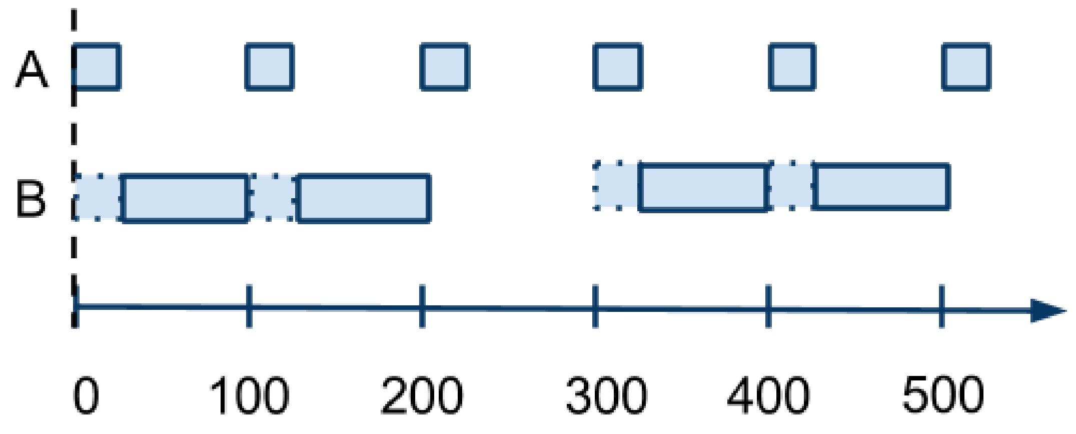
10.6 Scheduling

10.7 Preemptive scheduler

Scheduling approaches can be divided into preemptive and non-preemptive schedulers. A ***preemptive scheduler*** may temporarily stop (preempt) an executing task if a higher-priority task becomes ready. A ***non-preemptive scheduler*** does not stop an executing task; the task must run to completion before the scheduler can execute another task. Our earlier simple task scheduler is non-preemptive. Consider task A with period 100 ms and WCET 25 ms, and B with period 200 and WCET of 100 ms. Utilization is only (25+25+100)/200 or 75%. The following diagram shows microcontroller usage by each task for a preemptive scheduler and for A having higher priority than B.

Figure 10.7.1: Microcontroller usage diagram for a system with a preemptive scheduler.



The earlier simple task scheduler can be rewritten to support preemption:

Figure 10.7.2: Preemptive simple task scheduler.

```
typedef struct task {
  unsigned char running; // 1 indicates task is running
  int state;             // Current state of state machine.
  unsigned long period; // Rate at which the task should tick
  unsigned long elapsedTime; // Time since task's previous tick
  int (*TickFct)(int); // Function to call for task's tick
} task;
// ...
unsigned char runningTasks[3] = {255}; // Track running tasks, [0] always idleTask
const unsigned long idleTask = 255; // 0 highest priority, 255 lowest
unsigned char currentTask = 0; // Index of highest priority task in runningTasks

void TimerISR() {
  unsigned char i;
  for (i=0; i < tasksNum; ++i) { // Heart of scheduler code
     if (  (tasks[i].elapsedTime >= tasks[i].period) // Task ready
         && (runningTasks[currentTask] > i) // Task priority > current task priori
         && (!tasks[i].running) // Task not already running (no self-preemption)
        ) {

        tasks[i].elapsedTime = 0; // Reset time since last tick
        tasks[i].running = 1; // Mark as running

        currentTask += 1;
        runningTasks[currentTask] = i; // Add to runningTasks

        tasks[i].state = tasks[i].TickFct(tasks[i].state); // Execute tick

        tasks[i].running = 0; // Mark as not running
        runningTasks[currentTask] = idleTask; // Remove from runningTasks
        currentTask -= 1;
     }
     tasks[i].elapsedTime += tasksPeriodGCD;
  }
}
```

A few new global variables are added that help to track the currently executing tasks in the system. The task struct is updated with a new data member named running indicates whether or not the task is currently being executed. A new global array named runningTasks is introduced that maintains the currently executing tasks. The first element in the array is always an idle task that has the lowest priority of 255, thus the size of the array should be one larger than the total amount of tasks in the system. A global variable named currentTask holds the index of the currently executing task.

The if statement in the task scheduler for loop is updated with additional conditions that check if the task being evaluated has a higher priority than the currently running task, and if the task is not already running. If either of these conditions are not true, even if the task is ready to execute, then the task will not preempt the currently running task and will instead be delayed. The for loop contents were modified track the status of task execution. First, note that the elapsedTime data member is reset to 0 *before* executing the task, whereas the non-preemptive version resets elapsedTime after task execution. This modification is to ensure that a preempted task does not attempt to execute again immediately during the next scheduling. Second, the task is marked as running and added to the currentTasks array before being executed. Once the task completes, the task is marked as not running and removed from the array.

