Advanced Operating Systems
MS degree in Computer Engineering
University of Rome Tor Vergata
Lecturer: Francesco Quaglia

Kernel level task management

1. Advanced/scalable task/threads management schemes
2. (Multi-core) CPU scheduling approaches
3. Binding to the Linux architecture

Tasks vs processes/threads

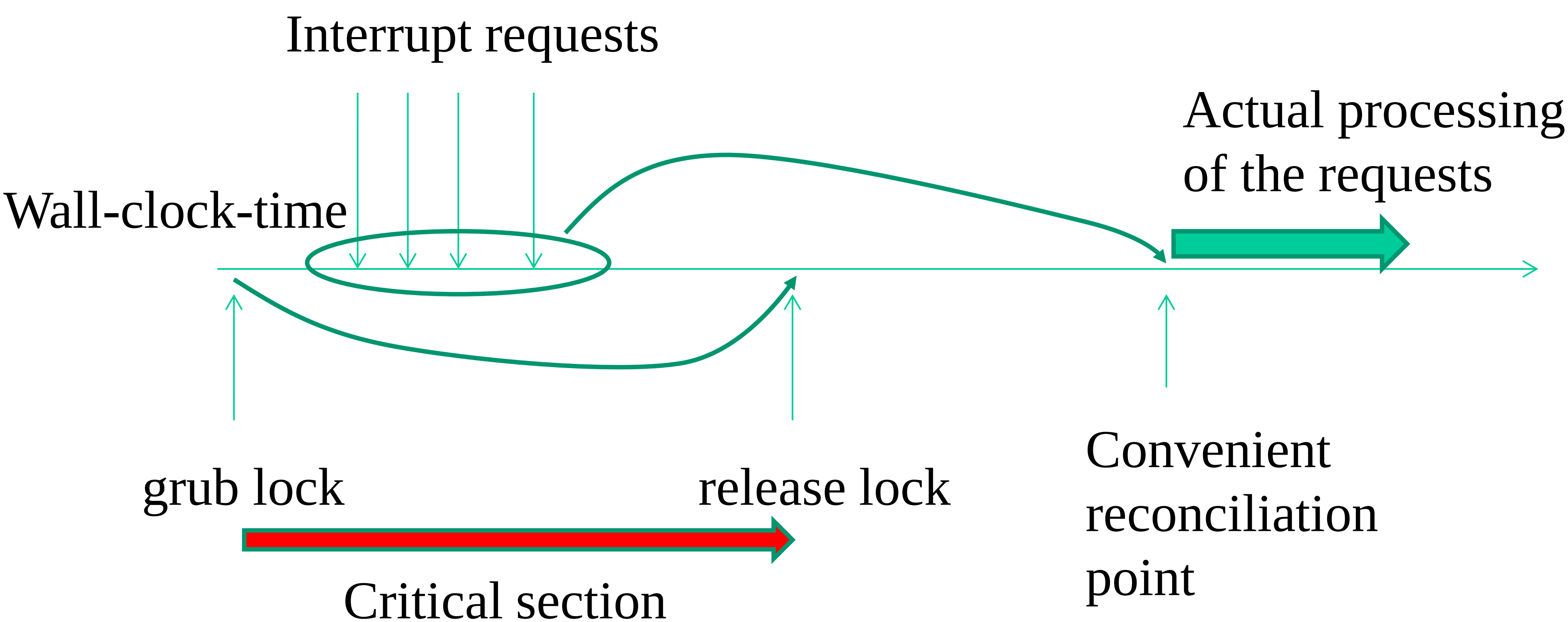
- Types of traces
  - User mode process/thread
  - Kernel mode process/thread
  - Interrupt management
- Non-determinism
  - Due to nesting of user/kernel mode traces and interrupt management traces
- Performance
  - Non-determinism may give rise to inefficiency whenever the evolution of the traces is tightly coupled (like on SMP and multi-core machines)
  - **<u>Timing expectations for critical sections can be altered</u>**

Design methodologies

Temporal reconciliation

- Interrupt management traces get nested into (mapped onto) process/thread traces according to temporal shift (**work deferring**)
- This mapping can lead to aggregating the management of the events within the system (many-to-one aggregation)
- Priority based scheduling mechanisms are required in order not to induce starvation, or to correctly manage different levels of criticality

An example timeline for work deferring



Reconciliation points

Guarantees

- “Eventually”

Conventional support

- Returning from syscall
  - This involves application level technology
- Context-switch
  - This involves idle-process technology
- Reconciliation in process-context
  - This involves kernel-thread technology

The historical concept: top/bottom half processing

- The management of **tasks associated with the interrupts** typically occurs via a two-level logic: top half e bottom half
- The top-half level takes care of executing a minimal amount of work which is needed to allow later finalization of the whole interrupt management
- The **top-half code portion is typically (but not manadatorily) handled according to a non-interruptible (hence non-preemptable) scheme**
- The finalization of the work takes place via the bottom-half level
- The top-half takes care of **scheduling the bottom-half task,** e.g., by queuing a record into a proper data structure

