System-Level Programming

20 Interrupts – Concurrency

J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Summer Term 2024

http://sys.cs.fau.de/lehre/ss24
Concurrency

**Definition: Concurrency**

Two executions \( A \) and \( B \) of a program are considered to be concurrent (\( A \mid B \)), if for every single instruction \( a \) of \( A \) and \( b \) of \( B \) it is not determined, whether \( a \) or \( b \) is executed first (\( a, b \) or \( b, a \)).

Concurrency is induced by

- Interrupts
  - IRQs can interrupt a program at an “arbitrary point”
- Real-parallel sequences (by the hardware)
  - other CPU / peripheral devices access the memory at “anytime”
- Quasi-parallel sequences (e.g., threads in an operating system)
  - OS can preempt tasks “anytime”

**Problem:** Concurrent access to a **shared** state
Problems with Concurrency

Scenario
- a light gate at the entrance of a parking lot should count cars
- every 60 seconds, the value is transferred to security agency

```c
static volatile uint16_t cars;
void main(void) {
    while (1) {
        waitsec(60);
        send(cars);
        cars = 0;
    }
}
```

// photo sensor is connected
// to INT2
ISR(INT2_vect) {
    cars++;
}

Where does the problem occur?
- both `main()` as well as `ISR` read and write `cars`;
- potential lost-update anomaly
- size of the variable `cars` is larger than one register;
Problems with Concurrency

Scenario
- a light gate at the entrance of a parking lot should count cars
- every 60 seconds, the value is transferred to security agency

```c
static volatile uint16_t cars;

void main(void) {
    while (1) {
        waitsec(60);
        send(cars);
        cars = 0;
    }
}
```

Where does the problem occur?
- both `main()` as well as ISR read and write `cars`
  - potential *lost-update* anomaly
Problems with Concurrency

Scenario
- A light gate at the entrance of a parking lot should count cars every 60 seconds, the value is transferred to the security agency.

```c
static volatile uint16_t cars;

void main(void) {
  while (1) {
    waitsec(60);
    send(cars);
    cars = 0;
  }
}

// Photo sensor is connected to INT2
ISR(INT2_vect) {
  cars++;
}
```

Where does the problem occur?
- Both `main()` as well as `ISR` read and write `cars`;
  - Potential lost-update anomaly
- Size of the variable `cars` is larger than one register;
  - Potential read-write anomaly
Problems with Concurrency (continued)

Where are the problems here?

- **lost-update**: both `main()` as well as `ISR` read and write `cars`
- **read-write**: size of the variable `cars` is larger than one register

This often gets obvious only when looking at the **assembly level**

```c
void main(void) {
  ...
  send(cars);
  cars = 0;
  ...
}
```

```c
ISR(INT2_vect) {
  cars++;
}
```

```assembly
main:
  ...
  lds r24,cars
  lds r25,cars+1
  rcall send
  sts cars+1,__zero_reg__
  sts cars,__zero_reg__
  ...

INT2_vect:
  ...
  lds r24,cars ; load cars.lo
  lds r25,cars+1 ; load cars.hi
  adiw r24,1 ; add (16 bit)
  sts cars+1,r25 ; store cars.hi
  sts cars,r24 ; store cars.lo
  ...
  ; save regs
  ; load cars.lo
  ; load cars.hi
  ; add (16 bit)
  ; store cars.hi
  ; store cars.lo
  ; restore regs
```
Concurrency Problems: *Lost-Update* Anomaly

**main:**

```assembly
...  
lds r24,cars  
lds r25,cars+1  
rcall send  
sts cars+1,__zero_reg__  
sts cars,__zero_reg__  
...  
```

**INT2_vect:**

```assembly
...  ; save regs  
lds r24,cars  
lds r25,cars+1  
adiw r24,1  
sts cars+1,r25  
sts cars,r24  
...  ; restore regs  
```
Concurrency Problems: *Lost-Update* Anomaly

Let $\text{cars}=5$ and let the IRQ ( نفسها) occur at this point
Concurrent Problems: *Lost-Update* Anomaly

Let \( \text{cars} = 5 \) and let the IRQ (\( rac{1}{2} \)) occur at *this point*

- *main* already read the value of *cars* (5) from the register (register \( \mapsto \) local variable)
Concurrency Problems: *Lost-Update* Anomaly

Let cars = 5 and let the IRQ (½) occur at **this point**

- **main** already read the value of cars (5) from the register (register $\rightarrow$ local variable)
- **INT2_vect** gets executed
  - registers are saved
  - cars gets incremented $\sim$ cars = 6
  - registers are restored
Concurrency Problems: *Lost-Update Anomaly*

```
main:
    ... 
    lds r24,cars
    lds r25,cars+1
    rcall send
    sts cars+1,__zero_reg__
    sts cars,__zero_reg__
    ...

INT2_vect:
    ... ; save regs
    lds r24,cars
    lds r25,cars+1
    adiw r24,1
    sts cars+1,r25
    sts cars,r24
    ... ; restore regs
```

