Archiving and Packaging
A Survey

Tim Kientzle
kientzle@freebsd.org
http://people.freebsd.org/~kientzle/

Or:
How I Accidentally Rewrote Tar

Outline

- A Story
- Libarchive
- Bsdtar and other tools
- Packaging: Principles and Concepts
- Towards libpkg

What am I talking about?

- Libarchive: Modular library for reading and writing “streaming archive formats”: tar.gz, cpio, zip, iso9660, some others.
- Bsdtar: Implementation of “tar” program built on libarchive. Comparable to GNU tar in overall functionality.
- FreeBSD 5.3: “bsdtar”, “gtar”, “tar” is alias for “gtar”.
- FreeBSD 6: “tar” is alias for “bsdtar”
- FreeBSD 7: “gtar” goes away

How I Got Here

A Story

- ~1998: Teaching FreeBSD classes
- Lessons for me: installer sucks
- New installer is a BIG job: try building one small component (package library)
- ~2003-2004: Unemployed
  - Prototyped a new pkg_add
  - Isolated archive management: libarchive
  - Test harness grew into bsdtar

What's wrong with pkg_add?

- Slow: Scans entire archive 4 times
  - Extract +CONTENTS packing list
  - Extracts files to temp directory
  - Archives temp directory
  - De-archives into final location
- Can't use it to build new tools.
- We need libpkg.

What if pkg_add didn't fork tar?

- Extract +CONTENTS (always first) into memory
- Use +CONTENTS to drive extraction directly into final location.
- Result: 3-4 times speedup.
- I've prototyped this, it works.
- But pkg_add is a lot more than just extracting files...

Towards reusable components

- Libarchive: reads/writes streaming archives
- Libpkg: higher-level package operations

Libarchive

What is libarchive?

- Static and shared library, programming headers.
- Writes: tar, cpio, shar (optional gzip, bzip2 compression)
- Reads: tar, cpio, zip, iso9660 (all with optional compress, gzip, bzip2 compression)
- Portable to FreeBSD, Linux, Mac OS, others.

Why libarchive?

- Mark Roth's libtar: Good, but heavily oriented around tar command-line ops. (Hard to extract to memory, modify items as they are archived, etc.)
- Other “multi-format” archiving libraries are seek-based: Can't read/write tapes, network connections, stdio, etc.
- Libarchive was originally tar-only, but I realized that it was easy to generalize to a large class of archiving formats.

Libarchive API Principles

- Stream oriented
- Allow client to drive archive/extraction
- Be smart, but not too smart
  - Format auto-detect
  - No threads in library, no forking
- Support standards
- API and ABI stability (no structures)
- Minimize link pollution

Minimize Link Pollution

- Avoid the printf() mistake
- Archive read and write are completely independent
- Layering: Higher layers use public APIs of lower layers
- archive_read_support_XXX()
- archive_write_set_XXX()
- Remember: libarchive was partly targeted for use in installer. Size matters!

Link Pollution Minimized

- 70k statically linked minitar (tar read and extract only, no decompression)[1]
- Smaller static binary than:

```
int main()
{
    printf(“hello, world”);
    return 0;
}
```

[1]In FreeBSD 5.3. 6.1 linker doesn't like me.

Libarchive API Tour

- Read
- Extract
- Write
- archive_entry
- Utility

General Usage

- Create a “struct archive *” (archive object)
- Set parameters
- Open archive
- Read/write archive entries
- Close archive
- Dispose of object

Overall Structure

```
struct archive *a;
struct archive_entry *entry;
a = archive_read_new();
archive_read_support_compression_gzip(a);
archive_read_support_format_tar(a);
archive_read_open_XXX(a,...);
while (archive_read_next_header(a, &entry) ==
   ARCHIVE_OK) {
    printf("%s\n", archive_entry_pathname(entry));
    archive_read_data_skip(a);
}
archive_read_finish(a);
```

Create Object

Set Parameters

Open Archive

Iterate over contents

Close and Dispose

Prefixes Indicate API

```
struct archive *a;
struct archive_entry *entry;
a = archive_read_new();
archive_read_support_compression_gzip(a);
archive_read_support_format_tar(a);
archive_read_open_XXX(a,...);
while (archive_read_next_header(a, &entry) ==
   ARCHIVE_OK) {
    printf("%s\n", archive_entry_pathname(entry));
    archive_read_data_skip(a);
}
archive_read_finish(a);
```

Usually: archive * is first arg

```
struct archive *a;
struct archive_entry *entry;
a = archive_read_new();
archive_read_support_compression_gzip(a);
archive_read_support_format_tar(a);
archive_read_open_XXX(a,...);
while (archive_read_next_header(a, &entry) ==
   ARCHIVE_OK) {
    printf("%s\n", archive_entry_pathname(entry));
    archive_read_data_skip(a);
}
archive_read_finish(a);
```

