Assignment K1

Edwin Zhang and Bill Cui

Path to executable

```
cs452-a1/build/kernel
```

Access, make, operate

```
git clone ist-git@git.uwaterloo.ca:b22cui/cs452-a1.git

cd cs452-a1

chmod a+x compile_target.sh

./compile_target.sh

cd ./build
```

Then, move the `kernel` onto the track computer

```
cp kernel /u/cs452/tftp/ARM/e42zhang

chmod o+r /u/cs452/tftp/ARM/e42zhang/kernel
```

To run the program, issue the following:

1. `load -h 10.15.167.5 ARM/e42zhang/kernel`
2. `go`

Operating Instructions

To run the program, issue the following:

1. `load -h 10.15.167.5 ARM/e42zhang/kernel`
2. `go`

Program Description

Asserts

Defined in `my_assert.h`, when an assertion fails, a sad train ascii art is printed onto the screen. Additionally, developers can add context about the failed assertion as part of the second parameter of `KASSERT`

Tasks

A TCB struct serves two purposes: 1. It is a free slab of memory that points to the next free slab of memory. 2. It contains information of a task.

There is a pointer to the next free slab of memory that can used to hold a new task. All the task memory slabs reside in a TCB array. This implements intrusive linkage and avoids the use of `free` and `malloc`, as all the memory that tasks will ever need is allocated on the stack as the `TCB[]` array.

Tasks contain the stack (an array of 32 bit integers), and its register struct. When the task is running, its register struct can also be accessed globally (and in assembly) as a global variable points to it.

Since the word size of ARM is 4 bytes, the stack is an array of `uint32_t`.

The stack size was chosen to be $2048 \cdot wordsize$. This allows for, theoretically, $\frac{32 \cdot 10^6}{2048 \cdot 4} \approx 3906$ tasks. In reality that is definitely not the case because available memory would have been used to store other data structures as well. Thus, each task has a stack size of $2048 \cdot wordsize = 8192$ bytes.

For K1, the maximum number of tasks allowed (`MAX_NUM_TASKS`) was chosen to be 10. This is because only a small fixed number of tasks are executed in this version of the kernel. This number will be increased for K2.

Scheduling

The scheduler uses a fixed size array based heap that stores the pointers of the TCB's. The actual contents of the TCB's are stored in the TCB array declared on the stack of the kernel's main function.

The heap is a max heap and thus allows for tasks with the highest priorities to be popped in $O(\log(n))$ time. For tasks that have equivalent priorities, the timestamp of when they were added to the queue is used as a tie breaker. This also ensures that the heap behaves like a FIFO queue when all the tasks have the same priorities.

Context Switching

The kernel register is a global struct, similarly the pointer to the register struct of the current task is also a global variable. This allowed for us to reference the register structs directly in assembly. Registers can be accessed and modified without needing to directly modify the task stack.

`switch_user` is used to store kernel registers and load the registers of the next task

`return_swi` is used to store the registers of the currently running task and load the kernel registers. It is the swi handler so it's memory address is stored at `0x28`, where the hardware uses as the swi vector.

The context switching implementation can be better explained by following through with a walkthrough of a sample program flow:

Upon creating and adding the first task, the kernel goes into a continuous loop. It first takes the task at the head of the ready queue and selects for it to be executed. The task register pointer `user_reg` is updated to point to the register struct of the chosen task. `switch_user` is then called which saves the kernel registers. In particular, the return address of `switch_user` is stored as the PC register in the register struct, so that when kernel state is restored the execution will resume past `switch_user`. The user task's registers are then loaded, and thus the user task continues execution.

When the user task performs a syscall, for example `Create(...)`, the arguments are placed on to `r0`. Then a software interrupt is triggered. PC is then set to `return_swi` by the hardware. User task registers are stored and the kernel state is reloaded. As mentioned earlier, since the PC of the kernel was saved to be the return register of `switch_user`, the kernel continues execution at the instruction right after `switch_user`. `switch_user` and `return_swi` combine to create the appearance that running the user task is as simple as calling a function. From the kernel's point of view, it has simply called a function and the function has returned what to do next. This luxury of simplicity enjoyed by the kernel is the materialization of the blood, sweat, and tears of assembly developers (me).

The arguments are then retrieved from the user task's `r0`. The type of the syscall is retrieved by retrieving the parameter of the `swi [...]` instruction that was executed by the user task. Since that instruction was the last instruction to be executed by the user task before the software interrupt, we were able to retrieve it by just decrementing the PC of the user task by 4. Once the system call has been serviced, the return result is stored in user task's `r0`. The user task is then placed back into the ready queue to be executed in the future. When the user task is selected in the future to continue execution, and once its registers and PC are restored, it would continue execution from the point after the `swi` instruction. The return result is stored in `r0`, which is retrieved and returned as a C function return. From the user task's point of view, it has simply called a function and the function has returned a value. This luxury of simplicity enjoyed by the user task is the materialization of the blood, sweat, and tears of kernel developers (also me).

Program Output Explained

```
Created: 1
Created: 2
Me: 3 Parent: 0
Me: 3 Parent: 0
Created: 3
Me: 4 Parent: 0
Me: 4 Parent: 0
Created: 4
FirstUserTask: exiting
Me: 1 Parent: 0
Me: 2 Parent: 0
Me: 1 Parent: 0
Me: 2 Parent: 0
```

The first task created is `task_k1init`. Tasks 1 and 2 created by `task_k1init` have lower priority than `task_k1init`. Therefore `task_k1init` is able to print `Created: 1` and `Created: 2` without interruptions. However once `task_k1init` creates task 3, which has higher priority than `task_k1init`, task 3 takes over the CPU. Even after making syscalls or yielding, task 3 is still brought back to the front of the ready queue because it has the highest priority. Therefore it is able to complete execution and print `Me: 3 Parent: 0`, `Me: 3 Parent: 0`. Once task 3 finishes execution `task_k1init` once again has the highest priority. It is able to continue execution and print `Created: 2`. Once `task_k1init` creates task 4 which has a higher priority, the same thing that happened with task 3 happens again. Task 4 is able to complete execution and print `Me: 4 Parent: 0`, `Me: 4 Parent: 0`. Once task 4 is completed, it is no longer in the ready queue and `task_k1init` can continue and print `Created: 4` and `FirstUserTask: exiting`. At this point `task_k1init` has finished execution and therefore only tasks 1 and 2 are in the ready queue. Tasks 1 and 2 alternate execution as they have the same priority and are moved to the back of the ready queue during syscalls and yields. The ready queue behaves like a FIFO queue in this case since both tasks have the same priorities. This is why we see:

```
Me: 1 Parent: 0
Me: 2 Parent: 0
Me: 1 Parent: 0
Me: 2 Parent: 0
```

Finally, both tasks complete and the kernel has no more tasks in the ready queue. The while loop terminates and the kernel exits.

Creating your own user tasks

Since user task's have their lr's initialized to call Exit(), there is no need for users to explicitly include `Exit()` in their user tasks.