The Theory & Practice of Concurrent Programming

Lectured by Azalea Raad and Alastair Donaldson

Typed by Aris Zhu Yi Qing

November 8, 2021

1 Synchronisation Paradigms

1.1 Properties in Asynchronous computation

1. Safety
   - Nothing bad happens ever
   - If it is violated, it is done by a finite computation
2. Liveness
   - Something good happens eventually
   - Cannot be violated by a finite computation

1.2 Problems in Asynchronous computation

1. Mutual Exclusion (Safety)
   - **cannot** be solved by transient communication or interrupts
   - **can** be solved by shared variables that can be read or written
2. No Deadlock (Liveness): Some event A eventually happens.

1.3 Protocols in Asynchronous computation

1. Flag Protocol (from B's perspective):
   - Raise flag
   - While A's flag is up
     - Lower flag
     - Wait for A's flag to go down
     - Raise flag
   - Do something
   - Lower flag
2. Producer/Consumer:
   - For A(producer), while flag is up wait. So when flag becomes down, do something, then raise the flag.
   - For B(consumer), while flag is down, wait. So when flag becomes up, do something, then put down the flag.
3. Readers/Writers:
   - Each thread `i` has `size[i]` counter. Only it increments or decrements.
   - To get object's size, a thread reads a "snapshot" of all counters.
   - This eliminates the bottleneck of "having exclusive access to the common counter".

1.4 Performance Measurement

Amdahl's law:

$$\text{Speedup} = \frac{\text{1-thread execution time}}{n\text{-thread execution time}} = \frac{1}{1 - p + \frac{p}{n}},$$

where p is the fraction of the algorithm having parallel execution, and n is the number of threads.

2 Concurrent Semantics

Notation

- $x, y, z, \dots$ shared memory locations
- $a, b, c, \dots$ private registers
- $E, E_1, \dots$ expressions over values (integers) and registers
- $a := x$ **read** from location x into register a
- $x := a$ **write** contents of register a to location x
- $a := E$ **assignment**: compute E and write it to a

ConWhile concurrent programming language

$$
\begin{array}{rcll}
B \in \text{Bool} & ::= & \texttt{true} \,|\, \texttt{false} \,|\, \dots & \\
E \in \text{Exp} & ::= & \dots \,|\, E + E \,|\, \dots & \\
C \in \text{Com} & ::= & a := E & \text{assignment} \\
& & |\, a := x & \text{(memory) read} \\
& & |\, x := a & \text{(memory) write} \\
& & |\, a := \texttt{CAS}(x, E, E) \,|\, \texttt{FAA}(x, E) & \text{(memory) RMWs} \\
& & |\, \texttt{skip} \,|\, C;\ C \,|\, \texttt{while } B \texttt{ do } C & \\
& & |\, \texttt{if } B \texttt{ then } C \texttt{ else } C, & \\
& & |\, \texttt{mfence} & \text{memory fence (TSO only)}
\end{array}
$$

where `FAA` (fetchAndAdd) is considered *weak* RMW because it enables synchronisation between two threads only, whereas `CAS` (compareAndSet) is considered *strong* RMW because it enables synchronisation among an arbitrary number of threads.

2.1 Sequential Consistency (SC)

Also called Interleaving Semantics. The instructions of each thread are executed in order. Instructions of different threads interleave arbitrarily.

Model Definitions

- We model ConWhile concurrent program as a map from thread identifiers ($\tau \in \text{Tid}$) to sequential commands:

$$P \in \text{Prog} \triangleq \text{Tid} \to \text{Com}.$$

- We use $\|$ notation for concurrent programs and write

$$C_1 \parallel C_2 \parallel \dots \parallel C_n$$

  for the n-threaded program P with

$$\text{dom}(P) = \{\tau_1, \dots, \tau_n\}$$

  and $P(\tau_i) = C_i$ for $i \in \{1, \dots, n\}$.

- For instance, we write $\text{dom}(P_{\text{sb}}) = \{\tau_1, \tau_2\}$, with $P_{\text{sb}}(\tau_1) = x := 1; a := y;$ and $P_{\text{sb}}(\tau_2) = y := 1; b := x;$, therefore

$$P_{\text{sb}} \quad \triangleq \quad x := 1; a := y; \ \parallel\ y := 1; b := x; \ .$$

- We model the shared memory as a map from locations to values:

$$M \in \text{Mem} \triangleq \text{Loc} \to \text{Val},$$

  where Val denotes the set of all values, including integer and Boolean values.

