Advanced Multiprocessor Programming

Jesper Larsson Träff
traff@par.tuwien.ac.at
Research Group Parallel Computing, 191-1
Faculty of Informatics, Institute of Computer Engineering
Vienna University of Technology (TU Wien)

"With the notable exception of simple atomic counters, lock-free programming is for specialists."

B. Stroustrup, The C++ Programming Language, 4th Ed., 2013

"The Art of multiprocessor programming": Two concerns

From "Intro. Parallel Computing"

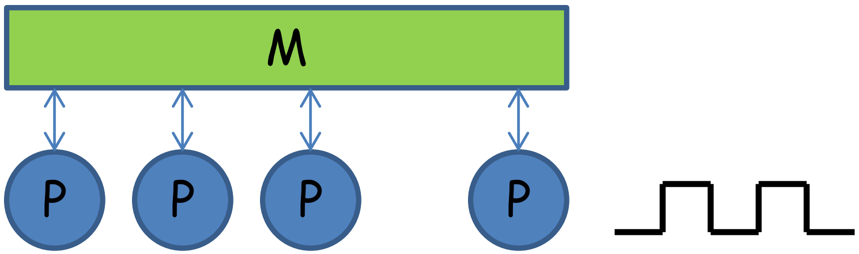
Parallel computing:
The discipline of efficiently utilizing dedicated parallel resources (processors, memories, …) to solve a single, given computational problem.

Performance, efficiency

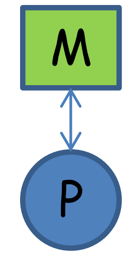
Concurrent computing:
The discipline of managing and reasoning about interacting processes that may (or may not) take place simultaneously

Coordination, correctness

Abstraction 1: PRAM (Parallel Random Access Machine)



PRAM natural generalization of the RAM



1. Synchronous processors in lock step
2. Unit time memory access (Conflicts: EREW, CREW, CRCW)

Good for studying parallel algorithms and especially establishing lower bounds (we do: lecture in WS)

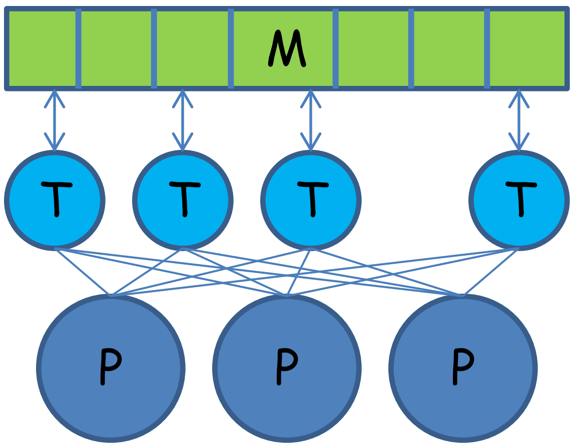
But (some say): Not realistic. This lecture: Different model, different concerns, close(r) to current multi-core, shared-memory processors

PRAM algorithms and complexity

- Algorithms: With p PRAM processors, solve given problem (sorting, connected components, convex hulls, linear equations, …) p times faster than best possible/best known sequential algorithm
- Data structures: With p processors, make individual or bulk operations p times faster
- Complexity: Establish lower bounds on possible improvements under various assumptions about the memory system; study problems that may be hard to parallelize (no fast algorithms known)

PRAM model: Interesting, relevant, non-trivial results in all 3 areas

Abstraction 2: (Symmetric) Shared-memory computer



Issues/problems:

- Threads need to synchronize and coordinate
- How can we reason about progress and complexity (speed-up)?
- How can we ensure/reason about correctness?
- Scheduling tasks to threads, treads to processors (cores)

1. Threads executed by processors, |T|≥|P|
2. Threads are not synchronized
3. Memory access not unit time
4. (Memory accesses not ordered, memory updates not necessarily in program order)

Shared-memory algorithms and complexity

- Algorithms: Algorithms that are correct for p asynchronous threads; faster than sequential algorithm under optimistic assumptions on thread progress and absence of resource conflicts.
- Data structures: p threads can work independently and each perform individual operations on data structure, complexity not too far from best known sequential implementation, guarantees on progress (for instance, deadlock freedom)
- Complexity: Lower bounds on resource requirements to achieve desired properties

Fact

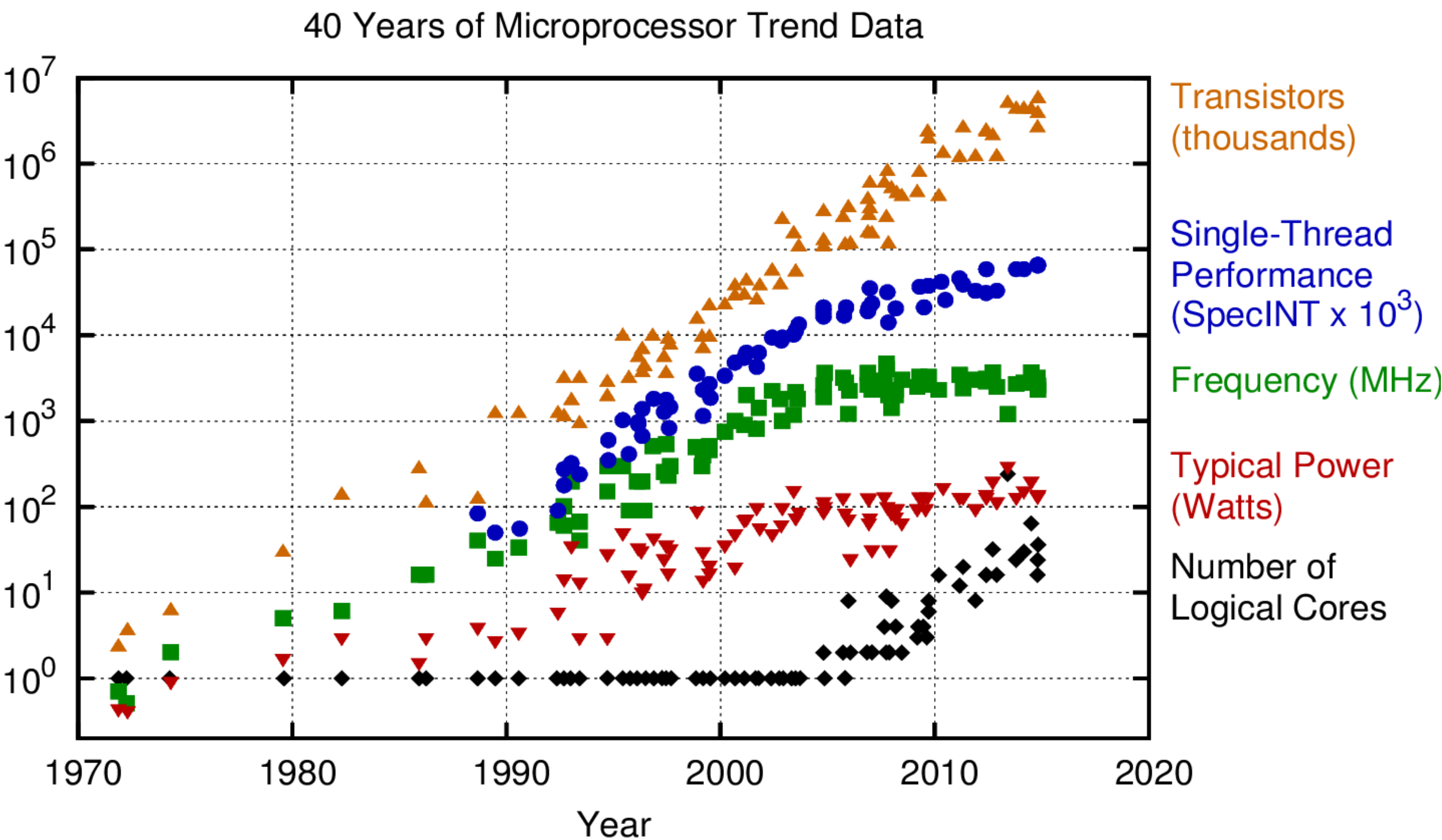
All current, modern (general-purpose) processors are parallel processors (multi-core cpu, GPU, …), mostly asynchronous

For CPUs:

- Clock-frequency has not been increasing (much) since ca. 2005
- Power-consumption: Same
- Single-core performance no longer following Moore's "law" (performance version)

- Number of transistors still increasing exponentially (Moore's observed and extrapolated "law")
- Number of cores per chip increasing

Kunle Olukotun, 2010, Karl Rupp (TU Wien), 2015



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

The first concern: Speeding up time to solution of given problem

Definitions:

Tseq(n): Time (or number of operations) needed to solve given problem sequentially by best possible or best known algorithm and implementation on given machine

Tp(n): Time (or number of operations), by slowest processor, needed to solve given problem of size n using p processors (given algorithm on given machine)

Absolute speed-up: SU(p) = Tseq(n)/Tp(n)

Can time to solve problem be improved?

Relative speed-up: T1(n)/Tp(n)

Does parallel algorithm scale?

Recall:

SU(p) =Θ(p) is best possible (while relative to best known sequential algorithm): linear speed-up

for $p \leq \text{maxP}(n)$

SU(p) = p: perfect speed-up

The larger the p for which linear/perfect speed-up is achieved, the better

Constants normalized out, assume same constant in both O

Example:
$T_p(n) = O(n/p+\log n)$ has linear speed-up for p in $O(n/\log n)$, assuming $T_{seq}(n) = O(n)$, namely (normalizing constants)

$$SU(p) = n/(n/p+\log n) = p/(1+(p \log n)/n) > p/(1+\varepsilon) \text{ for } p\leq\varepsilon n/\log n$$

Practical remarks:

Tseq(n), Tp(n) measured times for some (good) implementation of the algorithms, for some input (Worst-case? Average case? Average over many inputs?)

Measuring time is not trivial. Tp(n) usually defined as time of last processor to finish, assuming all p processors start at the same time (temporal synchronization problem)

T1(n) ≥ Tseq(n), since Tseq(n) is time of best known/possible algorithm

Reporting only relative speed-up with baseline T1(n) compared to Tseq(n) can be grossly misleading

Analogies

Tseq(n) sometimes called the "work" required to solve the problem of size n (in number of "operations")

A good parallel algorithm will effectively divide this work over the available p processors, such that the total parallel work is O(Tseq(n)). In this case

Tp(n) = O(Tseq(n)/p) with linear speed-up

Challenges:

- Load balancing (the work that has to be done must be evenly distributed, no processors idle for too long)
- Little extra work (redundancy, parallelization overhead)
- Small overhead (synchronization, shared data structures)

Definition (Throughput):

Number of operations that can be carried out in some given amount of time t

ThroughputSU(t) = Throughputp(t)/Throughput1(t)

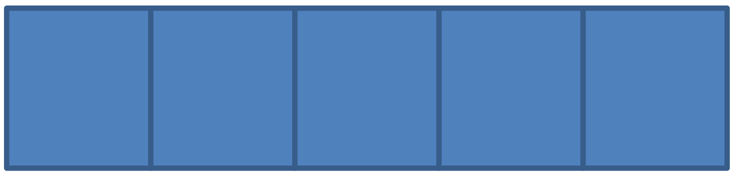
Definition (latency):

Time taken to carry out some given number of operations n

Relevant measures when benchmarking data structures. Important to decide what the operations are and in what distribution

Example: Performance

„painting 5 room flat…" (from Herlihy-Shavit book, apologies)



Amount of work, Tseq(n), here 5 rooms

If the amount of work can be trivially divided (5 rooms over 5 people) into p parts, Tp(n) = Tseq(n)/p, and

SU(p) = Tseq(n)/Tp(n) = p

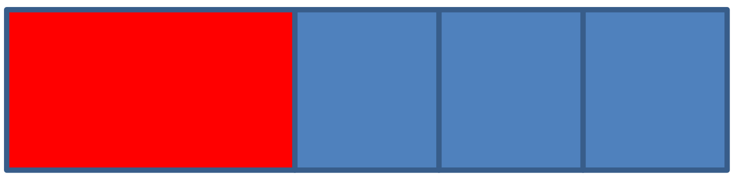
Trivially (embarrassingly) parallel:

- No coordination overhead
- No extra work done

Example: Performance

„painting 5 room flat…“ (from Herlihy-Shavit book, apologies)

5 people, s=2/5, SU = 1/(2/5+3/5/5) = 25/13 ≈ 1.9 < 2.5 = 5/2



Amount of work, Tseq(n), here 5 rooms

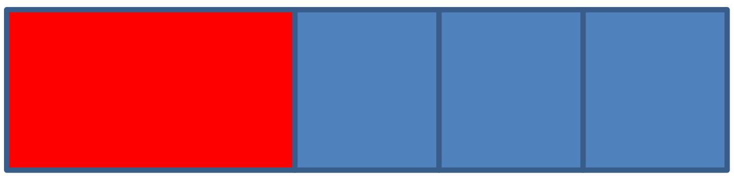
But… some part of the work may not be divisible (one large room), say, some fraction s of the total work, so that Tp(n) = sTseq(n)+(1-s)Tseq(n)/p. Then

