Instrumentierung des Kernels

Hajo Möller

Seminar “Leistungsanalyse unter Linux”

27.03.2012

Gliederung

- Einführung
  - Was ist der “Kernel”?
  - Was ist “Instrumentierung”?
- Kprobes
  - Einführung, Techniken, Beispiele
- Systemtap
  - Einführung, Architektur, Beispiele

Bedeutung

- Was ist der “Kernel”?
  - Kernel = Betriebssystemkern
  - Linux = Kernel
- Was ist “Instrumentierung”?
  - Instrumentierung = Erweitern von Quelltext zu Debugging + Analysezwecken

Kernel

- Grundkomponente des Betriebssystems
  - Hardwareabstraktion
  - Speicher- & Prozessverwaltung
  - Treiber
- Modular
  - Kernelkomponenten (Module) können bei Bedarf dazugeladen werden

Instrumentierung

- Instrumentierung im Userspace:
  - “Warum (und wo) hängt mein Programm?”
- Einfach:
  - “systematisches” hinzufügen von `printf()` im Quelltext
- Besser:
  - gdb, valgrind

Instrumentierung

- Instrumentierung im Kernel:
  - Kernel hat aktuell etwa 15.000.000 SLOC
  - “strategisches” Platzieren von printk() unmöglich
- Einfach: Teile des Kernels ändern:
  - printk() hinzufügen, kompilieren, neu booten
    → zu zeitaufwändig (~40 min)
- Besser:
  - Globale Grundlage für Instrumentierung einbauen
  - Instrumentierung per Module in den Kernel laden

Instrumentierung

- Kernel Probes (Systemtap)
  - 2.5.26 (Juli 2002) → 3.3+ (März 2012)
  - Wird zu Kernelmodul kompiliert
- Kernel Marker
  - 2.6.24 (Jan. 2008) → 2.6.32 (Dez. 2009)
  - Sollte Kprobes ersetzen bzw. ergänzen
- Tracepoints (LTTng, Systemtap)
  - 2.6.28 (Dez. 2008) → 3.3+
  - Ersetzt Kernel Marker

Kernel Probes (Kprobes)

- Ursprünglich (2000) von der IBM entwickelt
  - Grundlage (Low-Level-API) für Dprobes
  - Dprobes unpopulär, nie in-tree, letzter Commit: 2005
  - Kprobes populärer, ab 2002 in-tree
- SUN baut DTrace (2005)
  - DTrace (CDDL) nicht mit Linux (GPL) kompatibel
  - Linuxer neidisch
  - Plötzlich Interesse an freien Alternativen

Konzepte

- Eine Probe = ein Modul
  - Probe: Ereignis, auf das gewartet wird
- Probe registrieren
  - Modul initialisieren
- Probe deregistrieren
  - Modul entfernen

Kernel-Konfiguration

- Voraussetzungen in .config
  - CONFIG_KPROBES=Y

  - CONFIG_MODULES=Y
  - CONFIG_MODULE_UNLOAD=Y

  - CONFIG_KALLSYMS=Y (.text, .init.text)
  - CONFIG_KALLSYMS_ALL=Y (Rest)

Einfügen des Moduls

- Registrierung via `__init kprobe_init(void)`

```
int register_kprobe(struct kprobe *kp);
```

- Deregistrierung via `__exit kprobe_exit(void)`

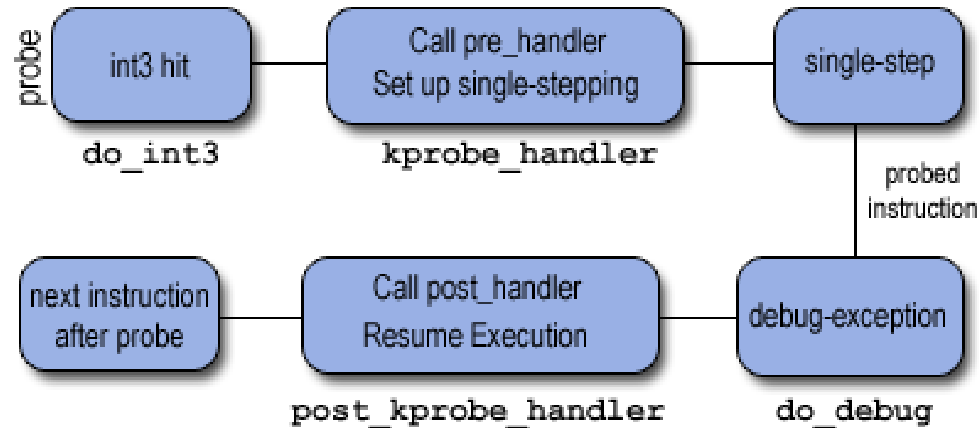
```
int unregister_kprobe(struct kprobe *kp);
```

