Concurrent Systems

Nebenläufige Systeme

IV. Critical Sections

Wolfgang Schröder-Preikschat

October 30, 2014

Agenda

Preface

Fundamentals
- Race Condition
- Sequential Control
- Concurrent Control

Patterns
- Data Race
- Control-Flow Race

Summary

Outline

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

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

- 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

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

- 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

Synchronisation v. Level of Abstraction of a Critical Section

i providing functionality as integral part of a critical section *or*

ii providing the same functionality on the basis of a critical section

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.

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

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" overleaps time data

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" overleaps time data
- **prevention** of especially critical races is a must, implicitly or explicitly
  - but before, the race condition behind must be even localised...

Heisenbug

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

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 <u>all</u> actions is defined by a **partial order** (cf. [9, p. 17])

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 <u>all</u> actions is defined by a **partial order** (cf. [9, p. 17])
- <u>loss of the total order</u> 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**

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

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 <u>all</u> actions is defined by a **partial order** (cf. [9, p. 17])
- <u>loss of the total order</u> 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.

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

Critical Section

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 (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

Critical Section

Definition (following computer science folklore)

A sequence of program statements controlled by a <u>sequential process</u> 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.

- 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.

A Matter of Reference

Level 2 or 3 ELOP? (cf. [9, p. 34])

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
  - abstraction used to represent both the list head and a list element:

```
typedef struct chain {
  struct chain *link;
} chain_t;
```

  - prepend specified *item* to *this* list (stack policy):

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

[1] *last in, first out*

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.

- by way of example, LIFO[1] storing (*push*) with a singly-linked list
  - abstraction used to represent both the list head and a list element:

```
typedef struct chain {
  struct chain *link;
} chain_t;
```

  - prepend specified *item* to *this* list (stack policy):

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

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

[1] *last in, first out*

- "take a sledgehammer to crack a nut"[2]: **blocking synchronisation**
  - extended (new) abstraction used to represent the list head:

```
typedef struct mutex_chain {
  chain_t head;
  pthread_mutex_t lock;
} mutex_chain_t;
```

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

```
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);
}
```

[2](Ger.) *mit Kanonen auf Spatzen schießen*.

- "take a sledgehammer to crack a nut"[2]: **blocking synchronisation**
  - extended (new) abstraction used to represent the list head:

```
typedef struct mutex_chain {
  chain_t head;
  pthread_mutex_t lock;
} mutex_chain_t;
```

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

```
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);
}
```

- *lock* blocks simultaneous processes out from *push*, till after *unlock*

[2](Ger.) *mit Kanonen auf Spatzen schießen*.

- avant-garde: **non-blocking synchronisation** of the *push* function
  - reused abstraction, for both list head and list element:

```
typedef struct chain {
  struct chain *link;
} chain_t;
```

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

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

Instruction Set Architecture Level

- avant-garde: **non-blocking synchronisation** of the *push* function
  - reused abstraction, for both list head and list element:

```
typedef struct chain {
  struct chain *link;
} chain_t;
```

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

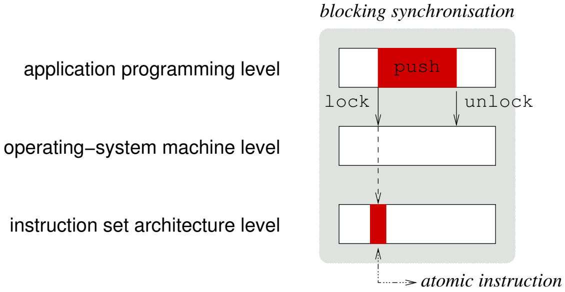
```
void cas_push(chain_t *this, chain_t *item) {
  do item->link = this->link;
  while (!CAS(&this->link, item->link, item));
}
```

