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, 26^{th}, 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. In `swi_call`, the user task registers r2-r12, lr are pushed onto its stack. Then the SP is copied into r0, and the `swi` instruction is triggered to switch to supervisor mode.

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 saves the SPSR onto the top of the user stack (r0) then pops the top of the kernel stack into r2 (this is an output parameter, `Arg_t**`, supplied by the kernel) and saves r1 (`Arg_t*`) into its address. Finally, the kernel state is restored by

moving r3-r12, pc into the registers.

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 SP is set to r0, which becomes the return value of this function. When returning to a user task, the result is passed as the first parameter so it resides in r0, and when we return to the user mode it already has its result. The user SP is passed as r1 so the kernel can restore the user state by switching to SYS mode. 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 user state and SPSR from the user stack, then MOVS to switch to user space. The next instruction is always going to be mov pc, lr (user space), so a constant label is used to branch to this instruction, and returns to where the user called `swi_call`.

2 MD5 Hashes

3 Program Output