Microkernel Construction

Real-Time Scheduling

Udo Steinberg

TU Dresden
Operating
Systems Group

Preemptive Priority-Based Scheduling

- User assigns each thread a set of scheduling attributes (time quantum, priority)
- When the current thread's time quantum is exhausted, the kernel selects a new current thread
- Scheduler always allocates the CPU to highest-priority thread that is ready to run at that time
- When a higher-priority thread becomes ready to run the kernel immediately preempts the current thread and switches to the higher-priority thread

Udo Steinberg

TU Dresden
Operating
Systems Group

Scheduling in Fiasco

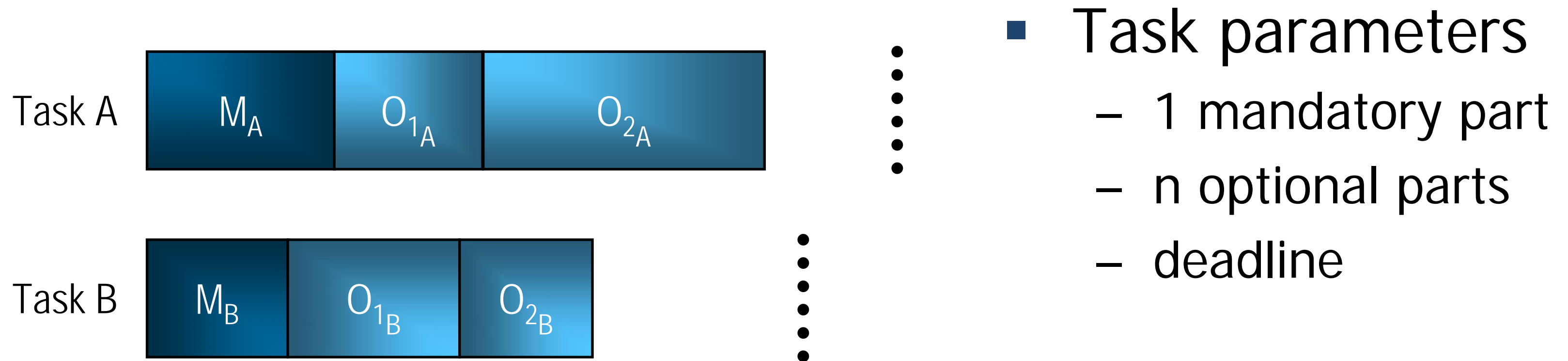
- Preemptive priority-based scheduling
- Round-robin scheduling among threads of the same priority level
- Priority inheritance via helping and IPC time donation mechanisms

- Quality-Assuring Scheduling with multiple time and priority reservations
  - strictly periodic threads
  - periodic threads with minimal inter-release times
  - periodic deadlines
  - preemption feedback mechanism (Preemption IPC)

