Lab 4

Link for Demo:

5.1 : https://youtu.be/j1jeYG0ZbL0

5.2: https://youtu.be/9zXEpe-KnGY

Group 4
Andrew Branum, Cade Garrett, Ankit Kanotra, Shawn Witt, Nathan Zabloudil

CSE 525

Abstract

The purpose of this lab was to develop our knowledge and understanding of user processes and system calls within a raspberry Pi architecture. As per usual, we utilized the Raspberry Pi as well as our laptops to fulfill the obligations of the lab. Exercise 5.1 assigned us with the task of attempting to access system calls in user mode, and to return an exception that must be handled. Meanwhile, Exercise 5.2 tasked us with implementing a system call to create task priority, and then to demonstrate how the priority of the tasks was dynamically adjusted. Overall, the tasks of this lab were completed to their fullest.

Body

Team 4 got off to an easy start on Lab 4 with none of the hiccups of the previous 3 labs. Having a fully functional Raspberry Pi and serial cable, they were easily able to begin work on Section 5. They started the process by reading chapter 5 within the provided GitHub, and then beginning work on the 2 provided questions. The only notable issue was a misplaced SD card, which returned no trouble as a replacement was immediately found.

Question 5.1 asked the team to return exception calls using a register after attempting to use system calls in user mode. To do this, a new register was created to handle system instructions that, once triggered, will cause a trap state to be formed and will thus throw an exception, in response.

Question 5.2 tasked the team to implement a system call for task priority and to showcase it dynamically altering task priority. To do this, a 5th system call was created within sys.h that would create a priority number. This priority number will be established to each process and increment based on the previous process. Using supervisor mode, these processes will be executed and the priority will dynamically change.

Source Code (Software)

Exercise 5.1

*From the **sysregs.h** file:*

```
#ifndef _SYSREGS_H
#define _SYSREGS_H

// ***************************************
// SCTLR_EL1, System Control Register (EL1), Page 2654 of AArch64-Reference-Manual.
// ***************************************

#define SCTLR_RESERVED                  (3 << 28) | (3 << 22) | (1 << 20) | (1 << 11)
#define SCTLR_EE_LITTLE_ENDIAN          (0 << 25)
#define SCTLR_EOE_LITTLE_ENDIAN              (0 << 24)
#define SCTLR_I_CACHE_DISABLED               (0 << 12)
#define SCTLR_D_CACHE_DISABLED               (0 << 2)
#define SCTLR_MMU_DISABLED              (0 << 0)
#define SCTLR_MMU_ENABLED               (1 << 0)

#define SCTLR_VALUE_MMU_DISABLED    (SCTLR_RESERVED |
SCTLR_EE_LITTLE_ENDIAN | SCTLR_I_CACHE_DISABLED |
SCTLR_D_CACHE_DISABLED | SCTLR_MMU_DISABLED)

// ***************************************
// HCR_EL2, Hypervisor Configuration Register (EL2), Page 2487 of
AArch64-Reference-Manual.
// ***************************************

#define HCR_RW                          (1 << 31)
#define HCR_VALUE                   HCR_RW

// ***************************************
// SCR_EL3, Secure Configuration Register (EL3), Page 2648 of AArch64-Reference-Manual.
// ***************************************

#define SCR_RESERVED                (3 << 4)
#define SCR_RW                      (1 << 10)
#define SCR_NS                      (1 << 0)
#define SCR_VALUE                            (SCR_RESERVED | SCR_RW | SCR_NS)

// ***************************************
// SPSR_EL3, Saved Program Status Register (EL3) Page 389 of AArch64-Reference-Manual.
// ***************************************
```

```
#define SPSR_MASK_ALL            (7 << 6)
#define SPSR_EL1h                (5 << 0)
#define SPSR_VALUE               (SPSR_MASK_ALL | SPSR_EL1h)

// ***************************************
// ESR_EL1, Exception Syndrome Register (EL1). Page 2431 of AArch64-Reference-Manual.
// ***************************************

#define ESR_ELx_EC_SHIFT         26

#define ESR_ELx_EC_SVC64         0x15
#define ESR_ELx_EC_SYS_INSTR 0x18        //Establishing a register for instructions
#endif
```

