Displaying and Watching directories using Lazarus

Michaël Van Canneyt

October 31, 2011

Abstract

Using Lazarus, getting the contents of a directory can be done in 2 ways: a portable, and a unix-specific way. This article shows how to get the contents of a directory and show it in a window. Additionally, it shows how to get notifications of the Linux kernel if the contents of the directory changes.

1 Introduction

Examining the contents of a directory is a common operation, both using command-line tools or a GUI file manager. Naturally, Free/Pascal and Lazarus offer an API to do this. In fact, there are 2 API's to get the contents of a directory: one which is portable and will work on all platforms supported by Lazarus. The other is not portable, but resembles closely the POSIX API for dealing with files and directories. Each API has its advantages and disadvantages.

Often, it is desirable to be notified if the contents of a directory changes: in a file manager, this can be used to update the display - showing new items or removing items as needed. This can also be done by scanning the contents of the directory at regular intervals, but it should be obvious that this is not as efficient.

There are other scenarios when a notification of a change in a directory is interesting: for instance, in a FTP server, one may want to move incoming files to a location outside the FTP tree, or to a new location based on some rules (e.g. images to one directory, sound files to another). Or one may simply wish to keep 2 directories synchronized for backup purposes. In all these cases, a regular scan of the directory is an option, but far better is to have an event-driven mechanism, which only springs into action whenever something has actually changed.

The Linux kernel offers a mechanism to receive notifications when the contents of a directory is changed: it is called *inotify*, and this is the mechanism which will be discussed here.

The various APIs will be explaine in a small GUI application which simply shows the contents of a directory in a listview, and which updates the display whenever the directory contents change.

2 The cross-platform way

The sysutils unit of Free Pascal contains the necessary structures and functions to read the contents of a directory. The API involves 3 functions:

```
Function FindFirst(Const Path: String; Attr: Longint;
```

```
                    out Rslt: TSearchRec): Longint;
Function FindNext(Var Rslt: TSearchRec) : Longint;
Procedure FindClose(Var F: TSearchrec);
```

Each function takes at least one argument of type `TSearchRec`, which has the following declaration:

```
TSearchRec = Record
  Time : Longint;
  Size : Int64;
  Attr : Longint;
  Name : TFileName;
  Mode : TMode;
end;
```

There are some other fields, but they are for internal use only, and should not be used in end-user code. The record describes one entry (file or directory), found in the directory.

The three functions must be used in the order as shown in the declaration above:

FindFirst starts a new directory scan operation. The `Path` argument should contain a full path to the directory, and a filename mask. The directory scan will only return filenames that match the mask. To retrieve all files, the '*' mask can be used. The `Attr` argument is a OR-ed mask of file attributes which should be returned *in addition to normal files*. A list is shown in table 1. The function returns 0 if the scan operation was started successfully, and an entry matching the criteria was found. On return the `Rslt` argument will be filled with the information for the first match. If something went wrong during start of the search operation, a nonzero error code is returned.

FindNext will return the next match for the directory scan. It returns zero if a next entry was successfully found. It returns a nonzero error code in case an error was encountered, or no more entries are available.

FindClose must always be called after a succesful call to `FindFirst`: it will free the resources allocated by `FindFirst` for the directory scan.

The entries found in the directory are reported in a `TSearchRec` record. The meaning of the fields speak for themselves, but there are some things to note:

- The `Time` field is actually a timestamp. It can be converted to a `TDateTime` value (used in all date/time calculations) with the `FileDateToDateTime` function.
- The `Attr` field is an OR-ed combination of the values in table 1

A typical routine to scan the contents of a directory looks like the following `repeat..until` loop:

```
procedure TForm1.OpenPortableDirectory
   (Const ADirectory: string);

Var
  Info : TSearchRec;
  DN : String;

begin
  DN:=IncludeTrailingPathDelimiter(ADirectory);
```

Table 1: File attributes

Attribute	Meaning
faReadOnly	Read-only file
faHidden	hidden file (starts with .)
faSysFile	System file (device, socket or fifo)
faVolumeId	Unused
faDirectory	Directory
faArchive	Always set on Unices
faSymLink	Symnlink
faAnyFile	Use in FindFirst to return all files

```
  if FindFirst(DN+AllFilesMask,faAnyFile,Info)=0 then
    try
      Repeat
        AddDirectoryEntry(Info);
      Until FindNext(Info)<>0;
    finally
      FindClose(info);
    end;
end;
```

The `FindFirst` call starts the scan operation, and in the code above specifies that all possible files must be returned by the scan: the `AllFilesMask` and `faAnyfile` constants are made for this. If only log files were needed, then `'*.log'` could have been specified instead of `AllFilesMask`.

