リアルタイム OS におけるソフトウェアの動的更新手法

2012SE177 新美敦士　2012SE231 祖父江良　2012SE257 寺田行一
指導教員：宮澤元

1　はじめに

ソフトウェアの更新には，新しいバージョンのインストールとソフトウェアの再起動という 2 つの処理が必要となる．そのため，更新前のソフトウェアの停止から更新後の実行再開まで，更新対象となるソフトウェアの動作は停止する．特に，更新対象の中でも OS はシステムの基礎をなす部分であり，OS が停止すると OS 上で動作するソフトウェア群も停止するので大きな影響が出る．

このような問題に対応する手段の一つとしてソフトウェアの動的更新手法が挙げられる [1][2][3][4][5][6][7][8]．動的更新手法は動作中のソフトウェアを停止させることなく更新を行う事を目的としている．そこで動的更新手法ではソフトウェアのメモリ上の実行バイナリを実行中に書き換え，実行内容を切り換えることで更新を行う．これにより，システムを再起動させずソフトウェアの更新を行う事が可能となり，停止させることが出来ないシステムの更新が容易に行えるようになる．

一方，高機能化が進んでいる組み込みシステムにおいても，機器の停止を防ぐためにはソフトウェアの動的更新手法は有効だと考えられる．しかし，組み込みシステムで多く用いられているリアルタイム OS には，既存の動的更新手法をそのまま適用できるかどうか不明である．

本論文では，リアルタイム OS におけるソフトウェアの動的更新手法を提案し，ハードリアルタイムシステムに対応することのできる動的更新機能をもつリアルタイム OS を実現する．書き換え処理の中断による不正な実行を防ぎ，動的更新処理を最低優先度で実行できるようにすることでシステムの信頼性を向上するとともにリアルタイム性を維持する．

2　ソフトウェアの動的更新手法

本節では，メモリ上の実行バイナリを書き換えることによって動的更新を実現する手法 [1][2][3] について述べる．更新対象の関数の先頭に jmp 命令を挿入することによって動的更新手法を実現する．

例として，関数 `function` を更新後の関数である `function_new` に更新する場合を示す．**図 1** は動的更新の実行前における関数の呼び出し関係を表している．まず最初に，更新後に実行される関数 `function_new` をメモリ上に配置する．その後，関数 `function` の先頭を関数 `function_new` への jmp 命令に書き換える．これにより，関数 `function` を実行せず jmp 命令を経て，関数 `function_new` が実行されるようになる (**図 2**)．また，jmp 命令で関数 `function_new` を呼び出すので，スタックには影響を及ぼさず，関数 `function_new` の終了時に問題無く関数 `function` の呼び出し元に戻る事が可能である．

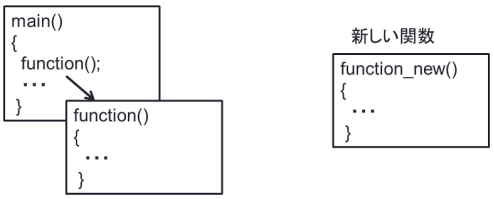

図 1　実行前の関数の呼び出し関係

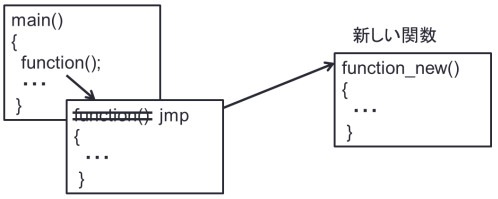

図 2　実行後の関数の呼び出し関係

3　リアルタイム OS における動的更新の問題点

リアルタイム性とは，システムがある入力を受けてから，入力に対応した出力を行うまでに時間的な制約を求められる性質のことを指す．リアルタイム性を満たすためには，決められた時間内に処理が完了することを保証できること，つまり，処理にかかる時間を見積もることができることが必要となる．

決められた時間内に処理が完了しなかった場合，システム全体にとって致命的なダメージが生じるシステムのことをハードリアルタイムシステムといい，より厳しい時間制約が課せられたシステムである．逆に，時間制約を守れなかったとしても，システム全体に致命的なダメージを与えることはなく，処理自体の価値が減少するシステムのことをソフトリアルタイムシステムという．