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

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

  - compiled by GCC into (x86) "`lock cmpxchgl`" instruction/ELOP

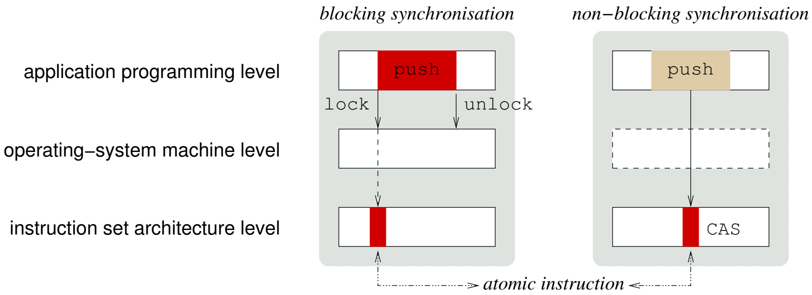
A Matter of Reference: Reconsidered



- *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*

[3] Atomic write or atomic *read-modify-write*, depending on the lock protocol.

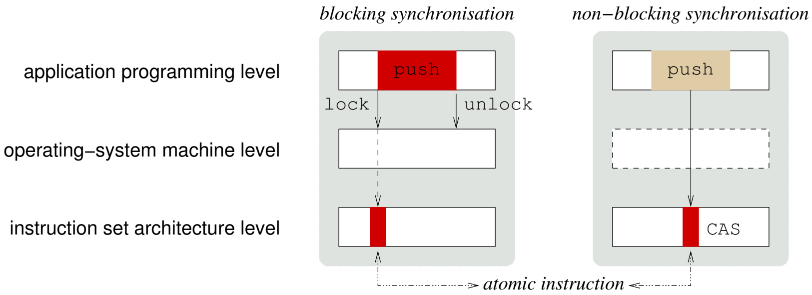
A Matter of Reference: Reconsidered



- *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.

A Matter of Reference: Reconsidered



- *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
- *push* is a **physical** (left-hand) or **logical** (right-hand) critical section

[3] Atomic write or atomic *read-modify-write*, depending on the lock protocol.

The Problem of Locks[4]

cf. [7, p. 58–59]

- 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

[4] This includes all variants of locks as well as other concepts of blocking synchronisation, particularly binary semaphore, mutex, or bolt variable resp.

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**

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

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:

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

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
  - 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

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
  - 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
- transactions may be concurrent, but not necessarily the work units

Devoid of Obstruction

Definition (Optimistic Concurrency Control [8])

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

Devoid of Obstruction

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

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
    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.

ACID Properties

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

- **reliability attributes** according to [4] or [5, p. 289–290], resp.
  - 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

ACID Properties

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

- **reliability attributes** according to [4] or [5, p. 289–290], resp.
  - 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
- 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**

Transaction Isolation

Separation of **simultaneous transactions** in a manner that none of them falls in an undefined state or causes permanent inconsistencies.

Transaction Isolation

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]:[5]

non-repeatable read
- repeated read actions deliver different values
- *read* $\rightarrow$ *write* dependency

dirty read
- not yet committed data is already (early) read
- *write* $\rightarrow$ *read* dependency

lost update
- already committed data is overwritten
- *write* $\rightarrow$ *write* dependency

[5] Concurrency is not affected by the order of read actions, which in particular commute: *read* $\rightarrow$ *read* dependency is thus excluded.

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]:[5]

  - 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

[5] Concurrency is not affected by the order of read actions, which in particular commute: *read* → *read* dependency is thus excluded.

Transaction Isolation

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]:[5]

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 data base systems

[5] Concurrency is not affected by the order of read actions, which in particular commute: *read* → *read* dependency is thus excluded.

Outline

Preface

Fundamentals
- Race Condition
- Sequential Control
- Concurrent Control

Patterns
- Data Race
- Control-Flow Race

Summary

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

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:

```
1 int twofold() {
2     extern volatile int i;
3
4     return i + i;
5 }
```

```
6 twofold:
7     movl i, %eax
8     addl i, %eax
9     ret
```

- 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

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:

```
1 int twofold() {
2     extern volatile int i;
3
4     return i + i;
5 }
```

```
6 twofold:
7     movl i, %eax
8     addl i, %eax
9     ret
```

- apparently correct, but...