Amdahl's law

SU(p) = Tseq(n)/Tp(n) = 1/(s+(1-s)/p) –> 1/s for p –> ∞, independently of n

Example: Performance

„painting 5 room flat…“ (from Herlihy-Shavit book, apologies)



<u>Scalable parallel algorithm</u>:

- No constant fraction that cannot be parallelized
- Find ways to address the seemingly sequential parts: coordination

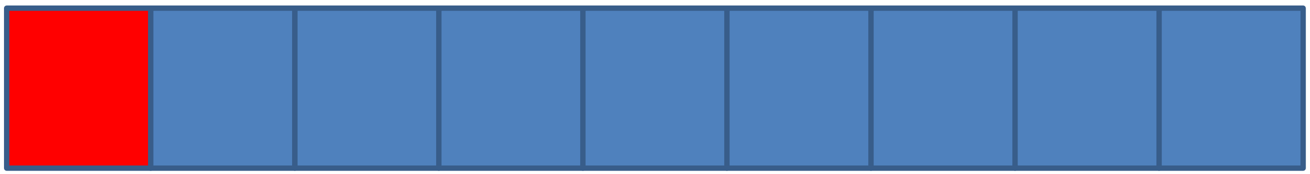
<u>Amdahl's law</u>: If there is a constant fraction (independent of n) of the total work Tseq(n) that cannot be parallelized, then

$$SU(p) = Tseq(n)/Tp(n) = 1/(s+(1-s)/p) \rightarrow 1/s \text{ for } p \rightarrow \infty$$

Example: Performance

„painting 5 room flat…" (from Herlihy-Shavit book, apologies)

Assume sequential part is constant (or slowly growing with n)



Tp(n) = (Tseq(n)-k)/p + k

SU(n) = Tseq(n)/((Tseq(n)-k)/p+k) = p/(1+(p-1)k/Tseq(n))

→ p for Tseq(n) → ∞ and fixed p

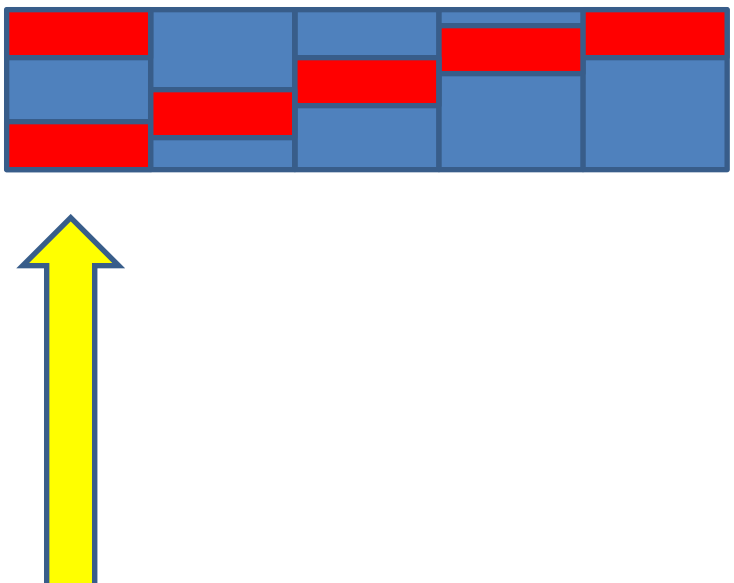
Not Amdahl

Scaled speed-up: A certain problem size needed for required speed-up

Example: Performance

„painting 5 room flat…“ (from Herlihy-Shavit book, apologies)

Non-parallelizable part may be distributed over the algorithm



- Sequential data structures
- Bad coordination constructs: locks (serialization)
- Threads unable to do useful work because blocked (waiting) by other threads
- Work done by "master thread"

Master thread assigned to innocent looking bookkeeping

Example: Performance

„painting 5 room flat…“ (from Herlihy-Shavit book, apologies)

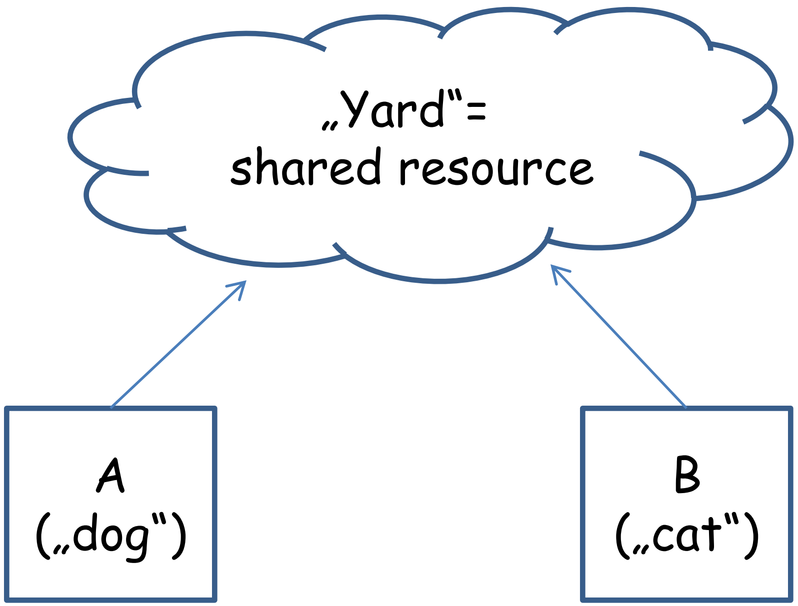
Non-parallelizable part may be distributed over the algorithm

Good parallel algorithm:

- No constant sequential fraction
- Work (total number of operations by all threads) is still O(Tseq(n))

Example: Coordination

„A and B's dog and cat…" (again from Herlihy-Shavit intro)



- A and B need to access infinitely often
- Either A or B has access, but not both (Mutual exclusion)
- If A and B want to access either A or B will succeed (deadlock-freedom)
- If A wants to access, A will eventually get access, independently of B; same for B (starvation freedom; fairness)

Example: Coordination

„A and B's dog and cat…" (again from Herlihy-Shavit intro)

A flag solution, two flags:

A's protocol:

1. Set flagA=1
2. Wait until flagB==0: ENTER
3. Set flagA=0

B's protocol

1. Set flagB=1
2. While flagA==1:
   a) Set flagB=0
   b) Wait until flagA==0
   c) Set flagB=1
3. ENTER
4. Set flagB=0

Example: Coordination

„A and B's dog and cat…" (again from Herlihy-Shavit intro)

A's protocol:

1. Set flagA=1
2. Wait until flagB==0: ENTER
3. Set flagA=0

B's protocol

1. Set flagB=1
2. While flagA==1:
   a) Set flagB=0
   b) Wait until flagA==0
   c) Set flagB=1
3. ENTER
4. Set flagB=0

Satisfies mutual exlusion: Assume not, assume both in yard, and consider last time A and B checked flags. A must have entered before B assigned flagB=1. B's last check must have been after A checked, and assigned flagA=1. Contradicts that B has ENTERED

Example: Coordination

„A and B's dog and cat…" (again from Herlihy-Shavit intro)

A's protocol:

1. Set flagA=1
2. Wait until flagB==0: ENTER
3. Set flagA=0

B's protocol

1. Set flagB=1
2. While flagA==1:
   a) Set flagB=0
   b) Wait until flagA==0
   c) Set flagB=1
3. ENTER
4. Set flagB=0

Satisfies deadlock freedom: If both wants to enter, B will eventually see flagA==1, set flagB=0, and A can enter

Example: Coordination

„A and B's dog and cat…" (again from Herlihy-Shavit intro)

A's protocol:

1. Set flagA=1
2. Wait until flagB==0: ENTER
3. Set flagA=0

B's protocol

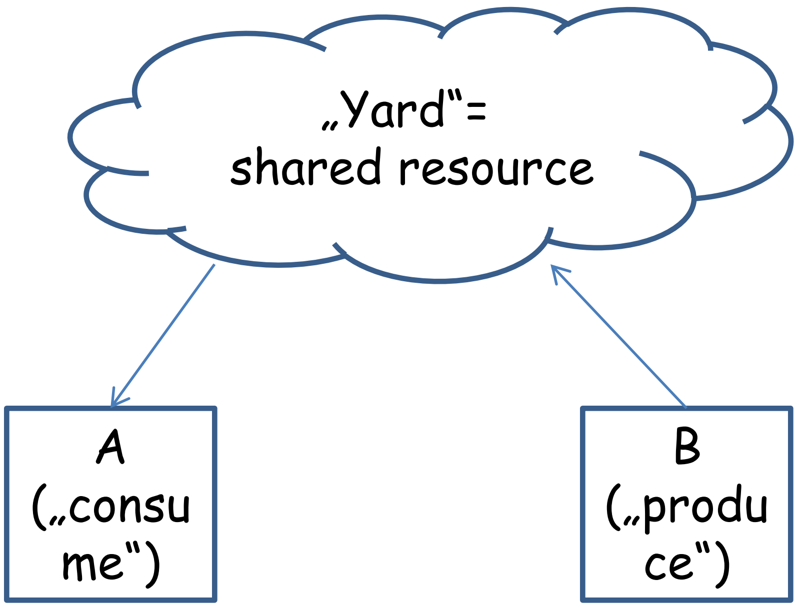
1. Set flagB=1
2. While flagA==1:
   a) Set flagB=0
   b) Wait until flagA==0
   c) Set flagB=1
3. ENTER
4. Set flagB=0

Does not satisfy starvation freedom: B always yields to A…

See also discussion in D. E. Knuth, The art of computer programming, Volume 4, Satisfiability, Addison-Wesley 2015, p. 20-24

Example: Coordination

„A and B's dog and cat…" (again from Herlihy-Shavit intro)



- B produces, A consumes
- Mutual exclusion: when B produces (delivers to resource), A cannot consume (receive from resource); when A consumes, B cannot deliver
- Starvation freedom: if B can produce infinitely often, and A consume infinitely often, both A and B can proceed
- Correctness: A will not consume unless B has produced

Example: Coordination

„A and B's dog and cat…" (again from Herlihy-Shavit intro)

A flag solution, one flag

A's protocol:

1. Wait until flag==0: ENTER
2. Set flag=1

B's protocol

1. Wait until flag==1: ENTER
2. Set flag=0

Satisfies mutual exclusion, starvation freedom, correctness

Example: Coordination

„A and B's dog and cat…" (again from Herlihy-Shavit intro)

A's protocol:

1. Wait until flag==0: ENTER
2. Set flag=1

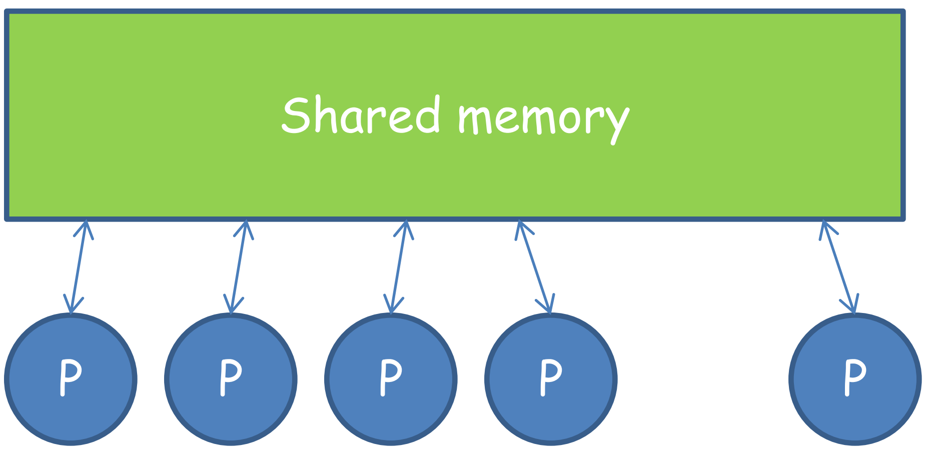
B's protocol

1. Wait until flag==1: ENTER
2. Set flag=0

Mutual exclusion: Initially flag is either 0 or 1. Assume it is 0, then only A can enter, and eventually sets flag to 1 upon exit; A will not enter again before flag is 0, so mutually exclusion holds. In order for flag to become 0, B must have exited (and will not enter again before flag is 1), so mutual exclusion holds. There are no other possibilities for flag to change value…

