平成13年度学士論文

タイムスタンプ付ストリームI/Oによる音の実時間処理

東京工業大学 理学部 情報科学科
学籍番号 98-0997-3
栗田 亮

指導教官
千葉 滋 講師

平成14年2月7日

概要

現在、マルチメディアアプリケーションへの要求が高まっているが、従来の Linux などの汎用 OS ではメディアデータを正確に処理できない場合がある。このような問題は、汎用 OS がリアルタイム処理機能を提供できていないことに原因がある。リアルタイム処理とはデバイスからの入力信号やプログラムからの要求に対して一定時間内にこれを処理することである。現在リアルタイム処理を実現するためにリアルタイムタスクを用いるリアルタイム OS が開発されているが、リアルタイム処理の実現は難しく、その構造は複雑で理解するのは困難なものである。

本研究ではストリーム I/O である音の録音に関して、タスクレベルではなく I/O レベルでの実時間処理を行うシステム TS-I/O を提案する。ストリームの処理においては I/O 管理や複数のストリーム間の同期が重要視されることなので、I/O のみのリアルタイム化のほうがシンプルであり、それで最低限のデータ品質が保証できる。TS-I/O は音飛びが生じた場合、その時間をアプリケーション側に知らせることができる。また、この時間をもとに音飛びした部分に無音データを挿入することにより、音源データと録音したデータを同じタイミングで再生することができる。現在、無音データの挿入処理は TS-I/O の機能としてカーネルが行なっているが、アプリケーションがこの処理を行なうようにすることもできる。

謝辞

本研究を進めるにあたり、研究の方針や論文の構成や書き方を助言して頂いた指導教官の千葉先生に感謝致します。
　また、研究の方針や実装の方針について助言していただいた東京大学の光来健一氏、筑波大学の立堀道昭氏、筑波大学の横田大輔氏に感謝致します。
　そして同研究室で苦楽を共にした学部生のみなさまに感謝致します。

目 次

図 目 次

表 目 次

第1章　はじめに

現在、ビデオ会議やビデオオンデマンドなど、動画や音声などのメディアストリームを自在に扱うアプリケーションへの要求が高まってきている。メディアストリームの中で代表となる音は、作曲や演奏、音声音響研究などさまざまな分野でコンピュータにより扱われている。

音のコンピュータによる処理の中で基本となるのがサンプリングである。サンプリングとは音をデジタル変換し、コンピュータに記録する技術のことである。サンプリング処理について Linux カーネルにおける音の録音処理の流れを簡単に説明すると、まずハードウェアによってデジタルに変換されたサウンドデータが割り込みによりカーネル内のバッファに記録される。次に、カーネルバッファ内のサウンドデータはアプリケーションによってアプリケーション内のバッファに転送される。このようにして録音が行なわれる。

音のサンプリングの際に問題となるのが音飛びという現象である。音飛びが生じると一部のデータが抜け落ち、抜け落ちた分余計に記録してしまい、本来記録すべきデータとは違うデータが記録されてしまう。音飛びの原因は、割り込み処理の遅れとアプリケーションの遅れの 2 つが考えられる。いずれもサンプリングと同時に複数のアプリケーションを実行させていると発生する。

このような音のサンプリングなどのメディア処理での問題は、Linux などの汎用 OS がリアルタイム処理機能を提供できていないことに原因がある。リアルタイム処理とは、デバイスからの入力信号やプログラムからの要求に対して与えられた時間、デッドライン内にこれを処理することである。

現在ではリアルタイム処理を行うことを主目的としたリアルタイム OS が開発されている。リアルタイム OS はプリエンプティブなマルチタスク OS で、リアルタイムタスクの処理をデッドライン内に確実に終わらせることができる。Linux ではこれまでいくつかリアルタイム処理機能を提供するリアルタイム OS が開発されている。

しかし、タスクレベルでのリアルタイム化は難しく、完全なリアルタイムシステムには至っていないものが多い。また、その構造の複雑さからプログラミングが困難になる、安定性が十分でなくなるなどの問題がある。

そこで我々は Linux において、タスクレベルでなく I/O レベルでストリーム I/O のリアルタイム処理を行うシステム TS-I/O (TimeStamp I/O) を提案する。なぜ I/O レベルでのリアルタイム処理に注目したかというと、ストリーム I/O の処理においてはシステムの応答性より、時間に基づく I/O 管理がおこなえるかどうかが重要である。ゆえに I/O のみのリアルタイム化でも、最低限のデータ品質が保証できる。本システムは音の録音に関してのリアルタイム処理を行う。

TS-I/O は Linux の機能にはない音飛びの検知機能と音飛びの通知機能を、追加したシステムコール `read_gettbuf()` を呼ぶことで提供する。これらの機能を提供するために、ストリーム I/O にタイムスタンプをつける手法をおこなった。このタイムスタンプはストリーム I/O、ここではサウンドデータがカーネルバッファに記録される時間を示す。各割り込みで処理されたサウンドデータのタイムスタンプを見比べて音飛びが生じたかどうか調べ、音飛びが発見された場合アプリケーションバッファにそのタイムスタンプを渡す。このタイムスタンプをみることによって、アプリケーションは音飛びが生じたかどうか知ることができる。

また、音飛びのタイムスタンプを調べることによって、音飛びに対するリカバリー処理を行なうことができる。音飛びにより取得できなかったサウンドデータのサイズ分の無音データをその場所に挿入する。これにより原音と同じ実時間にそった再生が可能である。

現在、この処理は TS-I/O の機能としてカーネルが行なっているが、音飛びのタイムスタンプは取得できるのでアプリケーションでもこの処理は可能である。それゆえ、アプリケーションが音飛びに対するリカバリー処理を行なうライブラリを提供するようにすることもできる。

本稿では、音のサンプリング処理とその問題点について、リアルタイム処理とリアルタイム OS について、そして提案するシステム TS-I/O について説明する。2 章では、そもそも音を処理するとはどういうことか、その処理ではどのような問題が生じるのかを説明し、現在の汎用 OS ではなぜそれが解決できないかその原因を述べる。3 章では、音の処理を正確に行うことができるリアルタイムシステムについて説明する。また、現在開発されているリアルタイム OS をいくつか紹介し、その問題点を述べる。4 章では、TS-I/O について、その特徴と利点、具体的な機能を説明する。5 章では、TS-I/O の実行例を述べる。6 章では、本稿で述べたことのまとめと今後の課題について述べる。

第2章　音の処理とその問題点

本研究ではストリーム I/O の中から音に注目した。音はメディアの中で最も古くから研究されて、現在ではさまざまな分野で扱われている。このセクションでは、音の処理ではどのような問題が生じるのか、また、一般的な OS ではなぜその問題を解決できないのかを説明する。

2.1　音とコンピューター

音は、自動作曲システムがコンピュータの発明とほぼ同時期に作られるなど、計算機技術の応用分野として古くから存在している。現在ではパソコンの普及やマルチメディアの振興により、コンピュータは音楽の様々な方面で利用されている。音楽の作曲や演奏、電子楽器などから、音声音響分析や音楽認知など、身近なところから専門分野まで、様々な方面で音はコンピュータにより扱われている。

音のコンピュータによる処理の中で一番基本となるのが、サンプリングである。サンプリングとは、アナログ的なものである音を不連続な値、デジタルなものに変換し、RAM やハードディスクといったデジタル保管媒体に記録する技術である。これにより、コンピュータは気圧の波である音をデータとして扱うことができるのである。

しかし、サンプリングと同時に他の複数のアプリケーションを同時に実行させていると、マシンに負荷がかかり正常に処理できない場合がある。サンプリングの中でも音の録音についてでは、この現象は一般に音飛びと呼ばれる。音飛びは、負荷のかかったマシンがサウンドデータの処理を終えることができず、その処理が終わるまで後からくるデータを処理できずに破棄してしまう現象である。この現象が生じたサンプルデータを再生してみると、音飛びが生じた時間のデータが抜け落ちて聞こえる。また、抜け落ちたデータ分余計にサンプリングをおこなっていしまう。このようにして音源とは違ったものになってしまうのである。

わかりやすように、図を使って説明しよう。図 2.1 は録音時の音飛びの様子である。音のデータが 1 秒ごとにマシンに転送されるとしよう。負荷のかかったマシンは、音のデータをすぐには処理できず、音のデータはマシンに設けられた記憶領域に一時的に置かれる。例えばこの記憶領域のサ

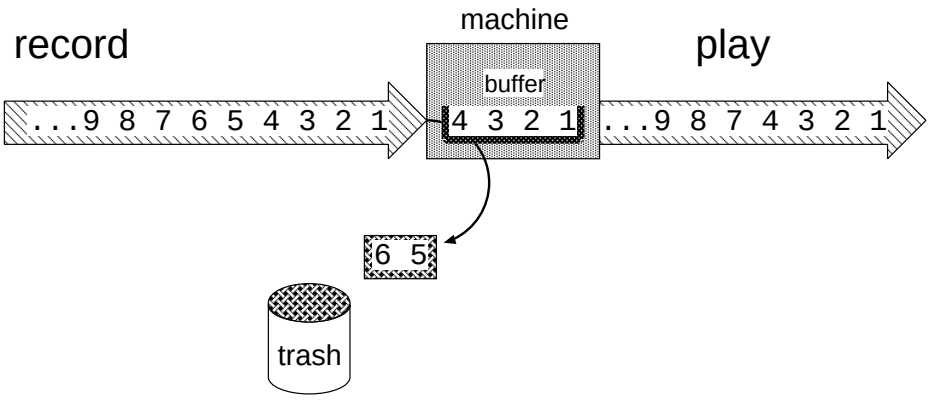

図 2.1: 音飛びの発生

イズが 4 秒分のデータまでしか入らないとする。そして、マシンがデータを処理するのが 6 秒かかるとすると、5 秒目と 6 秒目のデータは破棄されてしまう。その後マシンが 4 秒分のデータを処理し終わると、7 秒目、8 秒目 … とデータがたまって処理される。処理し終わった一連のデータを再生してみると、5 秒目と 6 秒目のデータが抜けて、4 秒目のデータが再生された後、7 秒目のデータが再生されるのである。これが音飛びである。

このような音飛び現象を起こさないようにするには、マシンを高性能なものにすれば解決できそうである。しかし近年、マルチメディアアプリケーションへの要求は高まり、１つのマシンで複数のマルチメディア処理を行うようになっている。このような要求にマシンの性能が追いついていないのが現状である。

2.2　サンプリングとその仕組み

サンプリング

まずは、サンプリング [10] とはどのような処理かを詳しく説明する。

音は気圧の変化で生じるもので、アナログ的なものである。音を時間の関数としてプロットすると、その音は連続なもので、声でも音楽でも両者の複雑な混合物でも、各時点で一意に表現できる。

コンピュータは不連続な値しか記憶できないので、音を処理、保存できるような形に変換するには、アナログ・デジタル変換（AD 変換）という処理が必要になる。同様に、コンピュータのメモリ中のファイルやデータを音で出力するには、デジタル・アナログ変換（DA 変換）という処理が必要となる。この機能をコントロールするデバイスは、D/A 変換器とか

DAC、A/D 変換器あるいは ADC と呼ばれる。

アナログ・デジタル変換のプロセスは、サンプリングとも言う。デジタルサンプルの質、つまりそれがどれだけオリジナルに近いかは、おおむねサンプル・レートとサンプルサイズの 2 つの要素で決まる。

サンプリングレートとは、アナログ波形の値が計測され、離散量に変換される一定の間隔のことを言う。通常は 1 秒あたりのサンプル数か、あるいは一般にヘルツで表現される。サンプリングレートが高ければ高いほど、それだけ元のアナログ波形を忠実に再現できる。逆にサンプリングレートが低すぎると、原音に関する情報が失われてしまう。また、サンプリングレートが上がると、それに比例して保存される情報量も増加する。

もうひとつ重要なのが、サンプルサイズである。アナログからデジタルに変換されたサンプルのそれぞれのとりうる値のことを、サンプルサイズという。一般的には 8 ビットで、この場合では各サンプルは 2^8、つまり 256 種類の値のどれかとなる。16 ビットの A/D 変換を使えば、2^{16}、つまり 65536 の値が使えるので、より原音に近い音となる。しかし、サンプルサイズを大きくするとそれだけ記憶容量も増大する。

それでは CD を例として説明しよう。音楽 CD は、44,100 サンプル/秒のサンプリングレートと、16 ビットのサンプルサイズを使っている。また、ステレオなので、そのデータを 2 チャンネル分持っている。すると普通の 60 分 CD に含まれるデータは、

```
60 分× 60 秒/分× 44,100 サンプル/秒× 2 バイト/サンプル× 2 チャンネル
= 635MB
```

となる。これは CD-ROM の一般的な容量（約 640MB）に近い数字となる。

サウンドデバイス

次に、アプリケーションはどのようにしてサウンドデバイスドライバにアクセスするかを説明する。

Linux に限らず、UNIX 互換システムすべてに言えることだが、ユーザープログラムが直接ハードにアクセスすることはほとんどない。ハードウェアデバイスのドライバは、すべてカーネルが持っている。ユーザーがこれらのデバイスをコントロールするには、システムコールと呼ばれる一部の関数呼び出しを行う。システムコールと呼ばれるのは、それが OS のカーネルに実装されているからである。

open システムコールは、デバイスへのアクセスを確立し、ほかのシステムコールで操作ができるようにする。read システムコールと write シス

テムコールはそれぞれデバイスからデータを返し、書き込みを行う。ioctl システムコールは、その他全てを担当するシステムコールである。ファイルやデバイスに対して read や write コールに納まらないさまざまな処理をするために使用する。最後に close システムコールを使い、OS にデバイスをもう使っていないことを告げる。ファイルやターミナルへのアクセスは、printf や scanf などを使用するほうが効率がよい。なぜなら読み書き操作はメモリにバッファされ、後で大きなブロックとしてまとめて処理されるからである。しかしマルチメディアデバイスへのアクセスの場合、データがすぐに必要であり、バッファのサイズについて自分でコントロールしたい場合が多いので、read や write を使用するのが適切である。

