Data Structures

BLG 223E

Project 4

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 01.01.2025

1. CFS variant using MinHeap

In this assignment, you will implement a simplified version of the Linux Completely Fair Scheduler (CFS). The goal is to allocate CPU time among processes fairly by using a Min-Heap, to track and manage processes based on their "virtual runtime" (vruntime).

1.1. Background

1.1.1. Process Scheduler

A process scheduler is a important part of an operating system, responsible for managing which process is assigned to the CPU at any given time. Its primary functions include:

- **Process Management**: Maintaining a record of all active processes within the system.
- **CPU Allocation**: Determining which process will be allocated CPU time based on its priority and needs.
- **Context Switching**: Handling the switching between processes when needed, ensuring that the CPU’s time is divided efficiently.
- **Priority Handling**: Managing the priority of processes to ensure that high-priority tasks get the appropriate CPU time over others.

1.1.2. Completely Fair Scheduler (CFS)

The Completely Fair Scheduler (CFS) is designed to allocate CPU time to processes in a way that maximizes CPU utilization while ensuring fairness. The scheduler uses a metric called "virtual runtime" (vruntime) to decide which process should be scheduled next. The process with the lowest vruntime is chosen to run, as it is considered to have consumed the least CPU time.

While the original CFS implementation in Linux uses a Red-Black Tree to manage processes, in this assignment, you will implement a simplified version using a Min-Heap.

1. **Virtual Runtime (vruntime)**
    - This is a critical metric that tracks how much CPU time each process has been allocated. It allows the scheduler to maintain fairness by ensuring that processes that have received less CPU time are given higher priority.
    - CFS uses vruntime to decide which process to schedule next. A process with a lower vruntime is considered to have had less CPU time and thus should be scheduled before others.

- The vruntime of a process increases over time, adjusted by the process's priority, so that processes that run for longer will have a higher vruntime and thus be scheduled less frequently.

2. **Nice Values**

   - These values determine the relative priority of processes. They range from -20 (highest priority) to +19 (lowest priority), with a default value of 0.
   - Lower nice values (e.g., -10) cause the process to receive more CPU time, as their vruntime increases more slowly.
   - Higher nice values (e.g., +10) make a process less CPU-hungry, as their vruntime increases more rapidly, causing the scheduler to select them less frequently.

3. **Min-Heap for Process Scheduling**

   - In the CFS, processes need to be managed in a way that allows for efficient insertion, deletion, and retrieval of the process with the lowest vruntime.
   - While the original CFS implementation uses a Red-Black Tree, we utilize a Min-Heap to achieve similar functionality in this assignment.
   - A Min-Heap is a binary heap structure where the process with the lowest vruntime is always at the root. This allows for constant-time access to the process that should be scheduled next.
   - Insertion and removal of processes in a Min-Heap can be done in $O(\log n)$ time, where n is the number of processes, ensuring efficiency even as the number of processes grows.
   - The Min-Heap maintains the order of processes based on their vruntime. When a process has used more CPU time, its vruntime will increase, and it will eventually be placed deeper in the heap, ensuring that it is scheduled less frequently compared to processes with lower vruntime.

In this assignment, we use a Min-Heap to simplify the implementation while still achieving a fair distribution of CPU time. This approach allows for efficient scheduling and quick access to the process that is most deserving of CPU time at any moment, based on the principle of minimizing vruntime.

2. Implementation

You are given the skeleton code for this project.

IMPORTANT: Before Starting Implementation

- Carefully read **ALL** comments in the provided header files and source files.
- The comments contain important implementation details, requirements, and hints that are part of the assignment.
- The comments explain function behaviors, edge cases to handle, and specific requirements not detailed in this document.
- Also, do not change the signature of the available functions in any files.

2.1. Part 1: Implementing the Min Heap (min_heap.c)

Implement a generic min heap data structure as defined in min_heap.h. The Min Heap must be generic, meaning it can store any type of data. To achieve this, we will use a void pointer to hold the heap elements. The size and type of each element will be determined dynamically, and the heap will require a comparison function to determine how elements are ordered within the heap.
Later, this data structure will be used by the scheduler to maintain process ordering.

