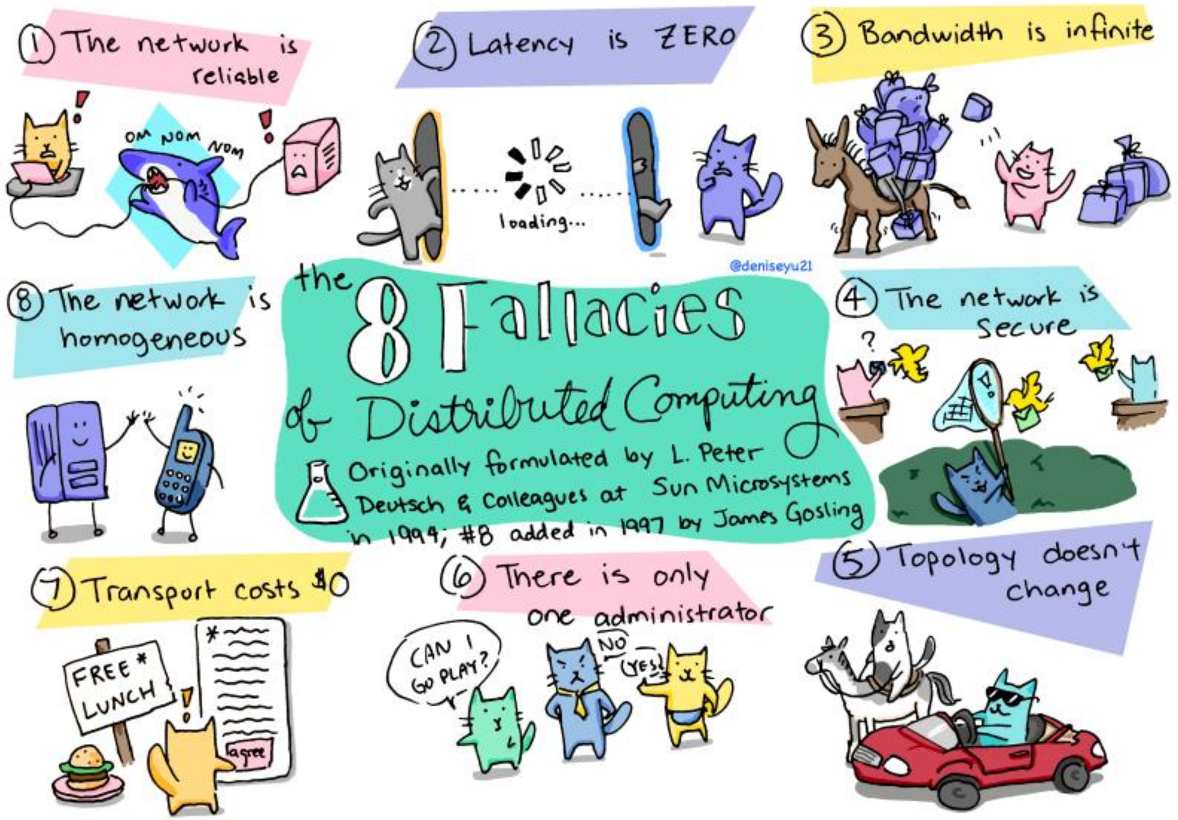
Leslie Lamport papers

- Time, Clocks, and the Ordering of Events in a Distributed System
- How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs
- The Byzantine Generals' Problem
- Distributed Snapshots: Determining Global States of a Distributed System
- The Part-Time Parliament

- Paxos algorithm for consensus.
- Bakery algorithm for mutual exclusion of multiple threads in a computer system that require the same resources at the same time.
- Chandy-Lamport algorithm for the determination of consistent global states (snapshot).
- Lamport signature, one of the prototypes of the digital signature.



Producer-Consumer Synchronization

```
--algorithm PC {
    variables in = Input,  out = ⟨ ⟩, buf = ⟨ ⟩ ;
    fair process (Producer = 0) {
      P: while (TRUE) {
            await Len(buf) < N ;
            buf  := Append(buf, Head (in)) ;
            in   := Tail (in)                    }}
    fair process (Consumer = 1) {
      C: while (TRUE) {
            await Len(buf) > 0 ;
            out  := Append(out, Head(buf)) ;
            buf  := Tail(buf)                    }}  }
```

$$\begin{bmatrix} in = \langle v_1, v_2, \ldots \rangle \\ out = \langle\,\rangle \\ buf = \langle\,\rangle \end{bmatrix} \xrightarrow{P} \begin{bmatrix} in = \langle v_2, v_3, \ldots \rangle \\ out = \langle\,\rangle \\ buf = \langle v_1 \rangle \end{bmatrix} \xrightarrow{C}$$

