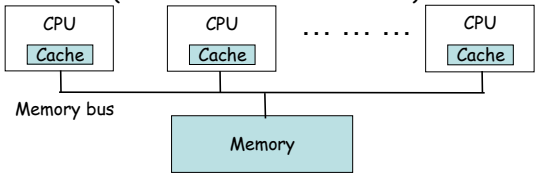
Multiprocessor Operating Systems

CS 256/456

Dept. of Computer Science, University of Rochester

Multiprocessor Hardware

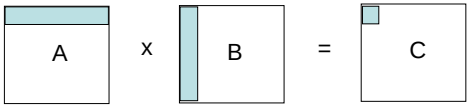
- A computer system in which two or more CPUs share full access to the main memory
- Each CPU might have its own cache and the coherence among multiple caches is maintained
  - write operation by a CPU is visible to all other CPUs
  - writes to the same location is seen in the same order by all CPUs (also called write serialization)



  - bus snooping and cache invalidation

Multiprocessor Applications

- Multiprogramming
  - Multiple regular applications running concurrently
- Concurrent servers
  - Web servers, … …
- Parallel programs
  - Utilizing multiple processors to complete one task (parallel matrix multiplication, Gaussian elimination)

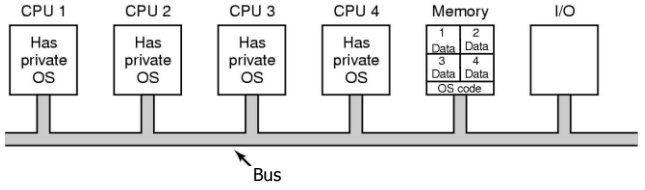


  - Strong synchronization

Single-processor OS vs. Multiprocessor OS

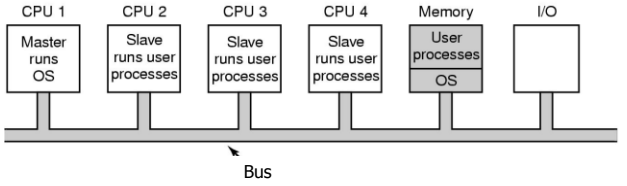
- Single-processor OS
  - easier to support kernel synchronization
    - coarse-grained locking vs. fine-grain locking
    - disabling interrupts to prevent concurrent executions
  - easier to perform scheduling
    - which to run, not where to run
- Multiprocessor OS
  - evolution of OS structure
  - synchronization
  - scheduling

Multiprocessor OS



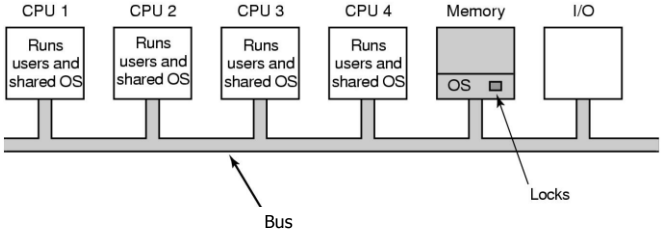
- Each CPU has its own operating system
  - quick to port from a single-processor OS
- Disadvantages
  - difficult to share things (processing cycles, memory, buffer cache)

Multiprocessor OS – Master/Slave



- All operating system functionality goes to one CPU
  - no multiprocessor concurrency in the kernel
- Disadvantage
  - OS CPU consumption may be large so the OS CPU becomes the bottleneck (especially in a machine with many CPUs)

Multiprocessor OS – Shared OS



- A single OS instance may run on all CPUs
- The OS itself must handle multiprocessor synchronization
  - multiple OS instances from multiple CPUs may access shared data structure

Preemptive Scheduling

- Use timer interrupts or signals to trigger involuntary yields
- Protect scheduler data structures by locking ready list, disabling/reenabling prior to/after rescheduling

yield:

```
disable_signals
enqueue(ready_list, current)
reschedule
re-enable_signals
```

Synchronization (Fine/Coarse-Grain Locking)

- Fine-grain locking – lock only what is necessary for critical section
- Coarse-grain locking – locking large piece of code, much of which is unnecessary
  - simplicity, robustness
  - prevent simultaneous execution

Simultaneous execution is not possible on uniprocessor anyway

Anderson et al. 1989 (IEEE TOCS)

