CS 452
Kernel 1

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

Overview

Write something about the kernel (tasks) and using the Task Launcher here.

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 my program exists in `/u3/marada/kernel1/`, which has the following list of files:

Kernel1 Task Output

Running a task that performs the operations outlined by the kernel1 assignment specification produces the following output:

```
1: Welcome to ferOS build 899
2: Built May 22 2014 14:58:05
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 created by the initialization sequence of the kernel

Lines 4,5:
The main task reported that it had created sub tasks with ids 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 children tasks in the ready queue of their respective priority level.

Lines 6-8:
The main task creates a subtask with id 4. This task has a higher priority than the main task causing the kernel to this task to run next. 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 was notified it created task 4 and outputs outputs its task creation message.

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

Line 12:
The main task has performed all of its duties and will now call `Exit()`.

Lines 13-16:
The first 2 tasks created by the now zombified main task are the the highest priority tasks in the scheduling queue. Since these 2 tasks are at the same priority level they will interleave their execution which causes the output to flip between the two remaining tasks until they reach the end of their execution at which point all of the non-system tasks have finished execution. This will cause the kernel to then reschedule the task launcer.

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	8401546fb3d728a111265e029d12ce25
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	3089874a375d2dc72520dbc6e611c2f6
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

The program can be compiled with the following command:

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

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

Structure

The kernel is organized into three pieces: scheduler, tasks, and support.

Task Descriptors

Tasks metadata is stored in a static array of task descriptors. We allow for up to 64 concurrent tasks. Each task descriptor contains the task identifer `tid`, the parent task's task identifier `p_tid`, a priority level `priority`, a task index pointing to the `next` scheduled task, some reserved space `reserved`, and the stack pointer for the current task `sp`.

The `tid` and `p_tid` are stored as `int` types, while `priority` and `next` are stored as `unsigned char` types, which necessitates the `reserved` space of size `short int` in order to keep memory explicitly aligned. Finally, `sp` is declared as a `unsigned int*`.

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 is used.

The `p_tid` identier is set at task allocation time as it 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.

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 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.

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 and will hopefully let the cache perform better when it is evetually enabled.

The `sp` is a pointer to the user stack where the task trap frame is currently being stored. The `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.

Scheduling