Reliability

1. Questions from reviews:
    a. Memory overhead?
    b. Why hard to prevent infinite loops
    c. Concurrent access to a kernel object?
        i. Acquire kernel locks
    d. Policy for object tracking?
    e. Impact of isolating multiple drivers?
    f. Deferred xpc?
    g. How evaluate reliability?
2. Intro
    a. Reliability: how long do you execute before a failure
        i. MTTF
    b. Availability: what is probability if you request service you get it
        i. MTTF / MTTF + MTTR
        ii. How make high availability?
            1. Make MTTF big (highly reliable) or MTTR small (fast to repair)
        iii. 99% ~3 days
        iv. 99.9% ~9 hours
        v. 99.99% ~1 hour
        vi. 99.999% ~5 minutes
        vii. 99.9999% ~30 seconds
    c. What is cost of an hour of downtime (in 2002)?
        i. Brokerage: $6,000,000
        ii. Ebay: $225,5000
        iii. Cell phone activation: $41,000
        iv. Home shopping channel: $113,000
    d. What is MTTF for a disk?
        i. 900,000 hours – 10 years
    e. What is MTTF for an OS?
        i. Windows 2000: 72 weeks
    f. Failures
        i. Terminology:
            1. Fault = bug in code
            2. Error = erroneous state as a result of executing code

            a. Latent errors: executed fault but did not cause failure yet
        3. Failure = system does not act according to its specification
    ii. Types
        1. Bohr bugs / deterministic bugs:
            a. Bugs that recur every time you do something – easily repeatable / predictable / can be tracked down and fixed / often found in testing
        2. Heisenbugs / nondeterministic bugs
            a. Bugs that don't recur every time / caused by an unlikely combination of events / hard to reproduce and repair
    iii. Causes of failure (old data)
        1. Hardware (cpu, devices) – 18%
            a. Fix: redundancy
        2. Environment (network, power) – 14%
            a. Fix: redundancy
        3. Software (OS, applications) – 25%
            a. Fix: that is what we will talk about
        4. Operations (maintenance, administration) – 42%
            a. Fix? MS reports 20% of the time an admin goes into a data center they mess something up. Solution: zero-maintenance systems
            b. Should you patch? Not if you haven't had problems yet (and it isn't a security problem…)
    iv. When do failures occur?
        1. Infant mortality – new, under tested
        2. Norma lifetime – highly reliable
        3. Wear-out period (for HW) – things break physically, or (for SW) assumption about world have changed too much
    v. Failure models – Why important?
        1. Timing failures occur when a component violates timing constraints.
        2. Output or response failures occur when a component outputs an incorrect value.

            3. Omission failures occur when a component fails to produce an expected output.
            4. Crash failures occur when the component stops producing any outputs.
            5. Byzantine or arbitrary failures occur when any other behavior, including malicious behavior, occurs

    - vi. Synthetic failure models
        1. Halt on failure
        2. Failure status
        3. Stable Storage
    - vii.
- g. Approaches:
    - i. Fault Avoidance: make sure failures don't happen
        1. Fault prevention: write code without bugs
            - a. better languages
            - b. better software engineering
            - c. tool usage during coding process
            - d. e.g. write a new OS in a new language, prove properties of implementation
        2. Fault removal: remove bugs from code
            - a. e.g. run testing tool (valgrind, purify)
            - b. windows static driver verifier – find bugs statically
        3. Fault workaround: make sure failures don't execute
            - a. Firewall / virus detector
            - b. "It hurts when I run" "don't run"
    - ii. Fault Tolerance
        1. Allow failures to occur, but keep system running
        2. Basic ideas:
            - a. Fault detection – figure out that something bad happened
            - b. Isolation – keep bad state from spreading to whole system
            - c. Recovery – get the bad part back into a good state
        3. Basic approaches to error detection

a. Check dynamically for error conditions and inconsistencies to detect failures early
b. Use heart beats to make sure a module is still executing
c. QUESTION: how easy it to do this generically?
    i. QUESTION: as code evolves?
    ii. QUESTION: at what cost?

4. Basic approaches to isolation
    a. Decompose into modules
        i. Unit of failure is small
    b. Check each module for errors
        i. Fails fast – doesn't spread corruption
        ii. Isolate from other modules
    c. Hardware / software boundaries around modules
        i. Whole machine
        ii. address space
        iii. extra instructions
5. Basic approaches to recovery
    a. Restore system to a functioning state
        i. E.g. configure extra modules to take over for failed module, restart failed module
    b. Forwards / Backwards
    c. Concealing / revealing
    d. Basic approaches:
        i. Logging / retry
        ii. Checkpoint / restore
        iii. Replicate (process pairs)
        iv. Alternate versions
        v. Transactions (undo)
        vi. Reveal faults up the stack
    e. Redundancy: do things twice or more (or store things more than once)
        i. On two machines
        ii. In two processes
        iii. In two places (state in memory / on disk checkpoint)

iv. At two times (e.g. checkpoint / restore)
v. QUESTION: what kinds of bugs are handled?

f. Diversity: do things multiple different ways
   i. Different platforms
   ii. Different implementations
   iii. Idea: unlikely to have common failure modes
   iv. Name: n-version programming, recovery blocks

6. Basic questions for fault tolerance: where do you do the fault tolerance?
   a. In the hardware (e.g. two processors, RAID with multiple disks)
   b. Between the HW and the OS (e.g. virtual machine)
   c. Within the OS
   d. Between the OS and the application
   e. Within the application

Nooks

A. What is this paper about?
   a. Making drivers more reliable?
   b. A kernel hack for stopping drivers from crashing the system?
   c. An approach to fault tolerance?

