Requirements

The following new DisOrder features are desirable:

- pausing tracks. This should have minimum latency and should allow the sound device to be closed while paused so that other things can use it.
- eliminating the gap between tracks
- broadcasting over a network

It should remain possible to use unpausible players, bizarro lash-ups, etc.

Network broadcast (multicast?) is mentioned only as something we don't want to rule out in this design, rather than something that is ruled in.

Left To Implement

- pausing
  - for pausable standalone (implemented, untested)
- pre-decoding
- test ao driver against mpg321; see if --signal required
- docs (doing well)
- general testing
- mcdp pre-release
- release

Design

Extended Plugin Interface

```
unsigned long disorder_player_type;

#define DISORDER_PLAYER_STANDALONE 0x00000000
/* this player plays sound directly */
```

```
#define DISORDER_PLAYER_PAUSES    0x00000001
/* standalone player that supports pausing */

#define DISORDER_PLAYER_RAW       0x00000002
/* player that sends raw samples to $DISORDER_RAW_FD */

#define DISORDER_PLAYER_TYPEMASK  0x0000000f
/* mask for player types */

#define DISORDER_PLAYER_PREFORK   0x00000010
/* call prefork function */

void *disorder_play_prefork(const char *track);
/* called outside the fork.  Should not block.  Returns a null pointer on error. */

/* If _play_prefork is called then its return value is used
 * as the =data= argument to the following functions.  Otherwise
 * the value of =data= argument is indeterminate and must not
 * be used. */

void disorder_play_track(const char *const *parameters,
                         int nparameters,
                         const char *path,
                         const char *track,
                         void *data);
/* Called to play a track.  Should either =exec= or only return when the track
 * has finished.  Should not call =exit= (except after a succesful =exec=).
 * Allowed to call =_exit=. */

int disorder_play_pause(long *playedp, void *data);
/* Pauses the playing track.  If the track can be paused returns 0 and
 * stores the number of seconds so far played via PLAYEDP, or sets it to
 * -1 if this is not known.  If the track cannot be paused then returns -1.
 * Should not block.
 */

void disorder_play_resume(long played, void *data);
/* Restarts play after a pause.  PLAYED is the value returned from
 * the original pause operation.  Should not block. */

void disorder_play_cleanup(void *data);
/* called to clean up DATA.  Should not block. */
```

The playing process now looks like:

- call `_play_prefork` if required
- call `_play_track` inside a subprocess
- possibly call `_play_pause` and `_play_resume` some number of times
- possibly send a `SIGINT` (or as configured) to the player
- call `_play_cleanup` after the subprocess has terminated

For a standalone player the `_play_track` function should just play the track. Ideally if the audio device is in use it should retry it, but only for a few seconds.

A raw interface player should write the sample format and then raw samples to the FD specified by the environment variable `DISORDER_RAW_FD`. It need not support pausing as that is handled by the calling process.

The minimum old standalone players need to do is define `disorder_player_type` as `DISORDER_PLAYER_STANDALONE` and support but ignore the extra `data` argument to `_play_track`.

Pausible Player for Libao Players

Overview

This scheme consists of the following components:

- a new libao driver, which existing players such as ogg123 use
- a speaker process, which actually plays sounds
- a new player plugin

Libao Driver

This is very similar to the existing `raw` driver except that it will write the sample format and then the sample data down an inherited fd that is connected to the speaker process.

Speaker Process

This process is connected to (and started by) the main DisOrder server. There is a datagram UNIX domain socket between them, connnected to the standard input and standard output of the speaker process.

The following messages are sent to the speaker:

- `prepare(ID)` - get ready to play ID. Accompanied by a file descriptor from which to read sample data.
- `play(ID)` - play ID. If the track has not been prepared then it will be accompanied by a a file descriptor as above.
- `pause()` - pause the current track. This will produce either a `paused` or `pausefailed` response. Only allowed if there is a track playing.
- `resume()` - resume the current track. The speaker process keeps track of the number of seconds played so it need not be included in the command. Only allowed if there is a track playing and it is paused.
- `cancel(ID)` - cancel a track (possibly the current track). Only allowed for known ID values. It is allowed to cancel a paused track. The next track played does not become paused.
- `reload()` - re-read configuration file

The following messages are sent from the speaker:

- `paused(ID, PLAYED)` - pause succeeded, PLAYED seconds into ID
- `finished(ID)` - ID finished
- `failed(ID)` - failed to play or prepare ID

When the connection to the main server is closed the speaker process terminates any outstanding players and then exits.

The audio device shall only be held open when actually playing from a raw-interface player, i.e. not when paused, not when idle, and not when using a standalone player.

