EE382N-4
Advanced Microcontroller Systems

Interrupts, Interrupt Handlers & Signals

Mark McDermott

Outline of This Lecture

- **Interrupts**
- **Interrupt handlers**
- **Writing an interrupt handler for the Zynq-UltraScale+**
- **Signals**

- **References:**

  Sreekrishnan Venkateswaran Essential Linux Device Drivers

  Karim Yaghmour Building Embedded Linux Systems

  Jonathon Corbet Linux Device Drivers

Review: Programmable Logic –> GIC Interrupts

- The PL can asynchronously assert up to 20 interrupts to the PS.
  - 16 interrupt signals are mapped to the interrupt controller as a peripheral interrupt where each interrupt signal is set to a priority level and mapped to one or both CPUs.
  - The remaining four PL interrupt signals are inverted and routed to the nFIQ and nIRQ interrupts which are sent directly to the private peripheral interrupt (PPI) unit of the interrupt controller. There is an nFIQ and nIRQ interrupt for each of the four CPUs.

S.No	Interrupt ID	Interrupt Name	Description	Type
PL-PS Interrupts (Interrupts that go from PL to PS)				
1	121-128, 136-143 (See Table 2-1 for Pin mapping to each interrupt ID)	IRQ-F2P[15:0]	Shared Interrupts from PL logic to GICs of real-time processing unit (RPU) or application processing unit (APU)	Shared Interrupts
2	31	A53-Core_0 nIRQ	Cortex™ A53 Core0 Private Peripheral Legacy IRQ Interrupt	Private Peripheral Interrupt
3	31	A53-Core_1 nIRQ	Cortex A53 Core1 Private Peripheral Legacy IRQ Interrupt	Private Peripheral Interrupt
4	31	A53-Core_2 nIRQ	Cortex A53 Core2 Private Peripheral Legacy IRQ Interrupt	Private Peripheral Interrupt
5	31	A53-Core_3 nIRQ	Cortex A53 Core3 Private Peripheral Legacy IRQ Interrupt	Private Peripheral Interrupt
6	28	A53-Core_0 nFIQ	Cortex A53 Core0 Private Peripheral Legacy FIQ Interrupt	Private Peripheral Interrupt
7	28	A53-Core_1 nFIQ	Cortex A53 Core1 Private Peripheral Legacy FIQ Interrupt	Private Peripheral Interrupt
8	28	A53-Core_2 nFIQ	Cortex A53 Core2 Private Peripheral Legacy FIQ Interrupt	Private Peripheral Interrupt
9	28	A53-Core_3 nFIQ	Cortex A53 Core3 Private Peripheral Legacy FIQ Interrupt	Private Peripheral Interrupt
PS -PL Interrupts (Interrupts coming from PS to PL)				
1	1	IRQ_P2F_RPU Performance Monitor 0	RPU Performance Monitor 0 Interrupt	Shared Interrupt
2	1	IRQ_P2F_RPU Performance Monitor 1	RPU Performance Monitor 1 Interrupt	Shared Interrupt
3	1	IRQ_P2F_OCM Error	On-chip RAM (OCM) Error Interrupt	Shared Interrupt
4	1	IRQ_P2F_LPD APB Interrupts	OR of all AMBA peripheral bus (APB) interrupts from LPD. Refer to the technical reference manual for APB Interrupt and Register Information.	Shared Interrupt

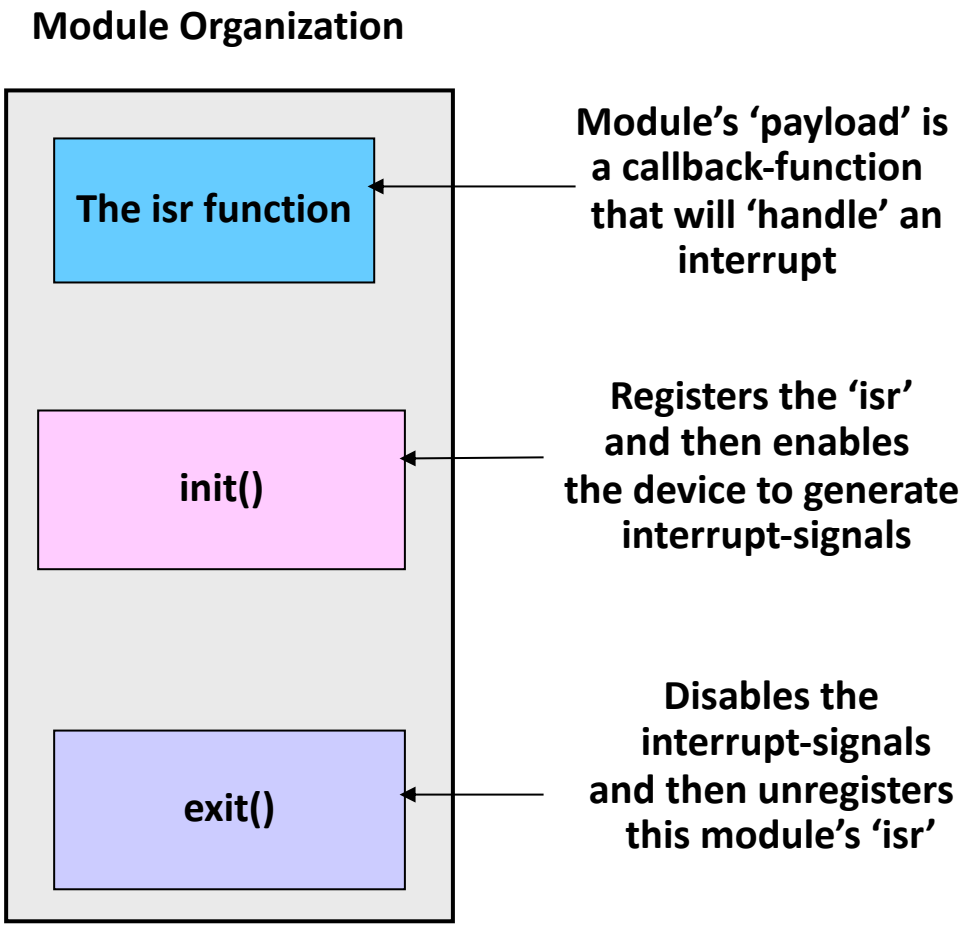
Interrupt Handlers

Jumping to the Interrupt Handler

- **Auto-vectored**
  - Processor-determined address of interrupt handler based on type of interrupt
- **Vectored**
  - Device supplies processor with address of interrupt handler
- **Why the different methods?**
  - If multiple devices uses the same interrupt type (IRQ vs. FIQ), in an Auto-vectored system the processor must poll each device to determine which device interrupted the processor
    - This can be time-consuming if there is a lot of devices
  - In a vectored system, the processor would just take the address from the device (which dumps the interrupt vector onto a special bus).

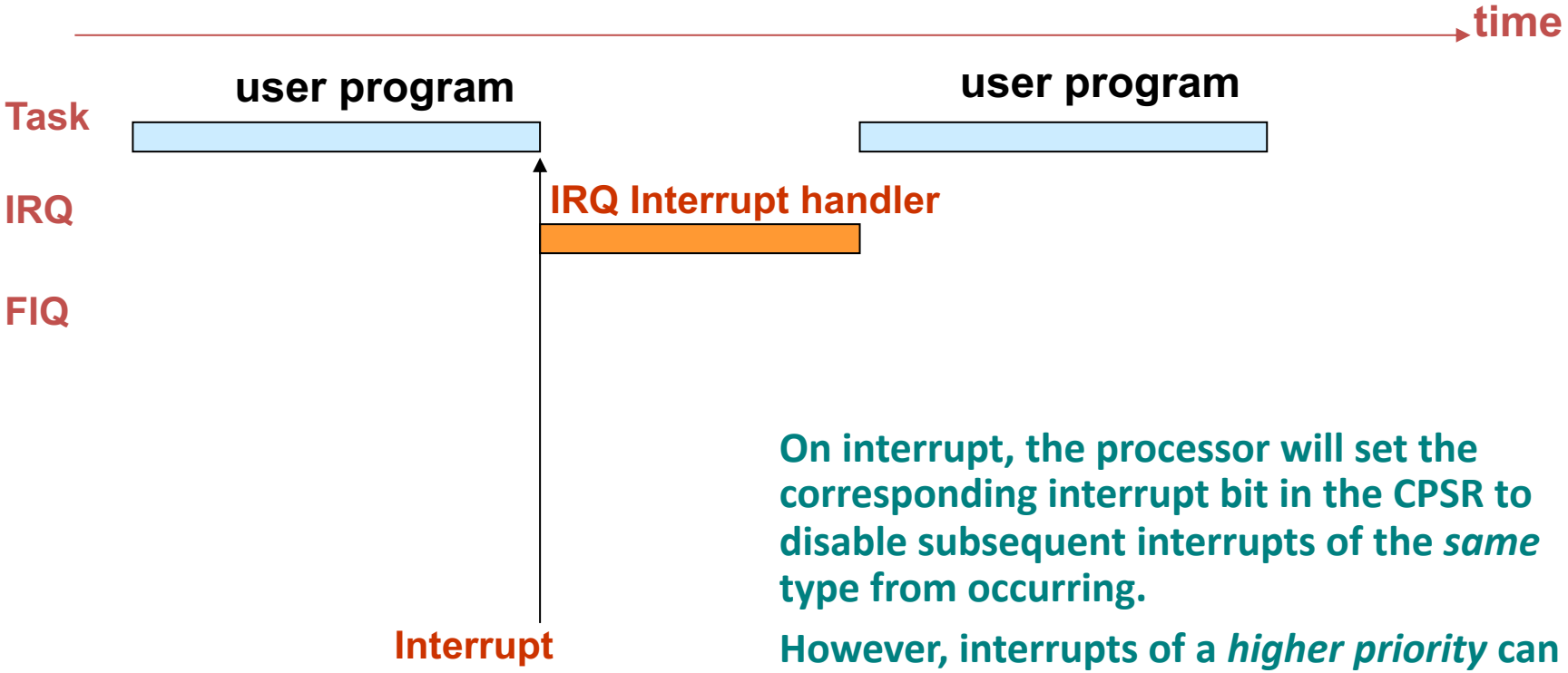
Interrupt Handlers

- **Kernel routine that runs in response to interrupt.**
  - More than one handler can exist per IRQ.
- **Must run quickly.**
  - Resume execution of interrupted code.
  - How to deal with high work interrupts?
    - Ex: network, hard disk



Interrupt Handlers

- **When an interrupt occurs, the hardware will jump to an "interrupt handler"**

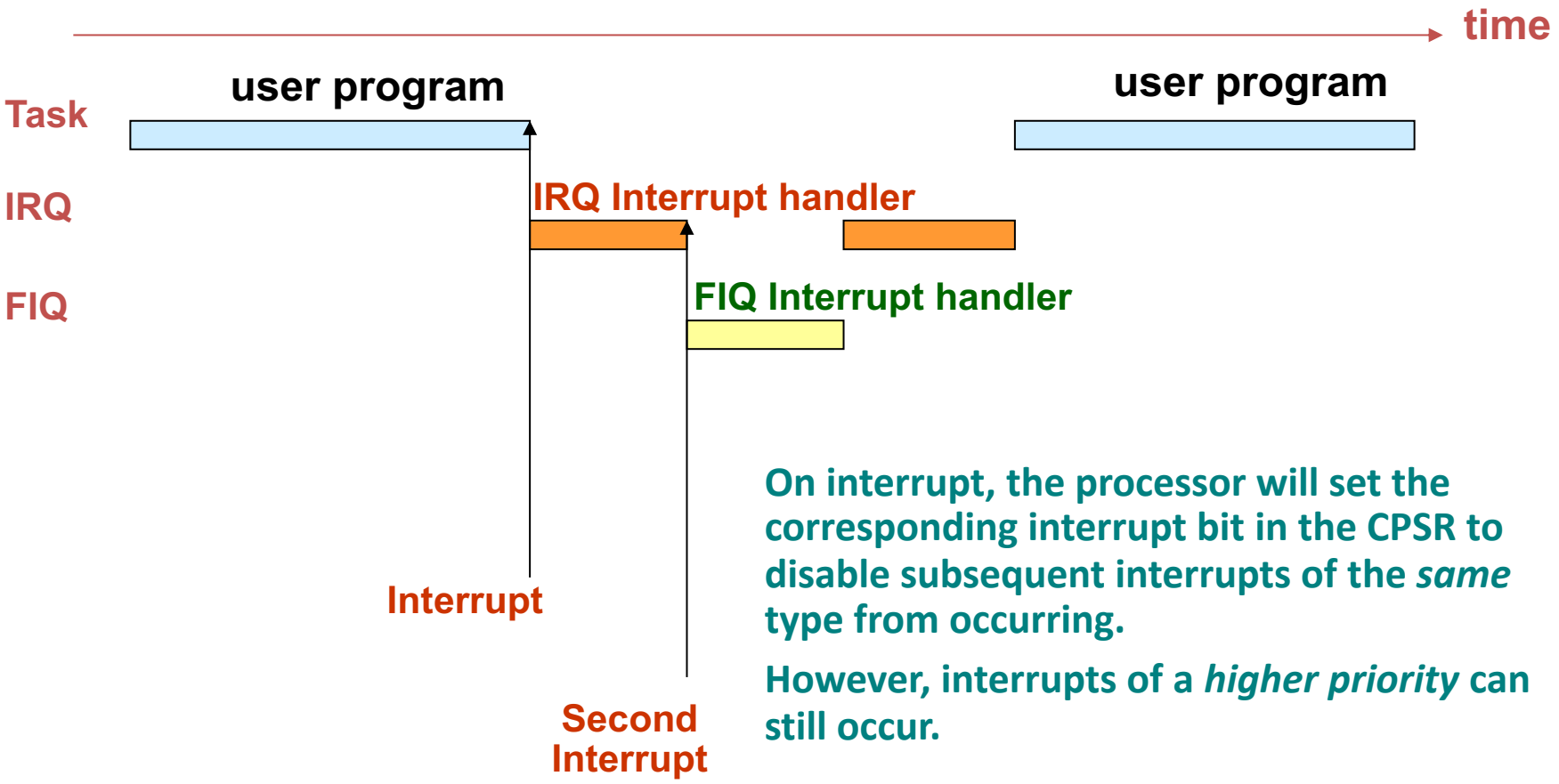


On interrupt, the processor will set the corresponding interrupt bit in the CPSR to disable subsequent interrupts of the *same* type from occurring.

