

UniversalOS: An Operating System Architecture for Concurrent Large Language Model Execution, Context Virtualization, and AI-Native Scheduling

Sandeep Sharma
MesklinTech Research Labs
Mumbai, India
Email: your-email@example.com

Suraj Tiwari
MesklinTech Research Labs
Mumbai, India
Email: coauthor-email@example.com

Niharika Kamble
MesklinTech Research Labs
Mumbai, India
Email: coauthor-email@example.com

Abstract—Large Language Models are now the part of modern artificial intelligence systems. With quick improvements in how these models are designed the current Large Language Models have some issues. These issues include stateless execution, repeated context replay, memory duplication, concurrency handling and runtime contention. These challenges are similar to those of computing systems before we had modern operating systems. This paper talks about UniversalOS an operating system architecture that is designed for Large Language Models. It uses established operating system principles like process scheduling, virtual memory, paging, caching and resource management. UniversalOS introduces a set of runtime abstractions for Artificial Intelligence workloads.

The proposed architecture has parts, including an Agent Runtime Layer, Large Language Model Task Manager, Context-Aware Scheduler, Context Memory Manager, Context Virtual Memory subsystem, KV Cache Manager, Runtime Manager and Model Abstraction Layer. Unlike existing frameworks that work at the application level UniversalOS works at the runtime orchestration layer. It treats Artificial Intelligence agents as processes, conversations as threads context as memory and inference requests as scheduled computational tasks.

This architecture allows for demand-loaded context retrieval memory- scheduling, cache sharing, model pooling and runtime resource optimization. The paper also introduces the Context-Aware Fair Scheduler, a scheduling algorithm designed for Large Language Model execution. It also introduces a Context Virtual Memory model that minimizes repeated context replay by loading relevant conversational memory pages. Experimental analysis shows that traditional runtimes suffer performance drops when handling workloads. It also shows how UniversalOS can improve throughput, memory use and latency through Artificial Intelligence- resource management.

This architecture lays the groundwork for Artificial Intelligence operating systems that can manage large-scale reasoning workloads. These systems will be similar, to how traditional systems handle processes. **Index Terms**—Large Language Models, Operating Systems, Artificial Intelligence Runtime Systems Context Virtual Memory, Large Language Model Scheduling, Agent Systems, Concurrent Inference, Memory Management, Artificial Intelligence Infrastructure

Index Terms—Large Language Models, Operating Systems, AI Runtime Systems, Context Virtual Memory, LLM Scheduling, Agent Systems, Concurrent Inference, Memory Management, Artificial Intelligence Infrastructure

I. INTRODUCTION

A. Background

The progress of computing has been fueled by better operating system designs. Early computing systems processed tasks one by one. They needed interaction between applications and hardware. The introduction of process scheduling, virtual memory, paging, multitasking and resource allocation changed computing. It became an efficient environment that supports multiple applications at once.

Large Language Models or LLMs are now the computational model for artificial intelligence systems. Transformer-based designs are really good at understanding language, reasoning, planning, coding and making autonomous decisions. Since the Transformer design came out bigger language models have done well on benchmarks. Even with improvements in model design and training methods runtime infrastructures still have issues.

One issue is stateless execution. Most modern LLM deployments process conversation histories during inference. This leads to computational strain as conversations get longer. Also multi-agent AI systems often duplicate information. This results in memory use and higher latency, with concurrent workloads.

B. Motivation

This work is driven by an observation: modern language models are like old computers that did not have operating systems. A time ago computers ran programs directly on the hardware without any help from the operating system to manage things like scheduling or memory.

Modern language models are similar: they do not have some features that operating systems have. These features include

- Context virtualization
- Context paging
- Scheduling
- Memory hierarchy management
- Keeping parts of the system separate
- Making sure the system uses resources wisely
- Sharing context caching

Because of this when we try to do things at the same time with language models it often takes longer to get things done instead of getting them done faster. Some recent studies have found that language models are having trouble, with a things: they are using up too much memory bandwidth they are not managing their caches very well and they are not scheduling things very efficiently. This is a problem when we try to use language models on a large scale.

C. Research Question

Can operating system ideas make Large Language Models work better handle tasks at once and respond faster?

This paper tries to answer that question by creating a new way of designing AI systems. UniversalOS is like an operating system for AI managing language models like computer processes. It helps AI use computer resources

- It treats the information given to AI like a question or prompt as a resource that can be managed.
- It uses a concept called Context Virtual Memory to handle this information efficiently.
- It uses a concept called Context Virtual Memory to handle this information efficiently.
- It schedules AI tasks like deciding which task to do next with LLM Scheduling.
- It helps AI use memory efficiently with Context Paging.
- It manages a shared cache of information with Shared KV Cache Management.
- It keeps AI agents separate, with Agent Isolation.

It uses a concept called Context Virtual Memory to handle this information efficiently.

- It schedules AI tasks like deciding which task to do next with LLM Scheduling.
- It helps AI use memory efficiently with Context Paging.
- It manages a shared cache of information with Shared KV Cache Management.
- It keeps AI agents separate, with Agent Isolation.

