Advanced Topics: Multiprocessor Scheduling

CS 537: Introduction to Operating Systems

Louis Oliphant

University of Wisconsin - Madison

Fall 2024

Administrivia

- Project 6 due Dec 6th
- Final Exam:
  - Dec 19th, 10:05-12:05

Review VMMs

- Types of virtualization
  - Language Based VM
  - Lightweight VM
  - System-level VM (Simulators, Type-1 & Type-2 Hypbervisors)
- Popek / Goldberg Theorem
- Older hardware work-a-rounds (para-virtualization, dynamic binary translation)
- Modern hardware approaches (new root bit in x86 64-bit)
- How CPU and Memory are virtualized (trap and emulate)

Quiz 21 VMMs

https://tinyurl.com/cs537-fa24-q21a

Agenda

- Remembering CPU Scheduling
- Multiprocessor Architecture
  - cache and cache coherency
- Complications in scheduling due to multiprocessors
  - Synchronization
  - Cache affinity
- Single-Queue Scheduling
- Multi-Queue Scheduling
- Completely Fair Scheduler (CFS)

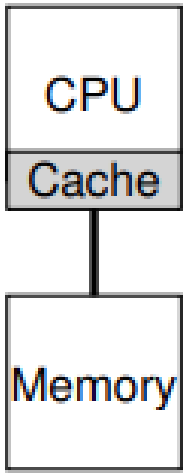
OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember addresses of...
	syscall handler
	timer handler
start interrupt timer	
	start timer
	interrupt CPU in X ms

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt	
	save regs(A) → k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) → proc_t(A)		
restore regs(B) ← proc_t(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) ← k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

Schedulers

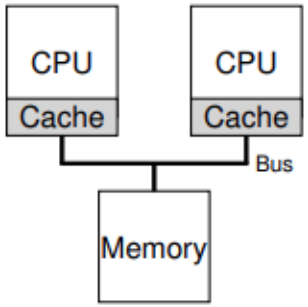
Keep a queue of runnable processes.
algorithms to select next process to run:

- FCFS, SJF, STCF, RR
- MLFQ (excellent overall performance for short jobs, fair & progress for long-running jobs)
- Stride (proportional-share)



Multiprocessor Architecture

- each cpu has its own cache
- Caches use the idea of locality (temporal and spatial) to speed performance




```
CPU 0       CPU 1
load a                      //load and store in cache
incr a
store a                     //only writes back to cache
            load a          //loads the old value from memory
            incr a
            store a
```

- **Cache Coherence** is the process of ensuring the information in each processor's cache is consistent with main memory.
  - **bus snooping** – each cache pays attentions to memory updates, can then **invalidate** or **update** its copy.

Synchronization

When accessing shared data items across CPUs, mutual exclusion primitives (e.g. mutex locks) should be used to guarantee correctness.

As the number of CPUs grows, access to a synchronized shared data structure becomes quit slow.

Cache Affinity

A process, when run on a particular CPU, builds up a fair bit of state in the caches of the CPU. The next time the process runs, it is often advantageous to run it on the same CPU, as it will run faster if some of its state is already present in the caches on that CPU.

A multiprocessor scheduler should consider cache affinity.

Single Queue Scheduling

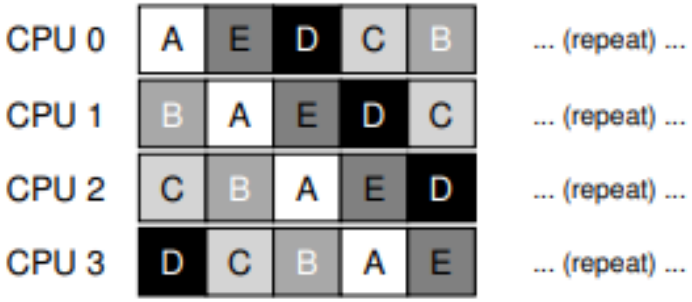
The most basic approach is to simply reuse the basic framework for single processor scheduling, by putting all jobs that need to be scheduled into a single queue.

This is called **single-queue multiprocessor scheduling (SQMS)**.

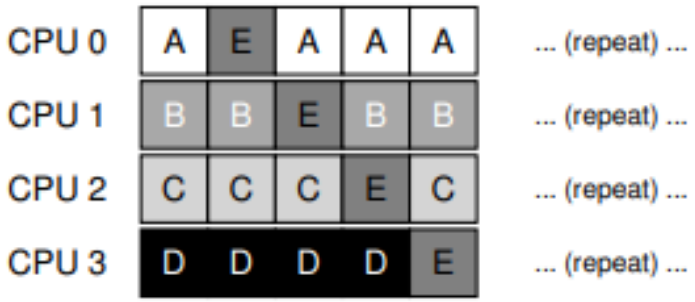
SQMS Shortcomings

- **Scalability** – many CPUs are accessing a shared data structure so **locking** can greatly reduce performance
- **Cache Affinity** – need an affinity mechanism.

