Order Is A Lie

LSE
Security System
Laboratory of Epita

Are you sure you know how your code runs ?

Order in code is not respected by

- Compilers
- Processors (*out-of-order* execution)
- SMP Cache Management

Understanding execution order in a multithreaded context is out of reach of a human mind.

Compilers and Order ?

Security System
Laboratory of Epita

Order and Side Effects

```c
int next() {
  static int x = 0; return x++;
}

void g() {
  int x = 0, y, tab[32];
  // can be equivalent to:
  // tab[0] = 1
  // tab[1] = 0;
  // ...
  tab[x++] = x++;
  // x = 2 - 1 or 1 - 1 ?
  y = x + --x;
  // x = 0 - 1 or 1 - 0 ?
  x = next() - next();
}
```

LSE Security System
Laboratory of Epita

Out Of Order ?

OoO

OoO

Do you know what a pipeline is ?
Out-of-order is the next step.

OoO

1990: first microprocessor

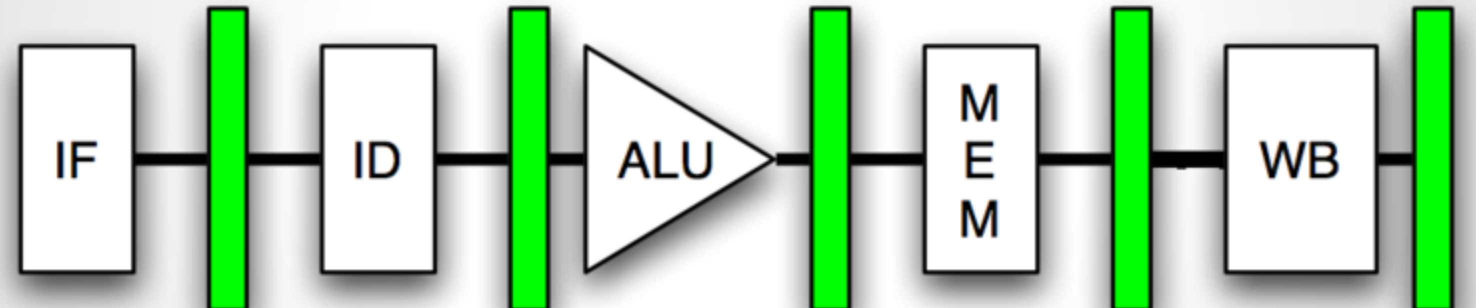
IBM Power 1

Not a new a idea

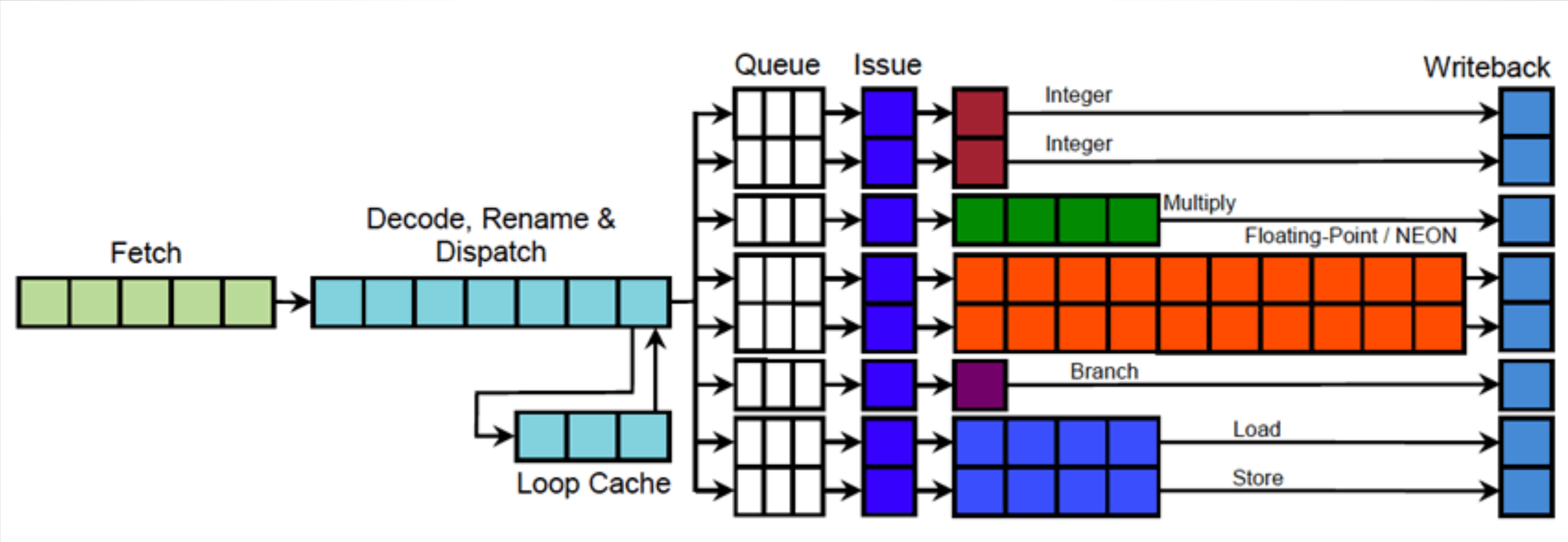
1964/1966: first *out-of-order* machine

CDC6600 & IBM 360/91

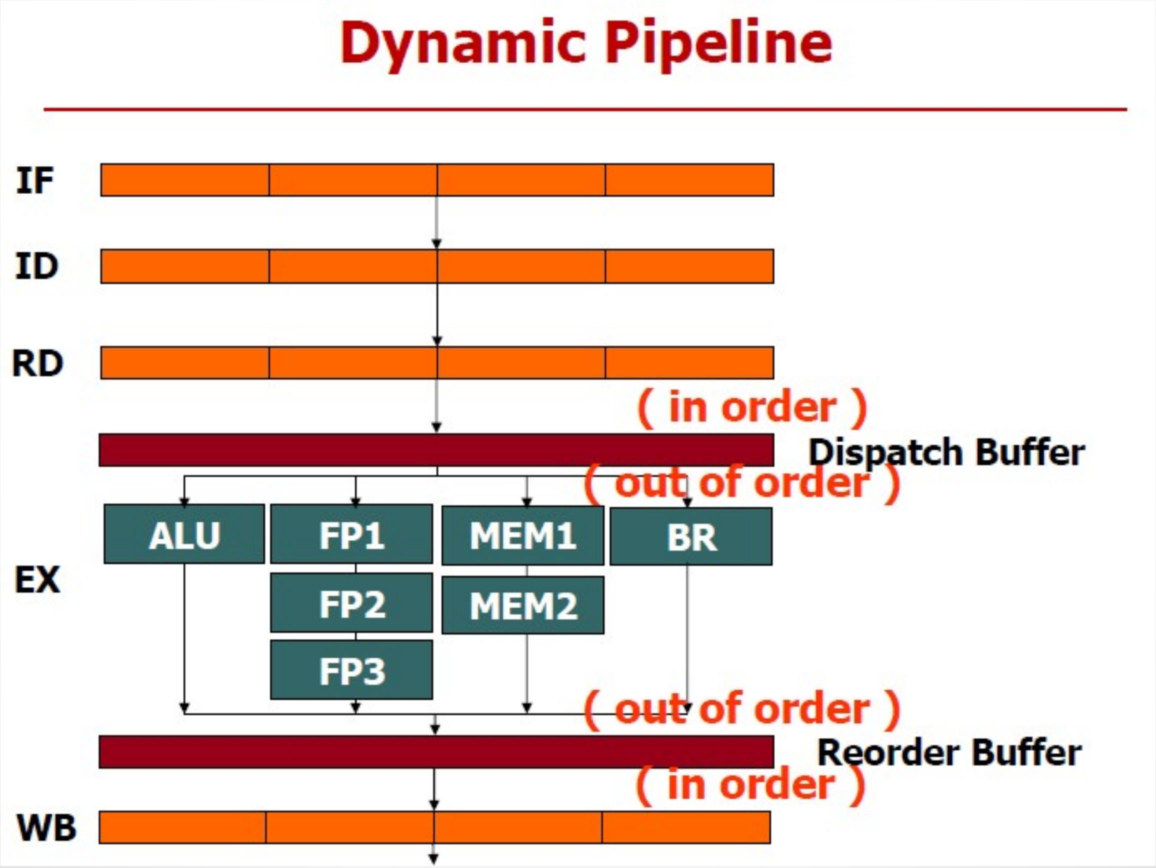
Pipeline ...