- We define store as a map from registers to values:

$$s \in \text{Store} \triangleq \text{Reg} \to \text{Val}.$$

- We define store map associating each thread with its private store:

$$S \in \text{SMap} \triangleq \text{Tid} \to \text{Store}.$$

- An SC configuration is a triple, (P, S, M), comprising a program P to be executed, the store map S, and the shared memory M.
- The program transitions describe the steps in program executions.
- The storage transitions describe how instructions interact with the storage (memory) system.
- An SC transition label, $l \in \text{Lab}$, may be:
  - the *empty* label ϵ to denote a silent transition
  - a *read* label (R, x, v) to denote reading value v from memory location x
  - a *write* label (W, x, v) to denote writing value v to memory location x
  - a *successful RMW* label $(\text{RMW}, x, v_0, v_n)$ to denote updating the value of location x to v_n when the old value of x is v_0

  - a *failed RMW* label $(\mathrm{RMW}, x, v_0, \bot)$ to denote a failed `CAS` instruction where the old value of x does not match v_0.

- Assume that store s has the mapping for all Boolean expressions B and program expressions E.

- SC Sequential Transitions (Familiar Cases):

$$\frac{C_1, s \xrightarrow{l}_c C_1', s'}{C_1; C_2, s \xrightarrow{l}_c C_1'; C_2, s'} \qquad \frac{}{\texttt{skip}; C, s \xrightarrow{\epsilon}_c C, s}$$

$$\frac{s(B) = \texttt{true}}{\texttt{if}\, B \,\texttt{then}\, C_1 \,\texttt{else}\, C_2, s \xrightarrow{\epsilon}_c C_1, s} \qquad \frac{s(B) = \texttt{false}}{\texttt{if}\, B \,\texttt{then}\, C_1 \,\texttt{else}\, C_2, s \xrightarrow{\epsilon}_c C_2, s}$$

$$\frac{}{\texttt{while}\, B \,\texttt{do}\, C, s \xrightarrow{\epsilon}_c \texttt{if}\, B \,\texttt{then}\, (C;\, \texttt{while}\, B \,\texttt{do}\, C) \,\texttt{else}\, \texttt{skip}, s}$$

$$\frac{s(E) = v \qquad s' = s[a \mapsto v]}{a := E, s \xrightarrow{\epsilon}_c \texttt{skip}, s'}$$

- SC Sequential Transitions (New Cases):

$$x := a \qquad \frac{s(a) = v}{x := a, s \xrightarrow{(\mathrm{W},x,v)}_c \texttt{skip}, s}$$

$$a := x \qquad \frac{s' = s[a \mapsto v]}{a := x, s \xrightarrow{(\mathrm{R},x,v)}_c \texttt{skip}, s'}$$

$$\texttt{FAA}(x, E) \qquad \frac{s(E) = v \qquad v_n = v_0 + v}{\texttt{FAA}(x, E), s \xrightarrow{(\mathrm{RMW},x,v_0,v_n)}_c \texttt{skip}, s}$$

$$\texttt{CAS}(x, E_0, E_n)\ (\text{success}) \qquad \frac{s(E_0) = v_0 \qquad s(E_n) = v_n \qquad s' = s[a \mapsto 1]}{a := \texttt{CAS}(x, E_0, E_n), s \xrightarrow{(\mathrm{RMW},x,v_0,v_n)}_c \texttt{skip}, s'}$$

$$\texttt{CAS}(x, E_0, E_n)\ (\text{failure}) \qquad \frac{s(E_0) = v_0 \qquad v \neq v_0 \qquad s' = s[a \mapsto 0]}{a := \texttt{CAS}(x, E_0, E_n), s \xrightarrow{(\mathrm{RMW},x,v,\bot)}_c \texttt{skip}, s'}$$

- SC (Concurrent) Program Transitions:

$$\frac{P(\tau) = C \quad S(\tau) = s \quad C, s \xrightarrow{l}_c C', s' \quad P' = P[\tau \mapsto C'] \quad S' = S[\tau \mapsto s']}{P, S \xrightarrow{\tau:l}_p P', S'}$$

- SC Storage Transitions (of the form $M \xrightarrow{\tau:l}_m M'$):

$$\text{Read} \qquad \frac{M(x) = v}{M \xrightarrow{\tau:(\mathrm{R},x,v)}_m M}$$

$$\text{Write} \qquad \frac{M' = M[x \mapsto v]}{M \xrightarrow{\tau:(\mathrm{W},x,v)}_m M'}$$

$$\mathrm{RMW}, x, v_0, v_n \qquad \frac{M(x) = v_0 \qquad M' = M[x \mapsto v_n]}{M \xrightarrow{\tau:(\mathrm{RMW},x,v_0,v_n)}_m M'}$$

$$\mathrm{RMW}, x, v, \bot \qquad \frac{M(x) = v}{M \xrightarrow{\tau:(\mathrm{RMW},x,v,\bot)}_m M'}$$

- SC Operational Semantics:

$$\text{silent transition} \qquad \frac{P, S \xrightarrow{\tau:\epsilon}_p P', S'}{P, S, M \to P', S', M}$$

$$\text{both program and storage systems take the same transition} \qquad \frac{P, S \xrightarrow{\tau:l}_p P', S' \qquad M \xrightarrow{\tau:l}_m M'}{P, S, M \to P', S', M'}$$

- We write $\to^*$ for the reflexive, transitive closure of $\to$.

