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

Topics:

1. Advanced task management schemes
2. Binding to the Linux architecture

Tasks vs processes

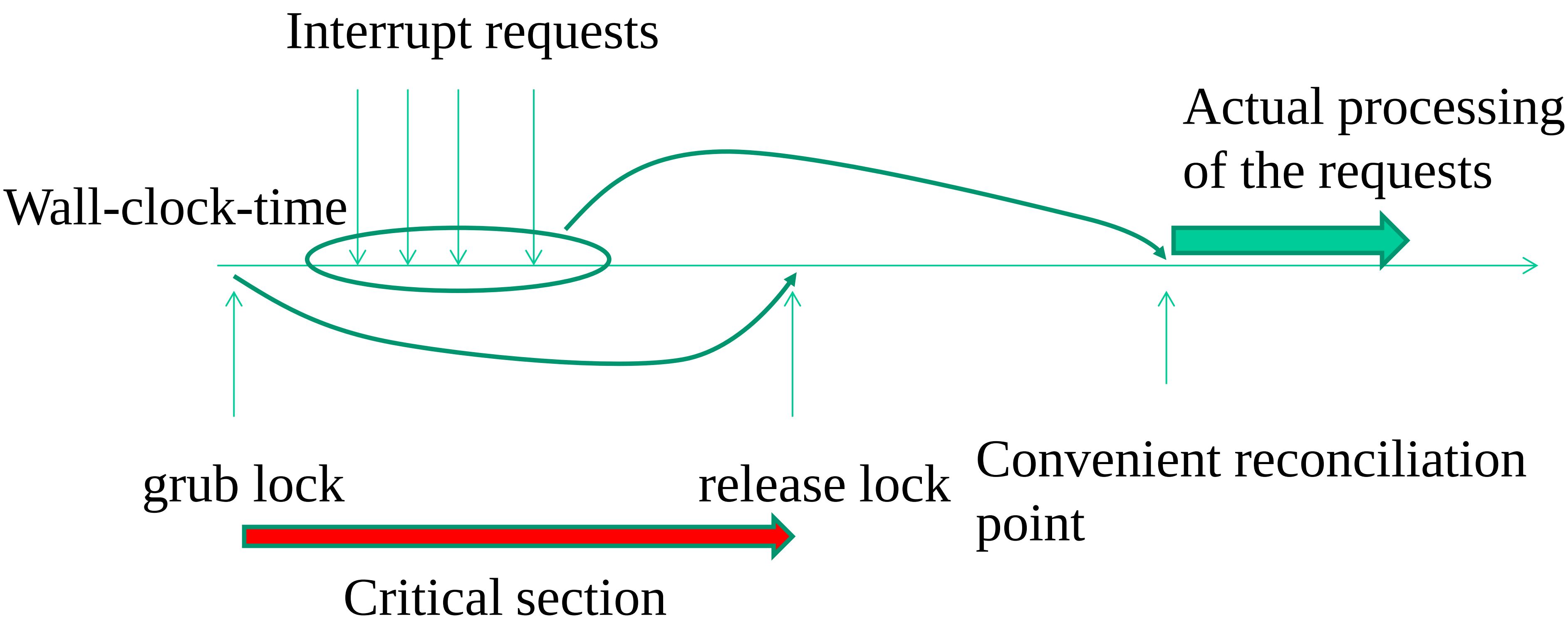
- Types of traces
  - User mode process/thread
  - Kernel mode process/thread
  - Interrupt management
- Non-determinism
  - Due to nesting of user/kernel mode traces and interrupt management traces
- Performance
  - Non-determinism may give rise to inefficiency whenever the evolution of the traces is 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
- 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

A schematization



Reconciliation points

Guarantees

- "Eventually"

Conventional support

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

The historical concept: top/bottom half processing

- The management of **<u>task associated with the interrupts</u>** 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 necessary to allow later finalization of the whole interrupt management
- The **<u>top-half code portion is typically (but not manadatorily) handled according to a non-interruptible scheme</u>**
- The finalization of the work takes place via the bottom-half level
- The top-half takes care of **<u>scheduling the botom-half task by</u>** queuing a record into a proper data structure

- The difference between top-half and bottom-half comes out because of
    - ✓ the need to manage events in a timely manner,
    - ✓ while avoiding to lock resources right upon the event occurrence
- Otherwise, we may incur the risk of delaying critical actions (**e.g. spinlock-release**) interrupted due to the event occurrence

One example: sockets

no top/bottom half

interrupt from network device

packet extraction

IP level

TCP/UDP level

VFS Level

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

top/bottom half

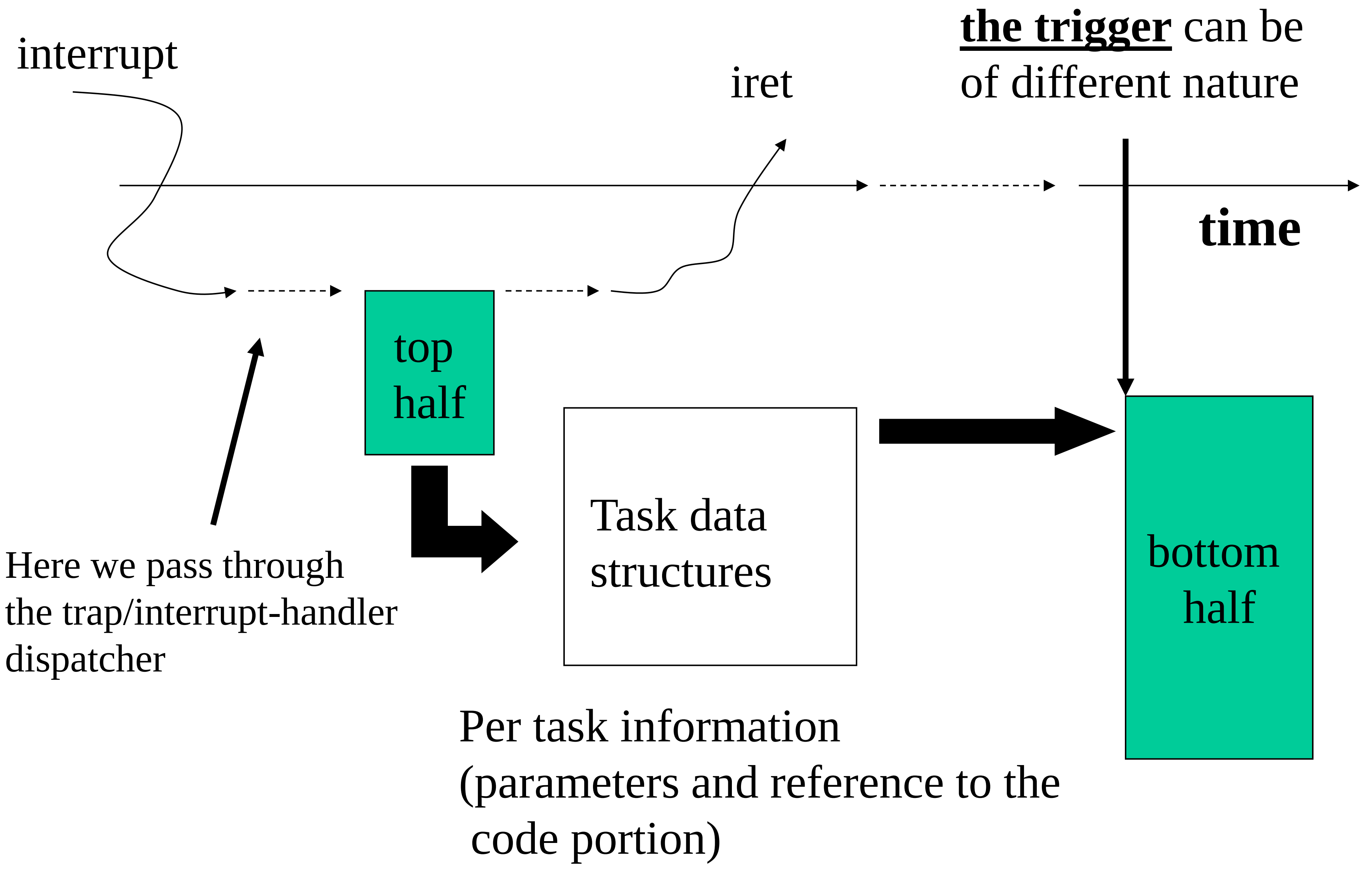
interrupt from network device

packet extraction

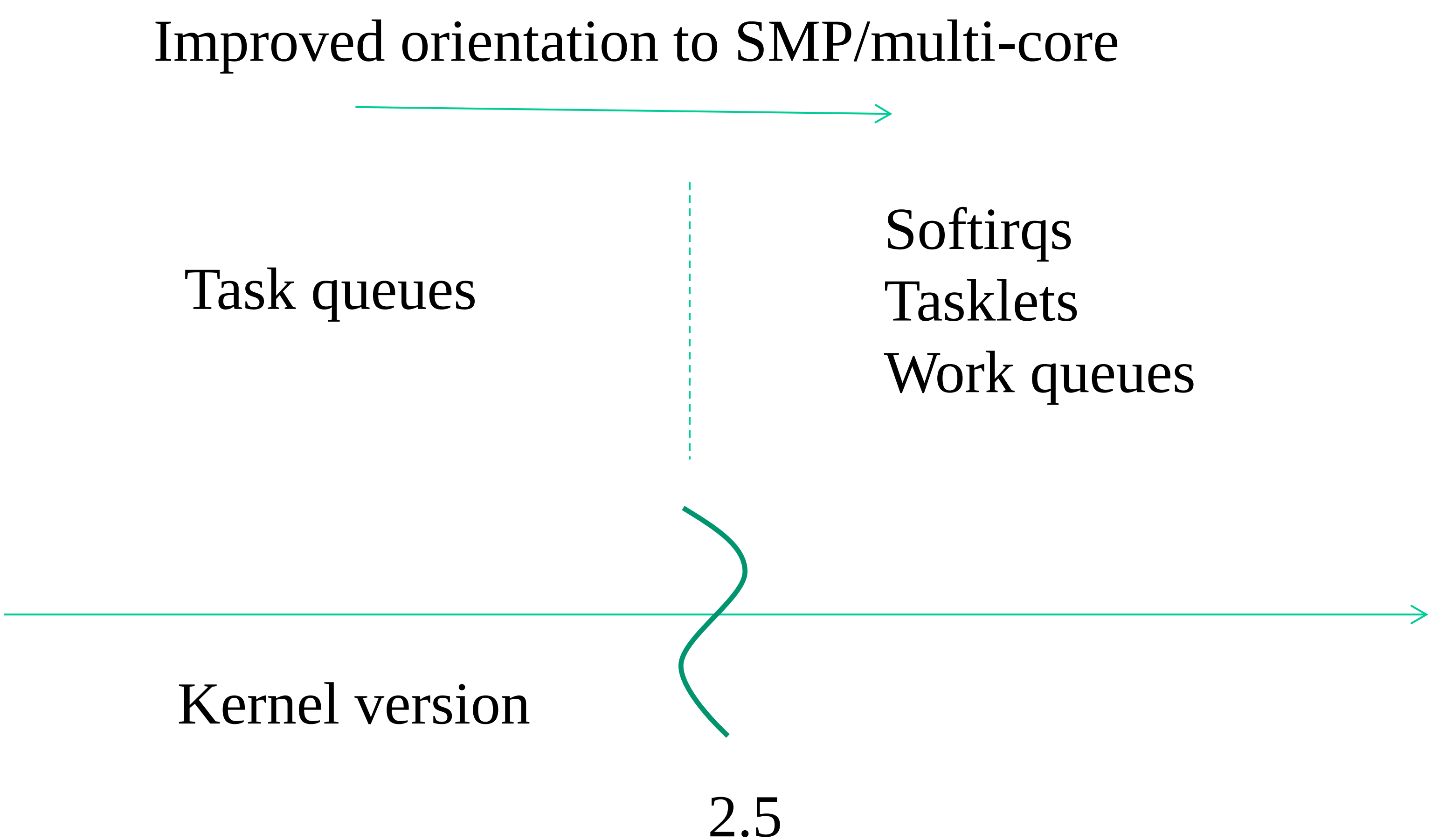
Task queuing

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

The historical concept: bottom-half queues



Historical evolution in Linux



Let's start from task queues

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

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

    (task to be executed upon timer-interrupt)
  - `tq_scheduler`

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

Task queues data structures

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

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

Task management API

- The queuing function has prototype `int queue_task(struct tq_struct *task, task_queue *list)`, where `list` is the address of the target task-queue structure
- This function is used to only register the task, not to execute it
- The task flushing (execution) function for all the tasks currently kept by a task queue is `void run_task_queue(task_queue *list)`
- When invoked, unlinking and actual execution of the tasks takes place
- For the `tq_schedule` task-queue there exists a proper queuing function offered by the kernel with prototype `int schedule_task(struct tq_struct *task)`
- **<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)

NOTE!!!!!