Pipeline … with OoO



OoO


Dynamic Pipeline
IF
ID
RD
(in order)
Dispatch Buffer
(out of order)
ALU
FP1
MEM1
BR
EX
FP2
MEM2
FP3
(out of order)
Reorder Buffer
(in order)
WB

OoO

```c
int f(int *a) {
  int x = 1, y;
  y = *a;
  x += 41; // Don't need previous statement
  *a = x;  // Require 2 previous statements
  return y;
}
```

And The Cache ?

Cache

multiple processors + slow memory

=

a lot of hardware caches !

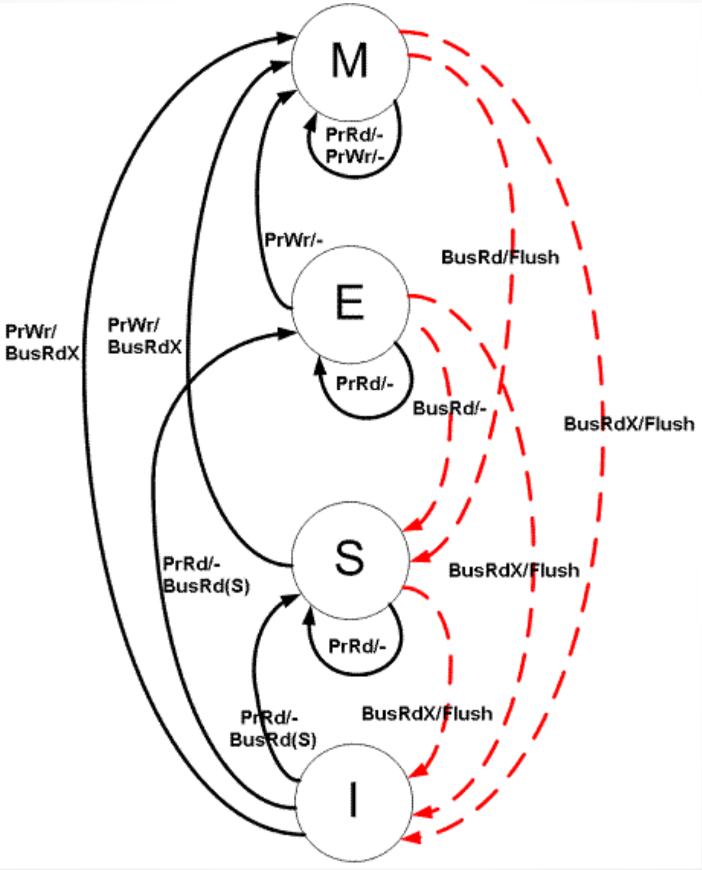
Cache Coherency

M	modified	line is owned by 1 core
E	exclusive	
S	shared	line is shared
I	invalid	line is E or M elsewhere

Cache Coherency

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

Cache Coherency



LSE Security System Laboratory of Epita

Cache Coherency

- Line invalidation is expensive
- To improve perf, procs use:
  - Store Buffer
  - Invalidate Queue
- We need barrier !

So what can we do ?

Theoretical View

Determinism can be defined through the observation of memory states history.

Theoretical View

A program is **deterministic** if we don't observe different states history through (all possible) executions.

