WINGERT
A Thread Migrating OS for Real-Time Applications

Alexander Züpke

alexander.zuepke@hs-rm.de

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

About Me

Alexander Züpke

1999 – 2003: Diploma in Computer Engineering

University of Applied Sciences Gelsenkirchen

2003 – now: Kernel Hacker on PikeOS

SYSGO AG, Klein-Winternheim

2012 – now: PhD Student

RheinMain University of Applied Sciences Wiesbaden

Wingert

Win·gert m., Pl: Win·ger·te

German word in Rhine-Hessian dialect for a *vineyard*

derived from the Middle High German word

wîngarte

(wine garden)

Wingert OS

WIesbaden

Next

Generation

Experimental

Real-**T**ime

Operating

System

… or: **WI**NGERT **I**s a **N**ew **G**reat **E**xperimental **R**eal-**T**ime **O**perating **S**ystem

Outline

- Motivation
- System Architecture
- Various Use Cases of Thread Migration
- Resource Management
- Futexes and Locking
- Current Status of the Implementation
- Conclusion
- Outlook

Motivation

Safety Critical Systems …

Linux kernel map

2.6.36

functionalities
layers

human interface | system | processing | memory | storage | networking

user space interfaces
system calls and system files

virtual

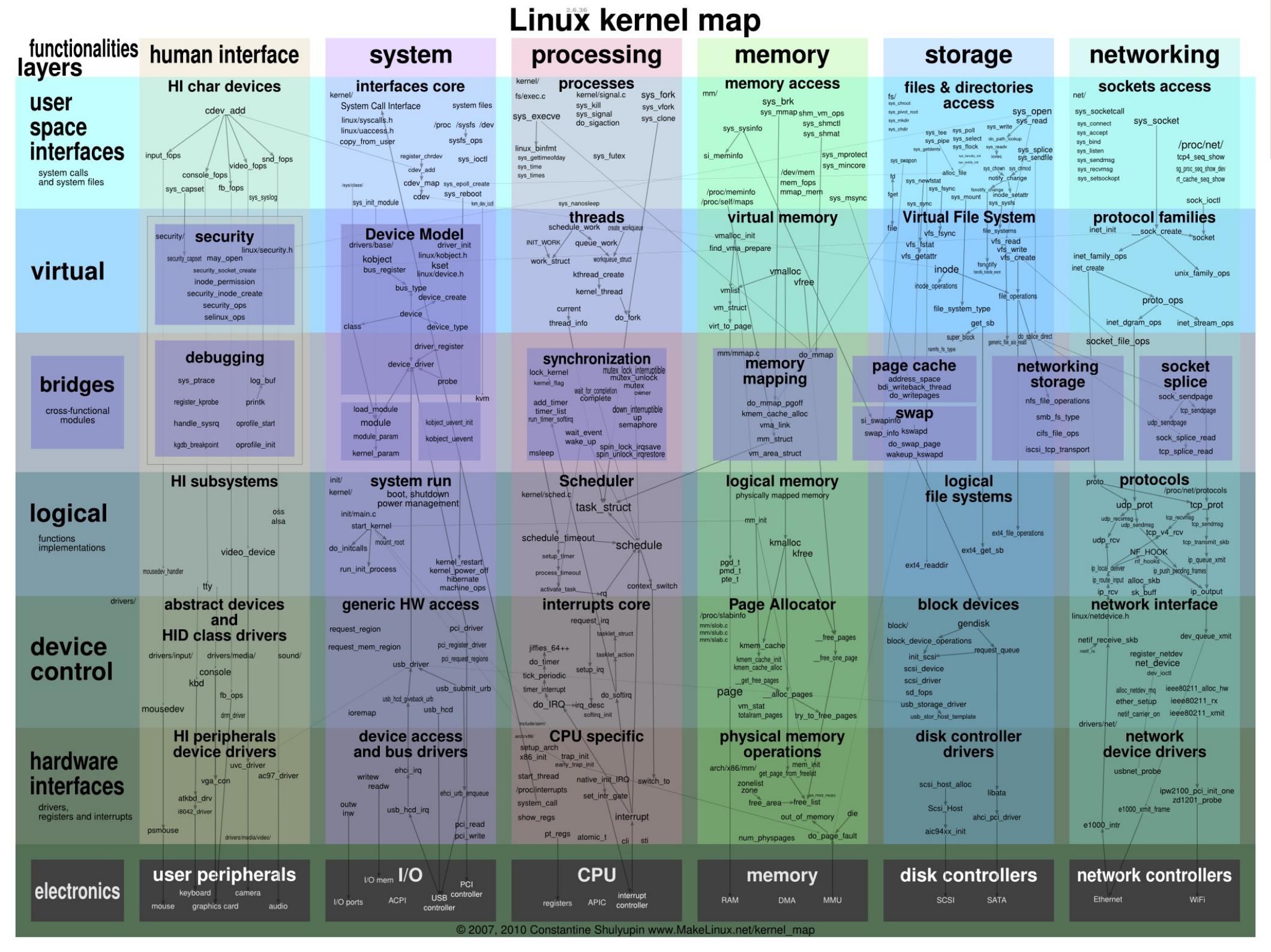
bridges
cross-functional modules

logical
functions implementations

device control

hardware interfaces
drivers, registers and interrupts

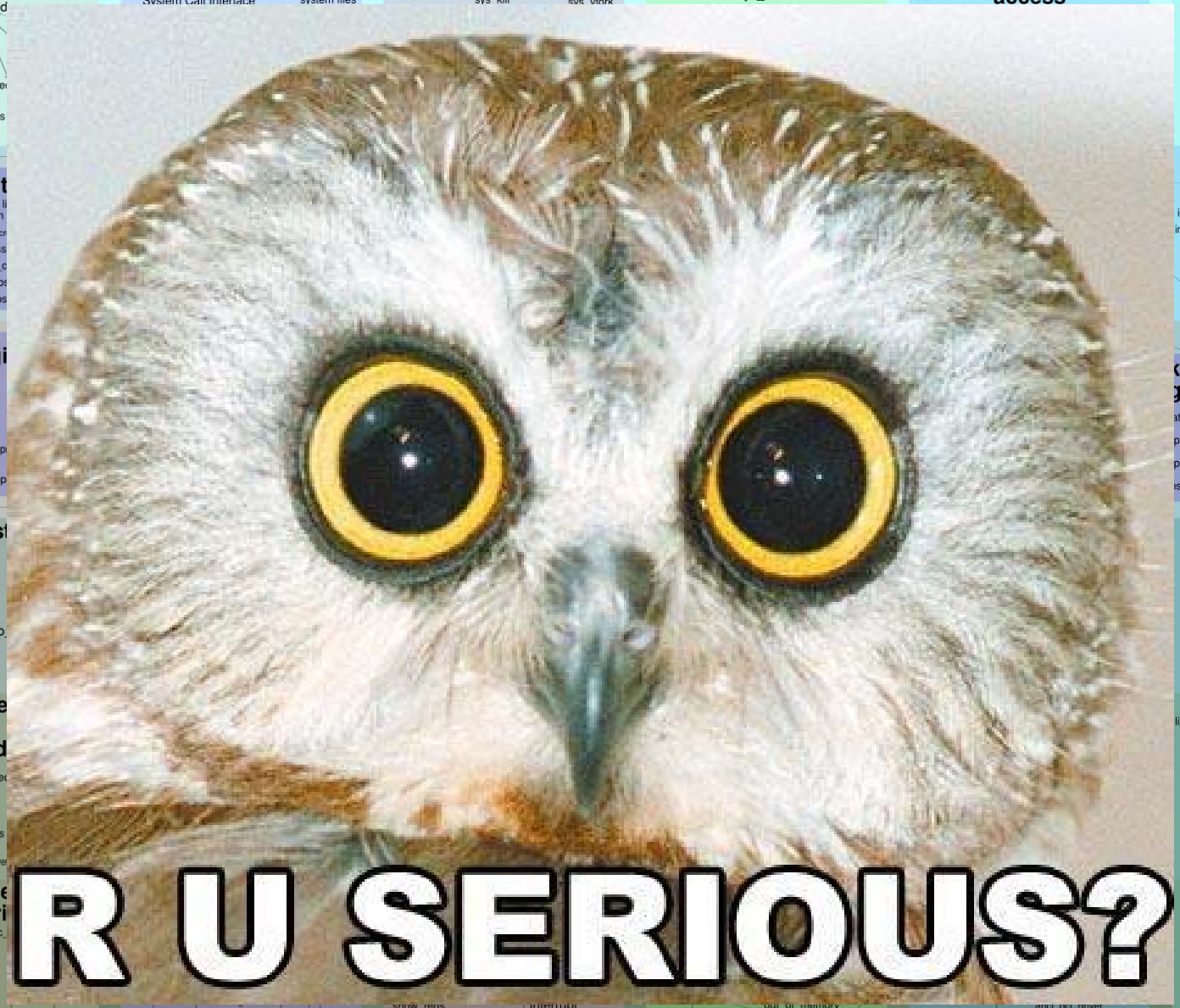
electronics



Linux kernel map

2.6.36

functionalities / layers	human interface	system	processing	memory	storage	networking
user space interfaces system calls and system files	HI char devices	interfaces core	processes	memory access	files & directories access	sockets access
virtual						protocol families
bridges cross-functional modules						socket splice
logical functions implementations	HI subsyst					protocols
device control	abstract de and HID class d					network interface
hardware interfaces drivers, registers and interrupts	HI periphe device dri					network device drivers
electronics	user peripherals	I/O	CPU	memory	disk controllers	network controllers



Motivation

Safety requirements for shared resources

- IEC 61508

  “An E/E/PE* safety-related system will usually implement more than one safety function. If the safety integrity requirements for these safety functions differ, unless there is sufficient independence of implementation between them, the requirements applicable to the highest relevant safety integrity level shall apply to the entire E/E/PE safety-related system.”