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

Bottom-half occurrences with task queues

Timely flushing of the bottom halves requires

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

The Linux kernel invokes **do_bottom_half()** (defined in kernel/softirq.c)

- within schedule()
- from ret_from_sys_call()

Be careful: bottom half execution context

- Even though bottom half tasks can be executed in process context, the actual context for the thread running them should look like “interrupt”
- No blocking service invocation in any bottom half function

Let’s go for more scalability

- Original task queues limitations
    - ✓ Single thread execution of the tasks
    - ✓ Not suited for maximizing locality
    - ✓ Not suited for heavy interrupt load
- The newer approach
    - ✓ Multithread execution of bottom half tasks
    - ✓ Binding of task execution to CPU-cores

Tasklets

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

- tasklet enabling/disabling functions are
  `tasklet_enable(struct tasklet_struct *tasklet)`
  `tasklet_disable(struct tasklet_struct *tasklet)`
- the function scheduling the tasklet is
  `void tasklet_schedule(struct tasklet_struct *tasklet)`
- for any tasklet schedule, it may be needed to reinitialize the parameter `data`
- **NOTE:**
  - Each tasklet represents a single task, it is not equivalent to a task-queue
  - Subsequent reschedule of a same tasklet may result in a single execution, depending on whether the tasklet was already flushed or not (hence there is no queuing concept)

Tasklets' execution

- 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, it will not be moved to another CPU if it's still pending (this is instead allowed for softirqs)
- Tasklets have schedule level similar to the one of `tq_schedule`
- The main difference is that the thread actual context should be an "interrupt-context" – thus with no-sleep phases within the tasklet (an issue already pointed to)

Finally: work queues

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

Work queues basic interface (default queues)

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

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

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

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

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

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

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

This eliminates the queue

```
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

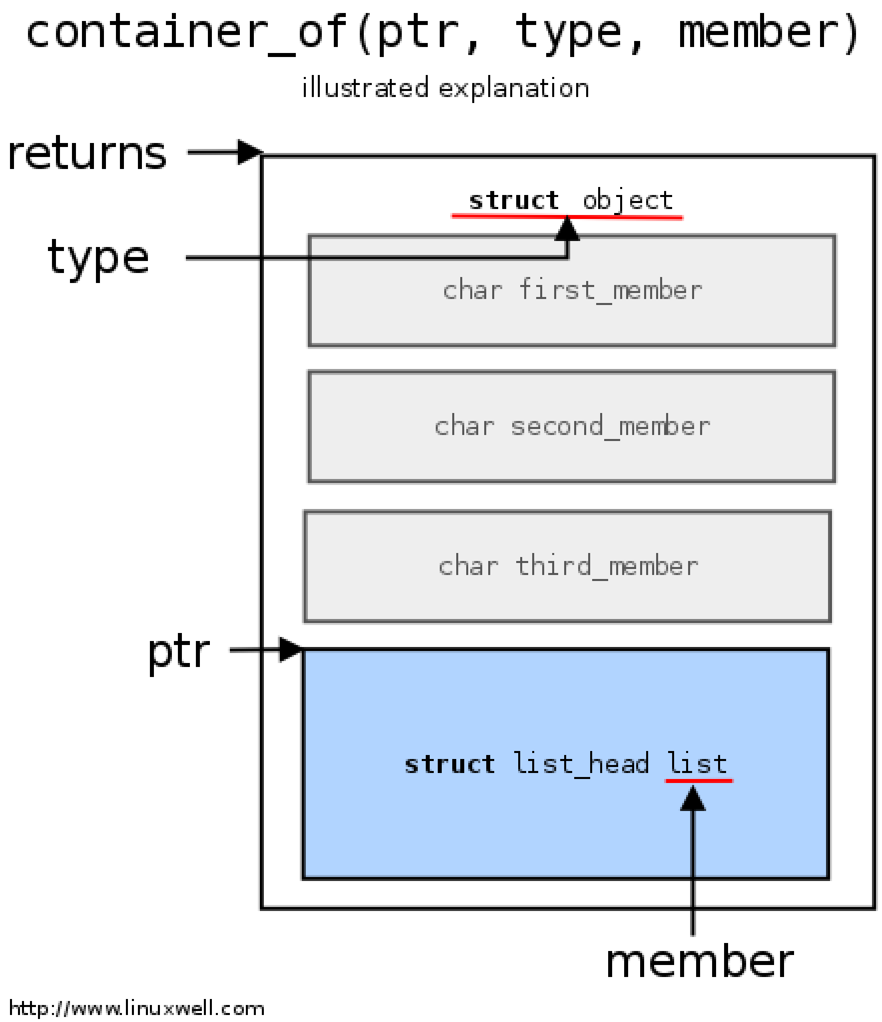
```
int cancel_delayed_work(struct work_struct *work);
```

This cancels a pending job

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

This runs any job

Managing dynamic memory with (not only) work queues



Timer interrupt

- It is handled according to the top/bottom half paradigm
- 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
  - **It checks whether the CPU scheduler needs to be activated**, and in the positive case flags `need_resched` within the PCB of the current process
- The bottom half is buffered within the `tq_timer` queue and reschedules itself upon execution (old style)

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

Timer-interrupt top-half module (old style)

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

```
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 (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

Linux Timer IRQ ICA

```
Linux Timer IRQ
IRQ 0 [Timer]
 |
\|/
|IRQ0x00_interrupt        //   wrapper IRQ handler
   |SAVE_ALL              ---
      |do_IRQ                |   wrapper routines
         |handle_IRQ_event  ---
            |handler() -> timer_interrupt  // registered IRQ 0 handler
               |do_timer_interrupt
                  |do_timer
                     |jiffies++;
                     |update_process_times
                     |if (--counter <= 0) { // if time slice ended then
                        |counter = 0;        //   reset counter
                        |need_resched = 1;   //   prepare to reschedule
                     |}
         |do_softirq
         |while (need_resched) { // if necessary
            |schedule             //   reschedule
            |handle_softirq
         |}
   |RESTORE_ALL
```

Where the functions are located in

- IRQ0x00_interrupt, SAVE_ALL [include/asm/hw_irq.h]
- do_IRQ, handle_IRQ_event [arch/i386/kernel/irq.c]
- timer_interrupt, do_timer_interrupt [arch/i386/kernel/time.c]
- do_timer, update_process_times [kernel/timer.c]
- do_softirq [kernel/soft_irq.c]
- RESTORE_ALL, while loop [arch/i386/kernel/entry.S]

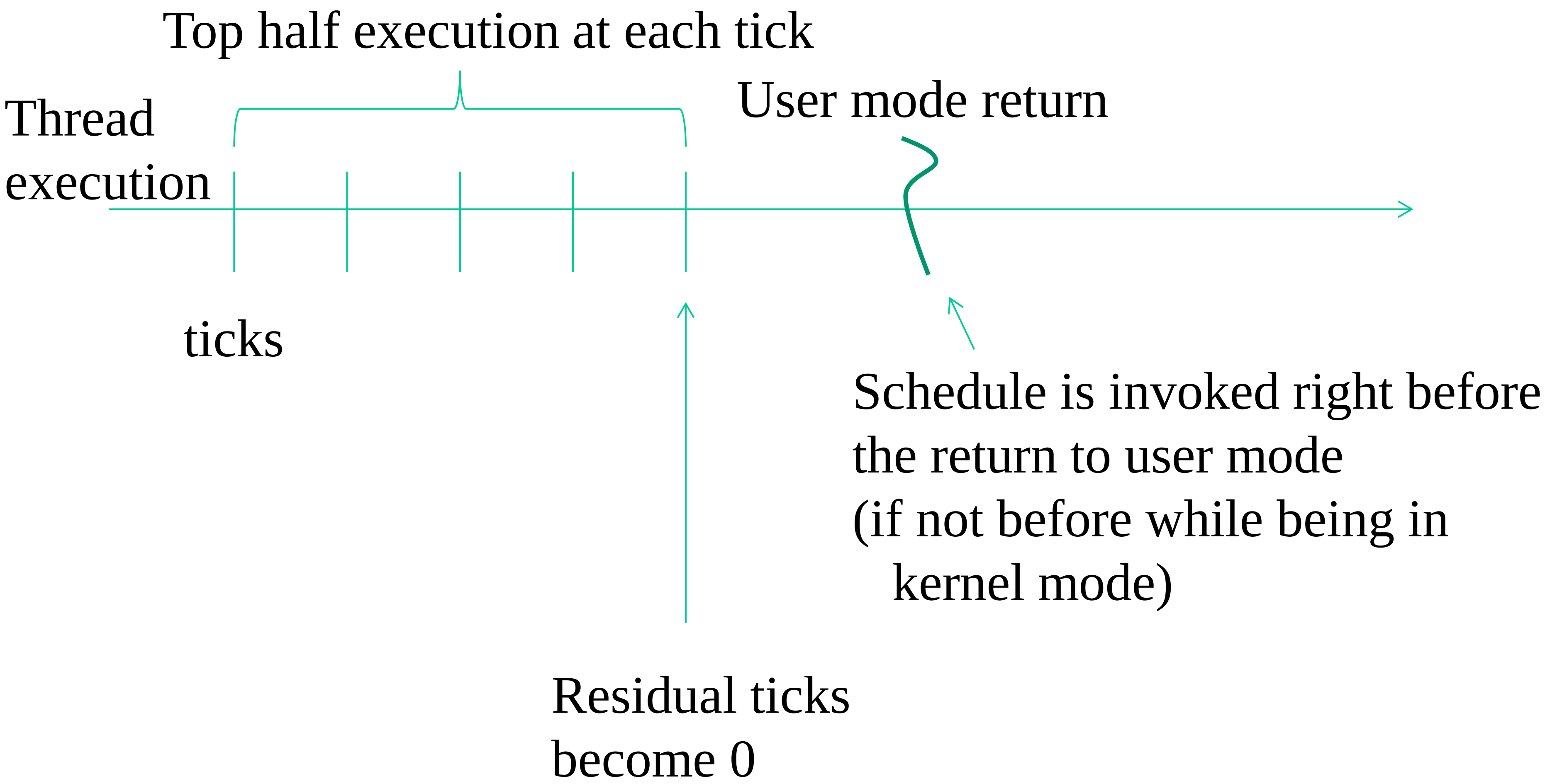
Kernel 3 example

```
__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);
}
```

