

EECS 373: Introduction to Embedded System Design Real-Time Systems and Operating Systems

Robert Dick

University of Michigan

Outline

1. Realtime systems
2. Rate monotonic scheduling overview
3. Real-time and embedded operating systems

Section outline

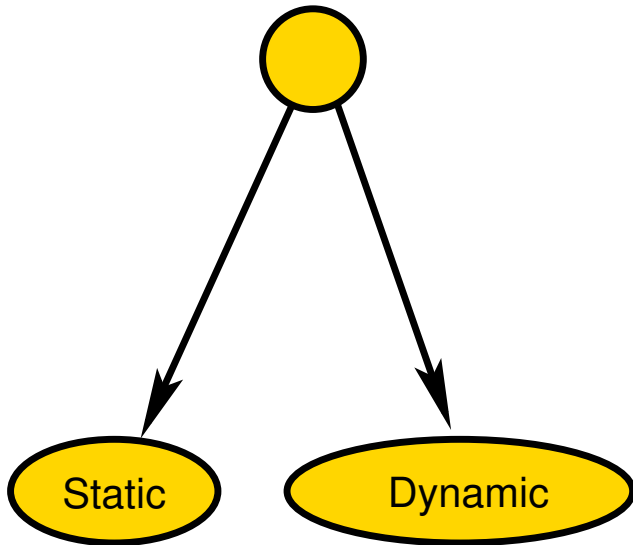
1. Realtime systems

Taxonomy

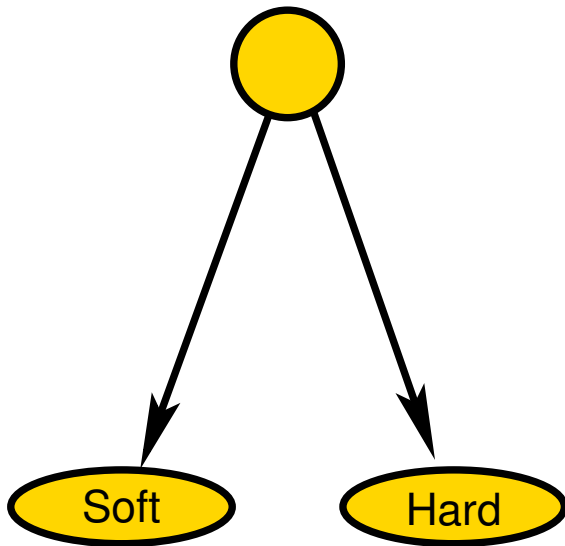
Definitions

Central areas of real-time study

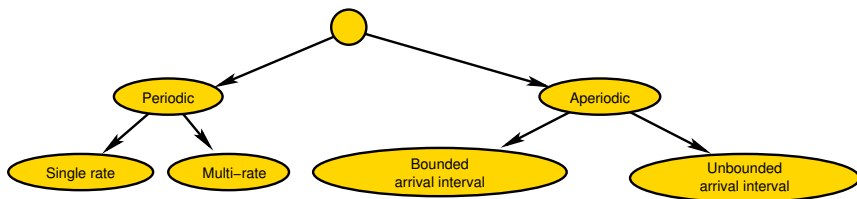
Taxonomy of real-time systems



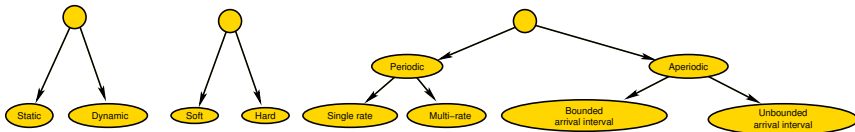
Taxonomy of real-time systems



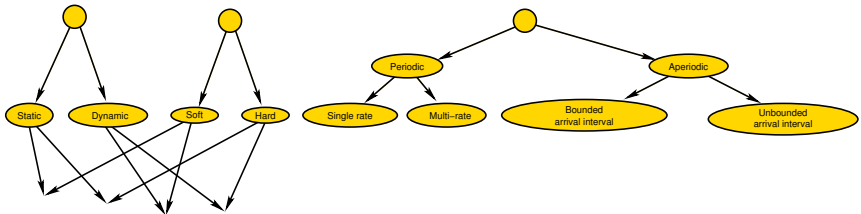
Taxonomy of real-time systems



Taxonomy of real-time systems



Taxonomy of real-time systems



Static

Task arrival times can be predicted.

Static (compile-time) analysis possible.

Allows good resource usage (low processor idle time proportions).

