Monitoring, Tracing, Debugging (Under Construction)

I was already tempted to drop this topic from my lecture on operating systems when I found Stephan Siemen's article "Top Speed" in Linux World 10/2003. He mentions a whole range of useful tools for performance tracking and bug finding on Linux and I decided to take his list and give some additional background on how these things work.After being a unix kernel hacker for some year I moved into user space and started to develop more and more user level code. I soon realized that the style of development used for kernel hacking is quite different to the one needed for servers, frameworks and applications. Some concepts still apply, like logging events. Others change, e.g. using garbage collectors. Some of the tools that made the biggest impression where Pure Atria's Purify (now IBM Rational) - object code instrumentation the easy way. And the Visual Age (now Eclipse) Debugger which made me give up on printf/println style of finding bugs. Look at the resource section at the end for links.

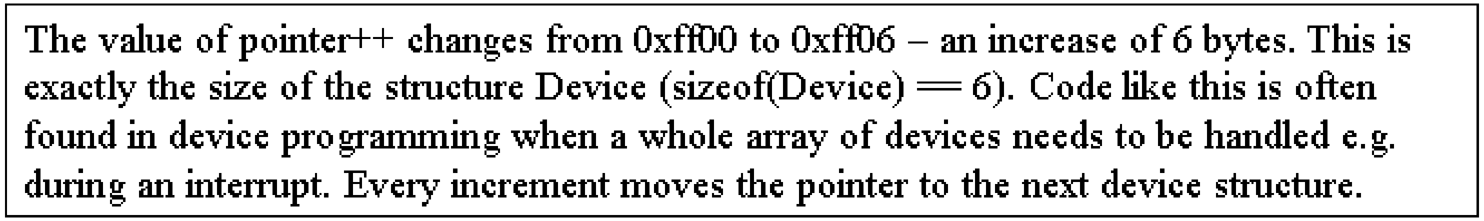
Introduction

Goals

1. A word on monitoring, tracing, debugging.
2. Modern approaches for tracing, logging and auditing.
3. Understanding interfaces within and between programs and environments. Remember: every problem in computer science is solved with one more level of indirection. And interfaces are the locations where program flow can be re-directed.
4. Monitoring Levels
5. Different techniques for monitoring
6. Performance and memory tracking. How garbage collection works will be explained in detail in our session on virtual memory. Here we will do wiht the basic concept of memory colloring to find wrong reads/writes.
7. There are still developers out there using printf/println for DEBUGGING or LOGGING. After these slides I don't want to see this practice EVER again in your work!. The basics of debugging (breakpoints, stack, threads).

What monitoring, logging and debugging really means

Overview


The value of pointer++ changes from 0xff00 to 0xff06 – an increase of 6 bytes. This is exactly the size of the structure Device (sizeof(Device) == 6). Code like this is often found in device programming when a whole array of devices needs to be handled e.g. during an interrupt. Every increment moves the pointer to the next device structure.

Overview (Continued)

The basic pattern behind monitoring, logging and

The interceptor pattern underlies all these mechanisms. To get information about a certain control flow it needs to be intercepted and written into a log somehow. Depending on where the program flow is intercepted we know different levels of monitoring.

Monitoring Levels

Kernel	**Kernel monitoring means tracking the performance or behavior of a whole system. It is only possible with the help of the kernel itself, either through device driver modules or through kernel functionality directly. Control and data are usually maintained in userspace.**
Process Level	**How does the process behave with respect to resource consumption? How much time is spent in user vs. system mode? How much memory is allocated? Garbage Collectors provide monitoring interfaces which allow the tracking of memory usage. Utilities like ps, top, taskmanager show process activity.**
System Call Level	**Applications (and user level servers) finally need to call the operating system for critical functions like I/O. This is done through kernel traps - an ideal place to watch what a program does without further interfering with it (or having to modify it)**
Library Level (static)	**Many applications use additional libraries or organize their own code into libs. If those libraries are linked statically into the program they become a fixed part of the application. Sometimes special versions of those libraries exist (with additional functions for tracking problems). The applications gets rebuilt wiht such a library and runs now in a "debugging" mode.**

Monitoring Levels (Continued)

