Microkernel Construction

Communication
SS2011

Torsten
Frenzel

TU Dresden
Operating
Systems Group

Outline of this Lesson

- Use cases
- Terms and primitives
- Types
  - Copying data
  - Mapping data
  - Resolving page faults
  - Sending interrupts
- A look at the IPC path
  - General structure
  - Reasons for complexity of the IPC operations
- Micro-benchmark

Torsten
Frenzel

TU Dresden
Operating
Systems Group

Use Cases

- Generally in microkernels:
  - Exchange data
  - Synchronization
  - Timeouts
  - Yield CPU (sleep)
- Specifically in L4:
  - Grant access to resources (memory, io-ports, capabilities)
  - Manage and handle interrupts and other resources
- Allows feature-rich user-level protocols on top
  - Tailored for client/server communication
  - Microkernel talks some simple protocols to support user applications (page fault IPC, exception IPC)
- Optimized for performance

Torsten Frenzel

TU Dresden
Operating Systems Group

POSIX IPC Primitives

- Semaphore
  - Atomic increment and decrement of counter with wait queue
  - Used to synchronize critical sections
- Shared memory
  - Special file type, that uses physical memory
- Message queue
  - Special file type for block-based synchronous communication
- Pipe
  - Untyped, directed communication channel
- FIFO
  - Special file type similar to a pipe
  - Blocking and non-blocking mode
- Regular files
- Signals
  - Asynchronous trigger; many have predefined actions

Torsten Frenzel

TU Dresden
Operating Systems Group

IPC Properties and Terms

- Connectionless vs. connection-oriented
  - What is the order of delivered messages
- Reliable vs Unreliable
  - Can a message get lost
- Point-to-point vs broadcast or multicast
  - How many receiver can be addressed
- Asynchronous vs. synchronous
  - Does the sender wait for the receiver
- Buffer vs. unbuffered
  - Message buffering within the kernel
- Direct vs. indirect
  - How is the destination addressed
- Data items
  - What type of data items are transferred

Torsten Frenzel

TU Dresden
Operating Systems Group

Synchronous vs. Asynchronous IPC

- Synchronous IPC
  - Sender blocks if receiver is not ready
  - Requires no data buffering inside the kernel
  - *Wait queue* for every receiving thread
    - Holds blocked senders for the receiver
    - Requires enqueue/deqeue policy
- Asynchronous IPC
  - Sender does not block if receiver is not ready
  - Requires data buffering inside the kernel
  - Suitable for interrupts because only one bit is transferred

Torsten Frenzel

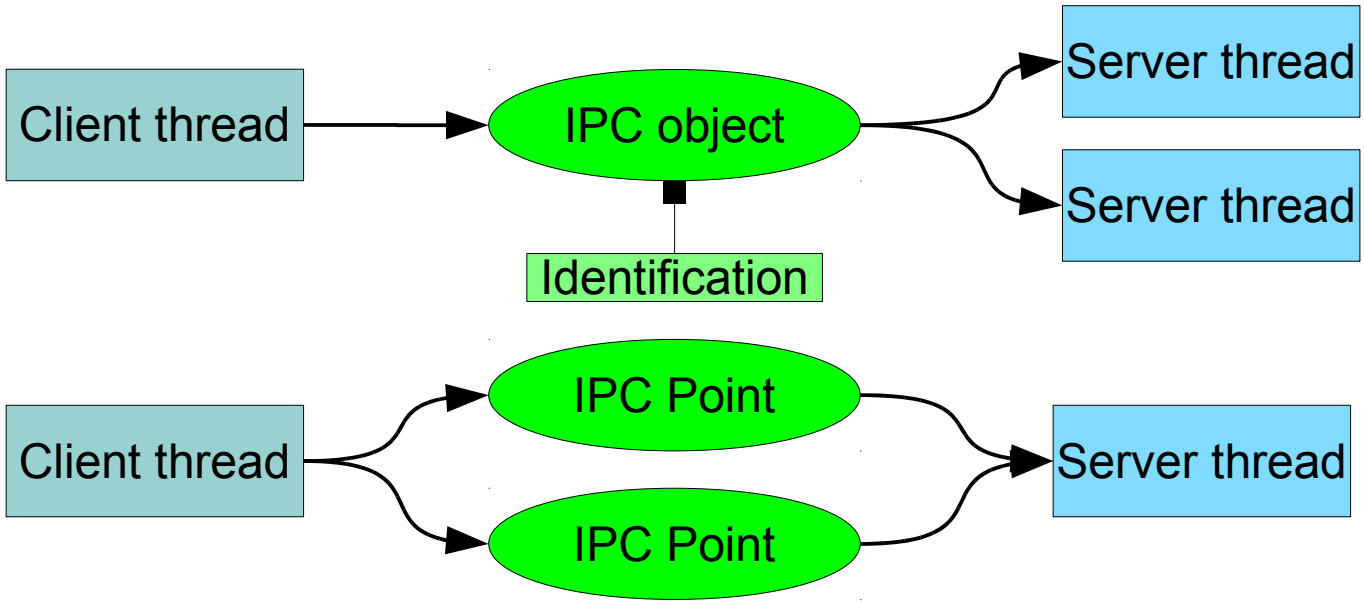
TU Dresden
Operating Systems Group

Direct vs. Indirect IPC

Direct IPC:

- Sending thread addresses another thread

- Indirect IPC:
  - Sending thread addresses communication object (called: port, portal, gate, endpoint)
  - IPC object redirects message to a receiver thread
  - Advantage 1: Hide the implementation of server threads
  - Advantage 2: Possibility to hold state for client-server connection



Torsten Frenzel

TU Dresden
Operating Systems Group

IPC Primitives

- Send
  - Send to specific thread/IPC object
- Closed receive
  - Receive from specific thread
- Open receive
  - Receive from any thread
- Send and closed wait
  - Send to and receive from specific thread/IPC object
  - Typical client operation - *call*
- Send and open receive
  - Receive from any thread and send to specific thread
  - Typical server operation – *reply-and-wait*
- Sleep
  - Neither send nor receive
  - Yield until timeout expires

Torsten Frenzel

TU Dresden
Operating Systems Group

Example: Mutual Exclusion with IPC

- User-level protocol provides mutual exclusion:
  - *Synchronization thread* protects critical section
  - *Worker thread* calls synchronization thread before entering
  - Synchronization thread replies if critical section can be entered
  - Worker thread blocks if critical section not available
  - Worker thread calls synchronization thread after leaving
  - Synchronization thread sends reply to next waiting worker thread
- Implementation not optimal for performance but correct
- Others:
  - Producer-consumer synchronization together with shared memory
  - Connection-oriented client-server communication

