Developing a coding contest platform

An **online judge** is a program used in competitive programming contests which contestants submit code to. Its job is to evaluate submissions against a series of test cases and performance criteria.

Since the purpose of an online judge is literally to run user-submitted code, it is imperative that we sandbox the code submissions when running them, as otherwise the system could be exploited by arbitrary **remote code execution** (RCE).

It is also important that the judge backend is implemented efficiently — test cases can be millions of characters long and the problems may have naïve solutions that far exceed the time limit. To avoid slow submissions from overwhelming the judge, it must:

1. cut submissions off once they exceed preset time and memory limits
2. prevent a single submission from blocking other submissions from running

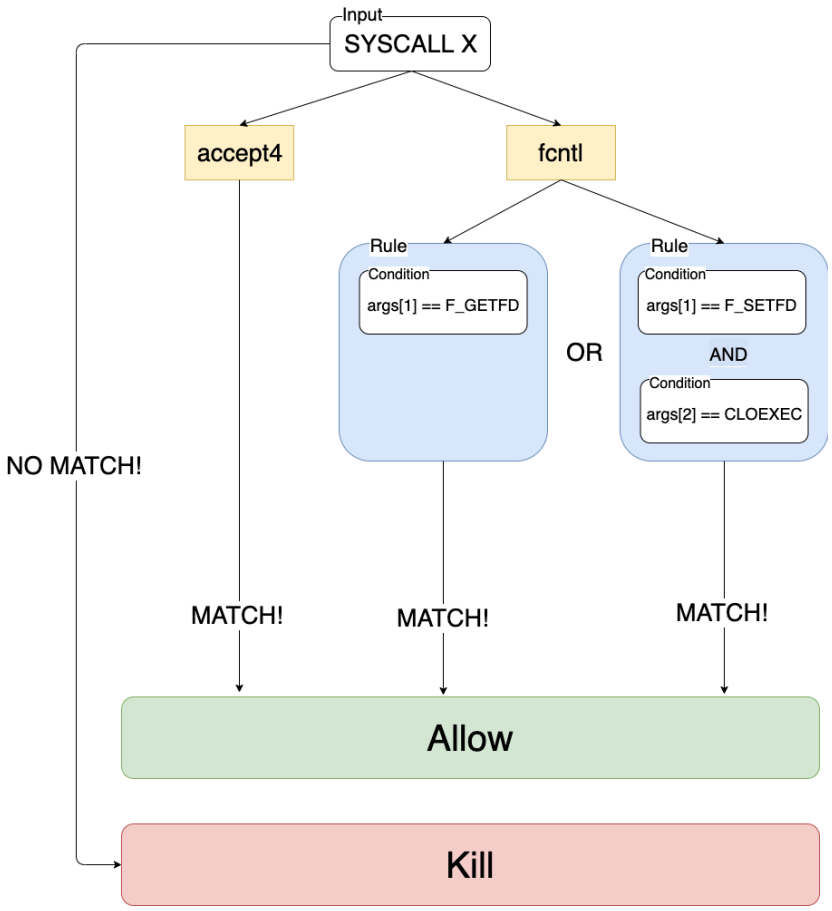
I developed the judge backend in Rust. This choice was motivated by Rust's strength at systems programming, concurrency and security, making it a perfect fit for this project.

`seccomp-bpf` API

One attack vector against the judge backend is through **system calls** or syscalls. The platform my judge runs on, Linux, has around 450 syscalls, many of which may be dangerous or undesirable as they give unfair advantages, affect other submissions or the state of the judge itself.

`seccomp-bpf` (SECure COMPuting with Filters) is a Linux API for filtering the syscalls programs are allowed to invoke. It uses the Berkeley Packet Filter (BPF) as its mechanism for filtering syscalls.

I used the `seccompiler` Rust crate to build and load `seccomp` filters. This allowed me to whitelist a series of allowed syscalls that judged programs would need to use, and blacklist the rest by returning a permission error when invoking a filtered syscall.



Terminating ("killing") programs that invoke restricted syscalls is another option offered by `seccompiler`. (source)

Disclaimer: as I developed my own security solution and am not a security professional, I am running the judge backend containerized with Docker, just in case.

`rlimit` and `wait4` API

Contest rules typically dictate a time and memory usage limit on submissions, to encourage algorithmically-efficient solutions and conserve judge resources. A common set of resource limits are:

- 1 second of CPU time
- 512 MB of memory

The POSIX `rlimit` API can be used to set soft and hard limits on resources that programs are allowed to use, including CPU time (`RLIMIT_CPU`) and memory usage (`RLIMIT_DATA`). When CPU limits are exceeded, the process is killed with `SIGXCPU` (CPU limit exceeded), and when memory limits are exceeded, `brk` and related allocation syscalls fail with `ENOMEM`.

