CS 452 Train Control

- Victor Lai
- 20426719
- v6lai@uwaterloo.ca

Path to Executable

On the linux.student.cs.uwaterloo.ca server:

```
/u/cs452/tftp/ARM/v6lai/shared/rtos.elf
```

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

- `cd /u/v6lai/cs452/kernel/`
- `sh cmake_setup.sh`
- `cd release` (or `cd debug`)
- `make`

The executable will now be located in `/u/v6lai/cs452/kernel/release/bin/rtos.elf`

Path to Source Code

The source code can be found at https://git.uwaterloo.ca/v6lai/CS452-Kernel/tree/master

The commit for the deliverable is tagged accordingly.

Tag	SHA
Kernel-1	d6a3212b
Kernel-2	167f7a77
Kernel-3	aedefb1d
Kernel-4.2	dc66978b
Train-Control-1	9db25545
Train-Control-2	8e66b52b

How to Operate the Program

The kernel must be loaded at the default address (0x218000). Once loaded, the kernel can be started with the `go` command.

```
> load b 0x00218000 h 10.15.167.5 "ARM/v6lai/shared/tc1.elf"
> ...
> go
```

Terminal

The program initializes as a mock terminal.

The GUI shows:

- the time since the start of the program
- the percent time that the kernel is idle
- the current state of the train switches
- a one-line command prompt
- a scrolling log message queue

The terminal supports several commands. The 1 and 2 letter commands are primarily used for debugging. The other commands should be used primarily.

- `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)

- `rs <train_no> <sensor_module> <sensor_no>`
    - reserve the sensor for the specified train
    - `train_no`, the number of the train on the track
    - `sensor_module`, the letter name of the sensor (supported A to E)
    - `sensor_no`, the number of the sensor (supported 1 to 16)

- `rl <train_no> <sensor_module> <sensor_no>`
    - release the sensor for the specified train
    - `train_no`, the number of the train on the track
    - `sensor_module`, the letter name of the sensor (supported A to E)
    - `sensor_no`, the number of the sensor (supported 1 to 16)

- `q`
    - quit the program

- `start <train_no>`
    - start the train running at the default speed
    - `train_no`, the number of the train on the track (supported 1 to 80)

- `stop <train_no>`
    - stop the started train immediately, and estimate where the train should end up
    - `train_no`, the number of the train on the track (supported 1 to 80)

- `goto <train_no> <sensor_module> <sensor_no> <distance>`
    - move the started train to stop at a given sensor
    - `sensor_module`, the letter name of the sensor (supported A to E)
    - `sensor_no`, the number of the sensor (supported 1 to 16)
    - `train_no`, the number of the train on the track (supported 1 to 80)
    - `distance`, the distance in cm from the sensor

- `speed <train_no> <train_speed>`
    - set the speed of the started 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)

- `reverse <train_no> <train_speed>`
    - slowly reverse the started train, to the speed of the 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)

- `switch <switch_no> <switch_position>`
    - set the position of a switch and notify all started trains
    - `switch_no`, the switch on the track (supported 1 to 22)
    - `switch_position`, the position to switch to (supported C or S)

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 lg_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 region from 0x00400000 to 0xFFF00000 into 512KB 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 reused.

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
    - The first task has no parent, so its parent id is set to itself
- Task State
    - The current state of the task
    - Can be Ready, Running, SendBlocked, ReceiveBlocked, ReplyBlocked, EventBlocked or Zombie
- Task Priority
    - The priority of the task
    - Higher numbers mean higher priority
- Stack
    - A description of region of memory that has been allocated for this task to allocate objects
- Mailbox
    - A list of 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 system 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 n1 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 registers R4 to 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 `kernel.asm::KernelLeave`. 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 `trap.asm::TrapEnter`. 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 `interrupt.asm::InterruptEnter`. 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

AwaitEvent

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 receive 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. The kernel operates successfully under the assumption that there is always a task to run. When no tasks are available to schedule, the idle task is scheduled instead. If `IdleQuit` is called, the idle task exits, and the kernel will exit safely.

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 name server's task ID is hardcoded to allow other tasks to find the name server. The name server must be one of the first tasks to be initialized.

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 ticks since the start of the program.

DelayUntil

The `DelayUntil` 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. Users 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 queuing 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 methods to send commands to the track to control the train.

The train server starts by registering a shutdown callback when the operating system is stopped. Next, it sends the track `go` command before listening for commands. When the operating system stops, the train server sets the speed of all trains to 0 and sends the track `stop` command 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. Additionally, the train server will let the location server know the target speed of the train.

