Session 5 Operating Systems

5.1 Exercises

Exercise 1 Write a user program that uses system calls to read two numbers from the user's keyboard. Afterwards, print the sum of these two numbers.

Exercise 2 Write a program which reads the name of the user from the keyboard. Afterwards, display a greeting message dialog with content "Welcome [name]". Make sure your program does not crash when the user presses cancel or for long inputs. Instead, display an appropriate error message dialog.

Exercise 3 Write a custom user-mode exception handler. The exception handler should do nothing but jump over the faulting instuction. Make sure the handler does not modify any regular registers (Hint: use `uscratch`). Refer to table 1 and table 2 to correctly set-up the handler.

Note: In RARS you currently still need to write the value `1` to `ustatus` in order to enable custom user-mode exception handlers. Do this with `csrrsi zero, ustatus, 1` at the start of your program.

Exercise 4 Extend the handler from previous exercise so that it prints:

- The cause of the exception
- The address of the instruction that caused the exception

Possible output:

```
Exception with cause 4 occured at address 0x00400074
-- program is finished running (0) --
```

Make sure to restore all register values before returning from the trap handler. In theory, you could use the call stack for this purpose. However, the stack pointer itself might be misaligned. Using the stack pointer would then cause an additional exception within the handler. A better alternative is to reserve space in the data section to back-up registers in the trap handler and use the `uscratch` register to get the address of this space.

Exercise 5 Create a RARS program that plays music notes using the RARS system calls `MidiOut` (number 31) and `Sleep` (number 32). To get started, download the `music.asm` skeleton from Toledo, and proceed as follows:

1. Read through the provided skeleton code in `music.asm`. How is the music represented as a string?

2. Read through the code for the provided function `next_tone_from_string` which converts a given string element into a MIDI tone value. Which values are returned in the `a0` and `a1` registers? How can they be used?

3. Implement `play_song` by iterating over the null-terminated song string and making use of the `next_tone_from_string` and `play_tone` helper functions. Make sure to adhere to the calling conventions.

4. Now implement `play_tone` by making use of system calls 31 (MIDI out) and 32 (sleep). Make sure to first read the documentation for system call 31 (MIDI out) at `https://github.com/TheThirdOne/rars/wiki/Environment-Calls`. How many parameters are required? Which parameters are provided as global constants in the code skeleton and which parameters vary depending on the music string?

As a bonus, after you have finished the other exercises, implement the "`-`" sign in the music string of this exercise to make notes sound longer. This would work as follows: "`C- B-- A`" would play C for 2 times the normal duration, B for 3 times the normal duration and A would be played as normal.

Number	Name	Description
User trap setup		
`0x000`	`ustatus`	User status register.
`0x004`	`uie`	User interrupt-enable register
`0x005`	`utvec`	User trap handler base address
User trap handling		
`0x040`	`uscratch`	Scratch register for user trap handlers
`0x041`	`uepc`	User exception program counter
`0x042`	`ucause`	User trap cause
`0x043`	`utval`	User bad address or instruction
`0x044`	`uip`	User interrupt pending

Table 1: RISC-V User-mode CSR registers used for trap handling

Example usage	**Description**
`csrrc t0, fcsr, t1`	Atomic Read/Clear CSR: read from the CSR into t0 and clear bits of the CSR according to t1
`csrrci t0, fcsr, 10`	Atomic Read/Clear CSR Immediate: read from the CSR into t0 and clear bits of the CSR according to a constant
`csrrs t0, fcsr, t1`	Atomic Read/Set CSR: read from the CSR into t0 and logical or t1 into the CSR
`csrrsi t0, fcsr, 10`	Atomic Read/Set CSR Immediate: read from the CSR into t0 and logical or a constant into the CSR
`csrrw t0, fcsr, t1`	Atomic Read/Write CSR: read from the CSR into t0 and write t1 into the CSR
`csrrwi t0, fcsr, 10`	Atomic Read/Write CSR Immediate: read from the CSR into t0 and write a constant into the CSR

Table 2: RISC-V instructions to modify CSR registers