ECEN 3753: Real-Time Operating Systems

Realtime Risk Mitigations

ARM Interrupt Optimizations, Priority Inheritance, Execution Complexity, RMA

Recalling Some Risks and Adding More

- Control Loop Latency
  - We looked at compensation for hold-off
  - *More mitigation* with **Interrupt Optimizations**
- Deadlock
  - We have looked at Shared Data mitigation
  - *Resource contention mitigation* with **Priority Inheritance**
- Execution Complexity
  - *How "completely-testable" is my system?* and what can help?
- Scheduling Guarantees
  - *Can I be assured of deadline conformance?* **Rate Monotonic Analysis (RMA)**

ARM Interrupt Optimizations

Recall last week, another possible help was mentioned for Control Loop Latency Mitigation:

- Use of Interrupts that have register banking

What is **Register Banking**?

ARM Interrupt Specializations (not just performance)

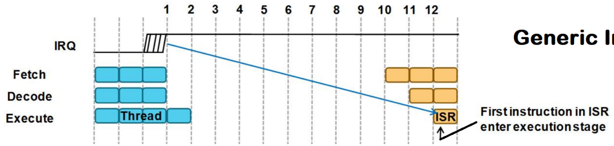
Considerations that may allow you to limit some risk:

- Execution mode (affects stack selection, banking)
  - Any interrupt (since configured only in Privileged mode) is Privileged.
  - Generally, ISR returns to mode it came from. Supervisory Calls (SVC exceptions) can force return to Supervisor Mode.
  - Dual stacks help security, robustness. How?
- Register Banking (GIC) / Tail Chaining (NVIC)

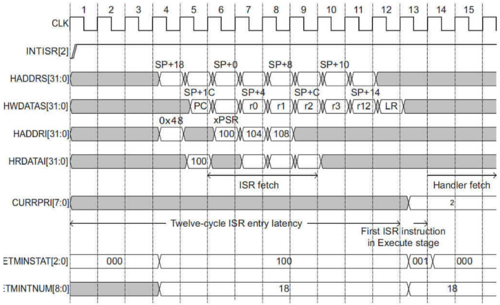
Operation → Execution ↓	**Thread Mode** (Process Stack)	**Exception Mode** (Main Stack)
Priviledged	"Supervisor"	"Handler" (and Reset)
Unpriviledged	Normal	<N/A>

This table is applicable for NVIC. GIC introduces more modes, including many inspired by Security-related concerns.

NVIC IRQ Latency

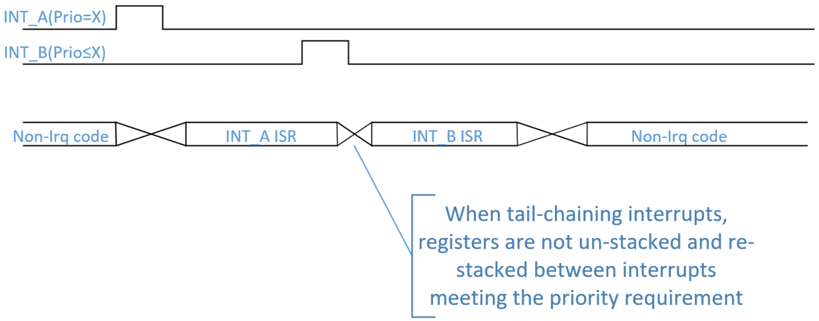


Generic Interrupt Latency

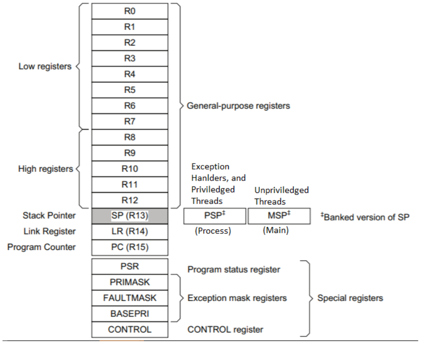


M4-NVIC IRQ Latency
(plus any additional register pushes!)