In the directory viewer application, the `AddDirectoryEntry` takes the `TSearchRec` instance, and adds its contents to a collection of directory entries, which is later shown in the form. In other applications, this call will be replaced with whatever logic is needed for the particular application.

3 The unix way

The same directory scan routine can be coded with Unix-specific routines as well. 4 routines are needed for this:

```
Function  FpOpendir(dirname : AnsiString): pDir;
Function  FpReaddir(var dirp : Dir) : pDirent;
Function  FpClosedir(var dirp : Dir): cInt;
Function  FpStat(path: String; var buf : stat): cInt;
```

The first three functions resemble the portable functions. Indeed, they serve roughly the same purpose:

FpOpendir starts a directory scan operation. The `DirName` argument specifies the directory to scan. The function returns a pointer to an (opaque) structure, or `Nil` if the call failed. The pointer must be used in the subsequent `fpReadDir` and `fpCloseDir` operations.

FpReaddir Returns the next entry from the directory scan. It is a pointer to a `DirEnt` record:

```
Dirent = packed record
```

```
    d_fileno : ino64_t;   // file number of entry
    d_reclen : cushort;   // length of string in d_name
    d_type   : cuchar;    // file type, see below
    d_name   : array[0..255] of char;  // File name
  end;
```

if no more entries are available, the function returns `Nil`. The contents of this `DirEnt` record must be copied if they are to be used later: a call to `FpReadDir` or `FpCloseDir` will invalidate the contents.

FpClosedir must always be called after a succesful call to `FpOpenDir`: it will free the resources allocated by `FpOpenDir` for the directory scan.

The above calls only return the names of the files in the directory. No additional information such as file size or time stamp is available. To get these, an addional call to `FpStat` is needed. `FpStat` returns information about a file in the file system in a `Stat` record:

```
Stat = packed record
  st_dev,  // Device ID
  st_ino,  // Inode number
  st_nlink  : qword; // Number of links
  st_mode, // Mode
  st_uid,  // Owner user ID
  st_gid,  // Group ID
  st_rdev   : qword; // Device id for special file
  st_size,  // Size of file
  st_blksize, // Block size
  st_blocks : int64; // Number of blocks.
  st_atime,    // Access time
  st_mtime,    // Modification time
  st_ctime : qword; // Creation time
end;
```

It contains the information that was present in the cross-platform `TSearchRec` record, and more: file ownership, various timestamps. Note that not all fields are meaningful for all filesystems. For example FAT32 filesystems do not have different filestamps, or file ownerships.

Using the above four routines, the directory scan would look as follows:

```
procedure TForm1.OpenUnixDirectory(Const ADirectory : string);

Var
  ADir : PDir;
  de : PDirEnt;
  dn,fn : string;
  Info : stat;
begin
  ADir:=fpOpenDir(ADirectory);
  dn:=IncludeTrailingPathDelimiter(ADirectory);
  if (ADir<>Nil) then
    try
      de:=fpReadDir(ADir^);
      While (de<>Nil) do
        begin
```

```
        fn:=de^.d_name;
        if fpStat(dn+FN,info)=0 then
          AddDirectoryEntry(FN,Info);
        de:=fpReadDir(ADir^);
        end;
    finally
      fpCloseDir(ADir^);
    end;
end;
```

The routine resembles the portable routine, except that an additional call to `fpStat` is needed. Note that `fpStat` may return an error code if the file which was being scanned was deleted between the time of the `FpReadDir` and `fpStat` calls, so this must be checked.

The Unix version of the portable API uses the above 4 calls to implement the portable API; It adds the filtering on name and attributes, but for instance does not return the group and owner IDs, nor does it keep the device or inode information.

4 Displaying a directory

The sample application that comes with this article shows the contents of a directory in a list view. To this end, it stores the directory entries in a collection of the following items:

```
TDirectoryEntry = Class(TCollectionItem)
  Property Name : String;
  Property TimeStamp : TDateTime;
  Property Size : Int64;
  Property Attributes : Integer;
  Property Mode : mode_t;
  Property Owner : uid_t;
  Property Group : gid_t;
end;
```

The structure allows to store the basic information found in a unix system.

