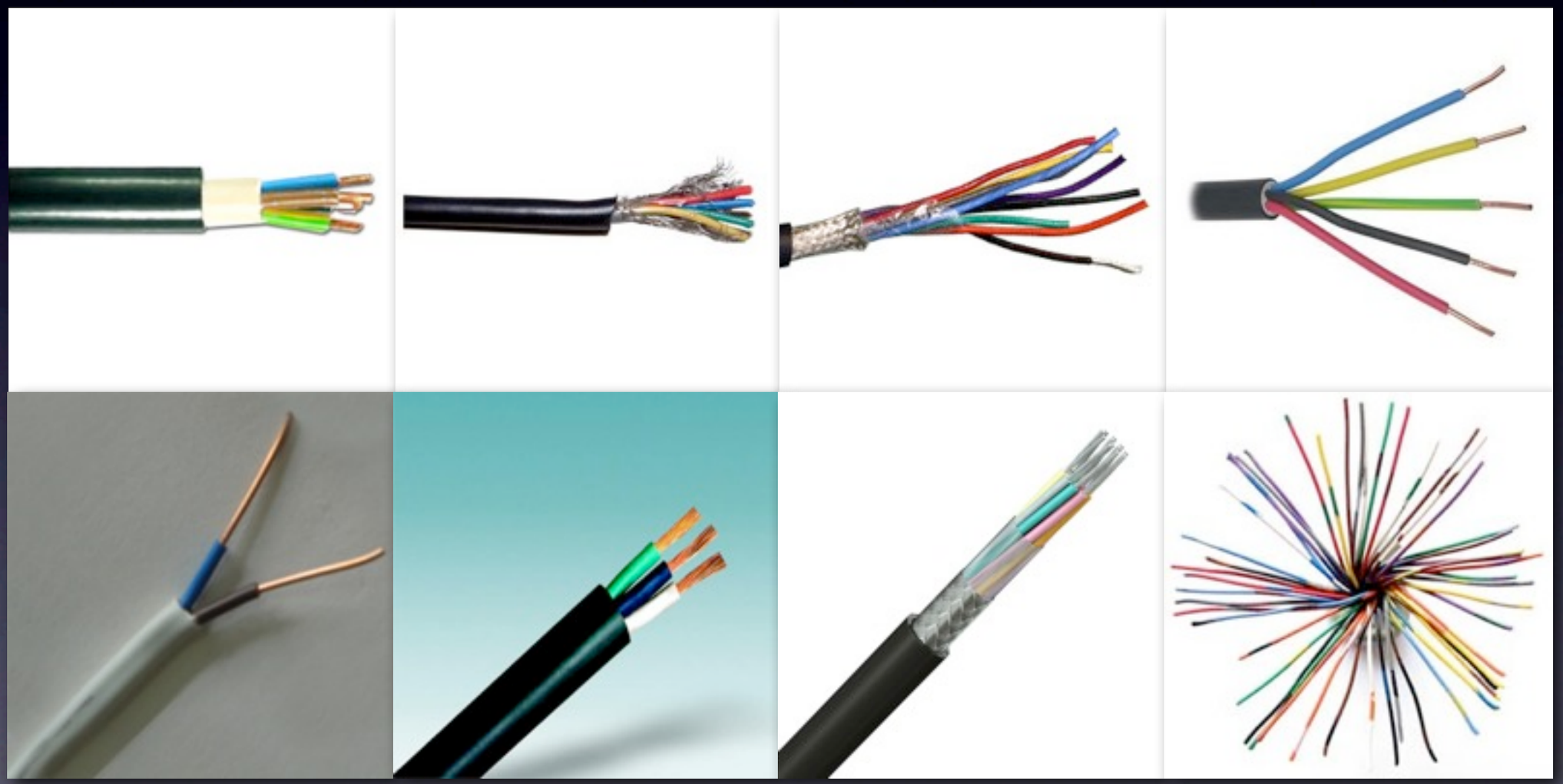
System Software in the Many-Core Era

Wolfgang Schröder-Preikschat

ESI Embedded Systems Institute

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

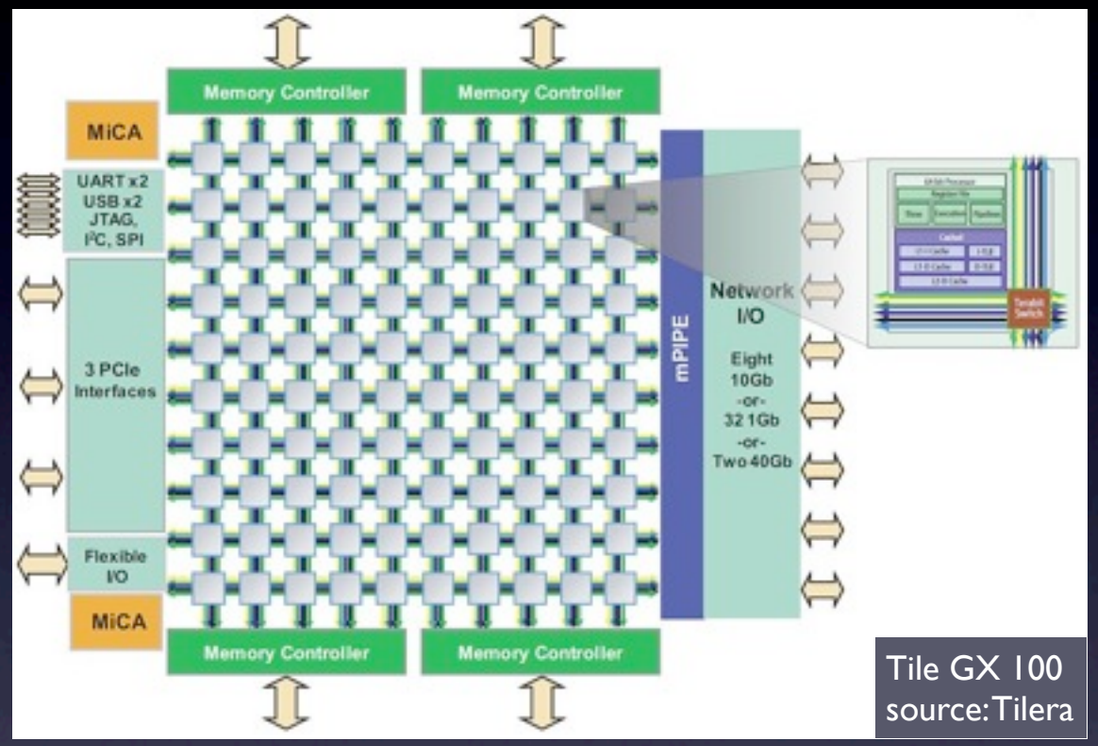
Multi-Core