- Raises issues of
  - Locality (per-processor data structures)
  - Granularity of scheduling tasks
  - Lock overhead
  - Tradeoff between throughput and latency
    - Large critical sections are good for best-case latency (low locking overhead) but bad for throughput (low parallelism)

Performance Measures

- Latency
  - Cost of thread management under the best case assumption of no contention for locks
- Throughput
  - Rate at which threads can be created, started, and finished when there is contention

Optimizations

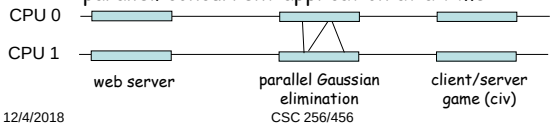
- Allocate stacks lazily
- Store deallocated control blocks and stacks in free lists
- Create per-processor ready lists
- Create local free lists for locality
- Queue of idle processors (in addition to queue of waiting threads)

Ready List Management

- Single lock for all data structures
- Multiple locks, one per data structure
- Local freelists for control blocks and stacks, single shared locked ready list
- Queue of idle processors with preallocated control block and stack waiting for work
- Local ready list per processor, each with its own lock

Multiprocessor Scheduling

- Timesharing
  - similar to uni-processor scheduling – one queue of ready tasks (protected by synchronization), a task is dequeued and executed when a processor is available
- Space sharing
- cache affinity
  - affinity-based scheduling – try to run each process on the processor that it last ran on
- cache sharing and synchronization of parallel/concurrent applications
  - gang/cohort scheduling – utilize all CPUs for one parallel/concurrent application at a time



SMP-CMP-SMT Multiprocessor

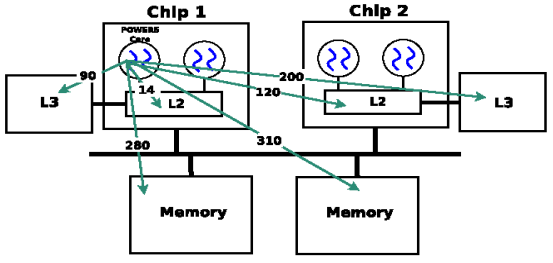
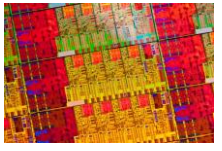


Image from http://www.eecg.toronto.edu/~tamda/papers/threadclustering.pdf

The Performance Transparency Challenge

Modern multicore systems…



10s to 100s of hardware contexts

- Resource sharing
  - Functional units, caches, on- and off-chip interconnects, memory
- Non-uniform access latencies

Operating System Level Resource Management To Date

- Capitalistic - generation of more requests results in more resource usage
  - Performance: resource contention can result in significantly reduced overall performance
  - Fairness: equal time slice does not necessarily guarantee equal progress
  - Sharing oblivious: extraneous communication due to poor placement

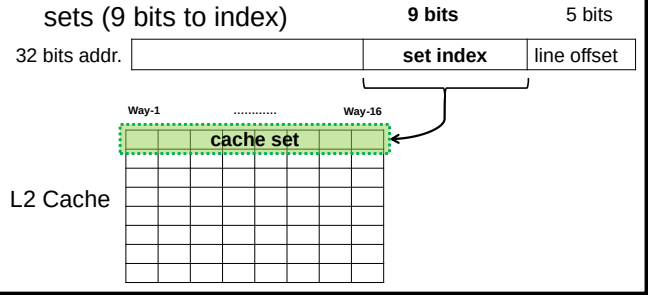
17

Multi-Core Cache Challenges

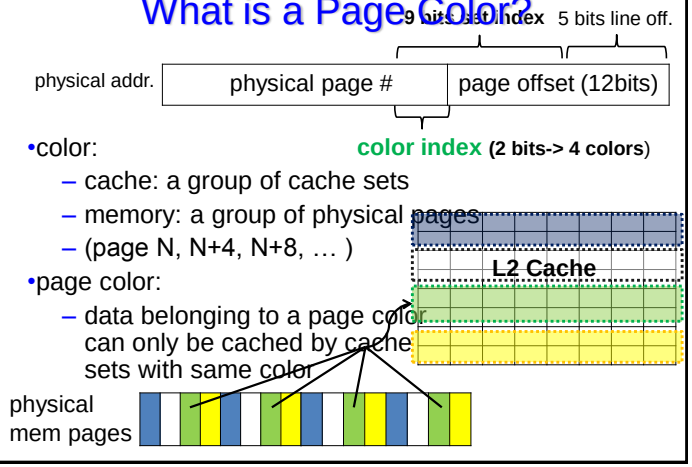
- Hardware manages cache at the grain of cache lines.
  - single program: data with different locality are mixed together
  - shared cache: uncontrolled sharing threads
  - compete for space -> interference
- Using OS as an **auxiliary** to manage cache
  - high-level knowledge of program
  - running state of the entire system
  - how? page coloring

