Kernel 4
CS452 - Spring 2014
Real-Time Programming

Team
Max Chen - mqchen
mqchen@uwaterloo.ca

Ford Peprah - hkpeprah
ford.peprah@uwaterloo.ca

Bill Cowan
University of Waterloo
Due Date: Monday, 23^{rd}, June, 2014

Table of Contents

1 Program Description

1.1 Getting the Program

To run the program, one must have read/write access to the source code, as well as the ability to make and run the program. Before attempting to run the pogram ensure that the following three conditions are met:

- You are currently logged in as one of `cs452`, `mqchen`, or `hkpeprah`.
- You have a directory in which to store the source code, e.g. `~/cs452_microkern_mqchen_hkpeprah`.
- You have a folder on the FTP server with your username, e.g. `/u/cs452/tftp/ARM/cs452`.

First, you must get a copy of the code. To to this, log into one of the aforementioned accounts and change directories to the directory you created above (using `cd`), then run one of

```
git clone file:////u8/hkpeprah/cs452-microkern -b kernel4 .
                        or
git clone file:////u7/mqchen/cs452/cs452-microkern -b kernel4 .
```

You will now have a working instance of our kernel4 source code in your current directory. To make the application and upload it to the FTP server at the location listed above (`/u/cs452/tftp/ARM/YOUR_USERNAME`), run `make upload`.

1.2 Running the Program

To run the application, you need to load it into the RedBoot terminal. Ensure you've followed the steps listed above in the "Getting the Program" settings to ensure you have the correct directories and account set up. Navigate to the directory in which you cloned the source code and run `make upload`. The uploaded code should now be located at

`/u/cs452/tftp/ARM/YOUR_USERNAME/assn4.elf`

To run the application, go to the RedBoot terminal and run the command

```
load -b 0x00218000 -h 10.15.167.4 ‘‘ARM/YOUR_USERNAME/assn4.elf’’; go
```

The application should now begin by running through the game tasks before reaching a prompt. The generated files will be located in `DIR/build` where `DIR` is the directory you created in the earlier steps. To access and download an existing version of the code, those can be found at `/u/cs452/tftp/ARM/mqchen/assn4.elf` and `/u/cs452/tftp/ARM/hkpeprah/assn4.elf`.

1.2.1 Bootup Sequence

Before the program can actually begin accepting input from the user, it has to put the kernel into a state that is ready to begin working. To this end, the boot sequence of the application does the following sequentially

1. Turns on the cache.
2. Initializes UART2.
3. Initializes UART1.
4. Initializes the debugging and display interface.
5. Initializes the memory.
6. Initializes the task handling.
7. Sets up the SWI handler.
8. Initialzies the logging system.
9. Turns on the train set.
10. Writes the display to the screen.
11. Seeds the pseudo random number generator.
12. Enables interrupts.

These are done using busy-wait before the application has been running any of the user tasks and moves the kernel into a state ready to begin running tasks.

1.2.2 Shutdown Sequence

The application does not terminate until the user issues a shutdown command ('q' on the shell). The purpose of the shutdown operations is to leave the kernel into a clean state such that another application can be reloaded without having to issue a hardware reboot to the box. To shutdown, the application busy-waits performing the following tasks sequentially

1. Disable interrupts.
2. Turns off the train set.
3. Clears any remaining tasks.
4. Disablees the idle timer.
5. Dumps the log.

1.2.3 Command Prompt

After the startup tasks have finished running, the user will reach a command prompt where they will be able to enter commands. A list of available commands the syntax are listed below in the `Shell` section.

2 Kernel Structure

2.1 Memory Allocation

Memory allocations is created in a trivial manner. As in accordance with maintaing the performance of a real time system, we make use of the stack to allocate memory, treating Memory as a linked list of addresses which represent a segment (or block) of the stack. These segments make up the stacks of the tasks. On allocation, the head of the linked list is returned and the next segment becomes the head, while on free, the block is added as the head of the Memory linked list. Since Memory protection and segmentation are not required, memory was implemented in the simplest manner.

2.2 Task Descriptor (TD)

Tasks represent a thread or unit of execution. A task descriptor describes a single task and stores the following information:

Field	Description
state	the task state, ie. READY, ACTIVE, or blocked on some events
tid	the task identifier (these aren't reusued)
parentTid	tid of the parent task (task that created thsi one)
priority	integer value indicating priority of task (larger number $\rightarrow$ higher priority)
sp	the task's stack pointer
next	next task in the task queue this task belongs to
waitQueue	waits until tasks on this queue exit before this task is run (not in spec)
addrspace	pointer to the block of memory the task can use as it's stack
inboxHead	pointer to head of inbound message queue
inboxTail	pointer to tail of inbound message queue
outbox	pointer to outbound message (reply location) OR inbound message location (Receive-Before-Send special case)

The CPSR of the task and the LR for the kernel to return to are stored on the stack instead of the TD.