Linearizability

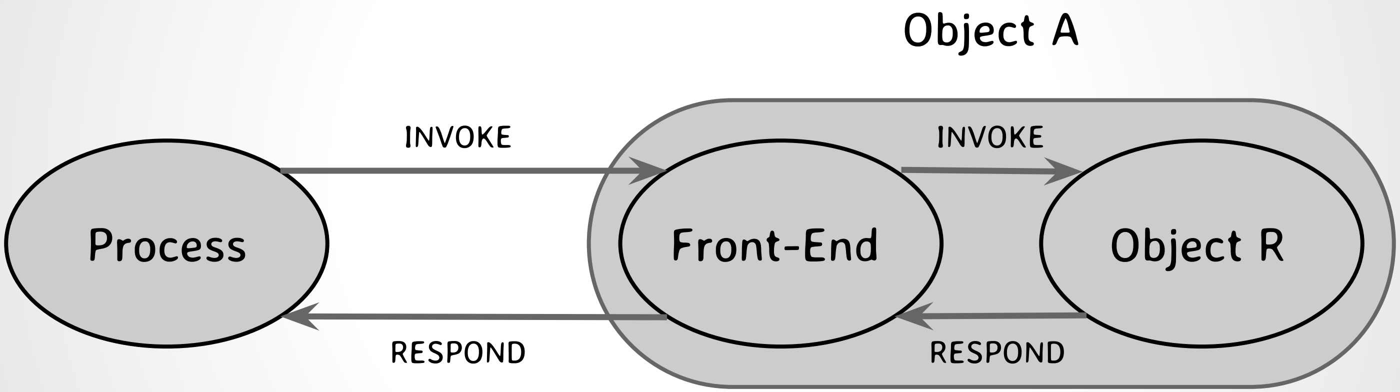
An history is atomic if:

- its invocations and responses can be reordered to yield a sequential history.
- that sequential history is correct according to the sequential definition of the object.
- if a response preceded an invocation in the original history, it must still precede it in the sequent reordering

Dealing With Memory

I/O Automaton can be used to describe properties and behavior independently of concrete hardware implementation.

Dealing With Memory



Main Results

- Wait-free operations are possible
- The only meaningful primitives are:
  - Compare-and-Swap (CAS)
  - Load-Link/Store-Conditional (ll/sc)
- Order is not required for determinism !

Compare And Swap

```
bool CAS(int *loc, int cmp, int newval) {
  if (*loc == cmp) {
    *loc= newval;
    return true;
  }
  return false;
}
```

LSE Security System Laboratory of Epita

ll/sc

- **Load** from memory and **link** to the cell
- **Store** in the cell **if** no write was made

- More powerful than CAS
- More RISC oriented
- Many implementations are *weak*

ll/sc *v.s.* CAS

- Hardware ll/sc is often broken
- Most broken ll/sc can simulate CAS
- **Most algorithms are described using CAS**

LSE Security System
Laboratory of Epita

Memory Barriers

- **Release:** force all write operations to be finished before the barrier
- **Acquire:** prevent all read operations to begin before the barrier
- **Full:** acquire and release at the same time

Barriers will also flush Store Buffers and Invalidate Queues.

Memory Barriers

```c
void worker0(char *msg, char *shr, int *ok) {
  for (char *cur = msg; *cur; ++cur, ++shr)
    *shr = *cur;
  // need a release barrier
  *ok = 1;
}
void worker1(char *shr, int *ok) {
  if (*ok) // need an acquire barrier
    printf("%s\n", shr);
}
```

Non Blocking

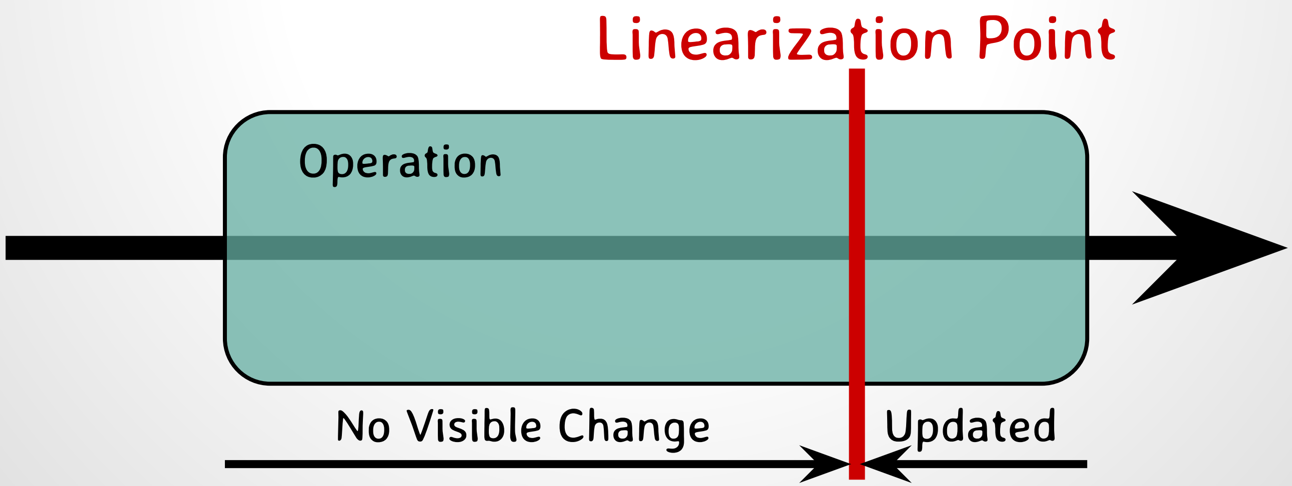
Non Blocking ?