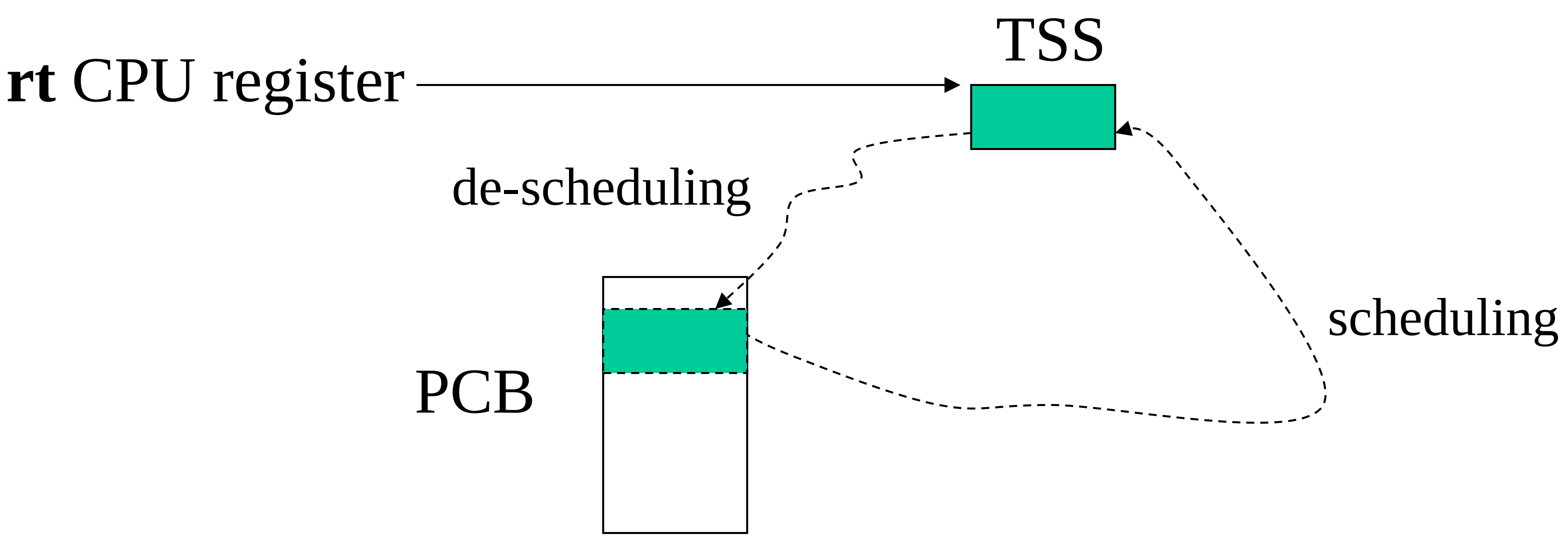
A final scheme for timer interrupts



Task State Segment

- The Kernel keeps some special information which, for i386/x86-64 machines, is called TSS (task state segment)
- This information includes the value of the stack pointers to the base of the kernel stack for the current process/thread
- The virtual memory buffer keeping TSS information is pointed by a proper CPU register (**tr** – task register)
- The buffer is also accessible via `struct tss_struct *init_tss,` where the pointed structure is defined in `include/asm-i386/processor.h`

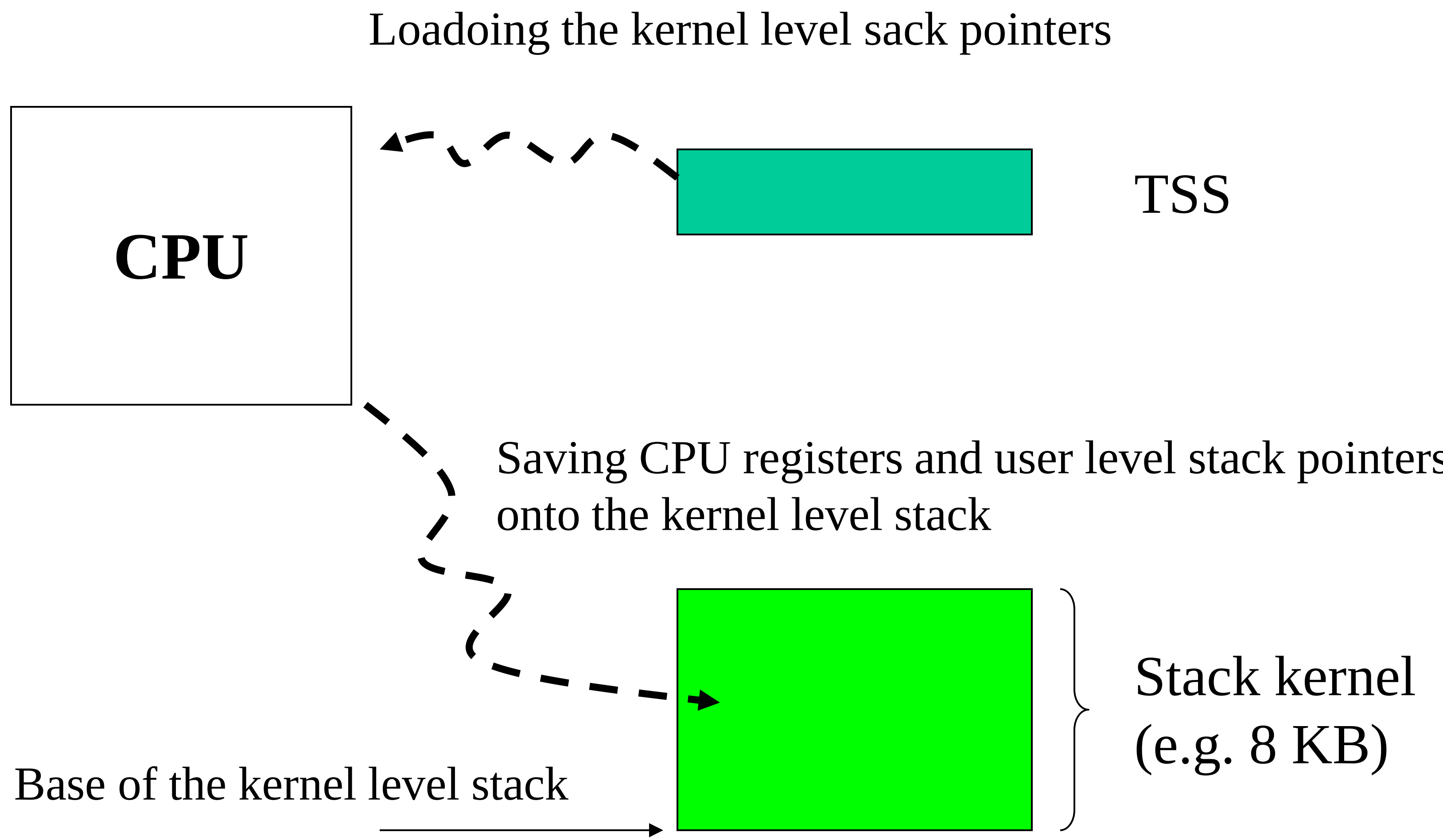
- The `schedule()` function saves the TSS info into the PCB/TCB upon any context switch
- This allows keeping track of the kernel level stack-base for the corresponding thread
- The kernel level stack for each process/thread consists of THREAD_SIZE pages kept into kernel level segments (typically **8 KB or more**), with the corresponding physical pages contiguous and aligned to the buddy system scheme (see `_get_free_pages()`)



TSS usage

- TSS information is exploited by the i386/x86-64 microcode while managing traps and interrupts leading to mode change
- It I also exploited by syscall-dispatching software
- Particularly, the TSS content is used to determine the memory location of the kernel level stack for the thread killed by the trap/interrupt
- The kernel level stack is used for logging user-level stack pointers and other CPU registers (e.g. EFLAGS)
- The TSS stack-pointers are loaded by the microcode onto the corresponding CPU registers upon traps/interrupts, hence we get a stack switch
- No execution relevant information gets lost since upon the mode-change towards kernel level execution the user-level stack positioning information is saved into the kernel level stack

The scheme


Loadoing the kernel level sack pointers
CPU
TSS
Saving CPU registers and user level stack pointers
onto the kernel level stack
Stack kernel
(e.g. 8 KB)
Base of the kernel level stack

Process control blocks

- The structure of Linux process control blocks is defined in `include/linux/sched.h` as `struct task_struct`
- The main fields 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`** */*per lo scheduling*/*

The `mm` field

- The `mm` of the process control block points to a memory area structured as `mm_struct` which his defined in `include/linux/sched.h`
- This area keeps information used for memory management purposes for the specific process, such as
  - Virtual address of the page table (`pgd` field)
  - A pointer to a list of records structured as `vm_area_struct` as defined in `include/linux/sched.h` (`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

