Agenda

Preface

Fundamentals
- Race Condition
- Sequential Control
- Concurrent Control

Patterns
- Data Race
- Control-Flow Race

Summary

Subject Matter

- discussion on abstract concepts as to critical sections:
  - race condition
    - caused by a fault in a non-sequential program
    - possibly effects an error of a non-sequential process
    - if so, eventually effects failure of a computing system
Subject Matter

- discussion on abstract concepts as to critical sections:
  - race condition:
    - caused by a fault in a non-sequential program
    - possibly effects an error of a non-sequential process
    - if so, eventually effects failure of a computing system
  - race management:
    - pessimistic, sequential control at a "chokepoint"
      - blocking synchronisation, lock-based
    - optimistic, concurrent control on a "multi-lane road"
      - non-blocking synchronisation, obstruction-free
- race management:
  - pessimistic, sequential control at a "chokepoint"
    - blocking synchronisation, lock-based
  - optimistic, concurrent control on a "multi-lane road"
    - non-blocking synchronisation, obstruction-free
- explanation of typical patterns of critical sections in system software:
  - excursion to database systems: transactions, isolation (ACID properties)
  - basic forms of inconsistency due to actions of simultaneous processes

Outline

- Preface
- Fundamentals
  - Race Condition
  - Sequential Control
  - Concurrent Control
- Patterns
  - Data Race
  - Control-Flow Race
- Summary
Critical Sequence or Timing of Processes

Definition (race condition, race hazard)
The behaviour of a computing system, hardware or software, in which the result or progress depends on the simultaneity of processes.

- the notion has origin in spurious behaviour of electronic circuits [10]
  - namely when (at least) two signals race to be first in defining an output
  - it becomes a critical race when the order of changing internal state can impact the outcome of the state machine of that circuit
- in software, the program flow discloses a fault that possibly results in:
  - critical race
    - an error that, in turn, may cause failure of a system
    - e.g., altimetry uses readings that reflect wrong height
  - non-critical race
    - unanticipated behaviour
    - e.g., the hand of the “clock app” overlaps time data
- prevention of especially critical races is a must, implicitly or explicitly
- but before, the race condition behind must be even localised...
A software fault (also: defect) that disappears or results in a different run-time behaviour when one attempts to probe or isolate it.

- in the minority of cases is the presence of a race condition obvious
  - necessary condition (i.e., mandatory) is a non-sequential program
  - sufficient condition are simultaneous processes of that very program
  \[\rightarrow\text{the sequence of all actions is defined by a \textit{partial order} (cf. [9, p. 17])}\]
- loss of the total order of all actions makes overlapping indeterminate

Program testing can be used to show the presence of bugs, but never to show their absence! [1, p. 85]

- but formal methods and proofs of correctness are widely unrealistic here
  \[\rightarrow\text{software must be designed for simultaneity from the very beginning}\]

- but each software measure of that kind will change the time elapsed
- as the case may be, a still present fault no longer manifest into an error
Heisenbug

A software fault (also: defect) that disappears or results in a different run-time behaviour when one attempts to probe or isolate it.

- in the minority of cases is the presence of a race condition obvious
  - necessary condition (i.e., mandatory) is a non-sequential program
  - sufficient condition are simultaneous processes of that very program
  ⇒ the sequence of all actions is defined by a **partial order** (cf. [9, p.17])
- loss of the **total order** of all actions makes overlapping indeterminate

*Program testing can be used to show the presence of bugs, but never to show their absence!* [1, p. 85]

- but formal methods and proofs of correctness are widely unrealistic here
  ⇒ **software must be designed for simultaneity from the very beginning**
- common practice is to approach the fault location by “test output”
  - but each software measure of that kind will change the time elapsed
  - as the case may be, a still present fault no longer manifest into an error
  ⇒ occasionally, fault diagnostics ends—and one is okay with that defect...

Critical Section

Definition (following computer science folklore)

A sequence of program statements controlled by a sequential process in order to assure consistency according to instructions.

- a sequential program that describes **mutual exclusive actions**
  - that is to say, activities that relate to each other as if they are indivisible because no activity interrupts or overlaps, resp., the other [6, p.137]
  - a "software ELOP" that is subject to **atomic execution** by its processor

Definition (without anticipation of an implementation decision)

A sequence of program statements that assure consistency according to instructions even in the presence of simultaneous processes.
Critical Section

Definition (following computer science folklore)
A sequence of program statements controlled by a sequential process in order to assure consistency according to instructions.

