CS 452 - K1

Team Members

Matthew Geng
m3geng
20820439

Erica Wang
e52wang
20825331

Repository: https://git.uwaterloo.ca/mg-cs452/cs452

Commit SHA: dfa51f05335adacba9d7e9aa26e745dc524f5d9e

Kernel Design:

Our kernel design is based on a microkernel where code is non-interruptable and synchronous. System calls are kept short and fast by doing as little as possible on the kernel side.

The kernel is comprised of the following core components: a scheduler, a context switcher, an exception handler, and an exception code handler.

Before diving deep, we'll look at our kernel's high-level flow.

Kernel Main

After kernel initialization, the **scheduler** finds the next task to run based on task priority. The kernel **context switches** to that task, saving all the required information about the current kernel state and loading the task state. Afterward, either once the user task finishes or makes another system call, we hit our exception handler. The **exception handler** does the opposite of the initial context switch by saving the task state and loading the previous kernel state. The kernel resumes where it last left, and we appropriately respond to the exception code with our **exception code handler**. The code handler takes the exception code and executes various functionalities such as creating new tasks, yielding, finding information, etc based on the exception code. This process repeats from the top until there are no tasks to run.

Scheduler

The scheduler is a generic priority queue that takes in a fixed-sized array and a custom comparator. In our case, the fixed-sized array is an array of task frame pointers, and a function to compare task priorities.

The priority queue is implemented as a min heap to provide O(log(n)) operations for pushing and popping the minimum element. These time complexities are guaranteed from a binary heap implementation with siftup and siftdown operations.

Since task priorities go from high to low starting at 0, a min heap suited our use case. For tasks that have the same priority, the timestamp of when a task is added to the queue is used for tiebreaks, where the oldest have a higher priority. This results in a FIFO queue for tasks with the same priority.

Context Switcher

The context switcher is an assembly function that stores registers for the kernel and loads registers of the scheduled task.

It begins by grabbing the address of the kernel task frame. Registers x0-x30 (note x30 is the link register) and the stack pointer are stored using offsets from the address of the kernel frame with help from C struct memory alignment. We don’t need to store the pc, or any special registers such as elr_1, spsr_1, etc in our kernel because these values are specific to an individual user task.

Afterward, the address of the current task frame is loaded. Registers x0-x30, the stack pointer, the pc, and spsr have been set from task initialization or a previous system call, and are loaded into their respective registers.

Specifically for loading our task, we store wherever a task is supposed to resume into elr_el1 in such that when we do an exception return, the exception return will jump to the correct spot. To illustrate, for the first context switch, we store the task’s function address in elr_el1.

For other special registers:

We store the user task’s SP into sp_el0 because since we are in the kernel, using “sp” will refer to sp_el1.

Lastly, we also set spsr_el1 to the correct bits such that SError and IRQ bits are masked and the sp_el0 bit is set

Exception Handler

The exception handler is composed of 2 parts: the exception vector table and a c function. As part of kernel initialization, the VBAR (vector base address register) is populated with the start of our vector table label so that for every exception, we have a defined exception handler.

As a result, from a system call using SVC, we've defined a c function for that specific exception to jump to.

In that c function, we enter our kernel again and do the opposite of what our initial context switch does.

We store all registers x0-x30, sp, elr_el1, spsr_el1 into our current task frame, and load our previously saved kernel frame x0-x30 and sp registers.

Exception Code Handler

The exception code handler is a group of "if" statements that check for various exception codes immediately after the exception handler and return to the kernel main. Depending on the type of exception, such as a create, we allocate a new task frame, initialize it, and add it and the original task to the scheduler queue. Furthermore, for exception types that need to pass in certain arguments, registers x9 and x10 are used and for returning values, x0 is used. Note the reason why we are currently not using x0-x7 for parameters is because intermediary function calls that overwrite x0-x7. In order to compensate for this, we would have to revamp how we store passed in arguments, which is something we will look into for the next assignment. However, for our current use case other scratch registers x9-x18 satisfy our current use case.

Memory Layout

.text.boot
.text.evt (exception vector table)
.text (code)
.rodata
.data
.bss
start of user task stack base

Memory Allocation/Task Frames

There is no dynamic memory allocation in this kernel at this point. All memory allocations are done at initialization. The most “dynamic” structure we have is our array/list of task frame descriptions.

Each of our task frames are given a base stack address to represent their stack. In order to have contiguous memory and decouple from the need of a memory allocator, each of these task frames are intrinsically linked to the next task frame through a field which is a pointer to the next task frame.

