Stratis Coding Style Guidelines

Last modified: 01/10/2019

Contents

Asking Questions and Making Changes to this Document

This document can be found in the stratis-docs repo, and is written using LyX 2.3.3. Please ask any questions by opening an issue, and propose changes as pull requests.

1 Introduction

A consistent style helps legibility and saves developers from needing to think about issues that are not really important.

For both Rust and Python, there are standard coding conventions, as well as tools that help keep code consistent with these conventions that we will try to use as much as possible. There are also some additional project-specific style guidelines, which we will document here to help developers new to the project.

2 Style (Rust)

1. All new files must start with the 3-line license as a comment.
2. Ordering of "use" declarations
   (a) "use" declarations within a module (such as a source file) should be grouped into up to four sections:

i. declarations from "std"
ii. declarations from other external dependencies except for devicemapper
iii. dependencies owned by the Stratis project
iv. declarations from other modules in the same crate

as needed, separated by blank lines.

(b) Each section should be a single "use" directive, except for declarations from external dependencies, which require a separate use directive for each crate.

3. Use "expect()" instead of "unwrap()" to unwrap things that should always succeed. "unwrap()" is not allowed in new code.

   (a) "expect()" text should say what "should never fail" condition was not met to trigger it.

4. When implementing a trait, the methods should be ordered as they are ordered in the trait definition.

2.1 Conventions (stratisd)

1. Stratisd has many instances of ranges of storage being allocated from other things. This can become confusing. In naming methods that relate to this, use the following conventions:

   (a) A method that attempts to allocate some number of unit of storage from an allocation pool container should be called "alloc" if the method returns an error if the entire amount requested is not available. If a method returns a partial allocation if the entire amount is not available, it should be called "request".

   (b) A method returning the total amount of storage units (i.e. sectors, DataBlocks, or similar) in an allocation pool should be called "size".

   (c) A method returning the total *usable* amount of storage units – the total number minus overhead (if any) – in an allocation pool should be called "usable" or "usable_size".

   (d) A method returning the number of previously-unallocated storage units should be called "available" or "avail_size".

   (e) A method returning the number of allocated storage units should be called "allocated".

3 Style (Python)

1. Use list comprehensions instead of for-loops when possible. In places where they can be used, comprehensions more clearly communicate the code's intent to filter and/or map a list into another list. Same goes for generator, dict, and set comprehensions.

2. Do not use boolean overloading in conditionals unless the type of the object is unknown. If the type of the object is known, use an explicit comparison such as "is None" or "== []". This way, if an object of an unexpected type arrives, the comparison will generate an error instead of a latent bug.

3. Use os.path methods when assembling or deconstructing paths. We are not concerned with cross-platform usage, so this is for clarity, to make it clear that it's a path, not a string.