Nebenläufigkeit

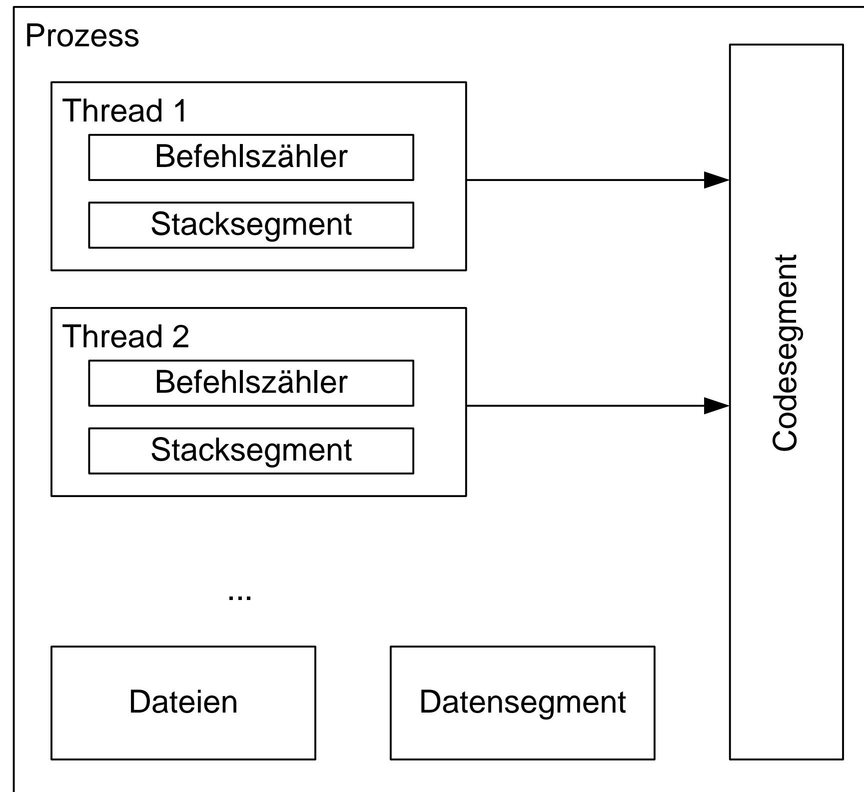
Threads

Leichtgewichtige Prozesse (Threads)

- Der Speicherbedarf von Prozessen ist in der Regel groß (CPU-Daten, Statusinformationen, Angaben zu Dateien und EA-Geräten...).
- Bei Prozesswechsel müssen die Prozessdaten ausgetauscht werden) hohe Systemlast, zeitaufwendig.
- Viele Systeme erfordern keine komplett neuen Prozesse.
- Vielmehr sind Programmabläufe nötig, die auf den gleichen Prozessdaten arbeiten.

⇨ Einführung von Threads

Threads



Prozesse vs. Threads

- Verwaltungsaufwand von Threads ist deutlich geringer
- Effizienzvorteil: bei einem Wechsel von Threads im gleichen Prozessraum ist kein vollständiger Austausch des Prozesskontextes notwendig.
- Kommunikation zwischen Threads des gleichen Prozesses kann über gemeinsamen Speicher erfolgen.
- Zugriffe auf den Speicherbereich anderer Prozesse führen zu Fehlern.
- Probleme bei Threads: durch die gemeinsame Nutzung von Daten kann es zu Konflikten kommen.

Nebenläufigkeit

Unterbrechungen

Binding Rechnersystem-Umwelt

- Es muss ein Mechanismus gefunden werden, der es erlaubt, Änderungen der Umgebung (z.B. Druck einer Taste) zu registrieren.
- **1. Ansatz**: Abfrage (Polling)
  Es werden die E/A-Register reihum nach Änderungen abgefragt und bei Änderungen spezielle Antwortprogramme ausgeführt.
  - Vorteile:
    - bei wenigen EA-Registern sehr kurze Latenzzeiten
    - bei einer unerwarteten Ereignisflut wird das Zeitverhalten des Programms nicht übermäßig beeinflusst
    - Kommunikation erfolgt synchron mit der Programmausführung
  - Nachteile:
    - die meisten Anfragen sind unnötig
    - hohe Prozessorbelastung
    - Reaktionszeit steigt mit der Anzahl an Ereignisquellen

Lösung: Einführung des Begriffs der Unterbrechung

- **2. Ansatz**: Unterbrechung (Interrupt)
- Eine Unterbrechung stoppt die Verarbeitung des laufenden Programms. Die Wichtigkeit des Ereignisses, welches die Unterbrechung ausgelöst hat, wird überprüft. Darauf basierend erfolgt die Entscheidung, welche Reaktion erfolgt.
- Vorteile:
  - Prozessorressourcen werden nur dann beansprucht, wenn es nötig ist
- Nachteile:
  - Nicht-Determinismus: Unterbrechungen asynchron zum Programmlauf (und zum Prozessorzustand) eintreffen.