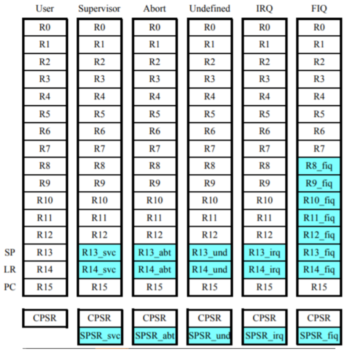
NVIC Interrupt Chaining


INT_A(Prio=X)
INT_B(Prio≤X)
Non-Irq code
INT_A ISR
INT_B ISR
Non-Irq code
When tail-chaining interrupts, registers are not un-stacked and re-stacked between interrupts meeting the priority requirement

ARM NVIC vs. GIC Banking



NVIC

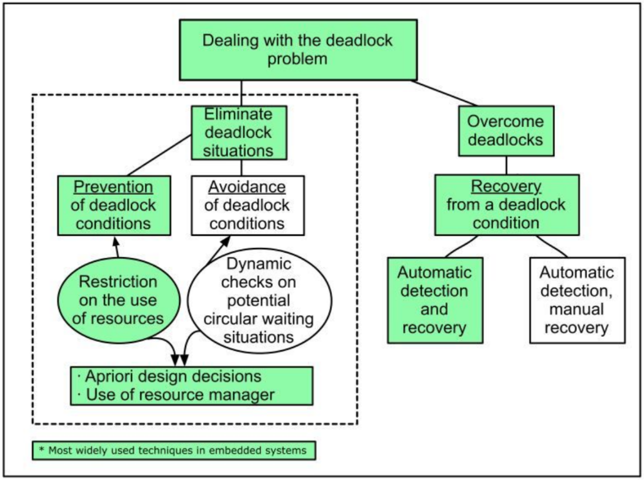


GIC

Resource Contention and Mitigation

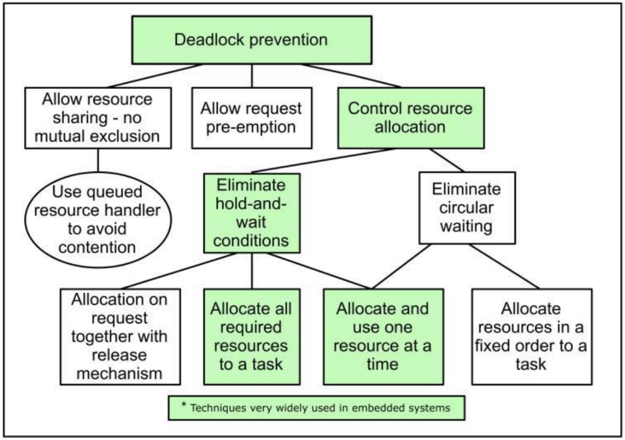
- Briefly looked at Priority Inversion and Inheritance a while back.
    - Will now generalize discussion to the “Deadlock” problem
        - Including illustration of *Circular Waiting* and *Fixed-Order Allocation*
- Will review Inversion/Inheritance with different pictures
- Will introduce another Inheritance method

What to do about Deadlock? High-Level Strategies


Dealing with the deadlock problem
Eliminate deadlock situations
Overcome deadlocks
Prevention of deadlock conditions
Avoidance of deadlock conditions
Recovery from a deadlock condition
Restriction on the use of resources
Dynamic checks on potential circular waiting situations
Automatic detection and recovery
Automatic detection, manual recovery
· Apriori design decisions
· Use of resource manager
* Most widely used techniques in embedded systems

From Textbook, Fig 4.9

What to do about Deadlock? Realistic Prevention Methods


Deadlock prevention
Allow resource sharing - no mutual exclusion
Allow request pre-emption
Control resource allocation
Use queued resource handler to avoid contention
Eliminate hold-and-wait conditions
Eliminate circular waiting
Allocation on request together with release mechanism
Allocate all required resources to a task
Allocate and use one resource at a time
Allocate resources in a fixed order to a task
* Techniques very widely used in embedded systems

From Textbook, Fig 4.10

Resource Allocation Order

↓Task : Resource→	Task A	Task B	Task C
4. ADC		Needed	Needed
3. DAC		Needed	Needed
2. Display	Needed		Needed
1. UART	Needed		

We can identify Tasks that can run concurrently, conflict-free by simply noting columns that have no common resource needs. Common use of resources implies contention.

If two or more tasks don't coordinate well, they can end up "circularly waiting": The one will be waiting for the other(s) to finish with resource(s) that the other will never finish with, because the other's waiting for the first to finish using the resource the first already took. . .

Ad Hoc Resource Allocation Order

↓Task : Resource→	Task A	Task B	Task C
4. ADC		Needed	Locked
3. DAC		Needed	Needed
2. Display	Needed		Needed
1. UART	Needed		

Ad Hoc Resource Allocation Order

↓Task : Resource→	Task A	Task B	Task C
4. ADC		Needed	Locked
3. DAC		Locked	Needed
2. Display	Needed		Needed
1. UART	Needed		

Will deadlock if (in either order):

- Task B tries to grab the ADC
- Task C tries to grab the DAC

before either of them successfully grabs their second resource of those two (which would allow it to finish).