However, interrupts of a *higher priority* can still occur.

Nested/Re-entrant Interrupts

- Interrupts can occur within interrupt handlers

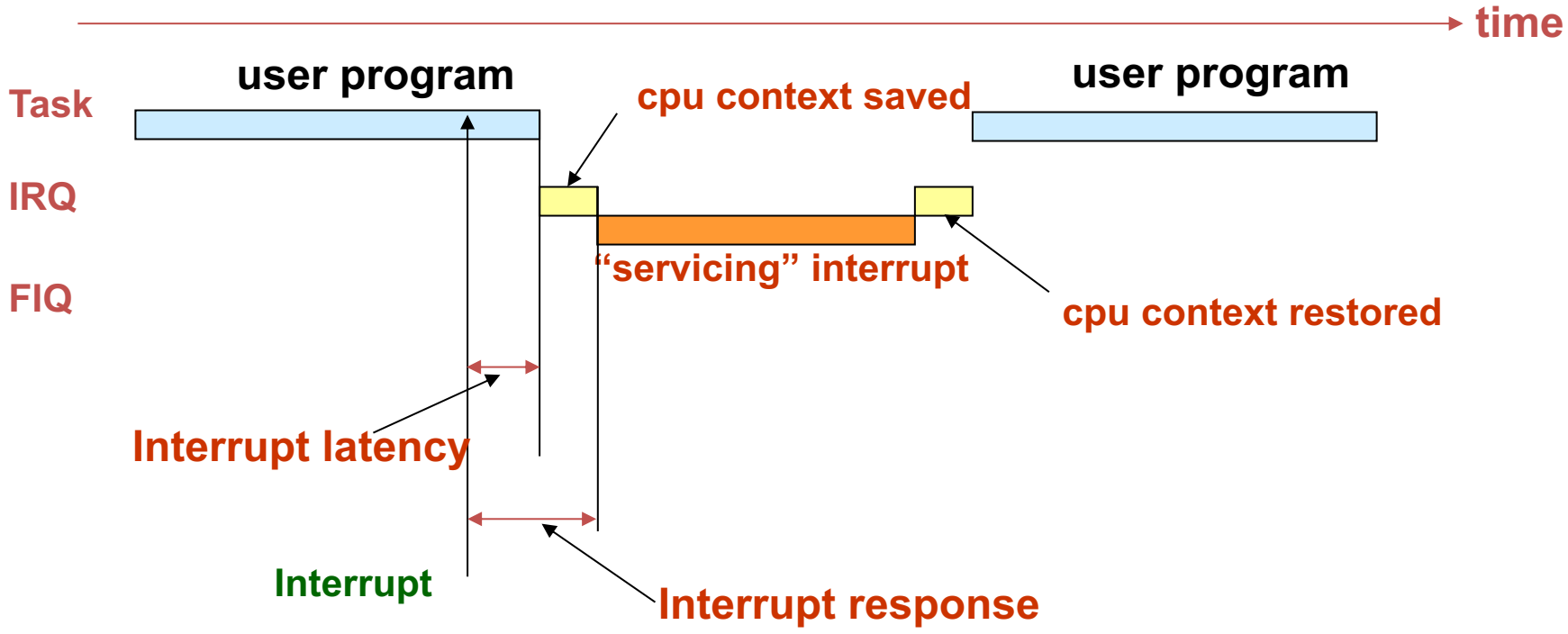


On interrupt, the processor will set the corresponding interrupt bit in the CPSR to disable subsequent interrupts of the *same* type from occurring.

However, interrupts of a *higher priority* can still occur.

Timing of Interrupts

- Before an interrupt handler can do anything, it must save away the current program's registers (if it touches those registers)
- That's why the FIQ saves lots of extra registers to minimize CPU context saving overhead



Interrupt Context

- **Not associated with a process.**
  - Cannot sleep: no task to reschedule.
  - Current macro points to interrupted process.

- **Shares kernel stack of interrupted process.**
  - Be very frugal in stack usage.

Registering an IRQ Handler

request_irq(…)

- Register an interrupt handler for a given interrupt input pin.

```
#include <linux/interrupt.h>

int request_irq(unsigned int   irq,
                irq_handler_t  handler,
                unsigned long  flags,
                const char    *dev_name,
                void          *dev_id)
```

```
irq:      the requested interrupt number (from DTB probe function)
handler:  the interrupt handler function pointer
dev_name: for /proc/interrupts
dev_id:   pointer for shared interrupt lines (can be set to NULL if not shared)
flags:
          IRQF_SHARED signals that the interrupt can be shared
          IRQF_TRIGGER_RISING signals that the interrupt is rising edge active
          IRQF_TRIGGER_FALLING signals that the interrupt is falling edge active
```

free_irq(…)

- Unregister a given interrupt handler.
- Disable interrupt line if all handlers unregistered.

Installing an Interrupt Handler

- On success, `request_irq(…)` returns zero
  - Common error is –EBUSY, which denotes given interrupt line is already in use and no sharing
- `request_irq(…)` can sleep
- Make sure device is completely set up before calling request_irq
  - Don't want to start servicing interrupts before device is ready

Do’s and Don'ts of Interrupt Handlers

- It's a programming offense if your interrupt context code goes to sleep. Interrupt handlers cannot relinquish the processor by calling sleepy functions such as schedule_timeout().

- For protecting critical sections inside interrupt handlers, you can't use mutexes because they may go to sleep. Use spinlocks instead and use them only if you must.

- Interrupt handlers are supposed to get out of the way quickly but are expected to get the job done. To circumvent this Catch-22, interrupt handlers split their work into two halves: top (slim) and bottom (fat).

From: Essential Linux Device Drivers - Venkateswaran

Do’s and Don'ts of Interrupt Handlers

- You do NOT need to design interrupt handlers to be reentrant. When an interrupt handler is running, the corresponding IRQ is disabled until the handler returns.

- Interrupt handlers can be interrupted by handlers associated with IRQs that have higher priority. You can prevent this nested interruption by specifically requesting the kernel to treat your interrupt handler as a “fast handler”.

From: Essential Linux Device Drivers - Venkateswaran

Writing an interrupt handler for the Zynq UltraScale+

Setup

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <linux/version.h>
#include <linux/errno.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/interrupt.h>
#include <linux/sched.h>
#include <asm/uaccess.h>
#include <linux/io.h>
#include <linux/module.h>
#include <asm/gpio.h>
#include <linux/vmalloc.h>
#include <linux/mman.h>
#include <linux/slab.h>
#include <linux/ioport.h>
#include <linux/platform_device.h>
#include <linux/bitops.h>
#include <linux/clk.h>
#include <linux/gpio/driver.h>
#include <linux/pm_runtime.h>
#include <linux/of.h>

#define MODULE_VER "1.0"
#define GPIO_MAJOR 240              // Need to run mknod /dev/gpio_int c 240 0
#define MODULE_NM "gpio_interrupt"

#undef DEBUG
//#define DEBUG                       // Uncomment to enable debug mode

int             interruptcount  = 0;
int             temp            = 0;
int             len             = 0;
unsigned int    gic_interrupt;


static struct proc_dir_entry *proc_gpio_int;
static struct fasync_struct *fasync_gpio_queue ;
```

```
root@ultra96_McD:/media/mcdermot/BOOT# ls -all /dev/gpio_int
crw-r--r-- 1 root root 240, 0 Feb 14 10:22 /dev/gpio_int
```

What happens if the Major number is taken??

- To determine which Major numbers are taken, execute:
  more /proc/devices
- Select 237
  - Must be a number between 234 - 254
- On our board 240 was taken

```
226 drm
237 gpio_interrupt
238 ttyPS
239 xlnxsync
240 xlnk
241 dma_interrupt
242 ttyGS
243 uio
244 bsg
245 watchdog
246 iio
247 ptp
248 pps
249 media
250 rtc
251 dma_heap
252 tpm
253 aie
254 gpiochip
```

Initializing the driver

```
static struct platform_driver zynq_gpio_driver = {
    .driver    = {
        .name = MODULE_NM,
        .of_match_table = zynq_gpio_of_match,
    },
    .probe = zynq_gpio_probe,
    .remove = zynq_gpio_remove,
};

/* ---------------------------------------------------------------------
 * function: init_gpio_int
 *
 * This function creates the /proc directory entry gpio_interrupt.
 */

static int __init init_gpio_int(void)
{

    int rv = 0;
    int err = 0;

    //platform_driver_unregister(&zynq_gpio_driver);


    printk("Ultra96 Interrupt Module\n");
    printk("Ultra96 Interrupt Driver Loading.\n");
    printk("Using Major Number %d on %s\n", GPIO_MAJOR, MODULE_NM);


    err = platform_driver_register(&zynq_gpio_driver);

    if(err !=0) printk("Driver register error with number %d\n",err);
    else        printk("Driver registered with no error\n");

    if (register_chrdev(GPIO_MAJOR, MODULE_NM, &gpio_fops)) {
        printk("gpio_int: unable to get major %d. ABORTING!\n", GPIO_MAJOR);
    goto no_gpio_interrupt;
    }
```

do not do this if the driver is not registered

Will Crash

The *&gpio_fops* pointer is used to point to the routines that are called when the device is accessed from the user application.

Initializing the driver (cont.)

```
    proc_gpio_int = proc_create("gpio_interrupt", 0444, NULL, &proc_fops );

    if(proc_gpio_int == NULL) {
          printk("gpio_int: create /proc entry returned NULL. ABORTING!\n");
    goto no_gpio_interrupt;
    }


    // Request interrupt

    rv = request_irq(gic_interrupt,
                     (irq_handler_t) gpio_int_handler,
                      IRQF_TRIGGER_RISING,
                     "gpio_interrupt", NULL);

    if ( rv ) {
        printk("Can't get interrupt %d\n", gic_interrupt);
    goto no_gpio_interrupt;
    }

    printk(KERN_INFO "%s %s Initialized\n",MODULE_NM, MODULE_VER);

    return 0;

    // remove the proc entry on error

no_gpio_interrupt:
    free_irq(gic_interrupt,NULL);                    // Release IRQ
    unregister_chrdev(GPIO_MAJOR, MODULE_NM);        // Release character device
    platform_driver_unregister(&zynq_gpio_driver);
    remove_proc_entry("gpio_interrupt", NULL);
    return -EBUSY;
};
```

This is the Interrupt # from DTB probe function (see next slide)

This is the Interrupt Handler

This is edge or level sensitivity

DTB probe mechanism for getting interrupt number

```
  struct  proc_ops proc_fops = {     // NEW for 5.15.36 kernel
  };


  static const struct of_device_id zynq_gpio_of_match[] = {
      { .compatible = "xlnx,ctmr-intout" },
      { /* end of table */ }
  };

  MODULE_DEVICE_TABLE(of, zynq_gpio_of_match);


/* ----------------------------------------------------------
 * zynq_gpio_probe - Initialization method for a ZYNQ gpio device
 * Return: 0 on success, negative error otherwise.
 */

static int zynq_gpio_probe(struct platform_device *pdev)
{
    struct resource *res;

    // This code gets the IRQ number by probing the system.

    res = platform_get_resource(pdev, IORESOURCE_IRQ, 0);

    if (!res) {
        printk("No IRQ found\n");
        return 0;
    }

    // Get interrupt number
    gic_interrupt  = res->start;

    return 0;


}
```

"compatible" values must be identical

DTB entry for lab_2_timer

```
Lab_2_Timer@a0030000 {
  clock-names = "s00_axi_aclk";
  clocks = <0x3 0x47>;
  compatible = "xlnx,ctmr-intout";
  interrupt-names = "interrupt_out";
  interrupt-parent = <0x4>;
  interrupts = <0x0 0x5c 0x4>;
```

gic_interrupt = 0x20 + 0x5c = 0x7c
= 32 + 94 = 124 (decimal)

Setting up the interrupt handler

```
static struct fasync_struct *fasync_gpio_queue ;

/* ==================================================================
 * function: gpio_int_handler
 *
 * This function is the gpio_interrupt handler. It sets the tv2
 * structure using do_gettimeofday.
 */


irq_handler_t gpio_int_handler(int irq, void *dev_id, struct pt_regs *regs)
{
  interruptcount++;

#ifdef DEBUG
   printk(KERN_INFO "gpio_int: Interrupt detected in kernel \n");  // DEBUG
#endif

/* Signal the user application that an interrupt occurred */

  kill_fasync(&fasync_gpio_queue, SIGIO, POLL_IN);

return  (irq_handler_t) IRQ_HANDLED;

}
```

More about kill_fasync() later

dmesg output after the interrupt handler has been installed:

```
[78892.065284] Ultra96 GPIO Interrupt Module
[78892.065292] Ultra96 GPIO Interrupt Driver Loading.
[78892.065299] Using Major Number 237 on gpio_interrupt
[78892.065496] In probe function
[78892.065504] Probe IRQ # = 67
[78892.065643] Driver registered with no error
[78892.065745] gpio_interrupt 1.0 Initialized
```

This is the Linux interrupt number NOTE: Varies depending on number of modules

Assigning Interrupt Numbers on the Zynq-UltraScale+

	CPU0	CPU1	CPU2	CPU3			
3:	1229843	1388132	1325883	1314033	GICv2 30 Level	arch_timer	PPI
6:	0	0	0	0	GICv2 67 Level	ff9905c0.mailbox	
7:	0	0	0	0	GICv2 175 Level	arm-pmu	
8:	0	0	0	0	GICv2 176 Level	arm-pmu	
9:	0	0	0	0	GICv2 177 Level	arm-pmu	
10:	0	0	0	0	GICv2 178 Level	arm-pmu	
12:	0	0	0	0	GICv2 156 Level	zynqmp-dma	
18:	0	0	0	0	GICv2 162 Level	zynqmp-dma	
19:	0	0	0	0	GICv2 163 Level	zynqmp-dma	
20:	0	0	0	0	GICv2 164 Level	Mali_GP	
21:	0	0	0	0	GICv2 109 Level	zynqmp-dma	
22:	0	0	0	0	GICv2 110 Level	zynqmp-dma	
23:	0	0	0	0	GICv2 111 Level	zynqmp-dma	
27:	0	0	0	0	GICv2 115 Level	zynqmp-dma	
28:	0	0	0	0	GICv2 116 Level	zynqmp-dma	SPI
31:	5	0	0	0	GICv2 50 Level	cdns-i2c	
32:	0	0	0	0	GICv2 42 Level	ff960000.memory-controller	
33:	0	0	0	0	GICv2 57 Level	axi-pmon	
34:	0	0	0	0	GICv2 58 Level	ffa60000.rtc	
35:	0	0	0	0	GICv2 59 Level	ffa60000.rtc	
36:	30234	0	0	0	GICv2 80 Level	mmc0	
37:	93	0	0	0	GICv2 81 Level	mmc1	
38:	0	0	0	0	GICv2 51 Level	ff040000.spi	
39:	0	0	0	0	GICv2 52 Level	ff050000.spi	
41:	2045	0	0	0	GICv2 54 Level	xuartps	
43:	0	0	0	0	GICv2 84 Edge	ff150000.watchdog	
44:	0	0	0	0	GICv2 88 Level	ams-irq	
45:	0	0	0	0	GICv2 154 Level	fd4c0000.dma	
46:	0	0	0	0	GICv2 151 Level	fd4a0000.zynqmp-display	
47:	0	0	0	0	GICv2 61 Level	ff9a0100.zynqmp_r5_rproc	
67:	195639382	0	0	0	GICv2 124 Edge	gpio_interrupt	
70:	0	0	0	0	GICv2 123 Edge	cdma-controller	
73:	1398110	0	0	0	GICv2 102 Level	xhci-hcd:usb1	
77:	0	0	0	0	gpio 23 Edge	sw4	

Linux Interrupt Number

Number of Interrupts per Processor

GIC Interrupt Number

Peripheral Component

Asynchronous Notification

- Polling is inefficient for asynchronous events such as interrupts.
- Solution: Asynchronous notification
  - Application receives a signal whenever data becomes available
  - Two steps
    - Specify a process as the owner of the file (so that the kernel knows whom to notify)
    - Set the FASYNC flag in the device via fcntl() command from the user application system calls:
      - /* create a signal handler */
        signal(SIGIO, &input_handler);
      - /* set current pid the owner of the stdin */
        fcntl(FILE_DESCRIPTOR, F_SETOWN, getpid());
      - /* obtain the current file control flags */
        oflags = fcntl(FILE_DESCRIPTOR, F_GETFL);
      - /* set the asynchronous flag */
        fcntl(FILE_DESCRIPTOR, F_SETFL, oflags | FASYNC);

gpio_fasync() routine

This is invoked by the kernel when the user program opens the /dev/gpio_int device and issues fcntl(F_SETFL) on the associated file descriptor.

fasync_helper() ensures that if the driver issues a kill_fasync(), a SIGIO is dispatched to the owning application.

```
* ============================================================================
* function: gpio_fasync
*
* This is invoked by the kernel when the user program opens this
* input device and issues fcntl(F_SETFL) on the associated file
* descriptor. fasync_helper() ensures that if the driver issues a
* kill_fasync(), a SIGIO is dispatched to the owning application.
*/


static int gpio_fasync (int fd, struct file *filp, int on)
{
        #ifdef DEBUG
           printk(KERN_INFO "\ngpio_int: Inside gpio_fasync \n");  // DEBUG
        #endif

   return fasync_helper(fd, filp, on, &fasync_gpio_queue);
};
```

`kill_fasync()` routine

- `kill_fasync()` is used to signal the interested process(es) when data arrives. "kill" is actually a misnomer. This function asynchronously delivers the SIGIO signal to the processes which requested it. Since the default action performed when receiving a signal is to terminate (aka, kill).

- The arguments are the signal to send (usually SIGIO) and the band, which is almost always POLL_IN plus a pointer to the queue with the list of processes to be notified "`fasync_gpio_queue`"

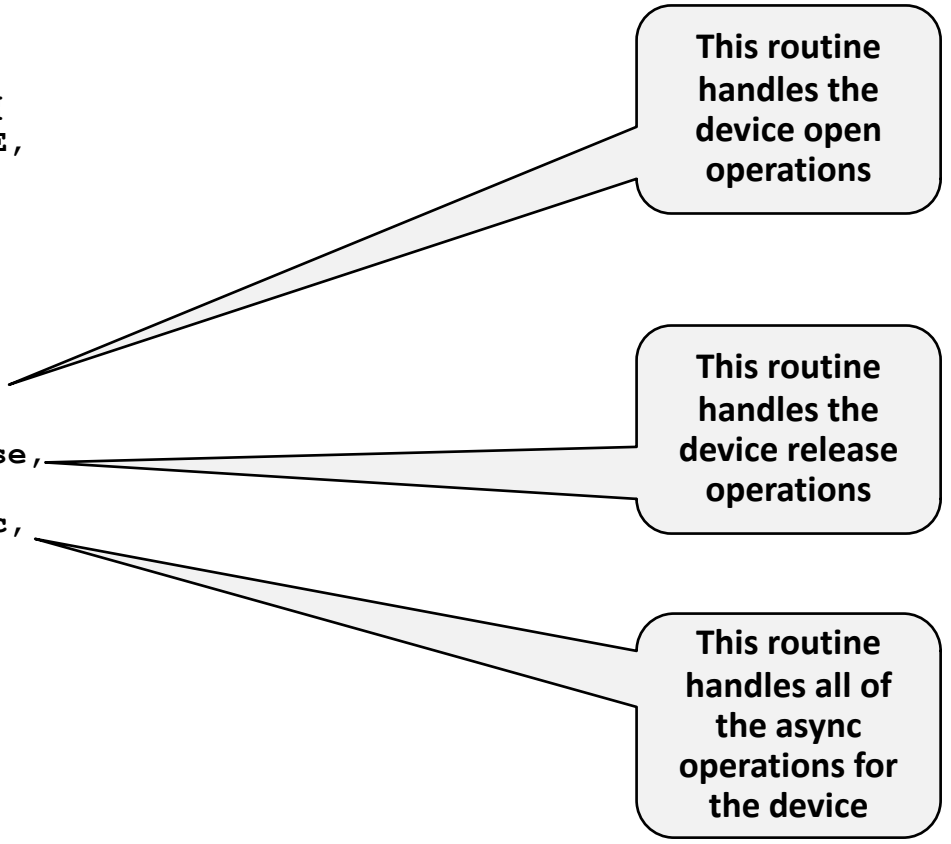
- Usage:

```
kill_fasync(&fasync_gpio_queue,  SIGIO,  POLL_IN);
```

Setting up the file handling operations

- **The next step is to assign routines to handle the various device calls through the *gpio_fops* structure.**

```
/*
*  Define which file operations are supported
*
*/

struct file_operations gpio_fops = {
    .owner          =    THIS_MODULE,
    .llseek         =    NULL,
    .read           =    NULL,
    .write          =    NULL,
    .poll           =    NULL,
    .unlocked_ioctl =    NULL,
    .mmap           =    NULL,
    .open           =    gpio_open,
    .flush          =    NULL,
    .release        =    gpio_release,
    .fsync          =    NULL,
    .fasync         =    gpio_fasync,
    .lock           =    NULL,
};
```



Miscellaneous functions

```
/* ================================================================
 * function: gpio_open
 *
 * This function is called when the gpio_int device is opened
 *
 */

static int gpio_open (struct inode *inode, struct file *file) {

// Add specific open code here

#ifdef DEBUG
        printk(KERN_INFO "gpio_int: Inside gpio_open \n");  // DEBUG
#endif
    return 0;
}



/* ================================================================
 * function: gpio_release
 *
 * This function is called when the gpio_int device is
 * released
 *
 */

static int gpio_release (struct inode *inode, struct file *file) {

// Add specific release code here


#ifdef DEBUG
        printk(KERN_INFO "\ngpio_int: Inside gpio_release \n");  // DEBUG
#endif
    return 0;
}
```

Miscellaneous functions (cont.)

```
static struct proc_dir_entry *proc_gpio_int;


/* ----------------------------------------------------------------------------
*    function: read_proc   --- Example code
*/

ssize_t read_proc(struct file *filp,char *buf,size_t count,loff_t *offp )
{

    count = count - 1;
    printk("read_proc count value = %ld\n", count);

return count;
}




/* ----------------------------------------------------------------------------
*    function: write_proc   --- Example code
*/


ssize_t write_proc(struct file *filp,const char *buf,size_t count,loff_t *offp)
{
    count = count + 1;
    printk("write_proc count value = %ld\n", count);


return count;
}
```

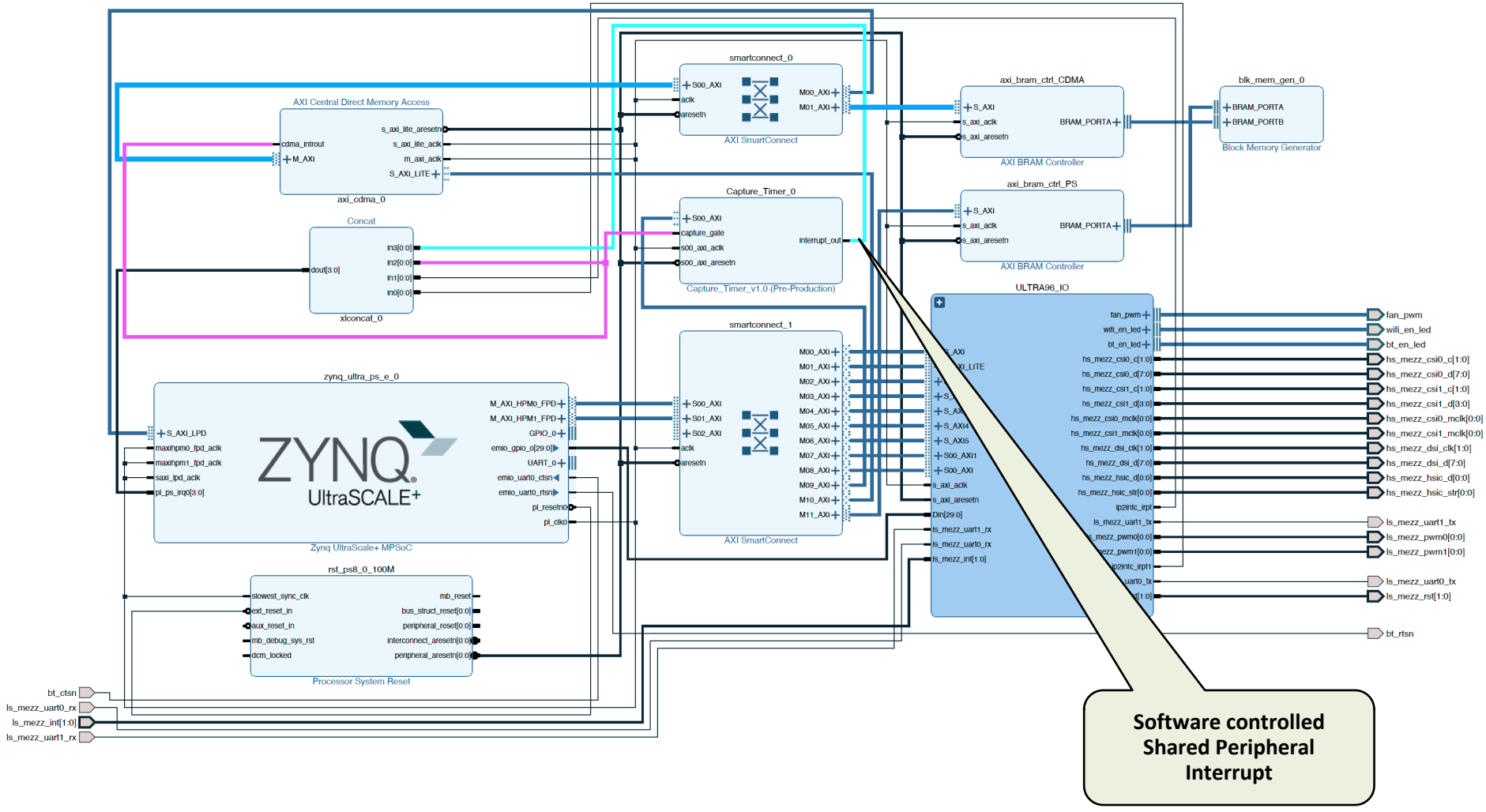
Miscellaneous functions (cont.)