それでは実際にサウンドドライバ用デバイスファイルへのアクセスのしくみを、デバイスファイル/dev/dsp を例に説明しよう。図 2.2 は/dev/dsp へのアクセスの仕組みである。

DSP デバイスから読み出すと、デバイスは A/D コンバータから得たデジタルサウンドサンプルを返す。まず、アナログ波形は、カーネルサウンドドライバの制御のもとで、A/D コンバータによってデジタルサンプルに変換される。この変換されたデジタルデータは、A/D コンバータのハードウェア割り込みによってカーネルのバッファに格納される。アプリケーションが read システムコールをすると、カーネルバッファのデータは読んだアプリケーションのバッファに転送される。

このとき、読み出しが速すぎると、カーネルサウンドドライバは必要量のデータが揃うまでプロセスをブロックする。データの読み出しがサンプリングレートより遅ければカーネル バッファがデータで一杯になり、余ったデータは捨てられ、デジタル化された音にギャップができてしまう。

DSP デバイスにデジタルサンプル値を続けて書き込むと、サウンド出力が得られる。データの書き込みが遅すぎるとカーネルバッファにデータが書き込まれるまで時間が空いてしまい、サウンド出力に間が生じる。サンプリングレートより速くデータを書くと、カーネルサウンドドライバは単純にプロセスをブロックして、サウンドカードのハードが新しいデータを処理する準備ができるまで待つ。

このようにしてアプリケーションは音のサンプリングを行なうことができる。

2.3　音飛びの 2 つの原因

音飛びとは、録音時にマシンに負荷がかかっている場合サウンドデータの処理が遅れ、後からくるデータを処理できずに破棄してしまう現象であると述べた。これについて、サウンドデータの処理の遅れとは何なのか、

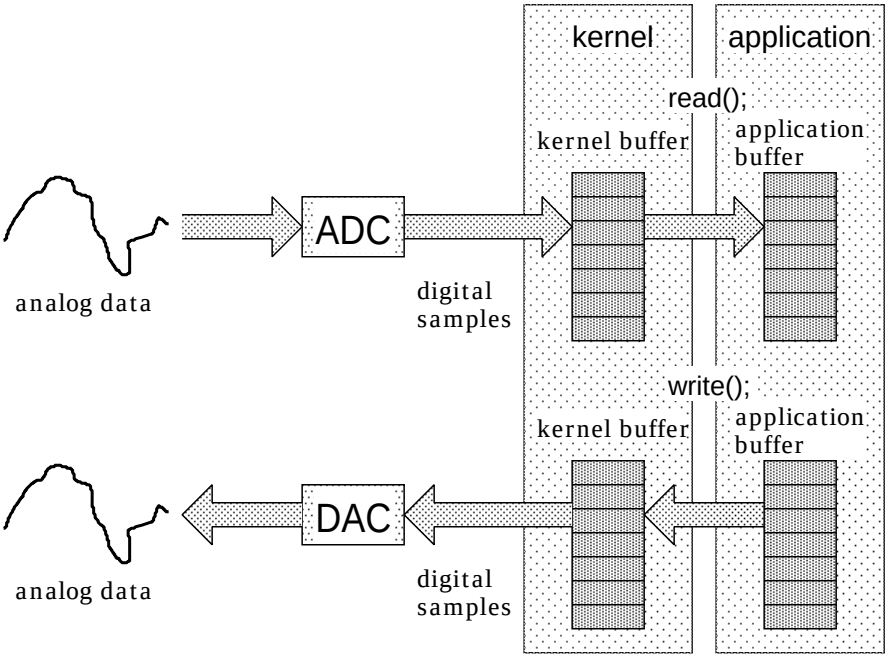

図 2.2: /dev/dsp へのアクセス

前に述べたカーネルでのサンプリング処理の流れをふまえて説明する。

サウンドデータは A/D コンバータ、カーネルバッファ、アプリケーションバッファと受け渡される。マシンの負荷により受け渡しの処理が遅れてしまうのは、A/D コンバータからカーネルバッファへの受け渡しの際と、カーネルバッファからアプリケーションバッファへの受け渡しの際である。

まず、A/D コンバータからカーネルバッファへの受け渡しの遅れを説明する。これは A/D コンバータの割り込み信号に対する、カーネルの割り込み処理の遅れのことである。これは他のハードウェアからの割り込みが頻繁に起ると生じる。カーネルがいつまでも割り込み処理を行なわないと A/D コンバータのバッファが溢れ、あとから来るサウンドデータは破棄されてしまう。これによって音飛びが生じてしまう。

そしてもう 1 つ、カーネルバッファからアプリケーションバッファへの受け渡しの遅れがある。これはサウンドデバイスの話で説明したが、サンプリングレートよりデータの受け渡しの方が遅くなり生じる。カーネルバッファがサウンドデータで一杯になり、カーネルバッファに格納されるはずのデータは破棄されてしまう。これにより音飛びが生じる。

このように音飛びには 2 つの原因がある。

2.4　Linux カーネルの問題点

それでは現在の汎用 OS ではなぜ音飛びを制御できないのか。汎用 OS のなかから Linux をピックアップして調べた。

Linux のカーネルは UNIX システムのインターフェイスである POSIX 1003.4 標準に従う、システムコール sched_setscheduler を使うことにより設定した処理を他の処理より優先して実行することが可能である [5]。このシステムコールはスケジューリングクラスを選択する。

スケジューリングクラス SCHED_FIFO と SCHED_RR にはリアルタイムのプロセスが入っている。 SCHED_FIFO クラスのプロセスは、制御を放棄するかまたはリアルタイム優先順位の高いプロセスが始まらない限り、実行を継続できる。SCHED_RR クラスのプロセスは、同じリアルタイム優先順位のプロセスが始まった場合にも中断される。それに対し、通常のプロセスはスケジューリングクラス SCHED_OTHER にあてはまる。リアルタイムプロセスはすべて SCHED_OTHER クラスのどのプロセスよりも高い優先順位になる。

```
/* sched_setscheduler(); */
int sched_setscheduler(pid_t pid, int policy,
                       struct sched_param *param);
```

sched_setscheduler の第 2 引数 policy に SCHED_FIFO または SCHED_RR を指定することによって、通常のプロセスよりも優先して実行することが可能である。

しかし、この機能だけでは、サンプリング時の音飛びを完全に制御することはできない。

なぜなら、Linux が管理できる時間は 10ms なので、それより短い時間管理を行うことができないからである。実際、サンプリング周期は数 ms、数百 μ s で行われる。このような時間単位は一般的には問題視されない短いものである。しかしサンプリングの研究ではこのような短い時間の精度が必要とされる。

そしてもう一つ、サンプリングにおける大きな問題がある。それは、Linux カーネルは音飛びが生じたかどうか判断することができないということだ。1 秒や 0.5 秒のずれなら人間の耳で判断できるかもしれないが、0.01 秒やそれより短い時間のずれを聞いて判断するのは不可能である。よってマシンが判断しなければならないが、現在の Linux にはそのような判断機能はない。また、いつ音飛びが発生したか、その時間をアプリケーションに知らせることもできないので、音飛びが発生したことがわかっても、それに対する処理をすることができない。

これらの理由から、Linux では正確なサンプリングを行うことはできない。

Linux などの汎用 OS ではサンプリングなどのマルチメディア処理を正確に行うことができない。その原因は、Linux は数 ms や数百 μ s のような細かい時間の管理を行うようには実装されていないことである。もう一つの原因は、Linux はストリーム I/O の処理が正常にできたかどうか、例えば音飛びが生じたかどうか判断できないということである。

第3章　リアルタイムシステム

マルチメディアの処理で問題が生じてしまうのは、OS がリアルタイム処理を提供できていないことが原因である。リアルタイム処理とは、デバイスからの入力信号やプログラムからの要求に対して与えられた時間、デッドラインまでにこれを処理することをいう。現在はリアルタイム処理を提供できるリアルタイム OS が開発されているが、既存のリアルタイム OS にも問題がある。ここでは、リアルタイムシステムに必要な機能、既存のリアルタイム OS とその問題について説明する。

デッドラインミスとは一般的に、割り込み処理のデッドラインが守れない、割り込み処理の遅れのことを指すが、本稿ではアプリケーションの遅れのこともデッドラインミスと呼ぶことにする。音の録音に関して言えば、カーネルバッファの溢れによる音飛びもデッドラインミスと呼ぶことにする。

3.1　リアルタイム処理

「リアルタイム」という言葉はいろいろな意味で使われる。まずはここで使う「リアルタイム」の正確な定義を説明する。

リアルタイム性というと、応答速度が速いことと誤解されることが多いが、厳密にはある時刻、これをデッドラインというが、それまでにデータを処理できること、あるいは一歩譲って、デッドラインを超えてしまった場合があるとしてもシステムの時間的挙動が予測可能ということである。たとえば、平均的に見れば応答速度は十分に速いが、ときどき極端に遅くなることがある場合には、最大実行時間を予測できないという意味でリアルタイム性はない。また，周期的に起動するタスクの起動時刻のばらつきが大きいとか、一度起動が遅れるとその遅れがシステム全体に伝わったり、累積してどんどん遅れてしまう場合もリアルタイム性がないといえる。[8]

リアルタイム処理は、ハードリアルタイム処理とソフトリアルタイム処理に大きく分類される [8, 7]。ハードリアルタイムとは、プロセスに要求する実行終了時刻が厳密に決められているリアルタイムシステムである。実行開始から終了までの要求時間が非常に短く、高精度が要求される。ソ

フトリアルタイムとは、目標とする実行終了時刻がハード・リアルタイムほど厳密ではないシステムである。通常の場合、実行開始から終了までの要求時間は比較的長く、必ずしもワースト・ケースの処理完了時刻に精度を要求されない。 この分類によれば、汎用 OS(MVS、VMS、UNIX、MacOS、Windows、Linux など) もソフトリアルタイム処理を行っていることになる。一方、ハードリアルタイム処理は、いわゆる専用リアルタイム OS や組み込み OS で行われてきた。VxWorks、pSOS、VRTX、ITRON、Windows CE などがあげられる。ハードリアルタイムアプリケーションの例は、自動車のエンジン、ロケットなどの制御装置や、医療機器などがあげられる。ソフトリアルタイムアプリケーションの例は、携帯電話や電子レンジなどの身近な家電製品があげられる。一般にリアルタイムというと、ハードリアルタイムを指して使われることが多い。

ハードリアルタイム処理では、タスクがデッドラインまでにデータ処理を終えるにはどうすればいいか、がよく扱われる問題である。タスクが一つしかない場合は簡単で、デッドラインよりも計算に必要な時間分だけ前に計算を開始すれば十分である。では、タスクが複数あって、各々にデッドラインと必要な計算時間が与えられた場合は、どのタスクの計算をいつ開始すればよいのか、これを解決するのが、ハードリアルタイムスケジューリング理論である。主に 2 つの理論がある。一つは、最悪実行時間の短いタスクほど高い優先度を割り当てる、rate-monotonic(RM) である。もう一つは、デッドラインの早いタスクほど優先度を高く割り当てる、earliest-deadline-first(EDF) がある。また、この理論ではタスクがプリエンプティブであること、つまりタスクをいつでも中断し、他のタスクに切り替えることができることを前提とする。これにより、各タスクのデッドラインを保証することができる。

しかし、タスクが独立である場合でなく、タスク間通信を行う場合では問題が生じる。あるタスクが他のタスクの計算結果を利用するため、計算終了まで待たなければならないことがある。このように、タスク間で同期が必要な場合には、優先度逆転と呼ばれる現象が発生し、タスクのデッドラインが保証できなくなることがある。これは、高優先度のタスクが低優先度のタスクの処理を待っていた場合、中優先度のタスクが実行されてしまうというものである。そこでもっとも簡単な回避手法として、優先度継承がある。これは、高優先度のタスクをブロックしている低優先度のタスクに、一時的にブロックされているタスクの優先度を継承して、中優先度のタスクが実行されるのを抑制するというシンプルなものである。

また、リアルタイムアプリケーションでは数 ms 秒から数百 μ s という時間間隔でタスクの切り替えが必用とされることがあるので、十分に細かい時間管理処理性能が必要である。

ソフトリアルタイム処理で必用なことは、デッドラインを越えてしまった場合になんらかの対応処置をすることである。処理を停止するか、そのままつづけるかなど、その後の処理に悪影響を与えないように、各処理の時間的制約を守れるようにする処置をとる必用がある。

リアルタイム処理にはこれらの機能が必用とされている。これらをふまえて、改めて現在の Linux を見てみると、リアルタイム処理機能を提供できないことがわかる。

Linux はプリエンプティブなマルチタスク機能を採用し、固定優先度を割り当てることでスケジューリングを行っている。しかし、Linux が管理できる時間の最小値は 10msec なので、それより短い時間管理が必用なリアルタイムアプリケーションを実行することはできない。

次に問題なのは、Linux では周辺デバイスからの割り込みが入ると、リアルタイムタスクよりも優先して実行されてしまうことである。これにより実行が中断され終了時間が遅れ、デッドラインを守れない可能性がある。