Address Mapping in Cache

- physical memory address and cache
  - cache size = line size * way size * # of set
  - 256KB, 16-way, 32B line size L2 cache: 512 sets (9 bits to index)

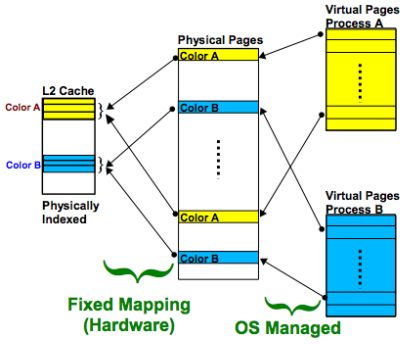


What is a Page Color?



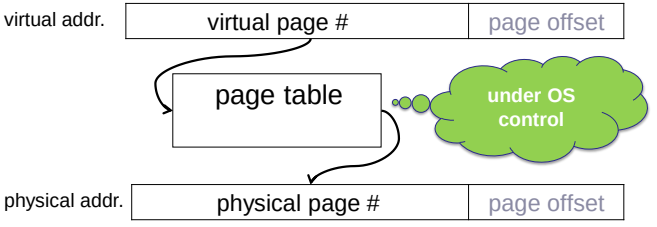
- color:
  - cache: a group of cache sets
  - memory: a group of physical pages
  - (page N, N+4, N+8, …)
- page color:
  - data belonging to a page color can only be cached by cache sets with same color

Software cache partitioning by page coloring – under OS control [EuroSys 2009]



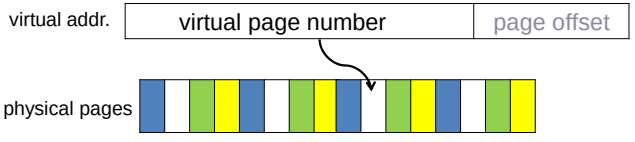
OS and Page Coloring

- What is the role of the OS?
  - control the mapping between virtual memory pages and physical pages via page table



OS and Page Coloring

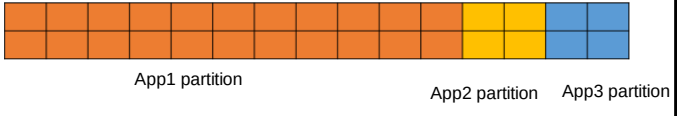
- Color a page: map a virtual page to a physical page with a particular color (lower bits of page #)



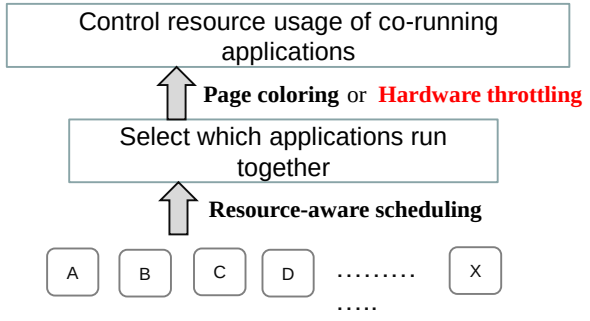
- Re-color a page: change the color at runtime
  - flush TLB, copy page data, modify page table
  - Incurs significant overhead (function of amount of data moved)

Intel's Hardware-Supported Cache Partitioning (2016)

- Intel Cache Allocation Technology – partition by ways (rather than sets) by writing to registers to specify the range of ways each core can access



Big Picture



Hardware Execution Throttling [Usenix 2009]

- Instead of directly controlling cache resource allocation, throttle the execution speed of application that overuses resource
- Available throttling knobs
  - Duty-cycle modulation
  - Frequency/voltage scaling
  - Cache prefetchers

New Mechanism: Hardware Execution Throttling [Usenix'09]

