CS452 Kernel 1

Christoph Ulshoefer (20751216) and Thomas Broadley (20522223)

January 26, 2018

1 Operation

This section discusses how to run our kernel and the different `make` targets specified in our Makefile.

1.a Running the kernel

Execute the following commands in RedBoot to start the kernel:

```
load -b 0x00218000 -h 10.15.167.5 "ARM/csulshoe/k1.elf"
go
```

The kernel immediately starts the first user task, which spawns four other tasks. Once all of these tasks have exited, the kernel exits to RedBoot. Starting the kernel repeatedly is also possible, i.e. as far as we know, we do not leave the ARM box in a 'bad' state.

1.b `make` targets

`make docs`: With Doxygen installed, this command will generate documentation from comment blocks in our source files. Once generated, you can find the documentation in the `docs/` folder.

`make arm`: This will compile code the same way as in the student CS environment, but is meant to be used outside that environment. Two environment variables need to be set. Firstly, the environment variable `armlibs` to set to a directory containing `libgcc.a` (e.g. `some/dir/gcc-arm-none-eabi-7-2017-q4-major/lib/gcc/arm-none-eabi/7.2.1` on our machines). In addition, your `PATH` needs to include a folder containing the GNU ARM Embedded Toolchain compilation suite (*e.g.* `arm-none-eabi-gcc` and `arm-none-eabi-ld`).

`make versatilepb`: Use this to compile for the Versatile/PB platform, to be run in QEMU. See our section on emulation for more information. If `make arm` works, this should also work. Note: Although Versatile/PB supports the ARMv5 architecture, we still target ARMv4 for compatibility with the ARM 920t processor used in the TS7200.

`make qemu` and `make qemuwin`: Runs `make versatilepb` and starts QEMU in GUI mode.

`make qemuconsole` and `make qemuwinconsole`: Runs `make versatilepb` and starts QEMU in console mode. Terminal output is printed to stdio.

`make test`: Runs Googletest unit tests. To run this, you need to set three environment variables: `GTEST`, which needs to point to your Googletest include directory (e.g. `/usr/local/include/gtest`; `LIBGTEST` (e.g. `/usr/local/lib/libgtest.a`); and `BOOST` (e.g. `/usr/local/include/boost`).

`make trainslab`: Builds using the compilation suite in `/u/wbcowan/gnuarm-4.0.2/bin`.

`make upload`: Runs `make clean`, `make trainslab` in order and then uploads the code to the directory `/u/cs452/tftp/ARM/<user>/`, where `<user>` is the user returned by `whoami`.

`make`: Does the same as `make upload`.

2 Implementation

This section discusses how we implemented each of the major parts of our kernel. It provides an overview of the design decisions we made and our justifications for them.

2.a Context switch

During the context switch, we store kernel and user registers on their respective stacks using a trapframe. We access these data in C via a `trapframe` struct. When creating a new task, we place a dummy trapframe on the top of the task's stack.

2.a.1 Switching out of the kernel

First, the scheduler wrapper function `schedule()` selects a task to be run by querying the scheduler for the next task.

After that, `schedule()` calls `task_activate()`. To enter the task, this function saves the kernel registers on the kernel stack. Then it enters system mode via the `msr` instruction to restore all user registers from the trapframe on the user stack. Finally, it returns to kernel mode and calls `MOVS pc, lr` to enter user mode.

Back in the task, execution starts at the top of the function, or just after the `swi` instruction in a syscall wrapper function.

2.a.2 Switching into the kernel

When a user task calls a syscall wrapper function, the `swi` instruction is executed. To handle this, we have place the address of the `kernel_entry` function in `0x28` when starting the kernel.

Once there, we save all user registers in a trapframe on the user's stack. We pass the address of the trapframe to the (`handle_interrupt`) function as a C data structure. This allows us to examine the syscall arguments, stored in registers `r1` and higher, in the function. We also save the user task's CPSR and the value of the kernel link register as set by the `swi` instruction.

To handle interrupts, we switch on the value of `r0` as stored in the trapframe, which contains the syscall code. Syscalls that return a value store this value in the `r0` field of the trapframe. We then return cleanly from `handle_interrupt()`.

