ADRIAN PERRIG & TORSTEN HOEFLER

Networks and Operating Systems (252-0062-00)
Chapter 3: Scheduling

Stack Overflow Could Explain Toyota Vehicles' Unintended Acceleration

timothy posted 4 days ago | from robertchin

650

Image (c) 2007 Shuets Udono [CC BY-SA 2.0] via Wikimedia Commons

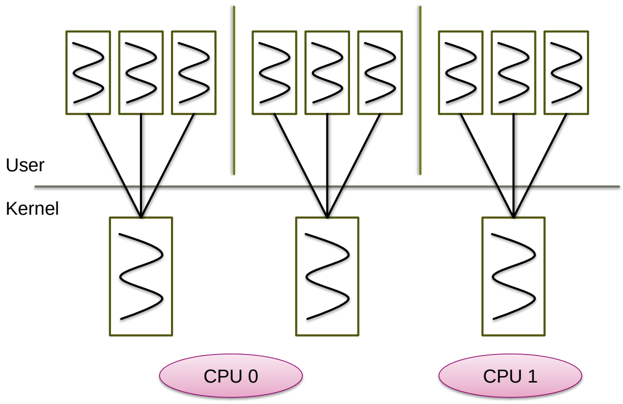
New submitter robertchin writes "Michael Barr recently testified in the Bookout v. Toyota Motor Corp lawsuit that the likely cause of unintentional acceleration in the Toyota Camry may have been caused by a stack overflow. Due to recursion overwriting critical data past the end of the stack and into the real time operating system memory area, the throttle was left in an open state and the process that controlled the throttle was terminated. How can users protect themselves from sometimes life endangering software bugs?"

Source: slashdot, Feb. 2014

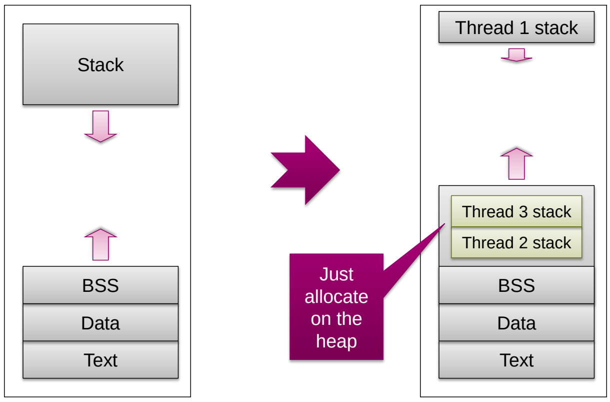
Many-to-one threads

- **Early "thread libraries"**
  - Green threads (original Java VM)
  - GNU Portable Threads
  - Standard student exercise: implement them!
- **Sometimes called "pure user-level threads"**
  - aka. lightweight threads, tasks (differences in control)
  - No kernel support required
  - Also (confusingly) "Lightweight Processes"

Many-to-one threads



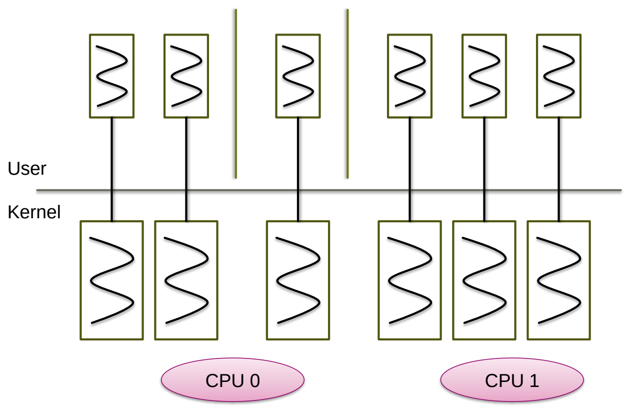
Address space layout for user level threads



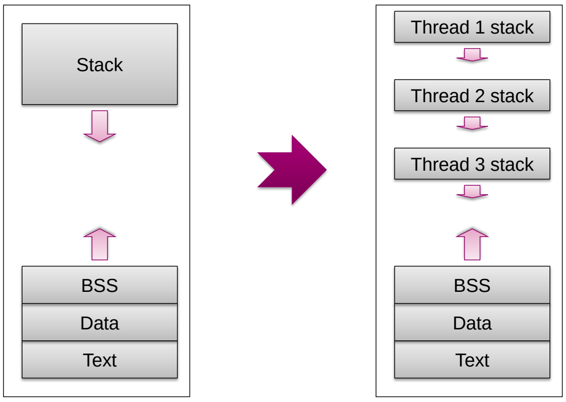
One-to-one user threads

- **Every user thread is/has a kernel thread.**
- **Equivalent to:**
  - multiple processes sharing an address space
  - Except that "process" now refers to a group of threads
- **Most modern OS threads packages:**
  - Linux, Solaris, Windows XP, MacOSX, etc.

