Threads and Scheduling

Simon Peter

Systems Group
Dept. Computer Science
ETH Zurich - Switzerland

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Outline

- Threads
  - Concepts
  - Usage/API
- *Scheduling*
  - *Concepts*
  - *Usage/API*

(Italics describe things subject to change)

Threads

Thread concept

Systems@ETH Zürich

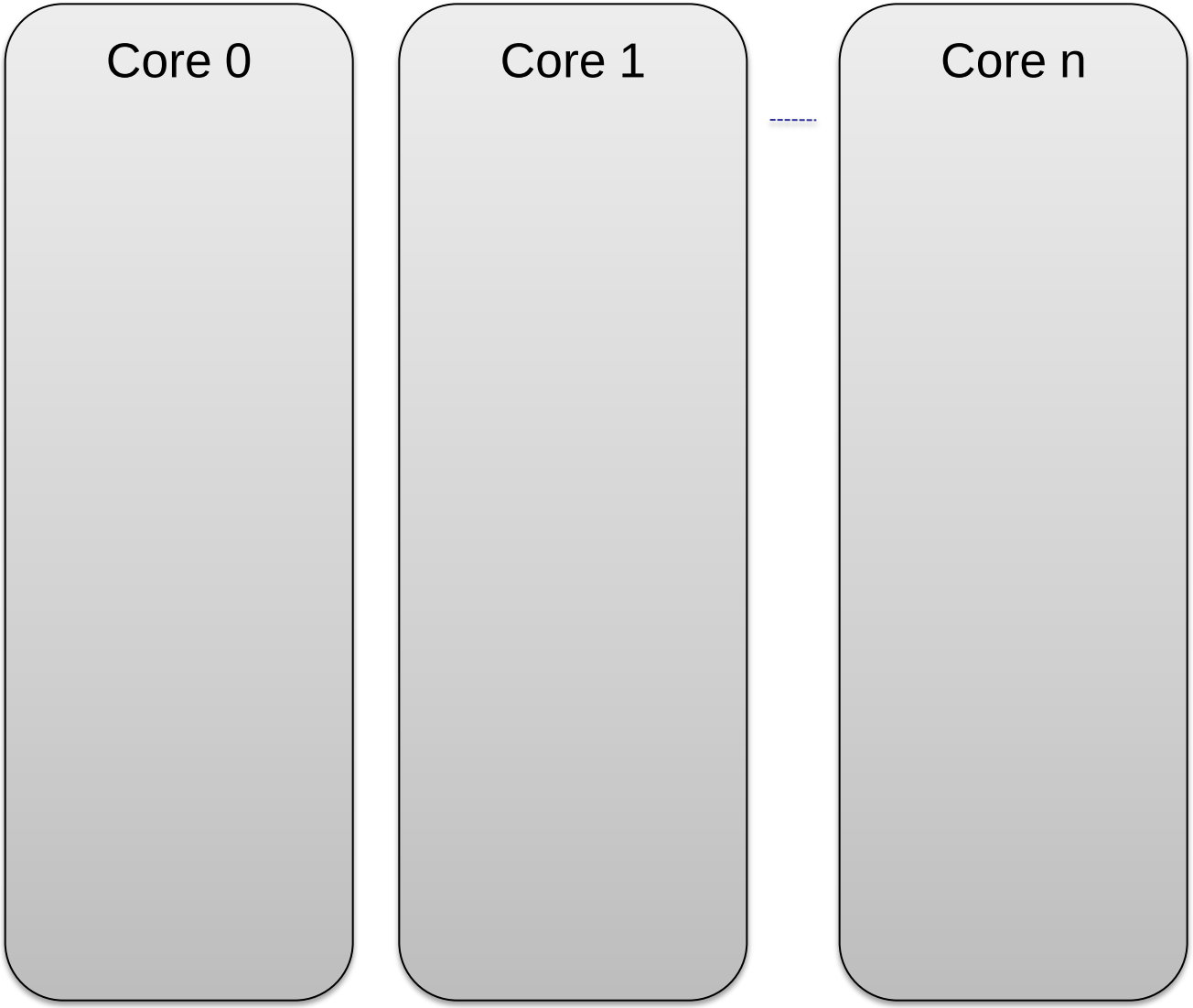
Classic

- Unit of execution
- Lowest schedulable entity
- Preemptable
- Affine to a core
- Can block
- Can be synchronized with

