ECE 254 / CPS 225
Fault Tolerant and Testable Computing Systems

Real Systems: Fault Tolerant Software

Copyright 2011 Daniel J. Sorin
Duke University

Outline

- Software Faults, Errors, and Failures
- Static Methods for Finding/Avoiding Software Faults
- Dynamic Software Error Detection
- Fault/Failure Isolation
- Interactions Between SW and HW Fault Tolerance

Software

- There are many kinds of software
- System software
  - Operating system (Windows, Linux, Solaris, Android, etc.)
  - Device driver (for printer, network card, etc.)
  - Compiler (gcc)
  - Library (DLLs)
- User-level software
  - E.g., simulator, word processor, spreadsheet, game, etc.
- Internet software
  - Google
  - Facebook
  - YouTube
  - Etc.

Software Faults/Errors

- We all know that software has bugs
- Types of bugs (or errors/failures that are due to bugs)
  - Incorrect algorithm
  - Array bounds violation
  - Memory leak (C, C++, but not Java)
    - » Allocating memory, but not deallocating it
  - Reference to NULL pointer (C, C++, but not Java)
  - Incorrect synchronization in multithreaded code
    - » Allowing more than 1 thread in critical section at a time
    - » Blocking when holding a lock
  - Inability to handle unanticipated inputs
- Any other bugs that we recall from the Oppenheimer paper ("Why Internet Services Fail")?

Operating System Errors

- "An Empirical Study of Operating System Errors" (Chou et al., SOSP 2001)
- Studied 21 different versions of Linux over 7 years
- Used static compiler analysis to find bugs
  - Check for NULL pointers
  - Do not block a thread when interrupts are disabled or lock is held
  - Several other checks, including array bounds checking
- Discovered that vast majority of bugs are in drivers
  - Some also in:
    - » arch/i386/ (architecture specific code)
    - » net/ (network code)
    - » fs/ (file system)
  - Why are drivers a primary culprit?

OS Bug Characteristics

- Bugs tend to be in larger functions
  - More complexity → more chance of a bug
- Tend to be clustered in small number of files
  - A bad programmer tends to mangle a small number of inter-related files
- Tend to be more prevalent in newer code and code that is less thoroughly integrated and tested
  - Code hacked onto the side of the main code is problematic
  - Helps to explain why drivers are so buggy

Software Failures

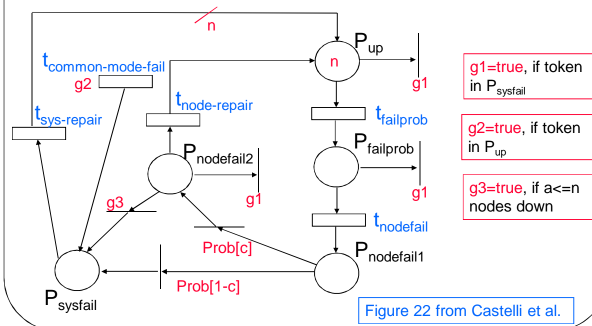
- What happens if we exercise a software bug? What kind of failures can occur?
- User-level software
  - Incorrect data
  - Livelock/deadlock (can escape with Control-C)
  - Exception that triggers OS to kill process
    - » Segmentation fault
    - » Bus error
- Operating system software (including device drivers)
  - Livelock/deadlock (may be able to escape with Ctrl-Alt-Delete)
  - Crash and reboot
  - Incorrect I/O

Software Aging

- "Proactive Management of Software Aging" (Castelli et al.)

SPN Example From Software Aging

Models system with n nodes, initially all up. Tokens represent nodes.