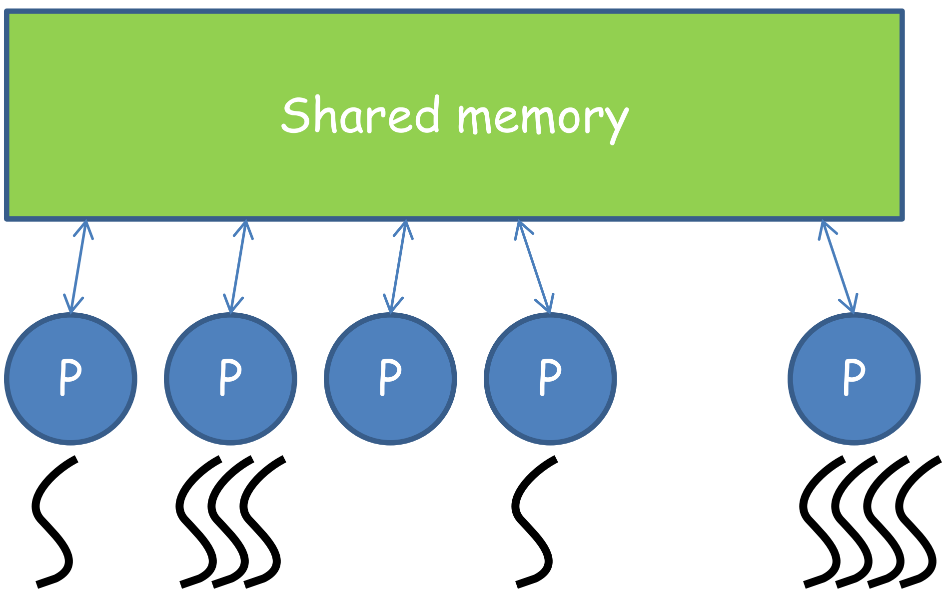
Shared-memory multi-(core-)processor

Homogeneous cores ("processors", P's) that communicate and coordinate through a shared memory; possibly with some hardware support for coordination and synchronization



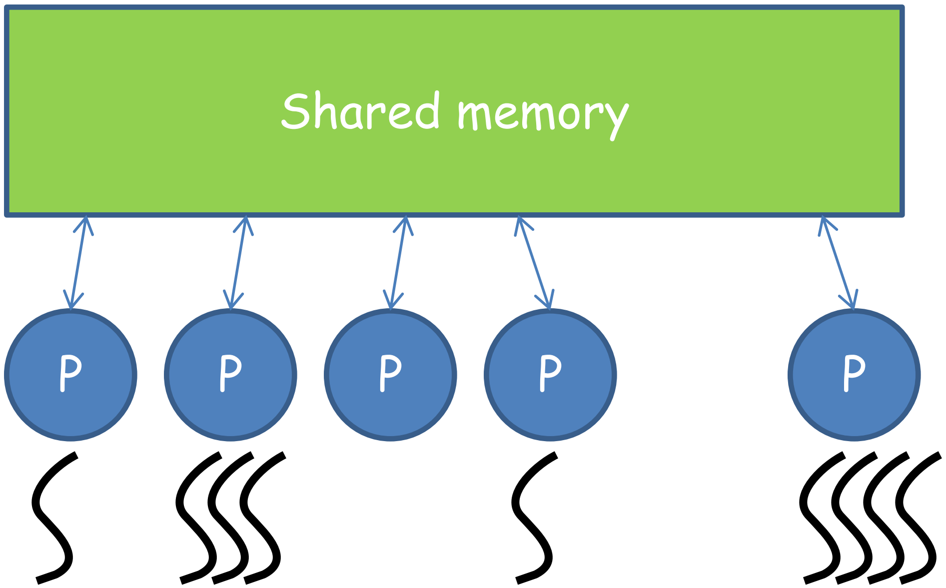
Each processor has own local program, some own local storage, is not synchronized with oher processors: MIMD or SPMD

- Processors (hardware entities) execute processes, processes execute threads (software entities)
- A process can execute zero or more threads
- Threads are uncoordinated and asynchronous. Execution of (processes and) threads can be arbitrarily interleaved, threads can be preempted, interrupted, …



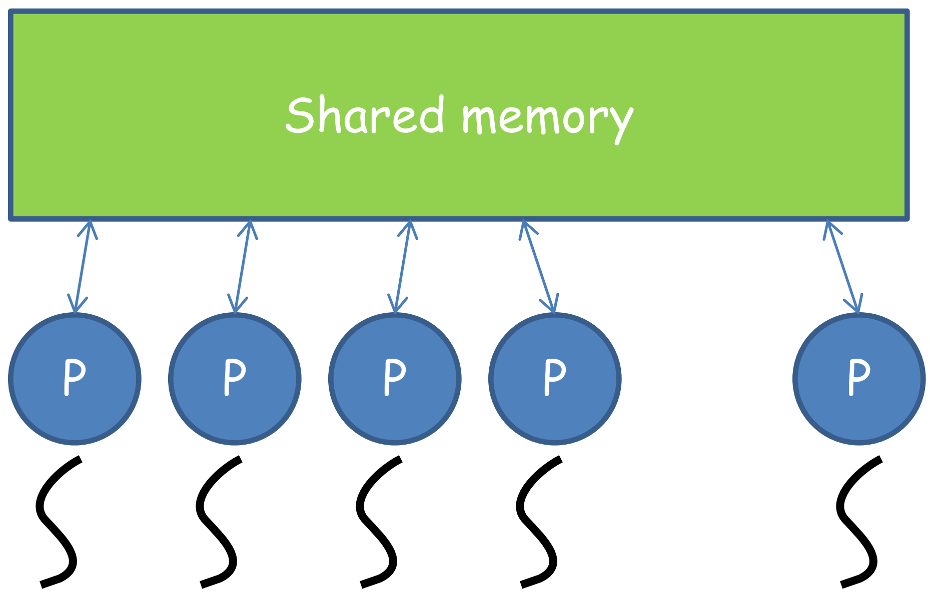
Threads are uncoordinated and asynchronous: Execution of threads can be arbitrarily interleaved, threads can be preempted, interrupted, …

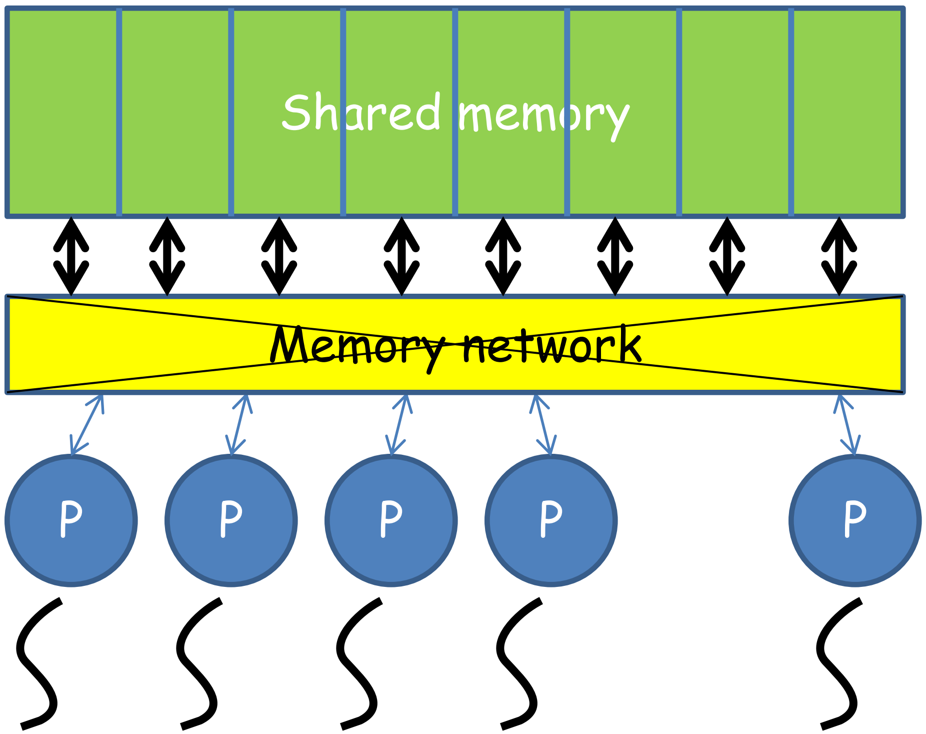
A thread cannot make any assumptions about when an action (“event”) of another thread will happen



A thread cannot make any assumptions about when an action ("event") of another thread will happen

… even in the dedicated case where each processor executes only one thread

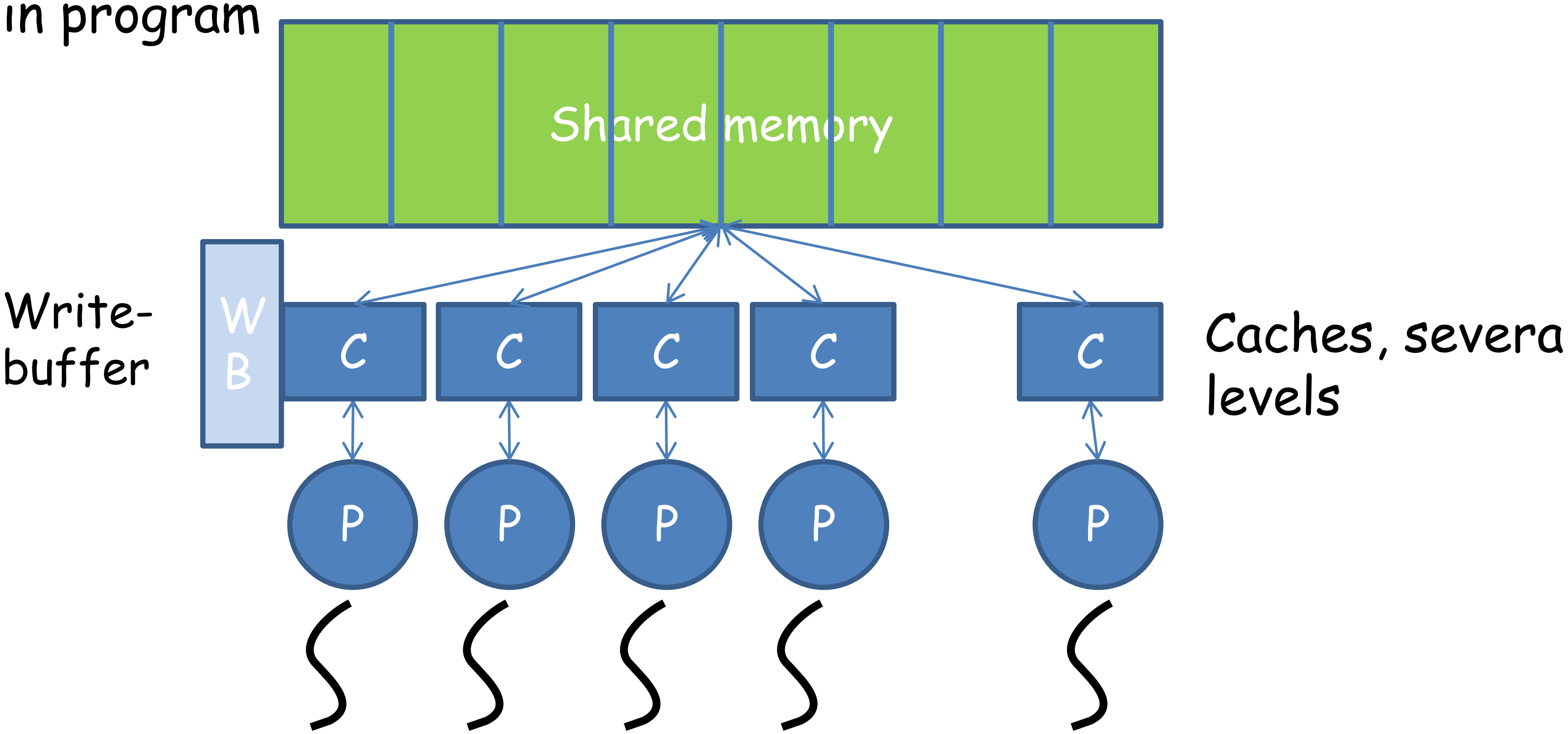




- Memory network
- Banked memory

- NUMA: Non-uniform memory access, access time to memory locations may differ per location and per thread

Writes to memory may be delayed or reordered: One thread may see values from other threads in a different order than written in program



BUT: Possibility to force writes by special flush/memory fence operations

Caches: Small (Kbytes to Mbytes), fast memory with copies of most recently used locations

- Updates in cache of one processor propagated to caches of other processors (cache coherence)?
- Updates in cache propagated to memory (memory consistency)?

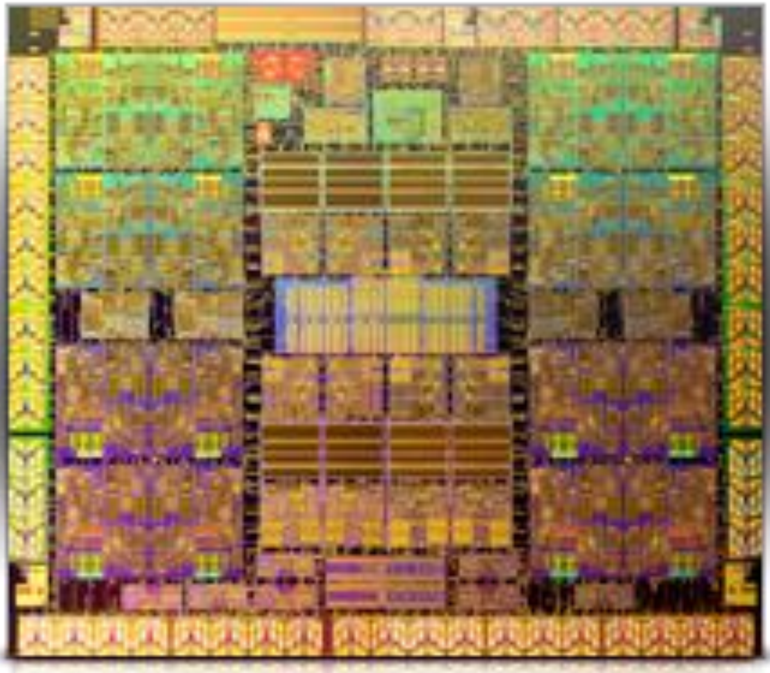
A bit more on when/how/performance issues with caches: Intro. Par. Comp. and HPC lecture

Write buffer: Small buffer storing pending writes (few K)

- Are writes kept in order (and which?)
- When are writes committed to memory (write buffer flush)?

