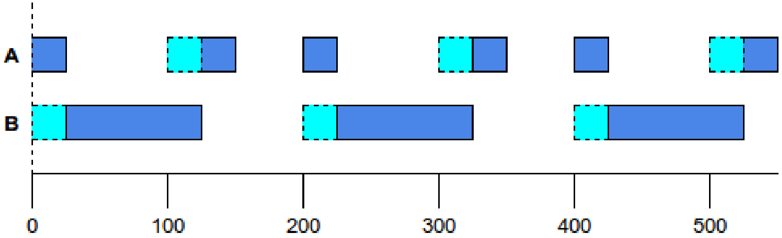
CS122A: Intermediate Embedded and Real Time Operating Systems

Jeffrey McDaniel

University of California, Riverside

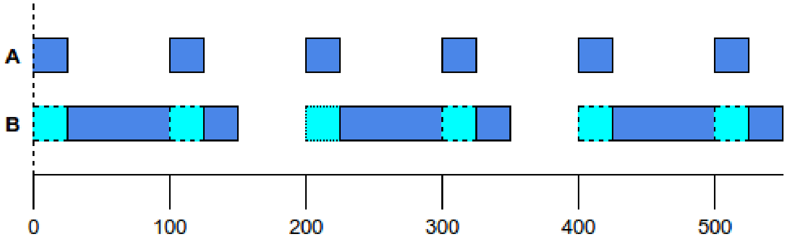
Non-Preemptive vs. Preemptive Scheduler

- Non-preemptive schedulers run a task until completion



Non-Preemptive vs. Preemptive Scheduler

- Non-preemptive schedulers run a task until completion
- Preemptive schedulers are able to interrupt a task it a higher priority task becomes available



Non-Preemptive Scheduler

```
typedef struct task{
  int state;
  unsigned long period;
  unsigned long elapsedTime;
  int (*TickFct)(int);
}

void TimerISR() {
  unsigned char i;
  for(i = 0;i < tasksNum;++i){
        if(tasks[i].elapsedTime >= tasks[i].period){
        tasks[i].elapsedTime = 0;
      tasks[i].state = tasks[i].TickFct(tasks[i].state);
    }
    tasks[i].elapsedTime += tasksPeriodGCD;
  }
}
```

Preemptive Scheduler

```
typedef struct task{
  unsigned char running;
  int state;
  unsigned long period;
  unsigned long elapsedTime;
  int (*TickFct)(int);
}

void TimerISR() {
  unsigned char i;
  for(i = 0;i < tasksNum;++i){
        if(tasks[i].elapsedTime >= tasks[i].period){
        tasks[i].elapsedTime = 0;
      tasks[i].state = tasks[i].TickFct(tasks[i].state);
    }
    tasks[i].elapsedTime += tasksPeriodGCD;
  }
}
```

Preemptive Scheduler

```
unsigned char runningTasks[3] = {255};// Track running tasks
const unsigned long idleTask = 255; // 0 highest priority, 255 lowest
unsigned char currentTask = 0; // Index of highest priority task

void TimerISR() {
  unsigned char i;
  for(i = 0;i < tasksNum;++i){
    if( (tasks[i].elapsedTime >= tasks[i].period) )
      && (runningTasks[currentTask] > i)
      && (!tasks[i].running) ) {// Prevent self-preemption
        tasks[i].elapsedTime = 0;
        tasks[i].running = 1;
        currentTask += 1;
        runningTasks[currentTask] = i;
        tasks[i].state = tasks[i].TickFct(tasks[i].state);
        tasks[i].running = 0;
        runningTasks[currentTask] = idleTask;
        currentTask -= 1;
    }
    tasks[i].elapsedTime += tasksPeriodGCD;
  }
}
```

Preemptive Scheduler

- unsigned char runningTasks[n] = {255}
  - Maintains currently running tasks
  - First element is always idle
  - n: one more than the number of tasks in the system