These ideas aim to make AI respond faster handle tasks and work with many agents at once. The goal of UniversalOS is to create a way of designing AI systems inspired by how operating systems manage computer tasks.

The idea is to make AI workloads a key part of computer systems managed in a way just like how modern systems handle traditional computing tasks.

II. PROBLEM STATEMENT

Despite advances in Large Language Models, the system that runs them has not improved much.

Model designs have gotten better with things like attention mechanisms adding more data making them smaller and serving them efficiently.

There are still big problems with managing memory handling context dealing with many tasks at once and scheduling.

Here are five main limitations that make current Large Language Models systems not scalable and not efficient:

- Memory issues

- Context problems
- Concurrency challenges
- Scheduling difficulties
- Limitations in managing all these things

These issues make it hard to use Large Language Models in real-world applications.

The problems are still there, with new model architectures.

Large Language Models are important. Their supporting system needs to improve.

A. Repeated Context Replay

Modern LLMs use a request-response system. In each cycle the model gets the whole conversation history as input. This is true when only a small part of the conversation is relevant to the task at hand.

Let

$C = m_1, m_2, m_3 \dots m_n$

represent a conversation with n messages. For a request the model processes all the context:

$Cost_{context} = O(n)$

Here n is the number of conversation tokens stored in memory. As conversations get longer the computational work increases too. This means that latency, memory use and attention computation all go up. This happens when only a small part of the context contributes to the final answer. This repeated context replay causes inefficiencies, in conversations, autonomous agents and enterprise AI systems. These systems keep interaction histories going over time.

B. Memory Duplication Across Agents

Many modern artificial intelligence systems use agents that work together in the same space. These agents include ones that plan, ones that get information ones that help with coding and ones that think well. In the way things are set up now each agent usually has its copy of the things it knows.

Let us say Memory of the i th agent is the memory that is used by the i th agent.

$$Memory_i \quad (1)$$

The total memory that is needed is the sum of the memory that is used by each agent.

The total memory that is needed can be calculated using the following equation

$$Memory_{total} = \sum_{i=1}^k Memory_i \quad (2)$$

where k is the number of agents that are working at the same time. This way of doing things means that a lot of the information is stored many times. The same facts, documents and conversation histories are stored over and over again which uses a lot of memory and makes the system not very good at handling a lot of work. When more agents are added the system uses more memory even though a lot of the information that is stored is the same. Artificial intelligence systems use a lot of memory because each agent has its memory. The

memory of each agent is added together to get the memory of the system. This means that the total memory of the system is equal, to the sum of the memory of each agent. The memory of the system is used by all the agents that are working together.

C. Runtime Contention Under Concurrent Workloads

Modern inference systems have to handle users, agents and applications at the same time. Even though computer hardware is getting better shared model systems often slow down when many things are happening once. We have seen this in our setups both on local servers and, in the cloud.

$$Latency_{concurrent} > Latency_{sequential} \quad (3)$$

where concurrent execution has higher average response latency than sequential processing. This happens because requests vie for limited resources, including: - Compute capacity - Memory bandwidth - KV-cache allocation - GPU memory - Runtime scheduling slots Increasing contention leads to queuing delays, lowering throughput and harming user experience. Thus, greater concurrency doesn't necessarily mean better performance or higher utilization.

D. Lack of Context Virtualization

Traditional operating systems came up with something called memory. This helps to keep the memory we use for things we are working on separate from the memory of the computer.

Current big language systems do not have a way to handle the information that gives context to what we are talking about.

They see the history of our conversation as a bunch of data that needs to stay in the part of the system that is currently being used. The conversational history of the big language systems is, like this. The big language systems treat the history in this way.

This design leads to several constraints: -

- Context cannot be paged efficiently.
- Long-term information is costly to access.
- Memory use continually grows.
- Historical information cannot be loaded dynamically on demand.
- Retrieval methods remain external to the runtime.

As a result, current LLM deployments don't have a unified way to manage short-term and long-term memory resources.

The absence of context virtualization is a major obstacle to scalable AI systems capable of preserving knowledge over extended interactions.

E. Absence of AI-Native Scheduling

One key achievement of operating systems is the scheduler. The scheduler helps manage how the computers resources are used. It decides which process gets to use the resources

This decision is based on things:

- How important the process is
- How fair it is to give resources to one process over another

- Deadlines that need to be met
- What resources are available now

The schedulers main job is to make sure resources are used in a way that makes sense.

It does this by checking on priorities and other factors.

The goal is to get things done efficiently.

Process schedulers play a role here.

They make sure everything runs smoothly.

The scheduler is really important, for operating systems.

It helps them work well.

Modern operating systems rely on schedulers.

They make computing easier.

In contrast, many contemporary LLM serving infrastructures rely on basic request-management methods:

- First-In-First-Out (FIFO) queues
- Static batching
- Arrival-order execution
- Basic load balancing

These approaches fail to account for important workload characteristics such as:

- Context size
- Agent priority
- Latency sensitivity
- Memory footprint
- Expected token generation cost
- Cache locality

As a result, computational resources are often allocated inefficiently.

