CS 452

Taylor Stark
tstark
20437098

Path to Executable

On the linux.cs.uwaterloo.ca server:

- /u/cs452/tftp/ARM/tstark/rtos.elf

If for some reason the executable is not present, the source code can be compiled in:

- cd /u/tstark/cs452/kernel/
- sh cmake_setup.sh
- cd release
- make

The executable will now be located in /u/tstark/cs452/kernel/release/bin/rtos.elf

Path to Source Code

https://git.uwaterloo.ca/tstark/CS452-Kernel

How to Operate the Program

Our kernel *must* be loaded at the default address (0x218000). Once loaded, the kernel can be executed with the “go” command.

```
load -b 0x00218000 -h 10.15.167.5 “ARM/tstark/rtos.elf”
go
```

Terminal

The program initializes as a mock terminal.

The GUI shows:

- the time since the start of the program
- the % time that the system is idle
- the current state of the train switches
- a one-line command prompt

The terminal supports 6 commands

- tr <train_no> <train_speed>
    - set the speed of a train
    - train_no - the number of the train on the track (supported 1 to 80)
    - train_speed - the desired speed of the train (supported 0 to 15)
- sw <switch_no> <switch_position>
    - set the position of a switch
    - switch_no - the switch on the track (supported 1 to 22)
    - switch_position - the position to switch to (supported C or S)
- rv <train_no>
    - reverse the speed of a train
    - train_no - the number of the train on the track (supported 1 to 80)
- go <train_no> <sensor>
    - Has the train drive from it’s current location to the sensor
    - The train will stop at the sensor
- rt <train_no>
    - Has the train generate a random sensor and drive to the random sensor
    - The train will stop at the sensor
    - When the train has reached its destination, it will generate a new random sensor to drive to
- q
    - quit the program

Data Structures

Circular Buffer

Circular buffers are an efficient data structure that can be used to emulate both a queue data structure and a list data structure, depending on your need. The circular buffer attempts to be as efficient as possible by copying memory in chunks, instead of 1 byte at a time. The circular buffer is essentially a wrapper around a standard C array. This allows the programmer to optimize memory usage, as the programmer must construct the underlying array and pass this array to the circular buffer (i.e. the circular buffer will not allocate the underlying array for you). The circular buffer makes no assumptions about the array that it is given. The circular buffer just knows that it has been given a chunk of memory. This can be a benefit as the circular buffer is generic enough to be used to efficiently manage an array of INT, CHAR, struct FOOBAR, etc. However, this benefit has a drawback in that the circular buffer has no type safety - you may construct a circular buffer with the intent of only putting struct FOOBAR in to it, but then accidentally place an INT inside the buffer instead.

Priority Queue

The priority queue data structure is a wrapper around an array of circular buffers. The priority queue pushes, pops and peeks in O(1) time, independent of the number of priorities. This is possible by only allowing priorities that are of the form 2^n, and quickly calculating the log of the priority to index for the queue, using an algorithm described in http://graphics.stanford.edu/~seander/bithacks.html. The priority queue uses a bitmask to remember which queue has items and calculates the log base 2 of the bitmask to find the highest priority nonempty queue to peek and pop.

Linked List

The linked list data structure provides an abstract method of grouping arbitrary data. It is implemented as a doubly linked list with a void pointer data member, so the linked list node's data must be stored separate from the node. Our linked list implementation provides O(1) insertion and O(1) deletion. It does not bound the maximum number of elements.

RT Kernel

Implements the operating system services that run in kernel mode.

Stack Allocator

The stack allocator divides the memory regions 0x00400000 to 0x01F00000 into 64KB blocks. These blocks represent the region of memory for stacks to use. These blocks are placed on a queue, and when a task is created it uses a stack pulled from the queue. When a task is destroyed, the stack is put back on the queue so it can be re-used.