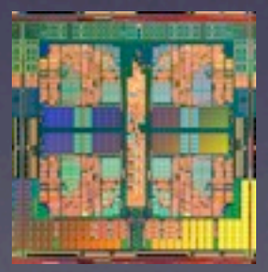
Many-Core

http://www.eschlimanphoto.com

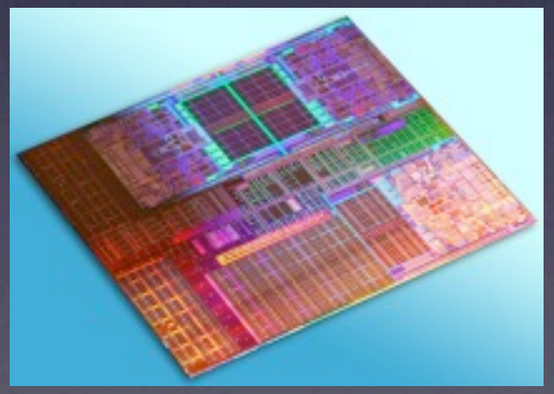
Parallel Systems in Little



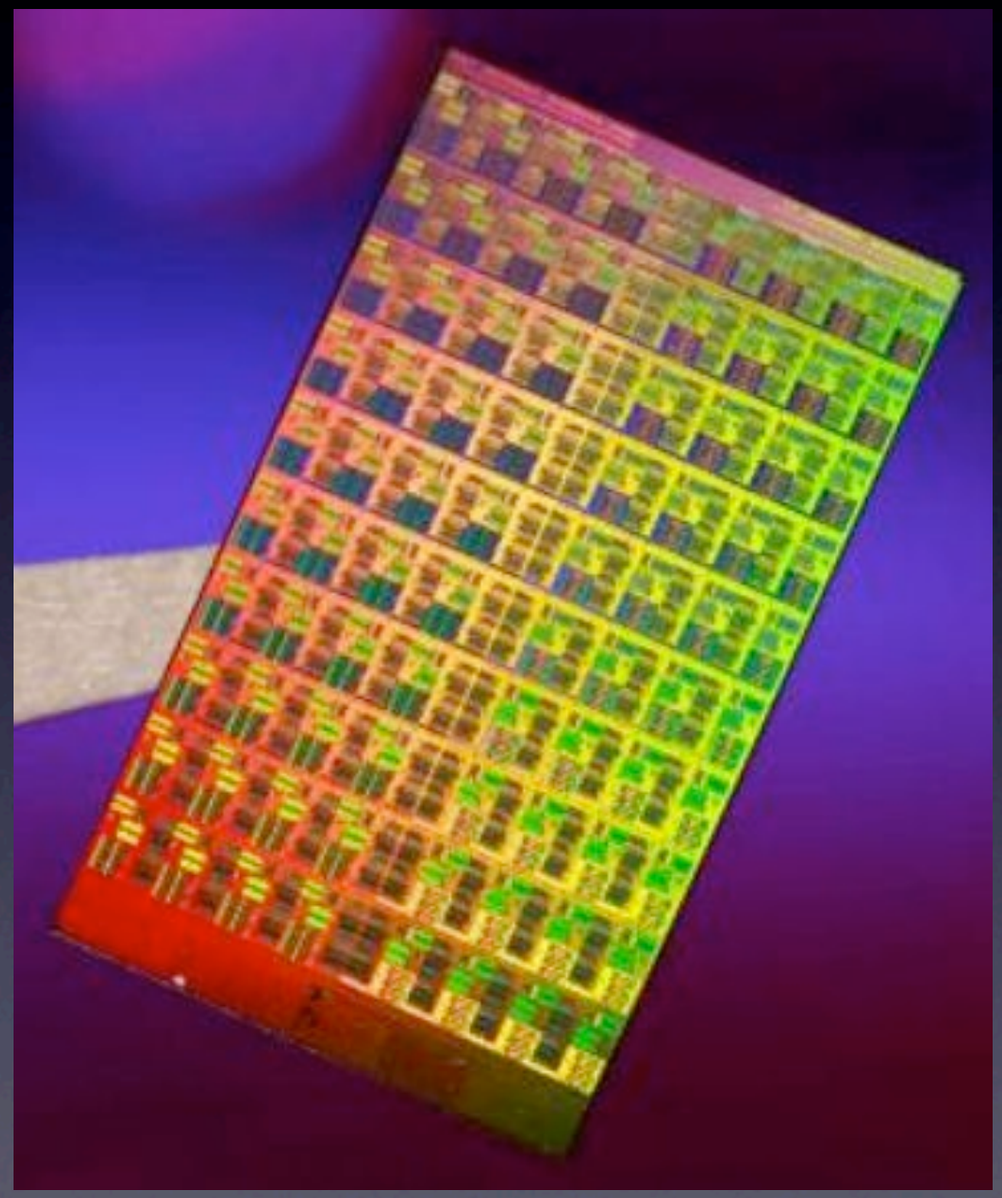
2



4

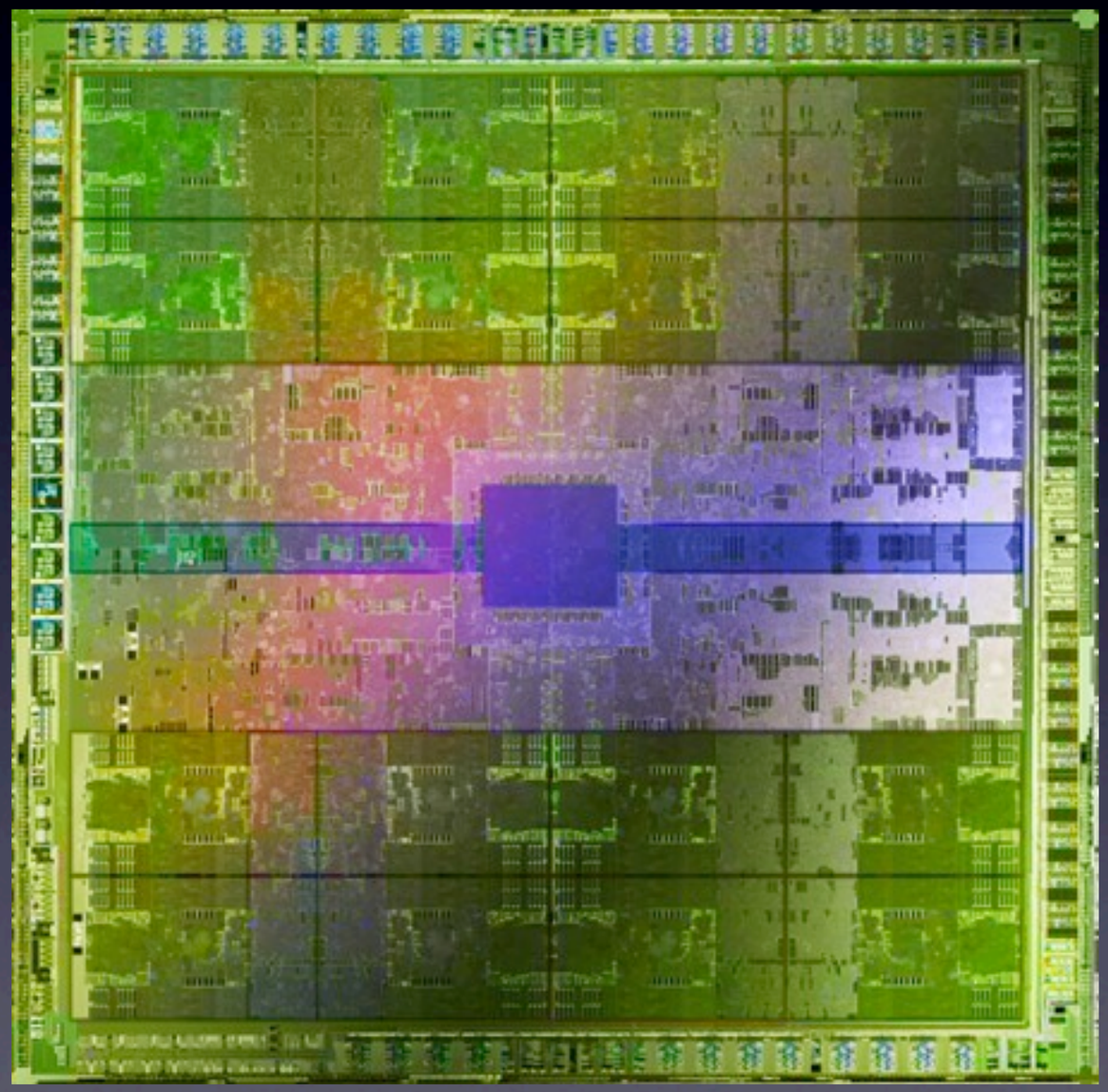


8



