A Musician Friendly Multilayered Music Recording System

Ole Martin Bjørndalen
Department of Computer Science
University of Tromsø
Advisor: Otto J. Anshus

December 9, 2002

Abstract

This report presents a simple sketchpad for layered recording of music. Musicians often need to record or try out ideas, but existing systems can be too complex. The system was implemented at user level in C and Python and tested on Linux and Windows XP. The recording is kept in RAM during a session. Audio I/O latencies below 6ms were achieved by applying paches to the Linux kernel and by compensating for expected latency. A simple user interface has been implemented to match the functionality of the system.

Contents

1 Introduction

1.1 Motivation

Musicians often need to record musical ideas, try out an arrangement or just spend half an hour or an afternoon recording some tunes.

For these simple tasks, existing recording systems can be too complex, requiring the user to think about the tool rather than the music. The result is that recording is not as fun as it should be, and the musician doesn't do it as often as he would like to.

What's needed is a simple sketch pad for music. This report presents one design for such a sketchpad, and shows how it was implemented.

1.2 Background

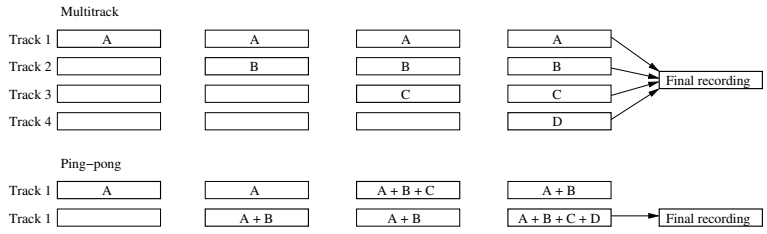


Figure 1: A song with four parts A-D (for example drums, bass, guitar, vocals) recorded on a four track system, and on a two track system using ping-pong.

Sound on sound recording is usually done on a multitrack system. Each instrument or part is recorded on to a separate track. Later, all or some of the tracks are mixed down to form a final recording.

The multitrack model is flexible. It is possible to re-record a track or to record several versions of a track and select the best take. One can adjust the level, pan, EQ and other parameters of each track individually.

In the past, track limitations forced many musicians to work in a different way. Systems with many tracks were expensive, and it is easy to run out of tracks when you have only a few of them. The solution is a technique known as ping-pong. You record the first part to track 1, then you record the track 1 and the next part to track 2, then the track 2 and the next part to track 1 and so on. By alternating between the two tracks in this way, you can record any number of parts. (On analog tape systems, some audio fidelity will be lost, but that is no longer a problem on digital system.)

1.3 The multilayer model

If you look beyond the actual physical tracks, what's going on here is that the music is built up layer by layer. You can always undo the last layer by switching back to the previous track, but you can't undo any of the layers below. You're stuck with them.

The layer model is in many ways less flexible than the track model, but it has some advantages. Once you have recorded a layer, and you're happy with

it, you never have to worry about it again. Sure, you can't adjust its level or add reverb to it, but on the other that's one less thing to worry about while recording. You can easily spend hours going back and forth between "Maybe that track needs some more reverb after all" and "No, it sounds like it was recorded at the bottom of a well inside a cathedral built in a cave". If you can't change it, you don't have to.

The limited functionality also means that the user interface can be simpler. All you need is four buttons: record, play, stop and undo, and perhaps some way to listen to the recording from the third verse to see if you got the riff right this time. There's no need for such things as track selection, track naming and input selection.

There's one other benefit: with less functionality, the system is easier to implement, and easier to implement correctly. That means fewer bugs, and less chance of it crashing or behaving strangely.

1.4 Functionality

The minimum functionality should be:

- build up a song by recording one layer of audio on top of another
- delete the layer last recorded (one level of undo)
- play back the recording
- store the recording to disk

It is tempting to add features, and some features were added to the layer model, and removed again when they were found to complicate the user interface unnecessarily. A description of these features and their implications can be found in Appendix A.

1.5 Requirements

simple It should have only the basic functionality outlined above, an maybe a little more. The urge to add neat features should be resisted. It should be possible to learn to use the program quickly.

less thinking The user's mind should be on the music, not on the tool.

responsive It should appear to act immediately to commands, rather than being sluggish.

reliable It must never crash or lose data, and always do what you expect.

no arbitrary limitations There should be no arbitrary limits on the number of layers or recording time, apart from the limits set by available RAM or disk space.

no mouse It should not require the use of a mouse during recording, since it is cumbersome to fumble with a mouse when both hands are busy playing an instrument.

no interruptions It should never pop up message boxes or questions during a recording session, or ask the user to "Please wait while processing data".

1.6 Hardware and software platform

The system will run as a normal application on a laptop computer running Linux.

1.7 Outline of the rest of the report

The report will first describe the user interface of the system with no clue as to how the functionality is implemented. Then the implementation is discussed, starting with the architecture and design of the system as a whole, and then talking about each aspect of the implementation in detail.

2 User interface

This section describes the program from the user's point of view.

Usability studies have not been performed. The user interface has only been tested on the author and his advisor.

2.1 Design criteria

The user interface should follow user interface conventions on the target platform where possible, to make it easier to learn. The interface should give the user a clear indication of the state of the program. It should be kept simple and uncluttered.

