H.5 Nebenläufigkeit

1 Überblick

- Definition von Nebenläufigkeit:
  zwei Programmausführungen sind nebenläufig, wenn für zwei einzelne Befehle a und b aus beiden Ausführungen nicht feststeht, ob a oder b tatsächlich zuerst ausgeführt wird

- Nebenläufigkeit tritt auf
  - ➤ bei Interrupts
  - ➤ bei parallelen Abläufen (gleichzeitige Ausführung von Code in einem Mehrprozessorsystem mit Zugriff auf den gleichen Speicher)
  - ➤ bei quasi-parallelen Abläufen (wenn ein Betriebssystem verschiedenen Prozesse den Prozessor jeweils für einen Zeitraum zuteilt und ihn nach Ablauf der Zeit wieder entzieht)

- Problem:
  - ➤ was passiert, wenn die nebenläufigen Ausführungen auf die gleichen Daten im Speicher zugreifen?

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Nebenläufigkeit durch Interrupts

- Interrupts unterbrechen Anwendungsprogramme "irgendwo"
- Interrupts haben Zugriff auf den gleichen Speicher
- Szenario:
  - ➤ eine Lichtschranke soll Fahrzeuge zählen und alle 10 Sekunden soll der Wert ausgegeben werden

```
int a;

  int main() {
  long i;
  while(1) {
    for (i=0; i<2000000; i++)
    /* Zählen dauert 10 Sek. */;
    print(a);
    a=0;
  }
}
```

```
/* Lichtschranken-
   Interrupt */
int count() {
        a++;
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Nebenläufigkeit durch Interrupts (2)

- Auf C-Ebene führt die Interrupt-Behandlung nur einen Befehl aus: a++
  - ➤ nur scheinbar ein Befehl
  - ➤ auf Maschinencode-Ebene (Bsp. AVR) sieht die Sache anders aus

```
...
    print(a);
    a=0;
...
```

```
int count() {
        a++;
}
```

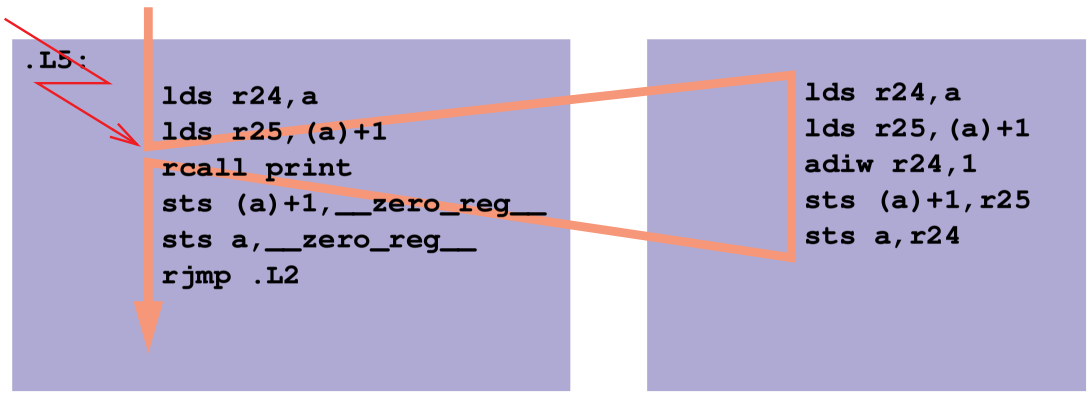
```
...
.L5:
        lds r24,a
        lds r25,(a)+1
        rcall print
        sts (a)+1,__zero_reg__
        sts a,__zero_reg__
        rjmp .L2
...
```

```
...
        lds r24,a
        lds r25,(a)+1
        adiw r24,1
        sts (a)+1,r25
        sts a,r24
...
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Nebenläufigkeit durch Interrupts (3)

- Annahme: Interrupt trifft folgendermaßen ein:



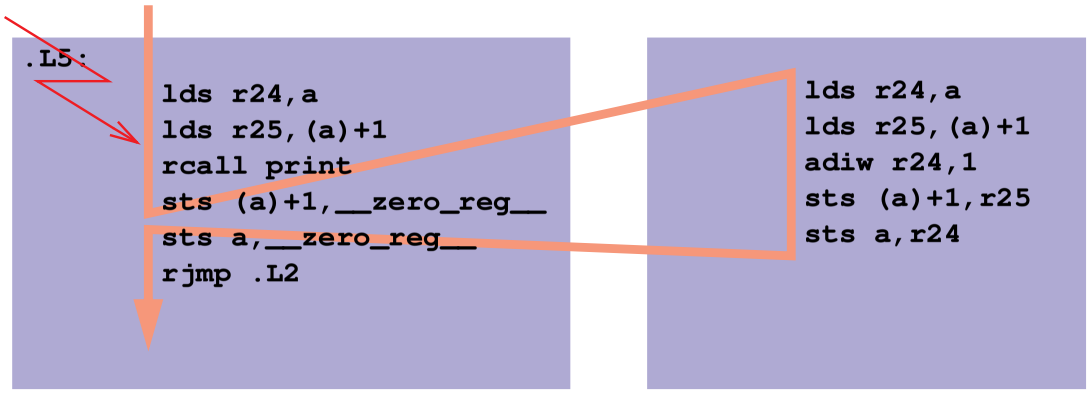
  - ➤ Folge: ein Fahrzeug wird nicht gezählt