Task Descriptor

A task descriptor contains the following elements:

- Task ID - Unique identifier for this task
- Parent ID - Task ID of the task that created this task
    - For the first task, it has no parent. Therefore, its parent is set to be itself
- State - The current state of the task. Could be ready, running, or blocked
- Priority - Higher numbers mean higher priority
- Stack pointer - A pointer to the task's stack at the last point it was run
- Stack - pointer to the region of memory this task can utilize to allocate objects
- Mailbox - Messages addressed to this task that have not yet been picked up

Task Descriptor Allocator

The task descriptor allocator is implemented using a static array. Task descriptors have their ID's placed on a queue. When a task descriptor needs to be allocated, an ID is pulled from the queue. The ID is then used as a modulus index into the static array of task descriptors. When a task is destroyed, its task descriptor has its ID incremented by the number of possible task descriptors, and then placed on to the queue. This ensures that we can detect if a task tries to send a message to a dead process. Additionally, it allows task IDs to continue to correspond to an index inside the array.

Task Management

When a task is created, a task descriptor and stack are allocated. The task then has its stack initialized to be as if the task had made a system call. This involves setting the PC to be the task's start function, and setting the CPSR to be 0x10 (which will cause the task to run in user mode). Additionally, the LR is set to the Exit() system call so that tasks don't need to manually call Exit(). As a safety precaution, the stack is set up with a canary. This value will be checked whenever the task is about to execute to ensure that the task's memory has not become corrupted.

Scheduler

The scheduler maintains the ready priority queue. It is implemented as a priority round robin scheduling algorithm. That is, tasks at priority *n* will get executed in round robin fashion. When no tasks at priority *n* are ready, tasks at priority *n-1* will get executed in a round robin fashion, and so on and so forth. The scheduler also maintains a global variable of the currently running task. Frequently, other subsystems need to know which task is currently executing so they query the scheduler.

Context Switch

Running The Scheduler

The kernel is entered via an interrupt. This could be a software interrupt or a hardware interrupt. After servicing the interrupt, the interrupt handler can cause the scheduler to be run by restoring the kernel context. This can be done be restoring R4-R12 and the PC from the kernel's stack. Before running the scheduler, the kernel will check to see if control should be returned to Redboot.

Activating The Next Task

This is implemented by the function KernelLeave, in kernel.asm. All interrupts will need to setup the task's stack in this *exact order*.

- This function needs to know which task to activate. Therefore, the next task's stack pointer will be passed to this function in R0
- Store kernel state (R4-R12 + LR) onto the kernel's stack
- Switch to system mode
- Use the value in R0 to restore the task's stack
- Pop the task's PC and CPSR from the task's stack
- Switch back to supervisor mode
- Restore the task's PC and CPSR by placing them in the LR and SPSR, respectively
- Switch to system mode
- Restore task's state (R0-R12 + LR) from the task's stack
- Switch back to supervisor mode
- Jump back to the task

Software Interrupts

Software interrupts, also known as traps/swi/svc, are handled by the function TrapEnter, in trap.asm. This function is installed when the kernel performs initialization by writing the address of the TrapEnter function to address 0x28.

- R0-R3 contain system call parameters. R4 may contain a system call parameter, if the system call has 5 parameters
- Switch to system mode to get access to the task's stack

- Store task's context (R4-R12 + LR) on to the task's stack
- The hardware interrupt handler needs to store R0-R3, but the software interrupt handler does not. For compatibility with the hardware interrupt handler, decrement the stack pointer by 16 so that it looks like R0-R3 were stored on the stack
- Cache the task's stack pointer in a register
- Switch back to supervisor mode
- Save LR_svc and SPSR_svc on the task's stack (using the cached stack pointer)
- Get the current task and update its stack pointer
- Place the 5th system call parameter on the stack
    - This is where gcc will expect to find it.
- Calculate the system call number
- Use the system call number to calculate an offset into the system call table
- Get the system call by using the offset into the system call table
- Call the system call
    - What's important to note is that we have maintained the values of R0-R3, so the parameters will be passed properly to the system call
- The result of the system call is now in R0
- Remove the 5th system call parameter from the stack
- Store the result of the system call on the task's stack
- Run the scheduler, as detailed above

Hardware Interrupts

Interrupt Handler

Hardware interrupts are handled by the function InterruptEnter, in interrupt.asm. This function is installed when the kernel performs initialization by writing the address of the InterruptEnter function to address 0x38.

