HELLENIC REPUBLIC
UNIVERSITY OF CRETE

Distributed Computing

Graduate Course

Section 3: Spin Locks and Contention

Panagiota Fatourou
Department of Computer Science

Spin Locks and Contention

- In contrast to uniprocessor programming, in multiprocessor programming, it is crucial to understand the underlying machine architecture.

- What do you do if you cannot acquire the lock (i.e., if you cannot enter the critical section)?
  - Spinning (or busy waiting)
    - Sensible when you expect the lock delay to be short
  - Blocking
    - Sensible when you expect the lock delay to be long
- Many Operating Systems (OS) apply a combination of both techniques
- Here, we turn our attention to spin locks.

Welcome to Real World: Peterson's algorithm

Experiment

- In practice, once both threads have finished the execution of the 5000000 accesses to the CS, we may discover that the shared counter's final value may be slightly off from the expected value.

⇒ **Even if this is a tiny error, why is there any error at all?**

```
shared int count = 0;

Code for Process p_i

while (lcnt++ < 5.000.000) {
    i = ThreadId.get(); // either 0 or 1
    flag[i] = true
    turn = 1-i
    while (flag[1-i] and turn == 1-i)
        noop;

    count++;          // critical section

    flag[i] = false

    remainder section;
}
```

- Compilers re-order instructions to enhance performance!
  - Program order is preserved for each individual variable but not always across multiple variables.
- Due to hardware, writes to multiprocessor memory do not necessarily take effect when they are issued!
  - Writes to shared memory are buffered in a special write (or store) buffer, to be written to memory only when needed.

Welcome to Real World

Memory Fences (or memory barriers)

- A *memory fence* forces outstanding operations to take effect!
- It is usually an expensive operation!
- Stronger primitives, like Get&Set() or Compare&Swap(), as well as reads and writes to volatile vars do not cause such errors (usually because they are implemented using memory fences).

```
ATOMIC boolean
        Test&Set(MemoryByte *pB) {
   boolean tmp = *pB;
   *pB = 1;
   return tmp;
}

ATOMIC void Reset(MemoryByte *pB) {
   *pB = 0;
}
```

```
ATOMIC int Fetch&Inc(MemoryWord *pW) {
    int tmp = *pW;
   *pW = *pW + 1;
   return tmp;
}
```

```
ATOMIC boolean Compare&Swap(
MemoryWord *pW, Value old, Value new) {
    Value tmp = *pW;
    if (*pW == old) {
       *pW = new;
       return TRUE;
    }
    return FALSE;
}
```

```
ATOMIC Value
    Get&Set(MemoryWord *pW, Value nv) {
     Value tmp = *pW;
     *pW = nv;
     return tmp;
}
```

⇒ Given that memory fences cost as much as synchronization instructions, it may make sense to design ME algorithms directly from such synchronization primitives!

The TAS and TTAS Locks

THE TAS LOCK

```
void lock(MemoryByte *pB) {
      while   (Test&Set(pB))   noop;

            // wait until Test&Set(pB) returns 0
}

void unlock(MemoryByte *pB) {
      reset(pB);
}
```

Theorem
The algorithm above is a correct ME algorithm.

- *Is it possible for some process to starve in this algorithm?*

The TTAS Lock

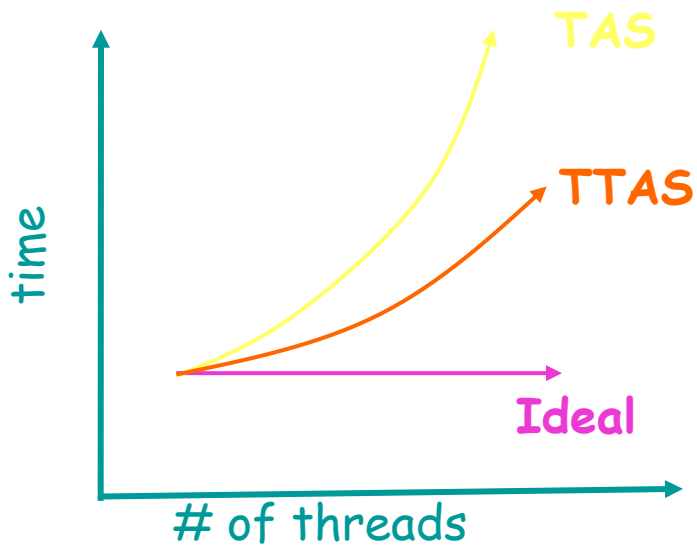
```
void lock(MemoryByte *pB) {
   while (TRUE) {
      while (*pB == TRUE) noop;
      if (Test&Set(pB) == FALSE)
         return;
   }
}

void unlock(MemoryByte *pB) {
      reset(pB);
}
```

Theorem
The algorithm above is a correct ME algorithm.

✓ The TAS and TTAS Locks are equivalent in terms of correctness!

The TAS and TTAS Locks

- But are they equivalent in terms of performance?



- The TAS Lock performs very poorly!
- The TTAS Lock performs substantially better but still falls far short from the ideal!

The TAS Lock – Some remarks

- Each Test&Set() call causes:
  - a broadcast on the bus, and
  - all other processors to discard their own cached copies of the lock!
- Additionally, when the thread holding the lock tries to release it, it may be delayed due to bus traffic caused by the spinners!

The TTAS Lock – Some important points

- What happens the first time a thread B reads the lock?
- What happens each time B rereads the lock (finding it occupied)?
- Is a thread that releases the lock delayed by other threads?
- What happens when the lock is released by its holder thread A?

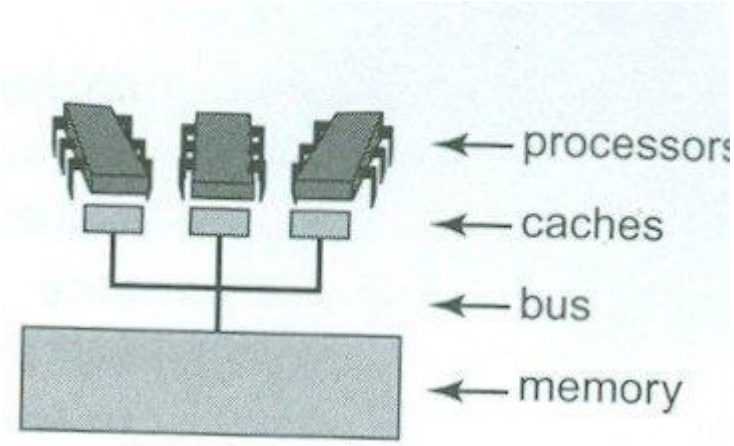


Figure B.4: M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, Morgan Kauffman, 2008

Exponential Backoff

The idea

- If some other thread acquires the lock between the read step and the Test&Set step in the TTAS algorithm, there is probably high contention for the lock. Therefore, we'll back off for some time, giving to the competing threads a chance to finish.

How long should the thread backoff before it retries?

- The larger the number of unsuccessful tries, the higher the likely contention and the longer the thread should backoff.

