The Theory & Practice of Concurrent Programming

Lectured by Azalea Raad and Alastair Donaldson

Typed by Aris Zhu Yi Qing

October 19, 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

2.1 Sequential Consistency (SC)

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

Notation

- $x, y, z, \ldots$ shared memory locations
- $a, b, c, \ldots$ private registers
- $E, E_1, \ldots$ 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

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

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.

Model Definitions

- We model ConWhile concurrent programs 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 $\parallel$ notation for concurrent programs and write

$$C_1 \parallel C_2 \parallel \ldots \parallel C_n$$

for the n-threaded program P with

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

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

- 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 $(\text{RMW}, x, v_0, \bot)$ 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):

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

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

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

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

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

- 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 $M \xrightarrow{\tau:l}_m M'$):

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

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

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

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

- 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.