Unterbrechungen

- **Unterbrechungen**: Stopp des Hauptprogrammablaufs, Aufnahme der Programmausführung eines „Unterbrechungsbehandlers (UBB)“ an einer anderen Stelle; nach Beendigung des UBB (zumeist) Rückkehr an die Stelle des Auftritts der Unterbrechung im Hauptprogramm.
- **Synchrone** Unterbrechungen: treten, falls sie auftreten, immer an *derselben Stelle* im Programmcode auf. Man bezeichnet sie auch als *Traps* oder *Exceptions* bzw. „Software-Interrupts“
- **Asynchrone** Unterbrechungen: Auftrittszeitpunkt ist unbestimmt; es kann nicht gesagt werden, an welcher Stelle der Hauptprogrammausführung der Prozessor zum Zeitpunkt der Unterbrechung ist. Asynchrone Unterbrechungen werden auch als Interrupts bezeichnet; weil sie von der Hardware-Peripherie erzeugt werden, auch als *Hardware-Interrupts*. Sie üben „Brückenfunktion“ zwischen Hardware und Software aus.

Synchrone Unterbrechungen (Traps/Exceptions)

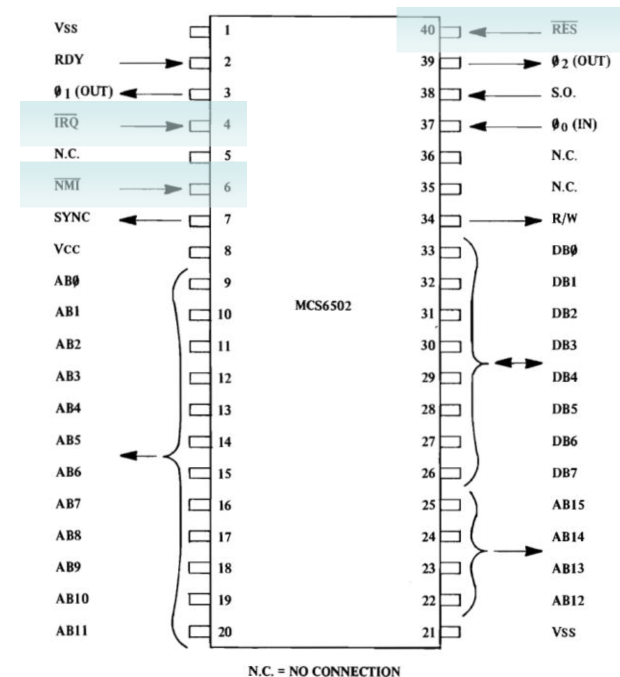
- Werden durch das Programm selbst ausgelöst, d.h. dasselbe Programm, ausgeführt mit denselben Parametern wird in der Regel an derselben Stelle dieselbe Unterbrechung auslösen (vorhersagbar in dieselbe „Falle“ laufen)
- **Auslösung bei Fehler** – Ausnahme/Exception, Beispiele:
  - Arithmetikfehler (Division by zero, overflow, not-a-number NaN, …)
  - Speicherfehler (Page Fault, segment Fault, memory full, …)
  - Befehlsfehler (Illegal instruction, privileged instruction, bus error, …)
  - Peripheriefehler (End-of-file EOF, channel blocked, unknown device, …)
- Bei Exceptions **nur dann** Rückkehr an den Auftrittspunkt, wenn die Fehlerbedingung im Ausnahmebehandler beseitigt werden kann, andernfalls Abbruch (resumption vs. termination)
- **Auslösung durch spezifische Instruktion**: Breakpoint, SWI, TRAP, INT, … entweder zum Zwecke des „Debuggings“ oder zum Aufruf von Betriebssystem-Diensten (z.B. MS-DOS „INT 21h“, siehe z.B. http://en.wikipedia.org/wiki/MS-DOS_API)
- Traps können auch benutzt werden, um einen Hardware-Interrupthandler zu testen.