↪ 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:

```
1 int twofold() {
2     extern volatile int i;
3
4     return i + i;
5 }
```

```
6 twofold:
7     movl i, %eax
8     addl i, %eax
9     ret
```

- apparently correct, but...

↪ 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

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:

```
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

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:

```
1  char string[80];
2
3  int fill(char c) {
4      int i;
5      for (i = 0; i < 80; i++) {
6          if (!string[i]) {
7              string[i] = c;
8              return i;
9          }
10     }
11     return -1;
12 }
```

- apparently correct, but...
- that *i* identified the first free entry in line 6
- must no longer hold in line 7

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:

```
1  char string[80];
2
3  int fill(char c) {
4      int i;
5      for (i = 0; i < 80; i++) {
6          if (!string[i]) {
7              string[i] = c;
8              return i;
9          }
10     }
11     return -1;
12 }
```

- apparently correct, but...
- that *i* identified the first free entry in line 6
- must no longer hold in line 7

↪ risk of error propagation

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:

```
1  char string[80];
2
3  int fill(char c) {
4      int i;
5      for (i = 0; i < 80; i++) {
6          if (!string[i]) {
7              string[i] = c;
8              return i;
9          }
10     }
11     return -1;
12 }
```

- 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.

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++

```
i++;
```

ASM

```
movl i, %eax
addl $1, %eax
movl %eax, i
```

  - apparently correct

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++

```
1 i++;
```

- apparently correct, but…

ASM

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

- 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, p. 14–15]

C/C++

```
1 i++;
```

- apparently correct, but...

↪ risk of wrong reading (cf. p. 31)

ASM

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

- 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, p. 14–15]

C/C++

```
1 i++;
```

- apparently correct, but...

↪ risk of wrong reading (cf. p. 31)

ASM

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

- 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)

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:

```
int total = 0;

int vary(int i) {
    return total += i;
}

void init(int i) {
    total = i;
}
```

  - apparently correct

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:

```
1  int total = 0;
2
3  int vary(int i) {
4      return total += i;
5  }
6
7  void init(int i) {
8      total = i;
9  }
```

```
10 vary:
11     movl total, %eax
12     addl 4(%esp), %eax
13     movl %eax, total
14     ret
15
16 init:
17     movl 4(%esp), %eax
18     movl %eax, total
19     ret
```

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

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:

```
1  int total = 0;
2
3  int vary(int i) {
4      return total += i;
5  }
6
7  void init(int i) {
8      total = i;
9  }
```

```
10 vary:
11     movl total, %eax
12     addl 4(%esp), %eax
13     movl %eax, total
14     ret
15
16 init:
17     movl 4(%esp), %eax
18     movl %eax, total
19     ret
```

- apparently correct, but...

↪ risk of blind writes

- the write in line 13 is unaware of the write in line 18

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:

```
1  int total = 0;
2
3  int vary(int i) {
4      return total += i;
5  }
6
7  void init(int i) {
8      total = i;
9  }
```

```
10 vary:
11     movl total, %eax
12     addl 4(%esp), %eax
13     movl %eax, total
14     ret
15
16 init:
17     movl 4(%esp), %eax
18     movl %eax, total
19     ret
```

- apparently correct, but...

↪ risk of blind writes

- the write in line 13 is unaware of the write in line 18

Solution Statement: same as "dirty read", but this problem is <u>not serializable</u>

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 <u>two</u> interacting processes:

check condition and block

```
while (this->load == 0)
    await(this);
```

invalidate condition and proceed

```
if (this->load++ == 0)
    cause(this);
```

- apparently correct

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 <u>two</u> interacting processes:

check condition and block

```
1 while (this->load == 0)
2     await(this);
```

- apparently correct, but...

invalidate condition and proceed

