

EZStubs Entwicklungsumgebung

Einführung

Fabian Scheler, Peter Ulbrich, Niko Böhm

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
`www4.informatik.uni-erlangen.de`

5. November 2009

Allgemein

Vorbereitung

- SVN-Repository

- Makefiles

Inside EZStubs

- Verzeichnisstruktur

- Makefilesystem

- Startup

- Unterbrechungssynchronisation

- Scheduler

- Timer

Testfälle

- Implementierung

- Ablaufkontrolle

- Hilfsbibliothek

- Fehlersuche

Tutorial

Intention

- ▶ Familie von Echtzeitbetriebssystemen
- ▶ etwas komplexere Klassenhierarchie
 - ▶ trotzdem: Fokus eher auf Verständlichkeit denn auf Effizienz
- ▶ Entwickelt für diese Übungen
- ▶ Verfügbar für
 - ▶ Lego RCX (H8/3297)
 - ▶ Gameboy Advance (ARM7)
 - ▶ Nintendo DS Lite (ARM9)
- ▶ Implementierung hat noch die ein oder andere Schwäche

Hinweis

ergänzende Informationen findet man hier:

http://www4.informatik.uni-erlangen.de/Lehre/WS09/V_EZS/Uebung/

Initialisierung des SVN-Repositories

1. Die Struktur des Repositories anlegen:

```
faii48b:~temp> mkdir tmpdir  
faii48b:~temp> cd tmpdir/  
faii48b:~/temp/tmpdir> mkdir  
faii48b:~/temp/tmpdir> mkdir trunk  
faii48b:~/temp/tmpdir> mkdir branches  
faii48b:~/temp/tmpdir> mkdir tags
```

2. vorgabe_0.tar.gz in das Unterverzeichnis trunk entpacken:

```
faii48b:~/temp/tmpdir> tar xzf vorgabe_0.tar.gz -C trunk  
faii48b:~/temp/tmpdir> ll trunk  
insgesamt 12  
drwxr-xr-x 2 scheler i4staff 512 2006-10-27 10:33 debug  
drwxr-xr-x 2 scheler i4staff 512 2006-10-27 10:02 devices  
...  
drwxr-xr-x 2 scheler i4staff 512 2006-10-13 10:35 tests  
faii48b:~/temp/tmpdir> rm vorgabe0.tar.gz
```

Initialisierung des SVN-Repositories (Forts.)

3. Die Verzeichnisstruktur in das Subversion-Repository importieren:

```
faii48b:~/temp/tmpdir> svn import . https://www4.informatik.uni-erlangen.de:8088/i4ezs/  
test -m 'initial repository layout'  
Hinzufuegen trunk  
...  
Hinzufuegen trunk/Makefile  
Hinzufuegen branches  
Hinzufuegen tags  
  
Revision 1 uebertragen.
```

4. Das importierte Verzeichnis ist **nicht** versioniert! → auschecken:

```
faii48b:/home/scheler> svn co https://www4.informatik.uni-erlangen.de:8088/i4ezs/test/  
trunk ezstubs  
A ezstubs/infra  
...  
A ezstubs/make/variables.mk  
Ausgecheckt, Revision 1.  
faii48b:/home/scheler> ll ezstubs/  
insgesamt 12  
drwxr-xr-x 3 scheler i4staff 512 2006-10-30 17:13 debug  
drwxr-xr-x 3 scheler i4staff 512 2006-10-30 17:13 devices  
...  
drwxr-xr-x 3 scheler i4staff 512 2006-10-30 17:13 tests  
/home/scheler>
```

Initialisierung des SVN-Repositories (Forts.)

5. Branch (`time_triggered`) für Aufgaben 1 und 2 anlegen:

```
faii48b:/home/scheler> cd ezstubs/  
faii48b:~/ezstubs> svn copy . https://www4.informatik.uni-erlangen.de:8088/i4ezs/test/  
branches/time_triggered
```

6. Auf den neu angelegten Branch wechseln:

```
faii48b:~/ezstubs> svn switch https://www4.informatik.uni-erlangen.de:8088/i4ezs/test/  
branches/time_triggered
```

7. `vorgabe_1.tar.gz` in das Verzeichnis `ezstubs` entpacken:

```
faii48b:~/ezstubs> tar xjf vorgabe1.tar.gz
```

8. Dateien aus `vorgabe_1.tar.gz` zum Repository hinzufügen:

```
faii48b:~/ezstubs> svn add ...  
A ...  
...  
A ...  
faii48b:~/ezstubs>svn commit . -m 'Dateien aus vorgabe_1.tar.gz'  
Hinzufuegen ...  
...  
Uebertrage Daten .....  
Revision 2 uebertragen.  
faii48b:~/ezstubs>
```

Pfade in den Makefile überprüfen/anpassen

In den Makefiles müssen drei Pfade korrekt gesetzt sein. Die passenden Werte für den CIP-Pool sind:

Dateiname	Variable	Wert
make/variables_arm.mk	TOOL_PATH	/proj/i4ezs/tools/gnuarm
make/variables_arm_nds.mk	GBA_DIR	/proj/i4ezs/tools/GBA
make/variables_arm_nds.mk	DESMUME_DIR	/proj/i4ezs/tools/desmume

Das Verzeichnis `ezstubs` enthält folgende Unterverzeichnisse:

Name	Beschreibung
<code>debug</code>	Bibliothek zum Schreiben von Testfällen. Enthält z.B. Makros und Funktionen zum gezielten Auslösen von Unterbrechungen, zur Zeitmessung oder um eine bestimmte Zeitspanne zu warten.
<code>devices</code>	Gerätetreiber. Hier finden sich alle Gerätetreiber, sowohl abstrakte Geräte, die eine einheitliche Schnittstelle zur Anwendung ermöglichen, als auch die Abbildung von real existierender Peripherie auf Software-Konstrukte.
<code>gen</code>	In diesem Verzeichnis landet alles, was in durch einen Aufruf von <code>make</code> erzeugt wird - Objektdateien, Doxygen-Dokumentation ...
<code>infra</code>	Typdefinitionen für EZStubs und andere kleine Helfer, die in den Implementierungsdateien von EZStubs verwendet werden, sich aber nicht sinnvoll zu anderen Modulen zuordnen lassen.
<code>interrupt</code>	Unterbrechungsbehandlung und Unterbrechungssynchronisation.
<code>make</code>	Alles was nötig ist, um EZStubs zu bauen.
<code>object</code>	Eine kleine Hilfsbibliothek für EZStubs, enthält z.B. eine verkettete Liste.
<code>shutdown</code>	Alles was EZStubs herunterfährt.
<code>startup</code>	Alles was EZStubs zum starten benötigt.
<code>tests</code>	Testfälle
<code>thread</code>	Der Thread-Abstraction-Layer. Hier findet man die Fadenimplementierung, den Dispatcher, den Scheduler ...

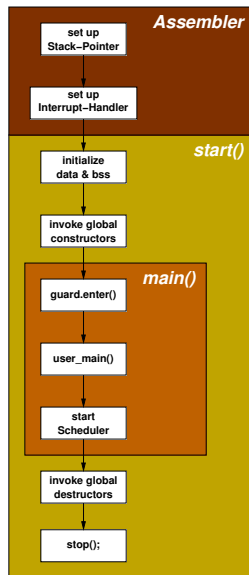
Die EZStubs-Makefile implementiert folgende Targets:

Target	Beschreibung
help	Gibt die verfügbaren Make-Targets zusammen mit einer Erläuterung auf der Konsole aus (entspricht in etwa dieser Tabelle).
all	Übersetzt den Quelltext von EZStubs und erzeugt die EZStubs-Bibliothek.
abgabe	Eine Aufgabe abgeben - die Aufgabe wird durch die Variable <code>EXERCISE = aufgabex</code> , <code>x = 1,2,3,4,5</code> bestimmt.
doxygen	Erzeugt die HTML-Quelltext-Dokumentation.
showtests	Zeigt die verfügbaren Testfälle an.
buildtests	Übersetzt und bindet alle verfügbaren Testfälle.

Die EZStubs-Makefile implementiert folgende Targets:

Target	Beschreibung
<test>.elf	Übersetzt und bindet den Testfall <test>.
<test>.clean	Räumt den Testfall <test> auf.
<test>.sim_exec	Führt den Testfall <test> auf dem DeSmuME aus.
<test>.sim_gdb	Startet den DeSmuME, startet den GDB-Debugger und lädt den Testfall <test> in den DeSmuME.
<test>.sim_ddd	Startet den DeSmuME, startet den DDD-Debugger und lädt den Testfall <test> in den DeSmuME.
<test>.target_exec	Lädt den Testfall <test> auf den Nintendo DS Lite und führt in dort aus (diese Option steht für den Nintendo DS Lite leider nicht zur Verfügung).
<test>.target_debug	Startet den DDD-Debugger und lädt den Testfall <test> in den Nintendo DS Lite (diese Option steht für den Nintendo DS Lite leider nicht zur Verfügung).
checkalltests_sim	Übersetzt und bindet alle Testfälle und führt sie anschließend auf dem Simulator aus. Die Testfallergebnisse werden protokolliert und angezeigt.

Startup



1. SP Register initialisieren:

- ▶ Interrupt Stack & System Stack
- ▶ Startup: nutzt den Interrupt Stack

2. bss & data initialisieren

- ▶ bss mit 0 beschreiben
- ▶ initialisierte Variablen kopieren

3. globale Konstruktoren ausführen

4. kritischen Abschnitt betreten

5. benutzerdefinierte main-Funktion

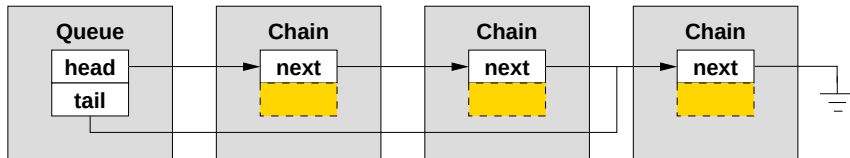
6. Scheduler starten

7. globale Destruktoren ausführen

8. Ende (z.B. warten oder reset)

Warum Unterbrechungssynchronisation?

