## Submission Checklist

Please make sure you go through this list before submitting a patch. The rules aren't hard and fast, but mostly adhering to them will make for quicker mergings.

- [ ] Does your PR add support for a new kernel API? For example, supporting a new syscall. If so, your patch should include at least one new test for the API. This is usually pretty easy. See `$rr/src/test` for examples.

- [ ] Did you run the rr test suite (including your new tests, if any), and pass all the tests? `make -C $objdir check`. Unfortunately, rr doesn't have automated infrastructure that can run the tests yet, so developers have to run them locally.

- [ ] If you created new files for your PR, did you `git add` them? Habitually (or with a script or push hook) checking `git status` is a good habit to acquire.

- [ ] If you changed the trace layout or format, did you bump `TRACE\_VERSION`?

- [ ] If you added new command-line parameters, did you update `print\_usage()` to document them?

- [ ] Does your PR apply cleanly on top of upstream/master HEAD? It's dangerous to have someone else sort out your merge conflicts, so just don't do it. Best of all is to have a PR \*rebased\* on top of upstream/master HEAD, so that the merge is simply a fast-forward.

- [ ] If your PR includes multiple changesets, do they all (i) build cleanly in sequence; (ii) pass all tests in sequence? This is important for bisecting over commit history.

- [ ] If your PR is a very large-scale change (for example, a rewrite in Rust to use the visitor pattern), did you discuss the proposed changes in an issue or the mailing list? It's hard to review large patches that just fall in ones lap. It's much easier to discuss the important changes at a high level and then approach the patch knowing what's important and what's not.

- [ ] If your PR is large or includes many changesets, would it have been possible to break the changes into a series of smaller PRs? For example, it's hard to review a big patch that, say, fixes whitespace errors in a file along with a one-line, important, bug fix. It's much easier to review one PR that fixes whitespace (which can just be skimmed), and then review another PR that makes the one-line bug fix (which would be scrutinized more). This approach is also better for the patch author in that it usually allows the work to land faster, and reduces the burden of continually un-bit-rotting large, trivial, changes.

- [ ] Did you check your code is formatted correctly? It's easiest to run `scripts/reformat.sh` on each commit.

## Coding Guidelines

rr uses assertions heavily, for code documentation, for automated checking that the code matches the documentation, and to improve the power of automated tests. Assertions are turned on in release builds. Whenever you depend on an invariant not immediately obvious, consider adding assertions to check it.

rr ships with debugging enabled and compiler optimizations disabled for the rr process itself. That's because rr performance almost always depends on algorithmic issues --- minimizing the number of system calls, and especially, minimizing the number of context switches between the tracees and the rr process --- much more than the performance of the code running in the rr process. For the same reason, rr-process code should be as simple as possible even if that's less efficient. To some extent, once we're running code in the rr process, we've already lost performance-wise. OTOH we do enable optimizations in `preload.c` because that runs in tracees.

## Coding Style

Put braces around all statement blocks, even one-line `if` bodies etc.

All C++ declarations are in the `rr` namespace.

All C++ types are in CamelCase; all C types are underscore\_names.