- Switch to system mode to get at the task's stack
- Store the task's context (R0-R12 + LR) on the task's stack
- Cache the task's stack pointer
- Switch back to irq mode
- Save LR_irq and SPSR_irq on the task's stack
- Switch to supervisor mode
- Get the current task and update its stack pointer
- Handle the interrupt
- Run the scheduler, as detailed above

Event Management

Interrupts are enabled when a task calls AwaitEvent on the given interrupt. In other words, if there is no task waiting for an interrupt, then there is no use in the interrupt being enabled, so it is disabled. Upon AwaitEvent being called, the current task is blocked and is designated as the handler for the interrupt. When the interrupt occurs, the interrupt handler will find the designated handler for the asserted interrupt. It will then unblock the handler, and disable the interrupt. The handler will once again need to call AwaitEvent for the interrupt to be enabled.

Timer Interrupts

The timer interrupt is configured to fire every 10ms. No special interrupt management is performed by the kernel. When the interrupt occurs, the designated handler is unblocked and the timer interrupt is disabled until AwaitEvent is called again.

UART Interrupts

No special interrupt management is performed by the kernel for COM2 transmit and receive interrupts. However, the kernel must keep additional state in order to properly handle COM1 transmission. The kernel maintains 2 variables: g_clearToSend and g_transmitReady. When the COM1 transmit interrupt is asserted, g_transmitReady is set to TRUE. When the COM1 modem status interrupt is asserted, g_clearToSend is set appropriately. When both g_clearToSend and g_transmitReady are TRUE, the interrupt is truly ready to be handled. The kernel will then unblock the designated handler.

Inter-Process Communication

Send

When a task tries to send a message to another task, a check is performed to see if receiving task is ready to receive. If so, the message is copied immediately to the receiving task's address space, the receiving task is unblocked, and the sending task is reply blocked. Otherwise, the message is placed into the mailbox of the receiving task and the sending task is reply blocked. In either case, the reply buffer is stored on the sending task's stack.

Receive

When a task tries to receive a message, the task checks its mailbox to see if there are any messages waiting for it. If so, the message is copied immediately to the receiving task's address space and the sender is set to be reply blocked. Otherwise, the receiver is blocked and the buffer is stored on the receiver's stack.

Reply

When a task tries to reply to a message, it verifies that the target task is reply blocked. If so, the reply is copied from the caller's address space to the target's address space. Otherwise, an error is returned.

RTOS

Implements the operating system services that run in user mode.

Idle

The idle task is the first task created by the first user task. It runs at the lowest priority, and is simply a while loop that continuously checks a flag. Other tasks can call IdleQuit(), which stops the idle task. This will in turn cause the kernel to quit, as no task will be available to run.

Name Server

The name server provides similar functionality as a DNS server. The name server allows tasks to register with a given name, and then allow tasks to look up the IDs of other tasks based off these well known names. The main problem is, if the name server is used to find the task IDs of other tasks, how do other tasks find the task ID of the name server? The answer is that the name server's task ID is hardcoded. Therefore, the init task must create the system tasks in a well known order.

The name server internally uses a hash table (with linear probing) for efficient lookups. This allows for worst case O(n) insertion and lookup, but in the average case it will be O(1) for insertion and lookup. There is no limit to the size of the name, as only a pointer to the name is kept for performance reasons. If the same name is registered twice, the previous entry is overwritten with the new entry.

Shutdown

RTOS provides a utility function to tasks, called ShutdownRegisterHook, that will allow a task to hook the system shutdown event. This will allow for tasks to perform any critical cleanup before the system exits (e.g. stopping all moving trains). It should be noted that this hook will be called in the context of the system shutdown task, and therefore the hook should simply be used to send a message to the appropriate task. The system shutdown task will give all tasks 1 second to perform any desired cleanup actions before quitting the idle task and thereby shutting down the system.

Clock Server

The clock server provides the Time, Delay and DelayUntil send wrapper functions. The clock server creates an internal clock notifier task, which continuously awaits for the ClockEvent. When the ClockEvent happens, 10ms has passed, and the notifier sends a message to the clock server to update the 10ms tick counter.

Time

The Time method sends a message to the clock server, which replies with the number of 10ms since the start of the program.