```
/* ================================================================
 *
 * zynq_gpio_remove - Driver removal function
 *
 * Return: 0 always
 */

static int zynq_gpio_remove(struct platform_device *pdev)
{
    //struct zynq_gpio *gpio = platform_get_drvdata(pdev)
    return 0;
}

/* ================================================================
 * function: cleanup_gpio_interrupt
 *
 * This function frees interrupt then removes the /proc directory entry
 * gpio_interrupt.
 */

static void __exit cleanup_gpio_interrupt(void)
{

    free_irq(gic_interrupt,NULL);                    // Release IRQ
    unregister_chrdev(GPIO_MAJOR, MODULE_NM);        // Release character devie
    platform_driver_unregister(&zynq_gpio_driver);   // Unregister the driver
    remove_proc_entry("gpio_interrupt", NULL);       // Remove process entry

    printk(KERN_INFO "%s %s removed\n", MODULE_NM, MODULE_VER);
}

/* REQUIRED */
module_init(init_gpio_int);
module_exit(cleanup_gpio_interrupt);

MODULE_AUTHOR("Mark McDermott");
MODULE_DESCRIPTION("gpio_interrupt proc module");
MODULE_LICENSE("GPL");
```

Let's write a user application that uses the gpio_interrupt handler to measure interrupt latency through the kernel

Interrupt latency measurement routine

- This routine measures the interrupt latency through the Linux Kernel. The block diagram for the PS/PL configuration is:



Interrupt latency measurement routine

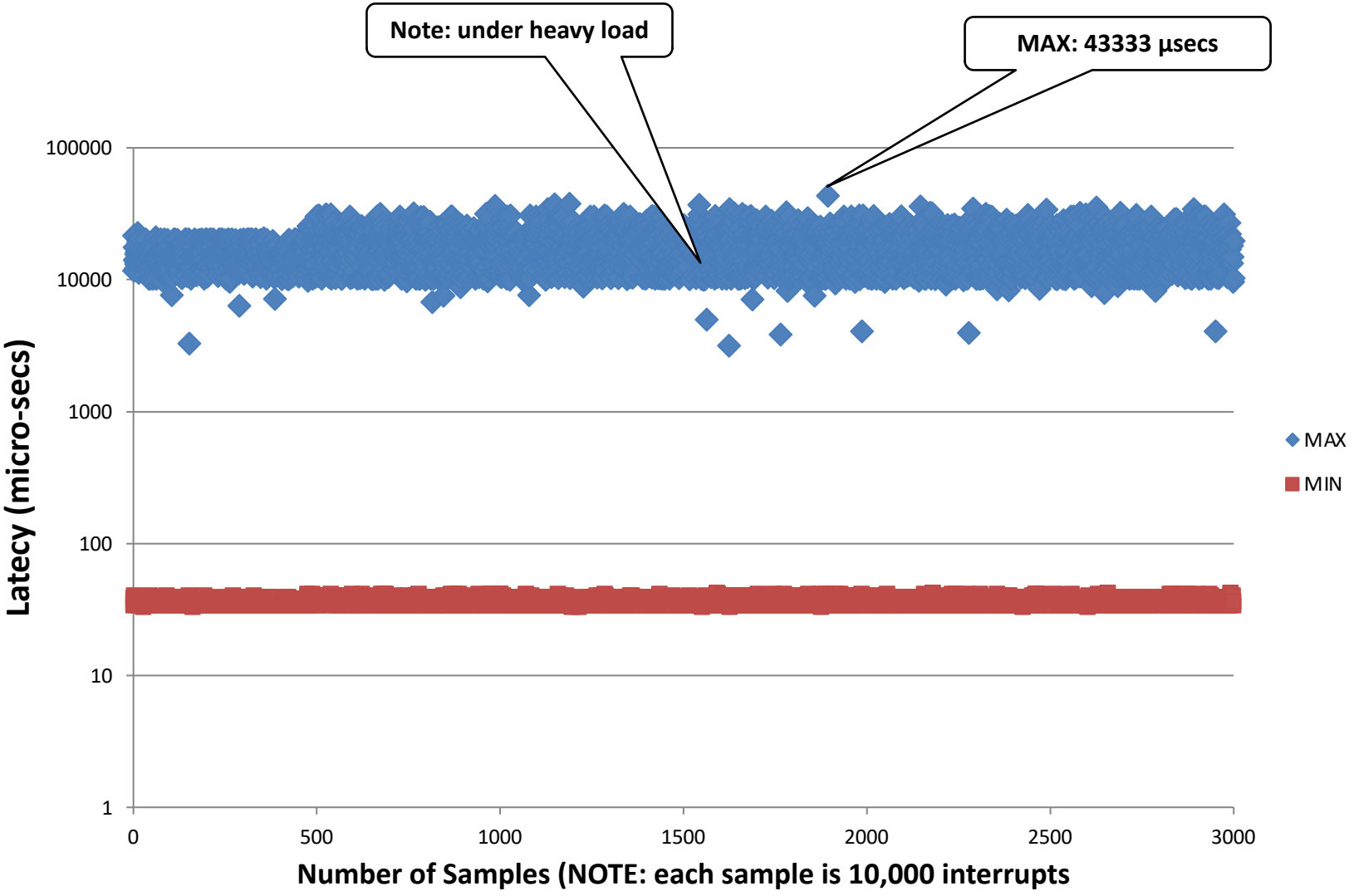
- **Routine measures the interrupt latency time through the kernel and reports the MAX, MIN, AVE and STDDEV times.**

```
************************************************************

Minimum Latency:    2205
Maximum Latency:    39236
Average Latency:    2538.000000
Standard Deviation: 956.000000
Number of samples:  10000
 67:     415977     GICv2 124 Edge      gpio_interrupt
************************************************************

Minimum Latency:    2143
Maximum Latency:    20221
Average Latency:    2486.000000
Standard Deviation: 685.000000
Number of samples:  10000
 67:     425977     GICv2 124 Edge      gpio_interrupt
************************************************************

Minimum Latency:    2199
Maximum Latency:    33927
Average Latency:    2521.000000
Standard Deviation: 841.000000
Number of samples:  10000
 67:     435977     GICv2 124 Edge      gpio_interrupt
************************************************************
```

Min/Max/Average Latency results for 10000 interrupts

Total number of interrupts from /proc/interrupts

Interrupt latency measurement results (3M interrupts)



Setting up the user space application:

- The user space setup involves setting up the actions to be performed and opening the appropriate device.

```
int main(void)
{
    volatile int rc;
    int i;
    struct timeval start_timestamp;

    /* ---------------------------------------------------------------------
     * Register signal handler for SIGIO signal:
     */

    struct sigaction sig_action;

    memset(&sig_action, 0, sizeof sig_action);
    sig_action.sa_handler = sigio_signal_handler;


    /* ---------------------------------------------------------------------
     * Block all signals while our signal handler is executing:
     */

    (void)sigfillset(&sig_action.sa_mask);


    rc = sigaction(SIGIO, &sig_action, NULL);


    if (rc == -1) {
        perror("sigaction() failed");
        return -1;
    }
```

Setting up the user space application (cont.)

```
/* ------------------------------------------------------------------------
 * Open the device file
 */
gpio_dev_fd = open(GPIO_DEV_PATH, O_RDWR);


 if(gpio_dev_fd == -1)    {
    perror("open() of " GPIO_DEV_PATH " failed");
    return -1;
}

/* ------------------------------------------------------------------------
 * Set our process to receive SIGIO signals from the GPIO device:
 */
rc = fcntl(gpio_dev_fd, F_SETOWN, getpid());

if (rc == -1) {
    perror("fcntl() SETOWN failed\n");
    return -1;
}

/* ------------------------------------------------------------------------
 * Enable reception of SIGIO signals for the gpio_dev_fd descriptor
 */

int fd_flags = fcntl(gpio_dev_fd, F_GETFL);
    rc = fcntl(gpio_dev_fd, F_SETFL, fd_flags | O_ASYNC);


if (rc == -1) {
    perror("fcntl() SETFL failed\n");
    return -1;
}
rc = gpio_set_pin(GPIO_DR, GPIO_PIN_NUM, 0);  // Clear output pin
if (rc != 0) {
    perror("gpio_set_pin() failed");
return -1;
}
```

`fcntl()` routine

`fcntl(int fd, int cmd)` manipulate open file descriptors. It performs one of various miscellaneous operations on fd. The operation in question is determined by cmd:

- **F_GETFL: Read the file descriptor's flags.**
- **F_SETFL: Set the file status flags part of the descriptor's flags to the value specified by arg. Remaining bits (access mode, file creation flags) in arg are ignored. On Linux this command can only change the O_APPEND, O_NONBLOCK, O_ASYNC, and O_DIRECT flags.**
- **F_SETOWN: Set the process ID or process group that will receive SIGIO and SIGURG signals for events on file descriptor fd.**

Signal Handling

- **The signal handling routine checks to see if the correct signal arrived.**

```
/* ------------------------------------------------------------------------
 * SIGIO signal handler
 */

void sigio_signal_handler(int signo)
{
    volatile int rc1;


    assert(signo == SIGIO);   // Confirm correct signal #
    sigio_signal_count ++;


    printf("sigio_signal_handler called (signo=%d)\n", signo);   // DEBUG


    /* ------------------------------------------------------------------------
     * Set global flag
     */

    sigio_signal_processed = 1;

    /* ------------------------------------------------------------------------
     * Take end timestamp for interrupt latency measurement
     */

     (void)gettimeofday(&sigio_signal_timestamp, NULL);

}
```

Measurement loop

```
/* ------------------------------------------------------------------
 * Take interrupt latency measurements in a loop:
 */
sigset_t signal_mask, signal_mask_old, signal_mask_most;

for (i = 0; i < NUM_MEASUREMENTS; i ++) {

    /* ------------------------------------------------------------------
     * Reset sigio_signal_processed flag:
     */

    sigio_signal_processed = 0;

    /* ------------------------------------------------------------------
     * NOTE: This next section of code must be executed each cycle to prevent
     * a race condition between the SIGIO signal handler and sigsuspend()
     */

    (void)sigfillset(&signal_mask);
    (void)sigfillset(&signal_mask_most);
    (void)sigdelset(&signal_mask_most, SIGIO);
    (void)sigprocmask(SIG_SETMASK,&signal_mask, &signal_mask_old);

    /* ------------------------------------------------------------------
     * Take a start timestamp for interrupt latency measurement
     */
    (void)gettimeofday(&start_timestamp, NULL);

    /* ------------------------------------------------------------------
     * Assert GPIO output pin to trigger generation of edge sensitive interrupt:
     */

    rc = gpio_set_pin(GPIO_DR, GPIO_PIN_NUM, 1);
    if (rc != 0) {
        perror("gpio_set_pin() failed");
    return -1;
    }
```

Measurement loop (cont.)

```
    /* ------------------------------------------------------------------
     * Wait for SIGIO signal handler to be executed.
     */
    if (sigio_signal_processed == 0) {


        rc = sigsuspend(&signal_mask_most);

        /* Confirm we are coming out of suspend mode correcly */
        assert(rc == -1 && errno == EINTR && sigio_signal_processed);
    }

    (void)sigprocmask(SIG_SETMASK, &signal_mask_old, NULL);

    assert(sigio_signal_count == i + 1);


    rc = gpio_set_pin(GPIO_DR, GPIO_PIN_NUM, 0);

    if (rc != 0) {
        perror("gpio_set_pin() failed");
    return -1;
    }

    /* ------------------------------------------------------------------
     * Compute interrupt latency:
     */
    intr_latency_measurements[i] =
        (sigio_signal_timestamp.tv_sec -
         start_timestamp.tv_sec) * 1000000 +
        (sigio_signal_timestamp.tv_usec -
         start_timestamp.tv_usec);

}  // End of for loop
```

SUPER Critical Assertion

Signals

Thanks to Princeton University

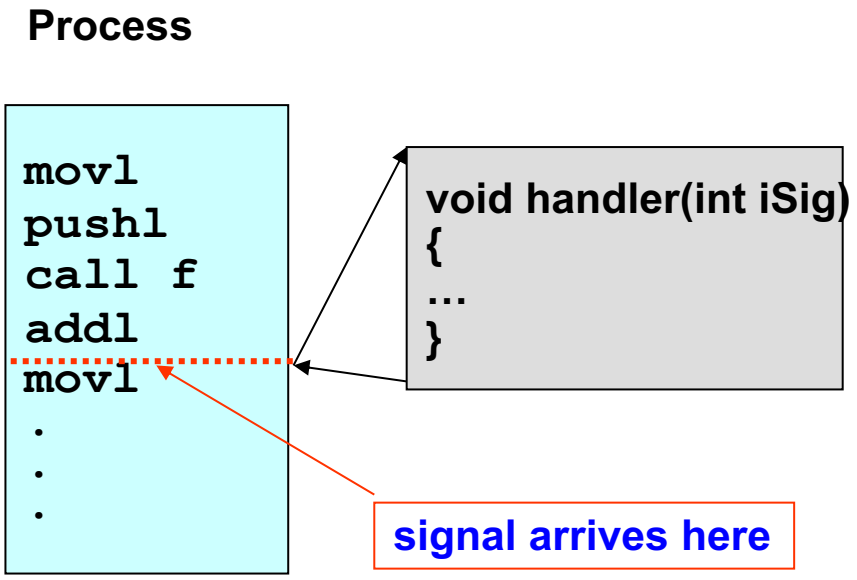
Signals

Signals are a limited form of inter-process communication (IPC), typically used in POSIX-compliant operating systems.

A **signal** is an asynchronous notification sent to a process or to a specific thread within the same process in order to notify it of an event that occurred.

Definition of Signal

- **Signal: A notification of an event**
  - Event gains attention of the OS
  - OS stops the application process immediately, sending it a signal
  - Signal handler executes to completion
  - Application process resumes where it left off



Predefined Signals

List of the predefined signals:

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1
36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4  39) SIGRTMIN+5
40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8  43) SIGRTMIN+9
44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13
52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9
56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6  59) SIGRTMAX-5
60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2  63) SIGRTMAX-1
64) SIGRTMAX
```