- Details des Szenarios zeigen mehrere Problemstellen:
  - ➤ int-Wert wird in zwei Schritten in zwei Register geladen (long: 4 Register)
  - ➤ Operationen erfolgen in Registern, danach wird in Speicher zurückgeschrieben

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Nebenläufigkeit durch Interrupts (4)

- Annahme: Interrupt trifft folgendermaßen ein:



➤ Folge: möglicherweise werden 255 Fahrzeuge zuviel gezählt

- Variable a ist auf 2 Register verteilt →a = 0 nicht atomar zuerst wird obere Hälfte auf 0 gesetzt
- falls a++ im Interrupt-Handler a zufällig von 255 auf 256 zählt → Bitüberlauf vom "unteren" in's "obere" Register
- nach Interrupt wird nur noch untere Hälfte auf 0 gesetzt → a = 256

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Nebenläufigkeit durch Interrupts (5)

- weiteres Problem bei Zugriff auf globale Variablen:
  - AVR stellt 32 Register zur Vefügung
  - Compiler optimiert Code und vermeidet Speicherzugriffe wenn möglich
    - ➤ Variablen werden möglichst in Registern gehalten
  - Registerinhalte werden bei Interrupt gesichert und am Ende restauriert
    - ➤ Änderungen der Interrupt-Funktion an einer Variablen gehen beim Restaurieren der Register wieder verloren

- Lösung für dieses Problem:
  - Compiler muss Variablen vor jedem Zugriff aus dem Speicher laden und anschließend zurückschreiben
    - ➤ Attribut `volatile`

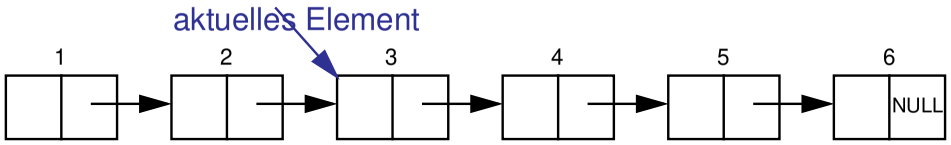
```
volatile int a;
```

  - Nachteil: Code wird umfangreicher und langsamer
    - ➤ nur einsetzen wo unbedingt notwendig!

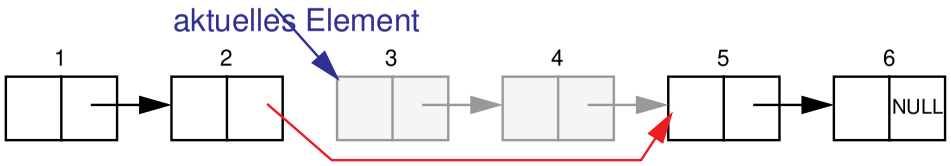
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Nebenläufigkeitsprobleme allgemein

- Zugriff auf gemeinsame Daten ist bei nebenläufigen Ausführungen generell kritisch
  - ➤ selbst bei einfachen Variablen (siehe vorheriges Beispiel)
  - ➤ Problem bei komplexeren Datenstrukturen
    (z. B. Einketten einer Struktur in verkettete Liste)
    noch gravierender: Datenstruktur kann völlig zerstört werden
- Beispiel: Programm läuft durch eine verkettete Liste



◆ Interrupt-Handler oder parallel laufendes Programm entfernt Elemente 3 und 4 und gibt den Speicher dieser Elemente frei



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Umgang mit Nebenläufigkeitsproblemen

- Gemeinsame Daten möglichst vermeiden
  - ➤ Interrupt-Funktionen sollten weitgehend auf eigenen Daten arbeiten
  - ➤ Parallele Abläufe sollten ebenfalls möglichst eigene Datenbereiche haben
- Kommunikation zwischen Anwendungsabläufen erfordert aber oft gemeinsame Daten
  - ➤ solche Daten sollten deutlich hervorgehoben werden
    z. B. durch entsprechenden Namen

```
volatile int INT_zaehler;
```

    - betrifft nur globale Variablen
    - lokale Variablen sind unkritisch
      (nur in der jeweiligen Funktion sichtbar)
  - ➤ Zugriff auf solche Daten sollte in der Anwendung möglichst begrenzt sein
    (z. B. nur in bestimmten Funktionen,
    gemeinsames Modul mit Interrupt-Handlern, vgl. Kap. D.9-3)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Umgang mit Nebenläufigkeitsproblemen (2)

- Zugriffskonflikte mit Interrupt-Handlern verhindern
  - das Programm muss vor kritischen Zugriffen auf gemeinsame Daten Interrupts sperren
    - ➤ Beispiel AVR:
      Funktionen `cli()` (blockiert alle Interrupts)
      und `sei()` (erlaubt Interrupts)
  - in dieser Zeit können Interrupts verloren gehen

  ➥ Zeitraum von Interruptsperren muss möglichst kurz bleiben!

