vLLM Scheduler

High-Level Overview

The **scheduler** is responsible for managing and optimizing the execution of different sequence groups (requests). It handles:

1. **Prefill Scheduling** (processing input prompts).
2. **Decoding Scheduling** (generating output tokens).
3. **Preempting Sequences** (swapping sequences to free GPU memory).
4. **Chunked Prefill** (dividing large inputs into smaller chunks for efficiency).
5. **Lookahead Decoding** (optimizing token generation by predicting multiple tokens ahead).

Key Components and Concepts

1. Preemption Modes

Defined in `PreemptionMode`, preemption refers to **removing a sequence from the GPU to free memory**:

- **SWAP** → Moves the sequence's memory to the CPU (slower but retains computation).
- **RECOMPUTE** → Discards the sequence and reprocesses it later as a new prompt.

2. Scheduling Budget (`SchedulingBudget`)

Tracks available computation slots in terms of:

- `token_budget`: Maximum allowed tokens in a batch.
- `max_num_seqs`: Maximum allowed sequence groups.
- `_num_batched_tokens`: Count of actual new tokens.
- `_num_cached_tokens`: Tokens retrieved from cache.
- `_num_curr_seqs`: Number of sequences being processed.

Ensures that scheduling does not exceed memory constraints.

3. Scheduled Sequence Groups

`ScheduledSequenceGroup`: A sequence group that is scheduled for processing.

`SchedulerOutputs`: Holds the results of a scheduling step, including:

- Sequences to be processed (`scheduled_seq_groups`).
- Number of prefill requests (`num_prefill_groups`).
- Blocks to swap between GPU/CPU (`blocks_to_swap_in` and `blocks_to_swap_out`).
- Number of ignored sequences (due to constraints).
- **Lookahead slots** (for speculative decoding).

Core Scheduling Functions

4. `schedule()`

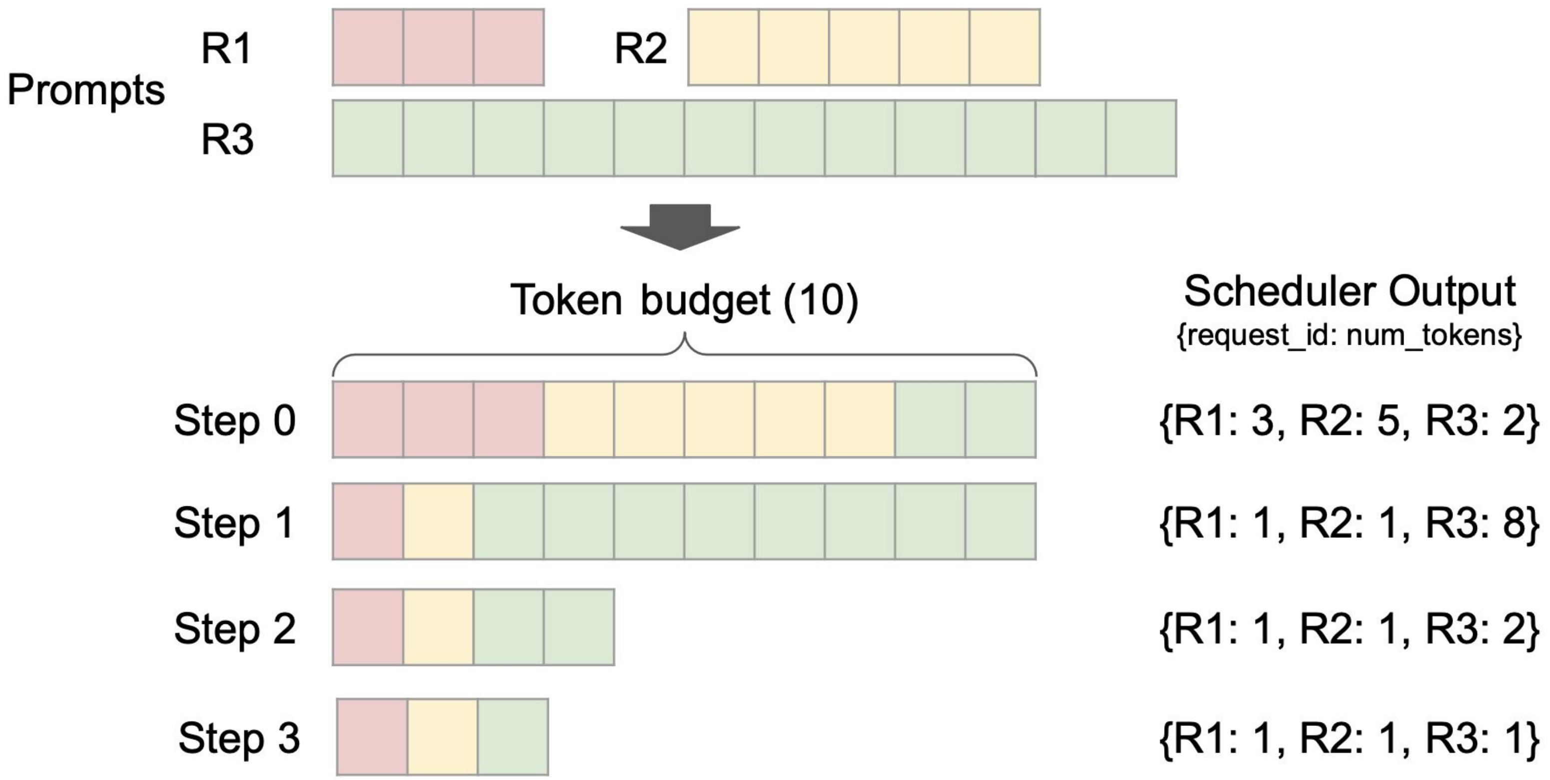
This is the main function that:

1. Runs the `_schedule()` method.
2. Returns the metadata for execution.
3. Handles caching optimizations.

5. `_schedule_default()`

- **First tries to schedule prefill requests.**
- If no prefills are available, it moves on to decodes.
- Handles **swapping sequences in/out** if memory is constrained.

