higher priority process is elevated to the higher priority's permission until the contended resource is released. As we can see in Figure 4, when T_3 tries to lock the resource at time, it gets blocked and T_1 begins to run. In this case, T_1's only task is to unlock the resource, and when the resource is unlocked, T_1 gets re-blocked and T_3 continues to execute. Then, when T_3 is finished, T_2 finishes and then T_1 finishes, thus the threads finished in priority order, which prevents the Priority Inversion condition. Therefore, *Jobbed* is not susceptible to priority inversion.

On the other hand, when running this process 4096 times on Linux, we saw the same execution completion order: 213. This indicates that not only the medium priority task finished before the high priority task, but also that the low priority task continued to execute despite releasing the resource that the higher priority task required to continue. In a general purpose operating system, this out-of-order execution is not as bad since tasks are not assumed to have hard deadlines or have a preferred execution order. For a Real-Time System, deterministic execution chains are important and so it is imperative that the tasks always finish in the 321 order, which was never observed on the Linux tests.

IV Qualitative Evaluation of Jobbed

The following tests are more general tests of the RTOS and so it does not make as much sense to evaluate them versus Linux. The main reason the Priority Inversion Qualitative test was evaluated on the Linux system was to highlight the non-determinism present on the OS.

IV.A Thread Prioritization Analysis

In section I.D, we had also mentioned that the tests important for an RTOS include where it places threads of the same priority, and if a low priority thread is created, is it blocked until the higher priority threads are finished?

To test both of these cases, we had the medium priority thread T_2 in the priority inversion test create a thread T_4, with the same priority as T_1. In all of our runs, T_4 completed after T_1 completed, verifying that new threads of the same priority are placed at the TAIL of the queue and that new threads are blocked from running until there are no higher priority threads.

The qualitative tests also included testing if higher priority threads are immediately run after creation, which is shown in the priority inversion test, and the last test seeks to determine if a thread yields and has the same priority as another thread, is the other thread ran. In our testing, and from how *Jobbed* was written, when two threads share the same priority and one yields, it is placed at the tail of the queue and so the other thread is allowed to run.

Thus, the qualitative tests are successfully completed by *Jobbed*, and in a way consistent with RTOS assumptions.

IV.B Deadlock Test

When running the Deadlock Test we got the following execution path:

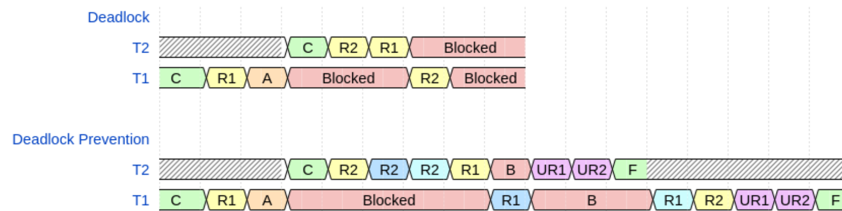


Figure 5: Deadlock Test Execution Path

The purple UR1 and UR2 means that the thread unlocks R1 or R2 respectively. The **A** indicates that T_1 creates T_2. The darker blue means that the *mutex lock* call automatically unlocked the resource and the lighter blue means that the *mutex lock* call automatically relocked the resource. Specifically in our diagram, the dark blue R2 in the second timing diagram indicates that the system mutex manager automatically unlocks R2 when T2 is trying to acquire R1, in order to see if any higher priority threads are waiting on R2. Since no higher priority threads are waiting on R2 it is also automatically relocked, shown by the lighter blue R2. Similarly, the dark blue R1 in the second timing diagram indicates that the system mutex manager automatically unlocked R1 when T1 is trying to acquire R2 in order to see if any higher priority threads are waiting on R1. In this case, T2 is waiting on R1, so it wakes up and completes its task. Execution then resumes on T1 and it is able to relock R1 and then successfully acquire R2, with the deadlock prevented.

Thus, the implicit unlocking prevents the tasks from establishing a complete cycle, eliminating one of the necessary conditions for a deadlock[17]. RTOS application designers should then ensure that they are locking all of the resources they need in each critical section at the beginning of the critical section. I.e. if a task needs a printer and a copier in one section, they should reserve them at the beginning of that section. Further, if they have another section where they don't need the copier but also need a UART connection, they should reserve those at the beginning of this other critical section. This approach ensures that *Jobbed*'s managed mutexes are not susceptible to the deadlock condition.

