Towards policy-free Microkernels

WAMOS 2015

Second Wiesbaden Workshop on
Advanced Microkernel Operating Systems

Olga Dedi 06.08.2015

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

Outline

- Introduction
- Motivation
- Difference between policy and mechanism
- CPU Inheritance Scheduling
  - General approach
  - Threads
  - Requesting CPU Time
  - Time and Accounting
  - Scheduling Overhead
- Conclusion

Jochen Liedtke
(* 26. Mai 1953;
† 10. Juni 2001)

Introduction

- Kernel is mandatory and common to all parts of the system
- Microkernel minimizes this part
- Serious attention in the late 90s
- Jochen Liedtke “On Microkernel Construction”
- Microkernels are following the design pattern of “separation of policy and mechanism”

Motivation

- Separation of policy and mechanism is not completely reached
- Scheduler implements a policy and is part of the kernel
- This makes scheduling more efficient
- But contradicts the microkernel idea and makes the system inflexible
- Today the need for user-level scheduling rises
- → A slightly loss in performance seams more acceptable

Difference: policy and mechanism

- Mechanisms are needed for the system to function
- Mechanisms are used to implement a policy
- Policies implement specific strategies
- Mechanisms describe what to do.
- Policies describe how to do it.

➢ Dispatcher implements a mechanism; Scheduler implements a policy

CPU Inheritance Scheduling

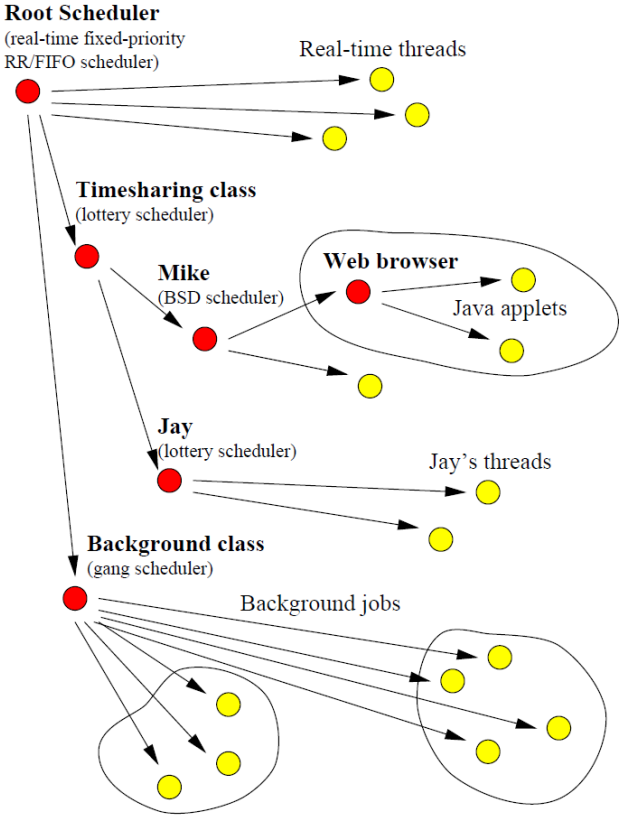
- Framework for user-level scheduling.
- Developed by Bryan Ford and Sai Susarla.
- System independent framework.
- A dispatcher must be provided by the underlying System.

CPU Inheritance Scheduling

- Dispatcher performs context switches.
- Dispatcher implements thread blocking, unblocking and CPU donation.
- Dispatcher has no notion of thread priority.

