Lab 3

Link for Demo: https://youtu.be/FZ4CXpdbF-o

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

CSE 525

Abstract

The purpose of this Lab was learning how to handle interrupts and processor scheduling, especially via priority, using both assembly and C. Using our laptops amd Raspberry Pi, we were able to successfully able to generate interrupts with a local timer, handle MiniUART interrupts, return the current memory and processor state for all main Kernel functions, and assign task priority so hat longer tasks have a higher priority than shorter tasks. Overall, we completed the tasks asked of us and our goals were accomplished.

Body

Team 4 got off to an easy start on Lab 3 as opposed to Labs 1 and 2. Having a fully functional Raspberry Pi, they were easily able to begin work on Sections 3 and 4. They started the process by reading both chapters 3 and 4 within the provided GitHub, and then beginning work on the 4 questions. There was an issue with the USB to TTL serial cable, which failed to output correctly to the console but this issue was resolved by using a different cable.

Question 3.1 asked the team to use a local timer in place of the system timer to generate interrupts. To do this, they defined a local timer control and flag instead of a global one, as well as an interrupt source and a local timer. They also adjusted the control for a fast interrupt request. From there, they altered every timer and interrupt function to use the established local timer instead of the global one.

Question 3.2 tasked the team to handle MiniUART interrupts. The provided instructions asked the team to replace the final loop in kernel_main with one that does nothing, and then generate an interrupt when the user inputs a new character. Finally, they should implement an interrupt handler responsible for printing each inputted character. The first step involved enabling the UART to receive interrupt requests via the miniUART_IRQ. The AUX_IRQ register, which is used to identify pending interrupts, is then declared at the proper address. A new function called handle_irq_uart was then created to accept and output user input. The handle_irq function in irq.c was then modified to account for the irq timer and the IRQ of the UART.

Question 4.1 required the team to print out the current memory and processor state of all main kernel functions. To do so, Team 4 added printf to sched.c and fork.c so that it will regularly print out both the Processor State (whenever it changes), and the current memory address. This causes these both to be printed whenever any of the main kernel functions are called.

Question 4.2 asked the team to introduce a way to assign task priority, so that higher runtime tasks have higher priority than lower runtime tasks. This was done by adding another field to the copy_process function to take a priority and updated to kernel main to grant the priority a value based on length of the runtime.

Source Code (Software)

Exercise 3.1

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

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

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()
{
    //Chnage for local timer
    unsigned int local_timer_ctrl = get32(TIMER_CTRL);          //Sets local timer controller
    put32(TIMER_CTRL, (local_timer_ctrl | (1 << 29)));          //Enables interrupt controller
```

```
}

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)
{
   //Change for local timer
   unsigned int irq = get32(CORE0_INT_SOURCE);          //Sets Interrupt Request
   switch (irq) {
      case (LOCAL_TIMER_INT):                           //When local timer is interrupted
         handle_timer_irq();
         break;
      default:
         printf("Unknown pending irq: %x\r\n", irq);
   }
}
```

*From the **timer.c** file*

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

const unsigned int interval = 200000;
unsigned int curVal = 0;

void timer_init ( void )
{
        put32(TIMER_CTRL, ((1<<28) | interval));        //Initializes Timer
}

void handle_timer_irq( void )
{
        //Changed to local timer
        printf("Timer interrupt recived: Local Timer\n");   //Prints when interrupt is called
        put32(TIMER_FLAG, (3<<30));                         //Triggers Timer Flag
}
```

*From the **base.h** file*

```
#ifndef_P_BASE_H
#define_P_BASE_H
//Change for local timer
#define PBASE 0x3F000000
#define PERIPHERAL_BASE 0x40000000                    //Defines Peripheral Base

#endif  /*_P_BASE_H */
```