Strategy

- The thread backoffs for a random duration.
- Each time the thread tries and fails to get the lock, it doubles the expected back-off time, up to a fixed maximum.

```
#define MIN_DELAY …
#define MAX_DELAY …

void BackOff(void) {
   static int limit = MIN_DELAY;
   int delay = rand_next(limit);
    // if 0 < limit < 32, that many low-order bits of the
    // returned value will be independently chosen bit
    //values
   limit = min{MAX_DELAY, 2*limit);
   usleep(delay);
}
```

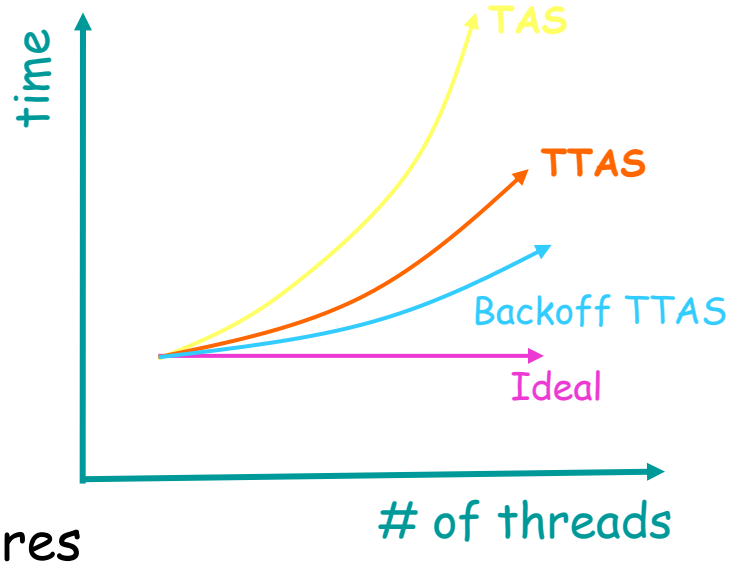
The Exponential BackOff TTAS Lock

```
void lock(MemoryByte *pB) {
   while (TRUE) {
       while (*pB == TRUE) noop;
       if (Test&Set(pB) == FALSE)
                          return;
        else BackOff();
   }
}
void unlock(MemoryByte *pB) {
       reset(pB);     }
```

Exponential Backoff

Advantages

- Easy to implement
- Has better performance than TTAS Lock



Drawbacks

- Must choose parameters carefully
- Not portable across different architectures

Problems

- Cache-Coherence Traffic
  - All threads spin on the same shared location causing cache-coherence traffic on every successful lock access.
- Critical Section Underutilization
  - Threads might back off for too long causing the critical section to be underutilized.

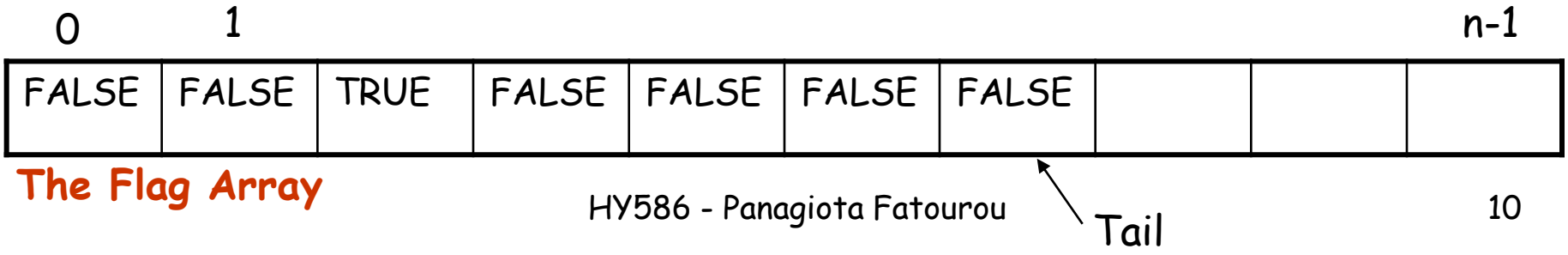
Queue Locks

Idea

- A queue is formed.
  - In a queue, every thread can learn if his turn has arrived by checking whether his predecessor has finished.
  - A queue has better utilization of the critical section because there is no need a thread to guess when is its turn.

Anderson's Algorithm: The Array-Based Lock

```
#define n <number-of-processes>

shared integer Tail = 0;
shared BOOLEAN flag[n] =
        {TRUE,FALSE,FALSE,... ,FALSE};

/* Code for process $p_i$ */
int slot = -1;  /* global variable for process $p_i$; */

void lock(void) {
    slot = Fetch&Inc(&Tail);
    while (!Flag[slot % n]) noop;
}

void unlock(void) {
    flag[slot % n] = FALSE;
    flag[(slot + 1) % n] = TRUE;
}
```

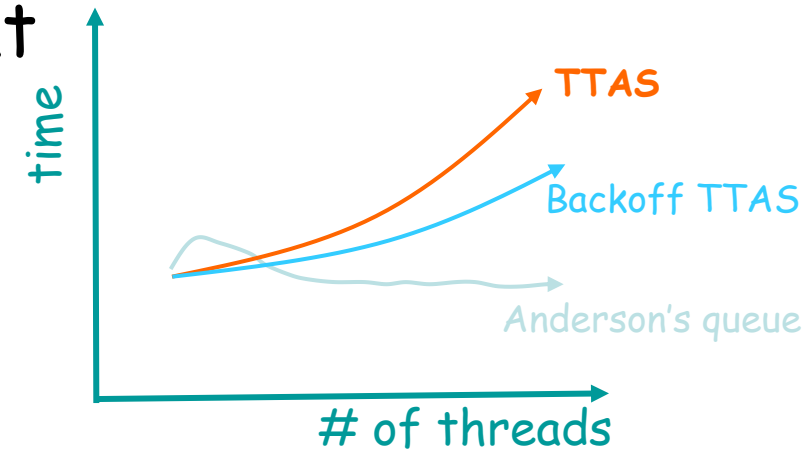


The Flag Array

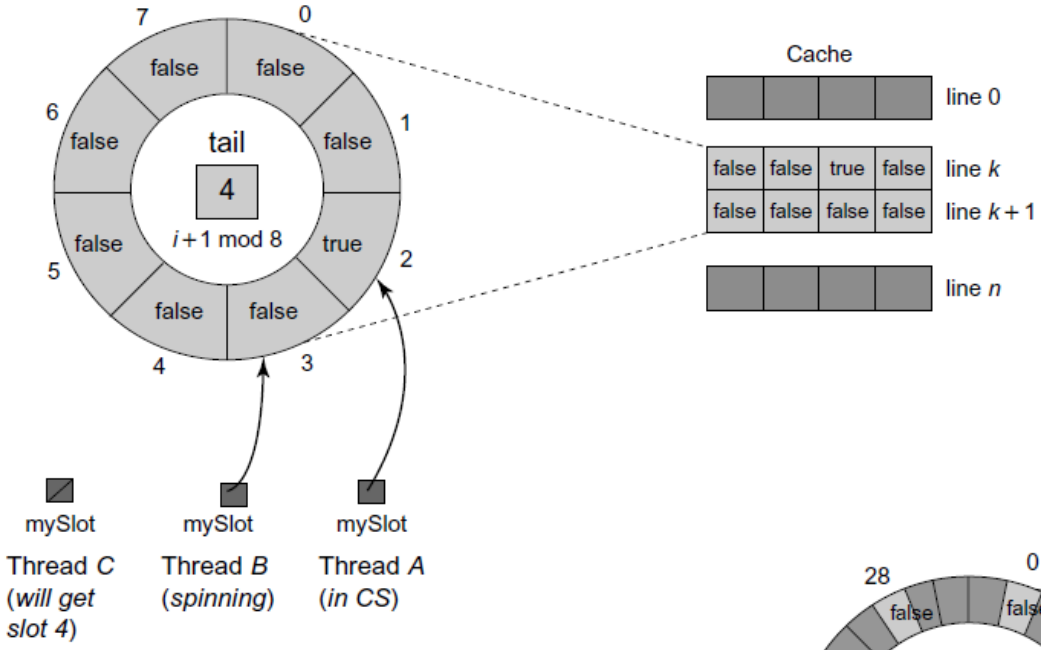
Anderson's Algorithm: The Array-Based Lock

