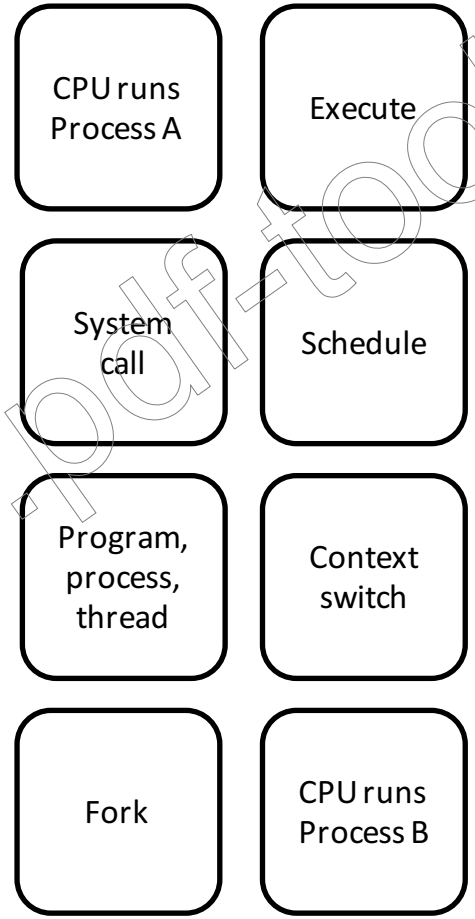
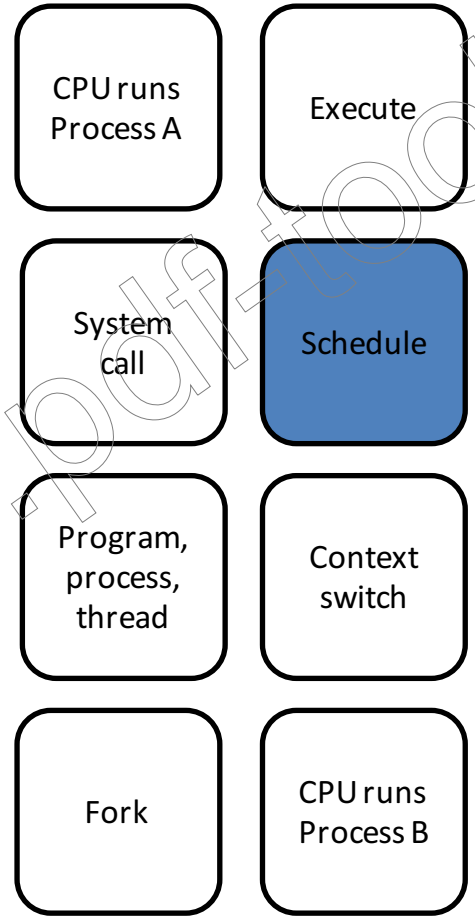
Operating System Design and Implementation

Process Management – Part II

Shiao-Li Tsao

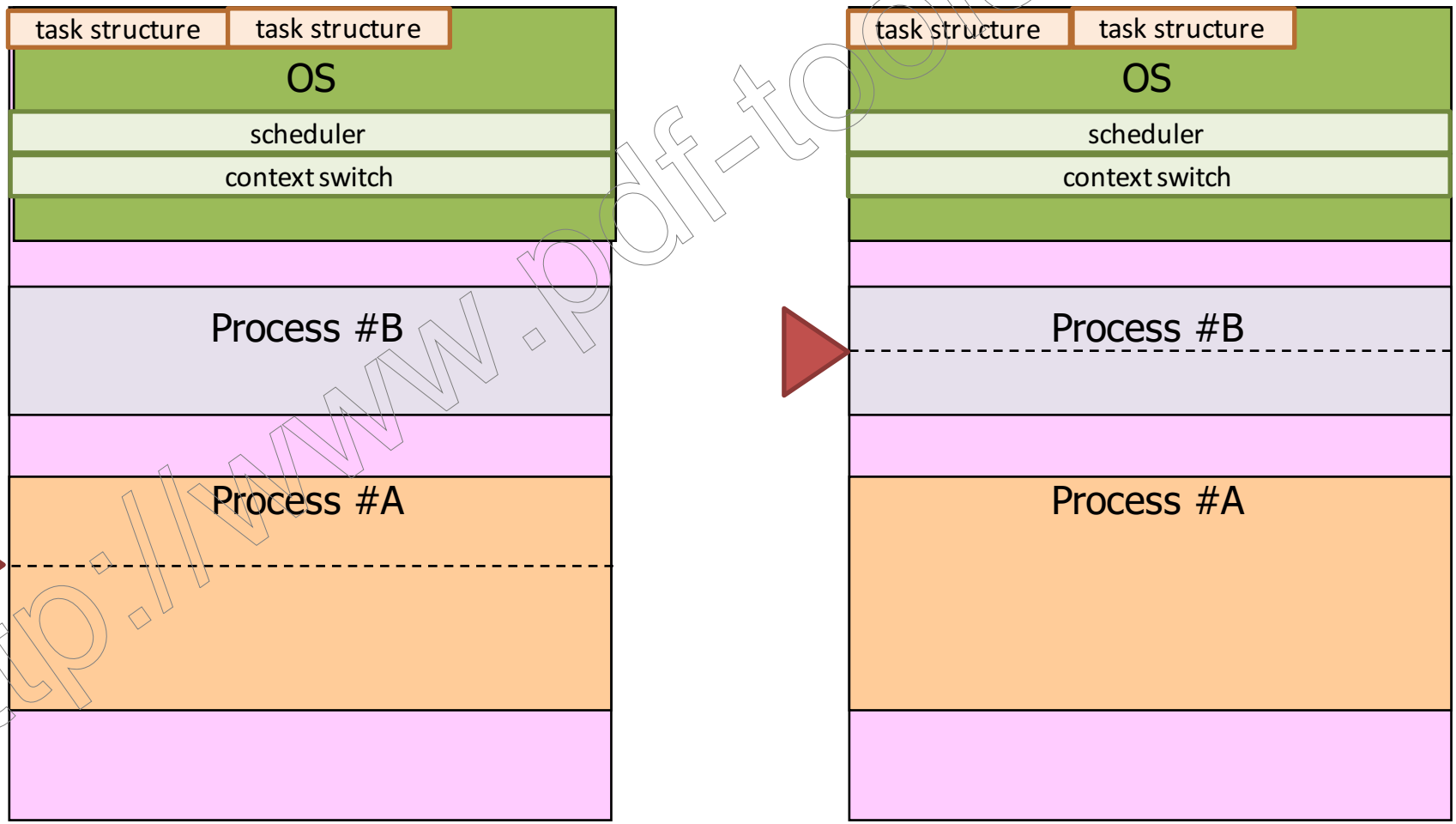




Process schedule and context switching in Linux

- Scheduling
  - Find the next suitable process to run
- Context switch
  - Store the context of the current process, restore the context of the next process

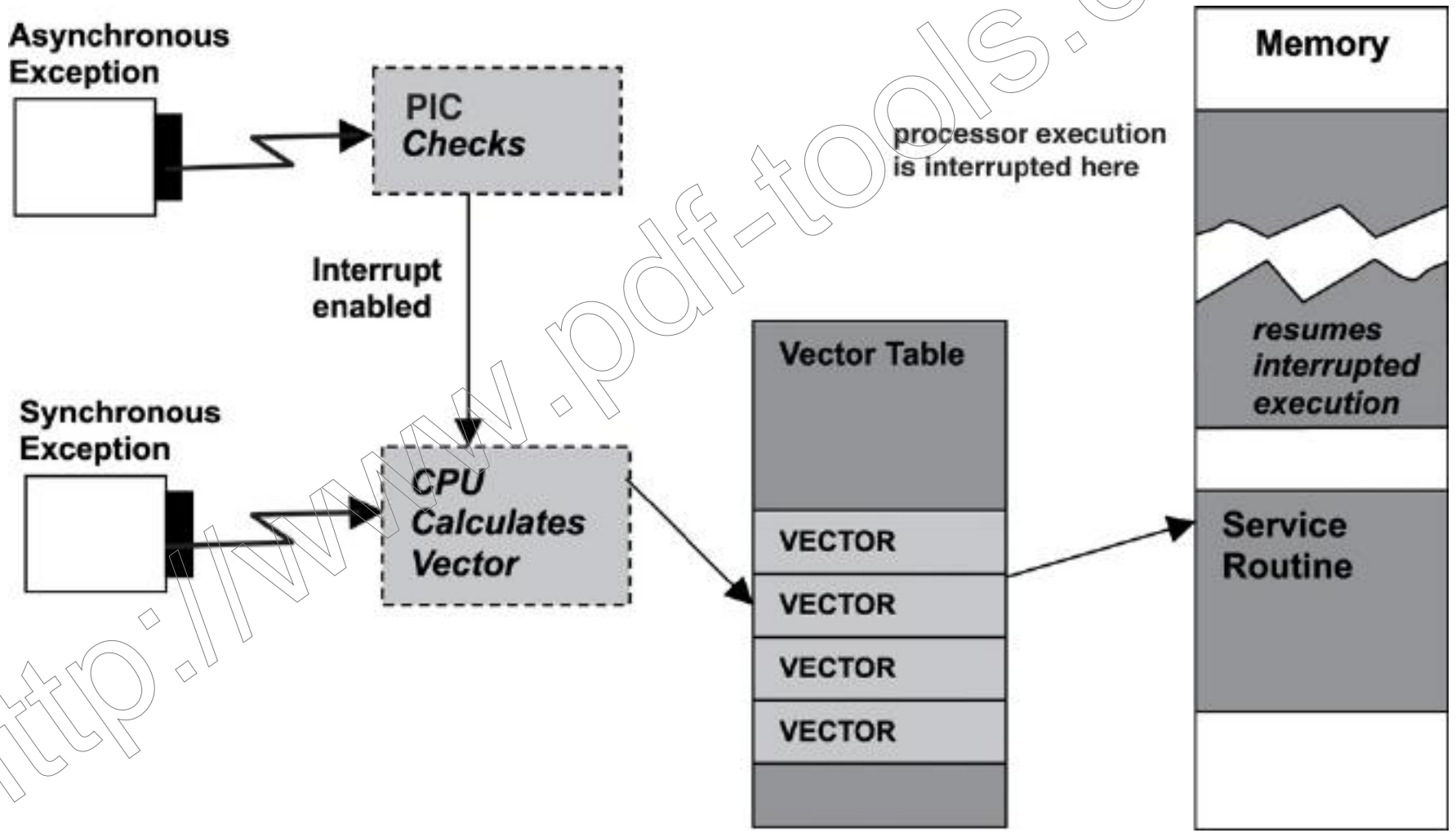
Scheduler in Details



Process schedule and context switching in Linux

- When is the scheduler be invoked
  - Direct invocation vs. Lazy invocation
  - When returning to user-space from a system call
  - When returning to user-space from an interrupt handler
  - When an interrupt handler exits, before returning to kernel-space
  - If a task in the kernel explicitly calls schedule()
  - If a task in the kernel blocks (which results in a call to schedule())

Interrupt Basics



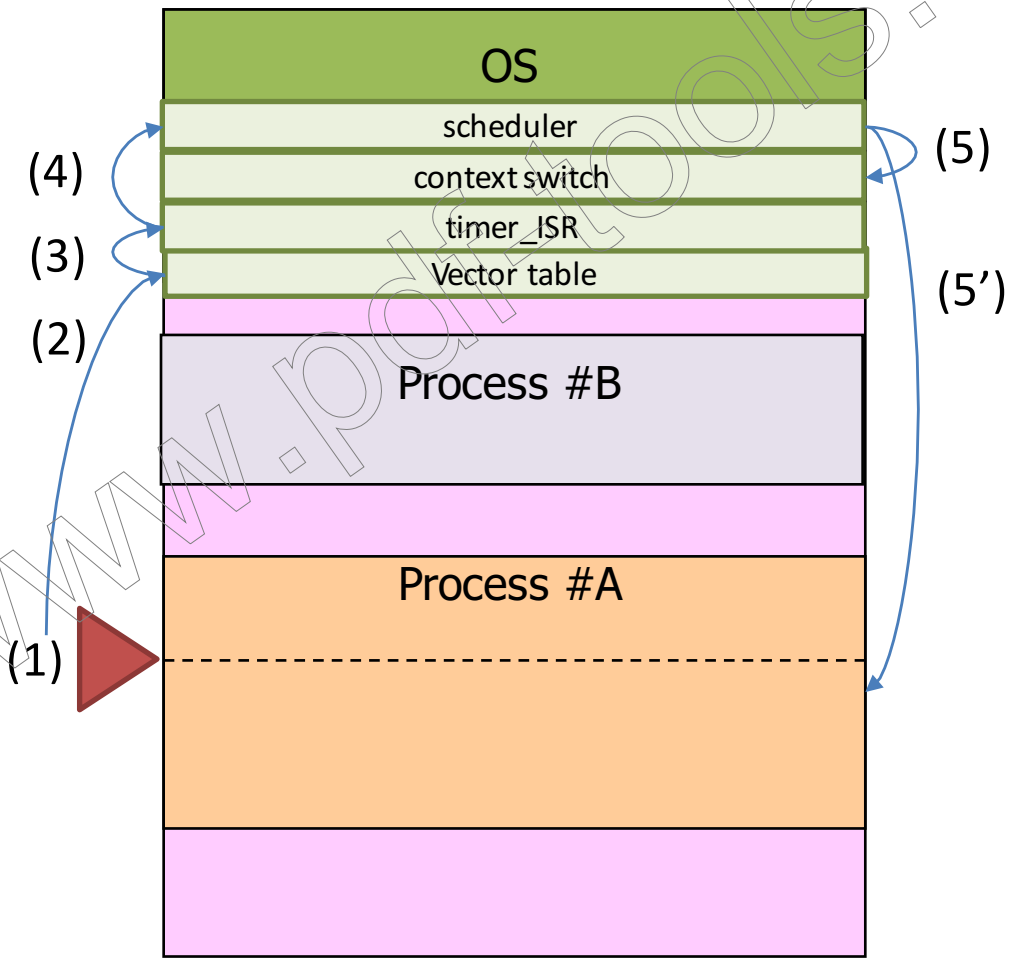
Source: Qing Li "real-time concepts for embedded systems"

X86 Interrupts

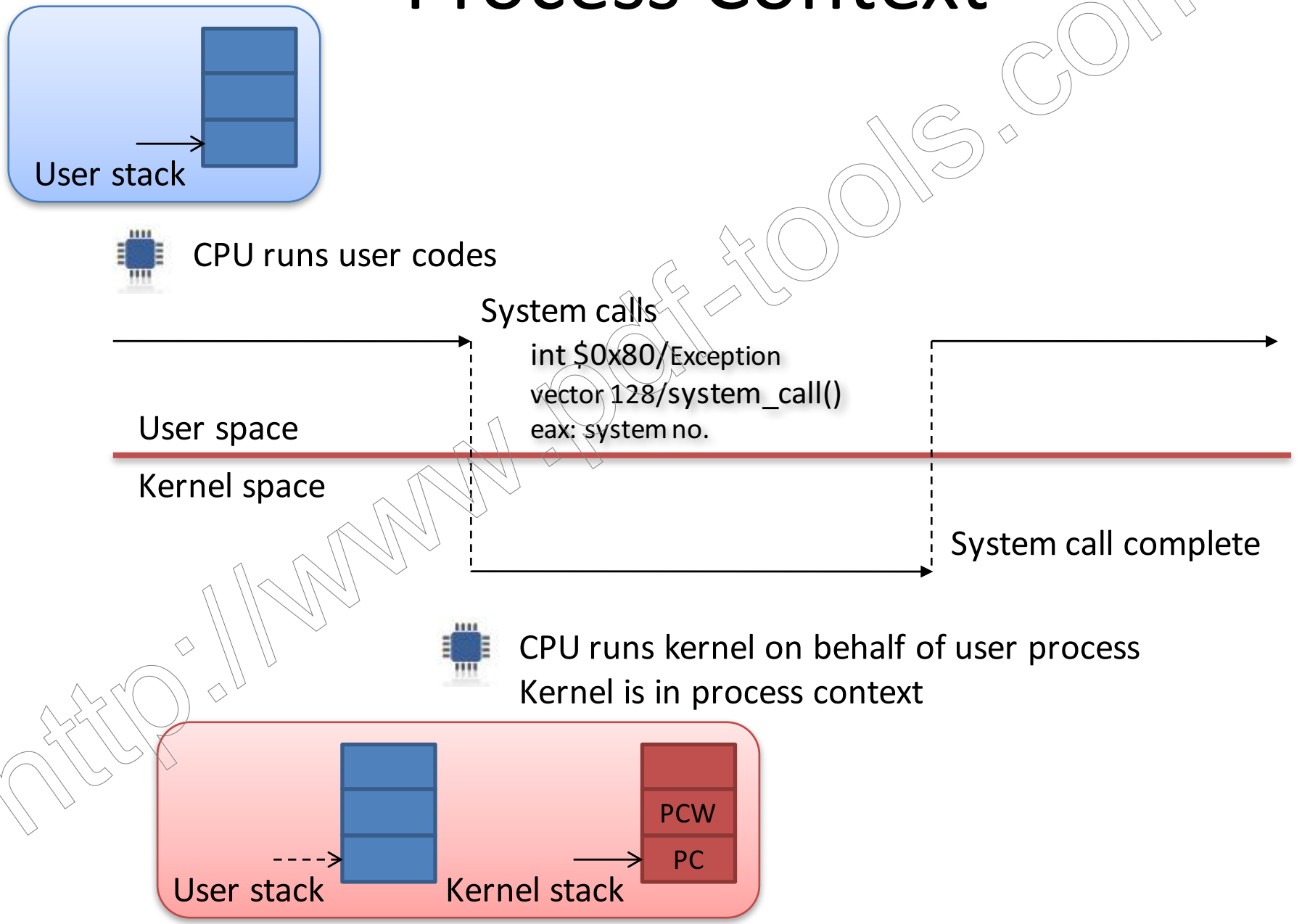
IRQ	Standard Function	Bus Slot	Card Type	Recommended Use
0	System Timer	No	-	-
1	Keyboard Controller	No	-	-
2	2nd IRQ Controller Cascade	No	-	-
8	Real-Time Clock	No	-	-
9	Avail. (as IRQ2 or IRQ9)	Yes	8/16-bit	Network Card
10	Available	Yes	16-bit	USB
11	Available	Yes	16-bit	SCSI Host Adapter
12	Mouse Port/Available	Yes	16-bit	Mouse Port
13	Math Coprocessor	No	—	—
14	Primary IDE	Yes	16-bit	Primary IDE (hard disks)
15	Secondary IDE	Yes	16-bit	2nd IDE (CD-ROM/Tape)
3	Serial 2 (COM2:)	Yes	8/16-bit	COM2:/Internal Modem
4	Serial 1 (COM1:)	Yes	8/16-bit	COM1:
5	Sound/Parallel 2 (LPT2:)	Yes	8/16-bit	Sound Card
6	Floppy Controller	Yes	8/16-bit	Floppy Controller
7	Parallel 1 (LPT1:)	Yes	8/16-bit	LPT1:

Vector range	Use
0–19 (0x0–0x13)	Nonmaskable interrupts and exceptions
20–31 (0x14–0x1f)	Intel-reserved
32–127 (0x20–0x7f)	External interrupts (IRQs)
128 (0x80)	Programmed exception for system calls
129–238 (0x81–0xee)	External interrupts (IRQs)

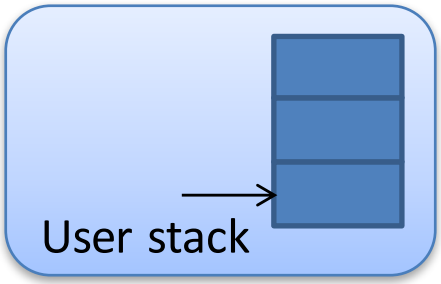
Time Interrupt Basics



Process Context



Interrupt Context



CPU runs user codes

Interrupt occurs

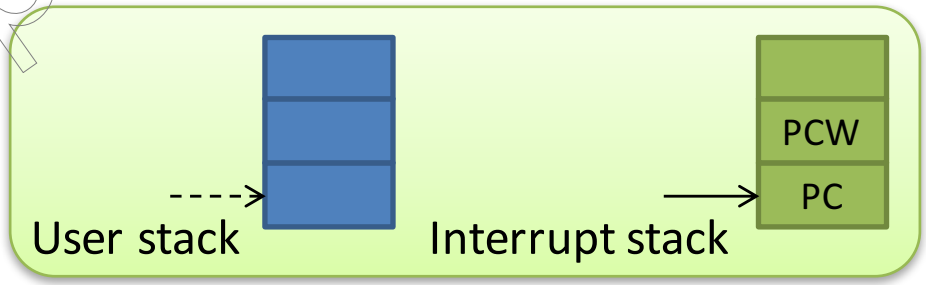
User space

Kernel space

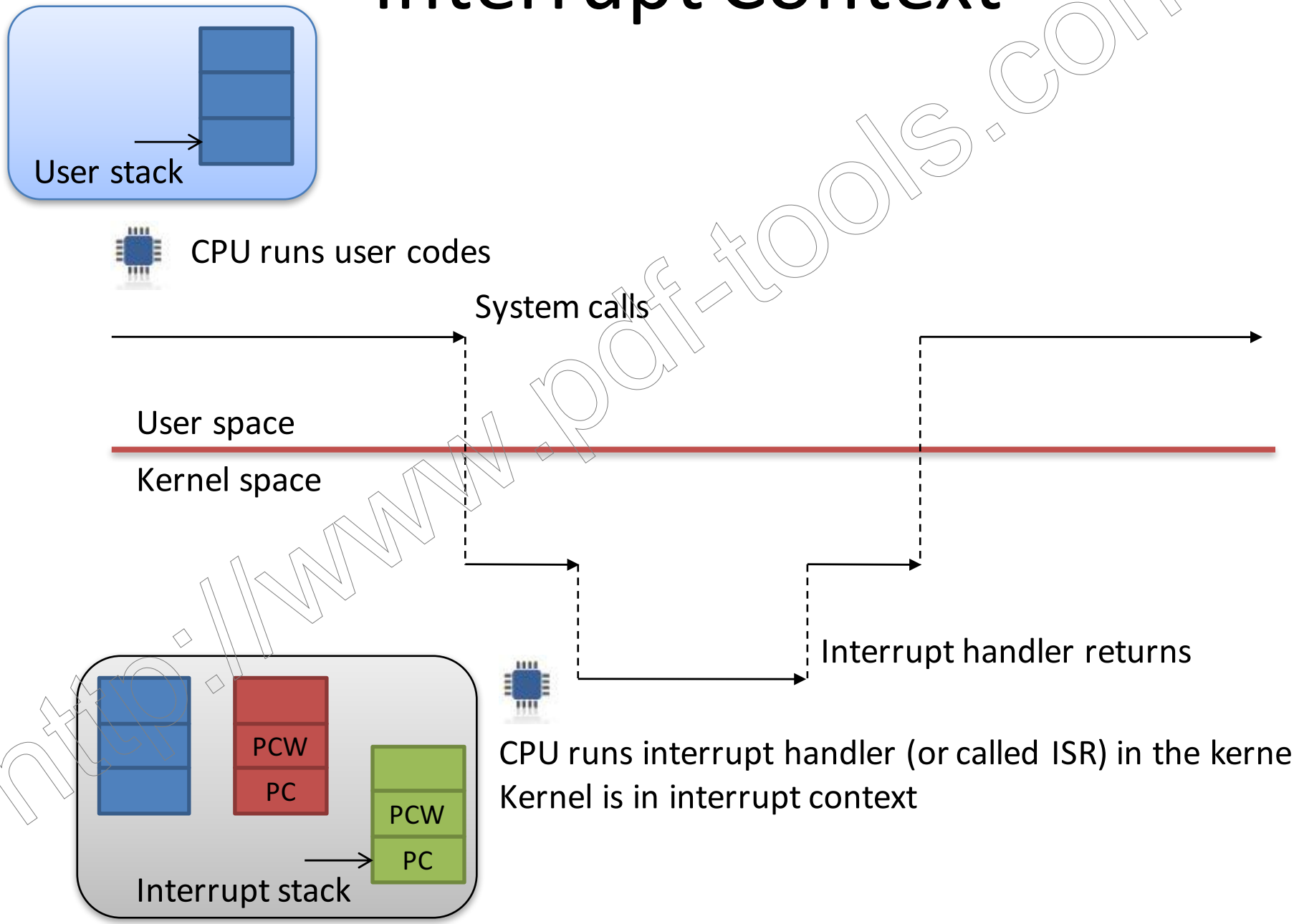
Interrupt handler returns

CPU runs interrupt handler (or called ISR) in the kernel space

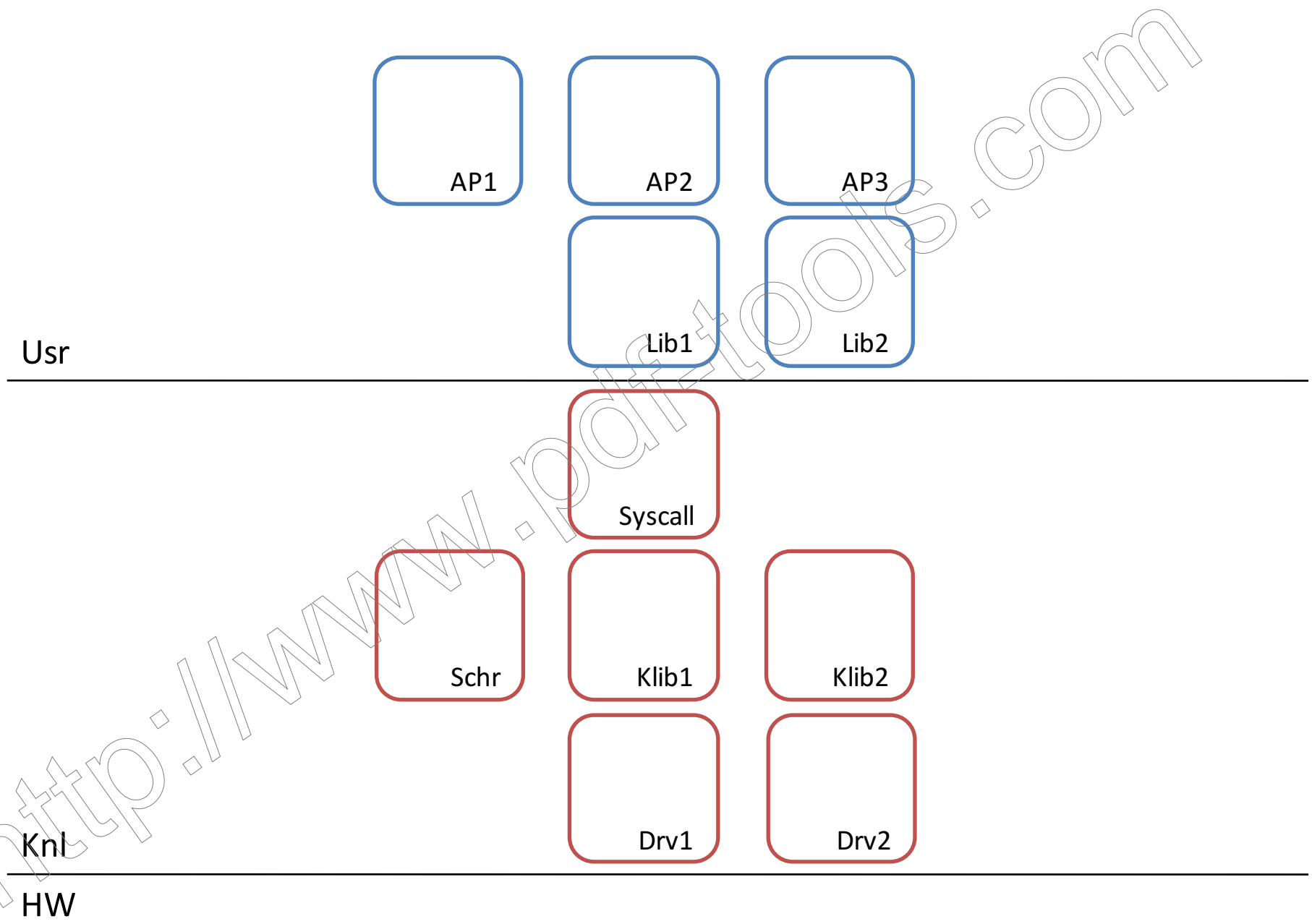
Kernel is in interrupt context



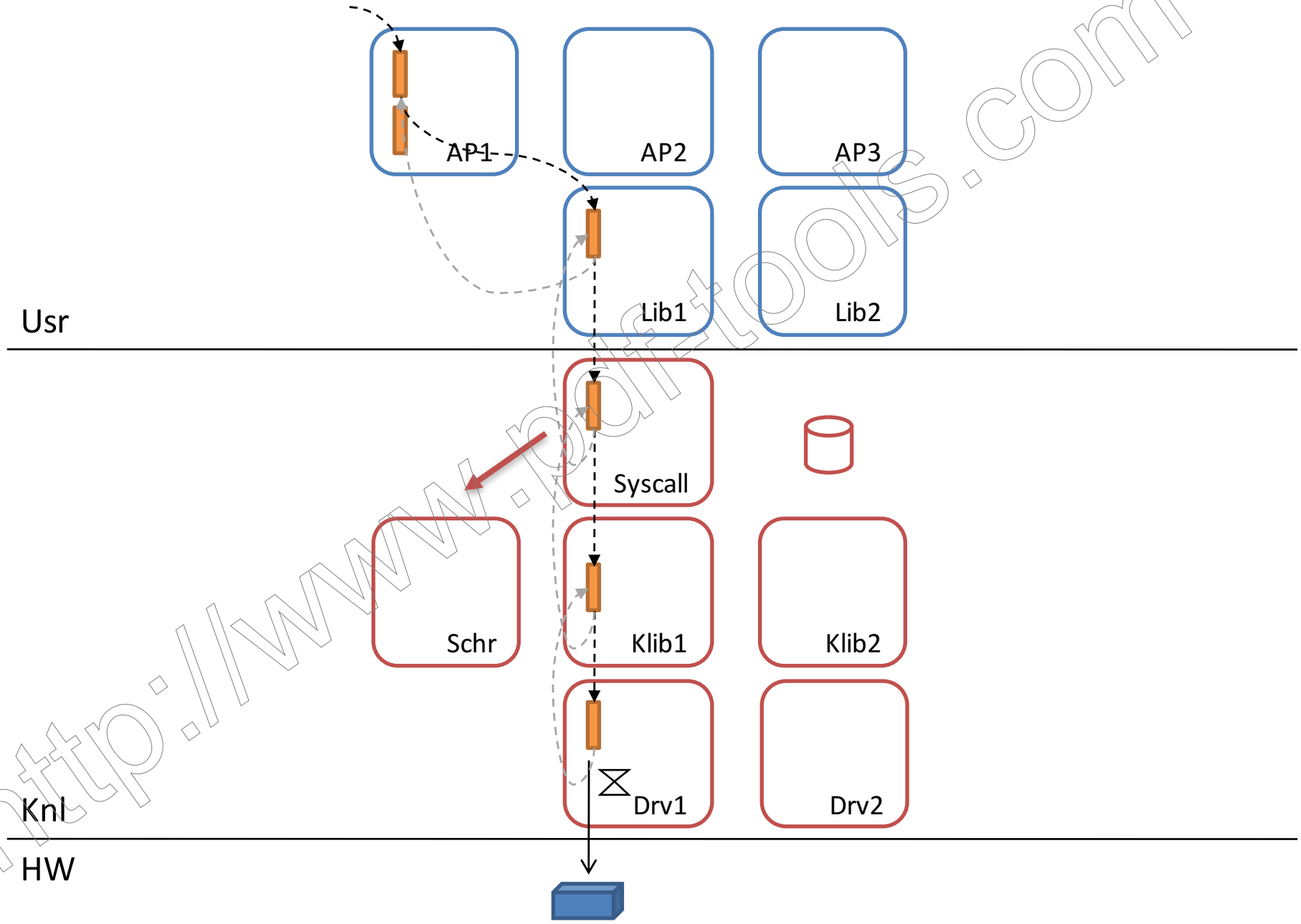
Interrupt Context



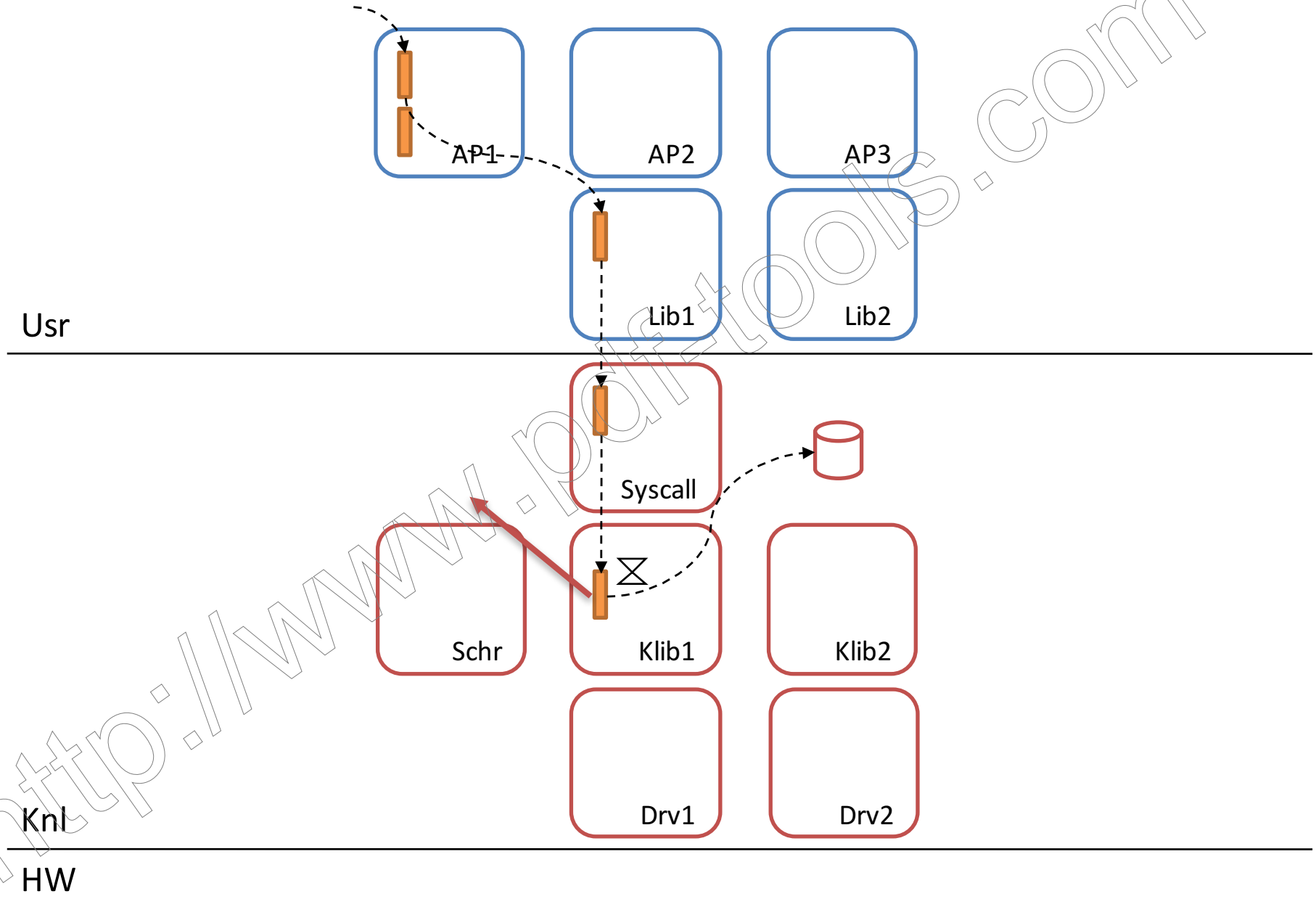
CPU runs interrupt handler (or called ISR) in the kernel space
Kernel is in interrupt context