Sometimes designers shoehorn dynamic problems into static formulations allowing a good solution to the wrong problem.

Dynamic

Task arrival times unpredictable.

Static (compile-time) analysis possible only for simple cases.

Even then, the portion of required processor utilization efficiency goes to 0.693.

In many real systems, this is very difficult to apply in reality (more on this later).

Use the right tools but don't over-simplify, e.g.,

We assume, without loss of generality, that all tasks are independent.

Soft real-time

More slack in implementation.

Timing may be suboptimal without being incorrect.

Problem formulation can be much more complicated than hard real-time.

Two common (and one uncommon) methods of dealing with non-trivial soft real-time system requirements.

- Set somewhat loose hard timing constraints.
- Informal design and testing.
- Formulate as optimization problem.

Hard real-time

Difficult problem. Some timing constraints inflexible.

Simplifies problem formulation.

Periodic

Each task (or group of tasks) executes repeatedly with a particular period.

Allows some nice static analysis techniques to be used.

Matches characteristics of many real problems. . .

. . . and has little or no relationship with many others that designers try to pretend are periodic.

Periodic \rightarrow single-rate

One period in the system.

Simple.

Inflexible.

This is how a *lot* of wireless sensor networks are implemented.

Periodic \rightarrow multirate

Multiple periods.

Can use notion of circular time to simplify static (compile-time) schedule analysis E. L. Lawler and C. U. Martel, "Scheduling periodically occurring tasks on multiple processors," *Information Processing Ltrs.*, vol. 7, pp. 9–12, Feb. 1981.

Co-prime periods leads to analysis problems.

Periodic \rightarrow other

It is possible to have tasks with deadlines less than, equal to, or greater than their periods.

Results in multi-phase, circular-time schedules with multiple concurrent task instances.

If you ever need to deal with one of these, see me (take my code). This class of scheduler is nasty to code.

Aperiodic

Also called sporadic, asynchronous, or reactive.

Implies dynamic.

Bounded arrival time interval permits resource reservation.

Unbounded arrival time interval impossible to deal with for any resource-constrained system.

Section outline

1. Realtime systems

Taxonomy

Definitions

Central areas of real-time study

Definitions

Task.

Processor.

Graph representations.

Deadline violation.

Cost functions.

Task

Some operation that needs to be carried out.

Atomic completion: A task is all done or it isn't.

Non-atomic execution: A task may be interrupted and resumed.

Processor

Processors execute tasks.

Distributed systems.

- Contain multiple processors.
- Inter-processor communication has impact on system performance.
- Communication is challenging to analyze.

One processor type: Homogeneous system.

Multiple processor types: Heterogeneous system.

Task/processor relationship

WC exec time (s)

Tooth	7.7E-6	...	
Road	330E-9	...	
FIR	4.1E-6	...	
Matrix	310E-3	...	

IBM PowerPC 405GP 266 MHz
 IDT79RC32364 100 MHz
 Imsys Cjip 40 MHz

Relationship between tasks, processors, and costs.

Examples: power consumption or worst-case execution time.

Cost functions

Mapping of real-time system design problem solution instance to cost value.

I.e., allows price, or hard deadline violation, of a particular multi-processor implementation to be determined.

Back to real-time problem taxonomy: jagged edges

Some things can dramatically complicate real-time scheduling.

Data dependencies.

Unpredictability.

Distributed systems.

Heterogeneous processors.

Preemption.

Section outline

1. Realtime systems

Taxonomy

Definitions

Central areas of real-time study

Central areas of real-time study

Allocation, assignment and **scheduling**.

Operating systems and **scheduling**.

Distributed systems and **scheduling**.

Scheduling is at the core of real-time systems study.

Operating systems and scheduling

How does one best design operating systems to

Support sufficient detail in workload specification to allow good control, e.g., over scheduling, without increasing design error rate.

Design operating system schedulers to support real-time constraints?

Support predictable costs for task and OS service execution.

Distributed systems and scheduling

How does one best dynamically control

The assignment of tasks to processing nodes...

... and their schedules.

for systems in which computation nodes may be separated by vast distances such that

Task deadline violations are bounded (when possible)...

... and minimized when no bounds are possible.

The value of formality: optimization and costs

The design of a real-time system is fundamentally a cost optimization problem.

Minimize costs under constraints while meeting functionality requirements.

Functionality requirements are actually just constraints.

Why view problem in this manner?

Without having a concrete definition of the problem.

- How is one to know if an answer is correct?
- More subtly, how is one to know if an answer is optimal?