2.2 Overview

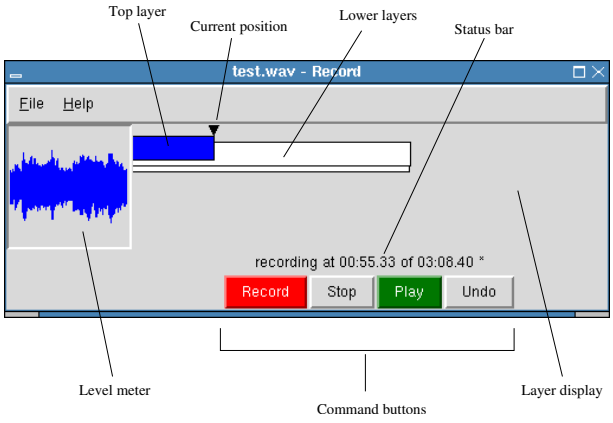


Figure 2: The desktop application while recording the third layer.

The figure shows the user interface of the desktop application. The program would look different on a palm top computer or embedded system, but it would have the same basic components.

At the top is the menu bar. The file menu contains the usual options to create a new recording, to load and save recording and to exit the program.

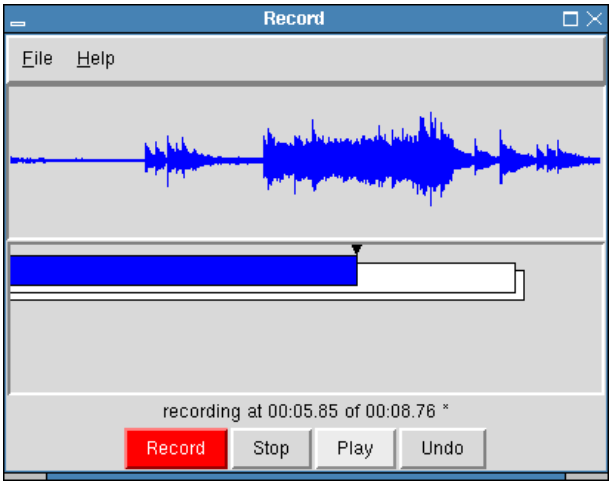


Figure 3: The same GUI, but with a larger level meter. The layout is cleaner and more balanced, but uses more screen real estate.

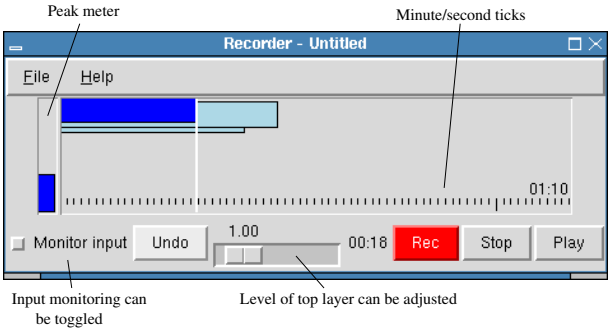


Figure 4: An earlier version of the GUI. Time is indicated with tick marks below the layer display. The tick marks were found to be more distractive than helpful, and were removed in later versions. The level of the top layer can be adjusted, as described in Appendix A.

At the center of the window is the layer display. Layers are stacked on top of each other as they are recorded. If the top layer is blue, pressing undo will remove it. The current position is indicated with an arrow running along the top of the display.

Below the layer display is the status bar which shows the mode ("stopped", "playing" or "recording"), the position and the length of the recording.

At the bottom is a row of command buttons. Each command can also be invoked with a button on the keyboard, so the on-screen buttons are rarely used.

On the left is the level indicator. It shows the combined level of the input from the sound card and the recorded sound over the last 4 seconds. The level indicator is helpful in adjusting the recording level.

2.3 The layer display

The purpose of the layer display is to give the user feedback on how many layers have been recorded, where each layer starts and ends, how the recording of the current layer is proceeding, and what the current position is.

Layers are shows stacked on top of each other in the order they were recorded. The top layer, the one last recorded, is colored blue to indicate that it can still be removed. White layers are frozen, and can no longer be removed.

The first layer is placed at the top of the screen. As new layers are recorded, existing layers will be pushed down. If the top layer is deleted, the lower layers will pop back up. An alternative way is to place the first layer at the bottom, and stack the layers up towards the top. The advantage of pushing down rather than building up is that the top layer will always be at the same place on the screen.

As the recording grows, the layer display will rescale to accommodate it. Similarity, when there are too many layers to fit, the display will rescale to make more room for twice as many.

The full recording is always visible. There is no way too zoom in for a closer look, but for the typical use there is not enough detail in the recording for zooming to be useful.

The layer display has proved to be intuitive and work well in practical use.

2.4 The level meter

When recording audio it is important to set the correct input level. If the level is too high, the samples are clipped, and the sound will be distorted. If the level is too low, background noise from the sound card, cables and other sources will come through.

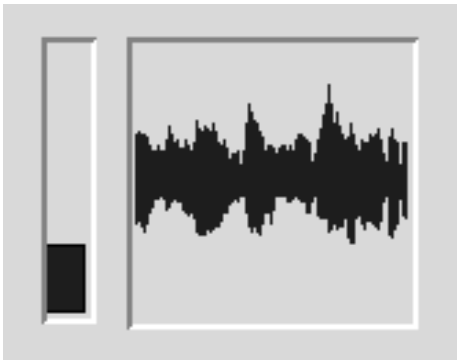


Figure 5: The peak meter at the left and the peak graph at the right.

Two meters were tried: A peak meter and a peak graph.

