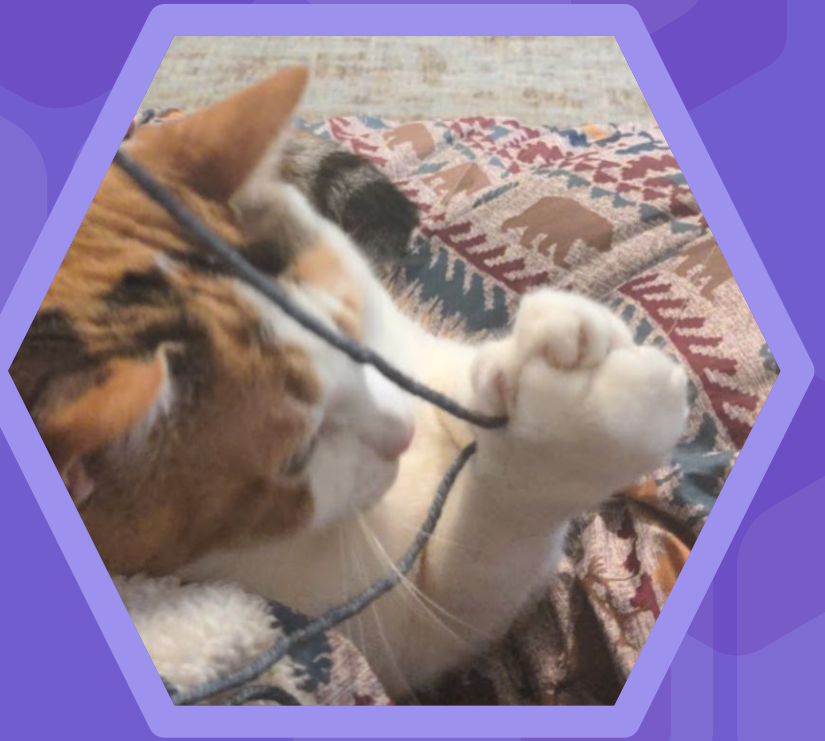
12: Concurrency Pitfalls



Project

Teams are assigned

Proposal due by next Tuesday at 11pm

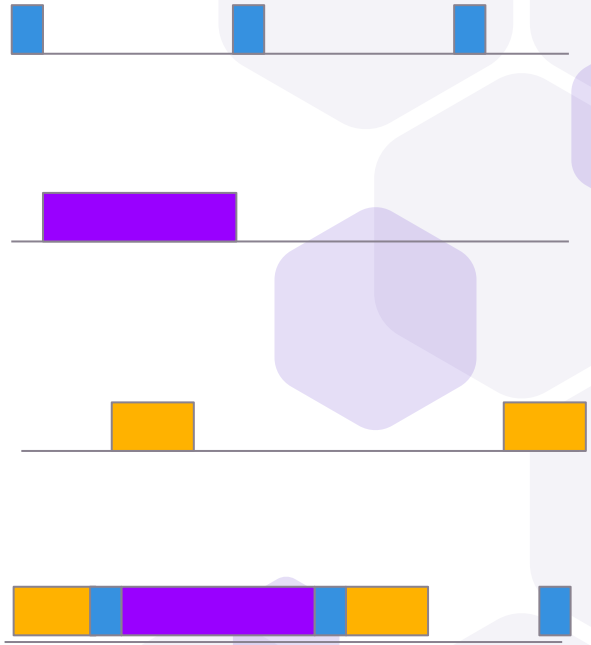
Parts will be ordered by Friday the 14th (watch your e-mail for a form after we look at your proposals)

Scheduling periodic tasks

n tasks each with a given period and worst case execution time

- Assume task's deadline is its period
- Assume independence and 0 cost context switching

Can we schedule this so that all tasks meet their deadlines?