- Rückgabewert < 0: Fehler

Grundlagen

- Breakpoint-basiert
- Kprobe
  - Kernel Probe
- Jprobe
  - Jump Probe
- Kretprobe
  - Kernel Return Probe

kprobe



Quelle: http://lwn.net/Articles/132196/

- Löst aus, sobald Code an angebene Adresse ausgeführt wird
- Single-Stepping durch die Funktion
- Ermöglicht (Schreib-)Zugriff auf CPU-Register

struct kprobe

```
struct kprobe {
        [...]
        /* location of the probe point */
        kprobe_opcode_t *addr;

        /* Allow user to indicate symbol name of the probe point */
        const char *symbol_name;

        /* Offset into the symbol */
        unsigned int offset;

        /* Called before addr is executed. */
        kprobe_pre_handler_t pre_handler;

        /* Called after addr is executed, unless... */
        kprobe_post_handler_t post_handler;

        /*
         * ... called if executing addr causes a fault (eg. page fault).
         * Return 1 if it handled fault, otherwise kernel will see it.
         */
        kprobe_fault_handler_t fault_handler;
        [...]
};
```

struct kprobe

```
struct kprobe {
        [...]
        /* location of the probe point */
        kprobe_opcode_t *addr;

        /* Allow user to indicate symbol name of the probe point */
        const char *symbol_name;

        /* Offset into the symbol */
        unsigned int offset;

        /* Called before addr is executed. */
        kprobe_pre_handler_t pre_handler;

        /* Called after addr is executed, unless... */
        kprobe_post_handler_t post_handler;

        /*
         * ... called if executing addr causes a fault (eg. page fault).
         * Return 1 if it handled fault, otherwise kernel will see it.
         */
        kprobe_fault_handler_t fault_handler;
        [...]
};
```

`addr` oder `symbol_name` erforderlich!

struct kprobe

```
struct kprobe {
        [...]
        /* location of the probe point */
        kprobe_opcode_t *addr;

        /* Allow user to indicate symbol name of the probe point */
        const char *symbol_name;

        /* Offset into the symbol */
        unsigned int offset;

        /* Called before addr is executed. */
        kprobe_pre_handler_t pre_handler;

        /* Called after addr is executed, unless... */
        kprobe_post_handler_t post_handler;

        /*
         * ... called if executing addr causes a fault (eg. page fault).
         * Return 1 if it handled fault, otherwise kernel will see it.
         */
        kprobe_fault_handler_t fault_handler;
        [...]
};
```

optional
sichtbar: Adresse vom Struct, Register

struct kprobe

```
struct kprobe {
        [...]
        /* location of the probe point */
        kprobe_opcode_t *addr;

        /* Allow user to indicate symbol name of the probe point */
        const char *symbol_name;

        /* Offset into the symbol */
        unsigned int offset;

        /* Called before addr is executed. */
        kprobe_pre_handler_t pre_handler;

        /* Called after addr is executed, unless... */
        kprobe_post_handler_t post_handler;

        /*
         * ... called if executing addr causes a fault (eg. page fault).
         * Return 1 if it handled fault, otherwise kernel will see it.
         */
        kprobe_fault_handler_t fault_handler;
        [...]
};
```

optional
sichtbar: Adresse vom Struct, Register

struct kprobe

```
struct kprobe {
        [...]
        /* location of the probe point */
        kprobe_opcode_t *addr;

        /* Allow user to indicate symbol name of the probe point */
        const char *symbol_name;

        /* Offset into the symbol */
        unsigned int offset;

        /* Called before addr is executed. */
        kprobe_pre_handler_t pre_handler;

        /* Called after addr is executed, unless... */
        kprobe_post_handler_t post_handler;

        /*
         * ... called if executing addr causes a fault (eg. page fault).
         * Return 1 if it handled fault, otherwise kernel will see it.
         */
        kprobe_fault_handler_t fault_handler;
        [...]
};
```

optional

*_handler

```
/* kprobe pre_handler: called just before the probed instruction is executed */
static int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
    printk(KERN_INFO "pre_handler: p->addr = 0x%p, ip = %lx, flags = 0x%lx\n",
        p->addr, regs->ip, regs->flags);

    /* A dump_stack() here will give a stack backtrace */
    return 0;
}

/* kprobe post_handler: called after the probed instruction is executed */
static void handler_post(struct kprobe *p, struct pt_regs *regs, unsigned long flags)
{
    printk(KERN_INFO "post_handler: p->addr = 0x%p, flags = 0x%lx\n",
        p->addr, regs->flags);
}
```

