Report for PA3

by **Yuanfang Xiang** (221300012)

Note: Completed until `PAL` of `PA3.3`. Most game operations are tested without triggering error.

Part I: Exception Handling and Yield

Major Implementations

Added several `CSR` registers to the `cpu` structure, including `mepc`, `mstatus` and `mcause`.

Implemented interruption handler `isa_raise_intr()`, which copies the `snpc` of current executio stream (i.e. address that interruption returns to normal execution stream) and reads the event code from register specified by calling convention (i.e. `a7` or `a5` in rv32). In `nemu`, then implemented instruction `ecall` which jumps to exception entry of CTE.

After handling event, implemented context restoration that switches the execution stream back to customer program.

Exercise 1: Context Structure

The context structure pointed by `c` is stored in the memory, on top of the calling stack. In `trap.S`, the exception handler pushes all register to the stack. The `context`, including `gpr`s and `csr`s, are organized as a structure type in `c` code and are actually simply ranged linearly by the sequence specified in the structure. (Same in `trap.S`.)

The `gpr`s are pushed by some macros, and `csr`s are pushed with their offsets respectively.

Exercise 2: Detailed Process of Interruption

In AM(software)'s `cte_init`, the exception entrance address `_am_asm_` (i.e. `trap.S`) is loaded to system register `mtvec`. In `yield_test`, especially, a user handler function (`simple_trap`) is loaded for `__am_irq_handle` to do some user customized handling.

First when `yield` function is called, it uses an `asm` instruction to load the `CAUSE` code (`-1`) to a `NEMU`'s general register. `NEMU` then executes `ecall`, which calls `raise_intr` to set system registers `mepc`, `mstatus` and `mcause`, and set the `dnpc` to exception entrance.

After executing `ecall`, the execution stream is switched to handler `__am_asm_handle`, push the context (i.e. value of `gpr`s and `csr`s) to the stack, call `__am_irq_handle` to handle exception, and pop the context back to `gpr`s and `csr`s. After restoring the scene, `dnpc` is restored to the saved `mepc` and execution stream is switched back to the normal one. Therefore, the computer-as-statemachine `<gpr, csr, pc, M>` looks same and 'nothing happend'.

Part II: Loader and System Call

Major Implementations

Implemented `loader` that loads all `LOAD`-type segments, from `ramdisk`. Read segments according to `offset` values in `elf`'s `header`, and copy them to corresponding memory address.

Found argument passing registers and filled it in `do_syscall()` of `nanos-lite`. Completed system call parsing and Implemented specified system reactions of syscall `yield` and `exit`, in `nanos-lite` and `navy`.

Implemented system call `write` to write character to `AM`'s abstract serial port, and `sbrk` to maintain a program break address.

Exercise: how does `hello` run

After `make update`, `hello.c` is compiled as `ELF` binary file and linked into `ramdisk.img`. It's filename and offset in `ramdisk.img` is recorded in `ramdisk.h`, i.e. file table in `nanos-lite`.

When running `nemu`, files in `ramdisk` are loaded to memory in `resources.S` of `nanos-lite`. Using `ramdisk`'s APIs, data in `ramdisk` can be read in `nanos-lite`. Therefore in `loader` of `nanos-lite`, it copies binary instructions and datas of `hello`'s`ELF` according to offset of program segment and file offset of `/bin/hello` to corresponding position of memory, i.e. `0x83000000`.

After loading, the program entry is returned to `naive_uloader` by `loader`, and `naive_uloader` jumps to the`entry` by dereferencing `loader`'s return value. `hello.c` is then executed.

In `hello.c`, system call `write` is compiled to `ecall` and several GPR operations that pass arguments. In `nemu`, when `ecall` is executed, it switches the execution stream to corresponding `irq_handler` function in `AM`, which will call `do_syscall` in `nanos_lite`. In `do_syscall`, `nanos_lite` uses `AM`'s APIs to write characters to serial, and `AM` would call corresponding APIs in `nemu` to print character to the terminal.