最後に、優先度逆転の制御機能をもっていないことがあげられる [13]。Linux ではさまざまなプロセス間通信機構が提供され、送信プロセスと受信プロセスとの間で同期がとられる。リアルタイムタスク間で通信が行われ、同期がとられると、優先度逆転が発生する可能性がある。よって、リアルタイムタスクでプロセス間通信機能を用いることができない。

これらの問題があるので、Linux はハードリアルタイムスケジューリング理論で必用とされる機能を提供できない。デッドラインミスを発生させないようにすることはできない。また、前にも説明したが、Linux にはデッドラインミスを検知する機能がないのでソフトリアルタイムで必要とされているデッドラインミスに対する対処処置機能がない。よって Linux はリアルタイム OS とはいえない。

3.2　現存するリアルタイム OS

リアルタイム処理の中でも特にハードリアルタイム処理を行うために、多くのリアルタイム OS が開発されている。リアルタイム OS はこれまで長い間、組み込みシステムで使われてきた。組み込みシステムの定義は、あいまいではあるが「パソコン、ワークステーション、大型コンピュータ、およびこれらをベースにしたシステム以外で、高性能のマイクロプロセッサや小型のマイクロコントローラが介在したシステム全て」のことを指す [6]。つまり、コンピュータの外観をしていない、CPU を搭載している製品、装置、機器のことである。しかし最近では、組み込みシステムにあてはまらない大規模なシステムにまで使われるようになってきている。

ここでは、組み込み OS である VxWorks、ITRON、LynxOS と、パソコンやワークステーション向けの OS である RT-Linux、ART-Linux の５つを紹介する。

3.2.1　VxWorks

WindRiver 社 [4] で開発されている Vxworks は、もともとは Lab 系、制御系を中心にネットワークを前提とした UNIX ライクな開発環境が使えるリアルタイム OS であることを売りとしていた。現在では、さまざまなデジタル家電に採用される代表的な組み込み OS となっている。

VxWorks の特徴は、UNIX ワークステーションをホスト開発マシンとすることである。ターゲット・ボードと LAN 接続して、UNIX ライクに組み込みアプリケーションが開発でき、最終的に ROM 化も可能である。この他にも、組み込み OS として TCP/IP を標準的に取り入れたこと、API としてソケット・インターフェースを含めた UNIX 互換のランタイム・ライブらいを実装したことも特徴といえる。

VxWorks は、開発環境として Tornado を提供する。Tornado では、デバック情報を保持し、ダウンロード経路もネットワークにとらわれず、余計なデバッグ・リソースをターゲットに負担させないようにした。その背景には、家電製品などネットワーク機能や余分なターゲット・リソースをもてない製品が対象になってきたことがあげられる。

Vxworks のような組み込み OS はスレッド・モデルを採用している。これに対し Windows や Linux などはプロセス・モデルを採用している。スレッド・モデルの場合、リアルタイム OS の基本的な機能を提供するカーネル部分や、それ以外のリアルタイム OS メーカが提供するライブラリとユーザが作成するアプリケーション・コードはスタティックにリンクされ、コード空間やデータ空間を共有することになる。そのため、組み込みのように関連するタスクが密接してデータを共有している場合は、効率の良いマルチタスクの手法といえる。

しかし、この手法はシステム規模が小さい場合には大変有益であるが、システム規模が大きくなると問題が生じる。アプリケーションやデバイスドライバなどの 1 つのミスが、システム全体のクラッシュにつながる可能性を含んでいるのである。 1 つのスレッド中のバグがシステム全体にとって壊滅的な結果をもたらしかねない。 アプリケーションのサイズが比較的小さい場合、このような問題が発生しても対処は簡単に行える。しかし、システムが大きく複雑になるに従い、システム全体を把握するのが困難になり、バグの発見などにも時間がかかるようになってしまう。 Vxworks にはこのような問題がある。

3.2.2　ITRON

ITRON[2] は日本製で、組み込みシステムで最も多く利用されている OS である。ITRON は正式にはリアルタイム OS の製品名ではなく、標準仕様の名称である。今日では、それをもとに製品化された ITRON 仕様の OS が、半導体メーカーやソフトウェア・ハウスから数多く出荷されている。

ITRON は、リアルタイムカーネル仕様に重点を置いて仕様の策定、公開が行われた。その理由は、小規模な組み込みシステムでは、カーネル機能のみが利用されるケースが多いためである。ITRON1 仕様は現在の ITRON4 まで受け継がれている。これにより、実行効率と割り込み応答性を高めている。

ITRON 仕様の特徴は、8 ビットのマイコンから 32 ビットの RISC マイコンにまで使えることである。これにより ITRON 仕様は多くの種類のマイコン用に実装され、極めて多くの組み込みシステムに応用された。

最近では、組み込みシステムの大規模化･複雑化の進行に伴い、ソフトウェア部品インターフェースの標準化に取り組んでいる。例として、TCP/IP API として ITRON TCP/IP API 仕様、また、Java の技術を組み込みシステムに適用した JTRON2.0 仕様があげられる。

ITRON の問題となることは、Vxworks と同様にスレッド・モデルを採用しているので、スレッド中のバグによりシステム全体がクラッシュする可能性があることである。組み込みでは問題とならなかったが、大規模なシステムでは問題である。

このように、組み込みシステムでは有益だったスレッド・モデルは、パソコンやそれ以上の大規模で複雑なシステムには適応できない。従来の組み込み OS をそのままパソコンなどでは使用しづらい。

次に紹介する LynxOS も組み込み分野で使われるリアルタイム OS だが、この OS はプロセス・モデルを採用している。

3.2.3　LynxOS

LynxOS は LynuxWorks[3] が開発しているリアルタイム OS で、リアルタイム UNIX ともよばれる。この OS はプロセス・モデルであり、カーネルは特権レベルによってユーザータスクから保護されていてる。

LynxOS の特徴は、リアルタイム処理に対応したデバイスドライバに、カーネルスレッドと呼ばれる技術を使っていることである。カーネルスレッドはリアルタイムタスクで、あるデバイスへの入出力はすべてこの

カーネルスレッドの中で行う。割り込みハンドラとその他のリアルタイムタスクは、カーネルスレッドと一種のタスク間通信を行うことで、間接的にデバイスにアクセスする。

LynxOS の問題点は、他の UNIX からの移植性が低いことである。なぜなら、カーネルスレッドという特殊な機構を用いてデバイスドライバを記述するので、移植には根本的なソースコードの書き換えが必要だからである。たとえ書き換えたとしても、LynxOS がバージョンアップごとにまた書き換えなければならない。また、最新のデバイスを利用したいとき、独自にドライバを開発しなければならないという問題が生じる場合がある。なぜなら、LynxOS のユーザー数は汎用 OS と比べてはるかに少ないので、サポートされるまでどのくらい待つかわからないからである。

LynxOS の問題点からわかるように、大規模なリアルタイム OS には、既存のソフトウェアとハードウェアを利用できる汎用機能が必用である。現在、Windows などの汎用 OS にはソフトウェアやハードウェアが豊富に存在するので、これらが利用できればたいへん便利である。

そのような機能を備えるために、汎用 OS である Linux にリアルタイム処理機能を付加したリアルタイム OS が開発されている。汎用 OS の中から Linux が選ばれた理由は、動作が安定している、無償であるなどあるが、一番の理由はソースが公開されていることである。これにより、様々な機能拡張がその機能を必用とする人たちによって自由に行うことができる。以下にあげる 2 つのリアルタイム Linux はその成果として生まれたと言える。

3.2.4　RT-Linux

RT-Linux[1] は、汎用 OS である Linux にリアルタイム処理機能を付け加えた OS である。RT-Linux ではリアルタイムカーネルがすべての制御を握り、Linux カーネルはリアルタイムタスクより優先度の低いタスクとみなされる。

リアルタイムカーネルは、リアルタイムタスクおよび Linux カーネルを対象とするスケージューリングを行う。Linux カーネルは、もっとも優先度の低いタスクのように取り扱われ、リアルタイムタスクの実行を妨げないので、リアルタイムタスクは時間制約を守ることができる。

リアルタイムタスクと Linux のプロセスは、RT-FIFO と呼ばれる仮想デバイスを通して通信を行うことができるので、リアルタイムタスクが

Linux アプリケーションの処理結果を利用したり、周辺デバイスへのアクセスに Linux のデバイスドライバを利用することができる。

また、すべてのハードウェア割り込みはリアルタイムカーネルが受け、フラグが立っていなければすぐに Linux カーネルの割り込みハンドラの実行を指示し、フラグが立っていなければ Linux カーネルが割り込み許可マクロを呼び出すまでその指示を遅らせる。

RT-Linux には問題がある。リアルタイムタスクはカーネルにダイナミックリンクされて実行される点である。これは VxWorks と同じで、カーネルがアプリケーションから保護されていないので、アプリケーションのバグによってカーネルがクラッシュしてしまう可能性がある。そのようなバグが発生するとデバッグが大変困難になる。

次に、リアルタイムタスクと Linux プロセスが全く別の構造として記述されているために、その間の優先度逆転を制御することができないという問題がある。高優先度のリアルタイムタスクが Linux プロセスに、RT-FIFO を通してメッセージを送っているときでも、その Linux プロセスよりも優先して別のリアルタイムタスクが実行されることによって、RT-FIFO がオーバーフローしてしまう可能性がある。[14, 8]

また、ユーザー的な問題点はカーネル空間でのリアルタイム・プログラミングが必要ということである。これの何が問題かというと、ハードウェアごとにドライバを開発しなければならないということ、そしてシステムコールを利用できないということである。これはユーザーに大きな負担をかけてしまう。

RT-Linux にはこのような問題がある。RT-Linux は完全なリアルタイム処理を提供できているとはいえない。

3.2.5　ART-Linux

RT-Linux の問題を解決したリアルタイム Linux が、ART-Linux[12, 13, 11] である。ART-Linux は日本の、電子技術総合研究所で開発されている Linux のリアルタイム拡張 OS である。

ART-Linux では従来の Linux のアプリケーションやデバイスドライバがそのまま利用できるのが利点である。ユーザーは ART-Linux に用意された追加システムコールを使うことにより、リアルタイム処理が可能である。これにより、ユーザーはリアルタイム処理のプログラムを容易に作成することができる。

ART-Linux は周辺デバイスからの割り込みによるリアルタイム処理の妨げを防ぐために、割り込みハンドラを周期的に実行している。周辺デバイスからの割り込みは、いつ、どのような頻度で発生するかわからないた

め、割り込み処理にどれくらいの計算時間を使うのか予測できず、デッドラインを保証できなくなる可能性がある。これに対し ART-Linux では、割り込みの発生直後に割り込みハンドラを起動するのではなく、周期的に割り込み信号を監視して、割り込みハンドラの登録要求があると、周期的に再開されるリアルタイムタスクを生成し、タスク内で割り込みハンドラを実行する。リアルタイムタスク内で割り込みハンドラを実行することによって、リアルタイムタスクと割り込みハンドラ間の優先度継承を行うことができる。

次に、ART-Linux には優先度逆転を制御する優先度継承機能がある。この機能をもつリアルタイムロック機構により、プロセス間の同期を行っている。ART-Linux では、リアルタイムタスクと非リアルタイムプロセス間の通信が行われ、非リアルタイムプロセスがリアルタイムタスクをリアルタイムロックによってブロックする場合があるので、リアルタイムタスクから非リアルタイムプロセスへ優先度の継承ができなければならない。

また、ART-Linux ではメモリ保護機能が有効な、非特権レベルでリアルタイム処理を行うので、万が一プログラムに不具合があっても、それ以外の処理やシステム全体は安全である。

ART-Linux の問題は、安定性が不十分なところである。ART-Linux ではシステムコールの実行中でもプロセスの切り替え処理を行なう。従来の Linux カーネルは複数のプロセスが利用する共有データがあり、あるプロセスがシステムコールを呼び出すとその処理が終了するまで他のプロセスの切り替えを禁止するプリエンプト禁止期間がある。この禁止期間がリアルタイム処理を妨げる場合があるので、ART-Linux ではプリエンプト禁止中でもプリエンプトを行なう。プリエンプトを禁止しなければならない部分があるのは確かなので、カーネルのデバッグを行なわなければ安定性は期待できない。

ART-Linux はオープンソースなので、このような問題は時間をかければ解決できると思われる。しかし現在の実装状態ではまだまだ不十分と言える。現在の ART-Linux を利用するには、より安定性を上げるためにマシンの CPU パワーが必要である。

3.3　リアルタイム OS の問題

リアルタイム OS は多方面で利用されているので、その用途に応じた特徴があり、またその特徴から生じる問題があることがわかった。これまで紹介したリアルタイム OS とその特徴を表 3.1 にまとめた。

VxWorks と ITRON はスレッド・モデルを採用している組み込み OS である。スレッド・モデルは組み込み分野のように小規模のシステムならば

OS	システム規模	汎用性	カーネル保護	開発期間
VxWorks	小さい	×	×	長い
ITRON	小さい	×	×	長い
LynxOS	小さい	×	○	長い
RT-Linux	大きい	○	×	やや短い
ART-Linux	大きい	○	○	短い

表 3.1: さまざまなリアルタイム OS とその特徴

便利であるが、カーネルがアプリケーションから保護されていないので、パソコンなどの大規模なシステムには適応できない。

LynxOS は組み込み OS だが独自の特殊な技術でプロセス・モデルを採用している。しかしこのような特殊なシステムでは、既存のソフトウェアやハードウェアを使うのは非常に困難である。