I also wanted the judge to report the resource usage of submissions to users, to allow them to understand and improve the performance of their solutions. This led me to use the `wait4` syscall, which is similar to that for waiting for a child process to exit, but it also returns the process' resource usage.

Compilation and judging configuration

The judge server is configured in TOML. The configuration files are shared with the frontend as well, which allows it to report the resource limits and available languages to the user.

Example of a judge configuration

```toml
[judge]
skip-count = 3

[judge.resource-limits]
cpu = 1 # seconds
cpu-tolerance = 0.1 # seconds
memory = 512_000_000 # bytes
memory-tolerance = 1000 # bytes

[[judge.languages]]
name = "C 99"
filename = "submission.c"
compile = ["gcc", "./submission.c", "-std=c99", "-O3", "-o", "./submission"]
run = ["./submission"]

[[judge.languages]]
name = "C++ 17"
filename = "submission.cpp"
compile = ["g++", "./submission.cpp", "-std=c++17", "-O3", "-o",
"./submission"]
run = ["./submission"]

[[judge.languages]]
name = "Python 3"
filename = "submission.py"
run = ["python3", "./submission.py"]

[scoring]
answer-score = 100
task-score = 100
subtask-score = 50
test-score = 5
```

The following can be configured:

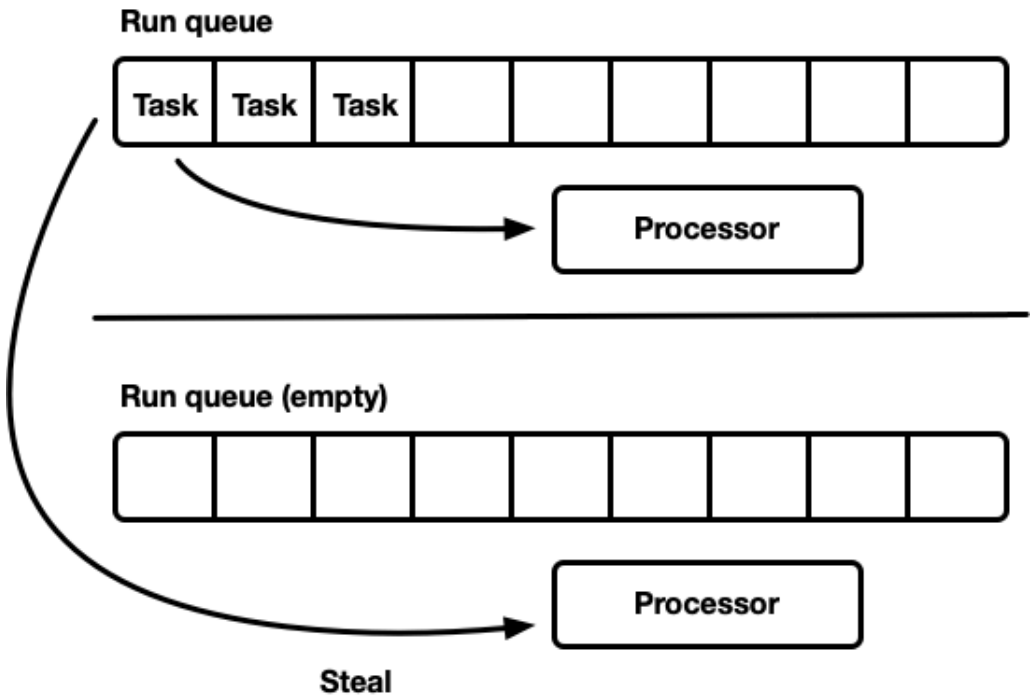
- `skip-count`: how many Time Limit Exceeded / Memory Limit Exceeded verdicts to tolerate before skipping a subtask
- Resource limits
- Language configurations — C/C++ and Python are shown here

As Python is interpreted, there is no need for a `compile` command.

Parallel execution

If submissions were to be judged sequentially, it would take quite long to completely judge a submission to a task with a lot of tests. This is even worse if each run takes up the whole time limit — for example, with a CPU time limit of 1 second and 100 tests serial judging would take 100 seconds.

Parallelism is a good way to increase judging throughput. I used the `rayon` crate, which provides a parallel iteration mechanism, to run submissions on test cases in parallel.



Work stealing scheduler (source)

`rayon` uses a **work-stealing scheduler**, which maintains separate task queues for each processor and allows idle processors to steal tasks from busier processors.

