平成15年度修士論文

感性を考慮した
ジョブスケジューリング

東京工業大学 情報理工学研究科
数理・計算科学専攻

学籍番号 02M37124

栗田 亮

指導教官

千葉 滋 助教授

平成16 年2 月24 日

概要

本稿では重要性の低いマルチメディア処理に対してユーザの感性を考慮したQoS 処理を行う新しいスケジューリングシステム Tanma について述べる。マルチメディア処理に対する従来の QoS 処理は、重要性の低いプロセスを犠牲にして、重要性の高いマルチメディア処理に常に一定以上の計算資源を割り当てるためのものであった。しかし近年では、BGM のように状況によっては処理を持続できなくても構わないとされる、重要性の低いマルチメディア処理が増えている。より重要性の高いプロセスが存在する場合は、そのような重要性の低い処理を行うプロセスから計算資源を迅速に移さなければならないが、急激に移し過ぎるとマルチメディア処理の場合、ユーザが不快に感じることがある。

そこで我々はこのような重要性の低いマルチメディア処理に対して、ユーザの感性を考慮した QoS 処理を行うジョブスケジューリング手法を提案し、この手法を実現するシステム Tanma を Linux 上に実装した。Tanma は音楽再生プロセスを処理の対象としている。Tanma は progress-based regulation と呼ばれる手法をもとにスケジューリングを行う。音楽再生プロセスの進捗状況を調べることで重要性の高いプロセスとの資源の競合を探知し、音楽再生プロセスを一定時間停止させる。これにより音楽再生プロセスに使われている資源を解放することができ、重要性の高いプロセスを優先して処理することが可能である。この際に、マルチメディアデータが途切れながら再生されないように即座に停止させることにより不快感をユーザに与えないようにする。停止させる際には音楽をフェードアウトさせ、再開させる際には音楽をフェードインさせることにより、突然音楽が途切れたり再開されたりすることによるユーザへの不快感を抑えることが可能である。さらに、フェードアウト前の部分から音楽が再開されるようにすることで (タイムマシン再生)、ユーザがフェードアウトのせいで音楽の一部を聴けないという事態を防ぐ。

Tanma はカーネル内に実装されておりアプリケーションを書き換える必要がないので、既存の音楽プレーヤー全てに適用することが可能である。このシステムについて性能を調べた結果、音楽再生プロセスがユーザの感性を考慮して制御されても、重要性の高いプロセスはほとんど遅れることなく処理された。

謝辞

本研究を支えていただいた多くの方々への感謝の意をここでは表したいと思います。

東京工業大学 千葉滋助教授には、私が千葉研究室に所属した 3 年間の間、さまざまな面でご指導いただきました。本研究においては、研究の方針や進め方、論文の書き方や研究発表に関する指導など、貴重な意見を数多くいただいきました。そして本研究におけるすばらしい論文題目を命名していただいきました。また、社会人の先輩として、人生の先輩としての貴重な意見も数多くいただき、大きな励みとなりました。深く感謝いたします。

東京工業大学 光来健一氏には、本研究を進めるにあたり最後の最後までお世話になりました。研究の進め方などの基礎的なことから、実装などの技術的なことまで数多くの助言をいただきました。深く感謝いたします。

また、貴重な助言をくださった東京工業大学 佐藤芳樹氏、西沢無我氏、中川清志氏に、そして、本研究に関して貴重な意見をくださった千葉研究室の方々に心から感謝いたします。

目 次

図 目 次

表 目 次

第1章　はじめに

従来マルチメディア処理はその時間的制約の特徴から優先すべき処理とされ、品質を損なわないように QoS(Quality of Service) 処理が開発されてきた。従来の QoS 処理はマシン上の CPU やディスクなどの計算資源を確保したり、資源が競合したときにはサンプルレートを下げたりすることで、マルチメディア処理の連続性が失われないようにすることであった。

しかし近年では、マルチメディア処理は必ずしも優先する必要のないものとして考えられる場合があり、そのようなマルチメディア処理に対する新しい QoS 処理が必要とされている。例えばバックグラウンドで行われるマルチメディア処理が挙げられる。マシン上で音楽を聴きながら他の処理をするユーザが最近では多く見られ、中にはテレビを多画面で見るようにマシン上で動画を再生しながら別の仕事をするユーザもいる。このような環境の中では、マルチメディア処理とユーザの仕事が互いに資源を奪い合い、十分な資源が得られず共に処理が遅れてしまう場合がある。バックグラウンドで行われるマルチメディア処理よりユーザの仕事の方が重要な場合には、ユーザの仕事に優先的に資源を割り当てたいのだが、資源の割り当てを急激に変更するとマルチメディア処理の QoS は損なわれる恐れがある。

また、別の例として、いろいろなマルチメディア処理の同時実行が挙げられる。マルチメディアアプリケーションは多様化し、いろいろな種類のマルチメディア処理を同時に行うようになった。ここで各マルチメディア処理には優先順位をつける必要があり、場合によっては優先順位の低いマルチメディア処理を停止させなければならない。例えば動画再生をするプレイヤーは、音楽を途切れることなく処理するために動画像の処理を停止させる場合がある。このような場合、優先順位の高いマルチメディア処理に対する QoS 処理は従来から開発されているが、優先順位の低いマルチメディア処理に対する QoS 処理は現在考えられてはいない。

我々はこれらのような重要性の低いマルチメディア処理に対して、ユーザの感性を考慮した QoS 処理を提案する。ユーザの感性を考慮するとは、ユーザがどのようなことに不快に感じるかを考え、それに対処することである。音楽や映像の途切れなどによるユーザへの不快感を抑えるように、マルチメディア処理への資源の割り当てを急激に絞らないようにする。

この QoS 処理を音楽再生処理に対し行うスケジューリングシステムとして Tanma を開発した。Tanma は progress-based regulation と呼ばれるスケ

ジューリング手法をマルチメディア処理に適応させて用いている。progress-based regulation とはプロセスの進捗状況をもとに資源の割り当てを動的に変更する手法である。プロセスの進捗を監視することで、他のプロセスとの資源の競合を探知し、プロセスを動的に制御することで資源の競合を解消する。Tanma は重要性の低い音楽再生処理の進捗状況を把握し、重要性の高い処理との資源の競合が探知された場合は音楽再生処理を停止させる。これにより音楽再生処理に使われている資源を解放することができ、重要性の高い処理を優先して処理することが可能である。また、資源不足による音の途切れなどの不具合が生じる前に音楽再生処理を停止させることによって、ユーザに不具合による不快感を与えないようにする。

この音楽再生処理の停止に伴い、Tanma はユーザに不快感を与えないための QoS 処理を行う。突然音楽の停止および再開が行われてしまうと、ユーザを驚かせてしまったり、システムに不具合があるのではないかと不安にさせたりさせてしまう。そこで、Tanma は音楽を停止させる前に音楽のフェードアウトを行い、音楽を再開させる際には音楽のフェードインを行う。これによりユーザを驚かせること無く音楽を停止させることができ、Tanma によって音楽が制御されたことをユーザに知らせることができる。さらに、フェードアウト前の部分から音楽が再開されるようにすることで (タイムマシン再生)、ユーザがフェードアウトのせいで音楽の一部を聴けないという事態を防ぐ。

我々は提案するスケジューリング手法を実現するシステム Tanma を Linux 2.4.20 上に実装した。Tanma はカーネル内のデバイスドライバに実装されており、アプリケーションを修正する必要はない。よって既存の音楽プレーヤー全てに Tanma の機能を提供することが可能である。Tanma はカーネル内のサウンドバッファを監視し、そこに保持されているサウンドデータの量を調べることで進捗状況を把握する。音楽再生プロセスの負荷が高いと、サウンドバッファにデータが書き込まれるのが遅れてしまい、データ不足により音の途切れが生じてしまう。このような状況になる前に、Tanma は音楽再生プロセスを制御する。

Tanma の性能を評価したところ、重要性の低いマルチメディア処理を制御することで、重要性の高く負荷の高い処理を遅れることなく処理することができた。また、ユーザに Tanma が提供する感性を考慮した QoS 処理に関して評価してもらったところ、多くのユーザから従来のシステムよりも良い評価を得ることができた。

本論文では以降、まず 2 章で QoS のための従来の手法についてと、重要性の低いマルチメディア処理にそれらの手法を用いる場合に生じる問題点について述べる。3 章で我々が提案する感性を考慮した QoS 処理に関して、そして、この QoS 処理を実現するスケジューリングシステム Tanma について詳しく述べる。4 章では Tanma の実装部分について述べ、5 章では Tanma に関連する実験を示し、6 章で本論文をまとめる。

第2章　QoS処理のための従来の手法

これまでマルチメディア処理には、その品質を損なわないようにするためにQoS処理が開発されてきた。QoS処理とはアプリケーションのサービスの品質を保証することである。マルチメディア処理にとってのQoSは2種類存在する。1つはマルチメディア処理の連続性を指す。音や画像が途中で途切れてしまうほど、マルチメディア処理のQoSは損なわれることになる。もう1つはマルチメディア処理の再生レートを指す。音楽に関してはサンプリングレートなどが、画像に関してはフレームレートなどが挙げられる。従来ではマルチメディア処理のこれらのQoSを保証するために、マシン上のCPUやディスクなどの計算資源を確保するようスケジューリングを行った。また、連続性のQoSを保証するために、もう一方のQoSである音楽や画像の再生レートを下げることで、必要な計算資源を減らす手法も用いられた。

しかし、マルチメディア処理にこのようなQoS処理を用いることができない場合がある。それは、マルチメディア処理と同時により優先させて実行させたい処理を行う場合である。マルチメディア処理と優先させたい処理が互いに同じ資源を必要とすると、この資源を奪い合う競合が生じてしまう。これにより互いの処理に十分な資源が割り当てられなくなり、互いの処理は遅れてしまう。優先させる処理に資源を優先的に割り当てなければならないが、このような場合には従来のQoS処理をマルチメディア処理に用いることはできない。なぜなら、従来のQoS処理はマルチメディア処理の連続性を保証するために、マルチメディア処理に資源を割り当てる処理を行うからである。このようなQoS処理をマルチメディア処理に用いると、他の優先させたい処理に必要な資源を割り当てられなくなる場合がある。しかし、従来のQoS処理をマルチメディア処理に使用しないと、資源を急激に絞られてマルチメディア処理のQoSが損なわれてしまう。

このような状況の例として、音楽を聞きながらの仕事が挙げられる。近年ではマシン上で音楽を聞く場合、他の処理は実行しないというユーザはほとんどおらず、音楽を聞きながらプログラミングを行ったりWebページを見たりするなど、他の処理と同時に実行することが多い。音楽を聞いている最中に、負荷の高い仕事を実行すると、この仕事が遅れてしまう場合がある。しかし仕事を優先させるようにすると、音の途切れなどの不具合が音楽に生じてしまう。

以降、このようなマルチメディア処理のことを「重要性の低いマルチメディア処理」と呼ぶ。その一方で、マルチメディア処理よりも優先されるべき処理のことを「重要性の高い処理」と呼ぶ。この重要性の高い処理は非マルチメディア処理とする。

本章では従来の QoS 処理で、重要性の低いマルチメディア処理の問題に対処できるかを述べる。従来からマルチメディア処理に対してはさまざまな QoS 処理が開発されている。これらの手法を用いた上で、重要性の低いマルチメディア処理と重要性の高い処理を同時に実行した場合、どのような問題が生じるかを説明する。

2.1　固定優先度スケジューリング

はじめに、従来の Linux スケジューリングについて述べる。この手法はプロセスに優先度を与え、優先度の高いプロセスを優先的に処理する。しかし、このような固定優先度を用いるスケジューリング手法では、重要性の低いマルチメディア処理の QoS は損なわれてしまう。代表的なリアルタイム Linux である RT-Linux も固定優先度を使用しており、重要性の低いマルチメディア処理の QoS 保証に用いるのは難しい。

2.1.1　CPU スケジューリング

従来のスケジューリング手法は CPU を割り当てる時間を制御するものである。Linux には CPU 優先度を変更する nice コマンドが存在し、ユーザが CPU の割り当てを変更することが可能である。また、Linux は UNIX システムのインターフェイスであるシステムコール 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。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 にはリアルタイムプロセスが入っており、他のプロセスよりも優先して処理を行うことが可能である。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。

マルチメディア処理よりも重要性の高いプロセスの優先順位を上げることで、重要性の高いプロセスを従来よりもすばやく処理することが可能である。しかし、重要性の高いプロセスを優先させればマルチメディア処理に割り当てられる CPU が一方的に減らされてしまうので、音や画像の途切れなどの不具合が生じてしまう。マルチメディア処理から重要性の高い処理に急激に資源を移しすぎると、マルチメディア処理の QoS が損なわれてしまう。重要性の高い処理が資源を急激には奪わないほどの優先度を与えられればよいが、

従来の CPU スケジューリングではそのような優先度を与えることはできない。なぜなら、システムの負荷の変化に対応することができないからである。アプリケーションは常に一定の資源を使用するものばかりではなく、資源をより多く必要としたり、あまり必要としない状態に変化したりするものが存在する。このようなアプリケーションに対し固定した優先度を与えてしまうと、資源不足に陥ったり資源を余分に受け取ったりしてしまう。資源が不足してしまうと処理時間が遅れ、資源を余分にもらうと同時に実行されているマルチメディア処理に影響を及ぼしてしまう。システムの変化に応じて適した資源の割り当てを行うことが必要であるが、従来の CPU スケジューリングではシステムの変化を探知することは考えられていない。

また、従来のスケジューリングは CPU の割り当てのみ変更を行っていることにも問題がある。アプリケーションは CPU 以外にもメモリやディスクなどの資源を使用する。重要性の高いプロセスに CPU を割り当てても、それ以外の資源の競合がする場合は重要性の高いプロセスは遅れてしまう。

2.1.2　RT-Linux

RT-Linux[1] 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ジューリングを行う。Linux カーネルは、最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。

RT-Linux はその時間的制約の厳しいスケジューリングにより、重要性の高い処理はリアルタイムタスクを用いることで遅れることなく処理される。しかし、マルチメディア処理に割り当てられる CPU は、従来の CPU スケジューリングの場合以上に急激に奪われてしまうので、音や画像の途切れなどの不具合が激しいのものとなる。また、リアルタイムタスクには必要以上の資源が割り当てられるので、マルチメディア処理以外の処理にも影響を与えてしまう。マルチメディア処理にリアルタイムタスクを用いれば、音の途切れなどの不具合は生じることはないが、これでは重要性が逆転してしまう。

RT-Linux は医療機器やロケットなどの決してミスの許されない処理をリアルタイムにおこなうことが目的である。音楽を聞きながら仕事をするなど複数のアプリケーションを同時に実行するような目的にはむいていない。RT-Linux はリアルタイム処理にのみ注目した手法なので、重要性の高い処理と低い処理の両方を考慮したスケジューリングをすることは難しい。

2.2　資源使用率に基づくスケジューリング

次に、CPU やメモリ、ディスクなどの資源の使用率を監視することで動的にスケジューリングを行う手法について述べる。資源使用率を監視することでシステムの状況の変化を探知し、状況に適するように資源の割り当てを変更することができる。これにより、重要性の低いマルチメディア処理や重要性の高いプロセスへの資源割り当てを動的に変更することが可能である。しかし、資源を監視するだけでは状況に適したスケジューリングを行うことができない場合がある。マルチメディア用リアルタイム Linux として有名な Linux-SRT と Linux/RK 、そして複数の資源を監視する手法についてそれぞれ述べる。

2.2.1　Linux-SRT

Linux-SRT[5] はリアルタイム・アプリケーションのための Linux カーネル拡張である。Linux-SRT はユーザの希望の QoS を指定させることでマルチメディア処理の QoS 保証を行う。Linux-SRT には POSIX でサポートされているスケジューリングクラスに新たなクラスを加えられている。QoS 保障され最低限の資源が割り当てられるようにするには SCHED_QOS クラスを用いればよい。スケジューリングクラスに加えて、Linux-SRT には多くのスケジューリングパラメータが用意されている。各プロセスには CPU 使用率の上限値を示すパラメタが設定されている。CPU 使用率は定期的に監視されており、プロセスが上限値以上の CPU を使用している場合、パラメタ Fallback policy に設定されているスケジューリングクラスに一定期間ランクダウンする。通常のプロセスのクラス SCHED_OTHER は新しく用意されたクラス SCHED_PAUSE や SCHED_IDLE にランクダウンし、一定時間停止させられる。一方 SCHED_QOS や SCHED_FIFO クラスのプロセスは SCHED_OTHER にランクダウンするので処理が続けられる。これにより通常のプロセスの CPU 使用を抑え、SCHED_QOS クラスのプロセスに CPU を割り当てることができる。Linux-SRT はこのようにスケジューリングクラスを変更するだけで QoS 処理を行っているので、アプリケーションを修正する必要はない。どのマルチメディアアプリケーションもサービスを受けることができる。

マルチメディア処理のスケジューリングクラスを SCHED_QOS にすることで QoS を損なうことのないように優先的に処理をすることができる。また、CPU 使用率の上限値を設定することで、マルチメディア処理が必要以上の資源を使用することのないように制限することも可能である。しかし重要性の高いプロセスに SCHED_OTHER クラスが割り当てられている場合、重要性の高いプロセスが資源を多く使用しようとすると一定時間停止させられてしまう。重要性の高いプロセスは制御されるわけにはいかないので、

SCHED_FIFO などのより優先順位の高いクラスを用いる必要がある。こうすることで重要性の高いプロセスとマルチメディア処理に資源が割り当てられるが、問題が生じる場合がある。これらの処理が共に資源を多く必要とすると場合、クラスを SCHED_OTHER にランクダウンさせても資源の競合が解消されることはない。重要性の高いプロセスは遅れ、マルチメディア処理の QoS が損なわれる恐れがある。Linux-SRT はこのような負荷の高い状況に対処することはできない [3]。

また、Linux-SRT では CPU 以外の資源の競合を探知することはできない。CPU の使用時間が短くても他の資源を多く使用する処理に対しては、CPU 以外の資源競合が生じていても CPU 時間にはその兆候がみられず、問題はないとみなしてしまう。重要性の高いプロセスがこのような処理である場合、マルチメディア処理の QoS は保証されない。

2.2.2　Linux/RK

Linux/RK[9] とは Linux / Resource Kernel を表し、リソースカーネルのアブストラクションをサポートするために Linux カーネルにリアルタイム拡張を組み込んでいる。リソースカーネルは Q-RAM[11, 12] という手法を用いており、utility とよばれる QoS を数値化したものが最大値となるように資源割り当てを行う。この手法ではまず、各アプリケーションに最低限必要な資源を割り当てる。資源の割り当て方として、CPU 使用時間とピリオドが指定されている。次に、余った資源を utility の総量が最大になるように分配する。この utility は utility function をもとに値が求められ、サンプリングレートやビットサイズなど、要求されている QoS によって異なる値となる。このように utility の総量に注目することでシステム全体の QoS の向上を行うことが、リソースカーネルの目的である。CPU が余っているかどうかは、全体の CPU 使用率を監視することで探知している。

2.3.1 で述べた Linux-SRT ではプロセスに資源利用の上限を設けたのに対し、Linux/RK はマルチメディア処理の資源利用の下限を設けている。各プロセスに最低限必要な資源を割り当て、全体的な QoS の向上が目的なので、重要性の高い処理に優先的に資源を渡すことにはむいていない。また、資源を監視することにも問題がある。この手法ではマルチメディア処理に最低限の資源が割り当てられるので、音の途切れなどの不具合が生じることはないように思える。しかし、Linux-SRT と同様に CPU にのみ注目しているので、他の資源の競合を探知できない。それゆえ CPU 以外の資源の競合が激しければ、必要な CPU が割り当てられていても、マルチメディア処理の不具合が生じてしまう。

2.2.3　各リソースを管理する手法

CPU に限らず各資源を管理するスケジューリング手法として、Performance Isolation[15] という手法が提案されている。この手法は各プロセスに最低限の CPU、メモリ、ディスクバンド幅を常に割り当てることを目的としている。Performance Isolation を実行するために、Software Performance Unit (SPU) と呼ばれるカーネル・アブストラクションを用いている。各 SPU には利用可能な資源が割り当てられており、プロセスはこれらの SPU を用いることで他のプロセスの負荷に影響されることなく、割り当てられた資源を使用することが可能である。SPU の資源管理の方法は各資源によって異なり、CPU、メモリ、ディスクバンド幅に関して特徴に合った方法がそれぞれ用いられている。また、SPU の資源使用率は定期的に監視されており、SPU に割り当てられた量以上の資源が使用されないように制御を行う。資源使用率の監視は資源を余らせている SPU を探知することにも用いられており、資源不足に陥っている SPU に一時的に資源を貸す処理を行う。資源を貸していた SPU が資源を返してほしくなった場合は資源の返還が行われるが、各資源によって何時返されるか、どのくらいの資源量が返されるかは異なる設定となっている。割り当てられる資源の量や、余らせた資源を貸すかどうかは各 SPU ごとに設定されている。ユーザは目的にあった SPU を用いることで、要求を満たすことが可能である。

Performance Isolation では CPU に限らず各資源を監視しているので、資源の競合を問題なく探知することが可能である。しかし、このシステムは複雑な設計となっているため、ユーザは利用するためには大きな負担を負わなければならない。まずは SPU システムが必要だが、Linux 用のシステムはまだ用意されていない。そして、アプリケーションにはどの SPU を使うかを指定しなければならず、マルチメディアアプリケーションごとに修正を行わなければならない。マルチメディアアプリケーションは現在数多く存在するので、この負担は大きいものとなる。

2.3　進捗に基づくスケジューリング

近年ではプロセスの作業の進み具合、すなわち進捗を調べることによってシステム状況を把握し、それをもとに動的に資源割り当てを変更する手法が注目されている。進捗を監視することで、CPU やメモリなどの資源の競合を探知することが可能である。これに伴い資源の競合を解消できるように QoS 処理を行えばよい。いくつかの研究が開発されているので、以降ではそれらについて述べる。まず、重要性の低い処理の進捗を監視し制御する手法 progress-based regulation について述べる。そして、マルチメディア処理用に開発された手法についてもいくつか述べる。

