


Advanced Operating Systems
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

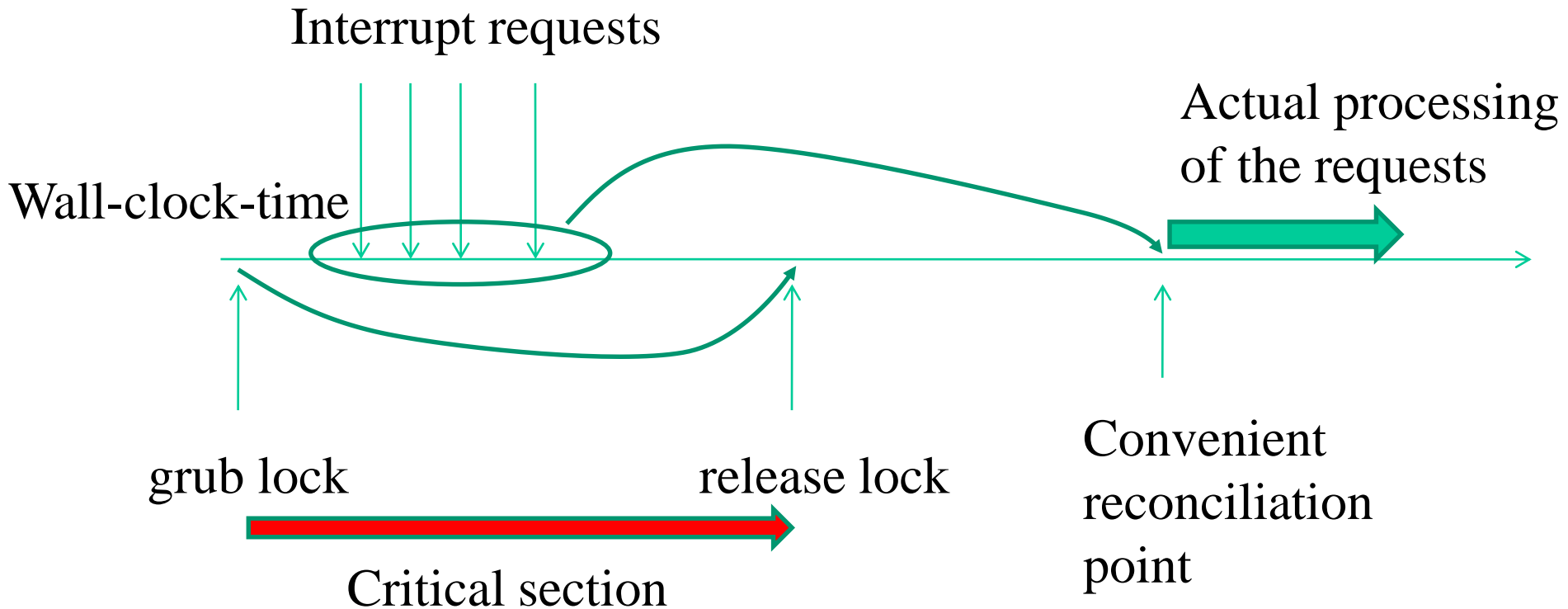
- Types of traces
 - User mode process/thread
 - Kernel mode process/thread
 - Interrupt management
- Non-determinism
 - Due to nesting of user/kernel mode traces and interrupt management traces
- Performance
 - Non-determinism may give rise to inefficiency whenever the evolution of the traces is tightly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

Design methodologies

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 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

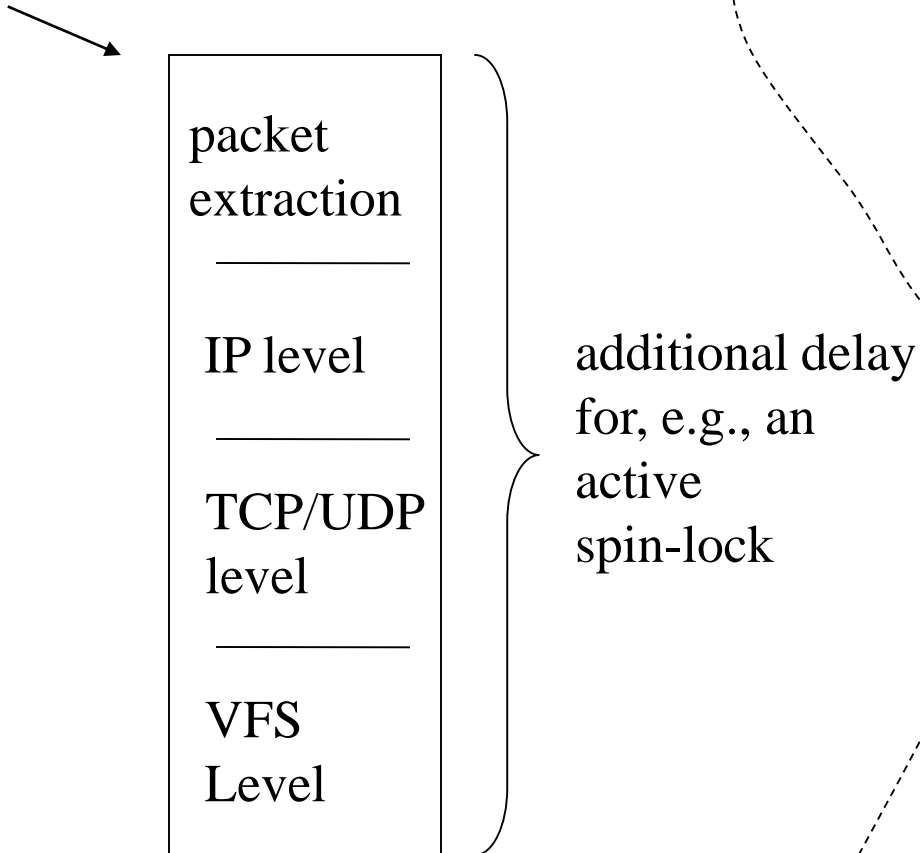
- The management of tasks associated with the interrupts typically occurs via a two-level logic: top half e bottom half
- The top-half level takes care of executing a minimal amount of work which is needed to allow later finalization of the whole interrupt management
- The top-half code portion is typically (but not manadatorily) handled according to a non-interruptible 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 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