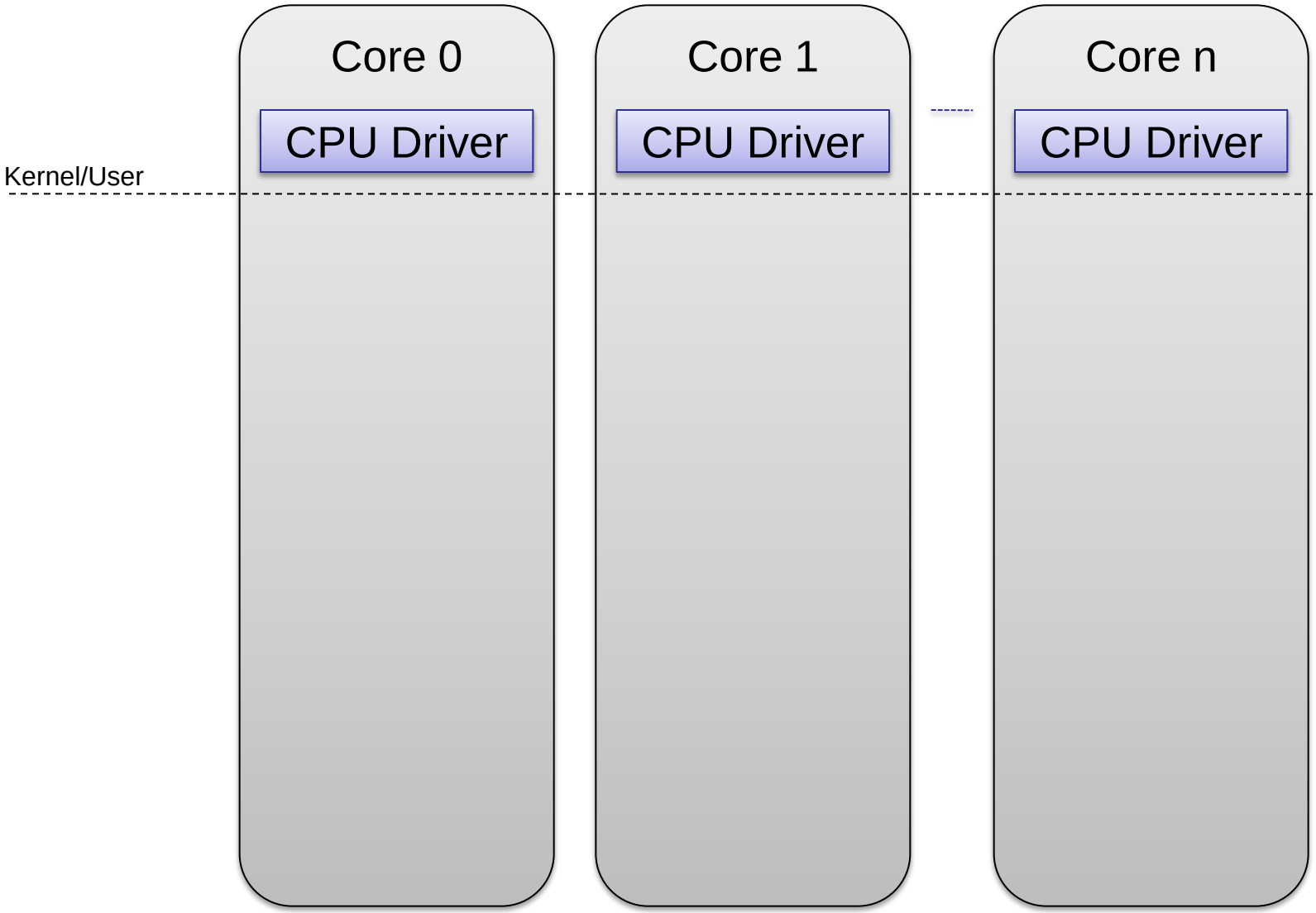
Barrelfish

- Can invoke capabilities
