Advanced Operating Systems (and System Security)
MS degree in Computer Engineering
University of Rome Tor Vergata
Lecturer: Francesco Quaglia

Kernel level task management

1. Advanced/scalable task management schemes
2. (Multi-core) CPU scheduling approaches
3. Kernel level threads
4. Automatic concurrency managers
5. Binding to the Linux architecture

Tasks vs processes/threads

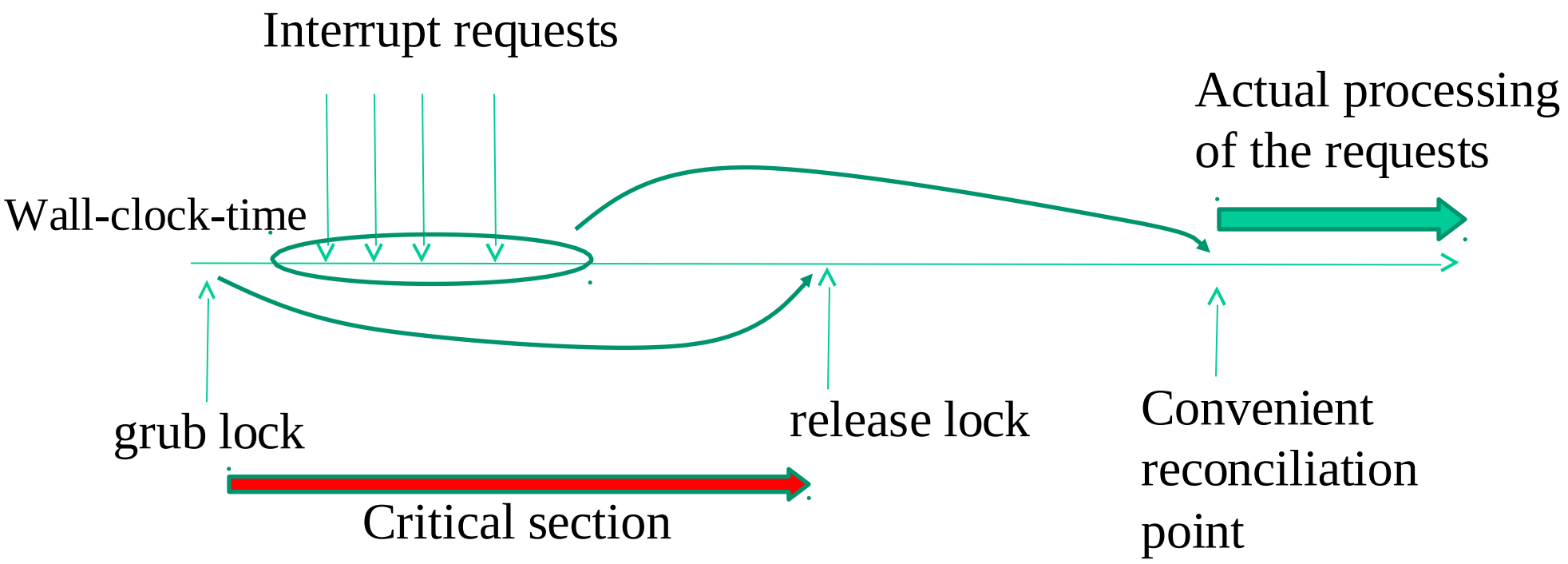
- Types of traces
  - User mode process/thread
  - Kernel mode process/thread
  - Interrupt management
- Non-determinism
  - Due to nesting of user/kernel mode traces and interrupt management traces
- Performance
  - Non-determinism may give rise to inefficiency whenever the evolution of the traces is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

Design methodologies

<u>Temporal reconciliation</u>

- 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 with 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 programming (i)

- The management of **tasks associated with the interrupts** typically occurs via a two-level logic → top half and 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 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

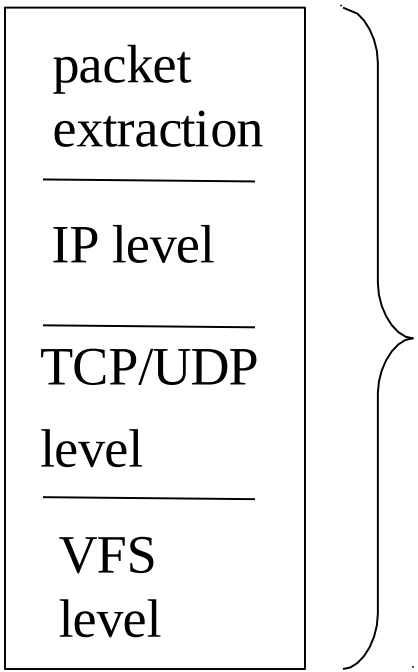
The historical concept - top/bottom half programming (ii)

- The difference between top-half and bottom-half comes out because of
  - ✓ the need to manage events in a timely manner
  - ✓ while avoiding to keep locked 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

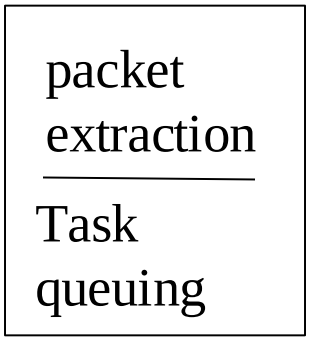
interrupt from network device



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

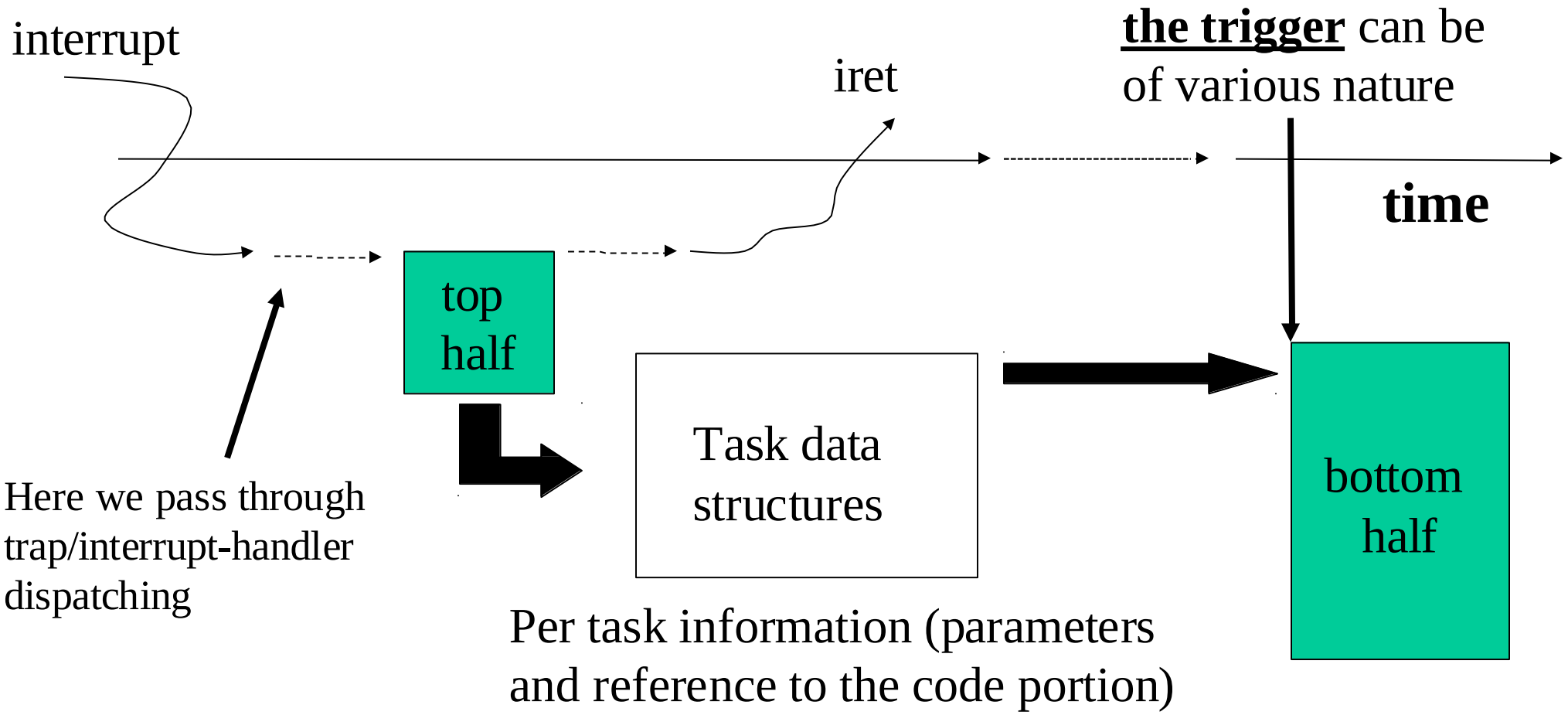
top/bottom half

interrupt from network device



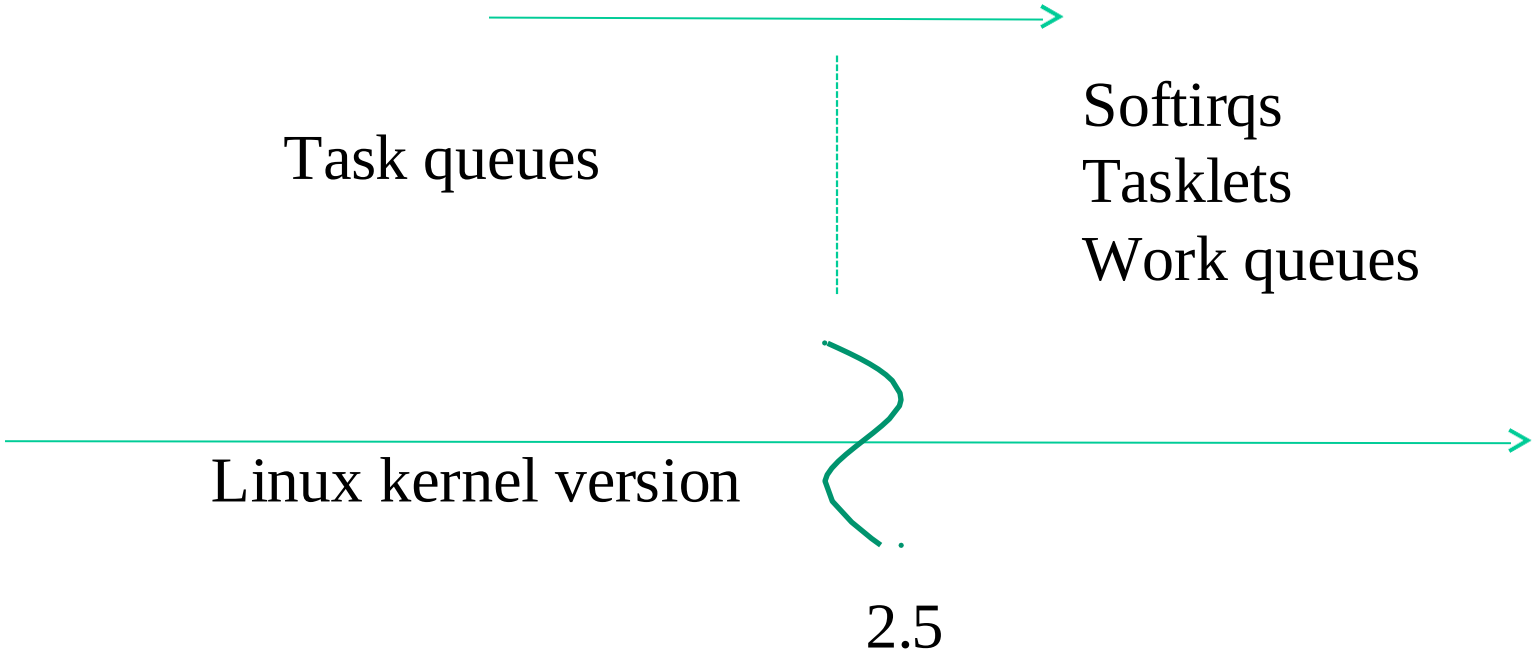
additional delay for, e.g., an active spin-lock

The historical architectural concept - bottom-half queues



Historical evolution in Linux

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



Let’s start from task queues

- task-queues are queuing structures, which can be associated with variable names
- Linux (ref. kernel 2.2) already declares a given amount of **predefined task-queues**, 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`

    (tasks to be executed in process context)

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 */
}
```

These are the task specific fields

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-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 take 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)`
- **<u>The return value of any queuing function is non-zero if the task is not already registered within the queue</u>** (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 **<u>an explicit call to 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

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()`**

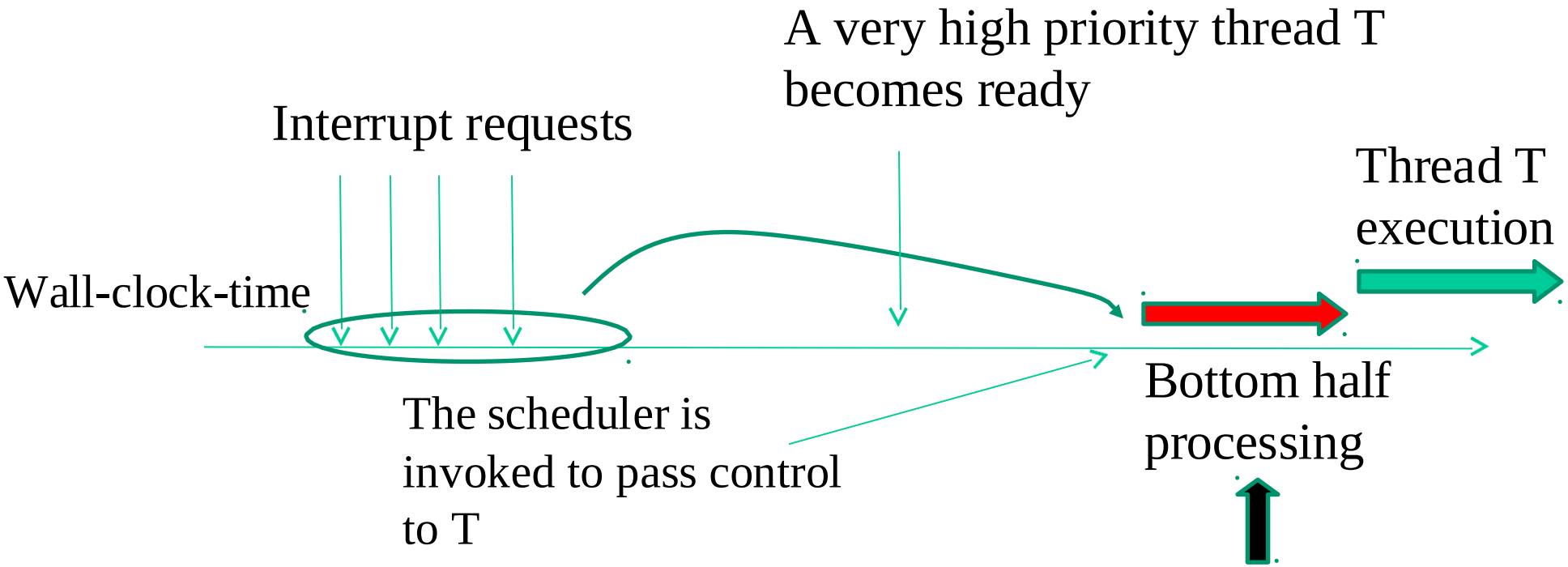
- within `schedule()`
- from `ret_from_sys_call()`

Be careful - the bottom half execution context

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

- <u>No blocking service invocation</u> in any bottom half function!!

Limitations of task queues - the actual timeline



Thread T is delayed by the whole time required 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

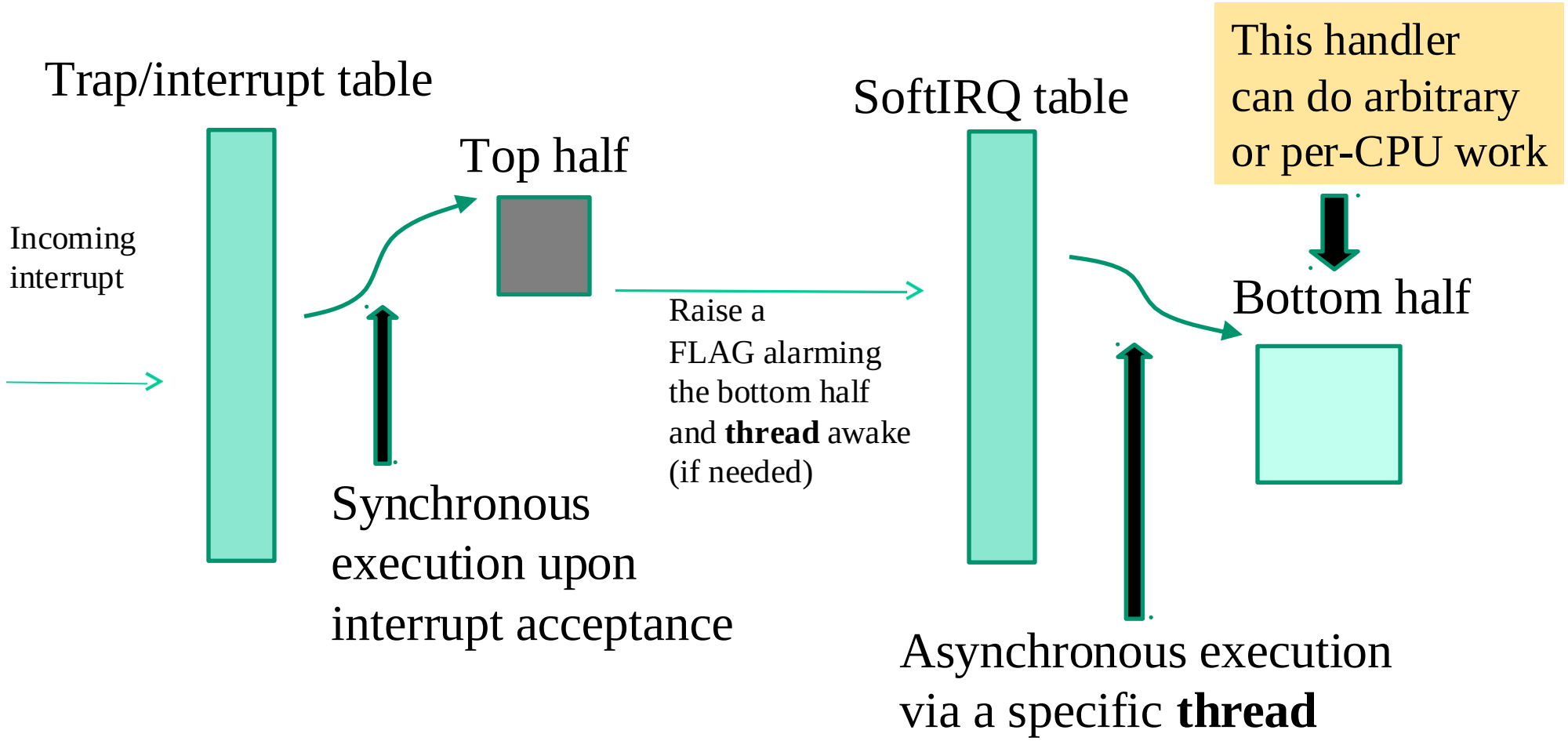
Parallelism vs interrupts vs device drivers

- “Interrupts” can also be raised by software
- This is the scenario of drivers for logical (not physical) devices
- Interrupt drivers may be requested to handle a load that may grow with the number of running threads (larger than the number of devices!?!)
- Clearly, the actual workload can be a function of the number of available CPU-cores
- 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 somewhere
- They can be seen as actual interrupt handlers triggered via software (by the top half)
- The queuing concept is still there for on demand usage, if required (e.g. for programmability of new bottom halves)
- 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 thread runs the SoftIRQ handler
- We can also build masks for avoiding the redirect and the processing of specific interrupts within SoftIRQ
- So we can create affinity between SoftIRQs and 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 CPU-cores

An example on interrupt affinity management

- Currently Linux tells us its current affinity of an interrupt line towards the CPUs via the pseudofile

  /proc/irq/$IRQ_NUMBER/smp_affinity

- These pseudofiles can be reconfigured so as to setup

  – new affinity for (in generic) interrupts

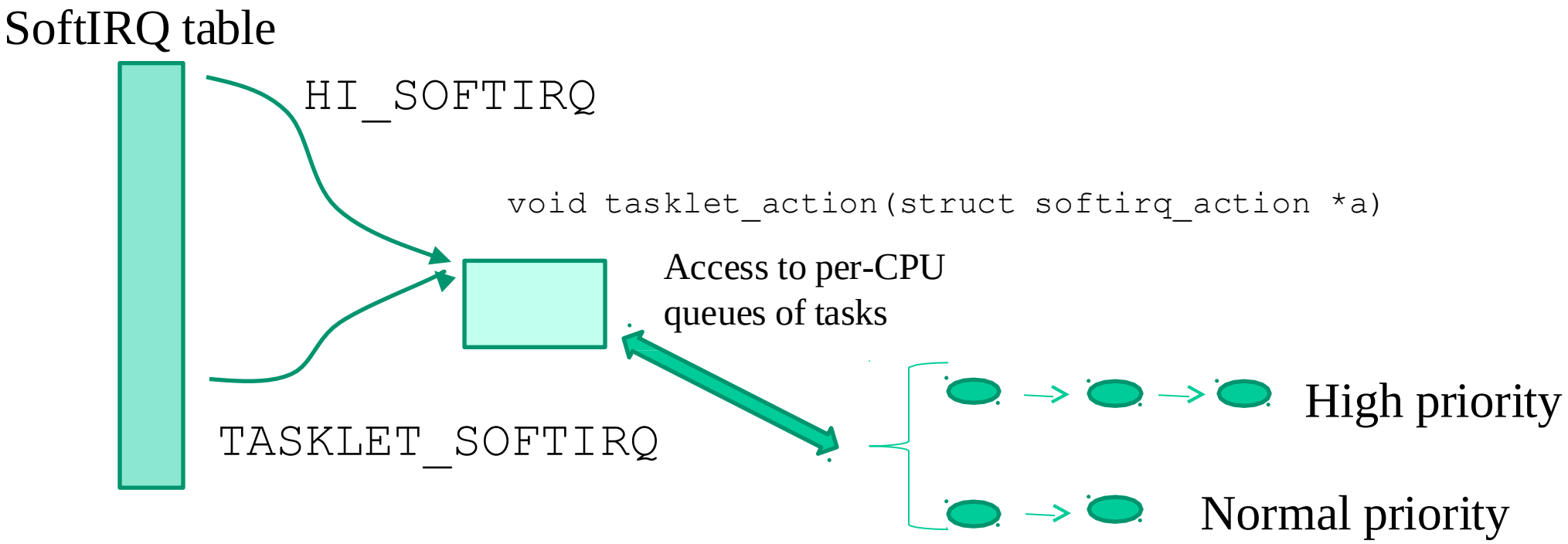
  – and (then) the affinity for the interrupts handled via SoftIRQ

- …… additional details will be coming when we will discuss the architectural organization of the interrupt system ….

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 (i)

- 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 explicitly enabled

Tasklet representation and API (ii)

- 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)
```

