Journaled File LIBrary (libjf) tutorial

“from the ground up”

Christian Ferrari

tiian@users.sourceforge.net

Journaled File LIBrary (libjf) tutorial: “from the ground up”

by Christian Ferrari

Table of Contents

List of Tables

List of Examples

About This Book

1. Acknowledgments

This is my first docbook work (in the past I used LaTeX) and as an "absolute beginner" I picked-up a good work and started "cut & paste" activity. The SGML of this book has initially copied from "Linux System Administration Guide", so special thanks to LarsWirzenius, JoannaOja, StephenStafford and AlexWeeks.

The subtitle of this book has been copied from "C++ from the ground up", a cornerstone in the history of programming books. A special thanks to HerbertSchildt and his easy to understand books.

2. Revision History

Revision History

Revision 0.1 2005-09-12 Revised by: Ch.F.
1. First version.

3. Source and pre-formatted versions available

The source code and other machine readable formats of this book can be found on the Internet via anonymous HTTP at the libjf home page http://libjf.sourceforge.net/ (http://libjf.sourceforge.net/). This book is available in at least its SGML source, as well as, HTML and PDF formats. Other formats may be available. HTML and PDF versions can be produced with this sequence of commands:

```
tar xvjf libjf-v.r.p-c.a.e-YYYYMMDDhhmm.tar.bz2
cd libjf-v.r.p
./configure
make html
make pdf
```

4. Typographical Conventions

Throughout this book, I have tried to use uniform typographical conventions. Hopefully they aid readability. If you can suggest any improvements please contact me.

Command names are expressed as: **jf_report**

C constants are expressed as: `JF_RC_OK`.

C functions are expressed as: `fflush`.

C structs are expressed as: jf_journal_opts_s

C struct fields are expressed as: *`recovery_enabled`*.

C types are expressed as: jf_word_t.

C vars are expressed as: `jf`.

Filenames are expressed as: `/opt/libjf`.

I will add to this section as things come up whilst editing. If you notice anything that should be added then please let me know.

5. English language and other ads

Unfortunately my English is very poor and this paper contains many mistakes and misunderstanding sentences; any help in fixing grammar and/or form will be appreciated.

This tutorial and the documented examples has been developed on a GNU/Linux system: if you use a different environment some details will need some tuning.

The examples contained in this tutorial has been developed and tested with libjf-0.3.5

Chapter 1. Introduction

1.1. Trademarks

AIX, z/OS, OS/400 are trademarks of IBM.

GNU is a registered trademark of the Free Software Foundation.

HP-UX is a trademark of Hewlett Packard.

Linux is a registered trademark of Linus Torvalds.

Mac OS X is a registered trademark of Apple.

Microsoft, Windows, Windows NT, Windows 2000, and Windows XP are trademarks and/or registered trademarks of Microsoft Corporation.

Solaris is a trademark of Sun Microsystems.

SuSE is a trademark of Novell.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Ltd.

Other product names mentioned herein may be trademarks and/or registered trademarks of their respective companies.

1.2. What’s libjf?

LIBJF stands for Journaled Files LIBrary, so libjf is a library that supplies file journaling features to a program.

Libjf is free software (LGPL license): please be sure you understood the license content before start to use this software.

Libjf is a Tiian’s original idea, design and implementation. In real life, Tiian is Christian Ferrari.

1.3. What's a "transaction"?

A transaction is a "unit of work" that has the following ("ACID") properties.

Atomicity:

a transaction is an indivisible unit of work; all of its actions either succeed or they all fail (logical units of work (LUW)). In the event of a failure of any operation, effects of all operations that make up the transaction should be undone, and data should be rolled back to its previous state.

Consistency:

after the transaction executes, it must leave the system in a correct state or abort (leaving the system in its initial state: COMMIT or ROLLBACK). For example, in the case of relational databases, a consistent transaction should preserve all the integrity constraints defined on the data.

Isolation:

a transaction's behavior is not affected by other transactions that execute concurrently. The effect of executing a set of transactions serially should be the same as that of running them concurrently. This requires two things:

- during the course of a transaction, an intermediate (possibly inconsistent) state of the data should not be exposed to all other transactions
- two concurrent transactions should not be able to operate on the same data. Database management systems usually implement this feature using locking.

Durability:

a transaction's effects are permanent (persistent) after it commits (information is saved in a recoverable storage resource).

1.4. What's file "journaling"?

Journaling is related to *transactions*: file journaling allows a program to "commit" or "rollback" changes performed against data.

Programmers used to deal with transactional databases like PostgreSQL™ and last releases of MySQL™ are already familiar with terms like "commit", "rollback", "data integrity" and so on.

I'll briefly explain these concepts; I'll assume you are a C addicted because libjf is a C library; I'll tend to use C standard I/O instead of *POSIX* I/O functions because libjf can be ported to non POSIX systems too.

Take a look to this piece of code:

. . .

```
FILE *file1, *file2;
...
fprintf(file1, "Hello ");
fprintf(file2, "world!\n");
...
```

Yeah, I mean, you'd prefer to write "Hello world!" on the same file (stream), but imagine you have to write 2 records on 2 different files and want to be sure than exactly one of this condition is true:

- all the two records are stored on file1, file2
- no record is stored on file1, file2

Why should you want this "strange thing"?

Many times you need this behavior: every time you have related information stored in two or more files you have to deal with a "data integrity issue".

1.4.1. Data integrity issue examples

File1 contains a "bit map" of used/free records of file2: if file2 contains billions of records, a bit map may improve dramatically the time wasted to look for a free record. File1 and file2 contents are strongly related and "data integrity issue" must be solved before your application reach "production status".

File1 contains "ack pending packets", file2 contains "OK packets": when an ack arrives, a record must be deleted from file1 and a record must be stored on file2. File1 and file2 are strongly related.

Sometimes, "data integrity issue" can be workarounded with a different data organization, sometimes every workaround exploits one or more "race condition" and the "data integrity issue" must be solved.

Come back to our code:

```
...
fprintf(file1, "Hello ");
fprintf(file2, "world!\n");
...
```

C standard I/O supplies a function can be used to flush application buffers and signal operating system a write must be performed: `fflush`; please pay attention there is **no** warranty data are stored on block device (hard disk) at `fflush` end (we will explain this later).

We can try to insert `fflush` in our program:

```
        ...
[A]
        fprintf(file1, "Hello ");
[B]
        fprintf(file2, "world!\n");
[C]
        fflush(file1);
[D]
        fflush(file2);
[E]
        ...
```

Now we can analyze what happens if the program is interrupted, for example with a POSIX signal, at step [A], [B], ... [E]

A. we are 100% sure files have not been updated
B. file2 has not been updated, file1 might be updated (it depends from buffer status, calling `fflush` is not a must to start buffer flushing)
C. file1 might be updated, file2 might be updated
D. file1 has been updated, file2 might be updated
E. file1 and file2 have been updated.

Even if we supposed data are flushed only by `fflush` (this is generally **not** true!), we could **not** workaround the integrity issue of step D.

What about *POSIX* I/O?

Using POSIX I/O implies usage of `write` instead of `fprintf` and `fsync`/ `fdatasync` instead of `fflush`. The functions do not perform the same actions because *POSIX* I/O does not use an application side buffer and `fsync`/`fdatasync` guarantees data are stored on block device (hard disk), but the “data integrity issue”, in the event of a system crash, is the same:

```
        ...
        int fd1, fd2;
        const char *str1 = "Hello ";
        const char *str2 = "world!\n";
        ...
[A]
        write(fd1, str1, strlen(str1));
[B]
        write(fd2, str2, strlen(str2));
[C]
        fsync(fd1);
[D]
        fsync(fd2);
[E]
```

```
...
```

The “data integrity issue” at step D has a pattern like the example based on C standard I/O.

Our examples show there are **two** type of matters:

- I/O functions (`fprintf`, `write`, ...) do **not** specify at what time file update happens: it may be any time between `fprintf` (`write`) and `fflush` (`fsync`/ `fdatasync`)
- there is not an **atomic** “flush”/“sync” function for 2 or more streams/file descriptors

1.4.2. Data integrity issue with only one file

Our previous examples showed us some typical data integrity issues that need a transactional tool to be solved.

There are situations that can benefit from a transactional support even when only one file is used; the best examples are text editors, office applications (word processors, spreadsheets, ...), configuration editors, and more...

All around the world there are programs writing many copies of the same file and checking file integrity at start-up to assure the text/document/configuration is consistent and is not affected by the consequences of an application/system crash. All that stuff might be replaced by a transactional tool like libjf.

1.5. Who should use libjf?

In the previous paragraphs we showed some issues can be addressed by a journaling/transactional tool, but why should I use libjf instead of a relational and transactional database?

A DBMS (Data Base Management System) is not only a transactional tool, it’s a more powerful object that implies a very different approach to coding, testing, software distribution and system management:

- query and updates are expressed as SQL statements: you may love it, you may hate it, you have to pay a powerful language interpreter overhead for it
- a DBMS must be up and running to support your application
- some tables must be defined using DBMS specific DDL (Data Description Language) statements
- DBMS objects (tables, indexes, ...) must be managed
- some DBMS are not easily embeddable in your application and you must manage one more dependency

- some DBMS do not provide an easy way to store/retrieve arbitrary data like structs and classes

DBMS have a lot of advantages too, especially if your application uses the most advanced features.

Now we can answer our original question: “Who should use libjf?”

If you need transaction support and are **not** interested in all the advanced features of a DBMS, libjf might be your preferred tool.

If your application needs the features of a relational DBMS and you are thinking to “re-write the wheel” using libjf, libjf is probably not what you are looking for.

1.6. What libjf is not?

It’s not a DBMS (Data Base Management System).

It’s not a journaled filesystem: libjf works on journaled (reiser, ext3, jfs, xfs...) and non journaled (ext2, hfs...) filesystems.

It’s not a “faster” tool to access files.

1.7. Collaboration

Development, porting and documentation are a hard work.

Are you a smart hacker?

Do you work with operating systems different than GNU/Linux?