Preemptive Scheduler

- unsigned char runningTasks[n] = {255}
  - Maintains currently running tasks
  - First element is always idle
  - n: one more than the number of tasks in the system
- runningTasks[currentTask] > i
  - Checks if the currently running task is lower priority

Preemptive Scheduler

- unsigned char runningTasks[n] $= \{255\}$
  - Maintains currently running tasks
  - First element is always idle
  - n: one more than the number of tasks in the system
- runningTasks[currentTask] $>$ i
  - Checks if the currently running task is lower priority
- tasks[i].elapsedTime $= 0$
  - Elapsed time is reset before execution to not attempt to execute again immediately

Preemptive Scheduler

- unsigned char runningTasks[n] = {255}
  - Maintains currently running tasks
  - First element is always idle
  - n: one more than the number of tasks in the system
- runningTasks[currentTask] > i
  - Checks if the currently running task is lower priority
- tasks[i].elapsedTime = 0
  - Elapsed time is reset before execution to not attempt to execute again immediately
- runningTasks[currentTask] = i
  - Add the task to the list of running tasks

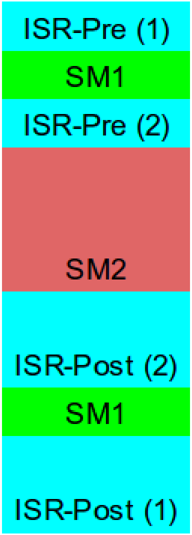
Preemptive Scheduler

- unsigned char runningTasks[n] = {255}
  - Maintains currently running tasks
  - First element is always idle
  - n: one more than the number of tasks in the system
- runningTasks[currentTask] > i
  - Checks if the currently running task is lower priority
- tasks[i].elapsedTime = 0
  - Elapsed time is reset before execution to not attempt to execute again immediately
- runningTasks[currentTask] = i
  - Add the task to the list of running tasks
- runningTasks[currentTask] = idleTask
  - Remove the task from the list of running tasks

Nested Interrupts

If the ISR is running when the timer ticks again, the ISR will be called again, causing a **nested interrupt**.

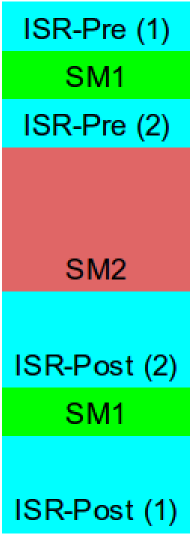
1. ISR (1) is called



Nested Interrupts

If the ISR is running when the timer ticks again, the ISR will be called again, causing a **nested interrupt**.

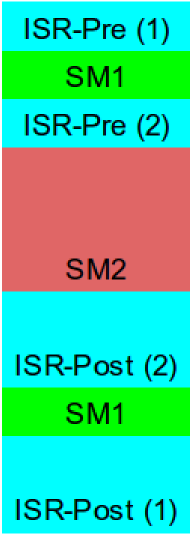
1. ISR (1) is called
2. SM1 begins executing



Nested Interrupts

If the ISR is running when the timer ticks again, the ISR will be called again, causing a **nested interrupt**.

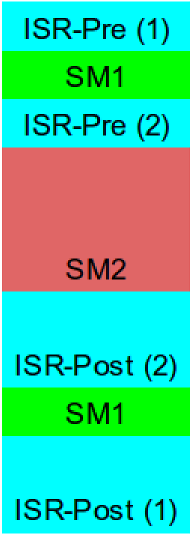
1. ISR (1) is called
2. SM1 begins executing
3. ISR (2) is called at next tick SM1 is still executing



Nested Interrupts

If the ISR is running when the timer ticks again, the ISR will be called again, causing a **nested interrupt**.

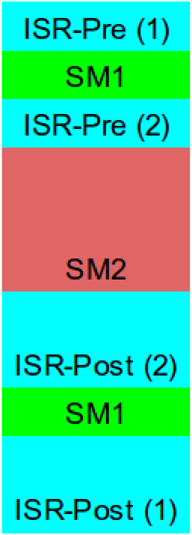
1. ISR (1) is called
2. SM1 begins executing
3. ISR (2) is called at next tick SM1 is still executing
4. SM2 has higher priority and so preempts SM1



Nested Interrupts

If the ISR is running when the timer ticks again, the ISR will be called again, causing a **nested interrupt**.