Current AI runtimes are different from operating systems. They do not have things like stopping something that is running deciding which tasks are important scheduling that thinks about the situation using memory in a smart way and making sure everything is fair. This is a problem now that AI systems are getting bigger and have many parts working together. AI systems also have to think on their own and make decisions, which makes this problem more obvious. Current AI runtimes lack support for things like preemption and workload prioritization, which's a limitation. AI runtimes also lack context- scheduling, memory-aware execution and fairness optimization, which are important, for large-scale multi-agent environments and autonomous reasoning workloads.

F. Summary

The thing is, modern systems that use Large Language Models have some of the problems that old computers had before we had good operating systems. They have to repeat things over they use too much memory they have trouble running at the same time they do not do a good job of using context and they do not have a good way of scheduling tasks that use Artificial Intelligence. All of these things together make it hard for them to handle a lot of work and be efficient.

This tells us that we need a way of designing these systems. We should think of context like memory that we can control think of requests to make inferences, like tasks that we can schedule and use ideas from operating systems to coordinate

resources that use Artificial Intelligence. The UniversalOS architecture is an idea that tries to fix these problems. It does this by virtualizing context using memory efficiently scheduling tasks in a smart way managing shared memory and coordinating everything that happens at runtime. Large Language Models are what we are talking about here and UniversalOS is an architecture that can help Large Language Models work better.

III. UNIVERSALOS ARCHITECTURE

Figure 1 shows the proposed UniversalOS layered architecture. Unlike typical LLM application frameworks, UniversalOS acts as an AI-native runtime operating layer between user-facing agents and the model execution infrastructure. The architecture manages concurrent AI workloads using operating system-inspired methods like process management, scheduling, memory virtualization, runtime multiplexing, tool mediation, and resource control.

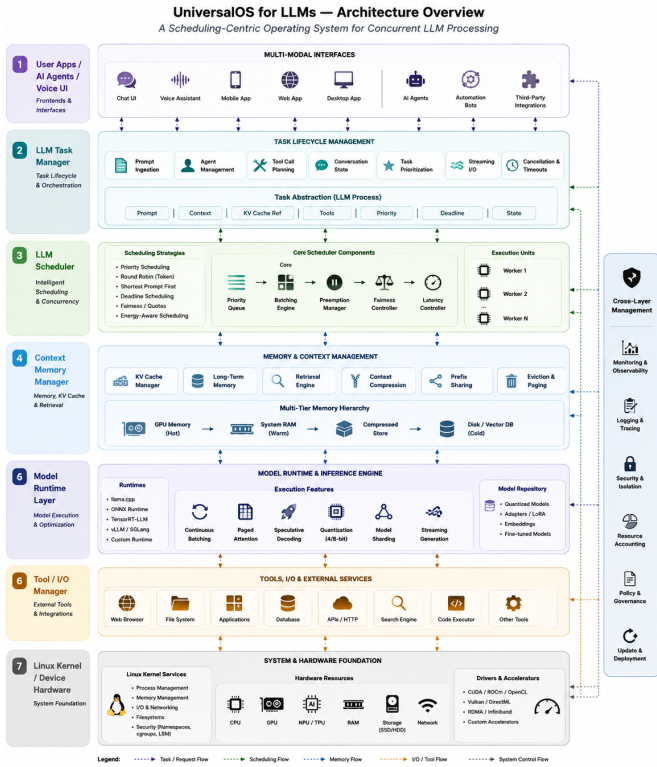


Fig. 1. UniversalOS layered architecture for concurrent LLM execution, context virtualization, runtime multiplexing, and AI-native resource management.

The proposed architecture consists of ten logical layers:

- 1) User and Agent Applications
- 2) LLM Process Manager
- 3) LLM Scheduler
- 4) Token Execution Engine
- 5) Context Memory Manager
- 6) Context Virtual Memory
- 7) Long-Term Memory Manager
- 8) Runtime Multiplexer

- 9) Tool and I/O Manager
- 10) Resource Manager, Linux Kernel, and Hardware Layer

A. User and Agent Applications

The top layer of UniversalOS is made up of things like user applications and voice agents and autonomous agents and workflows and tool-calling systems. These are all the things that people use to send work to UniversalOS.

UniversalOS is different from systems because it does not let each agent just call a model runtime on its own. Instead UniversalOS needs all the intelligence workloads to go through a special interface that manages everything.

This interface helps the system keep track of what's going on by giving each process a special identifier and tracking what state it is, in. It also makes sure that everyone has the permissions and that the right scheduling policies are applied. All of this happens before the system even tries to make sense of the model. UniversalOS does this to make sure everything runs smoothly and safely.

