CS 452 Assignment 3

Victor Scurtu (20678028) and Alex Danila (20516806)

February 10, 2020

Operating Notes

The executable file is found in the student environment under:

```
/u/cs452/tftp/ARM/vscurtu/k3/kernel
```

The code can be found in the following repository:

```
https://git.uwaterloo.ca/vscurtu/cs452_kernel
```

To build, use the following commands at the root of the repository on a student environment computer:

```
make
```

To run the precompiled kernel, use the following command in redboot:

```
load ARM/vscurtu/k3/kernel; go
```

The program executes, prints, then exits. No interaction is expected.

Table of Contents

This program description assumes that reports for past assignments have been read before hand. It describes only the set of changes from past assignments, and does not go into detail about kernel components implemented in them.

Context Switch

Originally, switching context was done through two assembly defined functions, `enter_kernel` and `enter_user`. These functions have been modified to accommodate hardware interrupts.

Enter Kernel

In essence, the process of saving the user state and restoring the kernel state remains the same. However, to support hardware interrupts, `enter_kernel` has been split into two handlers, `hardware_enter_kernel` and `software_enter_kernel`, and `finish_enter_kernel`, which is branched to by both handlers.

Both handlers switch to system mode to retrieve the user's stack pointer. In addition, `hardware_enter_kernel` switches back to IRQ mode and subtracts 4 from the LR, while `software_enter_kernel` switches back to supervisor mode.

Finally, `finish_enter_kernel` handles saving the user's registers, loading the kernel's and returning to the main C-language loop where the interrupt or system call is handled and another task is scheduled and run.

A minor change was made to the order in which user registers are saved. The user's PC is now the last register saved so it is at the top of the stack. This enables the atomic restore of user mode CPSR and PC using `movs`, described in the `enter_user` section.

A deliberate decision was made to execute kernel code in IRQ mode after `hardware_enter_kernel` and in supervisor mode after `software_enter_kernel`. This is because the kernel uses the current processor mode to determine if it is handling a hardware interrupt or a system call.

Enter User

`enter_user` has also been split into several parts. When `enter_user` is called from the kernel, it saves the stack pointer into `r3`, checks the current processor mode, and branches to that mode's version of `enter_user`. Both `supervisor_mode_enter_user` and `irq_mode_enter_user` ensure that the stack pointer for supervisor mode and IRQ mode

match by switching to their respective counterpart mode and copying the address from `r3` into the stack pointer. This is done so that entering the kernel can work correctly from either mode. Finally, both jump to `finish_enter_user`, which saves the kernel registers and restores the user registers from the stack.
Finally, `finish_enter_user` restores the user's saved PC into LR and the saved CPSR into SPSR, since they are now both at the top of the stack. After restoring the first 15 registers, the `movs pc, lr` instruction atomically restores the user's PC and switches the processor to user mode by loading SPSR into CPSR.

Hardware Interrupts

`interrupt.h` and `interrupt.c` provide an interface for enabling and disabling interrupt sources in the VIC and for handling interrupts when they are generated. Currently only `TC1UI`, `TC2UI`, and `TC3UI` are available to be enabled.

Initialization

Hardware interrupts are enabled in the VIC during the `kinit` phase using the `enable_interrupt` interface function which enables the appropriate bits in the VIC. For the implementation of `AwaitEvent`, Timer 1 is enabled in 2kHz rundown mode, with an initial value of 20. This will cause it to underflow and generate an interrupt after 10 milliseconds.

Handling

After returning from `finish_enter_kernel`, the kernel updates the stack pointer in the structure of the running task, as it did in K2, and then checks the current processor mode. If it is in supervisor mode, it is assumed that a system call occurred and `handle_swi` is called as usual. Otherwise, if the processor is in IRQ mode, `handle_interrupt` is called.

Currently, `handle_interrupt` assumes that only one interrupt will be generated at once. This is a valid assumption since interrupts will only be enabled on one clock for this assignment, with no other interrupt sources enabled.

Before handling the interrupt that occured, `handle_interrupt` first sets the current running task to `TASK_READY` and pushes it onto the schedule queue.

`handle_interrupt` checks both VICs for the interrupt source. Once found, the interrupt is handled by waking the task that was waiting on that event, and clearing the interrupt source. Here, waking a task means setting its return value, setting its state to `TASK_READY` and pushing it back onto the schedule queue.

Idle Task

The idle task is somewhat special. Since it is responsible for printing idle statistics, the kernel expects that it declares three global unsigned integers: `idle_time`, `start_time`, and `end_time`. These are declared in `idle_task.h`. The kernel will update these as it measures task run times.

