Operating Systems Project Guidelines

1. Overview

The course project is a core component of this class.
You will work directly with a real Linux kernel to explore, modify, and evaluate critical operating system subsystems such as **process scheduling**, **memory management**, **I/O**, and **file systems**.

---

2. Group Organization and Topic Selection

- **Team Size:** 2 students per group.
- **Topic Assignment:**
  - Each group will choose **one primary topic** from the official topic list.
  - Each topic focuses on a specific OS subsystem (Scheduler, Memory, I/O, or File System).
  - Topics will be distributed on a *first-come, first-served* basis to ensure diversity and **each topic** can be chosen by a **maximum of two teams**.

3. Project presentation and report

Each group will deliver both a **presentation** and a **final report**.

(a) Presentation

- Duration: **30 minutes per group and 10 minutes for Q&A**.
- The details will be announced later. **The two groups that choose the same topic will give their presentations in the same class**.
- Content should include:
  - Project motivation and objectives.
  - Key kernel components modified or analyzed.
  - Demonstration of implementation or results.
  - Key findings, challenges, and insights.
- Each member should participate in the presentation.

(b) Final Report

- The report should include:
  1. **Introduction & Motivation** — What problem are you solving and why is it interesting?
  2. **Background & Related Work** — Describe the subsystem and relevant kernel mechanisms.
  3. **Design & Implementation** — Detail your modifications or extensions to the kernel.
  4. **Experimental Setup & Methodology** — Explain how you tested and evaluated performance.

5. **Results & Analysis** — Present and discuss your findings using graphs and tables.
6. **Conclusion** — Summarize your project.

Topics

These topics and descriptions are provided for your reference. You can decide the specific experimental environment and project goals on your own.

I. Process Management (4 topics)

Topic 1: Modify Linux Kernel Scheduler

Objective: Understand and improve the Linux Completely Fair Scheduler (CFS).
Basic Requirements:

- Locate and modify the scheduling logic in the kernel source code.
- Implement custom scheduling policies (such as priority-based proportional allocation, random scheduling, or multi-core load balancing optimization).
- Conduct performance comparisons by running `sysbench`, or custom workloads.
- Display the process status through the GUI.

Advanced Options:

- Dynamically switch scheduling policies.
- Multi-core load balancing optimization or NUMA-aware scheduling policies.

Topic 2: Kernel-Level Thread Implementation and Analysis

Objective: To gain an in-depth understanding of the Linux thread model (shared task_struct structure).
Basic Requirements:

- Modify or expand the logic for creating kernel threads (kthreads).
- Add custom fields/statistics (such as the number of context switches, CPU affinity).
- Compare the differences between kernel threads and user threads in terms of performance and resource isolation.

Advanced Options:

- Implement a lightweight user-space thread library (coroutines) and compare it with kthreads.
- Support interaction between user-space threads and kernel threads through system calls.

Topic 3: Enhanced IPC Mechanism in the Kernel

Objective: Add a new IPC mechanism to the Linux kernel or optimize an existing one (such as pipe or shared memory).
Basic Requirements:

- Modify or expand files such as `fs/pipe.c` and `ipc/shm.c`.
- Test the latency and bandwidth of the custom mechanism.

Advanced Options:

- Introduce zero-copy IPC.
- Implement a high-performance communication channel using a lock-free ring buffer.
- Optimize cache locality in a multi-core scenario.

Topic 4: Lightweight Container Implementation with Namespaces

Objective: Implement process and environment isolation without using Docker.
Basic Requirements:

- Use Linux namespaces (pid, mnt, uts, net, ipc) directly via system calls to create an isolated container environment.
- Run a shell inside the container and verify isolation effects.

Advanced Options:

- Integrate with cgroups for resource limitation.
- Analyze the context switching and namespace clone() performance impact.
- Add simple image packaging/import functionality.

II. Memory Management (4 topics)

Topic 5: Implement a Custom Page Replacement Policy

Objective: Modify the Linux kernel's page replacement algorithm.
Basic Requirements:

- Implement an alternative policy (e.g., **Working Set**, **Adaptive LRU**).
- Use `vmstat` and `perf` to monitor page fault behavior.
- Display the page replacement behavior and page status through the GUI.

Advanced Options:

- Implement an **Access-Frequency-Aware Replacement** strategy.
- Compare hit rate and performance under different workloads.

