# CS 452: Kernel 3

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

June 9, 2015

This document builds upon Kernel 2 and Kernel 1 documentation, and only describes the changes.

## 1 Program Operation

Compiling by running `make prod`, which also copies the local `kernel.elf` to `/u/cs452/tftp/ARM/j53sun/k2.elf`.

```
> load -b 0x00200000 -h 129.97.167.12 "ARM/sunchang/k3.elf"
> go
```

See also Source Code location section.

## 2 Kernel Data Structures

### 2.1 DelayedTask

The DelayedTask 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;
```

### 2.2 DelayedQueue

The DelayedQueue struct is used by clock server to implement linked-list of delayed tasks. It only consists of a single tail pointer. The list is empty if the tail pointer is NULL, and the head of the list can be obtained from tail->next.

```
typedef struct DelayedQueue {
    DelayedTask *tail;
} DelayedQueue;
```

### 2.3 ClockRequest

The ClockRequest struct is used to communicate between the clock server, the clock notifier, and client tasks through calling the wrapper functions Time(), Delay(), DelayUntil(). The type can be one of {NOTIFICATION, TIME, DELAY, DELAY_UNTIL}, and data maybe an argument accepted by Delay() and DelayUntil().

```
typedef struct ClockReq
{
    int type;
    int data;
} ClockReq;
```

## 3 Kernel Algorithms

In the third part of the kernel, the following features are added:

1. Context switch support for handling hardware interrupt
2. Setup timer and interrupt control units
3. Hardware interrupt handler
4. Implementation of AwaitEvent()
5. Clock server and notifier, Time(), Delay(), and DelayUntil() functions

### 3.1 Context Switch on Hardware Interrupt

Before Kernel 3, 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, the 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:

### 3.1.1 kernelExit

- Store all kernel registers onto kernel stack

```
stmfd sp!, {r0-r12, lr}
```

- Change to system mode

```
msr cpsr_c, #0xdf
```

- Load task->sp into sp

```
ldr sp, [r0, #12]
```

- Load task->ret into r0

```
ldr r0, [r0, #8]
```

- Load sp into r1

```
mov r1, sp
```

- Update sp to position after popping user cpsr, pc

```
add sp, sp, #8
```

- Change back to supervisor mode

```
msr cpsr_c, #0xd3
```

- Load user cpsr, user pc -> supervisor lr

```
ldmfd r1, {r2, lr}
```

- Load user cpsr to supervisor's spsr

```
msr spsr, r2
```

- Change to system mode

```
msr cpsr_c, #0xdf
```

- Load user registers from user stack

```
ldmfd sp!, {r1-r12, lr}
```

- Change to supervisor mode

```
msr cpsr_c, #0xd3
```

- Execute user code

```
movs pc, lr
```

### 3.1.2 irqEnter

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

- Go to supervisor mode

  ```
  msr cpsr_c, #0xd3
  ```

- Push r0 on the kernel stack

  ```
  stmfd sp!, {r0}
  ```

- Go to irq mode

  ```
  msr cpsr_c, #0xd2
  ```

- Put lr - 4 (pc_usr) to r0

  ```
  sub r0, lr, #4
  ```

- Go to supervisor mode

  ```
  msr cpsr_c, #0xd3
  ```

- Put correct user pc to supervisor lr

  ```
  mov lr, r0
  ```

- Restore r0 from the kernel stack

  ```
  ldmfd sp!, {r0}
  ```

- Set spsr to user mode (irq enabled)

  ```
  msr spsr_c, #0x50
  ```

#### 3.1.3 kernelEnter

Change to system mode

```
msr cpsr_c, #0xdf
```

Store all user registers to user stack

```
stmfd sp!, {r1-r12, lr}
```

Put user sp in r1

```
mov r1, sp
```

Calculate user sp after pushing cpsr and pc

```
sub sp, sp, #8
```

Change back to supervisor mode

```
msr cpsr_c, #0xd3
```

Put spsr (user cpsr) in r2

```
mrs r2, spsr
```

Store r2 (user cpsr), lr (user pc) to user stack

```
stmfd r1!, {r2, lr}
```

Load r0 (*task)