IV.C Semaphore Signaling Test

In Section I.D, the semaphore test was proposed in order to ensure that the semaphore only blocks threads when it can provide no more resources, i.e. when its counter is zero. In our experiments, *Jobbed*'s execution path was (with the semaphore initialized to zero):

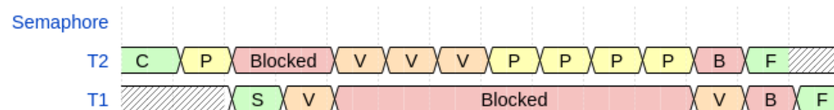


Figure 6: Semaphore Test Execution Path

Thus, we see that T_2 was only blocked when the semaphore counter was zero, after the first **P** and after the fourth **P** in the series of sequential **P**s. This is consistent with the expected behavior of a semaphore in an RTOS, ensuring the determinism of *Jobbed*'s semaphores.

IV.D CPSR Preservation Test

One of the context preservation mechanisms that are more difficult to directly measure is the CPSR. Ensuring that the program counter, stack pointer, and general registers is much easier since failures are typically spectacular. On the other hand, improperly preserving the CPSR may only cause a process to take a different execution path in very rare cases, causing cryptic errors. This rarity is generally be due to the natural grouping of compare and branch instructions, leaving a small surface for preemption to express this failure condition. In order to test that the scheduler properly stores and then restores the thread's CPSR upon preemption, we performed the following test, using ARM Assembly to ensure that the compare and branch instructions were separated by a context switch:

1. Set register 0's value to 5
2. Compare register 0's value with 5
3. Add a higher priority task, preempting the current thread

   (a) Set register 0's value to 5
   (b) Compare register 0's value with 4
   (c) End task

4. If r0 == 5, i.e. the CPSR's zero flag is set, output 'Y'
5. Otherwise, i.e. the CPSR's zero flag is not set, output 'N'

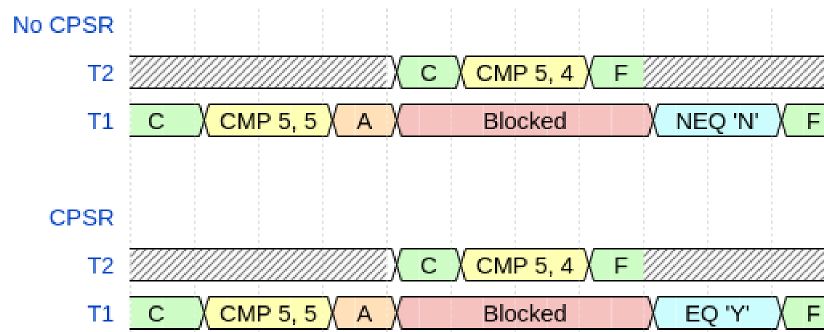


Figure 7: CPSR Test Timings

Figure 7 shows the two execution possibilities of the test. The first timing diagram, labeled No CPSR, shows that if the CPSR is not preserved then upon returning to T_1 the Not Equal branch will execute and 'N' will be output since T_2 altered the zero flag. The second timing diagram, labeled CPSR, shows that if the CPSR was preserved then upon returnint to T_1 the Equal branch will execute and output 'Y' since the zero flag will be restored to T_1's state before the thread switch occurred.

Jobbed passed this test and output 'Y' for all runs of this test, proving that the CPSR was successfully saved and restored upon returning to the thread. If the CPSR was not properly set, the second thread's CPSR would have cleared the zero flag since 5≠4 and so the original thread would have output 'N' in all tests, which would indicate a source of non-determinism since the flags would not be assured to stay constant over the execution of the thread.