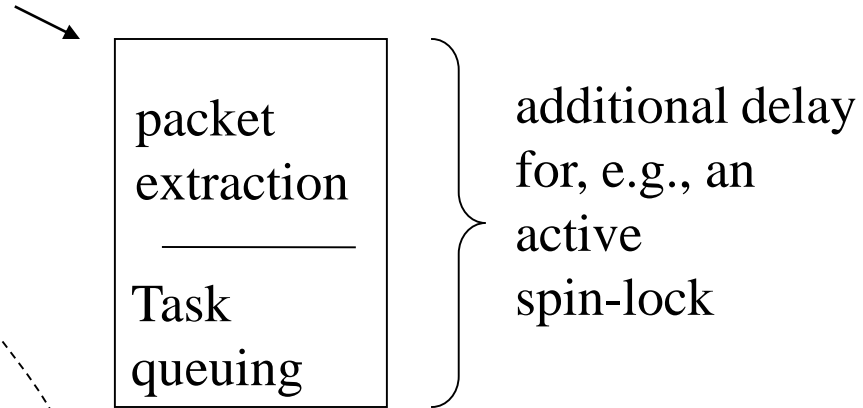
no top/bottom half

interrupt from network device

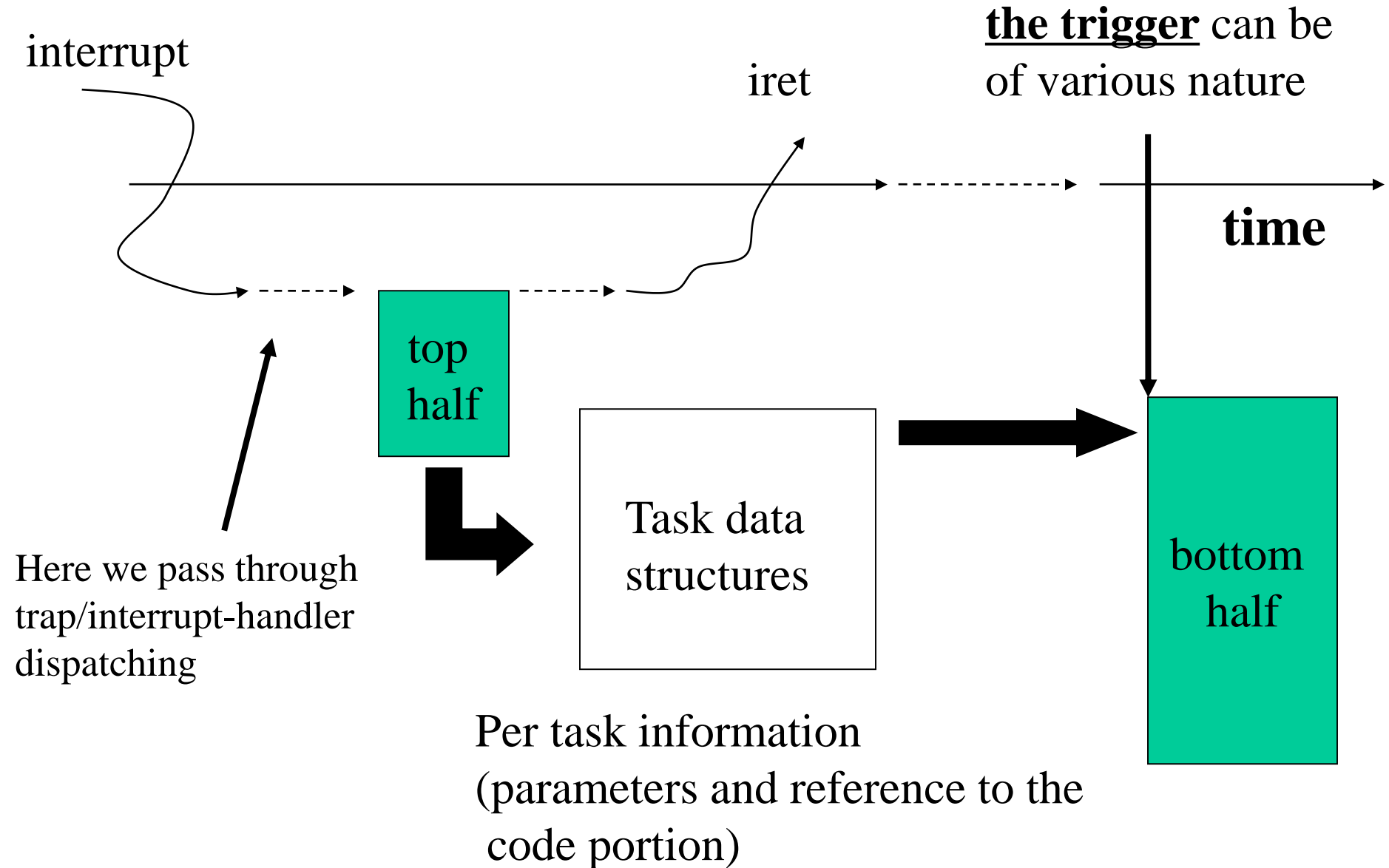


top/bottom half

interrupt from network device

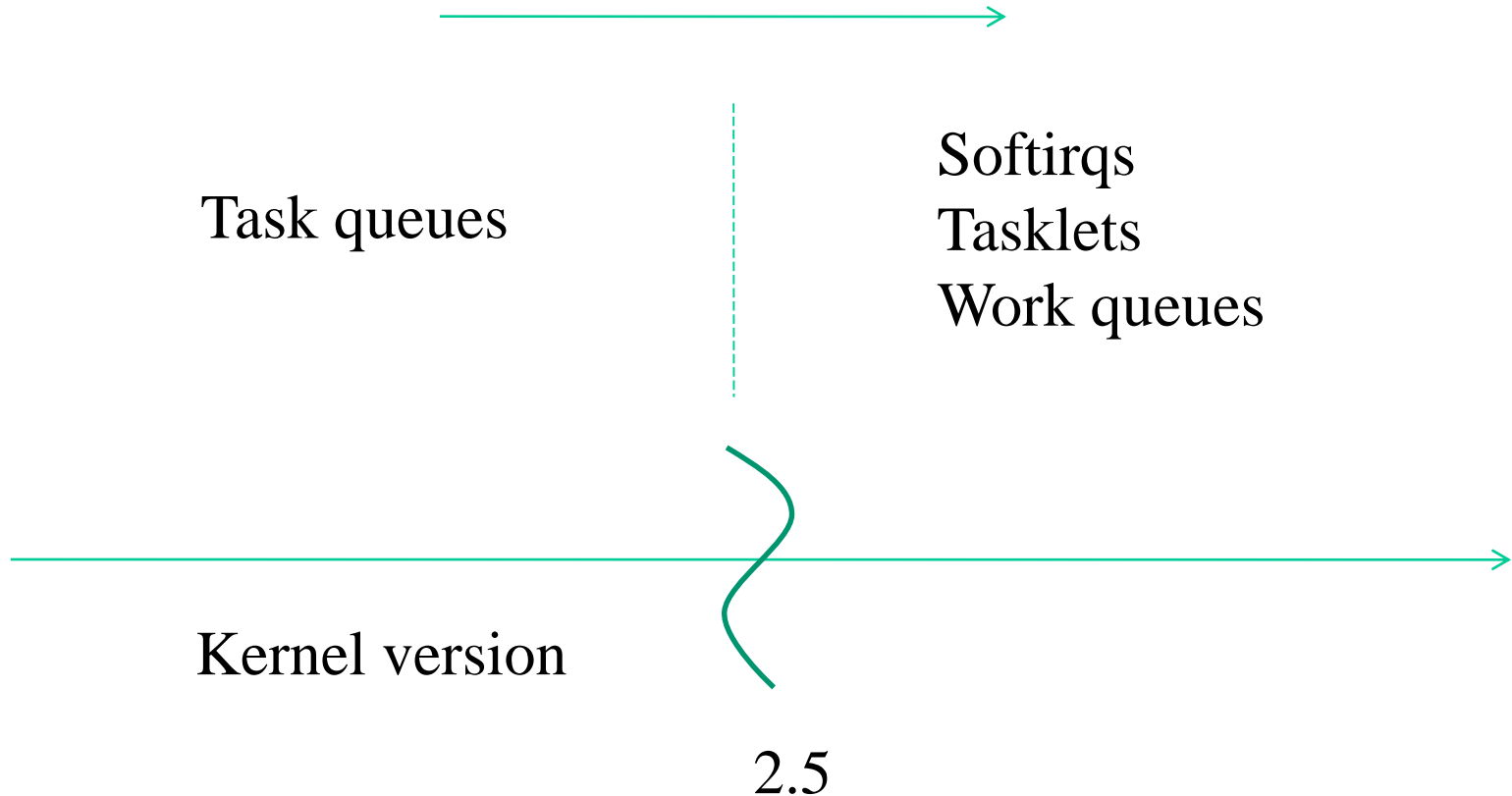


The historical architectural concept: bottom-half queues



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 */  
}
```

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 the function** `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
- **Note**: 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

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

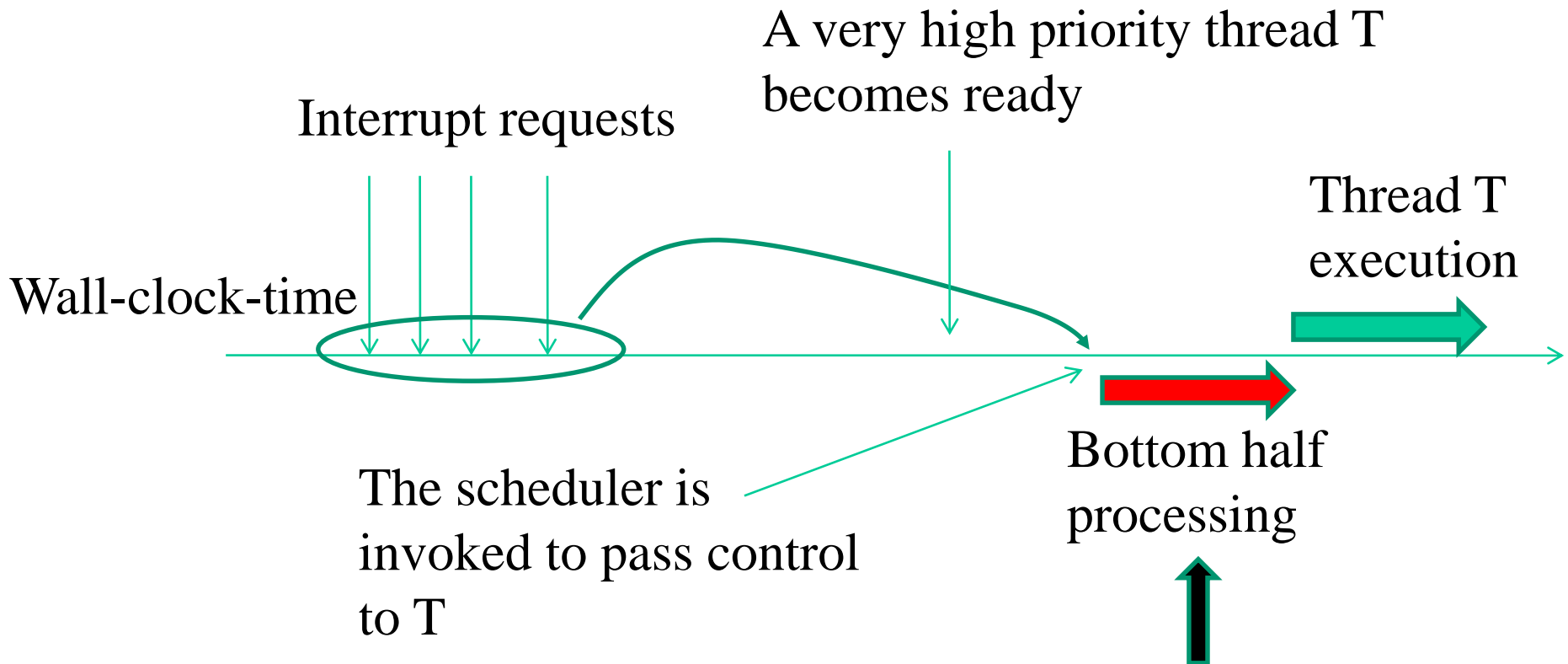
The Linux kernel (up to 2.5) invokes **do_bottom_half()**

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

Limitations of task queues: the actual 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

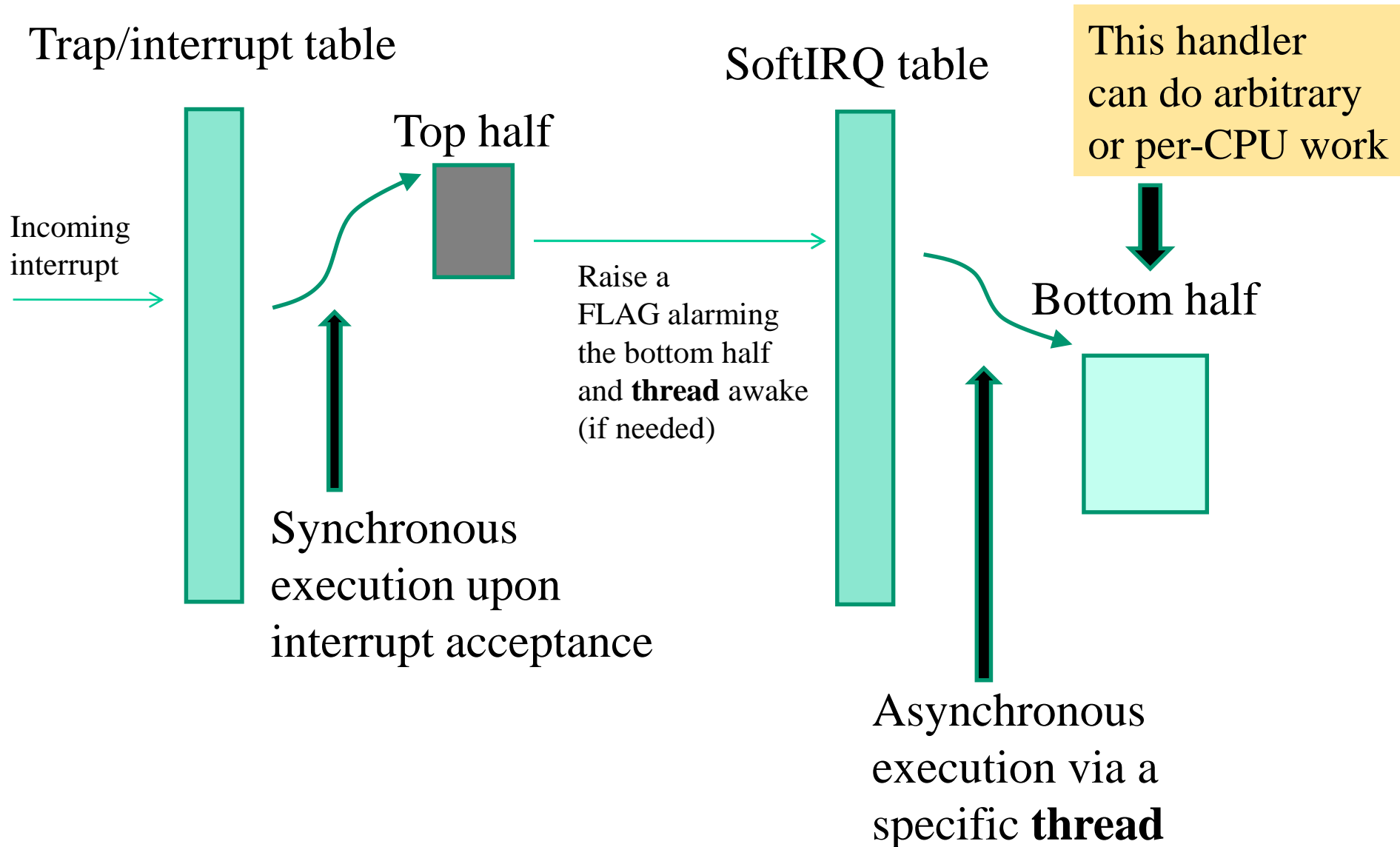
Parallelism vs interrupts vs device drivers

- “Interrupts” can be also be raised by software
- This is the scenario of drivers for logical (not physical) devices
- So interrupt drivers may be requested to handle a load that may grow with the number of running threads