- ▶ Unterbrechungen (nicht Traps!)
 - ▶ treten asynchron auf
 - ▶ sind nicht vorhersagbar in Häufigkeit und Zeitpunkt
- ▶ **Problem:** Wie soll der Zugriff auf gemeinsame Daten stattfinden?
- ▶ Beispiel: verkettete Liste:



Verkettete Liste

Klasse Chain, Klasse Queue

```
class Chain {  
public:  
    Chain* next;  
};  
  
class Queue {  
protected:  
    Chain* head;  
    Chain** tail;  
public:  
    Queue() { head = 0;  
              tail = &head; }  
    void enq(Chain* item);  
    Chain* deq();  
};
```

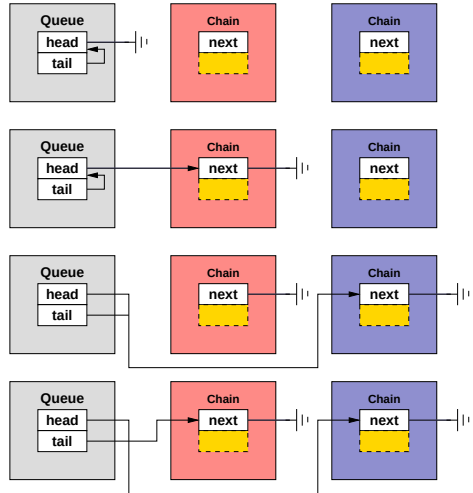
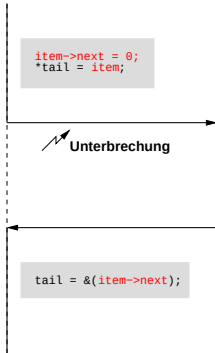
Implementierung

```
void Queue::enq(Chain* item) {  
    item->next = 0;  
    *tail = item;  
    tail = &(item->next);  
}  
  
Chain* Queue::deq() {  
    Chain* item;  
    item = head;  
    if(item) {  
        head = item->next;  
        if(!head) tail = &head;  
        else item->next = 0;  
    }  
    return item;  
}
```

Verkettete Liste - enqueue () überlappt enqueue ()

unterbrochenes
Programm

Unterbrechungs-
behandlung



ISR-/DSR-Synchronisation

- ▶ Lösung: Aufteilung der Unterbrechungsbehandlung
- ▶ ISR (**Interrupt Service Routine**)
 - ▶ werden **asynchron** zum Programm ausgeführt
 - ▶ dürfen keine gemeinsamen Datenstrukturen modifizieren
 - ▶ können die Ausführung einer DSR anfordern
- ▶ DSR (**Deferred Service Routine**)
 - ▶ werden **synchron** zum Programm ausgeführt
 - ▶ dürfen gemeinsame Datenstrukturen modifizieren
- ▶ Ausführung der DSRs kann verhindert/verzögert werden

Unterbrechungsbehandlung

- Klasse `Gate`: Grundlage für die Implementierung

Klasse `Gate`

```
class Gate {  
public :  
    // ISR  
    // Rueckgabewert true: fuehre DSR aus  
    // Rueckgabewert false: fuehre DSR nicht aus  
    virtual bool isr() = 0;  
    // DSR  
    virtual void dsr() = 0;  
    // Setze Interrupt-Quelle zurueck  
    virtual void acknowledge() = 0;  
};
```


Synchronisation

- ▶ Klasse `Guard` sichert kritische Abschnitte

Klasse `Guard`

```
class Guard {  
public :  
    // Aufrufe von enter() und leave() muessen  
    // immer paarweise erfolgen  
  
    // Betrete einen kritischen Abschnitt  
    // unterbinde Ausfuehrung von DSRs  
void enter();  
    // Verlasse einen kritischen Abschnitt  
    // Fuehre anstehende DSRs aus  
void leave();  
}
```

Synchronisation - Scoped Locking

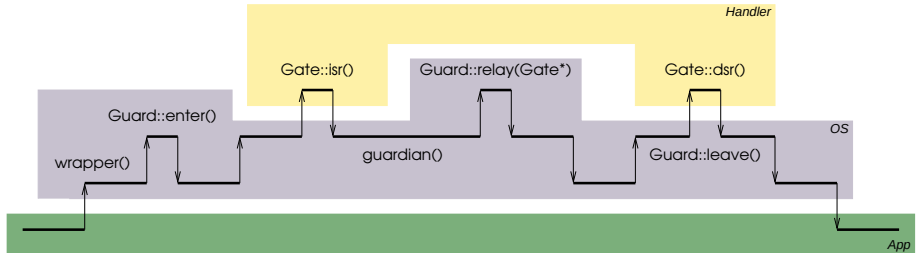
Klasse Secure

```
class Secure {  
public :  
    Secure() { guard.enter(); }  
    ~Secure() { guard.leave(); }  
};
```

