Contents

The `choochoos` Kernel

Written by Daniel Prilik (dprilik - 20604934) and James Hageman (jdhagema - 20604974).

Project Structure

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.

At the moment, both the `choochoos` kernel and userspace live under the same `include` and `src` directory, though we plan to separate kernel-specific files and userspace-specific files at some point.

Building `choochoos`

For details on building the project, see the README.md at the root of this repository.

Architectural Overview

Our entry point is not actually `main`, but `_start` defined in src/boilerplate/crt1.c. `_start` does couple key things for the initialization of the kernel: - The link register is stored in the `redboot_return_addr` variable, so we can exit from any point in the kernel's execution by jumping directly to `redboot_return_addr` - The BSS is zeroed - All global variable constructors are run

Then we jump into `main`, which configures the `COM2` UART and jumps to `kmain`.

`kmain` executes our scheduling loop, using a singleton instance of a class called `Kernel`:

```
static kernel::Kernel kern;

int kmain() {
    kprintf("Hello from the choochoos kernel!");

    kern.initialize(FirstUserTask);

    while (true) {
        int next_task = kern.schedule();
        if (next_task < 0) break;
        kern.activate(next_task);
    }

    kprintf("Goodbye from choochoos kernel!");

    return 0;
}
```

The scheduling loop differs from the one described in lecture only in that

`Kernel::activate` does not return a syscall request to handle. Instead, activate only returns after the syscall has been handled. Nonetheless, the kernel can be broken down into three main parts: scheduling, context switching, and syscall handling.

Scheduling

`schedule()` has a remarkably simple implementation:

```
int schedule() {
    int tid;
    if (ready_queue.pop(tid) == PriorityQueueErr::EMPTY) return -1;
    return tid;
}
```

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` `Tid`s onto `ready_queue` in order to see an overflow. This definitely won't happen in our `k1` demo, but it doesn't seem impossible in a longer running program that is rapidly switching tasks. We're experimenting with using the non-native `uint64_t`, and we will profile such a change as we wrap up `k2`.

Context Switching - Task Activation

Once the scheduler has returned a Tid, the `kern.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 hand the stack pointer to the `_activate_task` Assembly routine.

This assembly routine performs the following steps in sequence: - Saves the kernel's register context onto the kernel's stack (i.e: r4-r12,lr) - r0-r3 are not saved, as per the ARM C-ABI calling convention - *Note:* this does mean that in the future, when implementing interrupt-handling into our kernel, this routine will either have to be modified and/or supplemented by a second routine which saves / restores the entirety of a user's registers. - Switches to System mode, banking in the last user tasks's SP and LR - Moves the new task's SP from r0 to SP - Pops the saved user context from the SP into the CPU - Pops the SWI return value into r0, the SWI return address into r1, and the user SPSR into r2 - Registers r4-12 and LR are restored via a single ldmfd instruction (r0 through r3 are *not* restored, as the ARM C ABI doesn't require preserving those registers between method calls) - Moves the saved SPSR from r2 into the SPSR register - Calls the `movs` instruction to jump execution back to the saved swi return address (stored in r1), updating the CPSR register with the just-updated SPSR value

The task will then continue its execution until it performs a syscall, at which point the "second-half" of context-switching is triggered.

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, but in a nutshell, the `_swi_handler` routine preforms the following steps in sequence: - Save the user's r0 through r12 on the user's stack (requires temporarily switching back to System mode from supervisor mode) - Keep a copy of the user's SP in r4, which is used to stack additional data onto the user stack - Save the SWI return address & user SPSR from the *Supervisor* mode LR and SPSR onto the *User* mode SP (using the SP saved in r4) - Use the SWI return address offset by -4 to retrieve the SWI instruction that was executed, and perform a bitmask to retrieve it's immediate value (i.e: the syscall number) - Call the `handle_syscall` routine (implemented in C) with the syscall number and the user's SP - See the "Handling Syscalls" section below for more details - Push the `handle_syscall` return value onto the user stack - Restore the saved kernel context from the kernel stack, returning execution back to the instruction immediately after the most recently executed `_activate_task` call (i.e: back to the `kern.activate()` method)

The `kern.activate()` method proceeds to check to see what state the last-executed task is in, and queues it up to be re-scheduled if it's READY. This completes a single context switch, and execution flow is returned back to the scheduler.