80

Parallel Systems in Little



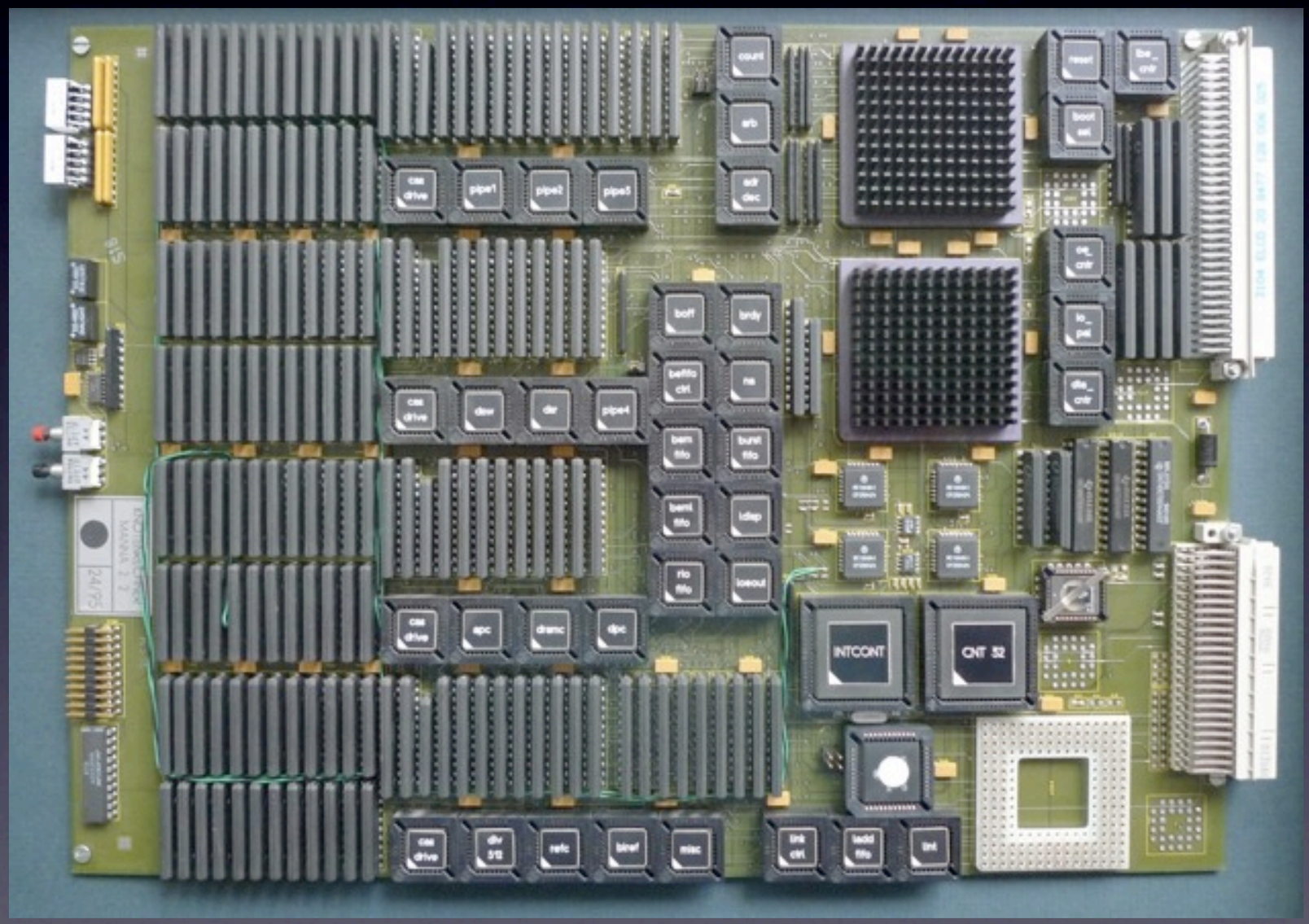
512

The end of the flagpole is still out of sight!

Déjà vu

SUPRENUM

dualprocessor

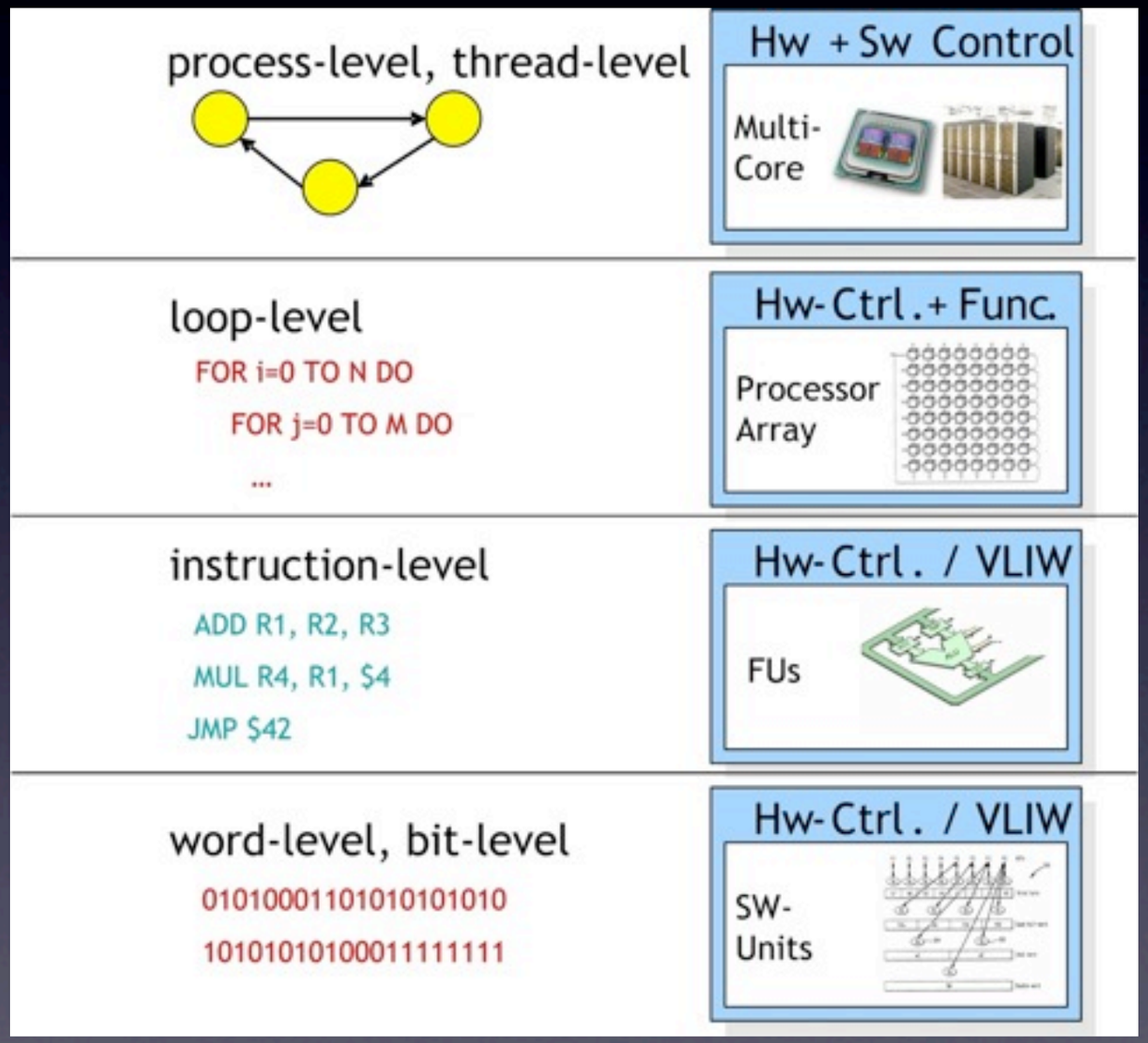


once upon a time in the last millenium...