Computer architecture/later this lecture

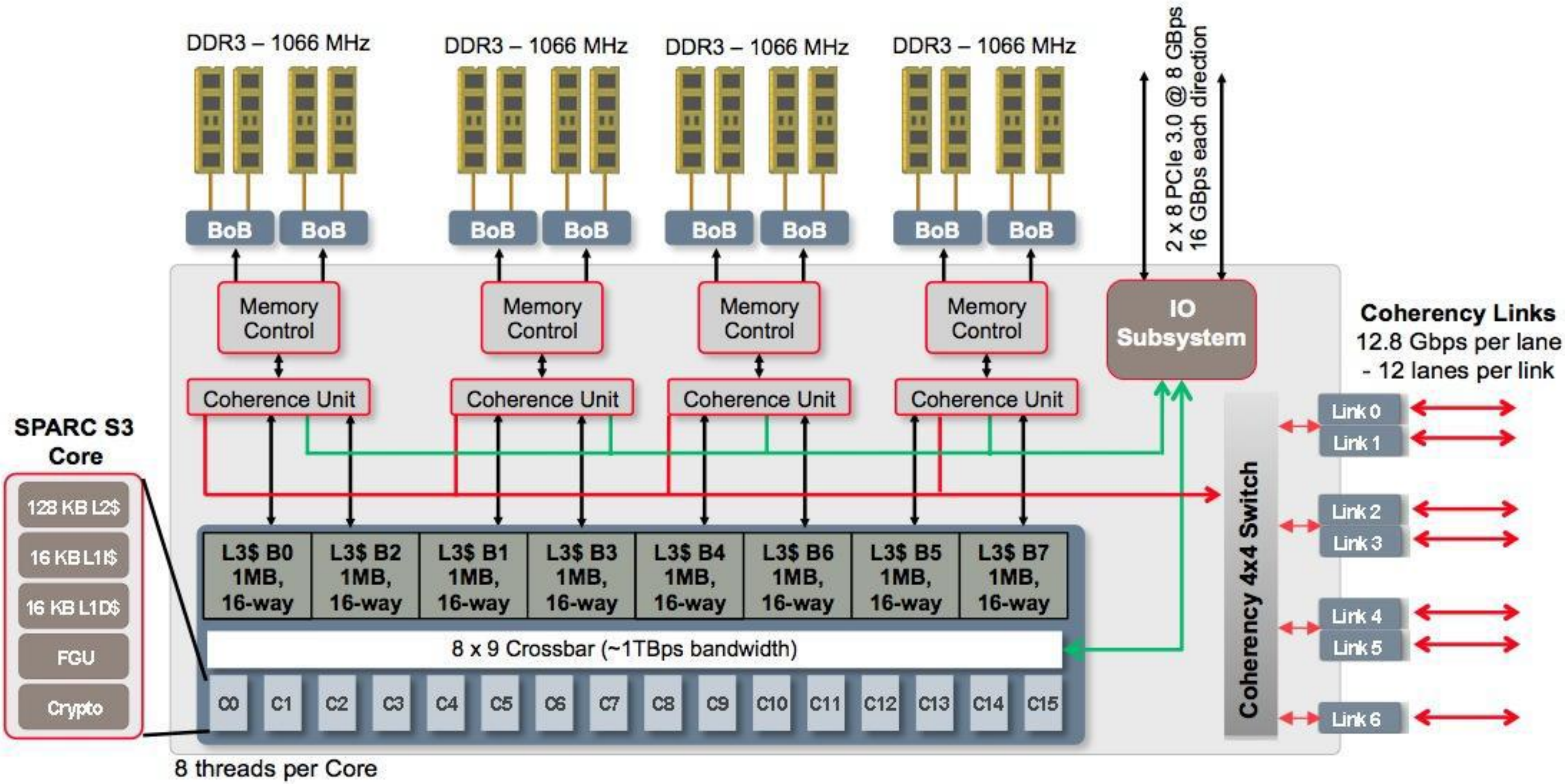
Modern multi-core processor (TU Wien: “Ceres”) ca. 2010



- 4x16 cores, 3.6GHz, 1TB memory
- HW support for 8 threads/core
- Programmer sees 512 threads

- HW thread waiting for memory transfer can be preempted

- Out-of-order execution, branch prediction, prefetching, …


DDR3 – 1066 MHz
DDR3 – 1066 MHz
DDR3 – 1066 MHz
DDR3 – 1066 MHz
2 x 8 PCIe 3.0 @ 8 GBps
16 GBps each direction
BoB
Memory Control
Coherence Unit
IO Subsystem
Coherency Links
12.8 Gbps per lane
- 12 lanes per link
SPARC S3 Core
128 KB L2$
16 KB L1I$
16 KB L1D$
FGU
Crypto
L3$ B0 1MB, 16-way
L3$ B2 1MB, 16-way
L3$ B1 1MB, 16-way
L3$ B3 1MB, 16-way
L3$ B4 1MB, 16-way
L3$ B6 1MB, 16-way
L3$ B5 1MB, 16-way
L3$ B7 1MB, 16-way
8 x 9 Crossbar (~1TBps bandwidth)
C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12 C13 C14 C15
Coherency 4x4 Switch
Link 0
Link 1
Link 2
Link 3
Link 4
Link 5
Link 6
8 threads per Core

Thread A

```
a = 0;
…
a = 1;
if (b==0) { <body> }
```

Thread B

```
b = 0;
…
b = 1;
if (a==0) { <body> }
```

Impossible for both threads to execute <body>:

If A executes body it has set a = 1, and read b==0 so B cannot read a==0. Vice versa…

(BUT: Can easily happen that neither A nor B executes body)

BUT only if writes to a and b are visible in that order to the other thread (and not delayed): Sequential consistency

Sequential consistency (informally):

The result of a concurrent execution of a set of threads is the result of some interleaving of the instructions of the threads as written in the threads' programs (program order)

Memory consistency model formalizes what can and cannot be observed in memory for a multithreaded computation

Sequential consistency is the most well-behaved model: Updates to memory are visible (instantaneous? not necessarily) to other threads in the order as executed by the thread

Modern, shared-memory multiprocessors typically do NOT provide sequential consistency!

(write buffers, caches, memory networks, …; and the compiler!)

Hardware memory consistency models, and how to deal with weaker models:

- Weak consistency
- Relaxed consistency
- Processor consistency
- Release consistency
- Total store order
- …

- Memory fences/barriers
- Compiler options

AMP:
We will study this in more detail!

For the "principles": We (mostly) assume sequential consistency

In "practice" make execution sequentially consistent enough by relying on atomic ordering instructions, and inserting memory fences that enforce writes to take effect

Java: **`volatile, synchronized`** keywords, and other means
C/C++: **`volatile`**, memory fences inserted by hand, *C++11* memory model

CAUTION:

- Too few fences make programs incorrect, with bugs that are sometimes very hard to find
- Too many fences make programs slow (cache invalidations, flushing of write buffers, …)

Assumptions:

Reads and writes ("events") are not simultaneous. Events can always be ordered, one before or after the other

Memory updates are valid, well-formed, consistent:

If thread A writes values a, b, c to variable x which already contains d, and thread B reads x, then B will see either a, b, or c, or d, not some mashup/undefined value

AMP:
We will study this in more detail!

Performance and coordination: An example

Task: compute the primes from 1 to $n = 10^9$

p threads.

Idea 1 (trivial parallelization): divide the interval [1..n] evenly among the p threads, each thread checks for primes in own interval

```
int i = ThreadID.get(); // get local thread ID
block = n/p; // each thread checks a block of integers
for (j=i*block+1; j<(i+1)*block; j++) {
  if (isPrime(j)) { <take action> }
}
```

This AMP lecture: Pseudo-Java

Drawbacks:

1. Primes are not evenly distributed. Prime number theorem(*) says many more primes in [1…n/p] than in [(p-1)*n/p+1…n]

2. Time for `isPrime()` varies (fast for non-primes with small prime factors, slow for large primes)

Thus: No reason to expect good load-balance, some task my be unnecessarily idle

(*) Number of primes smaller than x approx. x/ln x

Idea 2 (coordination): shared "work pool", each thread gets next integer to check from work pool

Use a shared counter to manage work pool

```
class Counter {
  private int value;
  public Counter(int c) { // constructor
    value = c;
  }

  public int getandinc() {
    return value++;
 }
}
```

```
Counter counter = new Counter(1);

int i = 0;
while (i<n) {
  i = counter.getandinc();
  if (isPrime(i)) { <take action> }
}
```

Does this work?

```
return value++;
```

compiles into (something like)

```
int temp = value;
value = temp+1; // may itself be several instructions
return temp;
```

Thread 0

```
temp = value;


value = temp+1;
return temp;
```

Thread 1

```
temp = value;


value = temp+1;
return temp;
```

time

Both threads return the same value!

…and even worse

Thread 0

```
temp = value;




value = temp+1;
return temp;
```

Obsolete value returned

Thread 1

```
temp = value;
value = temp+1;
return temp;

temp = value;
value = temp+1;
return temp;

temp = value;
value = temp+1;


return temp;
```

All increments by thread 1 lost

Classical solution:
Encapsulate the dangerous increment in "critical section", only one thread at a time can be in critical section and perform increment: Mutual exclusion property. Enforce m.e. by locking:

```
public interface Lock {
  public void lock(); // enforce mutual exclusion
                      // acquire lock
                      // enter critical section
  public void unlock(); // leave critical section
                        // release lock
}
```

A thread acquires the lock by executing `lock();` at most one thread can hold the lock at a time; a thread releases the lock by `unlock();`

"Atomic" counter with locks

```
class Counter {
  private int value;
  private Lock lock; // use lock for CS
  public Counter(int c) { // constructor
    value = c;
  }

  public int getandinc () {
    int temp;
    lock.lock(); // enter CS
    try {
      temp = value; // increment alone
      value = temp+1;
      return temp;
    } finally {
      lock.unlock(); // leave CS
    }
  }
}
```

AMP:
We will study this in more detail!

Problems:

- How can lock/unlock be implemented?
- What do locks provide? Will a thread trying to acquire the lock eventually get the lock?
- Are locks a good programming mechanism? Sufficient?

Properties:
Correctness/safety: Lock must guarantee mutual exclusion, at most one thread at a time in critical section
Liveness:
Deadlock freedom: If some thread tries to aquire lock, then some (other) thread will acquire lock. If a thread does not succeed, then other threads must be succeeding infinitely often
Starvation freedom: A thread trying to acquire the lock will eventually get the lock

The problems with locks

Thread 0

```
lock.lock();

<long and
complicated update
of shared data
structure (FIFO,
Stack, list,
Priority queue, hash
map, …>
```

Thread 1

```
lock.lock(); // will idle
```

Threads waiting for lock make no progress

Locks/critical sections easily become a sequential bottleneck. Bad: Amdahl's law

Possible to do better for specific data structures?

What if `lock.unlock();` forgotten?
What if Thread 0 fails? Or thread is preempted indefinitely?

The problems with locks: Assume `Lock` is a perfectly good lock (correct, fair, fast, …) that is used correctly in different parts of the program

Thread 0

```
Lock lock1, lock2;

lock1.lock();

lock2.lock();
… // work here
lock2.unlock();
lock1.unlock();
```

CORRECT (for some reason, two locks are needed)

The problems with locks: Assume Lock is a perfectly good lock (correct, fair, fast, …) that is used correctly in different parts of the program

CORRECT (for some reason, two locks are needed)

Thread 1

```
Lock lock1, lock2;

lock2.lock();

lock1.lock();
… // work here
lock1.unlock();
lock2.unlock();
```

The problems with locks: Assume `Lock` is a perfectly good lock (correct, fair, fast, …); put together correctness is lost

Thread 0

```
Lock lock1, lock2;

lock1.lock();


lock2.lock();
```

Thread 1

```
Lock lock1, lock2;

lock2.lock();

lock1.lock();
```

DEADLOCK!

Locks are error-prone, non-modular: Programs for Thread 0 and Thread 1 could have been written at different times by different programmers following different conventions…

Hardware based solution (to prime computation):

Provide `getandinc(&value);` as special instruction that reads and increments the counter (value) in one atomic, indivisible step

Perfect solution: No locks, each available thread will immediately get next value to work on, no waiting/idling

But global resource (counter)!