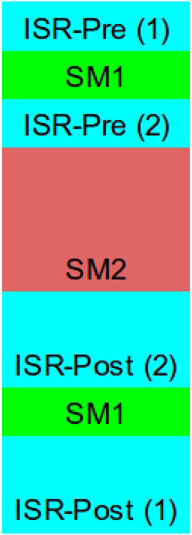
1. ISR (1) is called
2. SM1 begins executing
3. ISR (2) is called at next tick SM1 is still executing
4. SM2 has higher priority and so preempts SM1
5. SM2 completes



Nested Interrupts

If the ISR is running when the timer ticks again, the ISR will be called again, causing a **nested interrupt**.

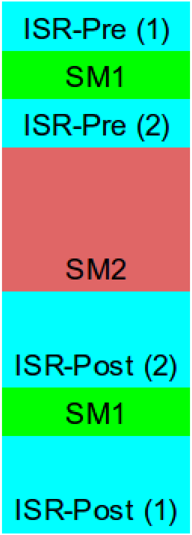
1. ISR (1) is called
2. SM1 begins executing
3. ISR (2) is called at next tick SM1 is still executing
4. SM2 has higher priority and so preempts SM1
5. SM2 completes
6. ISR (2) completes post processing
7. SM1 completes



Nested Interrupts

If the ISR is running when the timer ticks again, the ISR will be called again, causing a **nested interrupt**.

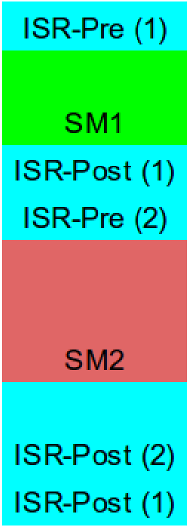
1. ISR (1) is called
2. SM1 begins executing
3. ISR (2) is called at next tick SM1 is still executing
4. SM2 has higher priority and so preempts SM1
5. SM2 completes
6. ISR (2) completes post processing
7. SM1 completes
8. ISR (1) completes post processing



Nested Interrupts

If the ISR is running when the timer ticks again, the ISR will be called again, causing a **nested interrupt**.

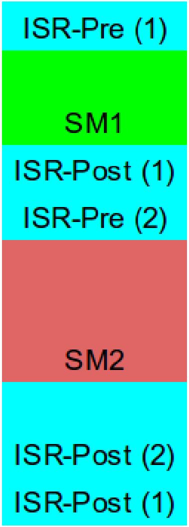
1. ISR (1) is called



Nested Interrupts

If the ISR is running when the timer ticks again, the ISR will be called again, causing a **nested interrupt**.

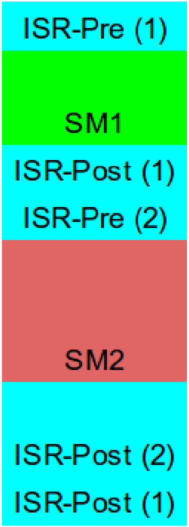
1. ISR (1) is called
2. SM1 begins executing



Nested Interrupts

If the ISR is running when the timer ticks again, the ISR will be called again, causing a **nested interrupt**.

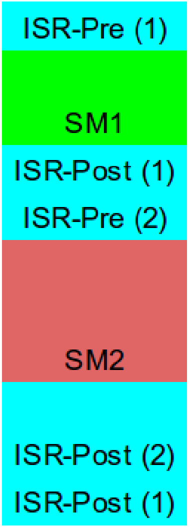
1. ISR (1) is called
2. SM1 begins executing
3. SM1 completes



Nested Interrupts

If the ISR is running when the timer ticks again, the ISR will be called again, causing a **nested interrupt**.