To create the task descriptors, since we do not make use of the heap, we have to allocate our task descriptors from a bank that already exists on the stack. To this end, we have an array of 32 task descriptors (an amount that will be tweaked depending on how much is needed in the future), that can be used to create new tasks. These tasks start as blank with their state marked as `FREE` denoting that they can be used to allocate new tasks. When a task is `READY`, it is stored in the `taskQueue` corresponding to its priority, of which priorities can range from $0 - 15$. To get constant performance when creating a new task descriptor, in the `initTasks` function, they are assigned to the `taskBank` as a linked list, each task descriptor pointing to the next free task descriptor in the list. When we wish to create a new task, we grab the task descriptor at the head of the bank as it will always be free to use. We then assign it a priority, task id, parent tid (if the parent exists), an address space to use by calling `getMem()` which returns a segment of memory for the task to use as its stack, and the task pointer which points to the bottom of the address space. The task is then added to the end of its respective queue. On deletion, we mark the state of the task as `ZOMBIE` and free its address space; for now we make use of the `ZOMBIE` state and do not add the task descriptor back to the head of the bank to adhere with assignment specifications, but in the future to allow for immediate garbage collection that would not hinder the performance of the system, the task would be added as a free task to the head of the bank.

2.3 Task Queues

Queues for tasks are implemented with two pointers; one pointer to the head of the queue and another pointer to the end of the queue. We use the tail pointer to enable constant time addition of a new task to the queue by simply having the tail task pointer to the added task as it's `next` field. The `head` pointer enables us to get the next task in the queue that is to be run and pop it off the queue. Queues are created from an array of queues on the stack, and each queue's index in the array corresponds to the priority of the tasks that are stored in that queue.

2.4 Scheduling

On a `schedule()` call, the following sequence of events takes place:

1. If the queues are empty, or there are no tasks of higher priority, return the current task (i.e. the last task that was running).
2. If the current task is null, the kernel has no more tasks to run, exit.
3. Otherwise, if the current task is in the ACTIVE (ie. not blocked) state, add the current task to the end of the priority queue.
4. Get the next task descriptor from the priority queue.
5. Move that task into the `ACTIVE` state and return it.

This sort of round-robin scheduling means that each task in the queue has an equal opportunity at being run and means that if a task passes and it is the only task in the queue, it will simply be run again. There are 16 priorities $(0 - 15)$ for tasks, with 15 being the highest priority a task can have. Each priority has its own queue of task descriptors that are implemented as a linked list and tracked by

1. `highestPriorityQueue` - integer, the highest priority queue that is not empty
2. `availableQueues` - integer, bit field for the queues, 1 = non-empty, 0 = empty

These two variables are updated each time a call is made to `addTask` to add a new/existing task descriptor to the end of a queue, which occurs either during creation or scheduling.

Tracking the queue state allows for a constant time retrieval of the next task to run. When a `schedule()` call results in the last task being removed from the queue of its priority, then the corresponding bit in `availableQueues` is set to 0, and `highestTaskPriority` is updated to the next highest task priority by doing a pseudo-linear search of the bits in `availableQueues` to find the first occurence of a 1 bit. A binary search is run, with the search space between

the last `highestTaskPriority` and 0. However, this binary search is inconclusive, in that it will only move the high index towards 0, and will break when doing so will make the high index lower than the next highest priority. Essentially, we want to find the first n such that n > next highest priority, but $\frac{n}{2}$ <= next highest priority. From here, a linear search is done counting down from n until the first non-zero bit of `availableQueues` is found.

This was done to give an optimization in the worst case (last priority was very high, and the next highest priority is low). A full binary search is not done since given the size of the search space, there doesn't seem to be much to be gained from doing so. Optimization seems premature at this time, so there was not much effort spent on it.

2.5 Context Switch

From both software and hardware interrupts, the same entry point is eventually reached in the kernel. The kernel then handles the request based on the request code and then switches back out to the next scheduled user task in the same way. For kernel SWI calls, the wrapper function will construct an `Args_t` struct on its stack and pass its address to the kernel. For hardware interrupts, a default `Args_t` struct is used with the code being INTERRUPT and no arguments.
On a context switch, all registers r0-r12, lr are pushed to the user stack, as well as the CPSR and LR. Context switch is accomplished with:

1. Switch to SYS mode with interrupts disabled (write 0x9F to CPSR).
2. Save r0-r12, lr to the user stack, move the SP to r0.
3. Switch back to the service mode (SWI/IRQ).
4. Push SPSR, LR of the respective service mode to r0 (user stack).

IRQ Only: The IRQ handler subtracts 4 from the LR when storing it to the user stack (interrupted instruction instead of next instruction).
SWI Only: The SWI handler pops the top of the kernel stack as an address and writes r1 into it. This is the method of passing the `Args_t` of the sys call to the kernel, as described above.
After the user state is saved, the processor is returned to the kernel by popping off the stored kernel registers from its stack. Since the kernel starts in SVC mode, the IRQ handler must first switch to SVC mode before making the pop from the stack. Both handlers will enter the kernel in the exact same way where it exited through the `swi_exit` call, returning the task SP (r0).
The kernel then handles the request, schedules the next task, and calls the assembly function `swi_exit` passing in the task's SP and an output pointer parameter for the next `Args_t` which does a context switch back to the user task.

2.6 Send-Receive-Reply

The `Envelope_t` struct is used to handle message passing, containing the following information:

Field	Description
void *msg	Source address to copy message from
int msglen	Length of msg
void *reply	Destination address to copy reply to
int replylen	Length of reply
Task_t *sender	Pointer to TD of the Sender task
Envelope_t *next	Next envelope in the inbox - these are in a linked list

A pool of `Envelope_t` structures are allocated for message passing, and are retrieved/freed with a simple linked list of free structs.
The states `SEND_BL`, `RECV_BL` and `REPL_BL` corresponding to the possible blocked states.
These structures are stored in the `inboxHead`, `inboxTail` and `outbox` parameters of the task descriptors.