V Future Work

In order to strengthen the security posture of this system, security features such as Address Space Layout Randomization and user stack guards are being explored so that any systems that are in an insecure network or contain insecure process routines can be hardened against potential exploits that take advantage of the static layout of the system. Apart from the security features, future work will be focusing on implementing deterministic drivers for the different peripherals of the system such as SPI, I2C, Ethernet, USB, and SD filesystem drivers. In addition to the drivers for these different peripherals, including any supported IRQs are also being focused on so that these peripherals can generate interrupts that can be turned on from the userspace if requested. These are important so that the full range of the Raspberry Pi can be used in the RTOS applications without developers manually writing these routines to get that functionality, and thus may introduce non-deterministic drivers to handle these processes. Putting these nondeterministic processes at the lowest priority level and putting tasks that depend on determinism/ bounded execution at higher priorities can ensure that these tasks do not conflict with critical components of the RTOS application.

Additionally, the boot process also provides the kernel with ATAG entries, which are currently ignored but are useful to get information about the hardware. Future releases are going to seek to use the ATAGS to get a more holistic picture of the system and can allow better tweaking of the execution based on them. This information includes things like how large the memory is, what type of board the system is on, and thus it can load code for different boards in the same compiled *kernel7.img*, etc.

Future releases will also explore waking up additional cores using the ARM Mailboxes for distributing loads and selecting cores to receive specific IRQ signals. Additionally, putting mutex queues on mutex objects rather than the scheduler will be explored to assess the space versus time costs of having mutex wait lists only on the scheduler versus on the mutex. In the same vein, this may also trigger an exploration into using a doubly linked list for the queues

rather than a singly linked list for backwards traversal optimizations should the queues be put on individual mutexes rather than the scheduler, and also make removing specific queue items quicker.

Future non-RTOS specific contributions that are being explored involve making much more syntactic sugars for RTOS developers to easily specify deadline and jitter conditions with try-catch blocks for different latencies.

VI Conclusion

From autonomous vehicles to flight controllers, Real-Time Operating Systems are widely used due to their determinism, bounded execution times, and emphasis on low latency execution of operating system tasks. The Raspberry Pi 2B is a Cortex-A7 SBC that is popular due to its small form factor, low cost, and high performance for its cost. Current Raspberry Pi Operating Systems focus on the user experience, which emphasizes user-responsiveness, and adhere to a general-purpose operating system paradigm. In order to bring a Real-Time Operating System, we presented *Jobbed* and tested it on a wide variety of metrics to ensure it adheres to expected RTOS behavior, has tight execution bounds, and prevents system faults that naïve scheduling algorithms may introduce. We also compared *Jobbed* to *Raspbian*, the standard operating system for the Raspberry Pi with Real-Time Policies, to compare both system's boundedness for tasks and average execution times. Through our testing, we show that when *Jobbed* is slower than *Raspbian* at a task, it is within one microsecond which is equal to the level of accuracy in timing the system. On the other hand, when *Jobbed* is faster than *Raspbian*, it is most often quicker by at least 10 microseconds, or much more. In addition, in all of the tests, we found that the longest execution case, execution times were always much smaller for *Jobbed*, and so it has a much tighter execution bounds, sometimes by milliseconds. When testing both system's responsiveness to stimuli or ability to delay executing a callback to $n\mu$s later, we also found that *Jobbed* also had much tighter jitter bounds, and average executions that were comparable to the target pulse or delay. In the behavioral tests, we found that *Jobbed*

achieves all of the expected behavior of an event-driven RTOS: the highest priority thread is running at any given time, it is resistant to priority inversion faults, and it prevents thread deadlock conditions by removing the cyclic wait condition. Thus, we found that *Jobbed*'s performance as an RTOS for the Raspberry Pi 2B provides a much better OS for Real-Time applications than the current OS for the Raspberry Pi, and fulfills the requirements for an RTOS.

References

[1] Alasdair Allan. *The config.txt file*. https://www.raspberrypi.com/documentation/computers/config_txt.html. [Online; accessed 25-March-2022]. 2021.

[2] ARM. *ARM Synchronization Primitives Development Article*. https://developer.arm.com/documentation/dht0008/a/swp-and-swpb/legacy-synchronization-instructions/limitations-of-swp-and-swpb. [Online; accessed 25-March-2022]. 2020.

[3] ARM. *Cortex-A7 MPCore Technical Reference Manual r0p5*. https://developer.arm.com/documentation/ddi0464/f/?lang=en. [Online; accessed 23-March-2022]. 2011-2013.

