ソフトリアルタイムシステムにおけるCPU使用率の向上を目指したDEADLINEスケジューラの拡張

山本 武蔵　岩崎 英哉

Linux カーネルのスケジューラのひとつに，DEADLINE スケジューラがある．プロセスに DEADLINE スケジューラを設定する際，カーネルは当該プロセスを含め DEADLINE スケジューラが設定されている全てのプロセスの CPU の使用率の和を計算する．その和がコア数を超えるような場合，そのプロセスに対しては DEADLINE スケジューラを設定しない．しかし，これにより CPU に余裕がある状態でも，プロセスに DEADLINE スケジューラを設定できないことがあり，ソフトリアルタイムシステムに対しては厳しすぎる条件となっている．本研究では，全てのプロセスの実行時間をランタイムより小さくすることで，使用率の和がコア数を超える場合でも，要求したプロセスに対して新たに DEADLINE スケジューラを設定できる手法を提案する．提案する手法を，シミュレーションプログラムとして実装し，その実用性を評価を行った．さらに，Linux カーネルへの実装についても考察した．

1　はじめに

Linux カーネルのスケジューラのひとつに，リアルタイムスケジューリングを行う DEADLINE スケジューラ (SCHED_DEADLINE) [1] [5] がある．DEADLINE スケジューラは，マルチメディアサービスや制御アプリケーションに適しており [1]，様々なリアルタイムシステムに用いられる．特に，自動車のエアバック制御システムやエンジン制御システムなど，システムに与えられたタスクが決められた刻限 (デッドライン) までに終了しなかった，すなわちデッドラインミスが起こったとき，致命的な影響を与えるハードリアルタイムシステムに用いられる．ハードリアルタイムシステムに対し，銀行の ATM やオンラインゲームなど，デッドラインミスが起こってもシステムの質は落ちるものの致命的とはいえないシステムを，ソフトリアルタイムシステムという．

DEADLINE スケジューラは，Earliest Deadline First (EDF) [6] アルゴリズムと Constant Bandwidth Server (CBS) [3] アルゴリズムによって実装されている．EDF アルゴリズムは，プロセスに設定されたデッドラインが最も近いプロセスから優先的に実行する．また，CBS アルゴリズムは，プロセスにデッドラインを割り当て，各プロセスが各ピリオドごとに最大でもランタイム分だけ実行されるようにし，異なるプロセス間の干渉を回避する．

ユーザは，指定したプロセスに DEADLINE スケジューラを設定する際，パラメータとして period (期間，ピリオド)， runtime (実行時間，ランタイム)，deadline (期限，デッドライン) を与える．これらのパラメータに基づき，カーネルはそのプロセスに可能であれば DEADLINE スケジューラを設定する．DEADLINE スケジューラは，各期間について，デッドラインまでにランタイム分の時間を実行することを保証する．

カーネルは，CPU に余裕がなくなりデッドラインの保証ができないプロセスに対しては DEADLINE スケジューラを設定しない．設定の可否は，次の計算により判定する．各プロセスについて，runtime/period を，そのプロセスの使用率という．DEADLINE ス

Extension of DEADLINE Scheduler to Improve CPU Utilization in Soft Real-Time Systems.

Musashi Yamamoto, 電気通信大学大学院情報理工学研究科, Dept. of Information and Computer Science, The University of Electro-Communications.

Hideya Iwasaki, 明治大学理工学部 情報科学科, Dept. of Computer Science, Meiji University.

ケジューラが既に設定されている全てのプロセスとDEADLINE スケジューラを設定しようとしているプロセスの使用率の和を計算する．この値がコア数以下であり，CPU に設定できる余裕があれば，DEADLINE スケジューラを設定し，余裕がなければ設定しない．その結果，ユーザは，CPU に余裕がなくなるようなプロセスに DEADLINE スケジューラは設定できない．これはハードリアルタイムシステムにおいては重要な条件である．ところがこれは，ソフトリアルタイムシステムでは厳し過ぎる．

そこで本研究では，ソフトリアルタイムシステムを想定し，DEADLINE スケジューラが設定されているプロセスの実行時間をランタイムより少しずつ小さくすることで，使用率が大きく，CPU に余裕がなくなるようなプロセスにも DEADLINE スケジューラを設定できるようにする手法を提案する．

2 Linux のスケジューラ

2.1 DEADLINE スケジューラ

スケジューラは，CPU 上で実行すべきプロセスが複数あるとき，プロセスの実行順を決定する．DEADLINE スケジューラは，Linux カーネル 3.14 から導入された SCHED_DEADLINE ポリシーに基づくリアルタイム CPU スケジューリングを行うスケジューラである．

DEADLINE スケジューラは，プロセスごとに定められた次の 3 つのパラメータに基づいてスケジューリングを行う．

- period (周期，ピリオド)
- runtime (実行時間，ランタイム)
- deadline (期限，デッドライン)

