Kernel 1
CS452 - Spring 2014
Real-Time Programming

Team
Max Chen - mqchen
mqchen@uwaterloo.ca

Ford Peprah - hkpeprah
ford.peprah@uwaterloo.ca

Bill Cowan
University of Waterloo
Due Date: Monday, 26th, May, 2014

Table of Contents

1 Kernel Structure

1.1 Memory Allocation

Memory allocations is created in a trivial manner. As in accordance with maintaing the performance of a real time system, we make use of the stack to allocate memory, treating Memory as a linked list of addresses which represent a segment (or block) of the stack. These segments make up the stacks of the tasks. On allocation, the head of the linked list is returned and the next segment becomes the head, while on free, the block is added as the head of the Memory linked list. Since Memory protection and segmentation are not required, memory was implemented in the simplest manner.

1.2 Task Descriptor (TD)

Tasks represent a thread or unit of execution. A task descriptor describes a single task and stores the following information:

Field	**Description**
TaskStat_t	the task state which is an enum in READY, ACTIVE, ZOMBIE, FREE
tid	the task identifier (these aren't reusued)
parentTid	tid of the parent task (task that created thsi one)
priority	integer value indicating priority of task (larger number $\rightarrow$ higher priority)
sp	the task's stack pointer
next	next task in the task queue this task belongs to
addrspace	pointer to the block of memory the task can use as it's stack
result	result of a recent kernel system call

The CPSR of the task is stored on the stack of the task alongside its registers. Further optimizations can be made to the task descriptor by storing the result on the stack of the process, as well as removing the `addrspace` pointer if a task is not going to call `Exit()`, but reaches the `ZOMBIE` state as a result of accomplishing what it was tasked to do and having no more instructions to run.

To create the task descriptors, since we do not make use of the heap, we have to allocate our task descriptors from a bank that already exists on the stack. To this end, we have an array of 32 task descriptors (an amount that will be tweaked depending on how much is needed in the future), that can be used to create new tasks. These tasks start as blank with their state marked as `FREE` denoting that they can be used to allocate new tasks. When a task is `READY`, it is stored in the `taskQueue` corresponding to its priority, of which priorities can range from $0 - 15$. To get constant performance when creating a new task descriptor, in the `initTasks` function, they are assigned to the `taskBank` as a linked list, each task descriptor pointing to the next free task descriptor in the list. When we wish to create a new task, we grab the task descriptor at the head of the bank as it will always be free to use. We then assign it a priority, task id, parent tid (if the parent exists), an address space to use by calling `getMem()` which returns a segment of memory for the task to use as its stack, and the task pointer which points to the bottom of the address space. The task is then added to the end of its respective queue. On deletion, we mark the state of the task as `ZOMBIE` and free its address space; for now we make use of the `ZOMBIE` state and do not add the task descriptor back to the head of the bank to adhere with assignment specifications, but in the future to allow for immediate garbage collection that would not hinder the performance of the system, the task would be added as a free task to the head of the bank.

1.3 Task Queues

Queues for tasks are implemented with two pointers; one pointer to the head of the queue and another pointer to the end of the queue. We use the tail pointer to enable constant time addition of a new task to the queue by simply having the tail task pointer to the added task as it's `next` field. The `head` pointer enables us to get the next task in the queue that is to be run and pop it off the queue. Queues are created from an array of queues on the stack, and each queue's index in the array corresponds to the priority of the tasks that are stored in that queue.

1.4 Scheduling

On a `schedule()` call, the following sequence of events takes place:

1. If the queues are empty, or there are no tasks of higher priority, return the current task (i.e. the last task that was running).
2. If the current task is null, the kernel has no more tasks to run, exit.
3. Otherwise, add the current task to the end of the priority queue.
4. Get the next task descriptor from the priority queue.
5. Move that task into the `ACTIVE` state and return it.

This sort of round-robin scheduling means that each task in the queue has an equal opportunity at being run and means that if a task passes and it is the only task in the queue, it will simply be run again. There are 16 priorities $(0 - 15)$ for tasks, with 15 being the highest priority a task can have. Each priority has its own queue of task descriptors that are implemented as a linked list and tracked by

1. `highestPriorityQueue` - integer, the highest priority queue that is not empty
2. `availableQueues` - integer, bit field for the queues, 1 = non-empty, 0 = empty

These two variables are updated each time a call is made to `addTask` to add a new/existing task descriptor to the end of a queue, which occurs either during creation or scheduling.

Tracking the queue state allows for a constant time retrieval of the next task to run. When a `schedule()` call results in the last task being removed from the queue of its priority, then the corresponding bit in `availableQueues` is set to 0, and `highestTaskPriority` is updated to the next highest task priority by doing a pseudo-linear search of the bits in `availableQueues` to find the first occurence of a 1 bit. A binary search is run, with the search space between the last `highestTaskPriority` and 0. However, this binary search is inconclusive, in that it will only move the high index towards 0, and will break when doing so will make the high index lower than the next highest priority. Essentially, we want to find the first n such that n > next highest priority, but n/2 <= next highest priority. From here, a linear search is done counting down from n until the first non-zero bit of `availableQueues` is found.