Advantages

- At any given time, each thread spins on its locally cached copy of a single array location
  - Shorter handover than backoff
  - Curve is practically flat
  - Better Scalability
- FIFO Fairness



Anderson's Algorithm: The Array-Based Lock



Drawbacks

- Padding is required, so that distinct elements are mapped to distinct cache lines in order for false sharing to be avoided. ☹

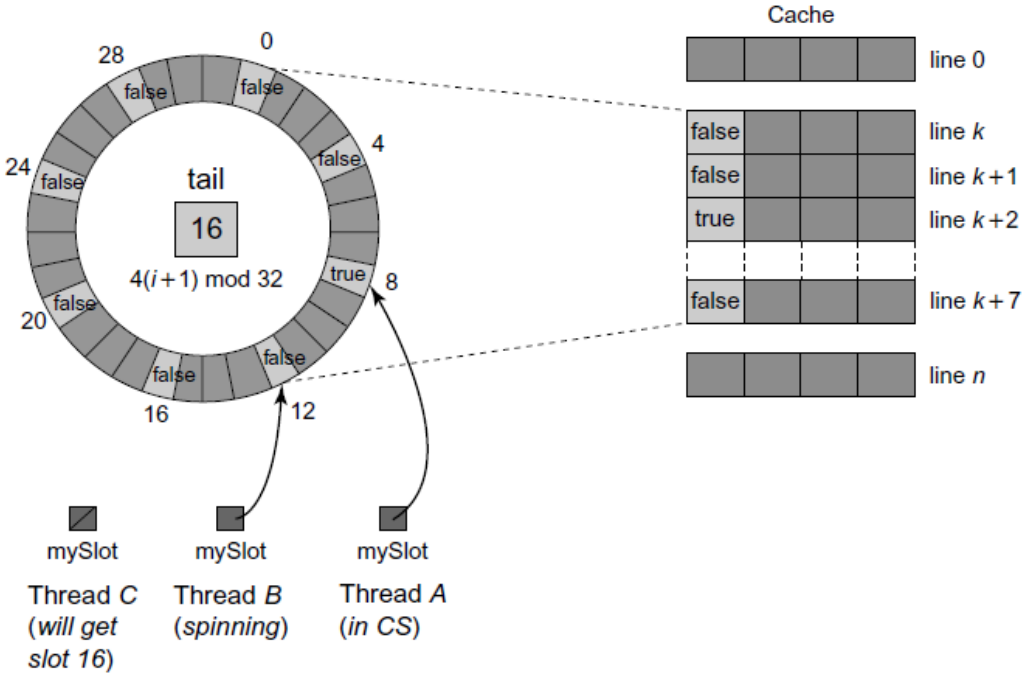


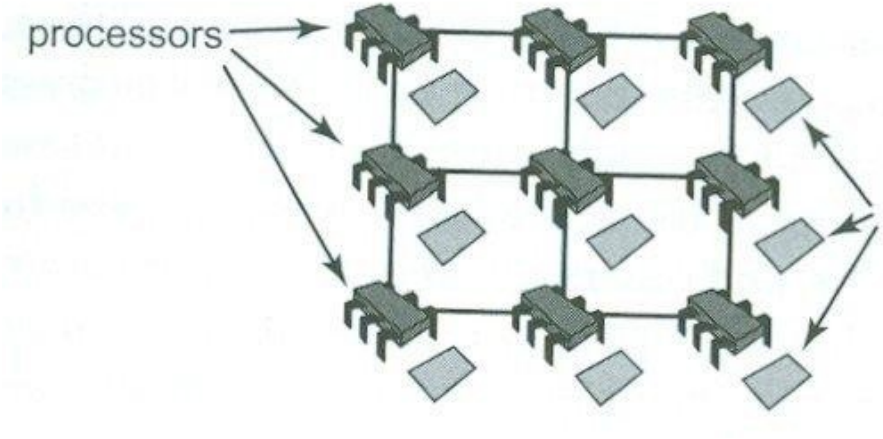
Figure 7.8: M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, Morgan Kauffman, 2008

Anderson's Algorithm: The Array-Based Lock

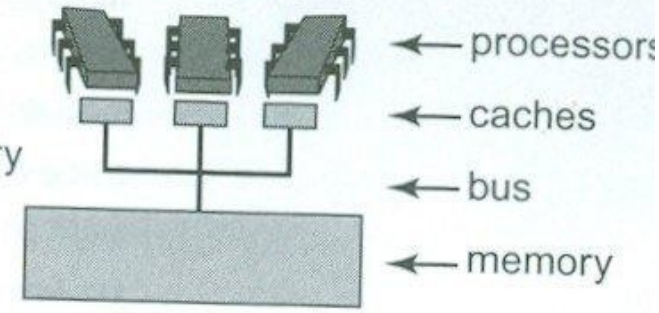
Drawbacks

- Not very space-efficient. ☹
  - It allocates an array of size O(n) per lock. Synchronizing L distinct objects requires O(Ln) space, even if a thread accesses only one lock at a time ☹
- On cache-less NUMA architectures, the algorithm does not have a good performance since spinning during lock() might be performed on a remote variable ☹

Figure B.4: M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, Morgan Kauffman, 2008



Cache-less NUMA architecture



SMP architecture with caches

The CLH Lock
A more space-efficient Algorithm

```
typedef struct node {
        BOOLEAN locked;
} NODE;


shared NODE *Tail;
/* initially, points to a NODE n with n.locked == FALSE */


/* Section of global variables for process $p_i$ */
NODE *MyNode, *MyPred = NULL;
        /* Variable MyNode initially points to a struct NODE

void lock(void) {
        MyNode->locked = TRUE;
        MyPred = Get&Set(&Tail, MyNode);
        while (MyPred->locked == TRUE) noop;
}

void unlock(void) {
        MyNode->locked = FALSE;
        MyNode = MyPred;
                /* recycling of nodes */
}
```

Brief Description

- An implicit list of NODES is created. Each thread allocates just one NODE.
- NODEs are not connected to each other. Rather, each thread refers to its predecessor thread through a (non-shared) global variable, called MyPred.
- NODEs are re-cycled by having each thread using the NODE pointed to by its MyPred Variable as its current NODE the next time it requires the lock.
- An initial NODE is placed in the queue implementing each lock.

