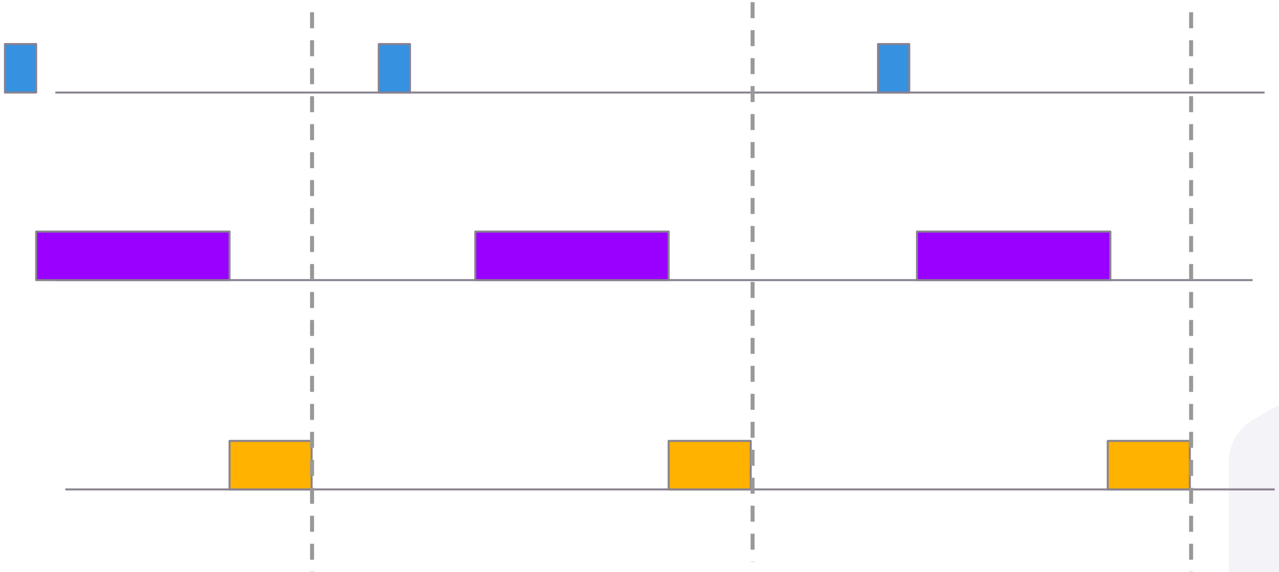
Project matching forms will be sent out after class today

- Teams will be made up of 4 people
- You can choose one other person to work with
- Matching will be done based on project preference

11: Concurrency

A preview: periodic tasks

n tasks each with a given period and worst case execution time (for now assume same period)



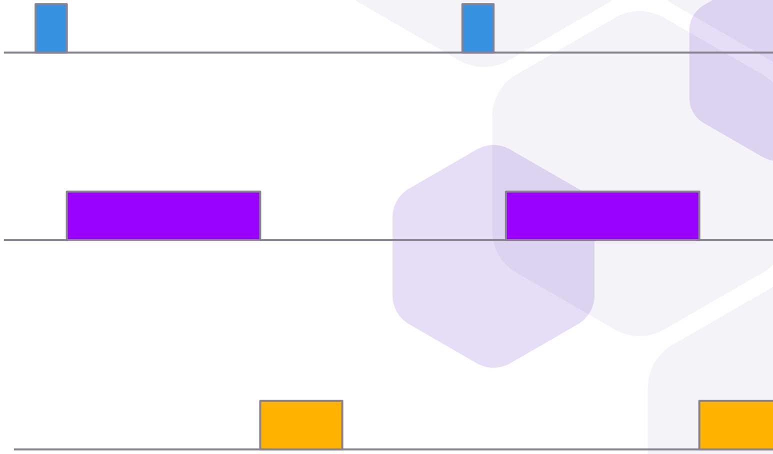
(read and store sensor)

(do complex computation on last sensor reading)

(log output to server)

What's the problem with this?

```
blueTask {
    … do stuff; …
    pet_watchdog; }
purpleTask {
    … do stuff; …
    pet_watchdog; }
goldTask {
    … do stuff; …
    pet_watchdog; }
```