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, e.g., 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 not 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-core, it will not be moved to another CPU-core if it is still pending (generic softirqs can instead be processed by different CPU-cores)
- Tasklets have schedule level similar to the one of `tq_schedule`
- The main difference is that <u>the thread actual context should be an "interrupt-context"</u> – 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 (we will see this in a while)
- If these interfaces are inappropriate, the schedule_work() interface can be used
- This interface queues the work to the kernel “events” (multithreaded) daemon, which executes it in process context

… work queues continued

- Interrupts are 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)
- The above point is anyhow tackled by more recent variants of work queues as we shall see

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);
```

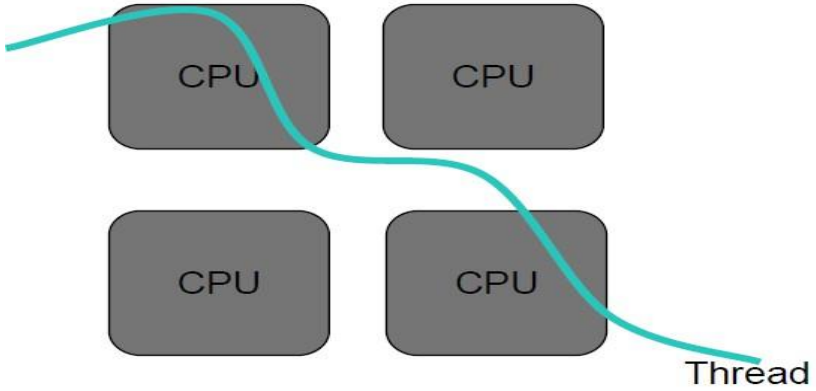
Both create a workqueue_struct (with one entry per processor)
The second provides the support for flushing the queue via a single worker thread (and no affinity of jobs)

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

This eliminates the queue

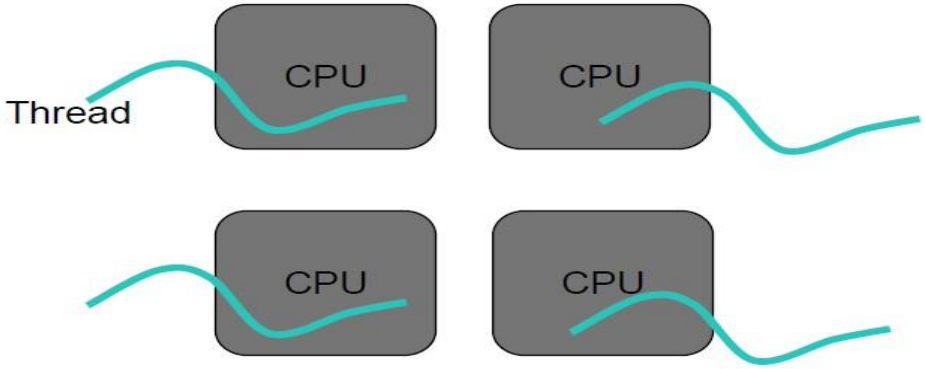
Actual scheme

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 resource usage 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

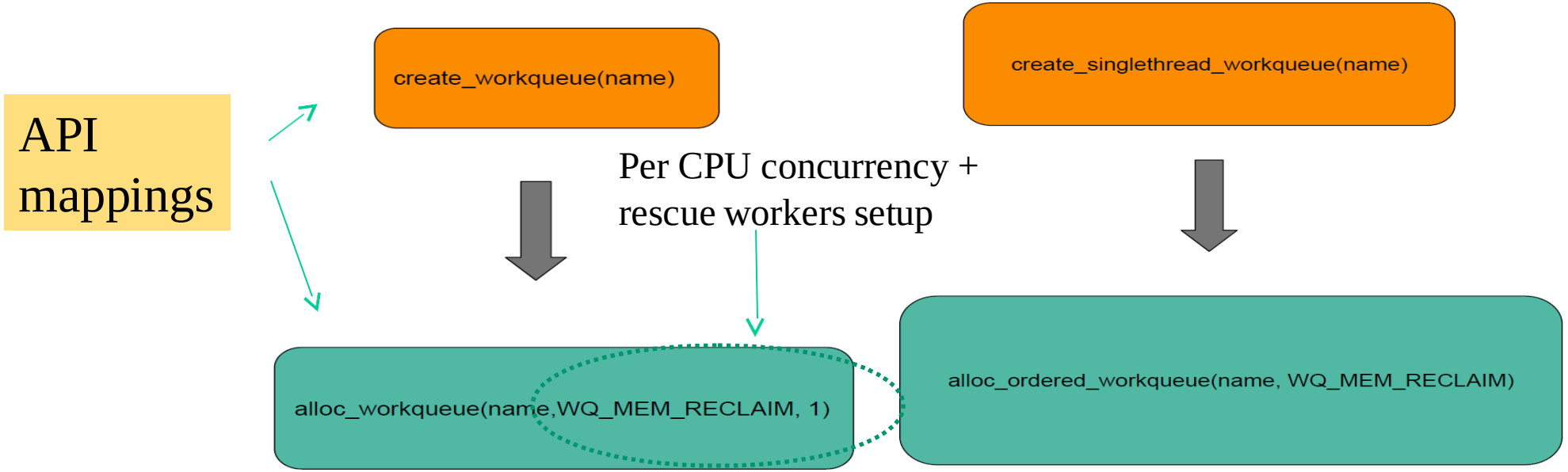
Interface and functionality evolution

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

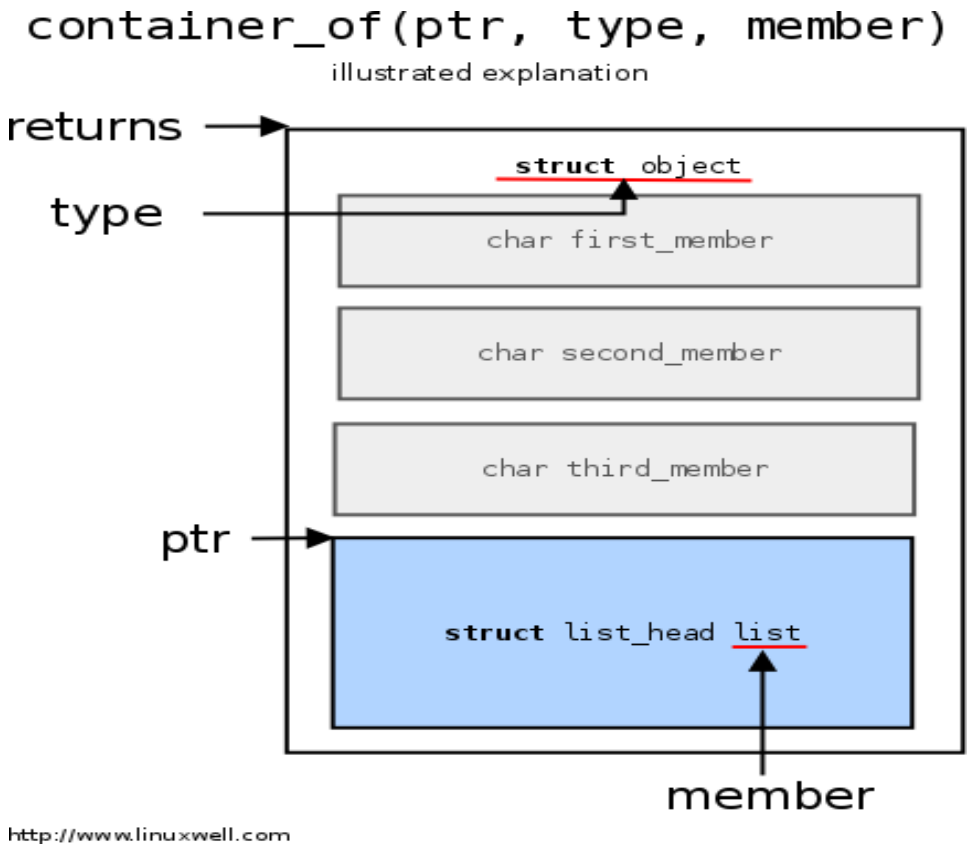
- **Older**: `create[_singlethread|_freezable]_workqueue()`
- **Newer**: `alloc[_ordered]_workqueue()`

Concurrency managed work queues

- Uses per-CPU unified worker pools shared by all work queues to provide flexible levels of concurrency on demand without wasting a lot of resources
- Automatically regulates the worker pool and level of concurrency so that the users don't need to worry about such details



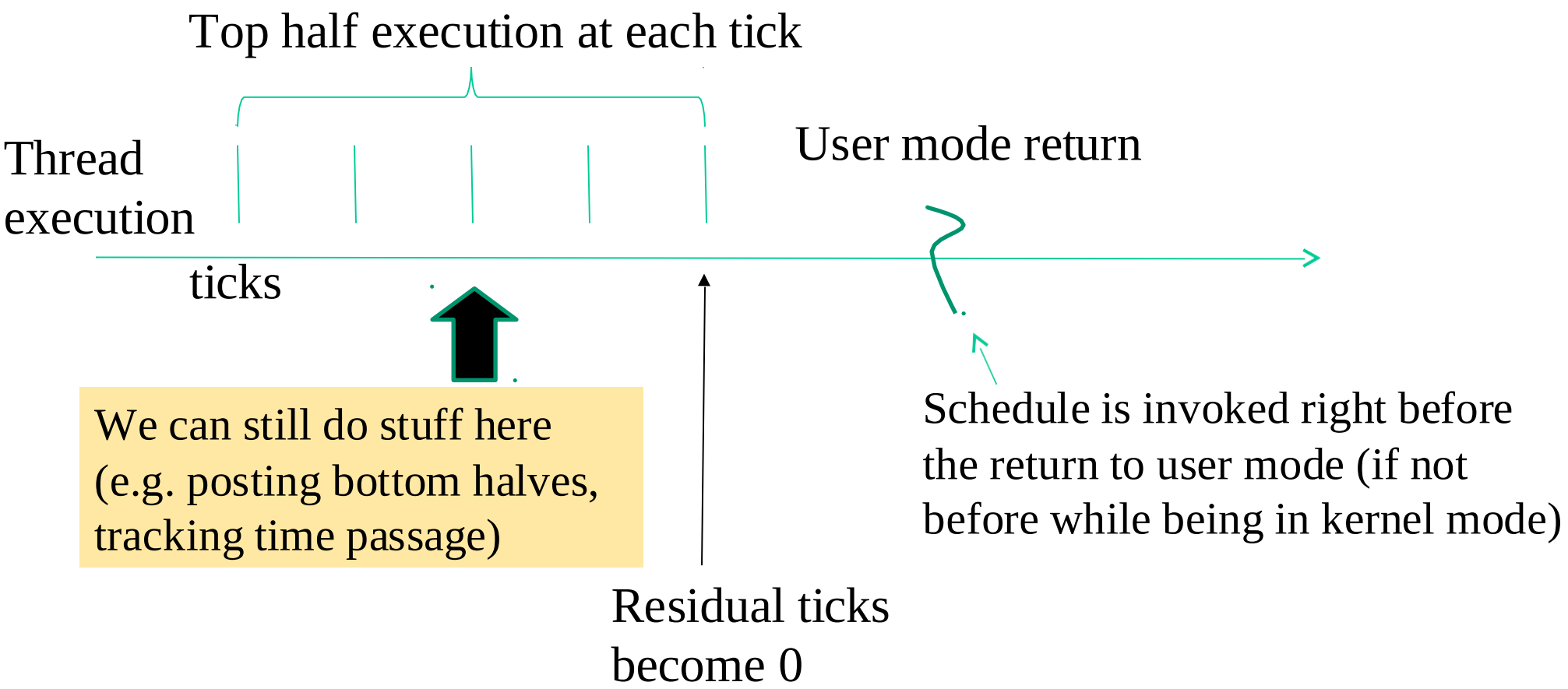
Managing dynamic memory with (not only) work queues



Interrupts vs passage of time vs CPU-scheduling

- The unsuitability of processing interrupts immediately (upon their asynchronous arrival) still stands there for TIMER interrupts
- Although we have historically abstracted a context switch off the CPU caused by the time-quantum expiration as an asynchronous event, it is not actually true
- <u>What changes asynchronously is the condition that tells to the kernel software if we need to synchronously</u> (at some point along execution in kernel mode) <u>call the CPU scheduler</u>
- Overall, timing vs CPU reschedules are still managed according to a top/bottom half scheme
- **NOTE => this is not true for preemption not linked to time passage, as we shall see**

A scheme for timer interrupts vs CPU reschedules



Could we be still effective disabling the timer interrupt on demand?

- Clearly no!!
- If we disable (timer) interrupts while running a kernel block of code that absolutely needs not to be preempted by the timer <u>we loose the possibility to schedule bottom halves along time passage</u>
- We also <u>loose the possibility to control timings and more generally events at fine grain</u>, which is fundamental on a multi-core system
- A CPU-core can in fact at fine grain interact with the others
- Switching off timer interrupts was an <u>old style approach</u> for atomicity of kernel actions on single-core/single-HT CPUs

A note on kernel mode execution vs busy waiting

- By the top/bottom half approach to handle timer-based reschedules, pure busy waiting on unguaranteed timeliness of changes of the corresponding condition is unsuitable in kernel mode

```
while (!condition) ; //this may lead to be trapped into
this block of code unlimited time
```

- A case is when the condition can only be fired by a time-shared thread

- This problem is there also for CONFIG_PREEMPT=y kernels (since we can anyhow disable preemption along any thread at any point in time, hence before that while())

What hardware timers do we have on board right now?

- Let's check with the x86 case (just limited to a few main components)
  - ✓ Time Stamp Counter (TSC) – It counts the number of CPU clocks (accessible via the `rdtsc` instruction)
  - ✓ Local APIC TIMER (LAPIC-T) – It can be programmed to send one shot or periodic interrupts, it is usually exploited for milliseconds timing and time-sharing
  - ✓ High Precision Event Timer (HPET) - It is a suite of timers that can be programmed to send one shot or periodic interrupts, it is usually exploited for nanoseconds timing

Linux timer (LAPIC-T) interrupts - the top half

- The top half executes the following actions
  - ➢ **<u>Flags the task-queue</u>** `tq_timer` as ready for flushing (old style)
  - ➢ Increments the global variable `volatile unsigned long jiffies` (declared in `kernel/timer.c`), which takes into account **the number of** ticks elapsed since interrupts' enabling
  - ➢ Does some minimal time-passage related work
  - ➢ **<u>It checks whether the CPU scheduler needs to be activated</u>**, and in the positive case flags the `need_resched/TIF_NEED_RESCHED` variable/bit via the TCB (Thread Control Block) of the current thread
- **<u>NOTE AGAIN: time passage is not the unique means for preempting threads in Linux, as we shall see</u>**

Effects of raising need_resched/TIF_NEED_RESCHED

- Upon finalizing any kernel level work (e.g. a system call) the `need_resched/TIF_NEED_RESCHED` variable/bit of the current process gets checked (recall this may have been set by the top-half of the timer interrupt)

- In case of positive check, the actual scheduler module gets activated

- It corresponds to the `schedule()` function, defined in `kernel/sched.c` (or `/kernel/sched/core.c` in more recent versions)

Timer-interrupt top-half module - old style

```
void do_timer(struct pt_regs *regs)
{
 (*(unsigned long *)&jiffies)++;
 #ifndef CONFIG_SMP
 /* SMP process accounting uses
    the local APIC timer */

 update_process_times(user_mode(regs));
 #endif
 mark_bh(TIMER_BH);
 if (TQ_ACTIVE(tq_timer))
    mark_bh(TQUEUE_BH);
}
```

Timer-interrupt bottom-half module - task queue based old style

```
void timer_bh(void)
{
 update_times();
 run_timer_list();
}
```

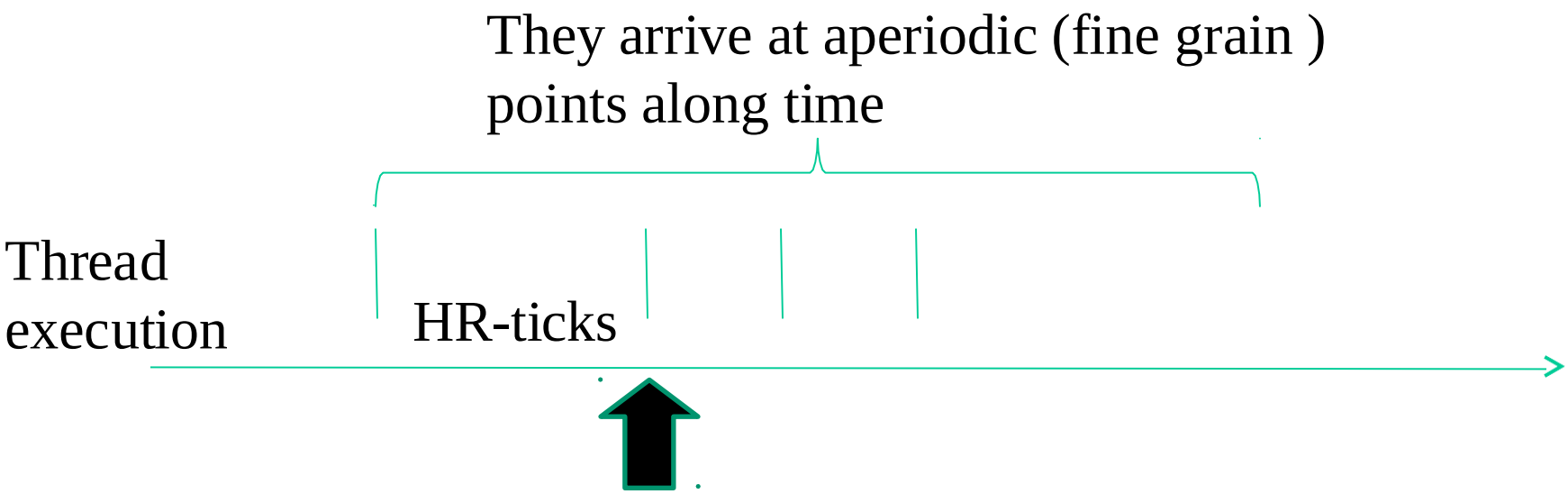
This runs any time-related action

SoftIRQ based newer versions - the top half - kernel 3 example