Optimization

Thinking of a design problem in terms of optimization gives design team members objective criterion by which to evaluate the impact of a design change on quality.

Know whether your design changes are taking you in a good direction

Outline

1. Realtime systems
2. Rate monotonic scheduling overview
3. Real-time and embedded operating systems

Primary publication

C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. of the ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

Rate monotonic scheduling (RMS)

Single processor.

Independent tasks.

Differing arrival periods.

Schedule in order of increasing periods.

No fixed-priority schedule will do better than RMS.

Guaranteed valid for loading $\leq \ln 2 = 0.69$.

For loading $> \ln 2$ and < 1 , correctness unknown.

Usually works up to a loading of 0.88.

Rate monotonic scheduling

1973, Liu and Layland derived optimal scheduling algorithm(s) for this problem.

C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. of the ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

Schedule the job with the smallest period (period = deadline) first.

Analyzed worst-case behavior on any task set of size n .

Found utilization bound: $U(n) = n \cdot (2^{1/n} - 1)$.

0.828 at $n = 2$.

As $n \rightarrow \infty$, $U(n) \rightarrow \log 2 = 0.693$.

Result: For any problem instance, if a valid schedule is possible, the processor need never spend more than 31% of its time idle.

Outline

1. Realtime systems
2. Rate monotonic scheduling overview
3. Real-time and embedded operating systems

Threads

Threads vs. processes: Shared vs. unshared resources.

OS impact: Windows vs. Linux.

Hardware impact: MMU.

Threads vs. processes

Threads: Low context switch overhead.

Threads: Sometimes the only real option, depending on hardware.

Processes: Safer, when hardware provides support.

Processes: Can have better performance when IPC limited.

Section outline

3. Real-time and embedded operating systems
 - RTOS functionality
 - Operating system scheduler examples
 - FreeRTOS

Real-time operating systems

Embedded vs. real-time.

Dynamic memory allocation.

Schedulers: General-purpose vs. real-time.

Timers and clocks: Relationship with HW.

Real-time operating systems

Interaction between HW and SW.

- Fast and predictable response to interrupts.
- HW interface abstraction.

Interaction between different tasks.

- Communication.
- Synchronization.

Multitasking

- Ideally fully preemptive.
- Priority-based scheduling.
- Fast and predictable context switching.
- Support for real-time clock.

General-purpose OS stress

Good average-case behavior.

Providing many services.

Support for a large number of hardware devices.

RTOSs stress

Predictable service execution times.

Predictable scheduling.

Good worst-case behavior.

Low memory usage.

Speed.

Simplicity.

Predictability

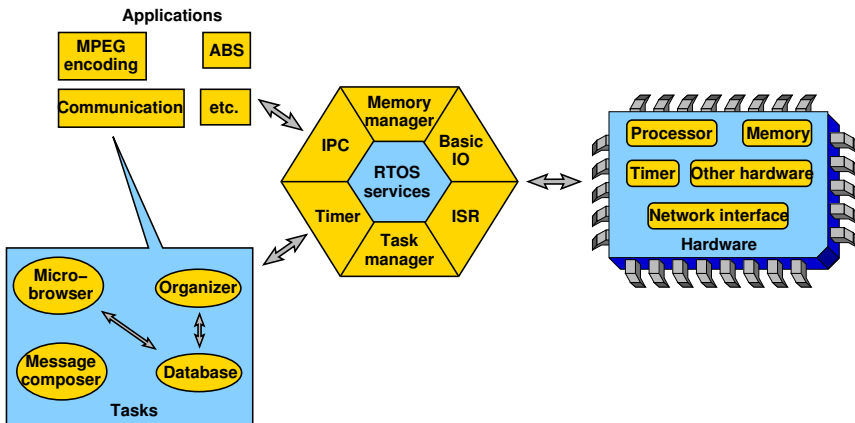
General-purpose computer architecture focuses on average-case.

- Caches.
- Prefetching.
- Speculative execution.

Real-time embedded systems need predictability.

- Disabling or locking caches is common.
- Careful evaluation of worst-case is essential.
- Specialized or static memory management common.

RTOS overview



Section outline

3. Real-time and embedded operating systems
 - RTOS functionality
 - Operating system scheduler examples
 - FreeRTOS

Software implementation of schedulers

TinyOS.

Light-weight threading executive.

μ C/OS-II.

Linux.

Static list scheduler.

TinyOS

Most behavior event-driven.

High rate \rightarrow livelock.

Research schedulers exist.

BD threads

Brian Dean: Microcontroller hacker.

Simple priority-based thread scheduling executive.