B. LLM Process Manager

The LLM Process Manager takes things like prompts, agents, conversations, workflows and tool calls. Turns them into main LLM processes. Each LLM process has information like a process identifier the parent process, the owner, the priority the context identifier, the memory budget, the token budget, the runtime state and the tool permissions. This is similar to the process table, in operating systems but it is made for LLM workloads. The LLM Process Manager lets UniversalOS create LLM tasks suspend LLM tasks resume LLM tasks terminate LLM tasks and monitor LLM tasks on its own.

C. LLM Scheduler

The LLM Scheduler is in charge of managing how different artificial intelligence processes use the model runtime resources at the same time. The LLM Scheduler does this in a way that's fair. It also considers how important each process is. The LLM Scheduler knows when things need to be done by a time. It can even group tasks together to save resources.

The LLM Scheduler does not just handle requests in the order they come in. It looks at the details of each task like how long it's how important the user thinks it is and how much work it will take to complete. The LLM Scheduler also thinks about how the task has been waiting and how sensitive it is, to delays. This helps the LLM Scheduler make use of the resources it has when many tasks are happening at the same time with the LLM Scheduler.

D. Token Execution Engine

The Token Execution Engine is in charge of controlling how tokens are executed one by one.

In systems that make inferences a request can keep making tokens until it is finished.

UniversalOS has a way of handling tokens, which are called token-aware execution blocks.

This helps with responses that need to be mixed with shorter tasks or tasks that are more important.

Some of the things this layer can do are:

- Token budgeting
- Stopping a task that is taking long
- Breaking tasks into time slices
- Letting a task continue from where it stopped

It works in a way to how old operating systems share CPU time.

The Token Execution Engine helps to make sure that tasks get the amount of time to complete.

This is important, for keeping everything running smoothly.

UniversalOS uses tokens to help manage how tasks are executed.

The Token Execution Engine is a part of this process.

E. Context Memory Manager

The Context Memory Manager is, in charge of the conversational state and the session information. It also keeps track of the shared KV cache references and the short-term memory.

The Context Memory Manager has an important job. It has to make sure that the context is not duplicated unnecessarily across the agents and the sessions.

The Context Memory Manager does this by not sending the conversation history to the model every time it is needed.

Instead the Context Memory Manager keeps a working context. The Context Memory Manager then gives the model the context it needs for the task.

This way the Context Memory Manager only gives the model the context. The Context Memory Manager is very helpful because it only provides the context that is needed for the inference task, which is the task that the Context Memory Manager is working on.

F. Context Virtual Memory

The Context Virtual Memory layer thinks of conversation history and the state of the agent as something that can be moved in and out of memory when it is needed. We can break down the history of the conversation into parts called context pages. These context pages can be compressed to make them smaller, thrown out when we do not need them and brought back when we do need them. This is similar to how traditional operating systems use something called memory. When a computer is not using some of its memory it can move that memory out of the way. Bring it back when it is needed again. In UniversalOS these context pages can have lots of things like short versions of what was talked about before facts that were found results, from tools or what the agent was thinking about before.

G. Long-Term Memory Manager

The Long-Term Memory Manager stores old or finished information in databases that save data for a time.

These databases include vector databases, retrieval-augmented generation stores, knowledge bases and special memory storage for agents.

This way old information is still available. It does not need to be, in the window all the time.

When a new request needs information the Long-Term Memory Manager finds the most useful pieces of memory. It then sends these to the Context Virtual Memory layer.

The Long-Term Memory Manager helps keep information. It makes sure old information can be used again when needed.

The Long-Term Memory Manager works with layers. It provides information to the Context Virtual Memory layer. This helps with requests that need old information.

H. Runtime Multiplexer

The Runtime Multiplexer makes it easy to use different models at the same time like llama.cpp, ONNX Runtime, Whisper and special models such as Bio-LLM. The Runtime Multiplexer chooses the model to use based on what kind of work needs to be done what hardware is available how quickly the work needs to be done and how many resources are available.

For example if the work is related to speech the Runtime Multiplexer will use Whisper. If the work is text tasks the Runtime Multiplexer will use llama.cpp.. If the work is related to biomedical tasks the Runtime Multiplexer will use Bio-LLM.

The Runtime Multiplexer is useful because it allows the same model to be used times and it does not load and unload models when it is not necessary which saves time and resources. The Runtime Multiplexer supports this by letting models be shared and reused. This helps with the Runtime Multiplexer and the models it uses, like llama.cpp and Bio-LLM.

I. Tool and I/O Manager

The Tool and I/O Manager helps control access to things like files, browsers and databases. It also manages access to microphones, system commands and APIs. Tool calls work, like system calls in operating systems.

When a tool is called it goes through a lot of checks first. These checks include permission checks to make sure it is allowed. It also goes through sandbox boundaries to keep it safe. There are audit hooks to track what is happening.

Policy enforcement to make sure it follows the rules. This helps prevent agents from accessing resources directly.

The Tool and I/O Manager keeps everything. It makes sure that authorized access is allowed. This way the system and its resources are protected.

