Vol.2015-SE-188 No.7
Vol.2015-EMB-37 No.7
2015/6/5

アスペクト指向プログラミングによるリアルタイムOSスケジューラのカスタマイズ

原田 祐輔[1,a)]　阿部 一樹[1,†1]　兪 明連[1,b)]　横山 孝典[1,c)]

概要：組込みシステムは様々な用途に使用されるため，アプリケーションによってリアルタイム OS に要求される機能は異なる．ところが組込みシステムには厳しいリソース制約が有るため，それら全ての機能を持つリアルタイム OS を使用することは困難であり，対象アプリケーションに特化したリアルタイム OS を提供することが望ましい．本論文では，アプリケーションによって要求されるリアルタイム OS の機能としてタスクスケジューリングを取り上げ，アスペクト指向プログラミングを用いることで，既存のソースコードを直接修正せずにスケジューラをカスタマイズする手法について述べる．具体的には自動車制御分野向けの OSEK OS を対象に，仕様で規定された固定優先度スケジューリングアルゴリズムを，EDF または RMCL スケジューリングアルゴリズムにカスタマイズするアスペクトを提案するとともに，実用上問題ないオーバヘッドとメモリ消費量で実装可能であることを示す．

キーワード：組込みシステム，リアルタイム OS，スケジューリング，アスペクト指向プログラミング

1. はじめに

組込みシステムは様々な用途に使用されるため，要求されるリアルタイム OS の機能もアプリケーションによって異なることが多い．ところが，組込みシステムにはリソースの制約があり，単一のリアルタイム OS が様々なアプリケーションに対応可能な機能を持つことは困難である．このため，それぞれのアプリケーションに特化したリアルタイム OS を提供することが望ましい．しかし，リアルタイム OS の各機能は実装上複雑に絡み合っていることが多く，一部を直接書換えて多数のバージョン (バリアント) を開発すると，それらのソースコードの構成管理は容易ではなくなる．より効率的なカスタマイズ手法が求められている．

そこで，横断的関心事を分離してモジュール化することのできるアスペクト指向プログラミング [1] を用いて，リアルタイム OS をカスタマイズする研究がなされている．アスペクト指向プログラミングを用いることで，ソースコードを直接修正することなく，リアルタイム OS の機能の追加・変更が可能になる．オリジナルのソースコードを直接修正せずに済むことは，オープンソースのリアルタイム OS をカスタマイズする場合に，特に有効と考える．

Beuche らは，アスペクト指向プログラミングを用いてアーキテクチャ非依存なリアルタイム OS の実現する手法を提案している [2]．その後，Lohman，Spinczyk らのグループは，彼らが開発したアスペクト指向言語 AspectC++[3] によるリアルタイム OS の実装を行い，そのオーバヘッドが十分小さいことを示した [4], [5]．さらに最初からアスペクトを意識した設計が必要として Aspect-Aware Design によるリアルタイム OS を提案し，実装している [6], [7]．

Afonso らは，リアルタイム OS の同期 (排他制御) やロギングにアスペクトを適用している [8]．Park らは，プログラミング言語非依存なアスペクト指向プログラミング環境 AOX を開発し，カスタマイズ可能なリアルタイム OS への適用を提案している [9]．また齋藤らは，既存のリアルタイム OS のソースコードを修正せずに，分散リアルタイム OS を実現するためのアスペクトを提案した [10]．

しかし，リアルタイム OS の重要機能であるスケジューリングについては，アスペクト指向による具体的なカスタマイズ手法は提案されていない．リアルタイム OS の多くは固定優先度スケジューリングを採用しているが，組込みシステムの多様化にともない他のスケジューリングアルゴリズムに変更することが求められる．

そこで我々の研究の目的は，既存のリアルタイム OS の

[1] 東京都市大学
Tokyo City University, Tokyo 158–8557, Japan
[†1] 現在，株式会社 PFU
Presently with PFU LIMITED
[a)] g1581519@tcu.ac.jp
[b)] yoo@tcu.ac.jp
[c)] yokoyama@tcu.ac.jp

ソースコードを直接修正せずに，アスペクト指向プログラミングを用いてスケジューリングアルゴリズムを静的にカスタマイズするアスペクトを提案し，実装することである．

我々はこれまでに，固定優先度スケジューリングを採用している自動車制御分野向けの標準リアルタイム OS である OSEK OS[11] をベースに，アスペクト指向プログラミングにより固定優先度スケジューリングを EDF(Earliest Deadline First) スケジューリングにカスタマイズする手法を提案した [12]．本論文ではさらに RMCL(Rate Monotonic Critical Laxity) スケジューリング [13] に置き換えるアスペクトを提案する．そしてアスペクトによってカスタマイズしたリアルタイム OS を対象に実用上問題ないオーバヘッド，メモリ消費量で実装可能であることを示す．