*From the **entry.S** file:*

```
.globl vectors
vectors:
	ventry	sync_invalid_el1t			// Synchronous EL1t
	ventry	irq_invalid_el1t			// IRQ EL1t
	ventry	fiq_invalid_el1t			// FIQ EL1t
	ventry	error_invalid_el1t			// Error EL1t
	ventry	sync_invalid_el1h			// Synchronous EL1h
	ventry	el1_irq					// IRQ EL1h
	ventry	fiq_invalid_el1h			// FIQ EL1h
	ventry	error_invalid_el1h			// Error EL1h
	ventry	el0_sync				// Synchronous 64-bit EL0
	ventry	el0_irq					// IRQ 64-bit EL0
	ventry	fiq_invalid_el0_64			// FIQ 64-bit EL0
	ventry	error_invalid_el0_64			// Error 64-bit EL0
	ventry	sync_invalid_el0_32			// Synchronous 32-bit EL0
	ventry	irq_invalid_el0_32			// IRQ 32-bit EL0
	ventry	fiq_invalid_el0_32			// FIQ 32-bit EL0
	ventry	error_invalid_el0_32			// Error 32-bit EL0
sync_invalid_el1t:
	handle_invalid_entry 1, SYNC_INVALID_EL1t
irq_invalid_el1t:
	handle_invalid_entry 1, IRQ_INVALID_EL1t
fiq_invalid_el1t:
	handle_invalid_entry 1, FIQ_INVALID_EL1t
error_invalid_el1t:
	handle_invalid_entry 1, ERROR_INVALID_EL1t
sync_invalid_el1h:
	handle_invalid_entry 1, SYNC_INVALID_EL1h
fiq_invalid_el1h:
	handle_invalid_entry 1, FIQ_INVALID_EL1h
```

```
error_invalid_el1h:
        handle_invalid_entry 1, ERROR_INVALID_EL1h
fiq_invalid_el0_64:
        handle_invalid_entry 0, FIQ_INVALID_EL0_64
error_invalid_el0_64:
        handle_invalid_entry 0, ERROR_INVALID_EL0_64
sync_invalid_el0_32:
        handle_invalid_entry 0, SYNC_INVALID_EL0_32
irq_invalid_el0_32:
        handle_invalid_entry 0, IRQ_INVALID_EL0_32
fiq_invalid_el0_32:
        handle_invalid_entry 0, FIQ_INVALID_EL0_32
error_invalid_el0_32:
        handle_invalid_entry 0, ERROR_INVALID_EL0_32
el1_irq:
        kernel_entry 1
        bl      handle_irq
        kernel_exit 1
el0_irq:
        kernel_entry 0
        bl      handle_irq
        kernel_exit 0
el0_sync:
        kernel_entry 0
        mrs     x25, esr_el1                    // reads syndrome register
        lsr     x24, x25, #ESR_ELx_EC_SHIFT     // exception class
        cmp     x24, #ESR_ELx_EC_SVC64          // puts SVC in 64-bit state
        b.eq    el0_svc
        cmp x24, #ESR_ELx_EC_SYS_INSTR
        b.eq el0_sys_instr
        handle_invalid_entry 0, SYNC_ERROR
sc_nr   .req    x25                             // number of system calls
scno    .req    x26                             // syscall number
stbl    .req    x27                             // syscall table pointer
el0_svc:
        adr     stbl, sys_call_table            // load syscall table
        uxtw    scno, w8                        // syscall number in w8
        mov     sc_nr, #Num_syscalls
        bl      enable_irq
        cmp scno, sc_nr                         // check upper syscall limit
        b.hs    ni_sys
        ldr     x16, [stbl, scno, lsl #3]       // address in syscall table
        blr     x16                             // call sys_* routine
        b       ret_from_syscall
ni_sys:
        handle_invalid_entry 0, SYSCALL_ERROR
```

```
ret_from_syscall:
        bl      disable_irq
        str     x0, [sp, #S_X0]                        // returned x0
        kernel_exit 0
el0_sys_instr:
        bl show_trapped_sys_instr                      //branch and link to show trapped
        ldr     x22, [sp, #16 * 16]                    // skip the trapped mrs instruction
        add x22, x22, #4                               // add the register value
        str x22, [sp, #16 * 16]
        b ret_to_user                                  //return to user
.globl ret_from_fork
ret_from_fork:
        bl      schedule_tail
        cbz     x19, ret_to_user                  // not a kernel thread
        mov     x0, x20
        blr     x19
ret_to_user:
        bl disable_irq
        kernel_exit 0
.globl err_hang
err_hang: b err_hang
```