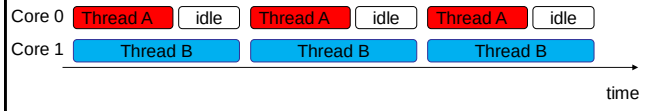
- Throttle the execution speed of app that overuses cache
  - Duty cycle modulation
    - CPU works only in duty cycles and stalls in non-duty cycles
    - Different from Dynamic Voltage Frequency Scaling
      - Per-core vs. per-processor control
      - Thermal vs. power management
  - Enable/disable cache prefetchers
    - L1 prefetchers
      - IP: keeps track of instruction pointer for load history
      - DCU: when detecting multiple loads from the same line within a time limit, prefetches the next line
    - L2 prefetchers
      - Adjacent line: Prefetches the adjacent line of required data
      - Stream: looks at streams of data for regular patterns

Comparing Hardware Execution Throttling to Page Coloring

- Kernel code modification complexity
  - Code length: 40 lines in a single file, as a reference our page coloring implementation takes 700+ lines of code crossing 10+ files
- Runtime overhead of configuration
  - Less than 1 microseconds, as a reference re-coloring a page takes 3 microseconds

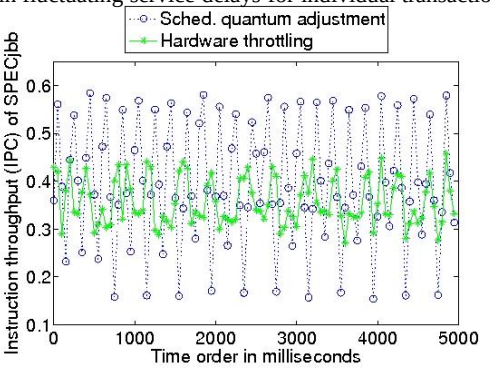
Existing Mechanism(II): Scheduling Quantum Adjustment

- Shorten the time slice of app that overuses cache
- May let core idle if there is no other active thread available



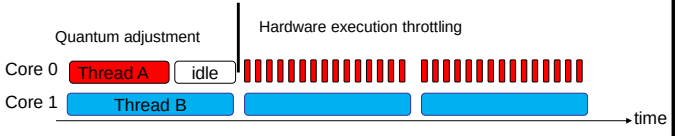
Drawback of Scheduling Quantum Adjustment

Coarse-grained control at scheduling quantum granularity may result in fluctuating service delays for individual transactions



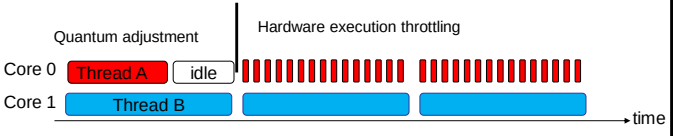
Comparison of Hardware Execution Throttling to other two mechanisms

- Comparison to page coloring
  - Little complexity to kernel
    - Code length: 40 lines in a single file, as a reference our page coloring implementation takes 700+ lines of code crossing 10+ files
  - Lightweight to configure
    - Read plus write register: duty-cycle 265 + 350 cycles, prefetcher 298 + 2065 cycles
    - Less than 1 microseconds, as a reference re-coloring a page takes 3 microseconds
- Comparison to scheduling quantum adjustment
  - More fine-grained controlling



Comparison of Hardware Execution Throttling to other two mechanisms

- Comparison to page coloring
  - Little complexity to kernel
    - Code length: 40 lines in a single file, as a reference our page coloring implementation takes 700+ lines of code crossing 10+ files
  - Lightweight to configure
    - Read plus write register: duty-cycle 265 + 350 cycles, prefetcher 298 + 2065 cycles
    - Less than 1 microseconds, as a reference re-coloring a page takes 3 microseconds
- Comparison to scheduling quantum adjustment
  - More fine-grained controlling



Policies for Hardware Throttling-Enabled Multicore Management

- User-defined service level agreements (SLAs)
  - Proportional progress among competing threads
    - Unfairness metric: coefficient of variation of threads' performance
  - Quality of service guarantee for high-priority application(s)
- Key challenge
  - Throttling configuration space grows exponentially as the number of cores increases
  - Quickly determining optimal or close to optimal throttling configurations is challenging

TEMM: A Flexible Framework for Throttling-Enabled Multicore Management [ICPP'12]

- Customizable performance estimation model
  - Reference configuration set and linear approximation
  - Currently incorporates duty cycle modulation and frequency/voltage scaling
- Iterative refinement
  - Prediction accuracy gets improved over time as more configurations are added into reference set