情報処理学会論文誌 Vol.59 No.2 762–774 (Feb. 2018)

ARM TrustZone for ARMv8-Mを利用した軽量メモリ保護RTOS

河田 智明[1,a)] 本田 晋也[1]

受付日 2017年5月25日, 採録日 2017年11月7日

概要：本研究ではARMv8-Mで追加されたTrustZone for ARMv8-Mを用いて，メモリ保護OSを低オーバヘッドで実現した．近年，IoTの台頭により組込みソフトウェアの安全性がますます重要となっている．安全性を確保する方法の1つがパーティショニングであり，メモリ保護はその重要な要素であるが，既存のメモリ保護OSではメモリ保護により無視できないオーバヘッドが発生する場合がある．本研究は一般的なメモリ保護OSで用いられるMPUを用いる代わりにTrustZoneのハードウェア支援機能を利用し，サービスコール呼び出しと割込み処理にともなうドメイン間遷移のレイテンシを低減させる方法を提案する．提案手法を用いることにより，既存のメモリ保護OSと同様のメモリ保護を，より低いオーバヘッドで実現できることを示した．

キーワード：組込みシステム，リアルタイムOS，TrustZone

Lightweight RTOS Utilizing TrustZone for ARMv8-M

Tomoaki Kawada[1,a)] Shinya Honda[1]

Received: May 25, 2017, Accepted: November 7, 2017

Abstract: With the emergence of IoT, safety is becoming ever more important matter in the field of embedded systems. Traditionally, embedded operating systems have employed memory protection mechanism to ensure the memory safety, which, however, had a tendency to lead to an intolerable overhead. In this paper, we present a real-time operating system that provides low-overhead memory protection. To achieve low-overhead memory protection, we propose a method that makes use of TrustZone in place of MPU which has been used in traditional operating systems, to significantly lower the inter-domain transition overhead involved in service calls and interrupt handling. Finally, we show that the proposed method achieves memory protection at much lower overhead, while maintaining the almost same level of safety as existing operating systems.

Keywords: embedded systems, real-time operating systems, TrustZone

1. はじめに

近年，IoTの台頭により，インターネットに接続する組込みシステムが増加し，組込みシステムの限られたリソース制約の中で安全性を確保することがますます重要化している．組込みシステムで安全性を確保する方法としては，パーティショニング機構を導入し，単一システム内のあるソフトウェアの故障が他の箇所に波及したり，情報が漏洩したりすることを防ぐことが一般的である．たとえば，IoT向けの組込みシステムでは，ネットワーク上に存在する脅威からシステムを保護するために，ネットワーク関連の機能を別のパーティションに分割することが望まれる．パーティショニングの方法の1つが，メモリ空間を保護対象ごとにドメインと呼ばれる単位で分割しドメイン間のメモリアクセスを制御するメモリ保護である．通常，メモリ保護をサポートしたOS（メモリ保護OS）ではシステムドメインとユーザドメインをそれぞれプロセッサの特権・非特権

[1] 名古屋大学大学院情報学研究科
Graduate School of Informatics, Nagoya University, Nagoya, Aichi 464–8603, Japan
[a)] tcpp@ertl.jp

モードに対応させることで保護を実現する．

MMU（Memory Management Unit）またはMPU（Memory Protection Unit）と特権・非特権モードを用いてパーティショニングを行うためのハードウェアをリングプロテクション機構と呼ぶ．MPUはメモリ保護のみをサポートするハードウェアである一方，MMUはアドレス変換もサポートしている．このための変換テーブルはメモリ上に持ち，さらにアドレス変換を高速に行うためのキャッシュであるTLB（Translation Lookaside Buffer）を持つことから，ページテーブルウォークの発生タイミングの予想が困難ため，MMUではリアルタイム性の確保が困難である．小規模な組込みシステム向けのメモリ保護OSでは仮想アドレスの必要性が薄いことから，MMUの必要性はなく，MPUのみをサポートしている．従来のメモリ保護OSでは，リングプロテクション機構を利用しメモリ保護を実現することが一般的である．

メモリ保護をサポートしていないOS（メモリ保護なしOS）では，サービスコールの呼び出しは関数呼び出しで実現することが一般的である．一方メモリ保護OSでは，ユーザドメインに所属するコードを非特権モードで実行するため，サービスコールを呼び出す際には実行モード遷移が必要となる．実行モード遷移はソフトウェア例外を発生させるトラップ命令を契機に行われる．トラップ命令が実行されると，実行モードが特権モードとなり，トラップハンドラが実行される．トラップハンドラの処理には，コンテクストの保存やサービスルコールの開始アドレスが登録された関数テーブルのルックアップと呼び出し等のソフトウェア処理がともなうため，実行モード遷移をともなう関数呼び出しは実行オーバヘッドが大きくなる．実際，一部のサービスコールの実行時間が2倍近く増加することが報告されている[1]．小規模な組込みシステムでは動作周波数が数十MHzであり計算機資源の制約が大きいため，このことは大きな問題である．サービスコールの一部をインライン化することでサービスコールの実行時間を抑える方法も提案されている[2]が，依然としてトラップの使用によるオーバヘッドは回避できておらず，また安全性を静的解析により担保しているため，セキュリティの面では不完全である．

また実行モード遷移をともなう関数呼び出しのオーバヘッドの大きさのため，メモリ保護OSの仕様によってはユーザドメイン所属の割込みハンドラをサポートしていない[1]．車載システム等ではユーザドメインで割込みハンドラを実行することが要求される[3]が，この場合タスク等を使用して実現する必要があり，システムドメイン所属の割込みと比べて起動オーバヘッドが大きい．これに加え，サービスコールの呼び出し方法の違いから，メモリ保護OSはメモリ保護なしOSとコードの共有が困難であり，その結果として，コードのメンテナンス性が悪化する．

実行モード遷移をともなう関数呼び出しを支援するハードウェア機構があれば，低オーバヘッドなメモリ保護OSの実現が可能である．具体的には，従来のメモリ保護OSのトラップハンドラの処理を高速化するハードウェアがあればよい．さらに，従来の関数呼び出しから少ない変更量で実行モード遷移をともなう関数呼び出しを実現できれば，メモリ保護なしOSに対する微小なコード変更のみでメモリ保護OSを実現でき，開発コストの低減につながる．

こうした機構を持ったプロセッサのアーキテクチャとして，ARM社は2015年に小規模組込みシステム向けのアーキテクチャであるARMv8-Mを発表した．ARMv8-Mに追加された主要な機能の1つが，TrustZone for ARMv8-Mであり，前述の特徴を持つハードウェア支援機構を有している．TrustZone for ARMv8-Mは詳細な仕様が公開されてから日が浅く，使用方法については検討の余地が大きい．特に，TrustZone for ARMv8-Mをベースとしたメモリ保護OSの事例は現時点では知られていない．そこで，メモリ保護OSを実現するための代替的手法としてTrustZone for ARMv8-Mを利用することを検討した．

本研究では，メモリ保護なしOSをベースとしてTrustZone for ARMv8-Mによりメモリ保護対応する方法について検討を行った．検討過程で，TrustZone for ARMv8-Mを使用したシステムの構成として考えられるものを列挙し，それらの性質について定性的な比較を行った．列挙した構成の1つであるシングルバイナリイメージ方式を，メモリ保護機構を持たないRTOSの1つであるTOPPERS/ASP3に適用し，メモリ保護機能を持つASP3+TZ（TrustZone for ARMv8-M対応のASP3）を実現した．TrustZone for ARMv8-Mをベースとしたメモリ保護OSの前例は報告されていないため，ASP3+TZに対し評価実験を行い評価値を示した．また，シングルバイナリイメージ方式のメモリ保護OSが従来のメモリ保護OSよりも低いオーバヘッドでメモリ保護を実現できることを示した．TrustZone for ARMv8-Mの実行モード遷移の性能評価はまだ報告されていないが，この評価実験により定量的評価値を示した．最後に，ベースとしたASP3からのコード変更量を評価することで，小規模のコード修正によりメモリ保護が実現できることを示した．