One-to-one user threads



One-to-one user threads



Comparison

User-level threads

- **Cheap to create and destroy**
- **Fast to context switch**
- **Can block entire process**
  - Not just on system calls

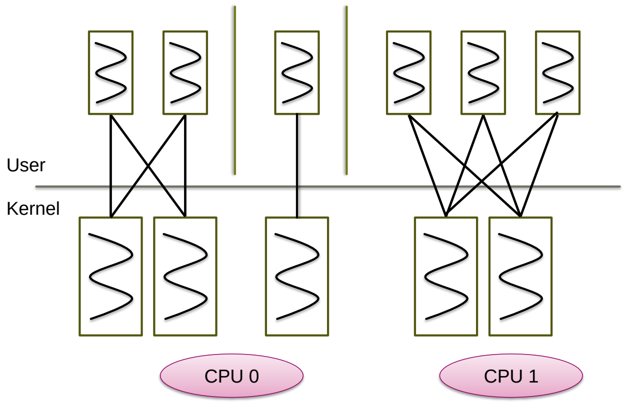
One-to-one threads

- **Memory usage (kernel stack)**
- **Slow to switch**
- **Easier to schedule**
- **Nicely handles blocking**

Many-to-many threads

- **Multiplex user-level threads over several kernel-level threads**
- **Only way to go for a multiprocessor**
  - I.e., pretty much everything these days
- **Can "pin" user thread to kernel thread for performance/predictability**
- **Thread migration costs are "interesting"…**

Many-to-many threads



Administrivia

- **I try to indicate book chapters**
  - But this will not be complete (no book covers 100%)
  - So consider it a rough approximation
  - Last lecture OSPP Sections 3.1 and 4.1
- **Lecture recording**
  - http://www.multimedia.ethz.ch/lectures/infk/2013/spring/252-0062-00L
  - Content of the OS part did not change
- **Please let me know if you find the quick quiz silly!**

A Small Quiz

- **True or false (raise hand)**
  - A process has a virtual CPU
  - A thread has a virtual CPU
  - A thread has a private set of open files
  - A process is a resource container
  - A context switch can be caused by a thread
  - When a process calls a blocking I/O, it is put into runnable state
  - A zombie is a dead process waiting for its parent
  - Simple user-level threads run efficiently on multiprocessors
  - A device can trigger a system call
  - A device can trigger an upcall
  - Unix fork() starts a new program
  - Windows CreateProcess starts a new program
  - A buggy process can overwrite the stack of another process
  - User-level threads can context switch without a syscall
  - The scheduler always runs in a kernel thread

Last time

- **Process concepts and lifecycle**
- **Context switching**
- **Process creation**
- **Kernel threads**
- **Kernel architecture**
- **System calls in more detail**
- **User-space threads**

- **This time**
  - OSPP Chapter 7

Scheduling is…

Deciding how to allocate a single resource among multiple clients
- In **what order** and for **how long**

- **Usually refers to CPU scheduling**
  - Focus of this lecture – we will look at selected systems/research
  - OS also schedules other resources (e.g., disk and network IO)

- **CPU scheduling involves deciding:**
  - Which task next on a given CPU?
  - For how long should a given task run?
  - On which CPU should a task run?

  Task: process, thread, domain, dispatcher, …

Scheduling

- **What metric is to be optimized?**
  - Fairness (but what does this mean?)
  - Policy (of some kind)
  - Balance/utilization (keep everything being used)
  - Increasingly: Power (or Energy usage)

- **Usually these are in contradiction…**

Objectives

- **General:**
  - Fairness
  - Enforcement of policy
  - Balance/utilization

- **Others depend on workload, or architecture:**
  - Batch jobs, interactive, realtime and multimedia
  - SMP, SMT, NUMA, multi-node

Challenge: Complexity of scheduling algorithms

- **Scheduler needs CPU to decide what to schedule**
  - Any time spent in scheduler is "wasted" time
  - Want to minimize overhead of decisions
    To maximize utilization of CPU
  - But low overhead is no good if your scheduler picks the "wrong" things to run!