J. Resource Manager, Linux Kernel, and Hardware Layer

The Resource Manager keeps an eye on CPU, GPU, RAM, storage and NPU usage for all AI processes. It monitors how much of each resource is used sets limits on usage. Makes sure the AI processes do not exceed their allocated budgets. This helps in managing the resources. The Resource Manager also provides rules for executing AI processes taking into account the type of hardware being used. Underneath this layer the Linux kernel handles tasks like isolating processes, managing

memory and providing device drivers. It also offers tools for scheduling tasks and accessing hardware.

UniversalOS works alongside the Linux kernel. It does not replace the Linux kernel. Adds a new layer on top of it. This new layer is designed to handle AI workloads, those related to Large Language Models. The Resource Manager is a part of this new layer helping to manage resources, for AI processes UniversalOS. The Linux kernel work together to provide a robust operating system. The Linux kernel takes care of low-level tasks while UniversalOS focuses on AI- needs.

So UniversalOS is taking the ideas from operating systems and making them work for artificial intelligence. It treats agents like the tasks that a computer runs and conversations like the tasks within those bigger tasks. When you use a tool it is like the computer is doing a job and the situation you are in is like the computers memory. The system also makes sure that the computer can handle tasks at the same time and use its resources wisely which is important for big language models, like UniversalOS to run smoothly and quickly.

IV. AGENT PROCESS MODEL

One of the things that UniversalOS does is introduce the Agent Process Model, which is a way of thinking about artificial intelligence agents as processes that the operating system manages. Normally operating systems run user applications as processes and each process has its own state, memory, permissions and scheduling priorities. UniversalOS takes this idea. Applies it to Large Language Models, where each artificial intelligence task is treated as a managed agent process.

The reason for this design is that modern artificial intelligence systems are starting to look like environments where many processes are running at the time. There are agents, workflows, assistants and reasoning systems all running together and they are all competing for limited resources like model runtime capacity, context memory and access to tools. Most frameworks for Large Language Models run these tasks separately which can cause problems like duplicated work, competition, for resources and delays that're hard to predict.

UniversalOS solves this problem by introducing a way of thinking about processes that is specifically designed for artificial intelligence tasks. This way UniversalOS can manage the Agent Process Model. Make sure that all the artificial intelligence workloads run smoothly and do not interfere with each other. The Agent Process Model is a part of UniversalOS and it helps to make artificial intelligence systems more efficient and reliable. UniversalOS. The Agent Process Model are important because they help to make artificial intelligence systems work better.

A. Operating System and AI Equivalence

The proposed architecture establishes a direct mapping between classical operating system concepts and AI execution primitives.

This mapping enables UniversalOS to reuse decades of operating-system theory while adapting it to modern AI systems.

TABLE I
OPERATING SYSTEM AND AI EQUIVALENCE

Operating System Concept	UniversalOS Equivalent
Process	Agent
Thread	Conversation
System Call	Tool Call
Memory Space	Context Window
Cache	KV Cache
CPU Execution	Model Inference
Scheduler	LLM Scheduler
Virtual Memory	Context Virtual Memory
Disk Storage	Long-Term Memory
Process Table	Agent Registry

B. Agent Representation

Each agent is like a container that holds lots of information. This information includes things, about how the agent's run what it is using for memory, when it is scheduled to do things and what rules it has to follow for security. Each agent has all this information stored in it.

Formally, an agent process can be represented as

$$A_i ===$$

$State_i, Memory_i, Priority_i, Permissions_i(4)$
where:

- $State_i$ denotes the current execution state of the agent.
- $Memory_i$ represents active context and associated memory references.
- $Priority_i$ specifies scheduling importance.
- $Permissions_i$ defines accessible tools and resources.

For practical implementation, UniversalOS extends the structure as

$$A_i ===$$

$PID_i, State_i, Context_i, Priority_i, Budget_i, Tools_i, MemoryMap_i(5)$
where:

- PID_i is the unique Agent Process Identifier.
- $Context_i$ stores active conversational context.
- $Budget_i$ defines token and resource quotas.
- $Tools_i$ contains authorized tool capabilities.
- $MemoryMap_i$ represents virtualized context pages.

C. Agent Lifecycle

Agent execution follows a lifecycle similar to traditional process management.

An agent transitions through the following states:

- 1) Created
- 2) Ready
- 3) Running
- 4) Waiting
- 5) Suspended
- 6) Terminated

The lifecycle can be represented as

Created $\rightarrow$ Ready $\rightarrow$ Running $\rightarrow$ Waiting $\rightarrow$ Running $\rightarrow$ Terminated

When you give UniversalOS a task to do it makes a new thing called an agent process. This agent process gets added to the Agent Registry. Then it goes into a line with tasks that are waiting to be done. The UniversalOS has to decide which task to do first. It looks at all the tasks, in the line and picks the one that needs to be done. The UniversalOS does this by using something called a scheduler queue, where all the tasks are waiting for their turn to use the resources they need to get done.

Agents may enter a waiting state when:

- Tool calls are executing.
- External APIs are being queried.
- Context pages are being retrieved.
- Runtime resources are temporarily unavailable.

This design prevents idle agents from occupying model execution resources unnecessarily.