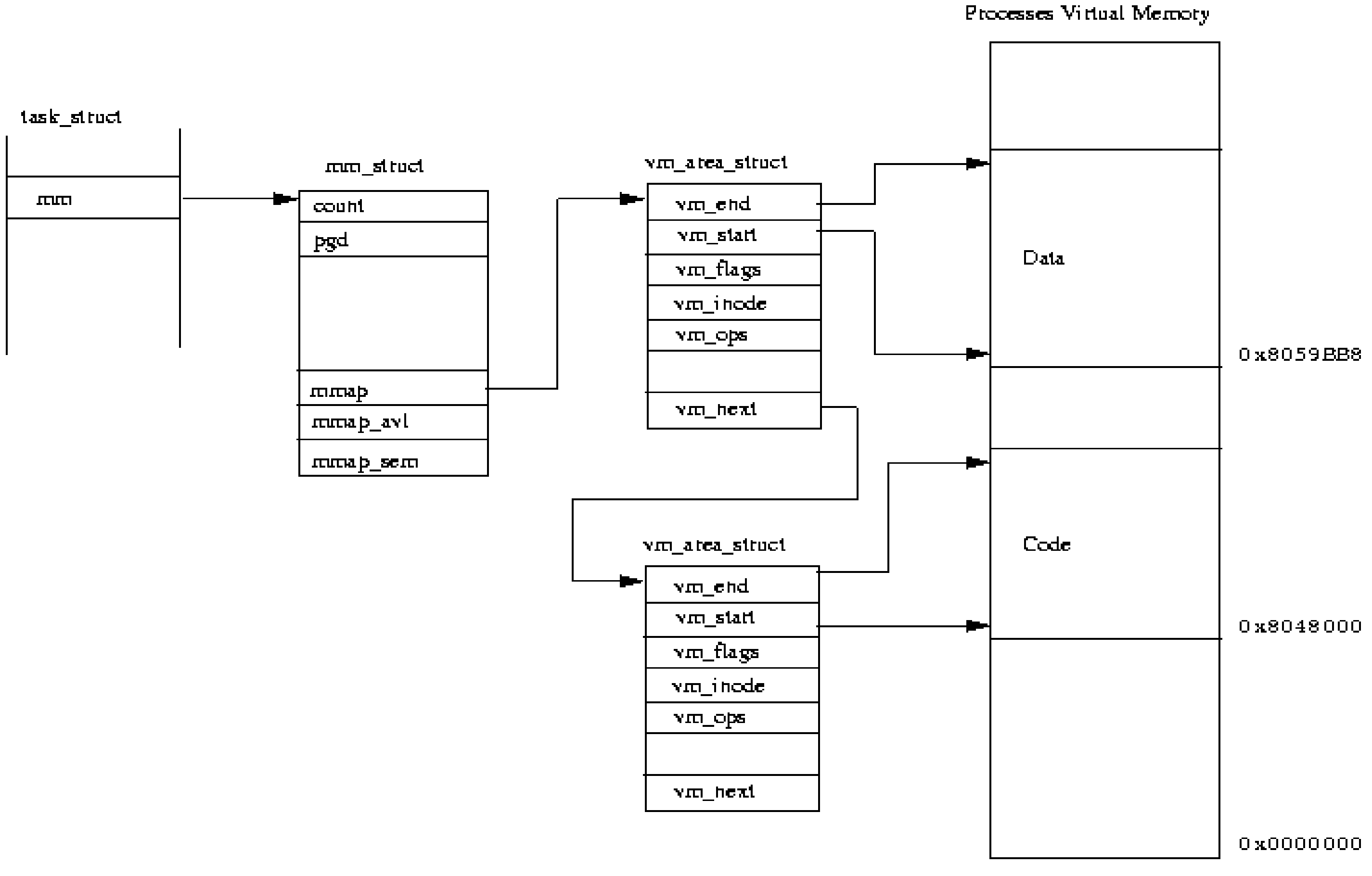
```
struct vm_area_struct {
  struct mm_struct * vm_mm;/* The address space we belong to. */
  unsigned long vm_start;  /* Our start address within vm_mm. */
  unsigned long vm_end;    /* The first byte after our end address
                                     within vm_mm. */
  struct vm_area_struct *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

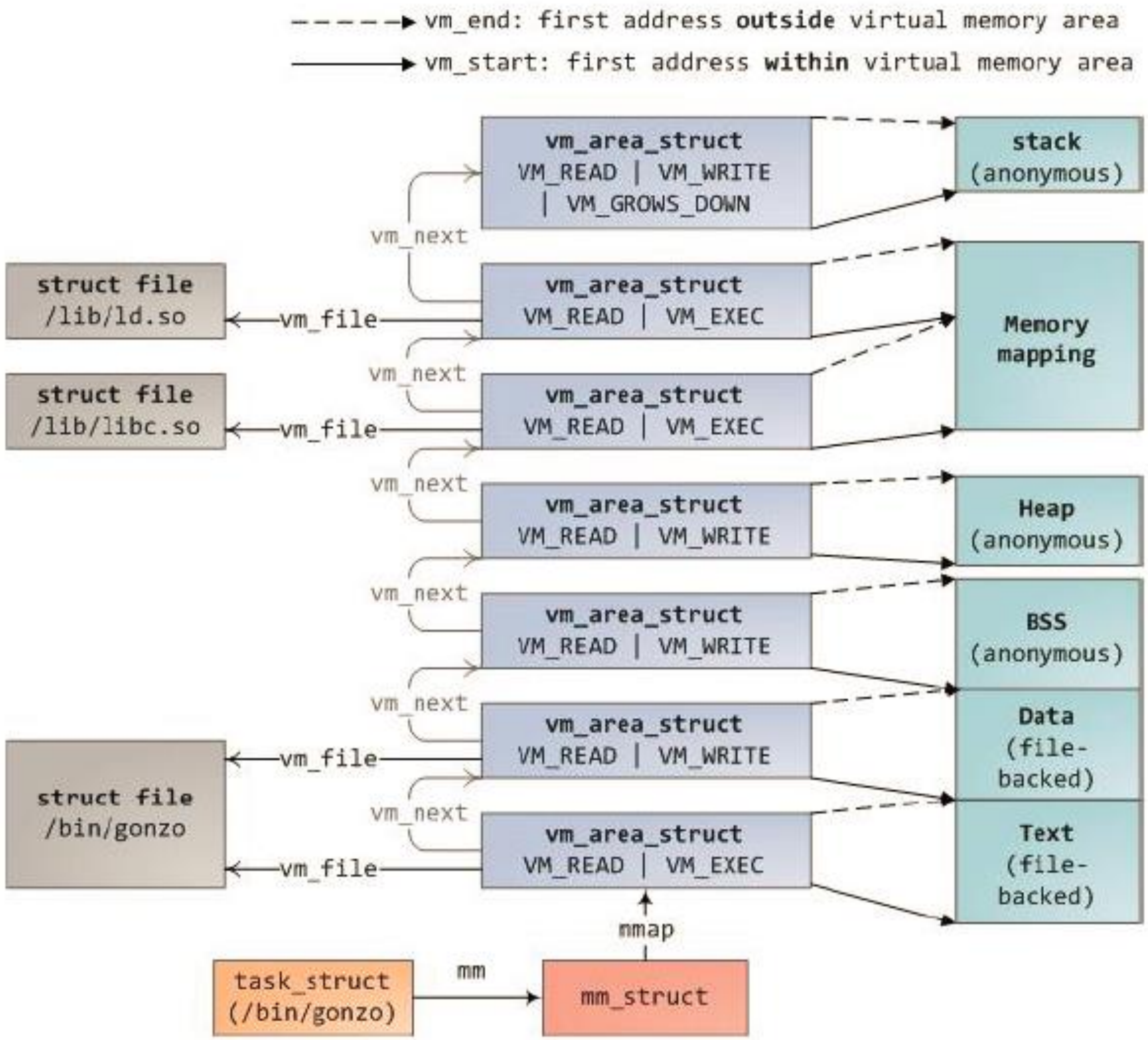
  `struct page * (*nopage)(struct vm_area_struct * area, unsigned long address, int unused)`

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



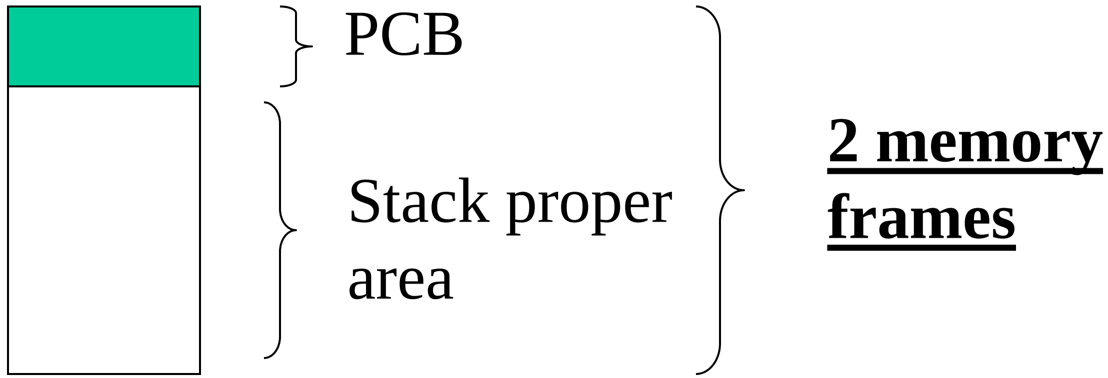
vm_end: first address outside virtual memory area
vm_start: first address within virtual memory area
vm_area_struct
VM_READ | VM_WRITE
| VM_GROWS_DOWN
stack
(anonymous)
vm_next
struct file
/lib/ld.so
vm_file
vm_area_struct
VM_READ | VM_EXEC
Memory
mapping
vm_next
struct file
/lib/libc.so
vm_file
vm_area_struct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

IDLE PROCESS

- The variable `init_task` of type `struct task_struct` declared in `arch/i386/kernel/init_task.c` corresponds to the PCB of the IDLE PROCESS (the one with PID 0)
- Data structure values for this process are initialized at compile time as specified in `arch/i386/kernel/init_task.c`
- Actually, the `vm_area_struct` list for this process looks empty (since it leaves in kernel mode only)
- Particularly, this process executes within the following set of functions
  - ➢ `start_kernel()` (in `init/main.c`)
    - ✓ `rest_init()` (in `init/main.c`)
      - ✓ `cpu_idle()` (in `arch/i386/kernel/process.c` this is a busy loop around a power management function, if any)

PCB allocation: the case up to kernel 2.6

- PCBs are allocated dynamically, whenever requested
- The memory area for the PCB 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`
- This address is initially loaded into stack/base pointers at boot time (this is done via the ASM routine `arch/i386/kernel/head.S`)



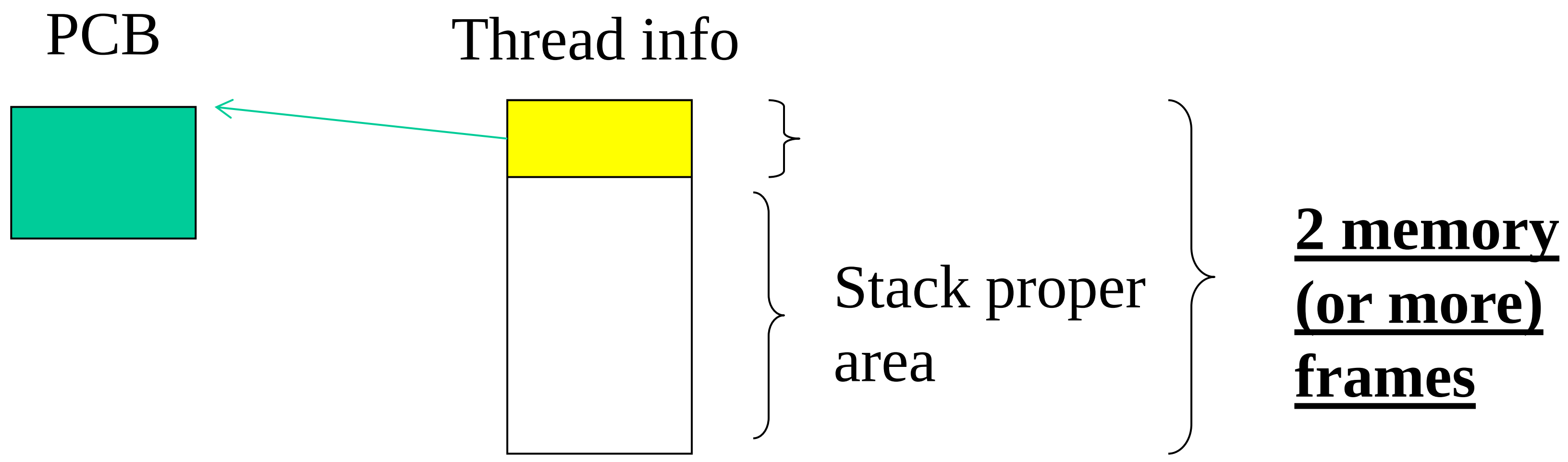
Actual declaration of the kernel level stack data structure

Kernel 2.4.37 example

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

PCB allocation: since kernel 2.6

- 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 stack 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 */
};
```

The `current` MACRO

- The macro `current` is defined in `include/asm-i386/current.h` (or x86 versions)
- It returns the memory address of the PCB 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, by exploiting that the stack is aligned to the couple 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`

Virtually mapped stacks

- Typically we only nee logical memory contiguousness for a stack area
- On the other hand stack overflow is a serious problem for kernel corruption
- 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
- This allows for tracking the overstepping of the stack boundaries via memory faults
- On the other hand this requires changing the mechanism for managing the stack by the fault-handler
- Also, the `thread_info` structure needs t be moved away form its current position

IDLE PROCESS cycle (classical style)

```
void cpu_idle (void)
{
        /* endless idle loop with no priority at all */
        init_idle();
        current->nice = 20;
        current->counter = -100;
        while (1) {
                void (*idle)(void) = pm_idle;
                if (!idle)
                        idle = default_idle;
                while (!current->need_resched)
                        idle();
                schedule();
                check_pgt_cache();
        }
}
```

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 PCB 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 PCBs within the list are also kept via the static variable `runqueue_head` of type `struct list_head{struct list_head *prev,*next;}`

- The PCB list gets scanned by the `schedule()` function whenever we need to determine the next process/thread to be dispatched

Wait queues (2.4 style)

- PCBs can be arranged into lists called wait-queues
- PCBs 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 PCB 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

- `void wake_up(wait_queue_head_t *q)`
  Reinstalls onto the ready-to-run queue all the PCBs currently kept by the wait queue `q`

- `void wake_up_interruptible(wait_queue_head_t *q)`
  Reinstalls onto the ready-to-run queue the PCBs currently kept by the wait queue `q`, which were queued as “interruptible”

- `wake_up_process(struct task_struct * p)`
  Reinstalls onto the ready-to-run queue the process whose PCB s pointed by `p`

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 – just one example

wait_event_interruptible

Prev **Wait queues and Wake events** Next

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.

