

ECEN 3753: Real-Time Operating Systems

Introduction and Overview



whoami

Course Goals and Expectations

- See Syllabus (separate)
- Come to lecture to get the theory
- Read the pre-reading to enhance lectured content
- Do the lab assignment. If it's not making sense:
 - Make sure you've done the relevant pre-reading, as it may help fill a gap
 - THEN come to office hours, or seek a classmate's help to understand better.
- Occasionally, post-lecture links will be provided for further reading. The intention here is to enrich your education, not to add more content for which you'd be tested.



Attribution

This course's presentation material has been created for the University of Colorado, Boulder, with extensive support in the form of materials (slide content and ideas) from both our selected textbook (Real-Time Operating Systems, Book 1: The Theory, by Jim Cooling), and training materials provided by Silicon Labs.

Additionally, Silicon Labs has provided SDKs for the coursework developers. It is the hope of them and me that students gain an appreciation for being able to use a quality commercial tool for debug.



Appreciation

In addition to Silicon Lab's support of this class's development, a group of engineers at Western Digital volunteered to jointly develop most of the lecture and lab material for this course.

Adam Harper	Biping Wu	Chad Hahnlen
Ed Hoskins	Isaac Muse	Steve Love



This Week's Lab

- If you have not already done ECEN3360.Lab1 or are not comfortable with the SDK, this is your chance. If you already have and are, this'll be a virtual vacation of a lab week.
- You should have acquired the Pearl Gecko development board already
- Get ECEN3360.Lab1 from our git repo and follow the instructions therein (SiLabs account, Simplicity IDE, Segger SystemView...)



What You're Doing With This Lab

In general in this class's future labs, RTOS and BSP functions will help you to more easily do what you did in a “bare metal” manner previously.

Explore the tools

- git
- Boot (what level of analysis/tracing can you reasonably do?)
- Power Profiler
- Debug (single-step software breakpoints)
- Segger System View



Slide Format

Continuity with ECEN 3360:

- Information
 - *Example or Narrative to illustrate a point*
- Discussion point or Question to answer
- Don't miss these!



Baseline question for this semester

Shall silence be YES or NO?

Credit to Prof. Adam Norris, who made enough of an impact with this that I heard about it through a student



Reading Assignments

Value proposition:

- Will benefit your understanding/context of week's topics
- Required. (Some is not until later, but...)



Recurring Theme: Use of Real-life Analogies

How we start, run, and debug computing environments has parallels to our carbon-based life.

Some of our first topics:

- Bootstrapping
- Scheduling and Execution
- Debugging



Carbon-based Booting

<https://en.wikipedia.org/wiki/Bootstrapping>



Carbon-based Booting

<https://en.wikipedia.org/wiki/Bootstrapping>

How did you start your day? (Provide some invariant parts of your routine)



Carbon-based Booting

<https://en.wikipedia.org/wiki/Bootstrapping>

How did you start your day? (Provide some invariant parts of your routine)

- Woke up
- Got out of bed
- Put on selection of clothing
- Put on footwear
- Recalled your planning system
- Compared day and time against your system to determine first activity to perform
- Transported to first activity



Human-scale Scheduling and Execution

Review the plan and priorities, and start executing to the plan.

Plan/priorities:

- *I have to be at class at 10A*
- *I want to go climbing before the air gets hot*
- *It's 8AM, so I'm going to go climbing first.*

Execution:

- *Drive to climbing site*
- *Climb, enjoy the view and some snacks, climb back down, gather gear*
- *Drive back to campus*



Human-scale Scheduling and Execution

Review the plan and priorities, and start executing to the plan.

Plan/priorities:

- *I have to be at class at 10A*
- *I want to go climbing before the air gets hot*
- *It's 8AM, so I'm going to go climbing first.*

Execution:

- *Drive to climbing site*
- *Climb, enjoy the view and some snacks, climb back down, gather gear*
- *Drive back to campus*
- ***Arrive at class at 10:15A.***



Human-scale Logging/Debugging

If/**when** things go off the rails and there's a train wreck before your eyes, think back to what happened BEFORE...

- *I didn't absorb what I needed to in class since I arrived late. Maybe going to go climbing at 8A was a poor plan, because there wasn't enough time to get back to campus afterwards.*

In the “Pre-Reading” video this week:

- Was the moment that we observed the derailment very relevant to learning why the train derailed?
- What early indicators were there?
- What other “debugging” info would the NTSB wish they had?
- Would it be worthwhile to plaster cameras and other recording equipment over every face of every car on a train? Why or why not?



In Memoriam: Grace Hopper (1906-1992)

1100 Started Cosine Tape (^{with changes} Sine check)
1525 Started Multi Adder Test.

1545



Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

~~1630~~ 1630 Arctangent started.

1700 closed down.



Bring these 3 topics into the Embedded RT space

- Booting
- Scheduling/Execution
- Logging/Debugging

. . . over this course.



What is an OS?

Software Providing:

- Encapsulated/standardized access to system resources
 - Memory
 - CPU
 - Storage (FileSystems, e.g. on HDD/SSD) & Networking
 - Boot
- Means to exclusively-use some resources
 - Virtual Memory
 - Abstract Processors (and scheduling algorithms to share real processors)
- Communication methods between requestors for sharing work and resources
- Security between data users when sharing is not appropriate
- Board Support Package (BSP) is not technically part of, but is usually provided with a RTOS, to encapsulate the hardware available in a particular embedded system

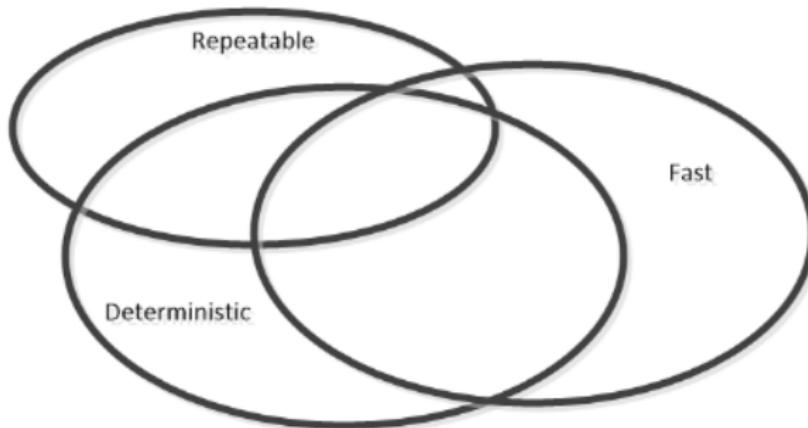


Generalized OS Boot Sequence – before main()

- Processor fetches code from ROM at boot address
 - May be slow
 - Interrupts are off, Exception table in ROM
- Initialize GPIO
 - Minimalist User Interface (UI) configured for reporting catastrophic boot errors before “full” UI (LEDs, beeps)
- Adjust clocks
 - Sometimes transitioning to faster execution
- Initialize / test memories
- Load any more code into RAM (from other places, possibly requiring additional initialization. . .)
 - Decompress if needed
 - Under what constraints would compression make sense?
 - Run from RAM



Is a “Real-Time” system the same as “Fast”?

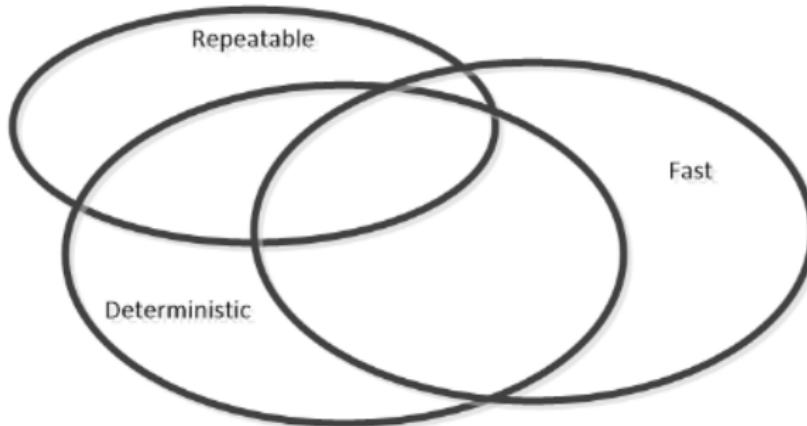


“Real-Time” means that a system is **deterministic**. That is, it will consistently meet all of the required time limits.

Is (Fast and Deterministic) better than (Slow and Deterministic)?



Is a “Real-Time” system the same as “Fast”?



“Real-Time” means that a system is **deterministic**. That is, it will consistently meet all of the required time limits.

Is (Fast and Deterministic) better than (Slow and Deterministic)?

- It Depends.

What is a Non-RT OS missing?

- Time Control (Deadlines, determinate behavior)
- Tightly-Coupled Coordination of Work or Shared Resources
- Constrained Scale (more costly resources and support systems are often required)

OS	RTOS	WinCE	Linux (server)	MSWin
Response	D, Fast	(N)D,Fast	ND, Very Fast	ND, *
Footprint	Tiny	Small	Small-Medium	Large



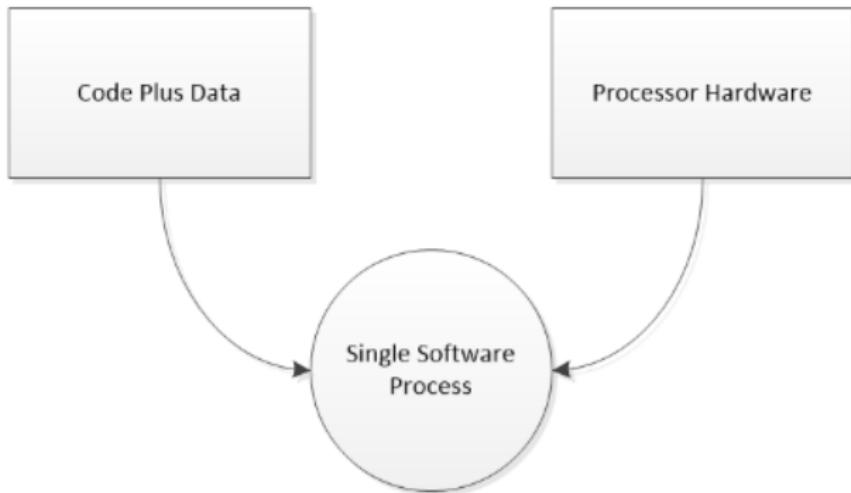
Communicating an Embedded, Real-Time Framework

We've got real (time) problems to solve...

How are we going to describe the solutions?



Single Software Process

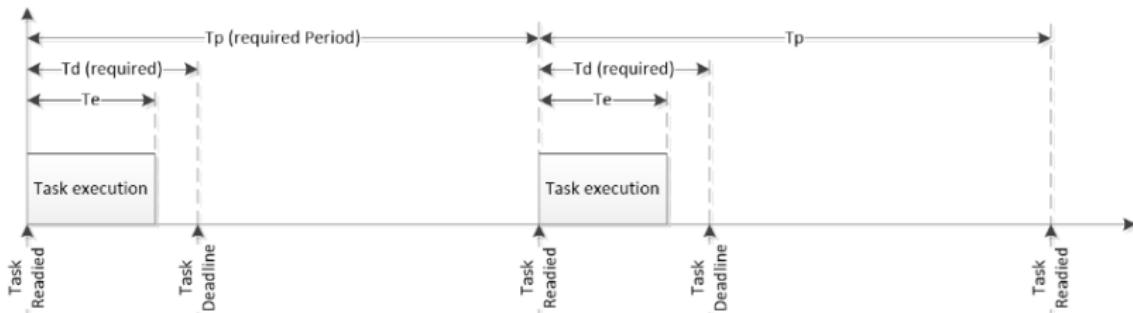


A **Program** is the non-volatile storage of one or more processes' descriptions that will lead to **Process(es)** when it is brought into a real computing environment on physical processor(s), and set up properly.



Process components

- Multiple, separable computational efforts can be treated separately to allow simpler, more robust solutions to be applied to each
- Some parts will require periodic work, some will likely be purely reactive, non-periodic work
- If sufficient processing resources are available, each task can effectively run at a periodicity that's needed (T_p), and with reasonable fulfilment within a deadline (T_d).
- Responsiveness: $(T_e > T_d) = \text{Missed Time Limit!}$



Classification of Process Components

A Process can be broken down into a plurality of one of these lighter-weight constructs that share the process memory, and some resources.

- Threads are lightweight processes, aligned to separable flows
 - Can be used to help group work in parallel computing or asynchronous programming
- Tasks are lightweight process components, aligned to abstract work that must be done
 - May be used in a degenerate case for synchronous programming

Stack is a non-shared resource for these components.



Types of RealTime Systems

Classify by the impact if time limits are not met:

- **Hard RT**: If the time limit is not met, damage (sometimes catastrophic) can result
- **Firm RT**: An occasional missed time limit is still a bad thing, and could cause damage or out-of-design-goal performance if left unmitigated. Mitigation is implemented when this is a known possibility.
- **Soft RT**: Missed time limits will degrade performance, but in a non-damaging manner
- **Non-RT**: *Time limits? What time limits?*



Classify RT Systems

What RT type do you think these are, and why?

(1:Hard, 2:Firm, 3:Soft, 4:Non)

- National Power Grid Protection Relay
- Refrigerator compressor control
- Microwave Oven Door Monitoring
- Solar Panel Grid-Tied Power Inverter
- Core Network Router
- Airplane Stall Mitigation
- Bitcoin mining



Approximately:

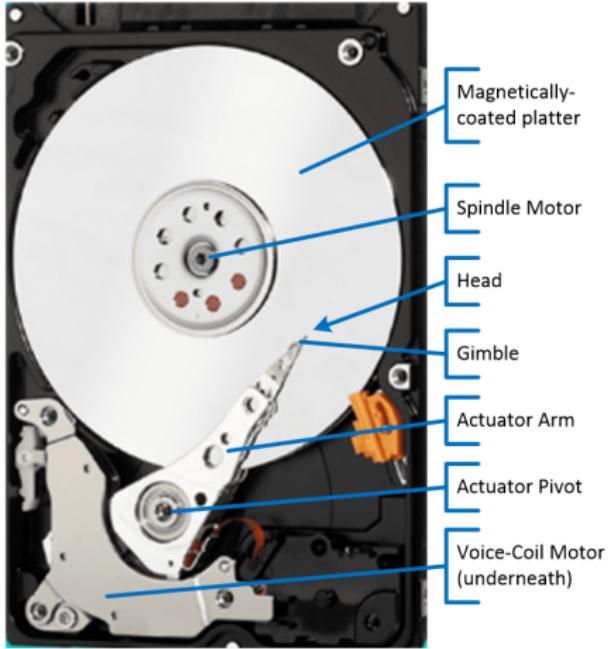
What RT type do you think these are, and why?

(1:Hard, 2:Firm, 3:Soft, 4:Non)

- National Power Grid Protection Relay (Hard)
- Refrigerator compressor control (Soft)
- Microwave Oven Door Monitoring (Hard*)
- Solar Panel Grid-Tied Power Inverter (Hard)
- Core Network Router (Firm)
- Airplane Stall Mitigation (Firm*)
- Bitcoin mining (Non)



Classify an HDD Storage System



Head: Radial Velocity $\text{o}(100 \text{ in/s})$ max.
Servo position: sampled $\text{o}(10\text{KHz})$

What kind of Real-time classification is the position-sampling system and its feedback control?

Angst!

Everything costs money

- ADC/DAC
- Processors
- Memory
- Power

Solution?



Angst!

Everything costs money

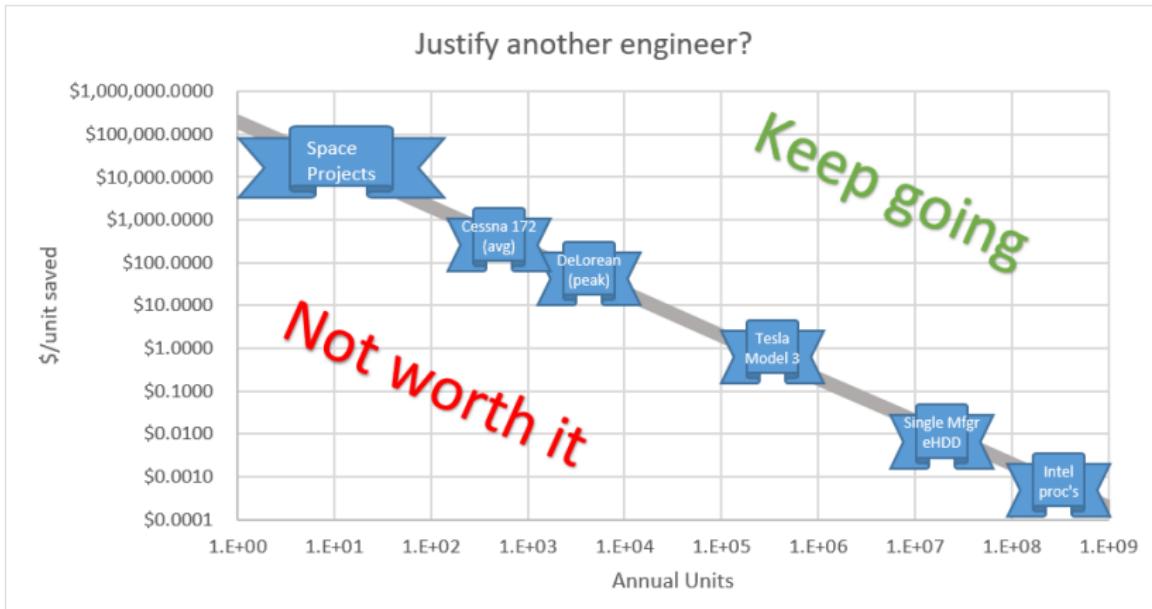
- ADC/DAC
- Processors
- Memory
- Power

Solution?

- Share resources.

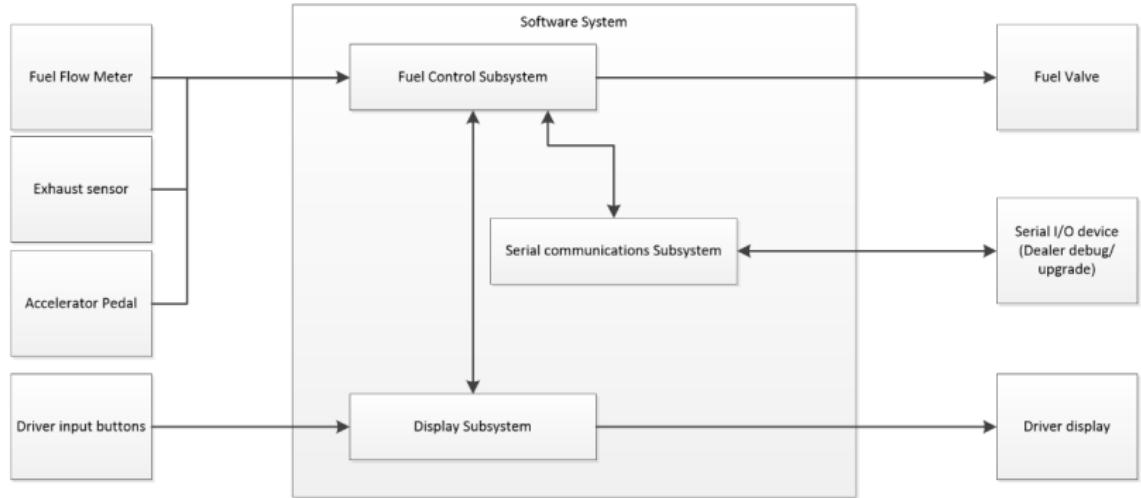


Cost Savings, at Scale



If you're going to make enough of something, extra engineering to reduce per-item costs can be compelling.

Resource Sharing – Software Design Model



Super-loop Construction

Using the text's simple "Super-Loop" method of doing each task to completion, we could code this system as:

```
{  
    Measure Fuel Flow  
    Measure Exhaust Characteristic  
    Measure Accelerator Position  
    Compute New Fuel Valve Setting  
    If (Incoming Data on Diagnostic Port)  
        Handle Diagnostic Command  
    If (User has pressed a dashboard button)  
        Handle User Command  
        Update Display  
}
```



Super-loops Incur Maintenance

```
{  
    Measure Fuel Flow  
    Measure Exhaust Characteristic  
    Measure Accelerator Position  
    Compute New Fuel Valve Setting  
    If (Incoming Data on Diagnostic Port)  
        Handle Diagnostic Command  
    If (User has pressed a dashboard button)  
        if (Part == 0)  
            Handle User Command  
        Update Display (Part = (Part++ % NUM_PARTS)) // 0..N-1  
}
```

Simple enhancement: break display update into roughly-similar-sized parts at convenient points.

And introduce bugs by trying to plan ahead...



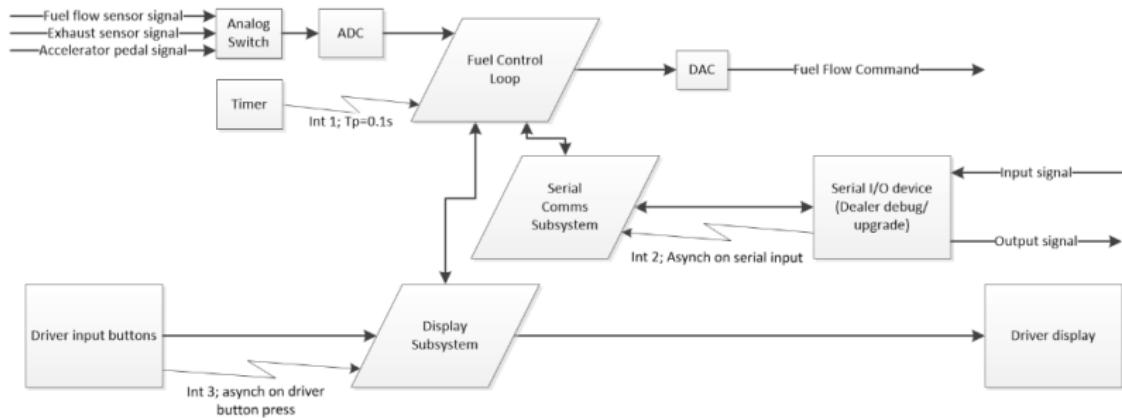
Super-loop with Yield()-ing

```
{  
    Measure Fuel Flow (till yield)  
    Measure Exhaust Characteristic (till yield)  
    Measure Accelerator Position (till yield)  
    Compute New Fuel Valve Setting (till yield)  
    If (Incoming Data on Diagnostic Port)  
        Handle Diagnostic Command (till yield)  
    If (User has pressed a dashboard button)  
        Handle User Command (till yield)  
        Update Dashboard (till yield)  
    Measure Fuel Flow (till yield--immediate if 1x)  
    Compute New Fuel Valve Setting (till yield--immed if 1x)  
}
```

Operating Systems can make this a bit simpler to program by allowing a task to yield() to the next (AKA sleep(0)).



Interrupt-driven Quasi-Concurrency



As long as there is sufficient processing capability in the shared physical processor, and there are no contentions for shared resources that interfere, could share a processor by reacting to interrupts for asynchronous events, and timers for periodic processing.

What would happen if one of the asynch $T_e > 0.1s$?

More that happens before main()...

- Compiler-directed initialization
 - Static construction / zero initialization
 - .text, .data, .bss
- Initialize/enable interrupts
- Additional HW initialization for the OS
- OS initialization
 - Tick timer
 - Idle task



What happens AFTER main()?

- Compiler related stuff
 - Static destruction
- Exit/terminate handling
 - Non-RT: return to prompt, close window, power-off, etc.
 - RT: There are no good choices. **Recall: for Hard-RT, "physical damage will result"**
 - Reboot
 - Hang
 - Crash / bluescreen



What happens AFTER main()?

- Compiler related stuff
 - Static destruction
- Exit/terminate handling
 - Non-RT: return to prompt, close window, power-off, etc.
 - RT: There are no good choices. **Recall: for Hard-RT, "physical damage will result"**
 - Reboot
 - Hang
 - Crash / bluescreen

A moving car with computer-controlled braking can become...



What happens AFTER main()?

- Compiler related stuff
 - Static destruction
- Exit/terminate handling
 - Non-RT: return to prompt, close window, power-off, etc.
 - RT: There are no good choices. **Recall: for Hard-RT, "physical damage will result"**
 - Reboot
 - Hang
 - Crash / bluescreen

A moving car with computer-controlled braking can become...

A stationary car with brakes latched which is now safe to...



What happens AFTER main()?

- Compiler related stuff
 - Static destruction
- Exit/terminate handling
 - Non-RT: return to prompt, close window, power-off, etc.
 - RT: There are no good choices. **Recall: for Hard-RT, "physical damage will result"**
 - Reboot
 - Hang
 - Crash / bluescreen

A moving car with computer-controlled braking can become...

A stationary car with brakes latched which is now safe to...

Power off (including exiting the brake-control task).



Value of Frameworks

What are FRAMEWORKS for?



Value of Frameworks

What are FRAMEWORKS for?

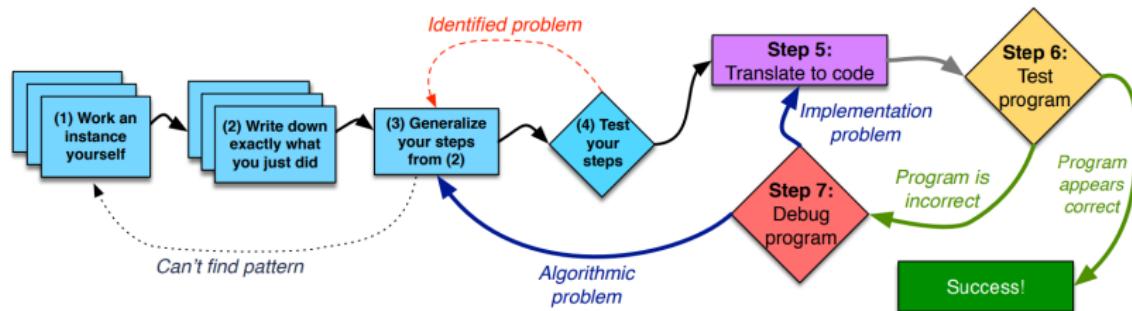
- *To guide inexperienced people to stay within bounds known to work?*
- *To ALWAYS force solutions to stay within the framework?*
- *To help provide a starting point for deconstruction of complexity? (... that has proven useful in the past, or by experts looking for how to help in the future)*

absolutes are rarely correct.



Very Useful Problem-Solving Method

This 7-Step Method was generated by Andrew Hilton, Genevieve Lipp, and Susan Rodger at Duke University.



Load/Store a redundant boot-time data set

<image from <http://adhilton.pratt.duke.edu/sites/adhilton.pratt.duke.edu/files/u37/itcse-7steps.pdf>>



GPIO, Interrupts, and Timers

Your connections to the real world, with a sense of time—

The enablers for an Embedded, Real-Time system!

We animals are amazing embedded *biological* systems!



GPIO, Interrupts, and Timers

Your connections to the real world, with a sense of time—

The enablers for an Embedded, Real-Time system!

We animals are amazing embedded *biological* systems!

In our carbon-based analogue, if we didn't have any of our 5 senses (Inputs), and the ability to act upon our environment (Outputs) with an awareness of time, we would be as unfit-for-use as an old PC on a dusty workbench, powered up and clocking a CPU, but without any useful effect (other than as a space-heater)!



General Purpose Input/Output (GPIO)

- Enabling
- Activity level (high/low)
- Electrical configuration
- Routing configuration

For our embedded systems' senses and muscles to be effective, we need to wire things up the right way...



General Purpose Input/Output (GPIO)

- Enabling
- Activity level (high/low)
- Electrical configuration
- Routing configuration

For our embedded systems' senses and muscles to be effective, we need to wire things up the right way...

- Choosing which inputs and outputs to use
- Setting them up to use/expect the right polarities
- ... with appropriate sensitivities/strengths
- Using the right neural pathways for the right activities

We're not yet talking about priorities or time sensitivity... but we'll get there soon.



General Purpose Input Electrical Configuration

Options:

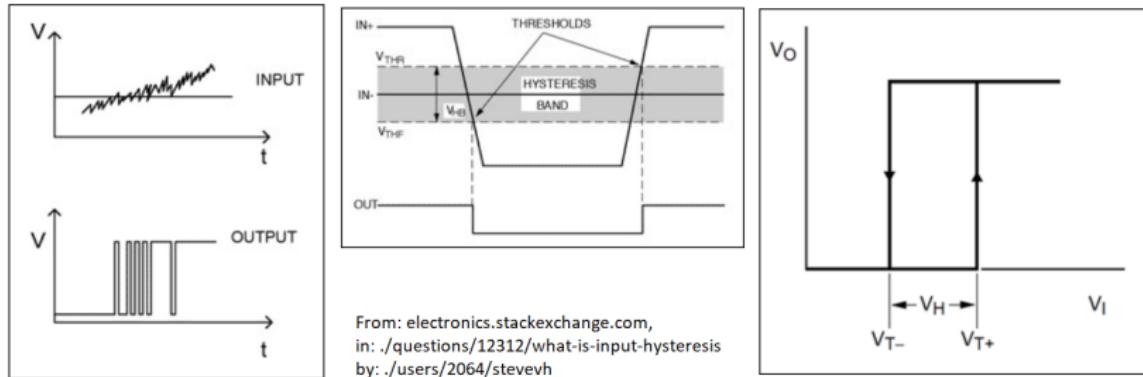
- Internal Pull-up resistor
- Internal Pull-down resistor
- High-impedance (no internal resistor)

Inputs are generally internally wired with hysteresis.

Hysteresis is helpful on inputs, to keep our real-time embedded system from being pestered by the equivalent of a flickering light bulb.



Input Hysteresis



- What you DON'T want to happen with a noisy input is shown on the left,
- A plot of behavior we DO want (with a clean input) is in the center.
- Alternatively you can view this as a transfer function, on the right.

General Purpose Output Electrical Configuration

Options:

- Push/Pull (MOSFET above and below, can actively source or sink current)
- Open Drain(FET)/Collector(BJT): can sink current. *Wired-OR*
 - With internal resistor (**often insufficient**)
 - Without internal resistor (**What's needed?**)
- “Speed”, “Slew-rate”, “Frequency”, or a “current-limit” setting
 - Helps drive the output *transition* more aggressively
 - May generate more noise
 - May use more power **Lab1: Did it significantly? Reason?**

When setting up the “muscles” of our embedded system, we need to configure the electrical properties to ensure that we can drive the eventual states the way we need. There is often an option about whether to put the muscles on steroids—if we do, there may be side effects.



GPIO Routing

On-chip and off-chip (de)multiplexors and analog switches allow general designs of chips to be useful in more solutions.

Designs likely need to optimize across several factors:

- Switching I to/from O requires all of the rest of the path to be resilient to this
 - Can you think of any I/O usage that switches back-and-forth?
- Keeping settings static (current/speed, multiplexors) especially on high-usage paths
- Switching settings more on slower-sampled signals may be acceptable
- Desire for dedicated pins (debug, board connects, PCB optimizations)



GPIO Routing

On-chip and off-chip (de)multiplexors and analog switches allow general designs of chips to be useful in more solutions.

Designs likely need to optimize across several factors:

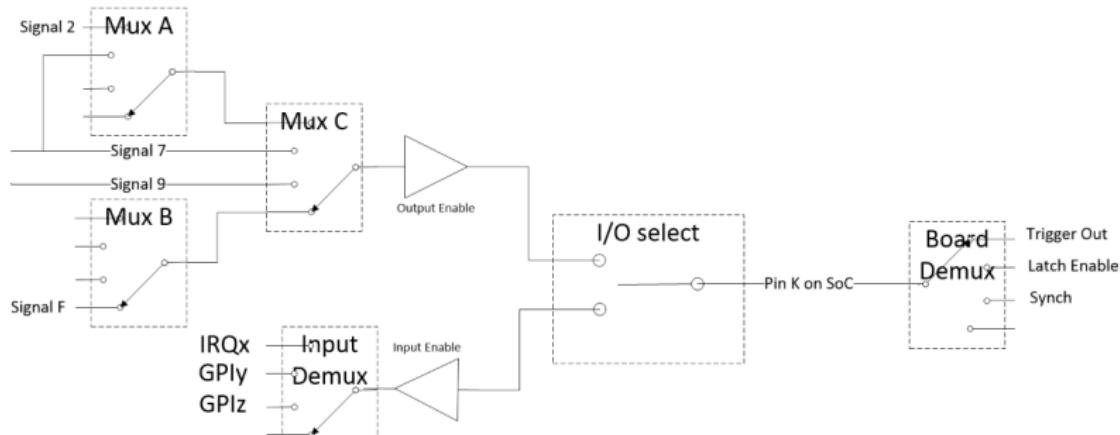
- Switching I to/from O requires all of the rest of the path to be resilient to this
 - Can you think of any I/O usage that switches back-and-forth?
- Keeping settings static (current/speed, multiplexors) especially on high-usage paths
- Switching settings more on slower-sampled signals may be acceptable
- Desire for dedicated pins (debug, board connects, PCB optimizations)

I2C : What I/O flipping does SerialDAta enjoy?



GPIO Routing example

- Signal 9 is needed for Synch:
 - MuxC=10b; I/O:O; BoardDemux:10b
- Signal 7 needs to route to TriggerOut (2 ways!):
 - MuxC:01b; I/O:0; BoardDemux:00b
 - MuxA:01b; MuxC:00b; I/O:O; BoardDemux:00b
- More complex when multiple pins are available, with **different routing options**. Why would these exist?



Interrupts

Historically, an “interrupt” was an external event requiring action. However, terminology is now ambiguous. (Internally-generated exceptions are sometimes called interrupts too (SWI, NMI), and the M4 manual refers only to “exception handlers”.)

- Setup required in advance:
 - Configuration of Interrupt Controller (incl. ISR location setup)
 - Enablement of IRQ at CPU (sometimes an additional step)
 - Level/latched-edge configuration
- Required at time-of-interrupt:
 - *Shared resources are needed for the interrupt to be serviced.*
 - *ISR code is at a known location.*
 - *Data may be pre-fetched from known locations for ISR operation.*

What Shared resources are needed?



What Needs to Happen when Execution is Interrupted?

The non-ISR code execution **context** must be saved so that the non-ISR work can pick up where it left off, later.

- What, Where, and How much?
 - In addition to the CPU itself, CPU registers (including the CPSR, PC, LR) are shared resources, but the values in these registers at any point in time are specific only to the currently running code, not JOINTLY between the interrupted code and ISR. So if the ISR is going to use any particular registers, their pre-interrupt content must be saved at start-of-ISR, and restored at end-of-ISR.
 - The stack seems like a reasonable default location to store the registers' prior contents...
 - Why is the amount not straightforward?



Why the sizeof(context) isn't obvious

- Simpler encapsulation by providing more freedom to use the full register set with abandon, but
- Requires more:
 - time, if using stack
 - hardware gates, if using dedicated register banking

In custom-designed hardware, banking the full register set may not be cost-prohibitive. Example (c.2001) was a \$1kUSD core internet router chip that was designed to fully pre-fetch an execution context for any one of hundreds-of-thousands of tasks and then to flush it back out after only getting about 40 cycles of the CPU in which to execute routing logic. Clearly, this was not a "general purpose" embedded controller—the cost was well-justified for its market.



Minimalist Interrupt Handling Hardware

Restating, when an interrupt is enabled and arrives with sufficient priority, at a minimum the hardware needs to:

- push register content that would otherwise be in the way (CPSR, Link Register at least) onto the stack
- copy the PC that was in use to the Link Register
- set the PC appropriately to handle the interrupt (start executing the ISR)

When we discuss shared resources and task switching, this same concept of the **context** will apply, as will the optimizing concerns about saving-and-restoring the “right amount”.



ARM Interrupt controllers

The Generic Interrupt Controller (for the Cyclone V SoC with a pair of A9 cores) or the Nested Vectored Interrupt Controller (for our M4) provide support for:

- Edge/Pulse interrupt latching (**Synchronous**)
- Level interrupt handling
- Prioritization
 - Programmable with groups [GIC and NVIC]
 - From humble origins c.1976: Fixed / Rotating [Intel 8259]
- Processor/Operating Modes (select: Stack & HW accel.)
 - Thread/Handler [NVIC]
 - System/Supervisor/User/IRQ/FIQ/Abort/Undefined [GIC]
- Routing/“distribution” (to one or more of the cores) [GIC]

We will circle back to these in a few weeks, to look at ways in which hardware support can shorten latencies and control priorities for interrupts to make our system more responsive for real-time apps.



Interrupts: almost as good as another CPU?

Reflecting—an interrupt essentially:

- Is giving you the same benefit as a dedicated polling CPU that then hands off the “real work” to the main CPU (and ISR code) once “something is ready to be done”.
- Is very cheap/easy to do with concurrent hardware, rather than CPU+PollingCode.

Rather than a super-loop’s polling code executing as frequently as the loop allows, the pseudo-polling “code” is ALWAYS running in interrupt hardware, until the Interrupt happens.

When might CPU cycles make sense instead?

- on the main CPU
- on a baby core (e.g. M0)



Interrupts: almost as good as another CPU?

Reflecting—an interrupt essentially:

- Is giving you the same benefit as a dedicated polling CPU that then hands off the “real work” to the main CPU (and ISR code) once “something is ready to be done”.
- Is very cheap/easy to do with concurrent hardware, rather than CPU+PollingCode.

Rather than a super-loop’s polling code executing as frequently as the loop allows, the pseudo-polling “code” is ALWAYS running in interrupt hardware, until the Interrupt happens.

When might CPU cycles make sense instead?

- on the main CPU
- on a baby core (e.g. M0)

When Jitter is acceptable, requirements are fuzzy, algorithms half-baked.



Timer Services

What are timers used for?

Commonly 3 things:



Timer Services

What are timers used for?

Commonly 3 things:

- Stopwatch
- Holdoff
- Repeating



Micrium OS Timer Services

Software Timers used for:

- Duration Measurement (Stopwatch)
- Single Shot Timer (Need to do something *then* (watchdog), or have nothing to do until *then*)
- Periodic Timer (with optional delay, to trigger recurring time-based work)

What is the first thing you need to know about ANY timer for it to be useful?



Micrium OS Timer Services

Software Timers used for:

- Duration Measurement (Stopwatch)
- Single Shot Timer (Need to do something *then* (watchdog), or have nothing to do until *then*)
- Periodic Timer (with optional delay, to trigger recurring time-based work)

What is the first thing you need to know about ANY timer for it to be useful?