Are you fluent in English language?

If you like, there’s room to collaborate at libjf project.

These topics must be addressed in the near future:

- porting to “old style” UNIX systems (AIX, HP-UX, Solaris, others...)
- porting to BSD systems (FreeBSD, OpenBSD, Mac OS X, others...)

- porting to native Microsoft Windows (2003, 2000, XP, etc...)
- porting to IBM z/OS
- porting to IBM OS/400
- performance tuning, system specific optimizations (after porting has been completed!)
- documentation review and improvement

If you would like to join this project, feel free to contact me at: <`tiian@users.sourceforge.net`>

Chapter 2. Getting started

2.1. “Supported” architectures

Is there a list of supported architectures?

libjf is free software developed on a voluntary basis: the concept of “supported” is not proper.

libjf project keeps track of environments successfully tested: if your system is in that list, you should not encounter big issues; if libjf has never been tested in an environment similar to your one, there might be a bit of porting job to be performed. The track of stress tested environments is hosted at libjf official site: http://libjf.sourceforge.net/ (http://libjf.sourceforge.net/)

libjf is currently developed on SuSE Linux operating system (SuSE Linux Professional 9.2): the closest to this environment the fewest problems you should have.

2.2. Retrieving and installing

libjf official site is hosted at *SourceForge.net*: http://libjf.sourceforge.net/ (http://libjf.sourceforge.net/)

Official packages are released only on SourceForge.net site, please avoid alternative sources.

Installing libjf on a “tested architecture” is a 6 steps task:

```
tar xvjf libjf-v.r.p-c.a.e-YYYYMMDDhhmm.tar.bz2
cd libjf-v.r.p
./configure
make
make check
sudo make install
```

if something goes wrong, please refer to this resources:

- FAQ (distributed with the package and available on line at official site)
- README (distributed with the package)
- forums and tracker hosted on SourceForge.net

Please note “make check” step does not perform an “in depth” test: if you are interested in testing all the features of libjf, take a look to shipped README file.

If you have completed your 6 steps installation, libjf should be installed at default path (`/opt/libjf`); should you prefer an alternative path, use `--prefix` option at **configure** step. From now on we will assume you have installed the library at its default path.

Installation procedure does not install documentation will remain in your package base sub-directory `doc`.

It's suggested to append libjf "bin" directory to your environment var `PATH`:

```
tiian@linux:~/tutorial> export PATH=$PATH:/opt/libjf/bin/
tiian@linux:~/tutorial> type jf_report
jf_report is /opt/libjf/bin/jf_report
```

this can save you a lot of typing.

2.2.1. Package name

What's the meaning of a so large name?

`libjf-v.r.p-c.a.e-YYYYMMDDhhmm.tar.bz2`

libjf

  library name

v

  major version number

r

  minor version number (release); even values for "stable" releases, odd values for "development" releases

p

  patch level

c

  "current" library version as understood by **libtool**

a

  "age" library version as understood by **libtool**

e

  "revision" library version as understood by **libtool**

YYYYMMDDhhmm

release timestamp as year, month, day, hours, minutes

tar

the file is a GNU tar file

bz2

the file is compressed with **bzip2** utility

2.2.2. Un-installing

If you are bored about libjf and want to un-install it, use the following command from package base directory:

```
sudo make uninstall
```

please pay attention directories are not removed; to clean-up directories too you may use this (dangerous) command:

```
rm -rf /opt/libjf
```

2.3. Hello world program

Every tutorial must start with “hello world” program and libjf can not violate this golden rule.

All the examples showed in this book are available in `doc/tutorial/examples` sub-directory distributed with software package.

Example 2-1. `hello_world.c`

```
#include <jf_file.h>;

int main()
{
        int rc;
        jf_file_t jf;
        size_t write;

        rc = jf_file_open(&jf, NULL, "jf_tut_foo", "w", NULL);
```

```
 8         if (JF_RC_OK != rc)
 9                 return 1;

10         rc = jf_file_printf(&jf, &write, "%s", "Hello world!\n");
11         if (JF_RC_OK != rc)
12                 return 1;

13         rc = jf_file_commit(&jf);
14         if (JF_RC_OK != rc)
15                 return 1;

16         rc = jf_file_close(&jf);
17         if (JF_RC_OK != rc)
18                 return 1;

19         printf("Hello world program is OK!\n");
20         return 0;
21 }
```

Hello world code explanation

Row 1

to use libjf a program must include at least `jf_file.h` header file

Row 5

declare object `jf` of type jf_file_t: `jf` is a “journaled file object”

Row 7

open (create & open) for write (“w”) a journaled file of name `jf_tut_foo` and associate it to object `jf`; second argument (`NULL`) indicates a private journal must be used, fifth argument (`NULL`) tells default parameters must be used

Row 10

write to journaled file some stuff; second argument is used to retrieve the number of bytes stored to journaled file, other arguments mimic `fprintf` function

Row 13

commit changes previously operated on journaled file `jf`

Row 16

close journaled file

2.3.1. Hello world compilation

To compile `hello_world.c` sample code and link it against libjf I suggest you to use **libtool** and its magic:

```
libtool --mode=link gcc -Wall -I/opt/libjf/include -L/opt/libjf/lib -ljf \
        -o hello_world hello_world.c
```

options:

`--mode=link`

source code must be compiled and linked

`gcc`

use GNU C compiler

`-Wall`

activate all C compiler warnings

`-I/opt/libjf/include`

specify where libjf include files must be searched

`-L/opt/libjf`

specify where libjf archive/shared object must be searched

`-ljf`

specify the name of the library must be linked to the produced executable

`-o hello_world`

name of the executable will be produced

`hello_world.c`

name of the source code file will be compiled

This version of “hello world” program is not so bad, isn’t it? All the stuff around the 6 described rows is an old tale: error checking and user feedback. You should note these essential points:

- libjf API (Application Programming Interface) is consistent to an object oriented model: the first argument is the object the function (method) is working on
- most libjf functions (methods) return a return code of type int; file `jf/jf_errors.h` enumerates the value set: `JF_RC_OK` is the “OK return code”, values greater than `JF_RC_OK` are “warning return codes”, values lesser than `JF_RC_OK` are “error return codes”

- object types are not pointers, but functions (methods) expects references to objects: user application can choose to allocate objects on stack (like in `hello_world.c` example) or on heap (using `malloc`/`free` functions); object sizes are not large enough to become a source of issues in today supercomputer powered PCs era
- all changes must be committed before a journaled file is closed: uncommitted changes are backed out (“implicit rollback”)

2.3.2. Hello world execution

If you correctly compiled `hello_world.c`, in current directory you should be able to see **hello_world** executable:

```
tiian@linux:~/tutorial> ls -la hello_world*
-rwxr-xr-x  1 tiian users 10079 2005-08-10 22:10 hello_world
-rw-r--r--  1 tiian users   578 2005-08-10 18:51 hello_world.c
```

to execute it type this command:

```
./hello_world
```

the program should print these sentence on your terminal:

```
Hello world program is OK!
```

And two files should appears in current directory:

```
tiian@linux:~/tutorial> ls -la jf_tut_foo*
-rw-r--r--  1 tiian users   13 2005-08-10 22:52 jf_tut_foo
-rw-r--r--  1 tiian users 8311 2005-08-10 22:52 jf_tut_foo.jf
```

`jf_tut_foo` is the journaled file with the content **hello_world** program stored and committed:

```
tiian@linux:~/tutorial> cat jf_tut_foo
Hello world!
```

`jf_tut_foo.jf` is the journal file implicitly created by libjf because **hello_world** program does not specify a journal file. The journal is a binary file you can browse with utility **jf_report**; try this command from your terminal:

```
tiian@linux:~/tutorial> jf_report -dt -j jf_tut_foo.jf
```

but this is a more intriguing tale can not be revealed at “hello world” step!

2.4. Hello world II

Every saga needs at least a second episode.

What happens if data are not committed to a journaled file? To discover the core of libjf transactionality it’s sufficient to remove `jf_file_commit` statement as in `hello_world2.c` example:

Example 2-2. `hello_world2.c`

```
#include <jf_file.h>

int main()
{
        int rc;
        jf_file_t jf;
        size_t write;

        rc = jf_file_open(&jf, NULL, "jf_tut_foo", "w", NULL);
        if (JF_RC_OK != rc)
                return 1;

        rc = jf_file_printf(&jf, &write, "%s", "Hello world!\n");
        if (JF_RC_OK != rc)
                return 1;

        rc = jf_file_close(&jf);
        if (JF_RC_OK != rc)
                return 1;

        printf("Hello world II program is OK!\n");
        return 0;
}
```

Compile `hello_world2.c` with command:

```
libtool --mode=link gcc -Wall -I/opt/libjf/include -L/opt/libjf/lib -ljf \
        -o hello_world2 hello_world2.c
```

remove temporary files wrote by hello_world:

```
tiian@linux:~/tutorial> rm -f jf_tut_foo*
```

and run **hello_world2** program:

```
tiian@linux:~/tutorial> ./hello_world2
Hello world II program is OK!
```

Take a look to files produced by **hello_world2** execution:

```
tiian@linux:~/tutorial> ls -la jf_tut_foo*
-rw-r--r--  1 tiian users    0 2005-08-11 11:05 jf_tut_foo
-rw-r--r--  1 tiian users 8278 2005-08-11 11:05 jf_tut_foo.jf
```

`jf_tut_foo` is now an empty file because data was not committed before calling `jf_file_close` (this behavior is named “implicit rollback”); just for the sake of curiosity, inspect the associated journal file:

```
tiian@linux:~/tutorial> jf_report -dt -j jf_tut_foo.jf
<?xml version="1.0" encoding="UTF-8"?>
<journal>
  <header magic_number='0x41524153' version='1' file_id_mask='0x8'
      file_id_mask_shift='3' size_mask='0xfffffff0' size_mask_shift='4'
      file_size='4194304' file_num='3' rotation_threshold='0.800'
      ctrl_recs='36' journal_recs='8278' />
  <journaled_file_table max_files='2' number_of_files='2'
      file_table='0x804b170'>
    <file id='0' name='jf_tut_foo.jf' last_pos='8278' last_size='0'
        status='0' last_uc_pos='0' last_uc_size='8278' stream='0x804b008' />
    <file id='1' name='jf_tut_foo' last_pos='0' last_size='0' status='0'
        last_uc_pos='0' last_uc_size='0' stream='(nil)' />
    </journaled_file_table>
  <records>
  </records>
</journal>
```

no records have been written.