The following routine shows the contents of the collection (named `Fentries`). It is called after the directory contents was read using one of the routines presented in the previous paragraphs:

```
procedure TForm1.ShowDirectory;

Var
  I : integer;
  Li : TListItem;

begin
  FEntries.Sort(@CompareDirs);
  With LVDir.Items do
    begin
    BeginUpdate;
    try
      Clear;
      For I:=0 to FEntries.Count-1 do
```

```
        begin
        LI:=LVDir.Items.Add;
        LI.Data:=FEntries[i];
        UpdateItem(Li);
        end;
    finally
      EndUpdate;
    end;
    end;
end;
```

The routine starts by sorting the entries. It then clears the current list (shown in the TListView component `LVDir`) and populates the list inside a `BeginUpdate`..`EndUpdate` block for efficiency. The `UpdateItem` takes care of copying the needed information of the `TDirectoryItem` collection to the `TListItem` items used in the listview:

```
procedure TForm1.UpdateItem (LI : TListItem);

Var
  E : TDirectoryEntry;

begin
  E:=TDirectoryEntry(Li.Data);
  LI.Caption:=E.Name;
  With LI.SubItems do
    begin;
    BeginUpdate;
    try
      Clear;
      Add(DateTimeToStr(E.TimeStamp));
      Add(FileSizeString(E.Size));
      Add(FilePermissionString(E.Mode));
      if FPortable then
        begin
        Add('?');
        Add('?');
        end
      else
        begin
        Add(FileOwnerString(E.Owner));
        Add(FileGroupString(E.Group));
        end;
    finally
      EndUpdate;
    end;
    end;
end;
```

The `FPortable` boolean indicates whether the portable version of the calls was used or not: if not, then the owner and group of the files are shown. The routines `FileSizeString`, `FilePermissionString`, `FileOwnerString`, `FileGroupString` are auxiliary routines to convert numerical arguments to human-readable strings. The result of all this is shown in figure 1 on page 7

Figure 1: The directory watch program in action

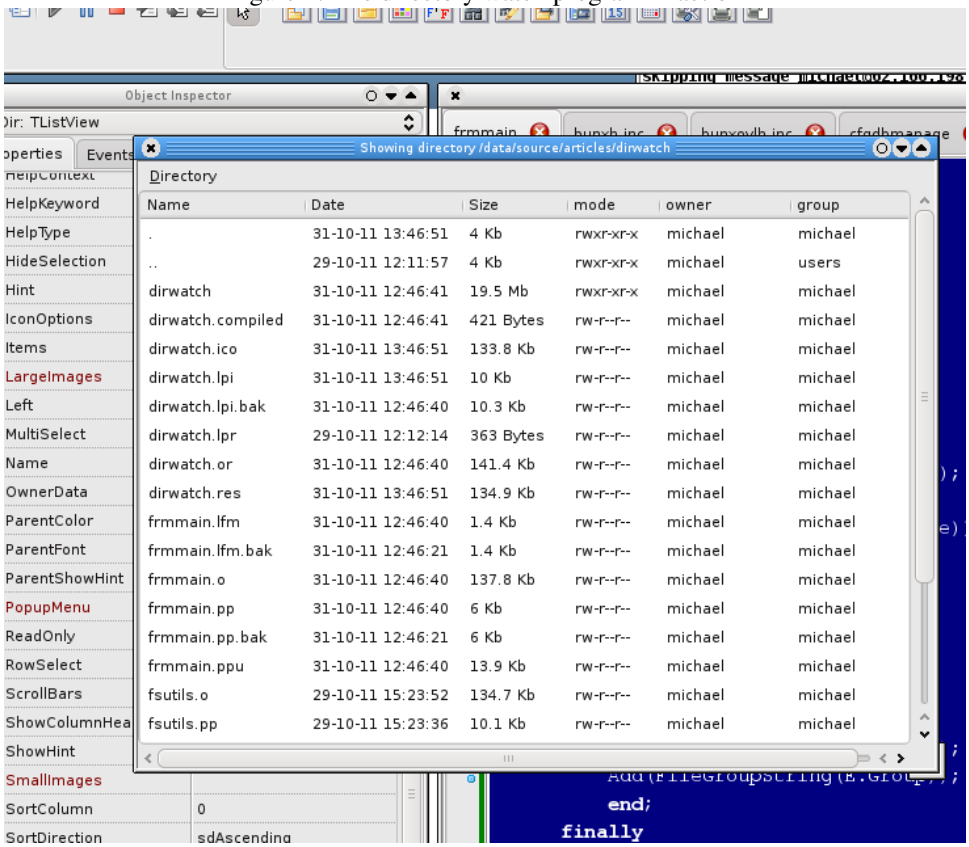

Table 2: Event masks used in `inotify`

Event	Meaning
IN_ACCESS	Data was read from file
IN_MODIFY	Data was written to file
IN_ATTRIB	File atrributes were changed
IN_CLOSE_WRITE	File opened for write was closed
IN_CLOSE_NOWRITE	File opened for read was closed
IN_OPEN	File was opened
IN_MOVED_FROM	File was moved away from watch
IN_MOVED_TO	File was moved to watch
IN_CREATE	A file was created.
IN_DELETE	A file was deleted.
IN_DELETE_SELF	The watch itself was deleted.