ClockTick: Timer Granularity (and inversely proportional:
Time-to-Aliasing)



Hardware Timer Implementations of ClockTick

- Hardware Timer

- Frequency Determination (that granularity question...)
 - Input clock source selection (frequency, power, cost, stability)
 - Configurable divider (e.g. TIMERn_CTRL.PRESC)
- Reload methods
 - None (free-running)
 - Software (set single-shot ISR)
 - Hardware reload register
- Up/Down variations

Limitation:



Hardware Timer Implementations of ClockTick

- Hardware Timer
 - Frequency Determination (that granularity question...)
 - Input clock source selection (frequency, power, cost, stability)
 - Configurable divider (e.g. TIMERn_CTRL.PRESC)
 - Reload methods
 - None (free-running)
 - Software (set single-shot ISR)
 - Hardware reload register
 - Up/Down variations

Limitation:

- Frequency scaling is often available only at values of 2 to an integer exponent (e.g. Pearl Gecko)



Example: Free-running counter, with SW Maintenance

Assume:

- We want a periodic timer of 10ms
 - The input clock is 1MHz.
 - Divider: Desired is $1\text{MHz} * 10\text{ms} = 10,000$.
 - 8192 is closest, leading to an actual frequency of $1\text{e}6/8192 = \text{approx } 122\text{Hz}$.
- Interrupt generated by timer on increment; ISR increments the OS “TICKS”, OS looks for tasks to unblock

Can you imagine any problems with other timer services from this configuration? (hint: stopwatch...)



Example: Free-running counter, with SW Maintenance

Assume:

- We want a periodic timer of 10ms
 - The input clock is 1MHz.
 - Divider: Desired is $1\text{MHz} * 10\text{ms} = 10,000$.
 - 8192 is closest, leading to an actual frequency of $1\text{e}6/8192 = \text{approx } 122\text{Hz}$.
- Interrupt generated by timer on increment; ISR increments the OS “TICKS”, OS looks for tasks to unblock

Can you imagine any problems with other timer services from this configuration? (hint: stopwatch...)

Is it a time-to-aliasing, or a granularity problem?



Evaluation of this free-running solution

Granularity:

- Stopwatch samples just before (N) and after (N+1)
 - Perceived time = 2 ticks * 8.192ms/tick
- Stopwatch samples just after (N) and before (N+1)
 - Perceived time = 0 ticks.

Anti-aliasing:

- 32b register: $(2^{32}) * 8.192\text{ms} = 48 \text{ day}$ anti-aliasing period.



Example: Software Reload of One-Shot

Counter initially loaded with the number of higher-frequency clocks needed for the desired clock tick.

- Interrupt is generated by underflow of count-down counter
- ISR reloads the value, restarts counter by reloading, increments timer ticks accumulation, looks for tasks to unblock, etc.
- Benefits?
- Disadvantages?



Example: Software Reload of One-Shot (cont.)

- Benefits:
 - stopwatch resolution greatly improved (can use the counter register as more bits)
 - if counter continues after underflow, SW can estimate/correct clock drift
- Disadvantages:
 - the extra bits will need to be normalized by the value pre-loaded to convert into tick units
 - The time that it takes to reload the counter to get it counting down again is “LOST”. (Can remove the bias, but not necessarily all variation.)
 - Most Risky: what if a critical section held off the timer interrupt? "**LOST TIME!**" (at least if counter stops upon underflow)



Example: Hardware Reload Register

Counter loaded with the number of higher-frequency clocks needed for the desired clock tick.

- Counter restarted BY HARDWARE by reloading value from a register designated for this.
- Interrupt is generated for underflow of the count-down counter.
- ISR increments timer ticks accumulation, looks for tasks to unblock, etc.
- What's better now?
- Are you guaranteed exactly the desired clock tick now?



Example: Hardware Reload Register

Counter loaded with the number of higher-frequency clocks needed for the desired clock tick.

- Counter restarted BY HARDWARE by reloading value from a register designated for this.
- Interrupt is generated for underflow of the count-down counter.
- ISR increments timer ticks accumulation, looks for tasks to unblock, etc.
- What's better now?
- Are you guaranteed exactly the desired clock tick now?

No Jitter! But the desired and fundamental clocks may not have much in common.



Time Jitter

Aside from the jitter that could be introduced by reloading counter values from software, how can we reduce jitter in stopwatch measurements?

- Poll a counter value
 - What additional steps are needed?
- Optimize OS clock tick
 - What optimization might you force on the tick selection?



Time Jitter

Aside from the jitter that could be introduced by reloading counter values from software, how can we reduce jitter in stopwatch measurements?

- Poll a counter value
 - What additional steps are needed?
- Optimize OS clock tick
 - What optimization might you force on the tick selection?
- Polling in a spin-loop: disable interrupts (critical section) to guarantee no lost counter observations
- Clock tick: make it smaller. (Performance disadvantages coming in future lecture)



ECEN 3753: Real-Time Operating Systems

Diagramming and Unit Testing



Why do we diagram?

- To abstract what's important
- To analyze critical connections/info
- To look for errors
- To check for completeness
- To communicate important info

There is no single reason, nor a single best solution.



Some popular types of diagrams

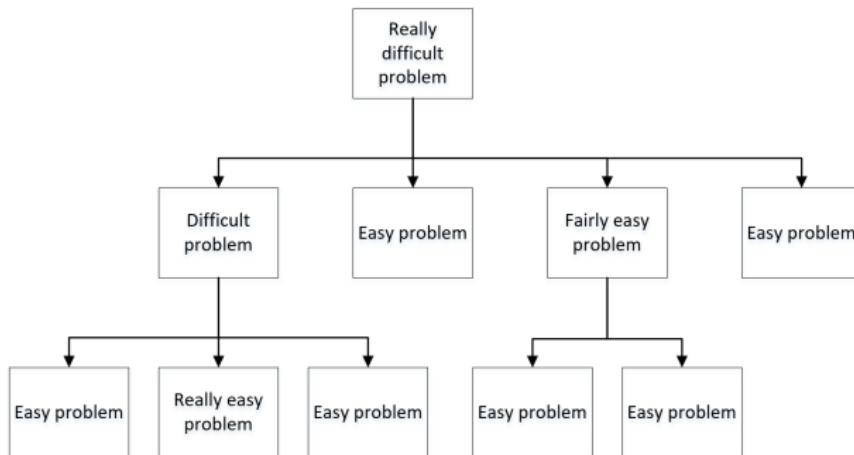
As listed in the main lecture series:

- **Structure Diagram:** Repeatedly divides a problem into smaller and smaller pieces
 - Useful for communicating top-down or bottom-up design
- **Data Flow Diagram:** Data and Transformations on data
 - The basis for what we call a Task Diagram, where:
 - Tasks are the “Transformations”
 - Shared data and Inter-Task Communications are the “Data”
 - Helps us see where data interfaces might be more obvious for unit test boundaries
- **State (Machine) Diagram:** States and transitions/triggers
 - Sometimes very useful to help analyze different states of critical data that may align with triggering of data transformations
- **Sequence Diagram:** How parts of a system do their part (computation or signaling) to fulfill a use case's demands
 - *Optional* example perusal:
 - SoftwareIdeas.net: Sequence-Diagrams



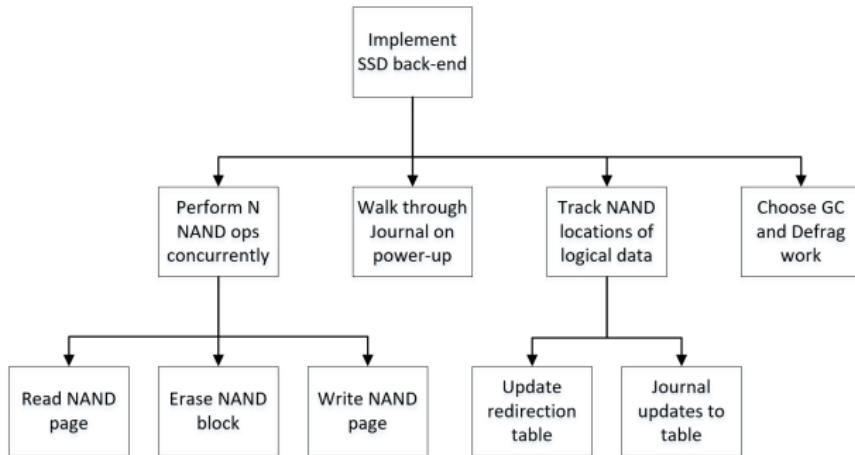
Generic Structure Diagram

While the general form of this may not seem like a big deal, it is a very powerful tool, regardless of whether you approach a problem from the top or from the bottom.



SSD Back-end Structure Diagram

For instance, that same structure shape can describe much of what is required to provide an SSD's back-end (the part that actually stores data, as compared to communicating with the server or PC it's plugged into).



Top-Down vs. Bottom-Up Design

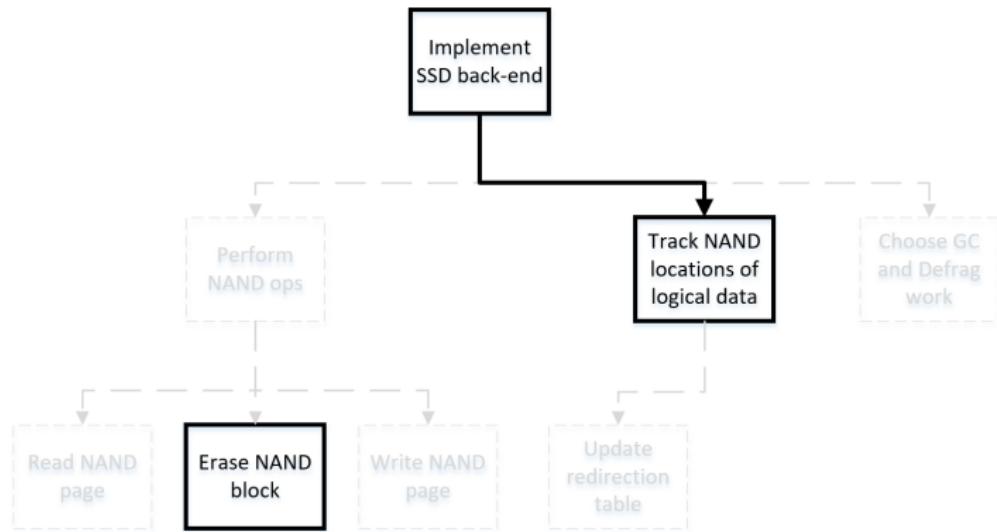
Neither is the *only* goal.

- Some people are more comfortable thinking in abstractions without needing to know the details at the bottom of the diagram.
- Others are more comfortable connecting to some low-level hardware or interfacing to a chip spec, without needing to know how the other parts of the system need to use that or do other work.

Most systems will require some top-down, and some bottom-up design, regardless of personal preferences.



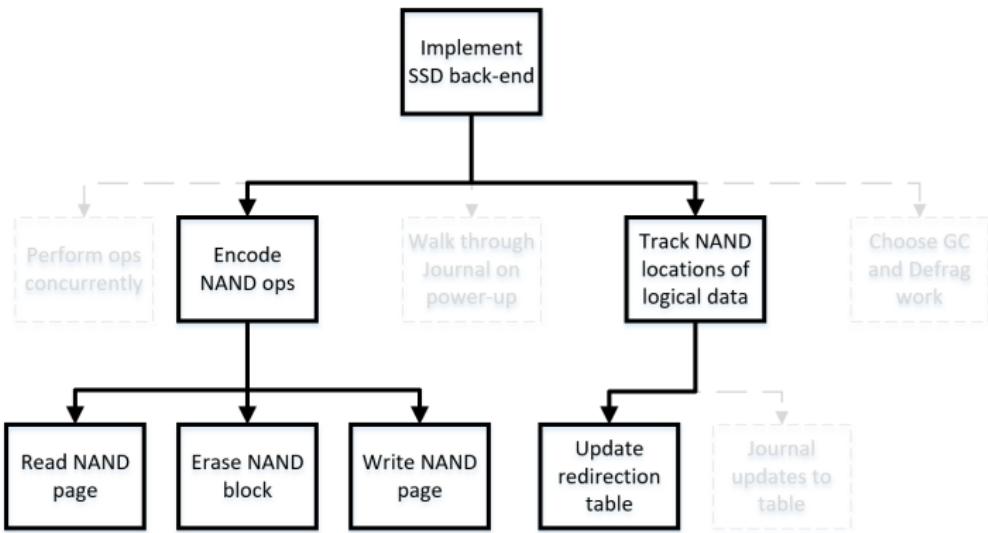
Example of a possible progression (1/4)



Two people might start with some vague ideas (gray, dashed), and some ideas that are getting *real* (black, solid):

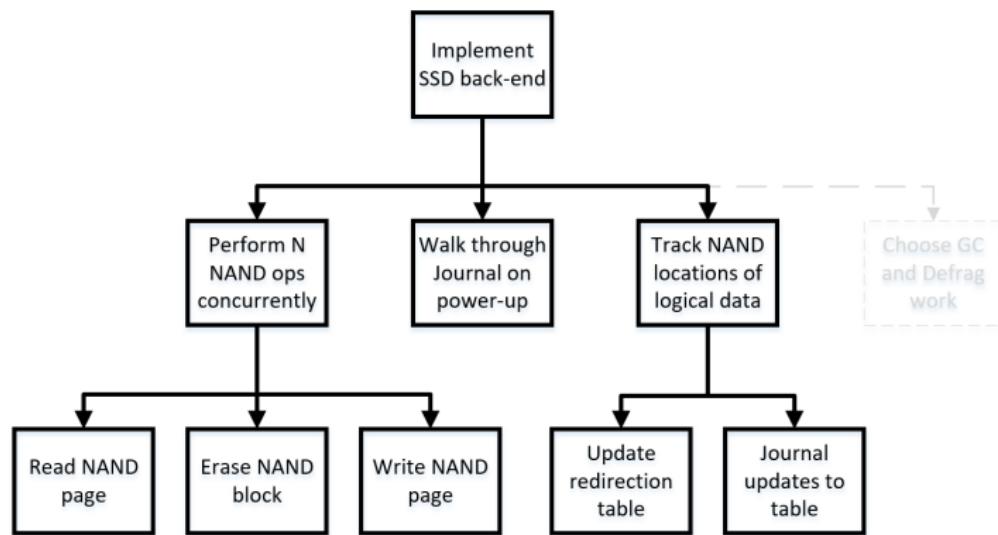
- ① Need to map logical data addresses to changing physical ones
- ② Start figuring out how to access Flash, without data payloads

Example of a possible progression (2/4)



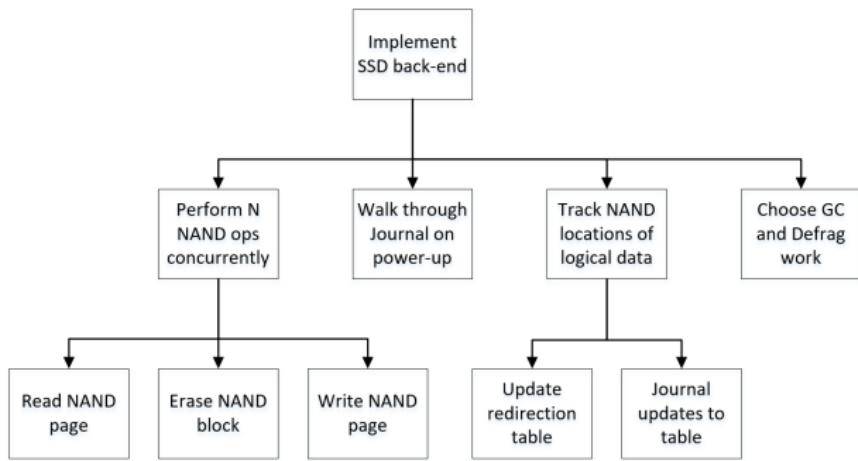
- ① Mapping *use* leads to updates *to* the map
- ② Generalization of one op leads to ideas of how to do many, concurrently

Example of a possible progression (3/4)



- ① Mapping updates require journaling for power-loss protection
- ② Concurrency in hardware for all required ops

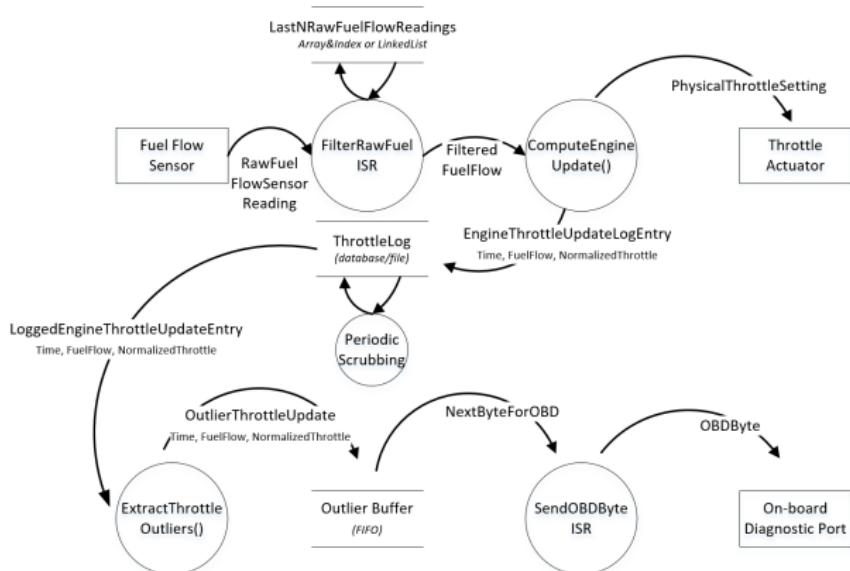
Example of a possible progression (4/4)



Finally, the whole design is done. Some (especially lower-level) solutions may already be coded from the process of building understanding. *Assume* that most of that should be re-designed/re-written, or get used to disappointment.

Data Flow Diagram

... the next diagram with very broad appeal to us as designers of software systems. It shows data (stores, or transient) and the transformations of data, as it exists between inputs and the outputs.



Identify Unit Test Boundaries

The question arises: Where should we cut that picture and implement testing of parts without the whole?

Read Now:

- Pragmatists.com: Test doubles, fakes, and stubs

Optional:

- MartinFowler.com: Mocks aren't stubs



Targeting Appropriate Unit Testing

- Why
- What
 - For this class
 - More generally
- Example
 - Scheduling



Why Unit Test?

By testing pieces before the whole solution is ready, you move the detection of design or implementation bugs earlier in your work, and are testing smaller (simpler) pieces. ("Shift Left" is a reference to a schedule)

Is this benefit free?



Why Unit Test?

By testing pieces before the whole solution is ready, you move the detection of design or implementation bugs earlier in your work, and are testing smaller (simpler) pieces. ("Shift Left" is a reference to a schedule)

Is this benefit free?

Of course not. Effort has to go into building a UT framework, mocks/fakes, building the test case implementations, and figuring out exactly what the UT outputs should be. The gamble (with **excellent** odds) is that debug using UTs is less *difficult* debug than system-wide debug you'd otherwise need more of later, so there is a net benefit.

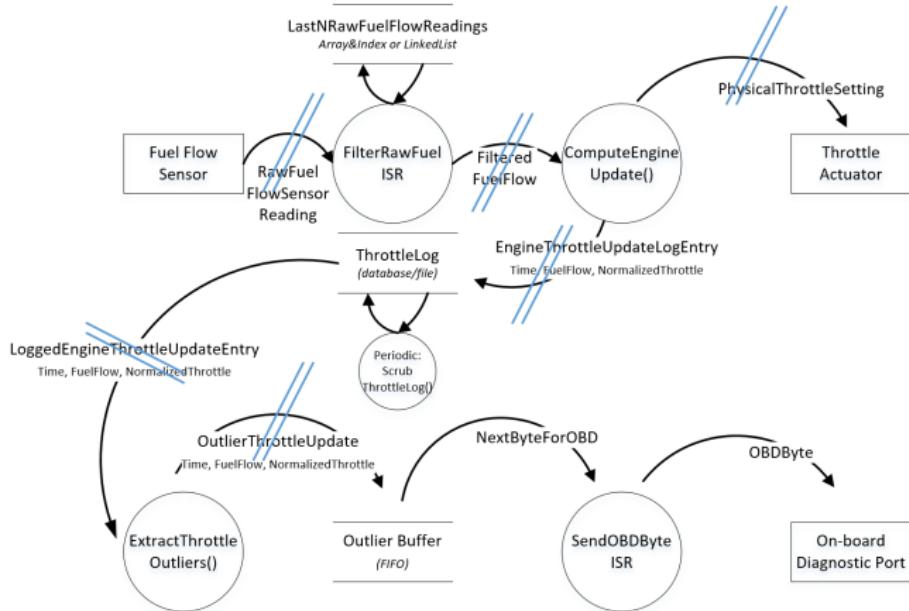


What should we Unit Test?

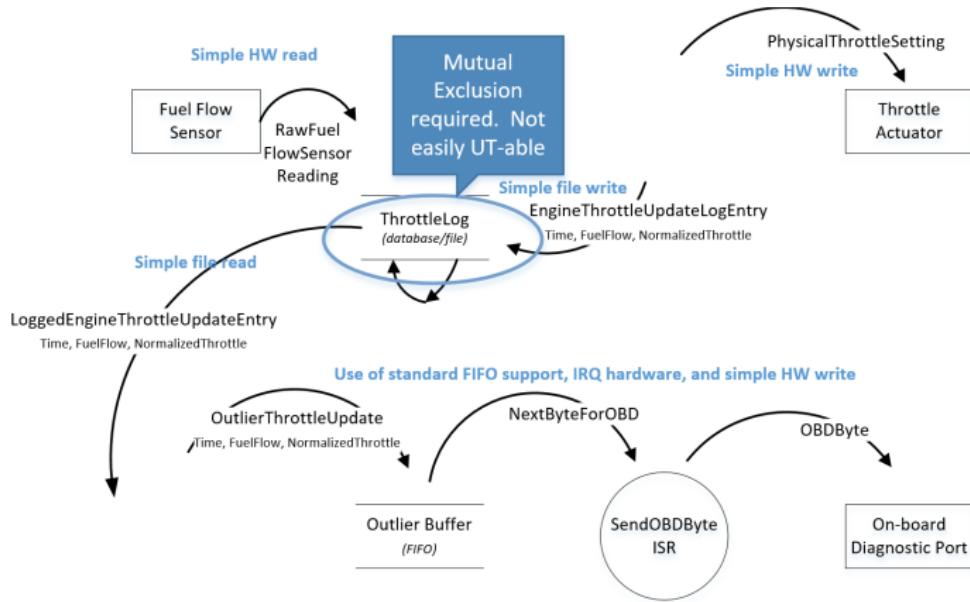
- “The stuff between parameters”
- In this class—Algorithms
 - Scheduling choices—Labs 4,5
 - A RT algorithm in your project, where it is separable from the hardware
- In general
 - It still depends, on maturity of HW (design/implementation), level of code reuse/maturity, tolerated risk parameters.
 - HW not yet made: HW Sim can support UT, or mocks/fakes can be built to allow much faster testing (with less precision).
 - HW still immature: sometimes UT (in HW sim) can allow code to be quickly verified on an updated HW design (Verif/Val)
 - If code is significantly reused generation-to-generation in products, a large amount of change will still go into new generation SW, which causes risk to “known working” SW.
 - Higher-risk, easier-to-UT code would be easier to justify than harder-to-UT, or very low risk (RTOS mechanisms, IRQ HW)
 - Hugely useful as part of speedy regression testing.



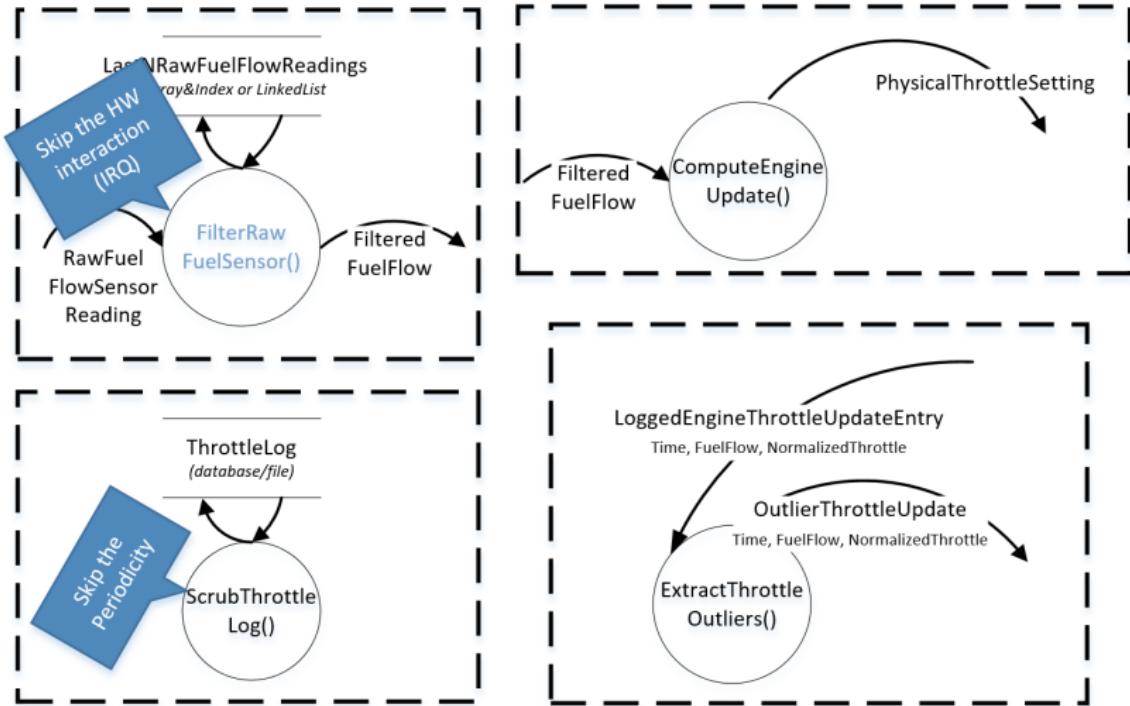
Data Flow Diagram, with Cuts



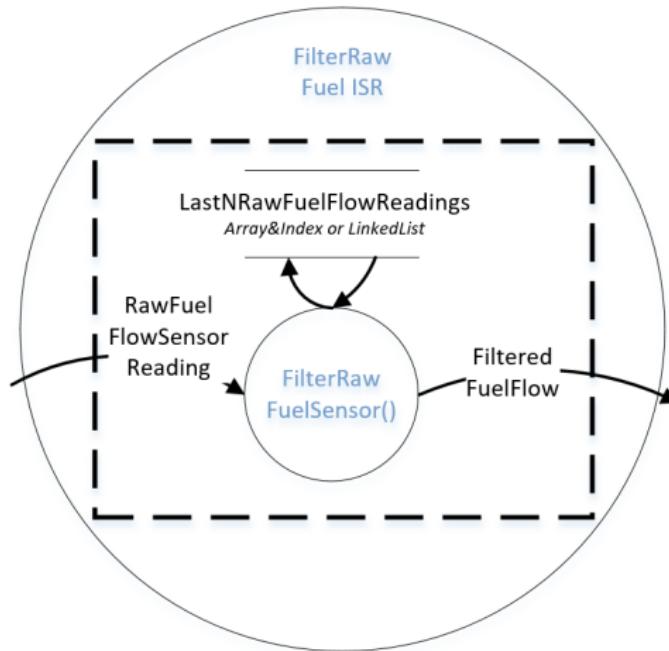
Judgement: What NOT to Unit Test?



Judgement: What TO Unit Test?



Unit Testing the Guts



Unit Testing in this Course

Prior to Lab 4 (you will write unit tests to verify your own scheduling implementations):

- Your TA will introduce you to the unit testing framework that you'll use to:
 - Initialize a test
 - Run the test
 - Verify results of the test

Starting from that, for Labs 4 & 5 you will build multiple tests in a suite, and possibly multiple suites, that check behavior beyond the ends as well as the middle of healthy parameter ranges, etc.

In your project, you'll need to plan your own unit tests, and execute them as part of your proof-of-demonstrability, along with functional tests. You must submit all of your test plans and results BEFORE the actual demo.



Onward to State Diagramming

Imagine that dual-copies are used to protect some really important data.

- Usually:
 - Write(A), Write(B) to save update.
 - Read (A) to read back up at power-up.

Let's use the 7-step programming method to derive some appropriate behavioral responses.



Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds



Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds \Rightarrow Write(A)



Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds \Rightarrow Write(A)
- Read(A), Read(B)



Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds \Rightarrow Write(A)
- Read(A), Read(B) : No writes needed



Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds \Rightarrow Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails



Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds \Rightarrow Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails \Rightarrow Write(B)



Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds \Rightarrow Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails \Rightarrow Write(B)
- Read(A) fails, Read(B) fails



Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds \Rightarrow Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails \Rightarrow Write(B)
- Read(A) fails, Read(B) fails \Rightarrow wave down a passing car



Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds \Rightarrow Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails \Rightarrow Write(B)
- Read(A) fails, Read(B) fails \Rightarrow wave down a passing car

We're going to turn that into a complete state diagram for the non-volatile states of the duplicated data, using sequence numbers to identify older/newer copies, denoted "S" in the diagrams.



Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds \Rightarrow Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails \Rightarrow Write(B)
- Read(A) fails, Read(B) fails \Rightarrow wave down a passing car

We're going to turn that into a complete state diagram for the non-volatile states of the duplicated data, using sequence numbers to identify older/newer copies, denoted "S" in the diagrams.

- Read(A) and Read(B) succeed, but $S(A) > S(B)$



Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds \Rightarrow Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails \Rightarrow Write(B)
- Read(A) fails, Read(B) fails \Rightarrow wave down a passing car

We're going to turn that into a complete state diagram for the non-volatile states of the duplicated data, using sequence numbers to identify older/newer copies, denoted "S" in the diagrams.

- Read(A) and Read(B) succeed, but $S(A) > S(B) \Rightarrow$ Write(B)



Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds \Rightarrow Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails \Rightarrow Write(B)
- Read(A) fails, Read(B) fails \Rightarrow wave down a passing car

We're going to turn that into a complete state diagram for the non-volatile states of the duplicated data, using sequence numbers to identify older/newer copies, denoted "S" in the diagrams.

- Read(A) and Read(B) succeed, but $S(A) > S(B) \Rightarrow$ Write(B)
- Can we only Read(A) on power-up to save time (as long as it's readable)?



Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

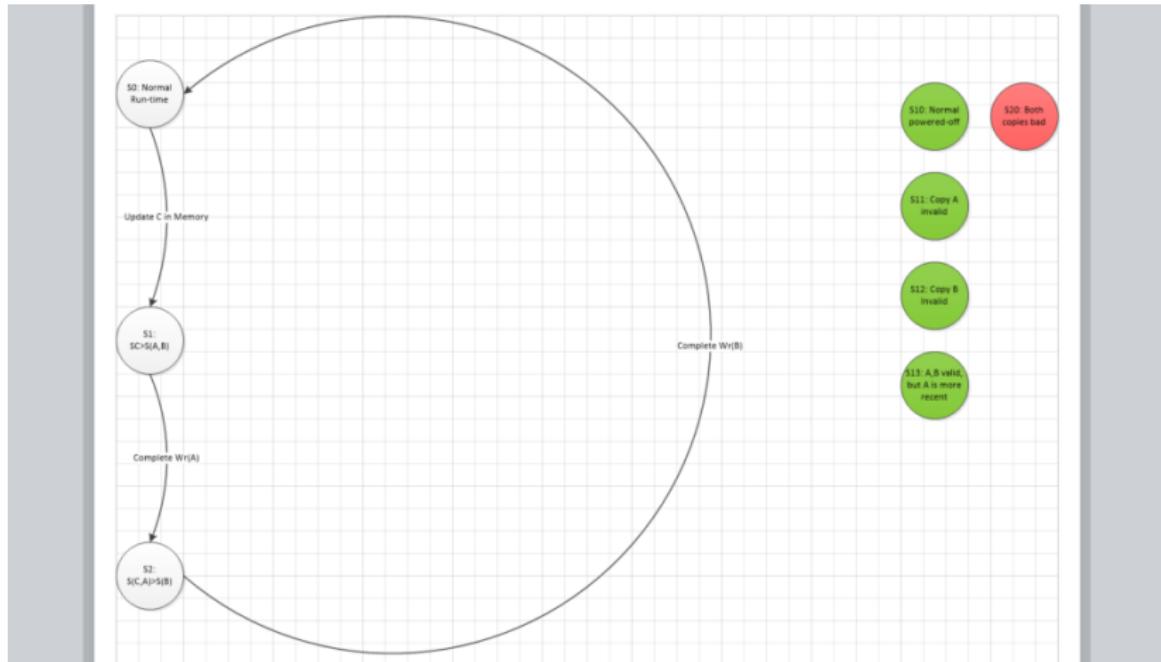
- Read(A) fails, Read(B) succeeds \Rightarrow Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails \Rightarrow Write(B)
- Read(A) fails, Read(B) fails \Rightarrow wave down a passing car

We're going to turn that into a complete state diagram for the non-volatile states of the duplicated data, using sequence numbers to identify older/newer copies, denoted "S" in the diagrams.

- Read(A) and Read(B) succeed, but $S(A) > S(B) \Rightarrow$ Write(B)
- Can we only Read(A) on power-up to save time (as long as it's readable)? \Rightarrow Look at the prior bullet.



Simple flow when update is made, and no problems arise



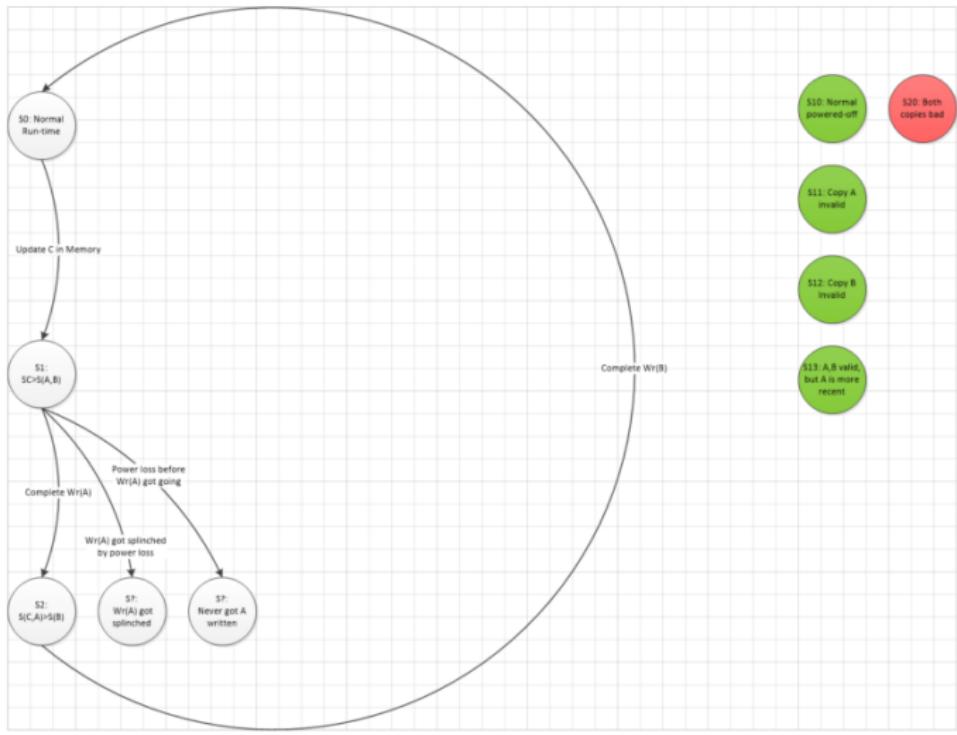
Who has Read-or-Watched Harry Potter?

If a Write is interrupted by an unanticipated power-loss, the data is “splinched” – that is, the earlier part of it is from a data update, but the rest is what was there from a prior write.

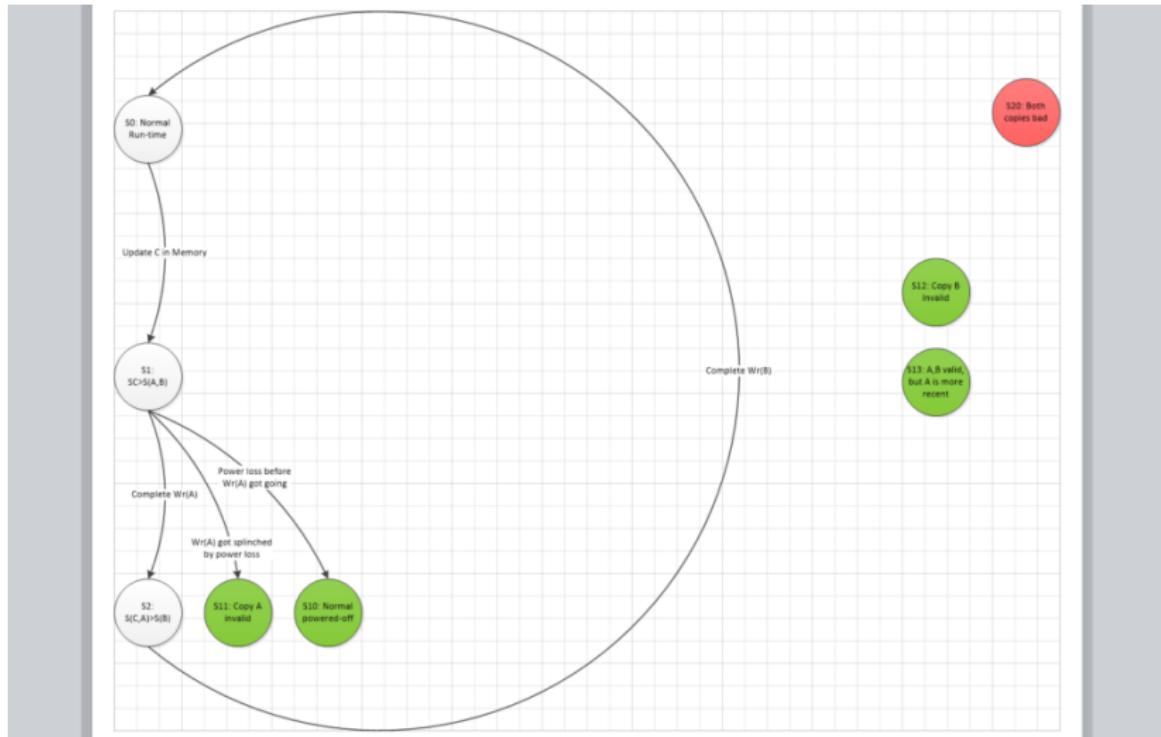
In systems with Error Detection Coding, this will be detected as “bad data”, rather than “Oop, I’ve been splinched!” Let’s assume that we have EDC.



