Karlsruhe Institute of Technology

Microkernel Construction
3 – TCBs and Address Space Layout

Lecture Summer Term 2017
Wednesday 15:45-17:15 R 131, 50.34 (INFO)

Jens Kehne | Marius Hillenbrand
Operating Systems Group, Department of Computer Science

Recap: Thread Switch A → B

- Thread A is running in user mode
- Thread A experiences its end of time slice or is preempted by a (device) interrupt
- We enter kernel mode
- The microkernel saves the status of thread A on A's TCB
- The microkernel loads the status of thread B from B's TCB
- We leave kernel mode
- Thread B is running in user mode

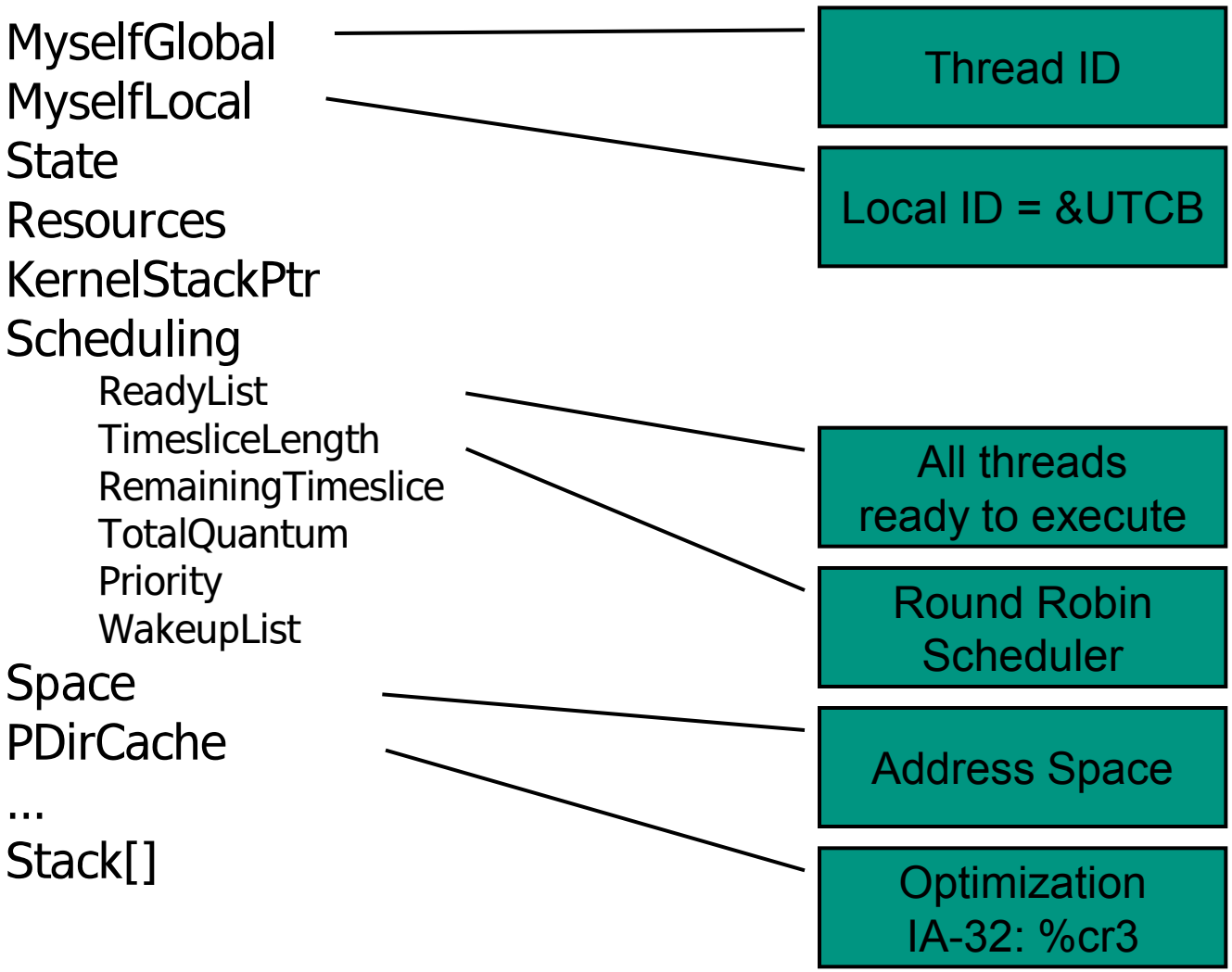
Recap: Thread Switch

- Thread state must be saved/restored on thread switch
- We need a Thread Control Block (TCB) per thread
- TCBs must be kernel objects
  - TCBs implement threads
- We often need to find
  - Any thread's TCB using its global ID
  - The currently executing thread's TCB (per processor)

At least partially. We have found some good reasons to implement parts of the TCB in user memory (→ IPC).

THREAD CONTROL BLOCKS (TCBS)

TCB Structure

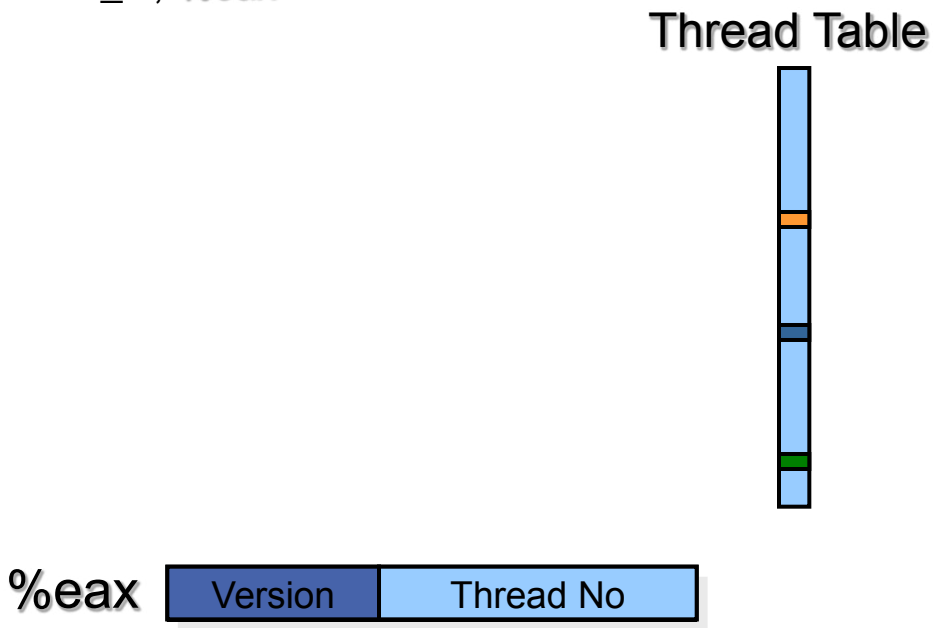


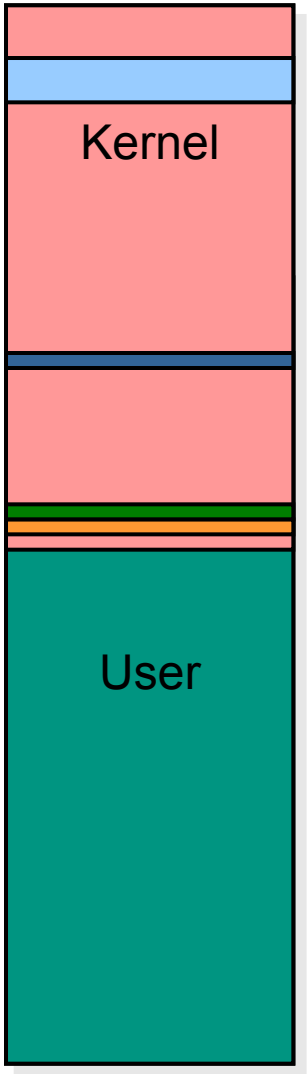
Thread ID

- Thread number
  - To find the TCB
- Thread version number
  - To make thread IDs “unique” in time