Asynchrone Unterbrechungen (Interrupts)

- Werden durch externe Prozesse ausgelöst, d.h. sind bezüglich des genauen Auftrittszeitpunkts unvorhersagbar und zumeist nicht reproduzierbar
- Beispiele:
  - Signalisierung „normaler“ externer Ereignisse durch periphere Einheiten (Timer, Schalter, Grenzwertüber-/unterschreitung, …)
  - Warnsignale der Hardware (Energiemangel, „Watchdog-timer“ abgelaufen, …)
  - Beendigung einer Ein-/Ausgabeoperation (Wort von serieller Schnittstelle komplett empfangen oder komplett gesendet, Operation von Coprozessor (DMA, FPU) komplett, …)
- Die Unterbrechungsbehandlung muss *nebeneffektfrei* verlaufen, d.h. das Hauptprogramm darf nach Abschluss der Behandlung keinen (ungewünschten) veränderten Ausführungs-Kontext vorfinden.
- Aber: typischerweise wird der Behandler zum Zwecke der Kommunikation über globale Variable mit dem Hauptprogramm kommunizieren.

Technische Realisierung von Interrupts

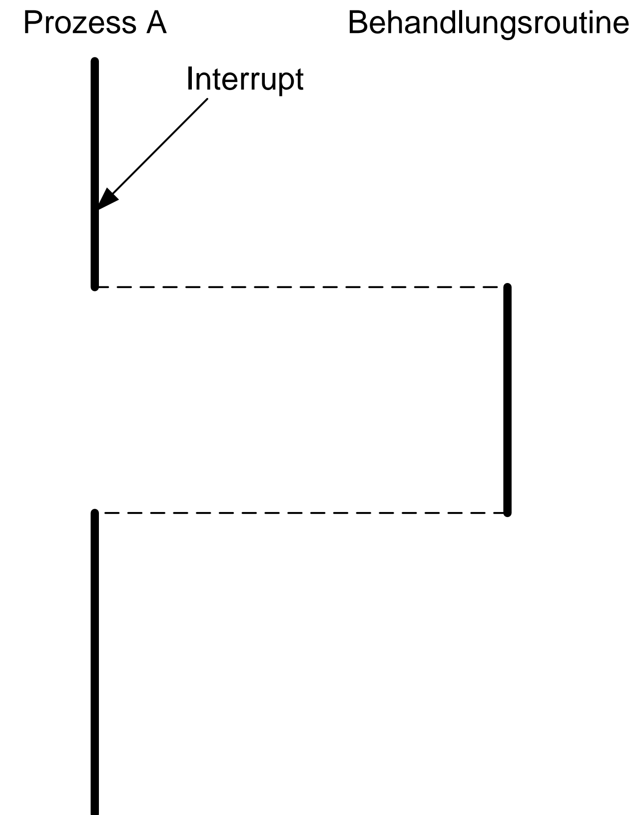
- Zur Realisierung besitzen Prozessoren einen oder mehrere spezielle Interrupt-Eingänge (typ. IRQ oder INT-Anschluß). Wird ein Interrupt aktiviert, so führt dies zur Ausführung einer Unterbrechungsbehandlungsroutine.
- Das Auslösen der Unterbrechungsroutine entspricht einem Unterprogrammaufruf. Der Programmablauf wird an einer anderen Stelle fortgesetzt und nach Beendigung der Routine (normalerweise) an der unterbrochenen Stelle fortgefahren.



Pin-Belegung des MOS 6502-Prozessors (z.B. Commodore 64).
Quelle: MCS 6500 HW-Manual, MOS Technology, Jan. 1976

Durchführung einer einfachen INT-Behandlung

1. IRQ-Anschluß wird durch peripheres Gerät aktiviert
2. Wenn Interrupts momentan zugelassen sind: Beendigung der Abarbeitung der gerade noch laufenden Instruktion
3. Sperren von Unterbrechungen mit gleicher oder niedrigerer Priorität
4. Sicherung der Register des Prozessors (Prozessorkontext) auf dem Stapelspeicher (Stack) – insbesondere Sicherung des Programmzählers; dafür spezielle Instruktionen verfügbar
5. Sprung an den Behandler (entspricht Laden des Programmzählers mit der Programmstartadresse des Behandlers) – dies kann auf verschiedene Arten erfolgen, siehe unten
6. Ausführung des Codes des Behandlers, an dessen Ende steht ein „Return from Interrupt“-Befehl; dabei Signalisierung an Peripherie, dass Behandlung abgeschlossen
7. Rückkehr zur Programmausführung (nicht unbedingt der unterbrochene Prozess) durch Zurückladen des gesicherten Prozessorregistersatzes.