```
__visible void __irq_entry smp_apic_timer_interrupt(struct pt_regs *regs)
{
        struct pt_regs *old_regs = set_irq_regs(regs);

        /*
         * NOTE! We'd better ACK the irq immediately,
         * because timer handling can be slow.
         *
         * update_process_times() expects us to have done irq_enter().
         * Besides, if we don't timer interrupts ignore the global
         * interrupt lock, which is the WrongThing (tm) to do.
         */
        entering_ack_irq();
        local_apic_timer_interrupt();
        exiting_irq();

        set_irq_regs(old_regs);
}
```

1) just flag the current thread for reschedule (if needed)
2) Raise the flag of `TIMER_SOFTIRQ`

High Resolution (HR) Timers



We can still do minimal stuff here such as
1) raising the `HRTIMER_SOFTIRQ`
2) programming the next HR timer interrupt
based on a log of requests
3) Raise a preemption request

Do we ever see HR-timers in our user programs?

- What about `usleep()`?

1) The calling thread traps to kernel

2) The kernel puts a HR-timer request into the log (and possibly reprograms the HR-timer component)

3) The scheduler is called to pass control to someone else

4) Upon expiration of the HR-timer for this request along the execution of another thread, this will be possibly unscheduled (as soon as possible) to resume the sleeping one

The HR-timers kernel interface

```
ktime_t kt;
kt = ktime_set(long secs, long nanosecs)
```

```
void hrtimer_init( struct hrtimer *timer,
   clockid_t which_clock, enum hrtimer_mode mode)
```

Specify the clocking mechanism (CLOCK_MONOTONIC vs CLOCK_REALTIME)

Specify 1) function pointer and 2) data

Specify timing base (relative/absolute)

The function will fire one or more times depending on its return value (HRTIMER_RESTART/HRTIMER_NORESTART)

```
int hrtimer_start(struct hrtimer *timer, ktime_t time,
    enum hrtimer_mode mode)
```

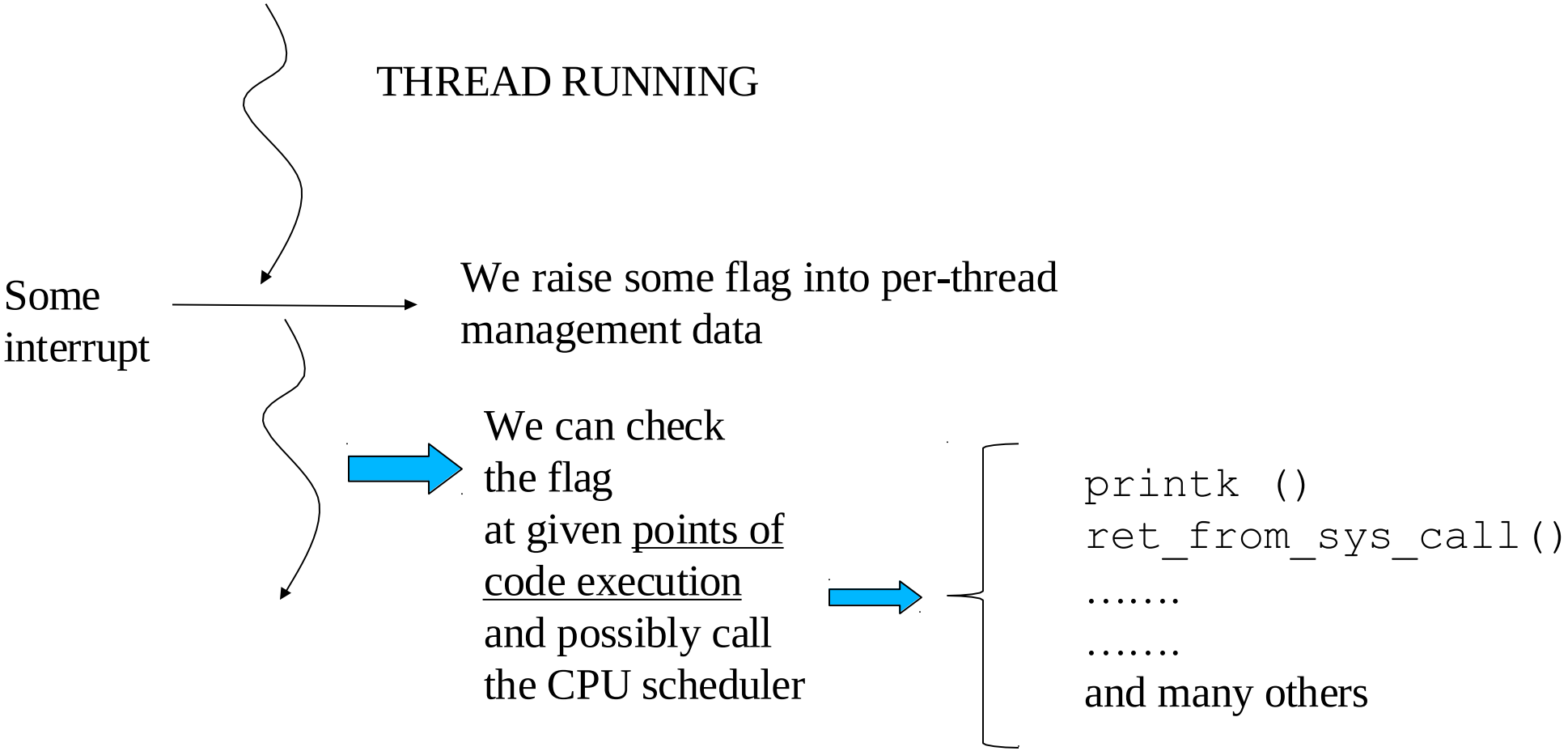
The HR-timers cancellation

Waits if the target function is already running

```
int hrtimer_cancel(struct hrtimer *timer);

int hrtimer_try_to_cancel(struct hrtimer *timer)
```

Does not wait if the target function is already running

What is a preemption request?



Can we save ourselves from preemptions?

- YES, we use per-thread preemption counters
- If the counter is not zero, then the preemption checking block of code will not lead to scheduler activation
- How do we exploit these counters transparently?
  - ✓ A set of specific API functions can be used
  - ✓ Lets' check with them

The API

```
preempt_enable() //decrement the preempt counter

preempt_disable() //increment the preempt counter

preempt_enable_no_resched() //decrement, but do not immediately
preempt

preempt_check_resched() //if needed, reschedule

preempt_count() //return the preempt counter
```

Preemption vs per-CPU variables

- Do you remember the `get/put_cpu_var()` API?
- They do a disable/enable of preemption upon entering/exiting, meaning that no other thread can use the same per-CPU variables in the meanwhile
- … and we are safe against functions that do the preemption check!!
- Clearly, <u>if the current threads explicitly calls a blocking service before "putting" a per CPU variable</u>, then the above property is no longer guaranteed

The role of TCBs (aka PCBs) in common operating systems

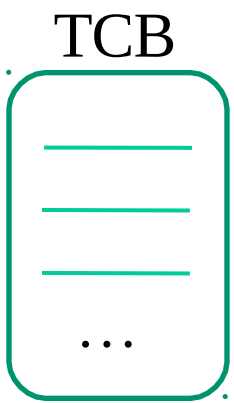
- A TCB is a data structure mostly keeping information related to

  ✓ Schedulability and execution flow control (scheduler specific information)

  ✓ Linkage with subsystems external to the scheduling one (via linkage to metadata)

  ✓ Multiple TCBs can link to the same external metadata (as for multiple threads within a same process)

An example



If and how the CPU scheduling logic should handle this thread

How the kernel should manage memory and its accesses by this thread (just to name, do you remember the mem-policy concept?)

How the kernel should manage VFS services on behalf of this thread

```
struct … {
        …
        …
}
```

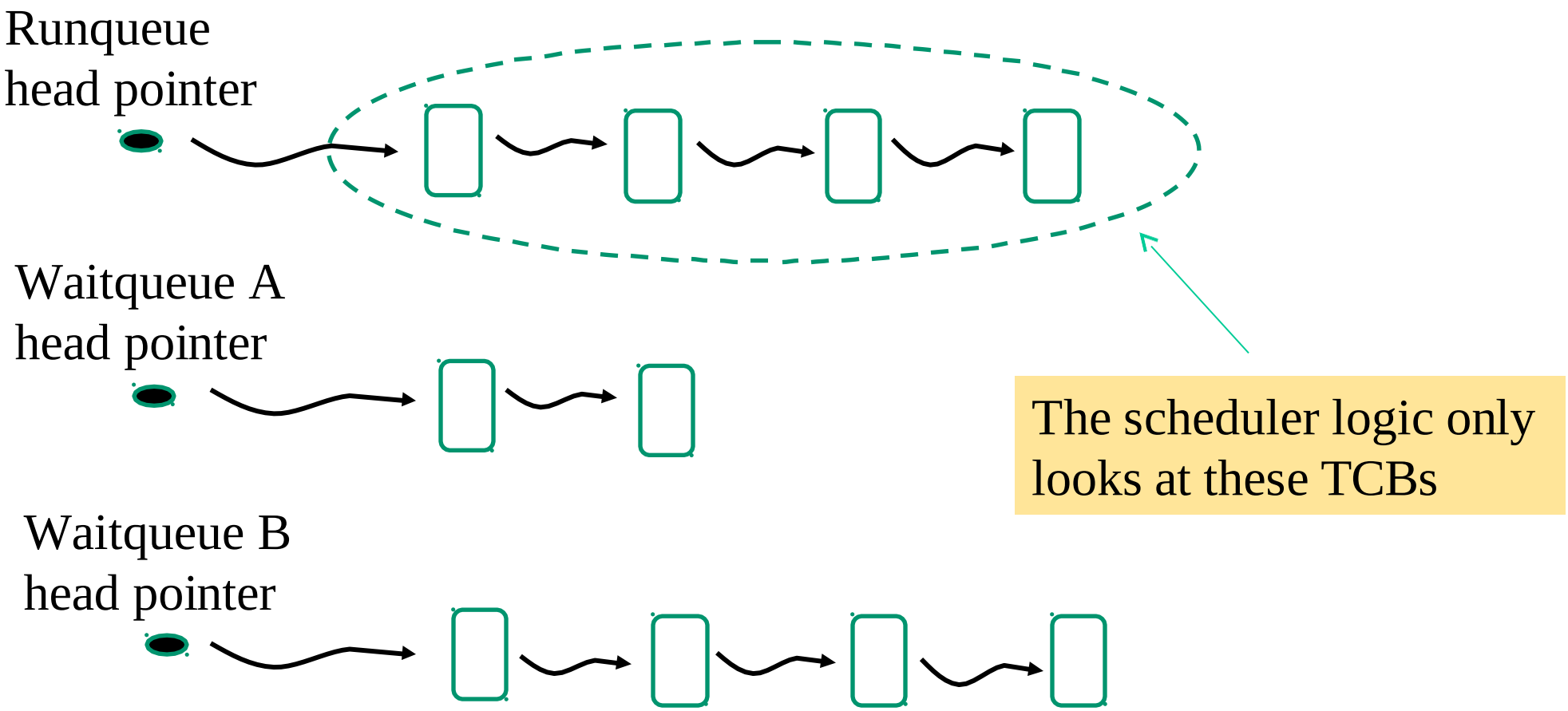
The scheduling part - CPU-dispatchability

- The TCB tells at any time whether the thread can be CPU-dispatched
- But what it the real meaning of “CPU-dispatchability”??
- Its meaning is that the scheduler logic (so the corresponding block of code) can decide to pick the CPU-snapshot kept by the TCB and install it on CPU
- CPU-dispatchability is not decided by the scheduler logic, rather by other entities (e.g. an interrupt handler)
- <u>So the scheduler logic is simply a selector of currently CPU-dispatchable threads</u>

The scheduling part - run/wait queues

- A thread is CPU-dispatchable only if its TCB is included into a specific data structure (generally, but not always, a list)
- This is typically refereed to as the **runqueue**
- The scheduler logic selects threads based on ``scans'' of the runqueue
- All the non CPU-dispatchable threads are kept on aside data structures (again lists) which are not looked at by the scheduling logic
- These are typically referred to as **waitqueues**

A scheme


Runqueue
head pointer
Waitqueue A
head pointer
Waitqueue B
head pointer
The scheduler logic only
looks at these TCBs

Scheduler logic vs blocking services

- Clearly the scheduler logic is run on a CPU-core within the context of some generic thread A
- When we end executing the logic the CPU-core can have switched to the context of another thread B
- Clearly, when thread A is running a blocking service in kernel mode it will synchronously invoke the scheduler logic, but its TCB is currently present on the runqueue
- How to exclude the TCB of thread A from the scheduler selection process?

Sleep/wait kernel services

- A blocking service typically relies on well structured **<u>kernel level sleep/wait services</u>**
- These services exploit TCB information to drive, in combination with the scheduler logic, the actual behavior of the service-invoking thread
- Possible outcomes of the invocation of these services:

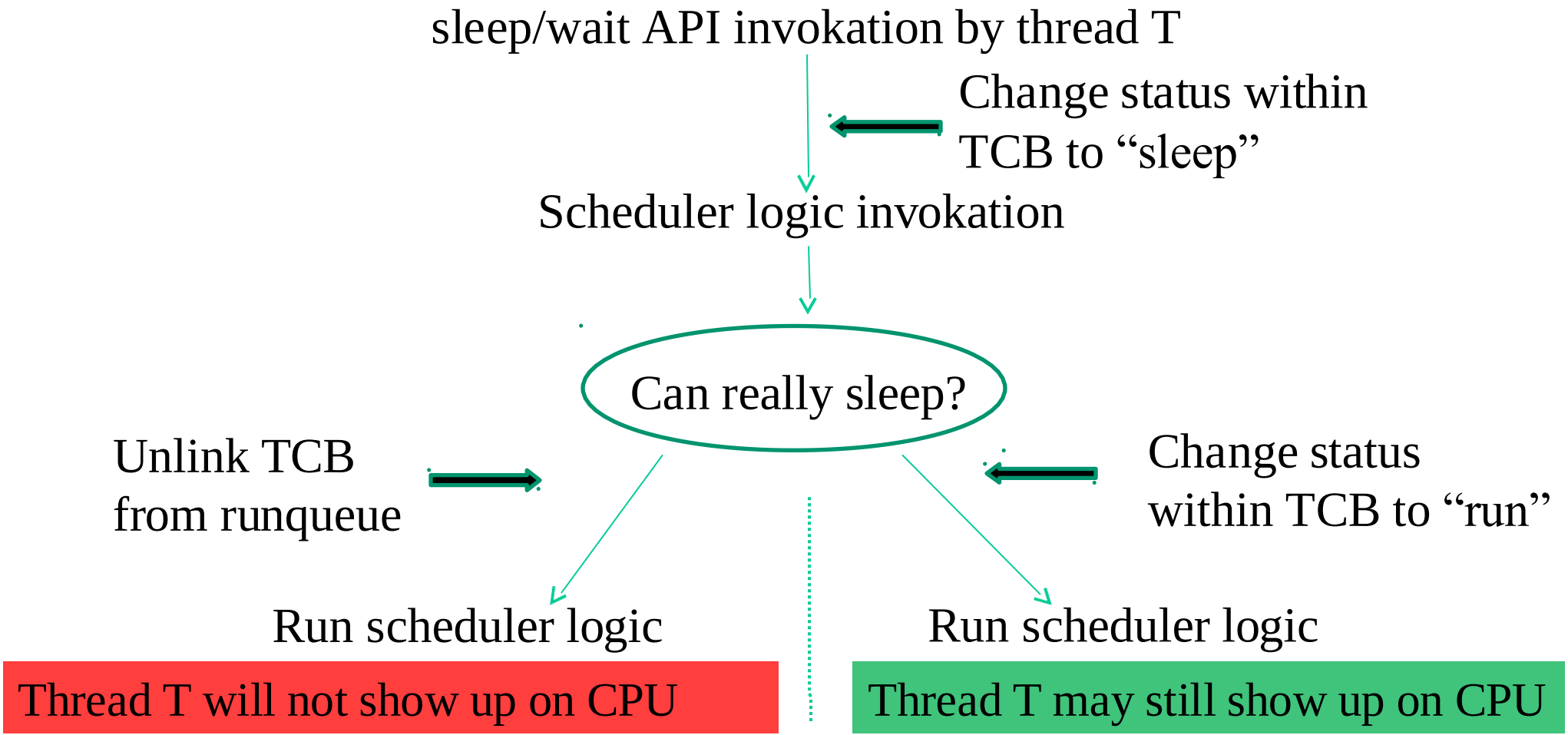
  ✓ The TCB of the invoking thread i<u>s removed from the runqueue by the scheduler logic before the actual selection of the next thread to run is performed</u>

  ✓ The TCB of the invoking thread <u>still stands on the runqueue</u> during the selection of the next thread to be run

Where does the TCB of a thread invoking a sleep/wait service stand?

- No way, it stands onto some waitqueue
- Well structuring of sleep/wait services is in fact based on an API where we need to pass the ID of some waitqueue in input
- Overall timeline of a sleep/wait service:
    1. Link the TCB of the invoking thread on some waitqueue
    2. Flag the thread as “sleep”
    3. Call the scheduler logic (will really sleep?)
    4. Unlink the TCB of the invocking thread from the wait waitque

The timeline



Additional features

- **Unlinkage from the waitqueue**
  - ✓ Done by the same thread that was linked upon being rescheduled
- **Relinkage to the runqueue**
  - ✓ Done by other threads when running whatever piece of kernel code such as
    - ➢ Synchronously invoked services (e.g. `sys_kill`)
    - ➢ Top/botton halves

Actual context switch

- It involves saving into the TCB the CPU context of the thread that is switched off the CPU
- It involves restoring from the TCB the CPU context of the CPU-dispatched thread
- One core point in changing the CPU context is related to the core kernel level ``private'' memory area each thread has
- This is the kernel level stack
- In most kernel implementations we say that <u>we switch the context when we install a value on the stack pointer</u>

Linux thread control blocks

- The structure of Linux process control blocks is defined in `include/linux/sched.h` as `struct task_struct`
- The main fields (ref 2.6 kernel) are
  - **`volatile long state`**
  - **`struct mm_struct *mm`**
  - **`pid_t pid`**
  - **`pid_t pgrp`**
  - `struct fs_struct *fs`
  - `struct files_struct *files`
  - `struct signal_struct *sig`
  - **`volatile long need_resched`**
  - `struct thread_struct thread /* CPU-specific state of this task – TSS */`
  - **`long counter`**
  - **`long nice`**
  - **`unsigned long policy`** `/*CPU scheduling info*/`

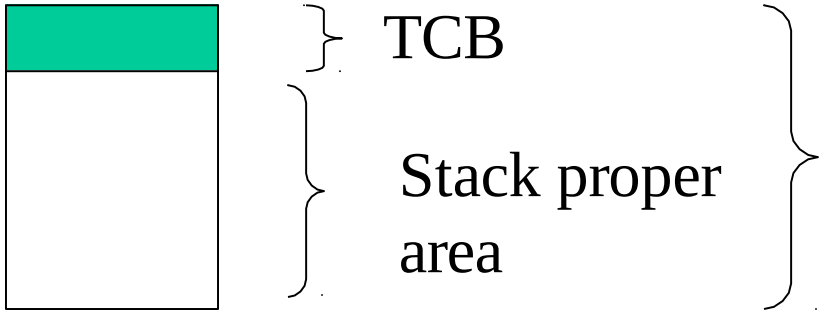
synchronous and asynchronous modifications

More modern kernel versions

- A few info is compacted into bitmasks
  - ✓ e.g. `need_resched` has become the `TIF_NEED_RESCHED` bit into a bit-mask
- The compacted info can be easily accessed via specific macros/APIs
- More field have been added to reflect new capabilities, e.g., in the Posix specification or Linux internals
- The main fields are still there, such as
  - `state`
  - `pid`
  - `tgid (the group ID)`
  - …

TCB allocation - the case before kernel 2.6

- TCBs are allocated dynamically, whenever requested
- The memory area for the TCB is reserved within the top portion of the kernel level stack of the associated process
- This occurs also for the IDLE PROCESS, hence the kernel stack for this process has base at the address `&init_task+8192`, where `init_task` is the TCB of the IDLE PROCESS



<u>THREAD_SIZE (typically 8KB located onto 2 buddy frames)</u>

Implications from the encapsulation of TCB into the stack-area

- A single memory allocation request is enough for making per-thread core memory areas available (see `_get_free_pages()`)
- However, TCB size and stack size need to be scaled up in a correlated manner
- The latter is a limitation when considering that buddy allocation entails buffers with sizes that are powers of 2 times the size of one page
- The growth of the TCB size may lead to
  - ✓ Buffer overflow risks, if the stack size is not rescaled
  - ✓ Memory fragmentation, if the stack size is rescaled

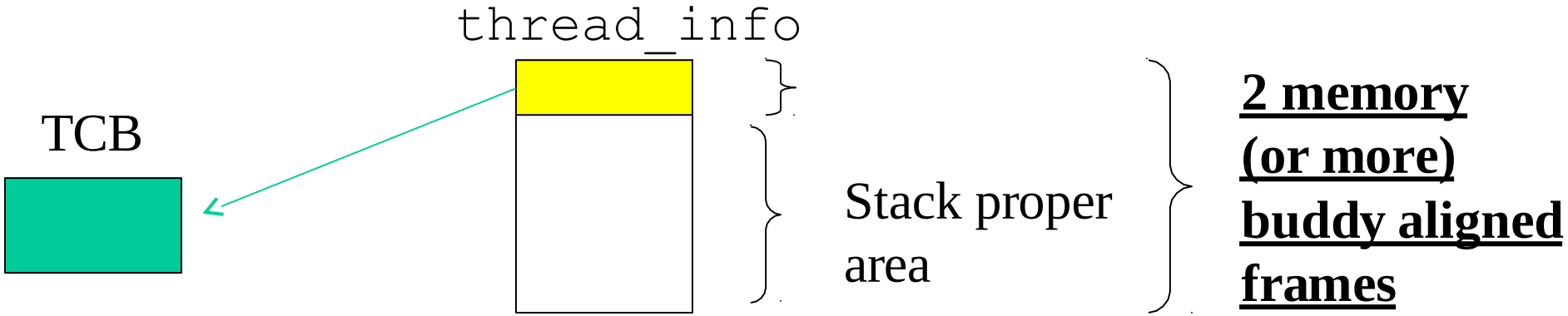
Actual declaration of the kernel level stack data structure

Kernel 2.4.37 example