2.6.1 Send

If the receiver is in the `SEND_BL` state, it is added back to the ready queues, and the send value is directly copied into the receiver's provided buffer space at its `outbox` parameter. The same envelope is then added as the sender outbox so it can be replied to.
Otherwise, the Receiver has yet to block. In this case, an `Envelope_t` is retrieved from the available pool, and if none are available, an error is returned (A more elegant solution could be implemented by blocking the task until one of structs is available). The msg, msglen, reply, and replylen parameters from the `Send` call are copied into the respective fields of the `Envelope_t`. In addition, the current task pointer is added to the envelope as the sender, and the envelope is set as the outbox pointer of the sender. The envelope is then added to the tail of the receiver's inbound message queue, inboxTail.
In both cases the sender is moved to the `RECV_BL` state.

2.6.2 Receive

On a Receive, the inbox of the current task is checked. If there is no inbound messages, the task is moved to the `SEND_BL` state, and an envelope is allocated to store the Receive parameters. This is stored as the outbox since a task can't be blocked on send and receive at the same time, and saves one field in the TD.
If there is a message available, then the corresponding values in the envelope are copied into the provided pointers, and the sender task is moved to the `REPL_BL` state. The envelope is then removed from the head of the queue so the next message can be received.

2.6.3 Reply

For Reply, the intended sender task's outbox parameter is used to find the envelope, and the provided reply message is copied into the provided pointer. The sender is then added back to the ready queue and the envelope is released back into the envelope pool.

3 System Calls/Primitives

System Call	Prototype	Description
Create	`int Create(int, void(*)())`	Creates a task with the specified priority to run the given code.
MyTid	`int MyTid()`	Returns the task ID of the calling task.
MyParentTid	`int MyParentTid()`	Returns the task ID Of the parent task of the calling task.
Pass	`void Pass()`	Calling task gives up control to another task of the same priority.
Exit	`void Exit()`	Calling task exists.
WhoIs	`int WhoIs(char*)`	Queries nameserver for task ID of task with the given name.
RegisterAs	`int RegisterAs(char*)`	Registers the given name to the TID of the calling task with the nameserver.
UnRegister	`int UnRegister(char*)`	Unregister the current task iff its name and tid match what is in the NameServer.
Send	`int Send(int, void*, int, void*, int)`	Sends a message to the specified task. Blocks on Reply.
Receive	`int Receive(int*, void*, int)`	Calling task blocks until it receives a message.
Reply	`int Reply(int, void*, int)`	Replies to the specified task with the given message.
Log	`int Log(const char*, ...)`	Logs a variable length formatted message to the logger.
Logn	`int Logn(const char*, n)`	Logs a fixed length message to logger.
AwaitEvent	`int AwaitEvent(int eventType, void *buf, int buflen)`	Blocks until the event identified occurs then returns.
WaitTid	`int WaitTid(unsigned int tid)`	Waits until the task specified by the `tid` exits, then returns.
Delay	`int Delay(int)`	Blocks the current task until the specified number of ticks have elapsed.

DelayUntil	`int DelayUntil(int)`	Blocks the current task until the specified time has been reached.
Time	`int Time()`	Returns the current number of ticks that have elapsed.
Getc	`int Getc(int)`	Returns first unreturned character from the given UART.
Putc	`int Putc(int, char)`	Transmits the given character to the given UART.
WaitOnSensor	`int WaitOnSensor(char, unsigned int)`	Blocks the calling task until the specified sensor triggers.
CpuIdle	`int CpuIdle()`	Returns the percentage of time CPU was idle.
SigTerm	`void SigTerm()`	Signals to halt the system (kill the NullTask).

3.0.4 Create

`Create` creates a new Task Descriptor with the given priority and code, giving it a stack and assigning it an ID. The created tasks has all the satate needed to run and is added to the priority queues so that it can run the next time it is scheduled. Returns:

- `tid` - Positive integer id of the newly created task, unique.
- -1 - if the priority is invalid.
- -2 - if the kernel is out of task descriptors.

3.0.5 MyTid

Returns the task id of the calling task.

3.0.6 MyParentTid

Returns the task id of the task that created the calling task.

- `tid` - Positive integer id of the parent task.
- 0 - If task was created by kernel / has no parent.

3.0.7 Pass

Moves the calling task from being active back onto the priority queue in a ready-to-run-state. Has no return value.

3.0.8 Exit

Ceases execution of the current task and eliminates it. Has no return values.

3.0.9 WhoIs

Lookup the task with the given name in the NameServer and returns its tid if found. Does not block waiting for registration. Returns

- `tid` - Non-negative task identifier on success.
- -1 - NameServer hasn't been created.
- -2 - Error in send.

3.0.10 RegisterAs

Register the current task with the name `name` in the NameServer. Returns

- 0 - Successfully registered.
- -1 - NameServer hasn't been created.
- -2 - Error in send.

3.0.11 UnRegister

Unregister the calling task iff its task ID and name exist in the NameServer. Does not remove the found task if the tid found at the hash location does not match the tid of the caller. Returns

- 0 - Successfully unregistered.
- -1 - NameServer hasn't been created.
- -2 - Error in send.

3.0.12 Send

On a Send, an `Envelope_t` is retrieved from the available pool, and if none are available, an error is returned (A more elegant solution could be implemented by blocking the task until one of structs is available). The msg, msglen, reply, and replylen parameters from the `Send` call are copied into the respective fields of the `Envelope_t`. In addition, the current task pointer is added to the envelope as the sender, and the envelope is set as the outbox pointer of the sender.
The sender is moved to the `RECV_BL` state and the envelope is added to the tail of the receiver's inbound message queue, inboxTail.
If the receiver is in the `SEND_BL` state, it is added back to the ready queues.

