Run-to-Completion Non-Preemptive Scheduler

These lecture notes created by Mr. James B. (Jim) Carlson, NCSU

In These Notes . . .

- What is Scheduling?
- What is non-preemptive scheduling?
- Examples
- Run to completion (cooperative) scheduler

Approaches to Sharing the Processor (Scheduling)

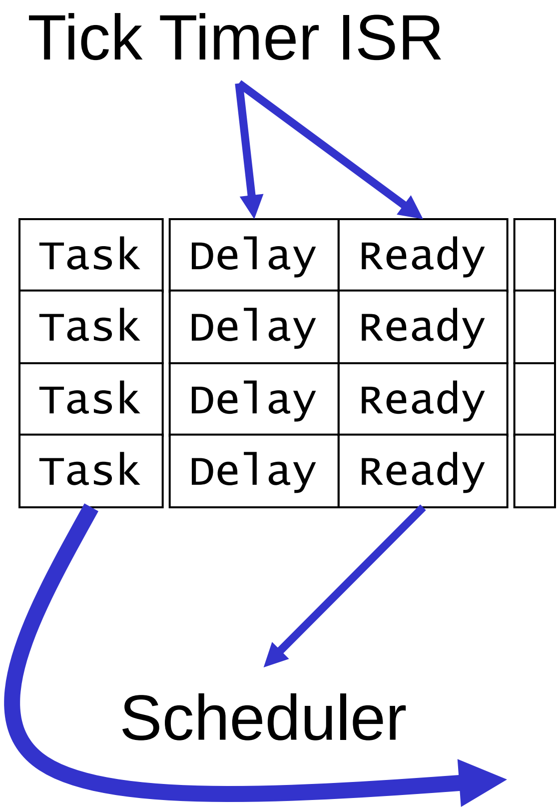
- We have seen two types of code in a program
  - Code in main, or a function called by main (possibly indirectly)
  - Code in interrupt service routines, executing asynchronously
- This approach makes certain behavior difficult to implement
  - E.g. run **Check_for_Overrun()** every 28 milliseconds
  - Run **Update_Display()** every 100 milliseconds
  - Run **Signal_Data_Available()** 300 milliseconds after UART0 Rx interrupt occurs
- Solution 1
  - Use a timer peripheral. Set it to expire after the desired time delay.
  - Problem: Need a timer per event, or a function which makes one timer act like many based on a schedule of desired events.
- Solution 2
  - Break the code into **tasks** which will run periodically
  - Add a **scheduler** which runs the tasks at the right times
    - Scheduling: deciding **which task to run next** and then **starting it running**

Scheduling Rules

- Define functions which need to run periodically to be **tasks**
- If there is a task ready to run, then run it
- Finish the current task before you start another one
  - "Run to completion" = non-preemptive
  - One task can't preempt another task
- If there is more than one task to start, run the highest priority task first

Implementing Scheduler Behavior

- Keep a table with information on each task
  - Where the *task* starts – so we can run it
  - The *period* with which it should run
  - How long of a *delay* until it should run again
    - Decrement this timer every so often
    - Reload it with *period* when it overflows
  - Whether it is *ready* to run now
  - Whether it is *enabled* – so we can switch it on or off easily
- Use a **periodic timer ISR** (e.g. a tick per millisecond) to update *delay* information in the table
- Use **scheduler** to run tasks with *ready* = 1



Task Table Initialization

```
#define MAX_TASKS  5
  // Set maximum number of tasks. Affects performance.
typedef struct {
  int period;                // period of task in ticks
  int delay;                 // time until next run
  int ready;                 // binary: 1 = “run now”
  int enabled;               // active task?
  void (* task)(void);       // address of function
} task_t;
task_t GBL_task_table[MAX_TASKS];

void init_Task_Table(void) {
  // Initialize all tasks entries to empty state
  int i;
  for (i=0 ; i<MAX_TASKS ; i++) {
    GBL_task_table[i].delay = 0;
    GBL_task_table[i].ready = 0;
    GBL_task_table[i].period = 0;
    GBL_task_table[i].enabled = 0;
    GBL_task_table[i].task = NULL;
  }
}
```

Initialize Tick Timer

Set up Timer to generate an interrupt every 1 millisecond

Enable Interrupt

Update Task Table With Each Tick

- On every time tick
  - Reduce *Delay*
  - If *Delay* becomes 0, mark task *Ready* to run and reload *Delay* with *Period*

```
// Make sure to load the vector table with this ISR addr
#pragma INTERRUPT tick_timer_intr
void tick_timer_intr(void) {
static char i;
for (i=0 ; i<MAX_TASKS ; i++) { // If scheduled task
   if ((GBL_task_list[i].task != NULL) &&
       (GBL_task_list[i].enabled == 1) &&
       (GBL_task_list[i].delay != 0)      ) {
          GBL_task_list[i].delay--;
          if (GBL_task_list[i].delay == 0){
              GBL_task_list[i].ready = 1;
              GBL_task_list[i].delay =
                   GBL_task_list[i].period;
          } // if delay == 0
       } // if && && &&
} // for
```

A Simple Example

- We will keep track of how long until the task will run ("delay") and if it is scheduled to run now ("ready")

	Priority	Length	Frequency
Task 1	2	1	20
Task 2	1	2	10
Task 3	3	1	5

Elapsed time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Task executed						**T3**					**T2**		**T3**			**T3**					**T2**		**T1**	**T3**		**T3**
time T1	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	20	19	18	17	16	15
time T2	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5
time T3	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5
run T1																					**1**	1	1			
run T2											**1**										**1**					
run T3						**1**					**1**	1	1			**1**					**1**	1	1	1		**1**