Beispiel INT beim 6502

- Bei Vorliegen eines IRQ wird der 16-Bit Programmzähler mit dem Inhalt der 8-Bit Adressen FFFE und FFFF geladen.
- FFFE/F enthält Adresse des (= *Vektor* auf) IRQ-Behandler
- FFFC/D enthält Adresse des Reset-Behandlers
- FFFA/B enthält Adresse des NMI-Behandlers. NMI: Non-Maskable-Interrupt, kann nicht abgeschaltet werden (also auch nicht durch fehlerhaftes Programm)

Speicher-Aufteilung für 6502 (in Hex-Adressen)

0000-00FF - RAM for Zero-Page & Indirect-Memory Addressing

0100-01FF - RAM for Stack Space & Absolute Addressing

0200-3FFF - RAM for programmer use

4000-7FFF - Memory mapped I/O

8000-FFF9 - ROM for programmer usage

FFFA - Vector address for NMI (low byte)

FFFB - Vector address for NMI (high byte)

FFFC - Vector address for RESET (low byte)

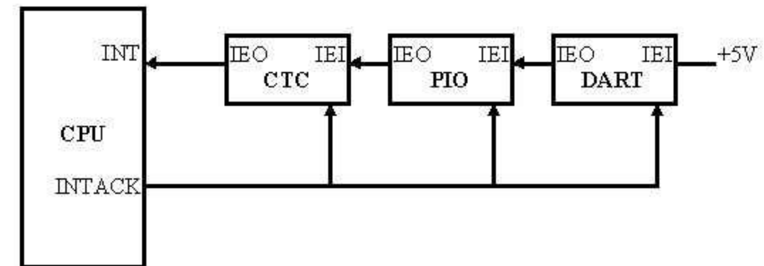
FFFD - Vector address for RESET (high byte)

FFFE - Vector address for IRQ & BRK (low byte)

FFFF - Vector address for IRQ & BRK (high byte)

Behandlung mehrerer Quellen von Interrupts

- Beim 6502 nur *ein* IRQ-Eingang und nur *ein* Vektor auf *einen* Behandler. Bei mehreren Quellen für Interrupts (z.B. serielle Schnittstelle, Parallele Schnittstelle, Timer) werden daher die Interrupt-Ausgänge dieser Einheiten „verodert" und an den IRQ-Eingang gelegt. Alle Quellen sind damit gleichberechtigt.
- Damit ist keine automatische HW-Priorisierung (nach Wichtigkeit der eintreffenden Interrupts) möglich.
- Der Behandler muß nach dem Auftritt des Interrupts Quelle für Quelle abfragen, welche den Interrupt verursacht hat (implizite Priorisierung je nach Abfragereihenfolge)
- Einfache Möglichkeit der HW-Priorisierung: Daisy-Chain (Prioritätskette). Die Einheit, die am nächsten an der CPU liegt, hat die höchste Priorität (siehe Bild rechts) und sperrt Interrupts anderer Quellen solange aus, bis der eigene Interrupt abgearbeitet ist (INTACK durch Prozessor signalisiert).



Daisy-Chaining beim Z80-Prozessor
CTC: Counter-Timer-Circuit
PIO: Parallel In-/Out
DART: Dual Asynchronous Receiver Transmitter

Vektorisierte Interrupt-Behandlung

- Verfügt ein Rechnersystem über viele Interrupt-Quellen (wie typischerweise im Bereich eingebetteter Systeme), ist es zweckmäßig, für jedes Gerät (mindestens) einen Behandler vorzusehen und diesen nach dem Ereignisauftritt auch direkt ausführen zu können.
- Dazu Einführung von vektorbasierten Interrupt-Systemen.
- Prinzip: Gerät erzeugt nicht nur einen Interrupt, sondern liefert dem Prozessor parallel auch eine eigene Kennung (z.B. 8-Bit Wert A). Dieser Wert A verweist auf einen Tabelleneintrag, an dem sich die Einsprung-Adresse des Behandlers befindet.
- Exceptions und Interrupts folgen dem gleichen Schema, d.h. Tabelleneintrag = Einsprungadresse für Exception/Interrupt-Handler

Address	Entry	No.
080H	32-255 User defined	
	14-31 Reserved	
040H	Coprocessor error	16
03CH	Unassigned	15
038H	Page fault	14
034H	General protection	13
030H	Stack seg overrun	12
02CH	Segment not present	11
028H	Invalid task state seg	10
024H	Coproc seg overrun	9
020H	Double fault	8
01CH	Coprocessor not avail	7
018H	Undefined Opcode	6
014H	Bound	5
010H	Overflow (INTO)	4
00CH	1-byte breakpoint	3
008H	NMI pin	2
004H	Single-step	1
000H	Divide error	0

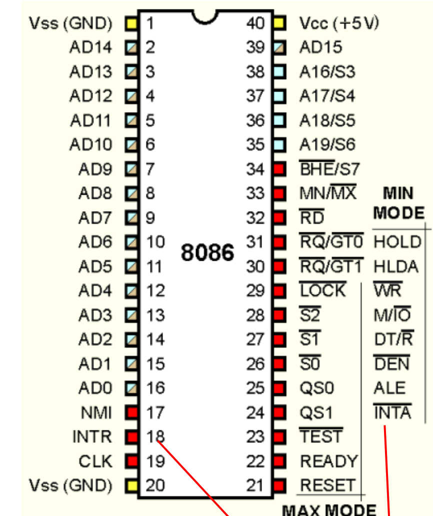
The interrupt vector table is located in the first 1024 bytes of memory at addresses 000000H through 0003FFH.