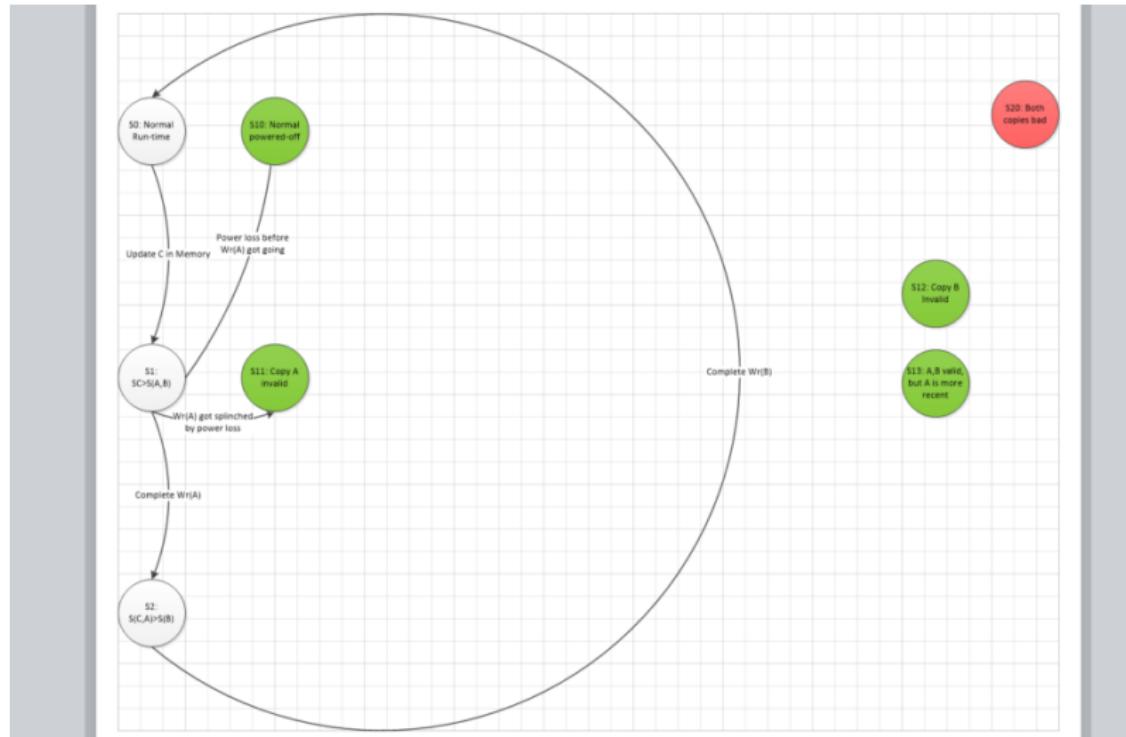
Add the 3 Wr(A) outcomes (success, splinch, not started)



Note equivalence to the NV states from the right



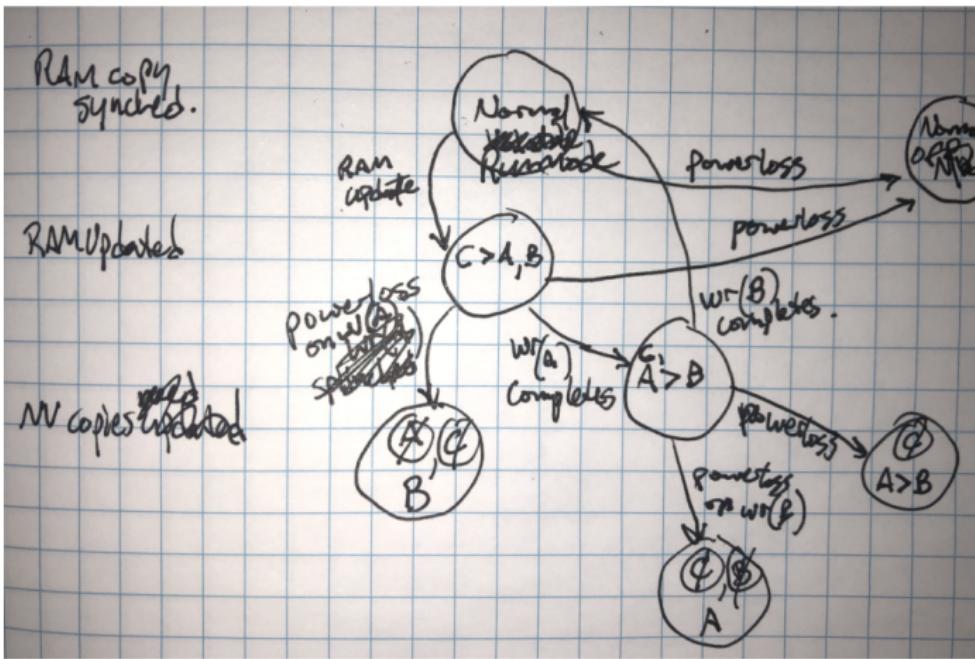
Now scoot the states upward in the diagram



Now add the 3 outcomes for Wr(B) as well.



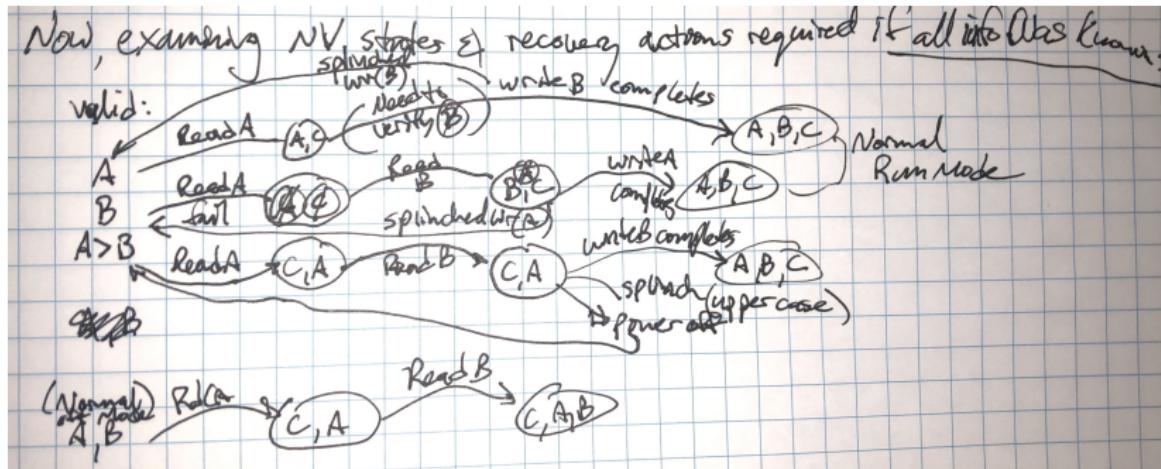
Level-setting: Reality



These things never just pop out in finished form. Even though I've solved this problem previously, I had a couple of false-starts with segmentation of the state space, and in breakdown of events.



Level-setting: Reality, cont.



back to it...

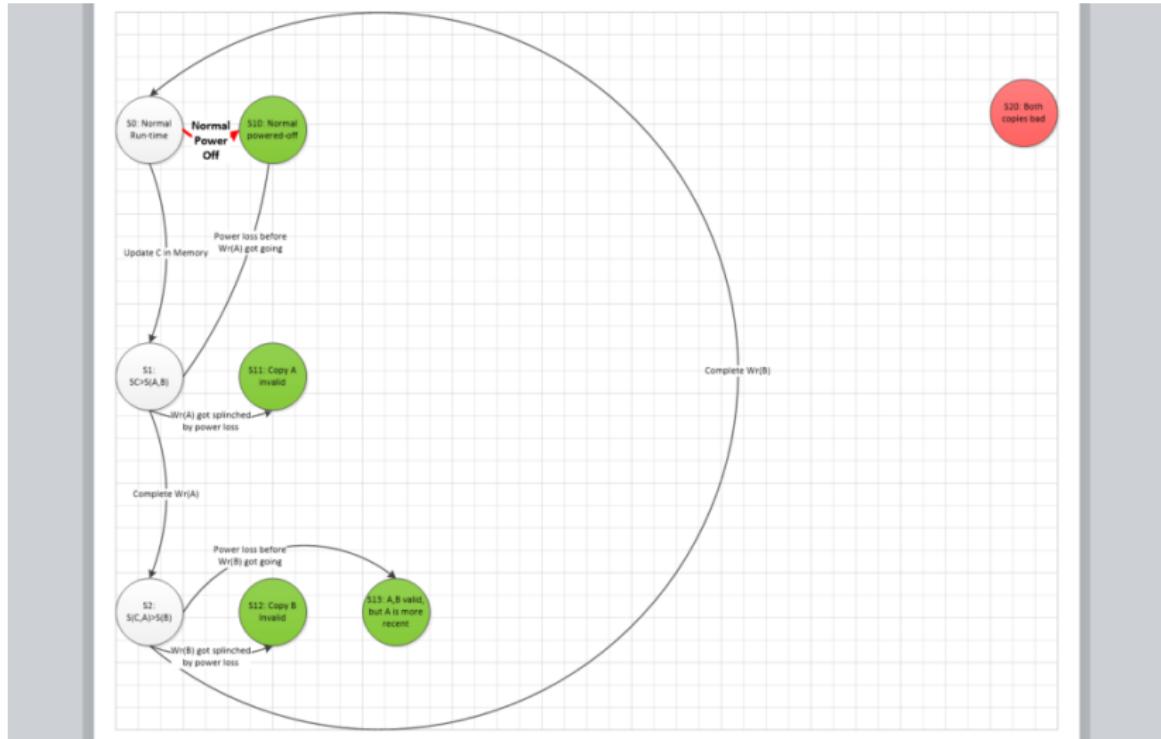
We completed the state diagram for

- being already powered-up,
- no remanence issues (data decay)
- through all possible combinations of updates and a single power loss
- OR DID WE?

Can you spot the missing combination/case?



All “Run-time” cases, with Remanence



Next Step: Determine which NV case

- Read Both A, B
 - Note Success, SequenceNum of each read
- Use the (a) new/good one
- Write the old or defective one

In the manner we defined A as "first-to-update", the old one must be B and the new one is A.



Concurrent vs. Sequential Reads

Depending on media, A and B could be accessible at the same time, or only sequentially.

- One HDD: sequential
- Multiple NAND chips: possibly concurrent

Support Concurrency?

- Is there a simple generalization we can make on the SD to allow concurrency?
 - Not simple, unless we change A/B to be first/last for writing, and older/newer on reading
- Is true concurrency safe, as the system is defined?
 - What if concurrent writes are both splinched?

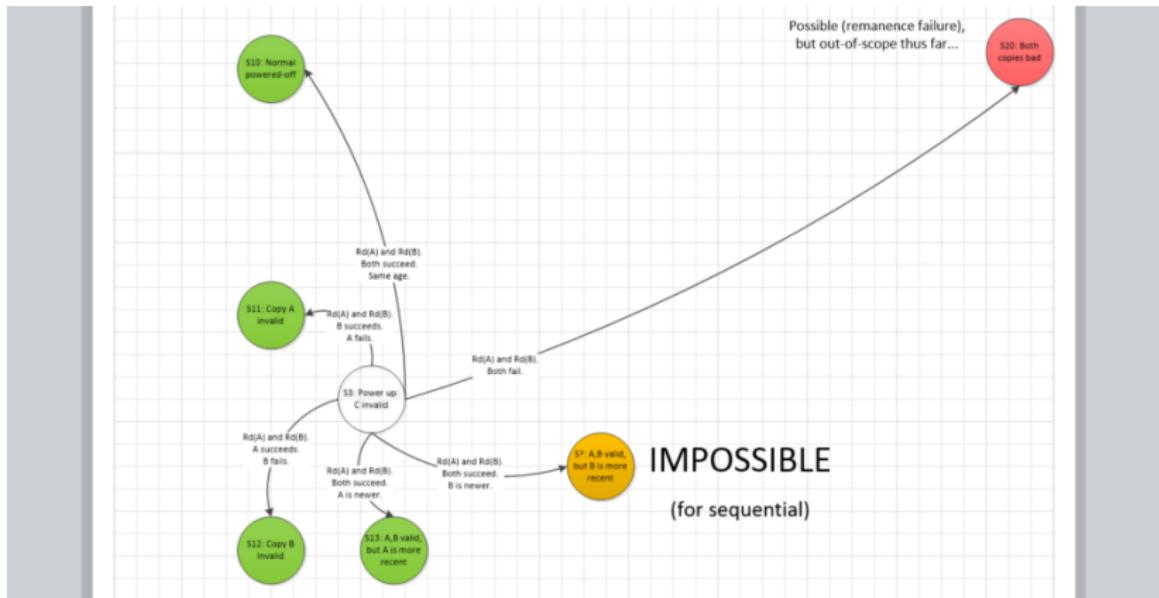


Continuing with Sequential Assumption

In this SD, we can't know which NV state the data is in when we begin the power-on-recovery. So the first step is to simply go from "unknown NV state" to unambiguous state.



Power-On Discrimination



We'll circle back on the non-remanent issues in a bit.



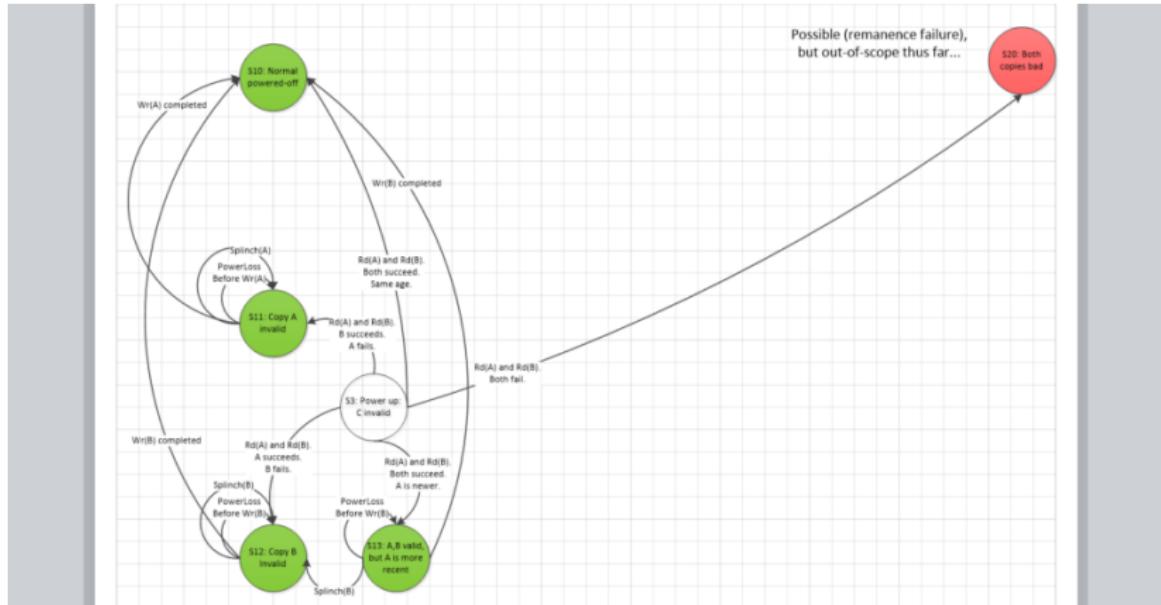
Back to the Bigger Picture

The state diagram that represents the NV states (which our original SD exceeded, by showing the normal volatile updates) doesn't need to show the Power-On Discrimination.

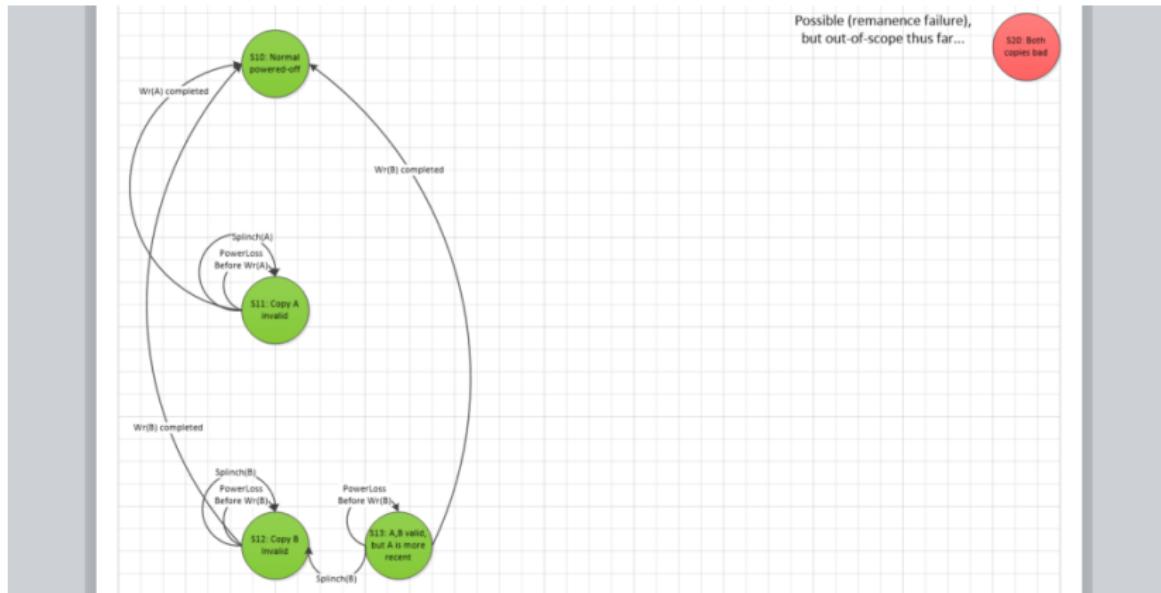
It does need to show the events that lead to different NV states.



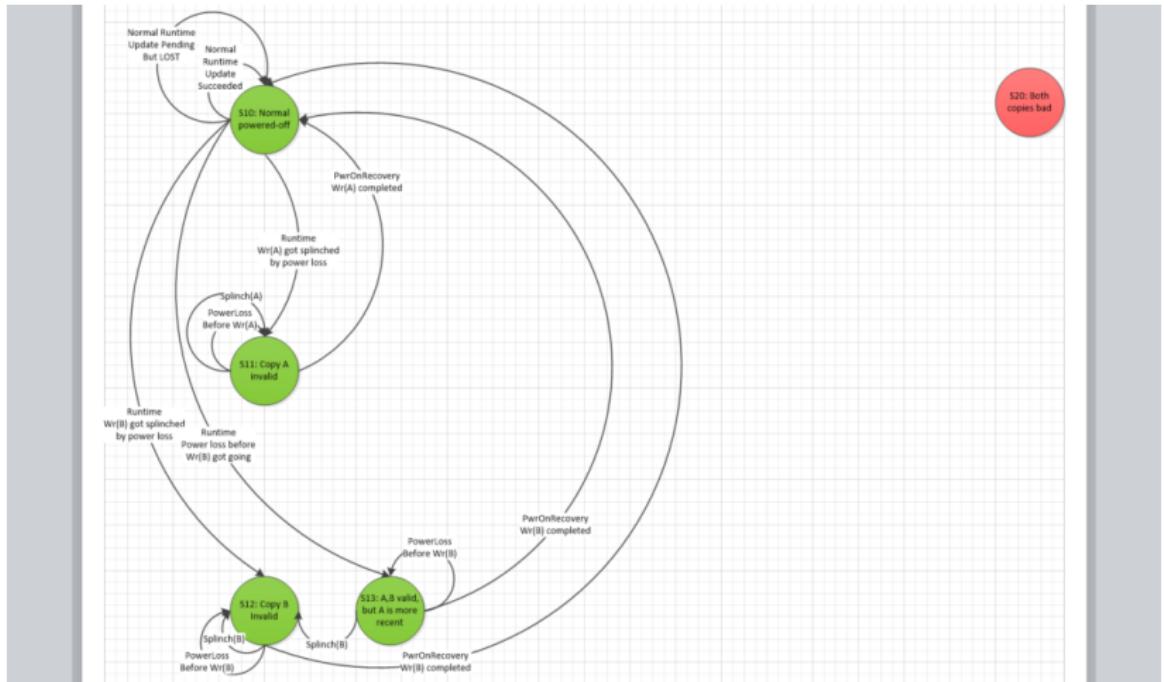
Overly-Busy Discrimination and Recovery



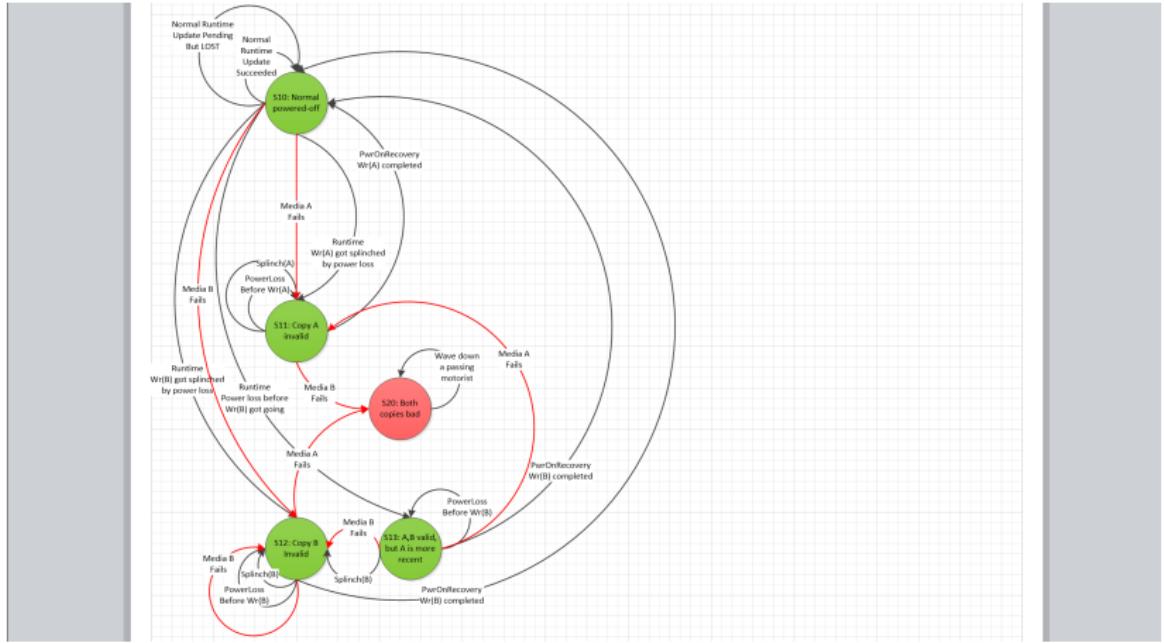
Power-On Recovery (NV State changes, stable media)



RunTime and Recovery (NV Only, Media Perfect)



Now, Add Single Media Failures



Unit testing of this system

Read:

- https://blog.cleancoder.com/uncle-bob/2014/05/14/TheLittleMocker.html

and reflect on how you could design unit tests from this state diagram.

There are parts of unit testing here that could be written to use Dummy/TestDouble, Stub, Spy, Mock, and Fake constructs. But Mocks and Fakes take a lot more effort and are more complicated, so we more likely would build several test cases from the state diagram, and then unit test the responsible functions with Dummies, Stubs, and Spies to ensure that the simple internal decisions are implemented correctly in code.

When you do the Scheduling labs in this course, you'll need to generate multiple specific tests to ensure that your scheduling logic is operating as expected.



ECEN 3753: Real-Time Operating Systems

Tasks



What Is A Task

- Sub-unit of program/application that can execute somewhat independently
- When using an RTOS, an application likely consists of more than one task.
- Each task may:
 - Generate data
 - Process data
 - Control hardware



Why Multi-Tasking

- In a typical embedded system, a program must wait for a physical operation to complete
 - Mechanical operations are *slow* relative to CPU execution.
- Polling loops are sub-optimal
 - Power: 100% Duty Cycle—max consumed
 - Latency: we can only benefit from operation completion at the point that we check it.
 - Coupling: If the CPU needs to do other work while waiting, this needs to be managed inside the polling loop.
- Using a multi-Tasking OS, if a program needs to wait for an operation to complete, a responsible Task can **block**.
 - A blocked Task must then be awakened by an interrupt or some signal from another Task.
 - Other Tasks can simply execute on the CPU while one task is waiting.

Compare to “multi-tasking”, which could be done with a superloop, interrupt-driven quasi-concurrency, etc.



Program Execution: Processes, Threads, and Tasks

- Process
 - Heavyweight (relatively) unit of program execution.
 - Typically used in virtual memory systems and paged in to switch between processes.
 - Owns its own memory space, including heap.
 - Windows '.exe' (after it has started running)
- Thread
 - Lightweight (relatively) unit of program execution.
 - A process may spawn one or more threads.
 - Threads share the memory space (including global data) of their parent process, but have separate stacks for program execution.
 - TEND to be aligned more to concurrent subtasks
- Task
 - Lightweight unit of program execution, very similar to a thread.
 - TEND to align with less-concurrent subtasks in real-time, but allowing decoupling



What Needs to Happen—Pulling our thoughts together

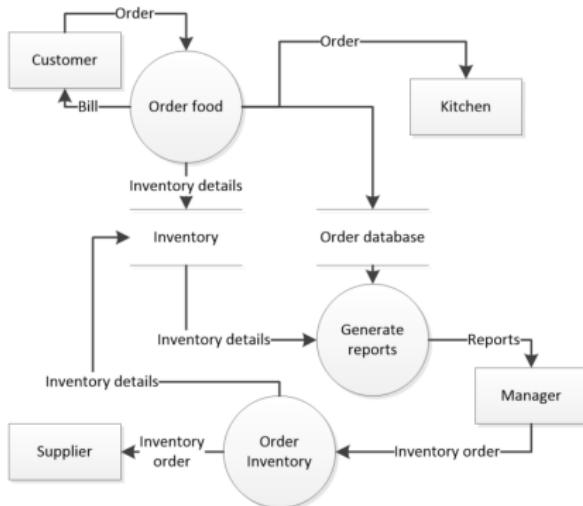
How We Can View, Model, and Document a Multi-Task Process

- Our Textbook shows Software Design Models, System Run Time Models, and some Task Development by examining System functions and sub-functions in an attempt to structure software (Chapter 1).
- Another method that can be useful is to work from use-cases through blocking events and messages, to diagram an algorithm (step 3 of the 7-Steps-to-Coding method), utilizing some tools that may be especially effective for this.
 - Data Flow Diagram (shows data between processing entities)
 - State Machine Diagram (shows (task) states, what transitions are caused by and effects)
 - Sequence Diagram (good to formally capture your use cases)



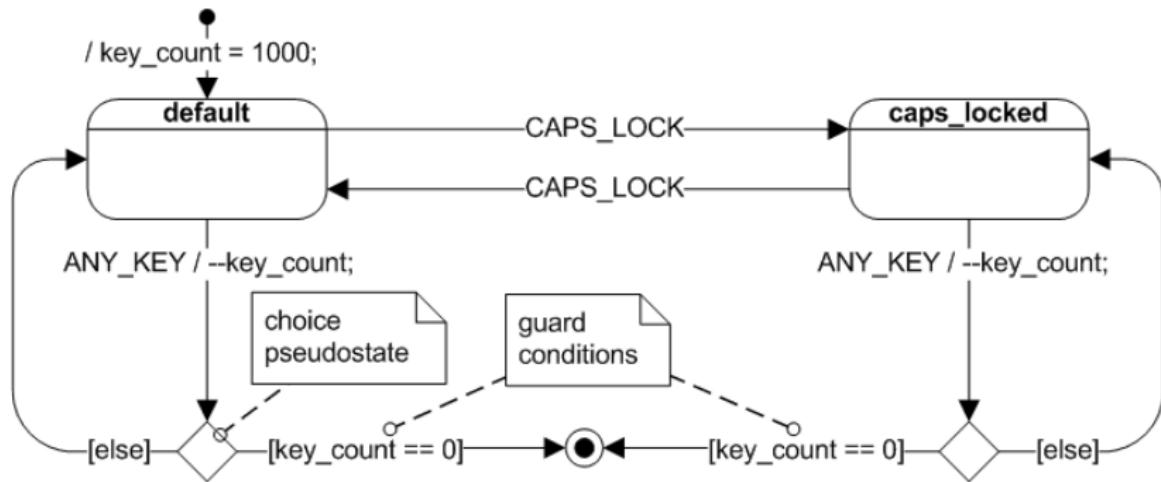
Data Flow Diagram

This tool can be especially useful in seeing the boundaries between tasks, by examining minimized data transfer boundaries, and action boundaries. Additionally, it may help you discriminate shared data stores from simple I/O data.



State Machine Diagram

This tool can be especially useful in seeing and documenting the changing behavior within a task, in reaction to external events or information.



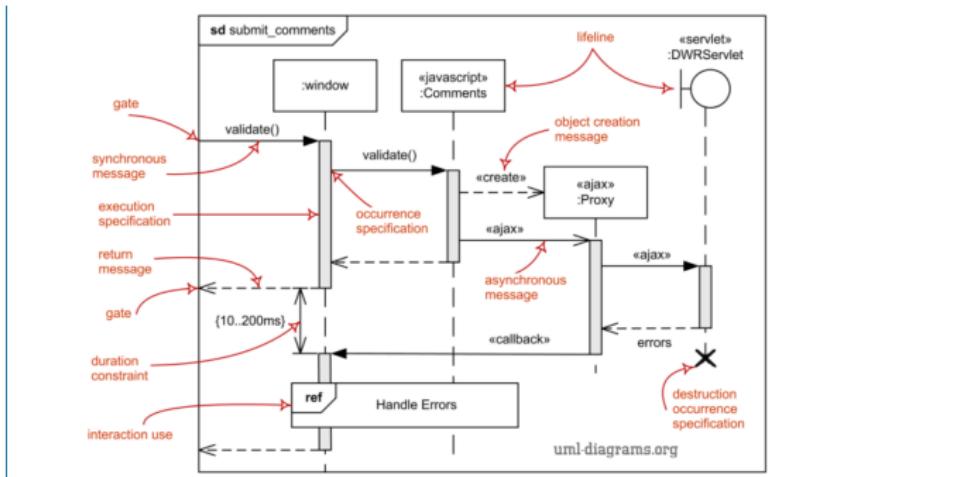
<image from https://en.wikipedia.org/wiki/UML_state_machine>

Extra-Task behavior and info passing will be covered in a few weeks, in Inter-Process Comms and Shared Data.



Sequence Diagram

This tool can be used to formally document well-defined interactions between different entities. It can be used to help clarify use cases, but due to its assumption of entities (e.g. tasks) it may be premature to use it when the task boundaries are not yet clear.



<image from <https://www.uml-diagrams.org/sequence-diagrams.html>>



Back to Some Nuts-and-Bolts...

Let's presume that we'll come up with the right algorithms, task divisions/messaging/etc.

What are the essentials that an RTOS juggles to make that all work?



Resources That A Program Needs To Execute

- CPU to run on
 - Only one task may run on a physical CPU at a time.
- Instructions to fetch
 - Program counter (PC) points to the current instruction being executed
- Mechanism to maintain the current state of the program we are executing, including:
 - What code is currently running
 - Where does code execution return to
 - After a function call
 - After an interrupt
 - After switching to a new task (context switch)
- Local data
 - Registers
 - Stack
 - Thread-local storage



Program State: The Task Stack

- Each task has its own stack (**All the same size?**)
- Interrupt handlers often have their own stack
- When a task executes, information on the stack provides information about what code is executing at any point in time
 - Local variables
 - Parameters that have been passed from one function to another
 - Function return addresses
- On ARM, stacks typically grow... **Aside: What does "upward"/"downward" mean to you?**



Program State: The Task Stack

- Each task has its own stack (**All the same size?**)
- Interrupt handlers often have their own stack
- When a task executes, information on the stack provides information about what code is executing at any point in time
 - Local variables
 - Parameters that have been passed from one function to another
 - Function return addresses
- On ARM, stacks typically grow... **Aside: What does "upward"/"downward" mean to you?**
 - toward address 0

The use of the terms **Upward/Downward** can fuel holy wars. Some insist on drawing memory with address zero at the top, some at the bottom. We'll try to be agnostic, and use unambiguous terminology.



The ARM Application Binary Interface (ABI)

- Parameters typically in registers R0..R3, possibly on stack for longer parameter lists or large data structures
- Function return value in R0
- Stack pointer (SP = R13)
- Link Register (LR = R14) (return address)
 - Often saved on the stack on function entry (to allow for nested calls), popped into PC on return.
- Program Counter (PC = R15)
- R4..R11 are generally volatile and should be preserved (on the stack) across function calls if they are needed locally.
- Compiler may optimize around ABI
 - Compile time computations (parameters + return values)
 - Inline functions



The ARM Application Binary Interface (ABI)

- Parameters typically in registers R0..R3, possibly on stack for longer parameter lists or large data structures
- Function return value in R0
- Stack pointer ($SP = R13$)
- Link Register ($LR = R14$) (return address)
 - Often saved on the stack on function entry (to allow for nested calls), popped into PC on return.
- Program Counter ($PC = R15$)
- R4..R11 are generally volatile and should be preserved (on the stack) across function calls if they are needed locally.
- Compiler may optimize around ABI
 - Compile time computations (parameters + return values)
 - Inline functions

Why review calls/params/returns now?



The ARM Application Binary Interface (ABI)

- Parameters typically in registers R0..R3, possibly on stack for longer parameter lists or large data structures
- Function return value in R0
- Stack pointer ($SP = R13$)
- Link Register ($LR = R14$) (return address)
 - Often saved on the stack on function entry (to allow for nested calls), popped into PC on return.
- Program Counter ($PC = R15$)
- R4..R11 are generally volatile and should be preserved (on the stack) across function calls if they are needed locally.
- Compiler may optimize around ABI
 - Compile time computations (parameters + return values)
 - Inline functions

Why review calls/params/returns now?

Looking for parallels as we look to perform task switching, instead of simply considering hierarchical calls.



Anatomy of a Function: Parameter Passing [32b ARM]

- The first four parameters to a function are usually passed in R0..R3
 - 64-bit values can be passed in a pair of registers (R0..R1, or R2..R3)
- Subsequent parameters (after the first 4) are passed on the stack
- For large data structures:
 - Pointer parameters are always of a `size_t` that fits in a CPU register, and may be passed in registers (regardless of what data is being pointed to)
 - If passed by value, a large data structure is passed on the stack.
 - Compiler may generate calls to `memcpy()`—**Star Wars: "wait for it..."**
 - Always more efficient to pass large data by pointer rather than by value
 - Efficiency aside, do you see any risks with a pointer to a `const`?



Anatomy of a Function: Entry and Exit

- When we enter a function
 - The following registers may be saved on the stack to preserve the current CPU context
 - LR (function return address)
 - R4..R11 (scratch registers)
 - Params passed in R0..R3 are often re-saved in another register or on the stack before they are needed to make a nested call ¹
- When we exit a function
 - The function return value is copied into R0 (or R0..R1)
 - Values that were originally pushed onto the stack are restored, including the original LR which will be the PC to which execution resumes.

¹ Params are passed in same position [R0..R3] by subsequent nested calls require fewer register/stack copies.



Anatomy of a Function: Stack Frames

- If a function declares local data, it may be stored in a stack frame
- When entering a function, the compiler is free to reserve an amount of memory at the top of the stack for local/scratch data.
 - Large local data structures—**again: ick.**
 - With optimized ARM code, a stack frame is often unnecessary, since local data can also be stored in registers (R0..R11)
- If a stack frame is allocated, the register used to point into the frame² will generally be saved on the stack at function entry (and restored on exit)
- The memory allocated for a stack frame will generally be reclaimed when a function returns.

² there is not necessarily a Stack Frame Pointer—compilers are good at math, so SP is generally sufficient



Now for some Parallels (ISR-FunctionCall-TaskSwitch)



Anatomy of a Task Switch

- Each task maintains its own stack
- CPU registers are a shared resource, but the values in these registers at any point in time are specific to only the currently running task.
- When the Micrium OS wants to switch to a different task it needs to do the following (`os_cpu_a.s`):
 - Allocate stack space to save all the CPU registers for the currently running task
 - Save SP in the current task's control block, so that it can be restored later.
 - Save CPU registers into this reserved stack space.
 - Find the control block for the task we are about to switch in.
 - Load SP from the new task's control block.
 - Load any remaining CPU registers from the new tasks's stack.
 - Set PC to the new task (SWITCH!)

The terms OUT/IN may be replaced with FROM/TO in some environments. The terminology has not converged



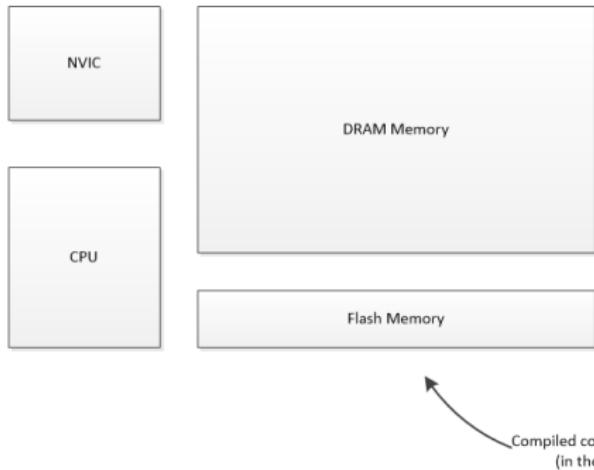
Creating a Task: What information is Needed

- Amount of memory to reserve for task stack
 - Each task owns its stack space
- Task name
- Entry point³
 - void entryFunction(void*)
- Task priority
- Task control block
 - Memory that the OS will use internally to manage/store task state
- Initialization options
 - Default time quanta (round-robin scheduling)
 - Is stack protection enabled
 - Is thread local storage enabled
 - etc.

³arg must be valid and accessible when the task starts (i.e. heap or global)

POV Switch: OS Internals to OS usage

Where does the Task Code go?



```
Task1_Code
{
    while (1)
    {
        updateOutput1(getAdcValue());
        sleep(1);
    }
}
Task2_Code
{
    while (1)
    {
        updateOutput2(getAdcValue());
        yield();
    }
}
```

Compiled code goes into a memory
(in the .TEXT segment)



main() Constructs Tasks

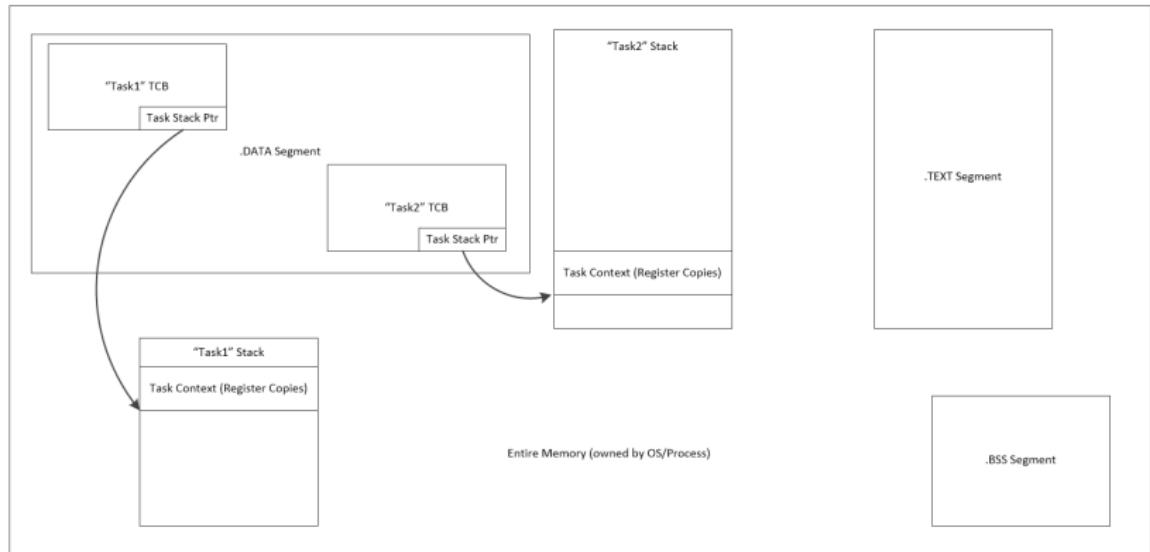
OS Calls are made to set up Tasks:

```
OSInit()  
OSTaskCreate("Task1", Task1_Code, ...);  
OSTaskCreate("Task2", Task2_Code, ...);  
    // or could create in Task1_Code, before while(1)  
OSTaskStart(...);
```



Memory Breakdown

Where do the Task constructs go?