Handling Syscalls

`handle_syscall` switches on the syscall number, calling the corresponing method on the `Kernel` instance. The user's stack pointer is cast to a `SwiUserStack*`, which allows the user's registers (and the rest of their stack) to be read by our C++ code. This allows us to pass parameters to our C++ syscall handlers. Let's enumerate the handlers:

- `MyTid()` returns the kernel's `current_tid`, which is updated to the most recent Tid whenever as task is activated.
- `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()`.
- `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
- `Yield()` does nothing. Since it leaves the calling task in the `READY` state, the task will be re-added to the `ready_queue` in `activate()`.
- `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()`.

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 hardcode 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.

New in K2

In K2 we implemented the Send-Receive-Reply mechanism, the Name Server, and the Rock Paper Scissor Server & Client.

Send-Receive-Reply

SRR is implemented by adding a `TaskState state;` field to each `TaskDescriptor`. `TaskState` is a tagged union with the following structure:

```
struct TaskState {
    enum uint8_t { UNUSED, READY, SEND_WAIT, RECV_WAIT, REPLY_WAIT } tag;
    union {
        struct {
        } unused;
        struct {
        } ready;
        struct {
            const char* msg;
            size_t msglen;
            char* reply;
            size_t rplen;
            int next;
        } send_wait;
        struct {
            int* tid;
            char* recv_buf;
            size_t len;
        } recv_wait;
        struct {
            char* reply;
            size_t rplen;
        } reply_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:

- `UNUSED`: the `TaskDescriptor` does not represent a running task.
- `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.

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`.

Reply

`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 ever `SEND_WAIT` task up, and writing the syscall return value of `-2`.

Name Server

The Name Server task provides tasks with a simple to use mechanism to discover one another's Tids.

Public Interface

The name server's public interface can be found under `include/user/tasks/nameserver.h`, a header file which defines the methods and constants other tasks can use to set up and communicate with the nameserver.

We've decided to use an incredibly simple closure mechanism for determining the Tid of the name server, namely, we enforce that the name server is the second user task to be spawned (after the FirstUserTask), resulting in it's Tid always being 1.

To make working with the name server easier, instead of having tasks communicate with it directly via SRR system calls, 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: nameserver is at capacity.)
  - *Note:* as described later, the nameserver currently panics if it runs out of space, so user tasks will never actually get back `-2`.

Implementation

The name server's implementation can be found under `src/user/tasks/nameserver.cc`.

Preface: A Note on Error Handling At the moment, the name server's implementation will simply panic if various edge-conditions are encountered, namely, when the server runs out of space in any of it's buffers. While it would have been possible to return an error code to the user, we decided not to, as it's unlikely that the user process would be able to gracefully recover from such an error.

Instead, we will be tweaking the size of our name server's buffers over the course of development, striking a balance between memory usage and availability. Alternatively, we may explore implementing userspace heaps at some point, in which case, we may be able to grow the buffer on-demand when these edge conditions are hit.

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).

Aside: 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 {
            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.

Future Improvements We would like to split up the Request into two parts: a request "header", followed by an optional request "body." This would lift the artificially imposed `NAMESERVER_MAX_NAME_LEN` limit on names, as the request header could specify the length of the upcoming message, which the name server could then read into a variable-sized stack-allocated array (allocated via `alloca` / a VLA). This comes with a minor security risk, whereby a malicious and/or misbehaving task could specify a extremely large name, and overlow the nameserver's stack, but given the the operating system will only be running code we write ourselves, this shouldn't be too much of an issue.

New in K3

In K3, we implemented the clock server, using the `AwaitEvent(int event)` kernel primitive.

Interrupt 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.

`_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 will then move any event-blocked tasks back to the `ready_queue` (see `AwaitEvent` documentation below). 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.

> Note: between K2 and K3, we've reworked quite a bit of our context-switch plumbing! As such, The K1 documentation is now **outdated**, and is not reflective of our current implementation. For details on our reworked context switch, please refer to the context-switch presentation slides.

`AwaitEvent`

The kernel contains an `event_queue`, which is an array of 64 optional `Tid`s. 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 interupt 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.

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

Clock Server

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.

Idle Task Implementation

For K3, we chose to implement the Idle Task as a simple busy loop. The Idle Task is spawned directly by the Kernel, and has the special priority of `-1`. Normal user tasks are not allowed to have negative priorities, and as such, the Idle Task is guaranteed to always have the lowest priority of any task on the system.

Idle Task Measurement

To measure how long the idle task is running for, we've added some instrumentation to the kernel's main schedule-activate loop.

