Kernel Debugger

Seminar

Martin Mehlmann
mehlmann@st.cs.uni-sb.de

Übersicht

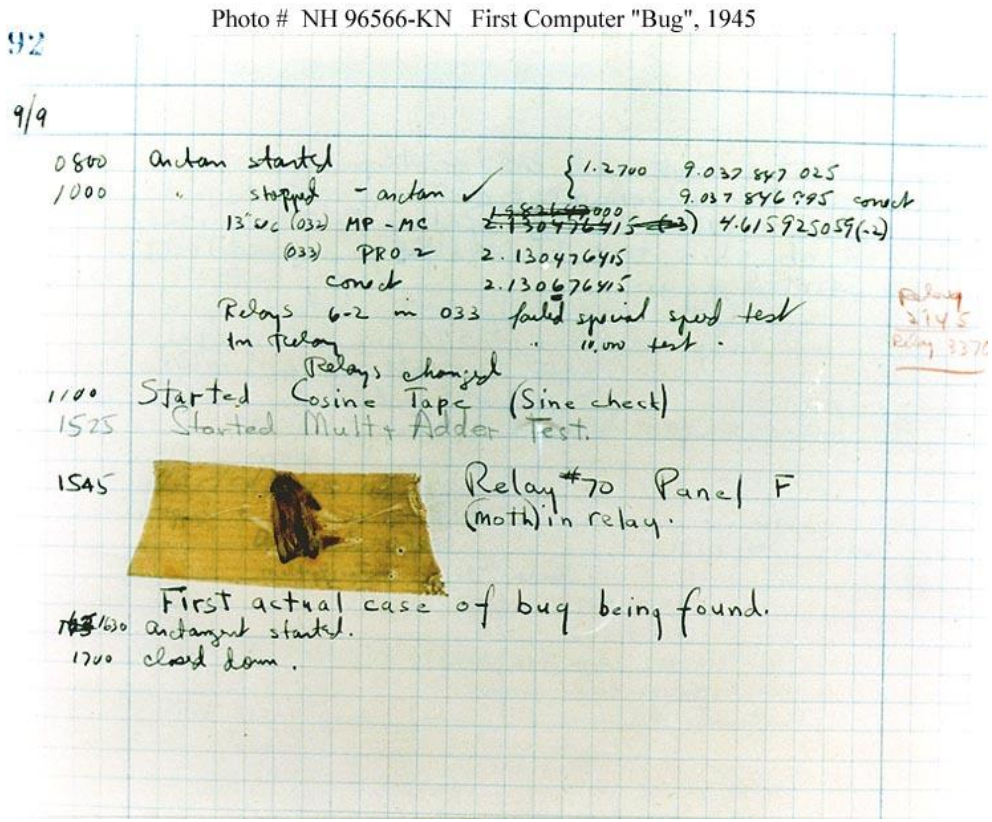
- Bug
- Debugging
- Debugger und Werkzeuge

- Kernel Debugging
- Kernel Debugger und Kernel Werkzeuge

Begriffsklärung Bug

- Woher kommt der Begriff Bug ?
  - Der erste Bug
    - 9. September 1945
    - Grace Hopper fand bei Fehlersuche in Mark 2 eine Motte, deren Flügel das Einlesen der Lochkarte blockierten
  - Resultierende Begriffe
    - Bug für Fehler
    - Debugging für Fehlersuche und Fehlerbehebung

Begriffsklärung Bug



Begriffsklärung Bug

- Fehler in Software
  - Fehler im Quellcode
  - Fehler im Programmzustand
  - Fehler im Design
- Fehler in Hardware
  - physikalischer Fehler
  - Fehler im Design
- oder doch ein Feature ?

