Current Research: *Radiation Tolerance*

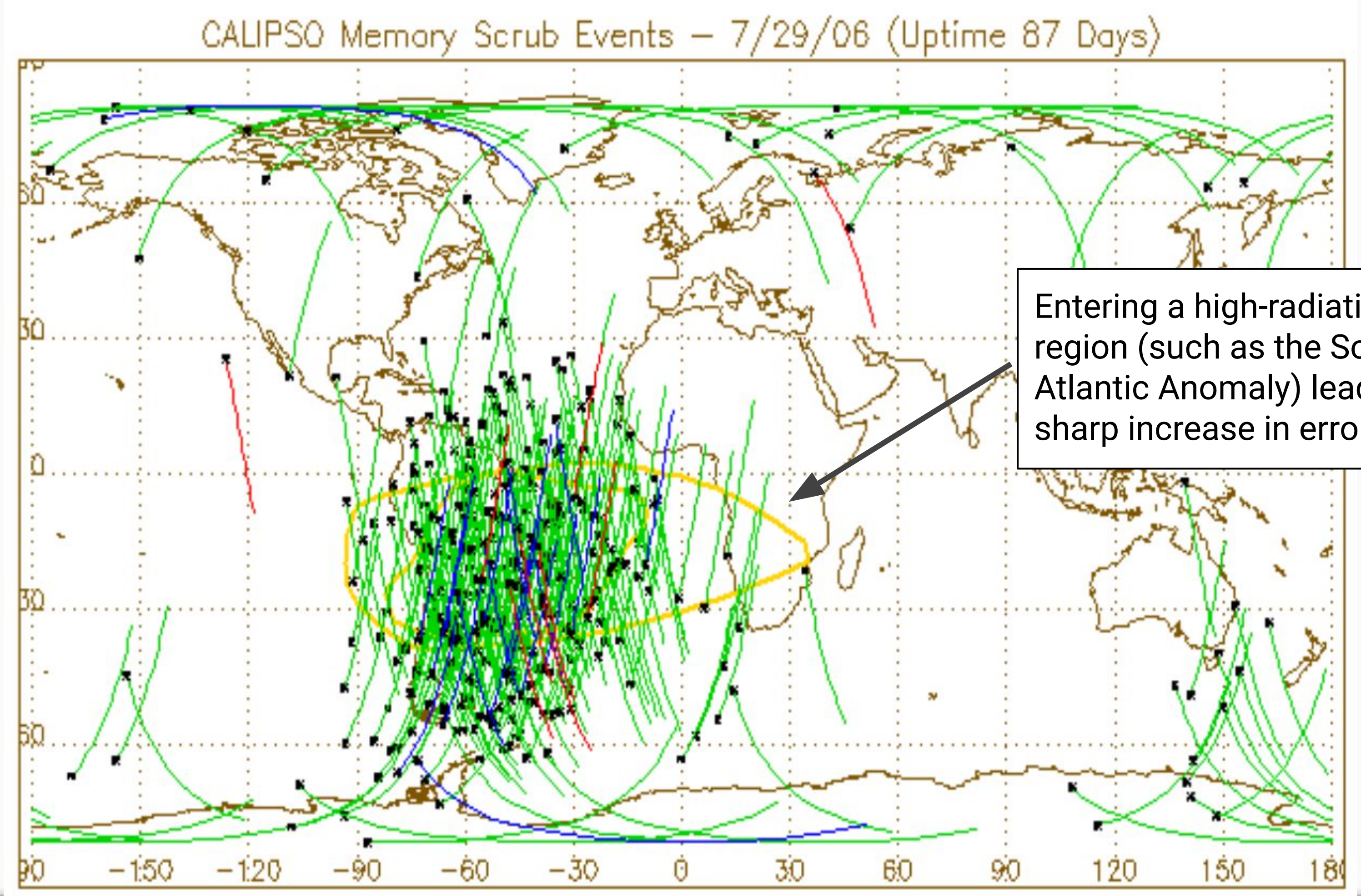
Fall 2021, TA: Cel Skeggs (they/them)

DISCLAIMER:

This is ongoing research…
any results are preliminary!

Case study in different design constraints

- Terrestrial computers are protected from intense radiation by the Earth's magnetic field.
  - Radiation-induced errors do occur on the surface, but they are **rare**.
- Spacecraft electronics must operate in **high-radiation environments**.
  - This radiation causes **bit flips** in code and data on the computer system.
  - Spacecraft computers are mission-critical: if a computer malfunctions, the spacecraft may not be able to reorient to Earth to receive commands, and **may never recover**.
  - One stray particle -> the loss of hundreds of millions of dollars of scientific investment.
- Spacecraft electronics must be **hardened** against – or at least **tolerate** – radiation errors that occur.



Entering a high-radiation region (such as the South Atlantic Anomaly) leads to a sharp increase in error rates.

SOURCE: R. DeCoursey, R. Melton, and R. Estes, "Non radiation hardened microprocessors in space-based remote sensing systems", Proc. SPIE Sensors, Systems, and Next-Generation Satellites, vol. 6361, 2006.

How can we mitigate these challenges?

GOLD STANDARD for robotic missions →

WELL ESTABLISHED →

Hardware Implementation	Software Implementation
Highly specialized **radiation-hard** processors and circuitry are used instead of off-the-shelf equipment.	Software is **recompiled or transformed** so that calculations are duplicated, and control flow is validated.
Hardware Architecture	**Software Architecture**
Multiple processors are placed in a **redundant configuration**. They vote on every action: the winning majority controls the spacecraft.	**Redundant multithreading** can mitigate radiation errors. A trusted unit of code performs voting between multiple individual threads.

← NOT YET TRUSTED

Why not the existing approaches?

Hardware Implementation Radiation-hardened processors are expensive, slow, power-hungry, and incompatible with normal development tools.	**Software Implementation** Control-flow errors can only be detected, not corrected; reinitialization may not occur fast enough for safe real-time control.
Hardware Architecture Redundant systems require multiple copies of expensive hardware. They can be extremely difficult to build and even harder to debug.	**Software Architecture** Redundant multithreading depends on an underlying hardened layer to perform the voting and context switching; how do you protect that?

Research Question

This is the topic of my in-progress research:

How can we reliably protect *the operating system itself* from radiation errors?

This is what is needed to make an approach based on redundant multithreading usable.

Goals for today

- Show what fundamental research on operating systems may look like
- Demonstrate how our designs change when our requirements change
- Help you think about alternative ways we can build operating systems

Part 1: Laying the groundwork

What do we need to solve this problem?

- The most important question to ask, when trying something new, is:

How will I know if I succeeded?

- In order to know if we have a good solution, we need to be able to **evaluate** a system and **quantitatively measure** how well it worked.
- We can build a system **without** our solution, and a system **with** our solution, and compare their performance.

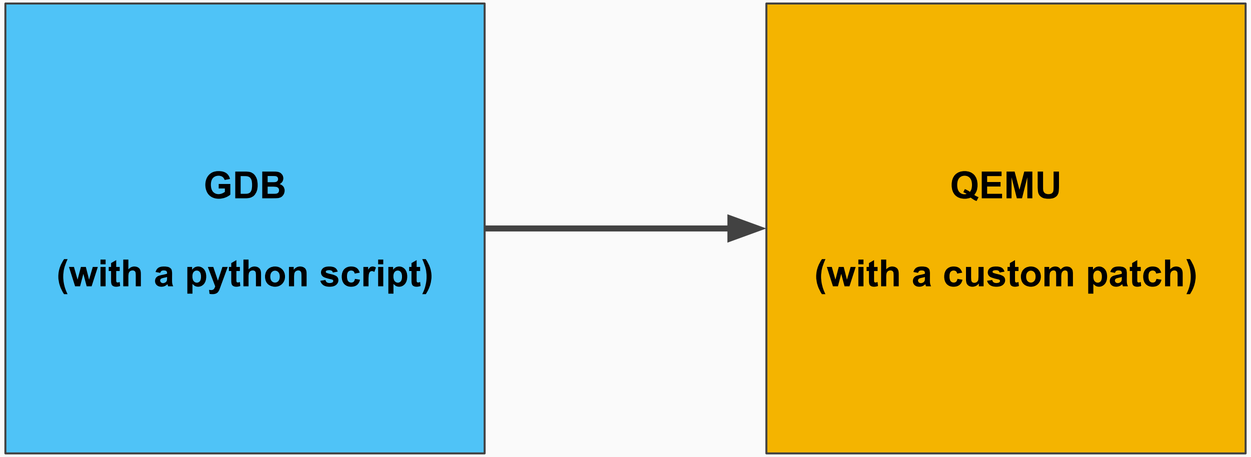
How can we test radiation tolerance?