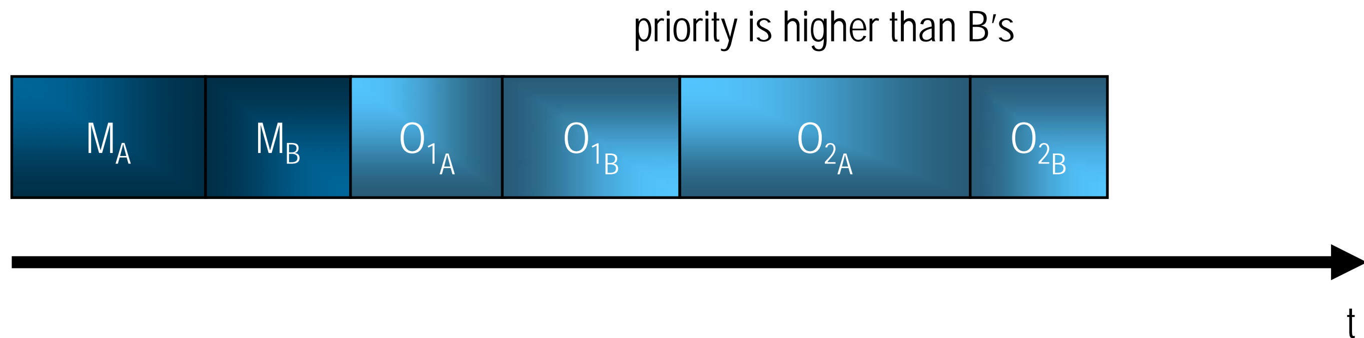
Udo Steinberg

TU Dresden
Operating
Systems Group

Quality-Assuring Scheduling



- Task parameters
  - 1 mandatory part
  - n optional parts
  - deadline



- QAS model supports imprecise computations

Microkernel Construction

Udo Steinberg

TU Dresden
Operating
Systems Group

Scheduling Context vs. Execution Context

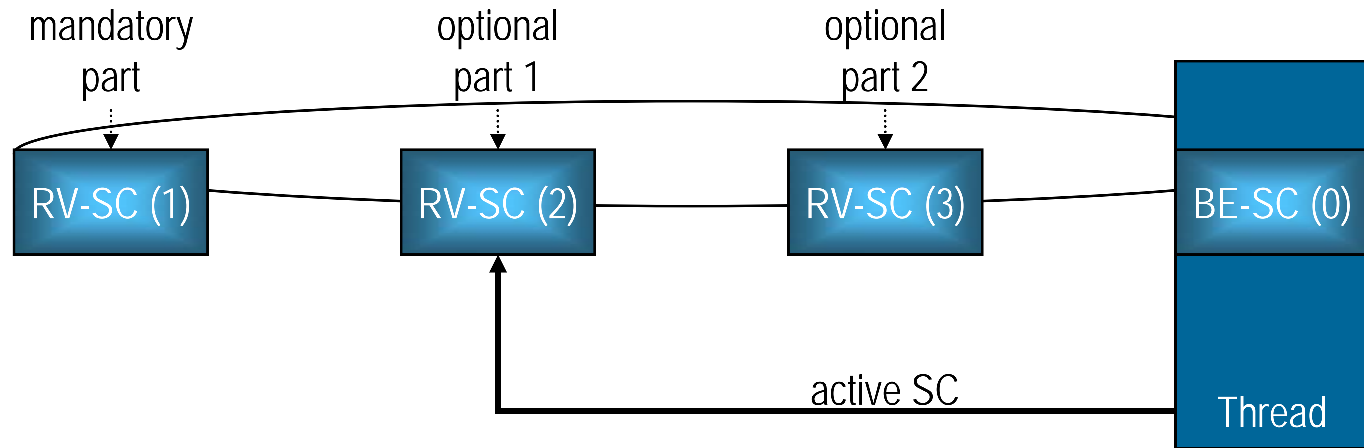
- Fiasco kernel implements:
  - Context/Thread (execution context)
  - Sched_context (scheduling context)

- Scheduling context is a time quantum coupled with a priority and comprises
  - owner pointer (execution context/thread owning this SC)
  - thread-local and sequential identification number (id)
  - priority (0 … 255)
  - total time quantum (in µs)
  - remaining time quantum (in µs)
  - preemption info (time and type of last preemption event)
  - sibling pointers (prev, next)

Udo Steinberg

TU Dresden
Operating
Systems Group

List of Scheduling Contexts



- Each execution context (thread) aggregates a **best-effort scheduling context** with default parameters in its thread control block (TCB)
- Optionally more **reservation scheduling contexts** with different time quanta/priorities can be dynamically assigned to the thread
- The scheduling context whose time the thread consumes is the **active scheduling context**