On my 16-core laptop, this ensures that judging a single submission always finishes within a couple of seconds, given a reasonable number of test cases. As resource limits measure CPU time and not real time, each test run gets a fair amount of time to execute.

Contest structure

A contest contains a problem set of independent tasks. A task is composed of subtasks, typically with different constraints on what the test inputs might contain each. Each subtask is then composed of a list of test cases, which specify an input and expected output.

Judge verdicts

The judge assigns a **verdict** to each test, subtask and task, which is one of:

- **Accepted**
- **Skipped**
- **Memory Limit Exceeded**
- **Time Limit Exceeded**
- **Wrong Answer**
- **Runtime Error**
- **Compile Error**

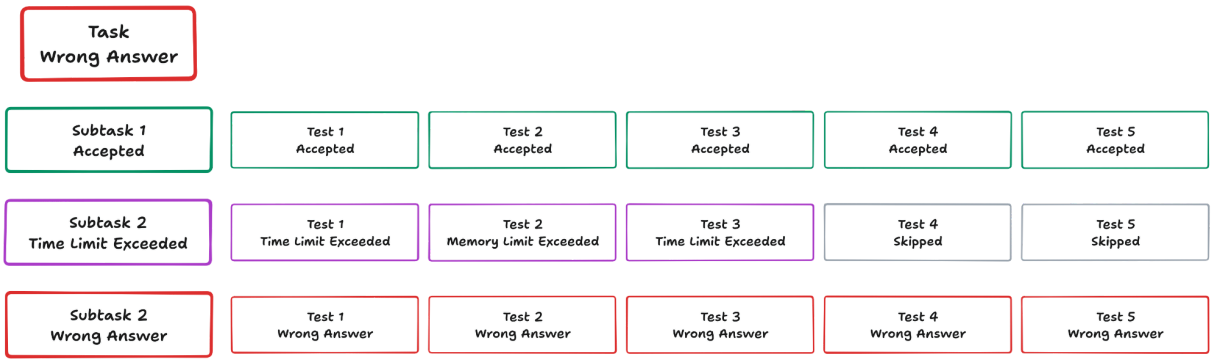
The verdicts are listed in order from best to worst. If two tests in a subtask have different verdicts, the worst one is taken as the subtask verdict.

Scoring

The judge configuration also includes a scoring section which defines a score for passing a single test, an entire subtask, or an entire task. However, to get the score from tests, the subtask must be passed as not all tests are guaranteed to run each time.

Subtask skipping

When multiple submissions occur at once, contention for resources is extremely high. I noticed that some judge programs I've used before will skip a subtask if the resource limits are exceeded too often, so I decided to implement this.



In the above example, subtask 2 is skipped and prematurely assigned the Time Limit Exceeded verdict as 3 tests exceeded resource limits and the remaining 2 were skipped. The threshold for skipping can be configured.

This is implemented using a `tokio::sync::watch` channel, which is a synchronization channel that retains the last value sent. This allows me to easily keep track of how many tests exceeded resource limits and use task cancellation to skip the rest of the tasks when the threshold is exceeded.

Rate limiting with semaphores

When performing load testing, I found that the judge server would still crash when processing too many concurrent submissions. This was because each test run would open multiple files, and the OS sets a limit (`ulimit`) on the number of simultaneously open files per user.

I fixed this by using a synchronization primitive known as a **semaphore**, used to control and limit access to a shared resource. A global semaphore is defined with a maximum number of concurrent submissions, and the server attempts to acquire a semaphore **permit** each time a submission is received.

If a permit is obtained, then the task processes the submission and returns the permit when finished. If no permits are currently available, the task has to yield and wait until one becomes available.

Streaming results with server-sent events (SSE)

When competing in the Hong Kong Olympiad in Informatics, it was often frustrating to have to wait for the judge to finish before getting any feedback, especially if I had just forgotten to print the answer out. Therefore, I thought it would be a good feature to stream judging results back from the judge server in real time.

```
enum Message {
    /// Queued for submission
    Queued { tests: u32 },
    /// Indicates that the compile step has been started (optional)
    Compiling,
    /// Provides compiler warnings and errors (optional)
    Compiled { exit_code: i32, stderr: String },
    /// Judging status
    Judging { verdict: Verdict },
    /// Tests were skipped due to exceeding resource usage
    Skipping { estimated_count: u32 },
    /// Judging completed successfully (final)
    Done { report: Report },
    /// The judge experienced an internal error (final)
```

```
    Error { reason: String },
}
```

I used **server-sent events (SSE)** to achieve this, developing my own JSON message protocol to transmit status updates to the judge. I also used the schemars crate to generate a JSON Schema that could be used by the frontend through generating TypeScript types, ensuring end-to-end type safety.