- The difference between top-half and bottom-half comes out because of
    - ✓ the need to manage events in a timely manner,
    - ✓ while avoiding to lock resources right upon the event occurrence
- Otherwise, we may incur the risk of delaying critical actions (**e.g. spinlock-release**) interrupted due to the event occurrence
- At worst we might even incur deadlocks when a slow interrupt management is hit by the activation of another one that needs the same resources

One example: sockets

no top/bottom half

interrupt from network device

packet extraction

IP level

TCP/UDP level

VFS Level

additional delay for, e.g., an active spin-lock

top/bottom half

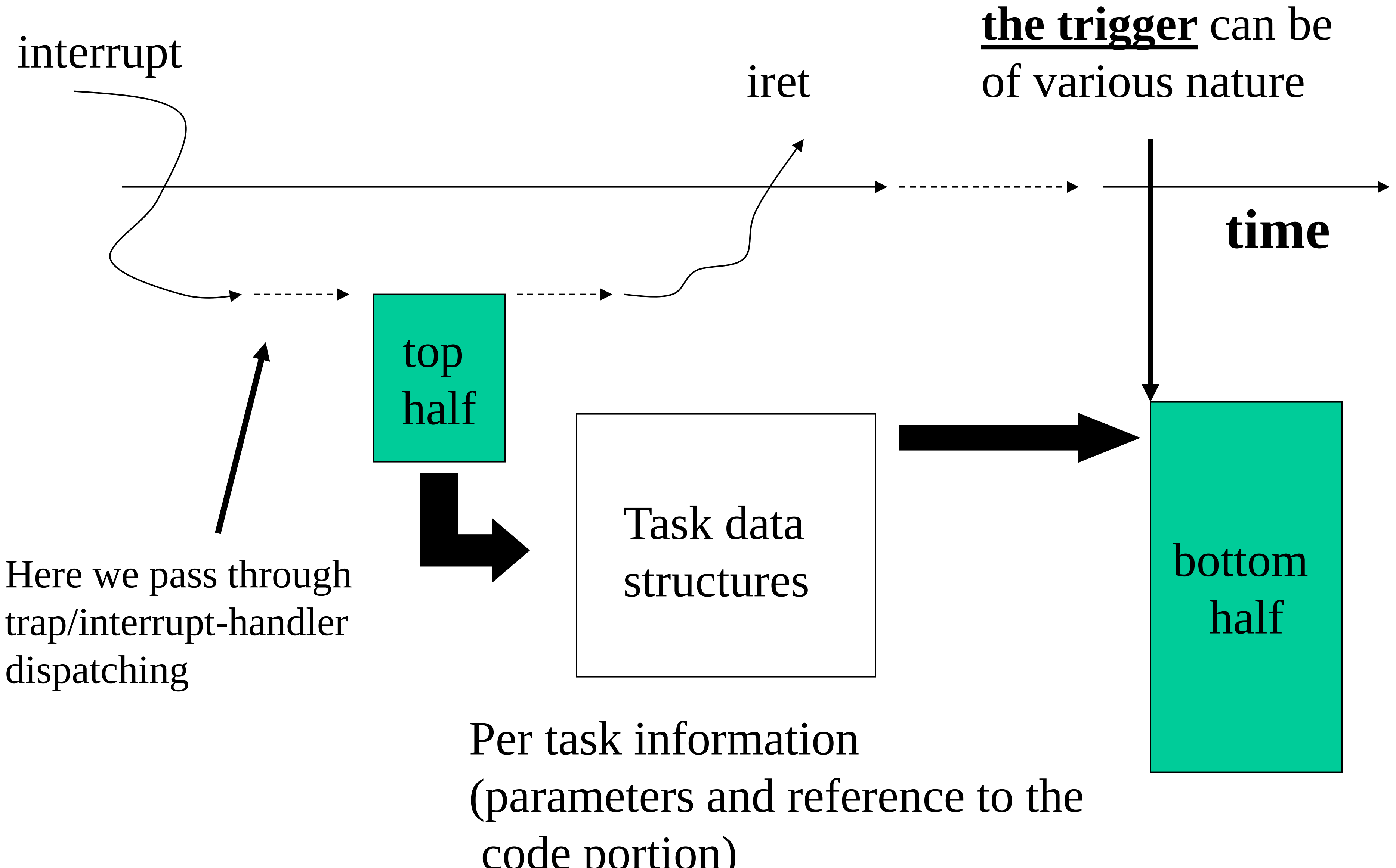
interrupt from network device

packet extraction

Task queuing

additional delay for, e.g., an active spin-lock

The historical architectural concept: bottom-half queues



Historical evolution in LINUX

Improved orientation to SMP/multi-core and automation (concepts that relevant to every operating system kernel so we can take the LINUX instances as archetypal solutions)

Task queues

Softirqs
Tasklets
Work queues

Kernel version

2.5

Let's start from task queues

- task-queues are queuing structures, which can be associated with variable names
- LINUX (ref. kernel 2.2) already declared a given amount of <u>**predefined task-queues**</u>, having the following names
  - `tq_immediate`