Udo Steinberg

TU Dresden
Operating
Systems Group

Scheduling System-Calls

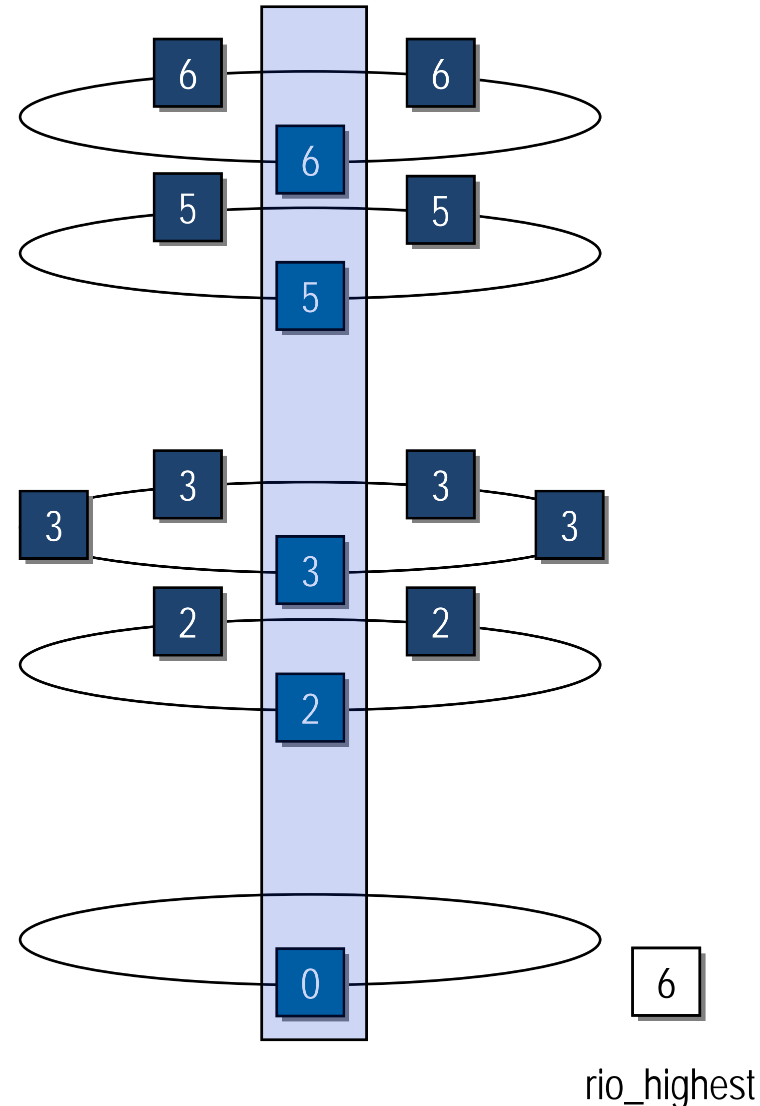
- l4_thread_schedule (get/set scheduling parameters)
  - Begin periodic execution (rt_begin_periodic)
  - End periodic execution (rt_end_periodic)
  - Add new scheduling context (rt_add)
  - Remove all scheduling contexts (rt_remove)
  - Set period length (rt_period)
- l4_thread_switch (yield current scheduling context)
  - Activate next scheduling context (rt_next_reservation)
- l4_ipc (synchronous message transfer)
  - Wait for beginning of next period (rt_next_period)

Udo Steinberg

TU Dresden
Operating
Systems Group

Ready List

- L4 supports 256 priorities
  - One ring for each priority
- Priority ring is doubly-linked circular list of ready threads
  - Formed via TCB links ready_next, ready_prev
- Rapid access to next thread for each priority
  - prio_next[256] array
- Rapid access to highest populated priority ring
  - prio_highest variable



Microkernel Construction

Udo Steinberg

TU Dresden
Operating
Systems Group

Enqueue/Dequeue Operation

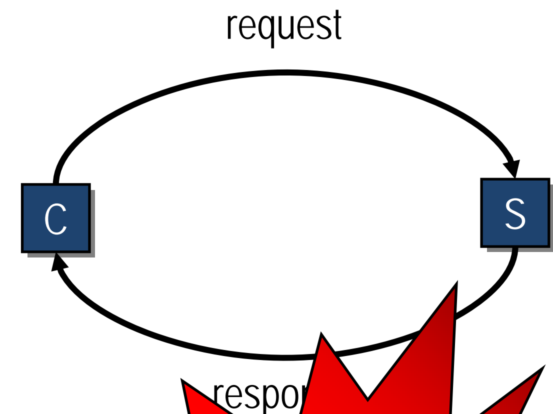
- ready_enqueue()
  - enqueue thread in ready list according to priority of thread's active scheduling context
  - boost prio_highest if thread's priority is higher
  - enqueue before prio_next[prio] or establish new ring
  - complexity always O(1)
- ready_dequeue()
  - dequeue thread from ready list if enqueued
  - if thread is prio_next[prio] rotate or disband ring
  - if thread was last of prio_highest, recompute prio_highest
  - dequeue last thread of highest priority always O(n)
  - dequeue any other thread always O(1)
- switch_sched()
  - Switch thread's active scheduling context
  - Dequeue from old ring, enqueue according to new priority

Udo Steinberg

TU Dresden
Operating
Systems Group

Typical IPC Scenario

- Initially
  - Client ready and enqueued
  - Server blocked and dequeued
- Client sends request and waits for response
  - Dequeue client
  - Enqueue server
- Server sends response and waits for next request
  - Dequeue server
  - Enqueue client
- Client receives the response
  - Same situation as initially



Lazy Queueing

- Observations
  - Current thread is always ready
  - ➢ Current thread need not be enqueued in ready list
  - ➢ … but must be enqueued when switching to another thread

  - Threads that block may become ready again before the invocation of the scheduler
  - ➢ Keep blocked threads in the ready list
  - ➢ … but let the scheduler kick them out if they are still blocked