RT-Linux は Linux にリアルタイム処理機能を付け加えた OS である。RT-Linux では Linux カーネルはリアルタイムタスクより優先度の低いタスクとみなされる。しかしカーネル保護機能がないので大規模なシステムにはむいていない。また、優先度逆転問題に対応する機能もないので、完全なリアルタイムスケジューリング処理はできない。

ART-Linux は RT-Linux にはないカーネル保護機能と優先度継承機能を備えたリアルタイム Linux である。安定性に問題があるので今後の開発に期待する。

では、既存のリアルタイム OS が音のサンプリング処理に適しているかどうか考えよう。既存の汎用リアルタイム OS の代表として、RT-Linux と ART-Linux をみていく。

RT-Linux や ART-Linux でそれぞれ説明したように、リアルタイム処理を提供するためにはさまざまな機能が必要であり、システムが複雑になってしまう。ハードリアルタイム処理の実現のためには、マルチタスク機能、スケジューリング機能、時間管理機能などさまざまな機能が必要となり、その実装の規模は大きく複雑なものとなる。この複雑なシステムを完全に把握するのは難しく、現在リアルタイム OS を使いこなせる技術者は不足しがちである [2]。リアルタイムアプリケーションの需要が高まってきている中で、この問題は大きいと思われる。

RT-Linux は割り込み処理の高速化を行なっている。これにより割り込み処理をリアルタイムに実行することができ、割り込み処理の遅延を制御す

ることができる。しかし問題はアプリケーションの遅延である。RT-Linux を扱うにはアプリケーション・プログラムをリアルタイム用に書き換えなければならない。しかし RT-Linux の場合これは、アプリケーション側がある程度カーネル・プログラミングを理解していなければならない。これはユーザーに負担をかけてしまう。

それに対し ART-Linux は、アプリケーション側はシステムコールを使うだけで容易にリアルタイム処理を行なうことができる。これによりユーザーはリアルタイム用にデバイスを用意する必要がなく、従来のプログラミングでリアルタイム処理を扱うことができる。しかし、複雑なシステムを OS に詰め込めば詰め込むほど、オーバーヘッドや安定性の問題があらわれてくる。リアルタイム処理ではなく通常の処理に支障がおきてしまうことが考えられる。

このような問題を解決するには、まず広く一般的に使われている汎用 OS を基にしたもので、その機能をできる限りそのまま使えるようにしなければならない。まったく新しい OS を開発するのはたいへんな時間と労力を必要とする。また、既存の普及しているソフトやハードが使えないのは不便である。

そして、リアルタイム処理機能をできるだけシンプルで簡単な方法で実現する必要がある。マシンに負担をかけず安心して使えるものにしなければならない。しかし、汎用マルチタスク OS にとってタスクの管理はシステム全体の管理であるので、リアルタイムタスクを用いることはシステム全体の複雑化につながってしまうのである。リアルタイムタスクを用いてデッドラインミスを防ぐには、システムの複雑化はさけられない。

第4章　TS-I/O

この章では我々が提案するシステム TS-I/O(TimeStamp I/O) について説明する。既存のリアルタイム OS はその複雑さが問題である。我々はストリーム I/O の処理においてタスクレベルでのリアルタイム化が絶対に必要なのか考え、本システムでは I/O レベルでのリアルタイム化に注目した。ここでは TS-I/O の特徴と利点を述べて、具体的な機能について説明する。

4.1　メディアストリームの処理で重要なこと

前節で紹介したような従来のリアルタイムシステムでは、外部からのイベントにいかに早く応答できるかといったシステムの応答性だけが注目されていた。それゆえ、これまでのリアルタイム OS では、デッドラインミスを発生させないようなハードリアルタイム処理が必要とされていた。

しかし、音声や動画のような連続したデータであるメディアストリームを処理する場合には、応答性よりも、各処理の時間的制約が満たされるかどうかが重要なことである。なぜそれが重要かというと、メディアストリームで求めらることは、そのデータが実時間に沿った連続なものかどうか、また、複数のメディアストリーム間で同期がとれるかどうか、ということだからである。これはつまり、データの処理をすばやく行なうことよりも各データの実時間に沿ったタイミングを重視するということである。例えば、衛星放送などで別々の回線から送られてくる画像と音声を合成する場合、同期をとることができないと唇の動きと聞こえてくる声がずれてしまう。

このような時間的制約のあるストリーム I/O の処理においては、デッドラインミスは起こるものと考え、ソフトリアルタイム処理を行うことで時間的制約を最低限守ることができる。ソフトリアルタイム処理で必要なことは、デッドラインミスの検知と、そのデッドラインミスの通知である。ストリーム I/O の処理中にデッドラインミスが生じた場合、それ以降のデータが実時間に沿うように、なんらかの処置をする必要がある。

4.2　TS-I/O の特徴

TS-I/O が支援すること

TS-I/O は Linux カーネルに対して、ストリーム I/O のソフトリアルタイム処理に必要な機能を提供する。ソフトリアルタイム処理では、デッドラインミスは起こるものと考えるので、そのデッドラインミスに対する処理機能が必要である。具体的には、**デッドラインミスの検知**、**デッドラインミスの通知**、それと**デッドラインミスに対するリカバリー処理**である。

TS-I/O ではこの 3 つの処理をストリーム I/O の特徴である連続性を活かして、ストリーム I/O に**タイムスタンプ**を付けることで行う。ストリーム I/O の割り込みは通常は周期的に行われる。これはサンプリングの解説のときに説明したように、アナログデータは一定間隔でサンプルされるからである。しかしマシンに過負荷がかかるとデッドラインミスが生じて、割り込み処理の遅延が生じることがある。この割り込み処理の遅延は、アプリケーションの遅延によっても生じる。このような割り込み周期の乱れを検知する機能は Linux にはない。

TS-I/O では、各 I/O 割り込みに対してタイムスタンプを付加し、タイムスタンプの時間間隔が割り込み周期間隔より大幅に越えている場所があった場合、デッドラインミスが生じたと判断する。デッドラインミスが生じた場合、カーネルはアプリケーションにデッドラインミスの時刻を示すタイムスタンプを転送する。アプリケーション側は、カーネルから取得したタイムスタンプをみることで、デッドラインミスが生じたことを知ることができる。また、このタイムスタンプをもとにデッドラインミスに対するリカバリー処理を行うことができる。例えばデッドラインミス以降のストリームデータがデッドラインミスによる遅延分、実時間とのずれが生じてしまわないように、デッドラインミスによりとりこぼしたデータサイズ分、空のデータを挿入することで、実時間に沿ったストリームデータを保つことができる。

TS-I/O の利点

I/O レベルでのリアルタイム化による簡易リアルタイム処理

従来のリアルタイム OS では、システム全体のリアルタイム化の実現を目的としていたが、現実にはその実現はかなり困難のものである。また、実現できても、その実装の構造は大きくて複雑なものとなり、技術者はその OS のシステムのしくみを理解するのに苦しむ。OS 全体のリアルタイム化は極めて難しいことといえる。

メディアストリームにおいては、デッドラインミスはシステム上では致命的な出来事ではない。よって無理にハードリアルタイムシステムにする必要はなく、ソフトリアルタイムシステムでも十分にデータの品質を保証できるのである。デッドラインミスより重要なことは、各処理の時間的制約を満たすようにI/O管理がおこなえるかどうか、また、複数のメディアストリーム間の同期がとれるかどうか、ということである。よってリアルタイム処理が必要な場所は、I/Oデータを処理する部分となる。

本システムではI/O処理部分のみリアルタイム処理を行う。これにより、従来のリアルタイムOSで使われる複雑な処理システムは必要なく、ストリームI/Oの品質は最低限保証することができる。

4.3　TS-I/Oの使用に必要なもの

アプリケーションはTS-I/Oを使うために、次で紹介する`read_gettbuf`システムコールと、ioctl命令`SNDCTL_DSP_EMPTYSET`を使用する。read_gettbufシステムコールを呼び出すことにより、TS-I/Oの提供する機能を使うことができる。ioctl命令 SNDCTL_DSP_EMPTYSET は read_gettbuf システムコールのデッドライン検知機能で必要なパラメータを設定する。

4.3.1　read_gettbufシステムコール

read_gettbufシステムコールを呼ぶことで従来の readシステムコールが行なう録音に加えて、デッドラインミスの通知のためにタイムスタンプを取得できる。この関数の形式は次の通りである。readシステムコールの形式と比較してみよう。

```
/* read_gettbuf(); */
ssize_t read_gettbuf(unsigned int fd, char *buf,
                     struct tftbuf *fromtobuf,
                     size_t count, int *pnt);

/* read(); */
ssize_t read(unsigned int fd, char *buf, size_t count);
```

read_gettbufシステムコールは引数として、通常の readシステムコールの引数に加えて、デッドラインミスの開始と終了を示すタイムスタンプを格納するバッファ fromtobufと、その最大の配列数を格納するポインタ pntを使う。

fromtobufはデッドラインミスの開始のタイムスタンプと終了のタイムスタンプの2つの値を格納できるように、新しく追加した構造体 struct tftbufを型とする。この構造体は以下のように実装した。

```
/*
 * include/linux/tftbuf.h
 */

struct tftbuf {
  unsigned long from;
  unsigned long to;
};
```

変数 from にデッドラインミスの開始を示すタイムスタンプを、変数 to にデッドラインミスの終了を示すタイムスタンプを表す。

read_gettbuf() の引数に fromtobuf を与えることで、アプリケーションはデッドラインミスのタイムスタンプを取得することができる。

デッドラインミスに対するリカバリー処理 -無音データの挿入-

また、この read_gettbuf() システムコールを呼ぶことにより、カーネルはデッドラインミスに対するリカバリー処理として、無音データを挿入する処理を行う。

アプリケーションが行う無音データの挿入に関して、図を使って簡単に説明しよう。図 4.1 は音飛びに対する通常のシステムの処理の様子と我々が提案するシステムの無音データを挿入する処理の様子を表したものである。通常のシステムでは音飛びが発生した場合、5 番、6 番のデータが破棄されるてカーネルバッファにデータがたまる。このデータをアプリケーションバッファが読み出して再生してみると、サウンドデータは 7 番以降のデータは実時間に対してずれて再生されてしまう。カーネルバッファはデッドラインミスを検知する機能がないので、どのデータが何番がわからないのである。

これに対し無音データの挿入処理では、アプリケーションバッファにたまっているデータのタイムスタンプを調べ、4 番から 7 番のデータが来るのはおかしい、デッドラインミスが生じたと判断し、4 番と 7 番の間に、5 番と 6 番分の無音のデータをアプリケーションに渡す。これによりできたサウンドデータは 5 番と 6 番のデータは抜けているものの、7 番以降のデータは実時間にそって再生される。

以上のように、read_gettbuf() システムコールを呼ぶことにより、アプリケーションバッファへのデッドラインミスのタイムスタンプの転送と、デッドラインミスに対する無音データの挿入処理を行うことができる。

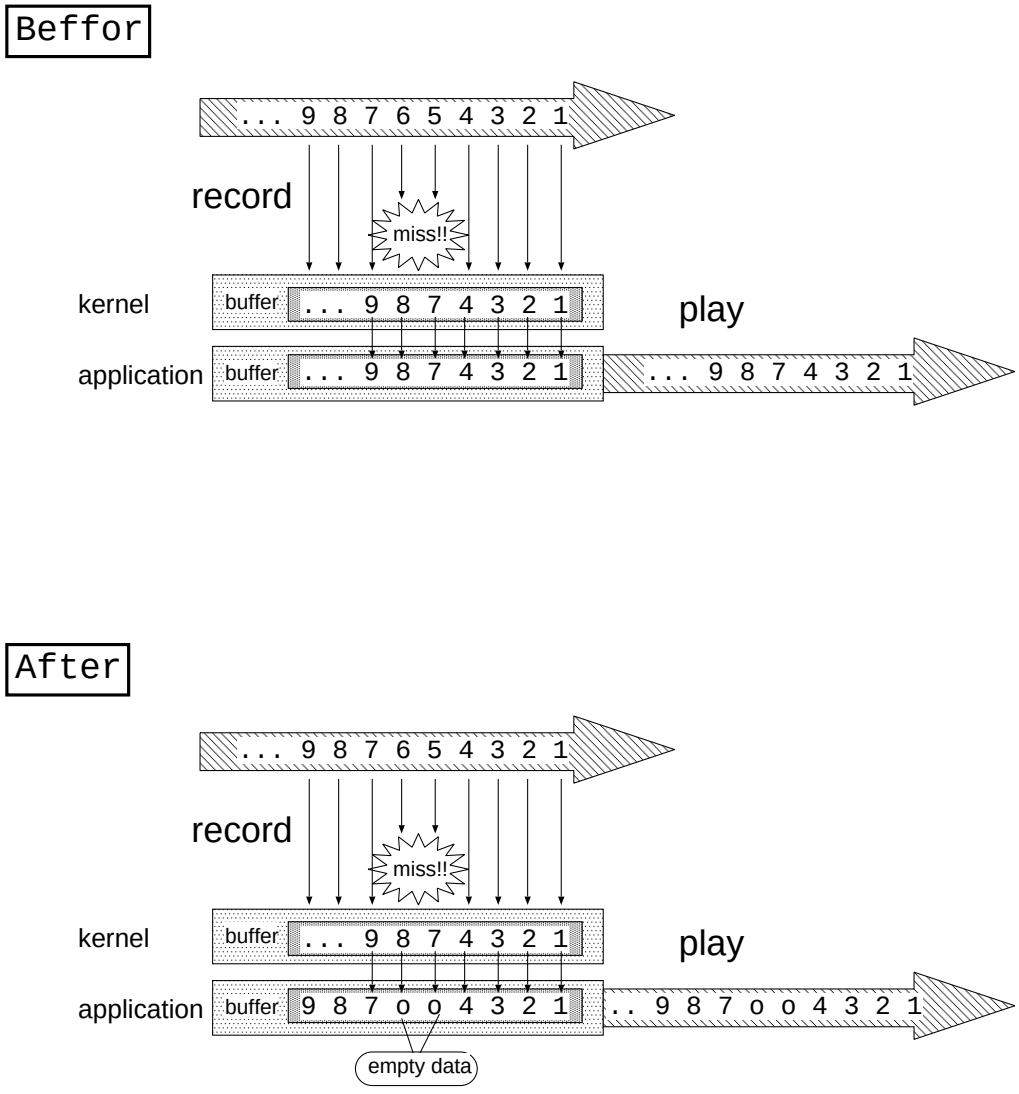
Beffor
... 9 8 7 6 5 4 3 2 1
record
miss!!
kernel
buffer
... 9 8 7 4 3 2 1
play
application
buffer
... 9 8 7 4 3 2 1
... 9 8 7 4 3 2 1
After
... 9 8 7 6 5 4 3 2 1
record
miss!!
kernel
buffer
... 9 8 7 4 3 2 1
play
application
buffer
9 8 7 o o 4 3 2 1
.. 9 8 7 o o 4 3 2 1
empty data