The TCB is defined by the user code (global variable) and passed in to micrium. See

<https://doc.micrium.com/display/osiiidoc/OSTaskCreate>



Starting the Micrium OS (from Main)

- OSInit(&err)
 - Creates OS system tasks
 - Tick Task
 - Idle Task
 - Timer Task
 - Statistics Task (optional)
- OSTaskCreate(..., &err)
 - Create your application main task
 - Your task can create as many additional tasks as it needs
- OSSStart(&err)
 - Starts the OS
 - Schedules the first task to run
 - Does not return

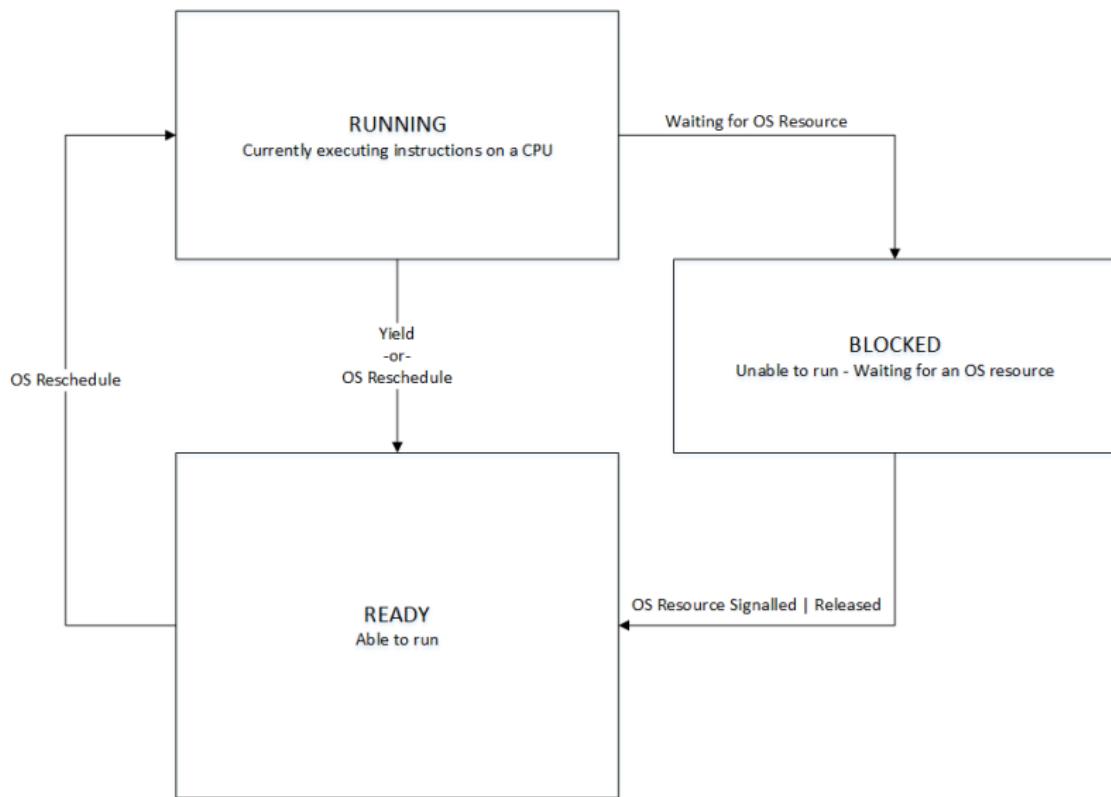


Inside Your Task Entry Point

- Initialize data needed by your task
- Infinite loop
 - Do some work...
 - Give up CPU (block or yield)



Overview of Task States



Task State: RUNNING

- For single core systems, only one task at a time may run on a CPU.
- Multicore systems can run one task on each CPU simultaneously.
 - CPU affinity describes which task(s) prefer to run on which CPU
- A running task may yield the CPU and **possibly** allow another task to run
- A running task may block, in which case we always allow another task to run.



Task State: BLOCKED

- Sometimes referred to as a “suspended” or “pending” state.
- Waiting for an OS event, mutex, semaphore, timer, etc.
- Whenever the OS resource we are blocked on is released, the blocked task is made eligible again via the ready list.
- Unblocking a task does not make this task run automatically.



Task State: READY

- All tasks in the system that are not blocked might be scheduled to run at any point in time.
- The OS maintains a list of tasks that **could** be run at any point in time in a sorted ready list.
- Each time an OS kernel call is made, the OS might reschedule tasks.
- When the kernel determines that a task switch is needed, the head of the ready list is popped and becomes the currently running task.



What About Tasks that are Interrupted

- Only the currently running task may be interrupted.
- When an interrupt handler returns, it can either:
 - Return to the running task directly (no state transitions)
 - Return via the OS scheduler with a different task running.
- Inside an interrupt handler:
 - We may -not- call any OS functions that block or yield.
 - We may call OS functions that signal a task to become unblocked.
 - Task state transitions only occur after all nested interrupts return.
 - Only need to reschedule once.

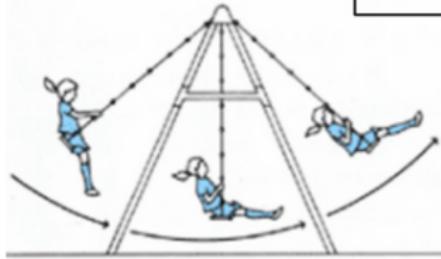
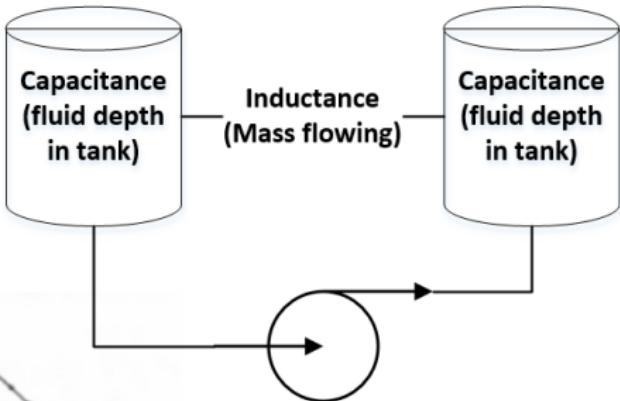
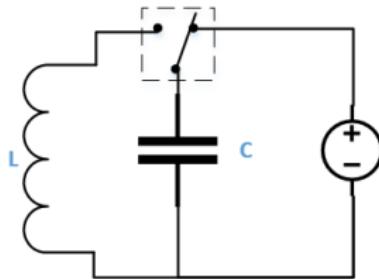


ECEN 3753: Real-Time Operating Systems

Capacitive Sensing

Capacitive Analogies

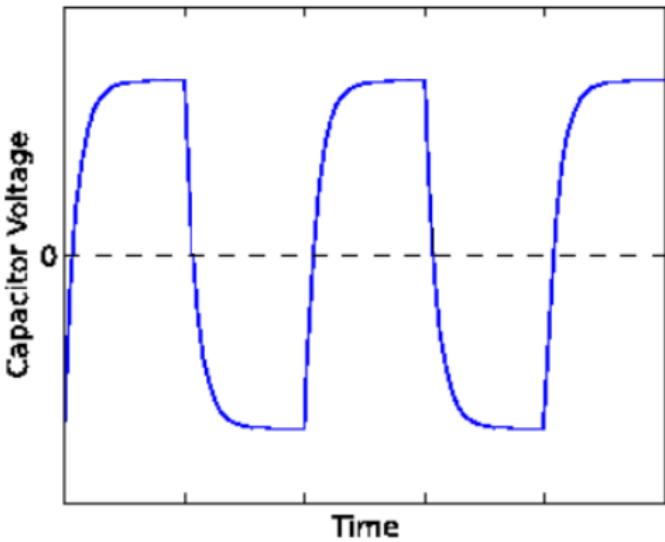
$$f = \frac{1}{2\pi\sqrt{LC}}$$



Swing picture from <http://www.puremusic.com/90cc2.html>

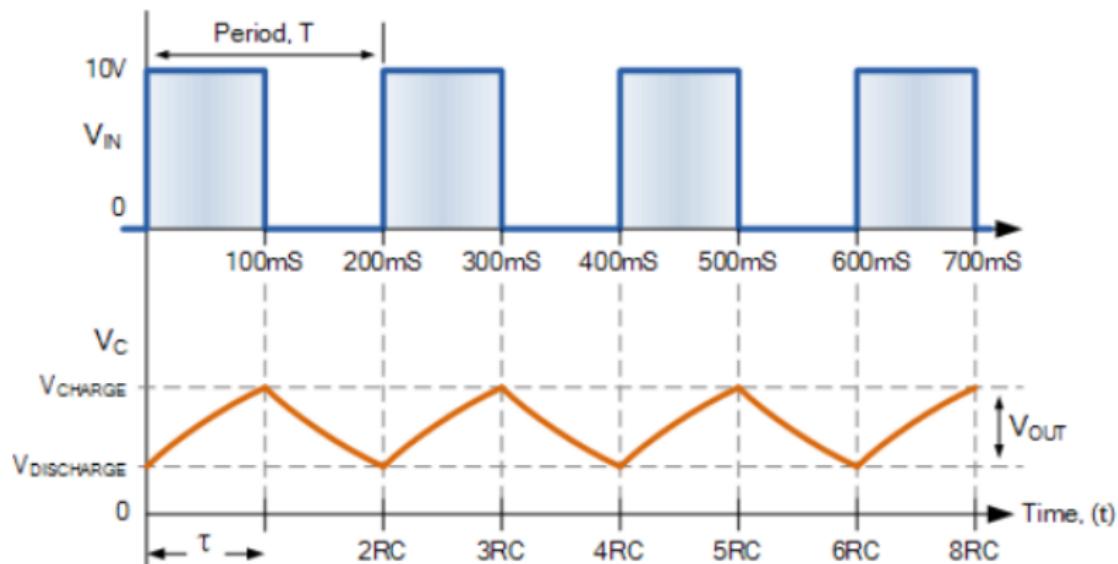


Saturating RC Oscillation



Derived from: <http://webpages.ursinus.edu/tcarroll/phys112/labs/node7.html>

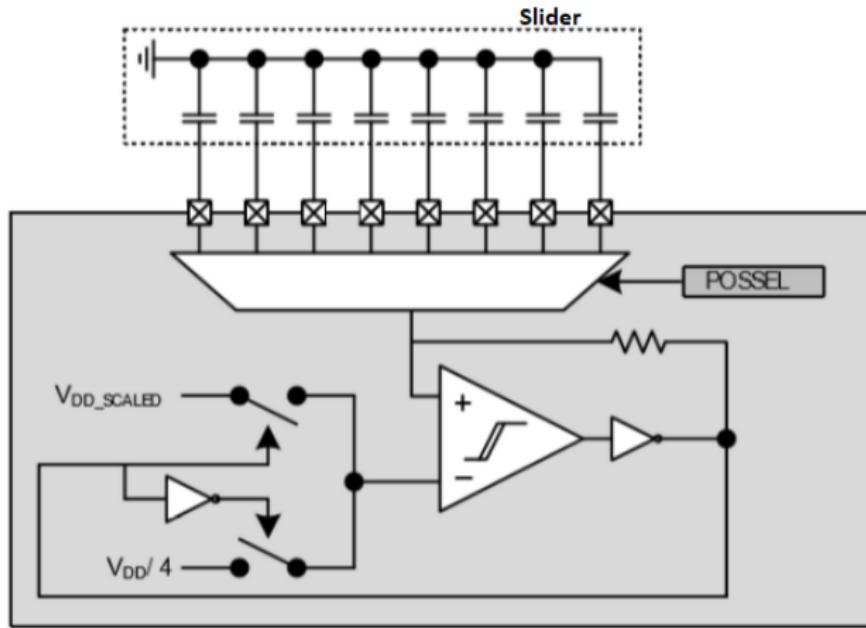
Self-adjusting RC Oscillation



Picture from: <https://www.electronics-tutorials.ws/rc/rc-integrator.html>



EFM-32 CapSense



Derived from: https://www.silabs.com/community/mcu/32-bit/knowledge-base.entry.html/2015/09/18/how_can_i_implement-hcxv



Low-Power was Key for the SDK Examples

It's clear that SiLabs wanted to show off their power miserliness, and so didn't consider the concurrent use of CapSense and Micrium OS in their firmware examples.

- EFM32PG12
 - Possibly only two timers exist: T0 and T1 (unclear if WideTimers reuse the 16-bit timers)
 - Can use timer interrupts to exit low-power mode
- CapSense, as written
 - T0: ($\text{SomeClock}(?) / 512$) for time reference. Interrupt when > 10 ticks
 - What is the value of this delay? Measure it in SDK example!
 - T1: counts free-running oscillator, $/1024$
 $(T1 \leftarrow \text{PRS}[0] \leftarrow \text{ACMP1.PosEdge})$ up to at least 64Ki counts
(assuming at least a 16b register)
- Micrium OS
 - T0 is the timer used for OS_tick
 - Idle task generally goes into low power (not needed for this



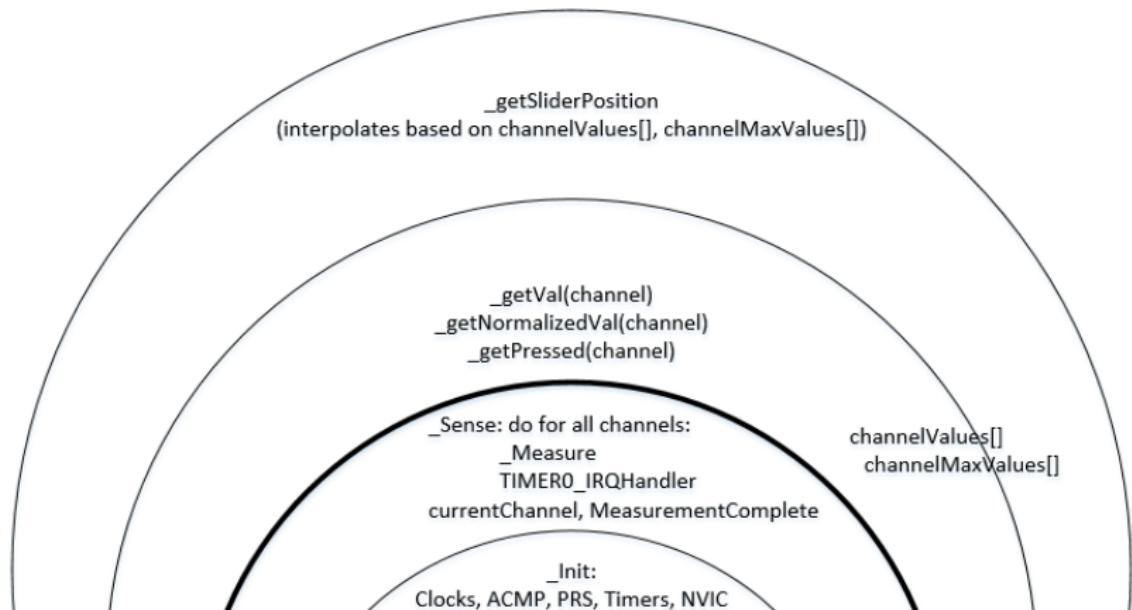
When Using Micrium's Timer Services

OS_tick Resolution: If we choose a magnitude that is:

- Too small: we'll have a precise and more accurate timebase, but will potentially spend more time context-switching
- Too large: we'll either lose accuracy in our measured frequency, or need to poll to synch up to the timer.



capsense.c



CAPSENSE prefix removed from function names, for brevity.



What's with the MaxValues per channel?

In-field calibration!

- The highest frequency every seen in each channel (since power-on) is retained and used for normalization of ongoing measurements of oscillation frequencies
- 75% of max-seen on a channel is used as that channel's "touch threshold" in CAPSENSE_touch().
- Silicon Labs uses a 32b variable, in spite of appearance that the underlying timer is 16b.



SDK's Touch Project: main()

main() in this project waited 100ms, did a CAPSENSE_Sense() cycle, printed results, repeat.

10Hz-ish seems to be sufficiently-frequent capacitive sensing to provide reasonable value in this course.



Desired Compatibility with Micrium

- Set up a reasonable OS clock frequency to supplant T0's use in capsense.c
 - T0 should NOT be touched by _Measure or _Init anymore.
- Move _Sense and T0's ISR data collection into a task that waits the “long time” (e.g. 100ms) to start sequentially working through all channels (the delay from main()), waiting appropriate OS_ticks to count oscillations on each channel.
 - No more ISR.
 - Priority? (High, at least when sensing a single channel's capacitance)
 - Varying? (Esp if there is other important work to do between channel sensings.)
 - Power? (Don't care too much. Maybe even leave ACMP enabled when it's not needed, and probably avoid low-power CPU modes (force the idle task to stay OUT of low power modes).)



Investigation Grit

Often, our plans are thwarted by details.

SMOP = Small Matter Of Programming

- Silicon Labs appears to have recently refactored their online docs (<https://www.silabs.com/developers/micrium>)
 - Micri μ m and μ C/OS docs are available at the top level now
 - Micri μ m half had broken links throughout when we were looking in the week of Feb 8, 2021.
 - μ C/OS half is not the same, but gave us leads.
- OS Clock appeared non-configurable in our Micri μ m source code (hard-configured in our port?)
- Prescaler couldn't be reduced (it was already 1 when we peeked from the debugger)
- TopValue of T0 couldn't be hacked to 1/N of the norm (TopValue was either not used, or was MAX_UINT—which could have been assumed throughout Micri μ m, so we viewed it as risky to muck with.)



Analysis and Use of OS timer

First of all, what time values are we trying to emulate?



What DID the documentation indicate?

Given that the documentation had lots of broken macros, and google search results merely pointed to the new top-level split in docs, it was challenging to find info about the OS tick.

- Micrium doc re: dynamic tick on MuCOS had an example where the OS tick rate was 100Hz
- Comment at embeddedcomputing.com (paraphrased):
 - Using uCOS as an example, on a 32-bit processor running at 300 MHz, the overhead required for the kernel to process 1000 ticks per second would likely not exceed 1 percent of the CPU's cycles. (Implies that the timer could run at 1KHz.)
- Micrium's Kernel Time Management API doc: provides a getter, but not a setter of the tick rate.
- uCOS had a constant definition that could be set to change the OS_tick rate, but this was not found in the code that come with Simplicity, so we're guessing that the port that we have may be limited to the 1ms that we measured.



Q: could we live with 1ms OS_tick?

Recall that the “counting period” for T1 counting of oscillations was about 2.5ms

- Counting a 2.5ms interval with a 1ms timer would have introduced a LOT of error (1 tick would equate to 40 percent error counting a channel’s oscillation)
- What if the 24 ms became somewhat less than 100ms (if we allow summing over 10ms/channel instead of 2.5ms/channel)
 - We’d be limited to about 5 samples/s instead of 8-10, if we maintain the 100ms inter-sampling period.
 - T1 would accumulate 4x as many counts. Are there enough bits in it?
 - Debug: 85 or so for the max counts with the non-Tasking CapSense implementation. x4 would almost fit in a uint8, and we have either a uint16 or uint32 register. Variables are already uint32.



A: Apparently, Yes we can

Students reported in that their experiments were slower at providing capsense data (as expected: should be sampling about 1/2 to 1/4 as fast), but were not flaking out after a bit of time (LEDs flashing, buttons seemingly ignored as they were previously.)



Post-analysis:

- There may be a way to adjust the OS_tick. That would be preferable in some situations, so that the finger position on the slider can be measured more frequently.
- Not sure if students experimented with a Delay() function that was built-up from a non-OS example, or OSTimeDly() (which is from the OS)
- Micri μ m's OSTimeTickRateHzGet() should be used whenever converting from units of time to ticks or vice-versa.
- Don't delay 100ms between channel samplings, if they're going to require nearly that long—perhaps simply always leave the new TaskingCapSense running (sleeping)
- If TaskingCapSense's priority is left low (therefore counts may vary more), add level of normalization before the Values/maxValues, to ensure those arrays are scaled by actual T1 periods. (Use OSTimeGet() around OSTimeDly() → task sleep duration)



ECEN 3753: Real-Time Operating Systems

Scheduling



Scheduling

- Why have scheduling?
 - Need ability to share resources between multiple Tasks
 - OS decides what to run next
 - Between decisions, OS needs to prepare the old and new Task
 - Context Switching
- How the scheduler chooses the next Task depends on the algorithm chosen
 - These scheduling decisions are regularly being applied



Scheduling Goals

- Meeting all of the following goals is difficult
- Goals:
 - Maximize throughput
 - Maximize how resources are utilized
 - Minimize response time
 - Minimize wait time
 - Minimize time it takes to start a new task
 - Share resources appropriately
 - What are some examples of resources that need sharing?



Criteria for Scheduling

- CPU Utilization
 - In order to have efficient use of CPU it needs to remain as busy as possible as long as there is work to do
- Response Time
 - Amount of time the task spends waiting to be executed
- Turnaround Time
 - Amount of time it takes from the time a Task was submitted until it finished completion
- Waiting Time
 - Amount of time a tasks waits for resources (CPU, Memory, etc)
- Throughput
 - The amount of “fully completed” work that can be completed in a specific amount of time. [TasksCompleted/Time]

Turnaround = Waiting + Execution.



Non Preemptive Scheduling

- Scheduling algorithm waits for the current Task to let go of the CPU (note: Task either terminates, yields or blocks)
 - Also known as Cooperative multitasking
- Why choose a non-preemptive scheduler?
 - What are the advantages?
 - What are the disadvantages?



Preemptive scheduling

- The OS forces the current Task to let go of the CPU
 - This allows another Task to resume or a new one to be started
 - Also known as preemptive multitasking
- Why choose a preemptive scheduler?
 - What are the advantages?
 - What are the disadvantages?
- What is a human-scale example of “good” line-cutting?



Context Switching

- Required for preemptive scheduling
- Likely required for non-preemptive if non-trivial or long-lived Tasks
- Allows the “pausing” of a Task
- Current Task’s information is stored for later retrieval
- Next Task’s information is then retrieved
- The time it takes to store and retrieve is known as **Context Switch Time**

Note similarities to Interruptability, Interrupt Latency from CPU context protection for ISR (Critical Sections)



Scheduling Algorithms

- First-Come, First-Served
- Shortest Job First
- Time Slice
- Time Slice with Background Tasks
- Round Robin
- Priority

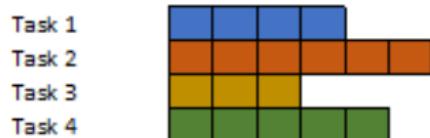


First Come First Served [FCFS]

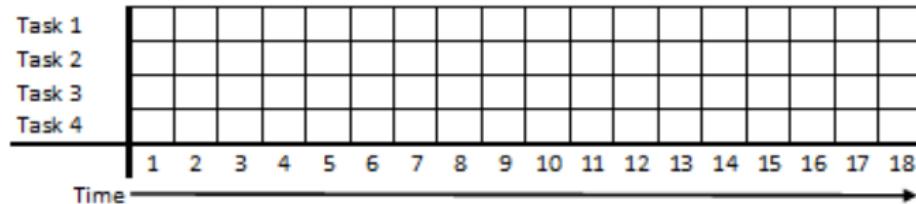
- Cooperative multitasking (generally)
- What Is It?
 - Also known as First-In, First-Out
 - Select the Task that is at the head of the ready queue FIFO
 - This scheduler is typically non preemptive
- Pros
 - FIFO is easy to implement
 - No context switch needed since the Task runs to completion
- Cons
 - If the first task is large, other tasks will be “stuck” behind it, resulting in low throughput



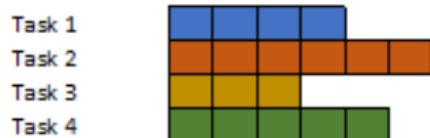
First Come First Served [FCFS] (continued)



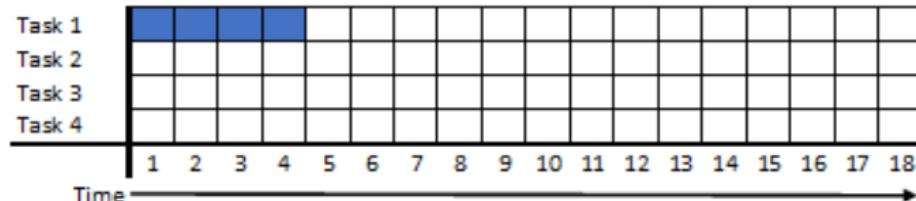
First Come First Served



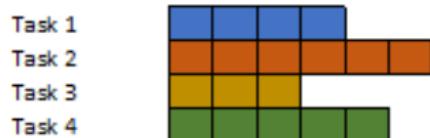
First Come First Served [FCFS] (continued)



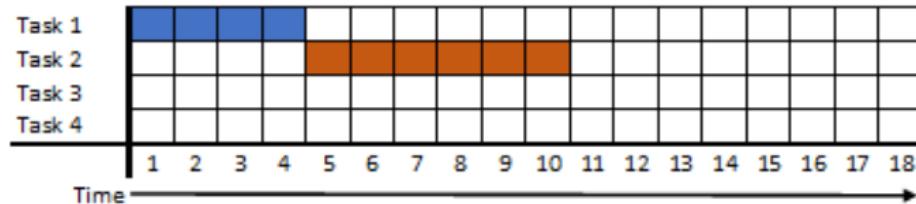
First Come First Served



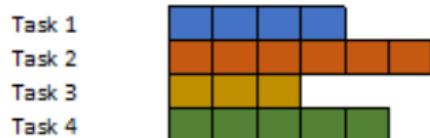
First Come First Served [FCFS] (continued)



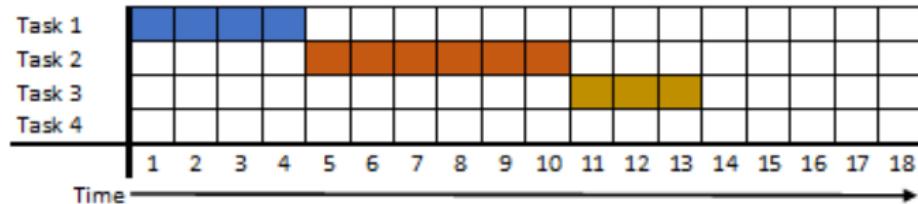
First Come First Served



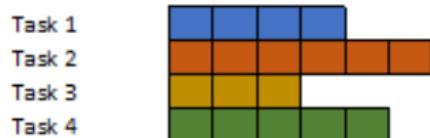
First Come First Served [FCFS] (continued)



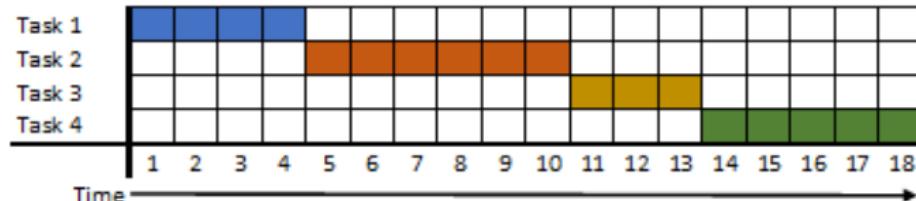
First Come First Served



First Come First Served [FCFS] (continued)



First Come First Served



First Come First Served Unit Tests

- Check if everything is initialized correctly
- Check if the first Task has a waiting time of zero
- Check if the second Task has a waiting time the same as the first Task's execution time, etc

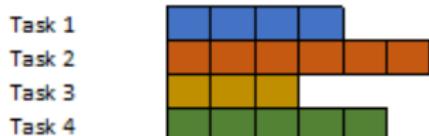


Shortest Job First [SJF]

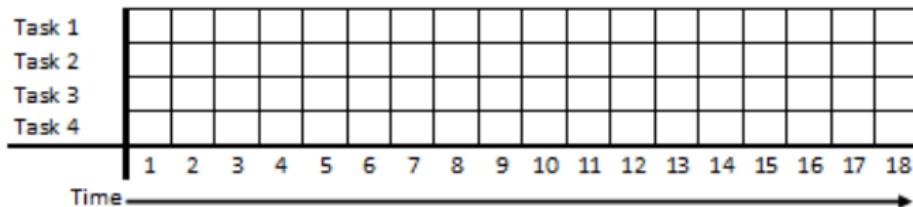
- What Is It?
 - Scheduler places jobs with shortest completion time as the highest priority
 - Non Preemptive:
 - Task with the shortest time gets to execute
 - Preemptive:
 - Task with the shortest remaining time gets to execute
- Pros
 - Maximizes Task throughput
- Cons
 - Larger Task potentially could never run, or at least suffer long latencies
 - Imposes hard requirements on knowledge in regards to time-complexity



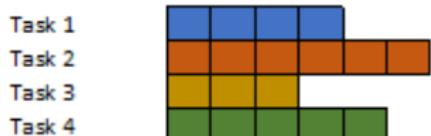
Shortest Job First [SJF] (continued)



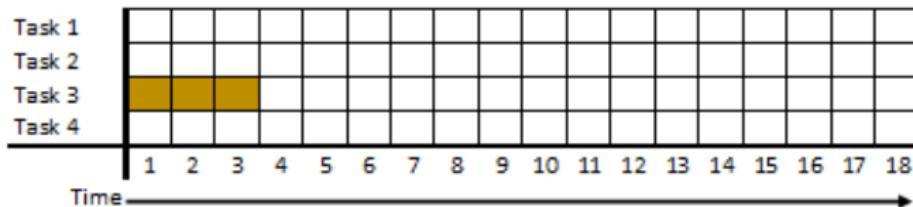
Shortest Job First



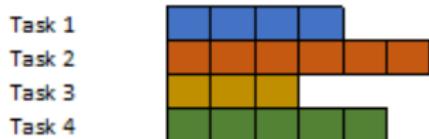
Shortest Job First [SJF] (continued)



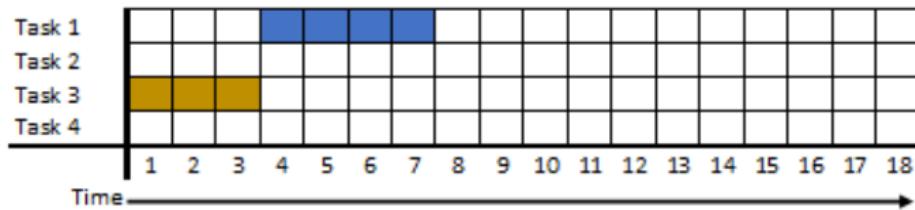
Shortest Job First



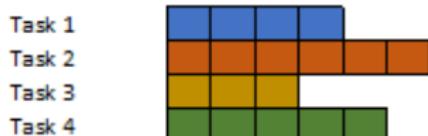
Shortest Job First [SJF] (continued)



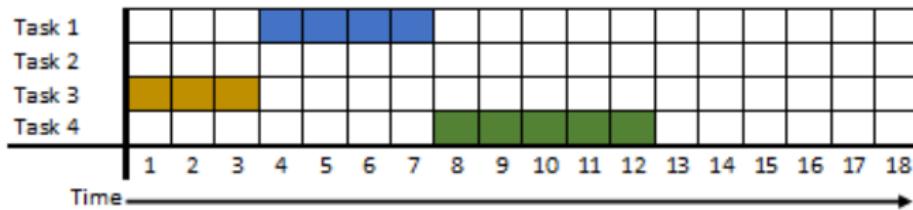
Shortest Job First



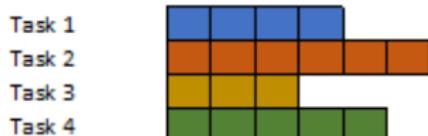
Shortest Job First [SJF] (continued)



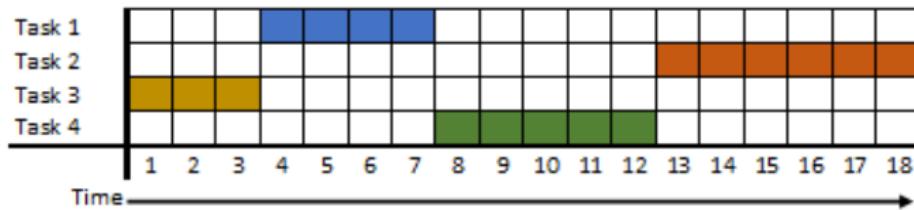
Shortest Job First



Shortest Job First [SJF] (continued)



Shortest Job First



Shortest Job First Unit Tests

- Check if everything is initialized correctly
- Check if the shortest Task has a waiting time of zero
- Check if the second shortest has a waiting time equal to the execution time of the first Task, etc

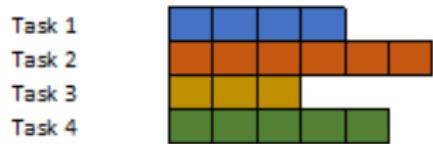


Time Slice [TS]

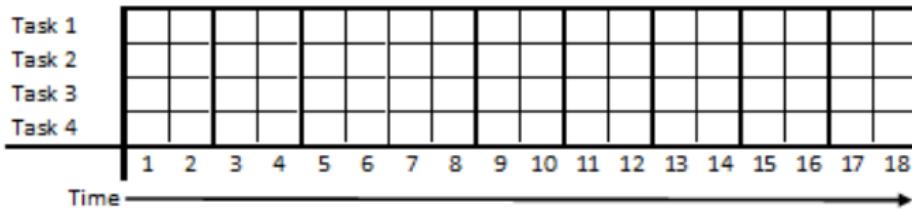
- Preemptive multitasking
- What Is It?
 - Time slots are created for each Task.
 - Once the “timer” ends an interrupt is thrown for the next task to start
- Pros
 - Fully deterministic
- Cons
 - If the Task has nothing to do during it's time then the slot time is wasted



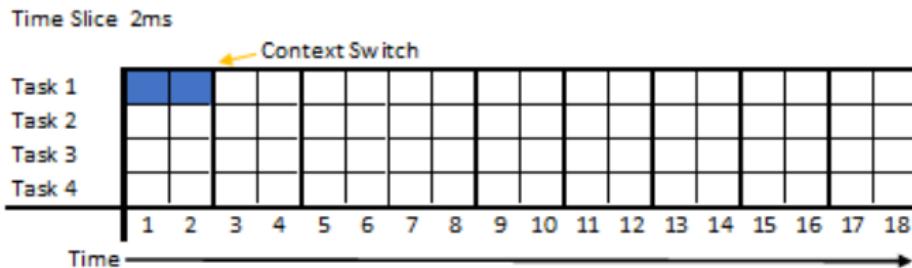
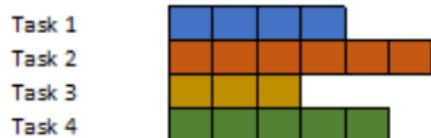
Time Slice (continued)



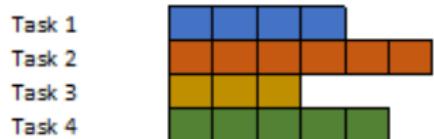
Time Slice 2ms



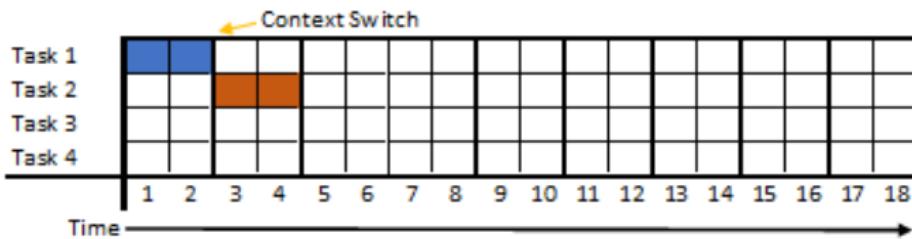
Time Slice (continued)



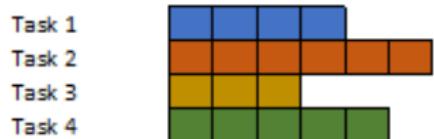
Time Slice (continued)



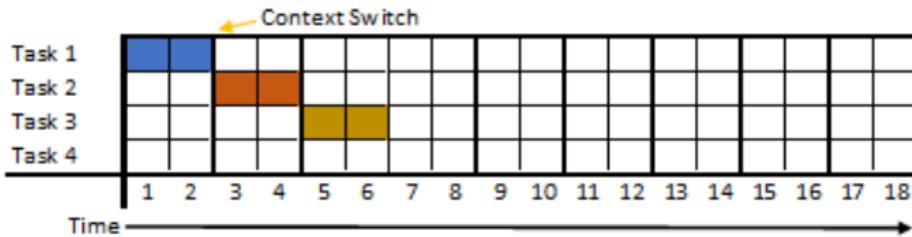
Time Slice 2ms



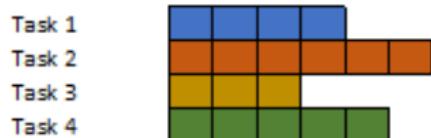
Time Slice (continued)



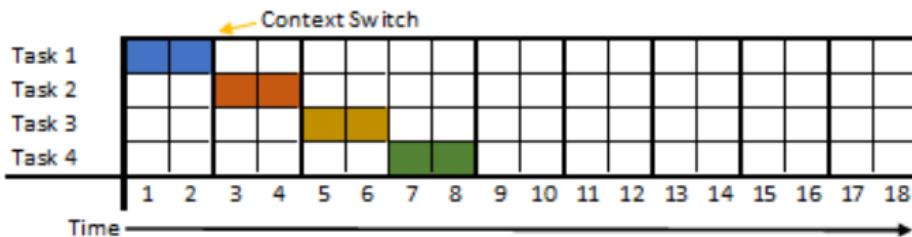
Time Slice 2ms



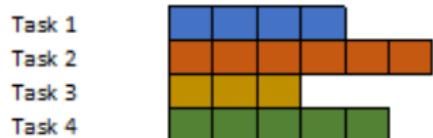
Time Slice (continued)



Time Slice 2ms



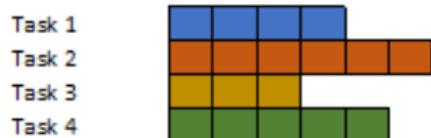
Time Slice (continued)



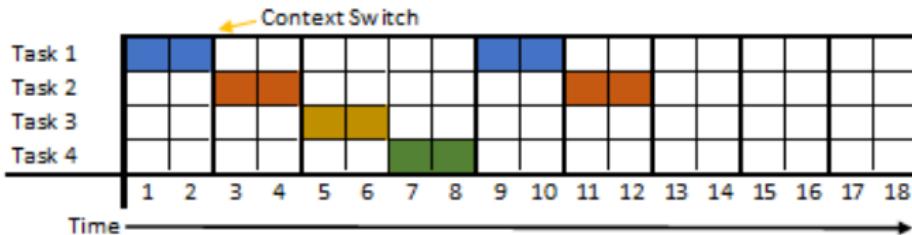
Time Slice 2ms



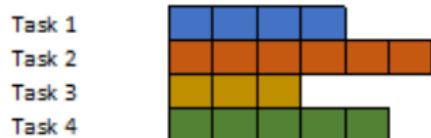
Time Slice (continued)



Time Slice 2ms



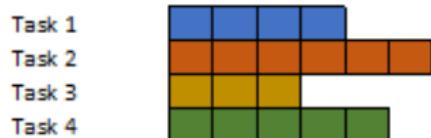
Time Slice (continued)



Time Slice 2ms



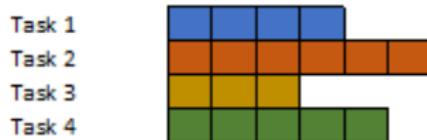
Time Slice (continued)



Time Slice 2ms

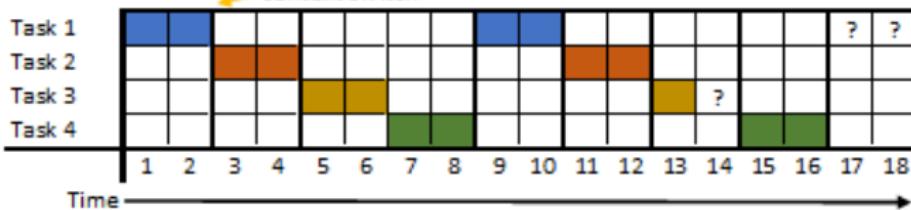


Time Slice (continued)



Time Slice 2ms

Context Switch



During '?':

- spin CPU and wait?
- suspend CPU to reduce power?
- Try to start other tasks sooner?



