Symmetric Multiprocessing

Main issues:

- locking,
- cache coherence,
- scheduling.

Good discussion of issues in [Sch94].

Kernel Locking

- Several CPUs can be executing kernel code concurrently.
  ⇒ Need mutual exclusion on shared kernel data.
- Issues:
  – Lock implementation
  – Granularity of locking

Mutual Exclusion Techniques

- Disabling interrupts (`CLI` — `STI`).
  ➜ Unsuitable for multiprocessor systems.
- Spin locks.
  ➜ Busy-waiting wastes cycles.
- Lock objects.
  ➜ Flag indicates object is locked.
  ➜ Manipulating lock requires mutual exclusion.

Spin locks

```
void lock (volatile lock_t *l) {
   while (test_and_set(l)) ;
}

void unlock (volatile lock_t *l) {
   *l = 0;
}
```

Busy waits. Good idea?

Spin lock busy-waits until lock is released:

- Stupid on uniprocessors, as nothing will change while spinning.
  ➜ Should release (yield) CPU immediately.
- Maybe ok on SMPs: locker may execute on other CPU.
  ➜ Minimal overhead.
  ➜ Still, should only spin for short time.

Generally restrict spin locking to:

➜ *short* critical sections,
➜ unlikely to be contended by the same CPU.
➜ local contention can be prevented
  ➜ by design
  ➜ by turning off interrupts

Alternative: Conditional lock

```
bool cond_lock (volatile lock_t *l) {
   if (test_and_set(l))
      return FALSE; //couldn't lock
   else
      return TRUE;  //acquired lock
}
```

➜ Can do useful work if fail to aquire lock.
➜ **But** may not have much else to do.
➜ Starvation: May never get lock!

More appropriate mutex primitive:

```
void mutex_lock (volatile lock_t *l) {
   while (1) {
      for (int i=0; i<MUTEX_N; i++)
         if (!test_and_set(l))
            return;
      yield();
   }
}
```

- Spins for limited time only
  ➜ assumes enough for other CPU to exit critical section
- Useful if critical section is shorther than N iterations.
- Starvation possible.

Multiprocessor spin lock:

```
void mp_spinlock (volatile lock_t *l) {
   cli();    // prevent local contention
   while (test_and_set(l)) ;   // lock
}

void mp_unlock (volatile lock_t *l) {
   *l = 0;
   sti();
}
```

➜ only good for short critical sections

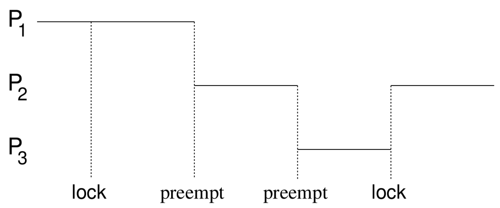
MULTIREADER LOCKS:

```
void rw_rdlock (volatile lock_ *l);
void rw_wrlock (volatile lock_ *l);
```

➜ Allow mutliple readers into the critical section concurrently.
➜ Write access is exclusive.
➜ Too much overhead for really short critical sections.
➜ Used in UNIX SysVR4.

Dangers of Locking: Priority Inversion

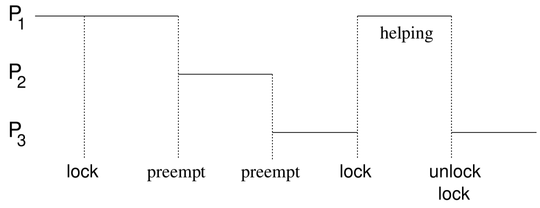
- Assume prio(P_1) < prio(P_2) < prio(P_3), all running on same CPU.



- P_2 prevents higher-priority process P_3 from executing.
- **Solution:** Avoid preempting processes holding a kernel lock.
  ➜ How?

SOLUTION: PRIORITY INHERITANCE

- Blocked high-prio process *helps* locker by *donating* time slices.



- Also called *wait-free locking* [CSL+87].
  ➜ Everything needs to be prioritised.
  ➜ Need to record holder of lock.
  ➜ No good if P_1 holds lock too long.