There are 256 4-byte entries (segment and offset in real mode).

Seg high	Seg low	Offset high	Offset low
Byte 3	Byte 2	Byte 1	Byte 0

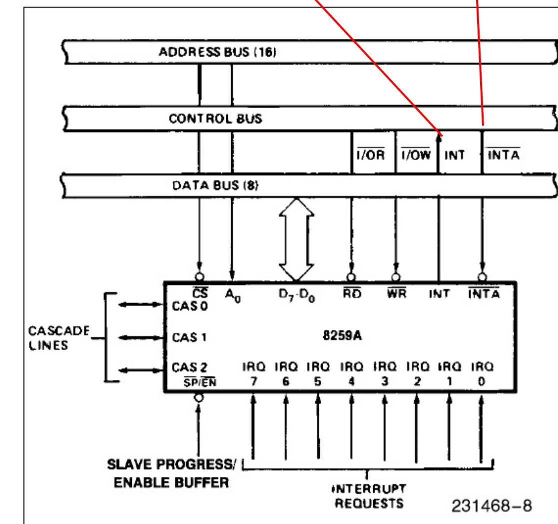
- Typische Interrupt-Vektor-Tabelle eines 8086-Systems im „Real Mode"
- 32-Bit-Adresse (4 Byte) pro Eintrag
- 0 bis 31 sind Prozessor-interne Ausnahmen
- Hardware-Interrupts können über Interrupt-Controller auf beliebige Tabellenplätze gelegt werden

Interrupt-(Priority)-Controller



- Typischerweise wird mit der vektorbasierten Verwaltung auch eine Prioritätsverwaltung eingeführt. Klassisches Beispiel: der Interrupt-Controller 8259A des IBM-PC.
- Verwaltet 8 Hardware-Interrupts und kann diese mit Prioritäten belegen. Ist kaskadierbar, der PC/AT hatte zwei 8259A.
- Typen der Interrupts beim PC/AT:

00	Systemtaktgeber	08	Echtzeitsystemuhr
01	Tastatur	09	Frei
02	Programmierbarer Interrupt-Controller	10	Frei
03	Serielle Schnittstelle COM2 (E/A-Bereich 02F8)	11	Frei
04	Serielle Schnittstelle COM1 (E/A-Bereich 03F8)	12	PS/2-Mausanschluss
05	Frei, oft Soundkarte oder LPT2	13	Koprozessor (ob separat oder in CPU integriert)
06	Diskettenlaufwerk	14	Primärer IDE-Kanal
07	Parallel (Drucker-) Schnittstelle LPT1 (E/A-Bereich 0378)	15	Sekundärer IDE-Kanal



- Heute Standard (seit ca. 2002): Nachfolgekonzept, Intel APCI 82093AA, sehr viel höhere Anzahl von Interrupts realisiert

Nebenläufigkeit

Probleme

Probleme

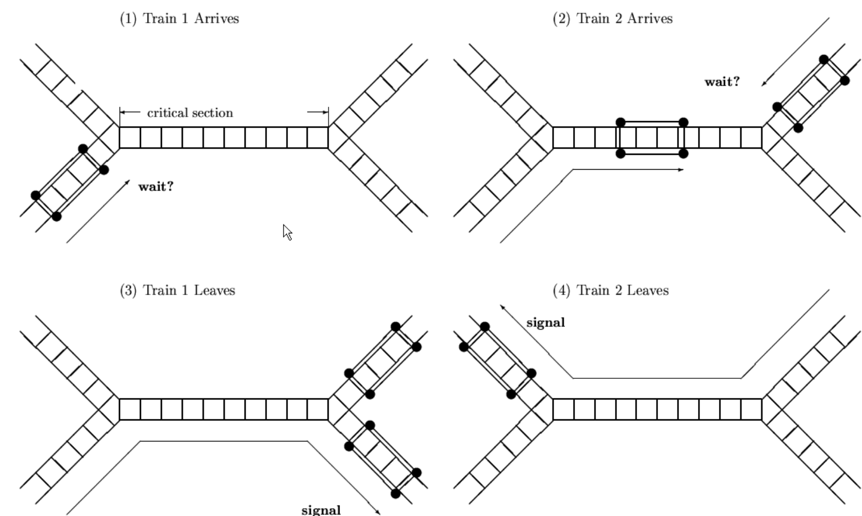
- **Race Conditions**:
  - Situationen, in denen zwei oder mehrere Threads/Prozesse, die gleichen geteilten Daten lesen oder schreiben und das Resultat davon abhängt, wann genau welcher Prozess ausgeführt wurde, werden Race Conditions genannt.
  - Lösung: Einführung von **kritischen Bereichen** und **wechselseitiger Ausschluss**.
- **Starvation** (**Aussperrung**):
  - Situation, in der ein Prozess unendlich lange auf ein Betriebsmittel wartet. Wichtig: sinnvolle Realisierung von Warteschlangen bei der Betriebsmittelvergabe, z.B. Prioritätenbasierte Warteschlangen
- **Priority Inversion** (**Prioritätsinversion**):
  - Wichtige Prozesse können durch unwichtigere Prozesse, die Betriebsmittel belegt haben verzögert werden, genaue Problemstellung siehe Kapitel Scheduling