- A sequential program that describes **mutual exclusive actions**
  - that is to say, activities that relate to each other as if they are indivisible because no activity interrupts or overlapped, resp., the other [6, p.137]
- a "software ELOP" that is subject to **atomic execution** by its processor

Definition (without anticipation of an implementation decision)
A sequence of program statements that assure consistency according to instructions even in the presence of simultaneous processes.

- a sequential or non-sequential program, according to **requirements**
- synchronisation as a constraint on the manner in which the sequence of program statements in question implements and delivers functionality
- **separation of concerns**: non-functional v. functional properties

A Matter of Reference

Separation of Concerns
Provide the wanted **functionality** in a first step and consider different implementation variants of that very functionality in a second step.

Beware of simultaneous processes: race condition in `push` (cf. p.30)

1 last in, first out

---

© wosch CS (WS 2016, LEC 4) Fundamentals–Sequential Control
Separation of Concerns

Provide the wanted functionality in a first step and consider different implementation variants of that very functionality in a second step.

by way of example, LIFO\(^1\) storing (push) with a singly-linked list

\begin{verbatim}
typedef struct chain {
    struct chain *link; /* next list element */
} chain_t;
\end{verbatim}

prepend specified item to this list (stack policy):

\begin{verbatim}
void push(chain_t *this, chain_t *item) {
    item->link = this->link; /* make head follow item */
    this->link = item; /* make item new head */
}
\end{verbatim}

beware of simultaneous processes: race condition in push (cf. p.30)

\(^1\)last in, first out

Operating-System Machine Level

“take a sledgehammer to crack a nut”\(^2\), blocking synchronisation

extended (new) abstraction used to represent the list head:

\begin{verbatim}
typedef struct mutex_chain {
    chain_t head;
} mutex_chain_t;
\end{verbatim}

extended and adapted (new) push resulting in a sequential process:

\begin{verbatim}
void mutex_push(mutex_chain_t *this, chain_t *item) {
    pthread_mutex_lock(&this->lock);
    item->link = this->head.link;
    this->head.link = item;
    pthread_mutex_unlock(&this->lock);
}
\end{verbatim}

lock blocks simultaneous processes out from push, till after unlock

\(^2\)(Ger.) mit Kanonen auf Spatzen schießen.

Instruction Set Architecture Level

avant-garde: non-blocking synchronisation of the push function

reused abstraction, for both list head and list element:

\begin{verbatim}
typedef struct chain {
    struct chain *link; /* next list element */
} chain_t;
\end{verbatim}

extended and adapted (new) push resulting in a non-sequential process:

\begin{verbatim}
void cas_push(chain_t *this, chain_t *item) {
    do item->link = this->link;
    while (!CAS(&this->link, item->link, item));
}
\end{verbatim}

lock blocks simultaneous processes out from push, till after unlock
avant-garde: non-blocking synchronisation of the push function
- reused abstraction, for both list head and list element:

```c
typedef struct chain {
    struct chain *link; /* next list element */
} chain_t;
```

- extended and adapted (new) push resulting in a non-sequential process:

```c
void cas_push(chain_t *this, chain_t *item) {
    do item->link = this->link;
    while (!CAS(&this->link, item->link, item));
}
```

- CAS (compare and swap) blocks simultaneous processes out, shortly
  - built-in function: `#define CAS __sync_bool_compare_and_swap:

```c
atomic bool cas(type *ref, type old, type new) {
    return (*ref == old) ? (*ref = new, true) : false;
}
```

- compiled by GCC into (x86) “lock cmpxchg” instruction/ELOP

---

push (left-hand) employs lock to “stonewall” simultaneous processes
- it also needs to rely on an atomic instruction\(^3\) to implement the lock
  - the sequential process lasts from that very instruction till after unlock
- alternatively (right-hand), push does not lock simultaneous processes
  - nevertheless, to this end, it needs to rely on an atomic instruction as well
  - but the sequential process lasts only for the duration of that instruction

---

\(^3\)Atomic write or atomic read-modify-write, depending on the lock protocol.
The Problem of Locks

- locks are not composable
- locks are prone to livelocks or deadlocks, resp.
- locks rely on programmers to strictly follow conventions
- locks are a synchronisation measure of global program property
  beyond that:
- locks increase sequentiality or decrease concurrency, resp.
- locks are prone to priority violation or inversion, resp.
- locks rather waste computing time
- locks are rather coarse-grained
- locks require alternatives

---

Transaction

Definition (cf. “logical critical section” on p.12)

Entity of one ore more independent units of work and by means of which data integrity is ensured on the basis of well-defined rules.