Begriffsklärung Bug - Fehlschlagen eines Programms

- Der Programmierer hinterlässt einen **Defekt** im Quellcode
  - Fehler im Quellcode
- Wenn dieser defekte Code ausgeführt wird, so ist das Resultat eine **Infektion** des Programmzustandes
  - Fehler im Programmzustand
- Die Infektion propagiert sich eventuell über mehrere Variablen
- Die **Infektion** resultiert in einem **Fehlschlagen** des Programms, welches vom Benutzer beobachtet wird
  - Fehler im Programm

Begriffsklärung Bug - Der Fluch des Testens

- Nicht jeder **Defekt** im Programmzustand muss in einem **Fehlschlagen** resultieren
  - Nicht jeder **Defekt** resultiert in einer **Infektion**
  - Nicht jede **Infektion** resultiert in einem **Fehlschlagen**
- Testen kann nur die Anwesenheit von Fehlern zeigen, nicht deren Abwesenheit (Dijkstra 1972)
- Nur Verifikation kann die Abwesenheit von **Defekten** garantieren

Begriffsklärung Debugging

- Jedes **Fehlschlagen** kann auf eine **Infektion** im Programmzustand zurückgeführt werden
- Jede **Infektion** kann auf einen **Defekt** zurückgeführt werden
  - Ausnahme: physikalische Einflüsse
- Debuggen bedeutet ein beobachtetes **Fehlschlagen** einem **Defekt** zuzuordnen und diesen zu entfernen

Begriffsklärung Debugging

- Debugging ist ein zweidimensionales Suchproblem
  - Zeit
    - Bestimmung des Infektionsbeginns
    - Zustandsübergang von gesund nach infiziert
  - Variablen
    - Aufteilung in gesunde und infizierte Variablen

Debugging - Vorgehensweise

- Beobachte das Fehlschlagen des Programms
- Reproduziere das Fehlschlagen unter denselben Umständen
  - Automatisierbar (*capture and replay*)
- Vereinfache die Umstände
  - Automatisierbar (Deltadebugging)
- Lokalisiere den Defekt
  - Automatisierbar (Deltadebugging)
- Beseitige den Defekt
  - *most general fix*

Debugging - Unterscheidung

- Statisches Debuggen
  - deduktiv
  - Debuggen ohne Programm auszuführen
- Dynamisches Debuggen
  - induktiv
  - Debuggen zur Laufzeit
    - Beobachten von einzelnen Programmläufen
  - Debuggen nach dem Fehlschlagen
    - Post Mortem Debuggen

Begriffsklärung Debugger

- Software, welche Debuggen anderer Software ermöglicht
  - Kontrollierte Ausführung
  - Überwachung
  - Manipulation
- Unterscheidung
  - Debugger auf Maschinenebene
  - Debugger auf Quellsprachenebene
    - GNU Debugger GDB

Debugger - Maschinensprachenebene

- Verfolgung des Programmgeschehens auf Maschinensprachenebene
- Vorteil
  - Untersuchung von *closed software* möglich
- Nachteil
  - *low level* Assemblercode
  - Kenntnis der Rechnerarchitektur nötig

Debugger - Quellsprachenebene

- Verfolgung des Programmgeschehens auf Quellsprachenebene
- Vorteile
  - Ausführen oder Überspringen von Funktionen
  - Auswerten von Ausdrücken in Quellsprache
- Nachteile
  - Benötigt Debuggingsymbole
    - Symboltabelle
      - Variablennamen, Funktionsnamen
      - Matching zwischen Quellcode und Assemblercode
  - Ausführbare Datei kann erheblich größer sein
  - Quellcode muss vorliegen

Debugger - Grundprinzipien

- Das Heisenberg-Prinzip
  - Die Verwendung eines Debuggers darf das Verhalten des untersuchten Programms nicht beeinflussen
    - Das untersuchte Programm ist auf einen Teil des Speichers angewiesen, der aber auch vom Debugger verwendet wird
- Das Wahrheitsprinzip
  - Die Informationen, die der Debugger liefert, müssen wahrheitsgemäß ein
    - Eine Variable wurde in ein Register geladen und verändert, der Debugger aber holt den Wert aus dem Speicher