[4] ARM. "ISAs, comminality and differentiation". In: (2020).

[5] Broadcom. "BCM2835 ARM Peripherals". In: (2012).

[6] Daniel Casini et al. "Response-time analysis of ROS 2 processing chains under reservation-based scheduling". In: *31st Euromicro Conference on Real-Time Systems*. Schloss Dagstuhl. 2019, pp. 1–23.

[7] Francis M David, Jeffrey C Carlyle, and Roy H Campbell. "Context switch overheads for Linux on ARM platforms". In: *Proceedings of the 2007 workshop on Experimental computer science*. 2007, 3–es.

[8] Raspberry Pi Foundation. *Welcome to Raspbian*. https://www.raspbian.org/. [Online; accessed 27-March-2022].

[9] FreeRTOS. *What is an RTOS?* https://freertos.org/about-RTOS.html. Online; accessed 26-March-2022].

[10] John Goodacre and Andrew N Sloss. "Parallelism and the ARM instruction set architecture". In: *Computer* 38.7 (2005), pp. 42–50.

[11] Global Market Insights. *Single Board Computer Market Size*. https://www.gminsights.com/industry-analysis/single-board-computer-sbc-market. [Online; accessed 20-March-2022]. 2021.

[12] Gert van Loo. "Quad-A7 Control". In: (2014).

[13] Peter Marwedel. *Embedded system design: Embedded systems foundations of cyber-physical systems*. Springer Science & Business Media, 2010.

[14] Les Pounder. *Pi in Short Supply: Site Claims 52-Week Wait for 4GB Model*. `https://www.tomshardware.com/news/raspberry-pi-4-in-short-supply`. [Online; accessed 20-March-2022]. 2021.

[15] Glenn E Reeves. "What really happened on Mars?" In: (1998).

[16] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. "Priority inheritance protocols: An approach to real-time synchronization". In: *IEEE Transactions on computers* 39.9 (1990), pp. 1175–1185.

[17] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system principles*. John Wiley & Sons, 2006.

[18] OSDev Wiki. *Raspberry Pi*. `https://wiki.osdev.org/ARM_RaspberryPi`. [Online; accessed 26-March-2022].

[19] OSDev Wiki. *Raspberry Pi Bare Bones*. `https://wiki.osdev.org/Raspberry_Pi_Bare_Bones`. [Online; accessed 26-March-2022].

[20] Paul R Wilson et al. "Dynamic storage allocation: A survey and critical review". In: *International Workshop on Memory Management*. Springer. 1995, pp. 1–116.

[21] Niklaus Wirth. "Toward a discipline of real-time programming". In: *Communications of the ACM* 20.8 (1977), pp. 577–583.

[22] Mingyang Zhang et al. "Which Is the Best Real-Time Operating System for Drones? Evaluation of the Real-Time Characteristics of NuttX and ChibiOS". In: *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE. 2021, pp. 582–590.

VII Appendix

VII.A Useful Resources

- https://www.raspberrypi.com/documentation/computers/processors.html
- https://github.com/dwelch67/raspberrypi
- https://github.com/raspberrypi/firmware/wiki/Mailbox-property-interface\#clocks
- https://jsandler18.github.io/extra/mailbox.html
- https://github.com/eggman/raspberrypi/tree/master/qemu-raspi2
- https://github.com/s-matyukevich/raspberry-pi-os
- https://github.com/bztsrc/raspi3-tutorial
- https://github.com/rust-embedded/rust-raspberrypi-OS-tutorials
- https://elinux.org/RPi_Framebuffer
- https://wiki.osdev.org/ARM_RaspberryPi
- https://developer.arm.com/documentation/ddi0406/c/System-Level-Architecture/The-System-Level-Programmers--Model/ARM-processor-modes-and-ARM-core-registers/ARM-core-registers?lang=en
- https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/index.html
- https://medium.com/@thiagopnts/raspberry-pi-bare-metal-programming-with-rust-a6f145e84024

VII.B Where to get the Latest Release

The latest release, including the source, tests, and the current assembled binary, can be found on my website https://christiancunningham.xyz in the form of a Gzipped Tarball download link.