Before calling `activate`, we check if the Tid that is about to be activated is the Idle Task Tid. If it is, then we read the value of Timer 3 (which is set to UINT32_MAX during kernel initialization) first. Once the task is preempted, we re-read the value of Timer 3, and calculate the time that the Idle Task was active for. We take this interval and add it to a running total value of idle time. The idle percentage of the system from boot-up is therefore given by `(total_idle_time / (UINT32_MAX - current_time))`

Future Improvements

- We are well-aware that a busy-loop idle task is not the most optimal approach, and we plan to replace it with a proper "switch to low-power mode" implementation at some point in the future.
- At the moment, we only display the idle percentage from kernel start, which provides an indication of *overall* system performance, but makes it difficult to spot sudden spikes / dips in system idle time.
  - As such, we plan to implement a *running* idle percentage, which tracks system idle time within a fixed window. This should make it easier to see any sudden changes in system performance.

Misc. Additions

Aside from the mandatory requirements documented above, we've also made some fairly substantial miscellaneous changes to our Kernel in between K2 and K3:

Switch to C++2a

We've opted to move the project over from C++17 to C++2a, which is the latest version of C++ supported by the arm-none-eabi toolchain on the student CS servers. The primary reason for this was to enable first-class support for C++20s new Designated Initializer Aggregate Initializer syntax, as opposed to relying on gcc's C++ language extension which enables using C's designated initializer syntax. Our kernel makes great use of tagged enumerations, which are significantly easier to initialize using designated initializer syntax.

This has also allowed us to enable the `-pedantic` warning flag, which should make our kernel more standards conformant (though we have opted to enable VLA support, given how useful it can be in embedded contexts).

Pervasive use of `std::optional`

In previous iterations of our kernel, we used a hodgepodge of sentinel values, "enabled/disabled" booleans, and "null states" to represent the absence of / unused data. Starting from K3, we've made the effort to rewrite much of our kernel to use C++17's new `std::optional` type.

`std::optional` enables adding an "out-of-band" null state to objects, and provides a single, unified approach to representing the absence of data. Moreover, by limiting ourselves to only using `std::optional`'s `value()` method, we are opting-in to runtime checks that ensure we do not accidentally use a null value (i.e: calling `value()` on `std::nullopt` results in a kernel panic and immediate abort).

To illustrate how the use of `std::optional` has impacted our codebase, consider the following two examples:

Intrusive Tid-based lists In K2, we would use a sentinel value of `-1` to represent a "null" link. While this worked, it required us to be very careful when working with those links to make sure that we didn't accidentally try to use the "null" Tid as an index into any sort of structure.

In K3, we represent links as a `std::optional<Tid>`, which enables to compiler to enforce *at compile time* that we don't use a "null" Tid to index into any structure. Instead, we are *forced* to first check the state of the optional (via it's `has_value()` method), before extracting the contained value (via the `value()` method).

Unused Tasks In K2, we had a "UNUSED" task state, which was set whenever a task descriptor wasn't being used. This resulted in lots of extraneous assertions and switch-statement cases to handle this special state, which affected the readability and maintainability of our code. Moreover, there were cases where we would forge to check if a task was in the "UNUSED" state, and access it's fields when we were not supposed to.

In K3, instead of having a special "UNUSED" task state, we've opted to change the type of our tasks array from a `TaskDescriptor[]` to a `std::optional<TaskDescriptor>[]`, and have eliminated the special "UNUSED" task state entirely. Now, whenever we want to work with a task from the tasks array, the compiler enforces that we first check if the task is not a `std::nullopt` before using it. Moreover, since we don't have an "UNUSED" state, we were able to remove many of the checks and extraneous switch branches that accounted for it.

Tid Newtype

Instead of using a raw `int` value for the Tid in the kernel, we've created a special type to represent Tids: the cleverly named `Tid` class.

Newtypes are a fantastic way to leverage a language's type system to prevent silly coding mistakes, such as accidentally swapping the arguments to a method, or attempting to perform "invalid" operations on certain types (e.g: what would it mean to add two Tids together?). Moreover, it's now a lot easier to see when Tids are being created and used, improving readability and maintainability.

Integrating the `mpaland/printf` library

While the provided `bwio` library came with it's own `bwprintf` method, we quickly opted to avoid using it, as it didn't support many of the formatting rules and specifiers supported by the standard library. Unfortunately, this meant that we had to rely on `snprintf` to write data into arbitrarily-sized, fixed-size buffers on the stack, passing the buffer to `bwputstr` upon completion. This approach worked, but came with a performance penalty (from having to write into a buffer, just to immediately flush it), and a limitation on output length (based on the size of the stack-allocated buffer).

To work around these limitations, we asked Prof. Karsten if we could integrate the `mpaland/printf` library into our project. Much to our surprise, he said yes!

`mpaland/printf` is an implementation of `printf` and `sprintf` for embedded platforms, supporting all standard formatting rules and specifiers. It's a single `.c` file with an associated `.h` file, and has no dependencies (not even to the C standard library). Most importantly for us though, this library provides an incredibly useful method that the standard library *does not* provide:

```
int fctprintf(void (*out)(char character, void* arg), void* arg, const char* format, ...);
```

This is a `printf` method which takes a function pointer to a *custom* output function!

This is awesome, since it means we can define our own output function which uses `bwputc`, and plug it in to this library! Moreover, once we get our UART servers up and running, we'll be able to define a output function backed by the userland `Putc` method to create a non-blocking `Printf` function with nearly zero effort!

The library is under `<src/include>/common/printf.<c/h>`, and includes all relevant licenses and attributions. We did make a few modifications to the library, which have been clearly noted and documented inline.

K3 Output

Transcript

```
Hello from the choochoos kernel!
time=10  tid=3 interval=10 completed= 1/20
time=20  tid=3 interval=10 completed= 2/20
time=23  tid=4 interval=23 completed= 1/9
time=30  tid=3 interval=10 completed= 3/20
time=33  tid=5 interval=33 completed= 1/6
time=40  tid=3 interval=10 completed= 4/20
time=46  tid=4 interval=23 completed= 2/9
time=50  tid=3 interval=10 completed= 5/20
time=60  tid=3 interval=10 completed= 6/20
time=66  tid=5 interval=33 completed= 2/6
time=69  tid=4 interval=23 completed= 3/9
time=70  tid=3 interval=10 completed= 7/20
time=71  tid=6 interval=71 completed= 1/3
time=80  tid=3 interval=10 completed= 8/20
time=90  tid=3 interval=10 completed= 9/20
time=92  tid=4 interval=23 completed= 4/9
time=99  tid=5 interval=33 completed= 3/6
time=100 tid=3 interval=10 completed=10/20
time=110 tid=3 interval=10 completed=11/20
time=115 tid=4 interval=23 completed= 5/9
time=120 tid=3 interval=10 completed=12/20
time=130 tid=3 interval=10 completed=13/20
time=132 tid=5 interval=33 completed= 4/6
time=138 tid=4 interval=23 completed= 6/9
time=140 tid=3 interval=10 completed=14/20
time=142 tid=6 interval=71 completed= 2/3
time=150 tid=3 interval=10 completed=15/20
time=160 tid=3 interval=10 completed=16/20
time=161 tid=4 interval=23 completed= 7/9
time=165 tid=5 interval=33 completed= 5/6
time=170 tid=3 interval=10 completed=17/20
time=180 tid=3 interval=10 completed=18/20
time=184 tid=4 interval=23 completed= 8/9
time=190 tid=3 interval=10 completed=19/20
time=198 tid=5 interval=33 completed= 6/6
time=200 tid=3 interval=10 completed=20/20
time=207 tid=4 interval=23 completed= 9/9
time=213 tid=6 interval=71 completed= 3/3
Goodbye from choochoos kernel!
```

Explanation

Each client task displays the current time as it executes. The `time` should always equal `interval * num_completed`, which holds for every line in the transcript. This means that when multiple tasks are blocked for different amounts of time, the tasks are woken up in the right order as early as possible.

We wrote a python script in `test/k3_expected.py` that calculates the exact output of the `k3` tasks (based on the above formula), and the output of the python script matches our execution transcript exactly.

Idle measurements are printed to the top right of the screen in real-time. However, since these measurements are not deterministic, they are omitted from the transcript.