After returning, we reload the kernel state and return to `task_activate`. This function immediately returns to `schedule()`, which will schedule the next task, if available.

2.b Task control

This section describes the data structures and algorithms we use to manage tasks.

2.b.1 Task control block and list of tasks

We create an array of task descriptors on the kernel stack when initializing the kernel. Each task descriptor contains, among other fields: pointers to the next and previous entries in its ready queue in the scheduler; a task state (active, ready, or zombie); a pointer to the task's trapframe; a pointer to the parent task's task descriptor; and the task's ID.

2.b.2 task_init

This function puts the task's trapframe on its stack location. Currently, we support 64 tasks, each with about 384 kB of memory. We picked this number because it is a power of two and because we don't expect to use more than 50 tasks in this course. We consider the stack size to be sufficiently large, since were able to run A0 as a user task with this configuration. The memory section used for user tasks is 384 kB below about `0x01FDCFFC` (i.e. 384 kB below the kernel stack). We still have about 7.8 MB left over, and will assign it to tasks/the kernel, should it become necessary.

2.b.3 task_activate

This function sets the given task's state to `TASK_ACTIVE`, changes the current task pointer to point to this task, and leaves the kernel as described above.

2.c Scheduler

A scheduler is a struct containing a maximum priority and a pointer to an array of ready queues (or RQs) of size `max_priority + 1`. The ready queues are ordered by priority ascending, so that `queues[n]` contains the ready queue of priority n.

Each ready queue is stored in a doubly-linked circular list. This data structure allows us to enqueue to the tail, dequeue from the head, and check for emptiness in constant time.

The ready queue itself is a pointer to the task descriptor (or TD) at the head of the list. Each task descriptor has a `next` and a `prev` field; the `next` field points to the task descriptor after it in the list, while `prev` points to the one before. The `prev` field of the head of the list points to the tail, while the `next` field of the tail points to the head. An empty ready queue is represented by `(task_descriptor *)0`. In a ready queue with one element, the head task descriptor's `next` and `prev` fields point to itself.

The scheduler uses these ready queue functions to provide the following functionality in `src/multitasking/scheduler.h`:

Function	Description	Runtime (m is # of priorities)
`scheduler_init`	Sets each of the scheduler's RQs to `(task_descriptor *)0`.	$O(m)$
`scheduler_register`	Adds the given TD to the correct RQ.	$O(1)$
`scheduler_next_task`	Searches for the highest-priority non-empty RQ and dequeues from it.	$O(m)$

We don't know yet how many priorities we will need. We plan to benchmark our kernel with different maximum priorities once we have set up more user tasks handling, for example, I/O via interrupts. In the most extreme case, every task receives a unique priority. Given that we have a maximum of 64 tasks, this means we may need 64 priorities. If we require fewer tasks in the future, we will not hesitate to change the maximum priority.

Given that the number of ready queues is less than constant, we think it will be fast enough for now to call `ready_queue_empty` on each ready queue when performing `scheduler_next_task`.

In the future, we plan to optimize the scheduler by using a bitmap to maintain a record of which ready queues are non-empty and extract the index of the highest-priority non-empty ready queue. We also plan to benchmark the scheduler and other parts of the kernel, to determine where we should improve performance.

3 Kernel output

Our kernel produces the following output:

```
Created: 2.
Created: 3.
In other task: MyTid(): 4, MyParentTid(): 1
In other task: MyTid(): 4, MyParentTid(): 1
Created: 4.
In other task: MyTid(): 5, MyParentTid(): 1
In other task: MyTid(): 5, MyParentTid(): 1
Created: 5.
FirstUserTask: exiting
In other task: MyTid(): 2, MyParentTid(): 1
In other task: MyTid(): 3, MyParentTid(): 1
In other task: MyTid(): 2, MyParentTid(): 1
In other task: MyTid(): 3, MyParentTid(): 1
```

Here is a step-by-step explanation of why our kernel produces this output:

1. The first user task is created first. It has a priority of 5 and is assigned the task ID 1.
2. The first user task creates a task with priority 4. It is assigned the task ID 2. The call to `Create` causes the kernel to reschedule. The first user task (with priority 5) is the highest-priority task, so it is scheduled and prints "Created: 2."
3. The first user task creates another task with priority 4. It is assigned the task ID 3. Again, the kernel reschedules. The first user task is still the only task with priority 5, so it is scheduled again and prints "Created: 3."
4. The first user task creates a task with priority 6. This time, the created task (which has task ID 4) has a priority higher than that of the first user task, so it is scheduled and prints "In other task: MyTid(): 4, MyParentTid(): 1". Since task 4 was created by the first user task, the result of `MyParentTid` is correct.
5. Task 4 calls `Pass`, which causes the kernel to reschedule. Since task 4 is still the only task with priority 6, it is scheduled again. It prints "In other task: MyTid(): 4, MyParentTid(): 1" again.
6. Task 4 exits. The kernel schedules the first user task, which has the next-highest priority. This task returns from `Create` and prints "Created: 4."
7. Steps 4 to 6 repeat, except that the task created by the first user task is assigned the task ID 5. After task 5 exits, the first user task is once again the only task with priority 5, so it is scheduled again and prints "FirstUserTask: exiting".
8. The first user task exits. The only remaining tasks are task 2 and task 3. Both tasks have priority 4 and task 2 was created first. Therefore, task 2 is scheduled first and prints "In other task: MyTid(): 2, MyParentTid(): 1".
9. Task 2 calls `Pass`. The kernel schedules task 3, which demonstrates the round-robin nature of the scheduler. Task 3 prints "In other task: MyTid(): 3, MyParentTid(): 1".

10. Task 2 calls `Pass` and task 2 is scheduled. This task prints the same line it printed in step 8.
11. Task 2 exits. Task 3 is scheduled and prints the same line it printed in step 9.
12. Task 3 exits. All tasks have exited, so the kernel exits to RedBoot.

4 Extra features

We used an emulator and kernel-level assertions to improve our Kernel 1 development experience. We also generated documentation for the user- and kernel-level APIs our kernel exposes. Finally, we implemented two features not in the kernel specification: user-level assertions and implicit exit at the end of user tasks.

4.a Emulation

We use the QEMU emulator to run our kernel on an emulated ARM Versatile Platform Baseboard for ARM926EJ-S. We determined that this board was similar enough to the Cirrus EP9302 to permit us to mainly develop and test our kernel using the emulator. Like EP9302, the Versatile/PB has three UARTs with 16-byte FIFOs whose baud rate, byte sizes, stop bit counts, and parity mode can be set up to match the lab terminals and train sets. It also has a 1 MHz 16-bit timer with interrupts that we believe is similar enough to the EP9302's timers.

We found that testing our kernel in QEMU both sped up and slowed down our development process. We enjoyed the faster compile-run loop and being able to work outside the lab. The emulator also helped us determine if problems were behind the keyboard or due to the specific system architecture. However, we did spent a significant chunk of time finding bugs that only appeared when running our kernel on the TS-7200.

4.b API documentation

We use Doxygen to generate API documentation from comments in our kernel's source code. To generate the documentation for yourself, install Doxygen and run `make docs`.

4.c Kernel assertions

The `kassert` macro can be used in the kernel to make assertions. If `kassert`'s first argument evaluates to a falsy value, it prints the file, line number, and calling function of the failed assertion, then calls `syscall_panic`, defined in `src/syscall/syscall.c`.

The `main` function in `main.c` is a wrapper for the kernel's actual functionality that sets up `kassert`. Above the call to `kmain`, it stores its frame pointer and stack pointer in global variables. Below the call, it places a global assembly label called `panic_exit`. The `syscall_panic` function simply loads the global variables into the frame pointer and stack pointer and branches back to the label. The effect of this is similar to that of returning from `kmain`, causing the kernel to exit "cleanly" to RedBoot.

4.d The `Panic` and `Assert` syscall wrappers

The `Panic` syscall wrapper causes the kernel (and, as a result, all user tasks) to exit. The `Assert` syscall wrapper is a conditional version of `Panic` that behaves similarly to `kassert`. Both use the `swi` instruction to make the kernel call `syscall_panic`.

