Contents

The choochoos Kernel **1**
- Project Structure . . . 1
  - Navigating the Repo . . . 1
  - Building choochoos . . . 1
- Architectural Overview . . . 2
  - Scheduling . . . 2
  - Context Switching - Task Activation . . . 3
  - Context Switching - SWI Handling . . . 4
  - Handling Syscalls . . . 4
- System Limitations . . . 5

New in K2 **5**
- Send-Receive-Reply . . . 5
  - Sender-first . . . 7
  - Receiver-first . . . 7
  - Reply . . . 7
  - Error cases . . . 7
- NameServer . . . 8

K2 Output (RPS) **8**
- Transcript . . . 8
- Explanation . . . 9

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, as well as the NameServer.

Send-Receive-Reply

SRR is implemented by adding a `state` field to each `TaskDescriptor`, which is the following tagged union:

```
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. Let's explain each of the states:

- `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 thus, must 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 procdure 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`.

NameServer

K2 Output (RPS)

Transcript

```
Hello from the choochoos kernel!
random seed (>= 0): 2
pause after each game (y/n)? n
num players (0-32): 3
player 1 priority  (default 1): 1
player 1 num games (default 3): 3
player 2 priority  (default 1): 2
player 2 num games (default 3): 5
player 3 priority  (default 1): 3
player 3 num games (default 3): 3
[Client 3] I want to play 3 games. Sending signup...
[Client 4] I want to play 5 games. Sending signup...
[RPSServer] matching tids 4 and 3
[Client 4] received signup ack
[Client 4] I want to play 5 more games. Sending scissors...
[Client 5] I want to play 3 games. Sending signup...
[Client 3] received signup ack
[Client 3] I want to play 3 more games. Sending paper...
[Client 4] I won!
[Client 4] I want to play 4 more games. Sending paper...
[Client 3] I lost :(
[Client 3] I want to play 2 more games. Sending rock...
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[Client 4] I won!
[Client 4] I want to play 3 more games. Sending rock...
[Client 3] I lost :(
[Client 3] I want to play 1 more game. Sending rock...
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[Client 4] it's a draw
[Client 4] I want to play 2 more games. Sending rock...
[Client 3] it's a draw
[Client 3] sending quit
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[RPSServer] tid 3 quit, but tid 5 is waiting. Matching tids 4 and 5
[Client 5] received signup ack
[Client 5] I want to play 3 more games. Sending scissors...
[Client 3] exiting
[Client 4] I won!
[Client 4] I want to play 1 more game. Sending scissors...
[Client 5] I lost :(
```

```
41  [Client 5] I want to play 2 more games. Sending rock...
42  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
43  [Client 4] I lost :(
44  [Client 4] sending quit
45  [Client 5] I won!
46  [Client 5] I want to play 1 more game. Sending paper...
47  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
48  [RPSServer] tid 4 quit, but no players are waiting.
49  [Client 4] exiting
50  [Client 5] other player quit! I guess I'll go home :(
51  [Client 5] exiting
52  Goodbye from choochoos kernel!
```

Explanation

- [1-10] game setup - the user's input is included. The random number generator is configured with a seed of 2, and we will pause for input after each game is finished. We are running 3 client tasks, with priotities (1,2,3) respectively, that want to play (35,3) games respectively.