MODELING OF PROCESSES AND HARDWARE-SOFTWARE INTERACTION IN MODERN OPERATING SYSTEMS AND 3D VISUALIZATION SYSTEMS

Petro Humennyy[1]

DOI: https://doi.org/10.30525/978-9934-26-715-4-51

Abstract. Modern computing platforms have reached a level of complexity in which the interaction of processes, threads, and graphics hardware within an operating system can no longer be adequately understood through textual event logs or two-dimensional diagrams alone. This creates the need for an integrated formal and visual approach capable of representing both the temporal dynamics and the structural organization of such interactions. *The purpose of the paper* is to construct an integrated concept for modeling operating system process and thread interaction together with hardware-software interaction between the central and graphics processors, and to demonstrate how three-dimensional spatial visualization can serve as an independent research instrument for analyzing this interaction rather than a merely illustrative supplement. *Methodology* of the study is based on the formal apparatus of extended temporal Petri nets for describing process and thread synchronization, a structural model of key interaction points between the operating system kernel and the graphics hardware stack (unifying the Windows Display Driver Model and Linux DRM/KMS architectures), and empirical methods of kernel event tracing, system-call tracing, and computational experimentation, which together allow the theoretical model to be verified against measured system behavior. *Results* of the study showed that the interaction of operating system processes and the interaction between the graphics processor and the kernel are governed by structurally similar regularities of parallel execution, resource competition, and synchronization, allowing both to be described within a single integrated model. It was empirically established that CPU–GPU synchronization through the fence mechanism constitutes

[1] Candidate of Technical Sciences, Associate Professor,
Senior Lecturer at the Department of Special Computer System,
West Ukrainian National University, Ukraine

© Petro Humennyy

the dominant share of total frame latency, and that hardware-accelerated rendering provides approximately an order of magnitude higher performance than software rasterization, which determines the practical limits of applicability of spatial visualization for large-scale systems. *Practical implications.* The construction of an integrated formal-spatial model is not only a means of representing operating system behavior more clearly, but also a practical tool for detecting deadlocks, performance bottlenecks, and behavioral anomalies during the development and debugging of real-time multithreaded applications and human-machine interfaces of automated process control systems. *Value/originality.* The effectiveness of the proposed integrated approach, which unites process-level modeling with graphics-hardware interaction modeling and three-dimensional visualization, creates opportunities for further research into heterogeneous computing systems, hierarchical data-aggregation methods for perceptual scalability, and the analysis of GPU-virtualization and real-time environments.

INTRODUCTION

A modern computing platform – ranging from an embedded controller of a cyber-physical system to a multicore workstation equipped with a discrete graphics accelerator – constitutes a multilevel environment in which dozens to hundreds of independent control flows are simultaneously deployed, competing for processor time, main and video memory, data transfer buses, and interrupts. The operating system acts as an intermediary that orders this competition: it schedules the execution of processes and threads, allocates resources, ensures inter-process communication, and manages hardware subsystems, including the graphics accelerator. The complexity of this mediation grows nonlinearly with the number of simultaneously active entities, since every pair of interacting threads potentially gives rise to its own synchronization scenario, and each new level of hardware abstraction – from the processor core to the *GPU* rasterization pipeline – adds its own set of states, delays, and potential points of failure.

Historically, analysis of operating system behavior relied on textual event logs, statistical performance counters, and theoretical models describing individual mechanisms – scheduling, synchronization, memory management – in isolation from one another. This approach was justified

as long as the number of concurrently executing threads could be counted in single digits and *CPU–GPU* interaction was limited to simple output of a raster image. However, modern systems are characterized by a fundamentally different scale of complexity: multicore processors execute hundreds of threads, graphics accelerators simultaneously service graphics, compute, and video tasks, and low-level graphics *APIs* (*Vulkan, DirectX 12*) have shifted a substantial share of responsibility for command-buffer management and synchronization directly to the application level. Under these conditions, textual representation of event logs ceases to be a sufficient means of analysis: a human researcher is physically unable to hold in mind the simultaneous dynamics of hundreds of states, and a linear text log does not reflect the spatial and causal structure of interactions.

This problem has two interrelated components. The first is the absence of a unified formalized model that would describe the interaction of processes and threads within the operating system in a manner consistent with the model of interaction between the OS kernel and the graphics processor – at present these two directions develop essentially independently: research into thread synchronization draws on the classical apparatus of concurrent-computing theory, whereas research into the graphics stack focuses on driver architecture and the rendering pipeline, with no systematic correlation established between them. The second component is the lack of adequate tools for presenting the results of such modeling, capable of simultaneously conveying temporal dynamics (the sequence of events, waiting duration, alternation of states) and structural organization (the topology of dependencies among processes, threads, resources, and hardware components). Two-dimensional diagrams and tables, traditionally used for this purpose, lose in formativeness as the number of simultaneously active entities grows, since they cannot naturally represent the multidimensional nature of these interrelations.

In this context, spatial modeling and three-dimensional visualization emerge not merely as a means of aesthetically appealing data presentation, but as an independent research instrument. Placing the states of processes, threads, resources, and graphics-pipeline stages in three-dimensional space, using a temporal axis, topological grouping, and color coding, makes it possible to simultaneously observe execution dynamics and dependency structure, and to detect waiting cycles, performance bottlenecks, and

behavioral anomalies that remain hidden under purely textual or numerical analysis. At the same time, it is precisely the hardware-software interaction between the central processing unit and the graphics processing unit that determines how efficiently such visualization can be implemented in real time, thereby closing the research loop: modeling of OS processes requires 3D visualization for its analysis, while 3D visualization itself is a process executed within the same operating system and subject to the same regularities of scheduling, synchronization, and hardware-software interaction.

The relevance of this chapter of the monograph thus stems from the need to construct an integrated concept that treats the operating system, its internal mechanisms of process and thread interaction, the graphics-hardware management subsystem, and the three-dimensional visualization pipeline as a single object of study rather than as a set of independent technical issues. The purpose of the discussion that follows is to formulate such a concept on the basis of generalizing and systematizing the results of the author's research in the field of modeling process and thread interaction, spatial modeling of control systems, and structural analysis of the interaction between the graphics processor and the operating system kernel.

1.Theoretical Foundations for Modeling Operating System Processes and Threads

The functioning of a multitasking operating system is determined by the dynamics of process and thread states and by the mechanisms governing transitions between these states. A process in an operating system is a unit of resource allocation – adderss space, file descriptors, access rights – whereas a thread is a unit of scheduling that executes within the address space of a process and competes for processor time on an equal footing with other threads of the same or other processes [4]. The distinction between these concepts is fundamental for subsequent modeling, since the interaction of processes (data exchange through pipes, shared memory, signals) and the interaction of threads (synchronization of access to shared data structures within a single address space) are governed by different mechanisms and require different means of formal description.