The peak meter is a bar indicating the peak value of the signal. If the signal is louder than the current value of the peak meter, the bar rises to the value of the signal. It falls back towards zero at a steady rate.

The peak graph displays a recent history of peak values. It draws a vertical line for every N samples. The height of the line is proportional to the maximum value of the N samples.

The peak meter didn't work very well at first. It was hard to get an idea of the recording level. The recording were often louder than expected. It turned out that the fall rate was too high, so when there was a peak in the signal, the meter dropped down so fast that you didn't see the peak. A fall rate of one second from maximum value to minimum value seems to work well.

The peak graph gives a good indication not only of the current input level. You can see at a glance how the signal has been for the last few seconds, whereas the peak meter shows only the current value.

2.5 Modes and commands

One early user interface had two toggle switches `running` and `recording` which controlled the state of the program. Four states were possible:

running	recording	State of the program
off	off	stopped
off	on	stopped
on	off	playing back the recording
on	on	recording a new layer

An additional button rewound to the start of the recording buffer. As a result, to record a new layer, the user would have to press one key and toggle none, one or two switches depending on the state of the program. This turned out to be very confusing in practice.

The solution was to organize the user interface around the user's actions. The most common actions are assumed to be:

- record a new layer starting at the beginning
- play back the recording from the beginning
- stop playback or recording
- remove the layer last recorded

One button is assigned to each action. Each button has one, and only one, well defined behavior, and puts the program in a known state. As an example, pressing the record button has the same effect (record a new layer from position 0) whether it is pressed while recording or while playing back.

The following table shows the new button scheme:

User action	Button	Mode	Start position
Record a new layer	Record	recording	0
Play back the recording	Play	playing	0
Stop	stop	stopped	0
Undo the last layer	Undo	stopped	0

2.6 A one-line command line user interface

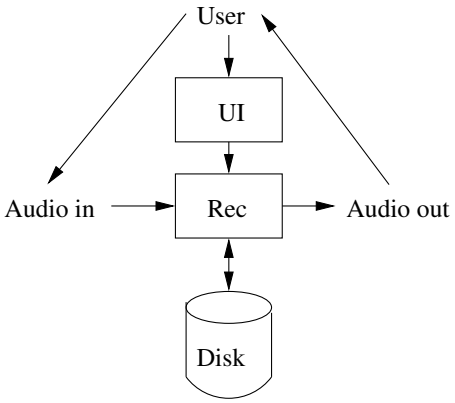
A simple command line user interface was developed for situations where a full GUI can not be run or is not desired, such as when running in single user mode under Linux. It currently looks like this:

```
recording   at 00:03.34 of 02:24.45 (undo)  [||||||||||||                    ]
```

The status line is the same same as in the desktop application. The layer display is replaced with a single word, "(undo)", which appears if undo is possible. At the far right is the level meter. The keyboard is used to give commands.

The program takes one command line argument, a file to open. Saving a recording is more tricky. One solution is to ask the user for a file name. Another solution is to make up a file name based on the system time, such as 20021202-144006.wav. That saves the user from coming up with a unique file name. It's a nice way to quickly jot down some ideas without thinking about naming files, but some way to annotate or browse existing recordings is needed, since the file names make little sense on their own.

3 Implementing the layer model



The figure shows the architecture of the system. The main components of the systems are the GUI and the recording engine.

3.1 Language

Two full implementations were made, one in C with a Python GUI one in pure Python.

The C implementation is written in run either with the Python GUI, or stand alone with a minimal one-line console UI. It runs on Linux and uses the OSS API for audio I/O. It is not ported to Windows.

The Python portion does the GUI. The C portion of the program does all the recording and file I/O, and is wrapped up in a Python extension module and imported into the GUI.

The Python implementation is written in Python 2.2.1. It uses Tkinter for (the Python wrapper for Tcl/Tk) for the GUI and PortAudio (http://www.portaudio.com/) for audio I/O. It runs without change on any platform to which Python, Tkinter and PortAudio are ported. It has been tested on Linux 2.4 and Windows XP.

At the time of writing the application consists of about 700 lines of code, most of which implements the user interface. In addition, a Python wrapper for the PortAudio library was written (about 350 lines of C code).

3.2 User interface

As mentioned above, the GUI was written in Tkinter. The layer display and level meter were both implemented as subclasses of the Canvas class.

The one-line command line interface is implemented as follows. The output is printed to the standard output on one line starting with a CR character. The standard output then flushed. The standard input is set to raw mode using `tcsetattr()` so each key press will return a character immediately.

3.3 Threads

The program divides naturally into two tasks:

GUI handle user input and redraw the user interface

audio process audio

There are at least three ways to divide the tasks into threads:

two kernel threads The audio processing can be done in a separate kernel thread. The audio thread blocks on audio I/O. The thread can run all the time, or it can be started and stopped as needed.

If it runs all the time, the GUI needs some way to tell it to switch to another mode (from 'recording' to 'playing' for instance). A message queue is used for this purpose. For each block it processes, the audio thread polls the message queue. When it gets a 'quit' message, it exits.

A tidier solution, which doesn't require message passing, is to spin up an audio thread only when it is needed. For instance, when the user presses 'record', the GUI spins up an audio thread which runs a function which record a new layer. When the user presses 'play', the GUI stops the thread, and starts a new thread which runs a function which plays back the recording. The threads are responsible for rewinding the position and opening and closing the audio device.