Torsten Frenzel

TU Dresden
Operating Systems Group

IPC Types

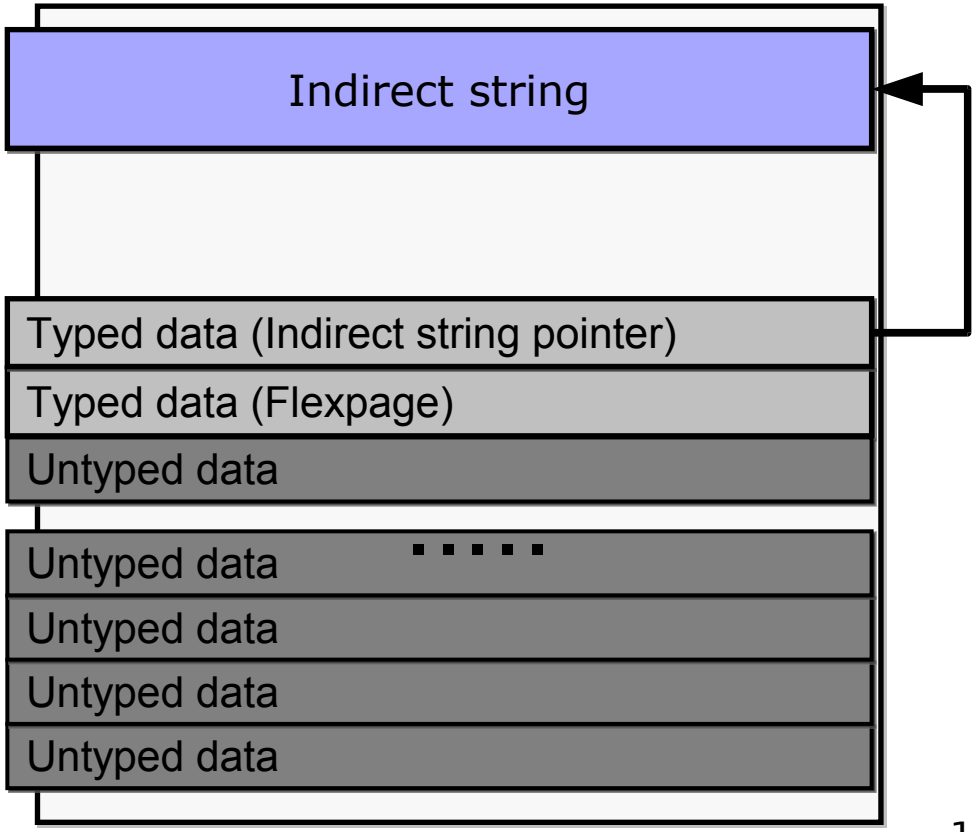
- Register-only IPC
  - Very fast but amount of data is limited to number of registers
- UTCB IPC
  - Copy data from one UTCB to another UTCB
  - Fast but amount of data is limited to UTCB size
- User-memory IPC
  - Copy of memory areas between user address-spaces
  - Amount of data is not limited
  - Page faults can occur
- Flexpage IPC
  - Mapping of memory areas and capabilities
  - Page fault IPC is special case
- Interrupt IPC
  - Relaying Interrupts as messages

Torsten Frenzel

TU Dresden Operating Systems Group

Message Buffer Descriptor

- Describes layout of transferred data items
- Located in registers / UTCB / user memory
- Contains untyped and typed items
- Typed items are interpreted by the kernel:
  - Indirect string pointers
  - Flexpages descriptors
  - Capability descriptors

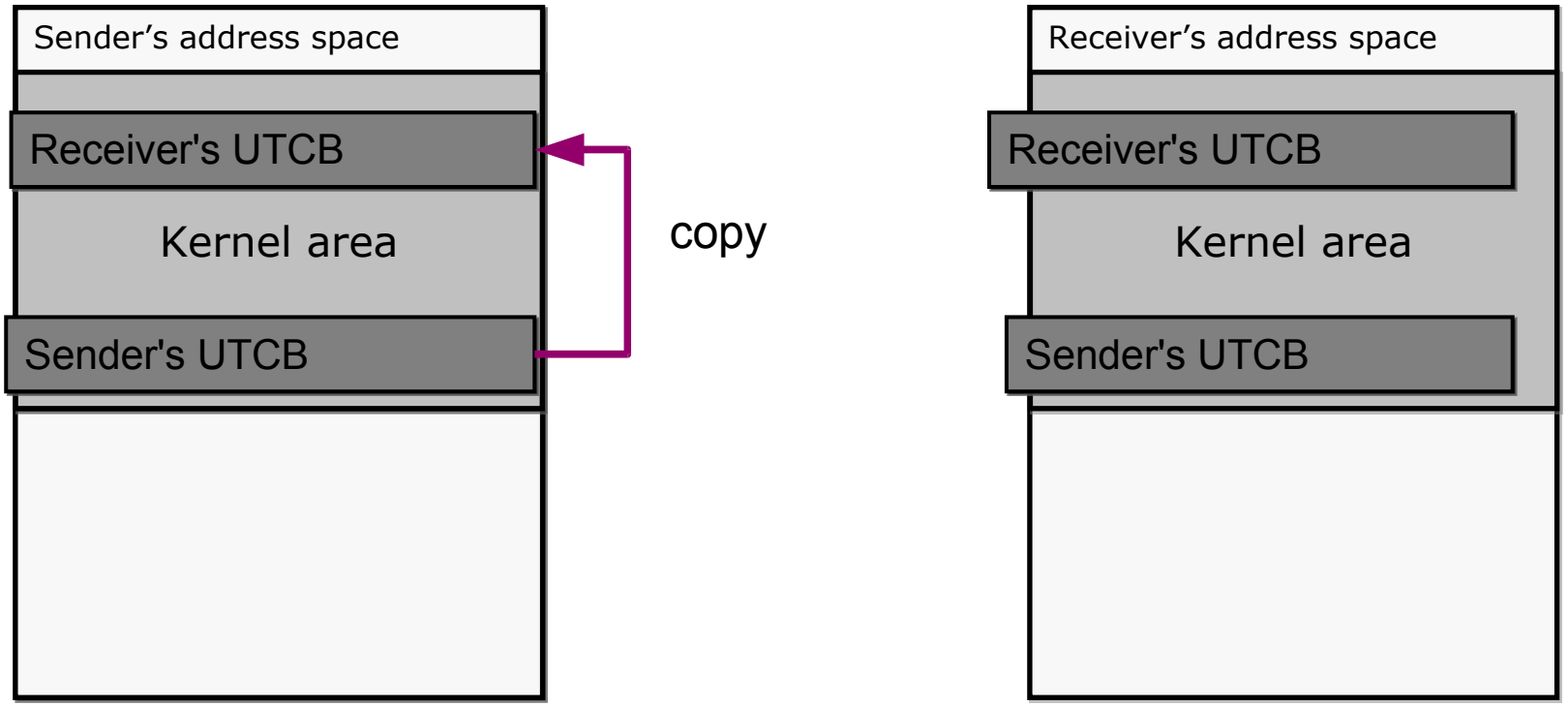


Torsten Frenzel

TU Dresden
Operating Systems Group

UTCB IPC

- UTCBs hold relevant information of a thread
  - Kernel and user accessible (read and write)
  - Task-local and associated with **one** or more threads
  - No page faults can occur on access
- Can be used as message buffer