⇒ **Trade-off between:**
scheduler complexity/overhead and
quality of resulting schedule

Challenge: Frequency of scheduling decisions

- **Increased scheduling frequency**
  ⇒ increasing chance of running something different

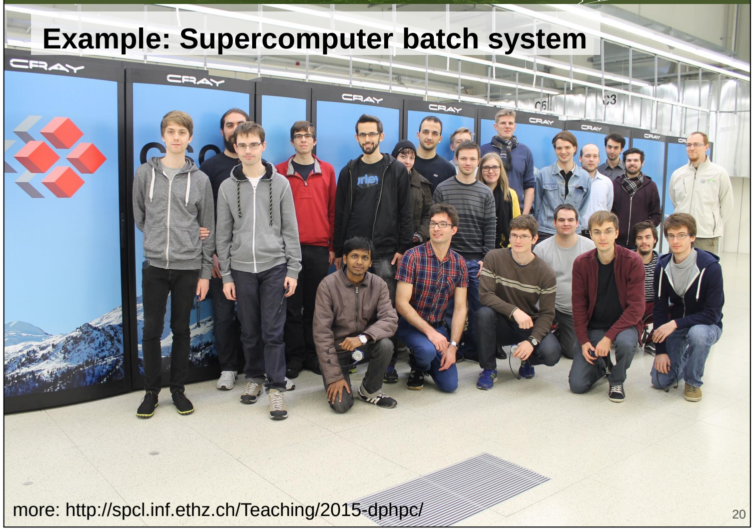
Leads to higher context switching rates,
⇒ lower throughput
- Flush pipeline, reload register state
- Maybe flush TLB, caches
- Reduces locality (e.g., in cache)

Batch workloads

- **"Run this job to completion and tell me when you're done"**
  - Typical mainframe or supercomputer use-case
  - Used in large clusters of different sorts …
- **Goals:**
  - Throughput (jobs per hour)
  - Wait time (time to execution)
  - Turnaround time (submission to termination)
  - Utilization (don't waste resources)

ETH zürich spcl.inf.ethz.ch @spcl_eth

Example: Supercomputer batch system



more: http://spcl.inf.ethz.ch/Teaching/2015-dphpc/

ETH zürich spcl.inf.ethz.ch @spcl_eth

Interactive workloads

- **"Wait for external events, and react before the user gets annoyed"**
  - Word processing, browsing, fragging, etc.
  - Common for PCs, phones, etc.
- **Goals:**
  - Response time: how quickly does something happen?
  - Proportionality: some things should be quicker

ETH zürich spcl.inf.ethz.ch @spcl_eth

Soft realtime workloads

- **"This task must complete in less than 50ms", or**
- **"This program must get 10ms CPU every 50ms"**
  - Data acquisition, I/O processing
  - Multimedia applications (audio and video)
- **Goals:**
  - Deadlines
  - Guarantees
  - Predictability (real time ≠ fast!)

ETH zürich spcl.inf.ethz.ch @spcl_eth

Hard realtime workloads

- **"Ensure the plane's control surfaces move correctly in response to the pilot's actions"**
- **"Fire the spark plugs in the car's engine at the right time"**
  - Mission-critical, extremely time-sensitive control applications
- **Not covered in this course: very different techniques required…**

ETH zürich spcl.inf.ethz.ch @spcl_eth

Scheduling assumptions and definitions

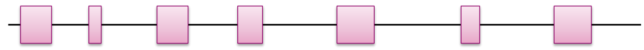
CPU- and I/O-bound tasks

CPU-bound task:

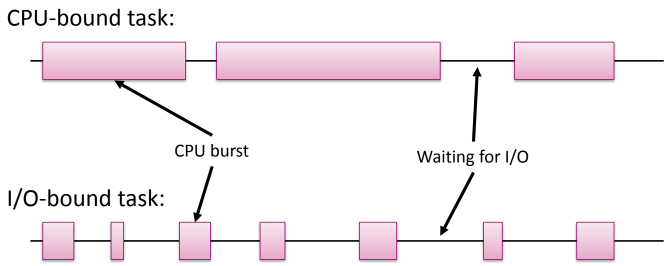
CPU- and I/O-bound tasks

CPU-bound task:

I/O-bound task:



CPU- and I/O-bound tasks



Simplifying assumptions

- **Only one processor**
  - We'll relax this (much) later
- **Processor runs at fixed speed**
  - Realtime == CPU time
  - Not true in reality for power reasons
  - DVFS: Dynamic Voltage and Frequency Scaling
  - In many cases, however, efficiency ⇒ run flat-out until idle

Simplifying assumptions

- **We only consider work-conserving scheduling**
  - No processor is idle if there is a runnable task
  - Question: is this always a reasonable assumption?
- **The system can always preempt a task**
  - Rules out some very small embedded systems
  - And hard real-time systems…
  - And early PC/Mac OSes…

When to schedule?

