ChatGPT

Latency Measurement in Operating Systems

Understanding Latency and Its Importance in OS

Latency in computing generally refers to the delay between a stimulus and response – in other words, the time it takes for a system to respond to a request or event. In the context of an operating system (OS), latency is the **delay between initiating an action and seeing its effect**. For example, it could be the time from when an application requests a resource or signals an event until the OS actually delivers the resource or schedules the task. In simple terms, *"latency in operating systems refers to the delay or time it takes for a system to respond to a given input or request"* [1] . Lower latency means the system responds faster, whereas higher latency means more delay.

Latency is a crucial metric because it directly impacts **system performance and user experience**. High latencies in an OS can lead to **sluggish responsiveness** – for instance, user interface lags, slow program launches, or missed deadlines in real-time systems. According to one analysis, excessive OS latency can cause *"decreased system responsiveness and throughput"* and lead to a *"poor user experience, particularly in interactive applications"*, ultimately resulting in user frustration and lower productivity [2] . Therefore, understanding and minimizing latency is important for both everyday computing (to keep systems feeling fast) and specialized real-time applications (to ensure timely reactions within deadline constraints).

Latency measurement is the process of quantifying these delays in an OS. Measuring latency allows developers and researchers to identify bottlenecks and verify that the system meets required performance criteria. It is *"crucial to improving system performance and responsiveness"* [3] . By carefully measuring how long certain operations take, one can gauge where the OS might be introducing delays. Common techniques for measuring latency include **benchmarking** (running standardized tests to time operations) and **profiling** (instrumenting or analyzing the system to time specific events) [4] . For example, an OS developer might benchmark how quickly the scheduler reacts to a high-priority task becoming ready, or profile the time taken for a process to send a message to another process. These measurements help in comparing different systems and in guiding optimizations.

It's important to note that latency is distinct from throughput. **Throughput** measures how much work can be done in a given time (e.g. how many operations per second), whereas **latency** measures how long a single operation takes from start to finish [5] . In many cases there is a trade-off: systems can be optimized for high throughput at the cost of individual operation latency, or vice versa, depending on requirements.

Designing latency measurements in an OS context requires careful planning to get accurate and meaningful results. Because latencies are often on the order of microseconds or nanoseconds, using a **high-resolution timer** is essential. For instance, on Linux, functions like `clock_gettime()` or processor timestamp counters (RDTSC on x86 CPUs) are used to measure short time intervals [6] . One study emphasized that using an **inaccurate timer can render the measurements invalid**, so calibration of timing methods is a necessary first step [7] . Test conditions should be controlled: the system may need to be isolated from unrelated load, CPU frequency scaling might be fixed (to avoid timing jitter), and processes might be pinned to specific CPU cores to avoid cross-core timing discrepancies [6] . Often, latency

measurements are done by repeating an operation many times and looking at averages or worst-cases. This helps account for variability and provides more statistically stable results.

In an OS, latency can manifest in many areas. This discussion will **focus on three key types of latency** and how to measure them: **(1) Task Scheduling Latency**, **(2) Inter-Partition (Inter-Process) Communication Latency**, and **(3) Memory Access Latency**. Each of these addresses a different aspect of OS performance: - *Task scheduling latency* deals with delays in the OS scheduler – how long it takes for a ready task to actually get CPU time. - *Inter-partition communication latency* (often equivalent to inter-process or inter-thread communication latency) deals with delays in data passing or messaging between protected domains (processes, threads, or partitions). - *Memory access latency* deals with delays in accessing memory, including the effect of caches and main memory speed.

Below, we delve into each of these in detail, explaining what each type of latency means, why it matters in operating systems, and how it can be measured or characterized.