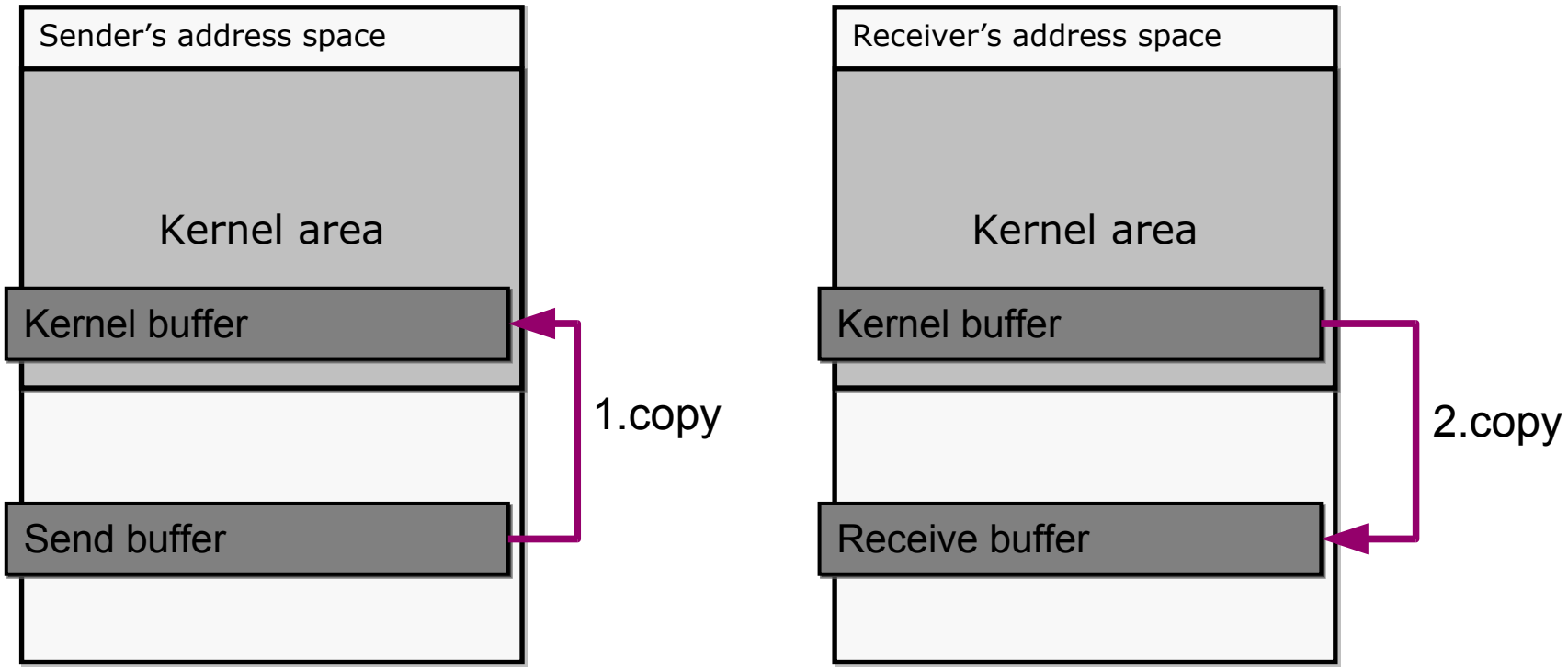


Torsten Frenzel

TU Dresden Operating Systems Group

User-Memory IPC with Kernel Buffer

- Two copy operations
  - Copy send buffer to kernel buffer in sender's address space
  - Switch to receiver's address space
  - Copy kernel buffer to receive buffer in receiver's address space

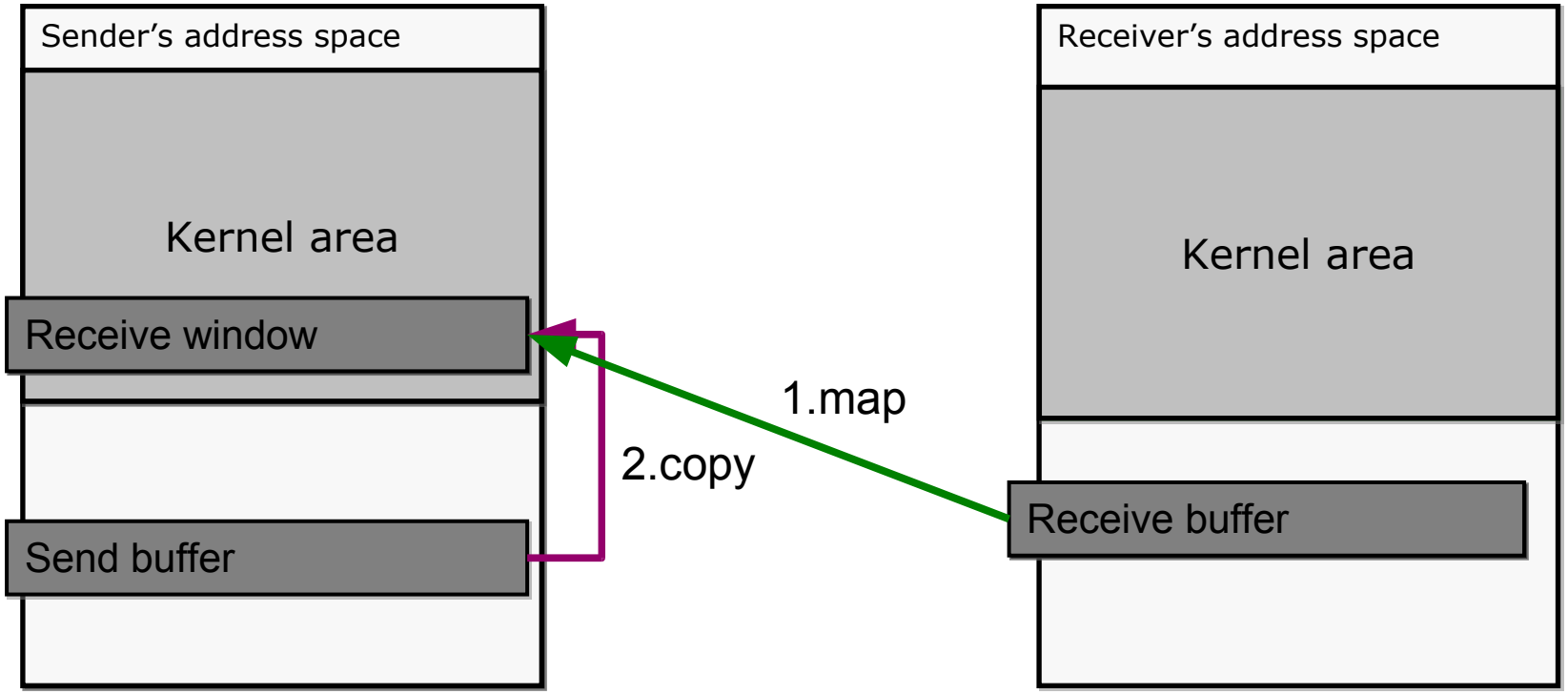


Torsten Frenzel

TU Dresden
Operating Systems Group

User-Memory IPC with IPC Window

- IPC window provides 'view' into another address space
  - Temporary mapping inside the kernel address space
  - Flushed on each thread switch
- One copy operation
  - Map receive buffer to IPC window
  - Copy send buffer to IPC window