- `size` - Non-negative integer representing the number of bytes copied.
- -1 - task ID is not possible
- -2 - task ID does not correspond to a valid task
- -3 - transaction incomplete

3.0.13 Receive

On a Receive, the inbox of the current task is checked. If there is no inbound messages, the task is moved to the `SEND_BL` state, and nothing else is done. In this case, the task will be unblocked by Send when a message is actually available, and the user task will have to make another call to Receive to get the message. This process is transparent to the user and is done through the user-mode Receive function.
If there is a message available, then the corresponding values in the envelope are copied into the provided pointers, and the sender task is moved to the `REPL_BL` state. The envelope is then removed from the head of the queue so the next message can be received.

- `size` - Non-negative integer representing the number of bytes copied.

3.0.14 Reply

For Reply, the intended sender task's outbox parameter is used to find the envelope, and the provided reply message is copied into the provided pointer. The sender is then added back to the ready queue and the envelope is released back into the envelope pool.

- `size` - Non-negative integer representing the number of bytes copied.
- -1 - task ID is not possible
- -2 - task ID does not correspond to a valid task
- -3 - target task is not reply blocked.

3.0.15 Log

Allocates a block of memory from the current memory space, and copies the passed string to that location in memory.

- 0 - Successful write
- -1 - Out of memory from where to write.

3.0.16 Logn

Allocates a block of memory from the current memory space, and copies a fixed sized string to that location in memory.

- 0 - Successful write
- -1 - Out of memory from where to write.

3.0.17 AwaitEvent

`AwaitEvent` blocks until the event identified by the passed integer, `eventType`, occurs as an interrupt then returns with the value generated by the interrupt. The value is non-zero. In the event that the passed integer is not a valid event, it returns -1 or if the queues are full it returns -2. Since we do not use event buffers, the previous correspondence for 0, -2 and -3 are irrelevant to our implementation.

3.0.18 WaitTid

`WaitTid` blocks on the wait queue of the specified task and returns when that task exists with the status of the exit. Returns -1 if the task does not exist.

3.0.19 Delay

Send a message to the ClockServer to block the curren task until the number of ticks have passed.

3.0.20 DelayUntil

Send a message to the ClockServer to block the current task until the given number of ticks have been reached.

3.0.21 Time

Returns the curren tick count by querying the ClockServer.

3.0.22 Getc

Returns the first unreturned character from the given UART; a wrapper for send to the serial IO server. Blocks until it receives a character.

- `character` - On success returns the read character.
- -1 - Serial IO server does not exist
- -2 - Error in send

3.0.23 Putc

Queues the given cahracter for trasmission to the specified UART; character may not have been transmitted on return. Is a wrapper to the IO serial server.

- 0 - On success
- -1 - Serial IO server does not exist
- -2 - Error in send

3.0.24 WaitOnSensor

Blocks the calling task until the specified sensor has been triggered. Parameters are passed as `(module, index)` where index is a positive integer corresponding to the identifier within that module. Returns

- 0 - On success and unblocked.
- -1 - If the given sensor does not exist.

3.0.25 CpuIdle

Returns the amount of time that the CPU has been idle (running the null task).

- `idle` - Non-negative integer corresponding to the time the CPU has been idle as a fraction of the total CPU time.

3.0.26 SigTerm

Tells the Kernel to kill the NullTask and cease execution. Has no return value.

4 Algorithms and Data Structures

4.1 HashMap/HashTable

Our HashTable implemented has three attributes:

1. `size` - The size of the array in the HashTable.
2. `data` - An array of stored integers that have been hashed to.
3. `assigned` - An array indicating whether an index in the array is occupied or not (1 for assigned, 0 unassigned).

And the following functions:

1. `init_ht(HashTable*)` - Initialize an hashtable by zero'ing out the assigned bits.
2. `insert_ht(HashTable*, char*, int)` - Insert the integer at the hashed index of the char*.
3. `exists_ht(HashTable*, char*)` - Returns 0 if the hashed index is unassigned, otherwise 1.
4. `lookup_ht(HashTable*, char*)` - Returns the value stored at the hashed index. Will return 0 if the hashed index is unassigned, thus a user should check `exists_ht` before calling for a lookup.
5. `delete_ht(HashTable*, char*)` - Delete what is pointed to by the hashed index. Does nothing if it is unassigned.

4.2 Randomization

To implement randomization, we used the Mersenne Twister algorithm for our psuedorandom number generator, as it is a common generator for random numbers, and will guarantee us a deterministic output provided we know the current index in the twister array. Random numbers can be generated by calling either `random()` or `random_range(lower_bound, upper_bound)` from either the user or kernel structure.

1. Populates array with initial values.
2. Randomizes all items in the array.
3. Generates and returns random value at index.
4. Increments to next index.

5 Nameserver

NameServer uses a hash map as the data structure to store the tids in; using the names passed by the registering tasks as the keys to hash with. It acts as a global lookup table for tasks to find other tasks. The tid of NameServer is determined at run time by it assigning a value to a global variable within the file and externed through the header files. It implements three primitives: `RegisterAs`, `UnRegister` and `WhoIs` that have been listed above.

5.1 NameServer Lookup

To implement lookup to allow the NameServer to register and identify tasks by name, a HashMap (mentioned in the section above) was used. To add items to the hash map we needed to convert the string names into integers. To do this, we used the `djb2` hashing algorithm. We ignore collisions as this is a closed space in which we can ensure there will never be a collision.