以後，プロセスの総数を N，i 番目のプロセスを p_i，p_i のピリオドを t_i，ランタイムを r_i，デッドラインを d_i とする．また全プロセスの使用率 r_i/t_i の合計 $\sum_{i=1}^{n} r_i/t_i$ を U とする．

プロセスはデッドラインが最も近いものから順に実行される．プロセスは，ピリオドごとに，デッドラインまでにランタイム分の時間を実行する必要があり，デッドラインを過ぎると次のピリオドまでそのプロセスを実行することができない．これらのアルゴリズムは Earliest Deadline First (EDF) アルゴリズムと Constant Bandwidth Server (CBS) アルゴリズムによって構成されている．

全てのプロセスのピリオドの値が等しく，同じ時刻に開始しているとき，ピリオドが同期している (同期ピリオド) という．また，ピリオドの値が異なっている，または同じ時刻に開始していないとき，ピリオドが同期していない (非同期ピリオド) という．

2.2 EDF と CBS

Earliest Deadline First (EDF) は，Linux カーネル内のデッドラインが最も近いプロセスをスケジューリングするアルゴリズムである．DEADLINE スケジューラの実装では，プロセスは赤黒木という平衡二分木によって保持されている．

Constant Bandwidth Server (CBS) は，プロセス間の時間的分離を提供する資源予約アルゴリズムである．CBS は，プロセス i について，2 つの状態変数 R_i と D_i を持つ．R_i は現在のピリオドで使用できる残りのランタイム，D_i は動的優先度を割り当てるために使用されるスケジューリング期限を表す．プロセスが時刻 T に生成されると，そのプロセスが実行可能か判定し，実行不可能であれば新しいスケジューリング期限を設定 ($D_i = T + t_i$) し，ランタイムを再計算する．プロセスが実行されると，実行された分だけ R_i が減少し，R_i が 0 になると，プロセスはスケジューラによって実行のために選択されなくなる．プロセスが選択されるようになるのは時刻 $T = D_i$ においてランタイムの再計算 ($R_i = r_i$) とスケジューリング期限の延期 ($D_i = D_i + t_i$) が行われ，優先度が下がったときである．

このように，DEADLINE スケジューラは，CBS によってプロセスにデッドラインやランタイムを割り当て，EDF によってどのプロセスから優先的にスケジューリングするかを決定する．

2.3 実行例

表 1 のようなパラメータを持つプロセスを考える．

2 つのプロセスが同時に生成されたとき (同期ピリオド)，図 1 のように実行される．

表 1: DEADLINE スケジューラのパラメータ

	r_i (ms)	d_i (ms)	t_i (ms)
p_1	5	6	10
p_2	3	9	10

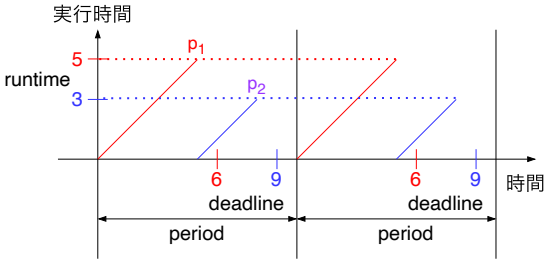

図 1: 同期ピリオドでの実行例

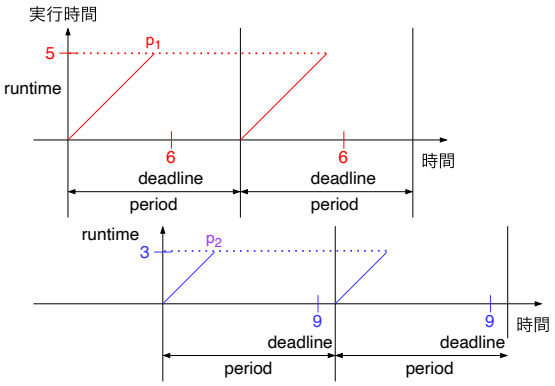

図 2: 非同期ピリオドでの実行例

はじめのピリオドでは，デッドラインが近い p_1 から先にランタイム分の時間 5ms 実行される．p_1 の実行が終了すると，そのピリオドでは p_1 は実行できないため，次にデッドラインが近い p_2 が実行される．p_1 と同様に p_2 はランタイム分の時間 3ms 実行された後は，そのピリオドでは実行されない．時刻がそれぞれのプロセスのデッドラインになったとき，ランタイムが再計算され，デッドラインが延期される．そして，次のピリオドでも同様に p_1, p_2 の順で実行される．