- 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?

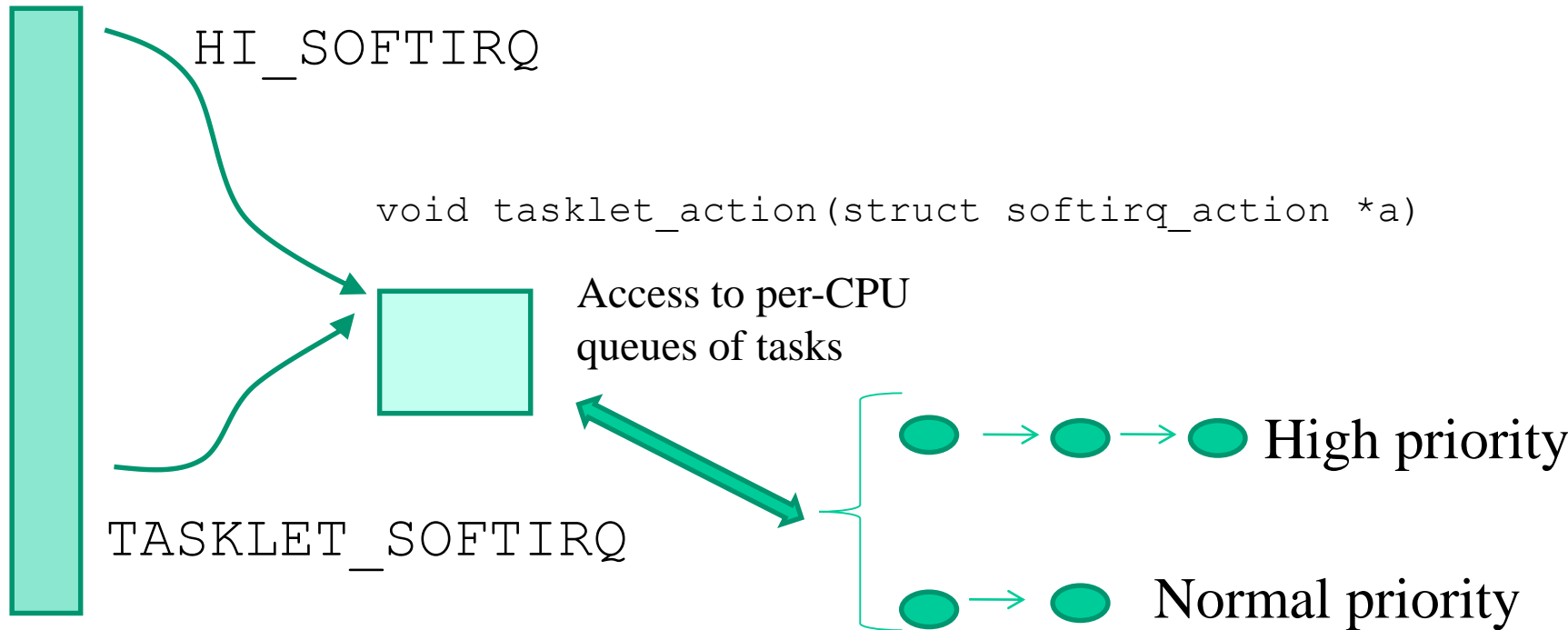
- The `ksoftirq` daemon (multiple threads with CPU affinity)
- This is typically listed as `ksoftirq[n]` where 'n' is the CPU-core it is affine with
- Once awoken, the threads look at the SoftIRQ table to inspect if some entry is flagged
- In the positive case the thread runs the softIRQ handler
- We can also build a mask telling that a thread awoken on a CPU-core X will not process the handler associated with a given 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

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

SoftIRQ table



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/linux/interrupt.h`:
 - `DECLARE_TASKLET(tasklet, function, data)`
 - `DECLARE_TASKLET_DISABLED(tasklet, function, data)`
- `name` is the tasklet 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, 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 via specific kernel threads (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's 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 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 (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);
```