1. String is hashed, which is bounded by $O(n)$ where n is the length of the string.
2. Lookup in the hash table is $O(1)$ at this point, by accessing the element in the array at the index generated by the hash.
3. Puts are $O(1)$ by the same methodology.

6 ClockServer

6.1 Implementation

The ClockServer registers with the NameServer then creates the ClockNotifier which handles processing of timer generated interrupts. We use the $508kHz$ 32-bit timer. It then writes

5080 to the timer load register; using the $508kHz$ timer and with 10 milliseconds, this corresponds to 5080 in the timer load register. It then writes to the timer control register to enable it, to set the mode to periodic so that the count resumes back at 5080 after counting down, and to set the timer to the $508kHz$ timer. The ClockServer then blocks on `Receive` to handle messages from other tasks. When the ClockServer recieves a message of type `Delay` or `DelayUntil`, it adds the task to its delay queue sorted by the number of ticks that the task is waiting; the array is sorted in ascending order. When it receives a message of type `Tick` it increments its internal counter, replies to the sending task immediately, then iterates through its delay queue waking up every task with a delay less than the current count by replying to them, and stopping as soon as it reaches a task with a delay greater than its current tick count. This ensures that we do not needlessly check tasks that will not wake up as the queue is sorted.

6.2 ClockNotifier

The ClockNotifier is responsible for notifying the ClockServer when a tick occurs; a tick is defined as ten milliseconds passing on the 32-bit hardware timer. Since the $508kHz$ clock is used, ten milliseconds is equivalent to setting the value of the clock to $508 \cdot 10 = 5080$ upon which it will countdown to 0 then generate an interrupt. The ClockNotifier blocks on the timer interrupt and on return it sends a message to the ClockServer indicating a tick took place, which the ClockServer immediately replies to, and then the ClockNotifier blocks on AwaitEvent again. This task never exits unless the system is shutting down.

6.3 API

Implements three primitives: `Tick`, `Delay`, and `DelayUntil` that have been described above.

7 Serial I/O

7.1 Interrupts

The following events are used by `AwaitEvent` for I/O:

1. UART1_XMT_INTERRUPT
2. UART1_RCV_INTERRUPT

3. UART1_MOD_INTERRUPT
4. UART2_XMT_INTERRUPT
5. UART2_RCV_INTERRUPT

These interrupts correspond to the OR'd interrupt values, bit 20/22 in VIC2 for UART1/UART2 respectively. Upon receiving an interrupt, the handler will read the corresponding UART status register to determine whether a XMT or RCV (or MOD) interrupt has actually occured, and unblock the corresponding waiting handler task.

7.2 Kernel Space Interrupt Handler

UART interrupts are dynamically enabled and disabled. When a task calls `AwaitEvent` and is blocked, the interrupt it is waiting on will be enabled by ORing the corresponding CTLR register with the enable mask. This ensures that, assuming no external corruption of the CTLR registers, no tasks should be waiting on an interrupt that will never be generated.
In addition, interrupts are disabled (OR the CTLR witht he negation of the enable mask) when they occur and no tasks are currently on them. This ensures that we do not lose the interrupt information without requiring the kernel to store state of the interrupts. Combined with enabling interrupts on `AwaitEvent`, this allows us to delay interrupts until the handler task is ready to service it. In the case of RCV, this could cause overrun errors in the UART, so additional flow control actions may be necessary (currently not implemented).
The following actions are taken for each interrupt, assuming that a task is waiting on the interrupt.

1. RCV: If the FIFO is in use, copy from the data register into the provided buffer until the RXFE flag is set. Otherwise, read a single byte and return it as the return value.
2. XMT: Reschedule the blocked XMT handler task and disable the XMT interrupt.
3. MOD: Clear the MOD interrupt (write to the INTR register) and reschedule the blocked task.

7.3 Handlers

There are 4 handler tasks in total, `Uart(1|2)(XMT|RCV)Handler`, each corresponding to one of the new interrupts, with the `Uart1XMTHandler` also registering for the MOD interrupt.

7.3.1 RCV Handlers

RCV handlers are the simplest handlers, and do nothing more than wait for their correspond interrupt event and send the result to their parent server. UART2 uses a FIFO buffer with 8 bytes while UART1 sends a single byte.

7.3.2 UART2 XMT Handler

The UART2 XMT handler is also rather simple: wait for the XMT interrupt, Send to the parent server to request up to 8 bytes of data then writes it to the UART2 data register.

7.3.3 UART1 XMT Handler

This handler is much more complicated than the others, mainly due to the fact that it requires the CTS bit to be asserted. The control flow is as follows:

```
FOREVER:
    AwaitEvent(UART1_XMT_INTERRUPT)
    if (not cts):
        AwaitEvent(UART1_MOD_INTERRUPT)
    Send(server)
    write byte
    if (cts):
        AwaitEvent(UART1_MOD_INTERRUPT)
```

The handler waits for CTS before writing to the UART, and waits for !CTS after writing to the UART before looping around. This handles the race condition of two write requests in the space of a single CTS assert, the second occuring before the train controller has had the chance to deassert CTS. There is a potential race condition here in the case of a MOD interrupt occuring before the CTS check, which will be masked but the corresponding AwaitEvent will never be called. However, given the massive speed differences between the kernel and the train controller, the fix for this has been deferred to a later time.