Time Slice Unit Tests

- Check if everything is initialized correctly
- Test that each Task runs for a max time each iteration
- Test with only 1 Task first then add more



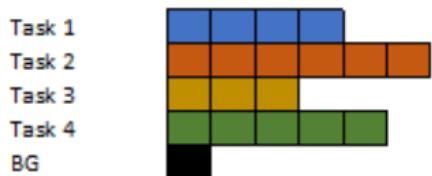
Time Slice with Background Task [TSBG]

- Preemptive multitasking
- What Is It?
 - Time Slice but adds in ability to run background tasks if a slot is free
- Pros
 - Fully Deterministic
 - Improves Time Slice
- Cons
 - There is a chance that background tasks will never be scheduled
 - Overhead due to context-switching

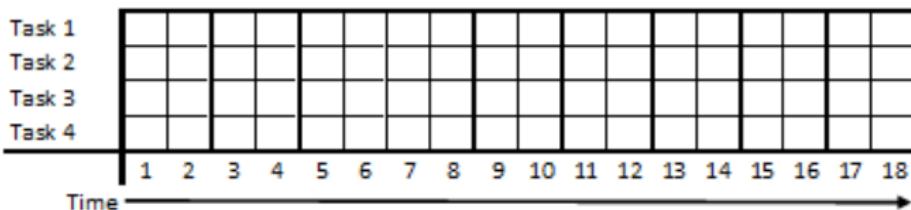
Real-life example: HDD, early 1990s: servo on timer interrupt, then R/W channel/sequencer, then background (host/queueing)... degenerate “preemption”



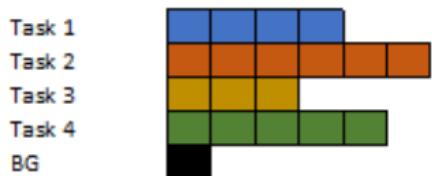
Time Slice with Background Task [TSBG] (continued)



Time Slice 2ms

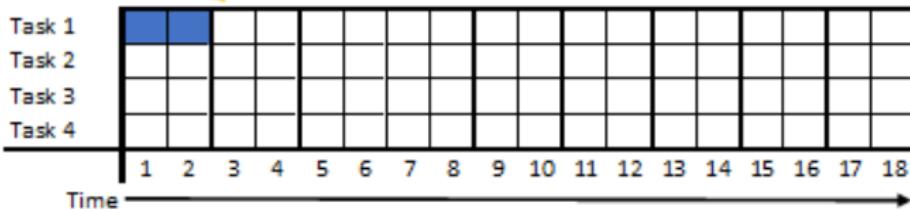


Time Slice with Background Task [TSBG] (continued)

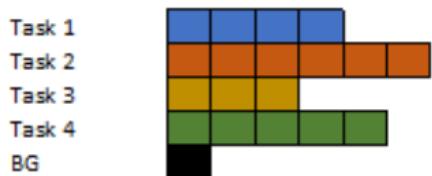


Time Slice 2ms

Context Switch

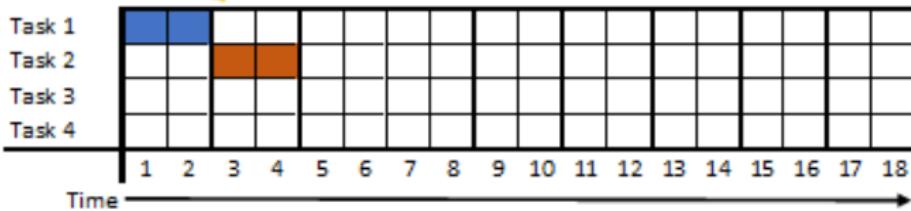


Time Slice with Background Task [TSBG] (continued)

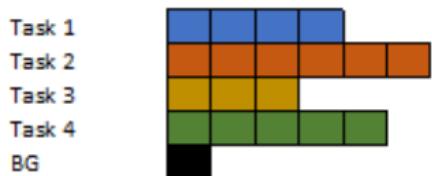


Time Slice 2ms

Context Switch

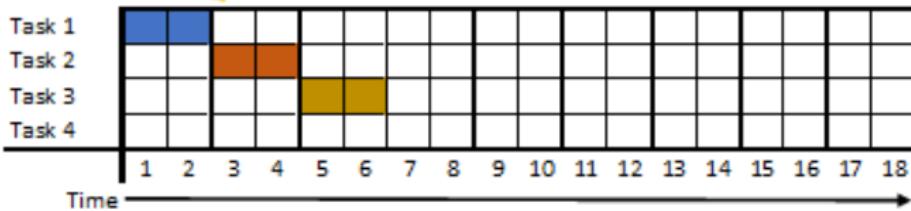


Time Slice with Background Task [TSBG] (continued)

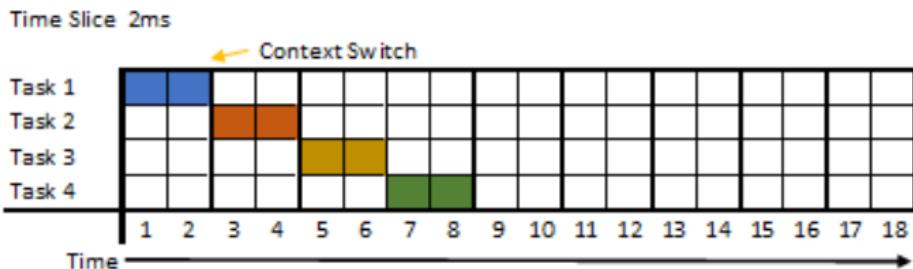
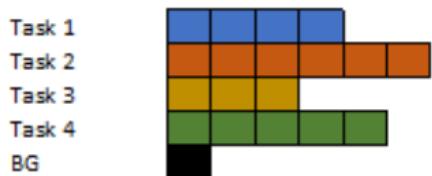


Time Slice 2ms

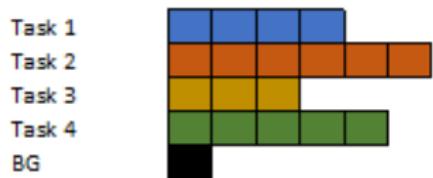
Context Switch



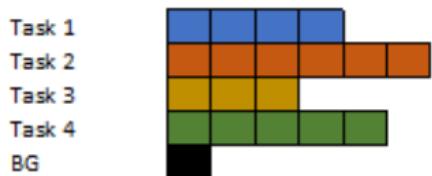
Time Slice with Background Task [TSBG] (continued)



Time Slice with Background Task [TSBG] (continued)

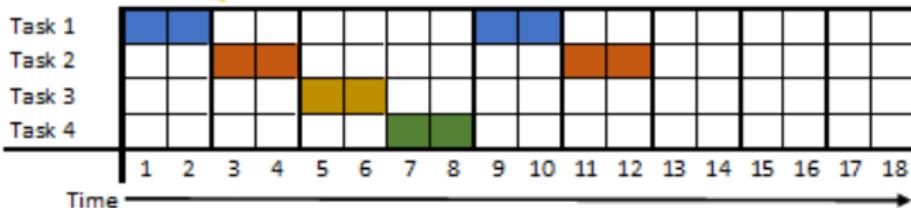


Time Slice with Background Task [TSBG] (continued)

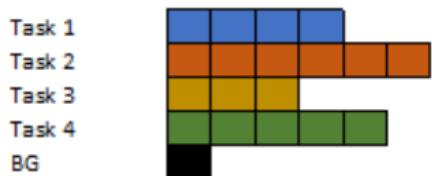


Time Slice 2ms

Context Switch

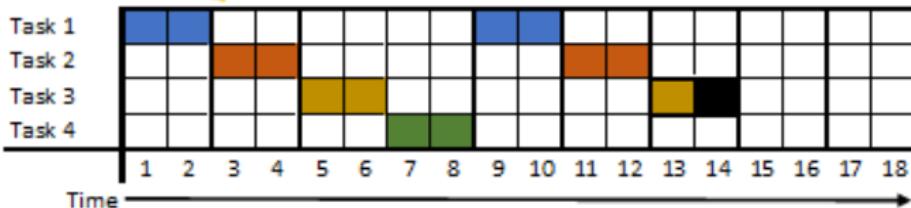


Time Slice with Background Task [TSBG] (continued)

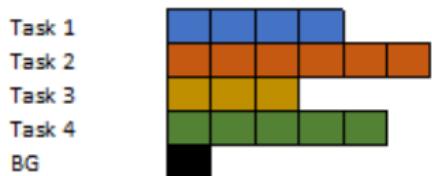


Time Slice 2ms

Context Switch

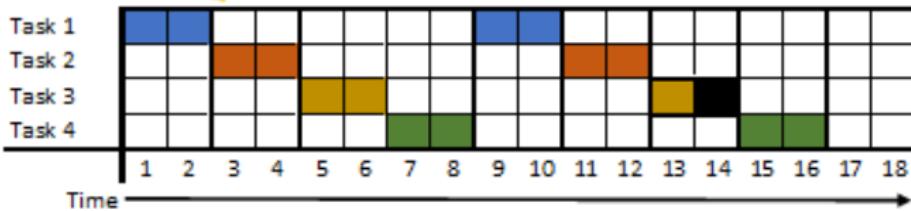


Time Slice with Background Task [TSBG] (continued)

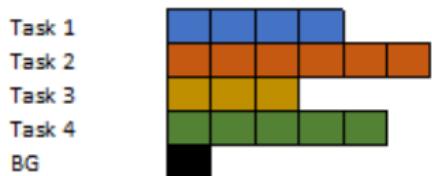


Time Slice 2ms

Context Switch

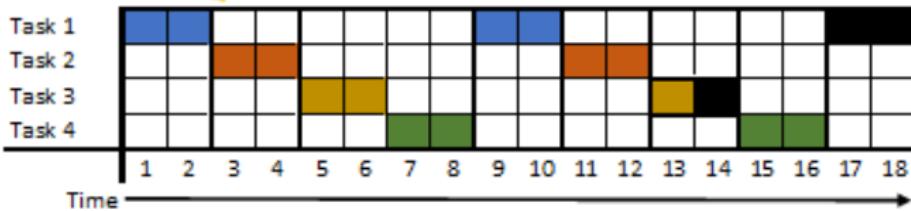


Time Slice with Background Task [TSBG] (continued)



Time Slice 2ms

Context Switch



Time Slice with BG Unit Tests

- Check if everything is initialized correctly
- Test that each Task runs for a max time each iteration
- Check that a background Task runs when a Task runs out of things to do



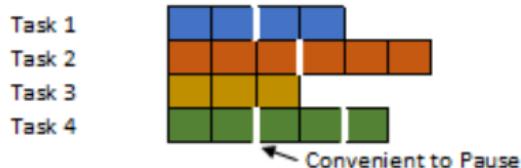
Round Robin [RR]

- Cooperative multitasking
- What Is It?
 - Strictly-ordered execution of Task segments
 - Typically the ready queue (FIFO) is a circular queue
 - Each Task gets the resources for a specific amount of time
 - The Task runs until it blocks, finishes, or uses allotted time
- Pros
 - Great for small Tasks
- Cons
 - Highest priority task is not run immediately

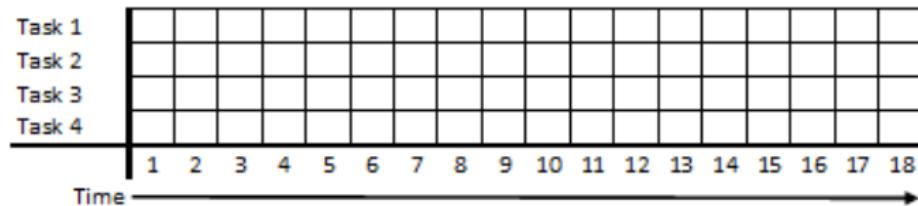
Real-life use: Network router with heavy hardware pipelining for execution context pre-fetch



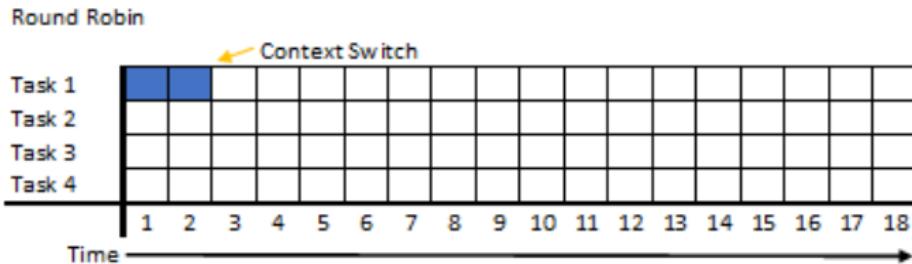
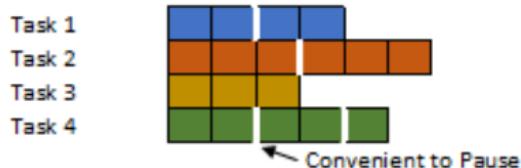
Round Robin [RR] (continued)



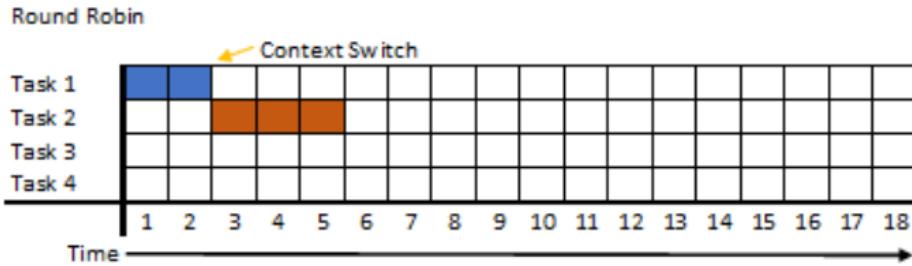
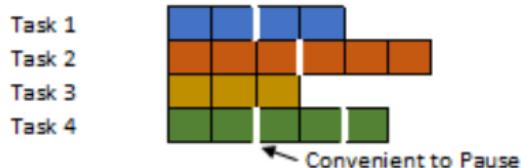
Round Robin



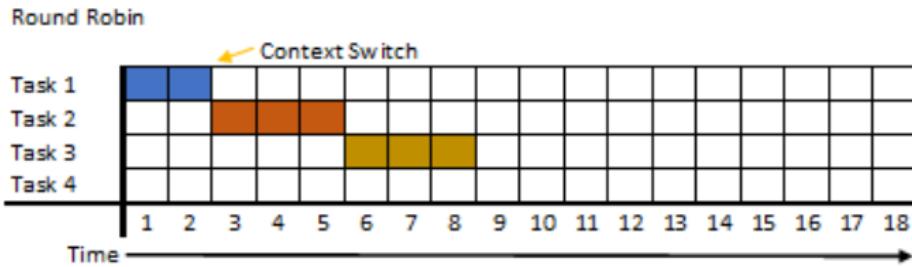
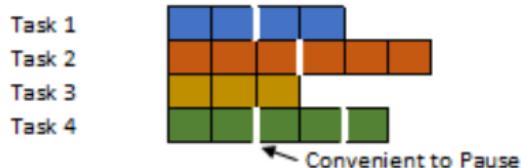
Round Robin [RR] (continued)



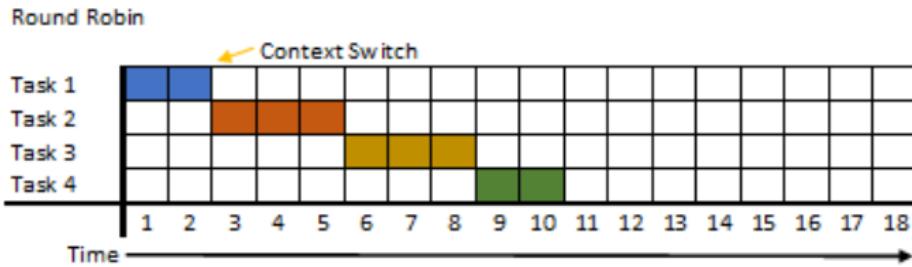
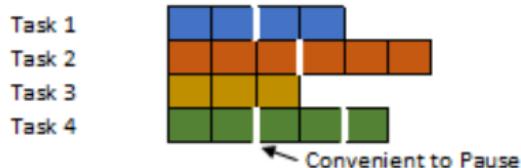
Round Robin [RR] (continued)



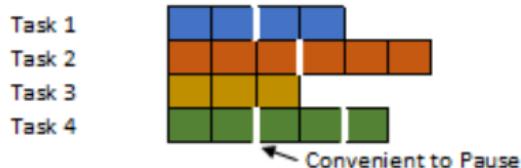
Round Robin [RR] (continued)



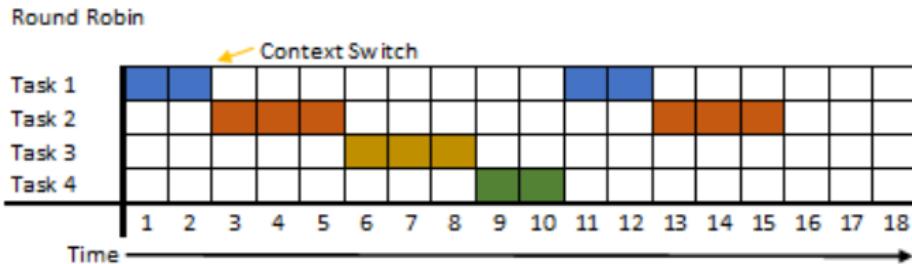
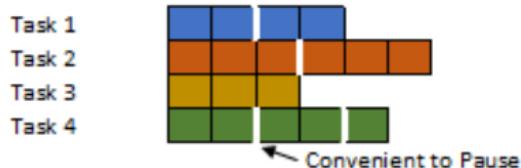
Round Robin [RR] (continued)



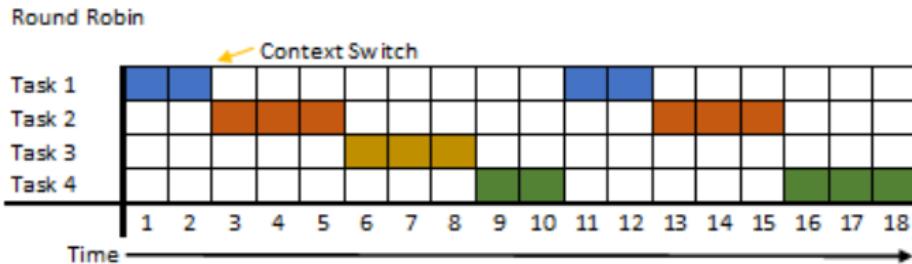
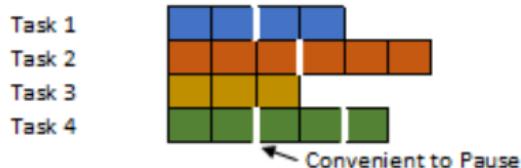
Round Robin [RR] (continued)



Round Robin [RR] (continued)



Round Robin [RR] (continued)



Round Robin Unit Tests

- Check if everything is initialized correctly
- Verify that each Task “pauses” during specific times
- Verify that each Task finished execution
- Verify that each Task didn’t hog resources and therefore finish early

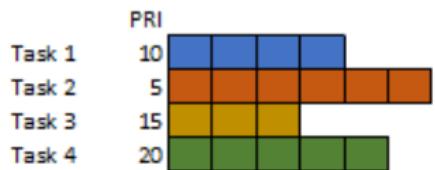


Priority

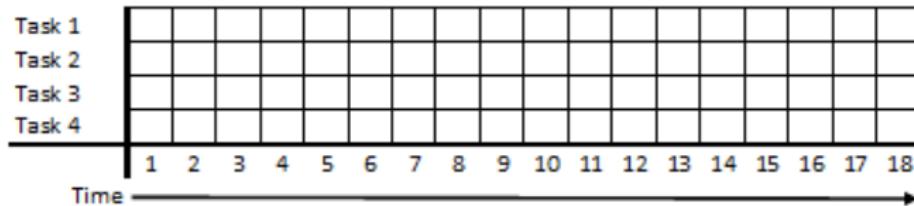
- What Is It?
 - Task is chosen with the highest priority
 - Priority can be based on memory requirements, CPU Time, Owner, etc
 - Can be non-preemptive or preemptive
 - Note: priorities can change during execution
- Pros
 - Highest priority jobs are run
- Cons
 - Low priority jobs can potentially wait forever
 - Overhead due to context-switching



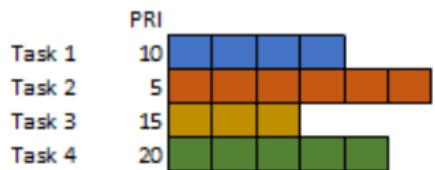
Priority (continued)



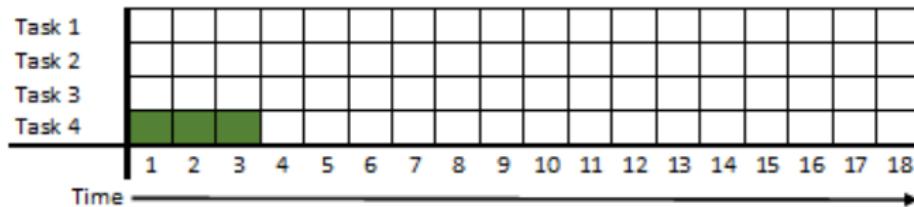
Priority



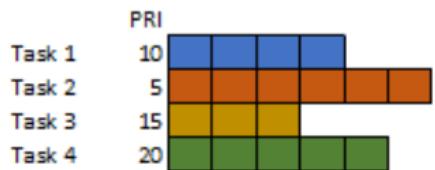
Priority (continued)



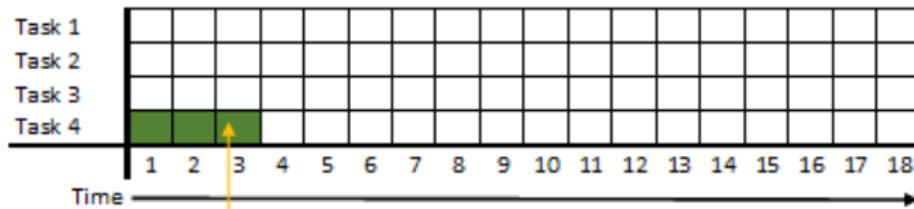
Priority



Priority (continued)



Priority



Priorities Changed

Task 1 10

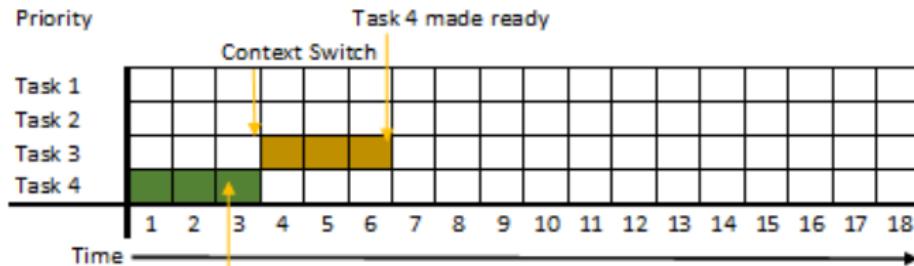
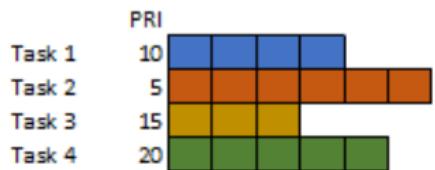
Task 2 5

Task 3 30

Task 4 20



Priority (continued)



Priorities Changed

Task 1 10

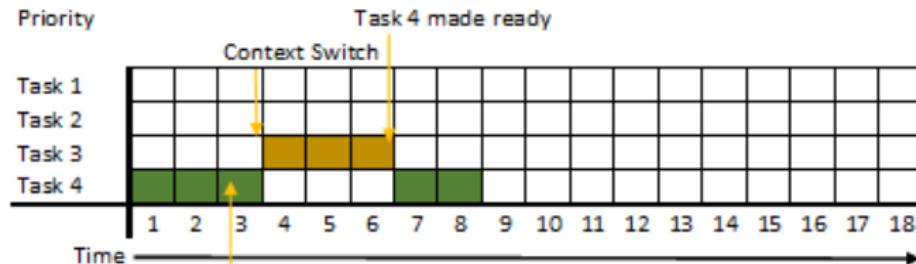
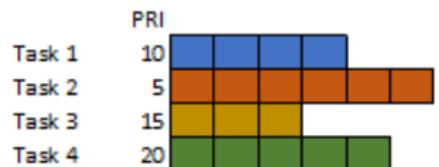
Task 2 5

Task 3 30

Task 4 20



Priority (continued)



Priorities Changed

Task 1 10

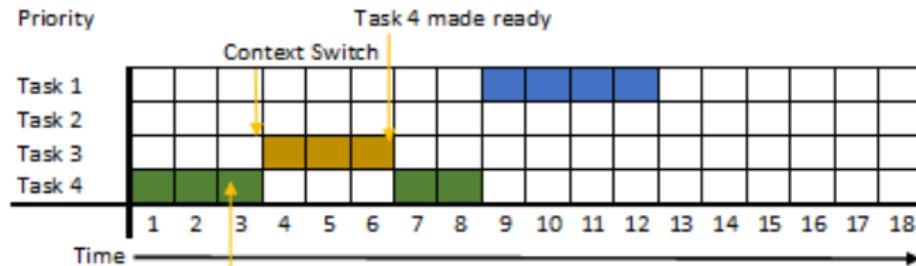
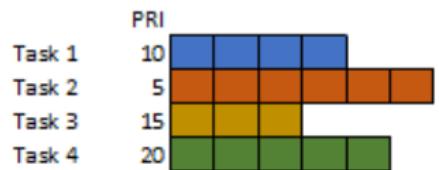
Task 2 5

Task 3 30

Task 4 20



Priority (continued)

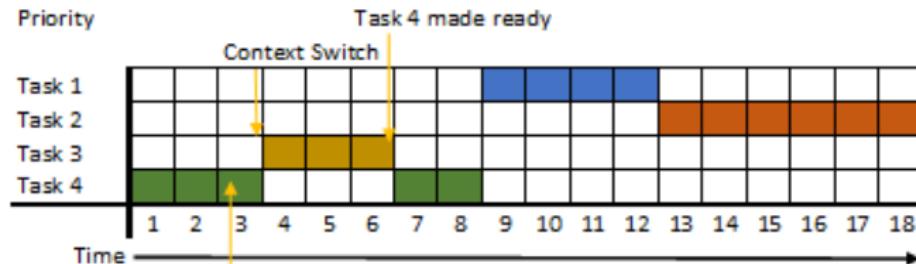
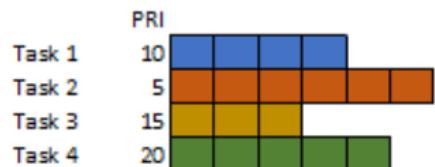


Priorities Changed

Task 1	10
Task 2	5
Task 3	30
Task 4	20



Priority (continued)



Priorities Changed

Task 1	10
Task 2	5
Task 3	30
Task 4	20



Priority Unit Tests

- Check if everything is initialized correctly
- Verify that the highest priority Task is always picked next
- Verify all Tasks finished execution
- Verify that new higher priority Tasks are executed in the right order



Scheduling

- Which Scheduling algorithm should you use?



Scheduling

- Which Scheduling algorithm should you use?
- It Depends.



ECEN 3753: Real-Time Operating Systems

Memory Management



Memory Management Topics

- Types of Memory
- Caching
- Heap vs. Stack
- Static vs. Dynamic Allocation
- Debug Stack Issues
- Address Heap Issues
- Memory Protection Unit
- Memory Management Unit
- Virtual Memory



Types of Memory

- Volatile: lose info on power off
 - Random Access Memory (RAM), Electrical read/write data
 - DRAM (Dynamic RAM)
 - SRAM (Static RAM)
- Non-Volatile: retain info on power off
 - Info not erasable
 - Mask Programmable Read-only Memory (ROM)
 - Info written electrically but not electrically erasable
 - Erasable Programmable ROM (EPROM)
 - Info written and erased electrically
 - Electrically Erasable Programmable ROM (EEPROM)
 - Flash (NAND, NOR)

SRAM vs DRAM

	SRAM	DRAM
Memory Density	Low	High
Cost per Bit	High	Low
Speed	Fast	Slow
Power Consumption	Low	High
Reliability	Good	Bad

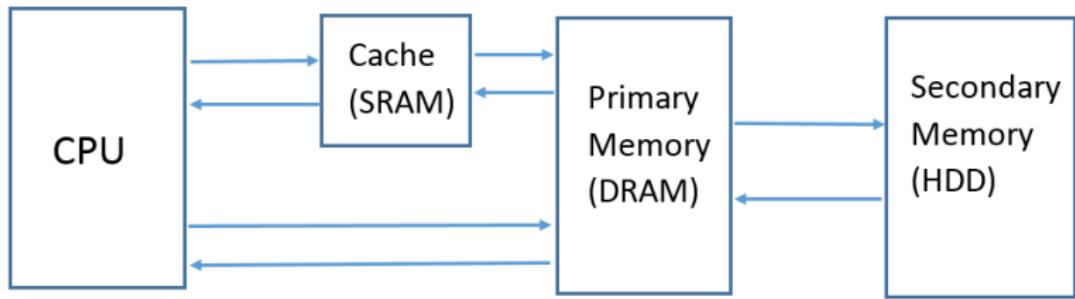
<modified slightly from Cooling, Table 6.1>

Data Reliability (one view of it...)

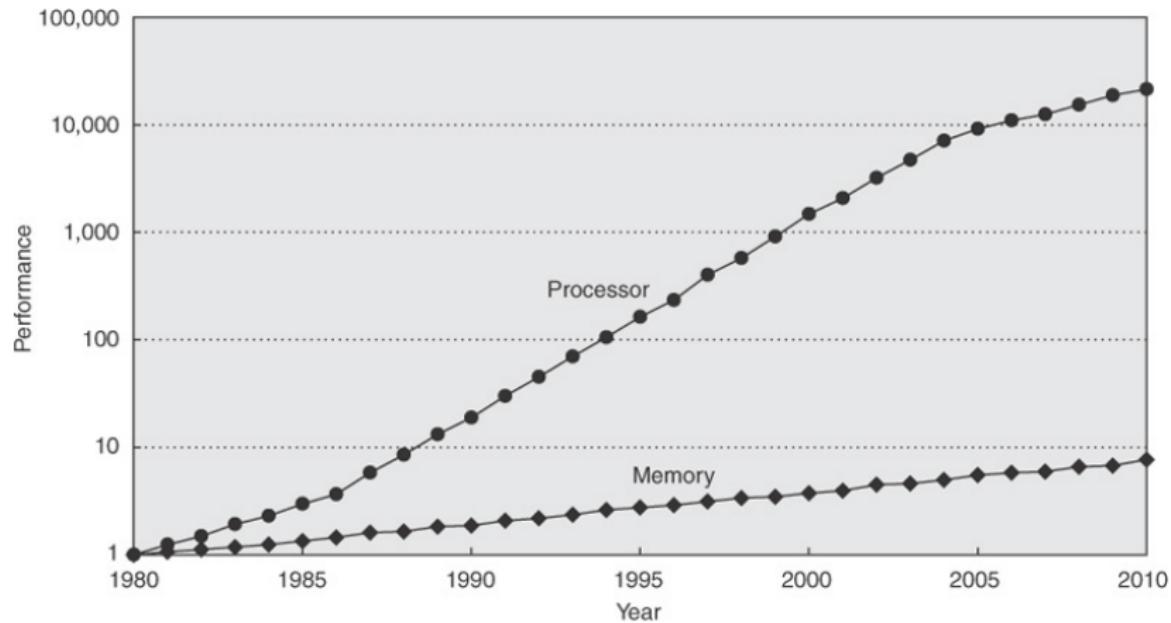
- SRAM data won't decay, so no data protection is needed
- DRAM data may decay, so data quality protection is needed



Basic Memory Structure: example



CPU time vs Memory access time



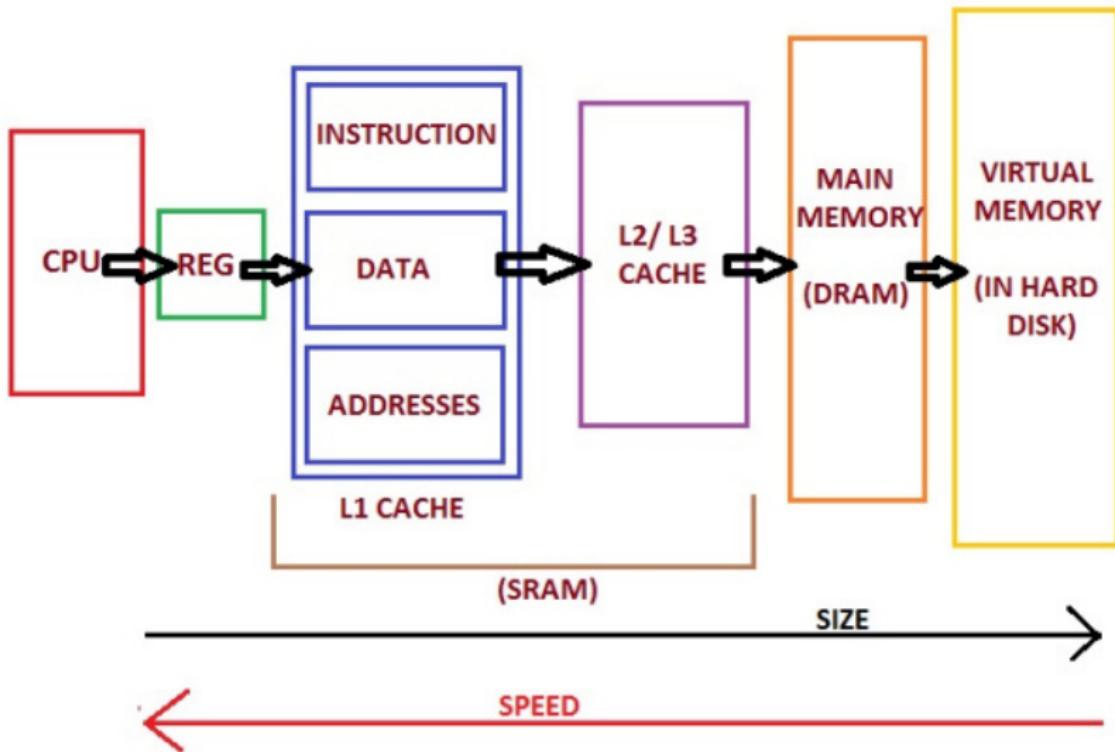
© 2007 Elsevier, Inc. All rights reserved.

<source: <https://www.extremetech.com/computing/261792-what-is-speculative-execution>>

- How can we fill the gap? Cache, more levels of cache



Memory Structure w/ Caching



Memory Access time

System Event	Actual Latency	Scaled Latency
One CPU cycle	0.4 ns	1 s
Level 1 Cache access	0.9 ns	2 s
Level 2 Cache access	2.8 ns	7 s
Level 3 Cache access	28 ns	1 min
Main memory access (DDR DIMM)	~100 ns	4 min
NVMe SSD I/O	~25 us	17 hrs
SSD I/O	50-150 us	1.5-4 days
Rotating HDD	1-10ms	1-9 months
Internet call: SFO to HKG	141 ms	11 years

<distilled from: <https://www.prowesscorp.com/computer-latency-at-a-human-scale>>



Caching Basics

- Caching works by two principles:
 - Temporal locality:
 - if a program accesses one memory address, there is a good chance that it will access the same address again.
 - Spatial locality:
 - if a program accesses one memory address, there is a good chance that it will access other nearby addresses.

Where are examples, below?

```
sum = 0;  
for (i = 0; i < Size; i++) {  
    sum = sum + a[i];  
}
```



Caching basics

- Cache Hit
 - the cache contains the data/code that the program is looking for
 - good: data/code is accessed faster in cache than main memory
- Cache Miss
 - the cache does not contain the data/code that the program is looking for
 - bad: CPU has to wait for the data/code from slower main memory
- Goal for HW and SW engineer: Improve Cache Hit Rate
 - Speculative execution pipeline
 - Branch taken/not-taken variants
 - Coding choices (loops, data structures)



How can a RealTime system POSSIBLY tolerate caching?

After all, doesn't this add variability that is not predictable?



How can a RealTime system POSSIBLY tolerate caching?

After all, doesn't this add variability that is not predictable?

