Microkernel Construction

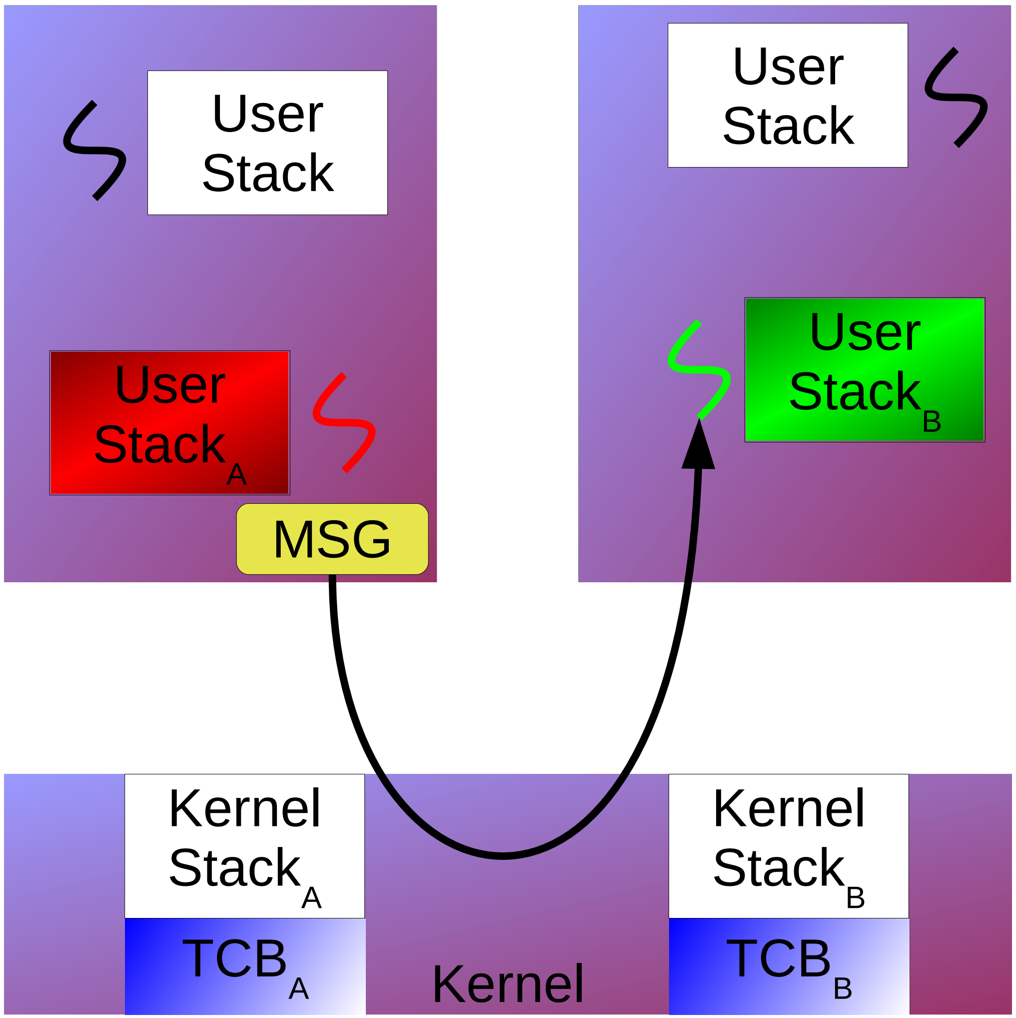
Capabilities

SS2013

Benjamin Engel

TU Dresden Operating Systems Group

Recap: Sending Messages



Kernel Objects

- Sender thread
- Receiver thread
- Sender address space
- Receiver address space
- IPC portal/endpoint

Benjamin Engel

TU Dresden Operating Systems Group

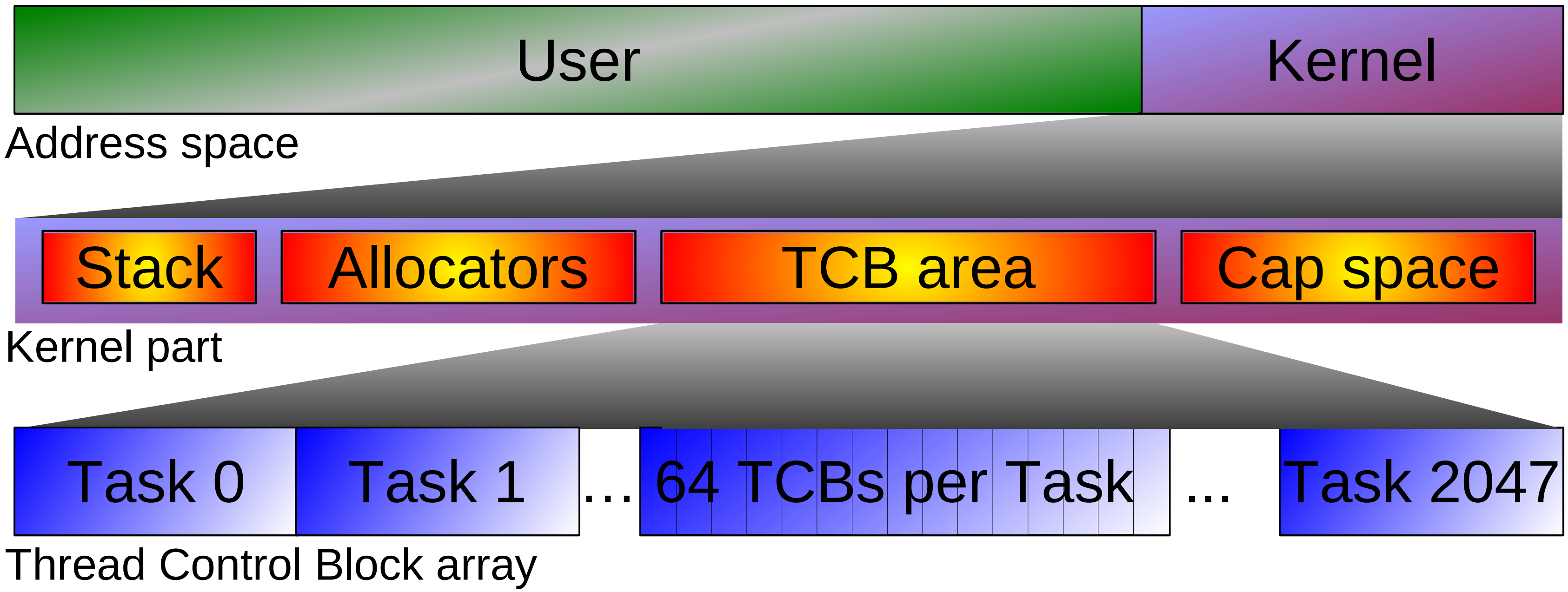
- How do we know whom to talk to
  - Find / name / lookup the receiver
  - Does the receiver knows who has sent a message?

- How to restrict communication
  - Can everybody send messages to everyone?
  - What can be communicated (restrict message payload)?

Task and Thread IDs

- Global IDs: 32bit word with Task_ID and Thread_ID

  - 2048 tasks (11 bit), each 128 threads (7 bit)
  - In-Kernel TCB area: 1KB Thread Control Block per thread
    → get directly TCB address from global ID (mask out version)
  - Sender/Receiver_ID & 0x0FFFFC00 + TCB_area_base_address



Benjamin Engel

TU Dresden Operating Systems Group

Global (Task/Thread) Names

- Sending to any thread possible, just pick the right task/thread ID → malicious thread can send garbage and thus disrupt others
- Difficult to manage system with global names
  - e.g. file system always task 3
  - Now we will have two file systems, well ...
  - What if this changes, reconfigure whole system ...
  - How to know which task IDs are unused/available?

1. Global names are not a good idea

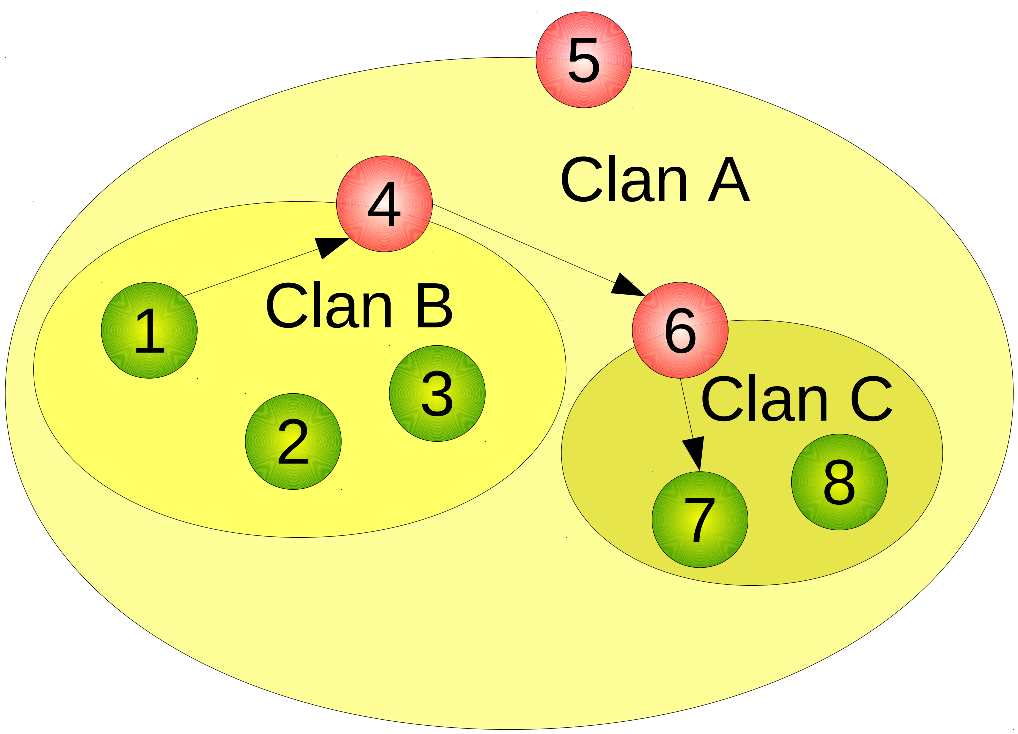
2. We want to restrict communication channels

Benjamin Engel

TU Dresden Operating Systems Group

Clans & Chiefs

- Addressing 2nd issue first: restrict IPC
- Direct communication within clans (group of tasks) only
- Crossing clan boundaries → kernel redirects IPC to chief, which might drop/modify/forward the message