```
3 if (this->load++ == 0)
4     cause(this);
```

- lines 3–4 can overlap adversely with lines 1–2
- unblocking (line 4) sidesteps the process that will block (line 2)

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:

check condition and block

```
1 while (this->load == 0)
2     await(this);
```

- apparently correct, but...

↪ risk of everlasting block

invalidate condition and proceed

```
3 if (this->load++ == 0)
4     cause(this);
```

- lines 3–4 can overlap adversely with lines 1–2
- unblocking (line 4) sidesteps the process that will block (line 2)

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 <u>two</u> interacting processes:

check condition and block

```
1 while (this->load == 0)
2     await(this);
```

- apparently correct, but...

↪ risk of everlasting block

invalidate condition and proceed

```
3 if (this->load++ == 0)
4     cause(this);
```

- lines 3–4 can overlap adversely with lines 1–2
- unblocking (line 4) sidesteps the process that will block (line 2)

Solution Statement (ROP)

1. non-blocking or 2. blocking synchronisation.

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

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

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

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:

```
1  #include <stdio.h>
2  #include <sys/stat.h>
3
4  int main(int argc, char *argv[]) {
5      struct stat buf;
6      if (lstat(argv[1], &buf))
7          perror("lstat failed");
8      else {
9          printf("%s is ", argv[1]);
10         if (buf.st_size != 0)
11             printf("not ");
12         printf("empty\n");
13     }
14 }
```

- apparently correct, but...
- file size may change between lines 6 and 10

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

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
1  #include <stdio.h>
2  #include <sys/stat.h>
3
4  int main(int argc, char *argv[]) {
5      struct stat buf;
6      if (lstat(argv[1], &buf))
7          perror("lstat failed");
8      else {
9          printf("%s is ", argv[1]);
10         if (buf.st_size != 0)
11             printf("not ");
12         printf("empty\n");
13     }
14 }
```

- apparently correct, but...
- file size may change between lines 6 and 10

↪ risk of false positive or negative, resp.

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

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:

```
1  #include <stdio.h>
2  #include <sys/stat.h>
3
4  int main(int argc, char *argv[]) {
5      struct stat buf;
6      if (lstat(argv[1], &buf))
7          perror("lstat failed");
8      else {
9          printf("%s is ", argv[1]);
10         if (buf.st_size != 0)
11             printf("not ");
12         printf("empty\n");
13     }
14 }
```

- apparently correct, but...
- file size may change between lines 6 and 10

↪ risk of false positive or negative, resp.

Solution Statement ☹

blocking synchronisation.

Outline

Preface

Fundamentals
- Race Condition
- Sequential Control
- Concurrent Control

Patterns
- Data Race
- Control-Flow Race

Summary

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, and 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, and 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, and 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, and 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, and 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, and 3. blocking synchronisation

Reference List I

[1] Dijkstra, E. W.:
Structured Programming.
In: Buxton, J. N. (Hrsg.) ; Randell, B. (Hrsg.): *Software Engineering Techniques*.
1970 (Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969), Kapitel 7.4, S. 84–87

[2] Gray, J. N.:
Granularity of Locks and Degrees of Consistency in a Large Shared Data Base.
In: *Proceedings of the IFIP Working Conference on Modelling in Data Base Management Systems*.
New York, NY, USA : Elsevier/North-Holland, 1976, S. 365–395

[3] Gray, J. N.:
Notes on Data Base Operating Systems.
In: Bayer, R. (Hrsg.) ; Graham, R. M. (Hrsg.) ; Seegmüller, G. (Hrsg.): *Operating Systems – An Advanced Course* Bd. 60, Springer-Verlag, 1978 (Lecture Notes in Computer Science). –
ISBN 978–3–540–08755–7, S. 393–481

Reference List II

[4] Gray, J. N.:
The Transaction Concept: Virtues and Limitations.
In: *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB 1981)*, IEEE Computer Society, 1981 (Invited Paper), S. 144–154