Initialization

During `kinit` the scrollback region is set to start from line 2 onwards using VT-100 escape sequences and `IDLE:  0%` is printed on the first line. This reserves the first line for the idle task to print the idle time percentage.

Also during `kinit`, the kernel writes `0xAA` to `SYS_SW_LOCK_ADDR` and enables bit 0 of `SW_HALT_ENABLE_ADDR` to enable transitions into Halt mode, as described in the EP93xx guide.

The kernel now creates both `umain` and `idle_task`. The idle task is created at the lowest priority, with the expectation that no other task will be given that priority, so that it will only run when all other tasks are blocked.

Running

Since it has the lowest priority, the idle task will only be scheduled to run when all other tasks are blocked. It runs an infinite loop where it first prints the idle percentage on the first line and restores the cursor to its original position using VT-100 escape sequences. Then it puts the processor into Halt mode, where it stays until an interrupt occurs. Since it only prints once per schedule and it does so much more quickly than the 10 millisecond interrupts are generated, the idle task will not be interrupted while printing on the reserved first line.

Measuring Idle Time

Task run times are measured by saving the debug lower 32 bits of the debug timer before and after a call to `enter_user`. If the idle task was the last running task, the task run time is added to `idle_time`. The way timing is done also implies that `end_time` is the value of the debug timer at measurement time, which is also approximately the total system runtime. The idle task percentage is thus `idle_time/(end_time/100)`.

Long Running Tasks

The tasks structure now keeps track of the number of active tasks. That is, tasks which have been created and have not exited, including blocked tasks. The expected number of long running tasks including the idle task is defined as `LONG_RUNNING_TASK_COUNT`.

Once the number of active tasks exceeds this number, it is taken to mean that initialization of long running tasks such as servers has finished. Long running tasks are those that infinitely loop. If the number of active tasks ever drops back down to equal `LONG_RUNNING_TASK_COUNT`, it is assumed that all the main user tasks have exited and no more tasks will be started, so the kernel can terminate.

In later assignments, checking against the `LONG_RUNNING_TASK_COUNT` will be replaced with a shutdown syscall if possible. This shutdown syscall would likely be called from a currently non existent user terminal task.

Timer

The timer interface has been expanded to work with timers 1, 2, and 3. It provides new functions for enabling and disabling the timers, changing their settings, reading, and clearing interrupts via reading and writing to the relevant memory registers.

AwaitEvent

The implementation of `AwaitEvent` is provided in `await.c` and `await.h`. The functioning is divided into two parts: the system call side and the interrupt side.

Implementation

Inside `await.c` is an array that holds the task ID of the singular task waiting on a particular event. This implementation assumes that only a server listener will ever wait on an event.

The array is initialized to hold all `TASK_INVALID` IDs by a call to `init_event_wait_tid_list` during `kinit`. The array indices correspond to the event id, which are defined, along with the number of events, in `await.h`.

The functions `event_await` and `event_wake` are used by the system call and interrupt sides to complete a call to `AwaitEvent`, respectively. `event_await` returns `-1` if the event ID is not valid and `0` if it is. `event_wake` returns the ID of the task waiting on the specified event, or `TASK_INVALID` if no task was waiting on that event.

System Call Side

A task that calls `AwaitEvent` is put into state `TASK_AWAIT` and has its ID placed on the wait slot for the event ID it requested via a call to `event_await`. If the event ID was not valid, `handle_swi` sets the task back to `TASK_READY`, sets its return value to `-1`, and pushes it back onto the schedule queue. Otherwise, it will remain blocked on the wait list until an interrupt occurs that wakes it up. The interrupt handler is expected to set the return values for all other results.

Interrupt Side

When an interrupt occurs, `handle_interrupt` calls `event_wake` for that interrupt event. If a valid task ID is returned, the handler sets the task's return value, sets it to `TASK_READY`, and pushes it back on the schedule queue before clearing the interrupt.

Clock Server and Clients

The clock server is implemented in much the same way as the name, and RPS servers when it comes to how it parses requests, and how the interface functions wrap message sends. The clock server is communicated with by users by using the `Time`, `Delay`, and `DelayUntil` functions in `clock_server.h`. These are wrappers for 5 byte message `Send` calls to the `clock_server` which store the operation done in the first byte and the integer argument in the last four bytes. Currently, the argument is only used by the `Delay` and `DelayUntil` functions to communicate the time to delay for or until respectively. The four byte integer arguments are stored in and read from the character array messages using the `pack_int` and `unpack_int` functions respectively. The server replies to these `Send` calls with a four byte message in the same manner; it stores an integer representing either the intended return value of the operation or an error code. The task ID of the clock server is obtained by making a `WhoIs` call to the `nameserver` for `"clock_server"`. The `clock_server` is created within the `clock_server_test` task in user.c.