WAIT-FREE SYNCH. OF LONG CRITICAL SECTIONS:

- Multiprocessor priority-inheritance protocol [HH01]
  – cross-CPU helping: B holds lock, A helps B
  – **remote helping:** A migrates to B's CPU
    ➜ only works if A becomes highest-prio on B's CPU
    ➜ need global "end-to-end" prio scheme
    ➜ otherwise not wait-free
    ➜ race condition: A migrates to B, B migrates away...
  – **local helping:** A execute's B's code on own CPU
    ➜ B's state must migrate to A's CPU
    ➜ totally wait-free: highest-prio always makes progress

ALTERNATIVE: LOCK-FREE SYNCHRONISATION:

- Ensure all data is always consistent
- Perform changes on shadow copies
- When completed, perform atomic swap
  ➜ eg swap pointers with atomic *compare-and-swap* instruction
  ➜ use `mp_spinlock` if no such instruction
- practically limited to simple data structures (linked lists)

Best to avoid long critical sections in kernel!

WHAT TO LOCK?

Giant lock: lock whole kernel.

➜ Only one process can execute in kernel.
➜ Similar to dedicated OS processor.

Coarse-grain locks: lock whole subsystems.

➜ E.g., all TCBs, file system.
➜ Marginal improvement over giant lock, **not scalable**.

Fine-grain locks: lock as little as possible at a time.

➜ Potential for large amount of parallelism.
➜ Only suitable approach for large numbers of CPUs.

ALL BUT GIANT LOCKS CAN LEAD TO DEADLOCKS!

➜ Usual deadlock-avoidance schemes apply (numbering locks).
➜ May not always know in advance which locks are needed.
  - Release lock temporarily to obtain lower-numbered one.
  - Must leave DS consistent when releasing.
  - Must recheck state after reacquiring.

Locking: Performance Considerations

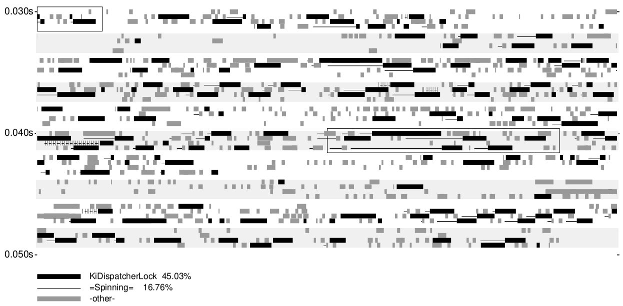
- Small lock granularity
  ➜ decreases lock contention,
  ➜ increases potential parallelism.
  ➜ Also increased scope for stuffing up.
- Even with careful design hard to avoid bottlenecks ("convoys").
  ➜ Important to measure lock contention.
  ➜ Instrument lock ops to keep stats.

Illustrative Example

Windows-NT Kernel dispatcher lock [PS96].

- DEC people investigated performance problems of Microsoft's SQL server running on Alphas under NT.
- No access to source code.
- Used tool to patch executable code (OS and apps).
- Instrumented code logged change of control flow to memory buffer.
- Reconstructed instruction trace from log.
- Visualised results.

Result for 4-CPU System



Summary of Results:

- Second box shows convoy effect on KiDispatcherLock.
  - ➜ Lock held for 200–900 cycles.
    - ➜ Partially due to interrupts being enabled during critical section.
    - ➜ Disk interrupt serviced while holding lock.
  - ➜ Lock held for about 45 % of total time.
  - ➜ 16 % of time spent spinning.
- This one lock limits OS scalability to about 6 CPUs!
- Shows the necessity of keeping critical sections **short**.

Effects of Memory Architecture

Example: end of a critical section

```
/* counter++; */
load   r1, counter
add    r1, r1, 1
store  r1, counter
/* unlock(mutex); */
store  zero, mutex
```

Relies on all CPUs seeing update of `counter` before update of `mutex`.