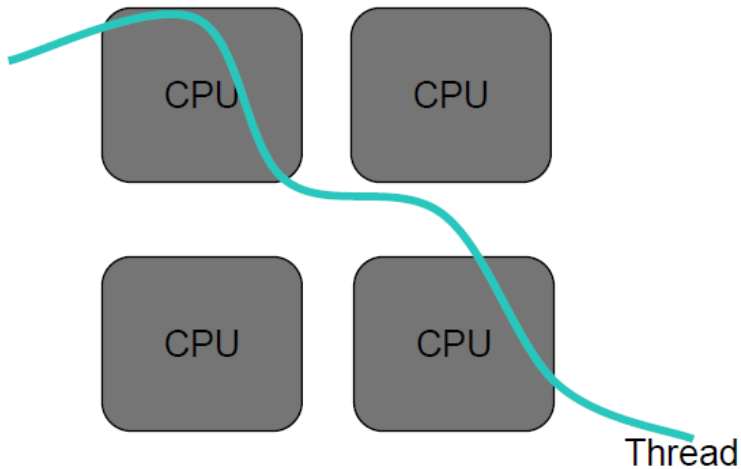
Both create a `workqueue_struct` (with one entry per processor)
The second provides the support for flushing the queue via a
single worker thread (and no affinity of jobs)