*From the **irq.c** file:*

```
#include "utils.h"
#include "printf.h"
#include "timer.h"
#include "entry.h"
#include "peripherals/irq.h"

const char *entry_error_messages[] = {
   "SYNC_INVALID_EL1t",
   "IRQ_INVALID_EL1t",
   "FIQ_INVALID_EL1t",
   "ERROR_INVALID_EL1T",

   "SYNC_INVALID_EL1h",
   "IRQ_INVALID_EL1h",
   "FIQ_INVALID_EL1h",
   "ERROR_INVALID_EL1h",

   "SYNC_INVALID_EL0_64",
   "IRQ_INVALID_EL0_64",
   "FIQ_INVALID_EL0_64",
   "ERROR_INVALID_EL0_64",

   "SYNC_INVALID_EL0_32",
```

```
    "IRQ_INVALID_EL0_32",
    "FIQ_INVALID_EL0_32",
    "ERROR_INVALID_EL0_32",

    "SYNC_ERROR",
    "SYSCALL_ERROR"
};

void enable_interrupt_controller()
{
    put32(ENABLE_IRQS_1, SYSTEM_TIMER_IRQ_1);
}

void show_invalid_entry_message(int type, unsigned long esr, unsigned long address)
{
    printf("%s, ESR: %x, address: %x\r\n", entry_error_messages[type], esr, address);
}

void show_trapped_sys_instr() {
    printf("trapped\r\n");                        //print if trapped
}

void handle_irq(void)
{
    unsigned int irq = get32(IRQ_PENDING_1);
    switch (irq) {
            case (SYSTEM_TIMER_IRQ_1):
                    handle_timer_irq();
                    break;
            default:
                    printf("Unknown pending irq: %x\r\n", irq);
    }
}
```

*From the **utils.h** file:*

```
#ifndef   _UTILS_H
#define   _UTILS_H

extern void delay ( unsigned long);
extern void put32 ( unsigned long, unsigned int );
extern unsigned int get32 ( unsigned long );
extern int get_el ( void );
extern unsigned int get_ex();                     //Function to get an exception

#endif  /*_UTILS_H */
```

Exercise 5.2

*From the **irq.c** file:*

```
#include "utils.h"
#include "printf.h"
#include "timer.h"
#include "entry.h"
#include "peripherals/irq.h"

const char *entry_error_messages[] = {
    "SYNC_INVALID_EL1t",
    "IRQ_INVALID_EL1t",
    "FIQ_INVALID_EL1t",
    "ERROR_INVALID_EL1T",

    "SYNC_INVALID_EL1h",
    "IRQ_INVALID_EL1h",
    "FIQ_INVALID_EL1h",
    "ERROR_INVALID_EL1h",

    "SYNC_INVALID_EL0_64",
    "IRQ_INVALID_EL0_64",
    "FIQ_INVALID_EL0_64",
    "ERROR_INVALID_EL0_64",

    "SYNC_INVALID_EL0_32",
    "IRQ_INVALID_EL0_32",
    "FIQ_INVALID_EL0_32",
    "ERROR_INVALID_EL0_32",
};

void enable_interrupt_controller()
{
    put32(ENABLE_IRQS_1, SYSTEM_TIMER_IRQ_1);
}

void show_invalid_entry_message(int type, unsigned long esr, unsigned long address)
{
    printf("%s, ESR: %x, address: %x\r\n", entry_error_messages[type], esr, address);
}

void handle_irq(void)
{
    unsigned int irq = get32(IRQ_PENDING_1);
    switch (irq) {
```

```
            case (SYSTEM_TIMER_IRQ_1):
                handle_timer_irq();
                break;
            default:
                printf("Unknown pending irq: %x\r\n", irq);
    }
}
```