- Can send/receive messages

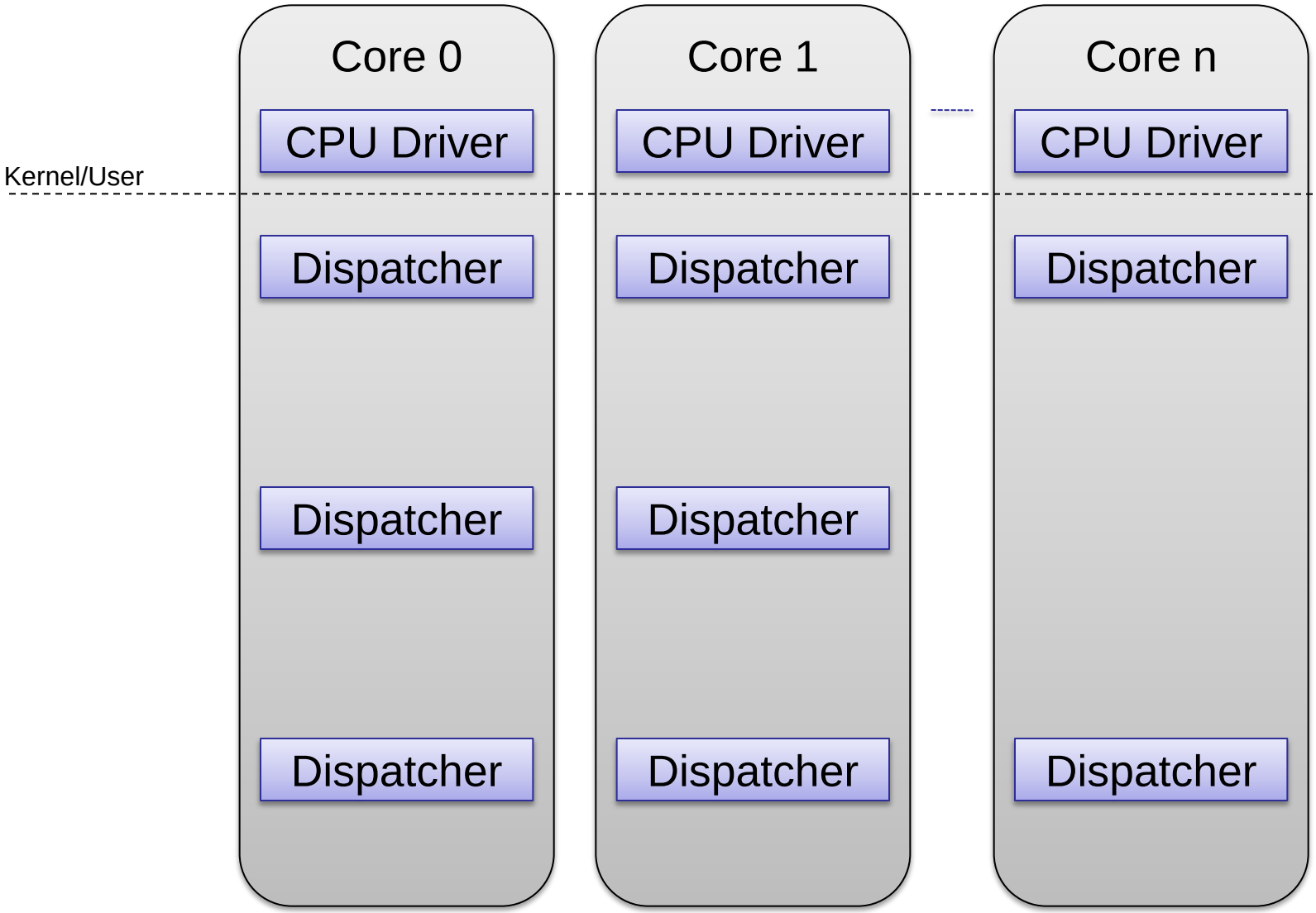
How are threads scheduled?