Fixed Resource Allocation Order

↓Task : Resource→	Task A	Task B	Task C
4. ADC		Needed	Needed
3. DAC		Needed	Needed
2. Display	Needed		Needed
1. UART	Needed		

Works best on simpler systems, where resource requirements are statically known.

Fixed Resource Allocation Order

↓Task : Resource→	Task A	Task B	Task C
4. ADC		Needed	Locked
3. DAC		Needed	Needed
2. Display	Needed		Needed
1. UART	Needed		

At this point, TaskB will not take the DAC, because it would first try to take the ADC.

Fixed Resource Allocation Order

↓Task : Resource→	Task A	Task B	Task C
4. ADC		Needed	Locked
3. DAC		Needed	Locked
2. Display	Needed		Needed
1. UART	Needed		

Will deadlock result if TaskA takes the Display?

Fixed Resource Allocation Order (pri(A)>Pri(B))

↓Task : Resource→	Task A	Task B	Task C
4. ADC		Needed	Locked
3. DAC		Needed	Locked
2. Display	Locked		Needed
1. UART	Needed		

At this point, TaskC can want the Display, but if the want becomes a demand, the OS will sleep TaskC until it is available.

Fixed Resource Allocation Order (pri(A)$>$Pri(B))

$\downarrow$Task : Resource$\rightarrow$	Task A	Task B	Task C
4. ADC		Needed	Locked
3. DAC		Needed	Locked
2. Display	Locked		Needed
1. UART	Locked		

TaskA will be able to complete its work.

Fixed Resource Allocation Order

↓Task : Resource→	Task A	Task B	Task C
4. ADC		Needed	Locked
3. DAC		Needed	Locked
2. Display	Locked		Needed
1. UART	Released		

Each task then releases in reverse order that it allocated when it finishes with a resource.

"Released" is used here simply to help with the illustration of the sequence. The table really still has a "Needed" entry there–the resources just are not needed right NOW.

Fixed Resource Allocation Order

↓Task : Resource→	Task A	Task B	Task C
4. ADC		Needed	Locked
3. DAC		Needed	Locked
2. Display	Released		Needed
1. UART	Released		

Now what can proceed?

Fixed Resource Allocation Order

↓Task : Resource→	Task A	Task B	Task C
4. ADC		Needed	Locked
3. DAC		Needed	Locked
2. Display	Released		Locked
1. UART	Released		

TaskC can now use all of the resources that it needed, and complete its work.

Fixed Resource Order

↓Task : Resource→	Task A	Task B	Task C
4. ADC		Needed	Locked
3. DAC		Needed	Locked
2. Display	Released		Released
1. UART	Released		

TaskC now releases the shared resources.

Fixed Resource Order

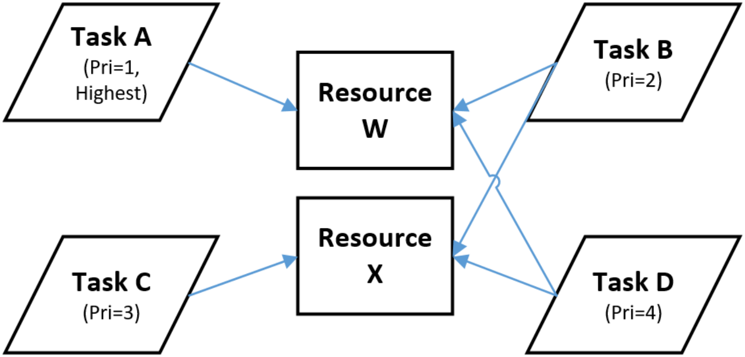
↓Task : Resource→	Task A	Task B	Task C
4. ADC		Needed	Locked
3. DAC		Needed	Released
2. Display	Released		Released
1. UART	Released		

Fixed Resource Order

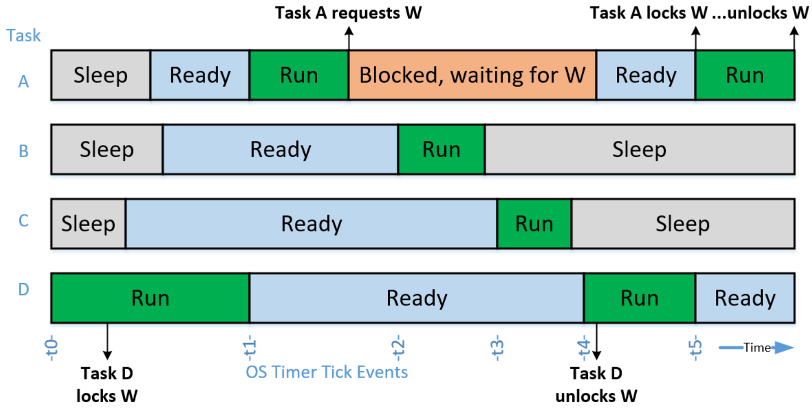
↓Task : Resource→	Task A	Task B	Task C
4. ADC		Needed	Released
3. DAC		Needed	Released
2. Display	Released		Released
1. UART	Released		