After handling system call `write`, the CTE recovers registers and `pc` saved before system call, switching the execution stream back to `hello`.

Part III: Simple File System and Virtual File System

Major Implementations

Implemented fs_open, fs_read, fs_write and fs_close functions, which support basic file operations with ramdisk APIs and write/read data to/from ramdisk. Replaced ramdisk APIs in loader with newly completed fs functions.

Added I/O abstract registers of AM to vfs's file table, with filename specified by nanos-lite's API. Implemented read and write function of serial and vga, using I/O abstract registers defined by AM. Implemented NDL library functions in navy with system calls to read/write abstract files of vfs.

Part IV: Library Functions and Applications

Major Implementations

Completed the implementations of library functions that are required to run navy apps, including n-slider, n-menu, n-term, flappy-bird and PAL.

Library functions implemented includes all functions in fixedpt fixed point computation; Update screen, event and tick functions in NDL direct media layer; Updating and bliting surface, getting event and event status list, get ticks and wait, init and quit functions in mini SDL; Image loading in SLD_Image.

A simple batch processing system is implemented, with calling a loader to load nterm or menu again if exit system call is triggered and term or menu is loaded as first customer program. A simple command parse system is implemented, which supports sudo poweroff and executing ramdisk apps using absolute path or relative path of /bin.

As for PAL, most game operations are tested without triggering any error. PAL can be launched through n_term or menu. It can also be launched through modifying string IMAGE_FILE in nanos-lite/include/common.c to /bin/pal and run nanos-lite.

Error Handling and Debugging

Handled 2 highly complicated bugs that are worth noting in the report.

Bug 1: Loader Failed to Initialize

When trying to run n-slider, nemu crushed due to customer program's visiting invalid memory address. The invalid address is always a large random number.

After basic detecting, it is located that the bug happens when a global static variable is defined and initialized as 0 in NAVY's customer program, it is always not initialized and is a large random number. If the variable is initialized again when runtime, n-slider can run as normal. If the variable is initialized as a non-zero number, it can be initialized.

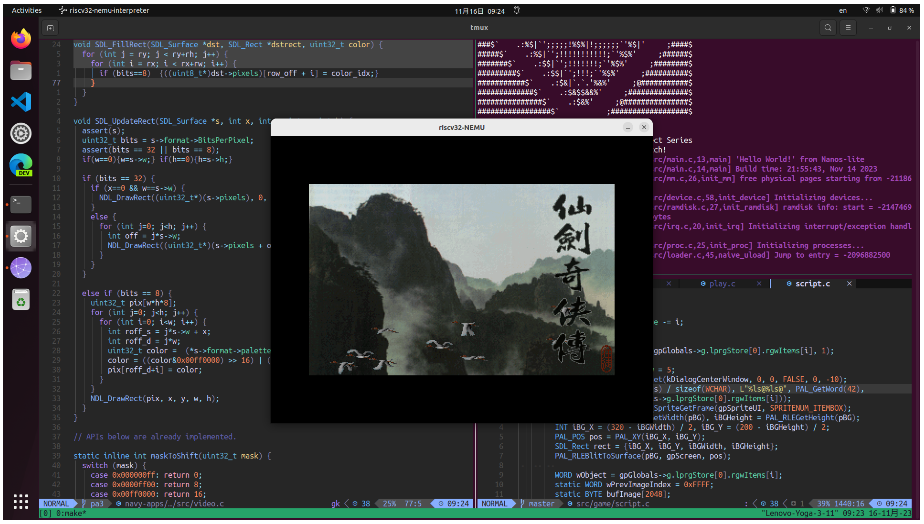

Figure 1: screenshot of running PAL (1)