- ISO 26262

  “Freedom of interference”

* E/E/PE: electrical / electronic / programmable electronic

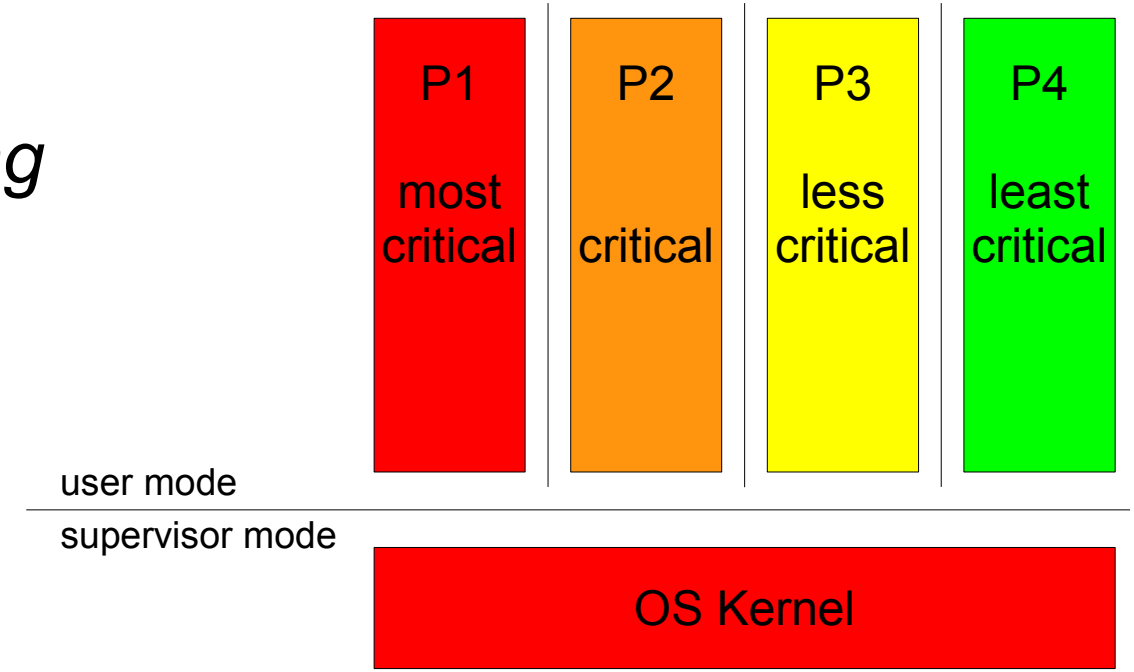
Motivation

Mixed-criticality system

Separation by Partitioning

ARINC 653:

- *Spatial Partitioning*
- *Time Partitioning*



Motivation

Own Experience:

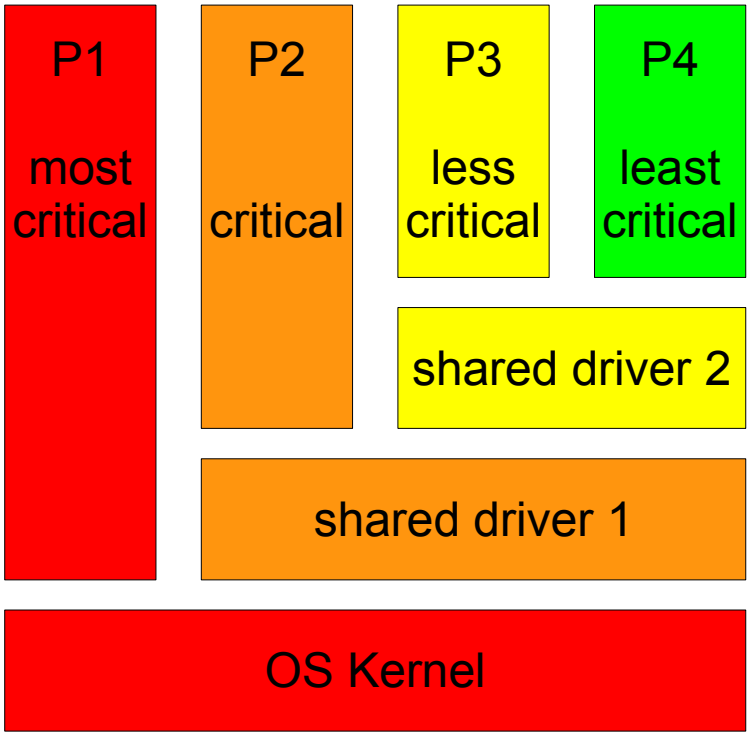
Micro kernels are nice, but building reliable systems with them is still too painful!

Lots of engineering challenges:

- bounded WCET when Linux runs on top?
- independent analyses of partitions?
- more threads + more synchronization = more safety?

System Architecture

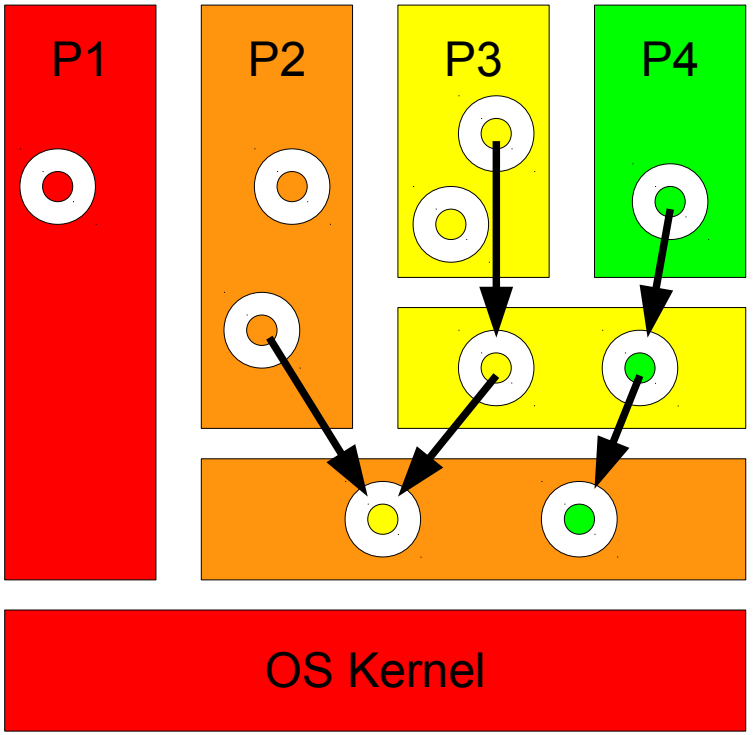
- Design Choices
  - Hierarchical system design
  - Small TCB
  - Minimalistic kernel
    - Address spaces
    - Threads
    - Capabilities
    - Resource partitioning
    - Preemptive kernel
    - State of the art scheduling
  - Thread migration



Thread Migration

Thread Migration

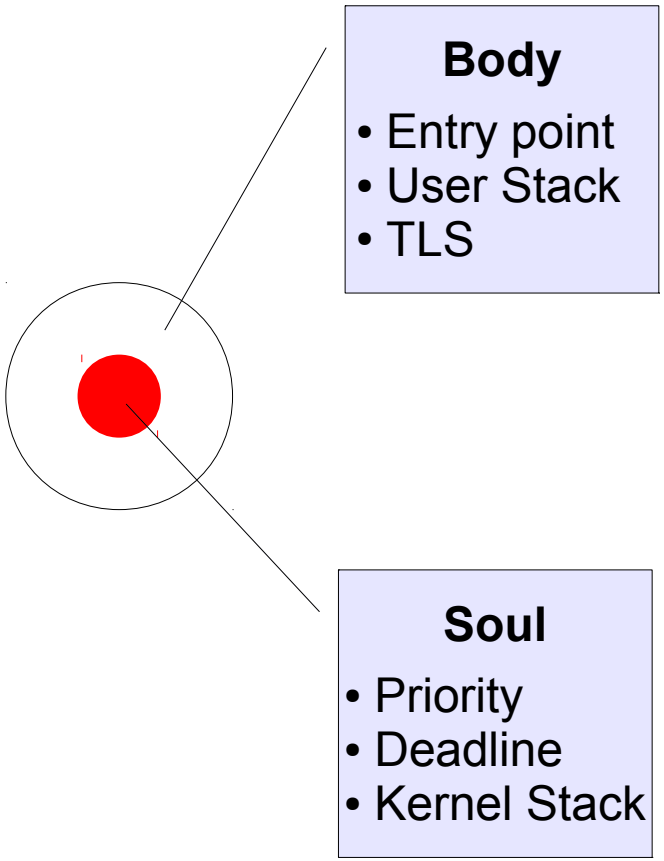
- Definition of “thread migration” in literature
  - a client lends its thread to the server
  - the server is a passive entity
- Examples
  - Mach (Ford)
  - Sun's Spring
  - Pebble
  - Composite
  - …



Thread Migration

Decompose a thread into ***Body and Soul:***

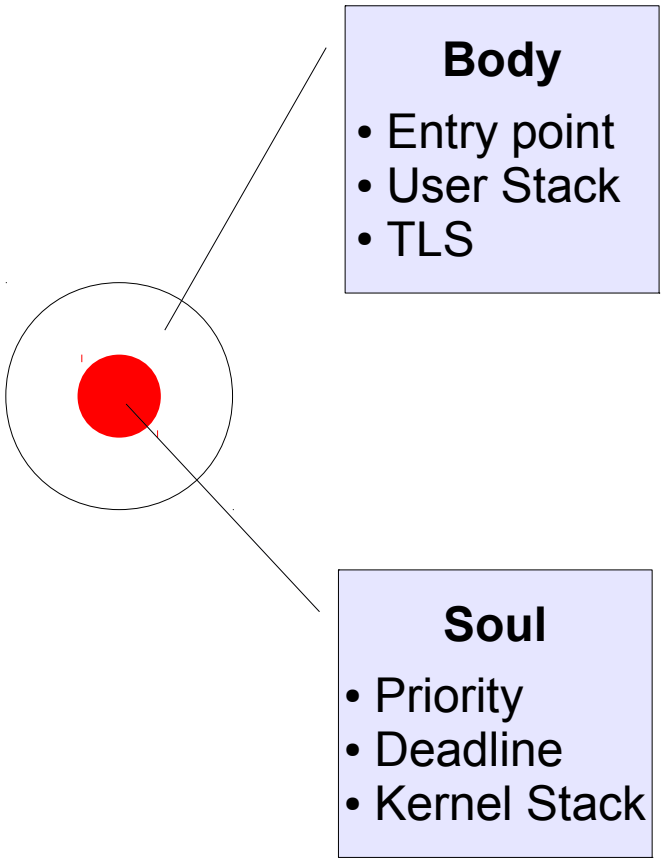
- Body: user part of a thread
  - register context
  - user stack
- Soul: kernel part of a thread
  - scheduling attributes
  - kernel stack



Thread Migration

Decompose a thread into ***Body and Soul:***

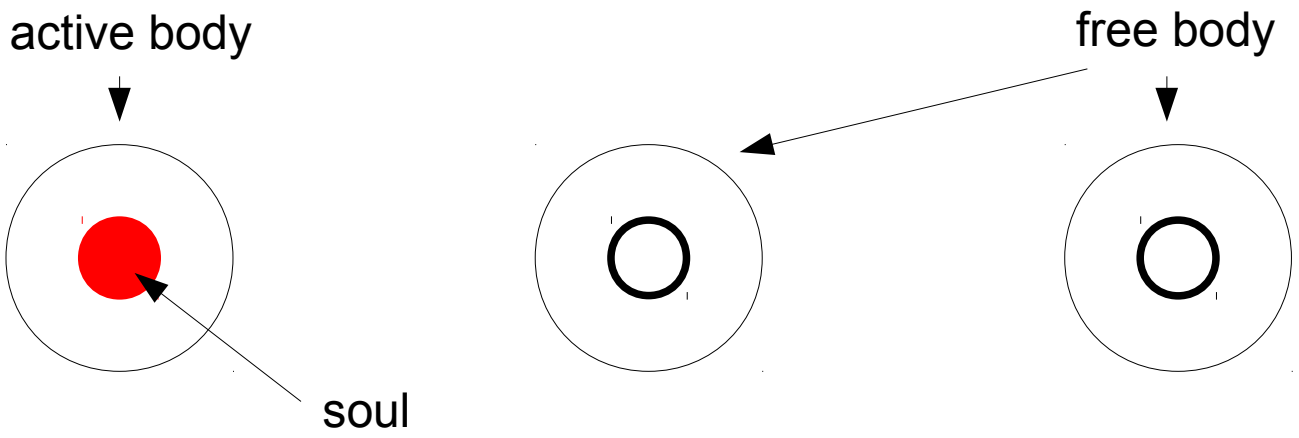
- Body: user part of a thread
  - register context
  - user stack
- Soul: kernel part of a thread
  - scheduling attributes
  - kernel stack
- Ghost: soul without a body
  - initial state
  - idle threads