    (tasks to be executed upon timer-interrupt or syscall return)
  - `tq_timer`

    (tasks to be executed upon timer-interrupt)
  - `tq_schedule`

    (task to be executed in <u>process context</u>)

Task queues data structures

- Additional task queues can be declared using the macro `DECLARE_TASK_QUEUE(queuename)` which is defined in `include/linux/tqueue.h` – this macro also initializes the task-queue as empty

- The structure of a task is defined in `include/linux/tqueue.h`

```
struct tq_struct {
    struct tq_struct *next; /*linked list of active bh's*/
    int sync; /* must be initialized to zero */
    void (*routine)(void *); /* function to call */
    void *data; /* argument to function */
}
```

Task management API

- The queuing function has prototype `int queue_task(struct tq_struct *task, task_queue *list)`, where `list` is the address of the target task-queue structure
- This function is used to only register the task, not to execute it
- The task flushing (execution) function for all the tasks currently kept by a task queue is `void run_task_queue(task_queue *list)`
- When invoked, unlinking and actual execution of the tasks takes place
- For the `tq_schedule` task-queue there exists a proper queuing function offered by the kernel with prototype `int schedule_task(struct tq_struct *task)`
- **The return value of any queuing function is non-zero if the task is not already registered within the queue** (the check is done by exploiting the `sync` field, which gets set to 1 when the task is queued)

Task management details

- Non-predefined task-queues need to be flushed via an explicit call to **<u>the function</u>** `run_task_queue(...)`
- Pre-defined task-queues are automatically handled (flushed) by the kernel
- Anyway, pre-defined queues can be used for inserting tasks that may differ from those natively inserted by the standard kernel image
- **<u>Note</u>**: upon inserting a task into the `tq_immediate` queue, a call to `void mark_bh(IMMEDIATE_BH)` needs to be made, which is used to set the data structures in such a way to indicate that this is not empty
- This needs to be done in relation to legacy management rules

Bottom-half occurrences with task queues

Timely flushing of the bottom halves requires

- Invokation by the scheduler
- Invokation upon entering and/or exiting system calls

The Linux kernel (up to 2.5) invokes **do_bottom_half()**

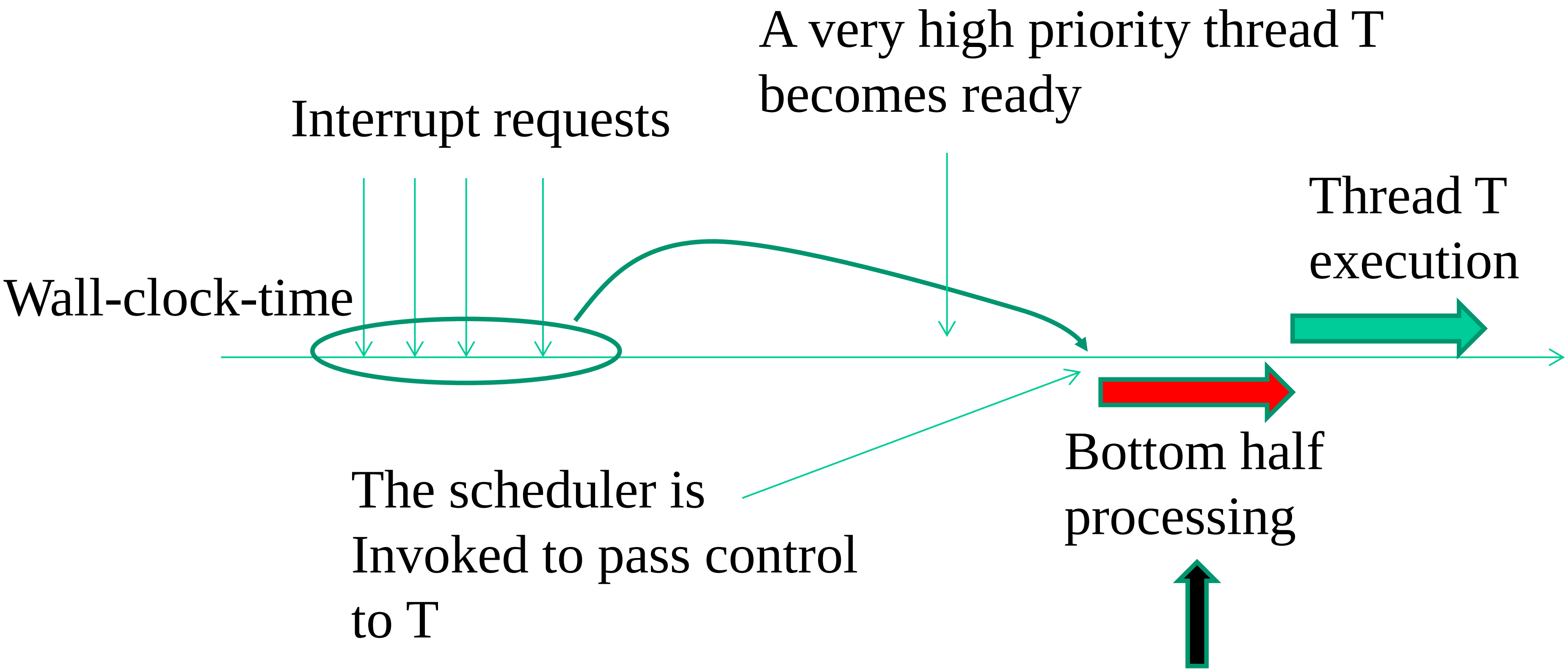
- within schedule()
- from ret_from_sys_call()