7.4 Servers

There are two servers, `InputServer` and `OutputServer`, each handling both channels. The servers create the corresponding Handler tasks, and then block on receive. Each server can be sent two different types of messages: READ and WRITE. The `OutputServer` only accepts READ calls from its handlers, and the `InputServer` only accepts WRITE calls from its handlers.
Each server, for each channel, uses a circular buffer to store characters written to it, and replies to READ requests from this buffer. READ requests on the `InputServer` with insufficient characters will be blocked until a subsquent write gives it enough characters. However, for the `OutputServer`, a READ request will return any number of characters, and indicate the length of characters. This is done so that the COM2 FIFO can be used effectively while still printing a smaller number of characters, ie. echoing user input.

8 TrainController

The TrainController registers with the NameServer and creates a TrainSensorSlave that polls for sensors and awaits characters being transmitted from UART1. It then blocks on `Receive` awaiting requests from other tasks that wish to get the state of a sensor, await a sensor being tripped, or for the notifier to message it with the latest sensor data.

8.1 TrainSensorSlave

Blocks on `AwaitEvent` for 10 characters to be transmitted from UART1 at a time. Extracts the individual bits from the given characters and sets the status of its sensor to 1 or 0 if that sensor tripped (bit value > 0 means that the sensor was tripped). It then sends the updated results to the TrainController, and repeats the process.

8.2 API

The API allows for tasks to wait on sensors by calling `WaitOnSensor` at which point the calling task is added to that sensors wait queue. When the sensor is tripped, all the tasks in that queue are unblocked with a return value of 0. Tasks may block forever if the sensor never trips.

9 Shell Task

9.1 Commands

When inputting commands, it is important to note that they must exactly match their prototype or they will be interpreted as garbage by the terminal and equivalent report back `command: error command not found`. For example, to move train 1 at speed 10, the following command would be entered "tr 1 10" followed by a `RETURN` or `ENTER`. All commands must be terminated by a `RETURN` or `ENTER` for the terminal to process them. The syntax for issuing a command is ``command_name ARGUMENT_1 ARGUMENT_2 .... RETURN''. The following table lists the commands supported by this Kernel's shell:

Command	Argument 1	Argument 2	Description
go	N/A	N/A	Start the train controller (if not started).
stop	N/A	N/A	Stop the train controller (if started).
tr	1 - 80	0 - 14	Set the train specified by the first argument to the speed specified by the second argument.
ax	1 - 80	16 - 31	Run the auxiliary function specified by the second argument on the train specified by the first argument.
rv	1 - 80	N/A	Reverse the direction of the train specified by the first argument.
li	1 - 80	N/A	Turn on the lights on the train specified by the first argument.
sw	0 - 255	S or C	Throw the switch specified by the first argument ot straight (S) or curved (C) specified by the second argument.
ho	1 - 80	N/A	Turn on the horn on the train specified by the first argument.
add	1 - 80	N/A	Add a train to the track (useful to introduce new trains to the track).
rps	N/A	N/A	Play a game of Rock-Paper-Scissors.
time	N/A		Reports the current time when the command was called.
q	N/A	N/A	Halt the system and return to RedBoot.

9.2 Introducing New Trains

By default, the `TrainUserTask` that listens to the `Shell` assumes that the trains 45, 48, 49 and 50 exist on the track to allow the kernel to have some idea of what trains are currently running and where they might be on the track. Should you wish to use another train that currently is not in the above list, that train can be added to the track by running the `add` command listed in the above subsection.

9.3 Shell Display

The following picture highlights overlay layout of the terminal display: Below the top two description lines at the top of the display are the current time (to within 10 milliseconds) and the percentage of time that the CPU has been idle (calculated as fraction of the time spent doing nothing, running the null task, over the total time the CPU has been running). Below those two are the current state of the switches ('C' for curved and 'S' for straight),

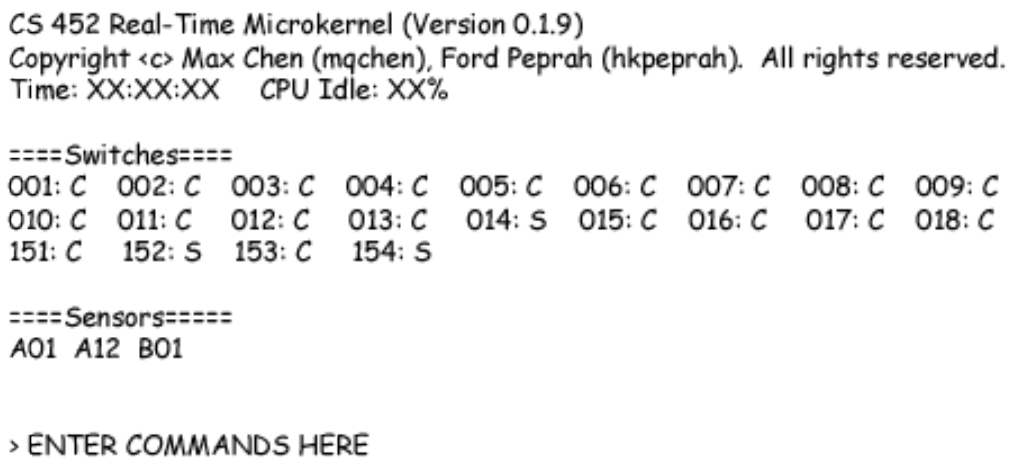

Figure 1: CS452 Microkernel Shell Display

and below those are the recently triggered sensors, read from left to right; the most recent triggered sensor is the farthest left and they scroll horizontally. Finally, below all of that information is the shell prompt where the user can enter information.

