Contents

ChoochoOS!

A microkernel RTOS for the TS-7200 Single Board Computer, as used in CS 452 - Real Time Operating Systems at the University of Waterloo.

Written by Daniel Prilik and James Hageman.

Project Structure

Building `choochoos`

This project uses a standard Makefile-based build system.

Building the latest deliverable is as simple as running:

```
make -j
```

Alternatively, this repo also includes several other programs built on top of the core kernel, as found under the `src/assignments/` directory. The executable which is built is decided by the `TARGET` makefile variable, which just happens to be defaulted to the latest deliverable.

e.g: to build `k3`:

```
make -j TARGET=k3
```

The output of the build process is an ELF file in the root of the repository. The name of the elf file is `$(TARGET).elf`.

Navigating the Repo

`choochoos` follows a fairly standard C/C++ project structure, whereby all header files are placed under the `include` directory, with corresponding implementation files placed under the `src` directory.

`src/` and `include/` are separate, the former containing implementation files (`.cc`, `.c` and `.s`), and the latter containing only header files. Both are further subdivided into 3 different folders: `kernel/`, `user/`, and `common/`.

`common/` contains modules that are used throughout the kernel and user space programs, including:

- data structures (`Queue`, `PriorityQueue`, `OptArray`)
- VT100 escape sequences
- The `bwio` library
- `ts7200.h`, which defines many constants and data types specific to the TS-7200 (such register locations and flag masks)

`user/` contains userland specific libraries:

- `syscalls.{h,c}` define the how user tasks interface with the kernel.
- `raw_syscalls.s` defines the user-side implementation of each syscall in ARM assembly.
- `debug.h` implements helper macros such as `assert`, `panic`, and `debug`.
- `clockserver.cc` and `uartserver.cc` implement servers which act as an interface over the hardware.

`kernel/` implements well, the kernel! Some notable places include:

- `kernel.h` which defines the interface of each of the namespaces under `kernel`.
- `kernel/handlers/` contains files that implement the kernel side of each syscall.
- `{irq,swi}_handler.cc` define the C++ side of the hardware and software interrupt handlers.
- `asm.s` defines the assembly side of the kernel, which includes the irq and swi handlers, as well as `_activate_task`
- `kernel.cc` defines the state of the kernel and `run()`, the main scheduling loop.

`assignments/` is a special folder - each subfolder is a seperate userland program, each with its own `FirstUserTask`. Each of the userland programs may be linked with the kernel by running:

```
make -j TARGET=<assignment>
```

Kernel Namespaces

The kernel is organized as a collection of namespaces with a few static state variables. The namespaces are as follows:

- `kernel` defines the kernel's state (`tasks`, `current_task`, `ready_queue`, `event_queue`) and the `run()` function that executes the main scheduling loop.
- `kernel::driver` implements the functions required by `kernel::run()` (`activate_task`, `schedule`, `initialize`, `shutdown`) along with the `swi` and `irq` handlers.
- `kernel::handlers` implements the functions to handle each syscall (`Create`, `MyTid`, `Send`, etc).

- `kernel::perf` keeps the state to track our idle time measurement.
- `kernel::helpers` contains functions used across multiple namespaces.

`kernel.h` defines the public interface of each namespace, but each namespace is implemented accross one or more separate files. Syscall handlers each get their own file (i.e. `src/kernel/handlers/send.cc`), as do the `irq` and `swi` handlers. Data structures such as `Tid` and `TaskDescriptor` are also separated.

Only the global state variables, `kernel::run()` and the `kernel::driver` routines used by `kernel::run()` are defined in `kernel.cc` - everything else is separate.

Kernel Initialization

Our entry point is the `_start` routine defined in `src/kernel/boilerplate/crt1.c`. `_start` does couple key things for the initialization of the kernel:

- The initial link register is stored in the `redboot_return_addr` variable.
  - This variable is used to implement an `_exit()` method, which can be called at any point during kernel execution to immediately return execution back to Redboot. See the "Kernel Shutdown" section for more details on kernel shutdown.
- The BSS is zeroed
- All C++ global constructors are run

`_start` proceeds to call `main`, which then immediately calls the `kernel::run()` method.

The first thing the kernel does is call the `driver::initialize()` method, which performs some critical hardware setup:

- Installs the SWI Handler and IRQ Handler function pointers to the ARM Vector Table
- Unlocks the Halt/Standby magic addresses on the System Controller
- Enables hardware caches
- Programs the VIC
  - Sets all interrupts to IRQ mode
  - Un-masks Timer and UART interrupts on the VIC
  - Enables the VIC protection bits (preventing user mode tasks from touching the VIC)

Additionally, the `driver::initialize()` method spawns the `FirstUserTask` and `nameserver::Task`. As the name implies, the `FirstUserTask` is the main entry point for user code, and is spawned at priority 0. For details on the `nameserver::Task`, and the rationale for spawning it from the Kernel itself, please refer to the "Common User Tasks - Name Server" section.