Be careful: bottom half execution context

- Even though bottom half tasks can be executed in process context, the actual context for the thread running them should look like “interrupt”

- No blocking service invocation in any bottom half function!!

Limitations of task queues: the timeline



Thread T is delayed by the whole time require to process all the standing bottom halves

Limitations of task queues: more general aspects

- Nesting of bottom halves on a single thread leads to
  - ✓ The impossibility to exploit multiple CPU-cores for interrupt (bottom half) management
  - ✓ The impossibility to optimize locality of operations and data accesses
  - ✓ Unsuitability for heavy interrupt load
  - ✓ Unsuitability for scaled up hardware parallelism

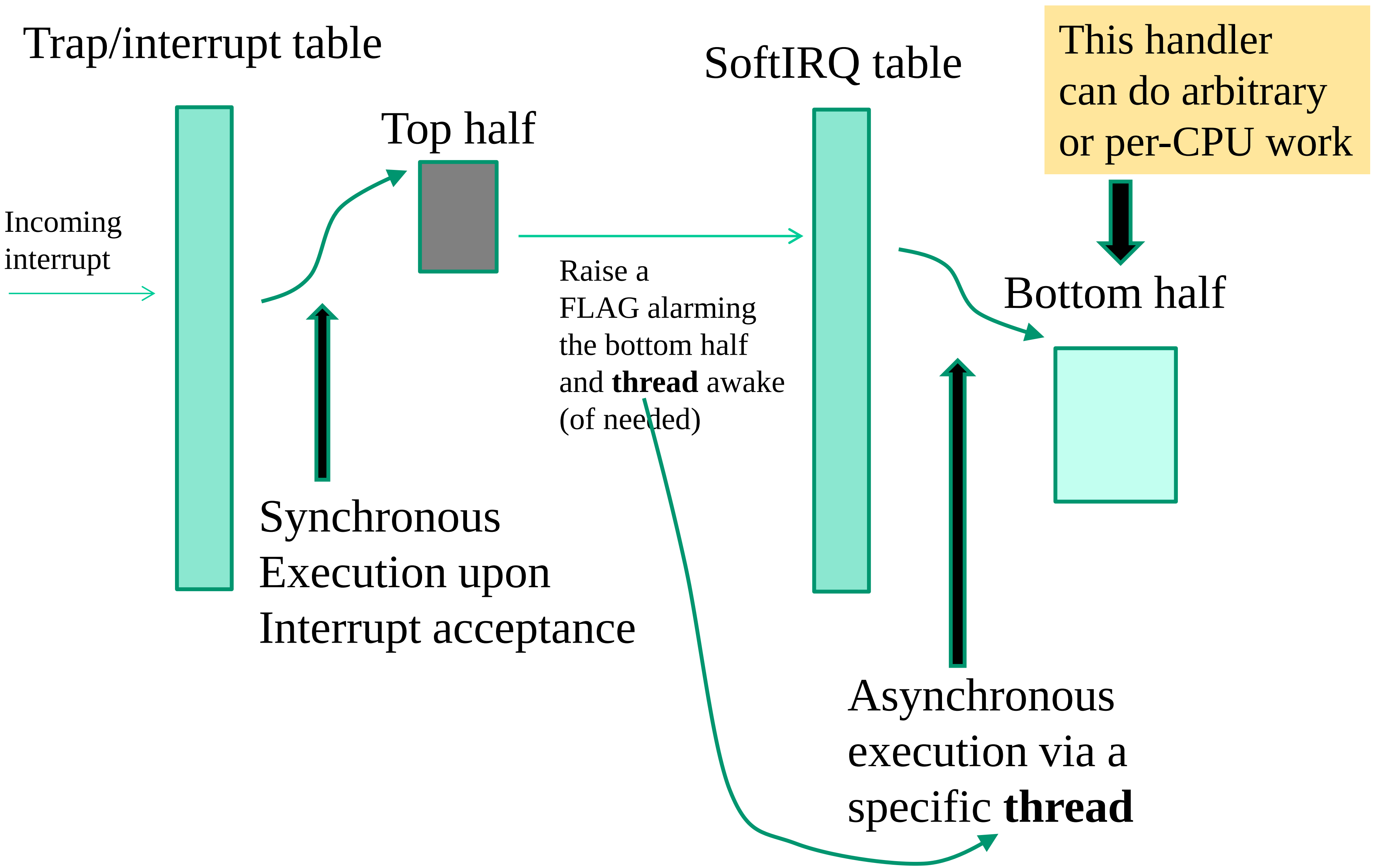
Parallelism vs interrupts vs device drivers