Review of Scheduler Information

- Details
  - Scheduler uses a software timer per task
  - All software timers are decremented using a timer tick based on the Timer hardware overflow interrupt
  - Each task runs to completion before yielding control of MCU back to Scheduler (*non-preemptive)*

Run-to-Completion Scheduler API

- Init_RTC_Scheduler(void)
  - Initialize tick timer B0 and task timers
- Add Task(task, time period, priority)
  - task: address of task (function name without parentheses)
  - time period: period at which task will be run (in ticks)
  - priority: lower number is higher priority. Also is task number.
  - automatically enables task
  - return value: 1 – loaded successfully, 0 – unable to load
- Remove Task(task)
  - removes task from scheduler.
- Run Task(task number)
  - Signals the scheduler that task should run when possible and enables it
- Run RTC Scheduler()
  - Run the scheduler!
  - Never returns
  - There must be at least one task scheduled to run before calling this function.
- Enable_Task(task_number) and Disable_Task(task_number)
  - Set or clear enabled flag, controlling whether task can run or not
- Reschedule_Task(task_number, new_period)
  - Changes the period at which the task runs. Also resets timer to that value.

Running the Scheduler

```
void Run_RTC_Scheduler(void) {  // Always running
  int i;
  GBL_run_scheduler = 1;
  while (1) {              // Loop forever & Check each task
    for (i=0 ; i<MAX_TASKS ; i++) {
      // If this is a scheduled task
      if ((GBL_task_list[i].task != NULL) &&
          (GBL_task_list[i].enabled == 1) &&
          (GBL_task_list[i].ready == 1)     ) {
            GBL_task_list[i].task(); // Run the task
            GBL_task_list[i].ready = 0;
            break;
      } // if && &&
    } // for i
  } // while 1
}
```

Adding a Task

```
int addTask(void (*task)(void), int time, int priority)
{
  unsigned int t_time;
  /* Check for valid priority */
  if (priority >= MAX_TASKS || priority < 0) return 0;
  /* Check to see if we are overwriting an already
   scheduled task */
  if (GBL_task_list[priority].task != NULL) return 0;
  /* Schedule the task */
  GBL_task_list[priority].task = task;
  GBL_task_list[priority].ready = 0;
  GBL_task_list[priority].delay = time;
  GBL_task_list[priority].period = time;
  GBL_task_list[priority].enabled = 1;
  return 1;
}
```

Removing a Task

```
void removeTask(void (* task)(void))
{
  int i;

  for (i=0 ; i<MAX_TASKS ; i++) {
    if (GBL_task_list[i].task == task) {
      GBL_task_list[i].task = NULL;
      GBL_task_list[i].delay = 0;
      GBL_task_list[i].period = 0;
      GBL_task_list[i].run = 0;
      GBL_task_list[i].enabled = 0;
      return;
    }
  }
}
```

Enabling or Disabling a Task

```
void Enable_Task(int task_number)
{
  GBL_task_list[task_number].enabled = 1;
}

void Disable_Task(int task_number)
{
  GBL_task_list[task_number].enabled = 0;
}
```

Rescheduling a Task

- Changes period of task and resets counter

```
void Reschedule_Task(int task_number, int new_period)
{
  GBL_task_list[task_number].period = new_period;
  GBL_task_list[task_number].delay = new_period;
}
```

Start System

To run RTC scheduler, first add the function (task):

```
addTask(flash_redLED, 25, 3);
addTask(sample_ADC, 500, 4);
```

Then, the last thing to do in the main program is:

```
Run_RTC_Scheduler(); // never returns
```

A More Complex Example

- Note at the end, things “stack up” (one T3 missed)

	Priority	Length	Frequency
Task 1	2	1	20
Task 2	1	2	10
Task 3	3	1	5
Task 4	0	1	3

Elapsed time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Task executed				**T4**		**T3**	**T4**			**T4**	**T2**		**T4**	**T3**		**T4**	**T3**		**T4**		**T2**		**T4**	**T1**	**T4**	**T3**
time T1	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	20	19	18	17	16	15
time T2	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5
time T3	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5
time T4	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2
run T1																					**1**	1	1	1		
run T2											**1**										**1**					
run T3						**1**					**1**	1	1	1		**1**	1				**1**	1	1	1	1	**1**
run T4				**1**			**1**			**1**			**1**			**1**			**1**			**1**	1		**1**	

Over-extended Embedded System

- This is an "overextended" system because some tasks are missed – several times. There is not enough processor time to complete all of the work. This is covered in more detail in a future lecture.

	Priority	Length	Frequency
Task 1	2	1	20
Task 2	1	2	10
Task 3	3	1	5
Task 4	0	**2**	3

Elapsed ti	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Task executed				T4		T3	T4			T4		T2		T4		T4		T3	T4		T2		T4		T1	T4		T4		T3	T4
time T1	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	20	19	18	17	16	15	14	13	12	11	10
time T2	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5	4	3	2	1	10
time T3	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5
time T4	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3
run T1																					**1**	1	1	1	1						
run T2											**1**	1									**1**										**1**
run T3						**1**					**1**	1	1	1	1	**1**	1	1			**1**	1	1	1	1	**1**	1	1	1	1	**1**
run T4				**1**			**1**			**1**			**1**	1		**1**			**1**			**1**	1		**1**	1		**1**			**1**