JITTY

eine Virtuelle Maschine unter dem Betriebssystem

24./25. September 2020

V. Sieh, B. Heinloth, D. Nguyen, W. Schröder-Preikschat

Department Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Wie alles anfing...

Gesucht: Code-Beispiele aus bekannten Betriebssystemen für Betriebssystem-Vorlesung

Problem:

Linux: Makro-/`#ifdef`-Hölle, selbst-modifizierender Code

***BSD:** viele Makros / viel `#ifdef`

Minix: Mikro-Kern

...: ...

Windows: keine Sourcen einsehbar

Beispiel Linux

Linux-Code:

- sehr viel Code (ca. 1 GByte Source)
- viele Makros
- viele `#define`s
- viele `#ifdef`s
- viele „Hacks”

$\Rightarrow$ sehr unübersichtlich, völlig unwartbar, ...

Linux-Sourcen

Beispiel (Linux-4.18.9) Spinlock-Implementierung:

include/linux/spinlock.h:

```
static __always_inline void spin_lock(spinlock_t *lock)
{
   raw_spin_lock(&lock->rlock);
}
```

include/linux/spinlock.h:

```
#define raw_spin_lock(lock)    _raw_spin_lock(lock)
```

include/linux/spinlock_api_smp.h:

```
#ifdef CONFIG_INLINE_SPIN_LOCK
#define _raw_spin_lock(lock) __raw_spin_lock(lock)
#endif
```

Linux-Sourcen

include/linux/spinlock_api_smp.h:

```
#if !defined(CONFIG_GENERIC_LOCKBREAK) \
    || defined(CONFIG_DEBUG_LOCK_ALLOC)
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    preempt_disable();
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
    LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
}
#endif
```

include/linux/preempt.h:

```
#define preempt_disable() \
do { \
    preempt_count_inc(); \
    barrier(); \
} while (0)
```

include/linux/preempt.h:

```
#define preempt_count_inc() preempt_count_add(1)
```

Linux-Sourcen

include/linux/preempt.h:

```
#if defined(CONFIG_DEBUG_PREEMPT) \
   || defined(CONFIG_PREEMPT_TRACER)
extern void preempt_count_add(int val);
#else
#define preempt_count_add(val)  __preempt_count_add(val)
#endif
```

include/asm-generic/preempt.h:

```
static __always_inline void __preempt_count_add(int val)
{
   *preempt_count_ptr() += val;
}
```

include/asm-generic/preempt.h:

```
static __always_inline volatile int *preempt_count_ptr(void)
{
   return &current_thread_info()->preempt_count;
}
```

Linux-Sourcen

include/linux/thread_info.h:

```
#ifdef CONFIG_THREAD_INFO_IN_TASK
#include <asm/current.h>
#define current_thread_info() ((struct thread_info *)current)
#endif
```

arch/x86/include/asm/current.h:

```
#define current get_current()
```

arch/x86/include/asm/current.h:

```
static __always_inline struct task_struct *get_current(void)
{
   return this_cpu_read_stable(current_task);
}
```

Linux-Sourcen

arch/x86/include/asm/percpu.h:

```
#define this_cpu_read_stable(var) percpu_stable_op("mov", var)
```

arch/x86/include/asm/percpu.h:

```
#define percpu_stable_op(op, var)       \
({                                      \
   typeof(var) pfo_ret__;               \
   switch (sizeof(var)) {               \
   case 1:                              \
      asm(op "b␣"__percpu_arg(P1)",%0" \
          : "=q" (pfo_ret__)            \
          : "p" (&(var)));              \
           break;                       \
   ...
   }                                    \
   pfo_ret__;                           \
})
```

Linux-Sourcen

include/linux/lockdep.h:

```
#define spin_acquire(l, s, t, i) \
        lock_acquire_exclusive(l, s, t, NULL, i)
```

include/linux/lockdep.h:

```
#define lock_acquire_exclusive(l, s, t, n, i) \
        lock_acquire(l, s, t, 0, 1, n, i)
```

include/linux/lockdep.h:

```
#ifdef CONFIG_LOCKDEP
extern void lock_acquire(struct lockdep_map *lock,
     unsigned int subclass, int trylock, int read,
     int check, struct lockdep_map *nest_lock,
     unsigned long ip);
#else
# define lock_acquire(l, s, t, r, c, n, i) \
        do { } while (0)
#endif
```

