IBIMA Publishing
Communications of International Proceedings
https://ibimapublishing.com/p-articles/45SE/2025/4533525/
Vol. 2025 (22), Article ID 4533525

A Signal-Based Dependency-Aware Execution Model for Task Scheduling in Multicore Systems*

Patryk SERAFIN

Faculty of Cybernetics, Military University of Technology,
00-908 Warsaw, Kaliskiego 2 Street, Poland
ORCID: 0009-0008-0573-1976

Correspondence should be addressed to: Patryk SERAFIN, patryk.serafin@wat.edu.pl

* Presented at the 45th IBIMA International Conference, 25-26 June 2025, Cordoba, Spain

Abstract

Modern multicore systems rely on operating system schedulers to allocate processor time among concurrently running threads. While these schedulers are effective at balancing workload priorities and preserving fairness, they typically lack visibility into logical task dependencies at the application level. As a result, threads may be prematurely dispatched and enter a passive waiting state, occupying CPU resources unnecessarily and degrading overall system efficiency. The proposed study addresses this limitation by introducing a dependency-aware execution model that integrates with existing schedulers without modifying kernel behavior. The model intercepts task dispatch requests and ensures that execution occurs only when all declared dependencies have been satisfied. This is achieved through a reactive, signal-driven mechanism that dynamically evaluates task readiness using externally managed dependency vectors. The framework supports both static and dynamically arriving workloads and is agnostic to the underlying operating system, making it adaptable to a wide range of environments. The approach is theoretically justified by analyzing thread behavior and dispatch logic in Linux and Windows systems, with emphasis on minimizing idle resource contention. The model aims to improve processor utilization in complex scheduling scenarios by preventing wasteful thread activation and enhancing execution order fidelity in the presence of inter-task constraints.

Keywords: Multicore Scheduling, Task Dependencies, Thread Dispatch, Execution Control

Introduction

Modern computing environments increasingly depend on multicore architectures to manage parallel workloads with growing complexity. The effective utilization of such systems hinges on the underlying scheduling policies employed to allocate processor time across threads and processes. General-purpose operating systems like Linux and Windows implement sophisticated kernel-level schedulers that prioritize responsiveness, fairness, and throughput across heterogeneous workloads (Microsoft, 2021; Linux Foundation, 2025).

Although these schedulers are robust and efficient in many scenarios, they typically lack awareness of task-level dependency structures. In many real-world computing problems, the execution of specific tasks is contingent on the completion of others. Without built-in dependency handling, the operating system may indiscriminately schedule tasks that are logically blocked. This leads to inefficient CPU usage, unnecessary context switches, and resource contention, as threads are activated only to wait passively for prerequisite conditions to be met (Microsoft, 2022; Linux Foundation, 2010).

This limitation motivates the introduction of an external model for task management that operates in user space and complements system schedulers by evaluating task dependencies before execution. Instead of attempting to replace kernel-level schedulers, this approach introduces a lightweight control layer that filters out tasks with

Cite this Article as: Patryk SERAFIN, Vol. 2025 (22) "A Signal-Based Dependency-Aware Execution Model for Task Scheduling in Multicore Systems " Communications of International Proceedings, Vol. 2025 (22), Article ID 4533525, https://doi.org/10.5171/2025.4533525

unresolved prerequisites. As a result, only tasks that are ready for execution are dispatched, leading to more efficient use of processor resources in systems with high concurrency and dependency density.

The proposed solution defines a dependency-aware task execution framework that accommodates both statically defined task graphs and dynamically arriving workloads. Dependencies are expressed through explicitly maintained vectors and monitored via inter-thread signaling mechanisms, without requiring kernel modifications. This architecture offers compatibility across platforms and supports transparent task management for parallel applications.

Existing literature on task scheduling has focused largely on optimizing execution order, minimizing makespan, and balancing processor loads in both static and dynamic contexts (Topcuoglu and Hariri, 2002; Pinedo, 2016). Some recent studies have introduced dependency-aware schedulers for distributed or heterogeneous systems (Lyberis et al., 2016; Cheng et al., 2024), but integration with system-level threading remains limited. Meanwhile, foundational documentation from Microsoft and the Linux Foundation describes low-level thread lifecycle management but does not incorporate dependency-driven execution control (Microsoft, 2025; Linux Foundation, 2010).

In light of these limitations, this work proposes a novel, dependency-aware execution model that overlays system-level scheduling. The approach offers a unified mechanism for reducing idle thread occupancy and improving scheduling fidelity without modifying operating system kernels. The proposed design contributes a practical and portable method for enhancing task execution in multicore systems.