Why first try to schedule prefill requests?



6. `_schedule_chunked_prefill()`

Allows multiple prefill requests to run **concurrently**.

Optimizes GPU utilization by interleaving **decoding & prefill**.

Ensures small requests are **not blocked** by large ones.

7. `_schedule_running()`

Manages sequences in the **running queue**.

Ensures sequences fit within **memory limits**.

Handles **preempting low-priority sequences** if necessary.

8. `_schedule_swapped()`

Brings **previously swapped-out** sequences back into the GPU.

Prioritizes sequences that can **fully fit within the current budget**.

9. `_schedule_prefills()`

Handles **waiting sequences** that need to be prefixed.

Checks if **memory can accommodate** the request.

Ensures **long sequences do not block smaller ones**.

Memory Management & Preemption

10. `_preempt()`

Decides whether to **swap out** or **recompute** a sequence.

Uses **priority scheduling** to preempt low-priority sequences.

Calls `_preempt_by_swap()` or `_preempt_by_recompute()`.

11. Swapping Functions

`_swap_in(seq_group, blocks_to_swap_in)` → Moves sequence back into GPU.

`_swap_out(seq_group, blocks_to_swap_out)` → Moves sequence to CPU.

Performance Optimizations

12. Caching & Lookahead Slots

- Uses `block_manager` to track **cached tokens**.
- Implements **speculative decoding** by allocating *lookahead slots*.
- Ensures optimal batch sizes for high **GPU efficiency**.

13. Chunking Prefills

- Large prefill sequences are **divided into smaller parts**.
- Improves responsiveness for short queries.

Detailed Overview of the Provided Functions in vLLM Scheduler

`abort_seq_group` - Handling Aborted Sequences

This function **removes a sequence group** from scheduling, **frees resources**, and ensures that it does not continue execution.

◆ What It Does

1. **Identifies which sequence(s) should be aborted**
    - The function first determines whether `request_id` refers to **one or multiple requests**.
    - It ensures that it correctly handles **parallel sampling cases** (where a request has multiple generated variations).
2. **Finds the sequence group(s) in the scheduler queues**
    - Checks across the **waiting**, **running**, and **swapped** queues.
    - Collects **all sequence groups** that match the request ID.

`abort_seq_group` - Handling Aborted Sequences

🔹 What It Does

3. Removes the identified sequence groups from the scheduler

- If a sequence group is found, it is **removed from the respective queue**.
- The request ID is **added to `_finished_requests_ids`**, ensuring that any associated states (such as memory allocations) are cleaned up.

4. Marks unfinished sequences as aborted

- If a sequence is still **in progress**, its status is set to **`FINISHED_ABORTED`**.
- This ensures that the scheduler **does not attempt to process it anymore**.

5. Frees memory associated with the sequence

- Calls `self.free_seq(seq)` to **release memory blocks** used by the sequence.

`abort_seq_group` - Handling Aborted Sequences

🔹 Why It's Needed

- Ensures **graceful termination** of a request that needs to be canceled (e.g., a user stops a request).
- Prevents **zombie processes** (orphaned sequences still consuming resources).
- Handles **parallel sequence generation** correctly.

`_schedule_running` - Scheduling Active Sequences

This function **manages and schedules sequences that are currently in the running state** (either **decoding** or **chunked prefill**).

🔹 What It Does

1. **Prepares a scheduling output object**
    - Retrieves a **cached** `SchedulerRunningOutputs` **object** (avoids unnecessary object creation).
2. **Determines the number of lookahead slots**
    - **Lookahead decoding** allows speculative execution to improve efficiency.

`_schedule_running` - Scheduling Active Sequences

Iterates over the running queue and schedules sequences

- If a sequence is using **cached tokens**, it ignores them.
- It checks if there are **available slots** for new tokens.
- If a sequence cannot be scheduled due to **budget constraints**, it is either:
  - **Preempted** (moved to waiting)
  - **Swapped out** (moved to CPU memory)

Handles preemption if no available slots exist

- Preemption removes **low-priority sequences** if memory is full.

`_schedule_running` - Scheduling Active Sequences

Allocates slots and schedules sequences

- **Decoding requests** are **assigned one token per step**.
- **Prefill requests** may receive a **larger chunk of tokens.**

Updates the scheduling budget

- **Updates the number of tokens and sequences** accounted for.
- Ensures **LoRA requests** (Low-Rank Adaptation for fine-tuning) are **correctly batched**.

`_schedule_running` - Scheduling Active Sequences

🔹 Why It's Needed

- Ensures **efficient allocation of GPU resources**.
- Prevents **starvation of decoding tasks** when prefill sequences are present.
- Allows **preemption of low-priority sequences** to avoid memory exhaustion.

`_schedule_swapped` - Managing Swapped-Out Sequences

This function **handles sequences that were previously swapped out** (moved to CPU memory) and **tries to bring them back** into active processing.