- Interrupts can be also be raised by software
- So interrupt drivers my be requested to handle a load that may grow with the number of running threads
- Clearly, the actual workload is can be a function of the number of available CPU-cores
- This is the scenario o drivers for logical (not physical) devices
- Overall, we need:
  - ✓ More scalability and locality
  - ✓ More flexibility
  - ✓ Reactiveness and predictability

SoftIRQ architectures

- The top half is further reduced
- It does not necessarily queue the bottom half, so it can be even more responsive
- Bottom halves can therefore be already present
- They can be are seen as actual interrupt handlers triggered via software (by the top half)
- The queuing concept is still there for on demand usage, of required
- Queues of tasks are not queues of bottom halves, they are queues of bottom half input data

The architectural scheme



LINUX SoftIRQs (kernels later than 2.5)

- The SoftIRQ table is an array of `NR_SOFTIRQS` entries, each of which is set to identify a `struct softirq_action`
- The entries are associated with different types/priorities of handlers, the set is:

```
enum {    HI_SOFTIRQ=0,
          TIMER_SOFTIRQ,
          NET_TX_SOFTIRQ,
          NET_RX_SOFTIRQ,
          BLOCK_SOFTIRQ,
          BLOCK_IOPOLL_SOFTIRQ,
          TASKLET_SOFTIRQ,
          SCHED_SOFTIRQ,
          HRTIMER_SOFTIRQ,
          RCU_SOFTIRQ,
          NR_SOFTIRQS  }
```