Thread states

- The `state` field within the PCB keeps track of the current state of the process/thread

- The set of possible values are defined as follows in `inlude/linux/sched.h`
  - `#define TASK_RUNNING 0`
  - `#define TASK_INTERRUPTIBLE 1`
  - `#define TASK_UNINTERRUPTIBLE 2`
  - `#define TASK_ZOMBIE 4`
  - `#define TASK_STOPPED 8`

- All the PCBs 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

Accessing PCBs

- PCBs are linked in various lists with hash access supported via the below fields within the PCB structure

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

- There exists a hashing structure defined as below in `include/linux/sched.h`

```
#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;
}
```

- Newer kernel versions (e.g. >= 2.6) support

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

- This is based on the notion of virtual pid
- The behavior is the same as the traditional API in case no actual virtual pids are used

Helps

```
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);
```

Helps

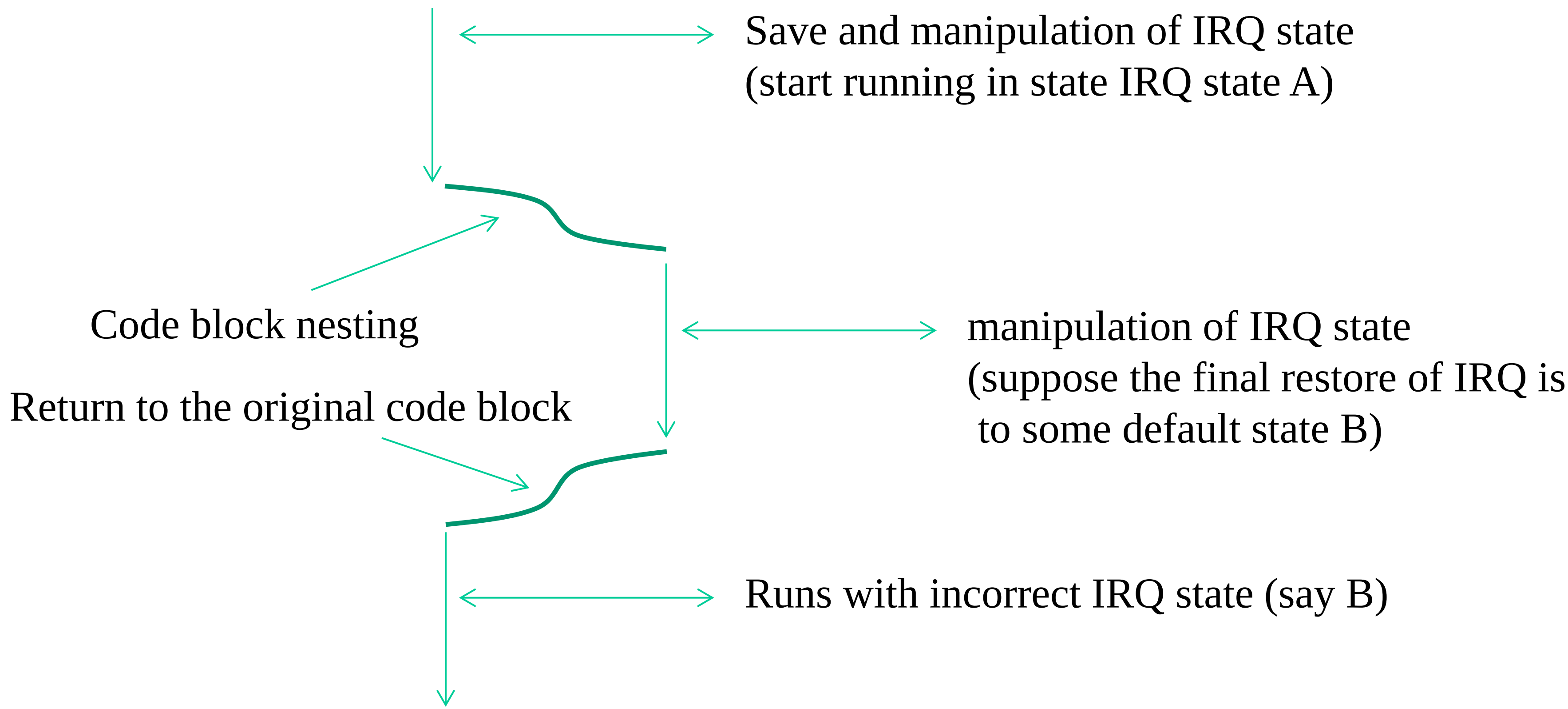
```
#include <linux/spinlock.h>

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 scheduler

- CPU scheduling is implemented within the `void schedule(void)` function, which is defined in `kernel/sched.c`
- Generally speaking, this function offers 3 different CPU scheduling policies, associated with the following macros defined in `include/linux/sched.h`

```
#define SCHED_OTHER        0
#define SCHED_FIFO         1
#define SCHED_RR           2
```

- The `SCHED_OTHER` policy corresponds to the classical multi-level with feedback approach
- The execution of the function `schedule()` can be seen as entailing 3 distinct phases:
  - ➢ check on the current process
  - ➢ Run-queue analysis (next process selection)
  - ➢ context switch

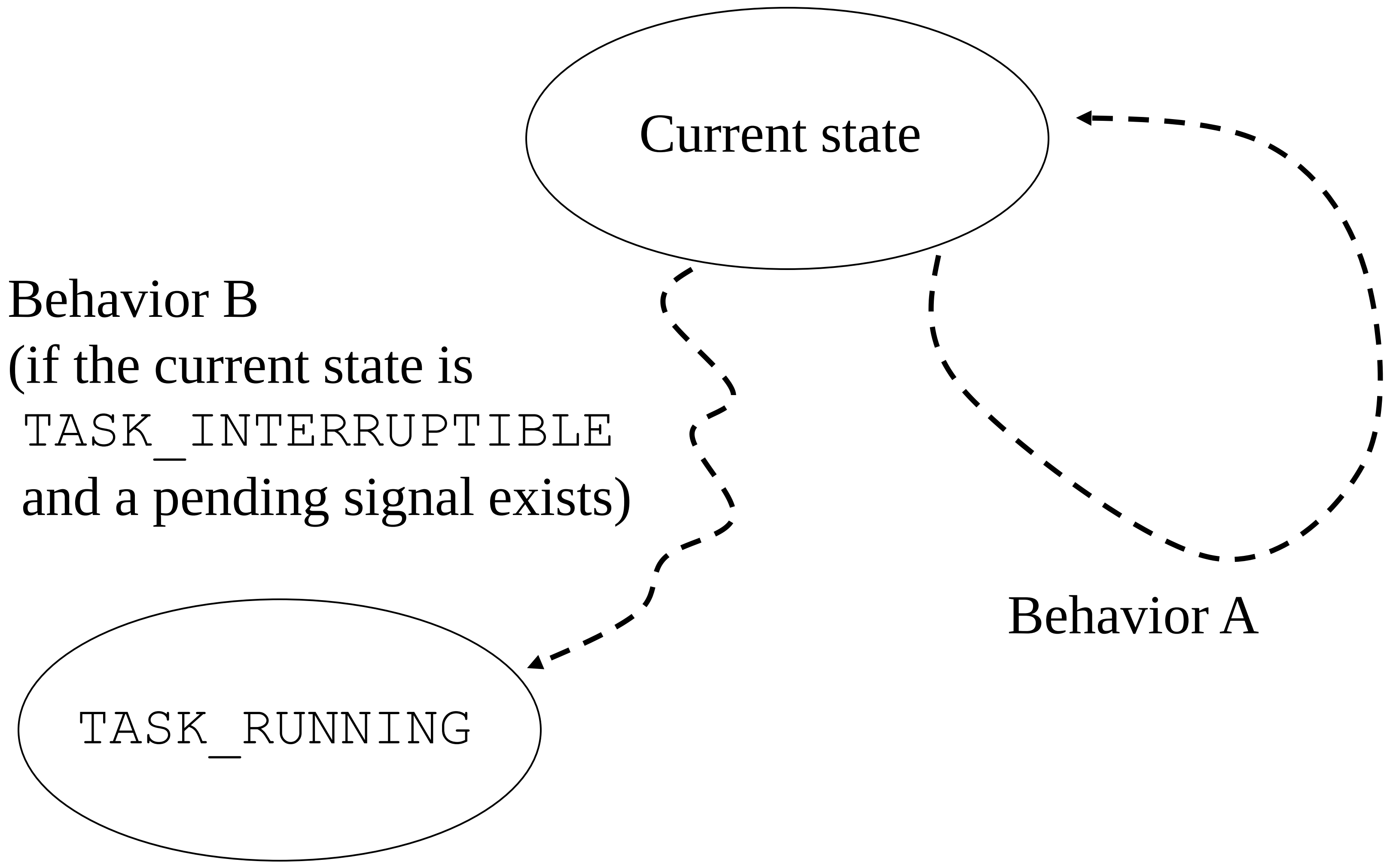
Check on the current process (update of the process state)

………

```
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)
TASK_RUNNING
Behavior A

Back to wait queues

- sleep functions for wait queues also manage the 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
}
```

Run queue analysis (2.4 style)

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