- It's all about progression
- We don't want locks
- We want minimal system interactions
- We want to scale upon heavy contention

LSE Security System
Laboratory of Epita

Linearization Point

- **Usual mistake:** *atomic means one instruction*
- For observers, an operation is atomic if there's a point marking the change



Lock-free

As long as one thread is active, the whole system makes **progress**.

A lock-free algorithm should leave shared data in *correct states* between linearization points.

Lock-free

- Rely only on **CAS**
- Usual schema is:
  a. Prepare
  b. Acquire entry data points
  c. Prepare update
  d. Update (CAS) if entry are valid or go to *b*
- *d* is the linearization point

Lock-free

Existing Algorithms (mostly in Java) for:

- Stack
- Queue
- Linked list
- Skip-list
- ...

Lock-free Queue

Lock-free Queue is a classic (PODC96)
Implemented for years in Java
Not in C++ due to lack of memory-model.

1. Acquire tail (push) or head (pop)
2. Prepare for update
3. When queue is in a temporary state (incomplete pop) finished the job and retry
4. In all cases, if acquired pointers have changed, retry, otherwise do the update.

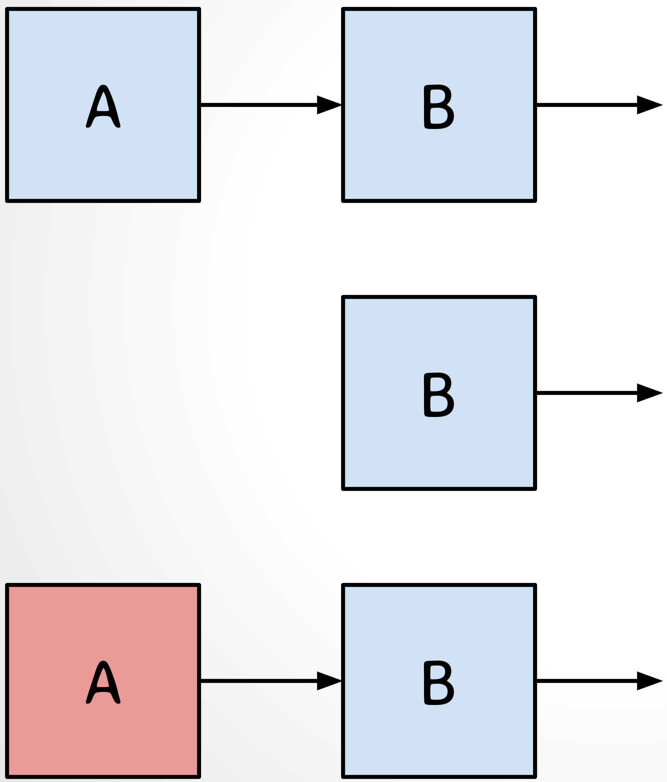
Lock-free and Memory

In most lock-free algorithms, threads can hold pointers that can be deleted by other threads.

Lock-free and Memory

- First attempt: use a recycler
  - avoid early free
  - do not protect from ABA issues
- Use a garbage-collector ?
  - solves early free and ABA issues
  - are GCs wait/lock free ? ...

ABA problem



Read pointer A

Entry is now B

Read pointer A

Lock-free and Memory

Two main solutions:

- Double-word based solutions
  - using pair pointer/counter
  - Only x86-64 provides 128b CAS
- Hazard Pointers
  - Simple
  - wait-free
  - not hardware dependant

Lock-free Performances

- **Academics:** better perf than lock-based algos
- **Java:** implementation agrees
- C++ ? None officials, mine has strange results.
- Pure bench speed-up are not clear
- Hybrid algorithms (TBB) can do better with limited number of threads.