```
union task_union {
 struct task_struct task;
 unsigned long stack[INIT_TASK_SIZE/sizeof(long)];
};
```

TCB allocation - since kernel 2.6 up to 4.8

- The memory area for the TCB is reserved outside the top portion of the kernel level stack of the associated process
- At the top portion we find a so called `thread_info` data structure
- This is used as an indirection data structure for getting the memory position of the actual TCB
- This allows for improved memory usage with large TCBs



Actual declaration of the kernel level thread_info data structure

Kernel 3.19 example

```
struct thread_info {
 struct task_struct  *task;        /* main task structure */
 struct exec_domain  *exec_domain;   /* execution domain */
 __u32               flags;        /* low level flags */
 __u32               status;  /* thread synchronous flags */
 __u32               cpu;     /* current CPU */
 int                 saved_preempt_count;
 mm_segment_t        addr_limit;
 struct restart_block restart_block;
 void __user         *sysenter_return;
 unsigned int         sig_on_uaccess_error:1;
 unsigned int         uaccess_err:1;  /* uaccess failed */
};
```

Kernel 4 thread size on x86-64 (kernel 5 is similar)

```
#define THREAD_SIZE_ORDER 2
#define THREAD_SIZE (PAGE_SIZE << THREAD_SIZE_ORDER)
```

Here we get 16KB

Defined in **arch/x86/include/asm/page_64_types.h for x86-64**

The current MACRO

- The macro `current` is used to return the memory address of the TCB of the currently running process/thread (namely the pointer to the corresponding `struct task_struct`)

- This macro performs computation based on the value of the stack pointer (up to kernel 4.8), by exploiting that the stack is aligned to the couple (or higher order) of pages/frames in memory

- This also means that a change of the kernel stack implies a change in the outcome from this macro (and hence in the address of the TCB of the running thread)

Actual computation by current

Old style

Masking of the stack pointer value so to discard the less significant bits that are used to displace into the stack

New style

Masking of the stack pointer value so to discard the less significant bits that are used to displace into the stack

Indirection to the `task` filed of `thread_info`

… the very new style of current

- It is a pointer located onto <u>per-CPU memory</u>
- The pointer is updated when a CPU-reschedule is carried out
- …. finally no longer buddy blocks aligned stacks!!!

```
struct task_struct;
DECLARE_PER_CPU(struct task_struct *,current_task);

Static __always_inline struct task_struct
*get_current (void) {
    return this_cpu_read_stable (current_task);
}
#define current get_current()
```

More flexibility and isolation - virtually mapped stacks

- Typically we only need logical memory contiguousness for a stack area
- On the other hand stack overflow is a serious problem for kernel corruption, especially under attack scenarios
- One approach is to rely on `vmalloc()` for creating a stack allocator
- The advantage is that surrounding pages to the stack area can be set as unmapped
- How do we cope with computation of the address of the TCB under arbitrary positioning of the kernel stack has been already seen thanks to per-CPU-memory (from kernel 4.9)

A look at the run queue - 2.4 style

- In `kernel/sched.c` we find the following initialization of an array of pointers to `task_struct`

```
struct task_struct * init_tasks[NR_CPUS] = {&init_task,}
```

- Starting from the TCB of the IDLE PROCESS we can find a list of PCBs associated with ready-to-run processes/threads
- The addresses of the first and the last TCBs within the list are also kept via the static variable `runqueue_head` of type `struct list_head{struct list_head *prev,*next;}`
- The TCB list gets scanned by the `schedule()` function whenever we need to determine the next process/thread to be dispatched

Wait queues - 2.4 style

- TCBs can be arranged into lists called wait-queues
- TCBs currently kept within any wait-queue are not scanned by the scheduler module
- We can declare a wait-queue by relying on the macro `DECLARE_WAIT_QUEUE_HEAD(queue)` which is defined in `include/linux/wait.h`
- The following main functions defined in `kernel/sched.c` allow queuing and de-queuing operations into/from wait queues
  - ➢ `void interruptible_sleep_on(wait_queue_head_t *q)`
    The TCB is no more scanned by the scheduler until it is dequeued or a signal kills the process/thread
  - ➢ `void sleep_on(wait_queue_head_t *q)`
    Like the above semantic, but signals are don’t care events

- `void interruptible_sleep_on_timeout(wait_queue_head_t *q, long timeout)`

  Dequeuing will occur by timeout or by signaling

- `void sleep_on_timeout(wait_queue_head_t *q, long timeout)`

  Dequeuing will only occur by timeout

Non selective

- `void wake_up(wait_queue_head_t *q)`

  Reinstalls onto the ready-to-run queue all the TCBs currently kept by the wait queue `q`

- `void wake_up_interruptible(wait_queue_head_t *q)`

  Reinstalls onto the ready-to-run queue the TCBs currently kept by the wait queue `q`, which were queued as "interruptible"

(too) Selective

- `wake_up_process(struct task_struct * p)`

  Reinstalls onto the ready-to-run queue the process whose PCB s pointed by `p`

Thread states

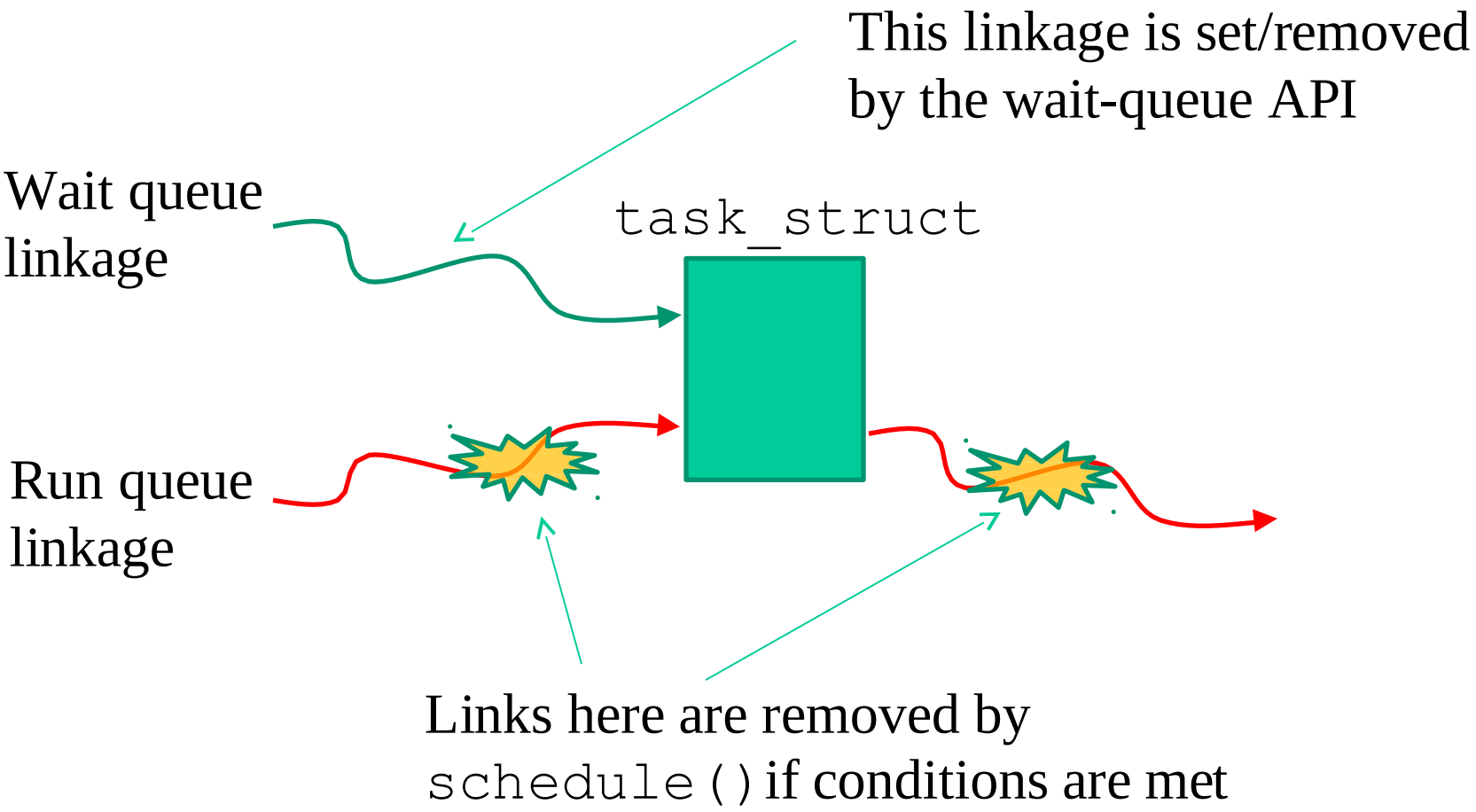
- The `state` field within the TCB keeps track of the current state of the process/thread
- The most relevant values are defined as follows in `include/linux/sched.h`
  - `#define TASK_RUNNING 0`
  - `#define TASK_INTERRUPTIBLE 1`
  - `#define TASK_UNINTERRUPTIBLE 2`
  - `#define TASK_ZOMBIE 4`
- All the TCBs recorded within the run-queue keep the value `TASK_RUNNING`
- The two values `TASK_INTERRUPTIBLE` and `TASK_UNINTERRUPTIBLE` discriminate the wakeup conditions from any wait-queue

Wait vs run queues

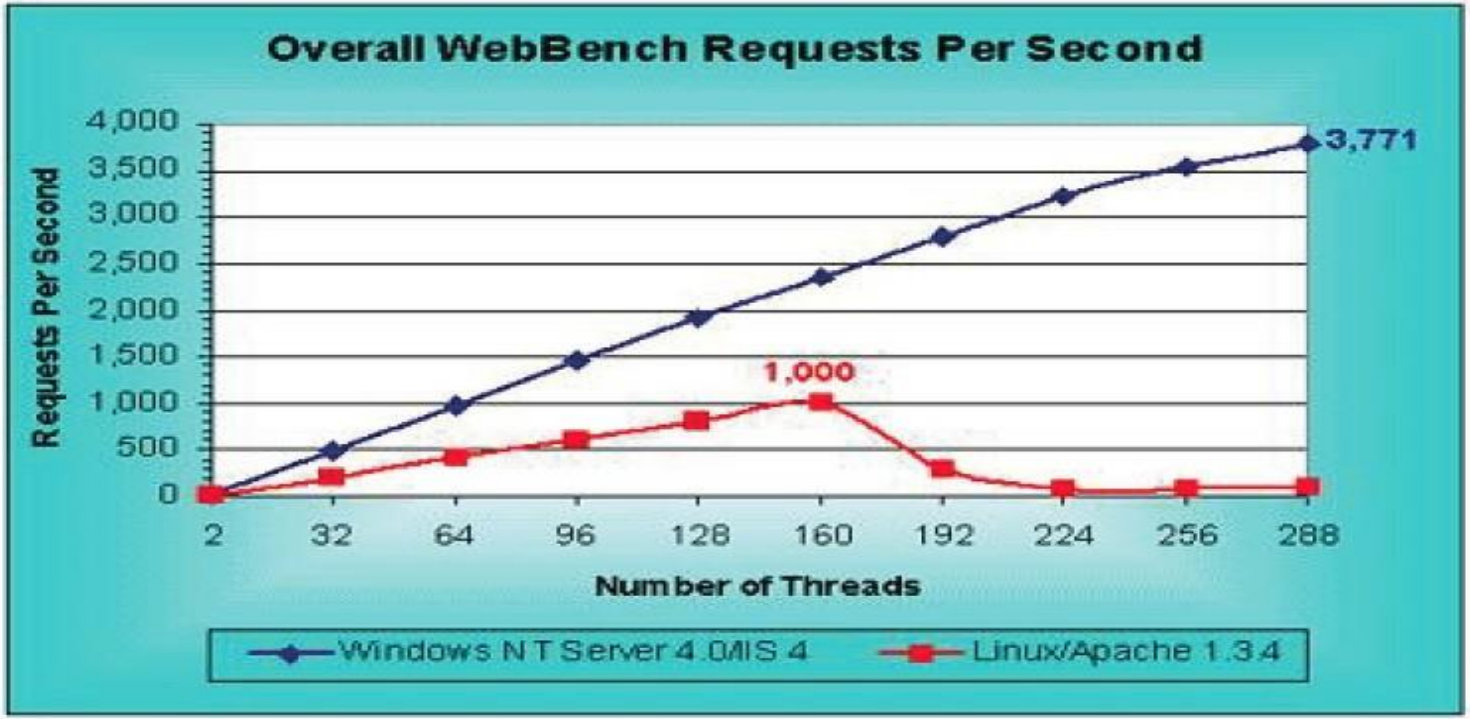
- wait queues APIs also manage the TCB unlinking from the wait queue upon returning from the schedule operation

```
#define SLEEP_ON_HEAD                    \
  wq_write_lock_irqsave(&q->lock,flags);       \
  __add_wait_queue(q, &wait);            \
  wq_write_unlock(&q->lock);

#define SLEEP_ON_TAIL                    \
  wq_write_lock_irq(&q->lock);               \
  __remove_wait_queue(q, &wait);             \
  wq_write_unlock_irqrestore(&q->lock,flags);

void interruptible_sleep_on(wait_queue_head_t *q){
  SLEEP_ON_VAR
  current->state = TASK_INTERRUPTIBLE;
  SLEEP_ON_HEAD
  schedule();
  SLEEP_ON_TAIL
}
```

TCB linkage dynamics



Thundering herd effect



Taken from 1999 Mindcraft study on Web and File Server Comparison

The new style - wait event queues

- They allow to drive thread awake via conditions
- The conditions for a same queue can be different for different threads
- This allows for selective awakes depending on what condition is actually fired
- The scheme is based on polling the conditions upon awake, and on consequent re-sleep

Conditional waits – one example

Name

wait_event_interruptible — sleep until a condition gets true

Synopsis

```
wait_event_interruptible (wq,
                          condition);
```

Arguments

wq

the waitqueue to wait on

condition

a C expression for the event to wait for

Description

The process is put to sleep (TASK_INTERRUPTIBLE) until the *condition* evaluates to true or a signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.

wake_up has to be called after changing any variable that could change the result of the wait condition.

The function will return -ERESTARTSYS if it was interrupted by a signal and 0 if *condition* evaluated to true.

Wider (although non-exhaustive) API

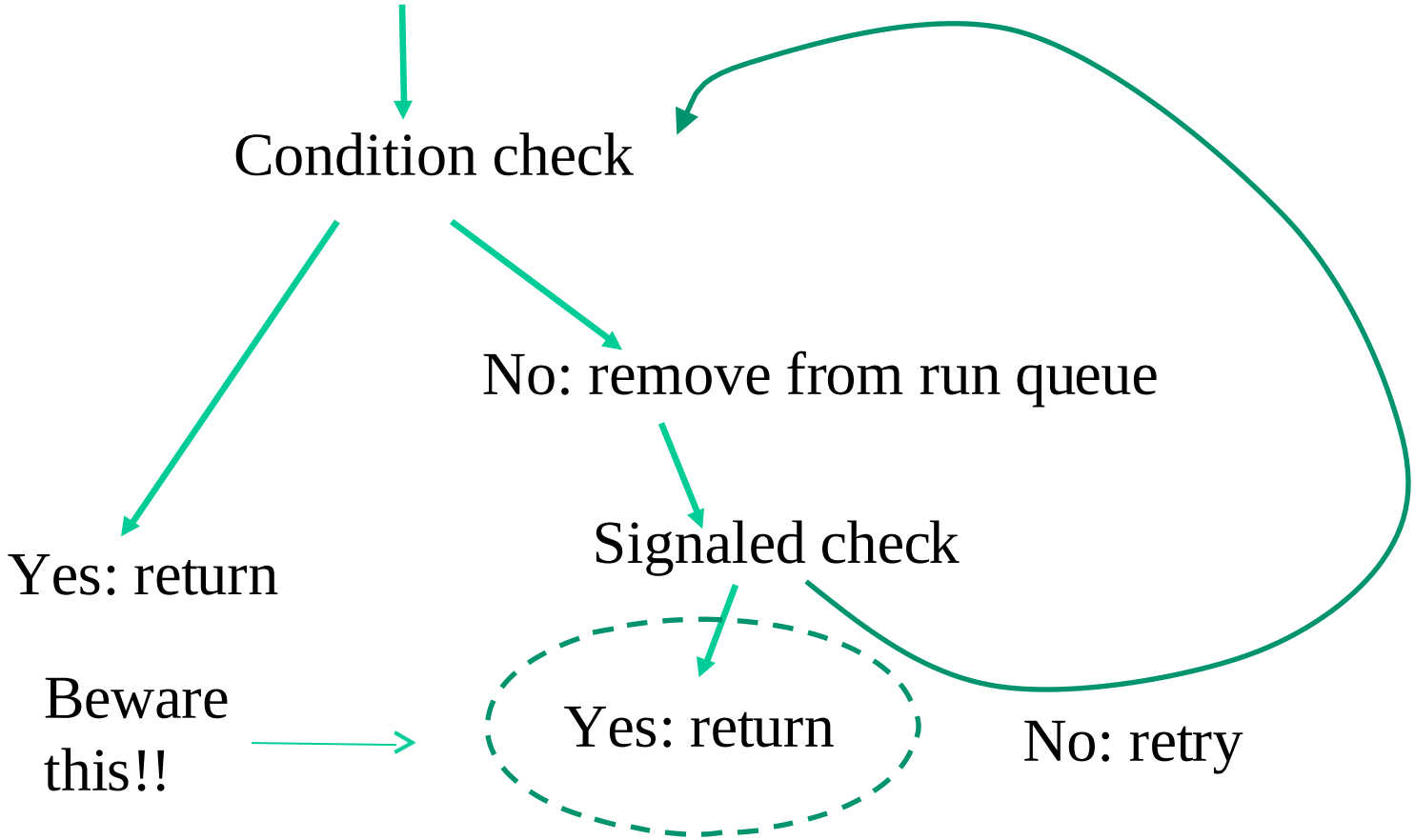
```
wait_event( wq, condition )