Thread ID → TCB
Indirect via Table

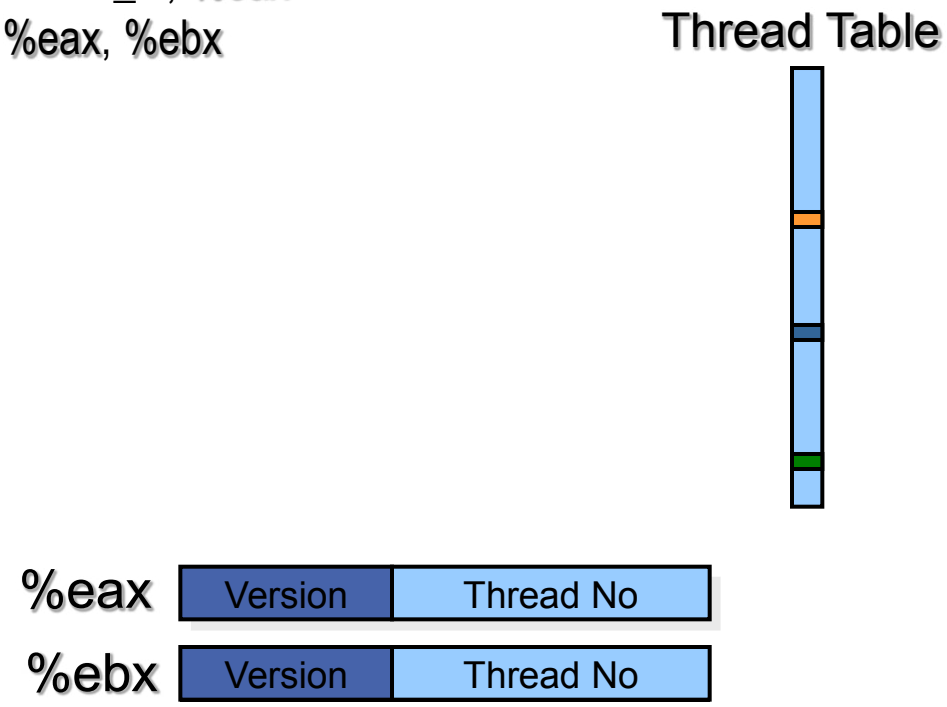
```
movl   thread_id, %eax
```

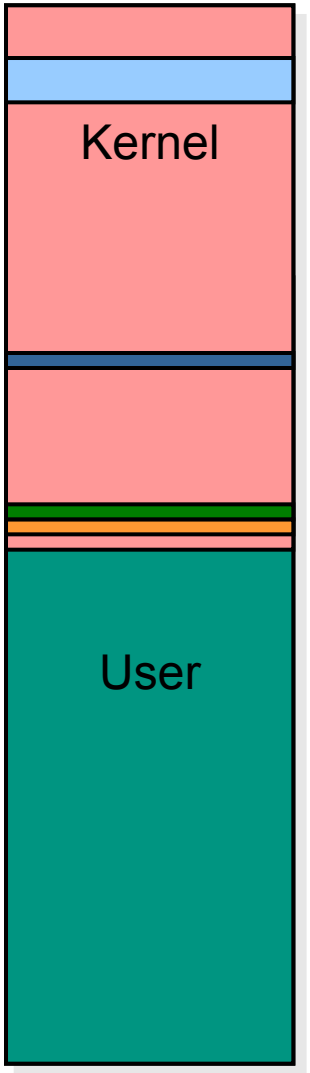




Thread ID → TCB
Indirect via Table

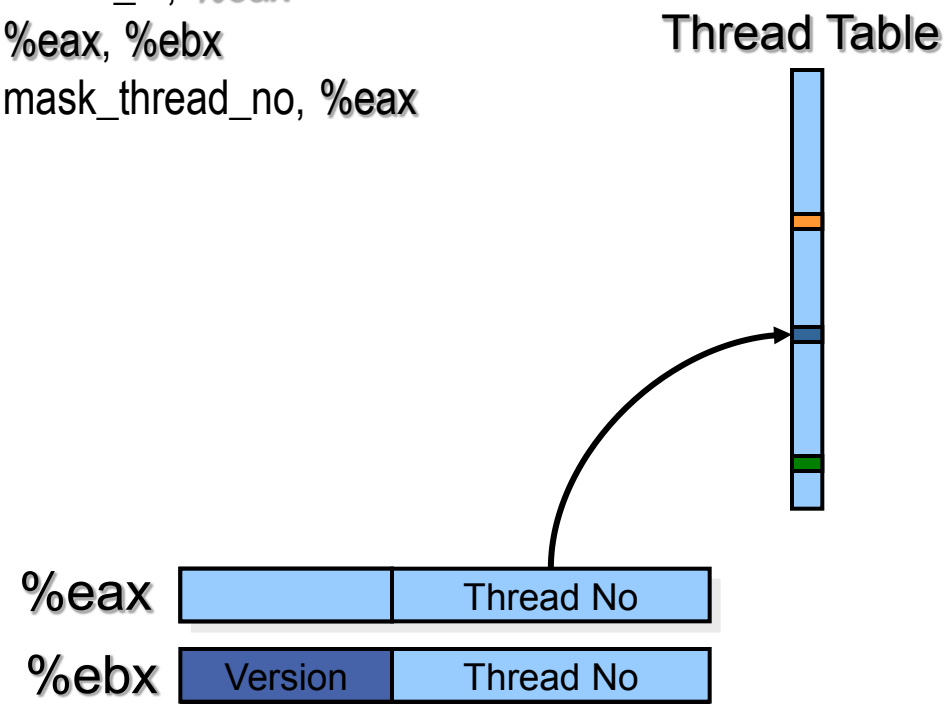
```
movl   thread_id, %eax
movl   %eax, %ebx
```

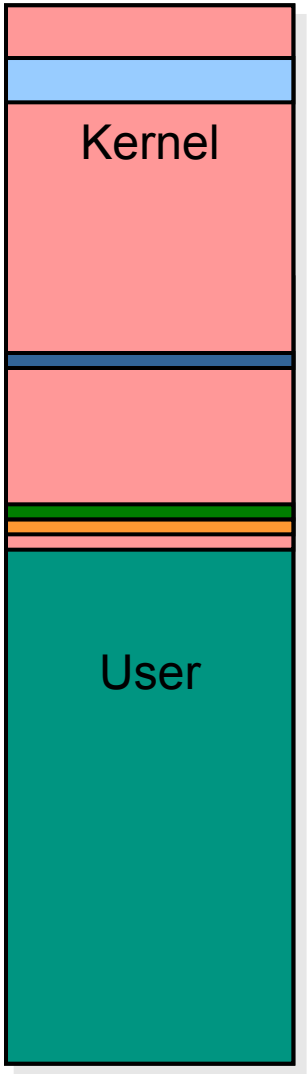




Thread ID → TCB
Indirect via Table

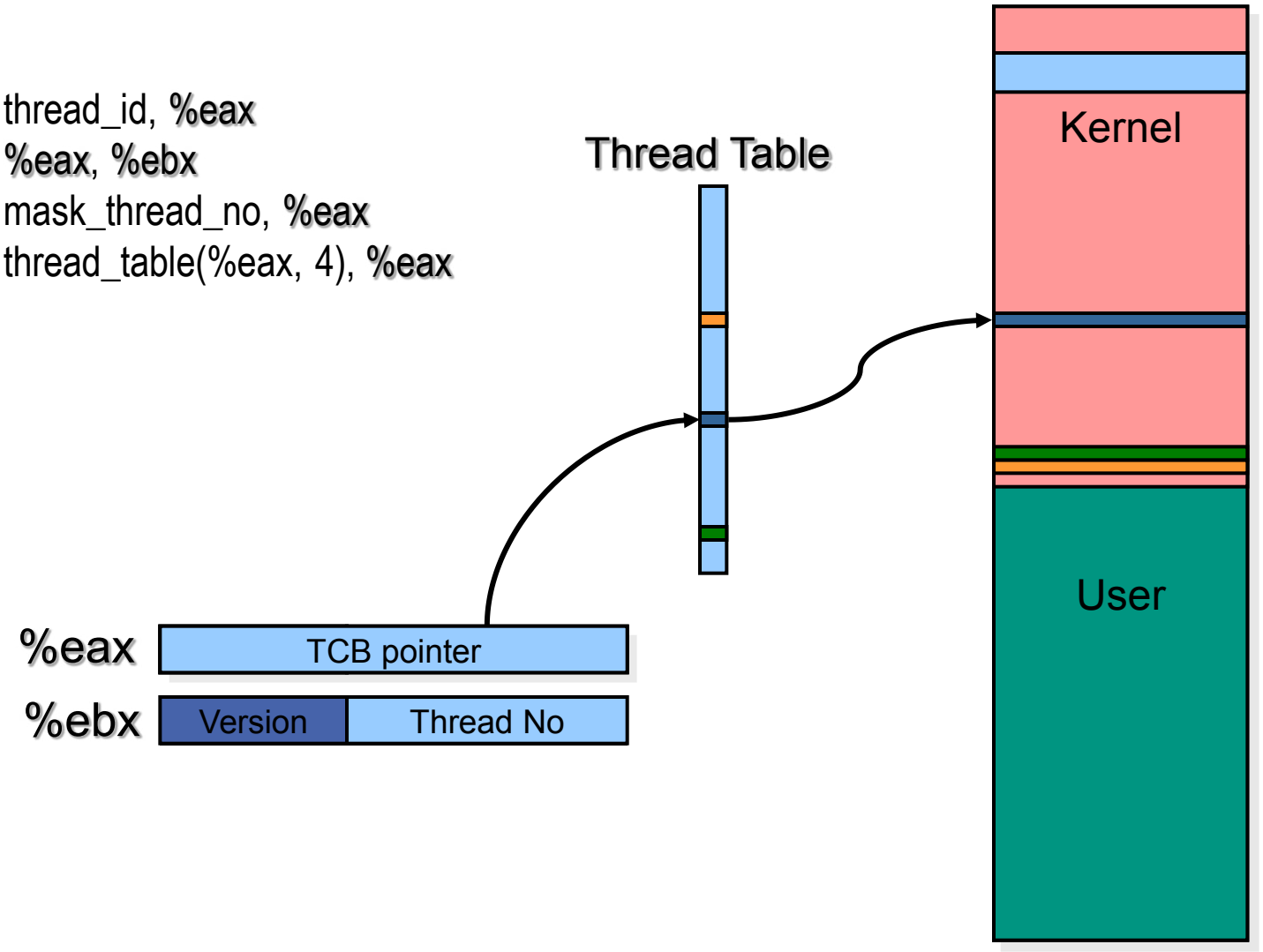
```
movl   thread_id, %eax
movl   %eax, %ebx
andl   mask_thread_no, %eax
```