5 Monitoring for changes

To monitor for changes in directories, the Linux kernel offers 2 mechanisms: the deprecated `dnotify`, and the newer and more versatile `inotify` mechanism, which appeared in kernel version 2.6.13. The inotify mechanism is based on the following calls:

```
function inotify_init: cint;
function inotify_init1(flags:cint):cint;
function inotify_add_watch(fd:cint; name:Pchar; mask:cuint32):cint;
function inotify_rm_watch(fd:cint; wd: cint):cint;
```

and works as follows:

- First, the `inotify_init1` call is used to obtain a file descriptor. It will be used for all directory or file monitoring operations. Like any other file descriptor, it should be closed with `fpClose` when it is no longer needed. The `flags` parameter can be set to an OR-ed combination of `IN_NONBLOCK` (do not block on read) or `IN_CLOEXEC` (close descriptor on exec) flags. The `inotify_init` call is the same, but with a default value for flags of 0.
- For each directory or file that is to be monitored, an entry is added to the `inotify` file descriptor using the `inotify_add_watch` call. It specifies a filename and mask of events to watch for. A list of possible events is shown in 2, they can be OR-ed together. The functions returns a watch descriptor which can be used to filter events or remove the watch from the inotify system.
- When a watch is no longer needed, the `inotify_rm_watch` call can be used to remove it from the inotify system. The watch descriptor returned by `inotify_add_watch` must be passed to this call.

The above calls can be used to manage the various watches, but do not describe how to be notified if something actually changes.

The kernel adds events to the inotify queue in real-time, and a read operation on the file descriptor returned by `inotify_init1` will remove any events that are in the queue. The file descriptor returned by `inotify_init1` acts as a real file descriptor, i.e. it can be used in a `select()` or `poll()` operation: these calls will report if any data is available for reading, which makes it perfect for integration in a GUI loop, or for use in a background daemon. If so desired, the daemon will block while waiting for an event to occur.

The kernel adds items to the inotify queue in the form of records of the following form:

```
inotify_event = record
 wd     : cint;    // Watch descriptor on which event occurred.
 mask   : cuint32; // OR-ed mask of events.
 cookie : cuint32; // Link together move events
 len    : cuint32; // Filename length, includes padding #0 chars
 name   : char;    // First character of name.
end;
```

Reading the file descriptor will read these records in binary form: enough space must be reserved to receive a complete record. Note that the kernel writes the blocks to the event queue in an aligned manner: this means that the name is appended with zeroes to the following alignment boundary (16 bytes on a 64-bit system). The `len` field gives the length of `name` including all padding bytes, so to get the real length of the name, the padding zeroes must be substracted.

The following small command-line program shows how to watch a directory:

```
Procedure WatchDirectory(d : string);

Const
  Events = IN_MODIFY or IN_ATTRIB or IN_CREATE or IN_DELETE;

Var
  fd, wd,fnl,len : cint;
  fds : tfdset;
  e : ^inotify_event;
  buf : Array[0..1023*4] of Byte; // 4K Buffer
  fn : string;
  p : pchar;

begin
  fd:=inotify_init;
  try
    wd:=inotify_add_watch(fd,pchar(d),Events);
```

The above sets up the watch descriptor, and watches for any file modification notifications in the directory `d`.

The next step is to set up for a `fpSelect` call, telling it to watch the file descriptor for read events. As soon as data becomes available, it will be read:

```
    fpFD_Zero(fds);
    fpFD_SET(fd,fds);
    While (fpSelect(fd+1,@fds,nil,nil,nil)>=0) do
      begin
      len:=fpRead(fd,buf,sizeof(buf));
```

For a daemon that monitors changes a directory, the daemon can wait till something actually changes in the directory. This is accomplished by not specifying a timeout in the `fpSelect` call (i.e. passing `Nil` as the last argument).

Available data is read in a 4Kb. buffer. The actual size can be something else, as long as it is large enough to keep a record written by the kernel. For directories that are in active use, it is better to use a larger buffer, so the kernel buffer for events is emptied faster. Failing to do so will result in the kernel dropping events.