Deadlock:Avoided! Yay! Success?!?!

If we've completely removed deadlocks from the system through appropriate mutual exclusion and methods, odds are we have introduced the opportunity for another problem: Priority Inversion.

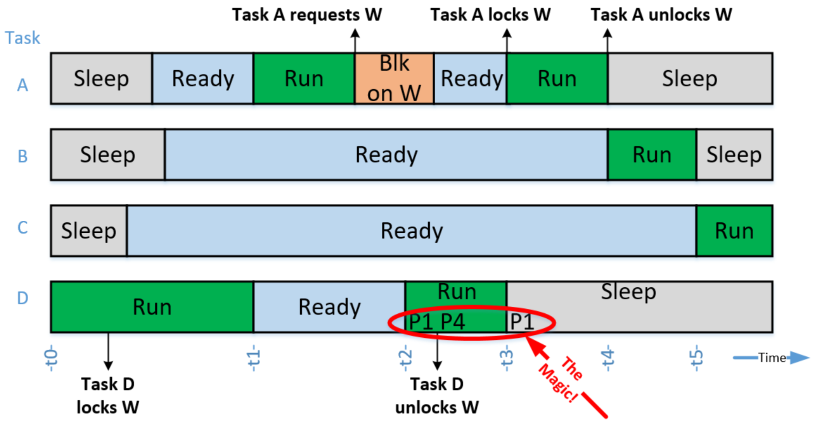


Priority Inversion (#2; the first was weeks ago)

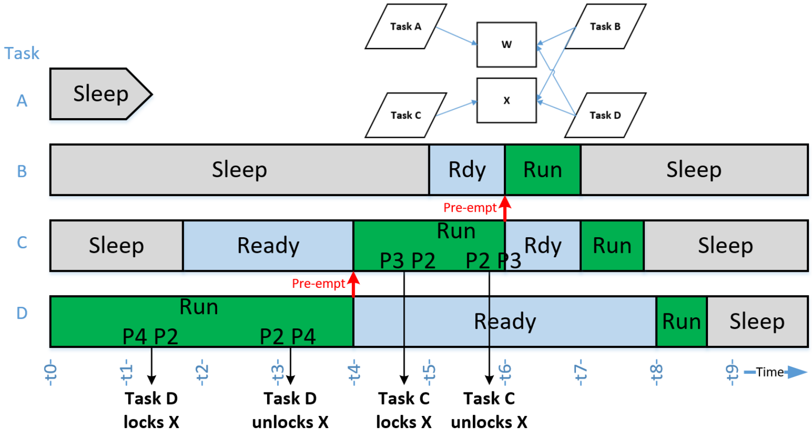


What are the 3 Task States, again? How do those map to the above? What do the sleep slivers (before t2, t3, t4) indicate about scheduling with this OS?

Priority Interitance (*Basic*, or Task-based)



Priority Inheritance, #3 (*Ceiling*, or Resource-based)



Mars Pathfinder Proves Inversion Is Real Problem

9 months after landing on Mars, lost communication due to repeating resets [1]

- Single RAD6000 processor (“Rad-hardened” RS6000)
  - $200k-300k each with similar compute to EFM32
- VxWorks RTOS (by Wind River Systems, with VERY high stability)
- Developers knew about Priority Inversion risks, and had planned for them
- 1992 Inception (Democrat Bill Clinton just elected) under mandate to shorten development time and costs
- 1993-1997 Development and Mission

[1] Not alien abduction as some early reports jokingly theorized, nor "doing too much" as less saavy reporters said.

Mars Pathfinder Priority Inversion

What Happened?

- Low priority task grabbed a semaphore and then was interrupted by mid-priority tasks long enough that the high-priority task (that also needed the semaphore) took too long
    - WATCHDOG! $\rightarrow$ RESET!
- Option for semMCreate() in selectLib() normally could be set to mitigate inversion, but was disabled in the port of VxWorks to the RS6000 architecture.