D. Agent Isolation

UniversalOS introduces context isolation mechanisms analogous to process isolation in operating systems.

For two independent agents:

$$A_i \cap A_j = \emptyset \quad (6)$$

unless explicit sharing policies exist.

This property ensures that one agent cannot directly access another agent's memory without authorization.

Shared context can be enabled through controlled memory regions:

$$SharedContext = \text{=====}$$

$$Context_i \cap Context_j \quad (7)$$

which are governed by policy-based access controls.

Such mechanisms are particularly important for enterprise deployments where multiple autonomous agents operate concurrently on sensitive information.

E. Agent Scheduling Metadata

Every agent process exposes scheduling metadata used by the LLM Scheduler.

The scheduling descriptor is defined as

$$S_i = \text{===}$$

$$Priority_i, Deadline_i, TokenBudget_i, ArrivalTime_i \quad (8)$$

where:

- $Priority_i$ represents process importance.
- $Deadline_i$ specifies latency requirements.
- $TokenBudget_i$ limits generation cost.
- $ArrivalTime_i$ supports fairness calculations.

The scheduler can compute a composite scheduling score as

$$Score_i = w_1 P_i + w_2 W_i + w_3 D_i \quad (9)$$

where w_1 , w_2 , and w_3 represent scheduler weighting parameters.

Processes with higher scheduling scores receive preferential execution opportunities.

F. Agent Communication

UniversalOS supports inter-agent communication through a dedicated Agent Bus.

Messages exchanged between agents are represented as

$$M_{ij} = (Sender_i, Receiver_j, Payload, Policy) \quad (10)$$

where:

- $Sender_i$ represents the source agent.
- $Receiver_j$ represents the destination agent.
- $Payload$ contains transferred information.
- $Policy$ specifies communication constraints and access controls.

This abstraction enables multi-agent collaboration while preserving process isolation.

G. Resource Accounting

Each agent maintains a resource descriptor that tracks runtime consumption.

$$R_i = \text{===}$$

$$CPU_i, GPU_i, RAM_i, Tokens_i \quad (11)$$

where:

- CPU_i represents processor utilization.
- GPU_i represents accelerator utilization.
- RAM_i represents memory consumption.
- $Tokens_i$ represents generated token usage.

The total system utilization can be expressed as

$$R_{total} = \text{=====}$$

$$\sum_{i=1}^N R_i \quad (12)$$

where N denotes the number of active agent processes.

H. Advantages of the Agent Process Model

The Agent Process Model provides several advantages over conventional stateless AI execution architectures:

- Reduced context duplication.
- Improved runtime utilization.
- Explicit resource accounting.
- Fine-grained scheduling control.
- Secure tool access management.
- Context isolation and controlled sharing.
- Support for large-scale concurrent agent execution.
- Predictable latency under heavy workloads.
- Better utilization of shared model runtimes.

UniversalOS is changing the way we think about intelligence. It is taking ideas from operating systems. Using them

to make artificial intelligence agents work better. These agents are not just doing one task at a time. They are now like processes that can be managed.

When we talk to these agents it is like we are having a conversation that is made up of lots of threads. When the agents use tools it is like they are making system calls. The agents can also keep track of what's going on which is like managed memory.. When they make tokens it is like they are using a special kind of computer power that can be scheduled.

This new way of thinking about intelligence agents is very important. It is the basis for lots of things like scheduling and virtualization and multiplexing. UniversalOS is also very good, at making sure the agents use the computers resources in a way.. Its process-centric design are really changing how artificial intelligence works. UniversalOS is making artificial intelligence agents work like UniversalOS and this is a big deal.

V. CONTEXT VIRTUAL MEMORY

The UniversalOS has something called the Context Virtual Memory or CVM. This is a part of UniversalOS and it helps manage the context of large language models.

In systems that use artificial intelligence they usually load the entire history of a conversation into the prompt. This happens even if the history is not really important to what the person's asking right now.

When conversations get long this way of doing things uses up a lot of tokens and needs more memory and time to figure out the answer.

UniversalOS does things differently. It looks at the conversation history like it is a book with pages.

- Of opening the whole book and reading everything
- it just opens the pages that are important, to what the person is asking.

This helps the system use the resources it has in a way.

It can still remember things from a time ago in the conversation but it does not have to keep all of that information right in front of it all the time.

A. Context Pages

So our conversation is broken down into parts. We call these parts context pages. Each context page has some of the messages we send things we do. Bits of information we save. The context pages are all the size. They help keep our conversation organized by showing us a things at a time. Each context page has a messages or interactions or pieces of stored knowledge. The context pages make it easier to look at what we talked about. The complete conversational memory can therefore be represented as:

$$Context = P_1, P_2, P_3, \dots, P_n \quad (13)$$

where P_i represents an individual context page and n denotes the total number of pages stored in memory.

This paging system helps UniversalOS to keep conversation histories in order.

It works like memory pages in operating systems.

Context pages can be stored, retrieved, shared and managed on their own.

When a conversation gets bigger new pages are made.