- Send 1 → 7
- Kernel: thread 1 and thread 7 in different clans → send to 4
- If communication allowed by chief (4), forward to 6
- 6 delivers message finally to thread 7

Clans & Chiefs

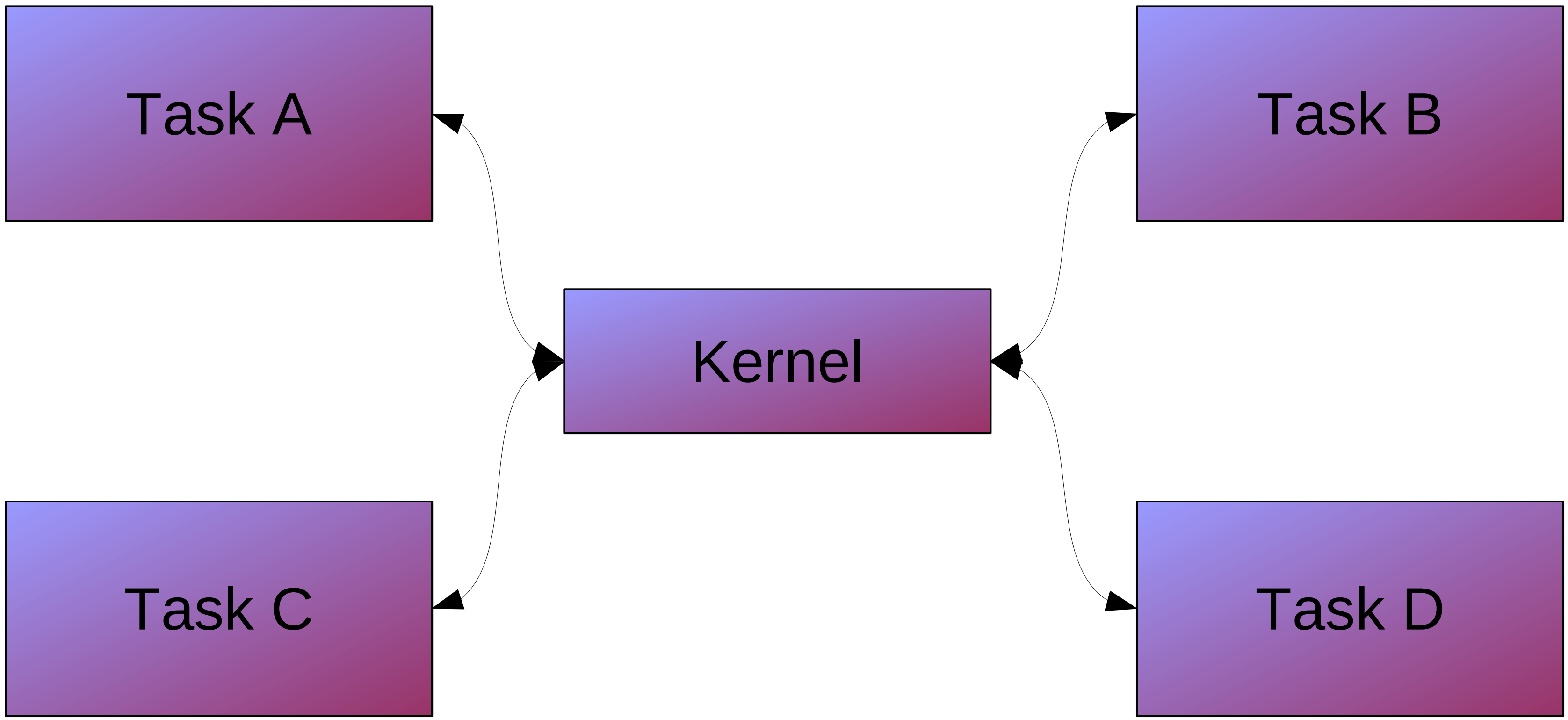
- Fine grained control
- Strictly hierarchical tree
  - Tasks cannot be part in two clans at the same time
  - “Wrapper” clans to express that subsystems may communicate
- Chief has full control over its minions
  - Drop messages entirely
  - Arbitrary modify message
  - Forward it to own chief
- High overhead (one IPC becomes multiple ones)

Benjamin Engel

TU Dresden Operating Systems Group

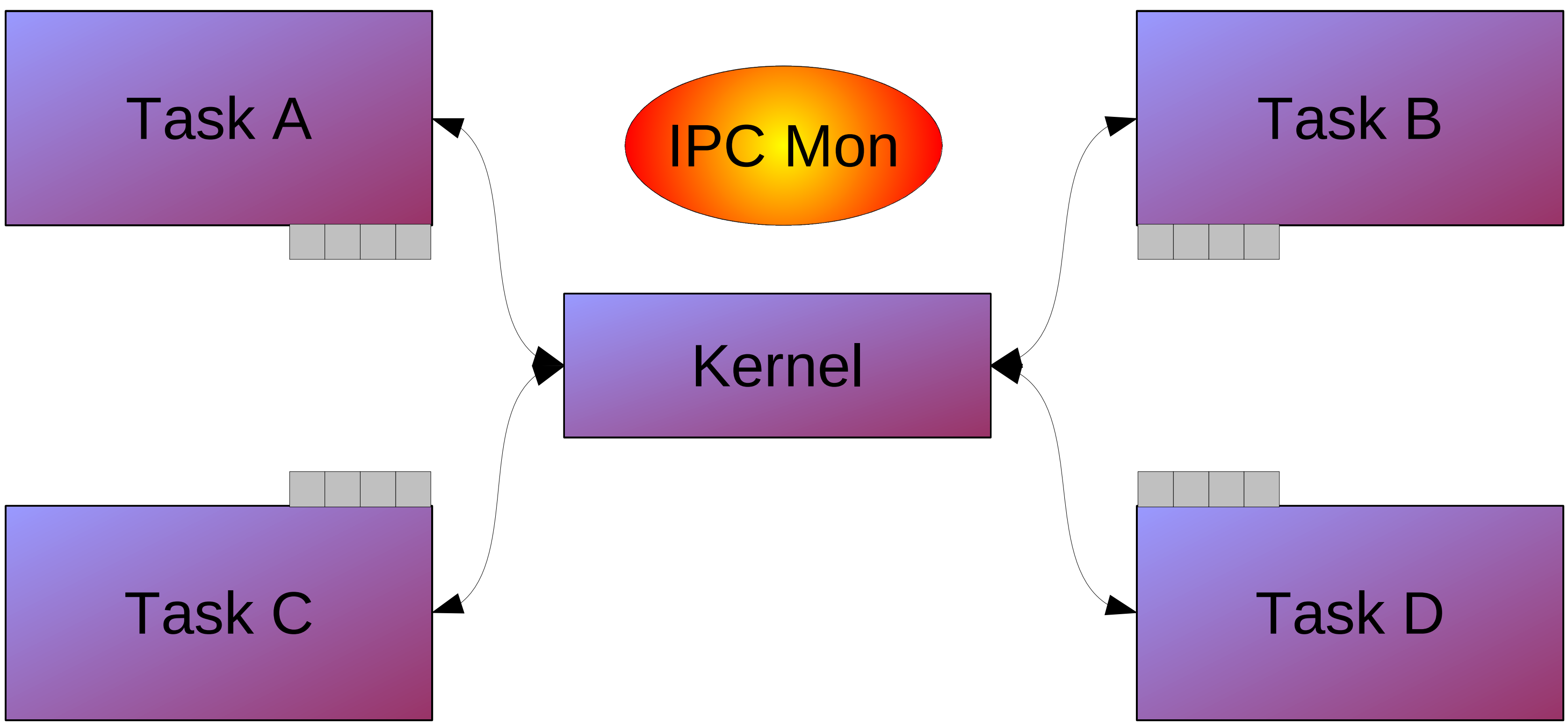
Non-hierarchical IPC between Tasks

- Tasks send messages to each other through the kernel
- Kernel does the actual work: lookups, security checks and finally the copy operation
- How to decide if a send operation is allowed?



IPC Permission Bitmap

- Bitmap attached to a task, masking to which other tasks it is allowed sending messages (compare x86 IO bitmap)
- Modifying this bitmap is a security critical operation → special privileged IPC Mon task
- Kernel checks IPC bit for every send operation