There is no need to protect shared data. The only data the user interface is interested in is the current position, the length of the recording and the value of the peak meter. These values are all integers; reading or writing each of them is an atomic operation, and can be done safely without protection. For operations which read or modify other data, such as clearing or saving the recording buffer, the program should be put in the 'stopped' mode first.

do both in the same thread Both tasks can be done in the same thread. The main loop would look something like this:

- check GUI event queue, and process any pending events
- read one block of audio
- process the audio block
- write one block of audio

do audio processing in a callback Most modern audio APIs allow audio processing to be done in callback function. The program opens the audio device and passes it a callback function to be called when there is data available. The callback function is passed a block of audio and should return another block of audio.

There are two arguments for placing the audio processing in a separate kernel thread. First, it saves the thread from blocking for user input. Second, it saves the audio thread from doing possibly lengthy operations like redrawing the display.

Doing everything in one thread will generally work well if the user input is something as simple as reading key presses from a file descriptor, but with a full GUI, processing the event queue could take some time of one of the events is a full redraw of the display.

One note on Python and threads: Python threads are not fully independent of each other. They share a global lock which they must acquire in order to execute Python code. For this reason, it is currently impossible to write code in Python where one thread is guaranteed never to block. Thus is may be just as well to do everything in one thread.

The third option was not tested.

3.4 Audio I/O

Audio I/O should as simple as:

```
open the audio device
in a loop:
  read a block
  process the block
  write a block
```

In reality, it is not so easy. Two constraints make it harder:

- there should be no perceptible audio latency
- there should be no dropouts (missing pieces of audio)

3.4.1 Audio latency

Here, audio latency is defined as the time it takes for a signal to travel from the input of the sound card, through the program, to the output of the sound card.

If you're monitoring through the program, high latency causes what you hear to lag noticeably behind what you're playing. Each new layer you record will also lag behind on playback. The result is musical timing problems and frustration. Latencies below 10ms are typically acceptable even for professional use.

High latencies are mainly due to buffering in the OS and application. It is possible to reduce latency by reducing the number and size of buffers, but doing so makes the process doing audio I/O more vulnerable to other events on the system. The process has to be scheduled in every time a buffer arrives from the sound device. If there are other processes using the CPU, the process may not

be scheduled in when it needs to, the queue will fill up, and buffers will have to be dropped.

Commonly suggested solutions include:

1. apply low latency patches to the Linux kernel
2. switch the process to FIFO scheduling using the `sched_setscheduler()` system call

In [MacMillan], MacMillan et. al. measure the latencies of desktop operating systems. They find that in Linux, stable latencies in the 3-4ms range can be achieved by applying Ingo Molnar's low-latency patch and Andre Morton's preemption patch[Molnar], and switching to FIFO scheduling.

The problem with `setscheduler()` is that the process must run with root privileges. It is unlikely that most people will want to run the program as root. The kernel patches present a similar problem, but there is some hope that they will make their way into the standard kernel. The preemption patch has already been accepted into the 2.5 branch of the kernel.

The problems of latency are by no means unique to this application, and many people are trying to solve it. One approach is a DeMuDi[DeMuDi], a Linux distribution for multimedia work, where the kernel is already patched and tuned for low latency performance.

3.4.2 Compensating for latency

If the latency can not be lowered to a tolerable level, all hope is not lost. If the latency is known, one can compensate for it by starting playback a little before one starts recording. The distance between playback and recording should equal to the latency.

Since the latency is mainly due to buffering, it is possible to get a rough estimate of the latency from the number and size of buffers used. If two buffers are used in each direction, the latency will be easily determined and stable. However, if more than two buffers are used in each direction, the total latency will vary depending on the load of the system. Under heavy load, more buffers will be in use, resulting in higher latencies.

3.5 Experiences with latency

Con Kolivas has made a Linux performance patch[Kolivas] which combines the low latency patch and the preemption patch with other patches which improve system performance and responsiveness. Several versions of the patch are available. I applied the lowest latency branch (llck5 2.4.19) on a stock 2.4.19 kernel. The test machine was a Thinkpad T22 with a CrystalClear SoundFusion sound card.

I wrote a small test program which echoed the input to the output so I could listen for dropouts. The dropouts could be clearly heard as clicks. The program was run for about 2 minutes under while various CPU and I/O intensive programs were run in the background.

On a standard system starting Netscape or running 'yes' in an xterm was enough to cause massive dropouts. Running as root and turning on realtime

scheduling helped a bit, but there were still frequent dropouts. With the kernel patches, I was not able to provoke any dropouts, even under heavy load.

The dropouts were gone, but the latency was still high. Due to limitations in the sound card or driver, the minimum buffer size was 2k, which amounts to 512 frames or about 12ms per block at 44.1Hz. With the minimum possible setting of two blocks of input and two blocks of output (double buffering), the total latency was still over 20ms, well above the tolerance of 10ms.

While the latency was high, it was at least predictable. Since there were only two buffers in either direction, the latency couldn't grow beyond the minimum. I now used the same buffer settings in the recording application. By calculating and then compensating for the latency, I was able to bring the perceived latency below 6ms, or half a buffer.

The remaining latency is due to the way the recordings are stored in memory. As described in 3.7.4, buffers are read from the audio device and inserted into a list. During playback, they are read one by one from the list and written to the audio device. The accuracy of latency compensation is therefore determined by the size of the buffers. With 12ms buffers (2k), the worst case is 6ms (half a buffer) off optimal compensation. A theoretical best case is 0ms perceived latency.