Thread Migration

Synchronous ***call and return*** operations:

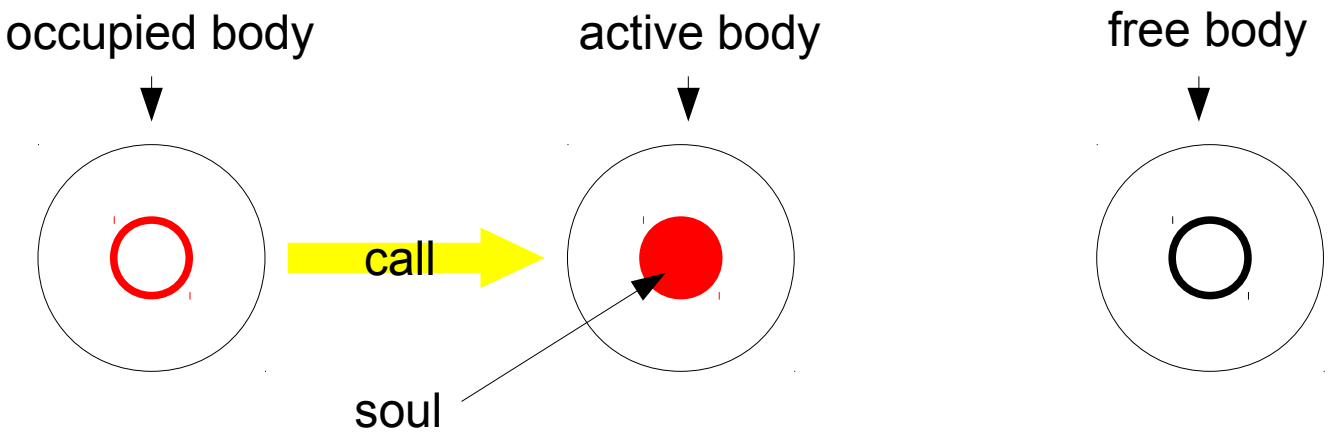
- a soul migrates back and forth between bodies
- forms a ***call chain***



Thread Migration

Synchronous ***call and return*** operations:

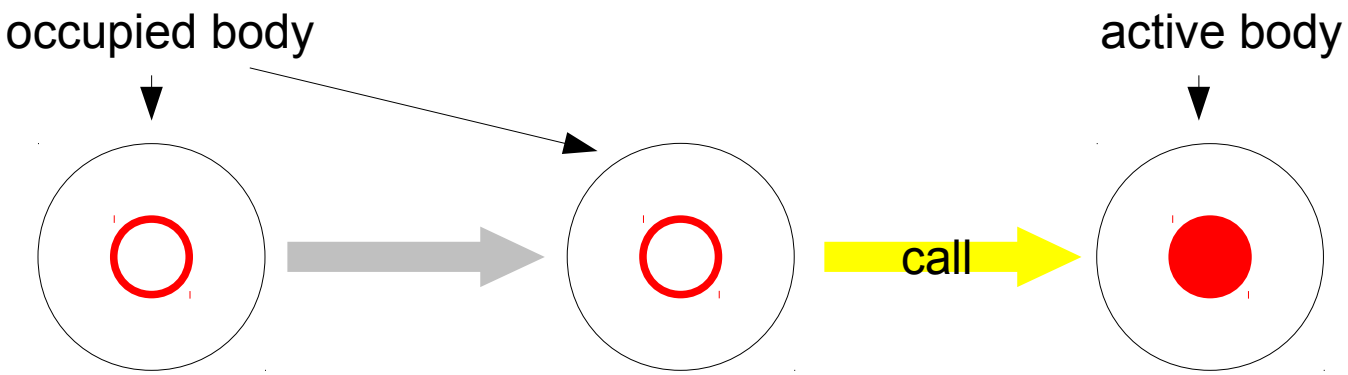
- a soul migrates back and forth between bodies
- forms a ***call chain***



Thread Migration

Synchronous ***call and return*** operations:

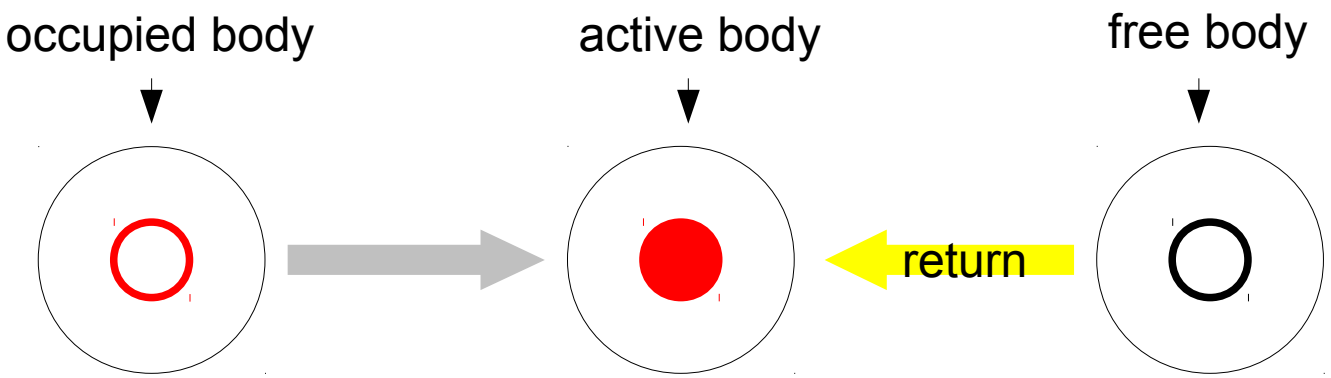
- a soul migrates back and forth between bodies
- forms a ***call chain***



Thread Migration

Synchronous ***call and return*** operations:

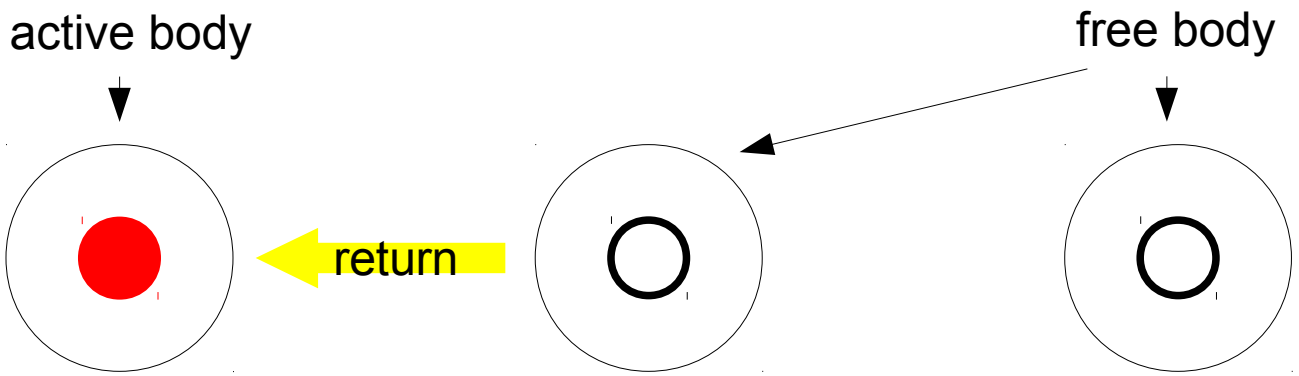
- a soul migrates back and forth between bodies
- forms a ***call chain***



Thread Migration

Synchronous ***call and return*** operations:

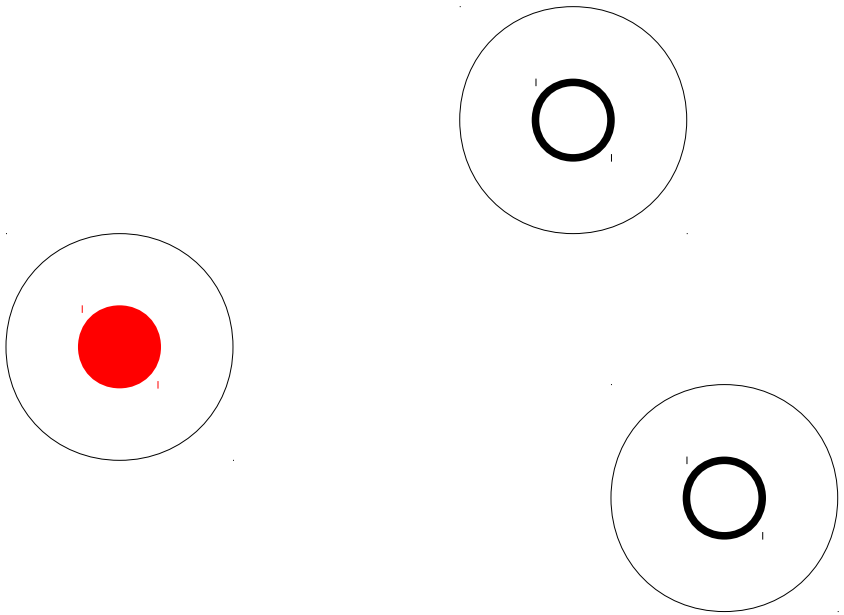
- a soul migrates back and forth between bodies
- forms a ***call chain***



Thread Migration

The ***forward*** operation:

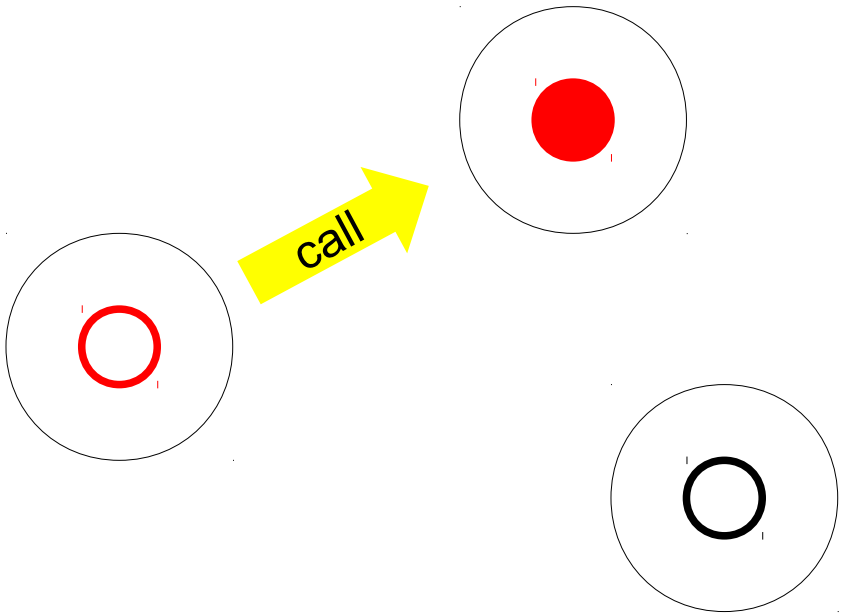
- to call another body
- without keeping the caller occupied



Thread Migration

The ***forward*** operation:

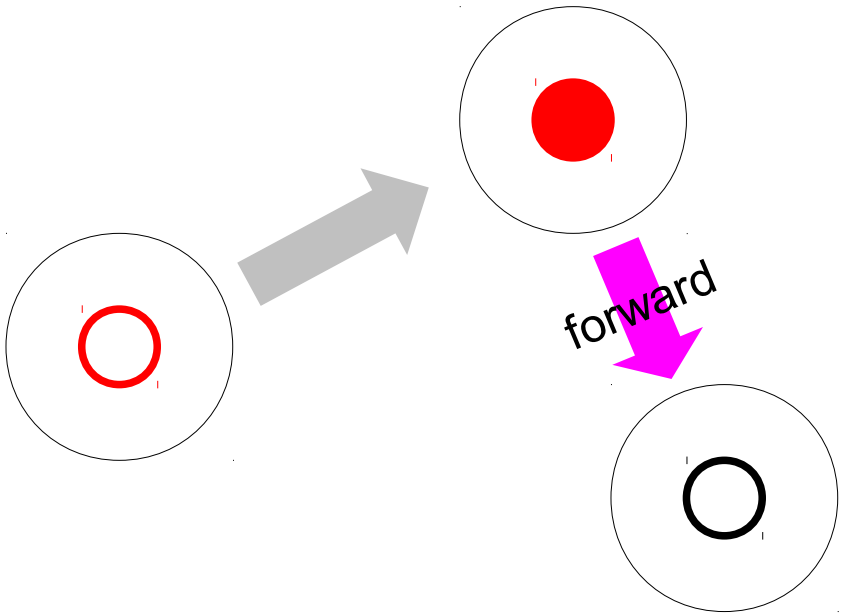
- to call another body
- without keeping the caller occupied



Thread Migration

The ***forward*** operation:

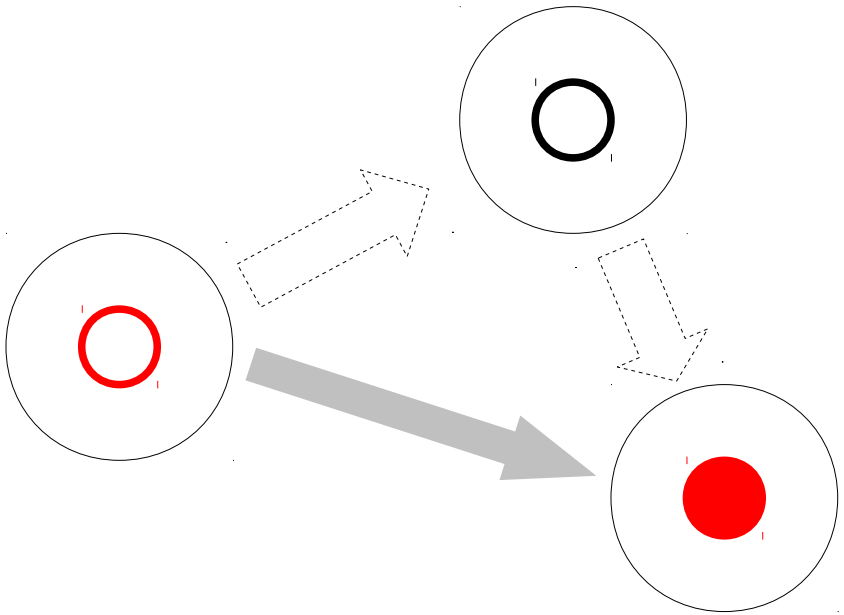
- to call another body
- without keeping the caller occupied



Thread Migration

The ***forward*** operation:

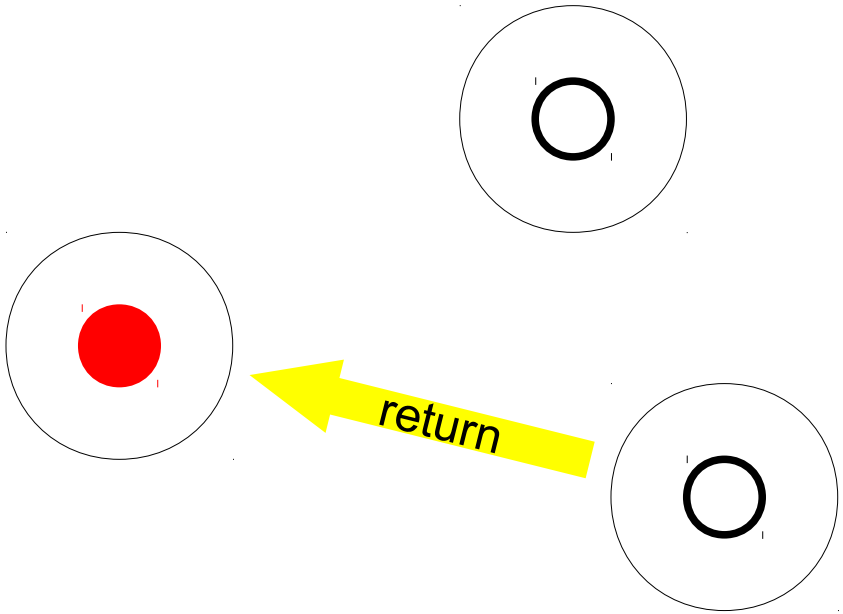
- to call another body
- without keeping the caller occupied



Thread Migration

The ***forward*** operation:

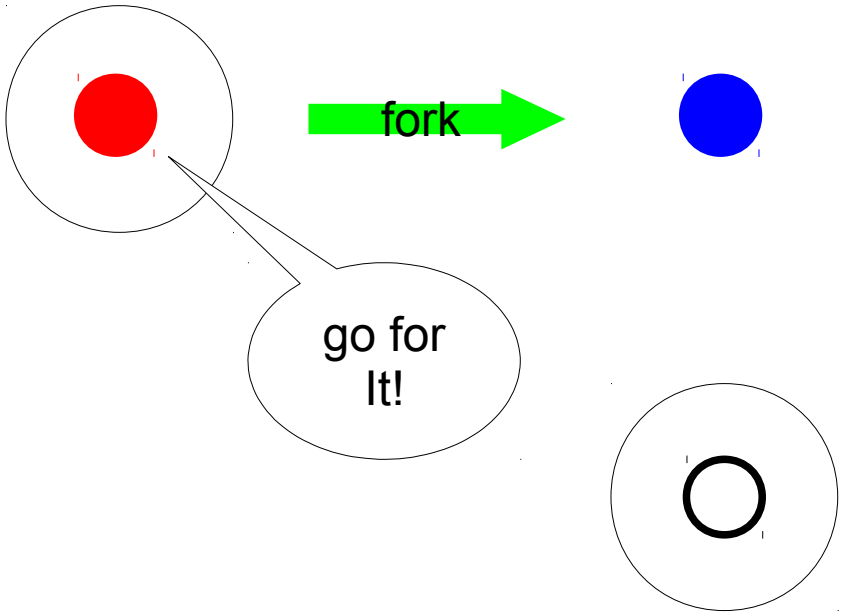
- to call another body
- without keeping the caller occupied



Thread Migration

Asynchronous ***fork / join*** operations:

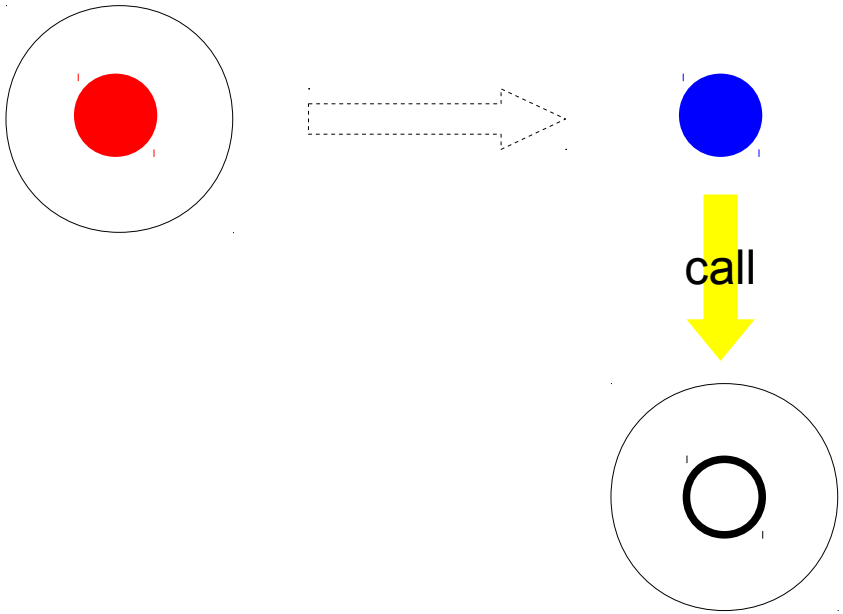
- fork: tell an idle soul to call a body
- join: asynchronous call returns



Thread Migration

Asynchronous ***fork / join*** operations:

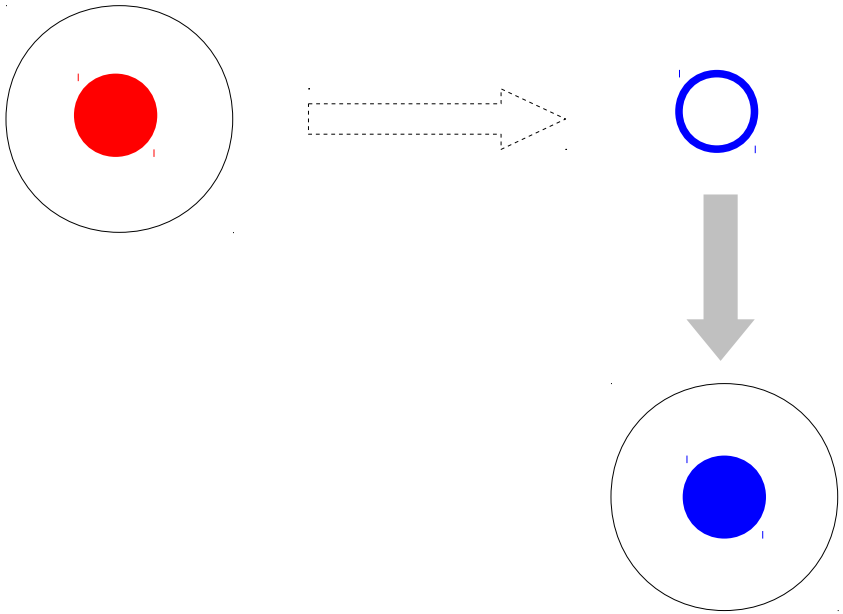
- fork: tell an idle soul to call a body
- join: asynchronous call returns



Thread Migration

Asynchronous ***fork / join*** operations:

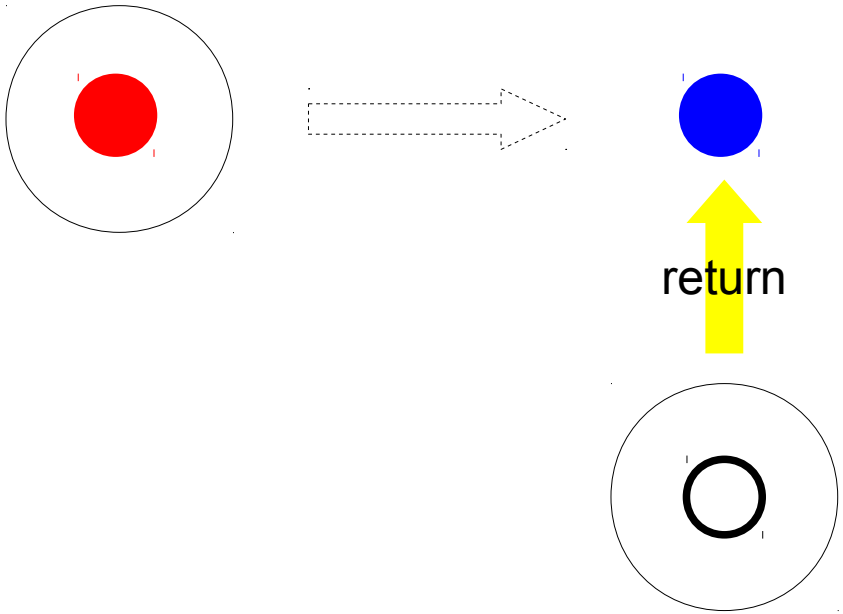
- fork: tell an idle soul to call a body
- join: asynchronous call returns



Thread Migration

Asynchronous ***fork / join*** operations:

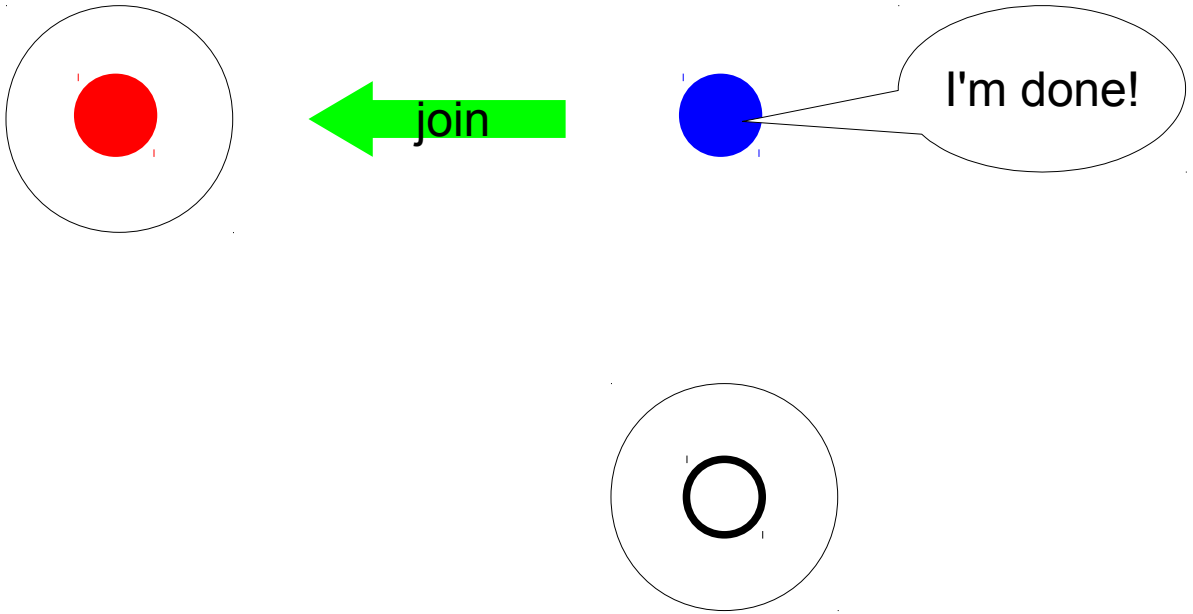
- fork: tell an idle soul to call a body
- join: asynchronous call returns



Thread Migration

Asynchronous ***fork / join*** operations:

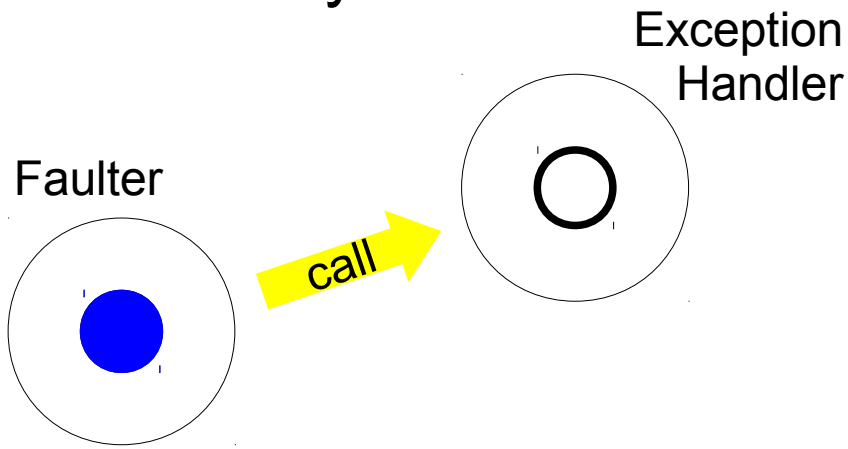
- fork: tell an idle soul to call a body
- join: asynchronous call returns



Thread Migration

Signals and Exception Handling:

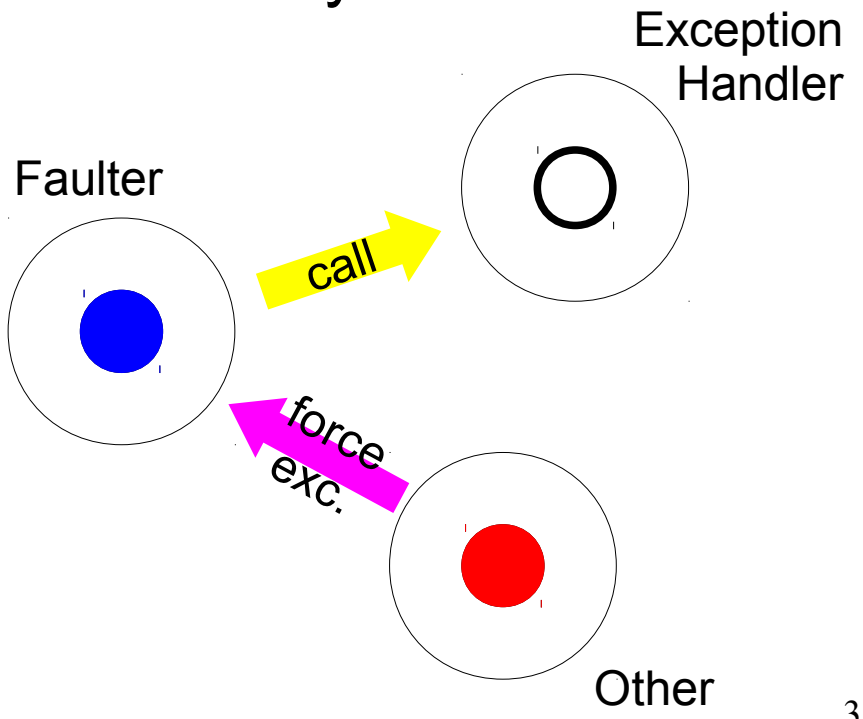
- Exceptions
  - implicitly turn exceptions into calls to exception-handlers
  - pass faulting register context to called body



Thread Migration

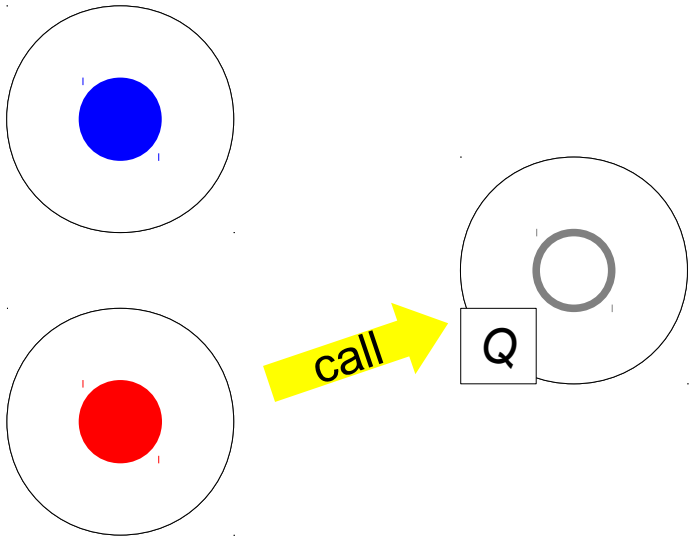
Signals and Exception Handling:

- Exceptions
  - implicitly turn exceptions into calls to exception-handlers
  - pass faulting register context to called body
- Signals
  - signal delivery: force a soul into a (non-voluntary) call
  - software raised exception



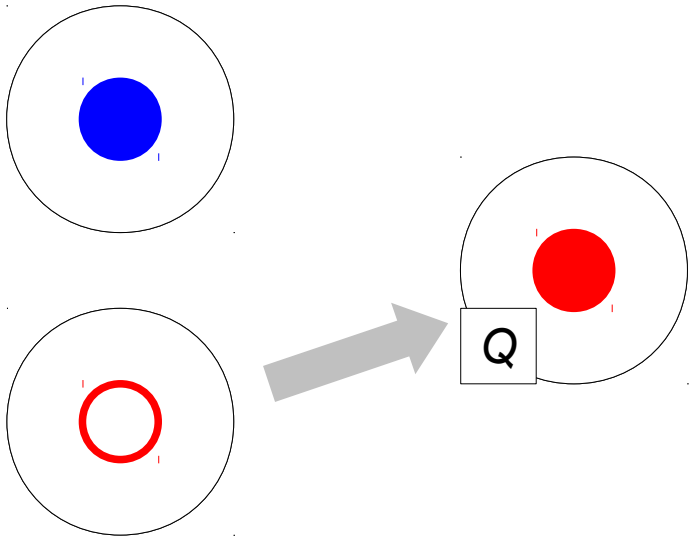
Concurrent Access

- Two souls want to enter the same body …
  - First come, first serve! The other soul has to wait.
  - Entry wait queue ***Q***
  - FIFO or priority ordering
  - Priority inheritance



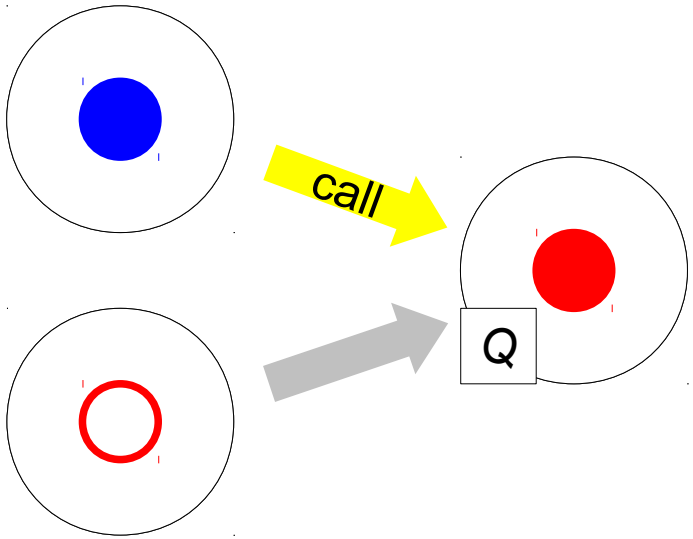
Concurrent Access

- Two souls want to enter the same body …
  - First come, first serve! The other soul has to wait.
  - Entry wait queue ***Q***
  - FIFO or priority ordering
  - Priority inheritance