2.3.1　Progress-based regulation

progress-based regulation とは処理の進捗状況に応じて資源の割り当てを変更する手法である。重要性の低い処理の進捗を調べ、遅れていれば資源の競合が生じていると判断し、重要性の低い処理を一時停止させる。これにより重要性の低い処理との資源の競合を解消し、重要性の高い処理をすばやく行うことが可能である。2 つの処理の間で資源の競合が生じる場合、互いの仕事の進み具合い、すなわち進捗が遅れてしまう。この進捗に注目することで資源の競合が生じていることを探知することが可能である。ある資源の使用率に注目することはないので、すべての資源競合に対処することが可能である。

progress-based regulation に基づくシステム MS Manners[2] は、バックアップ処理やディスクデフラグ処理などのバックグラウンドで実行される重要性の低い処理を制御することを目的としている。MS Manners ではテストポイントと呼ばれる時間ごとに重要性の低い処理の進捗を調べる。進捗となる値は、バックアップ処理ならばアップロードしたデータ量、ディスクデフラグ処理ならば移動させたファイルブロックの数など、処理によって異なる値をとる。この進捗値とそれまでの進捗値をもとに統計的に求められる基準値を比較し、進捗状況を判断する。進捗状況が思わしくないと判断された場合、一定時間まで重要性の低い処理を停止させる。これにより重要性の高い処理に資源を渡すことが可能である。一定時間停止した後に次のテストポイントで進捗を調べ、進捗状況が思わしくない場合は前回の倍の時間だけ処理を停止させる。進捗状況が問題なければ処理を再開させ、停止時間は初期値に戻る。その時点での進捗値では進捗状況を判断できない場合は、次のテストポイントでの進捗値を調べる。このような進捗の監視と処理の制御はアプリケーションレベルで行われる。図 2.1 は MS Manners 上のアプリケーション環境を示す。重要性の低い処理は定期的に進捗値を制御システムに渡し、進捗状況が思わしくないと判断されると制御システムが重要性の低い処理を制御する。これにより重要性の高い処理に資源を十分に割り当てることが可能である。

しかし、progress-based regulation では重要性の低い処理に対する QoS 処理は全く考えられていない。progress-based regulation は重要性の低い処理に注目はしているが、あくまで重要性の高い処理のための QoS 処理を行うことが目的である。そのため重要性の低い処理がマルチメディア処理の場合、マルチメディア処理の QoS が損なわれてユーザに不快感を与えてしまう。MS Manners が対象としているバックアップやディスクデフラグはユーザから見えない処理であったので問題はなかったが、マルチメディア処理はユーザから見える処理であることが問題を生みだしている。

まず挙げられる問題は、progress-based regulation がマルチメディア処理を突然停止および再開させることである。重要性の高い処理と資源の競合が生じた場合、重要性の低い処理はすぐに停止させられ、競合が解消されれば

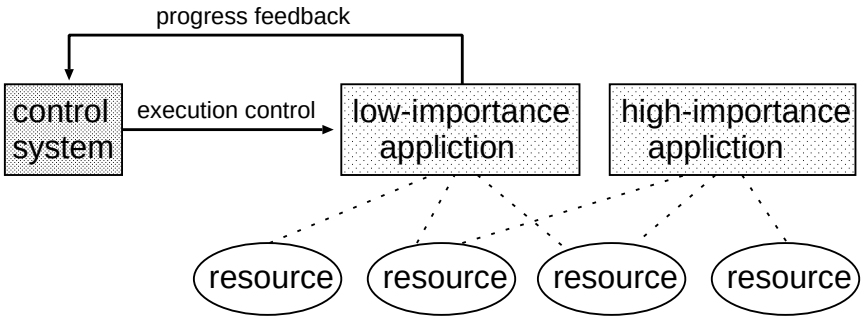


図 2.1: progress-based regulation

すぐに再開されてしまう。この手法を重要性の低いマルチメディア処理に対して利用すれば、マルチメディア処理を突然停止および再開させることになり、ユーザを驚かせて不安にさせてしまう。

次に問題なことは、progress-based regulation が行う進捗状況を調べるための一時的な処理再開である。progress-based regulation は一定時間停止した後に進捗状況を判断するまで少しの間だけ処理を再開させ進捗状況を調べるが、マルチメディア処理ではこの一時的な再開が問題である。わずかな時間だけマルチメディア処理を実行すると、ユーザにとっては短すぎて価値のない処理なのでユーザに不快感を与えてしまい、さらにその短い再生の分だけ再開する場所が先に進んでしまう。

また、従来の progress-based regulation には停止時間の設定方法にも問題がある。MS Manners では重要性の低い処理を停止させた後、進捗が思わしくない場合はまだしばらくは重要性の高い処理は終了されないと判断し、停止時間を指数関数的に増やしていく。それゆえ重要性の高い処理が終了してもしばらく停止したままとなるが、ユーザから見えない処理なので問題視されなかった。しかしマルチメディア処理の場合、重要な仕事が終了した後すぐに再開されないとユーザを苛立たせ不安にさせてしまう。

2.3.2　バッファ内のデータを監視

処理の進捗を用いる研究として、バッファ内のデータ量を監視する手法 [14, 4] が挙げられる。producer と consumer の関係である 2 つのアプリケーション間のデータ転送をすばやく行うことが目的である。2 つのアプリケーションの仲介を担うバッファ内のデータ量を監視することで、2 つのアプリケーションの進捗を調べる。producer アプリケーションはこのバッファにデータを転送し、consumer アプリケーションはバッファ内のデータを取得する。バッファ内のデータ量が一定値を保つ場合、2 つのアプリケーションには適した資源割り当てが行われているといえる。しかしバッファが満杯の状態が続く場合や

空になる場合は、2 つのアプリケーションへの資源の割り当て方が適していないことがわかる。この場合、資源割り当てを変更するために、reservationo based scheduling と呼ばれるスケジューリング手法を用いる。この手法では proportion（CPU 使用率）と period（CPU 使用期間）を指定することで、資源割り当てを行う。2 つのアプリケーションにバランスよく資源を割り当てることで、一連のデータ転送処理をすばやく行うことができる。

この手法をマルチメディア処理に適合させている QoS adaptation and real-rate scheduling という手法が開発されている [4]。マルチメディアデータが複数のプロセスを経由する場合に、一連のデータ転送が時間どおり行われるように QoS 処理を行う。この手法ではメディアデータの各パケットにタイムスタンプとプライオリティを付加し、これらを利用する。進捗状況の監視と判定はバッファ内のデータ量を調べるのではなく、バッファ内のデータのタイムスタンプを調べることで行われる。古いデータがいつまでも残っているか、あるいは新しいデータしか保持されていないような場合は、各プロセスの資源割り当てのバランスが悪く、進捗が思わしくないことが探知できる。これに対し proportion と period をプロセスに指定することで資源の割り当てを適したものにする。また、資源の割り当てが間に合わなくて QoS が保証できない場合は、バッファ内にあるプライオリティの低いパケットを破棄することにより QoS を保証する。

これらの手法はマルチメディア処理で使われている複数のプロセスへの資源割り当てを対象としており、今回議論している重要性の低いマルチメディア処理と重要性の高い処理を同時に実行するという状況とは少し異なる。しかし、アプリケーションが書き込みを行うカーネル内のサウンドバッファのデータを調べることで、重要性の低いマルチメディア処理の進捗状況を探知することは可能である。データ量を監視するのであれば、アプリケーションレベルの実装もカーネルレベルの実装も可能である。タイムスタンプやプライオリティを付加するにはアプリケーションレベルの実装を行わないと実現させるのは難しい。

2.3.3　レイテンシの監視

進捗情報を利用する手法として、処理の遅延を監視する方法 [8] がある。この手法は主にクライアントからサーバへのリクエストの際に用いられる。クライアント側にとってサーバのレイテンシを考えることが処理を行う上で重要なことである。これに対し、サーバプロセスにクライアントの希望に応じた資源割り当てを行う feedback control loop というシステムが提案されている。このシステムでは absolute guarantee model と proportional differentiated service model というレイテンシに対する 2 つの手法をハイブリッドして用いている。absolute guarantee model とは各クラスにソフトデッドラインを設

定し、レイテンシがこのデッドラインを越えることのないように資源を割り当てる手法である。この手法には delay を制限することができるという利点があるが、システム状態に応じてデッドラインの値を計算することが難しいという欠点がある。proportional differentiated service model とはレイテンシの比率を保つように資源を割り当てる手法である。この手法を用いることで、重要な処理と重要でない処理とで資源の割り当て方を区別することができるが、システムの負荷が高いと重要な処理に資源が割り当てられない場合がある。feedback control loop は、システムの負荷がそれほど高くない場合は proportional differentiated service model を用いて資源割り当てを行い、システムの負荷が高い場合は absolute guarantee model を用いて重要な処理に過度の遅れが生じないようにしている。feedback control loop はレイテンシの理想値と実際のレイテンシを比較し、理想値と誤差があるかどうかを調べる。レイテンシの理想値はクライアントによって設定される。

このシステムのは proportional differentiated service model により、複数のサービスについて資源の割り当て方を区別することができる。重要性の高いものと低いものの 2 つに限らず、いろいろな処理をバランス良く資源割り当てすることが可能である。また、absolute guarantee model により、最低限の資源が割り当てられることができるのもこのシステムの特徴である。これらのことを考えると、このシステムは 2.2.2 節で述べた Linux/RK と同様に、全体的な QoS の向上を目的としたシステムといえる。よって、重要性の高い処理を優先的に行うことには向いていない。マルチメディア処理に最低限必要な資源が割り当てられるので、少なくともサーバ・クライアント間の遅延により音や画像の不具合が生じる恐れはない。しかし、このシステムを提供させるには、クライアントとサーバ共に実装を行う必要がある。

2.3.4　再生レートの変更

進捗状況に基づくスケジューリング例として、パケット速度を監視する方法 [6, 7] があげられる。この手法は主にサーバ・クライアントでのマルチメディア転送で用いられる。CPU やネットワークバンド幅が不足すると、サーバから転送されたデータがクライアントに渡らずパケットロスしてしまう場合がある。このような状態であることをパケット速度を監視することで知ることができる。このような状態に対処する手法として、パケットドロップと呼ばれる手法が良く使用されている。この手法は、動画や音声の品質を下げることで転送するパケット数を減らし、割り当てられている資源に見合ったデータ転送を行う手法である。これにより、マルチメディア処理の連続性という意味での QoS を保証し、資源を節約することが可能である。パケットドロップの例である priority packet-dropping[6] では、動画像のパケットにプライオリティを付加させることで、転送すべきパケットや必ずしも転送する必

	重要性の高い処理の QoS（処理速度）	重要性の低いマルチメディア処理の QoS（連続性と再生レート）
CPU スケジューリング	△	×
RTLinux	○	×
MS Manners	○	×
Linux-SRT	×	○
Linux-RK	×	○
Performance Isolation	×	○
QoS adaptation and real-rate scheduling	×	○
feedback control loop	×	○
priority packet-dropping	△	△

表 2.1: 従来の手法を用いた場合の処理の QoS

要のないパケットを設定している。転送するパケット数を変更するには、プライオリティの閾値を変更することで行う。進捗情報はサウンドデータを取得するクライアント側からサウンドデータを転送するサーバ側に伝えられる。これに伴いサーバ側はサウンドデータを転送する前に、パケットドロップをサウンドデータに実行し、サウンドデータを転送する。このサウンドデータをクライアント側は取得し、パケットロスが生じているかを調べる。

再生レートを変更することで与えられた資源だけを用いて、マルチメディア処理が途切れることなく続けることができる。また、マルチメディア処理に渡る資源を節約することができるので、これまでに述べたマルチメディア処理用の手法よりも、重要性の高い処理に資源を渡すことが可能である。

この手法はサーバ・クライアント間でのパケットロスに注目しているが、ローカル内でのマルチメディア処理ではパケットロスはほとんど生じることはない。よって、今回問題としている、同じマシン上でマルチメディア処理と重要性の高い処理を実行するという状況では進捗状況を調べることはできない。また、この手法を利用するには、クライアントとサーバ共に実装を行わなければならない。

2.4　まとめ

ここまで資源の競合を探知する方法に注目をおきながら、従来の手法を順に述べてきた。ここで、それぞれの手法の目的を満たすように、重要性の高い処理と重要性の低いマルチメディア処理を同時に実行した場合の QoS の評

価を表 2.1 に示す。重要性の高い処理の QoS とは処理の速度を指す。この表を見てみると、ほとんどの処理がどちらか一方の QoS しか満たすことができていないことがわかる。

CPU スケジューリングや RTLinux は優先度を与えるという点から、重要性の高い処理を優先させることが目的であると言える。これらの手法を用いることで重要性の高い処理をよりすばやく処理することが可能だが、マルチメディア処理に対する QoS は考えられていなく、途切れなどの不具合が生じてしまう。また、進捗を元に資源割り当てを変更する MS Manners も重要性の高い処理を優先させるために、重要性の低いマルチメディア処理を停止させてしまう。途切れなどの不具合が生じる前に停止させられるが、この停止がユーザにとっては不快なものとなってしまう。

マルチメディア処理のための手法は、マルチメディア処理が途切れること無く実行されることが目的である。これらの手法を用いることで、マルチメディア処理に資源を十分に渡し、マルチメディア処理の QoS を高く保つことが可能である。しかし、重要性の高い処理に資源があまり割り当てられなくなることで処理の遅れが生じてしまい、重要性の高い処理の QoS が損なわれてしまう。

唯一どちらの処理の QoS にも配慮した手法が、マルチメディア処理の再生レートを調整する手法である。マルチメディア処理に必要な資源を減らすことで、重要性の高い処理にできるだけ資源を渡すことが可能である。しかし、重要性の高い処理が多くの資源を必要とする場合には、再生レートを下げるだけでは資源を完全に渡すことはできない。

従来の手法に対して、より重要性の高い処理をすばやく処理する手法が必要とされている。重要性の高い処理に多くの資源を割り当てなければならない場合、マルチメディア処理の連続性を保証することはできない。よって、できるだけマルチメディア処理の連続性を損ねることのないような手法が必要となる。ここで問題となるのは、連続性という意味でのマルチメディア処理の QoS は保証がされない場合は、それ以降どのように評価されるかということである。例えば、途切れが生じる場合と MS Manners のように停止させる場合とでは、どちらの QoS が高いといえるか判断しなければならない。しかし、従来ではマルチメディア処理のこのような場合の QoS の評価はなされなかった。なぜなら、従来の手法ではマルチメディア処理の連続性を保証し、音や画像の途切れなどの不具合が生じないようにすることが目的であった。重要性の低いマルチメディア処理に対し、新たな QoS の指標が必要である。

第3章　Tanma

この章では我々が提案するシステム Tanma について述べる。これまで、重要性の低いマルチメディア処理に対する QoS 処理は見落とされていた。我々は、このようなマルチメディア処理に対しユーザの感性を考慮した QoS 処理を考え、Linux の音楽再生に対してこの QoS 処理を提供する本システムを開発した。ここでは Tanma が提供する QoS 処理とそのための機能について説明する。

3.1　新たに必要：感性を考慮した QoS 処理

重要性の高い処理をすばやく処理するために、重要性の低いマルチメディア処理に対して従来の QoS が保証されなくなる場合がある。このようなマルチメディア処理に対する QoS 処理とは、ユーザの感性を考慮すること、すなわち、ユーザに不快感を与えないようにすることである。

3.1.1　感性を考慮した QoS 処理とは

重要性の低いマルチメディア処理に対し、連続性を保証できるほどの資源が割り当てられない場合、どのようにして連続性が損なわれるのをできるだけ抑えるかが問題となる。しかし、従来の QoS 処理では連続性を保証することにのみ注目を落ちていたので、連続性が保証されない場合に対処する方法は考えられていなかった。このような特殊な場合の連続性の QoS は、数値で評価することは難しい。そこで、ユーザがどのように感じるか、ユーザの感性によってこの QoS を評価することが考えられる。

ここで本文で使われる「感性」について定義しておく。広辞苑には以下のように記されている。本文では「感性」を、「ユーザが不快に感じる感受性」の意味で用いることにする。

> かん‐せい【感性】
>
> 1. 外界の刺激に応じて感覚・知覚を生ずる感覚器官の感受性。「―豊か」
> 2. 感覚によってよび起され、それに支配される体験内容。従って、感覚に伴う感情や衝動・欲望をも含む。

3. 理性・意志によって制御さるべき感覚的欲望。

4. 思惟の素材となる感覚的認識。

このような感性を考慮するとは、ユーザがどのようなことに対し不快に感じるかに注目するということである。ユーザの感性を考慮し、そのための QoS 処理を行うことで、重要性の低いマルチメディア処理に対してユーザが許容できるほどの QoS を保証することが可能である。

3.1.2　感性を考慮した QoS 処理の例

ユーザの感性を考慮した QoS 処理とは、どのようなものが考えられるかを示さなければならない。ここではその例について述べる。

まず考えられる QoS 処理は、重要性の低いマルチメディア処理がユーザに不快感を与えそうになった場合、瞬時に止めることである。これによりマルチメディア処理がユーザに与える不快感を最小限に抑えることができる。上で挙げた BGM の例の場合、BGM 処理に必要な資源が不足して音が途切れてしまう問題がある。一瞬であれば問題はないが、重要性の高いプロセスが競合する資源を長時間、大量に使用する場合は途切れる度合いが大きくなってしまう。頻繁な途切れが生じる音楽はユーザにとっては聞き苦しいものなので、このような場合は音楽を止める方がよい。そうすることで資源の解放にもなり、重要性の高いユーザの仕事を速く終わらせることができる。

次に、マルチメディア処理を停止および再開させる際に、ユーザに不快感を与えないようにする必要がある。突然停止や再開をさせるとユーザを驚かせてしまい、また、システムに不具合があるのではないかと不安にさせてしまう。マルチメディア処理をユーザの指示によらずに停止および再開させる際には、その前にそのことをユーザに知らせる QoS 処理が必要である。この QoS 処理にはメディアごとに様々な形が存在する。音楽を停止させる場合は、その前後で音楽をフェードアウト/インさせることにより、ユーザに違和感なく停止および再開させることができる。動画の場合は、動画像を暗転させたりモザイクをかけていくなどのフェードアウト/インを行ったり、停止中に「しばらくお待ちください」と表示させるなど、音楽よりも様々な QoS 処理が可能である。複数のウインドウを表示するアプリケーションでは、使用していないウインドウを徐々に小さくして画像処理を減らすことが考えられる。

これらの QoS 処理が、実際にユーザの感性を考慮したものといえるかどうか、ユーザアンケートを行った。音楽再生に関しての感性を考慮した QoS 処理について、ユーザの意見や感想を述べてもらった。その結果は 5.5 節で述べる。

3.2　Tanma の特徴

我々は重要性の低い音楽再生処理に対し QoS 処理を行うスケジューリングシステム Tanma を開発した。Tanma はマルチメディア処理の中でも基本的なものである音楽に特化したシステムである。Tanma は資源が競合したときに BGM のような重要性の低い処理によるユーザへの不快感を抑えるために、ユーザの感性を考慮した QoS 処理を音楽プレーヤーに施して制御する。Tanma は progress-based regulation により重要性の低い音楽プレーヤーを進捗状況に応じて停止させることで、音の途切れを未然に防ぐことが可能である。この停止および再開によるユーザへの不快感を抑えるために、音量の自動調整とタイムマシン再生を行う。音量の自動調整とは音楽を停止させる際にフェードアウトを行い、音楽を再開させる際にフェードインを行う処理である。この処理を行うことで、音楽の停止および再開が与えるユーザへの不快感を和らげることが可能である。また、タイムマシン再生とはフェードアウトなどにより聞こえなくなってしまった音楽を改めて再生する処理である。これらの機能を使うことでユーザに不快感を与えることなく音楽を停止および再開させることができる。

Tanma による音楽再生処理の制御は以下のような流れで行われる。

1. 音楽再生処理の進捗を調べて資源の競合を探知。
2. 音量の自動調整機能によりフェードアウトしながら音楽を停止。
3. 停止後は音量の自動調整機能によりフェードインしながら音楽を再開。この際、タイムマシン再生機能を実行。

まず Tanma は、音楽再生処理の進捗を調べて遅れが生じているかどうかを判断する。遅れが生じている場合は重要性の高いプロセスと資源の競合を起こしていると判断し、音量の自動調整機能により音量を徐々に下げる。遅れが生じていなければ、そのまま音楽を続行する。音量の自動調整機能により音量が 0 になれば一定時間経過するまで音楽再生処理を停止させる。一定時間停止した後、まだ遅れが生じる場合は停止を続け、問題ない場合は音量の自動調整機能により音量を徐々に上げながら音楽を再開させる。この際、タイムマシン再生機能によりフェードアウトを始めた場所から音楽を再開させる。この機能を使うことで、音量の自動調整機能によって聞こえにくくなってしまった部分を改めて再生しなおすことが可能である。

Tanma のスケジューリングは MS Manners のようにアプリケーションから進捗情報を得るのではなく、カーネル側からアプリケーションの進捗状況を監視することで行われる。これにより、音楽アプリケーションを修正する必要はない。全ての音楽アプリケーションに Tanma は対処することが可能である。

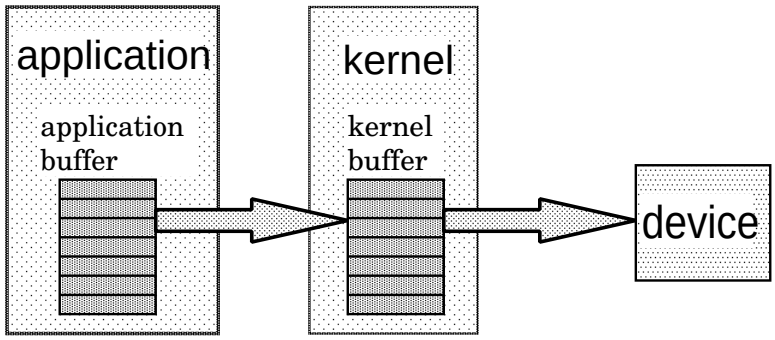


図 3.1: サウンドデータの転送

「Tanma」というシステム名の由来は、日本語の「たんま」からきている。子供が遊びを一時中断させる際にタイムの意味で用いる言葉である。本システムは音楽などのマルチメディア処理を一時的に中断させることから、「Tanma」と名付けられた。