Main Scheduling Loop

After initialization, the kernel enters it's main scheduling loop. This loop looks roughly as follows:

```
while (true) {
    std::optional<kernel::Tid> next_task = driver::schedule();
    if (next_task.has_value()) {
        driver::activate(next_task.value());
    } else {
        if (driver::num_event_blocked_tasks() == 0) break;
        // start idle measurement
        // ... idle until IRQ ...
        // end idle measurement
        driver::handle_interrupt();
    }
}
```

This loop roughly mirrors the one described in lecture, aside from the fact that `Kernel::activate` does not return a "syscall request" to handle. Instead, the `activate` method implicitly includes syscall handling (see the SWI handler documentation for more details).

Scheduling and the `ready_queue`

`schedule()` has a remarkably simple implementation:

```
namespace kernel::driver {
// ...
std::optional<Tid> schedule() { return ready_queue.pop(); }
// ...
}
```

We have a `ready_queue`, which is a FIFO priority queue of Tids. Tasks that are ready to be executed are on the queue, and `schedule()` simply grabs the next one.

Our priority queue is implemented as a template class in include/priority_queue.h. A `PriorityQueue<T, N>` is a fixed size object containing up to `N` elements of type `T`, implemented as a modified binary max-heap. Usually, when implementing a priority queue as a binary heap, the elements within the heap are compared only by their priority, to ensure that elements of higher priority are popped first. However, only using an element's priority does not guarantee FIFO ordering *within* a priority.

Instead, we extend each element in the priority queue to have a `ticket` counter. As elements are pushed onto the queue, they are assigned a monotonically increasing `ticket`. When elements with the same priority are compared, the element with the lower `ticket` is given priority over the element with the higher `ticket`. This has the effect of always prioritizing elements that were added to the queue first, and gives us the desired per-priority FIFO.

Using this ticketed binary heap has both drawbacks and benefits over a fixed-priority implementation (be that a vector of queues, or an intrusive linked list per priority). The benefit is that we can permit `p` priorities in `O(1)` space instead of `O(p)` space. This allows us to define priorities as the entire range of `int`, and will allow us to potentially reliable arithmetic on priorities in the future (however, whether or not we'll need such a thing is yet to be discovered).

The drawback is that we lose FIFO if the ticket counter overflows. Right now, the ticket counter is a `size_t` (a 32-bit unsigned integer), so we would have to push `2^32` Tids onto `ready_queue` in order to see an overflow. However, the implications of an overflow are relatively minor - one task may be scheduled out of order with respect to priority. This is a minor concern as user tasks should consider priorities to be more scheduling "advice" for the kernel instead of a hard guarantee.

Idle Task

When all user tasks are blocked waiting for IRQs, our kernel enters an idle state. While earlier iterations of our kernel used a busy-looping idle task implementation, our final version of the kernel uses the TS-7200's low-power Halt state to efficiently wait for interrupts.

During kernel initialization, we enable the Halt/Standby magic addresses in the System Controller. This is done in two simple lines of code:

- `*(volatile uint32_t*)(SYSCON_SWLOCK) = 0xaa;`
- `*(volatile uint32_t*)(SYSCON_DEVICECFG) |= 1;`

With this setup out of the way, the entire system can be put into the low power Halt mode by reading from the System Controller's Halt register (i.e: `*(volatile uint32_t*)(SYSCON_HALT);`).

As it turns out, using the System Controller to halt the CPU has a very useful side effect: the CPU does *not* need to have it's IRQs enabled for the System Controller to wake it up! This means that we can leave the CPU in the regular IRQ-disabled Kernel Mode before entering the Halt state, saving us from having to do any mode-switching assembly shenanigans. When an IRQ eventually arrives, the System Controller simply "unhalts" the CPU, which resumes executing whatever instructions come after the `SYSCON_HALT` read. Critically, the CPU does *not* jump to the global IRQ handler, which would be quite dangerous, given that the IRQ handler is typically only entered when a user task is preempted.

Thus, the entire "idle task" logic is as simple as:

```
if (/* there is nothing to schedule */) {
    // read the current time before going idle
    idle_timer = *TIMER3_VAL;
    // go idle
    *(volatile uint32_t*)(SYSCON_HALT);
    // update idle time counter after waking up
    idle_time::counter += idle_timer - *TIMER3_VAL;
    // handle the IRQ that just woke the CPU up
    driver::handle_interrupt();
}
```

Context Switching

Preface: Task State

In our kernel, we've opted to store task state directly on the task's stack. This greatly simplifies task preemption and activation, as user state can be saved/stored using ARM's standard `stmfd` and `ldmfd` methods.