➜ Depends on proper *ordering* of stores to memory.

Memory Models: Strong Ordering

- Loads and stores executed *in program order*.
- Memory accesses of different CPUs are sequentialised.
- Traditionally used by many architectures.

CPU 0	*CPU 1*
`store r1, adr1`	`store r1, adr2`
`load r2, adr2`	`load r2, adr1`

- At least one CPU must load the other's new value.

Other Memory Models

Modern hardware features can interfere with store order:

- write buffer (or *store buffer* or *write-behind buffer*),
- instruction reordering,
- superscalar execution,
- pipelining.

Each CPU keeps its own data consistent, but how about others?

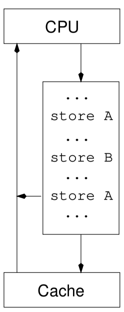
➜ SMP?
➜ DMA?

Total Store Ordering

- Stores to *write buffer* hide memory latency.
- Loads read from write buffer if possible.
- Stores are guaranteed to occur in *FIFO order*.

CPU 0	*CPU 1*
`store r1, adr1`	`store r1, adr2`
`load r2, adr2`	`load r2, adr1`

➜ Both CPUs may read old values!



Total store ordering breaks Decker:

```
void lock (volatile lock_t *l) {
  l->status[MYSELF] = LOCKED;
  while (l->status[OTHER] == LOCKED) {
    if (l->turn != MYSELF) {
      l->status[MYSELF] = !LOCKED;
      while (l->turn == OTHER) ;
      l->status[MYSELF] = LOCKED;
} } }
```

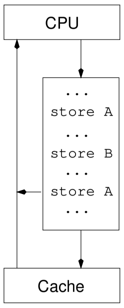
- Need hardware support for synchronisation, e.g.:
  - ➜ atomic swap,
  - ➜ test&set,
  - ➜ load-linked & store-conditional (LL&SC),
  - ➜ memory barriers.
- Stall pipeline and drain (& bypass) write buffer.

Partial store ordering

- All stores go through write buffer.
- Loads read from write buffer if possible.
- Redundant stores are cancelled.
  ➜ Breaks FIFO-order of stores!

```
load      r1, counter  // counter++;
add       r1, r2, 1
store     r2, counter
 barrier
store     zero, mutex  // unlock(mutex);
```

- Store to `mutex` can overtake store to `counter`.
- Need to use *memory barrier*.
- Failure to do so will introduce subtle bugs:
  ➜ Changing process state after saving context.
  ➜ Initiating I/O after setting up parameter buffer.



Cache Consistency

- Caching can lead to a processor in an SMP system reading stale data.
- Can even happen when reading *different* data:
  ➜ Different data may lie in same cache line!
- Need to ensure caches are coherent:
  ➜ by software, or
  ➜ by hardware (standard these days).

⇒ Need cache coherency protocols.

Hardware cache coherency

- Ensure consistency of all caches and RAM.

Write-invalidate protocols: Ensure that:
  ➜ only a single cached copy of the data exist at the time of a store,
  ➜ dirty lines will propagate to memory prior to being read into any other cache.

Write-update protocols: Update all cached copies at the time of a store.

- **Note:** Similar (software) protocols are used in distributed systems.

WRITE-THROUGH INVALIDATE PROTOCOL

Two versions:

① All stores write through the cache.
  ➜ RAM is always consistent with cache.
  ➜ No dirty cache lines ever.

② Cache snoops bus for write cycles and invalidates any copies.

Normal bus arbitration resolves race conditions.

➜ Can cache spin locks, Decker works...
➜ Cannot use write-back caching.
➜ Need bus cycle for each store ⇒ limited scalability.

WRITE-ONCE PROTOCOL

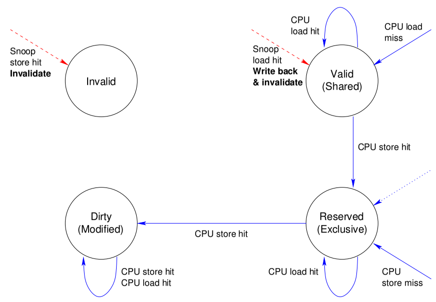
Works with write-back caches:

- First store to clean line writes through cache.
- Store to uncached line allocates in cache.
- Further stores to same line only write to cache.
- Cache snoops bus for write cycles and invalidates any copies.