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

This eliminates the queue

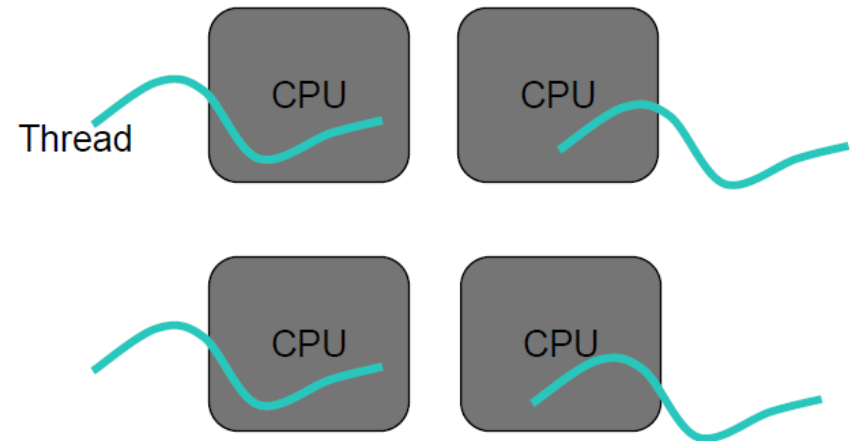
Actual scheme

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

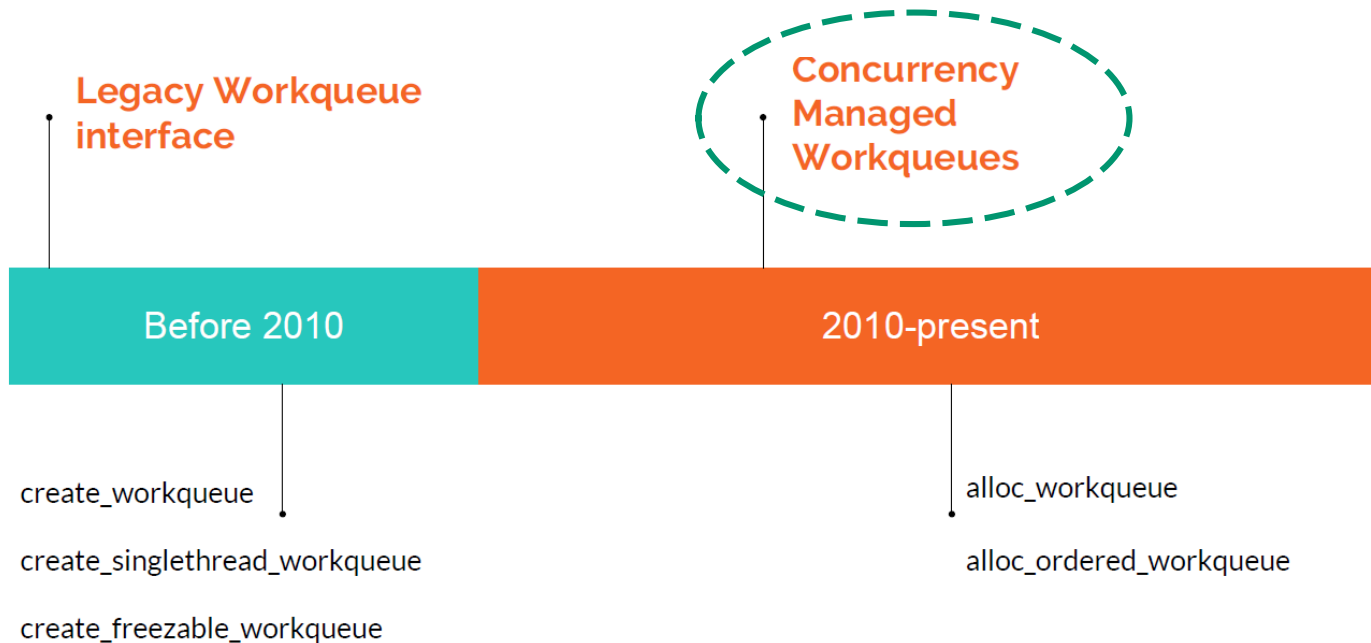
Interface and functionality evolution

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

- **Older:**

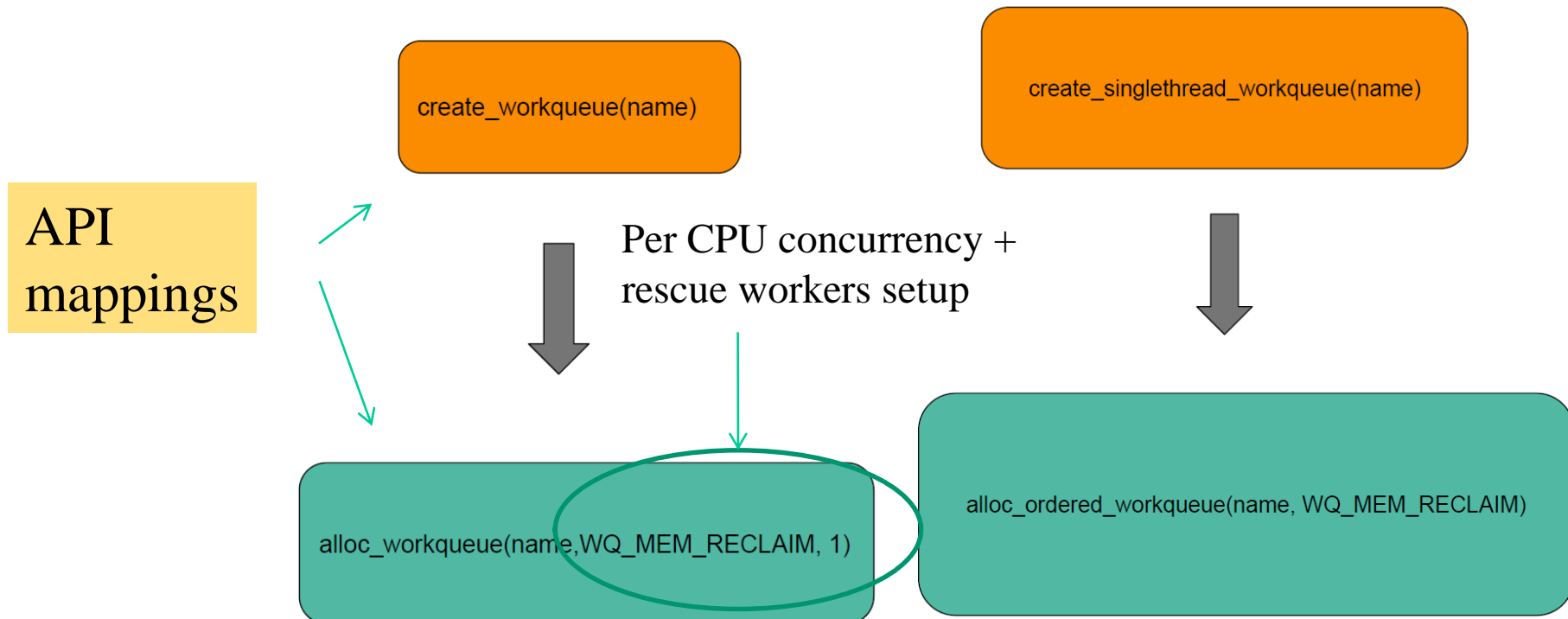
`create[_singlethread|_freezable]_workqueue()`

- **Newer:** `alloc[_ordered]_workqueue()`



Concurrency managed work queues

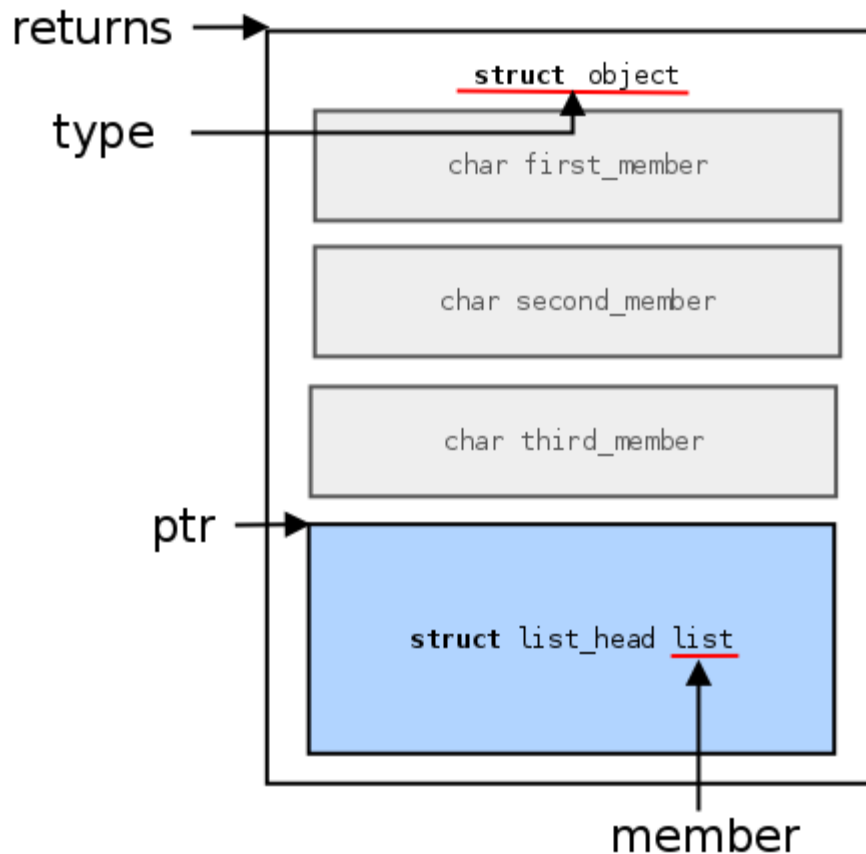
- Uses per-CPU unified worker pools shared by all work queues to provide flexible levels of concurrency on demand without wasting a lot of resources
- Automatically regulates the worker pool and level of concurrency so that the users don't need to worry about such details



Managing dynamic memory with (not only) work queues

`container_of(ptr, type, member)`

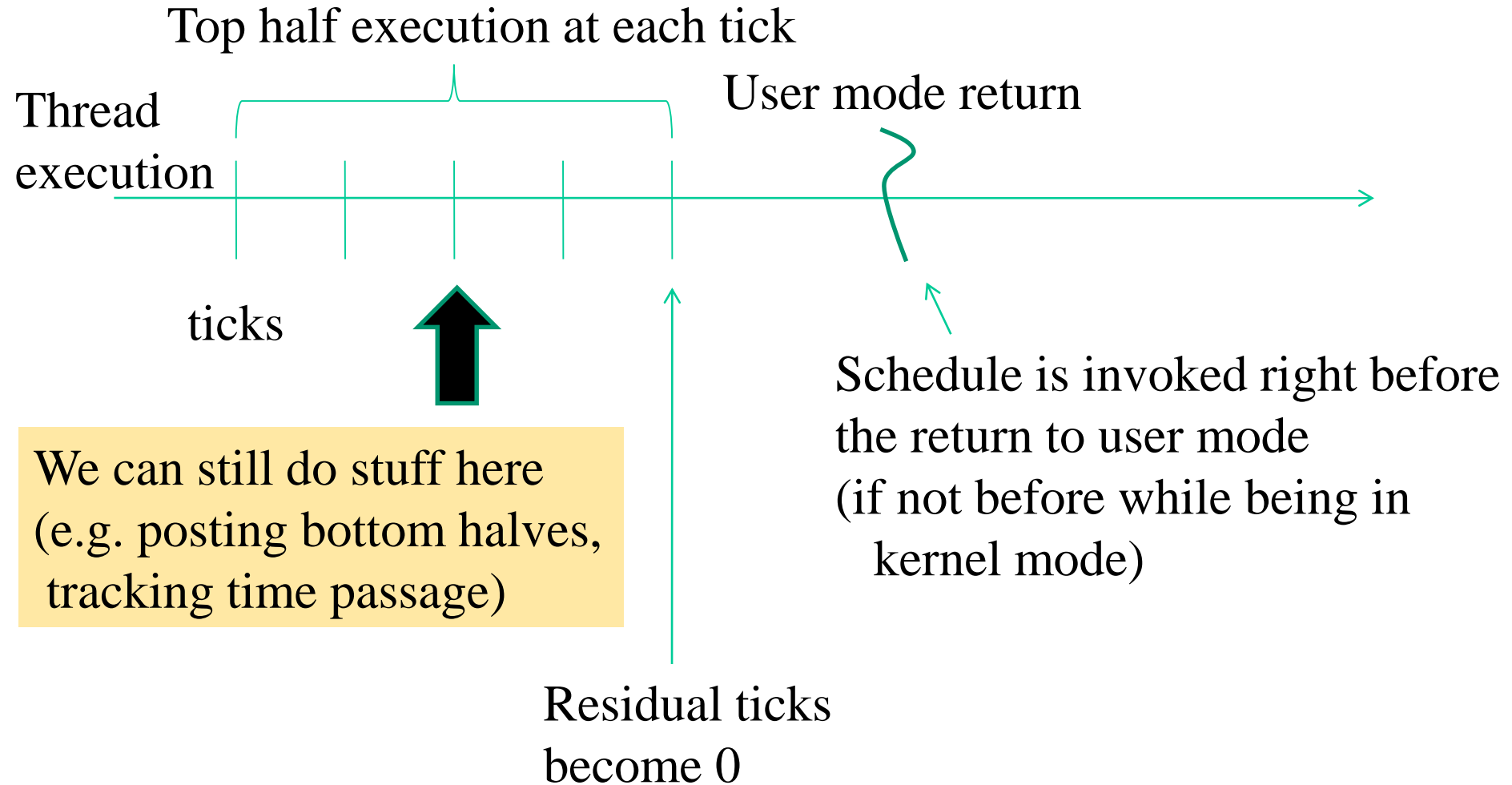
illustrated explanation



Interrupts vs passage of time vs CPU-scheduling

- The unsuitability of processing interrupts immediately (upon their asynchronous arrival) still stand 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
- What changes asynchronously is the condition that tells to the kernel software if we need to synchronously (at some point along execution in kernel mode) call the CPU scheduler
- 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 we loose the possibility to schedule bottom halves along time passage
- We also loose the possibility to control timings at fine grain, 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 old style approach for atomicity of kernel actions on single-core 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 ungauranteed 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