Questions:

- Are such instructions possible?
  - YES (see advanced computer architecture book, getandinc, CAS, …)
- Are such instructions equally powerful?
  - NO! These lectures
- How can such instructions be used to provide better data structure support than locks? We will see

Mutual exclusion (Chap. 2)

- Some machinery to reason about correctness
- Two classical solutions with "registers" (memory locations)
- An impossibility result

Assumption:
Atomic registers (see later) or sequential consistency:
Reads and writes are totally ordered events, writes to memory locations (registers) appear in program order, what a thread reads has been written by some other thread previously

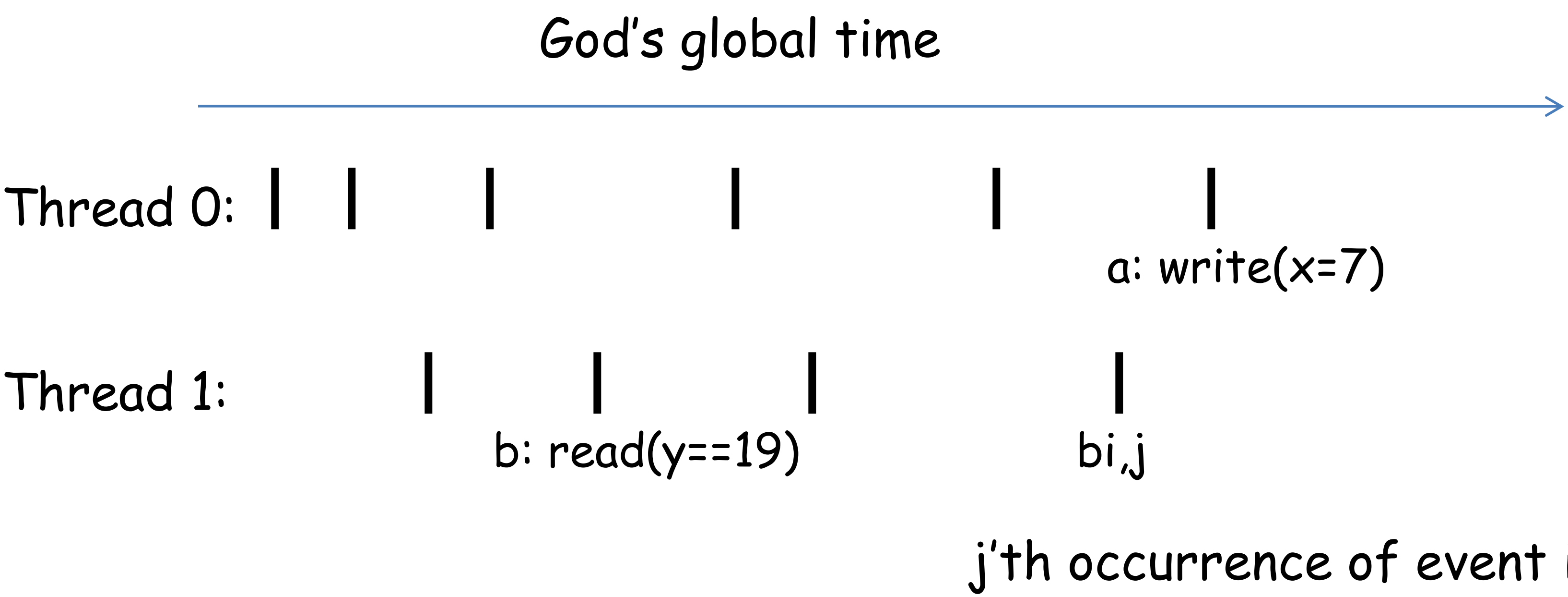
Newtonian (not Einsteinian) time:

There is a common, global time against which the actions of the threads can be ordered

Note: threads normally cannot refer to this time, and will not have to. Time is not an accessible, global clock/timer.

Actions (updates of variables, reading of variables: global state changes) by threads are called events. Events are instantaneous. Events can be ordered temporally, no two events take place at the same time, either is before or after the other: Total order

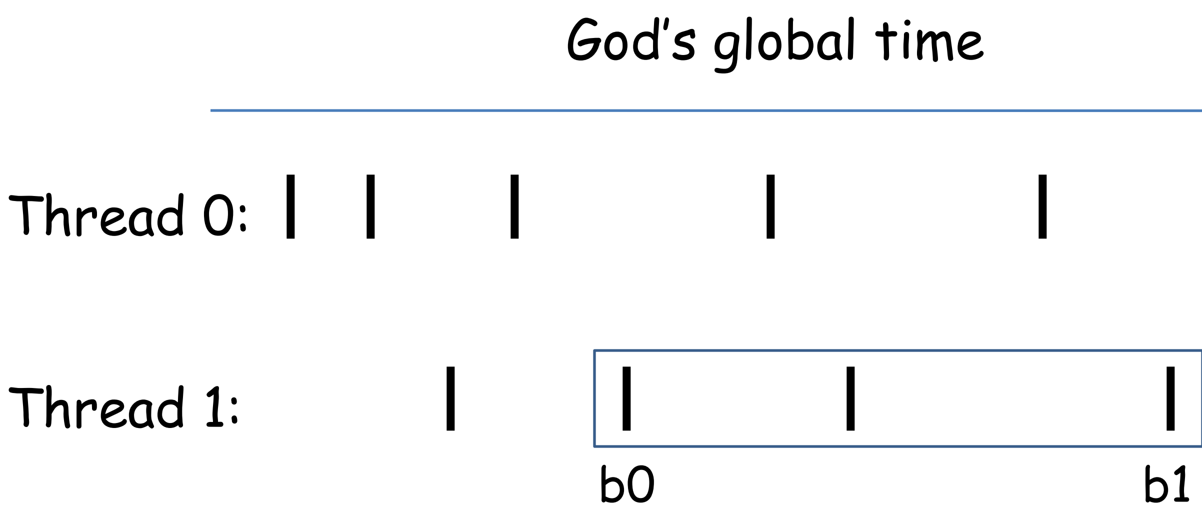
Historically:
Events affect special variables called registers. Updates to registers by one thread are seen in that order by other threads



Total "precede" order on events, e -> f:

- Not a -> a (irreflexive)
- If a -> b, then not b -> a (antisymmetric)
- If a -> b and b -> c, then a -> c (transitive)

- Either a -> b, or b -> a (total)



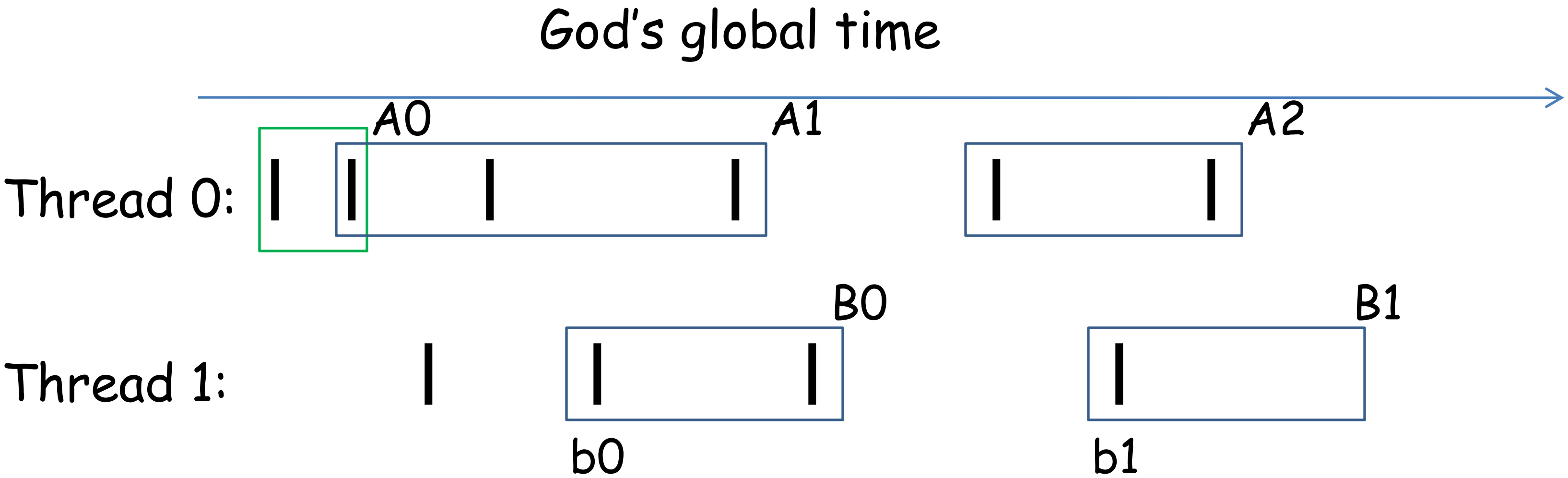
If b0 -> b1, then interval B = (b0,b1) is the duration between b0 and b1

A = (a0,a1), B = (b0,b1), then A -> B iff a1 -> b0

Partial "precede" order on intervals:

- Not A -> A (irreflexive)
- If A -> B, then not B -> A (antisymmetric)
- If A -> B and B -> C, then A -> C (transitive)

- But NOT (either A -> B or B -> A): intervals may overlap and be concurrent (interval order is partial)



- A1 -> A2
- Not A0 -> A1, and not A1 -> A0: Intervals A0 and A1 concurrent
- A0 -> B0
- B0 -> B1, implies A0 -> B1
- Not A1 -> B0, and not B0 -> A1: Intervals A1 and B0 concurrent

Critical Section (CS):

A section of code that can be executed by only one thread at a time: An interval (code) that must not be executed concurrently with certain other intervals (same code).

A program can want to execute CS many times. A program can have different CS, CS', CS''; different critical sections may be executed by different threads concurrently (but each by at most one thread)

E. W. Dijkstra: Solution of a problem in concurrent programming control. Comm. ACM, 8(9), p. 569, 1965.
E. W. Dijkstra: Co-operating sequential processes. In: Programming languages, pp. 43-112. Academic Press, 1965

Let CS(A,k) be the interval in which thread A is in the critical section CS for the k'th time

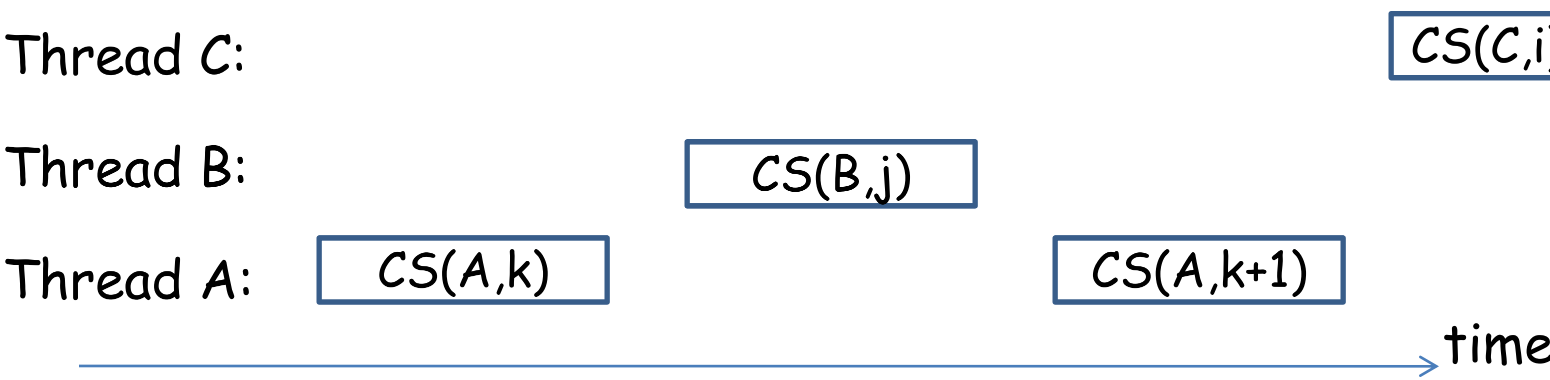
<u>Mutual exclusion property</u>:
For any threads A and B, and any j, k, either CS(A,k) -> CS(B,j), or CS(B,j) -> CS(A,k): critical section intervals are never concurrent

Challenge: Devise lock-algorithms (lock-objects) that can enforce mutual exclusion, and are

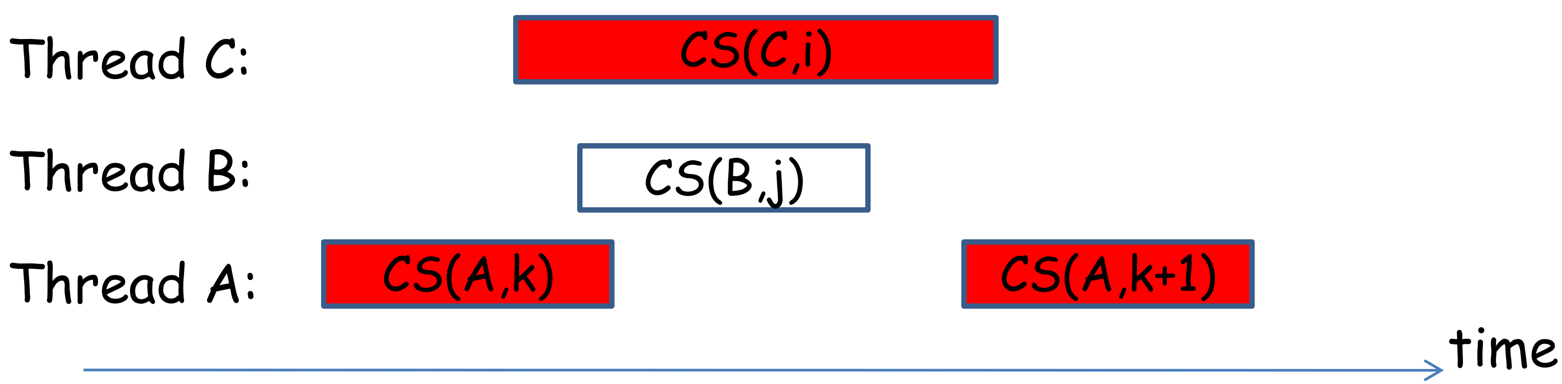
- Correct: At most one thread executes CS (intervals never concurrent, mutual exclusion property)
- Deadlock free: If some thread tries to execute CS, some thread will succeed
- Starvation free: If a thread tries to execute CS it will eventually succeed
- Resource efficient: As few registers as possible

Let CS(A,k) be the interval in which thread A is in the critical section CS for the k'th time

Mutual exclusion property:
For any threads A and B, and any j, k, either CS(A,k) -> CS(B,j), or CS(B,j) -> CS(A,k): critical section intervals are never concurrent



<u>Mutual exclusion property</u>:
For any threads A and B, and any j, k, either CS(A,k) -> CS(B,j), or CS(B,j) -> CS(A,k): critical section intervals are never concurrent



<u>Definitions</u> (dependent or blocking liveness properties):

An operation on a shared object is

- Deadlock free, if when some threads tries to execute CS, some (possibly other) thread will succeed. Conversely, if a thread never succeeds, other threads will succeed infinitely often
- Starvation free, if when a thread tries to execute CS it will eventually succeed,

provided that (all) threads takes steps.

