1 Designing Concurrent Programs

A **proc** is one application. A **thread** is a single seq of instrs in a proc. A proc may have multiple threads. Each proc has its own mem space, threads share the mem space of parent proc. **Parallelism** = many physical cores executing threads simultaneously. **Concurrency** means many threads making progress over time, but not necessarily simultaneously.

- **Safety**: Bad thing never happens. Violated by finite computation.
- **Liveness**: Good thing happens eventually. Cant be violated by finite comp.

Critical Sec cant be concurrent. **Amdahl's Law**: Speedup $= \frac{\text{1-thread exec time}}{n\text{-thread exec time}}$ or $\frac{1}{1-p+\frac{p}{n}}$ where p is % parallelisable.

2 Read Modify Write (RMW)

Atomic instr that reads old val and updates mem loc. **Weak** RMW supports 2 threads. **Strong** RMW (**CAS**) supports n.

3 Sequential Consistency (SC)

Only **strong** mem model, inefficient. Executed in *program order* and *interleaved arbitrarily*.

Assume ConWhile $C \in \text{Com}$ and thread $\tau \in \text{Tid}$. A concurrent prog $P \in \text{Prog} \triangleq \text{Tid} \to \text{Com}$. Shared mem $M \in \text{Mem} \triangleq \text{Loc} \to \text{Val}$. **Store map** $S \in \text{SMap} \triangleq \text{Tid} \to \text{Store}$ and **store** $s \in \text{Store} \triangleq \text{Reg} \to \text{Val}$. An **SC configuration** is $\langle P, S, M\rangle$.

An SC **transition label** may be:

- ϵ *silent transition*.
- $(\mathtt{R}, x, v)$ *read* v from x.
- $(\mathtt{W}, x, v)$ *write* v to x.
- $(\mathtt{U}, x, v_o, v_n)$ *RMW* at x from v_o to v_n.
- $(\mathtt{U}, x, v_o, \bot)$ *failed RMW* at x holding v_o.

3.1 Sequential Transitions

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

$$\frac{\text{eval}(s, B) = \mathtt{true}}{B\,?\,C_1 : C_2, s \xrightarrow{\epsilon}_c C_1, s}$$

$$\frac{}{\Updownarrow B : C, s \xrightarrow{\epsilon}_c B\,?\,(C; \Updownarrow B : C) : \mathtt{skip}, s}$$

$$\frac{\text{eval}(s, E) = v \quad s' = s[a \mapsto v]}{a := E, s \xrightarrow{\epsilon}_c \mathtt{skip}, s'}$$

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

$$\frac{\text{eval}(s, E) = v \quad v_n = v_o + v \quad s' = s[a \mapsto v_o]}{a := \mathtt{FAA}(x, E), s \xrightarrow{(\mathtt{U},x,v_o,v_n)}_c \mathtt{skip}, s'}$$

$$\frac{\text{eval}(s, E_e) = v_e \quad v_e = v_o \quad \text{eval}(s, E_n) = v_n \quad s' = s[a \mapsto 1]}{a := \mathtt{CAS}(x, E_e, E_n), s \xrightarrow{(\mathtt{U},x,v_o,v_n)}_c \mathtt{skip}, s'}$$

$$\frac{\text{eval}(s, E_e) = v_e \quad v_e \neq v_o \quad s' = s[a \mapsto 0]}{a := \mathtt{CAS}(x, E_e, E_n), s \xrightarrow{(\mathtt{U},x,v_o,\bot)}_c \mathtt{skip}, s'}$$

3.2 Program Transitions

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

3.3 Storage Transitions

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

$$\frac{M(x) = v_o \quad M' = M[x \mapsto v_n]}{M \xrightarrow{\tau:(\mathtt{U},x,v_o,v_n)}_m M'} \qquad \frac{M(x) = v_o}{M \xrightarrow{\tau:(\mathtt{U},x,v_o,\bot)}_m M}$$