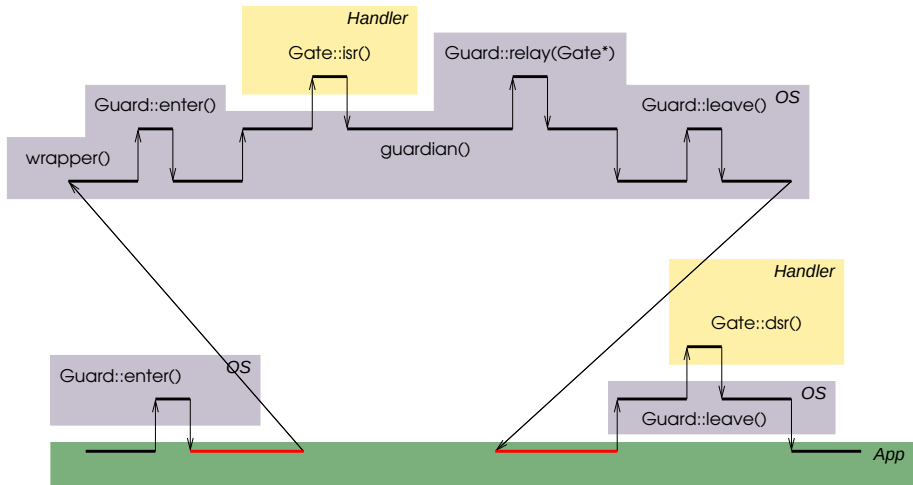
Verwendung

```
void do_something_crititical() {  
    Secure secure;  
  
    // do critical stuff ...  
}
```

Unterbrechung - Ablauf



Unterbrechung - Ablauf (secured)



Allgemein

Der Scheduler ...

- ▶ verwaltet das Betriebsmittel CPU
- ▶ kann verschiedene Strategien bzw. Verfahren verwenden
 - ▶ zeit-/ereignisgesteuert
 - ▶ deterministisch/probabilistisch
 - ▶ ...
- ▶ muss vom System aktiviert werden (sog. **Points of Rescheduling**)
 - ▶ diese entscheiden über die Verdrängbarkeit von Fäden
- ▶ hat (in EZStubs) zwei verschiedene Schnittstellen
 - ▶ Benutzerschnittstelle
 - ▶ Systemschnittstelle

Benutzerschnittstelle

- ▶ hängt von der konkreten Implementierung des Schedulers ab
- ▶ z.B. `Schedule_Table_Scheduler`
 - ▶ Ablauftabelle setzen
 - ▶ Ablauftabelle wechseln
- ▶ z.B. `Multilevel_Queue_Scheduler`
 - ▶ Faden aktivieren
 - ▶ Faden beenden
 - ▶ Kontrolle über den Prozessor abgeben
 - ▶ ...

Systemschnittstelle

- ▶ wird von EZStubs benutzt um den Scheduler zu *aktivieren*
- ▶ Systemschnittstelle:
 - ▶ `void reschedule()`
 - ▶ `void start()` (**darf nicht zurückkehren!!!**)
 - ▶ Klasse `Scheduler_Base`

Klasse `Scheduler_Base`

```
class Scheduler_Base {  
public :  
    // ein Thread-Wechsel muss durchgefuehrt werden  
    set_need_reschedule() ;  
    // muss ein Thread-Wechsel durchgefuehrt werden?  
    get_need_reschedule() ;  
};
```

Aktivierung

- ▶ beim Verlassen des Kerns in `Guard::leave()`

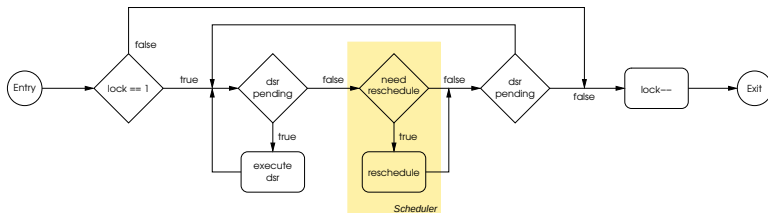


Abbildung: `Guard::leave()`

- ▶ nach der Benutzerdefinierten Startfunktion `user_main()`: Aufruf von `start()`
- ▶ wird in EZStubs nicht verwendet: explizit in blockierenden Systemaufrufen

Stack

- ▶ ARM-Prozessoren haben mehr als einen Stack Pointer
 - ▶ hängt vom jeweiligen Prozessor-Modus ab
 - ▶ Threads laufen auf dem System-Stack
 - ▶ Interrupts laufen auf dem IRQ- (ISRs) und Supervisor-Stack (DSRs)
- ▶ Fadenkontext kann nicht auf dem IRQ-Stack abgelegt werden
- ▶ im Interrupt-Handler sind Umschaltungen notwendig