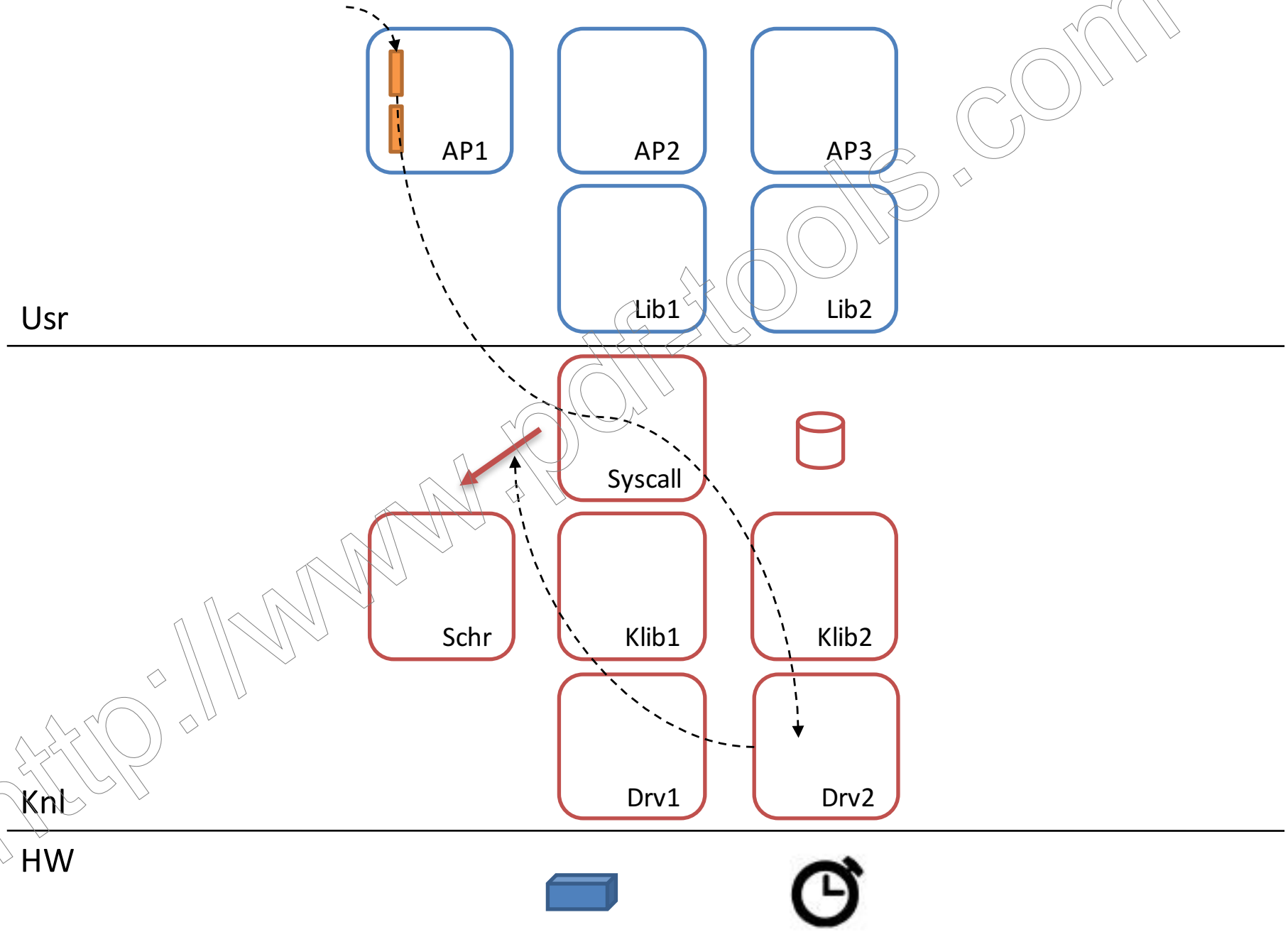
When returning to user-space from a system call



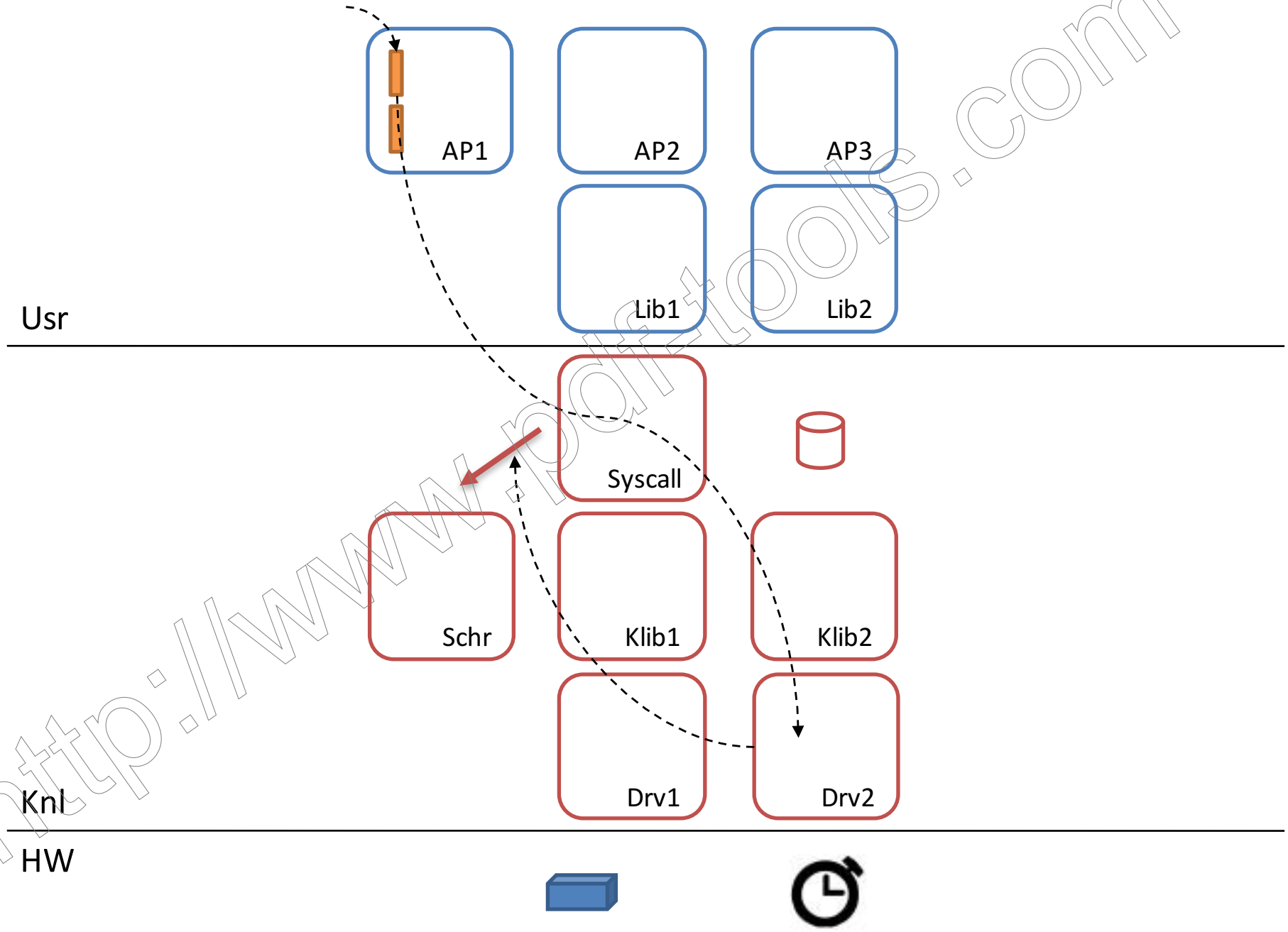
If a task in the kernel blocks (which results in a call to schedule())



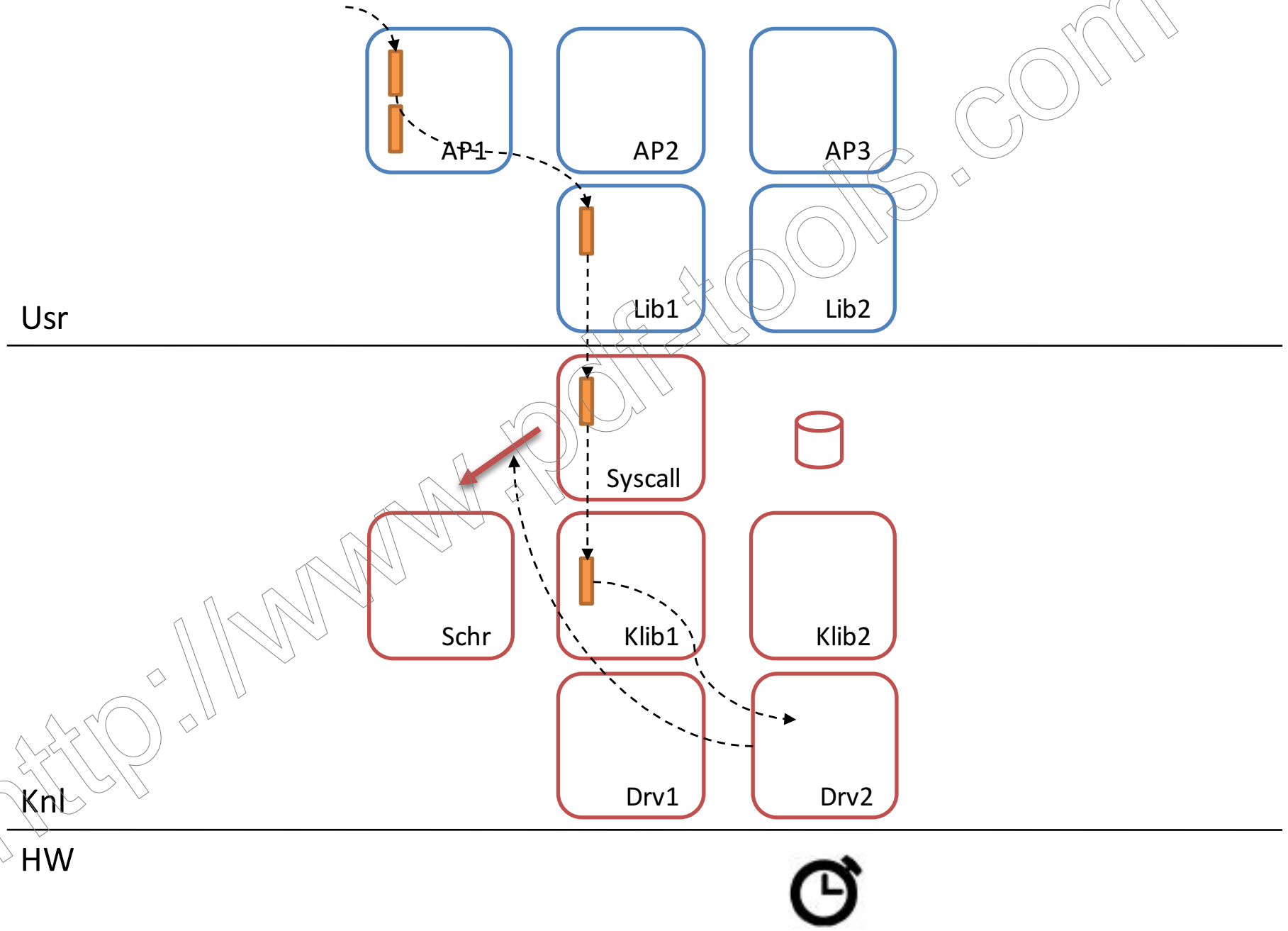
When returning to user-space from an interrupt handler



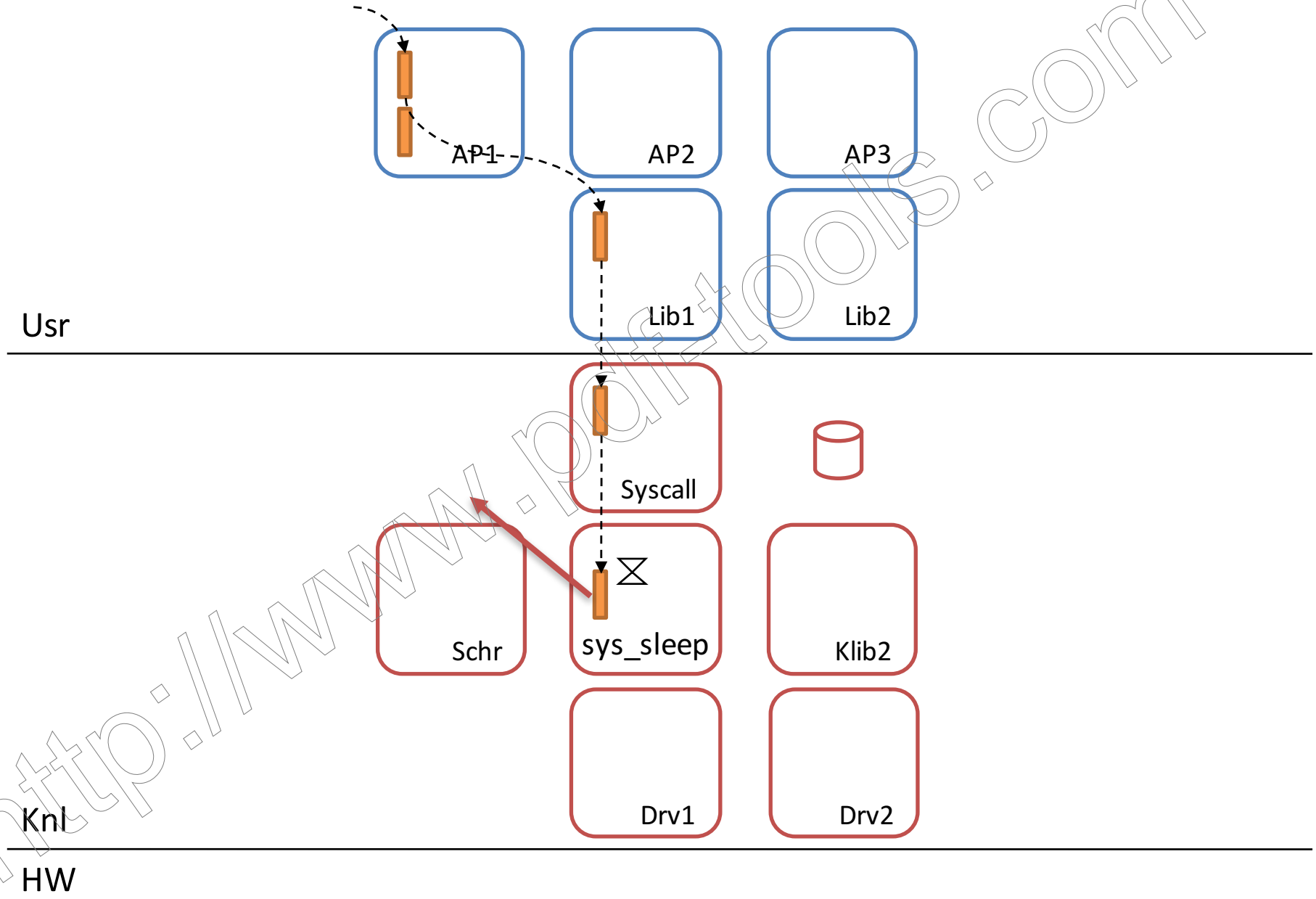
When returning to user-space from an interrupt handler



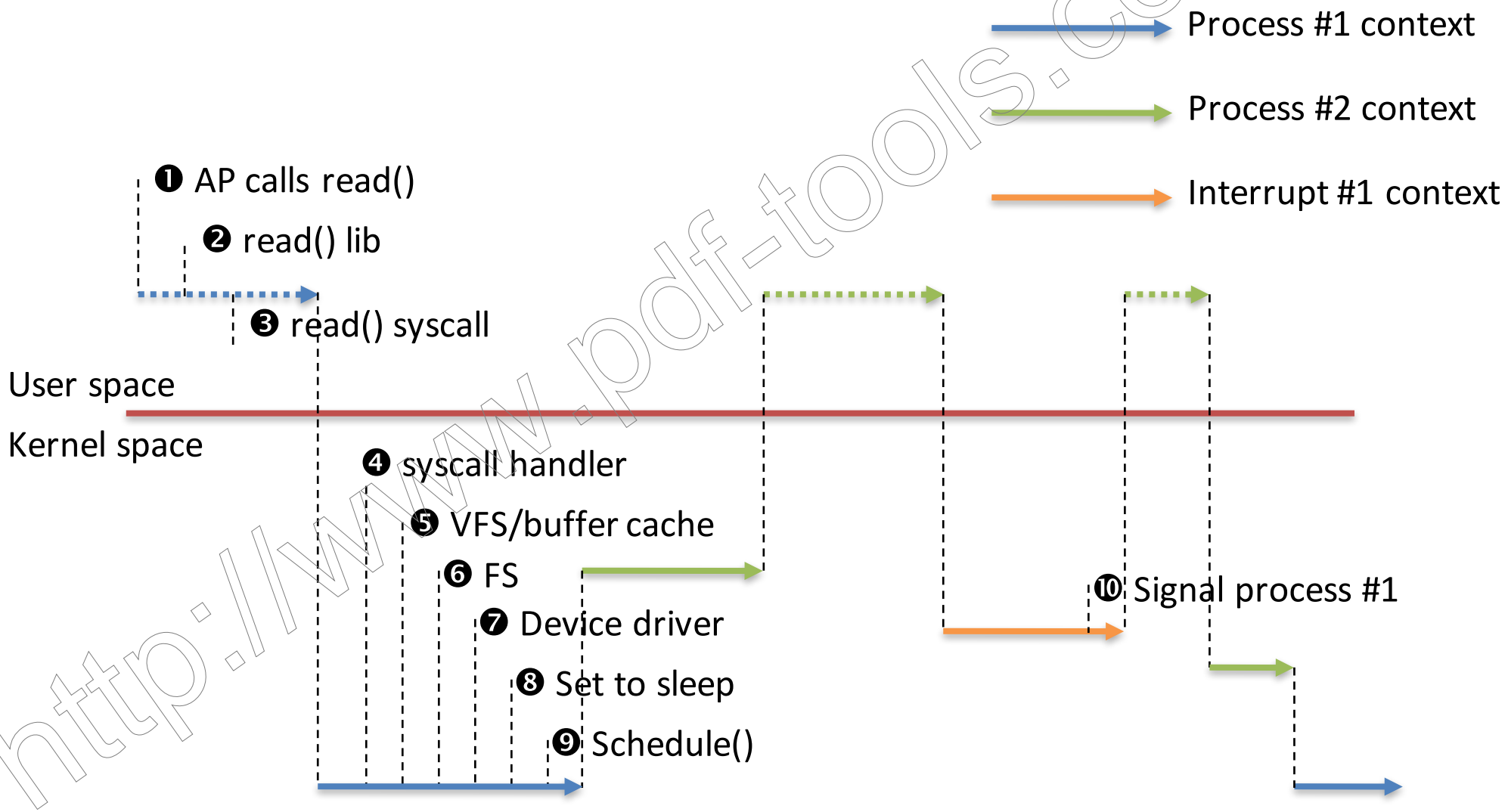
When an interrupt handler exits, before returning to kernel-space



If a task in the kernel explicitly calls schedule()