dualcore

today

Levels of Parallelism



granularity of parallelism

extent of parallelism

Research Challenges: Operating Systems

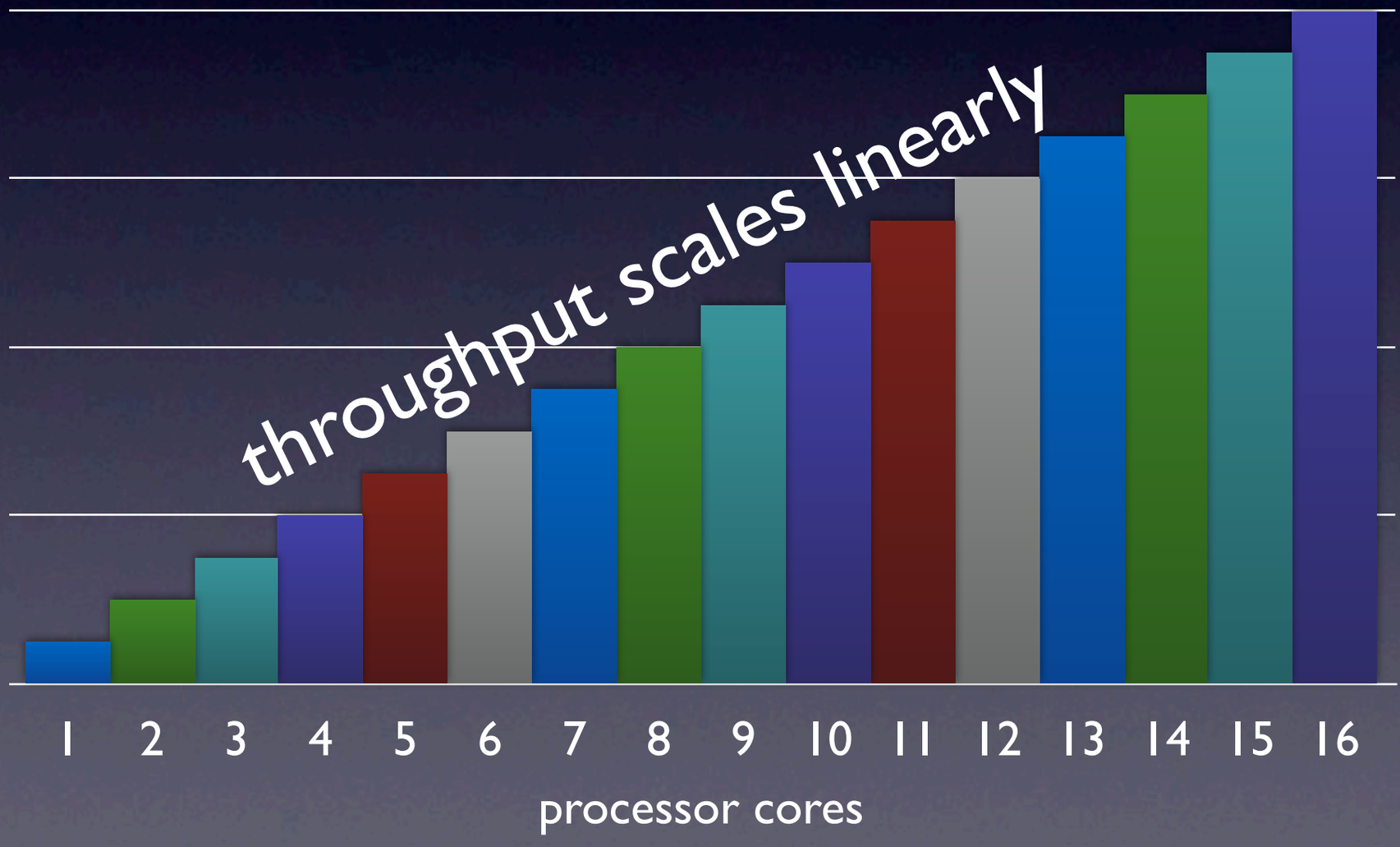
- making system-level data structures policed shareable by user-level threads
- more optimistic – and less pessimistic – concurrency control
- latency hiding through relocation of system activities to ``idle´´ cores
- re-engineering of legacy operating systems for time-sharing or real-time mode

Case Study

- file descriptor table
  - typical kernel-level shared data structure
- test run: costs in case of concurrent access
  - `dup`/`close` of thread-local descriptors
  - exactly one `dup`/`close`-thread per core
  - 16-core AMD Opteron, Linux 2.6.27

Theory

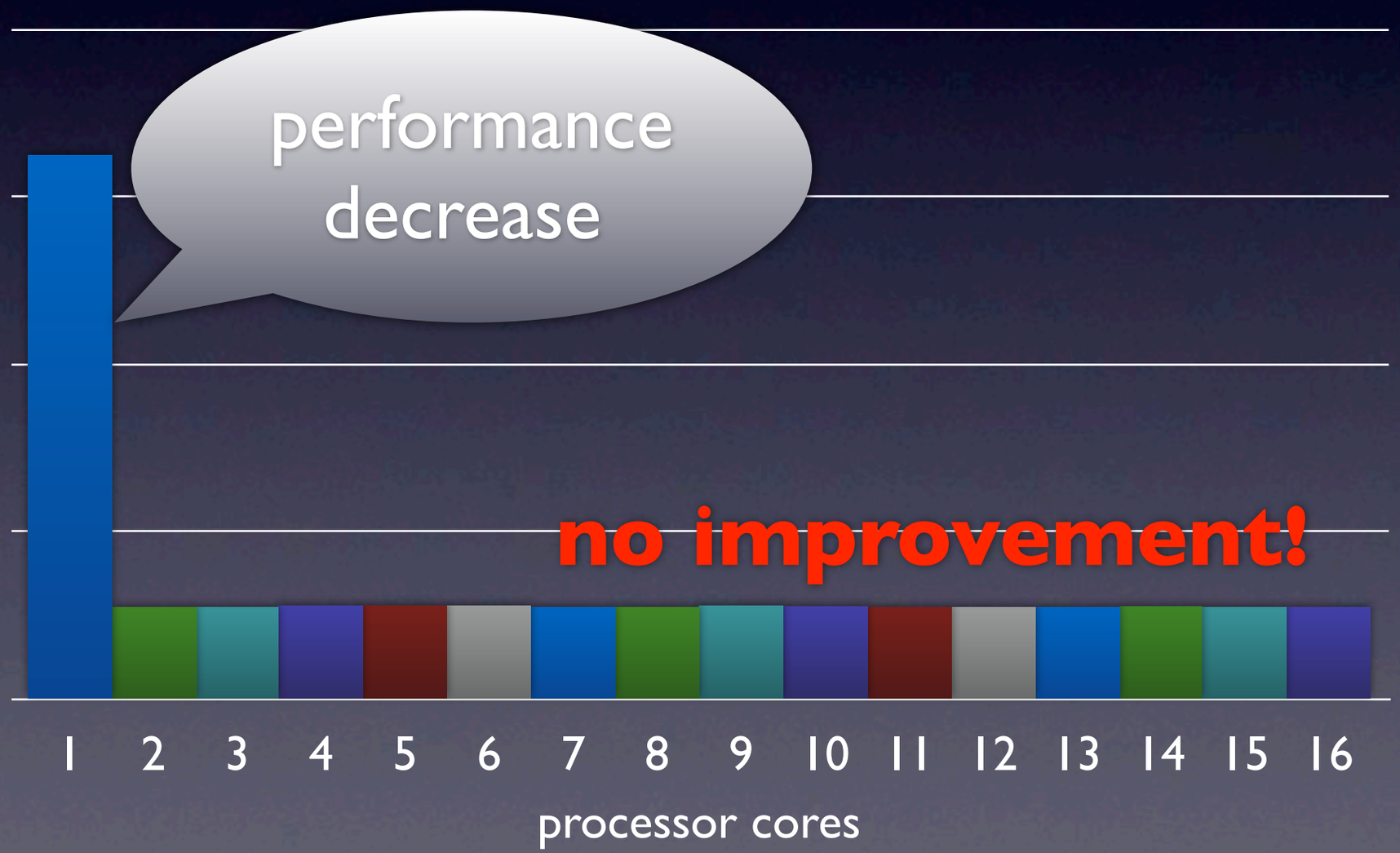
file descriptor table: # dup/close per second