The following diagram describes the layout of a user task's stack after it has been preempted / right before it is activated:

```
+----------- hi mem -----------+
| ... rest of user's stack ... |
|    [ lr                 ]    | /
|    [ r0                 ]    | | Saved General Purpose
|    ...                       | | User registers
|    [ r12                ]    | \
|    [ ret addr           ]    | - PC to return execution to
|    [ spsr               ]    | - saved spsr
|         <--- sp --->         |
| ....... unused stack ....... |
+----------- lo mem -----------+
```

Alternatively, it may be useful to think of a user's saved stack pointer as a pointer to the following C-structure:

```
struct UserStack {
    uint32_t spsr;
    void* ret_addr;
    uint32_t regs[13];
    void* lr;
    uint32_t additional_params[];
};
```

Please keep this structure in mind when referencing the following sections on task activation and preemption.

Task Activation

Once the scheduler has returned a Tid, the `driver::activate` method is called with the Tid. This method updates the kernel's `current_task` with the provided Tid, fetch the task's saved stack pointer from its Task Descriptor, and hands the stack pointer off to the `_activate_task` assembly routine.

This routine is quite simple, and reproduced here in it's entirety:

```
// src/kernel/asm.s

// void* _activate_task(void* next_sp)
_activate_task:
  stmfd   sp!,{r4-r12,lr} // save the kernel's context
  ldmfd   r0!,{r1,r2}     // pop ret addr, spsr from user stack
  msr     spsr,r1         // set spsr to the saved spsr
  stmfd   sp!,{r2}        // push ret val onto kernel stack
  msr     cpsr_c, #0xdf   // switch to System mode
  mov     sp,r0           // install user sp
  ldmfd   sp!,{r0-r12,lr} // restore user registers from sp
  msr     cpsr_c, #0xd3   // switch to Supervisor mode
  ldmfd   sp!,{pc}^       // pop ret addr into pc, `^` updates cpsr
```

The task will then continue its execution until it executes a syscall, or is preempted by a interrupt.

Context Switching - SWI Handling

Invoking a syscall switches the CPU to Supervisor mode, and jumps execution to the SWI handler. Our SWI handler is written entirely in Assembly, and was given the extremely original name `_swi_handler`.

This routine is a bit more involved than the `_activate_task` routine, mainly due to some tricky register / system-mode juggling. Nonetheless, it is quite short (and heavily commented). It is reproduced here in it's entirety:

```
_swi_handler:
  msr     cpsr_c, #0xdf     // switch to System mode
  stmfd   sp!,{r0-r12,lr}   // save user context to user stack
  mov     r4,sp             // hold on to user sp...
  msr     cpsr_c, #0xd3     // switch to Supervisor mode
  mrs     r0,spsr           // get user mode spsr
  stmfd   r4!,{r0, lr}      // ...and stash saved spsr + user return addr
  ldr     r0,[lr, #-4]      // load the last-executed SWI instr into r0...
  bic     r0,r0,#0xff000000 // ...and mask off the top 8 bits to get SWI no
  mov     r1,r4             // copy user sp into r1
  bl      handle_syscall    // call handler method (implemented in C)
  // void handle_syscall(int syscall_no, void* user_sp);
  mov     r0, r4            // return final user SP via r0
  ldmfd   sp!,{r4-r12,pc}   // restore the kernel's context
  // execution returns to caller of _activate_task
```

Execution is returned back to the `driver::activate` method, which proceeds to check to the state of last-executed task. If it's READY, it is queued up to be re-scheduled READY. This completes a single context switch, and execution flow is returned back to the scheduler.

The `swi_handler` routine calls the `handle_syscall` with a task's stack pointer and syscall number. This method is implemented as a small C-stub, which immediately calls the true handle_syscall method (implemented in C++): `driver::handle_syscall()`. At a high level, this method looks roughly as follows:

```
void handle_syscall(uint32_t no, UserStack* user_sp) {
    std::optional<int> ret = std::nullopt;
    using namespace handlers;
    switch (no) {
        case 0: ret = SpecificSyscallHandler(user_sp->regs[0], ...); break;
        case 1: // ...
        // ...
        default: kpanic("invalid syscall %lu", no);
    }

    if (ret.has_value()) {
        user_sp->regs[0] = ret.value();
    }
}
```

Notice how useful the `UserStack*` view is in this case:

- Marshalling arguments to individual syscalls is as simple as accessing struct fields
- Writing a return value to the user task is a simple write to the `regs[0]` field

Please refer to the "Syscall Implementations" section of this documentation for details on the concrete implementation of the various syscall handlers.

Context Switching - IRQ Handling

Upon startup, the kernel's `initialize` method configures the VIC to trigger IRQs, and installs the new `_irq_handler` function as the ARM920T's interrupt handler. See the "Kernel Initialization" section for more details.