2. アスペクト指向プログラミングによるリアルタイム OS のカスタマイズ

2.1 カスタマイズ対象

スケジューリングアルゴリズムには大別すると固定優先度アルゴリズムと動的優先度アルゴリズムがある．本研究では OSEK OS の固定優先度アルゴリズムを動的優先度アルゴリズムにカスタマイズする手法を提案する．代表的な動的優先度アルゴリズムには，デッドラインを用いて優先度を決定する手法と，余裕時間を用いて優先度を決定する手法がある．前者の代表として EDF スケジューリング，後者の代表として LLF(Least Laxity First) スケジューリングがある．しかし LLF は単位時間毎にスケジューラを起動する必要があるため実行効率に問題があり，実用性に欠ける．そこで本研究では余裕時間を用いる手法として，タスクの起動や終了などのタイミングでスケジューラを起動すればよい RMCL スケジューリングを選ぶ．すなわち，本研究では EDF スケジューリングと RMCL スケジューリングをカスタマイズ対象とする．

また，OSEK OS のリソースアクセスプロトコルはタスクの優先度を変更することで実装しているが，そのままでは動的優先度アルゴリズムには適用できない．そこで各スケジューリングアルゴリズムに対応したリソースアクセスプロトコルにカスタマイズする．以下本論文では，EDF でのリソースアクセスプロトコルを EDF 向け優先度上限プロトコル，RMCL でのリソースアクセスプロトコルを RMCL 向け優先度上限プロトコルと呼ぶ．

2.2 カスタマイズ方法

我々は，OSEK OS 仕様に基づくオープンソースのリアルタイム OS である TOPPERS/ATK1[14] を対象にカスタマイズを行う．TOPPERS/ATK1 の大部分は C 言語で記述されているため，我々は C 言語ベースのアスペクト指向言語 ACC (AspeCt-oriented C)[15], [16] を用いることとした．ACC は AspectJ[17] や Aspect C++と同様な，ジョインポイントモデルに基づくアスペクト指向言語である．

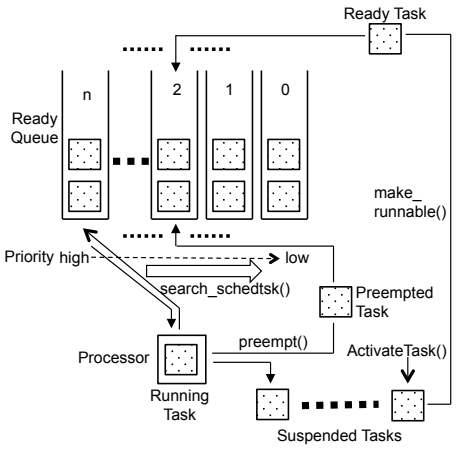

図 1　固定優先度スケジューリングのメカニズム

ACC で扱えるジョインポイントは，関数の呼び出し (call) と実行 (execution)，変数への値の書き込み (set) と値の読み出し (get) である．アスペクトはポイントカット (pointcut) とアドバイス (advice) から成る．ポイントカットはジョインポイントの集合を指定するもので，アドバイスはポイントカットに合致するジョインポイントで実行する処理を記述したものである．ACC は before，after，around の 3 つのアドバイスをサポートしており，対象ジョインポイントの前後，あるいはジョインポイントの代わりにアドバイスコードを実行することができる．

ACC はトランスレータとして実装されており，ACC および C のソースファイルを入力し，織り込みを行った後，C のソースファイルを出力する．出力されたファイルを C コンパイラによりコンパイルすることでオブジェクトコードを生成できる．

3. スケジューリングアルゴリズムのカスタマイズ

3.1 OSEK OS のスケジューリング

図 1 に TOPPERS/ATK1 の固定優先度スケジューリングのメカニズムを示す．タスクが実行可能状態になると，優先度ごとに用意されるレディキューに格納する．タスクの優先度はタスクコントロールブロック (Task Control Block, TCB) に設定する．実行タスクが終了すると，関数 search_schedtsk が優先度が最も高いレディキューの先頭タスクを選び実行する．システムコール ActivateTask またはアラーム機構によってタスクを起動する場合，関数 make_runnable がタスクを実行可能状態にし，優先度に応じたレディキューへ格納する．起動するタスクの優先度が実行中タスクの優先度よりも高い場合はプリエンプションが発生する．プリエンプションは関数 preempt が行い，プリエンプトされたタスクを優先度に応じたレディキューに格納する．

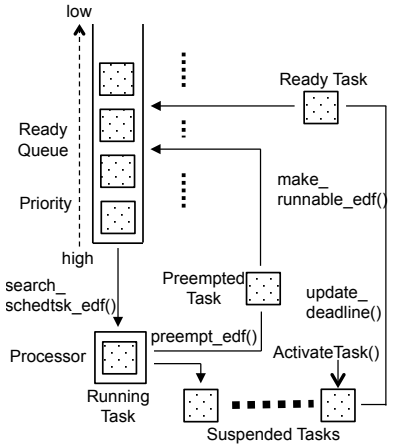

図 2　EDF スケジューリングのメカニズム

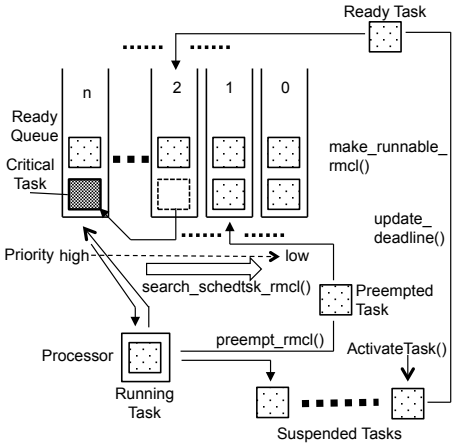

図 3　RMCL スケジューリングのメカニズム

3.2 EDF スケジューリング

EDF スケジューリングではタスクの優先度を絶対デッドラインで表す．タスクの相対デッドラインは OIL によって宣言し，SG によりコンフィグレーションデータに変換され記憶される．タスク起動時に，EDF スケジューラは相対デッドラインをもとに絶対デッドラインを求めて TCB に設定し，タスクの優先度として用いる．