2. ARMv8-Mアーキテクチャ

ARMv8-Mアーキテクチャ[4]は，ARM社が2015年に発表した，プロセッサアーキテクチャである．ARMv8-Mの主な特徴は，従来からあるARMv7-Mの機能に加え，ARMv8-M Security Extensions（場合によってはCortex-M Security Extensionsと呼ばれる場合もある．以下CMSEと略す）がオプション機能として追加されたことである．CMSEはコンセプト的には以前から提供されていたARMv7/8-AのTrustZone（以下TrustZone-Aと呼ぶ）に

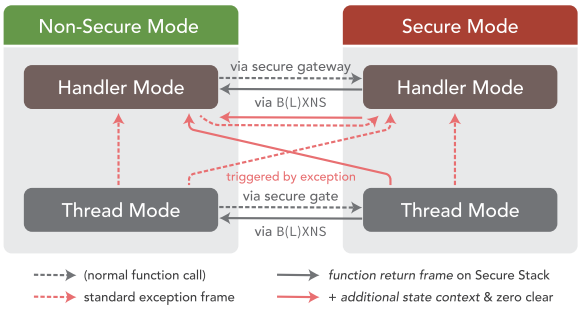


図 1 ARMv8-M の実行モード遷移
Fig. 1 Possible transitions between ARMv8-M processor states.

基づいた技術であり，システムをハードウェアレベルで Secure/Non-Secure に分離することで，Non-Secure がアクセス可能な領域を制限し，高水準のセキュリティを確保できる．このため，CMSE は TrustZone for ARMv8-M と呼ばれる場合がある．

TrustZone-A は文献 [5] や [6] 等多くの用途がすでに報告されているが，アーキテクチャ的な実現方法は CMSE と TrustZone-A ではまったく異なり，適用できない．特に，TrustZone-A では CMSE にあるような実行モードの遷移を支援する機構は提供していない．

以降，ARMv7-M/ARMv8-M に共通な事項については，ARM-M として説明する．

2.1 実行モード

ARM-M は，Handler/Thread の 2 つの実行モードを持つ．リセット直後や，通常のプログラムは Thread Mode で動作する．Handler Mode は例外/割込みハンドラで使用される実行モードであり，例外/割込みがアクティブになると Handler Mode へ遷移してから，対応するハンドラが起動する．その後，ハンドラの実行が完了して元の制御フローに戻る際に Thread Mode への復帰が行われる．また，プロセッサモードとして特権モードとユーザ（非特権）モードを持ち，Thread Mode は両方のモードで実行することが可能である．Handler モードは特権モードで動作する．

CMSE に対応した ARMv8-M プロセッサでは特権・ユーザモードの区別を残したまま，さらに Secure/Non-Secure という新たな軸が加わる（**図 1**）．Non-Secure は権限が制限された実行モードであり，アクセス可能な領域（メモリ，ペリフェラル等）がハードウェアレベルで制限される．Secure/Non-Secure 間は CMSE で追加された secure gateway や `B(L)XNS` 命令を利用することにより，高速かつ安全に遷移することが可能である（2.3 節で詳しく述べる）．さらに，割込みハンドラ起動時に対応付けられた実行モードに自動的に遷移し，その際にセキュリティの確保に必要な処理をハードウェア側で自動的に行う機能を備えている（2.4 節で詳しく述べる）．

2.2 メモリ保護

ARM-M はメモリ保護機構として，MPU をサポートしており，MMU はサポートしていない．加えて，CMSE では SAU（security attribution unit）および IDAU（implementation defined attribution unit）により，各メモリアドレスのセキュリティ属性を (a) Secure（Secure Mode のみアクセス可能），(b) Non-Secure Callable（Non-Secure からは制限付きでジャンプ可能．2.3 節で詳しく述べる），(c) Non-Secure（つねにアクセス可能）のうち 1 つに設定できる．SAU による属性の指定方法は MPU と類似しており，各リージョンの上界・下界アドレス・属性をレジスタを経由して設定する．

TT（Test Target）命令は指定されたメモリアドレスのセキュリティ・MPU 属性の判定を行う命令であり，Non-Secure から渡されたポインタの正当性の検証に使用できる．判定対象は単一のメモリアドレスであるが，結果に含まれる SAU，IDAU の対応するリージョンの番号を利用することで，アドレス範囲のアクセス権を判定できる．

2.3 関数呼び出し

CMSE は Secure/Non-Secure 境界を越えて関数呼び出しを行う際に，モードの遷移を安全に行うための機能を有している（図 1）．

Secure から Non-Secure への関数呼び出しは `BLXNS` 命令を用いて行う．この際，通常の関数呼び出しで用いる `BLX` 命令を使用した場合には SecureFault 例外が発生するため，意図しない遷移が発生する可能性を軽減できる．さらに，通常の関数呼び出しでは `LR` レジスタにリターン先の番地が格納されるが，`BLXNS` 命令による呼び出しの場合，代わりに `FNC_RETURN` という特殊な値が格納され，同時に Secure 側のスタックにリターン先番地と PSR が格納される．この `FNC_RETURN` を使用してリターンすることで，プロセッサが Secure モードの復帰に必要な処理を行う．この際，PSR の値等に異常があれば SecureFault が発生する．

Non-Secure から Secure への関数呼び出しを行う場合は `BLX` 命令を用いればよい．しかし，任意の Secure 領域内の場所にジャンプできてしまうとセキュリティ上の危険があるため，ジャンプ先は以下の条件を満たす必要があり，それ以外の場合は SecureFault 例外が発生し Secure 側に通知される．

(1) ジャンプ先が SAU および IDAU により Non-Secure Callable 領域として設定されている．
(2) ジャンプ先が SG（Secure Gateway）命令である．

以上の条件が満たされた場合に限りジャンプは成功し，Secure モードへの遷移が行われる（**図 2**）．なおリターンする際には，リターン先は Non-Secure 領域のコードであるため，`BXNS LR` 命令を使用する必要がある．この際も `LR` が指す先が Non-Secure 領域でなければ，やはり SecureFault

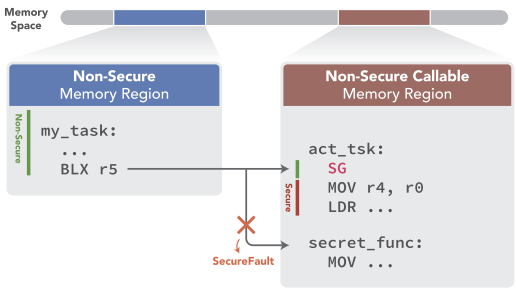


図 2 Non-Secure Callable 領域の関数を呼ぶ際の動作
Fig. 2 Calling into a Non-Secure Callable region.

が発生する．

2.4 例外

ARM-M では，例外[*1]発生時には現在実行中の処理が中断され，例外番号に対応する例外ベクタテーブルがハードウェアにより読み出され，それが指すハンドラに制御が移る．

CMSE をサポートした ARMv8-M プロセッサは，Secure/Non-Secure で独立した例外ベクタテーブルを持つ．すべての例外のうちハードウェア割込みに関しては，割込みを受付けると，NVIC（Nested Vectored Interrupt Contoller）はレジスタの設定に基づいて最初に割込みを Secure/Non-Secure のいずれかに振り分け，対応する例外番号の pending ビットをセットする．CPU 例外は本質的にいずれかのモードと対応付けられているので，振り分けは必要ない．

例外発生時には，図 1 に示しているように任意の実行モード遷移が発生する可能性がある．特に，Secure から Non-Secure への遷移（図 1 で実線で示した遷移）は割込みを無効にしていない限り Secure 側のコードの任意の箇所で起こりうるため，意図しない形で Non-Secure 側のコードが Secure 側の動作に影響を与えたり，Secure 側の情報が漏洩したりすることを防ぐための仕組みをハードウェアで提供している．

Secure から Non-Secure への遷移が起きると，通常保存される `r0-r3`，`r12` に加えて，残りの汎用レジスタがすべて Secure 側のスタックに保存される．この追加部分は additional state context と呼ばれる．さらに，その後にすべての汎用レジスタがゼロクリアされ，Secure 側の状態が Non-Secure 側の例外ハンドラから読み取られないようにする．Additional state context の先頭には，Secure から Non-Secure への遷移が起きたことを示すマジックナンバが付加される．