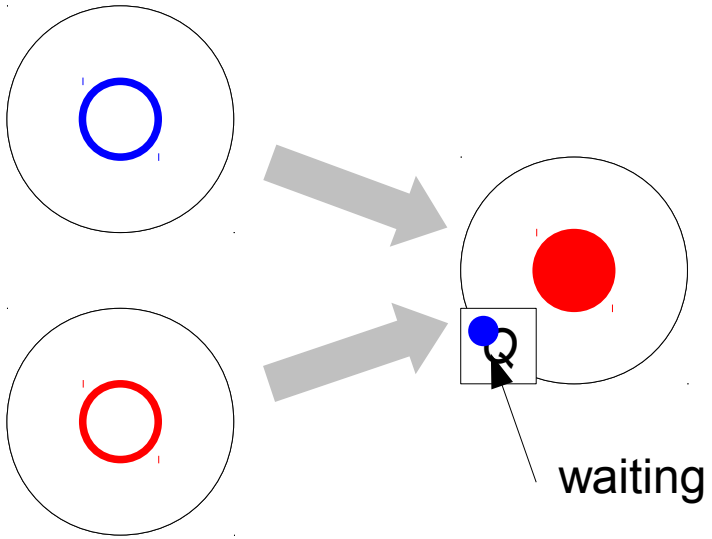
Concurrent Access

- Two souls want to enter the same body …
  - First come, first serve! The other soul has to wait.
  - Entry wait queue ***Q***
  - FIFO or priority ordering
  - Priority inheritance



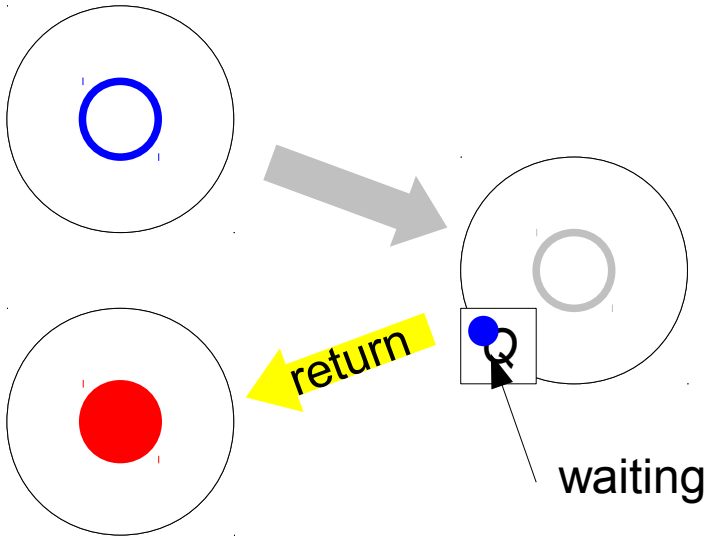
Concurrent Access

- Two souls want to enter the same body …
  - First come, first serve! The other soul has to wait.
  - Entry wait queue ***Q***
  - FIFO or priority ordering
  - Priority inheritance



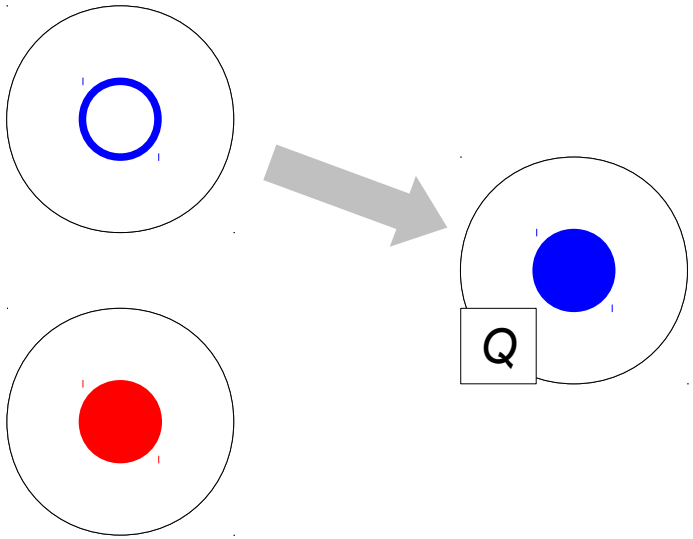
Concurrent Access

- Two souls want to enter the same body …
  - First come, first serve! The other soul has to wait.
  - Entry wait queue ***Q***
  - FIFO or priority ordering
  - Priority inheritance



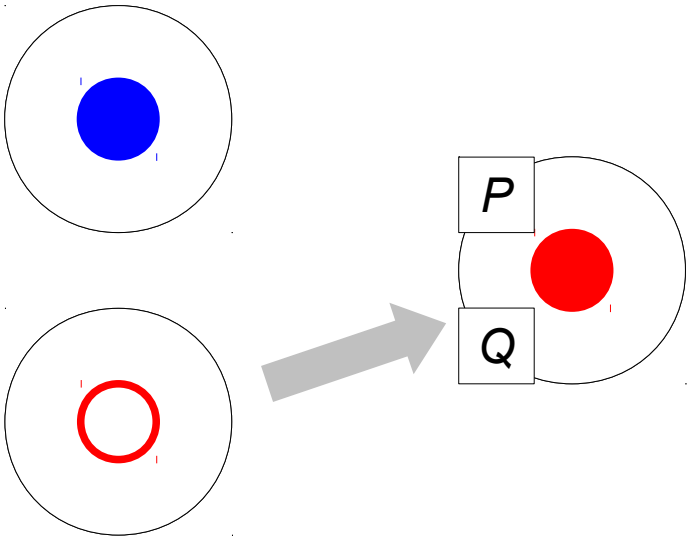
Concurrent Access

- Two souls want to enter the same body …
  - First come, first serve! The other soul has to wait.
  - Entry wait queue ***Q***
  - FIFO or priority ordering
  - Priority inheritance



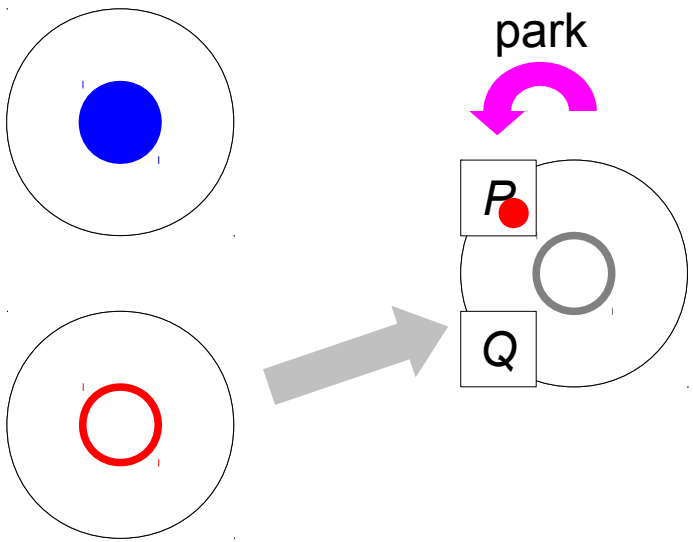
Parking Souls

- The Parking Lot:
  - a place where souls can rest ***outside*** the body
  - parking: put the currently active soul in ***P***
  - unparking: move a parked soul from ***P*** to ***Q***
  - the parking queue P is an unsorted queue



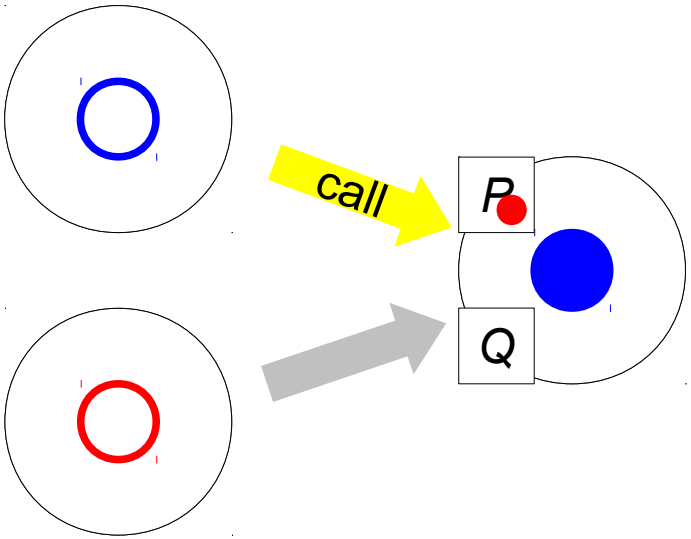
Parking Souls

- The Parking Lot:
  - a place where souls can rest ***outside*** the body
  - parking: put the currently active soul in ***P***
  - unparking: move a parked soul from ***P*** to ***Q***
  - the parking queue P is an unsorted queue



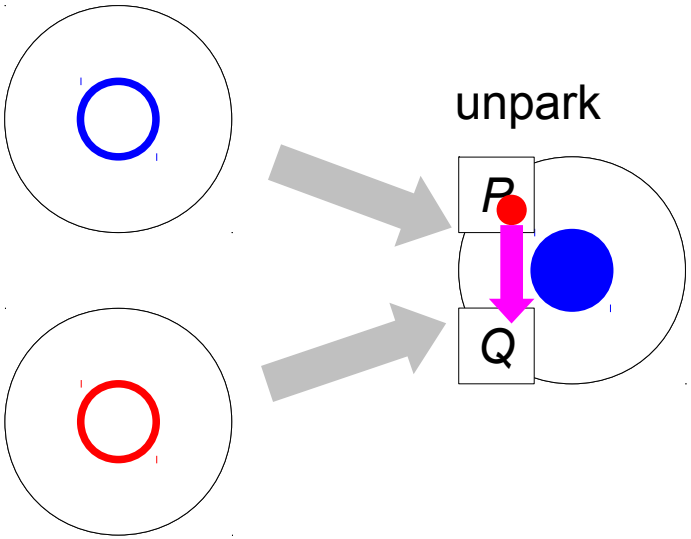
Parking Souls

- The Parking Lot:
  - a place where souls can rest ***outside*** the body
  - parking: put the currently active soul in ***P***
  - unparking: move a parked soul from ***P*** to ***Q***
  - the parking queue P is an unsorted queue



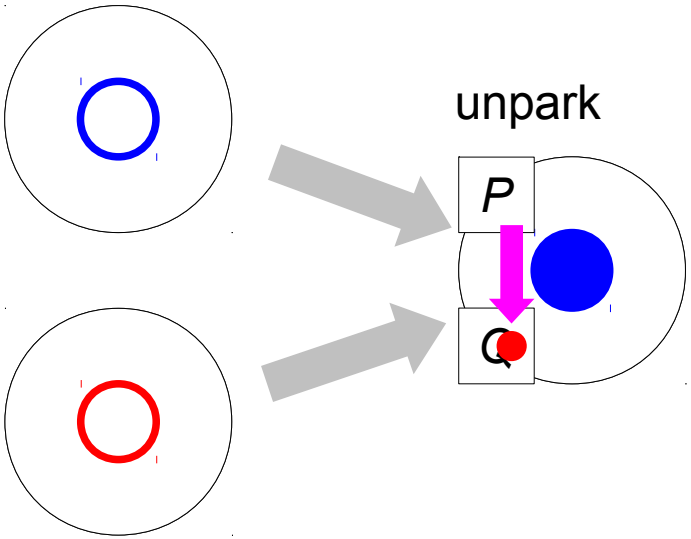
Parking Souls

- The Parking Lot:
  - a place where souls can rest ***outside*** the body
  - parking: put the currently active soul in ***P***
  - unparking: move a parked soul from ***P*** to ***Q***
  - the parking queue P is an unsorted queue