本節では，リアルタイム OS における動的更新を行う際に発生する問題点について述べる．

3.1　リアルタイム性の低下

2 節で述べた手法では，対象の関数の動的更新を行うことは可能であるが，動的更新を行っている最中に，他のタスクのリアルタイム性が維持される保証はない．動的更新の処理を高い優先度のタスクとして実行したり，割り込み処理として行う場合，動的更新処理を優先的に行ってしまい，本来優先されるべき処理が後回しになってしまう可能性がある．動的更新処理によって他のタスクの実行が阻害され，高優先度タスクの時間制約を守れなくなる恐れがある．

3.2　信頼性の低下

動的更新の処理では，jmp 命令を更新対象となる関数の先頭に書き込む必要がある．しかし，他のタスクで関数の先頭の命令を実行中である時に，jmp 命令の書き込みが行われると，他のタスクでは jmp 命令を途中から実行

して異常終了する可能性がある．リアルタイム OS に限らず，マルチタスクで動作するシステムで動的更新を行う際にこの問題が発生する可能性があり，特に可変長命令である CISC 方式のプロセッサで発生する．**図 3** は，更新対象の関数の先頭に jmp 命令を書き込む前後の様子を表している．

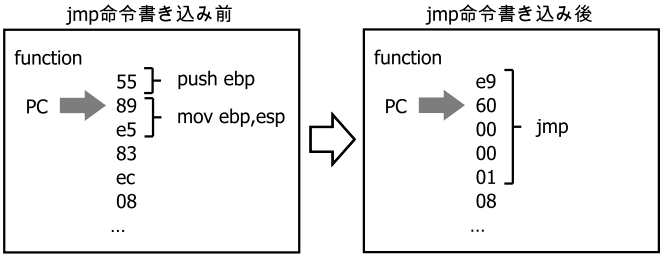

図 3 jmp 命令の途中実行

関数の先頭ではベースポインタの push 命令が実行されるが，push 命令の命令長は jmp 命令よりも短いので，push 命令の実行中に jmp 命令が書き込まれると jmp 命令を途中から実行してしまう．つまり，他のタスクが指すアドレスが，jmp 命令の書き込み範囲であった場合，不正な実行が行われる可能性がある．breakpoint 命令などを利用して，jmp 命令の書き込み範囲のアドレスを指さないように他のタスクの処理を一時停止させればこの問題は解決することができる．しかし，処理を一時停止させてしまうとリアルタイム性の維持ができない可能性がある．

また，動的更新を行っているタスクがスケジューラによって中断された場合も，不正な実行が行われる可能性がある．動的更新処理の優先度を低くした場合は，動的更新を行うタスクよりも優先度の高いタスクが実行可能状態になった場合や，割り込み処理が発生した場合，動的更新処理の途中であったとしても，タスクが切り換えられてしまい，動的更新処理が中断してしまう．この時，更新対象の関数の先頭に jmp 命令の書き込みを行っている途中だったとすると，メモリ上に存在する命令は不正なものとなってしまう可能性がある．その結果，他のタスクで更新中の関数を呼び出した際に不正な実行が行われることがありうる．

動的更新処理の実行によって不正な実行が行われ，システムの全体の停止に繋がってしまう可能性があることは，動的更新処理をハードリアルタイムシステムで行う場合，致命的な問題となる．システム全体が停止してしまうと，時間制約の厳しい処理を含め，全ての処理が実行されなくなる．その結果，リアルタイム性の維持ができず，システム全体に致命的な問題を引き起こすことになる．

4 動的更新機能をもつリアルタイム OS

本節では，3.1 節と 3.2 節で述べた問題点を解決し，リアルタイム OS において動的更新機能を実現する方法について述べる．

4.1 概要

本手法は関数単位での動的更新を行う．あらかじめ関数ごとに実行されない書き込み可能領域を用意しておくことで，jmp 命令の書き込み処理の中断による信頼性の低下を防ぐとともに，動的更新処理を最低優先度で実行できるようにする．動的更新処理を最低優先度で実行できるようにすることでリアルタイム性を維持し，ハードリアルタイムシステムに対応できる動的更新を行う．また，動的更新の処理を主に行うタスクを更新タスクと呼び，更新対象のリアルタイム OS を含むシステムにはネットワーク環境があるものとする．