図 4.1: リカバリー処理

read_gettbuf() システムコールを使うには、ヘッダーファイルとして、<linux/read_gettbuf.h>が必要である。

4.3.2　ioctl システムコールの命令の追加

SNDCTL_DSP_EMPTYSET

サンプリングデータを取得できなかった時間のどのくらいまでをデッドラインミスと判断するか、その時間を設定するための命令を追加した。
単位はマイクロ秒である。たとえば、

```
unsigned long arg = 10000;
ioctl(fd, SNDCTL\_DSP\_EMPTYSET, &arg);
```

とアプリケーション側が設定すると、10 ミリ秒以上の間サンプリングデータを取得できなかった場合をデッドラインミスと判断する。
このデッドラインを判断する時間のデフォルトの値を 10000 マイクロ秒 = 10 ミリ秒とした。

4.4　実装詳細

我々が機能を追加する対象とした OS は、linux-2.2.19 である。このサウンドデバイスファイルである linux/drivers/sound/es1371.c の録音処理部分に、I/O レベルでの実時間処理機能を追加した。現在、linux-2.4 系がリリースされているし、es1371.c は多くあるサウンドデバイスファイルの 1 つにすぎないが、今回追加した機能はどのシステムでも追加することは可能である。

デバイスドライバ内の関数

追加した機能の詳細を説明する前に、簡単にデバイスドライバ内の関数について説明する。
デバイスドライバには各システムコールに対応する関数が用意されている。アプリケーション側が例えば read システムコールを呼ぶと、カーネル内の fs/read_write.c の関数 sys_read() を通じて、今回の es1371.c に関して言えば es1371_read() が呼び出される。同様に es1371.c にはシステムコール open()、write()、ioctl()、close() に対してそれぞれ、es1371_open()、es1371_write()、es1371_ioctl()、es1371_release() が用意されている。

es1371_read()

es1371_read() はカーネルバッファからアプリケーションバッファにサウンドデータのコピーを行う関数である。以下に大まかな実装を書く。

```
/* es1371_read(); */
static ssize_t es1371_read(struct file *file, char *buffer,
                           size_t count, loff_t *ppos)
{
  struct es1371_state *s = (struct es1371_state *)file->private_data;
  unsigned swptr;
  int cnt;
  ...
  while (count > 0) {
    ...
    cnt = ... /*
               *  カーネルバッファにたまっていて、まだアプリケー
               *  ションバッファにコピーしていないデータサイズ
               */
     ...
     if (cnt <= 0) {
       ...
       schedule(); /* 再スケジューリングを行う */
       ...
     }
   /*
    * データのコピーを行う。
    * bufferはアプリケーションバッファ、s->dma_adc.rawbuf + swptr
    * はカーネルバッファのポインタ、cntはコピーするサイズを指す。
    */
     if (copy_to_user(buffer, s->dma_adc.rawbuf + swptr, cnt)) {
       ...
     }
     ...
     count -= cnt;
     buffer += cnt;
     ...
  }
  ...
}
```

es1371_read() はカーネルバッファにデータがたまっているかどうか、変数 cnt の値をチェックし、データがたまっていればそのデータだけカーネルバッファからアプリケーションバッファにコピーする。

データがカーネルバッファにたまっていなければ、データがたまるのを待つ（ブロッキング）ために schedule() 関数を呼ぶ。schedule() 関数は再

スケジューリングを行う関数で、プロセスがこの関数を呼ぶと他のプロセスに CPU の実行権を与え、schedule() 関数を呼んだプロセスはしばらく休眠状態にはいる。このようなカーネルバッファのチェックを繰り返し、指定されたデータサイズまでカーネルバッファからアプリケーションバッファへデータをコピーする。

音飛びが生じてしまうのは、負荷によりこのプロセスの休眠状態が長くなることが原因である。負荷がかかると CPU を与えなければならない処理が増え、再び CPU の実行権が回ってくるのが遅くなり、そのあいだにカーネルバッファが I/O データで溢れ返ってしまう。

es1371_interrupt() システムコール

次に、I/O 割り込みが生じた際にその処理を行う es1371_interrupt()—を説明する。この関数は、処理された I/O データはどのくらいか、例えば録音なら I/O データがカーネルバッファにどのくらいたまったかを調べる。以下に大まかな実装を書く。

```
/*
 * linux/drivers/sound/es1371.c
 */

/* void es1371_interrupt(); */
static void es1371_interrupt(int irq, void *dev_id,
                             struct pt_regs *regs)
{
  struct es1371_state *s = (struct es1371_state *)dev_id;
  ...
  es1371_update_ptr(s);
  ...
}
-----------------------------------------------------------------
/* es1371_update_ptr(); */
static void es1371_update_ptr(struct es1371_state *s)
{
  int diff;
  /* update ADC pointer */
  if (s->ctrl & CTRL_ADC_EN) {
    diff = get_hwptr(s, &s->dma_adc, ES1371_REG_ADC_FRAMECNT);
    s->dma_adc.total_bytes += diff;
    s->dma_adc.count += diff; /* カーネルバッファにたまったデー
タ */

   ...
  }
  /* update DAC1 pointer */
```

```
  if (s->ctrl & CTRL_DAC1_EN) {
    ...
  }
  /* update DAC2 pointer */
  if (s->ctrl & CTRL_DAC2_EN) {
    ...
  }
}
------------------------------------------------------------------
/*
 * get_hwptr();
 * 割り込みで処理された I/O データのサイズを返す。
 */
extern inline unsigned get_hwptr(struct es1371_state *s,
                                 struct dmabuf *db, unsigned reg)
{
  unsigned hwptr, diff;

  outl((reg >> 8) & 15, s->io+ES1371_REG_MEMPAGE);
  hwptr = (inl(s->io+(reg & 0xff)) >> 14) & 0x3fffc;
  diff = (db->dmasize + hwptr - db->hwptr) % db->dmasize;
  db->hwptr = hwptr;
  return diff;
}
```

es1371_interrupt() は es1371_update_ptr() を実行する。es1371_update_ptr() は get_hwptr() により得られたデータサイズを取得し、まだアプリケーションバッファにコピーされていないカーネルバッファのデータサイズ s->dma_adc.count を更新する。es1371_read() はこの変数をみて、カーネルバッファからアプリケーションバッファにどのくらいサウンドデータをコピーするかを決める。

4.4.1　タイムスタンプ

タイムスタンプに必要なのは現在時間を取得する処理である。この処理を行うために、カーネル内に新しく関数 test_timer() を追加した。

```
/*
 * test_timer();
 */

unsigned long long test_timer() {
  struct timeval tv;
  do_gettimeofday(&tv);
  return (unsigned long long)(tv.tv_sec)*1000000+tv.tv_usec;
```

```
}
```

test_timer() は gettimeofday システムコールのカーネル内の本体である do_gettimeofday() を使用する。gettimeofday システムコールを呼ぶと、UNIX 時間の秒とマイクロ秒を取得することができる。UNIX の内部では、時刻はすべて UNIX 時間で管理されている。UNIX 時間というのは、1970 年 1 月 1 日 0 時 0 分 0 秒からの経過時間のことである。このシステムコールの精度はマイクロ秒といわれている。

この関数を割り込み処理が行われるごとに呼び出して各時刻を比較することで、割り込み処理の周期がわかる。

4.4.2　割り込み処理部分での追加処理

デッドラインミスの検知

割り込み処理部分ではデッドラインミスが生じたかどうか判断し、その時間はどのくらいのものか調べる処理を行う。

es1371.c の割り込み関数 es1371_interrupt() 内で、追加した関数 test_timer() から現在時刻を取得して、その時刻を割り込みが生じた時刻としてタイムスタンプで記録するようにした。タイムスタンプの時間データは、test_timer() を呼んで得られた時刻から、録音を開始した時刻を差し引いたものとした。つまりタイムスタンプの時刻は録音開始時刻を 0 としたときの経過時刻である。

割り込み関数内では次に、現在のタイムスタンプの時刻から前回のタイムスタンプの時刻を差し引いた時間を得る。そして、その時間分のデータサイズと、前回の割り込み処理の後にカーネルバッファにたまった I/O データのサイズを比較する。I/O のサイズに比べて前回の割り込み処理から今回の割り込み処理までの経過時間が大幅に大きい場合、デッドラインミス、音飛びが生じたと判断する。これは I/O 割り込みの周期性に反しているからである。

図 4.2 は、割り込み関数が呼ばれた時間を示すタイムスタンプと、そのときカーネルバッファにたまった I/O データのサイズのグラフである。Line1 は音飛びの生じない通常の処理の場合のグラフである。Line2 は音飛びが生じてしまった処理の場合のグラフである。いずれもサンプリングレートを 48,000Hz、サンプルサイズを 16bit、ステレオと設定し、3 秒の録音を行った。3 秒データの総サイズは、

```
3sec × 48,000sample/sec × 16bit/sample ／ 8bit/byte × 2channel
 = 576,000byte
```

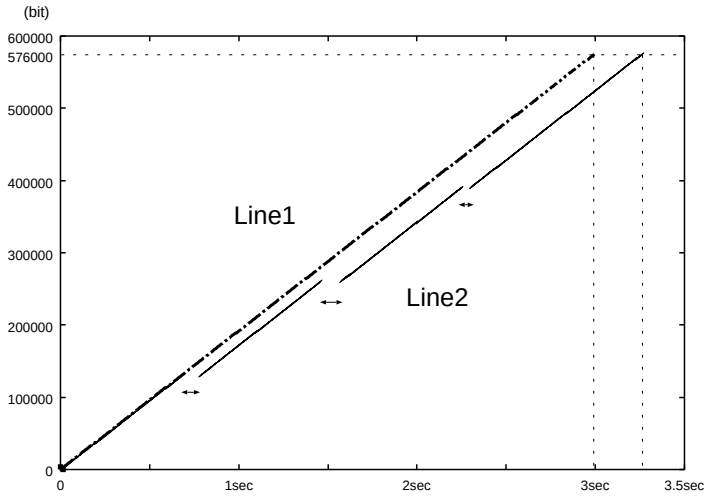

図 4.2: タイムスタンプとカーネルバッファにたまったデータサイズ

である。それに対し割り込みの周期は 5,333 μ sec であった。ちなみに 5,333 μ sec のデータサイズは 1,024byte である。

Line1、Line2 共にその周期は基本的には変わらないが、Line2 の場合、途中で 100,000 μ sec 前後の間が空いてから、また通常のように割り込みが生じてカーネルにバッファがたまっていっている。これはデッドラインミスにより割り込みの遅延が生じていることを指す。この空いてしまった間のデータはカーネルバッファにはたまらず破棄されている。その結果、Line2 は 3 秒のデータを録音するのに 3 秒以上かかっている。この誤差はデッドラインミスで空いてしまった総時間と同じである。