This was done to give an optimization in the worst case (last priority was very high, and the next highest priority is low). A full binary search is not done since given the size of the search space, there doesnt seem to be much to be gained from doing so. Optimization seems premature at this time, so there was not much effort spent on it.

1.5 Software Interrupt (SWI)

1.5.1 swi_call

For each system call, the calling task creates a struct of type `Arg_t` on its own stack, then calls the assembly function `int swi_call(int sp, void *args)` to initiate SWI. The function passes a dummy value 0 into the sp parameter, and the address of the `Arg_t struct` it created as the second parameter. The `swi_call` function simply triggers the `swi` assembly instruction, and its purpose is mainly to offload some of the register handling for arguments to GCC.

1.5.2 swi_handler

This assembly function is the kernel SWI handler, and is the address stored in 0x28 to be jumped to for `swi`. The function first switches to system mode to save the user registers r2-r12, lr onto its stack, and at the same time saves the SP into r0. After switching back to SVC mode, the SPSR is also added to the top of the user stack (r0). Now that the user state is saved, the top of the kernel stack is popped into r2 (this is an output parameter, `Arg_t**`, supplied by the kernel) and r1 (`Arg_t*`) is stored into its address. Finally, the kernel state is restored by popping its stack into r3-r12, pc.

1.5.3 Request Handling

The `Arg_t` structure gives the code for which request is required, as well as the arguments necessary. Kernel functions are called based on a switch statement around the argument code, and the result is stored into the TDs result field.

1.5.4 swi_exit

This assembly function returns from the kernel SWI handler into the user code, as well as the return point back into the kernel from a new SWI call. The function has the signature:

```
int swi_exit(int result, int sp, void** tf);
```

This takes advantage of the fact that r0 is used as the return parameter. When calling into the kernel, the user SP is written to r0, which then becomes the return value of this function, passing it to the kernel. When returning to a user task, the result is passed as the first parameter so it resides in r0, and is treated as the return value of the `swi_call` function. The user SP is passed as r1 so the kernel can restore the user state by switching to SYS mode, and writing it to the user SP register. r2 is an output parameter for `Arg_t*` that is saved as a part of the kernel state r2-r12, lr so that `swi_handler` can write the user arguments to it. The function restores the SPSR from the user stack, then restores the user state. The processor is switched into system mode, r1 (passed in by the kernel from the TD) is written to the user SP, then the stack is popped into r2-r12, pc$\hat{}$. The `ldmfd` instruction, with the ôption and the PC as one of the registers, will do the load as well as move the SPSR into the CPSR, thereby switching from system mode to user mode.

2 Program Output

The program outputs the following from the user tasks, which is later followed by the login prompt:

```
Created: 1
Created: 2
My Task Id: 3, My Parent's Task Id: 0
My Task Id: 3, My Parent's Task Id: 0
Created: 3
My Task Id: 4, My Parent's Task Id: 0
My Task Id: 4, My Parent's Task Id: 0
Created: 4
First: Exiting
My Task Id: 2, My Parent's Task Id: 0
My Task Id: 2, My Parent's Task Id: 0
My Task Id: 1, My Parent's Task Id: 0
My Task Id: 1, My Parent's Task Id: 0
```

Task 1 is created with priority 2, task 2 with priority 4, task 3 with priority 6, and task 4 with priority 8. The first task is created by the kernel, has a priority of 5, a task id of 0, and creates tasks 1 to 4. Now, since the first user task has a priority of 5, after creating task 1, schedule() will schedule task 0 again because it's priority is 5 which is greater than 2. Task 0 will then print out having created 1 and will proceed to create task 2, where the same thing happens again as task 2 has a priority less than task 0. When task 0 then resumes again, it creates task 3, which has priority $6 > 5$, so on schedule(), task 3 is run instead. Since there are no other tasks with priority 6, the Pass() that occurs in task 3's code will resume control to task 3 again, and it will exit having printed two times it's task id and it's parent task's id. On its exit, schedule() schedules task 0 again as it has the highest priority of any tasks, which upon resuming it prints out having created task 3, and then creates task 4, where the same thing occurs as it did with task 3 as task 4 has priority 8 and is the only task with such a priority (greater than any other priorities). On return from task 4, task 0 is resumed and prints itself exiting before exiting as it has no more tasks to create. Task 2 and task 1 then run sequentially, task 2 printing out twice followed by task 1 as task 2 has the highest priority of any tasks currently created and no task with the same priority as it.

3 Additional Testing

3.1 Task Creation, Priority and Scheduling

To ensure that our scheduling was working correctly, we needed to create several tasks at differing and same priorities and ensure the output matched what we expected. To do this, first we outlined four different task codes, determined the priorities and code of the varying tasks, and went through a hand simulation of the scheduling and output. This gave us an answer to check against to ensure out scheduling was working properly. First, we created the tasks and threw an assert failure if the task failed to be created; this ensured that task creation was doing what it was supposed to be doing. Afterwards, the kernel main loop began and ran through scheduling. In order to determine which task finished when, the tasks recorded their finish times in a globally accessible array before calling Exit(). We compared these finish times against the expected finish times; since they matched, we could be assured that scheduling was functioning as expected. To ensure priorities were correctly affecting the queues, several tasks were created with the same priority and called Pass(); given that each task at the same priority calls Pass(), they would all finish in the same initial order. By creating tasks at higher priorities, it was ensured that scheduling() was correctly running those tasks first before the lower priority tasks.