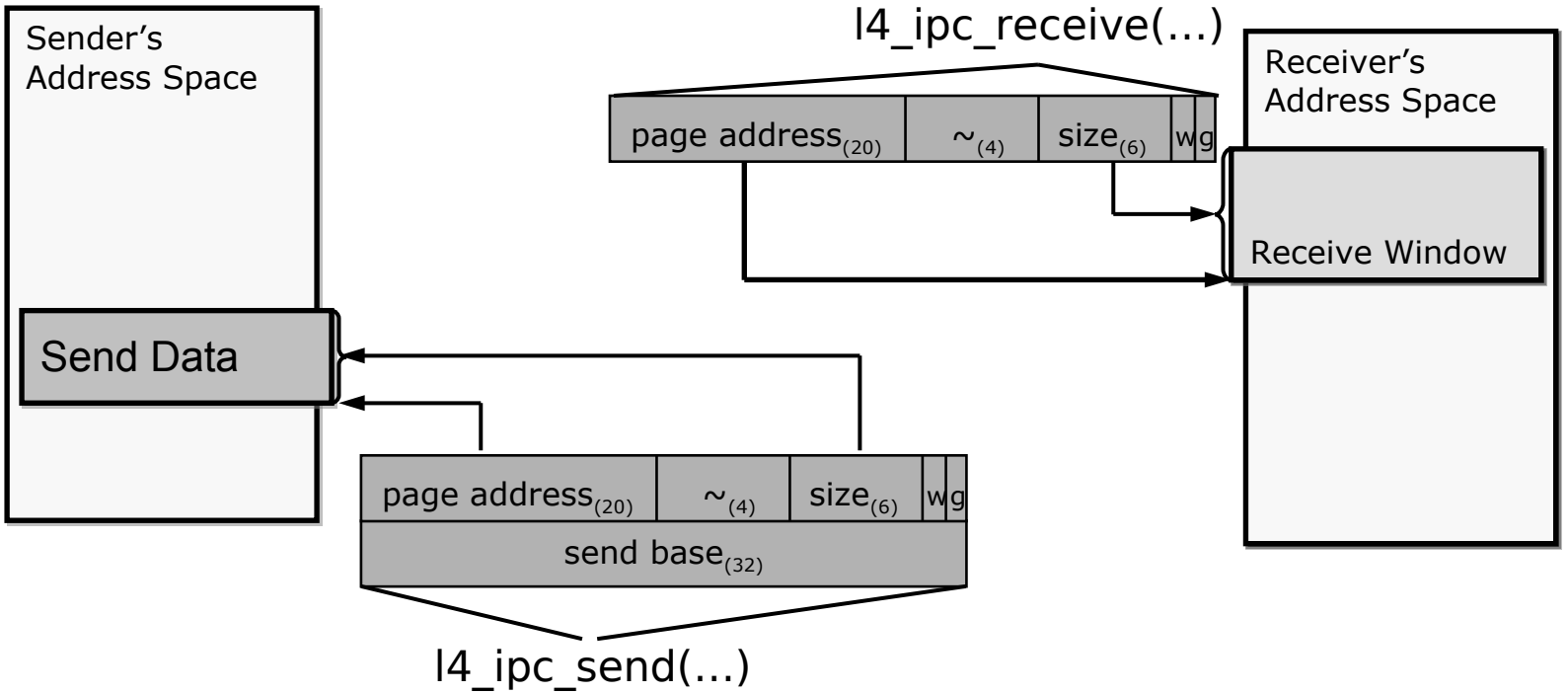


Torsten Frenzel

TU Dresden Operating Systems Group

Flexpage IPC (1)

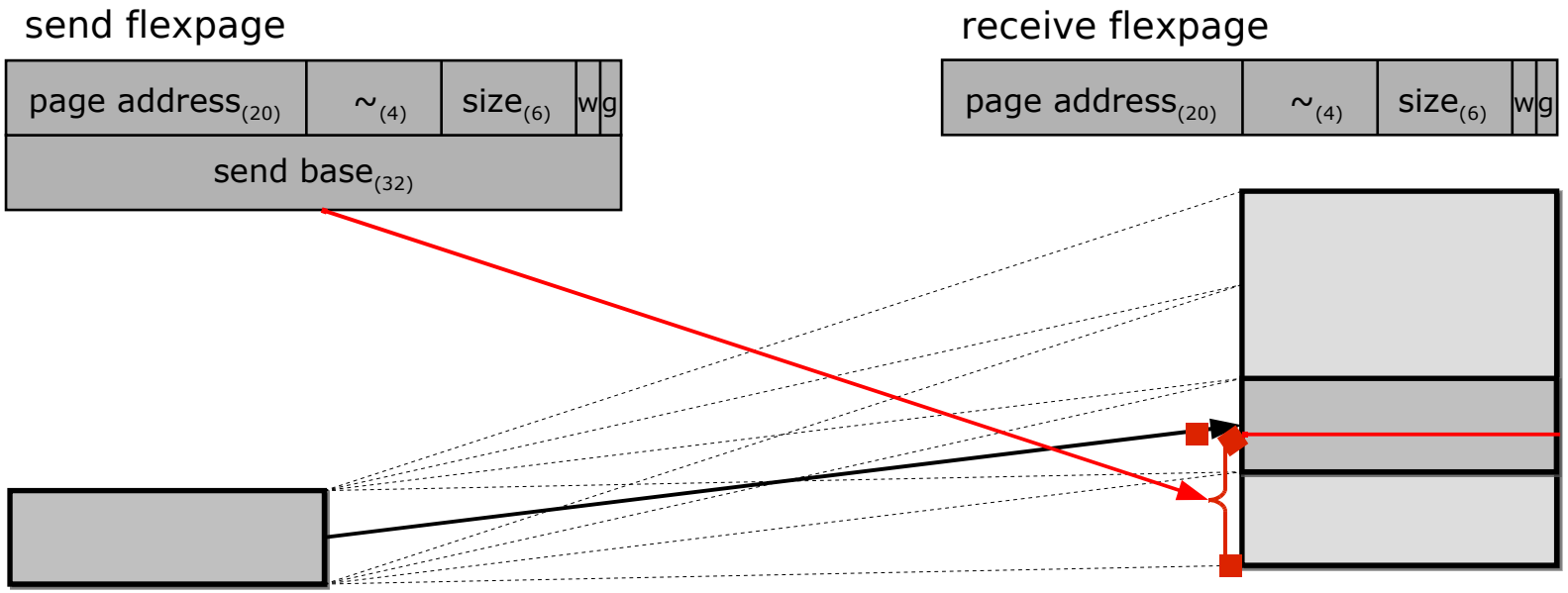
- Flexpages describe areas in address spaces (size aligned)
- Flexpage size 2^{size}, smallest is hardware page (4KByte on IA32)
- Sender specifies one or more Send-flexpages
  - Send-flexpage is a flexpage with send base in receive window
- Receiver specifies Receive window flexpage



Flexpage IPC (2)

Send-flexpage is smaller than the receive window

- Target position in receive window is derived from page alignment and send base



Example: send flexpage has ¼ size of receive flexpage
→4 possibilities to map flexpage into receive window

Torsten
Frenzel

TU Dresden
Operating
Systems Group

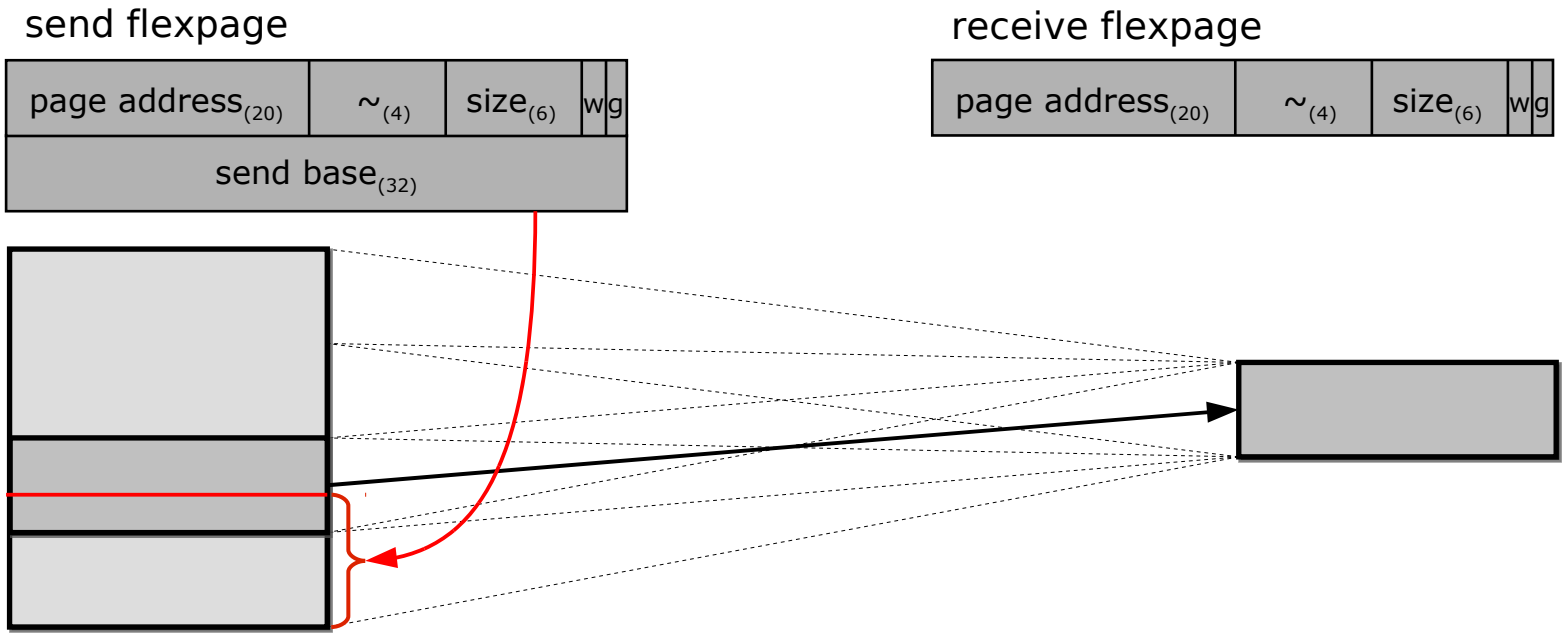
Flexpage IPC (3)