When:

1. **A running process blocks (or calls yield())**
   - e.g., initiates blocking I/O or waits on a child
2. **A blocked process unblocks**
   - I/O completes
3. **A running or waiting process terminates**
4. **An interrupt occurs**
   - I/O or timer

- **2 or 4 can involve *preemption***

Preemption

- **Non-preemptive scheduling:**
  - Require each process to explicitly give up the scheduler
    - *Start I/O, executes a "yield()" call, etc.*
  - Windows 3.1, older MacOS, some embedded systems
- **Preemptive scheduling:**
  - Processes dispatched and descheduled without warning
    - *Often on a timer interrupt, page fault, etc.*
  - The most common case in most OSes
  - Soft-realtime systems are usually preemptive
  - Hard-realtime systems are often not!

Overhead

- **Dispatch latency:**
  - Time taken to dispatch a runnable process
- **Scheduling cost**
  = 2 x (half context switch) + (scheduling time)
- **Time slice allocated to a process should be significantly more than scheduling overhead!**

Overhead example (from Tanenbaum)

- **Suppose process switch time is 1ms**
- **Run each process for 4ms**
  - What is the overhead?

Overhead example (from Tanenbaum)

- **Suppose process switch time is 1ms**
- **Run each process for 4ms**
  ⇒ 20% of system time spent in scheduler ☹
- **Run each process for 100ms**
  50 jobs ⇒ maximum response time?

Overhead example (from Tanenbaum)

- **Suppose process switch time is 1ms**
- **Run each process for 4ms**
  ⇒ 20% of system time spent in scheduler ☹
- **Run each process for 100ms**
  50 jobs ⇒ response time up to 5 seconds ☹
- **Tradeoff: response time vs. scheduling overhead**

Batch-oriented scheduling

Batch scheduling: why bother?

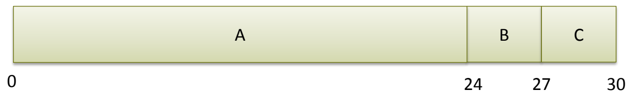
- **Mainframes are sooooo 1970!**
- **But:**
  - Most systems have batch-like background tasks
  - Yes, even phones are beginning to
  - CPU bursts can be modeled as batch jobs
  - Web services are **request-based**

facebook Google bing

First-come first-served

- **Simplest algorithm!**
- **Example:**
  - Waiting times: 0, 24, 27
  - Avg. = (0+24+27)/3 = 17
- **But..**

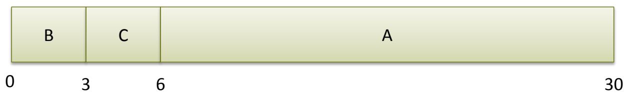
Task	Execution time
A	24
B	3
C	3



First-come first-served

- **Different arrival order**
- **Example:**
  - Waiting times: 6, 0, 3
  - Avg. = (0+3+6)/3 = 3
- **Much better ☺**
- **But unpredictable ☹**

Task	Execution time
A	24
B	3
C	3



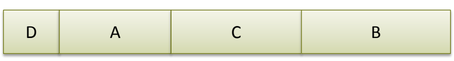
Convoy phenomenon

- **Short processes back up behind long-running processes**
- **Well-known (and widely seen!) problem**
  - Famously identified in databases with disk I/O
  - Simple form of self-synchronization
- **Generally undesirable…**
- **FIFO used for, e.g., `memcached`**

Shortest-job first

- **Always run process with the shortest execution time.**
- **Optimal: minimizes waiting time (and hence turnaround time)**

Task	Execution time
A	6
B	8
C	7
D	3



Optimality

- **Consider *n* jobs executed in sequence, each with processing time t_i, $0 \le i < n$**
- **Mean turnaround time is:** $$Avg. = \frac{1}{n}\sum_{i=0}^{n-1}(n-i)\cdot t_i$$
- **Minimized when shortest job is first**
- **E.g., for 4 jobs:** $$\frac{(4t_0 + 3t_1 + 2t_2 + t_3)}{4}$$