The CLH Lock

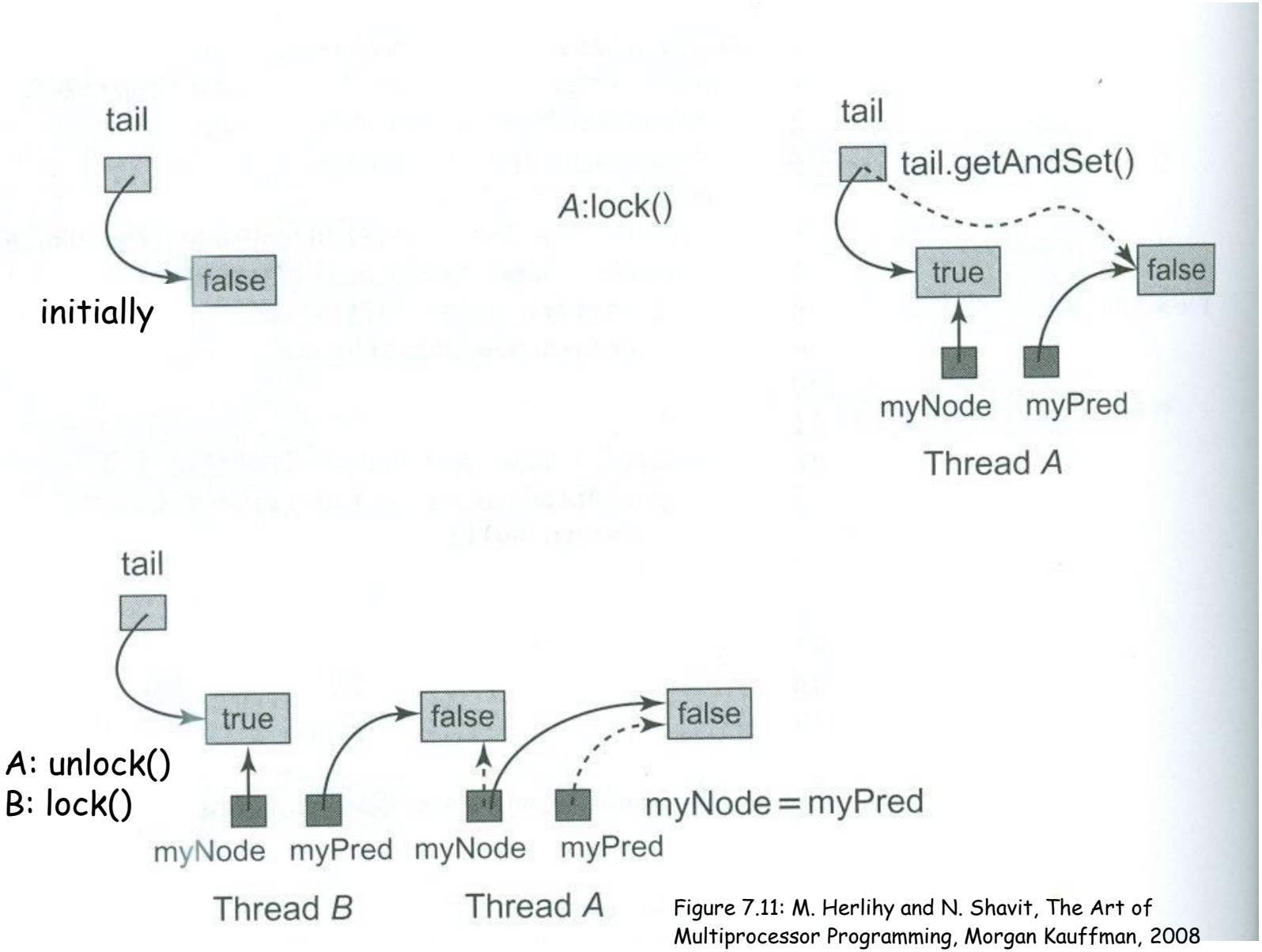


Figure 7.11: M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, Morgan Kauffman, 2008

The CLH Lock

Advantages

- Requires much less space than the Anderson's algorithm
  - Synchronizing L distinct objects requires O(L+n) space (much better than the Array-based Lock).
- Does not require knowledge of the number of processes that might access the lock.
- Has all the performance advantages of the Anderson's algorithm.

Drawbacks

- Like Anderson's algorithm, on cache-less NUMA architectures, the CLH algorithm does not have good performance since it causes a lot of Remote Memory References (RMRs); i.e., spinning is performed on a remote variable.

The MCS Lock

```
typedef struct node {
        BOOLEAN locked;
        struct node *next;
} NODE;

/* shared variables section */
shared NODE *Tail = NULL;

/* Section of global variables for process $p_i$ */
NODE *MyNode, *MyPred = NULL;
/* Variable MyNode initially points to a NODE */

void lock {
        MyPred = Get&Set(&Tail, MyNode);
        if (MyPred != NULL) {
                MyNode->locked = TRUE;
                MyPred->next = MyNode;
                while (MyNode->locked) noop;
        }
}

void unlock {
        if (MyNode->next == NULL) {
                if (Compare&Swap(&Tail, MyNode, NULL) == TRUE) return;
                 while (MyNode->next == NULL) noop;
        }
        MyNode->next->locked = FALSE;
        MyNode->next = NULL;
}
```

Major Ideas

- The created list of NODEs is now explicit.
- Each thread does not spin on the NODE pointed to by MyPred but on that pointed to by MyNode (which is a variable in the thread's local memory)

The MCS Lock

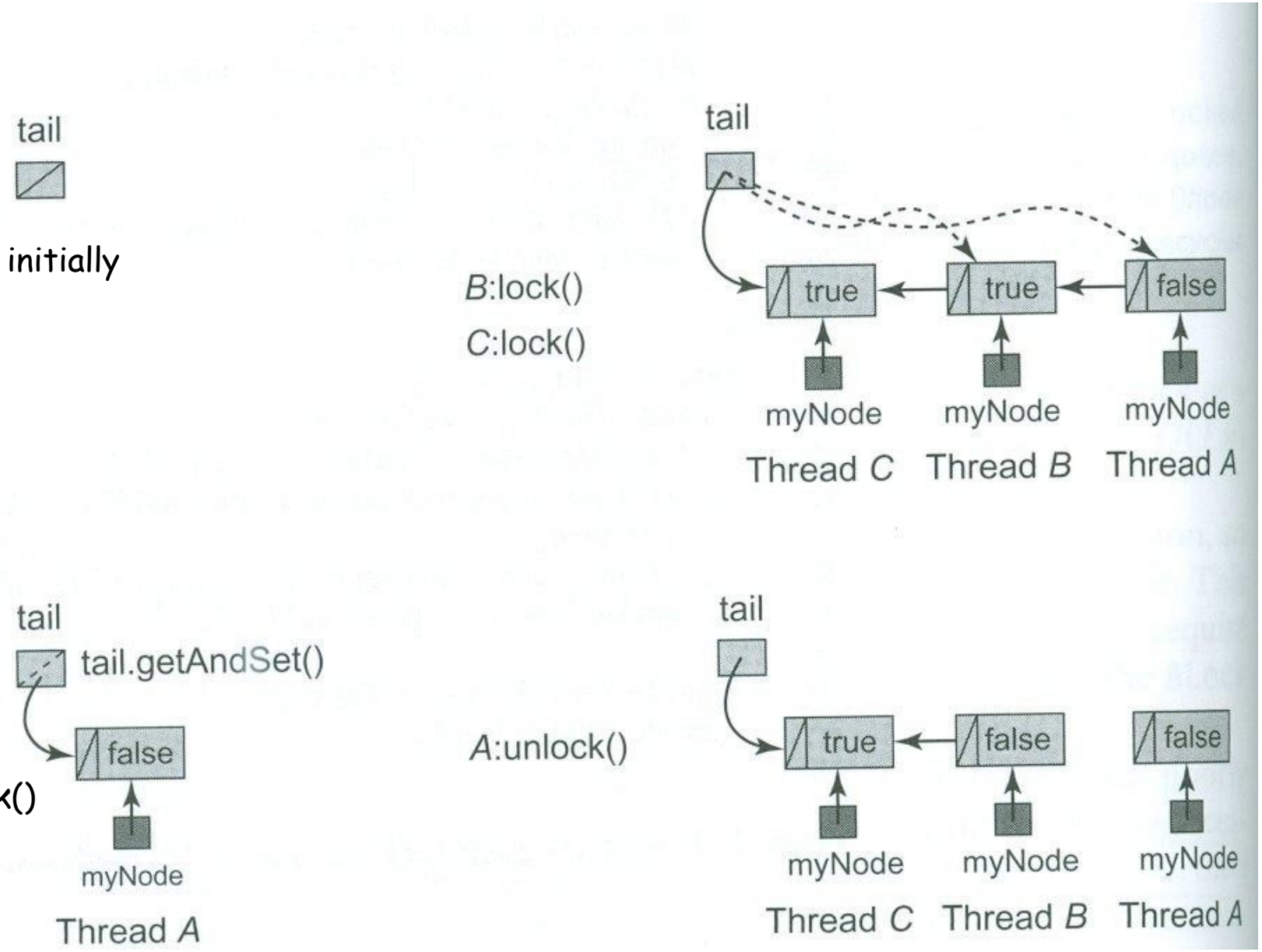


Figure 7.14: M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, Morgan Kauffman, 2008