Task Scheduling Latency

Task scheduling latency refers to the delay between the moment a task (process or thread) is ready to run and the moment it actually begins execution on the CPU. In a multitasking OS, when a task becomes runnable (for example, a previously sleeping task is woken by an event or a new task is created), ideally the scheduler should dispatch it onto the CPU as soon as possible, especially if it has high priority. In reality, there can be a gap – the OS may not schedule it immediately due to various reasons (ongoing operations, higher-priority work, or scheduler overhead). This gap is the scheduling latency. Formally, one can define scheduling latency as *"the time elapsed between the instant a thread becomes ready (while having the highest priority among ready threads) and the instant it is allowed to execute its code after the context switch"* [8]. In other words, if a task is ready and should be running, how long does it wait before the OS actually schedules it on the processor?

Another related term is **dispatch latency**. Dispatch latency is often used in OS literature (especially in real-time systems) to describe *"the amount of time it takes for a system to respond to a request for a process to begin operation"* [9]. It includes the time to stop one process and start the requested process. For real-time operating systems, a *bounded dispatch latency* is critical – it means there is an upper limit on how long scheduling will be delayed when a high-priority task needs to run [9]. In general-purpose operating systems (like standard Linux or Windows), scheduling latency can vary based on system load and design, whereas in a real-time OS, we strive to minimize and put an upper bound on this latency.

Why scheduling latency matters: This latency directly affects **responsiveness and real-time correctness**. For interactive applications, high scheduling latency could mean a noticeable lag after an input (e.g., you click a button and the application thread is ready, but the OS delays scheduling it, so the GUI pauses momentarily). In real-time systems, scheduling latency is even more critical: it *"affects the response times of all tasks and imposes a lower bound on the deadlines that can be supported by the system"* [10]. In other words, if the OS takes, say, 100 microseconds to schedule a task after an event, then no task can reliably have a deadline shorter than 100 microseconds because the OS itself introduces that much delay at minimum [10]. Therefore, to guarantee that tasks meet tight deadlines, the OS must have very low and predictable scheduling latency. Modern real-time variants of operating systems (for example, Linux with the PREEMPT_RT patch) put a lot of effort into reducing scheduling latency by making the kernel more preemptible and minimizing sections where interrupts or preemption are disabled [11].

Factors influencing scheduling latency: There are several internal OS behaviors that can introduce delays in scheduling. For instance, if interrupts are disabled for some time (to protect critical sections in the kernel), an incoming event that should wake a task might be delayed until interrupts are re-enabled [12] [10]. Similarly, if the kernel or a driver holds a lock or is in a non-preemptible section, it might postpone the context switch to the ready task. Other factors include the time to perform the context switch itself and hardware effects like cache or TLB flushes during a context switch [13]. On multiprocessor systems, acquiring locks in the scheduler or contention for shared resources can also add latency. Essentially, any overhead the OS has to deal with when stopping one task and starting another (be it saving/restoring CPU state, handling interrupts, updating scheduling queues) contributes to scheduling latency.

Measuring task scheduling latency: In practice, scheduling latency is measured using specialized tests or tools that timestamp the scheduling delay. A common approach is to use a **timer-based test**. For example, the Linux community often uses a tool called **cyclictest** to measure scheduling/wakeup latency [14]. The typical methodology is: a high-priority thread sets a timer to wake it at regular intervals. When the timer interrupt fires, the OS will wake the thread, and the thread then immediately checks the current time to see how late it woke up compared to the expected wake-up time. That difference is essentially the scheduling (and interrupt handling) latency for that wake-up. In a well-tuned real-time system, this difference will be very small (tens of microseconds or less), whereas in a general system under load it could be larger. The highest such latency observed (over many repetitions) is often recorded as the **worst-case scheduling latency**.

For example, in real-time Linux, *"scheduling latency is the principal metric"* used to evaluate the kernel's real-time performance and is **measured using the cyclictest tool** [14]. Cyclictest runs at high priority and continuously records latencies of timer wake-ups, giving insight into how the system behaves. Another tool in modern Linux (with PREEMPT_RT) is the `timerlat` tracer, which creates a real-time thread on each CPU that periodically wakes up and measures latency, helping to pinpoint *"sources of wake-up latencies for real-time threads"* [15] [16]. These tools can reveal, for instance, if a certain device driver or interrupt handler occasionally blocks interrupts for too long (causing an outlier in latency).