VII.C Using the Clock Cycle Counter to Benchmark *Jobbed*

Test	Linux			Jobbed		
In nanoseconds:	Avg	σ	Max	Avg	σ	Max
Tracing	238	160	2920	77.9	2.13	157
Thread Creation (L)	96600	26100	825000	744	48.1	2270
Thread Creation (H)	221000	56500	1470000	521	45.4	3100
Thread Switch	15400	625000	40000000	682	40.9	2150
Mutex Create	317	1040	41400	298	48.4	1300
Mutex Destroy	321	1300	45100	257	20.5	1290
Mutex Lock (C)	23700	6950	206000	2820	33.3	4430
Mutex Lock (NC)	365	309	8440	422	61.0	1400
Mutex Unlock	400	782	50100	430	3.96	540
Semaphore P	391	1550	95800	179	3.52	334
Semaphore V	2550	124000	7860000	421	4.32	1080
Semaphore V (NZ)	313	223	11700	179	3.05	190

Table 5: RTOS Core Functionality Benchmarking Results with Cycle Counter

As mentioned in the benchmarking section, the Raspberry Pi has an accurate counter and also provides things like a cycle counter which can infer the time taken for a process. The cycle counter can be used to infer the time taken for different processes, with a bit more inaccuracy, below 1μs. When using the Cycle counter on *Jobbed*, as it is used in Linux, the results for *Jobbed* only get better. I also obtained these results with 8192 samples for *Jobbed*.

VII.D Jobbed running in QEMU

```
Machine   View
Jobbed v0.9z           MMU Initialized!
00  00000000 00000000 00000000 00000000
UART STIMERCMP LTIMER
TIMER @ 62500 kHz 4293935787  | FFF042AB
VIDEO 1920x1080 RGB
SVC      IRQ      FIQ      User/SYS
00011FE8 00014000 00015000 00016000
Status Updated by Core #00000000
Sys Timer Status 0000000000000000000004201 00000000|0


44.382 ns     108.980 ns    297.821 ns    496.696 ns    185.420 ns    165.206 ns    105.408 ns    89.032 ns     167.173 ns    403.508 ns    143.171 ns    426.359 ns
45.891 ns     44.739 ns     45.787 ns     46.340 ns     46.027 ns     47.004 ns     46.537 ns     46.685 ns     46.875 ns     45.455 ns     46.235 ns     45.760 ns
3721.401 ns   4723.333 ns   4671.220 ns   4617.923 ns   4541.147 ns   4700.109 ns   4491.292 ns   3892.222 ns   4763.478 ns   4619.034 ns   4757.959 ns   4691.256 ns
```

Figure 8: Jobbed in QEMU running the tests with the Cycle Counter

Note: the benchmarking results listed above using the cycle counter or the system timer came from the results on the Physical Hardware since QEMU does not provide the accurate timings.

VII.E Testing Setup w/ Arduino and Push Button

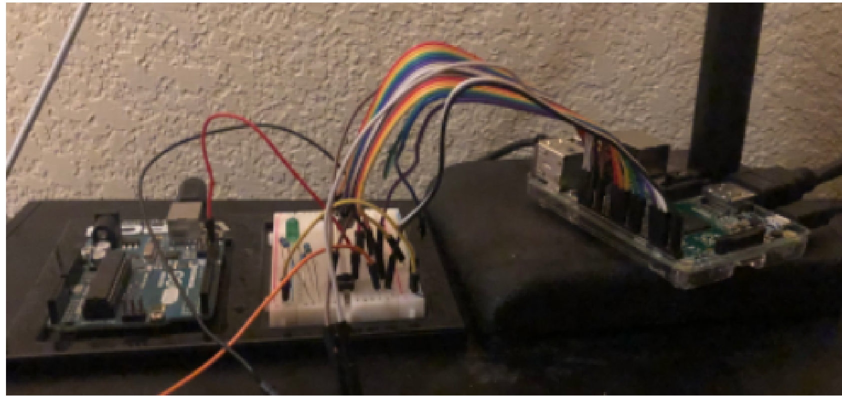


Figure 9: Setup

Pictured above is the setup where the arduino is hooked up to pin 16 and a pushbutton is hooked up to pin 12 with an LED for manual verification of signal reception.