Boyd-Wickizer et al., OSDI'08

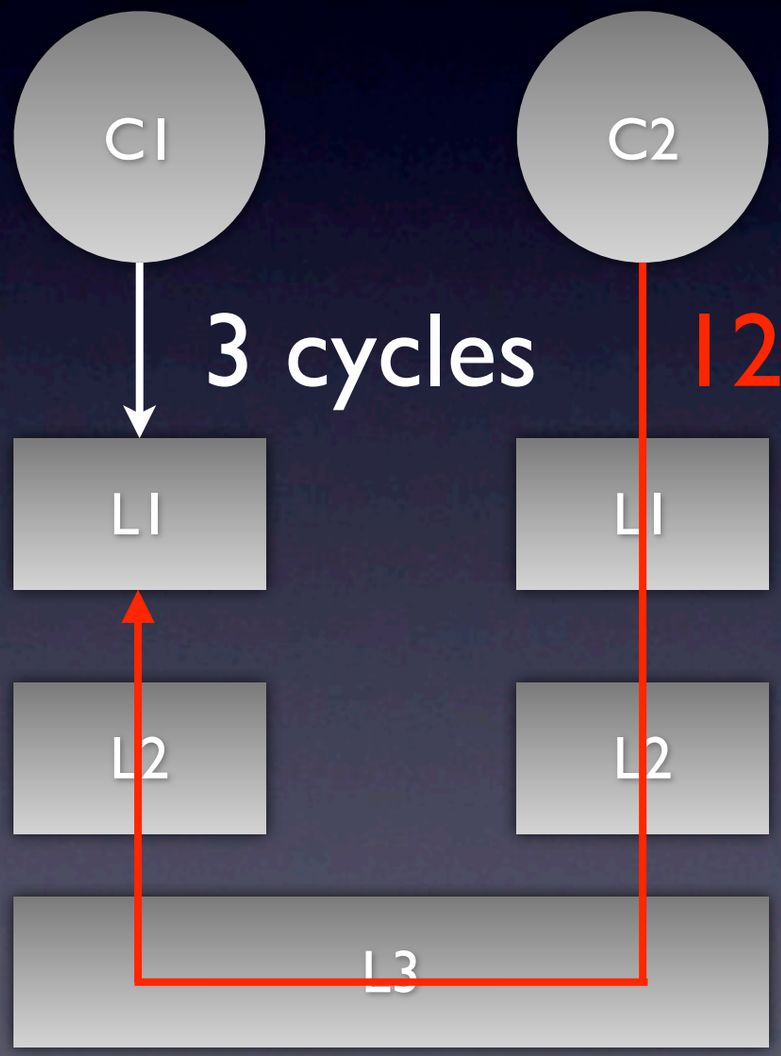
Practice

file descriptor table: # dup/close per second



Boyd-Wickizer et al., OSDI'08

Access Patterns




```
fd_alloc () {
    lock(fd_table);
    fd = get_free_fd();
    set_fd_used(fd);
    fix_smallest_fd();
    unlock(fd_table);
}
```

- cause of performance decrease

Competition

Blocking synchronization puts operations on `fd_table` in sequence although concurrent threads will access different table entries!

```
fd_alloc () {
    lock(fd_table);
    fd = get_free_fd();
    set_fd_used(fd);
    fix_smallest_fd();
    unlock(fd_table);
}
```

false sharing ▸ cause of no improvement

Boyd-Wickizer et al., OSDI'08

Scalability: Linux

- 2.4 scales barely above 4 cores
- 2.6 uses 91% of its time within spin locks
  - benchmarks in the EXT4 context

    mosts scale well up to 6-7 out of 8 cores

    many scale well up to 12 out of 16 cores

    an acceptable number scales well up to 32 cores
- forthcoming 32-core CPUs are a problem...

Recommendations (1)

- multi-core paket processing
- elimination of false sharing
- sloppy counters
- core-local data structures
- lock-free comparisons
- prevention/avoidance of unneeded blocking

Boyd-Wickizer et al., OSDI'10

Recommendations (2)

- atomic variables
- read/write locks
- fine-grain locking
- processing by bundle
  - combining (in software) of operations or incarnations of critical sections

Synchronization

Synchronization

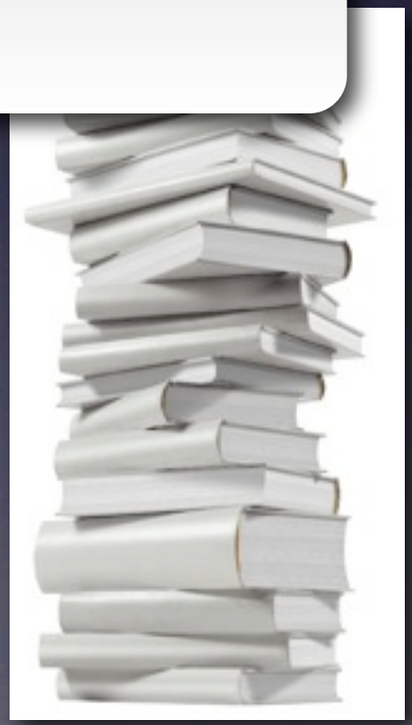
- well-known techniques dominate
  - lock variable, semaphore, monitor
- fairly carefree use of blocking operations
  - critical section ≠ indivisible section
  - CPU ≠ indivisible resource
- functional vs. non-functional properties

Example: LIFO List

```
void dos_push (chain_t *this, chain_t *item) {
   item->link = this->link;
   this->link = item;
}
```

```
typedef struct chain {
   struct chain *link;
} chain_t;
```

```
chain_t *dos_pull (chain_t *this) {
   chain_t *node = this->link;
   if (node != 0)
      this->link = node->link;
   return node;
}
```