- We can’t launch something into space.
    - At least, this isn’t the first step. We need to prove something on the ground first.
- We can’t point actual radiation at actual hardware.
    - My research budget is $0, and I can’t exactly get access to a nuclear research reactor.
- **We will have to *inject* faults into a system, rather than letting them occur naturally.**

Injecting fake radiation errors

- First approach: try to find an existing tool to inject radiation faults.
    - **Try not to build a tool yourself if an off-the-shelf tool will do!**
    - Unfortunately, while I found a number of fault injection tools during my literature review, none of them met my needs.
    - Particularly: I needed a *fast*, *controllable*, and *accurate* system for injecting faults.
- In the absence of an existing tool that met my needs, I built my own… out of some existing parts.

High-Level View



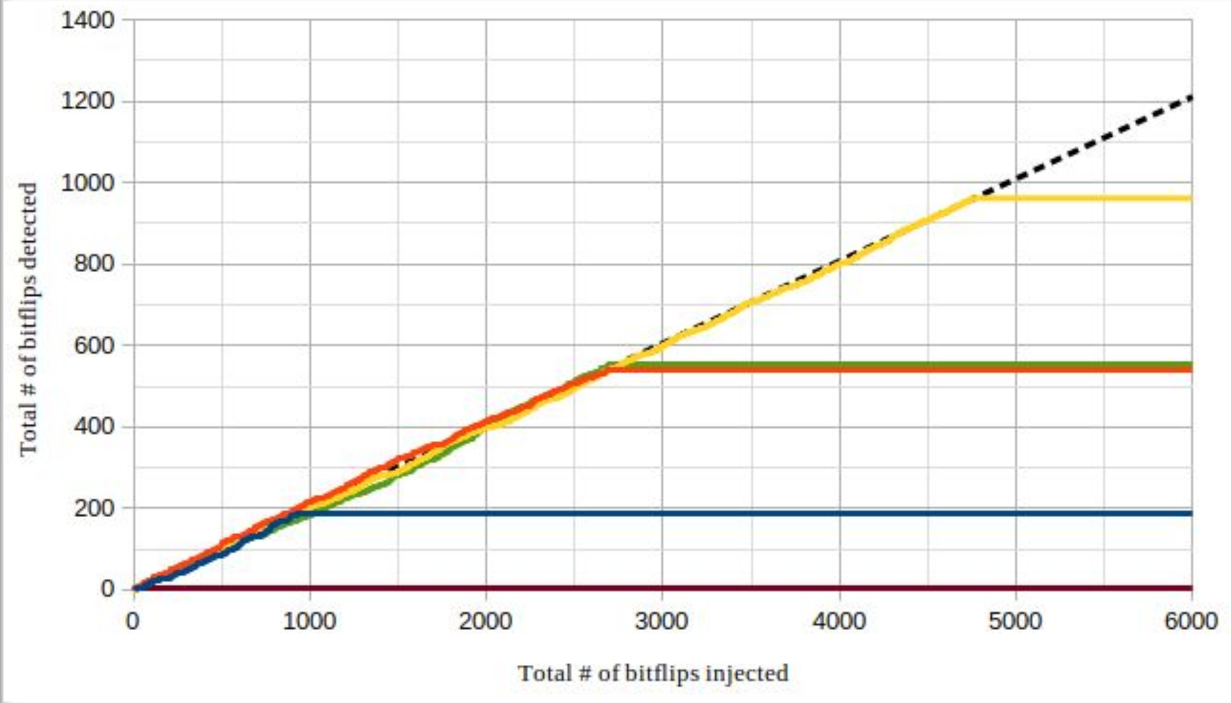
Very obscure and complicated components, I know. ;)

The approximate flow

1. Decide on the next time to inject a radiation error.
    a. We can sample a geometric distribution to model the next time that a radiation error would occur, based on the assumption that P(bitflip) is constant and independent per unit time.
2. Ask QEMU to execute the simulation up to that time.
    a. This requires a patch: while QEMU does have a sense of "virtual time," it doesn't expose any way to control it.
3. Pick a random register or memory location to inject an error.
    a. We can query the list of memory ranges from QEMU.
4. Use the regular GDB print/set commands to flip a single bit in the value.
    a. We just XOR the value by (1 << i), where 'i' is a random bit index.
5. Repeat from the top!

Testing the bitflip injections

- I wrote a simple Linux application to run in QEMU, under a regular Linux kernel.
- I gave it a big array of bytes (about 20% of the whole size of memory) and had it repeatedly scrub that memory for bitflips.
- I graphed the detected bitflips over five trials to the right.
- When the lines veer off to the right, that means the kernel crashed, and the scrubber couldn't continue to run!

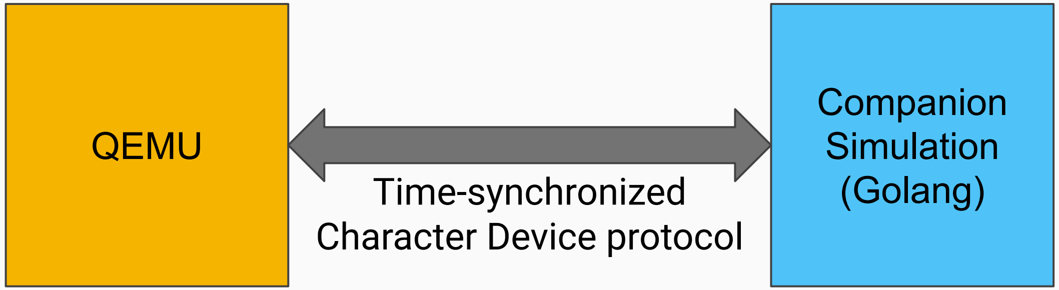


We need a target for the fault injection

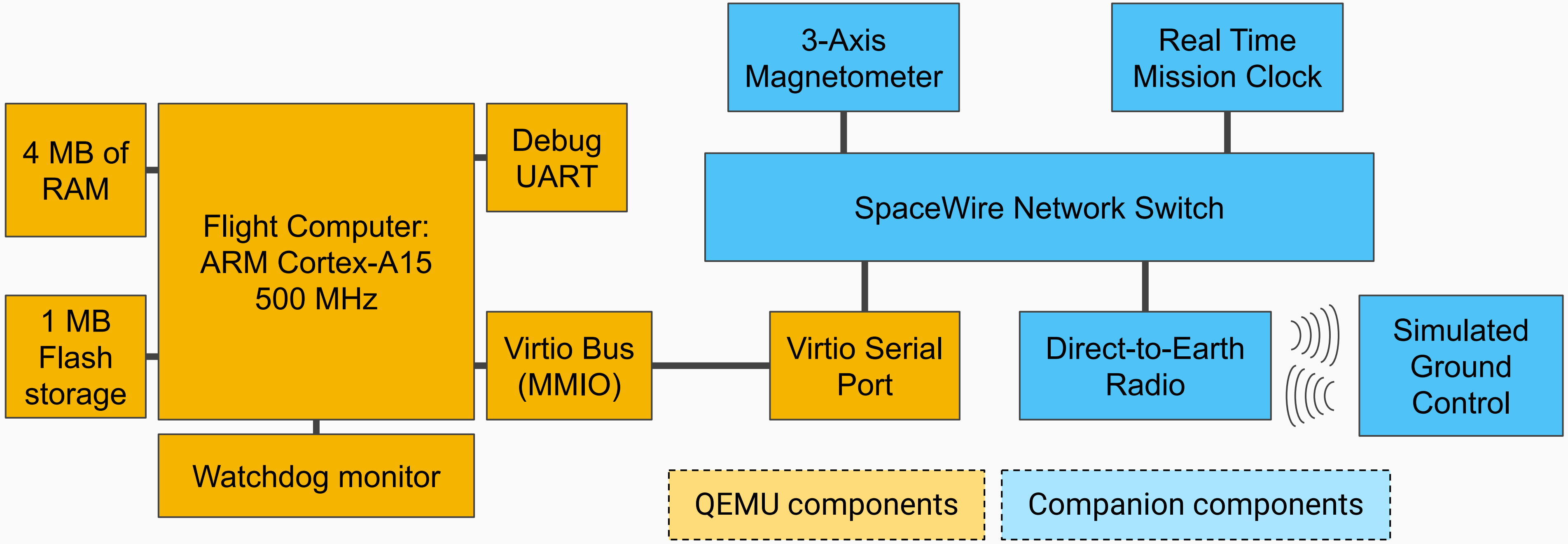
- In order to prove that we can protect spacecraft flight software from radiation errors, we will *need* spacecraft flight software to protect!
- Unfortunately, **real** flight software is generally extremely restricted, due to a combination of business competition and arms trafficking laws.
  - (Consider that the technology needed for a spacecraft bears several similarities to the technology needed for an missile.)
- Therefore: **we have to build our own!**
  - The goal will be to have flight software that places **representative demands** on the underlying operating system, not to be representative of actual vehicle control software.
  - **But first… we need a spacecraft to write software for!**