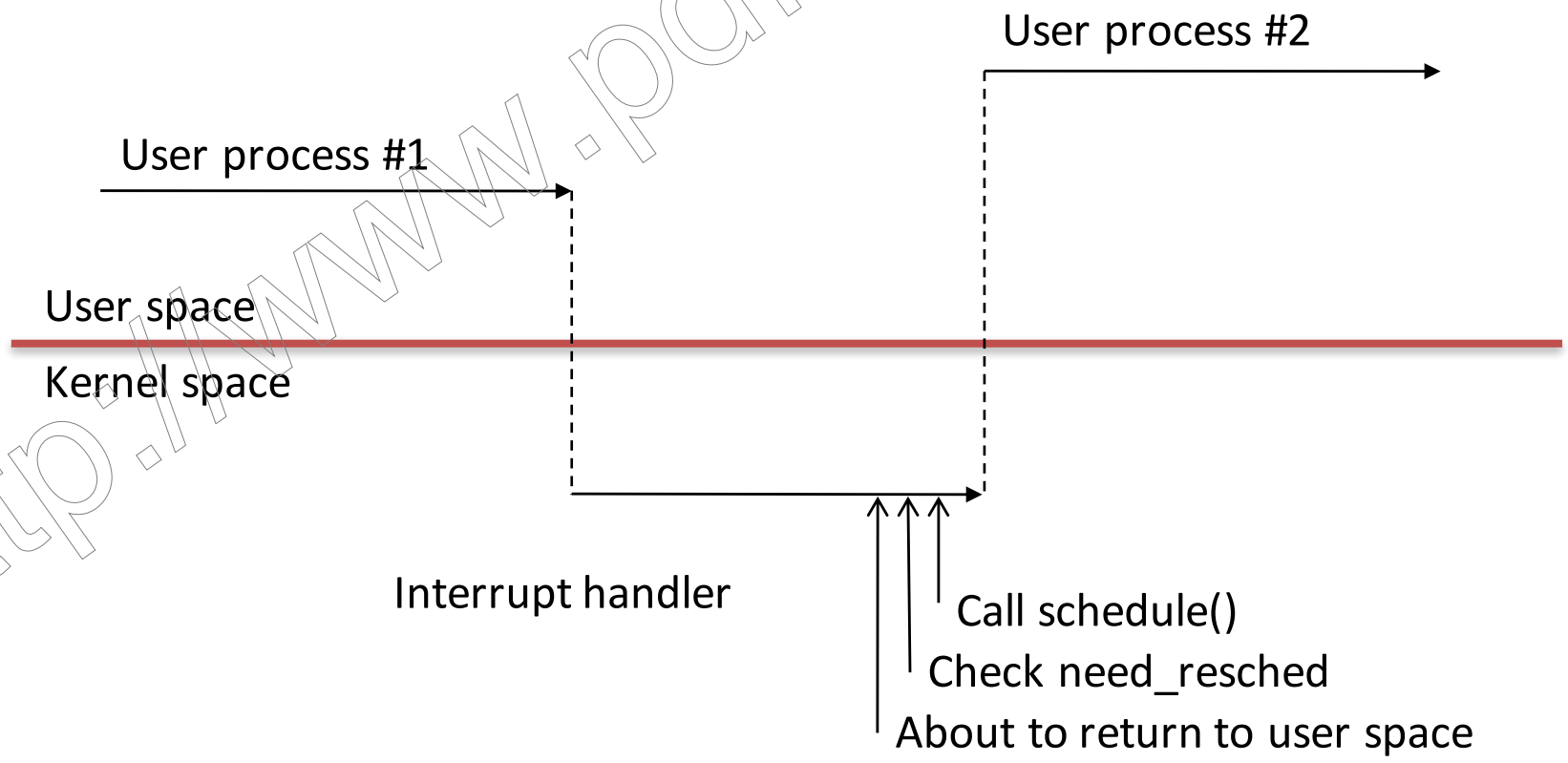


Process context + interrupt context



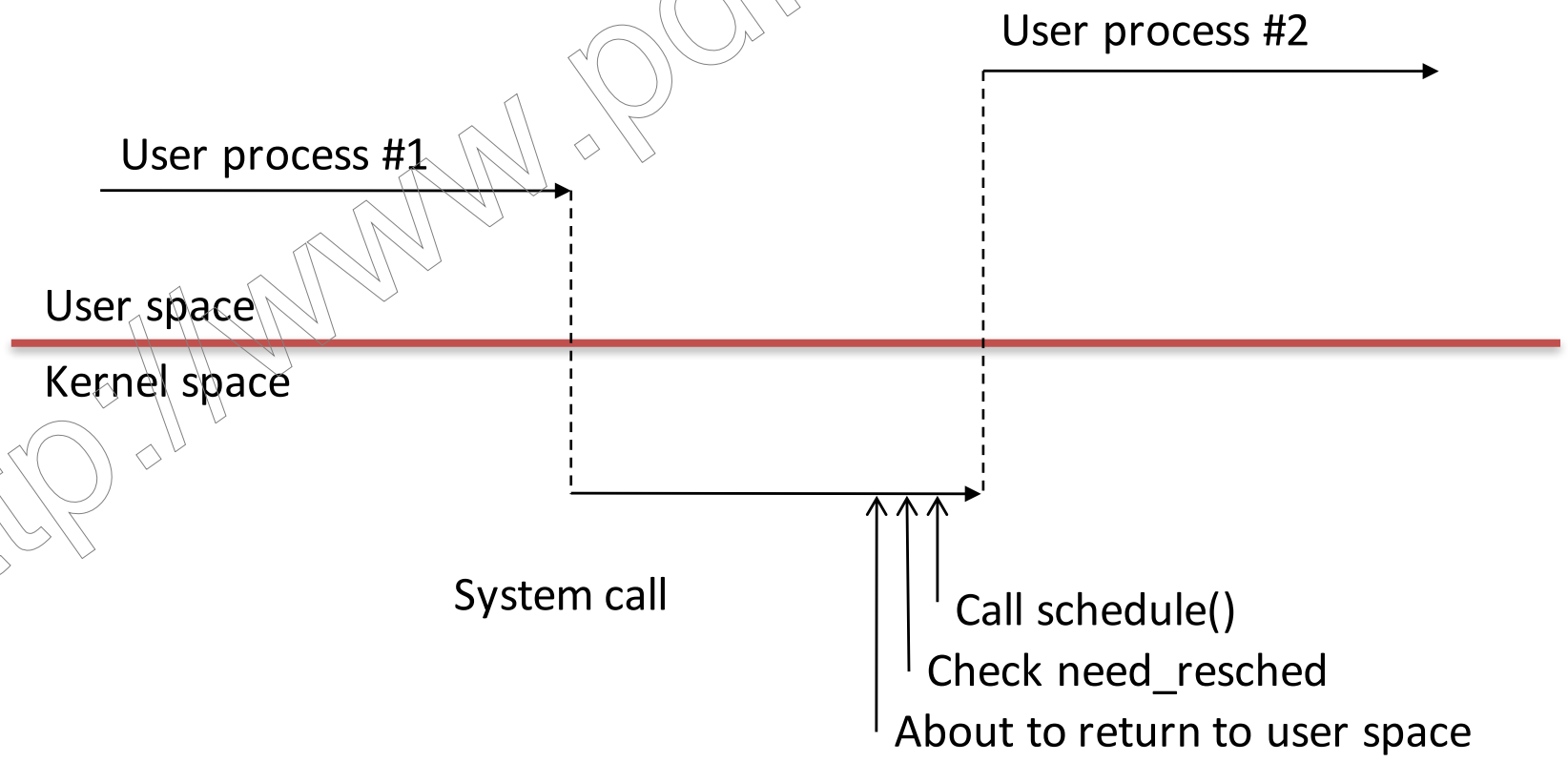
User preemption

- User preemption occurs when the kernel is in a safe state and about to return to user-space



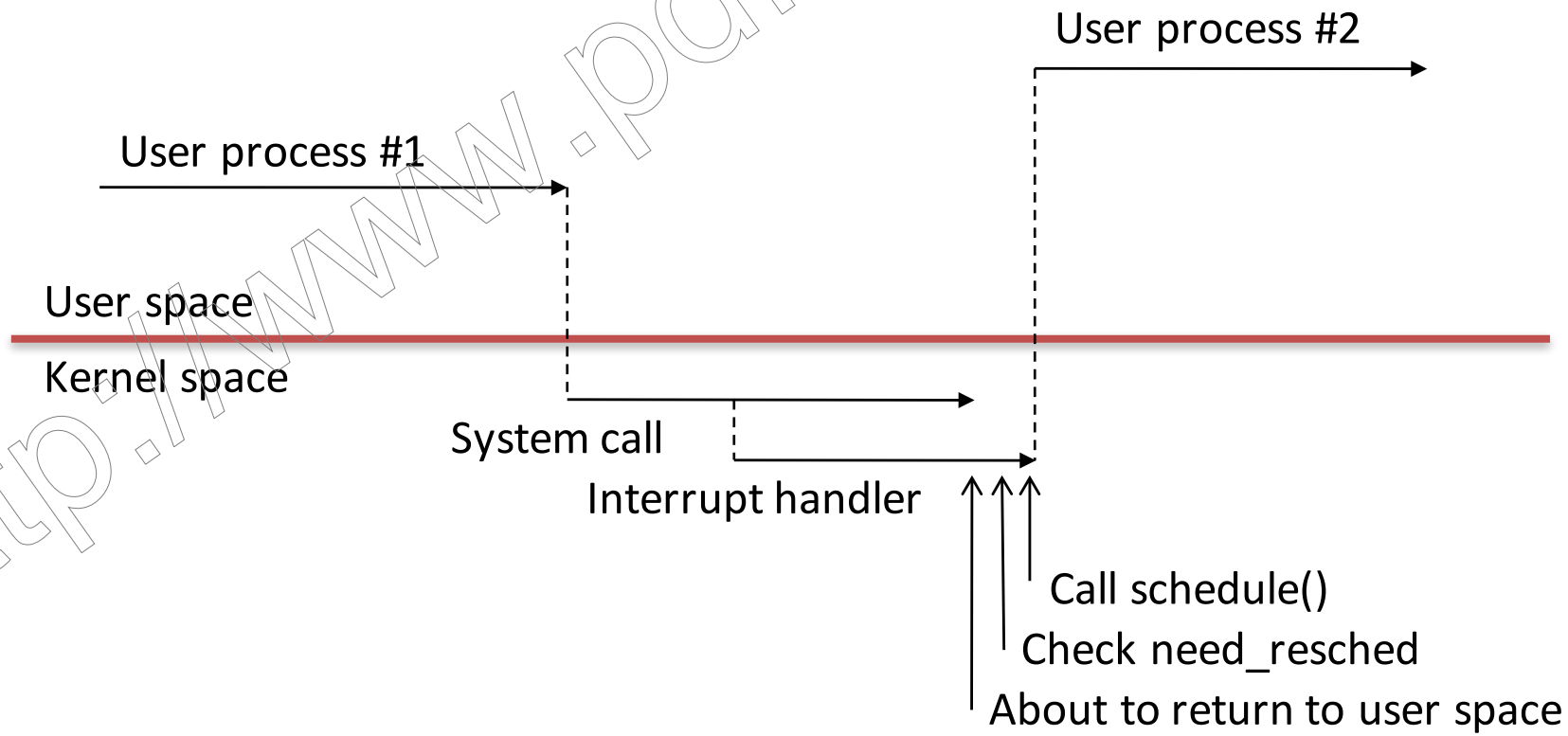
User preemption

- User preemption occurs when the kernel is in a safe state and about to return to user-space



Kernel preemption

- Linux kernel is possible to preempt a task at any point, so long as the kernel does not hold a lock



Preemptive Kernel

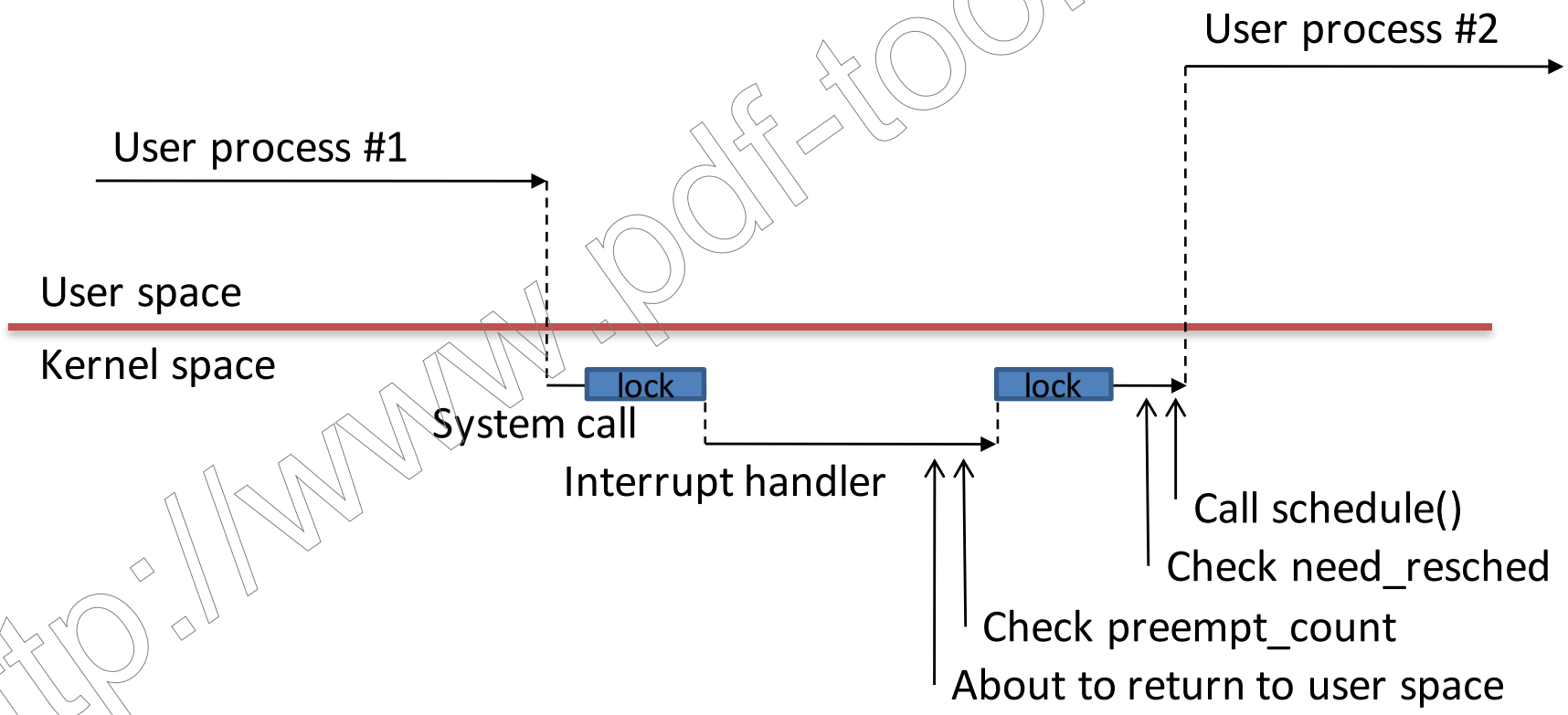
- Non-preemptive kernel supports user preemption
- Preemptive kernel supports kernel/user preemption

- Kernel can be interrupted ≠ kernel is preemptive
  - Non-preemptive kernel, interrupt returns to interrupted process
  - Preemptive kernel, interrupt returns to any schedulable process

Preemptive Kernel

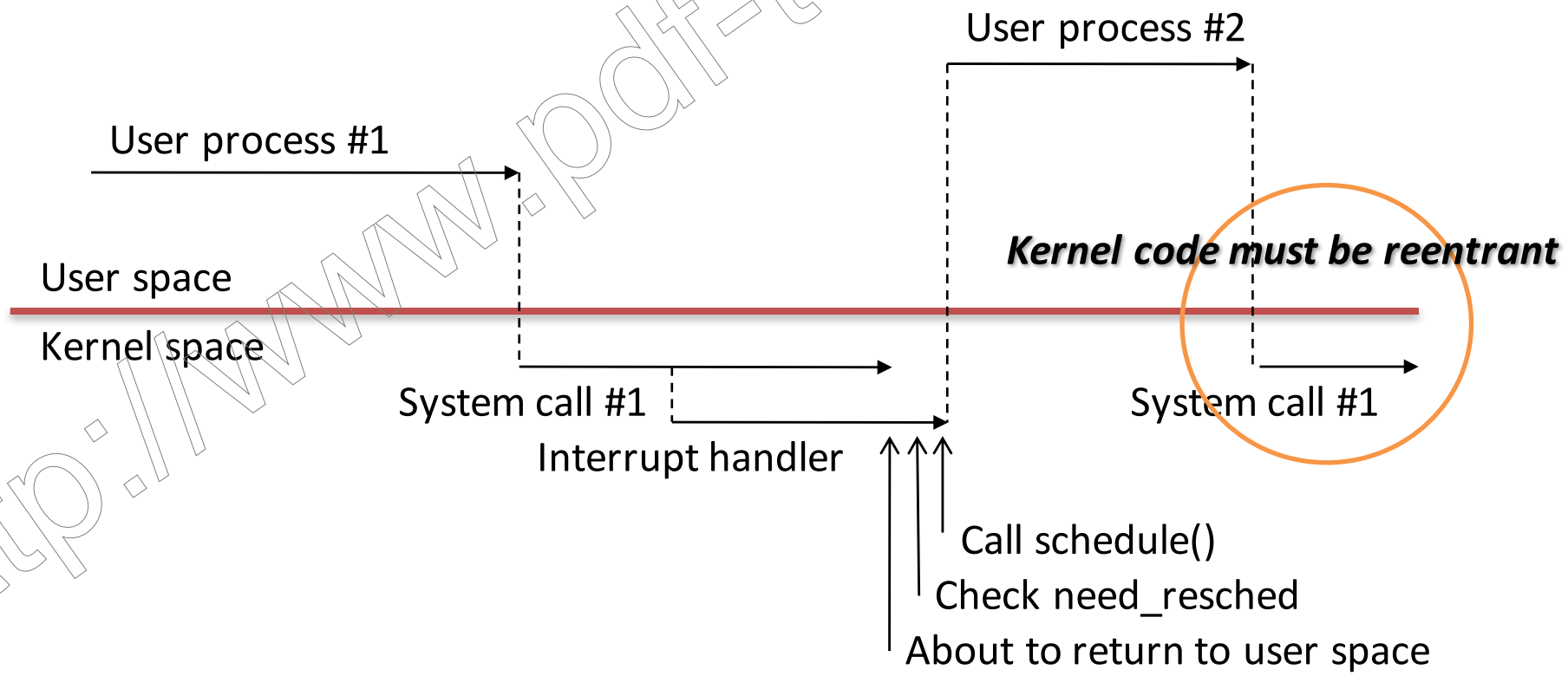
- 2.4 is a non-preemptive kernel
- 2.6 is a preemptive kernel
- 2.6 could disable CONFIG_PREEMPT

Preemptive Kernel

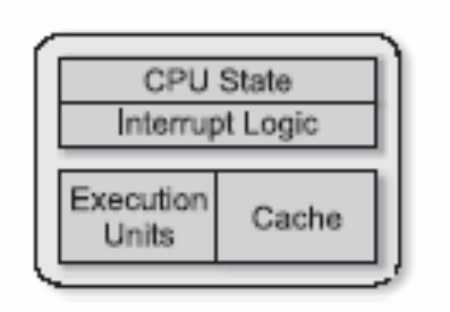


Preemptive Kernel

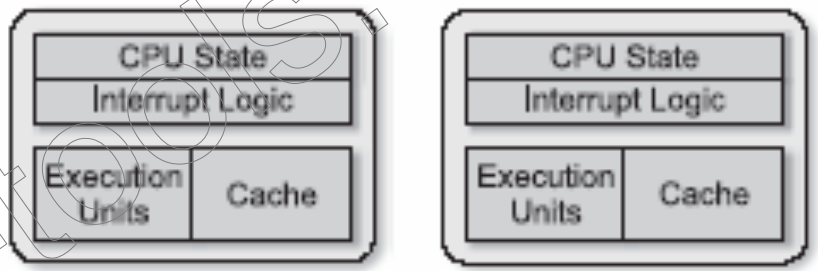
- How difficult to implement a preemptive kernel?