- Cache pages can be locked in memory for Hard/Firm RT tasks
- Write back/through options to decache writes can be handled via another bus
- Gross speed gains from caching can buy considerable margin, so that MOST of the time, Firm/Soft RT demands are much more easily fulfilled
- Hard/Firm demands can be further supported by dedicated hardware (we'll get to that in DMA and Multi-CPU lectures)

But yes, it needs to be **carefully** analyzed.



Write Cache Types

Write Back:

- Postpones the write to main memory until need or good opportunity

Write Through:

- Simpler logic (no “Dirty” flag), but ties up back-end of write pipeline to main memory immediately.
- Doesn’t inherently protect multi-processor system.



Stack

- LIFO (last in, first out)
- Limit on size (specified when task is created)
- Managed efficiently by CPU
- Store local variables
- Stack usage grows and shrinks (as functions push and pop variables)



Stack Overflow

- How it happens:
 - Long function call path
 - Recursive function
 - Large local variables (e.g. big array)
 - Overly stingy SoC planners
- How to debug (more to come)



Heap

- Variable can be accessed globally
- No limit on size
- Managed by programmer (allocating and freeing variable)
- (Relatively) Slower access
- Memory leak (allocated but not released by programmer)
- Fragmented (more to come)



Memory Allocation Type Comparison

	Static Allocation	Dynamic Allocation
Time	Performed at static or compile time	Performed at dynamic or runtime
Memory	Assigned to stack	Assigned to heap
Size	Size must be known at compile time	Size may be unknown at compile time
Order	First In, Last Out	No particular order of assignment



Code example on variable in Stack

```
// All the variables in below program
// are statically allocated.

void fun()
{
    int a;
}

int main()
{
    int b;
    int c[10];
    fun();
}
```



Code example on variables in Heap

```
int main()
{
    // Below variables are allocated memory dynamically.
    int *ptr1 = new int;
    int *ptr2 = new int[10];

    // Dynamically allocated memory is deallocated
    delete ptr1;
    delete [] ptr2;
}
```



Stack vs. Heap Comparison

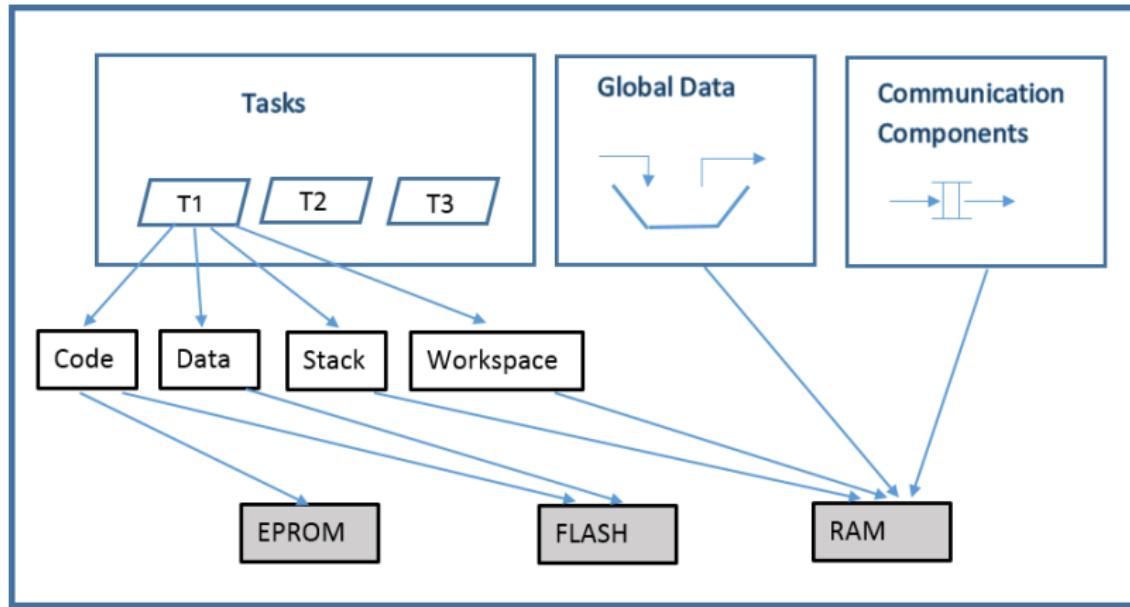
	Stack	Heap
Memory allocation	Contiguous block	Arbitrary order
Alloc/Dealloc Invoked	Auto by compiler	Manual by programmer
Alloc/Dealloc Time	Short	Long (and variable)
Access Time	Faster	Slower (indirect ref)
Locality of Reference	Excellent	Adequate
Flexibility	Fixed size	Data can be resized
Main Issue	Stack Overflow	Leakage and Fragmentation

The principles of locality mean that a processor tends to access the same set of memory locations repetitively over a relatively short period of time. Odds are somewhat better with Stack.

Heap may be used by multiple tasks—thereby requiring coordination (later in Shared Resources)



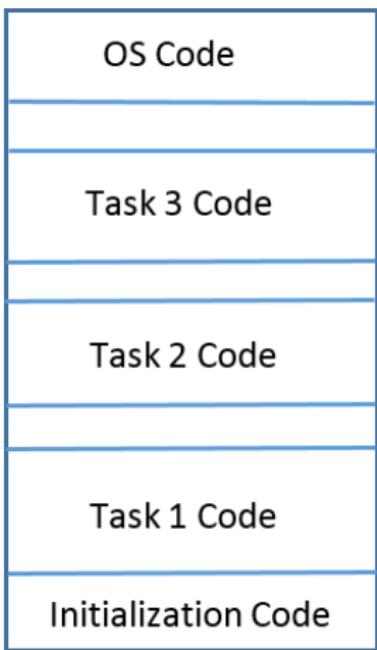
Task and Memory



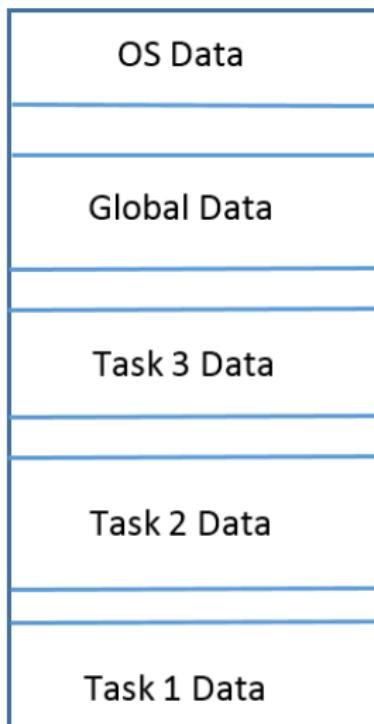
This picture shows the tie-together of Tasks, Memory, and multi-task access of global variables and needed inter-task coordinating methods.

Task and Memory, physical view

Memory Device 1



Memory Device 2



Those pesky problems with Stack and Heap

When comparing Stack and Heap, it was noted that:

Stack	Heap
Main Issue	Stack Overflow Leakage and Fragmentation



Stack issue: Stack Overflow

Recall in the “Task & Memory, physical view” picture, a task’s data was allocated adjacent to another task’s

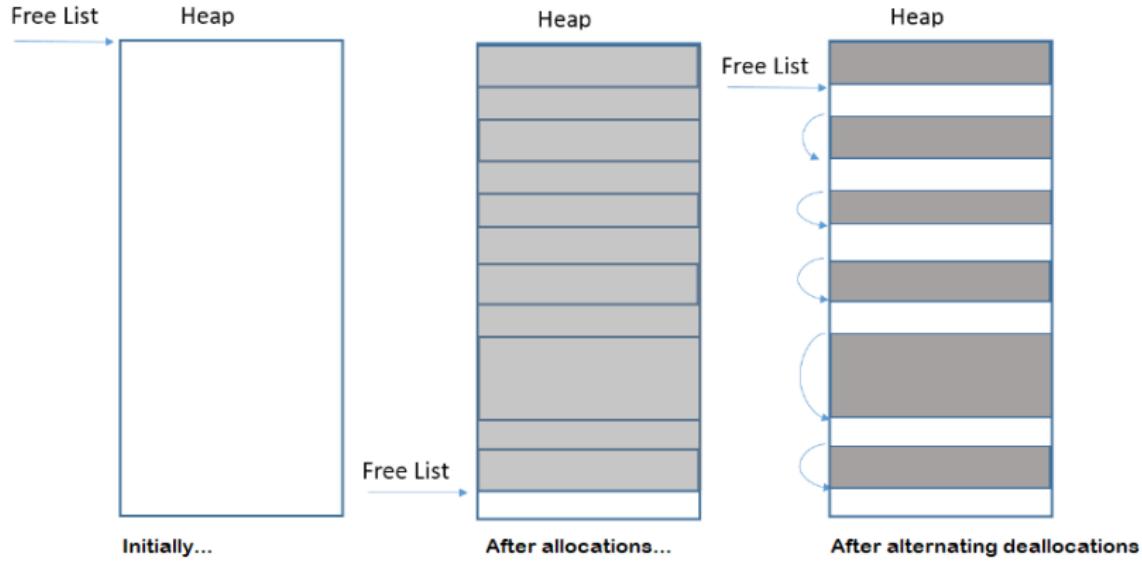
- When a stack overflows, one task’s data overwrites another task’s (or OS’s) data, which is **VERY** hard to debug
- Weak detection:
 - If there is gap between stacks
 - Fill gap w/ pattern, periodically check whether pattern changed
 - If there is not gap between stack, we can try to get lucky
 - Can run periodic checks (Stack data changing w/o code changing it)
 - Detect Invalid data value (e.g. invalid enum value)
- Robust detection:
 - Detect stack overflow in HW (Memory Protection Unit (MPU))

As long as stack overflow is prevented, the Stack solution looks pretty awesome compared to Heap!

Stack underflow is a bigger problem when people micro-manage the stack. Compilers are better at this than we are.



Heap issue: How Fragmented Memory Happens



(A very simplified example.)

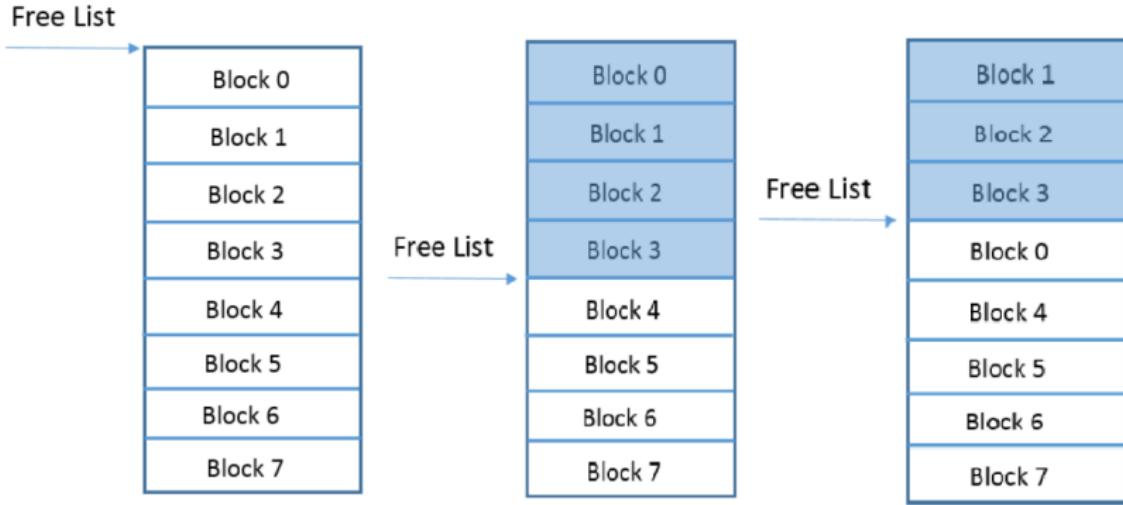


Fixing Fragmented Memory

- Defragmentation
 - feasible in theory, but generally not realistic in embedded systems (HIGHLY variable free/defrag times)
- Secure Memory Allocation
 - Heap memory is split into partitions (or sections)
 - Memory is allocated from selected partition
 - Only one block is allocated for each request
 - Deallocated memory is always returned to the partition it came from
 - Some RTOS implementations will put a “canary page” between blocks (which can be programmed into an MPU)



Secure Memory Allocation



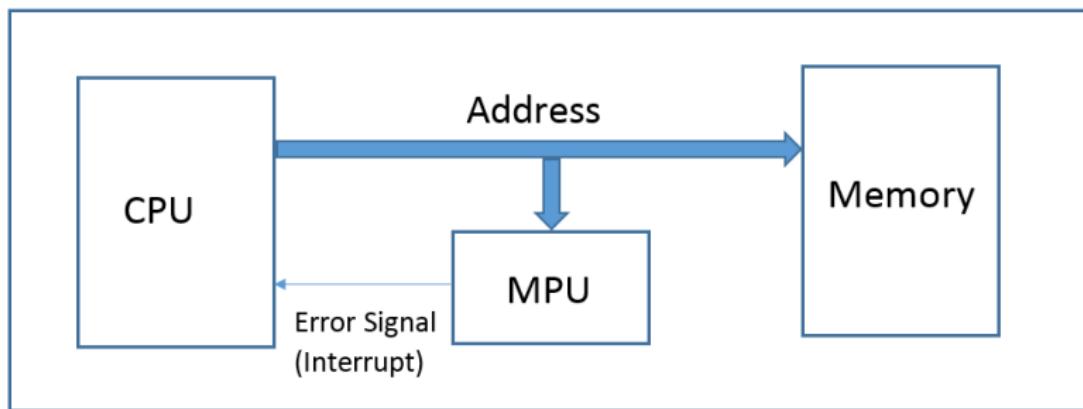
<from Cooling, section 6.4.3>

What are some advantages/disadvantages to returning to the start/end of free list?



Memory Protection Unit (MPU)

- Monitor the address info flowing between CPU and Memory
- Signal error (exception) for any violation



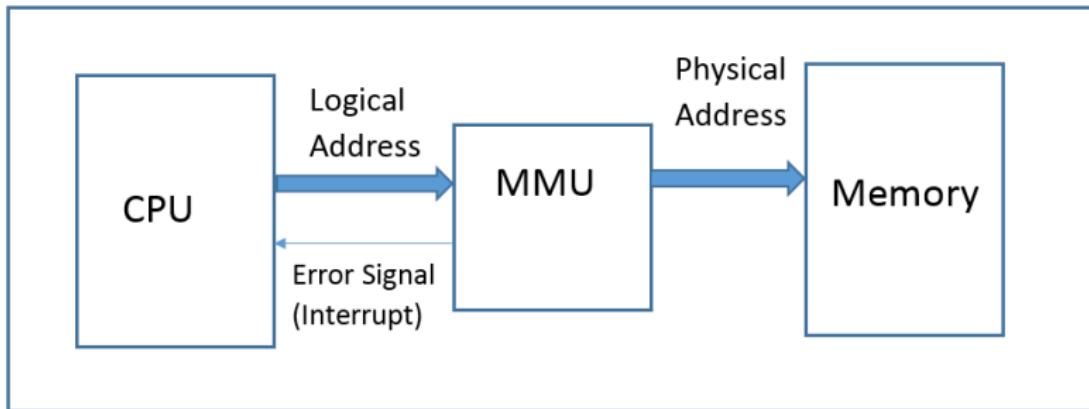
This is what the M3 has.

Memory Management Unit (MMU)

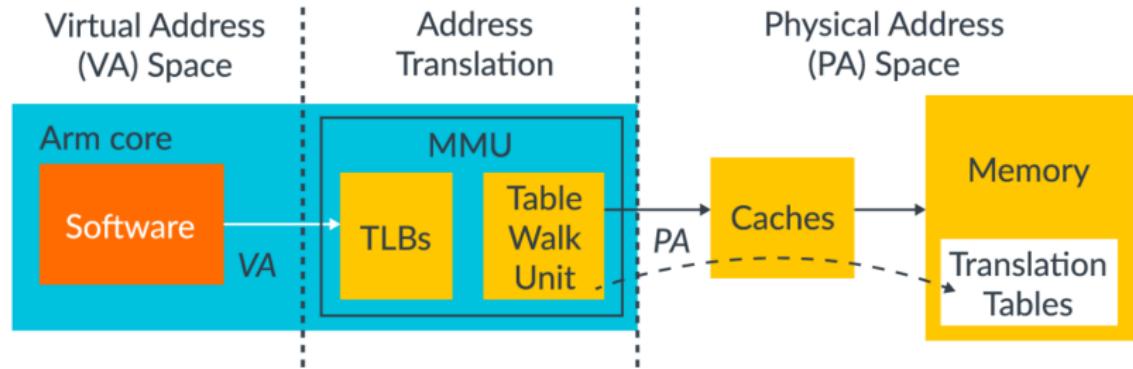
- Needed for most OSes that load programs into memory at runtime
 - “Physical address” is not known at compile time
 - Program runs at “logical address”
- Translate logical address to physical address
- Memory protection (same functionality as MPU)
- May handle virtual memory request



Memory Management Unit



Memory Management Unit (Hardware)



From: <https://developer.arm.com/architectures/learn-the-architecture/memory-management/the-memory-management-unit-mmu>

Memory Management Unit (SW response to HW)

When the TLB indicates a miss, why would a RT engineer want to signal the CPU?



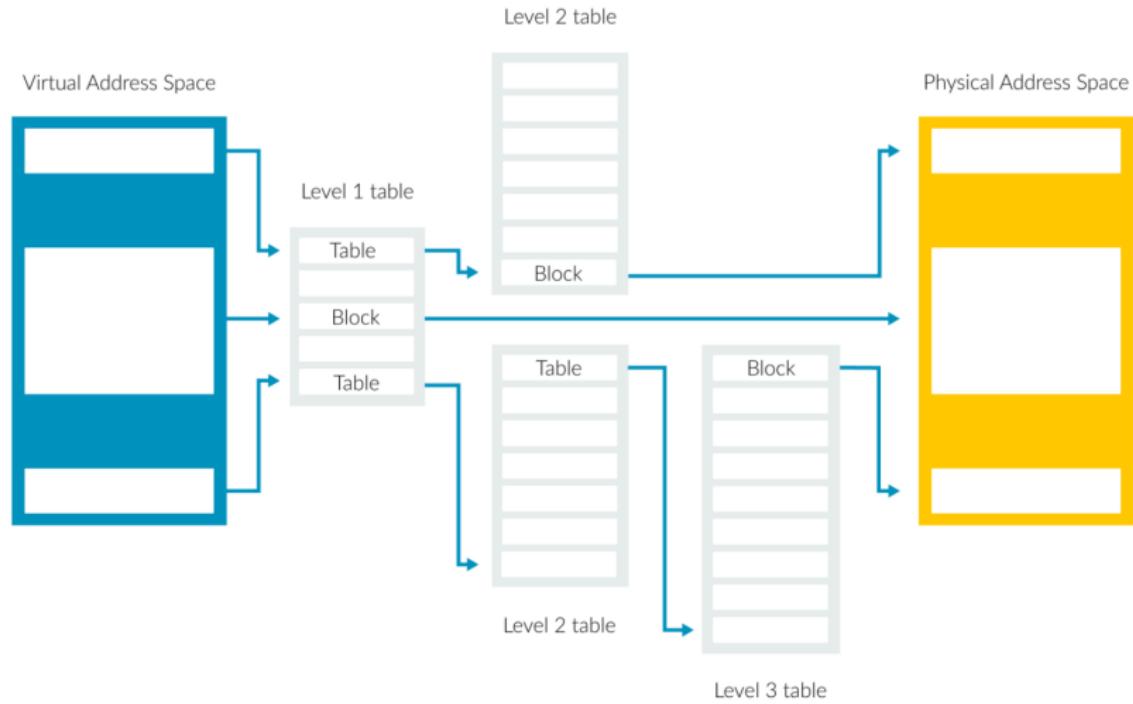
Memory Management Unit (SW response to HW)

When the TLB indicates a miss, why would a RT engineer want to signal the CPU?

- 10 to 1e6 cycles may be needed for fetch
 - 100 for DDR memory (Got any other work to do?)
 - 1e6 (and other code execution to do the fetch) for HDD (Definitely have other work to do, too!)



Memory Management Unit (4-level ARM)



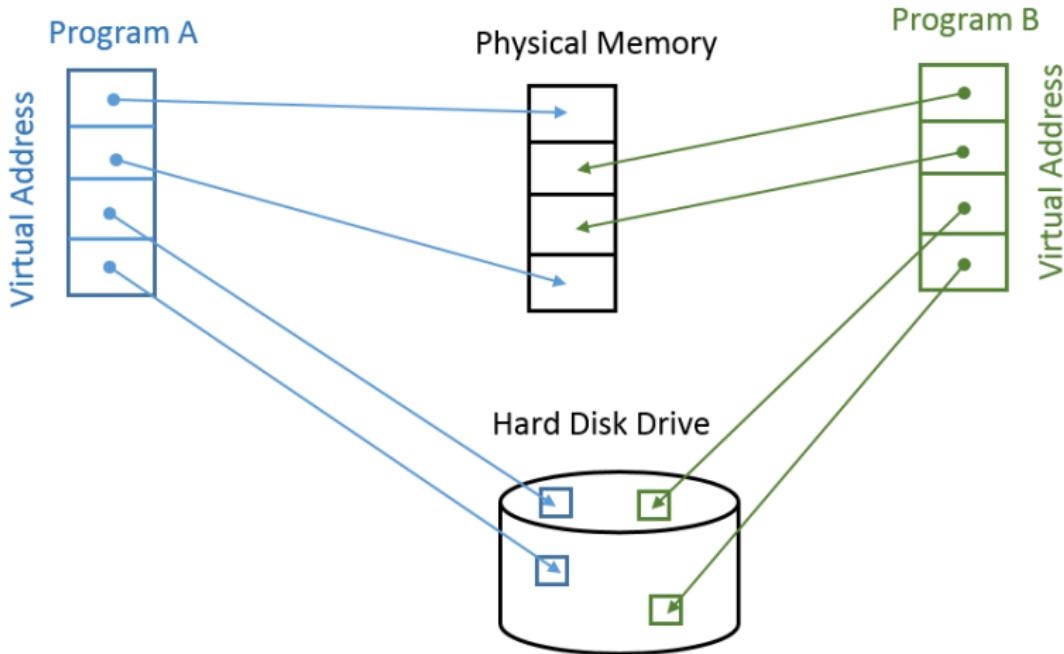
From: <https://developer.arm.com/architectures/learn-the-architecture/memory-management/the-memory->

Virtual Memory

- Virtual Memory is handled by MMU while in memories
- Virtual Memory is transparent to user program
- Virtual Memory needs HW (MMU) and SW (OS to set it up) support
- Virtual address can be mapped to physical memory, or to secondary storage (e.g. HDD) by having the OS support the fetch resulting from the “miss”
 - When a “miss” happens, the data needs to be fetched, then the code that faulted is re-started on the memory access instruction
- Same virtual address for different tasks is mapped to different physical addresses



Virtual Memory



ECEN 3753: Real-Time Operating Systems

Inter-Task Communication

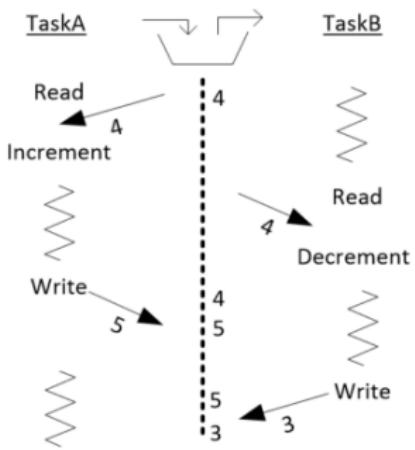


When Tasks Collide

- A single piece of HW can be accessed by both Task1 and Task2
 - The HW is a shared resource
- Task1 wants to write to the HW
 - ... but halfway through making its updates, it gets swapped out for Task2
- Task2 also wants to write to the same HW
 - ... and finishes its operation
- What state is the HW in?
- Is this behavior deterministic?



Tasks Collide: Illustrative Example



Types of Shared Resources

- Memory
 - Instances of data structures in RAM or ROM
 - Heap data
 - Global data
 - Static data within a function (non-reentrant)
 - Memory mapped peripherals (ADC, GPIO, communication ports, etc.)
 - Could be local to single CPU or shared across CPUs
- Files
- On-Board Peripherals
 - Shared across CPUs in multi-processing environment
 - DMA engine
- External Hardware Devices
 - Shared across a communication link or network (printer, motor, valve, switch, ...)
- Others? Counter-examples, by design?



It Is Not Just About Resources

Sometimes, tasks that are responsible for different parts of a algorithm must synchronize or coordinate their activities.

- a Rubic's cube solver with different motors for each axis cannot perform pre-computed operations
 - at the same time
 - in the wrong order

and expect to get the results you want.

Assembly robots, too!



Two Approaches For Resolving Conflicts

IPC: Allow entities to communicate with each other to minimize direct conflict (Inter-Process (or Task) Communication)

- Task1 may access a resource. (could be **SW Resource Mgr**)
- Task2 does not access the same resource directly.
 - If Task2 wants to access the resource, it signals Task1 to operate the resource appropriately. E.g. UI tasks typically don't touch HW outside the UI—they signal some other task to do what the user requested.

Mutual Exclusion: Prevent multiple tasks from accessing a shared resource simultaneously Scheduling is essentially a MutEx for the CPU.

- 1st task takes exclusive ownership of the shared resource
 - No other tasks are allowed to access the resource until this task indicates that it is 'done' with its updates.
- 2nd task may try to access the resource but must wait (or block) until the first task finishes.



Inter-Task (Process) Communication

- Overview
- Condition Flag
- Semaphore
- Event
- Signal
- Pool
- Message Queue
- Mailbox



Coordination vs Synchronization

- Coordination
 - Tasks execute in correct order
 - Meet specific condition
 - Ignore timing exactness
 - Tasks don't wait for each other
- Synchronization
 - Require timing exactness
 - In correct order
 - Meet specific condition
 - Task waits for other task



Overview

- Communication w/o Data Transfer
 - Coordination
 - Condition Flag
 - Synchronization
 - Semaphore
 - Event
 - Signal
- Data Transfer Only
 - Pool
 - Message Queue
- Synchronization w/ Data Transfer
 - Mailbox



Condition Flag

- One task sets the flag
- Another task reads it, and resets it (after it is read)
- Neither task waits

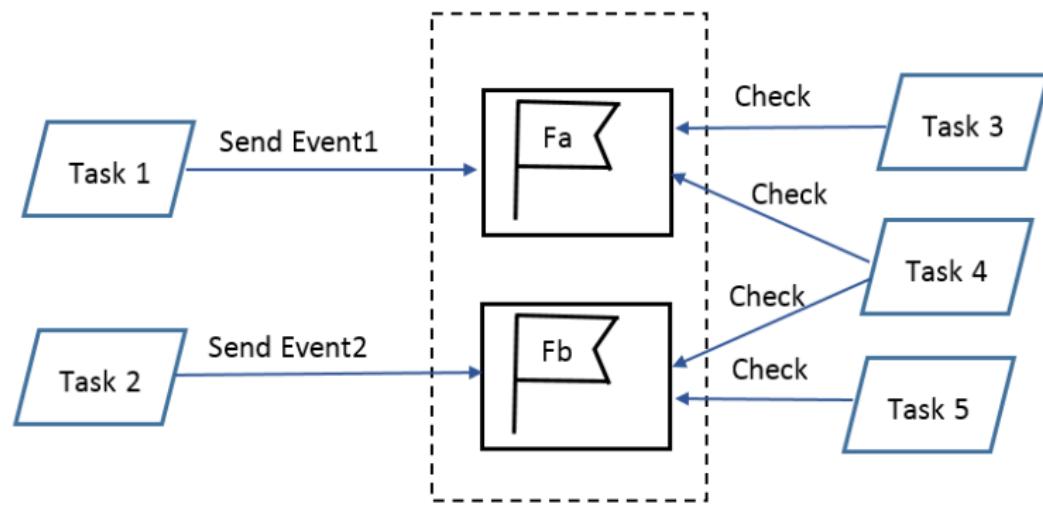


Condition Flag Group

- Group a set of flags into a single unit (i.e. a word)
- Each flag is a bit within the word
- Flags can be set/cleared with a single instruction
- Groups of bits can be changed using bit masking
- Task may read on a set of flags (OR, AND, etc)
- Task may broadcast flags to many tasks



Condition Flag Group example



Semaphore

- Semaphores are not 'owned' by any particular task.
- Semaphores are initialized with an initial counter value.
- Acquiring/Receiving ("Wait/Pend-ing") a semaphore when the counter is:
 - Zero-causes our task to block.
 - Non-zero-decrements the counter, and our task continues to run.
- Releasing/Sending ("Signal/Post-ing") a semaphore may increment the counter, if there are no other tasks waiting for the semaphore.
 - If a task(s) are waiting for the semaphore, the counter remains 0 and at least one waiting task is unblocked.
 - Semaphore options may be set to unblock either 'one' or 'all' waiting tasks when a semaphore is released.



Semaphore

- Used for inter-task communication
- One task reaches certain point, and waits for another task to finish before continue
- Unilateral: Only one task waits
- Can also be used to protect Shared Resource (next lecture), even by use (in “binary” mode) to implement mutual exclusion on shared resources, but are not explicitly associated with data.

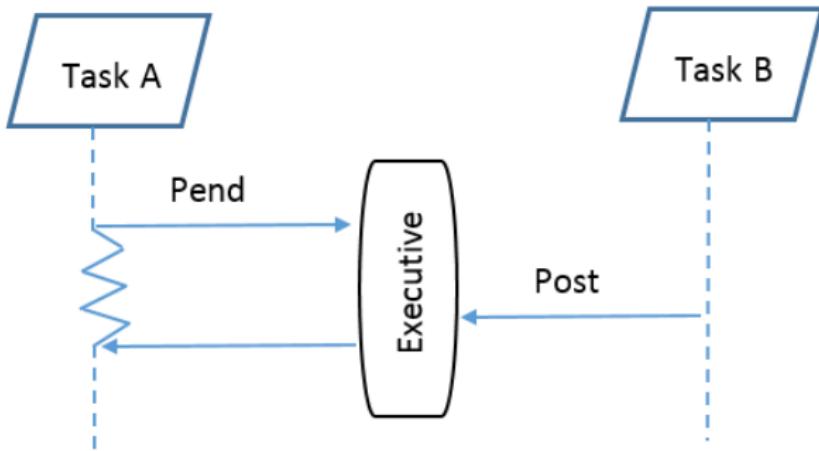


Semaphore flow

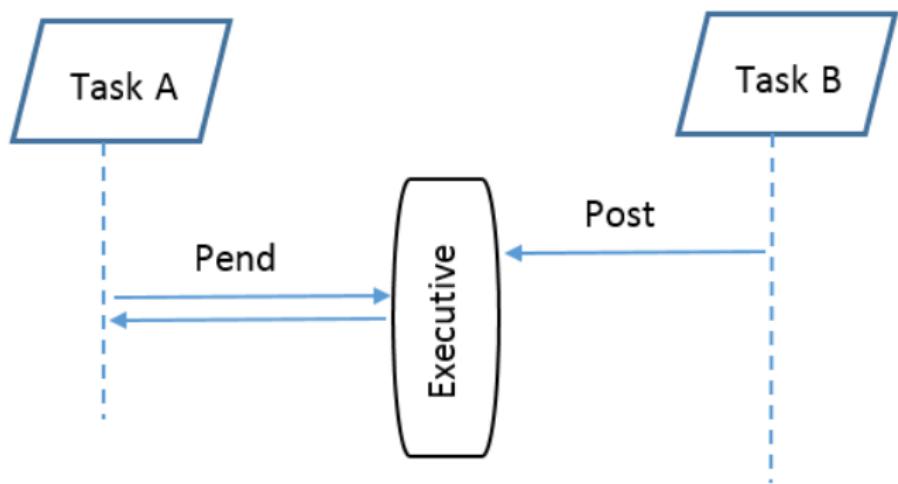
- Flag is initialized to clear state
- Receiver task calls Semaphore Pend
 - If flag is set, task continues
 - If flag is cleared, task waits
- Sender task calls Semaphore Post
 - If flag is in set state, just continue
 - If flag is in clear state, set state
 - If Receiver task is waiting, wake it up



Semaphore – Task A (receiver) reaches Pend first



Semaphore – Task B (sender) reaches Post first



Semaphores in Micrium

- Creating a semaphore
 - `void OSSemCreate(OS_SEM* p_sem, CPU_CHAR* p_name,
OS_SEM_CTR cnt, RTOS_ERR* p_err);`
- Acquiring (and possibly waiting for) a semaphore. Calling task may block.
 - `OS_SEM_CTR OSSemPend(OS_SEM* p_sem, OS_TICK
timeout, OS_OPT opt, CPU_TS* p_ts, RTOS_ERR* p_err);`
- Releasing (signaling) a semaphore
 - `OS_SEM_CTR OSSemPost(OS_SEM* p_sem, OS_OPT opt,
RTOS_ERR* p_err);`
 - [opt] can be specified such that on release, either one or all waiting tasks become ready



Semaphore example (uC/OS-II API)

```
OS_EVENT* Isr1Semaphore = null;

/* Create semaphore */
void CreateIsr1Semaphore() {
    Isr1Semaphore = OSSemCreate(1); /* count = 1 */
}

/* ISR to handle interrupt */
void ISR1() {
    ..... /* do stuff pre-SemPost */
    OSSemPost(Isr1Semaphore); /* semaphore to post */
    ..... /* if more to do */
}
```



Semaphore example (continued)

```
/* Task: the main flow */
void MyTask() {
    while(1) {
        .....
        /* wait for semaphore to continue */
        OSSemPend(Isr1Semaphore, /* semaphore to wait */
                  100,           /* timeout */
                  err);          /* error */
        .... /* do stuff post-ISR critical handling */
    }
}
```

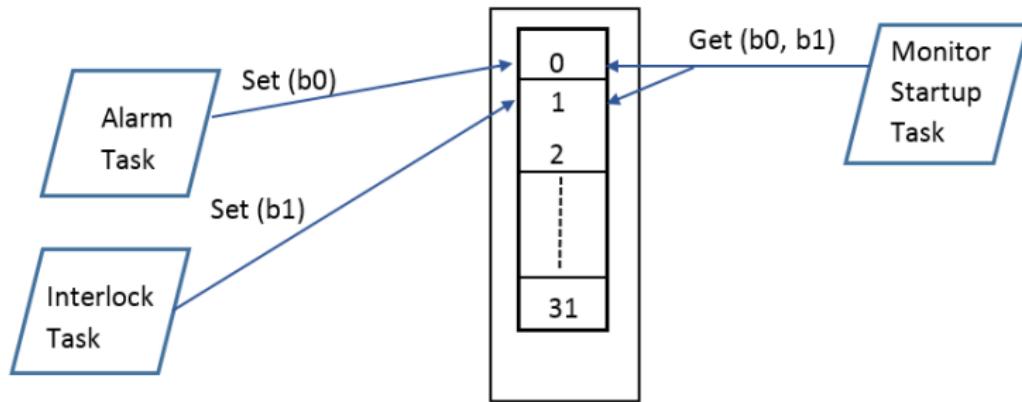


Events

- A series of bits in a word to hold current state of the events in the group
- Similar mechanism as Semaphore on Pend and Post
- Multiple tasks may post different flags
- Waiting task may wait for multiple flags (can be OR-ed, AND-ed)



Event example (from RTOS book)



Event example (from RTOS book, using uC/OS-II API)

```
/* Declaration, global variable on shared memory */
OS_FLAG_GRP* MotorStartEventGroup;
const OS_FLAGS AlarmEvent = 0x1;           /* bit 0 */
const OS_FLAGS InterLockEvent = 0x2;        /* bit 1 */

void main() {
    uint8_t error;
    .....
    MotorStartEventGroup= OSFlagCreate(
        0,           /* initial value */
        &error);   /* store error */
    .....
}
```



Event example (Pending)

```
void MotorTask() {  
    .....  
    OsFlagPend(MotorStartEventGroup, /* Wait on this */  
                AlarmEvent + InterLockEvent, /* Events to check */  
                OS_FLAG_WAIT_SET_ALL +      /* till all set */  
                OS_FLAG_CONSUME,           /* then consume */  
                0xFFFF);                  /* time out */  
    .....  
}
```



Event example (Posting)

```
void AlarmTask() {
    uint8_t error;
    OSFlagPost(MotorStartEventGroup, /* post in this */
               AlarmEvent,          /* Event to set */
               OS_FLAG_SET,          /* set bit */
               &error);             /* error */
}

void InterLockTask() {
    uint8_t error;
    OSFlagPost(MotorStartEventGroup, /* post in this */
               InterLockEvent,      /* Event to set */
               OS_FLAG_SET,          /* set bit */
               &error);             /* error */
}
```

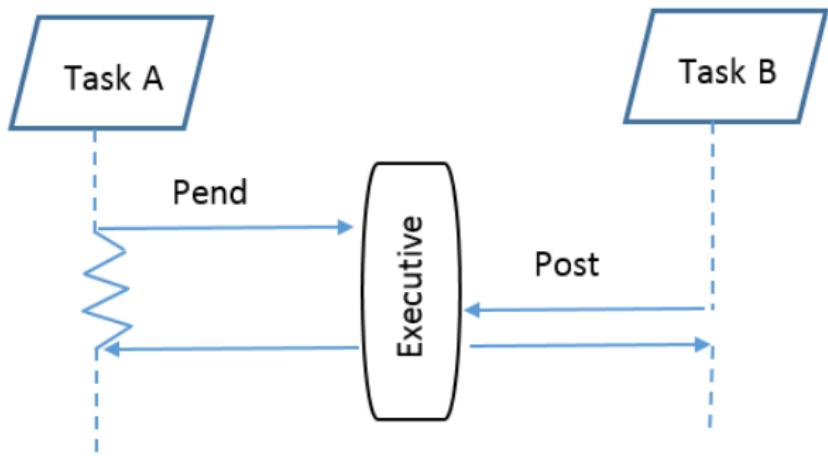


Signal

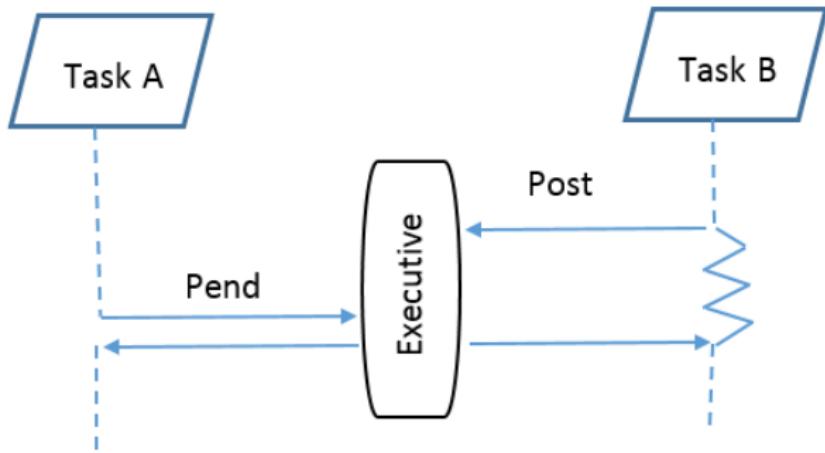
- Bilateral synchronization
- Both tasks wait for each other
- Both tasks are senders and receivers
- Few RTOS provide signal
 - Can be implemented by two Semaphores