Risk: Untested Execution Streams Cause Problems

One set of problems to which multi-tasking solutions (and distributed systems, including Multi-CPU, Parallel-operation dedicated hardware-using, etc) are exposed is the general class of "timing window" risk in untested (and possibly unexpected) execution streams.[2]

Two contributors-of-interest to this type of risk are:

- Unrepeatable testing / variable workloads
- Insufficient inter-task communication
    - To protect correctly in all cases

[2] The "stream" term is used here simply to illustrate that the concern is not just a particular task's CPU code execution order-it's the intermix of all Tasks' instructions and use of other resources taken together

Unrepeatable Testing

Repeatable Testing:

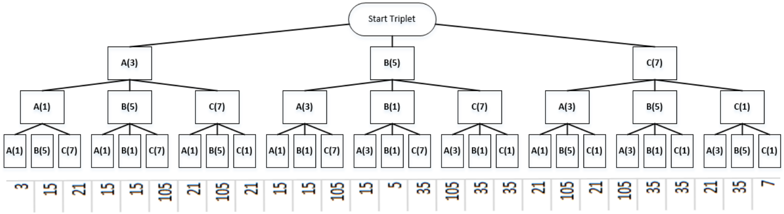
- Path analysis is only done for linear, single-task code.
- Unit Test is usually performed in single-task isolation, without interleaving with other tasks.

However, timing delays or differentially-variable loading on tasks may affect code behavior. Examples:

- Interrupt variability
- Control System Latency
- Concurrent processor (CPU, DMA, . . .) started before pre-emption

Why is it so hard to get execution predictability? (Simple)

Think through a 3-task, cooperative scheduling system, where each task has 3,5, and 7 sequential steps respectively which can independently yield or block.



There are over 1000 triplets!

For 1000 code (blocks) in each of 3 tasks, over 6e9 triplets!

But Won't The Tasks Stay Synched?

If you're thinking that it'll always execute like:

- (A1)(B4)(C9)(A2)(B5)(C10). . . (A3)(B8)(C15)
- and then repeat (every 3x5x7=105 3-tuples)

You might be correct without prioritized pre-emption (including interrupts), except that blocks/yields will eventually cause a task to skip a turn, and these will add up to total desynchronization.

In fact, the blocks/yields will tend to make things "clumpy", and to make some of the 1000 3-tuple patterns FAR more likely than others. Some will disappear from the possible set, too (e.g. if A3 always does sleep(1), you'll never see (A3)(A1) right next to each other).

Why is it so hard to get execution predictability? (Real?)

Still ignoring branching, and just considering the 7-long execution sequences that have 1 code (block) executed from 1000 each across N tasks, in numerical order, there are at least $(1e3)^7$ such sequences.

I'm pretty sure you can't test all of those[3], and if there is insufficient Inter-Task Communication (including shared resource protection), the occasional failure may be VERY hard to duplicate.

Should we perhaps critical-section protect every 1000 lines or so, to guarantee fewer tuples?

[3] If you think you can, consider asynchronous interrupts in the mix...

Why is it so hard to get execution predictability? (Real?)

Still ignoring branching, and just considering the 7-long execution sequences that have 1 code (block) executed from 1000 each across N tasks, in numerical order, there are at least $(1e3)^7$ such sequences.

I'm pretty sure you can't test all of those[3], and if there is insufficient Inter-Task Communication (including shared resource protection), the occasional failure may be VERY hard to duplicate.

Should we perhaps critical-section protect every 1000 lines or so, to guarantee fewer tuples?

Not likely. That would eat up a lot of the system margin.

Better: use ITC to appropriately decouple Tasks from each other, to allow them to be considered/tested in isolation.

[3]If you think you can, consider asynchronous interrupts in the mix...

What Did I Just Learn About Executing Chains?

In realistic systems, a reasonable number of tasks with reasonable interlocks will generate execution chains that are:

- Astronomically diverse
- Clumpy probabilistically

Pathfinder Risk Detectability/Repeatability

Speaking of Astronomical. . . why wasn't the priority inversion seen in earth-based testing over the prior years?

- Low priority task occasionally grabbed the semaphore and then was interrupted by mid-priority tasks long enough that the high-priority task took too long.

In fact, it was seen in flight. It was noted, and not recreated.

- Quality philosophy on US space projects had recently moved away from "it must work" philosophy (with attendant staffing), and the risk's estimated weighting was low enough that it was "ACCEPTED" as a risk at that point, with the more limited staff focusing on the landing phase of the program instead.

Chains of Risks: We're never done.