The MCS Lock

Advantages

- Each unlock causes just one invalidation (as with CLH)
- Lock/Unlock causes just O(1) RMRs on a NUMA architecture
- Recycling can be applied (as in CLH) to obtain space overhead O(n)

Drawbacks

- Releasing a lock requires spinning.
- The algorithm executes more reads and writes to execute lock/unlock, and it requires Compare&Swap() for unlock.

J. Mellor-Crumney and M. Scott, "Algorithms for Scalable Synchronization on shared-memory multiprocessors", ACM Transactions on Computer Systems, 9(1):21-65, 1991.

Lower Bounds

Fetch&Φ Primitives

```
atomic Value fetch&φ(MemoryWord *pW, InputStruct input) {
    old = *pW;
    *pW = φ(old, input);
    return(old);
}
```

- A comparison primitive conditionally updates a shared variable after first testing that its value meets some condition.
  - Compare&Swap()
  - Test&Set()
- Non-comparison primitives update variables unconditionally
  - Fetch&Increment, Fetch&Add
  - Fetch&Store
- **Lower Bound [Anderson & Kim, J. of Parallel and Distributed Computing]**
  Any n-process mutual exclusion algorithm based on reads, writes and comparison primitives causes $\Omega(\log n / \log \log n)$ remote memory references.
- Several algorithms with constant RMR complexity exist when non-comparison primitives are used. A generic algorithm using (any non-comparison) fetch&φ primitive is presented by Anderson and Kim.

Drawbacks of Queue Locks

- They perform well only in cases of no oversubscribing
- If oversubscribing occurs, their performance deteriorates significantly
- Binding may also affect their performance

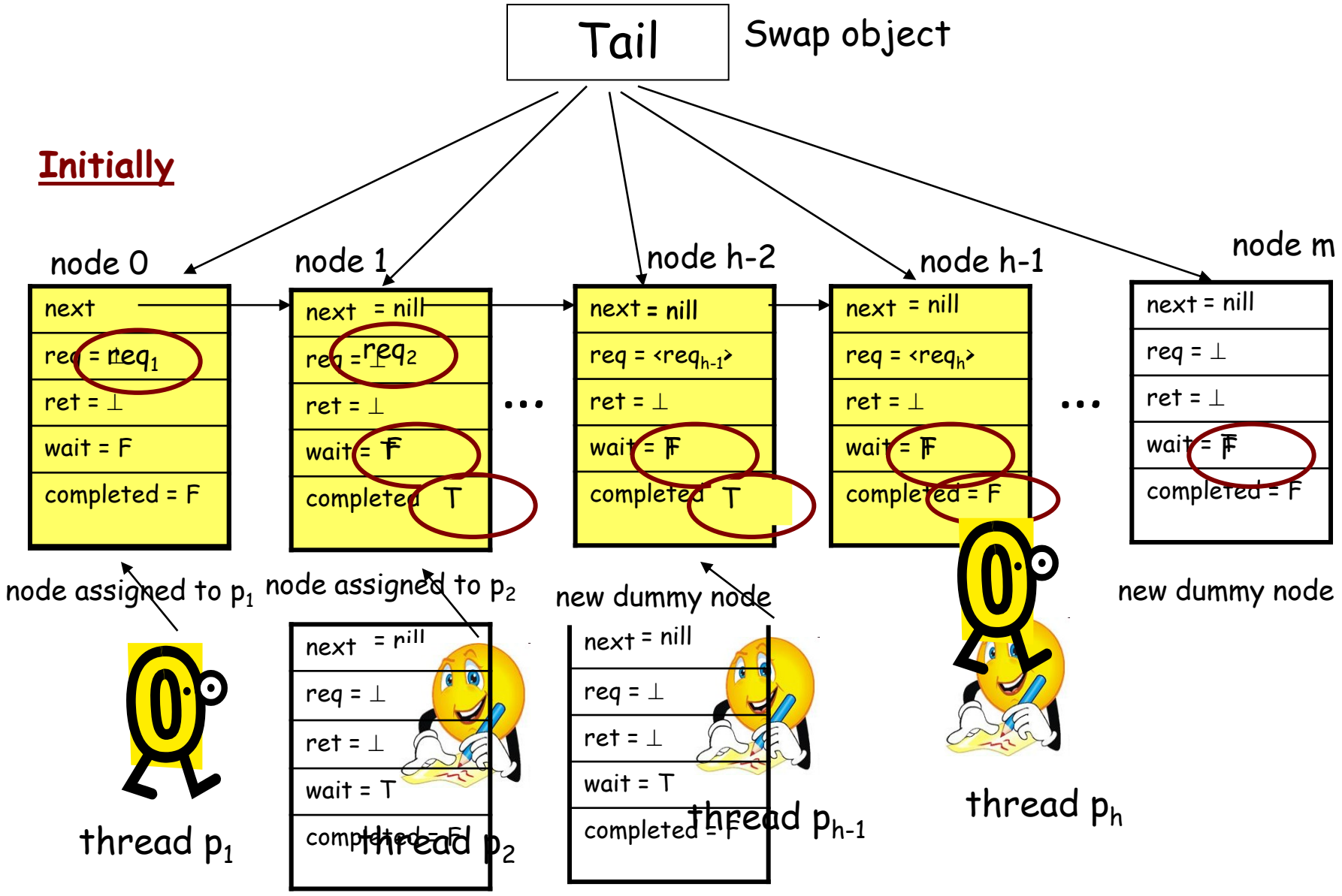
One Lock To Rule Them All?

- TTAS+Backoff, CLH, MCS.
- Each better than others in some way
- There is not one solution
- Lock we pick really depends on:
  - the application
  - the hardware
  - which properties are important

The Basics of the Combining Technique

- A thread attempts to become a combiner and serve in addition to its own request, active requests by other threads
- After announcing their requests , other threads may:
  - either perform the same actions (although not always "successfully")
    - synchronization by employing CAS or other similar primitives
  - or perform local spinning until the combiner performs their requests
    - synchronization by employing a coarse-grain lock

Blocking Combining: CC-Synch



Properties of CC-Synch

Fairness

- CC-Synch provides a strong notion of fairness: it serves requests in the order they enter the announced list.

Progress

- In CC-Synch, no thread ever starves.

Performance

- The lock is implemented using a highly-efficient queue-like lock.

- CC-Synch is the first that provides bounds on the number of RMRs it executes:
  - A combiner thread performs O(h+t) RMRs, where h is an upper bound on the number of requests that a combiner may serve, and t is the size of the shared data that should be accessed in order to execute the h requests.
  - Other threads perform just a constant number of RMRs.
  - The amortized number of RMRs performed is O(d), where d is the average number of RMRs required to serve a single request.