4.2 更新タスク

更新タスクでは動的更新の処理を主に行う．他のタスクのリアルタイム性が維持できるように，更新タスクの優先度は低く設定し，他のタスクの処理を阻害しないようにする．

動的更新処理が開始されると，更新タスクは最初に拡張領域の確保を行う．この時確保される拡張領域は更新後の関数を格納するためのメモリ空間である．その後，更新タスクはネットワーク経由で更新後の関数のデータを受け取り，拡張領域に更新後の関数を書き込む．更新後の関数の書き込みが終わると，更新タスクは更新対象の関数の先頭部分に jmp 命令を書き込む．しかしこの時，更新タスクは 2 節で示した通常の動的更新手法の場合とは異なり，関数の先頭に jmp 命令を書き込まない．jmp 命令を書き込むのは，各関数の先頭部分に用意された書き込み可能領域である．書き込み可能領域は，実行されることない処理としてあらかじめ用意してある領域である．最後に，更新タスクが書き込み可能領域を実行されるように変更することで動的更新処理が完了する．

また，更新タスクの処理は全て低い優先度で実行される．動的更新によって，他のタスクの処理の実行が阻害され，リアルタイム性を低下させる原因にならないよう更新タスクの処理は低い優先度で実行されるように設定する．

4.3 書き込み可能領域

動的更新処理を低い優先度で実行した場合，3.2 節で述べた信頼性低下の問題が発生してしまうことがある．この問題は，jmp 命令の書き込み処理を最優先で実行することで解決することができる．しかし，最優先で実行してしまうと，書き込み処理の分だけ他のタスクの処理が遅延し，リアルタイム性の低下に繋がってしまう．そこで，jmp 命令を更新対象の関数の先頭に直接書き込むのではなく，各関数の先頭部分に用意された書き込み可能領域に対して jmp 命令の書き込みが行われるようにする．これにより，信頼性の低下を抑え，動的更新処理を最低優先度で実行できるようにする．**図 4** は，書き込み可能領域を利用した動的更新処理を表している．

書き込み可能領域は，書き込まれる jmp 命令と同じデータサイズのダミー命令から構成され，通常の処理では実行されない部分である．動的更新処理完了後に書き込み可能領域にはダミー命令ではなく，更新後の関数への jmp 命令が存在するようになる．動的更新処理が完了して初めて書き込み可能領域が実行されるようになり，jmp 命令が実行されるようになる．そして，更新対象の関数が呼び出されるたびに jmp 命令が実行され，更新後の関数が実行されるようになる．動的更新処理を行うには，このダミー命令か

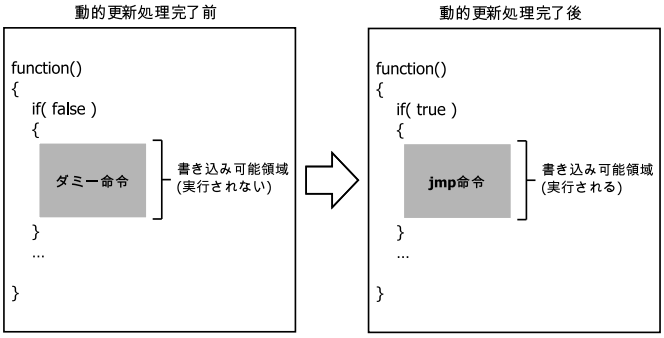

図 4 書き込み可能領域

らなる書き込み可能領域を各関数の先頭にあらかじめ用意しておくことが必要となる．

動的更新後に同じ関数を再び更新する可能性がある場合，2 つ目の書き込み可能領域を作成することで対応することができる．書き込み可能領域の直後に else if で囲まれた 2 つ目の書き込み可能領域を作成する．2 つの書き込み可能領域を交互に利用することで，2 回目以降の更新も行うことができるようになる．

書き込み可能領域に jmp 命令を書き込むようにすることで，実行されない部分のメモリを書き換えることになる．そのため，jmp 命令が途中から実行されたり，jmp 命令の書き込み処理が中断されたとしても不正な状態の命令を実行することはない．これにより，3.2 節で述べた問題を解決することができる．また，jmp 命令の書き込み処理を最優先で実行する必要性もなくなるのでリアルタイム性の維持も可能となる．

5 実装

提案手法を FreeRTOS[9] に実装した．本節では，jmp 命令の書き込み処理と書き込み可能領域の実装の詳細について述べる．

5.1 実装環境

実装環境を表にまとめたものを表 1 に示す．提案手法を実装する OS は FreeRTOS とした．リアルタイム OS を動作させる機器は Raspberry Pi Model B+[10] を使用した．