Outline

- Software Faults, Errors, and Failures
- Static Methods for Finding/Avoiding Software Faults
- Dynamic Software Error Detection
- Fault/Failure Isolation
- Interactions Between SW and HW Fault Tolerance

Static (Pre-Release) Fault Detection

- **As with hardware, can try to find faults before shipping the product**
  - Design reviews
  - Formal verification
  - Testing
- **Can try to add in redundancy to mask potential faults**
  - N-version programming
- **Can try to proactively "scrub" the software to remove latent errors (due to aging) before failures occur**
  - Software rejuvenation
  - (unclear if this is really static or dynamic)

Fault Avoidance with Design Reviews

- **Having one person write a piece of software with no input or oversight is a bad idea**
- **Real software engineering involves careful planning and periodic design reviews**
- **If you haven't read Fred Brooks' "The Mythical Man-Month", you should!**

Static Fault Detection with Formal Verification

- Formal verification is a systematic, mathematical way to prove that a system (software or hardware) is correct
- Correctness is based on a specification
  - E.g., the ISA of a microprocessor is its specification, and we would want to formally prove that the latest & greatest new implementation of this ISA, say the Pentium 13, is correct
- Two broad approaches for formal verification
  - Theorem proving
  - Model checking

Formal Verification: Theorem Proving

- Develop logical/mathematical equations that describe:
  - System to be verified
  - Specification of correctness for the system
- In the rules of this logic/mathematics, prove that the system is equivalent to its specification
- Several mathematical languages have been developed for this purpose
- Software has been developed to perform theorem proving (or to assist it, with some amount of hand-holding)
- Theorem proving is often intractable for very large complex systems, but can work on small sub-systems

Formal Verification: Model Checking

- Describe system as finite state machine (FSM)
- Develop logical/mathematical equations that describe required properties of the FSM
- Example properties:
  - Never ends up in state X
  - Can reach every desired state in FSM
- Software has been developed to perform model checking → logically, this is an exhaustive search
  - Example: Murφ model checker (from Stanford)
- Similar to theorem proving, model checking is intractable for large complicated systems
  - Algorithms tend to be exponential in number of states
  - Some optimizations possible (e.g., exploiting symmetry in FSM)

Software Tools for Static Analysis

- There are tools that can analyze software to determine if it has bugs
- Can check to see if:
  - All code is reachable
  - Deadlock/livelock is possible
  - Etc.
- Advantage of static analysis tools
  - Checks all possible control flow paths through application → can detect any possible specified problem, even if it would only occur very rarely in practice
- Disadvantages
  - Must have access to entire code base, e.g., can't deal with dynamically loaded libraries
  - Difficult to assess probability of error occurring in practice

Static Analysis with RacerX

- "RacerX: Effective, Static Detection of Race Conditions and Deadlock" by Engler and Ashcraft, SOSP 2003
- Analyzes all possible control flow paths in a multithreaded application to determine if livelock is possible
  - Ranks all possible livelocks by predicted likelihood of occurrence
- Requires some input from the programmer – must annotate lock_acquire() and lock_release()
- More efficient than model checking techniques for detecting if livelock is possible

Fault Masking with N-Version Programming

- Specify what the code should do, then implement it in N different ways (with N different teams), run each of the N versions, and compare the results
  - Goal: avoid bugs due to a bad implementation
- Heterogeneous redundancy
  - TMR is homogeneous redundancy – why would TMR not work here?
- Doesn't handle bad specification (remember GIGO)
- Challenges
  - Cost!! Software development is expensive
  - If N=2, how do we know which is correct?

Software Rejuvenation

- Recall what we learned from "Proactive Management of Software Aging" (Castelli et al.)
- Basic idea: periodically reboot system to flush out lingering latent problems due to aging
- Analogous to "memory scrubbing" for DRAM
- Key idea: if we allow latent errors to remain, then we're more likely to fail when another error occurs
- (Classifying this as "static" vs. "dynamic" is unclear)

Outline

- Software Faults, Errors, and Failures
- Static Methods for Finding/Avoiding Software Faults
- Dynamic Software Error Detection
- Fault/Failure Isolation
- Interactions Between SW and HW Fault Tolerance

Dynamic Error Detection

- Must add code to check software as it is running
  - Unless you're willing to wait for it to crash or hang
- Added code = redundancy!
- Most common form of error detection: assertions
  - E.g., `assert (GPA >= 0 && GPA <= 4)`
- Challenges
  - Knowing which invariants to check
  - Knowing when to check these invariants
  - Dealing with black box code (e.g., libraries)