Blocking vs. non-blocking functions

Simplest task scheduler:

```
void loop() {
  blueTask();
  purpleTask();
  goldTask();
}
```

Blocking function:

```
void goldTask() {
  res = 0;
  while (! res) {
    res = serverTask();
  }
  … // compute on res
}
```

Non-blocking function:

```
void goldTask() {
  res = serverTask();
  if (res) {
    // compute on res
  }
}
```

Blocking vs. non-blocking functions

Simplest task scheduler:

```
void loop() {
  blueTask();
  purpleTask();
  goldTask();
}
```

never gets back here

but gets back here

blueTask pets watchdog

Blocking function:

```
void goldTask() {
  int res = 0;
  while (! res) {
    res = serverSend();
  }
  … // compute on res
  petWatchdog();
}
```

Hangs here

Never reaches here

Watchdog isn't pet:
hang successfully detected

Non-blocking function:

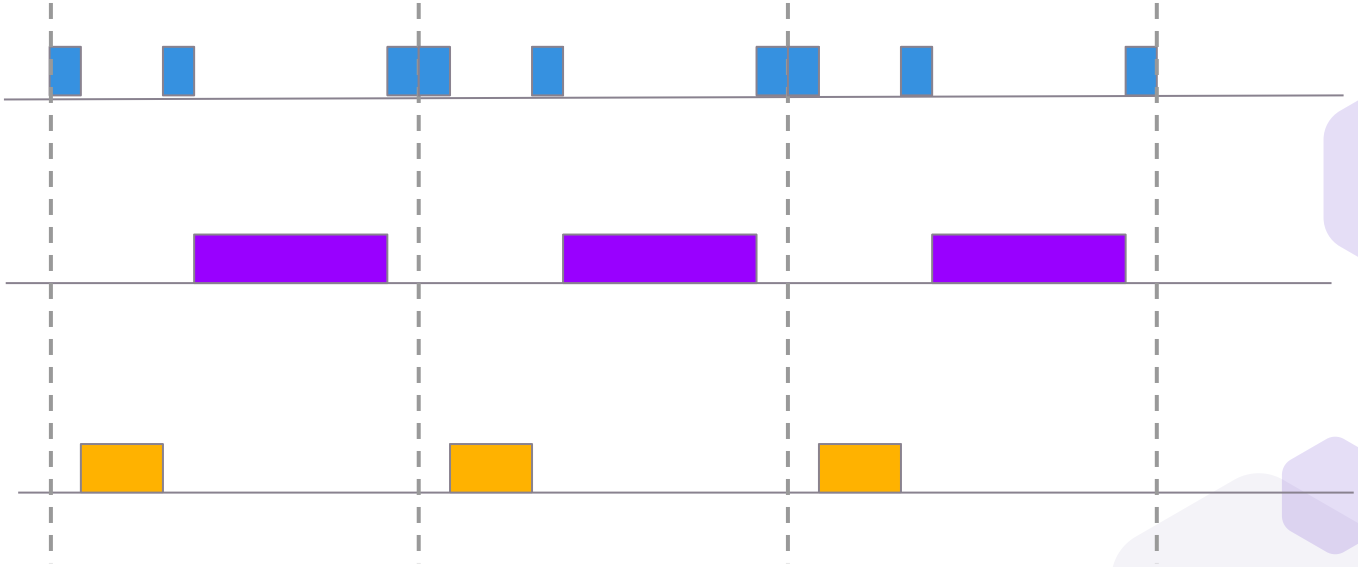
```
void goldTask() {
  int res = serverSend();
  if (res) {
    … // compute on res
    petWatchdog();
  }
}
```

Never reaches here

server hang is never detected!

How would you pet the watchdog for a multitasked system?