しかし，現実には全てのプロセスのピリオドが同期していることはほとんどなく，図 2 のようにプロセスは異なる時刻に生成され，ピリオドが同期せずに実行される (非同期ピリオド)．最初に p_1 が生成，実行される．その後，p_2 が生成，実行される．同期ピリオドと同様に，時刻が各プロセスのデッドラインになったとき，ランタイムとデッドラインが再計算される．p_2 の実行が終了したあと，p_1 は次のピリオドが始まるまで実行されない．

```
1  struct sched_attr {
2    __u32 size;
3    __u32 sched_policy;
4  ...
5    /* SCHED_DEADLINE */
6    __u64 sched_runtime;
7    __u64 sched_deadline;
8    __u64 sched_period;
9  ...
10 };
```

図 3: sched_attr 構造体の定義

2.4 スケジューラの設定

プロセスに対するスケジューラの設定やパラメータの変更を行うには，sched_setattr システムコールを用いる．これは，次のように，プロセス番号，設定するスケジューラとパラメータを表す sched_attr 構造体，フラグを引数に取る．

```
int sched_setattr(
  pid_t pid,
  const struct sched_attr *attr,
  unsigned int flags);
```

sched_attr 構造体の定義を図 3 に示す．DEADLINE スケジューラを設定する場合，sched_policy に SCHED_DEADLINE ポリシーの番号である定数 SCHED_DEADLINE を設定し，sched_runtime, sched_deadline, sched_period に設定したい値を指定する．それぞれ単位はナノ秒である．フラグはこのシステムコールの将来のために用意されており，現在は 0 を指定する．sched_policy に他のスケジューリングポリシーを指定することにより，sched_setattr システムコールを用いて，他のスケジューラを設定することもできる．DEADLINE スケジューラを設定したり，DEADLINE スケジューラのパラメータを変更したりする際は，この sched_setattr システムコールを用いる．

sched_setattr システムコールを使用して DEADLINE スケジューラを設定する際やパラメータを変更

表 2: 使用率の和が 1 を超えるプロセス

	r_i (ms)	d_i (ms)	t_i (ms)
p_1	5	6	10
p_2	7	9	10

する際に，明らかにデッドラインを超過するようなプロセスをなくすため，カーネルはスケジューラの設定やパラメータの変更が可能か否かを，次のような計算を行い判断する．最初に，使用率 $U_i = r_i/t_i$ を求める．次に，求めた使用率と CPU の使用状況から設定や変更が可能か判断する ($\sum U_i \leq$ コア数)．不可能であれば `sched_setattr` システムコールはエラーを返し，DEADLINE スケジューラの設定やパラメータの変更は行わず，元のスケジューラのままとする．

例として表 2 のようなパラメータを持つ 2 つのプロセスを考える．既に p_1 にのみ DEADLINE スケジューラが設定されている状況で，p_2 に DEADLINE スケジューラを設定しようとしたとする．このとき各プロセスの使用率の和を計算すると次のようになる．

$$U_1 = \frac{5}{10},\ U_2 = \frac{7}{10}$$

$$\sum U_i = \frac{5}{10} + \frac{7}{10} = \frac{12}{10} > 1$$

コア数が 1 の場合，使用率の和がコア数を超えてしまうため，p_2 に DEADLINE スケジューラは設定されず，元のスケジューラのままとなる．

2.5 現状の問題点

表 2 の例において，p_2 に DEADLINE スケジューラは設定されないため，DEADLINE スケジューラの全てのプロセスの使用率の和は 5/10 のままである．本来，使用率の和が 1 に近い値まで CPU を使用できるはずであるが，DEADLINE スケジューラを設定したりパラメータを設定する際，パラメータを適切な値にしない限り，設定に失敗し，使用率は低いままとなってしまう．ハードリアルタイムシステムの場合，デッドラインミスは致命的な問題となってしまうため，このような設定の可否判定は重要である．しかし，ソフトリアルタイムシステムの場合，デッドラインミスは致命的とまではいえないため，この判定は厳し過ぎ，CPU の有効利用をかえって損なっているとも考えられる．

3 設計

3.1 概要

本研究では，2.5 節で説明した，ソフトリアルタイムシステムにおいても CPU の現使用状態に応じた適切なパラメータを持つプロセスでなければ DEADLINE スケジューラを設定できないという問題を解決する．そのため，使用率がコア数を超える場合でもある程度の許容範囲内であれば DEADLINE スケジューラを設定できるようにし，使用率を向上させる手法を提案する．