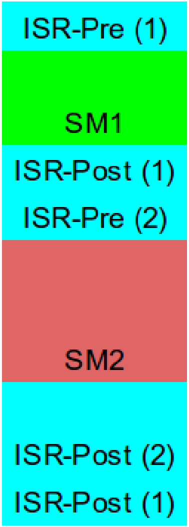
1. ISR (1) is called
2. SM1 begins executing
3. SM1 completes
4. ISR (1) begins post processing



Nested Interrupts

If the ISR is running when the timer ticks again, the ISR will be called again, causing a **nested interrupt**.

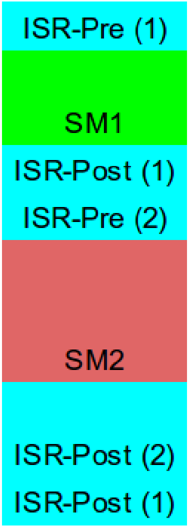
1. ISR (1) is called
2. SM1 begins executing
3. SM1 completes
4. ISR (1) begins post processing
5. ISR (2) is called at next tick ISR (1) has not completed post processing



Nested Interrupts

If the ISR is running when the timer ticks again, the ISR will be called again, causing a **nested interrupt**.

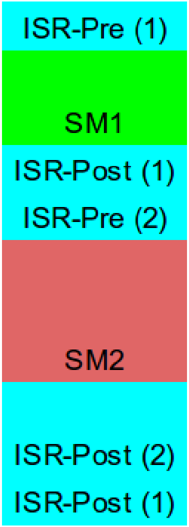
1. ISR (1) is called
2. SM1 begins executing
3. SM1 completes
4. ISR (1) begins post processing
5. ISR (2) is called at next tick ISR (1) has not completed post processing
6. SM2 begins executing



Nested Interrupts

If the ISR is running when the timer ticks again, the ISR will be called again, causing a **nested interrupt**.

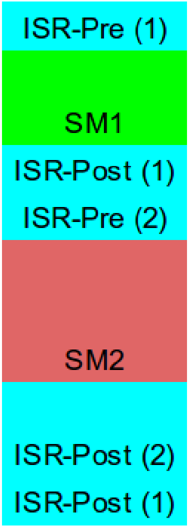
1. ISR (1) is called
2. SM1 begins executing
3. SM1 completes
4. ISR (1) begins post processing
5. ISR (2) is called at next tick ISR (1) has not completed post processing
6. SM2 begins executing
7. SM2 completes



Nested Interrupts

If the ISR is running when the timer ticks again, the ISR will be called again, causing a **nested interrupt**.

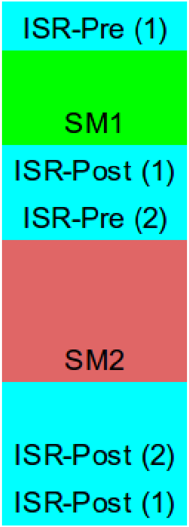
1. ISR (1) is called
2. SM1 begins executing
3. SM1 completes
4. ISR (1) begins post processing
5. ISR (2) is called at next tick ISR (1) has not completed post processing
6. SM2 begins executing
7. SM2 completes
8. ISR (2) completes post processing



Nested Interrupts

If the ISR is running when the timer ticks again, the ISR will be called again, causing a **nested interrupt**.

1. ISR (1) is called
2. SM1 begins executing
3. SM1 completes
4. ISR (1) begins post processing
5. ISR (2) is called at next tick ISR (1) has not completed post processing
6. SM2 begins executing
7. SM2 completes
8. ISR (2) completes post processing
9. ISR (1) completes post processing



Nested Interrupts

- These **nested interrupts** can cause serious error.

Nested Interrupts

- These **nested interrupts** can cause serious error.
- If the ISR is called after currentTask += 1; but before runningTasks[currentTask] = i; then runningTasks[currentTask] would be undefined.

Nested Interrupts

- These **nested interrupts** can cause serious error.
- If the ISR is called after currentTask += 1; but before runningTasks[currentTask] = i; then runningTasks[currentTask] would be undefined.
- This section of code would be called a **critical section**, a section of code that must not be interrupted.