When designing a custom measurement of scheduling latency, one must ensure the measuring task has the **highest priority** (so that nothing else in user-space preempts it) and that the system is otherwise idle or under a controlled load. This way, when the measuring task is woken, any delay observed is due to the OS's internal handling, not due to some other user process running instead. We also use high-resolution timers (sub-microsecond precision) to timestamp events. If measuring on a multi-core system, it can be useful to bind the test thread and its interrupt to the same CPU core to avoid cross-core migration effects. Researchers often run such tests for long durations or under varying loads to gather a distribution of latency measurements, paying special attention to the **maximum (worst-case)** latency, since that often matters most in real-time scenarios.

In summary, task scheduling latency tells us how quickly the OS can context-switch to a ready task. It's important for ensuring responsive scheduling. By carefully measuring it (with tools like cyclictest or custom timers) and understanding the factors that cause delays, OS developers can tune the scheduler (for example, by removing long critical sections or by using better algorithms) to reduce latency and make the system more predictable [11].

Inter-Partition (Inter-Process) Communication Latency

Inter-partition communication latency refers to the time delay involved in data exchange between two separate execution contexts. Here "partition" can be understood generally as isolated contexts such as processes, threads in different address spaces, or even distinct OS partitions or virtual machines. In a general-purpose OS, this essentially means **Inter-Process Communication (IPC) latency** – how long it takes for data to go from one process to another through some communication mechanism. This is another critical performance aspect, because many applications are composed of multiple processes or services that must communicate (for example, client-server architectures, microservices, or even just GUI process sending tasks to a worker process).

A simple definition of communication latency in this context is: *"the time between initiation of transmission at the sender and completion of reception at the receiver"* [17]. In other words, if Process A wants to send a message or signal to Process B, how long does it take from the moment A sends (or writes to a communication channel) to the moment B has received and is aware of that message? This latency encompasses any OS overhead in transferring the data, scheduling delays in the receiving process, and the actual data propagation time through the chosen IPC mechanism.

Why IPC latency matters: Efficient inter-process communication is vital for performance in systems where work is split across processes. If IPC latency is high, it can become a bottleneck. For example, in a **microkernel OS**, almost every service (file system, device driver, etc.) runs in its own process, so communication between these processes must be fast or the system will spend most of its time just shuffling messages. Even in monolithic kernels, user-space programs often communicate via OS-provided IPC (sockets, pipes, message queues, shared memory, etc.). A high communication latency means longer wait times for synchronization between processes and can severely degrade throughput in pipelines or client-server interactions. Imagine a web server that spawns worker processes: if passing a request to a worker and getting a result back incurs large IPC delays, the web server's overall response time to clients will increase. In real-time distributed systems, predictable communication latency is as important as CPU scheduling latency – tasks might be on different processors or cores and still need to coordinate quickly.

Factors affecting inter-process communication latency: The latency for IPC depends on the mechanism used and the amount of data. Some common IPC mechanisms include **pipes/FIFOs**, **Unix domain sockets**, **TCP/UDP loopback sockets**, **shared memory with synchronization primitives (e.g., semaphores)**, **POSIX message queues**, and others. Each has different overhead. For example, a pipe or socket involves the kernel copying data from the sender process into a kernel buffer and then copying from the kernel buffer to the receiver process. This involves at least two context switches (sender enters kernel to write, receiver enters kernel to read) and data copying, which add latency. Shared memory can avoid data copying (both processes access the same memory region), but then synchronization (like using futexes or semaphores to notify the other process) still incurs context switches and scheduling delays. The size of the message matters too – larger messages take longer to transfer (more data to copy or move). However, the relationship is not always linear; very small messages might be dominated by fixed overhead (system call and scheduling time), whereas large messages add transfer time.