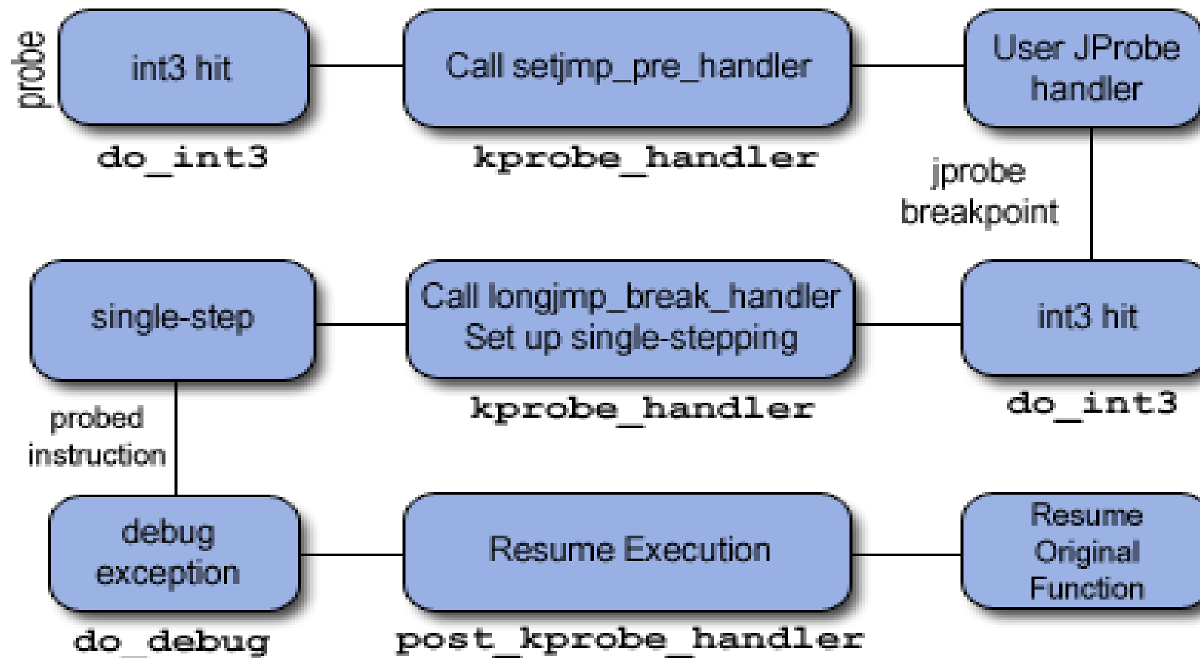
*_handler

```
/* kprobe pre_handler: called just before the probed instruction is executed */
static int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
    printk(KERN_INFO "pre_handler: p->addr = 0x%p, ip = %lx, flags = 0x%lx\n",
        p->addr, regs->ip, regs->flags);

    /* A dump_stack() here will give a stack backtrace */
    return 0;
}

/* kprobe post_handler: called after the probed instruction is executed */
static void handler_post(struct kprobe *p, struct pt_regs *regs, unsigned long flags)
{
    printk(KERN_INFO "post_handler: p->addr = 0x%p, flags = 0x%lx\n",
        p->addr, regs->flags);
}

/*
 * fault_handler: this is called if an exception is generated for any
 * instruction within t
 * single-steps the pro
 */
static int handler_faul
{
    printk(KERN_INFO "f
        p->addr, trapnr

    /* Return 0 because
    return 0;
}
```

```
[14325.417617] Planted kprobe at c043bf02
[14332.697427] pre_handler: p->addr = 0xc043bf02, ip = c043bf03, flags = 0x246
[14332.697433] post_handler: p->addr = 0xc043bf02, flags = 0x246
[14332.697700] pre_handler: p->addr = 0xc043bf02, ip = c043bf03, flags = 0x246
[14332.697704] post_handler: p->addr = 0xc043bf02, flags = 0x246
[14332.697885] pre_handler: p->addr = 0xc043bf02, ip = c043bf03, flags = 0x246
[14332.697889] post_handler: p->addr = 0xc043bf02, flags = 0x246
[14332.697973] pre_handler: p->addr = 0xc043bf02, ip = c043bf03, flags = 0x246
[14332.697977] post_handler: p->addr = 0xc043bf02, flags = 0x246
[14333.152958] pre_handler: p->addr = 0xc043bf02, ip = c043bf03, flags = 0x246
[14333.152965] post_handler: p->addr = 0xc043bf02, flags = 0x246
[14333.153560] kprobe at c043bf02 unregistered
```

