CPU Virtualization Code Summary

Overview

Notes on the virtualization code discussed in class.

1 Explanation of `cpu_affine.c`

This program shows how to get and set a process's **CPU affinity** in Linux. CPU affinity controls which CPU cores a process can run on.

- A CPU set is defined and cleared with `CPU_ZERO()`.
- CPU 0 is added using `CPU_SET(0, &set)`.
- `sched_setaffinity()` applies the set, limiting the process to CPU 0.
- The `print_affinity` function displays allowed CPUs before and after setting the affinity.
- The program loops, printing the current CPU using `sched_getcpu()`, and simulates CPU usage.

Printing the CPU Affinity: The `print_affinity` function calls `sched_getaffinity()` to get the set of allowed CPUs. It then iterates through possible CPUs using `CPU_ISSET()` to print which CPUs the process is allowed to run on. This function is used before and after setting the affinity to observe changes.

Key Functions and Macros:

- `sched_setaffinity()`: Applies the CPU mask to the process.
- `CPU_SET()`: Adds a CPU to the mask.
- `CPU_ISSET()`: Checks if a CPU is in the set.

2 Explanation of `fifo.c`

This program demonstrates setting CPU affinity and assigning real-time scheduling to threads using the **FIFO** scheduling policy.

Main Thread and Affinity: The main thread is pinned to **CPU 0** using `sched_setaffinity()`, ensuring all subsequent threads are created while the main thread is running on a known core.

Thread Creation and Affinity: Five threads are created, and each thread is individually pinned to **CPU 0** using `pthread_attr_setaffinity_np()`. This ensures all threads are running on the same core to observe scheduling effects clearly.

FIFO Scheduling Policy: Each thread is scheduled using the **FIFO** (First-In-First-Out) policy, with a fixed priority of **80**. In this policy, higher priority threads preempt lower ones, and threads with the same priority are scheduled in the order they were created.

Thread Behavior: Each thread continuously runs in a loop where it:

- Prints the CPU it is currently running on using `sched_getcpu()`.

- Calls `sched_yield()` to voluntarily yield the CPU, giving other threads a chance to run.

Calling `sched_yield()` in `SCHED_FIFO` causes the thread to re-enter at the end of the run queue for its priority level. If no other equal-priority threads are ready, it may resume immediately. Because `SCHED_FIFO` does not use time slices, omitting `sched_yield()` means the current thread can continue running indefinitely, causing other equal-priority threads to wait until it blocks, finishes, or yields.

Setting Affinity for Processes vs Threads:

- **Process Affinity:** `sched_setaffinity()` applies to the entire process and affects all threads within it.
- **Thread Affinity:** `pthread_attr_setaffinity_np()` sets CPU affinity on a per-thread basis, allowing different threads to run on different CPUs for parallelism.

Key Functions and Macros:

- `sched_setaffinity()`: Sets CPU affinity for a process.
- `pthread_attr_setaffinity_np()`: Sets CPU affinity for individual threads.
- `sched_yield()`: Causes the calling thread to yield the CPU and re-enter the run queue.
- `sched_getcpu()`: Returns the current CPU the thread is running on.

3 Explanation of fifo_prio.c

Overview: This program shows how Linux schedules threads using `SCHED_FIFO`, a real-time policy. It runs two threads—one low-priority and one high-priority—on CPU 0. The low-priority thread starts first and creates the high-priority thread, both of which run a CPU-intensive function that consumes processor time in a tight loop.

Threads: The low-priority thread starts execution, spawns the high-priority thread, and then continues its own workload. The high-priority thread begins running as soon as it is created and immediately preempts the lower-priority thread, as expected under `SCHED_FIFO`. Both threads execute the same CPU-burning routine three times.

Execution Flow:

- Low-priority thread starts execution and calls `burn_cpu()`.
- Low-priority thread creates the high-priority thread.
- High-priority thread begins execution immediately and preempts the low-priority thread.
- High-priority thread runs to completion, executing `burn_cpu()` three times.
- Low-priority thread resumes only after the high-priority thread finishes.

SCHED_FIFO Behavior: Under this policy, the highest-priority thread on a CPU runs uninterrupted until it blocks, yields, or finishes. Lower-priority threads are preempted immediately once a higher-priority thread becomes runnable. There is no time-slicing or automatic yielding.

Observed Behavior: Despite being created first, the low-priority thread is preempted almost immediately once the high-priority thread starts. Since the high-priority thread does not yield or sleep, it runs to completion, delaying the low-priority thread's execution entirely.

Mitigation: To observe both threads running, introduce a `sleep()` before creating the high-priority thread or use `sched_yield()` inside `burn_cpu()` to allow voluntary rescheduling.

4 Explanation of mixed.c

Overview: This program demonstrates the creation of threads with different scheduling policies in a Linux environment. The threads are pinned to CPU 0 and run a CPU-intensive task. Three threads are created with the following scheduling policies: `SCHED_OTHER`, `SCHED_RR`, and `SCHED_FIFO`. Each thread runs a loop that prints its status and performs some work in a tight loop.