- transactional processing of shared data is differentiated as follows:
  - pessimistic
    - data integrity is ensured by a lock-based protocol
    - typically, exclusive and shared locks are distinguished
    - prone to performance reduction and deadlock
  - optimistic
    - data integrity is ensured by a validation-based protocol
    - thereby, a transaction is structured into several phases
      - read and execute, validate, and write or rollback (retry)
    - prone to performance reduction if too many rollbacks
**Transaction**

**Make-or-Break, Rollback**

**Definition (cf. “logical critical section” on p. 12)**

Entity of one or more independent units of work and by means of which data integrity is ensured on the basis of well-defined rules.

- **pessimistic**
  - Data integrity is ensured by a lock-based protocol
  - Typically, exclusive and shared locks are distinguished
  - Prone to performance reduction and deadlock
- **optimistic**
  - Data integrity is ensured by a validation-based protocol
  - Thereby, a transaction is structured into several phases
    - Read and execute, validate, and write or rollback (retry)
  - Prone to performance reduction if too many rollbacks
- **semi-optimistic**
  - Block the actions of a transaction when integrity rules could be violated (lock-based approach)
  - Otherwise, delay integrity rule checking (i.e., validation) to the end similar to the optimistic approach

**Devoid of Obstruction**

**Concurrency Control Without Locks**

**Definition (Optimistic Concurrency Control [8])**

Method of coordination for the purpose of updating shared data by mainly relying on transaction backup as control mechanisms.

- Simplified, a transaction is an “all or nothing” approach of computing
  - Each action on shared data must either complete in its entirety or have no effect—except the time elapsed or energy consumed
  - In doing so, these actions must also adhere to a defined isolation level
Devoid of Obstruction

ACID Properties

Concurrency Control Without Locks

Definition (Optimistic Concurrency Control [8])

Method of coordination for the purpose of updating shared data by mainly relying on transaction backup as control mechanisms.

- simplified, a transaction is an “all or nothing” approach of computing
  - each action on shared data must either complete in its entirety or have no effect—except the time elapsed or energy consumed
  - in doing so, these actions must also adhere to a defined isolation level

- the method establishes the basis for non-blocking synchronisation

Do

Validation and, possibly, write phase:

read phase:
- save a private copy of the shared data to be updated;
- compute a new private data value based on that copy;
- validation and, possibly, write phase:
- try to commit the computed value as new shared data;
- while commit failed (i.e., transaction has not completed).

- validation and, possibly, write are an “indivisible action” (cf. p.11)

ACID Properties

Fundamental rules for an atomic transaction to establish well-defined system behaviour in the presence of faults and simultaneous actions.

  - atomicity: a transaction either happens or it does not
    - either all are bound by the contract or none are
  - consistency: each successful transaction commits only legal results
    - a transaction must obey legal protocols
  - isolation: events within a transaction must be hidden from other transactions running concurrently
    - and the key for it is synchronisation [2]
  - durability: once a transaction is committed, it cannot be abrogated
    - even in the event of power loss, crashes, or errors

- consistency: each successful transaction commits only legal results
  - a transaction must obey legal protocols
- isolation: events within a transaction must be hidden from other transactions running concurrently
  - and the key for it is synchronisation [2]
- durability: once a transaction is committed, it cannot be abrogated
  - even in the event of power loss, crashes, or errors

- the isolation property is particularly important for concurrent systems
  - it has a large impact on the degree of concurrency that can be achieved
  - synchronisation paradigms decide upon the loss of concurrency, even too
    - along with the trade-off between synchronisation granularity and frequency
Separation of simultaneous transactions in a manner that none of them falls in an undefined state or causes permanent inconsistencies.

- basic forms of inconsistency due to concurrency [3, p. 431]:
  - non-repeatable read
    - repeated read actions deliver different values
    - read → write dependency
  - dirty read
    - not yet committed data is already (early) read
    - write → read dependency
  - lost update
    - already committed data is overwritten
    - write → write dependency

- advanced read/write conflict: delayed use of “state-contingent data”
  - inconsistent read
    - same search criteria apply to different data sets

- data race patterns that remain valid beyond database systems

5 Concurrency is not affected by the order of read actions, which in particular commute: read → read dependency is thus excluded.
Non-Repeatable Read

When the same read operation consecutively executed by a process on a shared data set may deliver different results.