Additionally, if the communicating processes are on different CPU cores, **cache coherence effects** come into play. Data written by one core and read by another may require cache line transfers between CPUs, which introduce a hardware-level latency (on the order of tens of nanoseconds). If many processes communicate simultaneously, there could also be contention on shared resources like memory bandwidth

or the IPC buffers. A study evaluating IPC mechanisms noted that obvious variables like message size affect latency, but less obvious factors such as cache effects (e.g., cache misses, cache-coherence traffic) and even CPU scheduling decisions can impact the measured communication latency [18]. In other words, the OS might schedule the receiver a bit later, or the data might not be immediately available in cache, both of which add to the total latency beyond the raw data transfer time.

Measuring communication latency: To measure IPC latency accurately, one common approach is a **ping-pong test** (round-trip measurement). In this method, Process A sends a message to Process B; when B receives it, B immediately sends a reply back to A. A measures the total time from sending to receiving the reply. If the ping-pong is symmetric, half of that round-trip time is taken as the one-way latency [19]. This technique helps eliminate the need for perfectly synchronized clocks between processes – by using one process's measurement of a full round-trip, we assume the path there and back are similar in time. It also doubles the effect, making it easier to measure with coarse timers if needed, but modern high-resolution timers can measure even one-way microsecond latencies directly. In the round-trip method, sending many messages back-to-back and averaging (or taking maxima) gives a good sense of typical and worst-case latencies.

When implementing such a measurement, it's important to use a high-resolution timestamp (like CPU cycle counters or nanosecond clocks) and to pin processes to specific CPUs if possible (to avoid clock differences or migration issues). One study, for example, **pinned each process to a single CPU core and disabled dynamic frequency scaling** to ensure that the RDTSC timestamp counter was reliable across the measurement [6]. They also addressed multi-core timing issues by using the round-trip method as described, since reading clocks on two different cores can sometimes introduce an offset [19].

Furthermore, to get a comprehensive picture, IPC latency should be measured for different message sizes and perhaps different mechanisms. Researchers often measure a range of sizes (e.g. 4 bytes, 1 KB, 1 MB, etc.) and plot the average or median latency for each, as well as note the distribution or jitter in latency. For instance, an experiment might show that sending a 1-byte message via a Unix domain socket has, say, ~5 microseconds latency, whereas a 1 MB message might have significantly higher latency due to the time to transfer that much data. Indeed, some benchmarks have shown microsecond-range latencies for small local IPC messages, scaling up with message size (often roughly linearly after a certain point, once the fixed overhead is overcome) [20] [21]. In any case, **repeated trials** are important because OS IPC latency can fluctuate depending on system state (cache warm/cold, scheduler interruptions, etc.). By running thousands or millions of ping-pong iterations, one can compute statistics on the latency (min, mean, max, 99th percentile, etc.).

To sum up, inter-process (or inter-partition) communication latency is about measuring **how quickly data or signals can hop between processes**. It is measured by timing message exchanges, and it is influenced by the IPC mechanism overhead, context switches, data copying, and hardware effects. Reducing IPC latency might involve using more efficient mechanisms (for example, shared memory with lock-free queues can often be faster than sockets) or tweaking the OS (e.g., avoiding unnecessary context switches or buffer copies). In high-performance computing and low-latency trading systems, for instance, developers pay close attention to IPC latency, sometimes using busy-wait loops or kernel bypass techniques to cut it down to the bare minimum (nanoseconds). For a typical OS design project, one might not go to such extremes, but understanding these latency measurements is key to designing responsive inter-process interactions.

Memory Access Latency

Memory access latency refers to the delay that occurs when the CPU tries to read data from memory (or write data to memory). In modern computer systems, memory is organized in a hierarchy. At the very top are CPU registers (essentially zero latency from the CPU's perspective), then one or more levels of **cache** (small but very fast memory attached to the CPU cores), and then **main memory** (RAM), which is larger but significantly slower than caches. If data is not present in the closest cache, the CPU will have to fetch it from a slower part of the memory hierarchy, incurring a latency penalty. Formally, memory latency can be defined as *"the time between initiating a request for a byte or word in memory until it is retrieved by the processor"* [22].