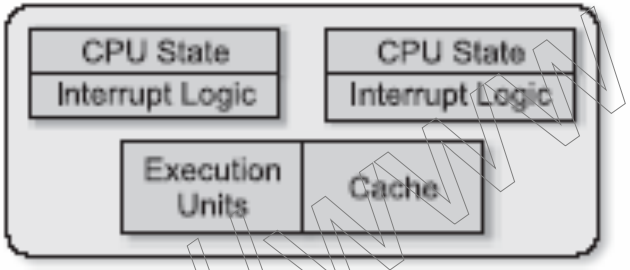
Single Core vs. Multi-core



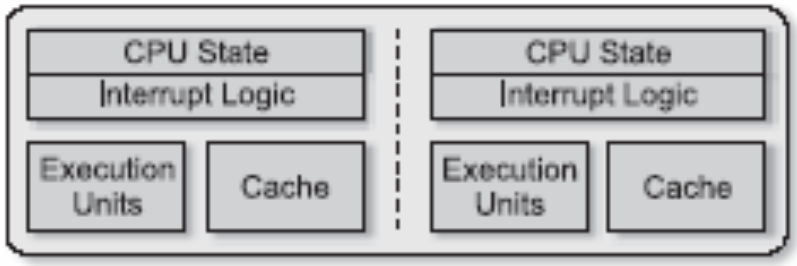
Single processor/single core



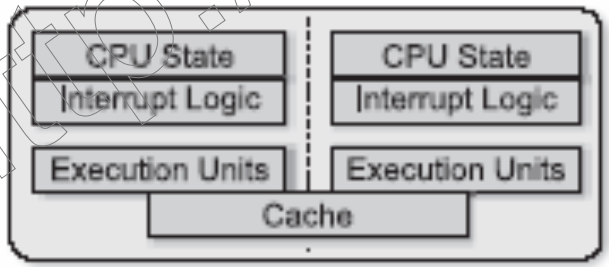
Multiple processor/single core



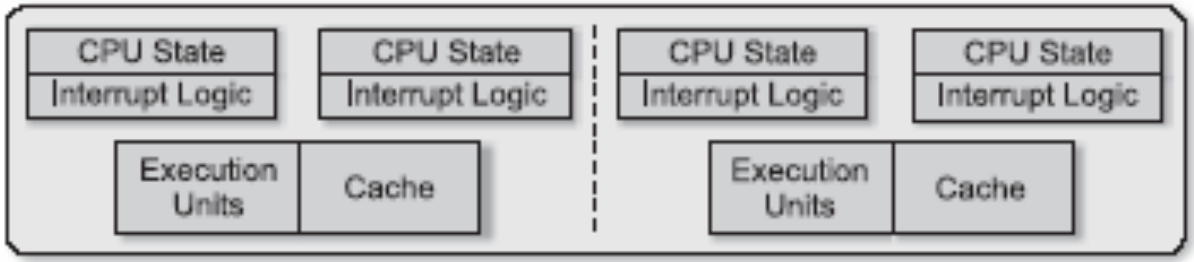
Single processor/single core/hyper threading



Single processor/multi core/separated cache

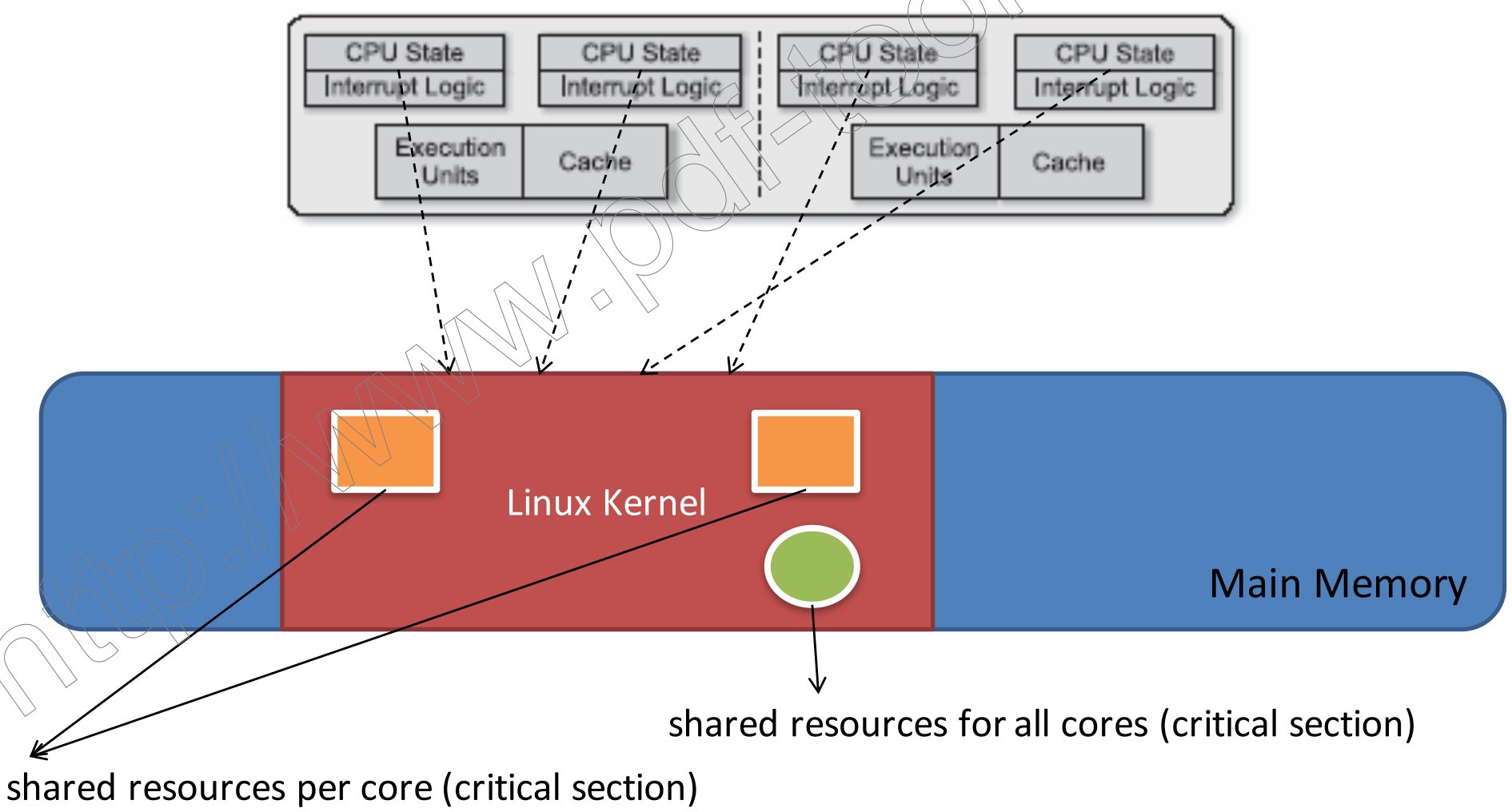


Single processor/multi core/shared cache



Single processor/multi core/hyper threading

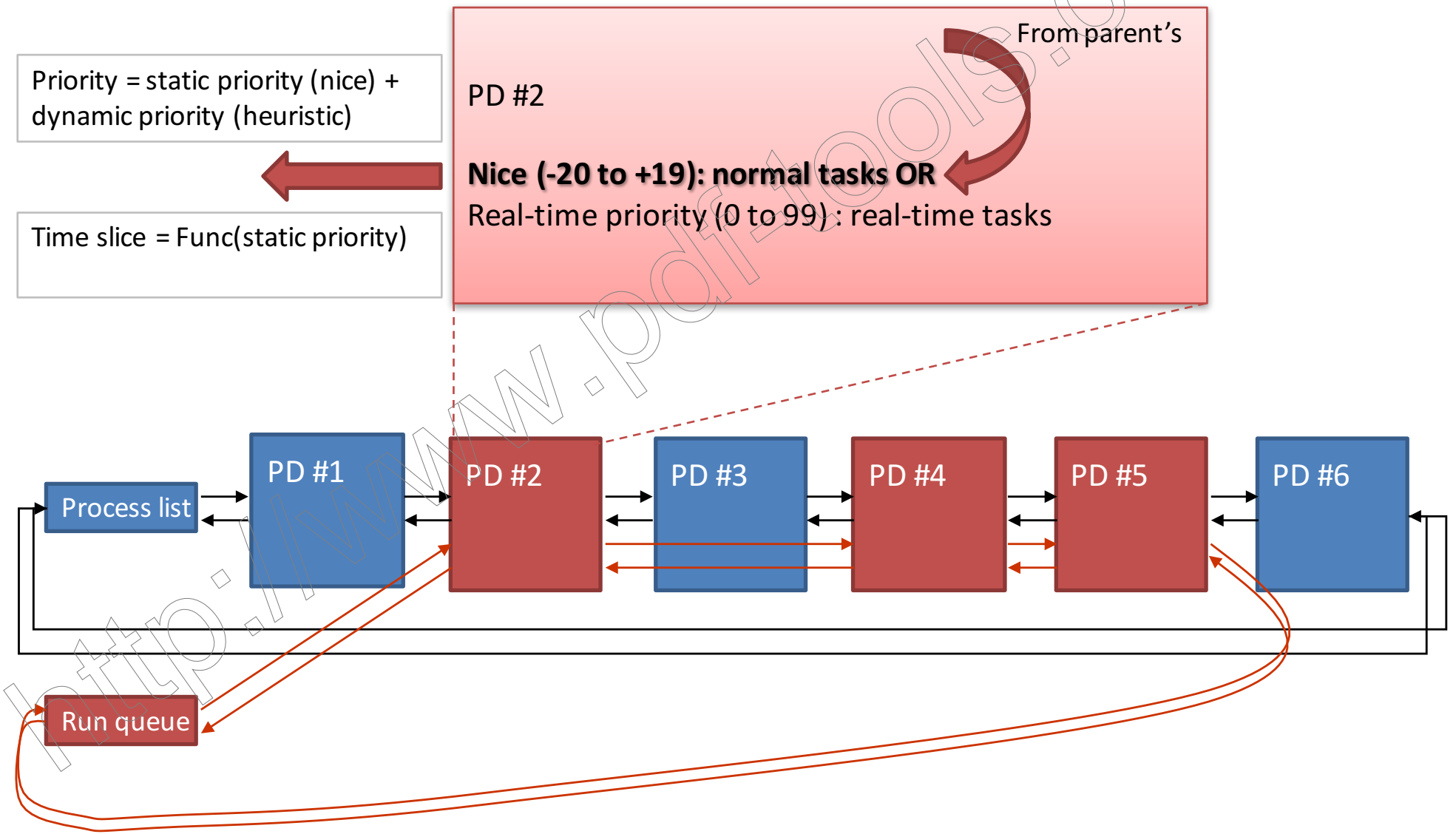
Single Core vs. Multi-core



Schedule Algorithms

- Think about yourself (your homework schedule)
  - Given that you have a lot of homework to do, each with a deadline
  - Profs. continue assigning new homework
  - What is the next homework to do? (next task to schedule)
  - Why should we stop a homework? (time to schedule)
  - How long can we concentrate on a homework? (scheduling period)
  - How long do we spend to determine the next homework? (scheduling algorithm overhead)
  - How much effort do we spend to switch homework? (context switch overhead)
  - What is the importance of a homework? (priority of a job)
  - How long does a homework need? (job length)

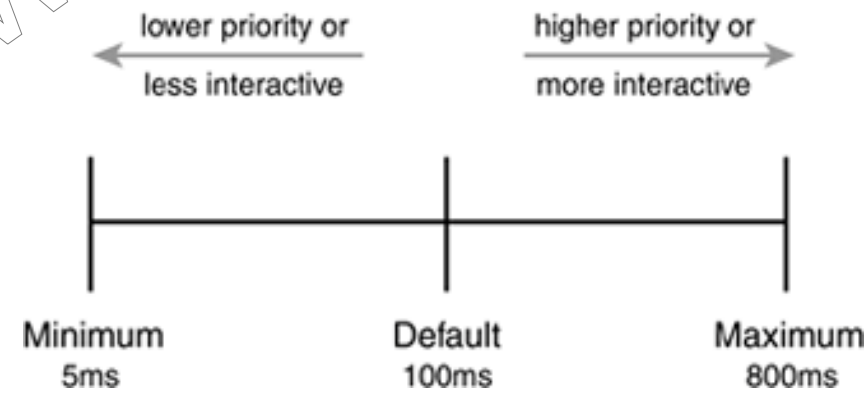
How Linux Scheduler Works



Timeslice

- Timeslice function

Type of Task	Nice Value	Timeslice Duration
Initially created	parent's	half of parent's
Minimum Priority	+19	5ms (MIN_TIMESLICE)
Default Priority	0	100ms (DEF_TIMESLICE)
Maximum Priority	-20	800ms (MAX_TIMESLICE)

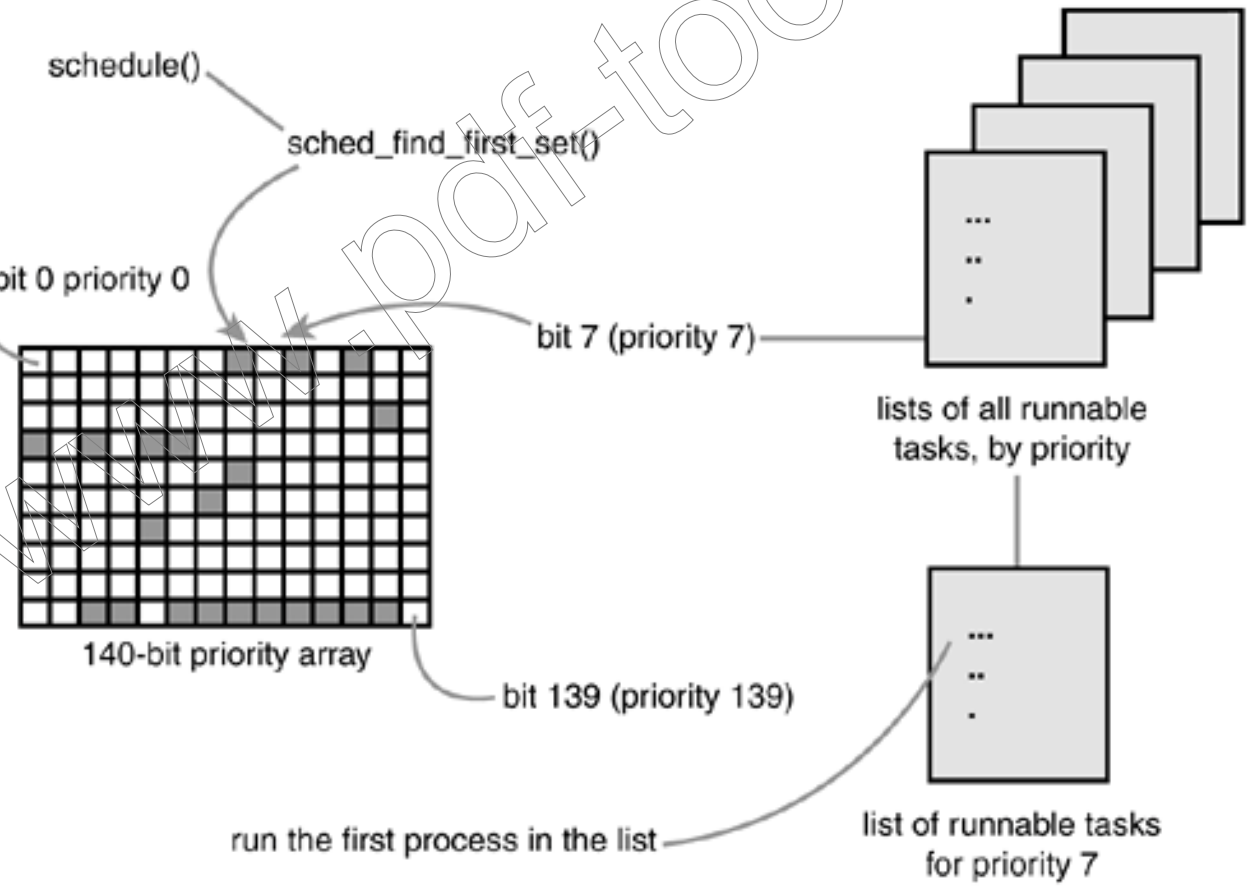


$$\text{base time quantum (in milliseconds)} = \begin{cases} (140 - \text{static priority}) \times 20 & \text{if static priority} < 120 \\ (140 - \text{static priority}) \times 5 & \text{if static priority} \geq 120 \end{cases} \quad (1)$$

