CS 450 Assignment K0

1. How To Operate Program

The .elf artifact is committed in the git repository at [TODO]. See section 4 for a link to the code repsoitory. Alternatively see section 5 for instructions to build from source. Once the artifact is obtained, load it into redboot and run `go`.

2. Group Member Names

Xuanji Li

Lennox Fei

3. Kernel Structure

Context Switch

Context switch from user mode to kernel is done by the `SWI <n>` instruction, where `<n>` specifies which syscall should be made, and context switch from kernel to user mode is done by restoring a saved user mode CPSR register from memory which sets the processor mode to user mode.

At startup, the `setUpSWIHandler` handler function is used to install the `sys_handler` as the handler for `SWI`. This is done by writing the absolute address of `sys_handler` to `0x28` and the instruction `LDR pc, [pc, #0x18]` to `0x08`. `sys_handler` expects to be in svc mode, with the sp pointing to a trap frame with the saved kernel context. It

1. Get the exact `SWI` instruction and pass it to the kernel syscall handling system
2. Get the user mode sp and write it to the current TD
3. Restore kernel context from the trap frame

The kernel syscall handling system is responsibe for removing syscall arguments from the stack and doing the call (e.g. creating a new task) and writing the syscall return value into the task's memory.

When a syscall is made from the user side, the following steps are taken:

1. User context is saved on the stack as a trap frame, with a modified PC
2. Syscall arguments are pushed onto the stack
3. `SWI <n>`

Modified PC: the PC saved in the trap frame is the instruction after the SWI, instead of the PC at the time when the trap frame is created.

Within the kernel, a loop runs the next available task by switching from kernel to user. The SWI handler returns control to immediately after, i.e., into a new iteration of the loop. The switch from kernel to user does:

1. Save kernel context onto a trap frame, with modified PC
2. Restore user context (including CPSR)

Task Descriptors

Task descriptors are stored a struct with the TID, the parent TID, the task priority, the stack base, and the task stack entry. The stack base is declared as `char STACK[STACK_SIZE]`. Since the TDs have static storage (via the global scheduler variable), the compiler allocates `STACK_SIZE` worth of space within the struct.

Task Initialization

When a task is created (e.g. first user task or via `Create`), we initialize the memory by creating an appropriate trap frame.

Scheduling

The TDs are stored in a priority queue (keyed with task priority).

4. Code repository

https://git.uwaterloo.ca/f5fei/chos

5. Output and explanation

```
Created: 2
Created: 3
4 1
4 1
Created: 4
5 1
5 1
Created: 5
FirstUserTask: Exit
2 1
3 1
2 1
3 1
```

The first two tasks (TID 2 and 3) don't run until FirstUserTask exits, since they are of lower priority. The next task (TID 4) has higher priority and so completes execution before FirstUserTask gets scheduled again (wherein it prints the return code from the Create). FirstUserTask must get scheduled again in order for the netx task (TID 5) to be created. When it is created, it completes execution between FirstUserTask gets scheduled again.