Simulation Architecture

- QEMU provides a framework for hardware emulation, but it would be difficult to model an entire spacecraft control system in its paradigm, so I built out a companion simulation.
- The normal protocol for attaching an external program as a character device did not allow for precise timing; message delivery would have randomly varied due to test host timing.
- I used a custom time-synchronized protocol to allow an external serial device to be attached: as far as QEMU is concerned, the spacecraft is simply a very odd serial port.


QEMU
Time-synchronized Character Device protocol
Companion Simulation (Golang)

Spacecraft system diagram



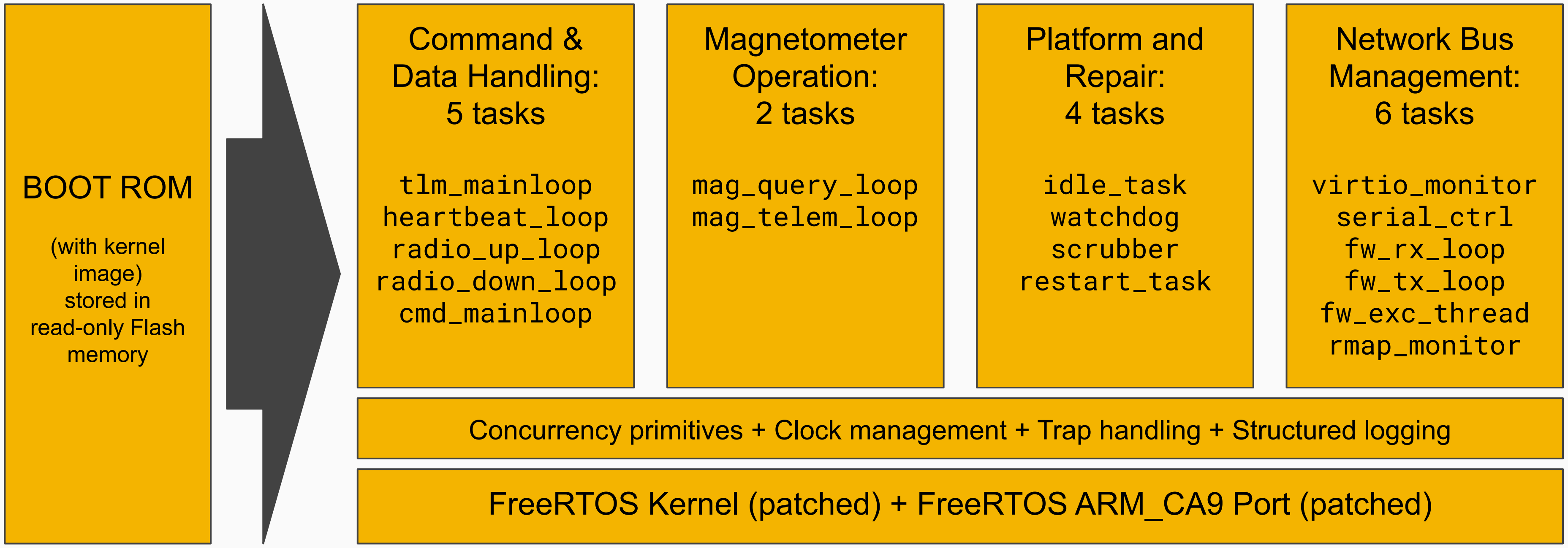
Spacecraft complexity

Here’s a simplified list of the spacecraft’s requirements:

1. Transmit telemetry to ground control via the radio
2. Receive commands from ground control via the radio
3. On command, power up or power down the magnetometer
4. While the magnetometer is powered, collect data from it at 10 Hz
5. Downlink collected magnetometer data to the ground within 10 seconds

This is around two orders of magnitude simpler than a real spacecraft’s requirements, but it should be representative enough.

Flight software overview



BOOT ROM

(with kernel image) stored in read-only Flash memory

Command & Data Handling: 5 tasks

```
tlm_mainloop
heartbeat_loop
radio_up_loop
radio_down_loop
cmd_mainloop
```

Magnetometer Operation: 2 tasks

```
mag_query_loop
mag_telem_loop
```

Platform and Repair: 4 tasks

```
idle_task
watchdog
scrubber
restart_task
```

Network Bus Management: 6 tasks

```
virtio_monitor
serial_ctrl
fw_rx_loop
fw_tx_loop
fw_exc_thread
rmap_monitor
```

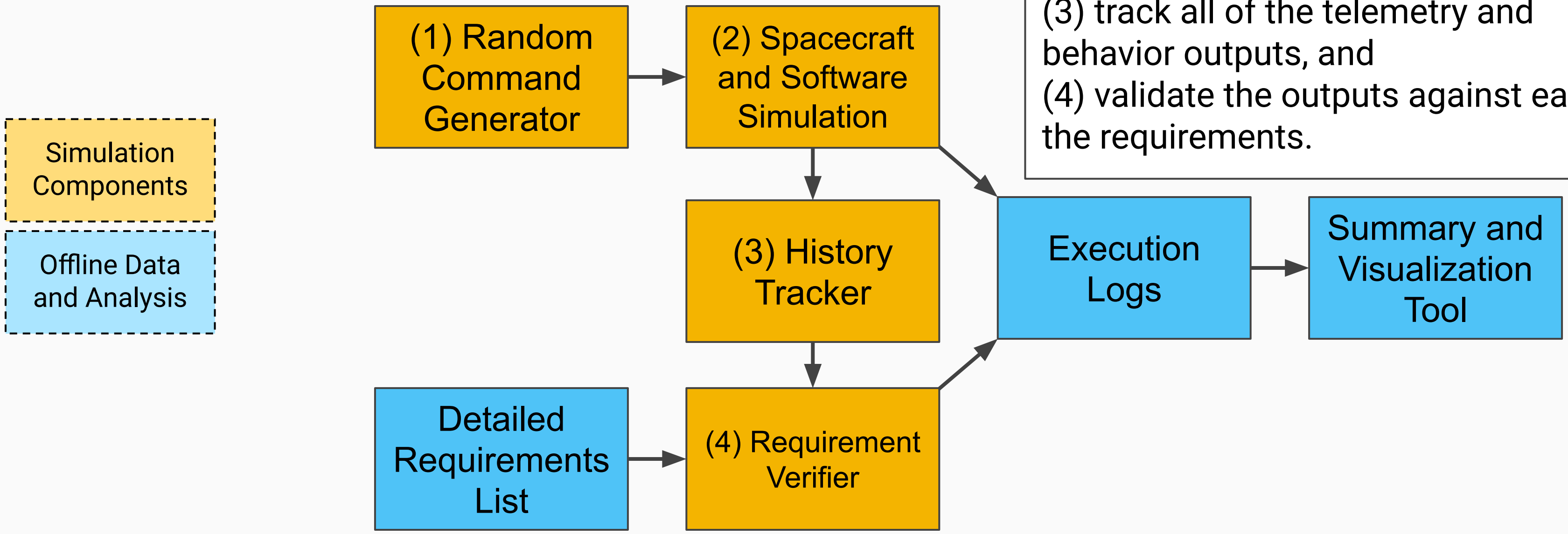
Concurrency primitives + Clock management + Trap handling + Structured logging

FreeRTOS Kernel (patched) + FreeRTOS ARM_CA9 Port (patched)