jprobe

- Erweiterung von kprobe, macht Funktionsparameter sichtbar



Execution of a JProbe

Quelle: http://lwn.net/Articles/132196/

jprobe

- `addr` muss auf Funktionsanfang zeigen
- `setjmp_pre_handler():` Stack + Register kopieren
- Eigene Funktion mit selber Signatur aufrufen.
  - Argumente der Originalfunktion sind hier sichtbar
  - `jprobe_return()` benutzen!
- `longjmp_break_handler():` Stack + Register wiederherstellen
- Originalfunktion wie kprobe weiter durchlaufen

kretprobe

- kprobe auf Funktion setzen
  - bei Bedarf in `entry_handler()` Zeitstempel speichern
- Returnadresse sichern, durch Trampolin ersetzen
- Returnadresse durch eigene Funktion ersetzen, hier ist der Rückgabewert sichtbar
- An ursprüngliche Returnadresse springen

Pitfalls

- Probes innerhalb Probes?
  - Möglich, werden aber deaktiviert.
- Probes in kprobes.c?
  - Nicht möglich
- Präemptives Multitasking?
  - Wird automatisch deaktiviert
- Probes auf `inline`-Funktionen?
  - Nicht garantiert
- "Rohes", vom User geschriebenes C im Kernel?
  - Systemcrash bei Programmierfehler

Zusammenfassung

- Kprobes: Zu Modulen kompilierter C-Code zur Instrumentierung
- 3 Typen: kprobe, jprobe, kretprobe
  - kprobe: pre_ & post_handler
    - Register, kprobe-Struct
  - jprobe: Jump-Probe, “Dummyfunktion”
    - Argumente, Register, kprobe-Struct
  - kretprobe: schreibt Returnadresse um
    - Rückgabewert, Register, kprobe-Struct

Beispiele

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>

static struct kprobe kp = {
        .symbol_name    = "do_fork",
};

static int __init kprobe_init(void)
{
        int ret;

        ret = register_kprobe(&kp);
        if (ret < 0) {
                printk(KERN_INFO "register_kprobe failed, returned %d\n", ret);
                return ret;
        }
        printk(KERN_INFO "Hello, World!");
        return 0;
}

static void __exit kprobe_exit(void)
{
        unregister_kprobe(&kp);
}

module_init(kprobe_init)
module_exit(kprobe_exit)
MODULE_LICENSE("GPL");
```

Systemtap

- Skriptsprache + Interpreter zur leichteren Instrumentierung
- Gründe:
  - IBMs Dprobes kam schlecht an
  - SUNs DTrace kam gut an
    - CDDL
  - LTT considered “outdated”
- Seit 2005: Entwicklung von Systemtap (...und LTTng)

Systemtap

- Skriptsprache, mit Kernelprivilegien?
- besser als kprobes?
  - stark erweiterte Sicherheitsmechanismen
  - erleichterte Programmierung

Kprobes vs. Systemtap

kp_helloworld.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>

static struct kprobe kp = {
        .symbol_name    = "do_fork",
};

static int __init kprobe_init(void)
{
        int ret;
        ret = register_kprobe(&kp);
        if (ret < 0) { return ret; }
        printk(KERN_INFO "Hello, World!");
        return 0;
}

static void __exit kprobe_exit(void)
{
        unregister_kprobe(&kp);
}

module_init(kprobe_init)
module_exit(kprobe_exit)
MODULE_LICENSE("GPL");
```