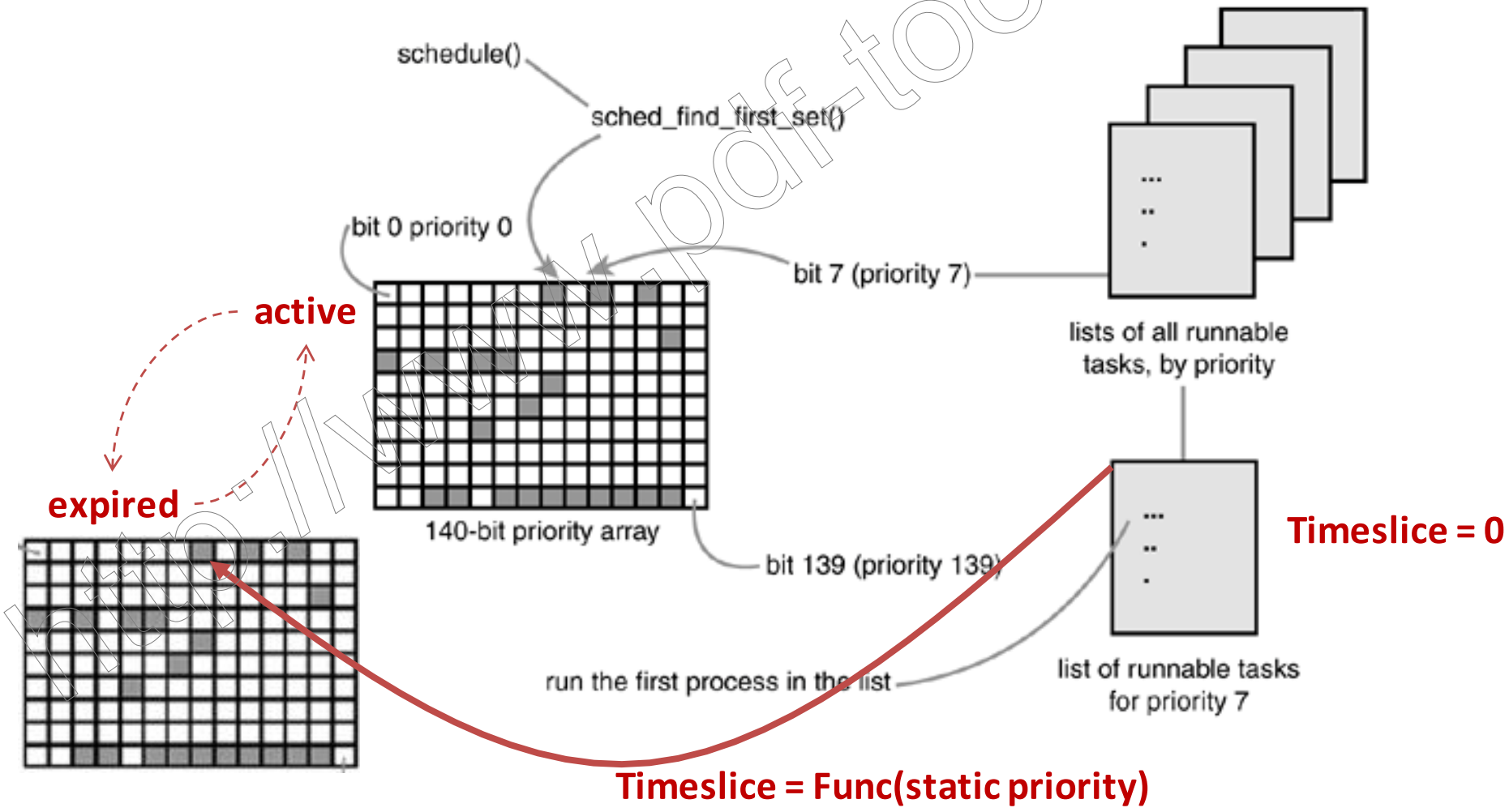
Process schedule and context switching in Linux

- Priority-based scheduler
- Dynamic priority-based scheduling
  - Dynamic priority
    - Normal process
      - nice value: -20 to +19 (larger nice values imply you are being nice to others)
  - Static priority
    - Real-time process
      - 0 to 99
  - Total priority: 140

Linux O(1) cheduler



Recalculating timeslice



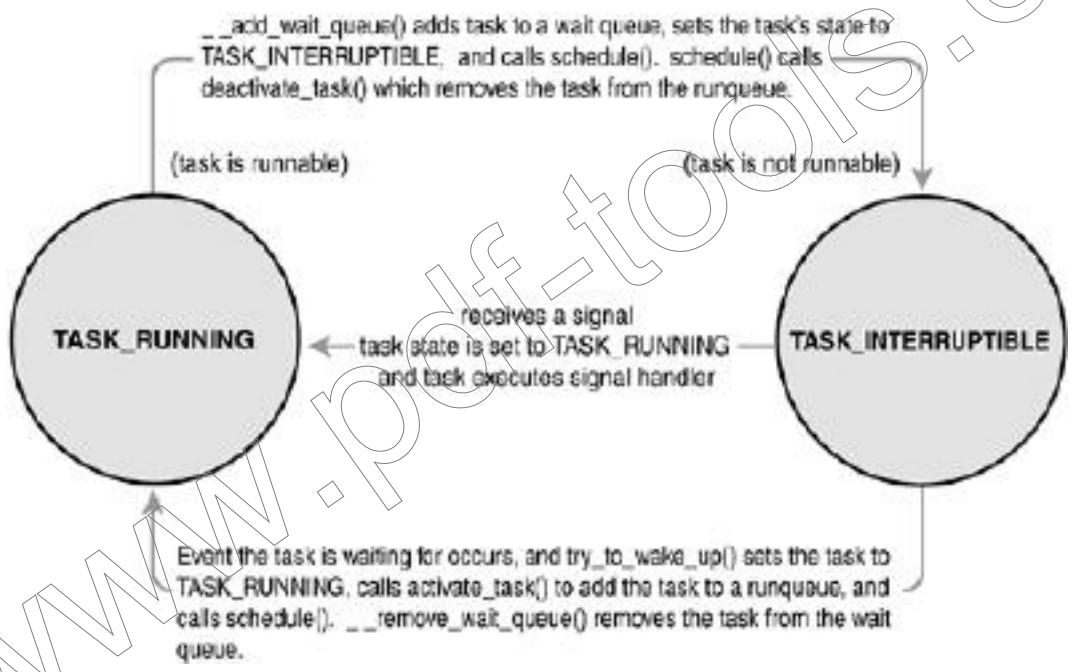
Calculating Priority

- static_prio = nice
- Prio = nice – bonus + 5

  dynamic priority = max (100, min (static priority - bonus + 5, 139))

- Heuristic
  - sleep_avg: (0 to MAX_SLEEP_AVG(10ms))
  - sleep_avg+=sleep (becomes runnable)
  - Sleep_avg-=run (every time tick when task runs)

Sleeping and waking up

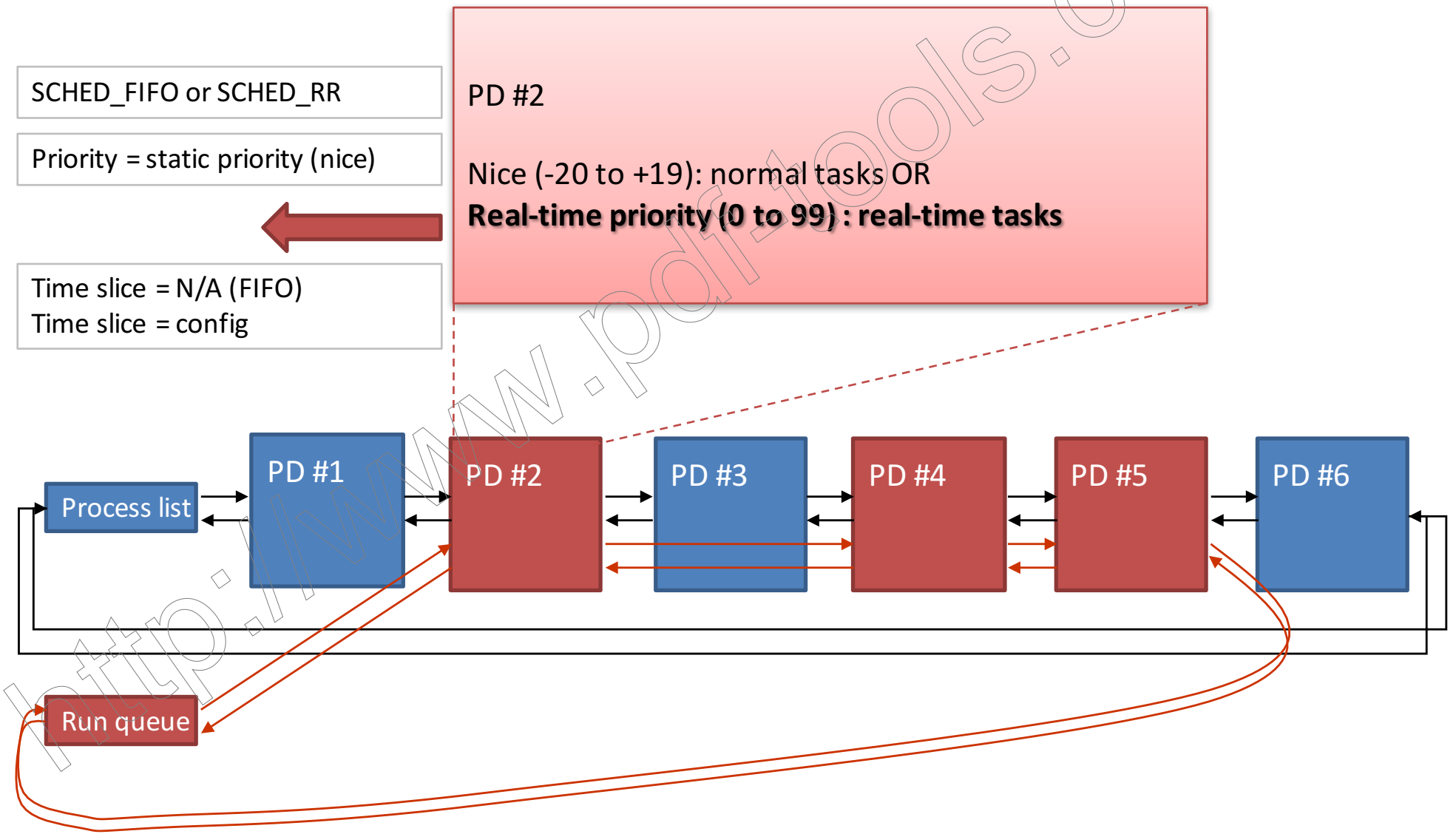


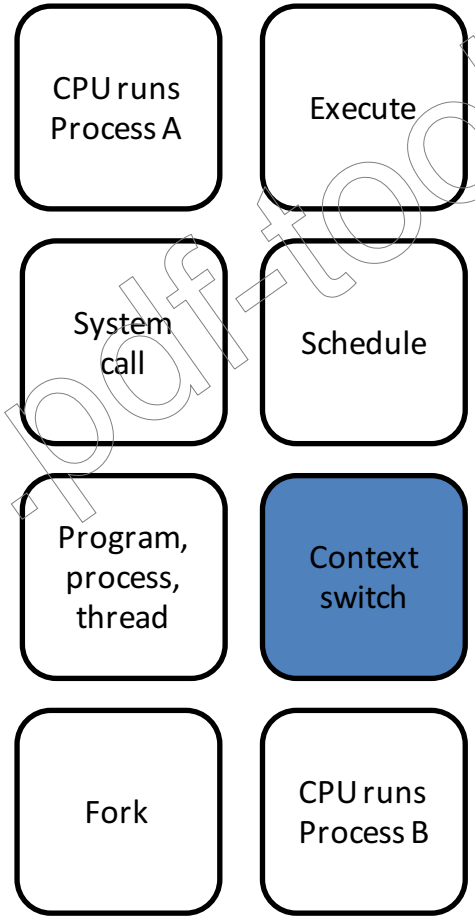

```
add_wait_queue(q, &wait);
while (!condition) {     /* condition is the event that we are waiting for */
      set_current_state(TASK_INTERRUPTIBLE); /* or TASK_UNINTERRUPTIBLE */
      if (signal_pending(current))
                /* handle signal */
      schedule();
}
set_current_state(TASK_RUNNING);
remove_wait_queue(q, &wait);
```

System calls related to scheduling

System call	Description
nice( )	Change the static priority of a conventional process
getpriority( )	Get the maximum static priority of a group of conventional processes
setpriority( )	Set the static priority of a group of conventional processes
sched_getscheduler( )	Get the scheduling policy of a process
sched_setscheduler( )	Set the scheduling policy and the real-time priority of a process
sched_getparam( )	Get the real-time priority of a process
sched_setparam( )	Set the real-time priority of a process
sched_yield( )	Relinquish the processor voluntarily without blocking
sched_get_ priority_min( )	Get the minimum real-time priority value for a policy
sched_get_ priority_max( )	Get the maximum real-time priority value for a policy
sched_rr_get_interval( )	Get the time quantum value for the Round Robin policy
sched_setaffinity( )	Set the CPU affinity mask of a process
sched_getaffinity( )	Get the CPU affinity mask of a process

How Linux Scheduler Works

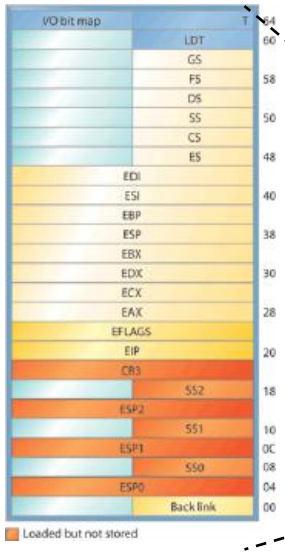


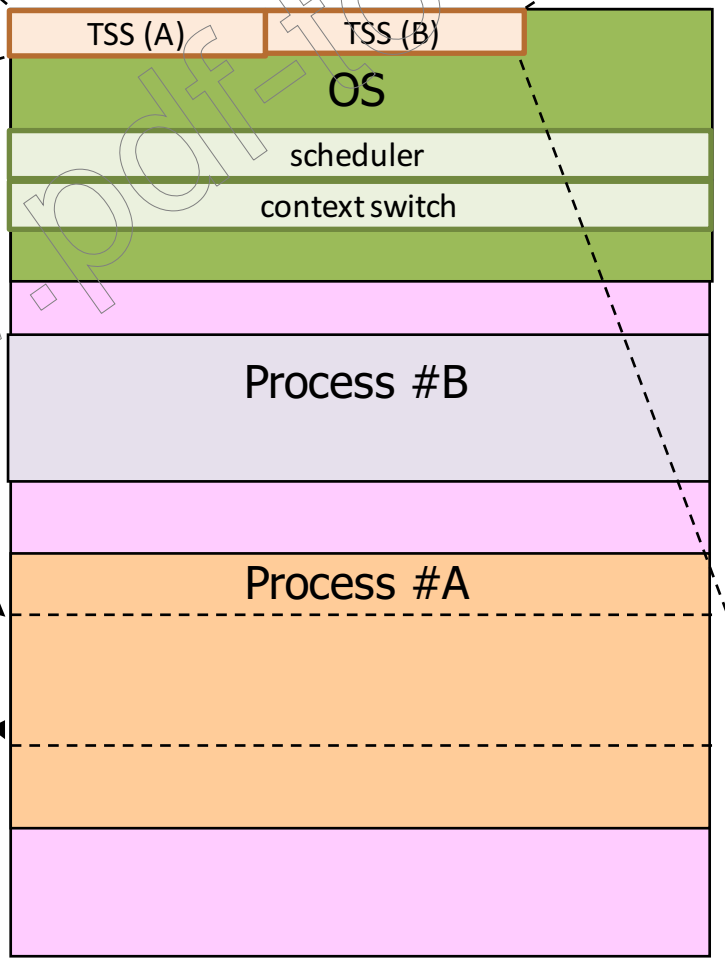


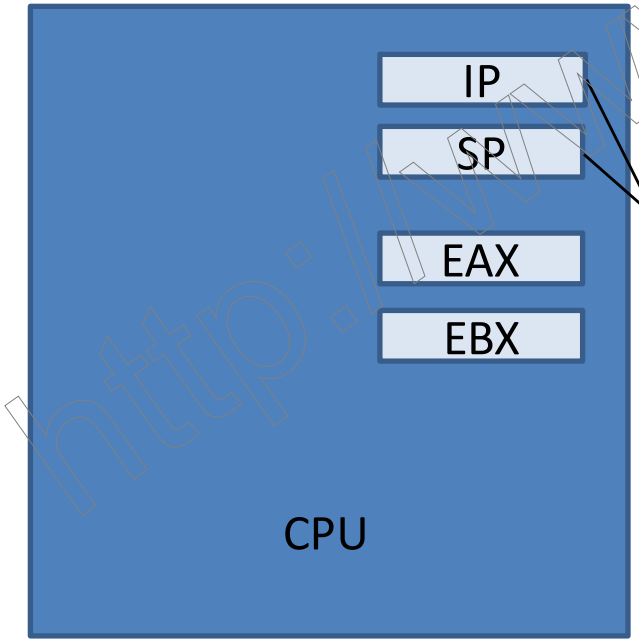
Process schedule and context switching in Linux

- Context switch
  - Hardware context switch
    - Task State Segment Descriptor (Old Linux)
  - Step by step context switch
    - Better control and optimize
- Context switch
  - switch_mm()
    - Switch virtual memory mapping
  - switch_to()
    - Switch processor state
- Process switching occurs only in kernel mode
- The contents of all registers used by a process in User Mode have already been saved