The `_irq_handler` is extremely similar to the `_swi_handler` function, with a few minor differences:

- Instead of switching to supervisor mode to retrieve the user's return address, the handler switches to IRQ mode instead.

- The IRQ handler omits any code related to reading the immediate value of the last executed user instruction.
- The IRQ handler adjusts the user's return address by `-4`.

When the `_irq_handler` is invoked, the task's context is saved, and execution is switched back to the kernel. `_irq_handler` proceeds to call the `handle_interrupt` method (implemented in C), which then trampolines execution to the kernel's `handle_interrupt` member method.

The `handle_interrupt` method is responsible for unblocking any event-blocked tasks back to the `ready_queue` (see the "Syscall Implementations - AwaitEvent" documentation for more details), and performing any interrupt specific logic (as documented in the subsequent subsections).

Once `handle_interrupt` completes, it returns back to the `_irq_handler`, which eventually yields execution back to the kernel's main scheduling loop.

We've taken care to ensure that the layout of the user's stack is identical between the `_swi_handler` and the `_irq_handler`, which greatly simplifies the implementation of `_activate_task`. Our kernel doesn't have to perform any additional bookkeeping to determine how exactly a task returned to the kernel.

Clock IRQs

Clock interrupts are supported for the 3 hardware timers as events `4` (timer 1), `5` (timer 2) and `51` (timer 3).

Whenever any of the 3 timers generates an interrupt, the kernel automatically writes to the corresponding timer's clear register, which prevents the interrupt from firing again once a the kernel switches back into user mode. No meaningful volatile data is returned from the `AwaitEvent()` syscall for Clock IRQs.

User tasks that wish to receive timer interrupts must properly configure the timers themselves via the timer control registers.

UART IRQs

The UART combined interrupts are supported in the kernel as events `52` for UART1 and `54` for UART2. UART interrupts are enabled in user space by writing to the UART's CTLR (control register). For ease of use, a the following struct is defined in `ts7200.h`:

```
union UARTCtrl {
    uint32_t raw;
    struct {
        bool uart_enable : 1;
        bool sir_enable : 1;
        bool sir_low_power : 1;
        bool enable_int_modem : 1;
        bool enable_int_rx : 1;
        bool enable_int_tx : 1;
        bool enable_int_rx_timeout : 1;
        bool loopback_enable : 1;
    } _;
};
```

User tasks may manipulate the control register by creating a `UARTCtrl` struct, setting `.raw` to the register's data, modifying the bitfields in `._`, and writing `.raw` back to the register. By default, only the `uart_enable` bit is set. Interrupts are enabled via the `enable_int_*` bits.

When UART interrupts are enabled, `AwaitEvent(52|54)` will block the calling task until the combined UART interrupt is raised. These calls to `AwaitEvent` will return the UART's INTR (interrupt register) as an `int`. Again we define a struct for ease of use:

```
union UARTIntIDIntClr {
    uint32_t raw;
    struct {
        bool modem : 1;
        bool rx : 1;
        bool tx : 1;
        bool rx_timeout : 1;
    } _;
};
```

The fields in `._` correspond to each of the possible combined interrupts. Whenever an interrupt is raised, the kernel will automatically disable that interrupt on the UART's control register to prevent unwanted reentry. Therefore, is is up the user task to re-enable each interrupt as required.

Syscall Implementations

`int MyTid()`

`MyTid()` returns the kernel's `current_tid`, which is updated to the most recent Tid whenever as task is activated.

`int MyParentTid()`

`MyParentTid()` looks up the `TaskDescriptor` by `current_tid`, which holds the parent Tid. (The parent Tid is set to the value of `current_tid` when a task is created). When called from `FirstUserTask`, `MyParentTid()` returns `-1`.

`int Create(int priority, void (*function)())`

`Create(priority, function)` determines the lowest free Tid, and constructs a task descriptor in `tasks[tid]`. The task descriptor is assigned a stack pointer, and the user stack is initialized by casting the stack pointer to a FreshUserStack*, and writing to that struct's fields. Notably, we write stack->lr = (void*)User::Exit, which sets the return address of function to be the Exit() syscall, allowing the user to omit the final Exit(). The task's parent_tid is the current value of MyTid().

void Yield()

Yield() puts a task back on the ready queue, allowing higher priority tasks to be run. The syscall implementation itself does nothing - since it leaves the calling task in the READY state, the task will be re-added to the ready_queue in activate().

void Exit()

Exit() clears the task descriptor at tasks[MyTid()]. This prevents the task from being rescheduled in activate(), and allows the Tid to be recycled in future calls to Create().

Send-Receive-Reply

Send(..), Receive(..) and Reply(..) are complicated syscalls, and are best described together.