Background and Related Work

Task scheduling remains a critical research topic in the field of parallel computing and multicore systems. With modern processors integrating multiple cores, the efficient execution of concurrent workloads has become central to achieving high system performance and resource utilization. Operating systems such as Windows and Linux have developed advanced kernel-level schedulers to manage this complexity. These schedulers use preemptive strategies, time-sharing mechanisms, CPU affinity handling, and dynamic prioritization to ensure that available cores are fairly and efficiently utilized.

Despite these capabilities, native operating system schedulers are not equipped to manage explicit inter-task dependencies. While they handle thread queuing and context switching based on priority or fairness, they lack awareness of higher-level logical constraints between tasks. In practical terms, this means that a thread may be launched by the scheduler even though it cannot proceed until another task completes. This can result in unnecessary CPU occupation and performance bottlenecks, particularly in dependency-rich execution scenarios.
The Microsoft documentation titled “Scheduling” and “Process and Thread Functions” outlines how threads are prioritized and dispatched in the Windows operating system. Threads are maintained in priority-based ready queues and are selected for execution by the kernel-mode dispatcher component, but no mechanism is provided to defer execution based on dependency conditions unless implemented explicitly in user space. Similarly, “Windows Kernel-Mode Process and Thread Manager” describes the internal structures used to manage threads (such as ETHREAD and EPROCESS), but does not offer native support for dependency vectors.

On Linux systems, the “Linux Scheduler – Kernel Documentation” details the behavior of the Completely Fair Scheduler (CFS), which balances load across cores without dependency awareness. The “Workqueue” subsystem allows the deferral of work but lacks integrated signaling or synchronization mechanisms for interdependent threads. Furthermore, the “Kernel Stacks (x86-64)” documentation elaborates on the low-level handling of thread state and stack context, but similarly does not extend into abstract dependency models.

This gap in functionality has led researchers to explore higher-level approaches that augment existing operating system scheduling with dependency management mechanisms. Several studies have proposed solutions that introduce external scheduling logic, dependency graphs, or cooperative execution policies to better align thread dispatching with application-level constraints. The lack of such mechanisms in general-purpose operating systems motivates the development of lightweight, user-space models that can manage task readiness and inter-task relationships without altering the operating system’s core behavior.

The model presented in this study is designed to address this exact limitation. Rather than attempting to replace the system scheduler, it operates as an additional decision layer that controls when a task is eligible for execution based on explicit dependency resolution. This architecture enables compatibility with both static and

dynamically generated workloads, allowing for flexible integration into existing task pipelines.

Example of Thread Lifecycle and Scheduling on Windows Systems

The Windows operating system provides a well-documented model for managing processes and threads at the kernel level. When a thread is created, the system allocates internal kernel-mode structures such as the EPROCESS object for processes and the ETHREAD object for individual threads. These objects encapsulate critical information including scheduling parameters, priority levels, and processor affinity, as described in the documentation titled About Processes and Threads (Microsoft, 2021) and Windows Kernel-Mode Process and Thread Manager (Microsoft, 2025).

Once initialized, a thread is placed into the system's ready queue, which is structured according to thread priorities. The Windows scheduler then selects threads from these queues using a priority-driven pre-emptive scheduling algorithm, as outlined in Scheduling (Microsoft, 2021). The dispatcher component handles context switching by saving the state of the currently running thread and restoring the state of the next selected thread. A simplified diagram illustrating the process of assigning a newly created thread to a CPU core is shown in the Fig. 1.

The scheduling mechanism also supports dynamic behavior. For instance, when I/O operations complete, the associated thread may receive a temporary priority boost to ensure responsiveness. Additionally, the system maintains fairness across threads by monitoring CPU usage and adjusting execution windows accordingly. These aspects are further detailed in Process and Thread Functions (Microsoft, 2022).

When a thread is dispatched for execution, it runs on an available processor core until it either completes its task, voluntarily yields control, or is pre-empted by a higher-priority thread. During execution, the thread may access shared resources, perform system calls, or generate new tasks depending on application logic. The entire lifecycle is managed by the kernel with minimal involvement from user-space components.

Understanding the mechanics of this lifecycle is crucial for identifying performance bottlenecks in concurrent execution. In particular, launching threads before their dependencies are resolved can lead to idle waiting or resource contention. These inefficiencies are not addressed by default in the Windows scheduler, which is not aware of inter-task relationships. The proposed dependency-aware execution model addresses this limitation by adding an intermediate user-level layer that evaluates dependency satisfaction before thread activation. This approach allows better coordination between task readiness and system-level scheduling decisions without modifying the internal workings of the kernel.