High priority queued stuff

Stuff to do on timers or reschedules

Normal priority queued stuff

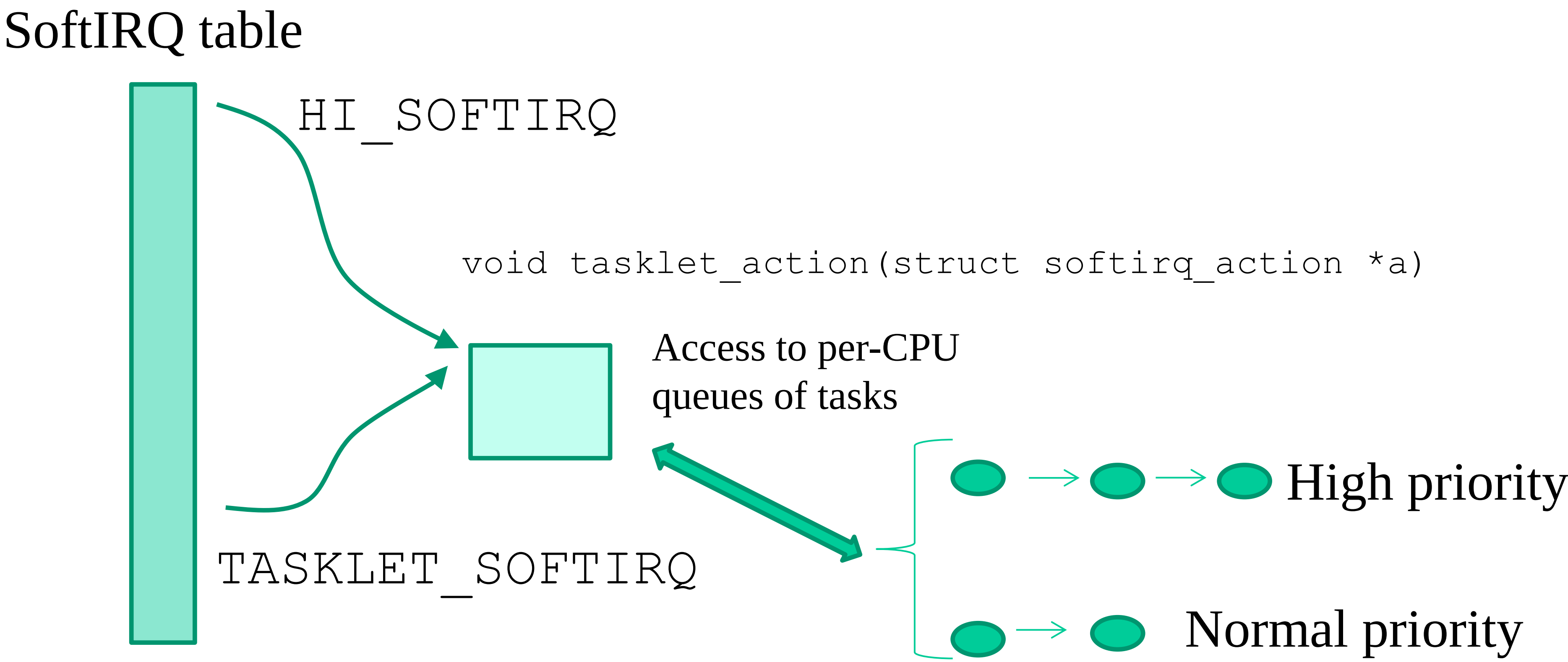
Who does the softIRQ work

- The `ksoftirq` daemon (multiple threads with CPU affinity)
- This is typically listed as `ksoftirq[n]` where ‘n’ is the CPU-core it is affine with
- Once awaken, the threads look at the softIRQ table to inspect if some entry is flagged
- In the positive case the threads runs the softIRQ handler (please check with `/proc/softirqs` for the audit)
- We can also build a mask telling that a thread awaken on a CPU-core X will not process the handler associated with a given softIRQ
- So we can create affinity between softIRQs anf CPU-cores
- On the other hand, affinity can be based on groups of CPU-core IDs so we can distribute the SoftIRQ load across the COU-cores

Overall advantages from softIRQs

- Multithread execution of bottom half tasks
- Bottom half execution not synchronous with respect to specific threads (e.g. upon rescheduling a very high priority thread)
- Binding of task execution to CPU-cores if required (e.g. locality on NUMA machines)
- Ability to still queue tasks to be done (see the `HI_SOFTIRQ` and `TASKLET_SOFTIRQ` types)

Actual management of queued tasks: normal and high priority tasklets



Tasklet representation and API

- The tasklet is a data structure used for keeping track of a specific task, related to the execution of a specific function internal to the kernel
- The function can accept a single pointer as the parameter, namely an `unsigned long`, and must return `void`
- Tasklets can be instantiated by exploiting the following macros defined in include `include/linux/interrupt.h`:
  - `DECLARE_TASKLET(tasklet, function, data)`
  - `DECLARE_TASKLET_DISABLED(tasklet, function, data)`
- `name` is the taskled identifier, `function` is the name of the function associated with the tasklet and `data` is the parameter to be passed to the function
- If instantiation is disabled, then the task will not be executed until an explicit enabling will take place

- tasklet enabling/disabling functions are

```
tasklet_enable(struct tasklet_struct *tasklet)

tasklet_disable(struct tasklet_struct *tasklet)

tasklet_disable_nosynch(struct tasklet_struct *tasklet)
```

- the functions scheduling the tasklet are

```
void tasklet_schedule(struct tasklet_struct *tasklet)

void tasklet_hi_schedule(struct tasklet_struct
  *tasklet)

void tasklet_hi_schedule_first(struct tasklet_struct
  *tasklet)
```

- **NOTE:**
  - Subsequent reschedule of a same tasklet may result in a single execution, depending on whether the tasklet was already flushed or not

The tasklet init function

```
void tasklet_init(struct tasklet_struct *t, void
(*func)(unsigned long), unsigned long data) {

      t->next = NULL;

      t->state = 0;

      atomic_set(&t->count, 0);

      t->func = func;

      t->data = data;

}
```

This enables/disables the tasklet

Important note

- A tasklet that is already queued and is not active still stands in the pending tasklet list, up to its enabling and then processing

- This is clearly important when we implement device drivers with tasklets in LINUX modules and we want to unmount the module for any reason

- In other words we must be very careful that queue linkage is no broken upon the unmount

Tasklets' recap

- Tasklets related tasks are performed <u>via specific kernel threads</u> (CPU-affinity can work here when logging the tasklet)
- If the tasklet has already been scheduled on a different CPU, it will not be moved to another CPU if it's still pending (this is instead allowed for softirqs)
- Tasklets have schedule level similar to the one of `tq_schedule`
- The main difference is that the thread actual context should be an "interrupt-context" – thus with no-sleep phases within the tasklet (an issue already pointed to)

Finally: work queues

- Kernel 2.5.41 fully replaced the task queue with the work queue
- Users (e.g. drivers) of `tq_immediate` should normally switch to tasklets
- Users of `tq_timer` should use timers directly
- If these interfaces are inappropriate, the `schedule_work()` interface can be used
- This interface queues the work to the kernel “events” (multithread) daemon, which executes it in process context
- Interrupts enabled while the work queues are being run (except if the same work to be done disables them)
- Functions called from a work queue may call blocking operations, but this is discouraged as it prevents other users from running (an issue already pointed to)

Work queues basic interface (default queues)

```
schedule_work(struct work_struct *work)
schedule_work_on(int cpu,
                struct work_struct *work)
```

```
INIT_WORK(&var_name, function-pointer, &data);
```

Additional APIs can be used to create custom work queues and to manage them

```
struct workqueue_struct *create_workqueue(const
char *name);

struct workqueue_struct
     *create_singlethread_workqueue(const char
*name);
```

