Duke Systems

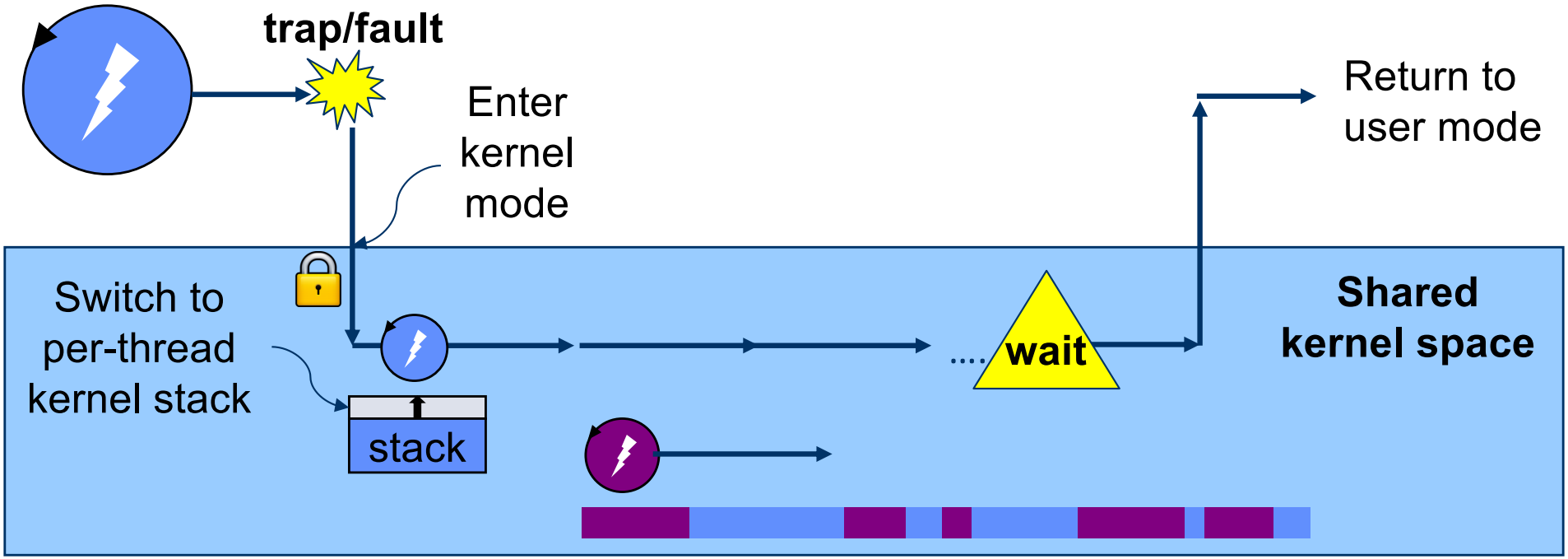
Advanced OS (CPS 510)

Kernel Synchronization

Jeff Chase

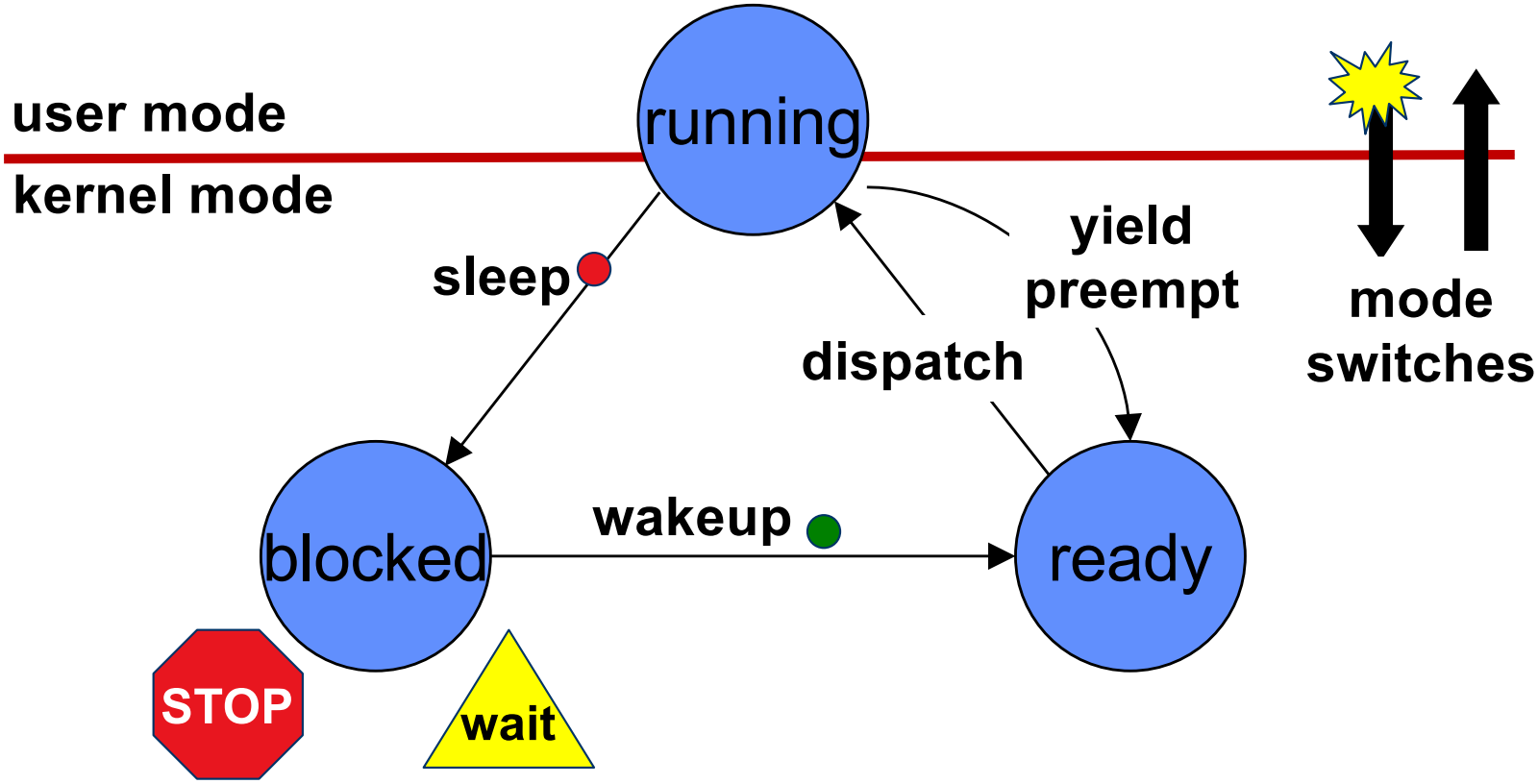
Duke University

Threads in the kernel: racy!



- Multiple threads/processes may run in the kernel at the same time.
- They access shared kernel data structures.
- They **interleave**—by yield, block, or dispatch on multiple cores.
- They can **race**. Concurrency: control it!