*From the **irq.h** file*

```
#ifndef_P_IRQ_H
#define_P_IRQ_H
#include "peripherals/base.h"
//Change for local timer
#define IRQ_BASIC_PENDING      (PBASE+0x0000B200)
#define IRQ_PENDING_1          (PBASE+0x0000B204)
#define IRQ_PENDING_2          (PBASE+0x0000B208)
#define FIQ_CONTROL            (PBASE+0x0000B20C)        //defines control for Fast
                                                           Interrupt Request
#define ENABLE_IRQS_1          (PBASE+0x0000B210)
#define ENABLE_IRQS_2          (PBASE+0x0000B214)
#define ENABLE_BASIC_IRQS      (PBASE+0x0000B218)
#define DISABLE_IRQS_1         (PBASE+0x0000B21C)
#define DISABLE_IRQS_2         (PBASE+0x0000B220)
#define DISABLE_BASIC_IRQS     (PBASE+0x0000B224)
#define SYSTEM_TIMER_IRQ_0     (1 << 0)
#define SYSTEM_TIMER_IRQ_1     (1 << 1)
#define SYSTEM_TIMER_IRQ_2     (1 << 2)
#define SYSTEM_TIMER_IRQ_3     (1 << 3)
#define CORE0_INT_SOURCE    (PERIPHERAL_BASE+0x60)   //defines interrupt source
#define LOCAL_TIMER_INT     (1<<11)                  //defines interrupt of local timer
#endif  /*_P_IRQ_H */
```

*From the **timer.h** file*

```
#ifndef_P_TIMER_H
#define_P_TIMER_H
#include "peripherals/base.h"
//Change for local timer
#define TIMER_CS       (PBASE+0x00003000)
#define TIMER_CLO      (PBASE+0x00003004)
#define TIMER_CHI      (PBASE+0x00003008)
#define TIMER_C0       (PBASE+0x0000300C)
```

```
#define TIMER_C1        (PBASE+0x00003010)
#define TIMER_C2        (PBASE+0x00003014)
#define TIMER_C3        (PBASE+0x00003018)
#define TIMER_CTRL  (PERIPHERAL_BASE+0x34)      //defines Local Timer Control
#define TIMER_FLAG  (PERIPHERAL_BASE+0x38)      //defines Local Timer Flag
#define TIMER_CS_M0     (1 << 0)
#define TIMER_CS_M1     (1 << 1)
#define TIMER_CS_M2     (1 << 2)
#define TIMER_CS_M3     (1 << 3)

#endif  /*_P_TIMER_H */
```

Exercise 3.2

*From the **mini_uart.c** file*

```
#include "utils.h"
#include "peripherals/mini_uart.h"
#include "peripherals/gpio.h"

void uart_send ( char c )
{
        while(1) {
                if(get32(AUX_MU_LSR_REG)&0x20)
                        break;
        }
        put32(AUX_MU_IO_REG,c);
}

char uart_recv ( void )
{
        while(1) {
                if(get32(AUX_MU_LSR_REG)&0x01)
                        break;
        }
        return(get32(AUX_MU_IO_REG)&0xFF);
}

void uart_send_string(char* str)
{
        for (int i = 0; str[i] != '\0'; i ++) {
                uart_send((char)str[i]);
        }
}

void uart_init ( void )
```

```
{
	unsigned int selector;

	selector = get32(GPFSEL1);
	selector &= ~(7<<12);                   // clean gpio14
	selector |= 2<<12;                      // set alt5 for gpio14
	selector &= ~(7<<15);                   // clean gpio15
	selector |= 2<<15;                      // set alt5 for gpio15
	put32(GPFSEL1,selector);

	put32(GPPUD,0);
	delay(150);
	put32(GPPUDCLK0,(1<<14)|(1<<15));
	delay(150);
	put32(GPPUDCLK0,0);

	put32(AUX_ENABLES,1);                   //Enable mini uart (this also enables access to its
registers)
	put32(AUX_MU_CNTL_REG,0);               //Disable auto flow control and disable
receiver and transmitter (for now)
	put32(AUX_MU_IER_REG, ENABLE_MU_REC_INT);    //Disable receive and
                                                        transmit interrupts
	put32(AUX_MU_LCR_REG,3);                //Enable 8 bit mode
	put32(AUX_MU_MCR_REG,0);                //Set RTS line to be always high
	put32(AUX_MU_BAUD_REG,270);             //Set baud rate to 115200

	put32(AUX_MU_CNTL_REG,3);               //Finally, enable transmitter and receiver
}


// This function is required by printf function
void putc ( void* p, char c)
{
	uart_send(c);
}
void handle_uart_irq( void )                    //Handles UART Interrupt Request
{
	char i = uart_recv();                   //Receive Interrupt from UART
	uart_send_string("UART interrupt received: "); //Sends String to UART
	uart_send(i);                           //Sends Interrupt to UART
	uart_send_string("\n\r");               //Sends New Line
}
```