When the available data is read in the buffer, it can be parsed and analysed:

```
      e:=@buf;
      While ((pchar(e)-@buf)<len) do
        begin
        fnl:=e^.len;
        if (fnl>0) then
          begin
          p:=@e^.name+fnl-1;
          While (p^=#0) do
            begin
            dec(p);
            dec(fnl);
            end;
          end;
        setlength(fn,fnl);
        if (fnl>0) then
          move(e^.name,fn[1],fnl);
```

At this point, `fn` contains the filename, and `e` is positioned on the `inotify_event` record. The command-line program simply displays the event it received:

```
        Writeln('Change ',e^.mask,' (',
                InotifyEventsToString(e^.mask),
                ') detected for file "',fn,'"');
        ptrint(e):=ptrint(e)+sizeof(inotify_event)+e^.len-1;
        end;
      end;
  finally
    fpClose(fd);
  end;
end;
```

The finally block closes the inotify file descriptor. The output can look something like this:

```
home: >./tinotify
Change 256 (File created) detected for file "newfile"
Change 2 (File modified) detected for file "newfile"
Change 2 (File modified) detected for file "newfile"
```

The events were triggered by the following command in another terminal:

```
( echo "aha " && echo "more" ) > newfile
```

6 GUI loop integration

To integrate the inotify mechanism in a GUI program requires a bit more work: if the code above is inserted as-is in the `OnShow` event of the main form, then the form would not be drawn, and the program would not respond to events: the program would be stuck in the select() loop. 2 things need to be done to make it work properly:

- Put a timeout on the call to `fpSelect` so it will return if no data is available within a short interval.

- Make sure the `fpSelect` call is executed on a regular basis - for instance when the program becomes idle.

The following `Application.OnIdle` handler does all this:

```
procedure TForm1.CheckDirChanges(Sender: TObject;
                                 var Done: Boolean);

Var
  fnl,len : cint;
  e : ^inotify_event;
  buf : Array[0..1023*4] of Byte; // 4K Buffer
  fn : string;
  p : pchar;
  fds : tfdset;
  Timeout : ttimeval;

begin
  Done:=true;
  fpFD_Zero(fds);
  fpFD_SET(fd,fds);
  timeout.tv_sec:=0;
  timeout.tv_usec:=10;
  if (fpSelect(fd+1,@fds,nil,nil,@Timeout)<=0) then
    exit;
  len:=fpRead(fd,buf,sizeof(buf));
  e:=@buf;
  While ((pchar(e)-@buf)<len) do
    begin
    fnl:=e^.len;
    if (fnl>0) then
      begin
      p:=@e^.name+fnl-1;
      While (p^=#0) do
        begin
        dec(p);
        dec(fnl);
        end;
      end;
    setlength(fn,fnl);
    if (fnl>0) then
      move(e^.name,fn[1],fnl);
    HandleInotifyEvent(e,fn);
    ptrint(e):=ptrint(e)+sizeof(inotify_event)+e^.len-1;
    end;
end;
```

The routine looks the same as in the command-line program, except that the `fpSelect` call now specifies a timeout.

The `HandleInotifyEvent` is where the program reacts on the inotify events:

```
procedure TForm1.HandleInotifyEvent(E : Pinotify_event;
                                    Const FN : String);
```

```
Var
  Msg : string;
  I : Integer;
  DE : TDirectoryEntry;

begin
  Msg:=Format('Change %d (%s) for file "%s".',
              [e^.mask,
               InotifyEventsToString(e^.mask),
               fn]);
  MLog.Lines.Add(Msg);
  if (fn<>'') then
    begin
    I:=Fentries.IndexOfEntry(FN);
    If (I=-1) then
      begin
      // New file created, add entry
      if (E^.Mask and IN_CREATE)=IN_CREATE then
        UpdateDirectoryEntry(FEntries.AddEntry(fn));
      end
    else
      // File deleted, remove entry
      if (E^.Mask and IN_DELETE)=IN_DELETE Then
        FEntries.Delete(I)
      else
        // Modified file, update entry.
        UpDateDirectoryEntry(FEntries[i]);
    end;
  ShowDirectory;
end;
```

The first two lines log the event: a line is added to a memo component on the form. The remainder of the routines tries to update the `FEntries` collection with directory entries: it looks for the entry based on the filename received from Inotify, and then updates the entry in a way that is as optimal as possible, depending on what event was reported by inotify. As the last line, the display is updated by displaying all entries again. This could be made more efficient by just updating the changed items in the list. The result can look as in figure 2 on page 13

7 conclusion

Getting the contents of a directory is an easy task in Free Pascal. Accessing the linux kernel for notification of file changes can help to improve the responsiveness of an application to changes in a directory, or help preserve system resources by not unnecessary scanning the contents of a directory. Both techniques should be in the standard toolbox of a unix system administrator.

Figure 2: Updates in the directory watch program