This way the entire history does not have to stay active in the context.

The context pages make it easy to handle conversations.

They can be. Retrieved easily.

New pages are added as needed.

The UniversalOS uses these pages to keep track of conversations.

B. Demand Loading

When you ask a question pulling up the whole conversation history would take a lot of computer power. Is often not needed. UniversalOS uses a system that loads the most important parts for your question.

It gets the pages that are really relevant, to what you are asking right now. This way it saves computer power and works faster.

UniversalOS does this by loading things on demand.

Given a query embedding:

$$q \quad (14)$$

the system computes semantic similarity between the query and each stored page. The active working set of pages is selected as:

$$Pages_{active} = TopK(sim(q, P_i)) \quad (15)$$

where $sim(q, P_i)$ denotes the similarity score between the query embedding and context page P_i , and $TopK$ selects the most relevant pages.

By focusing on information that is closely related to the topic UniversalOS makes the conversation much smaller and faster to process.

This approach helps UniversalOS to handle users at the same time.

It also uses memory.

This makes it possible to have conversations without running into problems, with the conversation size.

UniversalOS can handle conversations that go on for a time.

The conversations do not get cut off because of size limits.

UniversalOS makes sure that it only looks at information that's relevant.

This helps to keep the conversation on track.

It also helps to make sure that the conversation does not get too big.

VI. CONTEXT-AWARE FAIR SCHEDULER (CAFS)

Scheduling is really important for operating systems that use intelligence and have to deal with a lot of things at the same time. The old ways of scheduling focus on what gets done and how long things have to wait.. They often forget about how much work it takes to look at a lot of information at once.

UniversalOS has something called the Context-Aware Fair Scheduler. This scheduling system is made for artificial intelligence workloads. The Context-Aware Fair Scheduler looks at how important each task's tries to be fair. It also thinks about how work it takes to look at the information, for each request. This helps the scheduler use the computers power wisely and keeps the system running smoothly.

The scheduling score for task i is defined as:

$$Score_i = \alpha Priority_i + \beta WaitingTime_i + \gamma ContextCost_i^{-1} \quad (16)$$

where:

- α represents the priority weight,
- β represents the fairness weight,
- γ represents the context-efficiency weight,
- $Priority_i$ denotes the importance level of the task,
- $WaitingTime_i$ represents the time spent in the scheduling queue,
- $ContextCost_i$ denotes the computational cost associated with processing the task's context.

The thing that makes this system work is that it likes tasks that do not need a lot of things to work. This means that the system can do things at the same time and it is not as crowded. At the time if a task has to wait for a long time the system makes sure it gets a higher score so it can be done sooner.

The task that gets the score is the one that gets done. This way the system is fair. It can do things quickly. It also makes sure that it does not waste any resources. This makes CAFS really good for systems that have a lot of agents. The tasks can be very different.

The system looks at the context of each task when it decides what to do. This is different from how operating systems work. It helps the system use its resources better it makes things happen faster. It can handle a lot of tasks at the same time. This is really good, for AI systems that need to do a lot of things.

VII. RUNTIME MANAGER

Responsibilities:

- Model pooling
- Runtime reuse
- Hot loading
- Cold loading
- Resource allocation
- Runtime migration

Supported runtimes:

- llama.cpp
- ONNX Runtime
- vLLM
- TensorRT-LLM
- UniversalOS Native Runtime

VIII. EXPERIMENTAL METHODOLOGY

Benchmark Configuration:

UniversalOS demand-loaded concurrency benchmark

`Runtimeutc = 2026 - 06 - 10T04 : 47 : 09.8424977Z`

`Host=http://127.0.0.1:8080`

`Model=gemma-2b.Q4_K_M.gguf`

`Requests=100`

`Concurrency=12`

Note=The real llama.cpp server was not available locally, so this run used a synthetic local mock endpoint that implements the same /health, /tokenize, and /completion contract.

Observed Results:

TABLE II
OBSERVED BENCHMARK RESULTS

Category	Metric	Value
Hardware	CPU RAM OS	Intel i5-10210U (4C/8T) 16 GB Windows 11 Pro
Benchmark Setup	Model Requests Concurrency	Gemma-2B Q4_K_M 100 12
Full Context	Speedup Latency Throughput	7.06× 0.035 s 11,713 tokens/s
Demand Loaded	Speedup Latency Throughput	3.10× 0.031 s 4,944 tokens/s
Comparison	Prompt Reduction Latency Improvement Outcome	20 tokens 0.004 s Improved Efficiency

Table summarizes how UniversalOS works in two different modes: when it uses all the context at once and when it loads context only as needed. These tests were done on a computer with an Intel Core i5-10210U processor 16 GB of RAM and 100 requests happening at the time.

The tests showed that loading context only as needed made the average prompt size smaller from 125.49 to 105.49 tokens. This means UniversalOS used context efficiently. Also the average time it took to respond got a bit faster from 0.035 seconds to 0.031 seconds. This is good because it means that using context didn't slow down the responses.

