Task Scheduling & Real-Time Operating Systems

Alexander Nelson
November 12, 2018

University of Arkansas - Department of Computer Science and Computer Engineering
Review – Tasks

Task – Unique continuously executing behavior
Concurrent Tasks – Tasks that execute during same time window

Concurrency is difficult:

- Many embedded systems are single-core
- Limited resources for task switching
- Full OS not common
How do you manage concurrent tasks?
How do you manage concurrent tasks?
Periodic Tasks – Concurrent FSM w/ known timing

What about aperiodic tasks?
Managing Concurrent Tasks

How do you manage concurrent tasks?
Periodic Tasks – Concurrent FSM w/ known timing

What about aperiodic tasks?
Real-Time Operating Systems
Roles of an Operating System

Process or Task Management

- Creation, deletion, suspension, resumption
- Manage interprocess communication
- Manage deadlocks

Memory Management

- Track & control tasks in memory
- Monitor memory locality w.r.t processes
- Administer dynamic memory allocation if used

I/O System Management

- Manage access to I/O
- Provide consistent framework for I/O access
Roles of an Operating System

File System Management
- File creation, deletions, access
- Other storage maintenance

System Protection
- Restrict access to certain resources based on privilege

Networking - For distributed applications
- Facilitates remote scheduling of tasks
- Provides interprocess communications across a network

Command Interpretation
- Accessing I/O devices through devices drivers, interface with user to accept and interpret command and dispatch task
Real-Time Operating System

RTOS – Real-Time Operating System

**Real Time** – has time constraints that must be met

**Predictability** – OS will complete tasks by deadline

Predictability/Reliability more important than speed
Time Constraint Categories:

- Hard Real-Time
- Soft Real-Time
- Firm Real-Time
Review: Time Constraints

Time Constraint Categories:

- Hard Real-Time – Unmet condition = System Failure
- Soft Real-Time – Unmet condition = System Degradation
- Firm Real-Time – Degradation leading to failure
Real-Time Operating System Traits

In order to meet constraints, the following RTOS traits are advantageous

- Scheduling Algorithms supported
- Inter-process communication methods
- Preempting (time-based)
- Separate process address space
- Memory protection
- Low memory footprint (both RAM and program memory)
- Timing precision
- *Debugging and Tracing*
Timeliness – “quality of being done or occurring at a favorable or useful time”
Timeliness RTOS – Completing/updating tasks to meet real-time constraints
Achieved by scheduling algorithms
Scheduling – “Arrange to plan to take place at a particular time”
Scheduling (Computing) – “Method by which work specified is assigned to resources to complete the work”

For RTOS, scheduling enables the perception of concurrent execution & meeting of deadlines
Some useful terminology:

- **Execution Time** – Amount of time process requires to complete
- **Persistence** – Amount of time b/w start and termination of task
- **Time-Share** – CPU shares resources between tasks
- **Time Slice** – Short interval of time when processor deals w/ one task
- **Preemption** – A task is interrupted so another can process
Some common scheduling disciplines:

- First come, first served (FCFS)
- Priority Scheduling
- Shortest Time First
- Round-Robin
First Come, First Served (FCFS)
Exactly what it sounds like
Arrival time dictates order, regardless of:

- Priority
- Execution Time

Task runs to completion
First Come, First Served

Advantages:

- Easy to implement
- Low switching overhead
- No starvation
- No Preemption

Why might this be bad for a Real-Time system?
Priority Scheduling

Priority Scheduling – Higher priority tasks preempt lower priority tasks
Low latency for high priority tasks, potential low priority starvation
Shortest Remaining Time First

Shortest Remaining Time - Shorter tasks preempt longer tasks
High task throughput, potential task starvation
Earliest Deadline First – Task with next deadline is executed, preempts allowed
Starvation of tasks with long deadlines
Round Robin – Each task gets a set length time-slice for execution
No starvation, no guarantees to meet deadlines
For RTOS, must evaluate constraints and conditions

- Task priorities?
- Length of tasks
- Variable length tasks?
- Switching overhead, including memory

No single best scheduling algorithm
Multi-Tasking Challenges
Preempting – interrupt a currently executing task

To preempt a task:

- Save state of process
- Save program counter
- Save any registers

These in total are called the “context”

“Context switching” – saving of one state and loading another

Context switching creates overhead in multitasking/preemption
Threads

Thread – smallest set of information needed to run a task
Threads are objects/structures that hold task information
Makes context switching easier

Multithreading – process of running multiple threads on the same system
Ideally, each process should have its own private section of memory to work with, called its address space. Along with hardware support (memory protection unit MPU) an OS may be able to enforce that process do not access memory outside their address space.