Linux-Sourcen

kernel/locking/lockdep.c:

```
void lock_acquire(struct lockdep_map *lock,
        unsigned int subclass, int trylock,
        int read, int check, igned long ip)
{
   unsigned long flags;

   if (unlikely(current->lockdep_recursion))
      return;

   raw_local_irq_save(flags);
   check_flags(flags);

   current->lockdep_recursion = 1;
   trace_lock_acquire(lock, subclass, trylock, read,
             check, nest_lock, ip);
   __lock_acquire(lock, subclass, trylock, read, check,
             irqs_disabled_flags(flags), nest_lock, ip, 0, 0);
   current->lockdep_recursion = 0;
   raw_local_irq_restore(flags);
}
```

Linux-Sourcen

...und noch viele Makros/Schachtelungstiefen mehr...

Beispiel Linux

Linux-Code:

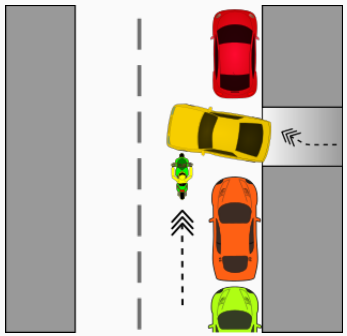
- Code Hand-optimiert für viele Architekturen; vieles in `arch/*`
- Locking Hand-optimiert
  - atomic-Operationen
  - Read/Write-Locks
  - Sequential-Locks
  - RCU-Locks
- z.T. Hand-optimierter, selbstmodifizierender Code (`memcpy`, ...)
- Binär-Code aber Compiler-optimiert für einfache Hardware

Linux-Distribution

Mehrere Optimierungsschritte:

- Debian/SuSE/RedHat/... liefern DVDs mit Kern passend zur Architektur aus (x86, x86-64, sparc, arm, arm64, ...)
- Anwender baut ggf. nochmals Kern für seinen Rechner (mit/ohne MMX, SSE, SSE2, ..., für Mono-/Multi-/Many-Core-Rechner, ...)
- Kern selbst modifiziert sich beim Booten nochmals

Betriebssystem $<-->$ Compiler



Eigentlich:

Aufgabe Betriebssystem-Entwickler (Betrieb):

- Algorithmen entwickeln, die z.B. Ressourcen effizient vergeben, Daten effizient speichern, Daten effizient versenden, ...