The life cycle of a thread in a general-purpose operating system is described by a finite set of states, among which it is customary to

distinguish readiness for execution (the thread waits in the scheduler queue), active execution (the thread occupies a processor core), blocking (the thread awaits release of a resource held by another thread), and waiting for a synchronization event (the thread is suspended until a certain condition occurs – a semaphore signal, satisfaction of a condition variable, completion of an input/output operation) [4]. Transitions between these states are initiated by the operating system scheduler, by the thread itself (through calls to blocking primitives), or by external events of hardware or software origin. The correctness and predictability of these transitions directly determine the stability and performance of the entire computing system: errors in the organization of transitions manifest as deadlock, resource starvation, or race conditions – phenomena that are classical objects of study in operating system theory.

For the formal description of process and thread dynamics, this chapter employs the apparatus of Petri nets extended with temporal characteristics [5], which naturally allows the representation of parallel execution, resource competition, and conditional event triggering. A Petri net is defined as the tuple [6; 7]:

$$PN = (P, T, F, W, M_0, \tau) \quad (1)$$

where the set of places P corresponds to the possible states of processes and system resources, the set of transitions T corresponds to events that change these states (resource acquisition, resource release, a change in thread state), the incidence relation F determines which places are connected to which transitions, the arc weight function W specifies the multiplicity of this connection, the initial marking M_0 fixes the initial distribution of resources and states, and the function τ assigns to each transition a temporal delay, which converts a purely logical model into a model suitable for the analysis of temporal execution characteristics. The advantage of this formalism over traditional state diagrams lies in the fact that a Petri net naturally and simultaneously represents both parallelism (several transitions may be active simultaneously) and competition (several transitions may contend for the same resource, represented by a shared place), without requiring an explicit enumeration of all possible combinations of system states – an inevitable drawback of classical finite automata as the number of concurrent threads grows [6; 7; 8].

Within this formalism, synchronization primitives acquire a transparent structural representation. A mutex, as a means of mutual exclusion of access to a critical section, is modeled by a place with a single token: the presence of a token means the resource is available, its absence means the resource is already held by another thread, and an attempt to execute an acquisition transition in the absence of a token in the place is automatically interpreted by the model as a transition of the thread into the blocked state. A counting semaphore generalizes this idea to a place with an arbitrary finite number of tokens, corresponding to the number of simultaneously available resource units; wait and signal operations on the semaphore are represented by decreasing and increasing the number of tokens, respectively, with the activity condition of the wait operation formally written as [9]:

$$M(sem) \geq 1 \Rightarrow wait() \text{ is active; } M'(sem) = M(sem) - 1 \quad (2)$$

If, at the time the wait operation is executed, the number of tokens equals zero, the thread transitions to the waiting state and remains there until another thread executes a signal operation, returning a token to the place. This description allows a single apparatus to model the classical synchronization problems – producer – consumer, readers – writers, dining philosophers – and also serves as a basis for the formal detection of deadlock [9].

Deadlock as a phenomenon arises only when four conditions hold simultaneously: mutual exclusion of resource access, retention of a resource by a process that is simultaneously waiting for another, absence of a mechanism for forced release of a held resource, and the existence of a circular chain of waiting among processes [4]. In terms of a Petri net, deadlock corresponds to a so-called dead marking – a system state from which no transition is active:

$$\forall t \in T: \neg enabled(t, M^d) \quad (3)$$

The practical detection of such markings is carried out by constructing a reachability graph of states, which systematically enumerates all possible sequences of transition firings from the initial marking, or, for greater computational efficiency, by constructing a resource allocation graph, in which the vertices are processes and resources and the edges represent holding and requesting relations; the presence of a cycle in such a graph is a necessary and sufficient condition for deadlock in the presence of single-instance resources, and the cycle-detection algorithm based on depth-first

graph traversal has time complexity linear in the total number of vertices and edges of the graph.

It is important to emphasize that a purely logical Petri net, devoid of temporal characteristics, allows one to answer only the question of the principal possibility of deadlock, but not the actual dynamics of thread competition for resources over time. This is precisely why the extension of transitions with temporal delays τ is essential for practical analysis: it makes it possible to estimate not only the fact that a hazardous state exists, but also the probability and frequency of its occurrence under a given load, which directly links the theoretical model to empirical measurements of the performance of a real system.

2.Modeling Interaction Among Operating System Components

Individually considered thread states and formal synchronization mechanisms acquire research value only in the context of a holistic model of interaction among operating system components, which, besides the processes and threads themselves, include the scheduler, the memory management subsystem, the mechanism for handling system calls and interrupts, and the kernel as the coordinating center. The scheduler determines the order in which processor time is granted to threads ready for execution, based on the chosen discipline (round-robin, priority preemption, fair share of processor time), and directly affects which transitions of the Petri net modeling a specific application will be activated at each moment in time. The scheduler thus acts as a factor external to the thread interaction model, determining the pace and sequence of transition activation, and its consideration is necessary for an adequate reproduction of real execution dynamics.

Inter-process communication mechanisms – pipes, message queues, shared memory, signals – extend the basic thread interaction model to the level of interaction among processes possessing separate address spaces [4; 11]. In this case, the places of the Petri net that previously corresponded to shared data structures within a single process are replaced by places modeling exchange buffers between address spaces, and the transitions additionally account for the cost of data copying or of context switching when control passes from one process to another. Empirical studies summarized within this line of research indicate that the choice of a specific synchronization

mechanism substantially affects performance characteristics: a comparative analysis of mutex-based locking strategies, lock-free approaches based on atomic operations, and hybrid solutions combining both approaches showed that, under a low level of contention, all three strategies exhibit close results, whereas under a high level of contention the lock-free approach provides a substantial advantage in throughput – the throughput gain relative to the baseline mutex-based solution in a system with sixteen threads reached approximately 67% – while conceding to the mutex-based strategy in average latency of processing a single request under low load, owing to the overhead of repeated attempts of atomic operations [1]. The hybrid strategy in this study demonstrated intermediate but more stable characteristics on both indicators, allowing it to be regarded as a practical compromise for applications with a variable level of thread contention.