For instance, if a program tries to read a piece of data at a certain address, and that data is not in the L1 cache, it might have to get it from L2 cache (a bit slower), or L3 cache, or worst-case from main memory (much slower). Each level has an access time: L1 might be only a few CPU cycles, L2 perhaps around a dozen cycles, L3 maybe dozens of cycles, and main memory on the order of hundreds of cycles. In absolute terms, a cache access might be on the order of nanoseconds, whereas a main memory access might be tens of nanoseconds. (For example, older DRAM had latencies ~70 ns, while *"modern DDR4 DIMMs have latencies under 15 ns"* in best cases [23]; however, 15 ns is still on the order of ~50 CPU cycles on a 3GHz processor.) This disparity is often referred to as the "memory wall" – CPU speeds have increased much faster than memory speeds, so memory latency is a critical performance limiter.

Why memory latency matters in OS: Whenever a running task needs data that is not already in cache, it will experience a stall while the data is fetched from memory. This slows down the task's execution. From the OS perspective, memory latency can affect scheduling decisions and overall throughput. For example, if one process is constantly causing cache misses (thus spending a lot of time stalling on memory), the OS might schedule another process in the meantime on some CPUs (this is a technique to hide latency by overlapping work, often used in throughput-oriented systems). In real-time systems, unpredictable memory latency (e.g., due to cache misses or memory contention) can cause task execution times to vary, which complicates guaranteeing deadlines.

The OS also introduces its own memory-related latencies: **page fault latency** (if a process accesses memory that is not in RAM, causing a disk access to load a page – this latency is in the millions of nanoseconds, i.e., milliseconds, much larger scale), and **context-switching costs related to caches** (when a new process is scheduled, it will cold-start with caches that contain the previous process's data, often causing more cache misses initially). The OS memory manager tries to mitigate latency with strategies like caching disk data in RAM, prefetching, and ensuring that frequently-used parts of the OS stay in memory.

Measuring memory access latency: Unlike scheduling or IPC, memory latency is typically measured not by instrumenting the OS, but by writing micro-benchmarks that access memory in controlled patterns. The goal is to measure how long a memory access takes under various conditions. A classic approach is to perform **pointer chasing or array traversal experiments** that can isolate the latency of different cache levels and main memory. For example, one can write a test that allocates a large buffer and then reads from it with a certain stride pattern to intentionally cause cache misses. By increasing the size of the buffer step by step, you can determine at what size the cache can no longer hold the working set, leading to a jump in access latency that indicates you've started hitting the next level of memory. In one simple methodology, described in an Alibaba Cloud community article, the steps to measure cache latency were: 1. Allocate a buffer the size of the cache you want to measure (say L1 cache size). 2. Initialize (preload) the buffer by

reading/writing all of it so that it is cached. 3. Time how long it takes to re-read the entire buffer sequentially (or via pointer chasing) such that all accesses hit that cache level [24].

If the data stays in L1 cache, this measures the L1 cache latency. Then you can do the same with a buffer slightly larger than L1 (up to L2 size) to see L2 latency, and so on. The use of **pointer chasing** (making a linked list in the buffer so each access jumps to a next location) is a common trick to ensure the accesses cannot be optimized out or prefetched easily – it ensures each memory load depends on the previous one, so you truly measure the sequential access latency.

Timing such memory accesses requires high precision because these operations are very fast. Typically, one uses CPU cycle counters (like RDTSC) or hardware performance counters that specifically measure memory events. Tools exist to simplify this; for example, Intel provides the *Memory Latency Checker (MLC)* tool [25], which can measure latency to memory under different access patterns and loads. Another well-known benchmark is *LMbench*, which includes `lat_mem` to measure memory read latency at various strides and working set sizes. These tools effectively automate the process of accessing memory in patterns and measuring the time per access.