Bedingungen an Lösung für wechselseitigen Ausschluss

- An eine gute Lösung für den wechselseitigen Ausschluss (WA) können insgesamt vier Bedingungen gestellt werden:
  1. Es dürfen niemals zwei Prozesse gleichzeitig im kritischen Bereich sein.
  2. Es dürfen keine Annahmen über die Geschwindigkeit oder Anzahl der Prozessoren gemacht werden.
  3. Kein Prozess darf außerhalb von kritischen Regionen andere Prozesse blockieren.
  4. Kein Prozess soll unendlich auf das Eintreten in den kritischen Bereich warten müssen.

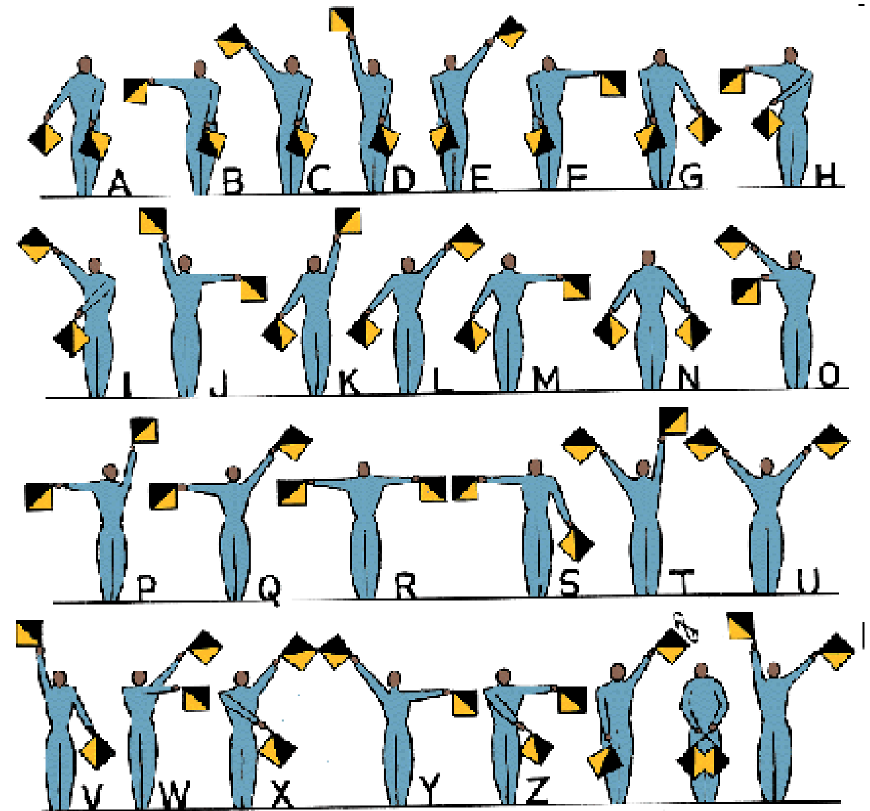
Kritische Bereiche

- Um einen kritischen Bereich zu schützen, sind Mechanismen erforderlich, die ein gleichzeitiges Betreten verschiedener Prozesse bzw. Prozeßklassen dieser Bereiche verhindern.
  – Darf maximal nur ein Prozess gleichzeitig auf den kritischen Bereich zugreifen, so spricht man vom **wechselseitigen Ausschluss**.
  – Wird verhindert, daß mehrere (unterschiedlich viele) Instanzen unterschiedlicher Prozeßklassen auf den Bereich zugreifen, so entspricht dies dem **Leser-Schreiber-Problem** (so dürfen beispielsweise mehrere Instanzen der Klasse Leser auf den Bereich gleichzeitig zugreifen, Instanzen der Klasse Schreiber benötigen den exklusiven Zugriff).