Race-intolerant LIFO List

```
void bs_push (CHAIN_t *this, chain_t *item) {
   enter(&this->lock);
   dos_push(&this->list, item);
   leave(&this->lock);
}
```

```
chain_t *bs_pull (CHAIN_t *this) {
   chain_t *node;
   enter(&this->lock);
   node = dos_pull(&this->list);
   leave(&this->lock);
   return node;
}
```

bs: *blocking synchronization*

Race-tolerant LIFO List

```
void nbs_push (chain_t *this, chain_t *item) {
   do item->link = this->link;
   while (!CAS(&this->link, item->link, item));
}
```

```
inline int CAS (word_t *ref, word_t exp, word_t val) {
   unsigned int aux;

   __asm__ __volatile__ (
      "lock\n\t"
      "cmpxchgl %2,%1\n\t"
      "sete %0"
      : "=q" (aux), "=m" (*ref)
      : "r" (val), "m" (*ref), "a" (exp)
      : "memory");

   return aux;
}
```

```
chain_t *nbs
   chain_t *
   do if ((n                                    reak;
   while (!CAS(&this->link, node, node->link));
   return node;
}
```

nbs: *non-blocking synchronization*

Race-tolerant FIFO List

```
void dos_aback (queue_t *this, chain_t *item) {
   item->link = 0;
   this->tail->link = item;
   this->tail = item;
}
```

```
typedef struct queue {
   chain_t  head;
   chain_t *tail;
} queue_t;
```

```
chain_t *dos_fetch (queue_t *this) {
   chain_t *node;

   if ((node = this->head.link)
   && !(this->head.link = node->link))
      this->tail = &this->head;

   return node;
}
```

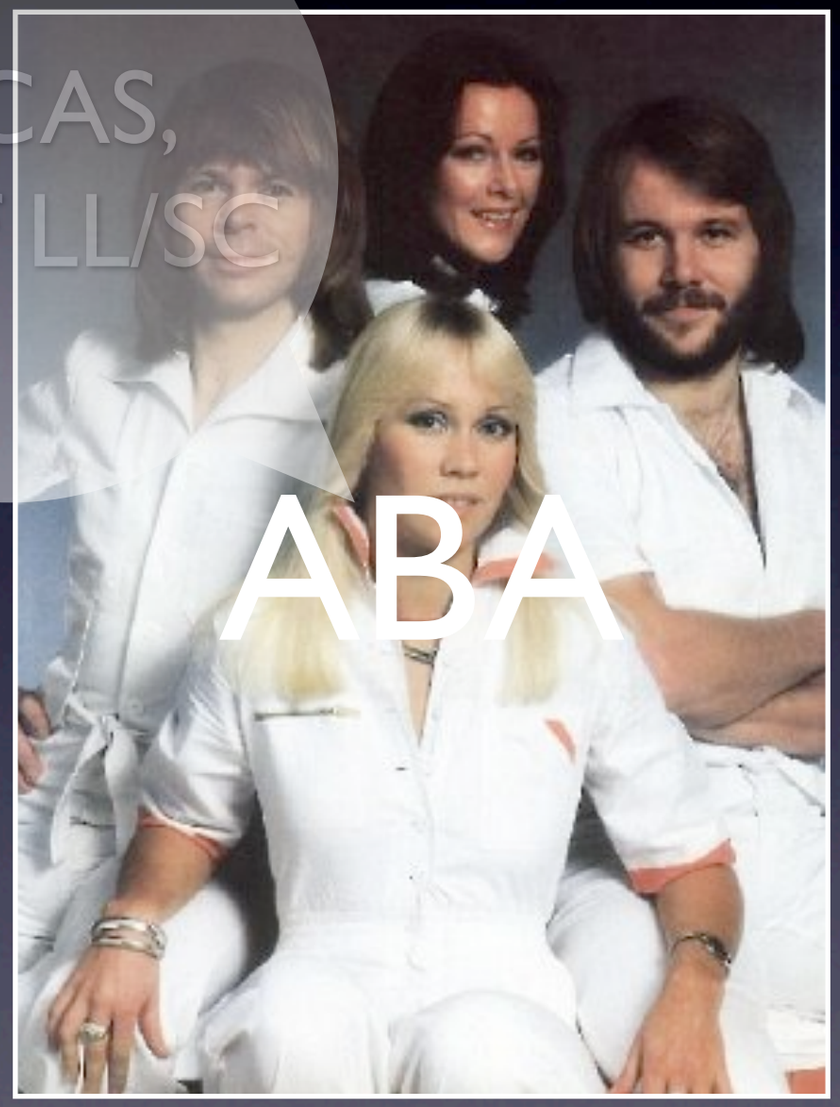
```
void nbs_aback (queue_t *this, chain_t *item) {
   chain_t *last, *self;

   item->link = link;

   do self = (last = this->tail)->link;
   while (!CAS(&this->tail, last, &item->link));

   if (!CAS(&last->link, self, item))
      this->head.link = item;
}
```

```
chain_t *nbs_fetch (queue_t *this) {
   chain_t *node, *next;

   do if ((node = this->head.link) == 0) return 0;
   while (!CAS(&this->head.link, node,
      ((next = node->link) == node ? 0 : next)));

   if (next == node) {
      if (!CAS(&node->link, next, 0))
         this->head.link = node->link;
      else CAS(&this->tail, &node->link, &this->head);
   }

   return node;
}
```

Problem

1. X reads A from FOO
2. Y preempts X
3. Y reads **A** from FOO
4. Y writes **B** to FOO
5. Y writes **A** to FOO
6. X preempts Y
7. X sees FOO unchanged: A