This risk exposure resulted from other risk ROAMing:

- Two other companies that could have previously provided an RTOS for the RAD6000 were tied up in merger.
  - Business risk MITIGATION: pick WindRiver, ask them to port VxWorks. (Specification was to NOT provide Semaphore Priority Inheritance, as a performance optimization.)
- Imperfect test coverage (limited time/testing available)
  - Was MITIGATED by intentionally relying heavily on "worst case" system loading under the belief that most of the problems were actually going to be there (under these conditions the system acted correctly).
  - The risk at lower-loading was ACCEPTED.
- Performance not known to be sufficient/excessive
  - MITIGATED by removing the selectLib() option in compliance with expected design

Inter-Task Communication is The Golden Key

The go-to power tool to get reliable, testable code is to separate tasks into nominally independent parts with Inter-Task Communication to ensure that they stay independent.

Since we are intentionally sharing resources in designs, we can only hope to **functionally** separate tasks from one another, by ensuring that they never operate out-of-order from what we design, with respect to any shared resources.

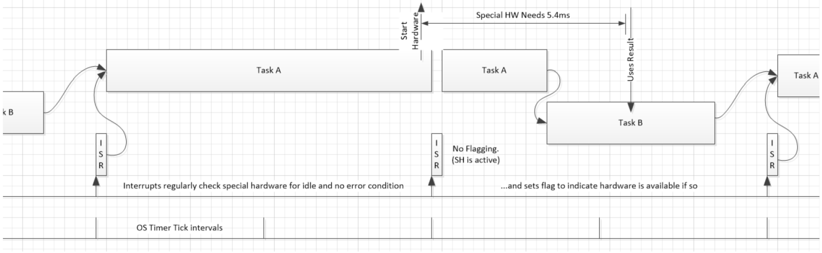
Example: New, Faster CPU Next Week…

Imagine a system with one IRQ, two tasks, and “Special Hardware”

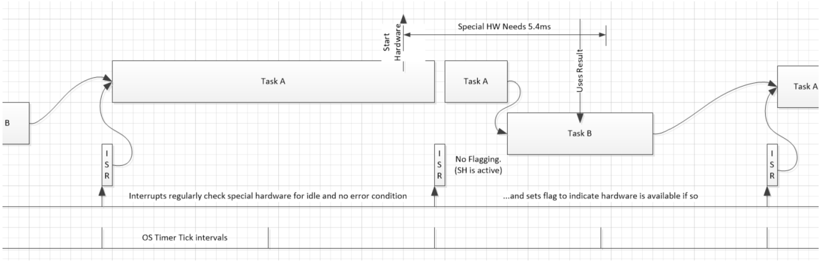
- TaskA needs 12.5ms, after 10m of that, starts Special HW (SH). TaskA activates when:
    - TaskB flags that it has consumed the prior SH results.
    - The IRQ Handler indicates that all-is-well to run the SH.
- TaskB collects the results from the SH halfway through its 5ms execution time, and then uses ITC to tell TaskA that it can run again, because B is no longer accessing the SH.
- IRQ handler checks the SH on timer-based intervals. If the hardware is both idle and error-free, a flag will be set that allows TaskA to restart the SH.

Pure Goodness, right?

Execution Sequence on Old Hardware


Start Hardware
Special HW Needs 5.4ms
Uses Result
Task A
Task A
Task A
k B
Task B
I S R
I S R
No Flagging. (SH is active)
I S R
Interrupts regularly check special hardware for idle and no error condition
...and sets flag to indicate hardware is available if so
OS Timer Tick intervals

Execution Sequence on New Hardware (5% faster CPU)


Start Hardware
Special HW Needs 5.4ms
Uses Result
Task A
Task A
Task A
B
Task B
I S R
I S R
I S R
No Flagging. (SH is active)
Interrupts regularly check special hardware for idle and no error condition
...and sets flag to indicate hardware is available if so
OS Timer Tick intervals

Where was the ITC skipped in this design?

Scheduling Guarantees

Can we **guarantee** that we'll meet our deadlines?

Scheduling Guarantees

Can we **guarantee** that we'll meet our deadlines?