Programmtechnik zum Schutz kritischer Bereiche

- Aus dem Alltag sind diverse Mechanismen zum Schutz kritischer Bereiche bekannt:
  - *Signale* im Bahnverkehr
  - *Ampeln* zum Schutz der Kreuzung
  - *Schlösser* für einzelne Räume
  - *Vergabe* von Tickets
- Erstes Mittel zum Schutz kritischer Bereiche in der Softwaretechnik: der/das Semaphor, griech. für „Zeichenträger“ (siehe später und rechts)



Falsche Lösung: Verwendung einer globalen Variable

Prozeß A

```
bool block = false; //global
variable
```

Prozeß B

```
...
while(block){}; //busy wait
block=true;
... critical section A ...
block=false;
...
```

```
...
while(block){}; //busy wait
block=true;
... critical section A ...
block=false;
...
```

- Die obige Implementierung ist nicht korrekt,
  - da der Prozess direkt nach dem while-Abschnitt unterbrochen werden könnte und evtl. dann fortgesetzt wird, wenn block bereits durch einen anderen Prozess belegt ist.
  - Zudem ist die Lösung ineffizient (busy wait)

1. Möglichkeit: Peterson 1981 (Lösung für zwei Prozesse)

Deklaration globale Variablen

```
int turn=0;
boolean ready[2];
ready[0]=false;
ready[1]=false;
```

Prozess 0

```
...
ready[0]=true;
turn = 1;
while(ready[1]
    && turn==1){};//busy waiting
... critical section ...
ready[0]=false;
...
```

Prozess 1

```
...
ready[1]=true;
turn = 0;
while(ready[0]
    && turn==0){}; //busy waiting
... critical section ...
ready[1]=false;
...
```

- Ausschluss ist garantiert, aber „busy waiting“ verschwendet immer noch Rechenzeit
- Die Realisierung für N Prozesse ist als „Lamport‘s Bakery Algorithmus“ bekannt: http://en.wikipedia.org/wiki/Lamport‘s_bakery_algorithm

2. Möglichkeit: Ausschalten von Unterbrechungen zum WA

- Prozesswechsel beruhen immer auf dem Eintreffen einer Unterbrechung (z.B. neues Ereignis, Ablauf einer Zeitdauer)
- Die einfachste Möglichkeit einen Kontextwechsel zu verhindern ist das Ausschalten von Unterbrechungen bevor ein Prozess in den kritischen Bereich geht.
- Vorteile:
  - einfach zu implementieren, keine weiteren Konzepte sind nötig
  - schnelle Ausführung, Schreiben von Bits in Register
- Nachteile:
  - Für Multiprozessorsysteme ungeeignet
  - Keine Gerätebehandlung während der Sperre
  - Lange Sperren kritisch bei Echtzeitanwendungen

5.4.5 Interrupt Enable Clear register (VICIntEnClear - 0xFFFF F014)

This is a write only register. This register allows software to clear one or more bits in the Interrupt Enable register (see Section 5.4.4 "Interrupt Enable register (VICIntEnable - 0xFFFF F010)" on page 52), without having to first read it.

Table 42: Software Interrupt Clear register (VICIntEnClear - address 0xFFFF F014) bit allocation

Reset value: 0x0000 0000

Bit	31	30	29	28	27	26	25	24
Symbol	-	-	-	-	-	-	-	-
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol	-	-	AD1	BOD	I2C1	AD0	EINT3	EINT2
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol	EINT1	EINT0	RTC	PLL	SPI1/SSP	SPI0	I2C0	PWM0
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	UART1	UART0	TIMER1	TIMER0	ARMCore1	ARMCore0	-	WDT
Access	WO	WO	WO	WO	WO	WO	WO	WO

3. Möglichkeit: Semaphor

- Semaphor (griechisch von Zeichenträger, Signalmast) wurden von Edsger W. Dijkstra im Jahr 1965 eingeführt.
- Ein Semaphor ist eine Datenstruktur, bestehend aus einer Zählvariable `s`, sowie den Funktionen `down()` oder `wait()` (bzw. `P()`, von probeer te verlagen) und `up()` oder `signal()` (bzw. `V()`, von verhogen).

```
Init(Semaphor s, Int v)
{
  s = v;
}

V(Semaphor s)
{
  s = s+1;
}

P(Semaphor s)
{
  while (s <= 0) {} ;  // Blockade, unterschiedliche Implementierungen
  s = s-1 ;            // sobald s>0 belege eine Ressource
}
```

- Bevor ein Prozess in den kritischen Bereich eintritt, muss er den Semaphor mit der Funktion `down()` anfordern. Nach Verlassen wird der Bereich durch die Funktion `up()` wieder freigegeben.
- **Wichtige Annahme**: die Ausführung der Funktionen von `up` und `down` darf nicht unterbrochen werden (atomare Ausführung), siehe Realisierung
- Solange der Bereich belegt ist (Wert des Semaphors <=0), wird der aufrufende Prozess blockiert.