Testing and verification

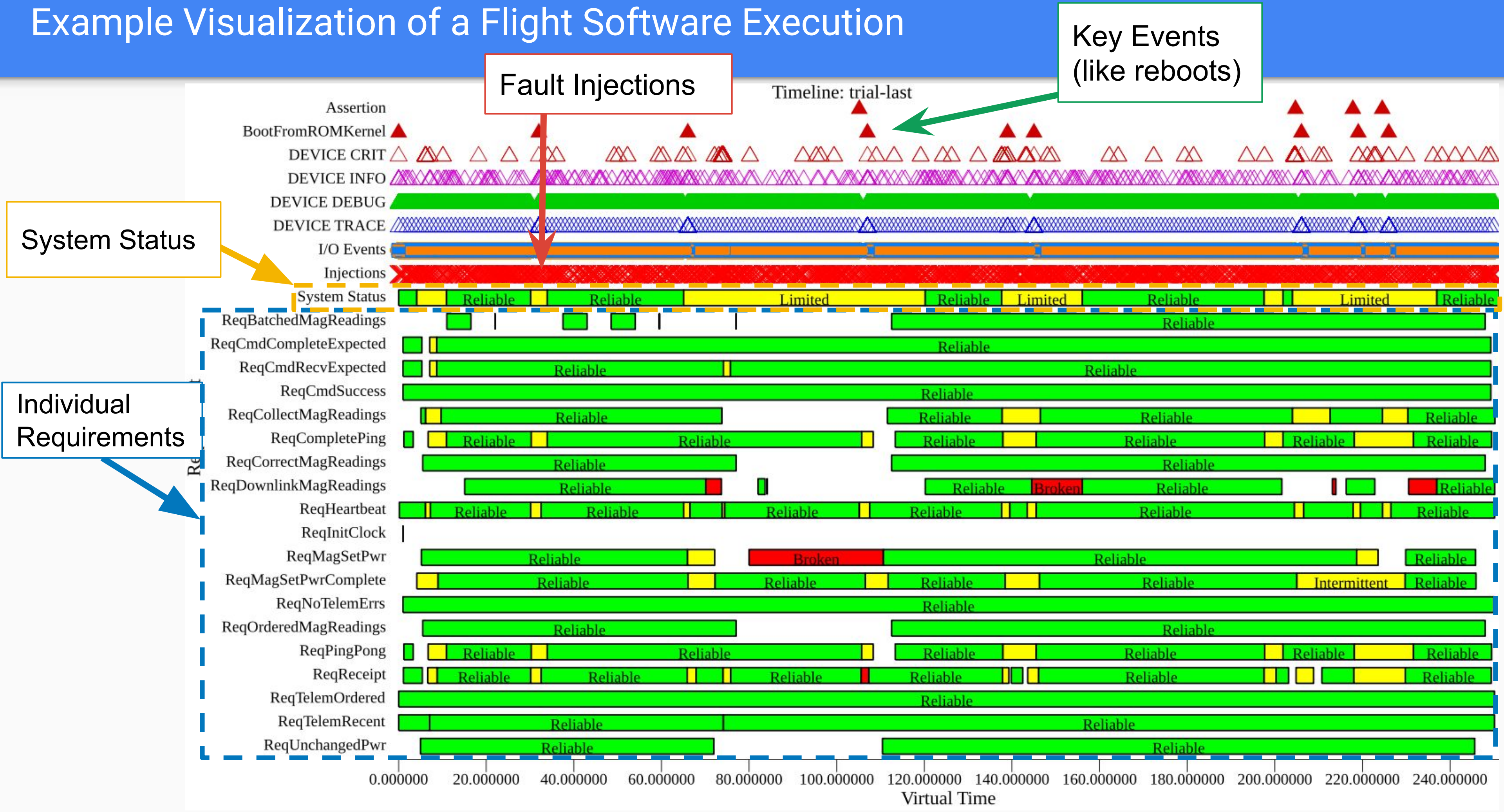
- Now we can inject errors into the software that runs a spacecraft.
- But how do we know if those errors impact the software?
- We can easily detect crashes and reboots, but those aren't *really* what we're after.
- We want to know whether or not the spacecraft software is **actually working** as we inject errors, so that we can **measure the severity of the impact** of those injected errors.
- Manual testing will not work; testing must be *automated* and *continuous*.

Automated & Continuous Testing


Simulation Components
Offline Data and Analysis
(1) Random Command Generator
(2) Spacecraft and Software Simulation
(3) History Tracker
Detailed Requirements List
(4) Requirement Verifier
Execution Logs
Summary and Visualization Tool

(1) Generate random command inputs,
(2) run them through the simulation,
(3) track all of the telemetry and behavior outputs, and
(4) validate the outputs against each of the requirements.

Example Visualization of a Flight Software Execution



Big Picture

In order to test whether a specific set of techniques for defending flight software from radiation errors work, I needed…

(1) A fault injection tool, that could inject errors into **(2)** a simulated processor, that ran **(3)** representative flight software, that operated **(4)** a simulated spacecraft, commanded by **(5)** a random input generator, and validated by **(6)** an automatic requirement verifier, which could be examined using **(7)** an execution visualization tool.

Reality check: It took around ~7-8 months to build everything up to this point.

Part 2: Fault Tolerance

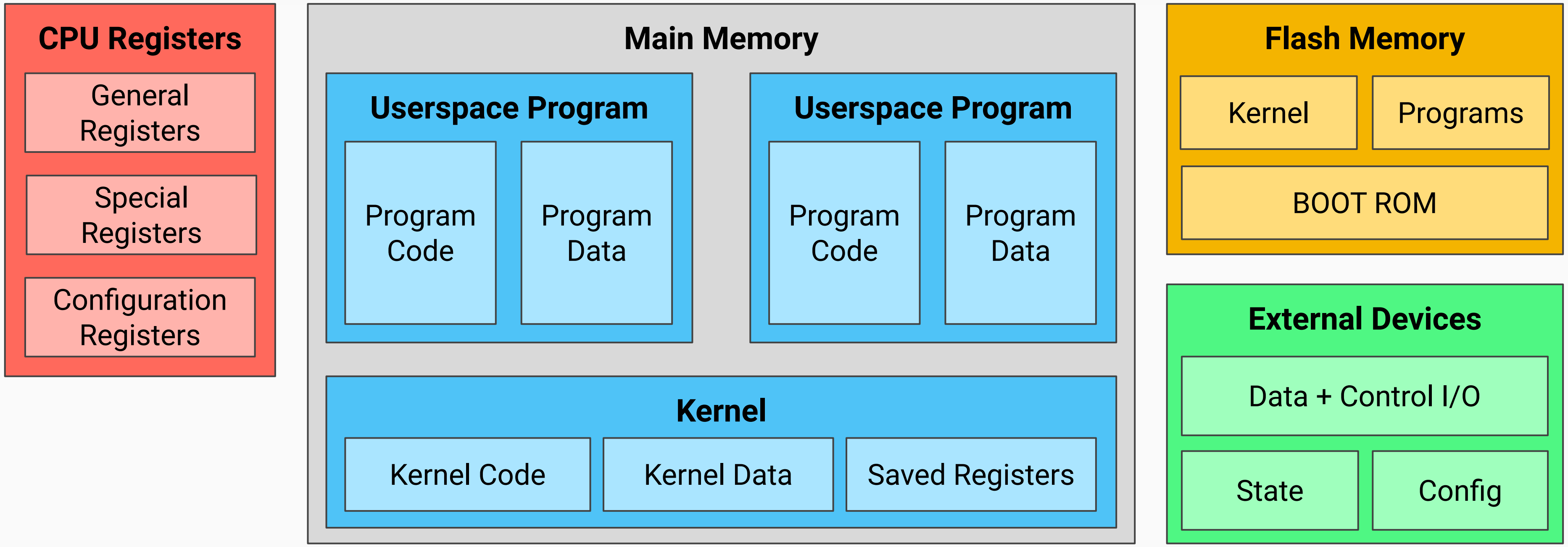
Layout of the conceptual terrain

When trying to defend a system from radiation faults, we want to consider:

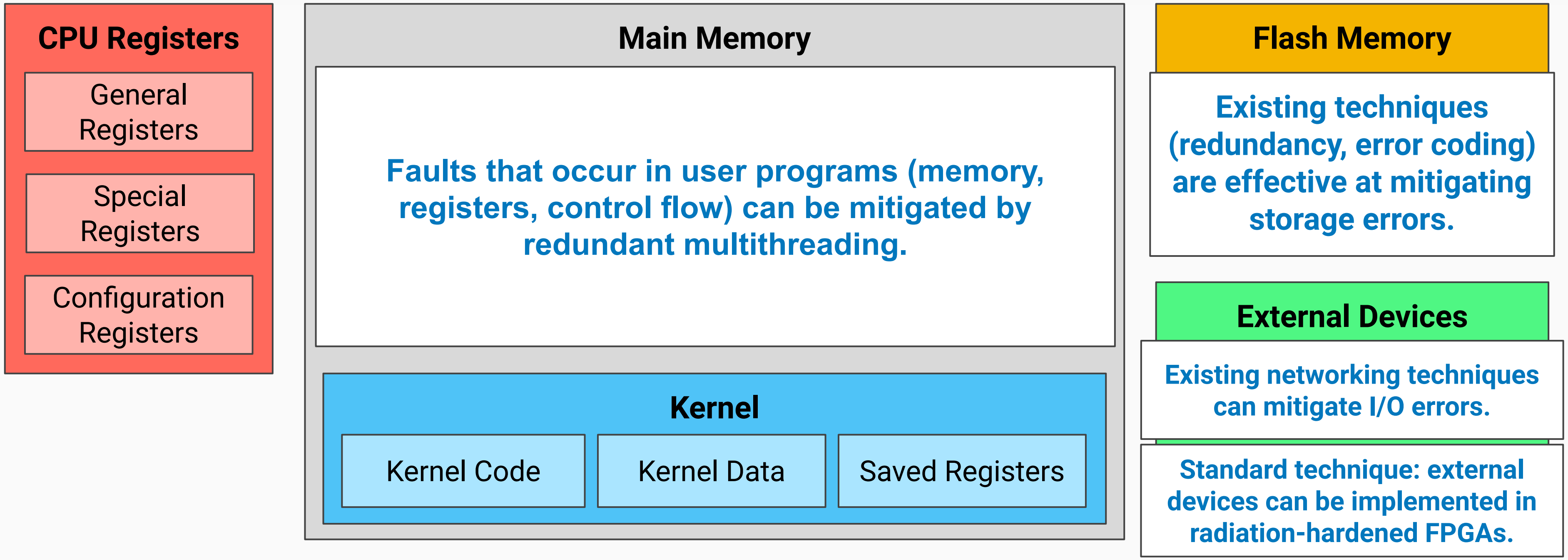
- Where can faults occur? What pieces of code and data can be corrupted?
- What types of errors can these faults cause?
- What defenses can we put in place to protect against these errors?

This is the part of the project I'm currently working on… I haven't implemented a lot of the ideas I'm about to discuss!

Fault locations: Registers, Memory, Devices



Fault locations: Registers, Memory, Devices



Faults that occur in user programs (memory, registers, control flow) can be mitigated by redundant multithreading.

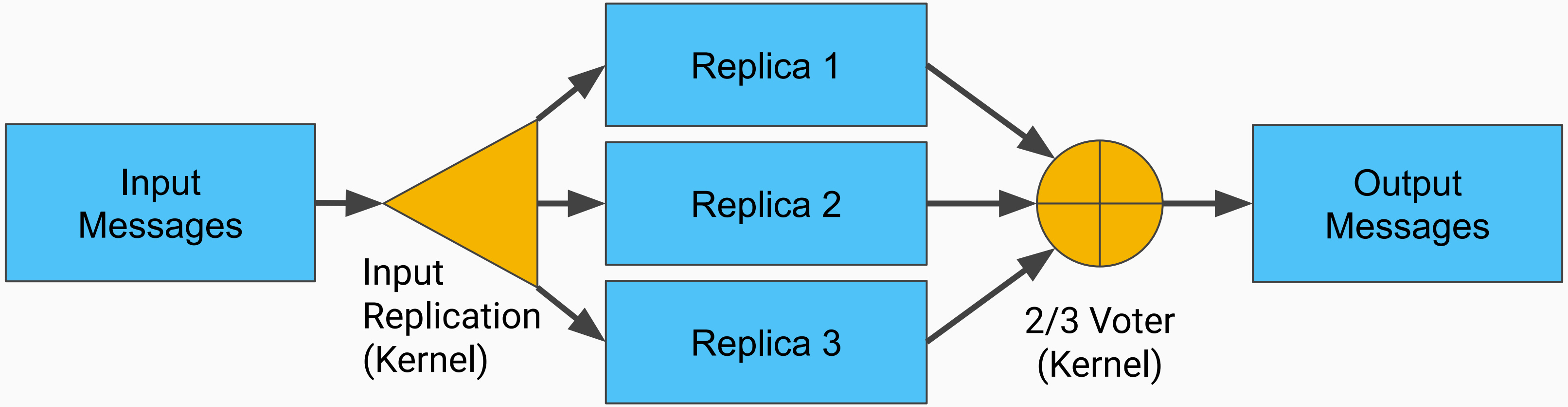
Existing techniques (redundancy, error coding) are effective at mitigating storage errors.

Existing networking techniques can mitigate I/O errors.

Standard technique: external devices can be implemented in radiation-hardened FPGAs.

Redundant Multithreading

The most common type: **Triple Modular Redundancy.** As long as two replicas agree on the correct output message, the system can operate without disruption, regardless of the presence of faults on the remaining replica.



Why does redundant multithreading work?

- Surely, it cannot accommodate the case where two replicas fail? **That's true.** But it isn't necessary.
- Radiation faults are *rare* and *randomly distributed in space and time*. Most faults flip a **single bit** in memory, and these happen rarely enough that there is plenty of time for detection and correction.
- Even if two bits (by randomness) are flipped at around the same time, this is only an issue if they are both **in different replicas of the same program**, which is also unlikely.
- Even if two replicas are affected at the same time, they are likely to be damaged in **different ways**. We can detect a three-way split in our vote.

Error modes for redundant multithreading

- What if a replica crashes? (Maybe it tries to dereference a NULL pointer.)
    - The remaining two replicas are enough to make progress, as long as they agree.
    - We still have normal process isolation capabilities in our kernel, so we can tear down the process that broke. Then, we can reload it anew from the program binary in Flash.
    - **Research Challenge:** how can we reinitialize the new program to the same state?
- What if a replica stalls? (Perhaps it gets into an infinite loop.)
    - No progress can be made by that replica, but it hasn't crashed.
    - The voter might wait forever for a message that will never come!
    - **Solution:** if the other two replicas agree, there is no need to wait. We won't normally need a stalled thread to break a tie, because of the single-error assumption.
    - After a timeout, if a result still has not been produced, we can force the replica to crash.

What’s wrong with redundant multithreading?

- The voter still needs to be hardened.
  - When replication is done in hardware, this is easier: a special hardened voter can be used. (And the processors can still be off-the-shelf.)
  - In software, this is a critical challenge: we cannot replicate a voter directly, because who would vote on the voter’s outputs?
- It’s slow: we have to run every operation *three times* to compare their results.
  - Since off-the-shelf processors are significantly faster than hardened processors, this balances out. But redundant multithreading is likely not to be very useful on Earth.
- Connecting redundant applications is a vulnerability point: the messages might be corrupted after voting, but before being replicated.

Fault locations: Registers, Memory, Devices

CPU Registers

Redundant multithreading can solve user program register errors, but not kernel errors.

Main Memory

Faults that occur in user programs (memory, registers, control flow) can be mitigated by redundant multithreading.

The kernel cannot directly use redundant multithreading, because it depends on kernel support!

Flash Memory

Existing techniques (redundancy, error coding) are effective at mitigating storage errors.