The speaker process is free to buffer as much as it likes but it may be desirable to limit the rate at which it feeds data to the hardware, if possible, to minimize the latency of cancels.

The byte format of the speaker commands is regarded as an implementation detail: they are private to DisOrder.

Player Plugin

The new player plugin, `execraw`, is identical to the `exec` plugin but declares support for the raw interface. (A

`shellraw` plugin could just as easily be made.)

Main Server

The main DisOrder? server starts the speaker process during its own startup and propogates reconfigure requests to it via the `reload` command.

The main server needs to gain the ability to pause tracks and to keep timing details sane while doing this. This has yet to be designed. Need to review existing code.

XXX

MCDP

Matthew's CD player would presumably issue the `CDROMPAUSE` and `CDROMRESUME` ioctls. It would have to declare `DISORDER_PLAYER_PAUSES` and might want to use the new `_prefork` mechanism.

Alternative Design

An alternative design would be to move most of the logic of the speaker into the main server, and write to the audio device inside a dedicated thread. The thread would receive open, close and play commands on a queue controlled by a mutex and a condvar, and would block all signals.

This eliminates most of the IPC but introduces a new dependency into DisOrder, i.e. thread support. I'm slightly loathe to do this. Though it is worth noting that the Mac OS X libao driver starts a thread anyway, so we cannot completely avoid it.

Threading and garbage collection are a poor mix, too: if the player thread never calls the allocator then the garbage collector cannnot run, if it does call it then it may be blocked for a long period in collection.

Even threading could be eliminated if writing audio samples could be guaranteed not to block significantly. This might be a platform-specific question however.

To Do

It would be convenient to allow multiple tracks to start decoding early. But the order in which things contact the server might not be the order in which they are to be played, and not all things might be played at all. The **cancel** operation obviously fits here but perhaps we want an additional operation to actually start playing a track which has already started decoding.

A thing worth thinking about is that it might be appropriate to start some kinds of player from the speaker process. This would eliminate the need for an id field in play commands, as the speaker would know which track it was from the FD.

If all players are started as a subprocess of the speaker then the logic looks like this:

- for a sample-generating player:
    - start the player at any time before the track is due to be played
    - don't read from the FD until it is needed
    - if the track is cancelled then just kill the process and dump the FD
    - when playing samples we must retry for up to a few seconds if the audio device is busy
- for a standalone player:
    - close the audio device before starting it
    - only start the player when it is actually needed

    - the player must retry for up to a few seconds if the audio device is still busy

This implies a protocol between the main disorder server and the speaker. The operations would be:

- prepare to play some track
- start playing a track (prepared or not)
- pause
- resume
- cancel a track (possibly the currently playing one)

Future Developments

Gap Elimination

The lower bound on the inter-track gap will already be pretty short with this logic.

With the above in place we can completely eliminate it however by starting a new track shortly before the old one ends.

Network Broadcast

This necessarily requires that all players send digital audio to a single place which can then redistribute it over the network. The speaker process is in exactly the right place to do this.

Network Protocol

The network protocol needs to be stable as not all endpoints may be upgraded simultaneously - they may be in different administrative domains and may be widely separated.

Multicast should work. This implies a connectionless-capable protocol. This would be convenient in any case.

It would be preferrable to have a single sample format rather than requiring endpoints to know how to cope with multiple sample formats. I note in passing that the Mac OS X libao driver only groks 44.1KHz.

Resampling in the speaker process does not sound much like fun, though on modern hardware it should be quite possible to do it faster than real time.

Timing

It may be that multiple endpoints are located very near one another. In that case it is desirable that they are as closely synchronized as possible. This means both getting them into step, which might reasonably require manual attention, and keeping them in step, which ought not to.

Suppose each frame contains a timestamp calculated by the network version of the speaker process. This would be calculated from the start of the track, or the point at which it was unpaused, plus the number of samples multiplied by the sample rate. This gives a target time to play that frame. The speaker should avoid sending out a frame much before its target time.

Each endpoint could then have a delta to add to this time, and play the frame as close as possible to the result. This delta effectively encompasses both the clock inaccuracy of the host and the latency of its sound device.

Adding a new endpoint should be a matter merely of tuning its own delta, though it might sometimes involve upping that of all endpoints a bit.

Another approach would be to keep the amount of data in flight to an absolute minimum and just have a delay in each endpoint. The tuning step would be much the same. This is simpler and would be less vulnerable to clock

jitter, but would be more vulnerable to variable network conditions.

It might be best to support both approaches and allow the operators to choose between them.

-- RichardKettlewell - 30 May 2005