B. Paper 2 - who read it?
   a. Compare it with nooks
      i. Nooks has an architecture, a design
      ii. Paper 2 has low-level details (e.g. sequence of calls); does not help you understand what is happening
      iii. Evaluation ; microbenchmarks, not apps
      iv.

C. General approaches to fault tolerance
   a. Fault avoidance: execute only correct code
      i. Fault Prevention: write good code
         1. Type-safe languages
         2. Software engineering
      ii. Fault removal: remove bugs from code
         1. Code reviews

        2. Testing
        3. Bug-finding tools (e.g. Coverity)
        4. Model checkers
    iii. Fault work-around: don't execute the bad code
        1. Firewalls
        2. Virus prevention systems
b. Fault tolerance: let things fail but clean up afterwards
    i. Reboot / restart in a process
    ii. Do it enough times that someone succeeds (redundancy / modularity)
c. Which is better?
    i. Can you prevent or remove all the bugs?
    ii. What if your hardware has problems or users are buggy?

1. Nooks
    a. Approach:
        i. Improve reliability by tolerating dominant cause of failure
            1. Don't bother making everything reliable
            2. Try to make it integrate well with existing OS
            3. Make it compatible with existing drivers / OS /applications
        ii. Key pieces
            1. Isolation / fault containment: prevent driver from corrupting os/application
            2. Failure detection
            3. Recovery: get driver running again after a failure
2. How do modularization?
    a. Device drivers
    b. Existing modules
    c. Known to cause errors
3. How do Isolation
    a. Isolation
        i. LW Kernel Prot Domains
        ii. Prevent driver from writing to OS
        iii. Allow writes to driver-private data
        iv. XPC – invoke code in another domain
    b. Interposition
        i. Inject code transparently
        ii. Like VMM – but boundary is kernel/driver

            iii. Done at load time, not compile time
                1. Note: can choose where to put it!
            iv. Wrappers on driver/kernel interface
            v. Result:
                1. Recompile driver because binary interface changes (macros -> functions)
                2. Pretty much no code changes to drivers
            vi. QUESTION: What happens when modules invoke other modules?
    c. Object tracking
        i. Allow safe-sharing
        ii. Validate shared parameters
        iii. Map between kernel and driver-private data
        iv. QUESTION: What happens on a multi-processor?
4. How detect failures?
    a. HW: processor fault
    b. SW
        i. Bad parameter
        ii. Excessive resource consumption
    c. External
        i. Human
        ii. SW agent

5. How do recovery?
    a. Normal approach (without nooks): what is it?
        i. Reboot
    b. Alternatives:
        i. unload driver
    c. Restart driver
        i. Unload completely
        ii. Prot domains, obj. track allows completely unloading w/o driver help
            1. Like a process can clean up for itself
        iii. Restart driver
            1. Needs user-level knowledge of how to restart
            2. Issues: where does configuration data come from?
                a. Solved in shadow drivers
6. Issues:
    a. Performance overhead
        i. Where does it come from?
            1. New code in system

            a. Wrappers
            b. Object tracking
            c. Domain change (change page table)
        2. Existing code running slower
            a. More TLB misses
            b. More cache misses due to copying
  b. Implementation overhead
  c. Dependence on interface stability
  d. Assumptions
    i. Are drivers fail stop?
        1. What if driver writes bad data to device?
    ii. Are driver failures heisenbugs?
    iii. Can we virtualize this interface? Is it too ugly?
  e. What happens to applications?
  f. QUESTION: What is real contribution?
    i. Pointing out that drivers are the problem
    ii. Pointing out that compatible driver isolation is possible
    iii. Pointing out that driver isolation can have reasonable performance
    iv. Pointing out the importance of recovery

7. Evaluation
  a. Fault-injection for testing ability to detect faults / recover
    i. QUESTION: is this a good technique?
    ii. QUESTION: What do we learn from these results?
        1. Nooks stopped the faults we injected
    iii. What are the limitations?
        1. How realistic are faults?
            a. Didn't wait a long time for faults to have an effect
        2. How realistic is the fault distribution?
            a. Uniform distribution across fault types
        3. How realistic was recovery?
            a. Reloaded same code w/o faults
  b. Performance
    i. Need to show speedup / CPU utilization separately
    ii. Else cpu increase is masked for non-cpu bound tests
    iii. QUESTION: what about multiple drivers at once?
  c. Complexity:
    i. 22,000 lines of code. Is this a lot or a little?
  d. QUESTION: how do you balance performance drops and increases in reliability / availability?

8. QUESTION: What is your take?
    a. Paper issues
        i. Could have written paper as "How to make Linux device drivers execute reliably"
            1. Talk about changes to Linux data structures
        ii. Instead, presented as:
            1. Architecture
                a. Generic approach, not many choices
                b. E.g. could use virtual machines, could use software fault isolation, could use java
            2. Implementation
                a. Specific set of choices, specific OS, specific isolation technique
        iii. Makes paper more general, stronger

3.