Debugger Funktionalität - Anzeigen des Programmzustands

- Suche in der Zeit
  - Haltepunkte
    - *break*
    - *continue*
  - Einzelschrittausführung
    - *step*
    - *next*
  - Programmzähler

Debugger Funktionalität - Anzeigen des Programmzustands

- Suche in den Variablen
  - Werte von Variablen
    - *print*
    - *info locals*
  - Funktionsstapel
    - *backtrace*
  - Registerinhalte
    - *info registers*

Debugger Funktionalität - Anzeigen des Programmzustands

- Suche in Zeit und Variablen
  - Datenhaltepunkte
    - *watch*
  - Bedingte Haltepunkte
    - *break if*

Debugger Funktionalität - Ändern des Programmzustands

- Werte von Variablen
  - *set*
- Funktionsstapel
  - *up*
  - *down*
- Programmzähler
  - *set*
- Registerinhalte
  - *set*

Debugger - Funktionsweise

- Betriebsystem führt Programme als Prozesse aus
- Debugger und untersuchtes Programm sind separate Benutzerprozesse
- Kommunikation erfolgt über vom Kernel zur Verfügung gestellte Debugschnittstelle zur Kontrolle der Ausführung von Prozessen durch andere Prozesse
  - *long ptrace(enum request, pid_t pid, void* addr, void* data);*
- Jede Interaktion zwischen Debugger und untersuchtem Programm läuft über diese Schnittstelle

Debugger - Funktionsweise

- Vorbereitung des untersuchten Prozesses
  - *PTRACE_TRACEME*
- Zugriff auf Register des untersuchten Prozesses
  - *PTRACE_GETREGS*
  - *PTRACE_SETREGS*
- Zugriff auf Speicher des untersuchten Prozesses
  - *PTRACE_PEEKTEXT, PTRACE_PEEKDATA*
  - *PTRACE_POKETEXT, PTRACE_POKEDATA*
- Fortsetzen und Einzelschrittausführung des untersuchten Prozesses
  - *PTRACE_SYSCALL*
  - *PTRACE_CONT*
  - *PTRACE_SINGLESTEP*
- Beenden des untersuchten Prozesses
  - *PTRACE_KILL*

Debugger - Funktionsweise

- Debugger (Vaterprozess)
  - *fork()*
  - *waitpid(child_pid, &ret_val, 0))*
    - Debugger wartet auf *SIGCHLD* Signal vom Kernel, dass Kindprozess gestoppt oder beendet wurde
    - Rückgabewert enthält Signal aufgrund dessen der Kindprozess gestoppt wurde
  - Verwendung von *ptrace* um den Kindprozess zu kontrollieren
  - Wiedereintritt in *waitpid*

Debugger - Funktionsweise

- Kindprozess
  - *ptrace (PTRACE_TRACEME, 0, 0, 0)*
    - Anmelden beim Kernel für Tracing
      - Spezielle Behandlung von Signalen
      - Prozess-Kontroll-Mechanismen
    - Erlaubt Vaterprozess Kontrolle über Kindprozess
  - *execvp (debuggee, args)*
    - Ersetzt sich selbst durch das zu untersuchende Programm
  - Kindprozess stoppt nach jedem empfangenen Signal
    - Wiederaufnahme bei: *kill, singlestep, continue*
    - Signal wird ignoriert
  - Vaterprozess wird vom Kernel durch *SIGCHLD* Signal informiert

Debugger - Setzen eines Haltepunkts

- Debugger möchte einen Haltepunkt an Adresse 0x109d4
- Falls Debugregister für Haltepunkte vorhanden, wird eines dieser Register verwendet (Hardwarehaltepunkt)
- Ansonsten wird die Instruktion, die unter der Adresse 0x109d4 zu finden ist gesichert und durch eine ***break instruction*** ersetzt (Softwarehaltepunkt)

```
int rc;
if (Haltepunkt-Register frei) {
    rc = ptrace (PTRACE_SETREGS, 0, 0x109d4)
} else {
    saved_instruction = ptrace (PTRACE_PEEKTEXT, 0x109d4, 0);
    rc = ptrace (PTRACE_POKETEXT, 0x109d4, break_instruction );
}
```

