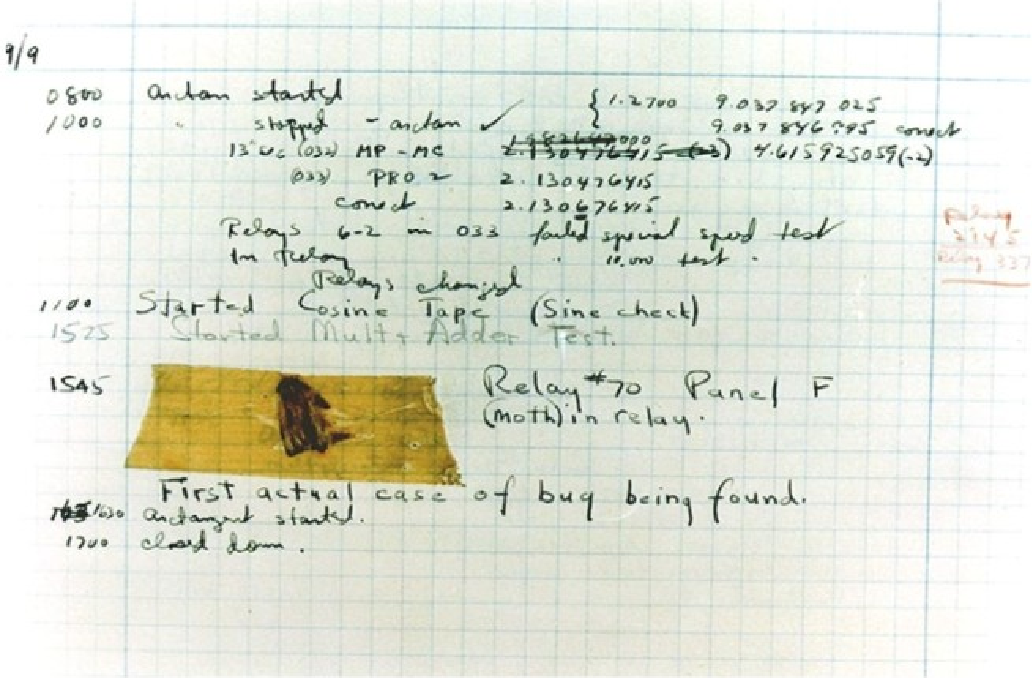
Debugging

NTI/PRK 2025, LenkaKT TUL

The very first bug

1942 a moth in a relay in

analog computer Mark II



Error types

- **Syntactical** - no debugging needed, syntax analysis identifies it.
- **Logical** - we have misbehaving, but running software.
  - **Typos and writing problems** reverted conditions - an typo error during writing, i. e. in conditionals
  - **Wrong input data**
  - **Underestimated limit conditions** (fence post error, wrong treating non-existent input etc.)
  - **Wrong memory management**
  - **Bad algorithm design**
  - **Concurency issues** (race condition, deadlocks etc.)

How errors are detected at a first glance

1. Program stops and raises error message.
2. Program stops and the OS raises error message.
3. Program stops with no significant output.
4. Program misbehaves.
5. Error is not detected when occurs, but later due to consequences (i. e. race condition).

What to do if we catch the error/misbehaving code

1. Reproduce the conditions and repeat the error.
2. Find the bug.
3. Determine the root cause.
4. Fix the bug.
5. Test to validate the fix.
6. Document the process.

Skipping one of those steps may lead to insufficient bug fix.

Reproducing the conditions

- Diagnose carefully the original computer and its environment (system logs).
- Repeat the process in identical OS, environment, application setup etc.
- Use the identical input data.
- Repeat the way of using the program.

- Companies keep installations of typical user-setups for debugging.
- If errors are reported via UI - take care to collect all the important information from your users.
- Create application logs on operation and ask users for logs.

Finding the bug

- We need to identify what code step raises the error.
- Time for debugging tools helping to watch the code internals and debug step by step.
- Take care for concurrency issues - always keep in mind. You cannot find it by steping/running in debugger mosty.
- Smart debugging with tools speeds up this part dramatically.
- Scripting the debugger, analyzing logs, AI is helpful here.

The root cause vs the bug itself

- The bug: the moth in a relay. The cause: insufficient insect protection.
- Identifying bug: i. e. some variable contains wrong data.
- Identifying the root cause: we understand why we have wrong data here.
- You need to analyze:
    - the code flow
    - the data flow
    - the algorithm itself
    - all the parts of the code (libraries, includes etc.)
- Any code documentation (architecture, blocks, class diagrams etc.) is extremely helpful.
- Visualisation (flows) is helpful.

You must not ignore even a “non significant bug” - it may result of serious hidden cause.

In finances - if your final sum is not correct even at 0.001, it means it is not correct and **you must hunt the bug!**

You never know what is the root cause and how incorrect it will be with different inputs!

Testing the fix

Incremental work!!!