2.5. Hello world III

The final cut.

How (explicit) “rollback” works? To discover this kind of magic we’re cooking `hello_world3.c` example.

Example 2-3. `hello_world2.c`

```
#include <jf_file.h>

int main()
{
        int rc;
        jf_file_t jf;
        size_t write;

        rc = jf_file_open(&jf, NULL, "jf_tut_foo", "w", NULL);
        if (JF_RC_OK != rc)
                return 1;

        rc = jf_file_printf(&jf, &write, "%s", "Hello");
11         if (JF_RC_OK != rc)
12                 return 1;

13         rc = jf_file_rollback(&jf);
14         if (JF_RC_OK != rc)
15                 return 1;

16         rc = jf_file_printf(&jf, &write, "%s", " world!\n");
17         if (JF_RC_OK != rc)
18                 return 1;

19         rc = jf_file_commit(&jf);
20         if (JF_RC_OK != rc)
21                 return 1;

22         rc = jf_file_close(&jf);
23         if (JF_RC_OK != rc)
24                 return 1;

25         printf("Hello world III program is OK!\n");
26         return 0;
27 }
```

Hello world III code explanation

Row 1

to use libjf a program must include at least `jf_file.h` header file

Row 5

declare object `jf` of type jf_file_t: `jf` is a "journaled file object"

Row 7

open (create & open) for write ("w") a journaled file of name `jf_tut_foo` and associate it to object `jf`; second argument (`NULL`) indicates a private journal must be used, fifth argument (`NULL`) tells default parameters must be used

Row 10

write to journaled file string "Hello"; second argument is used to retrieve the number of bytes stored to journaled file, other arguments mimic `fprintf` function

Row 13

rollback changes previously operated on journaled file `jf`: write of string "Hello" will be discarded (undone)

Row 16

write to journaled file different stuff (string " world!\n")

Row 19

commit changes previously operated on journaled file `jf`

Row 22

close journaled file

2.5.1. Hello world III compile & run

Use a command like this to compile `hello_world3.c` source code:

```
libtool --mode=link gcc -Wall -I/opt/libjf/include -L/opt/libjf/lib -ljf \
        -o hello_world3 hello_world3.c
```

remove temporary files wrote by first and/or second episode of hello world saga:

```
tiian@linux:~/tutorial> rm -f jf_tut_foo*
```

and run **hello_world3** program:

```
tiian@linux:~/tutorial> ./hello_world3
Hello world III program is OK!
```

Take a look to journaled file and associated private journal:

```
tiian@linux:~/tutorial> ls -la jf_tut_foo*
-rw-r--r--  1 tiian users    8 2005-08-11 14:11 jf_tut_foo
-rw-r--r--  1 tiian users 8306 2005-08-11 14:11 jf_tut_foo.jf
```

inspect journaled file jf_tut_foo:

```
tiian@linux:~/tutorial> cat jf_tut_foo
 world!
```

The first string (“Hello”) was not stored in journaled file because an explicit rollback was performed; inspect journal content:

```
tiian@linux:~/tutorial> jf_report -df -j jf_tut_foo.jf
<?xml version="1.0" encoding="UTF-8"?>
<journal>
  <header magic_number='0x41524153' version='1' file_id_mask='0x8'
      file_id_mask_shift='3' size_mask='0xfffffff0' size_mask_shift='4'
      file_size='4194304' file_num='3' rotation_threshold='0.800'
      ctrl_recs='36' journal_recs='8278' />
  <journaled_file_table max_files='2' number_of_files='2'
```

```
      file_table='0x804b170'>
    <file id='0' name='jf_tut_foo.jf' last_pos='8306' last_size='0'
        status='0' last_uc_pos='0' last_uc_size='8306' stream='0x804b008' />
    <file id='1' name='jf_tut_foo' last_pos='8' last_size='8' status='2'
        last_uc_pos='0' last_uc_size='0' stream='(nil)' />
  </journaled_file_table>
  <records>
    <append jrn_rec_off='8278' file_id='1' size='8' offset='0'>
      <data type='redo' format='hex'>20 77 6f 72 6c 64 21 0a </data>
      <data type='redo' format='text'> world! </data>
    </append>
    <commit jrn_rec_off='8302' file_id='1'/>
  </records>
</journal>
```

2.6. Hello world saga conclusions

These first three examples show basic usage of libjf library:

- journaled file creation
- how to write data
- how to commit data
- how to rollback data

in the following chapters we will dive into more advanced features of libjf.

Chapter 3. libjf basics

3.1. Many journaled files, one journal

In the previous chapter we introduced a bit of libjf using only one journaled file at a time: that usage may help you in writing easier programs because you don’t have to implement the “back-out” logic, it’s a gift of libjf.

Now it’s time to investigate how libjf can solve “data integrity issue” at its root; the following examples will exploit only two journaled files, but the principle can be extended to any number.

Example 3-1. `two_files.c`

```
#include <jf_file.h>

int main()
{
        int rc;
        size_t write;
        jf_journal_t j;
        jf_file_t jf1, jf2;
        struct jf_journal_opts_s jopts;
        struct jf_file_open_opts_s fopts;
        const char *file1_data1 = "First string for first file\n";
        const char *file1_data2 = "Second string for first file\n";
        const char *file2_data1 = "First string for second file\n";
        const char *file2_data2 = "Second string for second file\n";

        jf_set_default_journal_opts(&jopts);
        jopts.flags |= JF_JOURNAL_PROP_OPEN_O_CREAT |
                JF_JOURNAL_PROP_OPEN_O_EXCL;
        rc = jf_journal_open(&j, "jf_tut_foo-journal", 2, &jopts);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }

        jf_set_default_file_open_opts(&fopts);
        fopts.join_the_journal = TRUE;
        rc = jf_file_open(&jf1, &j, "jf_tut_foo-data1", "w", &fopts);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_file_open(&jf2, &j, "jf_tut_foo-data2", "w", &fopts);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }

        rc = jf_file_write(&jf1, file1_data1, strlen(file1_data1),
                           &write);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_file_write(&jf2, file2_data1, strlen(file2_data1),
                           &write);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_journal_rollback(&j);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }

        rc = jf_file_write(&jf1, file1_data2, strlen(file1_data2),
                           &write);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_file_write(&jf2, file2_data2, strlen(file2_data2),
                           &write);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_journal_commit(&j);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }

        rc = jf_file_close(&jf1);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_file_close(&jf2);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_journal_close(&j);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
```

```
83          printf("two_files program ended OK!\n");
84          return 0;
85 }
```

`two_files.c` code explanation

Row 1

: to use libjf a program must include at least `jf_file.h` header file

Row 6

: declare object `j` of type jf_journal_t: `j` is a “journal object”

Row 7

: declare object `jf1`, `jf2` of type jf_file_t: `jf1` and `jf2` are a “journaled file objects”

Row 8

: declare a struct will be used to store options related to journal

Row 9

: declare a struct will be used to store options related to journaled files

Row 14

: set default value for options related to journal

Rows 15-16

: add some flags to journal related options: the journal must be created and the journal must not exist

Row 17

: open the journal file `jf_tut_foo-journal` and associate it to journal object `j`, the journal will handle 2 journaled files and use options specified by struct `jopts`

Row 18

: check previous operation return code: print the return code and its human readable description if something goes wrong

Row 22

: set default value for options related to journaled files

Row 23

: specify the journaled file must “join the journal”; joining a journal means the journal will store all transactional information about the journaled file

Row 24

open journaled file `jf_tut_foo-data1`, associate to object `jf1`, use journal `j`, data will be written ("w") and additional options must be picked-up from `fopts`

Row 29

open journaled file `jf_tut_foo-data2`, associate to object `jf2`, use journal `j`, data will be written ("w") and additional options must be picked-up from `fopts`

Row 34

write string `file1_data1` to journaled file `jf1`

Row 40

write string `file2_data1` to journaled file `jf2`

Row 46

back out all the changes operated on all the journaled files managed by journal object `j`

Row 51

write string `file1_data2` to journaled file `jf1`

Row 57

write string `file2_data2` to journaled file `jf2`

Row 63

commit all the changes operated on all the journaled files managed by journal object `j`

Row 68

close journaled file `jf1`

Row 73

close journaled file `jf2`

Row 78

close journal `j`

3.1.1. two_files.c compile and run

To compile `two_files.c` you may use this **libtool** command:

```
libtool --mode=link gcc -Wall -I/opt/libjf/include -L/opt/libjf/lib \
        -ljf -o two_files two_files.c
```

run **two_files** program:

```
tiian@linux:~/tutorial> ./two_files
two_files program ended OK!
```

take a look to the files produced by two_files execution:

```
tiian@linux:~/tutorial> ls -la jf_tut_foo*
-rw-r--r--  1 tiian users    29 2005-08-11 16:50 jf_tut_foo-data1
-rw-r--r--  1 tiian users    30 2005-08-11 16:50 jf_tut_foo-data2
-rw-r--r--  1 tiian users 16607 2005-08-11 16:50 jf_tut_foo-journal
```

inspect first and second journaled file:

```
tiian@linux:~/tutorial> cat jf_tut_foo-data1
Second string for first file
tiian@linux:~/tutorial> cat jf_tut_foo-data2
Second string for second file
```

take a look to journal content with command

```
tiian@linux:~/tutorial> jf_report -j jf_tut_foo-journal -dt
```

the first strings, on all journaled files, were backed out by `jf_journal_rollback` at line 46; the second strings, on all journaled files, were committed by `jf_journal_commit` at line 63.