🔹 What It Does

1. **Prepares lists to track actions taken**
   - Tracks:
     - **Sequences to swap in**
     - **Sequences that can't be processed due to resource limits**
     - **Copying of memory blocks**
2. **Iterates through swapped-out sequences**
   - **Checks if a sequence can be swapped in** (i.e., whether there is enough memory).
   - Determines if the sequence should be:
     - **Brought back to active processing**.
     - **Ignored due to resource constraints**.

`_schedule_swapped` - Managing Swapped-Out Sequences

Handles cases where a sequence cannot be swapped in

- If a sequence **cannot be processed due to insufficient memory**, it is **marked as infeasible**.
- If it **can be processed but LoRA constraints apply**, it is **moved to a temporary list** for later retry.

Allocates memory and resumes processing

- If a sequence is successfully **swapped in**, it:
    - **Allocates the necessary memory blocks**.
    - **Moves it back into the scheduler's active queue**.

Updates scheduling constraints

- **Ensures memory allocations remain within the defined budget**.
- **Tracks LoRA requests** to ensure they do not exceed limits.

`_schedule_swapped` - Managing Swapped-Out Sequences

🔹 Why It's Needed

- Ensures that **swapped-out sequences can resume** when memory becomes available.
- Prevents **infinite swapping loops** by **tracking infeasible sequences**.
- Balances **compute-bound and memory-bound scheduling**.

`_schedule_priority_preemption` - Handling High-Priority Requests

This function ensures **higher-priority sequence groups** get scheduled **before lower-priority ones**, **even if they arrive later**.

🔹 What It Does

1. **Sorts the running queue by priority**
   - **Higher-priority requests** are moved to the **front** of the running queue.
   - Priority is determined based on `_get_priority()`, which factors in:
     - **User-defined priority values**
     - **Arrival time** (earlier requests get preference if priority is the same)
2. **Checks if a new high-priority request needs preemption**
   - The function **picks the highest-priority waiting request** from `self.waiting`.
   - If this new request **cannot fit** into the available compute/memory budget, it looks for **lower-priority running sequences** to **preempt**.

`_schedule_priority_preemption` - Handling High-Priority Requests

Preempts lower-priority running sequences if necessary

- If a **lower-priority sequence is using resources**, it is **forcibly stopped** to make space for the new request.
- The preempted sequence is **moved back to the waiting queue**.

Reinserts the new sequence into the waiting queue if preemption fails

- If it **fails to preempt** a running sequence, the high-priority request is **put back** into the waiting queue.
- This ensures **it remains in line** for the next scheduling attempt.

Returns the number of preemptions performed

- The function counts how many sequences **were forcefully removed** and returns this number.

`_schedule_priority_preemption` - Handling High-Priority Requests

🔹 Why It's Needed

- Ensures **important tasks** (e.g., system-critical requests) get scheduled **ASAP**.
- Prevents **priority inversion**, where a **low-priority task blocks** a more important one.
- Allows **dynamic adjustments** when new high-priority tasks enter the queue.

`_schedule_priority_preemption` - Handling High-Priority Requests

🔹 Example Scenario

Request ID	Priority	Current Status
`A`	High (5)	Waiting
`B`	Medium (3)	Running
`C`	Low (1)	Running

If `A` **cannot be scheduled due to memory limits**, the function will:

1. **Identify the lowest-priority running task (`C`)**.
2. **Preempt `C`** and put it **back into waiting**.
3. **Schedule `A` in its place**.

`_schedule_prefills` - Managing Prompt Prefill Requests

This function **schedules sequences in the "waiting" state** that are in the **prefill stage**. Prefilling means **loading the initial tokens** before decoding.

🔹 What It Does

1. **Checks if there is enough budget for prefill requests**
   - If the **remaining token budget is zero**, it **immediately exits** because no more sequences can be processed.
2. **Iterates over the waiting queue**
   - **Each request is examined** to see if it **can be scheduled**.

`_schedule_prefills` - Managing Prompt Prefill Requests

Handles very long prompts that exceed token limits

- If a prompt **exceeds the configured maximum length**, it is **marked as ignored** and **removed from processing**.

Determines prefill feasibility with lookahead slots

- **Lookahead decoding** is used when speculative execution is enabled.
- The scheduler determines **if enough blocks exist** to process the full request.

`_schedule_prefills` - Managing Prompt Prefill Requests

Schedules the valid prefill requests

- Sequences that **pass all checks** are:
    - **Allocated** in memory.
    - **Added to the running queue**.
    - **Processed for token generation**.

Updates scheduling metadata

- Keeps track of:
    - **Ignored sequences** that could not fit.
    - **Number of lookahead slots used**.
    - **Remaining token budget**.

`_schedule_prefills` - Managing Prompt Prefill Requests

🔹 **Why It's Needed**

- Ensures **new user requests can start** rather than getting stuck in the waiting queue.
- Prevents **very long prompts** from overloading the system.
- **Optimizes memory usage** by scheduling prefill sequences efficiently.

`_schedule_prefills` - Managing Prompt Prefill Requests

🔹 Example Scenario

Request ID	Prompt Tokens	Prefill Allowed?	Status
`X`	500	✅ Fits in budget	**Scheduled**
`Y`	10,000	❌ Too long, ignored	**Finished (ignored)**
`Z`	3,000	✅ Scheduled partially	**Running (chunked prefill)**

If request `Y` **exceeds the max length**, it is **removed** so it does not block the queue.
If request `Z` **is too large to fit at once**, **chunked prefill** is used to **schedule it in smaller parts**.

