The Kernel Latency

Advanced Operating Systems

Luca Abeni
`luca.abeni@santannapisa.it`

Latency

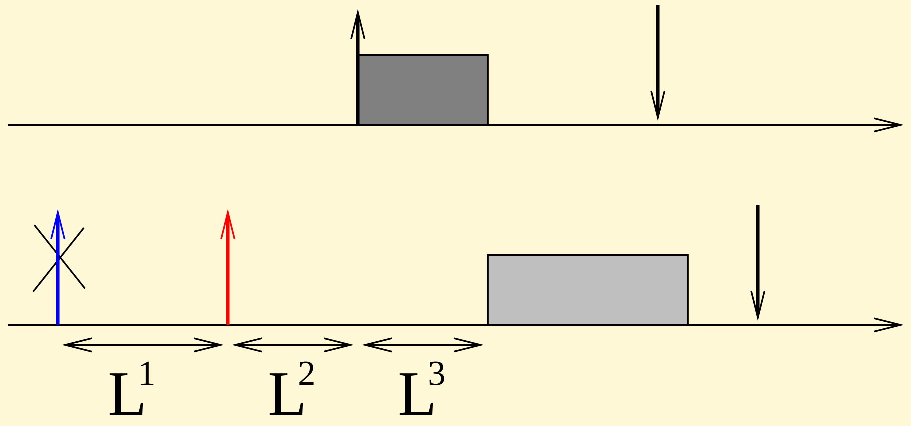
- Latency: measure of the difference between the theoretical and actual schedule
  - Task τ expects to be scheduled at time t ...
  - ... but is actually scheduled at time t'
  - $\Rightarrow$ Latency $L = t' - t$
- The latency L can be modelled as a blocking time $\Rightarrow$ affects the guarantee test
  - Similar to what done for shared resources
  - Blocking time due to latency, not to priority inversion

Effects of the Latency

- Upper bound for L? If not known, no schedulability tests!!!
  - The latency must be *bounded*: $\exists L^{max} : L < L^{max}$
- If L^{max} is too high, only few task sets result to be schedulable
  - Large blocking time experienced by *all tasks*!
  - The worst-case latency L^{max} cannot be too high

Sources of Latency

- A task τ_i is a stream of jobs $J_{i,j}$ arriving at time $r_{i,j}$
- Job $J_{i,j}$ is scheduled at time $t' > r_{i,j}$
  - $t' - r_{i,j}$ is given by:
    1. $J_{i,j}$'s arrival is signalled at time $r_{i,j} + L^1$
    2. Such event is served at time $r_{i,j} + L^1 + L^2$
    3. $J_{i,j}$ is actually scheduled at $r_{i,j} + L^1 + L^2 + L^3$

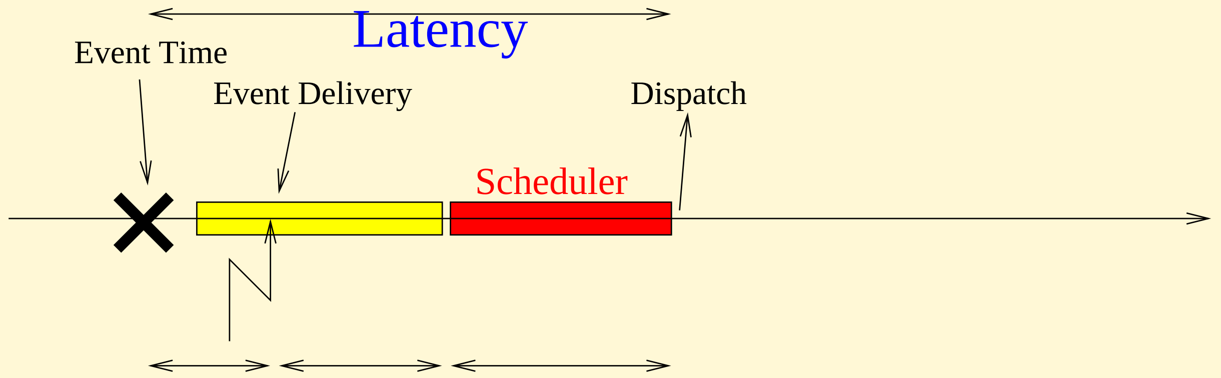


Analysis of the Various Sources

- $L = L^1 + L^2 + L^3$
- L^3 is sometimes called *scheduler latency*
  - But it is not really a latency!!!
  - Interference from higher priority tasks
  - Already accounted for by RTA / TDA or similar $\rightarrow$ let's not consider it
- L^2 is the *non-preemptable section latency* (L^{np})
- L^1 is due to the delayed interrupt generation