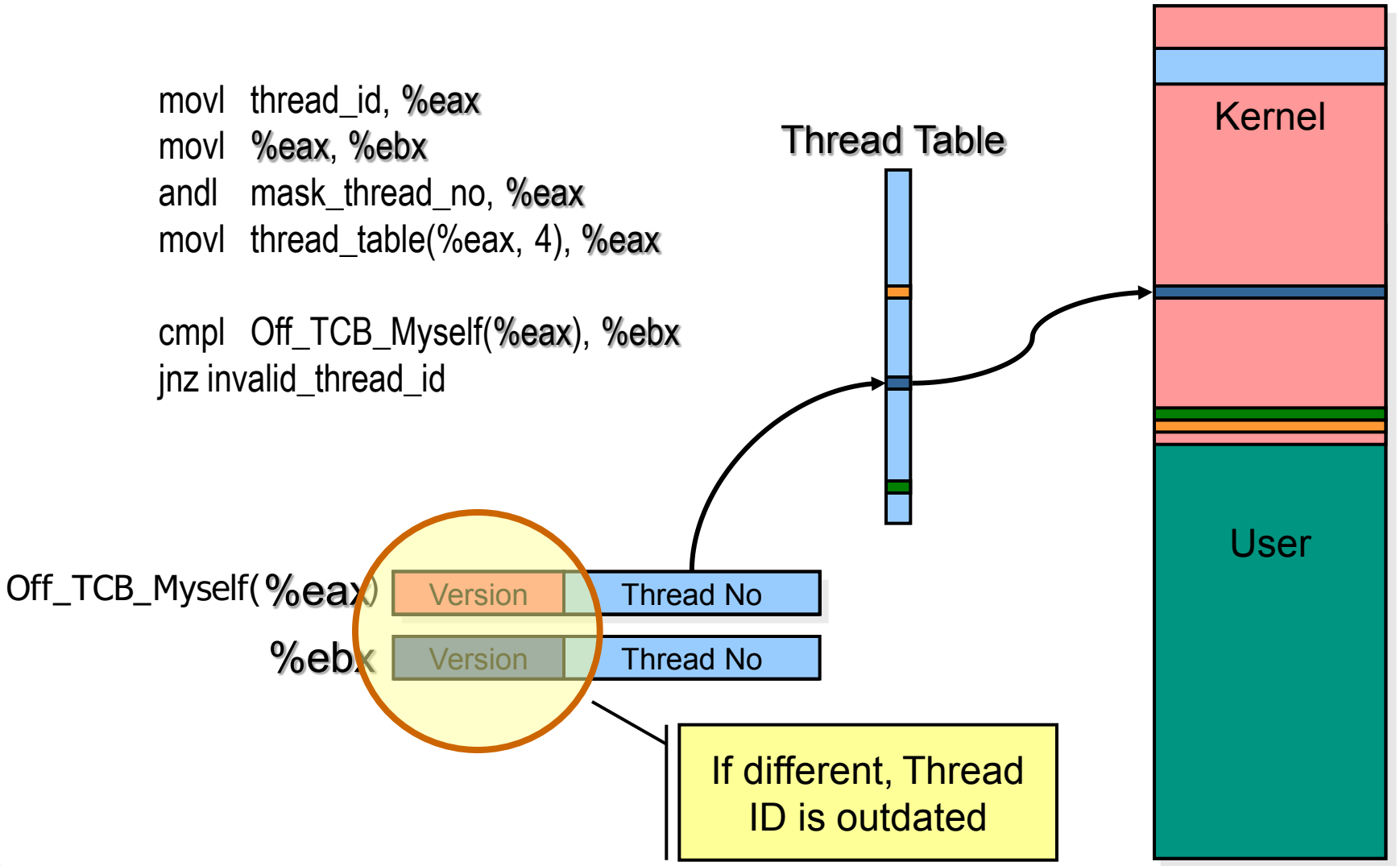
Thread ID → TCB
Indirect via Table

```
movl   thread_id, %eax
movl   %eax, %ebx
andl   mask_thread_no, %eax
movl   thread_table(%eax, 4), %eax
```



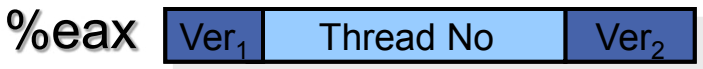
Thread ID → TCB
Indirect via Table

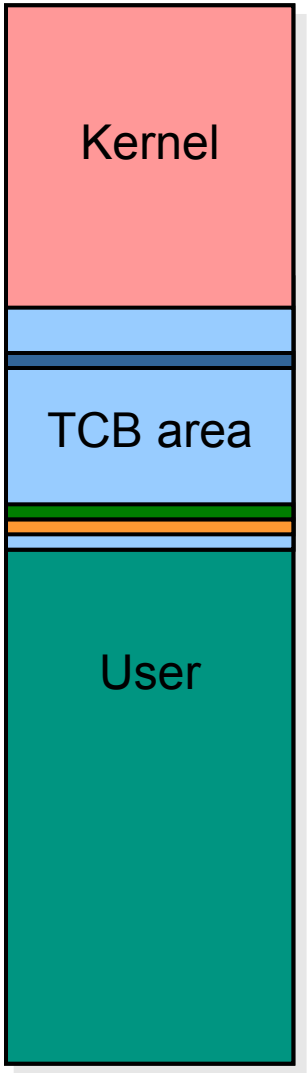
```
movl   thread_id, %eax
movl   %eax, %ebx
andl   mask_thread_no, %eax
movl   thread_table(%eax, 4), %eax

cmpl   Off_TCB_Myself(%eax), %ebx
jnz invalid_thread_id
```



Thread ID → TCB
Direct Address

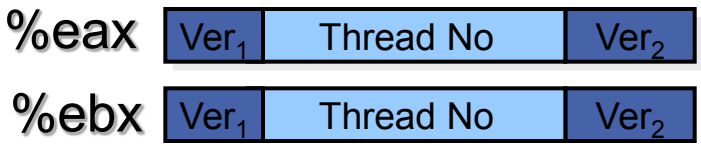
```
movl   thread_id, %eax
```

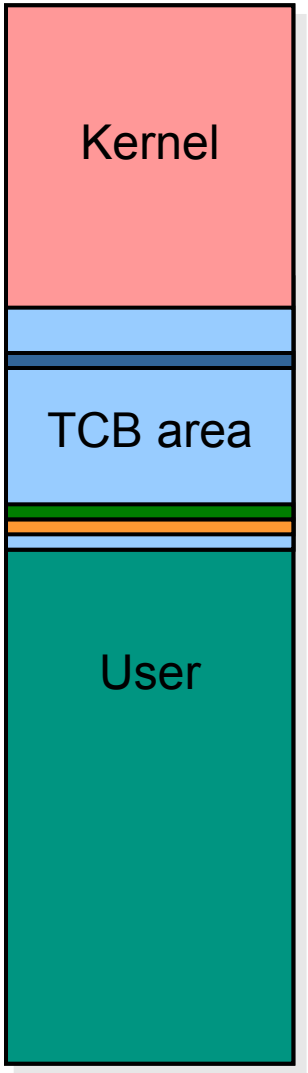




Thread ID → TCB
Direct Address

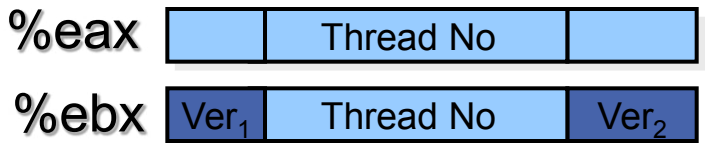
```
movl   thread_id, %eax
movl   %eax, %ebx
```