Topic 6: Investigating and Tuning the Linux Buddy Allocator

Objective: Study and improve the Linux memory allocation subsystem.
Basic Requirements:

- Add logging or statistical code to monitor allocation patterns.
- Modify the buddy allocation strategy (e.g., adjust compaction priority).

Advanced Options:

- Implement a **fragmentation-threshold-based adaptive buddy system**.
- Compare the performance of **slab**, **slub**, and **slob** allocators.

Topic 7: Copy-on-Write (COW) Optimization Study

Objective: Analyze the behavior and performance of Copy-on-Write.
Basic Requirements:

- analyze memory copy latency.
- Modify the COW trigger logic (e.g., delayed allocation, pre-copy).

Advanced Options:

- Implement a custom system call to manually trigger COW.

Topic 8: Kernel Virtual Memory Mapping Visualizer

Objective: Track and visualize kernel virtual memory mapping changes.
Basic Requirements:

- record process `mmap/unmap` operations.
- Output mapping region changes to `vmmap_log`.
- Develop a **user-space visualization tool** (Python or Web UI).

Advanced Options:

- Track **page table copies** during `fork()` or `exec()`.

III. I/O and Device Management (3 topics)

Topic 9: Implement and Evaluate a Custom Disk Scheduler

Objective: Implement a new I/O scheduling algorithm in the kernel block layer.
Basic Requirements:

- Study `block/blk-mq.c` and existing schedulers (e.g., `mq-deadline`, `bfq`).
- Implement a **priority-aware scheduling algorithm**.

- Evaluate performance using `fio`.

Advanced Options:

- Design an **adaptive scheduler** that switches strategy based on queue depth or I/O pattern.
- Compare results between **SSD** and **HDD** devices.

Topic 10: Kernel-Level Buffer Cache Instrumentation

Objective: Analyze kernel page cache hit and write-back behavior.
Basic Requirements:

- Modify `fs/buffer.c` to log cache hit/miss events.
- Expose statistics via `/proc/cache_stats`.
- Implement a **custom cache replacement strategy** (e.g., frequency-based).

Advanced Options:

- Compare cache behaviour under different read/write access patterns.

Topic 11: Asynchronous I/O Benchmark and Enhancement

Objective: Compare performance between `io_uring` and traditional synchronous I/O.
Basic Requirements:

- Write a user-space benchmark comparing `read/write` vs. `io_uring`.
- Measure latency, CPU utilization, and syscall counts.

Advanced Options:

- Modify the kernel's `io_uring` implementation (`fs/io_uring.c`) to add a **new submission strategy**.
- Optimize **queue contention** in multi-threaded scenarios.

IV. File System (4 topics)

Topic 12: Implement a Simple File System in Kernel Space

Objective: Implement a minimal file system (similar to `ext2`) in the Linux kernel.
Basic Requirements:

- Implement **inode**, **superblock**, **directory**, and **data block** management.
- Support basic file operations.

Advanced Options:

- Add **journaling** or **delayed write-back** mechanisms.
- Add **checksum validation** or **metadata compression**.

Topic 13: Journaling and Crash Recovery Mechanism

Objective: Study and modify the **ext4 journaling** mechanism.
Basic Requirements:

- Read and understand the `fs/jbd2/` subsystem.
- Add journal statistics or simulate crash recovery scenarios.

Advanced Options:

- Implement a **lightweight journaling** mechanism (metadata-only).
- Compare **journaling** vs. **copy-on-write (CoW)** mechanisms (as in btrfs).

Topic 14: File System Performance Characterization

Objective: Evaluate the performance of **ext4**, **XFS**, and **btrfs** file systems.
Basic Requirements:

- Use `fio`, `sysbench`, and `dd` to generate different workloads.
- Measure sequential/random I/O performance, latency, and CPU usage.

Advanced Options:

- Tune **mount parameters** (e.g., barrier, journaling mode).
- Design custom workloads (e.g., many small files vs. large sequential writes).

Topic 15: Kernel-Level Data Deduplication Mechanism

Objective: Implement data deduplication at the kernel page cache or file system level.
Basic Requirements:

- Modify or extend `fs/buffer.c` to detect duplicate blocks using hashing.
- Implement basic deduplication logic.

Advanced Options:

- Integrate with **compression algorithms** (zlib/lz4).
- Analyze **CPU–I/O trade-offs** introduced by deduplication.