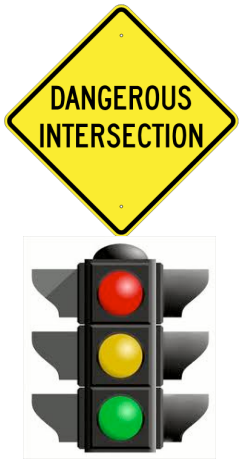
Thread states and transitions

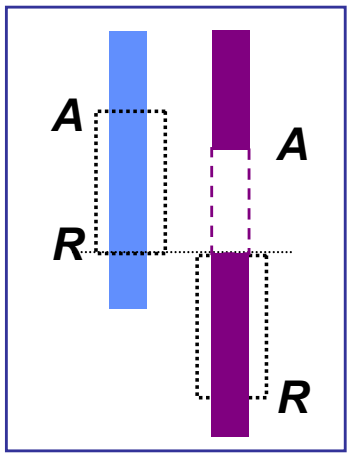


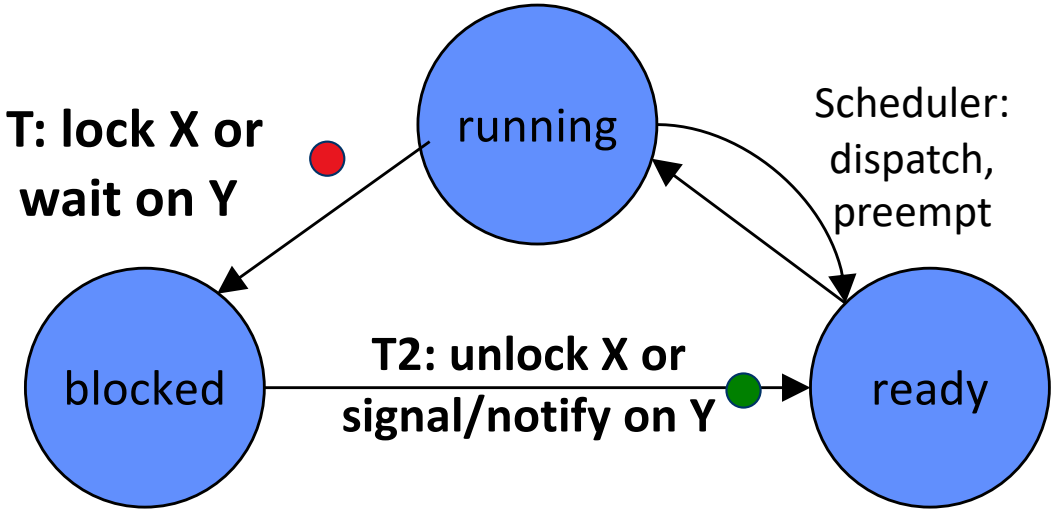
Transitions in and out of the **running** state are **context switches**.
Transitions between k/u mode in **running** state are **mode switches**.

Step 1: blocking synchronization

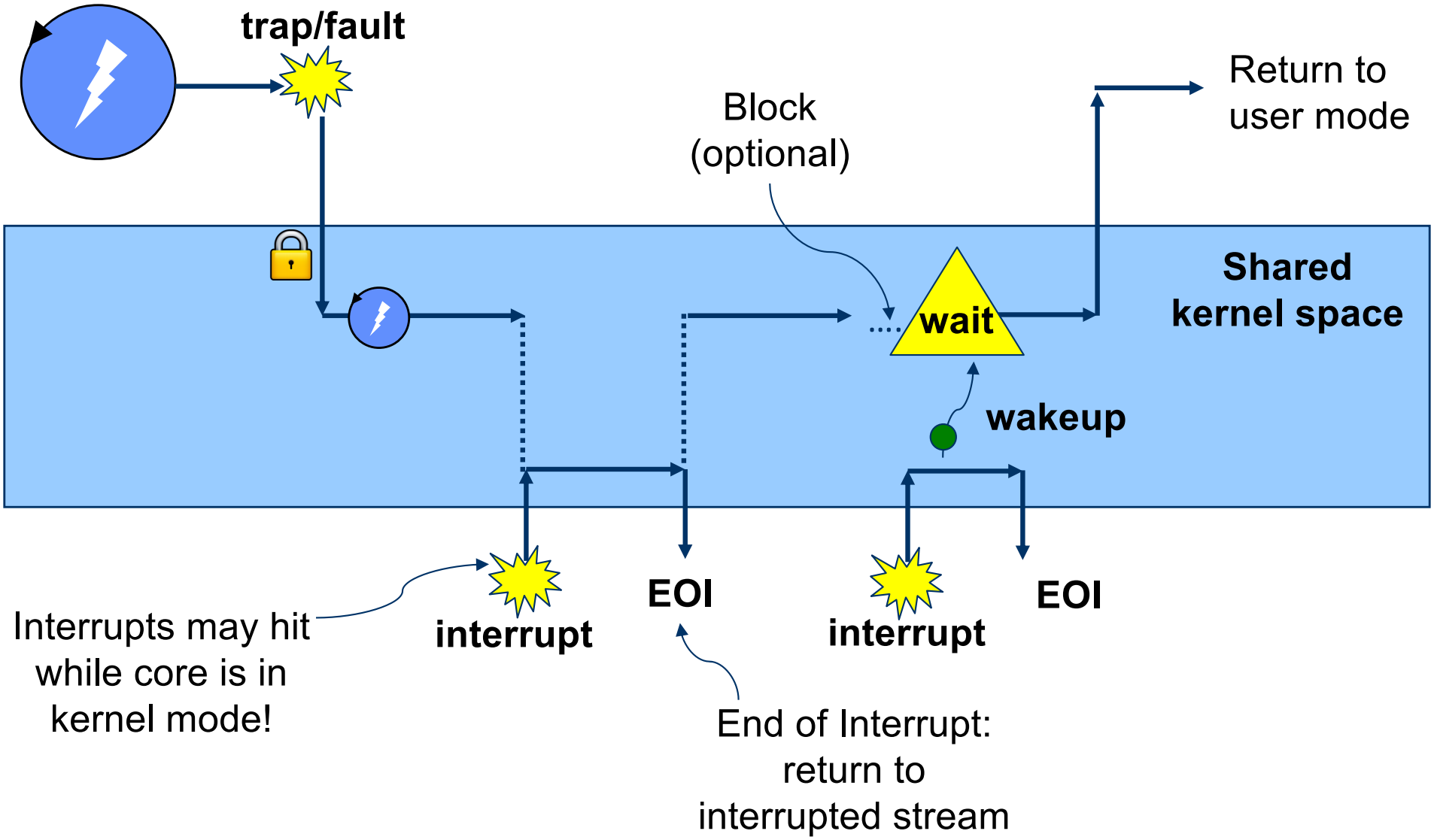
- Yes: kernel has mutex locks and wakeup conditions.
- (Linux uses semaphores—equivalent to monitors.)
- But that is not the whole story.