DelayUntil

The Delay method sends a message to the clock server with the tick count when it should be unblocked. The clock server puts the sender on sorted linked list based on the tick count. When the specified number of ticks has passed, it sends the reply, which unblocks the task.

Delay

The Delay method sends a message to the clock server with the number of ticks to wait. This number of ticks is added to the current total number of ticks, and then follows an equivalent path as the DelayUntil method.

I/O Framework

The I/O framework allows for different peripherals to be accessed in a uniform manner. A driver should register with the I/O framework before creating any I/O tasks. Consumers of the I/O framework should first call Open() to retrieve a handle. This handle is then passed to the Read() and Write() functions to determine which task to send a message.

Driver Registration

Drivers are responsible for handling the Open() function call. Therefore, a driver must register with the I/O framework before any calls to Open() may be made.

Open

The Open() function call is used to create handles to the I/O tasks that implement the Read() and Write() function calls. Essentially what this means is that a driver will create a read task and a write task. Then, when Open() is called, the driver will utilize the name server to query for handles to these tasks.

Read

The I/O read task creates a notifier task at the highest system priority. The notifier in turn creates a courier at a lower priority, which it will utilize to communicate with the I/O read task. This ensures that the notifier is almost always event blocked, which minimizes the likelihood of missing an interrupt.

The I/O read task handles buffering of unread data and queueing of clients waiting for data. Since the I/O read task performs buffering for the entire system, the implementation of many consumers is trivial. For example, when querying train sensors, the I/O read task is smart enough to know that it must wait for all 5 bytes to appear before unblocking the consumer. This reduces the overhead due to not having to call Read() for each character that needs to be read.

Write

The I/O write task creates a notifier task at the highest system priority. When a transmission event occurs, the notifier sends a message directly to the I/O write task. This is due to the fact that no interrupt can occur until the I/O write task has performed the write, and the notifier must be blocked so that it can’t call AwaitEvent so that the transmission interrupt remains disabled.

The I/O write task handles buffering of data to be sent out on the wire, and the blocking and unblocking of the notifier task. The I/O write task is more efficient than the trivial implementation, since more than 1 byte may be buffered in a single call to the Write() function. This helps to reduce message overhead.

UART Server

The UART servers are implemented using the I/O framework, with specific implementations for opening a handle to an UART server, reading from an UART peripheral, and writing to an UART peripheral. That is, the UART module first registers itself as a driver with the I/O framework. The UART module then creates 4 tasks: 1 for reading from COM1, 1 for writing to COM1, 1 for reading from COM2, and 1 for writing to COM2. This is done to minimize any hot spots in the system, as the I/O requests should hopefully be distributed uniformly across these 4 tasks.

Train Controller

Train Server

The train server provides the TrainSetSpeed and TrainReverse wrapper functions to set the train speed and reverse the train direction. The train server tries to keep track of the speed of any trains it knows about.

When the train server starts, the server registers a shutdown callback to stop running the server. Next, it sends a go command to the track before continually listening for TrainSetSpeed and TrainReverse commands. When the program shuts down, the server sets the speed of all trains to 0 and sends the stop command to the track before exiting.

TrainSetSpeed

The TrainSetSpeed() function sends a message to the train server validating and passing the train and the desired speed. The train server sends the two byte command to the track and updates its internal train speed state.

TrainGetSpeed

The train server maintains the speed that each train is travelling at. It exposes the TrainGetSpeed() function to allow other tasks to query for a train’s speed.

TrainReverse

The TrainReverse() function sends a message to the train server validating and passing the desired train to reverse. The train server responds by saving the previous train speed, setting the train speed to zero, waiting several seconds for the train to stop, sending the reverse command, and then setting the train speed back to the previous speed.

Switch Server

The switch server provides the SwitchSetDirection() send wrapper function to set the direction of a switch. Internally it keeps a list of all known switches. Helper functions are used to map the actual switch number to an index in the switch list.

When the switch server starts, the server initializes all switches to the curved direction, and turns off the solenoid before continuously listening for SwitchSetDirection requests.