3.3　Tanma が提供する各機能

ここでは Tanma が提供する各機能について述べる。Tanma にはスケジューリング機能、音量の自動調整機能、タイムマシン再生機能があり、これらが働くことによりユーザの感性を満たす QoS を保証することが可能である。

3.3.1　スケジューリング

progress-based regulation にもとづき、音楽再生プロセスの進捗状況に応じて資源の割り当てを変更する。音楽が途切れそうになれば音楽再生処理を止めることにより、不具合をできるだけユーザにみせることなく、重要性の高い処理に資源を割り当てることができる。

進捗状況の判断には再生データを格納するカーネルバッファを調べることで行う。音楽の再生は図 3.1 のようにアプリケーションからカーネル、カーネルからサウンドデバイス、とサウンドデータが転送されることによって行われる [17]。音楽再生処理によってアプリケーションからこのカーネルバッファにデータが転送されることで、カーネルバッファ内のデータは増えていく。これに対しカーネルバッファ内のデータはサウンドデバイスに DMA 転送され、バッファ内のデータは徐々に減っていく。音楽再生処理に資源があまり割り当てられなくなると、アプリケーションからカーネルへのデータ転送が遅れてしまう。これによりカーネルバッファ内のデータが減っていき、場合によっては DMA 転送するデータがなくなり音が途切れてしまう。

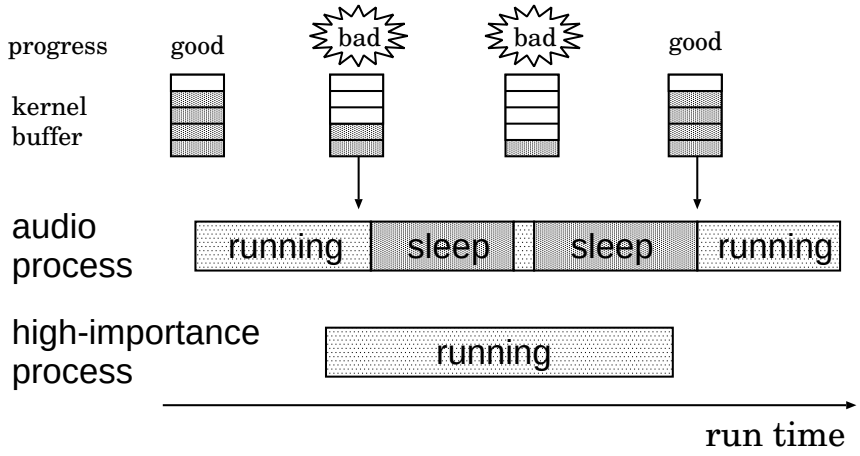


図 3.2: Tanma による進捗に基づくスケジューリング

Tanma のスケジューラはカーネルバッファ内に溜まっている再生データのサイズを定期的に調べ、データサイズが閾値を下回ると進捗が遅れていると判断する。音楽再生プロセスと同時に資源を多く消費する重要性の高い処理が起動すると、音楽再生処理が遅れてしまいバッファ内のデータサイズが閾値よりも小さくなるからである。この場合は資源の競合が生じていると判断し、音楽再生処理を一定時間停止させる。一定時間停止させた後、Tanma は現在の進捗状況を調べるために一時的に音楽再生処理を再開させる。再び進捗状況を調べ、音楽再生処理の資源がまだ競合していれば、閾値よりバッファ内のデータサイズが小さいままとなるので停止を続ける。音楽再生プロセスの資源の競合が解消すればバッファ内のデータサイズは閾値より大きくなるので、この場合は音楽を再開させる。Tanma のスケジューリングの様子を図 3.2 に示す。

停止時間はユーザの感性と処理性能のバランスを考え 3 秒に設定する。MS Manners では停止時間を指数関数的に増加させていたが、これでは重要性の高い処理の終了後、音楽が再開されるまでユーザは長い時間待たされてしまい、不快感を与えてしまう。Tanma が設定した 3 秒の停止であれば、ユーザは不快に思うことはなく、停止時間が短いことによるオーバヘッドもそれほどない。この停止時間の設定に関しては 5.4 節で述べる。

3.3.2　音量の自動調整

音量の自動調整機能は、停止させる前後で音量を調整することで、ユーザに与える不快感を取り除く機能である。progress-based regulation により音楽再生処理を突然停止や再開させると、ユーザを驚かせて不安にさせてしまう。

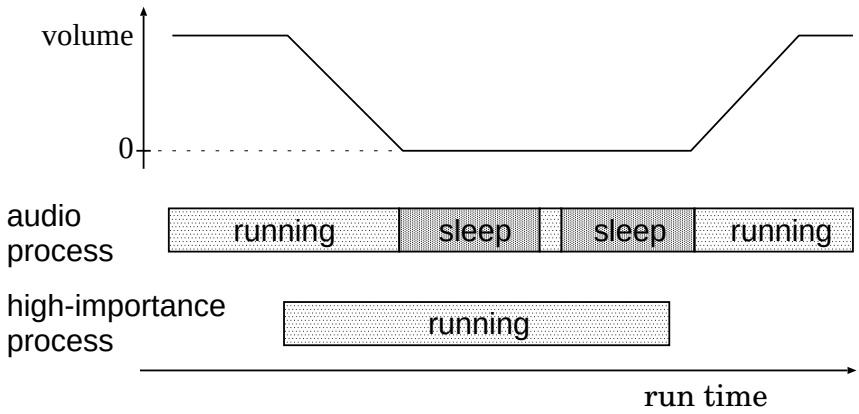


図 3.3: スケジューリングに伴う音量の自動調整

そこで、音楽を停止させる必要がある場合にこの音量の自動調整機能が働くことにより、ユーザに停止および再開させることを自然に知らせることができる。停止する前にフェードアウトしながら音楽を再生し、停止した後に再開する時はフェードインしながら音楽を再生する。フェードアウト/インは一定の周期で音量を徐々に変化させることで行われる。フェードアウトは音量が 0 になるまで、フェードインは音量が元の値に戻るまで行われる。フェードアウトは進捗が思わしくない場合に行われ、フェードインは進捗が問題ない場合に行われるが、進捗状況の変化に応じてフェードアウトからフェードイン、フェードインからフェードアウトに切り替えることが可能である。

また、音量の自動調整機能により progress-based regulation の問題点であった一定時間停止した後の短い再生に対処することができる。progress-based regulation では一定時間停止し終わった後に進捗状況を調べるために短い時間だけ音楽再生処理を再開しなければならず、この音楽再生がユーザに不快感を与えてしまう。音量の自動調整機能は、この短い時間の再生中の音量を 0 にしてユーザには聞こえないようにする。これによりユーザへの不快感を抑えることが可能である。

音量の自動調整はスケジューリングと連動して行われる。図 3.3 は音量の自動調整機能の処理の流れを示す。重要性の高い処理との資源の競合が生じ、音楽再生処理の進捗が思わしくない場合、音楽再生処理に対してフェードアウトが実行される。フェードアウトにより音量が 0 になった時点で、音楽再生処理は一定時間停止させられる。その後は音楽再生処理の進捗が問題なくなるまで音量は 0 のままとなる。よって、一定時間停止し終わった後に一時的な音楽再生処理はユーザに聞こえることはない。進捗が問題なくなれば、フェードインを実行しながら音楽再生処理を再開させる。

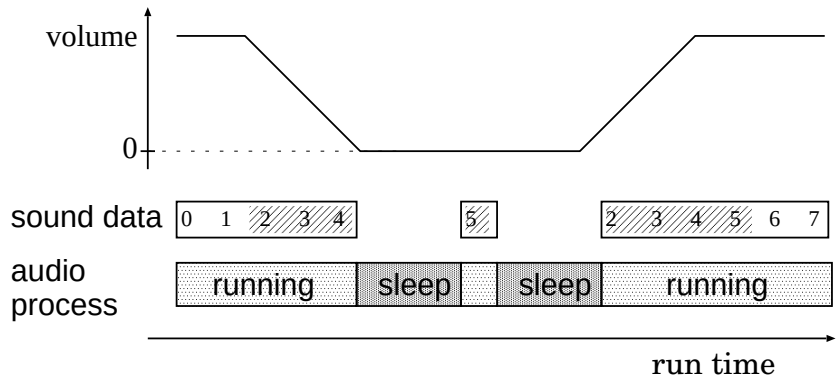


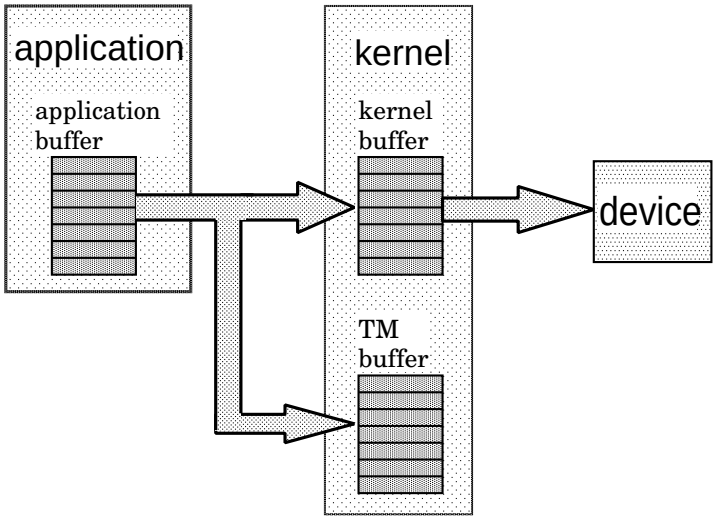
図 3.4: タイムマシン再生の処理の流れ

3.3.3　タイムマシン再生

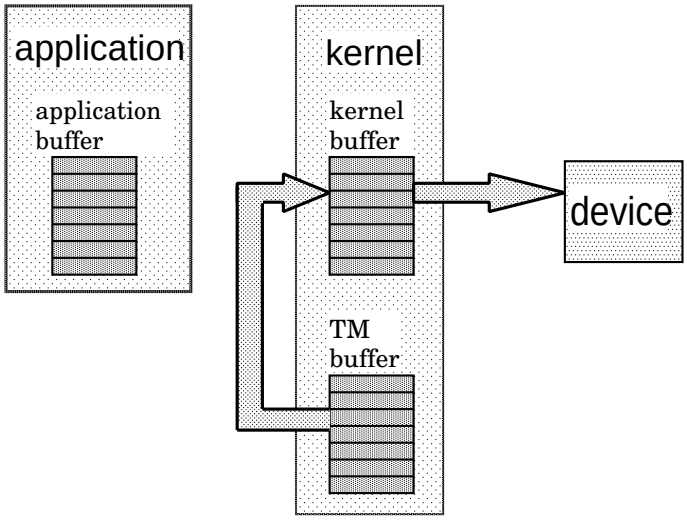
タイムマシン再生機能は音楽の再開時に、フェードアウト前の部分から音楽を再生させる機能である。フェードアウトを行うことにより音量が小さくてユーザには聞こえにくい部分ができてしまう。この部分をユーザがきちんと聞くことができるように、音楽が再開される際にその部分から改めて再生を行う。また、タイムマシン再生は、ユーザがどの部分で音楽が聞こえなくなってしまったかを思い出させるという働きもする。停止時間が長い場合、ユーザはどこまで聞いていたのかを思い出せなくなってしまう。タイムマシン再生が実行されることにより、聞いていたときのフィーリングを思い出すことが可能である。

タイムマシン再生機能がスケジューリング機能や音量の自動調整機能にともない、どのように実行されていくかを図 3.4 に示す。進捗が思わしくない場合に音楽のフェードアウトを行うが、このフェードアウト中に再生しているサウンドデータをカーネル内のタイムマシン再生用のバッファに保存する。一定時間停止し終わった後の一時的な音楽再生処理で再生されたサウンドデータも保存される。停止後に進捗状況が問題なくなれば、これまで保存されたサウンドデータは再生される。この再生を実行中は、アプリケーションからカーネルへのサウンドデータの転送は行われない。保存していたデータが全て再生された後で、元の位置から続けて音楽を再生する。

タイムマシン再生機能は図 3.5 のようなバッファリングによって処理を行う。再生中のサウンドデータを後でタイムマシン再生する必要がある場合、図 3.5（a）で示すようにタイムマシン再生用のバッファ（TM バッファ）にサウンドデータを保存する。タイムマシン再生用のバッファにはカーネルバッファに転送されるものと同じサウンドデータが転送される。タイムマシン再生を行うには、図 3.5（b）で示すようにタイムマシン再生用のバッファに保持されたサウンドデータをカーネルバッファに転送する。これにより以前再



(a) サウンドデータをタイムマシン再生用バッファに保存



(b) タイムマシン再生の実行

図 3.5: タイムマシン再生におけるバッファリング

生されたサウンドデータを改めて再生することができる。このとき、アプリケーションバッファからカーネルバッファへのサウンドデータの転送は、終了するまで待たされることになる。

第4章　開発したTanmaの実装

4.1　Linuxのサウンドメカニズム

以上述べてきた音楽再生処理にQoS処理を施し制御するスケジューリングシステムTanmaをLinux 2.4.20に実装した。サウンドカードEnsoniq AudioPCI (ES1371) のデバイスドライバes1371.cに実装を施し、音楽の再生処理を行う関数es1371_write() 内でTanmaの主な処理を実行する。

ここで今回実装対象となったデバイスドライバes1371.cの処理について説明する。es1371.cにはサウンドアプリケーションが呼んだシステムコールに対応する関数がそれぞれ用意されており、アプリケーションの命令を実行する。ここでは音楽の再生処理を行う関数es1371_write() について説明する。es1371_write() はアプリケーション側からのwriteシステムコールによって呼ばれ、アプリケーションバッファからカーネルバッファへサウンドデータの書き込みを行う。es1371_write() は以下のような流れで処理を行う。

1. 指定されたデータサイズの書き込みが行われるまで2から3を繰り返す。
2. カーネルバッファが満杯ならばschedule() を呼び、カーネルバッファに空きができるまでCPUを放棄。
3. アプリケーションバッファからカーネルバッファにサウンドデータを書き込み、1に戻る。
4. カーネルバッファに書き込まれたデータサイズをアプリケーションに返す。

es1371_write() はアプリケーションバッファから指定されたサイズのデータをカーネルバッファに書き込みをする関数である。es1371_write() はまず、カーネルバッファにデータサイズを書き込める空きがあるかどうかを調べる。カーネルバッファに空きが無い場合、サウンドデバイスが新しいサウンドデータを処理する準備ができるまで待つために、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サウンドデバイスからの割り込みが生じるまで休止状態にはいる。カーネルバッファ内のデータがデバイスに転送されてバッファに空きができれば、空きがある限りアプリケーションバッファからカーネルバッファへバッファにデータ

の書き込みを行う。このようにカーネルバッファの状態を調べ、アプリケーションが指定したサイズまでアプリケーションバッファからカーネルバッファへのサウンドデータの書き込みを繰り返す。

サウンドバッファ内で保持されるサウンドデータが極度に減ってしまう原因は主に 2 つ考えられる。1 つはアプリケーションが write システムコールを呼ぶのが遅れてしまう場合である。多くの音楽アプリケーションは音楽を再生する際に、書き込むサイズを小さくして write システムコールを複数回呼ぶ。アプリケーションが資源の競合により動作が遅れている場合、write システムコールを呼ぶインターバルが長くなってしまう。これによりサウンドデータの書き込みが遅れてしまい、サウンドバッファ内のサウンドデータが極度に減ってしまう。もう 1 つの原因は、schedule() 関数が呼ばれた後にプロセスが復帰するのに時間がかかってしまう場合である。音楽再生プロセス以外のプロセスに CPU 時間が多く割り当てられている場合、音楽再生プロセスに CPU が割り当てられるまでに長時間たってしまう。その結果、サウンドバッファ内のサウンドデータが極度に減ってしまう。

4.2　Tanma の実装概要

Tanma が提供するスケジューリング機能、音量の自動調整機能、タイムマシン再生機能は、デバイスドライバ es1371.c の音楽の再生処理を行う関数 es1371_write() 内に組み込まれている。サウンドドライバの音楽再生部分に実装を施しているので、全ての音楽再生プロセスの進捗状況が Tanma によって監視される。音楽再生プロセスは基本的には重要性の低いプロセスとして扱われ、他のプロセスと資源を競合して処理に遅延が生じれば Tanma により制御される。音楽再生プロセスの重要性を高く設定し、Tanma によって制御されないようにするには、nice を使い音楽再生プロセスの優先度を上げればよい。

デバイスドライバ es1371.c に Tanma を適用するために、以下のように Tanma の処理を組み込む。

1. 指定されたデータサイズの書き込みが行われるまで 2 から 3 を繰り返す。
2. カーネルバッファが満杯ならば schedule() を呼び、カーネルバッファに空きができるまで CPU を放棄。

☆ i. 進捗状況を調べる必要がある場合は実行し、それに応じてこのプロセスの制御や音量変更を行う。

☆ ii. タイムマシン再生するサウンドデータがあれば再生し、1 に戻る。

☆ iii. あとでタイムマシン再生が必要な場合、タイムマシン再生用バッファに書き込みを行う。

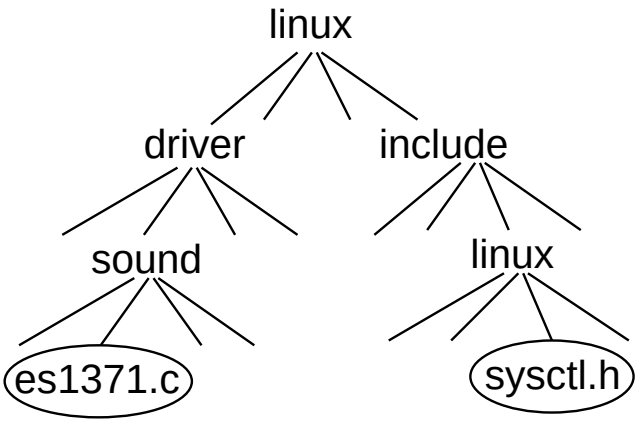


図 4.1: Tanma が実装される Linux ソース

3. アプリケーションバッファからカーネルバッファにサウンドデータを書き込み、1 に戻る。

4. カーネルバッファに書き込まれたデータサイズをアプリケーションに返す。

Tanma の処理はアプリケーションバッファからカーネルバッファへのデータ書き込みの前に行われる。まず現在の進捗状況を調べ、それに応じてプロセスを一定時間停止させたり音量の変更を行う。これらの処理は毎回呼ばれるのではなく、定期的に時間をあけて実行される。次に、現在タイムマシン再生をする必要がある状態ならば実行し、カーネルバッファの空きを調べる状態に戻る。タイムマシン再生が必要で無い場合は、サウンドデータの書き込みに移る。この際に、現在書き込み中のデータがあとでタイムマシン再生する必要がある場合、タイムマシン再生用のバッファにも書き込みを行う。

Tanma の実装はデバイスドライバ es1371.c と、sysctl システムコール用のヘッダ sysctl.h にのみ行われている。sysctl.h には Tanma のパラメータが登録されており、sysctl システムコールにより変更が可能である。ユーザはこれらのフェイルを適応させるだけで、Tanma を利用することが可能である。スケジューラ関数やファイルシステムなど、複雑なファイルを修正する必要はない（図 4.1）。

以下では Tanma の処理についての実装詳細を述べる。Tanma の機能であるスケジューリングと音量の自動調整は関連性の強いものなので、同じセクションで説明する。タイムマシン再生機能に関してはその後のセクションで述べる。これらの基本的な機能以外の細かい実装に関しても説明する。

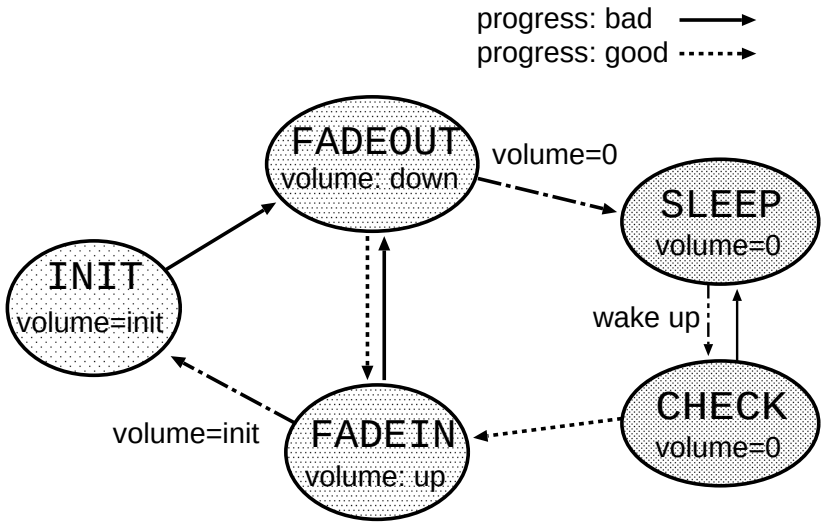


図 4.2: Tanma における音楽再生プロセスの状態遷移

4.3　スケジューリングと音量自動調整

4.3.1　実装の流れ

tanma_progress() を呼ぶことで Tanma が提供するスケジューリングと音量の自動調整を行う。tanma_progress() はカーネルバッファ内に保持されているサウンドデータのサイズを調べ、進捗状況を判断する。その後、進捗状況の変化に応じてプロセスの状態遷移を行い、プロセスに状態に応じた処理を行う。

Tanma による音楽再生プロセスの制御は図 4.2 で示す状態遷移によって行われる。プロセスの状態は 5 種類用意されている。

- INIT: 初期状態。
- FADEOUT: フェードアウト状態。音量が下げられる。
- FADEIN: フェードイン状態。音量が上げられる。
- SLEEP: スリープ状態。一定時間停止させられる。
- CHECK: スリープ後の状態。音量を 0 にして進捗状況を調べる。