3.1.2. `two_files.c` interesting aspects

You should note these aspects in `two_files.c` source code:

- in libjf API some data types are not "typedefed" and they must be used with the explicit reserved word "struct" (rows 8-9); this is a desired behavior because some structs should not be interpreted as "classes". This simple rule can help you understanding the idea:
  - "classes" are types with name terminating in "_t" and methods to create/destroy/manage: you must not access "properties" of an object with "methods" (functions) out of the scope of the "class"
  - "structs" are commodity data aggregation with name terminating in "_s" and no methods to manipulate them: you have to set the values of the interesting fields when necessary
- it's a good programming practice to close journaled files before closing journal: leaving objects in open status will cause a useless, time consuming, automatic recovery at next open time; forgetting object closure may lead to memory leak bugs.

The illustrated schema is really simple: two global transactions on two journaled files, first transaction was backed out, second transaction was committed.

What happens in the event of an application crash?

3.2. Two journaled files and an application crash

A simple way to simulate an application crash is forcing a division by zero exception: this may not be the cause of the crash of your application once it has been moved to production environment, but it's an example can light on libjf power.

Example 3-2. `two_files_crash.c`

```c
#include <jf_file.h>

int main()
{
        int rc;
        size_t write;
        jf_journal_t j;
        jf_file_t jf1, jf2;
        struct jf_journal_opts_s jopts;
        struct jf_file_open_opts_s fopts;
        const char *file1_data1 = "First string for first file\n";
        const char *file1_data2 = "Second string for first file\n";
        const char *file2_data1 = "First string for second file\n";
        const char *file2_data2 = "Second string for second file\n";
        int x, y;

        jf_set_default_journal_opts(&jopts);
        jopts.flags |= JF_JOURNAL_PROP_OPEN_O_CREAT |
                JF_JOURNAL_PROP_OPEN_O_EXCL;
        rc = jf_journal_open(&j, "jf_tut_foo-journal", 2, &jopts);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }

        jf_set_default_file_open_opts(&fopts);
        fopts.join_the_journal = TRUE;

        rc = jf_file_open(&jf1, &j, "jf_tut_foo-data1", "w", &fopts);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_file_open(&jf2, &j, "jf_tut_foo-data2", "w", &fopts);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }

        rc = jf_file_write(&jf1, file1_data1, strlen(file1_data1),
                           &write);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_file_write(&jf2, file2_data1, strlen(file2_data1),
                           &write);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_journal_commit(&j);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }

        rc = jf_file_write(&jf1, file1_data2, strlen(file1_data2),
                           &write);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_file_write(&jf2, file2_data2, strlen(file2_data2),
                           &write);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        /* a fool crash simulation */
        x = 0; y = 5;
        while (TRUE)
                x += 5 / y--;
        printf("This should not print x = %d\n", x);

        rc = jf_file_close(&jf1);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_file_close(&jf2);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }

        rc = jf_journal_close(&j);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }

        printf("two_files_crash program ended OK!\n");
        return 0;
}
```

`two_files_crash.c` is slightly different than `two_files.c`:

Row 47

a first commit is performed by our application to successfully close the first transaction

Rows 64-68

a trick has been introduced to force a “division by zero” exception (row 67) avoiding the situation is detected by most compilers.

To compile `two_files_crash.c` source code use our old friend **libtool** command:

```
libtool --mode=link gcc -Wall -I/opt/libjf/include -L/opt/libjf/lib -ljf \
        -o two_files_crash two_files_crash.c
```

if you executed **two_files** program too, the execution of **two_files_crash** should exploit an error condition:

```
tiian@linux:~/tutorial> ./two_files_crash
-15/ERROR: file can not be created because it already exists
```

the problem is due to **two_files_crash** request of “new journal creation” (take a look at row 16); remove old files and run it again:

```
tiian@linux:~/tutorial> rm jf_tut_foo-*
tiian@linux:~/tutorial> ./two_files_crash
Floating point exception
```

our application crashed as expected, look at journaled files:

```
tiian@linux:~/tutorial> ls -la jf_tut_foo-*
-rw-r--r--  1 tiian users    28 2005-08-11 17:34 jf_tut_foo-data1
-rw-r--r--  1 tiian users    29 2005-08-11 17:34 jf_tut_foo-data2
-rw-r--r--  1 tiian users 16605 2005-08-11 17:34 jf_tut_foo-journal
tiian@linux:~/tutorial> cat jf_tut_foo-data1
First string for first file
tiian@linux:~/tutorial> cat jf_tut_foo-data2
First string for second file
```

strings of first transaction are at their place as desired, strings of second transaction were backed out as expected and data kept by journaled files are consistent.

You may enjoy transactionality of this example moving rows 64-68 in different places like between rows 40 and 41.

3.3. Two journaled files and a partial transaction

When you are working with two or more journaled files, sometimes you need to close a “partial transaction” before the global transaction has completed: this happens when you want to save the fact you tried the transaction and a rollback would erase this information. The following example will show this behavior.

Example 3-3. `two_files_crash2.c`

```c
#include <jf_file.h>

int main()
{
        int rc;
        size_t write;
        jf_journal_t j;
        jf_file_t jf1, jf2;
        struct jf_journal_opts_s jopts;
        struct jf_file_open_opts_s fopts;
        const char *file1_data1 = "First string for first file\n";
        const char *file1_data2 = "Second string for first file\n";
        const char *file2_data1 = "First string for second file\n";
        const char *file2_data2 = "Second string for second file\n";
        int x, y;

        jf_set_default_journal_opts(&jopts);
        jopts.flags |= JF_JOURNAL_PROP_OPEN_O_CREAT |
                JF_JOURNAL_PROP_OPEN_O_EXCL;
        rc = jf_journal_open(&j, "jf_tut_foo-journal", 2, &jopts);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }

        jf_set_default_file_open_opts(&fopts);
        fopts.join_the_journal = TRUE;

        rc = jf_file_open(&jf1, &j, "jf_tut_foo-data1", "w", &fopts);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_file_open(&jf2, &j, "jf_tut_foo-data2", "w", &fopts);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }

        rc = jf_file_write(&jf1, file1_data1, strlen(file1_data1),
                           &write);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_file_write(&jf2, file2_data1, strlen(file2_data1),
                           &write);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_file_commit(&jf2);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }

        rc = jf_file_write(&jf1, file1_data2, strlen(file1_data2),
                           &write);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_file_write(&jf2, file2_data2, strlen(file2_data2),
                           &write);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        /* a fool crash simulation */
        x = 0; y = 5;
        while (TRUE)
                x += 5 / y--;
        printf("This should not print x = %d\n", x);

        rc = jf_file_close(&jf1);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_file_close(&jf2);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }

        rc = jf_journal_close(&j);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }

        printf("two_files_crash II program ended OK!\n");
        return 0;
}
```

At row 47, we changed `jf_journal_commit(&j)` with `jf_file_commit(&jf2)`: instead of committing the whole “unit of work”, we decided to commit only the changes operated against `jf2`.

The source `two_files_crash2.c` can be compiled with this command:

```
libtool --mode=link gcc -Wall -I/opt/libjf/include -L/opt/libjf/lib -ljf \
        -o two_files_crash2 two_files_crash2.c
```

execute it after you have deleted journal and journaled files created by **two_files_crash**:

```
tiian@linux:~/tutorial> rm jf_tut_foo-*
tiian@linux:~/tutorial> ./two_files_crash2
Floating point exception
```

take a look to files produced by **two_files_crash2**:

```
tiian@linux:~/tutorial> ls -la jf_tut_foo-*
-rw-r--r--  1 tiian users     0 2005-08-12 22:37 jf_tut_foo-data1
-rw-r--r--  1 tiian users    29 2005-08-12 22:37 jf_tut_foo-data2
-rw-r--r--  1 tiian users 16561 2005-08-12 22:37 jf_tut_foo-journal
tiian@linux:~/tutorial> cat jf_tut_foo-data2
First string for second file
```

first journaled file is empty because no data has been committed to it, second journaled file contains the first string because a partial commit has been performed.

Can libjf commit/rollback a random set of journaled files? At the time of this writing, libjf can commit/rollback only:

- all the journaled files associated to a journal
- a specific journaled file

In the future a partial commit/rollback could be implemented: it requires a partial rewrite of some core functions, but the design of libjf and the actual implementation do not obstacle this interesting feature.

3.4. The recovery pending status

What happens when an application or the whole operating system crashes?

After a crash, a journal and some of its journaled files, may be in “recovery pending” status: the journal contains all the necessary information to fix journaled files, but journaled files content may be *not* consistent.

The previous examples are trivial and inconsistency does not happen, but... please remove previous journal and journaled files, and execute again **two_files_crash** program:

```
tiian@linux:~/tutorial> rm jf_tut_foo-*
tiian@linux:~/tutorial> ./two_files_crash
Floating point exception
```

Now execute the utility program **jf_recover** specifying “test mode” and check the return code:

```
tiian@linux:~/tutorial> jf_recover -j jf_tut_foo-journal -t
tiian@linux:~/tutorial> echo $?
0
```

please take a look to on-line help:

```
tiian@linux:~/tutorial> jf_recover -h
Usage: jf_recover -j JOURNALFILENAME [-t [-d{n|h|t|f}] ] [-f]
Recover all files journaled by JOURNALFILENAME

        -j : specify the name of the journal file must be host FILE
        -t : test only mode, useful to understand if recovery is necessary
             exit code values:
                0 - recovery is necessary
                1 - forced recovery is necessary
                2 - an error happened
                3 - recovery is not necessary
        -d : specifies which data must be dumped to output (test only mode)
                -dn   no data are dumped (essential dump)
                -dh   hexadecimal format data dump
                -dt   text format data dump
                -df   full (hexadecimal and text) data dump
        -f : forced recovery mode, useful to recover a damaged journal;
             use only as a LAST resource
        -h : print this help
```

as the help explain, if “0” is returned, a recovery is necessary: this is exactly what we expect because the application crashed and libjf is part of the application, so libjf crashed with the application and the journaled files needs a recovery phase. To discover which operations will be performed by **jf_recover**, try:

```
tiian@linux:~/tutorial> jf_recover -j jf_tut_foo-journal -t -dt
<?xml version="1.0" encoding="UTF-8"?>
<recovery_report>
  <rotation_recovery>false</rotation_recovery>
  <journal_real_path>jf_tut_foo-journal</journal_real_path>
  <analyze_damaged_journal>false</analyze_damaged_journal>
  <journal_last_pos>16512</journal_last_pos>
  <damaged_journal>false</damaged_journal>
  <recovery_pending_status>true</recovery_pending_status>
  <patches>
```

```
   <patch type="redo">
    <append jrn_rec_off='16512' file_id='1' size='28' offset='0'>
      <data type='redo' format='text'>First string for first file </data>
    </append>
   </patch>
  </patches>
  <new_journal_last_pos>28</new_journal_last_pos>
  <patches>
   <patch type="redo">
    <append jrn_rec_off='16556' file_id='2' size='29' offset='0'>
      <data type='redo' format='text'>First string for second file </data>
    </append>
   </patch>
  </patches>
  <new_journal_last_pos>29</new_journal_last_pos>
  <write_rollback_record/>
</recovery_report>
```

jf_recover discovered two patches must be applied to journaled files and a rollback record must be written on journal: so we have just discovered ours journaled files incidentally are OK, but application crashed before libjf was able to mark the “no recovery pending status” (this behavior is the consequence of some optimizations: if you sync your files at every step, the resulting system becomes unusable).