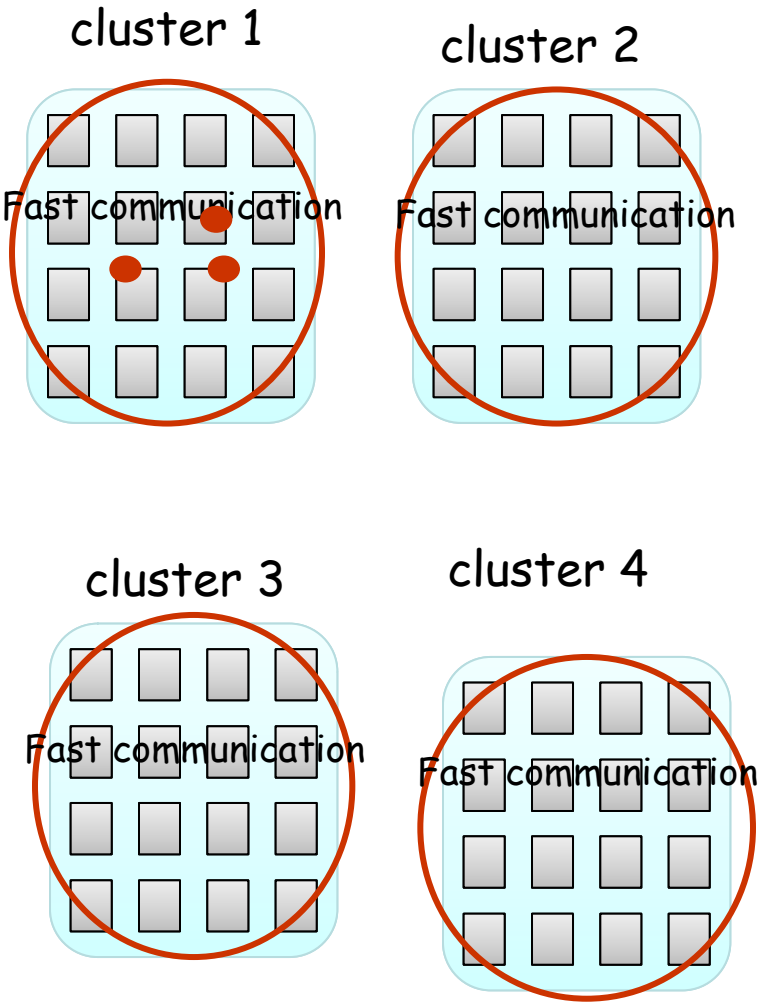
Required Primitives

- Only a swap object is used (no CAS) and r/w registers.

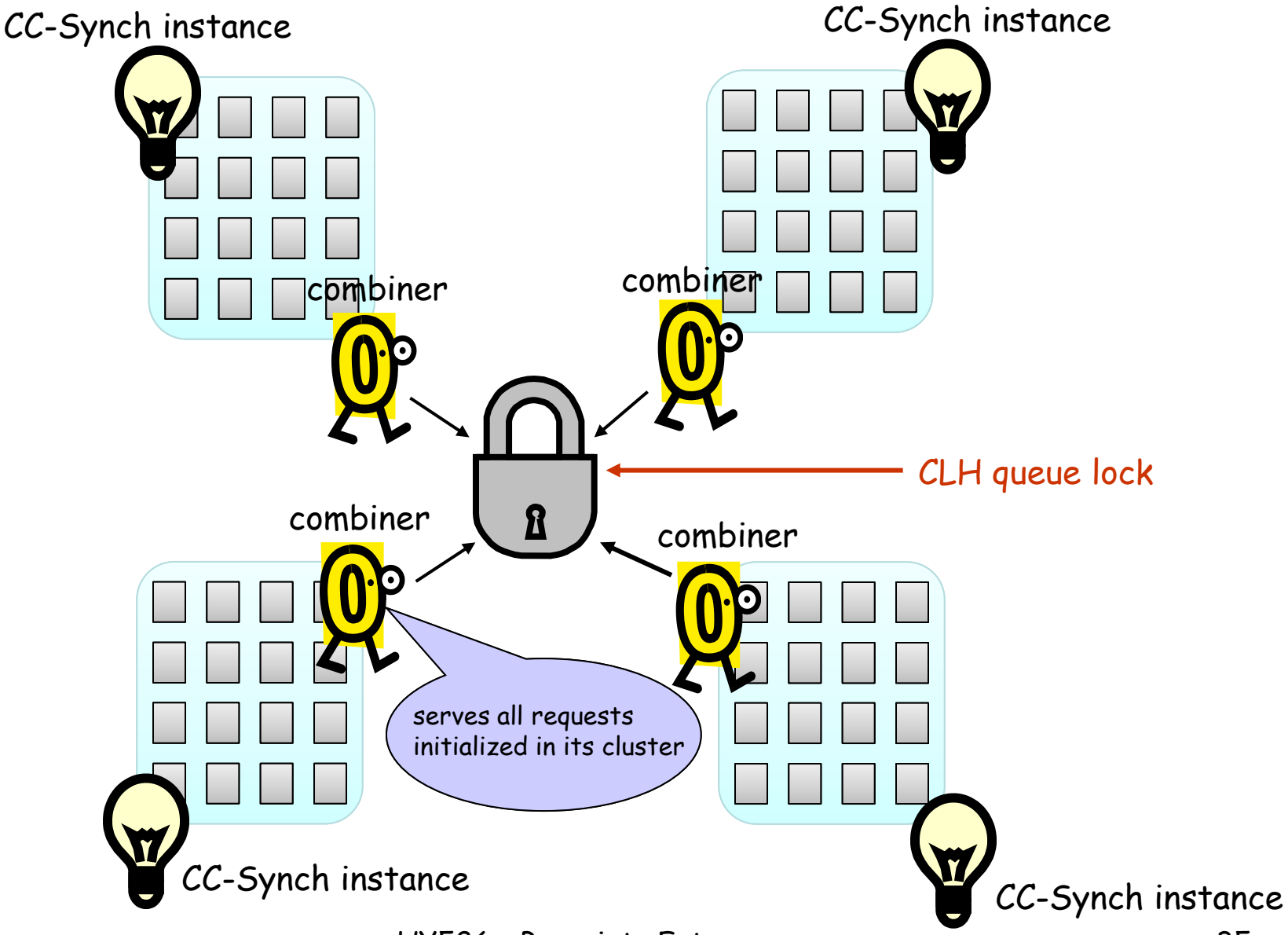
Hierarchically Structured systems
The H-Synch Algorithm

- **H-Synch** is an hierarchical version of CC-Synch

⇒ Exploits hierarchical communication nature to achieve better performance



H-Synch – More Details



Experimental Analysis

Machines on which the experiments were performed

- 32-core machine consisting of 4 AMD Opteron 6134 processors (Magny Cours)
- 128-way Sun consisting of 2 UltraSPARC-T2 processors (Niagara 2)

Synchronization approaches that were compared

- CC-synch and DSM-Synch
- P-Sim (Fatourou and Kallimanis, SPAA 2011)
- Flat-combining
- CLH-spin locks
- OyamaAlg
- Simple lock-free implementation
- H-Synch and Hierarchical NUMA lock (Marathe et al., SPAA 2011) in Niagara 2 machines