Send flexpage is larger than receive window

- Source position in send flexpage is derived from page alignment and send base

⇨ *send base* is a hot spot for mapping IPC

⇨ Actual *send base* depends on information about the receiver



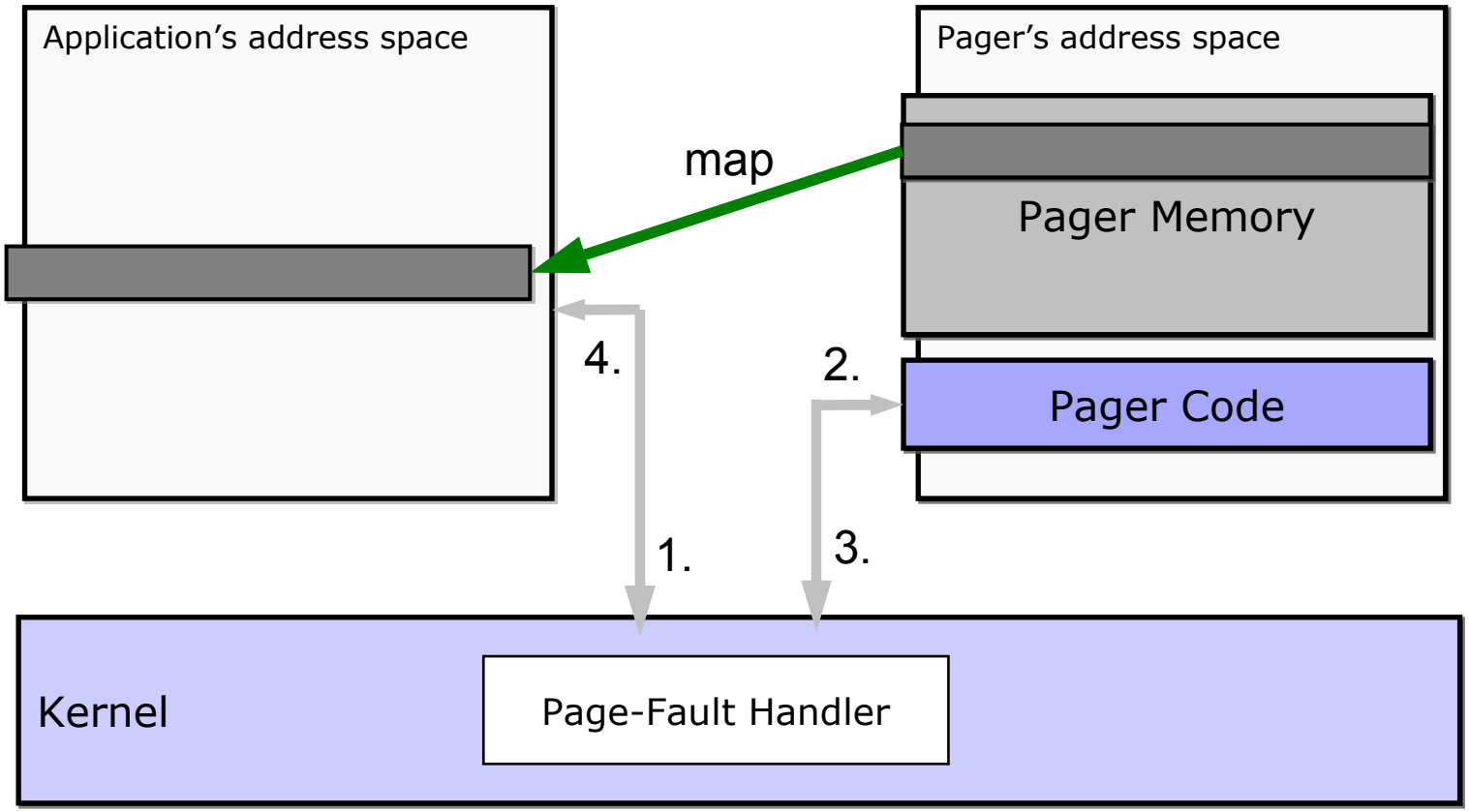
Example: receive flexpage has ¼ size of send flexpage

Torsten Frenzel

TU Dresden
Operating Systems Group

Pagefault Handling (Revisited)

- General pagefault mechanism:
  1. Application thread invokes handler due to a pagefault
  2. Pagefault handler sends Pagefault IPC to pager thread
  3. Pager thread replies with flexpage IPC
  4. Handler resumes application thread

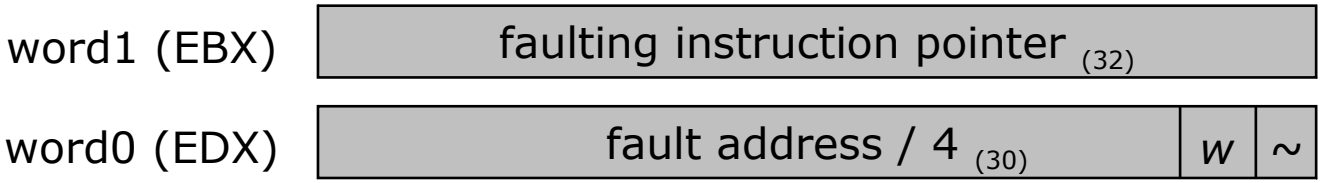


Torsten Frenzel

TU Dresden Operating Systems Group

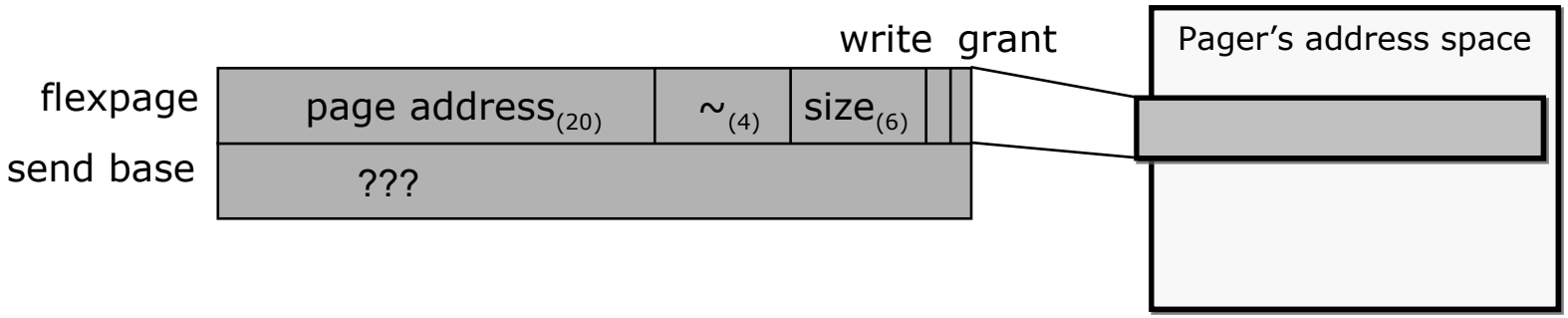
Pagefault IPC (1)