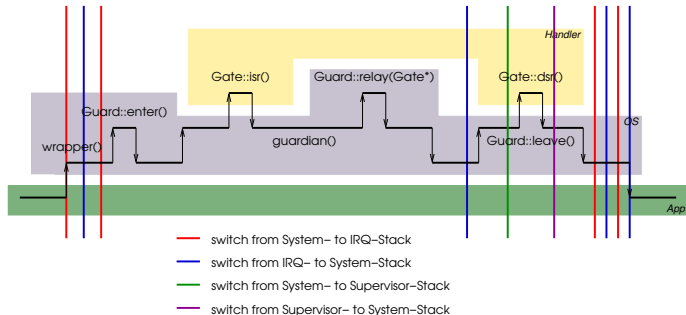


Abbildung: Umschaltungen des Stacks in der Unterbrechungsbehandlung

Stack - Guard::leave()

- nur DSRs werden auf dem IRQ-Stack abgearbeitet

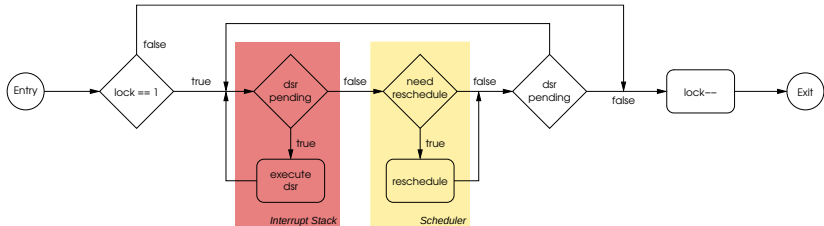
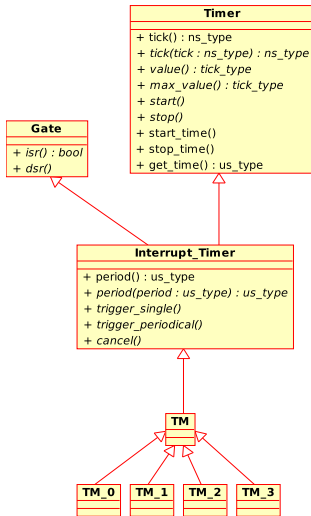


Abbildung: Guard::leave()

Überblick



► Timer auf dem NDS

- 4 Timer $TM_{\{0,1,2,3\}}$ für den ARM7
- 4 Timer $TM_{\{0,1,2,3\}}$ für den ARM9
- alle Timer sind identisch
- 16 bit Zählerregister
- kaskadierbar zu 32, 48 und 64 bit
- Unterbrechung beim Überlauf

► Klasse Timer

- Zeitmessung

► Klasse Interrupt_Timer

- Eieruhr
- Erzeugung von Unterbrechungen
 - *single-shot* oder zyklisch

Klasse Timer

ns_type tick() und **ns_type tick(ns_type)**

- ▶ Dauer eines Zählschritts
- ▶ wie lange dauert es, bis das Zählerregister inkrementiert wird
- ▶ nicht alle Zeitspannen darstellbar $\mapsto$ Rückgabewert
- ▶ invalidiert `us_type period(us_type)`

tick_type value() und **tick_type max_value()**

- ▶ aktuellen/maximal möglichen Wert des Zählerregisters abfragen

void start() und **void stop()**

- ▶ Zählvorgang starten und stoppen

void start_time() und **void stop_time()**

- ▶ Zeitmessung starten und stoppen

us_type get_time()

- ▶ Ergebnis der letzten Zeitmessung abfragen

Klasse Interrupt_Timer

us_type period() und **us_type period(us_type)**

- ▶ Dauer bis zur nächsten Unterbrechung
- ▶ ab dem nächsten Aufruf von `trigger_{single, periodical}()`
- ▶ nicht alle Zeitspannen darstellbar $\mapsto$ Rückgabewert
- ▶ es darf keine Unterbrechung erzeugt werden $\mapsto$ vorher `cancel()`
- ▶ invalidiert `ns_type tick(ns_type)`

void trigger_single()

- ▶ Erzeugen **einer** Unterbrechung

void trigger_periodical()

- ▶ **Periodisches** Erzeugen von Unterbrechungen

void cancel()

- ▶ das Erzeugen der Unterbrechungen wird gestoppt
- ▶ der Timer wird angehalten

Warum? Wie? Wann? Wo?

- ▶ Funktioniert das Programm?
 - ▶ Testen ...
 - ▶ andere Verifikationsmethoden (formal, ...)
- ▶ Testfälle sind **einfach**!
 - ▶ feingranular
 - ▶ testen nur **eine** Funktion
- ▶ eure Testfälle unterstützen den Entwickler
 - ▶ stehen **bald** zur Verfügung
 - ▶ sind hierarchisch strukturiert
- ▶ unsere Testfälle sind *Abnahmetests*
 - ▶ Testen das *fertige Produkt*
 - ▶ sind u.U. relativ komplex

Was ist ein Testfall?