Signal: Task A reaches Pend first



Signal: Task B reaches Post first



Signal example: implemented using Semaphore

```
OS_EVENT* SemaSync1, SemaSync2;

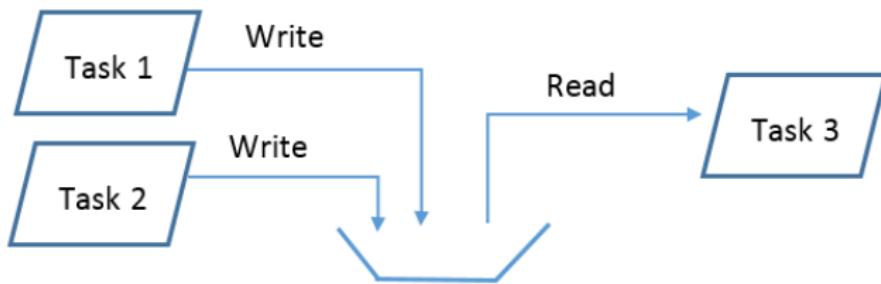
void main(int argc char* argv[]) {
    .....
    SemaSync1 = OSSemCreate(0); /* initial value */
    SemaSync2 = OSSemCreate(0); /* initial value */
    .....
}

void* Task1(void* param) {
    .....
    uint8_t error;
    OSSemPost(SemaSync1);           /* done w/ my part */
    OSSemPend(SemaSync2, 0xFFFF, &error); /* wait for other */
    .....
}
```



Pool

- Asynchronous info exchange
- Read-write random access (no requirement on read/write order)
- Non-destructive read
- Data may be protected (so that only one task access it)



Message Queue

- Other name: channel, buffer, pipe
- Asynchronous access
- One Task writes
- One Task read (destructive read)
- First In First Out
- Can be data or pointer to data
- Can be implemented by linked list or circular buffer

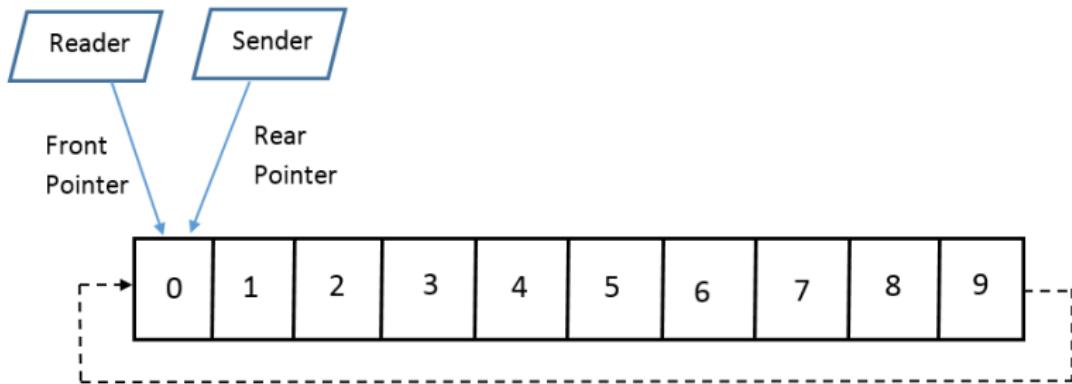


Message Queue: using circular buffer

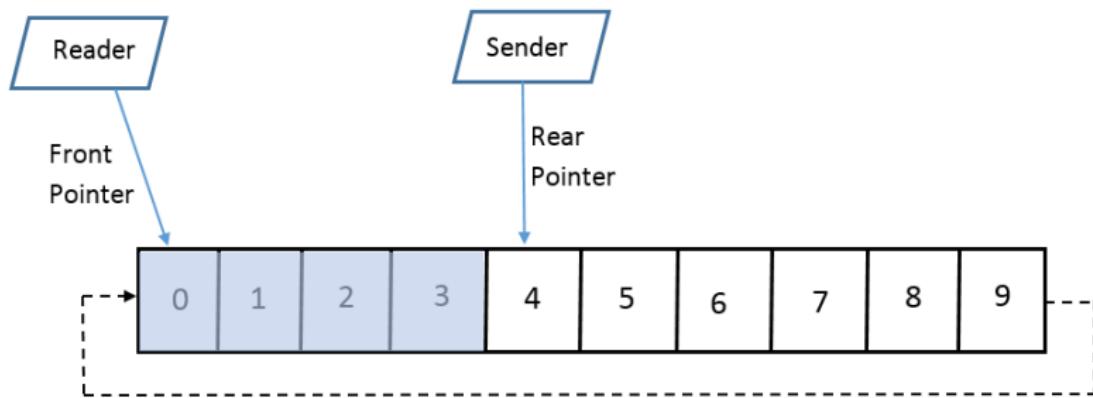
- Fixed memory space
- Reuse memory space after data is read
- Sender task may pend when buffer is full
- Reader task may pend when buffer is empty



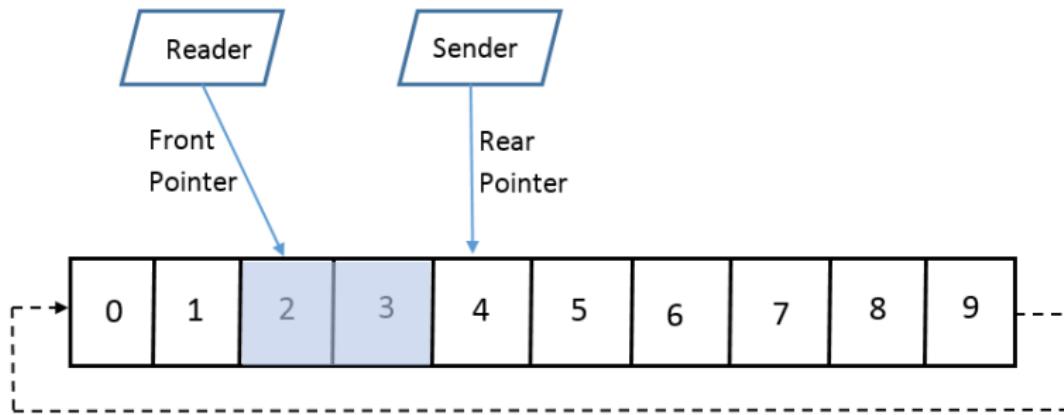
Circular Buffer Queue - Initial State



Circular Buffer Queue - Four Data Sent



Circular Buffer Queue - Two Data Read



Queue: Micrium example

```
/* Initialization Code */  
OS_Q App_QUSB;  
  
void main() {  
    .....  
    OSQCreate((OS_Q *) &App_QUSB,  
              (CPU_CHAR *)"USB Queue",  
              (OS_MSG_QTY)20,  
              (RTOS_ERR *)&err);  
    .....  
}
```

What exactly does the OS need to do in OSQCreate?



Queue example (continued)

```
void* Task1() {  
    .....  
    p_buf = Get Buffer from pool;  
    OSQPost((OS_Q      *)App_QUSB,  
            (void      *)p_buf,  
            (OS_MSG_SIZE)buf_size,  
            (OS_OPT      )OS_OPT_POST_FIFO,  
            (RTOS_ERR   *)&err);  
    .....  
}
```



Queue Example (continued)

```
void* Task2() {  
    .....  
    while (1) {  
        p_buf = OSQPend((OS_Q           *)&App_QUSB,  
                         (OS_TICK        )0,  
                         (OS_OPT)OS_OPT_PEND_BLOCKING,  
                         (OS_MSG_SIZE   *)&msg_size,  
                         (CPU_TS         *)&ts,  
                         (RTOS_ERR       *)&err);  
        Process packet;  
        Free buffer back to pool;  
    }  
    .....  
}
```

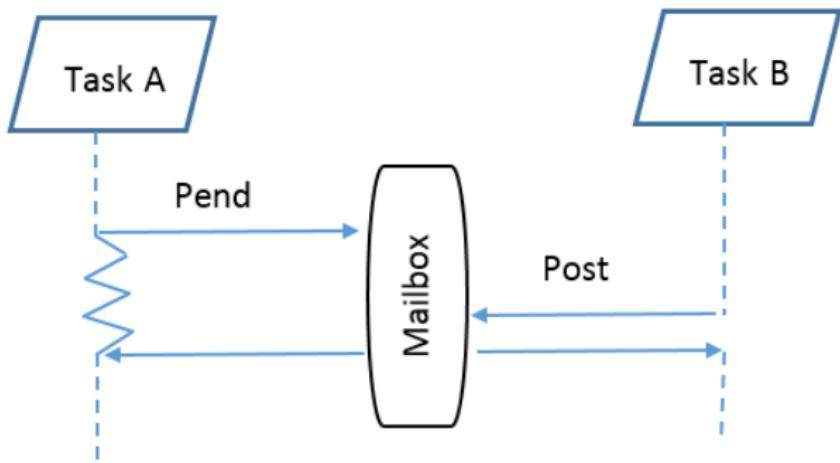


Mailbox

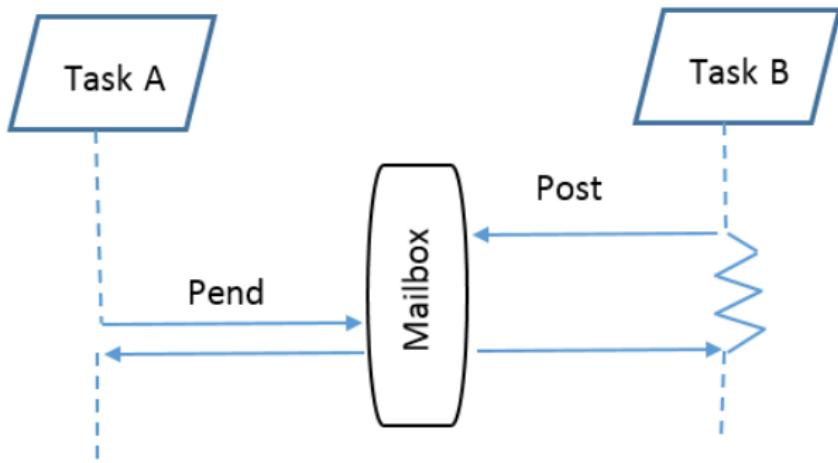
- Task synchronization with Data Transfer
- Signal + Data transfer
- May be many-to-one communication
 - The one can change over time!



Mailbox: Task A Reaches Post First



Mailbox: Task B Reaches Pend First



Mailbox example

```
/* Create a global mailbox*/
OS_EVENT* GlobalMailBoxA2B;

void main() {
    .....
    GlobalMailBoxA2B = OSMboxCreate((void*)0);
    .....
}

void* SenderTask() {
    .....
    uint8_t DataForMboxA2B[50];
    uint8_t err;
    err = OSMboxPost(GlobalMailBoxA2B,
                      (void*)&DataForMboxA2B[0]);
    .....
}
```



Mailbox example (continued)

```
void* ReceiverTask() {  
    .....  
    void* DataFromMboxA2B;  
    uint8_t Wait200 = 200;  
    uint8_t err;  
    DataFromMboxA2B = OS_MboxPend(GlobalMailBoxA2B,  
        Wait200, &err);  
    .....  
}
```



Summary

- Different inter-task communication approaches were covered
 - Coordination and Synchronization w/o data transfer
 - Condition Flag, Semaphore, Event, Signal
 - Semaphore is not covered in RTOS book, but it is widely used
 - Data transfer only: Pool, Message Queue
 - Synchronization w/ data transfer: Mailbox
- Only main APIs for each approach were covered
 - Many other APIs are available from OS, like OSSemDel, OSQFlush, OSQQQuery, etc
- Implementation of most approaches were covered
 - If you are writing OS, you may need to implement more



ECEN 3753: Real-Time Operating Systems

Shared Resources



Resource Contention Examples to be discussed

- ① Data Races
- ② Deadlock
- ③ Priority Inversion



Example of Mutual Exclusion : Data Races

- Generally occur when the order of CPU stores (writes) and loads (reads) are non-deterministic.
- Example 1 (conflicting writes)
 - void Task1(void*) {global_val = 1;}
 - void Task2(void*) {global_val = 2;}
 - When both tasks are finished, what is the value of global_val?



More On Data Races...

- Example 2 (read-modify-write with pass by value)
 - void Task1(void*) {global_val = AddOne(global_val);}
 - void Task2(void*) {global_val = SubtractOne(global_val);}
 - If the initial value of global_val is 0, what is its value when both tasks are complete?
 - Suppose Task1 runs, loads (reads) global_val and decides that it wants to set a new value of 1, but gets swapped out before completing its store (write)
 - Task2 now runs to completion, and captures the global_val parameter by value (global_val=0) before Task1 has rewritten it to 1.
 - Task2 runs to completion, changing global_val from 0 to -1
 - Task1 resumes, and sets global_val to what it had originally intended to write (global_val=1)

See visual representation of this type of problem, in IPC Lecture Topic.



The Solution to Data Races

If we can cause multiple potential users of a resource to mutually exclude their uses during each others', the result will be no unsafe/inappropriate sharing of data via race conditions!

The fundamental building block for Mutual Exclusion is the **Atomic Lock**.



Atomic and Volatile Data in C/C++

- Caveat: Other programming languages define these terms differently
- Volatile in C/C++
 - Does not prevent data races.
 - Used to prevent the compiler from optimizing out a read|write.
 - Memory mapped HW peripherals
- Atomic in C/C++
 - Can be used to prevent data races.
 - Helps enforce an order on loads/stores to memory
 - A happens before B (e.g. enumerated steps)
 - A synchronizes with B (e.g. indicator to block B)



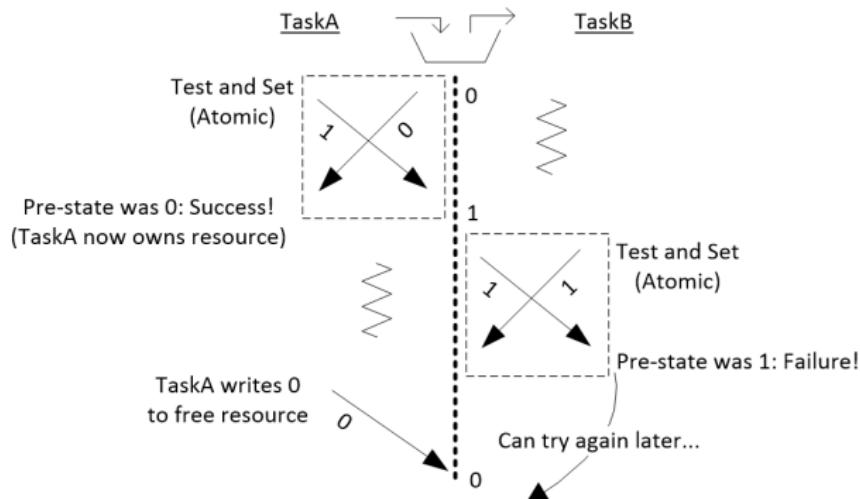
More on C/C++ Atomics

- Typically implemented with CPU word-wide default accesses for small data, without needing an OS mutex lock.
- For larger data, may use an OS mutex lock as part of the solution.
- Uses `compare_and_exchange` (alternatively: `test_and_set`) operations for read-modify-write
 - Loads data and compares to expected value
 - If the compare succeeds, the new value is written, otherwise the value that was just read is returned and no write occurs.
 - A task may check whether the `compare_and_exchange` succeeded to guarantee that it was the task that completed the write successfully.
 - If unsuccessful, a task may loop using the value it read (which is data written by some other task) as the new expected value, and retry its write.



What is Test_And_Set?

Preferably **Hardware** operation on a bit, for use in mutual exclusion.
(See Intel:"XCHG", ARM:"SWP")



Design Pattern: Critical Sections

- Enter critical region
 - Access shared resources
- Exit critical region

The shared resource MIGHT be the CPU itself.



Atomic Flag to Protect a Critical Region

- One way to protect a critical region is to use an atomic flag ensure that only one task at a time may enter.
 - Setup an atomic boolean flag, normally cleared to indicate that no tasks are inside the critical region.
 - When a task wants to enter the critical region it tries to do an atomic test_and_set
 - If the test_and_set succeeds, the task may enter the critical region and is responsible for re-clearing the flag on the way out.
 - If the test_and_set fails, some other task succeeded on its set and is inside the critical region. The current task must **wait** until the other task exits the critical region.

When might blocking on a mutually-exclusive lock via the OS be better/worse than spinyielding?



Mutual Exclusion Strategies

- Mutex
- Semaphore (binary)
- Disable interrupts
- Disable the OS scheduler



Mutex

A Mutex is a **Lock** for supporting Mutual Exclusion on a Shared Resource.

- With a mutex, a task can indicate when it is entering/exiting a critical code region.
- While one task is inside the critical region, other tasks may not enter this region, but interrupts still may occur.
- To enter the critical region, a task acquires (lock/pend) the mutex.
 - If some other task in the system tries to acquire a mutex owned by another task, it blocks waiting for the mutex to be released.
 - We may choose to wait for either an infinite duration or with a timeout.
 - In rare cases, a task may acquire a mutex more than once (recursive mutex)
- To exit the critical region, the task (or recursion) that acquired the mutex releases (unlock/post) it.



One Critical Region...

Or Many.

Up until here, I described ONE piece of code that would access a shared resource, but in general there could be MANY that access the same shared resource. Locking must be done by each section to keep ALL conflicting sections from accessing the shared resource at the same time.



Mutexes in Micrium

- Creating a mutex
 - `void OSMutexCreate(OS_MUTEX* p_mutex, CPU_CHAR* p_name, RTOS_ERR* p_err);`
- Acquiring a mutex. Calling task may block.
 - `void OSMutexPend(OS_MUTEX* p_mutex, OS_TICK timeout, OS_OPT opt, CPU_TS* p_ts, RTOS_ERR* p_err);`
- Releasing a mutex
 - `void OSMutexPost(OS_MUTEX* p_mutex, OS_OPT opt, RTOS_ERR* p_err);`



Semaphore (repeat from ITC Lecture)

Semaphores are a **signaling mechanism**, which (in “binary” mode) can be used to implement mutual exclusion on shared resources, but are not explicitly associated with data.

- Semaphores are not ‘owned’ by any particular task.
- Semaphores are initialized with an initial counter value.
- Acquiring (“**Wait/Pend-ing**”) a semaphore when the counter is:
 - Zero—causes our task to block.
 - Non-zero—decrements the counter, and our task continues to run.
- Releasing (“**Signal/Post-ing**”) a semaphore may increment the counter, if there are no other tasks waiting for the semaphore.
 - If a task(s) are waiting for the semaphore, the counter remains 0 and at least one waiting task is unblocked.
 - Semaphore options may be set to unblock either ‘one’ or ‘all’ waiting tasks when a semaphore is released.



Semaphores For Mutual Exclusion

- A binary semaphore may be used exactly like a mutex
 - Create the semaphore with an initial count of 1
 - Always pair up acquire and release operations on all tasks
 - A task acquires the semaphore.
 - The task that acquired the semaphore subsequently releases it.
 - All other tasks must acquire then release in the same order.



Disabling Interrupts

- On ARM, typically implemented as masking off the IRQ (and potentially FIQ) bit(s) in CPSR.
- Disabling interrupts prevents any type of preemption
- Necessary when resources are shared between tasks and interrupt handlers
- Must consider the consequences
 - Interrupt latency
 - Missed interrupts
- **Make sure to re-enable interrupts**

Implemented in Micrium by macros OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL()



Disabling the Scheduler

- Can be used when a resource is shared between tasks, but not shared with an ISR (IRQs are NOT blocked by disabling the scheduler)
- Prevents the OS from changing task states, effectively allowing one task to run indefinitely
 - No other tasks may unblock.
 - No other tasks may transition from READY to RUNNING (even if a higher priority, or unrelated to the region being protected).
- Repeated/recursive disabling is possible. Scheduling is held off until ALL locks are removed.

Implemented in Micrium by OSSchedLock()/OSSchedUnlock().

What would happen if you make a blocking call from a task while the Scheduler is disabled?

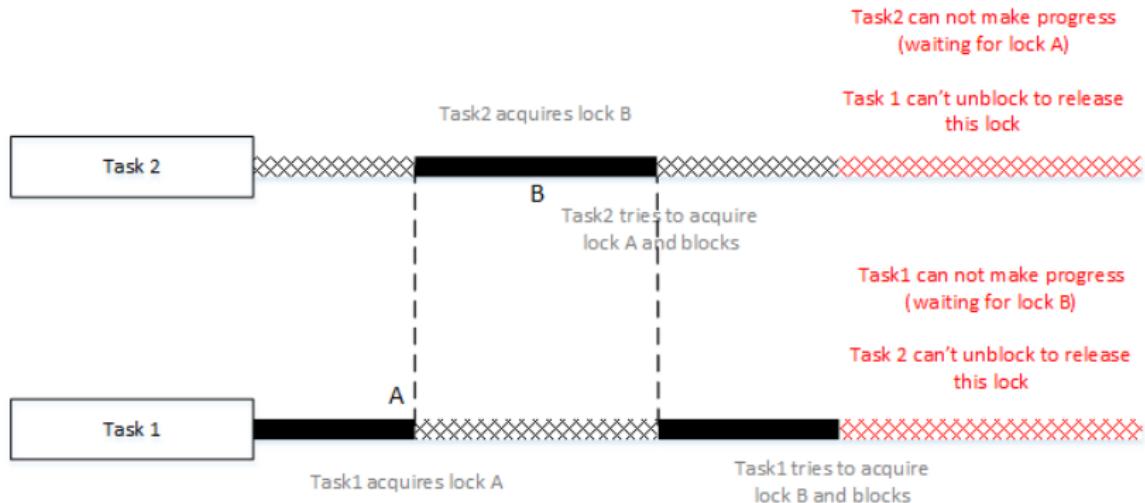


Consequences of Disabling Interrupts/Scheduling

- Late handling of interrupts
- Missed interrupts can lead to lost data
- Interference with timing requirements for unrelated tasks
- Higher priority tasks may not run, even if they have no relation to the critical resource being protected



Resource Contention Issue #2: Deadlock

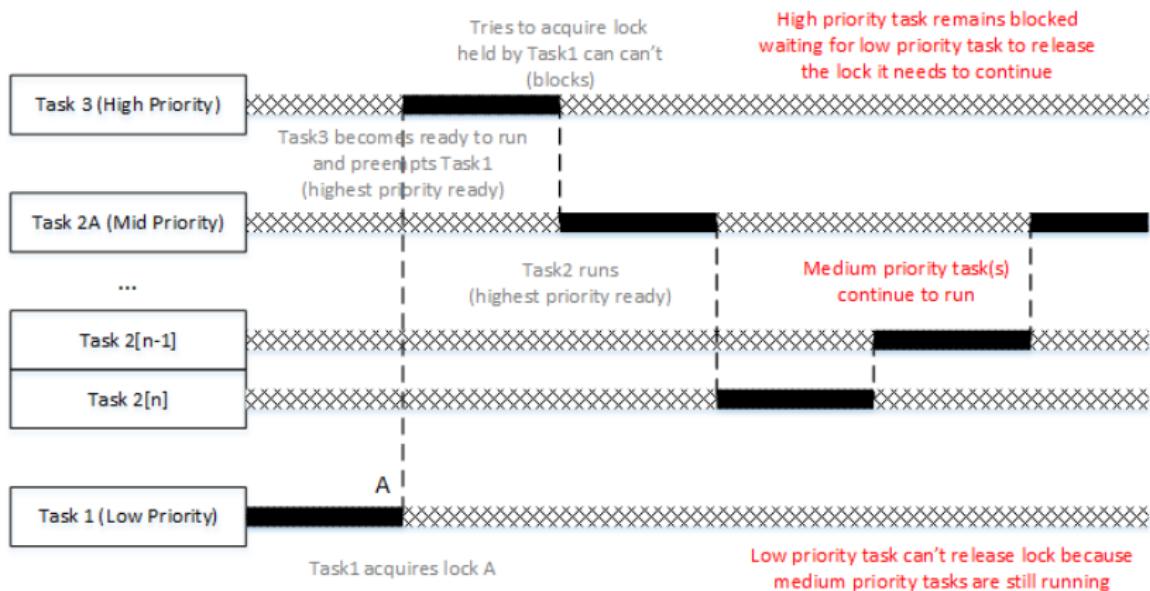


Preventing Deadlock

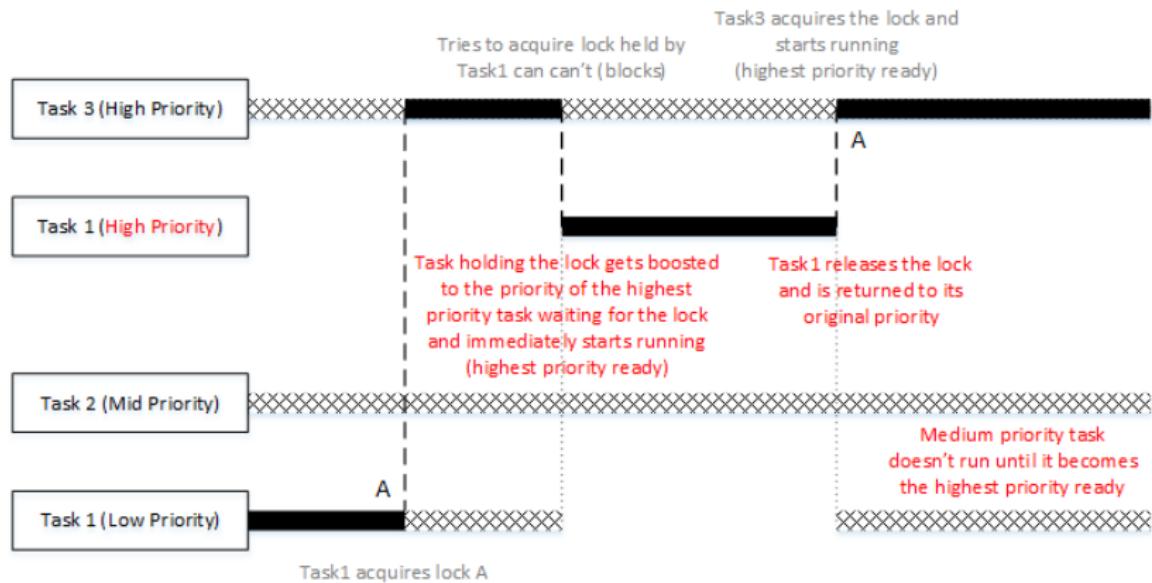
- Always acquire locks in the same order throughout the system
 - If a task needs to lock A before B, no one should ever lock B before A
 - Some static analysis tools (e.g. Coverity) are able to catch this heuristically; E.g. if your code locks A before B 95% of the time, chances are that the 5% case where B is locked before A is a bug.
- Always release locks in the reverse order that they were acquired.
 - If a task locks A then B, it should release B then A.
- Avoid creating circular dependencies between tasks. (More on this in a couple lectures)



Resource Contention Issue #3 : Priority Inversion



Priority Inheritance



Shared Resource Programming Mistakes

- Entering but forgetting to exit a critical region
 - Can occur when a function has more than one return point (if (error); return error;)
 - C++: Can use RAII (resource acquisition is initialization) to guarantee that all function exit paths exit critical regions.
- Improper nesting of OS locking calls
 - One function starts a critical region and calls to a second function that tries to exit the same critical region.
 - Prefer to avoid recursive mutexes, but if they are used, always enter/exit the critical region in the same function.
- Implementing critical region protection around too 'large' a region.
- Leaky API design: Allowing a pointer to protected data to leak out of a function.
- Inadvertently trying to block or wait inside an ISR.
- Others?



ECEN 3753: Real-Time Operating Systems

Risks

Illustrative Risk Types

- Risks are not just technical in nature (development unknowns, component failures), though they *often* are in our context
 - Head Assisted Magnetic Recording Heads for HDD
 - 5+ years from Demo to “Commercial Production”
 - 11 years from materials science doctorate to Demo, with lots of ablation!
- Human Factors (illness, accidents, major life events, burnout, size of currently-available talent pool)
- Business Flow (Supply Chains, drug/trade/political wars)
- “Acts of God”



Human Factor Risk Examples

- c.2001: Telecomm: Fastest routing product on market by significant margin
 - Fewer than 10 engineers who could write/adapt custom processor's assembly code fast enough to meet customer's schedules
 - Could trade off training-prep, tool expansion, coding
- c.2012: HDD project (Avg 100 engineers, 2 years)
 - 1 to prison
 - 2 deaths
 - a few retirements
 - multiple multi-month leaves-of-absence
 - half dozen left company
 - organization restructured once



Business Issues/Flow Risk Examples

- 1992: Presidential Political Party Flip: Aerospace imploded.
- 1994: Conner Peripherals: horizontal integration (6 months competitive lead)
- 1998: Seagate acquired Conner (vertical integration saved costs)
- 2000: 1" ~1GB Microdrive (vs. Flash)-cheapest per-device flip around 2003
- 2000+: cheap labor & market access v. IP (China)
- 2002: \$1B Palladium (Ford, Russia. Futures v. Engineering)
- 2018+: Cobalt for batteries (DR Congo: 60%)
- 2019+: Rare Earth Metals for HDDs (China: 80%)
- 1990+: Carbon Taxes
- Always: Energy costs

Expand your Horizons: read **The Innovator's Dilemma**, by Clayton Christensen



Playing Field is NOT level, nor linear

- Laws change with opinion (“artificial markets”)
 - Carbon Taxes (who pays?)
 - Renewable energy (rivers vs. CO2: Columbia River)
- Historical agreements
 - Western States water (Lakes Powell, Mead)
 - Hydro-power pricing (Niagara Falls)
- International agreements try to be “fair” (for some measures of fair), but rarely are seen that way by all countries.



“Acts-of-God” Risks

These are (nominally) the once-in-a-lifetime-or-rarer risks.

- Often discounted by businesses (quarters-to-decade vision)
- When they happen, either adapt or get out of the way
 - WDC, 2001 Thailand Floods (technical involvement, governmental mitigation). Stock fell 95%.
 - GM/Ford in 2006 Financial Crisis (governmental save vs. force to energy efficiency)
- Occasional mitigations by governments/companies/people
 - US/Russia: MAD policy: “Reduce chance of enemy first strike”
 - US, India Crop Insurance (for 1%, 43% of populations)
 - Beware of **Moral Hazard**
 - Elon Musk
 - Neuralink to counter AI takeover
 - SpaceX-to-Mars: to mitigate WW3. Side-benefit: counter Death of Solar life by asteroid?
 - Tesla (Cars-to-Batteries): Carbon/GlobalWarming mitigation



Risks: What to do, when?

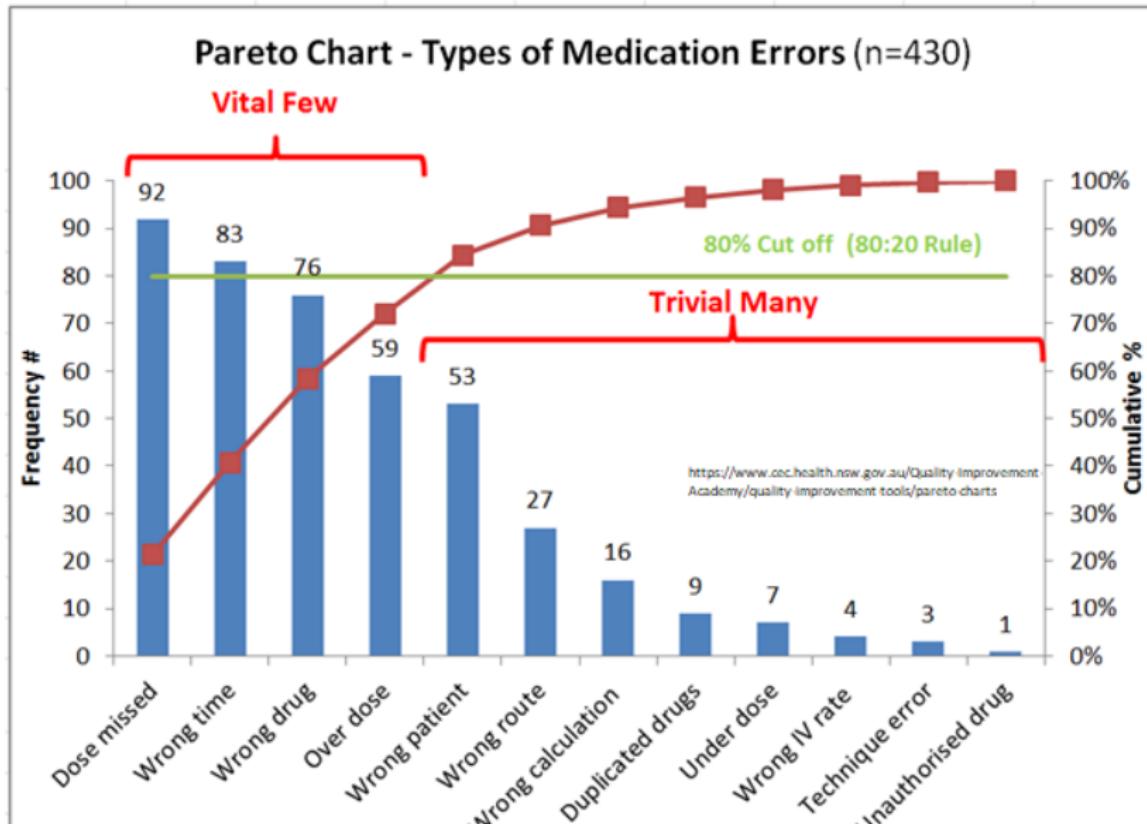
Risk Analysis:

- How bad is it?
 - What's the probability of the risk manifesting as an actual problem?
 - What's the likely impact if the risk manifests as an actual problem
- What's being done about it?

Multiple methods are used to quantify risks. Most treat the risk's "size" as the product of the probability and the impact IF it happens.



Pareto Chart (80/20 rule)



Agile Numbering

Conveniently an agile method of using a modified Fibonacci sequence, can be used for both probability and impact scales:

- 1, 2, 3, 5, 8, 13, 20, 40, 100 (can be %, and impact-out-of-100)
- E.g. Risk is of other coursework delaying you from Lab7 submission for a day
 - prior data suggests that about 5% of the labs will be submitted late
 - impact to your grade is about a 1 or 2 on this scale **Is everyone going to agree on the probability and impact?**

Always sort risks by their Risk value (product of the probability and impact), either with “Resolved” and “Mitigated” risks in a separate “completed” list, or not.



Risk Status: ROAM your risks

R esolved Truly, these are no longer risks. They may have occurred or not, but there is no remaining risk.

O wned These might happen, but we've got no plan yet. Whoever Owns a risk is seeking a mitigation or positive resolution.

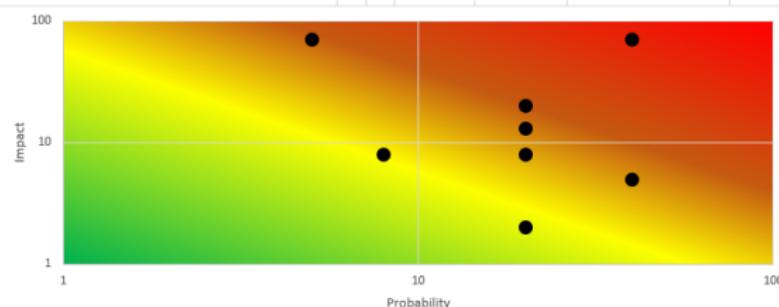
A ccepted These are going to hurt if they occur, but no further mitigation is deemed worth the price.

M itigated You've used your trained optimization skills to find ways to either reduce the *P* or *I* scores to the point that the remainder is another, lesser-valued risk.



Risk Register

Item	Risk (P*)	Recognized	Mitigated/ Resolved	ROAM	How
Registration snafu kills class	40 70 2800	4-Nov-19	12-Nov-19	R	Daily emails with supervisor, advisors
Too much lecture prep needed by 1st day	20 20 400	28-May-19	27-Aug-19	R	Enlisted volunteers at work (was M: JIT prep)
Insufficient student interest for CU funding	5 70 350	12-Nov-19	29-Jan-20	R	32 registered. (was M: gen compelling into)
Jon's day job interrupts too much for this	20 13 260	1-Nov-19	16-Nov-19	R	Jon's VP agreed to priority-swap
PPT fatigue for volunteers	40 5 200	27-Aug-19	3-Sep-19	R	Isaac set up markdown template, turnkey install
uC-Probe&Mac/Linux incompatibilities kill profiling	20 8 160	14-Jan-20	1-Feb-20	M	Supervisor installed uC-probe on lab Win boxes
Lab developer unable to finish lab development	8 8 64	3-Oct-19		A	If it happens, we'll react.
uC-Probe not licensed in time	20 2 40	4-Dec-19	7-Jan-20	R	(was O by Jon: if not in time, can use Segger only)



The figure is a risk matrix scatter plot. The vertical axis is labeled 'Impact' with values 1, 10, and 100. The horizontal axis is labeled 'Probability' with values 1, 10, and 100. A color gradient from green (low risk) to red (high risk) covers the matrix. Eight black dots represent specific risks: one at (10, 10), two at (20, 10), three at (20, 100), one at (40, 10), one at (40, 100), and one at (70, 10).

Here's an example from when this class was being developed.

Note that the items are things outside of my personal control—not “it might take me longer than I plan”. A valid risk is “I don’t know how long XYZ will take”, and you can mitigate that with “spike” activities (learning/analyzing/prototyping to gain better understanding).



Control Loop Risks

Very often, the HARD RT task in a RT system is a control loop, and we **need** to ensure sufficient performance.

Pre-“reading” for this lecture:

- <https://www.youtube.com/watch?v=wouhREkqZa0>
- Causes: Synch and Slack Time
- Effect: Delay Margin



Common Chain of Causes to Variable Latency

- Source of the readiness of some external thing is not exactly periodic
- Desire to share the CPU among many tasks leads to variable response time to the thingy (Interrupts, task switching latencies)

Approximately: a normal RV added to a couple of small uniform RVs.