Finally:

$$\frac{P, S \xrightarrow{\tau:\epsilon}_p P', S'}{P, S, M \longrightarrow P', S', M} \qquad \frac{P, S \xrightarrow{\tau:l}_p P', S' \quad M \xrightarrow{\tau:l}_m M'}{P, S, M \longrightarrow P', S', M'}$$

Then $\to^*$ is the *reflexive transitive closure* of $\to$. Now we can define:

- **Initial memory** $M_0 \triangleq \lambda x.0$
- **Initial store** $s_0 \triangleq \lambda a.0$
- **Initial store map** $S_0 \triangleq \lambda \tau.s_0$
- **Terminated program** $P_{\mathtt{skip}} \triangleq \lambda \tau.\mathtt{skip}$
- **Initial SC config** $P_0 \triangleq \langle P, S_0, M_0\rangle$

The **SC trace** of P is an eval path from $P_0 \to^* \langle P_{\mathtt{skip}}, S, M\rangle$ where $\langle S, M\rangle$ is the **SC outcome** of P.

P is **deterministic** if it has *exactly 1* SC trace. **Confluent** if all traces lead to the same outcome. *Determinacy implies confluence, not vice versa.* In general, P is neither.

4 Total Store Ordering (TSO)

TSO allows reordering of *adjacent* instrs:

- **WR** reordering on *different* locs.
- **WR** reordering on *same* locs provided the value of W is inlined into R.

Each τ has private **store buffer**, a FIFO queue temporarily holding writes before they become globally visible.

Memory fence instrs prevent reordering by *locking the thread* (bad!) until store buffer empty. Instead, use RMW that also flushes SB and writes directly to mem.

Now give each thread a SB:

$$B \in \text{BMap} \triangleq \text{Tid} \to \text{Buff}$$

$$b \in \text{Buff} \triangleq \text{Seq}\langle\text{WLab}\rangle$$

$$\text{WLab} \triangleq \{(\mathtt{W}, x, v) \mid x \in \text{Loc}, v \in \text{Val}\}$$

A **TSO config** is $\langle P, S, M, B\rangle$. Then:

4.1 Sequential Transitions

Same as SC with: $\frac{}{\mathtt{mfence}, s \xrightarrow{\mathtt{MF}}_c \mathtt{skip}, s}$

4.2 Storage Transitions

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

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

$$\text{get}(\cdots) \triangleq \begin{cases} v & \text{if } \exists b_1, b_2. \begin{bmatrix} b = b_1 \cdot (\mathtt{W}, x, v) \cdot b_2 \wedge \\ \nexists v'. \left[(\mathtt{W}, x, v') \in b_2\right] \end{bmatrix} \\ M(x) & \text{otherwise} \end{cases}$$

$$\frac{B(\tau) = \emptyset \quad M(x) = v_o \quad M' = M[x \mapsto v_n]}{M, B \xrightarrow{\tau:(\mathtt{U},x,v_o,v_n)}_m M', B}$$

$$\frac{B(\tau) = \emptyset \quad M(x) = v_o}{M, B \xrightarrow{\tau:(\mathtt{U},x,v_o,\bot)}_m M, B} \qquad \frac{B(\tau) = \emptyset}{M, B \xrightarrow{\tau:\mathtt{MF}}_m M, B}$$

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

4.3 Combining Transitions

$$\frac{P, S \xrightarrow{\tau:l}_p P', S'}{P, S, M, B \longrightarrow P', S', M, B} \qquad \frac{M, B \xrightarrow{\tau:l}_m M', B'}{P, S, M, B \longrightarrow P, S, M', B'}$$

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

$\to^*$, M_0, S_0, $P_{\mathtt{skip}}$ as before, **initial buffer map** $B_0 \triangleq \lambda\tau.\emptyset$. A **TSO trace** is $\langle P, S_0, M_0, B_0\rangle \to^* \langle P_{\mathtt{skip}}, S, M, B_0\rangle$.

5 Declarative Semantics

Candidate executions are represented as *graphs* with nodes of **events** and relations for po, rf, etc. An **event** $\langle n, \tau, \ell\rangle$ is:

- $n \in \mathbb{N}$ *unique event ID*.
- $\tau \in \text{Tid} \cup \{0\}$ *(0 for initial writes)*.
- $\ell \neq \epsilon$ *non empty label*.