- assuming that $i$ may be manipulated by a simultaneous process:

```
int twofold() {
    extern volatile int i;
    return i + i;
}
```

- apparently correct, but...

- apparently correct, but...

  - lines 7 and 8 describe the same read action on $i$
  - adversely, line 8 may read a value different from line 7
  - risk of wrong operation

- lines 7 and 8 describe the same read action on $i$
- adversely, line 8 may read a value different from line 7
Non-Repeatable Read

When the same read operation consecutively executed by a process on a shared data set may deliver different results.

assuming that $i$ may be manipulated by a simultaneous process:

```c
int twofold() {
    extern volatile int i;
    return i + i;
}
```

apparently correct, but...

risk of wrong operation

Solution Statement (rank-order priority, ROP)

1. reduction to hardware ELOP, 2. non-blocking, or 3. blocking synchronisation.

Inconsistent Read

When the value computed on the basis of the results of several read operations consecutively executed by a process would be different at the time the value is actually used by that process.

assuming that `string` is reduced/enlarged by a simultaneous process:

```c
char string[80];

int fill(char c) {
    int i;
    for (i = 0; i < 80; i++) {
        if (!string[i]) {
            string[i] = c;
            return i;
        }
    }
    return -1;
}
```

apparently correct, but...

that $i$ identified the first free entry in line 6

must no longer hold in line 7

risk of error propagation

Solution Statement (ROP)

1. non-blocking or
2. blocking synchronisation.
Inconsistent Read

When the value computed on the basis of the results of several read operations consecutively executed by a process would be different at the time the value is actually used by that process.

assuming that string is reduced/enlarged by a simultaneous process:

```c
char string[80];

int fill(char c) {
    int i;
    for (i = 0; i < 80; i++) {
        if (!string[i]) {
            string[i] = c;
            return i;
        }
    }
    return -1;
}
```

apparently correct, but...

that i identified the first free entry in line 6

must no longer hold in line 7

 риск of error propagation

Solution Statement (ROP)
1. non-blocking or
2. blocking synchronisation.

Dirty Read

When the same uncommitted shared variable is read by a process and written by another one, simultaneously.

assuming that the ASM part is run by two simultaneous processes:

reading of uncommitted data: counting revisited [9, p.14–15]

C/C++

```
int i ++;
```

apparently correct

ASM

```
2 movl i, %eax
3 addl $1, %eax
4 movl %eax, i
```

apparently correct

the old value of i can be read by both processes (line 2)

before the new value is written to i by any of them (line 4)

 риск of wrong reading (cf. p.31)

Dirty Read

When the same uncommitted shared variable is read by a process and written by another one, simultaneously.

assuming that the ASM part is run by two simultaneous processes:

reading of uncommitted data: counting revisited [9, p.14–15]

C/C++

```
int i ++;
```

apparently correct

ASM

```
2 movl i, %eax
3 addl $1, %eax
4 movl %eax, i
```

apparently correct

the old value of i can be read by both processes (line 2)

before the new value is written to i by any of them (line 4)
### Dirty Read

When the same uncommitted shared variable is read by a process and written by another one, simultaneously.

- assuming that the ASM part is run by two simultaneous processes:
  - reading of uncommitted data: counting revisited \([9, \text{p.14–15}]\)

#### ASM

<table>
<thead>
<tr>
<th>C/C++</th>
<th>ASM</th>
</tr>
</thead>
<tbody>
<tr>
<td>i++;</td>
<td>i++</td>
</tr>
<tr>
<td>int i</td>
<td>movl i, %eax</td>
</tr>
<tr>
<td></td>
<td>addl $1, %eax</td>
</tr>
<tr>
<td></td>
<td>movl %eax, i</td>
</tr>
</tbody>
</table>

- apparently correct, but...
  - risk of wrong reading (cf. p. 31)

#### Solution Statement (ROP)

1. reduction to hardware ELOP, 2. non-blocking, or 3. blocking synchronisation.

### Lost Update

When the same uncommitted shared variable is written by processes simultaneously without having noticed prior writes of each other.