表 1 実装環境

OS	FreeRTOS 8.2.1
CPU	700 MHz / ARM1176JZF-S コア
Memory(SDRAM)	512 MB
ストレージ	microSD メモリーカードスロット
ネットワーク	10 / 100Mbps イーサネット

5.2 jmp 命令の書き込み処理の実装

4.3 節で述べた jmp 命令の書き込み処理を実装する．jmp 命令の書き込み処理を実装するにあたり，メモリ上に存在する命令を書き換えるために，アセンブリ言語で関数 StoreMem を作成した．この関数はメモリ上の指定アドレスに存在する命令を書き換える処理を行う関数である．

StoreMem 関数を利用して，更新対象の関数の書き込み可能領域に jmp 命令を書き込む処理を実装する．書き込み可能領域の先頭から順に，ベースポインタの push 命令，更新後の関数への jmp 命令，呼び出し元に戻るための pop 命令を StoreMem 関数を使って書き込む．この時，書き込む命令を更新後の関数への jmp 命令のみにした場合，関数の呼び出し元に戻ることができず，正常に動作しないことを確認している．そのため，push 命令及び pop 命令も jmp 命令と同時に書き込む必要がある．StoreMem 関数を利用して書き込んだ命令群を Listing1 に示す．literal pool は更新後の関数のメモリアドレスを書き込む部分である．

Listing 1 jmpAPI

```
push    {fp, lr}
add     fp,     sp,     #4
sub     sp,     sp,     #8
str     r0,     [fp, #-8]
ldr     r0,     [fp, #-8]
ldr     r3,     [pc, #8]
blx     r3
sub     sp,     fp,     #4
pop     {fp, pc}
(literal pool)
```

この命令群を StoreMem 関数を利用して，更新対象の関数の書き込み可能領域に書き込むことで，更新前の関数が呼び出される度に jmp 命令を経由して更新後の関数が呼び出されるようになる．jmp 命令の書き込み処理を実際に行うのは更新タスクである．更新タスクが更新後の関数をメモリ空間上に配置した後に，StoreMem 関数による Listing1 の命令群の書き込み処理が始まる．書き込み処理は低い優先度の処理として実行される．しかし，動的更新処理完了後まで実行されることのない書き込み可能領域に対して書き込み処理を行うので，3.2 節で述べた問題は発生しない．

5.3 書き込み可能領域の実装

書き込み可能領域はあらかじめ更新対象の関数の先頭部分に用意されている必要がある．そのため，FreeRTOS の API 関数の先頭部分にダミー命令を埋め込んだ．ダミー命令は if 文中に存在し，nop 命令で構成されている．5.1 節で述べた通り，ARM プロセッサを搭載している Raspberry Pi Model B+ を実装環境として使用しているので，命令は全て固定長である．そのため，書き込み可能領域として確保するメモリ空間は，書き込む命令数と同数の nop 命令をダミー命令として書き込むことで確保することができる．

また，書き込み可能領域を FreeRTOS の API 関数の先頭に挿入するプログラムを作成した．FreeRTOS のソースコードを調べて関数の先頭部分を検知し，if 文を含めた書き込み可能領域を挿入する．検知した関数に書き込み可能領域を挿入するかどうかはそれぞれ選択することができる．

書き込み可能領域に書き込まれている命令は動的更新処理が完了するまで実行されることはない．更新タスクが全ての動的更新処理が完了したことを検知して初めて書き込み可能領域の命令は実行されるようになる．一度実行されるようになると，jmp 命令を経由して常に更新後の関数が実行されるようになる．

6 実験

本節では，5.1 節の環境で実装した提案手法について行った実験について述べる．提案手法によって，動的更新手法をリアルタイム OS で行う際の問題点が解決できているかどうかを確かめる実験を行った．動的更新処理よりも高い優先度で動作するタスクを複数動作させている最中に動的更新を行っても，正しく動的更新が行われるか，高優先度タスクのリアルタイム性が維持されているかを実験によって確かめる．

テストプログラムとして，周期的に起動されてある一定の処理を行う高優先度のタスクを 3 つ作成した．そして，タスクが起動されてから処理が完了するまでの時間をそれぞれ計測する．この高優先度のタスクが実行されている最中に動的更新処理を実行し，動的更新によって高優先度のタスクの最大実行時間が大きく変動するかどうかを調べることで，リアルタイム性が維持されているかを確認する．また，動的更新処理を行うタスクは最低優先度のタスクとし，低優先度の処理として実行しても正しく動的更新が行われるかを確かめた．表 2 は実験の結果を表している．

