CS452: Real-Time Programming

Kernel (Part 1): Context Switch and Task Scheduling

January 25, 2020

Hugh Pan
20868342

Leanora McVittie
20513649

Usage

The git repository can be found at https://git.uwaterloo.ca/lmcvitti/trains_kernel, and the exact commit hash will be emailed to Professor Karsten; it can also be found as the final commit on the k1 branch in that repository.
The executable file, `microkernel.elf`, can be compiled by cloning the above repository and then running `make` in the root directory. It can be run by loading it into RedBoot and using RedBoot's `go` command. No further input from the user is necessary after starting the program; it will execute and exit on its own.
A pre-existing executable `microkernel.elf` can be found in `/u/cs452/tftp/ARM/y247pan/` and can also be loaded into RedBoot and run.

Kernel Components

Our kernel consists of five main components, each of which is discussed in more detail below.

1) Set-up procedures that ensure the system is initialized in the correct state
2) A main event loop which schedules the tasks, yields to the next scheduled task, and then processes the result from that task
3) The set of task descriptors to keep track of the tasks in the system
4) A scheduling system
5) Two context switch functions, for switching into and out of kernel (supervisor) mode

Set Up

All of the setup code is within `src/kernel/bootstrap.c`. Currently the setting-up process starts by initializes the UARTs so they can be communicated with. This uses the sample code from A0. The software interrupt handler is defined to be the `enter_kernel` function, which will be discussed in more detail in the context switch section. The kernel state struct is a global variable which is allocated during the loading process in the data section of memory, therefore during setup it is initialized to its starting values to ensure there is no accidental reading of garbage data. Finally the first user task is created and added to the ready queue.

Main Event Loop

Immediately after setup is finished, the main event loop of the kernel begins. This loop continues to execute until there are no tasks in the ready queue to be run, at which time the program will exit. The code for this loop can be found in `src/kernel/kernel.c` in `k_main()`.
The loop calls the scheduler to get the next task to be run, runs that task until an interrupt is triggered, and then acts on the result code that is passed from that task. That code can tell the kernel to create a new task, return the just-run task's id, its parent's id, or exit that task. The

kernel can also pass data back to the task by overwriting the location on user's stack that will be re-loaded into `r0` the next time that task is scheduled.

Task Descriptors

The task descriptors are allocated within their own memory location in the kernel space. The task descriptor of a task can be found using the id of that task by taking that base memory location and then subtracting the size of the task descriptor times the id of the task plus one. In this way, the task descriptors behave like an explicitly-managed array of arbitrary (to a point) size.

Scheduling

Tasks are scheduled by their priority and then by the least recently run, with newly-created child tasks of the same priority as the parent scheduled before the parent next runs.
At the moment, scheduling uses one "priority queue" to manage the tasks that are in the ready state. Although no other queues are used at the moment, the same system will be easy to apply to them as well once they're needed in the next kernel assignment.
The ready queue is modeled as a binary heap, where the root node is the next task that will be scheduled. This heap is implemented as an array where a node at index `i` has child nodes at indices `i*2` and `i*2+1`, and functions are implemented to keep the heap sorted when adding or removing nodes.
When the kernel needs to schedule a new task, it removes the root node from the ready heap, and then resorts the heap so the next task to run becomes the new root node. Then, after the task runs, it is put back onto the ready queue, and the heap again ensures that it is sorted correctly.

Context Switch

There are two context switch functions, both of which are defined in `src/kernel/swi.S` and both are written in assembly. One, `enter_kernel`, is used to enter kernel mode, and the other, `leave_kernel`, is used to leave kernel mode (a true triumph of naming!). They are mirror opposites of each other.
The user process enters the kernel by calling `enter_swi` with one argument (`&args`) which is the address of the struct `Args` which stores all the parameters needed for the kernel to determine the call type, and is stored in `r0` when entering the kernel mode. This is the data structure that is used for passing information from the user tasks to the kernel. Through the context switching process, we ensure that the register holding `Args` never gets overwritten at any point so that it can be successfully transferred between the tasks and the kernel. Upon leaving the kernel, we overwrite the value in the user stack that will later be reloaded back to `r0` as the return value to the user task.