EDF スケジューリングのメカニズムを図 **2** に示す．タスクが実行可能状態になると，絶対デッドラインの近いものから順に単一レディキューに格納する．実行中のタスク以外で最も近い絶対デッドラインを持つタスクがレディキューの先頭になる．実行中のタスクが終了またはプリエンプトされると，スケジューラがレディキューの先頭から実行タスクを選び実行状態にする．プリエンプトされたタスクはレディキューの絶対デッドラインに応じた位置に格納する．

EDF スケジューリングにカスタマイズするアスペクトの詳細は既に発表している [12] ので本論文では省略する．

3.3 RMCL スケジューリング

RMCL スケジューリングはデッドラインまでの時間と残り実行時間の差である余裕時間を使用する．そこで EDF スケジューリングで用いられるデッドラインに加え，タスクの最悪実行時間 (Worst Case Executioin Time, WCET) を OIL で宣言し，SG によりコンフィグレーションデータに変換し記憶する．また TCB に残り実行時間を記憶する．タスク起動時に，最悪実行時間をタスクの残り実行時間として記憶するとともに，スケジューラがタスクの残り実行時間を更新する．RMCL では実行中のタスクの残り実行時間より余裕時間が小さいタスクが発生した場合，それをクリティカルタスクと呼び，その優先度を最高優先度とする．

RMCL スケジューリングのメカニズムを図 **3** に示す．タスクが実行可能状態になると優先度に応じたレディキューに格納する．その後レディキューからクリティカルタスクを探す．ない場合は OSEK OS の固定優先度スケジューリングと同様の動作を行う．クリティカルタスクがある場合，レディキューからクリティカルタスクを取出して実行する．実行中だったタスクは優先度に応じたレディキューの先頭へ格納する．

```
around(): execution(void search_schedtsk(void)) {
  search_schedtsk_rmcl();
} /* クリティカルタスクの探索 */
```

図 4　RMCL での実行タスク選択に置き換えるアスペクト

```
BOOL around(TaskType tskid) : execution(BOOL make_runnable(
     TaskType)) && args(tskid) {
  return(make_runnable_rmcl(tskid));
} /* 実行中タスクおよび実行タスクをキューへ挿入処理 */
```

図 5　RMCL でのレディキューへ格納処理に置き換えるアスペクト

RMCL スケジューリングにカスタマイズするアスペクトを図 **4**〜図 **8** を用いて説明する．

実行タスク探索のためのアスペクトを図 4 に示す．このアスペクトは around アドバイスで，最高優先順位タスクを探索する関数 search_schedtsk を search_schedtsk_rmcl の呼び出しに置き換える．search_schedtsk_rmcl はレディキュー中からクリティカルタスクの探索を行い，ある場合はクリティカルタスクに設定されているリソースの空き状況を確認する．使用中の場合はクリティカルタスクを最高優先度レディキューへ格納し，リソースを獲得しているタスクの優先度を引き上げて先に実行タスクに設定する．

タスクを実行可能状態へ遷移させるためのアスペクトを図 **5** に示す．このアスペクトは around アドバイスで，起動タスクを実行可能状態にし，実行タスクを決める関数 make_runnable を make_runnable_rmcl に置き換える．make_runnable_rmcl は起動タスクを実行可能状態にし，実行中タスクと共にレディキューへ格納する．その後レディキュー中からクリティカルタスクの探索を行い実行タスクを決める．

```
before() : execution(void preempt(void)) {
  if(nextpri < tcb_curpri[schedtsk]) {
    nextpri = tcb_curpri[schedtsk];
  }
} /* クリティカルタスク探索するための処理 */
```

図 6　RMCL でのプリエンプションに置き換えるアスペクト

```
StatusType around() : execution(StatusType Schedule()) {
  return(Schedule_rmcl());
} /* RMCL スケジューラの呼び出し */
```