使用率がコア数を超える場合でも DEADLINE スケジューラを設定できるようにするため，全てのプロセスの使用率の和がコア数に近い値になるように，全てのプロセスの実行時間をランタイムより小さく縮小する．その際，次に従うようにプロセス p_i の実行時間 e_i を決定する．p_i の実行時間 e_i をランタイム r_i で割った値を，縮小された実行率 E_i とする．

$$E_i = \frac{e_i}{r_i}$$

CPU を最大活用するため，全てのプロセスをある計算によりいくつかのグループに分け，プロセス全体で最小の実行率を最大化し，同じグループに属す全てのプロセスで実行率が等しくなるように，実行時間を調整する．全てのプロセスを M 個のグループ $G_1, \cdots, G_M$ に分ける，グループ k $(1 \leq k \leq M)$ に属す全てのプロセスの実行率を $E(G_k)$ と書くことにすると，次のようにする．

$$\text{Maximize} \min_{1 \leq k \leq M} E(G_k)$$

この条件は，線形計画問題として表現することができ，線形計画問題を解くことで実行時間を計算することできる．計算方法は，ピリオドが同期しているかどうかで異なる．ピリオドが同期していない場合より，同期している場合の方が簡単であるため，最初に同期している場合について考える．

例として，コア数が 1 でピリオドは同期しているとき，表 3 のプロセスを考える．ここで，p_1，p_2，p_3 には既に DEADLINE スケジューラが設定されており，p_4 にも DEADLINE スケジューラを設定したい

表 3: 提案手法のに基づく実行時間

	r_i (ms)	d_i (ms)	t_i (ms)	e_i (ms)	E_i
p_1	1	1	10	0.5	0.5
p_2	1	1	10	0.5	0.5
p_3	4	10	10	3.6	0.9
p_4	6	10	10	5.4	0.9

ものとする．全プロセスの使用率 U_i の和 U を計算すると

$$U_1 = \frac{1}{10},\ U_2 = \frac{1}{10},\ U_3 = \frac{4}{10},\ U_4 = \frac{6}{10}$$

$$U = \frac{1}{10} + \frac{1}{10} + \frac{4}{10} + \frac{6}{10} = \frac{12}{10} > 1$$

となる．よって現状では，p_4 に対して DEADLINE スケジューラを設定することはできない．そのため，提案機構では，全てのプロセスの実行時間をランタイムより小さく縮小する．この 4 つのプロセスの場合，後述するグループ分けに従うと，p_1 と p_2 のグループと，p_3 と p_4 のグループの，2 つのグループに分けられる．また，後述する実行時間の計算に従い，CPU を最大活用しつつ，プロセス全体で最小の実行率を最大化し，各グループで実行率の偏りをなくすように実行時間を決めると，表 3 のようになる．このように，p_1 と p_2 の実行時間を 0.5ms とすることで，最も小さい実行率の最大化を行い，p_3 と p_4 の実行時間をそれぞれ 3.6ms と 5.4ms にすることで，CPU を最大活用し，実行率の偏りがないようにする．

3.2 同期ピリオドの場合

同期ピリオドでのグループ分けの計算と実行時間の計算は，プロセスが生成され，最初のピリオドが開始したときに行う．ここで簡単のため，コア数は 1 とする．

3.2.1 線形計画問題による実行時間の決定

プロセス全体で最小の実行率を最大化するように実行時間を計算するという条件は，次のような線形計画問題に変換できる．全てのプロセスの個数を N 個とし，プロセスはデッドラインが近い順にソートされ，番号つけされているとする．すなわち，デッドラインが最も近いプロセスが p_1，最も遠いプロセスが p_N である．

表 4: 提案手法のプロセス

	r_i (ms)	d_i (ms)	t_i (ms)	e_i (ms)	E_i
p_1	1	1	10	0.5	0.5
p_2	1	1	10	0.5	0.5
p_3	4	10	10	2	0.5
p_4	6	10	10	3	0.5

$$\begin{aligned} \text{目的関数 Maximize } & \sum_{i=1}^{N} E_i \\ \text{制約条件 } \forall\, i\ (e_i \ &\leq\ r_i \ \wedge \\ \sum_{j=1}^{i} e_j \ &\leq\ d_i \ \wedge \\ E_i \ &=\ \frac{e_i}{r_i}\ ) \end{aligned}$$

プロセス全体で最小の実行率を最大化することが目的関数となる．制約条件は，実行時間がランタイムを超えないようにすること，デッドラインミスが起きないように，実行時間の和がデッドラインを超えないようにすることであり，最後に実行率を定義している．

表 3 のプロセスに対して，線形計画問題を解くソルバを用いて実行時間を求めると，表 4 のような解が得られることがある．このとき，確かに最小の実行率を最大化できているが，全ての実行率が等しくなってしまっている．p_3 と p_4 の実行率は 0.9 にできるはずであるが，そうなっておらず，CPU を最大活用できていない．

そこで，p_1 と p_2 のグループと p_3 と p_4 のグループに分け，実行時間を大きくすることができる p_3 と p_4 のグループに対して線形計画問題を再度解く．これを繰り返すことで，CPU を最大活用しつつ，プロセス全体で最小の実行率を最大化し，同じグループに属す全てのプロセスで実行率が等しくなるように，実行時間を調整する．

