CS 452
Kernel 1

Mark Rada (`marada`)
Nikolas Illerbrun (`nillerbr`)

Overview

Our kernel implements a simple, constant time scheduler and task descriptor system. The kernel handles system calls from tasks via software interrupts and can run up to 64 tasks concurrently at 32 priority levels.

Operation Instructions

A pre-compiled kernel exists at `/u/cs452/tftp/ARM/marada/k1.elf`, which can be loaded with default RedBoot command given in the course tutorial document:

```
load -b 0x00218000 -h 10.15.167.4 ARM/marada/k1.elf; go
```

The source code for the kernel exists in `/u3/marada/kernel1/`, and can be compiled with the following command chain:

```
cd /u3/marada/kernel1 && source /u3/marada/.cs452 && make clean && make
```

Which will produce a `kernel.elf` file in the same directory as the makefile.

The kernel and tasks consist of the following files:

Submitted Files

File	MD5 Hash
Makefile	c21bcb0341cd75fa9c8c73ba03260730
Rakefile	0eea2f177a467cebb89d6576a093d9a0
include/benchmark.h	582da5edd39f4be7f887b8464fdc54ca
include/circular_buffer.h	81e4baa79597efcc8ec89ed39c49daa5
include/clock.h	91be07dae60791fe056a541fbe7de893
include/debug.h	092cb7487511e3a87ced390014fc7103
include/io.h	c2bba0b20b2f6f1fc39f2c5b86e669a2
include/kernel.h	4a6d4be03ea5c42ad2200443101f393a
include/limits.h	23591f861c7138cddccb09448722b0af
include/math.h	a69371e32455303f58ccbd206b3be889
include/memory.h	9e31beee60b6a12ad7507eec766c2bf4
include/parse.h	066730d5da3f8b85c6f8401d44495dc9
include/scheduler.h	99aa1eb05485967ad954880442857e95
include/std.h	44d2c363d461a5a3e1ea2dc18070e914
include/stdarg.h	2107acfd362fb7033d0e1d03ad1955cd
include/syscall.h	9734d1ec435041c56ec644accf2354bb
include/task.h	d97b1ec4ee9ecf726e2c7f7ef48ed42a
include/tasks/a1_sub.h	4d261d38e292340a81559ca31d8000b6
include/tasks/a1_task.h	a5c6d07c4ec323af237445dc96ab38bb
include/tasks/pass_test.h	d23fa7838d48543ab034562ddb2023ae
include/tasks/task_launcher.h	0e64b23a14c633ffae41e366d09831d8
include/trains.h	63e837e87e02b25214b553ddc6f902e6
include/ts7200.h	051c699e70f9b70a3f9f90d4ba2857f3
include/vt100.h	b8dce1ed48bbe4896ebe2baa126320e8
orex.ld	ead048bf38990d0ebe9538c793be5d62
src/circular_buffer.c	fc6039c05f0b5ef8742c456351505388
src/clock.c	51bace2acedbe9c3bbd62cca0d0159f9
src/context.asm	73edb16c8a138671ade75a26254208f3
src/debug.c	3be77481d6742c09d81c39f5a5e4a465
src/io.c	acf510f6f614eed39b7a11ed09cf8790
src/ksyscall.c	5f723769a334e9043012d82d83a19219
src/main.c	554d19fb138873dab1258ffd57525b6c
src/memcpy.c	9713c08f400fc7761ce33a7de47d0902
src/memory.asm	8bcb9712e74f51a38f4a8f392c5c9759
src/parse.c	886048b905add653996a7ca0ad524934
src/scheduler.c	673f2a53c1f4ccc5ca5443b2425e95f2
src/std.c	1b7db5182ce4ceff34cc6d0398c2aff7
src/syscall.c	235a60626360eadbba4c5cb1ec07cb64
src/task.c	29f88050c9a520955a73acef8e2b51c4
src/tasks/a1_sub.c	64166d8870bb5fd2afefe6bab4542c13
src/tasks/a1_task.c	732e44dbcb848310b00f567da3d996f4
src/tasks/pass_test.c	973040c5aaad83ad00dd95a09a1665b2
src/tasks/task_launcher.c	ee506c53e7ad60773c91f7d4b640663e
src/vt100.c	3fc34ce45cd8e849184d8f35e756a223

Kernel1 Task Output

When the kernel boots up, a very simple shell, simply named “Task Launcher”, is run as the initial task. Pressing the `1` key on the keyboard will run a task that performs the operations outlined by the kernel1 assignment specification. Other tasks, meant for profiling and testing the kernel, can be run using some of the other keys. Pressing `h` will list all other available commands.