of CAS,
not of LL/SC



Workaround: Generation Counter

a) tagging of critical variables

- tag field added to pointer: ``unused bits´

- taking advantage of alignment

b) safeguard through ``protection variable´´

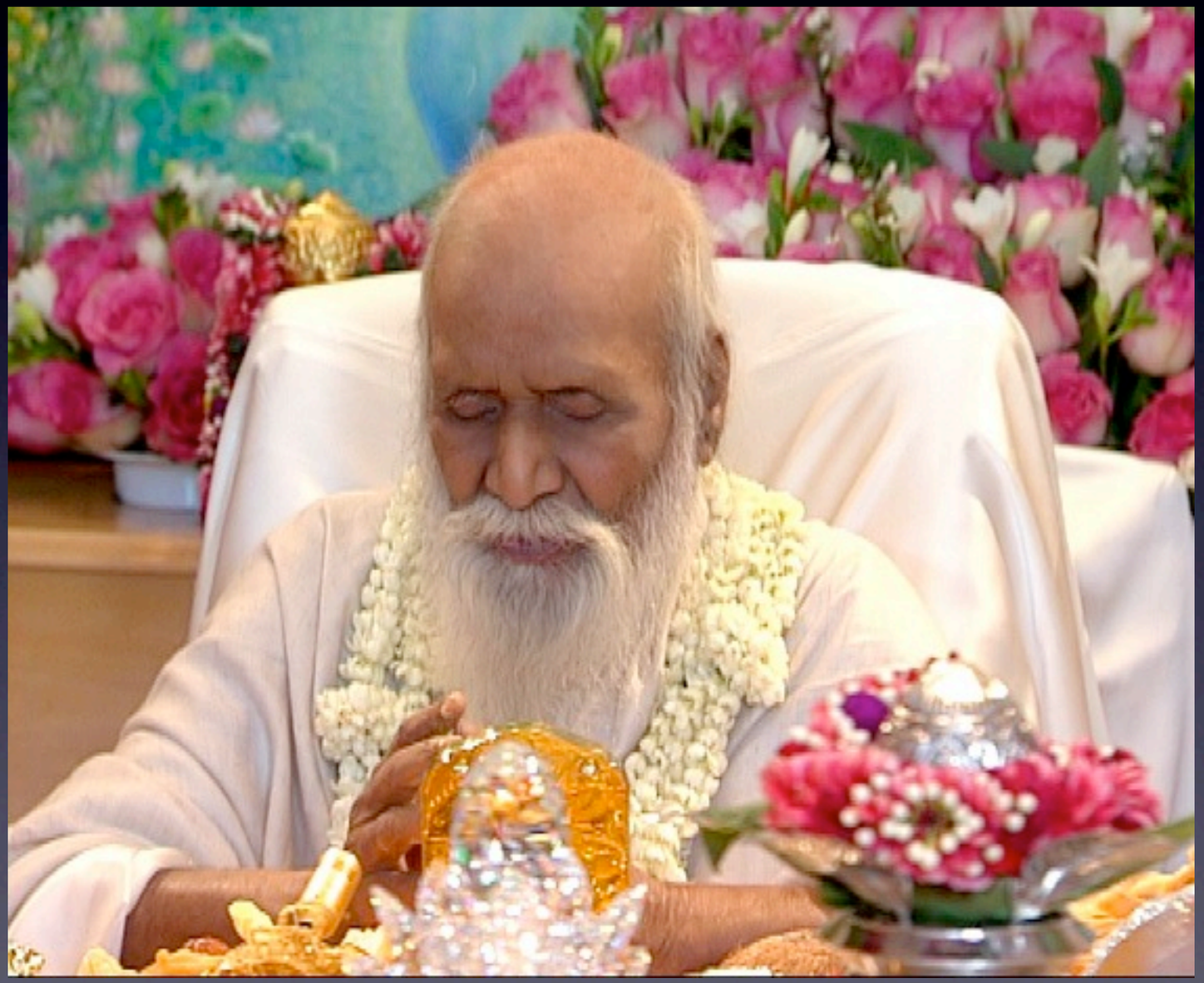
- extended atomic instructions

- DCAS, CMPXCHG8

Disillusion

- microscopic level: ABA remains unsolved
  - tag/counter neither prevents nor avoids
  - rather makes ABA unlikely to appear
- macroscopic level: ABA prevented/avoided
  - use case, context information, scheduling
  - identification of features: *domain scoping*

TM

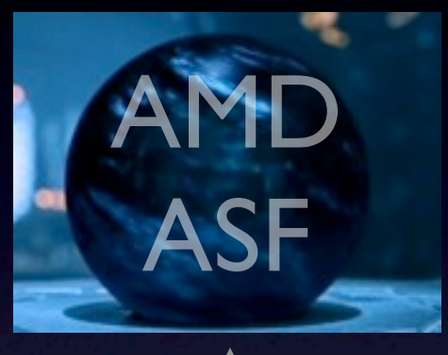


Transactional Memory

- Hardware (1986)
  - Sun Rock (ASPLOS'09) ?
- Software (1995)
  - using conventional atomic primitives
- Hybrid (2005)
  - HTM, as long as resources are on hand
  - STM, else...

HTM

STM



(2009)

TM...

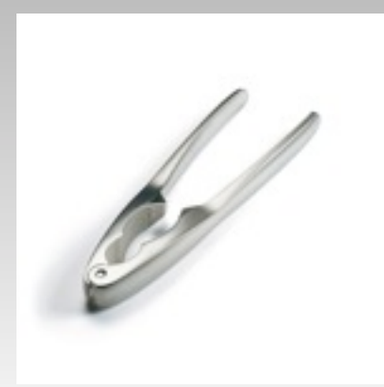
```
void stm_push (chain_t *this, chain_t *item) {
   transactional {
      item->link = this->link;
      this->link = item;
   }
}
```

```
chain_t *stm_pull (chain_t *this) {
   chain_t *node;
   transactional {
      node = this->link;
      if (node != 0)
         this->link = node->link;
   }
   return node;
}
```

...considered harmful!

```
void ewd_prolaag (semaphore_t *this) {
   transactional {
      if (this->load-- <= 0)
         sad_sleep(&this->line);
   }
}
```

```
void ewd_verhoog (semaphore_t *this) {
   transactional {
      if (this->load++ < 0)
         sad_awake(&this->line);
   }
}
```



ewd: *Edsgar Wybe Dijkstra*

Pros and Cons of TM

- suited for ``simple'' critical sections
- composable, deadlock-free: just as CAS or LL/SC — but different level of abstraction!
- use within operating systems is not easier compared to CAS or LL/SC
- requires hardware support — and claims a considerable lot of hardware resources

Last but not least: Progress Guarantees

- obstruction-free
  - non-competitive: progress guaranteed
- lock-free: a must for system software
  - system progress guaranteed
- wait-free: a must for *real-time* system software
  - progress of every thread guaranteed

Re-engineering of legacy software is non-trivial...

Shallows...

FreeBSD

lock_mtx
mtx_lock
mtx_lock_flags
_get_sleep_lock
mtx_lock_sleep
rnstile_wait
x_lock_spin
ock_spin_flags
ock_spin_flags
_get_spin_lock
_mtx_lock_spin
_obtain_lock
atomic_cmpset_acp_ptr
atomic_cmpset_acp_long
atomic_cmpset_long
cmpxchgq

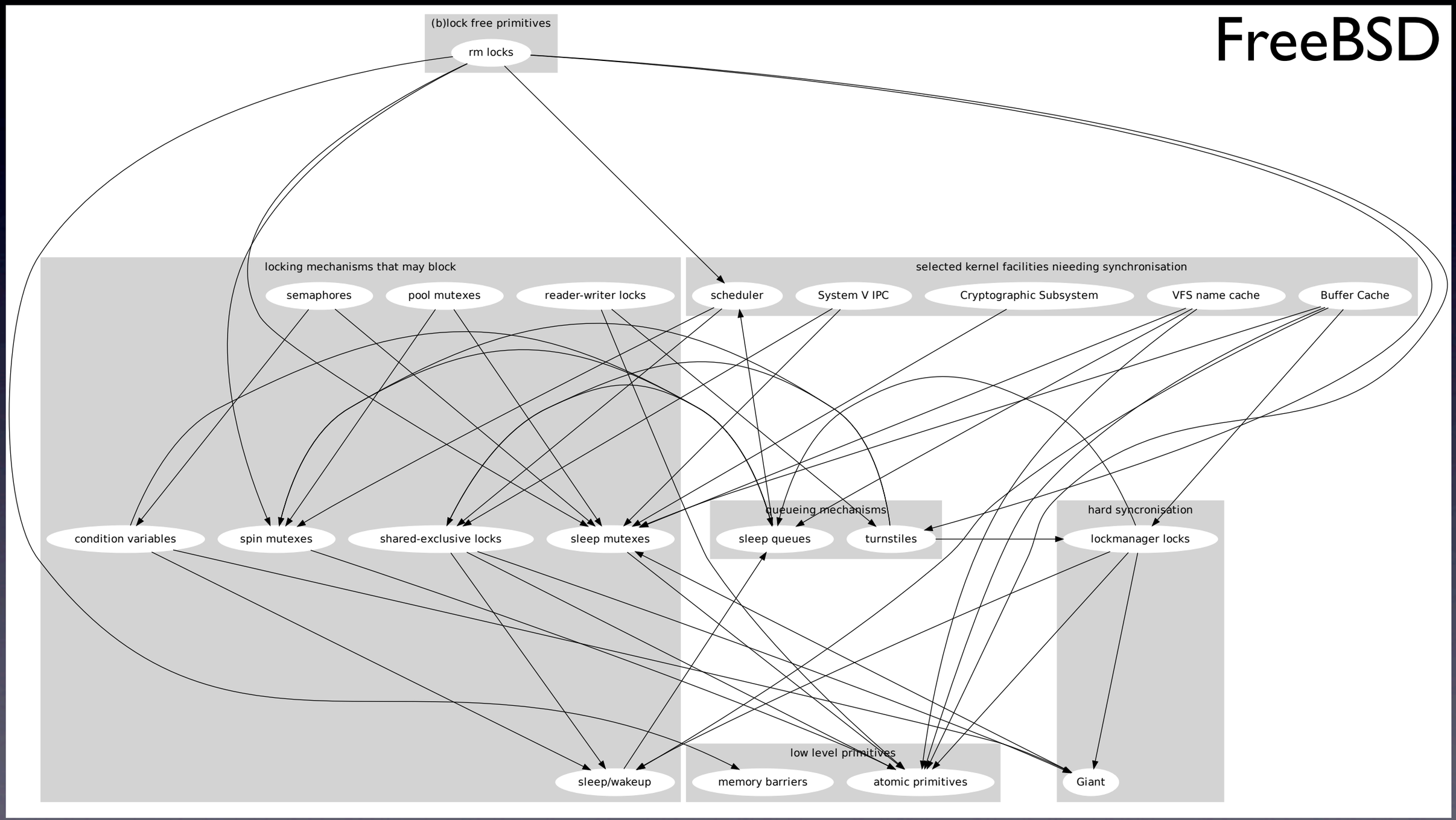
thread_lock
thread_lock_flags
_thread_lock_flags
spinlock_enter
intr_disable