We decided that failed user-level assertions should cause the kernel to exit because we expect that, in future versions of our kernel, a bug in one task will likely prevent other tasks from completing their duties. If this is the case, we will want to return to RedBoot immediately if an assertion fails, to speed up our debugging loop.

4.e Implicit `Exit`

Returning from a user task function has the same effect as calling `Exit` at the end of the function. To accomplish this, when entering the user task for the first time, we set the link register to the address of an assembly label called `sys_exit`. When the user task returns, it executes the instructions at this label, which execute the `swi` instruction with the syscall code corresponding to `Exit` in `r0`. We think it is cleaner to omit the call to `Exit` from the end of user tasks.

5 File and repository hashes

The commit before adding this report had SHA `d84c6b626f53261b252587bb0ac5b34b95881e3d`.

5.a MD5 Hashes

```
4de60761cbfa9de12b606fcd6911ccbc  ./.gitignore
02d8184c8c22d269f39b92be9923df26  ./.travis.yml
70e14abcde0ceaeb01a7e2a0765fe0f7  ./Makefile
e33b1460f48b4635609443c135260b9a  ./README.md
1e4b0158c10ed01065347d32271b6e95  ./main.c
3cd7e7ac4c1954c769f0388b2abc3e7e  ./include/kernel/glue/myio.c
aad66b93ffc4be9edf6409509efc1cb9  ./include/kernel/glue/myio.h
80c948fead3dfdfcd37e9b973a5f74ac  ./include/kernel/glue/mytimer.c
9351ad8074357964cf892f817bae84f4  ./include/kernel/glue/mytimer.h
423cd13b4b587cb0d6ad53c920d707c3  ./include/kernel/glue/timer_data.h
607b76d33b0c6df8bda127fe6634c6ce  ./include/kernel/labenv/timer.c
7620072e6ae233d69527ac1ee9777cce  ./include/kernel/labenv/timer.h
867d06b95f132fbd63d2ba63cae7ef21  ./include/kernel/labenv/ts7200.h
15a6247e87d0a75ad8fde9634ea06213  ./include/kernel/versatilepb/timer.c
0f3f45aa8f3ab83a6ccdf311d468f421  ./include/kernel/versatilepb/timer.h
a52772c927a3f3937e2af9f2536c5433  ./include/kernel/versatilepb/versatilepb.h
0cfc5d228504f5f930b3181504c26275  ./include/common/codes.c
fbc60200cdfe7d09abd5803652ab09ca  ./include/common/codes.h
04a99fa4e57230b2b2b7a525a1105097  ./main.ld
f203d61b324bd43535ae6b4f294fd95a  ./src/a0terminal.c
f5d9c4bdb35e35420d750134b58f4e7c  ./src/a0terminal.h
2f59b880230fd85f2ec9293ddb3f2453  ./src/a0trackstate.c
8bd851b3559b042968fdbd46d1cb8311  ./src/a0trackstate.h
0a22f4694653bf5a9ff1fd28554840f7  ./src/buffer.c
79b7f845892234c65a6bb914ee70e631  ./src/buffer.h
a531fe7fabb65931d7413e2efe399f2d  ./src/stdlib.c
c895052ff8f5c1ca59188c7fc1d66200  ./src/stdlib.h
af9cdf9014bec08c6df74cead3ca105d  ./src/track/README.txt
7707b57cb3a6c3703c0e3da3acbef763  ./src/track/parse_track.py
1e06c8505279ed9f4d10136465c81618  ./src/track/parts_tracka
a408ef1736b356024bf0dc2bc05d98b0  ./src/track/parts_trackb
d38663c8897d9b202ff02e75b085197c  ./src/track/track_data.c