Experiment

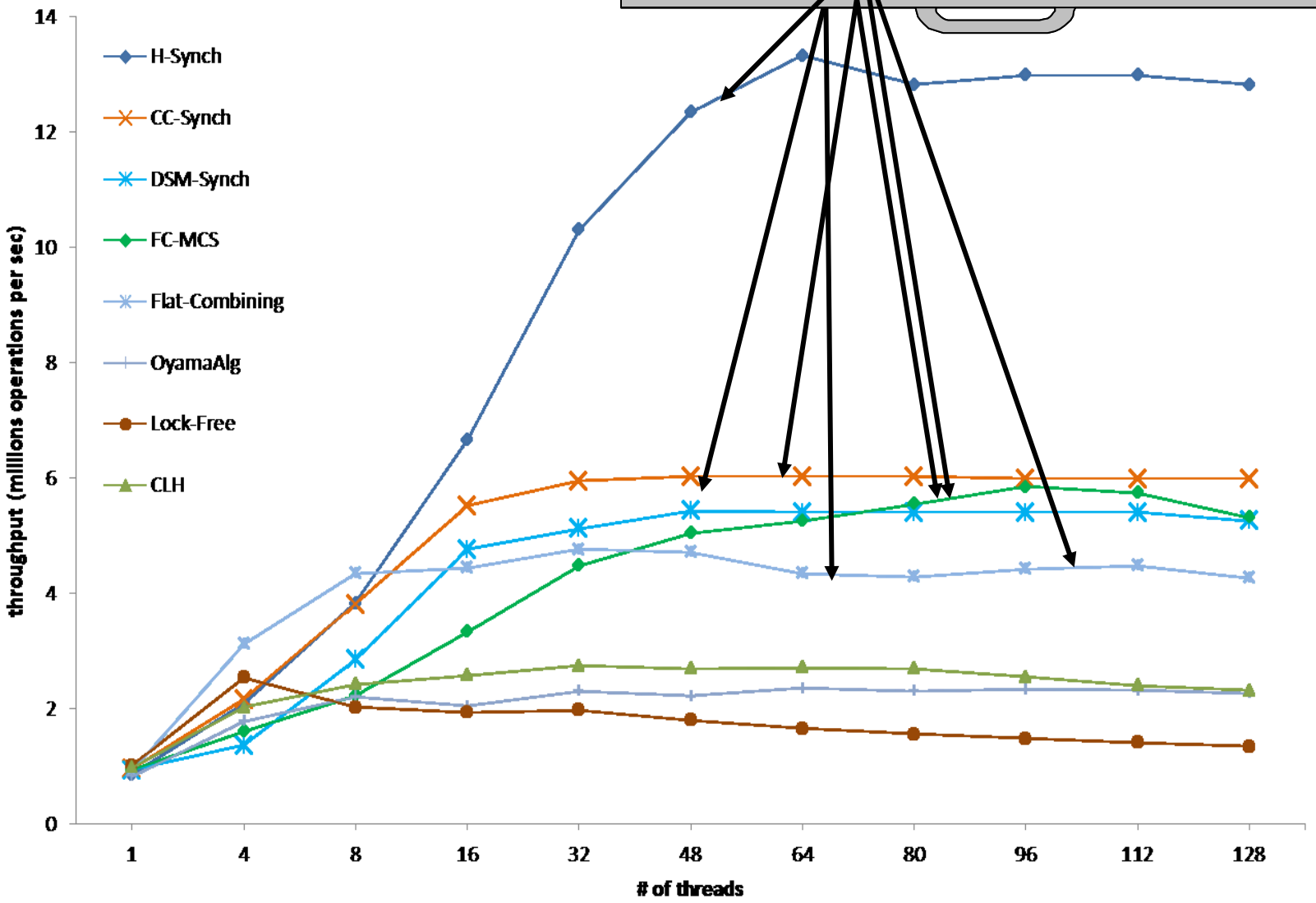
A simple Fetch&Multiply object is simulated:

We measure the average throughput (number of Fetch&Multiply executed per second) that each approach exhibits when it executes 10^7 Fetch&Multiply requests for different values of n.

A random number of dummy loop iterations are executed between the execution of two consecutive Fetch&Multoply by the same thread.

Experimental Analysis Niagara 2

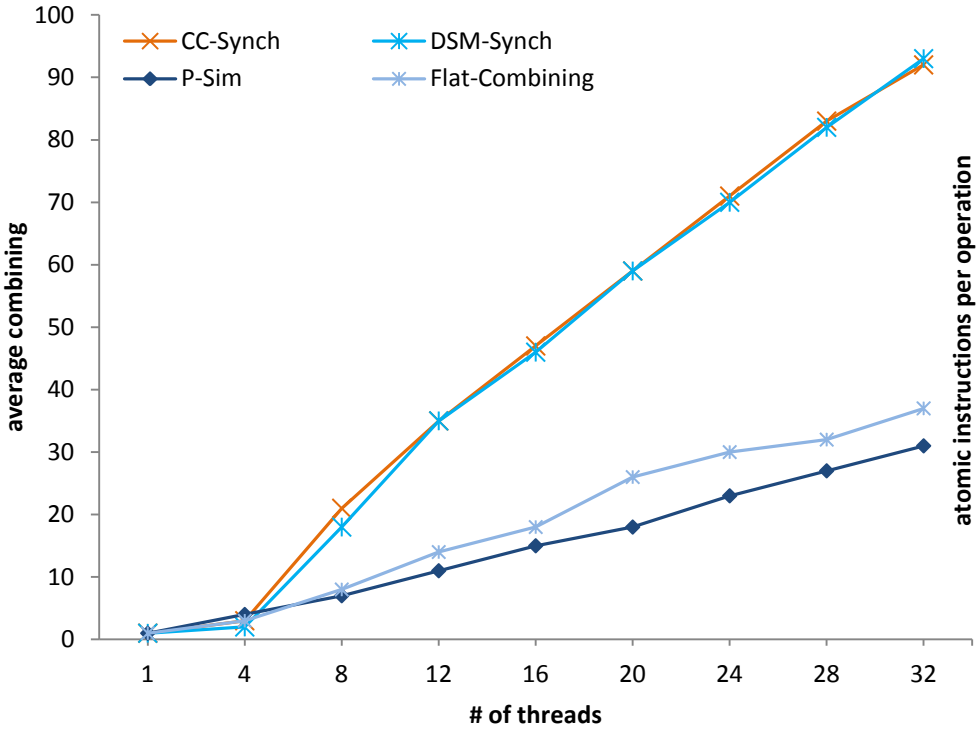
H-Synch is 2.65 times faster than the hierarchical NUMA lock and 3 times faster than flat-combining