From the kernel.c file:

```
#include "printf.h"
#include "utils.h"
#include "timer.h"
#include "irq.h"
#include "sched.h"
#include "fork.h"
#include "mini_uart.h"
#include "sys.h"

void user_process1(char *array)
{
    char buf[2] = {0};
    long priority = 1;                          //intialize priority to 1
    while (1){
        if(array[0] == '1'){                    //if initial priority == 1
            call_sys_priority(++priority);          //set next priority
        }
        for(int j = 0; j < 4; j++){             //write priority for each
            for (int i = 0; i < 5; i++){
                buf[0] = array[i];
                call_sys_write(buf);
                delay(100000);
            }
        }
    }
}

void user_process(){
    char buf[30] = {0};
    tfp_sprintf(buf, "User process start\n\r");
    call_sys_write(buf);
    unsigned long stack = call_sys_malloc();
    if (stack < 0) {
        printf("Error while allocating stack for process 1\n\r");
        return;
    }
    int err = call_sys_clone((unsigned long)&user_process1, (unsigned long)"12345", stack);
```

```
    if (err < 0){
            printf("Error while clonning process 1\n\r");
            return;
    }
    stack = call_sys_malloc();
    if (stack < 0) {
            printf("Error while allocating stack for process 1\n\r");
            return;
    }
    err = call_sys_clone((unsigned long)&user_process1, (unsigned long)"abcd", stack);
                                                                // err is pid of process2
    if (err < 0){
            printf("Error while clonning process 2\n\r");
            return;
    }
    //call_sys_priority(err, 0xa);
    call_sys_exit();
}

void kernel_process(){
    printf("Kernel process started. EL %d\r\n", get_el());
    int err = move_to_user_mode((unsigned long)&user_process);
    if (err < 0){
            printf("Error while moving process to user mode\n\r");
    }
}


void kernel_main(void)
{
    uart_init();
    init_printf(0, putc);
    irq_vector_init();
    timer_init();
    enable_interrupt_controller();
    enable_irq();

    int res = copy_process(PF_KTHREAD, (unsigned long)&kernel_process, 0, 0);
    if (res < 0) {
            printf("error while starting kernel process");
            return;
    }

    while (1){
            schedule();
    }
```

}

*From the **sys.S** file:*

```
#include "sys.h"

.globl call_sys_write
call_sys_write:
   mov w8, #SYS_WRITE_NUMBER
   svc #0
   ret

.globl call_sys_malloc
call_sys_malloc:
   mov w8, #SYS_MALLOC_NUMBER
   svc #0
   ret

.globl call_sys_exit
call_sys_exit:
   mov w8, #SYS_EXIT_NUMBER
   svc #0
   ret

.globl call_sys_clone
call_sys_clone:
   /* Save args for the child.  */
   mov    x10, x0                          /*fn*/
   mov    x11, x1                          /*arg*/
   mov    x12, x2                          /*stack*/

   /* Do the system call.  */
   mov     x0, x2                          /* stack  */
   mov    x8, #SYS_CLONE_NUMBER
   svc    0x0

   cmp    x0, #0
   beq    thread_start
   ret

.globl call_sys_priority                         //makes global
call_sys_priority:                               //calls sys_priority
   mov w8, #SYS_PRIORITY_NUMBER                  //move priority # to w8
   svc #0                                  //trigger supervisor mode
   ret                                     //return
```

```
thread_start:
    mov    x29, 0

    /* Pick the function arg and execute.  */
    mov    x0, x11
    blr    x10

    /* We are done, pass the return value through x0.  */
    mov    x8, #SYS_EXIT_NUMBER
    svc    0x0
```