The key element linking the theoretical model with the possibility of its empirical verification is the kernel event-tracing subsystem. Modern operating systems provide low-level instrumentation mechanisms that allow the collection of data on process creation and termination, resource acquisition and release operations, and thread state transitions with minimal overhead: in the Linux environment this role is fulfilled by the eBPF (extended Berkeley Packet Filter) mechanism, which allows verified code to be executed directly in the kernel context without the need to modify the kernel itself, whereas in the Windows environment the analogous role is performed by the Event Tracing for Windows subsystem. Data collected in this manner constitute empirical confirmation (or refutation) of the behavior predicted by the formal model, and at the same time serve as the source material for the spatial representation of system dynamics discussed in the following subsection.

It is architecturally expedient to distinguish four functional components of the system for modeling and analyzing operating system process interaction, each responsible for a separate stage of transforming raw execution data into an interpreted model: a tracing module that performs low-level event collection; an event analyzer that structures the collected data, constructs a graph of process and resource dependencies, and forms a resource allocation graph for the subsequent detection of waiting cycles; a formal model generator that transforms the empirical data into a mathematical representation in the form of an extended temporal Petri net, defining the

set of places and transitions, the initial marking, and parameterizing the temporal delay function; and, finally, a formal verification and statistical analysis subsystem, which, on the basis of the constructed model, searches for deadlock states, verifies the safety and liveness properties of the system [12], and computes quantitative performance metrics – processor utilization, average resource waiting time, and the degree of contention for critical sections.

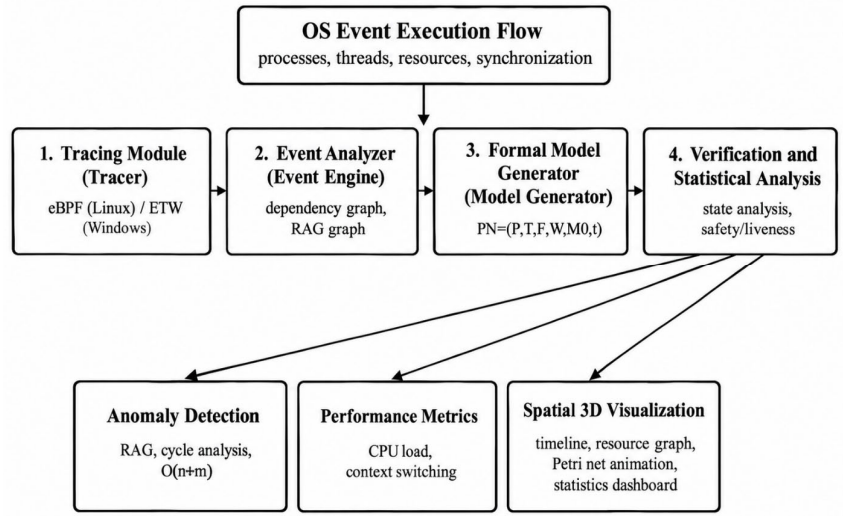


Figure 1. Functional model of the pipeline for analyzing operating system process and thread interaction

In Figure 1, the functional model is presented as a sequential data-processing pipeline connecting the four described components into a single research loop: the event stream captured by the tracing module during actual application execution is successively transformed by the event analyzer into a structured dependency, by the model generator into a formal description in the form of a Petri net, and by the verification subsystem into quantitative and qualitative conclusions regarding the correctness and effectiveness of synchronization. The fundamental difference between such a pipeline and traditional profiling tools lies in the fact that each of its stages retains a

connection both with specific system events (for debugging purposes) and with the formal mathematical model (for property verification purposes), which creates the foundation for the further spatial representation of results discussed in the following subsection.

3. Spatial Modeling and 3D Visualization of System Processes

The transition from textual and tabular representation of operating system state to spatial modeling is motivated by the fundamental limitation of two-dimensional means of information presentation in the presence of a large number of simultaneously interacting entities. A timeline diagram, displaying the activity of an individual thread along the horizontal time axis, remains informative for analyzing the behavior of one or a few threads, but loses clarity already at several dozen simultaneously active entities [13], since activity lines overlap one another, and causal relationships between events of different threads (for example, the transfer of a semaphore token from one thread to another) cannot be represented without additional graphical elements, which further complicates perception [14].

Spatial modeling solves this problem by introducing an additional dimension of information presentation: two dimensions of the plane are used to display structural interrelations (the topology of the process and resource dependency graph), while the third dimension or color coding additionally carries information about temporal dynamics or interaction intensity. Within the approach developed by the author, the three-dimensional representation is implemented in four complementary modes, each of which focuses attention on a separate aspect of system behavior. The timeline view remains the basic mode for displaying periods of thread activity, blocking, and waiting, but is supplemented with the ability to group by resource and color-code state. The resource graph displays the structure of process and resource dependencies in the form of a graph, in which the identification of cycles allows instant identification of potential or realized deadlocks. Petri net animation reproduces the step-by-step firing of model transitions, allowing observation of token movement between places in real time or in step-by-step playback mode, which is a means, unique to the spatial approach, of verifying the correspondence between the formal model and actual system behavior. Finally, the statistics dashboard aggregates quantitative performance metrics in a compact

form, supplementing the qualitative spatial representation with numerical characteristics.

Experimental verification of the proposed approach was carried out on the classical dining philosophers problem, a standard test case for deadlock research [9], as well as on a real multithreaded application – a parallel web server under load. The deadlock-detection system, built on the described principle of state-reachability analysis, achieved a one-hundred-percent success rate in detecting artificially created deadlocks in test scenarios, with an average analysis time of 127 milliseconds for a configuration of fifty processes and thirty resources, which is an acceptable indicator for interactive use during software development [1]. This result demonstrates that the spatial representation is not a purely illustrative supplement to formal analysis, but is directly integrated into the research loop: a cycle detected by the resource graph is immediately confirmed or refuted by step-by-step animation of the corresponding section of the Petri net.

The practical implementation of such a toolkit places a separate demand on the performance of the visualization tool itself, since displaying hundreds of simultaneously active elements in real time is a non-trivial computational task that, in turn, competes for processor and graphics-adapter resources with the application under study. A prototype interactive browser-based thread-interaction simulator developed by the author, built as a single-page React component without external dependencies on server infrastructure, illustrates this requirement: simulation of three synchronization strategies (mutex-based, lock-free, hybrid) with a parameterized number of threads, level of contention, and load intensity is reproduced using SVG animation and the Recharts charting library, with all performance metrics updated at an interval of 800 milliseconds, which imitates the character of real profiling while remaining within the computational capabilities of the client application without the need for hardware acceleration [1]. Notably, even under such a relatively modest load, the choice of client-application architecture – avoidance of redundant repainting, use of a declarative description of SVG primitives – becomes a decisive factor for the smoothness of display, which is a direct consequence of the regularities discussed further in the context of hardware-software interaction with the graphics processor.