通常、プロセスは進捗状況に問題がなければ INIT 状態にある。進捗が思わしくない場合は INIT 状態から FADEOUT 状態に移る。FADEOUT 状態では徐々に音量を下げていき、音量が 0 になった時点で SLEEP 状態に移り、一定時間停止する。一定時間停止した後は CHECK 状態に移り、進捗が思わしく

なければ SLEEP 状態に移り停止を続け、進捗が問題なければ FADEIN 状態に移る。FADEIN 状態では徐々に音量を上げていき、音量が元の値に戻った時点で INIT 状態に移る。FADEOUT、FADEIN 状態でも進捗状況を監視しているので FADEOUT 状態から FADEIN 状態、FADEIN 状態から FADEOUT 状態のようにプロセスの状態が遷移する場合もある。

tanma_progress() はこのようなプロセスの状態遷移を行い、その後にプロセスの状態に応じた関数を呼ぶ。FADEIN 状態の場合は tanma_fadein() が、FADEOUT 状態の場合は tanma_fadeout() が用意されており、音量の自動調整を行う。SLEEP 状態の場合は tanma_fadesleep() が呼ばれ、プロセスを一定時間停止させる処理を行う。tanma_progress() を es1371_write() に組み込んだ実装の概要図を図 4.3 に示す

4.3.2　tanma_progress

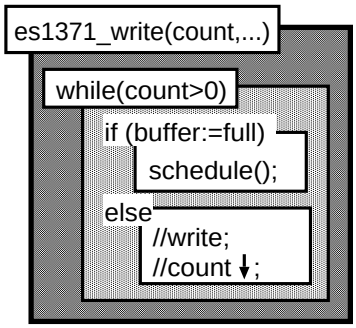
tanma_progress() は音楽再生プロセスの進捗状況を判定し、プロセスの状態に応じた関数を呼び、Tanma が提供するスケジューリングと音量の自動調整を行う 。この関数は es1371_write() 内で呼ばれ、サウンドデータの書き込みが行われる前に実行される。tanma_progress() 内の処理は定期的に行われるように設定されている。前回処理を行った時刻 prev_time と現在の時刻 now_time を比較し、前回の処理から設定されたインターバルの値まで時間が経過しているかどうかを調べる。インターバルの値は MS Manners の制御手法と同様に、0.5 秒に設定されている。現在の時刻は新しく用意した関数 return_time() を呼ぶことで取得できる。return_time() は現在の時刻をマイクロ秒単位で返す。

tanma_progress() は現在の進捗値と進捗に問題がない場合の進捗値を比較し、進捗状況を判断する。現在の進捗値 cur_prog はカーネルバッファ内に保持されているサウンドデータのサイズとする。進捗に問題がない場合の進捗値 avr_prog はそれまでの進捗値の平均値とし、以下の漸化式で求める。

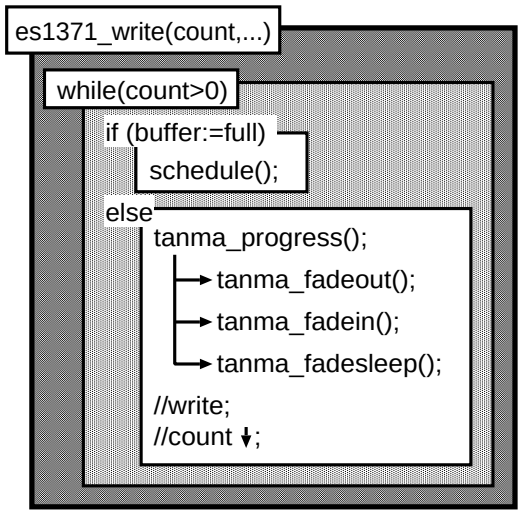
$$avr\_prog = avr\_prog \times 0.999 + cur\_prog \times 0.001;$$

進捗状況は cur_prog の値が avr_prog の値の 0.75 倍以下となった場合に、遅れていると判断される。avr_prog は進捗に問題がない場合の進捗値としているので、cur_prog の値が進捗が遅れている場合の値であれば、avr_prog は上記の式を実行する前の値に戻される。

その後は図 4.2 に従い、プロセスの状態遷移を行う。INIT 状態から FADEOUT 状態に移る場合は、このときの音量の値を取得しておく。この値は進捗状況が問題なくなり、音量を元の値に戻すために用いられる。状態遷移が行われた後、それぞれの状態のための関数が呼ばれる。FADEIN 状態の場合は tanma_fadein() を、FADEOUT 状態の場合は tanma_fadeout() を、SLEEP 状態の場合は tanma_fadesleep() を実行する。



(a) 導入前



(b) 導入後

図 4.3: スケジューリング機能と音量の自動調整機能の導入

4.3.3　tanma_fadeout

tanma_fadeout() は音楽のフェードアウトを行う関数である。呼ばれるごとに、現在の音量を一定の値だけ下げる処理を行う。この関数はプロセスが FADEOUT 状態である場合に進捗状況を調べる関数 tanma_progress() によって呼ばれる。tanma_progress() は 0.5 秒ごとに呼ばれるので、プロセスが FADEOUT 状態のままであるならば同様に 0.5 秒ごとに呼ばれる。変更する音量の値は、tanma_progress() で取得された INIT 状態での音量の値から 0 になるまでのフェードアウトが 3 秒以上かかるように設定される。この値は INIT 状態での音量の値から計算される。例えば音量の初期値が 60 であり 0.5 秒おきに tanma_fadeout() が呼ばれるとすると、音量の値は 10 ずつ減らされ 3 秒経過した時点で音量の値は 0 となる。音量の初期値が 30 であれば、音量の値は 5 ずつ減らされる。

音量はマスタ音量を変更する。これにより、mp3 や wav などの音楽ファイルの再生に限らず、CD プレイヤーや MIDI などによる音楽再生にも適用することが可能である。ステレオ再生では音量が左右で異なる場合があるので、これに対応できるように音量の調整は左右別々に行われる。例えば左の音量の初期値が 60、右の音量の初期値が 30 で 0.5 秒ごとに tanma_fadeout() が呼ばれるとすると、左の音量の値は 10 ずつ、右の音量の値は 5 ずつ減らされる。左右の音量が共に 0 となった場合は、SLEEP 状態に移り tanma_fadesleep() 関数を実行することで、このプロセスは一定時間停止させられる。

音楽再生プロセスに過度の負荷がかかる場合、tanma_progress() が呼ばれるインターバルが 0.5 秒以上かかり、tanma_fadeout() も遅れてしまいフェードアウトにかかる時間が 3 秒を越えて長くなってしまうが、これに関して問題となることはない。なぜならフェードアウトにかかる時間を長くしてでもユーザに不快感を与えないように処理していかなければならないからである。インターバルによらずに 3 秒でフェードアウトが終わる設計にした場合、インターバルが著しく長くなると音量の減少値も大きくなってしま。このため音量が突然大きく減少してしまい、ユーザからはフェードアウトしているようには聞こえなくなってしまう。フェードアウトを行う目的は、突然音楽が停止させられることによるユーザへの不快感を抑制することである。この目的を達成させるために、フェードアウトにかかる時間を長くしてでも音量を少しずつ小さくしていかなければならない。インターバルが長くなればそれだけ、フェードアウトを行っている間にカーネルバッファに保持されているデータがなくなってしまい、音が途切れやすくなってしまう。しかしフェードアウト中の音の途切れは、それほどユーザには聞こえなく、大きな不快感を与えることはないので問題視しないことにした。

4.3.4　tanma_fadein

tanma_fadein() は音楽のフェードインを行う関数である。フェードアウトを行う関数 tanma_fadeout() とは逆で、呼ばれるごとに現在の音量を一定の値だけ上げる処理を行う。この関数はプロセスが FADEIN 状態である場合に進捗状況を調べる関数 tanma_progress() によって呼ばれる。tanma_fadein() の処理は tanma_fadeout() とほぼ同じ流れ行われる。プロセスが FADEIN 状態のままであるならば 0.5 秒ごとに呼ばれる。変更する音量の値は、0 から INIT 状態での音量の値になるまでのフェードインが 3 秒以上かかるように設定される。音量はマスタ音量を変更し、左右の音量が共に初期値に戻った時点で INIT 状態に移る。

tanma_fadein() では tanma_fadeout() で議論された音量変更にかかる時間の問題は考えなくて良い。なぜなら tanma_fadein() は進捗状況に問題がない場合に呼ばれるので、tanma_fadein() が呼ばれるインターバルが著しく長くなることはないからである。

4.3.5　tanma_fadesleep

tanma_fadesleep() はこの関数を呼んだプロセスを一定時間停止させる処理を行う。この関数はプロセスが SLEEP 状態である場合に、tanma_progress() あるいは tanma_fadeout() によって呼ばれる。どちらの場合も実行中の音楽再生プロセスの進捗が思わしくなく、制御させる必要があるために呼んでいる。停止時間はユーザの感性と処理能力のバランスを考え、3 秒とした。この値は後述する 5 章の実験により設定した値である。ユーザによってはこの停止時間を変更したい場合もあるので、この要求に対応するために簡単なインタフェースを用意した。インタフェースに関しては後述する。

tanma_fadesleep() は一定時間停止させるために schedule_timeout() 関数を呼ぶ。引数に停止時間を与えることで、指定した時間がたつまで CPU を放棄し、この音楽再生プロセスを停止させる。指定した時間が経過した後は、CHECK 状態に移る。

一定時間経過するまでこのプロセスは停止させられるため、この間にアプリケーションが命令を送っても待たされてしまう。この命令は一定時間の停止が終了したあと実行される。このようにユーザからの命令が待たされてしまうことに、ユーザが不快に感じることが考えられる。しかし、停止時間は 3 秒ほどの短い時間なので、ユーザがこの時間だけ待たされても問題ないとした。

4.3.6　考えられる新たな実装手法

es1371_write() 内で tanma_progress() を呼ぶことで、音楽を再生するプロセスを制御することができる。しかし、アプリケーションの write システムコール呼び出しが長時間遅れると、フェードアウトを行う前に音が途切れてしまう場合がある。es1371_write() 内で進捗状況を調べるころにはすでにカーネル内のサウンドデータがなくなってしまい、音が途切れてしまう。この問題に対処するには、es1371_write() 以外でカーネル内のサウンドバッファを監視する方法が考えられる。こうすることでより一定時間ごとに進捗状況を監視でき、音が途切れる前にフェードアウトを実行することが可能である。別の手法として、あくまで es1371_write() 内で進捗状況の監視を行い続けるのであれば、音が途切れた場合にその時点で音楽を停止させる方法が考えられる。いったん途切れが生じてしまった場合は改めてフェードアウトをする必要はないという考えから、この手法はきている。これらのような実装を行うことが今後必要である。

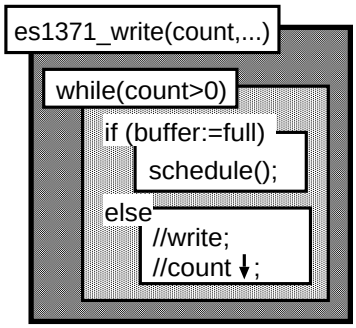
4.4　タイムマシン再生

4.4.1　実装の流れ

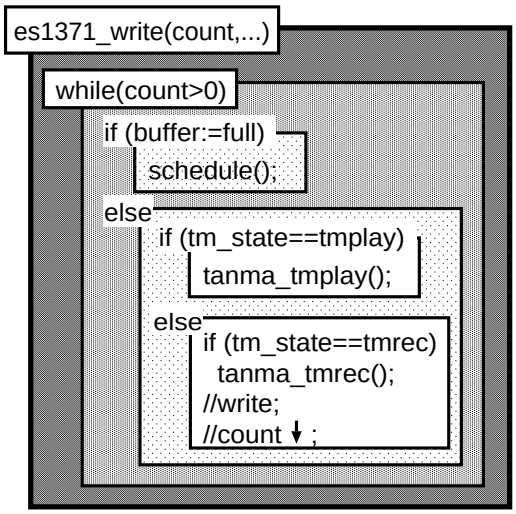
タイムマシン再生を実行するために、tanma_tmrec() と tanma_tmplay() の 2 つの関数を用意した。tanma_tmrec() がサウンドデータの保持を行い、tanma_tmplay() がサウンドデータの再生を行う。これらの関数は共に音楽再生処理を行う es1371_write() 内で呼ばれ、従来の処理であるアプリケーションバッファからカーネルバッファへのサウンドデータ書き込みを利用または制御する。tanma_tmrec() と tanma_tmplay() は es1371_write に図 4.4 のように組み込まれる。タイムマシン再生を実行する状態にある場合、tanma_tmplay() が呼ばれる。この関数が呼ばれる場合、es1371_write 内の従来の処理であるサウンドデータ書き込みは実行されない。よって、アプリケーションから指定されたサウンドデータを書き込んだことをアプリケーションに返すことはない。また、現在再生中のサウンドデータをあとでタイムマシン再生として改めて再生する必要がある場合、tanma_tmrec() が呼ばれる。この関数は従来のサウンドデータ書き込みと同時に行われる。

タイムマシン再生は図 4.2 に示すスケジューリングの状態遷移とは別の状態遷移によって処理が行われる。タイムマシン再生における状態遷移図を図 4.5 に示す。プロセスの状態は 3 種類用意されている。

- NORMAL: 初期状態。タイムマシン再生のための処理は行わない。
- TM_REC: タイムマシン再生を行う予定のサウンドデータを専用バッファに書き込む状態。



(a) 導入前



(b) 導入後

図 4.4: タイムマシン再生機能の導入

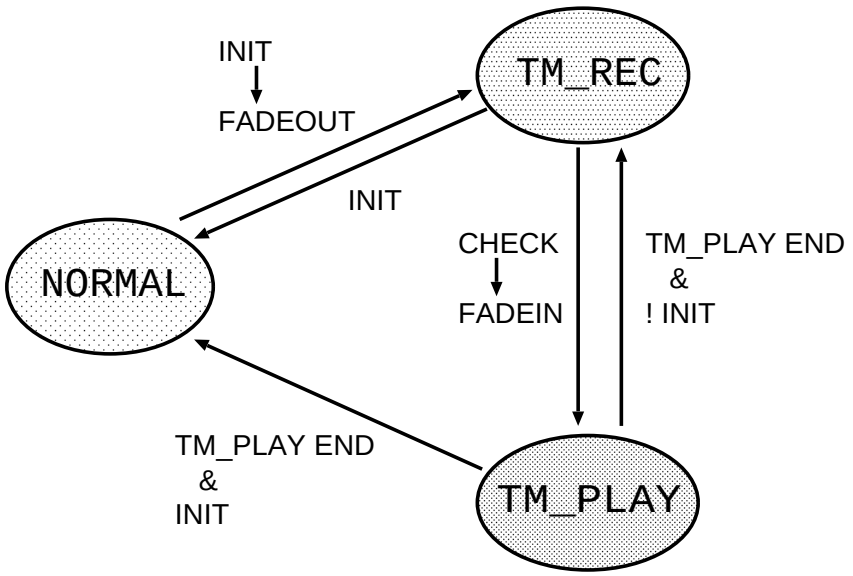


図 4.5: タイムマシン再生の状態遷移

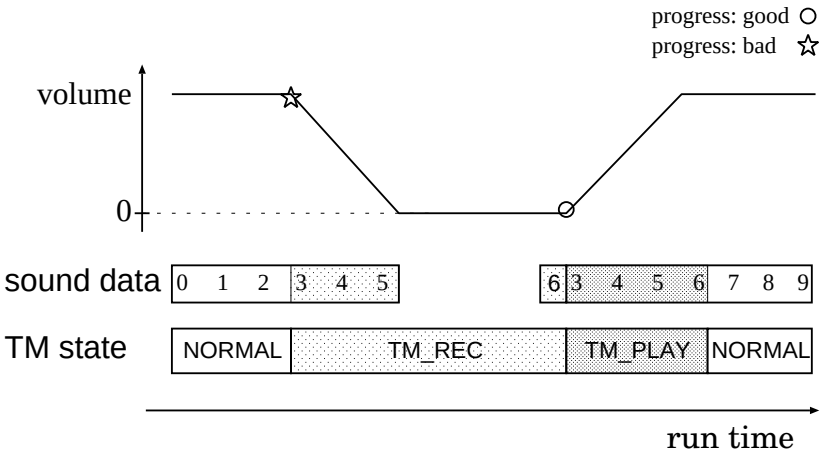
- TM_PLAY: タイムマシン再生を行う状態。

これらの状態は図 4.2 に示すスケジューリングの状態遷移に影響を受けて遷移する。プロセスのスケジューリング状態が INIT 状態にあるとき、タイムマシン再生の状態は NORMAL 状態にある。この状態ではタイムマシン再生のための処理は行わず、従来の処理であるアプリケーションバッファからカーネルバッファへのサウンドデータの書き込みを行う。進捗状況の悪化が探知され、スケジューリング状態が INIT 状態から FADEOUT 状態に遷移すると、タイムマシン再生の状態は NORMAL 状態から TM_REC 状態に遷移する。この状態では、従来のカーネルバッファへの書き込みに加えて、タイムマシン再生用にカーネル内に用意された別のバッファにも書き込みを行う。この書き込みはスケジューリング状態が SLEEP 状態になったあとも続けられ、CHECK 状態のときに再生されるサウンドデータをタイムマシン再生用のバッファに書き込む。スケジューリング状態が SLEEP 状態に移ることなく進捗状況が問題なくなり、スケジューリング状態が FADEOUT 状態から FADEIN 状態を経て INIT 状態に遷移すると、タイムマシン再生の状態は TM_REC 状態から NORMAL 状態に戻る。このとき、タイムマシン再生用のバッファにそれまで書き込まれたサウンドデータは破棄される。スケジューリング状態が SLEEP 状態に移った後に進捗状況が問題なくなり、スケジューリング状態が SLEEP 状態から CHECK 状態を経て FADEIN 状態に遷移すると、タイムマシン再生の状態は TM_REC 状態から TM_PLAY 状態に移る。この状態では、タイムマシン再生用のバッファに書き込まれたサウンドデー

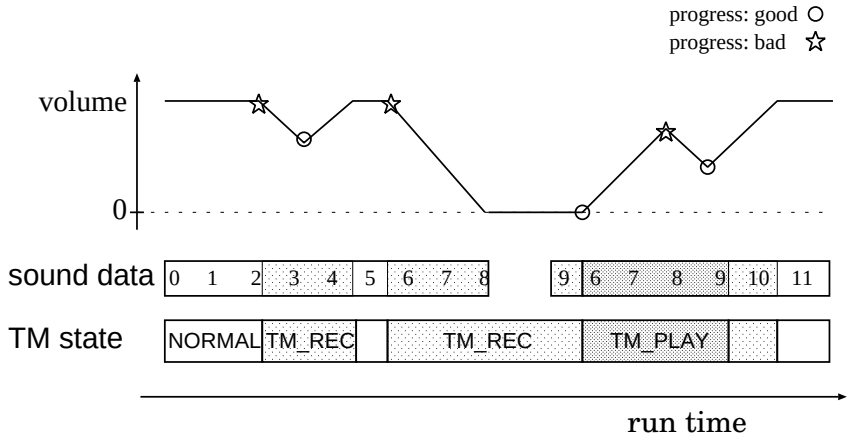
タをカーネルバッファに書き込む処理を行う。この状態のときでは、従来の処理であるアプリケーションバッファからカーネルバッファへのサウンドデータの書き込みは制御される。全てのタイムマシン再生用のサウンドデータが再生し終わると TM_PLAY 状態から別の状態への遷移が行われる。再生し終わったときのスケジューリング状態が INIT 状態であった場合、TM_PLAY 状態から NORMAL 状態に移る。スケジューリング状態が INIT 状態でなかった場合は、TM_PLAY 状態から TM_REC 状態に移る。TM_REC 状態に移る理由は、音量が初期値に達していなく、再生中のサウンドデータがユーザには聞こえづらかったとして改めて再生される可能性があるからである。

タイムマシン再生の状態遷移が実際どのように行われるかを図 4.6 で示す実行例で説明する。図 4.6（a）は音楽プロセスの状態が単純な遷移をしたときの、タイムマシン再生の状態遷移の様子を示す。音楽再生プロセスの進捗状況が悪化し音量のフェードアウトが実行されると、タイムマシン再生の状態は NORMAL 状態から TM_REC 状態に移る。この状態中に再生されているサウンドデータ 3〜6 はタイムマシン再生用のバッファに書き込まれ保持される。進捗状況が問題なくなり音量のフェードインが実行されると、タイムマシン再生の状態は TM_REC 状態から TM_PLAY 状態に移る。この状態ではタイムマシン再生用のバッファに保持されていたサウンドデータデータ 3〜6 が再生される。保持されていたサウンドデータが全て再生し終わったあと、プロセスのスケジューリング状態を調べる。（a）の場合、プロセスのスケジューリング状態は INIT 状態であったので、タイムマシン再生の状態は TM_PLAY 状態から NORMAL 状態に移る。これにより、初期状態に戻ったことになる。

図 4.6（b）は音楽プロセスの状態が（a）よりも複雑な遷移をしたときの、タイムマシン再生の状態遷移の様子を示す。（b）では進捗状況が目まぐるしく変わるためにスケジューリング状態の遷移も多くなり、それに伴いタイムマシン再生の状態も複雑に変化している。音楽再生プロセスの進捗状況が悪化するとタイムマシン再生の状態は NORMAL 状態から TM_REC 状態に移るが、すぐに進捗状況が問題なくなり元の INIT 状態に移った場合、タイムマシン再生の状態は TM_REC 状態から NORMAL 状態に移る。このとき TM_REC 状態中に保持していたサウンドデータ 3〜4 はタイムマシン再生される必要がなくなったので破棄される。再び進捗状況が悪くなり、今度はスケジューリング状態が SLEEP 状態に入る。その後進捗状況は問題なくなり、タイムマシン再生状態は TM_PLAY 状態に移り、タイムマシン再生用のバッファに保持されているサウンドデータ 6〜9 が再生される。保持されていたサウンドデータが全て再生し終わったあと、スケジューリング状態は INIT 状態ではないので、タイムマシン再生状態は TM_REC 状態に移る。このまま進捗状況が悪いまま停止する場合、現在再生しているサウンドデータを改めて再生する必要があるからである。