OK, we discovered our journal is in “recovery pending” and the operations will be performed, do them and check the status a second time:

```
tiian@linux:~/tutorial> jf_recover -j jf_tut_foo-journal
tiian@linux:~/tutorial> jf_recover -j jf_tut_foo-journal -t
tiian@linux:~/tutorial> echo $?
3
```

Now journal is not in “recovery pending” status and data in journaled files can be safely accessed by any application, for example a utility command like **cat** or **more**...

Why **jf_recover** returns “0”, “OK” code, if the journal is in “recovery pending” status? There are two reasons:

- **jf_recover** executed in “test mode” has to check for “recovery pending” status, so “0”, means “OK, recovery pending is TRUE”
- you can concatenate **jf_recover** in a shell script using the compact form “&&”:

  ```
  jf_recover -j <journal_name> -t -df && jf_recover -j <journal_name>
  ```

  a recovery phase is performed only if the journal is in “recovery pending status”.

3.4.1. Automatic recovery

Can an application open a journal in “recovery pending” status without a previous execution of utility program **jf_recover**?

Yes, but it’s not the default behavior because recovery is a potential dangerous operation and I think it’s not a good thing someone takes a decision without asking you!

The boolean *`recovery_enabled`* field of jf_journal_opts_s struct, passed to `jf_journal_open` method, must be set to `TRUE` if you want automatic cold recovery feature active; the boolean field *`recovery_damaged_journal`* does the same when the journal is damaged: pay attention a damaged journal is a very serious situation will probably lead to data corruption (do you have a backup of your files?!).

“libjf API reference guide” documents all the available options.

3.5. Text files

In UNIX derived systems there is no difference between text files and binary files, but in DOS derived systems like Microsoft Windows this is not true: “new line” code is translated to the two characters sequence “carriage return” “line feed”; Mac OS use a single but different code to represent “newline” concept.

POSIX I/O does not afford the issue because the API does not provide string related functions: the programs have to deal with “binary buffers” and these sort of issues are considered “application side problems”.

C standard I/O tried to mask the issue adopting the concept: program does not know the internals of the operating system and “newline” is transparently encoded/decoded by standard I/O library. This approach is very elegant but there’s a subtle problem: when a text file is moved from a UNIX style system to a DOS style one, the file must be “translated”. In the file transfer world, this was not an issue, of course: all the data mover since FTP age perform the “newline” translation. In the data sharing age the solution is not so easy: imagine a GNU/Linux system serving UNIX systems through NFS and Windows systems through SAMBA. With a bit of imagination you may think a multi platform application running on UNIX and Windows... what happens with “text” files? Which “standard” should be adopted?

- If “UNIX standard” is adopted, Windows applications have to use the file as “binary” and provide an application side “in flight” translation of newline.
- If “DOS standard” is adopted, the opposite issue must be solved at application level.

The elegant solution seems to be bugged when files are shared among UNIX, Windows, Mac, etc...

At the time of this writing libjf does not provide a “transparent” dealing of newline dilemma: instead of opening a “text” journaled file, an application can choose to open a “DOS text journaled file” appending a “D” to “open mode”.

Example 3-4. `dos_text.c`

```
#include %lt;jf_file.h>

int main()
{
        int rc;
        jf_file_t jf;
        size_t write;

        rc = jf_file_open(&jf, NULL, "jf_tut_foo", "wD", NULL);
        if (JF_RC_OK != rc)
                return 1;

        rc = jf_file_printf(&jf, &write, "%s", "Hello world!\n");
        if (JF_RC_OK != rc)
                return 1;

        rc = jf_file_commit(&jf);
        if (JF_RC_OK != rc)
                return 1;

        rc = jf_file_close(&jf);
        if (JF_RC_OK != rc)
                return 1;

        printf("DOS text program is OK!\n");
        return 0;
}
```

`dos_text.c` source code can be compiled with this command:

```
libtool --mode=link gcc -Wall -I/opt/libjf/include -L/opt/libjf/lib -ljf \
        -o dos_text dos_text.c
```

execute it and verify the produced journaled file:

```
tiian@linux:~/tutorial> ./dos_text
DOS text program is OK!
tiian@linux:~/tutorial> od -cx jf_tut_foo
0000000   H   e   l   l   o       w   o   r   l   d   !  \r  \n
        6548 6c6c 206f 6f77 6c72 2164 0a0d
0000016
```

you can note the produced journaled file is very likely the journaled file produced by **hello_world** program, but the newline sequence is now encoded following as “DOS standard” (carriage return, line feed).

3.5.1. Conclusions

- **hello_world** program writes UNIX text journaled files independently from the operating system used to compile and run it
- **dos_text** program writes DOS text journaled files independently from the operating system used to compile and run it
- if you do not like this “strange” behavior, simply use libjf in binary mode, like you are used with POSIX I/O.

3.5.2. Future developments

1. What about MAC OS X?

Unfortunately I don’t have it: when the port will be performed, this issue should be solved; I suppose a new “open mode” flag might be introduced, for example “M”, to specify a “MAC OS text journaled file”.

2. Will a “transparent flag” be provided in the future?

I don’t think a transparent flag like “T” (text) is useful because it’s a bit confusing: think to an application compiled as Microsoft Windows native and as Cygwin emulation... When executed as native it should adopt DOS standard, but when executed as a cygwin application it should adopt UNIX standard... Who’s taking care about user’s mind? I know I cannot change the world, so if a lot of people asked for it, it would be developed.

3.6. Restartable reads

An interesting libjf feature is the “restartable read” concept: many times a program has to process some input files to produce output files. What happens when an error occur processing a specific input record? If the program crashed the first time, there’s more than a chance it will crash twice or more when trying to process the “dirty record”... Take a look to this example program:

Example 3-5. `restartable_reads.c`

```
#include <jf_file.h>

int main()
{
        int rc, c;
        jf_file_t jf;
```

```
 6         rc = jf_file_open(&jf, NULL, "jf_tut_foo", "R", NULL);
 7         if (JF_RC_OK != rc)
 8                 return 1;

 9         if (jf_file_eof(&jf)) {
10                 rc = jf_file_rewind(&jf);
11                 if (JF_RC_OK != rc)
12                         return 1;
13         } /* if (jf_file_eof(&jf)) */

14         rc = jf_file_getc(&jf, &c);
15         if (JF_RC_OK != rc)
16                 return 1;

17         printf("Read char ’%c’ (0x%x)\n", c, c);

18         rc = jf_file_commit(&jf);
19         if (JF_RC_OK != rc)
20                 return 1;

21         rc = jf_file_close(&jf);
22         if (JF_RC_OK != rc)
23                 return 1;

24         printf("Restartable reads program is OK!\n");
25         return 0;
26 }
```

`restartable_reads.c` code explanation

Row 6

the special flag “R” is used at open time: as documented in “API reference guide”, the stream is positioned at last committed position

Row 9

check is the file pointer is at “end of file” position

Row 10

move file pointer to first file position; `jf_file_seek` might be used instead of `jf_file_rewind` if you preferred

Row 14

fetch only one char from journaled file

3.6.1. Compilation and execution

You may use this command to compile `restartable_reads.c` source code:

```
libtool --mode=link gcc -Wall -I/opt/libjf/include -L/opt/libjf/lib -ljf \
        -o restartable_reads restartable_reads.c
```

restartable_reads program needs a not empty `jf_tut_foo` journaled file to be executed; our old friend **hello_world** can help us one more time:

```
tiian@linux:~/tutorial> ./hello_world
Hello world program is OK!
tiian@linux:~/tutorial> ./restartable_reads
Read char ’H’ (0x48)
Restartable reads program is OK!
```

restartable_reads program fetched “H”, the first char in journaled file `jf_tut_foo`. What happens if we run **restartable_reads** again?

```
tiian@linux:~/tutorial> ./restartable_reads
Read char ’e’ (0x65)
Restartable reads program is OK!
```

“e”, the second char in journaled file `jf_tut_foo` is fetched... Are you guessing what will happen at next execution?

```
tiian@linux:~/tutorial> ./restartable_reads
Read char ’l’ (0x6c)
Restartable reads program is OK!
```

“l”, the third char in journaled file is fetched!

The same behavior will happen when using a different read method like `jf_file_gets` or `jf_file_read`: at commit time, the file pointer is moved and transactionally kept by journal.