```
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;
     }
  }
```

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

Management of the epochs

- Any epoch ends when all the processes 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 PCBs kept by the run-queue
- Upon epoch ending, the next quantum is computed for all the active processes
- The formula for the recalculation is as follows

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

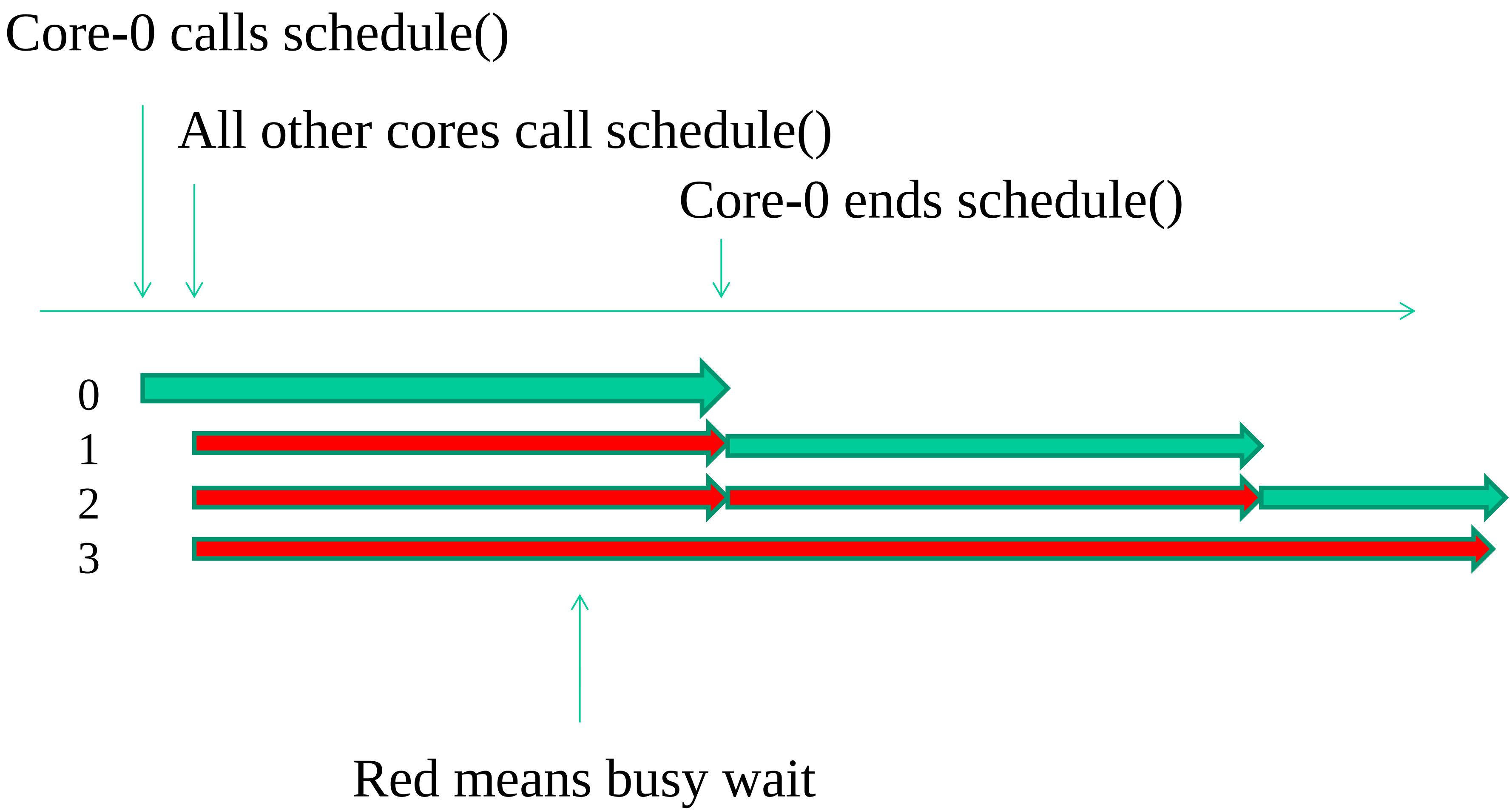
```
...............
/* Do we need to re-calculate counters? */
if (unlikely(!c)) {
      struct task_struct *p;

      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;
}
...............
```

O(n) scheduler causes

- A non-runnable task is anyway searched to determine its goodness
- Mixsture 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 crititical sections dependent on system load)

A time line example with 4 CPU-cores



2.4 scheduler advantages

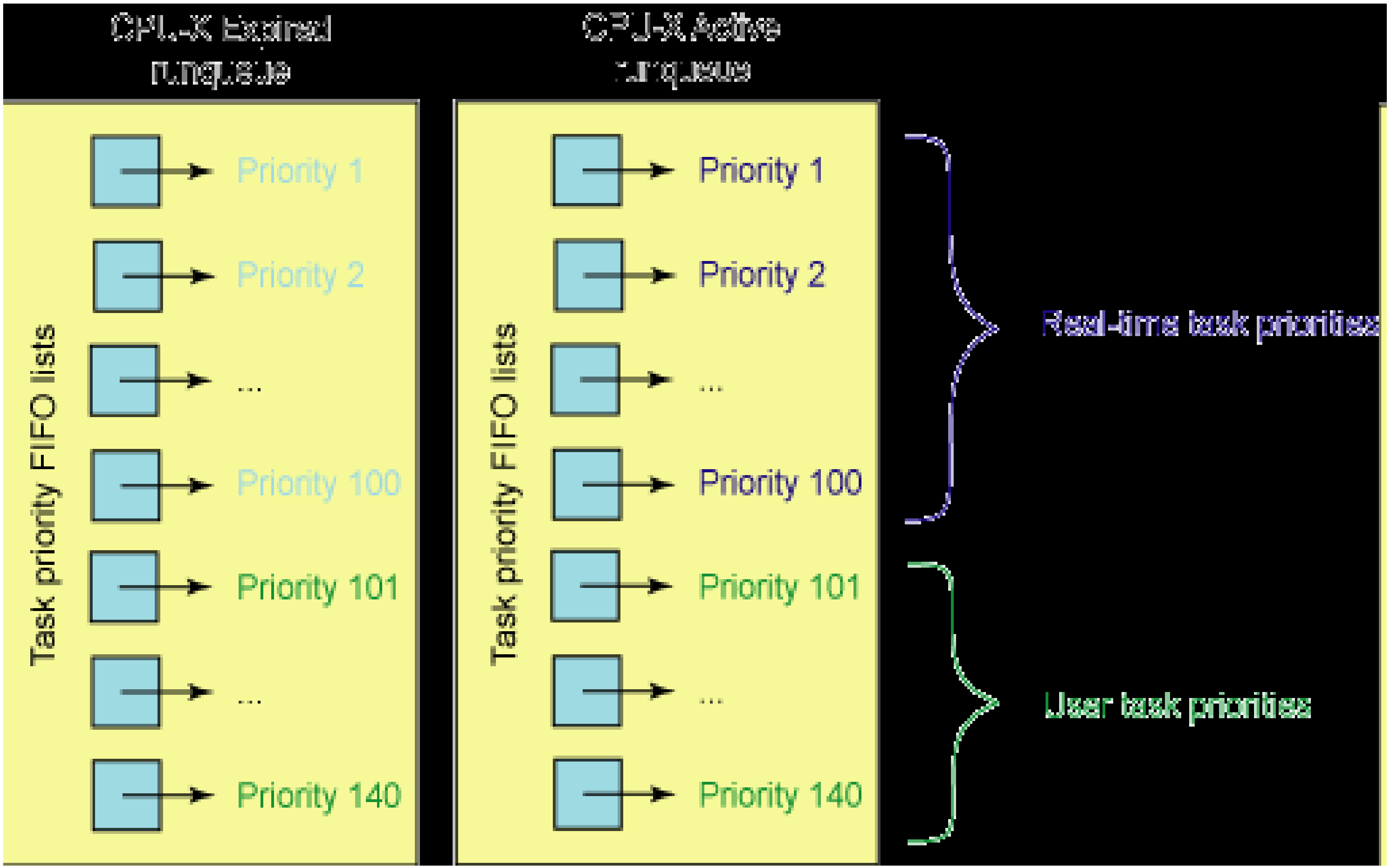
Perfect load sharing

- no CPU underutilization for whichever workload type
- no (temporaneous) binding of threads/processes to CPUs
- biased scheduling decisions vs specific CPUs are only targeted to memory performance

Kernel 2.6 advances

- O(1) scheduler in 2.6 (workload independence)
- Instead of one queue for the whole system, one active queue is created for each of the 140 possible priorities for each CPU.
- As tasks gain or lose priority, they are dropped into the appropriate queue on the processor on which they'd last run.
- It is now a trivial matter to find the highest priority task for a particular processor. A bitmap indicates which queues are not empty, and the individual queues are FIFO lists.

- You can execute an efficient find-first-bit instruction over a set of 32-bit bitmaps and then take the first task off the indicated queue every time.
- As tasks complete their timeslices, they go into a set of 140 parallel queues per processor, named the expired queues.
- When the active queue is empty, a simple pointer assignment can cause the expired queue to become the active queue again, making turnaround quite efficient.



CPU-X Expired runqueue
CPU-X Active runqueue
Task priority FIFO lists
Priority 1
Priority 2
...
Priority 100
Priority 101
...
Priority 140
Task priority FIFO lists
Priority 1
Priority 2
...
Priority 100
Priority 101
...
Priority 140
Real-time task priorities
User task priorities

(Ongoing) optimizations

- Shortcoming of 2.6 method. Once a task lands on a processor, it might use up its timeslice and get put back on a prioritized queue for rerunning—but how might it ever end up on another processor?
- In fact, if all the tasks on one processor exit, might not one processor stand idle while another round-robins three, ten or several dozen other tasks?
- To address this basic issue, the 2.6 scheduler must, on occasion, see if cross-CPU balancing is needed. It also is a requirement now because, as mentioned previously, it's possible for one CPU to be busy while another sits idle.

- Waiting to balance queues until tasks are about to complete their timeslices tends to leave CPUs idle too long.
- 2.6 leverages the process accounting, which is driven from clock ticks, to inspect the queues regularly.
- Every 200ms a processor checks to see if any other processor is out of balance and needs to be balanced with that processor. If the processor is idle, it checks every 1ms so as to get started on a real task earlier.

The source for the 2.6 scheduler is well encapsulated in the file /usr/src/linux/kernel/sched.c

Table 1. Linux 2.6 scheduler functions

Function name	**Function description**
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	Conservatively calculates the load of the source CPU (from which a task could be migrated).
target_load	Liberally calculates the load of a target CPU (where a task has the potential to be migrated).
migration_thread	High-priority system thread that migrates tasks between CPUs.

Explicit stack refresh

- Software operation
- Used when an action is finalized via local variables with lifetime across different stacks
- Used in 2.6 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;
        …….
};
```

Context switch (kernel 2.4)

- Actual context switch occurs via the macro `switch_to()` defined in `include/asm-i386/system.h`
- This macro executes a call (in the form of a jump) to the function `void __switch_to(struct task_struct *prev_p, struct task_struct *next_p)` defined in `arch/i386/kernel/process.c`
- **NOTE**: this code portion is machine dependent
- `__switch_to()` mainly executes the following two tasks
  - TSS update
  - CPU control registers update