Let `cars=5` and let the IRQ occur at *this point*

- main already read the value of `cars` (5) from the register (register $\rightarrow$ local variable)
- INT2_vect gets executed
  - registers are saved
  - cars gets incremented $\sim$ cars=6
  - registers are restored
- main passes the old value of `cars` (5) to send
Concurrency Problems: *Lost-Update* Anomaly

**main:**

```assembly
... 
lds r24, cars  
lds r25, cars+1  
rcall send  
sts cars+1, __zero_reg__  
sts cars, __zero_reg__  
... 
```

**INT2_vect:**

```assembly
... ; save regs  
lds r24, cars  
lds r25, cars+1  
adiw r24, 1  
sts cars+1, r25  
sts cars, r24  
... ; restore regs 
```

Let `cars=5` and let the IRQ (-half) occur at **this point**

- **main** already read the value of `cars` (5) from the register (register → local variable)
- **INT2_vect** gets executed
  - registers are saved
  - `cars` gets incremented ↝ `cars=6`
  - registers are restored
- **main** passes the **old value** of `cars` (5) to **send**
- **main** sets `cars` to zero ↝ **1 car is “lost”**
Concurrency Problems: Read-Write Anomaly

```
main:
  ...  
  lds r24,cars
  lds r25,cars+1
  rcall send
  sts cars+1,__zero_reg__
  sts cars,__zero_reg__
  ...

INT2_vect:
  ... ; save regs
  lds r24,cars
  lds r25,cars+1
  adiw r24,1
  sts cars+1,r25
  sts cars,r24
  ... ; restore regs
```

Let `cars = 255` and let the IRQ (INT) occur at this point. `main` has already transmitted `cars = 255` with `send`. `main` has already set the high byte of `cars` to zero; `cars = 255`, `cars.lo = 255`, `cars.hi = 0`. `INT2_vect` gets executed; `cars` is read and incremented, overflow in the high byte; `cars = 256`, `cars.lo = 0`, `cars.hi = 1`. `main` sets the low byte of `cars` to zero; `cars = 256`, `cars.lo = 0`, `cars.hi = 1`. During the next `send`, `main` will transmit too many cars (255 cars).
Concurrency Problems: *Read-Write* Anomaly

Let \( \text{cars} = 255 \) and let the IRQ (\( \mathcal{IRQ} \)) occur at *this point*

```assembly
main:
    ...  
    lds r24,cars  
    lds r25,cars+1  
    rcall send  
    sts cars+1,__zero_reg__  
    sts cars,__zero_reg__  
    ...  

INT2_vect:
    ... ; save regs  
    lds r24,cars  
    lds r25,cars+1  
    adiw r24,1  
    sts cars+1,r25  
    sts cars,r24  
    ... ; restore regs  
```
Let \( \text{cars} = 255 \) and let the IRQ (\( ⫶ \)) occur at this point

- main has already transmitted \( \text{cars} = 255 \) with \text{send}
Concurrency Problems: *Read-Write* Anomaly

<table>
<thead>
<tr>
<th>main:</th>
<th>INT2_vect:</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>lds r24,cars</td>
<td>lds r24,cars</td>
</tr>
<tr>
<td>lds r25,cars+1</td>
<td>lds r25,cars+1</td>
</tr>
<tr>
<td>rcall send</td>
<td>rcall send</td>
</tr>
<tr>
<td>sts cars+1, <strong>zero_reg</strong></td>
<td>adiw r24,1</td>
</tr>
<tr>
<td>sts cars, <strong>zero_reg</strong></td>
<td>sts cars+1,r25</td>
</tr>
<tr>
<td>...</td>
<td>sts cars,r24</td>
</tr>
</tbody>
</table>

Let \( \text{cars}=255 \) and let the IRQ (\( \div \)) occur at **this point**

- **main** has already transmitted \( \text{cars}=255 \) with `send`
- **main** has already set the **high byte** of \( \text{cars} \) to zero
  - \( \text{cars}=255, \text{cars}.lo=255, \text{cars}.hi=0 \)
Concurrency Problems: *Read-Write* Anomaly

Let \( \text{cars} = 255 \) and let the IRQ (\( \frac{\text{IRQ}}{2} \)) occur at this point:

- **main** has already transmitted \( \text{cars} = 255 \) with \text{send}
- **main** has already set the **high byte** of \( \text{cars} \) to zero
  \( \rightsquigarrow \) \( \text{cars} = 255, \text{cars.lo} = 255, \text{cars.hi} = 0 \)
- **INT2_vect** gets executed
  \( \rightsquigarrow \) \( \text{cars} \) is read and incremented, **overflow in the high byte**
  \( \rightsquigarrow \) \( \text{cars} = 256, \text{cars.lo} = 0, \text{cars.hi} = 1 \)
Concurrency Problems: *Read-Write Anomaly*

Let \( \text{cars} = 255 \) and let the IRQ (\( \div \)) occur at **this point**

- **main** has already transmitted \( \text{cars} = 255 \) with **send**
- **main** has already set the **high byte** of \( \text{cars} \) to zero
  \( \sim \text{cars} = 255, \text{cars.lo} = 255, \text{cars.hi} = 0 \)
- **INT2_vect** gets executed
  \( \sim \text{cars} \) is read and incremented, **overflow in the high byte**
  \( \sim \text{cars} = 256, \text{cars.lo} = 0, \text{cars.hi} = 1 \)
- **main** sets the **low byte** of **cars** to zero
  \( \sim \text{cars} = 256, \text{cars.lo} = 0, \text{cars.hi} = 1 \)
  \( \sim \) **During the next send**, **main** will transmit **too many cars (255 cars)**
void main(void) {
    while(1) {
        waitsec(60);

        send(cars);
        cars = 0;
    }
}

Where exactly is the **critical region**?
void main(void) {
    while(1) {
        waitsec(60);

        send(cars);
        cars = 0;
    }
}

Where exactly is the critical region?

- Reading of cars and setting it to zero have to be executed atomically.
Interrupt Locks: Avoid Data-Flow Anomalies

```c
void main(void) {
    while(1) {
        waitsec(60);
        cli();
        send(cars);
        critical region
        cars = 0;
        sei();
    }
}
```

Where exactly is the **critical region**?

- Reading of `cars` and setting it to zero have to be executed atomically
- This can be forced by using **interrupt locks**
  - ISR interrupts `main`, never the other way round
    - asymmetric synchronization (also unilateral synchronization)
Where exactly is the critical region?

- Reading of `cars` and setting it to zero have to be executed atomically
- This can be forced by using interrupt locks
  - ISR interrupts main, never the other way round
    ~ asymmetric synchronization (also unilateral synchronization)
- Attention: regions with blocked interrupts should be as short as possible
  - How long does the function `send` take?
  - Can `send` be excluded from the critical region?
Scenario, part 2 (function `waitsec()`)

- a light gate at the entrance of a parking lot should count cars
- every 60 seconds, the value is transferred to security agency

```c
void waitsec(uint8_t sec) {
    // setup timer
    sleep_enable();
    event = 0;
    while (! event) { // wait for event
        sleep_cpu(); // until next irq
    }
    sleep_disable();
}
```

Where exactly does the problem occur?

```c
static volatile int8_t event;
```

// TIMER1 ISR
// triggers when
// `waitsec()` expires
ISR(TIMER1_COMPA_vect) {
    event = 1;
}
Problems with Concurrency (continued)

Scenario, part 2 (function `waitsec()`)

- A light gate at the entrance of a parking lot should count cars every 60 seconds, the value is transferred to security agency

```c
void waitsec(uint8_t sec) {
    ... // setup timer
    sleep_enable();
    event = 0;
    while (! event) { // wait for event
        sleep_cpu(); // until next irq
    }
    sleep_disable();
}
```

```c
static volatile int8_t event;
// TIMER1 ISR
// triggers when waitsec() expires
ISR(TIMER1_COMPA_vect) {
    event = 1;
}
```

Where exactly does the problem occur?

- **Test, whether sth. is to be done**, followed by sleeping until there is sth. to do
Scenario, part 2 (function \texttt{waitsec()})

- a light gate at the entrance of a parking lot should count cars
- every 60 seconds, the value is transferred to security agency

```c
void waitsec(uint8_t sec) {
  ...  // setup timer
  sleep_enable();
  event = 0;
  while (! event) { // wait for event
    sleep_cpu();  // until next irq
  }
  sleep_disable();
}
```

```c
static volatile int8_t event;

// TIMER1 ISR
// triggers when
// waitsec() expires
ISR(TIMER1_COMPA_vect) {
  event = 1;
}
```

Where exactly does the problem occur?