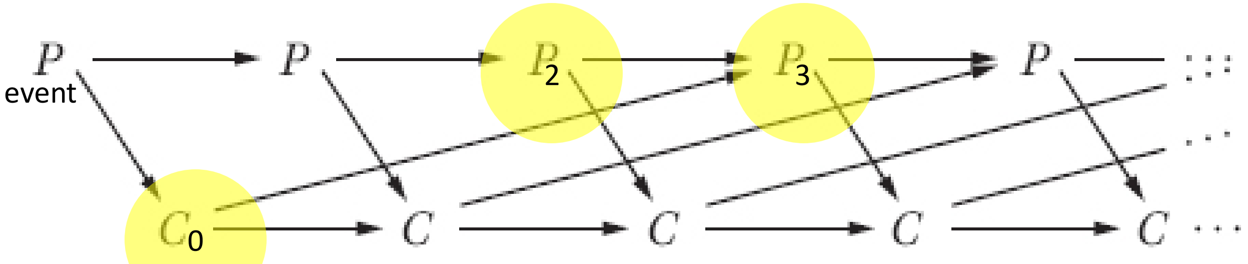
$$\begin{bmatrix} in = \langle v_2, v_3, \ldots \rangle \\ out = \langle v_1 \rangle \\ buf = \langle\,\rangle \end{bmatrix} \xrightarrow{P} \begin{bmatrix} in = \langle v_3, v_4, \ldots \rangle \\ out = \langle v_1 \rangle \\ buf = \langle v_2 \rangle \end{bmatrix} \xrightarrow{P} \ldots$$

An execution of the FIFO queue in standard model.

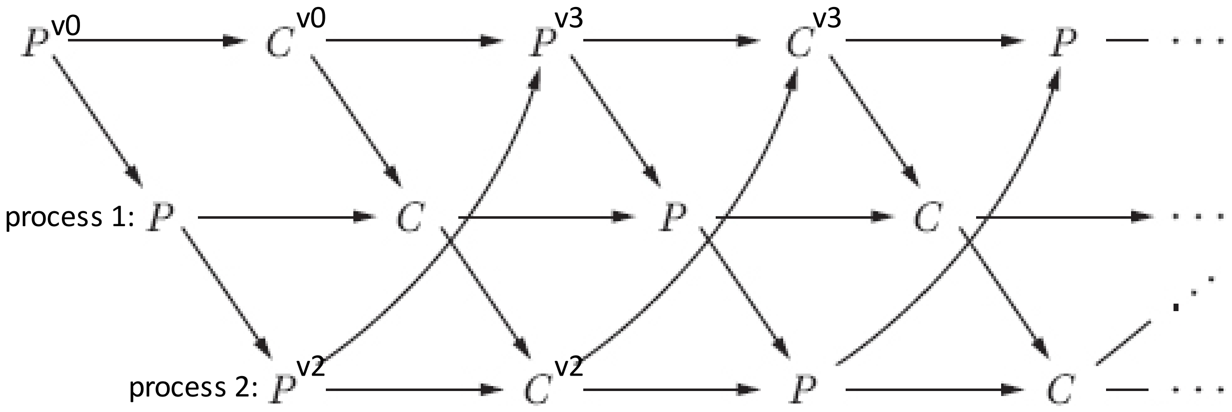
Producer-Consumer: specification ✓
Bounded FIFO queue: implementation ? (prove its correct)

A state predicate is an **invariant** iff it is true in every state of every execution.
$(Len(buf) \leq N) \ \wedge \ (Input = out \circ buf \circ in)$

Producer-Consumer Synchronization

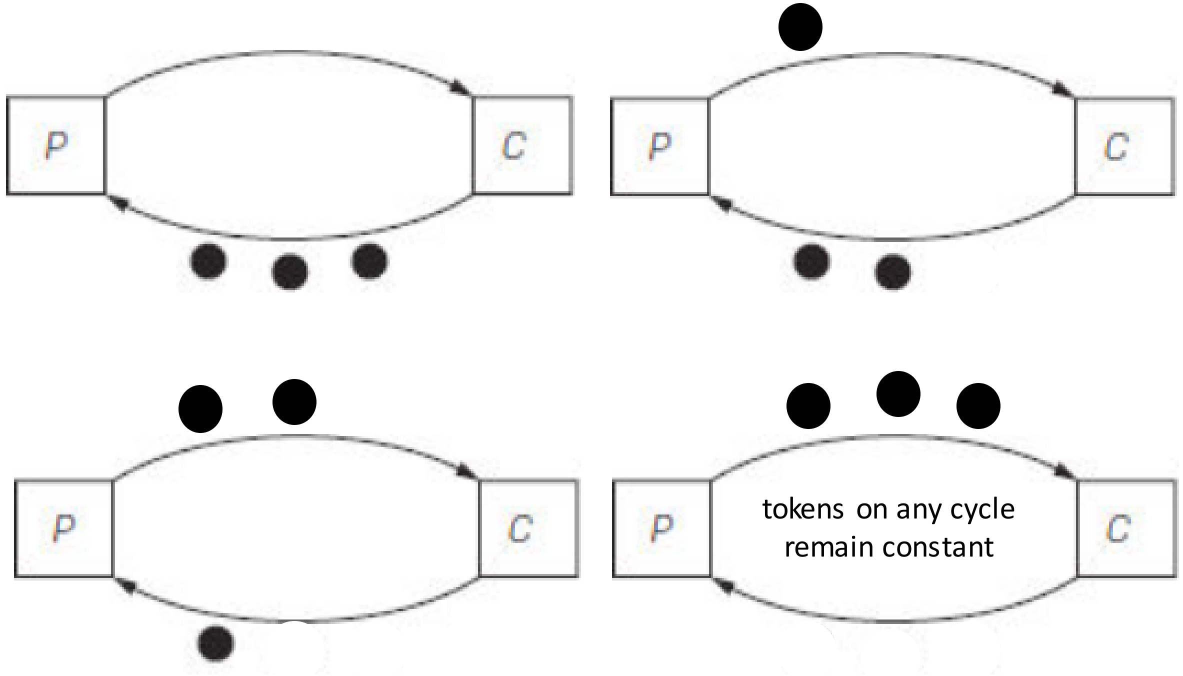


An event history for the FIFO queue with N = 3.



