CS 452: Kernel 4

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

June 18, 2015

Contents

Part I
Administrative Details

1 Source Code Location

- Repository location is `gitlab@git.uwaterloo.ca:j53sun/cs452team.git`
- Checkout by label: `v4.0`
- Or, checkout by SHA1 hash: `5b9775d57d8b16755a9a55c1569e7b14a6c49614`

- Compiling by running `make k4`, which also copies the local `kernel.elf` to `/u/cs452/tftp/ARM/sunchang/k4.elf`

1.1 List of Submitted Files

2 Program Operation

```
> load -b 0x00200000 -h 129.97.167.12 "ARM/sunchang/k4.elf"
> go
```

3 Program Output

```
RedBoot> go
```

3.1 Output Explanation

Part II
Kernel Structure Details

4 Context Switch

4.1 From kernel space to user space

1. `TaskDescriptor *td`, `Syscall **request` are passed to `KernelExit()`
2. Store all kernel registers onto kernel stack
3. Change to system mode
4. Put `td->sp` to user's `sp`
5. Put `td->ret` to `r0` which returns result of system calls to user task
6. Load all user registers: `r1` contains `pc` of user mode, whereas `r2` contains the saved `cpsr` of user mode
7. Change back to supervisor mode
8. Put saved user mode `cpsr` into `spsr` of supervisor mode
9. `movs pc, r1` to jump to user code while simultaneously change `cpsr`

4.2 From user space to kernel space

1. Put `lr` of supervisor mode in `r1`
2. Put `spsr` of supervisor mode, which is the saved `cpsr` of user mode, into `r2`
3. Change to system mode
4. Store user registers `r1-r12` and `lr`, to user stack
5. Move `sp`, `r0` to `r2`, `r3`
6. Change back to supervisor mode
7. Load multiple from stack into `r0`, and `r1`: `r0` contains pointer to task descriptor, `r1` contains pointer to pointer to request
8. Store `r2` to `td->sp`, `r3` to `*request`
9. Load the rest of the kernel's registers (`r2-r12`) from stack

4.3 ARM Assembly Code

The piece of code responsible for context switch is:

```
KernelExit:
    stmfd   sp!, {r0-r12, lr}
    msr cpsr_c, #0xdf
    ldr sp, [r0, #12]
    ldr r0, [r0, #8]
    ldmfd   sp!, {r1-r12, lr}
    msr cpsr_c, #0xd3
    msr spsr, r2
    movs pc, r1
KernelEnter:
    mov r1, lr
    mrs r2, spsr
    msr cpsr_c, #0xdf
    stmfd   sp!, {r1-r12, lr}
    mov r2, sp
    mov r3, r0
    msr cpsr_c, #0xd3
    ldmfd sp!, {r0, r1}
    str r2, [r0, #12]
    str r3, [r1]
    ldmfd sp!, {r2-r12, pc}
```

5 Trap Frame

When the user does a syscall, a trap frame is set up on 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.

6 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.

7 Tasks

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

7.1 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 pre-allocated
- 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.

7.2 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.

7.3 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.

7.4 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].

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

7.5 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.

8 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 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:

8.1 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

8.2 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.

8.3 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
`msr cpsr_c, #0xd3`	Change back to supervisor mode
`mrs r2, spsr`	Put spsr (user cpsr) in r2
`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

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 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.

12 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.

13 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.

13.1 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.

13.1.1 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.

13.1.2 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.

13.2 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
Train Control and Related Tasks

14 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.

15 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.

16 Sensor Polling

A sensor task polls the sensors continously. It records the poll into a circular buffer, and draws onto screen.

An improvement can be made, which is to only draw if data has changed.

17 Train Control

This is not implemented in K4. But there is design in the works.

The idea is that there is one planner for many trains. It acts like the trains' headquarters.

17.1 Planner

Planner measures train's (esitmated) positions speed on the track, and keeps a mapping of train number to their associated profiles.

Feedback loop for accurate train velocity estimation $v_1 = (1 - \alpha)v_0 + \alpha v'$ where α is the learning rate.

Train has to be able to stop at arbitrarily far from a sensor position.

17.2 Train

The goal is to track all the useful data associated with a train into a Train class. Useful data such as:

- Position
- Speed (from 0 to 14)
- Velocity
- Last hit sensors and time pair

In an infinite loop, it sends to the planner, and execute the command that was replied.

18 Sensors Calibration

This is not implemented in K4.

There is a delay from the moment a sensor is triggered, to the moment that it gets registers with the task responsible. These delay arises from a combination of the following.

Delay	Time	Reason
Polling	100	
Train controller	?	
Communication	$4(2 + 10) = 48msec$	
Computation	0	Very low number of assembly instructions required

Train is approximately 50 cm/s and in 1 ms can move 5 cm. The best precision in the demos are within 1 cm. Passing by different sensors is correlated.

18.1 Calibration

This is not implemented in K4.

Drive the train, give it a stop command so that the pickup is right at the sensor. Measure the time that you get the stop command and the time when the sensor trips, and calculate the time. Calculate it as a distance, so the constant part of the error go away and only the random parts are left.