- assuming that `vary` and `init` are run by own simultaneous processes:

```c
int total = 0;
int vary(int i) {
    return total += i;
}
void init(int i) {
    total = i;
}
```

- apparently correct, but...
  - the write in line 13 is unaware of the write in line 18

#### ASM

<table>
<thead>
<tr>
<th>C/C++</th>
<th>ASM</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl 4(% esp), %eax</td>
<td>movl total, %eax</td>
</tr>
<tr>
<td>movl %eax, total</td>
<td>movl %eax, total</td>
</tr>
</tbody>
</table>

- apparently correct, but...
  - risk of blind writes

### Write-Read Conflict

When the same uncommitted shared variable is read by a process and written by another one, simultaneously without having noticed prior writes of each other.

- assuming that `vary` and `init` are run by own simultaneous processes:

```c
int total = 0;
int vary(int i) {
    return total += i;
}
void init(int i) {
    total = i;
}
```

- apparently correct, but...
  - the write in line 13 is unaware of the write in line 18

#### ASM

<table>
<thead>
<tr>
<th>C/C++</th>
<th>ASM</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl 4(% esp), %eax</td>
<td>movl total, %eax</td>
</tr>
<tr>
<td>movl %eax, total</td>
<td>movl %eax, total</td>
</tr>
</tbody>
</table>

- apparently correct, but...
  - risk of blind writes
Lost Update

When the same uncommitted shared variable is written by processes simultaneously without having noticed prior writes of each other.

assuming that \textit{vary} and \textit{init} are run by own simultaneous processes:

\begin{verbatim}
1 int total = 0;
2 int vary(int i) {
3   return total += i;
4 }
5 void init(int i) {
6   total = i;
7 }
\end{verbatim}

\begin{verbatim}
10 vary:
11   movl total, %eax
12   addl 4(%esp), %eax
13   movl %eax, total
14   ret
16 init:
17   movl 4(%esp), %eax
18   movl %eax, total
19   ret
\end{verbatim}

\begin{itemize}
\item apparently correct, but...
\item risk of blind writes
\end{itemize}

\textbf{Solution Statement:} same as "dirty read", but this problem is not serializable.

Non-sequential and sequential execution results in different states (cf. p. 32).

Lost Wakeup

When a process identified a waiting condition and will seek blocking simultaneously to the invalidation of the condition by another process.

assuming that both sections are run by two interacting processes:

\begin{verbatim}
1 while (this->load == 0) 3 if (this->load++ == 0)
2   await(this); 4   cause(this);
\end{verbatim}

\begin{itemize}
\item apparently correct
\item risk of everlasting block
\end{itemize}

\textbf{Solution Statement (ROP)}

1. non-blocking or 2. blocking synchronisation.

© wosch CS (WS 2016, LEC 4) Patterns – Data Race
Lost Wakeup Check-Block Conflict

When a process identified a waiting condition and will seek blocking simultaneously to the invalidation of the condition by another process.

- assuming that both sections are run by two interacting processes:
  - check condition and block
    
    ```
    while (this->load == 0)
    
    await(this);
    ```
  - invalidate condition and proceed
    
    ```
    if (this->load++ == 0)
    
    cause(this);
    ```

- apparently correct, but...

  → risk of everlasting block

Solution Statement (ROP)
1. non-blocking or 2. blocking synchronisation.

“Time-of-Check to Time-of-Use” Problem TOCTOU

When a process checks the state of a resource before using this very resource, but the state is modified interim by another process.

- assuming that file “argv[1]” is modified by a simultaneous process:
  
  ```
  #include <stdio.h>
  #include <sys/stat.h>
  ```

  ```
  int main(int argc, char *argv[]) {
    struct stat buf;
    if (lstat(argv[1], &buf))
      perror("lstat failed");
    else {
      printf("%s is ", argv[1]);
      if (buf.st_size != 0)
        printf("not ");
      printf("empty\n");
    }
  }
  ```

  - apparently correct

  → file size may change between lines 6 and 10

  → risk of false positive or negative, resp.

Solution Statement / frownie
blocking synchronisation.
When a process checks the state of a resource before using this very resource, but the state is modified interim by another process:

assuming that file “argv[1]” is modified by a simultaneous process:

```c
#include <stdio.h>
#include <sys/stat.h>

int main(int argc, char *argv[]) {
    struct stat buf;
    if (lstat(argv[1], &buf))
        perror("lstat failed");
    else {
        printf("%s is ", argv[1]);
        if (buf.st_size != 0)
            printf(" not ");
        printf(" empty\n");
    }
}
```

Solution Statement: blocking synchronisation.