例外発生時には復帰先の状態に関する情報が含まれる値（`EXC_RETURN`）が `LR` レジスタに書き込まれる．CMSE ではこの `EXC_RETURN` に復帰先のモードの情報も含まれ，割込みからリターン（`BX LR`）する際にはこれに基づいて適切なモードへの復帰処理が行われる．このとき，Non-Secure から Secure に復帰する際，スタック上にある additional state context のマジックナンバの値の検証が行われる．このマジックナンバは Secure 側のプログラミングエラーがない限りは例外によって Secure から Non-Secure へ遷移したとき以外には存在しえないため，これによって Non-Secure コードが Secure→Non-Secure 遷移を「偽装」することを防ぐ．

[*1] ARM-M では，CPU 例外とハードウェア割込みの両方が含まれる．

3. CMSE を用いたメモリ保護 OS

3.1 従来のリングプロテクション機構を用いたメモリ保護 OS

TOPPERS/HRP2 [1] 等のリングプロテクション機構を用いた従来のメモリ保護 OS は，リンカが出力した単一のバイナリでシステム全体を構成するモデルを採用している（**図 3** 従来方式のメモリ保護 OS）．この方式ではユーザドメインに所属するコードを MPU による保護が有効な非特権モードで実行することで，メモリ保護を行う．ユーザドメインからはトラップにより呼び出されるハンドラを経由することで，カーネルが提供するサービスコールを特権モードで呼び出す．

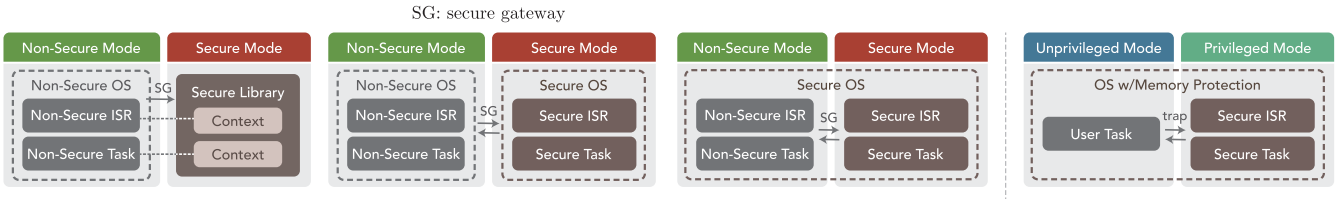
3.2 CMSE を用いたシステム構成方法

CMSE を用いてパーティショニングを実現する場合，以下のシステム構成方法が考えられる（図 3）．

セキュアライブラリ（SL）方式

Secure 側はライブラリ形式として機能を提供し，その実行コンテキストの管理は Non-Secure 側で行う方式である．Secure 側のコンテキストは Non-Secure 側から明示的に要求して切り替える必要がある．この方式は ARM CMSIS（Cortex Microcontroller Software Interface Standard）で想定している方式である．CMSIS では Non-Secure 側から Secure 側のコンテキストを切り替えるための共通 API を定義し，Non-Secure 側の OS がこの API を使用することを推奨している [7]．

この方式は，プログラムに機密情報が含まれる場合に，想定される開発者以外がその情報にアクセスするリスクを軽減したい場合に適している．たとえば，プログラムに暗号化鍵を埋め込む場合，暗号化機能のみを Secure 側で実装し，それ以外のコードは Non-Secure 側で実装し，Secure 側が提供する API を介して暗号化機能を利用する．Non-Secure 側の開発者はたとえデバッガを介しても Secure 側のコードにはアクセスできないため，暗号化鍵にアクセスできる開発者の数を最小限にでき，鍵が流出する危険を大きく減らせる．



(a) セキュアライブラリ方式　(b) デュアル OS 方式　(c) シングルバイナリイメージ(SBI)方式　(比較) 従来方式のメモリ保護 OS

図 3 CMSE を用いたシステム構成方法と，従来のメモリ保護 OS の比較
Fig. 3 System architectures for CMSE and those of traditional OSs with memory protection.

デュアル OS（D-OS）方式

Non-Secure 側と Secure 側のそれぞれで独立した OS を実行する方式である．この方式はセキュアライブラリ方式と類似しているが，Secure 側でも OS を動作させるため，それぞれの領域が実行コンテキストを独立して管理できる．

Secure 側でも OS を持つことにより，Secure 側で行える処理の幅は広がるが，両方の領域で OS を実行させるために，OS のコードが 2 重に必要となる．また，タスクの優先度の柔軟性が低く，混合スケジューリングを行うには OS の修正を要する．OS の境界を越えてサービスコールを発行することは簡単ではなく OS 間通信が必要であり，Secure 側と Non-Secure 側が密接に連携することが求められるシステムには向かない．

この方式は ARM-A 系アーキテクチャ向けの TrustZone を利用することで RTOS と同時に Linux 等の汎用 OS を実行する技術である SafeG [5] と類似している．しかし，ARM-M 系のプロセッサが使用される場面では Linux のような汎用 OS を使用できるほどの計算機資源の余裕がない場合が多く，これを使用するモチベーションは高くない．

シングルバイナリイメージ（SBI）方式

本研究で提案する SBI 方式は，単一の OS・バイナリが Non-Secure と Secure の両側の制御を行う方式である．前に述べた方式と大きく異なる点は，Non-Secure と Secure の両方で実行される単一のバイナリをリンカで出力する点であり，セキュアライブラリ方式が想定するような，開発チーム内での機密情報の厳密な管理には向かない．

この方式は 3.1 節で述べた，リングプロテクションを用いた従来のメモリ保護 OS に類似している．他の方式と比べると単一のバイナリのみを開発するため開発効率が良く，さらに 1 つの OS が両方の領域を管理するため，柔軟なスケジューリングが可能である．セキュアライブラリ方式と比較すると，セキュアライブラリ方式では Non-Secure 側の OS がコンテキストスイッチするごとに Secure 側が提供する API を呼び出して Secure 側のコンテキストを切り替える必要があるが，この方式では OS が両方のコンテキストを直接操作できるため，コンテキストスイッチによる遅延が少ない．また，デュアル OS 方式と比較しても，OS を 2 重に実行することによるオーバヘッドがない．

表 1 CMSE を用いたシステム構成方法の性質の比較
Table 1 Properties of system architectures for CMSE.

	SL	D-OS	SBI
Secure/Non-Secure の開発独立性	✓	✓	
メモリオーバヘッド	✓		✓
実行時オーバヘッド		†	✓
開発効率			✓

† 独立して動作する分には支障はないが，Non-Secure と Secure の境界を越えてサービスコールを発行するためには OS 間通信が必要であり，オーバヘッドが生じる．

一方で，SBI 方式では OS は両方の領域を考慮した処理を行う必要があるため，他の方式で OS を動作させる場合よりも，オーバヘッドが増加する可能性がある．たとえば，Non-Secure 側でのサービスコールの呼び出しは，Secure Mode の切替えが必要となるため，実行時間への影響が考えられる．しかし，CMSE ではこうしたモードの切替えをともなう呼び出しをハードウェアの支援により高速化する機構を備えているため，同じことを従来のメモリ保護 OS よりもはるかに少ないオーバヘッドで実現できると思われる．

これらの性質を Secure/Non-Secure の開発独立性，メモリオーバヘッド，実行オーバヘッド，および開発効率としてまとめて比較したものを**表 1** に示した．SBI 方式は Secure/Non-Secure の両側を支配する単一のバイナリをリンカで生成する必要があるため，他の方式と比べると Secure/Non-Secure の開発独立性は劣る．しかし，これ以外の観点ではすべての点で優れているため，この性質が重要でない状況においては最適な選択肢であるといえる．