Non-Preemptable Section Latency

- Delay between time when an event is generated and when the kernel handles it
  - Due to non-preemptable sections in the kernel, which delay the response to hardware interrupts
  - Composed by various parts: *interrupt disabling*, *bottom halves delaying*, . . .
- Depends on how the kernel handles the various events...
- Will talk about it later!



Interrupt Generation Latency

- Hardware interrupts: generated by devices
- Sometimes, an interrupt should be generated at time $t \ldots$
- $\ldots$ but it si actually generated at time $t' = t + L^{int}$
- L^{int} is the *Interrupt Generation Latency*
  - It is due to hardware issues
  - It is *generally* small compared to L^{np}
  - Exception: if the device is a timer device, the interrupt generation latency can be quite high
    - *Timer Resolution Latency* L^{timer}

The Timer Resolution Latency

- Interrupt generation latency for a hw timer device
- L^{timer} can often be much larger than the non-preemptable section latency L^{np}
- Where does it come from?
  - Kernel timers are generally implemented by using a hardware device that produces periodic interrupts
- Can we do anything about it?

Ticks and Timers

- Periodic timer interrupt $\rightarrow$ tick
- Example: periodic task (`setitimer()`, Posix timers, `clock_nanosleep()`, ...) τ_i with period T_i
- Job end $\rightarrow$ τ_i sleeps for the next activation
- Activations are triggered by the periodic interrupt
  - Periodic tick interrupt, with period T^{tick}
  - Every T^{tick}, the kernel checks if the task must be woken up
  - If T_i is not multiple of T^{tick}, τ_i experiences a timer resolution latency

The Periodic Tick

- Traditional operating systems: timer device programmed to generate a *periodic* interrupt
  - Example: in a PC, the Programmable Interval Timer (PIT) is programmed in *periodic mode*
- At every tick the execution enter kernel space
- The kernel executes and can
  - Wake up tasks
  - Adjust tasks priorities
  - Run the scheduler, when returning to user space $\rightarrow$ possible preemption

Tick Tradeoff

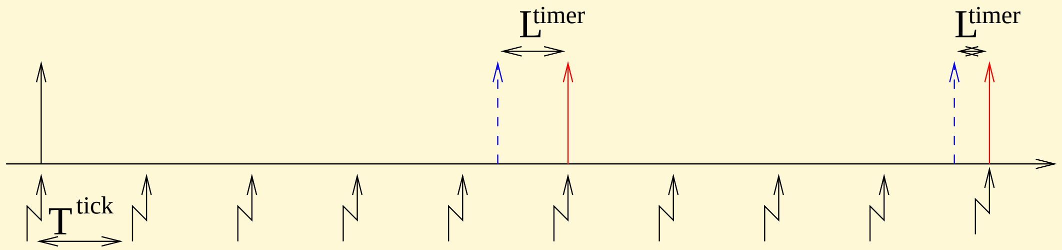
- Timer interrupt period: trade-off between responsiveness (low latency) and throughput (low overhead)
- Large $T^{tick} \rightarrow$ large timer resolution latency
- Small $T^{tick} \rightarrow$ high number of interrupts
    - More switches between US and KS
    - Tasks are interrupted more often
    - $\Rightarrow$ Larger overhead

Trade-off Examples

- For non real-time systems, it is possible to find a reasonable tradeoff...
- But it still depends on the workload!
  - Desktop or server?
- Example: the Linux kernel
  - Linux 2.4: $10ms$ (HZ = 100)
  - Linux 2.6: HZ = 100, 250, or 1000
  - Other systems: $T^{tick} = 1/1024$

Timer Resolution Latency

- Experienced by all tasks that want to sleep for a specified time T



- τ_i must wake up at time $r_{i,j} = jT_i$
- But is woken up at time $t' = \left\lceil \frac{r_{i,j}}{T^{tick}} \right\rceil T^{tick}$