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
 - **Flags the task-queue** `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
 - **It checks whether the CPU scheduler needs to be activated,** and in the positive case flags the `need_resched` variable/bit within the TCB (Thread Control Block) of the current thread
- **NOTE AGAIN: time passage is not the unique means for preempting threads in LINUX, as we shall see**

Effects of raising `need_resched`

- Upon finalizing any kernel level work (e.g. a system call) the `need_resched` variable/bit within the TCB 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)

- definito in `linux/kernel/timer.c`

```
void do_timer(struct pt_regs *regs)
{
    (*(unsigned long *)&jiffies)++;
#ifdef 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)

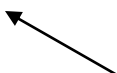
- definito in `linux/kernel/timer.c`

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

- Where the `run_timer_list()` function takes care of any timer-related action

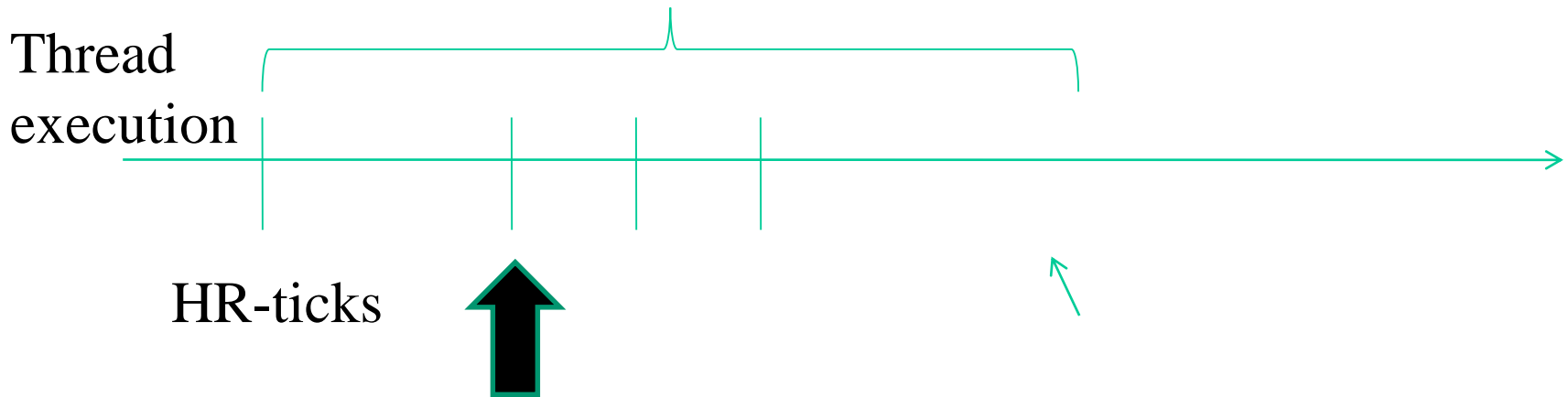
SoftIRQ based newer versions: the top half (kernel 3 example)

```
931 __visible void __irq_entry smp_apic_timer_interrupt(struct pt_regs *regs)
932 {
933     struct pt_regs *old_regs = set_irq_regs(regs);
934
935     /*
936      * NOTE! We'd better ACK the irq immediately,
937      * because timer handling can be slow.
938      *
939      * update_process_times() expects us to have done irq_enter().
940      * Besides, if we don't timer interrupts ignore the global
941      * interrupt lock, which is the WrongThing (tm) to do.
942      */
943     entering_ack_irq();
944     local_apic_timer_interrupt();
945     exiting_irq();
946
947     set_irq_regs(old_regs);
948 }
```

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

High Resolution (HR) Timers

They arrive at aperiodic (fine grain)
points along time



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 a `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

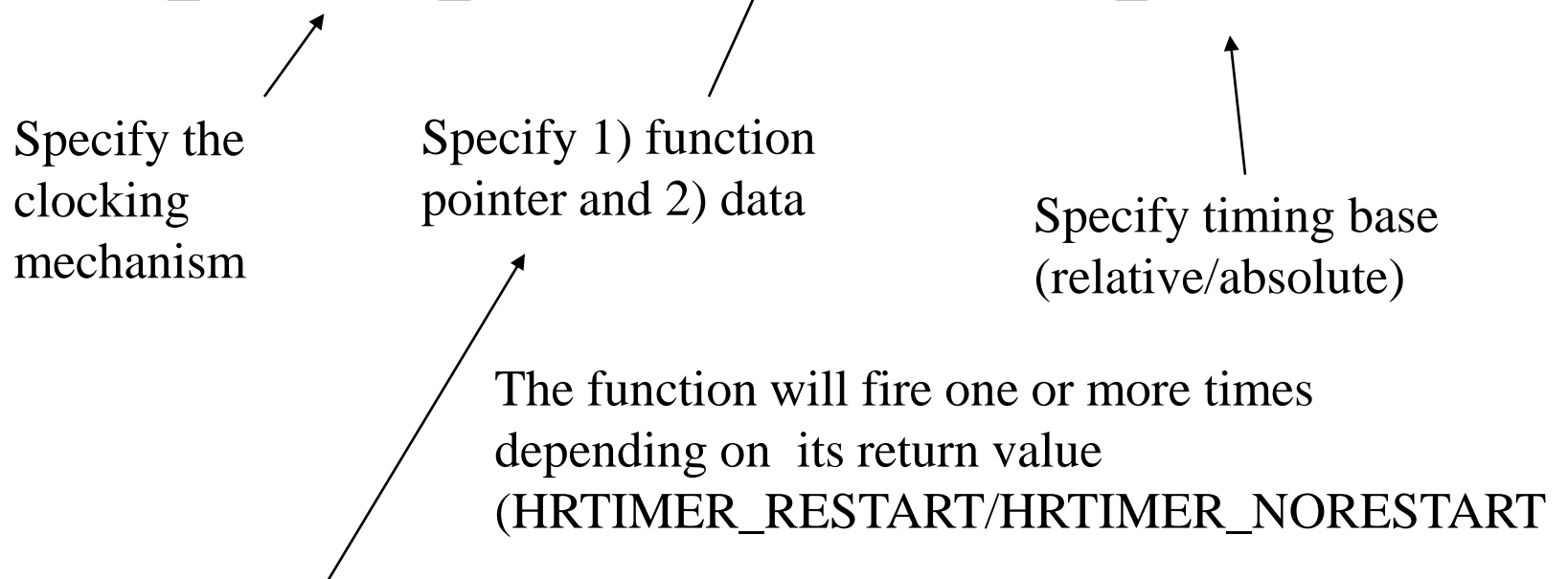
The HR-timers kernel interface

```
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



Specify 1) function
pointer and 2) data

Specify timing base
(relative/absolute)

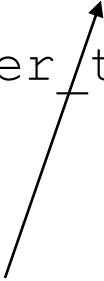
The function will fire one or more times
depending on its return value
(HRTIMER_RESTART/HRTIMER_NORESTART)

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


The HR-timers cancelation

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

```
int hrtimer_try_to_cancel(struct hrtimer *timer)
```