4. 設計

本章では，SBI 方式を用いた CMSE ベースの OS の設計について述べる．

本設計の目標は以下のとおりである．

- メモリ保護なし OS をベースとして開発するにあたって，相対的なオーバヘッドの増加分をできるだけ小さくする．
- コードの変更量を最小限にとどめる．

これを実現するために，ハードウェアによる自動的なレ

ジスタの保存・復帰等，CMSE が提供する機能を積極的に活用することを方針として設計を行った．

4.1 ドメイン

アプリケーションの故障の影響範囲を最小限にするために，メモリ保護 OS では一般的なシステムドメインとユーザドメインの概念を導入する．システムドメインはつねに Secure Mode で動作し，ユーザドメインのコードは Non-Secure Mode で動作させることで，ドメイン間の分離を実現する．ユーザドメインで実行するコードは Non-Secure Mode で実行され，アクセス可能なメモリ領域は SAU および IDAU により定義されるセキュリティ属性によって制限され，操作可能なオブジェクトも制限を受ける．

ユーザドメインについては，複数個存在させ，ドメイン間のアクセスを制限したり，ドメインごとに個別にペリフェラルやカーネルオブジェクトへのアクセス権を設定したりすることも考えられる．パーティショニングを行う目的の 1 つはアプリケーションの故障による影響を局在化させることであり，機能ごとにドメインを細かく分割することで，故障発生時の影響を最小限にし，故障の分析に要する期間を短縮することができる．こうした考え方は基本的には旧来のメモリ保護 OS でも変わらないが，CMSE ベースのシステムであれば実行モード遷移のオーバヘッドの低さのため，より積極的にドメインの分割を行えるようになる可能性がある．しかし，複数ユーザドメインに対応するには SAU または MPU の設定変更を要し，オーバヘッドとのトレードオフがあるため，単一ユーザドメインも選択肢としては有効である．

カーネルはシステムドメインと同じく Secure Mode で動作し，ユーザドメインからのサービスコール呼び出し時には必要に応じて secure gateway を利用して実行モード遷移を行う（4.2 節で詳しく述べる）．

4.2 サービスコール

メモリ保護 OS においては，特権モードを変更するためにトラップを使用することでユーザドメインから呼び出し可能なサービスコールを実現することが一般的であり，これがオーバヘッドの増加の原因の 1 つとなっている．提案する OS では，トラップの代わりに secure gateway を経由することで実現する（図 4）．Secure gateway では，ポインタ・操作対象オブジェクトが呼び出し元ドメインから操作することが許可されているか，等の引数の検査を行ったあと，サービスコールの本体を呼び出す．

4.3 ユーザ割込みハンドラ

ユーザ割込みハンドラ（ユーザドメイン所属の割込みハンドラ）は，割込み応答時間がシステムモードのものと比べて悪化するという問題がある．従来のリングプロテクション機構では，割込みを受付けた際には特権モードに自動的に切り替わり，制御が割込みハンドラに移る．このため，ユーザ割込みハンドラを実現するためには，カーネル内の割込みハンドラでいったん割込みを受付けた後，カーネルによってコンテキストの保存や MPU の設定を変更した後，割込みからのリターンによって非特権モードへの切替えを行い，ユーザ割込みハンドラを実行する必要がある．AUTOSAR OS 仕様 [8] では OS application 内で割込みハンドラを生成する方法について定義しているものの，こうした理由から，大きな処理遅延が生じる．このためユーザ割込みハンドラを元からサポートしない OS も多く（e.g., 文献 [1]），この場合はタスク等を利用して擬似的に実現する必要があった．

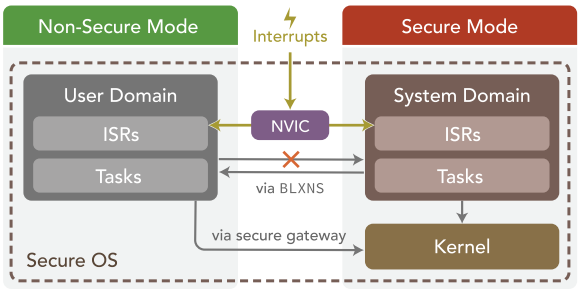


図 4 SBI 方式の OS の概念図
Fig. 4 The concept of operating system based on the SBI architecture.

CMSE は Secure/Non-Secure で独立した割込みベクタテーブルを持ち，現在の実行モードがどちらであるかにかかわらず，任意の実行モードの割込みハンドラを非同期に起動することが可能である．これを利用し，起動時に Secure 側のコードによる介在が不要な，ユーザ割込みハンドラを実現できると考えられる．

割込み優先度に関してはすべてのドメインで 1 つの空間を共有する．すなわち，システムドメイン内のものより優先度の高いユーザドメイン割込みハンドラも作成可能である．

4.4 特権関数と非特権関数

メモリ保護 OS では，システムドメインの関数にユーザドメインからアクセスする機能をユーザドメインに提供していることが多い（特権関数）．SBI 方式で CMSE を使用したメモリ保護 OS でも secure gateway を経由することで同様のことが可能であるが，それに加え，その逆となる，ユーザドメインの関数（非特権関数）をシステムドメインから CMSE を活用し高速に呼び出すことも考えられる．この場合，ユーザドメインで発生した例外を安全にシステムドメインで処理する方法が必要となる．

5. 実装

3.2 節で述べた SBI 方式のメモリ保護 OS と従来のメモ

リ保護 OS で実行時オーバヘッドを比較し，またこの方式の実現に必要なコード実装量を評価するために，4 章で述べた設計に基づいて，従来のメモリ保護なし OS である ASP3 をベースとして CMSE ベースのメモリ保護 OS の実装を行った．この実装を ASP3+TZ（TrustZone for ARMv8-M 対応の ASP3）と呼ぶ．実装を行った範囲は基本的な部分のみであり，複数ユーザドメインのサポートと，セマフォ等のカーネルオブジェクトに対するアクセス制御，および非特権関数は実装を行っていない．これらの機能については，今後実装を検討している．

ASP3+TZ の具体的な実装項目は以下のとおりである．

5.1 サービスコール用 Secure Gateway の実装

多くのサービスコールはカーネル内部の状態を操作するため，Secure Mode で実行しなければならない．したがって，システムドメインから呼び出す場合は従来どおり通常の関数呼び出しで十分である一方，ユーザドメインでこれらを呼び出すためには Secure Mode への切替えが必要である．CMSE では Non-Secure Callable リージョン内の SG 命令へジャンプすることにより，この切替えを自動で行える．切替えの際，LR（リンクレジスタ）の最下位ビットが 0 に変更され，リターン先が Non-Secure であることを表す値となる[*2]．リターンする際には，リターン先が Non-Secure であるため，BXNS 命令を使用する．

Non-Secure Callable リージョンに配置した，先頭に SG 命令を持つ，サービスコールへの橋渡しを行う関数を secure gateway と呼び，対応するサービスコールの名前に ns_というプレフィックスを付加したものをその名前とした（ユーザドメインからはこの名前で呼び出す）．**図 5** は secure gateway の例である．Secure gateway の SG 命令により Secure Mode への切替えは自動的に行われるものの，ソフトウェアによりいくつかの追加の処理が必要である．

ポインタの検証 サービスコールにはポインタを渡すものが一部存在する．サービスコールの本体は Secure Mode で実行されるため，引数として Secure リージョン内へのポインタを指定することにより許可されていない領域の値を不正に操作することが可能となってしまう．このため，CMSE が提供する TT 命令により，ポインタが指す領域のアクセス権情報を取得し，アクセスが禁止されている場合はエラーを返す処理が必要ある（図 5 右）．

ポインタの検証についてはサービスコールの本体側で実装することも考えられるが，呼び出し元ドメインにより分岐が必要なため，secure gateway での実装にとどめた．

```
ns_act_tsk:
    sg
    push {lr}
    bl act_tsk
    pop {lr}
    mov r1, #0
    mov r2, #0
    mov r3, #0
    mov ip, #0
    msr apsr_nzcvq, r1
    bxns lr
```

左：ポインタを渡さないサービスコールの secure gateway

```
ns_wai_flg:
    sg
    push {r4, lr}
    tst r3, #3
    tt r4, r3
    bne sg_perm_check2_fail
    tst r4, #TT_RESP_NSRW
    beq sg_perm_check2_fail
    bl wai_flg
    pop {r4, lr}
    mov r1, #0
    mov r2, #0
    mov r3, #0
    mov ip, #0
    msr apsr_nzcvq, r1
    bxns lr
```

右：第 4 引数の 32 ビット整数へのポインタのアクセス権を TT 命令により検証している．

図 5 Secure gateway の例
Fig. 5 An example of secure gateway.

