Kernel Assignment 2

Niclas Heun
nheun@uwaterloo.ca
21111245

Trevor J. Yao
t27yao@uwaterloo.ca
20830290

Professor Ken Salem
CS 452/652: Real-Time Programming (Fall 2023)
University of Waterloo
6 October 2023

Contents

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 enable more complex train-control, which would not be possible with a single sequential thread of execution, as 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 each assignment are found in `user/includes/kX.h`, where `X` is the assignment number, and the entrypoint executes the specified user programs.

1.1 Submission SHAs

The repository for the kernel is ist-git@git.uwaterloo.ca:t27yao/cs452-kernel.git. The submission SHAs for their respective assignments are listed below:

- A1: 6a8841836fc9ce629b573e94b37c901b4d71f2ee;
- A2: <insert-commit-sha>.

2 Execution

Build the program from the root directory by running one of `make` or `make all`. The specific user-level programs for each assignment can be built by specifying the target `make kX`, where `X` is the assignment number. 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
```

2.1 K2 Test Programs

The RPS Game and Performance measurements programs required for K2 are built using the targets `k2` and `k2-timings` (i.e. the default `k2` target will build the RPS Game).

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 nameserver;

(6) 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.

The nameserver names are limited in length to 256, since without heap allocation dynamic arrays are not really possible, nor necessary. It is incredibly unlikely that names will be longer than 256 characters, and if so, the user programmer should seriously reconsider a better naming scheme. The maximum number of names the nameserver can hold is also 256, since it is reasoned that most tasks do not need to be known, and thus being able to hold a quarter of tasks in the nameserver is more than enough.

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.

User tasks are allocated beginning with TID 50, since we know there will be longer running server tasks that are started by the kernel.

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`.

3.3.7 Message Passing

Message passing is implemented as part of three methods: `Send`, `Receive`, and `Reply`. These are defined in the header `include/msg.h`, and are implemented as system calls (Section 3.3.6. All together, these form the message passing algorithm, which allows user tasks to communicate between each other. For our discussion we will denote T_s and T_r be the sending and receiving task respectively. For the sending and receiving sections, we will discuss the algorithm in two cases, depending on which task executes first.

First, suppose T_s executes first. When T_s executes `Send`, it blocks until after T_r has called `Reply`, which is accomplished by the kernel moving T_s's ready state from `READY` to `SEND_WAIT`. It then attaches T_s to the back of T_r's waiting queue, which is implemented as an intrusive linked list, and is separate from the task scheduler (i.e. tasks stay on the scheduler queue even if they are in a task's waiting queue). Note that the kernel can easily determine that T_s was called first by checking the ready state of T_r (i.e. in this case, T_r will be `READY`, rather than `RCV_WAIT`). Next, when T_r calls `Receive`, the kernel knows T_s went first since its waiting queue is not-empty, and pops T_s of the waiting queue. It then moves T_s to state

`RPLY_WAIT`, and copies the necessary bytes (i.e. the minimum) between the given buffers, and returns the necessary information as in any system call. Note that since T_r needs to handle the receive, the kernel does not block it.

Secondly, we will consider the case for when T_r executes first, which the kernel can recognise if there are no waiting senders. In this case T_r is blocked by the kernel moving it to the `RCV_WAIT` state. Then, when T_s sends, the kernel blocks it and transitions it to state `RPLY_WAIT`, and moves the data as before. However, in this case, the kernel has to unblock T_r (i.e. ready state to `READY`) before returning the necessary parameters as previous. As mentioned previously, the kernel is able to check that T_r executed first by verifying that T_r is in `RCV_WAIT` state.

In either case, T_r processes the sent message, and needs to `Reply` to T_s. The kernel handles this by copying the appropriate number of bytes, and unblocks T_s by returning it to the `READY` state.

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 handled 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 (front) and the last element (back) 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, running, exited, send-wait (blocked), reply-wait (blocked), receive-wait (blocked));
- 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).

- An intrusive pointer to the next task which is waiting to send to the given task, which forms a waiting queue. See Section 3.3.7.

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.

3.4.3 Messages

3.4.4 Name Server