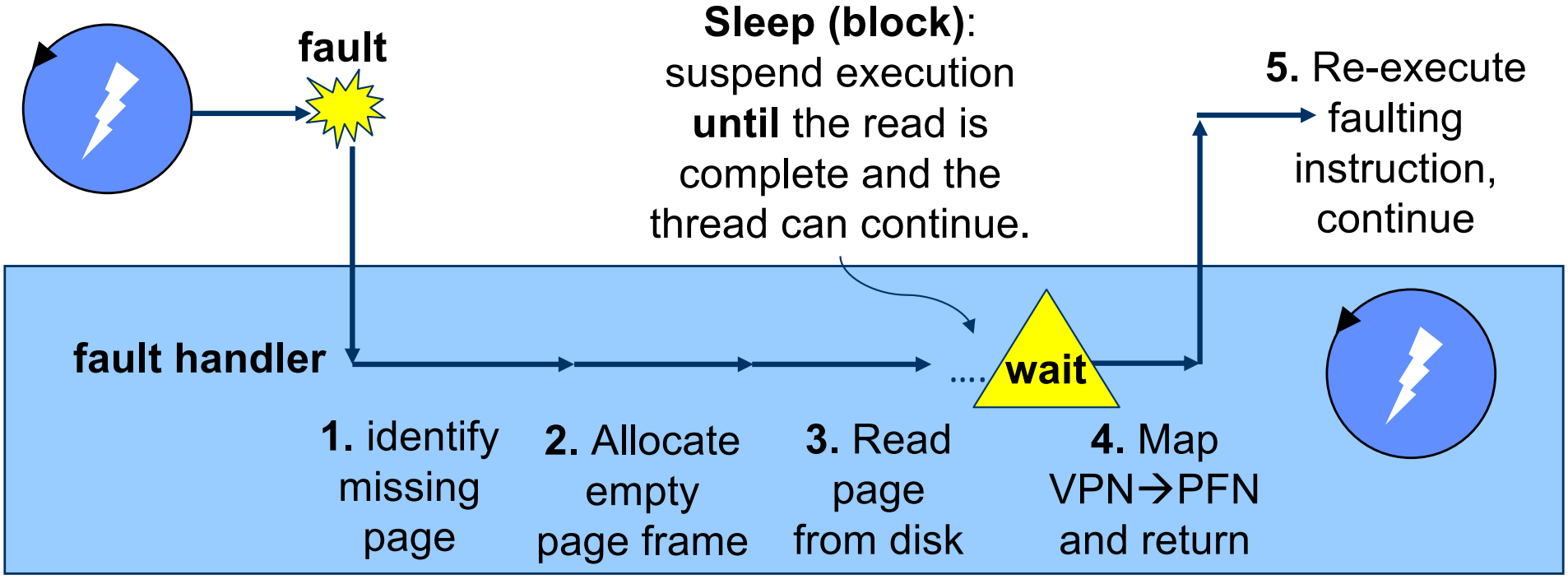




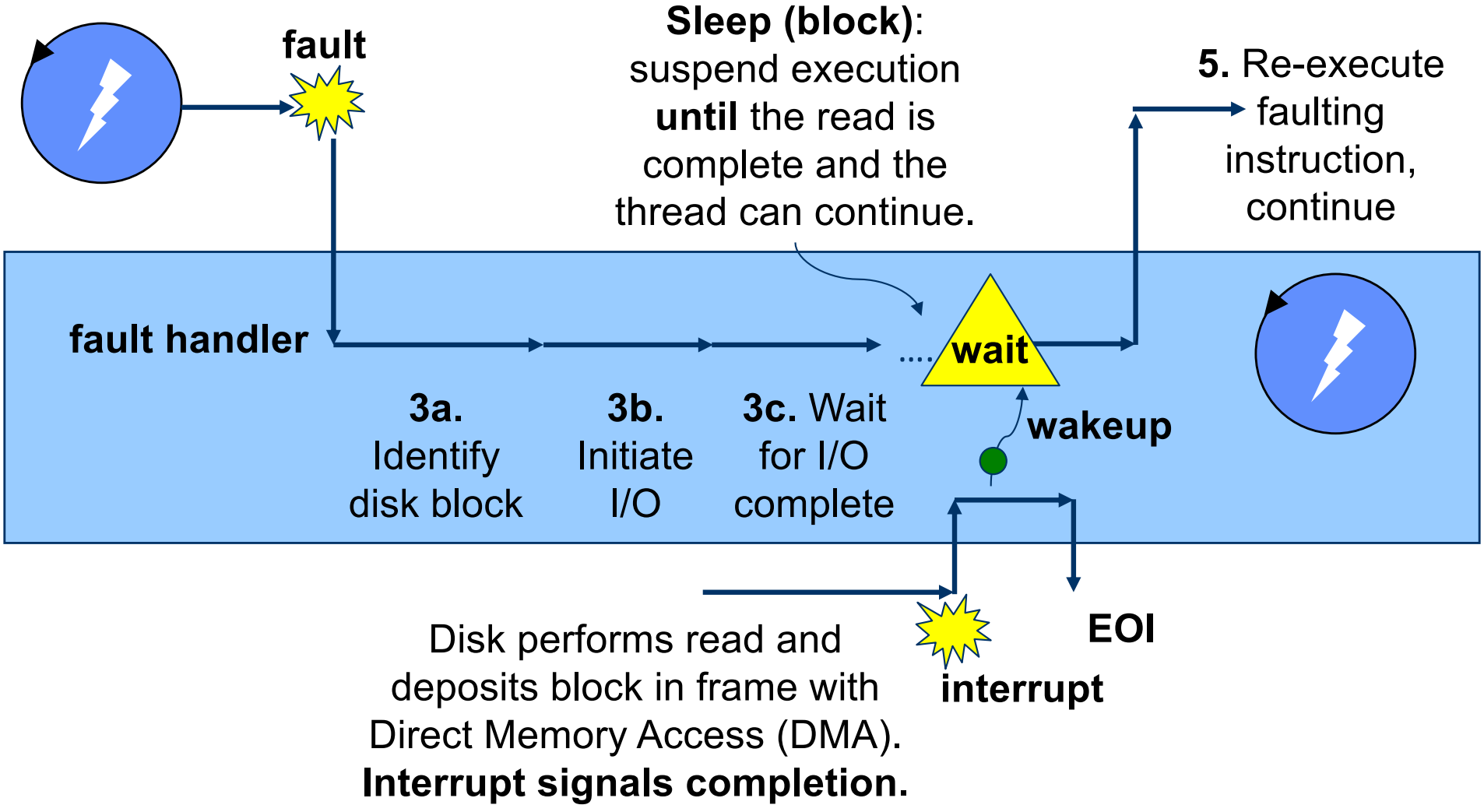
The role of interrupts



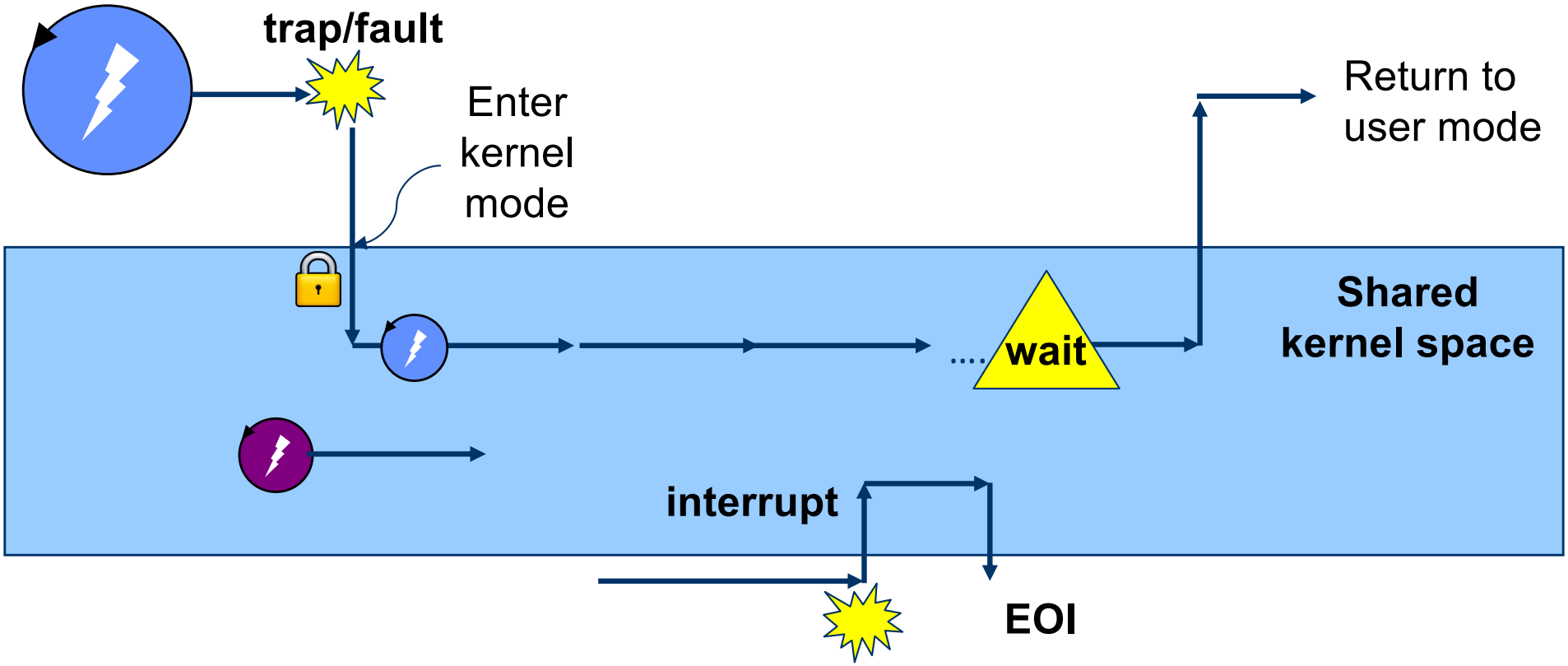
Example: handling a page fault



Example: handling a page fault



Threads + interrupts → more races



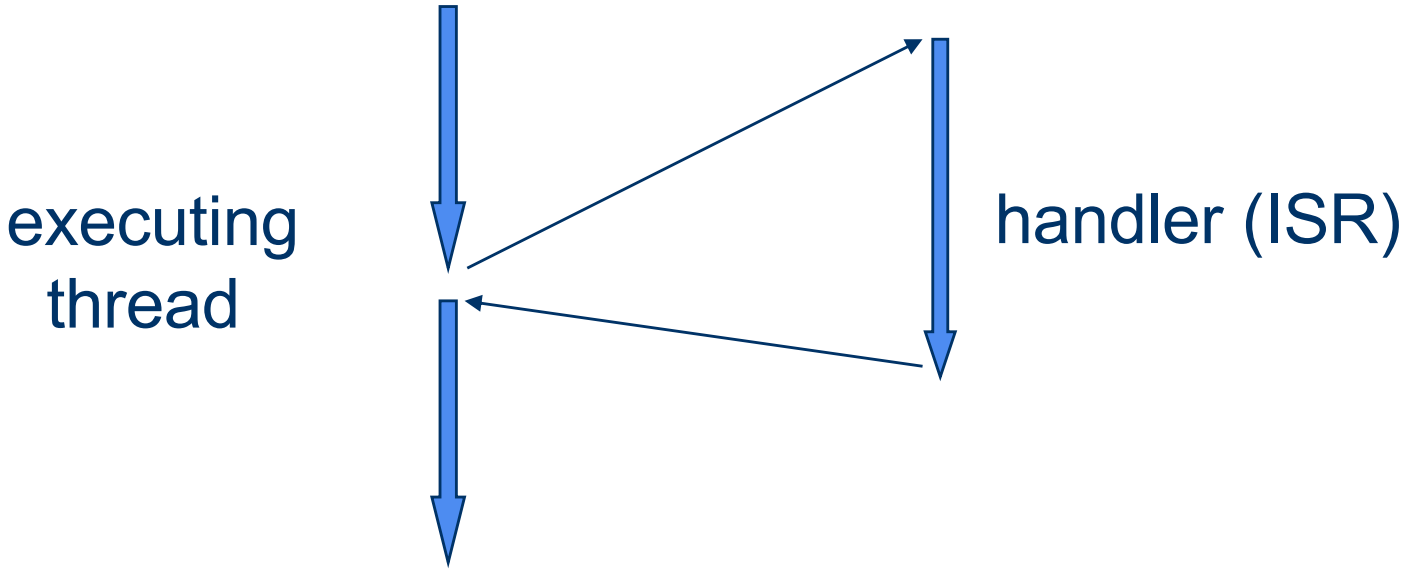
- **How to synchronize with interrupt handlers?**
- Interrupt handlers may race with the threads.
- And/or they may race with one another, possibly on other cores.

Uniprocessors: classic easy case

- So many years ago I worked on a BSD-derived kernel at the dawn of the multiprocessor (SMP) era.
- It was a classic Unix kernel for uniprocessors:
  - Locks and conditions were just flag bits in data structures: locked, waiting, etc.
  - No atomic instructions. Nobody cared, because:
  - **No preemptions in kernel mode!** (As in Linux.)
  - If you have the processor, you have it until you let it go—by sleeping or returning to user mode.
  - **Disable interrupts** when operating on data shared with interrupt handlers.

