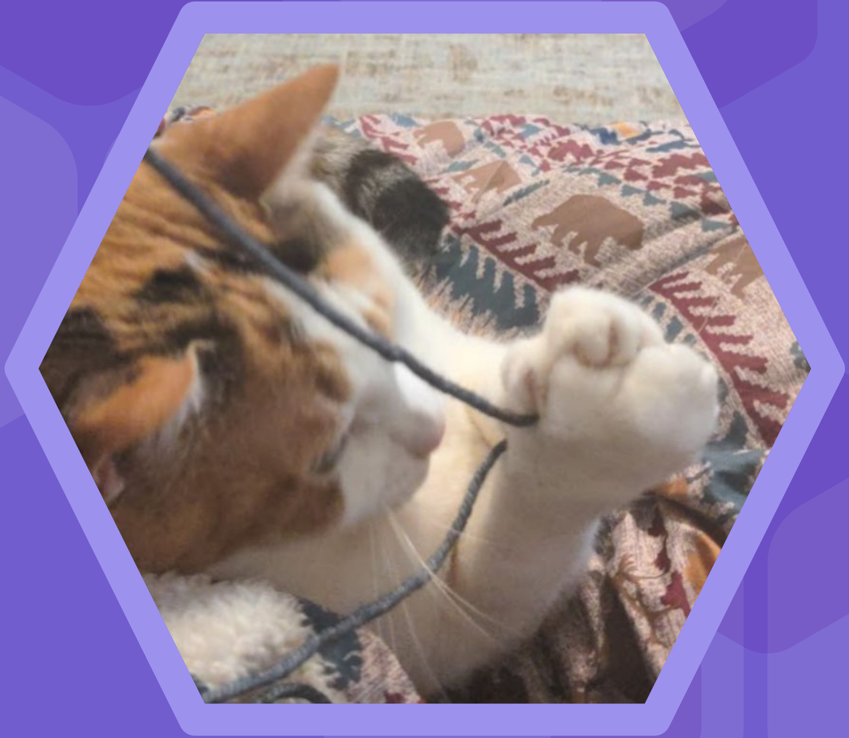
Fill out project matching form by tomorrow at noon!

12: Concurrency Pitfalls



“

What are the limitations of having interrupts as the only source of concurrency in embedded programming?

Cyclic Execution

Threading-*like* behavior without library/os/scheduler

“DIY concurrency”

Each task keeps track of the state it needs

```
void loop() {
  poll_inputs();
  task1();
  task2();
  task3();
}
```

Unlike generalized multi-threading, task finishes before another task executes

Pros/cons to cyclic execution?

Multi-rate cyclic execution

```
void loop() {
  poll_inputs();
  task1();
  poll_inputs();
  task2();
  poll_inputs();
  task3();
}
```

Or even...

```
void loop() {
  poll_inputs();
  task1_step1();
  poll_inputs();
  task1_step2();
  poll_inputs();
  task2_step1();
  poll_inputs();
  task3_step1();
  ...
}
```

Cyclic Execution timing analysis

```
void loop() {
  poll_inputs();
  task1();
  task2();
  task3();
}
```

Worst-case time:

$$T_{loop} = T_{poll\_inputs} + T_{task1} + T_{task2} + T_{task3}$$

(as long as worst-case time of tasks is known)

Timing analysis + interrupts

```
void loop() {
  task1();
  task2();
  task3();
}
```

```
void input_isr() {
  ...
}
```

Assume $T_{task1} + T_{task2} + T_{task3} = 200$ ms
Assume interrupt takes 2 ms, and time between interrupts is at least 20 ms
Worst case execution time of loop + interrupts = ?

11 interrupts can happen in 200ms (if we get unlucky and one is at the beginning of the loop and one is at the end). Accounting for those interrupts, the loop takes 222ms. In those extra 22 ms, an extra interrupt can happen – so the worst-case time is 224ms.

What are the challenges in statically computing worst-case execution time?