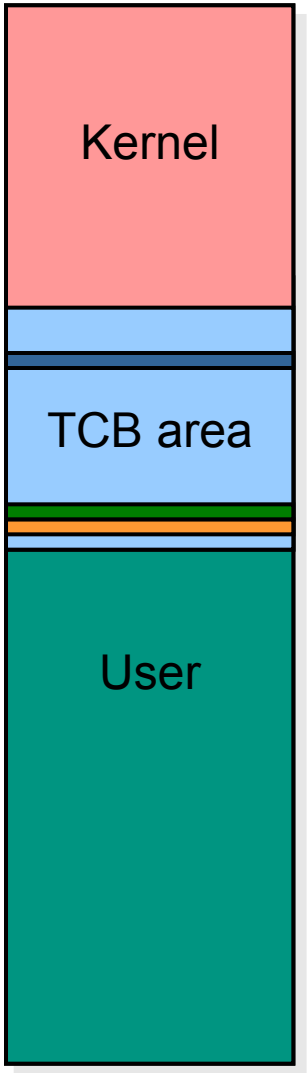




Thread ID → TCB
Direct Address

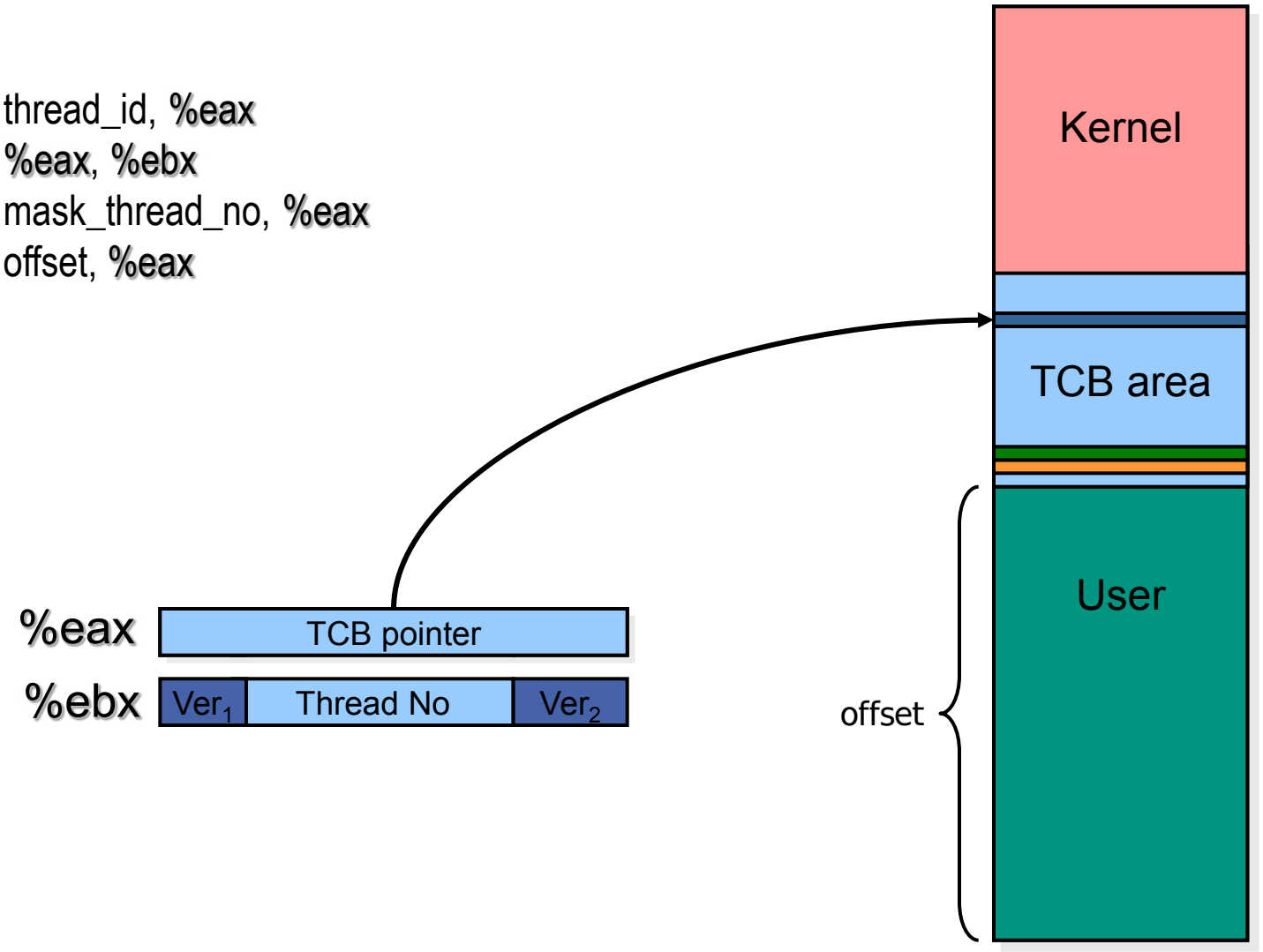
```
movl   thread_id, %eax
movl   %eax, %ebx
andl   mask_thread_no, %eax
```





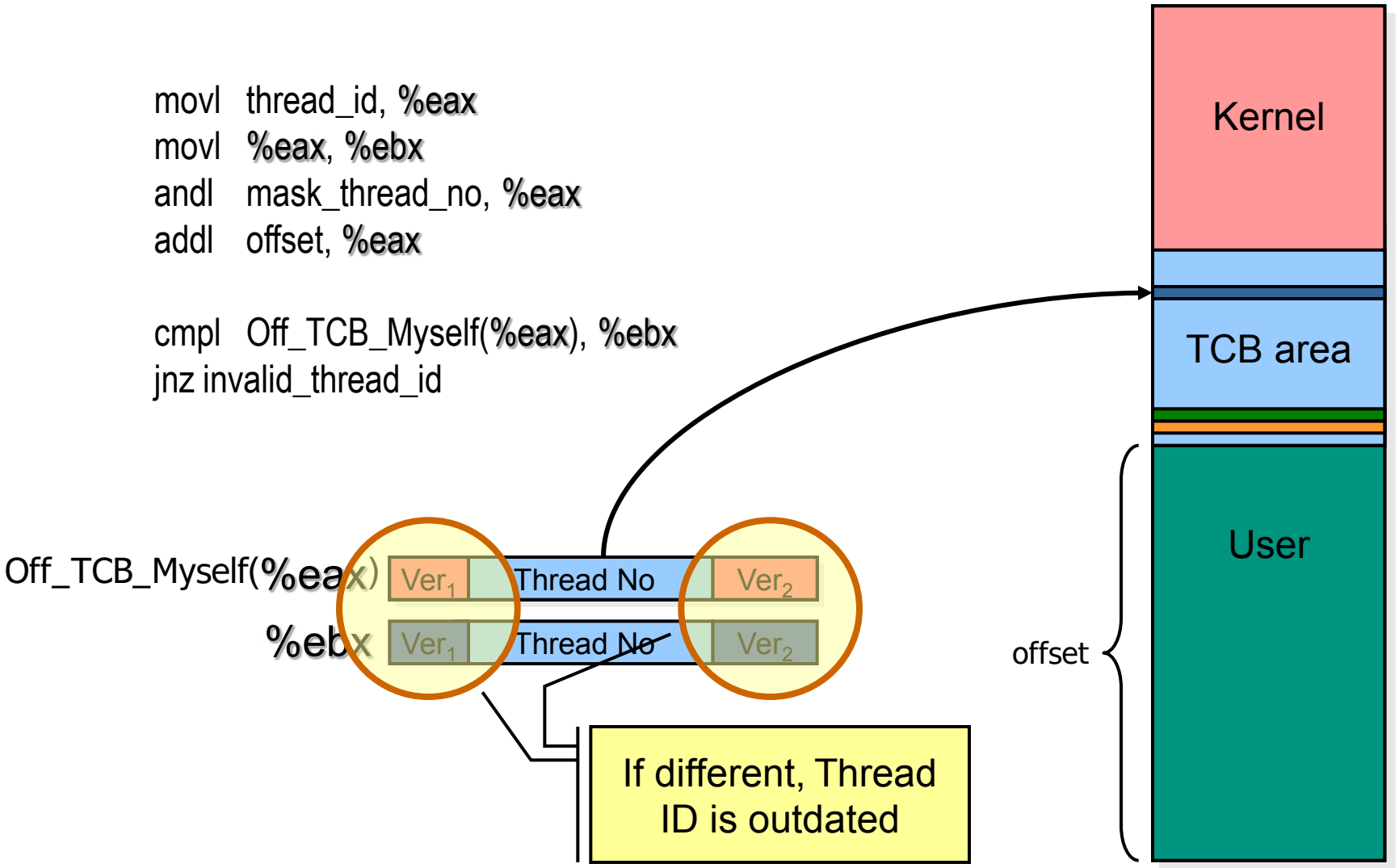
Thread ID → TCB
Direct Address

```
movl   thread_id, %eax
movl   %eax, %ebx
andl   mask_thread_no, %eax
addl   offset, %eax
```