レジスタのクリア Secure Mode と Non-Secure Mode でバンクされているレジスタは SP 等のごく一部に限られる．このため，サービスコールから復帰した後そのまま Non-Secure 側にリターンすると，レジスタに残った値から機密情報が漏れる恐れがある．図 5 の両方の例で r0-r3 をクリアしているのはこれを防ぐためである．

5.2 タスクへのスタックの追加

ARM-M アーキテクチャでは，スタックポインタは Handler/Thread モードでバンクされており，それぞれ MSP（Main Stack Pointer），PSP（Process Stack Pointer）と呼ばれる．CMSE では，スタックポインタはさらに Secure と Non-Secure でバンクされており，実行モード遷移時には，対応するモードのスタックポインタが自動的にアクティブになる．たとえば，Non-Secure Mode のタスクからサービスコールを呼び出した場合には，secure gateway に入る際に自動的に Secure 側の PSP がアクティブとなる．

Secure/Non-Secure のスタックポインタを同一のものにすることも可能であるが，セキュリティ上危険である．サービスコールの実装では一時変数をスタックに配置する場合があり，また Secure Mode での割込み発生時は Secure 用のスタックポインタに例外フレームが保存される．これらの値を Non-Secure Mode からアクセス可能な場所に配置してしまうと，他の Non-Secure Mode で実行するタスクによって不正にアクセスされる危険性が生じる．SAU によって Secure Mode での実行中は Non-Secure Mode から当該スタック領域へのアクセスを禁止するように制御したとしても，secure gateway に入った直後に隙が生じるため，完全な保護は不可能である．

以上の理由から，同じスタック領域を Non-Secure/Secure

[*2] この処理が自動で行われるのは，LR を Secure リージョンの任意の関数に変更して SG を呼び出す攻撃が考えられるためである．LR を最下位ビットが 0 のアドレスしておけば，BXNS 命令でリターンする際は最初に Non-Secure Mode への切替えが行われるため，Secure リージョンにジャンプしても例外が発生する．

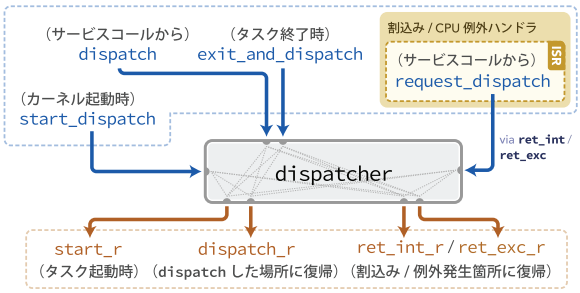


図 6 ASP3 のディスパッチャの出入り口
Fig. 6 Entries and exits of ASP3's dispatcher.

で安全に共用する方法はなく，Non-Secure/Secure で独立したスタック領域を割り当てる必要がある．

独立したスタック領域の割当てを行うために，カーネルコンフィギュレータに対して変更を行った．また，タスクのコンテクストを初期化する関数には，各モードのスタックポインタを対応するスタック領域に初期化する処理を追加した．

5.3 ディスパッチャ

ディスパッチャは，実行中のタスクを切り替える処理である．本節では，ASP3+TZ におけるディスパッチャの実装について説明する．まず，5.3.1 項では ASP3 の仕様で定めるディスパッチャの扱いについて説明し，次に 5.3.2 項では ARM-M 向け ASP3 の実装について説明する．最後に，5.3.3 項で ASP3+TZ の実装に際して行ったディスパッチャに対する修正について述べる．

5.3.1 ASP3 仕様での扱い

サービスコールが呼び出されたとき等，必要に応じて OS 内部の関数が次のタイミングで呼び出される（図 6）．

`start_dispatch` カーネル起動時に，カーネルの初期化処理から呼び出される．

`dispatch` タスクコンテキスト，たとえばサービスコールにより，高優先度のタスクが起動したり，現在のタスクが待ち状態が入ったりした状況で呼び出される．

`request_dispatch` 非タスクコンテキスト，たとえば割込みハンドラが実行中に，タスクの切替えが必要になった場合に呼び出される．この場合，実際のディスパッチは割込みハンドラの実行が完了し，タスクコンテキストに復帰する際（`ret_int`，`ret_exc`）に行われる．したがって，この関数自体は直接ディスパッチャを呼び出さない．

`exit_and_dispatch` タスクコンテキストにおいて，現在のタスクが終了中（すなわち，コンテキストを保存する必要がない場合）に呼び出される．

それぞれ具体的な処理内容は異なるが，最終的にはほとんど共通の方法でディスパッチャを起動する．ディスパッチャの最終的な目的は，(a) 現在実行中のタスクの実行番地/スタックポインタを保存し，(b) 移行先のタスクの実行番地/スタックポインタを復元することである．

ディスパッチャの行き先は，遷移先タスクの前回中断時の状況に応じて，以下のうちのいずれかとなる．

`start_r` タスクのエントリポイントを呼び出す．

`dispatch_r` そのタスクが以前 `dispatch` した場所に復帰する．

`ret_int_r` そのタスクが以前割込みにより中断された場所に復帰する．

`ret_exc_r` そのタスクが以前 CPU 例外により中断された場所に復帰する．

5.3.2 ARM-M 向け ASP3 の実装

ARM-M では OS のコンテクストスイッチのために使用されることを想定した，PendSV という種類のソフトウェア割込みをサポートしている．ASP3 の ARM-M 向け実装では，他の ARM-M 向けの RTOS と同様に基本的にはこの PendSV をディスパッチャの実装に使用する．

ARM-M のソフトウェア割込みをこの用途で使用することの利点は，コンテクストの保存・復元処理の大部分をハードウェアの割込み処理機構で実現できることである．ディスパッチャの処理として，レジスタの保存，ディスパッチャの復帰先の関数（`dispatch_r` 等）でレジスタの復元を行う必要がある．しかし，ARM-M では一部レジスタの保存・復元は割込み出入り口で自動的に行われるため，残りのレジスタのみディスパッチャで保存すればよい．特に，例外からの復帰関数（`ret_int_r`，`ret_exc_r`）に至ってはハードウェアで必要なレジスタはすべて保存するため，ソフトウェア実装はまったく必要ない．タスク起動時の `start_r` も同様に必要なレジスタがハードウェアで設定された後，エントリポイントが実行されるため，ソフトウェア実装が必要ない．

ディスパッチャで保存・復元を行うのは，具体的にはスタックポインタ（PSP）と，前回実行中断時の `LR` の値（`EXC_RETURN`），およびすべての callee-saved レジスタ（`r4-r11`，`s16-s31`）である．それ以外の，プログラムカウンタおよび caller-saved レジスタ（`r0-r3`，`s0-s15`）は，ディスパッチャに出入りする際に例外処理ハードウェアが自動的に PSP/MSP の場所から保存・復元する．

図 7 に ARM-M 用のディスパッチャの動作例を示す．この例では，ディスパッチャの起動は `dispatch` により行っている．ARM-M の `dispatch` では PendSV を発生させる（図 7 左）．ディスパッチャ用の例外ハンドラが起動する際，ハードウェアはレジスタの一部をスタックに保存するとともに，これを復元するのに必要な情報（`EXC_RETURN` と呼ばれる）を `LR` レジスタに書き込む．

このディスパッチャ用の例外ハンドラは，復帰先を変更できるという点で一般的な例外ハンドラとは異なる存在である．通常の例外ハンドラの場合はこの `EXC_RETURN` を

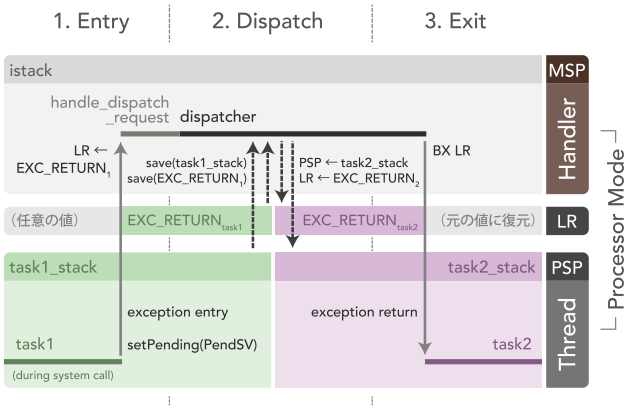


図 7 ARM-M におけるディスパッチャの動作例
Fig. 7 Example control flow of dispatcher for ARM-M.