音飛びが生じたら、現在のタイムスタンプからカーネルバッファにたまった I/O データ分の時間を引いたもの、つまり、I/O データがカーネルバッファに改めてたまり始めた時間、それと前回のタイムスタンプをバッファに記録する。このバッファは後でアプリケーションが、アプリケーション側のバッファにコピーする。これにより、アプリケーション側はバッファを調べることで、いつからどのくらいまで音飛びが発生したか知ることができる。

```
/*
 * linux/drivers/sound/es1371.c
 */

<linux/timefromtofile.h>

static void es1371_update_ptr(struct es1371_state *s)
```

```
{
  int diff;
  unsigned rate, size, channels;

  /* update ADC pointer */
  if (s->ctrl & CTRL_ADC_EN) {
/*
 * タイムスタンプ処理
 * s->dma_adc.timestart は es1371_read() の始めで設定
 */
    s->dma_adc.timeofnow
    = (unsigned long) (test_timer() - s->dma_adc.timestart);
    diff = get_hwptr(s, &s->dma_adc, ES1371_REG_ADC_FRAMECNT);
    ...
/*
 * read_gettbuf() を呼んだかどうかを調べる。
 * s->settimestamp は es1371_gettbuf() で設定
 */
    if(s->settimestamp > 0){
      rate = s->adcrate;          /* sample rate */
      size = s->adcsize;          /* 追加。 sample size */
      channels = s->adcchannels;  /* 追加。 sample channels */
  /*
   * 割り込みで処理したデータサイズがどのくらいの
   * 時間分かを表す。
   * μ sec= μ sec/sec × bit/byte /channels × sec/数× channels
   *       × byte ×数/bit
   */
      difftime = 1000000 * 8 / size / channels * diff / rate;

  /*
   * デッドラインミスの検知処理
   * s->time_interval は es1371_ioctl() の SNDCTL_DSP_EMPTYSET
   * 命令で設定
   */
      if((s->dma_adc.timeofnow - s->dma_adc.timeofprev)
                        >= s->time_interval + difftime){
        /* デッドラインミスの開始時刻 */
        timefromtobuf[s->dma_adc.timefromtocnt].from
        = s->dma_adc.timeofprev;
        /* デッドラインミスの終了時刻 */
        timefromtobuf[s->dma_adc.timefromtocnt].to
        = s->dma_adc.timeofnow - difftime;
        /*
         * デッドラインミス分の無音データのサイズを設定
         * デッドラインミスの開始時刻から終了時刻までの時間分の
         * データサイズを求める。
         */
        s->dma_adc.empty_sounddata[s->dma_adc.empty_sound_number]
        = rate*size*channels/8/1000
```

```
          *(timefromtobuf[s->dma_adc.timefromtocnt].to
            - timefromtobuf[s->dma_adc.timefromtocnt].from)/1000;
        ...
        s->dma_adc.timefromtocnt ++;
        s->dma_adc.empty_sound_number ++;
      }
    }
    s->dma_adc.timeofprev = s->dma_adc.timeofnow;
    /* デッドラインミス以降の I/O データのサイズ */
    s->dma_adc.sounddata[s->dma_adc.empty_sound_number] += diff;
    ...
  }
  ...
}
```

まず、test_timer() システムコールを呼び、現在の時刻のタイムスタンプを得る。そして、この録音は read_gettbuf() システムコールによるものかどうか、s->settimestamp をチェックする。変数 difftime は、今回の割り込みで処理された I/O データのサイズを表す変数 diff がどのくらいの時間分のものかを表す。そして、前回の割り込み関数のタイムスタンプ s->dma_adc.timeofprev と今回の割り込み関数のタイムスタンプ s->dma_adc.timeofnow を比較し、I/O データを取得していない時間が s->interval 以上かどうか調べる。s->interval を越えた場合、デッドラインミスと判断して カーネルバッファ timefromtobuf にデッドラインミスの開始時刻と終了時刻を代入する。

デッドラインミスの時間がわかったら、無音データを es1371_read() で挿入する処理をするために、デッドラインミスの時間分のデータサイズを無音データのサイズに設定する。

4.4.3　録音処理部分での追加処理

デッドラインミスに対するリカバリー処理 -無音データの挿入-

録音処理部分ではデッドラインミスの生じた部分に無音データを挿入する処理を行う。

録音処理部分である es1371_read 関数では、カーネルバッファのデータをアプリケーションバッファにコピーする前に、無音データを挿入しなければいけないかどうか確認する。デッドラインミスが生じるごとに、それ以降の I/O データを s->dma_adc.copy_number で番号分けをする。この番号を見ることで、カーネルバッファの任意の I/O データがどのデッドラインミスの後からカーネルバッファにたまったものか認識できる。

```
/* es1371_read(); */
static ssize_t es1371_read(struct file *file, char *buffer,
                           size_t count, loff_t *ppos)
{
  ...
  while (count > 0) {
    ...
    /*
     * s->settimestamp:無音データを挿入するかどうかの確認
     * copy_number:何回目のデッドラインミスかを示す。
     * このデッドラインミスの直前のストリーム I/O にもこの
     * copy_number が割り当てられる。
     */
    if (s->settimestamp > 0 && s->dma_adc.empty_sounddata[s->dma_adc.
copy_number] > 0 && s->dma_adc.sounddata[s->dma_adc.copy_number] <= 0) {
      cnt = s->dma_adc.empty_sounddata[s->dma_adc.copy_number];
      if (cnt > count)
        cnt = count;
      /*
       * 無音データの挿入
       * 無音データ =
       * case 16bit: 0 の列
       * case 8bit: 128 の列
       */
      if (s->adcsize == AFMT_S16_LE)
        memset(buffer, 0, cnt);
      else
        memset(buffer, 128, cnt);

      spin_lock_irqsave(&s->lock, flags);
      s->dma_adc.empty_sounddata[s->dma_adc.copy_number] -= cnt;
      /* 無音データをすべて挿入したかどうか確認 */
      if (s->dma_adc.empty_sounddata[s->dma_adc.copy_number] <= 0)
        s->dma_adc.copy_number ++;
      spin_unlock_irqrestore(&s->lock, flags);
    } else {
      /* 通常の I/O データのコピー処理 */
      ...
      if (copy_to_user(buffer, s->dma_adc.rawbuf + swptr, cnt)) {
        ...
      }
    } ...
    count -= cnt;
    buffer += cnt;
    ...
  } ...
}
```

挿入する無音データとは、データサイズが 8 ビットの場合は数値 128 の

データ列、16 ビットの場合は数値 0 のデータ列である。無音データを挿入した後は、それ以降の I/O データのコピーを通常どおり行う。

4.4.4　read_gettbuf システムコール

デッドラインミスの通知

割り込み処理部分はデッドラインミスに気づいた場合、デッドラインミスが発生した前後のタイムスタンプを用意したカーネルバッファに記録する。カーネルバッファにたまったサウンドデータをアプリケーションバッファにコピーするのと同様に、このカーネルバッファにたまったデッドライミスのタイムスタンプをアプリケーションバッファにコピーしなければならない。それを行うために、read システムコールにその機能を追加した read_gettbuf システムコールを新しく追加した。read_gettbuf システムコールは linux/fs/read_write.c で宣言する。

```
/*
 * linux/fs/read_write.c
 */

#include <linux/read_gettbuf.h>
#include <linux/timefromtofile.h>

/* タイムスタンプ用のカーネルバッファ */
struct tftbuf timefromtobuf[500];

/* sys_read adding gettimebuf */
asmlinkage ssize_t sys_read_gettbuf(unsigned int fd, char * buf,
struct tftbuf * fromtobuf, size_t count, int * pnt)

{
        ssize_t ret;
        struct file * file;
        ssize_t (*read)(struct file *, char *, size_t, loff_t *);
        int (*gettbuf)(struct file *);
        int empty_num1, empty_num2, copy_num;
        int i;

        ...
        if (!file->f_op || !(gettbuf = file->f_op->gettbuf))
           goto out;
        empty_num1 = gettbuf(file);
        if (!file->f_op || !(read = file->f_op->read))
           goto out;
        ret = read(file, buf, count, &file->f_pos);
```

```
        if (!file->f_op || !(gettbuf = file->f_op->gettbuf))
           goto out;
        empty_num2 = gettbuf(file);

        if (*pnt > empty_num2 - empty_num1)
           copy_num = empty_num2 - empty_num1;
        else
           copy_num = *pnt;
    /*
     * タイムスタンプのコピー
     * fromtobuf はアプリケーションバッファ、timefromtobuf は
     * カーネルバッファを指す。
     */
        for(i = 0; i < copy_num; i++) {
          copy_to_user(fromtobuf+i, timefromtobuf+empty_num1+i,
                       sizeof(*fromtobuf));
        }
       *pnt -= copy_num;
out:
       fput(file);
bad_file:
       unlock_kernel();
       return ret;
}
```

read_gettbuf システムコールが呼ばれると、この sys_read_gettbuf を通して es1371.c の関数が呼ばれる。、通常の read システムコールと同様に es1371_read() 関数を呼ぶ前後で追加した関数 gettbuf() を呼ぶ。この関数は es1371.c に追加した関数 es1371_gettbuf() 関数を呼び、現在デッドラインミスが何回生じたか、その数を返す。

read システムコールの前後で呼ぶ gettbuf システムコールの返り値 empty_num1、empty_num2 に差があれば、その差は read の処理でデッドラインミスが生じた回数を表す。このデッドラインミスの回数と、pnt が指すアプリケーションバッファの最大の配列数を比較し、デッドラインミスの回数のほうが小さければその数だけ、いつ発生したデッドラインミスかを示すタイムスタンプをカーネルバッファからアプリケーションバッファにコピーする。デッドラインミスの回数のほうが大きければ、 pnt に格納されている値だけデッドラインミスのタイムスタンプをカーネルバッファからアプリケーションんバッファにコピーする。そして、pnt に格納されている値からその数を引く。これにより、アプリケーションは pnt に格納されている値を調べることで、その時点で何回デッドラインミスが生じているかを知ることができる。

カーネルバッファからアプリケーションバッファにコピーしたデータの数やサイズは本来、システムコールの返り値でアプリケーション側に返すように設計されている。しかし、このシステムコールでは read システムコールと同様に、返り値として読み込んだサウンドデータのサイズを返さなければならない。このためデッドラインミスの回数は read_gettbuf システムコールの引数 *pnt のように、アプリケーションが指定したアドレスに格納することによってアプリケーション側に知らせるようにした。

4.4.5　ioctl システムコールの命令の追加

SNDCTL_DSP_EMPTYSET

割り込み処理部分で行うデッドラインミスの判定の際、サンプリングデータを取得できなかった時間がいくつ以上であるとき、デッドラインミスと判断するか、その時間を設定するための命令を追加した。

この命令の追加の設定は、include/linux/soundcard.h に新しい命令の名前と番号を追加し、es1371.c ファイルにある es1371_ioctl() 関数に命令に対する処理を記述することで行う。

es1371_ioctl() 関数内に記述した、SNDCTL_DSP_EMPTYSET に対する処理は以下のようにした。

```
/* es1371_ioctl(); */
/* 第 4 引数 arg の型を int から unsigned long に変更 */
static int es1371_ioctl(struct inode *inode, struct file *file,
                        unsigned int cmd, unsigned long arg)
{
  struct es1371_state *s = (struct es1371_state *)file->private_data;
  ...
  unsigned long longval; /* 追加した unsigned long 型の変数 */
  ...

  switch (cmd) {
  case ...
  ...

  /* SNDCTL_DSP_EMPTYSET */
  case SNDCTL_DSP_EMPTYSET:
          /* longval に *arg を代入 */
          if (get_user(longval, (unsigned long *)arg))
            return -EFAULT;
          if (longval >= 0) {
            if (file->f_mode & FMODE_READ) {
              stop_adc(s);
              s->dma_adc.ready = 0;
```

```
            s->time_interval = longval;
          }
          if (file->f_mode & FMODE_WRITE) {
            stop_dac2(s);
            s->dma_dac2.ready = 0;
            s->time_interval = longval;
          }
        }
        /* *arg に s->time_interval を代入 */
        return put_user(s->time_interval, (unsigned long *)arg);
  }
  ...
}
```

es1371_ioctl の引数 arg の型を int から unsigned logn に変えた。arg は設定したい値を格納する引数である。今までは int で十分だったが、今回設定する値は 100,000 や 1,000,000 などもありうるので unsigned long に変更した。

get_user() は第 1 引数に第 2 引数がさす値を代入する。put_user() はその逆で、第 1 引数の値を第 2 引数がさす値に代入する。

SNDCTL_DSP_EMPTYSET の処理は、es1371_state 構造体の追加した変数 s-¿time_interval に値を代入することである。この変数は割り込み処理部分でデッドラインミスの検知処理で使われる。

4.5　もうひとつのリカバリー処理方法

TS-I/O ではデッドラインミスに対するリカバリー処理は、read_gettbuf システムコールを呼ぶことでカーネル内で処理を行なっている。リカバリー処理とはここではサウンドデータの間に無音データを挿入する処理である。

TS-I/O では、サウンドデータをカーネルバッファからアプリケーションバッファにコピーする際、デッドラインミスの直前までのサウンドデータをアプリケーションバッファにコピーした後、デッドラインミス分の無音データをアプリケーションバッファにコピーする。無音データをコピーし終わったらデッドラインミス直後のサウンドデータは通常と同様の処理を行なう。このようにして無音データを挿入し、デッドラインミスに対するリカバリーを行なう。

これに対して、別の方法によるデッドラインミスに対するリカバリー処理、無音データを挿入する処理が考えられる。

通常はデッドラインミスが生じたかどうか、どこで生じたかどうかアプリケーションは知ることができないので、リカバリー処理はアプリケー

ションは行なうことができない。しかし TS-I/O が提供するデッドラインミスの検知機能とデッドラインミスの通知機能を使うことにより、アプリケーションがリカバリー処理を行なうことが可能である。

まず、アプリケーションはカーネルからデッドラインミスの生じているサウンドデータを取得する。TS-I/O によるデッドラインミスのタイムスタンプをもとに、このサウンドデータに無音データを挿入する。

無音データを挿入することで、サウンドデータが指定したサイズ以上のものになってしまう。これを避けるため、デッドラインミスが生じている場合にカーネルの録音処理部分で、アプリケーションが指定したデータサイズからデッドラインミス分のデータサイズを引いた、データサイズまで録音を行なう、すなわち、カーネルバッファからアプリケーションバッファに転送する。後でアプリケーションが無音データを挿入することで、サウンドデータは最初に指定したデータサイズのものになる。

わかりやすいように図 4.3 で説明する。アプリケーションがサイズ 9 のサウンドデータを録音しようとしている。このときデッドラインミスにより 5 と 6 のデータを取得できなかった場合、カーネルはアプリケーションにサイズ 7 のデータをコピーする。アプリケーションはこのデータをもらったあと、TS-I/O によるデッドラインミスのタイムスタンプをもとに無音データを挿入する。これにより、サイズ 9 の実時間にそったサウンドデータを再生することができる。