FIFO queue as N process system.

Producer-Consumer: deterministic
Mutual exclusion: non-deterministic (race w/ arbiter)
(arbiter decides which of 2 events happened first)



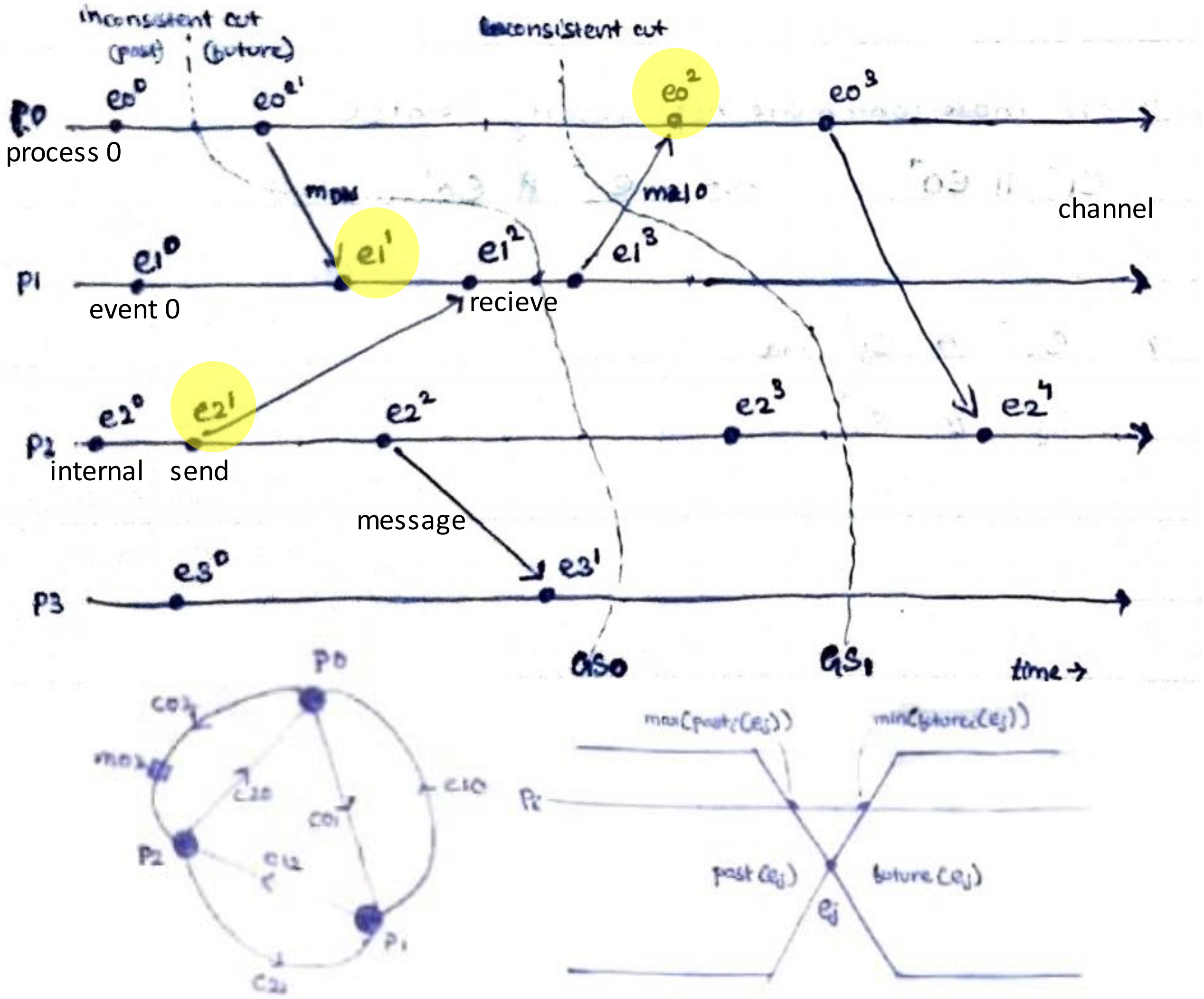
Generalizing to marked graph synchronization.

Marked graph: Direct graph with edge-marking w/ finite set of tokens.

Node fire: remove 1 token from each input + add 1 to each output
Firing sequence: fire until no node can fire

Event History

Pictures of event histories were first used to describe distributed systems.



- No global shared clock.
- Communication delay unpredictable.