[5] Haerder, T. ; Reuter, A. :
Principles of Transaction-Oriented Database Recovery.
In: *Computing Surveys* 15 (1983), Dez., Nr. 4, S. 287–317

[6] Herrtwich, R. G. ; Hommel, G. :
Kooperation und Konkurrenz – Nebenläufige, verteilte und echtzeitabhängige Programmsysteme.
Springer-Verlag, 1989. –
ISBN 3–540–51701–4

[7] Jim Larus, H. S.:
Software and the Concurrency Revolution.
In: *Queue – Multiprocessors* 3 (2005), Sept., Nr. 7, S. 54–62

[8] Kung, H.-T. ; Robinson, J. T.:
On Optimistic Methods for Concurrency Control.
In: *ACM Transactions on Database Systems* 6 (1981), Jun., Nr. 2, S. 213–226

Reference List III

[9] Schröder-Preikschat, W. :
Processes.
In: Lehrstuhl Informatik 4 (Hrsg.): *Concurrent Systems*.
FAU Erlangen-Nürnberg, 2014 (Lecture Slides), Kapitel 3

[10] Unger, S. H.:
Hazards, Critical Races, and Metastability.
In: *IEEE Transactions on Computers* 44 (1995), Jun., Nr. 6, S. 754–768

Simultaneous Push Considered Harmful

push	simultaneous processes			
	$A : push(list, X)$		$B : push(list, Y)$	
list	action	*item*	action	*item*
Z	`item->link = this->link`	$X \rightarrow Z$		–
Z		–	`item->link = this->link`	$Y \rightarrow Z$
$Y \rightarrow Z$		–	`this->link = item`	$Y \rightarrow Z$
$X \rightarrow Z$	`this->link = item`	$X \rightarrow Z$		$Y \rightarrow Z$

- 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

push	simultaneous processes			
	$A : push(list, X)$		$B : push(list, Y)$	
list	action	*item*	action	*item*
Z	`item->link = this->link`	$X \to Z$		–
Z		–	`item->link = this->link`	$Y \to Z$
$Y \to Z$		–	`this->link = item`	$Y \to Z$
$X \to Z$	`this->link = item`	$X \to Z$		$Y \to Z$

- 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

push	simultaneous processes			
	$A : push(list, X)$		$B : push(list, Y)$	
list	action	*item*	action	*item*
Z	`item->link = this->link`	$X \to Z$		–
Z		–	`item->link = this->link`	$Y \to Z$
$Y \to Z$		–	`this->link = item`	$Y \to Z$
$X \to Z$	`this->link = item`	$X \to Z$		$Y \to Z$

- 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: $Y \to Z$

Simultaneous Push Considered Harmful

push	simultaneous processes			
	A : *push*(*list*, X)		B : *push*(*list*, Y)	
list	action	*item*	action	*item*
Z	`item->link = this->link`	$X \rightarrow Z$		–
Z		–	`item->link = this->link`	$Y \rightarrow Z$
$Y \rightarrow Z$		–	`this->link = item`	$Y \rightarrow Z$
$X \rightarrow Z$	`this->link = item`	$X \rightarrow Z$		$Y \rightarrow Z$

- 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: $Y \rightarrow Z$
  - 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: $X \rightarrow Z$

Simultaneous Push Considered Harmful

push	simultaneous processes			
	$A : push(list, X)$		$B : push(list, Y)$	
list	action	*item*	action	*item*
Z	`item->link = this->link`	$X \to Z$		–
Z		–	`item->link = this->link`	$Y \to Z$
$Y \to Z$		–	`this->link = item`	$Y \to Z$
$X \to Z$	`this->link = item`	$X \to Z$		$Y \to Z$

- 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: $Y \to Z$
  - 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: $X \to Z$

$\hookrightarrow$ two *push* operations executed, but only one more item was stored...