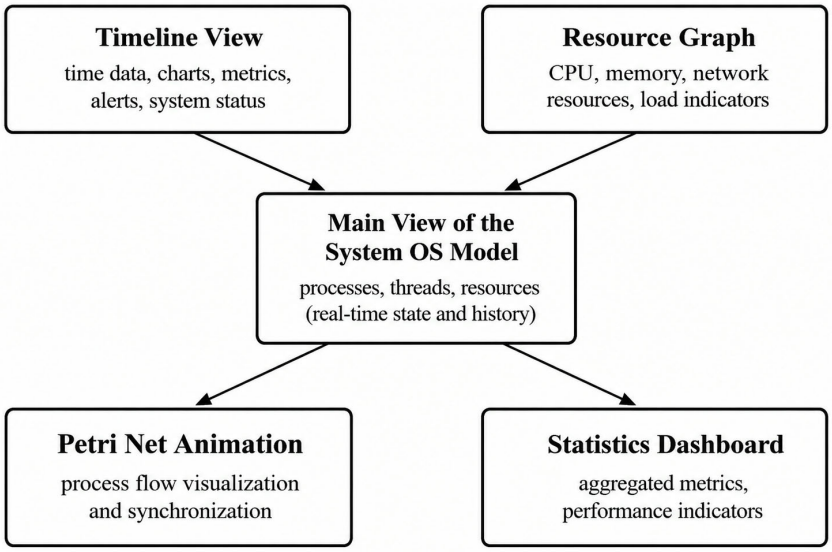


Figure 2. Multi-mode system for the spatial representation of process and thread interaction

Synthesizing the results of this subsection, it can be stated that spatial modeling performs a dual function in operating system research: on the one hand, it is a means of verification – the spatial representation allows the researcher to directly observe the correspondence or discrepancy between the formal model and the actual behavior of the system; on the other hand, it is an independent analytical instrument that allows the detection of regularities (for example, the clustering of deadlocks around certain types of resources) that remain hidden under purely numerical or textual analysis. At the same time, the real-time implementation of such a representation inevitably leads the researcher to the question of hardware-software interaction between the central and graphics processors, since it is precisely this interaction that determines the boundary of practical applicability of spatial visualization for large-scale systems – a question addressed in the following subsections.

4. Hardware-Software Interaction Among the CPU, GPU, and Operating System Kernel

The real-time implementation of three-dimensional visualization of internal operating system processes shifts the focus of research from the logic of the processes themselves to the architecture of the hardware-software interaction that enables the formation of an image on the basis of geometric data. This interaction encompasses a sequence of transformations: from the representation of scene-object geometry as a set of vertices in a local coordinate system – through affine transformations of the model, view (camera), and projection – to the formation of a raster image by the graphics processor under the coordination of the operating system kernel.

The orientation of an object in three-dimensional space – whether the position of a conditional thread marker in the scene or the orientation of a moving control-object element in industrial automation systems – is specified by a composition of elementary rotations about the coordinate axes, which in matrix form is written as the product of three rotation matrices parameterized by Euler angles:

$$R = Rz(\gamma) \cdot Ry(\beta) \cdot Rx(\alpha) \tag{4}$$

where R_x, R_y, R_z are the matrices of elementary rotations about the corresponding coordinate axes through angles α, β, γ. This composition allows the correct reproduction of an arbitrary orientation of moving scene elements, including both abstract process markers in research visualization and physical control objects – valves, actuators, rotary platforms – in applied human-machine interaction systems of the SCADA/HMI type [2].

After the transition to camera space, the vertex coordinates of the scene undergo perspective projection, which forms the subjective impression of depth – a fundamental property without which the three-dimensional representation would degenerate into an ordinary two-dimensional projection, deprived of the advantages of spatial information coding. The simplified dependence of the clip-space coordinate on the corresponding camera-space coordinate has the form

$$x_{clip} = (2n / (r - l)) \cdot x_{eye} + ((r + l) / (r - l)) \cdot w_{eye} \tag{5}$$

where n and f are the distances to the near and far clipping planes, and l and r are the horizontal bounds of the visible volume; analogous relations are defined for the y and z coordinates. After division by the homogeneous component w, normalized device coordinates are obtained, which the

graphics subsystem of the operating system converts into frame-buffer pixel coordinates according to the resolution of the output window [2].

It is essential for understanding the distribution of responsibility between the processor and the graphics adapter that the aforementioned geometric transformations in modern graphics systems are performed directly on the GPU within the vertex shader, whereas the central processor and operating system are responsible for forming the graphics context, preparing command buffers, and synchronizing the exchange between frame buffers (the swap chain). This distinction is key for the subsequent analysis of the rendering pipeline, since it is precisely the boundary between what is executed on the CPU and what is passed for execution on the GPU that determines the nature and cost of interaction between the two computing subsystems.

The qualitative perception of a three-dimensional scene is determined not only by the geometric correctness of the projection but also by the photometric plausibility of the image, which is ensured by illumination models. The most widespread among local illumination models is the Phong model, which describes the total illumination of a surface point as the sum of ambient, diffuse, and specular components:

$$I = k_a I_a + k^d I_l (N \cdot L) + k_s I_l (R \cdot V)^n \quad (6)$$

where k_a, k^d, k_s are the ambient, diffuse, and specular reflection coefficients of the surface material; I_a, I_l are the intensities of the ambient and point light sources; N, L, R, V are the unit vectors of the surface normal, the direction to the light source, the reflected ray, and the direction to the observer, respectively; n is the shininess exponent of the material, determining the size and sharpness of the specular highlight [2]. In the applied context of human-machine interaction systems, this model acquires an additional semantic function: the diffuse-reflection parameter can be dynamically modulated by the current value of a process parameter – for example, the color of a filled tank volume depending on liquid level, or the color of a pipeline depending on ambient temperature - which transforms the *3D* scene from a purely illustrative element into a full-fledged information layer of the automated process control system, directly integrated into the graphics subsystem of the operating system.

The performance of the described chain of transformations is determined not only by the computational complexity of the individual stages, but primarily by the nature of interaction between the central and graphics

processors within the operating system. Frame rate, as an integral indicator of the quality of interactive visualization, is inversely proportional to the total time required to form a single frame:

$$FPS = 1 / (T_{CPU} + T_{GPU} + T_{sync}) \quad (7)$$

where T_{CPU} is the time spent by the central processor preparing commands and scene logic, T_{GPU} is the time spent executing the graphics pipeline directly on the GPU, and T_{sync} is the synchronization waiting time, in particular the wait for the display's vertical blanking interval, administered by the graphics subsystem of the operating system [2]. The structure of this formula directly indicates that the performance of three-dimensional visualization is not a property exclusively of the graphics processor, but the result of coordinated operation of three components, any of which may become the dominant delay factor depending on the nature of the load and the specific implementation of the operating system's graphics stack – a question whose detailed structural analysis is conducted in the following subsection.