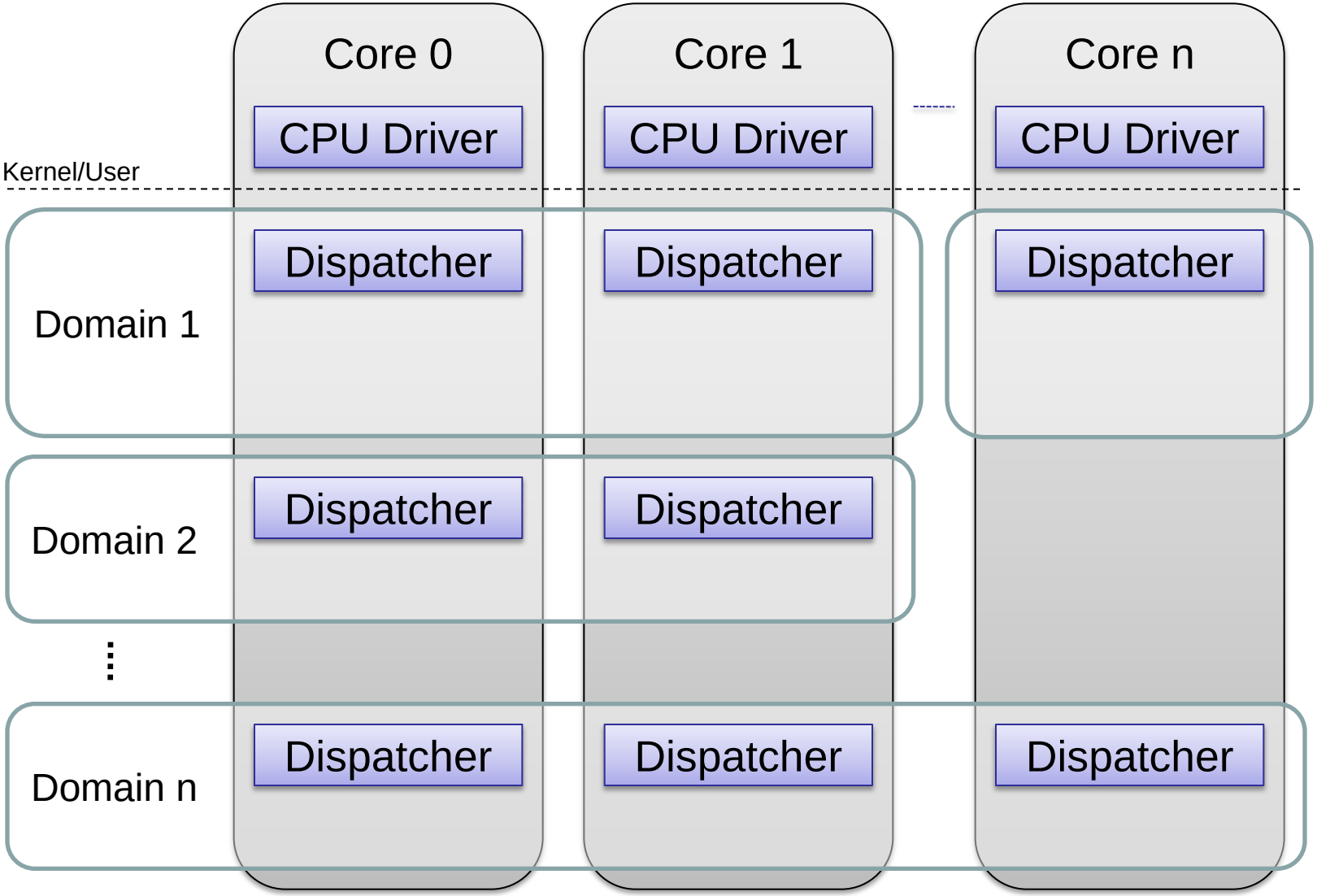
How are threads scheduled?