Overview of sample research work

Numerous research efforts have addressed improvements in task scheduling by incorporating knowledge of task dependencies and execution constraints. One influential contribution is the work by Topcuoglu and Hariri, who proposed the HEFT (Heterogeneous Earliest Finish Time) algorithm for heterogeneous computing environments. This algorithm considers both task computation times and inter-task communication costs to generate an optimized execution schedule. It is particularly well suited for static task graphs where the complete dependency structure is known in advance, but it lacks adaptability to dynamic workloads (Topcuoglu and Hariri, 2002).

In the context of production-level computing systems, several studies by Nowicki and colleagues have introduced enhanced scheduling techniques. These include decision-making models for task allocation in constrained environments, as described in Nowicki (2012), as well as productivity-oriented cooperative approaches to project scheduling (Nowicki and Waszkowski, 2017). Additionally, Burzyński, Nowicki, and Waszkowski (2025) proposed the use of genetic algorithms to solve complex scheduling problems involving large sets of interdependent tasks. These approaches aim to improve system throughput and execution efficiency but generally require centralized planning and preprocessing.

More recent approaches explore scalable, decentralized scheduling mechanisms. The Myrmics system introduced by Lyberis et al. (2016) presents a dependency-aware task scheduler designed for manycore processors. It employs a region-based memory model and hierarchical task decomposition to minimize coordination overhead. Shankar et al. (2013) examined the role of operating system-level mechanisms in promoting energy-efficient scheduling and advocated for kernel-level frameworks that can dynamically adjust

task execution based on system state and workload characteristics.

Research by Cheng et al. (2024) investigated dependency-aware scheduling in real-time systems, specifically within connected and autonomous vehicles. Their model uses diffusion-based reinforcement learning to manage complex task graphs, emphasizing the risks of improper execution sequencing in time-sensitive applications.

Despite these advances, most existing solutions either require significant architectural changes, depend on specialized hardware, or are tailored to specific application domains. There remains a lack of lightweight mechanisms that can be integrated with general-purpose operating systems to improve task scheduling without modifying the kernel. The model proposed in this study attempts to fill this gap by introducing a user-level component that handles dependency verification before task dispatch. It provides a practical approach to enforcing execution correctness without altering the underlying scheduling policies of systems such as Windows or Linux.