1. **Unit tests** test the individual code segment changed to fix the bug.
2. **Integration tests** test the unit and the rest of the code all together.
3. **System tests** test the whole system.
4. **Regression tests** ensure that that the fix has no impact at application performance.

Debugging techniques - Logging

- **Log** - a record stored in the computer; contains the applications relevant system outputs.
- Logging showcases events related to changes in a system's state.
- Records of events with a timestamp and contextual payload.
- Log is designed by the app author; you have no guarantee that the bug appears here.
- Applications typically support several levels of logging (and it can be modified in app settings or input configuration such as registry).
- Non intrusive and limited observation; different results for different apps.
- !!! Concurrency issues hard to identify - timing changes.

Debugging techniques - Tracing

- Tracing records a request or transaction as it travels across system components.
- Each trace describes communication between components, indicating who communicated and what was communicated.
- It helps to analyze the data and code flow.
- You need to have codes and external tracing tool (such as strace in Linux).
- Non intrusive and limited observation; same results for different apps.
- AI based tools emerging.
- !!! Concurrency issues hard to identify - timing changes.

Tracing 2 - Main tracing techniques

- **Program tracing:** Used to analyze the addresses of infrastructures and variables signaled by an active application and to diagnose issues like memory flow and excessive resource consumption.
- **Code tracing**: A process that inspects the flow of source codes in an application when performing a specific function.
- **End-to-end tracing**: Tracks data transformation together with the service request path. When an application commences a request, it sends data to other software components for further processing.

Tracing granularity and levels

- Granularity: high level events vs each function call; time steps size
- Levels:
  - Which function entry and exit
  - The function’s duration
  - Parameters passed
  - Variable values
  - Timestamps
  - Memory usage

Tracing tools in Linux userspace

- **strace** - traces the syscalls in userspace, it can reveal how the app interacts with the kernel, including file I/O, network operations, and process management. Useful in concurrency issues.
- **ltrace** - traces the library calls in userspace, it can help to analyze the environments and problems raised due to different versions API and similar.
- **dtrace -** a complex tracer used to trace the whole system, originally designed in Sun Microsystems, widely used in Linux, ported to Windows
- **eBPF based tracers:** C programs written for eBPF, executed within a kernel-space tracing selected applications

Probing

- When tracing, you need “probes” detecting the data flow.
- You do not want to modify the code itself (it changes the behavior and you can “lost” the error).
- I. e. in Linux the kernel must support user-probes (kernel option).
- Linux kernel has robust subsystem for probing in drivers and probing the kernel itself.
- **uprobes -** user-space probing in Linux, **kprobes** kernel-space probing

Backtracking

- Examining the sequence of program execution that led to an error.
- Involves stepping "backward" through the call stack to understand the path taken.
- Most developers do “intuitive way”.
- How to do it (often with debuggers):

How to do it:

- Inspect the call stack to see the history of function calls (tracing useful to get the call stacks).
- Examine variable values at each step of the call stack.
- Potentially "step out" of the current function to the caller.
- Debuggers useful in this step.

Cause elimination

- You already have a hypothesis where is the problem and change the code accordingly and then try to reproduce the bug.
- You are just trying and speculating.
- Widely used, but tracing/backtracking and other systematic approaches are more reliable.
- To do so, you need the code and you need to understand it.

Divide and conquer

- The complex application is splitted into parts and analyzed independently.
- Dividing into: functions, modules, class methods or other testable logical divisions.
- The goal is to identify the problematic segment and then in next debugging process to find the bug and root cause.

Automated debugging

- AI/machine learning based algorithms can help analyze the code.
- !!! Take care - it speeds up the process, but it may skip some error or mislead you - you still need a technical experience and some kind of senior knowledge.
- Mostly based on running the automated tests.
- Involved in CI/CD processes.
- Test suite must be designed before it starts, sometimes AI helps/designs the tests as well.

Brute force debugging

- When all the analysis and tests fails:
- Step by step, line by line through the whole codebase from beginning to the end.
- Expensive and exhausting; we try to avoid this phase.

Race condition debugging

- Race condition is tricky and hard to find.
- A tool helping to analyze running threads is useful (JBrains, IntelliJ have views to threads)
- You need to analyze what threads run in parallel.
- Then you need to try to identify the critical sequences in the code accessing the critical (shared data).
- Focus on global and shared variables.
- Traces and probes are useful.
- Breakpoints when threads enter the critical sections.

Parallel to debugging - profiling

- Analyzing a program's execution to understand its performance.
- **The goal:** to identify performance bottlenecks and areas where the code can be optimized to run faster and more efficiently.
- It involves collecting data on various aspects of the program's runtime behavior, such as:
  - Execution time: How long each part of the code takes to run.
  - CPU usage: How much processing power different parts of the code consume.
  - Memory usage: How the program allocates and manages memory.
  - Function call counts: How many times each function is called.