No Affinity



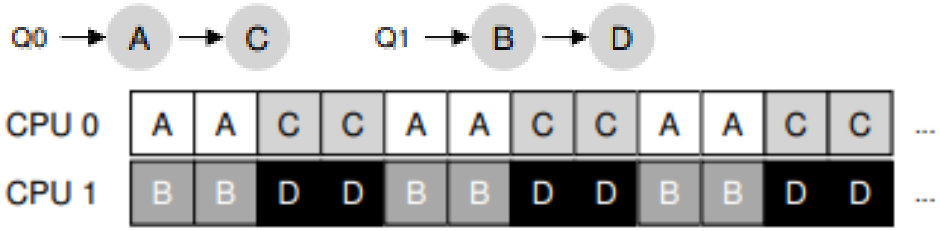
Affinity



Job E ends up **migrating** from CPU to CPU

Multi-Queue Scheduling

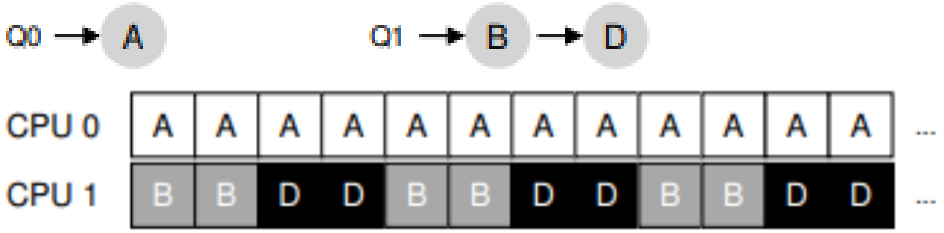
In **multi-queue multiprocessor scheduling (MQMS)** jobs are divided into multiple queues, one for each CPU. This avoids the problems of information sharing and synchronization.



MQMS is more scalable, as the number of CPUs grows, so too does the number of queues. It also intrinsically provides cache affinity.

New Problem – **load imbalance**

Load Imbalance



By **migrating** jobs from one queue to another true load balance can be achieved.

One basic approach is to use **work stealing**. A source queue that is low on jobs will occassionally peek at another target queue, to see how full it is. It the target is more full than the source, the source will "steal" one or more jobs.

Linux and FreeBSD Multiprocessor Schedulers

- Completely Fair Scheduler (CFS)
- O(1) scheduler
- BF Scheduler
- Unison Long Execution Scheduler (ULE) for FreeBSD

CFS

Uses fair-share scheduling.

As each process runs, it accumulates **virtual runtime (`vruntime`).** When a scheduling decision occurs, CFS will pick the process with the *lowest* `vruntime`.

CFS has various control parameters:

- **`sched_latency`** – deterimine how long one process should run before considering a switch (e.g. 48ms). CFS divides this value by the number (`n`) of processes running on the CPU to determine its time slice. Thus over this period of time CFS will be completely fair.
- **`min_granularity`** – With "too many" processes running, too many context switches will happen. This parameter controls the smallest time slice possible.

Additional CFS Features

- User's or administrators can give some processes a higher share by the Unix command `nice`. Values can be between -20 and +19 with a default of 0. Positive nice values imply *lower* priority. Nice values map to a weight for the process, which determine the process's time slice.
- CFS uses **red-black trees** for each CPU's queue. Modern systems sometimes have 1000s of processes. Processes are ordered by `vruntime`, and most operations are $O(logn)$ rather than $O(n)$ for a queue.
- CFS alters the `vruntime` of a newly awoken process to the minimum value in the tree (so the process doesn't dominate the CPU). This means processes that frequently sleep for short periods do not get their fair share of the CPU.
- cgroups – fairness has evolved to mean fairness between applications, so all threads in an application are in the same cgroup. `vruntimes` are calculated by cgroup.

CFS Multiprocessor Features

- cache usage – when a thread wakes up, it checks the difference between `vruntimes` of the thread and the currently running thread. It is is small ($\sim <$ 1ms) then the current running thread is not preempted.
- load balancing
  - Evens out *work* on all cores (not the same as evening out # threads per core). For instance, 1 CPU-intensive thread may be the same amount of work as 10 threads that mostly sleep. A *load* metric is calculated for each thread and core.
  - Every 4ms every core tries to steal work form other cores.
  - balancing occurs within a NUMA node. Balancing occurs between nodes if the load is high ($\sim >$ 25%) and longer distances means a higher imbalance must exist.

ULE Scheduler

- ULE has three queues per core – one for interactive threads, one for batch threads, and one for idle threads. A thread's interactivity is caculated based on the last 5 seconds. Newly created threads inherit from their parent.
- Inside each queue, threads are sorted by priority (a combination of *niceness* and *interactivity penalty* for interactive queue, and *niceness* and *runtime* for batch queue). Inside of the queues, there is one FIFO per priority.
- When picking next thread, first search interactive queue then batch queue.

ULE Multiprocess Features

- Thread assignment:
  - Newly created or awoken thread, uses affinity heuristic (if thread considered cache affine from last run)
  - Otherwise looks at highest topology that is affine, then finds core with minimum priority higher that this thread.
  - If this fails then pick core with lowest number of threads
- Thread balancing:
  - every 500-1500ms (randomly chosen) run load balancer, which can move at most one thread per core. A thread from the most loaded core is moved to the less loaded core. Iterates until no donor or receiver is found.
  - When interactive and batch queues are empty, ULE tries to steal from the most loaded cores.