Normal bus arbitration resolves race conditions.

➜ Introduces new state for a cache line: *reserved*.

WRITE-ONCE STATE DIAGRAM:



➜ **Note:** Store miss can occur from any state, not only invalid:
➜ The line may have held different valid or dirty data.

MESI PROTOCOL

Named after initials of states: **M**odified-**E**xclusive-**S**hared-**I**nvalid.

- Like write-once, except that a load miss on a line which is *not in any cache* goes directly to the exclusive state.
- Snoop load hits require cache to assert it has the line.

Used in many modern SMP architectures.

WRITE-INVALIDATE PROTOCOLS:

- Based on the assumption that shared data is likely to remain shared.
- Basic protocol similar to MESI, but:
  ➜ stores to shared data update all copies,
  ➜ updating cache assert share status,
  ➜ move to exclusive state if no other CPU holds copy.
- MIPS R4000 update protocol includes additional *modified-shared* state, which updates other caches but not RAM.

Wastes bus cycles if lines cease to be shared.

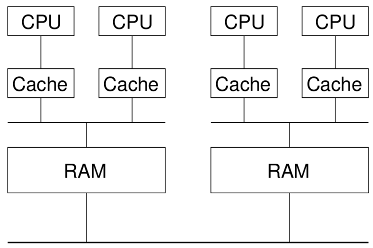
H/W cache coherency issues

- On miss may read data from other cache (faster).
- Some architectures (MIPS R4000) offer choice of protocols.
  ➜ Must chose most appropriate one for application.
- Cache coherency is based on cache lines.
  ➜ Potential of *false sharing*.
- H/W coherency generally restricted to *physical caches*.
  ➜ No problem with L2 cache.
  ➜ Use *inclusion property* for L1 cache: $L_1 \subset L_2$.

Why physical caches only?
Only physical addresses are on bus (normally)
⇒ no snooping possible with virtual caches

Non-Uniform Memory Architecture (NUMA)

Cache-coherent NUMA (cc-NUMA):



- Distributed system with hardware memory coherency.
- Performance depends critically on high hit rates in local RAM.

SMP Scheduling

How schedule a multiprocessor?

Issues:

Scalability: How many CPUs to support?

Application mix: SMP for

- time-shared multi-tasking environment,
- web server,
- highly parallel applications?

Architecture: caching, memory bandwidth...

Scheduler organisation

Single scheduler for all CPUs

- Not really SMP.
- Not scalable.

Global ready queue: CPU schedules itself from global queue.

- Course-grain locking of ready queue.
- Limited scalability.

Per-CPU ready queues

- Scalable.
- Load balancing?
- Process migration?

Issues: Address-space distribution

Restrict address spaces (tasks) to a single CPU.

+ Most sharing is within task.
  ➜ Good cache performance (maybe?)
+ Unmapping pages only affects single CPU.
  ➜ Only requires invalidating local TLB entries.
– No performance gain for multithreaded tasks.
  ➜ Multiple CPUs only enhance throughput.
  ➜ Not a general solution.

Gang scheduling (co-scheduling):

Always schedule all threads of a task at once on different CPUs.

+ Maximum concurrency for parallel applications.
+ Minimises intra-task communication latency.
– High bus contention.
+ Appropriate. for parallel number-crunching
– May have some CPUs idle.

⇒ Used mostly on supercomputers.

Fixed processor assignment

A thread has a *processor affinity* and will only run on that CPU:

+ Minimal contention for kernel data structures.
+ Minimal kernel communication overhead.
+ Cache friendly.
+ Highly scalable.
– No strict global priorities.
– No load balancing.