Send-Receive-Reply is implemented via a TaskState state; field in TaskDescriptor. TaskState is a tagged union with the following structure:

```
struct TaskState {
    enum uint8_t { READY, SEND_WAIT, RECV_WAIT, REPLY_WAIT, EVENT_WAIT } tag;
    union {
        struct {
        } ready;
        struct {
            const char* msg;
            size_t msglen;
            char* reply;
            size_t rplen;
            std::optional<Tid> next;
        } send_wait;
        struct {
            int* tid;
            char* recv_buf;
            size_t len;
        } recv_wait;
        struct {
            char* reply;
            size_t rplen;
        } reply_wait;
        struct {
        } event_wait;
    };
};
```

So each `TaskDescriptor` can be in any of the states described by `tag`, at which point the fields in the corresponding struct under the `union` will be populated:

- `READY`: the task is on the `ready_queue`, waiting to be scheduled.
- `SEND_WAIT`: the task is waiting to `Send()` to another task that hasn't called `Receive()` yet.
- `RECV_WAIT`: the task has called `Receive()`, but no task has sent a message to it yet.
- `REPLY_WAIT`: the task called `Send()` and the receiver got the message via `Receive()`, but no other task has called `Reply()` back at the task.
- `EVENT_WAIT`: the task is blocked waiting on an interrupt. This is explained in more detail in the `AwaitEvent()` section below.

To implement these state transitions, each task has a `send_queue` of tasks that are waiting to send to it (and must therefore be in the `SEND_WAIT` state). This send queue is built as an intrusive linked list, where the `next` "pointer" is actually just another Tid, or `-1` to represent the end of the list. When a task wants to `Send()` to another task that is not in `RECV_WAIT`, it will be put on the receiver's send queue (denoted by the fields `send_queue_head` and `send_queue_tail`, also Tids), and the sending task will be put in `SEND_WAIT`. Note that the `send_wait` branch of the union contains everything that the sender passed to `Send()`, plus the `next` pointer, if any other task get added to that same send queue.

If a task is in `SEND_WAIT`, it can only be blocked sending to one receiving task, so we only need at most one `next` pointer per task. The result is very memory efficient: by storing one word on each task descriptor, we can support send queues up to length `MAX_SCHEDULED_TASKS - 1`.

Let's step through the two possibilities for an SRR transaction: sender-first and receiver-first.

Sender-first

If a sender arrives first, that means the receiver is not in `RECV_WAIT`, and therefore does not yet have a `recv_buf` to be filled. If this is the case, we add the sender to the end of the receiver's send queue, and put the sender in `SEND_WAIT`.

When the receiver finally calls `Receive()`, it will see that is has a non-empty send queue. So it will pop the first `TaskDescriptor` off the front of its send queue, copy the `msg` into `recv_buf`, and transition the sender into `REPLY_WAIT`, using the same `char* reply` that the sender saved when they went into `SEND_WAIT`.

Receiver-first

If the receiver arrives first, it checks its send queue. If the send queue is non-empty, then we follow the same procedure as sender-first, and the new sender is placed on the send of the receiver's send queue. If the send queue is empty, the receiver goes into `RECV_WAIT`, holding onto the `recv_buf` to be copied into once a sender arrives.

When the sender arrives, it notices that the receiver is in `RECV_WAIT`, so it writes the `msg` directly into `recv_buf`, wakes up the receiver (transitioning it to `READY` and pushing it onto the `ready_queue`), and goes into `REPLY_WAIT`.

`int Reply(int tid, char* reply, int rplen)`

`Reply(tid, reply, rplen)` only delivers the reply if the task identified by `tid` is in `REPLY_WAIT`. The kernel can then immediately copy the `reply` into the `char* reply` stored in the `reply_wait` branch of the task's `state` union. This way, reply never blocks - it only returns an error code if the task is not in `REPLY_WAIT`.

Error cases

The SRR syscalls return two possible error codes: `-1` and `-2`. `-1` is returned whenever a `tid` does not represent a task - either it is out of range, or it points to a TaskDescriptor in the `UNUSED` state. `-2` is returned in two cases where a SRR transaction cannot be completed:

- If a task tries to `Reply()`to a task that is not in `REPLY_WAIT` - this would mean that the task never called `Send()` and thus is not expecting a reply.
- If a task is in `SEND_WAIT` in a receiver's send queue, and the receiver exits.

When a receiver exits, if it has a non-empty send queue, those tasks will never get a reply because they won't make the transition into `REPLY_WAIT`, and thus would never wake up. Instead, we iterate through the send queue, waking every `SEND_WAIT` task up, and writing the syscall return value of `-2`.

`int AwaitEvent(int eventid)`

`int AwaitEvent(int eventid)` blocks the calling task until the interrupt identified by `event_id` is fired. The returned integer is the interrupt's "volatile data", which is specific to each interrupt.