Instead of utilizing the stack directly with push and pop operations in order to save user and kernel registers, we have opted into storing our registers with a global pointer to a memory address, representing our task frame, on the kernel stack. We have done this in order to more effectively debug our code by being able to examine specific registers globally.

Kernel Initialization

Our kernel initialization comprises of the following steps

- Initialization the base of our exception vector table
- GPIO + uart initialization
- Kernel task frame initilziation and setting global pointer to kernel task frame (usage in context switching)
- User task initializations (setting TID’s and stack base addresses)
- Scheduler (heap) initialization
- First task initialization

System Parameters

- Maximum of 20 user task frames (excluding task frame for
    - After taking a look at our future use case of a train system, we estimated each train would need ~5 tasks (controller, input, output, sensor, etc) which at most we’d probably have ~3 trains at a time. This is a total of 15 tasks, which leaves 5 other tasks for our kernel including our clock and other items. This was a lower bound estimate.
- Kernel stack base address is 0x7FFF0
    - To mimimize potential kernel stackoverflow issues, we’ve ensure that our stack can “grow” downwards until 0x0, such that when we do reach stackoverflow, we should be able to handle that promptly or automatically through a hardware error.
- Each user stack size is 1024 bytes
    - This was chosen as a rough estimate by checking the runtime stack size of our kernel which was < 1024 bytes. As a result, we thought this would be a decent start for our user task size

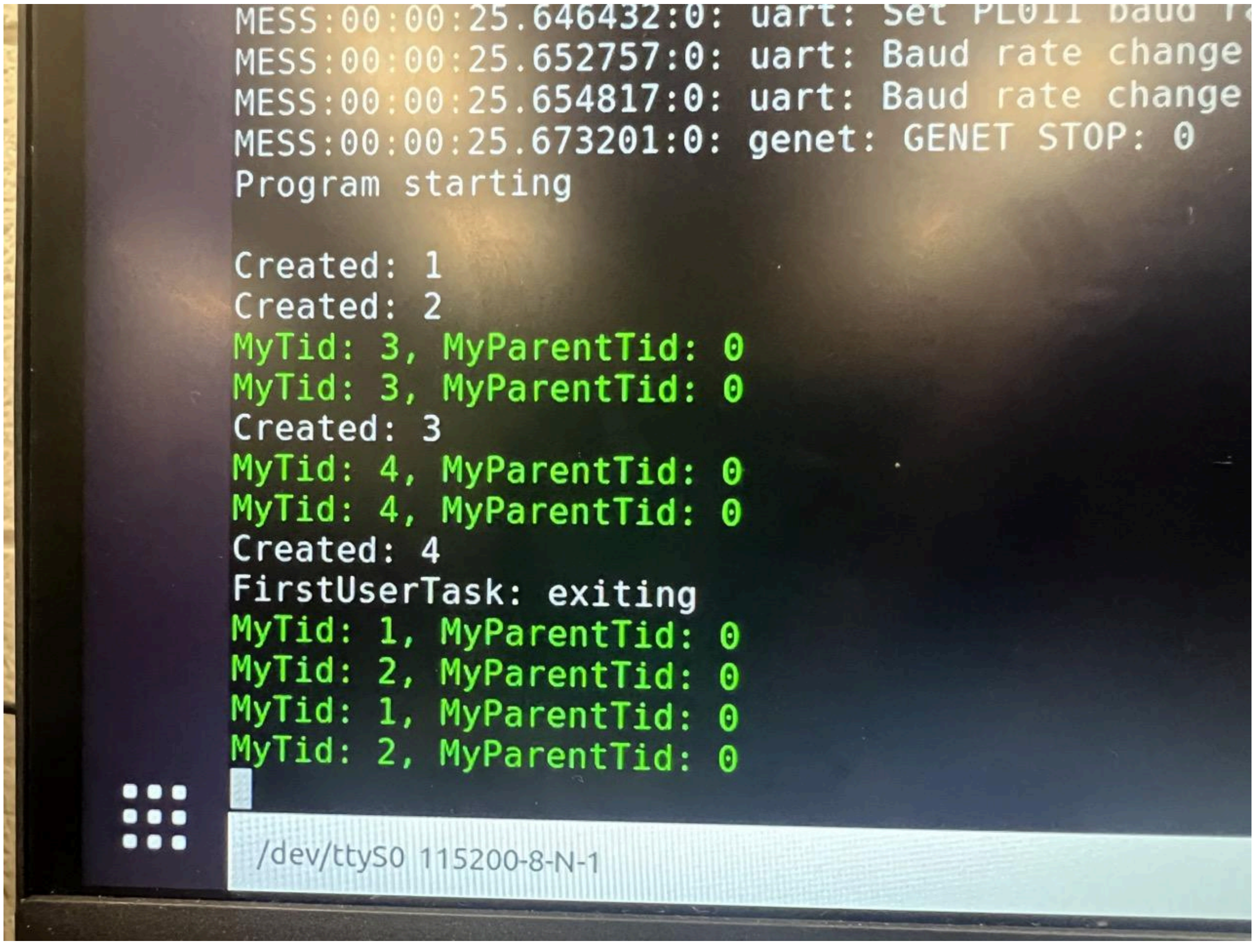
- First user stack base address is 0x8D9A0 (grows upwards for every base stack address, so next stack base address would be 0x8DDA0 that grows downwards)
    - With our current stack size, the lowest address a user stack “should” touch is 0x8D5A0 which is far higher than our highest piece of memory defined in our linker, bss, which ends at 0x82f88. As a result, we this stack base should be adequate for our usage especially for A1.