c22056be41b20cbf8745ec91c2956067  ./src/track/track_data.h
54c5c88889790a67c9d774436a8317ca  ./src/track/track_node.h
```

```
50e0b1150b39a2425cb2180c33e0e57f  ./src/track/tracka
dab2764f1d7f07aa454bad2ec01158b8  ./src/track/trackb
ba7f19183e5a720ed78097b9ccd324dd  ./src/cp_vec.c
503fb56ff1929bcc8d7a6b3d19b2524e  ./src/cp_vec.h
38424f2856725ba4ce4919eefe5e3142  ./src/crash.h
608c3732de6719e206bf1c76772bd549  ./src/crash.s
cc77c33107051286e099c587bc31ab8e  ./src/interrupt.c
48fd1c9b9d204b2c0a1b5292fa6116be  ./src/interrupt.h
eafdaf77048f8e5a5e228a8cb2fc9ea2  ./src/kernToUser.s
9528cb7fa090bbf3c55d94aca56f51d4  ./src/startup.s
bae792256451c56f9dbd4101f12b58bf  ./src/trap.s
27693fc5a8ba138f8358ba505ca708f1  ./src/assert.c
d8338a43bb4b734d6d3c1567378164f4  ./src/assert.h
bec5edf5b40af4209c289f8c83f248ef  ./src/attributes.h
a290bb10acff6e83c1c3611ff9095377  ./src/multitasking/ready_queue.c
8a7c04896ef89aa935a7092691112c0e  ./src/multitasking/ready_queue.h
b2034a32d12ce68a1d2dfe519aecb072  ./src/multitasking/schedule.c
606aa652bf2fe698ec92269ae21943c4  ./src/multitasking/schedule.h
4412b55a51c1db2b4438f980342597f2  ./src/multitasking/scheduler.c
e765318e5fc805bb319cd98dbe2d1d1f  ./src/multitasking/scheduler.h
444b11d827bfb8fbfbe4c4a5a02cb363  ./src/multitasking/task.c
43b3795cddfa87ce9f7cf5900925b025  ./src/multitasking/task.h
5d11a1704f438c49308c733f97dbb5b2  ./src/syscall/syscall.c
26e09b17488f6e0001ad29851c5e0de5  ./src/syscall/syscall.h
000a214c2834c915e605751e799411a7  ./start-qemu.sh
6420f56d16b729f3b1f7df4ada222bae  ./test-resources/assert.c
fb1ea3f271425694eb40bfa638aacf0c  ./test-resources/assert.h
2716c62882d905eba8f133bee500e68f  ./test/Makefile
1d5637c2d357f516ab04839882b14d62  ./test/test_buffer.cc
3c0a4da6fa46fa56be8cabd6b3dc09d7  ./test/test_buffer.h
17941c03ce0e9198777bf6f72be31729  ./test/test_ready_queue.cc
d2b2dfee163e44acd00c3a2e55c5a89a  ./test/test_ready_queue.h
ccf16fd8cae91360a368d2a85346db65  ./test/test_scheduler.cc
1d192198f74418639a6647d117764d21  ./test/test_scheduler.h
4c2e697ff2653b49cf5ebffc34149fef  ./test/test_task.cc
87853074cd0921b02d1b6030a07769c7  ./test/test_task.h
bc2c6f76374c2e9c7ee3e73475392e97  ./valgrind.sh
e9107b09ba9b4f83b5646ff02ae6f1dc  ./Doxyfile
d9cd792c1413ac79bf45f799ae903f42  ./a0-csulshoe.pdf
abe6d3f79abf3f85a07b893b32787c01  ./qemu/Makefile
43c94cb9a2a2e5471b59be8016938b32  ./qemu/python/socket_io.py
4479217661adc53de034d9289587d187  ./qemu/qestartup.s
a2276c9a3852f9c6814bbe9b0725754f  ./qemu/run_qemu_cli.sh
529bbb530b43af32e5b5640162084652  ./qemu/test.c
e7dd3a9878639e374e93508f5cde030d  ./qemu/test.ld
d41d8cd98f00b204e9800998ecf8427e  ./qemu/startup.s
76deb0bc5db82d6493c7960d10c7f846  ./usr/a0.c
e45dd106ef4922cded3a722f9605fc5a  ./usr/a0.h
2ccde6ee5109cfb72df91409de424754  ./usr/tasks.c
5481417ee976c754965d93edc30674dc  ./usr/tasks.h
664c61ea9d8e8cd3c89929b30ea852d4  ./versatilepb.ld
```