2.1.1. Requirements

Implement all functions declared in min_heap.h:

- `heap_create()`: Initialize a new heap
- `heap_destroy()`: Clean up heap memory
- `heap_insert()`: Add new element
- `heap_extract_min()`: Remove and return minimum element
- `heap_peek()`: View minimum element without removing
- `heap_size()`: Return current number of elements
- `heap_merge()`: Combine two heaps

2.2. Part 2: Implementing the Scheduler (scheduler.c)

Implement the scheduler functionality as defined in scheduler.h. The scheduler uses a simplified version of the CFS algorithm.

2.2.1. Scheduler Flow

1. **Process Creation and Scheduling**
    - Processes are created with PID and nice value
    - Each process tracks its virtual runtime (vruntime)
    - Processes are stored in min-heap ordered by vruntime
2. **Virtual Runtime Calculation**
    - vruntime increases based on execution time and priority
    - Higher nice values make vruntime increase faster
    - Formula is already implemented in process.c

2.2.2. Implementation Requirements

Implement the following functions in scheduler.c:

1. `create_scheduler()`: Initialize scheduler with given capacity
2. `destroy_scheduler()`: Clean up scheduler resources
3. `schedule_process()`: Add new process to queue
4. `get_next_process()`: Select next process to run
5. `tick()`: Update scheduler state for one time unit

2.3. Evaluation Criteria

1. Correctness of min heap implementation
2. Proper scheduler operation following CFS principles
3. Memory management
4. Code quality and documentation
5. Handling of edge cases

2.4. Simplifications from Real CFS

- Fixed time slice instead of dynamic
- Simpler vruntime calculation
- No CPU load balancing
- No group scheduling
- Single run queue

2.5. Tips for Success

1. Understand the min heap properties thoroughly
2. Study the CFS algorithm flow provided
3. Test with various process combinations
4. Consider edge cases in both components
5. Ensure proper memory management
6. Use provided test framework

3. Compiling and Debugging

To make the coding experience smoother, Makefile and VS Code's JSON files are updated. We added a make rule called `valgrind`. So, after completing the implementation, you can now call `make valgrind` and it will create a `valgrind-out.txt` file that shows your memory leaks and errors if there are any. Also, it suggested for you to look for the other make rules in the file.

Additionally, to find potential bug sources, it is suggested to compile with `-Wall -Werror` flags in the Makefile, but this is completely optional.

Also, you can now directly debug the test files. Select the "Test (build and debug)" option in the debug menu in the VS Code interface.

4. Submission

You have already provided the skeleton code, so you should not submit the whole project. We prepared a **submission script** for you that creates the zip folder automatically. You must use this script:

```
sh submission_script.sh
```

Then directly upload the created zip file to Ninova.

Also, note that the provided test files will not be the only ones used for grading. Your code will be evaluated on additional test cases, so make sure your implementation is robust and can handle various scenarios.

Good luck! For any questions, feel free to e-mail me

IMPORTANT NOTE:

- You must stick with the container environment that is provided to you. This is crucial for the grading of your work. If your code can not be compiled in the container, you will receive a zero grade.
- Copying code fragments from any source, including books, websites, or classmates, is considered plagiarism.
- You are free to conduct research on the topics of the assignment to gain a better understanding of the concepts at an algorithmic level.
- Refrain from posting your code or report on any public platform (e.g., GitHub) before the deadline of the assignment.
- You are **NOT** permitted to use the STL for any purposes in this assignment.
- Only C language is allowed. Do not use C++ in anyway.
- Your code will be analyzed via Valgrind for memory leaks. Check all of the memory allocations and make sure you don't have any memory leaks. If you have any, it will be penalized severely.
- Additionally, please refrain from using any AI tools to help you complete your work. While I cannot directly detect AI-generated code, relying on such tools is not in your best interest. Everything you learn in this class forms the foundation for future courses, and if you do not build a strong foundation now, you will likely struggle in the semesters to come.