`_schedule_default` - General Scheduling Strategy

This function **handles scheduling of all queued requests** while optimizing for **maximum throughput**.

🔹 **What It Does**

1. **Initializes the scheduling budget**
   - Determines the **max tokens and sequences** that can be processed in a batch.
   - Includes **already running requests** in the budget calculation.
2. **Handles different scheduling cases**:
   - If **swapped-out requests** exist → prioritize scheduling them.
   - If no requests were swapped → schedule **prefill requests** first.
   - If prefill requests **were not scheduled**, try running **decode requests**.

`_schedule_default` - General Scheduling Strategy

Ensures no overcommitment

- Verifies that the **number of batched tokens** and **running sequences do not exceed the configured max**.

Updates state queues

- Moves **preempted sequences** back to the waiting queue.
- Adds **scheduled prefills** and **decoding tasks** to the running queue.
- Updates **swapped-out sequences**.

`_schedule_default` - General Scheduling Strategy

Merges scheduling outputs

- Combines **scheduled prefill requests, decodes, and swapped-in requests** into a final batch.

Returns final scheduling results

- Includes:
  - Number of **prefill sequences scheduled**.
  - Number of **batched tokens used**.
  - Requests that need **swapping in/out**.
  - **Ignored sequences** that couldn't be scheduled.

`_schedule_default` - General Scheduling Strategy

🔹 Why It's Needed

✅ Ensures **efficient GPU utilization** by **prioritizing batch scheduling**.
✅ Prevents **starvation** by ensuring waiting requests get **processed fairly**.
✅ **Dynamically balances** prefill and decode stages to **maximize throughput**.
✅ **Handles memory pressure** by **swapping out low-priority sequences** when needed.

`_schedule_default` - General Scheduling Strategy

🔹 Example Scenario

Request ID	Type	Priority	Status
`A`	Prefill	**High**	Waiting
`B`	Decode	**Medium**	Running
`C`	Swapped	**Low**	Swapped

Step 1: Prefill request `A` is scheduled first **(since prefills are prioritized)**.

Step 2: Decode request `B` runs **if prefill scheduling leaves space**.

Step 3: If enough memory remains, **swapped request `C` is restored**.

`_schedule_chunked_prefill` - Optimized Prefill & Decode Scheduling

This function **allows prefill requests to be chunked**, meaning **they are processed in smaller parts** alongside decode requests.

🔹 What It Does

1. **Schedules decoding requests first**
    - Ensures **already running requests** get priority.
    - Uses **First Come, First Serve (FCFS)** scheduling.
2. **Schedules swapped-out requests if possible**
    - If no preemptions happened, it **restores previously swapped-out sequences**.

`_schedule_chunked_prefill` - Optimized Prefill & Decode Scheduling

Schedules chunked prefill requests

- If a prefill request is **too large to fit**, it is **processed in smaller chunks**.

Prioritizes prefill sequences that are close to finishing

- Ensures sequences that **are about to move to the decode stage** get **scheduled first**.
- Prevents inefficient scheduling where some sequences **stay in prefill for too long.**

Merges outputs and returns the final batch

- Combines **prefill, decode, and swapped sequences**.
- Returns the **updated scheduling budget, number of processed tokens, and preempted sequences**.

`_schedule_chunked_prefill` - Optimized Prefill & Decode Scheduling

🔹 **Why It's Needed**

✅ **Maximizes GPU efficiency** by **batching decode & prefill requests together**.
✅ **Reduces token latency** by **processing decodes without blocking prefill requests**.
✅ **Handles large requests better** by **chunking prefills instead of ignoring them**.
✅ **Ensures fairness** by **prioritizing sequences that are close to decoding**.

`_schedule_chunked_prefill` - Optimized Prefill & Decode Scheduling

🔹 Example Scenario

Request ID	Type	Tokens	Status
`X`	Prefill	**3000**	Waiting
`Y`	Decode	**500**	Running
`Z`	Prefill	**10,000**	Waiting

Step 1: Decode request `Y` **runs first**.

Step 2: Prefill request `X` **is chunked and partially scheduled**.

Step 3: Prefill request `Z` **is split into smaller chunks** to **fit in the available memory**.

🔹 Default Scheduling vs. Chunked Prefill Scheduling

1️⃣ Default Scheduling (`_schedule_default`)

- **Prefill → Decode → Swap**
- Optimized for **throughput** when prefill requests are dominant.
- Prefers **batching** multiple prefill sequences together before moving to decode.

2️⃣ Chunked Prefill Scheduling (`_schedule_chunked_prefill`)

- **Decode → Swap → Prefill**
- Optimized for **low-latency decoding** by ensuring decode requests are prioritized first.
- Prefill requests are scheduled **incrementally** to avoid blocking decode requests.

🔹 Default Scheduling vs. Chunked Prefill Scheduling

🟢 When to Use Default Scheduling (`_schedule_default`)?

✅ **Best for batch inference** where multiple prefill requests are expected.
✅ Ensures **maximum prefill throughput** before moving to decoding.
✅ Useful when we have **many queued requests** and want to start them all together.

🟢 When to Use Chunked Prefill (`_schedule_chunked_prefill`)?

✅ **Best for interactive workloads** where decoding must not be blocked by prefill requests.
✅ Ensures **low latency for decoding**, preventing long wait times.
✅ Uses **speculative decoding and chunked execution** to improve efficiency.