Challenge mode



Time and date tales

Imperative programs

(using book definition)

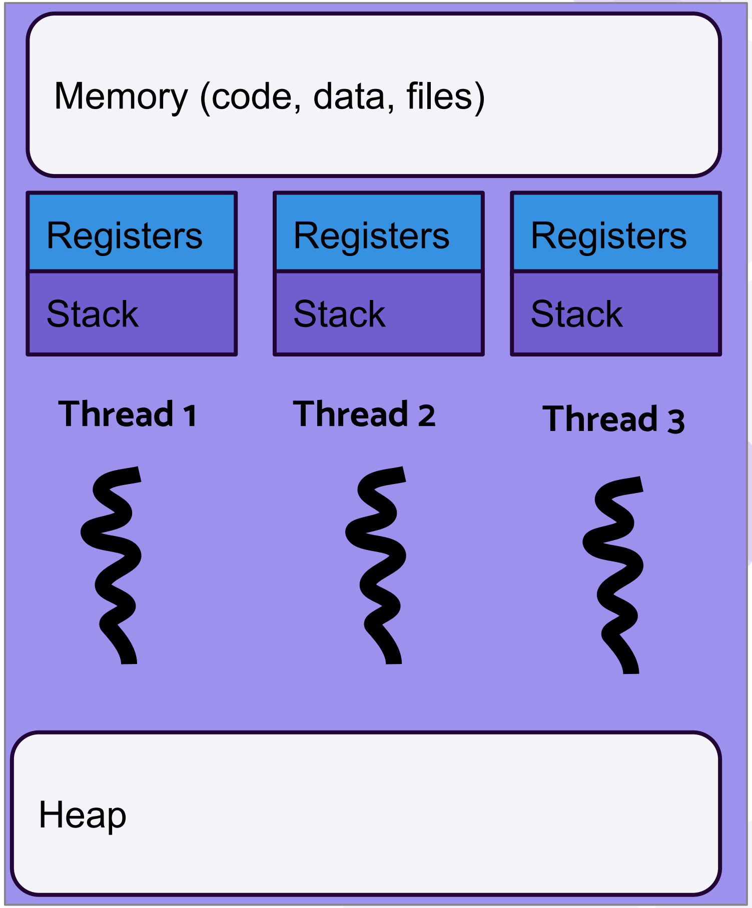
Computation is expressed as a sequence of operations

Each step changes the state of memory on the machine

Threads

Individual imperative programs that run concurrently and share a memory space

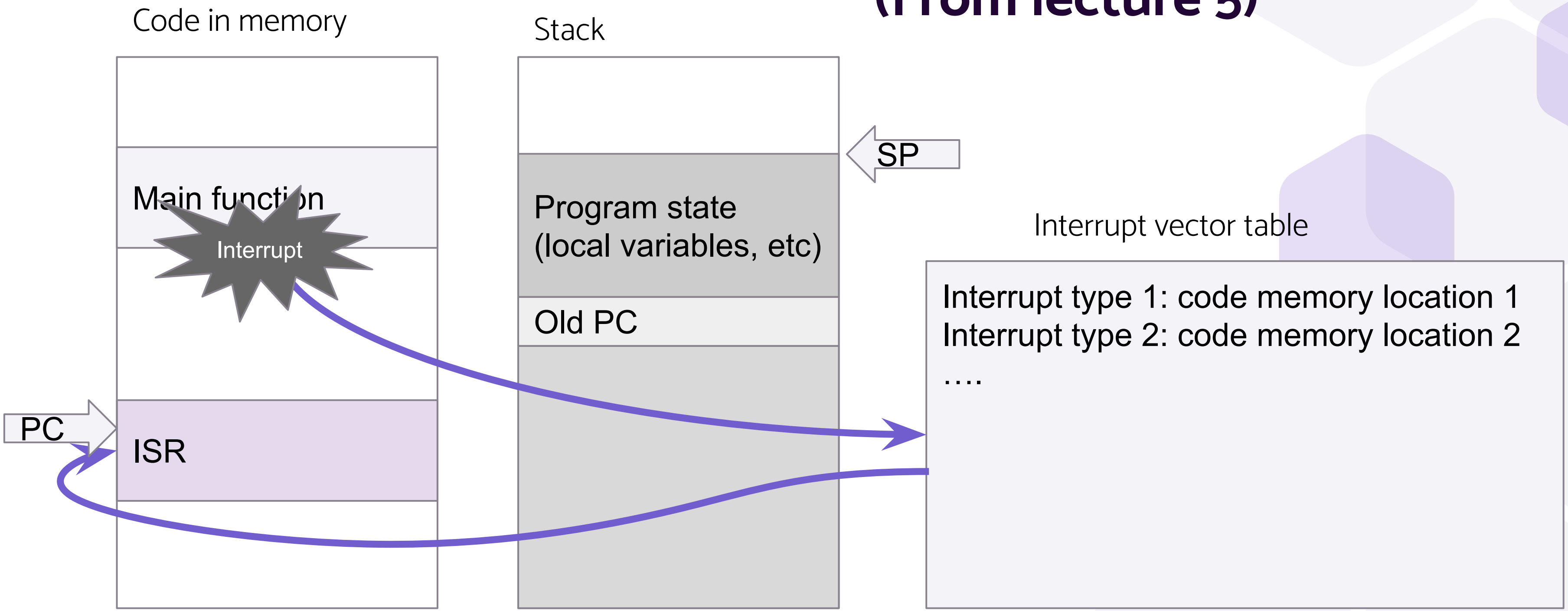
On single-CPU systems, technically only one thread is executing at a given time, but multiple may be "active" (pending computation)