Timer Resolution Latency - Upper Bound

- The timer resolution latency is bounded:
  - $t = r_{i,j}$
  - $t' = \left\lceil \frac{r_{i,j}}{T^{tick}} \right\rceil T^{tick}$

$$L^{timer} = t' - r_{i,j} = \left\lceil \frac{r_{i,j}}{T^{tick}} \right\rceil T^{tick} - r_{i,j} =$$

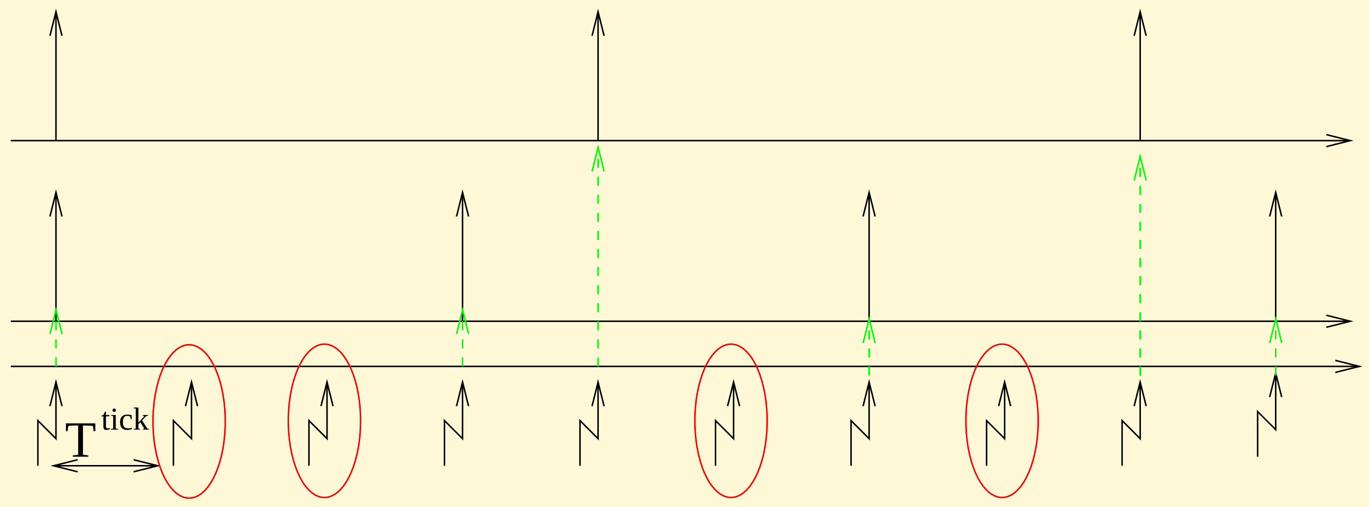
$$= \left(\left\lceil \frac{r_{i,j}}{T^{tick}} \right\rceil - \frac{r_{i,j}}{T^{tick}} \right) T^{tick} \leq T^{tick}$$

Problems with Periodic Ticks

- Reducing T^{tick} below $1ms$ is generally not acceptable...
- ...So, periodic tasks can expect a blocking time due to L^{timer} up to $1ms$
  - How large is the effect on the schedulability tests?
- Additional problems:
  - Tasks' periods are rounded to multiples of T^{tick}
  - Limit on the minimum task period: $\forall i, T_i \geq T^{tick}$
  - ...

Useless Timer Interrupts

- Additional problem: a lot of useless timer interrupts might be generated



Timers and Clocks

- Remember?
  - Timer: generate an event at a specified time t
  - Clock: keep track of the current system time
- A timer can be used to wake up a periodic task τ, a clock can be used to read the system time (`gettimeofday()`)
- **Timer Resolution**
- **Clock Resolution**

Timer and Clock Resolution

- **Timer Resolution**: minimum interval at which a periodic timer can fire
  - If periodic ticks are used, the timer resolution is T^{tick}
- **Clock Resolution**: minimum difference between two different times returned by the clock
  - What’s the expected clock resolution?