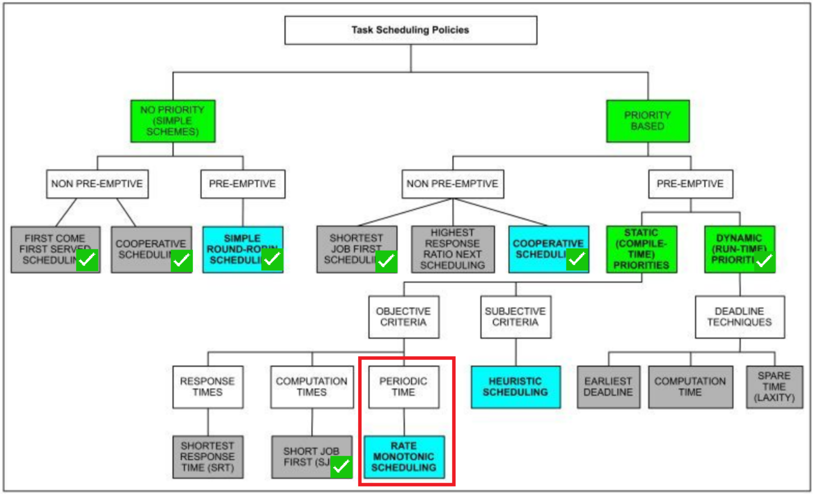
Deterministic/Predictable execution is hard to come by.

- Deadlock avoidance techniques in dynamic priority schemes add more possibilities for execution variation
- Inversion avoidance adds more

Static prioritization methods optimize on a particular **objective** criteria, which can make analysis easier:

- Shortest Response Time [**Start Time**]
- Shortest Job First [**Completion Time**]
- Rate Monotonic Scheduling [**Periodic Time**]

Task Scheduling Policies, revisited



Rate Monotonic Scheduling

Also known as “Rate Monotonic Analysis”, this technique is often well suited to systems performing periodic work only, with no resource contention, where the deadline is all that matters about the period. Some of the strong arguments for it:

- Can provably assure that you’ll meet your deadlines[4]
- Is a static prioritization scheme-easy to understand
- Simpler scheduling than Earliest-Deadline-First, which sometimes provides better RT support
    - RMA can reach 100% CPU utilization for some workloads, but asymptotically can only guarantee deadlines if $<$69% utilization
    - EDF can reach 100%. The tradeoff is that it requires run-time mods to priorities based on next deadline.

[4] As long as there is enough CPU horsepower!

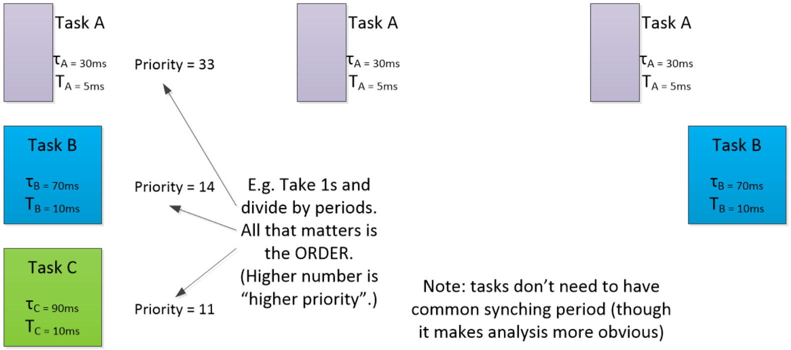
RMA: What IS It?

Fundamentally, the idea is to allow the task with the shortest periodic execution time to be the highest priority when it is ready to run.

- Minimal Task Switching Time-**Wishful Thinking!** What can we do to inject realism?
- Deadlines at end of period, τ
- Activation at start of period (e.g. interrupt for each period's start readies a task)
- Shortest period task that is ready runs. (pre-emptive, prioritized)
  - Essentially, priority $= 1/\tau$.

RMA: Worst-Case Moment

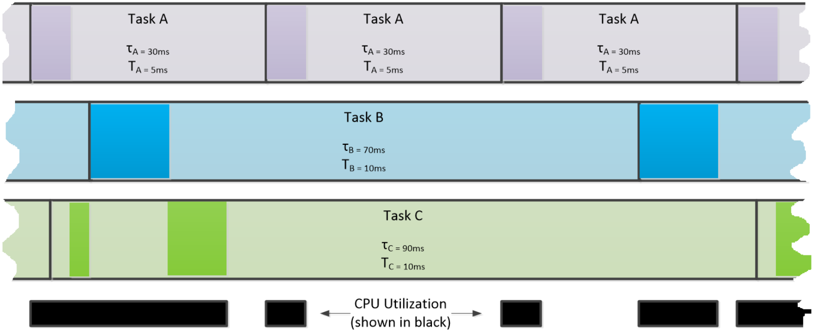
The “Critical Moment”: as bad as *readying* can possibly get.