wait_event_timeout( wq, condition, timeout )

wait_event_freezable( wq, condition )

wait_event_command( wq, condition, pre-command,
       post-command)

wait_on_bit( unsigned long * word, int bit,
       unsigned mode)

wait_on_bit_timeout( unsigned long * word, int bit,
       unsigned mode, unsigned long timeout)

wake_up_bit( void* word, int bit)
```

Macro based expansion

```
#define ___wait_event(wq_head, condition, state, exclusive, ret, cmd)                    \
 ({                                                                                        \
        __label____out;                                                                    \
        struct wait_queue_entry __wq_entry;                                                \
        long __ret = ret; /* explicit shadow */                                            \
        init_wait_entry(&__wq_entry, exclusive ? WQ_FLAG_EXCLUSIVE : 0);          \
        for (;;) {                                                                         \
                long __int = prepare_to_wait_event(&wq_head, &__wq_entry, state);  \
                if (condition)                                                             \
                        break;                                                             \
                if (___wait_is_interruptible(state) && __int) {                    \
                        __ret = __int; \ goto __out;                                       \
                }                                                                          \
                cmd;                                                                       \
        }                                                                                  \
        finish_wait(&wq_head, &__wq_entry); \ __out: __ret;                        \
})
```

Cycle based approach

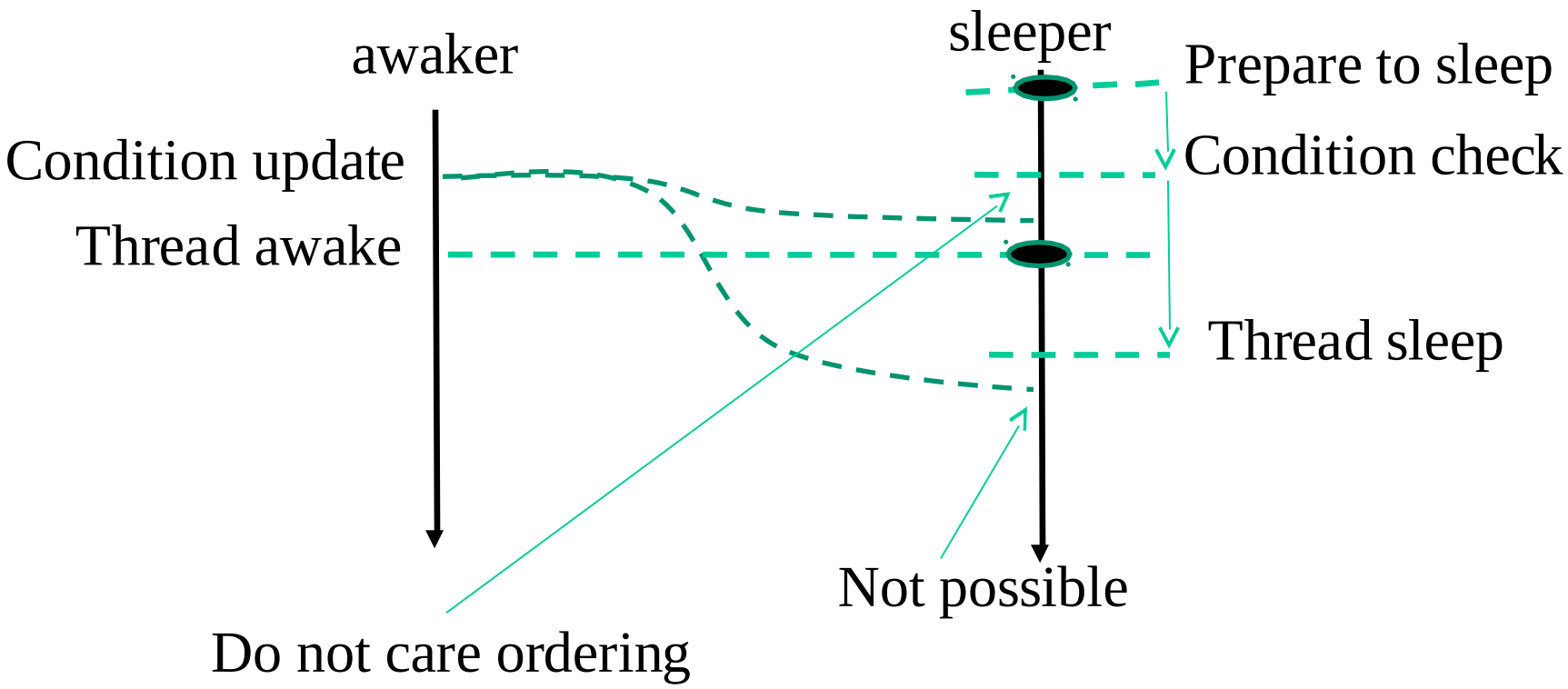
The scheme for interruptible waits



Linearizability

- The actual management of condition checks <u>prevents any possibility of false negatives in scenarios with concurrent threads</u>
- This is still because removal from the run queue occurs within the `schedule()` function and the removal leads to spinlock the TCB
- However the awake API leads to spinlock the TCB too for updating the thread status and (possibly) relinking it to the run queue
- This leads to memory synchronization (TSO bypass avoidance)
- The locked actions represent the linearization point of the operations
- An awake updates the thread state after the condition has been set
- A wait checks the condition before checking the thread state via `schedule()`

A scheme


awaker
sleeper
Prepare to sleep
Condition update
Condition check
Thread awake
Thread sleep
Not possible
Do not care ordering

The mm field in the TCB

- The `mm` of the TCB points to a memory area structured as `mm_struct`
- This area keeps information used for memory management purposes for the specific process, such as
  - ➢ Virtual address of the page table (`pgd` field) – top 4KB kernel, bottom 4KB user in case of PTI
  - ➢ A pointer to a list of records structured as `vm_area_struct` (`mmap` field)
- Each record keeps track of information related to a specific virtual memory area (user level) which is valid for the process

vm_area_struct

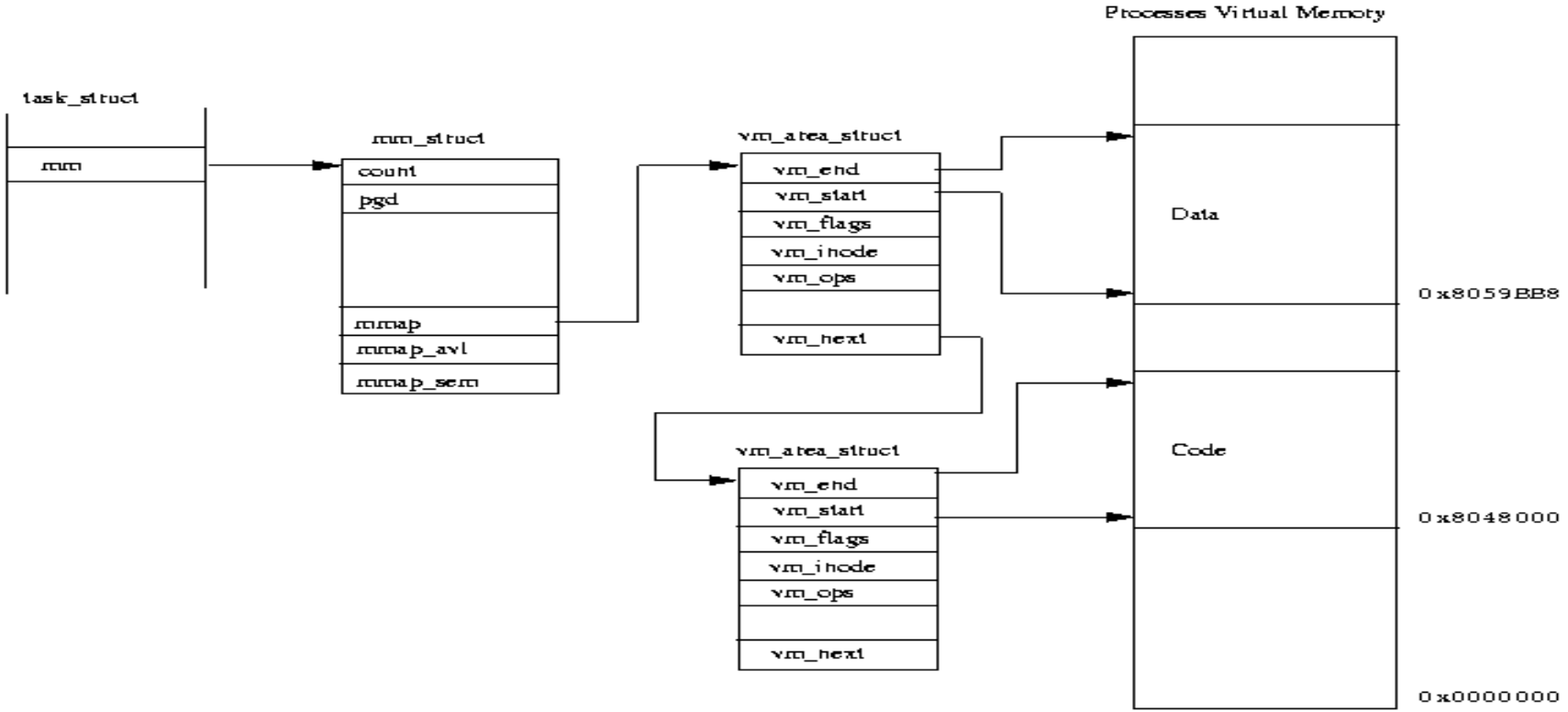
```
struct vm_area_struct {
  struct mm_struct * vm_mm;/* The address space we belong to. */
  unsigned long vm_start; /* Our start address within vm_mm. */
  unsigned long vm_end;   /* The first byte after our end address
                          within vm_mm. */
  struct vm_area_struct *vm_next;
  pgprot_t vm_page_prot;  /* Access permissions of this VMA. */

  ……................
  /* Function pointers to deal with this struct. */
  struct vm_operations_struct * vm_ops;
  ……............
};
```

The `vm_ops` field points to a structure used to define the treatment of faults occurring within that virtual memory area → this is specified via the field `nopage` or `fault`

As and example this pointer identifies a function signed as

```
struct page * (*nopage)(struct vm_area_struct * area, unsigned
  long address, int unused)
```

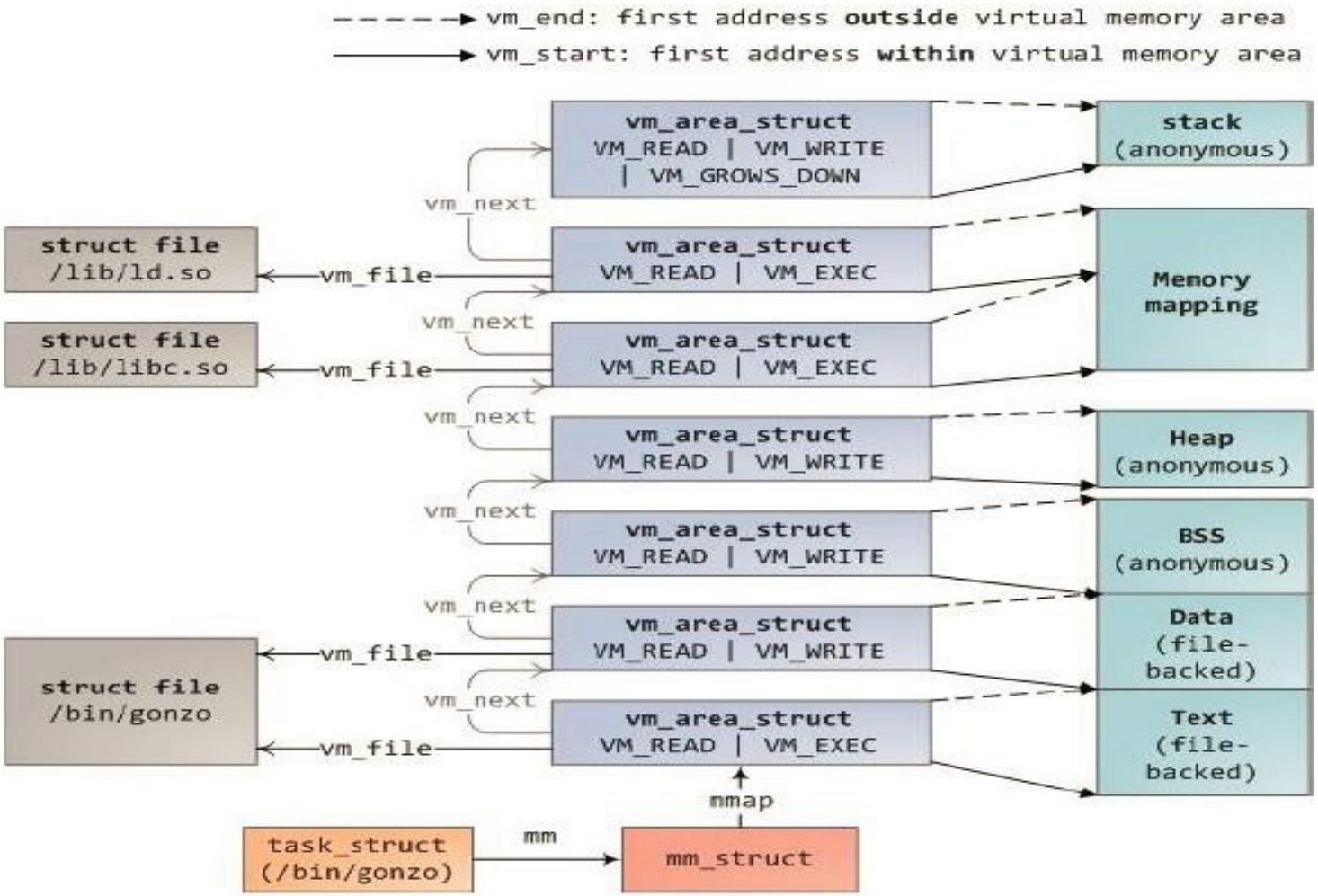
A scheme



The executable format for Linux is ELF

This format specifies, for each section (text, data) the positioning within the virtual memory layout, and the access permission

An example


vm_end: first address outside virtual memory area
vm_start: first address within virtual memory area
vm_area_struct
VM_READ | VM_WRITE | VM_GROWS_DOWN
stack (anonymous)
vm_next
struct file /lib/ld.so
vm_file
vm_area_struct
VM_READ | VM_EXEC
Memory mapping
struct file /lib/libc.so
vm_area_struct
VM_READ | VM_EXEC
vm_area_struct
VM_READ | VM_WRITE
Heap (anonymous)
vm_area_struct
VM_READ | VM_WRITE
BSS (anonymous)
vm_area_struct
VM_READ | VM_WRITE
Data (file-backed)
struct file /bin/gonzo
vm_area_struct
VM_READ | VM_EXEC
Text (file-backed)
nmap
task_struct (/bin/gonzo)
mm
mm_struct

Red-black tree re-organization

- The vmarea list has been replaced in recent kernels (e.g. Kernel 6) by a red-black tree, for improving the complexity of additions/deletions
- The Linux kernel offers anyhow some API for traversing the vmarea entries
- Here is an example of what you can exploit:

```
void traverse_vm_areas(struct mm_struct *mm) {
    struct rb_node *node;
   struct vm_area_struct *vma; // Start with the leftmost node in the red-black tree

   for (node = rb_first(&mm->mm_rb); node; node = rb_next(node)) {
          vma = rb_entry(node, struct vm_area_struct, vm_rb);
         pr_info("VMA: start=0x%lx, end=0x%lx\n", vma->vm_start, vma->vm_end);
    }
}
```

- For searching an entry with a given area you can use:

```
struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr);
```

Threads identification

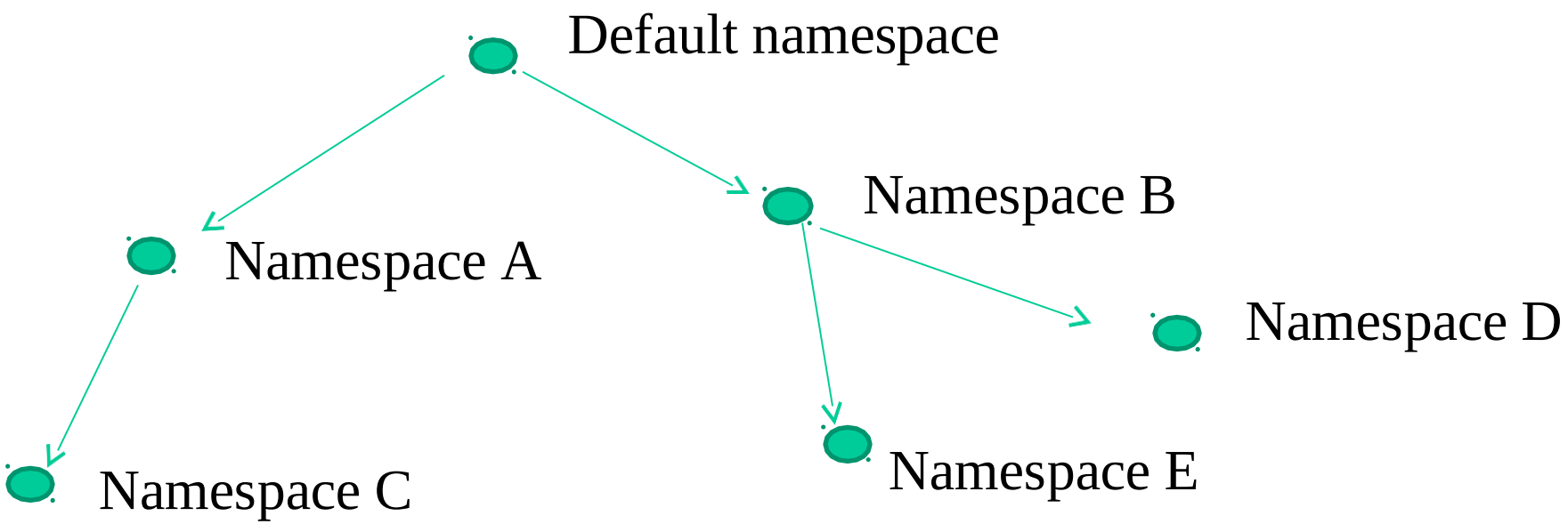
- In modern implementations of OS kernels we can also virtualize PIDs
- So each thread may have more than one PID
  - ✓ a real one (say `current->pid`)
  - ✓ a virtual one
- This concept is linked to the notion of **namespaces**
- Depending on the namespace we are working with then one PID value (not the other) is the reference for a set of common operations
- As an example, if we call the `ppid()` system call, then the ID that is returned is the PID of the parent thread referring to the current namespace of the invoking one

PID namespace scheme

- The baseline kernel namespace is by default used to set the value `current->pid`
- When a new thread is created, then we can specify to move to another PID namespace, which becomes a child level PID namespace with respect to the current one
- A maximum of 32 levels of PID namespaces can be used in Linux, based on the define