Clock Resolution

- Traditional OSs use a “tick counter”
  - Very fast clock: return the number of ticks (jiffies in Linux) from the system boot
  - Clock Resolution: T^{tick}
- Modern PCs have higher resolution time sources...
  - On x86, TSC (TimeStamp Counter)
  - High-Resolution clock: use the TSCto compute the time since the last timer tick...
- Summary: High-Resolution clocks **are easy**!
  - Every *modern* OS kernel provide them

Clock Resolution vs Timer Resolution

- Even using a “traditional” periodic timer tick, it is easy to provide high-resolution clocks
  - Time can be easily read with a high accuracy
- On the other hand, timer resolution is limited by the system tick T^{tick} (= `1 / HZ`)
  - It is impossible to generate events at arbitrary instants in time, without latencies

Timer Devices

- Timer devices (ex: PIT - i8254) generally work in 2 modes: *periodic* and *one-shot*
- Programmed writing a value C in a counter register
- The counter register is decremented at a fixed rate
- When the counter is 0, an interrupt is generated
  - If the device is programmed in periodic mode, the counter register is automatically reset to the programmed value
  - If the device is programmed in one-shot mode, the kernel has to explicitly reprogram the device (setting the counter register to a new value)

Using the One-Shot Mode

- The periodic mode is easier to use! This is why most kernels use it
- When using one-shot mode, the timer interrupt handler must:
  1. Acknowledge the interrupt handler, as usual
  2. Check if a timer expired, and do its usual stuff...
  3. Compute when the next timer must fire
  4. Reprogram the timer device to generate an interrupt at the correct time
- Steps 3 and 4 are particularly critical and difficult

Reprogramming the Timer Device - 1

- When the kernel reprograms the timer device (step 4), it must know the current time...
- ...But the last known time is the time when the interrupt fired (before step 1)...
  - A timer interrupt fires at time t_1
  - The interrupt handler starts (enter KS) at time t'_1
  - Before returning to US, the timer must be reprogrammed, at time t''_1
  - Next interrupt must fire at time t_2; the counter register is loaded with $t_2 - t_1$
  - Next interrupt will fire at $t_2 + (t''_1 - t_1)$

Reprogramming the Timer Device - 2

- The error described previously accumulates
- $\Rightarrow$ Risk: drift between real time and system time
- A *free run counter* (not stopped at t_1) is needed
- The counter is synchronised with the timer device $\Rightarrow$ the value of the counter at time t_1 is known
- This permits to know the time t_1'' $\Rightarrow$ the new counter register value can be computed correctly
- On a PC, the second PIT counter, or the TSC, or the APIC timer can be used as a free run counter

High Resolution Timers

- Serious real-time kernels $\rightarrow$ *High-Resolution Timers* (use hw timer in one-shot mode)
  - Already implemented in RT-Mach
  - Also implemented in RTLinux, RTAI and others
- General-Purpose kernels are more concerned about stability and overhead
  - Too much overhead for GP kernels?
- Fixed: hrtimers are in Linux since version 2.6.21

HRT and Timer Ticks

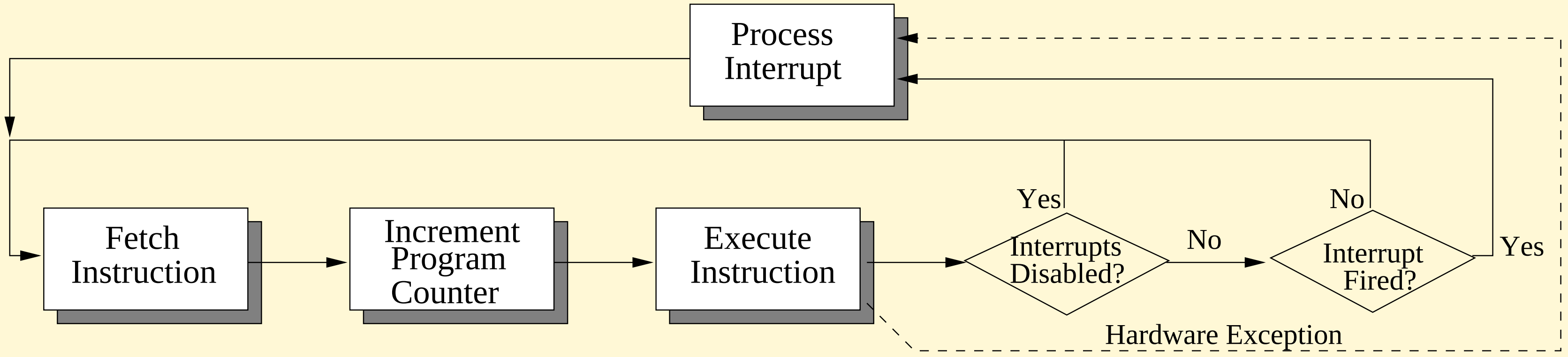
- Compatibility with “traditional” kernels:
  - The tick event can be emulated through high-resolution timers
  - $\Rightarrow$ Timer device programmed to generate interrupts both:
    - When needed to serve a timer, and
    - At tick boundaries
- ...But the “tick” concept is now useless
  - Tickless (or `NO_HZ`) system
  - Good for saving power