Ein Testfall ...

- ▶ ... ist eine kleine Anwendung für das EZStubs-Betriebssystem
- ▶ ... ist eine Ansammlung von Funktionen, die vom Betriebssystem aufgerufen werden:
 - ▶ die Methode `void action()` eines Fadens
 - ▶ die Methoden `bool isr()` und `void dsr()` einer Unterbrechungsbehandlung
 - ▶ Callback-Funktionen von Alarmen
- ▶ ... beginnt immer in der Funktion `user_main()`

Beispiel

```
// evtl. benoetigte Header werden eingebunden

#include "debug/testing.h"
#include "thread/guarded_scheduler.h"
#include "thread/thread.h"

// Aufzeichnung des Verlaufs wird initialisiert

const char* right_sequence = "abcdabcdabcd";
    char sequence[12];
volatile int counter = 0;

...
```


Beispiel (Forts.)

```
...  
// Fadenimplementierung fuer den Testfall , enthaelt  
// die Anwendungslogik des Testfalls  
  
class Test_Thread : public Thread {  
    static unsigned char runs;  
    static Thread* checker_thread;  
    char thread_char;  
public:  
    Test_Thread(void* tos , char c) : Thread(tos) , thread_char(c) {}  
    void action() {  
        while(1) {  
            runs++; // Anzahl der Gesamtausfuehrungen zaelen  
            sequence[counter++] = thread_char; // Verlauf protokollieren  
            if(runs == 12) scheduler.add(checker_thread);  
            if(runs > 8)  scheduler.exit();  
            scheduler.yield();  
        }  
    }  
};  
  
unsigned char Test_Thread::runs = 0;  
...
```

Beispiel (Forts.)

```
...  
// der Checker_Thread ueberprueft abschliessend die Sequenz  
  
class Checker_Thread : public Thread {  
public:  
    Checker_Thread(void* tos) : Thread(tos) {}  
  
    void action() {  
  
        // aufgezeichnete Sequenz und Sollsequenz stimmen nicht ueberein  
        // => Testfall ist fehlgeschlagen  
        if (!check_sequence(right_sequence, sequence))  
            Panic("Wrong sequence!");  
  
        // andernfalls ist der Testfall erfolgreich  
        TestOK("Test successful finished!");  
    }  
};  
...
```

Beispiel (Forts.)

```
...  
// Stack fuer die verschiedenen Faeden allokkieren  
static unsigned int stack[4][256];  
  
// Faeden, Unterbrechungsbehandlungen etc. werden als  
// globale Objekte instantiiert  
Test_Thread t1(&stack[0][0], 'a'), t2(&stack[1][0], 'b');  
Test_Thread t3(&stack[2][0], 'c'), t4(&stack[3][0], 'd');  
Checker_Thread checker(&stack[1][0]);  
Thread* Test_Thread::checker_thread = &checker;  
  
// hier werden die Faeden aktiviert und  
// der Scheduler initialisiert und gestartet  
void user_main() {  
    init_sequence(right_sequence, sequence); // Sequenz initialisieren  
    scheduler.add(&t1);  
    scheduler.add(&t2);  
    scheduler.add(&t3);  
    scheduler.add(&t4);  
    scheduler.set_timeslice(30000);  
}
```

Wann ist ein Test erfolgreich?

- ▶ Testfall erfolgreich: `TestOK ("Message") ;`
 - ▶ alle API-Aufrufe haben sich wie erwartet Verhalten (Seiteneffekte, Rückgabewerte, ...).
 - ▶ bei mehrfädigen Testfällen entspricht der tatsächliche dem vorgesehenen Ablauf.

- ▶ Testfall fehlgeschlagen: `Panic ("Message") ;`
 - ▶ falscher Rückgabewert
 - ▶ fehlender oder nicht-erwarteter Seiteneffekt
 - ▶ falscher Verlauf des Testfalls

Verlauf protokollieren

- ▶ **Idee:** Vergleiche Sollsequenz gegen eine aufgezeichnete Sequenz
- ▶ Aufzeichnen der Sequenz:

```
const char* right_sequence = "abcd"; // Sollsequenz
      char sequence[4];           // aufgezeichnete Sequenz
volatile int counter = 0;        // wohin kommt der Buchstabe
                                   // in der Sequenz

...
sequence[counter++] = 'a';        // Sequenz aufzeichnen
...
if (!check_sequence(right_sequence, sequence)) { // Sequenz
    Panic("Wrong sequence!");                // ueberpruefen
}

TestOK("Test successful finished!");
```

Verlauf protokollieren (Forts.)