Execution time estimation

- **Problem: what is the execution time?**
  - For mainframes or supercomputers, could punt to user
  - And charge them more if they were wrong
- **For non-batch workloads, use CPU burst times**
  - Keep exponential average of prior bursts
  - cf., TCP RTT estimator $\tau_{n+1} = \alpha \cdot t_n + (1-\alpha) \cdot \tau_n$
- **Or just use application information**
  - Web pages: size of web page

SJF & preemption

- **Problem: jobs arrive all the time**
- **"Shortest remaining time next"**
  - New, short jobs may preempt longer jobs already running
- **Still not an ideal match for dynamic, unpredictable workloads**
  - In particular, interactive ones

Scheduling interactive loads

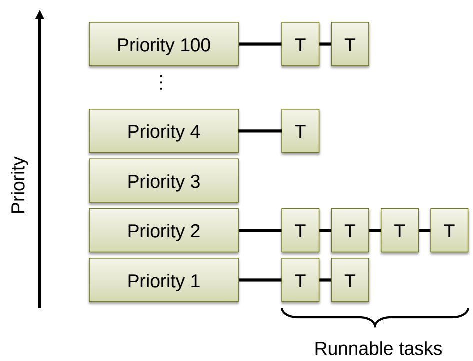
Round-robin

- **Simplest interactive algorithm**
- **Run all runnable tasks for fixed quantum in turn**
- **Advantages:**
  - It's easy to implement
  - It's easy to understand, and analyze
  - Higher turnaround time than SJF, but better *response*
- **Disadvantages:**
  - It's rarely what you want
  - Treats all tasks the same

Priority

- **Very general class of scheduling algorithms**
- **Assign every task a priority**
- **Dispatch highest priority runnable task**
- **Priorities can be dynamically changed**
- **Schedule processes with same priority using**
  - Round Robin
  - FCFS
  - etc.

Priority queues



Multi-level queues

- **Can schedule different priority levels differently:**
  - Interactive, high-priority: round robin
  - Batch, background, low priority, real time: FCFS
- **Ideally generalizes to *hierarchical scheduling***

Starvation

- **Strict priority schemes do not guarantee progress for all tasks**
- **Solution: *Ageing***
  - Tasks which have waited a long time are gradually increased in priority
  - Eventually, any starving task ends up with the highest priority
  - Reset priority when quantum is used up

Multilevel Feedback Queues

- **Idea: penalize CPU-bound tasks to benefit I/O bound tasks**
  - Reduce priority for processes which consume their entire quantum
  - Eventually, re-promote process
  - I/O bound tasks tend to block before using their quantum ⇒ remain at high priority
- **Very general: any scheduling algorithm can reduce to this (problem is implementation)**

Example: Linux o(1) scheduler

- **140 level Multilevel Feedback Queue**
  - 0-99 (high priority):
    static, fixed, "realtime"
    FCFS or RR
  - 100-139: User tasks, dynamic
    Round-robin within a priority level
    Priority ageing for interactive (I/O intensive) tasks
- **Complexity of scheduling is independent of no. tasks**
  - Two arrays of queues: "runnable" & "waiting"
  - When no more task in "runnable" array, swap arrays

Example: Linux "completely fair scheduler"

- **Task's priority = how little progress it has made**
  - Adjusted by fudge factors over time
  - Get "bonus" if a task yields early (his time is distributed evenly)
- **Implementation uses Red-Black tree**
  - Sorted list of tasks
  - Operations now O(log n), but this is fast
- **Essentially, this is the old idea of "fair queuing" from packet networks**
  - Also called "generalized processor scheduling"
  - Ensures guaranteed service rate for all processes
  - CFS does not, however, expose (or maintain) the guarantees

Problems with UNIX Scheduling

- **UNIX conflates protection domain and resource principal**
  - Priorities and scheduling decisions are per-process (thread)
- **However, may want to allocate resources across processes, or separate resource allocation within a process**
  - E.g., web server structure
    Multi-process
    Multi-threaded
    Event-driven
  - If I run more compiler jobs than you, I get more CPU time
- **In-kernel processing is accounted to nobody**

Resource Containers [Banga et al., 1999]

New OS **abstraction** for explicit resource management, separate from process structure

- Operations to create/destroy, manage hierarchy, and associate threads or sockets with containers
- Independent of scheduling algorithms used
- All kernel operations and resource usage accounted to a resource container