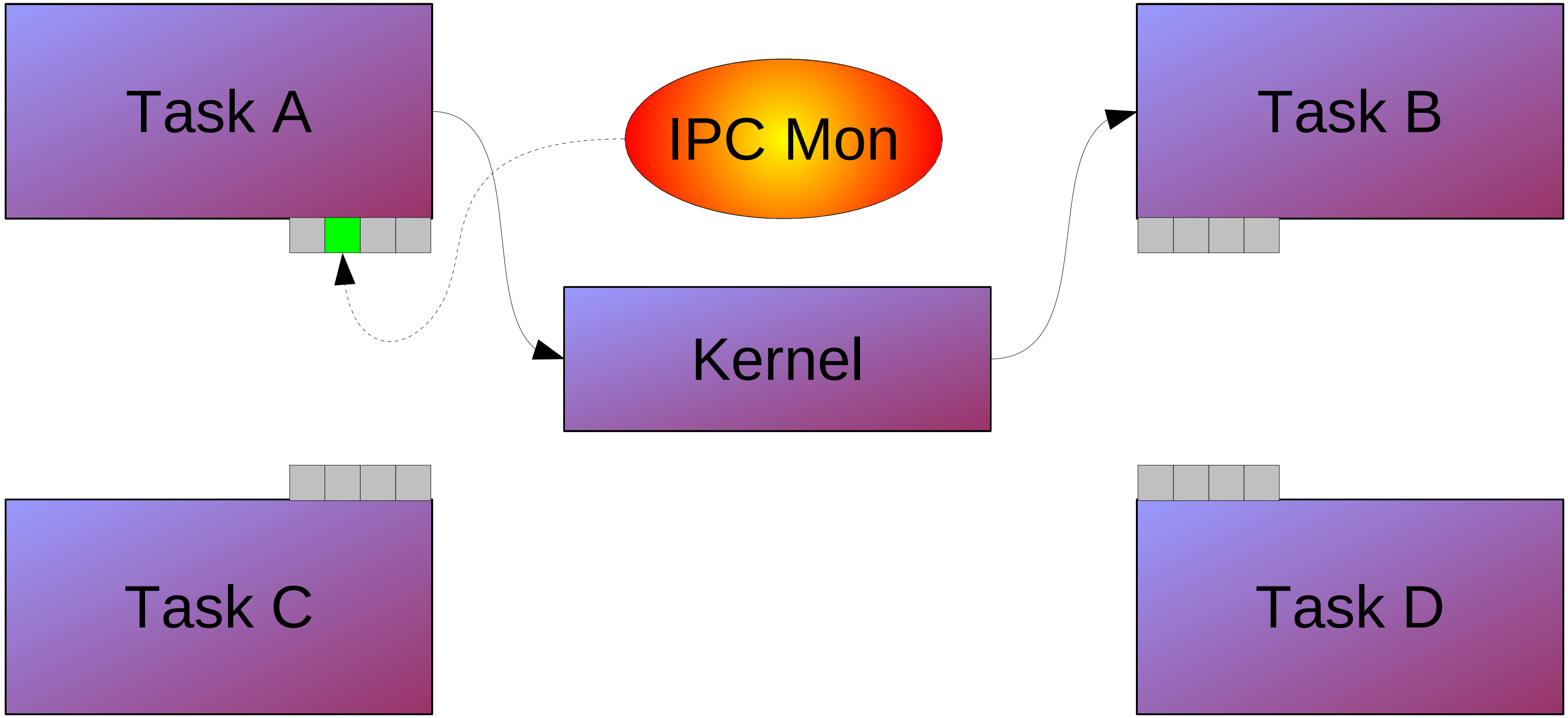


IPC Mon

- Dedicated, privileged task with write access to IPC bitmap
- Single and ultimate policy maker
- Very special, has to know all tasks (again: global names)
- More expressive than clans & chiefs, very fine grained control, effectively a matrix: who can talk to whom
- Binary decision only: Task A can send messages to Task B or not, no means to further restrict *what* to send

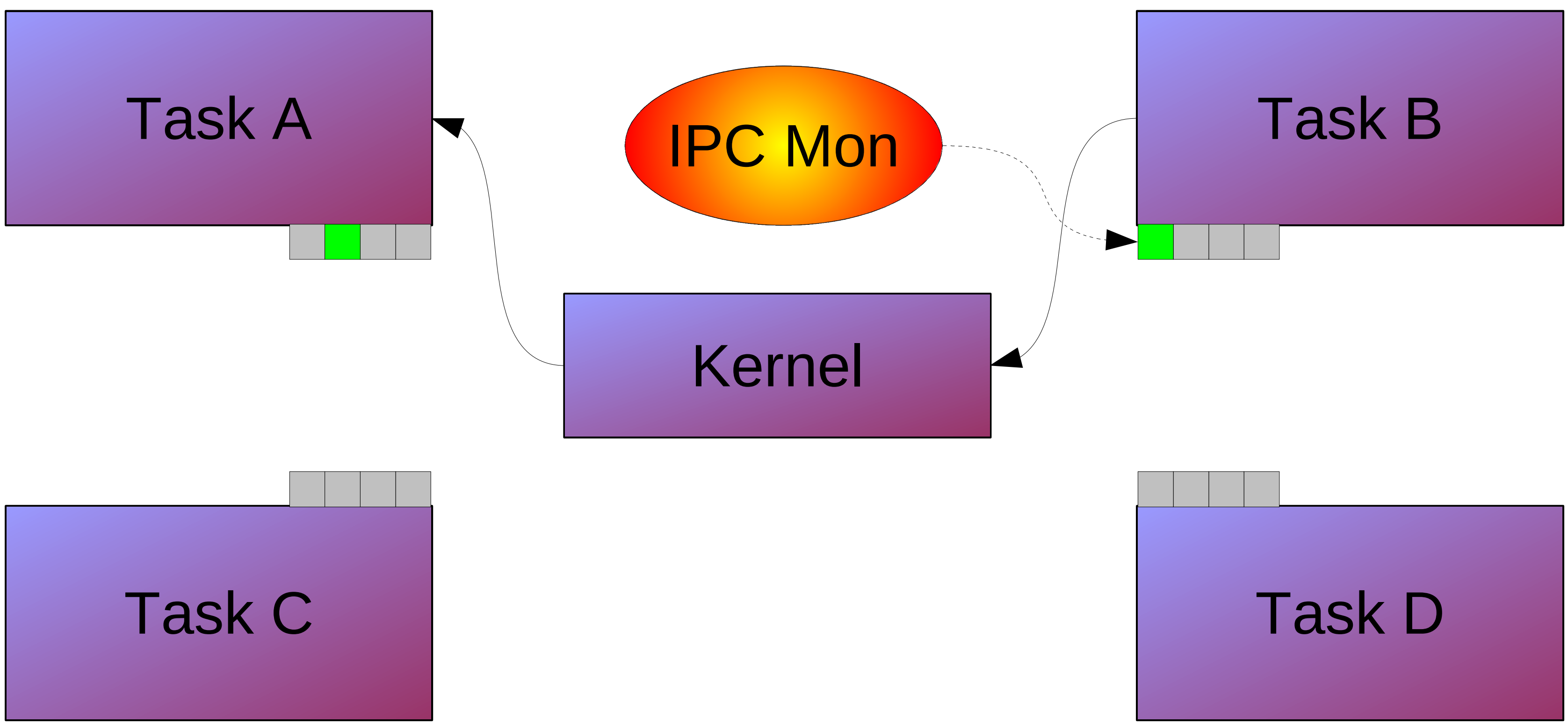
IPC Mon: Examples

- A → B: ✔ in task A's bitmap bit for task B is set



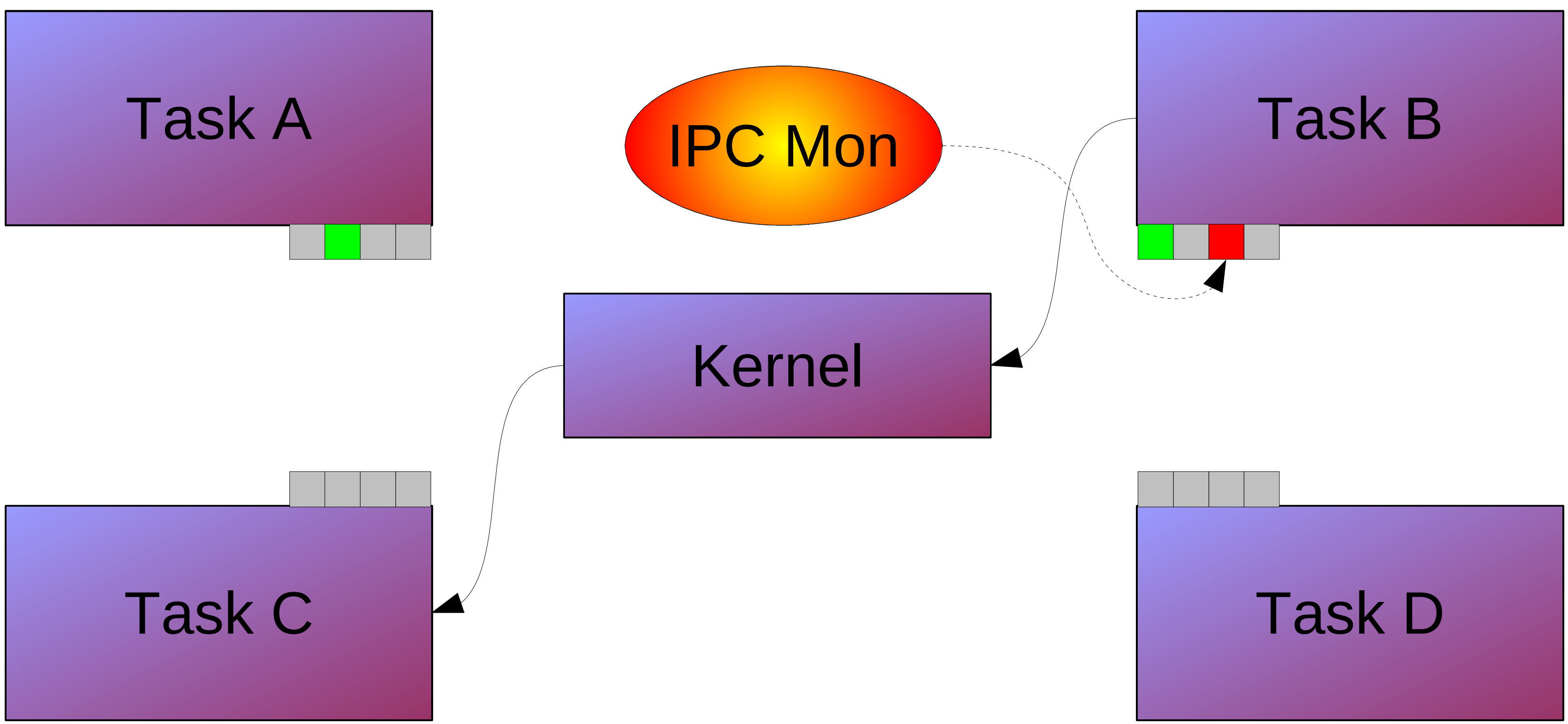
IPC Mon: Examples

- A → B: ✔ in task A's bitmap bit for task B is set
- B → A: ✔ in task B's bitmap bit for task A is set



IPC Mon: Examples

- A → B: ✔ in task A's bitmap bit for task B is set
- B → A: ✔ in task B's bitmap bit for task A is set
- B → C: ✘ no, bit is not set, kernel will abort IPC