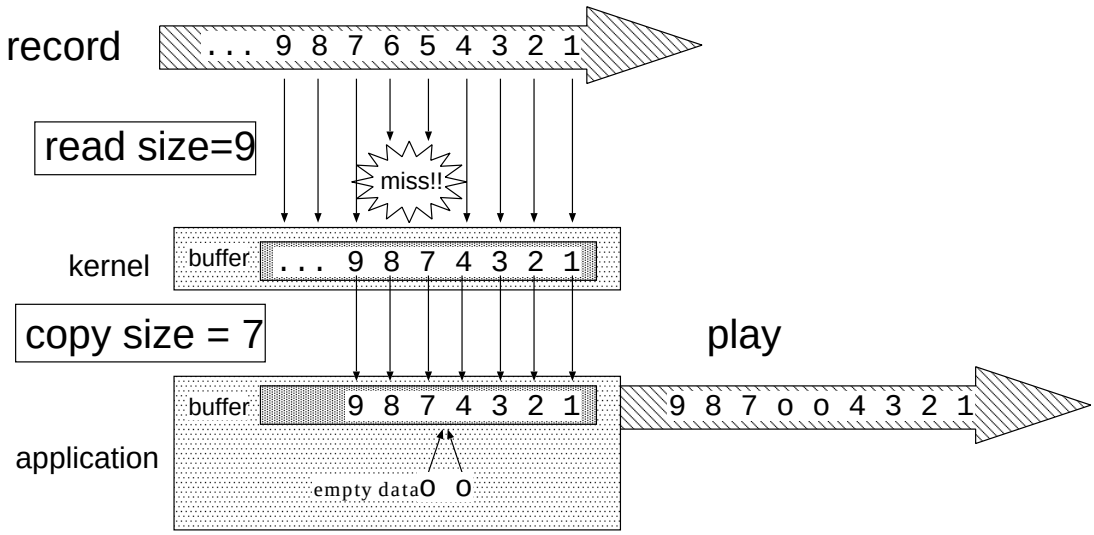

図 4.3: アプリケーションによるリカバリー処理

現時点では、このようなリカバリー処理のシステムの実装にはまだとりかかっていない。今後このようなライブラリの実装を行なう予定である。

第5章　実験

この章では実際に TS-I/O を使用し、それをもとに無音データを挿入したサウンドデータがどういうものになったかを示す。無音データがちゃんと挿入されて実時間にそった再生ができているかどうかを確認する一番の方法は、実際にできたサウンドデータを耳で聞きことである。本稿では、視覚的に確認するために、サウンドデータの波形を調べることによって確認する。

また、音飛びはどのぐらいの負荷をかけると生じるのか、それについて記述する。。

5.1　TS-I/O の使用例

5.1.1　サウンドデータの波形の比較

TS-I/O により無音データを挿入したサウンドデータと、音飛びが生じずに通常どおり録音されたサウンドデータ、それと従来の録音により音飛びしてしまったサウンドデータを比較する。

これらを比較する方法は 2 つある。1 つは、互いのサウンドデータを同時に再生し、実時間にそって同じタイミングで再生され続けるかどうか、途中でずれて再生されるかどうか、耳で聞いて比較する方法である。もう 1 つは、サウンドデータの波形を見比べ、同じ形かどうか比較する方法である。本稿では、サウンドデータの波形による比較を行う。

録音を行うプログラムは、サンプリングレート 48000Hz、サンプルサイズ 16bit、モノラルのサウンドデータを処理する。ステレオでは左チャンネルと右チャンネルの 2 つの波形を要するので、ここでは単純なモノラルを選んだ。

図 5.1 は音飛びが発生していない、通常のサウンドデータの波形である。音源元の曲調であるのだろうか、あるタイミングに対して一定の振幅が確認できる。

それに対し、図 5.2 は無音データを挿入したサウンドデータの波形である。図 5.1 と図 5.2 は同じ時間軸である。

図 5.2 の波形は、図 5.1 の波形の一部を空欄にしたような波形に見える。

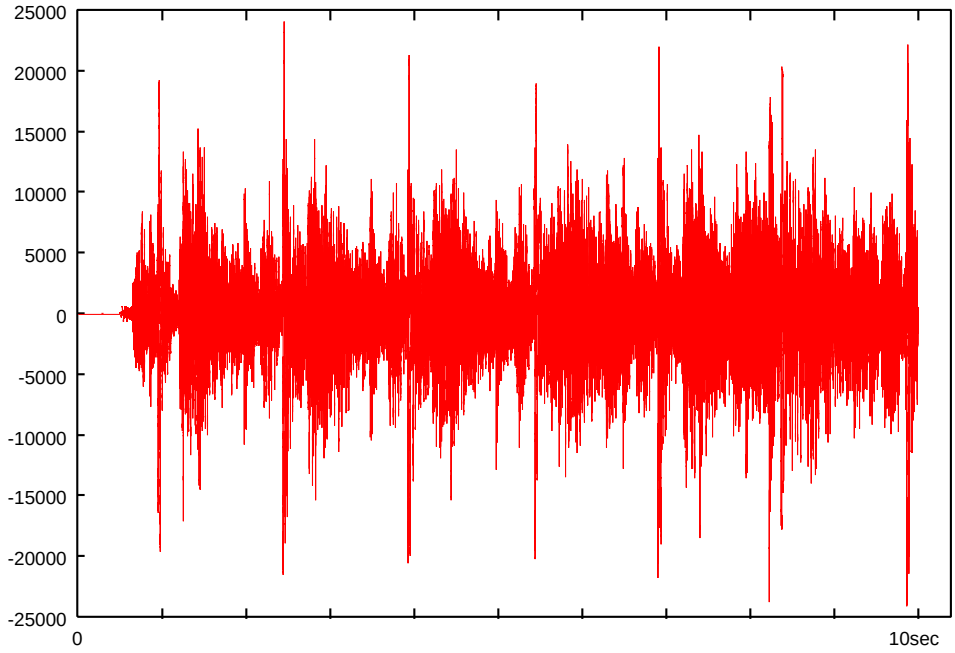

図 5.1: 通常時の波形

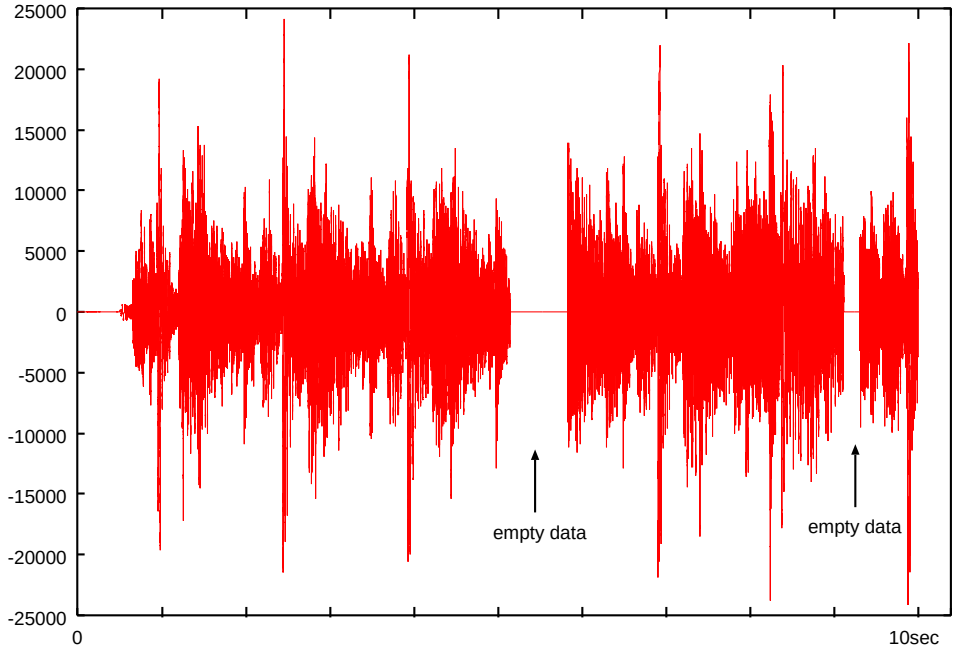

図 5.2: 無音データを挿入した波形

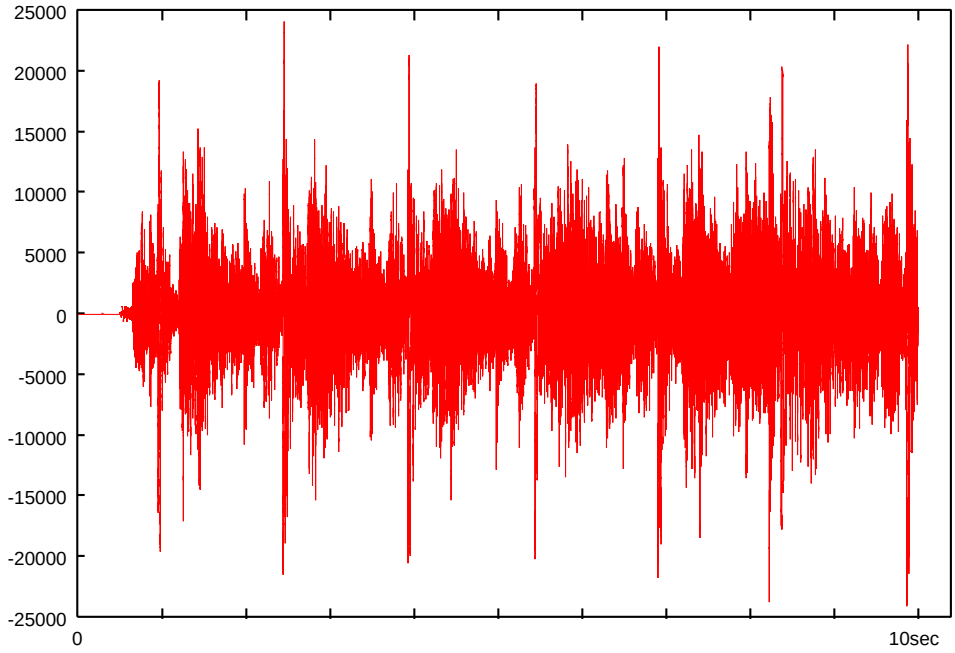

図 5.3: 通常時の波形

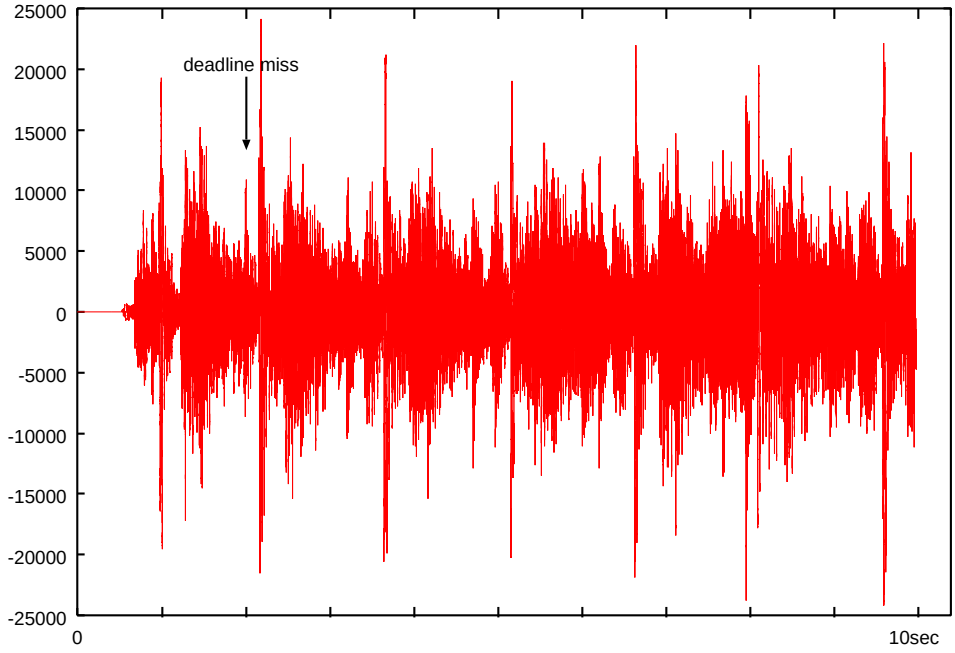

図 5.4: 音飛びした波形

この部分は値 0 のデータ列である無音データを挿入した部分である。無音データ以降のデータの波形が通常の波形と同じタイミングで振幅していることがわかる。このことは、無音データの挿入によりその後のデータが通常の実時間にそって再生されていることを示す。

図 5.4 の波形は通常の read システムコールで音飛びが生じてしまったサウンドデータの波形である。図 5.3 と図 5.4 は同じ時間軸である。図中の deadline miss でデッドラインミスが生じ、それ以降のデータは通常よりも早い時刻に再生されているのがわかる。

5.1.2　無音データの挿入処理で注意すること

無音データとはデータサイズが 8 ビットの場合は数値 128 のデータ列、16 ビットの場合は数値 0 のデータ列である。無音データを挿入する際に、単純にデッドラインミスの時間分のデータサイズをアプリケーションバッファに入れると問題が生じる場合がある。その問題はデータサイズとチャンネルによって異なる。