5.1 Relational Algebra

Given a set A and relations $r, r' \subseteq A \times A$:

- $[A] \triangleq \{\langle a, a\rangle \mid a \in A\}$ **identity relation**.
- $\text{dom}(r) \triangleq \{a \mid \exists b.\langle a, b\rangle \in r\}$ **domain**.
- $\text{rng}(r) \triangleq \{a \mid \exists b.\langle b, a\rangle \in r\}$ **range**.
- $r^{-1} \triangleq \{\langle b, a\rangle \mid \langle a, b\rangle \in r\}$ **inverse**.
- $r; r' \triangleq \{\langle a, c\rangle \mid \exists b.\left(\langle a, b\rangle \in r \wedge \langle b, c\rangle \in r'\right)\}$ **composition**.
- $r^? \triangleq r \cup [\text{dom}(r) \cup \text{rng}(r)]$ **reflexive closure**.
- $r^+ \triangleq \bigcup_{i \in \mathbb{N}} r^i$ where $r^0 \triangleq r$ and $r^{i+1} \triangleq r; r^i$ **transitive closure**.
- $r^* \triangleq (r^+)^?$ **reflexive transitive closure**.
- $\text{irreflexive}(r) \overset{\text{def}}{\Leftrightarrow} \nexists a.\langle a, a\rangle \in r$ **irreflexivity**.
- $\text{acyclic}(r) \overset{\text{def}}{\Leftrightarrow} \text{irreflexive}(r^+)$ **acyclicity**.

- $\text{typ}(\ell) \in \{\mathtt{R}, \mathtt{W}, \mathtt{U}\}$ *denotes type of label*. Similarly, $\text{val}_{\mathtt{r}}(\ell)$ and $\text{val}_{\mathtt{w}}(\ell)$.
- $\text{tid}(e) \triangleq \tau$ and $\text{lab}(e) \triangleq \ell$ for *event* e.
- $A_0 \triangleq \{a \in A \mid \text{tid}(A) = 0\}$ **init events**.
- $A_\tau \triangleq \{a \in A \mid \text{tid}(A) = \tau\}$ **τ's events**.
- $A_x \triangleq \{a \in A \mid \text{loc}(A) = x\}$ **events on loc x**.
- $r_\tau \triangleq r \cap (A_\tau \times A_\tau)$ *restriction of r to τ*.
- $r_x \triangleq r \cap (A_x \times A_x)$ *restriction of r to loc x*.
- $r\mathtt{i} \triangleq \{\langle a, b\rangle \in r \mid \text{tid}(a) = \text{tid}(b) \vee \text{tid}(a) = 0\}$ *restriction of r to* **internal edges**.
- $re \triangleq r \setminus r\mathtt{i}$ *restriction to* **external edges**.
- $r|_{\mathtt{loc}} \triangleq \{\langle a, b\rangle \in r \mid \text{loc}(a) = \text{loc}(b)\}$ *restriction of r to edges on the same location*.

Let $G = \langle E, \text{po}, \mathtt{rf}\rangle$ be a **candidate exec graph**:

- E is a finite set of events.
- **Program Order** $\text{po} \triangleq (\bigcup_{\tau \in \text{Tid}} \text{po}_\tau) \cup (E_0 \times (E \setminus E_0))$ is a *strict total order* on E.
- **Reads From** $\mathtt{rf}$ on E s.t. $\mathtt{rf}^{-1}$ is a function and $\forall \langle w, r\rangle \in \mathtt{rf}.[w \neq r \wedge \text{typ}(w) \in \{\mathtt{W}, \mathtt{U}\} \wedge \text{typ}(r) \in \{\mathtt{R}, \mathtt{U}\} \wedge \text{loc}(w) = \text{loc}(r) \wedge \text{val}_{\mathtt{w}}(w) = \text{val}_{\mathtt{r}}(r)]$.
- $G.\text{E} \triangleq E$, $G.\text{po} \triangleq \text{po}$, $G.\mathtt{rf} \triangleq \mathtt{rf}$
- $G.\mathtt{R} \triangleq \{r \in \text{E} \mid \text{typ}(r) = \mathtt{R}\}$ **read events**.
- $G.\mathtt{W} \triangleq \{w \in \text{E} \mid \text{typ}(w) = \mathtt{W}\}$ **write events**.
- $G.\mathtt{U} \triangleq \{u \in \text{E} \mid \text{typ}(u) = \mathtt{U}\}$ **update events**.
- $G.\mathtt{RU} \triangleq G.\mathtt{R} \cup G.\mathtt{U}$ **read/update events**.
- $G.\mathtt{WU} \triangleq G.\mathtt{W} \cup G.\mathtt{U}$ **write/update events**.