```
#define MAX_PID_NS_LEVEL 32
```

A representation

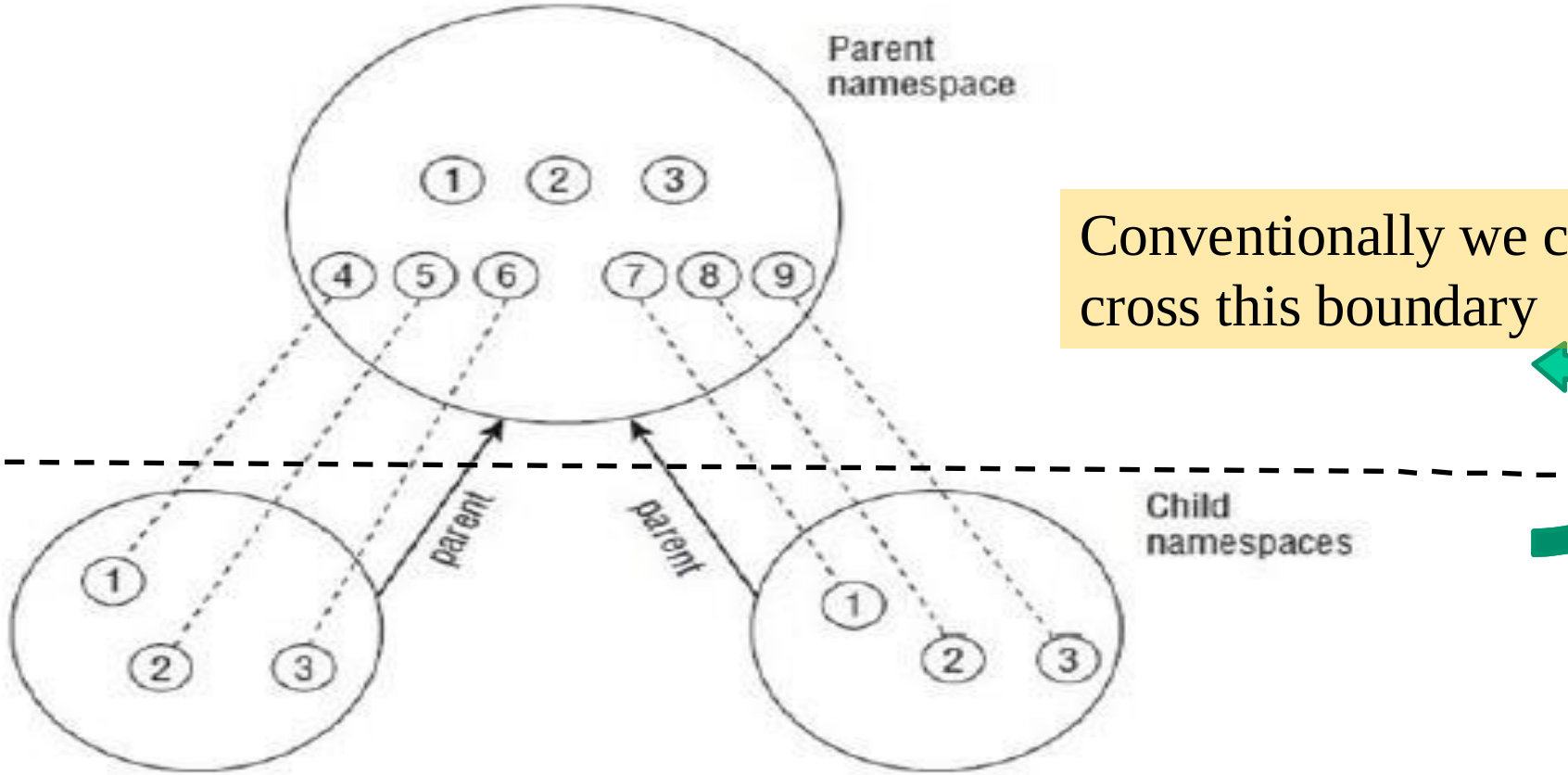


The thread whose creation leads to create a new namespace has virtual PID set to 1 in that namespace, and its ancestor is PID zero

Namespace visibility

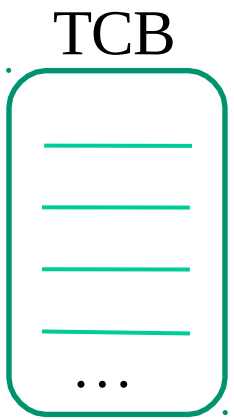
- By relying on common OS kernel services, a thread that leaves in a given namespace has no visibility of ancestor namespaces
- So it cannot “exploit” the existence of ancestor threads
- As an example, we cannot kill threads living into ancestral namespaces
- A namespace is therefore a sort of container (a concept you should be already familiar with)
- **NOTE** → all the above is true in an agreed upon environmental settings, it can change if we modify kernel operations

A scheme



Conventionally we cannot cross this boundary

The implementation



struct nsproxy *nsproxy;

The PID namespace
(and other namespaces not related to PIDs)

The PID value in the reference PID namespace

```
struct … {
    …
    …
}
```

Some more details

```
struct pid_namespace {
    …
    struct pid_namespace *parent;
    …
}
```

```
struct nsproxy {
    …
    struct pid_namespace *pid_ns_for_children;
    …
}
```

```
struct pid_namespace {
    …
    unsigned int pid_allocated;
    …
    struct pid_namespace *parent;
    …
}
```

PID to task_struct mapping

- A lot of kernel services work by using the address of the TCB of a thread (see awake from sleep/wait queues)
- So we need a mapping between PIDs and TCB addresses
- The mapping is based on linked data, such as TCB linkage or namespaces linkage
- Linux offers services for transparently traversing these linkages

Accessing TCBs in the default namespace (the only one existing originally)

- TCBs are linked in various lists with hash access

- They were supported via the below fields within the TCB structure

```
/* PID hash table linkage. */
struct task_struct *pidhash_next;
struct task_struct *pidhash_pprev;
```

- There existed a hashing structure was defined as below

```
#define PIDHASH_SZ (4096 >> 2)
extern struct task_struct *pidhash[PIDHASH_SZ];
#define pid_hashfn(x) ((((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1))
```

- We also have the following function (of `static` type), still defined in `include/linux/sched.h` which allows retrieving the memory address of the PCB by passing the process/thread pid as input

```
static inline struct task_struct *find_task_by_pid(int pid) {
  struct task_struct *p,
     **htable = &pidhash[pid_hashfn(pid)];

  for(p = *htable; p && p->pid != pid;
        p = p->pidhash_next) ;
  return p;
}
```

Querying across namespaces

- The newer kernel versions (e.g. >= 2.6) support is

```
struct task_struct *find_task_by_vpid(pid_t vpid)
```

- This is based on the notion of virtual pid (so the one in the current namespace we are working with)
- We access a hashing system that more or less directly links vPIDs to TCBs
- The vPID of thread by default coincides with its PID if no namespace different from the default one is setup

vPIDs hashing

```
struct pid {
        atomic_t count;
        unsigned int level;
        /* lists of tasks that use this pid */
        struct hlist_head tasks[PIDTYPE_MAX];
        struct rcu_head rcu;
        struct upid numbers[1];
};
```

This keeps the number and the reference to the namespace

When accessing the target PID records we can match with the namespace of the caller

We can query for individuals or groups (see TGID)

Enabling lower level namespace visibility

```
static bool access_pidfd_pidns(struct pid *pid) {
	struct pid_namespace *active = task_active_pid_ns(current);
	struct pid_namespace *p = ns_of_pid(pid);
	for (;;) {
		if (!p)
			return false;
		if (p == active)
			break;
		p = p->parent;
	}
	return true;
}
```

This is called, e.g., when a kill from the current thread is issued towards another threads

Managing virtual PIDs in Linux modules

```
struct task_struct *pid_task(struct pid *pid, enum
 pid_type);
```

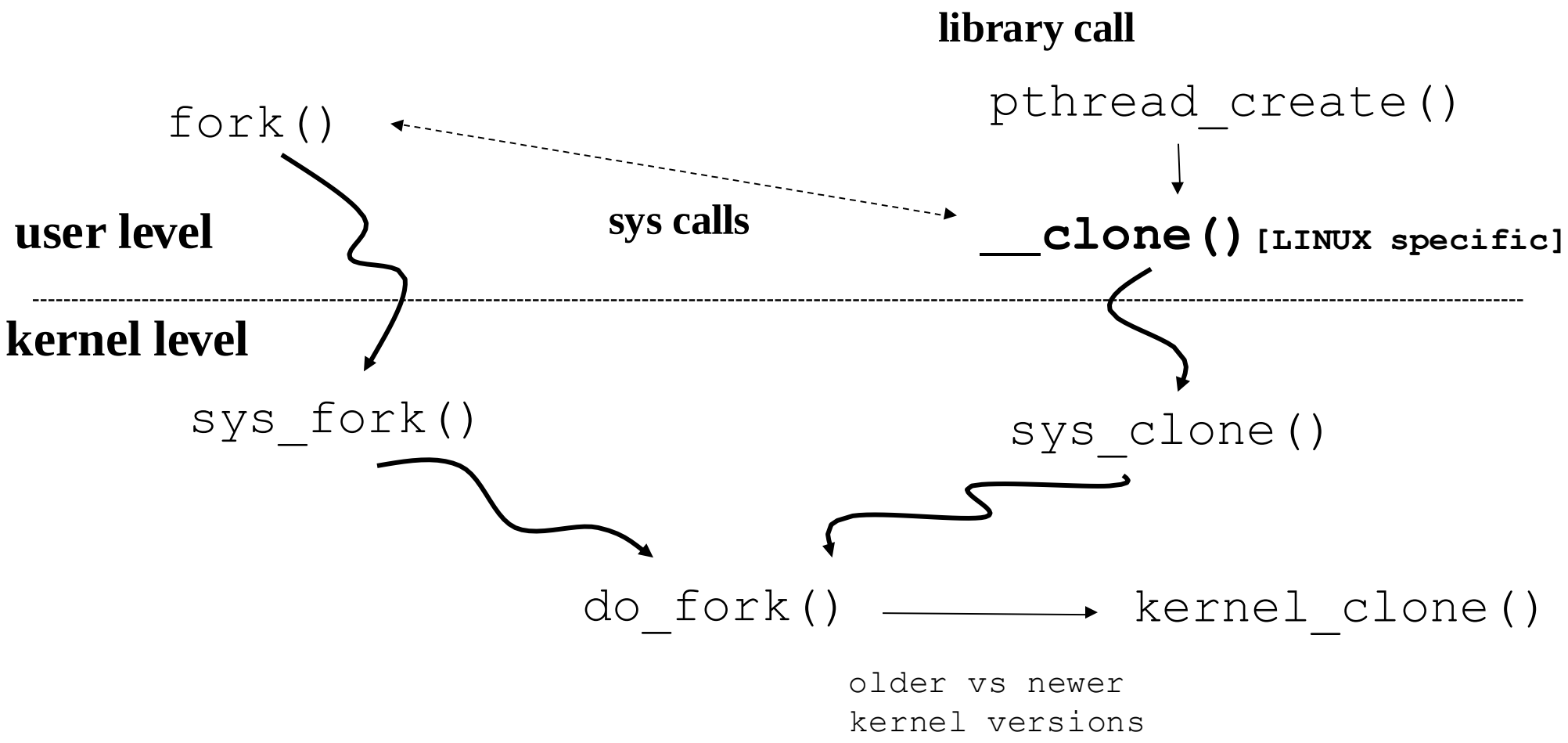
`PIDTYPE_PID` or other

`find_vpid(pid)`

```
pid_task(find_vpid(pid), PIDTYPE_PID);
```

Querying the TCB address by the default PID

Process and thread creation



The glibc interface

Return value mapped to thread exit code

Parameters can vary in number and order

NAME top

clone, __clone2 - create a child process

SYNOPSIS top

```
/* Prototype for the glibc wrapper function */

#define _GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
          /* pid_t *ptid, void *newtls, pid_t *ctid */ );

/* For the prototype of the raw system call, see NOTES */
```

DESCRIPTION top

clone() creates a new process, in a manner similar to fork(2).

Architecture specific interfaces

Newer `pthreadXX()` services

The raw system call interface on x86-64 and some other architectures (including sh, tile, and alpha) is:

```
long clone(unsigned long flags, void *child_stack,
           int *ptid, int *ctid,
           unsigned long newtls);
```

On x86-32, and several other common architectures (including score, ARM, ARM 64, PA-RISC, arc, Power PC, xtensa, and MIPS), the order of the last two arguments is reversed:

```
long clone(unsigned long flags, void *child_stack,
           int *ptid, unsigned long newtls,
           int *ctid);
```

On the cris and s390 architectures, the order of the first two arguments is reversed:

```
long clone(void *child_stack, unsigned long flags,
           int *ptid, int *ctid,
           unsigned long newtls);
```

The flags (not exhaustive)

CLONE_VM	VM shared between processes
CLONE_FS	fs info shared between processes
CLONE_FILES	open files shared between processes
CLONE_PARENT	we want to have the same parent as the cloner
CLONE_NEWPID	create the process/tread in a new PID namespace
CLONE_SETTLS	the TLS (Thread Local Storage) descriptor is set to newtls
CLONE_THREAD	the child is placed in the same thread group as the calling process

do_fork/kernel_clone overview

- Allocate a TCB
- Allocate a stack area
- Get the proper PID (real/virtual)
- Link the parent memory map?
- Link the parent FS view?
- Link the parent files view?
- **….. possibly share ticks with parent!!!**

Synchronization abstractions

```
DECLARE_MUTEX(name);
/* declares struct semaphore <name> ... */

void sema_init(struct semaphore *sem, int val);
/* alternative to DECLARE_... */
void down(struct semaphore *sem); /* may sleep */

int down_interruptible(struct semaphore *sem);
/* may sleep; returns -EINTR on interrupt */

int down_trylock(struct semaphone *sem);
/* returns 0 if succeeded; will no sleep */

void up(struct semaphore *sem);
```

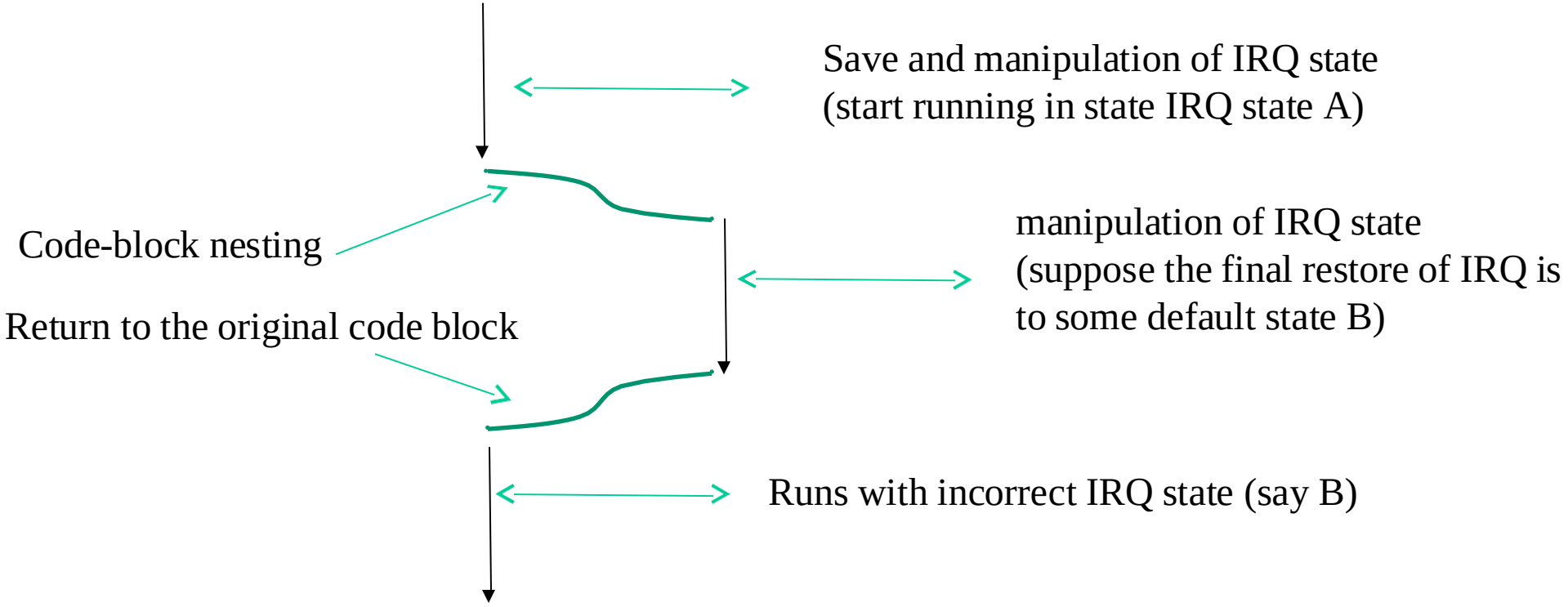
Spinlock API

```
#include <linux/spinlock.h>

spinlock_t my_lock = SPINLOCK_UNLOCKED;
spin_lock_init(spinlock_t *lock);
spin_lock(spinlock_t *lock);
spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
spin_lock_irq(spinlock_t *lock);
spin_lock_bh(spinlock_t *lock);

spin_unlock(spinlock_t *lock);
spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
spin_unlock_irq(spinlock_t *lock);
spin_unlock_bh(spinlock_t *lock);
spin_is_locked(spinlock_t *lock);
spin_trylock(spinlock_t *lock)
spin_unlock_wait(spinlock_t *lock);
```

The “save” version

it allows not to interfere with IRQ management along the path where the call is nested

a simple masking (with no saving) of the IRQ state may lead to misbehavior

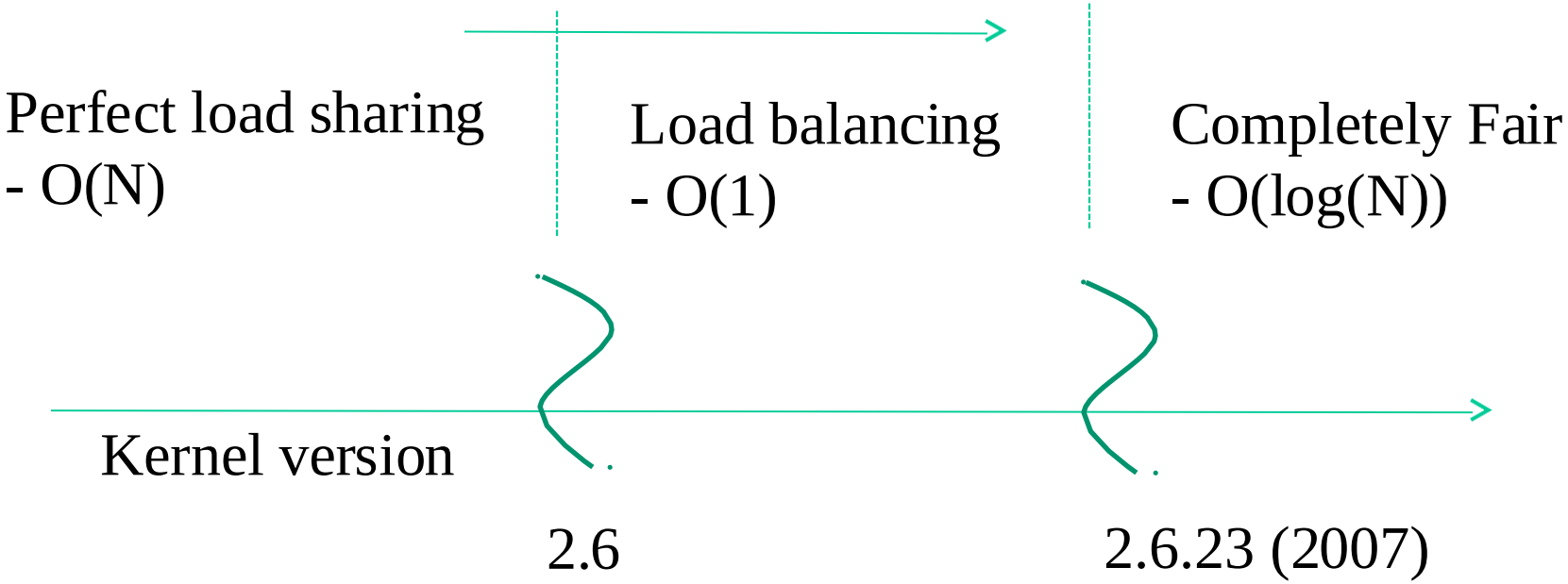


Variants - discriminating readers vs writers

```
rwlock_t xxx_lock = __RW_LOCK_UNLOCKED(xxx_lock);
unsigned long flags;

read_lock_irqsave(&xxx_lock, flags);
.. critical section that only reads the info ...
read_unlock_irqrestore(&xxx_lock, flags);


write_lock_irqsave(&xxx_lock, flags);
.. read and write exclusive access to the info ...
write_unlock_irqrestore(&xxx_lock, flags);
```

The Linux scheduler logic evolution

Improved orientation to SMP/multi-core and fairness



Scheduler logic - traditional baseline aspects

- The planning of tick usage <u>is based on epochs</u>
- An epoch ends when all threads on the runqueue have already ended their ticks
- <u>Threads on waitqueues may still have residuals</u>
- When an epoch ends we recompute the ticks to be assigned to all threads for the next epoch
- Assigned tick volumes reflect priorities

Actual priority scheme - Posix classic