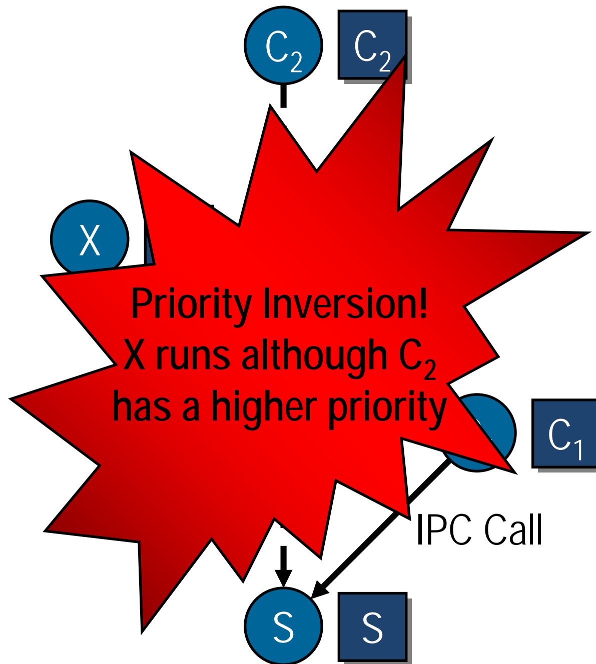
- Invariant: All ready threads must be enqueued in the ready list, except for the currently executing thread. Blocked threads may remain in the ready list until the next time the scheduler runs

Udo Steinberg

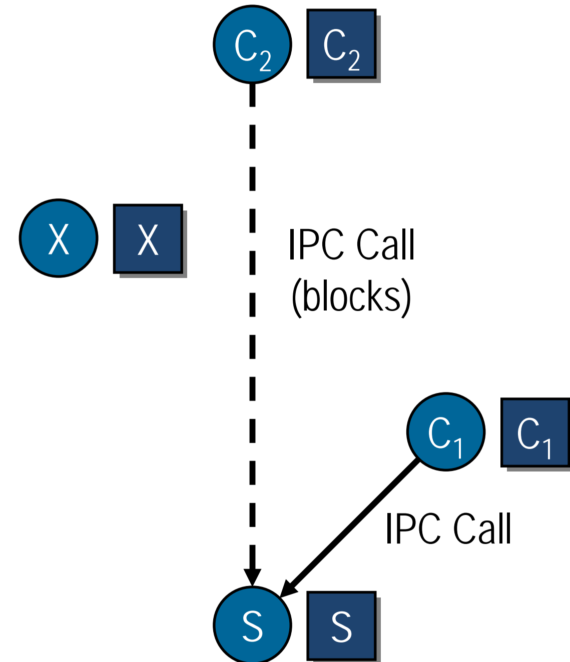
TU Dresden
Operating
Systems Group

IPC and Scheduling

- Busy Destination Thread



- Solution: Time Donation



- Time Donation implements Priority Inheritance
- Priority of a thread is always the priority of the scheduling context it executes on

Udo Steinberg

TU Dresden
Operating
Systems Group

Scheduling and Context Switching

- schedule() – pick next thread to run
  - find highest-priority thread in ready list that is ready to run, thread = prio_next[prio_highest]
  - dequeue blocked threads from ready list when found
  - call switch_to (thread)

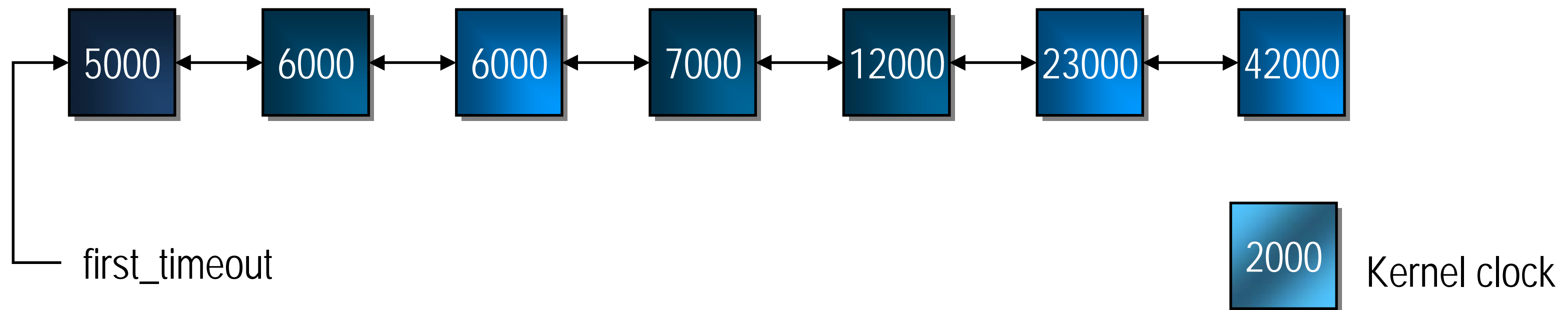
- switch_to() – switch scheduling & execution context
  - switch to thread's active scheduling context which becomes the current scheduling context - **current_sched()**
  - call switch_exec (thread)