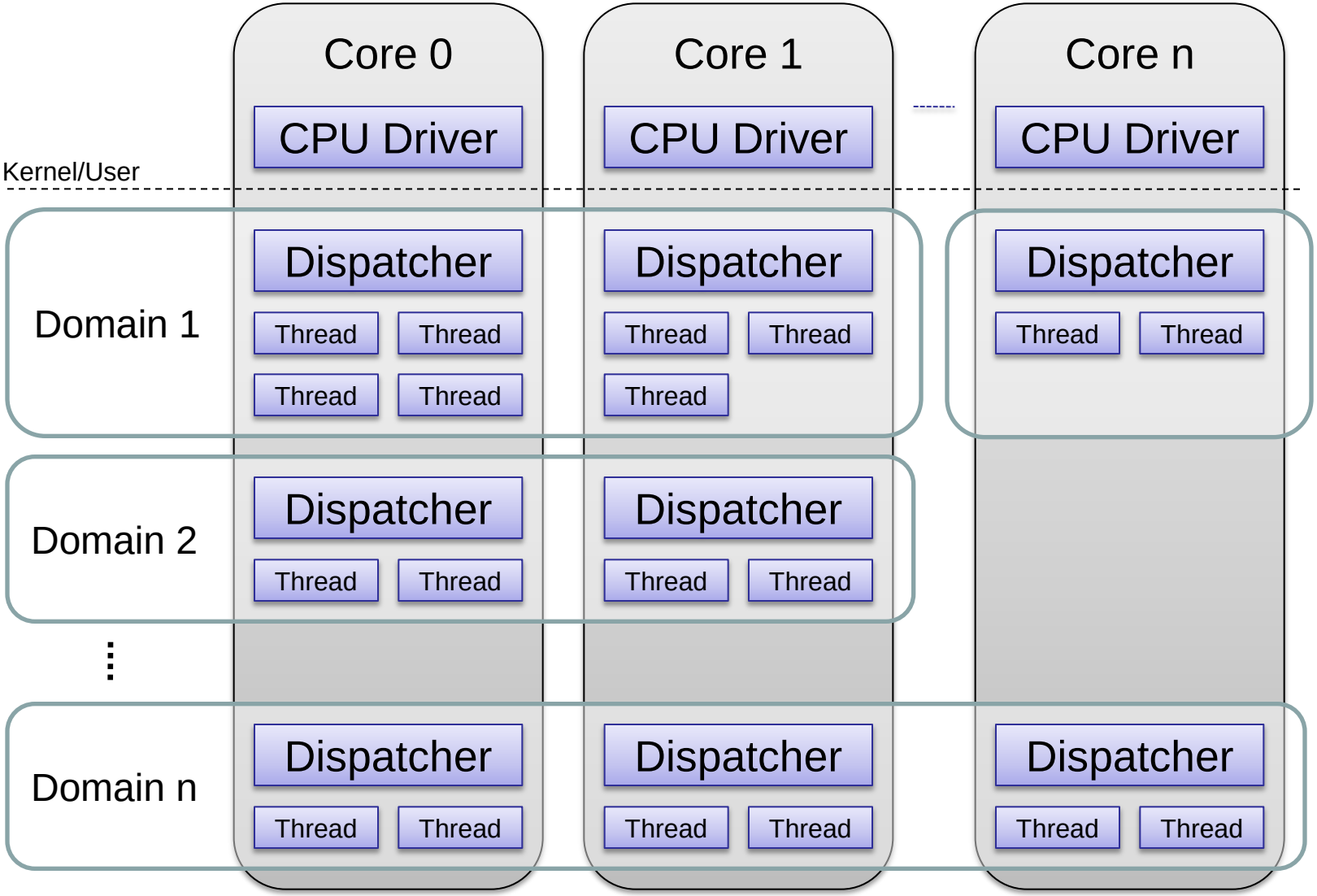
How are threads scheduled?