Organizational Concepts:

- Multi-process execution – multiple distinct processes running in separate threads
- Multi-threaded process – a process with several execution threads (likely sharing memory and managing resource use internally)
- Note – intraprocess thread context switching is typically less expensive than interprocess context switching
Thread Reentrancy

Reentrant – Function can be interrupted in execution and run again (re-entered) before previous invocations complete

Counterexample:

```c
int tmp;

void swap(int* x, int* y){
    tmp = *x;
    *x = *y;
    *y = tmp; /* invoke isr() here. */
}
```
Thread-Safe code – Other threads or processes can run safely at the same time.
For single core, a different task can preempt and execute fully with no effect on other tasks.
Thread Safe Counterexample

```c
int tmp;
void swap(int* x, int* y){
    int s; /* Save global variable. */
    s = tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;  /* Hardware interrupt isr() here. */
    tmp = s; /* Restore global variable. */
}
void isr(){
    int x = 1, y = 2;
    swap(&x, &y);
}
```
void swap(int* x, int* y){
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;    /* Hardware interrupt isr() here. */
}

void isr(){
    int x = 1, y = 2;
    swap(&x, &y);
}

tmp variable allocated on stack instead of heap
Thread safe & reentrant
Multitasking Coding Practices

Dangerous:

- Multiple calls access same variable/resource
  e.g. Global variables, process variables, pass-by-ref, shared resources

Safe:

- Local Variables – only using local variables makes code reentrant
Implementing RTOS
Most common design philosophies:

- **Event-Driven** – Context switch on event of higher priority
- **Time-Sharing** – Context switch on regularly clocked interrupt

Time-sharing gives smoother multitasking at added overhead cost
Time-sharing may miss deadlines if timing ill-defined
Kernel – “Core” OS functions

- Perform Scheduling – Handled by scheduler
- Dispatch Processes – Handled by the dispatcher
- Facilitate inter-process communication

Must structure tasks to pass between scheduler/dispatcher/OS
Kernel Design Strategies

Some common kernel designs (see if these look familiar):

- Polled Loop – Single instruction tests a flag for event
- Cyclic Executive – Round robin execution of short processes to give illusion of concurrency
- Cooperative Multitasking – Multiple tasks executed in state-driven fashion (code-driven finite state automata)
- Interrupt-Driven System – Main program is a single jump-to-self instruction (e.g. while(1);)
  Short processes executed by ISRs
- Foreground-Background – Interrupts do timely processing of external events – Set up process for main program
If none of those strategies are sufficient, a full-featured RTOS may be needed.

Full-featured RTOS - extension of foreground-background pattern
Adds additional functionality, such as:

- Choice of scheduling algorithm
- OS handles interrupt & pass to relevant tasks
- Support intra-process/task communication
- etc.
Task Control Block (TCB)

OS needs to keep track of tasks

**Process/Task Control Block** – structure containing tasks/processes

Tasks in TCB stored in a “Job Queue”
typedef struct tcb{
    void(*taskPtr)(void * taskDataPtr); //Task function & argument
    void *taskDataPtr; //Pointer for data passing
    void *stackPtr; //Individual Task’s stack
    unsigned short priority; //Priority Information
    struct TCB * nextPtr; //If linked list
    struct TCB *prevPtr; //If doubly linked list
} TCB;
TCB should be generic
Task can be anything, so generic template is needed

Each task written as function, conforming to same generic interface
e.g.

```c
void aTask(void * taskDataPtr){
    //Task Code
}
```
Task’s data is stored in customized container
Task must know the structure
OS refers to data block with generic pointer

typedef struct taskData{
    int taskData0;
    char taskData1;
    ...
} TASKDATA;
A task is typically in one of three states:

- Running – Active task, currently executed by CPU
- Ready – Waiting, ready to be executed
- Blocked – Waiting for an event (e.g. I/O, timing)

Tasks that are ready/blocked are in the TCB job queue
Queue is array of function pointers
Round robin execution of functions in queue
Structure TCB

Queue is array of TCB structures

Round robin execution with data sharing
Scheduler – determine what tasks should run
Tasks in TCB ready queue are selected based on scheduling algorithm

*Choose algorithm based on your system!* – No one-size-fits-all scheduling algorithm
Dispatcher – Gives control of CPU to process selected by scheduler
Also handles interrupts Must:

- Switch Context
- Call function for new task

Dispatch Latency – Amount of time to stop one process & start another
RTOS Issues
Timer Overrun

Tasks have a non-negligible finite execution time
If execution time + overhead > tick frequency, can miss deadlines