グループ分けは，次のように行う．p_1 から実行時間を足し，その値がデッドラインと一致したとき，すなわち，

$$\forall x < i \quad \sum_{j=1}^{i} e_j < d_i \ \text{ かつ } \ \sum_{j=1}^{x} e_j = d_x$$

のとき，p_1 から p_x をひとつのグループとする．その後，p_{x+1} から p_N までに対し，同様の線形計画問題を再度解き，グループ分けを行う．

このようにして線形計画問題を複数回解くことにより，表 3 と同じ結果を得ることができる．しかし，実際に Linux カーネルへの実装を想定すると，数理最適化モジュールを用いて実行時間を求めるのは，現実的ではない．そこで，線形計画問題を段階的に解いた場合と同等の解が求まるようなアルゴリズムを用いることにする．このアルゴリズムでは，最初に，線形計画問題を段階的に解いた場合のグループと同等になるようにグループ分けを行い，その後実行時間を計算する．

3.2.2 実行時間を決定するアルゴリズム

グループ分けの計算

プロセスのグループ分けの計算を次のように行う．まず，p_1 から順にデッドラインをランタイムの和で割った値の比較を行う．

$$\frac{d_i}{\sum_{j=1}^{i} r_j} \geq 1$$

これは，e_i を r_i としても，d_i を超過しないかを判定している．最初にこの式を満たさないプロセスを p_x とする．つまり，

$$\frac{d_x}{\sum_{j=1}^{x} r_j} < 1$$

のとき，p_1 から p_x をひとつのグループとする．その後，p_{x+1} から順に同様に条件式の判定を行うが，既に決められたグループを考慮する必要があるため，条件式は次のように変わる．

$$\frac{d_i - d_x}{\sum_{j=x+1}^{i} r_j} \geq 1$$

デッドラインまでの時間をランタイムの和で割った値を圧縮率という．圧縮率は，実行時間をランタイムより小さくする際の比率を表す．そのため，圧縮率が 1 より大きくなることはなく，p_i から p_j までの圧縮率は次のようになる．ただし，$d_0 = 0$ とする．

$$S_i^j = \min\left\{1,\ \frac{d_j - d_{i-1}}{\sum_{k=i}^{j} r_k}\right\}$$

条件式は，次のようになる．

$$S_i^j \geq 1$$

つまり，p_{x+1} から順に圧縮率 S_{x+1}^j が条件式を満たすか判定する．次に $p_{x'}$ のときに条件式を満たさなかったとき，p_{x+1} から $p_{x'}$ までをひとつのグループとする．このとき，$S_1^x \geq S_{x+1}^{x'}$ であれば，p_1 から p_x までの圧縮率を小さくし，p_{x+1} から $p_{x'}$ までの圧縮率を大きくすることができる．そこで，p_1 から p_x までのグループと p_{x+1} から $p_{x'}$ までのグループを統合し，ひとつのグループとする．グループの統合を行った際，圧縮率を再計算する．このとき，S_1^x は小さくなっている．つまり，

$$S_1^x > S_1^{x'}$$

である．このようにグループの統合が起きた際，デッドラインが近いグループの圧縮率は小さくなるため，その前のグループの圧縮率より小さくなる可能性がある．そのため，前のグループの圧縮率と比較し，小さければそのグループと統合することを繰り返す．

これを p_N まで繰り返し，残りのプロセスをひとつのグループとし，前のグループと圧縮率の比較を行い，グループの統合の必要があるかを判定する．

以上述べた操作を擬似コードに示すと，図 4 のようになる．

例として表 3 のプロセスに対してグループ分けの計算を行う．p_1 から順に圧縮率を計算する．

$$S_1^1 = \frac{d_1 - d_0}{r_1} = \frac{1}{1} = 1 \geq 1$$

$$S_1^2 = \frac{d_1 - d_0}{r_1 + r_2} = \frac{1}{2} < 1$$

最初に条件式を満たさないプロセスは p_2 なので，p_1 と p_2 をひとつのグループにする．このグループの圧縮率は，$S_1^2 = 1/2$ である．前のグループはないため，グループの統合は行われない．次に，p_3 から順に圧縮率を求める．

$$S_3^3 = \frac{d_3 - d_2}{r_3} = \frac{10 - 1}{4} = \frac{9}{4} \geq 1$$

$$S_3^4 = \frac{d_4 - d_2}{r_3 + r_4} = \frac{10 - 1}{4 + 6} = \frac{9}{10} < 1$$