The kernel contains an `event_queue`, which is an array of 64 `std::optional TidOrVolatileData`s. `TidOrVolatileData` is a `std::variant<Tid, VolatileData>`, representing two possible states for an interrupt: a task is blocked waiting for the interrupt data, or the interrupt data arrived and no task has asked for the data yet.

When a task wants to wait for an interrupt, it calls `AwaitEvent(int eventid)`, where `eventid` is a number between 0 and 63, corresponding to the interrupt of the same number. The kernel records the `Tid` in the `event_queue` array. In `handle_interrupt`, if the `event_queue` has a task blocked on the interrupt, the Tid is removes from `event_queue`, and the task is moved back to the `ready_queue`. The preempted task remains ready, and thus is put back on the `ready_queue` as well.

If an interrupt arrives before any tasks has started waiting for it, it is stored in the `VolatileData` case of the variant. When a task calls `AwaitEvent`, if the data is already stored in `event_queue`, the task is immediately rescheduled with the data, and the entry is removed from the `event_queue`.

Multiple tasks waiting on the same event is not currently supported, simply because the clock and UART servers do not require such functionality.

Additional syscalls

To simplify the user programs that are to be written on our kernel, we implemented two additional syscalls that are not required by the CS452 specification: `Perf` and `Shutdown`.

`Perf(struct perf_t*)`

We decided that we needed a mechanism whereby user tasks could query the kernel for system performance information. As such, we decided to introduce a new syscall to our kernel: `Perf`.

`Perf` uses syscall number `9` and has the following C signature: `void Perf(struct perf_t*)`, where `struct perf_t` contains all sorts of useful system performance information. At the moment, `struct perf_t` only contains a single field: `uint32_t idle_time_pct`, which as the name implies, is the current idle time percentage of the system.

With this new Perf syscall, individual userland programs to are able to query and display system performance metrics whenever and however they like!

Each call to `Perf` resets the idle time measurement, providing a form of “windowed” idle time measurement, where the window size depends on the frequency at which `Perf` is called.

`Shutdown()`

Userland programs often make liberal use of `assert` and `panic(const char* fmt, ...)` to notify users if any of their invariants are broken. These macros could be implemented with the regular old `Exit` syscall, which would terminate an individual task if something went wrong. This would be *okay*, but it typically results in users having to manually restart the TS-7200, as their programs would most likely hang once an assertion / panic fired.

At the same time, we found that as programs became more complex, and spawned more and more long-lived tasks, it became quite difficult to trigger a kernel exit (i.e: by exiting all running tasks).

A clean solution to both these problems was to introduce a new syscall: `Shutdown`.

`Shutdown` uses syscall number `10` and has the following C signature: `void Shutdown(void) __attribute__((noreturn))`.

The kernel's Shutdown handler routine is incredibly simple, simply invoking `kexit`, which triggers the kernel's shutdown sequence, and eventually returns execution back to the Redboot.

The `assert` and `panic` routines to use `Shutdown` instead of `Exit` - and therefore are able to halt the entire system.

Kernel Shutdown

All good things must come to an end, which means that at some point, `ChoochoOS` may need to be shutdown. This can happen for one of three reasons:

1. No running tasks

- i.e: all user tasks have `Exit`ed
- *Note:* this includes terminating the Name Server (via it's Shutdown method)

2. The `Shutdown` syscall

- A user task has notified the kernel that for whatever reason, all of userland should be terminated

3. `kpanic` or `kassert`

- A critical kernel invariant has been broken, and the kernel was not able to recover
- (This should *not* happen during normal kernel operation. Please notify the nearest programmer if this happens to you)

Regardless as to *why* the kernel has decided to shut down, the kernel must call the `driver::shutdown` method at some point prior to returning to Redboot. This method mirrors the `driver::initialize` method, except instead of enabling interrupts, `driver::shutdown` *disables* all the interrupts on the VIC, and asserts any remaining interrupts. If this routine isn't run, then the next time the kernel is re-loaded, it will immediately jump to the irq_handler, even before it's had a chance to initialize. This is Very Bad, as the IRQ handler expects the kernel to already have been initialized, and would result in undefined and broken behavior.

`kexit`

While not strictly required, it's very useful to have a simple method which can be called from anywhere in the kernel to immediately return to Redboot. That method is called `kexit`, and it's the backbone of the `Shutdown` syscalls, and the `kpanic` and `kassert` facilities.

Given the non-linear execution flow required to exit the kernel from anywhere in the call stack, the `kexit` method is implemented using some inline assembly magic.

In the `ctr1.c` file, the initial link register which points to Redboot is stashed away in a global variable. This variable is later referenced in the `_exit` method, which uses some inline assembly to set the PC to the stashed link register. Note that the `_exit` method does *not* call the critical `driver::sutdown` routine.

`kexit` is a simple wrapper around `_exit`, with the added functionality of first calling `driver::Shutdown`.