The theoretical computational performance of the GPU, which determines the upper bound of the TGPU component, is estimated as the product of the number of computing cores, the clock frequency, and the number of floating-point operations per cycle per core:

$$P_{GPU} = N_{cores} \cdot f_{clk} \cdot NFLOP/cyc \quad (8)$$

However, the practically attainable performance rarely approaches this theoretical limit owing to the limited bandwidth of video memory, which determines the speed of transferring vertex and texture data between system memory and video memory and is estimated by the relation

$$B_{eff} = V_{data} / T_{transfer} \quad (9)$$

where V_{data} is the volume of data transferred and $T_{transfer}$ is the transfer time. The effective value of bandwidth is directly influenced by the operating system, since it is precisely the graphics-processor driver that controls buffer placement in video memory and its caching policy [2]. A conducted computational experiment showed that for small volumes of transferred data (units to tens of megabytes), the effective bandwidth of both common types of video memory – GDDR6, with a peak value of 448 gigabytes per second, and HBM2, with a peak value of 1024 gigabytes per second – is substantially lower than the peak value owing to transfer-initialization overhead, whereas for volumes above 500 megabytes both characteristics

reach saturation, approaching their peak values [2]. For a typical scene of three-dimensional visualization of control systems, containing a moderate volume of geometry but a significant number of interface textures (mimic diagrams, status indicators), this saturation effect acquires practical significance in designing the structure of graphics resources and the operating system's caching policy for them.

5.Modeling the 3D Visualization Pipeline

The mathematical apparatus of geometric transformations and illumination discussed in the previous subsection is implemented not as a monolithic procedure, but as a sequence of clearly delineated stages of the graphics pipeline, each of which engages different combinations of central and graphics-processor computing resources under the coordination of the operating system. Scene assembly, shader-program compilation, and command-buffer construction take place predominantly on the application and user-space driver side, whereas rasterization, fragment processing, and post-processing are performed directly by the graphics processor; the intermediate link between these two groups of stages is the operating system kernel, which accepts the formed command buffer, schedules its execution on the GPU's hardware queues, and ensures synchronization of the moment the result becomes ready.

The architectural implementation of this mediation differs substantially between the major platforms. In operating systems of the Windows family, the graphics driver is built on the two-tier Windows Display Driver Model (WDDM), in which the kernel-mode component (Kernel-Mode Driver) executes in the privileged processor protection ring and has direct access to GPU hardware through memory-mapped input/output space, while the user-mode component (User-Mode Driver) is loaded into the context of each graphics application separately and is responsible for translating graphics API calls into a sequence of commands for the specific hardware [15-16]. This division provides a balance between performance – most of the work is performed without system calls, in user mode – and system stability, since an error in the user-mode component leads only to the crash of the corresponding application, whereas an error in the kernel-mode component is critical for the entire system [3]. In the Linux kernel, the analogous function is performed by the Direct Rendering Manager

subsystem in conjunction with the Kernel Mode Setting subsystem (DRM/KMS), whose architecture, unlike WDDM, is predominantly monolithic: the bulk of graphics-processor management logic – task scheduling, memory management, synchronization, display configuration – is placed directly in the kernel, which minimizes interaction overhead but simultaneously raises the cost of a driver error to the level of complete system failure [17].

A comparative analysis of the two architectures, summarized in Table 1, demonstrates that, despite the fundamental difference in the distribution of responsibility between driver levels, both models solve an identical set of fundamental tasks: video-memory management with isolation of process address spaces, task scheduling across several hardware queues of the graphics processor, synchronization between the central and graphics processors through the fence mechanism, and atomic output of the formed frame to the screen.

Table 1

Comparative characteristics of graphics processor management architectures

Characteristic	WDDM (Windows)	DRM/KMS (Linux)
Driver structure	two-tier (KMD + UMD)	predominantly monolithic (kernel)
Video memory management	VidMM subsystem	GEM / TTM frameworks
Cross-process buffer sharing	shared handle mechanism	dma-buf / PRIME
GPU task scheduling	VidSch, HAGS mode	DRM scheduler (drm_gpu_scheduler)
CPU–GPU synchronization	monitored fence	dma_fence primitive
Consequence of driver failure	application termination (UMD)	kernel failure (kernel panic)

Regardless of the specific implementation, analysis of both platform architectures makes it possible to distinguish five key points of interaction between the operating system kernel and the graphics hardware stack, each characterized by its own initiator, direction of control transfer, and involved components. The first interaction point corresponds to the allocation of video

memory for a newly created graphics resource – a vertex buffer, texture, or constant buffer – when the application's request is translated by the driver into a kernel system call, which reserves the corresponding memory block and establishes its mapping into the virtual address space of the graphics processor allocated to that process. The second and third interaction points cover the recording and submission of the command buffer: the application, through the graphics API, forms a sequence of rendering commands, which, after validation by the driver, is passed to the kernel, where the scheduler places the corresponding task into one of several independent hardware queues of the graphics processor – the graphics, compute, copy, or video-decode queue – which allows parallel execution of heterogeneous workloads on a single hardware device [18-19].

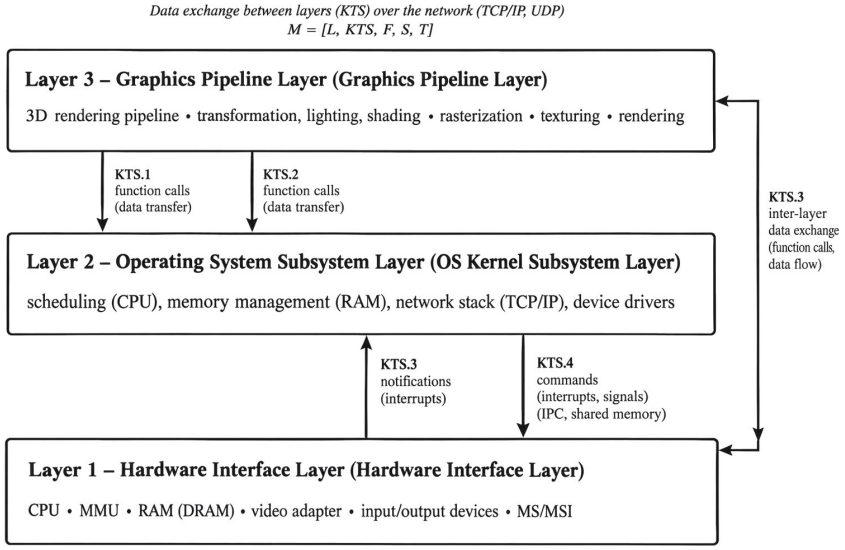


Figure 3. Three-level structural model of the interaction between the graphics processor and the operating system kernel, and the five key interaction points between levels