次に条件式を満たさないプロセスは p_4 なので，p_3 と p_4 をひとつのグループにする．このグループの圧縮率は，$S_3^4 = 9/10$ でである．前のグループの圧縮率と比べると，$S_1^2 < S_3^4$ であるため，前のグループと統合せず，p_3 と p_4 をひとつのグループとする．

実行時間の計算

実行時間の計算は次のように行う．プロセス p_i の実行時間を e_i，デッドラインが近いプロセスのグルー

$i \leftarrow 1$
$j \leftarrow 1$
$R \leftarrow 0$
while $j \leq N$ **do**
  $R \leftarrow R + r_j$
  $S_i^j = (d_j - d_{i-1})/R$
  if $S_i^j < 1$ **then**
    i から j までをひとつのグループとする
    ひとつ前のグループと統合を行うか判定
    while グループの統合を行った **do**
      圧縮率を再計算
      ひとつ前のグループと統合を行うか判定
    end while
    $R \leftarrow 0$
    $i \leftarrow j + 1$
  end if
  $j \leftarrow j + 1$
end while
p_i から p_j までをひとつのグループとする
ひとつ前のグループと統合を行うか判定
while グループの統合を行った **do**
  圧縮率を再計算
  ひとつ前のグループと統合を行うか判定
end while

図 4: 同期ピリオドでのグループ分けの擬似コード

プから順に $G_1, G_2, \cdots, G_M$ とし，グループ G_k の圧縮率を $S(G_k)$ とする．プロセスがグループ分けされ，圧縮率が求められているため，実行時間を次のように計算する．

$$e_i = r_i \cdot S(G_k)\ (p_i \in G_k)$$

実行率を計算すると，

$$E_i = \frac{e_i}{r_i} = S(G_k)$$

となり，グループ内で実行率は等しくなる．

例として表 3 のプロセスに対して実行時間を計算する．p_1 と p_2 のグループの縮小率は，$S_1^2 = 1/2$ であり，p_3 と p_4 のグループの縮小率は，$S_3^4 = 9/10$ である．そのため，実行時間は次のようになる．

$$e_1 = 1 \cdot \frac{1}{2} = 0.5,\ e_2 = 1 \cdot \frac{1}{2} = 0.5$$

$$e_3 = 4 \cdot \frac{9}{10} = 3.6,\ e_4 = 6 \cdot \frac{9}{10} = 5.4$$

3.3 非同期ピリオドの場合

同期ピリオドではグループ分けの計算と実行時間の計算は，プロセスが生成され，最初のピリオドが開始したときに行っていた．しかし，非同期ピリオドでは，各プロセスのピリオドの開始時刻が異なる．そのため，全てのプロセスについてピリオドが開始したときにグループ分けと実行時間の計算を行う．さらに，新しくプロセスが生成されたときも同様に，グループ分けと計算を行う．

非同期ピリオドでも同期ピリオドと同様に，線形計画問題に変換する方法，線形計画問題を用いない方法の双方が考えられる．しかし，非同期ピリオドでは，あるプロセスのピリオドの途中で別プロセスのピリオドが始まり，実行時間等の計算が行われる可能性があることを考慮する必要がある．実際には，同期ピリオドと同様の計算を行うが，プロセスがそれまでに実行した時間や，計算が行われるときにはデッドラインまでの時間は d_i より小さくなっている可能性があるため，計算が行われる時刻から短くなったデッドラインまでの時間を考慮する．

非同期ピリオドにおいて，全てのプロセスが同時に参入し，ピリオドが等しく，それまでに実行した時間が 0 であるとき，非同期ピリオドは同期ピリオドと同じである．つまり，同期ピリオドは，非同期ピリオドのひとつのケースである．

4 シミュレーション

4.1 概要

本節は，提案する DEADLINE スケジューラの設計を，シミュレーションプログラムとして実装し，評価する．3.2.2 節にあるアルゴリズムを非同期ピリオド向けに拡張したものを C 言語で実現したシミュレーションプログラムを作成した．なお，このプログラムによって求められたプロセスの実行率は，線形計画問題を解く PuLP [2] という Python ライブラリを用いて求めた実行率との誤差の平均が $\pm 1.0 \times 10^{-5}$ 以内に収まることを確認済みである．

4.2 入力データ

各プロセスに関する入力データとして，次のパラメータを与える．

- プロセス番号
- ピリオド
- ランタイム
- デッドライン
- 参入時刻
- 残りの総実行時間

プロセス番号は，プロセスを区別するための一意の番号である．参入時刻は，プロセスが生成され参入する時刻を表すパラメータである．残りの総実行時間は，プロセスが実行される合計の時間を表すパラメータであり，プロセスが実行されると減少する．