cpu_spinwait
turnstile_try

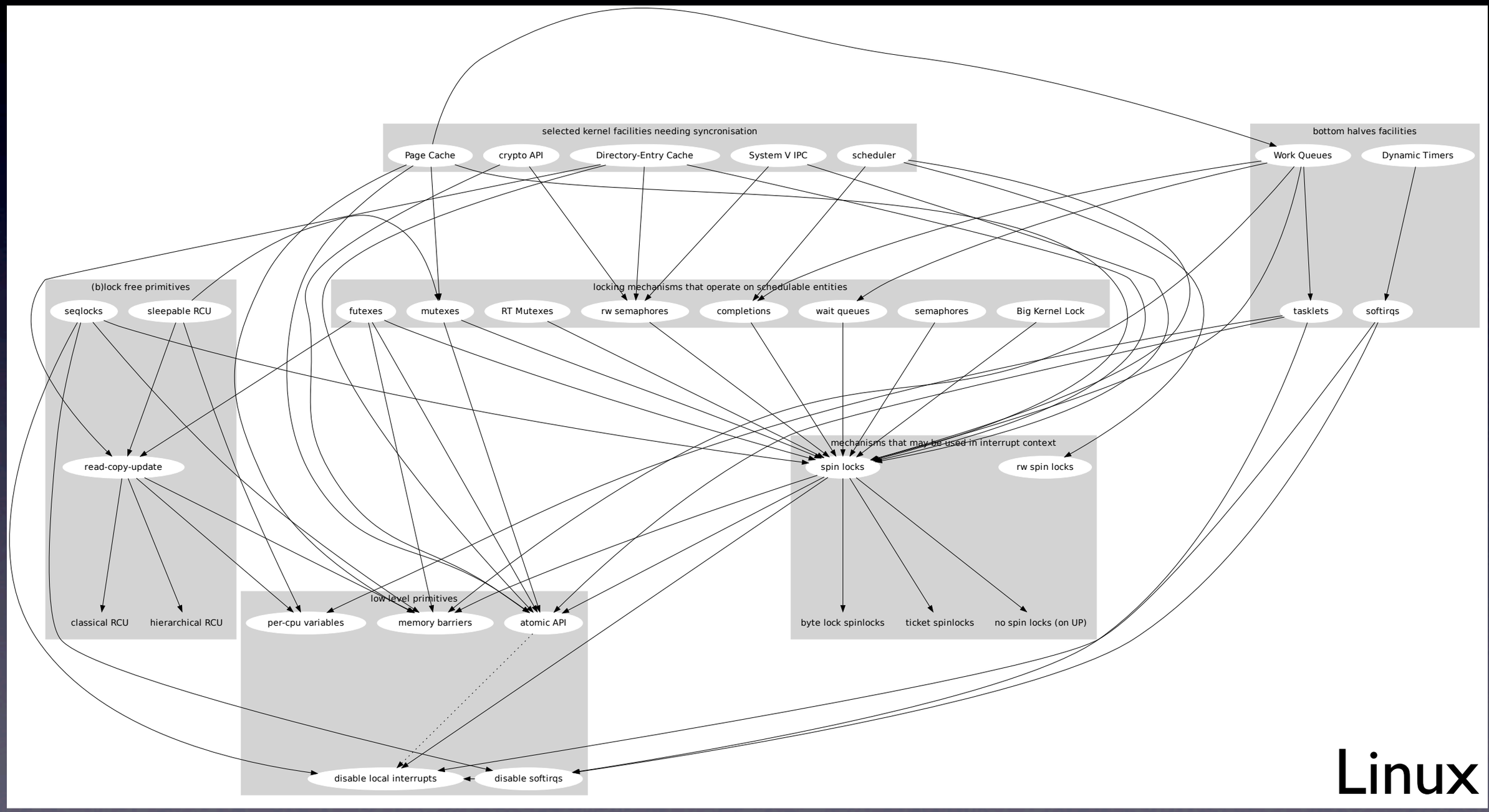
```
void enable_mmiotrace (void) {
   mutex_lock(&mmiotrace_mutex);
   if (is_enabled())
      goto out;
   if (nommiotrace)
      pr_info("MMIO trace disabled\n");
   kmmio_init();
   enter_uniprocessor();
   spin_lock_irq(&trace_lock);
   atomic_inc(&mmiotrace_enabled);
   spin_unlock_irq(&trace_lock);
   pr_info("enabled.\n");
out:
   mutex_unlock(&mmiotrace_mutex);
}
```

Linux

Shallows...


FreeBSD
(b)lock free primitives
rm locks
locking mechanisms that may block
semaphores
pool mutexes
reader-writer locks
selected kernel facilities nieeding synchronisation
scheduler
System V IPC
Cryptographic Subsystem
VFS name cache
Buffer Cache
queueing mechanisms
sleep queues
turnstiles
hard syncronisation
lockmanager locks
condition variables
spin mutexes
shared-exclusive locks
sleep mutexes
low level primitives
sleep/wakeup
memory barriers
atomic primitives
Giant

Shallows...


selected kernel facilities needing syncronisation
Page Cache
crypto API
Directory-Entry Cache
System V IPC
scheduler
bottom halves facilities
Work Queues
Dynamic Timers
(b)lock free primitives
seqlocks
sleepable RCU
locking mechanisms that operate on schedulable entities
futexes
mutexes
RT Mutexes
rw semaphores
completions
wait queues
semaphores
Big Kernel Lock
tasklets
softirqs
mechanisms that may be used in interrupt context
read-copy-update
spin locks
rw spin locks
classical RCU
hierarchical RCU
low level primitives
per-cpu variables
memory barriers
atomic API
byte lock spinlocks
ticket spinlocks
no spin locks (on UP)
disable local interrupts
disable softirqs
Linux

Latency Hiding

- well-known technique in HPC
  - start-up time minimization (IPC, I/O)
  - asymmetric multiprocessor system
- largely unused in real-time systems
  - deterministic processes
- processor cores for operating-system use

Application of Cores

many-core processor	
universal cores	*load balancing*
special cores	*latency hiding (system-inherent activities)*
changing core	*system call handling as a whole*
fixed core	*system call handling in pieces*
driver core	*off-line I/O*
device core	*peripheral control*
interrupt core	*interrupt handling*
service core	*off-line service*
scheduling core	*task delivery*
I/O core	*file operation*
communication core	*protocol stack processing*
paging core	*external pager (replacement policy)*

All that glitters is not gold

Caches!

Summary

- revisit Hoare: The Emperor's Old Clothes
- prevent contention by construction
- prefer optimistic concurrency control
- do not hype TM (for system software)
- aim for latency hiding by means of cores
- last but not least: beware of the caches...