SwitchSetDirection

The SwitchSetDirection() function sends a message to the switch server validating and passing the desired switch and direction. The switch server responds by sending the correct 2 byte command to the track, sending the command to disable the solenoid, and updating its internal switch state list.

Sensor Server

The sensor server allows for tasks to register for changes in sensor states. This is done using a function called SensorAwait() which is semantically identical to AwaitEvent(). If a task wants to react to changed sensors, it will create a helper notifier task that will continuously call SensorAwait(). The helper notifier task will then send a message to the actual server for processing.

The sensor server reads sensor data continuously from the COM1 port. It reads 10 bytes in a row before performing a delta on the previous sensor values and the current sensor values. Any changed sensors are sent to tasks that have called SensorAwait(). This entire process takes about 50 milliseconds due to the baud rate of COM1.

Attribution Server

The attribution server is responsible for knowing which sensor is expected to be tripped next by each train. The attribution server keeps track of known moving trains as well as trains that have not yet been located. To perform this task, the attribution server creates a sensor notifier that will let it know when a sensor is triggered. The attribution server exposes a function called AttributedSensorAwait() for tasks that need to receive updates when a train triggers a sensor. If a triggered sensor matches the next sensor a train is expected to trigger, the attribution server notifies all tasks that are currently waiting on AttributedSensorAwait(). If the sensor has not been matched to any train, the attribution server attempts to check for “off by 1” sensors (i.e. is the triggered sensor 1 ahead of the expected sensor). This handles dead sensors dynamically, so that the attribution server doesn’t need a hardcoded list of dead sensors. However, if there are 2 dead sensors in a row, then the attribution server will run into issues. If the attribution server still has not matched the sensor to a train and the attribution server is looking for a train, then it will assume that the train it is looking for is at the triggered sensor. If the sensor still has not been attributed to a train, the attribution server will log a warning and then ignore the sensor.

Location Server

The location server tracks each train’s location on the track. A location is specified as a sensor plus an offset from that sensor. The location server creates a notifier that waits on the attribution server. When a sensor is attributed to a train, the location server updates the train’s position and dynamically calculates the train’s velocity by calculating the time since the train last hit a sensor. Additionally, the location server creates a helper task that will send it a message every 30 milliseconds. Therefore, the location server will update its estimated position of each train every 30 milliseconds by using the train's last known position and last known velocity.

The location server exposes a function called LocationAwait(). Tasks in the system that need to perform computations based on a train's location can call this function in order to be sent location updates.

Route Server

The route server uses data from the location server to determine a route from the train's current location to the train's destination. That is, the route server creates a notifier which calls LocationAwait(). This implies that the route server recalculates routes every 30 milliseconds (since that's how often the train locations are updated). The route server then utilizes the received location and the train's destination as the start and end nodes in Dijkstra's algorithm. Since the graph is a directed graph, the route server performs Dijkstra's algorithm in the forward direction and in the reverse direction. The route server then compares the distance of the forward path and the reverse path. However, it first needs to calculate how long it will take the train to perform a reverse. Furthermore, the reverse route pays an additional penalty depending on the train's velocity. It is very costly to reverse a train moving at top speed and not costly at all to reverse a train that has stopped. Additionally, the route server remembers the locations of trains. Let's say for example that train 1 and train 2 are driving on the track. When the route server is calculating a route for train 2, it removes train 1's location from the graph before performing Dijkstra's algorithm. This ensures that the route server does not select a route which will have train 2 collide with train 1 (this algorithm currently has a bug that I was not able to fix in time for the final demo). Finally, the route server exposes the RouteAwait() function for any task interested in utilizing the newly calculated train route.

Stop Server

The stop server is responsible for stopping trains at desired locations. The stop server creates a notifier which calls RouteAwait(). The stop server will calculate the distance from a train's location to the end of its route. At the appropriate time, the stop server will issue a stop command to a train.