10 Known Bugs and Errors

There are known and unknown bugs and errors in our kernel system. Below we have listed some of the known bugs and errors and grouped them categorically. Recoverable errors are ones the system can compensate for at runtime in order to correct them and continue the state of the kernel. Unrecoverable errors are those that would require reprogramming in order to function properly. Lastly, User Interface Bugs are those tricky bugs that occur due to some configuration of the Kernel and generally do not affect how the application runs, but would affect how the user sees the interface.

10.1 Recoverable Errors

- Switch 14 is broken on th second track; to compensate this, the kernel switches it to the straight state which does not cause a derail, though a user could switch it back to curve and allow a derail to become possible again.

10.2 Unrecoverable Errors

- Train derails because of a switch state in the middle switches; in order to remedy this, the train functions should never allow a switch state involving the center switches that

would cause a derail.

- Known broken switches (e.g. switch 14 on the second track), and known faulty sensors can cause derails or locations to not accurately be picked up; to compensate for this, the kernel would have to know which sensors are broken and to compensate by looking one sensor ahead or behind that instead to estimate distance.

- There is a potential race condition that is well specified in the Serial IO section between the Kernel and the Train Hardware if it somehow beats the Kernel's CPU; refer to Serial IO section for more detail (section 7.3.3).

10.3 User Interface Bugs

- Depending on the state of the box prior to the application being loaded, if the previous group had not exited gracefully, cleared their FIFOs, or otherwise, there may be garbage input read by the IO servers when the kernel is loaded; this is characterized by random printing at the top of the terminal repeatedly that do not seem to be valid characters being typed by the user. To rectify this issue, reboot the box and reload the program onto it.

- During the boot operation, newlines may sometimes be printed erroneously by the terminal causing the display to be printed incorrectly, characterized by a display state not resembling the one listed above in the `Shell` section. This can be remedied by quitting the program, and starting it again.