*From the **kernel.c** file*

```
#include "printf.h"
#include "timer.h"
#include "irq.h"
#include "mini_uart.h"


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

	while (1){                                  //Removes UART Send
	}
}
```

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

```
#include "utils.h"
#include "printf.h"
#include "timer.h"
#include "entry.h"
#include "mini_uart.h"                          //Includes Mini_UART.h
#include "peripherals/mini_uart.h"              //Includes Mini_UART.h from
                                                Peripherals Folder
#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 | en_AUX_INT);     //Enables
                                                                   Interrupt Request, System Timer
                                                                   Interrupt Request, and Auxiliary
                                                                   Flag Interrupt
}

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)                                      //Handles Interrupt Request
{
        unsigned int irq0 = get32(IRQ_PENDING_1);          //Sets irq0 to Interrupt Request
                                                           Pending
        unsigned int irq1 = get32(AUX_IRQ);                //Sets irq1 to  Auxiliary Interrupt
                                                           Request

        if (irq0 & SYSTEM_TIMER_IRQ_1)                    //if interrupt request is pending and
        {                                                    system timer has an interrupt
        handle_timer_irq();                                  request, then handle timer
        }                                                    interrupt request
        if (irq1 & miniUART_IRQ)                           //if auxiliary interrupt request and
        {                                                    miniUART has an interrupt request
                                                             then handle UART interrupt request
        handle_uart_irq()                                  //if Auxiliary Interrupt Request and
                                                             MiniUart has an Interrupt Request,
                                                             then handle Uart Interrupt Request

}
```

*From the **irq.h** file*

```
#ifndef_P_IRQ_H
#define_P_IRQ_H
```

```
#include "peripherals/base.h"

#define IRQ_BASIC_PENDING      (PBASE+0x0000B200)
#define IRQ_PENDING_1          (PBASE+0x0000B204)
#define IRQ_PENDING_2          (PBASE+0x0000B208)
#define FIQ_CONTROL            (PBASE+0x0000B20C)
#define ENABLE_IRQS_1          (PBASE+0x0000B210)
#define ENABLE_IRQS_2          (PBASE+0x0000B214)
#define ENABLE_BASIC_IRQS      (PBASE+0x0000B218)
#define DISABLE_IRQS_1         (PBASE+0x0000B21C)
#define DISABLE_IRQS_2         (PBASE+0x0000B220)
#define DISABLE_BASIC_IRQS     (PBASE+0x0000B224)


#define SYSTEM_TIMER_IRQ_0   (1 << 0)
#define SYSTEM_TIMER_IRQ_1   (1 << 1)
#define SYSTEM_TIMER_IRQ_2   (1 << 2)
#define SYSTEM_TIMER_IRQ_3   (1 << 3)

#define miniUART_IRQ           (1 << 0)        //Defines MiniUart Interrupt Request
#define en_AUX_INT             (1 << 29)       //Defines Auxiliary Interrupt Flag

#endif  /*_P_IRQ_H */
```

*From the **mini_uart.h** file*

```
#ifndef _P_MINI_UART_H
#define_P_MINI_UART_H

#include "peripherals/base.h"

#define AUX_ENABLES     (PBASE+0x00215004)
#define AUX_MU_IO_REG   (PBASE+0x00215040)
#define AUX_MU_IER_REG  (PBASE+0x00215044)
#define AUX_MU_IIR_REG  (PBASE+0x00215048)
#define AUX_MU_LCR_REG  (PBASE+0x0021504C)
#define AUX_MU_MCR_REG  (PBASE+0x00215050)
#define AUX_MU_LSR_REG  (PBASE+0x00215054)
#define AUX_MU_MSR_REG  (PBASE+0x00215058)
#define AUX_MU_SCRATCH  (PBASE+0x0021505C)
#define AUX_MU_CNTL_REG (PBASE+0x00215060)
#define AUX_MU_STAT_REG (PBASE+0x00215064)
```

```
#define AUX_MU_BAUD_REG (PBASE+0x00215068)

#define AUX_IRQ                 (PBASE+0x0021506C)      //Defines Auxiliary Interrupt Request
#define ENABLE_MU_REC_INT (0xFD)                        //Defines Enable Mini_Uart
                                                        Receiving Interrupt

#endif  /*_P_MINI_UART_H */
```