図 7　RMCL でのスケジューラに置き換えるアスペクト

```
before(TaskType tskid) : execution(BOOL make_runnable(
     TaskType)) && args(tskid) {
  tcb_lftexetm[tskid] = tinib_wstexetm[tskid];
} /* 残り実行時間の初期化処理 */
before() : execution(void search_schedtsk(void)) {
  if(schedtsk != TSKID_NULL) {
    TickType exetm = diff_tick(cntcb_curval[alminib_cntid[
         tinib_almid[schedtsk]]], tcb_exebgntm[schedtsk],
         cntinib_maxval2[alminib_cntid[tinib_almid[schedtsk
         ]]]);
    if(exetm > tcb_lftexetm[schedtsk]) {
      tcb_lftexetm[schedtsk] = 0;
    } else {
       tcb_lftexetm[schedtsk] -= exetm;
    }
  }
} /* 残り実行時間の更新処理 */
after() : execution(void search_schedtsk(void)) {
  if(schedtsk != TSKID_NULL) {
    tcb_exebgntm[schedtsk] = cntcb_curval[alminib_cntid[
         tinib_almid[schedtsk]]];
  }
} /* 残り実行時間の更新処理 */
```

図 8　タスクの残り実行時間管理のためのアスペクト

プリエンプションのためのアスペクトを図 **6** に示す．このアスペクトは before アドバイスで，プリエンプションを行う関数 preempt の前に次に実行するタスクの優先度の更新を行う処理を追加している．クリティカルタスクの探索をプリエンプションの処理内で行うためである．

スケジューラの呼び出しのためのアスペクトを図 **7** に示す．このアスペクトは around アドバイスで，スケジューリングを行うシステムコールの Schedule を Schedule_rmcl に置き換える．Schedule_rmcl はプリエンプションを発生させクリティカルタスクを探索し実行タスクを決める．

タスクの最悪実行時間の管理のためのアスペクトを図 8 に示す．このアスペクトは before アドバイスで関数 make_runnable 実行前に最悪実行時間でタスクの残り実行時間を処理する．また before アドバイスと after アドバイスで search_schedtsk の実行前に残り実行時間の更新を行い，実行後にタスクの実行開始時刻の設定を行う．これらはタスクの余裕時間の算出に用いられる時間である．

絶対デッドライン管理のためのアスペクトを図 **9** に示す．このアスペクトは before アドバイスで, 起動タスクの起動処理を行う関数 make_active の実行前に関数 update_deadline を呼び出す．関数 update_deadline は，タスクの相対デッドラインをもとに，タスクの絶対デットラインを TCB に設定する．

```
before(TaskType tskid): execution(BOOL make_active(TaskType
     )) && args(tskid) {
  update_deadline(tskid);
} /* タスクの絶対デッドライン更新処理 */
```

図 9　絶対デッドライン管理のためのアスペクト

これら以外に，RMCL スケジューリングで用いるデータの初期化のためのアスペクトとタスクの終了処理のためのアスペクトも定義した．

4. リソースアクセスプロトコルのカスタマイズ

4.1 OSEK 優先度上限プロトコル

OSEK OS でのリソース管理はスタックリソースポリシーをベ固定優先度向けに簡略化したもので，OSEK 優先度上限プロトコル (OSEK Priority Ceiling Protocol) と呼ばれる．その仕様を以下に示す [11].

- SG にてそれぞれのリソースに対して静的に上限優先度を設定する．リソースにアクセスするタスクの最高優先度が設定される．上限優先度はリソースにアクセスしないすべてのリソースの最低優先度よりも低くなる．また，少なくともリソースにアクセスするすべてのタスクの優先度よりも高い優先度を持つ．
- 実行中のタスクの優先度がリソースの上限優先度よりも低いタスクがリソースを獲得する時，タスクの優先度はリソースの上限優先度へ引き上げる．
- タスクがリソースを解放する時，タスクの優先度はリソースを獲得する前の優先度に戻る．

TOPPERS/ATK1 ではリソースの上限優先度をそのリソースにアクセスするタスク中の最高優先度と同じ値としている．

リソースを獲得するにはシステムコール GetResource をリソースを解放するにはシステムコール ReleaseResource を用いる．共有リソースにアクセスする前にタスクが GetResource を呼ぶ時，タスクの優先度を共有リソースの上限優先度へ引き上げる．共有リソースにアクセスした後にタスクが ReleaseResource を呼ぶ時，タスクの優先度はもとの優先度へ戻る．

4.2 EDF 向け優先度上限プロトコル

EDF スケジューリングで，タスクの優先度は絶対デッドラインにて表され，動的に変化する．そこで，リソースを要求したタスクの中で絶対デッドラインが最も近いタスクのデッドラインを上限優先度としてリソースの管理を行う．

- タスクがリソースを獲得すると，上限優先度にリソースを獲得しうる実行中または実行可能状態タスクの中で最も高い優先度を設定する．上限優先度はリソースにアクセスしない実行可能タスクの最低優先度よりも低く，リソースにアクセスする実行中または実行可能状態タスクの最高優先度よりも高くなる．
- タスクがリソースを獲得中に，自身よりも高い優先度にリソースの上限優先度が更新されたら，タスクの優先度は高い優先度に引き上げる．
- 複数リソースを獲得しているタスクがあるリソースを解放する時，タスクの優先度は他のリソースの最高優先度と同じ優先度に更新する．すべてのリソースをタスクが解放すると，優先度はもとの優先度へ戻る．

タスクの優先度は絶対デッドラインによって表されるので，リソースの上限優先度はすべての実行中もしくは実行可能状態のリソースを共有するタスクの中で最も早い絶対デッドラインで表される．