The output on screen should look like the following:

```
1: Welcome to ferOS build 945
2: Built May 26 2014 12:09:44
3: Welcome to Task Launcher (h for help)
4: Created: 2
5: Created: 3
6: Id: 4 Parent: 1
7: Id: 4 Parent: 1
8: Created: 4
9: Id: 5 Parent: 1
10: Id: 5 Parent: 1
11: Created: 5
12: First: exiting
13: Id: 2 Parent: 1
14: Id: 3 Parent: 1
15: Id: 2 Parent: 1
16: Id: 3 Parent: 1
```

Lines 1-3:
Standard output generated by the initialization sequence of the kernel.

Lines 4-5:
The main task reports that it had created sub tasks with task identifiers 2 and 3 respectively. Since these tasks have a lower priority than the main task the kernel continues to schedule the main task and holds the current child task in the ready queue of their respective priority level.

Lines 6-8:
The main task creates a subtask with task identifier 4. This task has a higher priority than the main task causing the kernel to schedule task 4 to run on the next context switch to user land. Since task 4 has the highest priority in the system it runs until its execution is complete. After task 4 completes execution the kernel schedules the main task and the main task is notified that created task 4 and outputs the task creation message.

Lines 9-11:
Output is similar to lines 6-8 however the created subtask has a task identifier of 5.

Line 12:
The main task has performed all of its duties so it `Exit()`s.

Lines 13-16:
The first two tasks created by the now zombified main task are the highest priority tasks in the scheduling queue. Since these tasks are at the same priority level they will interleave their execution. This causes the output to flip messages between the two tasks until they reach the end of their execution. at this point all of the non-system tasks have finished execution and the kernel schedules the task launcher to execute once again.

Kernel Structure

The kernel is organized into three sections: task descriptors, scheduling, and system calls.

Task Descriptors

Task metadata is stored in a static array of task descriptors. We allow for up to 64 concurrent tasks. Each task descriptor contains the following:

- Task Identifier (`int tid`)
- Parent Task Identifier (`int p_tid`)
- Task's priority level (`unsigned char priority`)
- An index for the next scheduled task (`unsigned char next`)
- Some reserved space (`short reserved`)
- Task's stack pointer (`unsigned int* sp`)

The `tid` and `p_tid` are stored as `int` types in order to maintain consistency with required API for `Create()`.

`priority` and `next` are stored as `unsigned char` types, adding up to 16 bits, which necessitates the `reserved` space of size `short int` in order to keep memory explicitly aligned. We chose small sizes for those values to reduce the size of a task descriptor in the hopes of leaving more cache space for other things later. Explicit memory alignment is required by one of the many warning flags we have enabled for GCC, and would have implicitly been added to the structure when compiling with `-O2` or higher, which we are using. Currently, a task descriptor requires four words of memory, so we can fit two task descriptors on a single cache line with our given architecture.

Finally, `sp` is declared as a `unsigned int*`; we avoided using `void*` for `sp` because we want word sized indexing on the user stack for manipulating the saved task context.

Task `state` is not stored in a task descriptor because we believe that for the states; `ACTIVE`, `READY`, `BLOCKED`, and `ZOMBIE`; that our scheduling and allocation logic will never actually need check the state. This is certainly true at the moment, but if this turns out to not be the case we can use our reserved task descriptor space to store task state later.

Task Identifiers

Task identifiers are assigned based on the array index of the task descriptor. Each time that a descriptor is reused, the `tid` for the descriptor is incremented by 64. This allows us to calculate the array index for a descriptor in a single instruction given a `tid`. It also trivializes the need to make sure two tasks never have the same identifier.

Ensuring that task identifiers never repeat is not possible without infinite memory and computation time, however, our strategy for allocating identifiers does maximize the available number of identifiers given the constraints of the API. Furthermore, task descriptors are allocated in a round robin fashion, using a circular buffer for a free list, so that the entire 31 bit space of valid identifiers is used.

The `p_tid` identifier is set at task allocation time as that is the only time that the parent is guaranteed to be alive. The parent of a task is always the task that was active when we entered the kernel to handle the system call and so this value is trivial to find.

Priority Levels

Tasks have 32 priority levels. Level 0 is the lowest priority, level 31 is the highest priority. The range of priority levels fits nicely into a bit field that is one word in length. This allows us to optimize the scheduler selection process by checking for the highest set bit in the bit field, which can be done reasonably efficiently by calculating the integer logarithm with base 2 of the bit field (and subtracting 1). The range of levels is also more than enough for what professor suggested in class.

We found a constant time algorithm for calculating base 2 logarithms on the internet. Proper attribution is given in the source code. We also include a modified form of the algorithm optimized for short integers, should we ever need to reduce our priority range to 16 in order to save a few cycles of CPU time between context switches.

Priority Queues

The `next` index of the descriptor is used as part of the scheduler for maintaining the priority queue list. Storing the priority queue within the descriptor table saves memory space compared to creating individual queues for each priority level by exploiting the fact that a task can only be in one queue at a time.