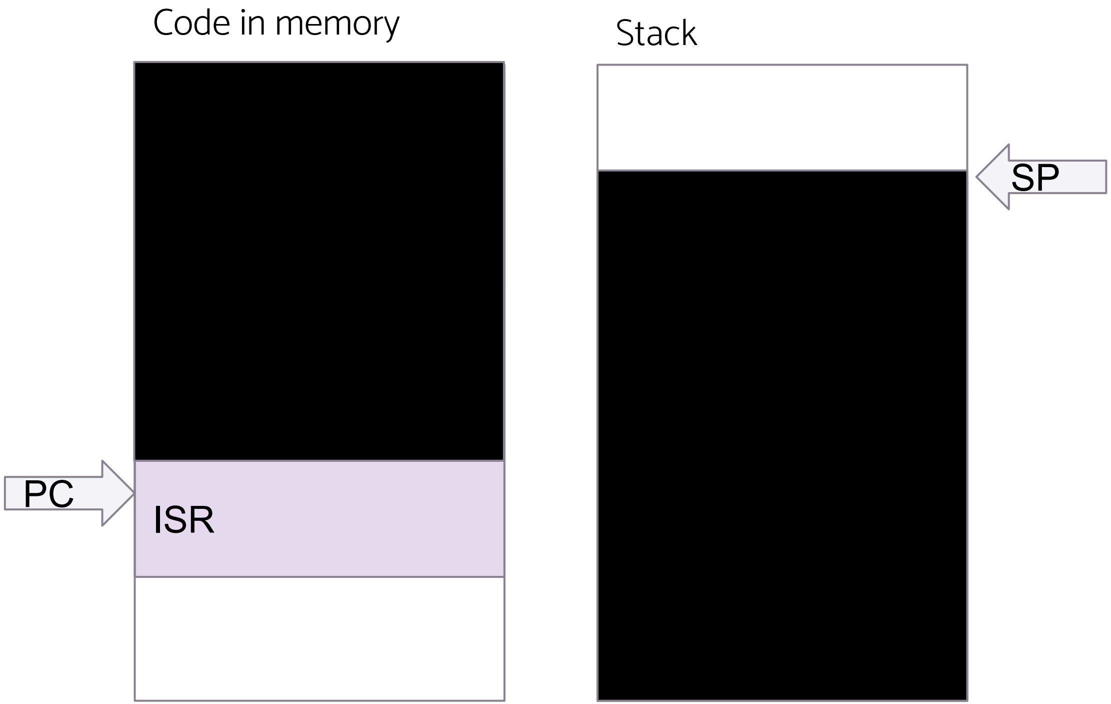
What example of thread-like behavior have we seen so far in this class?