- SC Traces
  - The initial memory, $M_0 \triangleq \lambda x.0$.
  - The initial store, $s_0 \triangleq \lambda a.0$.
  - The initial store map, $S_0 \triangleq \lambda \tau.s_0$.
  - The terminated program, $P_{\texttt{skip}} \triangleq \lambda \tau.\texttt{skip}$.
  - Given a program P, an **SC-trace** of P is an evaluation path s.t.

$$P, S_0, M_0 \to^* P_{\texttt{skip}}, S, M$$

    where the pair (S, M) denotes an **SC-outcome**.

- SC is **neither** deterministic **nor** confluent.

2.2 Total Store Ordering (TSO)

TSO = SC + write-read reordering. This allows the weak Store Buffering (SB) behaviour. We can stop the reordering by using memory fences or RMWs, which can impede performance.

Model Definitions

- In addition to the concurrent program, shared memory, store, and store map defined in the SC, we have an addition buffer associating each thread, modelled as a FIFO sequence of (delayed) write label:

$$b \in \text{Buff} \triangleq \text{Seq}\,\langle\text{WLab}\rangle \qquad \text{WLab} \triangleq \big\{(\text{W}, x, v) \,|\, x \in \text{Loc} \wedge v \in \text{Val}\big\}\,.$$

  That is, a buffer entry (W, x, v) denotes a delayed write on x with value v.

- We define buffer map associating each thread with its private buffer:

$$B \in \text{BMap} \triangleq \text{Tid} \rightarrow \text{Buff}.$$

- An TSO configuration is a quadruple, (P, S, M, B), comprising the program P to be executed, the store map S, the shared memory M and the buffer map B.

- A TSO transition label, $l \in \text{Lab}$, may be:
  - an SC label, namely ϵ, (R, x, v), (W, x, v), $(\text{RMW}, x, v_0, v_n)$, $(\text{RMW}, x, v_0, \bot)$
  - a *memory fence* label `MF` for executing an `mfence`.

- TSO Sequential Transition (New case):

$$\texttt{mfence} \qquad \frac{}{\texttt{mfence}, s \xrightarrow{\texttt{MF}}_c \texttt{skip}, s}$$

- TSO Program Transitions: the same as SC Program Transition.

- TSO Storage Transitions (of the form $M, B \xrightarrow{\tau:l}_m M', B'$):

Read

$$\frac{B(\tau) = b \qquad \texttt{get}(\text{M}, b, x) = v}{M, B \xrightarrow{\tau:(\text{R},x,v)}_m M, B}, \text{ where}$$

$$\texttt{get}(\text{M}, b, x) \triangleq \begin{cases} v & \text{if } \exists b_1, b_2 \text{ s.t. } b = b_1.(\text{W}, x, v).b_2 \\ & \quad \wedge \neg\exists v' \text{ s.t. } (\text{W}, x, v') \in b_2 \\ M(x) & \text{otherwise} \end{cases}$$

Write

$$\frac{B(\tau) = b \qquad b' = b.(\text{W}, x, v) \qquad B' = B[\tau \mapsto b']}{M, B \xrightarrow{\tau:(\text{W},x,v)}_m M, B'}$$

Memory Fence

$$\frac{B(\tau) = \emptyset}{M, B \xrightarrow{\tau:\texttt{MF}}_c M, B}$$

RMW, x, v_0, v_n

$$\frac{B(\tau) = \emptyset \qquad M(x) = v_0 \qquad M' = M[x \mapsto v_n]}{M, B \xrightarrow{\tau:(\text{RMW},x,v_0,v_n)}_m M', B}$$

$\text{RMW}, x, v, \bot$

$$\frac{B(\tau) = \emptyset \qquad M(x) = v}{M, B \xrightarrow{\tau:(\text{RMW},x,v,\bot)}_m M, B}$$

unbuffer

$$\frac{B(\tau) = (\text{W}, x, v).b \quad M' = M[x \mapsto v] \quad B' = B[\tau \mapsto b]}{M, B \xrightarrow{\tau:\epsilon}_m M', B'}$$

- TSO Operational Semantics:

silent transition in program

$$\frac{P, S \xrightarrow{\tau:\epsilon}_p P', S'}{P, S, M, B \rightarrow P', S', M, B}$$

silent transition in storage system

$$\frac{M, B \xrightarrow{\tau:\epsilon}_m M', B'}{P, S, M, B \rightarrow P, S, M', B'}$$

both program and storage system take the same transition

$$\frac{P, S \xrightarrow{\tau:\epsilon}_p P', S' \quad M, B \xrightarrow{\tau:\epsilon}_m M', B'}{P, S, M, B \rightarrow P', S', M', B'}$$

- We write $\rightarrow^*$ for the reflexive, transitive closure of $\rightarrow$, the same as the SC's.