- **p**: async process (i)
- **c**: channel, unidirectional (ij)
- **m**: message (k)

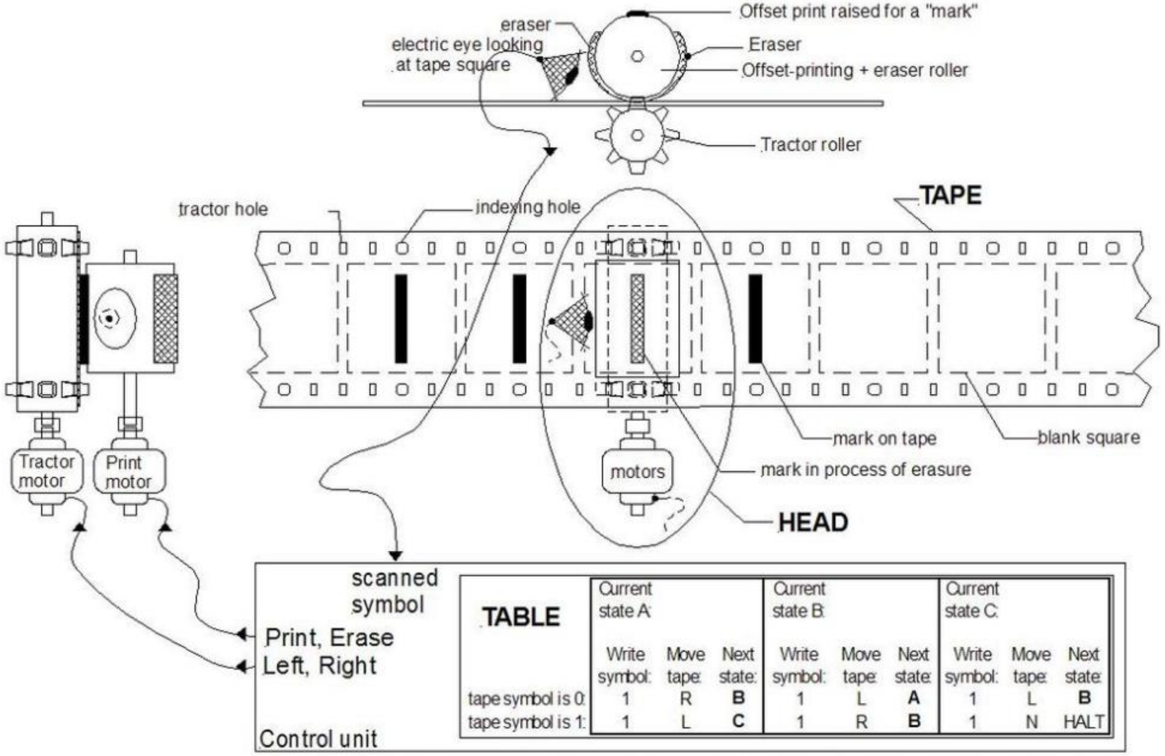
- **Causality**: message must be sent before it was received.
- Within a process, all events causally follow.
- Send & receive of a message causally follow.
- **Concurrent events**: those not causally related.

- **Process state**: events in process
- **Channel state**: message sent but not yet received
- **Global state**: process states U channel states
- - consistent: a message not sent is neither in channel or recieved
- - transitless: all channels are empty (strong: consistent + transitless)

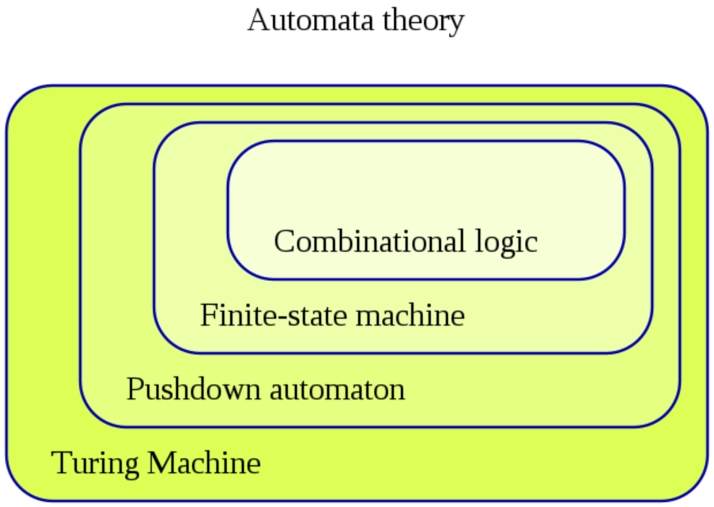
- **past(e)**: all events that causally happen before e
- **max_past(e)**: events that happen just before e (each process)
- **min_future(e)**: events that happen just after e

Standard Model

Execution is represented as a sequence of atomic transitions states.



A fanciful mechanical Turing machine's TAPE and HEAD. The TABLE instructions might be on another "read only" tape, or perhaps on punch-cards. Usually a "finite state machine" is the model for the TABLE.



- Event history describes all sequences of states that represent executions in the standard model.
- The executions are not inherently different, but artifacts of the standard model. It requires concurrent executions of two operations to be modeled as ocurring in an order.
- The problem of implementing a distributed system can often be viewed as that of maintaining a global invariant even though different processes may have incompatible views of what the current state is.
- The standard model provides the most practical way to reason about invariance.