Scheduling

Decide when CPU runs what task so that deadlines are met

Soft: correctness "degrades" if deadlines aren't met

vs

Hard: correctness fails if deadlines aren't met

Dynamic: done at run-time

vs

Static: done at compile-time

Preemptive: task can interrupt lower-priority task

vs

Non-preemptive: tasks can't interrupt each other

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();
}
```

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();
  …
}
```

Latency

Time that a task has to wait to start executing

Cyclic tasks - time between execution of task

    Basically: main loop time

Interrupts - time between trigger and ISR entry

Threads - time between arrival and start

Cyclic Execution timing analysis

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

Worst-case time:

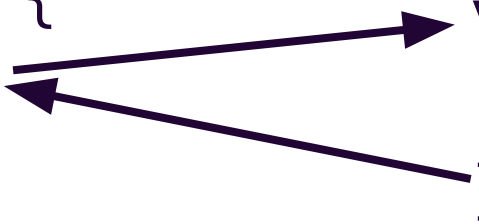
$$T_{loop} = T_{poll\_inputs} + T_{task1} + T_{task2} + T_{task3}$$

(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 happens at most every 20 ms

Worst case execution time of loop + interrupts = ?

① ② 200 + 11*2 = 222 / ③ 222 + 2 = 224 ✓

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)

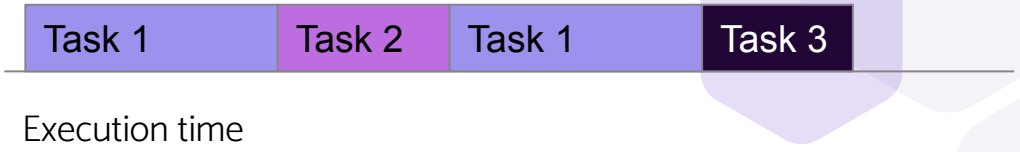
- Many tools on the market do this

Threads and scheduling

Instead of this

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

CPU schedules each task as its own thread



More general 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

Race condition - circular buffer

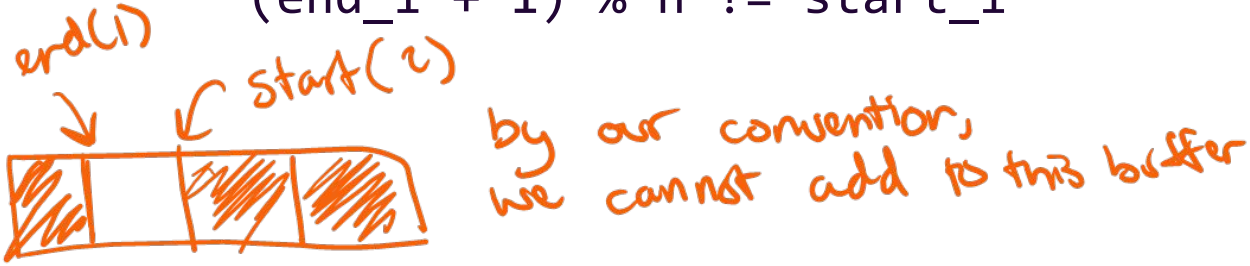
Race condition: order in which two threads access a resource affects outcome of the program

Check that a circular buffer is empty (assume we know it isn't full):

```
start_i == end_i
```

Check that a circular buffer is not about to be full:

```
(end_i + 1) % n != start_i
```

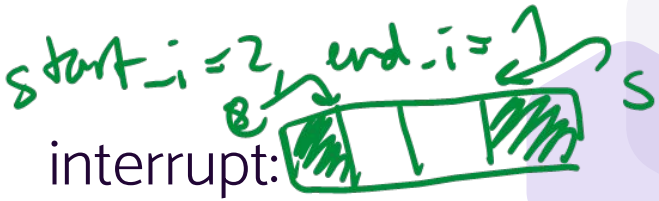


n = 4 , start_i = 2, end_i = 1

main loop:

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

interrupt:




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

end_i = 2

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);
```

Memory consistency

```
w = 1;
x = y;
```

```
y = 1;
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!!

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