Task Orchestration: Scheduling and Mapping on Multicore Systems

Course TBD
Lecture TBD
Term TBD

Module developed Spring 2013
by Apan Qasem

This module created with support from NSF under grant # DUE 1141022
Outline

- Scheduling for parallel systems
  - Load balancing
  - Thread affinity
  - Resource sharing

- Hardware threads (SMT)
- Multicore architecture

- Significance of the multicore paradigm shift
Increase in Transistor Count

Moore’s Law
Increase in Performance

- Improvements in chip architecture
- Increases in clock speed
- Multiple instructions per cycle
- Speculative out-of-order execution
- Instruction pipeline
- Internal memory cache
- Full Speed Level 2 Cache
- MMX (multimedia extensions)
- Longer issue pipeline double speed arithmetic
- Hyper threading
- Dual-core
- Quad-core
- Hex-core

Image adapted from Scientific American, Feb 2005, “A Split at the Core”
Figure 1. In CPU architecture today, heat is becoming an unmanageable problem. (Courtesy of Pat Gelsinger, Intel Developer Forum, Spring 2004)
The Power Wall

- Going with Moore’s Law results in too much heat dissipation and power consumption
- Moore’s law still holds but does not seem to be economically viable

- The multicore paradigm
  - Put multiple simplified (and slower) processing cores on the same chip area
  - Fewer transistors per cm$^2$ implies less heat!
Why is Multicore Such a Big Deal?

- Improvements in chip architecture
- Increases in clock speed
- Longer issue pipeline
- Double speed arithmetic
- Speculative out-of-order execution
- Internal memory cache
- Multiple instructions per cycle
- MMX (multimedia extensions)
- Instruction pipeline

- Hyper threading
- Full Speed Level 2 Cache
- 3 GHz
- 2.6 GHz
- 3.3 GHz
- 2.93 GHz

- More responsibility on software

Theoretical maximum performance (millions of operations per second)


Clock speeds: 16 MHz, 25 MHz, 33 MHz, 50 MHz, 66 MHz, 200 MHz, 300 MHz, 733 MHz, 2000 MHz, 3 GHz, 2.6 GHz, 3.3 GHz, 2.93 GHz
Why is Multicore Such a Big Deal?

Parallelism is mainstream

“In future, all software will be parallel”
- Andrew Chien, Intel CTO (many concur)

• Parallelism no longer a matter of interest just for the HPC people
• Need to find more programs to parallelize
• Need to find more parallelism in existing parallel applications
• Need to consider parallelism when writing any new program
Why is Multicore Such a Big Deal?

Parallelism is Ubiquitous
OS Role in the Multicore Era

- There are several considerations for multicore operating systems
  - Scalability
  - Sharing and contention of resources
  - Non-uniform communication latency

- Biggest challenge is in scheduling of threads across the system
Scheduling Goals

- Most of the goals don’t change when scheduling for multicore or multi-processor systems

- Still care about
  - CPU utilization
  - Throughput
  - Turnaround time
  - Wait time
  - Fairness

definitions may become more complex
Scheduling Goals for Multicore Systems

- Multicore systems give rise to a new set of goals for the OS scheduler
  - Load balancing
  - Resource sharing
  - Energy usage and power consumption
Single CPU Scheduler Overview

- Long-term Queue
- Ready Queue
- CPU
- I/O Queue

Diagram shows the flow of processes through the Long-term Queue, Ready Queue, and CPU, with an I/O Queue connection.
Multicore Scheduler Overview 1

Cores sharing the same short-term queues
Multicore Scheduler Overview II

Cores with individual ready queues

Long-term Queue

Ready Queue

I/O Queue

Core 0

Core 1

Core 2

Core 3

No special handling required for many of the single CPU scheduling algorithms
Load Balancing

- Goal is to distribute tasks across all cores on the system such that CPU and other resources are utilized evenly.
- A load-balanced system will generally lead to higher throughput.

![Graphs showing load balancing scenarios]

Scenario 1
- Bad load distribution

Scenario 2
- Good load distribution
The OS as a Load Balancer

Main strategy

- Identify metric for balanced load
  - average number of processes waiting in ready queues
  - (aka load average)

- Track load balance metric
  - probe ready queues
  - uptime and who utilities

- Migrate threads if a core exhibits a high average load
  - If load average for cores 0-3 is 2, 3, 1, and 17 then move threads from core3 to core 2
Thread Migration

• The process of moving a thread from one ready queue to another is known as thread migration

• Operating systems implement two types of thread migration mechanism
  • Push migration
    • Run a separate process that will migrate a thread from one core to another
    • May be integrated within the kernel as well
  • Pull migration (aka work stealing)
    • Each core fetches threads from other cores
Complex Load Balancing Issues

Assume only one thread executing on each core

Does it help to load balance this workload?

Will it make a difference in performance?