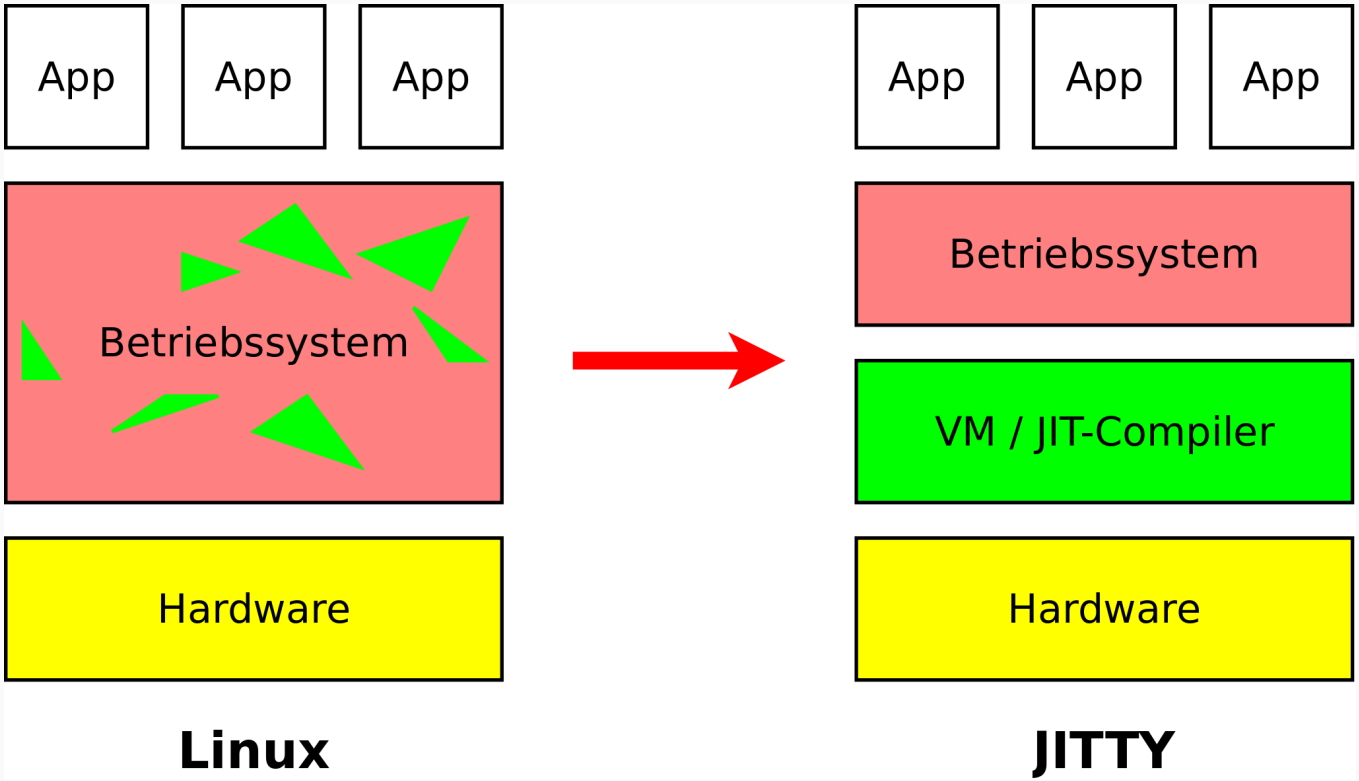
vs.

Aufgabe Compiler-Entwickler (Infrastruktur):

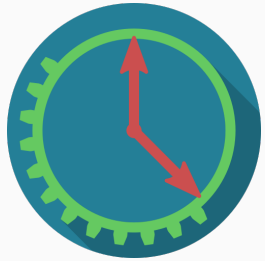
- Ideen entwickeln, wie gegebene Algorithmen auf gegebener Hardware schnell abgearbeitet werden können

JITTY – Idee

Idee **JITTY**-Projekt:



JITTY – Idee



Idee **JITTY**-Projekt:

- BS-Entwickler schreibt „sauberes” BS
- Compiler generiert/optimiert Zwischen-Code für VM/JIT-Compiler
- JIT generiert/optimiert Code für aktuelle Hardware **beim Booten** oder **zur Laufzeit**

JITTY – Einfachheit

Einfachheit:

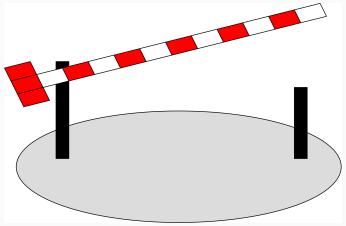
- keine Makro-Magie zur Unterstützung mehrerer Architekturen
- weniger `#ifdef`-Varianten

```
#if CONFIG_XX
    ...
#endif
```

$\Rightarrow$

```
if (CONFIG_XX) {
    ...
}
```

- nur einfaches Locking



Einfachheit:

Ziel: alle Koordinierung mittels „`lock()`” und „`unlock()`”.

- einfach zu verstehen
- z.T. statisch überprüfbar (z.B. LLVM-Thread-Safety-Tests)
- automatisch umwandelbar in bessere Koordinierung

LLVM-Thread-Safety-Tests – Beispiel

```
int var;
```

```
int func(void) {

    int ret = var;

    return ret;
}
```

Keine Warnung!

