MP5: Simple Kernel Threads

The Scheduler class implements a basic thread scheduling mechanism for an operating system, utilizing a First-In-First-Out (FIFO) approach to manage threads within the ready queue. It provides fundamental operations such as yielding, resuming, adding, and terminating threads.

## scheduler.h

The header file for the Scheduler class defines the essential structure of the scheduler, including the ready queue (implemented as a linked list), private members for the current thread, and function declarations for thread management operations. It also defines a nested LinkedList class, which provides efficient management of the thread queue.

**Scheduler Class:** The Scheduler class manages thread scheduling operations. It contains the following.

Static Members:

* **ready\_queue:** A linked list of threads representing the ready queue.
* **current\_thread:** Pointer to the currently running thread.

LinkedList Class:

* A simple singly-linked list that serves as the backbone of the ready queue. It includes basic operations for adding, removing, and popping threads from the list.

### Functions:

**yield:** Allows the current thread to give up the CPU and selects the next thread from the ready queue.

**resume:** Adds a thread to the ready queue, making it eligible for execution.

**add:** Adds a newly created thread to the ready queue by invoking resume.

**terminate:** Safely removes a thread from the ready queue and, if it’s the current thread, calls yield to switch execution.

## scheduler.cpp

The implementation of the Scheduler class provides the functional logic for managing thread scheduling and context switching. The file includes definitions for the linked list methods and the main scheduler functions (yield, resume, add, and terminate).

**LinkedList Class Implementation**

The LinkedList class manages the threads within the ready queue. It features the following methods:

* **Constructor/Destructo**r: Initializes or cleans up the linked list.
* **add**: Adds a thread to the end of the list, updating tail.
* **remove**: Finds and removes a specific thread by traversing the list. It maintains list integrity by updating pointers to avoid broken links.
* **pop\_front**: Removes and returns the first thread in the list, typically used in yield.
* **is\_empty**: Checks if the list has any nodes.

## Scheduler Class Implementation

**Static Initialization**: The current\_scheduler pointer is initialized to nullptr and set in the Scheduler constructor to ensure the scheduler is accessible globally as a singleton.

**Scheduler Constructo**r: The Scheduler() constructor initializes the scheduler instance and sets current\_thread to nullptr. It may also be extended to set up an idle thread to handle cases when no other threads are available.

**yield()**

* Adds current\_thread to the end of the ready\_queue.
* Pops the next thread from the front of the ready\_queue and performs a context switch to this thread using Thread::dispatch\_to.
* If the queue is empty, outputs a message indicating no threads are available.

**resume()**

* Directly adds a thread to the ready\_queue, making it runnable.

**add()**

* Calls resume() to add a new thread to the queue, abstracting away the ready queue management for newly created threads.

**terminate()**

* Removes a specified thread from the ready\_queue.
* If the terminating thread is the current\_thread, it calls yield() to switch control, ensuring the terminated thread does not continue execution.

## vm\_pool.cpp

The implementation of the Thread class provides the functional logic for initializing, starting, terminating, and dispatching threads. This file also includes helper functions, thread\_start and thread\_shutdown, which handle interrupt management and cleanup, respectively. The following are the functions I have changed and implemented.

**thread\_start()**

* Enables interrupts, allowing the thread to begin execution in a state where it can respond to interrupts and preemptions. This function is set as the entry point in setup\_context and is called when the thread starts running.

**thread\_shutdown()**

* Terminates a thread by releasing resources and notifying the scheduler. Calls SYSTEM\_SCHEDULER->terminate() on the current thread, ensuring it is removed from the scheduler’s ready queue and gracefully exits the CPU.
* Prints the thread ID to console output for debugging purposes, providing a clear indication of when threads are terminated.

## Debugging

Debug print statements are used in critical areas to verify thread initialization and termination. Thread Creation and Shutdown: Outputting thread IDs during creation and shutdown helps ensure that the thread lifecycle is functioning correctly.