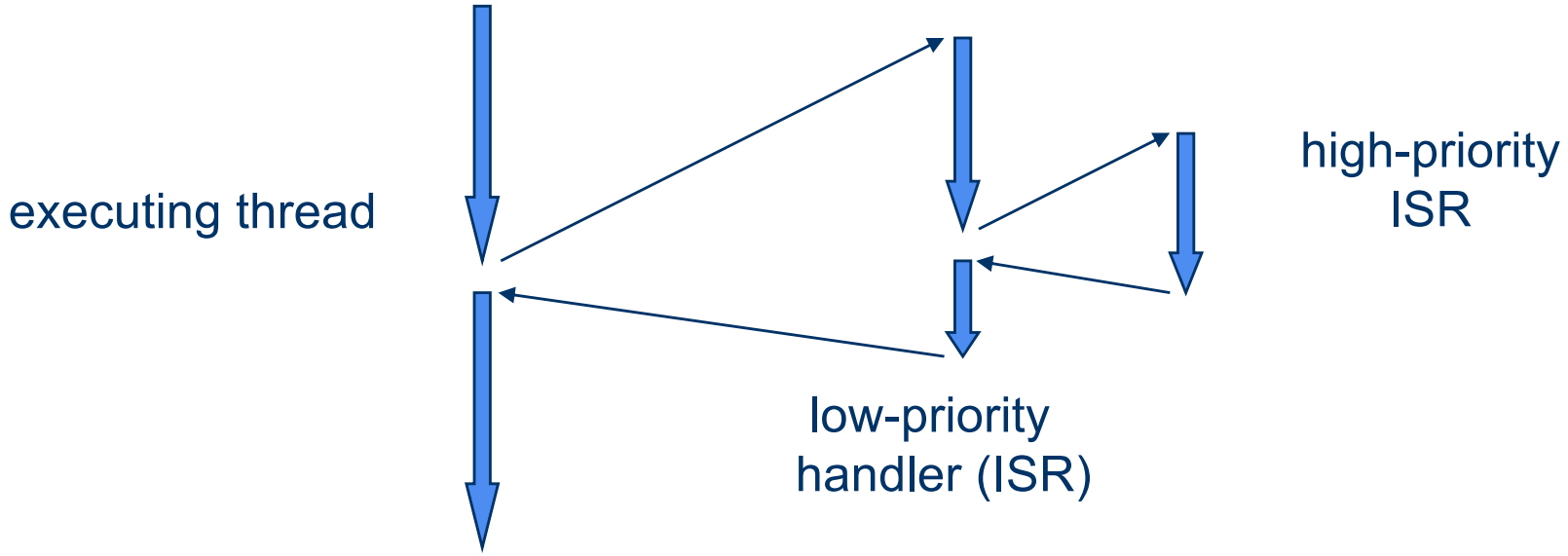
Interrupts

- Interrupt delivery transfers control to the corresponding handler (**Interrupt Service Routine** or **ISR**).
- ISR runs kernel code in kernel mode in kernel space.
- Each type of interrupt—an interrupt **vector** or **interrupt request line** (IRQ)—has a registered ISR.



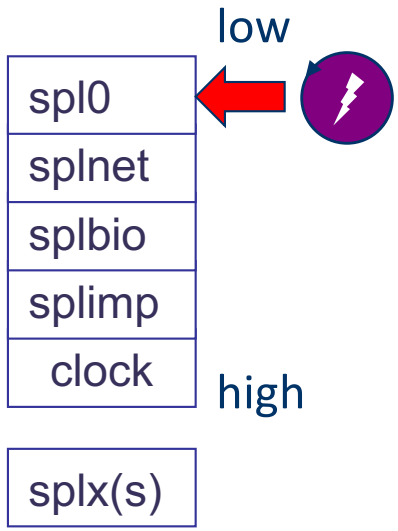
Interrupt nesting and priority

- Each registered IRQ type has an associated **priority**.
- E.g., Intel-64 has 256 IRQs and 16 (14) priority classes.
- Mapping to specific events and devices is OS-defined.
- Hardware dispatches interrupts to core in priority order.
- A high-priority IRQ/ISR may interrupt a lower one.



Interrupt priority: rough sketch

- N interrupt priority classes
- When an ISR at priority *p* runs, CPU blocks interrupts of priority *p* or lower.
- Kernel software can query/raise/lower CPU interrupt priority level (IPL/PPL).
  - Raising to IPL **p** defers or masks delivery of interrupts at IPL **p** or lower.
  - Avoid races with ISR at IPL **p** by disabling: raise CPU IPL to **p**.
  - Be sure to put it back!
  - e.g., BSD Unix *spl*/splx* primitives.
- **Summary**: Kernel code can enable/disable interrupts as needed.



BSD example

```
int s;
s = splhigh();
/* all interrupts disabled */
splx(s);
/* IPL is restored to s */
```

Details vary by machine (I-64)

The processor-priority class determines the priority threshold for interrupting the processor. The processor will deliver only those interrupts that have an interrupt-priority class higher than the processor-priority class in the PPR. If the processor-priority class is 0, the PPR does not inhibit the delivery any interrupt; if it is 15, the processor inhibits the delivery of all interrupts. (The processor-priority mechanism does not affect the delivery of interrupts with the NMI, SMI, INIT, ExtINT, INIT-deassert, and start-up delivery modes.)

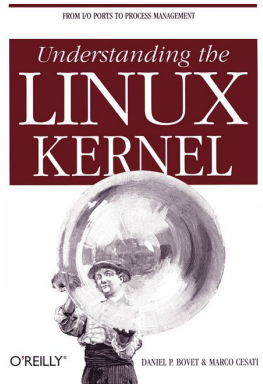
The processor does not use the processor-priority sub-class to determine which interrupts to delivery and which to inhibit. (The processor uses the processor-priority sub-class only to satisfy reads of the PPR.)

While the processor is servicing the highest priority interrupt, the local APIC can send additional fixed interrupts by setting bits in the IRR. When the interrupt service routine issues a write to the EOI register (see Section 10.8.5, "Signaling Interrupt Servicing Completion"), the local APIC responds by clearing the highest priority ISR bit that is set. It then repeats the process of clearing the highest priority bit in the IRR and setting the corresponding bit in the ISR. The processor core then begins executing the service routing for the highest priority bit set in the ISR.

If more than one interrupt is generated with the same vector number, the local APIC can set the bit for the vector both in the IRR and the ISR. This means that for the Pentium 4 and Intel Xeon processors, the IRR and ISR can queue two interrupts for each interrupt vector: one in the IRR and one in the ISR. Any additional interrupts issued for the same interrupt vector are collapsed into the single bit in the IRR.

If the local APIC receives an interrupt with an interrupt-priority class higher than that of the interrupt currently in service, and interrupts are enabled in the processor core, the local APIC dispatches the higher priority interrupt to the processor immediately (without waiting for a write to the EOI register). The currently executing interrupt handler is then interrupted so the higher-priority interrupt can be handled. When the handling of the higher-priority interrupt has been completed, the servicing of the interrupted interrupt is resumed.

Details vary by OS (Linux)



4.2.1 Interrupt and Exception Vectors

Each interrupt or exception is identified by a number ranging from to 255; for some unknown reason, Intel calls this 8-bit unsigned number a *vector*. The vectors of nonmaskable interrupts and exceptions are fixed, while those of maskable interrupts can be altered by programming the Interrupt Controller (see Section 4.2.2).

Linux uses the following vectors:

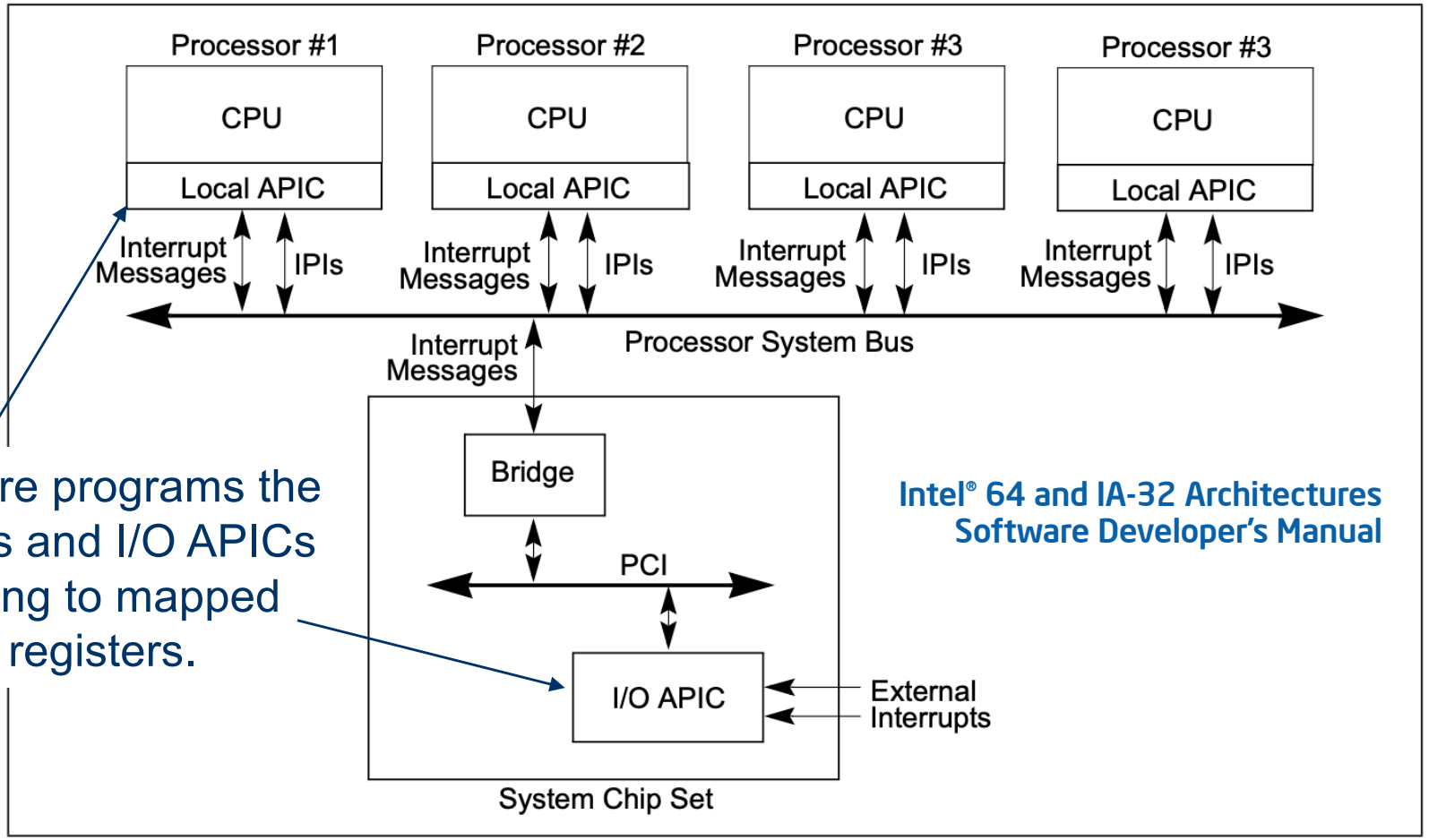
- Vectors ranging from to 31 correspond to exceptions and nonmaskable interrupts.
- Vectors ranging from 32 to 47 are assigned to maskable interrupts, that is, to interrupts caused by IRQs.
- The remaining vectors ranging from 48 to 255 may be used to identify software interrupts. Linux uses only one of them, namely the 128 or `0x80` vector, which it uses to implement system calls. When an `int 0x80` Assembly instruction is executed by a process in User Mode, the CPU switches into Kernel Mode and starts executing the `system_call( )` kernel function (see Chapter 8).

4.2.2 IRQs and Interrupts

Each hardware device controller capable of issuing interrupt requests has an output line designated as an *IRQ* (Interrupt ReQuest). All existing IRQ lines are connected to the input

The IRQ lines are sequentially numbered starting from 0; thus, the first IRQ line is usually denoted as IRQ0. Intel's default vector associated with IRQ*n* is *n*+32; as mentioned before, the mapping between IRQs and vectors can be modified by issuing suitable I/O instructions to the Interrupt Controller ports.

OS controls which cores receive which I/O interrupts (I-64)



Software programs the LAPICs and I/O APICs by writing to mapped control registers.

Intel® 64 and IA-32 Architectures Software Developer's Manual

Figure 10-2. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems

Interrupts: a few important details

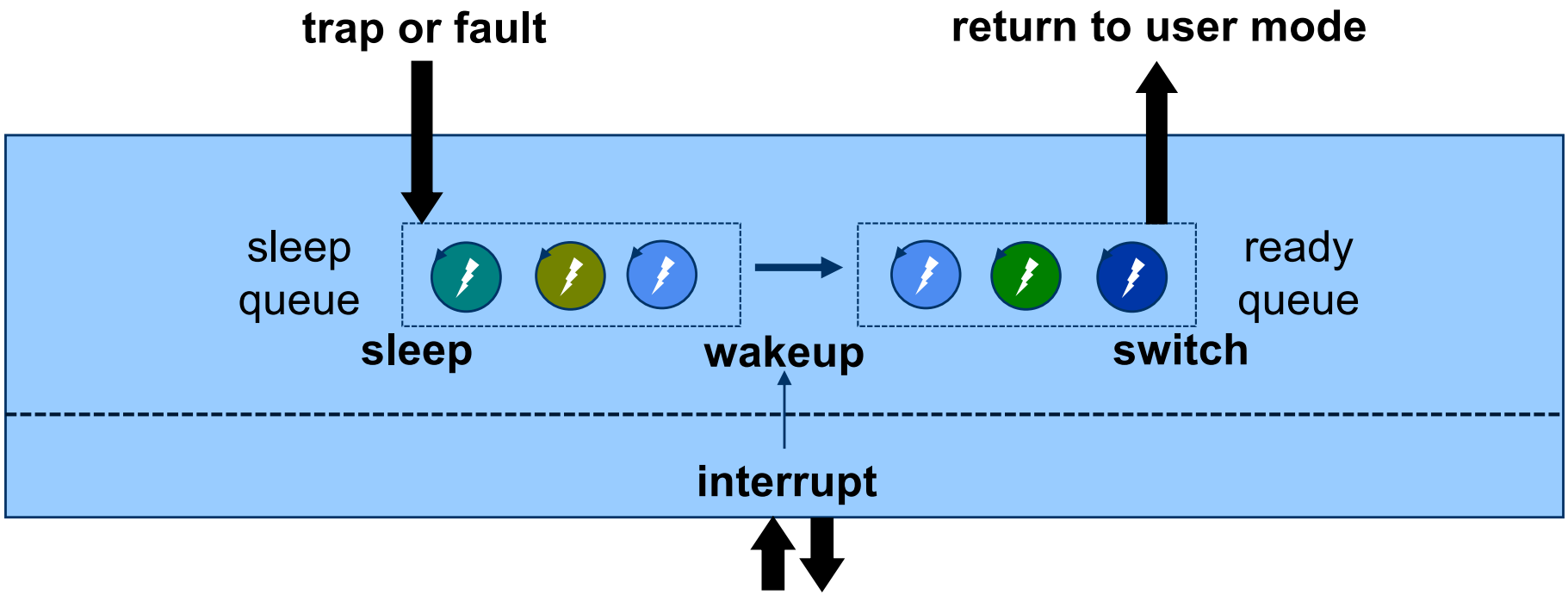
- **Privileged software controls interrupt functions.**
  - Kernel(s) as mediated by optional hypervisor
  - Invisible to user mode (except by timing)
- IPL/PPL and enable/disable are **per core**.
- Cores can send **inter-processor interrupts** (IPI).
  - E.g., PTE invalidation → **TLB shootdown** IPI.
- Software controls: handler (ISR) selection, stack mapping, priority levels, and mapping to cores.
  - Typical: one **interrupt stack** per core.
- Hypervisor may mediate interrupt injection to VMs, or configure hardware for direct (**exitless**) dispatch to VMs.

What ISRs do

- Interrupt handlers:
  - trigger involuntary thread switches (preempt—if thread is preemptable, e.g., not in kernel mode)
  - bump counters, set flags
  - throw I/O packets on/off queues
  - …
  - wakeup waiting threads
- Wakeup puts a thread on the ready queue.