Résumé

A race condition is an adverse behaviour of a computing system caused by a fault (“concurrency bug”) in a non-sequential program possibly effects an error of a non-sequential process as a further consequence, possibly effects failure of the system.

The faulty area in the non-sequential program is a “critical section” it must be protected by making pessimistic or optimistic assumptions.

Computer science folklore goes the pessimistic way: mutual exclusion avant-garde takes up a position that favours an optimistic approach.

In the strict sense, it is still a matter of the level of abstraction at which the folkloristic critical section appears.

In practice, one is faced with different patterns of race conditions data races: non-repeatable read, inconsistent read, dirty read, lost update control flow races: lost wakeup, TOCTOU

Problem solving should respect differentiated views and techniques

1. hardware ELOP
2. non-blocking
3. blocking synchronisation

Résumé

A race condition is an adverse behaviour of a computing system caused by a fault (“concurrency bug”) in a non-sequential program possibly effects an error of a non-sequential process as a further consequence, possibly effects failure of the system.

The faulty area in the non-sequential program is a “critical section” it must be protected by making pessimistic or optimistic assumptions.

Computer science folklore goes the pessimistic way: mutual exclusion avant-garde takes up a position that favours an optimistic approach.

In the strict sense, it is still a matter of the level of abstraction at which the folkloristic critical section appears.

In practice, one is faced with different patterns of race conditions data races: non-repeatable read, inconsistent read, dirty read, lost update control flow races: lost wakeup, TOCTOU

Problem solving should respect differentiated views and techniques

1. hardware ELOP
2. non-blocking
3. blocking synchronisation
a race condition is an adverse behaviour of a computing system caused by a fault ("concurrency bug") in a non-sequential program possibly effects an error of a non-sequential process as a further consequence, possibly effects failure of the system the faulty area in the non-sequential program is a "critical section" it must be protected by making pessimistic or optimistic assumptions computer science folklore goes the pessimistic way: mutual exclusion avant-garde takes up a position that favours an optimistic approach

In the strict sense, it is still a matter of the level of abstraction at which the folkloristic critical section appears.

in practice, one is faced with different patterns of race conditions data races: non-repeatable read, inconsistent read, dirty read, lost update control flow races: lost wakeup, TOCTOU problem solving should respect differentiated views and techniques 1. hardware ELOP, 2. non-blocking, and 3. blocking synchronisation

In the strict sense, it is still a matter of the level of abstraction at which the folkloristic critical section appears.

1. hardware ELOP, 2. non-blocking, and 3. blocking synchronisation

© wosch CS (WS 2016, LEC 4) Summary 26
Simultaneous Push Considered Harmful

<table>
<thead>
<tr>
<th>push</th>
<th>simultaneous processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: push(list, X)</td>
<td>Z \rightarrow \text{item-&gt;link = this-&gt;link} \rightarrow X \rightarrow Z</td>
</tr>
<tr>
<td>B: push(list, Y)</td>
<td>Y \rightarrow Z \rightarrow \text{item-&gt;link = this-&gt;link} \rightarrow Y \rightarrow Z</td>
</tr>
</tbody>
</table>

- assuming that the stack is implemented as singly-linked list (cf. p. 9)
- initially, Z is the only stored entry and placed on the top of stack
- **misadventure**: processes A and B simultaneously apply the stack
Simultaneous Push Considered Harmful

<table>
<thead>
<tr>
<th>push</th>
<th>simultaneous processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>A : push(list, X)</td>
</tr>
<tr>
<td>Z</td>
<td>item-&gt;link = this-&gt;link</td>
</tr>
<tr>
<td>Z</td>
<td>item-&gt;link = this-&gt;link</td>
</tr>
<tr>
<td>Y → Z</td>
<td>this-&gt;link = item</td>
</tr>
<tr>
<td>X → Z</td>
<td>this-&gt;link = item</td>
</tr>
</tbody>
</table>

assuming that the stack is implemented as singly-linked list (cf. p. 9)
- initially, Z is the only stored entry and placed on the top of stack

misadventure: processes A and B simultaneously apply the stack
- process A first attempts to push item X but gets overlapped by B
  - X already points to the top of stack, but the latter has not yet been updated