Read API

- Object Creation
- Parameter setup
  - “set” calls force values
  - “support” calls enable auto-detect
- Open Archive
  - Core “open” method accepts callback pointers for open/read/skip/close
  - Library provides “open_filename”, “open_fd”, “open_FILE”, “open_memory” for convenience

Read API (cont)

- Iterator model
  - Each call to “read_next_header()” gives header for next entry
  - Header returned as archive_entry object
  - Data can be read after header

Inside Auto-Detect

- read_support_format_tar(a) registers with read core:
  - Header read
  - Data read
  - Bidder (taster)
- Read core has no functional dependencies on tar code
- If you don't call “support_tar()”, no tar code is linked
- Bid value is approx # bits checked

Read I/O Layering

- Three layers:
  - Client read() callback
  - Compression layer
  - Format layer
- Peek/consume I/O
  - Each layer returns pointer/count
  - Separate “consume” advances file position
  - Best case: no copying through entire library
- Future: mmap(), async I/O

Libarchive extract() API

- Creates objects on disk from archive_entry
  - Creates intermediate dirs, device nodes, links
  - Invokes archive_read_data(), but otherwise separate from read core
- Extraction holds a surprising amount of state
  - Permission/ownership updates are deferred
  - Caches GID/UID lookups
  - Link resolution (cpio-only)

Correctly Restoring Permissions

- Some ugly cases:
  - Non-writable directories
  - Hard links to privileged files
  - Restoring directory mtimes
  - Mixed ownership
- Remember: tar does not promise file ordering! (tar -u)
- Solution: Certain permissions are restored only at archive close

Libarchive Write API

- Write core
  - Two-phase: header, then data
  - Note: Header must include size
- No “write file” layer (yet?)
- Client callbacks write bytes to archive

Writing one Entry

```
entry = archive_entry_new();
archive_entry_copy_stat(entry, &st);
archive_entry_set_pathname(entry, filename);
archive_write_header(a, entry);
fd = open(filename, O_RDONLY);
len = read(fd, buff, sizeof(buff));
while ( len > 0 ) {
        archive_write_data(a, buff, len);
        len = read(fd, buff, sizeof(buff));
}
archive_entry_free(entry);
```

Libarchive Write Internals

- Simpler than read.
- One source file per format, etc.
- Write blocking is a little tricky

Archive_entry

- Represents “header” of an entry in the archive
- Think: “struct stat” on steroids
  - Filename
  - Linkname
  - File flags
  - ACLs
  - Implicit narrow/wide filename conversions
- Used both by read and write

Utility API

- Set/extract error messages
- Get format code, name
- Get compression code, name

Questions about Libarchive?

tar

Some things you probably didn't know:

- POSIX specified tar and cpio programs in 1988, but dropped them in 2001.
- “pax” utility (1993-) now defines tar & cpio formats.
- “Pax Interchange Format” (2001) extends “ustar”, which extends historical tar.
- Pax interchange format does (almost) everything you want.
- www.unix.org/single_unix_specification/

Pax Interchange Format

- Allows arbitrary key=value attributes to be attached to any entry.
  - Values are in UTF-8
  - Arbitrary lengths (up to 8GB total in theory)
- Standard attributes include arbitrary-size versions of standard fields (name, file size, time, uid, uname, etc).
- Vendor-specific extensions support ACLs, file flags, etc. (libarchive supports most 'star' keys, can support others).

Bsdtar and friends

- Started as test harness and second client for libarchive API checks (pkg_add prototype was first)
- Eventually grew into full-featured replacement for GNU tar.
- Supports most GNU tar options, reads gtar format, etc.
- Still needed: libarchive-based cpio, pax
- Special thanks: Kris Kennaway

Tar security

- Libarchive's two-phase permissions extract helps a lot.
- During restore, directories have restricted permissions.
- Other cases that bsdtar handles:
  - Absolute pathnames, .. components, symlink traversal
- Bsdtar prohibits all of these by default.
- -P option suppresses these checks.

Bsdtar vs GNU tar

- BSD license
- Full auto-detect
- Implements POSIX standards
- Multiple format support (ZIP, cpio, ISO9660)
- Reusable libarchive

- GPL
- Writes sparse files
- Multi-volume support
- RMT support
- Well-tested, reliable

Bsdtar vs star

- BSD license
- Full auto-detect
- Multiple format support (ZIP, cpio, ISO9660)
- Reusable libarchive

- GPL
- Writes sparse files
- Multi-volume, RMT support
- Fast
- Well-tested, reliable

Questions about bsdtar?

