CS 452: Train Control Part 1

Jason Sun (#20387090) and Shuo Chang (#20378235)

July 27, 2015

Contents

Part I
Administrative Details

1 Source Code Location

- Repository location is `gitlab@git.uwaterloo.ca:j53sun/cs452team.git`
- Checkout by label: `tc1`
- Or, checkout by SHA1 hash: 26cd1dcd
- Compiling by running `make k5`, which also copies the local `kernel.elf` to `/u/cs452/tftp/ARM/sunchang/k5.elf`

1.1 List of Submitted Files

See git commit at hash.

2 Program Operation

```
> load -b 0x00200000 -h 129.97.167.12 "ARM/sunchang/tc1.elf"
> go
```

Command	Action
`tr train_num speed`	set train speed
`rv train_num`	reverse train
`sw train_num direction`	switch a turnout
`e train_num speed`	create engineer for train and give speed
`h train_num sensor_char sensor_num`	halt on sensor
`m sensor1_char sensor1_num sensor2_char sensor2_num`	display timing from sensor1 to sensor2
`g`	go at previous train, speed
`0`	stop previous train
`c train_number speed sensor_group sensor_num num_loops`	Calibration mode: stop at sensor and delay, then repeat
`k track_A_or_B`	Load track A or B
`x nodeIndex offset`	Stop the train at offset millimeters before that node
`q`	Quit
`o`	Print turnouts again
`p`	Print track drawing again
`d`	Debug printout of active tasks
`?`	Prints out help message

Part II
Kernel Structure Details

3 Stack Layout

When the user does a syscall, the top of the user stack to store user's current registers. The layout of registers stored is:

```
[ R1 (PC)   ] <-- SP after storing trap frame
[ R2 (CSPR) ]
[   ...     ]
[ R12       ]
[ LR        ]
```

```
    [   ...     ] <-- SP at SWI instruction
```

Initializing the trap frame for the first time is done in `taskCreate()`, and for later context context switches, the trap frame is handled in `context_switch.s`, written in assembly code. On return, the result of the syscall is stored in `r0`, and execution resumes at function `swi()`, where the syscall occurred.

4 Syscalls

Syscalls defined in C functions, in `syscall.{c,h}` files. There is a `Syscall` structure that contains the syscall type, and args 1 and args 2.

- `int Create(int priority, void (*code)())` Schedule a task with specified `priority` and function pointer `code`.
- `int MyTid()` Return the task id for the calling task.
- `int MyParentTid()` Return the task id of the parent of the calling task.
- `void Pass()` No-op for entering the kernel.
- `void Exit()` Exits the calling task and never schedule it again.
- `int Send( int tid, void *msg, int msglen, void *reply, int replylen )`
- `int Receive( int *tid, void *msg, int msglen )`
- `int Reply( int tid, void *reply, int replylen )`

Added `arg3, arg4, arg5` to Syscall data structure to support a maximum of 5 system call arguments.

5 Tasks

A task can be created off a function pointer and represents a chunk of code to execute.

5.1 Task Descriptor

The kernel's TaskDescriptor:

```
typedef struct TaskDescriptor {
    int id;
    int parent_id;
    int ret;
    unsigned int *sp;
    Status status;
```

```
        int *send_id;
        void *send_buf, *recv_buf;
        unsigned int send_len, recv_len;
        struct TaskDescriptor *next;
    } TaskDescriptor;
```

Where a Status is

```
    typedef enum {
        ready,
        send_blocked,
        receive_block,
        reply_block,
    } Status;
```

Descriptor Notes

- Task id contains an index into a global table of task descriptors preallocated
- The `spsr` is manipulated by context switch code
- Not all 7 task status are needed
- The `send_id, send_buf, send_len, recv_buf, recv_len` are for IPC.

5.2 Task Creation

A task is created by specifying a priority, a function pointer, and parent task id. The `Create()` syscall is implemented by this function.

A task descriptor is filled in to the task table. Then a stack is allocated, a size of 4096 words. There is syscall to change a task's stack size. It also initializes a trap frame by setting`pc` to the value of the function pointer and saved stored program register. Currently only can create 128 tasks before failing to create more tasks.