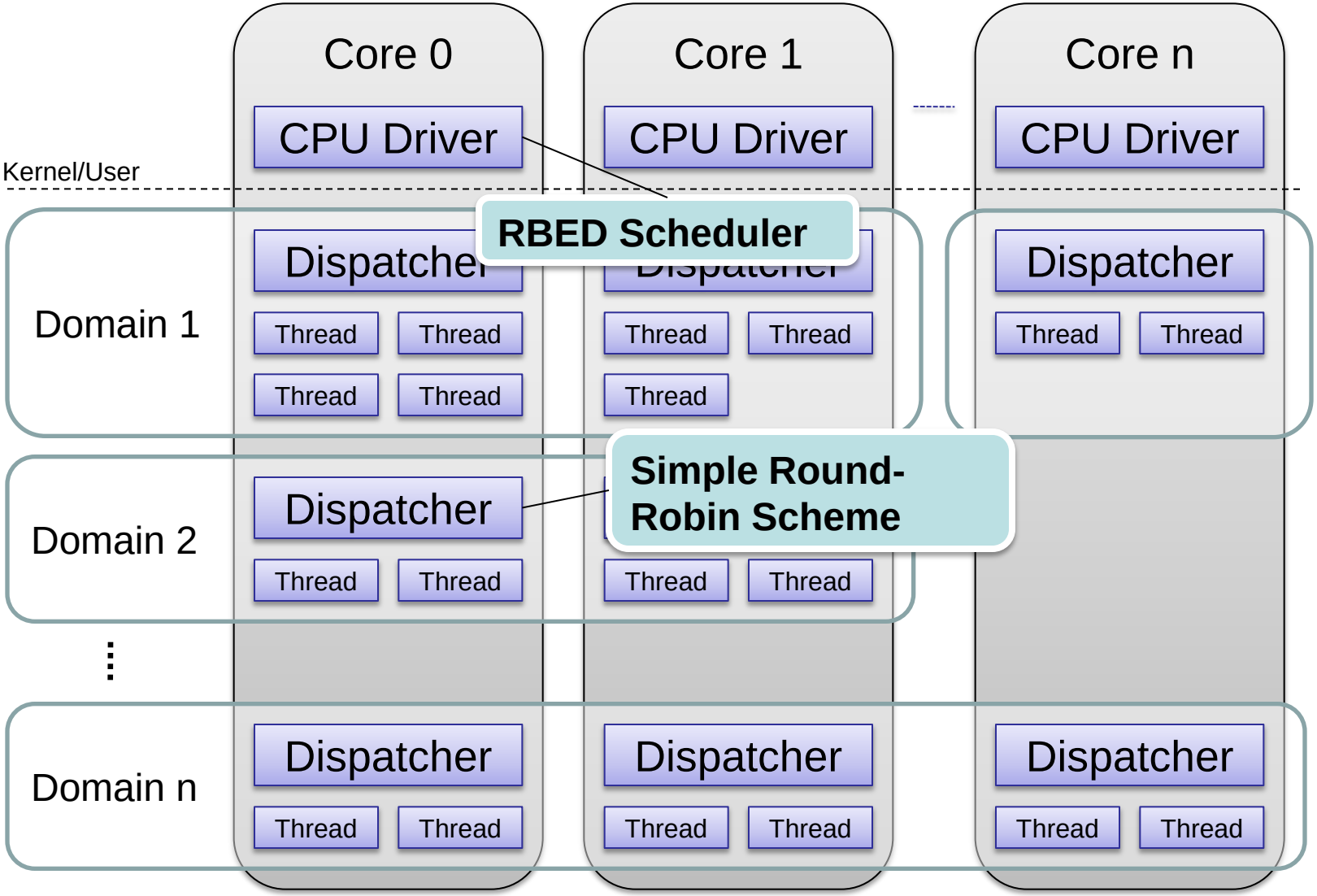
How are threads scheduled?



How are threads scheduled?



How are threads scheduled?



Thread support

- Implemented in user-space
- Provided by a threads package
- Currently contained in `libbarrelfish`
  - Public API in `include/barrelfish/threads.h`
  - Provides POSIX-like behavior

Thread data types

- `struct thread *thread = thread_self();`
  - `pthread_t` equivalent, designates a thread
- `int (*thread_func_t)(void *data)`
  - Thread start routine
- `struct thread_mutex mutex = THREAD_MUTEX_INITIALIZER;`
  - Mutex (can be nested and tested)
- `struct thread_cond cond = THREAD_COND_INITIALIZER;`
  - Condition variable (signal and broadcast semantics)
- `struct thread_sem sem = THREAD_SEM_INITIALIZER;`
  - Semaphore (can be tested)