But how do we synchronize with ISRs?

ISRs race and they never sleep

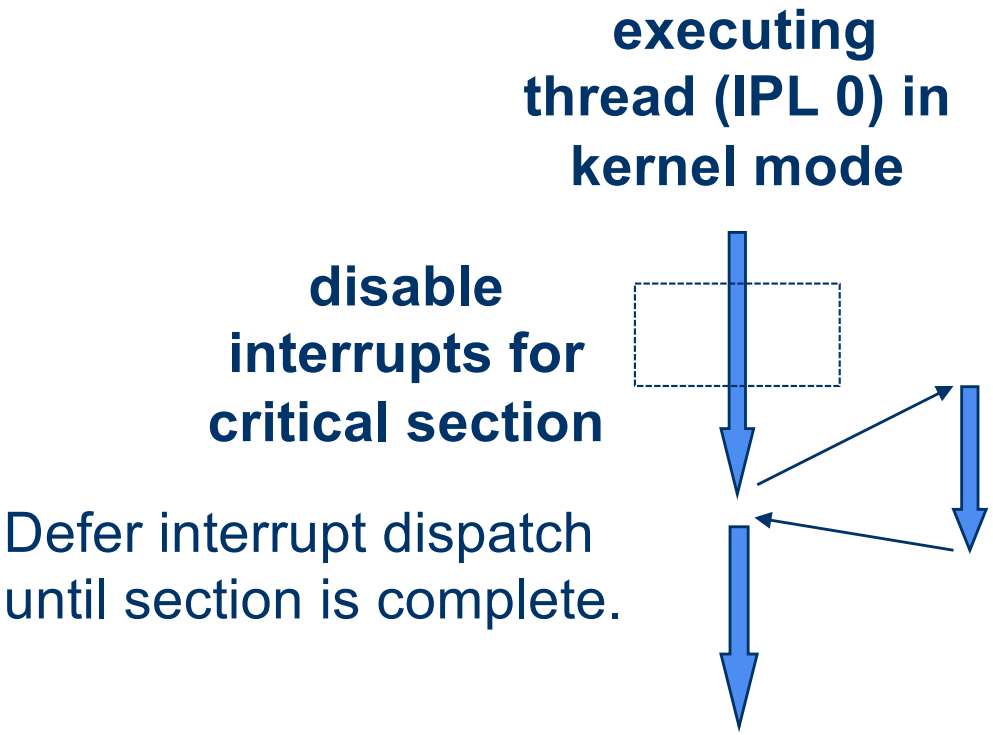


Interrupt handlers access state shared with thread-level (IPL 0) code.

Interrupt handlers do not block: If an interrupt arrived while another handler was sleeping, it corrupts the interrupt stack. **No mutexes!** Semaphores (e.g., in Linux) permit ISR to wakeup without risk of blocking. And we can **disable them**.

Synchronizing with ISRs

- Interrupt delivery can cause a race if the ISR shares data (e.g., a thread queue) with the interrupted code.
- **Solution:** disable interrupts for interrupt-critical sections.
- Interrupt dispatch at each priority *p* is serial (per-core).




```
int s;
s = splhigh();
/* critical section */
splx(s);
```

Next step: synchronizing SMPs

- Modern systems: **symmetric multiprocessor** (SMP).
  - Multiple processors and/or multiple cores per chip.
  - Kernel code can run on any core.
- **Now what?**
  - ISRs run on multiple cores, and so may race with other ISRs or threads on other cores.
  - **Disabling interrupts is no longer sufficient.**
  - (Cores do not disable one another's interrupts.)
  - And: **no blocking locks** for ISRs.
- We need fast+effective synchronization for lots of short critical sections, with no blocking: **spinlocks**.

Non-blocking synchronization

We need hardware support for atomic read-modify-write operations on data item X by a thread T.

- **Atomic** means that no other code can operate on X while T's operation is in progress.
- We can use **atomic instructions** for safe low-level data access without locks (e.g., fetch-and-add).
- We use them as a "toehold" to implement SMP-safe blocking synchronization.
- We use them to implement **spinlocks:** locks that busy-wait in a loop when not free, instead of blocking.

Spinlock: Intel

Intel® 64 and IA-32 Architectures
Software Developer's Manual

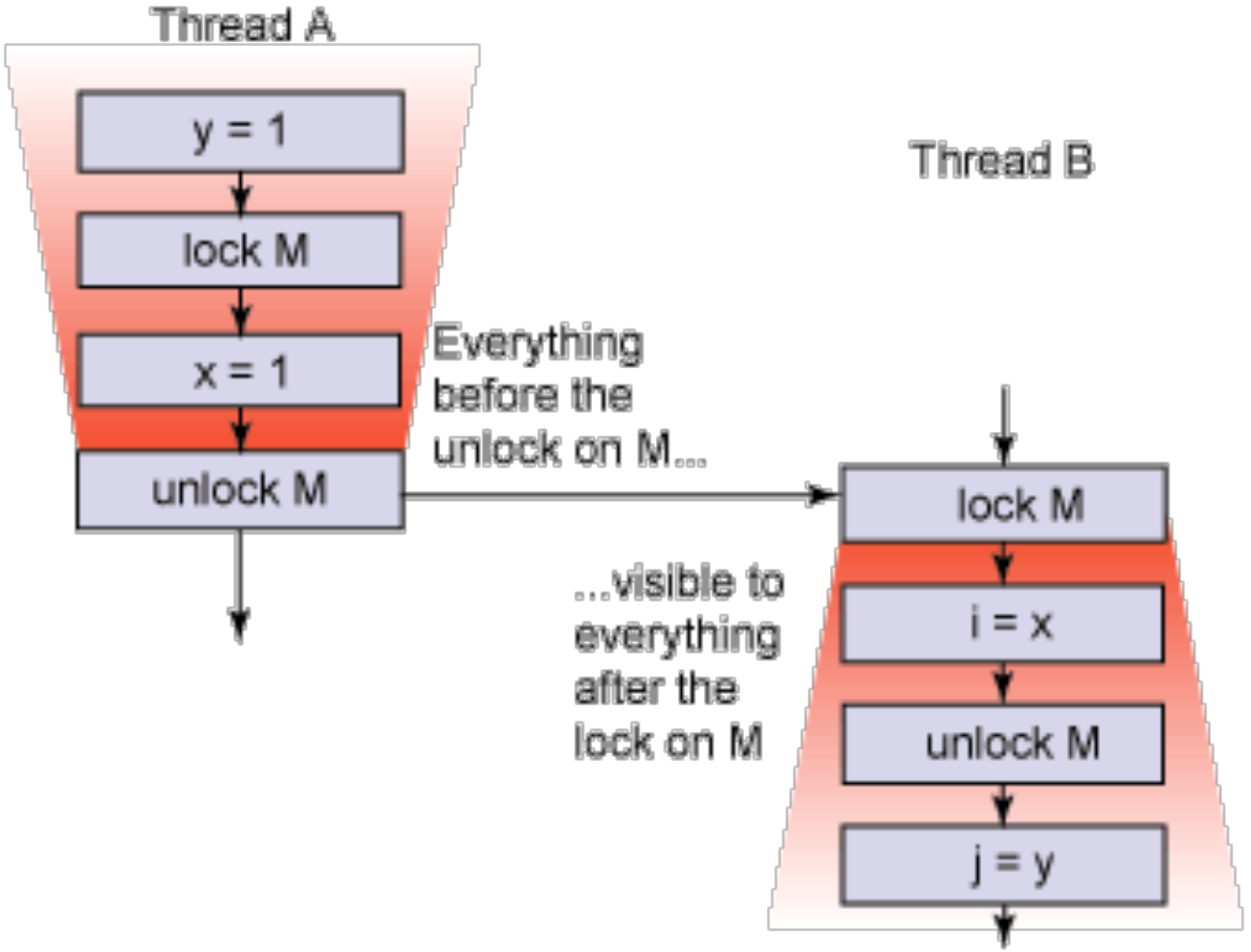
Idle the core for a contended lock.