そのまま PC レジスタに書き込む（`BX LR`）ことでハードウェアが自動的に復帰処理を行う．ディスパッチャ用の例外ハンドラでも復帰の方法は同様であるが，復帰する前にスタックポインタと LR レジスタの値を書き換え，遷移先のタスクに復帰するように変更する．これにより，`BX LR` を実行したときにハードウェアがこのタスクのコンテクストの復帰処理を自動的に行い，このタスクが前回実行を中断した箇所に制御が移る．

通常の例外・割込みハンドラでディスパッチが必要になった場合は，ディスパッチ処理は例外・割込みハンドラが終了するまで遅延される（遅延ディスパッチ）．遅延ディスパッチを実現するため，PendSV は他の例外・割込みハンドラよりも低優先度に設定されている．この結果，他の例外・割込みハンドラが実行中にサービスコールを呼ぶことで PendSV が発行されても，現在実行中のハンドラの実行が終了するまで PendSV のアクティブ化は遅延される．これを利用すると，`request_dispatch` では PendSV を発行することで，当該割込みハンドラの実行が終了した後に，PendSV がアクティブになり，ディスパッチが行われる．ARM-M の割込みコントローラは多重割込みに対応しているが，この場合も待機中の割込みの処理がすべて完了した後に PendSV がアクティブになる．

5.3.3 ASP3+TZ での変更点

Non-Secure Mode でのタスクの実行をサポートするためには，CMSE で追加されたレジスタ等の状態の保存・復元をディスパッチャで行う必要があるが，追加されたのは現在の実行モード（Secure/Non-Secure）と Non-Secure 用のスタックポインタだけであり，必要な変更はごくわずかである．

実行モードに関しては例外発生時に生成される `EXC_RETURN` の値に元の実行モードの情報が含まれる．タスクの実行が再開される際にはこの値をそのまま使用して例外ハンドラを抜けるため，ソフトウェア側で追加の実装を行わなくても自動的に元の実行モードに復帰する．スタックポインタに関しては Secure の PSP に加えて Non-Secure の PSP も保存するだけで十分である．

各ディスパッチャの入り口について，Non-Secure Mode との関係を分析してみると，次のようになる．

`start_dispatch` カーネル起動時に呼び出されるため，Non-Secure Mode から呼び出される可能性はない．

`dispatch` サービスコール内で呼び出されるため，Non-Secure Mode から直接呼び出される可能性はない．Non-Secure Mode から呼び出されたサービスコールでディスパッチが発生する場合，サービスコール前後で実行モード遷移が行われるため，`dispatch` が呼び出された時点では Secure Mode である．

`request_dispatch` `dispatch` と同じ理由で Non-Secure Mode から直接呼び出されることはないものの，この関数自体はディスパッチャを直接呼び出すものではなく，現在実行中の割込みハンドラ終了時にディスパッチャが起動されることを要求するものである．このため，これを契機としてディスパッチャが起動した場合のみに限り，現在実行中のタスクが Non-Secure Mode で実行中である可能性がある．

`exit_and_dispatch` タスク終了時にカーネルによって呼び出されるため，Non-Secure Mode から直接呼び出される可能性はない．

5.4 ユーザ割込みハンドラ

ユーザドメイン（Non-Secure 側）で割込みを受け取るためには，割込みコントローラの設定を変更して割込みを Non-Secure に割り振る必要がある．ASP3 では割込みラインの設定を行う `CFG_INT` というコンフィギュレーションが存在するため，これに新たなフラグ `TA_NONSECURE` を指定できるようにした．

例外ベクタテーブルは Secure/Non-Secure で独立したものが必要である．このため，カーネルコンフィギュレータに対して修正を行い，Non-Secure 用の例外ベクタテーブルが追加で生成されるようにした．各例外ベクタテーブルには対応するドメインの割込みハンドラのみが含まれる．

これらの 2 つの例外ベクタテーブルはまったく同一のコピーであっても，割込みが発生した際，振り割られてない側のベクタテーブルの内容は無視されるため，原理上は動作する．それにもかかわらず独立したベクタテーブルを用いるのは，こうすることでシステムドメインの関数ポインタ等，ユーザドメインからアクセス可能な情報を最小限に抑え，セキュリティリスクを軽減できるためである．

5.5 リンカスクリプト

SAU をコンフィギュレーションしメモリのセキュリティ属性を設定するためには，各ドメインに属する変数やコードが属するアドレス範囲の情報が必要である．このため，

リンカスクリプトにより各ドメインのシンボルがまとまった範囲に定義されるようにし，その前後にリンカシンボルが生成されるようにした．カーネルのターゲット依存部の初期化コードでは，そのシンボルのアドレスを使用して，SAU のコンフィギュレーションを行う．

6. 評価

CMSE をベースとした SBI 方式のメモリ保護 OS が従来のメモリ保護 OS よりも低いオーバヘッドでメモリ保護を実現できることを示すために，開発した ASP3+TZ に対して評価実験を行った．CMSE をベースとしたメモリ保護 OS の前例は知られていないため，本評価実験はこうした OS の性能に対して初めて評価値を示したものである．4 章で述べたように，本設計はメモリ保護なし OS をベースとして開発するためのものであるが，小規模な変更のみでメモリ保護を実現できることを示すために，コード変更量の評価を行った．

ASP3+TZ のサービスコールの実行時間を計測し，メモリ保護機構を持たない TOPPERS/ASP3 およびメモリ保護機構を持つ TOPPERS/HRP2 [1] と比較することにより，オーバヘッドの評価を行った．また，割込み応答時間についても同様に評価を行った．その後，ASP3+TZ を実装する際に修正を要した行数を計測することで，コード変更量の評価を行った．

TOPPERS/ASP3 に関してはドメインの概念を持たないため，システムドメインしか存在しないものと仮定して評価を行った．

6.1 評価環境

評価用のターゲットボードとして，ARM 社製の Cortex-M Prototyping System（MPS2）を使用した．MPS2 は FPGA を搭載しており，FPGA イメージを書き込むことで ARM-M の実装である各種 Cortex-M コアを使用した開発を行うことができる．本実験では FPGA イメージとして Cortex-M33 example IoT FPGA image for MPS2+を使用した．このイメージは ARMv8-M Mainline Profile の Cortex-M33 プロセッサと CoreLink SIE-200 System IP が含まれている．プロセッサは AHB5 および AHB5 Memory Protection Controller（MPC）を経由して基板上の計 8 MB のシングルサイクル SRAM と 16 ビットバスで接続されており，ここにコードとデータを配置できる．また，UART インタフェースを内蔵しており，これを介して PC と通信を行うことができる．

6.2 サービスコールの実行時間

評価実験では，ASP3+TZ におけるサービスコール `act_tsk` の実行時間を計測した．`act_tsk` は TOPPERS 第 3 世代カーネル統合仕様書 [9] で定義されている，タスクを起動するサービスコールである．`act_tsk` は実行中タスクと対象タスクの優先度の相対関係によって，ディスパッチが発生する場合と発生しない場合の 2 通りの動作がある．ディスパッチが発生しない場合は `act_tsk` を呼び出してからリターンするまでの時間を計測し，ディスパッチが発生する場合は `act_tsk` を呼び出してから対象タスクの最初のステートメントが実行されるまでの時間を計測し，呼び出し元と起動対象タスクのドメインが異なる場合についても計測を行った（図 8）．評価条件の一覧を表 2 に示す．

