Übung zu Betriebssysteme

Aufgabe 6: Ereignisbearbeitung und Synchronisation

Wintersemester 2021/22

Bernhard Heinloth

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

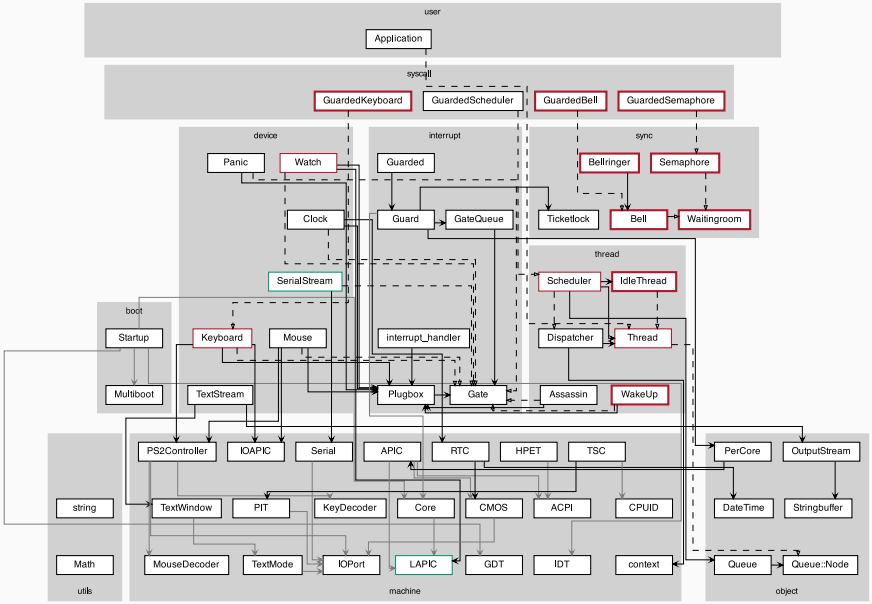
TECHNISCHE FAKULTÄT

Lernziel

- Synchronisation mit Hilfe von Semaphoren und anderen Kernobjekten

Aufgabe

- Implementierung von Semaphoren mit passivem Warten
  - Verwendung in der neuen `getkey()` Funktion
- Zeitgesteuertes Schlafen der Threads
  - *Optional:* Demonstration mittels PC Speaker
- Leerlauf des Prozessor (falls keine Threads vorhanden)
  - *Optional:* Energiesparender „Tickless Kernel“
- Kapselung in Systemaufrufschnittstellen (`syscall`)


user
Application
syscall
GuardedKeyboard
GuardedScheduler
GuardedBell
GuardedSemaphore
device
interrupt
sync
Panic
Watch
Guarded
BellringerSemaphore
Clock
Guard
GateQueue
Ticketlock
Bell
Waitingroom
thread
SerialStream
Scheduler
IdleThread
boot
Startup
Keyboard
Mouse
interrupt_handler
Dispatcher
Thread
Multiboot
TextStream
Plugbox
Gate
Assassin
WakeUp
PS2Controller
IOAPIC
Serial
APIC
RTC
HPET
TSC
PerCore
OutputStream
string
TextWindow
PIT
KeyDecoder
Core
CMOS
ACPI
CPUID
DateTime
Stringbuffer
Math
MouseDecoder
TextMode
IOPort
LAPIC
GDT
IDT
context
Queue
Queue::Node
utils
machine
object

(viel zu) viele Wartezimmer

- jede Semaphore ist gleichzeitig ein Wartezimmer
- und jeder Wecker (`bell`) ebenfalls
  - auch wenn bei unseren Weckern nur je ein Faden darin weilt
  - → einfache Implementierung, da wir beim Wecker die neuen Ablaufplanmethoden für die Semaphoren verwenden können.
    Wirkt aber auf den ersten Blick halt komisch.
- Umsetzung durch Ableitung von `waitingroom`
- Wecker werden zur Laufzeit als temporäre Objekte erzeugt (d.h. sie liegen auf dem Stapelspeicher)
- Alle aktiven Wecker werden selbst wieder in einer verketteten Liste (vom `bellringer`) verwaltet

Richtig ticken

Der `bellringer` prüft regelmäßig die Wecker

- unter Verwendung des LAPIC Timers
- welcher mit `windup(1000)` auf Millisekundentakt gestellt wird
- es reicht, wenn eine CPU das übernimmt

Leerlauf

Problem: zu wenig Threads bereit

Lösung: je ein `IdleThread` pro CPU

```
void IdleThread::action() {
    while (true) {
        if (!Scheduler::isEmpty())
            GuardedScheduler::resume();
    }
}
```

CPU fungiert effektiv als Heizkörper, besser wäre jedoch ein Schlafzustand

Leerlauf

`Core::idle()` hält CPU bis zum nächsten Interrupt an
(mittels atomaren `sti` und `hlt`)

```
void IdleThread::action() {
    while (true) {
        if (Scheduler::isEmpty())
            Core::idle();
        else
            GuardedScheduler::resume();
    }
}
```

Thread ready

Durch Aufwachen eines wartenden Threads (oder Neueinplanung bei MPStuBS) kann ein **Lost-Wakeup** passieren!

Leerlauf

`Core::idle()` hält CPU bis zum nächsten Interrupt an
(mittels atomaren `sti` und `hlt`)

```
void IdleThread::action() {
    while (true) {
        Core::Interrupt::disable();
        if (Scheduler::isEmpty())
            Core::idle();
        else {
            Core::Interrupt::enable();
            GuardedScheduler::resume();
        }
    }
}
```

Leerlauf mittels `Core::idle()`

```
namespace Core {
    inline void idle() {
        asm volatile("sti\n\t"
                     "hlt\n\t"
                               ::: "memory");
    }
}
```

Optional: Nicht mehr ticken

Im Leerlauf wird der Kern jedoch immer noch durch Unterbrechungen von unserer `watch` aufgeweckt → **erhöhter Energieverbrauch**

- Im Leerlauf kann der LAPIC Timer deaktiviert werden (Interrupt maskieren)
- Erweiterung der `watch` im `IdleThread` anwenden:

```
if (Scheduler::isEmpty()){
    Watch::block();
    Core::idle();
    Watch::unblock();
}
```

- Sonderbehandlung bei der für den `bellringer` verantwortlichen CPU berücksichtigen!

Leerlauf in MPStuBS

Problem: Sobald ein Thread bereit ist, soll eine CPU im Leerlauf sofort mit der Abarbeitung beginnen

Lösung: Aufwecken der CPU mittels IPI