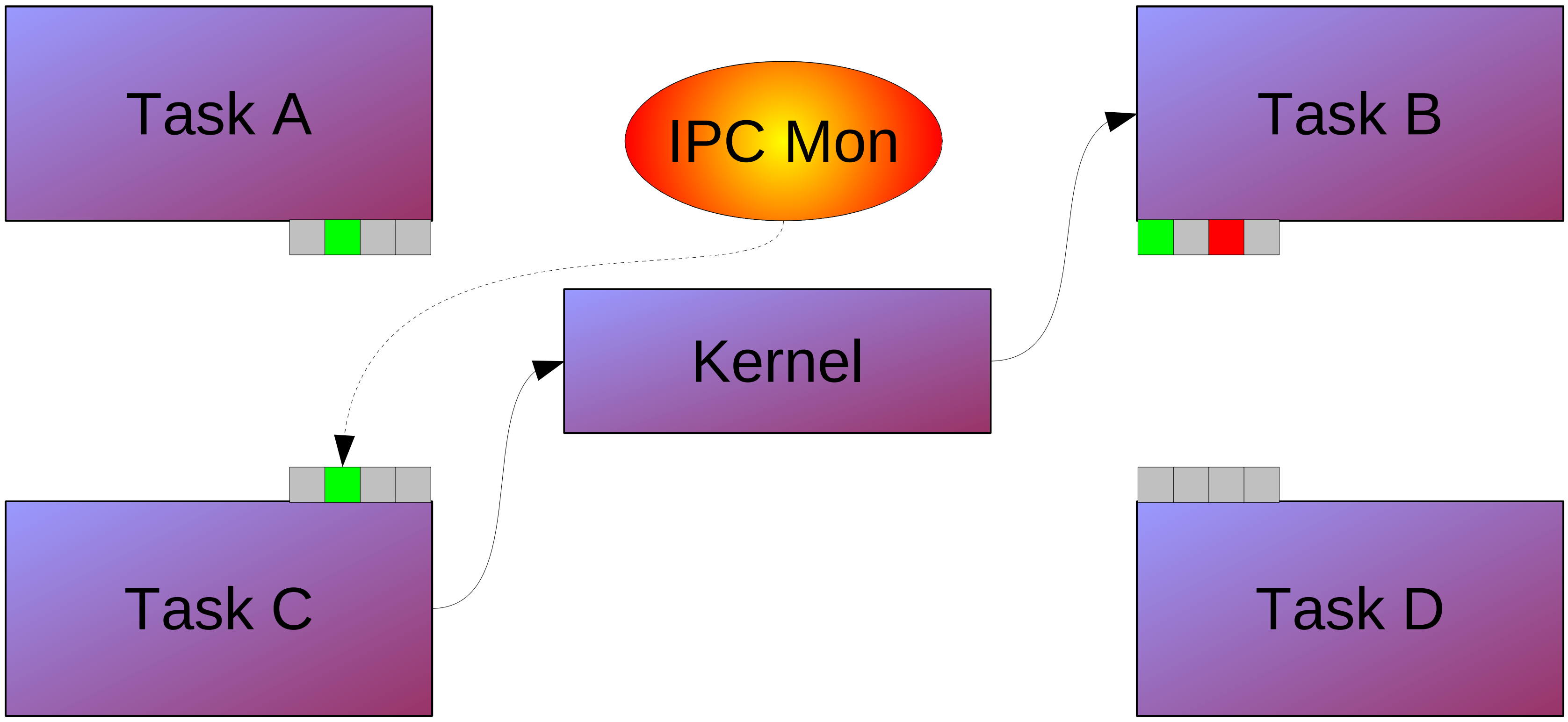


Benjamin
Engel

TU Dresden
Operating
Systems Group

IPC Mon: Examples

- A → B: ✔ in task A's bitmap bit for task B is set
- B → A: ✔ in task B's bitmap bit for task A is set
- B → C: ✘ no, bit is not set, kernel will abort IPC
- C → B: ✔ allowed, bit is set

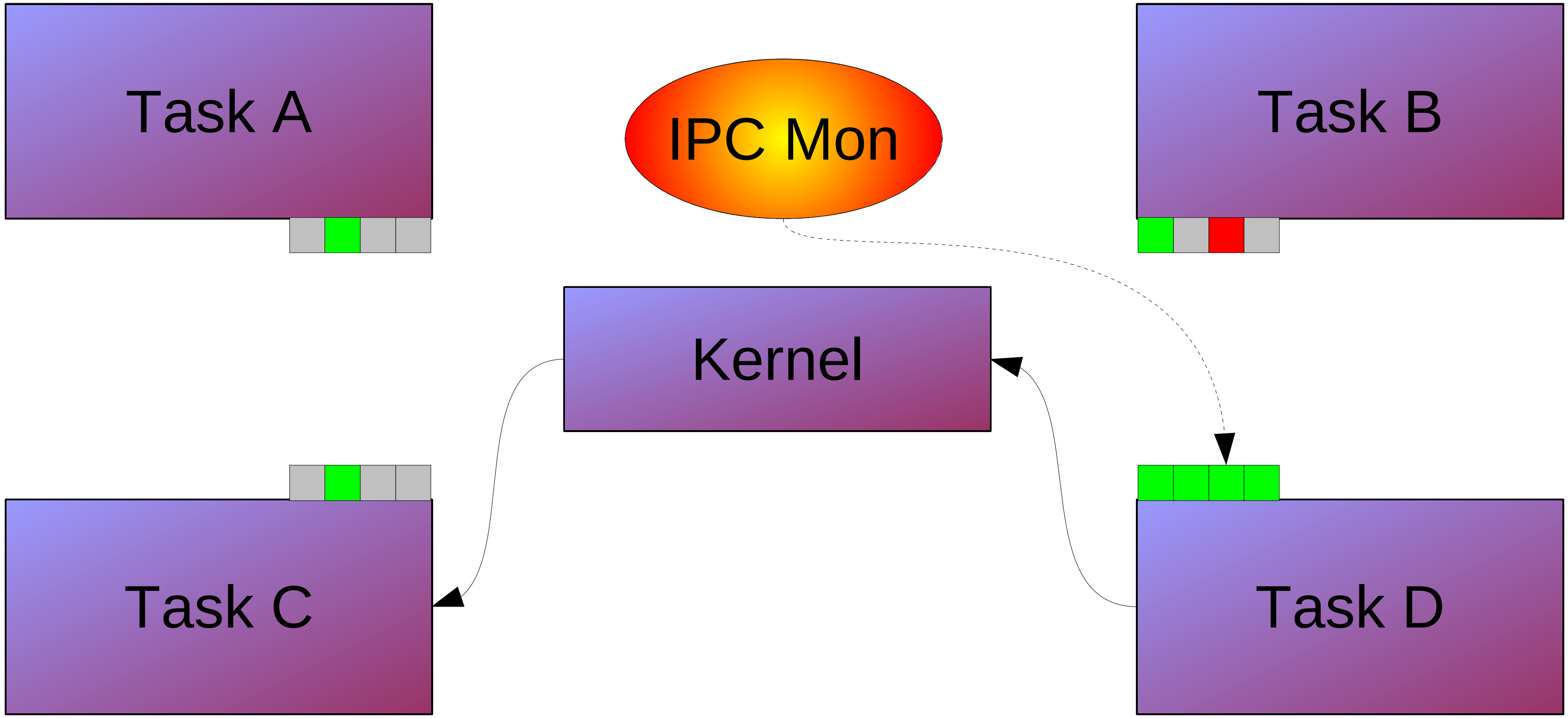


Benjamin
Engel

TU Dresden
Operating
Systems Group

IPC Mon: Examples

- A → B: ✔ in task A's bitmap bit for task B is set
- B → A: ✔ in task B's bitmap bit for task A is set
- B → C: ✘ no, bit is not set, kernel will abort IPC
- C → B: ✔ allowed, bit is set
- D → C: ✔ all bits are set, thus D can send everywhere



Benjamin Engel

TU Dresden Operating Systems Group

IPC Permission Bitmap

- Indirection layer allowing to have fine-grained control on IPC communications
  → Send might fail with an exception

- Per address space permissions
  - Does is make sense to have per thread permissions? Why not?

- Single bit per Task × Task

- But still: omniscient task (typically sigma0) knowing all existing tasks and their IPC policy
  → *we got to get rid of these global names*

Benjamin
Engel

TU Dresden
Operating
Systems Group

Local Names

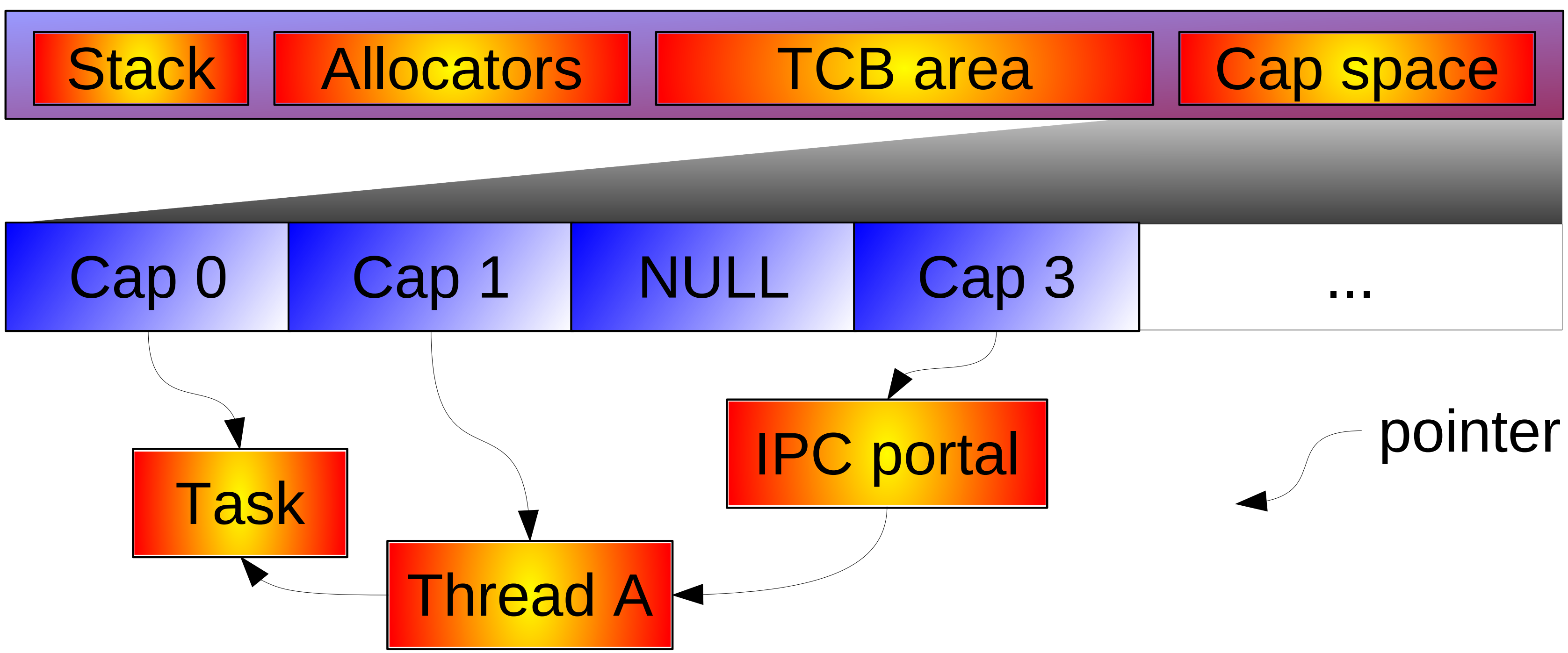
- We don't need global names (task/thread IDs)
- Names (or IDs) are only valid within a task and have no meaning elsewhere
- Kernel objects are referenced through local IDs, comparable to POSIX file descriptors or handles
- Creating a new (kernel) object returns an index into a task-local table, where in turn the pointer to the object is stored
- Kernel protects this capability table, therefore unforgeable