Nested Interrupts

- These **nested interrupts** can cause serious error.
- If the ISR is called after currentTask += 1; but before runningTasks[currentTask] = i; then runningTasks[currentTask] would be undefined.
- This section of code would be called a **critical section**, a section of code that must not be interrupted.
- Interrupts need to be disabled during the **critical section**.

```
void TimerISR() {
  unsigned char i;
  for(i = 0;i < tasksNum;++i){
    if( (tasks[i].elapsedTime >= tasks[i].period) )
      && (runningTasks[currentTask] > i)}
      && (!tasks[i].running) ) {// Prevent self-preemption
        DisableInterrupts();
        tasks[i].elapsedTime = 0;
        tasks[i].running = 1;
        currentTask += 1;
        runningTasks[currentTask] = i;
        EnableInterrupts();
        tasks[i].state = tasks[i].TickFct(tasks[i].state);
        DisableInterrupts();
        tasks[i].running = 0;
        runningTasks[currentTask] = idleTask;}*//
        currentTask -= 1;
        EnableInterrupts();
    }
    tasks[i].elapsedTime += tasksPeriodGCD;
  }
}
```

Stack Overflow

- The amount of code has become significantly larger.

Stack Overflow

- The amount of code has become significantly larger.
- In RIMS the scheduler takes 10ms per task to executed.

Stack Overflow

- The amount of code has become significantly larger.
- In RIMS the scheduler takes 10ms per task to executed.
- A period less than 10ms will cause **stack overflow**.

Stack Overflow

- The amount of code has become significantly larger.
- In RIMS the scheduler takes 10ms per task to executed.
- A period less than 10ms will cause **stack overflow**.
- The microcontroller runs out of memory, causing a fatal error, due to repeated nested calls of TimerISR().

Context Switch

- Switching between concurrently executing tasks is known as a **context switch**.

Context Switch

- Switching between concurrently executing tasks is known as a **context switch**.
- Hardware registers representing the state of the currently executing tasks are save to memory,

Context Switch

- Switching between concurrently executing tasks is known as a **context switch**.
- Hardware registers representing the state of the currently executing tasks are save to memory,
- and are loaded with the values of the other task.

Context Switch

- Switching between concurrently executing tasks is known as a **context switch**.
- Hardware registers representing the state of the currently executing tasks are save to memory,
- and are loaded with the values of the other task.
- This may require low level assembly for this switching

Context Switch

- Switching between concurrently executing tasks is known as a **context switch**.
- Hardware registers representing the state of the currently executing tasks are save to memory,
- and are loaded with the values of the other task.
- This may require low level assembly for this switching
- The above scheduler handles context switches with nested calls to the ISR

Real-Time Operating Systems RTOS)

- Provides an interface between the hardware and the software.

Real-Time Operating Systems RTOS)

- Provides an interface between the hardware and the software.
- Windows and Unix are complex operating systems.

Real-Time Operating Systems RTOS)

- Provides an interface between the hardware and the software.
- Windows and Unix are complex operating systems.
- There are many simpler real-time operating systems (RTOS) available for microprocessors

Real-Time Operating Systems RTOS)

- Provides an interface between the hardware and the software.
- Windows and Unix are complex operating systems.
- There are many simpler real-time operating systems (RTOS) available for microprocessors
- Allow users to define tasks, including perios and priorities

Real-Time Operating Systems RTOS)

- Provides an interface between the hardware and the software.
- Windows and Unix are complex operating systems.
- There are many simpler real-time operating systems (RTOS) available for microprocessors
- Allow users to define tasks, including perios and priorities
- Provides a scheduler (such as the one we have created here).

Real-Time Operating Systems RTOS)

- Provides an interface between the hardware and the software.
- Windows and Unix are complex operating systems.
- There are many simpler real-time operating systems (RTOS) available for microprocessors
- Allow users to define tasks, including perios and priorities
- Provides a scheduler (such as the one we have created here).
- Frequently supports both preemptive and non-preemptive scheduling.