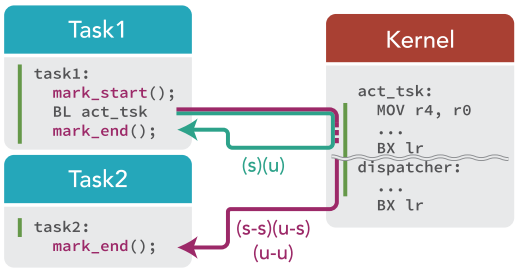
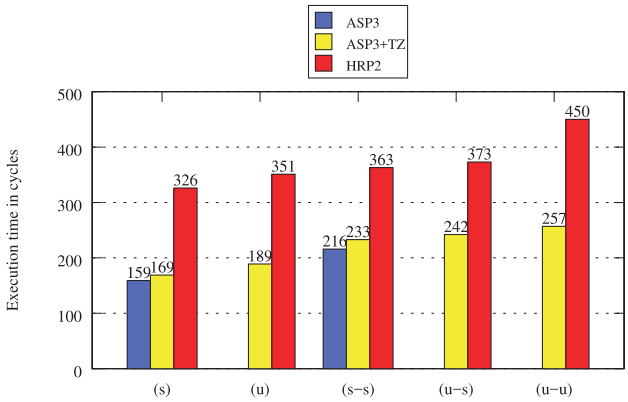


図 8 `act_tsk` の実行時間の計測方法
Fig. 8 The method for measuring the execution time of `act_tsk`.

表 2 `act_tsk` の実行時間の評価条件
Table 2 Conditions for measuring the execution time of `act_tsk`.

	呼び出し元タスク 所属ドメイン	起動対象タスク 所属ドメイン	ディスパッチ
(s)	システム		なし
(u)	ユーザ		なし
(s-s)	システム	システム	あり
(u-s)	ユーザ	システム	あり
(u-u)	ユーザ	ユーザ	あり



† ASP3 はユーザドメインを持たない為，(s)(s-s) 以外の計測値は示していない．

図 9 `act_tsk` の実行時間の比較 †
Fig. 9 The execution time comparison of `act_tsk`.

6.2.1 結果・考察

各計測対象の RTOS の `act_tsk` の実行時間を図 9 に示す．この結果は，ASP3+TZ はいくつか制約があるものの，HRP2 と比較して大幅に低いオーバヘッドでメモリ保護を

```
  200044:   f000 b82c   b.w 2000a0 <__ns_act_tsk_veneer>
002000a0 <__ns_act_tsk_veneer>:
  2000a0:   f85f f000   ldr.w   pc, [pc]
10000114 <ns_act_tsk>:
10000114:   e97f e97f   sg
10000118:   b500        push    {lr}
1000011a:   f002 fb89   bl  10002830 <act_tsk>
10002830 <act_tsk>:
```

__ns_act_tsk_veneer は long jump (b 命令で参照不可能な範囲へのジャンプ) を実現する為にリンカが自動生成した関数である．

図 10 ASP3+TZ のユーザドメインから act_tsk を呼び出した際に実行される命令列

Fig. 10 The instruction sequence executed during the call to act_tsk from the user domain in ASP3+TZ.

実現できることを示している．

このような差が生じた理由の 1 つとして，ユーザドメインからサービスコールを呼び出す際に実行される命令数に大きな差があることが考えられる．どちらのドメインから呼び出した場合でも最終的にはカーネル内にあるサービスコールの本体にたどり着くが，ユーザドメインから呼び出す場合はモードの遷移等に追加の処理が必要である．HRP2 ではこの処理に必要な命令数は 33 命令であるが，ASP3+TZ ではそれより大幅に少ない 5 命令で済んでいた（**図 10**）．

ディスパッチによるオーバヘッドが ASP3 と比べてほとんど増加していないのは，ディスパッチャに対するソフトウェア的な変更がごくわずかなためである．5 章で述べたように，追加した処理は Non-Secure 用スタックポインタの保存・復元のみである．適切なプロセッサモードへの切替えは以前のアーキテクチャから存在した EXC_RETURN の新たなビットフィールドに埋め込まれた値に従ってハードウェア側で自動で行われるため，モード切替えを行うにあたってソフトウェアがすべきことはほとんどない．

4 章でユーザドメインの細分化により故障を局在化できることについて述べたが，この結果は，大きなオーバヘッドを生じることなくより多くのコードをユーザドメインで実行することが可能であり，その結果，より厳密なアクセス権限の設定が可能になることを意味している．

6.3 割込み応答時間

さらに，同様の実験環境を用いて，各ドメインに所属する ISR の割込み応答時間の評価を行った．各計測対象の OS 上で，バックグラウンドでタスクを実行した状態でタイマ割込みを発生させ，割込みハンドラからタイマの値を計測することで割込み応答時間の値を得た．

各 OS について，システムドメインの割込み応答時間を計測するとともに，以下で説明する方法によりユーザドメイン割込みを作成し，同様の計測を行った．

TOPPERS/ASP3 ドメインの概念を持たないため，未計測である．

ASP3+TZ CFG_INT に TA_NONSECURE フラグを設定することで，割込みをユーザドメイン側に振り分け，DEF_INH に Non-Secure 領域内で定義した割込みハンドラを指定した．

TOPPERS/HRP2 この OS ではユーザドメイン上で割込みハンドラを直接定義することはできない．このため，ユーザドメイン所属のタスクを作成し，このタスクをシステムドメインの本物の割込みハンドラから起動することで，擬似的なユーザ割込みハンドラを実現した．

評価条件の一覧を**表 3** に示す．

表 3 割込み応答時間の評価条件

Table 3 Conditions for measuring the interrupt response time.

	バックグラウンドタスク 所属ドメイン	割込みハンドラ 所属ドメイン
(s-si)	システム	システム
(s-ui)	システム	ユーザ
(u-si)	ユーザ	システム
(u-ui)	ユーザ	ユーザ

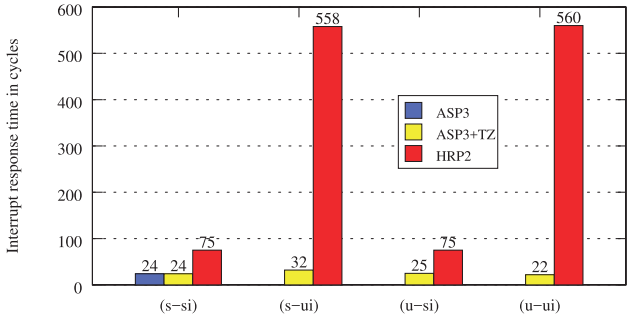


図 11 割込み応答時間の比較

Fig. 11 Comparison of interrupt response times.

6.3.1 結果・考察

各計測対象の RTOS の割込み応答時間を**図 11** に示す．この結果から，ASP3+TZ では TOPPERS/ASP3 とほとんど変わらない割込み応答時間で，システムドメインとユーザドメインの両方の割込みハンドラを実現できることが示されている．特に HRP2 の擬似的なユーザ割込みハンドラとの差は目覚ましく，ユーザドメイン所属の割込みハンドラが求められるアプリケーションにおける ASP3+TZ の有用性を示している．

割込みハンドラの所属ドメインにかかわらず割込み応答時間がほとんど変化しないのは，割込みが発生してから割込みハンドラが起動するまでの処理の大部分がハードウェアにより高速に行われている行われているためである．HRP2 と異なり，実行中のソフトウェア側でドメインを判別して切り替える処理も不要なため，システムドメイン所属の割込み（s-si）（u-si）も高速に処理することができる．ユーザドメイン所属の割込み（s-ui）（u-ui）に関

表 4 カーネルのコード変更行数
Table 4 Kernel code additions and deletions.