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 methods to get and set the direction of a switch in the track.

When the switch server starts, the server initializes its internal switch state list by setting all switches to the curved direction.

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.

SwitchGetDirection

The `SwitchGetDirection` function sends a message to the switch server validating and passing the desired switch. The switch server responds with the switch direction stored in its internal switch state list.

Sensor Reader

The sensor reader reads sensor data continuously from the `COM1` port.

When the sensor reader starts, it waits 10 seconds and flushes the `COM1` input buffer to get rid of delayed sensor data from the track.

The sensor reader continuously reads 10 bytes in a row. It compares the new 10 bytes with the previous 10 bytes, and calculates which sensors have changed. If sensors have changed to the on state, then they are sent to the sensor server.

Sensor Server

The sensor server allows multiple tasks to await for changes in sensor states.

The sensor server provides a `SensorAwait` function semantically similar to `AwaitEvent`. A task calling `SensorAwait` will block until a sensor state changed is detected, and the task will receive the list of changed sensors.

Track Server

The track server is used to represent the current global state of the track. It provides send wrapper methods to get specific track nodes.

The track server relies on the state of switches from the switch server to determine the state of the track.

- GetSensorNode
    - given a sensor, get the associated sensor node

- GetDistanceBetweenNodes
    - given two nodes, get the distance between the two nodes if it is possible
- GetNextSensorNode
    - given a node, get the next expected sensor node
- GetPathToDestination
    - given two nodes, get the path of nodes between the two nodes if it is possible
- GetNextNodesWithinDistance
    - given a node and a distance, get all the expected nodes within the distance of the node

Track Reserver

The track reserver is used to reserve portions of the track for a train.

A train can reserve either a single track node or a list of track nodes, and release either a single node or all of its nodes. If a track node is already reserved by another train, the reservation will fail.

Each reservation will also reserve the node in the opposite direction.

Location Server

The location server is responsible for keeping track of the position of the train.

It combines the following information to determine the position of the train:

- per sensor
    - sensor notifications and distances
    - sensor attribution and track switch state
    - sensor arrival delay
- per tick
    - pre-calculated train velocity, acceleration and deceleration model

Sensor Attribution and Track Switch State

While a train is running, the only source of data available are sensor notifications. These notifications can be semi-accurately attributed to trains given the expected location of all the trains.

When a train is initialized, it is put on a lost trains queue. If a sensor notification is received that does not match the expected location of any other train, then the sensor is attributed to this lost train.

As the train continues around the track, it will continue to trigger other sensors. Because the location server knows the sensor that the train last triggered, it can walk through track model using the track switch states to determine if a triggered sensor implies that the train has reached the sensor or not.

The location server will walk through the next 3 expected sensor nodes for each train. This ensures that a train can miss triggering at least 2 sensor nodes before the location server loses track of it. If none of the next 3 sensor nodes match, the location server will check for any unexpected branches taken. This accounts for potentially failed switch states.

Sensor Attribution and Distances

After attributing each sensor trigger to a train, a series of consecutive sensor triggers can provide additional timing information. Particularly, by knowing the time of the last and current sensor triggered by a train, as well as the distance between the sensors, the velocity of the train can be calculated.

This velocity is used to determine the train’s true velocity at the time.

Sensor Arrival Delay

To correct for processing overhead in the operating system, each sensor node is given a delay value in ticks whenever the train passes over the sensor node. This processing overhead is calculated as the difference in expected and actual arrival times at the node.

When another calculation for the expected arrival time is performed, this processing overhead is included to that time. After several passes over the sensor, the sensor overhead converges to a more accurate number, ensuring that the location server can accurately anticipate the time that a sensor will be triggered.

Train Calibration

Every train has different motion characteristics. Therefore, to more accurately track the position of the train, a series of tests were used to determine the acceleration, steady-state velocity and deceleration constants for each train.

These constants are used to update the per-tick train location model. The location server will update the position of each train every 20ms by adding the velocity multiplied by the elapsed time, and update the velocity by using the acceleration and deceleration to update the velocity. This ensures that the train location is up-to-date even without continually sensor triggers.

Conductor

The conductor is responsible for sending commands to the track. It is used to de-couple communication with the track with the scheduling of the trains.

The conductor has a queue of workers available to send the `TrainSetSpeed`, `TrainReverse` and `SwitchSetDirection` commands to the train, and update the location server when these commands are invoked.

Scheduler

The scheduler is responsible for coordinating the train's movement along the track.