3.6.2. Restartable reads and rollback

What happens when a restartable read transaction is backed out by an explicit or implicit rollback? Take a look to this example:

Example 3-6. `restartable_reads_rollback.c`

```
#include <jf_file.h>

int main()
```

```
 3 {
 4         int rc, c;
 5         jf_file_t jf;

 6         rc = jf_file_open(&jf, NULL, "jf_tut_foo", "R", NULL);
 7         if (JF_RC_OK != rc)
 8                 return 1;

 9         if (jf_file_eof(&jf)) {
10                 rc = jf_file_rewind(&jf);
11                 if (JF_RC_OK != rc)
12                         return 1;
13         } /* if (jf_file_eof(&jf)) */

14         rc = jf_file_getc(&jf, &c);
15         if (JF_RC_OK != rc)
16                 return 1;

17         printf("Read char ’%c’ (0x%x)\n", c, c);

18         rc = jf_file_rollback(&jf);
19         if (JF_RC_OK != rc)
20                 return 1;

21         rc = jf_file_close(&jf);
22         if (JF_RC_OK != rc)
23                 return 1;

24         printf("Restartable reads rollback program is OK!\n");
25         return 0;
26 }
```

The only difference between `restartable_reads.c` and `restartable_reads_rollback.c` is at row 18 where `jf_file_commit` has been replaced with `jf_file_rollback`.

Compile and run this example:

```
libtool --mode=link gcc -Wall -I/opt/libjf/include -L/opt/libjf/lib -ljf \
        -o restartable_reads_rollback restartable_reads_rollback.c
tiian@linux:~/tutorial> ./hello_world
Hello world program is OK!
tiian@linux:~/tutorial> ./restartable_reads_rollback
Read char ’H’ (0x48)
Restartable reads rollback program is OK!
tiian@linux:~/tutorial> ./restartable_reads_rollback
Read char ’H’ (0x48)
Restartable reads rollback program is OK!
tiian@linux:~/tutorial> ./restartable_reads_rollback
Read char ’H’ (0x48)
Restartable reads rollback program is OK!
```

Every execution reads the first char because the read transaction is not committed. Now you can play some minutes with “restartable_reads” and “restartable_reads_rollback” to simulate committed and backed out transactions.

3.6.3. Conclusions

Using libjf you are able to write restartable applications:

- you don’t have to deal with saving in some “safe place” the last read record to avoid multiple processing
- using partial transactions, you can commit a read before the global transaction commit and avoid a crash generated by dirty input at next restart

3.7. Other “open mode” options

It’s strongly suggested to take a look to `jf_file_open` page in libjf “API reference guide” because some more “open mode” flags are explained: you will find “append” and “read/write” combined flags as used with C standard I/O library. Trying it is a straightforward programming exercise.

Chapter 4. Diving into libjf

In the previous chapters we discussed about transactions and recovery but we didn't afford the argument of data synchronization we announced in introduction. It's time to discover this intriguing land.

4.1. Synchronization type

Rule number one: every operating system has some differences when dealing with data synchronization. libjf should be portable across many environments and it's difficult to take benefit of some specific operating system related features when the software must be portable.

Rule number two: documentation from standards are very weak; just to figure out what "very weak" means, take a look to documentation available in IEEE std. 1003-2001

libjf supply two type of synchronization: "fast" and "safe".

4.1.1. libjf fast synchronization

This type of synchronization prevent data loss in case of application crash and does not supply any warranty in case of system crash.

Fast synchronization uses `fflush` function to flush buffer content to operating system: in the event of application crash, operating system closes all open file descriptors and queues pending data for writing. If the application crashed its data would be saved by operating system.

4.1.2. libjf safe synchronization

This type of synchronization prevent data loss in case of system crash.

Safe synchronization uses `fdatasync` (`fsync` when the previous is not available) function to sync device content.

4.1.3. How can an application choose the type of synchronization?

An application may hard code the type of synchronization specifying flag `JF_JOURNAL_PROP_SYNC_SAFE` or `JF_JOURNAL_PROP_SYNC_FAST` at `jf_journal_open` time:

```
jf_journal_t j;
struct jf_journal_opts_s jopts;

jf_set_default_journal_opts(&jopts);
jopts.flags |= JF_JOURNAL_PROP_SYNC_SAFE;
rc = jf_journal_open(&j, "jf_tut_foo-journal", 2, &jopts);
```

this method has all the benefits and the disadvantages of “hard wired” parameters. libjf allows you to specify the type of synchronization at run time: this is the default behavior, but you may ask for it by your own:

```
jf_journal_t j;
struct jf_journal_opts_s jopts;

jf_set_default_journal_opts(&jopts);
jopts.flags |= JF_JOURNAL_PROP_SYNC_ENV_VAR;
rc = jf_journal_open(&j, "jf_tut_foo-journal", 2, &jopts);
```

an application that uses `JF_JOURNAL_PROP_SYNC_ENV_VAR` searches for environment variable `JF_JOURNAL_SYNC_TYPE` to establish the type of synchronization must be used:

- environment variable is defined and its value is “0”: *fast* synchronization is adopted
- environment variable is defined and its value is “1”: *safe* synchronization is adopted
- else: `JF_JOURNAL_PROP_SYNC_SUGGESTED` synchronization is adopted (take a look to “API reference guide”)

4.1.4. Playing with synchronization type

Showing the effects of different synchronization type is a hard job out of the scope of this tutorial, but an example to empirically verify the performance gap is easy to build.

Example 4-1. `many_hello_world.c`

```
#include <jf_file.h>

int main()
{
        int rc, i;
        jf_file_t jf;
        size_t write;

        rc = jf_file_open(&jf, NULL, "jf_tut_foo", "w", NULL);
        if (JF_RC_OK != rc)
                return 1;

        for (i = 0; i < 10000; ++i) {
                rc = jf_file_printf(&jf, &write, "%s",
                                    "Hello world!\n");
                if (JF_RC_OK != rc)
                        return 1;

                rc = jf_file_commit(&jf);
                if (JF_RC_OK != rc)
                        return 1;
        } /* for (i = 0; i < 10000; ++i) */

        rc = jf_file_close(&jf);
        if (JF_RC_OK != rc)
                return 1;

        printf("Many hello world program is OK!\n");
        return 0;
}
```

`many_hello_world.c` is like `hello_world.c` but it performs 10000 transactions instead of only 1. We do not specify `JF_JOURNAL_PROP_SYNC_ENV_VAR` because it's the default option. To compile `many_hello_world.c` you can use this command:

```
libtool --mode=link gcc -Wall -I/opt/libjf/include -L/opt/libjf/lib -ljf \
        -o many_hello_world many_hello_world.c
```

execute it:

```
tiian@linux:~/tutorial> rm jf_tut_foo*
tiian@linux:~/tutorial> export JF_JOURNAL_SYNC_TYPE=0
tiian@linux:~/tutorial> time ./many_hello_world
Many hello world program is OK!

real    0m0.499s
user    0m0.149s
sys     0m0.345s

tiian@linux:~/tutorial> rm jf_tut_foo*
tiian@linux:~/tutorial> export JF_JOURNAL_SYNC_TYPE=1
tiian@linux:~/tutorial> time ./many_hello_world
Many hello world program is OK!
```

```
real    0m3.478s
user    0m0.130s
sys     0m0.390s

tiian@linux:~/tutorial> rm jf_tut_foo*
tiian@linux:~/tutorial> unset JF_JOURNAL_SYNC_TYPE
tiian@linux:~/tutorial> time ./many_hello_world
Many hello world program is OK!

real    0m0.507s
user    0m0.173s
sys     0m0.331s
```

second execution take 7 times the first; third execution is very like the first: this means current value of `JF_JOURNAL_PROP_SYNC_SUGGESTED` is `JF_JOURNAL_PROP_SYNC_FAST` but in the future it might be changed. To check the journaled files contains 10000 rows issue this command:

```
tiian@linux:~/tutorial> wc -l jf_tut_foo
10000 jf_tut_foo
```

To check journal contains 10000 commits issue this command:

```
tiian@linux:~/tutorial> jf_report -j jf_tut_foo.jf | grep commit | wc -l
10000
```

Please pay attention **many_hello_world** is *not* a benchmark program! To measure libjf performances, utility program **jf_bench** is supplied, but this is another tale.

4.1.5. How is synchronization tested?

To test synchronization, crashes must be reproduced. Application crash is easy to simulate: a division by zero exception, a segmentation fault exception, etc... Simulating a system crash is a much difficult task; a realistic simulation is probably an impossible task without hacking the operating system kernel. Despite this fact, some types of test must be performed against a “journaled files library”...

libjf implements a “crash simulation feature” used to stress the library with crashes in all the interesting code steps: this simulation should be sufficiently closed to a real crash to declare “libjf should be a safe journaling tools”. Nothing is engraved in the stone and some stuff might be changed in the future.

4.2. Journaling and caching

Designing a super safe journaling tool without keeping in consideration the performance point of view is a useless academic exercise: no one would use a very slow “safe journaling tool” instead of standard I/O libraries. libjf is not already optimized and a lot of code review, in the future, would be probably increase performances, but from an architectural point of view, the library adopt some strategies to limit performance degradation when compared with standard I/O libraries.

The most important feature is a high level cache we can explain with few words: every time the application updates a journaled file, the change is not propagated to the underlining file, but simply kept in the cache managed by libjf. Data are copied to file when cache reaches maximum size or a commit is requested by application. If the cache is large enough, no underlining file is touched until commit point and, in case of rollback, no file is touched at all. Managing a new level of cache is expensive in terms of CPU and virtual memory, but updating files before commit or rollback dramatically increases elapsed times because every time a bit is touched, its undo record must have been saved and synchronized in a safe place (call it “journal”, “log” or “rollback tablespace” does not alter the concept).

If libjf was kernel stuff at filesystem level, its performances would be closer to native file access operations, but a lot of big issues should be solved:

- GNU/Linux has 4 different “official” filesystems: ext3 (and ext2), reiser, xfs, jfs and libjf should have 4 different implementations only for GNU/Linux
- proprietary UNIX are not so easy to hack: there might be problems related to licenses; some proprietary UNIX does not supply kernel source code and a modification would be quite impossible
- not to mention the Microsoft Windows operating system families...

The efficient kernel level implementation of libjf would not exist and we are not discussing about libjf...