`_order_finishing_prefills_first` - Prioritizing Prefill Completion

🔹 What It Does

- This function **reorders prefill sequences** so that those that are **about to finish** prefilling are processed **first**.
- It **splits** the `scheduled_prefill_seqs` list into two groups:
  1. **Finishing sequences** → those that will complete prefilling in the next step.
  2. **Not-finishing sequences** → those that still need more prefill steps.
- It **returns a reordered list** where finishing sequences come **before** non-finishing ones.

`_order_finishing_prefills_first` - Prioritizing Prefill Completion

🔹 Why It’s Needed

✅ Ensures that sequences **close to completion** move to the next phase quickly.
✅ Reduces **stalling** by making sure prefills don't **linger unnecessarily**.
✅ Improves **overall efficiency** and **minimizes delays** in scheduling.

`_order_finishing_prefills_first` - Prioritizing Prefill Completion

🔹 Example Scenario

Seq Group ID	Uncomputed Tokens	Chunk Size	Status
`A`	5	5	✅ Finishing
`B`	10	5	❌ Not Finishing
`C`	2	2	✅ Finishing

💡 **Before Sorting:** `[B, A, C]`
💡 **After Sorting:** `[A, C, B]`

Now, **A and C** will **finish first**, preventing unnecessary delays.

`_schedule` - Deciding Which Scheduling Strategy to Use

🔹 What It Does

- Determines **whether to use**:
  - **Chunked prefill scheduling** (`_schedule_chunked_prefill`) if enabled.
  - **Default scheduling** (`_schedule_default`) if chunked prefill is disabled.

`_can_append_slots` - Checking if More Slots Can Be Added

🔹 What It Does

- Determines **whether additional memory (KV cache slots)** can be allocated for a sequence group.
- Takes into account:
  1. **Artificial preemption** (used for testing forced preemptions).
  2. **Lookahead slots** (future memory space needed for ongoing computations).
  3. **Current memory availability** in the **block manager**.

`_can_append_slots` - Checking if More Slots Can Be Added

🔹 Why It's Needed

✅ Prevents **out-of-memory issues** by **checking memory constraints** before allocating.
✅ Ensures **prefill and decode tasks** can run smoothly **without exceeding limits**.
✅ Supports **multi-step and chunked-prefill modes**, enabling **efficient batch processing**.

`_can_append_slots` - Checking if More Slots Can Be Added

🔹 Example Scenario

Seq Group ID	Current Slots	Lookahead Slots Needed	Can Append?
`A`	50	10	✅ Yes
`B`	95	10	❌ No (Exceeds 100 capacity)

If `B` tries to append more slots, **it will be rejected** due to memory limits.

This prevents **instability** and **optimizes scheduling**.

`_allow_async_output_proc` - Allowing Asynchronous Postprocessing

🔹 What It Does

- Determines **whether async output processing** (post-processing step) is allowed **for a sequence group**.
- **Conditions:**
  - If **sampling parameters are `None`**, or
  - If **only one sequence (`n == 1`) is present in the group**, then **async processing is allowed**.

`_allow_async_output_proc` - Allowing Asynchronous Postprocessing

🔹 **Why It's Needed**

✅ Ensures that **sequences are processed efficiently** after completion.
✅ **Prevents unnecessary async processing** when multiple sequences are being handled.
✅ **Optimizes post-processing** by allowing **parallel execution** when safe.

`_allow_async_output_proc` - Allowing Asynchronous Postprocessing

🔹 Example Scenario

Seq Group ID	Sampling Params	Allowed?
`X`	`None`	✅ Yes
`Y`	`n = 1`	✅ Yes
`Z`	`n = 5`	❌ No

- **X and Y can be processed asynchronously**, but **Z must be handled synchronously**.
- This ensures **efficient scheduling without overloading resources**.

Detailed Breakdown of schedule

Detailed Breakdown of `schedule`

🔹 What This Function Does

The `schedule` function is responsible for:

1. **Scheduling sequence groups** → Decides which sequence groups should be executed next.
2. **Preparing metadata** → Constructs data structures required for executing sequences.
3. **Handling memory and caching** → Updates KV cache blocks and async processing flags.
4. **Tracking execution time** → Measures scheduling latency and updates metrics.

This function **coordinates scheduling, execution, and memory management**, ensuring sequences are processed **efficiently**.

1️⃣ Scheduling Sequence Groups

- Starts a **timer** (`scheduler_start_time`) to track scheduling time.
- Calls `_schedule()` to **determine which sequence groups should run**.
- Captures the current time (`now`) for logging.

2️⃣ Handling Prefix Caching

If **prefix caching is disabled**, initializes an **empty list** for computed block numbers.

Prefix caching allows reusing parts of previously computed sequences, improving efficiency.

Example

Prefix Caching Enabled?	Behavior
✅ Yes	Uses previously computed sequence parts to reduce computation.
❌ No	Recomputes everything from scratch.

3️⃣ Processing Scheduled Sequences

Loops over **all scheduled sequence groups**.

Extracts **token chunk size** (the number of tokens assigned for processing).

Sets the **first scheduled time** if it's the first time the sequence is running.

4️⃣ Preparing Metadata for Sequence Groups

- Retrieves **a cached metadata object** to avoid unnecessary allocations.
- **Clears old data** (ensures fresh metadata for each scheduling cycle).

Why This Matters?

✅ Prevents **memory bloat** by reusing cached objects.
✅ Ensures **accurate tracking** of scheduled sequences.