Exercise 4.1

*From the **sched.c** file*

```
#include "sched.h"
#include "irq.h"
#include "printf.h"
#include "utils.h"                                      //include utils.h

static struct task_struct init_task = INIT_TASK;
struct task_struct *current = &(init_task);
struct task_struct * task[NR_TASKS] = {&(init_task), };
int nr_tasks = 1;

void preempt_disable(void)
{
        current->preempt_count++;
}

void preempt_enable(void)
{
        current->preempt_count--;
}


void _schedule(void)
{
        preempt_disable();
        int next,c;
        struct task_struct * p;
        while (1) {
                c = -1;
                next = 0;
                for (int i = 0; i < NR_TASKS; i++){
                        p = task[i];
                        if (p && p->state == TASK_RUNNING && p->counter > c) {
```

```
                        c = p->counter;
                        next = i;
                }
        }
        if (c) {
                break;
        }
        for (int i = 0; i < NR_TASKS; i++) {
                p = task[i];
                if (p) {
                        p->counter = (p->counter >> 1) + p->priority;
                }
        }
    }
    printf("\r\nswitch to task %d\r\n", next);        //Prints the switched to task
    printf("Tasks state:\r\n");                       //Prints Task's Processor State

    struct task_struct *tasks = task[0];               //Pointer/array of tasks
    for (int i = 0; (i < NR_TASKS) && tasks; i++){    //While there are still tasks to print out
        tasks = task[i];                              //Sets tasks to current task
        if (tasks) {                                          //if tasks
    printf("    %d: sp: 0x%x\r\n", i, tasks->cpu_context.sp); //Prints Task Number
                                                              and the current Processor State
        }
    }

    switch_to(task[next]);
    preempt_enable();
}

void schedule(void)
{
    current->counter = 0;
    _schedule();
}

void switch_to(struct task_struct * next)
{
    if (current == next)
        return;
    struct task_struct * prev = current;
    current = next;
    cpu_switch_to(prev, next);
}

void schedule_tail(void) {
```

```
	preempt_enable();
}


void timer_tick()
{
	--current->counter;
	if (current->counter>0 || current->preempt_count >0) {
		return;
	}
	current->counter=0;
	enable_irq();
	_schedule();
	disable_irq();
}
```

*From the **fork.c** file*

```
#include "mm.h"
#include "sched.h"
#include "entry.h"
#include "printf.h"                                          //Includes printf.h

int copy_process(unsigned long fn, unsigned long arg)
{
	preempt_disable();
	struct task_struct *p;

	p = (struct task_struct *) get_free_page();
	if (!p)
		return 1;
	p->priority = current->priority;
	p->state = TASK_RUNNING;
	p->counter = p->priority;
	p->preempt_count = 1; //disable preemption until schedule_tail

	p->cpu_context.x19 = fn;
	p->cpu_context.x20 = arg;
	p->cpu_context.pc = (unsigned long)ret_from_fork;
	p->cpu_context.sp = (unsigned long)p + THREAD_SIZE;
	int pid = nr_tasks++;
	task[pid] = p;
	printf("New pid: %d, sp: 0x%x\r\n", pid, p->cpu_context.sp);        //Prints New memory
                                                                        id and processor state
	preempt_enable();
```

```
	return 0;
}
```