Mutual Exclusion

Synchronizing N processes, each with a section of code called its critical section.

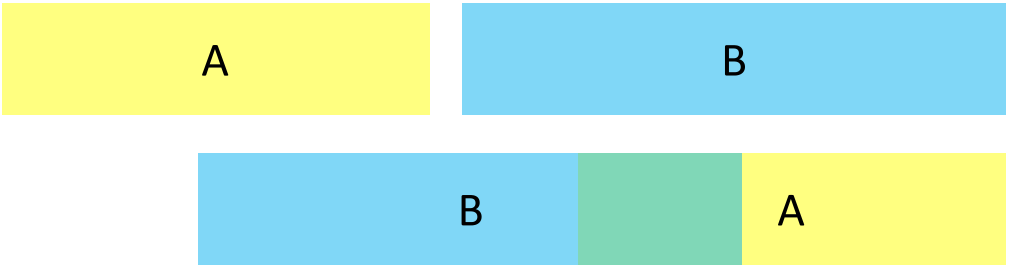
- **Safety**: no 2 ciritical section executed concurrently.
- **Liveness**: some process eventually executes its CS.
- Assumption: "process fairness" every process eventually takes a step.

- **BAKERY ALGORITHM**
- First to implement mut-ex without any lower level mut-ex.
- Customers take tickets, and lowest number ticket is served first.
- Each process computes its ticket number from others' tickets.
- Proof does not require read/write of entire number be atomic.

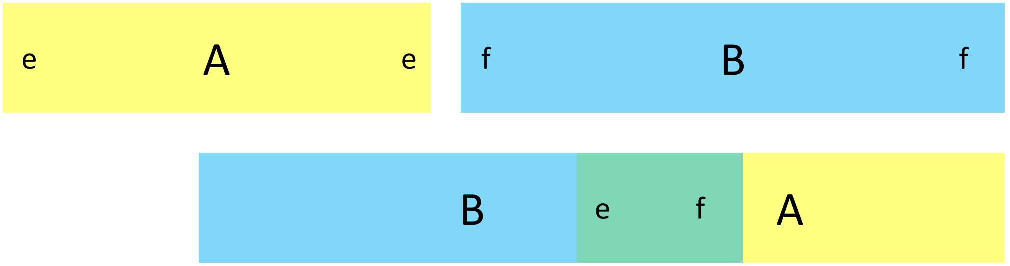
```
// declaration and initial values of global variables
Entering: array [1..NUM_THREADS] of bool = {false};
Number: array [1..NUM_THREADS] of integer = {0};

lock(integer i) {
    Entering[i] = true;
    Number[i] = 1 + max(Number[1], ..., Number[NUM_THREADS]);
    Entering[i] = false;
    for (integer j = 1; j <= NUM_THREADS; j++) {
        // Wait until thread j receives its number:
        while (Entering[j]) { /* nothing */ }
        // Wait until all threads with smaller numbers or with the same
        // number, but with higher priority, finish their work:
        while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i))) { /* nothing */ }
    }
}

unlock(integer i) {
    Number[i] = 0;
}

Thread(integer i) {
    while (true) {
        lock(i);
        // The critical section goes here...
        unlock(i);
        // non-critical section...
    }
}
```

Two-arrow Model

$A \longrightarrow B$ is true iff (if and only if) memory barrier
A ends before B begins.
$A \dashrightarrow B$ is true iff A begins before interprocess communication
B ends.



$A \longrightarrow B$ iff $\forall\, e \in A, f \in B : e \prec f,$
$A \dashrightarrow B$ iff $\exists\, e \in A, f \in B : e \prec f.$