(a) 単純な場合



(b) 複雑な場合

図 4.6: タイムマシン再生の実行例

4.4.2　tanma_tmrec

tanma_tmrec() はタイムマシン再生が必要なサウンドデータをカーネル内の専用バッファに書き込む処理を行う。この関数は、あとでタイムマシン再生をする必要がある状態になった場合に、es1371_write() 内でのアプリケーションバッファからカーネルバッファへのデータ書き込みの際に呼ばれる。つまり、アプリケーションバッファからのサウンドデータは、カーネルバッファとタイムマシン再生用のバッファの両方に書き込まれていく。あとでタイムマシン再生をする必要がある状態とは、FADEOUT 状態、FADEIN 状態、CHECK 状態のいずれかであることを指す。ただし、tanma_tmplay() によって再生されるデータが存在する場合は、この関数は呼ばれない。tanma_tmrec() が書き込みを行うタイムマシン再生用のバッファは同じカーネルの es1371.c 内に用意されている。このバッファはリングバッファとなっており、バッファサイズはデフォルトでは 50 秒分のサウンドデータが保持できるサイズとなっている。3 秒かけてフェードアウトし 3 秒間の停止と 0.5 秒間の進捗状況確認を繰り返すと仮定すると、継続した停止時間が 3300 秒以内ならばこのバッファが溢れることはない。それ以上停止を継続した場合は古いデータから少しずつ破棄されてしまうが、音楽の重要性が低いので長時間停止した場合は、どこまで再生されていたかをユーザは意識しなくなると考えられる。そのため、始めの部分の再生ができなくてもそれほど問題ではないであろう。

4.4.3　tanma_play

tanma_tmplay() はタイムマシン再生を行う関数である。tanma_tmrec() によって専用バッファに保持されていたサウンドデータの一定量だけカーネルバッファに書き込む処理を行う。この関数は SLEEP 状態から FADEIN 状態に移った後に es1371_write() 内で呼ばれ、それ以降は専用バッファに保持されているサウンドデータがなくなるまで呼ばれ続ける。この関数を呼ぶ状態である限り、es1371_write() 従来の処理であるアプリケーションバッファからカーネルバッファへのサウンドデータの書き込みは後回しにされる。専用バッファ内のサウンドデータが空になった場合、そのときのプロセスの状態が INIT 状態であるならば NORMAL 状態に移り、プロセスの状態がそれ以外の場合では TM_REC 状態に移る。tanma_tmplay() はそれ以降、再び SLEEP 状態から FADEIN 状態への遷移が行われない限り、呼ばれることはない。

tanma_tmplay() によるカーネルバッファへのサウンドデータ書き込みは、es1371_write() 従来のサウンドデータ書き込みに取って代わって呼びだされる。es1371_write() はタイムマシン再生を実行する際、カーネルバッファに空きがある場合は tanma_tmplay() によるサウンドデータ書き込みを行い、カーネルバッファが一杯である場合は schedule() 関数をよび CPU を放棄する。

4.4.4　タイムマシン再生とは異なる QoS 処理

タイムマシン再生では、前に再生した部分から音楽の再開を行うが、それとは異なる手法が考えられる。それは、音楽を停止していた時間だけ先に進んだ部分から音楽の再開を行うことである（図 4.7 に示す）。つまり、実時間軸にそって再開させることである。このようにすることで、ユーザに一定時間の停止がおこったことを感じさせにくくすることが可能である。この手法は長い時間の停止では音楽の大部分を聞くことができなくかえって問題である。しかし短い時間の停止ならばその効果を発揮できる。音楽を聞きながらリズムをとっているユーザには、音楽が時間軸にそって先に進んでいた方が違和感なく音楽が再開されているように聞こえるだろう。一方のタイムマシン再生は、長い時間の停止でその効果を発揮できる。ユーザがどの部分で音楽が聞こえなくなってしまったかを思い出させてくれる。短い時間の停止では、前に述べたリズムをとっているユーザには、時間軸とずれた音楽再生がおこなわれるので違和感を与えてしまう可能性がある。

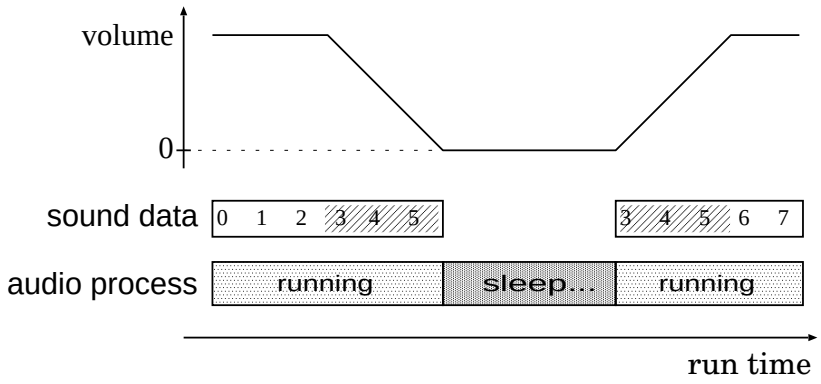
時間軸にそっての再生を実現させるには、タイムマシン再生とは異なる技術を用いなければならない。それは、タイムスタンプを使うことである。音楽を再開させる際にタイムスタンプを調べ、どこまで先のサウンドデータを再生し始めなければならないのかを決める。その後、必要なサウンドデータが渡されるまで音楽を再生させずにアプリケーションに再生したと返す。このような設計が考えられる。

しかし、この手法には問題がある。他のプロセスとの整合性を保証することができない。音楽アプリケーションによっては音楽を再生するプロセスだけでなく、さまざまなプロセスが同時に処理を行っている場合がある。この場合、音楽だけを先に進めてしまうと他のプロセスとの整合性に問題が生じる可能性がある。

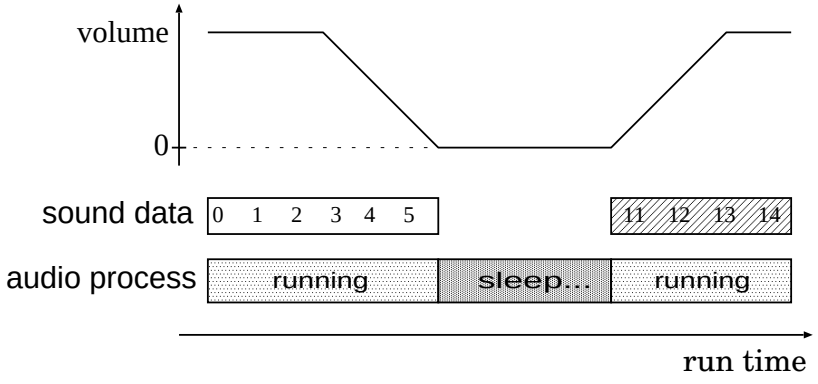
今回実装した Tanma にも同様の問題が生じてしまうが、音楽再生プロセスと関連のあるプロセスを全て停止させることで対処することが可能と考えられる。動画再生プロセスや再生秒数の表示など、関連するプロセスを全て停止させることで整合性が保証される。

4.5　Tanma のインタフェース

Tanma の設定をユーザの要求に応じて簡単に変更できるように、sysctl コマンドによる設定変更を可能にした。sysctl コマンドはカーネルパラメータを読み書きすることができるコマンドである。このコマンドを利用し、Tanma のために用意したパラメータを変更することで、カーネルコンパイルをせずに Tanma の設定を変えることが可能である。Tanma のパラメータは /proc/sys/dev/ 以下に用意されたディレクトリ tanma/に納められている。tanma_system は



(a) タイムマシン再生



(b) 時間軸にそって再生

図 4.7: タイムマシン再生とは別の可能性

Tanma を使用するかどうかを設定できるパラメータである。この値を変更することで、Tanma による音楽再生プロセスの制御を行うかどうかを切り替えることができる。sleep_time は Tanma による音楽再生プロセスの停止を何秒間行うかを設定できる。値の単位は秒数で設定されており、デフォルトではこの値は 3 秒に設定されている。3 秒に設定した理由はユーザの感性と処理能力のバランスを考えた結果である。これについては後述する 5.3 節で説明する。デフォルトの値に満足がいかない場合は変更を行えば良い。重要性の高い処理が終了したあと、即座に音楽再生プロセスを再開させたい場合は sleep_time の値を小さく設定すればよい。より早く重要性の高い処理を済ませてしまいたい場合は、sleep_time の値を大きく設定することで実現できる。

4.6　その他の実装

4.6.1　状態のリセット

Tanma により音量が変更された状態でデバイスを閉じた場合、もとの音量に戻るように実装した。アプリケーションからの close システムコールに対応する es1371.c 内の関数 es1371_release() に実装を施した。Tanma により変更されたものを、変更される前の初期状態のものにリセットする。音量を初期値に戻し、プロセスの状態を INIT 状態、NORMAL 状態にする。

4.6.2　ユーザからの音量変更命令に対応

音量が変更された状態でユーザにより音量の値が変更された場合、音量の初期値を変更するようにした。ユーザからの命令にはきちんと対応できるようにしなければならない。このように設定しなければ、INIT 状態に戻れなくなる可能性がある。初期値の変更を行うと、その変更は INIT 状態に戻る際に反映される。この処理を行うために、アプリケーションからの ioctl システムコールに対応する es1371.c 内の関数 es1371_ioctl() に実装を施した。マスタ音量の変更命令が出された場合、Tanma が保持する音量の初期値を変更する。

4.7　発展：複数の音楽再生処理

4.7.1　重要性の高いプロセスが音楽再生プロセスにより生じる問題点

ここまで Tanma は重要性の低い音楽再生プロセスと重要性の高い非音楽再生プロセスを対象としていた。しかし、重要性の高いプロセスが音楽再生

プロセスの場合は問題が生じる重要性の低い音楽再生プロセスを制御する際に音量を変更すると、重要性の高い音楽再生プロセスの音量も変更してしまう。これは Tanma がマスタ音量を変更しているために生じる。これに対し、互いに異なる音量レベルを変更する方法が考えられる。例えば、片方の音楽再生プロセスが PCM 音量を扱っており、もう片方のプロセスが CD 音量を扱っているような場合ならば、互いの音量変更は影響を与えることはない。しかし、必ずしもこのような条件が満たされるとは限らない。このような音量の変更の問題に対する方法が、複数の音楽を再生する上で必要である。

4.7.2　Linux における複数の音楽再生処理のメカニズム

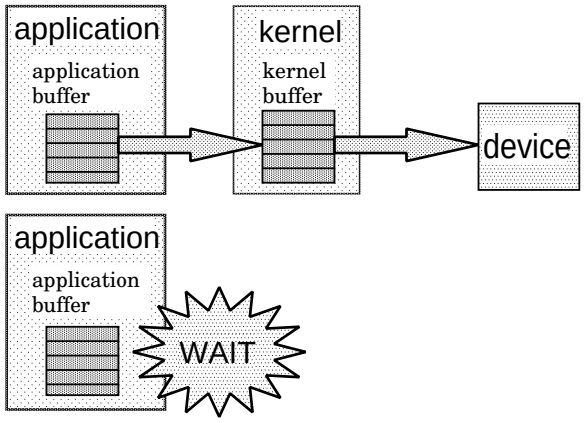
複数の音楽再生処理は Linux の中ではどのように行われているのかを述べる。図 4.8 で示すように 3 通りの処理方法が存在する。Linux のシステム設計では、1 つのプロセスしかサウンドデバイスを開くことはできない（図 4.8 (a)）。あるプロセスがサウンドデバイスを開くと、このプロセスがサウンドデバイスを閉じるまで他のプロセスは利用することはできない。Linux で複数の音楽を再生するには、主に 2 つの方法が挙げられる。まず 1 つは、サウンドデバイスを複数用意し、各プロセスがそれぞれ異なるサウンドデバイスを利用することである（図 4.8（b））。もう 1 つの方法は、サウンドデータを合成することである（図 4.8（c））。Esound などのサウンドデーモンを利用することで、各アプリケーションから転送されるサウンドデータはサウンドデバイスに渡る前にミックスされ、その結果がサウンドデバイスに転送される。サウンドデバイスはデーモンプロセスからのサウンドデータを処理することで、複数の音楽を再生することが可能である。

4.7.3　複数の音楽再生処理に対処

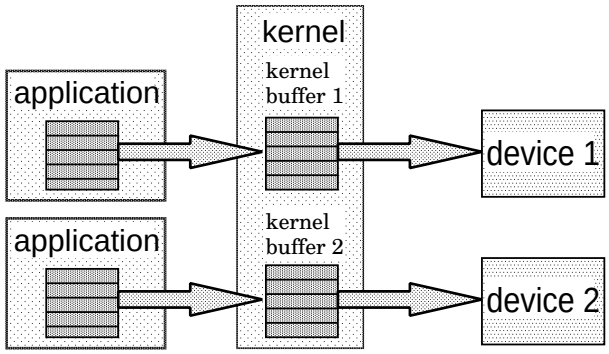
複数の音楽再生処理にはメカニズムの異なる 3 つの方法が利用されているので、これらの方法に対処する方法を別々に考える必要がある。3 つの方法には音量の変更の問題に限らず、Tanma のこれまでの実装では解決できない独自の問題点を抱えているので、その問題にも対処する方法をここでは示す。

オープンロックされた音楽再生処理に対する QoS 処理

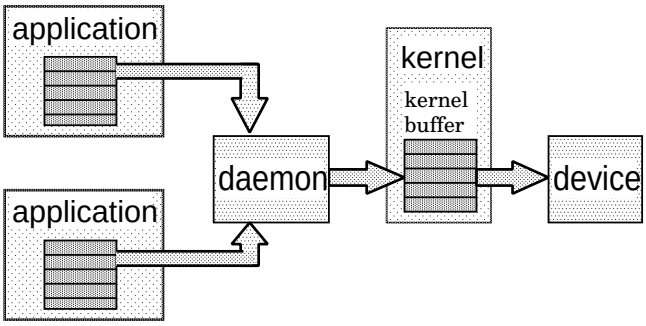
デバイスのオープン制御に対処する手法を説明する。Tanma の競合を回避する目的とは違うが、通常 1 つのデバイスに複数のアプリケーションがアクセスする場合、早いもの勝ちとなり、遅れたものは終わるまで待たされてしまう。重要性の違いを意識すると、これは問題である。そこで、重要性の高い音楽を優先させて再生するようにした。重要性の低いプロセスを制御させ



(a) デバイスオープンのロック



(b) 複数のサウンドデバイス使用



(c) サウンド合成

図 4.8: 複数の音楽再生の処理方法

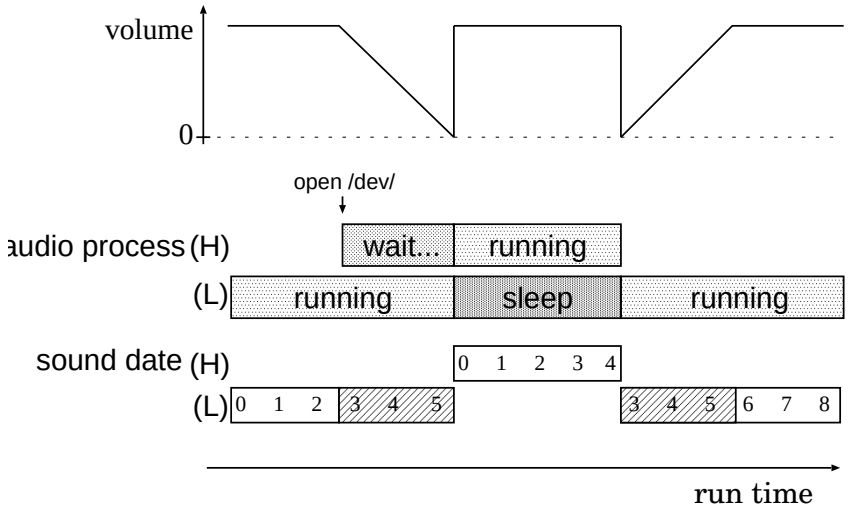


図 4.9: オープンロック解除による重要性の高い音楽再生処理の実行

た後に重要性の高いプロセスを再生する方法が考えられる。制御する際にはフェードアウトさせて制御する手法と、フェードアウトさせないで制御する手法が考えられる。重要性の高い音楽再生プロセスが長い時間の処理ならば、フェードアウトさせて制御したほうがよい。重要性の高い音楽再生プロセスが短い時間の処理ならば、わざわざフェードアウトさせずに重要性の低いプロセスを制御したほうが、違和感を覚える前にもとに戻るので制御されたことが気づかれにくくてよい。この例では、音楽を聞いている最中にメールの着信音がなるシチュエーションが考えられる。

フェードアウトさせて制御する手法の様子を図 4.9 に示す。音楽再生プロセス L を実行中に、より重要性の高い音楽再生プロセス H がデバイスをオープンしようとする。このとき音楽再生プロセス H にはオープンロックがかかり待機させられる。その間に音楽再生プロセス L をフェードアウトさせながら停止させる。このとき再生したサウンドデータ 3〜5 はタイムマシン再生用のバッファに書き込まれる。音楽再生プロセス L が停止した後、音楽再生プロセス H のオープンロックは解除され、音楽再生が実行される。このときフェードアウトにより 0 となった音量は初期値に戻る。音楽再生プロセス H が終了したあと、音量は 0 になり、フェードインをさせながら音楽再生プロセス L を再開させる。このときタイムマシン再生によってサウンドデータ 3〜5 は改めて再生される。

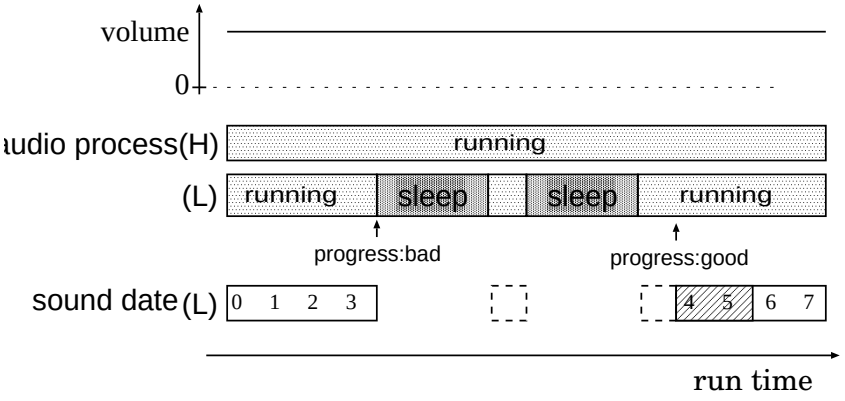
複数のサウンドデバイス使用時における QoS 処理

複数のサウンドデバイス利用することによる複数の音楽処理における問題に対処する方法について述べる。停止後に進捗を調べるための再開はユーザには不快である。しかし、音量をゼロにするば重要性の高い音楽の音量も 0 になってしまう。そこで、重要性の低い音楽は再生しないが、アプリケーションには再生しているように伝える。本来再生されるはずのサウンドデータは、タイムマシン再生機能によって別のバッファに書き込まれる。この別のバッファへの書き込みの進み具合を進捗状況とする。なぜなら、本来 Tanma で進捗値として監視されていたカーネル内のサウンドバッファにはサウンドデータが書き込まれないからである。タイムマシン再生用のバッファはサウンドバッファとは異なり、サウンドデータはデバイスに DMA 転送されることはないので、データが減ることはない。よって、バッファ内のデータ量を調べる方法を用いることはできない。これに対し、一定時間内にこのバッファに書き込まれたデータサイズを進捗値として用いる方法が考えられる。転送されるサウンドデータのサンプリングレート、ビットサイズ、ステレオ/モノラルをもとに一定時間内に書き込まれるべきデータサイズの理想値を計算する。この理想値と進捗値を比較し、進捗状況を把握する。重要性の高い音楽が動いていれば、重要性の低い音楽の再開をすぐに実行する。突然音楽を再開させることはユーザに不快感を与える場合もあるが、重要性の高い音楽の音量を変えるわけには行かない。また、重要性の高い音楽が再生されている最中での音楽の再開なので、ユーザへの不快感は和らぐことが考えられる。逆に重要性の高い音楽が終了していたならば、フェードインさせながら音楽を再開することが考えられる。

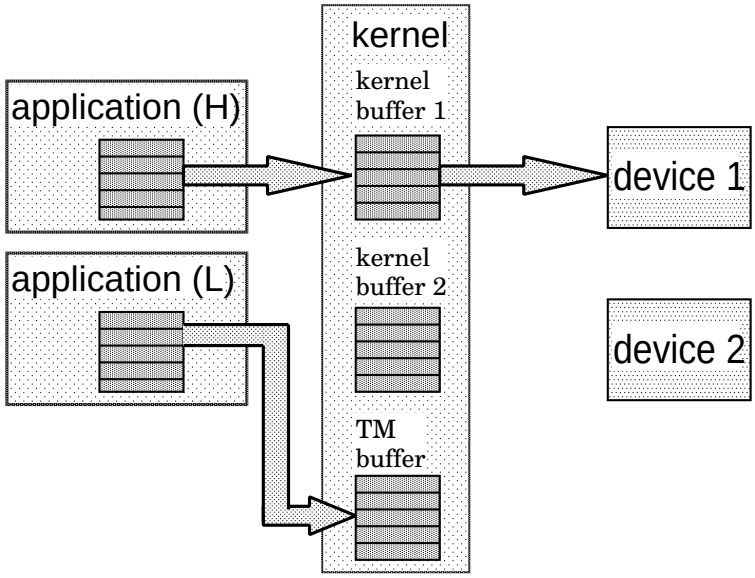
複数のサウンドデバイスを使用する場合の音楽の制御の様子を図 4.10 に示す。図 4.10（a）は制御の概要を示す。重要性の高いプロセスが音楽プロセスの場合、重要性の低い音楽プロセスの進捗状況が悪化したらすばやく制御する。このとき従来の Tanma の機能である音量の自動調整機能は働かない。一定時間停止した後、進捗状況を調べるための一時的な処理再開ではアプリケーションバッファからカーネルバッファへのサウンドデータ書き込みは行われない（図 4.10（b））。サウンドデータ 4〜5 は再生されず、タイムマシン再生用のバッファに書き込まれ保持される。進捗状況が問題なくなれば、保持されていたサウンドデータ 4〜5 のタイムマシン再生をおこなう。

