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.

K1 Output

Transcript

```
1  | Created: 1
2  | Created: 2
3  | MyTid=3 MyParentTid=0
4  | MyTid=3 MyParentTid=0
5  | Created: 3
6  | MyTid=3 MyParentTid=0
7  | MyTid=3 MyParentTid=0
8  | Created: 3
9  | FirstUserTask: exiting
10 | MyTid=1 MyParentTid=0
11 | MyTid=2 MyParentTid=0
12 | MyTid=1 MyParentTid=0
13 | MyTid=2 MyParentTid=0
```

Explanation

- Lines 1-2
  - `FirstUserTask` calls `Create(3, OtherTask)` twice, and is immediately re-scheduled both times, printing out the two lines. This stems from the fact that `FirstUserTask` is spawned with a priority of `4`, which is a higher priority than the two `OtherTask`s it spawned. As such, the scheduler kept re-prioritizing the `FirstUserTask` for execution.
- Lines 3-5
  - When `FirstUserTask` calls `Create(5, OtherTask)`, instead of immediately being rescheduled, the newly-created `OtherTask` is run instead, as it was spawned at a higher priority than the `FirstUserTask`.
  - When `OtherTask` calls the `MyTid` syscall, it gets back a value of 3. This makes sense, as we use a simple increasing Tid counter (starting at 0) when spawning tasks. As this task is the 4rd task to have been spawned by the system, it was assigned a Tid of 3.
  - When `OtherTask` calls the `MyParentTid` syscall, it gets back a value of 0. This is as expected, as its parent task is `FirstUserTask`, which is always assigned the Tid of 0.
  - Even though the `OtherTask` yields between its two print statements, at the time that it is yielded, it remains as the highest priority task on the system, and is immediately rescheduled.
  - Only once the `OtherTask` completes does execution return to the `FirstUserTask`, at which point it prints out "Created: 3"
- Lines 6-8
  - These lines are identical to lines 3-5, as once the first high-priority `OtherTask` calls exit, its Tid is returned to the free Tid pool, which

is then re-used for the newly spawned `OtherTask`.

- Lines 9-13
  - Once the second high-priority `OtherTask` has completed, and execution has returned to the `FirstUserTask`, the `FirstUserTask` is done, and calls exits itself. At this point, there are only two remaining tasks left on the system: The two low-priority `OtherTask`s
  - Since both tasks have the same priority, they are scheduled in a round-robin fashion, which results in the interleaved output between the two tasks. When the first `OtherTask` prints its line, it immediately yields, allowing the second `OtherTask` to print its line. The second `OtherTask` then yields, allowing the first `OtherTask` to print its line (and so on and so forth).
  - NOTE: the two tasks yield execution during the `MyTid` and `MyParentTid` calls as well, interweaving execution even further. That being said, since there is no output between these syscall invocations, it is more difficult to see the interwoven execution flow.