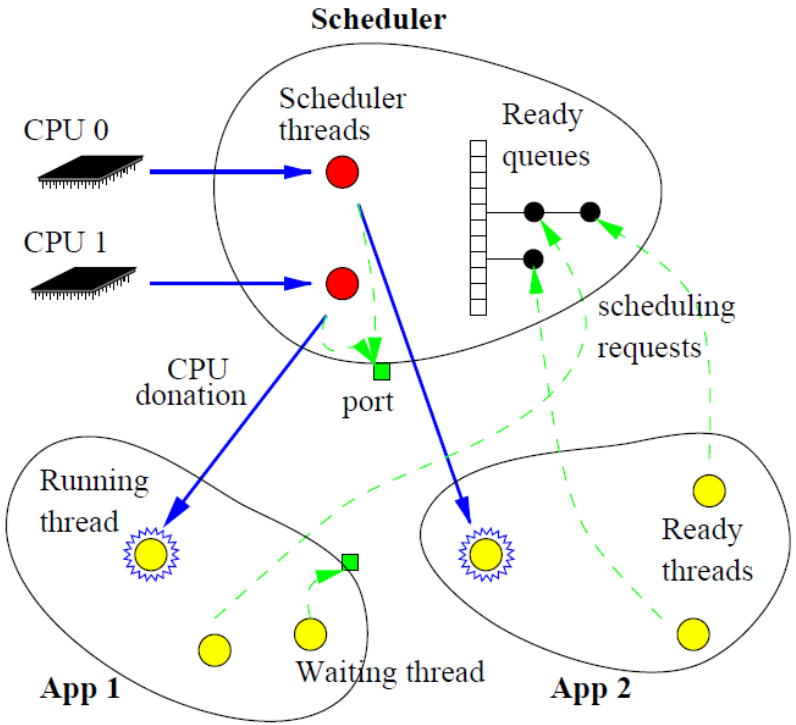
General approach

- Threads are acting as schedulers for other threads by donating CPU time
- By stacking threads it is possible to build a logical scheduling hierarchy
- Root scheduler with fixed-priority → can be used for scheduling real-time threads



General approach

- Works naturally for multi-core architectures
- Each CPU core is assigned to one root scheduler
- Fixes-priority scheduling for multi-core architecture possible
- Mutual ready queue from which both schedulers get threads assigned



Threads

- Defined as virtual CPUs, purpose to execute programs.
- Thread is running, if real CPU is assigned to it.
- Running threads can be preempted.
- Threads are managed by schedulers.

Threads

- Traditionally schduled in OS kernels, here: by other threads.
- Threads with real CPU assigned can donate CPU time.
- Root scheduler determines the base scheduling policy.
- Priority inversion happens automatically through CPU donation.

Requesting CPU Time

- Only the root scheduler has real CPU assigned to it.
- Other threads rely in CPU time donation.
- Each thread has a scheduler associated with them.
- When a thread awakens the responsible scheduler is notified.
- Scheduler can decide to give CPU time to the awoken thread or resume the preempted thread.

Requesting CPU Time

- If the scheduler has no CPU time left the next scheduler in the hierarchy will be woken.
- If the process reaches an already woken thread which is preempted it indicates that this event is irrelevant .

Time and Accounting

- Usually a notion of time and some kind of accounting is needed.
- Generally a periodic timer interrupt is sufficient to implement a dispatcher.
- Accounting is needed e.g. for billing customers or dynamically adjust scheduling policies.
- Two well known approaches exist.

Statistical CPU Accounting

- Scheduler has to wake up every tick.
- Checks running thread and charges the time quantum to this thread.
- Approach is considered very efficient, since a scheduler usually wakes almost every tick.
- But this approach provides only limited accuracy which is limited by the timer tick intervals.

Timestamp-based CPU Accounting

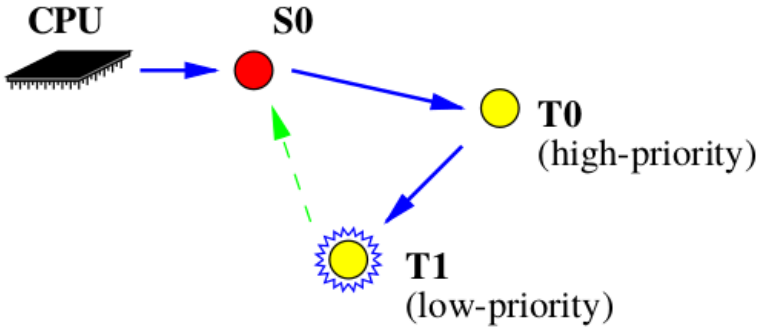
- The scheduler checks the timestamp with each context switch.
- It charges the thread for the exact amount of time.
- This approach provides a much higher accuracy.
- It also has a higher cost, because the context switch takes more time.
- Especially on systems on which it is expensive to read the current time.

Accounting problems and CPU Donation

- T0 has to wait for a resource by T1 and donates CPU time to T1.
- Whom to charge?

- It seems fair to charge T1.
- But since high priority threads are mostly more expensive, the scheduler has to charge T0 to avoid outsourcing.

- → “with privilege comes responsibility”



Scheduling Overhead

- Overhead caused by the dispatcher
  - Cost to compute the next thread depends on the depth of the hierarchy.
  - Concerns because there is no limitation.
  - The dispatcher is always the component with the highest priority → source for unbound priority inversion.
  - Possible solution for hard real-time systems is to limit the depth to 4-8 levels.

Scheduling Overhead

- Overhead caused by additional context switches
  - Additional context switches between scheduling threads.
  - Overhead mostly dependent on system design.
  - E.g. context switches on monolithic kernels are much higher then on microkernel, since context switches in the same address space is cheaper.
  - Also a chain of wake up calls can cause additional context switches, if the schedulers have no CPU time.

Conclusion

- CPU Inheritance Scheduling causes some scheduling overhead and therefore some performance loss.
- But the framework provides the flexibility that modern systems require.
- The loss of performance seems to be acceptable in order to gain more scheduling flexibility.
- However, the real performance loss has still to be determined in more practical tests.