2. Pagefault handler sends Short IPC (register-only transfer):



w = 0 read page fault
w = 1 write page fault

3. Pager has to reply with Flexpage IPC:

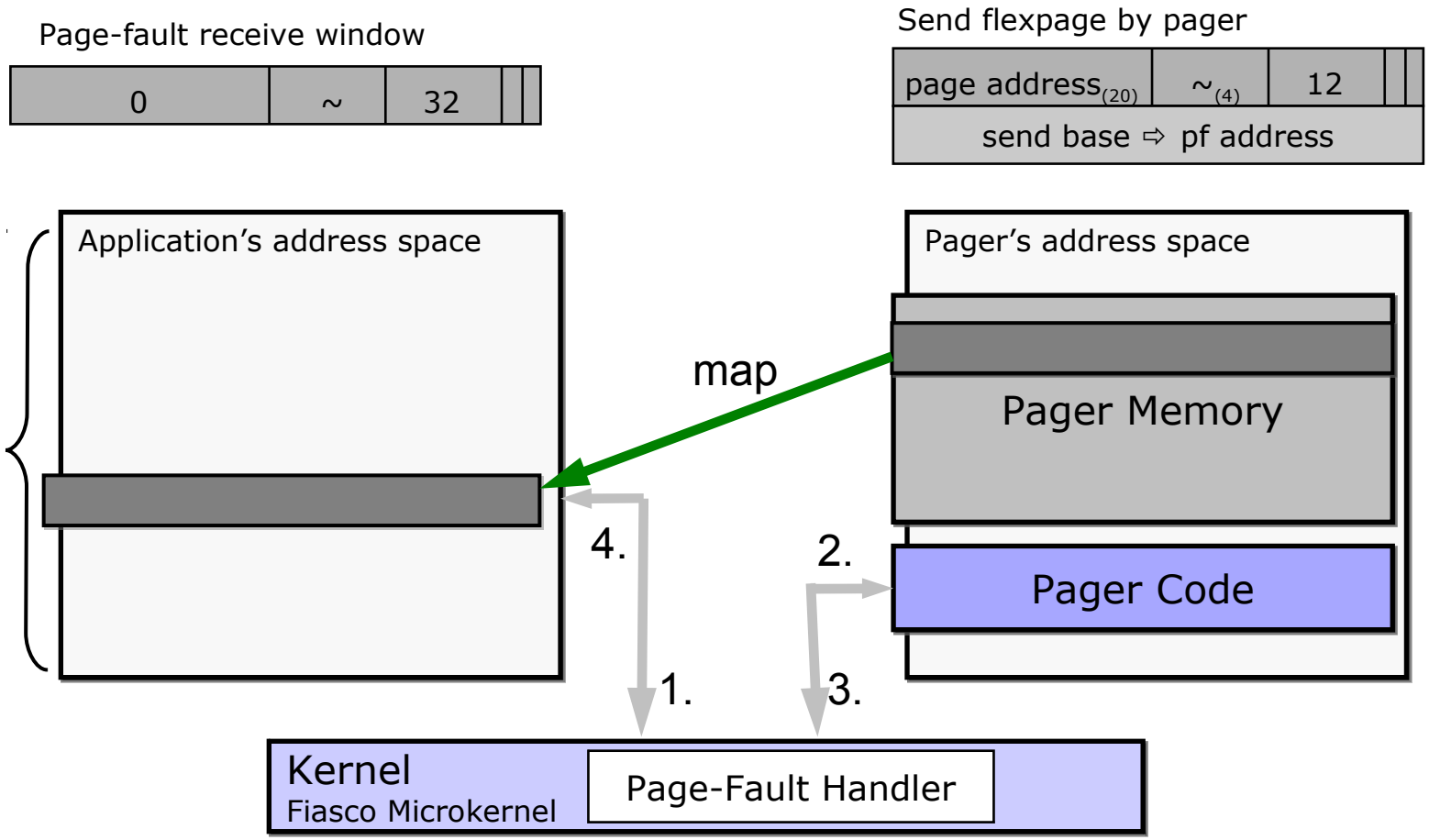


⇨ Maps one or more pages from pager address space to application address space

Pagefault IPC (2)

- Kernel pagefault handler sets receive window to whole address space

⇨ Pager can map more than just the page where the page fault happened to the client



Torsten Frenzel

TU Dresden
Operating Systems Group

Interrupt IPC

- Kernel has driver to program interrupt controller
- Interrupts from peripheral devices (disk,network, …) preempt current activity and trap in the kernel
- Interrupts are implemented as special sender object
  - Threads can *attach* itself to interrupts
  - Thread can wait and receive message from an interrupt
  - Thread can wait on many interrupts using open wait
  - One bit of information is transferred
  - Interrupt is enqueued into sender queue of receiver thread if thread is not ready
- Interrupt source is disabled after it is has triggered and needs to be explicitly enabled by the receiver

Torsten Frenzel

TU Dresden
Operating Systems Group

IPC Phases

- Handshake phase
  - Find partner and check state of partner
  - If partner is ready for IPC:
    - Start data transfer phase
  - If partner is not ready for IPC:
    - Set up timeout and wait for partner
- Data transfer phase
  - Copy data and map resources
  - Handle page faults if necessary
- Finishing phase
  - Wake up partner

Torsten Frenzel

TU Dresden
Operating Systems Group

Principle Algorithm (Symmetric)

- Case 1: Receiver arrives first
  - Receiver arrives
    - ➔ Prepare ready-to-receive and block
  - Sender arrives
    - Transfer data
    - Wake up receiver
- Case 2: Sender arrives first
  - Sender arrives
    - ➔ Prepare ready-to-send, enqueue into wait queue and block
  - Receiver arrives
    - Dequeue sender from send queue
    - Transfer data
    - Wake up sender

Torsten Frenzel

TU Dresden
Operating Systems Group

Principle Algorithm (Sender Driven)

- Only sender transfers data
- Receiver can only trigger transfer phase
- Case 1: Receiver arrives first
  - … same as symmetric algorithm
- Case 2: Sender arrives first
  - Sender arrives
    - ➔ Prepare ready-to-send, enqueue into *wait queue* and block
  - Receiver arrives
    - Dequeue sender from send queue
    - Wake up sender and block
  - Sender wakes up
    - Transfer data
    - Wake up receiver

Torsten Frenzel

TU Dresden Operating Systems Group

Why is IPC Complex?

- IPC operation is non-atomic
  - Complex state transitions between phases
  - Sequence of send and receive part
  - Synchronization between sender and receiver
  - Synchronization with other operations
  - *Preemption points* can add further complexity
- IPC can be aborted for various reasons
  - Due to timeouts, exceptions or preemptions due to concurrent operations
  - Requires extensive error checking
- Page faults can happen, if user-memory is touched
  - Resolving a page fault involves a third thread
  - Current state of the ongoing IPC operation needs to be preserved during page fault handling