Reading and Processing Uncommitted Data

i++		simultaneous processes[6]			
		A		*B*	
i	step	action	%eax	action	%eax
42					

[6]Each process has a *software prototype* of `%eax` available.

Reading and Processing Uncommitted Data

i++		simultaneous processes[6]			
		A		*B*	
i	step	action	%eax	action	%eax
42	1	`movl i, %eax`	42		–

[6] Each process has a *software prototype* of `%eax` available.

Reading and Processing Uncommitted Data

i++		simultaneous processes[6]			
		A		*B*	
i	step	action	%eax	action	%eax
42	1	`movl i, %eax`	42		–
42	2	`addl $1, %eax`	43		–

[6] Each process has a *software prototype* of `%eax` available.

Reading and Processing Uncommitted Data

i++		simultaneous processes[6]			
		A		*B*	
i	step	action	%eax	action	%eax
42	1	`movl i, %eax`	42		–
42	2	`addl $1, %eax`	43		–
42	3		–	`movl i, %eax`	42

[6] Each process has a *software prototype* of %eax available.

Reading and Processing Uncommitted Data

i++		simultaneous processes[6]			
		A		*B*	
i	step	action	%eax	action	%eax
42	1	`movl i, %eax`	42		–
42	2	`addl $1, %eax`	43		–
42	3		–	`movl i, %eax`	42
42	4		–	`addl $1, %eax`	43

[6] Each process has a *software prototype* of `%eax` available.

Reading and Processing Uncommitted Data

`i++`		simultaneous processes[6]			
		A		*B*	
i	step	action	%eax	action	%eax
42	1	`movl i, %eax`	42		–
42	2	`addl $1, %eax`	43		–
42	3		–	`movl i, %eax`	42
42	4		–	`addl $1, %eax`	43
43	5		–	`movl %eax, i`	43

[6] Each process has a *software prototype* of %eax available.

Reading and Processing Uncommitted Data

i++		simultaneous processes[6]			
		A		B	
i	step	action	%eax	action	%eax
42	1	`movl i, %eax`	42		–
42	2	`addl $1, %eax`	43		–
42	3		–	`movl i, %eax`	42
42	4		–	`addl $1, %eax`	43
43	5		–	`movl %eax, i`	43
43	6	`movl %eax, i`	43		–

[6] Each process has a *software prototype* of %eax available.

Reading and Processing Uncommitted Data

`i++`		simultaneous processes[6]			
		A		*B*	
i	step	action	`%eax`	action	`%eax`
42	1	`movl i, %eax`	42		–
42	2	`addl $1, %eax`	43		–
42	3		–	`movl i, %eax`	42
42	4		–	`addl $1, %eax`	43
43	5		–	`movl %eax, i`	43
43	6	`movl %eax, i`	43		–

- 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

[6] Each process has a *software prototype* of `%eax` available.

Reading and Processing Uncommitted Data

i++		simultaneous processes[6]			
		A		*B*	
i	step	action	%eax	action	%eax
42	1	`movl i, %eax`	42		–
42	2	`addl $1, %eax`	43		–
42	3		–	`movl i, %eax`	42
42	4		–	`addl $1, %eax`	43
43	5		–	`movl %eax, i`	43
43	6	`movl %eax, i`	43		–

- 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

[6] Each process has a *software prototype* of `%eax` available.

Serialisability Unresolved

- correct (indivisible) update of the global counter variable *total*:

```
#include "aaf.h"

extern int total;

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

- 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`

Serialisability Unresolved

- correct (indivisible) update of the global counter variable *total*:

```
1  #include "aaf.h"
2
3  extern int total;
4
5  int vary(int i) {
6      return AAF(&total, i);
7  }
```

- 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`

- but parallel actions *vary*||*init* are not equivalent to a serial schedule:

```
8  vary:
9      movl   4(%esp), %ecx
10     movl   %ecx, %eax
11     lock
12     xaddl  %eax, total
13     addl   %ecx, %eax
14     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