Packaging and libpkg

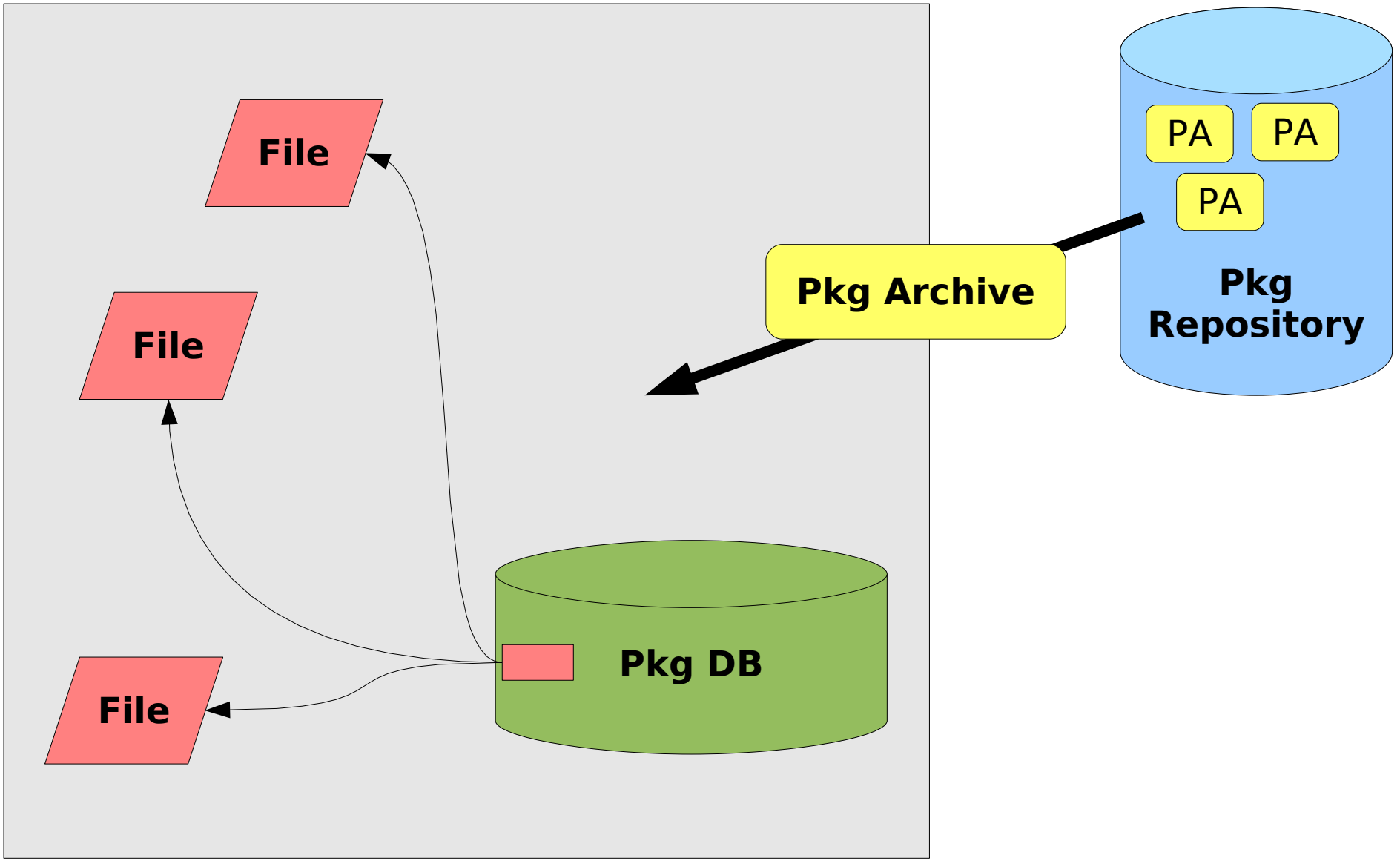
Towards libpkg

- Survey of overall package system
- Proposed libpkg architecture
- Status Report

Elements of a Package System

- “Package Archive” describes a group of files that can be installed onto a system (tar.gz or tar.bz2 file)
- “Package Repository” holds package archives (CD-ROM, HTTP or FTP site, etc.)
- “Package Database” tracks files on local system (/var/db/pkg)
- “Package” is a collection of files plus management information.