Benjamin
Engel

TU Dresden
Operating
Systems Group

Capability Space

- In-Kernel memory table with pointers to kernel objects
- Sending a message to thread A merely requires the sender to have a capability to the portal cap, here cap 3
- Sender does not know which thread/task will receive it
- Receiver does not know who sent it (in general)
- Separation of subsystems, combinable, independent



Benjamin Engel

Capabilities to Kernel Objects

- Objects involved sending a message:
  - 2 threads (sender and receiver)
  - 2 tasks (address space of these)
  - 1 IPC endpoint

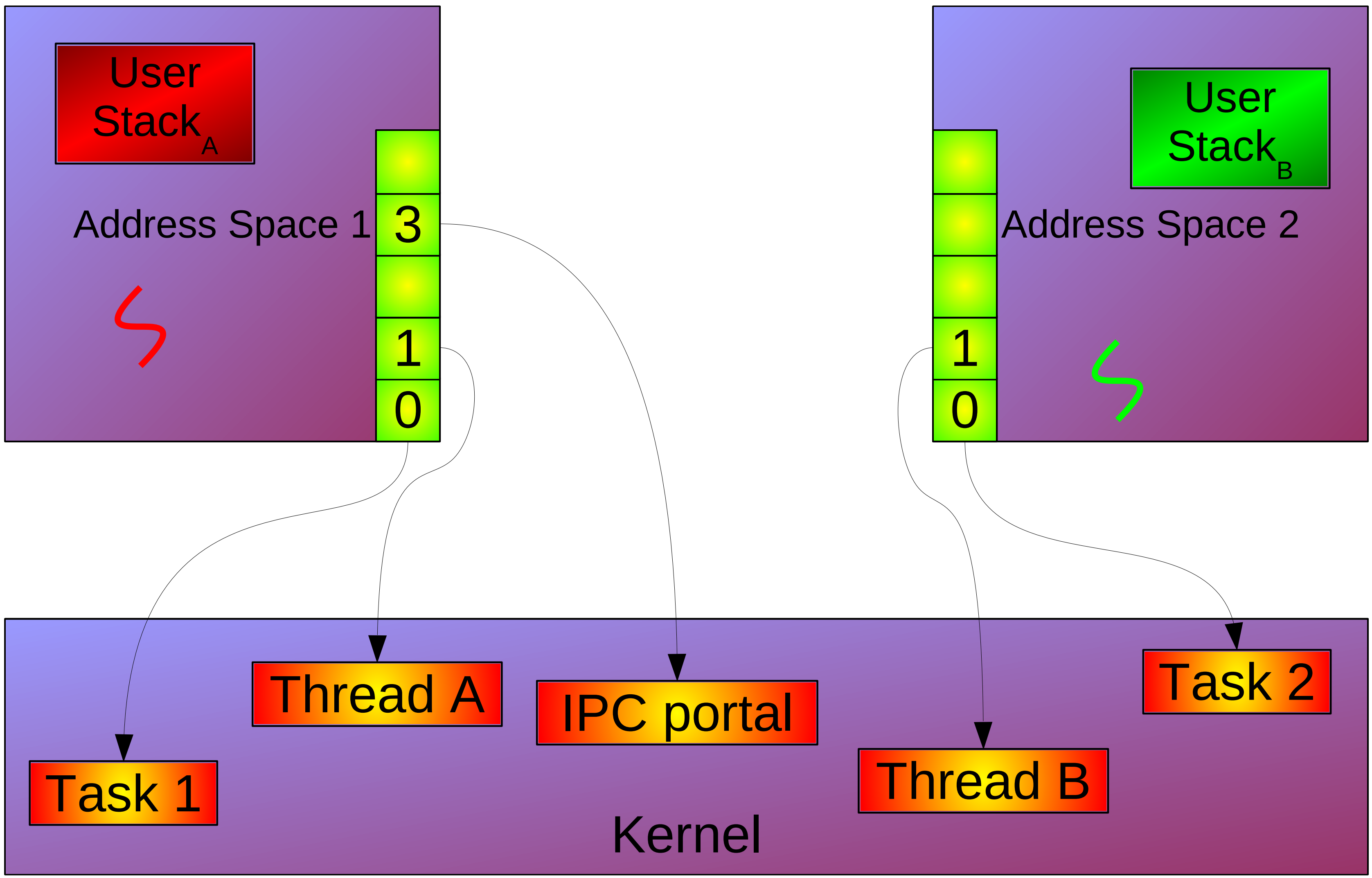
- **Thread**: register state, link to it's task (address space)
- **Task**: page tables, hardware resources
- **IPC endpoint**: reference to receiving thread

- Further kernel objects
  - Semaphores
  - Scheduling contexts (time abstraction, used for scheduling)
  - Factories (creation of new objects)
  - ...

Benjamin
Engel

TU Dresden
Operating
Systems Group

Kernel Object Capabilities

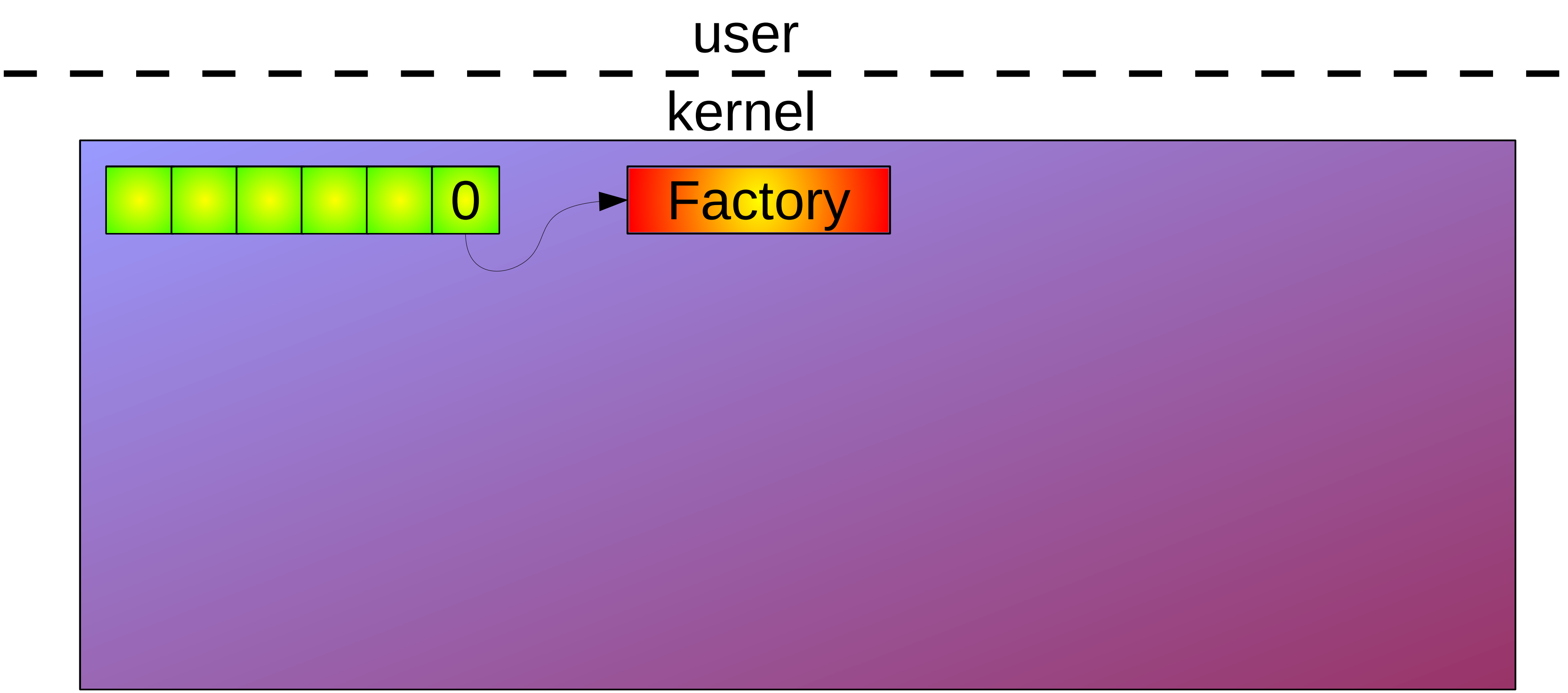


Benjamin
Engel

TU Dresden
Operating
Systems Group

Kernel Objects in Concert

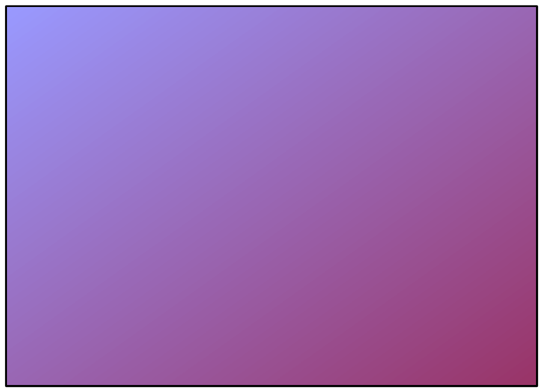
- Initially there is only a factory
  - Kernel object used to create new kernel objects of various types (tasks, threads, IPC portals, Semaphores, more Factories, ...)



