
🧩 Interactive Activity: *"Signal Relay Race with a Stopwatch"*

Concept: Students will simulate a “relay race” where a program acts like a runner passing signals back and forth, while `clock_gettime` and `timespec` measure how long each step takes. The activity is hands-on, with each new feature unlocking another layer of timing detail.

Step 1: Starting the Stopwatch

- Introduce `clock_gettime` and `struct timespec`.
- Activity: Students write a small function `print_time_diff(start, end)` that shows elapsed time in nanoseconds.
- Interactive element: Have them measure *just* how long a simple function call takes.
  - Compare with and without `-O0` vs `-O2` compiler optimizations.
  - Prompt: *“Why does optimization change timing of function calls?”*

Step 2: Signal Relay Setup

- Introduce signals, `sigaction`, and `sigset_t`.
- Students set up a handler that prints “Signal received!” and records `clock_gettime` into a global `timespec`.
- Activity: Start a timer before sending a signal with `kill(getpid(), SIGUSR1)` and stop it in the handler.
- Goal: Measure round-trip latency of delivering a signal.

Step 3: Waiting for the Baton

- Introduce `pause()` or `sigwait()` for waiting until a signal arrives.
- Students now send a signal from *another* process (or child process).
- Activity:
  - Parent process records start time and sends a signal to the child.
  - Child handles it, then sends a signal back.
  - Parent records stop time when it gets the return signal.
- Result: Students measure end-to-end signal latency.

Step 4: Optimizing the Race

- Compare results when handler work is heavy vs minimal (e.g., printing vs only recording `clock_gettime`).
- Run the program with `-O0` and `-O3` and compare measured times.
- Students discuss: *How do compiler optimizations affect system call overhead vs pure function calls?*

Step 5: Classroom Game Version

To make it more interactive:

- Each student (or small group) represents a process.
- A “signal” is represented by passing a baton/ball.
- A “timer” student starts/stops a stopwatch when a signal leaves/returns.
- Different rules simulate optimizations:
  - No talking = optimized (less overhead).

    - Adding extra chatter before passing the baton = unoptimized (more work in the handler).

This physical activity mirrors the program while students code the digital version in parallel.

👉 Outcome: By the end, students will have:

- Practiced using `timespec` and `clock_gettime`.
- Measured real-world timing of function calls, compiler optimization, signal delivery, and handler execution.
- Understood why system-level timing isn't always predictable.