Similarities with interrupt handlers

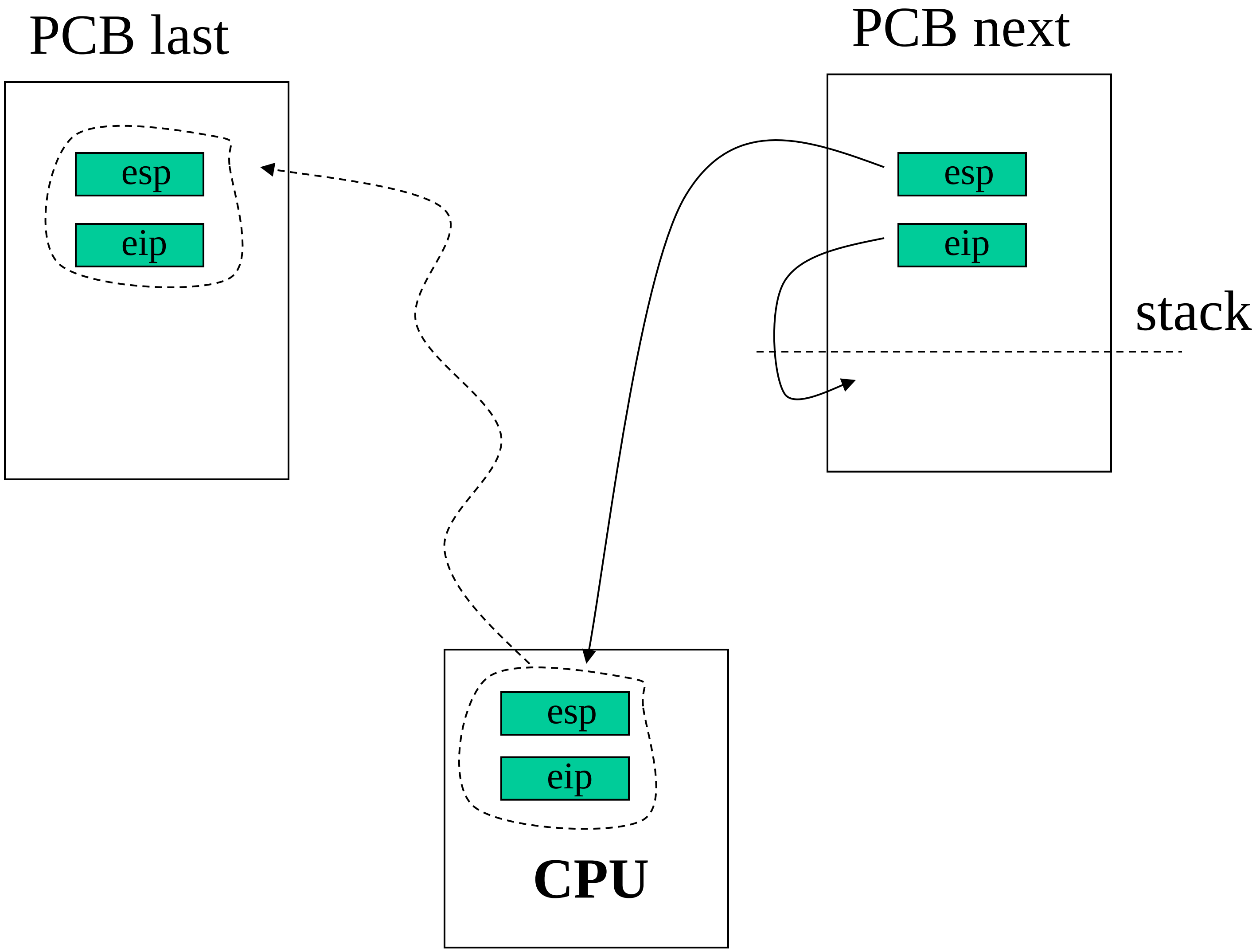
- Control bounces back to a point from which no call has been done
- This is exactly what happens for signal handlers
- The basic approach for supporting this execution scheme consists in pre-forming the stack frame for allowing the return of the activated module
- Hence, the stack frame gets assembled such in a way that the return point coincides with the instruction that follows the call to the code portion that updates the stack pointer

switch_to()

```
#define switch_to(prev,next,last) do {                              \
      asm volatile("pushl %%esi\n\t"                                \
              "pushl %%edi\n\t"                                \
              "pushl %%ebp\n\t"                                \
              "movl %%esp,%0\n\t"      /* save ESP */       \
              "movl %3,%%esp\n\t"      /* restore ESP */    \
              "movl $1f,%1\n\t"        /* save EIP */\
              "pushl %4\n\t"           /* restore EIP */    \
              "jmp __switch_to\n"                              \
              "1:\t"                                           \
              "popl %%ebp\n\t"                                 \
              "popl %%edi\n\t"                                 \
              "popl %%esi\n\t"                                 \
              :"=m" (prev->thread.esp),"=m" (prev->thread.eip),\
               "=b" (last)                                     \
              :"m" (next->thread.esp),"m" (next->thread.eip),\
               "a" (prev), "d" (next),                        \
               "b" (prev));                                    \
} while (0)
```

salva l'indirizzo della label 1 forward

Context switch scheme



__switch_to()

```
void __switch_to(struct task_struct *prev_p,
                           struct task_struct *next_p){

      struct thread_struct *prev = &prev_p->thread,
                           *next = &next_p->thread;
      struct tss_struct *tss = init_tss + smp_processor_id();
      ……

      /* Reload esp0, LDT and the page table pointer: */
      tss->esp0 = next->esp0;

      /*  Save away %fs and %gs. No need to save %es and %ds, as
       * those are always kernel segments while inside the kernel.
       */
      asm volatile("movl %%fs,%0":"=m" (*(int *)&prev->fs));
      asm volatile("movl %%gs,%0":"=m" (*(int *)&prev->gs));

      /* Restore %fs and %gs. */
      loadsegment(fs, next->fs);
      loadsegment(gs, next->gs);
      ……
}
```

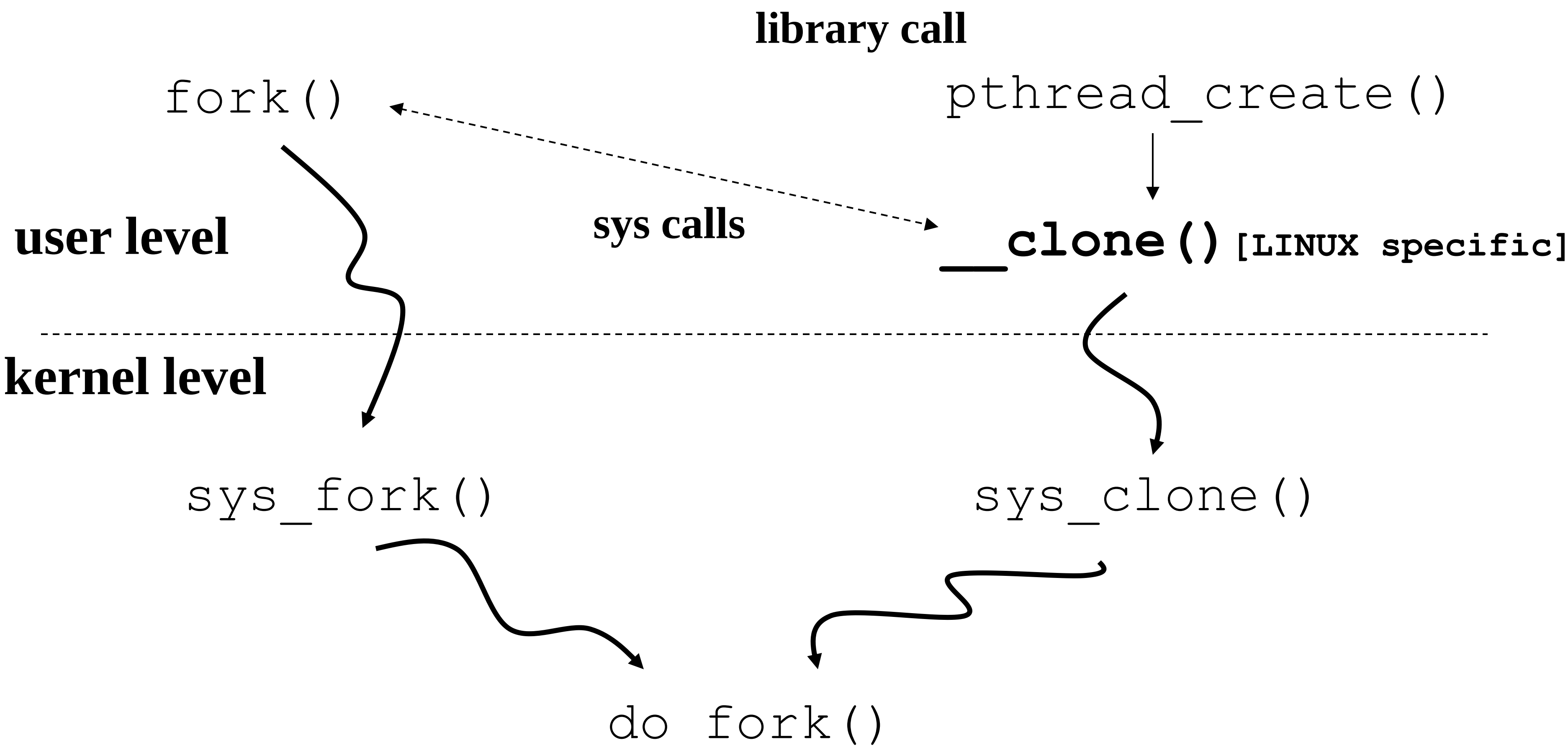
`fork()` initialization

- Initialization of the fork subsystem occurs via `fork_init()` which is defined in `kernel/fork.c`
- This sets some fields of the PCB associated with the IDLE PROCES to specific values, which will be inherited by other processes

```
void __init fork_init(unsigned long mempages){
  /*
   * The default maximum number of threads is set to a
            safe
   * value: the thread structures can take up at most half
   * of memory.
   */
  max_threads = mempages / (THREAD_SIZE/PAGE_SIZE) / 8;

  init_task.rlim[RLIMIT_NPROC].rlim_cur = max_threads/2;
  init_task.rlim[RLIMIT_NPROC].rlim_max = max_threads/2;
}
```

Process and thread creation



- `sys_fork()` and `sys_clone()` are defined in `arch/i386/kernel/process.c`
- `do_fork()` is defined in `kernel/fork.c`

```
asmlinkage int sys_fork(struct pt_regs regs)
{
      return do_fork(SIGCHLD, regs.esp, &regs, 0);
}

asmlinkage int sys_clone(struct pt_regs regs)
{
      unsigned long clone_flags;
      unsigned long newsp;

      clone_flags = regs.ebx;
      newsp = regs.ecx;
      if (!newsp)
            newsp = regs.esp;
      return do_fork(clone_flags, newsp, &regs, 0);
}
```

Note!!!!

- When relying on the `__clone()` system call, preventive allocation of the stack for the new thread needs to take place
- This is because thread of a same process share the same address space, which does not occur when forking processes
- The virtual address of the base of the stack of the new thread must be written into the **ecx** register right before giving control to the kernel for executing `sys_clone()`
- The thread activation flags destined as input to `do_fork()` need to be written into **ebx**

Note!!!!

- The documented system call `__clone()` is a wrapper for the actual system call
- The latter takes as input 2 parameters:
  - the operation flags
  - The address of the new user-level stack
- Activation of the thread function takes place as a subroutine call followed by a call to `exit()`

`do_fork()` (main tasks)

- Fresh PCB/kernel-stack allocation
- Copy/setup of PCB information
- Copy/setup of PCB linked data structures
- What information is copied or inherited (namely shared into the original buffers) depends on the value of the flags passed in input to `do_fork()`
- Admissible values for the flags are defined in `include/linux/sched.h`
  - `#define CLONE_VM 0x00000100 /* set if VM shared between processes */`
  - `#define CLONE_FS 0x00000200 /* set if fs info shared between processes */`
  - `#define CLONE_FILES 0x00000400 /* set if open files shared between processes */`
  - `#define CLONE_PID 0x00001000 /* set if pid shared */`
  - `#define CLONE_PARENT 0x00008000 /* set if we want to have the same parent as the cloner*/`

```c
int do_fork(unsigned long clone_flags, unsigned long stack_start,
           struct pt_regs *regs, unsigned long stack_size)
{
      ……
      p = alloc_task_struct();
      if (!p) goto fork_out;
      *p = *current;

      ……
      p->state = TASK_UNINTERRUPTIBLE;
      ……
      p->pid = get_pid(clone_flags);
      if (p->pid == 0 && current->pid != 0)
             goto bad_fork_cleanup;

      p->run_list.next = NULL;
      p->run_list.prev = NULL;
      ……
      init_waitqueue_head(&p->wait_chldexit);
      ……
```

```
p->sigpending = 0;
init_sigpending(&p->pending);
……
p->start_time = jiffies;
……
/* copy all the process information */
if (copy_files(clone_flags, p))   goto bad_fork_cleanup;
if (copy_fs(clone_flags, p))      goto bad_fork_cleanup_files;
if (copy_sighand(clone_flags, p)) goto bad_fork_cleanup_fs;
if (copy_mm(clone_flags, p))      goto bad_fork_cleanup_sighand;
retval = copy_namespace(clone_flags, p);
if (retval)  goto bad_fork_cleanup_mm;
retval = copy_thread(0, clone_flags, stack_start,
                                  stack_size, p, regs);
if (retval) goto bad_fork_cleanup_namespace;
p->semundo = NULL;

……
p->exit_signal = clone_flags & CSIGNAL;
…….
```

```
/*
	 * "share" dynamic priority between parent and child,
	        thus the
	 * total amount of dynamic priorities in the system
	        doesn't change,
	 * more scheduling fairness. This is only important in
	        the first
	 * timeslice, on the long run the scheduling behaviour is
	        unchanged.
	 */
	p->counter = (current->counter + 1) >> 1;
	current->counter >>= 1;
	if (!current->counter)
	        current->need_resched = 1;

	/*
	 * Ok, add it to the run-queues and make it
	 * visible to the rest of the system.
	 *
	 * Let it rip!
	 */
	retval = p->pid;
	……
```

```
/* Need tasklist lock for parent etc handling! */
        write_lock_irq(&tasklist_lock);

        /* CLONE_PARENT re-uses the old parent */
        p->p_opptr = current->p_opptr;
        p->p_pptr = current->p_pptr;
        if (!(clone_flags & CLONE_PARENT)) {
                p->p_opptr = current;
                if (!(p->ptrace & PT_PTRACED))
                        p->p_pptr = current;
        }
        ……
        SET_LINKS(p);
        hash_pid(p);
        nr_threads++;
        write_unlock_irq(&tasklist_lock);
        ……
        wake_up_process(p);        /* do this last */
        ++total_forks;
        ……
  fork_out:
        return retval;
        ……
}
```