Need a construct for detecting timer overruns
unsigned char processingRdyTasks = 0;

void TimerISR() {
    unsigned char i;
    if (processingRdyTasks) {
        printf("Period too short to complete tasks.\r\n\n");
        return;
    }
    processingRdyTasks = 1;

    for (i = 0; i < tasksNum; ++i) { // Heart of scheduler
        if (tasks[i].elapsedTime >= tasks[i].period) { // Ready
            tasks[i].state = tasks[i].TickFct(tasks[i].state);
            tasks[i].elapsedTime = 0;
        }
        tasks[i].elapsedTime += tasksPeriodGCD;
    }
    processingRdyTasks = 0;
}
Utilization – period of time in use
Can be used to predict if timer overrun likely to occur

Utilization = time-per-task-tick / task period

e.g.
Time-per-task-tick = 50ms, task period = 500ms:
Utilization = 50/500 = 10%

If utilization > 100%, overrun will occur
Calculating the time for a task is difficult
Some tasks (e.g. sorting) have a stochastic runtime

Worst Case Execution Time (WCET) – useful to make sure no deadlines missed
Calculating WCET can be done through static (no execution) or dynamic (measured execution) means
Reduction of Utilization

Utilization is composed of two factors:

- WCET Task Time
- Tick Frequency

To reduce utilization, one of these factors must change

WCET Task Time – Optimize Code or split function into two states
Tick Frequency – Decrease tick frequency

What are some advantages/disadvantages of these strategies?
Multitasking & Utilization

Utilization for multiple tasks = \( \frac{\text{sum}(\text{time-per-task})}{\text{task period}} \)

One long task can greatly impact tick frequency for other tasks
hyperperiod – LCM of the task’s periods
During the hyperperiod, the tasks will line up to execute at same time
Thus, utilization during hyperperiod is the same as if they have the same period
Jitter

Jitter – Delay between time task was ready & when it starts executing

Causes:

- Other tasks executing/ready
- Overhead in scheduling/dispatching
What does that mean practically?

1https://stackoverflow.com/a/23352577
Jitter

Timing assumptions are dangerous when multitasking

Best-case/worst-case analysis
Determine what the smallest/largest jitter can/will be
Design real-time system with respect to time-constraints & jitter
Some scheduling is based on task’s period/rate. Deadline is to process task before scheduled again. E.g. Sampling from A/D.

Jitter + Multitasking + Hyperperiod can cause potential missed deadlines.
To handle priority in scheduling, a queue structure can hold tasks in order of priority.

How would you design a task-insert function based on an Earliest Deadline First scheduling into a queue?
Preemptive vs. Non-Preemptive Scheduling

If timing can guarantee deadlines – Non-preemptive much easier to handle
Non-preemptive – no interrupting
Preemptive – interrupts can occur

For preemptive, need to determine if nested interrupts are allowed
Dangers of preemptive scheduling:

- **Stack Overflow** – Too many interrupts & context switches overflow the stack
- **Task Corruption** – Interrupt modifies variables needed by interrupted task
Critical Section

Critical Section – section of code that must occur sequentially & not interrupted
Can disable interrupts surrounding critical section

Critical section should be short and necessary

Long critical sections can break timing constraints
Unnecessary critical sections may be better to roll back & restart than hold up execution
Instead of a `while(1)` or `while(!TimerFlag)`, a `sleep()` may be used. Sleep modes typically reduce active power consumption.

**Table 1**: MSP430fr2633 Sleep Modes (16MHz)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>Typ</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>Executing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LPM0</td>
<td>CPU off</td>
<td>420µA</td>
<td>—</td>
</tr>
<tr>
<td>LPM3</td>
<td>CPU &amp; Main Oscillator off</td>
<td>1.18µA</td>
<td>1.65µA</td>
</tr>
<tr>
<td>LPM4</td>
<td>Everything off (must be woken by external interrupt)</td>
<td>0.49µA</td>
<td>—</td>
</tr>
</tbody>
</table>

Apple Watch (v1) battery capacity = 205mAh — 205 hours of operation at 1mA average consumption.
Sleep modes often have varying wake-up times – jitter overhead

Timed events can use timer interrupts to wake from sleep (and account for wake-up)
When talking about sleep modes, “duty-cycles” are times when controller is active
Reduce Power = Reduce duty-cycle time
Duty Cycle analysis useful to determine total power consumption

Average power = Average active power × duty cycle %
+ Average sleep power × 1 - duty cycle %

Example:
2% duty cycle, LPM3 sleep, typical active power:
Average Power = 1000μA × 0.02 + 1.18μA × 0.98
Average Power = 20μA + 1.156μA = 21.156μA
Active Power still consumes ≈ 94.5% power despite active only 2% of the time
Recent debate – battery vs. supercapacitor
Battery stores energy through chemical reaction
Supercapacitor stores energy through capacitance (a medium’s ability to hold charge)

Battery holds more charge – degrades over time, charges slower
Supercapacitor holds less charge – rechargeable more times, charges quickly