Thread creation

- Create thread on local core

```
struct thread *thread_create(thread_func_t start_func,
void *data);
```

```
struct thread *thread_create_varstack(thread_func_t
start_func, void *arg, size_t stacksize);
```

Thread stack caveats

- *Allocated at thread creation time*
- *Don't grow dynamically*
- *Not protected against overflow*

Thread-local storage

- *One `void *` can be associated with each thread*
  - *Bad composability, we know*
- Pthreads has thread local key-value store instead

```
void thread_set_tls(void *tls);
```

```
void *thread_get_tls(void);
```

Other useful thread operations

Systems@ETH zürich

- Return thread ID

```
struct thread *thread_self(void);
```

- Yield timeslice of this thread

```
void thread_yield(void);
```

- Exit thread

```
void thread_exit(void);
```

- Threads are only cleaned up when joined with
  - Unless detached

```
errval_t thread_join(struct thread *thread, int *retval);
errval_t thread_detach(struct thread *thread);
```

What we don't have but POSIX does

- Forking-related operations (`pthread_atfork()`)
- Attributes
  - Scheduling policy set via scheduler API
  - No guarded stacks
  - No contention scopes
- Barriers
  - Done elsewhere
- Cancelable threads
- Timed synchronization primitives
- Concurrency levels
- Thread signals
- Read/write locks

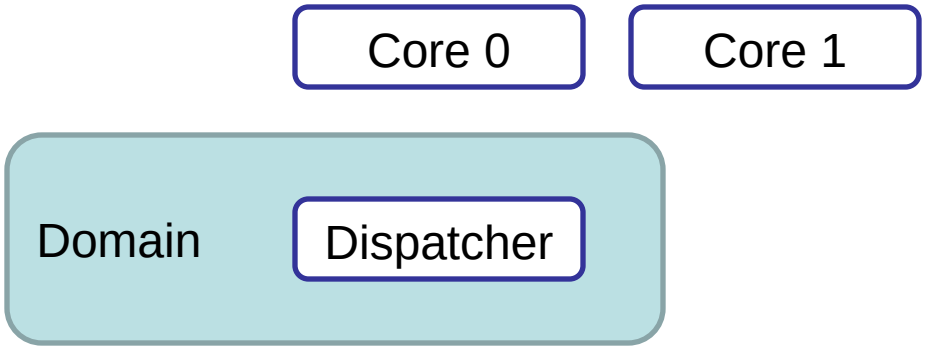
Multi-core Threading

- “Span” your domain to other cores:

```
errval_t domain_new_dispatcher(uint8_t core_id,
domain_spanned_callback_t callback, void *callback_arg);
```

```
static void domain_spanned(void *arg, errval_t err);
```

- Asynchronous operation



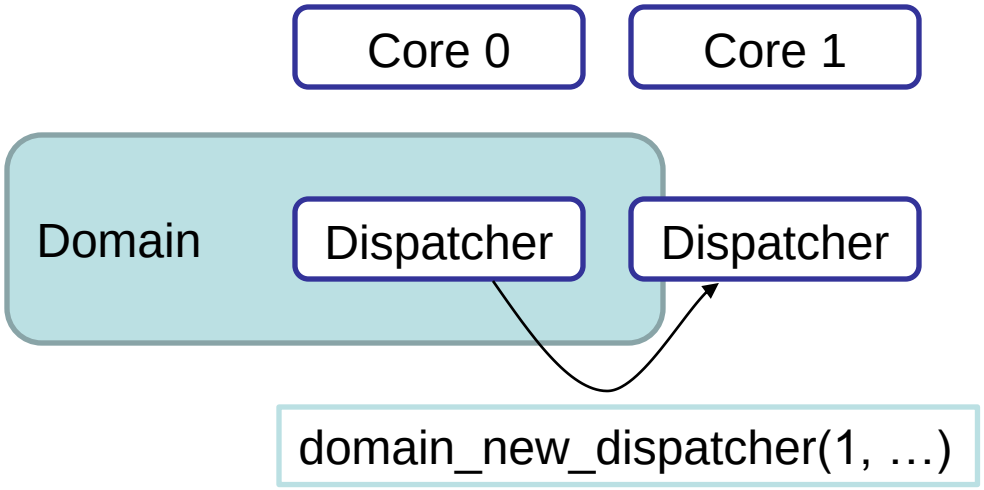
Multi-core Threading

- “Span” your domain to other cores:

```
errval_t domain_new_dispatcher(uint8_t core_id,
domain_spanned_callback_t callback, void *callback_arg);
```

```
static void domain_spanned(void *arg, errval_t err);
```