Applications can define their own signals

- An application can define signals with unused values

Examples of Signals

- **User types a Ctrl-c**
  - Event gains attention of OS
  - OS stops the application process immediately, sending it a 2/SIGINT signal
  - Signal handler for 2/SIGINT signal executes to completion
    - Default signal handler for 2/SIGINT signal exits process

- **Process makes illegal memory reference**
  - Event gains attention of OS
  - OS stops application process immediately, sending it a 11/SIGSEGV signal
  - Signal handler for 11/SIGSEGV signal executes to completion
    - Default signal handler for 11/SIGSEGV signal prints “segmentation fault” and exits process

Sending Signals via Keystrokes

- Three signals can be sent from keyboard:
  - Ctrl-c → 2/SIGINT signal
    - Default handler exits process
  - Ctrl-z → 20/SIGTSTP signal
    - Default handler suspends process
    - 'fg' to restart process in foreground
    - 'bg' to restart process in background
  - Ctrl-\ → 3/SIGQUIT signal
    - Default handler exits process

Sending Signals via Commands

- **kill Command**
  - kill -signal pid
    - Send a signal of type signal to the process with id pid
    - Can specify either signal type name (-SIGINT) or number (-2)
  - No signal type name or number specified => sends 15/SIGTERM signal
    - Default 15/SIGTERM handler exits process
  - Editorial comment: Better command name would be *sendsig*
- **Examples**
  - kill –2 1234
  - kill -SIGINT 1234
  - Same as pressing Ctrl-c if process 1234 is running in foreground

Sending Signals via Function Call

- **raise()**

```
int raise(int iSig);
```

  - Commands OS to send a signal of type iSig to current process
  - Returns 0 to indicate success, non-0 to indicate failure

- **Example**

```
int ret = raise(SIGINT); /* Process commits suicide. */
assert(ret != 0);        /* Shouldn't get here. */
```

- **Note: C90 function**

https://www.gnu.org/software/libc/manual/html_node/Signaling-Yourself.html

Sending Signals via Function Call

- **kill()**

```
int kill(pid_t iPid, int iSig);
```

  - Sends an `iSig` signal to the process whose id is `iPid`
  - Equivalent to raise(`iSig`) when `iPid` is the id of current process
  - Editorial comment: Better function name would be *`sendsig()`*

- **Example**

```
pid_t iPid = getpid();// Process gets its id
kill(iPid, SIGINT);   // Process sends itself a SIGINT signal
                      // and (commits suicide?)
```

- **Note: POSIX (not C90) function**

Signal Handling

- Each signal type has a default handler
  - Most default handlers exit the process

- A program can install its own handler for any type of signal except *9/SIGKILL* and *19/SIGSTOP*

```
#include<stdio.h>
#include<signal.h>

// Handler for SIGINT, caused by
// Ctrl-C at keyboard

void handle_sigint(int sig)
{
    printf("Caught signal %d\n", sig);
}

int main()
{
    signal(SIGINT, handle_sigint);
    while (1) ;
    return 0;
}
```

Uncatchable Signals

- Special cases: A program cannot install a signal handler for signals of type:
  - 9/SIGKILL
    - Default action is “terminate”
    - Catchable termination signal is 15/SIGTERM
  - 19/SIGSTOP
    - Default action is “stop until next 18/SIGCONT”
    - Catchable suspension signal is 20/SIGTSTP

Installing a Signal Handler

signal()

```
sighandler_t signal(int iSig, sighandler_t pfHandler);
```

Installs function `pfHandler` as the handler for signals of type `iSig`

`pfHandler` is a function pointer:

```
typedef void (*sighandler_t)(int);
```

Returns the old handler on success, `SIG_ERR` on error
After call, `pfHandler` is invoked whenever process receives a signal of type `iSig`

Installing a Handler Example 1

Program testsignal.c:

```
#define _GNU_SOURCE     /* Use modern handling style */
#include <stdio.h>
#include <assert.h>
#include <signal.h>

static void myHandler(int iSig) {
   printf("In myHandler with argument %d\n", iSig);
}
…
```

https://www.cs.princeton.edu/courses/archive/spring20/cos217/precepts/24shell/

Installing a Handler Example 1 (cont.)

Program testsignal.c (cont.):

```
…
int main(void) {
    void (*pfRet)(int);
    pfRet = signal(SIGINT, myHandler);
    assert(pfRet != SIG_ERR);

    printf("Entering an infinite loop\n");
    for (;;)
        ;
    return 0;
}
```

https://www.cs.princeton.edu/courses/archive/spring20/cos217/precepts/24shell/

Installing a Handler Example 2

- **Program testsignalall.c:**

```
#define _GNU_SOURCE
#include <stdio.h>
#include <assert.h>
#include <signal.h>

static void myHandler(int iSig) {
   printf("In myHandler with argument %d\n", iSig);
}
…
```

https://www.cs.princeton.edu/courses/archive/spring20/cos217/precepts/24shell/

Installing a Handler Example 2 (cont.)

- **Program testsignalall.c (cont.):**

```
…
int main(void) {
   void (*pfRet)(int);
   pfRet = signal(SIGHUP,  myHandler);   /* 1 */
   pfRet = signal(SIGINT,  myHandler);   /* 2 */
   pfRet = signal(SIGQUIT, myHandler);   /* 3 */
   pfRet = signal(SIGILL,  myHandler);   /* 4 */
   pfRet = signal(SIGTRAP, myHandler);   /* 5 */
   pfRet = signal(SIGABRT, myHandler);   /* 6 */
   pfRet = signal(SIGBUS,  myHandler);   /* 7 */
   pfRet = signal(SIGFPE,  myHandler);   /* 8 */
   pfRet = signal(SIGKILL, myHandler);   /* 9 */
   …
```

https://www.cs.princeton.edu/courses/archive/spring20/cos217/precepts/24shell/

Installing a Handler Example 2 (cont.)

- **Program testsignalall.c (cont.):**

```
    …
    /* Etc., for every signal. */

    printf("Entering an infinite loop\n");
    for (;;)
        ;
    return 0;
}
```

https://www.cs.princeton.edu/courses/archive/spring20/cos217/precepts/24shell/

Installing a Handler Example 3

- **Program generates lots of temporary data**
  - Stores the data in a temporary file
  - Must delete the file before exiting

```
…
int main(void) {
    FILE *psFile;
    psFile = fopen("temp.txt", "w");
    …
    fclose(psFile);
    remove("temp.txt");
    return 0;
}
```

https://www.cs.princeton.edu/courses/archive/spring20/cos217/precepts/24shell/

Example 3 Problem

- **What if user types Ctrl-c?**
  - OS sends a 2/SIGINT signal to the process
  - Default handler of 2/SIGINT exits the process
- **Problem: The temporary file is not deleted**
  - Process dies before remove("tmp.txt") is executed
- **Challenge: Ctrl-c could happen at any time**
  - Which line of code will be interrupted???
- **Solution: Install a signal handler**
  - Define a “clean up” function to delete the file
  - Install the function as a signal handler for 2/SIGINT

Example 3 Solution

```
…
static FILE *psFile; /* Must be global. */
static void cleanup(int iSig) {
   fclose(psFile);
   remove("tmp.txt");
   exit(EXIT_FAILURE);
}
int main(void) {
   void (*pfRet)(int);
   psFile = fopen("temp.txt", "w");
   pfRet = signal(SIGINT, cleanup);
   …
   raise(SIGINT);
   return 0;  /* Never get here. */
}
```

https://www.cs.princeton.edu/courses/archive/spring20/cos217/precepts/24shell/

Predefined Signal Handler: SIG_IGN

Pre-defined signal handler: SIG_IGN

Can install to ignore signals

```
int main(void) {
    void (*pfRet)(int);
    pfRet = signal(SIGINT, SIG_IGN);
    …
}
```

Subsequently, process will ignore 2/SIGINT signals

Predefined Signal Handler: SIG_DFL

Pre-defined signal handler: SIG_DFL

Can install to restore default signal handler

```
int main(void) {
   void (*pfRet)(int);
   pfRet = signal(SIGINT, somehandler);
   …
   pfRet = signal(SIGINT, SIG_DFL);
   …
}
```

Subsequently, process will handle 2/SIGINT signals using the default handler for 2/SIGINT signals

Exam question: What is the output of the following code

```
#include<stdio.h>
#include<signal.h>
#include<wait.h>
int val = 10;
void myhandler(int sig)
{
    val += 5;
}
int main()
{
    pid_t pid;
    signal(SIGCHLD, myhandler);
    if ((pid = fork()) == 0)
    {
        val -= 3;
        exit(0);
    }
    waitpid(pid, NULL, 0);
    printf("val = %d\n", val);
    exit(0);
}
```

When a child process stops or terminates, **SIGCHLD** is sent to the parent process.

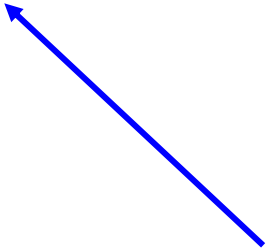
Race Conditions in Signal Handlers

A race condition is a flaw in a program whereby the correctness of the program is critically dependent on the sequence or timing of other events.

Race conditions can occur in signal handlers…

Race Condition Example

```
void addSalaryToSavings(int iSig) {
    int iTemp;
    iTemp = iSavingsBalance;
    iTemp += iMonthlySalary;
    iSavingsBalance = iTemp;
}
```



Handler for hypothetical
“update monthly salary” signal

Race Condition Example (cont.)

(1) Signal arrives; handler begins executing

```
void addSalaryToSavings(int iSig) {
    int iTemp;
    iTemp = iSavingsBalance; 2000
    iTemp += iMonthlySalary;
    iSavingsBalance = iTemp;
}
```

Race Condition Example (cont.)

(2) Another signal arrives; first instance of handler is interrupted; second instance of handler begins executing

```
void addSaliaryToSavings(int iSig) {
    int iTemp;
    iTemp = iSavingsBalance;
    iTemp += iMonthlySalary;
    iSavingsBalance = iTemp;
}
```

2000

```
void addSaliaryToSavings(int iSig) {
    int iTemp;
    iTemp = iSavingsBalance;
    iTemp += iMonthlySalary;
    iSavingsBalance = iTemp;
}
```

2000

Race Condition Example (cont.)

(3) Second instance executes to completion

```
void addSalaryToSavings(int iSig) {
    int iTemp;
    iTemp = iSavingsBalance;
    iTemp += iMonthlySalary;
    iSavingsBalance = iTemp;
}
```

2000

```
void addSalaryToSavings(int iSig) {
    int iTemp;
    iTemp = iSavingsBalance;
    iTemp += iMonthlySalary;
    iSavingsBalance = iTemp;
}
```

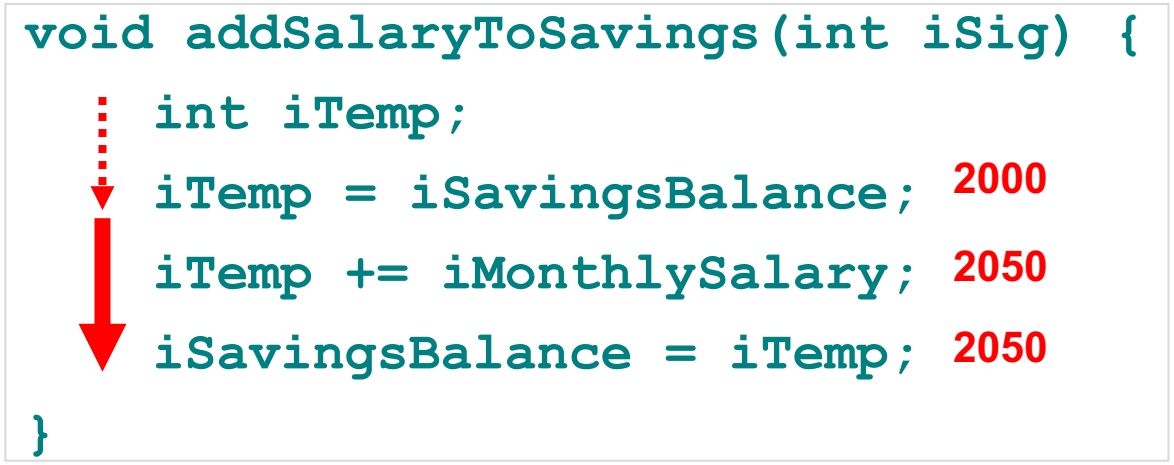
2000

2050

2050

Race Condition Example (cont.)

(4) Control returns to first instance, which executes to completion



Lost $50 !!!

Race Conditions in General

Race conditions can occur elsewhere too

```
int iFlag = 0;

void myHandler(int iSig) {
    iFlag = 1;
}

int main(void) {
    if (iFlag == 0) {
        /* Do something */
    }
}
```

Problem: `iflag` might become `1` just after the comparison!