Debugger - Erreichen eines Haltepunkts

- Debugregister oder ***break instruction*** löst Trap aus welche vom Kernel verarbeitet wird
- Kernel sendet *SIGTRAP* Signal an Prozess und lässt diesen an Adresse 0x109d4 stoppen
- Debugger wird vom Kernel durch das *SIGCHLD* Signal informiert
- Der Debugger versichert sich, dass das untersuchte Programm aufgrund eines Haltepunktes gestoppt wurde und stellt die gesicherte Instruktion wieder her
- Nun kann der Debugger die Werte aus dem Speicher oder aus Registern holen, um den Benutzer über den Zustand des Programms zu informieren

```
int rc;
if ( !(Aufruf von Register) ) {
    rc = ptrace (PTRACE_POKETEXT, 0x109d4, saved_instruction);
}
```

Debugger - Fortsetzen des Programms

- Falls Debugregister für Haltepunkt verwendet wurde, wird das untersuchte Programm sofort fortgesetzt
- Ansonsten wird zuerst eine einzelne Anweisung ausgeführt, um anschließend die ***break instruction*** wiederherzustellen
- Danach wird das beobachtete Programm fortgesetzt

```
int rc;
if ( Unterbrechung wurde von Haltepunkt-Register ausgelöst ) {
    rc = ptrace (PTRACE_CONT, 0, 0);
} else {
    rc = ptrace (PTRACE_SINGLESTEP, 0, 0);
    rc = ptrace (PTRACE_POKETEXT, 0x109d4, break_instruction);
    rc = ptrace (PTRACE_CONT, 0, 0);
}
```

Debugger - Einzelschrittausführung

- Falls Debugregister für Einzelschrittausführung vorhanden, wird dieses Register verwendet
  - Trap flag im Debug Register wird gesetzt, welches Trap nach jeder Instruktion auslöst
  - Kernel sendet *SIGTRAP* Signal an Prozess und lässt diesen stoppen
  - Debugger wird vom Kernel durch das *SIGCHLD* Signal informiert
- Ansonsten werden Haltepunkte nach jeder Instruktion gesetzt