Other approaches

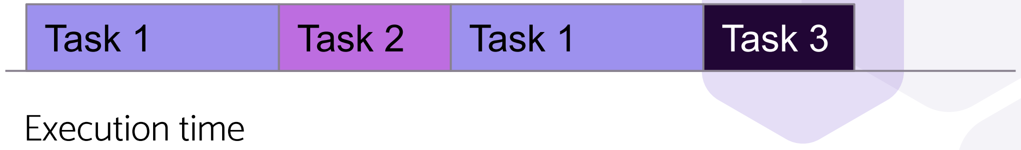
- Time it dynamically
  - Using special debug registers
  - Approximate with timer/counter
  - **Issues?**
- Hybrid (dynamically measure short paths and statically add it up)
  - Many tools on the market do this

Threads and scheduling

Instead of this

```
void loop() {
  task1();
  task2();
  task3();
}
```

CPU schedules each task as its own thread



Generalized multithreading

OS exposes an API for control

(...what OS?!)

Library (like pthreads in C) takes care of things

```
pthread_create(&threads[i], NULL, perform_work, &thread_args[i]);
```

Scheduler schedules threads

More open to control/data pitfalls

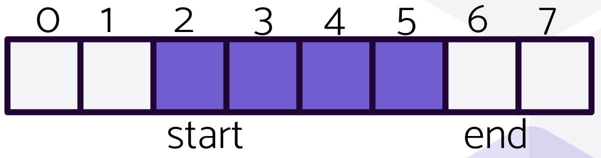
For now: we are talking about single-processor systems

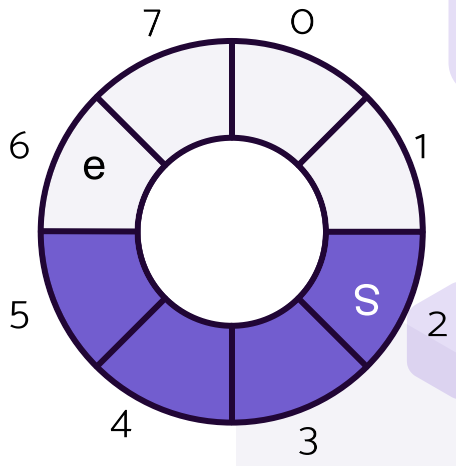
Sharing data – circular buffer

Task/thread A: takes from the start of buffer

Task/thread B: adds to the end of buffer

start/end constantly changing, wrap around





Circular buffer computations

Add to buffer: `buf[e] = data;`

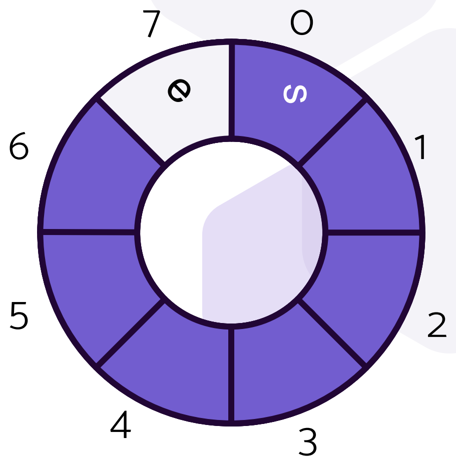
`e = (e + 1) % buf_size`

Take from buffer: `data = buf[s];`

`s = (s + 1) % buf_size`

Is empty? `s == e`

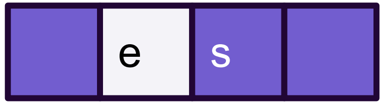
Is full? `(e + 1) % buf_size == s`



Race condition - circular buffer

***Race condition**: order in which two tasks access a resource affects outcome of the program*

n = 4 , s = 2, e = 1



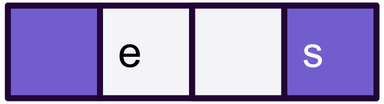
main loop:

```
// if not empty, take from buffer
if(s != e) {
  Serial.println(buffer[s]);
  s = (s + 1) % n
}
```

interrupt:

```
// if still room, store in buffer
if((e + 1) % n != s) {
  buffer[e] = something;
  e = (e + 1) % n
}
```

Race condition - circular buffer



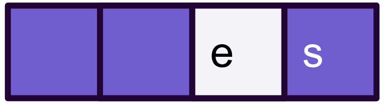
main loop:

```
// if not empty, take from buffer
if(s != e) {
  Serial.println(buffer[s]);
  s = (s + 1) % n
}
```

interrupt:

```
// if still room, store in buffer
if((e + 1) % n != s) {
  buffer[e] = something;
  e = (e + 1) % n
}
```

Race condition - circular buffer



main loop:

```
// if not empty, take from buffer
if(s != e) {
  Serial.println(buffer[s]);
  s = (s + 1) % n
}
```

interrupt:

```
// if still room, store in buffer
if((e + 1) % n != s) {
  buffer[e] = something;
  e = (e + 1) % n
}
```

Memory consistency

```
w = 1;                  y = 1;

x = y;                  z = w;
```

Can we guarantee that at least one of {x, z} will be 1 by the time both threads finish executing?