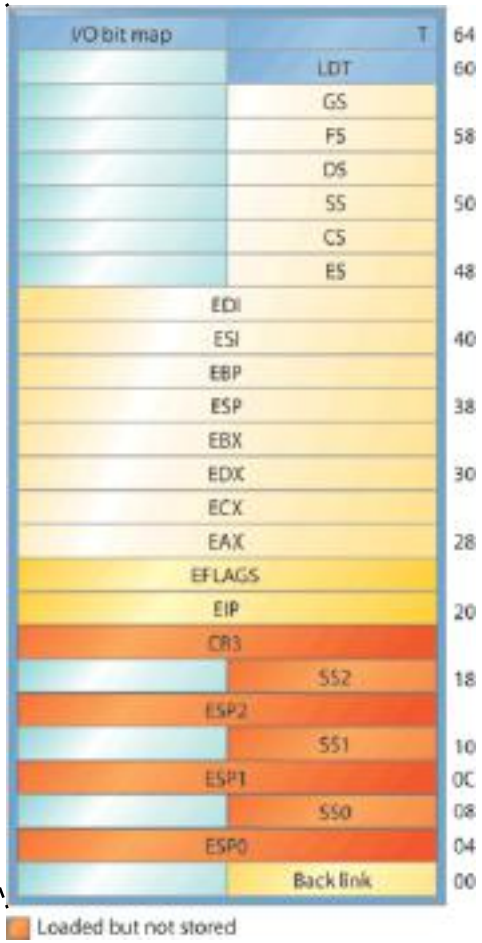
Scheduler in Details



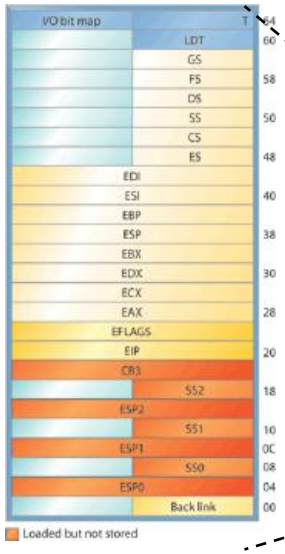


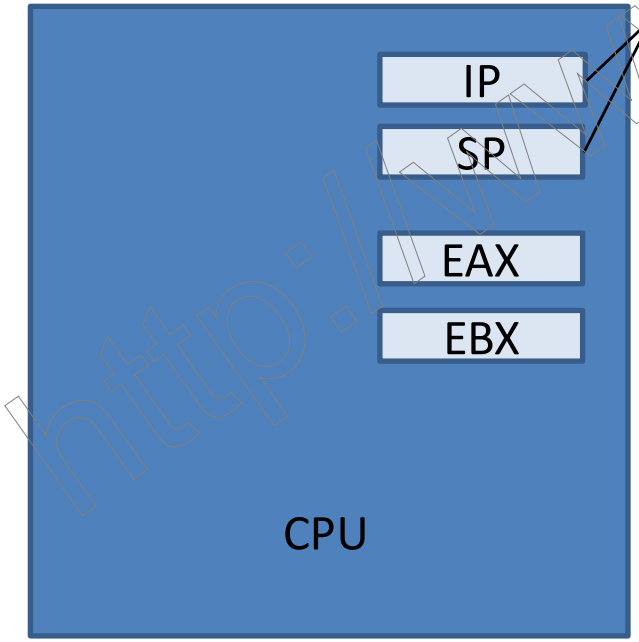


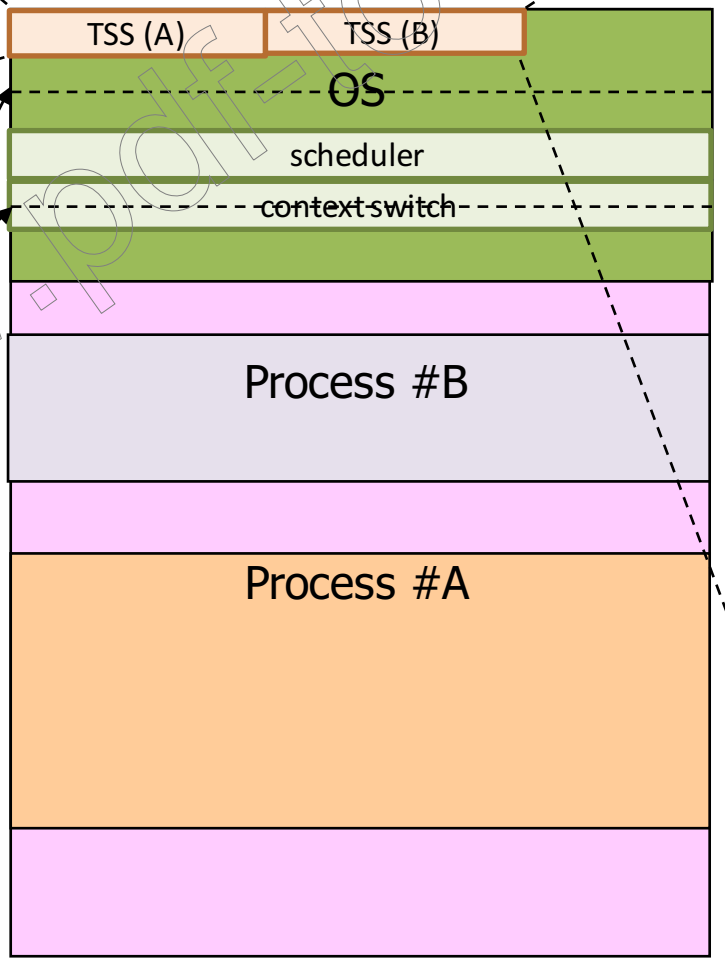
Task state segment



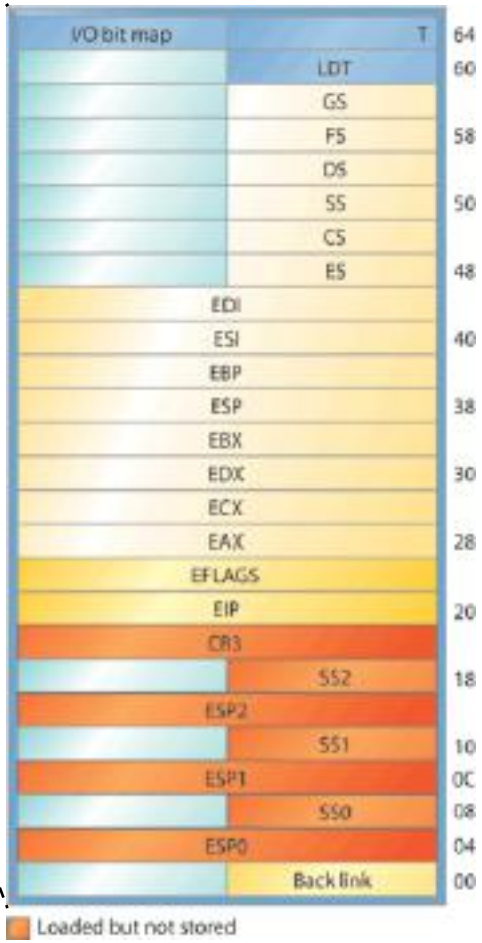
Scheduler in Details



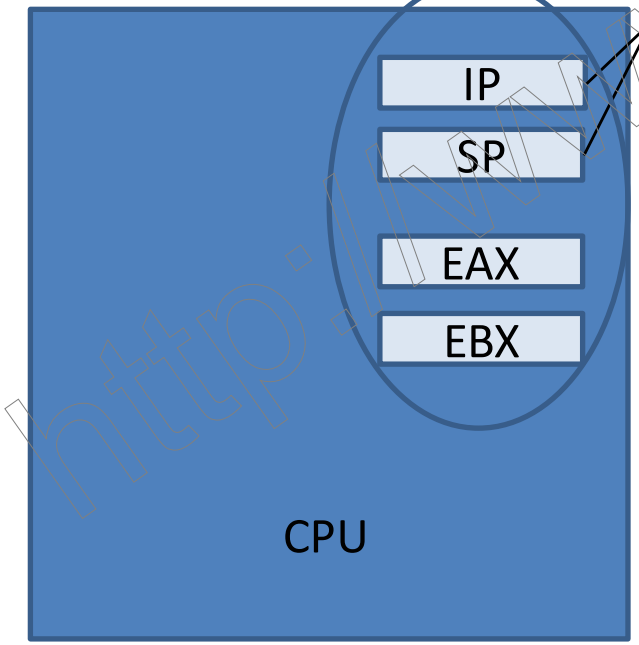


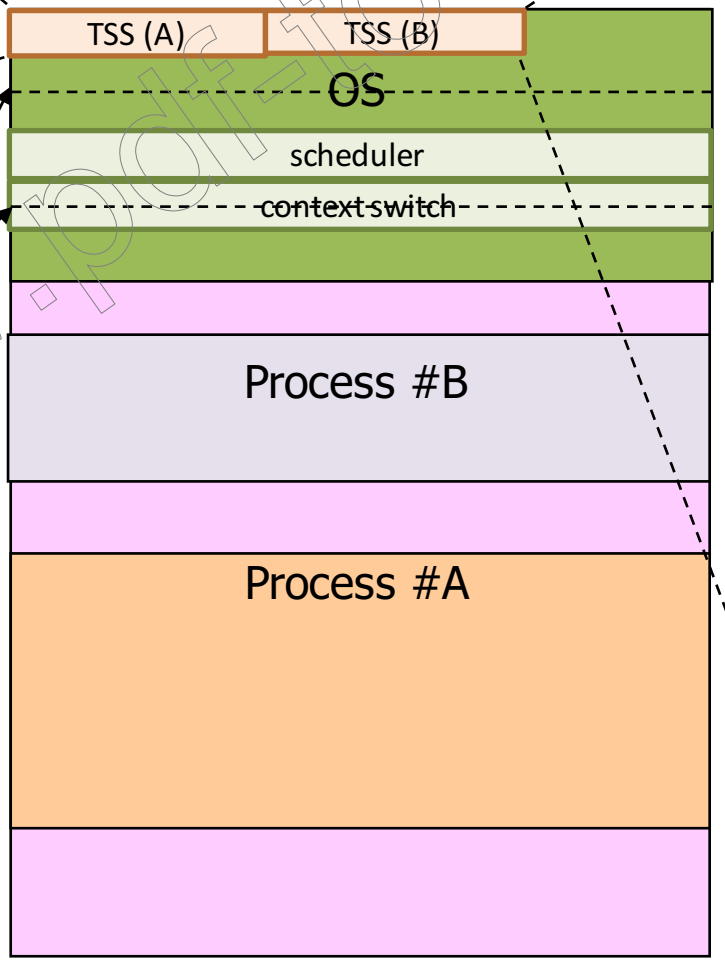


Task state segment

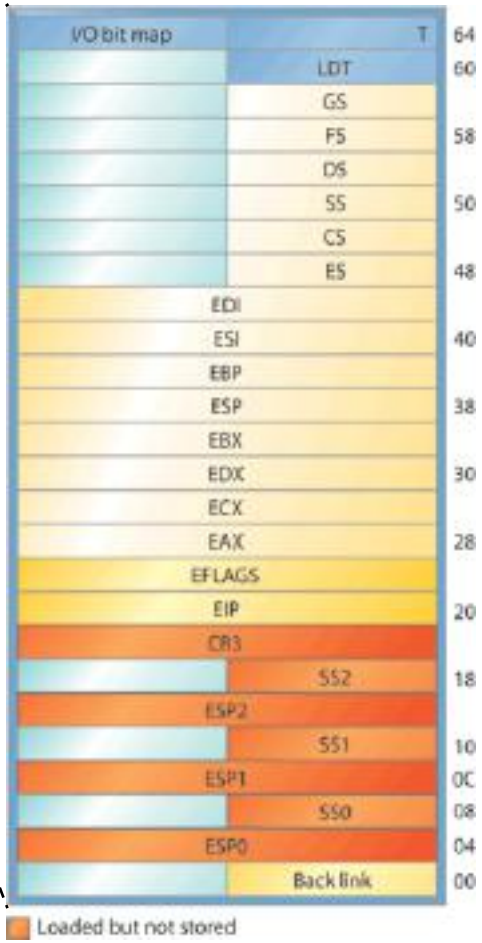


Scheduler in Details

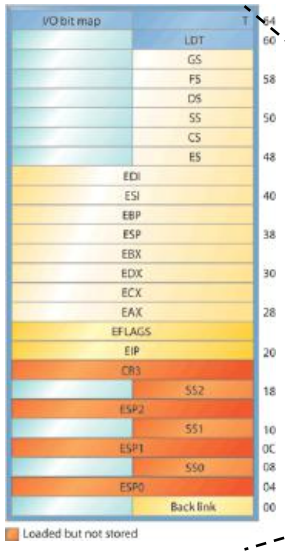


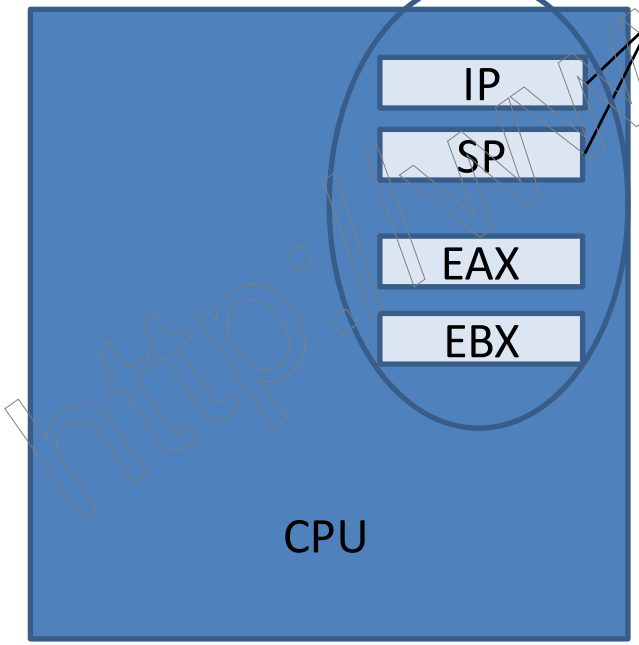


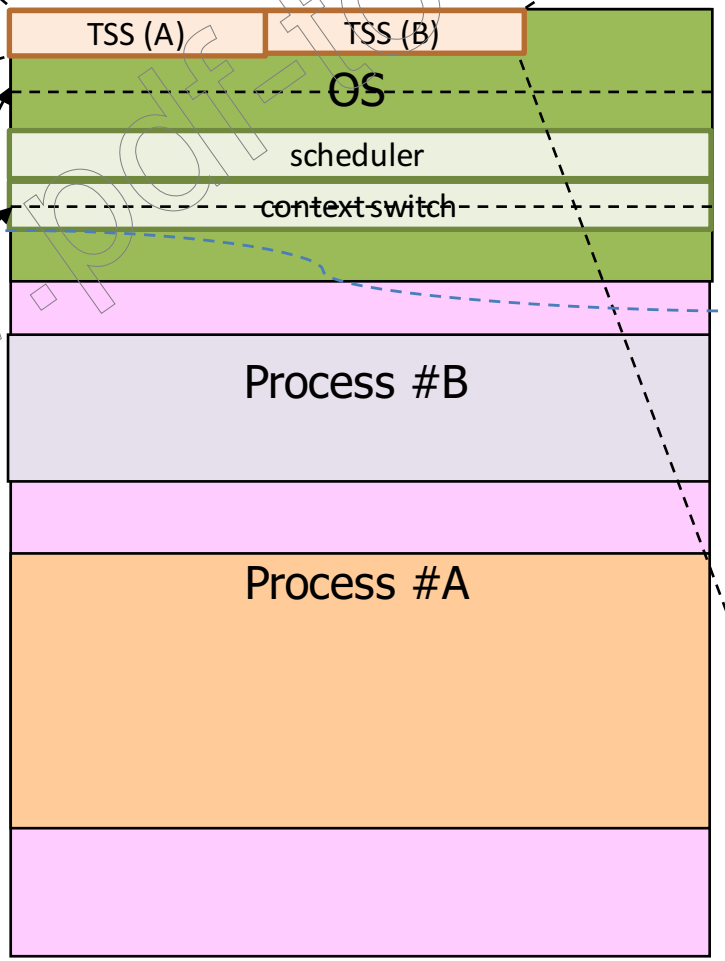
Task state segment



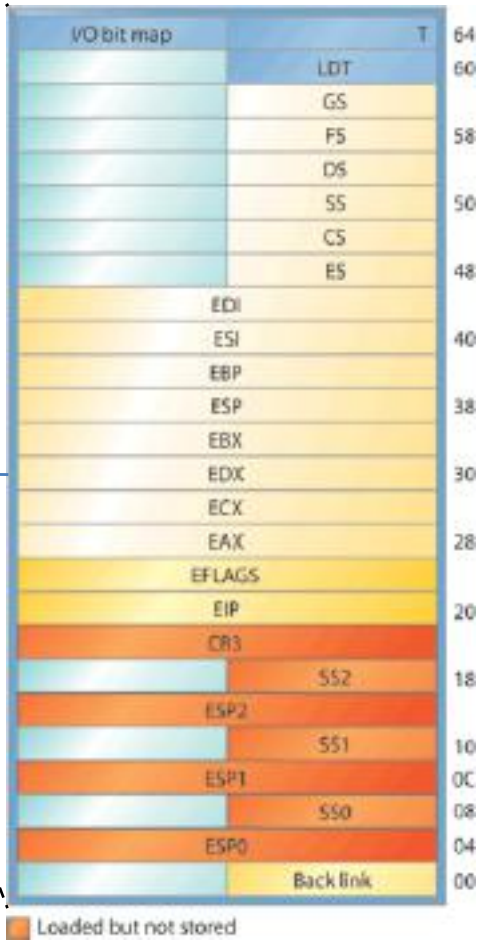
Scheduler in Details



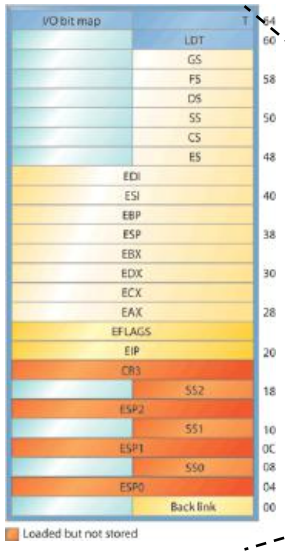


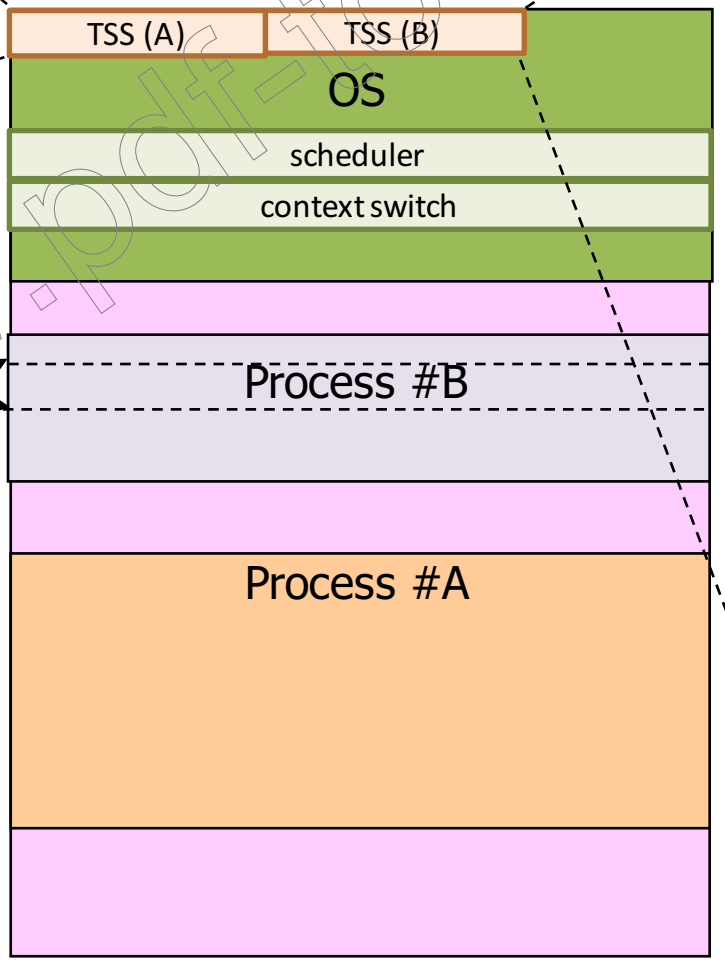


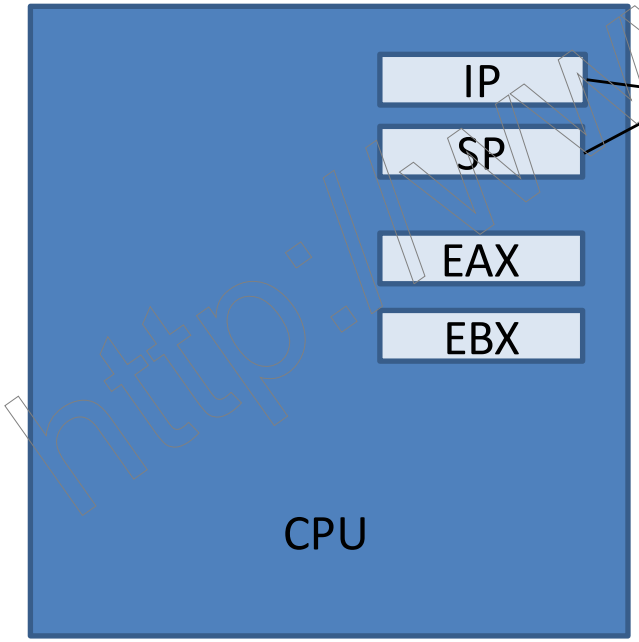
Task state segment



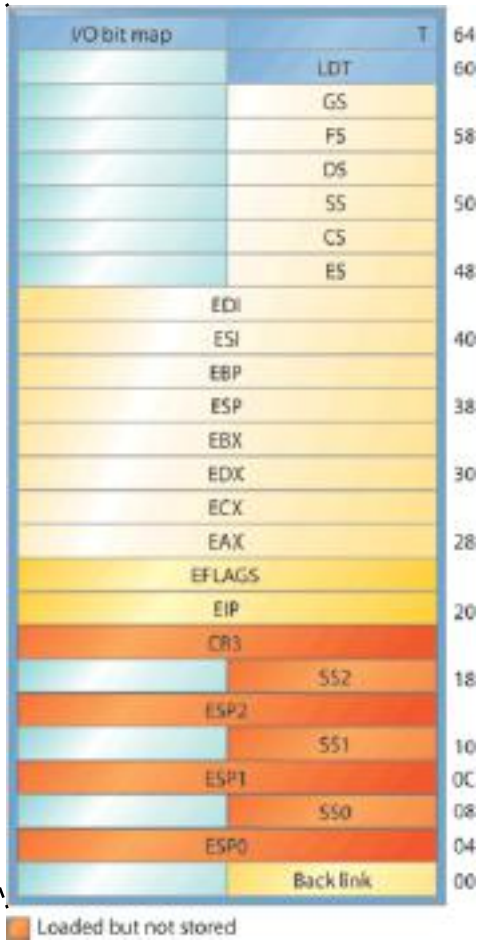