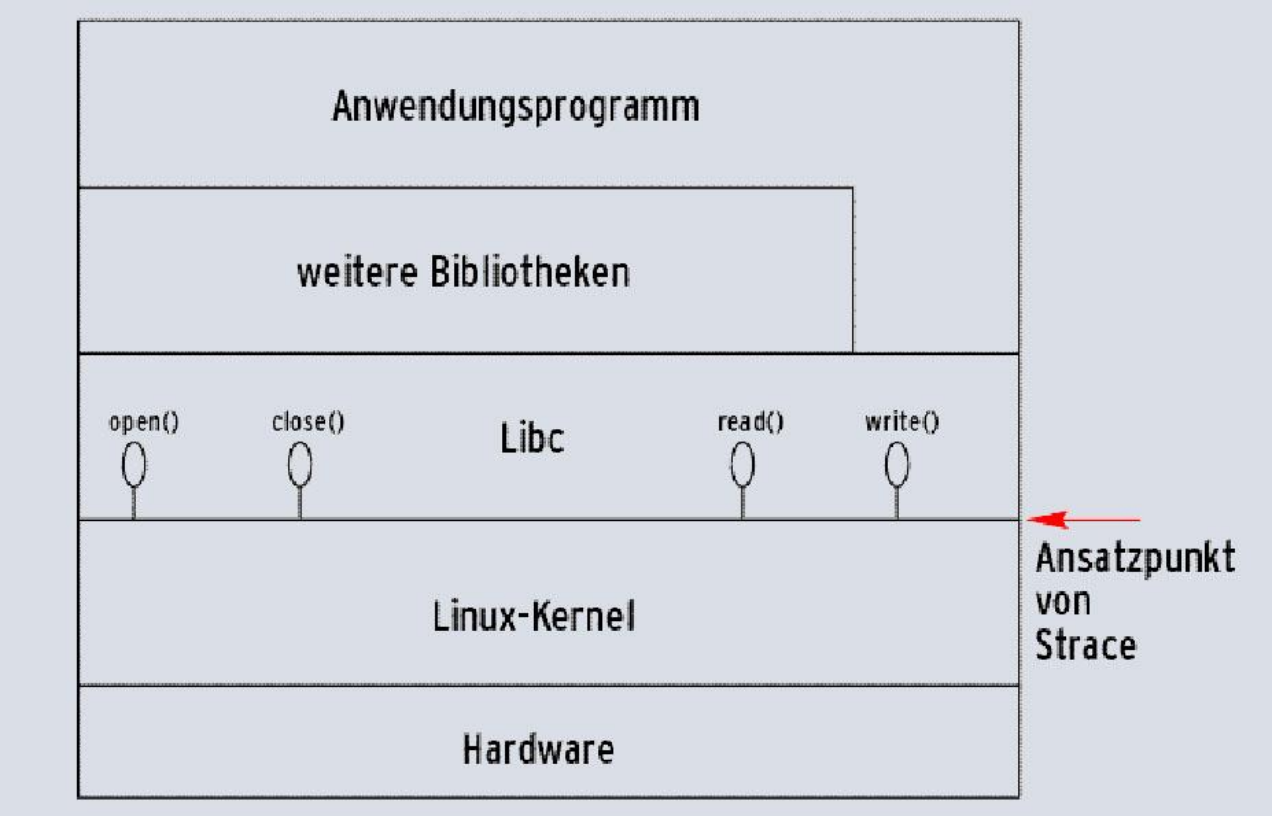
Debugger - Datenhaltepunkte

- Falls Debugregister für Datenhaltepunkte vorhanden, wird dieses Register verwendet
  - Trap flag im Debug Register wird gesetzt, welches Trap auslöst, sobald sich der Wert an der registrierten Adresse ändert
  - Kernel sendet *SIGTRAP* Signal an Prozess und lässt diesen stoppen
  - Debugger wird vom Kernel durch das *SIGCHLD* Signal informiert
- Falls keine Debugregister für Datenhaltepunkte vorhanden oder der beobachtete Ausdruck berechnet werden muss wird Einzelschrittausführung verwendet

Debugging Werkzeuge - STRACE

- Beobachten des Systemverhaltens
  - Verfolgung aller *system calls* die vom beobachteten Prozess aufgerufen werden
    - Name
    - Argumente
    - Rückgabewert
  - Protokollierung aller Signale die vom beobachteten Prozess empfangen werden
- Vorteile
  - Programm muss nicht mit Debuginformationen übersetzt werden
- Nachteile
  - Suche begrenzt auf Schnittstelle zwischen *libc* und Kernel

Debugging Werkzeuge - STRACE



Debugging Werkzeuge - STRACE

- Realisierung durch Verwendung von *ptrace*
  - *strace* weist das nach *fork()* entstandene Kind an, ihm durch *PTRACE_TRACEME* das Recht zur Kontrolle zu geben
  - Kernel sendet *SIGSTOP* Signal an Kindprozess um diesen anzuhalten
  - Kernel sendet *SIGCHLD* an Elternprozess um diesen zu benachrichtigen
  - *strace* kann nun Kindprozess mithilfe von *ptrace* kontrollieren
    - *PTRACE_SYSCALL*