Exercise 4.2

*From the **fork.c** file*

```
#include "mm.h"
#include "sched.h"
#include "entry.h"
#include "printf.h"                                          //Include printf.h

int copy_process(unsigned long fn, unsigned long arg, unsigned int pr) //Sets priority as an input
{
	preempt_disable();
	struct task_struct *p;

	p = (struct task_struct *) get_free_page();
	if (!p)
		return 1;
	p->priority = pr;                                    //Sets Process Priority to input pr
	p->state = TASK_RUNNING;
	p->counter = p->priority;
	p->preempt_count = 1; //disable preemption until schedule_tail

	p->cpu_context.x19 = fn;
	p->cpu_context.x20 = arg;
	p->cpu_context.pc = (unsigned long)ret_from_fork;
	p->cpu_context.sp = (unsigned long)p + THREAD_SIZE;
	int pid = nr_tasks++;
	task[pid] = p;

	printf("\n\r|||||||||||||||| Task[%d] created ||||||||||||||||||\r\n", pid); //Prints current task
	printf("\n\rTasks allocated at 0x%08x.\r\n", p);                  //Prints where task allocated
	printf("x19 = 0x%08x.\r\n", p->cpu_context.x19);               //Prints Process fn
	printf("x20 = 0x%08x.\r\n", p->cpu_context.x20);               //Prints Process arg
	printf("program counter  = 0x%08x.\r\n", p->cpu_context.pc);  //Prints Program Counter
	printf("stack pointer  = 0x%08x.\r\n", p->cpu_context.sp);   //Prints value of stack ptr

	preempt_enable();
	return 0;
}
```

*From the **kernel.c** file*

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

void process(char *array)
{
	while (1){
		for (int i = 0; i < 5; i++){
			uart_send(array[i]);
			delay(100000);
		}
	}
}

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

	int res = copy_process((unsigned long)&process, (unsigned long)"12345", (unsigned int) 1);
                                                                //Sets pr to 1
	if (res != 0) {
		printf("error while starting process 1");
		return;
	}
	res = copy_process((unsigned long)&process, (unsigned long)"abcde", (unsigned int) 2);
                                                                //Sets pr to 2
	if (res != 0) {
		printf("error while starting process 2");
		return;
	}

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

Schematics (Hardware) - None

Analysis

The purpose and lessons from this lab are not dissimilar to the previous lab. In addition to working with the kernel and learning how to utilize it further like last lab, this lab focuses on interrupts and how the kernel utilizes them as well as new processes that the kernel generates and how it keeps track of them. Kernels are at the base of modern operating systems such as Windows and Linux, and interrupts are used even in single function microcontrollers. As such, it makes sense that as CSE and ECE majors it is important to get a general understanding of these labs, especially in regards to aspects as important as interrupts and processes.

Much of the knowledge acquired in this lab was through tutorials and exercises created and hosted on Github by Sergei Matyukevich. Lesson 3 largely pertained to timers and interrupts. Timers are an integral component of operating systems, as they are used extensively for process scheduling, since they possess the capability to interrupt the processor at variable frequencies. A process scheduler utilizes timer interrupts to measure the duration of executed processes. These durations are used to determine the subsequent process to be executed. In the ARM.v8 architecture, there are 4 types of exceptions: Synchronous exceptions, Interrupt Requests, Fast Interrupt Requests, and System Errors. Synchronous exceptions arise due to instructions in the software, such as the svc instruction, whereas asynchronous exceptions are never caused by software, and are instead generated due to hardware. Synchronous exceptions can also be used to induce software interrupts. Unlike synchronous exceptions, interrupt requests are asynchronous and can only be caused by hardware. Fast interrupt requests are also asynchronous and are sometimes referred to as fast requests. Fast interrupt requests allow exceptions to be prioritized. Fast interrupts have higher priority than normal interrupts, and are also handled by a different handler than normal interrupts. System Errors are also asynchronous, and only occur due to errors within the software. A unique handler exists for each exception type. Additionally, there should be a different handler for each unique execution state. There are 4 unique execution states when dealing with Exception Level 1: EL1t, EL1h, EL0_64, and EL0_32. Since there are 4 different exception types and 4 different execution states, there will be a total of 16 exception handlers. An exception vector table is a structure that contains the addresses of each exception handler. Interrupts can be masked or unmasked. An interrupt is masked when it is disabled or ignored by the processor. Contrarily, an interrupt is unmasked if it is unable to be disabled or ignored by the processor.

Conclusion

The purpose of this lab was to grow our understanding of interrupts and processor scheduling with a Raspberry Pi, and how to create and use a local timer to generate interrupts, use priority, and keep track of the processor state and memory address of kernel functions. This lab also served as a wonderful guide on Interrupts and their utilities as seen throughout the provided reading material.

References

BCM2837 ARM Peripherals: https://cs140e.sergio.bz/docs/BCM2837-ARM-Peripherals.pdf