It is possible to get around this limitation by writing some code to go between the audio I/O and buffering, but this has not been attempted.

3.6 Memory vs. disk

Keeping the whole recording in memory rather than on disk has a few advantages:

- easier to implement
- shorter and more predictable seek times than disk
- disk latency doesn't have to be taken into account
- no temporary files (for undo or ping-pong)

On the other hand, a recording takes up a great deal of space. If the program is used on a computer with little RAM, such as an old laptop, the recording should be streamed directly to disk.

In a traditional multitrack system, keeping the recording in RAM would put a limit on the number of tracks that could be recorded. The layer model with only one level of undo requires only two (possibly stereo) tracks no matter how many layers are recorded.

The memory usage per second is roughly:

```
num_bytes = num_channels * sample_size * frame_rate
```

With CD resolution, the memory usage is:

```
2 bytes * 2 channels * 44100 Hz = 176400 bytes per second
```

or about 10 MB per minute. With undo, that's 20 MB per minute. A typical recording is expected to be around 3 minutes, and would take up about 60 MB of RAM.

If memory is tight, another solution is to lower the resolution. Lowering the sample rate to 22050 and cutting one channel cuts memory usage to one fourth, or about 15 MB for the three minute song, with sound quality which is still acceptable for many uses.

Another possibility is to encode the sound with a lossy encoding like mp3 or ogg vorbis. The tradeoff is higher CPU usage and more complex code. Also, for each layer added, some fidelity will be lost in the lower layers, as they are decoded and re-encoded. In typical use, there should not be enough layer for it to be noticeable, though. And again, the sound quality will still be acceptable for many uses.

3.7 Buffers and undo

The layer model could be implemented like a multitrack system. Each layer would be recorded to a separate file or memory buffer, and all layers would be mixed on playback. However, this takes up a great deal of storage. Available storage and CPU time (for mixing) will also limit how many layers can be recorded.

If the only operations the user can do is to add (record) or remove the top layer, it is possible to reduce the storage requirements to 2 times the size of the recording. This sections shows a few ways to implement this.

Some of the buffer designs below were designed for a more feature-rich version of the layer model, where layers could be recorded starting anywhere in the buffer. The complexity of the solutions should be seen in light of this.

3.7.1 Ping-Pong

One can implement a tape-like ping-pong with two memory buffers. Each buffer is initially empty (all-zero). Recording are done by alternating between the two buffers. The input is first written to buffer A, writing to the current buffer the sum of the other buffer and the input. One level undo/redo is possible by switching the two buffers.

The buffers should be large enough to accommodate a typical recording, or they should grow along with the recording.

The model has the same limitations as tape-based ping-pong:

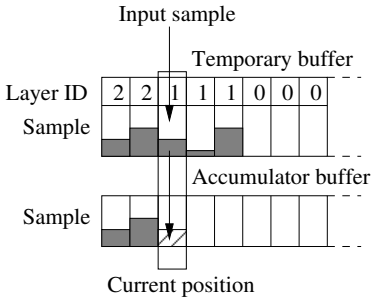
- if the new layer is shorter than the previous layers, the end of the of the previous layers will be lost, since it is not copied across
- for the same reason, all recordings must start at the beginning of the buffer

The user will have to take this into account while recording.

The limitation can be removed by copying the remaining samples across. However, it is likely that the copy operation would take noticeable time. At best this would make the program appear sluggish. At worst, it would force the user to wait.

Layer table

ID	Count	State
0	buffersize − 5	DISCARD
1	3	KEEP
2	2	CURRENT
3	0	
N	0	



3.7.2 Tagged samples

Tagged samples were introduced as way to get around the limitations of the ping-pong model.

Two memory buffers are used: a temporary buffer and an accumulator buffer. New layers are recorded to the temporary buffer. If the user decides to keep the layer, its samples are added to the accumulator, which contains the total recording.

To hide the fact that the add operation takes time, the samples are not added all at once. Instead, each sample is added the next time a record or playback operation passes over it.

The temporary buffer may now contain samples belonging to many different layers, some of which should be kept and some of which should be discarded. To keep the samples apart, each layer is given a unique ID and each sample is tagged with the ID of the layer it belongs to. A layer table records the number of unprocessed samples in each layer, as well as a flag telling whether the layer should be kept or discarded.

During playback or recording, the layer ID of each sample is checked. If layer is discarded, the sample is simply overwritten. If layer to be kept, the sample is added to the accumulator layer before being overwritten. In either case, the sample count of the layer is decreased by one.

When the sample count of a layer reaches 0, the layer ID can be reused. Initially, all samples belong to to layer 0, a special layer which is always discarded, and has a sample count so high that it will never reach 0. When samples are stored or discarded during playback, they revert to layer 0.

When the recording is saved, all samples are checked, and the ones that should be kept are added to the accumulator buffer. The accumulator buffer is then written to disk.

3.7.3 Segmented buffers

The same two buffers are used as in the tagged sample approach. Instead of tagging the samples, a list of segments is used to keep the samples apart. Each segment has a start position and an end position. When a new layer is recorded, a new segment is created. As samples are written to the temporary buffer, the end position of the segment is increased.

When the segment is committed, it is inserted into a list of dirty segments. While recording the next layer, the dirty segment list is checked. If a sample is within one of the dirty segments, the sample is added to the accumulator buffer

and the extent of the segment is adjusted.