System Limitations

Our linker script, ts7200_redboot.ld, defines our allocation of memory. Most notably, we define the range of memory space allocated to user stacks from `__USER_STACKS_START__` to `__USER_STACKS_END__`. Each user task is given 256KiB of stack space, so the maximum number of concurrent tasks in the system is `(__USER_STACKS_END__ - __USER_STACKS_START__) / (256 * 1024)`. However, given the variable size of our BSS and data sections (as we change code), we can't compute the optimal number of concurrent tasks until link time. So instead, we hard-code a `MAX_SCHEDULED_TASKS`, and assert at runtime that no task could possibly have a stack outside of our memory space. Currently, this value is set to 48 tasks. The kernel is given 512KiB of stack space. These numbers are quite arbitrary, and subject to change if it turns out we require additional user tasks / user tasks are using too much RAM.

Common User Tasks

Name Server

The Name Server task provides tasks with a simple to use mechanism to discover one another's Tids. It's implementation can be found under `<src/include>/kernel/tasks/nameserver.<cc/h>`

Public Interface

We've decided to use an incredibly simple closure mechanism for determining the Tid of the name server: It's Tid must *always* be 1. To enforce this invariant, we have the kernel spawn the NameServer immediately after spawning the `FirstUserTask`.

To make working with the name server easier, instead of having tasks communicate with it directly via Send-Receive-Reply syscalls, a pair of utility methods are provided which streamline the registration and query flows:

- `int WhoIs(const char*)`: returns the Tid of a task with a given name, `-1` if the name server task is not initialized, or `-2` if there was no registered task with the given name.
- `int RegisterAs()`: returns `0` on success, `-1` if the name server task is not initialized, or `-2` if there was an error during name registration (e.g: Name Server is at capacity.)
  - *Note:* as described later, the Name Server actually panics if it runs out of space, so user tasks should never actually get back `-2`.

These two methods are also exposed as `extern "C"` methods, with corresponding definitions in the `include/user/syscalls.h` file.

Associating Strings with Tids

The core of any name server is some sort of associative data structure to associate strings to Tids.

While there are many different data structures that fit this requirement, ranging from Tries, Hash Maps, BTree Maps, etc. . . , we've decided to use a simple, albeit potentially inefficient data structure instead: a plain old fixed-length array of ("String", Tid) pairs. This simple data structure provides O(1) registration, and O(n) lookup, which shouldn't be too bad, as we assume that most user applications won't require too many named tasks (as reflected in the specification's omission of any sort of "de-registration" functionality).

Efficiently Storing Names

Note that we've put the term "String" in quotes. This is because instead of using a fixed-size char buffer for each pair, we instead allocate strings via a separate data structure: the `StringArena`.

`StringArena` is a simple class which contains a fixed-size `char` buffer, and a index to the tail of the buffer. It exposes two methods:

- `size_t StringArena::add(const char* s, const size_t n)`: Copy a string of length `n` into the arena, returning a handle to the string's location within the arena (represented by a `size_t`). This handle can then be passed to the second method on the arena. . .
- `const char* StringArena::get(const size_t handle)`: Return a pointer to a string associated with the given handle, or `nullptr` if the handle is invalid.

Whenever `add` is called, the string is copied into the arena's fixed-size internal buffer, incrementing the tail-index to point past the end of the string. `get` simply returns a pointer into the char buffer associated with the returned handle (which at the moment, is simply an index into the char array).

The `StringArena` approach allows us to avoid having to put an explicit limit on the size of name strings, as strings of varying lengths can be "packed" together in the single char buffer.

Incoming and Outgoing Messages

The `WhoIs` and `RegisterAs` functions abstract over the name server's message interface, which is comprised of two tagged unions: `Request` and `Response`.

```
enum class MessageKind : size_t { WhoIs, RegisterAs, Shutdown };

struct Request {
    MessageKind kind;
    union {
        struct {
        } shutdown;
        struct {
            char name[NAMESERVER_MAX_NAME_LEN];
            size_t len;
        } who_is;
        struct {
            char name[NAMESERVER_MAX_NAME_LEN];
            size_t len;
            int tid;
        } register_as;
    };
};

struct Response {
    MessageKind kind;
    union {
        struct {
        } shutdown;
        struct {
            bool success;
            int tid;
        } who_is;
        struct {
            bool success;
        } register_as;
    };
};
```

There are 3 types of request, each with a corresponding response:

- `Shutdown` - terminate the name server task
- `WhoIs` - Return the Tid associated with a given name
- `RegisterAs` - Register a Tid with a given name

The latter two messages return a non-empty response, indicating if the operation was successful, and for `WhoIs`, a Tid (if one was found).

The server uses a standard message handling loop, whereby the body of the server task is an infinite loop, which continuously waits for incoming messages, switches on their type, and handles them accordingly.