Interrupts as threads

(From lecture 5)

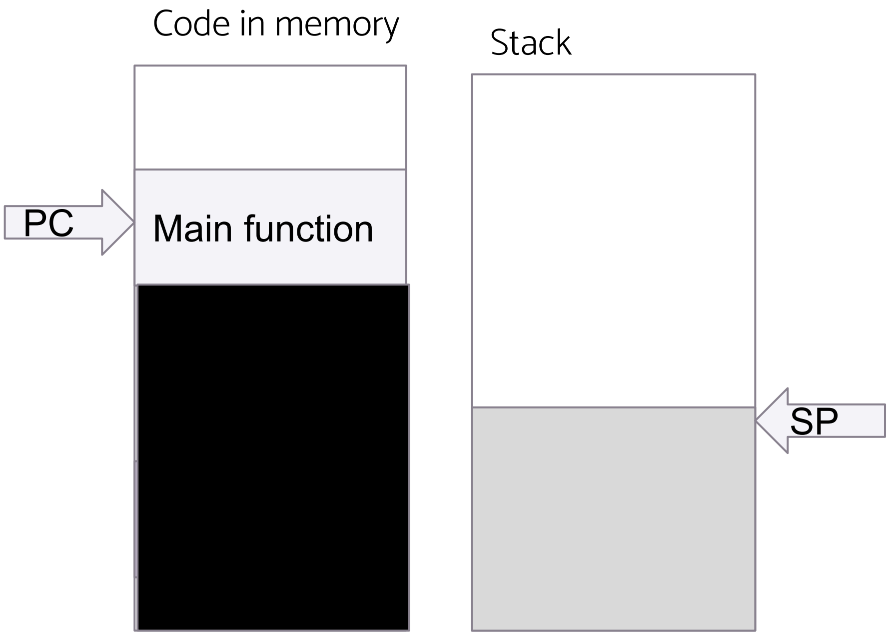


Interrupt's view of execution

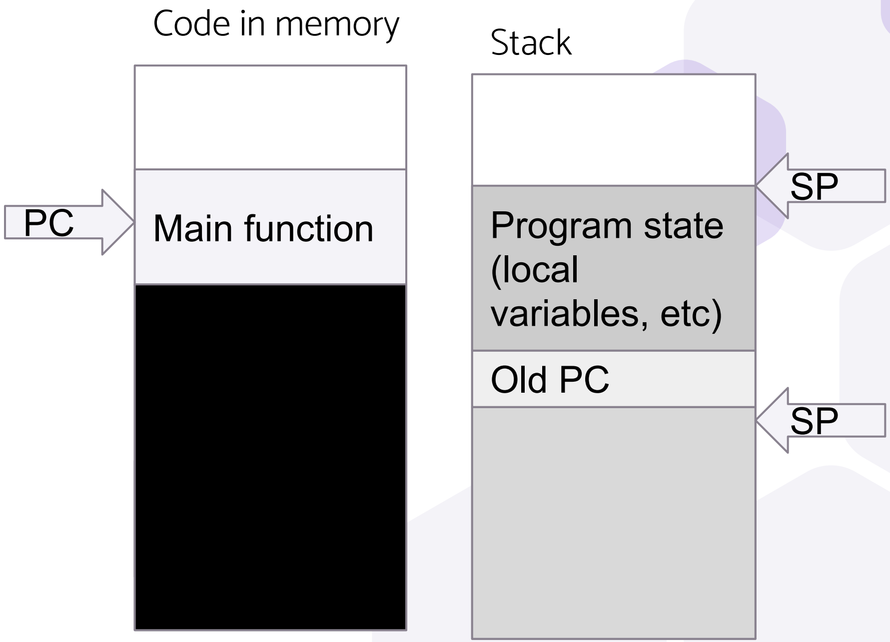


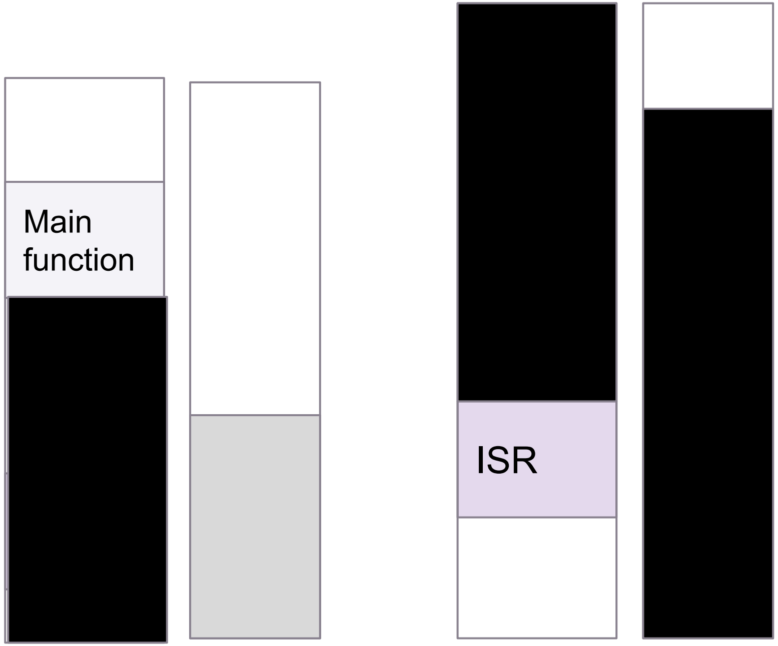
Main process' view of execution

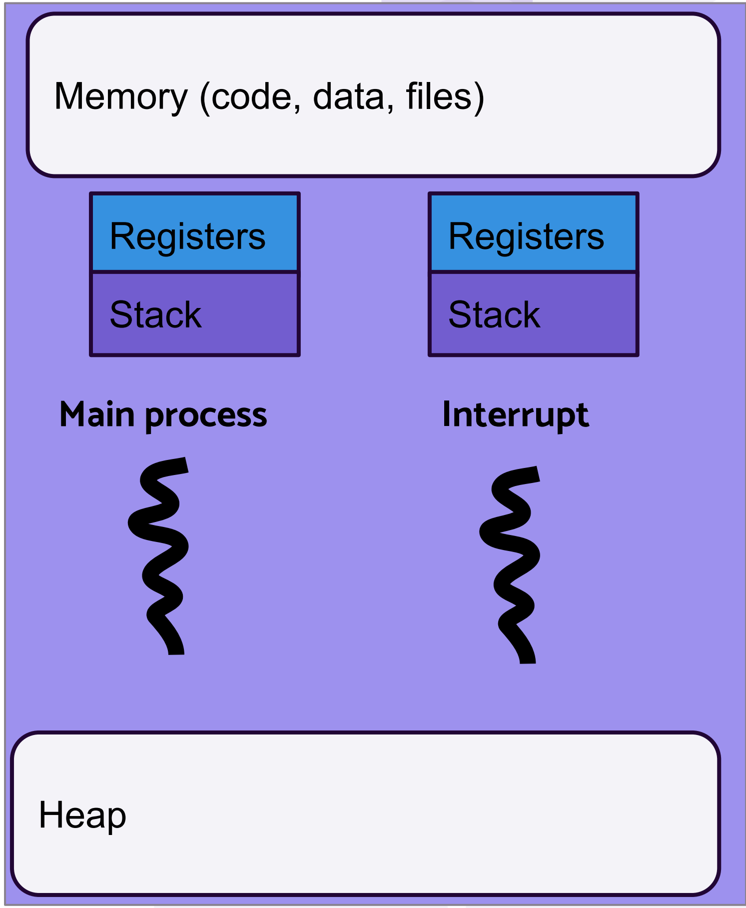
Before interrupt



After interrupt




Main
function
ISR


Memory (code, data, files)
Registers
Stack
Registers
Stack
Main process
Interrupt
Heap

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();
}
```

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 happens at most every 20 ms
Worst case execution time of loop + interrupts = ?

Other approaches

Time it dynamically

- Using special debug registers
- Approximate with timer/counter
- **Issues?**

Hybrid (dynamically measure short paths and statically add it up)

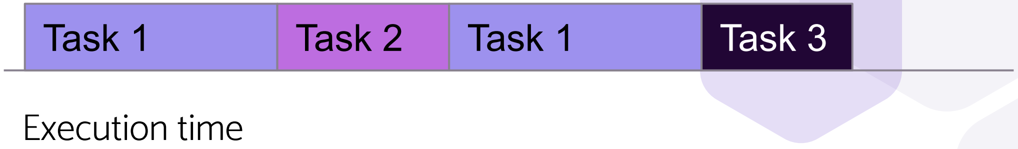
- Many tools on the market do this

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