Finally the kernel adds the task descriptor to priority queue.

5.3 Task Scheduling

Tasks each has a priority level. The scheduler tracks this tasks' priority via 32 ring buffer queues.

A bitmask keeps track of which of the 32 queues contains tasks. Using this bitmask, we efficiently computing the number of right leading zeroes in the bitmask with De Bruijn table lookup.

The kernel calls`taskSchedule()` on each loop, and the queue with the highest priority is returned. The head of that queue is rotated to be the tail and the pointer is returned as the next task to be scheduled.

5.4 Task Communication

Tasks who are able to make a system call are not in any queues.

There are three handlers for `Send()`, `Receive()`, and `Reply()` system calls, and sets the appropriate values.

1. `handleSend()` checks whether the intended receiver is currently receive blocked.
    (a) If receiver's status is in receive block, then copy the buffer from sender to receiver directly, and set it's status to reply blocked.
    (b) Else, enqueue to the receiver's send queue then update it's status to send blocked.
    (c) It also copies sender's reply buffer and reply buffer length into recv_buf, recv_len in sender's task descriptor for later use in `handleReply()`.
2. `handleReceive()` checks whether there are senders in receiver's send queue.
    (a) If there exists senders, then dequeue sender and move the message. Sender's status is set to reply blocked, and move the receiver to the ready queue.
    (b) If there are no queued senders, set receiver's status to receive blocked. Store the pointer to the buffer and length to the receiver's task descriptor.
3. `handleReply()` copies the message from the receiver to the sender. The sender's reply buffer address is tracked in its task descriptor, put there by `handleSend()`.

Memcpy The `memcpy()` is implmented in assembly to make message passing faster. Four registers are used, along with multiple load and unload assembly instruction. To deal with non-padded copying, the extra bits are copied by jumping into an unrolled loop [1].

5.5 Task Exit

Once a task is removed from the priority queues, the task will not be scheduled again. No effort is made to reclaim task descriptors.

6 Handling Interrupts

Before Kernel 3, software interrupts are triggered using `swi` via system calls, and IRQ is disabled in user mode. The context switch of the kernel made

[1] Duff's Device http://en.wikipedia.org/wiki/Duff%27s_device

assumptions regarding the registers on kernel entry. Specifically, `r0` was thought to contain a fixed pointer to the address of the static request struct, and thus was not saved and restored during the context switch. Furthermore, registers `r2` and `r3` are dedicated to store/load user `pc` and user `cpsr`. The solution worked at the time because the GCC handled saving/restoring of scratch registers in system call stubs.

By enabling hardware interrupt in Kernel 3, these assumptions are no longer true: in addition to the new IRQ mode introduced by hardware interrupt, interrupts can occur any stage of the user program execution. Scratch registers must be properly saved in order for the kernel to function correctly. The following lists will describe how the new context switch code work:

6.1 Kernel Exit

Assembly	Explaination
`stmfd sp!, {r0-r12, lr}`	Store all kernel registers onto kernel stack
`msr cpsr_c, #0xdf`	Change to system mode
`ldr sp, [r0, #12]`	Load task sp into sp
`ldr r0, [r0, #8]`	Load task return value into r0
`mov r1, sp`	Load sp into r1
`add sp, sp, #8`	Update sp to after popping user cpsr, pc
`msr cpsr_c, #0xd3`	Change back to supervisor mode
`ldmfd r1, {r2, lr}`	Load user cpsr and user pc to supervisor lr
`msr spsr, r2`	Load user cpsr to supervisor's spsr
`msr cpsr_c, #0xdf`	Change to system mode
`ldmfd sp!, {r1-r12, lr}`	Load user registers from user stack
`msr cpsr_c, #0xd3`	Change to supervisor mode
`movs pc, lr`	Execute user code

6.2 Hardware Interrupt Enter

Assembly	Explaination
`msr cpsr_c, #0xd3`	Go to supervisor mode
`stmfd sp!, {r0}`	Push r0 on the kernel stack
`msr cpsr_c, #0xd2`	Go to irq mode
`sub r0, lr, #4`	Put lr - 4 (pc_usr) to r0
`msr cpsr_c, #0xd3`	Go to supervisor mode
`mov lr, r0`	Put correct user pc to supervisor lr
`ldmfd sp!, {r0}`	Restore r0 from the kernel stack
`msr spsr_c, #0x50`	Set spsr to user mode (irq enabled)

Note The program counter automatically advances into kernelEnter after the last instruction in this routine.