The maximum size of cache allocated for every journaled file can be specified setting the field *`cache_size_limit`* of struct jf_journal_opts_s of struct jf_file_open_opts_s. Take a look to this sample program:

Example 4-2. `cache_size.c`

```
#include <jf_file.h>

int main()
{
        int rc;
        jf_journal_t j;
        jf_file_t jf1, jf2, jf3;
        struct jf_journal_opts_s jopts;
        struct jf_file_open_opts_s fopts;

        jf_set_default_journal_opts(&jopts);
        jopts.flags |= JF_JOURNAL_PROP_OPEN_O_CREAT |
                JF_JOURNAL_PROP_OPEN_O_EXCL;
        rc = jf_journal_open(&j, "jf_tut_foo-journal", 2, &jopts);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        jf_set_default_file_open_opts(&fopts);
        fopts.join_the_journal = TRUE;
        fopts.journal_opts.journal_file_opts.cache_size_limit = 123400;

        rc = jf_file_open(&jf1, &j, "jf_tut_foo-data1", "w", &fopts);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        printf("Cache limit for first journaled file: "
               JF_OFFSET_T_FORMAT "\n",
               jf_file_get_cache_limit(&jf1));

        fopts.journal_opts.journal_file_opts.cache_size_limit = -1;

        rc = jf_file_open(&jf2, &j, "jf_tut_foo-data2", "w", &fopts);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        printf("Cache limit for second journaled file: "
               JF_OFFSET_T_FORMAT "\n",
               jf_file_get_cache_limit(&jf2));

        rc = jf_file_open(&jf3, NULL, "jf_tut_foo-data3", "w", NULL);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        printf("Cache limit for third journaled file: "
               JF_OFFSET_T_FORMAT "\n",
               jf_file_get_cache_limit(&jf3));

        rc = jf_file_close(&jf1);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_file_close(&jf2);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_file_close(&jf3);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
        rc = jf_journal_close(&j);
        if (JF_RC_OK != rc) {
                printf("%d/%s\n", rc, jf_strerror(rc));
                return 1;
        }
```

```
    65         printf("two_files program ended OK!\n");
    66         return 0;
    67 }
```

`cache_size.c` source code explanation

Rows 19-20

: set cache size to value 123400 bytes for journaled file `jf1`

Row 27

: retrieve the size of cache associated to journaled file `jf1`

Rows 28-29

: set cache size to default value for journaled file `jf2`

Row 36

: retrieve the size of cache associated to journaled file `jf2`

Row 37

: open journaled file `jf3` with default values

Row 44

: retrieve the size of cache associated to journaled file `jf3`

4.2.1. Compilation and execution

To compile **cache_size** program you can use this command:

```
libtool --mode=link gcc -Wall -I/opt/libjf/include -L/opt/libjf/lib -ljf \
        -o cache_size cache_size.c
```

executed it:

```
tiian@linux:~/src/tutorial> rm jf_tut_foo*
tiian@linux:~/src/tutorial> export JF_JOURNALED_FILE_CACHE_SIZE=765000
tiian@linux:~/src/tutorial> ./cache_size
Cache limit for first journaled file: 123400
Cache limit for second journaled file: 262144
Cache limit for third journaled file: 765000
two_files program ended OK!
tiian@linux:~/src/tutorial> rm jf_tut_foo-*
```

```
tiian@linux:~/src/tutorial> export JF_JOURNALED_FILE_CACHE_SIZE=437900
tiian@linux:~/src/tutorial> ./cache_size
Cache limit for first journaled file: 123400
Cache limit for second journaled file: 262144
Cache limit for third journaled file: 437900
two_files program ended OK!
```

- cache associated to first journaled file is 123400 bytes large and it's the same at first and second execution because it's the value explicitly coded by the program
- cache associated to second journaled file is 262144 bytes large and it's the same at first and second execution because it's the *default* value
- cache associated to third journaled file varies according to the value of `JF_JOURNALED_FILE_CACHE_SIZE`; the same behavior can be obtained avoiding explicit setting of *`cache_size_limit`* in jf_file_open_opts_s struct.

4.2.2. How cache size limit can be tuned

After you developed your application you can try to expand the cache size limit and measure elapsed times: only if the performance improves significantly the cache size expansion is suggested. For most applications, default value should be fine.

> **Note:** the parameter has the meaning of "cache size limit": only *necessary* memory are allocated by the application.

4.3. libjf object options

In the previous sections we inspected some options can be set when opening a journal and/or a journaled file. To get the complete up-to-date list of available options, please refer to "API reference guide". The following table is a summary of the structs used to pass options to create/open methods:

Table 4-1. Create/open methods struct summary

Method	Struct	Sub-struct	Sub-sub-struct
`jf_journal_open`	jf_journal_opts_s	jf_journal_file_opts_s	
`jf_file_open`	jf_file_open_opts_s	jf_journal_opts_s	jf_journal_file_opts_s

These option structs are passed by reference, but the content is not changed because they are read only arguments of open methods.

Chapter 5. Utility programs

libjf is not only a *library* (static and or shared), but a complete tool to develop and manage journaled applications. As seen in the previous chapter, some utility programs are supplied to help in journal management.

5.1. jf_create: journal creation

If you have to create a journal without writing your own “hello world derived application”, you can use utility program **jf_create**. This is a usage example:

```
tiian@linux:~/tutorial> jf_create -j jf_tut_foo-journal -n 5
tiian@linux:~/tutorial> ls -la jf_tut_foo-journal
-rw-r--r--  1 tiian users 8278 2005-08-29 22:12 jf_tut_foo-journal
tiian@linux:~/tutorial> jf_report -j jf_tut_foo-journal
<?xml version="1.0" encoding="UTF-8"?>
<journal>
  <header magic_number='0x41524153' version='1' file_id_mask='0x38'
      file_id_mask_shift='3' size_mask='0xffffffc0' size_mask_shift='6'
      file_size='4194304' file_num='3' rotation_threshold='0.800'
      ctrl_recs='36' journal_recs='32980' />
  <journaled_file_table max_files='8' number_of_files='1'
      file_table='0x804b170'>
    <file id='0' name='jf_tut_foo-journal' last_pos='32980' last_size='0'
        status='0' last_uc_pos='0' last_uc_size='32980' stream='0x804b008' />
  </journaled_file_table>
  <records>
  </records>
</journal>
```

a journal able to manage at least 5 journaled files has been created. Looking at the output produced by **jf_report** we can see the created journal can manage up to 7 (8 - 1) journaled files. First journaled file is reserved because it’s the journal itself.

5.2. jf_join: join a journal

Sometimes you have a journal and a standard, not journaled, file you would like to use as a journaled file: you can use **jf_join** utility program to add your file to the list of journaled files managed by the journal. This is a usage example (the journal has been created at the previous paragraph):

```
tiian@linux:~/tutorial> ls -la > jf_tut_foo-data1
tiian@linux:~/tutorial> ls -la jf_tut_foo-*
```

```
-rw-r--r--  1 tiian users 2038 2005-08-29 22:22 jf_tut_foo-data1
-rw-r--r--  1 tiian users 8278 2005-08-29 22:12 jf_tut_foo-journal
tiian@linux:~/tutorial> jf_join -j jf_tut_foo-journal jf_tut_foo-data1
tiian@linux:~/tutorial> jf_report -j jf_tut_foo-journal
<?xml version="1.0" encoding="UTF-8"?>
<journal>
  <header magic_number='0x41524153' version='1' file_id_mask='0x38'
      file_id_mask_shift='3' size_mask='0xffffffc0' size_mask_shift='6'
      file_size='4194304' file_num='3' rotation_threshold='0.800'
      ctrl_recs='36' journal_recs='32980' />
  <journaled_file_table max_files='8' number_of_files='2'
      file_table='0x804b170'>
    <file id='0' name='jf_tut_foo-journal' last_pos='32980' last_size='0'
        status='0' last_uc_pos='0' last_uc_size='32980' stream='0x804b008' />
    <file id='1' name='jf_tut_foo-data1' last_pos='0' last_size='791'
        status='0' last_uc_pos='0' last_uc_size='0' stream='(nil)' />
  </journaled_file_table>
  <records>
  </records>
</journal>
```

file `jf_tut_foo-data1` has been joined to journal `jf_tut_foo-journal` and can now be used with libjf API.

5.3. jf_rename: rename a journaled file

A journaled file can not be renamed using standard operating system command (**mv** if you are playing in a UNIX-like environment) because journal must be updated with the new name. Utility program **jf_rename** has been designed to help you when a journaled file rename must be performed:

```
tiian@linux:~/tutorial> jf_rename -j jf_tut_foo-journal -n jf_tut_foo-data2 \
> jf_tut_foo-data1
tiian@linux:~/tutorial> jf_report -j jf_tut_foo-journal
<?xml version="1.0" encoding="UTF-8"?>
<journal>
  <header magic_number='0x41524153' version='1' file_id_mask='0x38'
      file_id_mask_shift='3' size_mask='0xffffffc0' size_mask_shift='6'
      file_size='4194304' file_num='3' rotation_threshold='0.800'
      ctrl_recs='36' journal_recs='32980' />
  <journaled_file_table max_files='8' number_of_files='2'
      file_table='0x804b170'>
    <file id='0' name='jf_tut_foo-journal' last_pos='32984' last_size='0'
        status='0' last_uc_pos='0' last_uc_size='32984' stream='0x804b008' />
    <file id='1' name='jf_tut_foo-data2' last_pos='0' last_size='791'
        status='0' last_uc_pos='0' last_uc_size='0' stream='(nil)' />
  </journaled_file_table>
  <records>
    <rollback jrn_rec_off='32980' file_id='0'/>
  </records>
```

```
</journal>
```