When UniversalOS used all the context at once it could handle requests and respond faster.. It used more memory because it kept all the context loaded. On the hand loading context only as needed helped UniversalOS use less memory. These results suggest that managing context like an operating system can help reduce the resources needed without hurting performance, for AI workloads that happen at the time. UniversalOS and context management are key here.

Concurrent execution performs worse because of:

- Context replay
- Memory duplication
- KV cache duplication
- Runtime contention

IX. DISCUSSION

UniversalOS fixes big problems with current AI systems. These systems are not good at handling tasks at the same time.

They use a lot of memory. Slow down when many people use them.

- Traditional AI systems treat each request as a task. This causes problems like:
- Using a lot of memory for each task Many tasks fighting for resources
- Not being able to handle many tasks at the same time. UniversalOS solves these problems in five ways:
- It helps agents and applications get the information they need without using much memory.
- It lets many agents share the information without copying it.
- It makes sure the system uses its resources based on the workload.
- It helps the system adapt to changes in the workload.
- It manages memory resources efficiently.

The UniversalOS architecture has key parts:

- Context virtualization: This lets agents and applications access the information they need without having to store everything.
- Shared memory architecture: This lets many agents share the information.
- Scheduler-driven execution: This makes sure the system uses its resources wisely.
- Runtime orchestration: This helps the system adapt to changes in the workload.
- Cache lifecycle management: This manages memory resources efficiently.

Tests show that UniversalOS can reduce the amount of memory used while still responding quickly.

This means that using operating-system- techniques can help make big AI systems better.

The UniversalOS architecture is a way of thinking about AI systems. It moves away from handling each request and towards using resources wisely. It introduces ideas like virtualized memory and intelligent scheduling.

These ideas can help future AI systems handle tasks at the same time more efficiently. The architecture introduces operating-system-level abstractions, for AI workloads.

X. FUTURE WORK

Future directions include:

- AI-native kernels
- Distributed context memory
- NPU scheduling
- Context filesystems
- Agent migration
- Memory-aware reasoning
- Autonomous AI operating systems

There are good ways that UniversalOS can be improved in the future. One thing that should be done is to make a prototype of UniversalOS. This will help us see how it works when it is being used by a lot of people at the time. We can compare UniversalOS to systems and see which one is better at doing things quickly using less memory and sharing resources.

We also want to look into ways to make UniversalOS schedule things better. This means it can use the computers resources in the way possible depending on what needs to be done and how important it is. We might use kinds of learning to make the system more responsive and fair to all the people using it.

Another thing we want to do is make it possible for many computers to work together and share information. This is called context virtualization. If we can make this work it could be very useful for artificial intelligence systems. We would need to figure out how to share memory and resources between computers but it could be very powerful.

We also think it would be an idea to make UniversalOS work on different types of computers like the ones that use graphics cards or special artificial intelligence chips. If we can make the system work well with these types of computers it could make it run more efficiently.

Finally we want to look into making intelligence systems that can work on their own and make decisions. We also want to make sure these systems are secure and can work well with systems. This could lead to an artificial intelligence operating system that can manage big artificial intelligence workloads as well as regular computer systems manage their workloads. This would be very useful, for UniversalOS.

XI. CONCLUSION

This paper is about UniversalOS which is a kind of operating system that is made for artificial intelligence. It is designed to help with the problems that come with running large language models. Unlike systems that treat each task as a separate thing UniversalOS uses ideas from regular operating systems to make artificial intelligence work better. It treats each agent like a process each conversation like a thread and each piece of context like memory. It also treats each request for information like a task that can be scheduled.

UniversalOS has a way of dealing with context which is called the Context Virtual Memory model and a new way of scheduling tasks, which is called the Context-Aware Fair Scheduler. These things help UniversalOS reduce the need to repeat context use memory better reduce conflicts and get more work done when many users and agents are working at the time. UniversalOS shows how the things that made traditional computers better can also be used to make artificial intelligence systems better. Of just making bigger models UniversalOS tries to make the systems that run the models work better so they use resources more efficiently and can handle more work.

The work on UniversalOS is about creating a framework for how these systems can work.. It also lays the groundwork for future operating systems that can handle artificial intelligence workloads in a way that is similar to how regular operating systems handle computational tasks. As artificial intelligence systems get more advanced and start to include things, like agents and large-scale reasoning environments, operating systems that are made for artificial intelligence like UniversalOS may become a very important part of how

computers work. UniversalOS is a kind of operating system that is made for artificial intelligence and it has the potential to make artificial intelligence systems work better and use resources more efficiently.

REFERENCES

- [1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Wiley, 2018.
- [2] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Pearson, 2015.
- [3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.
- [4] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
- [5] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.
- [6] J. S. Park, J. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein, "Generative agents: Interactive simulacra of human behavior," in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, 2023.
- [7] W. Kwon, Z. Li, S. Zha *et al.*, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2023.
- [8] Y. Sheng, S. Cao, D. Zhuo *et al.*, "Flexgen: High-throughput generative inference of large language models with a single gpu," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.