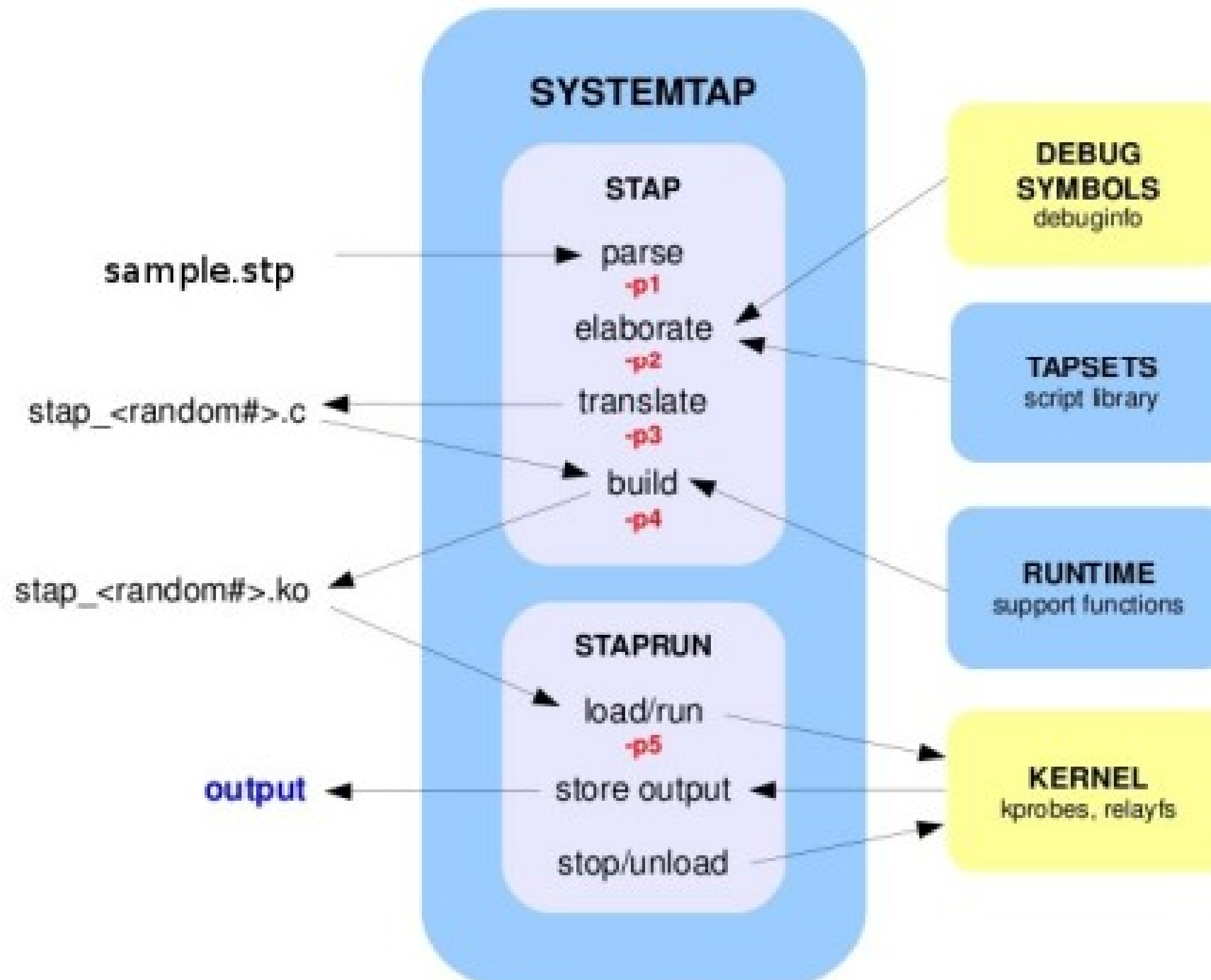
hello-world.stp

```
probe begin {

        print("Hello, World!\n")

        exit()

}
```

Architektur

- Benutzt Kprobes-API + Tracepoints
  - Tracepoints: fest einkompilierte Funktionen
    - Heye erzählt mehr
- Nimmt User Arbeit ab
  - kompiliert Skriptsprache zu Kernelmodul
  - lädt und entlädt Modul
  - erweiterter Funktionsumfang (Timer, ...)