Torsten Frenzel

TU Dresden
Operating Systems Group

IPC Timeouts

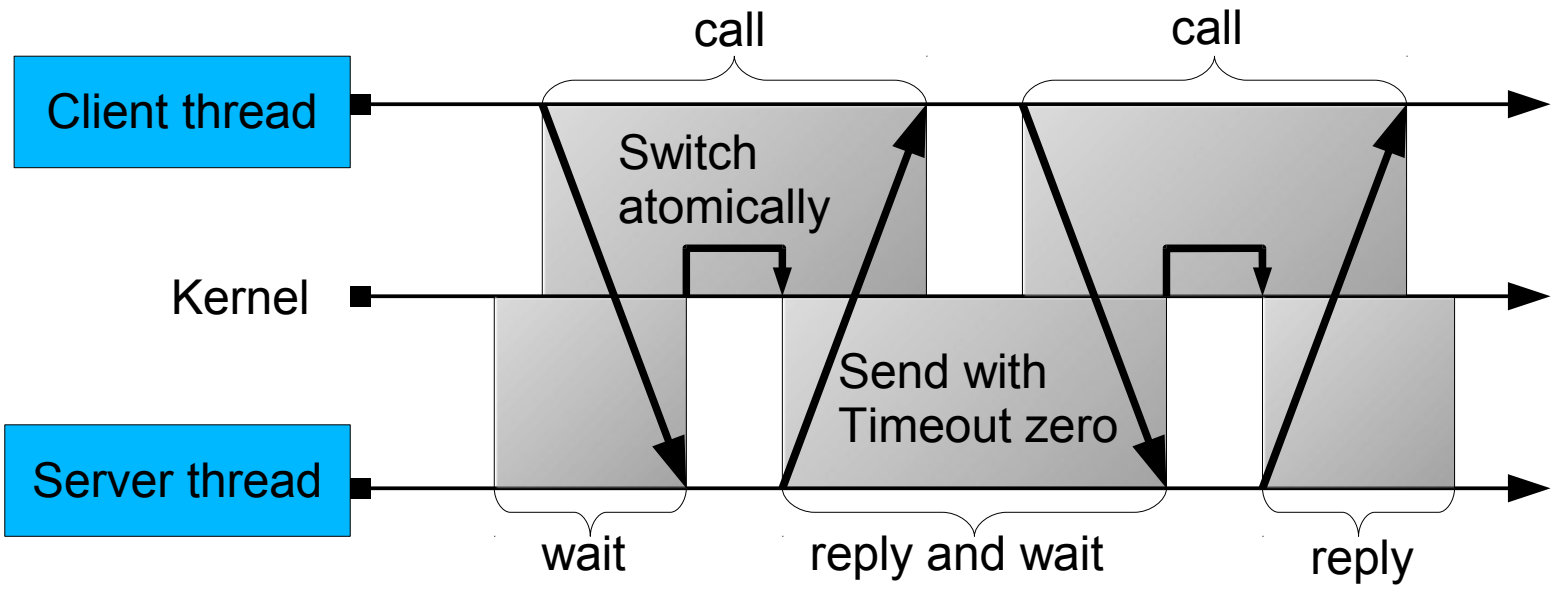
- Timeouts control how long to wait for the partner
- Send and receive timeouts
  - Infinite
    - Specified by communication with trusted partner
    - Example: client waits for an answer from the server
  - Zero
    - Specified by communication with untrusted partner
    - Example: server replies to a client request
  - Finite
    - Waiting for a specific amount of time
    - Values range from microseconds to hours
    - Example: sleep a specific amount of time
- Page fault timeouts
  - Used during long IPC
  - Limit's partner's page fault resolve time

Torsten Frenzel

TU Dresden
Operating Systems Group

Switching from Send to Receive

- Switch from send part to receive part **atomically**
- Why prepare receive part atomically?
  - Servers do not trust clients
  - → Servers reply with timeout zero
  - → Client needs to be ready to receive immediately after sending
  - → Flip one bit to switch from send to receive part



Torsten
Frenzel

TU Dresden
Operating
Systems Group

Pagefaults during IPC

- Long IPC can touch user memory
  - ➔ Page faults can occur if virtual address is not mapped to physical memory
  - – Page fault in sender's address space
  - – Page fault in IPC window (receiver's address space)
- Page fault handler suspends ongoing IPC operation
  - – Save current IPC state (somewhere)
  - – Setup *nested* page fault IPC
    - ➔ Start another IPC during ongoing IPC
  - – If page fault IPC succeeded:
    - • Restore suspended IPC operation
  - – If page fault IPC failed:
    - • Abort ongoing IPC operation
- Even more complicated if ongoing page fault IPC is canceled

Torsten Frenzel

TU Dresden Operating Systems Group

IPC Termination

- Normally
- Abnormally because of non-existing partner
- Abnormally because of memory copy failure
  - Due to too small send or receive buffer
- Abnormally because of memory map failure
  - Due to shortage of page tables
- IPC partner is canceled by another thread
  - Cancel operation executed before rendezvous
- IPC partner is aborted by another thread
  - Cancel operation executed after rendezvous
- IPC partner is killed by another thread
- Timeout occurs
  - Send/receive IPC timeout
  - Send/receive page fault timeout

Torsten Frenzel

TU Dresden Operating Systems Group

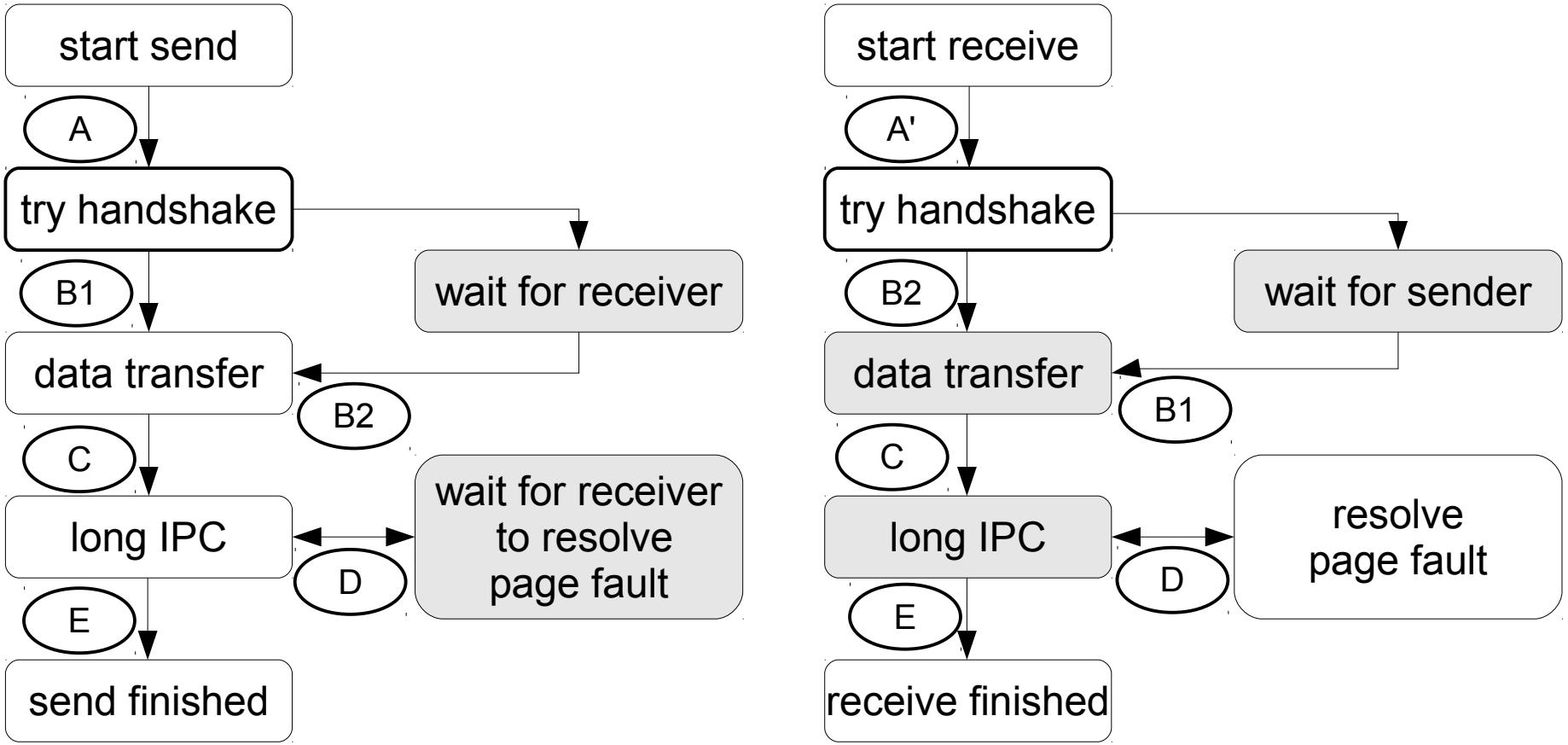
IPC States

- IPC operation is not atomic and requires protocol
- State is encoded in state word of TCB
- Consistency between state of sender and receiver
- IPC code uses complex state machine
- Examples of IPC states that can occur:
  - Receiver waits for sender to arrive
  - Sender waits for receiver to arrive
  - Sender handshakes with receiver
  - Data transfer has started after handshake
  - Data transfer for long IPC has started (page faults can occur)
  - Sender waits for receiver to handle page fault during long IPC
  - Receiver sleeps during data transfer in long IPC
  - IPC operation has abnormally terminated
- Atomic transitions between states (not always possible)

