

Admin

Momentum through end-game
Reach out if you need a pep talk

Interrupts (resumed)

Last time

Exceptional control flow
(low-level mechanisms)

Today

Using interrupts as client
Configure, enable, attach handler

Coordination of activity

Exceptional and non-exceptional code, multiple handlers
Data sharing, writing code that can be safely interrupted



Looking ahead

Last regular week: Lab 7, Assign 7

Put on finishing touches

Nab that complete system bonus!

Labs 8 & 9 will be group work on final project

Begin brainstorming final project ideas and forming teams (2 people is best)

Interrupted control flow

```
static volatile int gCount;
```

```
void update_screen(void)
{
    console_clear();
    for (int i = 0; i < N; i++)
        console_printf("%d", gCount);
}
```

```
bool button_pressed(unsigned int pc)
{
    if (gpio_check_and_clear_event(BUTTON)) {
        gCount++;
        return true;
    }
    return false;
}
```

23 23 23 23 23
23 23 24 24 24

Questions about how to suspend, process interrupt, resume?

Interrupt checklist

Client must:

- ✓ Initialize interrupt module
- ✓ Enable detection of desired kind of event
 - E.g., detect falling clock edge on GPIO_PIN3 (PS/2 CLK)
- ✓ Write handler function to process event
- ✓ Attach handler to interrupt source (enables source)
 - E.g., GPIO interrupts
- ✓ Globally enable interrupts
 - Throw the big switch to turn it all on when ready

All steps essential

Fiddly code, easy to forget or mix up steps,

Bug symptom is often no interrupt at all, revisit checklist to find what's off

Attach handler

Every interrupt starts with same actions:

- Executes instruction at vectors[IRQ] which jumps to `interrupt_asm` which calls C function `interrupt_dispatch`
- But how to handle key event versus button event versus timer event...?

`interrupt_dispatch` operates as *dispatcher*:

- Client passes a function pointer to "attach" as handler for source
- Client's function added to array of handlers
- On interrupt, `interrupt_dispatch` calls each handler in turn
- A handler responsible for determining if this is "its" interrupt
 - If so, process it, clear state, return true
 - Otherwise do nothing, return false
- Processing stops at first handler that reports interrupt has been handled
- Review our code in `interrupts.c`

Detecting GPIO events

Register pin for event detection

- Different options: falling edge, rising edge, high level, etc.

Read/clear status in event detect register

- Bit is set when an event on the given pin occurs
- Must clear event bit or will re-trigger interrupt!

References

- P. 96-99 in BCM2835 ARM Peripherals doc
- Review our code in `gpioextra.c`

A most frustrating page...



BCM2835 ARM Peripherals

ARM peripherals interrupts table.

#	IRQ 0-15	#	IRQ 16-31	#	IRQ 32-47	#	IRQ 48-63
0		16		32		48	smi
1		17		33		49	gpio_int[0]
2		18		34		50	gpio_int[1]
3		19		35		51	gpio_int[2]
4		20		36		52	gpio_int[3]
5		21		37		53	i2c_int
6		22		38		54	spi_int
7		23		39		55	pcm_int
8		24		40		56	
9		25		41		57	uart_int
10		26		42		58	
11		27		43	i2c_spi_slv_int	59	
12		28		44		60	
13		29	Aux int	45	pwa0	61	
14		30		46	pwa1	62	
15		31		47		63	

Huh??

The table above has many empty entries. These should not be enabled as they will interfere with the GPU operation.

code/interrupt_party

We're done ... ?

Vector table installed in correct location

- Copy vector table to 0x0 in interrupts_init
- Embed addresses with table so jumps are absolute

Correct transfer of control to/from interrupt mode

- Assembly instructions to save registers, state
- Call into C code
- Assembly instructions to restore registers, resume

Enable checklist

- Enable event detection, attach handler, global enable interrupts

Not quite

An interrupt can fire at any time

- Interrupt handler adds a PS/2 scan code to a queue
- Could do so right as main() is trying to pull a scan code out of the same queue
- Need to maintain integrity of shared queue

Must write code so that it can be safely interrupted

Atomicity

main code

interrupt handler

```
static int nevents;
```

```
    nevents--;
```

```
static int nevents;
```

```
    nevents++;
```

Q. What is the atomic (i.e., indivisible) unit of computation?

Q. Can an update to nevents be lost when switching between these two code paths?

A problem

main code

interrupt handler

```
static int nevents;
```

```
nevents--;
```

```
8074: ldr  r3, [pc, #12]
```

```
8078: ldr  r2, [r3]
```

```
807c: sub  r2, r2, #1
```

```
8080: str  r2, [r3]
```

```
8088: .word 0x0000a678
```

```
static int nevents;
```

```
nevents++;
```

```
808c: ldr  r3, [pc, #12]
```

```
8090: ldr  r2, [r3]
```

```
8094: add  r2, r2, #1
```

```
8098: str  r2, [r3]
```

```
80a0: .word 0x0000a678
```

How can an increment be lost if interrupt occurs here?