6.3 Kernel Enter

Assembly	Explaination
`msr cpsr_c, #0xdf`	Change to system mode
`stmfd sp!, {r1-r12, lr}`	Store all user registers to user stack
`mov r1, sp`	Put user sp in r1
`sub sp, sp, #8`	Calculate user sp after pushing cpsr and pc
`mrs r2, sps3`	Put spsr (user cpsr) in r2
`msr cpsr_c, #0xd3`	Change back to supervisor mode
`stmfd r1!, {r2, lr}`	Store r2 (user cpsr), lr (user pc) to user stack
`ldmfd sp!, {r0}`	Load r0 (*task)
`str r1, [r0, #12]`	Store r1 (user sp) in task->sp
`str r0, [r0, #8]`	Store r0 in task->ret
`ldmfd sp!, {r1-r12, pc}`	Load the rest of the kernel registers from stack

7 AwaitEvent

Description of int `AwaitEvent(int eventType)`. A table of 64 task descriptor pointers keeps track which task is registered to await a given interrupt.

- When AwaitEvent is called, it adds the calling task into the table and sets the enable bit for that interrupt code.
- At most one task to block on a single event. Although if needed, multiple tasks can be changed up through the next field in the task descriptor if needed.
- If the event is related to IO, then the interrupt is set to enabled.

8 Hardware Interrupt Handler

The interrupt handler handles 6 interrupts (listed in order of handling priority):

- Timer 3 underflow interrupt
- UART 1 modem interrupt
- UART 1 transmit interrupt
- UART 1 receive interrupt
- UART 2 receive interrupt
- UART 2 transmit interrupt

When an interrupt occurs, the highest priority interrupt is handled. Only one interrupt is handled per kernel entry. If the interrupt is related to UARTs, then the interrupt is disabled in the UART. Furthermore, the handling of UART 1 transmit event is slightly different than the other interrupts; the waiting task is only unblocked when the kernel handles a transmit interrupt as well as a modem interrupt where both the CTS bit and the DCTS bit are asserted.

Part III
Core Servers

9 Nameserver

The nameserver should be created during kernel initialization by the first user task, such that its task id is entirely deterministic, and is be shared as a constant between tasks (for those tasks that do not use the syscall). Calls supported are:

```
int RegisterAs(char *name)
int WhoIs(char *name)
```

The nameserver keeps track of tasks in a `{name, task_id entry}`, in table. The table has a static size of 256. Insertion, and lookup, scans the entire table.

Linear scan is used instead of something with better performance because the nameserver is anticipated not to be called after a task initializes, therefore its performance is not important.

10 Clock Server

The clock server, together with the notifier, provides timing functionality for other user programs. Three functions are provided:

1. `Time()` returns the current system tick; a tick is defined to be 10 ms.
2. `Delay()` delays a task for certain number of ticks; and
3. `DelayUntil()` delays a task till a certain time.

These functions wrap `Send()` to the clock server; therefore, their only difference is the type of request they send. A `ClockReq` struct, composing of a request type and request data, is passed around the clock server, the clock notifier, and client tasks, to communicate requests.

```
typedef struct ClockReq {

    int type;
    int data;

} ClockReq;
```

The wrapper functions also loop up the clock server on the name server, and set it to a static variable declared within each function. It is build using the message passing mechanisms. It also registers with the nameserver so it could be discovered by other tasks.

The clock server keeps track of requests through a clock notifier.