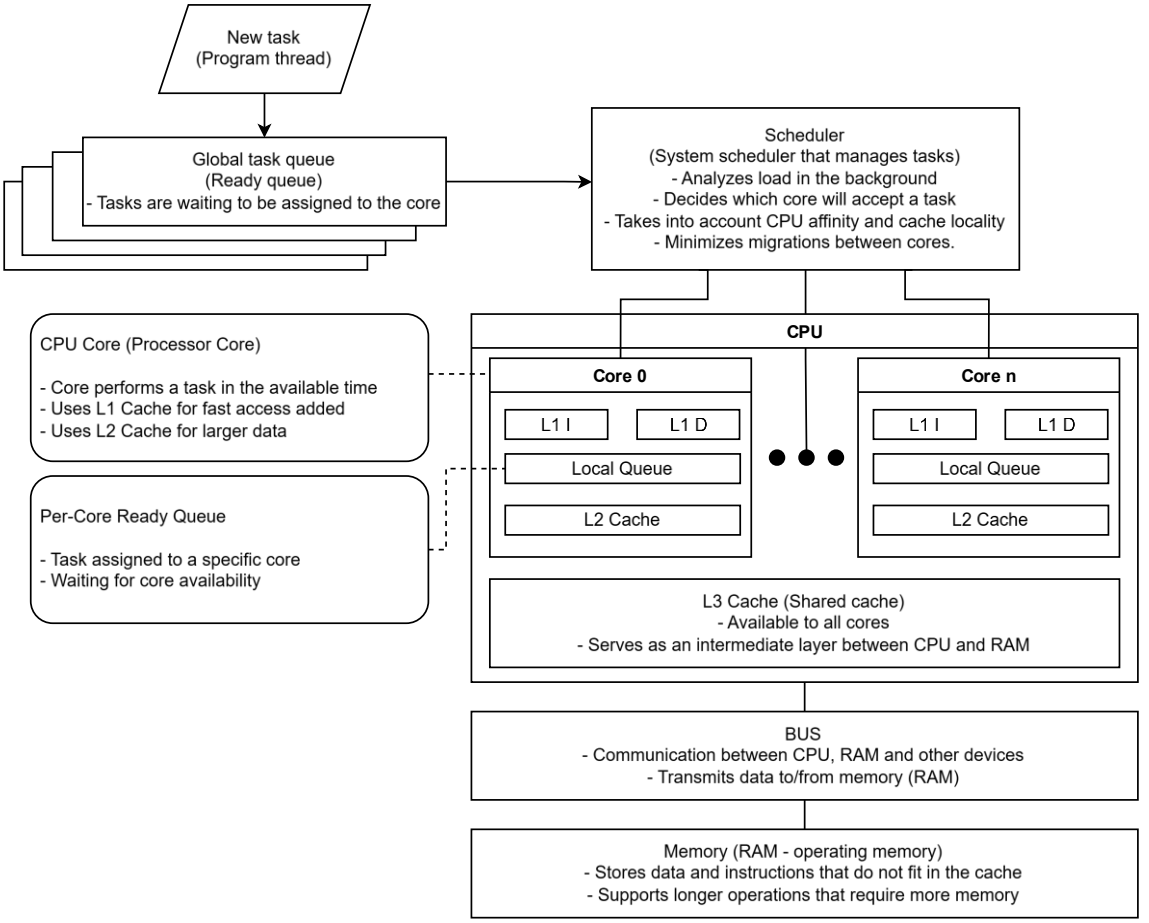

Fig. 1 Simplified diagram illustrating the process of assigning a newly created thread to a CPU core, including its transition through the scheduling system and final execution on a logical processor.

Model Description

Efficient execution of dependent tasks in modern multicore systems requires a robust mechanism that ensures tasks are launched only when all necessary prerequisites are satisfied. The proposed model introduces a lightweight, scalable solution by enhancing each task with an independent termination signal, effectively decoupling the dependency management from direct task identifiers.

Dependency Management Through Signals

Instead of representing dependencies through task IDs, each task in the model is associated with a dedicated termination signal. Upon the successful completion of a task, the corresponding signal is activated. Dependent tasks maintain a vector of such signals, and their readiness is determined by monitoring the activation of these

signals rather than checking task states explicitly. This signal-based approach simplifies the readiness checking mechanism, enabling faster, lock-free verification of task dependencies. Tasks automatically transition to the execution-ready state when all associated signals are set, reducing synchronization overhead compared to models requiring active dependency tracking via task identifiers.

Execution Policies

The model allows flexible policies regarding when tasks are launched relative to their dependencies:

- Strict Dependency Policy: A task is allowed to start only once all its dependency signals have been set. This ensures that no task consumes processor resources unless it is fully ready for execution.
- Partial Dependency Policy (alternative but not primary focus): A task may start after the fulfillment of the first dependency, but it must perform passive waiting whenever subsequent dependencies are not yet satisfied. While theoretically possible, this approach may lead to inefficient processor utilization, as partially started tasks could block valuable computational resources.

Given the above, the strict policy is considered the primary and recommended approach within the model. It maximizes processor efficiency by ensuring that tasks compete for resources only when they are fully ready, minimizing idle or blocking periods on cores.

Task Structure and Scheduling Flow

Each task is initialized with:
- A unique termination signal.
- A vector of dependency signals corresponding to its prerequisites.
- An optional CPU affinity hint to favor certain cores during scheduling.

The overall execution flow is as follows:
- Tasks are generated or arrive dynamically.
- Tasks with no dependencies are immediately ready for execution.
- Tasks with dependencies monitor their signal vector.
- Once all signals in a task's dependency vector are activated, the task is dispatched for execution on an available core.

A comprehensive illustration of the internal operation of the proposed model is provided in Fig. 2. This schematic representation presents the logical flow within the Dependency-Aware Model (DAM), beginning from the moment a task enters the system. Tasks without dependencies are directly forwarded to the system scheduler for immediate execution. Tasks with an associated dependency vector are placed under the supervision of DAM, which monitors the readiness of each dependency signal. DAM operates in an event-driven manner, reacting only when notified of a change in dependency status thereby eliminating unnecessary polling overhead. Once all required signals are set, the task is dispatched for scheduling and execution on an available core. Upon completion, the task emits its own signal, enabling the readiness evaluation of downstream dependent tasks.