Simultaneous Push Considered Harmful

<table>
<thead>
<tr>
<th>push</th>
<th>simultaneous processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>A : push(list, X)</td>
</tr>
<tr>
<td>Z</td>
<td>item-&gt;link = this-&gt;link</td>
</tr>
<tr>
<td>Z</td>
<td>item-&gt;link = this-&gt;link</td>
</tr>
<tr>
<td>Y → Z</td>
<td>this-&gt;link = item</td>
</tr>
<tr>
<td>X → Z</td>
<td>this-&gt;link = item</td>
</tr>
</tbody>
</table>

assuming that the stack is implemented as singly-linked list (cf. p. 9)
- initially, Z is the only stored entry and placed on the top of stack

misadventure: processes A and B simultaneously apply the stack
- process A first attempts to push item X but gets overlapped by B
  - X already points to the top of stack, but the latter has not yet been updated
  - X already points to the top of stack, but the latter has not yet been updated

Simultaneous Push Considered Harmful

<table>
<thead>
<tr>
<th>push</th>
<th>simultaneous processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>A : push(list, X)</td>
</tr>
<tr>
<td>Z</td>
<td>item-&gt;link = this-&gt;link</td>
</tr>
<tr>
<td>Z</td>
<td>item-&gt;link = this-&gt;link</td>
</tr>
<tr>
<td>Y → Z</td>
<td>this-&gt;link = item</td>
</tr>
<tr>
<td>X → Z</td>
<td>this-&gt;link = item</td>
</tr>
</tbody>
</table>

assuming that the stack is implemented as singly-linked list (cf. p. 9)
- initially, Z is the only stored entry and placed on the top of stack

misadventure: processes A and B simultaneously apply the stack
- process A first attempts to push item X but gets overlapped by B
  - X already points to the top of stack, but the latter has not yet been updated
  - process B conducts its push of item Y uninterrupted, deposits its item
  - the top of stack has been updated, the stack contains two entries: X → Z
  - process A resumes and completes its push of item X, deposits its item
  - the top of stack has been updated, the stack contains two entries: X → Z

Simultaneous Push Considered Harmful

<table>
<thead>
<tr>
<th>push</th>
<th>simultaneous processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>A : push(list, X)</td>
</tr>
<tr>
<td>Z</td>
<td>item-&gt;link = this-&gt;link</td>
</tr>
<tr>
<td>Z</td>
<td>item-&gt;link = this-&gt;link</td>
</tr>
<tr>
<td>Y → Z</td>
<td>this-&gt;link = item</td>
</tr>
<tr>
<td>X → Z</td>
<td>this-&gt;link = item</td>
</tr>
</tbody>
</table>

assuming that the stack is implemented as singly-linked list (cf. p. 9)
- initially, Z is the only stored entry and placed on the top of stack

misadventure: processes A and B simultaneously apply the stack
- process A first attempts to push item X but gets overlapped by B
  - X already points to the top of stack, but the latter has not yet been updated
  - process B conducts its push of item Y uninterrupted, deposits its item
  - the top of stack has been updated, the stack contains two entries: X → Z
  - process A resumes and completes its push of item X, deposits its item
  - the top of stack has been updated, the stack contains two entries: X → Z
  - two push operations executed, but only one more item was stored...
<table>
<thead>
<tr>
<th>i++</th>
<th>simultaneous processes $^b$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$^a$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>step</th>
<th>action</th>
<th>%eax</th>
<th>action</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>1</td>
<td>movl i, %eax</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>2</td>
<td>addl $1, %eax</td>
<td>43</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$^a$Each process has a software prototype of %eax available.

$^b$When both reads are executed simultaneously, the value reads as stored on the last step, even though it was updated on an earlier step.

$^c$Although it is an important question, whether each process has a software prototype of %eax available.

$^d$Each process has a software prototype of %eax available.