The fourth interaction point, unlike the three preceding ones, has the reverse direction: after completing execution of the command buffer, the

graphics processor informs the central processor and the operating system kernel through the fence mechanism – a lightweight synchronization object whose state changes from active to signaled after the corresponding operation is completed at the hardware level, causing a hardware interrupt whose handler wakes the central-processor threads that were waiting for this event [18; 19]. It is precisely this interaction point, as further empirical analysis shows, that is decisive for the total latency of the rendering pipeline. The fifth and final interaction point corresponds to the output of the formed frame to the screen: the ready frame buffer is passed to the display-management subsystem, which coordinates the moment of scan-address replacement with the display's vertical blanking period, preventing the visual artifact of frame tearing [20].

The presented structure allows the interaction between the graphics processor and the operating system kernel to be formally generalized as a hierarchical three-level model, a detailed examination of which is the subject of the following subsection, devoted to an integrated description of operating system processes and hardware-software interaction within a unified formal apparatus.

6. Integrated Model of Process and Hardware-Software Interaction

The material of the preceding subsections makes it possible to form a generalized conceptual model that combines, within a unified formal apparatus, three levels of abstraction: the hardware interface level, the operating system kernel subsystem level, and the three-dimensional visualization graphics-pipeline level – connected by the five key interaction points discussed in the previous subsection. Formally, this structural model is defined as the tuple

$$M = (L, KTB, F, S, T) \tag{10}$$

where $L = \{L_1, L_2, L_3\}$ is an ordered set of abstraction levels (hardware interface, kernel subsystems, graphics pipeline), the order relation of which specifies the hierarchy of subordination among the levels; $KTB = \{KTB_1, \ldots, KTB_5\}$ is the set of key interaction points; the function $F: KTB \to L \times L$ maps each interaction point to an ordered pair of levels (source, receiver), specifying the direction of control transfer; S is the extended set of synchronization states of a GPU task, encompassing active execution, waiting for input dependencies, the

signaled (completed) state, the error state, and the state of preemption by the scheduler with the possibility of subsequent resumption [21]; the function $T: KTB \rightarrow R_+$ assigns a measurable delay to each interaction point, enabling quantitative analysis and optimization of pipeline performance [3].

The fundamental distinction of the proposed integrated model from the separately considered models of process interaction (subsections 2–3) and the structural model of GPU–kernel interaction (subsection 6) lies in the fact that both formalisms - the extended Petri net $PN = (P, T, F, W, M_0, \tau)$ and the structural model $M = (L, KTB, F, S, T)$ - describe a fundamentally homogeneous class of phenomena: competition for shared resources under conditions of parallel execution, resolved by synchronization mechanisms, and differ only in the scale and nature of the entities considered. In the process interaction model, the places of the Petri net are the states of threads and operating-memory or file-system resources, and the transitions are calls to synchronization primitives; in the structural model of interaction with the GPU, an analogous role is played by the states of graphics processor tasks (active execution, waiting, completion signaling) and the key interaction points, which perform a function analogous to Petri net transitions – they change the state of the system and can be blocked by the absence of a necessary precondition (for example, the absence of a signaled state of an input dependency at interaction point KTB-3). This structural similarity makes it possible to formally regard the complete execution cycle of a single frame of three-dimensional visualization as a sequential composition of key interaction points, by analogy with the sequence of transition firings in a Petri net:

$$Frame = KTB_1 \rightarrow KTB_2 \rightarrow KTB_3 \rightarrow KTB_4 \rightarrow KTB_5 \qquad (11)$$

where the sequence symbol denotes a temporal precedence relation: each subsequent interaction point can begin only after the successful completion of the preceding one, or after the signaling of the corresponding synchronization fence in the case of asynchronous execution [3]. For pipelined rendering with several frames simultaneously in flight, the model is naturally generalized to a parallel composition: interaction points KTB_2 and KTB_3 of frame $k+1$ are capable of executing simultaneously with interaction points KTB_4 and KTB_5 of frame k, provided there is no resource conflict between these frames, which is the principal architectural mechanism for hiding CPU–GPU synchronization latency and, at the same

time, a direct analogy to the parallel firing of independent transitions in the extended Petri net modeling the interaction of operating system threads.

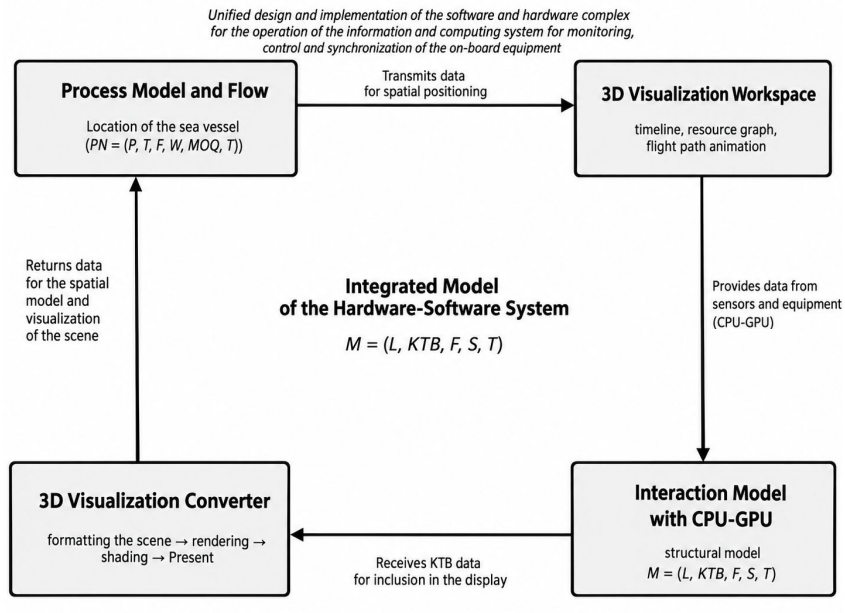


Figure 4. Integrated conceptual model of operating system process interaction, CPU–GPU hardware-software interaction, and the three-dimensional visualization pipeline

The unification of the two formalisms within a single integrated model makes it possible to formulate the end-to-end research chain that determines the structure of the entire chapter of the monograph: the processes and threads of the operating system, interacting through synchronization mechanisms, form the execution dynamics of the three-dimensional visualization application; this dynamics is transmitted to the graphics pipeline in the form of command buffers, whose scheduling is determined by the interaction between the OS kernel and the graphics-processor driver; the result of pipeline execution, in turn, forms precisely the spatial representation that the researcher uses to analyze the original

dynamics of processes and threads – closing the research loop into a self-consistent system in which the object of study (OS process interaction) and the means of study (three-dimensional visualization of this interaction) are subject to unified regularities of parallel execution, resource competition, and synchronization.