5 Handling Encoder-Decoder Models

If the **model is encoder-decoder** (e.g., **T5, BART**), it retrieves:

- **The encoder sequence** (`encoder_seq`).
- **Cross-attention block table** (used in attention layers).

Otherwise, sets them to `None`.

6️⃣ Retrieving Sequence Data & Memory Blocks

- Loops through **running sequences** and:
  - Retrieves **sequence data**.
  - Retrieves **KV cache memory blocks** (`block_table`).
  - **Marks blocks as accessed**, preventing them from being evicted.

Why This Matters?

✅ Ensures that **memory blocks stay available** while sequences are running.
✅ **Prevents cache eviction issues** that could cause recomputation.

7️⃣ Managing Prefill Sequences & Sampling

- Determines if this **sequence is in the prefill stage** (`is_prompt`).
- Checks if this is the **first prefill** (`is_first_prefill`).
- If the sequence **is still being prefetched**, **sampling is disabled** (`do_sample = False`).

Example

Condition	Behavior
First prefill (`num_computed_tokens == 0`)	✅ Sampling enabled.
Prefill still in progress	❌ Sampling disabled.
Sequence is decoding	✅ Sampling enabled.

8️⃣ Constructing Sequence Group Metadata

- Creates **metadata** for each scheduled sequence.
- Stores **important details**, including:
  - **Request ID**.
  - **Sampling settings**.
  - **Memory block locations**.
  - **Whether this is a prefill or decode request**.

Why This Matters?

✅ Ensures **workers have all necessary data** to process the sequence.
✅ Prevents unnecessary **recomputations** by tracking memory locations.

9️⃣ Marking Blocks as Computed

- Marks **processed memory blocks** as **computed**.
- Ensures that **subsequent requests don’t need to recompute the same blocks**.

🔟 Updating Scheduling Time for Metrics

- Measures **how long the scheduling process took**.
- Updates **scheduling time for all running sequences** (used for latency tracking).

Detailed Overview of Memory Management & Scheduling Functions

1️⃣ Forking Sequences (`fork_seq`)

🔹 What It Does

This function **creates a duplicate (child) sequence** from a parent sequence. It ensures that the child sequence inherits necessary data while remaining independent.

🔹 Why It's Important

- Allows **parallel sequence processing**, useful for **beam search** or **sampling multiple candidates**.
- Ensures that **child sequences** have the same memory state as the parent before diverging.

🔹 Example Use Case

Imagine a model generating multiple sentence completions from the same starting text:

- **Parent Sequence:** `"The quick brown fox"`
- **Child Sequences:** `"The quick brown fox jumps"`, `"The quick brown fox runs"`

By **forking the parent**, both child sequences start with the same state.

2️⃣ Freeing Sequences (`free_seq`)

🔹 What It Does

This function **deallocates** a sequence from memory.

🔹 Why It's Important

- **Prevents memory leaks** by releasing unused sequences.
- Ensures that **only active sequences consume resources**.

🔹 Example Use Case

- A sequence reaches the **end of generation** → It should be removed from memory.
- A user **aborts a request** → The sequence should be freed immediately.

3️⃣ Freeing Finished Sequences (`_free_finished_seqs`)

🔹 **What It Does**

- Iterates through a **sequence group** and **removes finished sequences**.
- Calls `free_seq` for each **completed sequence**.

🔹 **Why It's Important**

- Ensures that **partially finished groups keep running** while **completed ones are removed**.
- Avoids **keeping unnecessary sequences in memory**.

🔹 **Example Use Case**

- A batch contains **5 sequences**; 3 are **finished**, but 2 are **still generating**.
- The finished ones **should be freed**, while the rest **continue running**.

4️⃣ Freeing a Finished Sequence Group (`_free_finished_seq_group`)

🔹 What It Does

- If a **sequence group** is **fully completed**, it:
  1. **Frees cross-attention memory** (for models like T5, BART).
  2. **Adds the request ID to the finished requests list** (for tracking).
  3. **Calls `_free_finished_seqs`** to clear individual sequences.

🔹 Why It's Important

- Properly **cleans up groups** when they are **completely done**.
- Ensures that **no resources are wasted** on finished groups.

5️⃣ Freeing All Finished Sequence Groups (`free_finished_seq_groups`)

🔹 What It Does

1. **Iterates over running sequences** and **removes finished ones**.
2. If sequences were **asynchronously stopped** (e.g., exceeded model limits), they are also freed.

🔹 Why It's Important

- Prevents **old sequences from accumulating in memory**.
- Handles **edge cases** where sequences were **stopped due to model constraints**.

6️⃣ Allocating and Running Sequences (`_allocate_and_set_running`)

🔹 What It Does

- Allocates memory for a **sequence group**.
- Updates the status of **waiting sequences** to **running**.

🔹 Why It's Important

- Ensures **sequences transition smoothly** from **waiting** to **execution**.
- Prevents scheduling issues where sequences are **stuck in the queue**.

🔹 Example Use Case

A request is **waiting** due to memory constraints.
Once memory is available, `_allocate_and_set_running` moves it to **execution**.

7️⃣Appending Memory Slots (`_append_slots`)

🔹 **What It Does**

- Allocates **new memory slots** for sequences **if needed**.
- Handles **multi-step scheduling**, where sequences **expand over time**.

🔹 **Why It's Important**

- Supports **chunked prefill**, where sequences **receive memory incrementally**.
- Prevents out-of-memory (OOM) errors by **dynamically expanding allocation**.