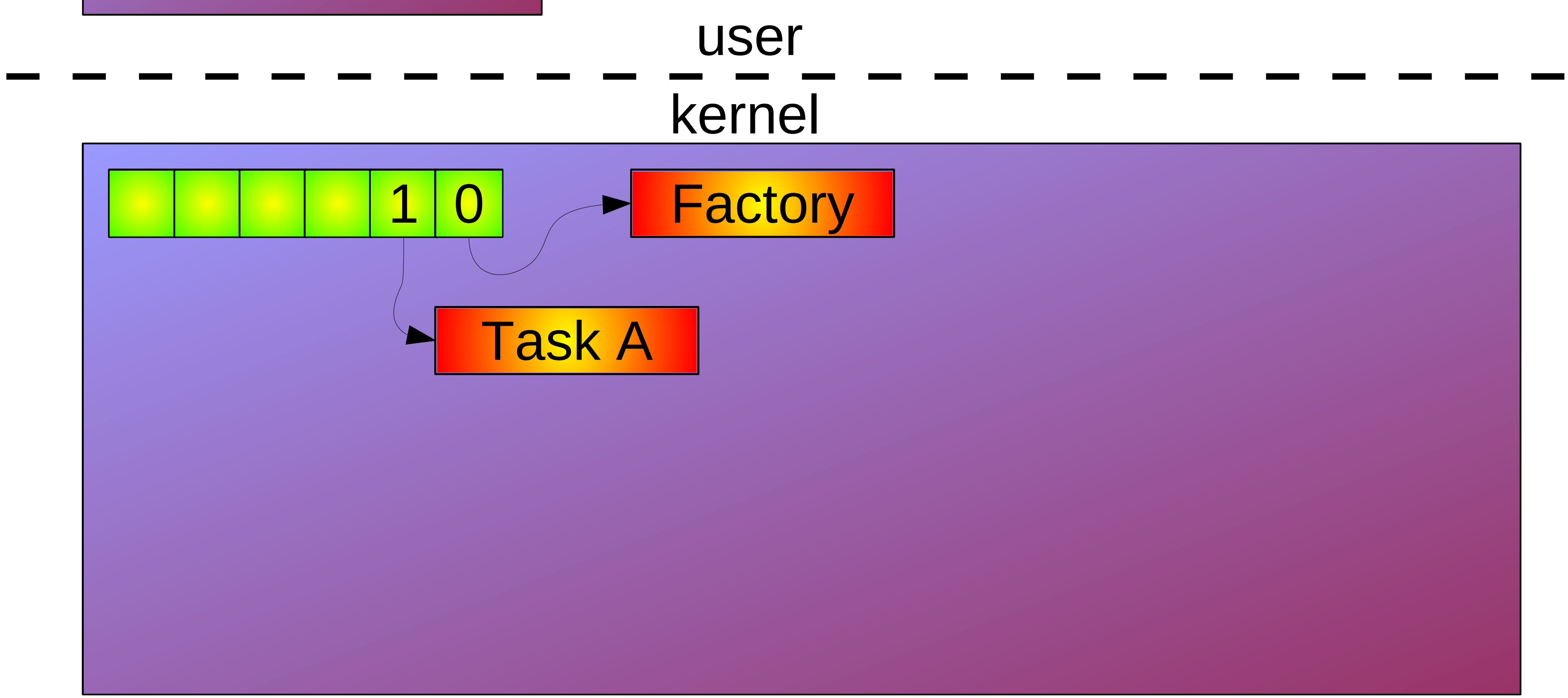
Benjamin
Engel

TU Dresden
Operating
Systems Group

Kernel Objects in Concert



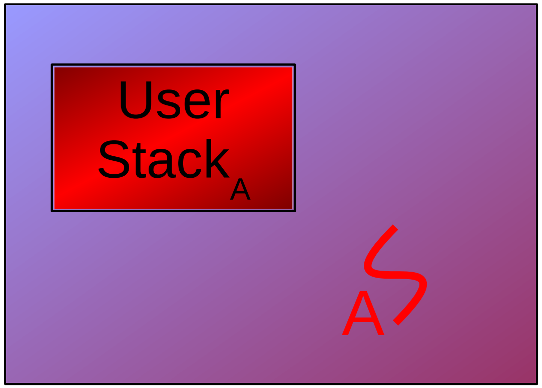
- Create a new task using the factory
- Return a new capability to this task



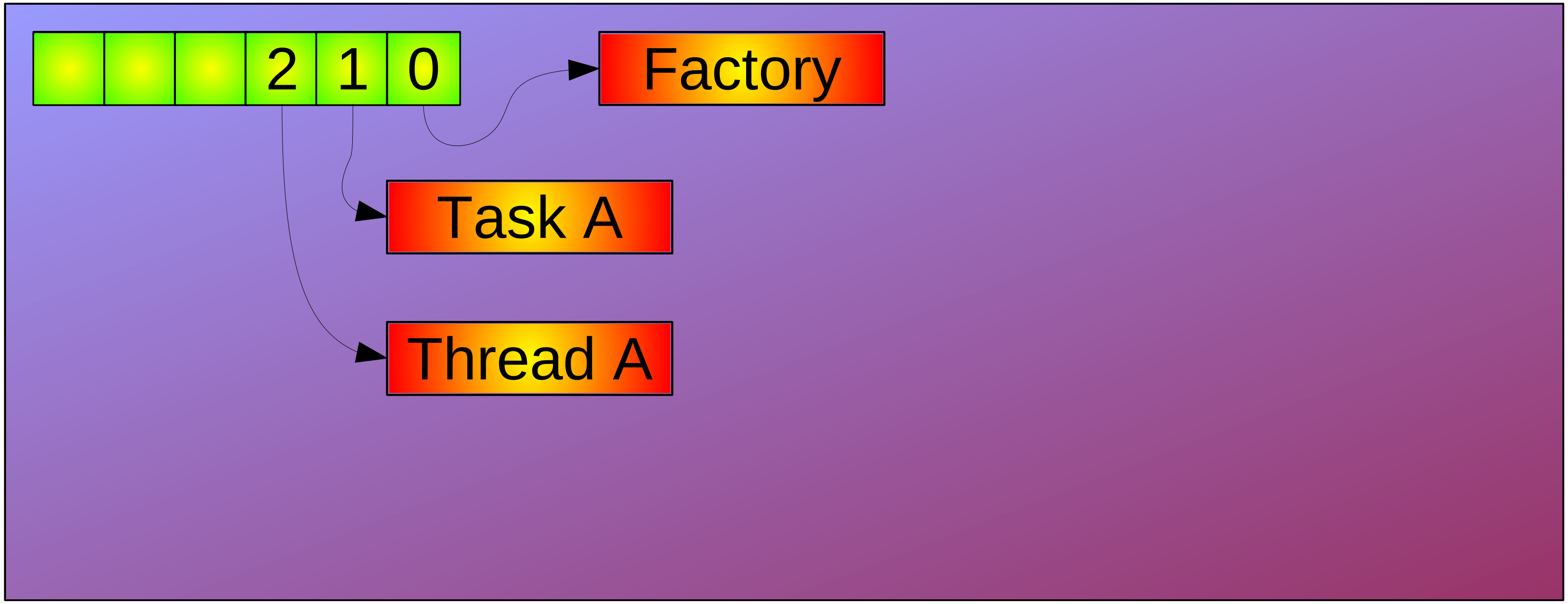
Benjamin
Engel

TU Dresden
Operating
Systems Group

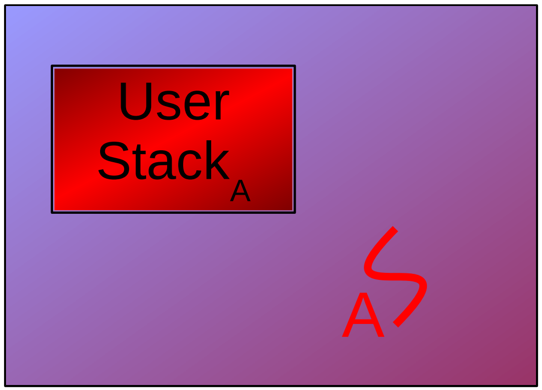
Kernel Objects in Concert



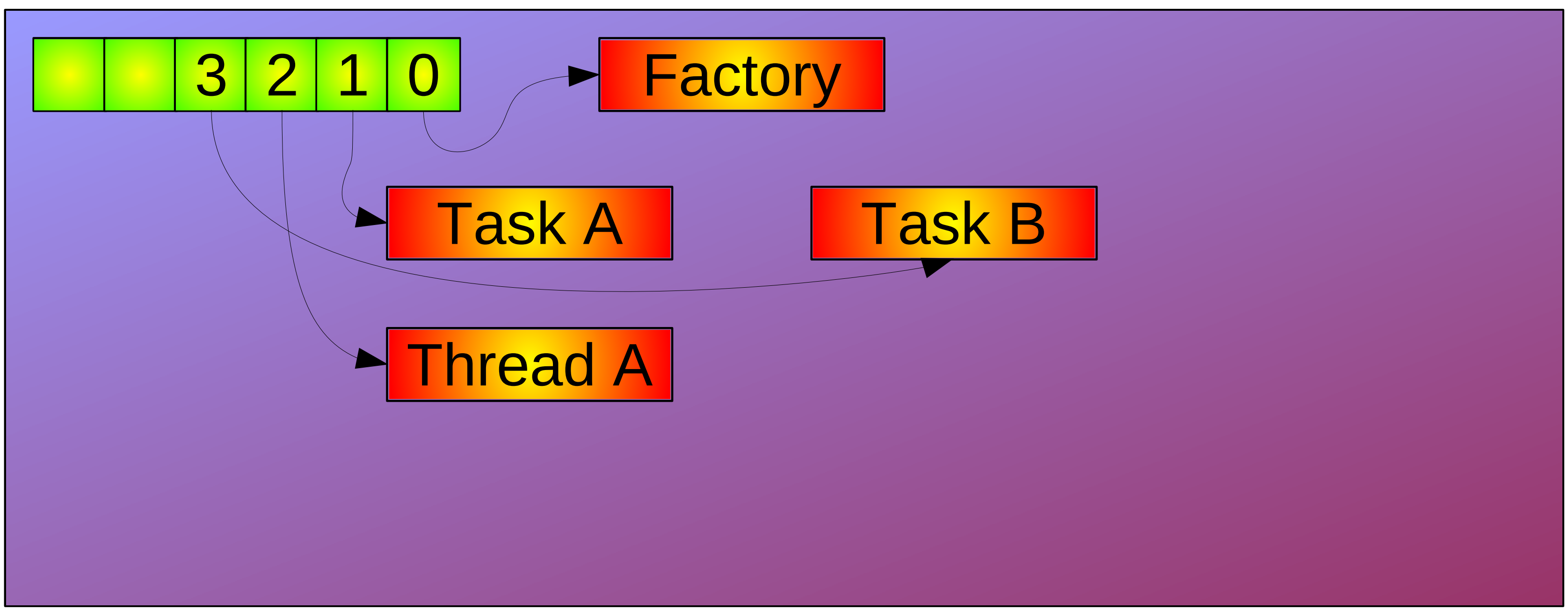
- Within the newly created task, a first thread A is spawn
- The kernel object comprises the thread state, scheduling info, ...
- Further a user stack is needed



