Kernel Assignment 1

Niclas Heun
nheun – 21111245

Trevor J. Yao
t27yao – 20830290

University of Waterloo: CS 452/652

29 September, 2023

1 Overview

This assignment begins the implementation of the Train Control Kernel in order to enable a task-based parallelism model for task management. The end goal is to better enable complex train-control, which would not be possible with the single sequential thread of execution used in A0.

For this assignment, there is no interactive interface. The Kernel starts up and executes the user-level entrypoint function `void user_main(void)`. The User level programs required for this assignment are found in `user/src/k1.c`, and the entrypoint executes the first program.

The submission SHA of the last commit is <commit-SHA>.

2 Execution

The kernel image is found in the repository root – aptly named `kernel.img`. Build the program from the root directory by running one of `make`, `make all`, or `make kernel`. The prebuilt image is located in the repo root, titled `kernel.img`. It may be necessary to set the `XDIR` variable to point to the directory containing the cross-compiler. For example, in the `linux.student.cs` environment, `XDIR` should be `/u/cs452/public/xdev` (i.e. the parent of the `bin` directory). Therefore, the program can be compiled in the `linux.student.cs` environment by changing directory to the program root and executing:

```
make XDIR=/u/cs452/public/xdev
```

3 Kernel Structure

The Program files are structured into three parts – Kernel, User Libraries, and User. These parts each have their own directories, `kernel/`, `./` (i.e. the repository root), and `user/` respectively. In each of these directories, the header and implementation files are split into the directories `includes` and `src` respectively.

3.1 Generic Kernel Loop

The kernel is designed as a generic loop, rather than as an event handler. As part of the initialisation, the kernel performs a number of tasks:

(1) Initialise singleton data structures for task/stack allocation and scheduling;

(2) Initialise the initial kernel state for context switching;

(3) Install the address of the exception vector into the `VBAR_EL1` system register;

(4) Initialise the UARTs;

(5) Allocate and initialise the first user task (with default entrypoint `user_main`).

Then, on each iteration of its main loop, the kernel does the following:

(1) Remove the next task from the queue as given by the scheduling algorithm (Section 3.3.5);

(2) Context switch (Section 3.3.1) into the selected task;

(3) On execution and handling by the hardware of the `svc N` instruction, context switch back to the kernel, returning the exception code;

(4) Handle (Section 3.3.6) the given system call as described in the interface.

The kernel loop ends when all created user tasks (including the initial task) has exited, and restarts the machine.

3.2 Critical System Parameters and Limitations

The stack size is limited to `0x80000` bytes, or 512 KiB per task. The maximum number of simultaneous running tasks is 1024. These numbers were chosen because they provide a modest stack size while allowing the possibility of a large number of simultaneously running smaller tasks, and both being powers of 2 (2^{19} and 2^{10} respectively) to allow for faster calculations. In total, this gives the program 500 MiB to work with in total task stack size, which, taking into account that we have approximately 1 Gb of program memory to work with, and leaving room for the bootloader and the program load address of `0x80000`, gives more than enough room.

The TIDs are limited by the address space of `uint16_t`, i.e. 65,535 TIDs. This is far more than the total number of tasks, while being reasonably space efficient. The TIDs are reassigned, and the algorithm for this is discussed in Section 3.3.4. The larger address space for TIDs compared to the number of simultaneous active tasks is to prevent more expensive reassignment searches from occurring frequently.

3.3 Algorithms

3.3.1 Context Switching

Context switching is implemented in `kernel/src/context-switch-asm.S` as an assembly routine linked as a C routine. On boot, the kernel starts first and creates an entrypoint user-level task, and context switches it via the following routine:

```
void context_switch_out(task_t *curr_user_task, kernel_state *kernel_task);
```

This assembler routine pushes the addresses of the task descriptors onto the stack (important for context switching back into the kernel). It then uses `x0` to point to the beginning of the `kernel_task`, and directly stores registers `x1` to `x30` into the state using offsets of `x0`. It then lastly takes `x0` which was stored onto the stack and saves it into the state. Next, the routine has to load the saved registers of the user task. It does so again by using `x0` and `x1` as scratch registers, loading the address of `curr_user_task` into `x0` from the stack. First, it directly loads registers `x2` to `x30` using offsets of `x0`. It then loads the `pc` into `ELR_EL1`, `sp` into `SP_EL0`, and the saved `pstate` into `SPSR_EL1`, taking care to only load 32 bits. After this, it then loads registers `x1` and `x0`, in that order, since we used them as scratch registers.

Context switching in is much the same but in inverse, but makes use of the fact that after the `svc` instruction is executed, we are on the kernel stack, and thus is able to re-save the user task and restore the kernel task. However, before storing the user task, we need to push any registers we need as scratch (i.e. `x0`) onto the stack before doing anything else, and rewrite it to the task descriptor at the end.

3.3.2 Stack Allocation

User level stacks are allocated in memory following the kernel stack, with the previously discussed constraints in Section 3.2. Stack allocation is managed by the singleton `stack_alloc` structure defined in `kernel/include/stack-alloc.h`. They are allocated in linear fashion after the kernel stack, but not zeroed out to increase performance of task creation, since there was a significant lag when clearing a stack size of 512 KiB. Freed task stacks are reclaimed linearly to reduce fragmentation. It was considered to use slab allocation to better reduce fragmentation, but the overhead for managing this does not seem worth it at this time. If we see fragmentation later on when working with more tasks, this change will be made.