*From the **sys.c** file:*

```
#include "fork.h"
#include "printf.h"
#include "utils.h"
#include "sched.h"
#include "mm.h"

void sys_write(char * buf){
    printf(buf);
}

int sys_clone(unsigned long stack){
    return copy_process(0, 0, 0, stack);
}

unsigned long sys_malloc(){
    unsigned long addr = get_free_page();
    if (!addr) {
            return -1;
    }
    return addr;
}

void sys_exit(){
    exit_process();
}

void sys_priority(long priority)                        //schedules tasks based on priority
{
        current->priority = priority;
}

void * const sys_call_table[] = {sys_write, sys_malloc, sys_clone, sys_exit, sys_priority};
```

*From the **sys.h** file:*

```
#ifndef    _SYS_H
#define    _SYS_H

#define Num_syscalls          5                    //Number of system calls is 5

#define SYS_WRITE_NUMBER      0          // syscal numbers
#define SYS_MALLOC_NUMBER   1
#define SYS_CLONE_NUMBER     2
#define SYS_EXIT_NUMBER        3
#define SYS_PRIORITY_NUMBER 4          //priority number is 4

#ifndef __ASSEMBLER__

void sys_write(char * buf);
int sys_fork();

void call_sys_write(char * buf);
int call_sys_clone(unsigned long fn, unsigned long arg, unsigned long stack);
unsigned long call_sys_malloc();
void call_sys_exit();
void call_sys_priority(long priority);            //Calls System Priority

#endif
#endif  /*_SYS_H */
```

Schematics (Hardware) - None

Analysis

All of the knowledge acquired in this lab was through tutorials and exercises created and hosted on Github by Sergei Matyukevich.The lessons in this lab pertain to user processes and system calls. Along with process management, process isolation should also be a present functionality in an operating system. One important technique that can be done is moving all user processes to exception level 0. Doing so ensures that a user cannot perform privileged processor operations. This is actually a necessary step to ensure proper process isolation. System calls are methods in an API provided by the OS that allow the OS to remain in exception level 0 while allowing interaction between a device and its users, thus maintaining process isolation. System calls are basically synchronous exceptions, and are generated by the svc instruction. Since system calls are synchronous exceptions, they are dealt with at exception level 1. The operating system then performs a series of of tasks, after which execution resumes at exception level 0. These tasks include verifying each argument, executes the required task, and initiates normal exception return. RPi OS contains four syscalls:

1. **Write** - Uses the UART device to output something on a screen. The string to be printed is one its arguments.
2. **Clone** - Causes a new user thread to be created. The first argument for this system call is the location in the stack allocated for the new user thread.
3. **Malloc** - Allocates a memory page for a user process.
4. **Exit -** Is called by all processes after they are done executing.

These system cals are defined in the sys.c file. An array called “sys_call_table” contains pointers to each syscall handler. Each system call has a corresponding number, which acts as an index for each system call in sys_call_table. Before a system call can actually occur, a task must be running in user mode. There are two ways in which new user tasks can be created. Either a kernel thread can be moved to user mode, or a user task can fork itself to create a new user task.

Conclusion

The purpose of this lab was to develop students' understanding of system calls and user processes within the Raspberry Pi architecture. Group 4 showcased this knowledge by returning exceptions to attempted system calls and creating task priority to deal with tasks as they dynamically come up. As computer scientists and engineers, it’s important students understand low level processes such as system calls. Not only are they good practical knowledge, especially for anyone designing a system or kernel, but knowledge of such processes are often the difference between a computer science engineer, and someone who only took a programming bootcamp.

References

S-Matyukevich. “S-Matyukevich/Raspberry-Pi-OS: Learning Operating System Development Using Linux Kernel and Raspberry Pi.” *GitHub*, https://github.com/s-matyukevich/raspberry-pi-os.