LLVM-Thread-Safety-Tests – Beispiel

```
lock_t var_lock;
int  var GUARDED_BY(var_lock);
```

```
int  func(void) {


    int ret = var;


    return  ret;
}
```

reading ’var’ requires holding mutex ’var_lock’

```
extern  void  lock(lock_t  *l)
              REQUIRES(irq_disabled)  ACQUIRE(*l);
extern  void  unlock(lock_t  *l)
              REQUIRES(irq_disabled)  RELEASE(*l);
```

```
extern  lock_t  irq_enabled,  irq_disabled;
extern  void  disable_irq()
              RELEASE(irq_enabled)  ACQUIRE(irq_disabled);
extern  void  enable_irq()
              RELEASE(irq_disabled)  ACQUIRE(irq_enabled);
```

LLVM-Thread-Safety-Tests – Beispiel

```
lock_t var_lock;
int  var GUARDED_BY(var_lock);
```

```
int  func(void) {

     lock(&var_lock);
     int ret = var;
     unlock(&var_lock);

     return  ret;
}
```

calling 'lock' requires holding mutex 'irq_disabled'

```
extern  void  lock(lock_t  *l)
              REQUIRES(irq_disabled)  ACQUIRE(*l);
extern  void  unlock(lock_t  *l)
              REQUIRES(irq_disabled)  RELEASE(*l);
```

```
extern  lock_t  irq_enabled, irq_disabled;
extern  void  disable_irq()
              RELEASE(irq_enabled)  ACQUIRE(irq_disabled);
extern  void  enable_irq()
              RELEASE(irq_disabled)  ACQUIRE(irq_enabled);
```

LLVM-Thread-Safety-Tests – Beispiel

```
lock_t var_lock;
int  var GUARDED_BY(var_lock);
```

```
int  func(void) {
     disable_irq();
     lock(&var_lock);
     int ret = var;
     unlock(&var_lock);
     enable_irq();
     return  ret;
}
```

releasing mutex 'irq_enabled' that was not held

```
extern  void  lock(lock_t  *l)
              REQUIRES(irq_disabled)  ACQUIRE(*l);
extern  void  unlock(lock_t  *l)
              REQUIRES(irq_disabled)  RELEASE(*l);
```

```
extern  lock_t  irq_enabled, irq_disabled;
extern  void  disable_irq()
              RELEASE(irq_enabled)  ACQUIRE(irq_disabled);
extern  void  enable_irq()
              RELEASE(irq_disabled)  ACQUIRE(irq_enabled);
```

LLVM-Thread-Safety-Tests – Beispiel

```
lock_t var_lock;
int  var GUARDED_BY(var_lock);
```

```
int REQUIRES(irq_enabled) func(void) {
    disable_irq();
    lock(&var_lock);
    int ret = var;
    unlock(&var_lock);
    enable_irq();
    return ret;
}
```

```
extern void lock(lock_t *l)
            REQUIRES(irq_disabled) ACQUIRE(*l);
extern void unlock(lock_t *l)
            REQUIRES(irq_disabled) RELEASE(*l);
```

```
extern lock_t irq_enabled, irq_disabled;
extern void disable_irq()
            RELEASE(irq_enabled) ACQUIRE(irq_disabled);
extern void enable_irq()
            RELEASE(irq_disabled) ACQUIRE(irq_enabled);
```

LLVM-Thread-Safety-Tests – Beispiel

```
lock_t var_lock;
int var GUARDED_BY(var_lock);
```

```
int REQUIRES(irq_enabled) func(void) {
    disable_irq();
    lock(&var_lock);
    int ret = var;
    if (ret < 0) {
        return 0;
    }
    unlock(&var_lock);
    enable_irq();
    return ret;
}
```

warning: mutex ’irq_disabled’ is not held on every path

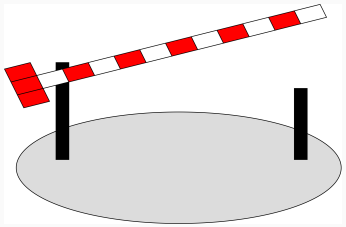
LLVM-Thread-Safety-Tests – Beispiel

```
extern int REQUIRES(irq_enabled) func(void);
```

```
    ...
    disable_irq();
    ...
    int ret = func();
    ...
    enable_irq();
    ...
```

calling ’func’ requires holding mutex ’irq_enabled’