コード	ASP3 SLoC	追加	削除
アーキテクチャ依存部	1884	419	11
非依存部	7502	28	4

しても，実行モード遷移が自動的に行われるため，オーバヘッドが少ない．(s-ui) の割込み応答時間がわずかに長いのは，ハードウェア側で行われる割込みハンドラ起動処理の一部として，Secure Mode で使用中であったレジスタをスタックにすべて退避させ，ゼロクリアする処理が含まれるためである．

6.4 カーネルのコード変更量

ASP3+TZ の実装方針は，最小限の変更で，メモリ保護 OS を実現することであると述べた．そこで，この方針の達成度合いを検証するために，開発した ASP3+TZ と元となった ASP3 のソースコードを行単位で比較することで，コード変更量の評価を行った．

結果を**表 4** に示す．アーキテクチャ依存部と非依存部の両方について，変更量は 2 割程度にとどまっていることが確認できる．特に，非依存部の変更量は非常に少ない．この非依存部の変更はカーネルコンフィギュレータへの修正で，Non-Secure スタックを生成するようにするためのものである．以上の結果から，カーネルのコード変更量を最小限にするという目標は十分に達成できていると考えられる．

6.5 未実装の機能

ASP3+TZ では基本的な機能のみを実装しており，4 章で述べた機能の一部は未実装である．未実装の機能について，実装した場合の実行オーバヘッドや ASP3 に対する変更量について考察する．

カーネルオブジェクトのアクセス制御 一般的なメモリ保護 OS で使用されている方法が適用可能である．基本的には（ドメイン，オブジェクト，操作）の各組合せについてアクセスの許可・不許可を定義すればよい．文献 [1] で使用されている単純な方法として，各組合せに対するアクセス許可フラグを ROM で静的に持つことが考えられる．この場合，サービスコールの本体の先頭でアクセス許可フラグの値を読むだけでアクセス制御が行えるため，サービスコールの実行時間は増加するものの，影響は微小であると思われる．

複数ユーザドメインのサポート 4.1 節ですでに述べたように，複数ユーザドメインをサポートするためには，ドメインを切り替える際に SAU または MPU の設定変更を行う必要があり，これにはオーバヘッドがともなう．このため，ディスパッチをともなうサービスコールの実行時間が増加すると考えられる．

複数ユーザドメインのサポートで特に問題になるのが，ユーザ割込みハンドラの実装方法である．CMSE では割込み発生時に自動的に Non-Secure へ実行モード遷移を行うことが可能であるものの，遷移先ドメインに合わせて SAU や MPU の設定変更を行うことはできない．このため，いったん Secure Mode 側で割込みを受付けて，SAU や MPU の設定変更を行い，その後 Non-Secure 側の割込みハンドラを呼び出す必要がある．これは従来のリングプロテクション機構に基づいたメモリ保護 OS での実現方法と類似しているが，CMSE では Secure 側の割込みハンドラからリターンすることなく，Non-Secure 領域の関数を Handler Mode のまま直接呼び出せる．したがって，実行モード遷移の回数が少なくなるため，オーバヘッドは従来のメモリ保護 OS よりも小さくなるものと予想される．

非特権関数 非特権関数は，レジスタの機密情報をクリアした後に `BLXNS` 命令を使用して Non-Secure 領域内の関数を呼び出すラッパ関数として実装できる．この機能は他の機能と独立して実装できるため，これを実装することでサービスコールや割込みの性能に悪影響を与えることはない．

前述のように，カーネルオブジェクトのアクセス制御や，非特権関数を実装したとしても，実行オーバヘッドや ASP3 に対する変更量は大きくならない．一方，複数ユーザドメインをサポートすると，実行オーバヘッドが従来のメモリ保護 OS よりは小さくおさえられるが，現状の ASP3+TZ よりは大きくなると考えられる．しかしながら，システムによっては，複数ドメインが必要でないケースも存在する．たとえば，信頼性の高い自社で開発したコードと，信頼性が低い外部から調達したコード，安全系と非安全系の組合せでシステムを実現することは多いと考えられる．そのため，ドメインを 2 個のみサポートとして，実行オーバヘッドや変更量を抑えた OS を実現するのは有用であると考える．

7. おわりに

IoT 向けの組込みシステムは高い安全性が求められており，パーティショニングによりドメイン間のアクセスを制限し，故障の波及や情報の漏洩を防ぐことが重要である．しかし，従来型のメモリ保護 OS はメモリ保護の導入により大きなオーバヘッドを生じるという欠点があった．そこで本論文では，従来のメモリ保護 OS の欠点を克服する，TrustZone for ARMv8-M を用いた軽量なメモリ保護 OS を，非メモリ保護 OS をベースとして実現する方法について述べた．この方法をメモリ保護機構を持たない RTOS である TOPPERS/ASP3 に適用し，メモリ保護対応を行った ASP3+TZ を実現した．ASP3+TZ に対しサービスコー

ルのオーバヘッドと割込み応答時間の評価を行い，既存のメモリ保護 OS と比較して非常に少ないオーバヘッドでメモリ保護を実現できることを示した．さらに，ベースとした ASP3 からのコード修正量を比較することで，最小限の変更で ASP3+TZ を実現できることを示した．

現時点で残されている課題は，HRP2 にあるような時間保護やオブジェクトのアクセス権制御に対応することである．これらはパーティショニングに必要不可欠な要素であり，現在実装を検討している．

謝辞 本研究の一部は JSPS 科研費 JP17K00075 の助成を受けたものです．

参考文献

[1] 石川拓也，本田晋也，高田広章：静的なメモリ配置を行うメモリ保護機能を持ったリアルタイム OS，コンピュータソフトウェア，Vol.29, No.4, pp.161–181 (2012).
[2] Danner, D., Müller, R., Schröder-Preikschat, W., Hofer, W. and Lohmann, D.: SAFER SLOTH: Efficient, hardware-tailored memory protection, *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium* (*RTAS*), pp.37–48 (online), DOI: 10.1109/RTAS.2014.6925989 (2014).
[3] 本田晋也，岡部 亮，攝津 敦：車載システム向けの仮想マシンの割込み応答性向上手法，技術報告 14，名古屋大学大学院情報科学研究科，三菱電機株式会社情報技術総合研究所，三菱電機株式会社情報技術総合研究所 (2017).
[4] ARM Ltd.: ARMv8-M Architecture, ARM Ltd. (online), available from ⟨https://www.arm.com/products/processors/instruction-set-architectures/armv8-m-architecture.php⟩ (accessed 2017-01-27).
[5] 中嶋健一郎，本田晋也，手嶋茂晴，高田広章：セキュリティ支援ハードウェアによるハイブリッド OS システムの高信頼化，情報処理学会研究報告，EMB，組込みシステム，Vol.2008, No.116, pp.1–7（オンライン），入手先 ⟨http://ci.nii.ac.jp/naid/110007099117/⟩ (2008).
[6] Guan, L., Liu, P., Xing, X., Ge, X., Zhang, S., Yu, M. and Jaeger, T.: TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone, *CoRR*, Vol.abs/1704.05600 (online), available from ⟨http://arxiv.org/abs/1704.05600⟩ (2017).
[7] ARM Ltd.: Application Note 291: Word final word Using TrustZone on ARMv8-M, ARM Ltd. (online), available from ⟨http://www.keil.com/appnotes/docs/apnt_291.asp⟩ (accessed 2017-02-06).
[8] AUTOSAR: Specification of Operating System, Automotive Open System Architecture GbR (online), available from ⟨http://www.autosar.org/standards/classic-platform/release-43/software-architecture/system-services/⟩ (accessed 2017-02-07).
[9] NPO 法人 TOPPERS プロジェクト：TOPPERS 第 3 世代カーネル（ITRON 系）統合仕様書 Release 3.0.0，NPO 法人 TOPPERS プロジェクト（オンライン），入手先 ⟨https://www.toppers.jp/docs/tech/tgki_spec-300.pdf⟩（参照 2016-02-15）．

河田 智明

名古屋大学大学院情報科学研究科博士前期課程在学中．組込み向けセキュリティに関する研究に従事．

本田 晋也 （正会員）

2002 年豊橋技術科学大学大学院情報工学専攻修士課程修了．2005 年同大学院電子・情報工学専攻博士課程修了．名古屋大学大学院情報科学研究科付属組込みシステム研究センター助教等を経て，2014 年より名古屋大学大学院情報科学研究科情報システム学専攻准教授．リアルタイム OS，ソフトウェア・ハードウェアコデザインの研究に従事．博士（工学）．2002 年度情報処理学会論文賞受賞．ACM，IEEE，電子情報通信学会，日本ソフトウェア科学会各会員．本会シニア会員．