If a recording starts somewhere in the middle of a segment, the segment must be split into two new segments.

3.7.4 Lists of blocks

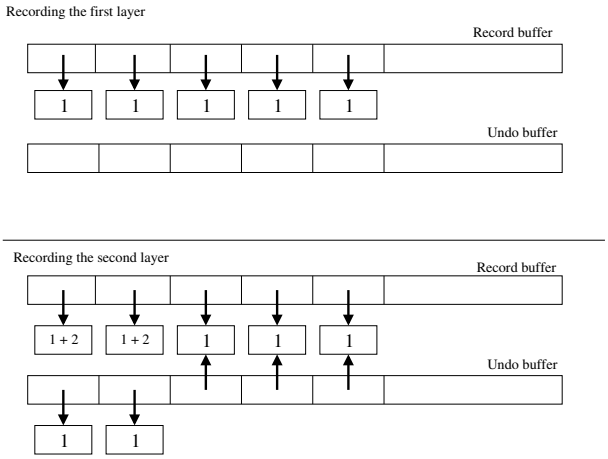


Figure 6: Top: Recording the first layer. Bottom: Recording the seconds layer. The undo buffer contains all the blocks from layer 1.

The recording buffers are split up into blocks of equal size. Each buffer is a list (or an extensible array) of references to blocks. Two buffers are used: a recording buffer and an undo buffer. When a new recording is made, the

Since a block can be a member of more than one list, there must be some way to keep track of how many times each block is referenced. In languages like Python and Java, reference counting or garbage collection comes for free. In C, a simple reference counting scheme should be implemented.

Recording is done as follows. As blocks are read from the audio device, they are inserted into the list at the current position. If there is already a block at that position in the list, it is replaced with a new block which contains the sum of the old block and the input block.

3.8 Implementing buffers

Ping-pong was not implemented, because it was assumed not to work well. It should probably have been implemented to provide a baseline for comparison.

Tagged samples and segmented buffers were implemented in C. Both turned out to be complicated to implement. The main loop of the recording thread was full of tests and special cases, and there were frequent bugs.

The lists of blocks were implemented in Python as lists of strings. Python strings are immutable and can contain binary data. Python lists are extensible arrays of references to objects of any type. Because built-in data structures could be used, the implementation was very simple. Undo took only three lines of code:

```
if undo_buffer:
```

```
record_buffer = undo_buffer
undo_buffer = []
```

At first, the program barely ran on a 900MHz Pentium III. It used about 90% CPU. The bottleneck was assumed to be the function which sums the samples of two blocks of audio. When it was removed with a dummy function, the CPU usage dropped to around 1-2%. The function was replaced with one written in C, (`add()` from the `audioop` module which is included with Python). The result was a 100 times speedup of the add function, and a CPU usage of 1-2%.

There is some memory overhead for the lists and object headers. Python strings have a 20 byte header followed by the content of the string. There is also a 32-bit pointer to each block in the list. That gives a 24 byte overhead per block. With 2k blocks, the memory overhead is 24/2048, or about 1.2 per cent. The real overhead is likely a little higher due to details in the memory allocation and alignment.

The actual overhead was measured by loading a big file into memory as one big string, or as a list of 2k strings, and comparing the memory usage of the two cases. A series of tests were run, and the memory usage was gauged with the 'top' program. The memory overhead was stable at 2.7 per cent.

3.9 Evaluation

The tagged sample model used a lot of memory. A separate array of 16-bit ints were used for the tags. There was one tag for every pair of 16-bit samples (stereo), and two buffers were used, so the memory overhead of the tags is 25%. This obvious flaw led to the tagged sample model, which is functionally equivalent and should be used instead.

The segmented buffers have the advantage that they can be used when the recording directly to disk. This is not true for the list of blocks, since it relies on pointers, but it would be possible to use some ideas from the list of blocks data structure to create a block based buffer scheme for use on disk.

The lists of blocks turned out to be the most flexible of the three models. It could easily be adapted to implement unlimited undo (keep a stack of old versions of the list of blocks) or multitrack recording (an array of lists of blocks).

One surprise was how well the Python program performed, but when in retrospect, it should not have been as surprising. Python is very bad at CPU intensive tasks, but pretty good at I/O intensive tasks. The way the program is written, Python is used as a control language, routing audio I/O and user input, and the CPU intensive parts are done in C.

3.10 Disk IO

There are at least two reasons to record directly to disk. First, it allows the system to be run on a computer with little RAM, such as an old laptop. Second, saving to disk takes some time. If the data is already on disk, the user doesn't have to wait for it to be saved.

Four implementations were attempted: direct read and write, prefetch threads (reader and writer), ring buffers and memory mapped files.

Each implementation reads frames from a file, adds to them the input from the audio device, and writes them back to the same file.

Undo was not implemented.

3.10.1 Direct read and write

Data frames are read from the files as they were needed, and are written back to the same file. There is no buffering, except what the file system provides.

3.10.2 Reader and writer threads

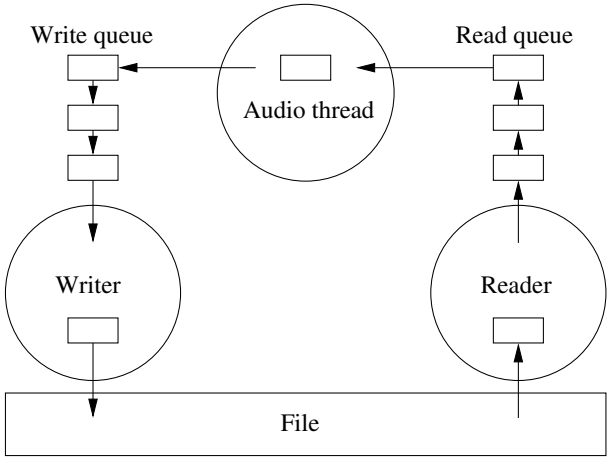


Figure 7: The reader and writer threads.