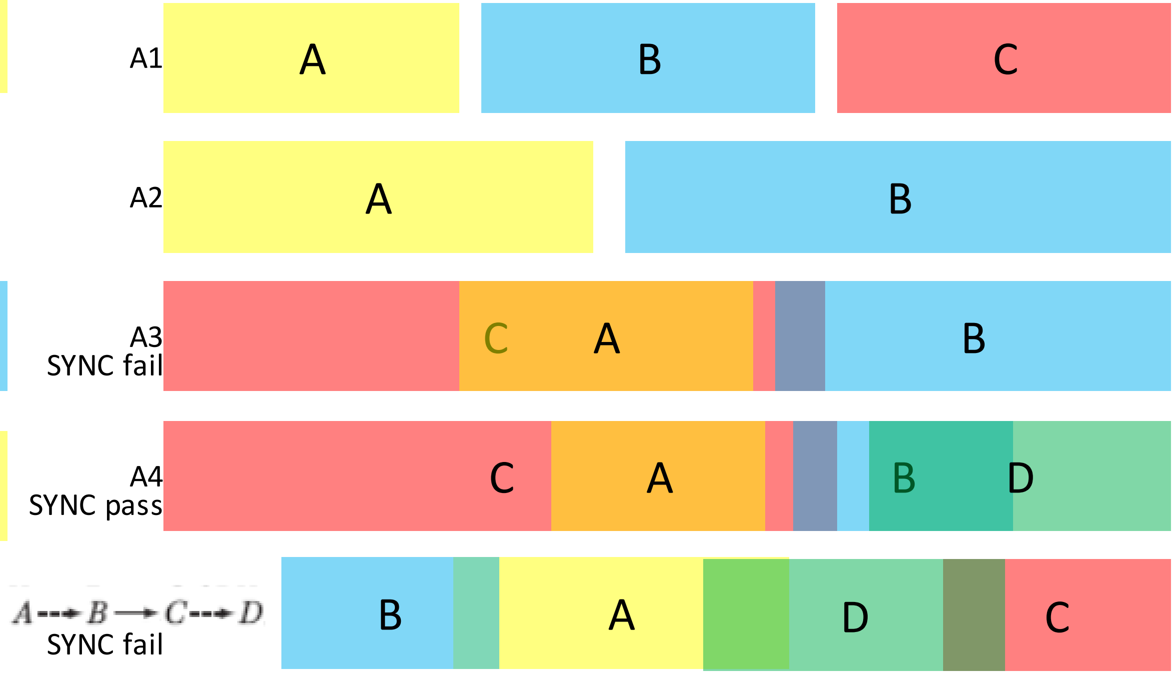


Two-arrow model for event history

A1. (a) $A \longrightarrow B \longrightarrow C$ implies $A \longrightarrow C$
($\longrightarrow$ transitively closed)
(b) $A \not\longrightarrow A$. ($\longrightarrow$ irreflexive)
A2. $A \longrightarrow B$ implies $A \dashrightarrow B$ and
$B \not\dashrightarrow A$.
A3. $A \longrightarrow B \dashrightarrow C$ or $A \dashrightarrow B \longrightarrow C$
implies $A \dashrightarrow C$.
A4. $A \longrightarrow B \dashrightarrow C \longrightarrow D$ implies $A \longrightarrow D$.

For A4:
A: data = 10
B: sent = T

C: if (sent)
D: read data



Compiler Instruction Reordering

```c
int A, B;

void foo()
{
    A = B + 1;
    B = 0;
}
```

RULE: *Thou shalt not modify the behavior of a single-threaded program.*

```c
int Value;
int IsPublished = 0;

void sendValue(int x)
{
    Value = x;
    IsPublished = 1;
}
```

recvValue() can read garbage value.

```
$ gcc -S -masm=intel foo.c
$ cat foo.s
        ...
        mov     eax, DWORD PTR _B
        add     eax, 1
        mov     DWORD PTR _A, eax
        mov     DWORD PTR _B, 0
        ...
```

```
$ gcc -O2 -S -masm=intel foo.c
$ cat foo.s
        ...
        mov     eax, DWORD PTR B
        mov     DWORD PTR B, 0
        add     eax, 1
        mov     DWORD PTR A, eax
        ...
```

Problematic with Lock-free programming.
Use C++11 atomics.

Explicit Compiler Barrier

```c
int A, B;

void foo()
{
    A = B + 1;
    asm volatile("" ::: "memory");
    B = 0;
}
```

```
$ gcc -O2 -S -masm=intel foo.c
$ cat foo.s
        ...
        mov     eax, DWORD PTR _B
        add     eax, 1
        mov     DWORD PTR _A, eax
        mov     DWORD PTR _B, 0
        ...
```

```c
#define COMPILER_BARRIER() asm volatile("" ::: "memory")

int Value;
int IsPublished = 0;

void sendValue(int x)
{
    Value = x;
    COMPILER_BARRIER();          // prevent reordering of stores
    IsPublished = 1;
}

int tryRecvValue()
{
    if (IsPublished)
    {
        COMPILER_BARRIER();      // prevent reordering of loads
        return Value;
    }
    return -1;  // or some other value to mean not yet received
}
```

recvValue() now works.

Implied Compiler Barrier

```
#define RELEASE_FENCE() asm volatile("lwsync" ::: "memory")

void sendValue(int x)
{
    Value = x;
    RELEASE_FENCE();
    IsPublished = 1;
}
```

CPU fence instructions also prevent compiler reordering.

```
int Value;
std::atomic<int> IsPublished(0);

void sendValue(int x)
{
    Value = x;
    // <-- reordering is prevented here!
    IsPublished.store(1, std::memory_order_release);
}
```

C++11 atomic instructions prevent compiler reordering.

Implied Compiler Barrier

```
void doSomeStuff(Foo* foo)
{
    foo->bar = 5;
    sendValue(123);       // prevents reordering of neighboring assignments
    foo->bar2 = foo->bar;
}
```

Except inline, pure functions a call to an external function is even stronger than a compiler barrier, since the compiler has no idea what the function's side effects will be.

```
$ gcc -O2 -S -masm=intel dosomestuff.c
$ cat dosomestuff.s
        ...
        mov     ebx, DWORD PTR [esp+32]
        mov     DWORD PTR [ebx], 5            // Store 5 to foo->bar
        mov     DWORD PTR [esp], 123
        call     sendValue                     // Call sendValue
        mov     eax, DWORD PTR [ebx]          // Load fresh value from foo->bar
        mov     DWORD PTR [ebx+4], eax
        ...
```