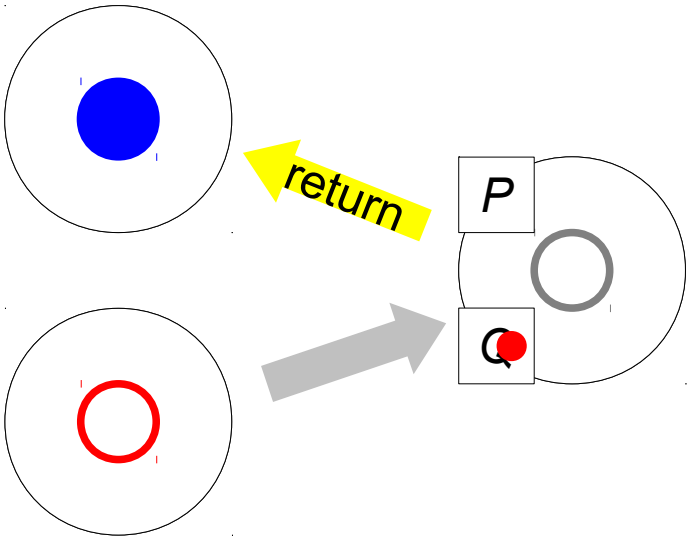
Parking Souls

- The Parking Lot:
  - a place where souls can rest ***outside*** the body
  - parking: put the currently active soul in ***P***
  - unparking: move a parked soul from ***P*** to ***Q***
  - the parking queue P is an unsorted queue



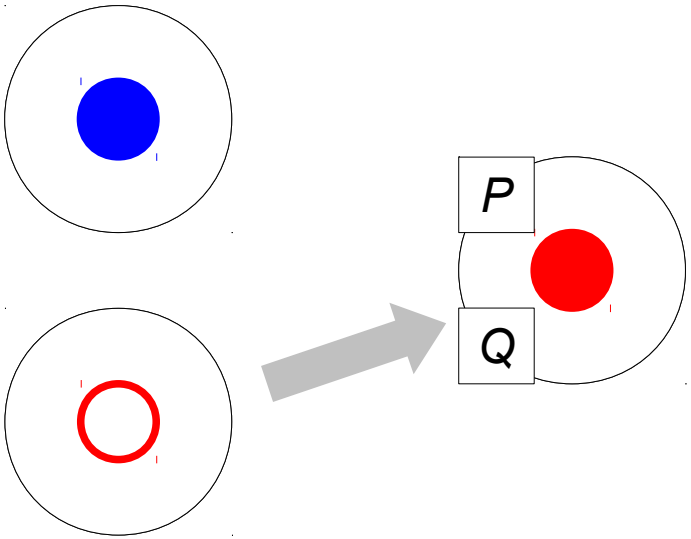
Parking Souls

- The Parking Lot:
  - a place where souls can rest ***outside*** the body
  - parking: put the currently active soul in ***P***
  - unparking: move a parked soul from ***P*** to ***Q***
  - the parking queue P is an unsorted queue



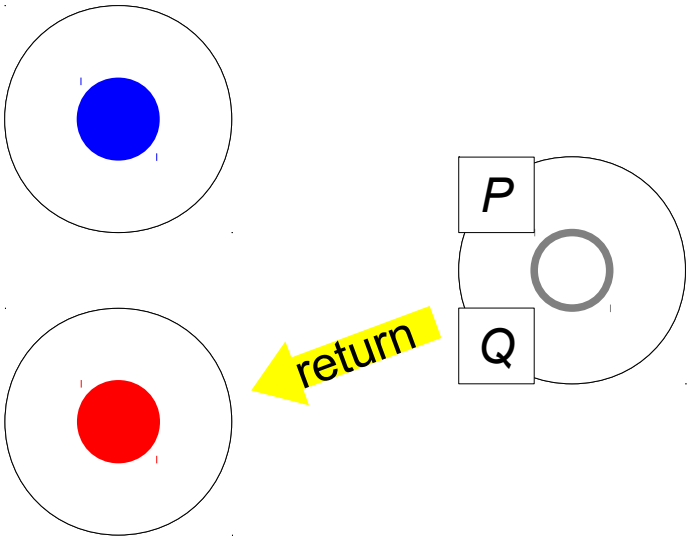
Parking Souls

- The Parking Lot:
  - a place where souls can rest ***outside*** the body
  - parking: put the currently active soul in ***P***
  - unparking: move a parked soul from ***P*** to ***Q***
  - the parking queue P is an unsorted queue



Parking Souls

- The Parking Lot:
  - a place where souls can rest ***outside*** the body
  - parking: put the currently active soul in ***P***
  - unparking: move a parked soul from ***P*** to ***Q***
  - the parking queue P is an unsorted queue



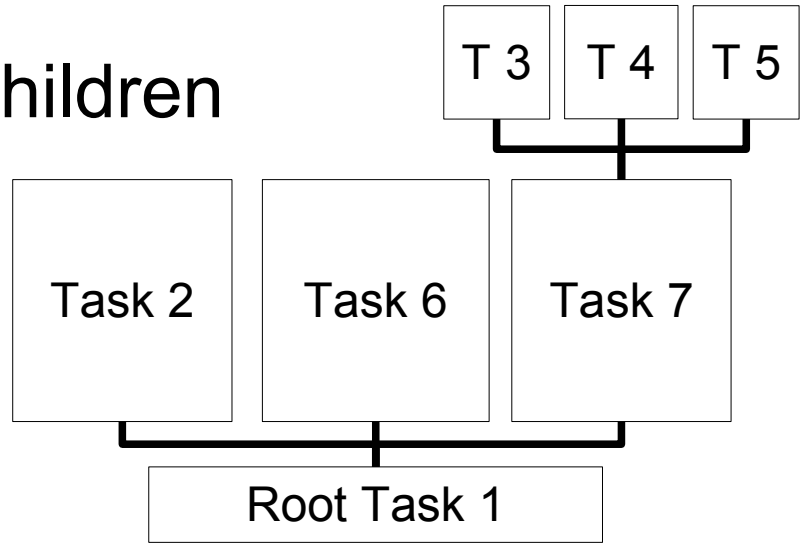
Interrupt Handling

- Threaded interrupt handlers
  - A dedicated soul is waiting on an interrupt
    - interrupt happens, interrupt source is masked
    - The waiting soul calls the associated body
  - Upon return
    - the interrupt is handled
    - unmask the interrupt source again

Resource Management

Resource Management

- Strict Task Hierarchy
  - Strict parent ↔ child relation
  - Initial task started by the kernel
  - Tasks can only grant their own resources to their children
  - Deleting a task deletes all children and grand children



Resource Management

- Resources managed by the kernel
  - Kernel and User Threads
  - Address Spaces
  - Communication Channels
  - Interrupts
  - Kernel memory
  - Free system memory
- Allocators
  - Coarse granular (4K pages)
  - Fine granular (Object Space → *capabilities*)

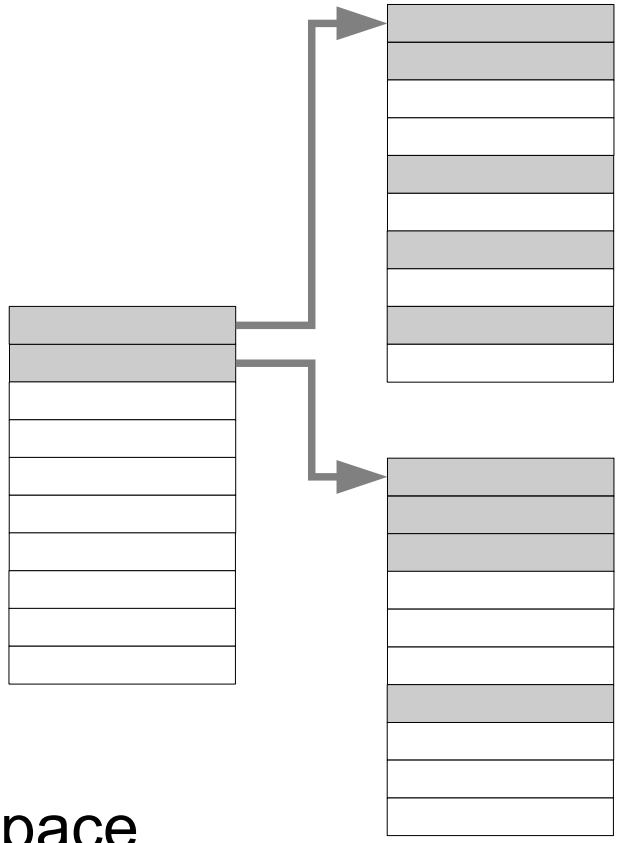
Resource Management

- Kernel memory
  - Accounted per task
  - FIFO list with free 4K pages
- Coarse memory allocations
  - 4K sized pages (MMU granularity)
  - Task descriptors
  - Thread Control Blocks + kernel stack (souls)
  - Page tables
  - Object Space pages

Resource Management

- Object Space
  - One OS per task
  - Object = single capability
  - Fine granular memory allocator
    - 2-level lookup-table
    - using 4K pages
  - 16K+ entries of 64 byte
  - OS can grow, but not shrink

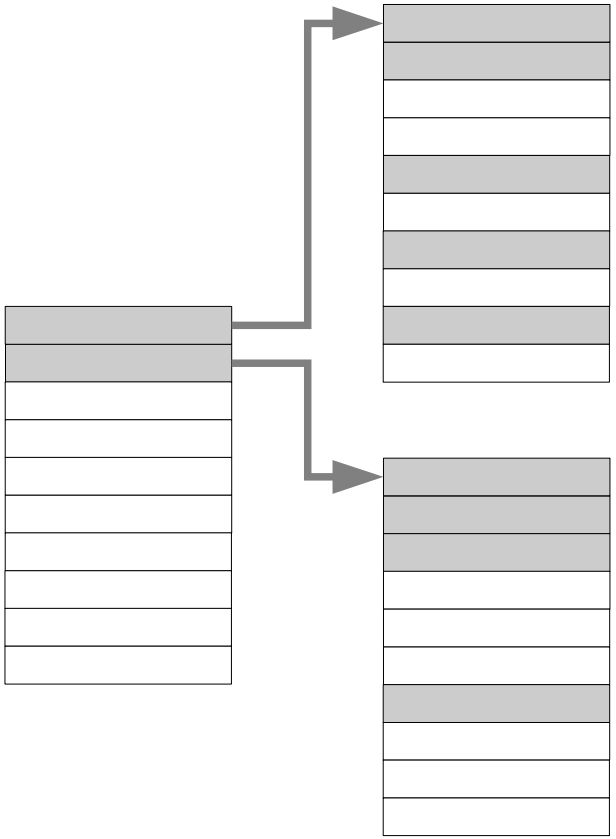
    → lock free access!