Kernel Debugging - Problematik

- Keine übergeordnete Schicht, welche die Ausführung des Kernels kontrolliert
- Kernel läuft in eigenem speziell geschützten Adressraum
- Gängige Debuggerfunktionalität ist schwieriger zu erreichen
  - Haltepunkte, Datenhaltepunkte
  - Einzelschrittausführung

Kernel Debugging mit GDB

- laufender Kernel kann mit GNU Debugger GDB untersucht werden
  - Aufruf: *gdb /usr/src/vmlinux /proc/kcore*
  - */usr/src/vmlinux* ist unkomprimierte Version des Kernels mit Debugsymbolen
  - */proc/kcore* ist Name der Coredatei
    - Repräsentiert den momentan laufenden Kernel im Speicher
    - /proc/kcore wird beim Auslesen erzeugt

Kernel Debugging mit GDB

- Kernel wird nicht angehalten
  - Eingelesene Daten werden von GDB zwischengespeichert
  - Wiederholter Zugriff liefert Daten aus Zwischenspeicher
  - Aktuelle Werte durch Reinitialisierung des Zwischenspeichers
    - *core-file /proc/kcore* leert den Cache des GDB
- Nachteile
  - Keine Änderung des Kernelzustands durch den GDB
  - Keine kontrollierte Ausführung des Kernels durch den GDB
    - Keine Haltepunkte
    - Keine Datenhaltepunkte
    - Keine Einzelschrittausführung