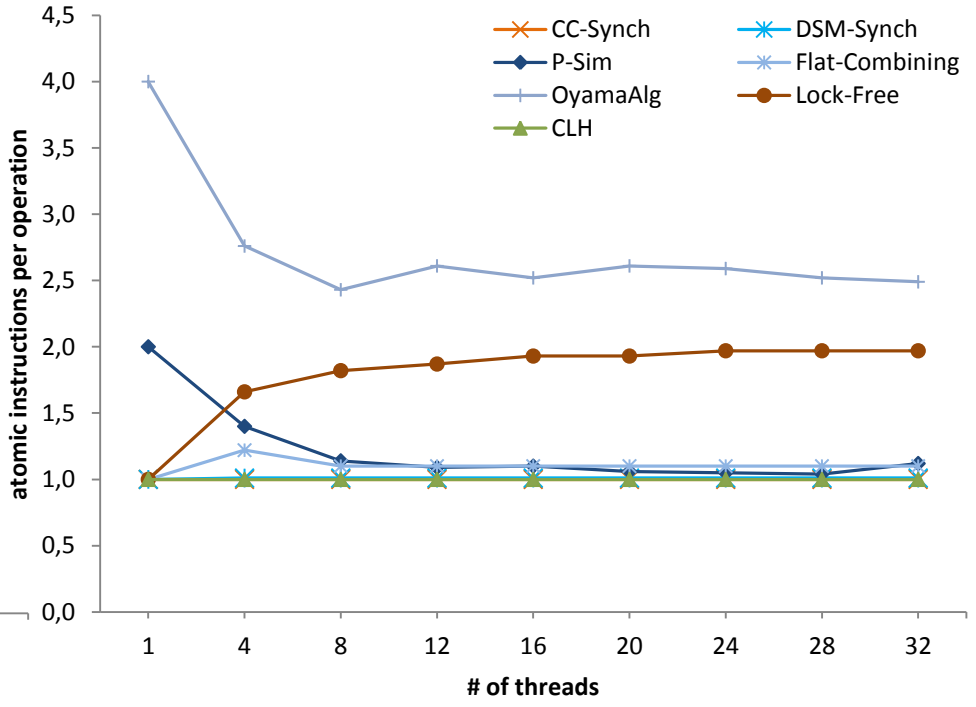


Cache Misses – Memory Stalls

Algorithm	cache misses	cpu cycles spent in memory stalls
CC-Synch	4.1	2747
Flat-combining	5.8	6501

Why is CC-Synch Efficient?

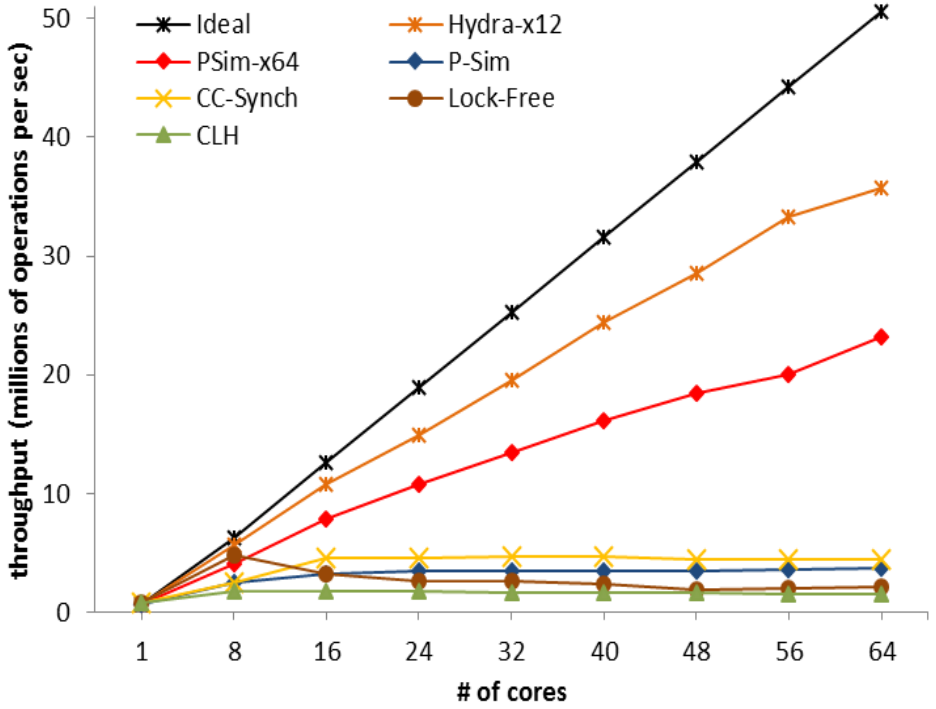




Factors that significantly impact performance

- Combining degree
- Number of primitives performed

Scheduling-Aware Synchronization



HYDRA

- Blocking combining implementation
  - employs user-level threads
  - schedules them appropriately

PSimX

- Simple (wait-free) variant of PSim employing user-level threads

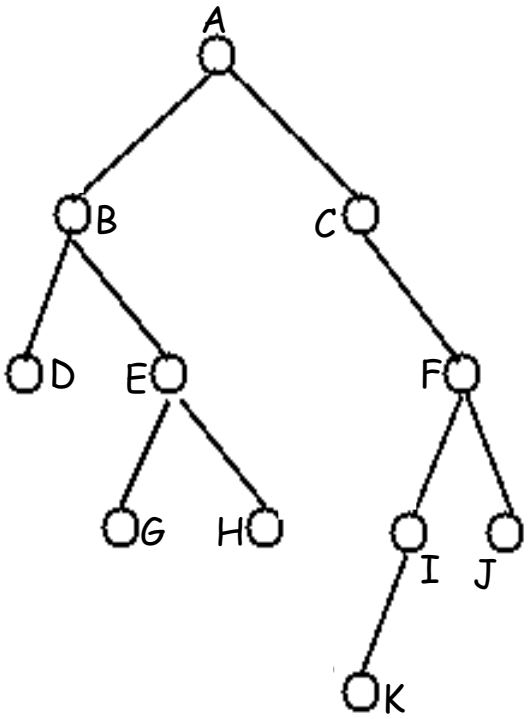
Performance Advantages

- The throughput of HYDRA is better than that of CC-Synch by a multiplicative factor of up to 7.9
- The throughput of PSimX is better than that of PSim by a multiplicative factor of up to 5.6.
- The throughput of HYDRA and PSimX are very close to the ideal.

PART III: Limitations of the Combining Technique

- What is the cost to apply k concurrent search operations on a tree?

- A "sophisticated" concurrent implementation could allow multiple searches proceeding in parallel and being executed in O(h) time in total, where h is the height of the tree.



Bibliography

These slides are based on material that appears in the following books and papers:

- M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, Morgan Kauffman, 2008 (Chapter 7)
- Panagiota Fatourou, Nikolaos D. Kallimanis: Revisiting the combining synchronization technique. PPOPP 2012: 257-266
- Panagiota Fatourou, Nikolaos D. Kallimanis: A highly-efficient wait-free universal construction. SPAA 2011: 325-334

End of Section

Financing

- The present educational material has been developed as part of the educational work of the instructor.
- The project "Open Academic Courses of the University of Crete" has only financed the reform of the educational material.
- The project is implemented under the operational program "Education and Lifelong Learning" and funded by the European Union (European Social Fund) and National Resources

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ

ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Notes

Licensing Note

- The current material is available under the Creative Commons Attribution-NonCommercial-NoDerivs 4.0[1] International license or later International Edition. The individual works of third parties are excluded, e.g. photographs, diagrams etc. They are contained therein and covered under their conditions of use in the section «Use of Third Parties Work Note».

[1] http://creativecommons.org/licenses/by-nc-nd/4.0/

- As Non-Commercial is defined the use that:
  - Does not involve direct or indirect financial benefits from the use of the work for the distributor of the work and the license holder
  - Does not include financial transaction as a condition for the use or access to the work
  - Does not confer to the distributor and license holder of the work indirect financial benefit (e.g. advertisements) from the viewing of the work on website

- The copyright holder may give to the license holder a separate license to use the work for commercial use, if requested.

Reference Note

Preservation Notices

Any reproduction or adaptation of the material should include:

- the Reference Note
- the Licensing Note
- the declaration of Notices Preservation
- the Use of Third Parties Work Note (if is available)

together with the accompanied URLs.