$^e$Reading and Processing Uncommitted Data

<table>
<thead>
<tr>
<th>i++</th>
<th>simultaneous processes $^b$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$^c$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>step</th>
<th>action</th>
<th>%eax</th>
<th>action</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>1</td>
<td>movl i, %eax</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>2</td>
<td>addl $1, %eax</td>
<td>43</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reading and Processing Uncommitted Data

<table>
<thead>
<tr>
<th>i++</th>
<th>simultaneous processes $^b$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$^d$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>step</th>
<th>action</th>
<th>%eax</th>
<th>action</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>1</td>
<td>movl i, %eax</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>2</td>
<td>addl $1, %eax</td>
<td>43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>3</td>
<td></td>
<td></td>
<td>movl i, %eax</td>
<td>42</td>
</tr>
</tbody>
</table>
Reading and Processing Uncommitted Data

<table>
<thead>
<tr>
<th>i++</th>
<th>simultaneous processes(^b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>step</td>
</tr>
<tr>
<td>42</td>
<td>1</td>
</tr>
<tr>
<td>42</td>
<td>2</td>
</tr>
<tr>
<td>42</td>
<td>3</td>
</tr>
<tr>
<td>42</td>
<td>4</td>
</tr>
<tr>
<td>43</td>
<td>5</td>
</tr>
<tr>
<td>43</td>
<td>6</td>
</tr>
</tbody>
</table>

\(^6\)Each process has a software prototype of %eax available.

Reading and Processing Uncommitted Data

<table>
<thead>
<tr>
<th>i++</th>
<th>simultaneous processes(^b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>step</td>
</tr>
<tr>
<td>42</td>
<td>1</td>
</tr>
<tr>
<td>42</td>
<td>2</td>
</tr>
<tr>
<td>42</td>
<td>3</td>
</tr>
<tr>
<td>42</td>
<td>4</td>
</tr>
<tr>
<td>43</td>
<td>5</td>
</tr>
<tr>
<td>43</td>
<td>6</td>
</tr>
</tbody>
</table>

\(^6\)Each process has a software prototype of %eax available.

Reading and Processing Uncommitted Data

<table>
<thead>
<tr>
<th>i++</th>
<th>simultaneous processes(^b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>step</td>
</tr>
<tr>
<td>42</td>
<td>1</td>
</tr>
<tr>
<td>42</td>
<td>2</td>
</tr>
<tr>
<td>42</td>
<td>3</td>
</tr>
<tr>
<td>42</td>
<td>4</td>
</tr>
<tr>
<td>43</td>
<td>5</td>
</tr>
<tr>
<td>43</td>
<td>6</td>
</tr>
</tbody>
</table>

\(^6\)Each process has a software prototype of %eax available.

- dirty read or lost update—that is the legitimate question
- \(\text{step 3}\) reads \(i\) although it was (logically) already dirtied in \(\text{step 2}\)
- \(\text{step 6}\) writes \(i\) unaware of the update already committed in \(\text{step 5}\)
Reading and Processing Uncommitted Data

Simultaneous processes:

<table>
<thead>
<tr>
<th>i</th>
<th>step</th>
<th>action</th>
<th>%eax</th>
<th>action</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>1</td>
<td>movl i, %eax</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>2</td>
<td>addl $1, %eax</td>
<td>43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>3</td>
<td></td>
<td></td>
<td>movl i, %eax</td>
<td>42</td>
</tr>
<tr>
<td>42</td>
<td>4</td>
<td></td>
<td></td>
<td>addl $1, %eax</td>
<td>43</td>
</tr>
<tr>
<td>43</td>
<td>5</td>
<td></td>
<td></td>
<td>movl %eax, i</td>
<td>43</td>
</tr>
<tr>
<td>43</td>
<td>6</td>
<td>movl %eax, i</td>
<td>43</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dirty read or lost update—that is the legitimate question:
- Step 3 reads i although it was (logically) already dirtied in step 2
- Step 6 writes i unaware of the update already committed in step 5
- Both is true, each one ends in a reading of one less than expected
  - i++ twice-executed but its value reads as only once-counted

Each process has a software prototype of %eax available.

Serialisability Unresolved

Procedural abstraction:
- Reduction to an ELOP: AAF
  - Add and fetch
  - Atomic read-modify-write
- With GCC atomic built-in function:
  #define AAF __sync_add_and_fetch

Correct (indivisible) update of the global counter variable total:

```c
#include "aaf.h"
extern int total;

int vary(int i) {
    return AAF(&total, i);
}
```

But parallel actions `vary` and `init` are not equivalent to a serial schedule:

```c
movl 4(%esp), %ecx
movl %ecx, %eax
lock
xaddl %eax, total
addl %ecx, %eax
ret
```

Init (cf. p. 22) may be carried out either before or after lines 11–12
- Consequently, the final value of total remains indeterminate
  - It may read i or total + i, depending on which action came first.