The goal of the reader and writer implementation is to hide disk latency from the audio thread. The audio thread should never have to wait for disk reads and writes to complete.

There are two threads: a reader and a writer thread. Each thread has a queue of buffers, initially empty. The job of the reader thread is to prefetch data from the file. It reads blocks from the file and puts them into the read queue. It blocks when the queue is full. As long as the reader thread manages to fill the queue, the audio thread can fetch data without blocking.

When the audio thread wants to write a block to the disk, it puts the block into the write queue. The writer thread fetches each block from the write queue and writes it to disk. It blocks when the queue is empty.

When recording is over, or a new layer is to be added, all blocks in the writer queue must be written to disk first. If they are not, the end of the current layer will be lost. One solution is to send the writer thread an empty block to signal the end of the stream, and then wait for it to exit. The reader thread can be killed with no ill effects. Both threads can be restarted when the next layer is recorded.

3.10.3 Ring buffer

The ring buffer is a simplification of the reader and writer model, using one prefetch thread and a circular array of audio frames (sample pairs).

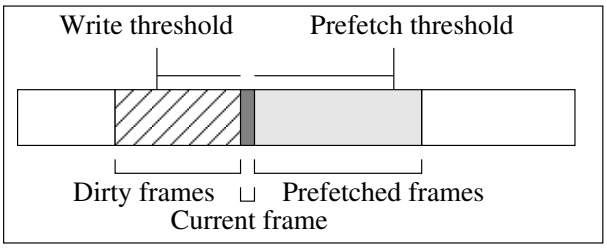


Figure 8: The ring buffer

The audio thread walks through the buffer with an index. For each frame, it adds the input from the sound card. The result is written back to the same position in the buffer. The frame is now dirty, and must be written back to disk.

The disk thread wakes every 50ms and checks the following conditions:

- If the number of dirty frames is above a threshold, it writes them all to the file.
- If the number of prefetched frames is below threshold, it fill the buffer with frames from the file.

Dirty frames are given priority, because they must be flushed to make room for new frames from the file.

No synchronization is used between the threads. The disk thread must always stay ahead of the audio thread. If they cross, the audio thread will get the wrong data.

The file is opened with the `O_SYNC`, to ensure that the data is written to file immediately.

3.10.4 Memory mapped file

The file is mapped into memory using the `mmap()` function call. A stepper thread is used to hide latency from the audio thread. It wakes up a few times per second and reads one byte from every memory page ahead of the audio thread in order to force a page fault. This will spare the audio thread from waiting for the page fault to complete.

There was one problem: Linux `mmap()` does not support the `MAP_AUTOGROW` flag, so the file will not grow as memory pages are touched. However, there is a way to implement the missing functionality. In Linux, it is possible to `truncate()` a file to a size larger than the original size, like 2G, and truncating it back to the actual size when recording is over. Disk blocks will not be allocated until data is actually written to the file, so this has the exact same effect as `MAP_AUTOGROW`.

3.10.5 Evaluation

The Linux audio driver reports recording overruns and playback underruns to the syslog, and also prints them to the console. If the program is run from the console, the error messages can be seen interleaved with the program's

output. The implementations were tested by looking for these error messages and listening for clicks and pops.

To force worst case performance, each test was run on a file which was too large to fit in physical memory. The file was created by copying the required number of bytes from `/dev/zero`. At the end of the copying, the start of the file would be swapped out. The file was recreated at the start of each run.

It was expected that the direct read and write would perform poorly, but the others would work better, due to prefetching.

In the first round of tests, all implementations performed exactly the same: poorly. They ran perfectly for many seconds, but then there were bursts of dropouts every second or so. The fact that four different implementations behaved the same, pointed to an outside factor.

The dropouts seemed to coincide with disk activity. A test program was written where the disk thread wrote a byte to the file every second. The file was opened with the `O_SYNC` flag, so write() would not return until the data had been written physically to disk. When the program was run, the dropouts coincided perfectly with the disk writes.

It turned out that the test computer didn't have DMA on the hard disk. The tests were rerun on another computer with DMA, and ran practically without dropouts. To test if the lack of DMA was causing the problem, DMA was turned on and off with `hdparm` program. Every time DMA was turned off, the dropouts started, and when it was turned back on, they went away.

The implementations were then tested on the computer with DMA. The direct read and write still performed poorly. There were frequent dropouts. There are two possible causes. First, only one frame, or 4 bytes, was read and written at a time. Second, only one file pointer was used, and it had to be rewound before each write operation. No further investigation was made.

The reader and writer threads had only occasional dropouts, but when the file was played back, parts of the recording was missing. There is a bug in the implementation.

The ring buffer had fewest dropouts of all implementations. It was tested with different buffer sizes. A prefetch threshold of 5 seconds and a buffer size of 10 seconds seemed to work well.

The memory mapped file worked about as well as the reader and writer threads, but also managed to record the audio correctly.

