Multitasking and Real-Time Systems
Today

Multitasking
Scheduling
RTOS
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 2)

Interrupt type 1: code memory location 1
Interrupt type 2: code memory location 2 ....
Interrupt’s view of execution

Code in memory

Stack

PC

ISR

SP
Main process’ view of execution

**Before interrupt**

- **Code in memory**
- **Stack**
  - Program state (local variables, etc)
  - Old PC

**After interrupt**

- **Code in memory**
- **Stack**
  - Program state (local variables, etc)
  - Old PC
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

```c
void loop() {
    poll_inputs();
    task1();
    task2();
    task3();
}
```
Pros/cons to cyclic execution?
Cyclic Execution timing analysis

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

Worst-case time:

\[ T_{\text{loop}} = T_{\text{poll_inputs}} + T_{\text{task1}} + T_{\text{task2}} + T_{\text{task3}} \]

(as long as worst-case time of tasks is known)
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 stimulus and ISR entry

Threads - time between arrival and start
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();
    ...
}
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 = ?
Timing analysis + multiple interrupts

Loop time without interrupts = 200ms

Interrupt 1: 2ms, at most every 20ms

Interrupt 2: 1ms, at most every 10ms

Compute the limit:
In 200 ms, 11x interrupt1, 21x interrupt2: 200 + 22 + 21 = 243ms
In 243 ms, 13x interrupt1, 25x interrupt2: 200 + 26 + 25 = 251ms
In 251 ms, 13x interrupt1, 26x interrupt2: 200 + 26 + 26 = 252ms
In 252 ms, 13x interrupt1, 26x interrupt2: 200 + 26 + 26 = 252ms

← Highest possible latency of this interrupt?
More general multithreading

OS exposes an API for control

(...what OS?!)  

Library (like pthreads in C) takes care of things

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

Scheduler schedules threads

  We’ll talk scheduling strategies soon

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

Recall from lab:

- Check that a circular buffer is empty (assume we know it isn’t full):
  \[
  \text{start}_i == \text{end}_i
  \]
- Check that a circular buffer is not about to be full:
  \[
  (\text{end}_i + 1) \mod n \neq \text{start}_i
  \]
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
}
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

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

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

\[
\begin{align*}
  w &= 1; \\
  x &= y; \\
  y &= 1; \\
  z &= w;
\end{align*}
\]

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

Lower priority task
(elevated)

Mutex
released

Higher priority task

Needs mutex

Mutex acquired

Lower priority task

Mutex acquired
Latency and priority

High priority interrupt: A (4 ms every 10 ms)

Lower priority interrupts: B (7 ms every 100ms), C (1ms every 15 ms)

Can C fail to execute within 15 ms?

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C arrives

A arrives
Questions on threading?
Real-time systems
Correctness depends on the time an answer is delivered, not just the answer

```c
float const_mult(float x, float y) {
    return x * y * C;
}
```
(on your computer, it’s probably fine if this takes more time than anticipated)

```c
float determine_speed(float rpm, float radius) {
    return rpm * radius * CONVERSION_FACTOR;
}
...
void safety_critical_loop() {
    ...
    if (determine_speed(rpm, r) >= SPEED_LIMIT) {
        brake = ON;
    }
    ...
}
```

Same function, different context
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
Threads and scheduling

Instead of this

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

CPU schedules each task as its own thread

```
Task 1  Task 2  Task 1  Task 3
```

Execution time
One approach: cooperative multitasking

Thread is not interrupted unless it calls a procedure saying it’s done

Then other thread starts

**Fairness** concern - can lead to **starvation** of some threads
How can we enforce fairness?
Terminology

Also “arrival time”

For periodic tasks: release time = period offset from start of execution
Deadline = period $p_i$ (an assumption)

Figure 12.1: Summary of times associated with a task execution.

[Lee/Seshia chapter 12]
Criteria for comparing schedulers

**Feasibility**: feasible if $f_i \leq d_i$ for all $i$

**Utilization**: % of time CPU spends executing tasks (vs idle)

**Maximum Lateness**:
Feasibility of scheduling periodic tasks

1) Sum of $e_i/p_i$ for all $i$ is at most 1
   Aka utilization $\leq 100$
   **Necessary but not Sufficient**

2) Can you figure out a way to schedule all tasks during the LCM of all task periods?
   Then you can always schedule the tasks
Types of schedulers

Static - figure it out ahead of time, CPU follows the set schedule

Dynamic:

- Earliest deadline first (EDF)
- Least laxity first (LLF) ($\text{laxity} = d_i - e_i$)
Exercise: statically schedule the following tasks

Board exercise

Periodic scheduling

Utilization $\leq 100\% \rightarrow$ are there cases where a scheduler does not achieve feasibility (non-preemptive vs preemptive EDF)
Rate Monotonic Scheduling (RMS)

Fixed-priority, determined ahead of time

Each task has its own priority

Task with smallest period = highest priority

Pre-emptive (higher priority tasks interrupt lower-priority tasks)

Guarantee of scheduling when utilization < 69.3%
Real-Time Operating Systems

OS - manages system resources and provides services to programs/processes/threads

RTOS - an OS with real-time constraints

- Scheduling policies
- Often support for prioritization
- Libraries for mutexes/semaphores
- Memory management
Pros/cons to using an RTOS?
Would you want to write your own RTOS?
“Free” RTOS considerations

Expertise for being versed in RTOS use isn’t free

Usually when you buy software you also buy support

Patching in updates isn’t free

Industry use of open-source is tricky

License may require release of code
Does your current embedded project use an operating system, RTOS, kernel, software executive, or scheduler of any kind?

81% of those not using OS/RTOSes, said the main reason for NOT using is simply that they are not needed.
Please select ALL of the operating systems you are currently using.

**Regional Breakout**
EMEA uses Embedded Linux much **more** than other regions. APAC uses Android much **more** than other regions and uses Embedded Linux much **less** than others.

<table>
<thead>
<tr>
<th>Most Used</th>
<th>World</th>
<th>Americas</th>
<th>EMEA</th>
<th>APAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embedded Linux</td>
<td>21%</td>
<td>21%</td>
<td>30%</td>
<td>15%</td>
</tr>
<tr>
<td>Android (Google)</td>
<td>13%</td>
<td>9%</td>
<td>14%</td>
<td>27%</td>
</tr>
</tbody>
</table>

Only Operating Systems with 2% or more are shown.

Base: Currently using an operating system

2019 Embedded Markets Study © 2019 Copyright by AspenCore. All rights reserved.
Summary

Multitasking introduces complexity
  Data/control dependencies
  Scheduling
RTOS is a layer to manage scheduling