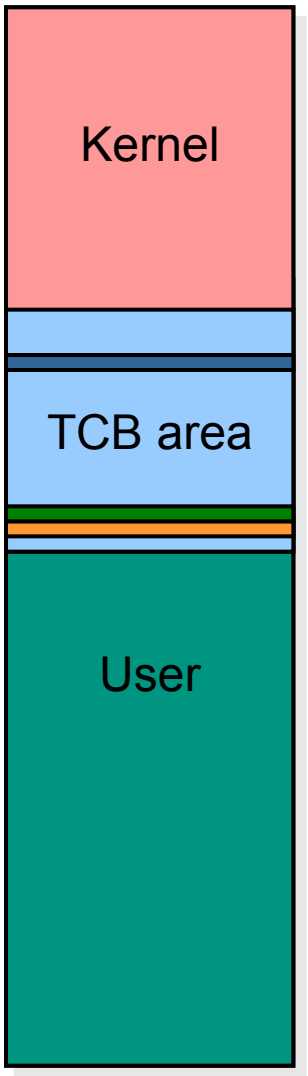
Thread ID → TCB
Direct Address

```
movl   thread_id, %eax
movl   %eax, %ebx
andl   mask_thread_no, %eax
addl   offset, %eax

cmpl   Off_TCB_Myself(%eax), %ebx
jnz invalid_thread_id
```



Thread ID → TCB
Direct Address

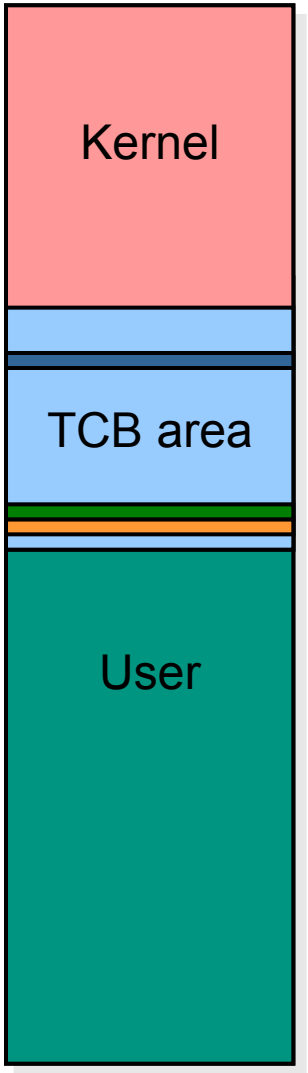
```
movl   thread_id, %eax
movl   %eax, %ebx
```



Thread ID → TCB
Direct Address

```
movl   thread_id, %eax
movl   %eax, %ebx
andl   mask_version, %eax
```

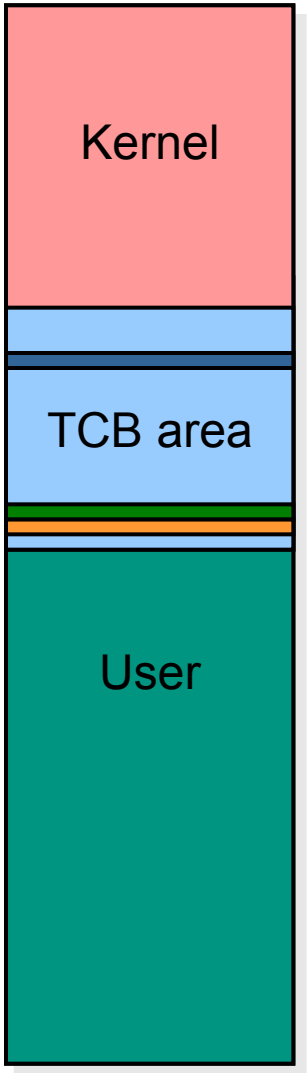
- Mask out lower bits



Thread ID → TCB
Direct Address

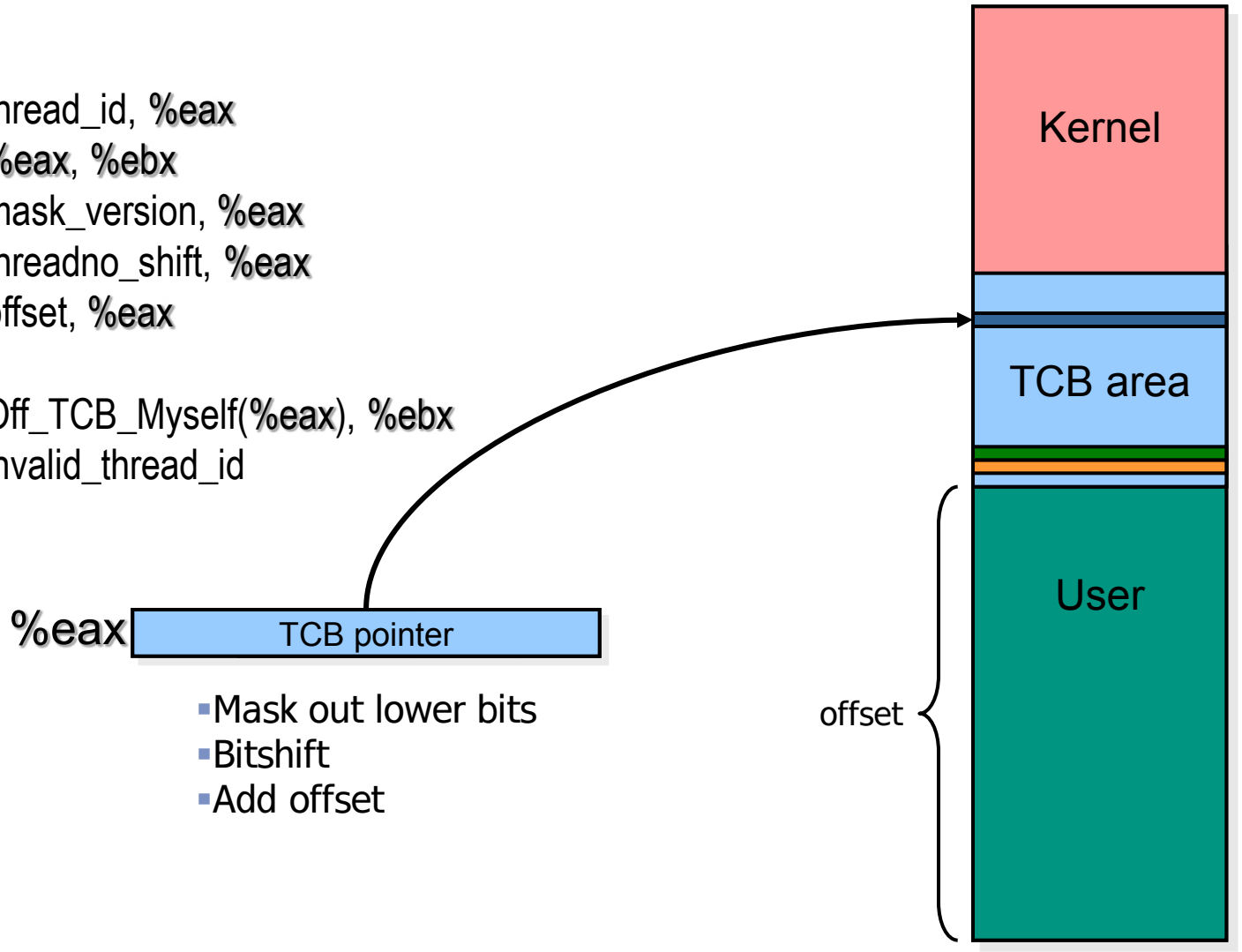
```
movl   thread_id, %eax
movl   %eax, %ebx
andl   mask_version, %eax
shrl   threadno_shift, %eax
```

- Mask out lower bits
- Bitshift



Thread ID → TCB
Direct Address

```
movl   thread_id, %eax
movl   %eax, %ebx
andl   mask_version, %eax
shrl   threadno_shift, %eax
addl   offset, %eax

cmpl   Off_TCB_Myself(%eax), %ebx
jnz    invalid_thread_id
```



- Mask out lower bits
- Bitshift
- Add offset

Thread ID Translation

- Via Table
  - Table access per TCB
  - Many TCBs per TLB entry (TCBs on superpages)
  - TLB entry for table (?)
- Via Computation
  - No table access
  - Few TCBs per TLB entry (sparsely populated area)

- TCB pointer array requires 1 MB virtual memory for 256k potential threads
- Virtual TCB array requires ≥ 256 MB virtual memory for 256k potential TCBs

Thread ID Translation

Examples:
- 4 kB pages, 4 kB TCBs
  - ➡ 1 TCB per TLB entry
- 16 kB pages, 2 kB TCBs
  - ➡ 8 TCBs per TLB entry

Via Table

- Table access per TCB
- Many TCBs per TLB entry (TCBs on superpages)
- TLB entry for table (?)