Observation: Starvation freedom implies deadlock freedom

The Peterson Lock

A lock for mutual exclusion on 2 threads

Combination of two ideas. "Building block" for generalization to n threads

G. Peterson: Myths about the mutual exclusion problem. Information Processing Letters, 12(3): 115-116, 1981

Idea 1: Each thread has an own flag to indicate that it wants to enter CS; check that other thread is not entering before entering

```
class Lock1 implements Lock {
  private boolean[] flag = new boolean[2];

  public void lock() {
    int i = ThreadID.get(); // get own thread id
    int j = 1-i; // other thread
    flag[i] = true;
    while (flag[j]) {} // wait for other thread
  }

  public unlock() {
    int i = ThreadID.get();
    flag[i] = false;
  }
}
```

i either 0 or 1

Lemma: Lock1 satisfies mutual exclusion

Proof: By contradiction.

Assume there exists two concurrent CS intervals, i.e., not(CS(A,j) -> CS(B,k)) and not(CS(B,k) -> CS(A,j)).

Consider A's and B's last execution of `lock()` before entering CS(A,j) and CS(B,k)

write(A,flag[A]=true) -> read(A,flag[B]==false) -> CS(A,j)
write(B,flag[B]=true) -> read(B,flag[A]==false) -> CS(B,k)

write(A,flag[A]=true) -> read(A,flag[B]==false) -> CS(A,j)
write(B,flag[B]=true) -> read(B,flag[A]==false) -> CS(B,k)

Also:

read(A,flag[B]==false) -> write(B,flag[B]=true)

since once flag[B] is set to true it remains true, and A could not have read flag[B]==false. By transitivity

write(A,flag[A]=true) -> read(B,flag[A]==false)

Contradiction!

BUT:
Deadlocks if the execution of the two threads is interleaved

write(A,flag[i]=true) -> write(B,flag[j]=true) ->
read(A,flag[j]==true) -> read(B,flag[i]==true) -> FOREVER

Both threads are stuck in while loop repeating the read-events, since no further write events can happen

Idea 2: Use one variable to trade entry to the lock (“victim” for some historical reasons, intuition is “not me” or “you first”)

```
class Lock2 implements Lock {
  private volatile int victim;

  public void lock() {
    int i = ThreadID.get();
    victim = i;
    while (victim==i) {} // wait for other thread
  }

  public unlock() {}
}
```

Lemma: Lock2 satisfies mutual exclusion

Proof: By contradiction. Again, consider A's and B's last execution of `lock()` before entering CS

write(A,victim=A) -> read(A,victim==B) -> CS(A,j)
write(B,victim=B) -> read(B,victim==A) -> CS(B,k)

For A to read victim==B it must be that

write(A,victim=A) -> write(B,victim=B)

since this assignment is the last write by B. And since B now reads, and there are no other writes, this read cannot return victim==A. Contradiction!

BUT:
Deadlocks if one thread runs completely before the other. The lock depends cooperation by the other thread

Thread A

```
lock();



unlock(); return;
```

Thread B

```




lock(); unlock();
lock(); // thread hangs
```

The Peterson lock combines the two ideas and overcomes deadlock

```
class Peterson implements Lock {
  private boolean[] flag = new boolean[2];
  private volatile int victim;

  public void lock() {
    int i = ThreadID.get();
    int j = 1-i; // other thread
    flag[i] = true;
    victim = i;
    while (flag[j] && victim==i) {} // wait
  }
  public unlock() {
    int i = ThreadID.get();
    flag[i] = false;
  }
}
```

Proposition: The Peterson lock satisfies mutual exclusion

Proof: By contradiction. Look at what happens the last time A and B enter CS:

write(A,flag[A]=true) -> write(A,victim=A) -> read(A,flag[B]) -> read(A,victim) -> CS(A,j)

write(B,flag[B]=true) -> write(B,victim=B) -> read(B,flag[A]) -> read(B,victim) -> CS(B,k)

at the moment not knowing the values read for victim and flag.

If A (wlog) was the last to write victim, then

write(B,victim=B) -> write(A,victim=A)

For A to be in CS, it must have read flag[B]==false, so

write(A,victim=A) -> read(A,flag[B]==false)

Here we use atomicity/sequential consistency: events occur/appear in program order

By transitivity

write(B,flag[B]=true) -> write(B,victim=B) -> write(A,victim=A) -> read(A,flag[B]==false)

Contradiction, since there were no other writes to flag[B]

Proposition: The Peterson lock is starvation free

Proof: By contradiction. Assume thread A is waiting forever in `lock();` it waits for either flag[B]==false or victim==B.

3 cases:
Either B is outside of CS, or B is repeatedly entering CS. Or B is also waiting in `lock();`

- B outside of CS: flag[B]==false. Contradiction.
- B reentering CS: it sets victim=B, so A can enter. Contradiction.
- B waiting in `lock();`: victim cannot be both ==A and ==B at the same time, so either thread must enter. Contradiction.

Corollary: The Peterson lock is deadlock free

Proof:
Starvation freedom always implies deadlock freedom

The filter Lock

Extending the idea to n threads. Let there be n-1 locking levels. At each level, roughly

- At least one thread trying to enter level j succeeds
- If more than one thread is trying to enter level j, one is blocked

The Boolean `flag[2]` is replaced by `level[n]`: the level at which thread i is trying to enter. For each level j there is a `victim[j]` keeping track of the last thread that did not enter level j

```
class Filter implements Lock {
  private int[] level;
  private int[] victim;

  public Filter(int n) {
    level = new int[n];
    victim = new int[n];
    for (i=0; i<n; i++) level[i] = 0;
  }
```

Object constructor…

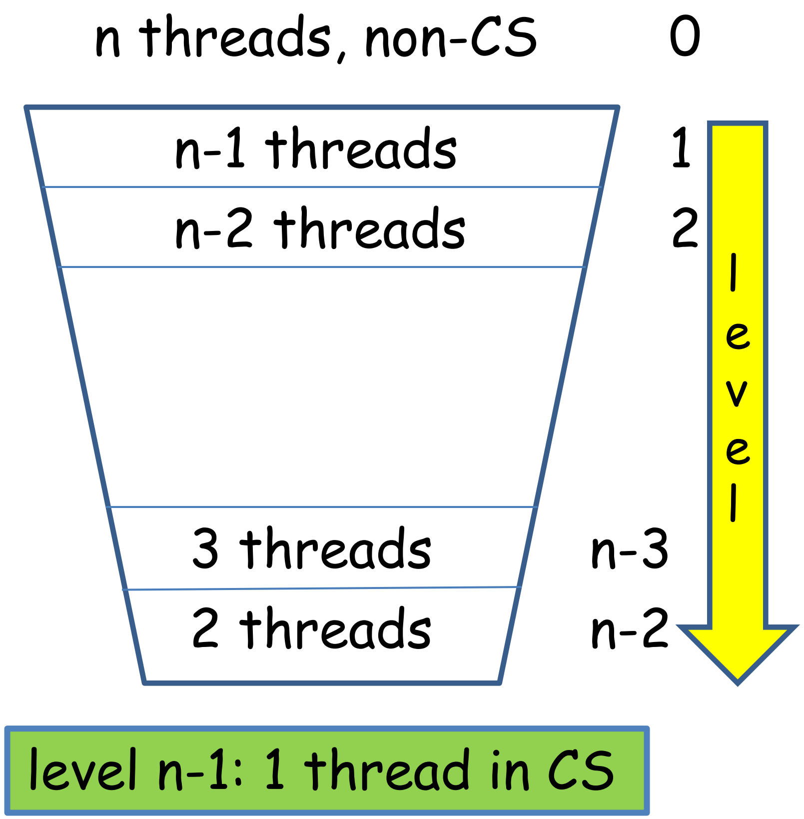
```
class Filter implements Lock {
  private int[] level;
  private int[] victim;

  public void lock() {
    int i = ThreadID.get();
    for (int j=1; j<n; j++) { // try to enter level j
      level[i] = j;
      victim[j] = i;
      while (EXIST k≠i:level[k]>=j && victim[j]==i);
  }

  public unlock() {
    int i = ThreadID.get();
    level[i] = 0; // through the filter
  }
}
```

Note:
Checking n level values and a victim is not assumed to be done atomically

```
while (EXIST k≠i:level[k]>=j && victim[j]==i);
```

is shorthand for

```
for (k=0; k<n; k++) {
  if (k==i) continue;
  while (level[k]>=j && victim[j]==i);
}
```



Intuition:

At most n-j threads can complete level j, j≥1, and proceed to level j+1.

At each level, any thread attempting CS will eventually succeed (proceed to next level)

Note:
For n=2, Filter lock equivalent to two-thread Peterson

Proposition: The Filter lock satisfies mutual exclusion

Proof:
Thread i has reached level j when it starts iteration j of the for loop, and has completed level j when it exits

```
while (EXIST k≠i: level[k]≥j && victim[j]==i);
```

When thread i completes level j, it can reach level j+1. By induction, we show that for j, $0 \le j < n$, at most n-j threads can complete level j.

Observe that j=n-1 implies mutual exclusion in CS

Induction hypothesis: At most n-j threads can complete level j.

Base case, j=0, trivial, the n threads not in CS

Assume hypothesis, at most n-(j-1) threads have completed level j-1, j≥1. At level j, at most n-(j-1) = n-j+1 threads enter. Have to show that at least one thread cannot complete level j. Assume the contrary: All n-j+1 threads complete level j.

Let A be the last thread to write `victim[j]`, so for any other B:

write(B,level[B]=j) ->
write(B,victim[j]=B) -> write(A,victim[j]=A)

B has now reached level j or higher, so A will read level[B]>=j and therefore cannot complete level j. Contradiction.

Proposition: The Filter lock is starvation free

Proof:
Will have to show that any thread that wants to enter CS will eventually succeed. By induction on levels in reverse order

Induction hypothesis: Every thread that enters level j or higher, will eventually enter CS

Base case, level j=n-1 contains only one thread, in CS

Assume thread *A* remains stuck at level j. Thus victim[j]==*A* and level[B]≥j for at least one B.

Two cases:

- Some thread B sets `victim[j] = B`. Since *A* is the only thread that can set `victim[j] = A`, this contradicts that *A* is stuck
- No thread sets `victim[j] = B`. Thus, there is no thread entering level j (such B would set victim[j] = B) from below. By the induction hypothesis, all threads at level j or higher will eventually enter CS. When this happens, eventually level[B] becomes smaller than j (first reset to 0), contradicting that level[B]≥j, so *A* can enter level j+1

Corollary: The Filter lock is deadlock free

(since starvation freedom implies deadlock freedom)

However:

Proposition: The Filter lock is not fair (what does that mean?)

Exercise: Show that threads can overtake up to a certain number of times

Peterson and Filter locks have nice properties (mutual exclusion correctness, starvation free)

BUT:

1. 2n shared integer variables for n-thread mutual exclusion
2. Weak liveness properties (starvation only means "eventually")

Can this be improved?