Depending on compiler optimization, “independent” operations may be rearranged within a thread!!

Mutual exclusion (mutex/lock)

Mechanism that can only be owned by one thread at a time

- Commonly: blocks execution of thread until lock is acquired
- Acquire lock before accessing shared resource, then release it

```
pthread_mutex_lock(&x_lock); // blocks until lock is free
//access x
pthread_mutex_unlock(&x_lock);
```

Deadlock

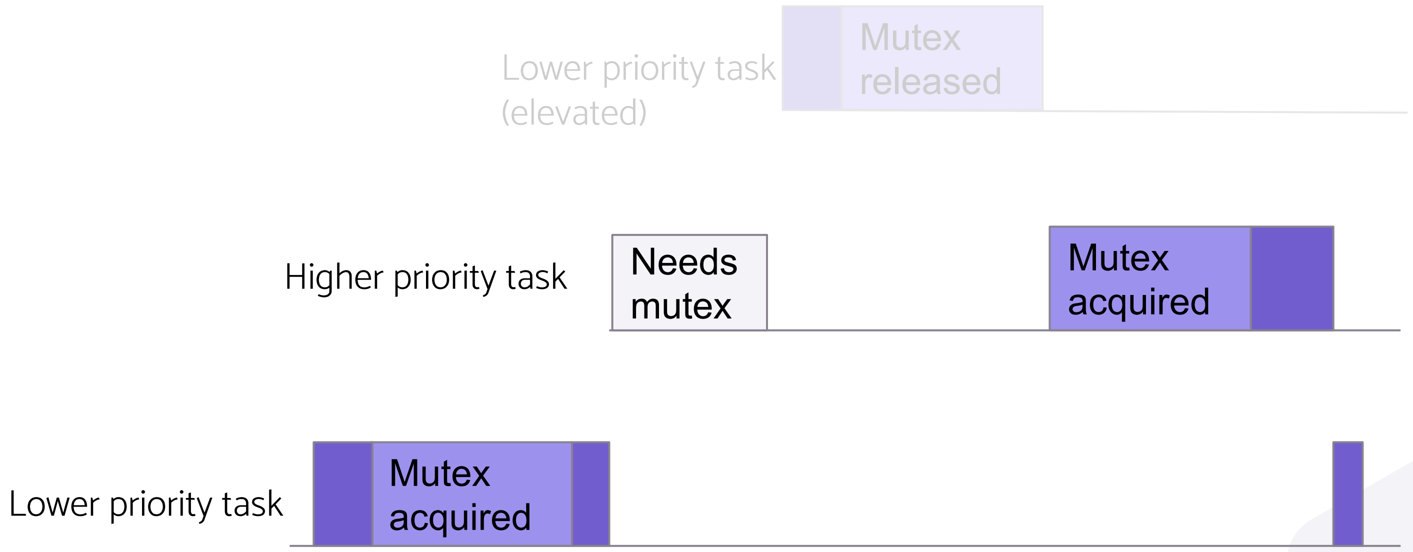
```
pthread_mutex_lock(&lock1);
pthread_mutex_lock(&lock2);
// thread A task
pthread_mutex_unlock(&lock2);
pthread_mutex_unlock(&lock1);
```

```
pthread_mutex_lock(&lock2);
pthread_mutex_lock(&lock1);
// thread B task
pthread_mutex_unlock(&lock1);
pthread_mutex_unlock(&lock2);
```

Priority

- Remember interrupt priorities?
    - Higher-priority interrupt can interrupt lower-priority interrupt but not the other way around
- Task/thread priorities are the same idea
    - In preemptive system, higher-priority tasks can start executing before lower-priority tasks are done
- Various configuration of # of supported priorities, dynamic vs static priorities, etc

Priority inversion


Lower priority task (elevated)
Mutex released
Higher priority task
Needs mutex
Mutex acquired
Lower priority task
Mutex acquired