Wait-free

In a given set of processes, each process can perform its action in a finite (bounded) number of steps.

Wait-free

- Far more difficult than lock-free
- Implementation are far more expensive
- Can't use failure/retry loop
- Most implementation use helping system:
  1. Make a forward step for another thread
  2. Start its own action step by step
- All pending operations have progression !

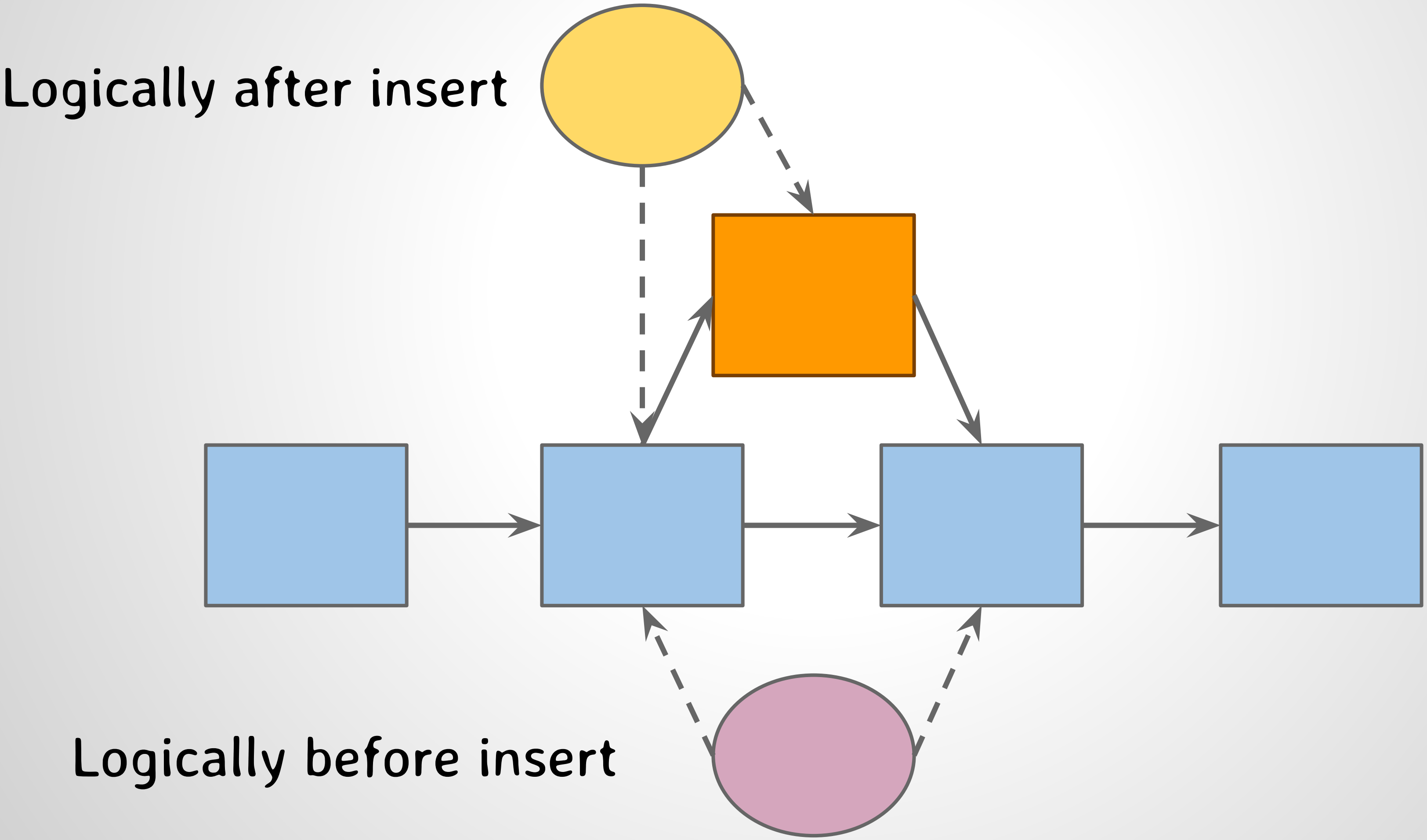
Wait-free

Recently (2011) a new approach appears:

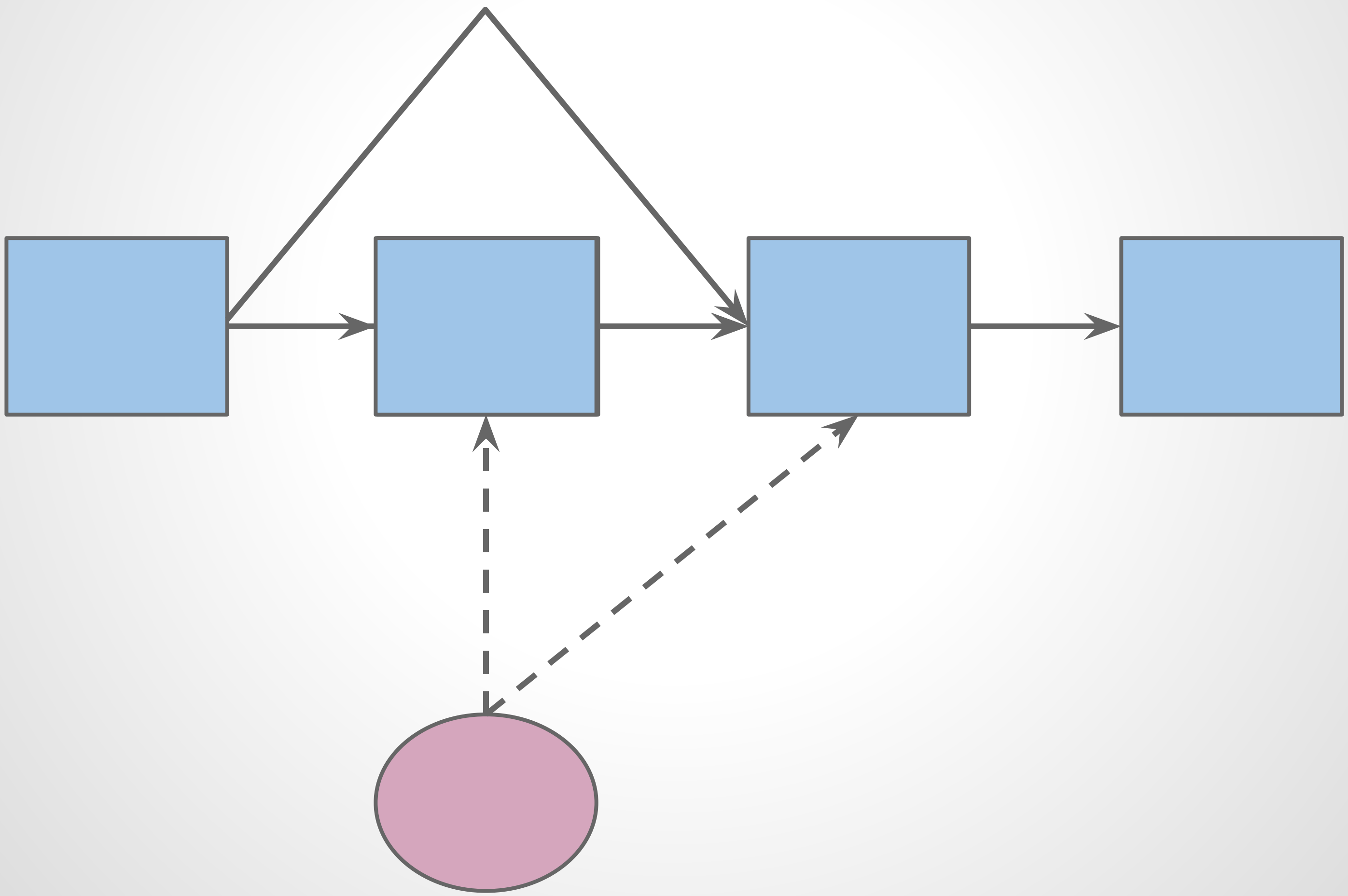
- Mix lock-free algo with helping mechanism:
  1. Try to help every *N* calls
  2. Bounded failure/retry loop (lockfree)
  3. Fail ? Move to helping mechanism
- Provide similar perf as lock-free algos.

Security System
Laboratory of Epita

RCU by Example



RCU by Example



Security System
Laboratory of Epita

Conclusion

?