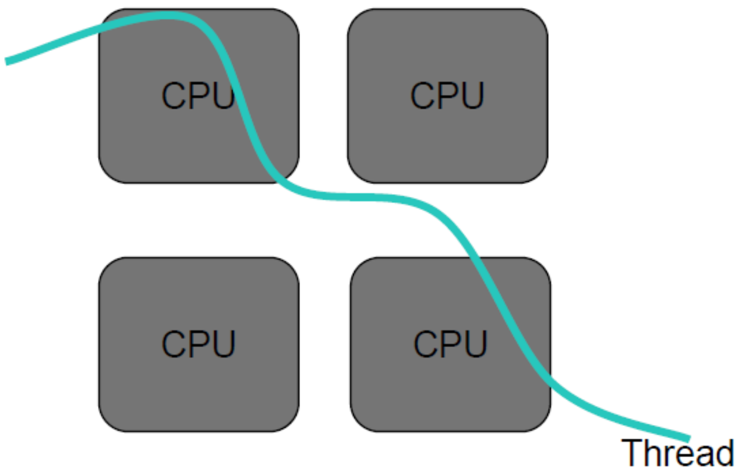
Both create a workqueue_struct (with one entry per processor)
The second provides the support for flushing the queue via a single worker thread (and no affinity of jobs)

```
void destroy_workqueue(struct workqueue_struct
*queue);
```

This eliminates the queue

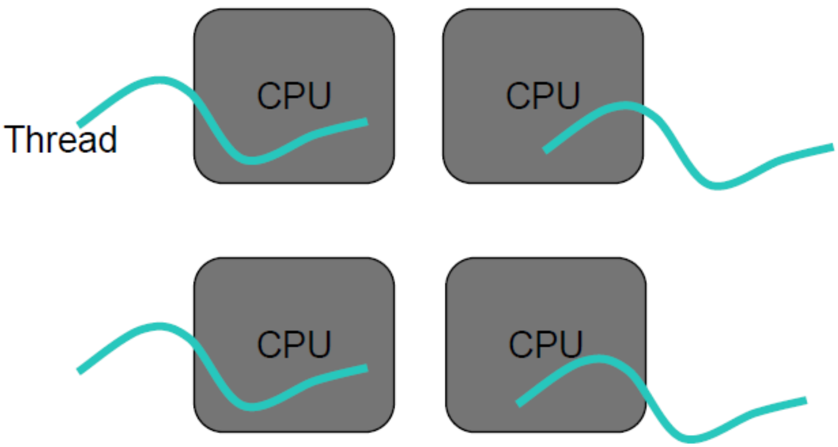
Actual scheme

Single threaded workqueue



A single threaded workqueue had one worker thread system-wide.

Multi threaded workqueue



A multi threaded workqueue had one thread per CPU.

```
int queue_work(struct workqueue_struct *queue,
            struct work_struct *work);

int queue_delayed_work(struct workqueue_struct *queue,
            struct work_struct *work, unsigned long delay);
```

Both queue a job - the second with timing information

```
int cancel_delayed_work(struct work_struct *work);
```

This cancels a pending job

```
void flush_workqueue(struct workqueue_struct *queue);
```

This runs any job

Work queue issues

➔ **Proliferation of kernel threads** The original version of workqueues could, on a large system, run the kernel out of process IDs before user space ever gets a chance to run

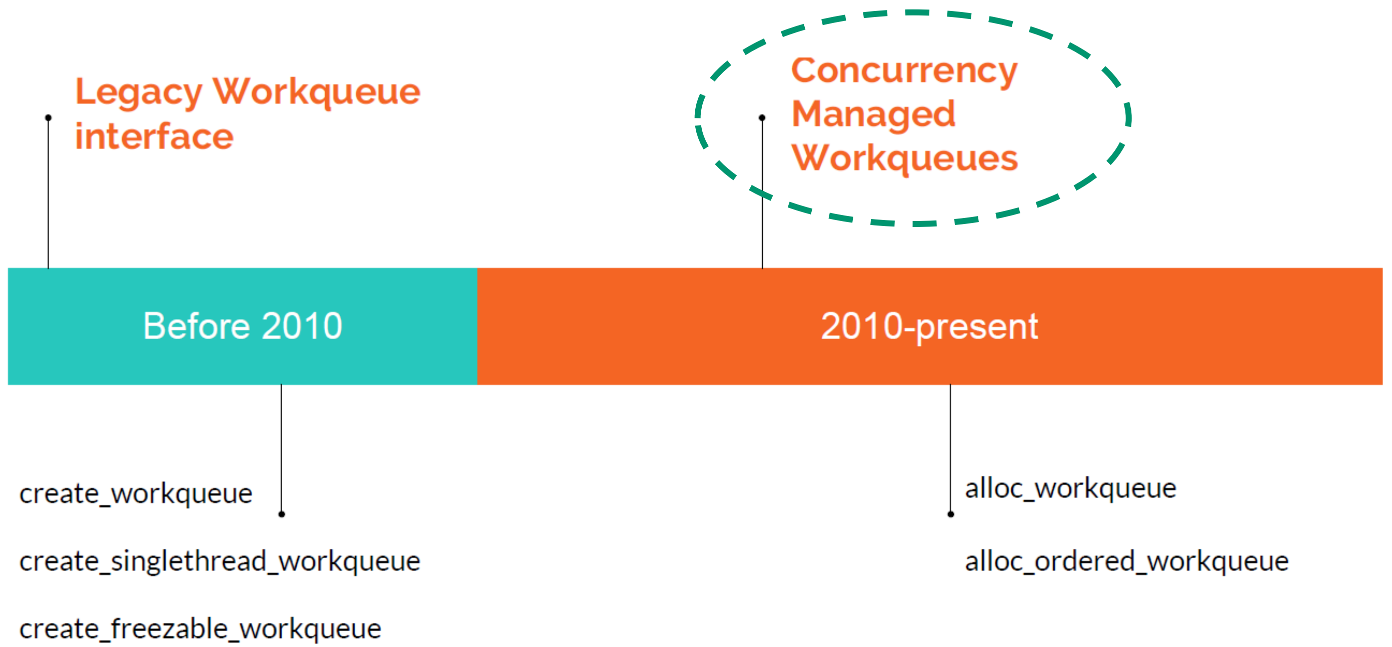
➔ **Deadlocks** Workqueues could also be subject to deadlocks if locking is not handled very carefully