- ▶ **Achtung:** Jede Abweichung vom vorgesehenen Ablauf muss sich in der Sequenz bemerkbar machen:

```
const char* right_sequence = "abcd"; // Sollsequenz
char sequence[4]; // aufgezeichnete Sequenz
void MyThread1::action() {
    sequence[counter++] = 'a'; // Sequenz aufzeichnen
    ...
    scheduler.exit();
    // dieses Statement sollte nie erreicht werden
    sequence[counter++] = 'x';
}
```

- ▶ ...oder anderweitig erkannt werden:

```
void MyThread1::action() {
    ...
    scheduler.exit();
    Panic("Should not come here!");
}
```

Unterbrechungen synchron/asynchron auslösen

- ▶ Klassen `Test_Interrupt_0` und `Test_Interrupt_1`

Klasse `Test_Interrupt`

```
template< class BASE > class Test_Int
    : public BASE
{
public:
    void trigger();           // synchron
    void trigger_async();    // asynchron
    void acknowledge();
};

typedef Test_Int<Device_0> Test_Interrupt_0;
typedef Test_Int<Device_1> Test_Interrupt_1;
```

- ▶ Für `void trigger_async()` muss die Klasse `BASE` eine Methode `bool is_triggered()` implementieren.

Unterbrechungen - Verwendung

```
# include "debug/test_interrupt_0.h"

class My_Interrupt : public Test_Interrupt_0 {
public:
    bool isr() { acknowledge(); /* ... */ }
    void dsr() { /* ... */ }
};

My_Interrupt my_int;

void Test_Thread::action() {
    // ...
    my_int.trigger();
    // ...
}
```


Interrupt_Timer

- ▶ gemeinsame Schnittstelle für Testfälle
- ▶ Klasse `Test_Interrupt_Timer_0`

```
typedef my_timer Test_Interrupt_Timer_0 ;
```

- ▶ Verwendung beispielsweise für Hardware-Counter

```
#include "debug/interrupt_test_timer_0.h"
```

```
Hardware_Counter< Test_Interrupt_Timer_0 >  
    my_hw_counter(10000);
```

Den Testfall im Debugger (DDD) ausführen

- ▶ Debugger starten und den Testfall laden:

```
make tests/thread/Scheduler/RoundRobin/test1/test1.sim_ddd
```

- ▶ die Ausführung des Testfalls starten:

- ▶ *Cont*-Button klicken
- ▶ auf der GDB-Konsole folgendes eingeben:

```
(gdb) cont
```

Achtung

Der *Run*-Button bzw.

```
(gdb) run
```

wird **nicht** funktionieren!

Den Testfall neu starten ohne den DDD zu beenden

- ▶ zunächst muss man die Verbindung zum GDB-Stub trennen

```
(gdb) detach
```

- ▶ erneut eine Verbindung aufbauen (<PORT> steht in der Ausgabe von make)

```
(gdb) target remote localhost:<PORT>
```

- ▶ das Testfallprogramm laden

```
load
```

- ▶ das hat einige Vorteile:
 - ▶ Breakpoints und
 - ▶ Watches bleiben erhalten

Debugger neu starten

Manchmal kann es vorkommen, dass der GDB-Stub bzw. der deSmuME sich komisch verhält. Man hat dann drei Möglichkeiten:

1. DDD beenden und den Testfall erneut laden

- ▶ Breakpoints und Watches gehen verloren

2. nur den GDB beenden

- ▶ den `arm-elf-gdb` beenden

```
killall arm-elf-gdb
```

- ▶ DDD wird das bemerken und folgender Dialog erscheint

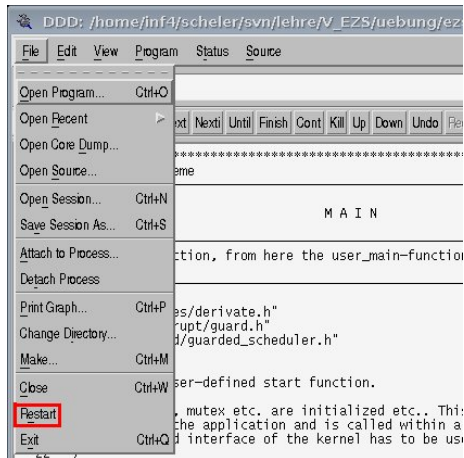


- ▶ auf den *“Restart GDB”*-Button klicken

Debugger neu starten

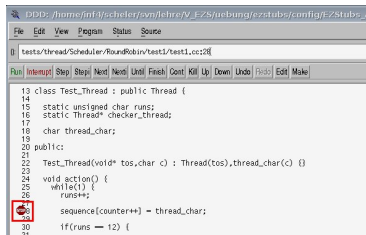
3. den kompletten DDD neu starten

- ▶ aus dem Menü *File* die Option *Restart* auswählen



Breakpoints setzen ...