Ok for: ➜ non-real-time systems,
➜ mostly short processes,
➜ NUMA machines,
➜ with additional load balancing & process migration.

Real-time OS Issues

- Real-time processes characterised by a deadline.
- OS must be able to guarantee completion by deadline.
- Requires:
  ➜ predictable execution,
  ➜ predictable and limited system overheads,
  ➜ preemtability of long system calls,
    ➜ kernel locking, reentrancy...
    ➜ similar requirements as for SMP
  ➜ analysis of schedulability prior to process admission,
  ➜ careful scheduling.

Simplified Real-Time Process Model

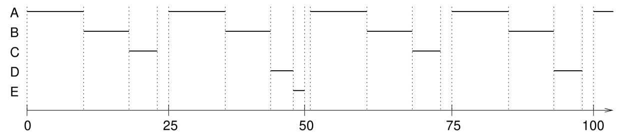
- Fixed set of processes,
- all processes periodic with known periods T_i,
- processes independent (note: no IPC!),
- ignore system overheads,
- deadline, D_i, equal to period,
- fixed (and known) worst-time execution time C_i,
- T_i, D_i, C_i are multiples of *minor cycle time* t_0.

Allows static analysis and a *static schedule*.

EXAMPLE

Process	A	B	C	D	E
T	25	25	50	50	100
C	10	8	5	4	2

- Worst case if all come at once: *critical instant*.
- Fixed schedule covers one *major cycle time* $t_1 = \text{lcm}\{D_i\}$.
- Schedule is just a list of executions ($t_0 = 25, t_1 = 100$):
  A, B, C,   A, B, D, E,   A, B, C,   A, B, D.



Note: No preemption $\Rightarrow$ **no** concurrency control necessary!

MORE FLEXIBLE ALTERNATIVE: USE PRIORITIES

- Priorities based on timeliness requirements, not “importance”.
- Higher-priority processes preempt lower-priority ones.

Frequently used scheme is *rate-monotonic priority assignment* (RMPA):

- Priority is based on period: $T_i < T_j \Rightarrow P_i > P_j$.
- Is *optimal* in a sense:
  ➜ Everything that *can* be scheduled can be done *statically* by RMPA.
  everything doable statically is doable with RMPA.

Schedulability

- Real-time OS must decide at process admission time whether all deadlines can be met.
- General result for RMPA [LL73]: Can do if

$$\sum_{i=1}^{N} \frac{C_i}{T_i} < N\left(2^{1/N} - 1\right).$$

- This is a sufficient (but not necessary) condition.

Limit (%):

N	1	2	3	4	5	10	∞
C_i/T_i	100.0	82.8	78.0	75.7	74.3	71.8	69.3

Sporadic (Non-Periodic) Processes

Use minimum (or average) inter-arrival interval for T_i.

- Generally $D_i \ll T_i$ for these.
- T_i may be irrelevant.
- Use D_i rather than T_i for priority assignment:
  ➜ *Deadline-monotonic priority ordering* (DMPO).

Note: soft/hard real-time guarantees average/minimal inter-arrival times.

See e.g., [BW96] for more.

Issues

- Hybrid systems:
  ➜ real-time plus best-effort tasks
- Stochastic real-time systems
  ➜ guarantee deadline is met with probability p
    ➜ more flexibility for OS
    ➜ hard to analyse

[BW96] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 2nd edition, 1996.

[CSL+87] D. Cornhill, L. Sha, J. Lehoczky, R. Rajkumar, and H. Tokuda. Limitations of Ada for real-time scheduling. *Ada Lett.*, pages 33–39, 1987. Wait-free locking.

[HH01] Michael Hohmuth and Hermann Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proc. 2001 Techn. Conf.*, Boston, MA, USA, 2001.

[LL73] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.

[PS96] Sharon Perl and Richard L. Sites. Studies of

Windows-NT performance using dynamic execution traces. In *Proc. 2nd OSDI*, pages 169–183, Oct 1996.

[Sch94] Curt Schimmel. *UNIX Systems for Modern Architectures*. Addison Wesley, 1994.