- Test, whether sth. is to be done, followed by **sleeping until there is sth. to do**
Problems with Concurrency (continued)

Scenario, part 2 (function `waitsec()`)
- a light gate at the entrance of a parking lot should count cars
- every 60 seconds, the value is transferred to security agency

```c
void waitsec(uint8_t sec) {
    // setup timer
    sleep_enable();
    event = 0;
    while (! event) { // wait for event
        sleep_cpu(); // until next irq
    }
    sleep_disable();
}
```

```c
static volatile int8_t event;
// TIMER1 ISR
// triggers when
// waitsec() expires
ISR(TIMER1_COMPA_vect) {
    event = 1;
}
```

Where exactly does the problem occur?
- Test, whether sth. is to be done, followed by sleeping until there is sth. to do
  ◼️ Potential `lost-wakeup` anomaly
Concurrent Problems: *Lost-Wakeup-Anomaly*

```c
void waitsec(uint8_t sec) {
    ...  // setup timer
    sleep_enable();
    event = 0;
    while (!event) {
        sleep_cpu();
    }
    sleep_disable();
}
```

```c
static volatile int8_t event;

// TIMER1 ISR
// triggers when
// waitsec() expires
ISR(TIMER1_COMPA_vect) {
    event = 1;
}
```

Suppose, at **this point** a timer-IRQ (‡) occurs
Concurrency Problems: *Lost-Wake up-Anomaly*

```c
void waitsec(uint8_t sec) {
    // setup timer
    sleep_enable();
    event = 0;
    while (! event) {
        sleep_cpu();
    }
    sleep_disable();
}
```

```c
static volatile int8_t event;

// TIMER1 ISR
// triggers when
// waitsec() expires
ISR(TIMER1_COMPA_vect) {
    event = 1;
}
```

Suppose, at **this point** a timer-IRQ (‡) occurs

- waitsec already determined that event is not set
Concurrent Problems: *Lost-Wake-up-Anomaly*

```c
void waitsec(uint8_t sec) {
    sleep_enable(); // setup timer
    event = 0;
    while (!event) {
        sleep_cpu();
    }
    sleep_disable();
}
```

```c
static volatile int8_t event;
```

```c
// TIMER1 ISR
// triggers when
// waitsec() expires
ISR(TIMER1_COMPA_vect) {
    event = 1;
}
```

Suppose, at this point a timer-IRQ (払い戻し) occurs

- `waitsec` already determined that `event` is not set
- ISR gets executed $\rightsquigarrow$ `event` is set to 1
Concurrent Problems: *Lost-Wake-Up-Anomaly*

```c
void waitsec(uint8_t sec) {
    // setup timer
    sleep_enable();
    event = 0;
    while (!event) {
        sleep_cpu();
    }
    sleep_disable();
}
```

```c
static volatile int8_t event;
```

```
// TIMER1 ISR
// triggers when
// waitsec() expires
ISR(TIMER1_COMPA_vect) {
    event = 1;
}
```

Suppose, at **this point** a timer-IRQ (‡) occurs

- `waitsec` already determined that `event` is **not set**
- **ISR** gets executed ⇒ `event` **is set to 1**
- Even though `event` is set to 1, the **sleep state is entered**
  ⇒ If no further IRQ occurs, **sleeping forever**
Where exactly can the **critical region** be located?
Where exactly can the **critical region** be located?

- Evaluation of the condition and entry of the sleeping state (Can *this* be solved by interrupt blocking?)
Where exactly can the **critical region** be located?

- Evaluation of the condition and entry of the sleeping state
  (Can *this* be solved by interrupt blocking?)
- Problem: The IRQs have to be unblocked prior to `sleep_cpu()`!
Lost-Wakeup: Prevention of Deep Sleep

```c
void waitsec(uint8_t sec) {
    ... // setup timer
    sleep_enable();
    event = 0;
    cli();
    while (! event) {
        sei();
        sleep_cpu();
        cli();
    }
    sei();
    sleep_disable();
}
```

```c
static volatile int8_t event;
// TIMER1 ISR
// triggers when
// waitsec() expires
ISR(TIMER1_COMPA_vect) {
    event = 1;
}
```

Where exactly can the **critical region** be located?

- Evaluation of the condition and entry of the sleeping state
  (Can this be solved by interrupt blocking?)
- Problem: The IRQs have to be unblocked prior to `sleep_cpu()`!
- Works thanks to specific **hardware support**:
  - Sequence `sei, sleep` is executed as an **atomic** instruction
Summary

- Handling of interrupts is asynchronous to the program flow
  - unexpected $\rightarrow$ current state has to be saved in the interrupt handler
  - source of concurrency $\rightarrow$ synchronisation required

- Measures for synchronization
  - shared variables shall (always) be declared as volatile
  - blocking arrival of interrupts: cli, sei (when working with non-atomic accesses that translate to more than one machine instruction)
  - Locking for longer times leads to the loss of IRQs!

- Concurrency induced by interrupts is enormous source for errors
  - lost-update and lost-wakeup problems
  - indeterministic $\rightarrow$ cannot efficiently be tested for

- Important for controlability: modularization
  - Interrupt handler and functions accessing a shared state (static variables!) should be encapsulated in their own module.