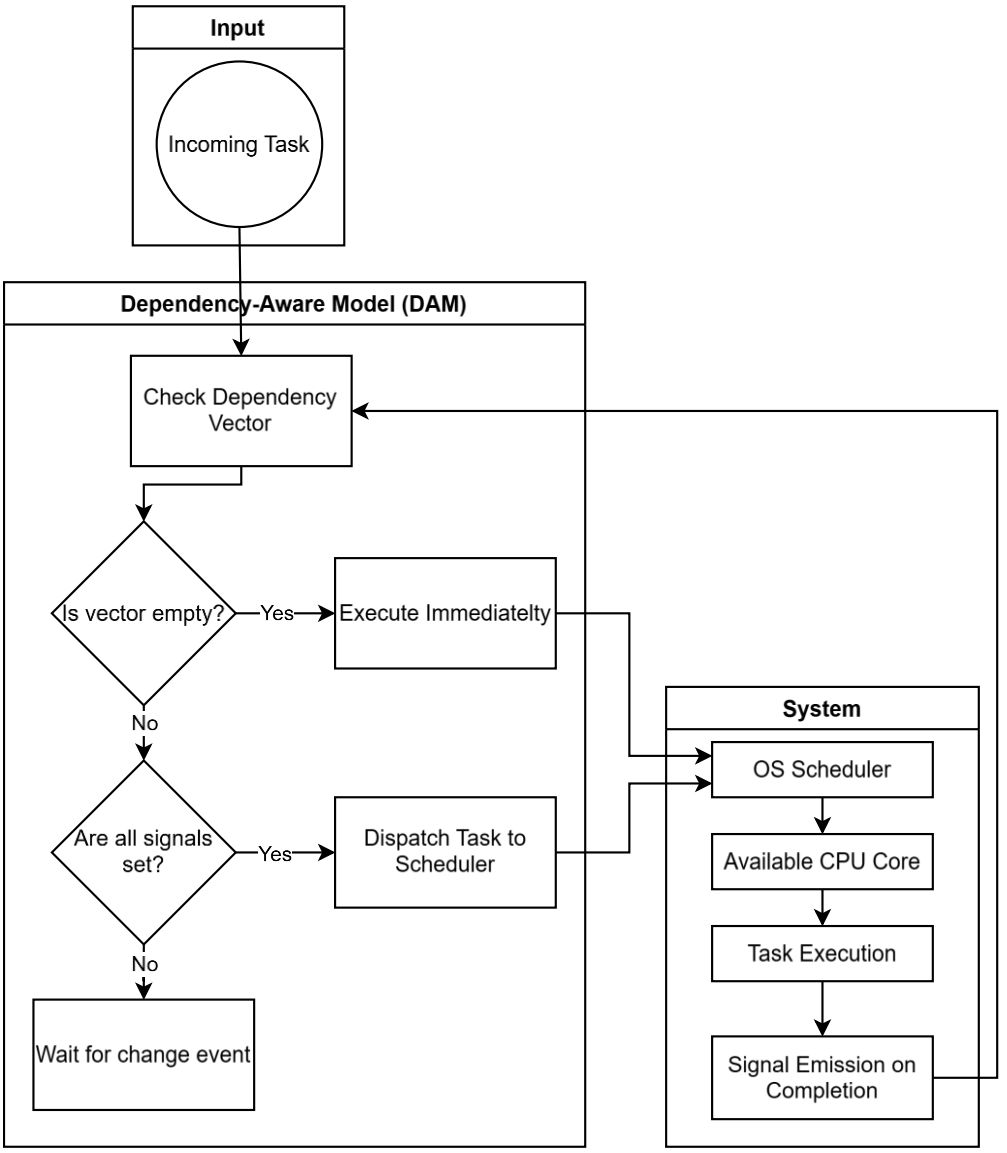

Fig. 2 Logical Architecture of the Dependency-Aware Model (DAM)

Expected Benefits

The primary theoretical advantage of the model is its ability to prevent premature task activation. By ensuring that only fully ready tasks enter the execution phase, the model aims to:

- Minimize processor idle times caused by unnecessary waiting inside tasks.
- Reduce contention for CPU resources.
- Optimize overall task throughput, especially in workloads characterized by significant inter-task dependencies.

The effectiveness of the model is expected to be particularly noticeable in environments with a high ratio of dependent tasks, where traditional schedulers might otherwise suffer from excessive passive waiting or context-switch overheads.

Formal mathematical notation

A representative example of a task dependency graph is presented in Fig. 3. In this graph:

- T – denotes an individual task, uniquely identified by its number.
- Dep. (Dependencies) – indicates a list of signals from preceding tasks that must be completed before a given task can start.
- SR-x (Signal Ready x) – represents the completion signal emitted by Task x once its execution is finished.

Each directed edge in the graph symbolizes a dependency between tasks: when a task is complete, it emits a

corresponding signal (SR-x) that satisfies the dependencies of subsequent tasks. This modeling approach allows for a flexible yet precise representation of task relationships without requiring direct task ID management during execution.

The task dependency graph can be formally described as a directed acyclic graph (DAG) $G = (V, E)$, where:

- V is a finite set of vertices representing tasks T_i.
- $E \subseteq V \times V$ is a set of directed edges, where an edge (T_i, T_j) indicates that task T_j depends on the completion of task T_i.