Must make sure that critical sections of code are not interrupted

Blocking Signals

- **Blocking signals**
  - To block a signal is to queue it for delivery at a later time
  - Differs from ignoring a signal

- **Each process has a signal mask in the kernel**
  - OS uses the mask to decide which signals to deliver
  - User program can modify mask with `sigprocmask`()

Blocking Signals: C90 vs. POSIX

- How to block signals in general?
  - Not possible in C90
  - Possible using POSIX functions…
- C90 standard
  - Defines signal() and raise() functions
    - Work across all systems (UNIX, LINUX, Windows), but…
    - Work differently across some systems!!!
      - On some systems, signals are blocked during execution of handler for that type of signal -- but not so on other (older) systems
      - On some (older) systems, handler installation for signals of type x is cancelled after first signal of type x is received; must reinstall the handler -- but not so on other systems
  - Does not provide mechanism to block signals in general
- POSIX standard
  - Defines kill(), sigprocmask(), and sigaction() functions
    - Work the same across all POSIX-compliant UNIX systems (Linux, Solaris, etc.), but…
    - Does not work on non-UNIX systems (e.g., Windows)
  - Provides mechanism to block signals in general

Blocking Signals in General

sigprocmask()

```
int sigprocmask(int iHow, const sigset_t *psSet,
                sigset_t *psOldSet);
```

psSet: Pointer to a signal set
psOldSet: (Irrelevant for our purposes)
iHow: How to modify the signal mask
SIG_BLOCK: Add psSet to the current mask
SIG_UNBLOCK: Remove psSet from the current mask
SIG_SETMASK: Install psSet as the signal mask
Returns 0 iff successful

- **Functions for constructing signal sets**
  - `sigemptyset()` initializes the signal set given by *set* to empty, with all signals excluded from the set.
  - `sigfillset()` initializes *set* to full, including all signals.
  - `sigaddset()` and `sigdelset()` add and delete respectively signal `signum` from *set*.

Blocking Signals Example

```
sigset_t sSet;
int main(void) {
    int iRet;
    sigemptyset(&sSet);
    sigaddset(&sSet, SIGINT);
    iRet = sigprocmask(SIG_BLOCK, &sSet, NULL);
    assert(iRet == 0);
    if (iFlag == 0) {
        /* Do something */
    }
    iRet = sigprocmask(SIG_UNBLOCK, &sSet, NULL);
    assert(iRet == 0);
    …
}
```

Blocking Signals in Handlers

- **How to block signals when handler is executing?**
  - **While executing a handler for a signal of type x, all signals of type x are blocked automatically**
  - **Previous "update monthly salary" race condition cannot happen!!!**
  - **When/if signal handler returns, block is removed**

- **Additional signal types to be blocked can be defined at time of handler installation…**

Installing a Signal Handler

sigaction()

```
int sigaction(int iSig,
              const struct sigaction *psAction,
              struct sigaction *psOldAction);
```

iSig: The type of signal to be affected
psAction: Pointer to a structure containing instructions on how to handle signals of type iSig, including signal handler name and which signal types should be blocked
psOldAction: (Irrelevant for our purposes (NULL))

Installs an appropriate handler
Automatically blocks signals of type iSig
Returns 0 iff successful

```
struct  sigaction action;        // Structure for signaling
  sigemptyset(&action.sa_mask);
  sigaddset(&action.sa_mask, SIGIO);
  action.sa_handler = sighandler;
  action.sa_flags = 0;
  sigaction(SIGIO, &action, NULL);
```

Installing a Handler Example

Program testsigaction.c:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

static void myHandler(int iSig) {
   printf("In myHandler with argument %d\n", iSig);
}
…
```

Installing a Handler Example (cont.)

Program testsigaction.c (cont.):

```
…
int main(void) {
    int iRet;

    struct sigaction sAction;
    sAction.sa_flags = 0;
    sAction.sa_handler = myHandler;
    sigemptyset(&sAction.sa_mask);
    iRet = sigaction(SIGINT, &sAction, NULL);
    assert(iRet == 0);

    printf("Entering an infinite loop\n");
    for (;;)
        ;
    return 0;
}
```

sigsuspend

- `sigsuspend` function sets signal mask and suspends process until a signal is caught by the process

- `sigsuspend` returns when signal handler of the caught signal returns

- `sigsuspend` installs the given signal mask as temporary process mask, and then blocks the calling process until an unblocked signal arrives. That signal is handled in the normal way, by calling the signal handler; then, the original signal mask is restored

```
#include <signal.h>
int sigsupend(const sigset_t *sigmask);
```

https://linux.die.net/man/3/sigsuspend

Example: sigsuspend

- What's wrong?

```
/* assume signum signal was not masked */
…
sigfillset(&sigmost);
sigdelset(&sigmost, signum);
sigsuspend(&sigmost);
…
```

- `signum` is the only signal unblocked that can cause `sigsuspend` to return
- If the signal `signum` is delivered before the start of the code segment, the process still suspends itself and deadlocks if another `signum` signal is not generated

Example: sigsuspend (A Correct Way to Wait for a Single Signal)

```
static volatile sig_atomic_t sigreceived = 0;
/*assume signal handler has been setup for signum and it
  sets sigreceived = 1 */
sigset_t maskall, maskmost, maskold;
int signum = SIGUSR1;

sigfillset(&maskall);
sigfillset(&maskmost);
sigdelset(&maskmost, signum);
sigprocmask(SIG_SETMASK, &maskall, &maskold);
…
if (sigreceived == 0) {
 rc = sigsuspend(&maskmost);
 assert(rc == -1 && errno == EINTR && sigio_signal_processed); }
sigprocmask(SIG_SETMASK, &maskold, NULL);
```

- Quiz: can this code deadlock because `SIGUSR1` was delivered between "if statement" and the call to `sigsuspend`???

  No, because SIGUSR1 is masked before testing sigreceived!

Example: sigsuspend (A Correct Way to Wait for a Single Signal)

```
static volatile sig_atomic_t sigreceived = 0;
/*assume signal handler has been setup for signum and it
  sets sigreceived = 1 */
sigset_t maskall, maskmost, maskold;
int signum = SIGUSR1;

sigfillset(&maskall);
sigfillset(&maskmost);
sigdelset(&maskmost, signum);
sigprocmask(SIG_SETMASK, &maskall, &maskold);
…
if (sigreceived == 0) {
 rc = sigsuspend(&maskmost);

/* Confirm we are coming out of suspend mode correcly */
assert(rc == -1 && errno == EINTR && sigio_signal_processed); }
sigprocmask(SIG_SETMASK, &maskold, NULL);
```

"volatile" has been used to notify the compiler that `sigreceived` might change at any time due to the asynchronous execution of the signal handler

What type is `sig_atomic_t`?

type sig_atomic_t

- To avoid uncertainty about interrupting access to a variable, you can use a particular data type for which access is guaranteed to be atomic: `sig_atomic_t`
- Reading and writing this data type is guaranteed to happen in a single instruction, so there's no way for a handler to run "in the middle" of an access.
- The type `sig_atomic_t` is always an integer data type, but which one it is, and how many bits it contains, may vary from machine to machine.
- In practice, you can assume that `int` type is atomic. You can also assume that pointer types are atomic; that is very convenient. Both of these should be true on all of the machines that the GNU C library supports.

From: Nahrstedt, Angrave, Abdelzaher

Pitfalls with signals

- Assume to have a shared data structure between the signal handler and the regular execution flow; can I use a mutex to protect the critical section?
- Signal handlers do not multi-task: there is just one single process going off and handling various signals. If there is a shared data structure between the signal handler and the regular execution flow, do not use a mutex, otherwise when the signal handler blocks, the entire process blocks.
- To avoid race conditions, you can use one of the following techniques:
  - if the process shares a data structure with a signal handler, mask the signal before modifying the data structure and then unmask the signal again;
  - keep signals masked and catch them synchronously by using sigsuspend(), but this could hang the user process until an unblocked signal arrives;

From: Nahrstedt, Angrave, Abdelzaher

sigwait

```
#include <signal.h>
int sigwait(const sigset_t *sigmask, int *signo);
```

- sigwait suspends the calling thread until one of the signals in sigmask is delivered to the calling thread. It then stores the number of the signal received in the location pointed to by signo and returns.

- The signals in sigmask must be blocked and not ignored on entrance to sigwait. If the delivered signal has a signal handler function attached, that function is not called.

- This function avoids the problem of asynchronous execution of signal handlers and the risk of race conditions!!!!

From: Nahrstedt, Angrave, Abdelzaher

How to use sigwait