An important consideration in measuring memory latency is to **distinguish pure memory latency from bandwidth effects**. Latency is about the first word access delay, while bandwidth is about sustaining many accesses per second. Many memory benchmarks will carefully access one word per cache line and then jump far enough (stride) to avoid cache prefetching bringing in the next line, so that each access incurs a full latency. For example, using a stride equal to the cache line size or larger ensures you don't get the benefit of hardware prefetchers or sequential access optimizations, thus measuring the worst-case latency per access. If instead you stream through memory continuously, you might measure a mix of latency and throughput since modern CPUs can overlap memory requests.

When done correctly, these experiments reveal the characteristic latency of each level. Hypothetically, you might see something like: L1 hits ~4 nanoseconds, L2 hits ~10 ns, L3 ~30 ns, and main memory ~100 ns (numbers for illustration). Indeed, hardware specs often say things like "L1 hit = 4 cycles, L2 = 12 cycles, memory = 50ns" etc. Real measurements help validate those and see the effect of things like TLB (translation lookaside buffer) misses – a TLB miss adds latency because the OS has to walk the page table (often hardware does it, but it's effectively an extra memory access). If measuring at a very low level, one might even consider the **OS page size and TLB coverage** as factors (large pages can reduce TLB miss frequency, thus reducing average memory access latency for large arrays).

From an OS design perspective, measuring and understanding memory access latency is important for making decisions about scheduling (e.g., whether to schedule threads on the same core to benefit from cache warmth or separate cores to avoid contention), about **NUMA (Non-Uniform Memory Access)** placement (on multi-socket systems, memory attached to a different CPU socket has higher latency; the OS tries to allocate memory "near" the CPU using it), and about caching policies (what data to keep in memory vs. swap to disk).

To measure memory latency in an OS context, one might write a kernel module or use existing profiling tools to see how long certain memory operations take. However, since memory latency is largely a hardware characteristic, OS-level measurement focuses more on observing the effects (like how many cache misses occur during a certain workload and how that correlates to performance). Modern processors have performance counters that the OS can read (via tools like `perf` on Linux) to count events such as cache

misses or memory load latency cycles. For example, one could run a program and measure “last-level cache miss latency” by recording the time between a cache miss and its resolution.

In summary, **memory access latency** is a fundamental limit on how fast programs can run. It is measured with carefully designed micro-benchmarks that isolate the time to retrieve data from various memory levels. An operating system’s job is often to help hide or reduce the impact of memory latency (through scheduling, prefetching, caching of disk data, etc.), and understanding these latencies allows OS developers to optimize memory management. As a rule of thumb, **reducing latency** (whether by using faster memory technology, better caching, or OS optimizations) will generally improve performance, since the CPU spends less time idle waiting for data. But when latency cannot be easily reduced (e.g., physical limits of memory speed), the OS and software must find ways to overlap latency with useful work (via concurrent execution) or tolerate it through buffering.

Conclusion

Latency measurement in operating systems involves quantifying delays in various subsystems to ensure the system meets performance and responsiveness goals. We discussed three critical types of latency: - **Task scheduling latency:** the delay in dispatching ready tasks, which is key for responsiveness and real-time guarantees. We saw how it can be measured with high-priority timing threads and how it’s influenced by OS behavior (interrupt masking, context switch overhead, etc.) [8] [10]. - **Inter-process (inter-partition) communication latency:** the delay in exchanging data between processes, important for any OS that runs multiple cooperating processes. We looked at how to measure it (ping-pong tests) and the factors like context switches and data copies that contribute to it [17]. - **Memory access latency:** the delay in accessing data from memory, which affects virtually every instruction a CPU executes. We described the memory hierarchy’s effect on latency and how to benchmark memory latency using targeted microbenchmarks [22].