JITTY



Einfachheit:

Umwandlung von Lock-Code per JIT-Compiler in

- Atomic-Operationen
- Read-/Write-Locks
- Sequential-Locks
- (RCU-Locks)

oder (im zu komplizierten Fall): Locks bleiben

JITTY – Locking-Optimierung

Beispiel:

```
struct obj {
    lock_t lock;
    int count GUARDED_BY(lock);
    ...
};
```

Original-Code:

```
lock(&obj->lock);
obj->count++;
unlock(&obj->lock);
```

```
lock(&obj->lock);
count = --obj->count;
unlock(&obj->lock);
if (count == 0) {
    free(obj);
}
```

optimierter Code:

```
atomic_inc(&obj->count);
```

```
count = atomic_dec(&obj->count);
if (count == 0) {
    free(obj);
}
```

Beispiel:

Original-Code:

```
lock(&obj->lock);
assert(1 <= obj->count);
obj->count++;
unlock(&obj->lock);
```

```
lock(&obj->lock);
assert(1 <= obj->count);
count = --obj->count;
unlock(&obj->lock);
if (count == 0) {
    free(obj);
}
```

```
lock(&obj->lock);
assert(1 <= obj->count);
unlock(&obj->lock);
```

optimierter Code:

```
write_lock(&obj->lock);
assert(1 <= obj->count);
obj->count++;
write_unlock(&obj->lock);
```

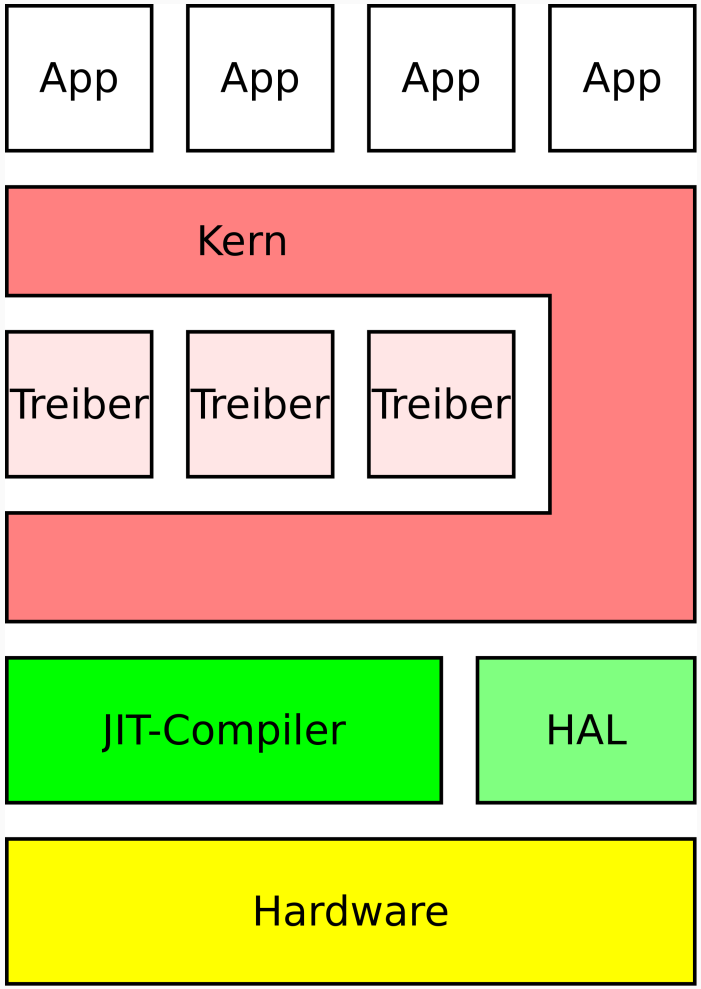
```
write_lock(&obj->lock);
assert(1 <= obj->count);
count = --obj->count;
write_unlock(&obj->lock);
if (count == 0) {
    free(obj);
}
```

```
read_lock(&obj->lock);
assert(1 <= obj->count);
read_unlock(&obj->lock);
```