On startup, the kernel creates the name server before any other process with TID 1. The main loop of the nameserver consists of a simple receive-process-reply loop. The nameserver saves the entries in a storage array of length 256, but maintains a bucket-hash map on top of the storage array for efficient get and insert operations.

On each register request, the server computes a hash on the name and inserts it to the back of the linked pointer list in the given hash bucket. The hashing algorithm is a simple and efficient dbj2 algorithm, which provides good distribution of hash values.

Respectively, on each `Register` request, the server hashes the requested name and maps the name to one of the 64 buckets. Each bucket contains a linked list of pointers (to the corresponding storage location in the storage array) of the entries hashed to this bucket. In most cases the algorithm only iterates a small number of entries, and if it finds an entry of the same name, it overwrites it. Otherwise, it adds it to the linked list.

The `WhoIs` method works similarly, by calculating the hash value of the given string and searching the list for a matching name.

3.4.5 Game Server

The Game Server enables pairs of clients to play Rock, Paper, and Scissors against each other. On initialisation, the game server registers its TID with the nameserver and transitions to its main loop, waiting for clients to send signup requests. Whenever the server receives a signup request, it adds the clients to a queue, implemented as a circular array.

When two players are in the queue, and there is no active game, it dequeues two players from the buffer and sends a reply, asking for their move. The active players' status and current move are stored in a local struct to keep track of the ongoing game. After it has received their moves, the server evaluates the outcome and waits till they either send their next move or a quit message.

On receiving a quit message from one client, it will reply at the next opportunity with a corresponding quit message to the other client. Afterwards, if possible, it dequeues two

players from the buffer or waits for additional signup messages.

Furthermore, when receiving an invalid request, the server replies with an error message so the client is not trapped in a waiting state.

4 User Tasks

4.1 Kernel Assignment 1

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.

4.2 K2 - RPS Test

4.3 K2 - Performance Measurement

In order to measure performance as accurately as possible, we borrowed the structures used to measure time in A0 and declared them globally in order to share them between sending and receiving tasks. That way, the whichever task (i.e. sender or receiver) that went first started the clock, and once the sender returned, the clock was stopped. This means that

the timer overhead is incredibly small, since the number of instructions between accessing the clock registers and the execution is very small. The following description of the timing algorithm is adapted from Trevor's A0 documentation.

> In order to perform and report times for the documentation, the `timer_stats` structure (`user/includes/timer_stats.h`) is used to calculate the average and maximum length of time for multiple timers between start and stop. To avoid keeping a huge number of arbitrary number of times in memory, the average is calculated accumulatively, that is, the total average is calculated using the average of the previous values. This is because given values $a_1, \ldots, a_n$, we may calculate the average recursively as follows:
>
> $$\overline{a}_n = \frac{1}{n}\sum_{i=1}^{n} a_i = \frac{1}{n}\sum_{i=1}^{n-1} a_i + \frac{a_n}{n} = \frac{n-1}{n}\left(\frac{1}{n-1}\sum_{i=1}^{n-1} a_i\right) + \frac{a_n}{n} = \frac{(n-1)\overline{a}_{n-1} + a_n}{n}.$$
>
> Thus, we only keep the total number of times added and the last average to calculate the total average recursively, and save space.

In order to reduce variability, for each test situation we execute it 1000 times and take the average, using the structure above.

Optimisation Enabled?	First Process	Message Size	Time per SRR Operation (μs)
N	Sender	4	207
N	Receiver	4	208
N	Sender	64	522
N	Receiver	64	592
N	Sender	256	1528
N	Receiver	256	1824
Y	Sender	4	78
Y	Receiver	4	78
Y	Sender	64	111
Y	Receiver	64	112
Y	Sender	256	215
Y	Receiver	256	237

Figure 1: A2 Performance Testing Results

The values in the `PerfTesting.csv` file are reproduced in Figure 1. From these test results, we can see first and foremost that as the message size increases, the average time per SRR operation increases approximately linearly. This makes sense because memory accesses are expensive. We can see also that enabling optimisation (i.e. compiling with the `-O3` flag) significantly reduces the operation time, and more crucially, reduces the impact on operation time due to the increase in message size, as we saw without the optimisation. This is likely the compiler removing repeated reads/writes and using register caching to speed performance up.

Secondly, we can consistently see that when the receiver starts first, the execution time is slightly longer, and this appears to increase as the message size gets larger. Part of this is because there needs to be an extra context switch back to the receiver when the receiver sends first in order for the receiver to reply to the message.