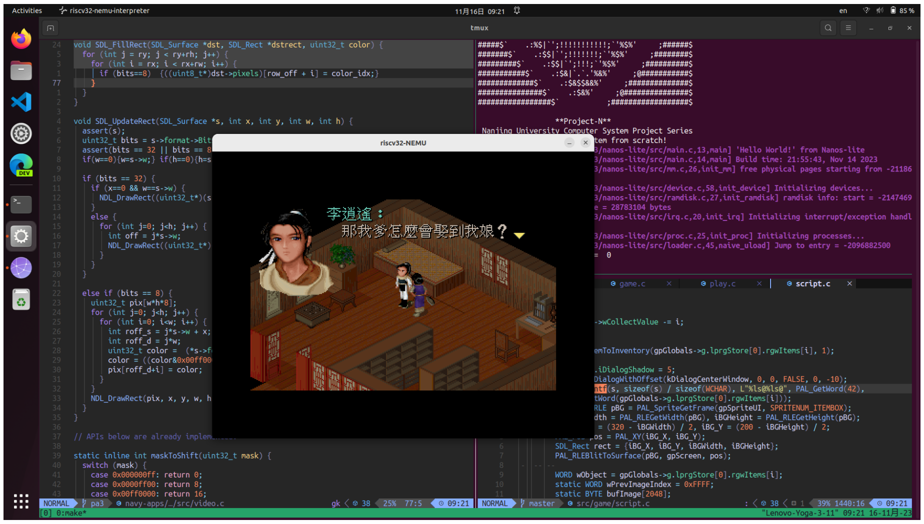

Figure 2: screenshot of running PAL (2)

Such problem is quite confusing, and seems like something went wrong in memory management. Finally it is located that in `loader` of `nanos-lite`, when loading data segment from `ELF` file to `nemu`'s memory, bits between `vaddr+filesz` and `vaddr+memsz` weren't zeroed out correctly. Specifically, I mistakenly called `memset` with `memset(*buf, num, val)`, as it should be `memset(*buf, val, num)`.

Locating this bug is meaningful, which significantly deepened my understanding to `ELF` and variable relocating. This can also be the answer to an elective exercise in `PA3.2`: the space between `filesz` and `memsz` stores the value of static global pointer and numeric variables. And should be zeroed out, because if a variable is initialized as 0 or not initialized, it will be exactly the number in corresponding memory address, whether it is zeroed out by the system or not.

Bug 2: Calling Stack Overflow

When running `PAL`, starting a new game will immediately trigger an address out of bound error, and starting with a presenting game record will trigger similar error after several operations.

After using different debugging methods, the buf is located at `SDL_UpdateRect()`. After a `write` system call, the `pc` value stored in `CTE`'s `contex` structure might be modified to a 24-bit number. Note that this situation not always happens, and in fact it's triggered very hardly.

Since the address causes error is always a 24-bit number, it seems like a color value. Therefore I printed the address of `context` structure and `fb` array, noticing that they are overlapped. `readelf` and `SDB` reveals that the stack pointer register `sp`'s value is inside `.bss` section of `nanos-lite.elf`. Thus it should be caused of stack overflow by using large local array.

Exercises: Cranes in PAL

In function `PAL_SplashScreen()`, the program first load the graph file of splash screen with a function `PAL_MKFReadChunk()` whose arguments specifies the file name and read buffer. In this function, `PAL` calls several `file` library APIs and library functions would trigger a corresponding system call. Once received system call, `fs` functions in `nanos-lite` would read file content from `ramdisk` accordingly.

After reading pictures, `PAL_SplashScreen()` enters an endless loop to show the splash screen. In the first `1500` ms, it scales the color palette of PAL surface to display the game homepage gradually. After that, `PAL` will update the position of cranes in each loop, record the current frame with a counter and update the crane pictures to the `SDL` surface with `PAL_PLEBlitToSurface()`. This function will process the graphs and call `SDL_BlitSurface()` in `SDL` library. At the end of each loop, `PAL` will update the screen with `SDL` library function `SDL_UpdateRect()`. This function uses `NDL` API to trigger a system call, which writes to `VGA` of the virtual file system. Once received system call, program jumps from PAL to system handler in nanos-lite and call AM APIs to write screen data to corresponding I/O abstract register and AM will call several SDL functions (imitating hardware of a real computer) in nemu to draw the cranes to a SDL window (imitating a screen).