Register replacement

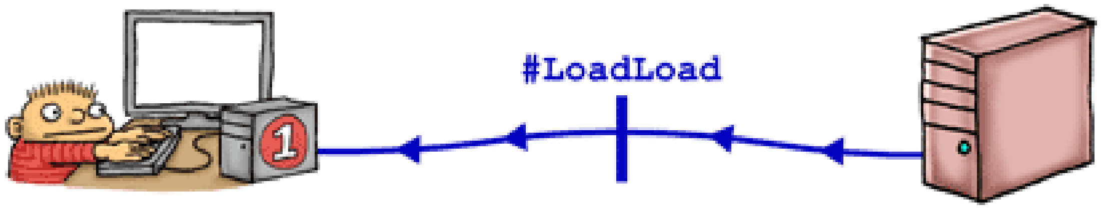
```
int A, B;

void foo()
{
    if (A)
        B++;
}
```

```
void foo()
{
    register int r = B;    // Promote B to a register before checking A.
    if (A)
        r++;
    B = r;          // Surprise! A new memory store where there previously was none.
}
```

What if multiple increments happen on regitser before return? **Overwrite**.

C++11 doesn't avoids register replacement on shared memory locations.

Types of Memory Barrier




```
if (IsPublished)                   // Load and check shared flag
{
    LOADLOAD_FENCE();              // Prevent reordering of loads
    return Value;                  // Load published value
}
```


```
Value = x;                         // Publish some data
STORESTORE_FENCE();
IsPublished = 1;                   // Set shared flag to indicate availa
```

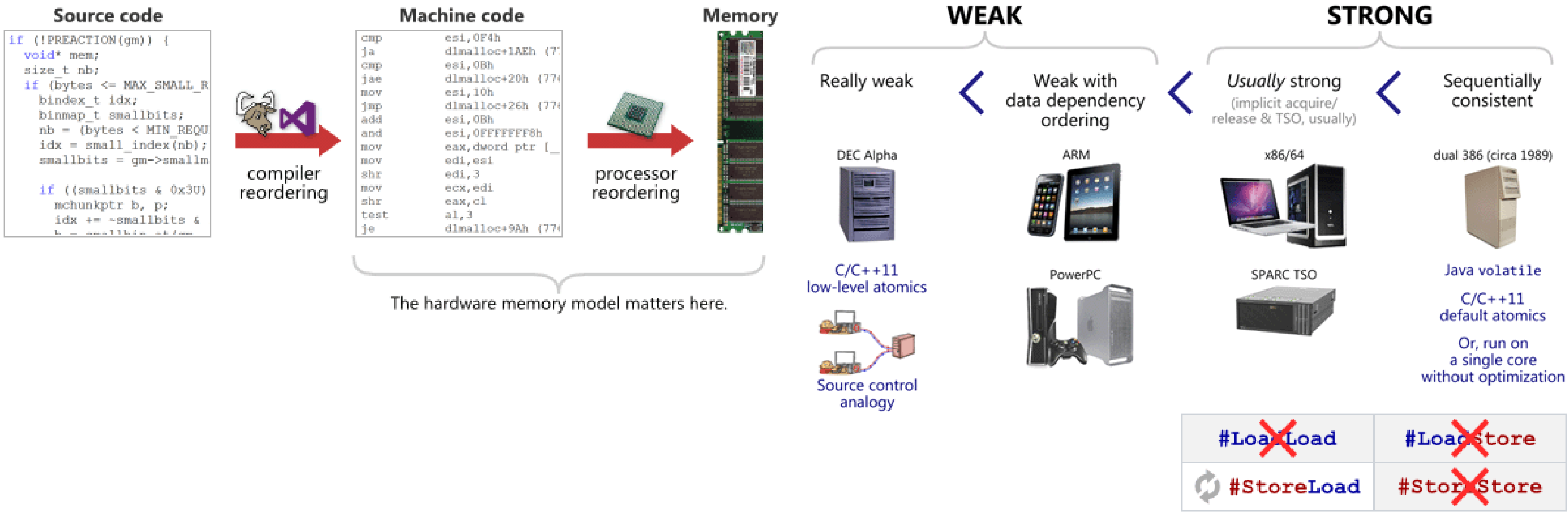
#LoadLoad	#LoadStore
#StoreLoad	#StoreStore

lwsync:
#LoadLoad, #StoreStore, #LoadStore

#StoreLoad is unique: a = b = 0
#StoreLoad acts as a full memory fence

DataDependency is a barrier too.

Weak vs. Strong Memory models



Temporal Logic

A logic for specifying properties over time.

$\mathbf{GP}(\text{Prior invents TL})$
- It will always be the case that Prior invented TL.

$\text{dark} \wedge \mathbf{P}(\text{light}) \wedge \mathbf{F}(\text{light})$
- It is dark, it was light, and it will be light again.

$\text{try } \mathbf{U} \text{ succeed}$
- I will keep trying until I succeed.

$(\text{Joe unhappy} \wedge (\text{Joe drinks } \mathbf{U} \neg(\text{Joe conscious}))) \mathbf{\ S\ } (\text{Mia leaves})$
- Ever since Mia left home, Joe has been unhappy and has been drinking until losing consciousness.

$\mathbf{G}(\text{Sent} \rightarrow (\neg\text{MarkedSent } \mathbf{U} \text{ AckReturned}))$
- Every time when a message is sent, an acknowledgment of receipt will eventually be returned, and the message will not be marked ‘sent’ before an acknowledgment of receipt is returned.