Coherence is *per-location SC*, whereas **release-acquire** also allows mutual exclusion with message passing with observed writes.

- **SC**: *(1)* respect po, *(2)* respect rf, *(3)* always read latest write.
- **SC-ALT**: *(1)* G is complete, *(2)* mo where read-before rb means if a read reads from a write, any later write in mo must come after the read.
- **TSO**: *(1)* total on everything but reads (can be reordered), *(2)* respects po, except for write-read, *(3)* respects rf, *(4)* always read from latest write.
- **TSO-ALT**: *(1)* G is complete, *(2)* internal rf and rb respect po, *(3)* mo where preserved program order ppo relates all non-write-read pairs in po, *(4)* rf and rb must respect mo.
- **COH**: *(1)* G is complete, *(2)* respects po per loc, *(3)* respects rf per loc with no interleaving writes.
- **COH-ALT**: *(1)* G is complete, *(2)* respects mo on the same loc.
- **RA**: *(1)* G is complete, *(2)* respects $\text{hb}_{\mathtt{ra}}$ happens-before, relating events ordered by po or rf on the same loc.

6 C11 Semantics

We give each memory access **mode**:

- **Read** $\mathtt{R}^{m \in \{\mathtt{na}, \mathtt{rlx}, \mathtt{acq}, \mathtt{sc}\}}$
- **Write** $\mathtt{W}^{m \in \{\mathtt{na}, \mathtt{rlx}, \mathtt{rel}, \mathtt{sc}\}}$
- **RMW** $\mathtt{U}^{m \in \{\mathtt{rlx}, \mathtt{acq}, \mathtt{rel}, \mathtt{acq\text{-}rel}, \mathtt{sc}\}}$
- **Fences** $\mathtt{F}^{m \in \{\mathtt{acq}, \mathtt{rel}, \mathtt{acq\text{-}rel}, \mathtt{sc}\}}$

$\text{mod}(e)$ is the mode of event e. Mode strength is ordered as $\mathtt{na} \sqsubset \mathtt{rlx} \sqsubset \mathtt{acq}, \mathtt{rel} \sqsubset \mathtt{acq\text{-}rel} \sqsubset \mathtt{sc}$. For events E:

$$E^{\sqsupseteq m} \triangleq \{e \in E \mid m = \text{mod}(e) \vee m \sqsubset \text{mod}(e)\}$$

G is **C11-consistent** if it is *(1)* complete, *(2)* respects a defined mo with synchronises-with sw defined as:

$$\mathtt{sw} \triangleq \left([\mathtt{WU}^{\sqsupseteq \mathtt{rel}}] \cup [\mathtt{F}^{\sqsupseteq \mathtt{rel}}]; \text{po}\right); \mathtt{rf}^+; \left([\mathtt{WU}^{\sqsupseteq \mathtt{acq}}] \cup \text{po}; [\mathtt{F}^{\sqsupseteq \mathtt{acq}}]\right)$$