この手法にはかなりの制限が生じる。1 つは、サウンドデバイスを複数用意しなければならないことである。もう 1 つは、ADPCM のような圧縮方式を利用する場合、進捗状況を把握することができない。この手法では従来の Tanma の進捗状況監視とは異なり、一定時間内に書き込まれたデータサイズを進捗として監視している。しかし圧縮方式を利用してしまうと本来のデータサイズを計算することができない。

また、この手法ではカーネルの遅延に対応することができない。進捗が遅



(a) 重要性の低い音楽再生処理の制御の流れ



(b) タイムマシン再生用バッファリング

図 4.10: 複数の音楽再生処理における QoS 処理

れる原因の 1 つに挙げられる、schedule() からの復帰の遅れに対応することができない。カーネル内のサウンドバッファに書き込みは行わなれず、このバッファがいっぱいになり schedule() 関数を呼ぶことで待つという処理を行わないからである。もう 1 つの原因である、アプリケーションの write システムコール呼びだしの遅れには対応することは可能である。

サウンドが合成される場合での対処法

サウンドデーモンを利用することによる複数の音楽処理に対しては、音量の変更の問題よりも深刻なことがある。それは、各アプリケーションの進捗状況を調べることができないことである。この原因は、カーネル側からは Esound などの 1 つのプロセスがサウンドデータを転送しているように見えるからである。各アプリケーションのサウンドデータが合成されたものを受け取っているので、カーネルレベルでアプリケーションそれぞれの進捗状況を把握することはできない。この問題に対処するには、サウンドデーモン内に転送されてくるサウンドデータの流量を調べる方法が考えられる。サウンドデーモン内から各アプリケーションから転送されたサウンドデータのデータ量と、その再生レートから予想される理想のデータ量を比較することで、進捗状況を把握することができる。しかし、このようにサウンドデーモンを適応させるためのライブラリを作成することが困難である。また、各アプリケーションが進捗状況を監視する方法も考えられる。

4.7.4　オープンロックへの対処法の実装

前説で述べた複数の音楽制御への対処の中から、オープンロックの問題への対処法を実装した。通常 1 つのサウンドデバイスをプロセスがオープンすると、他のプロセスは使用することができなくなる。そこで、重要性の低いプロセスを制御させた後に重要性の高いプロセスがデバイスをオープンできるようにした。今回の場合、従来の Tanma のスケジューリングである progress-based regulation を用いることはできない。重要性の違いはプロセスの nice 値をみて判断することにした。

デバイスオープンのロックは、アプリケーションからの open システムコールに対応する、es1371.c 内の関数 es1371_open() で行われる。現在実行中の音楽再生プロセスの nice 値を取得しておく。そして、別の音楽再生プロセスがデバイスをオープンしようと es1371_open() を呼んだ場合、このプロセスの nice 値を調べる。実行中の音楽再生プロセスのほうが低い nice 値の場合は重要性の高いプロセスなので、従来と同じくこのプロセスを待機させる。実行中の音楽再生プロセスのほうが高い nice 値の場合は重要性の低いプロセスなので、この場合はまず実行中の音楽再生プロセスを停止させる。この停止

は音楽再生を行う関数 es1371_write() 内で行われ、フェードアウトしたあとに停止する。この停止は従来の Tanma の一定時間停止というものではなく、新たに実行される音楽再生プロセスが終了するまで停止を続ける。実行中であった音楽再生プロセスを停止させたあと、es1371_open() 内のデバイスオープンのロックを解除し、このオープンを行った音楽再生プロセスを実行する。この際にフェードアウトさせていた音量はもとの値にもどす。この重要性の高い音楽再生プロセスが終了したあとに、停止させられていた重要性の低い音楽再生プロセスが再開される。重要性の高い音楽再生プロセスは終了の際に close システムコールに対応する関数 es1371_release() で、次に実行すべき nice 値をもつ音楽再生プロセスを起こす。これにより停止させられていた重要性の低い音楽再生プロセスが再開される。再開させる際には音量をいったん 0 にし、それからフェードインしながら再開される。このときタイムマシン再生によりフェードアウト部分から再開される。この制御手法は音楽再生プロセスが 3 つ以上でも有効である。

第5章　実験

5.1　Tanma上でのアプリケーションの動作検証

今回実装したTanmaは音楽再生プロセスを対象にしている。実装はカーネルにのみ行っているのでアプリケーションを書き換える必要はない。これにより全ての音楽アプリケーションを制御することが可能である。実際に検証したところ、wavプレイヤーやmp3プレイヤーでも問題なく動作した。Linuxの音楽プレイヤーであるrealplayer、xmms、mpg123、wavplayに対してTanmaによる制御が実行されたことが確認された。Tanmaはマスタ音量を変更しているので、CDやMIDIの音楽制御も可能である。動画ファイルであるmpeg再生に関しても、動画像は突然停止するが音楽に対してはTanmaが提供するフェードアウト処理を行うことができた。しかし、音楽と動画像とで整合性がとれなくなり、音楽と動画像は共に実時間軸にそった部分から再開された。タイムマシン再生やフェードイン処理は行われなかった。この問題に対処する方法として、音楽再生プロセスと関連のあるプロセスを全て停止させる方法が考えられる。動画再生プロセスや再生秒数の表示など、関連するプロセスを全て停止させることで、整合性が保証されると考えられる。

5.2　実験環境

実験にはCPUがPentium 733MHz、メモリが512MBの計算機を使用し、Linux 2.4.20上で測定した。音楽再生プロセスには音楽プレイヤー「realplayer[13]」を使用し、これと同時に実行する重要性の高い処理には3つの異なる処理を用意した。1つめの処理は、常に一定の負荷がかかるπ計算アプリケーション「pi_fftc[10]」を用いた。指定した桁数までπの値を計算する処理を行う。2つめの処理は、wavファイルからmp3ファイルに変換を行うMP3エンコーダ「午後のこ〜だ[16]」を用いた。3つめの処理として、複数の画像ファイルの画像処理をとりあげた。サイズの異なる数十枚のJPEGファイルのサイズを80%に縮小したものを別のファイルにそれぞれコピーする処理を行う。変換にはconvertコマンドを用いた。

5.3　性能評価

我々が開発した Tanma に関して性能評価とオーバヘッド測定を行った。音楽再生プロセスと重要性の高いプロセスを同時に実行した場合、Tanma がどのように働くのかを調べる。

5.3.1　資源の競合に対する Tanma の性能検証

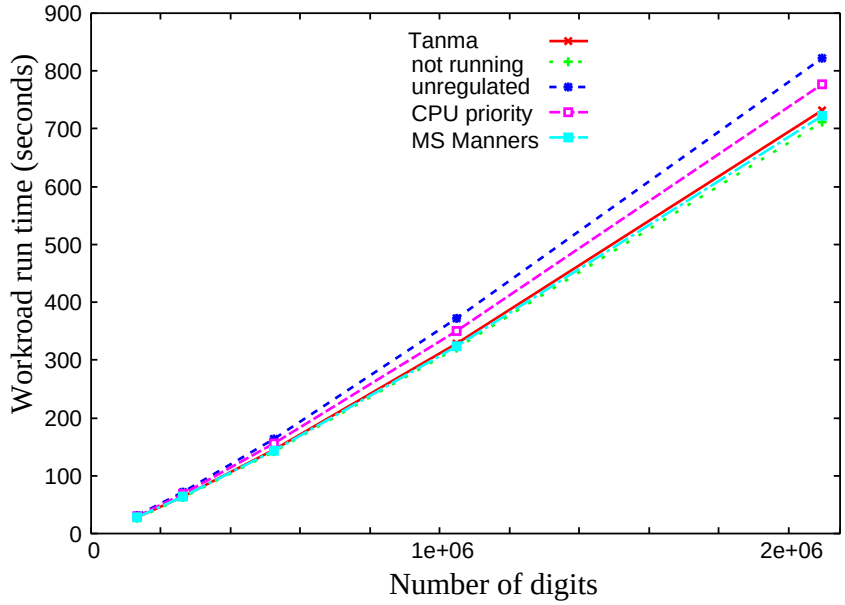
今回実装した Tanma は音楽再生プロセスによる資源の競合を防ぐことが可能である。そこで、音楽再生プロセスの実行中に重要性の高いプロセスを実行するという状況で、重要性の高いプロセスの性能を調べるための実験を行った。音楽プレイヤーとの資源の競合による重要性の高いプロセスの遅延を、Tanma を用いることでどの程度減らすことができるかを測定した。重要性の高い処理として、π計算、エンコード処理、画像処理をそれぞれ用いた。

重要性の高い処理がπ計算の場合

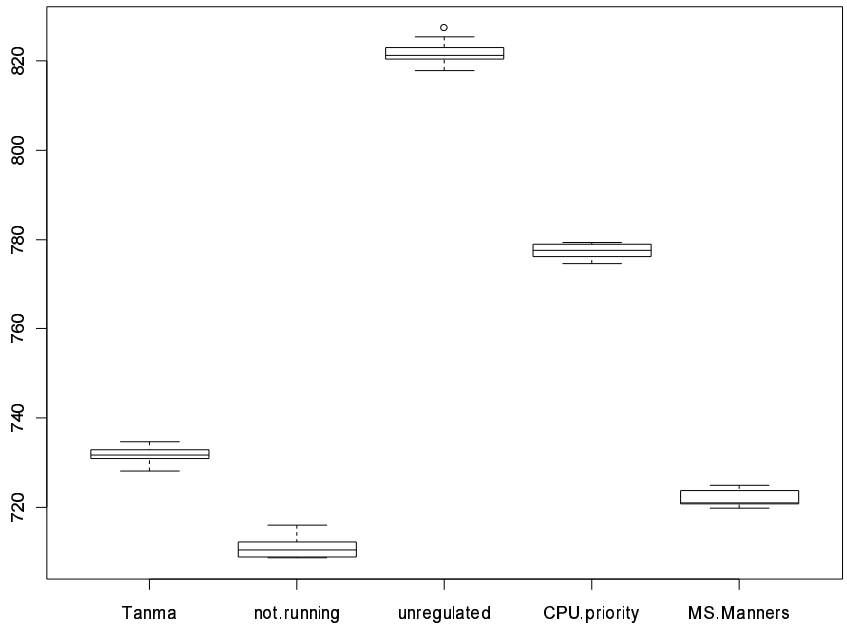
はじめに、重要性の高い処理としてπ計算を実行したときの測定結果を述べる。計算する桁数を変えてπ計算の処理時間をそれぞれ測定した。

図 5.1（a）は音楽プレイヤーが動作している状態でのπ計算にかかった時間の平均値を示す。この図で示されている Tanma が Tanma を用いた場合、not running がπ計算のみで実行を行った場合、unregulated が制御する手法を用いずに音楽を再生させている場合、CPU priority がπ計算に優先的に CPU を割り当てている場合、MS Manners が MS Manners を用いた場合のπ計算の処理時間をそれぞれ表している。πの値を 2,000,000 桁まで計算する時間は、他にプロセスが実行されていない場合に 711 秒かかった。通常のシステム下で、音楽プレイヤーを動作させながら実行したπ計算にかかった時間は、資源の競合が生じ 821 秒となり、15%多く時間がかかった。CPU 優先度を変更しπ計算を優先的に実行したところ 777 秒かかり、9.3%の遅延が残った。これは CPU の競合が解消されていないか、あるいは CPU 以外の資源の競合が生じていることが原因と考えられる。そのため音楽プレイヤーには十分な資源が割り当てられず、音の途切れが激しいものとなってしまった。これらに対し Tanma を働かせながら CPU 優先度を変更したπ計算を実行したところ 731 秒となり、2.8%のオーバヘッドであった。音楽プレイヤーはπ計算の実行開始時に Tanma により停止させられ、π計算が終了するまで 730 秒間停止を継続した。この間に進捗状況の判定のための一時的な処理再開は平均 207 回行われたが、ユーザ側からは停止し続けているようにみえた。この音楽プレイヤーの制御により、π計算だけを実行する場合の時間とかなり近い時間で処理を終えることができた。

また、Tanma と MS Manners とで性能の比較を行った。Tanma では音楽



(a) 計算される桁数あたりの処理時間



(b) 1,000,000 桁のπ計算の処理時間

図 5.1: π計算の処理時間

のフェードアウトを行う時間だけ音楽プレイヤーを止める時間が遅れてしまう。また、停止時間が MS Manners のように初期値 1 秒から進捗状況を判定するたびに倍に設定されていくのではなく、Tanma では 3 秒に固定されているのでより多く進捗状況の判定が行われる。これらの違いによるπ計算の処理時間の差があるかどうかを調べた。MS Manners の方式が動作している状態でのπ計算にかかる時間を測定した結果、πの値を 2,000,000 桁まで計算する時間は 722 秒となり、1.5%のオーバヘッドであった。これは Tanma 上での計算時間とほぼ同じ値となったので、MS Manners の方式との違いによる遅れはほとんどないことがわかった。しかし音楽プレイヤーはπ計算の実行開始から 1028 秒も停止を続け、π計算が終了して 300 秒ほど待たされたあとに音楽が再開された。これはユーザにとっては大変不快である。

2,000,000 桁のπ計算の処理時間に関しての分布を図 5.1（b）の箱ひげ図に示す。箱ひげ図の中央の太線は平均値を表し、下端、中央、上端の水平線は各 25、50、75%タイル値を示している。この図からも Tanma、π計算のみ、MS Manners それぞれ用いた場合はほぼ同じ処理時間となったことがわかる。

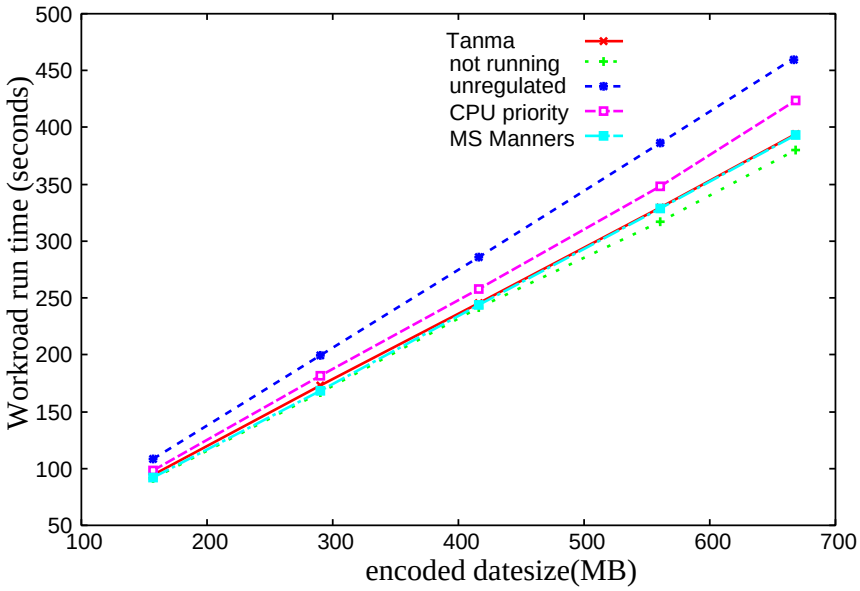
重要性の高い処理が MP3 エンコードの場合

次に、重要性の高い処理として MP3 エンコード処理を実行したときの測定結果を述べる。異なるデータサイズの WAV ファイルを MP3 にエンコーディングする時間をそれぞれ測定した。

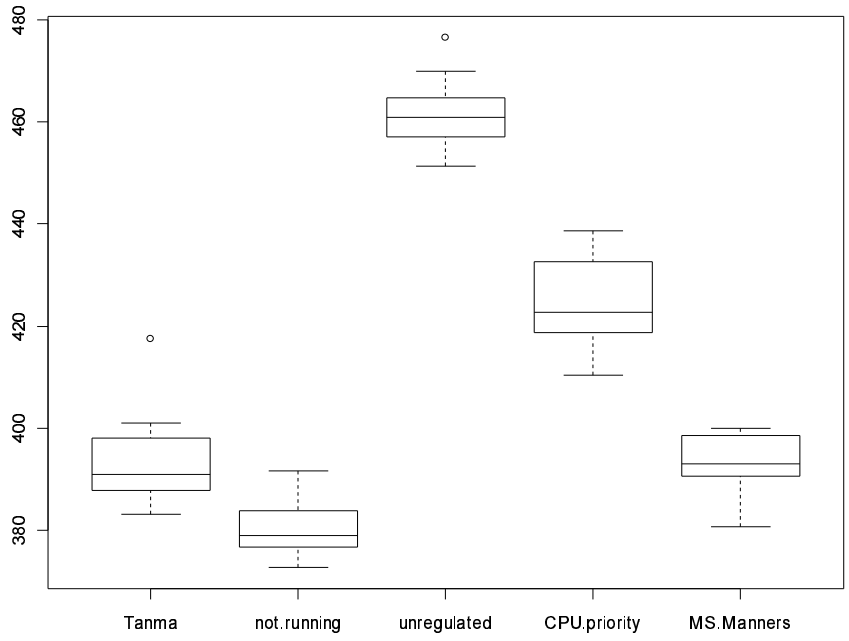
図 5.2（a）は音楽プレイヤーが動作している状態での MP3 エンコード処理にかかった時間の平均値を示す。WAV ファイルのデータサイズ 668Mbyte のときの MP3 エンコード処理時間は、他にプロセスが実行されていない場合に 380 秒かかった。通常のシステム下で、音楽プレイヤーを動作させながら MP3 エンコードを行ったところ、資源の競合が生じ 462 秒かかり、21.5%の遅延が生じた。CPU 優先度を変更し MP3 エンコード処理を優先的に実行したところ 424 秒かかり、11.6%の遅延が残った。しかもこのとき、音楽プレイヤーは音が途切れながら再生処理を行っており、ユーザにとっては不快な処理となっていた。これらに対し Tanma を働かせたところ 394 秒となり、3.7%のオーバヘッドであった。このときの音楽プレイヤーはπ計算の場合と同様に、実行開始時に Tanma により停止させられ、エンコード処理が終了するまで停止を継続した。この制御により、MP3 エンコードのみを実行する場合の処理時間とほぼ変わらない値となった。

また、MS Manners の方式を用いた場合、MP3 エンコード処理時間は 393 秒となり、Tanma 上での計算時間とほぼ同じ値となった。重要性の高い処理が MP3 エンコード処理の場合でも、Tanma は MS Mannres と大差ない処理性能であった。

WAV ファイルのデータサイズ 668Mbyte のときの MP3 エンコード処理時



(a) エンコード対象ファイルのデータサイズあたりの処理時間



(b) 668Mbyte の WAV ファイルに対する MP3 エンコード処理時間

図 5.2: MP3 エンコードの処理時間

間の分布を図 5.2（b）に示す。重要性の高い処理がπ計算の場合と同様に、Tanma を用いることで MP3 エンコード処理をすばやく処理できていることがわかる。

重要性の高い処理が画像処理の場合

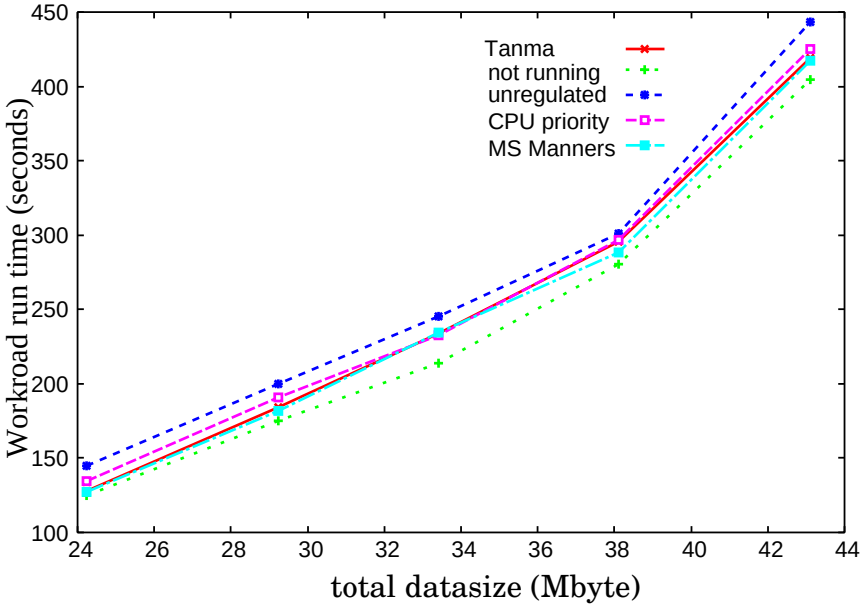
最後に、重要性の高い処理として画像処理を実行したときの測定結果について述べる。4 から 6Mbyte のデータサイズの JPEG 画像ファイル複数枚を、convert コマンドにより大きさを 80%に縮小して別のファイルに保存する処理を行う。処理を行う画像ファイルの枚数別に、それぞれ測定を行った。

図 5.3（a）は音楽プレイヤーが動作している状態での画像処理時間の平均値を示す。画像ファイルの枚数が 90 枚で総データサイズが 43Mbyte のときの画像処理時間は、他にプロセスが処理を行っていない場合は 404 秒であった。音楽プレイヤーを制御させずに実行していた場合、画像処理には 444 秒かかり、9.9%の遅延が生じた。これに対し、CPU を画像処理に優先的に与えたところ、処理時間は 425 秒となり、5%の遅延が残った。そこで、Tanma 上でこれらの処理を実行したところ、画像処理にかかる時間は 419 秒となり、3.7%の遅延となった。重要性の高い処理がπ計算や MP3 エンコードの場合と比べ、あまり遅延の削減には至らなかった。これは、画像処理と音楽プレイヤーとの間に資源の競合が常に探知されたわけではなく、進捗状況が激しく変化したことが原因である。そのため、Tanma 上での音楽プレイヤーは画像処理を実行中は、停止や再開を繰り返していた。MS Mannners を用いた場合も同様に音楽プレイヤーは制御され続けることはなく、画像処理にかかる時間は 417 秒となり、3.2%の遅延となった。