リソースの上限優先度管理について説明する．タスクがリソースを獲得する時，そのタスクの優先度をリソースの上限優先度に設定する．リソースを共有するほかのタスクが起動しそのタスクの優先度のほうが高い場合は上限優先度を更新する．リソースの上限優先度はタスクがリソースを解放するまでリソースを共有するタスクの中で最高の優先度が設定される．どのタスクもリソースを獲得していない時はリソースの上限優先度は設定しない．

タスクの優先度管理について説明する．タスクの優先度は獲得したリソースの上限優先度に一致させる．リソースを共有するほかのタスクが起動されると，リソースの上限優先度を引き上げる．その時リソースを獲得しているタスクの優先度を一時的に引き上げる．タスクが全てのリソースを解放すると，一時的に引き上げられたタスクの優先度はもとの優先度へ戻る．

図 **10** に EDF 優先度上限プロトコルの動作例を示す．タスク TaskA, TaskB, TaskC はリソース Res を共有する．タスク A の絶対デッドラインは TaskB と TaskC よりも早い．タスクの優先度はタスクが起動されるときに設定される．

t1 で TaskC が起動され，Res 上限優先度は TaskC の優先度と同じになる．t2 で TaskC が GetResource(Res) を呼び Res を獲得する．t3 で TaskB が起動されると，Res の上限優先度は TaskB の優先度と同じになり TaskC の優先度は一時的に上限優先度へ引き上げられ，TaskC は TaskB にプリエンプトされない．t4 で TaskA が起動されると，Res の上限優先度は TaskA の優先度と同じになり，TaskC は再び一時的に上限優先度に引き上げられ，TaskC は TaskB にプリエンプトされない．t5 で TaskC が ReleaseResource(Res) を呼ぶと TaskC の優先度はもとの優先度へ戻ると，TaskA は TaskC をプリエンプトする．t8 で TaskA の実行が完了すると，TaskB を実行する．t11 で TaskB が実行完了すると TaskC を再開する．

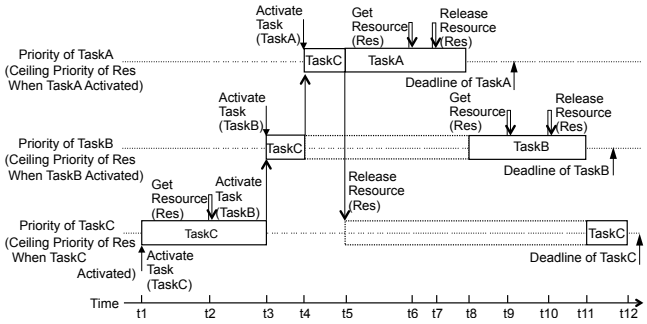

図 **10** EDF 向け優先度上限プロトコル

OSEK 優先度上限プロトコルを EDF 向け優先度上限プロトコルに置き換えるアスペクトの詳細は既に発表している [12] ので本論文では省略する．

4.3 RMCL 向け優先度上限プロトコル

RMCL スケジューリングアルゴリズムでのリソースアクセスプロトコルを OSEK 優先度上限プロトコルをベースに設計する．クリティカルタスクがない場合は OSEK OS における OSEK 優先度上限プロトコルに準じた動作をする．ここではクリティカルタスクがある場合の動作について説明をする．

- リソースの上限優先度は OSEK 優先度上限プロトコルで設定される同様の優先度を持つ．
- クリティカルタスクがリソース獲得時にタスクの優先度はタスクの中で最高優先度のままとなる．あるタスクがクリティカルタスクと共有するリソースを獲得している時クリティカルタスクの優先度を最高優先度へ引き上げる前に，共有するリソースを獲得しているタスクの優先度を先に最高優先度へ引き上げ実行タスクとする．
- クリティカルタスクと共有するリソースを獲得しているタスクがリソースを解放するとタスクの優先度はもとの優先度へ戻り，最高優先度であるクリティカルタスクが実行タスクとなる．

リソースの上限優先度管理について説明する．あるタスクがクリティカルタスクになると獲得しているリソースと他のタスクと共有しているリソースの上限優先度を，クリティカルタスクの優先度と同様のタスクの中で最高の優先度に設定する．リソースを解放するとリソースの上限優先度をもとに戻す．

タスクの優先度管理について説明する．クリティカルタスクがリソースを獲得する時はタスクの優先度はタスク内での最高優先度のまま変更されない．あるタスクがクリティカルタスクになる前に，リソースを共有している他のタスクがリソースを獲得している場合，タスクがクリティカルタスクとなった時点でクリティカルタスクとリソース

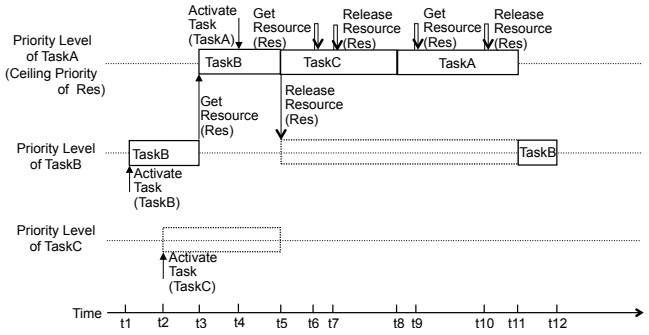

図 **11** RMCL 向け優先度上限プロトコル

を共有しているタスクの優先度をタスク内での最高優先度に引き上げる．クリティカルタスクとリソースを共有するタスクがリソースを解放すると優先度はもとの優先度へ戻る．クリティカルタスクがリソースを解放すると優先度はタスク内での最高優先度のまま変更されず実行を続ける．