When entering kernel mode, supervisor mode is switched to, then the current registers are stored on the user stack, the values that were stored on the kernel stack are loaded into the registers, and the kernel resumes execution where it left off.
When leaving the kernel, the current registers are stored on the kernel stack, the values that were stored on the user stack are loaded into the registers, user mode is switched back into, and the user task resumes from where it left off.
User tasks can trigger a software interrupt by calling `enter_swi`, then entering kernel which is defined as triggering the `enter_kernel` function, which switches to kernel execution. And the kernel hands off to a user task by calling `leave_kernel` directly.

Memory Layout

The memory has two main sections. The kernel stack is assigned the highest chunk of memory, with the stack pointer starting at `0x2000000` and allocation ending at `0x1F00000`, and this is where kernel task descriptor resides. We define the maximum number of task descriptors to be `256`, and we allocate enough space to the task descriptors for all of them to be active at the same time. Each user task is allocated its own stack of size `0x10000` (64k bytes), starting at `0x1F00000` and decreasing for each additional task that is created. We do not recycle a task's id or its stack at this moment, which means an id and stack space can only taken by one task, and cannot be reclaimed for future tasks after the original task has exited. If this isn't changed it may limit the total operating life of the system.

User Tasks

Our program also contains the example task code. The first task starts four identical child tasks, two with a lower priority than the parent task, and two with a higher priority. Note that we have inverted the typical priority system for operating systems: in our kernel a priority of `0` is the lowest possible priority a task can have. These tasks are discussed in more detail in the next section, which goes through the output of the program.

Program Output

The full output of the program is as follows:

```
1     Task Id <0> Parent Task Id <-1>
2     Created <1>
3     Created <2>
4     Task Id <3> Parent Task Id <0>
5     Task Id <3> Parent Task Id <0>
6     Created <3>
7     Task Id <4> Parent Task Id <0>
8     Task Id <4> Parent Task Id <0>
9     Created <4>
10    FirstUserTask: exiting
11    Task Id <1> Parent Task Id <0>
12    Task Id <2> Parent Task Id <0>
13    Task Id <1> Parent Task Id <0>
14    Task Id <2> Parent Task Id <0>
```

Analysis

Let's look at this output line by line:

Line 1:
The first user task (hereafter referred to as the parent task) immediately prints its own task number: 0, and it's parent's task number: -1. The parent's task number is -1 because the parent task is the kernel, which does not have a designated task number, so -1 is used as a placeholder.

Line 2, 3:
The parent task has at this point created the first child task which has a lower priority than the parent task. Because the parent task has the higher priority, it is scheduled next, and returns to print line 2 before the task it created runs. This exact same priority situation causes the second child task to also be created but not executed, and then execution returns to the parent task so it can print line 3.

Line 4,5:
The parent task creates the third child task with a priority higher than itself. Therefore, when scheduling occurs after the kernel has initialized that child task, it is scheduled to be run next. It gets its own id and it's parent's id, two re-scheduling events in kernel mode, and then prints Line 4. It then yields to the kernel, and prints Line 5, all without interleaving with another task, because it is the task with the highest priority throughout all the switching to the kernel.

Line 6:
The third child task has now exited, and the parent task once again has the highest priority and resumes execution, printing that it has created the third child task

Line 7, 8, 9:
The exact same thing happens with the fourth user task as the third user task, again leading to the fourth user task returning to kernel mode and getting rescheduled three times without any other task having a chance to run. In the end execution is returned to the parent task after the fourth child task exits.

Line 10:
The parent task has now created all its tasks and its execution is now complete

Line 11, 12, 13, 14:
The two remaining child tasks have the same priority, and the same three re-entrances to kernel mode with their chances for rescheduling. Because they are both on the ready queue and they both have the same priority, their executions are interleaved, and therefore their print statements are too.