Threads: Three threads are created with the following properties: - `Thread 0` uses the default `SCHED_OTHER` policy. - `Thread 1` uses the `SCHED_RR` policy with a priority of 10. - `Thread 2` uses the `SCHED_FIFO` policy with a priority of 20. Each thread runs a CPU-intensive routine in a loop, printing its ID, policy, priority, and iteration number.

Execution Flow:

- Each thread is pinned to CPU 0 using `pthread_attr_setaffinity_np()`.
- `pthread_attr_setschedpolicy()` is used to set the scheduling policy for each thread.
- `pthread_attr_setschedparam()` is used to set the priority for the `SCHED_RR` and `SCHED_FIFO` threads.
- Threads are created using `pthread_create()`, with the scheduling policy and affinity set beforehand.
- Each thread runs the `thread_func()` function, which prints its execution status and runs a CPU-intensive loop.

SCHED_FIFO: A real-time, first-in-first-out policy where threads run to completion unless they yield or block. The highest-priority thread is always given the CPU.
SCHED_RR: A real-time, round-robin policy where threads are given the CPU for a fixed time slice before being preempted in favor of other threads of the same priority.
SCHED_OTHER: The default scheduling policy where threads are scheduled based on time-sharing and priority levels, with time slices allocated by the system.

Observed Behavior: **Thread 0** with `SCHED_OTHER` will be scheduled as a regular process and may be preempted by other processes. **Thread 1** with `SCHED_RR` will receive time slices based on its priority (10) and will be preempted after its time slice expires. **Thread 2** with `SCHED_FIFO` will run without being preempted by other threads of lower priority, monopolizing the CPU if it doesn't yield.

Mitigation: No explicit mitigation is required, but the threads' execution will follow the priority rules of their scheduling policies, and depending on the system load, the `SCHED_FIFO` thread may monopolize the CPU unless yielding or blocking occurs.

5 Explanation of nice.c

Overview: This program demonstrates CPU affinity, process priority manipulation, and forking in Linux. It creates two child processes with different priorities and runs CPU-bound tasks on them to observe their behavior.

Execution Flow:

- The main process sets its CPU affinity to CPU 0 using `sched_setaffinity()`.
- The main process forks two child processes:
  - The first child runs with the default priority (nice value 0) and executes the CPU-bound task.
  - The second child runs with a higher nice value (19) and executes the same CPU-bound task.
- The parent process waits for 1 second and then prints the PIDs of the two child processes.
- After a 10-second delay, the parent kills both child processes using `kill()` and waits for their termination with `wait()`.

Functionality: The function `cpu_bound_task()` simulates a CPU-intensive task by running an infinite loop that increments a counter and prints progress messages. This task is used in both child processes to demonstrate how different priorities affect execution.

Key Concepts:

- **Priority and Nice Value:** The first child runs with the default priority (nice value 0). The second child has a higher nice value (19), lowering its priority and making it run less frequently than the first child.
- **Forking and Process Control:** The program demonstrates `fork()` to create two child processes. The parent monitors the execution, kills the children after a while, and waits for their termination.

Expected Behavior: The child with the higher nice value (lower priority) will be scheduled less frequently than the default-priority child. Both processes run indefinitely, printing progress messages, until terminated by the parent process.

6 Explanation of rr_prio.c.c

Overview: This program demonstrates thread priorities and CPU affinity with pthreads. It creates two threads, both running CPU-bound tasks with round-robin scheduling. The low-priority thread spawns a high-priority thread, and both threads execute tasks on CPU 0.

Execution Flow:

- The main thread pins itself to CPU 0 using `sched_setaffinity()`.
- The main thread creates a low-priority thread using `pthread_create()`.
- Inside the low-priority thread:
  - The low-priority thread creates a high-priority thread with `pthread_create()`.
  - Both threads run CPU-bound tasks.
- The low-priority thread waits for the high-priority thread to finish using `pthread_join()`.

Functionality: The `burn_cpu()` function simulates a CPU-intensive task, running an infinite loop and printing progress messages. Both threads use this function to simulate CPU-bound tasks, and the high-priority thread, if higher in priority, can preempt the low-priority thread.

Key Concepts:

- **Thread Priority:** Both threads use the same priority in this case, leading to round-robin scheduling. However, if the priorities were unequal, the higher-priority thread would preempt the lower-priority one, potentially causing starvation.
- **CPU Affinity:** Threads are pinned to CPU 0 to ensure they run exclusively on the same CPU core using `pthread_setaffinity_np()` and `sched_setaffinity()`.
- **Preemption and Scheduling:** With `SCHED_RR`, both threads share CPU time. However, unequal priorities would lead to the higher-priority thread preempting the lower one, potentially causing the lower-priority thread to starve.