3.2 Context Switch, Processor State

Looking at the generated user task .s files, most of them did not utilize that many registers. To be sure of the correctness of our context switch, we created our own tests that uses additional registers aggressively and interchanged them with context switching. We turned on gcc optimization, then found a small function which used a good number of registers - in our case, we picked `atoi`. The function was copied into the body of a task, with `Pass()` calls added into parts of its execution. After inspecting the assembly code, we verified that registers were used on either side of the `Pass()` call without being stored into memory. Multiple instances of the task was created with different values to convert to integer, then the result of the inline `atoi` were verifed against the hard-coded expected results.

4 MD5 Hashes

```
1efb2e496393dc5465abf0480940928d   /u7/mqchen/cs452/cs452-microkern/src/orex.ld
0f60a59fdd1e9a6cb4ec4aa01acc54ad   /u7/mqchen/cs452/cs452-microkern/src/mem.c
b59c60300d76a7443e5e4260f70a8232   /u7/mqchen/cs452/cs452-microkern/src/stdio.c
4a4a162d7de498e89ab976f72601e7ae   /u7/mqchen/cs452/cs452-microkern/src/string.c
070b92bb919655be671153cf336ac75d   /u7/mqchen/cs452/cs452-microkern/src/syscall.c
1b9af90c069360f5eed30ea3f6c8b80d   /u7/mqchen/cs452/cs452-microkern/src/task.c
719958ebaa876c06e82df6cb8e0cefd8   /u7/mqchen/cs452/cs452-microkern/src/term.c
9825389de9563fc632538c111d41e973   /u7/mqchen/cs452/cs452-microkern/src/utasks.c
394f7ed086c4465cc86fcf573d40a011   /u7/mqchen/cs452/cs452-microkern/src/main.c
5e4ef8a626c50e440f460ac0e76d9289   /u7/mqchen/cs452/cs452-microkern/src/stdlib.c
dc8da36f9c8167f47d71894bbca47655   /u7/mqchen/cs452/cs452-microkern/src/k_syscall.c
59bc24234c49f9c247bb97e4b125959c   /u7/mqchen/cs452/cs452-microkern/src/kernel.c

0e521943c728ffaabac44d3485ca4a1e   /u7/mqchen/cs452/cs452-microkern/include/types.h
742e69b52b3f4cda1898363ef1246fb0   /u7/mqchen/cs452/cs452-microkern/include/ts7200.h
40e6f5862869392d9733ea2d6defbb68   /u7/mqchen/cs452/cs452-microkern/include/bwio.h
```

```
d898edf77661ac9a98e2a0c6b9d9b9a6   /u7/mqchen/cs452/cs452-microkern/include/vargs.h
045d5b074001cbda021d83b6a76a0a84   /u7/mqchen/cs452/cs452-microkern/include/mem.h
19fbec18bdcd2bf2169e51577153fbcf   /u7/mqchen/cs452/cs452-microkern/include/string.h
a2b4f767be509e16fae1d8bb671cae87   /u7/mqchen/cs452/cs452-microkern/include/syscall.h
2868c18b136cc0dd88307aa3dafe59ea   /u7/mqchen/cs452/cs452-microkern/include/task.h
0d4f70ea95299ed99ce1169d4d19fdda   /u7/mqchen/cs452/cs452-microkern/include/stdio.h
19694c1d15b796e9f5ced7df853524c5   /u7/mqchen/cs452/cs452-microkern/include/stdlib.h
16a3912586e2c05d0d219c124a3565b1   /u7/mqchen/cs452/cs452-microkern/include/term.h
45219fc297d19330b76e5fc1b80e7727   /u7/mqchen/cs452/cs452-microkern/include/utasks.h
85461ecf5ec811924ee3035eadb8dfec   /u7/mqchen/cs452/cs452-microkern/include/k_syscall.h
afad1686254c89fe51cce7ce4fa1b3da   /u7/mqchen/cs452/cs452-microkern/include/syscall_types.h
87c6cd6bf280cb0a927e6c15a2ab8db6   /u7/mqchen/cs452/cs452-microkern/include/kernel.h

befff66a349448e4e08aab2f7abbaa51   /u7/mqchen/cs452/cs452-microkern/tests/tasks1.c

2111d426d0eafde7010394f25a13aab3   /u7/mqchen/cs452/cs452-microkern/Makefile

8402c682ff31b15549689baa00ea45b6   /u7/mqchen/cs452/cs452-microkern/lib/libbwio.a

8afa04fd4ff12bc483271286d52dfa00   /u7/mqchen/cs452/cs452-microkern/bin/cs452-upload.sh
923048dea1528d355b4bd2889d7cf1be   /u7/mqchen/cs452/cs452-microkern/bin/md5.sh
```