11 MD5 Hashes

```
8afa04fd4ff12bc483271286d52dfa00  /u8/hkpeprah/cs452-microkern/bin/cs452-upload.sh
de6700ffc18bb2c8f15a491fc2929d13  /u8/hkpeprah/cs452-microkern/bin/md5.sh
7d3d938f3360ca46d07b07d6fed3711c  /u8/hkpeprah/cs452-microkern/bin/profiler.sh
40e6f5862869392d9733ea2d6defbb68  /u8/hkpeprah/cs452-microkern/include/bwio.h
bfcb67b08f11ead2eb221f89218918ff  /u8/hkpeprah/cs452-microkern/include/mem.h
19fbec18bdcd2bf2169e51577153fbcf  /u8/hkpeprah/cs452-microkern/include/string.h
d48a7284d156dec281b35b09b3ff7089  /u8/hkpeprah/cs452-microkern/include/syscall.h
3f3a66bb146ae1bcdcd7e194ecd0f5a9  /u8/hkpeprah/cs452-microkern/include/task.h
91425c50507432ecf2bfc92ae70589c4  /u8/hkpeprah/cs452-microkern/include/ts7200.h
972709eaa6994731bf2b26bc2bde6ce9  /u8/hkpeprah/cs452-microkern/include/types.h
d898edf77661ac9a98e2a0c6b9d9b9a6  /u8/hkpeprah/cs452-microkern/include/vargs.h
bff22a8329a113c8bf64d0607d71cb55  /u8/hkpeprah/cs452-microkern/include/utasks.h
e4386c39cbde55145ad3d3b161366929  /u8/hkpeprah/cs452-microkern/include/k_syscall.h
eac9f764270af5a8683da7e4ac5ee3c3  /u8/hkpeprah/cs452-microkern/include/stdio.h
101e3ed58d4739b0a7ae52df389e84dd  /u8/hkpeprah/cs452-microkern/include/stdlib.h
77c2f09c15c52f48075c3981553ba8a7  /u8/hkpeprah/cs452-microkern/include/term.h
4a2ec8e1f872ea9bad2494d72a9c1d60  /u8/hkpeprah/cs452-microkern/include/syscall_types.h
1624fa508a85025bed31a50a05048f6d  /u8/hkpeprah/cs452-microkern/include/kernel.h
02da5c5ed64ddf5a5ff08090e1d407cf  /u8/hkpeprah/cs452-microkern/include/hash.h
```

```
717b31cadb3b90f022dc0690530d1aab  /u8/hkpeprah/cs452-microkern/include/perf_test.h
65b51124e5ee634ebdbba3664aa22a63  /u8/hkpeprah/cs452-microkern/include/random.h
21519cac55397ad4c69970d00978d733  /u8/hkpeprah/cs452-microkern/include/server.h
ce8542472ac5ee1d570df56c365f4f12  /u8/hkpeprah/cs452-microkern/include/shell.h
973c15973ba0d9c7d687b936cc1010da  /u8/hkpeprah/cs452-microkern/include/clock.h
ec340a6df8d3a5537318f799b3824e3f  /u8/hkpeprah/cs452-microkern/include/util.h
7cb6397fc4af9f54ac2bf6ba897ee4bf  /u8/hkpeprah/cs452-microkern/include/interrupt.h
7ff8faa9d929453fa8cd82d0e7c32b19  /u8/hkpeprah/cs452-microkern/include/rps.h
1512795a5385a5e631e672e6d97fe228  /u8/hkpeprah/cs452-microkern/include/sl.h
89e35e2cf35d247f24787d12aaaa55e3  /u8/hkpeprah/cs452-microkern/include/uart.h
cd31bb05b0e8cdd7f5fc4e216615d9a2  /u8/hkpeprah/cs452-microkern/include/logger.h
cc4251fb50d53e34042d5a8e1193f58d  /u8/hkpeprah/cs452-microkern/include/train.h
a28d1c44081e0456ae6d9382a524b0a8  /u8/hkpeprah/cs452-microkern/include/controller.h
3296682e40d4b19c236a01b0b5c20427  /u8/hkpeprah/cs452-microkern/include/null.h
c4bee24fcb42fadd00dca64817d6c87c  /u8/hkpeprah/cs452-microkern/include/idle.h
8402c682ff31b15549689baa00ea45b6  /u8/hkpeprah/cs452-microkern/lib/libbwio.a
b881056cd427f67360bbfc3d79372ddb  /u8/hkpeprah/cs452-microkern/Makefile
8da00e714e5f8f92edb2fdce848f750b  /u8/hkpeprah/cs452-microkern/src/mem.c
e7bf3de21aa43bee5557e8f9bdb31017  /u8/hkpeprah/cs452-microkern/src/orex.ld
4a4a162d7de498e89ab976f72601e7ae  /u8/hkpeprah/cs452-microkern/src/string.c
84c2a87e1ee252f5aee7c425acc3de20  /u8/hkpeprah/cs452-microkern/src/syscall.c
a555344303bf524211bc9fbe497e0b84  /u8/hkpeprah/cs452-microkern/src/task.c
8806906ec643fa253b447e0d04e8e78e  /u8/hkpeprah/cs452-microkern/src/main.c
e8c071d962017d206731e8e46b36d89f  /u8/hkpeprah/cs452-microkern/src/k_syscall.c
e8356dc5dba2e1a64332eb15a422e86f  /u8/hkpeprah/cs452-microkern/src/stdio.c
4552615d00cd50b286bf2df8fcbbe1d2  /u8/hkpeprah/cs452-microkern/src/stdlib.c
50ccc6c9f639fc51d9ba527d39072657  /u8/hkpeprah/cs452-microkern/src/kernel.c
ac2f4cb4941838cf93dcce2b7808254e  /u8/hkpeprah/cs452-microkern/src/term.c
fb02e0ba097afaabae2927e17634270b  /u8/hkpeprah/cs452-microkern/src/hash.c
ab4b0499e37884b85ff6e4b30a5c1d4e  /u8/hkpeprah/cs452-microkern/src/random.c
ebdc392d17c3a0e8d3f2278dcc0f8315  /u8/hkpeprah/cs452-microkern/src/interrupt.c
9c0a318f9b13c28970bae4757f15ddca  /u8/hkpeprah/cs452-microkern/src/tasks/null.c
c882bc4746cd7b70a6d9a02fffb5a503  /u8/hkpeprah/cs452-microkern/src/tasks/shell.c
5e05c451d1fbb9eeb82e5e4577efeddf  /u8/hkpeprah/cs452-microkern/src/tasks/utasks.c
e410a8b26d9827107f987c0a6cb9e23a  /u8/hkpeprah/cs452-microkern/src/tasks/rps.c
9453e45434cea32c42a71bbec56c9bf8  /u8/hkpeprah/cs452-microkern/src/servers/clock.c
cc8ef3fd964f0a056444b32cd753eb74  /u8/hkpeprah/cs452-microkern/src/servers/server.c
eea71461300a2b127cbdc6c8a2c13712  /u8/hkpeprah/cs452-microkern/src/servers/uart.c
b00f6a438536a3ecdee7073b404e07dd  /u8/hkpeprah/cs452-microkern/src/servers/controller.c
5f7f0c87a919b38813d6e7426e385189  /u8/hkpeprah/cs452-microkern/src/util.c
32418c5808a85c6a90c438ecbeaf5a66  /u8/hkpeprah/cs452-microkern/src/logger.c
b367a8767446cd6d1d1e8bee0fba360d  /u8/hkpeprah/cs452-microkern/src/train.c
98637ce23460d9d7b8fe892ea6627d17  /u8/hkpeprah/cs452-microkern/src/idle.c
0c9c7e1968f6fa419d4342138e197149  /u8/hkpeprah/cs452-microkern/tests/tasks1.c
f1f8c6fd425032444162c32949728fee  /u8/hkpeprah/cs452-microkern/tests/tasks2.c
8ee655142dc66971e1479e506155655b  /u8/hkpeprah/cs452-microkern/tests/tasks3.c
cc584ca632e63d06fff15334d78bacd8  /u8/hkpeprah/cs452-microkern/tests/sltest.c
7393bef53f92842bc08ba2dcd0ac530d  /u8/hkpeprah/cs452-microkern/tests/uarttest.c
6ee1f24aac0ad5f8c8ca40b74954bd87  /u8/hkpeprah/cs452-microkern/tests/logtest.c
5dfb150de46a48f1cf5907abf7151196  /u8/hkpeprah/cs452-microkern/tests/fig8test.c
7bdc3688319a8fc4f27da5e99f76c313  /u8/hkpeprah/cs452-microkern/tests/perf_test.c
82e6a6dcc6b5a9ec57de149f94d6948c  /u8/hkpeprah/cs452-microkern/tests/traintest.c
5f5461846b8af8c1a18b82cc4c85176a  /u8/hkpeprah/cs452-microkern/tests/sensortest.c
```