10.1 Clock Notifier

The clock notifier waits for a timer interrupt to occur using the kernel primitive `AwaitEvent()` and uses timer 3 underflow interrupt. On interrupt, it gets unblocked by the kernel and sends a notification to the clock server, signaling it to increase current tick.

10.2 Delayed Task

This struct is used by the clock server to keep track of tasks blocked by `Delay()` and `DelayUntil()`. The tid is the tid of the task, the finalTick is the time that the delay expires; the next pointer is to support inserting/removing into DelayedQueue, a singly circular linked-list.

```
typedef struct DelayedTask {

    int tid;
    unsigned int finalTick;
    struct DelayedTask *next;

} DelayedTask;
```

10.3 Timer and Interrupt Control Units

The timer 3 in the EP9302 SoC is used to track time. It is set up to use 508 kHz clock in periodic mode, with load register set to 5080 (10 milliseconds interval between interrupts).

The SoC contains 2 PL190 interrupt controllers. On kernel start, the 20th bit (1<<19), of the second interrupt controller, which is the bit corresponds to timer interrupt, is enabled. Both interrupt control units' select bits are set to 0, which enables IRQ mode. When the program is about to exit, the 20th bit of the second interrupt controller is cleared.

11 Interrupt Driven I/O

There are kinds of 4 servers and 4 notifiers:

- Monitor-in server/notifier, handling input from the keyboard.
- Monitor-out server/notifier, for sending to the terminal.
- Train-in server/notifier, handling input from the train control unit.
- Train-out server/notifier, for sending to the train control unit.

This seperation is necessary because there are different interrupts that need to be handled, and it is conceptually easier to program them seperately despite the input servers being similar and output servers being similar.

Notifiers do similar tasks, which we will cover.

11.1 Servers

On startup the server is responsible for creating its own notifier. The server is almost always receive blocked, waiting on the notifier. However, servers are responsible for dealing with interrupts and handling (somewhat complicated) interrupt procedures. We want the notifiers, which are the interrupt handlers, to return to waiting for interrupts as quickly as possible.

11.1.1 Send Data

Data can be accepted as a string, using `PutString(struct String*) or PutStr(char*)`, and put into a send buffer. Or it can be accepted as a character at a time. Sending data only accepts a string to guarantee entire command is put into the send buffer atomically. This is to avoid interleaving commands from two different tasks, if we were to provide a single character sending command such as `Putc(char)`.

The server then provides data to the notifier from its send buffer when it checks in.

11.1.2 Receive Data

The receive server gets data from a notifier event. It buffers that data and replies to the appropriate task. There is only `Getc()` supported.

11.2 Notifiers

Notifiers are very simple, they wait for input interrupts by calling AwaitEvent(). When AwaitEvent() unblocks, it the notifier returns the volatile data back to the server.

Part IV User Tasks

12 Input Parsing

Parsing takes place after a complete string has been read in, meaning the user has pressed the return key. Then each character is fed into a parser. The parser is a finite state machine, and transitions the state upon acceptable valid. Upon unacceptable input the machine goes into Error state and all subsequent characters are colored red. For valid instruction, it calls the appropriate functions.

Quitting calls the syscall `Halt()`, which breaks out of the main loop of the kernel. Essentially it dumps all the active tasks on the floor.

13 Clock Drawer

Calls `Time()` and goes into an infinite loop that:

1. Prints time.
2. Increment time by 10 clicks (1 ms).
3. Calls `DelayUntil`.

DelayUntil is used instead of Delay to avoid clock slew, because the clockserver calls Time() each time we use Delay. Calculating the DelayUntil time ourselves therefore eliminates clock slewing.

Part V
Train Control and Related Tasks

14 Sensor Update

The sensor updates from the track is received by the trainOutServer. It is polled by the sensorWorker task by calling Getc(COM1), time-stamped by calling Time(), displayed onto the screen, then delivered to the sensor server. The sensor server then stores them into a buffer; the sensorCourier then calls Send() to the sensor server, which is replied with data from the buffer.

All sensor updates are attributed to the same engineer task as we only support one train at this moment.