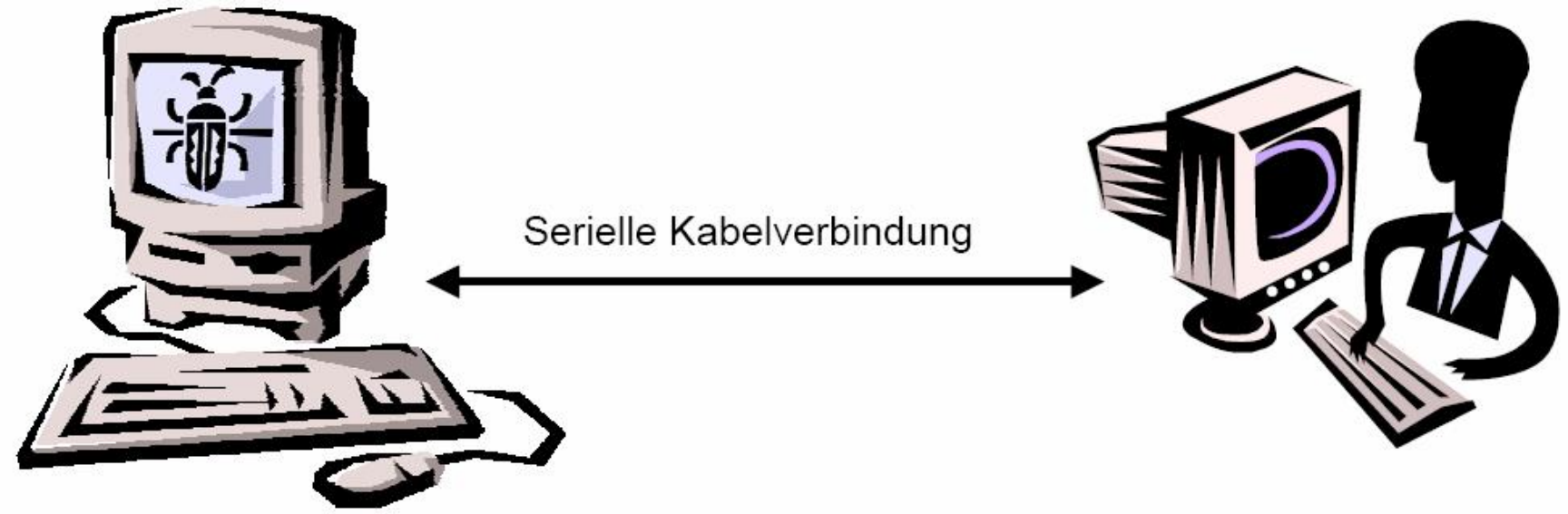
Kernel Debugging mit User Mode Linux

- Portierung des Linux Kernels als virtuelle Maschine
  - Hardwareschicht wird durch *system calls* an das Hostsystem emuliert
- Kernel läuft als separater Prozess im User Adressraum auf einem Host System
- Vorteile
  - Ein defekter Kernel kann nicht das Hostsystem zerstören
  - Verschiedene Konfigurationen können leicht auf dem gleichen Rechner getestet werden
  - Kernel kann mit GDB ohne Einschränkungen manipuliert werden

Kernel Debugger - KGDB

- Kernel Patch (http://oss.sgi.com)
  - Volle Debuggerfunktionalität
    - Haltepunkte, Datenhaltepunkte
    - Einzelschrittausführung
  - Komponenten
    - GDB stub
      - Verarbeitet ankommende Anfragen vom GDB
    - Änderungen der Fehlerbehandlung
      - Kernel gibt Kontrolle an Debugger wenn ein unerwarteter Fehler auftritt
  - Es werden 2 Rechner mit serieller Schnittstelle benötigt
    - Testmaschine
      - Enthält gepatchten zu untersuchenden Kernel
    - Entwicklungsmaschine
      - GDB kommuniziert über serielle Verbindung mit Kernel

Kernel Debugger - KGDB



Kernel Debugging - LKCA

- Linux Kernel Crash Analyzer (http://*oss.sgi.com*)
  - Kernel Oops
    - LKCD schreibt Kopie des aktuelles Systemzustand in ein Dump Gerät
    - Dump Gerät: System Swap Bereich
  - Hilfprogramm LCRASH
    - Aktiv bei nächstem Neustart des Systems
      - Erzeugt Zusammenfassung des Crash
      - Schreibt Zusammenfassung in eine konventionelle Datei
    - Interaktiv
      - Debugger ähnliche Funktionalität
  - Nachteile
    - Nur x86 32-Bit-Architektur
    - Nur Swap-Partitionen von SCSI-Festplatte

Kernel Debugging - Dynamic Probes

- Entwickelt von IBM (http://oss.software.ibm.com)
  - Einsetzen einer Sonde im System
    - Sowohl im User als im Kernel Adressraum
  - Sonde besteht aus Code
    - spezielle stackorientierte Sprache
    - Zurückliefern von Daten an den User Adressraum
    - Ändern von Registern
- Vorteile
  - Einmaliges Einbauen von DProbes in den Kernel
  - Einfügen von Sonden an beliebigen Stellen ohne Neukompilieren des Kernels oder Neustart des Systems
  - Zusammenarbeit mit LTT
    - Einfügen von Tracing Ergebnissen an bestimmten Stellen
- Nachteile
  - Nur x86 32-Bit-Architektur

Kernel Debugging - Linux Trace Toolkit

- Kernel Patch inklusive Hilfsprogramme
  - Verfolgen von Ereignissen im Kernel
    - Kernel kann Ereignisse loggen
    - Kernel Modul speichert diese in einem Puffer
    - Hilfsprogramm liest den Puffer und präsentiert Ereignisse in lesbarer Form
  - Timing Informationen
    - Was ist zu welchem Zeitpunkt passiert
    - Einkreisen von Performanzproblemen
      - Finden von Bottlenecks

Kernel Debugging - Kernel-Oops Meldungen

- Kernel-Oops Meldungen
  - Fehlermeldungen des Kernels
  - Informationen zu Zustand von Registern und Kernelstapel
  - Ausgabe hexadezimal
- klogd
  - Loggt Kernel-Oops Meldungen
- ksymoops
  - Decodiert Kernel-Oops Meldungen
  - Anzeige der decodierten Informationen

Ende

- Vielen Dank für die Aufmerksamkeit !
- Fragen ?