```
#include <signal.h>
int sigwait(const sigset_t *sigmask, int *signo);
```

- First block all signals
- Put the signals you want to wait for in `sigmask`
- Call `sigwait`
- `sigwait` blocks the process until at least one of these signals is pending.
- It removes one of the pending signals and gives you the corresponding signal number in the second parameter.
- Do what you want: no signal handler is needed.
- It returns 0 on success and -1 on error with errno set.

From: Nahrstedt, Angrave, Abdelzaher

pthread_kill

- The `pthread_kill()` function provides a mechanism for asynchronously directing a signal at a thread in the calling process. Note that `pthread_kill()` only causes the signal to be handled in the context of the given thread.
- The signal action affects the process as a whole: sending a `SIGKILL` signal to a specific thread using `pthread_kill()` would kill the process, not just the specified thread.
- The signal sent by `pthread_kill()` is handled like any other signal: if target thread has signal masked, it will be marked pending against that thread

You cannot send a `pthread_kill` to a thread of another process!!!

From: Nahrstedt, Angrave, Abdelzaher

Problems with POSIX signals

- POSIX signals provide one basic mechanism for asynchronous communication between processes.

- There are some serious shortcomings when you want to use the POSIX signals:
  - Lack of signals for application use
  - Lack of signal queueing
  - No signal delivery order
  - Poor information content

- To solve many drawbacks of POSIX signals, POSIX.4 real-time signals were introduced as an extension (check them out if you are curious!)

From: Nahrstedt, Angrave, Abdelzaher

Summary of Signals

- **Signals**
  - A signal is an asynchronous event mechanism
  - C90 raise() or POSIX kill() sends a signal
  - C90 signal() or POSIX sigaction() installs a signal handler
    - Most predefined signals are “catchable”
  - Beware of race conditions
  - POSIX `sigprocmask()` blocks signals in any critical section of code
    - Signals of type x automatically are blocked while handler for type x signals is running
  - POSIX `sigaction()` installs a signal handler, and allows blocking of additional signal types during handler execution

Backup

ARM Interrupts

Review of ARM Exceptions

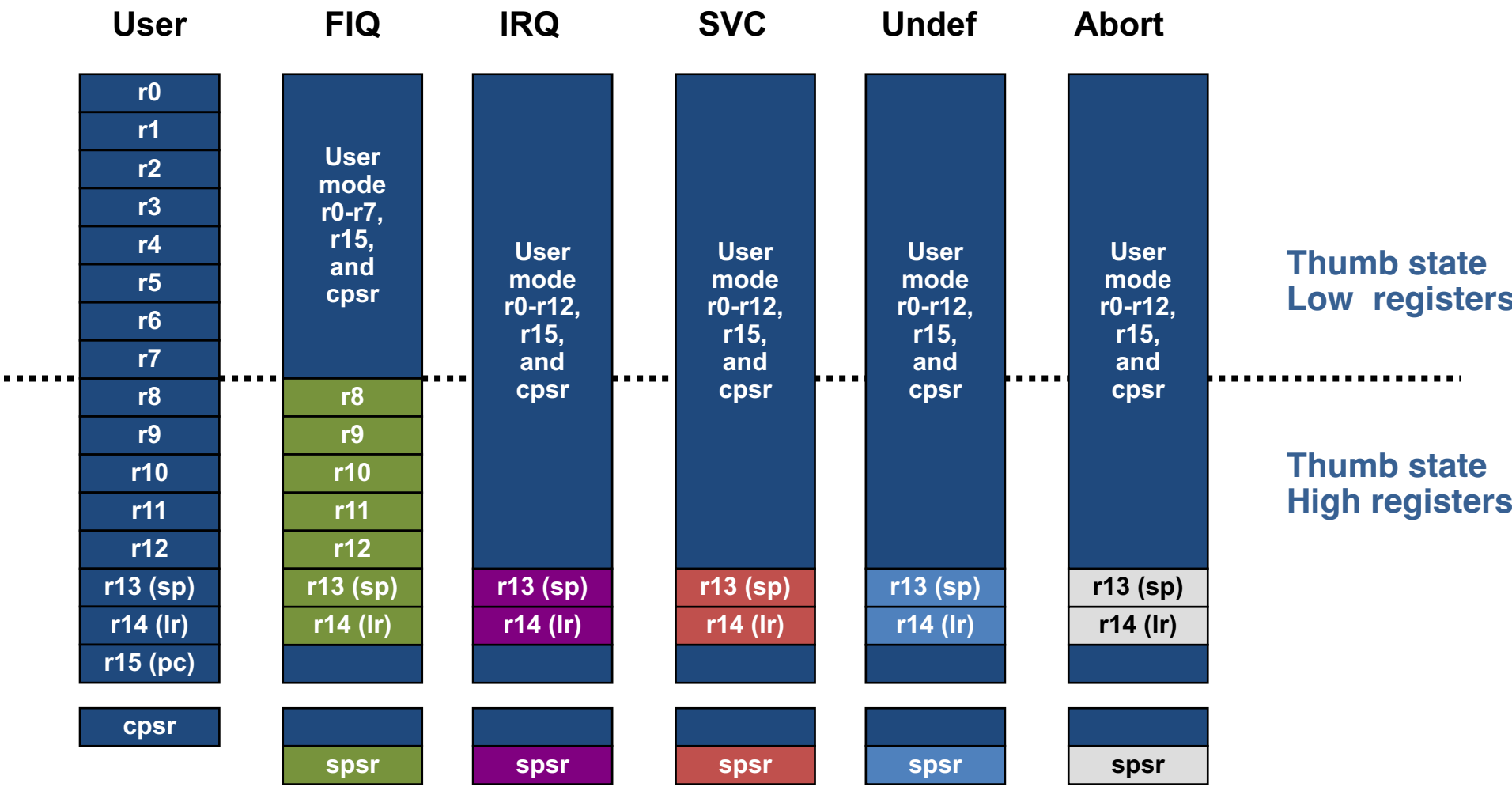
Exception	Description
Reset	Occurs when the processor reset pin is asserted
Undefined Inst.	Occurs if processor does not recognize the currently executing instruction
Software Interrupt	This is a user defined synchronous interrupt instruction. It allows a program running in User Mode to request a privileged operation that runs in Supervisor mode.
Prefetch Abort	Occurs when the processor fetches an instruction from an illegal address e.g. unaligned.
Data Abort	Occurs when the processor fetches data from an illegal address e.g. unaligned.
IRQ	Occurs when the processor external interrupt request pin is asserted (LOW) and the I bit in the CPSR is clear
FIQ	Occurs when the processor external fast interrupt request pin is asserted (LOW) and the F bit in the CPSR is clear.

Review of ARM Exceptions (cont)

- **Vector table**
  - Reserved area of 32 bytes at the bottom of the memory map
  - One 32 bit word for each exception type
  - Contains a Branch or Load PC instruction to the exception handler

- **Exception modes and registers**
  - Handling exceptions changes program from user to non-user mode
  - Each exception handler has access to its own set of registers
    - Its own r13 = stack pointer
    - Its own r14 = link register
    - Its own SPSR (Saved Program Status Register)
  - Exception handlers must save (restore) other register on entry (exit)

Register Organization Review



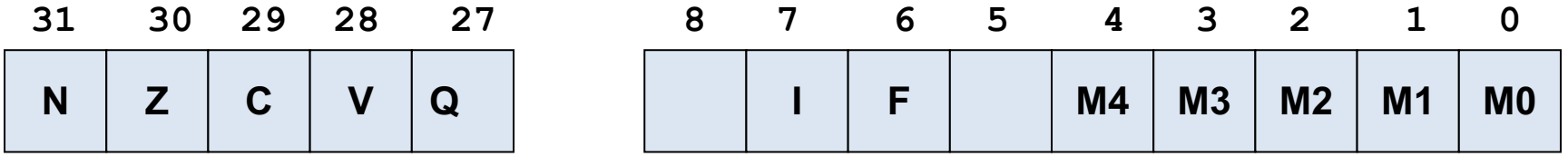
Note: System mode uses the User mode register set

Cortex-A9 Exception Priorities

Vector Address	Exception Type	Exception Mode	
0x0000_0000	Reset	Supervisor (SVC)	1
0x0000_0004	Undefined Inst.	UNDEF	6
0x0000_0008	SWI	Supervisor (SVC)	6
0x0000_000c	Prefetch Abort	Abort	5
0x0000_0010	Data Abort	Abort	2
0x0000_0014	*Reserved*	*N/A*	*N/A*
0x0000_0018	IRQ	IRQ	4
0x0000_001c	FIQ	FIQ	3

Enabling IRQ and FIQ

- **Program Status Register**



- **To disable interrupts, set corresponding "F" or "I" bit to 1**
- **On interrupt, processor does the following**
  - Switches register banks
  - Copies CPSR to SPSR_mode (saves mode, interrupt flags, etc.)
  - Changes the CPSR mode bits (M[4:0])
  - Disables interrupts
  - Copies PC to R14_mode (to provide return address)
  - Sets the PC to the vector address of the exception handler
- **Interrupt handlers must contain code to clear the source of the interrupt**

Interrupt Details

- On an IRQ interrupt, the ARM processor will ...
  - If the "I" bit in the CPSR is clear, the current instruction is completed and then the processor will
    - Save the address of the next instruction plus 4 in r14_irq
    - Save the CPSR in the SPSR_irq
    - Force the CPSR mode bits M[4:0] to 10010 (binary)

- This switches the CPU to IRQ mode and then sets the "I" flag to disable further IRQ interrupts

- On an FIQ interrupt, the processor will ...
  - If the "F" bit in the CPSR is clear and the current instruction is completed, the ARM will
    - Save the address of the next instruction plus 4 in r14_fiq
    - Force the CPSR mode bits M[4:0] to 10001 (binary)
      - This switches the CPU to FIQ mode and then sets the "I" and "F" flags to disable further IRQ or FIQ interrupts

IRQ vs. FIQ

- **FIQs have higher priority than IRQs**
  - When multiple interrupts occur, FIQs get serviced before IRQs
  - Servicing an FIQ causes IRQs to be disabled until the FIQ handler re-enables them
    - CPSR restored from the SPSR at the end of the FIQ handler

- **How are FIQs made faster?**
  - They have five extra registers at their disposal, allowing them to store status between calls to the handler
  - FIQ vector is the last entry in the vector table
    - The FIQ handler can be placed directly at the vector location and run sequentially after the location
    - Cache-based systems: Vector table + FIQ handler all locked down into one block

Types of Interrupts

- **Synchronous**
  - Produced by the processor while executing instructions.
  - Issues only after finishing execution of an instruction.
  - Often called exceptions.
  - Example: SWI, page faults, system calls, divide by zero

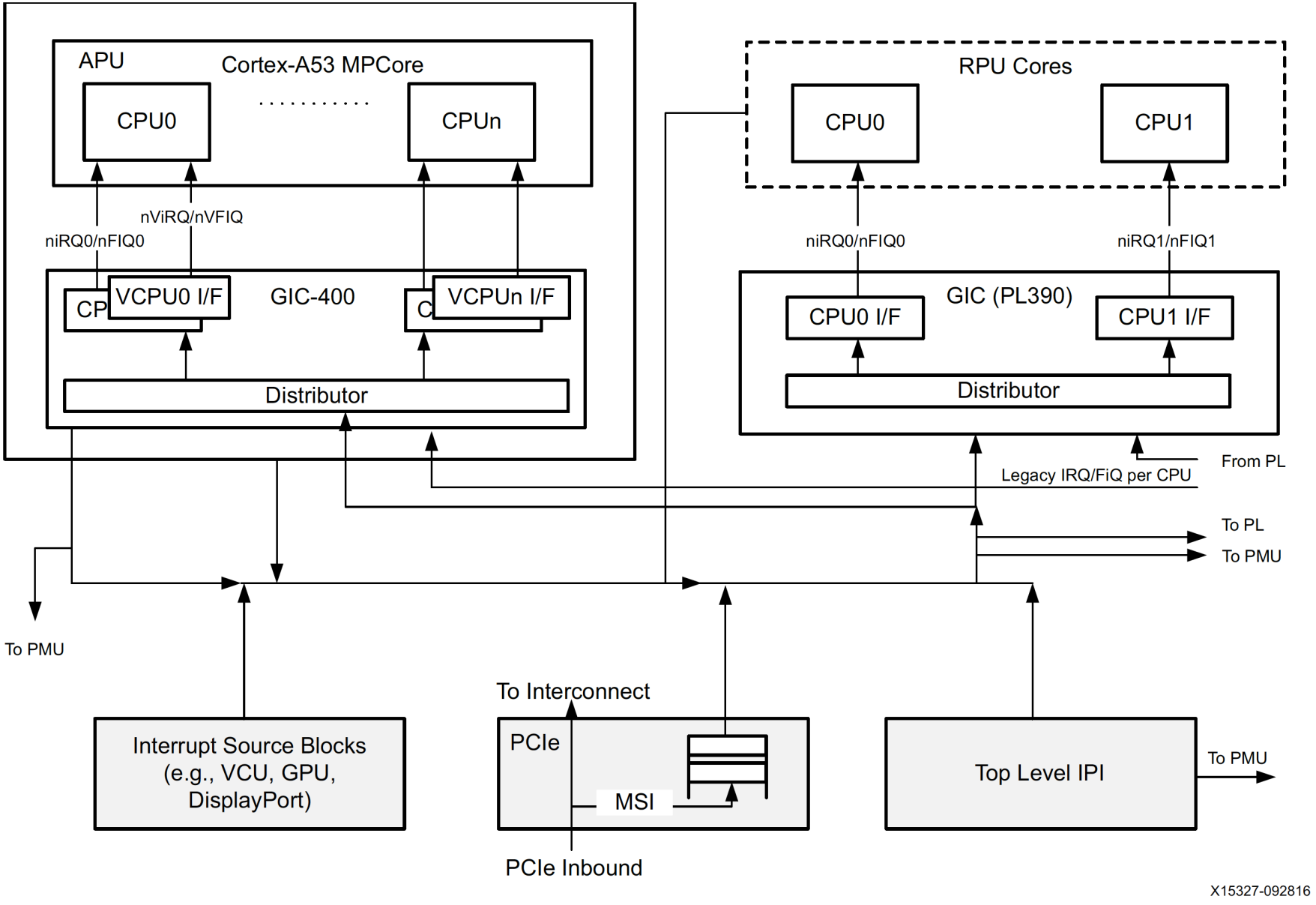
- **Asynchronous**
  - Generated by other hardware devices.
  - Occur at arbitrary times, including while CPU is busy executing an instruction.
  - Ex: I/O, timer interrupts

Zynq-UltraScale+ Interrupts

Zynq-UltraScale+ ARM Generic Interrupt Controller (GIC)

- **The GIC performs the following functions:**
  - Supports up to 1024 interrupt sources
  - Supports fast and normal interrupts
  - Selects normal or fast interrupt request from any interrupt source
  - Indicates pending interrupt sources via a register for normal and fast interrupts
  - Indicates highest priority interrupt number via register (can be used as a table index)
  - Independently enable or disable any interrupt source
  - Provides a mechanism for software to schedule an interrupt
  - Supports up to 16 software-controlled priority levels for normal interrupts and priority masking

Zynq-UltraScale+ Interrupt Flow Diagram (Dual Core)



Programmable Logic –> GIC Interrupts

- The PL can asynchronously assert up to 20 interrupts to the PS.
  - 16 interrupt signals are mapped to the interrupt controller as a peripheral interrupt where each interrupt signal is set to a priority level and mapped to one or both CPUs.
  - The remaining four PL interrupt signals are inverted and routed to the nFIQ and nIRQ nterrupt directly to the signals to the private peripheral interrupt (PPI) unit of the interrupt controller. There is an nFIQ and nIRQ interrupt for each of two CPUs.

PL-PS Interrupts (Interrupts that go from PL to PS)				
1	121-128, 136-143 (See Table 2-1 for Pin mapping to each interrupt ID)	IRQ-F2P[15:0]	Shared Interrupts from PL logic to GICs of real-time processing unit (RPU) or application processing unit (APU)	Shared Interrupts
2	31	A53-Core_0 nIRQ	Cortex™ A53 Core0 Private Peripheral Legacy IRQ Interrupt	Private Peripheral Interrupt
3	31	A53-Core_1 nIRQ	Cortex A53 Core1 Private Peripheral Legacy IRQ Interrupt	Private Peripheral Interrupt
4	31	A53-Core_2 nIRQ	Cortex A53 Core2 Private Peripheral Legacy IRQ Interrupt	Private Peripheral Interrupt
5	31	A53-Core_3 nIRQ	Cortex A53 Core3 Private Peripheral Legacy IRQ Interrupt	Private Peripheral Interrupt
6	28	A53-Core_0 nFIQ	Cortex A53 Core0 Private Peripheral Legacy FIQ Interrupt	Private Peripheral Interrupt
7	28	A53-Core_1 nFIQ	Cortex A53 Core1 Private Peripheral Legacy FIQ Interrupt	Private Peripheral Interrupt
8	28	A53-Core_2 nFIQ	Cortex A53 Core2 Private Peripheral Legacy FIQ Interrupt	Private Peripheral Interrupt
9	28	A53-Core_3 nFIQ	Cortex A53 Core3 Private Peripheral Legacy FIQ Interrupt	Private Peripheral Interrupt