14.1 Sensor Worker

The sensor worker is created by the sensor server. On start, the sensor worker calls Putc(COM1, 133) to request for sensor data for all sensors. It then receives 10 bytes of sensor data. When the first of the 10 bytes is received, the courier calls Time(), which is used as the timestamp for all 10 sensor updates. On receiving each byte, the sensor worker prints out the triggered sensors onto the screen. For each byte, the sensor worker then sends to the sensor server with both the sensor and the timestamp.

To prevent stuck sensor from repeatedly being triggered, the courier saves each byte it receives in an character array of size 10. When a 0 byte is received by the courier, it sets the stored state to 0; else, the sensor worker AND the byte with the NEGATION of the stored byte. The result is printed and delivered only when the AND result is non-zero.

14.2 Sensor Server

The sensor server receives from both the sensor courier and and the sensor worker. The sensor worker moves sensor updates from the COM1 output server to the sensor server; the sensor courier moves sensor updates from the buffer of sensor server to the engineer task.

15 Train Control

There is one controller for many trains. At the moment there is just one train control supported so we have one engineer and no controller.

15.1 Track Data Structure

The provided track data structure is used. On start of the kernel, the user could load either track A or track B using the k {a|b} command.

15.2 Track Controller

Not implemented in TC1. The train controller will manage track reservation.

15.3 Engineer

A train engineer is the task that calculates information such as displacement and velocity of a train. It is a server which receives from the LocationWorker, CommandWorker, SensorCourier, and the parser. Associated with the Engineer are data such as:

- State
  - Init: the initial state when the engineer is created.
  - Stopped: when the train has been given speed 0, and has finished "Decelerating"
  - Running: when the train is running in constant velocity
  - Reversing: when the train has came to a complete stop, and just issued a reverse command
  - Decelerating: when the train is going in constant velocity, and the command of setting speed to 0 is executed
  - Accelerating: when the speed is transitioning from 0 to a value that is greater than 0
- Position and displacement
  - Direction of the train, which forward is defined as the pick up being at the front of the train
  - Previous sensor node
  - Next sensor node
  - Previous track node
  - Next track node
  - Distance after the previous track node
- Speed (from 0 to 14)
- Velocity
- Timestamps for the previous sensor update
- A buffer of queued commands

15.3.1 Update And Display Of Location Information

The display of location information is managed by both the engineer and the LocationWorker. The LocationWorker is a worker task created by the Engineer that calls Send() to the engineer with an "update_location" request to the 5 ticks. After the engineer process the request, it is replied with data to be printed onto the screen. Upon being unblocked, it then prints out the replied data to COM2 output server via a printf() call.

On receiving an "update_location" request, the engineer computes it's velocity based on the current state of the train. For example, if the state is "Decelerating", then the acceleration estimation of the train is computed, and velocity is decreased until zero; otherwise, if the state of the train is anything other than "Stopped", the velocity is computed by `(distance between previous and next sensor) / (calibrated time between the previous and next sensor)`. If the current train state is "Reversing", the previous and the next node, as well as the previous and the next sensor nodes recorded by the engineer, are swapped.

With velocity calculated, the task then could compute how far it's gone over previous track node, and the remaining distance to the next track node. If the remaining distance is equal to or smaller than 0 and that the next node is not a sensor node, then the current node and the next node is updated and replied to the LocationWorker. If the engineer has yet to reach the next node, the LocationWorker is replied with the updated distance to next node.

If the next node is a sensor and the distance to the sensor node is smaller or equal to zero, a negative distance is outputed. This is done because we only reach a sensor when the engineer receives a sensor update.

15.3.2 Linear Deceleration

The deceleration is computed using the following formula:

When the train enters into the "Decelerating" state, this acceleration value is computed once, and the velocity is decremented according to the deceleration until it reaches zero, then the state of the train is set to "Stopped".

15.3.3 Sensor Update

The engineer task notified of sensor update via the sensorCourier, with the other end being the sensorServer. The sensorCourier first calls Send() to the sensorServer, and when the sensor server receives an update, the sensorCourier is replied with the sensor node index and a timestamp. The courier then call Send() to the engineer task with the exact data.