One final observation: If the recording is too large to fit in physical memory, caching will not work. Audio recording is linear: it starts at the beginning of the file and ends at the end. When the end of the recording is reached, the start of the recording will be swapped out. When recording the next layer, every page read from the disk will force out another page ahead of it. The cached frames will never be touched. Caching will only work if the entire recording can be kept in memory.

4 Evaluation

The program works well for its intended purpose. It is quick and easy to record a tune, and both users (the author and his advisor) have found themselves recording more often and enjoying it more, which was the goal. Several other musicians have expressed an interest in trying the program.

The audio I/O latency are not completely resolved. It is possible to get low latencies with Linux, but until the Linux kernel can deliver low latency audio out of the box, the user will have to patch and recompile the kernel.

The program works well if you have enough RAM, which both current users have.

One surprise was how well Python performed. Python was expected to:

be too slow because it is interpreted

use too much memory due to the overhead of the object system

have bad I/O response due to slow context switching

Instead, it was found to perform about the same as C, as long as one key function (the one that adds two audio blocks) was moved to C.

5 Related work

Alternative solutions can be divided roughly into two categories: multitrack systems such as hard disk recorders and portastudios, and single track systems such as MiniDisc recorders and simple sound recording programs.

The multitrack systems come with a whole range of functions that are not needed for simple tasks. The typical goal is to give the user a complete recording studio. As a result, they are expensive and take time to learn and master. Simpler systems are often easier to use, but lack features such as overdubbing.

MacMillan et. al. have measured the audio latencies of several desktop operating systems[MacMillan].

Martin Walker has written article[Walker] in Sound on Sound where he gives a lay man's explanation of audio latency, and offers some solutions to musicians struggling with it.

6 Conclusion

The layer model can be implemented effectively on a laptop computer running Linux if the recording is done in RAM. It is still unclear whether direct to disk recording can be done effectively without recording each layer to a separate file.

Creating something simple is not always easy. The layer model is easy to explain and understand, but implementing it turned out to be more complicated than expected. The temptation to add features added to the complications.

7 Future work

The prototypes will be developed further and released as free software for everyone to enjoy and build upon.

One possible research direction is to implement the layer model on a small portable device like a palm top computer or a custom-built embedded device.

More work is needed on the disk I/O, especially figuring out a way to implement undo.

Usability testing should be performed. At least the program should be tested by a few musicians who have not been involved in the project.

8 Acknowledgements

I would like to thank the following people for interesting discussions, help and support: John Markus Bjørndalen, May-Lis Bjørndalen, Kjell Dokka, Ron Røstad, Karen Bjørndalen, Ken Arne Jensen and my advisor advisor Otto J. Anshus.

9 Appendix A: Additional features

When you're developing a program, it's tempting to add features. The layer model as described in the introduction of this report has very little functionality:

- record a new layer
- play back the recording
- undo the last layer

Features that crept into the program include:

record anywhere In the original model, it is only possible to record from the start of the recording. Sometimes this is impractical, such as when you want to add a guitar solo after the second chorus.

This means there has to be a way to select a point where the recording should starts, which in turn means that there must be some way to determine where the second chorus actually ends, so you can go there and start recording. A number of approaches were implemented and tested:

- Let the user forward and rewind or scrub through the recording to find the spot, and then press record.
- Punch in recording. The user listens to the recording, and presses a button when he wants to start recording the solo.
- Fly back. The user places one or more markers in the recording and presses a key to fly back to a marker and record or play back from that position.
- Let the user click with the mouse in the layer display. Clicking the left button plays back the recording from the mouse position. Clicking the right mouse button records a new layer starting from the mouse position.
- Combinations of the above.

A detailed account of all these implementations and their implications would take many pages, so it will be left out here.

reverb for when you're recording in a very dry room. It was implemented using Freeverb, a reverb model developed by recording engineer Jeremy "Jezar" Wakefield. The reverb required at least two knobs to be added to the user interface, to control the room size and wetness of the reverb.

reverse recording/playback It's fun to record something, and then play it back in reverse to listen for secret messages, and it may even be used for artistic purposes (which was the idea). This required a forward/reverse toggle or a couple of new command keys "reverse record" and "reverse play". It also complicated the implementation.

half speed/double speed Record a guitar solo and play it back at double speed, or record a flute and play it back at half speed. Kids had a lot of fun playing with this feature, making smurf and troll noises.

The problem with all these features is that they:

- were not essential, but just "kind of neat sometimes"
- made the user interface more complex
- make the implementation complex
- distracted me from the real research

Some of the features might still be useful, such as the reverb. The fun features like half speed and reverse recording, could be used in a toy program.

References

[MacMillan] Audio Latency Measurements of Desktop Operating Systems
Karl MacMillan, Michael Droettboom, Ichiro Fujinaga
Peabody Institute of the Johns Hopkins University

[Molnar] Ingo Molnar's low latency patch
http://www.zip.com.au/ akpm/linux/schedlat.html
http://people.redhat.com/mingo/lowlatency-patches/

[Kolivas] Con Kolivas' performance patch
http://members.optusnet.com.au/ckolivas/kernel/

[Love] The Linux kernel preemption project
http://kpreempt.sourceforge.net/

[DeMuDi] DeMuDi - Debian Multimedia Distribution
http://www.demudi.org/

[Walker] Mind the Gap - dealing with computer audio latency
Martin Walker
Sound on Sound, April 1999
http://www.sospubs.co.uk/sos/apr99/articles/letency.htm