Library Level (dynamic, dll)	**Libraries can be loaded at runtime. In this case it is easy to replace the lib with a dummy or debug version or just intercept the call from the application to the library function, do some monitoring and then forward the call to the real lib.**
Method or Function call level (with or without VM)	**Most programming languages offer no easy way to intercept program-internal calls to its own methods or functions. The compiler needs to help here. Driven by compile time arguments the compiler inserts extra code between function calls which performs monitoring functions. Applications need to be recompiled of course. It is easier if the program runs under the control of a virtual machine because the VM usually has a monitoring interface which allows the user to switch into special monitoring or profiling modes. At this level people start talking about "profiling" the applications control flow via the "call graph" (the order of funtion or method calls). Performance optimizations usually require this level of analysis.**
Intra-function or intra-method level	**Due to the lack of interfaces this level is tracked by inserting debugging code (println/printf) into the program. Requires code changes of course and needs to be removed after the problems are found. An alternative is the use of a debugger which frequently also requires a recompilation with special debug**

Monitoring Levels (Continued)

	arguments.
Intra-expression level	Only debuggers can reach this level. With virtual machines one could possibly also single step through the bytecode.

Memory Tracking

Memory Access Errors

A dynamically allocated piece of memory can be accessed in several illegal ways.

1. **Read before write: This is a read of uninitialized data and can lead to random results. Repeated runs of the program cause different results depending on which memory chunk is allocated.**
2. **Write before or after allocation: This happens when a process writes beyond borders, e.g. makes mistakes in array arithmetics. Or the memory has been given back to the system (via free) and the process still writes to it.**

Memory tracking techniques

Two very powerful ones are the placement of control areas e.g. after arrays by writing a certain unique pattern after the end of the array. If the pattern is destroyed then we know that the process writes out of bounds. Memory coloring tracks the state of each memory chunk as "uninitialized, initialized or freed" and logs violations in the order of access. (Purify)

Debugging Technology

How do breakpoints work?

Control of the child process in systems with virtual

Remote Debugging

Linux Tools and Utilities for Monitoring

System Call Tracking

STrace	**Use STrace to track all activity between a user process and the kernel. No application change necessary. Good if you want to know about the major external functions that your program calls.** **The utility probably needs to find all kernel traps and route those to itself**
LTrace	**Same as STrace just for dynamic link libraries (shared libraries)**
ldd	**All dynamic mechanisms have the problem that one has to know exactly WHICH library will be loaded, WHICH class loaded. This utility does it and there is one also for the Windows world but I forgot it, sorry. BTW: the unix shell also has a dynamic load mechanism and that is what the WHICH utility is for: it tells you which utility is loaded from which path. There might be duplicates in different places and the PATH variable will decide which one gets loaded.**

Method or function profiling

GProf	Use GProf to create call graph information for an application. Take the functions which use the most processing time and start optimizing those. The tool writes profiling output to a file. The application needs to be compiled with special profiling options.
KProf	Nicer GUI for GProf output.
Function Check	Generates profiling information but needs an extra library.
GCov	Also a profiler. Writes results right into source code of program.

Monitoring Memory

Valgrind	Processor simulator which tracks all memory accesses by a process. Use -g -O0 to prevent optimization by the compiler.
Libmalloc	Instrumented versions of malloc/free exist which can be linked into a program as a replacement for the original versions. The debugging versions allow fine-grained tracking of memory allocation errors
Memprof	No program changes are necessary. This tool just tracks which function needs how much memory. It also tracks unused memory areas which have been allocated but are no longer reachable.

System Monitoring

OProfile

Collects profile data about all processes. Needs to load a kernel module dynamically. No program changes or recompilations are needed. Typical design with kernel part to collect data and postprocessing in user space.

Resource Tracking Utilities

Try "PS" and "TOP" to see how many resources are allocated by your process.

Virtual Machines

Java Profiling

13. Debug Tracer -------------------------------------- Categories: Product , JPDA by Olivier Dedieu - rating: 1/5 Debug Tracer is an XML-based scripting tool for debugging, tracing, and monitoring the JavaTM Virtual Machine (JVM). It is useful for debugging problems quickly, almost "real-time," when a number of problems manifest themselves. The tool does not require any modification to code and can be used to monitor not only the... http://www.java-channel.org/display.jsp?id=c_16823

Eclipse plug-ins for memory and call-graph profiling

Exercises

Java Based

1. Check the JVM command line arguments for monitoring memory or generating a profile.
2. Take a small Java program, create a profile and inspect it with the java profiler
3. Download an Eclipse profiling plug-in (sourceforge.com or eclipse.org) and generate a detailed call-graph and memory usage diagram. Which method takes most of the time? Which one uses the most memory?
4. Get your favorite Java debugger and try setting breakpoints. Can you find the calling hierarchy (stack)? How many threads do you have? Monitor the value of a variable in a for or while loop. Dump the values of local class members of a class.
5. Dump the bytecode of your java program using jprof.
6. Use a decompiler (DJ) to recreate the java code from your program.

Unix/Linux Process related Exercises

1.