When a sensor update is delivered to the engineer, the engineer knows that a sensor has been triggered. The sensor updates the timeDeltas table that stores time differences between pairs of sensors, updates the previous landmark, next landmark, distance to next landmark, previous sensor (which is the sensor in this sensor update), current velocity (`length of segment between previous & next sensor / time since last sensor`), expected time (which is looked up from timeDeltas table), the actual time (which is the timestamp of the sensor update), and error (`abs(actual time - expected time)`).

15.3.4 Execution Of Train Commands And Command Notifications

Previously, the execution of train commands are directly sent to the COM1 input server by the parser. The downside of this solution is that it is not scalable for multiple trains, since reverse require two Delay() calls. Another problem that associates with this is that in the original solution where the parser task puts commands to COM2 input server, the engineer task is unaware of when these bytes are actually put into the buffer. The commandWorker is introduced to resolve these issues.

On start, the engineer creates a commandWorker. The commandWorker then immedietly becomes receive blocked on the engineer. When the engineer receives a command from the parser, the engineer puts the command into it's command buffer. The command worker would then be taken off the send queue by an Reply() call with the first command on the command queue.

The execution of the `tr` and `rv` command is carried out by the commandWorker. Setting the speed is straight forward, where the worker simply print two bytes, indicating the desired speed and the train number. For reverse, 300 ticks of delay is inserted between setting the train to speed 0 and setting reverse, and a delay of 15 ticks is inserted between the reverse command and the setting of previous speed.

After each set speed or set reverse command, the commandWorker calls Send() to the engineer, letting it know that the certain action has been carried out. For an `rv` call, the command worker receives three notifications: SetSpeed (0), SetReverse, and SetSpeed (original speed). In this way, the engineer is able to garauntee to have the correct internal state when various commands are executed.

15.3.5 X Marks The Spot

For the train has to be able to stop at arbitrarily far from a sensor position, the engineer receives a "X Marks The Spot" command from the parser. The user inputs "x [node index] [offset]" from the terminal, and the parser task would Send() to the engineer task. On receiving the message, the track node is retrieved using the track index. In subsequent update_location messages, the displacement between the current landmark and the target landmark is checked, with the offset taken into consideration. If the distance is smaller or equal to the stopping distance of the current speed, the stop command is issued.

15.3.6 Path Finding

Pathing is not implemented. But in the future, when the engineer sent an instruction to navigate to another position, it calculate a path using Djikstra's algorithm, and avoid revisiting the same nodes over and over. Exit nodes would have a distance of infinity to make the path avoid using it. After planning, the engineer requests ownership of the track it needs before it starts moving forward. As it moves, it sets the turnouts (which it owns) that are at most its stopping distance along its path.

16 Calibration

Two kinds of data are needed in order to achieve the goal of stopping at a given landmark on the track: the constant velocity and the stopping distance.

16.1 Constant Velocity

A two dimensional array is declared in the engineer task to store time differences between any two sensors: int timeDeltas[NUM_SENSORS][NUM_SENSOR];

Constant velocity is calibrated by populating the timeDeltas table described above. The actual data recorded in the two-dimensional table, with the two indices being sensor node indices, are time difference between the two sensors. The table is updated upon hitting a new sensor in a feedback loop fashion for accurate train velocity estimation $v_1 = (1 - \alpha)v_0 + \alpha v'$ where α is the learning rate. We have $\alpha = 85\%$.

16.2 Stopping Distance

The stopping distance of of the train is manually measured, with the help of our calibration command, which issues stopping command when a particular sensor is measured. The stopping distance is then measured by measuring the distance between the head of the train and the sensor. There is one stopping distance for each train speed of a particular calibrated train.

Currently the data for stopping distances, which is an arrays of size 15, is stored within the engineer task. This set of stopping distances, mapped to speed 0 to 14, is measured with the pick-up of the train in the front (defined as forward direction of the train). In the case that the train is running in the "backward" direction, which means that the pick up is at the end of the train, an additional offset of 105mm is added to the stopping distance.