Should we ever have other types of scheduling queues, such as for blocking, we can use the same `next` field of the descriptor, as the task will only be in one queue at a time.

The memory savings will hopefully let the cache perform better when it is eventually enabled by making more space for other things.

Task Context

The `sp` is a pointer to the user stack where the task trap frame is currently being stored. `sp` is the minimum runtime information that the descriptor needs to store. All other information is stored on the user stack in the trap frame.

When a task is created, we setup the initial stack for the user process so that if the task falls off the end (does not call `Exit()` explicitly), the `Exit()` function will implicitly be called. An initial CPSR value is also setup on the stack such that the context switch into the process will cause the CPSR to be correctly initialized. And, of course, an initial program counter is set for the task.

Scheduling

The task scheduler is implemented as a bit field, which maps bits to priority levels, and an array of `head` and `tail` pointers which are indices into the descriptor array. Technically, the rest of the priority queue is stored within the task descriptors as described above, but we do not duplicate explanations here.

When a task is scheduled, the scheduler is given the descriptor table index so that the descriptor can be looked up.

Using the descriptor, the `priority` level can be looked up, and the correct bit in the bit field can be turned on. Then the correct pair of `head` and `tail` pointers can be looked up.

Using the values of the `tail` pointer, the descriptor that is currently at the end of the list can have its `next` pointer updated to point to the task being scheduled. If there is no other task in the queue, then the `head` pointer is set instead.

The task being scheduled will always have its `next` pointer set to `TASK_MAX`. `TASK_MAX` is set to 64, which is just outside of the valid descriptor index range. This value is checked for later when selecting the next task to activate as a signal that the queue is empty and should have the appropriate bit in the bit field turned off.

Activation Selection

When the scheduler is asked to find the next task to be activated during a system call, it will simply look for the highest bit set in the bit field to select the priority queue and look up the `head` pointer for that queue and then retrieve the correct task descriptor.

Then `head` is updated with `head->next`. If `head->next` is `TASK_MAX` then we know that the queue is empty and the bit in the bit field should be turned off. This means that we can be sure that the `head` pointer will be valid if the bit was turned on.

Activation

Once all pointers are updated, the `task_active` pointer, which is the index of the active task, is set to the correct task. Once this is done, a context switch will enter the newly activated task.

System Calls

Software system calls are invoked via the wrapper function `_syscall()`. The reason we use this is so we can ensure the the parameters are placed in the correct registers when the `swi` call is performed. Another advantage to this is that GCC will ensure that any registers `r0-r3` we that need going forward have their contents backed up.

When the user task is rescheduled, the return value from the system call will be in `r0`, which follows the correct calling convention.

The immediate value associated with the `swi` instruction is not used. Instead, we pass the system call number via `r0`. Additional arguments for the system call are stored in a `kernel_request` structure on the task's stack and a pointer to the structure is left in `r1`.

When the system call jump is performed the `spsr` and `lr` are placed into `r3` and `r2` respectively. After this all registers except for `r1` and the `sp` are saved onto the calling task's stack. The reason only `r1` is omitted is because the `r0` location is used by the kernel to write back return value of the system call and `r2` and `r3` now hold information required to restore the user back to its current state. After this the saved kernel registers are loaded back in, except for registers `r0-r3`, and then a branch link instruction calls `syscall_handle()` so the values placed into `r0-r3` can be used directly.

When `syscall_handle()` has completed execution the kernel will return to right after it performed the `scheduler_activate()`. When activate is called all of the above steps are performed in reverse, entering the currently scheduled user task.

Priority

We added an `int myPriority()` system call to allow tasks to query for their priority level. We imagine that some tasks will have the same priority level in every instantiation of the task, while other tasks may have a dynamic priority level for different instantiations. As a task is not given its priority level during instantiation, we needed a way to expose the information to tasks that were interested. The `myPriority()` system call has the same semantics as `myTid()` in that it will simply return a value stored somewhere in the kernel and never fail.

Parent Task Identifier

Since a task descriptor stores the parent task identifier at creation time, the semantics of `myParentTid()` in our implementation is that the call will never fail and will always return the correct task identifier for the parent task. Since we map a task identifier to a specific descriptor in the descriptor array, it will be trivial to check if the parent task is actually still alive for other system calls that will be added for message passing.

Task Structure

When the operating system has initialized the first task scheduled is a simple task launcher. This allows us to statically compile in a couple tasks that we can use for testing and run them all dynamically.

The task launcher is set to the lowest priority level so that it will only be scheduled when there are no other tasks in the system to be run. The Task launcher works by just calling the system call `Create()`.

The most significant difference between the Task Launcher and other tasks is that it is the first task, and so its parent task identifier will be its own task identifier.