3.3.3 Task Allocation

The task allocation mechanism employs a sophisticated slab allocation strategy to optimise memory usage and mitigate memory fragmentation. Initially, the memory is segmented into 128 slabs, each containing 8 tasks. Whenever the free list is exhausted, a new slab is initialised, with its 8 tasks added to the free list and the slab marked as active. To track the usage, a counter is associated with each slab to monitor its active status and the number of used tasks; additionally, every task contains a variable to indicate the slab it belongs to.

When a task is added back to the freelist, it is not just added to the front of the free list. Instead, the free list is structured by slab-id, ensuring the utilisation of lower slabs first. When all tasks from a specific slab are added back to the free list, and consequently, the task usage counter for that slab resets to zero, all tasks from that slab are extracted from the free list and the slab is deallocated. If again additional memory is required, the lowest unused slab is (re)activated and its tasks are (re)integrated into the free list. This procedure ensures minimal fragmentation and optimal utilisation of available memory. This is also less expensive to do than other algorithms, such as precisely tracking allocations in a separate structure.

3.3.4 Task IDs

The task ID (TID) allocation is inspired by the algorithm UNIX uses for allocating process IDs. TIDs are allocated sequentially to tasks until they reach the maximum TID value (65,535), in which allocation returns to 300 and increases again. This is used because it is less likely that long running tasks were allocated with higher TIDs, and can obviously be adjusted in the future as we see fit to increase performance.

On each choice of TID, the kernel checks that the TID is not in use, and if so tries the next one. This is obviously worst-case expensive, but as mentioned previously, the large TID space in comparison to the number of tasks makes the search unlikely to be long. The slight increase in cost per task allocation is marginal, and if it works for UNIX, it works for us.

The Task ID algorithm is implemented in the static `task_queue_get_tid` method located in `kernel/src/task-queue.c`.

3.3.5 Scheduling

The scheduling follows a straightforward round-robin approach, housed within the task queue. Starting with the highest priority queue, it cycles through the queue to return the first task in a `STATE_READY` condition. Tasks not in ready state are skipped and maintain their position within the queue. Should the queue be empty or only comprise of non-ready tasks, the scheduler advances to the next lower priority queue in search of ready tasks. New tasks are always added to the end of their respective priority queue.

3.3.6 System Calls

The individual System Calls implemented for this assignment (`Create`, `Exit`, etc) are implemented in `includes/task.h`. These methods are simply a wrapper on the `syscall` procedure implemented in `includes/syscall.h`. This method takes advantage of the compiler moving call arguments into registers `x0` to `x7`, and simply calls `svc #0`, with the syscall number being stored as the first argument, and then returns (when the kernel eventually switches us back as the active task).

These arguments are preserved during the context switch, and saved in the task descriptor for processing. During this, the kernel executes code based on the syscall number in `x0`, and returns the result (if any) in `x0`, following aarch64 convention. This means that when the application is switched back, and returns to the location of the `svc` instruction, it will return the value in `x0`.

3.4 Data Structures

3.4.1 Task Queue

Task Queue maintains one queue per priority, which keeps track of all tasks of this priority. Whenever a new task is created, it is added to the end of the respective priority queue. Removing tasks is handeled by the scheduler, which decides which task is returned / removed from the queue.

The implementation is straightforward as it uses the next pointer in the task structure to maintain the queues. Task Queue itself only stores pointers to the first element ($\bar{\text{f}}$ront) and the last element ($\bar{\text{b}}$ack) of each priority queue.

3.4.2 Task Descriptor

Each running task is associated to a Task Descriptor, which is used in for scheduling, handling system calls, and context switching. The Task Descriptor stores the following information:

- Registers `x0` – `x30`, `pc` (program counter), `sp` (stack pointer), and `pstate` (processor state), at time of last context switch into the kernel;
- TID (Task ID) – see Section 3.3.4;
- Pointer to parent task descriptor;
- The task's current run state (ready, blocked, exited);
- The task's priority;
- An intrusive pointer to the next task in the queue, used by the Task Queue (Section 3.4.1) and the Allocator (Section 3.3.3).

This task descriptor structure is defined in `kernel/include/task-state.h`. This file also defines a kernel state structure, which holds the registers of the kernel for context switching.

In order to simplify context switching, all of the registers are stored in the task descriptor, rather than on the stack. This allows the kernel to maintain ownership, and easily return system call return values to the user-level programs. All the other fields of the task descriptor facilitate the data structures and algorithms which operate on the task descriptors.

4 User Task Output

The Console output of the program is as follows:

```
Created: 2
Created: 3
Task 4 parent: 1
Task 4 parent: 1
Created: 4
Task 5 parent: 1
Task 5 parent: 1
Created: 5
FirstUserTask: exiting
Task 2 parent: 0
Task 3 parent: 0
Task 2 parent: 0
Task 3 parent: 0
```

The first user task runs and created tasks 2 and 3 at a lower priority, and thus continues to run. When it creates task 4, since it is at a higher priority than it, it runs to completion first, even as it calls `Yield` since it is moved to the back of its empty priority queue. When it finishes, it returns to the first user task since it is higher priority than 2 and 3, and we see the same pattern with task 5. After 5, it returns to the first user task, which exits. Subsequent tasks have parent as 0 (i.e. the kernel) after it exits, and task 2 runs first since it was created (and thus added to the queue) first. Tasks 2 and 3 then alternate execution since they are at equal priority.