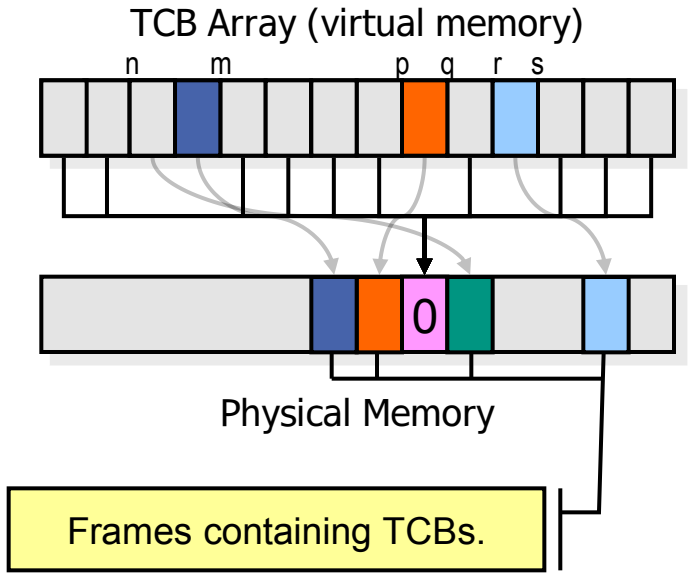
n

- No table access
- Few TCBs per TLB entry (sparsely populated area)

- TCB pointer array requires 1 MB virtual memory for 256k potential threads
- Virtual TCB array requires ≥ 256 MB virtual memory for 256k potential TCBs

0-Mapping Trick
Direct Addressing

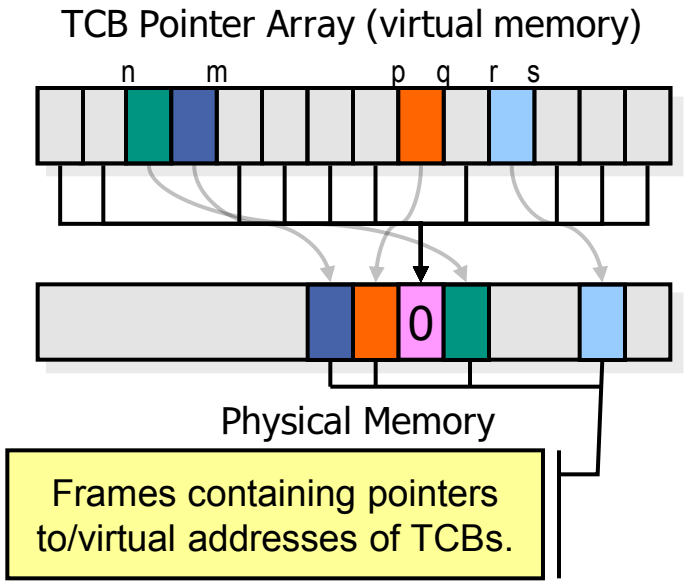
- Allocate physical memory for TCBs on demand
  - Dependent on the max. number of allocated TCBs
- Map all remaining TCBs to a 0-filled read-only page
  - Any access to unused threads will result in “invalid thread ID” (0)
  - Avoids additional check




```
cmpl Off_TCB_Myself(%eax), %ebx
jnz  invalid_thread_id
```

- Virtual TCB array requires ≥ 256 MB virtual memory for 256k potential TCBs

0-Mapping Trick
Indirect Addressing



- TCB pointer array requires 1 MB virtual memory for 256k potential threads

```
cmpl Off_TCB_Myself(%eax), %ebx
jnz  invalid_thread_id
```

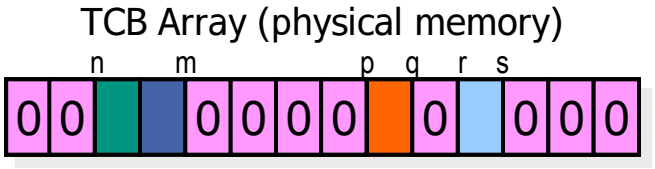
- Allocate physical parts of table on demand
  - Dependent on the max. number of allocated TCBs
- Map unused parts to a 0-filled read-only (r/o) page
  - Any access to unused threads will result in a NULL pointer
  - Requires extra check à la
    cmpl %eax, $0
    jnz invalid_thread_id
- Or: Map unused parts to a r/o page filled with pointers to a 0-filled r/o page
  - Any access to unused threads will result in an “invalid thread ID” (0)
  - Avoids additional check

Physical TCB array (seL4)

- Problem: Virtual TCB lookups cause TLB misses
  - Virtual TCB lookup is on IPC path!
- Solution: Use physical memory instead

+ No TLB misses
+ Significantly faster overall (Nourai 2005)
+ Easy to verify

– Requires ≥ 256 MB of physical memory!
– MMU may not permit physical addressing
  - Can still emulate physical memory using huge pages + pinning

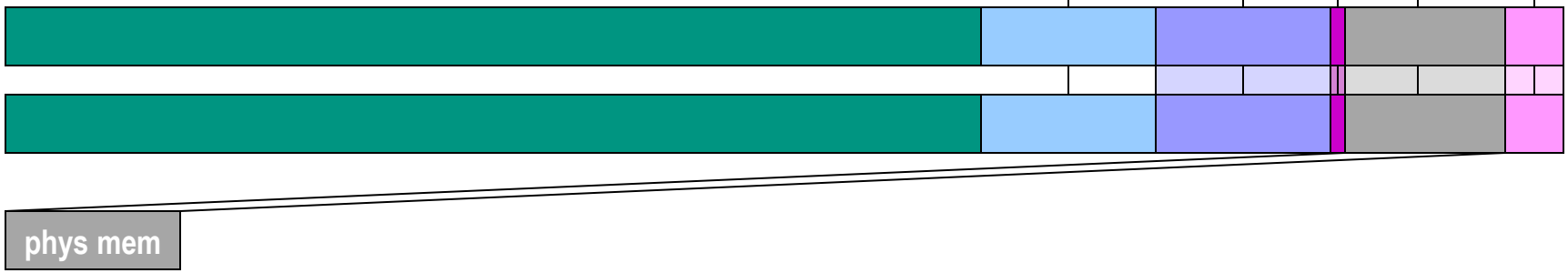


BASIC ADDRESS-SPACE LAYOUT

Address-Space Layout
32 bit, Virtual TCB Array

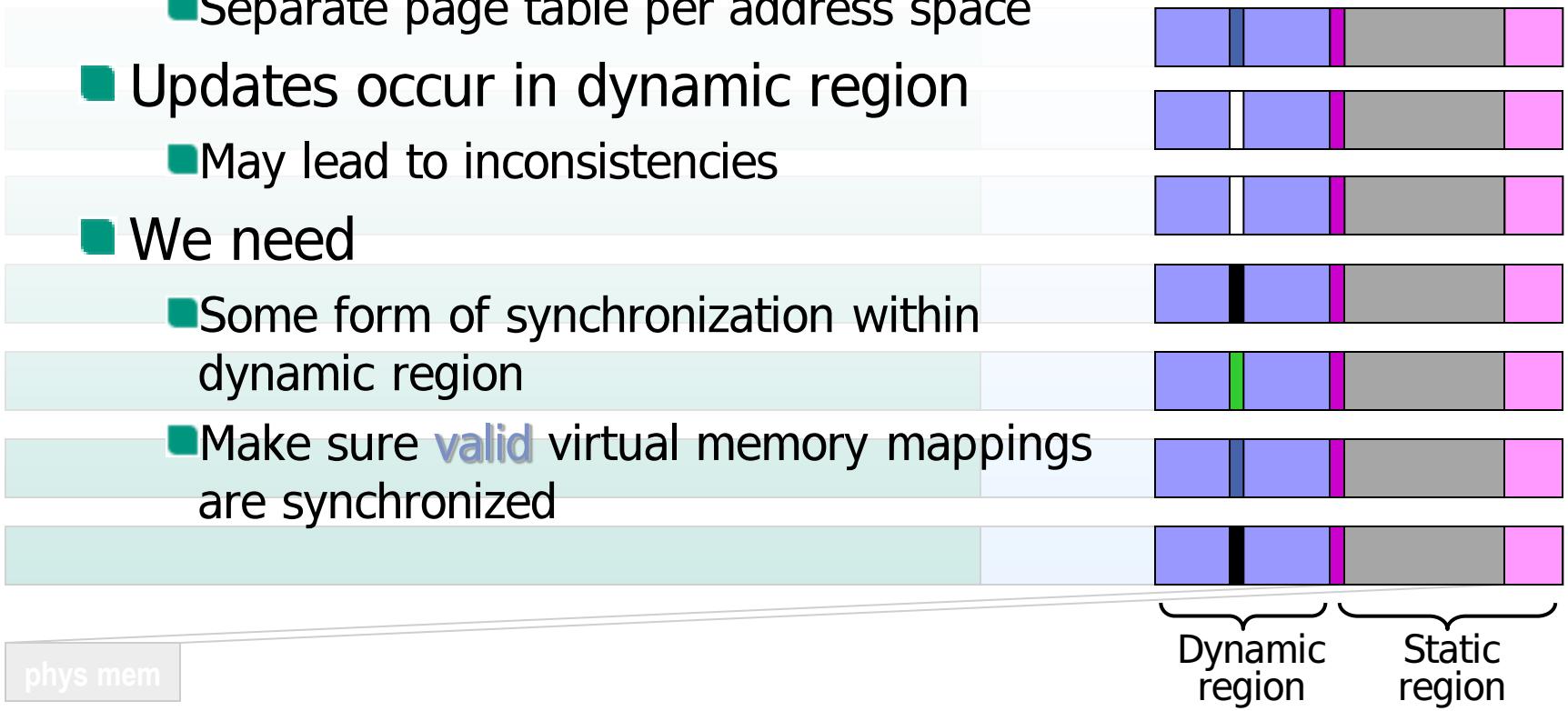
- User regions
- Shared system regions
- Per-space system regions