Workflow



Quelle: www.admin-magazin.de/Das-Heft/2010/03/Anwendungen-tracen-mit-Oprofile-und-Systemtap

Skriptsprache .stp

- Systemtap liefert Probe-Punkte und Variablen
- Ausgabe (pretty-printing) von nützlichen, vordefinierten und internen Variablen

→ leichtes Tracing möglich

Benannte Probe-Punkte

`begin`	Start der Systemtap-Session
`end`	Ende der Session
`kernel.function(“sys_open”)`	Eintrittspunkt von sys_open im Kernel
`syscall.close.return`	Rücksprung vom close-Syscall
`timer.ms(200)`	Alle 200 ms

- mehr: `man stapprobes`
- Usage:

```
probe timer.ms(200) { … }
```

Vordefinierte Variablen

`tid()`	Aktuelle Thread-ID
`execname()`	Name des aktuellen Prozesses
`cpu()`	Nummer des aktuellen Prozessors
`gettimeofday_s()`	Timestamp, Sekunden seit epoch
`$$vars`	Wenn verfügbar: alle sichtbaren lokalen Variablen

- mehr: `man stapfuncs`
- Welche Variablen sind in `kernel.functions(“*”)` verfügbar?

```
stap -L 'kernel.functions(“*”)' | less
```

- Rückgabewerte: Strings oder Zahlen

Ausgabe

`print(`*`var`*`)`

- `var`: String oder Nummer
- gibt Variable auf eigener Zeile aus

`printf(“`*`format`*`”, `*`argumente`*`, …)`

- Formatierung wie in C
- Ausgabe komplexer Strings
- “`\n`” nicht vergessen

Kontroll- und Datenstrukturen

- Kommentare
  - “#”, “/* … */”, “//”
- Blöcke
  - { … }
- Conditionals
  - if (*EXPR*) *STATEMENT* [else *STATEMENT*]
  - while (*EXPR*) *STATEMENT*
  - for (*A*; *B*; *C*) *STATEMENT*
- Variablen (lokal + global)
  - foo = “hello world” // lokaler String
  - global array[400] // globales Array
  - a <<< delta_timestamp // Aggregat

“Target Variables”

- Existieren nur bei Probes auf Breakpoints
- Erlauben Zugriff auf Werte aus Programm-Kontext
  - Adressen: &
  - Inhalte: `$, $$`
  - Existenzprüfung: `@defined`
- Bereits bekannt:
  - `$$vars`
- Mehr: `man stap`

Funktionen

- Benannte Blöcke

```
function trace_common() {
        printf("%d %s(%d)", gettimeofday_s(), execname(), pid());
}
```

- Kein Rückgabewert erforderlich
- Rekursion erlaubt

Aggregate

- “automatische” Listen
- Es kann nur angehängt werden
- Beispiele
  - Durchschnitt aller in a akkumulierten Werte:
    - `@avg(a)`
  - ASCII-Art Histogramm von a:
    - `print(@hist_linear(a,0,100,10))`

Tapsets

- Skripte werden in Ordnern zu Tapsets zusammengefasst
- Bei undefinierten Symbolen
  - Systemtap durchsucht Tapsets
  - deklarierende Dateien werden inkludiert

Sicherheit

- Endlosschleifen, -rekursion?
  - Zeitbegrenzung bei Probehandling
- Speicher läuft voll?
  - Speicher wird vollständig bei Initialisierung allokiert
- Null-Pointer-Exceptions? Division by Zero?
  - Generierter C-Code wird vor Kompilierung geprüft
- Wirklich?
  - “Ja”, naja, fast. Fehlerfreiheit ist nicht garantiert
  - Gefundene Bugs werden so schnell es geht behoben

Zusammenfassung

- Systemtap: Skriptsprache & Interpreter, statt direkt Kprobes zu programmieren
- Wandelt einfache Skripte in Module um
- Erleichtert Instrumentierung durch leichten Zugriff auf Probes, Variablen, Funktionen
- Automatisches Laden & Entfernen der Module

Beispiele

```
probe timer.ms(500) {
        print("Hello, World!\n")
}
```

Zusammenfassung

- Kprobes
  - API für Leistungsmessung per Kernelmodul, in C
  - Kaum Schutz vor Programmierfehlern
- Systemtap
  - bequeme Programmierumgebung für Leistungsmessung
  - benutzt intern Kprobes
  - Weitergehender Schutz vor Fehlern

Quellen

- http://sourceware.org/systemtap/
- http://www.kernel.org/doc/Documentation/kprobes.txt
- http://lwn.net/Articles/132196/
- http://sourceware.org/systemtap/kprobes/