A problem

main code

interrupt handler

```
static int nevents;
```

```
nevents--;
```

```
8074: ldr  r3, [pc, #12]
8078: ldr  r2, [r3]
807c: sub  r2, r2, #1
8080: str  r2, [r3]
```


```
8088: .word 0x0000a678
```

```
static int nevents;
```

```
nevents++;
```

```
808c: ldr  r3, [pc, #12]
8090: ldr  r2, [r3]
8094: add  r2, r2, #1
8098: str  r2, [r3]
```

```
80a0: .word 0x0000a678
```



Instruction uses value copied into r2; increment of global by interrupt code is lost

Will volatile solve this?

Disabling interrupts

main code

interrupt handler

```
interrupts_global_disable();  
nevents--;  
interrupts_global_enable();
```

```
nevents++;
```

Q. Does increment need bracketing also?

Preemption and safety

Very hard, lots of bugs.

You'll learn more in CS110/CS140.

Two simple answers

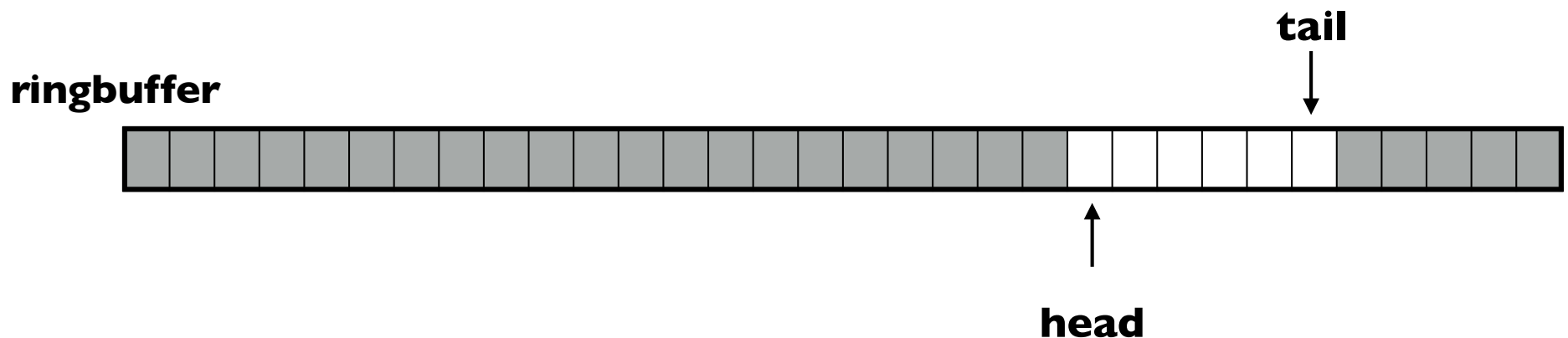
1. Use simple, safe data structures
 - write once, but not always possible
2. Otherwise, temporarily disable interrupts
 - always works, but easy to forget

Safe ringbuffer

A simple approach to avoid interference is for different code paths to not write to same variables

Queue implemented as ring buffer:

- Enqueue (interrupt) writes element to tail, advances tail
- Dequeue (main) reads element from head, advances head



Ringbuffer code

```
bool rb_enqueue(rb_t *rb, int elem)
{
    if (rb_full(rb)) return false;

    rb->entries[rb->tail] = elem;
    rb->tail = (rb->tail + 1) % LENGTH; // only writes tail
    return true;
}
```

```
bool rb_dequeue (rb_t *rb, int *elem)
{
    if (rb_empty(rb)) return false;

    *elem = rb->entries[rb->head];
    rb->head = (rb->head + 1) % LENGTH; // only writes head
    return true;
}
```

C Mastery carryover

Code style

Reading code to learn

Tuning your development process