➔ **Unnecessary context switches** Workqueue threads contend with each other for the CPU, causing more context switches than are really necessary

Interface and functionality evolution

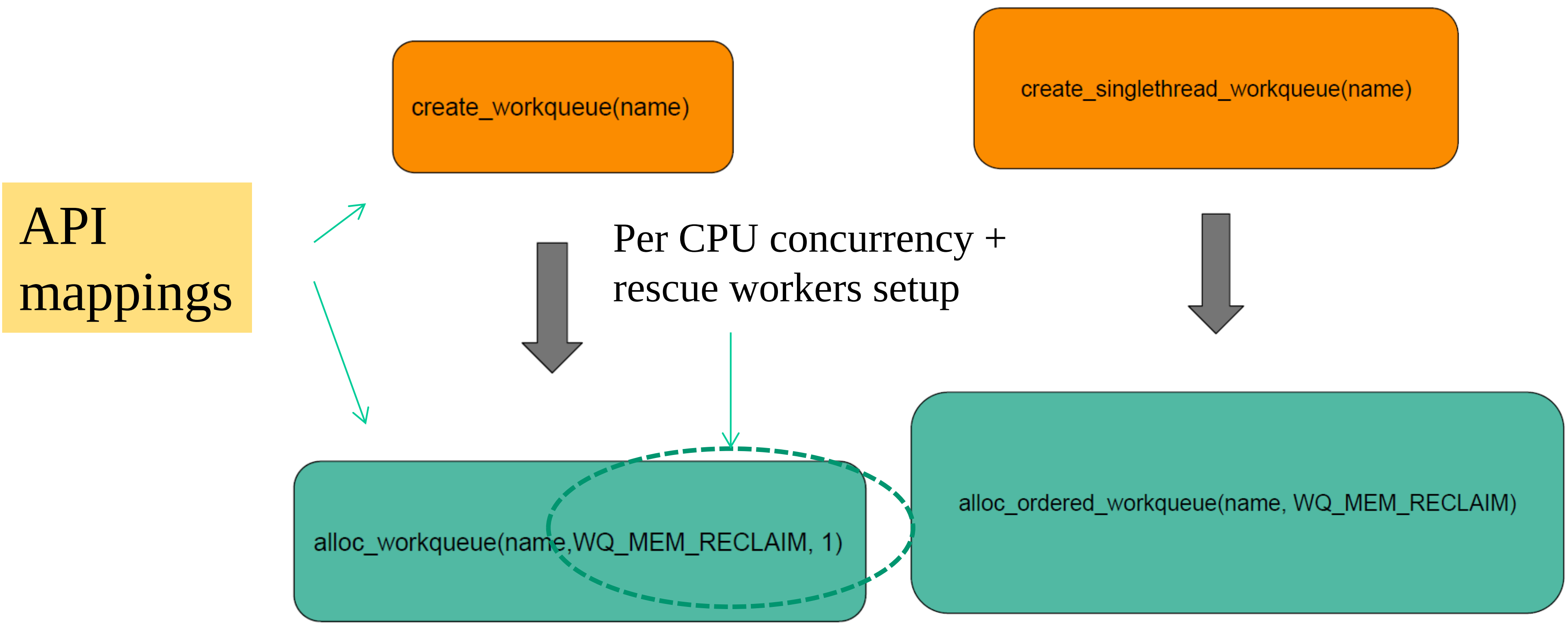
Due to its development history, there currently are two sets of interfaces to create workqueues.

- **Older**: `create[_singlethread|_freezable]_workqueue()`
- **Newer**: `alloc[_ordered]_workqueue()`



Concurrency managed work queues

- Uses per-CPU unified worker pools shared by all wq to provide flexible level of concurrency on demand without wasting a lot of resources
- <u>Automatically regulates worker pool</u> and level of concurrency so that the API users don't need to worry about such details.



Managing dynamic memory with (not only) work queues