By **measuring these latencies**, system designers and researchers can pinpoint bottlenecks – for example, discovering that scheduling latency spikes when a particular driver runs, or that IPC latency is too high using a certain mechanism, or that memory latency dominates the runtime of a workload. With that knowledge, they can take steps to optimize the OS: e.g., making the scheduler more preemptive to cut scheduling latency, choosing a faster IPC method or avoiding unnecessary data copying, and improving memory access patterns or using larger caches.

In an academic and practical sense, latency measurements serve as a reminder that “**fast**” in a computer is not just about high clock speeds or throughput, but also about minimizing wait times. Optimizing an OS for latency often leads to a more responsive and deterministic system. As systems continue to evolve (with more cores, distributed services, and larger memories), latency remains a critical metric. Therefore, understanding how to measure and improve latency in task scheduling, communication, and memory access will continue to be an important skill for operating system engineers and researchers [2] [10].

Lastly, it’s worth noting that while we examined these latency types separately, they can interplay. For example, a message send involves both communication latency and scheduling latency (the receiver must be scheduled to handle the message), and memory latency can affect both (cache misses in the scheduler or in message copying will add to those latencies). Thus, a holistic view is needed when fine-tuning an OS. Through careful measurement and analysis, one can achieve a balance that suits the target use case – whether it’s a general-purpose OS where average performance is the goal, or a real-time system where

predictably low latency is the priority. By following the measurement approaches and principles outlined above, one can obtain the detailed insights necessary to guide such optimizations.

Sources:

1. Sarah Lee, *Understanding Latency in OS* [1] [2] [4]
2. Daniel Bristot de Oliveira, *Demystifying Real-Time Linux Scheduling Latency* (Red Hat Research) [8] [14]
3. Felipe Cerqueira, Björn B. Brandenburg, *A Comparison of Scheduling Latency in Linux, PREEMPT RT, and LITMUSRT* [10]
4. Oracle Documentation, *Dispatch Latency (SunOS Real-Time Programming)* [9]
5. Aditya Vikram, *Evaluation of Inter-Process Communication Mechanisms* (study report) [17] [6] [18]
6. Wikipedia, *Memory latency* [22] [23]
7. Alibaba Cloud Blog, *Measuring Cache Access Latency* (Shuai, 2022) [24]

---

[1] [2] [3] [4] Understanding Latency in OS
https://www.numberanalytics.com/blog/ultimate-guide-latency-operating-systems

[5] [22] [23] Memory latency - Wikipedia
https://en.wikipedia.org/wiki/Memory_latency

[6] [7] [17] [18] [19] pages.cs.wisc.edu
https://pages.cs.wisc.edu/~adityav/Evaluation_of_Inter_Process_Communication_Mechanisms.pdf

[8] [14] Demystifying real-time Linux scheduling latency - Red Hat Research
https://research.redhat.com/blog/article/demystifying-real-time-linux-scheduling-latency/

[9] docs.oracle.com
https://docs.oracle.com/cd/E19620-01/805-5141/chap7rt-21297/index.html

[10] [11] [12] [13] people.mpi-sws.org
https://people.mpi-sws.org/~bbb/papers/pdf/ospert13.pdf

[15] [16] Chapter 13. Measuring scheduling latency using timerlat in RHEL for Real Time | Optimizing RHEL for Real Time for low latency operation | Red Hat Enterprise Linux for Real Time | 10 | Red Hat Documentation
https://docs.redhat.com/en/documentation/red_hat_enterprise_linux_for_real_time/10/html/optimizing_rhel_for_real_time_for_low_latency_operation/measuring-scheduling-latency-using-timerlat-in-rhel-for-real-time

[20] [21] Measuring the latency of different IPC mechanisms - iceoryx.io
https://iceoryx.io/v1.0.1/getting-started/examples/iceperf/

[24] The Mechanism behind Measuring Cache Access Latency - Alibaba Cloud Community
https://www.alibabacloud.com/blog/the-mechanism-behind-measuring-cache-access-latency_599384

[25] Intel® Memory Latency Checker v3.11
https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html