Non-Preemptable Section Latency

- The *non-preemptable section latency* L^{np} is given by the sum of different components
  1. Interrupt disabling
  2. Delayed interrupt service
  3. Delayed scheduler invocation
- The first two are mechanisms used by the kernel to guarantee the consistency of internal structures
- The third mechanism is sometimes used to reduce the number of preemptions and increase the system throughput

Disabling Interrupts

- Remember? Before checking if an interrupt fired, the CPU checks if interrupts are enabled...



- Every CPU has some *protected* instructions (`STI/CLI` on x86) for enabling/disabling interrupts

Interrupts and Latency

- In modern system, only the kernel (or code running in KS) can enable/disable interrupts
- Interrupts disabled for a time $T^{cli} \rightarrow L^{np} \geq T^{cli}$
- Interrupt disabling is used to enforce mutual exclusion between sections of the kernel and ISRs

Delayed Interrupt Service - 1

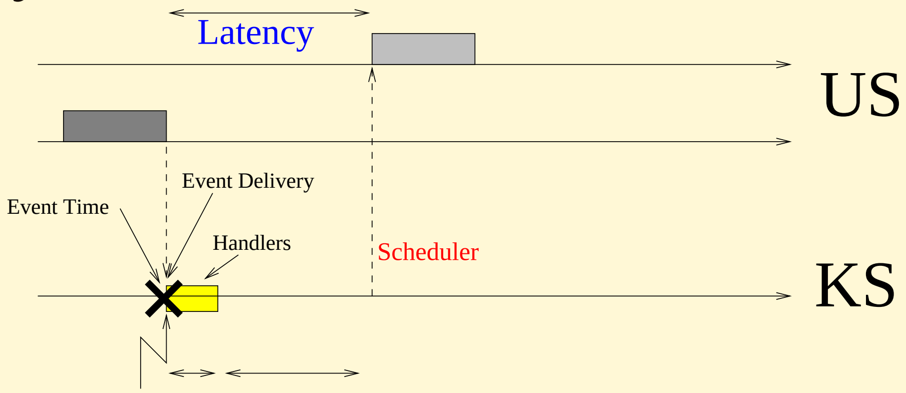
- When the interrupt fire, the ISR is ran, but the kernel can delay interrupt service some more...
    - ISRs are generally small, and do only few things
    - An ISR can set some kind of *software flag*, to notify that the interrupt fired
    - Later, the kernel can check such flag and run a larger (and more complex) interrupt handler
- Hard IRQ handlers (ISRs) va “Soft IRQ handlers”