図 10 に RMCL 優先度上限プロトコルの動作例を示す．TaskC がクリティカルタスクになる場合について説明する．タスク TaskA, TaskB, TaskC はリソース Res を共有し，TaskA の優先度は TaskB と TaskC よりも高い優先度とする．

t1 で TaskB が起動する．t2 で TaskC が起動するが TaskC の方が優先度が低いため待ち状態となる．t3 で TaskB が GetResource(Res) を呼び，Res を獲得し TaskB の優先度は上限優先度へ引き上げられる．t4 で TaskA が起動される時，TaskC がクリティカルタスクとなり TaskC の優先度はタスク内での最高優先度の TaskA の優先度へ引き上げられる．しかしこの時 TaskB が Res を保持しているため TaskB の優先度を一時的に最高優先度へ引き上げリソースが解放されるまで TaskB の実行を再開する．t5 にて TaskB が ReleaseResource(Res) を呼ぶと TaskB の優先度はもとの優先度へ戻り，t4 の時点でクリティカルタスクとなった TaskC の実行を開始する．t8 にて TaskC の実行が完了すると TaskA を実行する．t11 にて TaskA の実行が完了すると TaskB の実行が再開する．

RMCL 向け優先度上限プロトコルのためのアスペクトを図 **12** に示す．このアスペクトは around アドバイスで，リソースの獲得を行うシステムコール GetResource を GetResource_rmcl に置き換える．GetResource_rmcl はクリティカルタスクの場合はリソースの上限優先度をクリティカルタスクの優先度と同様のタスク内での最高優先度に設定する．また同様に around アドバイスで，リソースの解放を行うシステムコール ReleaseResource を ReleaseResource_rmcl に置き換える．ReleaseResource_rmcl は，クリティカルタスクの時はリソース解放後も最高優先度から変更しない．クリティカルタスクとリソースを共有しているタスクの場合は，タスクの優先度に応じたレディキューへ格納しクリティカルタスクを最高優先度レディキューから取出し実行タスクに設定する．

```
StatusType around(ResourceType resid) : execution(StatusType
      GetResource(ResourceType)) && args(resid) {
  return(GetResource_rmcl(resid));
} /* RMCL 向けリソース獲得処理 */
StatusType around(ResourceType resid) : execution(StatusType
      ReleaseResource(ResourceType)) && args(resid) {
  return(ReleaseResource_rmcl(resid));
} /* RMCL 向けリソース解放処理 */
```

図 **12** RMCL 向け優先度上限プロトコルに置き換えるアスペクト

```
before(ResourceType resid) : execution(StatusType
     GetResource(ResourceType)) && args(resid){
  if (callevel == TCL_TASK) {
    rescb_bitmap[resid/16] |= RESMAP_BIT(resid%16);
    rescb_usingtask[resid] = runtsk;
    rescb_used++;
  }
} /* リソース獲得更新処理 */
before(ResourceType resid) : execution(StatusType
     ReleaseResource(ResourceType)) && args(resid){
  if (callevel == TCL_TASK) {
    rescb_bitmap[resid/16] &= ~RESMAP_BIT(resid%16);
    rescb_usingtask[resid] = TSKID_NULL;
    rescb_used--;
  }
} /* リソース解放更新処理 */
```

図 **13** リソースの獲得状況管理のためのアスペクト

リソース使用状況管理のためのアスペクトを図 **13** に示す．このアスペクトは before アドバイスで，リソース獲得を行う GetResource またはリソースの解放を行う ReleaseResouece の実行前にリソースの使用状況の更新を行う．リソースの獲得時はリソースを使用状態に更新し，どのタスクがリソースを獲得するかを設定する．リソースの解放時はリソースを解放状態に更新し，リソースが空いている状態に設定する．このアスペクトは RMCL の優先度上限プロトコルでも共通で利用するアスペクトである．

また，RMCL 向け優先度上限プロトコルで用いるデータの初期化のためのアスペクトも定義した．

4.4 OIL の拡張

今回提案した複数のスケジューリングアルゴリズムに対応するために OIL を拡張する．まず，OS オブジェクトのスケジューリングアルゴリズムを選択する SCHEDULER を追加する．固定優先度の場合は SCHEDULER は FPRIORITY，EDF の場合は EDF，RMCL の場合は RMCL とする．次に，EDF の場合は TASK オブジェクトにタスクの相対デッドラインである DEADLINE，RMCL の場合はタスクの最悪実行時間の WCET(Worst Case Execution Time) を追加する．図 **14** に拡張した OIL の記述例を示す．この例では OS オブジェクトにて RMCL スケジューラを選択している．タスクオブジェクトでは sample_task1 の相対デッドラインを 10，最悪実行時間を 3 に設定している．OIL 記述からコンフィグレーションデータを出力す

```
OS sample_os { /* OS オブジェクトの記述 */
  STATUS = EXTENDED;
  STARTHOOK = TRUE;
  SCHEDULER = RMCL; /* 固定優先度, ＥＤＦ, ＲＭＣＬから選択 */
  ・・・・・・
}
TASK sample_task1 { /* タスクオブジェクトの記述 */
  AUTOSTART = TRUE { APPMODE = sample_mode1; };
  DEADLINE = 10; /* デッドライン */
  WCET = 3; /* 最悪実行時間 */
  RESOURCE = sample_res1;
  ・・・・・・
}
```