```
ldmfd sp!, {r0}
```

Store r1 (user sp) in task->sp

```
str r1, [r0, #12]
```

Store r0 in task->ret

```
str r0, [r0, #8]
```

Load the rest of the kernel registers from stack

```
ldmfd sp!, {r1-r12, pc}
```

### 3.2 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.

## 3.3 Hardware Interrupt Handler

The interrupt handler only handles timer 3 underflow interrupt at the moment. Status bit of the second ICU is checked for interrupt; if interrupt occurred, clear the interrupt in timer device and if there is a task blocked on timer event, put it back to the ready queue.

## 3.4 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.

## 3.5 Clock Server and Notifier

The clock server, together with the notifier, provides timing functionality for other user programs. Three functions are provided: Time(), which returns the current system tick; Delay(), which delays a task for certain number of ticks; and DelayUntil(), which delays a task till a certain time. A single tick is defined to be 10 milliseconds (see 2.2 Timer Setup for details).

### 3.5.1 Clock Server

The clock server serves requests from the notifier task and client tasks calling Time(), Delay(), and DelayUntil(), using the message passing mechanisms built in Kernel 2. It also registers with the name server so it could be discovered by other tasks. Here are the algorithms:

**clockServerTask:**

1. Initialize data structures
2. Register with name server
3. Spawn notifier
4. Forever loop
   (a) Receive ClockRequest
   (b) If request is notification
      i. Reply
      ii. Increment tick

iii. removeExpiredTasks()

(c) If request is from Time()

i. Reply with tick

(d) If request is Delay()

i. insertDelayedTask(tid, request.data + tick)

(e) If request is DelayUntil()

i. insertDelayedTask(tid, request.data)

**removeExpiredTasks:**

1. If tail pointer is NULL
   (a) return
2. cur := tail->next
3. Forever loop
   (a) If curr->finalTick > current time
      i. tail->next := curr;
      ii. break
   (b) Reply to curr->tid (unblocks a task)
   (c) If curr = tail
      i. tail = NULL
      ii. break
   (d) curr := curr->next

**insertDelayedTask(task,finalTick):**

1. If tail pointer is NULL
   (a) tail := task
   (b) task->next := task
   (c) return
2. curr := tail
3. Forever loop
   (a) If curr->next->finalTick >= finalTick
      i. task->next := curr->next
      ii. curr->next := task
      iii. return

(b) If curr->next = tail

i. task->next := curr->next

ii. curr->next := task

iii. tail := task

iv. return

(c) curr := curr->next

### 3.5.2 Clock Notifier

The clock notifier waits for a timer interrupt to occur using the kernel primitive AwaitEvent(); on timer 3 underflow interrupt, it gets unblocked by the kernel and sends a notification to the clock server, signaling it to increase current tick. The Send() call is subsequently unblocked by the clock server.

**clockNotifier:**

1. pid := MyParentTid()
   (a) pid now contains the tid of the clock server
2. Create a ClockRequest, req
3. req.type := NOTIFICATION
4. Forever loop
   (a) AwaitEvent(TIMER_EVENT)
   (b) Send(pid, &req, sizeof(req), 0, 0)

### 3.5.3 Time(), Delay(), and DelayUntil()

Time(), Delay(), and DelayUntil() are functions wrapping Send() to the clock server; therefore, their only difference is the type of messages they send. The wrapper functions also loop up the clock server on the name server, and set it to a static variable declared within each function.

1. static clockServerTid := -1
2. If clockServerTid $< 0$
   (a) clockServerTid := WhoIs(CLOCK_SERVER_NAME)
3. Put argument and type of request {TIME, DELAY, DELAY_UNTIL} in a ClockRequest object
4. Send the ClockRequest object to clockServerTid
5. return 0

### 3.6 Game Clients Tasks

A client task is created by the first user task. It immediately sends to its parent, the first user task, requesting a delay time,$t$, and a number,$n$, of delays. It then uses WhoIs to discover the tid of the clock server. It then delays$n$ times, each time for the time interval,$t$. After each delay it prints its tid, its delay interval, and the number of delays currently completed on the RedBoot terminal. Finally it exits.

### 3.7 First User Task

The first user task creates the name server, the clock server, and four client tasks. It then executes Receive four times, and Replies to each client task in turn. It then exits.

## 4 Source Code Location

- Repository location is `gitlab@git.uwaterloo.ca:j53sun/cs452team.git`
- SHA1 hash of commit for submission: `FIXME`

## 5 Program Output

```
RedBoot> go
clientTask 8 interval 71 delays completed 3 total time 213
clientTask 7 interval 33 delays completed 6 total time 213
clientTask 5 interval 23 delays completed 9 total time 217
clientTask 4 interval 10 delays completed 20 total time 253
clientTask 4 exiting...
clientTask 5 exiting...
clientTask 7 exiting...
clientTask 8 exiting...
```

**Explanation** The first 4 lines of the program output are the output from the client tasks, after calling Delay() and being unblocked by the clock server. Task 4 delayed by 20 times with 10 as the delay interval; task 5 delays 9 times, 23 ticks each; task 7 delays 6 times with 33 as delay interval, and task 8 delays 3 times, 71 ticks per interval.

Task 4 is supposed to delay 200 ticks in total, task 5 is supposed to delay 207 seconds in total, task 7 is supposed to delay 198 ticks in total, and task 8 is supposed to be delayed 213 ticks in total. As we can see, task 8 finished first, followed by 7, 5, and 4. This suggests that although task 4's total delay is relatively shorter than 7, it had to be delayed more times, therefore is scheduled more often. Since the 4 client tasks have the same priority, the task with the least delay times completed first.