Dynamic Error Detection with Meta-Compilation

- Recent research from Berkeley explores how to have the compiler automatically integrate error checking to code
- User can specify general high-level invariants
  - E.g., every `lock_acquire()` must have corresponding `lock_release()`
- Compiler automatically integrates invariant checking into the code

Automatic Dynamic Error Detection

- "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code" (Engler et al., SOSP 2001)
- Compiler automatically adds error checking to code
  - Compiler can infer "beliefs" about the system
- Example #1: 99% of lock_acquire() calls have associated lock_release()
  - Then that other 1% is probably wrong
- Example #2:

```
if (ptr == NULL){
   printf(%d, ptr→data)   // what's wrong here?
}
```


Other Forms of Dynamic Sanity Checking

- Java has automatic array bounds checking, and it won't let you write beyond the bounds of the array
- Operating system will not let an application process access memory that doesn't belong to it. This is what is happening when you see "segmentation fault"!
- FTP software uses a checksum to make sure that the data that was received is the same as the data that was sent
- Other examples?

Self-Checking Code

- Can we write software that checks that its output is correct?
- Example: if we divide A/B = C, we can check the result by multiplying B*C. If B*C != A, then the division was incorrect.
  - Detects hardware faults (famous Pentium FDIV bug)
  - Detects software faults (assuming more complicated operation than just division, which is a single instruction)
- Can you think of more useful examples?
- Key: checking a computation is always at least as easy as performing it (result from computational complexity theory)
- See me for a good reference paper on this idea

Dynamically Checking Liveness

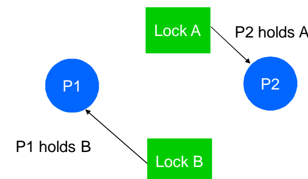
- Assertions may not help you determine that your program isn't making forward progress
- Similar to the case with hardware, we can use watchdogs to check for progress
- Example: the simulator code that I use is checked every night by a software program called a regression tester
  - But what if the regression tester fails to run? Who checks it?
  - One option is a watchdog program that runs some time later and checks to see if the regression tester wrote its output. If not, it emails me to detect this lack of liveness.

More Dynamic Liveness Checking

- There are many software techniques for trying to detect livelock
- Example: Database management systems (DBMSs) detect livelock due to either a fault or overly optimistic concurrency
- But how does this work … ?

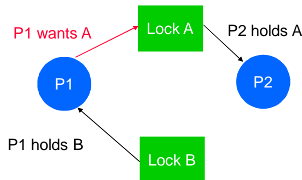
Even More Dynamic Liveness Checking

- In general, we dynamically construct a resource dependence graph (e.g., for locks) and try to discover cycles in this graph
  - Some performance penalty and storage overhead usually incurred to construct and analyze these graphs



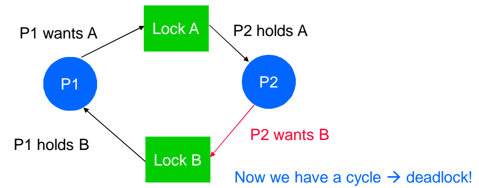
Even More Dynamic Liveness Checking

- In general, we dynamically construct a resource dependence graph (e.g., for locks) and try to discover cycles in this graph
  - Some performance penalty and storage overhead usually incurred to construct and analyze these graphs



Even More Dynamic Liveness Checking

- In general, we dynamically construct a resource dependence graph (e.g., for locks) and try to discover cycles in this graph
  - Some performance penalty and storage overhead usually incurred to construct and analyze these graphs