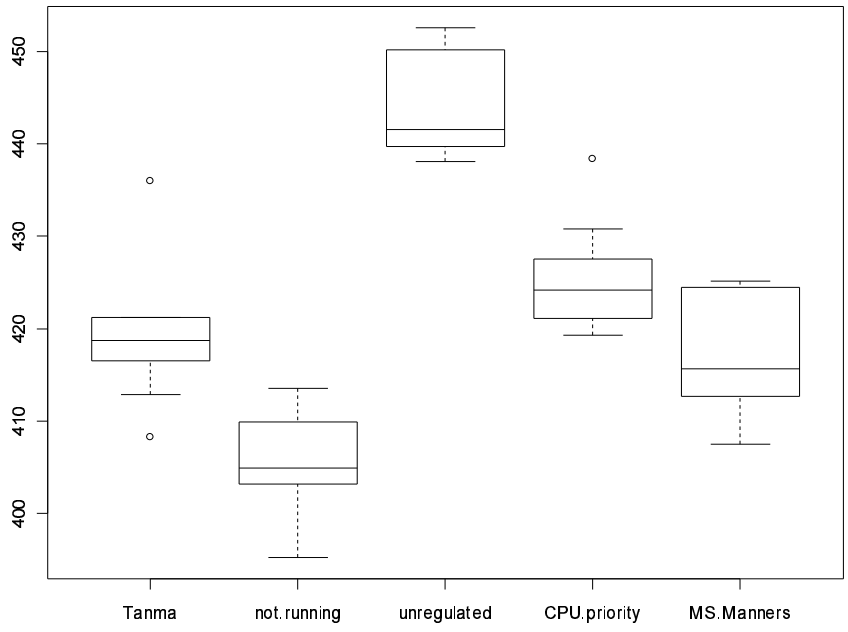
画像ファイルの枚数が 90 枚で総データサイズが 43Mbyte のときの画像処理時間の分布を図 5.2（b）に示す。Tanma 上での画像処理の時間にはばらつきが生じ、早ければ 409 秒かかり、遅ければ 436 秒かかった。これは画像処理のみを実行した場合の時間にもばらつきがあることが原因の 1 つと考えられる。また、画像処理と音楽プレイヤーによる資源の競合が生じた時間にばらつきがあることも原因と考えることができる。また、この図から Tanma、MS Manners とそれらを用いない CPU 優先度を変えた場合では、画像処理時間の差があまりないことがわかる。

まとめ

重要性の高い処理がπ計算や MP3 エンコード処理のように、音楽再生プロセスとの競合が激しい処理の場合、Tanma を利用することでこれらの処理をすばやく処理することができた。また、Tanam と同じく progress-based regulation を用いてスケジューリングを行う MS Manners と比較してみたが、



(a) 画像ファイルの総サイズあたりの処理時間



(b) 総サイズ 43Mbyte の画像ファイルの処理時間

図 5.3: 画像処理の処理時間

処理時間の差はほとんど見られなかった。このことから、Tanma が提供する感性を考慮した QoS 処理を音楽再生プロセスに実行しても、重要性の高い処理の処理速度に影響を与えることはないと言える。

それに対し、重要性の高い処理が画像処理のように、音楽再生プロセスとの競合が頻繁には生じない処理の場合は Tanma を用いても、よりすばやく処理することはほとんできなかった。このように進捗が目まぐるしく変化するような場合に対処するには、進捗を判断するのに時間をかける方法が考えられる。調べた時点での進捗値で判断するのではなく、一定回数だけ進捗値を調べ、それらの値から進捗状況を判断する。これにより、より長い期間内に生じた資源の競合を探知することが可能となり、音楽再生プロセスを制御することが可能である。処理速度以外に考慮しなければならないことは、音楽再生プロセスの QoS である。CPU 優先度を変える手法や MS Manners による音の途切れと、今回の Tanma によるフェードアウトやフェードインが頻繁に生じる音楽とでは、どちらがよりユーザに不快感を与えるものか、今後調査が必要である。

5.3.2　Tanma のオーバヘッド

Tanma では進捗状況の監視と判定、音量の自動調整、タイムマシン再生を行うが、これらの処理によるオーバヘッドを測定した。音楽再生とπ計算を同時に行い、音楽再生の進捗状況を表すカーネルバッファ内の再生データサイズの測定を行う。Tanma の処理オーバヘッドによりアプリケーションプロセスに割り当て可能な資源が減ってしまった場合は、アプリケーションからカーネルへの再生データの転送に遅延が生じ、カーネルバッファ内の再生データが減ってしまうはずである。データサイズの測定は 700 秒間行い、バッファ内のデータの増減がオーバヘッドにより変化するのかどうかを調べた。カーネルバッファのサイズは 131,072 バイトである。実験は (1) 通常のシステム、(2) 進捗状況の監視と判定のみ行う Tanma、(3) 音楽のフェードアウト/インを繰り返す Tanma、これら 3 つの環境下で実行し比較した。(2) ではカーネルバッファ内のデータサイズを取得し閾値と比較する処理を行う。ただしバッファ内のデータサイズが閾値より小さい場合でも、音量の自動調整やタイムマシン再生で行われる処理は実行しないものとした。(3) では本来は進捗状況が変化したときに行われる音量の自動調整やタイムマシン再生によるオーバヘッドを測定するために、これらの処理を延々と実行する。

実験を行った結果、表 5.1 で示す値が測定された。(1) と (2) をそれぞれ比較してみると、どの値も大差ないことから進捗状況の監視と判定によるオーバヘッドはほとんどないことがいえる。図 5.4 と図 5.5 は (1) と (2) のデータサイズの分布をそれぞれ表すが、この 2 つの図を見比べても大差ないことがわかる。それに対し (3) の環境下での値を比較してみても、(1) の場合との差

表 5.1: カーネルバッファ内のデータサイズの測定結果

	平均値 (byte)	標準偏差 (byte)	最小値 (byte)
(1) 通常のシステム	101220	9964	83200
(2) 進捗状況の監視と判定のみ行う Tanma	100880	9726	82496
(3) フェードアウト/インを続ける Tanma	100734	9790	82112

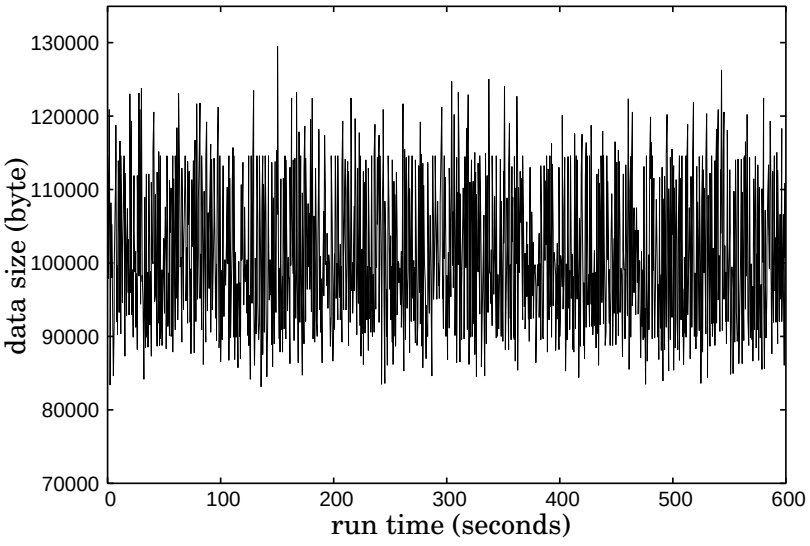


図 5.4: (1) 通常のシステム上でのカーネルバッファ内のデータサイズ

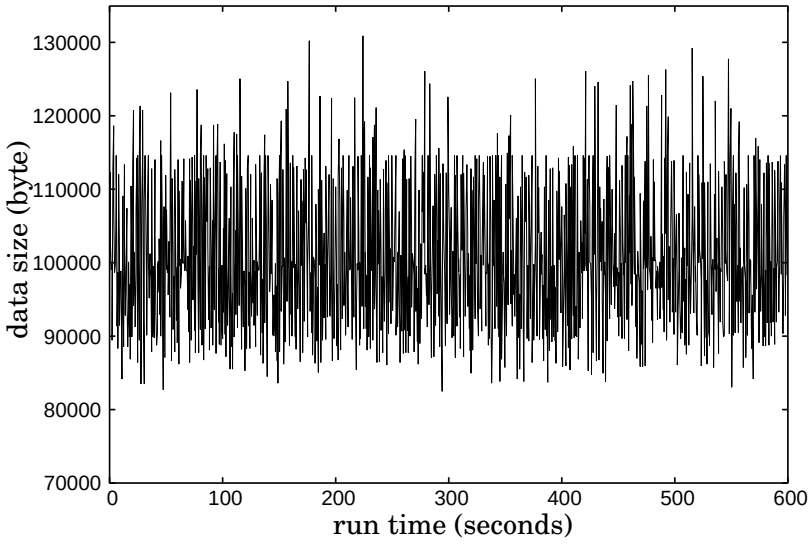


図 5.5: (2) 進捗状況の監視と判定のみ行う Tanma 上でのカーネルバッファ内のデータサイズ

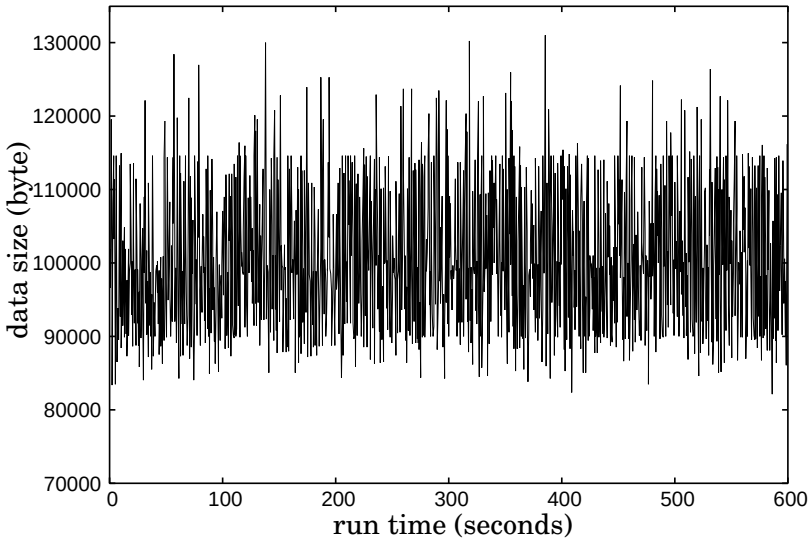


図 5.6: (3) フェードアウト/インを繰り返す Tanma 上でのカーネルバッファ内のデータサイズ

はほとんどない。図 5.6 は (3) の環境下でのデータサイズの分布を示すが、この図をみても、(1) の値とほぼ同じであることがわかる。データサイズの最小値も (1)(2)(3) 共に大差ない値となったので、Tanma のオーバヘッドが原因で長い時間遅れてしまうことはない。このことから Tanma によるオーバヘッドは問題視するほど大きいものではない。

5.4　停止時間の選考

Tanma の重要性の低いプロセスを停止させる時間を決めるために、オーバヘッド測定とユーザアンケートをおこなった。ユーザ側からみると、資源競合が生じている限り重要性の低いプロセスは停止を続ける。実際は、設定された停止時間まで停止をし、時間がくれば進捗状況を調べるために短い時間音楽を再開させ、進捗が思わしくない限り再び一定時間停止を続ける。停止時間が短ければ短いほど、資源競合が解消されればすぐさま音楽を再開することができる。逆に停止時間が長ければ、重要性の高いプロセスが終了してもなかなか音楽が再開されず、ユーザを苛立たせ不安にさせてしまう。しかし、停止時間が短いと、停止後の進捗状況を調べるために音楽を再開する回数が増えてしまい、処理オーバヘッドが増え、タイムマシン再生に必要なバッファサイズが大きくなってしまう。このようなユーザの感性と処理能力の問題があるので、それぞれのバランスを考えた停止時間に設定する必要がある。

5.4.1　停止時間によるオーバヘッドの差

最初に、停止時間の長さの違いによるオーバヘッドの違いを測定した。停止時間別に Tanma 上で音楽プレイヤーと重要性の高いプロセスを同時に実行し、重要性の高いプロセスの処理時間を測定した。停止時間が短ければ短いほど進捗状況を調べるために音楽を再生する時間が長くなるので、この時間に比例してオーバヘッドが増加することが予測される。停止時間は 10 秒から 1 秒までの 1 秒おきと、0.5 秒、0 秒の 12 通りの秒数を用意し、これらの時間に設定された Tanma を使用した場合での重要性の高いプロセスの処理時間を比較した。

重要性の高い処理がπ計算の場合

重要性の高いプロセスがπ計算であったときの実験結果について示す。πの値を 1,000,000 桁まで計算する時間を計測した。π計算のみ実行した場合、処理時間は 320 秒かかった。音楽プレイヤーを動かしながらπ計算を実行した場合、処理時間は 371 秒かかり 17%の遅れが生じた。π計算を nice コマンドにより優先させて実行した場合、処理時間は 350 秒かかり、まだ 9%の遅

延が残った。

図 5.7（a）は、π計算の処理時間と一時的な音楽再生の総時間の関係を表す。図の折れ線は平均値、最大値、最小値を示している。図の左から、停止時間が 10 秒から 0 秒のときの処理時間がそれぞれプロットされている。この図をみると、一時的な音楽再生の総時間に比例してπ計算の処理時間が増加していることがわかる。停止時間が 10 秒から 2 秒では一時的な音楽再生の総時間の差はそれほどなく、π計算の処理時間は 325 秒から 330 秒におさまり、オーバヘッドは 1.6%から 3.1%の遅れとなった。停止時間が 1 秒のときは一時的な音楽再生が長く、そのためオーバヘッドは 334 秒となり、停止時間 10 秒から 2 秒の倍ほどの値となった。停止時間が 0 秒のときは Tanma を使用しない場合とほぼ同じ値となった。

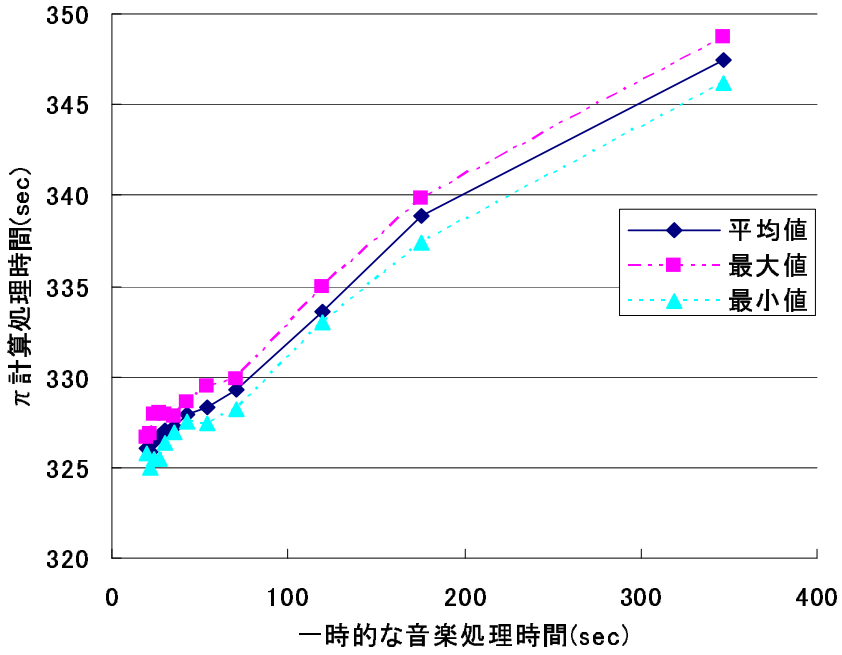
図 5.7（b）は停止時間ごとのπ計算の処理時間の分布を表す箱ひげ図である。図中の pi はπ計算のみ実行した場合、unreg は音楽プレイヤーを動かしながらπ計算を実行した場合、cpu はπ計算を nice コマンドにより優先させて実行した場合の処理時間をそれぞれ示している。t10 から t0 は停止時間が 10 から 0 に設定されている Tanma を使用した場合の処理時間を示している。今回用いたπ計算は一定の資源を使い続ける処理なので、どの場合でもまとまった分布となった。（a）と同様にこの図からも、一時的な音楽再生の総時間が増加にともないπ計算の処理時間も増加していることがわかる。

重要性の高い処理が MP3 エンコードの場合

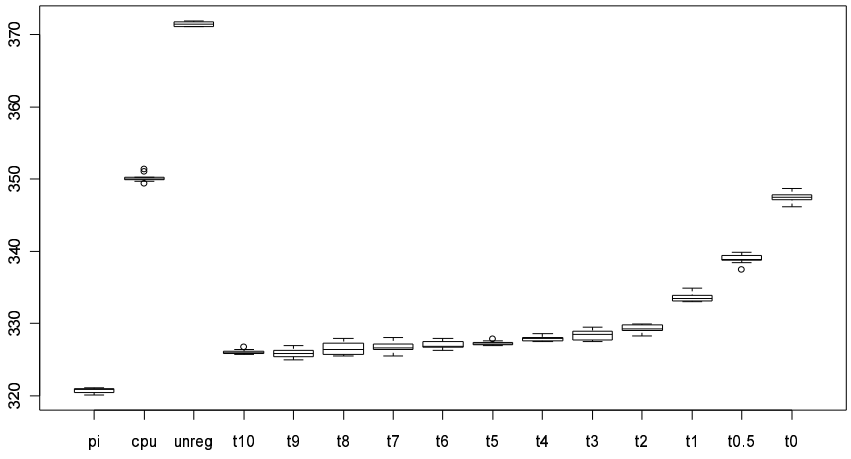
次に、重要性の高いプロセスが MP3 エンコードであったときの実験結果について示す。WAV ファイルのデータサイズ 580Mbyte のときの MP3 エンコード処理時間を測定した。エンコードのみ実行した場合、処理時間は 317 秒かかった。音楽プレイヤーを動かしながらエンコードを実行した場合、処理時間は 386 秒かかり 22%の遅れが生じた。エンコードを nice コマンドにより優先させて実行した場合、処理時間は 348 秒かかり、まだ 9.8%の遅延が残った。

図 5.8（a）は、エンコードの処理時間と一時的な音楽再生の総時間の関係を表す。エンコードの場合は処理時間にばらつきがみられ、π計算の直線のグラフとは異なり、いびつな形となった。しかし平均値に関してみてみると、π計算の場合と同様に停止時間が 10 秒から 2 秒では処理時間にほとんど差はなく、327 秒から 331 秒におさまり、3.1%から 4.4%との遅れとなった。

図 5.8（b）は停止時間ごとのπ計算の処理時間の分布を表す箱ひげ図である。図中の mp3 は MP3 エンコードのみ実行した場合の処理時間を示す。この図をみてわかるが、MP3 エンコードの処理時間にはもともとばらつきがあり、そのため停止時間が 3 秒以上での処理時間の差はエンコードの処理時間のばらつきにより、埋まってしまっている。しかし全体的な分布をみてみる



(a) 音楽再開時間あたりのπ計算処理時間



(b) π計算処理時間の分布

図 5.7: 停止時間ごとのπ計算時間

と、停止時間が増加するほどそれに伴いエンコードの処理時間が増加していることがわかる。

重要性の高い処理が画像処理の場合

最後に、重要性の高い処理が画像処理であったときの実験結果について示す。画像ファイルの枚数が 70 枚で総データサイズが 33Mbyte のときの画像処理時間を計測した。画像処理のみ実行した場合、処理時間は 213 秒かかった。音楽プレイヤーを動かしながら画像処理を実行した場合、処理時間は 245 秒かかり 15%の遅れが生じた。画像処理を nice コマンドにより優先させて実行した場合、処理時間は 232 秒かかり、まだ 8.9%の遅延が残った。

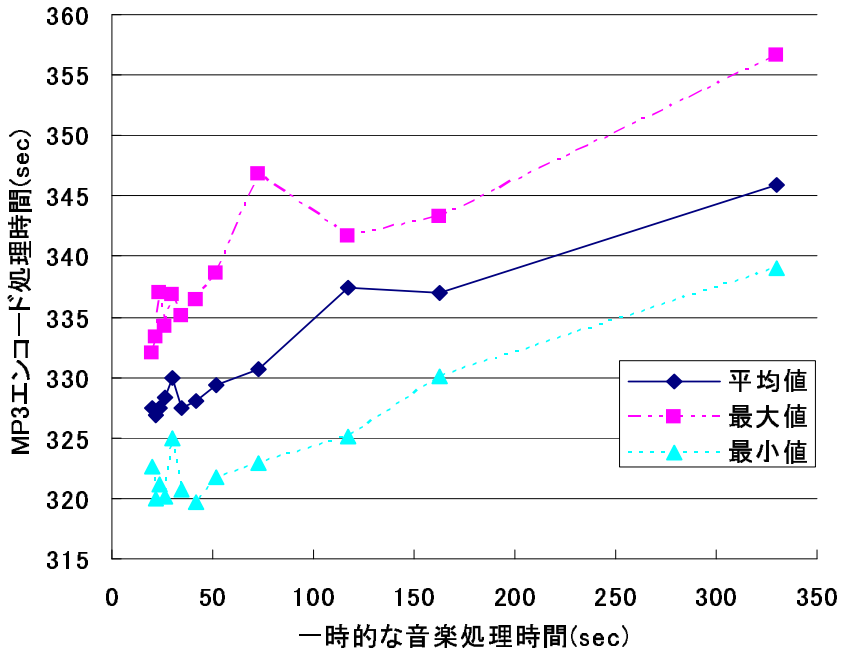
図 5.9（a）は、画像処理の処理時間と一時的な音楽再生の総時間の関係を表す。エンコードの場合以上に処理時間にばらつきがでるものとなった。画像処理の場合は進捗状況の変化が激しく、制御される場合とされない場合があったため、このような結果となった。よって一時的な処理再開の総時間にもばらつきがみられたので、(a）から比例関係を読み取るのは難しい。しかし平均値に関してみてみると、他の処理と同様に停止時間が 10 秒から 2 秒では処理時間にほとんど差はなく、225 秒から 229 秒におさまり、5.6%から 7.5%との遅れとなった。