Its primary responsibilities are:

- directing trains to their destinations
- stopping the train at its destination
- ensure trains do not collide

Moving to Destination

Moving trains to their destination is done optimistically. At each sensor node, breath-first search is used to determine the route of the train to the destination. When the route is determined, the train will attempt to set the direction of all switches required to reach the destination. This will happen at every sensor node until the train reaches its destination.

If a train cannot set the direction of a switch because the switch is reserved, it will simply continue along its path and attempt to set the direction again at the next sensor node. Eventually, the train will either set the switch to the correct direction, or fail to set the switch and pass along the other path. If this occurs, then the path to the destination will simply be recalculated at the next sensor node.

The optimistic routing cleanly de-couples moving the train to its destination from collision avoidance. Re-calculating the path at every sensor node also ensures that the algorithm is robust against failure. If the current path is faulty, the path calculated at the next node may be successful and return the train to a correct destination.

Path-Finding

Path-finding is done using a breadth-first search algorithm starting at the initial node. Only the forward path from the current node is visited, so there are no opportunities to reverse around a merge node, and starting a path into an exit node will fail.

At every step of the algorithm, every node is visited and its parent visiting node is set. If the destination node is found, the algorithm returns the path from that parent node to the initial node.

Breadth-first search is a simple algorithm that finds a path that traverses through the last number of nodes. This is essentially similar to an algorithm using Dijkstra's due to the small size of the graph and similar distances, but computationally less complex. The simpler path calculation fits with the more optimistic route planning model better due to the lessened emphasis on time.

Stopping

At every tick, the scheduler will re-calculate the expected stopping distance for the train. If it knows the train's destination, it can determine the distance between the train and the stopping distance. If this distance is too low, then it knows that the train should begin stopping now to ensure it does not overshoot its destination.

Collision Avoidance

Collision avoidance is simply done by reserving all the anticipated nodes within the train's stopping distance. At every tick, the stopping distance is calculated, and the train reserves all the nodes ahead of it.

If the train fails to reserve a node, it will stop and reverse in direction.

Deadlock Avoidance

Through the combination of optimistic path-finding and train reversals, deadlocks can be mostly avoided unless there is a situation where the track is a circle and no train can make progress to reach a sensor.

I/O

Input and output is implemented in a separate module and exposed in a public header.

Input Parser

The input parser reads continuously from the `COM2` port. It reads a singles a single character into a buffer, and display the character in the command-line input. If it encounters a backspace, it will clear a character from a buffer. When it reads a carriage return from a newline, it will parse and clear the buffer.

During the parsing, it will look for one of the commands in the format specified in the Terminal section of this document, and try to execute it if it matches.

Display Server

The display server continually listens to display commands from other processes. Tasks that want to display a specific request can call one of the supported methods which draw to a specific location on the GUI, or log any arbitrary data.

Currently, the display server supports:

- command-line input
- clock
- idle percentage
- switch state
- train arrival status
- train location status

The display server registers a shutdown hook which will clear 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.

- there are a lot of things I would like to change about the priorities now that they're written out like this …

System Tasks

Priority	**Task**	**Comments**
HighestSystemPriority	All Notifiers	Minimizes the likelihood of missing an interrupt
Priority30	All Couriers	Run at a lower priority than the notifier, but a higher priority than all recipients
Priority29	Name Server	Ensure that name server requests are handled quickly
Priority29	Clock Server	Ensure that the clock is updated quickly
Priority29	COM1 Read Server	Ensure that COM1 communication occurs as soon as possible
Priority29	COM1 Write Server	Ensure that COM1 communication occurs as soon as possible
Priority28	I/O Handler Server	I/O Register and Open occur infrequently, and usually during start-up where performance is not critical

Priority	**Task**	**Comments**
LowestSystemPriority	Shutdown Server	Ensures that all user tasks will receive the shutdown callback
LowestSystemPriority	First System Task	Allow all system tasks to be created and initialized in order
Priority11	COM1 Read Server	Allow `COM2` communication to be delayed
Priority11	COM1 Write Server	Allow `COM2` communication to be delayed
IdlePriority	Idle Task	Runs at a special lowest priority so it only runs if no other tasks are running

User Tasks

In general, the priorities are ordered to prioritize the scheduling of the trains. This involves keeping the scheduler and location server at a higher priority. Non-essential tasks for train operation such as the display and idle percentage are run at the lowest priorities. Intermediate tasks such as the conductor, train and sensor servers are run at intermediate priorities.