8 ビット、ステレオの場合

ステレオの場合、ストリームデータは右チャンネルと左チャンネルで交合に振り分けられる。8 ビットの場合 1 バイトがひとつのサウンドデータなので、例えば 124 125 126 127 128 129 ... とデータがあると 左チャンネルは 124 126 128...、右チャンネルは 125 127 129... のように振り分けられる。

この時、無音データが奇数のデータサイズの場合に問題が生じる。無音データを挿入したあとのサウンドデータはそれぞれ通常と逆のチャンネルに振り分けられてしまうのである。図**??**は音飛びによりチャンネルが逆転してしまったサウンドデータと、本来のデータの波形である。

図**??**の左半分は音飛びのない本来のデータの波形である。図**??**の右半分は音飛びによりチャンネルが逆転してしまったデータの波形である。それぞれ上の波形が左チャンネルを表し、下の波形が右チャンネルを表す。

このようなチャンネルの逆転を避けるには、無音データを偶数のデータサイズで挿入する必要がある。

16 ビット、モノラルの場合

16 ビットの場合、2 バイトがひとつのサウンドデータである。ひとつのサウンドデータは上位桁の 1 バイトと下位桁の 1 バイトをもつ。

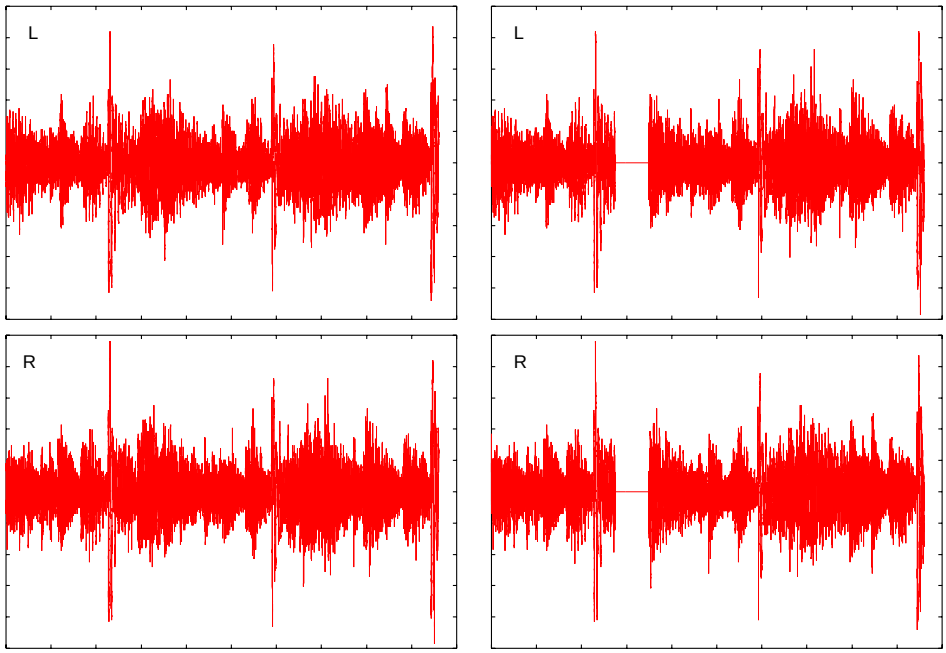

図 5.5: ステレオでの問題 -チャンネルの逆転-

このとき、無音データが奇数のデータサイズの場合に問題が生じる。上位桁の値と下位桁の値が逆転してしまうのである。これにより、サウンドデータは本来のものとまったくことなるものになってしまう。

図 5.6 はこの上位桁と下位桁の値の逆転が起ってしまったサウンドデータの波形である。デッドラインミスに d1、d2、d3、d4 と名前をつけた。d1 と d2 では挿入した無音データのサイズは偶数だったので問題は生じなかった。しかし、d3 で奇数のサイズの無音データを挿入してしまったのであとのデータはその前までのサウンドデータとは異常に違ったものとなってしまった。ここで、d4 で奇数のサイズの無音データが挿入されたので、上位桁と下位桁の値は再び元に戻った。

このような問題を解決するには、ステレオでの問題のときと同様に、無音データを偶数のデータサイズで挿入する必要がある。

16 ビット、ステレオの場合

16 ビットでステレオの場合はストリームデータは、左チャンネル、右チャンネルに交互に 2 バイトずつ振り分けられる。

この場合、前に述べたステレオの問題と 16 ビットの問題の両方を考えなければならないので、挿入する無音データのサイズは 4 の倍数にする必

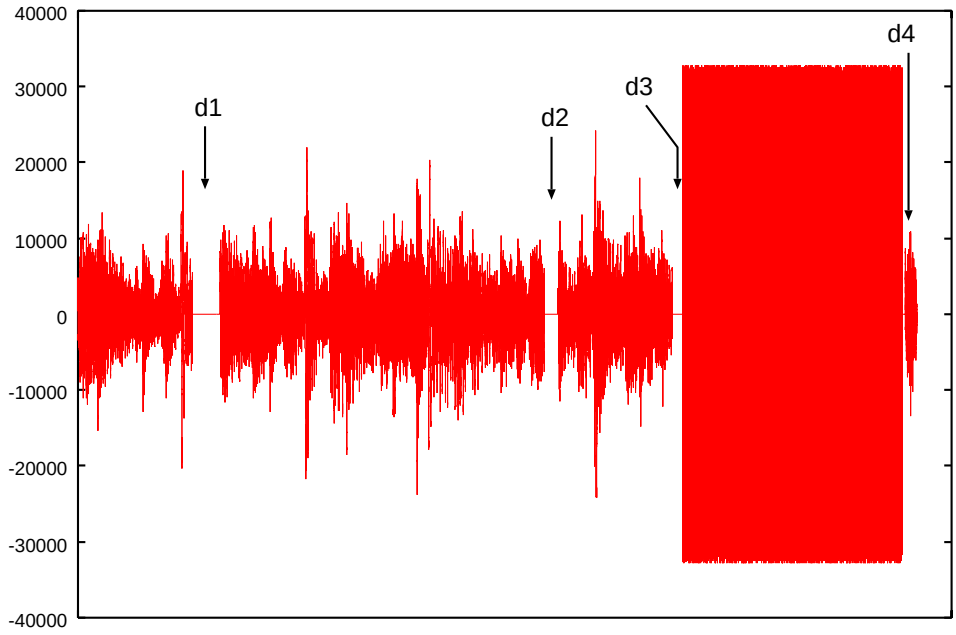

図 5.6: 16 ビットでの問題 -上位桁と下位桁の逆転-

要がある。

5.2　デッドラインミスが生じる負荷

音飛びの原因は、マシンにかかる過負荷である。どのくらいの負荷をかければ音飛びが生じるのかを実験して調べてみた。実験は以下の環境で行った。[9]

- 実験に使用した計算機：Pentium III 733MHz メモリ 512MB
- 負荷をかけるためのソフト：pi_fftc １つから６つ同時に起動
- 録音を行うプログラム：read_gettbuf システムコールを n 回呼び 10 秒録音するプログラム

ソフト pi_fftc をいくつか動かしながら、 read_gettbuf システムコールを 10 秒 n 回呼ぶプログラムを動かし、音飛びが発生するかどうかを調べた。録音データはサンプリングレート 48000Hz、サンプルサイズ 16bit、ステレオという最も大きいサイズの設定にした。この録音プログラムでは read_gettbuf システムコールと read_gettbuf システムコールでは余計な処理は行わないようにした。また、録音プログラムの優先度を一時的に下げ

	nice 0	nice 5	nice 10	nice 15	nice 20
n=1	音飛びなし	pi5	pi4	pi4	pi3
n=10000	音飛びなし	pi4	pi4	pi3	pi2
n=192000	pi6	pi4	pi3	pi1	-
n=960000	pi3	pi1	-	-	-
n=1920000	pi1	-	-	-	-

表 5.1: read_gettbuf() の数と優先度に対する、音飛びする負荷の量

た場合もコマンド nice を使って調べた。優先度とは-20 から 20 までの値であり、小さければ小さいほど優先される。

表 5.1 は、録音プログラム中の read_gettbuf システムコールを呼ぶ回数とコマンド nice による優先度の変更に対して、音飛びが発生するのはソフト pi_fftc をいくつ同時に実行したときかを表す。表中の「pi 数字」の数字はソフト pi_fftc の数を示す。

表 5.1 をみてわかるように、優先度を下げないと音飛びはほとんど発生しない。優先度を下げない録音処理では、割り込み遅延はみられず安定して録音することができた。10 秒のデータサイズは 1920000 である。つまり n=960000 や 1920000 のように、read_gettbuf システムコールを十分多く呼び出してやっと音飛びする。このことから、現在のマシン性能ならカーネルでの処理での音飛びはほとんど発生しないと言える。

ではなぜ、音飛びが現実に発生するのだろうか。それは、アプリケーションの処理が問題である。システムコールとシステムコールの間で別の処理を行っているうちに、その間の I/O データを処理できなくなるなど、アプリケーションレベルでのデッドラインミスが生じることが考えられる。

現在マルチメディアの発展とマシンの高性能化に伴い、アプリケーションがあらゆる機能を集約的に実行する設計の傾向がある。アプリケーションがさまざまな仕事をこなすにつれて、このようなデッドラインミスの危険性が増してしまうと考えられる。

第6章　まとめ

マルチメディアアプリケーションへの要求が高まり、実時間処理を行うリアルタイム OS が注目されている。リアルタイム OS はもともと組み込み分野で扱われていたシステムだが、現在ではマルチメディアの発展、パソコンの普及などから、従来の汎用 OS をリアルタイム化したリアルタイム OS が開発されている。

これらのリアルタイム OS は、時間制約の厳しいハードリアルタイム処理を目的としている。デッドラインミスを制御し、リアルタイムタスクによるリアルタイム処理を行う。

しかしこれらのリアルタイム OS のようなハードリアルタイム処理を実現するのは難しく、この実現のためには複雑なシステムを用意しなければならない。

そこで我々は、I/O に限定したリアルタイム化を行い、ストリームデータのリアルタイム処理を行うシステム TS-I/O を提案した。タイムスタンプ付ストリーム I/O を利用することで、デッドラインミスを検知し、その情報をアプリケーションに伝えることが可能である。また、デッドラインミスに対するリカバリー処理である無音データの挿入処理を行い、ストリーム I/O の実時間制を保つことができる。

セクション 4-5 で述べたように、リカバリー処理をアプリケーションが行なう方法がある。この処理を行なうライブラリの実装が今後の課題である。

TS-I/O と現存するリアルタイム Linux を比較してみる (表 6.1)。TS-I/O のメリットはシステムのシンプルなところである。I/O のみのリアルタイム化なので OS 全体に手を加える必要がなく、ユーザーは従来の Linux のシステムをそのまま使用することができる。

OS	システムの複雑さ	カーネル保護	実時間性
RT-Linux	複雑	×	ハードリアルタイム
ART-Linux	複雑	○	ハードリアルタイム
TS-I/O	シンプル	○	I/O のみのソフトリアルタイム

表 6.1: 現存するリアルタイム Linux と TS-I/O の比較

しかし、このシステムのシンプルさはデメリットでもある。I/O に限ってしかリアルタイム処理が行なえず、提供できる機能には限りがあるということである。機能を追加すればそれだけシステムは複雑となる。現在のシステムのシンプルさをできるだき維持しながら、どれだけの機能を追加できるかが今後考えなければならないことである。

今回は音の録音についてのリアルタイム処理のシステムを提案した。今後は音の再生についてのリアルタイム処理や、音以外のメディア、例えば動画のような I/O のリアルタイム処理について TS-I/O を適応させる研究を試みるつもりである。

参考文献

[1] : FSMLabs Community, http://fsmlabs.com/community/.

[2] : ITRON Project Home Page (in Japanese), http://tron.um.u-tokyo.ac.jp/TRON/ITRON/home-j.html/.

[3] : Lynux Works, http://www.lynuxworks.com/.

[4] : ウインドリバー株式会社, http://www.windriver.co.jp/.

[5] Beck, M., Dziadzka, M., Magnus, R., Kunitz, U., Dirk Verworner 著, 株式会社クイック訳: Linux カーネルインターナル, 株式会社ピアソン・エデュケーション (1999).

[6] 今村義幸: ポスト PC 時代のキーワード「エンベデッド」のすべて, http://www.kumikomi.net/article/explanation/2001/03postpc/.

[7] 森下功: Linux によるリアルタイム処理の可能性を考える, インターフェース, Vol. 25, No. 11, pp. 68–71 (1999).

[8] 藤倉俊幸: 特集 産業用オペレーティングシステム完璧マスタ, インターフェース, Vol. 26, No. 6, pp. 52–58 (2000).

[9] 大浦拓哉: FFT と AGM による円周率計算プログラム, http://www.kurims.kyoto-u.ac.jp/~ooura/pi_fft-j.html.

[10] Jeff Tranter 著, 山形 浩生訳: Linux マルチメディアガイド, オライリージャパン (1997).

[11] 石綿陽一: ART-Linux における実時間処理と Linux kernel 2.2 シリーズ対応版について, http://www.movingeye.co.jp/~you1/art-linux/rlc200101/text.html.

[12] 石綿陽一: ART-Linux の誕生の経緯と使い方, インターフェース, Vol. 25, No. 11, pp. 109–118 (1999).

[13] 石綿陽一: リアルタイム処理を実現する ART-Linux の設計と実装, インターフェース, Vol. 25, No. 11, pp. 119–129 (1999).

[14] 船木陸議: 特集 産業用 OS としての Linux 活用法, インターフェース, Vol. 27, No. 6, pp. 65–85 (2000).