The practical value of such a unification is confirmed empirically: by the method of system-call tracing under a real reference workload – rendering a scene with one million triangles – it was established that the total frame latency in the Windows 11 environment is approximately 718 microseconds, of which 86.9% is attributable to the fourth interaction point, that is, to the synchronization of the central and graphics processors through the fence mechanism, whereas the aggregate contribution of the resource-preparation stages (the first, second, and third interaction points) does not exceed 6.5% of the total latency [3]. A comparative measurement in the Linux environment (DRM/KMS subsystem) yielded a close result – 733 microseconds, exceeding the Windows figure by approximately 2.1% and lying within the measurement error attributable to differences in the implementation of the command ring-buffer scheduler between the two subsystems rather than to a fundamental architectural distinction [3]. This empirical result is in direct agreement with formula (7), localizing the dominant term T_{sync} precisely within the interaction point modeling CPU–GPU synchronization, and at the same time confirms the applicability of the proposed integrated model for quantitative, and not merely qualitative, analysis of the performance of a 3D visualization system for internal operating system processes.

7.Analysis of the Proposed Approach: Advantages, Limitations, and Application Domains

A critical analysis of the proposed integrated model should be conducted along several interrelated lines: comparison with traditional means of representing operating system processes, evaluation of the scalability of the spatial approach, consideration of the role of the graphics processor as a computing platform for the analysis itself, and generalization of the synchronization problems arising at the boundary between the software and hardware levels of the system.

Compared with traditional textual event logs and numerical performance tables, the spatial model provides a fundamentally higher information

density of presentation: a single frame projection of a three-dimensional scene is capable of simultaneously conveying the state of dozens of threads, the structure of their dependencies on shared resources, and the dynamics of this state's change over time – an amount of information that, in textual presentation, would require sequential review of many pages of a log. At the same time, this advantage has its cost: constructing and maintaining the formal model (a Petri net or the structural model of GPU interaction) necessary for correct interpretation of the visualization requires substantial expenditure at the design stage of the analysis system, and the interpretation of a complex three-dimensional scene requires the researcher to have a certain level of preparation, whereas a numerical table, despite its limited in formativeness, remains unambiguously and directly interpretable without additional training.

The question of the scalability of the proposed models requires separate consideration in two aspects – computational and perceptual. From the computational standpoint, the deadlock-detection algorithm based on constructing a resource allocation graph has linear time complexity with respect to the total number of processes and resources, which is confirmed empirically: analysis of a configuration with fifty processes and thirty resources is performed in 127 milliseconds, which allows the approach's suitability to be projected for systems an order of magnitude larger without a qualitative change in the nature of computational cost [1]. A considerably more significant limiting factor is perceptual scalability: a human researcher is able to effectively interpret a three-dimensional scene with a limited number of simultaneously visible elements, and growth in the number of threads beyond a few dozen inevitably requires the application of additional techniques – hierarchical collapsing of thread groups, filtering by activity criteria, temporal scaling – which go beyond purely geometric representation and require separate research into data-aggregation methods for large-scale scenes, which constitutes one of the promising directions for further development of the proposed approach.

The dual role of the graphics processor within the concept under consideration deserves particular attention: on the one hand, the GPU is an object of study – it is precisely its interaction with the operating system kernel that constitutes the subject of the structural modeling discussed in subsections 7.5–7.6; on the other hand, the GPU is a research instrument –

a means of hardware acceleration of the three-dimensional visualization itself, which makes it possible to display complex scenes in real time. The conducted computational experiment quantitatively confirms the practical significance of this second aspect: for scenes of medium and high geometric complexity, hardware-accelerated rendering provides approximately an order of magnitude higher performance compared with software rasterization by the central processor – at a scene polygon count of one million, the hardware implementation demonstrates approximately 714 frames per second versus 71 frames per second for the software implementation [2]. This ratio has direct practical significance for the design of operating system process visualization systems: a complex scene displaying hundreds of simultaneously active threads with detailed geometry of markers and connecting lines, in the absence of hardware acceleration, risks itself becoming an analysis bottleneck, introducing its own delay comparable to or exceeding the characteristic time scales of the events under study.

The problem of CPU–GPU synchronization, discussed in detail in subsection 7.6 with respect to the graphics pipeline, acquires an additional dimension in the context of this research, since the system for 3D visualization of internal OS processes is itself an application executed within the same operating system and subject to the same regularities of scheduling and synchronization that it is intended to study. This gives rise to a methodological problem of the recursive influence of the observation tool on the object under study: intensive load on the graphics pipeline from the visualization system itself is capable of altering the operating system's thread-scheduling characteristics (for example, through competition for processor time between the threads of the application under study and the rendering threads), which requires consideration during the interpretation of results and, where possible, execution of the visualization in a separate process or even on a separate computing node isolated from the system under study.

To summarize, it should be stated that the proposed integrated approach is not a universal tool suitable for an arbitrary scale and an arbitrary class of tasks, but a specialized analysis method most effective in the range from several dozen to several hundred simultaneously active entities, characterized by a comparatively high cost of initial implementation

(constructing the formal model, developing tracing and visualization software tools) and, at the same time, a substantial advantage in clarity and speed of detecting hidden regularities in system behavior compared with traditional analysis tools. The most promising areas of application for this approach are the development and debugging of real-time multithreaded applications, the study and optimization of operating system graphics subsystems, and the design of human-machine interfaces for automated process control systems (SCADA/HMI), where three-dimensional visualization simultaneously performs a research and an applied function – both as a means of analyzing system behavior and as a means of providing the operator with real-time information on the state of the controlled object.

CONCLUSIONS

The conducted research made it possible to form a coherent scientific concept for modeling process and hardware-software interaction in modern operating systems and three-dimensional visualization systems, integrating the results of the analysis of process and thread interaction, spatial modeling of control systems, and structural analysis of the interaction between the graphics processor and the operating system kernel within a single formal apparatus. Based on the results of the chapter, the following scientific results and conclusions can be formulated.