図 **14** OIL 記述例

る SG は現在開発中である．このため現在は，コンフィグレーションデータのソースコードを直接記述している．

5. 実装および評価

5.1 実験環境

本研究で評価に用いた実験環境を以下に示す．マイクロコントローラ H8S/2638F を搭載した評価ボードを使用した．カスタマイズする RTOS として TOPPERS/ATK1 Release1.0 を用いた．アスペクト指向言語には ACC ver0.9 を用いた．

5.2 オーバヘッドの評価

アスペクト指向プログラミングによるオーバヘッドを評価するため，アスペクト指向によりカスタマイズした 3 つのシステムコールについて実行時間を測定し，ソースコードを直接書き換えて実行した同一のリアルタイム OS の実行時間とを比較する．参考のため，固定優先度の場合の実行時間も示す．実行時間の測定には 5MHz のハードウェアタイマを用いた．表に示した値は各システムコールの発行から終了までにかかる時間を 100 回計測した平均値である．

システムコール ActivateTask の実行時間を表 **1** に示す．ここでは，タスク起動の際にタスクスイッチが発生する場合と発生しない場合について計測した．システムコール GetResource，ReleaseResource の実行時間を表 **2** に示す．また RMCL スケジューラにて，ActivateTask ではクリティカルタスクを探索する時間，GetResource, ReleaseResource のリソース機能ではクリティカルタスクが行う実行時間を示す．

アスペクト指向プログラミングにより実装すると，アドバイスコードを呼び出すための関数呼び出しが発生するため，直接書き換えて実装した場合と比較して，関数呼び出し 1 回分のオーバヘッドが発生する．アスペクト指向プログラミングにより実装した場合の実行時間は，直接書き換えて実装した場合に比べ，一部を除いて 10%以下の増大であり十分に小さいと考える．EDF におけるリソース機能の場合のみ 20〜30%程度増大しているが，6μsec 程度の増大のため大きな問題にはならないと考える．

表 **1** タスク起動の実行時間の評価

対象 OS		タスク切換えなし	タスク切換えあり	
				クリティカルタスクあり
EDF	AOP	39.0	42.9	-
	直接書換	37.1	41.5	-
RMCL	AOP	33.9	84.6	94.1
	直接書換	33.5	84.0	93.3
オリジナル OS		25.5	26.9	-

[μsec]

表 **2** リソース機能の実行時間の評価

対象 OS		GetResource		ReleaseResource	
			クリティカルタスクあり		クリティカルタスクあり
EDF	AOP	27.3	-	33.3	-
	直接書換	21.3	-	27.7	-
RMCL	AOP	32.6	31.8	90.5	29.5
	直接書換	30.5	30.5	89.5	26.9
オリジナル OS		22.0	-	21.7	-

[μsec]

表 **3** メモリ消費量の評価

対象 OS		データ		コードサイズ
		RAM	ROM	
EDF	AOP	164	848	15084
	直接書換	165	848	13760
RMCL	AOP	170	872	15904
	直接書換	170	872	14112
オリジナル OS		133	812	12440

[Byte]

5.3 メモリ消費量の評価

表 **3** に，アスペクトによるカスタマイズによる，メモリ消費量の評価結果を示す．

アスペクトを用いず人手でカスタマイズした OS と比較すると，コードサイズは 10〜15%程度の増加となる．これは，アスペクトを C 言語コードへ変換する際，処理内容を記述したアドバイスコードを関数に変換するためで，直接 OS のコードを書き換えるよりもメモリ消費量は大きくなる．増大量は 1.3kB〜1.8kB で，コードが書き込まれる ROM 領域の容量の 256kB に対して 1%以下であり，実用上問題ない範囲と考える．

本研究において追加したデータによるメモリ消費量を表 **4** に示す．これらは全て，アスペクトを用いず直接書き換えてカスタマイズした場合にも必要になる変数および定数であり，アスペクト指向プログラミングによる影響はない．