Each task T_j is associated with a dependency set $D(T_j) \subseteq V$, which contains all tasks that must completed before T_j can start execution. To model readiness for execution:

- Each task T_i emits a signal SR_i upon its completion.
- The readiness of task T_j is defined as:

$$Ready(T_j) = \bigwedge_{T_i \in D(T_j)} SignalReady(T_i)$$

where:

- $Ready(T_j)$ is a boolean function indicating whether task T_j can be executed,
- $SignalReady(T_i)$ is a boolean signal indicating the completion of task T_i,
- $\bigwedge$ denotes the logical AND operation over all required signals.

Thus, a task becomes executable only when all required signals from its dependencies are set.

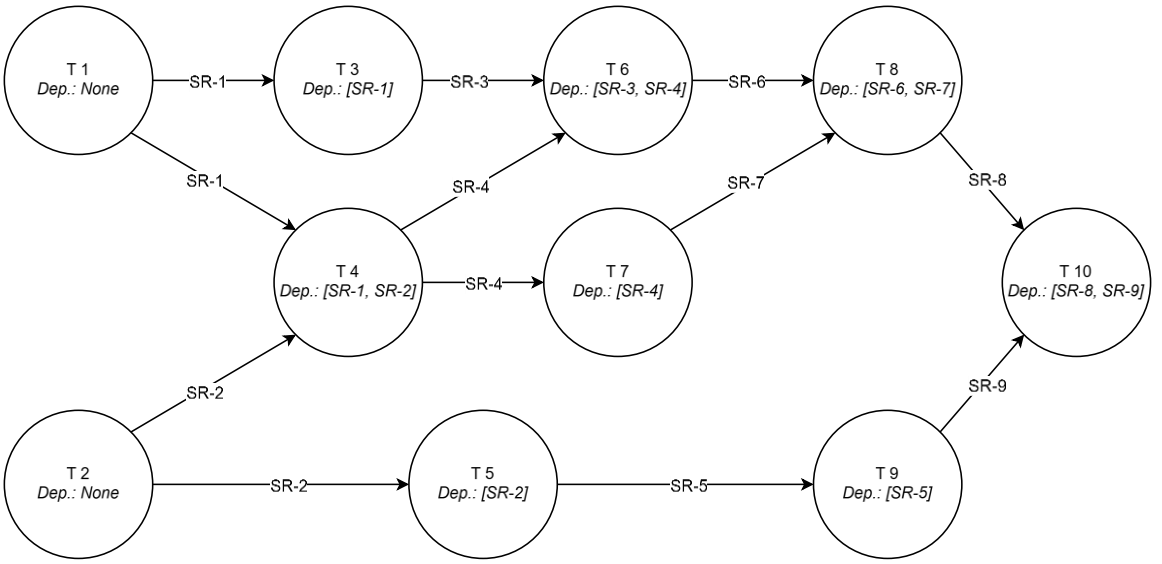

Fig. 3 Example of a task dependency graph

Operational Model Overview

After establishing the theoretical foundations of task dependency management, it is essential to present how the proposed model operates within a real system environment. Fig. 4 provides a conceptual overview of the interaction between the standard operating system scheduler, the CPU architecture, and the introduced Dependency-Aware Model (DAM). Initially, tasks are created either dynamically or predefined and placed into a general Ready Queue, representing tasks available for execution. Tasks that possess unresolved dependencies are intercepted by the DAM and temporarily redirected to a Hold Queue, preventing their premature dispatch to the operating system scheduler. The DAM continuously monitors the system by listening for completion signals emitted by previously executed tasks. Rather than performing continuous active polling, the DAM remains event-driven and reacts only when state changes occur in the system. Upon receiving such signals, the DAM reassesses the tasks in the Hold Queue. Tasks whose dependency conditions have been fully satisfied are transferred to the Ready Queue, making them eligible for scheduling on available CPU cores through the standard scheduler. This architecture ensures that CPU resources are allocated exclusively to tasks that are fully ready for execution. As a result, processor idle times caused by blocked or passively waiting tasks are

minimized, and the throughput and efficiency of the task execution pipeline are significantly improved.