4.3 評価手順

シミュレーションプログラムを用いて，ピリオドが同期している場合と，していない場合の両方のデータを与え，提案手法の評価を行った．入力データとして与えるデータでは 10 個のプロセスを用意し，パラメータはある範囲内に一様にランダムに生成した [5] [4]．ピリオドは，1 ns から 10,000,000 ns の範囲で生成した．そして，プロセス全体の使用率の和が指定した値になるように各プロセスの使用率をランダムに生成し，その値とピリオドの積をランタイムの値とした．デッドラインは，ランタイムの値からピリオドの値の範囲，参入時刻は 0 ns からピリオドの 5 倍の値の範囲，残りの総実行時間は 10,000,000 ns で全て固定とした．ピリオドが同期している場合の参入時刻は，全て 0 ns とした．

プロセス全体の使用率の合計 U が $U = 1.1, 1.2, 1.3, 1.4, 1.5$ のときについて，プロセスが各ピリオドの終わりを迎えたとき，そのピリオドで実行した時間を記録し，各ピリオドでの実行率を求めた．最小の実行率を最大化できているかを評価するため，プロセスの最後のピリオドを除いた全てのピリオドの実行率の最小値を求めた．プロセスの最後のピリオドの結果を除いたのは，残りの総実行時間が小さくなり，その分の時間しか実行されず，実行率が他のピリオドと比べ，小さくなるためである．また，$U = 1.5$ のとき，同期ピリオドと非同期ピリオドの両方の実行の一部を図として表した．

4.4 結果と考察

同期ピリオドの実行した結果を表 5 に，実行の一部を図 5 に示す．また，非同期ピリオドの実行した結果を表 6 に，実行の一部を図 6 に示す．

プロセスがデッドライン順に並んでいないので分かりにくいが，表 5，表 6 より，同期ピリオドと非同期ピリオドにおいて，最小の実行率が複数あり，これらが最大化されていることがわかる．また，最小の実行率以外に同じ値の実行率を持つプロセスがあることから，同じグループに属すプロセスで実行率が等しくなるように実行時間が計算されていることがわかる．このことから，3 節にあるアルゴリズムを用いて実行時間を計算した結果，同期，非同期の両方で最小の実行率の最大化し，同じグループに属す全てのプロセスで実行率が等しくなるように実行時間の計算ができているといえる．

しかし，今回のアルゴリズムでは，全てのプロセスのピリオドが始まるたびに実行時間を計算し直している．そのため，プロセスの数が増えると，何度も再計算が発生してしまう．

5 カーネルへの実装

既存の DEADLINE スケジューラでは，最もデッドラインの近いプロセスを簡単に選択するため，赤黒木を用いてプロセスを管理している [5]．本研究では，プロセスの実行時間を計算する際，プロセスがデッドラインが近い順に整列している必要がある．そこで，既存のプロセスの管理機構を変更し，デッドラインが近い順に並んだ線形リストによってプロセスを管理する．

現在はこの方針に基づいて，カーネルへの実装を進めているところである．

6 おわりに

本研究では，現在の CPU の使用状態に応じた適切なパラメータを持つプロセスでなければ DEADLINE スケジューラを設定できないという問題を解決するた

表 5: 同期ピリオドでの提案手法の実行率の最小値

U	1.1	1.2	1.3	1.4	1.5
p_1	0.87036	0.83234	0.59635	0.50483	0.48420
p_2	0.87034	0.83234	0.59636	0.50484	0.48420
p_3	0.87034	0.83234	0.59636	0.50484	0.48420
p_4	0.87036	0.83234	0.59636	0.50484	0.48419
p_5	0.87036	0.83234	0.59636	1.00000	0.48420
p_6	0.87036	0.83234	0.59634	0.50483	0.48418
p_7	0.95842	0.83234	0.59636	0.50484	0.48419
p_8	0.87036	0.83234	0.59636	0.50484	1.00000
p_9	0.87036	0.83234	0.59636	0.50484	0.48420
p_{10}	0.87036	0.83234	0.59636	0.50484	0.48420

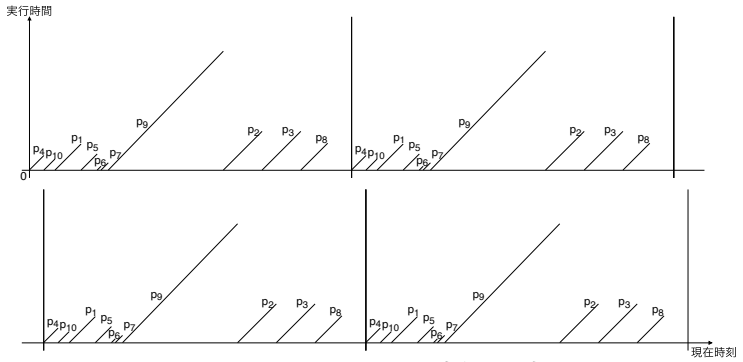

図 5: 同期ピリオドでの実行の一部

め，使用率がコア数を超える場合でも新しく DEADLINE スケジューラを設定できるように，全てのプロセスの実行時間をランタイムより小さくする手法を提案した．さらに，提案手法をシミュレーションプログラムとして実装し，その有効性を確認した

今後の課題としては，プロセスが増えることによる実行時間の再計算の増大化の抑制や Linux カーネルへの実装が挙げられる．

参 考 文 献

[1] Deadline Task Scheduling, `https://www.kernel.org/doc/html/latest/scheduler/sched-deadline.html`.