  - es kann sinnvoll sein, nur Interrupts des Geräts zu sperren, dessen Handler auch auf die kritischen Daten zugreift
    (hängt von Details der Hardware ab!)
- Zugriffskonflikte bei parallelen Abläufen
  - spezielle atomare Maschinenbefehle
    (z. B. test-and-set oder compare-and-swap bei Intel-Architekturen)
  - Software-Synchronisation (lock-Variablen, Semaphore, etc.)
  - Kommunikation mittels Nachrichten statt gemeinsamer Daten

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

H.6 Programmierung eingebetteter Systeme

- Vorteile von C in eingebetteten Systemen:
  - ➤ leichter Zugriff auf die Hardware
  - ➤ gute Kontrolle über die verwendeten Ressourcen (Speicher, CPU)
    - ➥ effizienter Code
    - ➥ geringer Speicherbedarf
- Probleme mit C:
  - ➤ stark eingeschränkte Laufzeitüberprüfungen
  - ➤ ISO C-Standard oft nicht eindeutig
    Verhalten ist implementationsabhängig oder undefiniert

  ➥ in C kann man viele Fehler machen (v. a. Anfänger)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 MISRA C

- 1998 von der MISRA entwickelt
  (MISRA = Motor Industry Software Reliability Association)
- Richtlinien für den Gebrauch von C in sicherheitskritischen Systemen (1998, 2004 überarbeitet)
- Regeln, die den Umgang mit bestimmten Konstrukten in C präzisieren z. B. bei Zeigeroperationen
  - ➤ 141 Regeln (121 verbindlich, 20 empfohlen)
  - ◆ Beispiele
    - ➤ Anweisungen nach if, while, ... müssen immer in { } geklammert sein
    - ➤ spezielle Typen (z.B. uint32_t) statt der Standardtypen
- ursprünglich Automobilbereich, seit 2004 neue verallgemeinerte Version

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Java für eingebettete Systeme?

- mögliche Vorteile:
  - ➤ objektorientiert
  - ➤ typsicher

3 Objektorientierte Programmierung

- Zusammenfassung von Funktionen und ihren Daten
  - ➥ Strukturierung
- Vorteile / Nachteile
  - \+ Zusammengehörendes kommt zusammen
  - \+ kaum noch globale Variablen
  - \+ höhere Abstraktion möglich
  - – weniger Kontrolle über Ressourcenverbrauch
  - ➥ Werkzeuge müssen für eingebettete Systeme optimiert sein

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Typsichere Sprachen

- Beispiele: Java, .NET-Sprachen (z. B. C#)
- Grundprinzip:
  - ➤ allen Werten ist immer ein Typ zugeordnet
  - ➤ Umwandlung in einen anderen Typ ist nicht ohne Weiteres möglich
- Umwandlung muss möglicherweise zur Laufzeit geprüft werden
  - ➥ keine Zeigeroperationen
- Automatische Speicherbereinigung möglich
- Typsystem zur Laufzeit nötig
  - ➥ meist interpretiert

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Typsichere Sprachen (2)

- Vorteile / Nachteile:
  - – mehr Ressourcenbedarf durch Laufzeitüberprüfungen
  - – schlechtes Laufzeitverhalten bei Interpretation
  - \+ Interpretation ermöglicht leichte Erweiterbarkeit zur Laufzeit (dynamisches Nachladen von Klassen)
  - \+ Typsicherheit ermöglicht die Kapselung der Anwendung ohne Hardwareunterstützung (vertrauenswürdiger Compiler!)
- Für tief eingebettete Systeme ist Ressourcenbedarf wichtig
  - ➤ starke Einschränkung der dynamischen Erweiterbarkeit
  - ➤ compilieren zu statischem Programm

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Java für eingebettete Systeme

- Java Micro-Edition (J2ME):
  schlanke JVM und angepasste Klassenbibliothek
  - ➤ als J2ME/CLDC-MIDP auf den meisten Handys
  - ➤ als J2ME/CDC auf einigen PDAs
- JavaCard: stark abgespeckte JVM
  - ➤ für Chipkarten

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6 Modellbasiertes Design

- MDA: Model Driven Architecture
- Beschreibung der Anwendung in Form eines Modells
  auf einer abstrakten Ebene
- Konkretes Programm wird von einem Werkzeug erstellt
- Vorteile/Nachteile
  - \+ architekturunabhängig
  - \+ Konzentration auf die Funktionalität der Anwendung
  - +/– Detailwissen zur Implementierung in den Werkzeugen
  - – Werkzeug für die Zielplattform wird benötigt
- Beispiel: Labview, Simulink

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.