Clock Server

The clock server allows tasks to be delayed, and query the current time with 10-millisecond resolution. Here is the public interface:

```
namespace Clock {
void Server();

int Time(int tid);
int Delay(int tid, int ticks);
int DelayUntil(int tid, int ticks);
void Shutdown(int tid);

extern const char* SERVER_ID;

}  // namespace Clock
```

The clock server is implemented using one timer and two tasks: the server and notifier. Timer 2 is set to fire interrupts every 10 milliseconds, to correspond to a kernel "tick". The clock server starts up the notifier, and then follows the usual server pattern: a `Receive()` loop that never blocks.

The notifier runs a simple `AwaitEvent(5)` loop (interrupt 5 is for Timer 2 underflow), and then `Send`s to the clock server. The clock server tracks `current_time`, which is incremented every time the notifier sends a tick. `Time()` is implemented by simply replying with `current_time`. `Delay()` and `DelayUntil()` are a bit more involved.

The clock server has a priority queue of delayed tasks (called `pq`), where tasks that want to wake up sooner have higher priority. When `Delay()` and `DelayUntil()` are called, the Tid of the caller is pushed onto the priority queue with `tick_threshold` set to the time that the task would like to wake up. When the notifier ticks, it pops tasks off `pq` until `pq.peek()->tick_threshold > current_time`, and replies to them, waking them up.

`Clock::Shutdown` is an extension to the spec, which cleanly exits the clock server and notifier. The notifier shuts down if the clock server replies with `shutdown=true` when the notifier sends a tick. Clean shutdown functionality is useful for tests, which are expected to terminate quickly.

UART Server

The UART server is implemented as a single server task for both UARTs, and one notifier task for each UART. The notifier tasks simply `AwaitEvent(<irq_no>)` in a loop, `Send()`ing the interrupt data to the UART server. The server follows the conventional pattern of a `Receive()` loop that never blocks. The exposed interface contains the required `Putc` and `Getc` functions, along with a few other helpful routines:

```
namespace Uart {
extern const char* SERVER_ID;
void Server();
int Getc(int tid, int channel);
int Putc(int tid, int channel, char c);

// Getn blocks until n bytes are received from the UART, writing the bytes to
// buf.
int Getn(int tid, int channel, size_t n, char* buf);

// Putstr and Printf can atomically write up to 4096 bytes to the UART in a
// single call.
int Putstr(int tid, int channel, const char* msg);
int Printf(int tid, int channel, const char* format, ...)
    __attribute__((format(printf, 3, 4)));

// Drain empties the RX FIFO for the given channel.
void Drain(int tid, int channel);

// Getline reads a line of input from the uart into `line`, treating the
// backspace and enter keys appropriately.
void Getline(int tid, int channel, char* line, size_t len);
}  // namespace Uart
```

Notably, `Putstr` and `Printf` atomically write an entire string to the UART (up to a maximum length of 4096 bytes), and `Getn` blocks until `n` bytes are received from the UART. Each of the output routines are written in terms of `Putstr`, and similarly, each of the input routines are written in terms of `Getn`.

The UART server has two key pieces of state: output buffers (for TX) and blocked tasks (for RX). The output buffers are a 4096-byte ring buffer per UART.

When handling a `Putstr`, data is written to the UART until the FIFO fills up, at which point further data is pushed onto the output ring buffer and the `tx` interrupt is enabled. Once the tx interrupt is received, data is pulled off the ring buffer and written to the UART's data register until either the buffer is emptied or the FIFO is filled (in which case `tx` interrupts will be enabled again). `Putstr` replies to the sender immediately, ensuring only that the string has been buffered, not flushed.

`Getn` is handled by putting storing a "blocked task" per UART. When a task calls `Getn` (or `Getc`, which is just a special case of `Getn`), a blocked task entry is created for the UART, which contains a buffer to store bytes, along with a counter of how many bytes have been received and the task's Tid. The UART server then reads bytes off the UART's RX FIFO until either the FIFO is empty (in which case the `rx` and `rx_timeout` interrupts are enabled) or the desired `n` bytes have been received (in which case a reply is sent and the "blocked task" is cleared).

Only one reading task per UART is supported, as concurrent readers leads to confusing behavior - should received bytes be sent to both tasks concurrently? In a round robin? Or in sequential order? The semantics of multi-reader is confusing, and likely unnecessary for any applications that will be built on the kernel. So "bocked tasks" are implemented as simple `std::optional`s per UART, and concurrent reads to the same UART will panic.

`Drain` is a notable addition to the specification - it clears any bytes that may be sitting in the RX FIFO and any blocked task's buffer. This is useful when we know that certain bytes are expected to be received after a certain event. Calling `Drain` ensures that all received bytes must have been received after the `Drain` call, which is useful when doing request-response style calls over the UART.