- TSO Traces
  - In addition to the initial memory, initial store, initial store map, and the terminated program defined in SC, we have the initial buffer map, $B_0 \triangleq \lambda\tau.\emptyset$.
  - Given a program P, the initial TSO-configuration of P is (P, S_0, M_0, B_0).
  - Given a program P, a **TSO-trace** of P is an evaluation path s.t.

$$P, S_0, M_0, B_0 \rightarrow^* P_{\texttt{skip}}, S, M, B_0$$

  where the pair (S, M) denotes a **TSO-outcome**.

- TSO is also **neither** deterministic **nor** confluent.

3 Linearization

Notation

- `A q.enq(x)` Invocation: ⟨thread⟩ ⟨object⟩.⟨method⟩(⟨arguments⟩)
- `A q:void` Response: ⟨thread⟩ ⟨object⟩:⟨result⟩
- H Sequence of invocations and responses, which looks like:

$H =$
```
A q.enq(3)
A q:void
A q.enq(5)
B p.enq(4)
B p:void
B q.deq()
B q:3
```

Definitions

- Invocation and response match if thread and object names agree
- Object Projections:

$H =$
```
A q.enq(3)
A q:void
A q.enq(5)
B p.enq(4)
B p:void
B q.deq()
B q:3
```
$\Longrightarrow$ $H|q =$
```
A q.enq(3)
A q:void
A q.enq(5)


B q.deq()
B q:3
```

- Thread Projections:

$H =$
```
A q.enq(3)
A q:void
A q.enq(5)
B p.enq(4)
B p:void
B q.deq()
B q:3
```
$\Longrightarrow$ $H|B =$
```
B p.enq(4)
B p:void
B q.deq()
B q:3
```

- An invocation is pending if it has no matching response. It may or may not have taken effect.
- A complete subhistory is a history where pending invocations are discarded.
- A sequential history is one whose invocations are always *immediately* followed by their respective responses.
- A well-formed history is one whose per-thread projections are sequential.
- Equivalent histories are those which have the same threads and their per-thread projections are the same.
- A sequential specification is some way of telling whether a single-thread, single-object history, is legal.
- A sequential history H is legal if for every object x, $H|x$ is in the sequential specification for x.
- A method call precedes another if its response event precedes the other's invocation event.
  - Given history H, method executions m_0, m_1 in H, we say

    $$m_0 \to_H m_1$$

    if m_0 precedes m_1.
  - The above relation is a partial order. It is total order if H is sequential.
- History H is **linearizable** if
  - it can be extended to a *complete* history G
  - G is equivalent to a *legal sequential* history S, where $\boldsymbol{\to_G} \subseteq \boldsymbol{\to_S}$.
- Remarks on linearizability:
  - For pending invocations which took effect, keep them, and discard the rest.
  - $\boldsymbol{\to_H}$ stands for the set of all precedence relations in history H.
  - Focus on total(defined in every state) method.
  - Partial methods are equivalent to thread blocking, and blocking is unrelated to synchronisation.
  - We can identify "linearization points" to help check if executions are linearizable. The point

        - is between invocation and response events
        - correspond to the effect of the call
        - "justify" the whole execution

- **Composability Theorem**:

  History H is linearizable $\iff$ $\forall$ object x, $H|x$ is linearizable.

- History H is **sequentially consistent (SC)** if
    - it can be extended to a *complete* history G
    - G is equivalent to a *legal sequential* history S, ~~where $\rightarrow_G \subseteq \rightarrow_S$~~.

- Remarks on SC:
    - *Cannot* re-order operations done by the same thread
    - *Can* re-order non-overlapping operations done by different threads
    - SC is too strong for hardware architecture, yet too weak for software *specification*.
    - SC is useful for abstracting software *implementation*.

- (non-examinable) Progress conditions (from least ideal to most ideal):
    - Deadlock-free: *some* thread trying to acquire the lock eventually succeeds.
    - Starvation-free: *every* thread trying to acquire the lock eventually succeeds.
    - Lock-free: *some* thread calling a method eventually returns.
    - Wait-free: *every* thread calling a method eventually returns.