⇒ Explicit and fine-grained control over resource usage
⇒ Protects against some forms of DoS attack

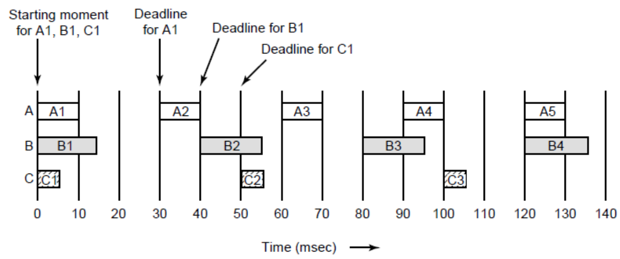
- Most obvious modern form: *virtual machines, containers*

Real Time

Real-time scheduling

- **Problem: giving real time-based guarantees to tasks**
  - Tasks can appear at any time
  - Tasks can have deadlines
  - Execution time is generally known
  - Tasks can be periodic or aperiodic
- **Must be possible to reject tasks which are unschedulable, or which would result in no feasible schedule**

Example: multimedia scheduling



Rate-monotonic scheduling

- **Schedule periodic tasks by always running task with shortest period first.**
  - Static (offline) scheduling algorithm
- **Suppose:**
  - m tasks
  - C_i is the execution time of i'th task
  - P_i is the period of i'th task
- **Then RMS will find a feasible schedule if:**

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m\,(2^{1/m} - 1)$$

- **(Proof is beyond scope of this course)**

Earliest deadline first

- **Schedule task with earliest deadline first (duh..)**
  - Dynamic, online.
  - Tasks don't *actually* have to be periodic…
  - More complex - O(n) – for scheduling decisions
- **EDF will find a feasible schedule if:**

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- **Which is very handy. Assuming zero context switch time…**

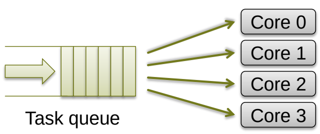
Guaranteeing processor rate

- **E.g., you can use EDF to guarantee a rate of progress for a long-running task**
  - Break task into periodic jobs, period *p* and time *s*.
  - A task arrives at start of a period
  - Deadline is the end of the period
- **Provides a *reservation* scheduler which:**
  - Ensures task gets *s* seconds of time every *p* seconds
  - Approximates weighted fair queuing
- **Algorithm is regularly rediscovered…**

Multiprocessor Scheduling

Challenge 1: sequential programs on multiprocessors

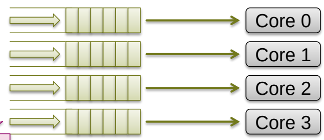
- **Queuing theory ⇒ straightforward, although:**
  - More complex than uniprocessor scheduling
  - Harder to analyze



but…

It's much harder

- **Overhead of locking and sharing queue**
  - Classic case of scaling bottleneck in OS design
- **Solution: per-processor scheduling queues**



In practice, each is more complex e.g., MFQ

It's much harder

- **Threads allocated arbitrarily to cores**
  - ⇒ tend to move between cores
  - ⇒ tend to move between caches
  - ⇒ really bad locality and hence performance
- **Solution: affinity scheduling**
  - Keep each thread on a core most of the time
  - Periodically rebalance across cores
  - Note: this is *non-work-conserving!*
- **Alternative: hierarchical scheduling (Linux)**

Challenge 2: parallel applications

- **Global barriers in parallel applications ⇒ One slow thread has huge effect on performance**
  - Corollary of *Amdahl's Law*
- **Multiple threads would benefit from cache sharing**
- **Different applications pollute each others' caches**
- **Leads to concept of "co-scheduling"**
  - Try to schedule all threads of an application together
- **Critically dependent on *synchronization* concepts**

Multicore scheduling

- **Multiprocessor scheduling is two-dimensional**
  - When to schedule a task?
  - Where (which core) to schedule on?
- **General problem is NP hard ☹**
- **But it's worse than that:**
  - Don't want a process holding a lock to sleep
    ⇒ Might be other running tasks spinning on it
  - Not all cores are equal
- **In general, this is a wide-open research problem**

Little's Law

- **Assume, in a train station:**
  - 100 people arrive per minute
  - Each person spends 15 minutes in the station
  - How big does the station have to be (house how many people)
- **Little's law: "*The average number of active tasks in a system is equal to the average arrival rate multiplied by the average time a task spends in a system*"**