- Usually, a pretty small range of latencies
 - Rarely, a significant outlier
- Mitigate with variable slack time that re-synchs everything



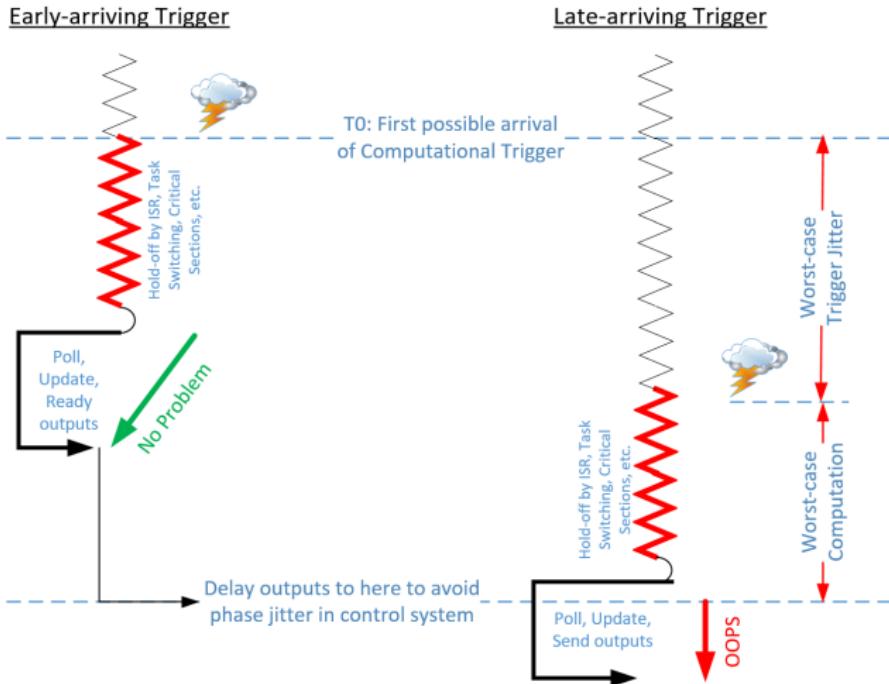
Control-Loop Latency Mitigation

Ideas that are relevant in most RT systems, particularly to help ensure the Hard RT task gets the service it needs:

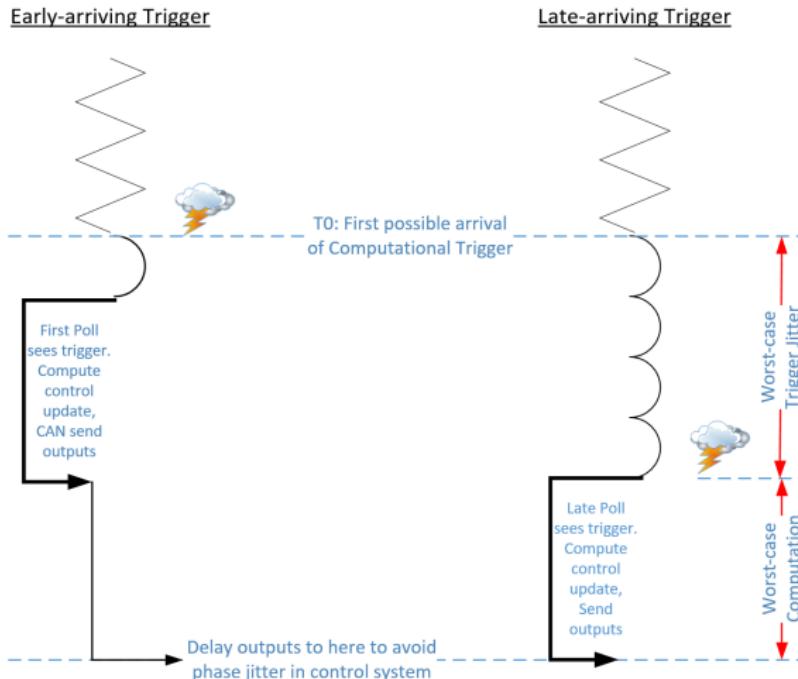
- Use timer to wake up before computation trigger, and poll for its arrival (see next pages)
 - E.g. Start-of-packet + Shortest-packet to begin polling, rather than end-of-packet interrupt
 - Subtract off the longest possible holdoff (longest critical section plus longest ISR/TaskSwitch, above HardRT priority)
- Pre-computation
 - Apply math ahead of time to get intermediate results
 - Compute estimated/expected values for nominal usage
- Use of Interrupts that have register banking or other accelerations



Why not just use Interrupt Triggering?



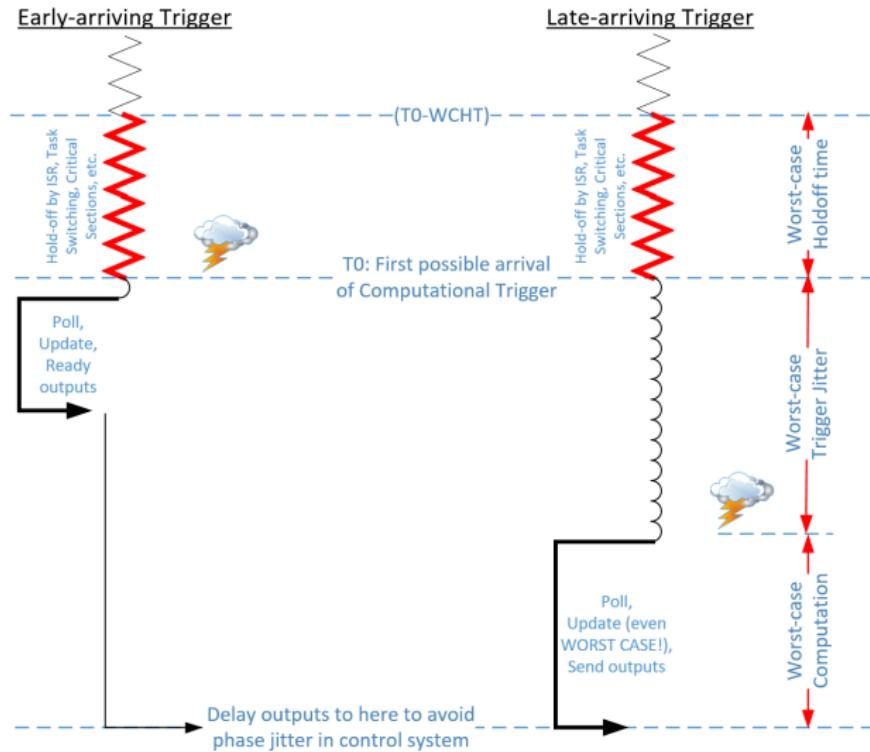
Begin Polling at Earliest Possible Trigger



What's this back to assuming?



Polling, with Compensation for CPU Holdoff



How does a cache miss on a low priority task's code factor into this?



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:

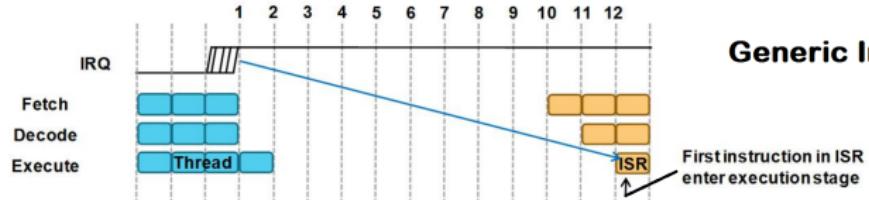
- Execution mode (affects stack selection, banking)
 - Any interrupt (since configured only in Privileged mode) is Privileged.
 - Generally, ISR returns to mode it came from. Supervisory Calls (SVC exceptions) can force return to Supervisor Mode.
 - Dual stacks help security, robustness. [How?](#)
-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

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

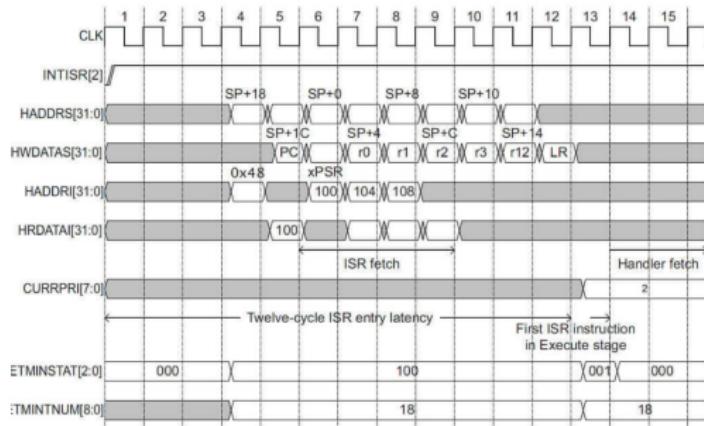
This table is applicable for NVIC. GIC introduces more modes, including many inspired by Security-related concerns.



NVIC IRQ Latency



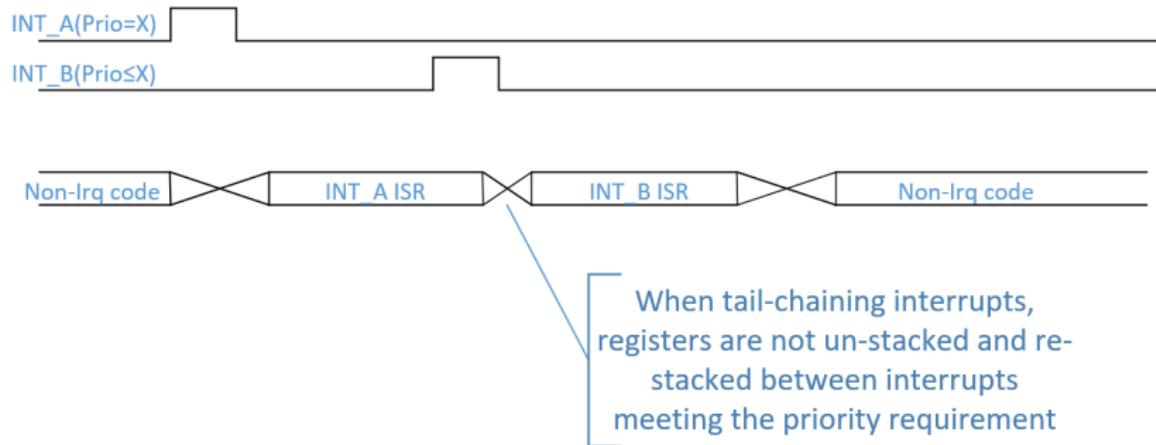
Generic Interrupt Latency



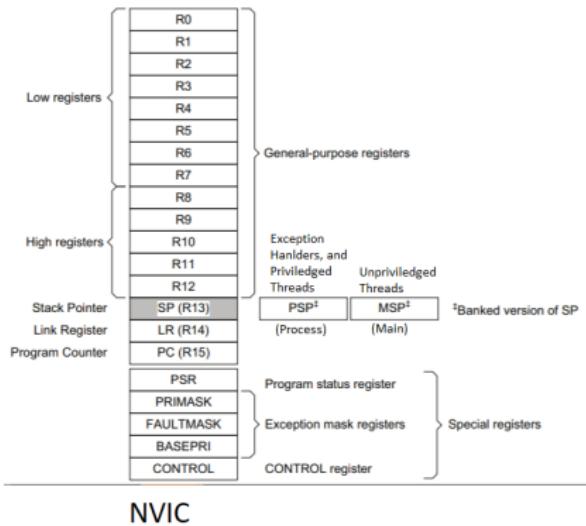
M4-NVIC IRQ Latency
(plus any additional register pushes!)



NVIC Interrupt Chaining



ARM NVIC vs. GIC Banking



User	Supervisor	Abort	Undefined	IRQ	FIQ
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12_fiq
SP	R13	R13_svc	R13_abt	R13_und	R13_irq
LR	R14	R14_svc	R14_abt	R14_und	R14_irq
PC	R15	R15	R15	R15	R15
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

GIC

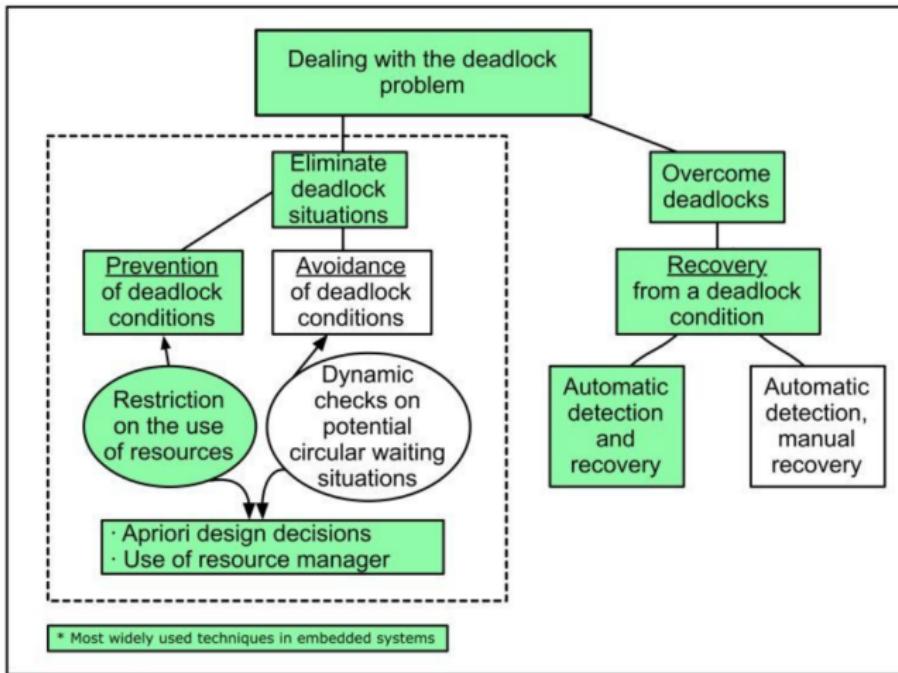


Resource Contention and Mitigation

- 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



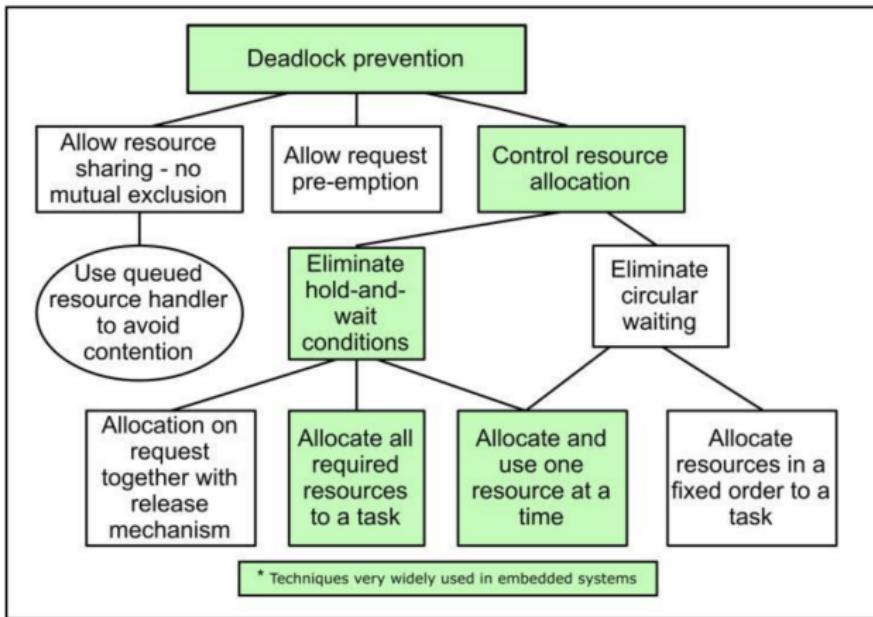
What to do about Deadlock? High-Level Strategies



From Textbook, Fig 4.9



What to do about Deadlock? Realistic Prevention Methods



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 ($\text{pri}(A) > \text{Pri}(B)$)

↓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 ($\text{pri}(A) > \text{Pri}(B)$)

↓Task : Resource→	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		



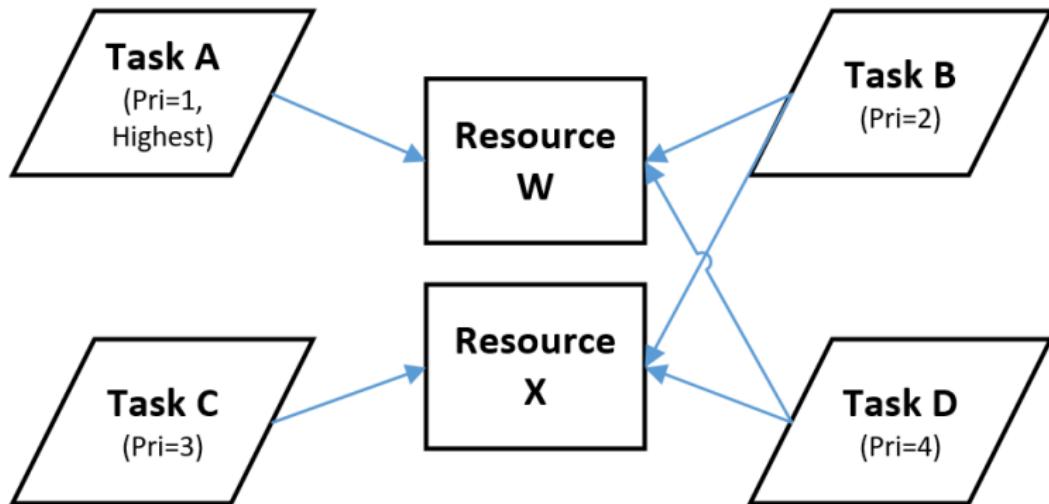
Fixed Resource Order

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

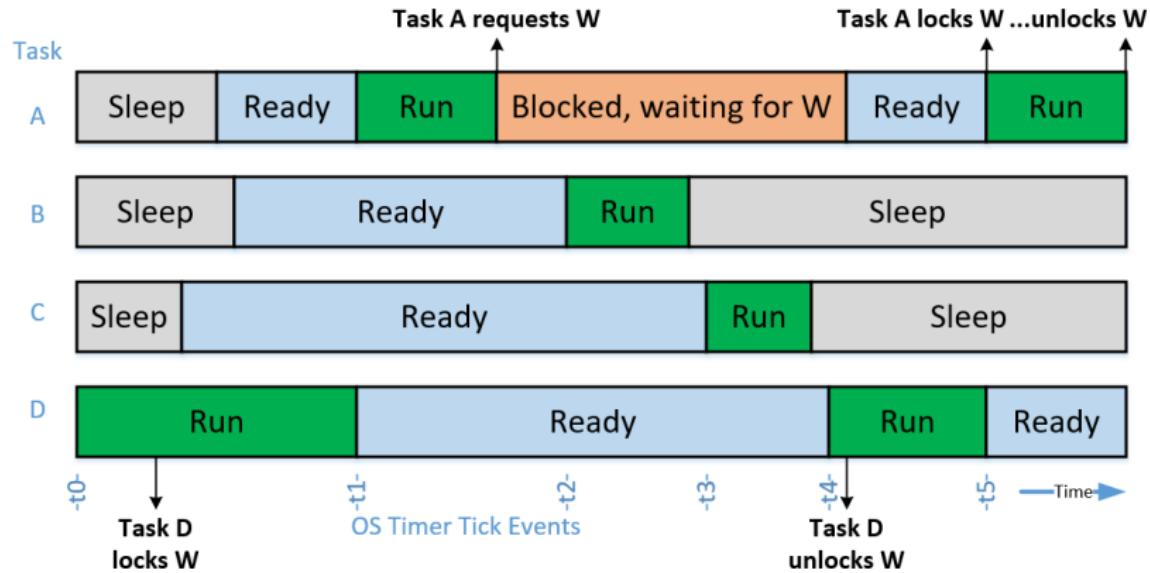


Deadlock: Avoided! Yay! Success?!?!

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.



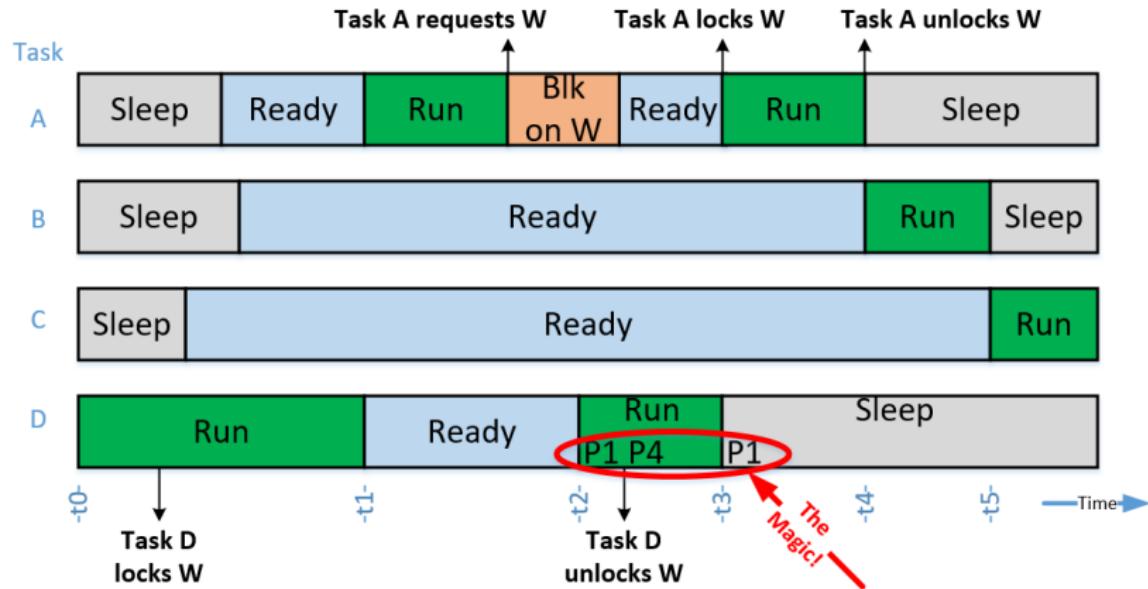
Priority Inversion (#2; the first was weeks ago)



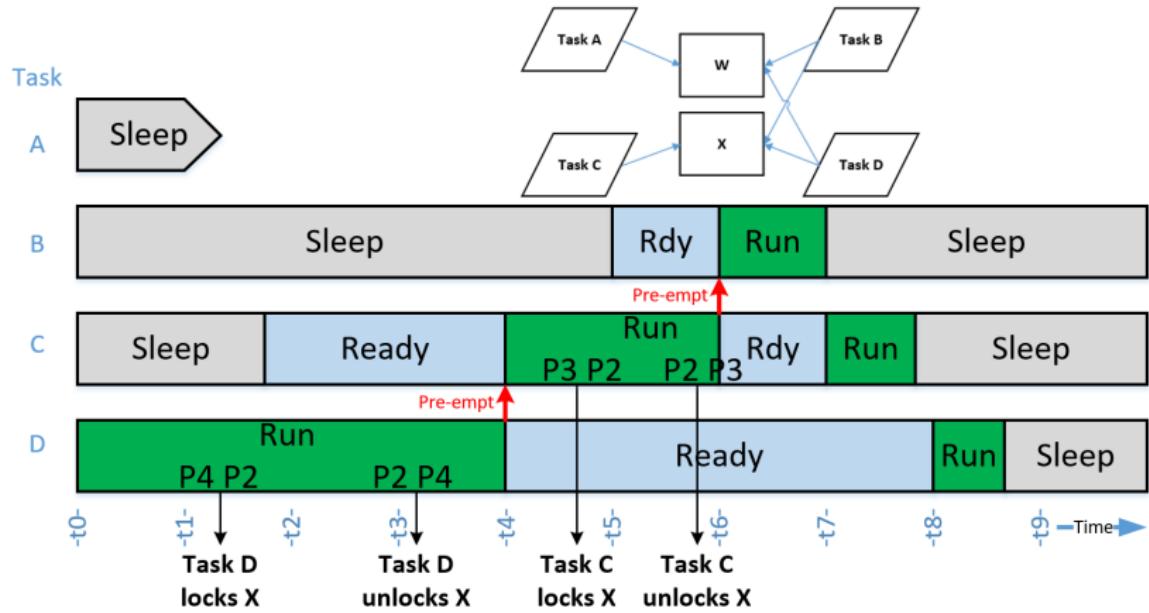
What are the 3 Task States, again? How do those map to the above? What do the sleep slivers (before t2, t3, t4) indicate about scheduling with this OS?



Priority Inheritance (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 resets¹

- 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

¹Not alien abduction as some early reports jokingly theorized, nor "doing too much" as less savvy 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! → 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

²The “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.

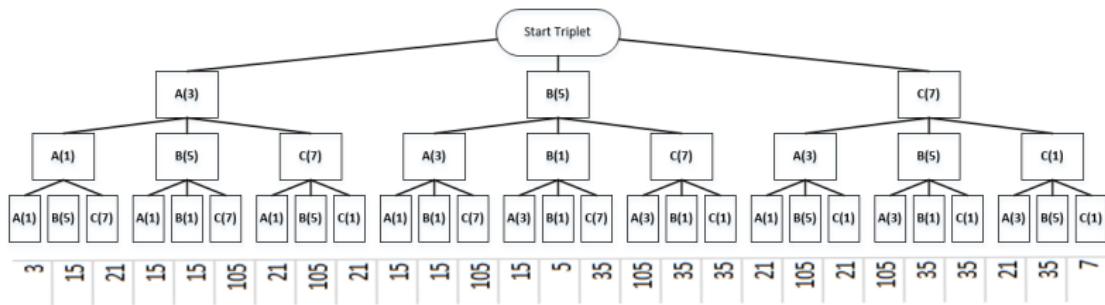
However, timing delays or differentially-variable loading on tasks may affect code behavior. Examples:

- 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 Synced?

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

- (A1)(B4)(C9)(A2)(B5)(C10). . . (A3)(B8)(C15)
- and then repeat (every $3 \times 5 \times 7 = 105$ 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 those³, 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?

³If 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 those³, 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.

³If 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...

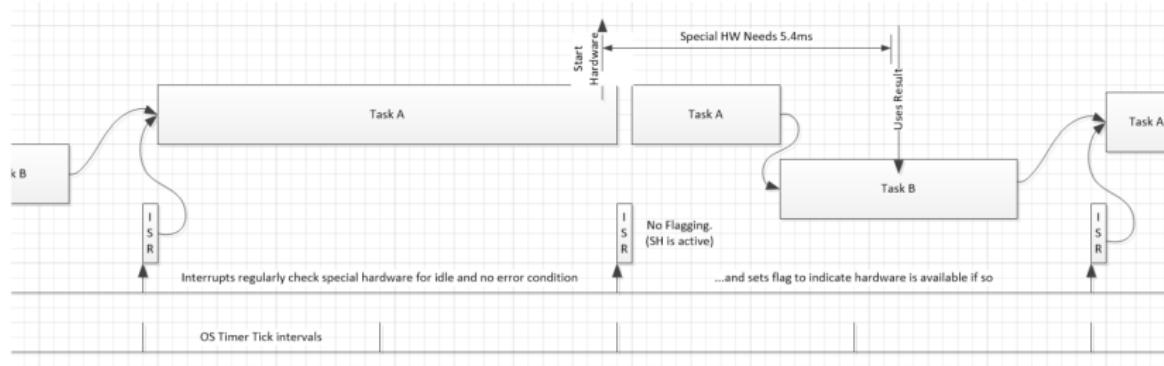
Imagine a system with one IRQ, two tasks, and “Special Hardware”

- 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.

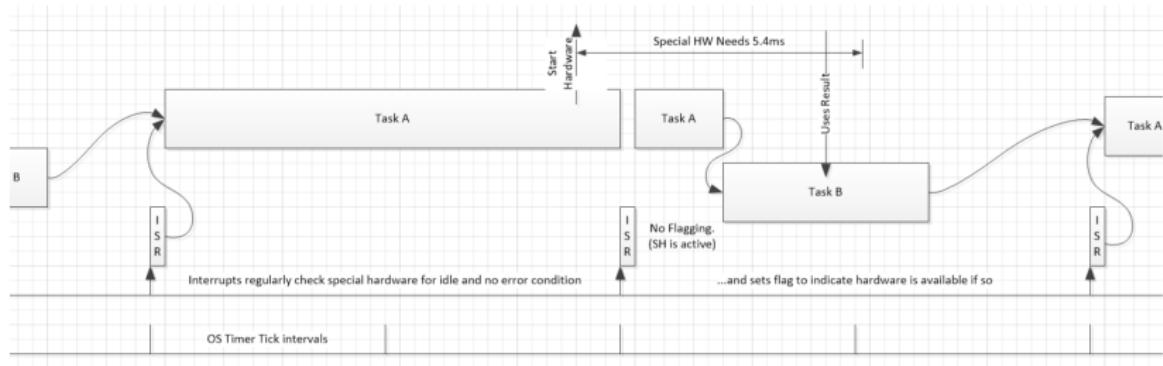
Pure Goodness, right?



Execution Sequence on Old Hardware



Execution Sequence on New Hardware (5% faster CPU)



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.

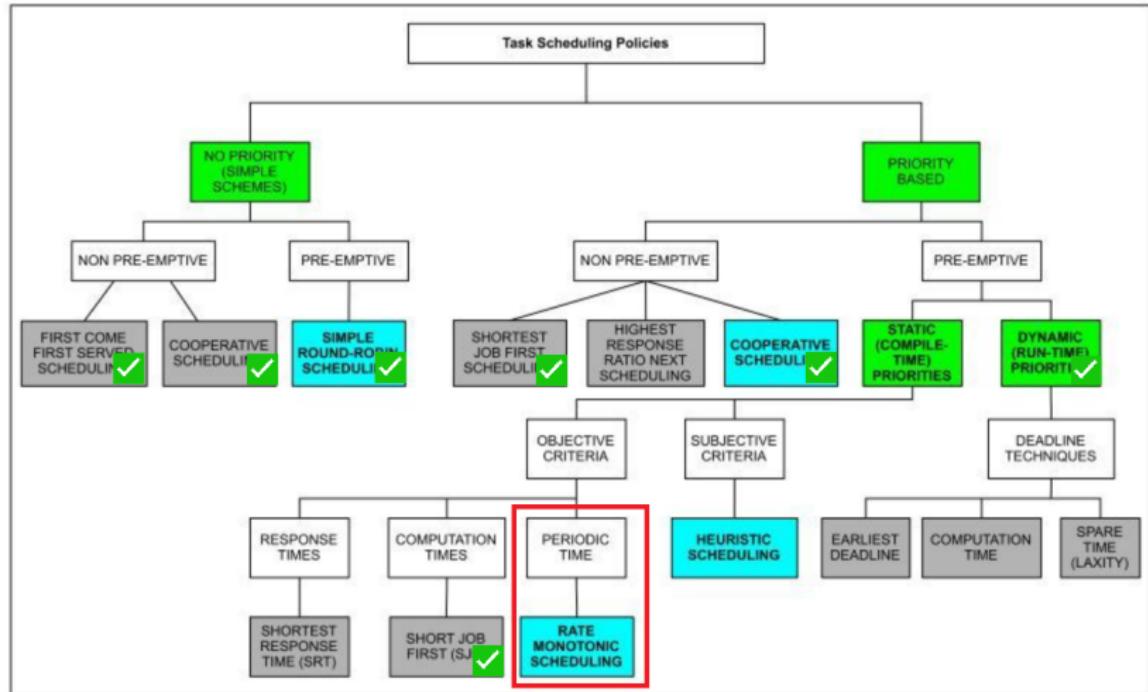
- Deadlock avoidance techniques in dynamic priority schemes add more possibilities for execution variation
- Inversion avoidance adds more

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 deadlines⁴
- 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 <math><69\%</math> utilization
 - EDF can reach 100%. The tradeoff is that it requires run-time mods to priorities based on next deadline.

⁴As long as there is enough CPU horsepower!



RMA: What IS It?

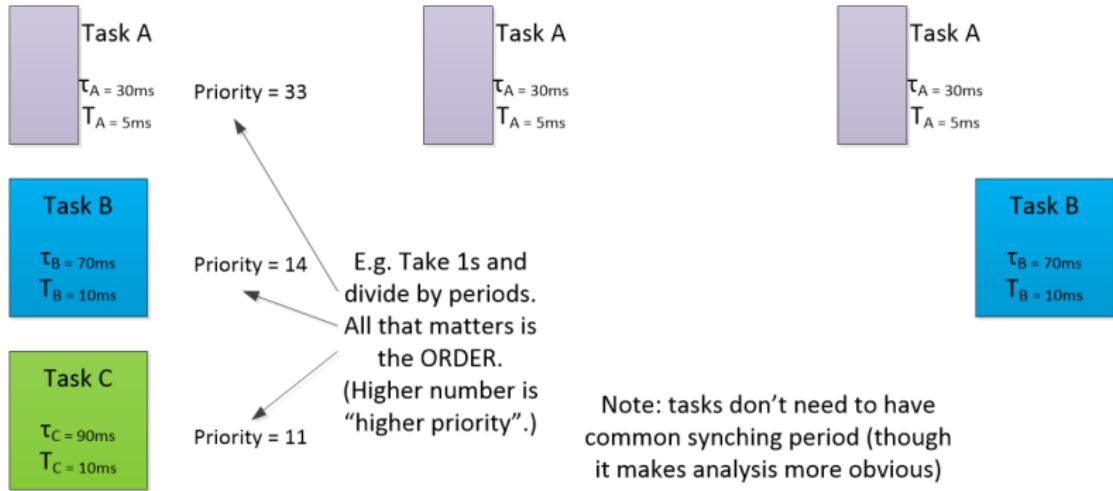
Fundamentally, the idea is to allow the task with the shortest periodic execution time to be the highest priority when it is ready to run.

- 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.

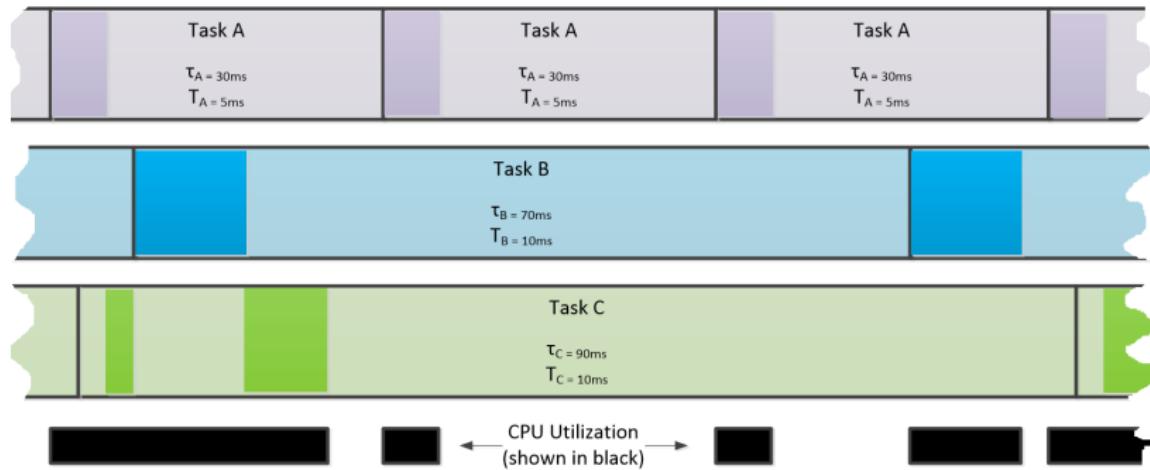


- The priorities need to be consistent with the OS's range. Micrium supports INT_8U in OSTaskChangePrio(), and only the idle task can be “1”.
- With RMA, CPU usage above 69% in the asymptotic case is essentially needed to deal with the Critical Moment.



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: $\tau=0.20\text{s}$, $T=0.05\text{s}$
- Task B: $\tau=0.5\text{s}$, $T=0.10\text{s}$
- Task C: $\tau=1.0\text{s}$, $T=0.15\text{s}$
- Task D: $\tau=0.25\text{s}$, $T=0.05\text{s}$

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:

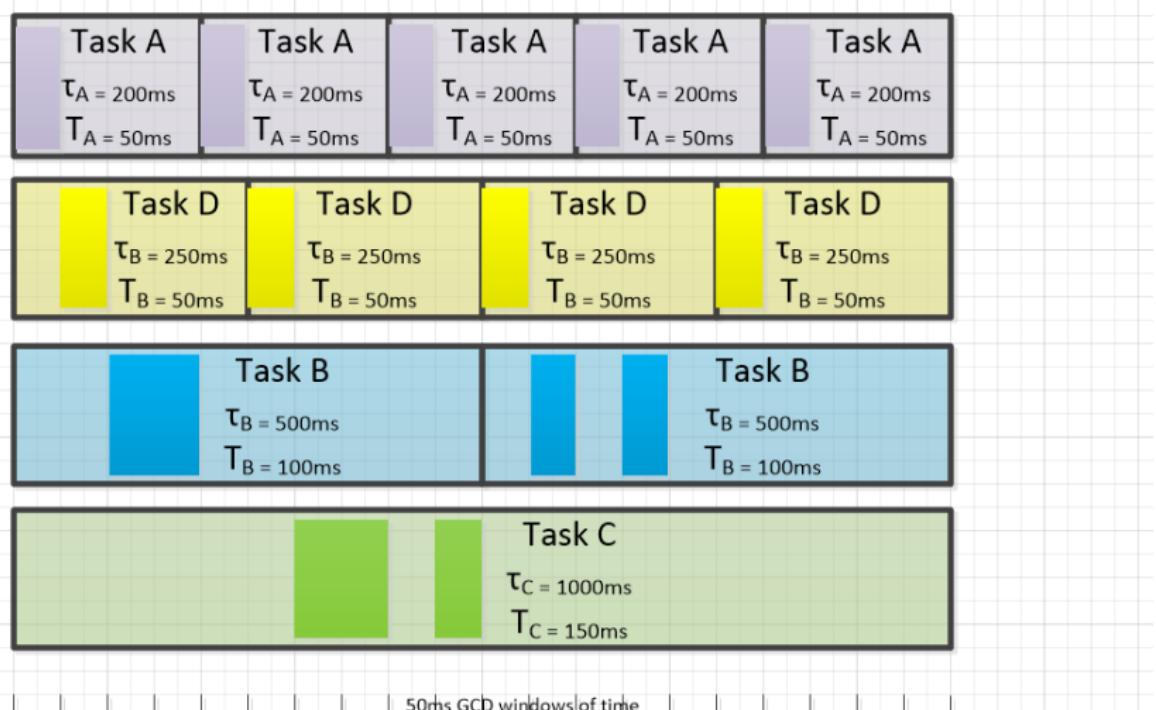
- Task A: $\tau=0.20\text{s}$, $T=0.05\text{s}$! %Util = 25%
- Task B: $\tau=0.5\text{s}$, $T=0.10\text{s}$! %Util = 20%
- Task C: $\tau=1.0\text{s}$, $T=0.15\text{s}$! %Util = 15%
- Task D: $\tau=0.25\text{s}$, $T=0.05\text{s}$! %Util = 20%

Total CPU Utilization expected: 80%

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



RMA: Example 2, Solution



CPU Used (Black periods)

Success or Failure known here (LCM(Periods))



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 $\text{LCM}(\text{periods})$ 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.