以上の評価から，アスペクトを用いたカスタマイズでのメモリ消費量の増加は，実用上問題の無い範囲と考える．

表 4　追加したデータによるメモリ消費量

データ名	データの用途	メモリ消費量
tinib_deadline	[定数] タスクの相対デッドライン	$t \times 32bit$
tinib_almid	[定数] タスクを起動するアラームの ID	$t \times 8bit$
tinib_resource	[定数] タスクのリソース共有状況を表すビットマップ	$t \times (r \div 16 + 1) \times 16bit$
tinib_wstexetm	[定数] タスクの最悪実行時間	$t \times 32bit$
tcb_deadline	[変数] タスクの絶対デッドライン	$t \times 32bit$
tcb_lftexetm	[変数] タスクの残り実行時間	$t \times 32bit$
tcb_exebgntm	[変数] タスクの開始時刻	$t \times 32bit$
rescb_bitmap	[変数] リソース獲得状況を表すビットマップ	$(r \div 16 + 1) \times 16bit$
rescb_usingtask	[変数] リソース獲得中のタスクの ID	$r \times 8bit$

t : タスクオブジェクト数, r : リソースオブジェクト数

6. おわりに

本論文では OSEK OS 仕様に基づく TOPPERS/ATK1 の固定優先度スケジューリングを，アスペクト指向プログラミングを用いて EDF または RMCL スケジューリングにカスタマイズする手法を提案した．これにより，既存のソースコードを直接修正せずにアプリケーションに適したスケジューリングアルゴリズムに置き換え可能となる．そして，実用上問題ないオーバヘッドとメモリ消費量で実現可能であることを示した．

謝辞　本研究で使用した TOPPERS/ATK1 の開発者と ACC の開発者に感謝する．また RMCL スケジューラアスペクトのベースを開発した本学卒業生の舩坂拓海氏に感謝する．本研究の一部は JSPS 科研費 2450046 および 15K00084 の助成を受けたものである．

参考文献

[1] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M. and Irwin, J.: Aspect-Oriented Programming, *Proceedings of the 11th European Conference on Object-Oriented Programming*, pp.220–242 (1997).

[2] Beuche, D., Fröhlich, A. A., Reinhard, M., Papajewski, H., Schön F., Schröder-Preikschat, W., Spinczyk, O. and Spinczyk, U.: On Architecture Transparency in Operating Systems, *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating system,* pp.147–152 (2000).

[3] Spinczyk, O., Gal, A. and Schröder-Preikschat, W.: Aspect C++: An Aspect-Oriented Extension to the C++ Programming Language, *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems*, pp.53–59 (2002).

[4] Spinczyk, O. and Lohmann, D.: Using AOP to Develop Architecture-Neutral Operating System Components, *Proceedings of the 11th workshop on ACM SIGOPS European workshop Article,* No.34 (2004).

[5] Lohmann, D., Scheler, F., Tartler, R., Spinczyk, O. and Schröder-Preikschat, W.: A Quantitative Analysis of Aspects in the eCos Kernel, *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems,* pp.191–204 (2006).

[6] Lohmann, D., Hofer, W., Schröder-Preikschat, W. and Spinczyk, O.: Aspect-Aware Operating System Development, *Proceedings of the 10th International Conference on Aspect-Oriented Software Development 2011,* pp.69–80 (2011).

[7] Lohmann, D., Spinczyk, O., Hofer, W. and Schröder-Preikschat, W: The Aspect-Aware Design and Implementation of the CiAO Operating-System Family, *Transactions on Aspect-Oriented Software Development IX,* Lecture Notes in Computer Science Vol.7271, pp 168–215 (2012).

[8] Afonso, F., Silva, C., Montenegro, S. and Tavares, A.: Applying Aspects to a Real-Time Embedded Operating System, *Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software,* Article No.1 (2007).

[9] Park J. and Hong, S.: Building a Customizable Embedded Operating System with Fine-Grained Joinpoints Using the AOX Programming Environment, *Proceedings of the 2009 ACM symposium on Applied Computing,* pp.1952-1956 (2009).

[10] Saito, N., Yoo, M. and Yokoyama, T.: A Distributed Real-Time Operating System Built with Aspect-Oriented Programming for Distributed Embedded Control Systems, *20th IEEE International Conference on Parallel and Distributed System,* pp436-443 (2014).

[11] OSEK/VDX,*Operating System, Version 2.2.3* (2005).

[12] Abe, K., Yoo, M. and Yokoyama, T.: Aspect-Oriented Customization of the Scheduling Algorithm and the Resource Access Protocol of a Real-Time Operating System, *Proceedings of the IEEE 16th International Conference on Computational Science and Engineering,* pp.627–634 (2013).

[13] 加藤真平, 山崎信行, Linux カーネル用リアルタイムスケジューリングモジュール, 情報処理学会論文誌 コンピューティングシステム, Vol.2 No.1, pp.75-86, (2009).

[14] TOPPERS Project, http://www.toppers.jp/

[15] Gong, M., Zhang, C. and Jacobsen, H.-A.: Systems Development with AspeCt-oriented C (ACC), *Connections 2007 (ECE Graduate Symposium, University of Toronto),* Talk 5.6 (2007).

[16] AspeCt-oriented C, https://sites.google.com/a/gapp.msrg.utoronto.ca/aspectc/

[17] Kiczales, G., Hilsdale, E., Hugonin, J., Kersten, M. Palm J. and Griswold, W. G.:An Overview of AspectJ, *Proceedings of the 15th European Conference on Object-Oriented Programming*, pp.327–353 (2001).

[18] OSEK VDX, *System Generation OIL: OSEK Implementation Language Version 2.5* (2004).