```
<- higher priority

-20_ _ _ _ _ 0 _ _ _ _ _+19
```

We can move across priority values by exploiting thread niceness

Perfect load sharing scheduler

- What TCB do we look at upon the execution of `schedule()`?
- <u>ALL those that are not on a waitqueue</u>
- Ideally any thread can be CPU-dispatched on any CPU-core at any time instant
- CPU-scheduling decisions based on priorities and on the target of maximizing hardware effectiveness (e.g. caching)

The 2.4 kernel perfect load sharing scheduler

- The execution of the function `schedule()` can be seen as entailing 3 distinct phases:

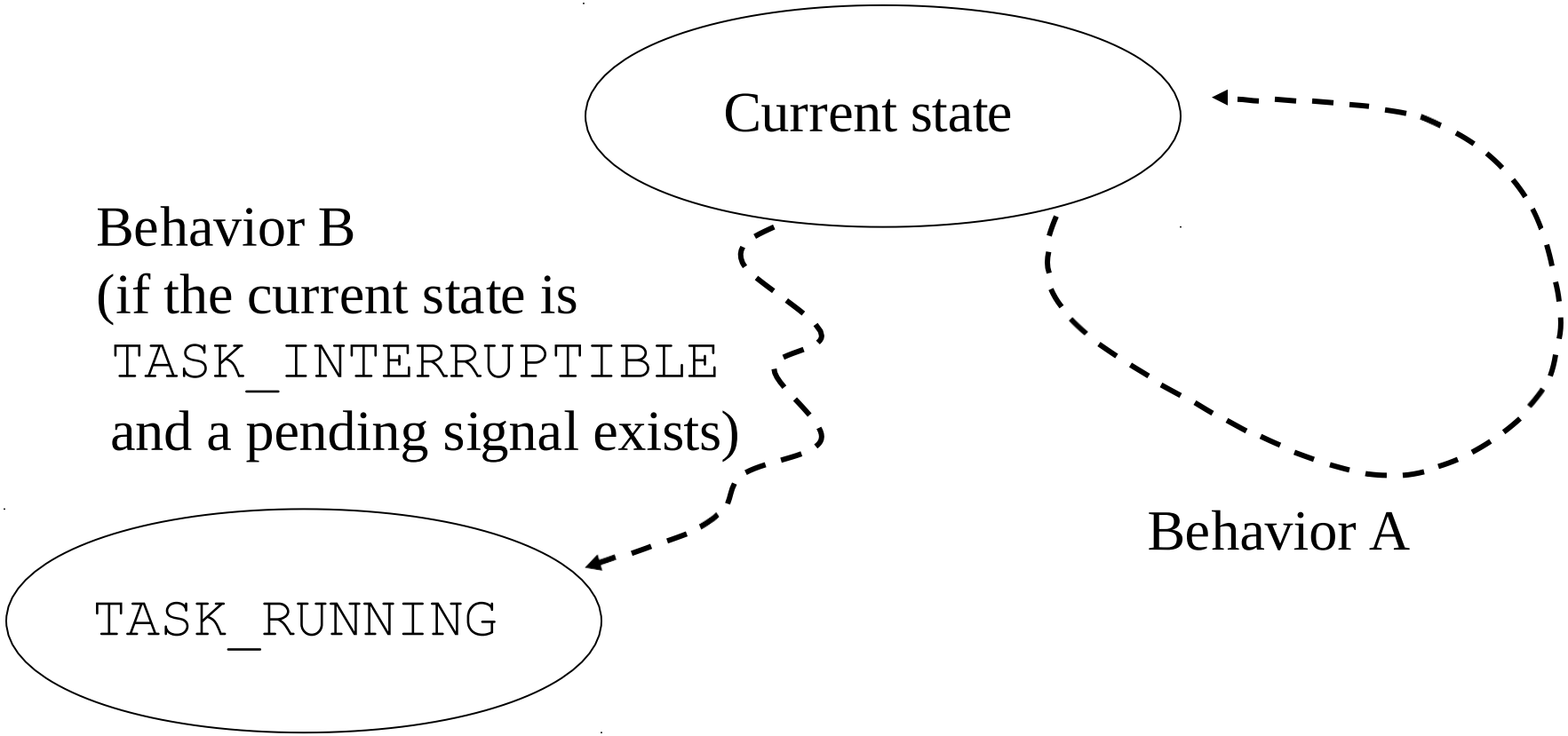
1) check on the current process (does it really need to be removed from the runqueue?)

2) “Run-queue analysis” (next process selection) of the unique runqueue in the overall system – affinity still works here

3) context switch to the next process (actually thread)

Check on the current process - update of the process state

```
……
prev = current;
……
switch (prev->state) {
        case TASK_INTERRUPTIBLE:
            if (signal_pending(prev)) {
                prev->state = TASK_RUNNING;
                break;
            }
        default:
            del_from_runqueue(prev);
        case TASK_RUNNING:;
    }
prev->need_resched = 0;
```


Current state
Behavior B
(if the current state is
TASK_INTERRUPTIBLE
and a pending signal exists)
Behavior A
TASK_RUNNING

Helps

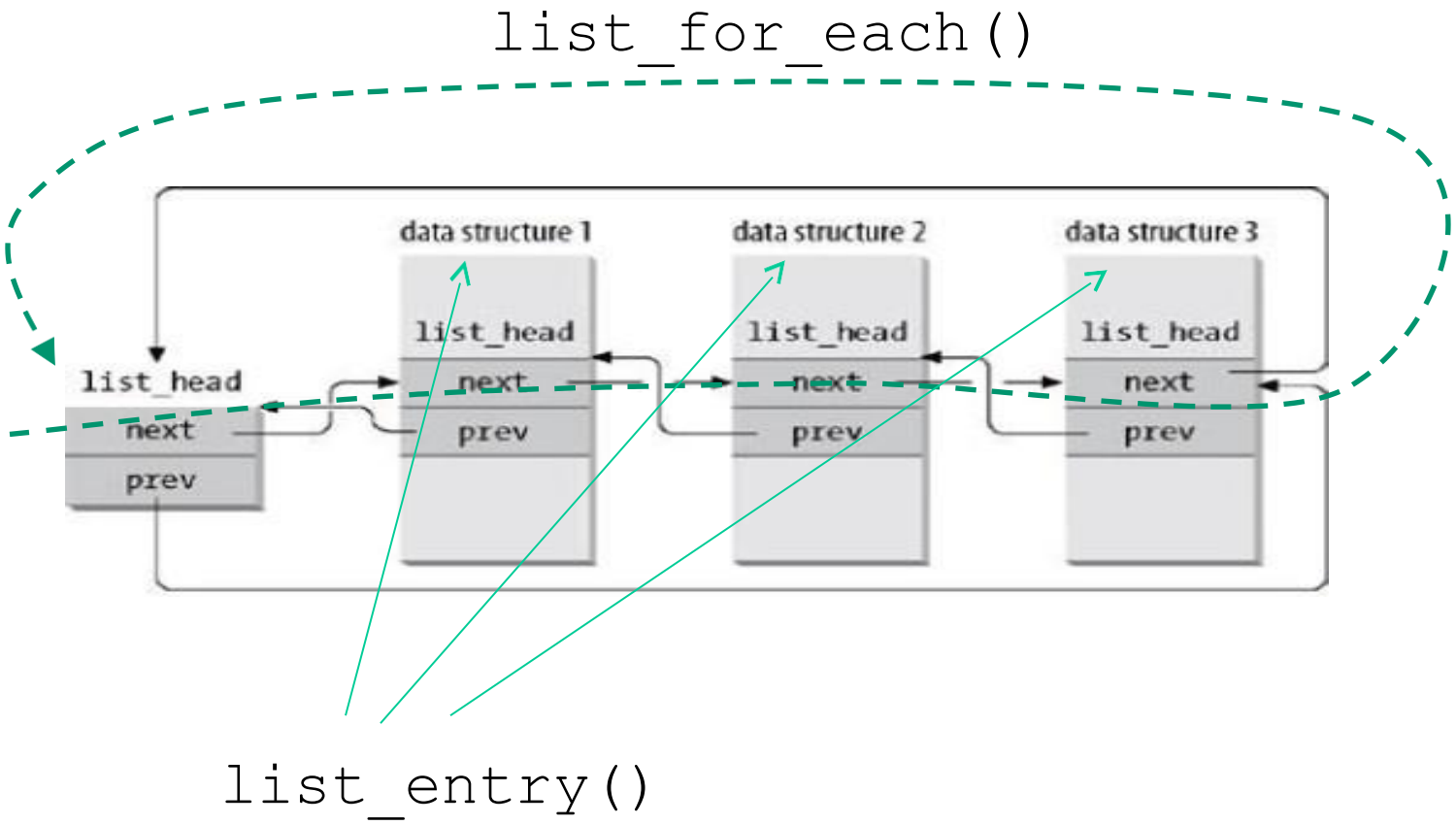
```
#define list_for_each(pos, head) \
for (pos = (head)->next; pos != (head); pos = pos->next)
```

Scan of a circular list through a cursor (i.e. `pos`)

Access to the container element in the list linkage

```
#define list_entry(ptr, type, member) \
container_of(ptr, type, member)
```

A scheme


list_for_each()
data structure 1
data structure 2
data structure 3
list_head
next
prev
list_head
next
prev
list_head
next
prev
list_head
next
prev
list_entry()

Run queue analysis

- for all the TCBs currently registered within the run-queue a so called **goodness value** is computed
- the TCB associated with the best goodness value gets pointed by `next` (which is initially set to point to the idle-process TCB)

```
repeat_schedule:
        /* Default process to select..*/
    next = idle_task(this_cpu);
    c = -1000;
    list_for_each(tmp, &runqueue_head) {
          p = list_entry(tmp, struct task_struct, run_list);
          if (can_schedule(p, this_cpu)) {
           int weight = goodness(p, this_cpu, prev->active_mm);
           if (weight > c)
           c = weight, next = p;
          }
   }
```

The role of memory mappings

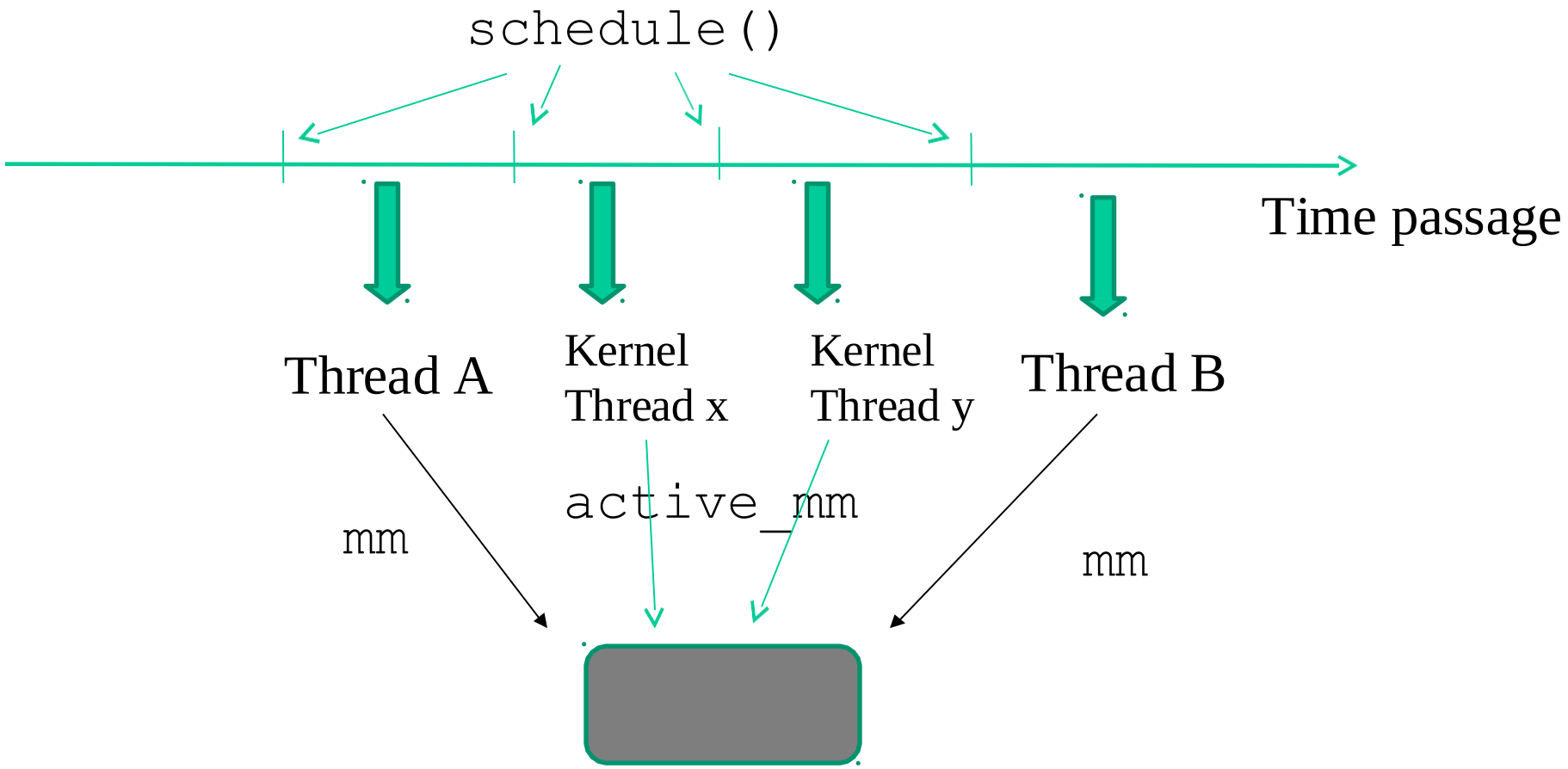
`mm_struct` fileds in the TCB are 2 (not just one)

```
struct mm_struct *mm;
struct mm_struct *active_mm;
```

This is the user space memory mapping of the last thread run on this same CPU

- ✓ For an application thread `mm == active_mm` is an invariant
- ✓ For a kernel level thread `mm == NULL` but `active_mm` can be different from `NULL`

Memory mappings and timelines



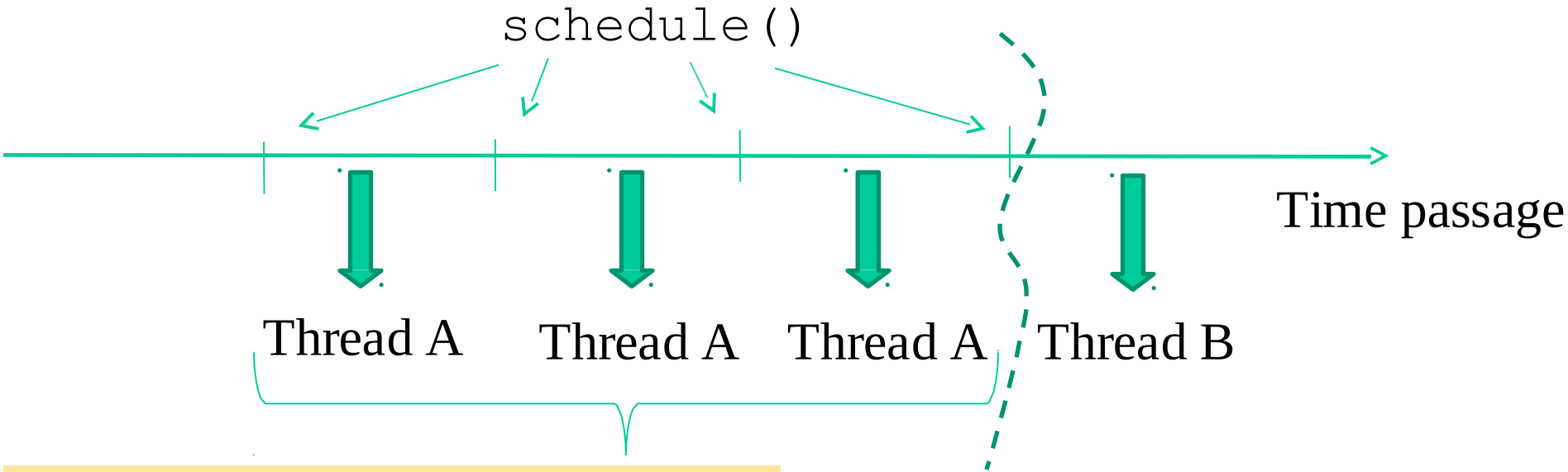
Computing the goodness

goodness (p) =	20 – `p->nice`	(base time quantum)
	+ `p->counter`	(ticks left in time quantum)
	+1	(if page table is shared with the previous process)
	+15	(in SMP, if p was last running on the same CPU)

NOTE: goodness is forced to the value 0 in case `p->counter` is zero

Kind of batch ticks usage

The +15 bonus tends to cluster tick usage by threads on a same CPU



Extreme exploitation of program flow and architectural support for locality

`p->counter == 0`
for thread A

Management of the epochs

Any epoch ends when all the threads registered within the run-queue already used their planned CPU quantum

This happens when the residual tick counter (`p->counter`) reaches the value zero for all the TCBs kept by the run-queue

Upon epoch ending, the next quantum is computed for all the active threads

The formula for the recalculation is as follows

```
p->counter = p->counter /2 + 6 - p->nice/4
```

```
……
/* Do we need to re-calculate counters? */
if (unlikely(!c)) {
        struct task_struct *p;