5.4. jf_leave: leave a journal

Sometimes you have to update a journaled file with a tool that’s not libjf enabled: there are a lot of them around the world... Take a look to this example:

```
tiian@linux:~/tutorial> echo "John" > jf_tut_foo-data3
tiian@linux:~/tutorial> echo "Patty" >> jf_tut_foo-data3
tiian@linux:~/tutorial> cat jf_tut_foo-data3
John
Patty
tiian@linux:~/tutorial> jf_join -j jf_tut_foo-journal jf_tut_foo-data3
tiian@linux:~/tutorial> jf_report -j jf_tut_foo-journal
<?xml version="1.0" encoding="UTF-8"?>
<journal>
  <header magic_number='0x41524153' version='1' file_id_mask='0x38'
      file_id_mask_shift='3' size_mask='0xffffffc0' size_mask_shift='6'
      file_size='4194304' file_num='3' rotation_threshold='0.800'
      ctrl_recs='36' journal_recs='32980' />
  <journaled_file_table max_files='8' number_of_files='3'
      file_table='0x804b170'>
    <file id='0' name='jf_tut_foo-journal' last_pos='32988' last_size='0'
        status='0' last_uc_pos='0' last_uc_size='32988' stream='0x804b008' />
    <file id='1' name='jf_tut_foo-data2' last_pos='0' last_size='1075'
        status='0' last_uc_pos='0' last_uc_size='0' stream='(nil)' />
    <file id='2' name='jf_tut_foo-data3' last_pos='0' last_size='11'
        status='0' last_uc_pos='0' last_uc_size='0' stream='(nil)' />
  </journaled_file_table>
  <records>
    <rollback jrn_rec_off='32980' file_id='0'/>
    <rollback jrn_rec_off='32984' file_id='0'/>
  </records>
</journal>
```

one second after **jf_join** we have realized our journaled file `jf_tut_foo-data3` must be fixed. We can not update the journaled file using shell tools because journal would not be aware of them: we have to temporarily detach the journaled file from journal:

```
tiian@linux:~/tutorial> jf_leave -j jf_tut_foo-journal jf_tut_foo-data3
tiian@linux:~/tutorial> jf_report -j jf_tut_foo-journal
<?xml version="1.0" encoding="UTF-8"?>
<journal>
  <header magic_number='0x41524153' version='1' file_id_mask='0x38'
      file_id_mask_shift='3' size_mask='0xffffffc0' size_mask_shift='6'
      file_size='4194304' file_num='3' rotation_threshold='0.800'
      ctrl_recs='36' journal_recs='32980' />
  <journaled_file_table max_files='8' number_of_files='2'
      file_table='0x804b170'>
    <file id='0' name='jf_tut_foo-journal' last_pos='32984' last_size='0'
        status='0' last_uc_pos='0' last_uc_size='32984' stream='0x804b008' />
    <file id='1' name='jf_tut_foo-data2' last_pos='0' last_size='1075'
        status='0' last_uc_pos='0' last_uc_size='0' stream='(nil)' />
  </journaled_file_table>
  <records>
    <rollback jrn_rec_off='32980' file_id='0'/>
  </records>
</journal>
```

now we can update and join again:

```
tiian@linux:~/tutorial> echo "Roger" >> jf_tut_foo-data3
tiian@linux:~/tutorial> echo "Kelly" >> jf_tut_foo-data3
tiian@linux:~/tutorial> jf_join -j jf_tut_foo-journal jf_tut_foo-data3
tiian@linux:~/tutorial> jf_report -j jf_tut_foo-journal
<?xml version="1.0" encoding="UTF-8"?>
<journal>
  <header magic_number='0x41524153' version='1' file_id_mask='0x38'
      file_id_mask_shift='3' size_mask='0xffffffc0' size_mask_shift='6'
      file_size='4194304' file_num='3' rotation_threshold='0.800'
      ctrl_recs='36' journal_recs='32980' />
  <journaled_file_table max_files='8' number_of_files='3'
      file_table='0x804b170'>
    <file id='0' name='jf_tut_foo-journal' last_pos='32988' last_size='0'
        status='0' last_uc_pos='0' last_uc_size='32988' stream='0x804b008' />
    <file id='1' name='jf_tut_foo-data2' last_pos='0' last_size='1075'
        status='0' last_uc_pos='0' last_uc_size='0' stream='(nil)' />
    <file id='2' name='jf_tut_foo-data3' last_pos='0' last_size='22'
        status='0' last_uc_pos='0' last_uc_size='0' stream='(nil)' />
  </journaled_file_table>
  <records>
    <rollback jrn_rec_off='32980' file_id='0'/>
    <rollback jrn_rec_off='32984' file_id='0'/>
  </records>
</journal>
```

Warning

If you update a journaled file with a program that does not use libjf API, *your data can be loss* when a libjf based application opens the journal and/or access the specific journaled file. If you are planning to use the files produced by libjf enabled application in a legacy environment, batch procedures to “leave & update & join” journal must be implemented.

5.5. jf_report: inspecting a journal

We've done it many times through this tutorial using **jf_report** utility program; the only info you may need is the `-d` flag meaning: you can specify if data must be showed as hexadecimal, text or both. Use `-h` option to show a brief help.

5.6. jf_recover: recover a journal

To recover a journal after an application/system crash you should use **jf_recover** utility program we have discussed in a previous chapter (see Section 3.4). The only warning we can suggest you is this:

Warning

using a journaling tool like libjf does not mean you don't need your old friend backup tool! Please pay attention: journal too must be backed up.

5.7. jf_bench: performance measurement

Utility program **jf_bench** is specifically designed to measure libjf absolute performances and to compare the performance of libjf with standard C I/O. The detailed description of this utility is outside the scope of the tutorial, but you can find out more information in FAQ distributed with package and/or available online.

Chapter 6. Debugging applications

"A bug is a test case you haven't written yet."

Mark Pilgrim - "Dive Into Python"

Writing source code is only the first step in application development. In the real life a programmer is a debugger, not a coder.

6.1. `printf` approach

The first debugging tool is `printf` function. Most libjf function returns a "return code" of type int. File `jf/jf_errors.h` contains all the available return codes, but you probably would a more human readable error code than an integer value. Function `jf_strerror` returns a description for every return code documented in `jf/jf_errors.h`. This is a little usage example:

Example 6-1. `jf_strerror.c`

```
#include <jf_file.h>

int main()
{
        int rc;
        jf_file_t jf;

        rc = jf_file_close(&jf);
        if (JF_RC_OK != rc) {
                fprintf(stderr, "libjf error: %s (%d)\n",
                        jf_strerror(rc), rc);
                return 1;
        }
        return 0;
}
```

You can compile it with this command:

```
libtool --mode=link gcc -Wall-I/opt/libjf/include -L/opt/libjf/lib -ljf \
        -o jf_strerror jf_strerror.c
```

Trying to execute this foolish program you should get an error like this one:

```
tiian@linux:~/tutorial> ./jf_strerror
```

```
libjf error: ERROR: object is corrupted (-9)
```

6.1.1. Error codes' rule of thumb

`JF_RC_OK` is the return code every API should return. Warning values are values larger than `JF_RC_OK`. Error values are values smaller than `JF_RC_OK`.

Warning

You should never use numeric constant when checking libjf API return codes.

6.2. The trace approach

Many times the code returned by the invoked function is not sufficient to understand what's happening: libjf is developed using an "exception oriented programming style" I named "sequential programming" some times ago. The idea at the root of this programming style is: "nidification is a bad thing, try to write code with as small nidification as possible".

To see the "stack trace" of the called function, you must do two things:

- activate "debug" feature when building libjf:

  ```
  ./configure --enable-debug
  make
  make check
  sudo make install
  ```

- set environment variable `JF_TRACE_MASK` before start your program:

  ```
  export JF_TRACE_MASK=0xffffffff
  ```

Once you performed these two steps, you can run your program again:

```
tiian@linux:~/tutorial> ./jf_strerror
jf_file_close
jf_file_close/excp=0/ret_cod=-9/errno=0
libjf error: ERROR: object is corrupted (-9)
```

When running a complex application, mask “0xffffffff” will produce a lot of messages, too many messages. To determine the value you need, take a look to file `jf/jf_trace.h`: you can activate/deactivate the trace feature at module level and get only the messages you need. This is an excerpt of `jf/jf_trace.h`:

```
[...]
/**
 * trace module for library module "jf_cache_file"
 */
#define JF_TRACE_MOD_LIB_CACHE_FILE                                   0x00000001

/**
 * trace module for library module "jf_crash_simul"
 */
#define JF_TRACE_MOD_LIB_CRASH_SIMUL                                  0x00000002

/**
 * trace module for library module "jf_file"
 */
#define JF_TRACE_MOD_LIB_FILE                                         0x00000004

/**
 * trace module for library module "jf_journal_file_tab"
 */
#define JF_TRACE_MOD_LIB_JOURNAL_FILE_TAB                             0x00000008

/**
 * trace module for library module "jf_utils"
 */
#define JF_TRACE_MOD_LIB_UTILS                                        0x00000010

/**
 * trace module for library module "jf_journal"
 */
#define JF_TRACE_MOD_LIB_JOURNAL                                      0x00000020
[...]
```

if you need only messages printed by “jf_journal” and “jf_file” modules, you have to set `JF_TRACE_MASK` to “0x00000024”. Any combination of values is allowed.

Warning

Don’t use a library built with “debug” feature in production environment: performances may degrade even if `JF_TRACE_MASK` is not set.

6.2.1. How can I guess if libjf was compiled with debug feature?

If you don’t remember how libjf was compiled or someone but you installed it, you can retrieve configuration features with this command:

```
tiian@linux:~> strings /opt/libjf/lib/libjf | grep ’feature/’
feature/timer = yes
feature/debug = yes
feature/crash_simul = no
feature/cache_stress = no
feature/extra_check = no
```

pay attention the actual name of the library is system dependent: it could be `libjf.a`, `libjf.so`, `libjf.sl`, etc...

6.3. The debugger approach

Sometimes the bug is very hard to discover and/or to fix and something more than `printf` and trace must be used. If you need to debug your application using a tool like **gdb**, you must build libjf activating debug feature as shown in Section 6.2.

Appendix A. GNU Free Documentation License

A.1. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.2. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of

A.3. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.4. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.6. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

A.7. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is

rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.11. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.12. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Sample Invariant Sections list

Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

Sample Invariant Sections list

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public