LEHD SVN to Git transition

Current structure

Currently, a single SVN repository is used, with the following root structure:

```
/branches
/trunk
/releases
```

Key concept

A "production process" (PROCESS) is a defined set of code, producing a particular (internal or public) data product. For instance, the "production process" `EHF` ingests UI data, parameters, and outputs a set of files called `EHF` (employment history files).

Production processes are PRIMARY objects of LEHD coding structures. They appear as SECOND-LEVEL directories under `trunk` and `releases`, and may appear in combinations under `branches`.

Trunk

Under trunk are reviewed, functional code packages. The trunk structure has

```
/trunk/PROCESS
```

e.g.

```
/trunk/
   ehf/
   ecf/
   icf/
   ...
```

Releases

At defined points (time or trigger), a release is made. Production code is always run from a defined release, never from `trunk`. Release number follow LEHD release numbering policy, with `x.y.z` and suffixes. Non-suffixed releases are "official" releases, suffixed releases (-eg. "-testing", "-ticket345") are not to be used for production. The release structure has

```
/releases/PROCESS/x.y.z[-suffix]
```

Branches

For every issue, a branch is created. It may include one or more PROCESSes, with no particular combination hard-coded or prescribed. A branch structure should mirror the (partial) trunk structure, though. It is usually identified by programmer ID, ticket number, but no hard nomenaclature is enforced. It is typically populated from `trunk` but may be populated by a specified `release` (rare). Structure is

```
/branches/[branch name]/PROCESSes
```

e.g.

```
/branches/
   ticket-345/
      ehf/
      ecf/
      icf/
```

for a branch addressing ticket 345, which has issues identified in PROCESS `ehf`, `ecf`, `icf` that need to be solved together.

Testing `environments` can be set up in the production system, and be populated from branches.

Mapping SVN to Production Code

In production, code is organized in the same structure as TRUNK:

```
/trunk/
   ehf/
   ecf/
   icf/
   ...
```

Individual PROCESSes are, however, checked out from specific releases, as recorded in the metadata. At present the metadata is post-updated, i.e., a gatekeeper executes a command updating a PROCESS to a specific release, and this is recorded ex-post in the metadata. However, at any point in time during production processing, metadata and code-on-disk point to the same release number.

Future Development Process

(FILL IN with BEST PRACTICES)

Simple Development

A simple development cycle affects only one PROCESS.

- Two options (depending on authentication configuration):

    - Create a new branch within repository for PROCESS.
        - may not be possible with specific authentication setups
    - Fork the repository for PROCESS
        - may cause conflicts if developer working on multiple issue all of which affect this PROCESS (can only fork once)
        - "forking" only defined for Github product?
- Do development
- Send pull request to gatekeeper.

Complex development

Complex development affects multiple PROCESS. A different way of framing this is that there is a key issue, with multiple dependencies.

One way to solve:

- Proceed as for Simple Development/Branches
- Create a new repo for testing purposes (`ticket-345`)
- Record as sub-modules all dependencies, referencing the newly created branches
- Do development
- Issue multiple pull requests, referencing a "milestone" ticket or similar.

Future Structure

Each PROCESS exists in a network of upstream and downstream dependencies. These should be maintained given the current production process/workflow.

Production Code

The most current production code is maintained as a "virtual repository" with multiple sub-modules.

```
/production-code/
   ehf/ (sub-module, pointing at release x1.y1.z1)
   ecf/ (sub-module, pointing at release x2.y2.z2)
   icf/ (sub-module, pointing at release x3.y3.z3)
```

- Contains only sub-modules
- By cloning with recursive updating of sub-modules, specific versions are pulled from their respective repositories, fixed at specific tags/releases.
- The structure replicates the current Production code structure described above.
- Updating this repository is done by the gatekeeper, updating the locked releases to whatever the most current/authorized release is.
- Official development versions can be maintained by "branching" the master branch of `production-code` and updating the sub-modules to specific tags of the PROCESS repository.

Repositories

Official Repositories

The main organization ("LEHD Production") maintains a number of repositories, **one for each PROCESS**.

```
https://git.census.org/lehdproduction/
     ehf/
     ecf/
     icf/
```

Each repository has the same generic structure:

- Directories under the `root` (e.g., `https://git.census.org/lehdproduction/ehf/`) are grandfathered in from the current SVN trunk.
- Tags (all) and releases (depends on git vendor) reflect officially designated versions of the code base for consumption by other parts of the system.
- As before, Production Code MUST come from releases, not from branch `master`.

Development Structure

Developers have their own repositories for development only. They can fork/clone individual PROCESS repos, or the `production-code` repo. All work is regularly committed to their own repositories, and can be monitored in the managed code repository.

(OPTIONAL) A convention to rename the repository to reflect current branching terminology can be created.

(OPTIONAL, SUGGESTED) Requiring structured commit messages to identify issues, milestones, etc. should help to maintain tracking.

Migrating Repositories

- The full history of releases should be preserved.
- Preserving the full development history is optional.
- Each PROCESS is migrated into its own repository, including all related releases.

A simple draft pseudo-code might look like this:

```
PROCESS=ehf
SVNURL=https://redmine/svn/lehd/
RELEASES=$(svn ls $SVNURL/releases/$PROCESS)

localgit=$(mktemp -d)
cd $localgit
git init
for release in $RELEASES
do
  vnum=$(echo $release | lastcomponent)
  svn export $release
  git add *
  # check that this processes deleted files!
  git commit -m "Recording $PROCESS release $vnum" -a
  git tag -a -m "$PROCESS release $vnum" $vnum
```

```
  rm *
done
# finalize with any unreleased trunk code
svn export $SVNURL/trunk/$PROCESS
git add *
git commit -m "Last trunk for $PROCESS on SVN" -a
```

followed by a push to the official LEHD Git server.

Branches will need to follow a manual process if in active use (inactive branches, see "Historical Archive").

- Apply the appropriate development strategy (see above)
- (Optional) Replay the branches' SVN history on top of the fork/branch (`for history in (commits); do svn export; git add *; git commit; done` similar to above TO BE REFINED)
- Export the last known state of the branch on top of the fork/branch, commit.

Historical Archive

The current SVN repository should be frozen (read-only access). Options for maintenance are two-fold:

- continue maintaining a SVN repo (Apache + SVN or other), allowing for sparse storage of the entire history (15,000+ commits) of the repository. This should also work seamlessly in order to maintain a transitional copy of `Redmine`.
- fully expand the SVN repo (instantiate each commit), and store as simple HTML files. This is likely to be storage-intensive (TBD), but has no future dependencies or maintenance. May work seemlessly (depends on crawler configuration). May work with `httrack` or others. The same could be done for `Redmine`.