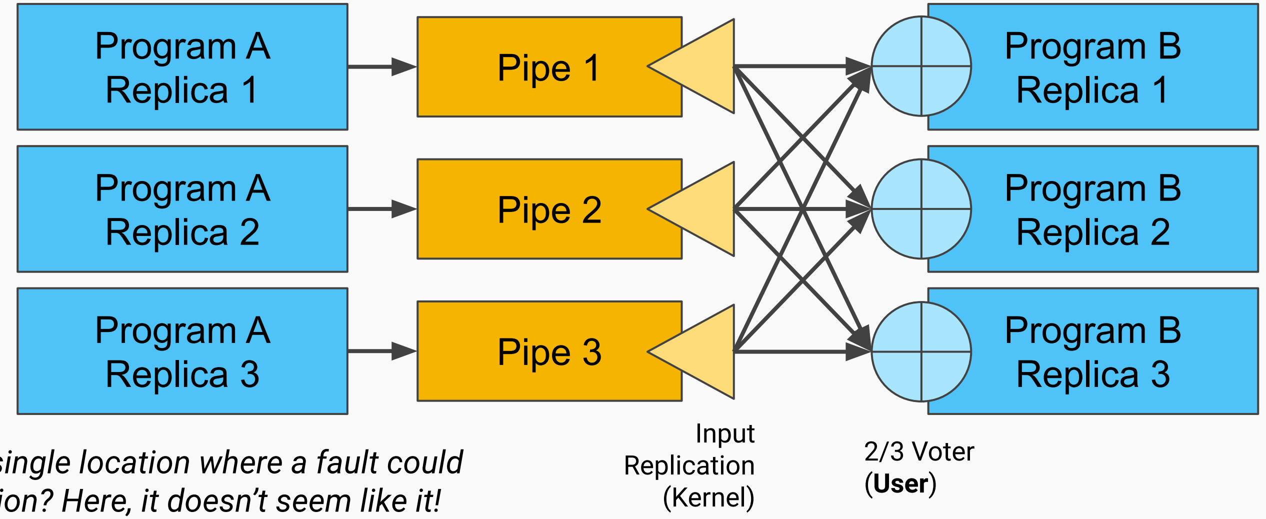
External Devices

Existing networking techniques can mitigate I/O errors.

Standard technique: external devices can be implemented in radiation-hardened FPGAs.

Possible Solution: Matrix Redundancy

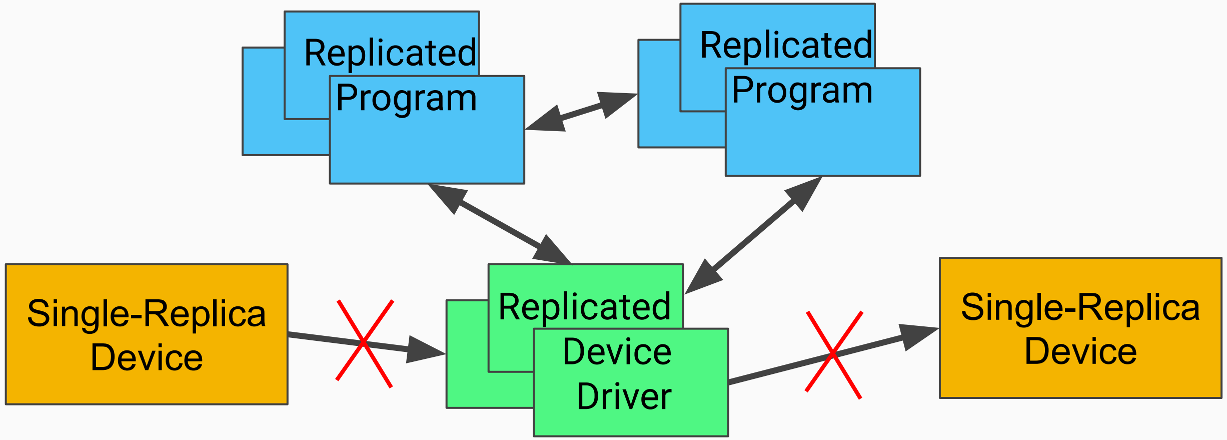
We can safely pass messages between TMR programs without fault exposure.
Notably, this eliminates the need to have a hardened TMR voter in the kernel!



Consider: is there any single location where a fault could cause incorrect execution? Here, it doesn't seem like it!

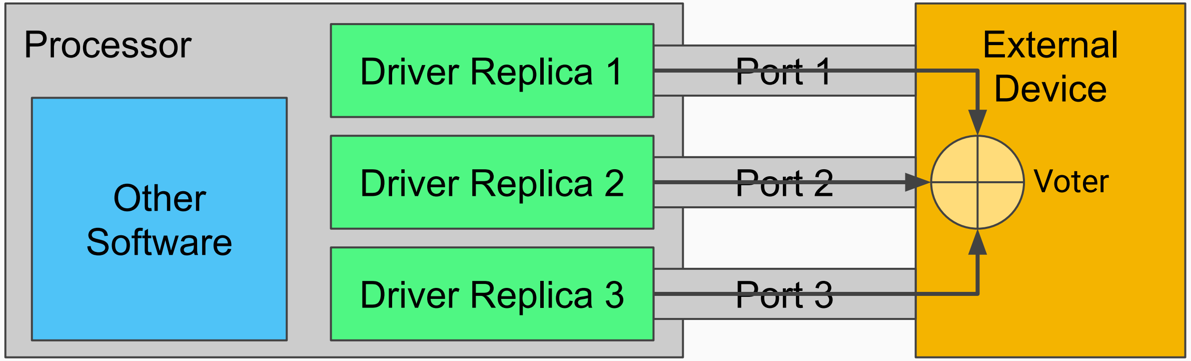
Does matrix redundancy solve our problems?

- It should solve our *internal* communication issues. But we still need an input replicator and a 2/3 voter to communicate with external devices, and those are points of vulnerability. **These are the input and output challenges.**



How can we address the output challenge?

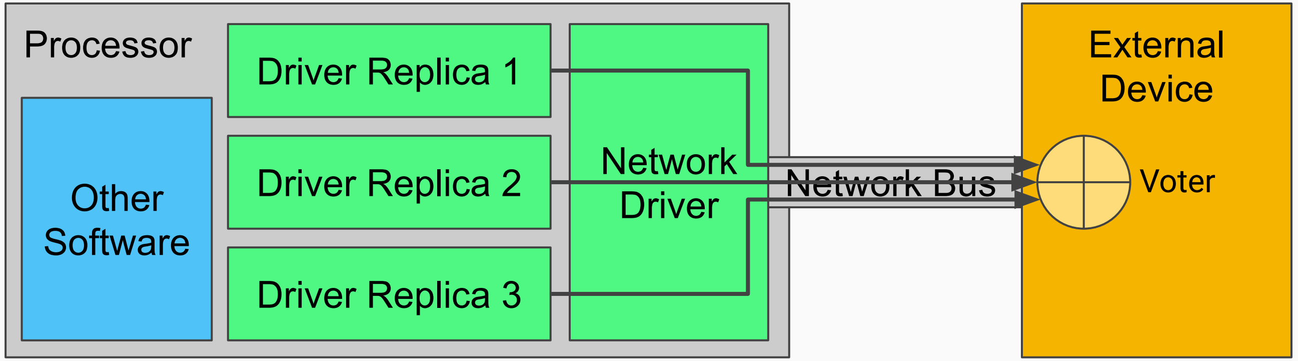
- First possibility: have three I/O ports to the device, and have the device (which is presumably hardened) compute the vote.



- This works if the device is **specially designed** for the spacecraft, and it **directly connects** to the processor. But many devices don't qualify.

How can we address the output challenge?

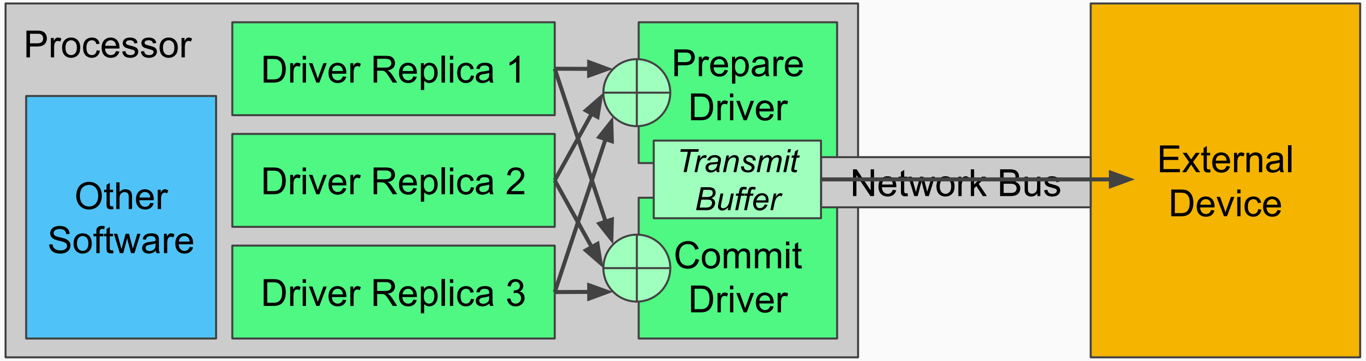
- Second possibility: if a device is connected over a network bus, then we can send **three messages**, and the device can vote.