図 5.9（b）は停止時間ごとのπ計算の処理時間の分布を表す箱ひげ図である。図中の jpeg は画像処理のみ実行した場合の処理時間を示している。画像処理の場合は各停止時間での分布にはばらつきがあるものの、全体的な分布は停止時間が 2 秒以上ではほとんど変わらないものとなった。

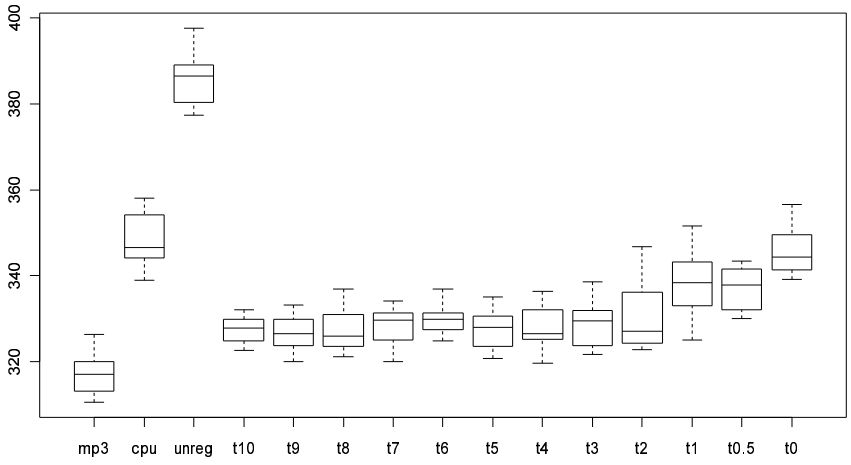
オーバヘッドの差に関するまとめ

各重要性の高い処理に関して、音楽の停止時間の違いによるオーバヘッドの差をみてきた。重要性の高い処理がπ計算の場合は、一時的な音楽処理時間に比例したオーバヘッドが測定された。2 秒以上の停止時間ではその差はほとんどないものとなった。重要性の高い処理が MP3 エンコード処理の場合では、もともとの処理時間のばらつきが影響し、3 秒以上の停止時間ではオーバヘッドは違いのないものとなった。平均値に関していえば、停止時間が 2 秒の場合でも 3 秒以上の場合とほぼ同じ処理時間となった。重要性の高い処理が画像処理の場合は、MP3 エンコード処理の場合以上に処理時間のばらつきがみられた。しかし平均値は 2 秒以上の停止時間では大差ないものとなった。

以上の結果から、音楽を停止させる時間の違いによるオーバヘッドの差は、停止時間が 3 秒以上の場合はほぼ変わらないことがいえる。また、処理時間の平均値に関していえば、停止時間が 2 秒以上の場合はほぼ同じ時間で処理

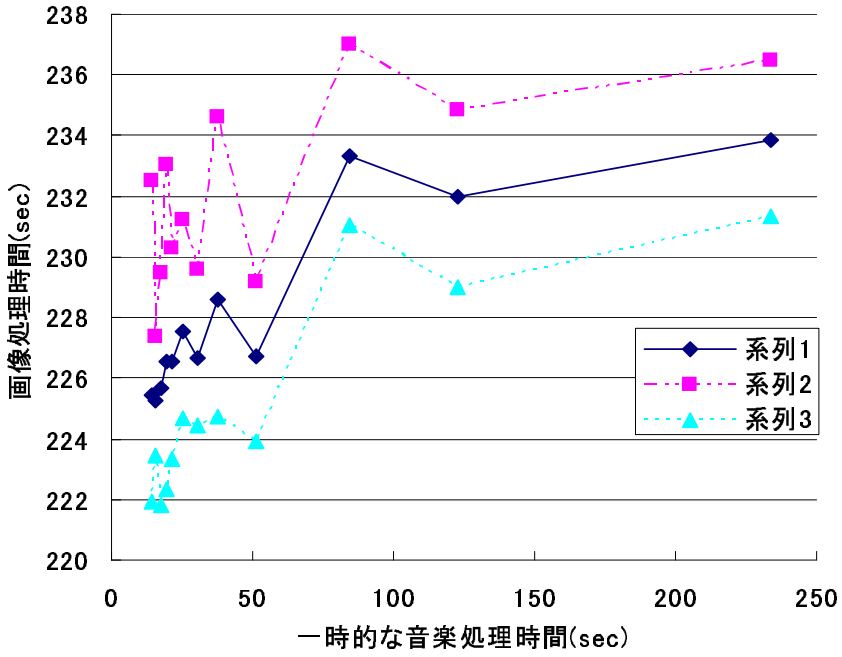


(a) 音楽再開時間あたりの MP3 エンコード処理時間

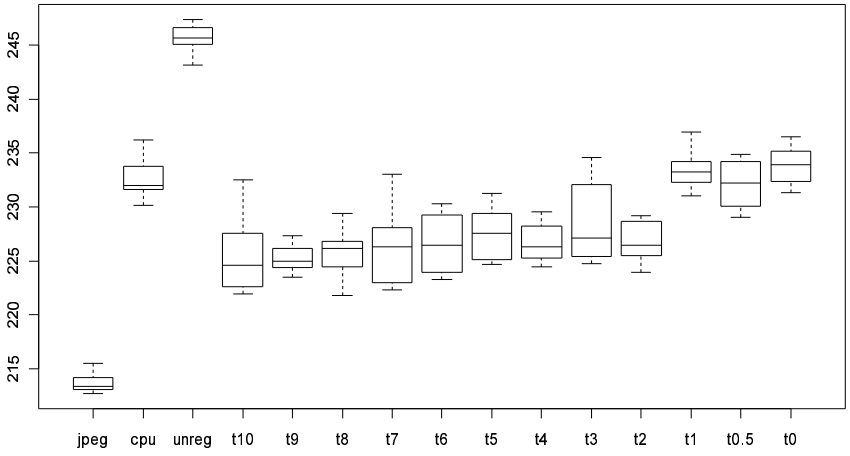


(b) MP3 エンコード処理時間の分布

図 5.8: 停止時間ごとの MP3 エンコード時間



(a) 音楽再開時間あたりの画像処理時間



(b) 画像処理時間の分布

図 5.9: 停止時間ごとの画像処理時間

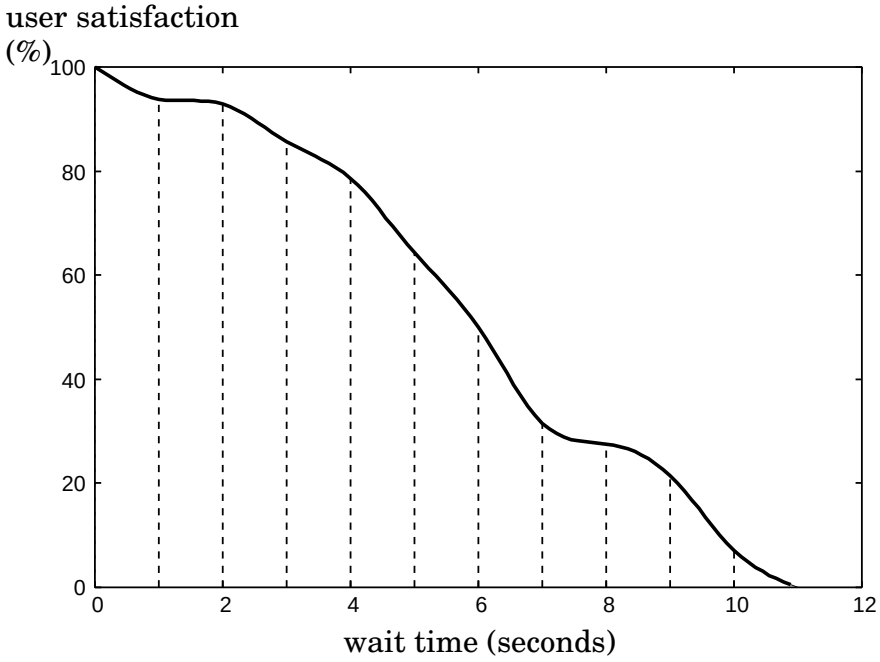


図 5.10: 待ち時間を変えたときのユーザの満足度

を行うことが可能である。

5.4.2　音楽の再開待ち時間に対するユーザの感性

次に、ユーザの感性を調べるアンケートを行った。重要性の高い処理が終了してから音楽が再開されるまで、何秒間待たされても許容できるかを調べた。重要背の高い処理にはπ計算を用いた。被験者にπ計算の終了から音楽の再開まで、1 秒から 10 秒の間だけ待たされる状況を体験してもらい、音楽の再開までの待ち時間が長くて不快に感じるかをそれぞれ調べてみた。

図 5.11 のような結果となった。待ち時間が 1 秒あるいは 2 秒の場合、95%のユーザが問題ない時間だと答えた。待ち時間が 4 秒あたりから長くなるにつれて、許容できると答えたユーザは大きく減っていった。待ち時間が 11 秒以上かかる場合は、全てのユーザが待たされると不快に感じると答えた。

まとめ

5.4.1 節の結果も考慮し、停止時間のデフォルト値を 3 秒に設定した。3 秒の停止時間のオーバヘッドに関しては、5.4.1 節の結果から停止時間が 2 秒以上ではオーバヘッドの差はほとんど生じないことがわかった。停止時間に関す

るアンケートの結果では、85%のユーザが 3 秒の間待たされても許容できると答えた。Tanma にはこのデフォルト値をユーザが変更できるインタフェースが用意されているので、この値は自由に変更することが可能である。

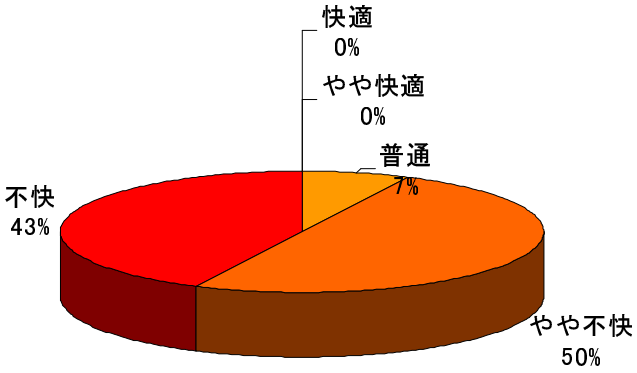
5.5　感性を考慮した QoS に対するアンケート

Tanma による音楽プレイヤーの制御に関して、ユーザがどのように感じるかを調べるためにユーザアンケートを行った。Tanma は音の途切れのようなユーザへの不快感を抑えるために、音楽のフェードアウト／インやタイムマシン再生などのユーザの感性を考慮した QoS 処理を提供している。これらの QoS 処理が実際にユーザへの不快感を抑えることができているのかを調べた。

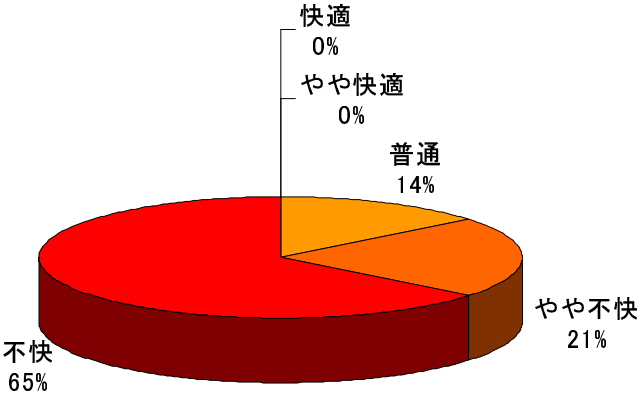
アンケートは Tanma と従来のスケジューリング手法を比較してもらい、それぞれの手法に対する不快度を答えてもらった。音楽プレイヤーを制御するスケジューリング手法として、(1) 従来の Linux スケジューラ、(2) MS Manners、(3) Tanma の使用の 3 つの手法を用意した。(1) は従来の Linux スケジューラを用いる。この手法を用いた場合、資源の競合が激しいと、音の途切れが生じてしまう。(2) は MS Manners[2] が用いている progress-based regulation を使用する。この手法を音楽プレイヤーに用いると、資源の競合が探知された場合は即座に音楽プレイヤーは停止させられる。一定時間停止したあと、MS Manners は進捗状況を調べるために一時的に音楽プレイヤーを再開させ、まだ資源の競合がみられる場合は音楽プレイヤーを倍の時間だけ停止させ、資源の競合が解消された場合は音楽プレイヤーをそのまま実行し続ける。(3) は Tanma を用いる。Tanma は資源の競合が探知された場合は音楽プレイヤーをフェードアウトさせながら停止される。MS Manners とは異なり停止時間は 2 秒に固定した。一定時間停止したあと、MS Manners と同様に音楽プレイヤーを一時的に再開させるが、このときの音量は 0 に設定されている。資源の競合が解消された場合は音楽プレイヤーをフェードインさせながら再開させる。

各手法を用いて、音楽を再生している途中で nice コマンドにより CPU 使用を優先させた重要性の高い処理を同時に実行し、重要性の高い処理が終了するまでの音楽プレイヤーの挙動に関して、不快感を覚えたかを被験者が評価する。不快に感じるから快適に感じるまでの 5 段階でそれぞれ評価される。音楽プレイヤーには realplayer を、重要性の高い処理にはπ計算アプリケーション pi_fftc を用いた。12,3 秒間処理が必要な、50,000 桁のπ計算を実行した。

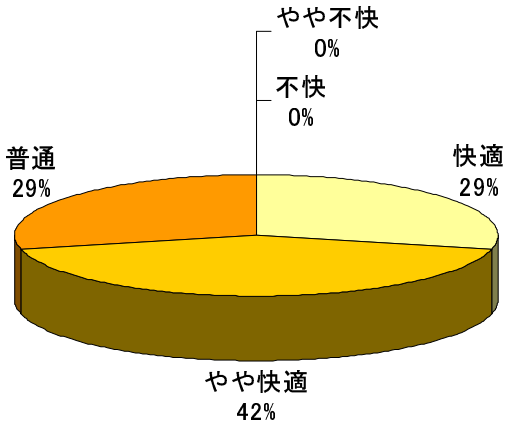
アンケートの結果は図 5.11 のようになった。(1) 従来の Linux スケジューラや (2) MS Manners に対し、(3) Tanma が高く評価されていることがわかる。70%以上の被験者がやや快適あるいは快適であると評価した。また、やや不快あるいは不快であると評価した被験者はみられなかった。Tanma が



(a) 従来の Linux スケジューラ



(b) MS Manners



(c) Tanma

図 5.11: 音楽プレイヤーに対する不快度に関するアンケート結果

提供する音楽のフェードアウト/インなどの感性を考慮した QoS 処理により、ユーザに与える不快感が抑えられていることがいえる。

（2）MS Manners と（3）Tanma は共に progress-based regulation をもとにスケジューリングを行っているが、その評価は大きく異なるものとなった。この原因は、progress-based regulation をそのままマルチメディア処理に用いることはできないことにある。progress-based regulation による音楽プレイヤーの突然の停止および再開は、ユーザに大きな不快感を与えてしまう。この影響は音の途切れ以上に問題であることが（1）と（2）を比較することでわかる。（１）では不快に感じた被験者が 45%であったのに対し、（2）では 65%の被験者が不快に感じている。このような progress-based regulation を、我々の Tanma はユーザの感性を考慮する QoS 処理を加えることでマルチメディア処理に適用できるようにした。

意見・感想

アンケートに伴い、Tanma を使用した感想や意見をいただいたので紹介する。

- Tanma で音楽が止まる瞬間に途切れるのが少し不快に感じる。
- 重要な処理が終了したのをみた時にはすぐに音楽が再開されてほしい。
- 再開の際にフェードインで始まるのが快適であった。
- 音が途切れるよりはフェードアウトして止まるほうが快適。
- 音の途切れより MS Mannners によるたまに再生のほうがかなり不快。
- フェードアウトは耳に心地好かった。
- 重要な処理が終了したあとの待ち時間は多少なら何回も聞くと慣れて気にならないかも。
- 本当に音楽が消える瞬間が少し気になる。
- 重要性の高い処理によって、重要な処理が終了したあとの待ち時間の感じ方は変わる。
- 作業中という前提ならば、Tanma の制御は許容できる。他の手法は許容できない。
- 重要な処理終了後の待ち時間は、極端に長くならない限り、その違いはわからない。
- 音楽が切れるのがわかると不快なので、近い音が出し続けられているほうがいい。

- 音楽が停止しないで音量が小さいままのほうが、個人的にはよい。
- 重要な処理終了後の待ち時間のイメージはどのくらいの長さでもあまり変わらないように聞こえた。

多かった意見感想は、「重要な処理終了後の待ち時間はあまり気にならない」「音楽が停止する様子が聞こえてしまうと不快に感じる」とうものである。音量 0 で音楽を再生させるのと音楽を停止させることの違いを感じるユーザが何人かおり、その点が気になるようである。しかし、Tanma には重要性の高いプロセスをすばやく処理するという目的があるので、停止させずに音量 0 のまま音楽再生を行うと資源が競合したままとなってしまう。重要性の高い処理が遅れても問題はなく、音が途切れるのだけが問題であるというユーザにはこの手法が有効と考えられる。この手法を使用し、重要性の高い処理をすばやく処理したい場合は、Tanma により音量が小さく設定されているときにユーザが手動で止めればよい。

第6章　まとめ

本稿では、重要性の低いマルチメディア処理に対し QoS 処理を行うジョブスケジューリングシステム Tanma について述べた。重要性の低いマルチメディア処理の資源を急激に奪うことでユーザに不快感を与えないよう、ユーザの感性を考慮した新しい QoS 処理を行う。Tanma では progress-based regulation をマルチメディア処理に適応させ、再生中の音楽のフェードアウト/インなどの処理を行うことで、ユーザに不快感を与えることなく音楽再生プロセスをすばやく制御することができる。我々が行った実験の結果、Tanma を使うことにより音楽再生プロセスと重要性の高いプロセスとの資源の競合を制御することができた。

今回実装した Tanma は音楽再生処理を対象としているが、動画再生処理に対して QoS 処理を行うのは難しい。動画再生処理の場合、動画像のフレームレートと実際に処理されたフレーム数を比較することで、進捗状況を把握することが可能だと考えられる。しかし、動画再生処理は音楽再生処理よりも複雑なものであり、カーネルレベルでフェードアウトなどの感性を考慮した QoS 処理を行うことは困難である。

また、Tanma はネットワーク上のサウンドデータの再生処理を制御するにあたって問題が生じる。カーネル内のサウンドバッファを監視するだけでは進捗状況を把握することができない。アプリケーションがネットワークからサウンドデータを取得した後、ある程度バッファリングを行ってからカーネルにデータ転送を行うからである。アプリケーションはある程度のサウンドデータを溜めることで、音楽が途切れることを防いでいる。しかし、カーネル側はこのようなバッファリングを行っているのか、負荷により遅れが生じているのかを判断することができない。この問題に対処するには、ネットワーク流量を監視することで進捗状況を把握する方法が挙げられる。また、アプリケーションバッファを監視することでも対処が可能だと考えられる。

これらの機能は、それぞれの状況に特化したものであるが、Tanma をより適応させやすくさせるには、汎用的な機能が必要となる。音楽処理、動画像処理、ネットワークを用いたマルチメディア処理に対し、Tanma が共通の機能を提供できるようにすることが今後の課題である。

参考文献

[1] Barabanov, M. and Yodaiken, V.: Reaal-time linux, *Linux journal* (1997).

[2] Douceur, J. R. and Bolosky, W. J.: Progress-based regulation of low-importance processes, *In Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pp. 247–260 (1999).

[3] Goel, A., Abeni, L., Krasic, C., Snow, J. and Walpole, J.: Supporting Time-Sensitive applications on a Commodity OS, *In Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation* (2002).

[4] Goel, A., Shor, M. H., Walpole, J., Steere, D. and Pu, C.: Using Feedback Control for a Network and CPU Resource Management Application, *In Proceedings of 2001 American Control Conference* (2001).

[5] Ingram, D.: Integrated Quality of Service Management, *University of Cambridge Computer Laboratory Technical Report*, No. 501 (2000).

[6] Jacobs, S. and Eleftheriadis, A.: Streaming Video using Dynamic Rate Shaping and TCP Congestion Control, *Journal of Visual Communication and Image Representation* (1998).

[7] Krasic, C. and Walpole, J.: QoS Scalability for Streamed Media Delivery, *CSE Technical Report CSE-99-011* (1999).

[8] Lu, C., Abdelzaher, T. F., Stankovic, J. A. and Son, S. H.: A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers, *In Proceedings of IEEE Real-Time Technology and Applications Symposium*, pp. 51–62 (2001).

[9] Oikawa, S. and Rajkumar, R.: Portable RK: A Portable Resource Kernel for Guaranteed and Enforced, *In Proceedings of IEEE Real Time Technology and Applications Symposium*, pp. 111–120 (1999).

[10] Ooura, T.: 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`.

[11] Rajkumar, R., Lee, C., Lehoczky, J. and Siewiorek, D.: A Resource Allocation Model for QoS Management, *In Proceedings of IEEE Real-Time Systems Symposium*, pp. 298–307 (1997).

[12] Rajkumar, R., Juvva, K., Molano, A. and Oikawa, S.: Resource Kernels:A Resource-Centric Approach to Real-Time and Multimedia Systems, *In Proceedings of SPIE/ACM Conference on Multimedia Computing and Networking*, pp. 150–164 (1998).

[13] Real.com: RealNetworks, `http://www.jp.real.com/`.

[14] Steere, D. C., Goel, A., Gruenberg, J., McNamee, D., Pu, C. and Walpole, J.: A Feedback-driven Proportion Allocator for Real-Rate Scheduling, *In Proceedings of 3rd Symposium on Operating Systems Design and Implementation* (1999).

[15] Verghese, B., Gupta, A. and Rosenblum, M.: Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors, *In Proceedings of the 8th ASPLOS*, pp. 181–192 (1998).

[16] ＰＥＮ@海猫: @MARINECAT, `http://www.kurims.kyoto-u.ac.jp/~ooura/pi\_fft-j.html`.

[17] Jeff Tranter 著 山形 浩生訳: Linux マルチメディアガイド, O'Reilly Japan (1997).