Which expands to:

$$\begin{aligned} \mathtt{sw} \triangleq\ & \left[\mathtt{WU}^{\sqsupseteq \mathtt{rel}}\right]; \mathtt{rf}^+; \left[\mathtt{RU}^{\sqsupseteq \mathtt{acq}}\right] \\ & \left[\mathtt{WU}^{\sqsupseteq \mathtt{rel}}\right]; \mathtt{rf}^+; \text{po}; \left[\mathtt{F}^{\sqsupseteq \mathtt{acq}}\right] \\ & \left[\mathtt{F}^{\sqsupseteq \mathtt{rel}}\right]; \text{po}; \mathtt{rf}^+; \left[\mathtt{RU}^{\sqsupseteq \mathtt{acq}}\right] \\ & \left[\mathtt{F}^{\sqsupseteq \mathtt{rel}}\right]; \text{po}; \mathtt{rf}^+; \text{po}; \left[\mathtt{F}^{\sqsupseteq \mathtt{acq}}\right] \end{aligned}$$

$\langle a, b\rangle \in G.\text{race}$ is a **C11-race** if:

- $a \neq b$ *distinct events*.
- $\text{loc}(a) \neq \text{loc}(b)$ *different locs*.
- $\{\text{typ}(a), \text{typ}(b)\} \cap \{\mathtt{W}, \mathtt{U}\} \neq \emptyset$ *≥ 1 is write*.
- $\mathtt{na} \in \{\text{mod}(a), \text{mod}(b)\}$ *≥ 1 is non-atomic*.
- $\{\langle a, b\rangle, \langle b, a\rangle\} \cap G.\text{hb} = \emptyset$ *not related by* hb.

Then G is **C11-racy** and program P has **undefined behaviour**.

7 Concurrent Objects

A **history** is a seq of **invocation** and **response** events. Assuming SC, we can project a concurrent history to a sequential one by only looking at the non-overlapping method calls (the bits protected by the locks). Assume each method takes effect instantaneously at some point between its invocation and response, if this is possible, the object is **linearizable**.

To linearizablability, find **linearization point** for each method call, s.t. the seq history formed by these points satisfies the sequential specification. If this can't be done, the object is not linearizable. *In a history, pick any point between invocation and response as the linearization point.* H is linearisable if extended to G by **appending** or **discarding** *pending invocations* s.t. $G \cong S$ and $\to_G \subseteq \to_S$.

A `q.enq(x)` for invocation, A `q:void` for response. **Legal** if sequential per τ and respects the seq spec of the object. **Equivalent** ($\cong$) if per-thread projections are the same. Method call m_0 **precedes** m_1 ($m_0 \to_H m_1$) if its *response* is before the others *invocation*. Otherwise, they are **overlapping**. $\to_H$ is a *partial order*, and *total* if sequential.
H is linearisable iff $\forall x \in \text{Obj}.H \mid x$ is linearisable (**composability thrm**). H is **sequentially consistent** if by extending to G, $G \cong S$ where S is *legal* & *sequential*. SC is *not composable*.
4 **progress conditions** for conc objs:

- **Deadlock free**: *some* thread acquiring a lock will succeed.
- **Starvation free**: *every* thread acquiring a lock will succeed.
- **Lock free**: *some* thread calling a method will complete.
- **Wait free**: *every* thread calling a method will completre.

8 C++: Threads & Locks

A mutex from `std::mutex` can be `lock` and `unlock`. We can make a **scoped lock** with a *destructor*:

```cpp
class ScopedLock {
  ScopedLock(std::mutex& mtx) : mtx_(mtx) {
    mtx_.lock();
  }
  // End of lifetime, not GC:
  ~ScopedLock() { mtx_.unlock(); }
};
```

Also impl as `std::scoped_lock<std::mutex>`. This is **Resource Acquisition as Initialisation (RAII)**. A **race condition** is **non-deterministic** behaviour. A **data race** occurs when *(1)* distinct threads access a memloc, *(2)* at least one is a write, *(3)* at least one is atomic, *(4)* not ordered by synchronisation. This is **undefined behaviour** - program has no semantics. A `std::unique_lock` similar to scoped, but allows *relocking*, *deferred locking* and *ownership transfer*. A **condition variable**:

```cpp
#include <condition_variable>
std::condition_variable cond_;
cond_.notify_one(); // Wake one thread
cond_.notify_all(); // Wake alll threads
```

To wait on a **condition variable**:

1. Associate a **mutex** with the condvar.
2. Lock mutex with `std::unique_lock`.
3. Call `wait(mutex, predicate)`.

For example, a locking queue:

```cpp
class LockedQueue {
  LockedQueue(size_t s) { contents_.resize(s); }
  void enq(int element) {
    std::unique_lock<std::mutex> lock(mtx_);
    not_full_.wait(lock, [this]() -> bool {
      return count_ < contents_.size(); });
    contents_[tail_] = element;
    tail_ = (tail_ + 1) % contents_.size();
    count_++;
    not_empty_.notify_one(); // Notify waiting deq
  }
  int deq() {
    std::unique_lock<std::mutex> lock(mtx_);
    not_empty_.wait(lock, [this]() -> bool {
      return count_ > 0; });
    int result = contents_[head_];
    head_ = (head_ + 1) % contents_.size();
    count_--;
    not_full_.notify_one(); // Notify waiting enq
    return result;
  }
};
```

9 C++: Atomics

We can perform **RMW**s on atomics:

```cpp
#include <atomic>
void store(T value); // Atomic store
T load(); // Atomic load
T exchange(T value) // Atomic RMW
bool compare_exchange_strong(T& exp, T desired);
bool compare_exchange_weak(T& exp, T desired);
```

Weak version may fail spuriously (behave as if compare failed even if it succeeded). There is also `T fetch_{add,sub,and,or,xor}(T value);`.

We can use a **memory order** in `store`/`load`:

- `memory_order_relaxed`: allow *store buffering*.
- `memory_order_release`: only *stores*.
- `memory_order_acquire`: only *loads*.
- `memory_order_seq_cst`: *sequential consistency*.

Under *release-acquire*, SB flushed only when loading with pending stores on same location.

10 C++: Spinlocks

```cpp
class ExpBackoffSpinlock {
public:
  ExpBackoffSpinlock() : lock_bit_(false) {}
  void lock() {
    const int kMinBackoffIters = 4;
    const int kMaxBackoffIters = 1 << 10;
    int it = kMinBackoffIters;
    while (lock_bit_.exchange(true, mem_ord_acq)) {
      do {
        for (int i = 0; i < it; i++)
          __mm_pause(); // CPU Pause
        it = std::min(it * 2, kMaxBackoffIters);
      } while (lock_bit_.load(mem_ord_rlx));
    }
  }
  void unlock(){lock_bit_.store(false,mem_ord_rel)}
private:
  std::atomic<bool> lock_bit_;
};
```

Ticket lock avoids starvation by serving tickets in order:

```cpp
class TicketLock {
public:
  TicketLock() : next_(0), now_serving_(0) {}
  void lock() {
    const unsigned my_ticket = next_.fetch_add(1);
    while (now_serving_.load() != my_ticket);
  }
  void unlock() {
    now_serving_.store(now_serving_.load() + 1);
  }
private:
  std::atomic<unsigned> next_;
  std::atomic<unsigned> now_serving_;
};
```

11 C++: Futexes

The **futex** syscall works on userspace data:

- `futex_wait(int *p, int v)` returns if $*p \neq v$, otherwise adds thread to a wait queue p.
- `futex_wake(int *p, int n)` wakes n threads waiting on queue p.

To implement a mutex, 3 states: *0* - lock free, *1* - locked no waiters, *2* - locked with waiters. On lock, compare exchange state 0 to 1:

- *On success*, lock is acquired and has no waiters.
- *On failure*, state either 1 or 2. Set to 2 and call `futex_wait`.