- This works if the device is **specially designed** for the spacecraft, and if enough **network capacity** is available to triplicate the messages.

How can we address the output challenge?

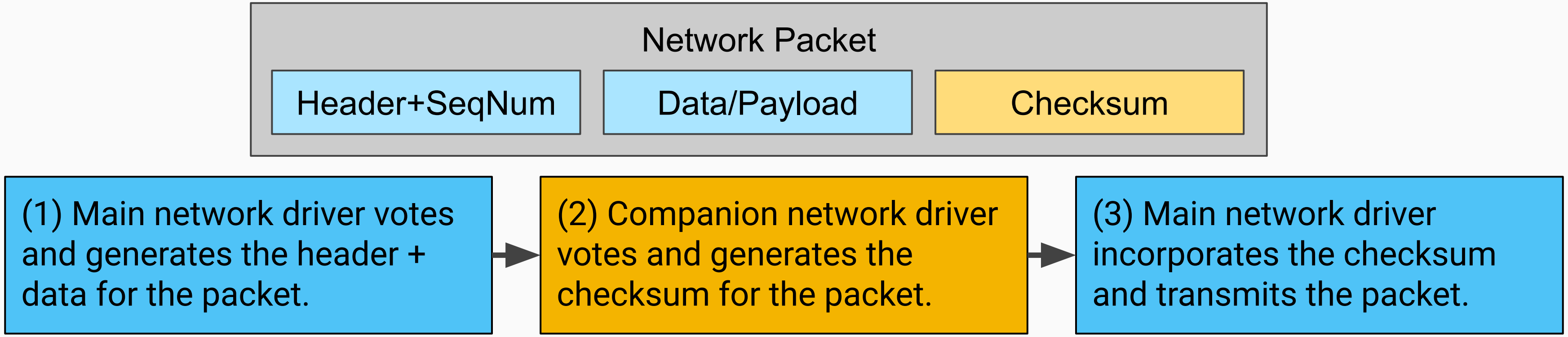
- Third possibility: we can use a PREPARE / COMMIT functionality split.



- Consider an E1000 example: PREPARE has access to descriptors, COMMIT has access to the tail register. (Descriptors are a common feature.)
- Both processes have to collaborate, so either can halt transmission if the other is malfunctioning. Both perform the same vote.

How can we address the output challenge?

- Variation on the third possibility: if the network device doesn't support separating "prepare" and "transmit" operations, we can use checksums to allow I/O validation by two separate driver processes.



Network Packet
Header+SeqNum
Data/Payload
Checksum
(1) Main network driver votes and generates the header + data for the packet.
(2) Companion network driver votes and generates the checksum for the packet.
(3) Main network driver incorporates the checksum and transmits the packet.

On error, either A) the packet will have an invalid checksum, and get rejected by the destination device, or B) the same packet will be sent, but the re-used sequence number will let it be rejected.

And what about the input challenge?

Similar options:

- Replicated input ports
- Replicated network messages
- Checksum validation

Checksum validation is probably the easiest approach for network-based devices. Note that we need to be able to handle the case of an invalid checksum: normally, we can do this using the same code we already need for handling dropped messages.

What’s left?

- We have a way to protect state within each process.
  - Redundant Multithreading.
- We have a way to protect messages between processes.
  - Matrix Redundancy
- We have a way to protect device I/O
  - Several ways, in fact.
- **But we still haven’t protected the kernel itself.**
  - We’ve eliminated some of the jobs that the kernel needs to fulfill, so that they can be protected using our existing approaches, but there’s a lot left.

Errors in kernel code

- What happens if there’s an error in kernel code that causes misbehavior?
  - We can have a scrubber task that corrects the bitflip in the code region.
  - **But this doesn’t correct the misbehavior itself; only limits its duration.**
- What if the scrubber stops working?
  - We can have a second scrubber. As long as one works, we’re okay.
- What if there’s an error in kernel code that causes a crash or hang?
  - We can attach an external “watchdog device.” If a watchdog task doesn’t feed the watchdog every time unit, the watchdog forces the flight computer to reset.
- What if the kernel only schedules the watchdog task?
  - We can have the watchdog device monitor the scrubbers. If they don’t get a chance to scrub, it refuses to feed the watchdog, and the flight computer gets reset.

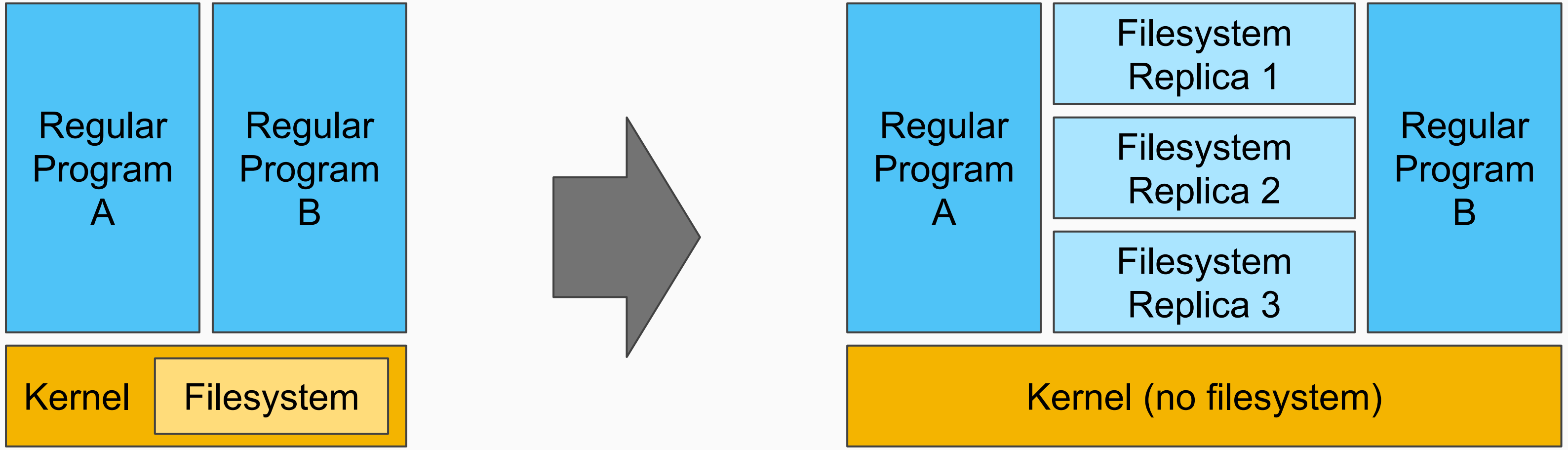
What is the kernel still doing?

This is where we need to worry about data corruption and misbehavior:

- Bootstrapping
- Task scheduling
- Saving and reloading program registers
- Inter-process communication (pipes)
- Memory allocation
- Process loading and restarting
- Filesystem handling

First: Filesystem handling

- We can use the **microkernel strategy**. We can move the filesystem into a user process, so that it can be protected by redundant multithreading.



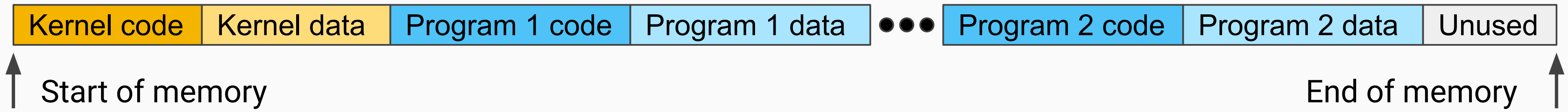
What is the kernel still doing?

This is where we need to worry about data corruption and misbehavior:

- Bootstrapping
- Task scheduling
- Saving and reloading program registers
- Inter-process communication (pipes)
- Memory allocation
- Process loading and restarting
- ~~Filesystem handling~~

Second: Memory Allocation

- Fun fact: flight software usually doesn't allocate and free memory!
  - Imagine if your plane ran out of memory and fell out of the sky. It would be bad!
  - In order to solve this, flight software *pre-allocates* all of the memory that could be needed for each task. All data structures have fixed or maximum sizes.
- Therefore, we can avoid needing runtime memory allocation.
- We can allocate memory to each process during compilation.
  - We can pre-generate the page tables for all of the processes based on known sizes.
  - The kernel doesn't need to do *anything* at runtime except load existing tables.