Torsten Frenzel

TU Dresden Operating Systems Group

Interlocking of IPC states

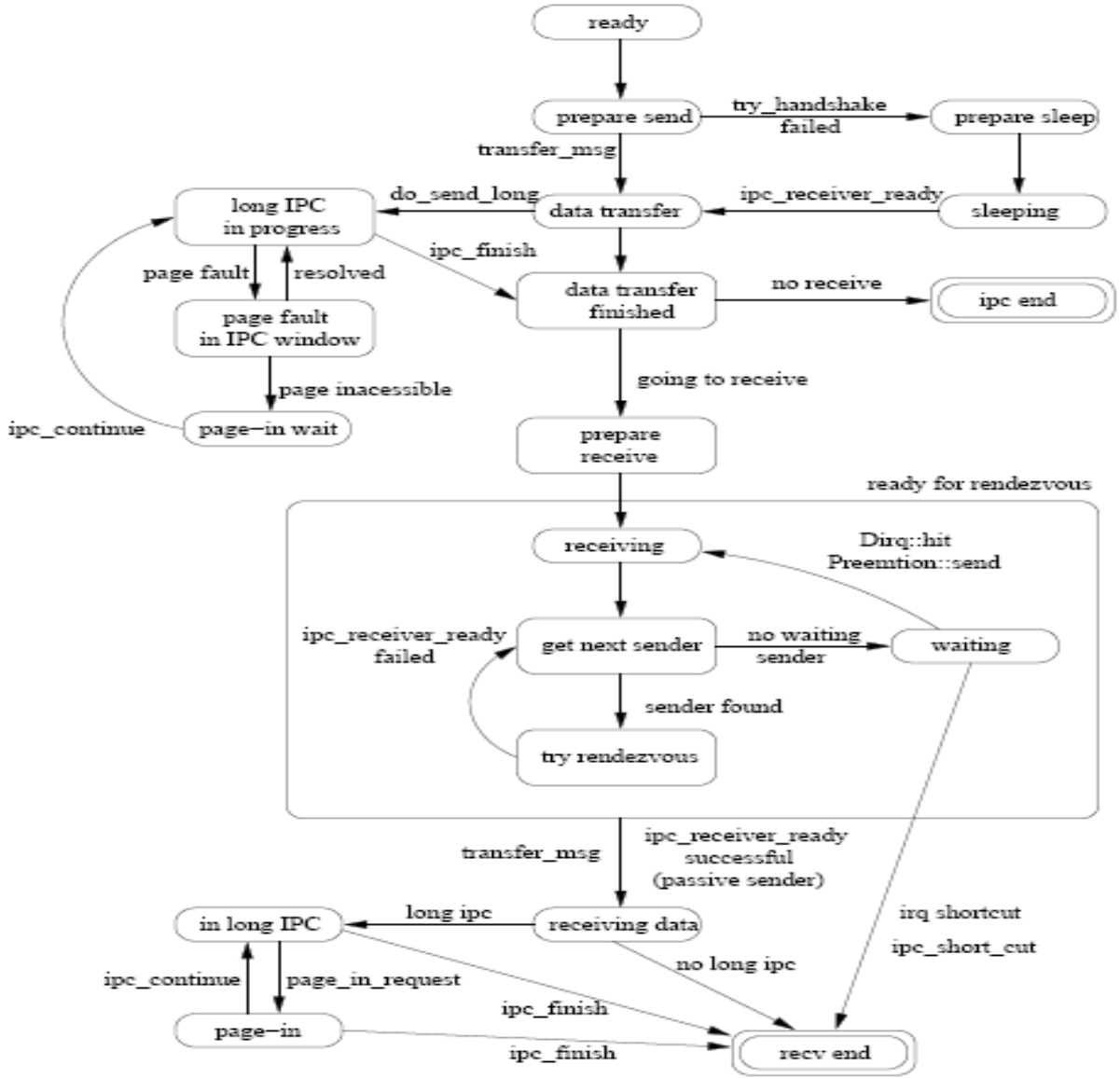


active state

passive state

state transition

Example: Fiasco IPC State Diagram



Torsten
Frenzel

TU Dresden
Operating
Systems Group

Fiasco IPC Performance - Pingpong

- Artificial micro-benchmark
- Measures time for round-trip IPC between two threads
- Best-case scenario with hot TLB and caches

CPU: Intel Core 2 at 2663 MHz
...
Register only IPC and both threads are in the same address space:
int30: **1441** cycles, 541 ns
sysenter: **448** cycles, 168 ns
...
Register only IPC and both threads are in different address spaces:
int30: **1962** cycles, 736 ns
sysenter: **895** cycles, 336 ns
...
UTCB IPC and both threads are in the same address space:
4 dwords (16 B): **1938** cycles, 727 ns
8 dwords (32 B): **2030** cycles, 763 ns
...

Torsten Frenzel

TU Dresden
Operating Systems Group

IPC and Scheduling

- Reminder: threads have priorities
- Scheduling decision after finishing the send phase:
  - Wake up (old) receiver?
  - Wake up new sender?
  - Thread with highest priority should run
- Scheduling decision after finishing the receive phase:
  - Wake up (old) sender?
- IPC can cause priority inversion
  - Sender queue has to be priority sorted
  - Priority inheritance limits priority inversion
  - Blocked high priority thread can *help* ongoing IPC
  - Implementation of helping is tricky!
- More to come in scheduling and synchronization lessons

Torsten Frenzel

TU Dresden
Operating Systems Group

IPC and Synchronization

- Reminder: IPC can be preempted
  - Long IPC has always interrupts enabled
  - On multi-processor systems operations are really parallel
- Concurrent cancel of ongoing IPC is allowed
  - Just an atomic bit switch in the state flag
  - Check state when canceled thread resumes operation
- Concurrent deletion of partner thread is **not** allowed
  - Deletion removes the TCB and even the address space
  - Invalid thread pointer would corrupt the kernel
  - Validation of partner thread after every possible preemption
- Receiver thread is locked during IPC
  - Locked thread cannot be deleted
  - Lock is only released during page fault handling in long IPC
  - Partner thread needs to be validated

Torsten Frenzel

TU Dresden Operating Systems Group

What's Left

- Special types of IPC:
  - Exception IPC
  - Preemption IPC
- Fast paths for special cases
  - Assembler short cut (hand optimized)
  - Short cuts
- Complete state protocol of the IPC path
  - Too much states and state transitions
  - Validation of the IPC path is currently out of scope
- Controlling IPC is crucial for security
  - Communication channel between isolated subsystems
  - Control of 'who can send information to whom' is crucial
- More to come:
  - Scheduling
  - Synchronization

Torsten Frenzel

TU Dresden
Operating Systems Group

Summary

- IPC is flexible and powerful
  - Has many use cases
  - Typed data items allow mapping of resources
  - Many problems can be solved with IPC
- IPC is optimized
  - Combined send-and-received IPC primitive
  - Avoiding unnecessary copy operation for data
  - Avoiding scheduling overhead
- IPC is complex
  - Complex protocol and state transitions
  - Many different cases for termination
  - Page faults during IPC result in nested IPC
  - Requires synchronization with other operations

Torsten Frenzel

TU Dresden
Operating Systems Group