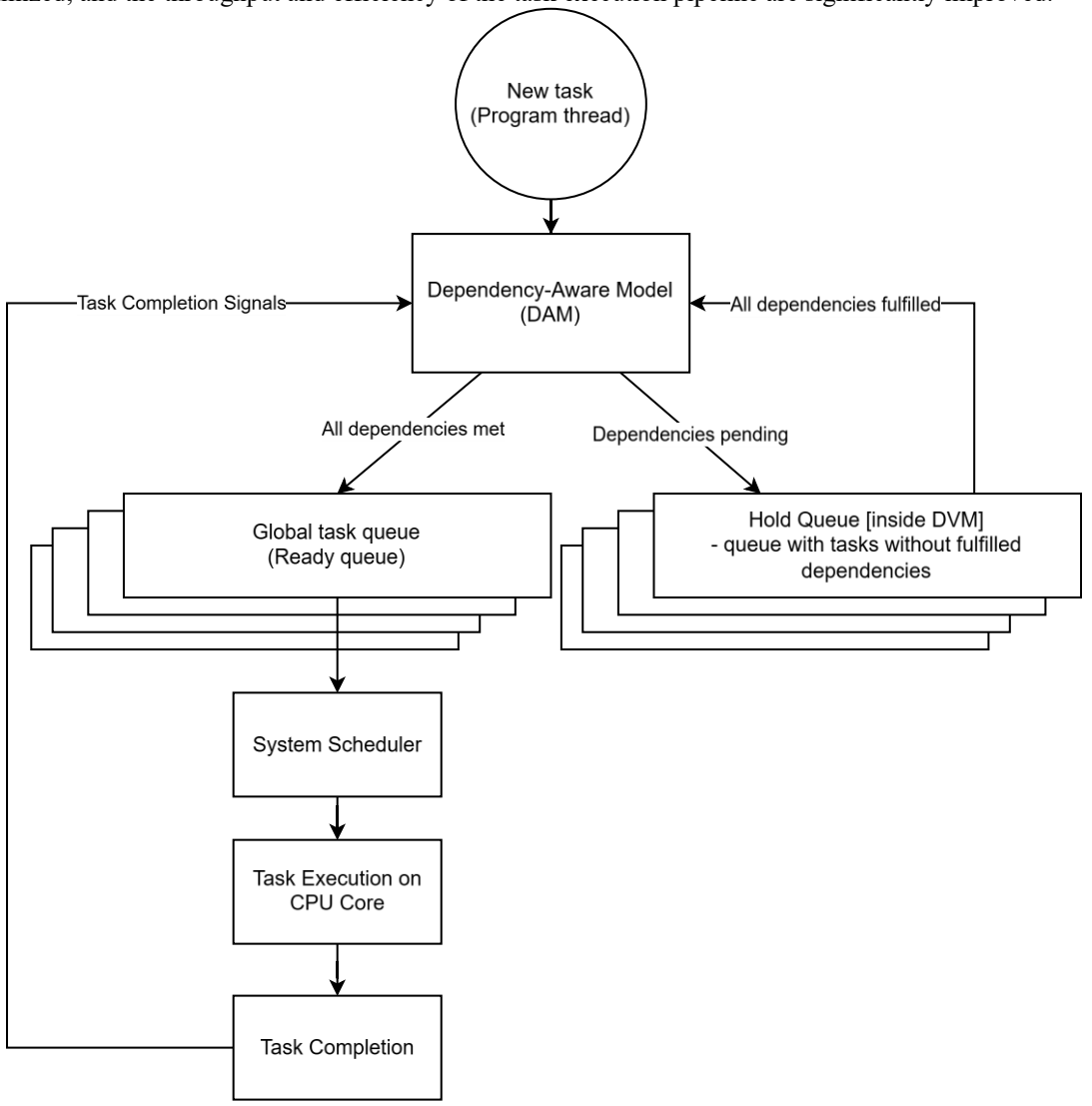

Fig. 4 Enhanced model of task scheduling utilizing the Dependency-Aware Model (DAM)

Discussion

The proposed model introduces an event-driven mechanism for enforcing task dependencies at runtime. This mechanism operates independently of the operating system's native scheduler and relies on the emission of task-specific completion signals. A task is dispatched only after all prerequisite signals have been received, ensuring that dependency constraints are satisfied prior to execution.

One of the key strengths of this approach is its simplicity. Unlike optimization-based schedulers that aim to compute an optimal task ordering, this model applies a rule-based filter that only requires knowledge of each task's dependency list. This is particularly relevant in systems with dynamic task generation, where dependency graphs may evolve during execution. While solving the full scheduling problem under dependency constraints is known to be NP-complete (Pinedo, 2016), this model avoids the complexity by enforcing correctness through readiness checking.

In addition, the model's architecture is inherently reactive. It monitors dependency signals without polling, responding only when a relevant event occurs. This design eliminates idle CPU usage from background loops and reduces resource contention. Tasks that are not yet ready remain inactive and are not introduced into the scheduler's domain, thereby minimizing unnecessary preemptions and context switches.

This strategy is especially advantageous in high-load scenarios involving dense dependency graphs. Instead of attempting to determine execution order in advance, the model allows execution to proceed naturally based on

the actual fulfillment of dependency conditions. As a result, system responsiveness and resource utilization improve, particularly in environments where interdependencies introduce scheduling bottlenecks.