Waits of the target
function is already
running

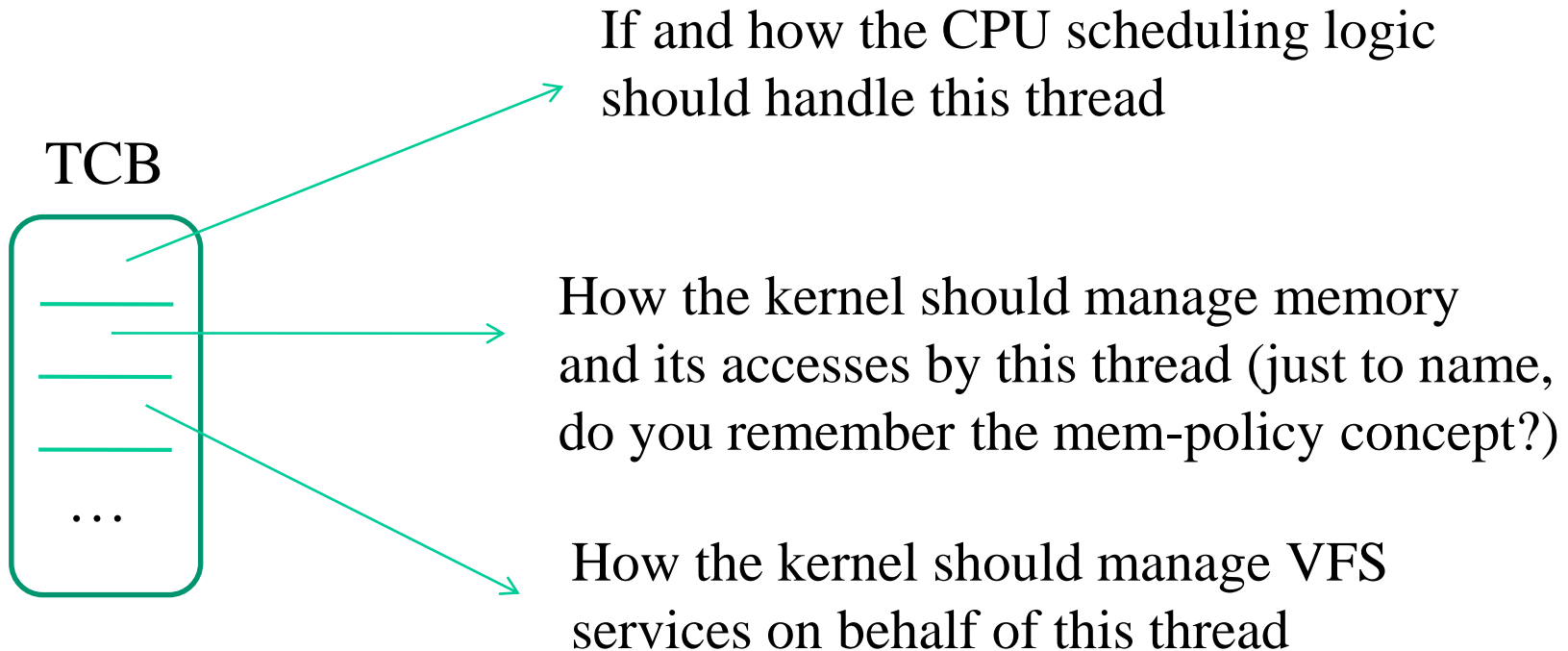


Does not wait of the target
function is already running

The role of TCBs in common operating systems

- A TCB is a data structure mostly keeping information related to
 - ✓ Schedulability and execution flow control (so scheduler specific information)
 - ✓ Linkage with subsystems external to the scheduling one (via linkage to metadata)
 - ✓ Multiple TBCs can link to the same external metadata (as for multiple threads within a same process)

An example



```
struct ... {  
...  
...  
}
```


The scheduling part: CPU-dispatchability

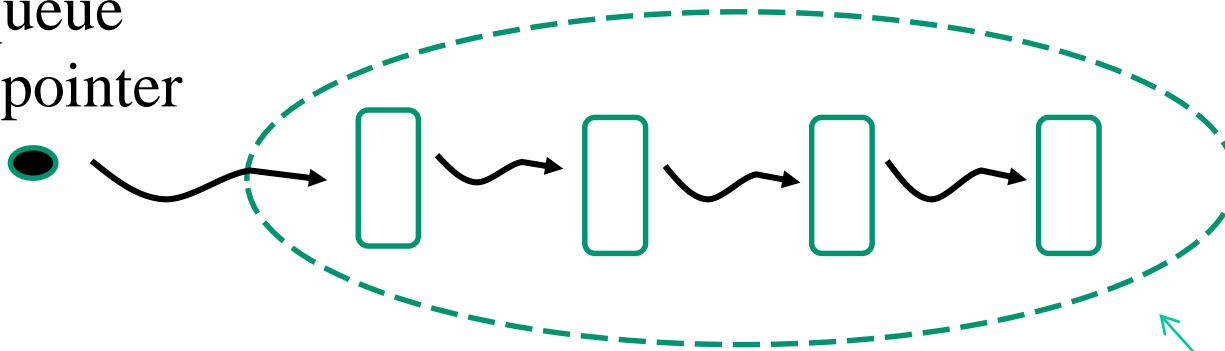
- The TCB tells at any time whether the thread can be CPU-dispatched
- But what is the real meaning of “CPU-dispatchability”??
- Its meaning is that the scheduler logic (so the corresponding block of code) can decide to pick the CPU-snapshot kept by the TBC and install it on CPU
- CPU-schedulability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

The scheduling part: run/wait queues

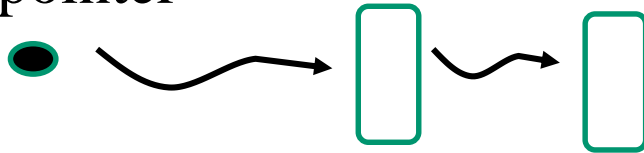
- A thread is CPU-schedulable only if its TCB is included into a specific data structure (generally a list)
- This is typically referred to as the **runqueue**
- The scheduler logic selects threads based on ``scans'' of the runqueue
- All the non CPU-schedulable threads are kept on aside data structures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

A scheme

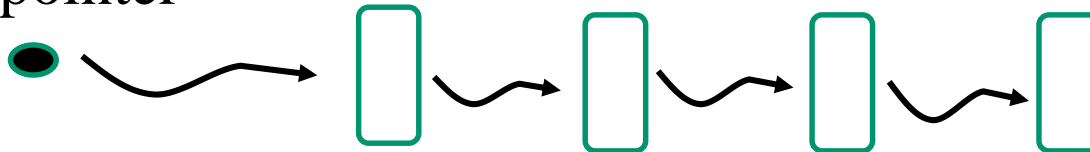
Runqueue
head pointer



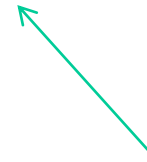
Waitqueue A
head pointer



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 **kernel level sleep/wait services**
- 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 is removed from the runqueue by the scheduler logic before the actual selection of the next thread to run is performed
 - ✓ The TCB of the invoking thread still stands on the runqueue 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 invoking thread from the wait queue

The timeline

sleep/wait API invocation by thread T

← Change status within
TCB to “sleep”

Scheduler logic invocation

Can really sleep?