A notable difference from past server implementations is that the `clock_server` creates a new task called `clock_notifer`. This task's job is to infinitely loop, notifying the `clock_server` every 10 milliseconds that a tick has occurred. At the start of every iteration it calls `AwaitEvent` on `EVENT_TIMER1_INTERRUPT` which happens every 10 milliseconds

(20 timer cycles). When the event occurs, it then sends a `TICK_OCCURED` message to the clock server, ending the iteration and repeating.

Unlike the name and RPS servers, the implementations of `Time`, `Delay`, and `DelayUntil`, are implemented directly within a switch in the clock server that parses a received message for the command type and argument. The switch also includes a case for a `TICK_OCCURED` command which does not make use of the integer argument and is received via a `Send` from the `clock_notifier` when a new 10 millisecond tick has occurred (the `clock_notifer` has been interrupted by timer 1).

The clock server is implemented using variables that store the state of who is being delayed and the time elapsed since its inception. The `SortedList waiting` stores all the currently delayed tasks, each element being a node storing the ID of the task being delayed and the time that it is to be delayed until. It stores these nodes sorted in ascending order by the time they are to be delayed until.

The `clock_server` function is the task's entry point. It first registers itself as `"clock_server"` on the name server and then begins the `clock_notifier` task. Afterwards, it initializes the `time_elapsed` variable to `0` and an empty `SortedList waiting` that stores all currently delayed tasks. It then begins an infinite loop which works as follows. At the start of the loop a `Receive` call is made for a message. The message is then parsed, using the value of the first character to pick what the operation is to be done, and the value of the last four characters as an integer argument if the operation requires it. The first character of the message will have been set appropriately by one of the three wrappers or the `clock_notifier`. This parsing is done by a switch statement and the code in each case is the implementation of each operation. If `TICK_OCCURED` or `TIME_ELAPSED` was sent then the tasks are replied to immediately. In the case of `TIME_ELAPSED`, the reply contains the ticks elapsed since the clock servers creation in the last 4 bytes. If the `DELAY` or `DELAY_UNTIL` commands are sent and their argument is non negative, then the sending task's ID is placed on the `waiting` sorted list, and they are not replied to. Additionally, `DELAY` is internally converted into a `DELAY_UNTIL` for timekeeping. If the argument was negative, then they are replied to immediately with an error code in the last 4 bytes of the message. After this switch statement, a while loop replies to all messages at the front of the waiting `SortedList` whose `DELAY_UNTIL` times have passed.

The client tasks used to test the clock server are defined and created in `user.c`. The FirstUserTask mentioned in the assignment description is the `clock_server_test` task found in `user.c`. The phrase "It then executes `Receive()` four times, and replies to each client task in turn." was interpreted to mean that the `FirstUserTask` first calls `Receive` four times in a row and only after all four `Receive` calls have been made does it `Reply` to each client.

Structures Changed

`Frame` has been updated to have `r15` at the top, in correspondence with the updated `enter_user` and `enter_kernel` functions described in the Context Switch section.

Structures Added

SortedList

This structure is similar to the priority queue implemented in K1 and the message queue implemented in K2 in that it uses a similar linked list internally. It's purpose is to keep track of a list of task ID and time pairs that is sorted in ascending order by the time values (smallest times first). These are meant to represent tasks being delayed, and the time which they are supposed to be delayed until. An instance of this list is kept in `clock_server.c`. The interface and it's implementation for this structure can be found in `clock_server.h` and `clock_server.c` respectively.

The interface for the `SortedList` structures includes the following:

`init_sorted_list`: Initializes a `SortedList` passed to it by pointer and must be called before calling any of the following functions on a `SortedList` instance.

`add_item`: Takes a `SortedList`, `tid`, and `time` in ticks and inserts the `tid` and `time` pair into the `SortedList` in the correct place such that the `SortedList` remains sorted. To do this `add_item` first checks the cases where the `SortedList` is empty or the `time` provided is less than the current time. In these cases the `tid` and `time` pair would go at the start of the list. If these are not the case, then it scans through the list until it finds the least element with a time less than or equal to the time provided and places the new element after this one.

`peek_front_tid`: Returns the `tid` at the front of a sorted list. Does not change the list in any way.

`peek_front_time`: Returns the `time` associated with the `tid` at the front of a sorted list. Does not change the list in any way.

`remove_front`: Removes the `tid` and time pair at the front of the sorted list.