Conclusions and Future Work

This paper presents a lightweight and scalable model for handling inter-task dependencies in multicore systems through the use of signal-based verification. By replacing identifier-based dependency checks with signal vectors, the model introduces an efficient mechanism that ensures tasks are executed only when all necessary conditions are met. The approach is scheduler-agnostic and does not require reengineering of core system components, allowing seamless integration with existing task management frameworks.

The key benefit of the model lies in its ability to prevent premature execution and idle processor occupancy by tasks that are not fully ready. This reduces resource contention and minimizes the overhead typically associated with waiting mechanisms. The model is designed to support dynamic task inflow, enabling it to handle a wide range of real-time and batch-processing scenarios where dependencies are complex, but well-structured.

Future work should focus on comprehensive simulation and performance evaluation in diverse workload conditions. In particular, it would be valuable to assess the model's efficiency gains across systems with varying core counts, dependency densities, and task arrival patterns. Additional exploration could investigate integration with priority-based scheduling or energy-aware execution policies. These extensions would further validate the model’s applicability and potential for improving execution efficiency in real-world computing environments.

Acknowledgment

The work was financed by the Military University of Technology in Warsaw, Poland as part of the project No. UGB 531-000023-W500-22.

References

- Burzyński, R., Nowicki, T. and Waszkowski, R. (2025), ‘Genetic Algorithm for Solving a Scheduling Problem,’ in Borowik, G., Chmaj, G. and Waszkowski, R. (eds), *Models and Methods for Systems Engineering*, Studies in Big Data, vol. 165, Cham: Springer. doi:10.1007/978-3-031-76440-0_14
- Cheng, X., Mao, Z., Wang, Y. and Wu, W. (2024), ‘Dependency-Aware CAV Task Scheduling via Diffusion-Based Reinforcement Learning,’ *arXiv preprint* arXiv:2411.18230.
- Linux Foundation., ‘Kernel Stacks (x86-64) – Kernel Documentation’. [Online], [Retrieved May 2025], Available: https://docs.kernel.org/next/x86/kernel-stacks.html
- Linux Foundation., ‘Linux Scheduler – Kernel Documentation’. [Online], [Retrieved May 2025], Available: https://docs.kernel.org/scheduler/index.html
- Linux Foundation. (2010), ‘Workqueue – The Linux Kernel Documentation’. [Online], [Retrieved May 2025], Available: https://docs.kernel.org/core-api/workqueue.html
- Microsoft. (2021), ‘About Processes and Threads’. [Online], [Retrieved May 2025], Available: https://learn.microsoft.com/en-us/windows/win32/procthread/about-processes-and-threads
- Microsoft. (2022), ‘Process and Thread Functions’. [Online], [Retrieved May 2025], Available: https://learn.microsoft.com/en-us/windows/win32/procthread/process-and-thread-functions
- Microsoft. (2021), ‘Scheduling’. [Online], [Retrieved May 2025], Available: https://learn.microsoft.com/en-us/windows/win32/procthread/scheduling
- Microsoft. (2025), ‘Windows Kernel-Mode Process and Thread Manager’. [Online], [Retrieved May 2025], Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/windows-kernel-mode-process-and-thread-manager
- Nowicki, T. (2012), ‘The task scheduling model and method in a specific decision-making problem,’ in Jastriebow, A., Kuźmińska-Sołśnia, B. and Raczyńska, M. (eds), *Computer Technologies in Science, Technology and Education*, Radom: Publishing House of Radom University of Technology, ISBN: 978-83-7351-499-7.
- Nowicki, T. and Waszkowski, R. (2017), ‘Productivity oriented cooperative approach to scheduling IT project tasks,’ in Saeed, K., Homeda, W. and Chaki, R. (eds), *Proceedings of the 16th IFIP International Conference on Computer Information Systems and Industrial Management (CISIM 2017)*, Lecture Notes in Computer Science, vol. 10244, Cham: Springer, pp. 254–365.
- Pinedo, M. L. (2016), *Scheduling: Theory, Algorithms, and Systems*, 5th ed., Springer.

- Shankar, S., Tamir, D. and Qasem, A. (2013), 'Towards an Operating System Based Framework for Energy-Efficient Scheduling of Parallel Workloads,' in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2013.
- Topcuoglu, H. and Hariri, S. (2002), 'Performance-Effective Task Scheduling Algorithms for Heterogeneous Systems,' *IEEE Transactions on Parallel and Distributed Systems*, 13 (3), pp. 260–274.
- Lyberis, S., Pratikakis, P., Mavroidis, I. and Nikolopoulos, D. S. (2016), 'Myrmics: Scalable, Dependency-aware Task Scheduling on Heterogeneous Manycores,' *arXiv preprint* arXiv:1606.04282.