Unlink TCB
from runqueue →

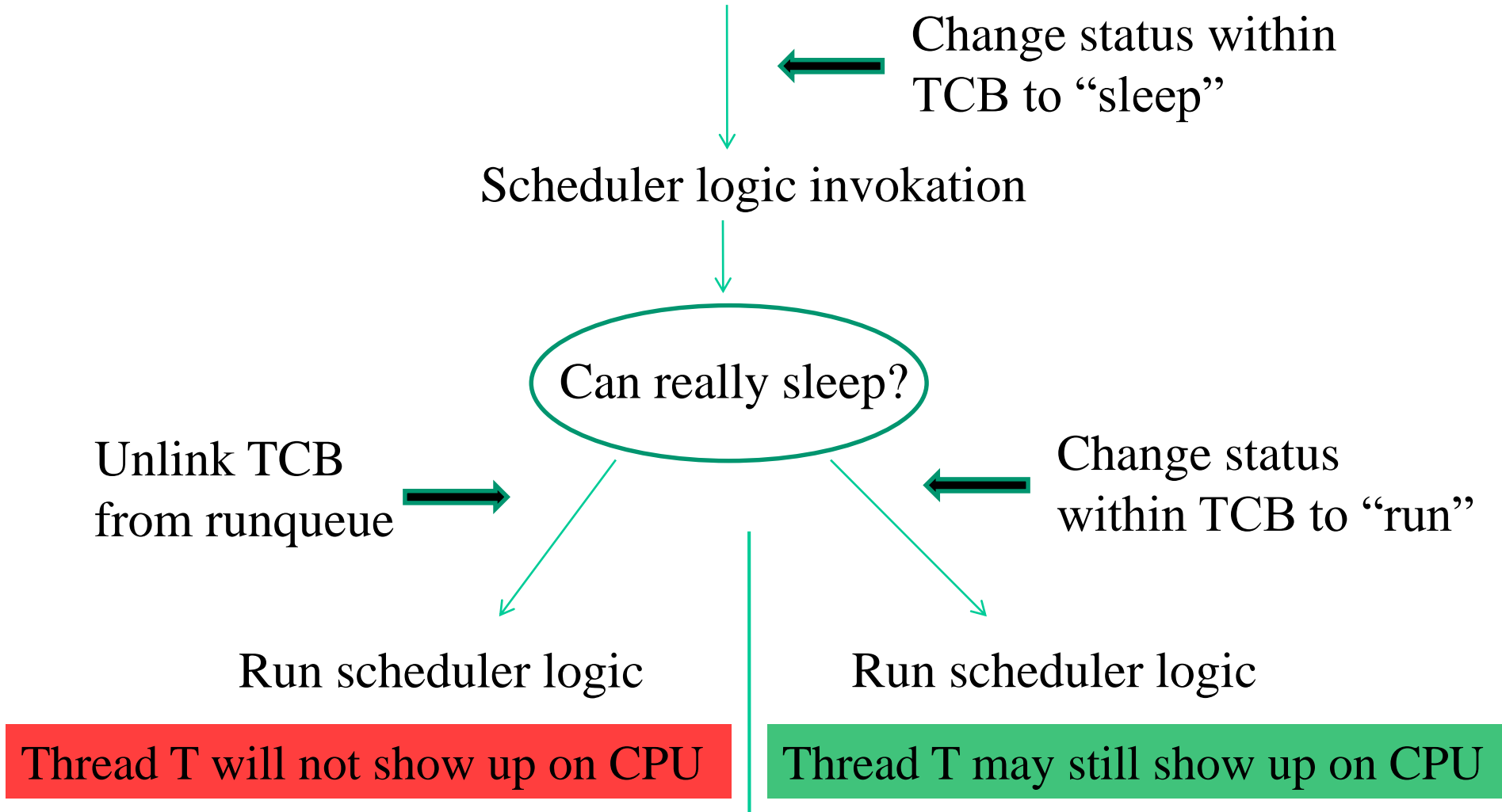
← Change status
within TCB to “run”

Run scheduler logic

Run scheduler logic

Thread T will not show up on CPU

Thread T may still show up on CPU



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 switched off the CPU thread
- 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 unique kernel level ``private'' memory are each thread has
- This is the kernel level stack
- In most kernel implementations we say that we switch the context when we install a value on the stack pointer

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*/

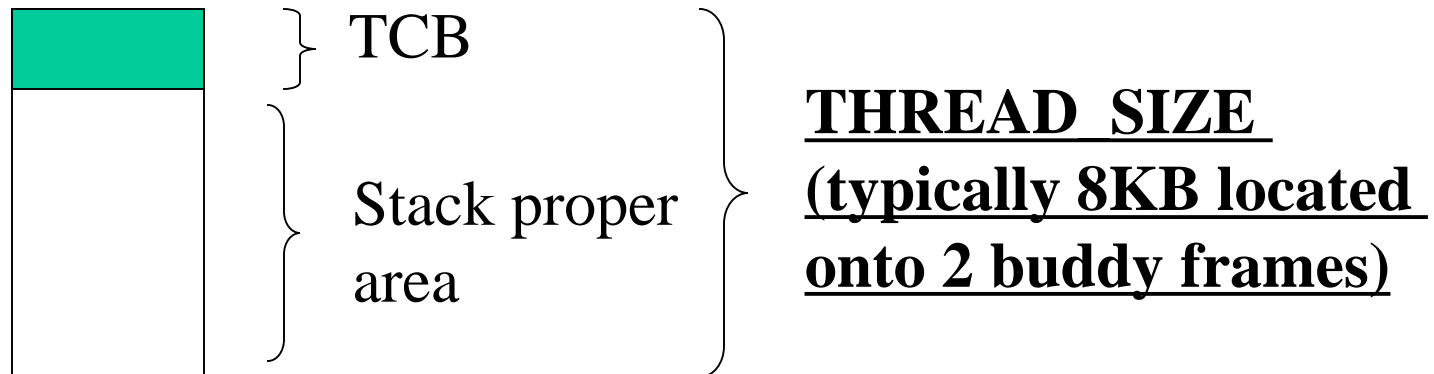
synchronous and
asynchronous
modifications

More modern kernel versions

- A few info is compacted into bitmasks
 - ✓ e.g. `need_resched` has become a single 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 address of the IDLE PROCESS TCB



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 later 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

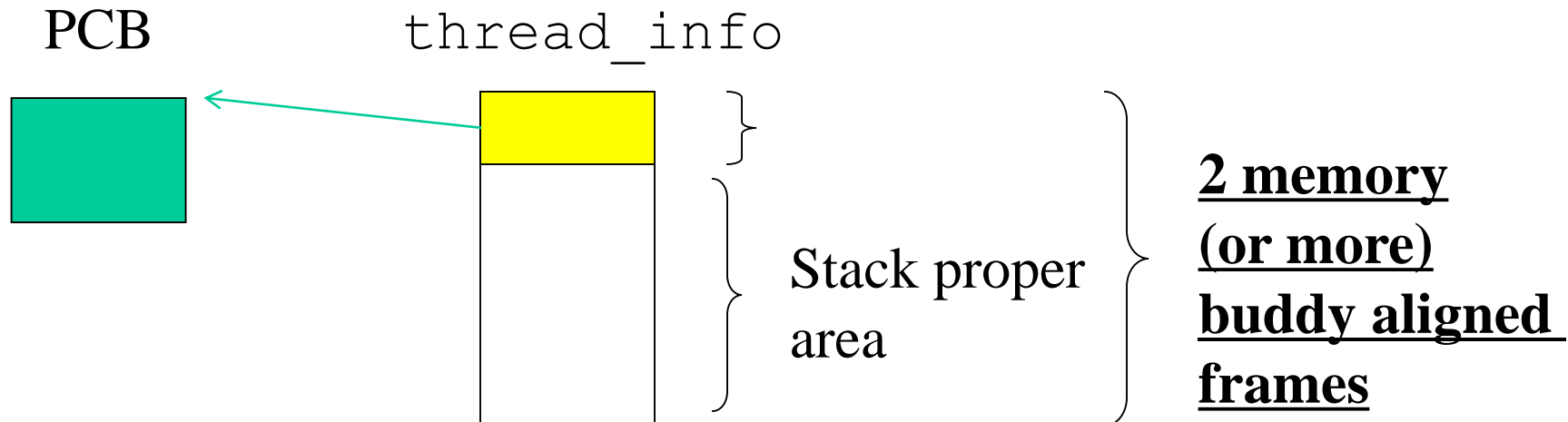
Actual declaration of the kernel level stack data structure

Kernel 2.4.37 example

```
522 union task_union {  
523     struct task_struct task;  
524     unsigned long stack[INIT_TASK_SIZE/sizeof(long)];  
525 };
```

PCB allocation: since kernel 2.6 up to 4.8

- The memory area for the PCB 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 PCB
- This allows for improved memory usage with large PCBs



Actual declaration of the kernel level **thread_info** data structure

Kernel 3.19 example

```
26 struct thread_info {
27     struct task_struct *task;          /* main task structure */
28     struct exec_domain *exec_domain;    /* execution domain */
29     __u32 flags;                        /* low level flags */
30     __u32 status;                       /* thread synchronous flags */
31     __u32 cpu;                          /* current CPU */
32     int saved_preempt_count;
33     mm_segment_t addr_limit;
34     struct restart_block restart_block;
35     void __user *sysenter_return;
36     unsigned int sig_on_uaccess_error:1;
37     unsigned int uaccess_err:1;        /* uaccess failed */
38 };
```


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 PCB 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 per-CPU memory
- 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)