Conductor

The conductor is responsible for having a train follow its desired route. Therefore, the conductor creates a notifier which calls RouteAwait(). There are a variety of maneuvers a train may perform in order to follow its route. All of which are executed by the conductor. For example, the conductor will: issue a reverse command if the train's route goes in the opposite direction, speed up a train if the train has a route and is not moving, flip switches just in time to ensure the train follows its route. There's only one slight caveat. Due to uncertainties in a train's position while the train is accelerating or decelerating, the conductor assigns a confidence value in the train’s location. If the conductor is not confident enough in the train’s position, and the train is near a switch, and the train is meant to take the switch, then the conductor will not change the switch to the correct position. This will cause the train to go off its route, but will prevent a switch from being changed while a train is on top of it (which would likely derail the train). However, this is fine since the route server will simply recalculate a new route in 30 milliseconds based off the fact that the train missed the switch.

Safety Server

The safety server listens to various events that are emitted by other system components. It then utilizes this information to determine if the system is in a safe state. For instance, the safety server will utilize information from the attribution server and location server to determine if a train is about to drive off the track. If so, the safety server will issue an emergency stop to the train, as well as logging an error.

Scheduler

The scheduler is responsible for making predictions on when a train will reach the next sensor. To perform this, the scheduler makes several tasks that call either AttributedSensorAwait() or LocationAwait(). After the LocationAwait() event has occurred, the scheduler updates its prediction on when the train will arrive at the next sensor. After the AttributedSensorAwait() event has occurred, the scheduler checks to see how accurate its prediction was. If the prediction was off by more than 50ms, the scheduler logs an error to the screen. In summary, the scheduler has no vital purpose in the operation of the system system, but can be provide invaluable information as when incorrect predictions are continuously made there is probably something wrong elsewhere in the system.

Destination Server

The destination server really just coordinates among the various tasks in the system that need to know the train’s destination (i.e. the route server and the stop server). Other than that, the only thing the destination server does is generate random destinations if it has been told to run a train forever.

Input Parser

The input parser reads continuously from the COM2 port. It waits for data from the COM2 read server, and writes the data to a string buffer. When it encounters a newline (specifically the carriage return character), it parses the string buffer for one of the 6 accepted terminal commands, executes the response if the command matches, and clears the buffer. Backspace is supported by writing over the buffer.

Display Server

When a task wants to display something on the command line terminal, it can send a message to the display task, which will draw the information sent in the message to the appropriate location in the server.

When the display server starts, the display server registers its id with the name server and registers a shutdown hook to clean up the display. It clears the terminal screen and hides the cursor before beginning to draw.

Currently, the display server supports messages to display:

- command-line input
- clock since start of program
- idle percentage
- switch state
- train arrival time at sensor if over a threshold
- all log messages

When Shutdown is called, the shutdown hook sends a message to the display server, which stops the display server from accepting any new replies and clears the screen to avoid leaving artifacts on the screen after the program ends.

Task Priorities

Priorities are separated into System level and User levels to ensure user tasks should not run above system level processes.

System Tasks

Priority (Descending)	Tasks	Justification
HighestSystemPriority	All notifiers	Minimize the likelihood of missing an interrupt.
Priority 30	All couriers	Notifiers utilize couriers. The couriers should be at a lower priority than the notifier, but a higher priority than the recipient.
Priority 29	Name server Clock server COM1 I/O servers	These tasks provide services of similar priority. They should be near the highest priority in the system, to ensure that requests are satisfied in a timely manner.
Priority 28	I/O handle server	Opening handles occurs fairly infrequently and usually only happens during system startup where performance is not critical.
LowestSystemPriority	First System Task Shutdown Task	The First System Task is given a lower priority than any task that it creates. This is to ensure that system tasks are created in a guaranteed and well known order. The shutdown task is given this priority so that it has a higher priority than any user/train tasks to make sure all user tasks receive the shutdown command.
Priority 11	COM2 I/O servers	The COM2 servers have a lower priority as COM2 has such a high baud rate that it can afford to be slightly delayed.
IdlePriority	Idle Task	The Idle task runs at a special lowest priority to ensure it runs only if no other tasks are running.

User Tasks

In general, if a user task runs at priority *n* then it consumes a service from a task that is running at priority *n+1, n+2,* etc. Therefore, the task priorities model the system's dependency graph. This should eliminate any possibility of a circular dependency.