- ... durch einen Doppelklick ins Quelltextfenster



- ... auf der GDB-Konsole für eine bestimmte Funktion

```
(gdb) break <function_name>  
(gdb) break <file_name>:<line >
```

Achtung

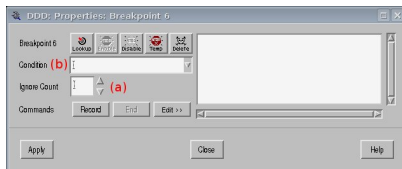
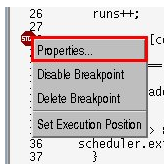
Auf eingebettete Funktionen kann man keinen Breakpoint setzen!

Bedingungen für Breakpoints

Bevor das Programm an einem Breakpoint tatsächlich angehalten wird, kann man ...

- (a) ... den Breakpoint n-mal ignorieren
- (b) ... auf die Erfüllung einer Bedingung warten.

► aus dem Quelltextfenster heraus



► auf der GDB-Konsole

```
(gdb) ignore <breakpoint_nr> <n> # (a)
(gdb) condition <breakpoint_nr> runs == 7 # (b)
```

Bedingungen für Breakpoints (Forts.)

Achtung

bedingte Breakpoints können **Ein-/Ausgabe-Operationen** zur Folge haben, die das **Laufzeitverhalten** des Programms beeinflussen können.

Sinnvolle Stellen für Breakpoints

- ▶ Warum wird mein DSR nicht ausgeführt? → Breakpoint auf ...
 1. `...wrapper_body` → Wurde ein IRQ ausgelöst?
 2. `...void guardian(unsigned int)` → Welches Gerät?
 3. `...den_isr()` → Wurde ein DSR aktiviert?
 4. `...void Guard::leave()` → Welchen Wert hat `lock`?
- ▶ Warum wird der Scheduler nicht aktiviert? → Breakpoint auf ...
 1. `...void Guard::leave()` → Welchen Wert hat `lock`?
 2. `...reschedule()` → Wurde der Scheduler aktiviert?
 3. `...Thread* schedule()` → Welcher Faden kommt als nächstes?

Achtung

1. Ein Debugger ersetzt nicht das Verständnis des Programms!
2. Ohne sinnvolle Annahmen kann man keine Breakpoints setzen!

Weitere Möglichkeiten, das Programm zu kontrollieren

► Break

- Anhalten des Programms “*von außen*”.
- normalerweise per `Ctrl-C` in der GDB-Konsole

► Watchpoints

```
(gdb) watch xyz
```

- Programm anhalten wenn Variable `xyz` ihren Wert ändert

Werte von Variablen beobachten

► globale Variablen und Objekte

```
3 class Test_Thread : public Thread {
4
5     static unsigned char runs;
6     static Thread* checker;
7
8     char thread_char;
9
10 public:
11     Test_Thread(void* tos, c
12
13     void action() {
14         while(1) {
15             runs++;
16
17             sequence[counter++]
18
19             if(runs == 12) {
20
21
22     scheduler.add(checker, thread);
```

Print runs
Display runs
Print *runs
Display *runs
What is run
Lookup runs
Break at run
Clear at run

(gdb) # Achtung: voll qualifizierter Name!
(gdb) graph display Test_Thread::runs

► lokale Variablen und Argumente

► Menü *Data* →

► *Display Local Variables*

► *Display Arguments*

► Beobachtung problematisch:

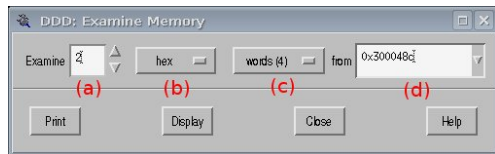
► Speicherstellen/Register werden vom GCC wiederverwendet

► passende Debug-Information wird (noch) nicht erzeugt

🔗 **Var Tracking Assignments [VTA]** ab GCC 4.5

Speicherstellen beobachten

- ▶ Menü *Data* →
 - ▶ *Memory*



- ▶ Parameter
 - (a) Wie viele Bytes, Half Words, Words, ...
 - (b) Hexadezimal, Oktal, ...
 - (c) Bytes, Half Words (2 Bytes), Words (4 bytes), ...
 - (d) Startadresse
- ▶ Hilfreich zum Kontrollieren von I/O-Registern

Tutorial

1. die Datei `tutorial.tar.gz` entpacken
2. Makefiles anpassen
3. EZStubs-Bibliothek und Testfälle übersetzen und binden
4. Doxygen-Dokumentation erzeugen
5. Testfälle auf dem Simulator ausführen
6. Testfälle mit Hilfe des DDD debuggen
 - ▶ Breakpoints setzen (siehe Folie 49)
 - ▶ dort das Programm schrittweise ausführen
 - ▶ und die Abläufe auf den Folien 11, 19, 20 und 24 nachvollziehen
7. Selbst ein, zwei Testfälle schreiben
 - ▶ einen Interrupt-Handler implementieren (die Klasse `Test_Interrupt_0` steht zur Verfügung).
 - ▶ im Interrupt-Handler einen Faden aktivieren