Scheduler in Details



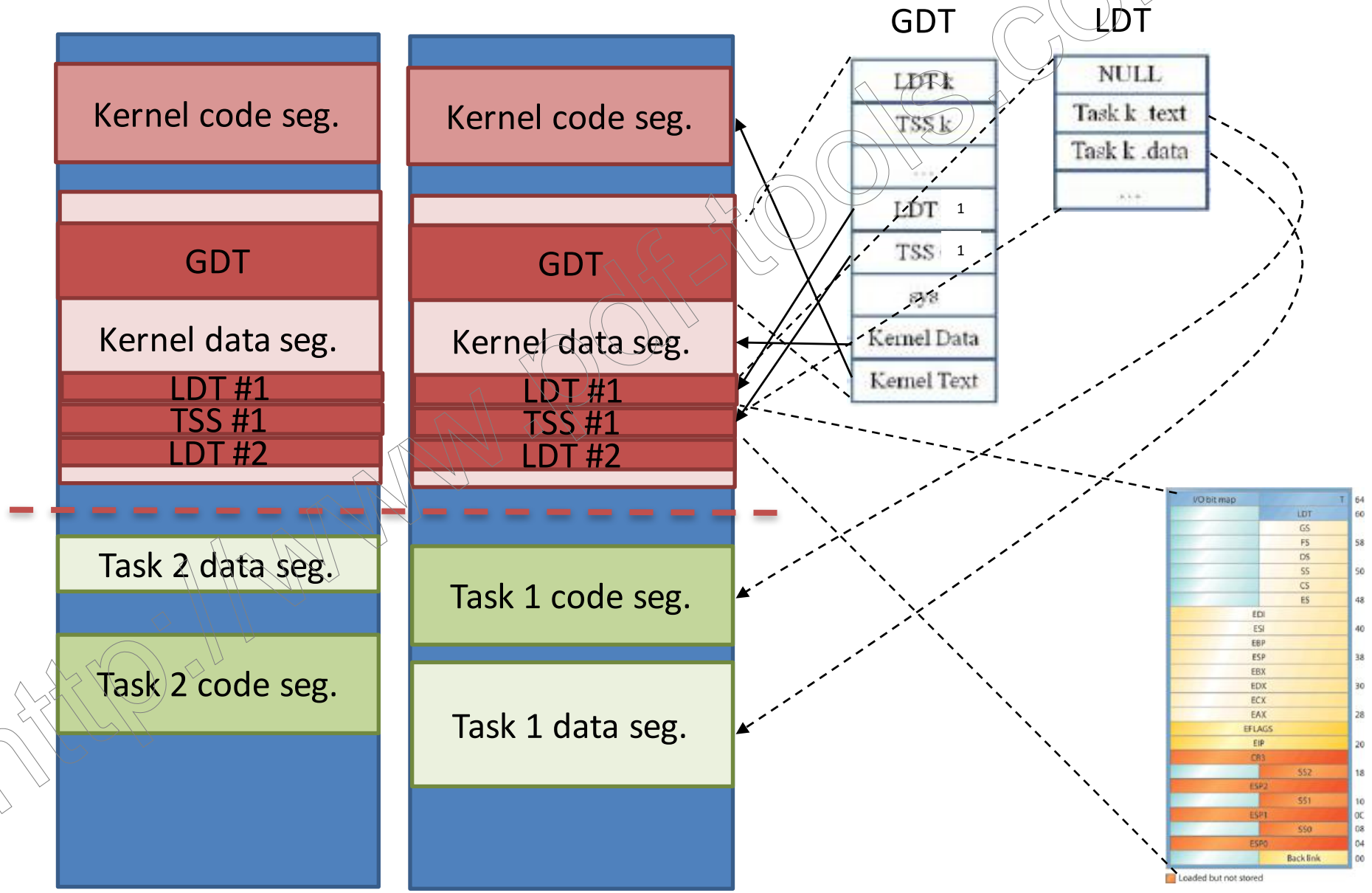




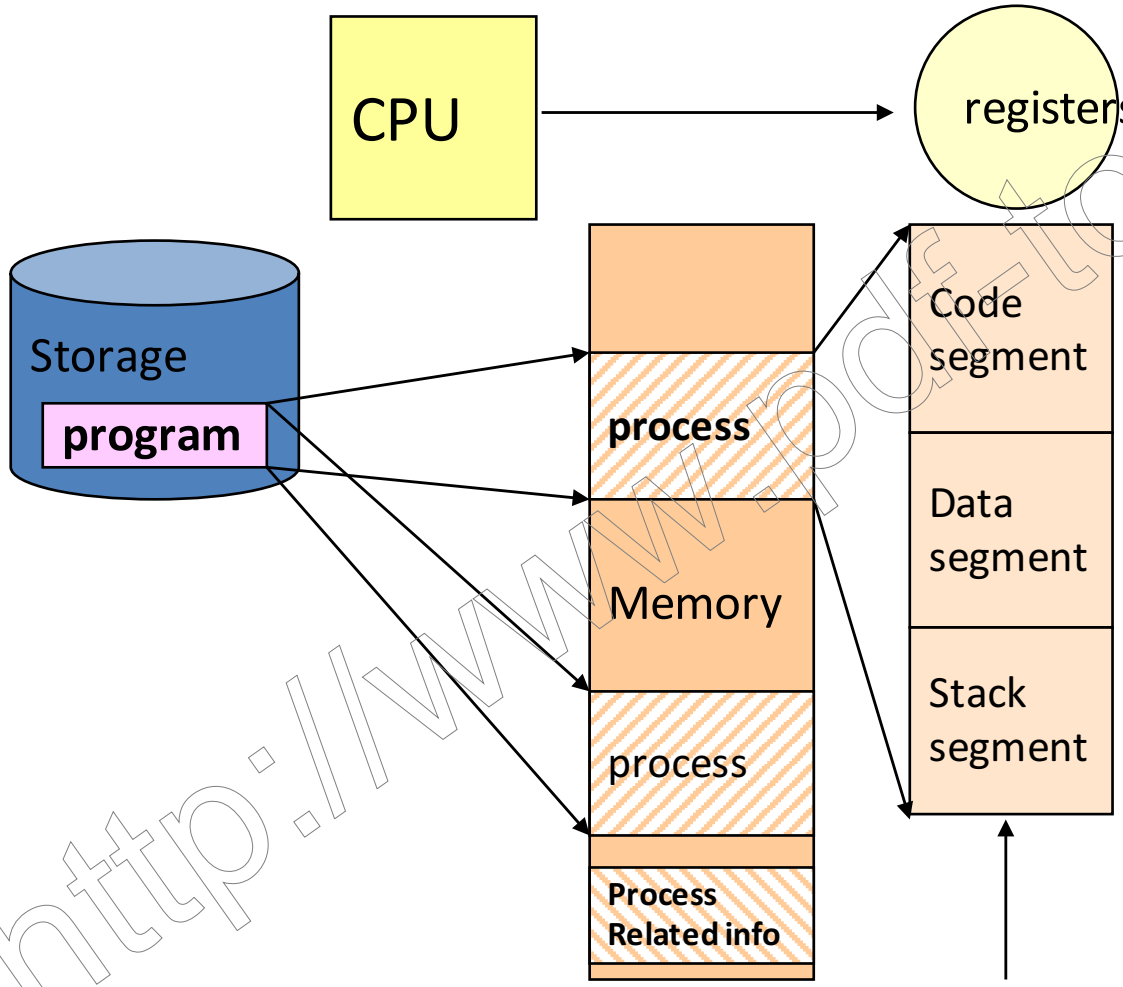
Task state segment



How x86 helps in Context Switch



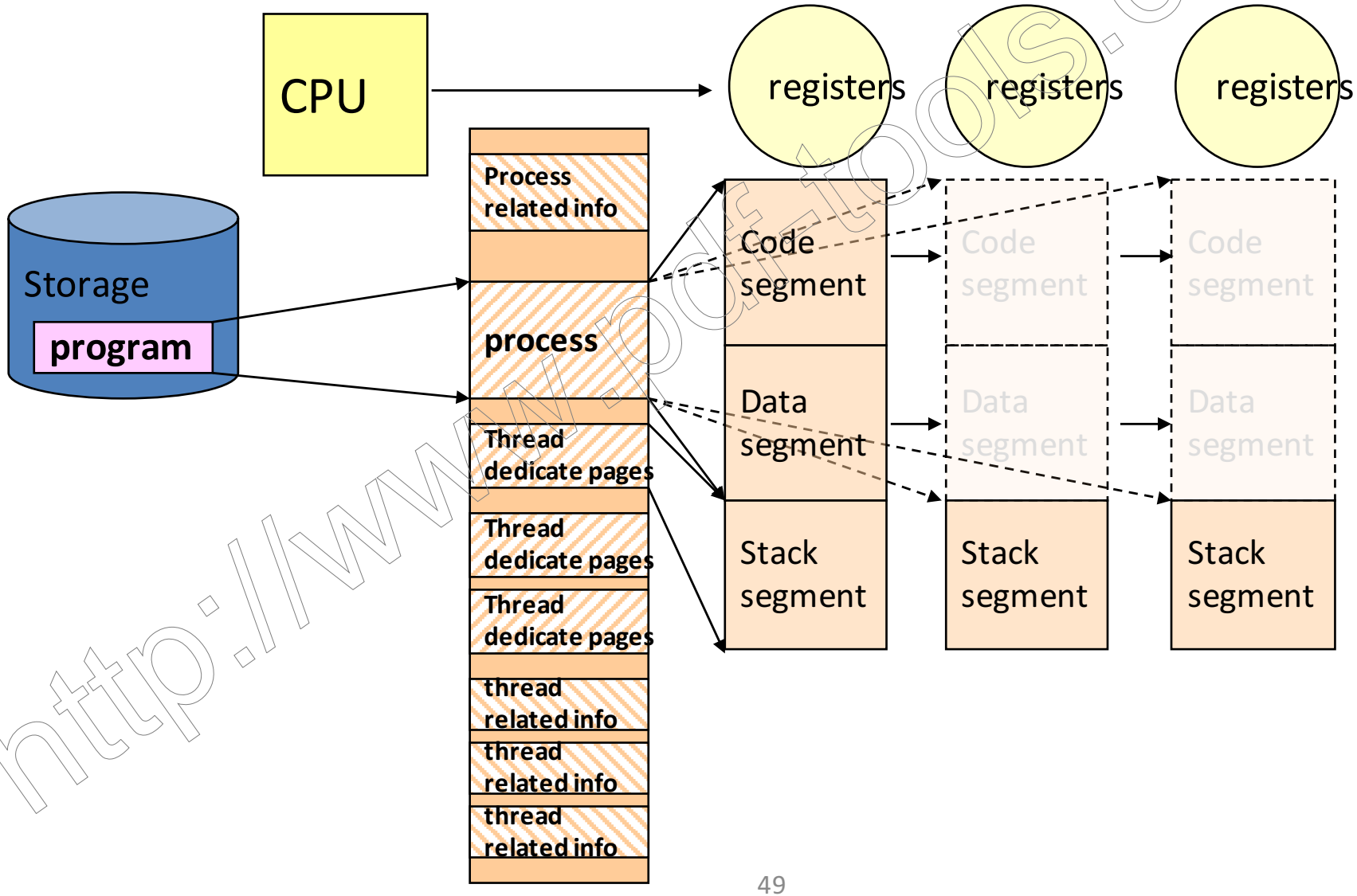
Process related terms



Is that good enough ?
If not, why ?

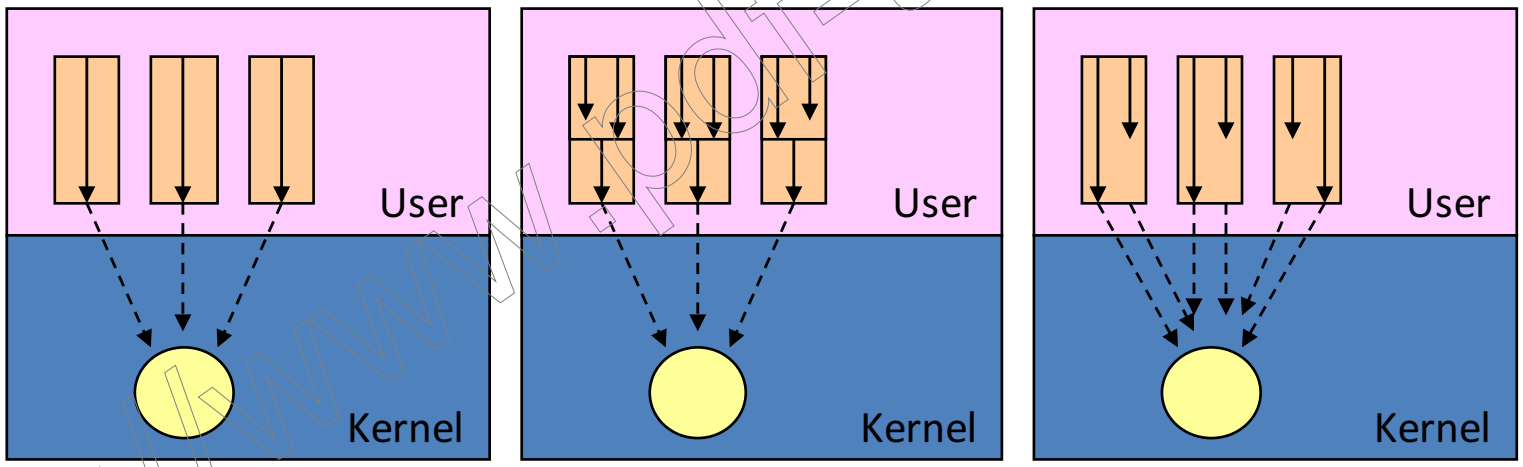
Physical memory might be discontinuous

Process related terms (Cont.)



Process related terms (Cont.)

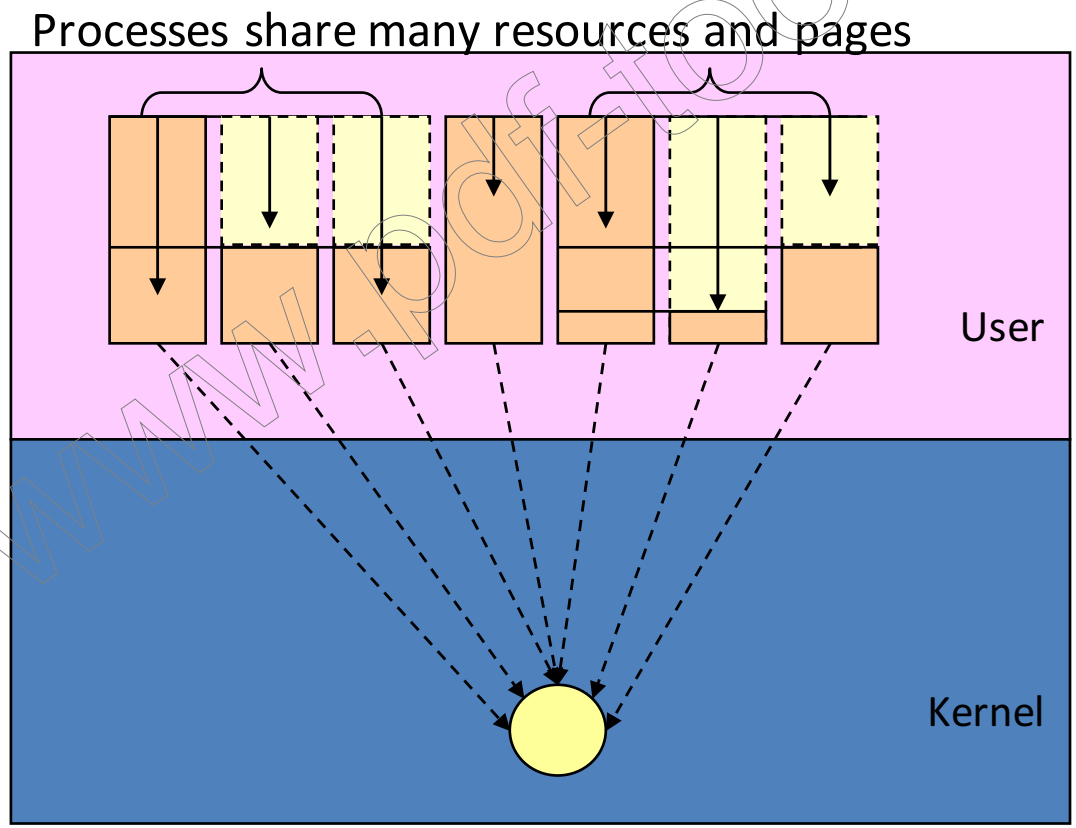
- Depending on OS designs



Processes only User thread kernel thread

Process related terms (Cont.)

- Linux lightweight process



Discussion on 3/22

```c
#include<stdio.h>
void main()
{
      int x= 38;
      printf("Address of x = %p\n", &x);
}
```

```
/a.out
Address of x = 0×00010000
```

Cache, TLB, virtual address, logical address, physical address, …