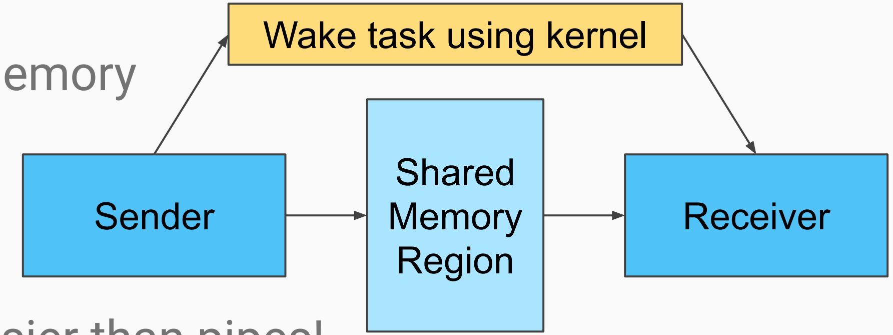
What is the kernel still doing?

This is where we need to worry about data corruption and misbehavior:

- Bootstrapping
- Task scheduling
- Saving and reloading program registers
- Inter-process communication (pipes)
- ~~Memory allocation~~
- Process loading and restarting
- ~~Filesystem handling~~

Third: Inter-process communication

- Because we know the full list of processes we need, we can also know the complete list of IPC channels we need.
- Instead of building IPC data structures in the kernel, we can offload most of the work to user space.
- Data can be passed via shared memory regions, which can be baked into the page tables.
- Kernel ONLY needs to support waking up another task: much easier than pipes!



What is the kernel still doing?

This is where we need to worry about data corruption and misbehavior:

- Bootstrapping
- Task scheduling (including wakeup requests)
- Saving and reloading program registers
- ~~Inter process communication (pipes)~~
- ~~Memory allocation~~
- Process loading and restarting
- ~~Filesystem handling~~

Fourth: saving and reloading program registers

- The kernel does have to save registers. There’s not much of a way around that.
- But it **doesn’t** have to reload them.
- If the program registers are saved at a known location *in program memory*, then the program can reload them itself.
- This way, the scheduler doesn’t need to track register state in its memory. It only needs to jump to a predefined entry point.
- Less data in the kernel -> lower chance of kernel errors.

What is the kernel still doing?

This is where we need to worry about data corruption and misbehavior:

- Bootstrapping
- Task scheduling (including wakeup requests)
- Saving ~~and reloading~~ program registers
- ~~Inter process communication (pipes)~~
- ~~Memory allocation~~
- Process loading and restarting
- ~~Filesystem handling~~

Fifth: process loading and restarting

- Instead of trying to load program code separately, we can load it as part of the kernel’s code.
    - **Because the kernel’s code is already protected by a scrubber, we can avoid ever having to explicitly detect and reload code for any program in the kernel.**
- We’ll still map it into the program’s address space as normal; the program won’t even know.
- Now the only thing we have to reload/restart is the program’s data.
    - **In combination with leaving reloading of register state to the user, we can avoid doing this in the kernel.**
    - The fixed entry point used for restoring the user process is protected by the scrubber.
    - **We can safely leave the decision of whether to restart to the program itself.**

What is the kernel still doing?

This is where we need to worry about data corruption and misbehavior:

- Bootstrapping
- Task scheduling (including wakeup requests)
- Saving ~~and reloading~~ program registers
- ~~Inter process communication (pipes)~~
- ~~Memory allocation~~
- ~~Process loading and restarting~~
- ~~Filesystem handling~~

Those last three requirements are hard to eliminate.

Our new system call API

Complete list of system calls:

1. `wake_up(process_id)`

… That’s it.

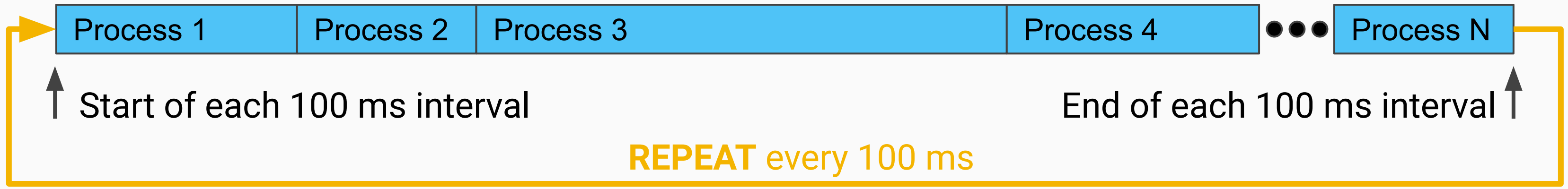
Handling the remaining errors

Now that we've shrunk the kernel to nearly nothing, we need to think about how we'll handle errors in the remaining elements:

- Bootstrapping
  - If we fail to bootstrap correctly, the watchdog will starve, and the flight computer will reset.
- Task scheduling
  - If the kernel code is broken, then either the scrubbers will resolve the error, or the watchdog task will refuse to feed the watchdog (because the scrubbers aren't running), or the watchdog will starve because the watchdog task isn't running.
  - **But we still have to think about the *mutable scheduler state*.** ← *big remaining challenge*
- Saving registers
  - This is equivalent to injecting an error into the process, which we already handle.

Corrupted scheduler state (runqueues, etc)

- What if we accidentally drop a task from our scheduling queue?
- We can't (easily) scrub mutable data. Only immutable data.
- So… what if we eliminated this piece of mutable data?
  - We've already eliminated everything else.
- **Let's build a hardcoded schedule, and repeat it.**
  - It's immutable, so we can scrub it.
  - We can simply look up the next process to run based on the processor clock!



Is that too restrictive?

- Probably.
- We don't want to have to wait 100 ms to hear back from a process.
    - We can schedule a process multiple times per 100 ms, but that might not be enough.
- **But we don't have to – we still have our wake_up API!**
    - We don't need to modify scheduler state for this… we can just directly switch our context to the target process of the wake_up call!
    - This is basically allowing one process to "donate" their time to another process.
- If we want to more easily reallocate time, if there's extra, we can pass this responsibility to a **userspace** scheduler.
    - Every task just donates its unused time to the "scheduler" task, which needs no special privileges. The scheduler task donates its received time as it sees fit.

Does that solve all of our issues?

- Not quite.
- We have one more thing we might want to worry about.
- **The only way we can handle many of our possible kernel errors is by letting the watchdog reboot the flight computer.**
    - There's no way around this – there are always ways for the CPU to get stuck.
- **Rebooting a flight computer can take a very long time.**
    - There are lots of processes that need to be restarted. Even just reloading their code can take a long time!
- **Even worse, we really don't want to lose all of our program state.**
    - Redundant multithreading can't help us if every program crashes simultaneously.
- Key question: *is there any way for us to restart faster?*

One final technique: kernel-only reboots

- **Let’s reload the kernel *without* reloading the processes running on it.**
- RAM only necessarily gets cleared on power failure.
  - If we reset due to a crash or watchdog action resulting from a radiation error, program memory should generally still be intact. We usually clear it on boot. **But we don’t have to.**
- As long as we know where in memory we can find all of the state from the previous boot, we can *resume executing it from exactly where we were.*
  - Individual processes can decide to restart themselves on their own schedule if they got corrupted, but the rest of the system will continue operating.
- If we bake all of our memory allocations into the kernel at compile time (as discussed before), then we can trivially find all of the state!
  - And we don’t even need to move it, because it’s already in the right place!

The big picture

- In theory, with the combination of mechanisms just discussed, we should be able to build an operating system – and accompanying flight software – that is *extremely* robust to radiation errors.
- This operating system could run on off-the-shelf hardware with no radiation protection.
    - (At least, with no protection for single-bit upsets… total ionizing dose is another matter, but that problem is solved with very different approaches.)
- In theory, this *could* end up being a cheaper and more reliable approach to protecting flight software from radiation errors than the existing approaches. (Maybe.)

But does it actually ***work?***

Great question!

That’s what I’m trying to answer right now in my research. These are complicated techniques, and it takes time to evaluate them.

Maybe some of these ideas don’t actually work at all!

Until someone tries them, we can’t know for sure.

Questions?

Source code for my thesis work so far is publicly available at:

github.com/celskeggs/hailburst

(Note: source code is not cleaned up for ease of use. You might not be able to run it without a lot of work.)