JITTY

Trennung:

- JIT
  CPU-abhängig
- HAL
  Board-/MMU-abhängig
- Treiber
  Bus-abhängig
- Kern
  Architektur-unabhängig



JITTY

Sicher**er**es System möglich:

Kern: Programmierung in sicherer Programmiersprache (Java, safe-Rust, safe-C#, ...) möglich

Treiber: Programmierung in sicherer Programmiersprache möglich mit „Bus-Objekten” (aber DMA!)

JIT: könnte Sicherheit von Kern/Treibern garantieren

HAL: der „unangenehme” Rest...

JITTY – Sicherheit

Sicherheit:

- Zwischencode kann vor dem Übersetzen vom JIT geprüft werden
- nur sichere Zeiger im Kern und in den Treibern
- Locking über Thread-Safety-Analyse im Voraus überprüft
- keine Deadlocks, da streng hierarchisch

JITTY – Optimierungen

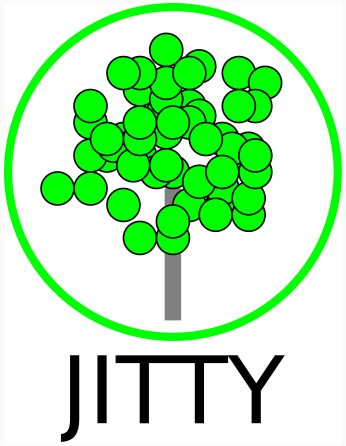
Alle Optimierungen denkbar, die die JITs von z.B. LLVM, JVM, Mono, ... auch durchführen. Z.B.:

- Inlining
- Link-Time-Optimization
- ...
- Integration Debugger/Statistik
- **Anpassungen an aktuelle Hardware**

ggf. zusätzlich:

- **Anpassungen an aktuellen Ablauf**
- Randomization
- ...

JITTY – Optimierungen



Optimierungen:

- Performance (`-O3`)
- Speicherbedarf (`-Os`)
- Energiebedarf („`-Oe`”)

JIT nutzt z.B. verschiedene Locking-Algorithmen für verschiedene Szenarien (Mono-Prozessor, SMP, NUMA, ...)

Keine Änderungen am Kern!

Stattdessen: Aufruf anderer Optimierungsläufe des JIT-Compilers

JITTY – Performance

Performance:

- Locking automatisch zu Atomic-Instruktionen, R-/W-Locks, ... umgebaut
- da sauberer Programm-Code, gute Optimierungen vom Compiler möglich
- Link-Time-Optimization möglich
- durch JIT Anpassung an Hardware möglich (`-march=native`)

$=>$ Performance ähnlich Linux möglich

JITTY – aktueller Stand

Aktueller Stand JIT-Compiler:

JAVA-JIT (OpenJVM):

- bootet
- kann Programme ausführen
- JNI jedoch sehr klein

LLVM-JIT:

- bootet
- übersetzt LLVM-Zwischen-Code

CIL-JIT (Mono):

- bootet
- übersetzt CIL-Zwischen-Code

(Genutzt wird jeweils „StuBSmI”.)

JITTY – aktueller Stand

Aktueller Stand BS:

- JITTY-OS
  - bootet (x86, x86_64, aarch64)
  - startet Debian-Linux
    (3.1 bis 6.0) Userland (binär-kompatibel!)
  - X11/GNOME funktionieren z.T.
  - Zeilen Code: Kern ca. 55.000, HAL ca. 10.000,
    Treiber ca. 20.000

Aktueller Stand Optimierer:

- Locking-Optimierer:
  - noch nicht funktionsfähig

JITTY

Danke für die Aufmerksamkeit!

Fragen?

jitty@lists.informatik.uni-erlangen.de