When unlocking, state is either 1 or 2:

- *If 1*, set to 0 and return.
- *If 2*, set to 0 and call `futex_wake`.

```cpp
class MutexSmart {
public:
  MutexSmart() : state_(0) {}
  void lock() {
    int old_value = compare_exchange(0, 1);
    if (old_value == 0) return;
    do {
      if (old_value == 2 || compare_exchange(1, 2)
       != 0) {
        syscall(SYS_futex, reinterpret_cast<int*>(&
      state_), FUTEX_WAIT, 2, nullptr, nullptr, 0);
      }
      old_value = compare_exchange(0, 2);
    } while (old_value != 0);
  }
  void unlock() {
    if (state_.exchange(0) == 2) {
      syscall(SYS_futex, reinterpret_cast<int*>(&
      state_), FUTEX_WAKE, 1, nullptr, nullptr, 0);
    }
  }
private:
  int compare_exchange(int exp, int desired) {
    state_.compare_exchange_strong(exp, desired);
    return expected;
  }
  std::atomic<int> state_;
}
```

12 Haskell Concurrency

An **MVar** is a **mutable variable** that is either *empty* (X) or *full* (O). All operations on MVar are atomic:

```haskell
newMVar :: a -> IO (MVar a) -- Create full MVar
newEmptyMVar :: IO (MVar a) -- Create empty MVar
takeMVar -- Block till full, remove & return
putMVar -- Block till empty then write
readMVar -- Block till full then read
```

We can do a **thread join**:

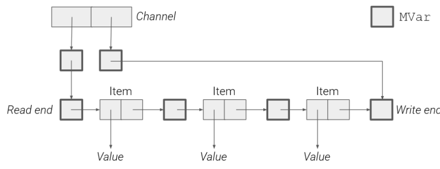
```haskell
printThenJoin s handle = do
  print s
  putMVar handle () -- Thread is done

main = do
  hSetBuffering stdout NoBuffering
  handle1 <- newEmptyMVar
  handle2 <- newEmptyMVar
  forkIO (printThenJoin "I am thread 1" handle1)
  forkIO (printThenJoin "I am thread 2" handle2)
  takeMVar handle1
  takeMVar handle2
  putStrLn "Both threads done"
  return ()
```

We can do a **mutex**:

```haskell
thread :: ... -> MVar () ... -> IO ()
thread ... mutex ... = do
  putMVar mutex () -- Lock
  -- Critical section
  takeMVar mutex -- Unlock
main = do
  mutex <- newEmptyMVar
  forkIO (thread ... mutex ...)
  forkIO (thread ... mutex ...)
```

To replicate a monadic action n times we can do `replicateM :: Monad m => Int -> m a -> m [a]`.

13 Unbounded Channels



Consumers read from the *read end* and **producers** write to the *write end*.

```haskell
data Channel a = Channel (MVar (Stream a)) (MVar (
    Stream a))
type Stream a = MVar (Item a)
data Item a = Item a (Stream a)

newChannel = do
  emptyStream <- newEmptyMVar
  readEnd <- newMVar emptyStream
  writeEnd <- newMVar emptyStream
  return (Channel readEnd writeEnd)
readChannel (Channel readEnd _) = do
  readEndStream <- takeMVar readEnd
  (Item value remainder) <- takeMVar readEndStream
  putMVar readEnd remainder
  return value
writeChannel (Channel _ writeEnd) value = do
  newEmptyStream <- newEmptyMVar
  wEndStream <- takeMVar writeEnd
  putMVar wEndStream (Item value newEmptyStream)
  putMVar writeEnd newEmptyStream
```

14 Datarace Detection

The **vector clock** algorithm uses a vector clock, mapping $\text{Tid} \rightarrow \mathbb{N}$ where each **logical clock** is an integer ≥ 0 incremented when thread releases a mutex. Algorithm state is given by $\langle C, L, R, W \rangle$ where:

- $C : \text{Tid} \rightarrow \text{VC}$ maps *threads* to their vector clocks.
- $L : \text{Locks} \rightarrow \text{VC}$ maps *locks* to their vector clocks.
- $R : \text{Loc} \rightarrow \text{VC}$ maps *mem locs* to the VC of their last *read*.
- $W : \text{Loc} \rightarrow \text{VC}$ maps *mem locs* to the VC of their last *write*.