表 2 高優先度タスクの最大実行時間

タスク名	最大実行時間 [ms]	
	動的更新の実行なし	動的更新の実行あり
タスク 1	38	38
タスク 2	77	77
タスク 3	193	193

優先度が高い順にタスク 1，タスク 2，タスク 3 となっている．動的更新処理を行っていない時と比べてタスクの最大実行時間が大きく変動することはなく，動的更新も正しく行われていることを確認した．これにより，提案手法によって動的更新をリアルタイム OS で行う際の問題点が解決できていることを確認した．

7 まとめ

本論文では，リアルタイム OS におけるソフトウェアの動的更新手法を提案した．提案手法は，動的更新処理の一つである実行バイナリの書き換え処理を更新対象の関数の先頭に対して行うのではなく，あらかじめ書き込み可能領域として関数の先頭に確保した領域に対して書き換え処理を行う．これにより，動的更新の処理によって発生するリアルタイム性と信頼性の低下を防ぎ，ハードリアルタイムシステムに対応することのできるリアルタイム OS の動的更新手法を行うことが可能となる．提案手法による動的更新を実際に行い，高優先度タスク実行下でも，リアルタイム性を低下させることなく正しく動的更新が行われ，リアルタイム OS における動的更新を実現できていることを確認した．

今後の課題としては，リアルタイム OS の根幹部分に対しても提案手法が適用可能であるかを確認する必要がある．第 6 節で行った実験ではリアルタイム OS の API を更新対象として実験を行っているので，スケジューラといった OS の根幹部分に対しても提案手法が有効であるかを確認できていない．そのため，提案手法がリアルタイム OS の根幹部分に対しても適用可能であるかどうかを確かめる実験を行う必要がある．また，書き込み可能領域の if 文の条件を変えた時に，投機実行の予測が外れる場合がある．この時にオーバーヘッドが発生し，リアルタイム性に影響する可能性がある．しかし，本論文ではこのオーバーヘッドを考慮していないので，リアルタイム性に影響があるか確かめる必要がある．

参考文献

[1] Jeff Arnold, M. Kaashoek:“Kspliece:Automatic Rebootless Kernel Updates”, Proceedings of the 4th ACM European conference on Computer systems, pp.187-198 (2009)

[2] 鵜飼 文敏 : livepatch, `http://ukai.jp/software/livepatch`. (2015/4/6 アクセス)

[3] 小澤 駿, 斎藤 彰一 : “プログラム領域の Copy on write を抑制した複数プロセスの動的更新手法の提案”, 情報処理学会研究報告 Vol.2014-DBS-160 No.2 Vol.2014-OS-131 No.2 Vol.2014-EMB-35 No.2 (2014).

[4] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, Pen-Chung Yew : “Plous:A Powerful Live Updating System”, Software Engineering, 2007. ICSE 2007. 29th International Conference on, pp.271-281 (2007).

[5] 佐伯 智之, 河口 信夫, 稲垣 康善 : “データ構造の変更に対応したソフトウェアの動的更新手法”, 第 6 回プログラミングおよび応用のシステムに関するワークショップ SPA2003 (2003).

[6] 佐伯 智之, 河口 信夫, 稲垣 康善 : “ユビキタス環境におけるモバイルエージェントを用いたソフトウェアの動的更新手法”, マルチメディア，分散，協調とモバイル (DICOMO) シンポジウム論文集, pp.593-596 (2003).

[7] 佐伯 智之, 河口 信夫, 稲垣 康善 : “ソフトウェアの動的更新におけるデータ変換のための XSL コード作成支援”, 第 7 回プログラミングおよび応用のシステムに関するワークショップ SPA 2004 (2004).

[8] Michael Wahler, Stefan Richter, Manuel Oriol : “Dynamic Software Updates for Real-Time Systems”, HotSWUp ’09 Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades (2009).

[9] FreeRTOS, `http://www.freertos.org/`. (2015/5/25 アクセス)

[10] Raspberry pi Model B+, `http://www.farnell.com/datasheets/1883763.pdf`. (2015/9/24 アクセス)