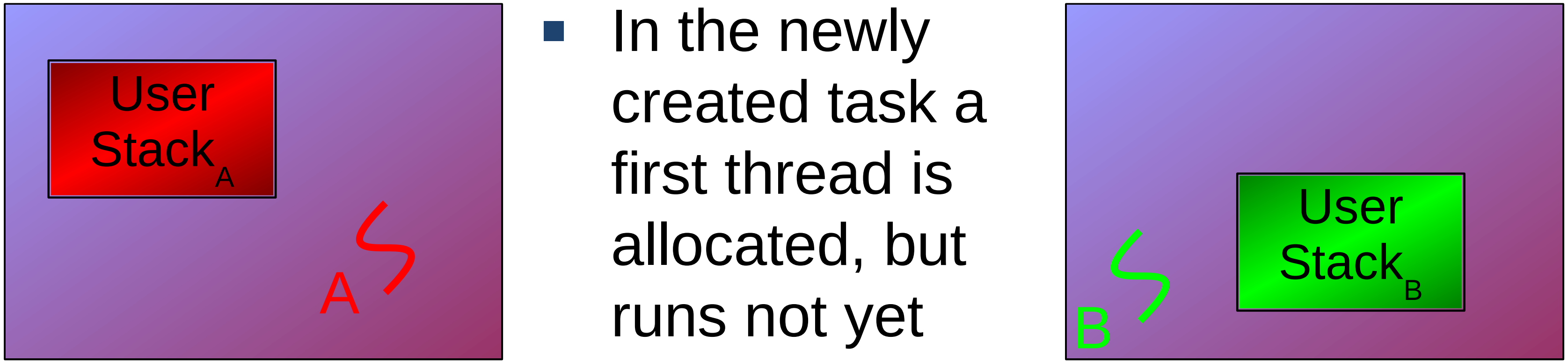
Kernel Objects in Concert



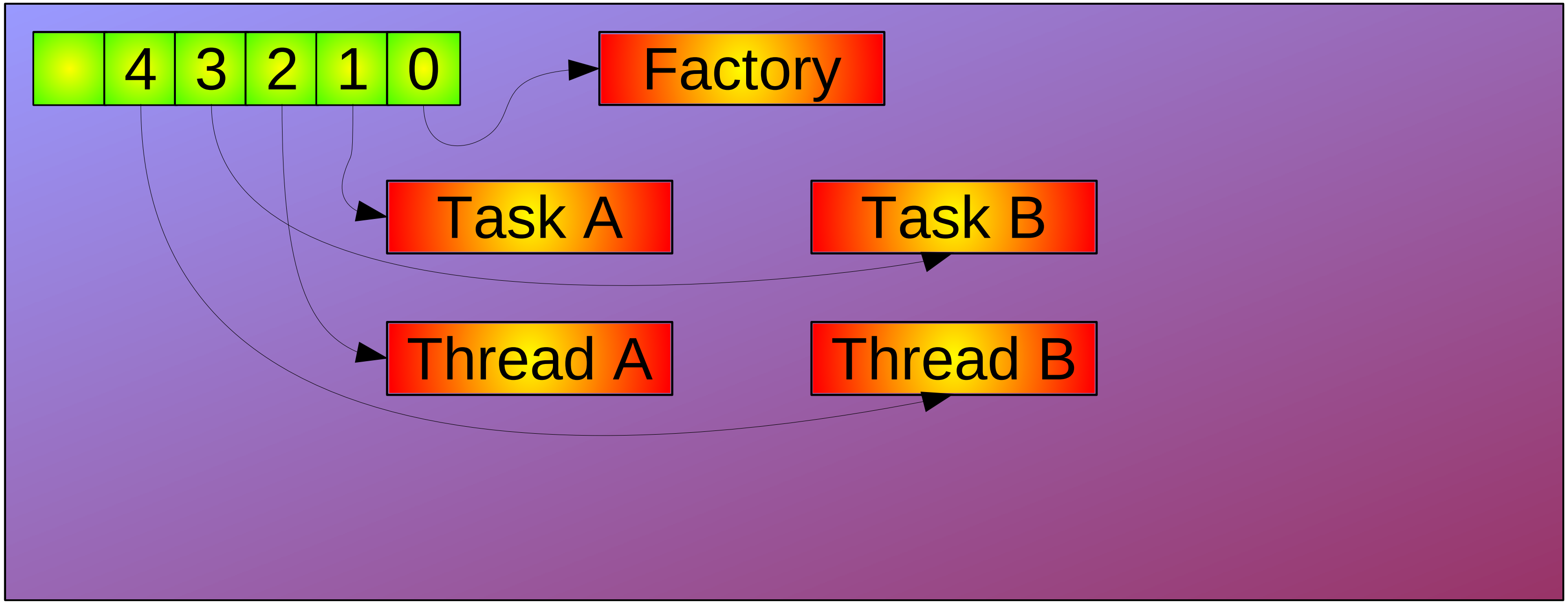
- The thread runs and creates another task B, receiving a new capability



Kernel Objects in Concert



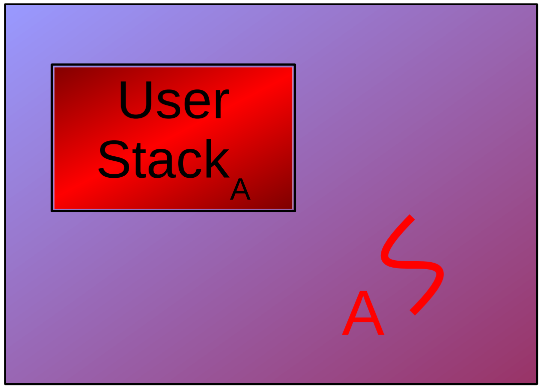
- In the newly created task a first thread is allocated, but runs not yet