- switch_exec() – switch execution context only
  - switch FPU
  - switch CPU, thread becomes current thread - **current()**

Udo Steinberg

TU Dresden Operating Systems Group

Timeout List



- Timeouts kept in doubly-linked list sorted by ascending wakeup time - list head is first_timeout
- Each timeout is associated with an owner object
- Kernel calls virtual function Timeout::expired() when kernel clock >= wakeup time of the timeout
- Currently there are 3 different timeout types

Udo Steinberg

TU Dresden
Operating
Systems Group

IPC Timeout

- Expires when IPC partner does not rendezvous within time limit (also used for page-fault IPC)
- Timeout values 0 and infinite do not use IPC timeouts
- IPC timeout is allocated on kernel stack of thread involved in the IPC

- When IPC timeout expires:
  - Stop IPC for thread to which IPC timeout belongs (owner)
  - Enqueue owner in ready list

  - Trigger schedule() if thread's active scheduling context has a higher priority than the current scheduling context

Udo Steinberg

TU Dresden
Operating
Systems Group

Timeslice Timeouts

- Expires when time quantum on current scheduling context is depleted
- One timeslice timeout per CPU (statically allocated)

- When timeslice timeout expires...
  - on a reservation scheduling context:
    - send "Timeslice Overrun" Preemption-IPC to preempter
    - switch to owner's next scheduling context
  - on a best-effort scheduling context:
    - refill and remain on owner's best-effort scheduling context

  - Trigger schedule() to select new current scheduling context

Udo Steinberg

TU Dresden
Operating
Systems Group

Deadline Timeouts

- Expires when thread's periodic deadline occurs
- One deadline timeout for each periodic thread

- When deadline timeout expires...
  - if the thread is not waiting for the begin of the next period
    - send "Deadline Miss" Preemption-IPC to preempter

  - program new deadline timeout (end of next period)
  - switch to thread's first reservation scheduling context
  - potentially wake thread up

  - Trigger schedule() if thread's new active scheduling context has a higher priority than the current scheduling context

Udo Steinberg

TU Dresden
Operating
Systems Group

Timer Interrupt

- Uses a hardware timer source (PIT, RTC, Local APIC)
- Drives timeout infrastructure and 64 bit kernel clock
- Timer-Interrupt Handler:
  - acknowledges timer interrupt at PIC/Local APIC
  - updates kernel clock
  - checks for expired timeouts
  - invokes schedule() if a woken-up thread has a higher priority than the current thread
- Different implementations of timer interrupt
  - periodic
    - interrupt occurs periodically, i.e. every 1 ms
    - timer hardware programmed once at system startup
  - one-shot (aperiodic)
    - interrupt programmed to wakeup time of first_timeout
    - requires frequent reprogramming of timer hardware

Hardware Timer-Interrupt Sources

Timer Source	Period	Granularity	Mode	Handler Overhead (P4 / Athlon)
PIT	1000 µs	1000 µs	Periodic	1484 / 1343 clk 928 ns / 1.7 µs
RTC	976 µs	976 µs	Periodic	22096 / 18861 clk 14 µs / 23.6 µs
Local APIC	1000 µs	1000 µs	Periodic	272 / 40 clk 170 ns / 50 ns
Local APIC	N/A	1 µs	One-Shot	816 / 200 clk 510 ns / 250 ns

Preemption IPC

- Notification sent by kernel to preempter of a thread
- Preemption event stored inside scheduling context
- Later events overwrite earlier events

- Deadline Miss
  - Thread missed its periodic deadline
  - Thread not in "wait for next period" state when deadline timeout expires

- Timeslice Overrun
  - Time quantum on thread's reservation scheduling context depleted
  - Not sent for best-effort scheduling contexts (ID = 0)

Preemption IPC: Details

- Short-IPC (register-only message transfer)
- Timestamp of preemption event (56 bits)
- Identification number of scheduling context on which the event occurred (6 bits)
- „Lost“-Bit (1 bit), set if previous events were lost
- Type of preemption event (1 bit)
  - Deadline Miss (0)
  - Timeslice Overrun (1)

Udo Steinberg

TU Dresden
Operating
Systems Group