Aside: Dekker's algorithm (as quoted by Dijkstra, ca. 1965)

```
class Dekker implements Lock {
  private Boolean[] flag = new Boolean[2];
  private volatile int turn = 0;

  public void lock() {
    int i = ThreadID.get(); // get own thread id
    int j = 1-i; // other thread
    flag[i] = true;
    while (flag[j]) {
      if (turn!=i) {
        flag[i] = false;
        while (turn!=i) {}
        flag[i] = true;
      }
    }
  }
```

Home exercise: Prove correctness and liveness of Dekker's algorithm

```
class Dekker implements Lock {
  private Boolean[] flag = new Boolean[2];
  private volatile int turn = 0;

  public unlock() {
    int i = ThreadID.get();
    turn = 1-i;
    flag[i] = false;
  }
}
```

Similar to Peterson; according to Dijkstra, the(?) first correct solution to the mutual exclusion problem (before 1965)

Aside: Mutual exclusion and bypassing

Peterson's/Dekker's algorithms makes it possible for processes to overtake each other (see exercises); this can be avoided, e.g.,

K. Alagarsamy: A mutual exclusion algorithm with optimally bounded bypasses. Inf. Process. Lett. 96(1): 36-40 (2005)

Lamport's Bakery algorithm

A mutual exclusion with stronger fairness guarantees (first-come-first-served, FIFO, …)

Idea: Take a ticket that is larger than the ones already in the bakery (or having been served); wait until my ticket is smallest

L. Lamport: A new solution of Dijkstra's concurrent programming problem. Comm. ACM, 17(5): 543-545, 1974
See also: research.microsoft.com/en-us/um/people/lamport

Note: The "Bakery algorithm" in book and here is NOT quite Lamport's Bakery algorithm, more like this:

G. L. Peterson: Observations on l-exclusion. Allerton Conf. on Communication, Control and Computing, pp.568-577, 1990.

Array of n flags for each thread to signal need to enter CS; array of n labels for Bakery tickets

```
class Bakery implements Lock {
  private boolean[] flag;
  Label[] label; // unbounded integer label
  public Bakery(int n) {
    flag = new boolean[n];
    label = new Label[n];
    for (int i=0; i<n; i++) {
      flag[i] = false; label[i] = 0;
    }
  }
  …
}
```

```
class Bakery implements Lock {
 private boolean[] flag;
  Label[] label; // unbounded integer label
  …
  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0],…,label[n-1])+1;
    while (EXIST k≠i:
              flag[k] &&
              (label[k],k)<<(label[i],i)) { }
  }

  public unlock() {
    flag[ThreadID.get()] = false;
  }
}
```

Take ticket

Wait till smallest

Both computations can be done in arbitrary order, e.g.,

```
max(label[0],label[1],…,label[n-1]);
```

as

```
max = label[0];
for (k=1; k<n; k++) if (max<label[k]) max = label[k];
```

Observations:

The sequence of labels for each thread are strictly increasing

Two (or more) threads trying to acquire the lock may generate the same label. Need to break ties between such threads:

Standard tie-breaking idea: Use thread id

Note:
The label computation is not atomic ("snapshot"); a thread may use labels of others set at different times, thus compute its label from a set of labels that were never in memory at the same time…

Rachel Ruysch (1664-1750),
Akad. Bild. Künste, Wien

Note:
The label computation is not atomic ("snapshot"); a thread may use labels of others set at different times, thus compute its label from a set of labels that were never in memory at the same time...

```
(label[k],k)<<(label[i],i)
```

means lexicographic order, holds iff

```
label[k]<label[i]
```

or

```
label[k]==label[i] && k<i
```

Standard tie-breaking scheme:
If two threads have the same label, the thread with smaller ID "wins"

```
while (EXIST k≠i:flag[k]&&(label[k],k)<<(label[i],i));
```

is not supposed to be atomic, and can be implemented as

```
for (k=0; k<n; k++) {
  if (k==i) continue;
  while (flag[k]&&
         (label[k]<label[i]||
         (label[k]==label[i]&&k<i)));
}
```

Proposition: The Bakery lock is deadlock free

Proof: Assume all threads waiting to enter CS. No labels change. There is a unique least `(label[A],A)` pair. The corresponding thread can acquire the lock. Contradiction

Proposition: The Bakery lock satisfies mutual exclusion

Proof: By contradiction. Assume threads A and B in critical section, and let labeling(A), labeling(B) denote the (non-atomic) sequences of instructions generating the labels. Assume (wlog) that (label[A],A)<<(label[B],B). When B entered it must therefore have read flag[A]==false, so

labeling(B) -> read(B,flag[A]==false) -> write(A,flag[A]=true) -> labeling(A)

which contradicts (label[A],A)<<(label[B],B), since A's label would be at least label[B]+1

Note:
Even though the labeling(A) steps are not atomic, the algorithm is correct (satisfies mutual exclusion).

Even if two threads compute the same `label[i]`, they will be strictly ordered (lexicographically), e.g.,

write(A,flag[A]=true) -> read(A,label) -> write(B,flag[B]=true) -> read(B,label) -> write(A,label[A]) -> read(A,flag[B]==true) -> read(A,label[B]) ->
write(B,label[B]) -> read(B,flag[A]==true) -> read(B,label[A]) ->
… -> read(A,label[B]) -> CS(A,j)

assuming (wlog) that A<B

Fairness: Informally, would like that if A calls `lock();` before B then B cannot overtake A, that is B cannot enter the critical section before A

Divide the lock method into two sections:

- A doorway, whose execution interval D has a bounded number of steps (greater than one)
- A waiting room, whose execution interval W may be unbounded

Definition: A lock is first-come-first-served if, whenever thread A finishes its doorway before thread B starts its doorway, then A cannot be overtaken by B:

If D(A,j) -> D(B,k) then CS(A,j) -> CS(B,k)

```
class Bakery implements Lock {
  private boolean[] flag;
  Label[] label; // unbounded integer label
  …
  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0],…,label[n-1])+1;
    while (EXIST k≠i:
           flag[k] &&
           (label[k],k)<<(label[i],i)) { }
  }
  public unlock() {
    flag[ThreadID.get()] = false;
  }
}
```

Doorway

Waiting room

Proposition: The Bakery lock is first-come-first-served

Proof:
If D(A) -> D(B), then label[A]<label[B] since

write (A,flag[A]) -> write(A,label[A]) ->
read(B,label[A]) -> write(B,label[B]) -> read(B,flag[A]==true)

so B must wait as long as flag[A]==true, since label[B]≥label[A]+1

Corollary: The Bakery lock is starvation free (because any algorithm that is both deadlock free and FCFS is starvation free: Exercise)

Note: Bakery doorway takes $\Omega(n)$ operations

Bakery lock has nice properties (mutual exclusion correctness, first-come-first-served)

BUT:

1. 2n shared integer variables for n-thread mutual exclusion
2. Labels grow without bounds

Can this be improved?

Ad 2: It is possible to construct a bounded, concurrent, wait-free timestamping system, such that a thread can read the other threads time stamp and assign itself a later timestamp. A sequential solution is described in Herlihy/Shavit book.

Bakery lock has nice properties (mutual exclusion correctness, first-come-first-served)

BUT:

1. 2n shared integer variables for n-thread mutual exclusion
2. Labels grow without bounds

Can this be improved?

Ad 2: There are Bakery-like algorithms where labels are bounded, e.g. Black-White Bakery

G. Taubenfeld: The black-white bakery algorithm. Proc. DISC. LNCS 3274, pp. 56-70, 2004

Other remarks:

1. Filter lock: all threads write to victim[i]
2. Bakery: each location flag[i] and label[i] written by only one thread but read by many)
3. Both: even in the absence of contention $\Omega(n)$ locations are read

1-2: Bakery can do with MRSW (Multiple Readers, Single Writer – see later) (atomic) registers, Filter requires MRMW

3: Some „fast path" mutual exclusion algorithms exist

L. Lamport: *A fast mutual exclusion algorithm. ACM TOCS* 5(1): 1-11, 1987

Aside: The original Bakery (from Taubenfeld)

```
public void lock() {
  int i = ThreadID.get();
  flag[i] = true;
  label[i] = max(label[0],…,label[n-1])+1;
  flag[i] = false;
  for (int j=0; j<n; j++) {
    while (flag[j]);
    while (label[j]>0 && (label[j],j)<<(label[i],i)));
  }
}
public unlock() {
  label[ThreadID.get()] = 0;
}
```

Exercise

- Labels can *still* grow unboundedly
- More complicated correctness proof
- Careful with maximum computation, can break correctness

Maximum (correct)

```
max = 0; k = 0;
for (i=0; i<n; i++) {
  m = label[i];
  if (max<m) max = m;
}
```

Maximum (wrong)

Home exercise

```
k = 0;
for (i=0; i<n; i++) {
  if (label[k]<label[i]) k = i;
}
max = label[k];
```

Aside: Implementing register based mutual exclusion

For correct implementation of the register locks in C/C++, correct event ordering and data race freedom (see later) must be ensured. Technically, necessary to use, eg.

```
#include <stdatomic.h>

atomic_int flag[2];
atomic_int victim;
```

Theorem:
Any deadlock-free algorithm that solves mutual exclusion for n threads by reading and writing memory locations (registers) must use at least n distinct locations

Problem with registers: Any value written by a thread can be overwritten, without other threads seeing the first value

Filter and Bakery are thus optimal (within a factor 2)

Exercise: Reduce number of locations to n for the Bakery algorithm

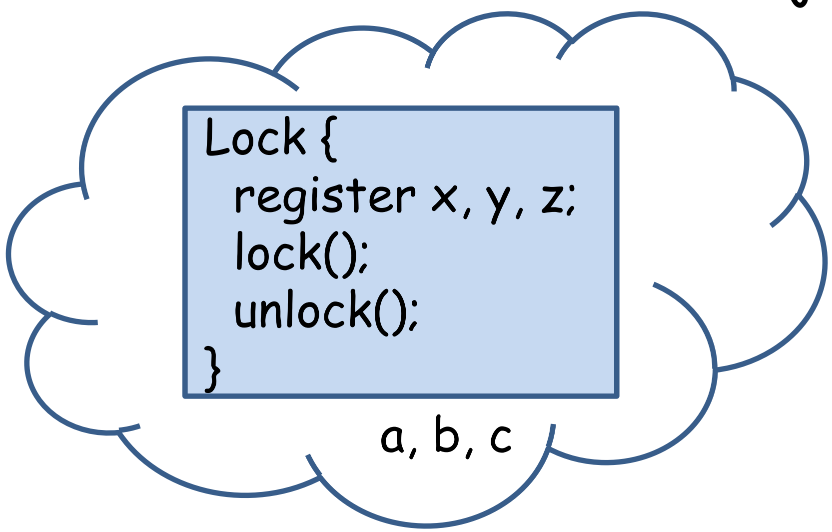
To do better, stronger mechanisms are needed: "hardware support" (later lecture)

Observations for any mutual exclusion algorithm with registers:

1. Any thread A entering CS must write a value to at least one register, otherwise other threads would have no way of determing that A is in CS.

2. If only single-writer registers are used (as in Bakery), at least n such are required

System state: state of all threads and all objects



Object state: state (values) of all object fields

Local thread state: state of all local variables and program counter

Proof (2 threads):
By contradiction, assume 1 register x suffices for 2 threads. Find a scenario where both threads A and B can enter CS.

There is a "covering state" where thread A is about to write to x, but the lock (state) still looks as if no thread is in or trying to enter CS.

Let A be in the covering state, let B run, and enter CS (it can, because the algorithm satisfies mutual exclusion and deadlock freedom). Now, A is resumed, writes x – overwriting whatever B has written to x. Since there is no trace of B anymore in x, A can likewise enter CS. Contradiction!

Base case for induction proof for n threads (not here, but)…

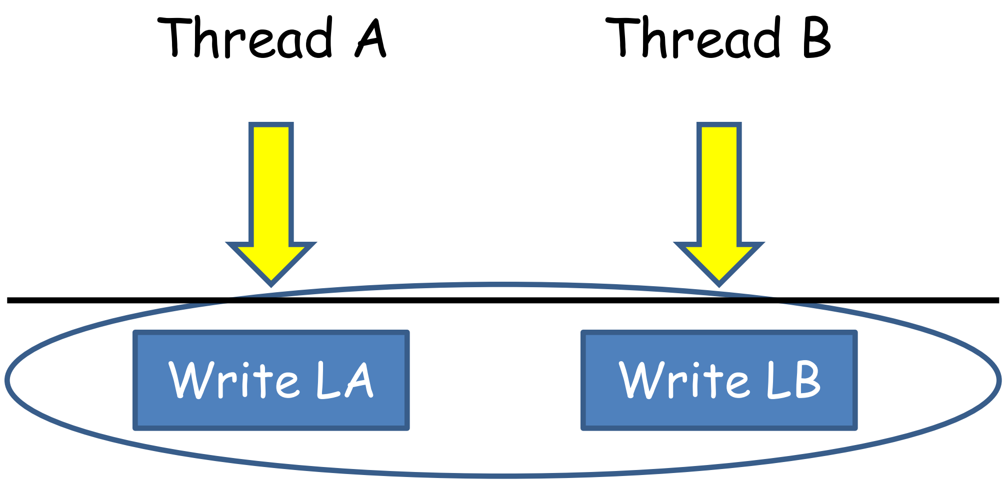
Proof sketch (3 threads):
By contradiction, assume 2 locations suffice for 3 threads.