First, it was established that the interaction of operating system processes and threads and the interaction of the graphics processor with the operating system kernel, despite the substantial difference in the scale and nature of the entities considered, are subject to structurally similar regularities of parallel execution, competition for shared resources, and synchronization through wait-signal primitives, which allows both classes of phenomena to be described by consistent formalisms – an extended Petri net with temporal characteristics and a structural model of key interaction points, respectively – and to unite them within a single integrated model of hardware-software interaction $M = (L, KTB, F, S, T)$.

Second, it was proven that spatial modeling and three-dimensional visualization are not merely a means of visually presenting operating system research results, but an independent analytical instrument that allows the simultaneous observation of temporal dynamics and structural organization of the interaction of processes, threads, and hardware resources, ensuring the

detection of deadlocks, performance bottlenecks, and behavioral anomalies not accessible to detection by traditional textual or tabular analysis.

Third, it was quantitatively confirmed that the hardware-software interaction of the central and graphics processors within the operating system is a determining factor for the performance both of the three-dimensional visualization pipeline itself and of the OS process research tools built on its basis: it was empirically established that CPU–GPU synchronization through the fence mechanism constitutes the dominant share of total frame latency (86.9% according to tracing results in the Windows 11 environment), and that hardware-accelerated rendering provides approximately an order of magnitude higher performance compared with the software implementation for scenes of medium and high geometric complexity, which determines the practical limits of applicability of spatial visualization for large-scale systems.

Fourth, a methodological feature of the proposed approach was identified, consisting in the recursive nature of the relationship between the object and the means of study: the system for three-dimensional visualization of operating system processes is itself an application executed within the same operating system and subject to the same regularities of scheduling and synchronization that it is intended to study, which requires consideration during the architectural design of similar research instruments, in particular the isolation of the observation tool's load from the load of the system under study.

The practical value of the results obtained lies in the possibility of using the proposed integrated model and the developed software toolkit for the analysis and optimization of multithreaded applications at the development stage, for the study and improvement of operating system graphics subsystems, and for the design of human-machine interfaces of automated process control systems with support for three-dimensional visualization. Promising directions for further research include extending the integrated model to heterogeneous computing systems combining several types of accelerators (CPU, GPU, specialized neural processors), developing methods of hierarchical data aggregation to ensure the perceptual scalability of spatial visualization for numbers of simultaneously active entities exceeding several hundred, as well as investigating the applicability of the proposed approach in graphics-processor virtualization environments and in real-time systems with hard timing constraints.

References:

1. Humennyy P. V. (2026). Modeliuvannia ta vizualizatsiia vzaiemodii protsesiv i potokiv u suchasnykh operatsiinykh systemakh [Modeling and visualization of the interaction of processes and threads in modern operating systems]. *Informatyka ta matematychni metody v modeliuvanni,* vol. 16, no. 2, pp. 294–304. DOI: https://doi.org/10.15276/imms.v16.no2.294

2. Humennyy P. V. (2026). Metody prostorovoho modeliuvannia ta 3D-vizualizatsii v suchasnykh operatsiinykh systemakh [Methods of spatial modeling and 3D visualization in modern operating systems]. *Visnyk Natsionalnoho universytetu vodnoho hospodarstva ta pryrodokorystuvannia,* vol. 2, no. 114, pp. 89–107. DOI: https://doi.org/10.31713/vt220267

3. Humennyy P. V. (2026). Strukturna model vzaiemodii hrafichnoho protsesora z yadrom operatsiinoi systemy v konteksti realizatsii konveiera 3D-vizualizatsii [Structural model of interaction between the graphics processor and the operating system kernel in the context of implementing a 3D visualization pipeline]. *Nauka i tekhnika sohodni*, no. 4(58), pp. 3108–3132. DOI: https://doi.org/10.52058/2786-6025-2026-4(58)-3108-3132

4. Microsoft Corporation (2024). Windows Display Driver Model (WDDM) Architecture. Microsoft Learn.

5. Popova-Zeugmann L. (2013). Time and Petri Nets. Berlin: Springer, 206 p.

6. Wang J. (1998). Timed Petri Nets: Theory and Application. Boston: Springer, 254 p.

7. Clarke E. M., Grumberg O., Peled D. A. (1999). Model Checking. Cambridge: MIT Press, 314 p.

8. Gnilenko A. (2022). Imitatsiine modeliuvannia bahatopotokovoi arkhitektury kompiuternoi systemy zasobamy ANYLOGIC [Simulation modeling of a multithreaded computer system architecture using ANYLOGIC]. DOI: https://doi.org/10.34185/1562-9945-1-138-2022-15

9. Herlihy M., Shavit N. (2020). The Art of Multiprocessor Programming. 2nd ed. Burlington: Morgan Kaufmann, 536 p.

10. Murata T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580.

11. Peterson J. L. (1981). Petri Net Theory and the Modeling of Systems. Englewood Cliffs: Prentice Hall, 290 p.

12. Schönberger C., Schaefer I., Knapp A. (2013). Runtime verification of multi-threaded programs. Formal Methods and Software Engineering. *Lecture Notes in Computer Science*, vol. 8144, pp. 492–507.

13. Geimer M., Wolf F., Wylie B. J. N., Ábrahám E., Becker D., Mohr B. (2010). The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience,* vol. 22, no. 6, pp. 702–719.

14. The Linux Kernel Community (2024). GPU Driver Developer's Guide: Introduction to the DRM Subsystem. The Linux Kernel Documentation.

15. Zimmermann T. (2023). The Linux graphics stack in a nutshell, part 1 & 2. Linux Kernel Developers Blog.

16. OpenLib (2025). KMS, DRM, and Direct Rendering in Linux. OpenLib.IO.

17. Guo Z., Li J., Zhang Y. (2025). Algorithmic Techniques for GPU Scheduling: A Comprehensive Survey. *Algorithms*, vol. 18, no. 7, p. 385. DOI: https://doi.org/10.3390/a18070385

18. Singh J., Olmedo I., Capodieci N., et al. (2022). Reconciling QoS and Concurrency in NVIDIA GPUs via Warp-Level Scheduling. Proceedings of Design, Automation & Test in Europe (DATE). DOI: https://doi.org/10.5555/3539845.3540146.

19. Fais A., Panagiotou S., Georgiou E. (2020). Multithreaded Rendering for Cross-Platform 3D Visualization Based on Vulkan API. Proceedings of the 15th International Conference on Computer Graphics Theory and Applications, pp. 80–88. DOI: https://doi.org/10.5220/0010108900002628

20. Huang S., Li X., Zhang Y. (2025). NeutriNo: Fine-grained GPU Kernel Profiling via OS Paradigm. Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2025).

21. Corbet J., Rubini A., Kroah-Hartman G. (2005). Linux Device Drivers. 3rd ed. O'Reilly Media, 636 p.