- Asynchronous operation



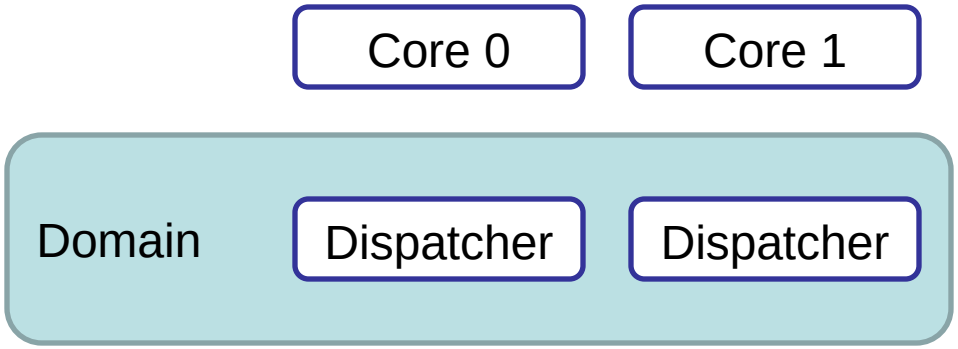
Multi-core Threading

- “Span” your domain to other cores:

```
errval_t domain_new_dispatcher(uint8_t core_id,
domain_spanned_callback_t callback, void *callback_arg);
```

```
static void domain_spanned(void *arg, errval_t err);
```

- Asynchronous operation



Spanning a domain

- New dispatcher has its own resources
  - Monitor, memory server bindings
  - Slabs, slots and heaps
  - *Page tables*
  - *Capabilities*
  - *All other bindings*
- All other resources are shared
  - Virtual address space
  - Locks
  - Semaphores
  - Condition variables
  - …

Multi-core threading

Systems@ETH zürich

- Start thread on different core:

```
errval_t domain_thread_create_on(coreid_t core_id,
thread_func_t start_func, void *arg);
```

- Move thread to different core:
  - *Currently only supports self-migration*
  - *Doesn't migrate open connections!*

```
errval_t domain_thread_move_to(struct thread *thread,
coreid_t core_id);
```

Scheduling

CPU Driver Scheduler

Systems@ETH zürich

- Rate-Based Earliest Deadline First (RBED)
  - Algorithm by Scott Brandt, UC Santa Cruz
  - Deterministic, versatile, unified model
- Best-effort tasks with priorities
  - UNIX-style I/O priority boost
- Rate-based tasks
  - Worst-case execution time, deadline, period
- Soft-realtime
  - Upcall when deadline missed
- Hard-realtime
  - Admission control

Scheduling at multiple timescales

- Long-term
  - Scheduler manifests
  - Divide app into phases with RBED parameters
- Mid-term
  - Phase changes
- Short-term
  - Scheduled by RBED

Scheduler API

- Public API in `include/barrelfish/resource_ctrl.h`
- Submit **scheduling manifest** on local dispatcher

```
errval_t rsrc_manifest(const char *manifest, rsrcid_t
*id);
```

- Join the manifest from other dispatchers

```
errval_t rsrc_join(rsrcid_t id);
```

- Change resource **phase** from any joined dispatcher

```
errval_t rsrc_phase(rsrcid_t id, uint32_t phase);
```

Manifest Example

```
static const char *my_manifest =
    "B 1\n"                 // Normal phase
    "H 20 160 160\n";       // Hard real-time phase
```

- Phases are implicitly numbered from 0
- Phase 0: Best-effort with priority 1
- Phase 1: Hard real-time
  - Worst-case execution time 20ms
  - Period 160ms
  - Deadline 160ms

Thank you

Backup