Task A
τA = 30ms
TA = 5ms
Task B
τB = 70ms
TB = 10ms
Task C
τC = 90ms
TC = 10ms
Priority = 33
Priority = 14
Priority = 11
E.g. Take 1s and divide by periods. All that matters is the ORDER. (Higher number is “higher priority”.)
Task A
τA = 30ms
TA = 5ms
Task A
τA = 30ms
TA = 5ms
Task B
τB = 70ms
TB = 10ms
Note: tasks don’t need to have common synching period (though it makes analysis more obvious)

- The priorities need to be consistent with the OS’s range. Micrium supports INT_8U in OSTaskChangePrio(), and only the idle task can be “1”.
- With RMA, CPU usage above 69% in the asymptotic case is essentially needed to deal with the Critical Moment.

RMA: Example 1

Starting out mid-stream (NOT Critical Moment), already desynchronized:



Name(τ,T): A(30,5),B(70,10),C(90,10) $\rightarrow$ 16.7%,14.2%,11.1% $\rightarrow$ 42% CPU Utilization

RMA: Example 2

Class Exercise @ Critical Moment:

- Task A: τ=0.20s, T=0.05s
- Task B: τ=0.5s, T=0.10s
- Task C: τ=1.0s, T=0.15s
- Task D: τ=0.25s, T=0.05s

Use GCD(T[i]) for analysis time interval. *Keep this in mind to revisit the question about a safe number of task switches upon which to assume the "tax" of a Task Switch time, to have a more realistic analysis.*

RMA: Example 2

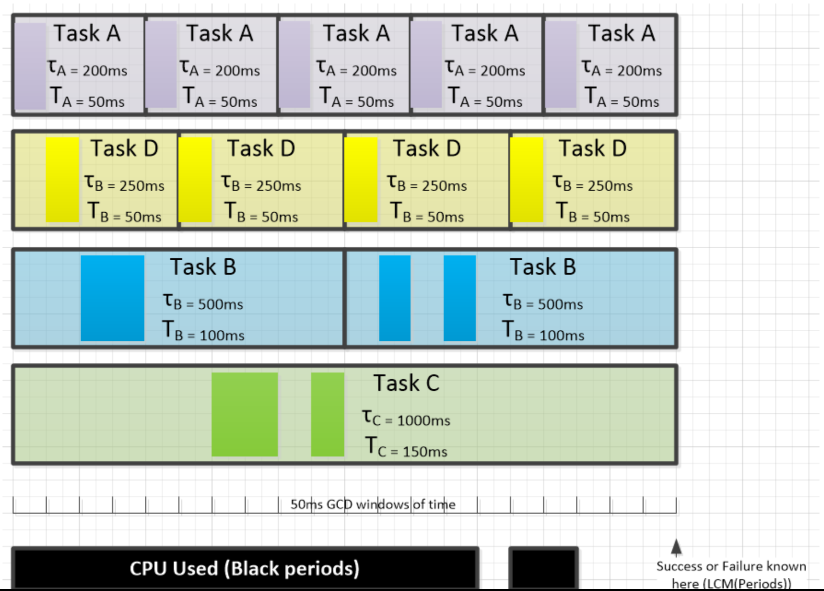
Calculate CPU duty cycles:

- Task A: τ=0.20s, T=0.05s ! %Util = 25%
- Task B: τ=0.5s, T=0.10s ! %Util = 20%
- Task C: τ=1.0s, T=0.15s ! %Util = 15%
- Task D: τ=0.25s, T=0.05s ! %Util = 20%

Total CPU Utilization expected: 80%

Don't go on to next page until example has been worked!

RMA: Example 2, Solution



RMA: Example 2, Comments

- The use of a GCD was simply to allow quanta-based counting of CPU execution. This allowed us to fill out “time slots” from the highest priority task downward, knowing that what remained of CPU execution for tasks that ran would be a simple counting exercise.
- With unrelated periods, one can still draw out repeating periods from left to right for each task from the Critical Moment, and evaluate from the top priority task downward when each would begin executing. The computation of remaining execution time will need to be evaluated at the each context switch point, though, and will not be a simple counting exercise.
- Proof of sufficient CPU may also be more tedious, since LCM(periods) may be essentially the product of all of the periods, if they do not have a GCD bigger than a “clock tick”.

So, Always Use RMA?

Always, **only if** you can guarantee:

- <69% CPU needed
- Only Periodic Tasks
- Jitter within period is fine (counter-example: Control systems!)
- NO resource contention (hmmm. . .)

But **often** it can provide an excellent starting place for Heuristic Scheduling, where experience allows the subjective balancing of deadlines and other factors in the cases where it truly critical.