A thread's clock $C_t = C(t)$ represents what thread t knows about the logical clocks of other threads:

- $C_t(t)$ is *my* clock, **always positive**.
- $\forall u \neq t \in \text{Tid}. C_t(u)$ means *I* know u's clock is $\geq C_t(u)$.
- *When locking, I get info on logical clocks of threads that previously held the lock.*

A lock's clock $L_m = L(m)$ means the last thread to release m knew t's logical clock was $\geq L_m(t)$ when it released the mutex. A loc's clock $R_x = R(x)$ means the last thread to read x knew t's logical clock was $\geq R_x(t)$ when it read. Initially, $\langle \{\mathtt{inc}_0(\bot), \cdots, \mathtt{inc}_{N-1}(\bot)\}, \lambda m.\bot, \lambda x.\bot, \lambda x.\bot \rangle$, then:

14.1 Shared Memory Rules

$$\frac{W_x \sqsubseteq C_t \quad R' = R[x \mapsto R_x[t \mapsto C_t(t)]]}{\langle C, L, R, W \rangle \xrightarrow{\mathtt{rd}(t,x)} \langle C, L, R', W \rangle}$$

$$\frac{W_x \sqsubseteq C_t \quad R_x \sqsubseteq C_t \quad W' = W[x \mapsto W_x[t \mapsto C_t(t)]]}{\langle C, L, R, W \rangle \xrightarrow{\mathtt{wr}(t,x)} \langle C, L, R, W' \rangle}$$

14.2 Lock Rules

$$\frac{C' = C[t \mapsto (C_t \sqcup L_m)]}{\langle C, L, R, W \rangle \xrightarrow{\mathtt{acq}(t,m)} \langle C', L, R, W \rangle}$$

$$\frac{L' = L[m \mapsto C_t] \quad C' = C[t \mapsto \mathtt{inc}_t(C_t)]}{\langle C, L, R, W \rangle \xrightarrow{\mathtt{rel}(t,m)} \langle C', L', R, W \rangle}$$

14.3 Datarace Rules

$$\frac{\exists u. W_x(u) > C_t(u)}{\langle C, L, R, W \rangle \xrightarrow{\mathtt{rd}(t,x)} \mathbf{WriteReadRace}(u, t, x)}$$

$$\frac{\exists u. W_x(u) > C_t(u)}{\langle C, L, R, W \rangle \xrightarrow{\mathtt{wr}(t,x)} \mathbf{WriteWriteRace}(u, t, x)}$$

- The **bottom VC** is $\bot = \{0, 0, \cdots, 0\}$.
- There is a **partial order** on VC: $V_1 \sqsubseteq V_2 \Leftrightarrow \forall t. V_1(t) \leq V_2(t)$.
- To **join** VCs: $V_1 \sqcup V_2 = \{\forall t. \max(V_1(t), V_2(t))\}$.
- An **increment** function $\mathtt{inc}_t(V) = V[t \mapsto V(t) + 1]$.

Although this correct, it is *inefficient*. Better ones exist but arent correct.

15 Rust Concurrency

Rust mutexes **own** their data. An **atomically reference counted (Arc)** object has a *non-owning ref* to T and ref counter x:

- **Cloning** ARC points to the same obj, incrementing x.
- **Dropping** ARC decrements x. When $x = 0$, obj dropped.
- x manipulated with atomics, so its thread safe.

```rust
let mut data = Vec::<u32>::new();
for _ in 0..max { data.push(1); }
let arc_t1 = Arc::new(Mutex::new(data));
let arc_t2: Arc<Mutex<Vec<u32>>> = arc_t1.clone();
let res_arc = Arc::new(AtomicU32::new(0));
let res_arc_t1 = res_arc.clone();
let res_arc_t2 = res_arc.clone();
let t1 = thread::spawn(move || {
  let mut r: u32 = 0;
  for i in 0..(max / 2) {
    r += arc_t1.lock().unwrap()[i];
  }
  res_arc_t1.fetch_add(r, Ordering::Relaxed);
});
let t2 = thread::spawn(move || {
  let mut r: u32 = 0;
  for i in (max / 2)..max {
    r += arc_t2.lock().unwrap()[i];
  }
  res_arc_t2.fetch_add(r, Ordering::Relaxed);
});
t1.join().unwrap();
t2.join().unwrap();
println!("{}", res_arc.load(Ordering::Relaxed));
```