Thread migration or load balancing will only help with overall performance if t1 runs faster when moved to core0, core2 or core3
Load Balancing for Power

This unbalanced workload can have huge implications for power consumption:

Power consumption is tied to how fast a processor is running:

\[ P = cv^2f \quad \text{and} \quad P \propto t \]

- \( P \) = power
- \( f \) = frequency
- \( c \) = static power dissipation
- \( v \) = voltage
- \( t \) = temperature

One core running at a higher frequency than the others may result in more heat dissipation and overall increased power consumption.
Load Balancing for Power

- Operating Systems are incorporating power metrics in thread migration and load balancing decisions
- Achieve power-balance by eliminating hotspots

Can also try to change the frequency
  - AMD Powernow, Intel Sidestep
Can utilize hardware performance counters
  - core utilization
  - core temperature
Linux implements this type of scheduling
  - sched_mc_power_savings
Load Balancing Trade-offs

- Thread migration may involve multiple context switch on more than one core
- Context switches can be very expensive
- Potential gains from load balancing must be weighed against the increased cost from context switches

- Load balancing policy may conflict with CFS scheduling
  - balanced load does not imply fair sharing of resources
Resource Sharing

Current multicore and SMT systems share resources at various levels

OS needs to be aware of resource utilization in making scheduling decisions
Thread Affinity

• Tendency for a process to run on a given core for as long as possible without being migrated to a different core
  • aka CPU affinity and processor affinity
• The operating system uses the notion of thread affinity in performing resource-aware scheduling on multicore systems

• If a thread has affinity for a specific core (or core group) then priority should be given to schedule the map and schedule the thread to that specific core (or group)

• Current approach is to use thread affinity to address skewed workloads
  • Start with default (all cores)
  • Set affinity of task I to core j if core j is deemed underutilized
Adjusting Thread Affinity

- Linux kernel maintains the `task_struct` data structure for every process in the system.
- Affinity information is stored as a bitmask in the `cpus_allowed` field.
- Can modify or retrieve the affinity of a thread from user-space:
  - `sched_setaffinity()`, `sched_getaffinity()`
  - `pthread_setaffinity_np()`, `pthread_getaffinity_np()`
  - `taskset`
Thread Affinity in the Linux Kernel

/* Look for allowed, online CPU in same node. */
for_each_cpu_and(dest_cpu, nodemask, cpu_active_mask)
    if (cpumask_test_cpu(dest_cpu, &p->cpus_allowed))
        return dest_cpu;

/* Any allowed, online CPU? */
dest_cpu = cpumask_any_and(&p->cpus_allowed, cpu_active_mask);
if (dest_cpu < nr_cpu_ids)
    return dest_cpu;

/* No more Mr. Nice Guy. */
if (unlikely(dest_cpu >= nr_cpu_ids)) {
    dest_cpu = cpuset_cpus_allowed_fallback(p);
    /*
     * Don't tell them about moving exiting tasks or
     * kernel threads (both mm NULL), since they never
     * leave kernel.
     */
    if (p->mm && printk_ratelimit()) {
        printk(KERN_INFO "process %d (%s) no 
        "longer affine to cpu%d\n",
                task_pid_nr(p), p->comm, cpu);
    }
}
Affinity Based Scheduling

- Affinity based scheduling can be performed under different criteria, using different heuristics.

- Assume 4 cores with two L2 shared between core 0 and core 1, core 2 and core 3, L3 shared among all cores.
Affinity Based Scheduling

Scheduling for data locality in L2

- A thread $t_i$ that shares data with $t_j$ should be placed in the same affinity group
- Can lead to unbalanced ready-queues but improved memory performance
Affinity Based Scheduling

Poor schedule for producer-consumer program

Good schedule for producer-consumer program
Affinity Based Scheduling

- Scheduling for better cache utilization
  - Thread $t_i$ and $t_j$ only utilize 10% of the cache, $t_p$ and $t_q$ each demand 80% of the cache
  - Schedule $t_i$ and $t_p$ on core 0 and core 1, $t_j$ and $t_q$ on core 2 and core 3
  - May lead to loss of locality
Affinity Based Scheduling

Scheduling for better power management

- $t_i$ and $t_j$ are CPU-bound while $t_p$ and $t_q$ are memory-bound
- Schedule $t_i$ and $t_p$ on core 0 and $t_j$ and $t_q$ to core 2

![Diagram showing scheduling of tasks on cores](image)
Gang Scheduling

- Two-step scheduling process
  - Identify a set (or gang) of threads to and adjust affinity for them to execute in a specific core group
    - Gang formation can be done based on resource utilization and sharing
    - Suspend the threads in a gang to let one job have dedicated access to the resources for a configured period of time

- Traditionally used for MPI programs running on high-performance clusters

- Becoming mainstream for multicore architectures
  - Patches available that integrates gang scheduling with CFS in Linux