- Tasks can only access their own Object Space
  - Safety: no partition interference through locking
  - Security: no covert channels

Resource Management

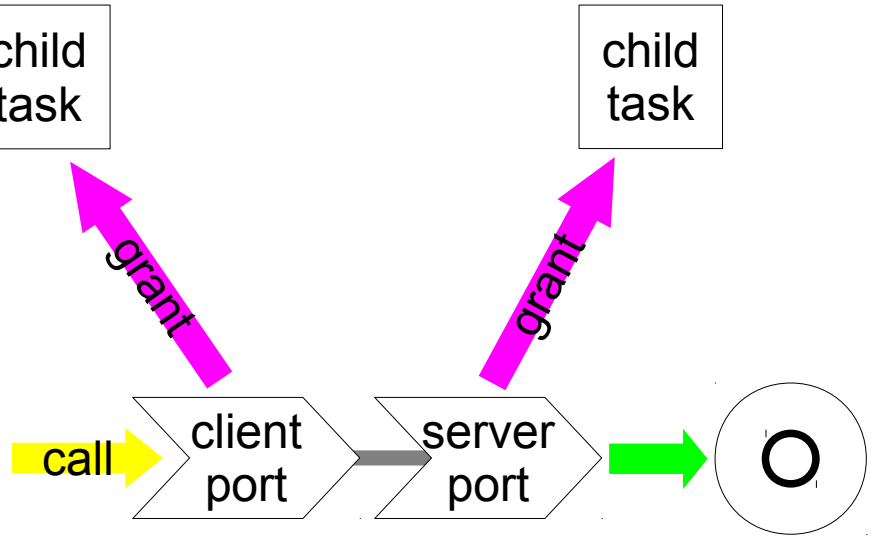
- Capabilities in Object Space
  - Reference to own task (entry #0)
  - Child Tasks
  - Child Address Spaces
  - Souls
  - Bodies
  - *Ports* (communication endpoints)
  - Interrupts



- Other Capabilities
  - Memory → implicit by virtual address

Resource Management

- Communication Channels
  - Handle cross task communication
  - *Port:* channel endpoint
  - *Channel*: two endpoints
    - *client* and *server* side objects
    - granted to child tasks
- Operations
  - Server *binds* body to port
  - Client *calls* port
  - Channel remains open until closed

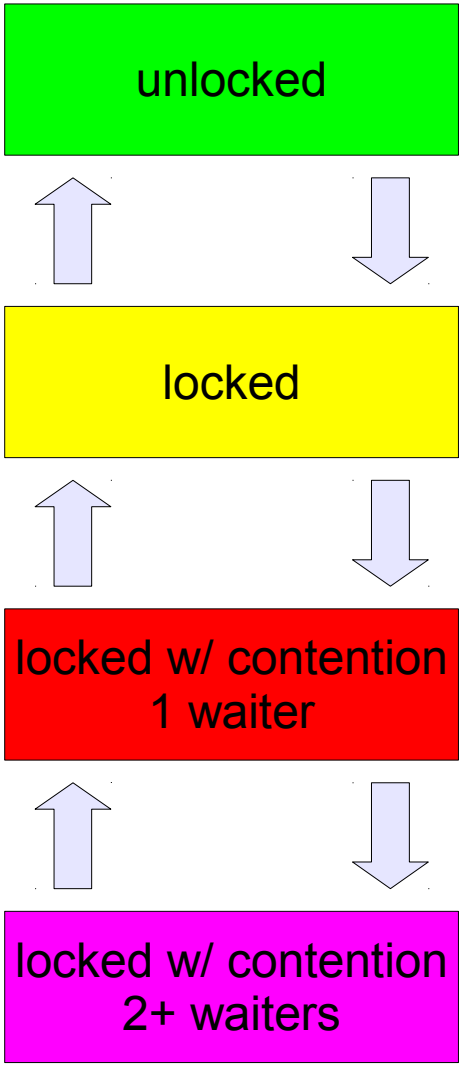


Locking

Futexes

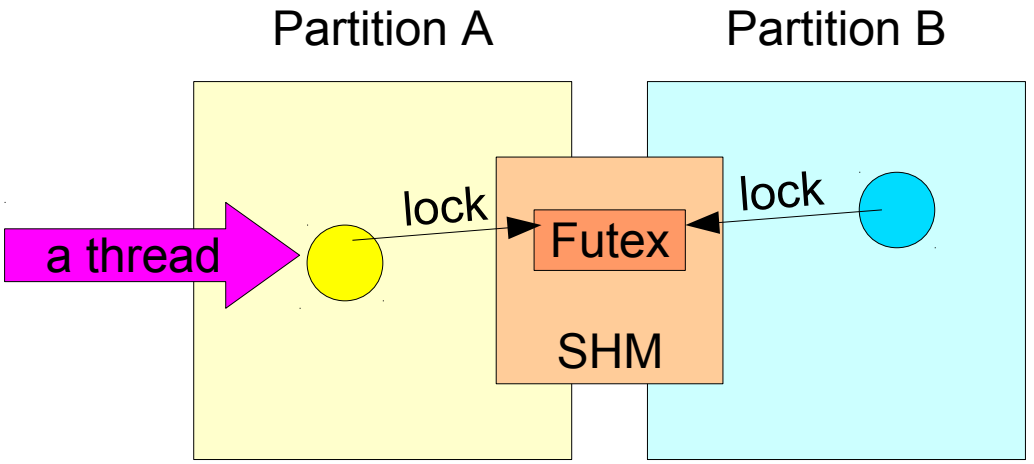
- Derived from Linux
- 32-bit Integers in user space
- Fast path: use atomic ops
- Use syscalls only on contention
- The kernel maintains a wait queue

- Tricky with resource partitioning!



Futexes

Partitioned Environment

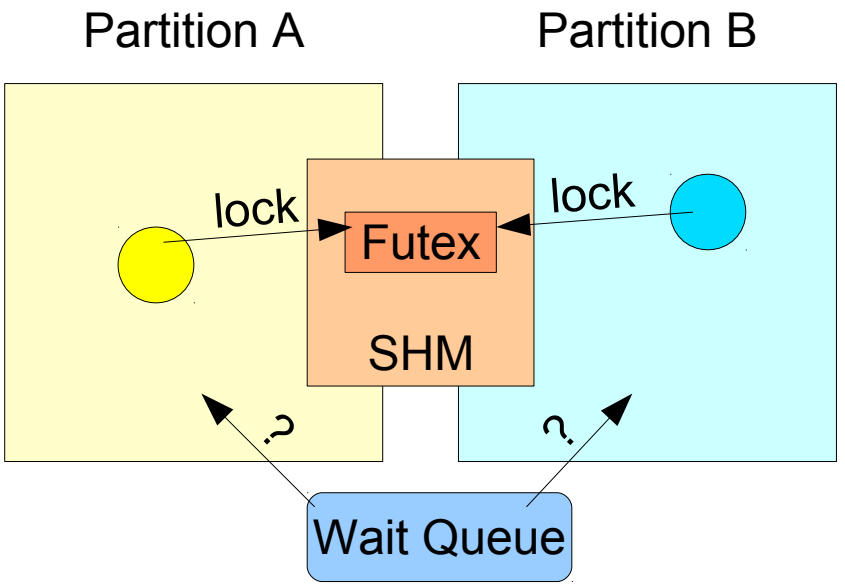


Futexes

Partitioned Environment

Problem

- Q: Wait queue belongs to Partition A or Partition B?
- Pre-allocate wait queues?

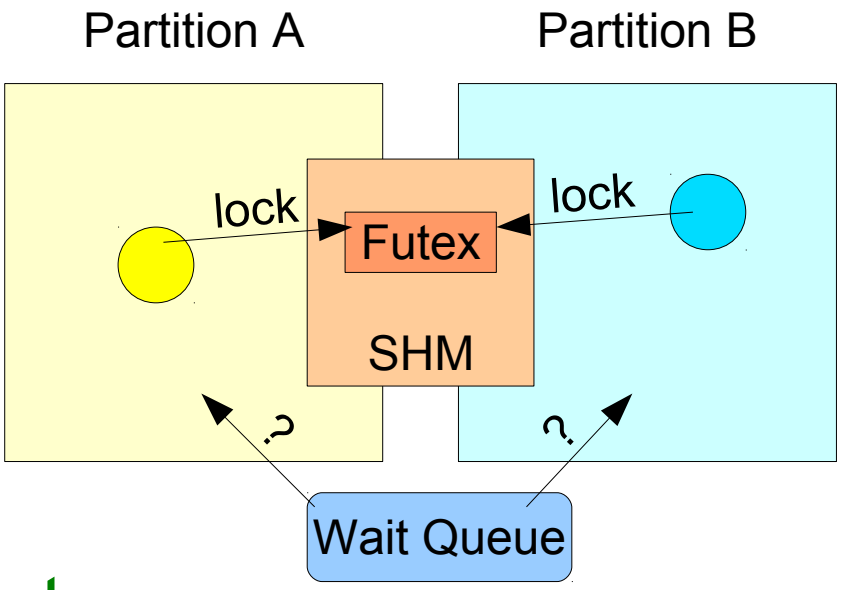


Futexes

Partitioned Environment

Problem

- Q: Wait queue belongs to Partition A or Partition B?
- Pre-allocate wait queues?



Place head of wait queue into user space, next to the futex

Futexes

- Futex wait queue
  - FIFO sorted *O(1) time*
  - Priority sorted *O(log n) time*
  - Implementation in linear space
- Primitives
  - Mutexes
  - Condition variables
  - Barriers
  - Counting semaphores
  - Reader-writer locks

Locking Architecture

Futexes

- use between threads in the same partition
- shareable between multiple address space
  - but only if all parties trust each other
  - must have the same level of criticality

Body & Soul RPC

- Parking concept → monitor
- use between threads in different partitions
  - encapsulate critical operations in a dedicated body
  - caller must trust callee, but not vice versa

Locking Architecture

Locking Improvements:

- Apply to both Futexes and RPC
- Priority Ceiling Protocol
  - Combine futexes with “lazy user space prio switching”
  - Raise and lower thread priority in user space
  - Kernel synchronizes scheduling priority on IRQ / syscall
- (Migratory) Priority Inheritance Protocol
  - Blocked threads boost the priority of lock holders
  - May inherit CPUs as well
  - Problem: robust implementation …
    - … what if the lock holder blocks?
    - … limit recursions?

Finally ...

Implementation

- Implementation in C99 with GNU extensions
- GCC 4.3 to 4.8
- LLVM/Clang 3.3 to 3.4
- Supported Architectures:
  - X86 32 bit and 64 bit
  - ARM v6 and v7
  - PowerPC e500+ cores or newer
- Open Source License

Implementation

- Work in progress (January 2014):
  - Shown features are 70% implemented
  - 20,000 lines of C code (including tools + test code)
  - 2,000 lines of assembler code
- TODO:
  - Priority sorting in RPC calls and Futexes
  - Cross-address space calls
  - Soul parking and Interrupt Handling
  - Internal SMP locking in the kernel
  - Priority Inheritance!

Conclusion

- Body & Soul: building-block for multi-threaded execution environments like POSIX Pthreads
  - Pthread_create/join → fork / join
  - Signals → “forced call” + Exception Handling
  - Synchronization → Futexes

- Overall concept should fit the requirements of mixed criticality systems

Outlook

- Focus on extending Futexes first
  - Papers for my PhD!!!
- Userspace?
  - Bionic Libc (Android) + OpenMP
  - PikeOS paravirtualized Linux
  - Benchmark-Suite
- Drivers?
  - Rump kernel (NetBSD drivers and stacks)
  - Genode OS Framework

Thank you for your attention!

Questions?