Package System



libpkg

- pkgdb: Keeps track of files and packages.
- Pkg: An object in the pkgdb. A pkg object describes files with attributes.
- pkg_repo: A connection to a repository
- pkg_archive: A tool for examining, extracting, and creating package archives
- pkg_manifest: list of files and attributes (with textual representation)

Questions

- Pkgdb: “What pkg contains this file?”
- Pkgdb: “Is pkg XYZ installed?”
- Pkg: “What files do you contain?”
- Pkg: “Please add/remove file ABC.”
- Pkg_repo: “Give me archive for XYZ.”
- Pkg_archive: “Give me manifest.”
- Pkg_manifest: “Tell me files/attributes, dependencies.”

pkg_add outline

- Contact pkg_repo
- Ask pkg_repo for file handle
- Create pkg_archive around file handle
- Extract and parse manifest
- Create package entry in pkgdb
- Iterate over pkg_archive contents
- Copy each item to disk/add to package

pkg_create

- Build new manifest (possibly from pkgdb entries, possibly from separate description)
- Create pkg_archive
- Write manifest to archive
- Write each file to archive

Other Utilities

- pkg_delete: Operation on pkgdb
- pkg_register: Create pkgdb entry from description of installed files
- pkg_check: Iterate over packages in pkgdb, check each file in each package (optionally: Enumerate files in /usr/local, identify files not in any package.)
- pkg_modify? Add/remove/rename single files in package, update pkgdb from files on disk, etc.

Problem: Dependencies

- “Flow-through” installation is nice.
- But: Definitive dependency info must come from manifest in archive.
- Problem: stalled download.
- Partial solution #1: Async streaming.
- Partial solution #2: Dependency info from pkg_repo. (Maybe incomplete?)
- Partial solution #3: Two-phase commit.

Possibility: Async Streaming

- Idea: Use threads (or forked processes) to separate install from download.
- Dependency handling can then defer the install without stalling the download.
- Minus: Requires disk space to store the package archive.
- Plus: Straightforward to implement.

Possibility: pkg_repo dependency info

- Idea: Ask pkg_repo (via INDEX file?) for (possibly incomplete) dependency information, install dependencies first.
- Minus: This complicates rollback.
- Minus: Not all repositories can support it (e.g., local NFS-mounted package dir)
- Minus: Incomplete information can reduce stalls, but false dependencies need to be rolled back?

Possibility: Two-phase commit

- Create “tentative” entries in pkg_db, extract files tentatively, finalize all at once.
- Model: Add file by asking package for file handle, package uses temp filename, then renames on commit.
- Plus: Simplifies package clients.
- Plus: Enables some nice tricks.
- Minus: More work to implement.

Problem: Conflicts

- Principle: Files conflict, not packages.
- If there is conflict, do we:
  - Skip entire package?
  - Skip single files?
  - Rename/move files?
- Libpkg should be agnostic about UI.
  - Some tools will want to know in advance.
  - Some tools will want to handle on-the-fly.

Problem: Rollback

- Reasons a single pkg_add can fail: dependencies, conflicts, failed downloads.
- Want to rollback everything together.
- Otherwise, pkg_add has to track a lot of information, possibility of stranded installs.
- Two-phase commit should make this easy.

Libpkg status

- Early design document on people.freebsd.org/~kientzle
- Basic pkg.h header.
- Skeletal implementations of key objects.
- Minimal pkg_add built on current implementation.
- Two-phase commit is in progress.

Miscellany: Directory Traversals

Dir Traversals: First Attempt

- Recursive opendir()
  - Opendir()
  - Visit and stat() each entry
  - Recurse if it's a directory
  - Closedir()
- Plus: Simple, handles wide trees
- Minus: Deep trees (file descriptors)

Dir Traversals: Second Attempt

- Recursive opendir() with pre-read
  - Opendir()
  - Read all entries into memory()
  - Closedir()
  - Visit and stat() each one
  - Recurse for directories
- Plus: Handles deep trees, hook for sorting
- Minus: Wide trees (memory)
- Fts(3) does this (but has API problems)

Dir Traversals: Third Attempt

- Lazy Descent
  - Opendir()
  - Visit and stat() each entry
  - Put directories on a work list
  - Closedir()
  - Visit next item on work list
- Plus: Deep trees, wide (files)
- Minus: Many subdirs (memory), order can be surprising
- tar/tree.c does this

Dir Traversals: Summary

	Recursive	**Fts(3)**	**tar/tree.c**
Deep	*Filehandles*	64k path	Yes
Many Files	Yes	*Memory*	Yes
Many Subdirs	Yes	*Memory*	*Memory*
Complexity	Simple	High	Medium

Archiving and Packaging
A Survey

Tim Kientzle
kientzle@freebsd.org
http://people.freebsd.org/~kientzle/