This structure contains an array of `list_nodes` called `nodes` that's the size of the maximum number of tasks, since at most all tasks could be delayed. This works as a fixed size allocator, providing nodes the sorted list. All nodes in this list are initially considered to be free, and have their next pointers pointing to their neighbor. Thus, the first node represents the head of a linked list of free nodes. The field `free` holds this head. Finally we have the `list` field which holds a pointer to the head of the sorted list. The sorted list is a singly linked list.

`add_item` and and `remove_front` work by moving nodes between the sorted list and the free list. All nodes sit in the same `nodes` array, but the lists that they represent change. When we `add_item`, a node is taken from the free list and is inserted somewhere in the sorted list. When we `remove_front` the node at the front of the sorted list is removed and becomes the new head of the free list. These two processes represent allocating and deallocating a `list_node` respectively.

Known Bugs

- Since only the lower 32 bits of the debug timer are used for idle timing, the idle percentage will become inaccurate after about 72 minutes of run time when those bits overflow.

Clock Server Output

When running the kernel, it will print the output of four clients created by a `FirstUserTask` interacting with the clock server as outlined in the assignment description.

The following are the outputs of the four clients as they appear in the terminal:

```
IDLE: 92%
umain: exiting
Tid: 6, Delay Interval: 10, Delays completed: 1
Tid: 6, Delay Interval: 10, Delays completed: 2
Tid: 7, Delay Interval: 23, Delays completed: 1
Tid: 6, Delay Interval: 10, Delays completed: 3
Tid: 8, Delay Interval: 33, Delays completed: 1
Tid: 6, Delay Interval: 10, Delays completed: 4
Tid: 7, Delay Interval: 23, Delays completed: 2
Tid: 6, Delay Interval: 10, Delays completed: 5
Tid: 6, Delay Interval: 10, Delays completed: 6
Tid: 8, Delay Interval: 33, Delays completed: 2
Tid: 7, Delay Interval: 23, Delays completed: 3
Tid: 6, Delay Interval: 10, Delays completed: 7
Tid: 9, Delay Interval: 71, Delays completed: 1
Tid: 6, Delay Interval: 10, Delays completed: 8
Tid: 6, Delay Interval: 10, Delays completed: 9
Tid: 7, Delay Interval: 23, Delays completed: 4
Tid: 8, Delay Interval: 33, Delays completed: 3
Tid: 6, Delay Interval: 10, Delays completed: 10
Tid: 6, Delay Interval: 10, Delays completed: 11
Tid: 7, Delay Interval: 23, Delays completed: 5
Tid: 6, Delay Interval: 10, Delays completed: 12
Tid: 6, Delay Interval: 10, Delays completed: 13
Tid: 8, Delay Interval: 33, Delays completed: 4
Tid: 7, Delay Interval: 23, Delays completed: 6
Tid: 6, Delay Interval: 10, Delays completed: 14
Tid: 9, Delay Interval: 71, Delays completed: 2
Tid: 6, Delay Interval: 10, Delays completed: 15
Tid: 6, Delay Interval: 10, Delays completed: 16
Tid: 7, Delay Interval: 23, Delays completed: 7
```

```
Tid: 8, Delay Interval: 33, Delays completed: 5
Tid: 6, Delay Interval: 10, Delays completed: 17
Tid: 6, Delay Interval: 10, Delays completed: 18
Tid: 7, Delay Interval: 23, Delays completed: 8
Tid: 6, Delay Interval: 10, Delays completed: 19
Tid: 8, Delay Interval: 33, Delays completed: 6
Tid: 6, Delay Interval: 10, Delays completed: 20
Tid: 7, Delay Interval: 23, Delays completed: 9
Tid: 9, Delay Interval: 71, Delays completed: 3
Kernel: exiting
```

Here we can see that the processor is idle 92% of the time. This makes sense since the clients spend most of their time delaying.

If we multiply each line's Delay Interval by its Delays Completed we get the total time that the task has spent delaying up to that print. If we do this we get the sequence of times:

```
10, 20, 23, 30, 33, 40, 46, 50, 60, 66, 69, 70, 71, 80, 90, 92,
99, 100, 110, 115, 120, 130, 132, 138, 140, 142, 150, 160, 161,
165, 170, 180, 184, 190, 198, 200, 207, 213
```

This sequence of times makes sense since it is strictly increasing. This indicates that every print is occuring at the correct time relative to the others. The total time that a task has spent delaying at a print indicates a lower bound for the time it should have taken for the print to occur (since the task's creation). Assuming that all the client tasks are created at roughly the same time, this indicates what order the prints should occur in. If the total delay of one print is higher than another's, then the task printing it would have had to spend more time delaying prior to it being printed, and so it makes sense that it's after.