Atomic exchange to ensure safe acquire of an uncontended lock.

```
Spin_Lock:
    CMP lockvar, 0          ;Check if lock is free
    JE Get_Lock
    PAUSE                   ; Short delay
    JMP Spin_Lock
Get_Lock:
    MOV EAX, 1
    XCHG EAX, lockvar       ; Try to get lock
    CMP EAX, 0              ; Test if successful
    JNE Spin_Lock
```

XCHG is a variant of **compare-and-swap**: compare **x** to value in memory location **y**; if **x** != ***y** then exchange **x** and ***y**. Determine success/failure from subsequent value of **x**.

Atomic instructions also drive hardware memory consistency



XCHG: some details (I-64)

Synchronization mechanisms in multiple-processor systems…can use a locking instruction such as the **XCHG** instruction or the LOCK prefix to ensure that a read-modify-write operation on memory is carried out **atomically**. Locking operations typically operate like I/O operations in that they **wait for all previous instructions to complete** and for all buffered writes to drain to memory.

Any locked instruction (either the XCHG instruction or another read-modify-write instruction with a LOCK prefix) appears to execute as an **indivisible and uninterruptible** sequence of load(s) followed by store(s).

8.2.3.8 **Locked Instructions Have a Total Order**
The memory-ordering model ensures that all processors agree on a single execution order of all locked instructions.

8.2.3.9 **Loads and Stores Are Not Reordered with Locked Instructions**
…

Intel® 64 and IA-32 Architectures
Software Developer's Manual

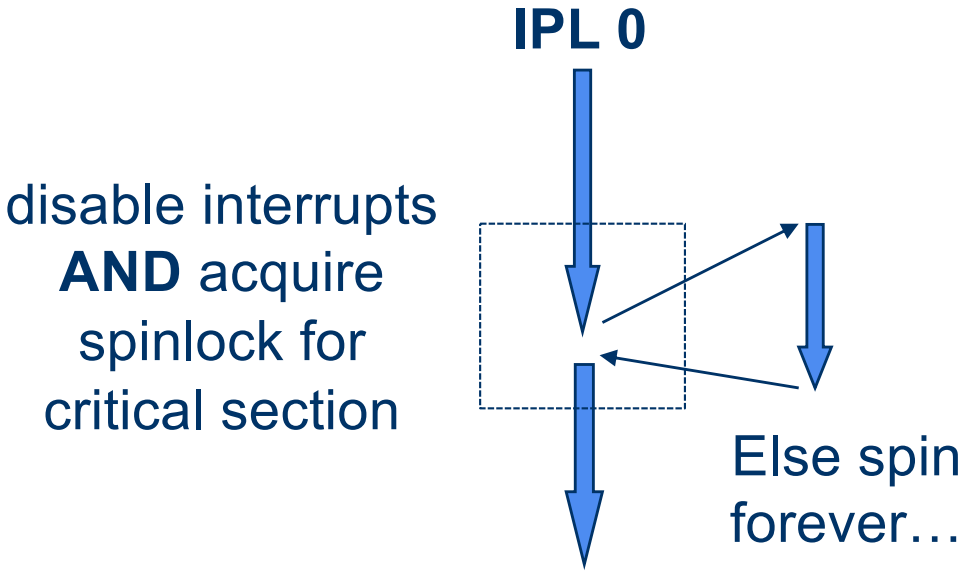
Example: Linux spinlocks

Table 11-5. Various Kernel Spin Locks	
Spin Lock	**Protected Resource**
`console_lock`	Console
`dma_spin_lock`	DMA's data structures
`inode_lock`	Inode's data structures
`io_request_lock`	Block IO subsystem
`kbd_controller_lock`	Keyboard
`page_alloc_lock`	Buddy system's data structures
`runqueue_lock`	Runqueue list
`semaphore_wake_lock`	Semaphores's waking fields
`tasklist_lock (rw)`	Process list
`taskslot_lock`	List of free entries in task
`timerlist_lock`	Dynamic timer lists
`tqueue_lock`	Task queues' lists
`uidhash_lock`	UID hash table
`waitqueue_lock (rw)`	Wait queues' lists
`xtime_lock (rw)`	`xtime` and `lost_ticks`

Understanding the Linux Kernel (Bovet and Cesati). Example only: out of date.

Synchronizing with ISRs, revisited

- **Suppose**: Core at IPL=0 (thread context) holds spinlock, interrupt occurs, ISR attempts to acquire spinlock….
- **That would be bad.** Solution: disable interrupts.
- On SMP, disabling interrupts is no longer **sufficient**, but it is still **necessary** for potential ISR races.




```
int s;
s = splhigh();
spinlock_acquire(l);
/* critical section */
spinlock_release(l);
splx(s);
```

Example: Linux spinlock macros

Table 11-4. Read/Write Spin Lock Macros on a Multiprocessor System	
Function	**Description**
`read_lock_irq(rwlp)`	`_ _cli( ); read_lock(rwlp)`
`read_unlock_irq(rwlp)`	`read_unlock(rwlp); _ _sti( )`
`write_lock_irq(rwlp)`	`_ _cli( ); write_lock(rwlp)`
`write_unlock_irq(rwlp)`	`write_lock(rwlp); _ _sti( )`
`read_lock_irqsave(rwlp,flags)`	`__save_flags(flags); __cli( ); read_lock(rwlp)`
`read_unlock_irqrestore(rwlp,flag)`	`read_unlock(rwlp); _ _restore_flags(flags)`
`write_lock_irqsave(rwlp,flags)`	`__save_flags(flags); __cli( ); write_lock(rwlp)`
`write_unlock_irqrestore(rwlp,flags)`	`write_unlock(rwlp); __restore_ flags(flags)`

Note that Linux has reader/writer (“SharedLock”) spinlocks.

Understanding the Linux Kernel (Bovet and Cesati). Example only: out of date.

Spinlocks in user mode?

- **Don’t use spinlocks in user mode.**
- They waste CPU time and they are dangerous: what if a thread is preempted while holding a spinlock?
- We typically disable preemption for threads in kernel mode, so not risky there.
- But for user mode threads, managing spinlocks and preemption safely is a research topic.
- (More on this topic later.)

Summary

- Spinlocks are fast locks for short critical sections.
- Useful, necessary, and common inside an SMP kernel.
- For synchronizing threads and ISRs across cores.
- On a uniprocessor, just enable/disable interrupts instead.
- On an SMP, interrupt disable/enable is **necessary** but not **sufficient:** we **also** need spinlocks.
- On an SMP, spinlocks etc. use atomic instructions.
- On an SMP, these also assure memory consistency for properly synchronized software.