- Other kernel tables
- Physical memory
- Kernel code
- TCBs



Shared Region Synchronization

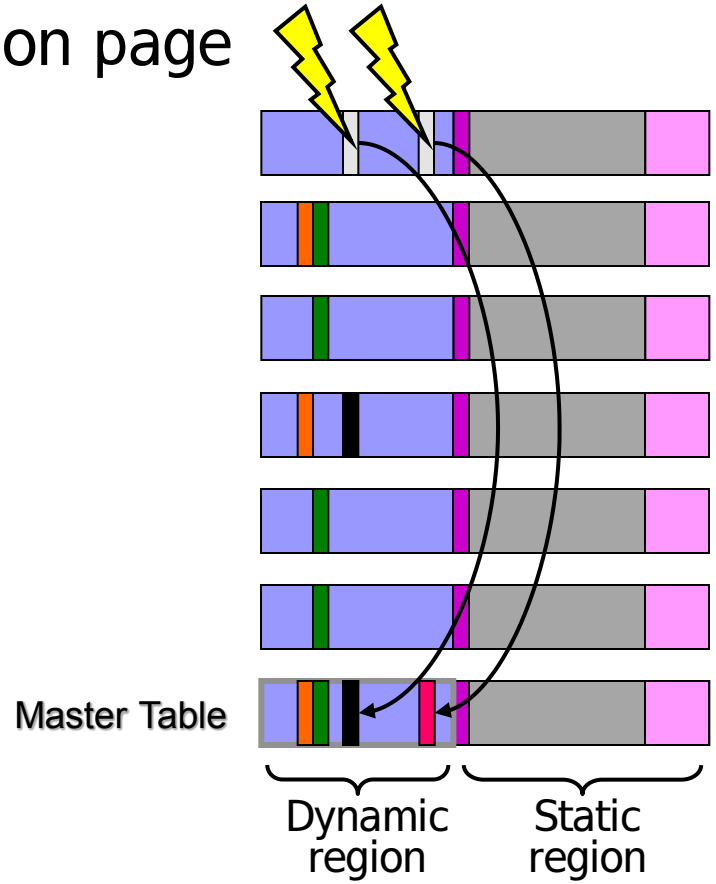
- We have
  - Regions shared among all address spaces
  - Separate page table per address space
- Updates occur in dynamic region
  - May lead to inconsistencies
- We need
  - Some form of synchronization within dynamic region
  - Make sure valid virtual memory mappings are synchronized



TCB Area Synchronization
Basic Algorithm

- Dedicate one table as master
- Synchronize with master table on page faults
  - Page fault algorithm:

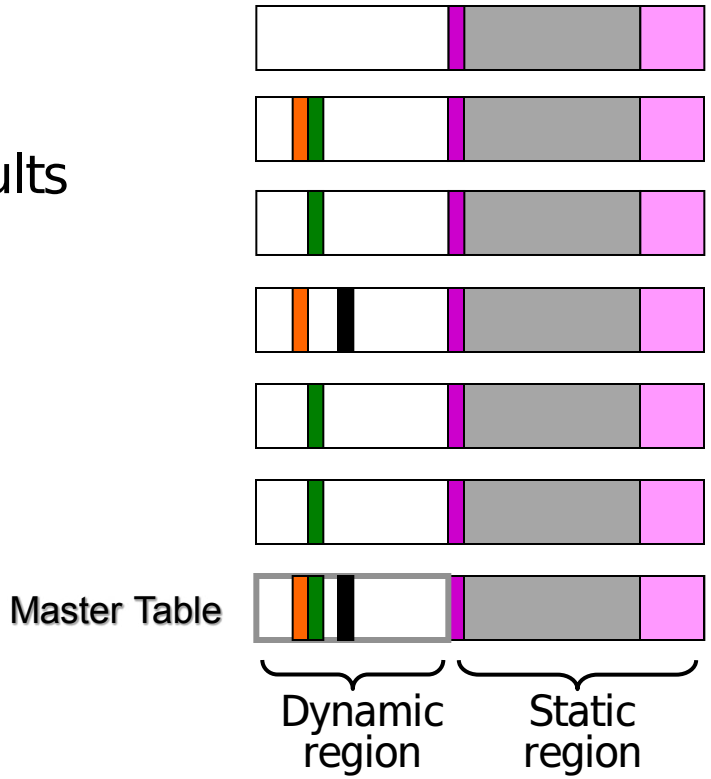
```
if (master entry valid) {
      copy entry from master
} else {
      create new entry in master
      copy entry from master
}
```



TCB Area Synchronization
Algorithm with 0-Mappings

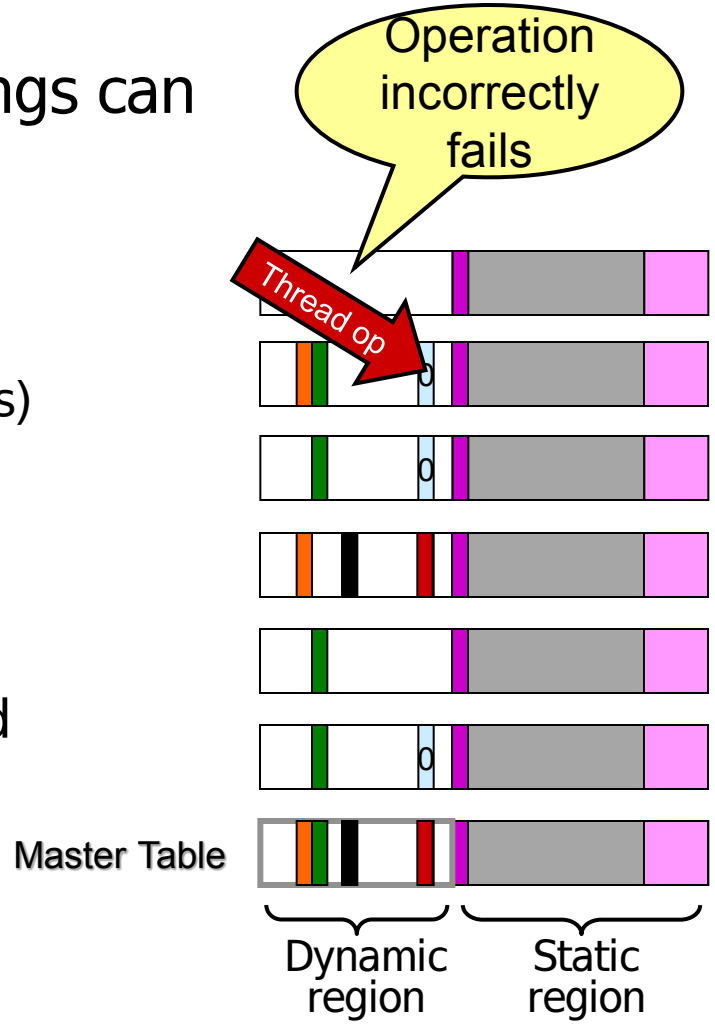
- Use 0-mappings for invalid TCBs
- Thread creation requires TCB modification
  - Create 0-mappings on read faults
  - Create TCB mappings on write faults

```
if (master entry not valid) {
    if (read fault) {
        create 0-mapping in master
    } else {
        create TCB entry in master
    }
}
copy entry from master
```



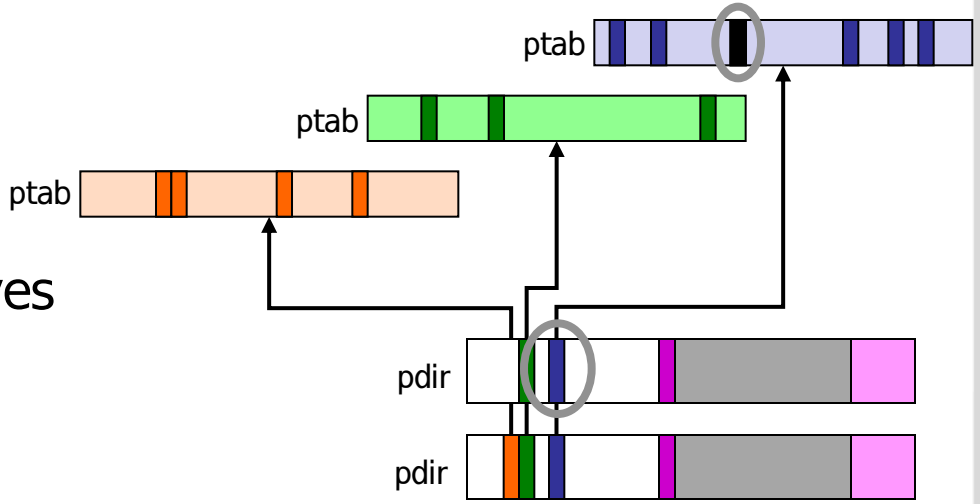
TCB Area Synchronization
Modifying Mappings