        spin_unlock_irq(&runqueue_lock);
        read_lock(&tasklist_lock);
        for_each_task(p)
                p->counter = (p->counter >> 1) +
                NICE_TO_TICKS(p->nice);
        read_unlock(&tasklist_lock);
        spin_lock_irq(&runqueue_lock);
        goto repeat_schedule;
}
……
```

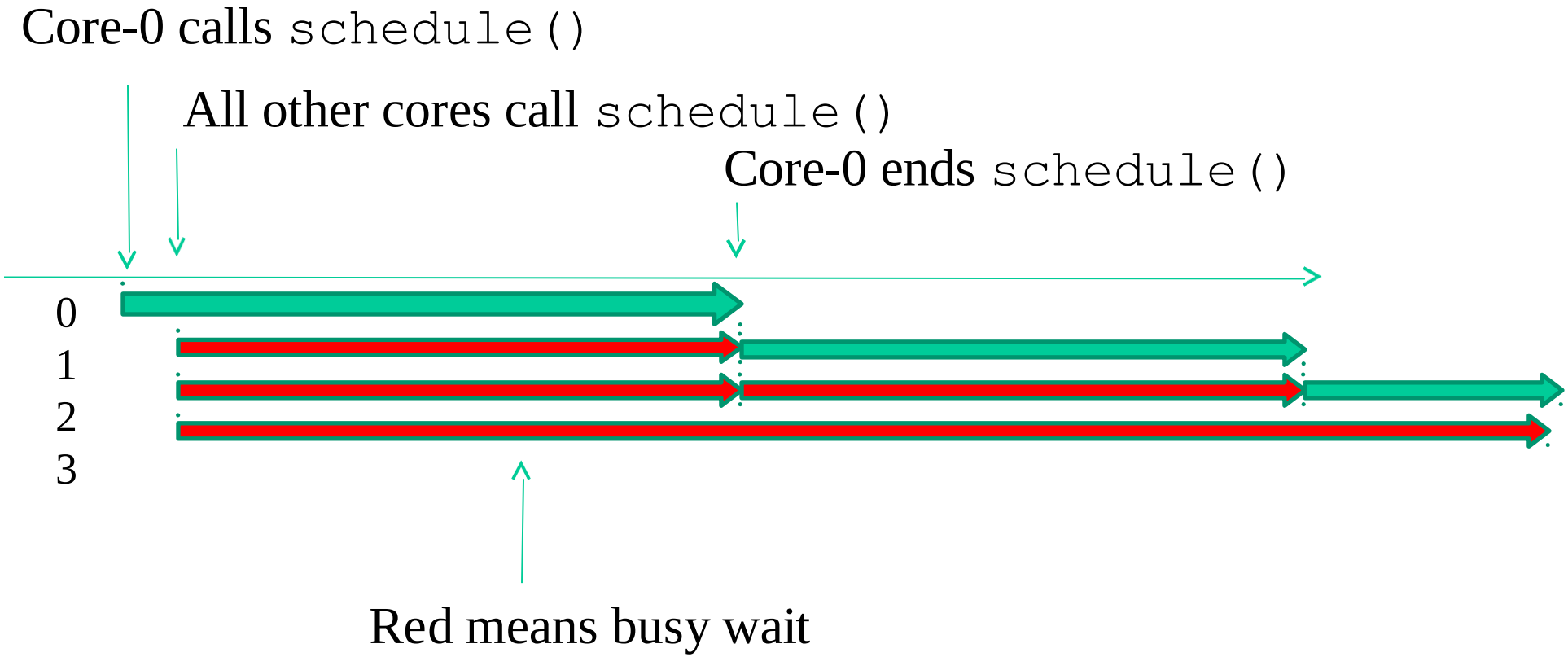
Perfect load sharing - O(n) scheduler causes

- A non-runnable task is anyway searched to determine its goodness

- Mix of runnable/non-runnable tasks into a single run-queue in any epoch

- Chained negative performance effects in atomic scan operations in case of SMP/multi-core machines (length of critical sections dependent on system load)

A timeline example with 4 processing units



Newer CPU-scheduling internals - load balancing

- Constant-time – O(1) – scheduling

- Very low frequency of collisions by CPUs in inspecting a same run-queue

- Still keep the workload balanced (in compliance with affinity)

- Still distinguish priorities (even more levels with respect to what done before)

Constant time scheduling with load balancing

- **No mix of runnable and non-runnable tasks on a runqueue**

- Clear separation of runnable tasks into multiple run queues

  we **do not search for priorities into the TCBs**, we already know it, based on the runqueue a TCB stands onto

Infrequent CPU-conflicts in the access to runqueues

- Fully separated runqueues, one per CPU

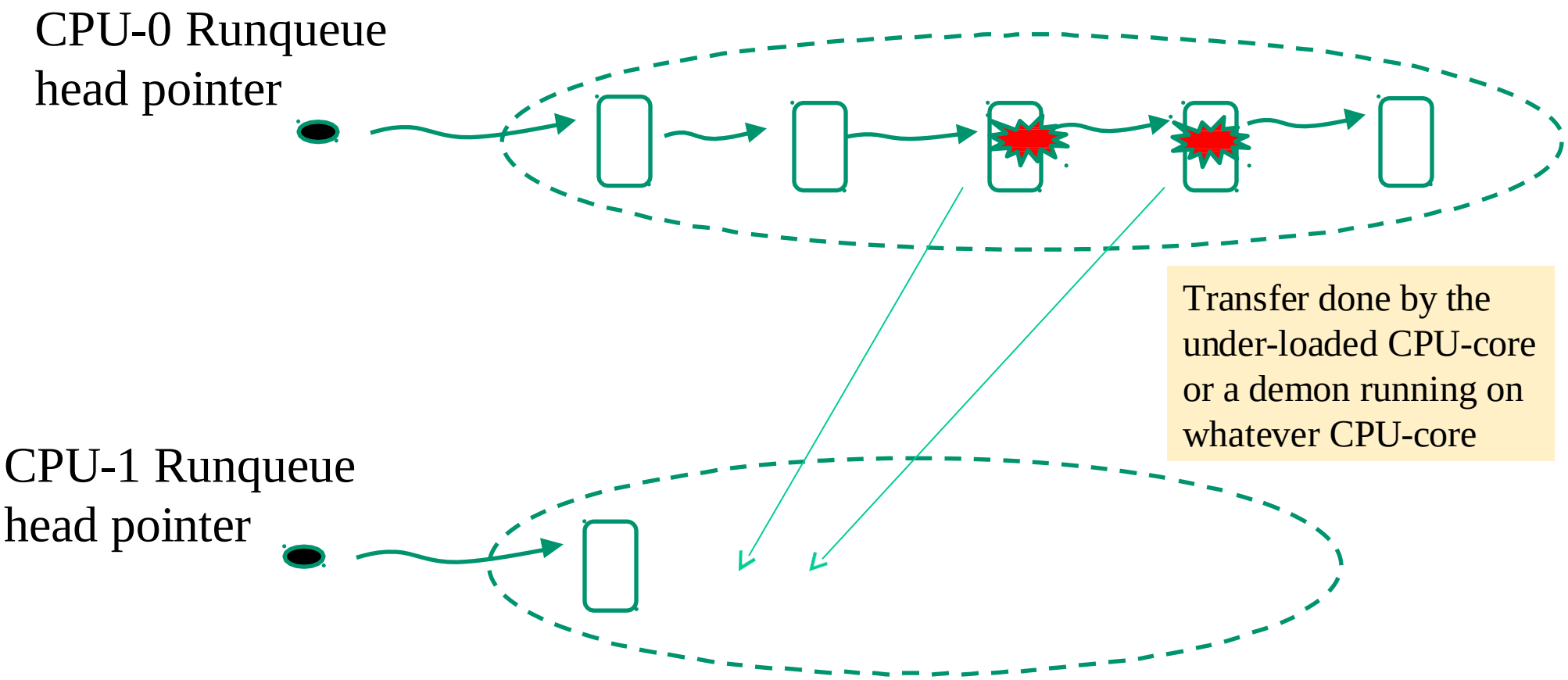
- Each CPU-core accesses its own runqueue when running the scheduler logic

- A CPU-core can access the runqueue of another one (hopefully infrequently) when

  → An explicit linkage of the TCB on that run queue is requested

  → This is for load balancing or for promptness of reschedule

Load balancing example



Actual implementation on Linux kernel 2.6

- The run queue of each CPU-core is a multiqueue with 140 different levels

- 40 levels (say [100-139]) map to classical Unix time-sharing

- 100 levels (say [0-99]) map to Unix real-time scheduler extensions

- It is also separated into

    The active queue, keeping runnable threads

    The expired queue, keeping non-runnable threads

The priority scale - kernel level representation

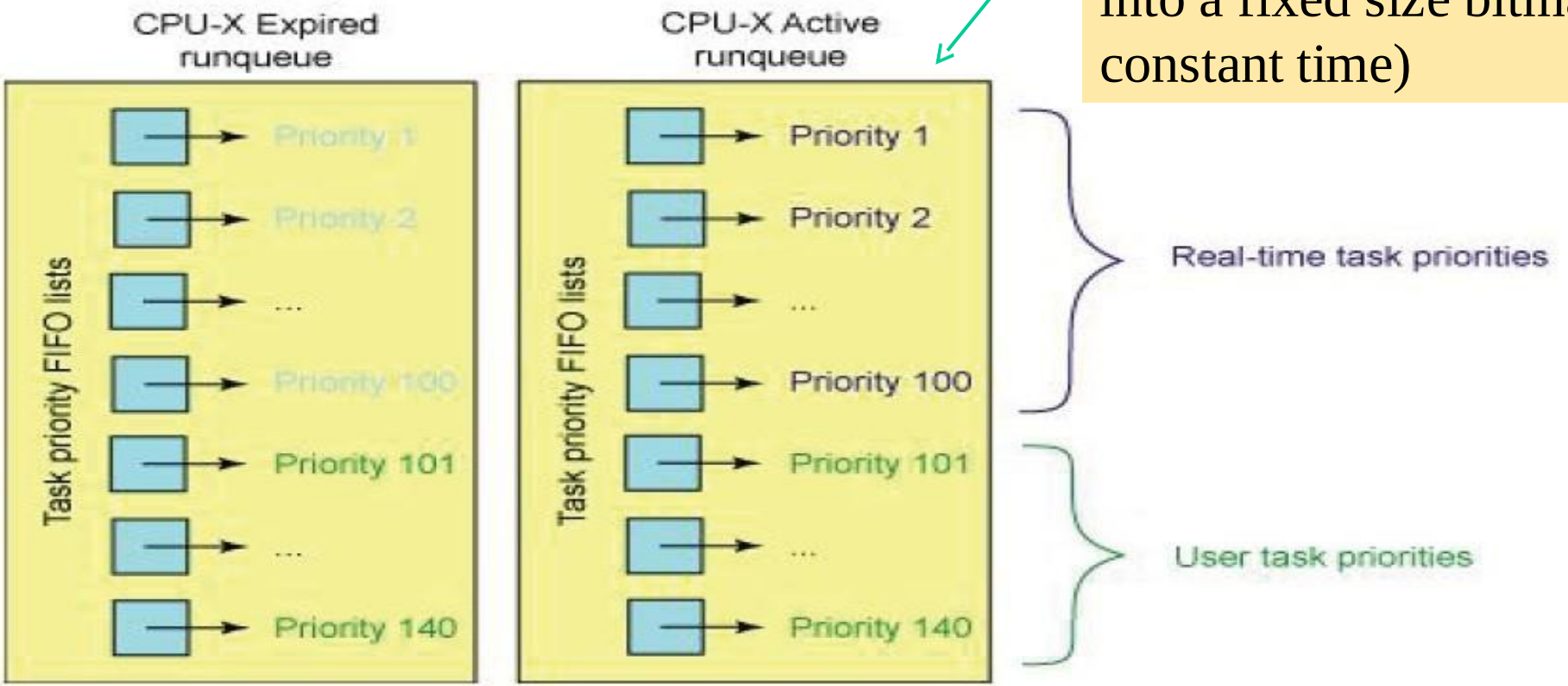
```
                    <- higher priority

0_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _99|100_ _ _ _ _ _ _ _ _ _ _ _139
          real-time                        |          normal
                                           |
                                           |
       SCHED_RR/SCHED_FIFO                 |       SCHED_OTHER
```

Manageable with the `sched_setscheduler()` syscall or the `chrt` shell command

A scheme

We simply switch the queues upon a new epoch

We search for a non empty queue level by searching into a fixed size bitmap (in constant time)



Relations with the thread wakeup API

```
wake_up_process(…)
```

Can the thread run on this CPU?
If YES put on the local runqueue

If NO, get affinity info from TCB and put in some remote runqueue via the below API

```
void ttwu_queue(struct task_struct *p,
        int cpu, int wake_flags)
```

“Load” vs ticks

- In load sharing, the assignment of ticks to be spent by a thread is based on the notion of “load”
- This is an information kept within a new field of the TCB structured as

```
struct sched_entity {
    struct load_weight  load;
  …
}

struct load_weight {
    unsigned long weight;
    u32 inv_weight;
};
```

This value is assigned on the basis of the niceness and is used in a calculation to assign the number of ticks …….

Weight assignment vector (kernel 3 snippet)

```
static const int prio_to_weight[40] = {
    /* -20 */     88761,     71755,     56483,     46273,     36291,
    /* -15 */     29154,     23254,     18705,     14949,     11916,
    /* -10 */      9548,      7620,      6100,      4904,      3906,
    /*  -5 */      3121,      2501,      1991,      1586,      1277,
    /*   0 */      1024,       820,       655,       526,       423,
    /*   5 */       335,       272,       215,       172,       137,
    /*  10 */       110,        87,        70,        56,        45,
    /*  15 */        36,        29,        23,        18,        15,
};
```

Moving one entry up or down (depending on niceness) leads to achieve 10% more or less CPU time to exploit

Renamed to const int sched_prio_to_weight[40] in Kernel 4/5/6

Additional priority details

- A non-real-time thread has two characterizing priority values
  - ✓ **the static priority** – this is defined by the users (linked to niceness) and defines the level at which the thread will appear in the runqueue
  - ✓ **the dynamic priority** – this is based on a reward or a penalty (applied to the static priority) depending on whether the thread is interactive or not
- <u>Thread is interactive if its sleep time is high enough</u>, and the reward is based on a formula that considers the sleep time
- Both these priority values appear as recorded into the TCB
- The one that is looked at when we run the `schedule()` function is the dynamic priority

The effect of dynamic priorities

- A thread that calls the schedule function can be preempted by one that has higher dynamic priority (although lower static priority)
- A classical scenario
  1. The thread calls wakeup of some other thread
  2. The thread calls schedule
- Another classical scenario
  1. Someone calls wakeup putting a thread on the queue of another CPU
  2. The CPU is then hit by a cross-CPU reschedule-request

CPU-scheduling API - a wider view

p->time_slice	The residual ticks in the current epoch
schedule	The main scheduler function. Schedules the highest priority task for execution.
load_balance	Checks the CPU to see whether an imbalance exists, and attempts to move tasks if not balanced.
effective_prio	Returns the effective priority of a task (based on the static priority, but includes any rewards or penalties).
recalc_task_prio	Determines a task's bonus or penalty based on its idle time.
source_load	Calculates the load of the source CPU (from which a task could be migrated).
target_load	Calculates the load of a target CPU (where a task has the potential to be migrated).

Explicit stack refresh

It is a software operation

It is used when an action is finalized via local variables with lifetime across different reschedules

Used in 2.6 or later versions for `schedule()` finalization

Local variables are explicitly repopulated after the stack switch has occurred

```
asmlinkage void __sched schedule(void)
{
     struct task_struct *prev, *next;
     unsigned long *switch_count;
     struct rq *rq;
     int cpu;

need_resched:
     preempt_disable();
     cpu = smp_processor_id();
     rq = cpu_rq(cpu);
     rcu_qsctr_inc(cpu);
     prev = rq->curr;
     switch_count = &prev->nivcsw;

     release_kernel_lock(prev);
need_resched_nonpreemptible:

     ……..
     spin_lock_irq(&rq->lock);
     update_rq_clock(rq);
     clear_tsk_need_resched(prev);
     ……..
```

```
    ……
#ifdef CONFIG_SMP
        if (prev->sched_class->pre_schedule)
            prev->sched_class->pre_schedule(rq, prev);
#endif
        if (unlikely(!rq->nr_running)) idle_balance(cpu, rq);

        prev->sched_class->put_prev_task(rq, prev);
        next = pick_next_task(rq, prev);

        if (likely(prev != next)) {
            sched_info_switch(prev, next);

            rq->nr_switches++;
            rq->curr = next;
            ++*switch_count;

            context_switch(rq, prev, next); /* unlocks the rq */
            /* the context switch might have flipped the stack from under
               us, hence refresh the local variables. */
            cpu = smp_processor_id();
            rq = cpu_rq(cpu);
        } else spin_unlock_irq(&rq->lock);

        if (unlikely(reacquire_kernel_lock(current) < 0))
            goto need_resched_nonpreemptible;
        preempt_enable_no_resched();
        if (unlikely(test_thread_flag(TIF_NEED_RESCHED)))
            goto need_resched;
}
```

Struct rq (run-queue)

```
struct rq {
    /* runqueue lock: */
    spinlock_t lock;

    /* nr_running and cpu_load should be in the same cacheline because remote CPUs
use both these fields when doing load calculation. */
    unsigned long nr_running;
    #define CPU_LOAD_IDX_MAX 5
    unsigned long cpu_load[CPU_LOAD_IDX_MAX];
    unsigned char idle_at_tick;
        ……….
    /* capture load from *all* tasks on this cpu: */
    struct load_weight load;

    …….
    struct task_struct *curr, *idle;
    ……..
    struct mm_struct *prev_mm;
    ……..
};
```

Finally - completely fair scheduling - kernel 2.6.23 or later ones

- No longer run queues for selecting time-shared TCBs

- A red/black tree is used and threads are ordered by used VCPU (Virtual CPU) time (the lower the better)

- Granularity of measurements is nanoseconds

- The actual ordering within the red/black tree reflects dynamic priorities at much better granularity compared to heuristics based on waiting time

Completely fair scheduling concepts

- *N* equally important threads should have exactly *1/N* of the CPU time over an observation window

- In real scenarios this is only approximated by the fact that we typically use the tick timer with a minimum granularity (to avoid context switch over frequency)

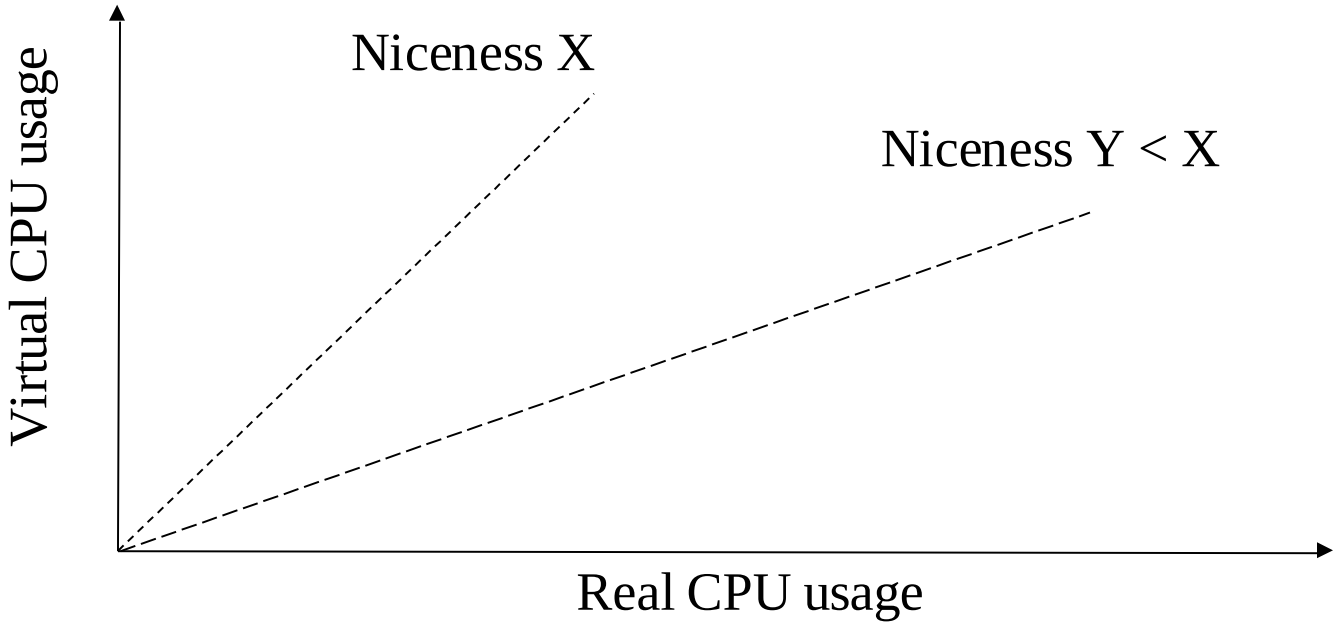
- Also, threads not all have the same importance

- In this scheduler we use load weights to determine the VCPU time advancement of threads

VCPU advancement

- It is computed as real CPU usage normalized by the schedulable entity weight
- The more the weight, the less the VCPU usage (fixed the real CPU usage)
- Schedulable entities are ordered into a red/black tree based on VCPU usage - O(log(N)) cost
- The less the VCPU usage, the sooner the schedulable entity will take control of the CPU

A graphical representation



Kernel threads - initial 2.4/i386 binding …

kernel threads can be generated via the function `kernel_thread()` defined in `kernel/fork.c`

This function relies on an ASM function called `arch_kernel_thread()` which is `arch/i386/kernel/process.c`

The latter does some job before calling `sys_clone()`

Upon returning within the child thread, the target thread function is executed via a call

In this scenario, the base of user mode stack is a don’t care since this thread will never bounce to user mode

```
long kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
{
    struct task_struct *task = current;
    unsigned old_task_dumpable;
    long ret;

    /* lock out any potential ptracer */
    task_lock(task);
    if (task->ptrace) {
        task_unlock(task);
        return -EPERM;
    }

    old_task_dumpable = task->task_dumpable;
    task->task_dumpable = 0;
    task_unlock(task);

    ret = arch_kernel_thread(fn, arg, flags);

    /* never reached in child process, only in parent */
    current->task_dumpable = old_task_dumpable;

    return ret;
}
```

```
int arch_kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
{
     long retval, d0;

     __asm____volatile__(
     "movl %%esp,%%esi\n\t"
     "int $0x80\n\t"         /* Linux/i386 system call */
     "cmpl %%esp,%%esi\n\t" /* child or parent? */
     "je 1f\n\t"        /* parent - jump */
     /* Load the argument into eax, and push it.   That way, it does
      * not matter whether the called function is compiled with
      * -mregparm or not.   */
     "movl %4,%%eax\n\t"
     "pushl %%eax\n\t"
     "call *%5\n\t"          /* call fn */
     "movl %3,%0\n\t"   /* exit */
     "int $0x80\n"
     "1:\t"
     :"=&a" (retval), "=&S" (d0)
     :"0" (__NR_clone), "i" (__NR_exit),
     "r" (arg), "r" (fn),
      "b" (flags | CLONE_VM)
     : "memory");

     return retval;
}
```

More recent (module exposed) API

The thread function

The function param

```
truct task_struct *kthread_create(int (*function)(void *data),void *data,
const char name[])
```

Exec style naming

In the end this service relies on the core thread-startup function seen before plus others

Thread features with kthread_create

The created thread sleeps on a wait queue

So it exists but is not really active

We need to explicitly awake it

As for signals we have the following:

- We can kill, if thread (or creator) enables
- Killing only has the effect of awakening the thread (if sleeping)
- Terminating threads via kills is based on the thread polling a termination bit in its TCB or on polls on the signal mask

Kernel threads vs affinity

```
truct task_struct *kthread_create_on_cpu(int (*function)(void *data),
                                         void *data,
                                         unsigned int cpu_id,
                                         const char name[])
```

Affinity settings for the new thread