Tiny footprint (fine for AVR).

Low overhead.

No MMU requirements.

μ C/OS-II

Similar to BD threads.

More flexible.

Bigger footprint.

Old Linux scheduler

Single run queue.

$\mathcal{O}(n)$ scheduling operation.

Allows dynamic goodness function.

$O(1)$ scheduler in Linux 2.6+

Written by Ingo Molnar.

Splits run queue into two queues prioritized by goodness.

Requires static goodness function.

No reliance on running process.

Compatible with preemptable kernel.

$O(\log n)$ scheduler in Linux 2.6.23+

Written by Ingo Molnar.

Used red-black tree to maintain accumulated task times.

Always schedules task with least accumulated time.

Weights accumulated time based on priority.

Real-time Linux

Run Linux as process under real-time executive.

Complicated programming model.

RTAI (Real-Time Application Interface) attempts to simplify.

Colleagues still have problems at > 18 kHz control frequency.

Section outline

3. Real-time and embedded operating systems
 - RTOS functionality
 - Operating system scheduler examples
 - FreeRTOS

Many RTOS options

eCos.

LynxOS.

MontaVista Linux.

QNX.

RTAI.

RTLinux.

Symbian OS.

VxWorks.

FreeRTOS.

Etc.

Example: FreeRTOS

Free and open source, complete with source.

Well documented.

Easy to use.

Does the basics well.

FreeRTOS

Each task is a function that must not return.

You inform the scheduler of

- The task's resource needs (stack space, priority).
- Any arguments the tasks needs.

All tasks here must be of void return type and take a single void* as an argument.

You cast the pointer as needed to get the argument.

Example task

```
void vTask1( void *pvParameters )
{
    const char *pcTaskName = "Task 1 is running\r\n";
    volatile unsigned long ul;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation. There is
            nothing to do in here. Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Example task creation code I

```
portBASE_TYPE  
xTaskCreate(pdTASK_CODE pvTaskCode,  
            const char * const pcName,  
            unsigned short usStackDepth,  
            void *pvParameters,  
            unsigned portBASE_TYPE uxPriority,  
            xTaskHandle *pvCreatedTask  
);
```

- Create a new task and add it to the list of tasks that are ready to run. `xTaskCreate()` can only be used to create a task that has unrestricted access to the entire microcontroller memory map. Systems that include MPU support can alternatively create an MPU constrained task using `xTaskCreateRestricted()`.
- `pvTaskCode`: Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).

Example task creation code II

- `pcName`: A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by `tskMAX_TASK_NAME_LEN` – default is 16.
- `usStackDepth`: The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and `usStackDepth` is defined as 100, 200 bytes will be allocated for stack storage.
- `pvParameters`: Pointer that will be used as the parameter for the task being created.
- `uxPriority`: The priority at which the task should run. Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit `portPRIVILEGE_BIT` of the priority parameter. For example, to create a privileged task at priority 2 the `uxPriority` parameter should be set to $(2 \text{ — } \text{portPRIVILEGE_BIT})$.
- `pvCreatedTask`: Used to pass back a handle by which the created task can be referenced.

Example task creation code III

- pdPASS: If the task was successfully created and added to a ready list, otherwise an error code defined in the file errors.h.

Task creation

```
int main( void )
{
    /* Create one of the two tasks. Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1, /* Pointer to the function that implements the task. */
                  "Task 1", /* Text name for the task. This is to facilitate
                             debugging only. */
                  1000, /* Stack depth - most small microcontrollers will use
                             much less stack than this. */
                  NULL, /* We are not using the task parameter. */
                  1, /* This task will run at priority 1. */
                  NULL ); /* We are not going to use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();
}
```

Task behavior

Task will run if there are no other tasks of higher priority.

And if others the same priority will RR.

But that begs the question: "How do we know if a task wants to do something or not?"

The previous example gave always wanted to run.

Just looping for delay (which is bad).

Instead should call `vTaskDelay(x)`.

Delays current task for X "ticks".

There are a few other APIs for delaying.

Other task-related functionality

Change priority of a task.

Delete a task.

Suspend a task.

Get priority of a task.

Etc.

Other functionality

Interrupts.



Including deferred interrupts.

Memory management.

Standard I/O interfaces.

Fast context switch.

Locks so only one task can use certain resources at a time.

-  E. L. Lawler and C. U. Martel, "Scheduling periodically occurring tasks on multiple processors," *Information Processing Ltrs.*, vol. 7, pp. 9–12, Feb. 1981.
-  C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. of the ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.