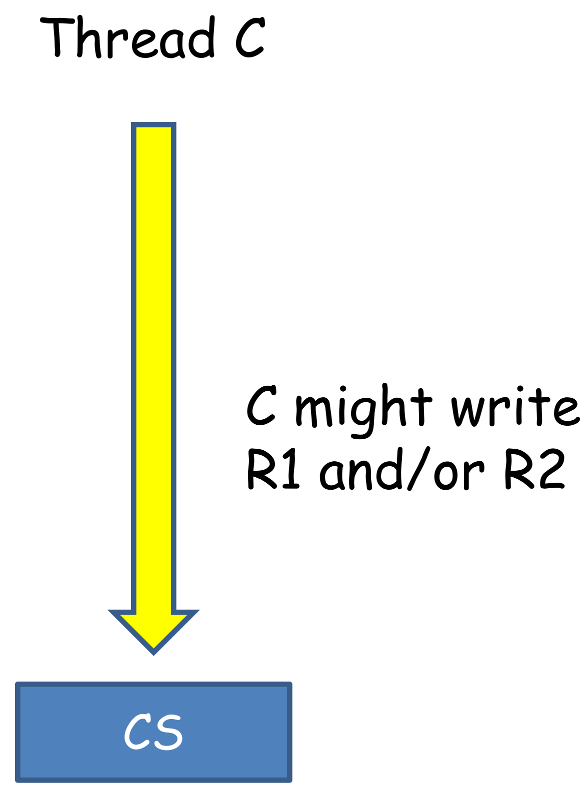
Claim: There is a "covering state" where thread A and B are about to write to the two locations, but the lock still looks as if no thread is in or trying to enter CS.

Let thread C run alone and enter CS. Then let A and B continue, each updates each of the locations, overwriting what C wrote. Now, neither A nor B can determine that C is already in CS.

A or B could enter CS, contradicting mutual exclusion.

Locations: R1, R2. LA either R1 or R2 (the location A is about to write to), likewise LB is either R1 or R2



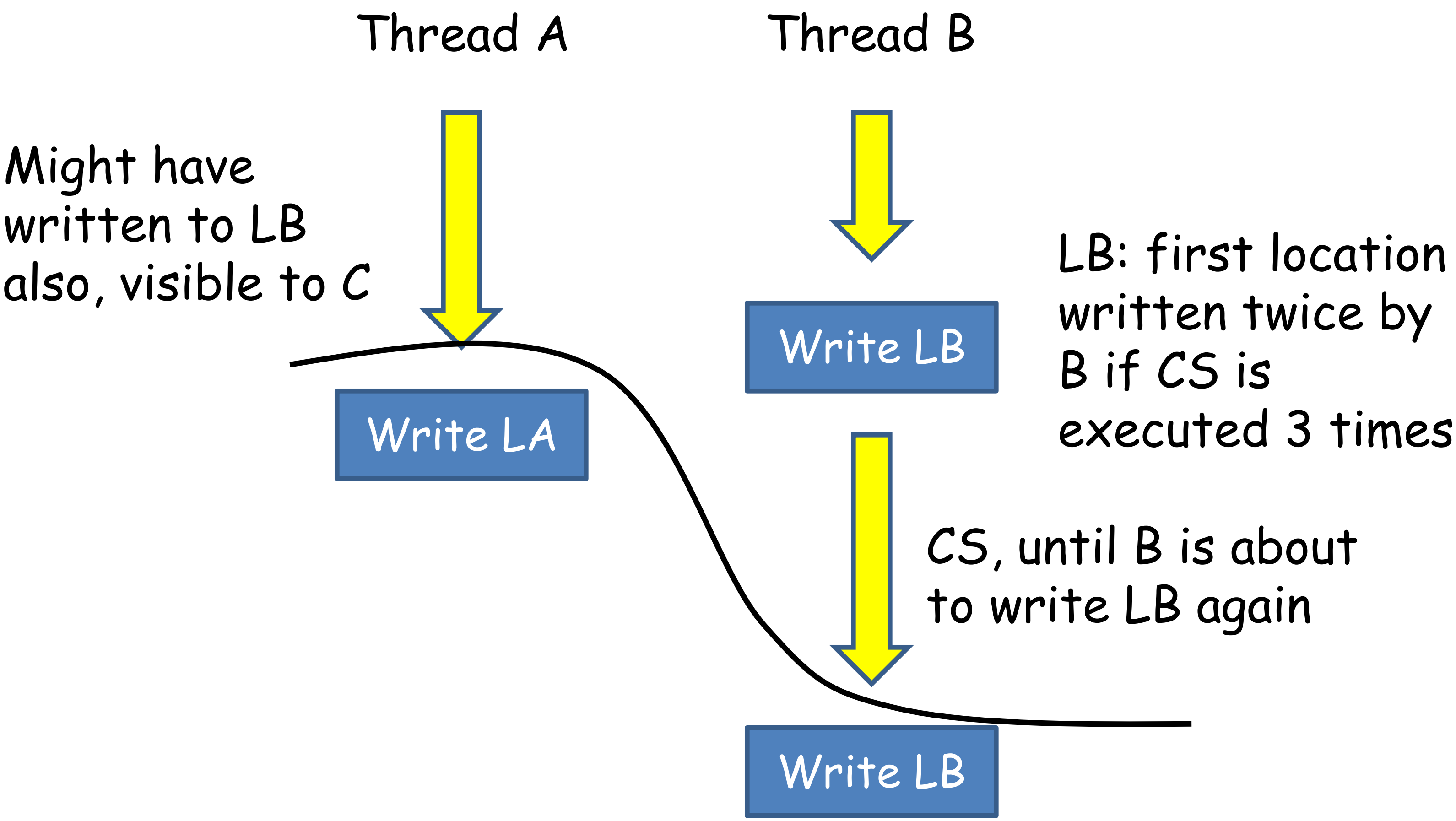


"Covering state": Both locations about to be written, but still possible for thread C to enter CS

"Covering state": Consider execution where B runs through CS 3 times, and consider the first location that B writes to. Since there are only 2 locations, there must be such a first location that is written two times. Call this LB

Let B run to the point just before it writes LB for the first time. This is a state where both A and C can possibly enter CS. If A runs now, it can enter CS since B has not yet written. Let A run to the point just before it writes the other location LA (if it wrote only LB, then B would overwrite, and B would not be able to tell that A is in CS).

Still, A could have written to LB before writing to LA (which could have effect for C). Let B run through CS at most three times until it again covers LB. Then it would have overwritten whatever A might have written to LB, and be in the state where both A and C (and B) could enter CS



Covering state: A and B about to write LA and LB, but no thread in or entering CS

Proof (n threads):

By induction, it is possible to find a covering state that covers k registers, $k \leq n-1$ …

Full proof (combinatorial "covering argument"):
See Nancy Lynch: Distributed Algorithms. Morgan-Kaufman 1996.

Burns, Lynch: Bounds on shared memory for mutual exclusion. Information&Computation 107(2): 171-184, 1993.

Black-white bakery lock

A (simple) solution to the problem of Lamport's bakery algorithm with unbounded labels

Idea: tickets (label's) are colored. A thread that wants to enter CS takes a colored ticket, and waits until it is smallest with that color. Algorithm needs to maintain a global color (bit).

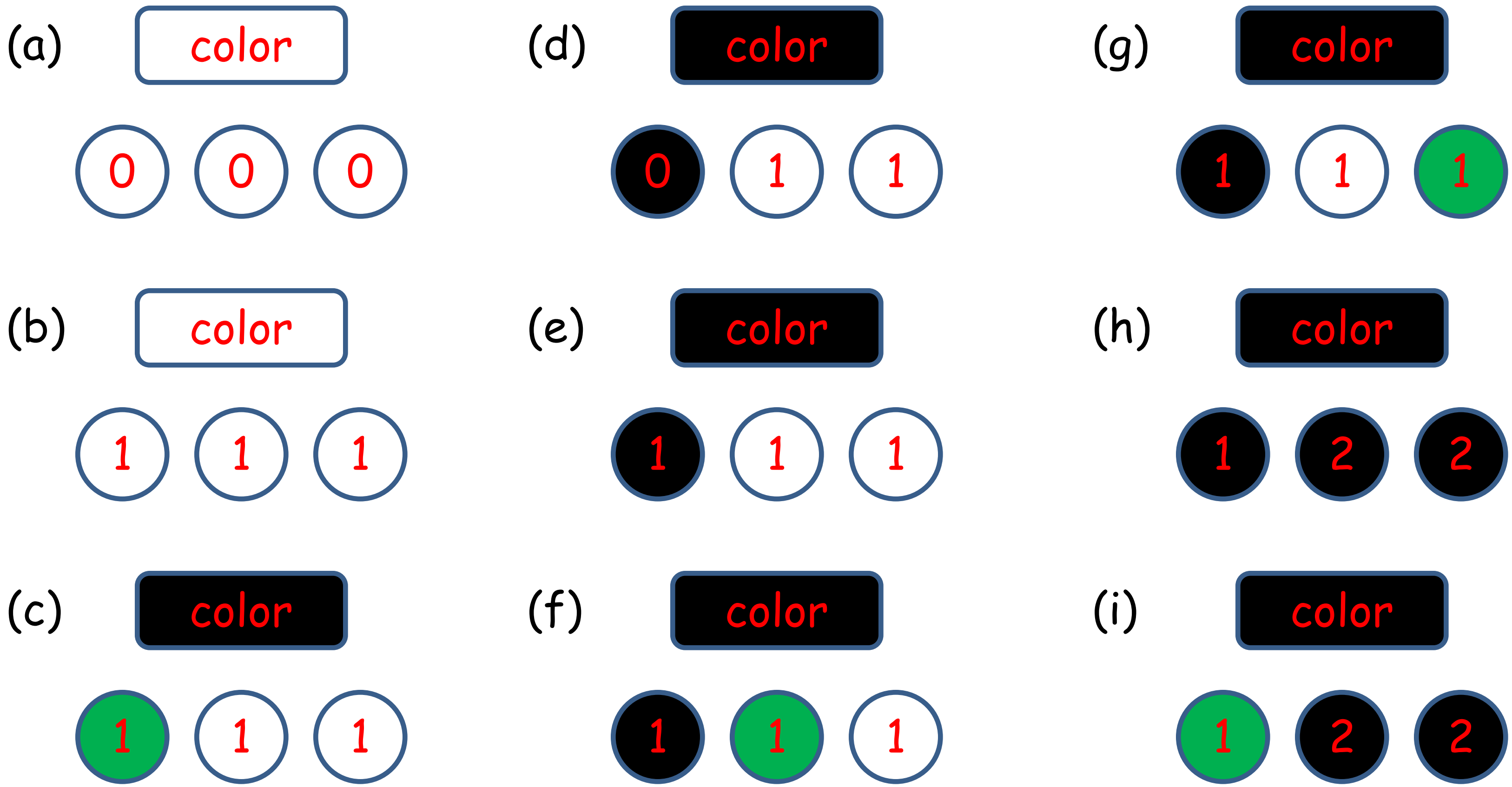
G. Taubenfeld: The black-white bakery algorithm. DISC, LNCS 3274, 56-70, 2004

```
class BlackWhite implements Lock {
  private boolean[] flag;
  Label[] label; // bounded integer label
  Color[] color; Color c;// global color
  public Bakery(int n) {
    flag  = new boolean[n];
    color = new Color[n];
    label = new Label[n];
    c = white;
    for (int i=0; i<n; i++) {
      flag[i] = false; label[i] = 0; color[i] = white;
    }
  }
  public void lock() {
    …
  }
  public unlock() {
    …
  }
}
```

```
public void lock() {
  int i = ThreadID.get();
  flag[i] = true;
  color[i] = c; // set own color from global color
  label[i] =
    max(label[j] | 0<=j<n&&color[j]==color[i])+1;
  flag[i] = false;

  for (j=0; j<n; j++) {
    while (flag[j]);
    if (color[j]==color[i]) {
      while (label[j]>0 && (label[j],j)<<(label[i],i) &&
                color[j]==color[i]);
    } else {
      while (label[j]>0 &&
                color[i]==c && color[j]!=color[i]);
    }
  }
}
```

```
public unlock() {
  int i = ThreadID.get();
  if (color[i]==black) c = white; else c = black;
  label[i] = 0;
}
```

3 threads, all wants to enter CS



Theorem: Black-white bakery algorithm
- satisfies mutual exclusion
- is first-come, first-serve
- uses only bounded registers (label[i] at most n, for n threads)

Proof: Exercise (or Taubenfeld book/paper)

Other bounded register fair locks:

Alex A. Aravind, Wim H. Hesselink: Nonatomic dual bakery algorithm with bounded tokens. Acta Inf. 48(2): 67-96 (2011)
Alex A. Aravind: Yet Another Simple Solution for the Concurrent Programming Control Problem. IEEE Trans. Parallel Distrib. Syst. 22(6): 1056-1063 (2011)
Boleslaw K. Szymanski: A simple solution to Lamport's concurrent programming problem with linear wait. ICS 1988: 621-626
Prasad Jayanti, King Tan, Gregory Friedland, Amir Katz: Bounding Lamport's Bakery Algorithm. SOFSEM 2001: 261-270