Priority (Descending)	Tasks	Justification
HighestUserPriority	Any notifier	All notifiers (e.g. a task that calls LocationAwait()) need to be at the highest priority to ensure that they never miss an event.
Priority 25	Sensor server	Sensor data is the most valuable data in the system. Therefore, the sensor server is given the highest priority possible to ensure that sensor data is retrieved as soon as possible.
Priority 24	Train Server	Many services utilize the train server, and the train server depends on no other task, so it is run at the next highest priority. This way, it can respond to events, such as stopping the train, in a timely manner.
Priority 23	Switch Server	Many services utilize the switch server, and the switch server depends on no other task. The switch server is run at the next highest priority to ensure that when a switch needs to be switched immediately (i.e. so that it won't cause a train to derail), it will be switched in a timely manner.
Priority 22	Attribution Server	The attribution server consumes data from the sensor server and tries to match sensors to trains. Many other tasks depend on attributed sensors, so the attribution server is run at a high priority.
Priority 21	Location Server	The location server consumes data from the attribution server. It is run at the next highest priority to ensure that attributed sensors are dealt with as soon as possible. This ensures that as accurate as possible location information is produced, which is vital to the operation of

		some other system components (e.g. the stop server).
Priority 20	Route server	The route server consumes data from the location server. The route server is the most CPU intensive task and the system, so I’d like if it could be run at a lower priority. Unfortunately, due to the dependency graph, the route server must be run at the next highest priority.
Priority 19	Stop server	The stop server consumes data from the route server and runs at a high priority in order to produce as accurate of a stop as possible.
Priority 18	Conductor	The conductor consumes data from the route server, but is not quite as important as the stop server (where missing a deadline could mean a train not stopping accurately).
Priority 17	Safety server	The safety server uses information from other services to determine if any catastrophes have occurred in the system. The safety server runs at a slightly lower priority, since responding to a catastrophe 1 millisecond slower probably won’t make a difference.
Priority 16	Scheduler	The scheduler runs whenever there is free time and determines how accurately the system is meeting its predictions. The scheduler plays no part in the actual operation of the system, therefore it is run at a lower priority.
Priority 15	Destination server	The destination server runs at a low priority since a train arriving at its destination occurs infrequently, and generating a random new destination is neither very important nor very CPU intensive.
Priority 10	Display Server	The display server runs at a low priority because updating the display is not too important and COM2 runs very quickly so the system will catch up on displaying data

		when there is nothing else for the system to do.
Priority 9	Input Parser	The Input Parser should respond to keyboard input as soon as possible, but updating the UI is slightly more important.
LowestUserPriority	Performance UI Task Clock UI Task First User Task First Train Task	The Performance and Clock tasks are simple, unimportant polling loops to update the Terminal display, so they should run only if nothing else is running. Initialization tasks are run at the lowest priority to ensure that system initialization occurs in a known, well defined order.

IPC Measurement

4 bytes	cache off	send first	optimization off	768 us
64 bytes	cache off	send first	optimization off	829 us
4 bytes	cache on	send first	optimization off	57 us
64 bytes	cache on	send first	optimization off	61 us
4 bytes	cache off	receive first	optimization off	681 us
64 bytes	cache off	receive first	optimization off	740 us
4 bytes	cache on	receive first	optimization off	49 us
64 bytes	cache on	receive first	optimization off	53 us
4 bytes	cache off	send first	optimization on	358 us
64 bytes	cache off	send first	optimization on	378 us
4 bytes	cache on	send first	optimization on	22 us
64 bytes	cache on	send first	optimization on	24 us
4 bytes	cache off	receive first	optimization on	337 us
64 bytes	cache off	receive first	optimization on	358 us
4 bytes	cache on	receive first	optimization on	20 us

64 bytes	cache on	receive first	optimization on	22 us

We took our best measurement (20us) and kept it the same except for modifying the Send-Receive-Reply system calls to be a Pass() system call. This should give us a pretty good rough estimate of how long the context switch takes (i.e. make an swi, do nothing, run the scheduler, then run the next task). Of the 20us spent for Send-Receive-Reply, we found that 10us are spent doing the context switch. This leaves 10us leftover for performing the actual Send/Receive/Reply, which makes sense since those system calls need to validate input, retrieve data from memory, and copy around memory.