Delayed Interrupt Service - 2

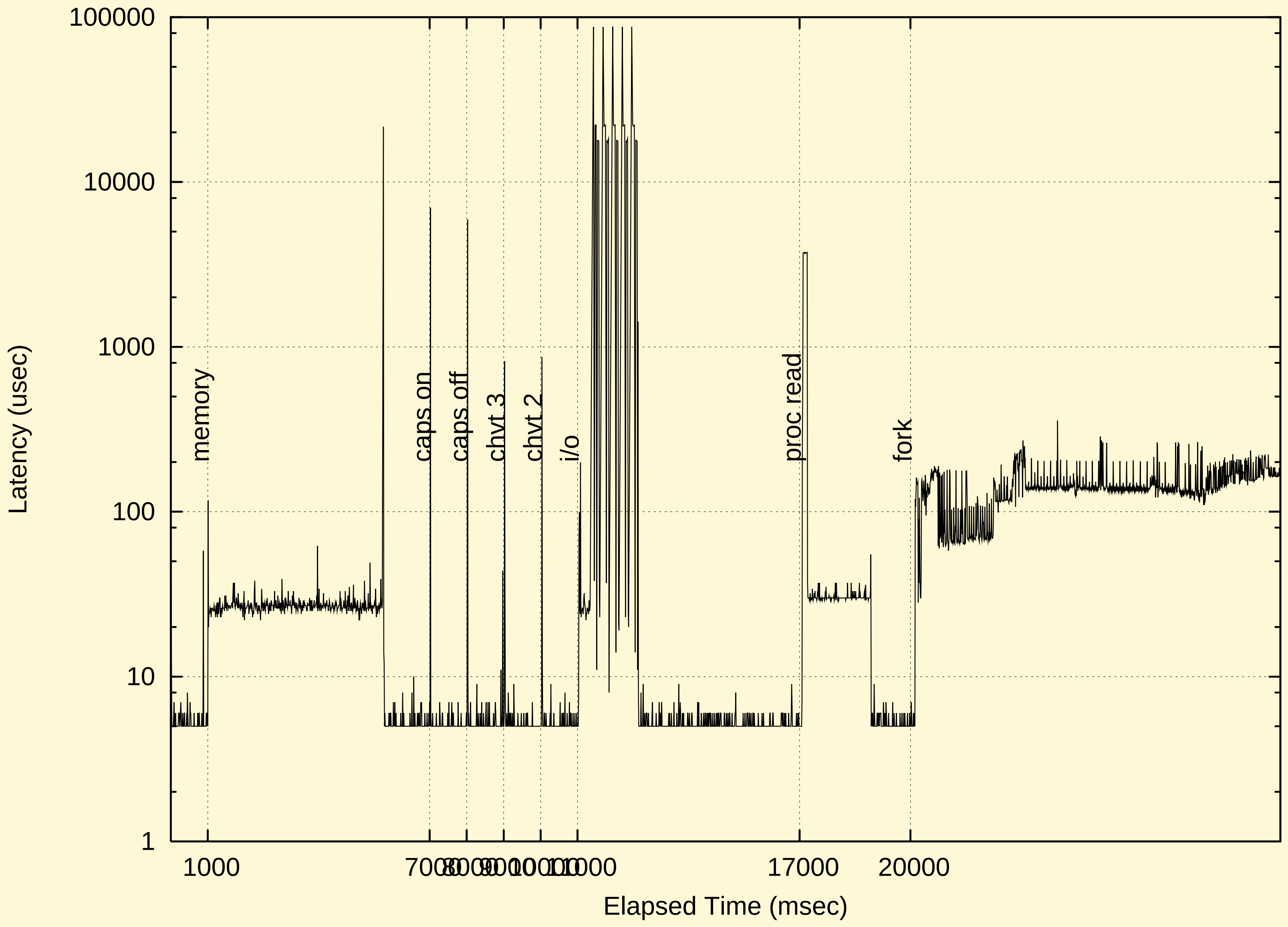
- Advantages of “soft IRQ handlers”:
  - ISRs generally run with interrupts disabled,
  - Soft IRQ handlers can re-enable hardware interrupts
  - Enabling/Disabling soft handlers is simpler/cheaper
- Disadvantages:
  - Increase NP latency: $L^{np} >> T^{cli}$
  - “Soft IRQ handlers” are often non-preemptable increasing the latency for other tasks too...

Deferred Scheduling

- Scheduler invoked when returning from KS to US
- Sometimes, return to US after a lot of activities
  - Try to reduce the number of KS $\leftrightarrow$ US switches
  - Reduce the number of context switches
  - Throughput vs low latency
- ISR executed at the correct time, soft IRQ handler ran immediately, but scheduler invoked too late



Latency in the Standard Kernel



Summing Up - 1

- L^{np} depends on some different factors
- In general, no hw reasons $\rightarrow$ it almost entirely depends on the *kernel structure*
    - Non-preemptable section latency is generally the result of the strategy used by the kernel for ensuring mutual exclusion on its internal data structures

Summing Up - 2

- To analyze / reduce L^{np}, we need to understand such strategies
- Different kernels, based on different structures, work in different ways
- Some activities causing L^{np}:
  - Interrupt Handling (Device Drivers)
  - Management of the parallelism

Example: Data Structures Consistency

- HW interrupt: *breaks* the regular execution flow
  - If the CPU is executing in US, switch to KS
- If execution is already in KS, possible problems:

1. The kernel is updating a linked list
2. IRQ While the list is in an inconsistent state
3. Jump to the ISR, that needs to access the list...

- Must *disable interrupts* while updating the list!
- Similar interrupt disabling is also used in spinlocks and mutex implementations...