🔹 **Example Use Case**

- A sequence needs **more memory** to **continue generation**.
- Instead of **pre-allocating** a huge chunk, `_append_slots` **expands only as needed**.

Detailed Overview of Preemption and Swapping Functions

1️⃣Preempting Sequences (`_preempt`)

🔹 What It Does

- When there isn't enough **KV cache memory**, this function **preempts** a sequence group to **free space**.
- It decides whether to:
    1. **Recompute** the sequence later (`PreemptionMode.RECOMPUTE`).
    2. **Swap** it out to CPU memory (`PreemptionMode.SWAP`).

🔹 Why It's Important

- Prevents **out-of-memory errors** by dynamically freeing memory.
- Balances **performance vs. memory usage**:
    - **Recompute:** Uses less memory but may slow execution.
    - **Swap:** Uses CPU memory but avoids recomputation.

1️⃣ Preempting Sequences (`_preempt`)

🔹 Decision Logic

Case	Mode Used
Sequence group has **only 1 sequence**	**Recompute** (preferred for lower overhead)
Sequence group has **multiple sequences** (e.g., beam search)	**Swap** (recompute isn't supported)
User explicitly selects **swap mode**	**Swap**
User explicitly selects **recompute mode**	**Recompute**

🔹 Example Use Case

- A **beam search operation** with **5 candidate sequences** is running.
- The system **runs out of KV cache memory**.
- `_preempt` decides to **swap out** the entire sequence group.

2️⃣ Preempting via Recompute (`_preempt_by_recompute`)

🔹 What It Does

- **Clears the sequence's memory** so it can be recomputed later.
- Resets the sequence's state and **moves it back to the waiting queue**.

🔹 Why It's Important

- Saves **memory** by discarding intermediate values.
- Can **restart the sequence later** instead of discarding it.

🔹 Example Use Case

- A user **starts generating text**, but the model **runs out of memory**.
- Instead of discarding the request, the system **removes it from active execution** but **keeps it in the queue** for later processing.

3️⃣ Preempting via Swap (`_preempt_by_swap`)

🔹 **What It Does**

- Moves a **sequence group from GPU to CPU swap space** to **free up memory**.
- Calls `_swap_out` to execute the actual swap operation.

🔹 **Why It's Important**

- Ensures **active execution continues** by **moving less urgent sequences** to CPU.
- Reduces **the need for recomputation**, making it more efficient.

🔹 **Example Use Case**

- A **large model** with **multiple active users** is running.
- Some **low-priority requests** are **moved to CPU swap space** so that higher-priority ones can continue running.

4️⃣ Swapping In a Sequence (`_swap_in`)

🔹 **What It Does**

- Moves a **sequence group from CPU back to GPU memory**.
- Updates the sequence’s status to **running**.

🔹 **Why It's Important**

- Enables **previously swapped-out requests** to resume execution **without recomputation**.

🔹 **Example Use Case**

- A **sequence was swapped out** due to memory limits.
- Now that memory is available, `_swap_in` **brings it back** to GPU for execution.

5️⃣ Swapping Out a Sequence (`_swap_out`)

🔹 **What It Does**

- Moves a **sequence group from GPU to CPU memory**.
- Marks the sequence as **swapped**.

🔹 **Why It's Important**

- **Prevents memory crashes** by ensuring **GPU memory is efficiently used**.
- Avoids **killing requests** when memory runs low.

🔹 **Example Use Case**

- A **low-priority request** is **swapped out** to free up GPU memory for **more urgent tasks**.

Detailed Overview of Scheduling & Memory Management Functions

1️⃣ Delaying Scheduling for Efficiency (`_passed_delay`)

🔹 **What It Does**

- **Introduces a delay before scheduling new requests** to allow the waiting queue to fill up.
- Helps **batch multiple requests together** for **better throughput**.

🔹 **Why It's Important**

- **Prevents scheduling small inefficient batches.**
- **Optimizes GPU usage** by processing multiple requests together.

🔹 **Example Use Case**

- If a **single request arrives**, instead of processing it immediately, it waits **until more requests arrive** to create a batch.

2️⃣ Determining Lookahead Slots (`_get_num_lookahead_slots`)

🔹 **What It Does**

- Determines **how many extra slots** to allocate for speculative decoding.
- **Prefill requests generally don’t use lookahead slots**, except in **multi-step chunked-prefill mode**.

🔹 **Why It's Important**

- **Reduces recomputation** by preallocating space.
- **Improves throughput** in speculative decoding.

🔹 **Example Use Case**

- A **multi-step model** that predicts **multiple tokens at once** would **allocate additional slots** in anticipation of speculative decoding.

3️⃣Calculating New Cached & Uncached Tokens (`_get_num_new_uncached_and_cached_tokens`)

🔹 What It Does

- Computes **how many tokens are newly generated and require computation** vs. **how many tokens are already cached and can be reused**.
- If **cache is enabled**, some tokens are **already precomputed and stored in memory**.
- If **cache is disabled**, all tokens need recomputation.

🔹 Why It's Important

- **Minimizes computation by reusing cached tokens** where possible.
- **Prevents memory overuse by limiting how many new tokens can be scheduled**.

3️⃣ Calculating New Cached & Uncached Tokens (`_get_num_new_uncached_and_cached_tokens`)

🔹 Example Use Case

- A model **generates a long response**, but **only the last token needs recomputation** because previous tokens are already cached.