Propositional Logic

A proposition is a declarative sentence that is either true or false, but not both.

- If it is raining, then the home team wins. $\mathbf{p \rightarrow q}$
- If the home team does not win, then it is not raining. $\mathbf{\neg p \leftarrow \neg q}$

- If the home team wins, then it is raining. $\mathbf{p \leftarrow q}$
- If it is not raining, then the home team does not win. $\mathbf{\neg p \rightarrow \neg q}$

- **p** : You have the flu.
- **q** : You miss the final examination.
- **r** : You pass the course.

- $p \rightarrow q$, $\neg q \leftrightarrow r$, $q \rightarrow \neg r$, $p \vee q \vee r$
- $(p \rightarrow \neg r) \vee (q \rightarrow \neg r)$, $(p \wedge q) \vee (\neg q \wedge r)$

Amir Pnueli in 1977

Goranko, Valentin and Antje Rumberg, "Temporal Logic", The Stanford Encyclopedia of Philosophy (Summer 2020 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/sum2020/entries/logic-temporal/>.

Discrete Mathematics and Its Applications: With Combinatorics and Graph Theory
Rosen, K.H.; Krithivasan, K.; (2012)

π-calculus

Formal foundation of concurrent programming.

$$
\begin{aligned}
P, Q ::= \; & x(y).\,P && \text{Receive on channel } x \text{, bind the result to } y \text{, then run } P \\
\mid \; & \overline{x}\langle y\rangle.\,P && \text{Send the value } y \text{ over channel } x \text{, then run } P \\
\mid \; & P|Q && \text{Run } P \text{ and } Q \text{ simultaneously} \\
\mid \; & (\nu x)P && \text{Create a new channel } x \text{ and run } P \\
\mid \; & !P && \text{Repeatedly spawn copies of } P \\
\mid \; & 0 && \text{Terminate the process}
\end{aligned}
$$

$$
\begin{aligned}
(\nu x)\;(\; & \overline{x}\langle z\rangle.\;0 \\
\mid\; & x(y).\;\overline{y}\langle x\rangle.\;x(y).\;0\;) \\
\mid\; & z(v).\;\overline{v}\langle v\rangle.0
\end{aligned}
$$

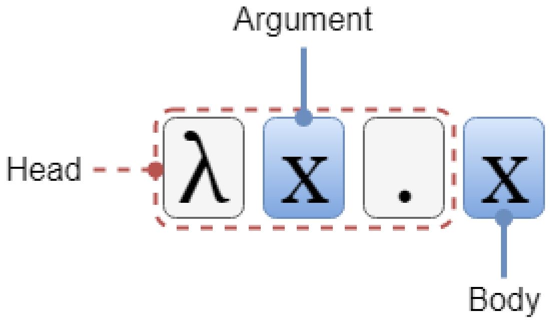
$$
\begin{aligned}
(\nu x)\;(\; & 0 \\
\mid\; & \overline{z}\langle x\rangle.\;x(y).\;0\;) \\
\mid\; & z(v).\;\overline{v}\langle v\rangle.\;0
\end{aligned}
$$

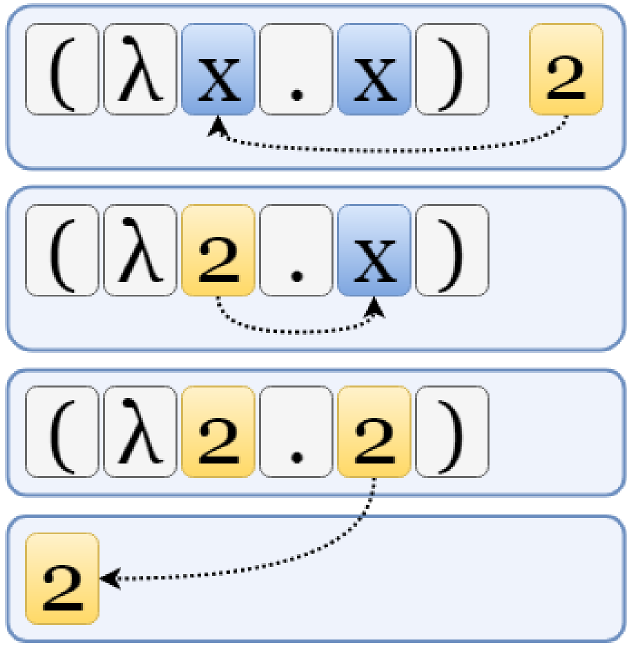
$$
\begin{aligned}
(\nu x)\;(\; & 0 \\
\mid\; & x(y).\;0 \\
\mid\; & \overline{x}\langle x\rangle.\;0\;)
\end{aligned}
$$

$$
\begin{aligned}
(\nu x)\;(\; & 0 \\
\mid\; & 0 \\
\mid\; & 0\;)
\end{aligned}
$$

Lambda Calculus

Express computation as function abstraction and application w/ binding & substitution.






```
(λx.x) 10
10
---
(λx.x * 2) 2
4
---
(λx.1 + x) 2
3
```

https://www.simonholywell.com/post/the-lambda-calculus/