- Removing or modifying mappings can not be handled lazily
  - Must be handled brute force
  - Avoid removing mappings
    (i.e., do not remove TCB mappings)
- Potential problem
  - Create 0-Mappings (invalid TCBs)
  - Create a real TCB mapping
  - 0-Mappings must now be updated



TCB Area Synchronization
Modifying Mappings

- Page tables have multiple levels
  - IA-32: page directories and page tables
- We only synchronize top level (page directory)
- Modifications in lower levels visible in all spaces



Conclusion:

- Synchronization of pdirs solves the modification problem

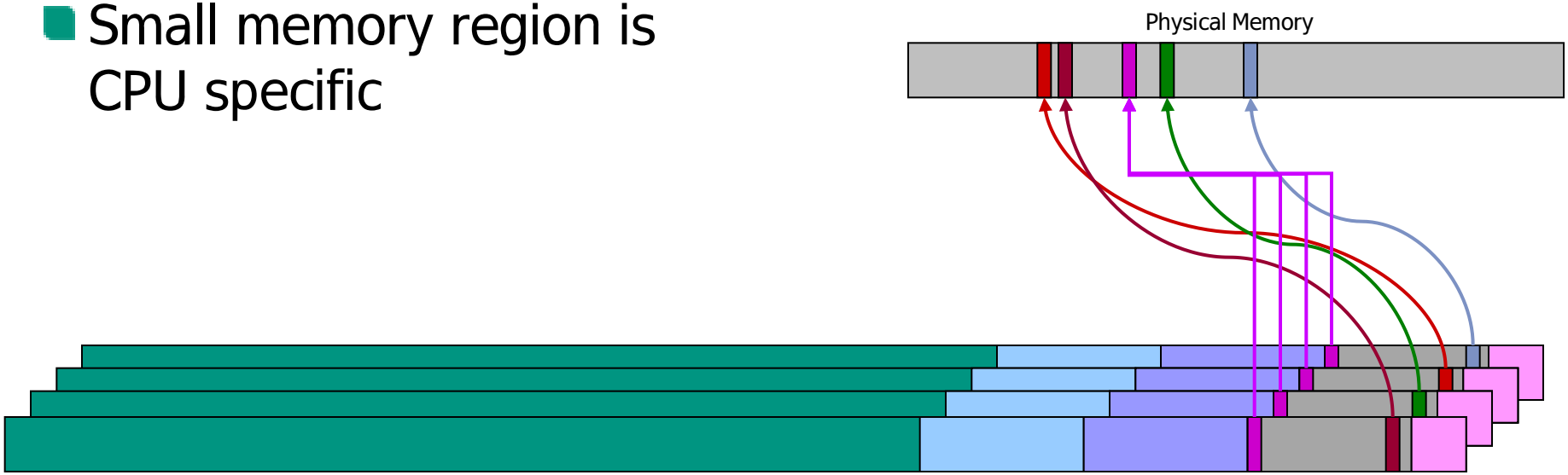
Processor-Specific Memory

- Certain objects and variables should be processor local
  - Ready queues, CPU ID, etc.
  - Prevents cache conflicts
- Will require frequent access

- Solution: per-CPU memory regions
  - Same virtual address
  - Different backing store
  - Avoids indirection table
    (i.e., no extra memory access)

Processor-Specific Memory

- One page table per CPU
- Most content identical
  - Requires synchronization
    (eagerly or lazily)
  - Synchronization at page directory level
- Small memory region is CPU specific



Processor-Specific Memory

- Per-CPU region
  - Avoids indirection
  - Dynamically adjusts to # CPUs
  - Scales to large # CPUs
  - No access to other CPU's data
- Array of per-CPU items
  - Linux approach
  - Compiled with max. number
  - Wastes memory for unused entries

Physical Memory Window

- Used by the kernel for
  - Page tables
  - Kernel memory
  - Kernel debugger

- Map and unmap
- Copy IPC

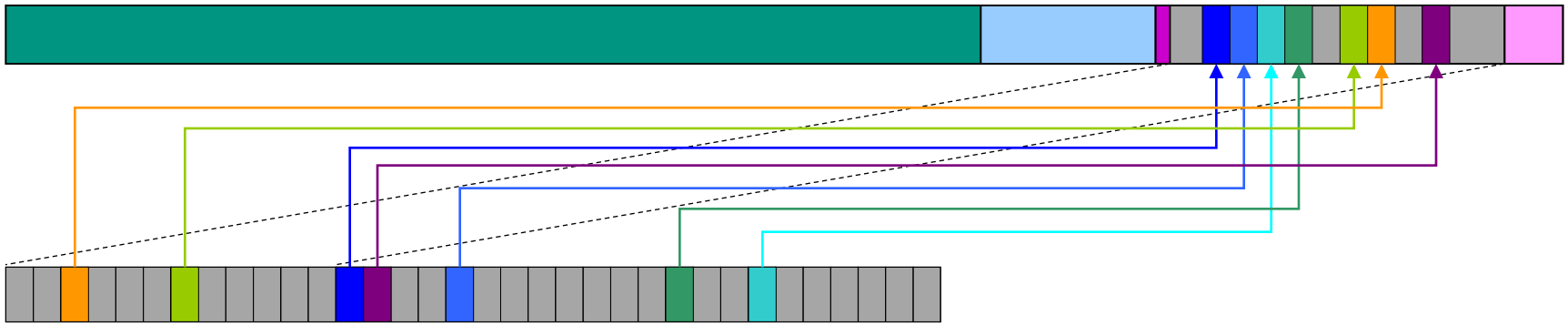
- Address spaces
- UTCBs

- KDB output
- Mem Dump

- Only used when the kernel accesses physical addresses
  - Limit valid physical range to remap size (256 MB)
  - Or ...

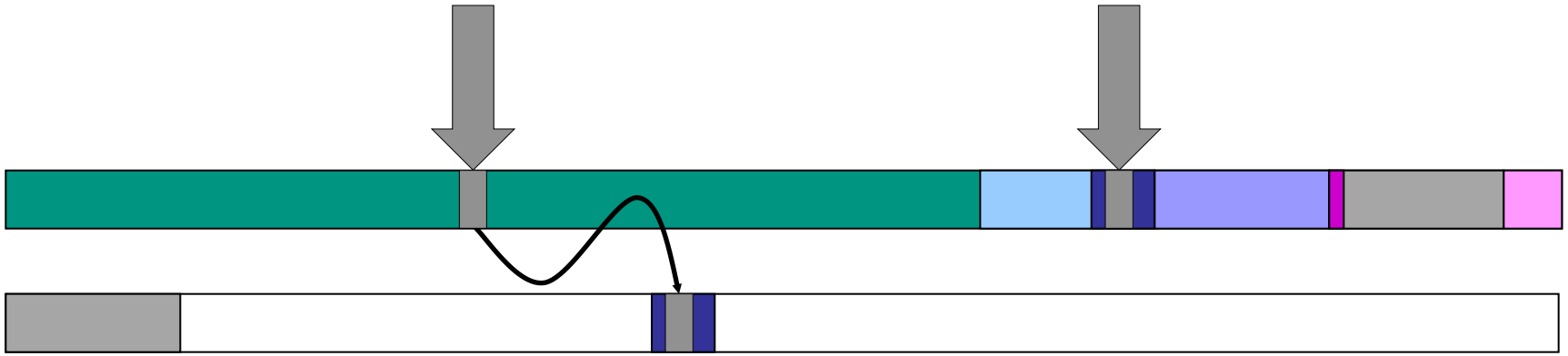
Physical-to-Virtual Pagetable

- Remap kernel-used pages
- Obtain virtual from physical address
  - Walk physical-to-virtual ptab in software
- Access physical memory via virtual address
- Costs?
  - Cache, TLB, runtime



Kernel Debugger (not performance critical)

- Might want/need to access memory (maybe in different address space)
- Walk page table in software
- Remap on demand (4 MB)
- Optimization: check if already mapped



Summary

- TCBs
  - Implement threads
  - Need to find them by Thread ID
- TCB area
  - Virtual/physical array
  - 0-mapping trick
- AS Layout
  - Shared region synchronization
  - Per-processor memory
  - Physical memory window