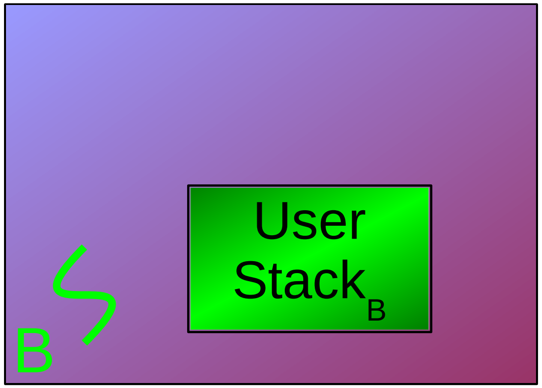
Benjamin
Engel

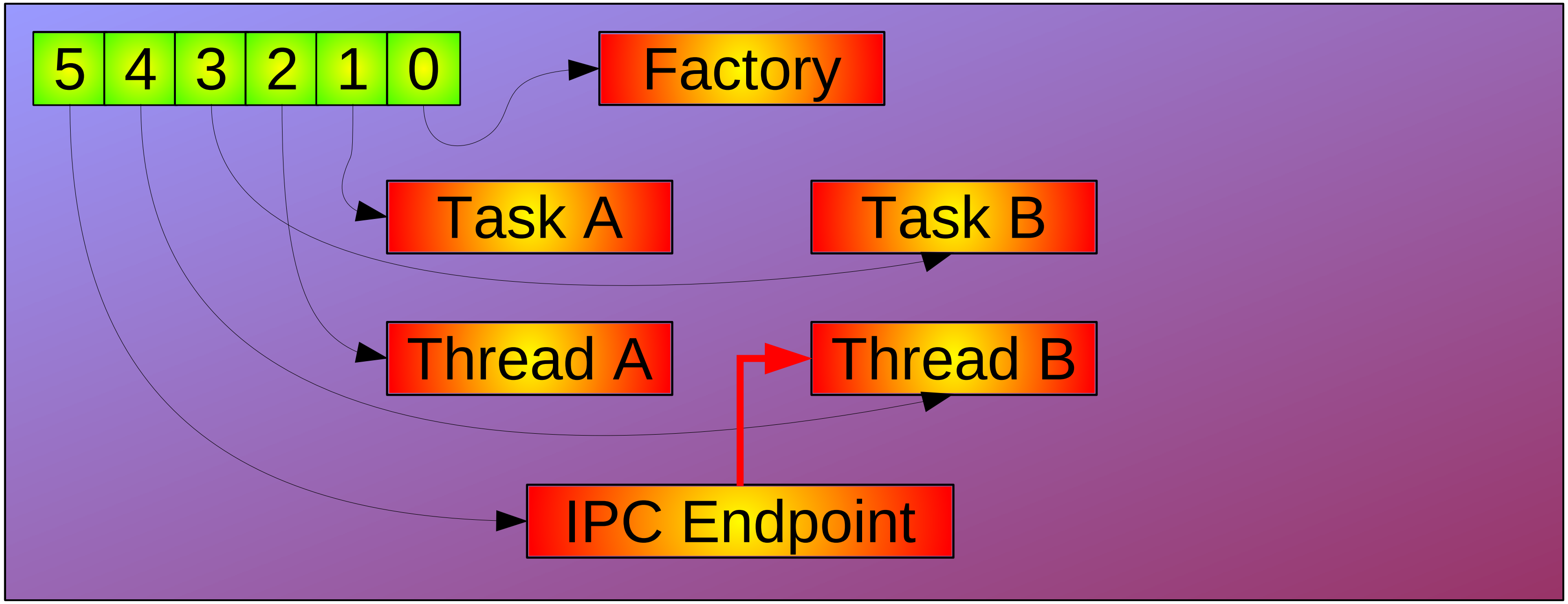
TU Dresden
Operating
Systems Group

Kernel Objects in Concert



- Thread A creates an IPC portal, pointing to thread B

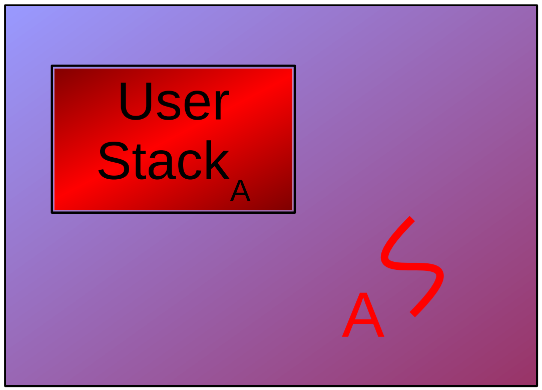




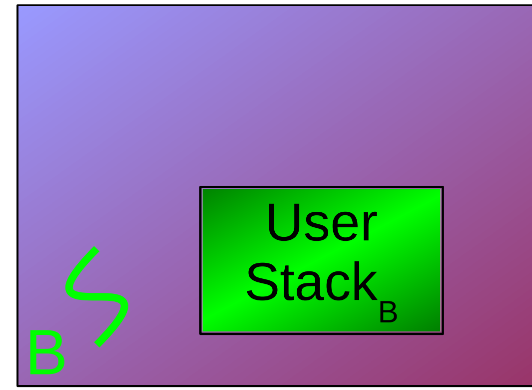
Benjamin Engel

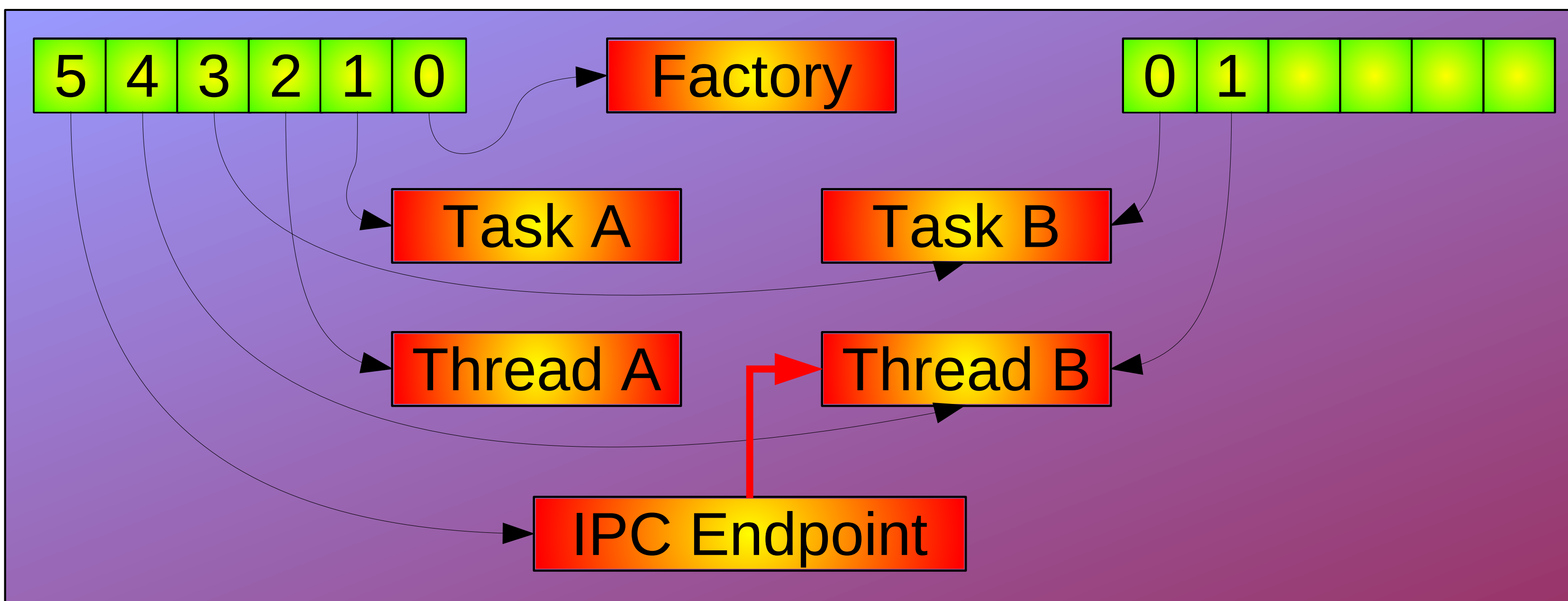
TU Dresden Operating Systems Group

Kernel Objects in Concert



- Caps for task B and thread B are mapped to B's cap space

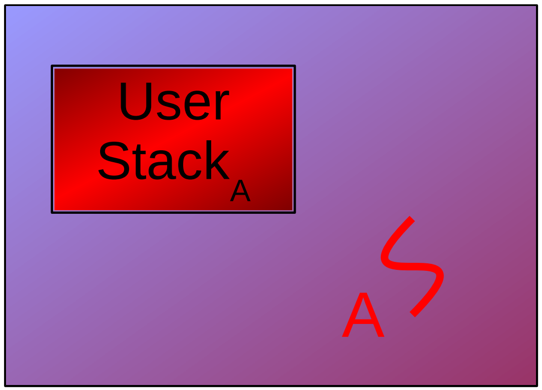




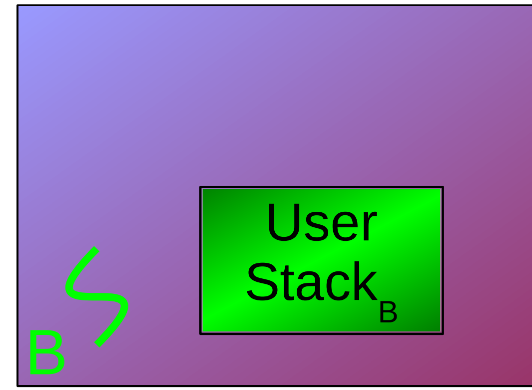
Benjamin Engel

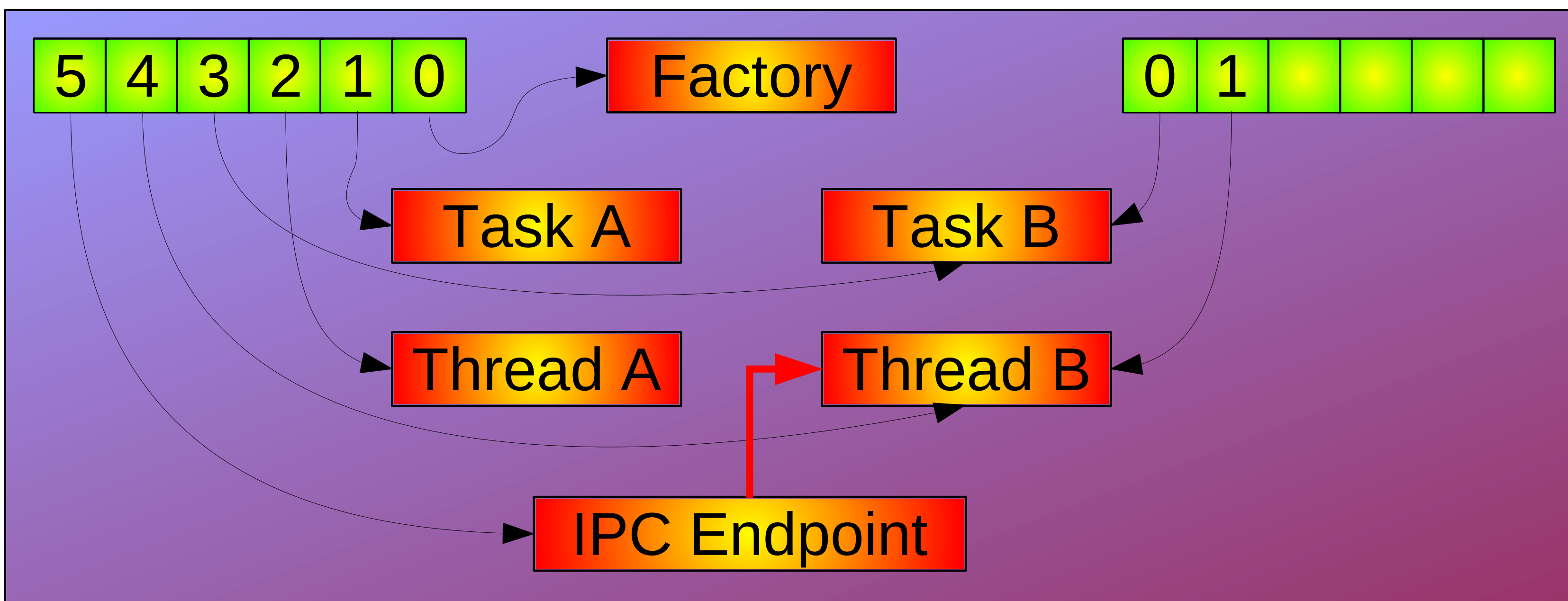
TU Dresden Operating Systems Group

Kernel Objects in Concert



- Now thread A can send messages to B through the IPC endpoint





Benjamin
Engel

TU Dresden
Operating
Systems Group

Capabilities

- Everything is a file → Everything is a capability
- Object capabilities
  - Tasks, threads, IPC portals, factories, semaphores, …
  - Handles/pointers to kernel objects, can be created, delegated and destroyed
- Memory capabilities
  - Resembles virtual memory pages
  - Sending (mapping) a memory capability established shared memory between sender and receiver
- IO capabilities
  - Abstraction for access to IO ports, delegating IO caps allows the receiving Task/Address space to access denoted IO ports

Benjamin
Engel

TU Dresden
Operating
Systems Group

Capability Implementation

- Objects are created through allocators in the kernel

- Cap space
  - per-task kernel protected area
  - Initially (almost) empty, not even paged
  - User indicates an index, kernel page faults there, maps zero page
  - Capability creation: a index to a NULL cap is required
  - Capability delegation: a pointer to an existing object is copied (mapped) to another capability space, optionally reducing rights

- sys_create_thread (cap_factory f, cap_index idx)
  - A new kernel object (thread) will be allocated
  - The pointer to this objects will be stored in the cap space at index idx

Benjamin Engel

Summary

- Local names aka capabilities facilitate fine grained access control and delegation (see mapping database)
  - Principle Of Least Authority
- Mere possession of a cap constitutes the rights to use it, no further permission checks are required nor wanted
  - Prevents Confused Deputy Problem
- Capabilities can be send over IPC (delegated)
  - Memory capabilities → establishes shared memory regions
  - IO capabilities → delegated access to IO ports
  - Object capabilities
    - IPC endpoints → establish more communication chancels
    - Factories → allow others to create new objects
    - Semaphores → facilitate cross-address space synchrinization
    - Tasks and Threads → share control on those
    - Scheduling contexts → give “time” to other components

Retrospective ...

- Threads and Address Spaces
- Kernel Entry and Exit
- Inter Process Communication
- Capabilities