Using a preemptive scheduler introduces ***nested interrupts***, which occur when an interrupt handler interrupts itself. For example, consider a long-running task that is not completed before the timer period expires – the scheduler was already executing a tick function, but is called again before the last tick could be executed. Not all microcontrollers allow for such nested interrupts, and many may disable them by default. RIMS contains an option (via the Edit dropdown menu) that must be enabled to allow for the preemptive scheduler to work.

Nested interrupts may introduce a serious error if the scheduler is interrupted in the middle of adding or removing a task from the runningTasks array. For example, consider if the statement `currentTask += 1;` executes, but the scheduler is then interrupted before `runningTasks[currentTask] = i` can execute; the value held in runningTasks[currentTask] would be undefined. A ***critical section*** indicates a section of code that must not be interrupted (or accessed by multiple tasks at once, in the context of multi-threaded systems). A programmer can disable interrupts, or in the case of RIMS disable the timer, to ensure critical sections are not interrupted. Note that disabling the timer will skew time slightly, since the instructions in the critical section require a few cycles to execute. A better implementation disables interrupts without affecting timer counting.

Figure 10.7.3: Critical sections should not be interrupted.

```
void TimerISR() {
  unsigned char i;
  for (i=0; i < tasksNum; ++i) { // Heart of scheduler code
     if (  (tasks[i].elapsedTime >= tasks[i].period) // Task ready
         && (runningTasks[currentTask] > i) // Task priority > current task priori
         && (!tasks[i].running) // Task not already running (no self-preemption)
        ) {

        DisableInterrupts(); // Enter critical section. Prevent timer interrupts.
        tasks[i].elapsedTime = 0; // Reset time since last tick
        tasks[i].running = 1; // Mark as running

        currentTask += 1;
        runningTasks[currentTask] = i; // Add to runningTasks
        TimerOn(); //Exit critical section

        tasks[i].state = tasks[i].TickFct(tasks[i].state); // Execute tick

        EnableInterrupts(); // Enter critical section. Prevent timer interrupts.
        tasks[i].running = 0; // Mark as not running
        runningTasks[currentTask] = idleTask; // Remove from runningTasks
        currentTask -= 1;
        TimerOn(); //Exit critical section
     }
     tasks[i].elapsedTime += tasksPeriodGCD;
  }
}
```

The amount of code within the preemptive scheduler has become larger. RIMS requires about 10 ms per task to execute the scheduler overhead. Setting the timer period smaller than the amount of time needed to complete the scheduler loop may cause a ***stack overflow***, a fatal error caused by running out of memory from repeated nested calls of TimerISR().

Switching between concurrently executing tasks is commonly known as a ***context switch***. When a context switch occurs, the values contained in the microcontroller's hardware registers, which represent the state of the currently executing task, are saved to memory. The registers are then loaded with the values of another task and execution continues as though that task was never interrupted. Such saving and loading of registers often requires low-level assembly code for efficiency. Depending on the microcontroller being used, a programmer may need to manually add in assembly code to save registers at the beginning of the scheduler code, and restore those registers at the end of the scheduler code. In the above scheduler code, context switching is handled automatically via nested ISR function calls and returns.

An operating system is a program that runs on a microprocessor to provide an interface between the user's program and hardware. Microsoft Windows and Unix are well-known and complex operating systems intended for desktop computers. Embedded systems sometimes use a ***real-time operating system*** (***RTOS***) (Wikipedia: RTOS). A key part of an

RTOS is a mechanism to allow users to define tasks (sometimes called *threads* or *processes*), including their periods and priorities. Another key part is a task scheduler, such as the scheduler we created above. RTOSes typically support preemptive scheduling, as well as non-preemptive. Numerous microcontroller RTOSes are available, several of them free. (Wikipedia: FreeRTOS).

Participation Activity 10.7.1: Preemptive schedulers.

Determine whether or not the given description applies to a preemptive or non-preemptive scheduler.

#	Question	Your answer
1	A task is paused temporarily so that a higher-priority task can execute. ✓ Preemptive schedulers assign each task a priority and can interrupt low-priority tasks.	[✓ preemptive] non-preemptive
2	A high priority task that becomes ready to execute must wait for a low priority task to complete. ✓ Non-preemptive schedulers can not interrupt currently executing tasks.	preemptive [✓ non-preemptive]
3	Efficient at executing tasks consisting of infinite loops. ✓ Preemption allows a task with an infinite loop to yield control to another task.	[✓ preemptive] non-preemptive

Exploring further:

- RIOS scheduler -- Free non-preemptive and preemptive schedulers written entirely in C, developed by same creators of RIMS and RIBS.
- FreeRTOS -- Free non-preemptive and preemptive real-time OS for microcontrollers.

10.8 Triggered synchSMs