Realisierungen von Semaphoren

- Die Implementierung eines Semaphors erfordert spezielle Mechanismen auf Maschinenebene; der Semaphor ist für sich ein kritischer Bereich.
→ Die Funktionen `up()` und `down()` dürfen nicht unterbrochen werden, da sonst der Semaphor selbst inkonsistent werden kann.
- Funktionen die nicht unterbrechbar sind, werden **atomar** genannt.
- Realisierungsmöglichkeiten:
  1. Kurzfristige Blockade der Prozesswechsel während der Bearbeitung der Funktionen `up()` und `down()`. Implementierung durch Verwendung einer Interrupt-Sperre, denn sämtliche Prozesswechsel werden durch **Unterbrechungen** (**Interrupts**) ausgelöst.
  2. **Spinlock**: Programmiertechnik auf der Basis von Busy Waiting. Vorteil: Unabhängig vom Betriebssystem und auch in Mehrprozessorsystemen zu implementieren, jedoch massive Verschwendung von Rechenzeit. Im Gegensatz dazu können die anderen Lösungen mit Hilfe von Warteschlangen sehr effizient realisiert werden.
  3. **Test&Set**-Maschinenbefehl: Die meisten Prozessoren verfügen heute über einen Befehl „**Test&Set**“ (oder auch Test&SetLock). Dieser lädt atomar den Inhalt (typ. 0 für frei, 1 für belegt) eines Speicherwortes in ein Register und schreibt ununterbrechbar einen Wert (typ. $\neq 0$, z.B. 1 für belegt) in das Speicherwort.

Beispiel: Bankkonto

- Durch Verwendung eines gemeinsamen Semaphors `semAccount` kann das Bankkonto `account` auch beim schreibenden Zugriff von zwei Prozessen konsistent gehalten werden:

Prozess A

```
down(semAccount);
x=readAccount(account);
x=x+500;
writeAccount(x,account);
up(semAccount);
```

Prozess B

```
down(semAccount);
y=readAccount(account);
y=y-200;
writeAccount(y,account);
up(semAccount);
```

- Zur Realisierung des wechselseitigen Ausschlusses wird ein binärer Semaphor mit zwei Zuständen: 0 (belegt), 1 (frei) benötigt. Binäre Semaphore werden auch *Mutex* (von *mutal exclusion*) genannt.

Erweiterung: zählender Semaphore

- Nimmt ein Wert auch einen Wert größer eins an, so wird ein solch ein Semaphor auch als **zählender Semaphor** (counting semaphore) bezeichnet.
- Beispiel für den Einsatz von zählenden Semaphoren: In einem **Leser-Schreiber-Problem** kann die Anzahl der Leser aus Leistungsgründen z.B. auf 100 gleichzeitige Lesezugriffe beschränkt werden:

```
semaphore sem_reader_count;
init(sem_reader_count,100);
```

- Jeder Leseprozess führt dann folgenden Code aus:

```
P(sem_reader_count);
read();
V(sem_reader_count);
```

- Leser-Schreiber-Probleme sind vielfältig modifizierbar, je nach Priorität der prozesse. LS-Problem: Keine Prioritäten. Erstes LS-Problem: Leserpriorität. Zweites LS-Problem: Schreiber-Priorität.

Fragestellung: Invers zählender Semaphor

- Aufgabenstellung: Implementierung des Leser-Schreiber-Problems mit Schreiber-Priorität
- Erläuterung:
  - Auf einen Datensatz können mehrere Leser gleichzeitig oder aber ein Schreiber zugreifen.
  - Sobald ein Schreiber den Schreibwunsch äußert, soll kein weiterer Leser (oder Schreiber) mehr auf den Datensatz zugreifen können. Zum Zeitpunkt der Signalisierung bestehende Lesevorgänge können regulär beendet werden, erst danach darf der Schreiber auf die Daten zugreifen.
- Problem: Häufig wird versucht das Problem mit einem *„invers zählenden Semaphor"* zu lösen, also einem Semaphor, der bei 0 freigibt und sonst blockiert.
- Wie geht es richtig?