Yet More Dynamic Liveness Checking

- Recent work at Duke (by Tong Li) efficiently detects if a multithreaded application is livelocked
- Basic idea (with complexity omitted):
  - Detect if threads are spinning/blocked
  - If all spinning/blocked, then trap to OS and invoke livelock detection mechanisms (by being on-demand, no common-case penalty)
  - OS speculatively un-spins/un-blocks a thread and lets it execute speculatively (in a "sandbox")
  - If speculative thread performs a store that would un-spin/un-block another thread, then speculatively wake it up (repeat this process)
  - This way we can dynamically construct a dependence graph among threads
  - If a loop in dependence graph, then we've detected livelock

Dynamic Checking for Subtler Bugs

- What about the bugs that we classified under "software aging"?
  - E.g., memory leaks
- There are software tools that can help uncover these subtle types of bugs, but they all slow down performance by a LOT
  - Valgrind (open source, multi-platform)
  - Purify (only for SPARC)

Outline

- Software Faults, Errors, and Failures
- Static Methods for Finding/Avoiding Software Faults
- Dynamic Software Error Detection
- Fault/Failure Isolation
- Interactions Between SW and HW Fault Tolerance

Fault/Failure Isolation

- As with hardware, we'd like to be able to isolate a fault/failure and prevent it from affecting the rest of the system
- Some of the tricks we used for hardware fault isolation involved using software
  - Logical partitioning
  - Virtual machines
- Can these schemes tolerate software bugs?
- Other options for software fault/failure isolation
  - Clustering (Google, Amazon.com)
  - Modular OS (Hive)

Fault/Failure Isolation with Clustering

- "Web Search for a Planet: The Google Cluster Architecture" (Barroso et al.)
- Note: this paper is obviously about much more than just fault/failure isolation!

Fault/Failure Isolation by the OS

- "Hive: Fault Containment for Shared-memory Multiprocessors" (Chapin et al.)
- We will NOT cover this paper in this class, but you may want to read it if you're interested in OS issues
  - ECE 254 doesn't have OS as a prerequisite

Outline

- Software Faults, Errors, and Failures
- Static Methods for Finding/Avoiding Software Faults
- Dynamic Software Error Detection
- Fault/Failure Isolation
- Interactions Between SW and HW Fault Tolerance

Hardware for Software FT

- Difficult for HW to know that SW is in error, because HW doesn't know what SW is trying to do
- Some sanity checks possible
  - E.g., it's unlikely that a program really wants to divide by zero
  - Any others?
- Recent work at Duke explored hardware support for detecting livelock and starvation
  - Use hardware to detect if a thread is spinning (i.e., not making forward progress)
  - Detects spinning by comparing state of the system each time thread reaches backward branch
  - If all threads are spinning at same time, this may be livelock
  - If one or more threads are spinning for a very long time, this might be mutual livelock or starvation

More Hardware for Software FT

- "A 'Flight Data Recorder' for Enabling Full-System Multiprocessor Deterministic Replay" (Xu, Bodik, and Hill, ISCA 2003)
- A "flight data recorder" (FDR) helps software debuggers by allowing them to replay the last few moments before the crash
- Keys: replay is deterministic!
  - This is non-trivial for a multiprocessor system
  - FDR must keep track of certain event orders

Software for Hardware FT

- Many examples of using software to tolerate HW faults, e.g., SWIFT (soon!), Google
- In fact, all schemes for tolerating software errors will detect hardware errors that manifest themselves in the same way (i.e., they have the same error model)
  - E.g., self-checking software will detect a hardware fault if it leads to an incorrect result

More Software for Hardware FT

- Why not just use software for FT?
- Why bother with hardware FT?
- Doesn't the "End-to-End" argument suggest that hardware FT is too low-level?

SW Detection of HW Faults

- "SWIFT: Software Implemented Fault Tolerance" (Reis et al.)
- "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design" (Li et al.)

SW Repair of Hard HW Faults

- "Detouring: Translating Software to Circumvent Hard Faults in Simple Cores." (Meixner and Sorin)

Outline

- Software Faults, Errors, and Failures
- Static Methods for Finding/Avoiding Software Faults
- Dynamic Software Error Detection
- Fault/Failure Isolation
- Interactions Between SW and HW Fault Tolerance