Zynq-UltraScale+ Interrupt Sources

Software Generated Interrupts:

IRQ ID#	Name	SGI#	Type	Description
0	Software 0	0	Rising edge	A set of 16 interrupt sources that are private to each CPU that can be routed to up to 16 common interrupt destinations where each destination can be one or more CPUs.
1	Software 1	1	Rising edge	
~	...	~	...	
15	Software 15	15	Rising edge	

Private Peripheral Interrupts:

Name	Interrupt ID	Description
Virtual maintenance interrupt	25	Configurable event generated by virtual CPU interface to indicate a situation that might require hypervisor action.
Hypervisor timer	26	Physical timer event in hypervisor mode, PPI5 (CNTHP IRQ).
Virtual timer	27	Virtual timer generated event, PPI4 (CNTV IRQ).
Legacy FIQ signal	28	FIQ signal from the PL.
Secure physical timer	29	Secure physical timer event, PPI1 (CNTPS IRQ).
Non-secure physical timer	30	Non-secure physical timer event, PPI2 (CNTPNS IRQ).
Legacy IRQ signal	31	IRQ signal from the PL.

Zynq-US+ Shared Peripheral Interrupt Sources

- **Shared Peripheral Interrupts:**

IRQ Name	IRQ Number (GIC)	IDCICR	Bits	Required Type	GICPx_IRQ Bits (GIC Proxy)	Description
RPU0_Perf_Mon	40	2	[17:16]	High level	GICP0 [8]	RPU0 performance monitor (ARM_PMU)(1).
RPU1_Perf_Mon	41	2	[19:18]	High level	GICP0 [9]	RPU1 performance monitor (ARM_PMU)(1).
OCM	42	2	[21:20]	High level	GICP0 [10]	OCM CE and UE ECC errors.
LPD_APB	43	2	[23:22]	High level	GICP0 [11]	OR of all APB slave interface errors in LPD.
RPU0_ECC	44	2	[25:24]	High level	GICP0 [12]	RPU0 errors combined: FPU, memory ECC, and APB access.
RPU1_ECC	45	2	[27:26]	High level	GICP0 [13]	RPU1 CE errors combined: FPU, memory, ECC, and APB access.
NAND	46	2	[29:28]	High level	GICP0 [14]	NAND memory controller.
QSPI	47	2	[31:30]	High level	GICP0 [15]	Quad-SPI controller.
GPIO	48	3	[1:0]	High level	GICP0 [16]	GPIO controller.
I2C0	49	3	[3:2]	High level	GICP0 [17]	I2C0 controller.

Zynq-US+ Shared Peripheral Interrupt Sources

IRQ Name	IRQ Number (GIC)	IDCICR	Bits	Required Type	GICPx_IRQ Bits (GIC Proxy)	Description
I2C1	50	3	[5:4]	High level	GICP0 [18]	I2C1 controller.
SPI0	51	3	[7:6]	High level	GICP0 [19]	SPI0 controller.
SPI1	52	3	[9:8]	High level	GICP0 [20]	SPI1 controller.
UART0	53	3	[11:10]	High level	GICP0 [21]	UART 0 controller.
UART1	54	3	[13:12]	High level	GICP0 [22]	UART 1 controller.
CAN0	55	3	[15:14]	High level	GICP0 [23]	CAN 0 controller.
CAN1	56	3	[17:16]	High level	GICP0 [24]	CAN 1 controller.
LPD_APM	57	3	[19:18]	High level	GICP0 [25]	OR of the LPD and OCM APM interrupts.
RTC_Alarm	58	3	[21:20]	High level	GICP0 [26]	RTC alarm interrupt.
RTC_Seconds	59	3	[23:22]	High level	GICP0 [27]	RTC seconds interrupt.
ClkMon	60	3	[25:24]	High level	GICP0 [28]	Clock monitor in LPD.
IPI_Ch7	61	3	[27:26]	High level	GICP0 [29]	IPIs targeting channel 7.
IPI_Ch8	62	3	[29:28]	High level	GICP0 [30]	IPIs targeting channel 8.
IPI_Ch9	63	3	[31:30]	High level	GICP0 [31]	IPIs targeting channel 9.

Zynq-US+ Shared Peripheral Interrupt Sources

IPI_Ch10	64	4	[1:0]	High level	GICP1 [0]	IPIs targeting channel 10.
IPI_Ch1	65	4	[3:2]	High level	GICP1 [1]	IPIs targeting channel 1.
IPI_Ch2	66	4	[5:4]	High level	GICP1 [2]	IPIs targeting channel 2.
IPI_Ch0	67	4	[7:6]	High level	GICP1 [3]	IPIs targeting channel 0.
TTC0	68:70	4	[13:8]	High level	GICP1 [4:6]	Triple-timer counter 0.
TTC1	71:73	4	[19:14]	High level	GICP1 [7:9]	Triple-timer counter 1.
TTC2	74:76	4	[25:20]	High level	GICP1 [10:12]	Triple-timer counter 2.
TTC3	77:79	4	[31:26]	High level	GICP1 [13:15]	Triple-timer counter 3.
SDIO0	80	5	[1:0]	High level	GICP1 [16]	SDIO 0 controller.
SDIO1	81	5	[3:2]	High level	GICP1 [17]	SDIO 1 controller.
SDIO0_Wakeup	82	5	[5:4]	High level	GICP1 [18]	SDIO 0 wake-up interrupt.
SDIO1_Wakeup	83	5	[7:6]	High level	GICP1 [19]	SDIO 1 wake-up interrupt.

Refer to UG1085 Chapter 13 for more details:

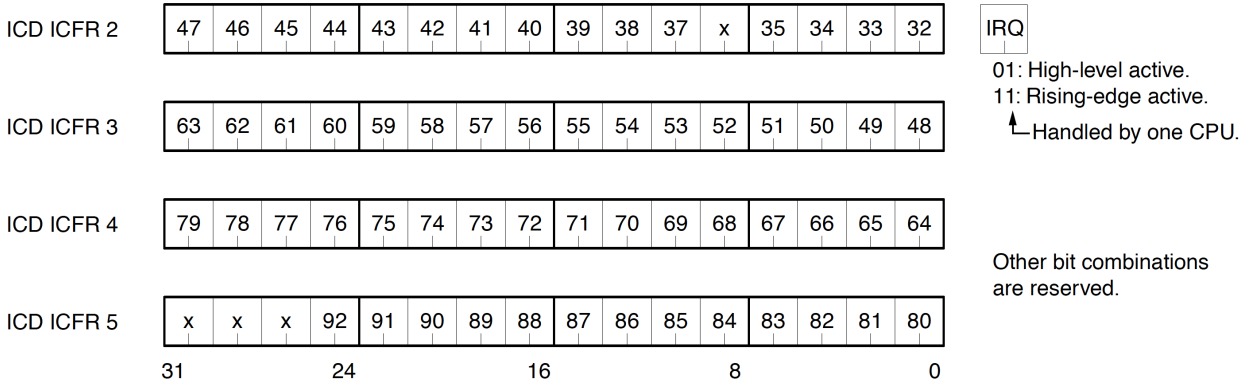
https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf

Interrupt Sensitivity, Targeting and Handling

- There are three types of interrupts that come into the GIC: SPI, PPI and SGI.
- The interrupt signals includes a sensitivity setting, whether one or both CPUs handle the interrupt, and which CPU or CPUs are targeted: zero, one, or both.
- The functionality of most interrupt signals include fixed settings, while others are partially programmable.

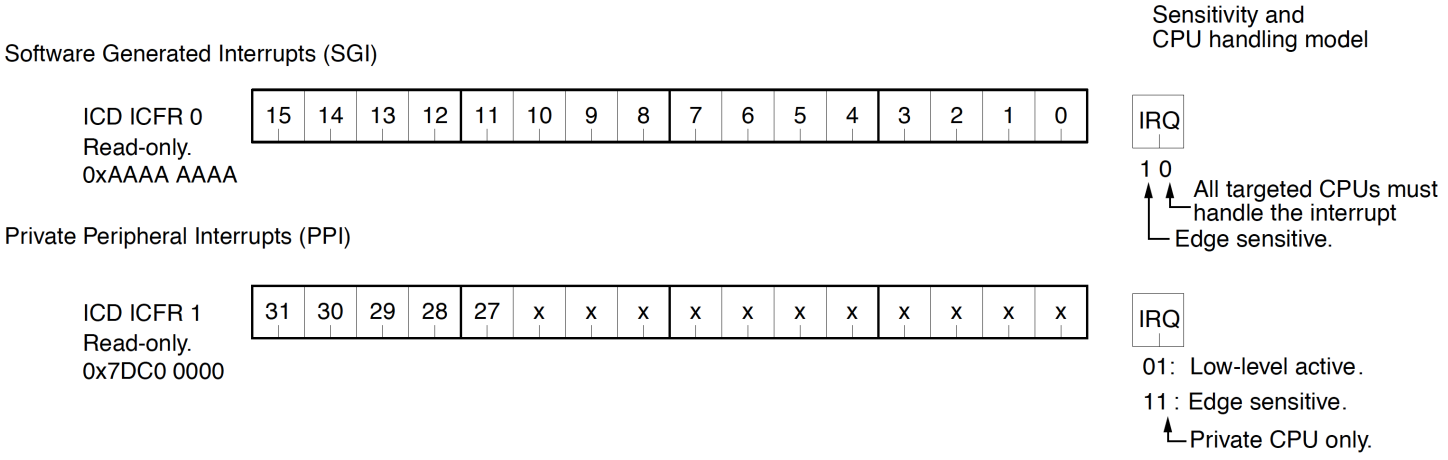
Shared Peripheral Interrupts (SPI)

- The SPI interrupts can be targeted to any number of CPUs, but only one CPU handles the interrupt.
- If an interrupt is targeted to both CPUs and they respond to the GIC at the same time, the MPcore ensures that only one of the CPUs reads the active interrupt ID#.
  - The other CPU receives the Spurious ID# 1023 interrupt or the next pending interrupt, depending on the timing.
- The sensitivity of each SPI interrupt must be programmed to match those listed in the table below



PPI & SGI Interrupts

- **Private Peripheral Interrupts (PPI)**
  - Each CPU has its own separate PPI interrupts with fixed functionality; the sensitivity, handling, and targeting of these interrupts are not programmable.
  - Each interrupt only goes to its own CPU and is handled by that CPU.
- **Software Generated Interrupts (SGI)**
  - The SGI interrupts are always edge sensitive and are generated when software writes the interrupt number to ICDSGIR register. All of the targeted CPUs defined in the ICDIPTR [23:8] must handle the interrupt in order to clear it.



GPIO Interrupts on the Zynq-UltraScale+

- Every general purpose input can be configured as an interrupt and each interrupt can be defined as either:
  - rising-edge triggered
  - falling-edge triggered
  - level sensitive
- The interrupts can be masked using a 32-bit mask register.
- Two levels of interrupt masking are provided. Interrupts can be individually masked at the bit level or at the port level.
- The interrupt status register bits corresponding to the interrupts waiting for service are stored as a value of 1. The interrupt status register is Write 1 to Clear (w1c).

Assigning Interrupt Numbers on the Zynq-UltraScale+

	CPU0	CPU1	CPU2	CPU3			
3:	1229843	1388132	1325883	1314033	GICv2 30 Level	arch_timer	PPI
6:	0	0	0	0	GICv2 67 Level	ff9905c0.mailbox	
7:	0	0	0	0	GICv2 175 Level	arm-pmu	
8:	0	0	0	0	GICv2 176 Level	arm-pmu	
9:	0	0	0	0	GICv2 177 Level	arm-pmu	
10:	0	0	0	0	GICv2 178 Level	arm-pmu	
12:	0	0	0	0	GICv2 156 Level	zynqmp-dma	
18:	0	0	0	0	GICv2 162 Level	zynqmp-dma	
19:	0	0	0	0	GICv2 163 Level	zynqmp-dma	
20:	0	0	0	0	GICv2 164 Level	Mali_GP	
21:	0	0	0	0	GICv2 109 Level	zynqmp-dma	
22:	0	0	0	0	GICv2 110 Level	zynqmp-dma	
23:	0	0	0	0	GICv2 111 Level	zynqmp-dma	
27:	0	0	0	0	GICv2 115 Level	zynqmp-dma	
28:	0	0	0	0	GICv2 116 Level	zynqmp-dma	SPI
31:	5	0	0	0	GICv2 50 Level	cdns-i2c	
32:	0	0	0	0	GICv2 42 Level	ff960000.memory-controller	
33:	0	0	0	0	GICv2 57 Level	axi-pmon	
34:	0	0	0	0	GICv2 58 Level	ffa60000.rtc	
35:	0	0	0	0	GICv2 59 Level	ffa60000.rtc	
36:	30234	0	0	0	GICv2 80 Level	mmc0	
37:	93	0	0	0	GICv2 81 Level	mmc1	
38:	0	0	0	0	GICv2 51 Level	ff040000.spi	
39:	0	0	0	0	GICv2 52 Level	ff050000.spi	
41:	2045	0	0	0	GICv2 54 Level	xuartps	
43:	0	0	0	0	GICv2 84 Edge	ff150000.watchdog	
44:	0	0	0	0	GICv2 88 Level	ams-irq	
45:	0	0	0	0	GICv2 154 Level	fd4c0000.dma	
46:	0	0	0	0	GICv2 151 Level	fd4a0000.zynqmp-display	
47:	0	0	0	0	GICv2 61 Level	ff9a0100.zynqmp_r5_rproc	
48:	195639382	0	0	0	GICv2 126 Edge	gpio_interrupt	
53:	0	0	0	0	GICv2 125 Edge	cdma-controller	
73:	1398110	0	0	0	GICv2 102 Level	xhci-hcd:usb1	
77:	0	0	0	0	gpio 23 Edge	sw4	
Linux Interrupt Number	Number of Interrupts per Processor				GIC Interrupt Number	Peripheral Component	

More Backup

Top and Bottom Halves

- Interrupt handling sometimes needs to perform lengthy tasks.
- This problem is resolved by splitting the interrupt handler into two halves:
  - Top half responds to the interrupt
    - The one registered to request_irq()
    - Saves data to device-specific buffer and schedules the bottom half
    - Current interrupt disabled, possibly all disabled.
    - Runs in interrupt context, not process context. Can't sleep.
    - Acknowledges receipt of interrupt.
    - Schedules bottom half to run later.
  - Bottom half is scheduled by the top half to execute later
    - With all interrupts enabled
    - Wakes up processes, starts I/O operations, etc.
    - Runs in process context with interrupts enabled.
    - Performs most work required. Can sleep.
    - Ex: copies network data to memory buffers.

Top and Bottom Halves

- Three mechanisms may be used to implement bottom halves
  - SoftIRQs
    - Have strong locking requirements
    - Only used for performance sensitive subsystems – networking, SCSI, etc.
    - Reentrant
  - Tasklets
    - Built on top of SoftIRQs
    - Should not sleep
    - Cannot run in parallel with itself
    - Can run in parallel with other tasklets on SMP systems
    - Guaranteed to run on the same CPU that first scheduled them
  - Workqueues
    - Can sleep
    - Cannot copy data to and from user space