`copy_thread()`

- Part of the job of `do_fork()` is carried out by the `copy_thread()` function, which is defined in `arch/i386/kernel/process.c`
- This function set-up PCB information such in a way the user level stack pointer gets correctly initialized
- It also sets-up the return value (zero) for the `clone()` system call thus indicating whether we are running into the child process/thread
- This return value is as usual written into the **eax** register

```
int copy_thread(int nr, unsigned long clone_flags, unsigned long esp,
        unsigned long unused,
        struct task_struct * p, struct pt_regs * regs)
{
  struct pt_regs * childregs;

  childregs = ((struct pt_regs *) (THREAD_SIZE + (unsigned long) p)) - 1;
  struct_cpy(childregs, regs);
  childregs->eax = 0;
  childregs->esp = esp;

  p->thread.esp = (unsigned long) childregs;
  p->thread.esp0 = (unsigned long) (childregs+1);

  p->thread.eip = (unsigned long) ret_from_fork;

  savesegment(fs,p->thread.fs);
  savesegment(gs,p->thread.gs);

  unlazy_fpu(current);
  struct_cpy(&p->thread.i387, &current->thread.i387);

  return 0;
}
```

copy_mm()

```
static int copy_mm(unsigned long clone_flags,
                                  struct task_struct * tsk)
{
      struct mm_struct * mm, *oldmm;
      int retval;
      ……
      tsk->mm = NULL;
      tsk->active_mm = NULL;
      ……
      oldmm = current->mm;
      ……
      if (clone_flags & CLONE_VM) {
            atomic_inc(&oldmm->mm_users);
            mm = oldmm;
            goto good_mm;
      }

      retval = -ENOMEM;
      mm = allocate_mm();
      if (!mm)
            goto fail_nomem;
```

```
        /* Copy the current MM stuff.. */
        memcpy(mm, oldmm, sizeof(*mm));
        if (!mm_init(mm)) goto fail_nomem;
        ……

        down_write(&oldmm->mmap_sem);
        retval = dup_mmap(mm);
        up_write(&oldmm->mmap_sem);

        if (retval) goto free_pt;
   /*
     * child gets a private LDT (if there was an LDT in the parent)
    */
        copy_segments(tsk, mm);
good_mm:
        tsk->mm = mm;
        tsk->active_mm = mm;
        return 0;

free_pt:
        mmput(mm);
fail_nomem:
        return retval;
}
```

Principal functions exploited by **`copy_mm()`**

- in `kernel/fork.c`
  - ➢ `mm_init()`
    - ✓ Allocation of a fresh PGD
  - ➢ `dup_mmap()`
    - ✓ Sets up any information for memory management within the new process context

```
static struct mm_struct * mm_init(struct mm_struct
                                         * mm)
{
     atomic_set(&mm->mm_users, 1);
     atomic_set(&mm->mm_count, 1);
     init_rwsem(&mm->mmap_sem);
     mm->page_table_lock = SPIN_LOCK_UNLOCKED;
     mm->pgd = pgd_alloc(mm);
     mm->def_flags = 0;
     if (mm->pgd)
          return mm;
     free_mm(mm);
     return NULL;
}
```

Note!!!!!

- The macro `pgd_alloc()`, which is defined in `include/asm-i386/pgalloc.h`, beyond allocating a frame for the PGD also executes the following operations
  - Resets the PGD (the first 768 entries) for the portion associated with user space addressing (0-3 GB)
  - Copies kernel level addressing information from the current process PGD to the PGD associated with the new process (interval starting from 3 GB, namely from the entry 768)
  - All these tasks take place via a call to the function `get_pgd_slow()` definied in `include/asm-i386/pgalloc.h`

```
static inline int dup_mmap(struct mm_struct * mm)
{
	struct vm_area_struct * mpnt, *tmp, **pprev;
	int retval;
	……
	mm->mmap = NULL;
	mm->mmap_cache = NULL;
	mm->map_count = 0;
	……
	pprev = &mm->mmap;

	………
	for (mpnt = current->mm->mmap ; mpnt ; mpnt = mpnt->vm_next){
		……
		tmp = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
		if (!tmp)	goto fail_nomem;
		*tmp = *mpnt;
		tmp->vm_flags &= ~VM_LOCKED;
		tmp->vm_mm = mm;
		tmp->vm_next = NULL;
		……
		retval = copy_page_range(mm, current->mm, tmp);
	……
}
```

`copy_page_range()`

- Is defined in `linux/mm/memory.c`
- For any range of addresses associated with the `vm_area_struct` structure, this function set-up the PTE page table
- This may lead to cover the user level addressing range only partially
- In such a case, additional PTE tables may be allocated as a result of the call to `malloc()`

Copy-on-write (cow)

```
int copy_page_range(struct mm_struct *dst, struct mm_struct *src,
                          struct vm_area_struct *vma){
        pgd_t * src_pgd, * dst_pgd;
        unsigned long address = vma->vm_start;
        unsigned long end = vma->vm_end;
        unsigned long cow =
                (vma->vm_flags & (VM_SHARED | VM_MAYWRITE)) == VM_MAYWRITE;
        ……
        for (;;) {
        ……
                do {
                        pte_t * src_pte, * dst_pte;
        ………
                        src_pte = pte_offset(src_pmd, address);
                        dst_pte = pte_alloc(dst, dst_pmd, address);
        ……
                        do {
                                pte_t pte = *src_pte;
                                ……
/* If it's a COW mapping,  write protect it both in the parent and the child */
                                if (cow && pte_write(pte)) {
                                        ptep_set_wrprotect(src_pte);
                                        pte = *src_pte;
                                }
        ……
}
```

Kernel threads (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

```c
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

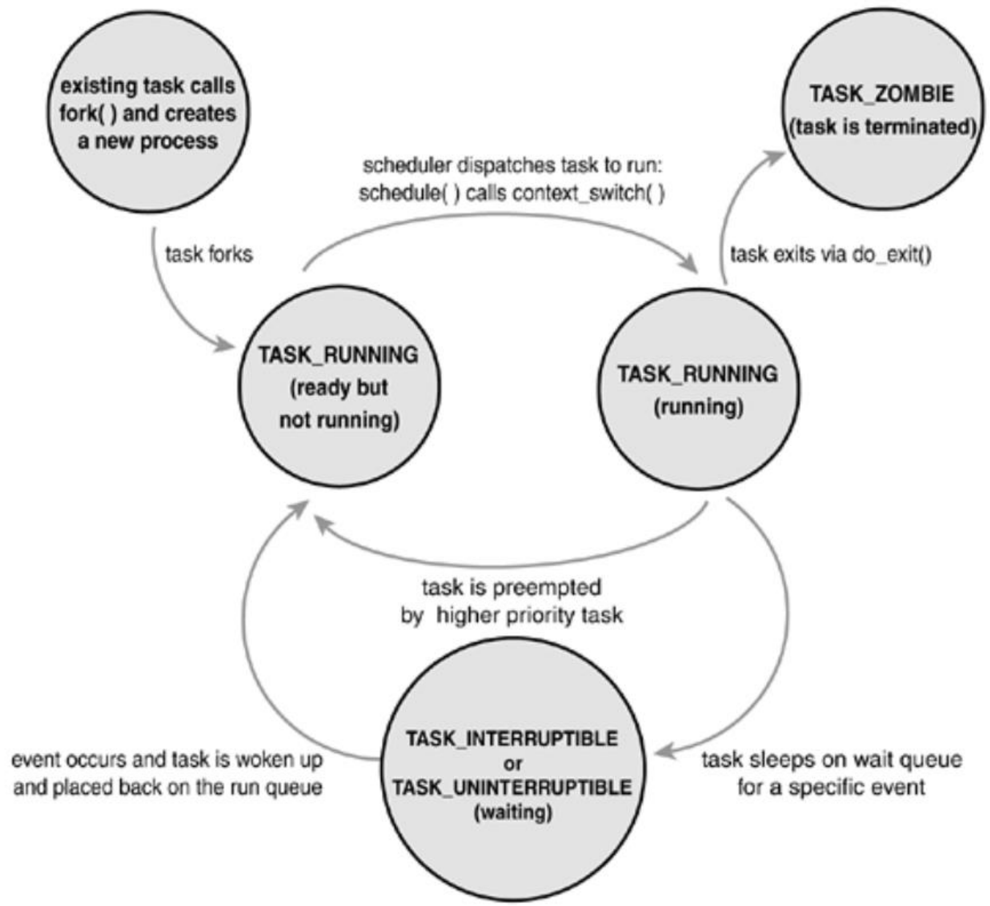
- The created thread sleeps on a wait queue
- So it exists but is not really active
- We need to explicitly awake it
- As for signals we have the following:
  - ✓ We can kill
  - ✓ Killing only has the effect of awakening the thread (if sleeping) but no message delivery is logged in the signal mask
  - ✓ We have to explicitly enable the delivery if we need to catch that the signal has arrived

A reference kernel-thread functions suite

- `start_kthread`: creates a new kernel thread. Can be called from any process context but not from interrupt. The functions blocks until the thread started.
- `stop_kthread`: stop the thread. Can be called from any process context but the thread to be terminated. Cannot be called from interrupt context. The function blocks until the thread terminated.
- `init_kthread`: sets the environment of the new threads. Is to be called out of the created thread.
- `exit_kthread`: needs to be called by the thread to be terminated on exit

Creation of new Thread

A new thread is created with `kernel_thread()`. The thread inherits properties from its parents. To make sure that we do not get any weired properties, we let `keventd` create the new thread.


existing task calls fork( ) and creates a new process
TASK_ZOMBIE (task is terminated)
scheduler dispatches task to run: schedule( ) calls context_switch( )
task forks
task exits via do_exit()
TASK_RUNNING (ready but not running)
TASK_RUNNING (running)
task is preempted by higher priority task
event occurs and task is woken up and placed back on the run queue
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE (waiting)
task sleeps on wait queue for a specific event