Priority	**Task**	**Comments**
HighestUserPriority	Sensor Reader	Receive sensor updates quickly to ensure accuracy
Priority24	Scheduler	Control trains as soon as possible to avoid collisions
Priority24	Track Server	Unblock tasks waiting for track information as soon as possible
Priority23	Location Sensor Notifier	Receive sensor updates quickly
Priority23	Location Server	Update the location of the train as much as possible
Priority22	Location Tick Notifier	Only tick if there is no scheduling or location update required
Priority20	Train Server	Prioritize sending commands to the train
Priority19	Scheduler Update Notifier	
Priority19	Switch Server	
Priority19	Reservation Server	

Priority	Task	Comments
Priority18	Sensor Server	
Priority17	Conductor Server	
Priority14	Conductor Worker	
Priority13	Location Server Worker	
Priority10	Display Server	Update the UI sparingly
Priority9	Input Parser	Command-line input will occur very slowly
LowestUserPriority	Performance Task	Showing the idle percentage on the GUI is not time-sensitive
LowestUserPriority	Clock Task	Showing the clock on the GUI is not time-sensitive
LowestUserPriority	First User Task	
LowestUserPriority	First Apps Task	
LowestUserPriority	First Io Task	
LowestUserPriority	First Train Task	

Folder Structure

- **ext**
    - **bwio**
        - bwio.c
        - bwio.h
        - CMakeLists.txt
    - **rtosc**
        - assert.c
        - assert.h
        - bitset.h
        - buffer.c
        - buffer.h
        - CMakeLists.txt
        - linked_list.c
        - linked_list.h
        - math.c
        - math.h
        - priority_queue.c
        - priority_queue.h
        - stdlib.c
        - stdlib.h
        - string.c
        - string.h
    - **track**
        - track_data.c
        - track_data.h
        - track_lib.c
        - track_lib.h
        - track_node.h
        - CMakeLists.txt
- **inc**
    - **rt**
        - defs.h
        - limits.h
        - params.h
        - pointers.h
        - rt.h
        - rtstatus.h

        - types.h
        - vararg.h
    - **rtos**
        - **user**
            - apps.h
            - io.h
            - trains.h
            - users.h
        - rtkernel.h
        - rtos.h
        - ts7200.h
- **src**
    - CMakeLists.txt
    - orex.ld
    - **kernel**
        - arm.asm
        - arm.h
        - cache.asm
        - cache.h
        - CMakeLists.txt
        - interrupt.asm
        - interrupt.c
        - interrupt.h
        - ipc.c
        - ipc.h
        - kernel.asm
        - kernel.c
        - kernel.h
        - main.c
        - performance.c
        - performance.h
        - scheduler.c
        - scheduler.h
        - stack.c
        - stack.h
        - syscall.c
        - syscall.h
        - task.c
        - task.h
        - task_descriptor.c
        - task_descriptor.h

        - trap.asm
        - trap.h

    - **os**
        - clock_server.c
        - clock_server.h
        - CMakeLists.txt
        - courier.c
        - courier.h
        - idle.c
        - idle.h
        - init.c
        - io.h
        - io_lib.c
        - io_read.c
        - io_server.c
        - io_write.c
        - name_server.c
        - name_server.h
        - rtos.asm
        - shutdown.c
        - shutdown.h
        - uart.c
        - uart.h

    - **user**
        - CMakeLists.txt
        - **apps**
            - calibration.c
            - calibration.h
            - CMakeLists.txt
            - final_demo.c
            - final_demo.h
            - init.c

        - **io**
            - clock.c
            - clock.h
            - CMakeLists.txt
            - display.c
            - display.h
            - init.c
            - input_parser.c
            - input_parser.h

            - performance.c
            - performance.h
        - **trains**
            - CMakeLists.txt
            - conductor.c
            - conductor.h
            - init.c
            - location_server.c
            - location_server.h
            - physics.c
            - physics.h
            - scheduler.c
            - scheduler.h
            - sensor_server.c
            - sensor_server.h
            - switch_server.c
            - switch_server.h
            - train_reserver.c
            - train_reserver.h
            - train_server.c
            - train_server.h
        - **users**
            - CMakeLists.txt
            - init.c
- **tests**
    - CMakeLists.txt
    - test_buffer_main.c
    - test_linked_list_main.c
    - test_priority_queue_main.c
    - test_scheduler_main.c
    - test_string_main.c
    - test_task_descriptor_main.c
    - test_track_data_main.c
- CMakeLists.txt
- cmake_setup.sh
- README.md