[2] Optimization with PuLP, `https://coin-or.github.io/pulp/`.

[3] Abeni, L. and Buttazzo, G.: Integrating multimedia applications in hard real-time systems, *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*, IEEE, 1998, pp. 4–13.

[4] Emberson, P., Stafford, R., and Davis, R. I.: Techniques for the synthesis of multiprocessor tasksets, *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, 2010, pp. 6–11.

[5] Lelli, J., Scordino, C., Abeni, L., and Faggioli, D.: Deadline scheduling in the Linux kernel, *Software: Practice and Experience*, Vol. 46, No. 6(2016), pp. 821–839.

[6] Liu, C. L. and Layland, J. W.: Scheduling algorithms for multiprogramming in a hard-real-time environment, *Journal of the ACM (JACM)*, Vol. 20, No. 1(1973), pp. 46–61.

表 6: 非同期ピリオドでの提案手法の実行率の最小値

U	1.1	1.2	1.3	1.4	1.5
p_1	0.82246	0.58696	0.81242	0.72543	0.52661
p_2	1.00000	0.85624	0.64078	0.80412	0.59075
p_3	0.89749	0.64819	0.56110	0.71590	0.70569
p_4	0.82245	0.74550	0.81975	0.71590	0.59075
p_5	1.00000	0.58696	0.85605	0.74011	0.56369
p_6	1.00000	0.64822	0.61601	0.71590	0.76425
p_7	0.84351	0.67678	0.56110	0.72543	0.52661
p_8	0.82641	0.63753	0.56110	0.74012	0.52661
p_9	0.94044	0.68206	0.56110	0.72543	0.56745
p_{10}	1.00000	1.00000	0.56111	0.74012	0.56882

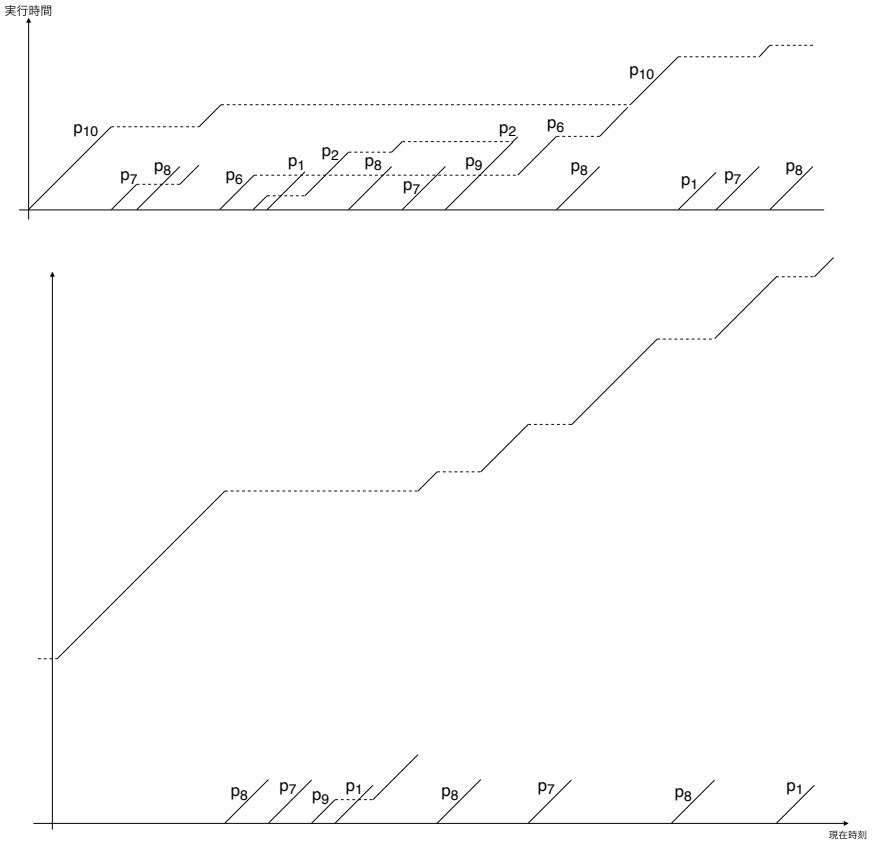

図 6: 非同期ピリオドでの実行の一部