Contents

IRQ Context Switch

Kernel To User Task

When activating a user task, the kernel now checks the TaskDescriptor interrupt flag to determine if it is reloading a SWI or a hardware interrupted task. When reloading an IRQ, kernel state is saved, all registers are restored and `SUBS pc, lr, #4` is called.

User Task to Kernel

The user state is first copied onto the irq stack and then migrated to the user stack. The kernel state is then restored, and the SPSR is read from irq_mode. A special flag is set in the user's TaskDescriptor to note that it has hardware interrupted and should do a full restore (reload r0).

Interrupt Handling

Interrupt Handler

The Interrupt Handler supports the syscall `AwaitEvent` which user tasks can invoke to block waiting for an interrupt and be awoken from the Interrupt Matrix. The kernel will run the `event_register` command to register the task on the interrupt specified by the user task's r0 (subsequently saved to the user stack after context switch).

When a hardware interrupt occurs, the kernel enters via IRQ_mode, notes itself as a hardware interrupt and then runs `event_wake` which does a lookup in the Interrupt Matrix and wakes up all tasks blocking on the highest priority, non-empty interrupt. Any active interrupts with higher priority (which have no waiting tasks) will also be cleared to avoid multiple context switch overhead.

Interrupt initialization and cleanup are handled at the beginning and end of the kernel. Interrupt init creates sets up the Interrupt Matrix, sets the IRQ Mode stack base, and enables interrupts.

Interrupt cleanup diables all interrupts (so we don't have to reset the box after every session).

Note: In the spirit of microkernels, enabling/disabling specific interrupts may be moved to a user task instead.

```
void init_irq( interrupt_matrix *im );
void cleanup_irq();
void event_register( interrupt_matrix *im, TaskDescriptor *td);
void event_wake(interrupt_matrix *im);
```

Interrupt Matrix

The Interrupt Matrix is a data strcture which holds all tasks blocked waiting on an interrupt. The matrix is a `struct` including circular buffers for each interrupt the kernel handles (improve as we go). Currently, the Interrupt Matrix only supports storing and loading tasks for the T3CUI interrupt.

Insertion and head deletion for the task both take O(1) time. However, when waking all tasks for a corresponding interrupt requires O(n) time, for n being the number of tasks waiting on the interrupt (rarely large).

```
void im_init(interrupt_matrix *im);
int im_push(interrupt_matrix *im, TaskDescriptor *task, InterruptEvent ie);
TaskDescriptor *im_top(interrupt_matrix *im, InterruptEvent ie);
int im_pop(interrupt_matrix *im, InterruptEvent ie);
unsigned int im_eventsize(interrupt_matrix *im, InterruptEvent ie);
```

User Task Metrics

IdleTask

The IdleTask serves the purpose of computing kernel metrics as well as a keep alive for hardware interrupts to occur.

The IdleTask measures its running time using the T3Timer and the ClockServer. Every busy loop iteration, the IdleTask compares its last checked time against the ClockServer to the current. If the tick values differ, then a hardware interrupt must have occured.

The IdleTask on every busy loop iteration performs a delta on the T3Timer. If we've yet to be interrupted, sum the delta to the measured session. Once an interupt has occured, sum the remaining delta (prev to 0) to session and sum the session to total runtime. Once an interrupt occurs, the IdleTask asks the IdleTaskInterface for permission to exit.

This method provides a finer granularity than ClockServer ticks as the IdleTask will context switch in and out multiple times in one hardware interrupt. The IdleTask has the ability to measure fractions of a tick using the 502KHz clock (5020 cycles per tick).

Limitation: There will be a margin of error if the IdleTask is interrupted between queries to the clock server. This will result in the IdleTask thinking it has ran longer than it actually has. The margin of error will only ever be a fraction of a tick, and corrects itself when the IdleTask realizes the interrupt has occured.

Note: Running time metrics may be moved to the kernel in the future to provide a more accurate and flexible measurement.

When IdleTask exits, it creates a ClockServerStop and NameServerStop task to stop the ClockServer and NameServer respectively. The kernel will then gracefully shutdown as no tasks exist.

`METRICS GO HERE`

IdleTaskInterface

The IdleTaskInterface serves as a gateway between the state of user tasks and the IdleTask. Since the IdleTask cannot block (unless for a short query), the IdleTaskInterface acts as a server listening to events happening in user land.

The first user task generates the IdleTaskInterface and sends a message about the number of client tasks which should run.

The IdleTask will periodically query the IdleTaskInterface (once every irq) for a request to exit. The IdleTaskInterface will reply `K3_NOT_FINISHED` unless all client tasks have finished.

When client tasks complete, they notify the IdleTaskInterface of their exit. Once all client tasks complete, IdleTaskInterface will respond to the next IdleTask query with `K3_FINISHED`.