Bugs and Limitations:

- Creating more than 20 task frames (without reclaiming a task frame)
- No check for stack overflow for kernel or user tasks

Output Explanation:

Output:



```
MESS:00:00:25.646432:0: uart: Set PL011 baud ra
MESS:00:00:25.652757:0: uart: Baud rate change
MESS:00:00:25.654817:0: uart: Baud rate change
MESS:00:00:25.673201:0: genet: GENET STOP: 0
Program starting

Created: 1
Created: 2
MyTid: 3, MyParentTid: 0
MyTid: 3, MyParentTid: 0
Created: 3
MyTid: 4, MyParentTid: 0
MyTid: 4, MyParentTid: 0
Created: 4
FirstUserTask: exiting
MyTid: 1, MyParentTid: 0
MyTid: 2, MyParentTid: 0
MyTid: 1, MyParentTid: 0
MyTid: 2, MyParentTid: 0
```

Lets walkthrough the output. At the beginning of our kernel, we create a root task that creates these 4 sub tasks.

Initially, our root task is at priority 2. The root task first tries to create its first task at priority 3. Since priority 3 is less than our root task's priority, we continue and get our first line of output

```
Created: 1
```

The root task creates a new task at priority 3 as well. Since that is also less than our root's priority, we again continue and get out second line of output with a new TID.

```
Created: 2
```

The root task now creates a new task at priority 1. Since that is greater than our current root task's priority, we are pre-empted and the new task runs to get our new line of input.

```
MyTid:3, MyParentTid: 0
```

At this point, the task yields control back to the kernel. However, since it has the highest priority, it is rescheduled and re-prints.

```
MyTid:3, MyParentTid: 0
```

Once the task is finished, it exits, which allows us to context switch back into our root task to get our next output line.

```
Created: 3
```

Simialr to task 3, the root task now creates another task at priority 1. Since that is also greater than our current root task's priority, we are pre-empted and the new task runs to get our next line of input.

```
MyTid:4, MyParentTid: 0
```

Afterwards, the task yields and is rescheduled due to its high priority of 1 causing us to come back to this task and print the next line.

```
MyTid:4, MyParentTid: 0
```

Afterwards, task 4 exits and the root task is rescheduled and outputs.

```
Created: 4
```

Now, since the root task is finished, the root task exits and prints the following.

```
FirstUserTask: exiting
```

However, we are not done yet as we have two tasks, 1 and 2, still in our queue due to their low priority. Since these two tasks are at the same priority, the timestamp of when they are added to the queue is used to tiebreak these tasks. Because we always scheduled the oldest first to provide FIFO behaviour, task 1 should run and does.

```
MyTid:1, MyParentTid: 0
```

Task 1 yields and is added back to our queue. However, unlike tasks 3 and 4, task 1 has the same priority as task 2 and using our timestamp tiebreak, task 2 is scheduled instead by being the oldest task.

```
MyTid:2, MyParentTid: 0
```

Task 2 now yields, and using similar logic as above, task 1 is now scheduled.

```
MyTid:1, MyParentTid: 0
```

As this task now exits, the last task to be scheduled is task 2.

```
MyTid:2, MyParentTid: 0
```