Scenario	Uncached Tokens	Cached Tokens
Full recomputation required	**All tokens**	**0**
Some tokens are already cached	**Only new tokens**	**Remaining tokens**
Fully cached sequence	**1 (last token)**	**All previous tokens**

4️⃣Chunking New Tokens Based on Budget (`_chunk_new_tokens_to_schedule`)

🔹 What It Does

- **Reduces the number of tokens scheduled at once** if memory constraints exist.
- Ensures that **new tokens don’t exceed the available budget**.
- When **prefix caching is enabled**, ensures that **cached tokens fit into existing memory blocks**.

🔹 Why It's Important

- Prevents **out-of-memory issues**.
- **Balances memory vs. computational efficiency**.

4️⃣ Chunking New Tokens Based on Budget (`_chunk_new_tokens_to_schedule`)

🔹 Example Use Case

- A **request wants to generate 100 tokens**, but **memory only allows for 50 at a time** → **chunks it into 50-token steps**.

Scenario	Original Tokens Requested	Tokens Scheduled (Chunked)
Memory available	**100**	**100**
Limited memory	**100**	**50**
Prefix caching enabled (block-based allocation)	**100**	**Multiple of block size (e.g., 64)**

How Does Lookahead Work?

Step 1: Determine How Many Slots Are Needed

The function `_get_num_lookahead_slots(is_prefill, enable_chunking)` determines the number of lookahead slots based on:

- **If the sequence is in the prefill stage**:
  - If **multi-step scheduling and chunked prefill** are enabled, lookahead slots **are allocated in advance**.
  - Otherwise, **no lookahead is allocated for prefills**.
- **If the sequence is in the decoding stage**:
  - The **configured number of lookahead slots** is used.
  - This number is typically **7 for an 8-step speculative decoding** (meaning 1 step happens as normal, and 7 more are speculated).

🔹 How Does Lookahead Work?

Step 2: Allocate Lookahead Slots

- If a sequence is in **speculative decoding**, the scheduler **allocates slots** for potential future token computations.
- If **multi-step chunking is enabled**, the prefill sequences get **extra slots** to ensure smooth transitions.

Step 3: Use or Discard the Lookahead Slots

- **If the guessed tokens are correct** → Computation continues **without redoing previous work**.
- **If the guess is wrong** → The lookahead slots are **discarded**, and the correct tokens are recomputed.

Final Overview of the Scheduler Logic

🔹 What the Scheduler Does

The scheduler is responsible for efficiently managing **sequence execution** and **memory allocation** in a system where multiple sequences (requests) are being processed. It **balances performance and memory constraints** while ensuring fairness among sequences.

The logic consists of several key components:

1. **Managing Different Sequence States**
   - Sequences move through **Waiting** → **Running** → **Swapped** states.
   - The scheduler ensures that **requests are executed in the right order** while **prioritizing efficiency**.
2. **Handling Preemption and Swapping**
   - When **memory is full**, **low-priority sequences are preempted** (paused or swapped out).
   - The scheduler **decides whether to recompute or swap out** sequences based on constraints.

🔹 What the Scheduler Does

1. **Optimizing Execution Order**
   - It **prioritizes certain sequences based on importance** (e.g., prefill vs. decoding).
   - Implements **priority-based preemption** to **avoid priority inversion**.
2. **Memory Management and Token Processing**
   - **Tracks cached vs. uncached tokens** to reduce recomputation.
   - **Uses chunking** when necessary to fit within memory budgets.
   - **Allocates lookahead slots** for speculative execution.

🔹 How the Scheduler Works Step-by-Step

1️⃣ Prefill & Priority Preemption

- **Prefill requests** (input sequences) are scheduled **first.**
- If there’s no room, **lower-priority running sequences are preempted**.
- Preempted sequences are either:
  - **Recomputed** later (**lower overhead** but delays processing).
  - **Swapped out** to CPU memory (**faster recovery but higher overhead**).

🔷 How the Scheduler Works Step-by-Step

2️⃣ Scheduling Running Sequences

- Decoding (generating new tokens) is **prioritized** when prefills are done.
- Checks **how many tokens can be scheduled** based on memory constraints.
- If memory is full, the scheduler **preempts lower-priority sequences**.

🔹 How the Scheduler Works Step-by-Step

3️⃣ Handling Swapped Sequences

- When memory is freed, **swapped-out sequences are reloaded** if space allows.
- If there’s **not enough space**, these sequences **remain in storage**.

🔹 How the Scheduler Works Step-by-Step

4️⃣ Finalizing Execution

- Completed sequences are **freed from memory**.
- Their **cached tokens are reused** to minimize recomputation.
- The scheduler **tracks compute time** to optimize for future requests.

🔹 Key Trade-Offs in the Scheduler

Decision	Impact
Prefill First vs. Decode First	Prefill-first maximizes throughput, while decode-first improves response time.
Recompute vs. Swap	Recompute is **cheaper** but **slower**; Swap is **faster** but **memory-intensive**.
Delaying Scheduling	Batching increases efficiency, but too much delay increases latency.
Chunking Tokens	Helps prevent memory overflow but can slow down execution.

🔹 Conclusion: What the Scheduler Achieves

The scheduler **ensures fast, efficient, and memory-aware execution** of sequences by: ✅ **Dynamically prioritizing requests** (prefill, decoding, swapped-in).
✅ **Handling memory limitations** via swapping & preemption.
✅ **Minimizing computation overhead** with caching & chunking.
✅ **Maintaining fairness** while optimizing throughput.

vLLM Platform