Linux のしくみを学ぶ - プロセス管理とスケジューリング

プロセスとマルチタスクの実現

Linux は、多数のプロセスを同時に動作させる事が出来るマルチタスク環境を実現しています。
ps コマンドや top コマンドを実行しプロセスのリストを取得すると多数のプロセスが実行中である事がわかります。また、これらのプロセス全てが同時に平行して動作しているように見えます。これはどのようにして実現されているのでしょうか。

一般的に、１つのプロセッサは同時に複数のプログラムを実行する事が出来ません[*1]。
そこで、マルチタスクをサポートする OS では複数のプロセスを非常に短い時間ずつ切り替えながら実行する事で、体感上は複数のプロセスが同時に実行されているように見せています (図 1)。

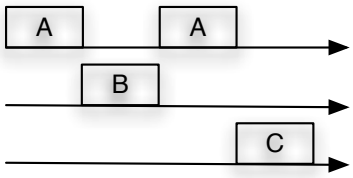

図 1　マルチタスク

このような仕組みを実現する為にどんな機能がカーネルへ実装されているのか順に見ていきましょう。

プロセスの切り替え

CPU 上で処理中のデータを失う事なく現在のプロセスから別のプロセスに切り替えを行うには、現在のプロセスが使用しているレジスタやフラグなど CPU の状態を保存・復帰出来るようにする必要があります。

この CPU の状態の事をコンテキスト、保存と復帰を行いコンテキストを切り替える事をコンテキストスイッチと呼びます。

カーネルでプロセスの切り替えをサポートするには、プロセス毎にコンテキストの保存領域を用意・管理し適切なタイミングでコンテキストスイッチを実行する必要があります。

[*1] Intel の HyperThreading や Sun の CMT のように１プロセッサで複数スレッドを処理可能なプロセッサも存在します。しかしこの場合 OS はスレッド数分の論理プロセッサが存在すると認識する為、この論理プロセッサの上ではやはり同時に複数のプログラムを動かす事が出来ません。

プロセス毎の CPU 時間の管理とプロセス切り替え

プロセス切り替えは、プロセスがスケジューラに割り当てられた CPU 時間を使い切った時、あるいはプロセスが IO 待ちなどの理由で自発的に CPU を手放した時に実行されます。

CPU 時間の管理の仕組み

Linux カーネルはハードウェアタイマにより定期的にタイマ割り込みが発生するように設定されています。

この割り込みによって呼ばれる割り込みハンドラでは一定の時間が経過する度にプロセスの CPU 利用時間を更新します。

プロセス切り替えの仕組み

プロセスの CPU 利用時間がスケジューラにより割り当てられた CPU 時間を超えたら、割り込みからの復帰時に現在のプロセスの状態を保存し、別のプロセスへ切り替えを行います。

以下に例を示します (図 2)。

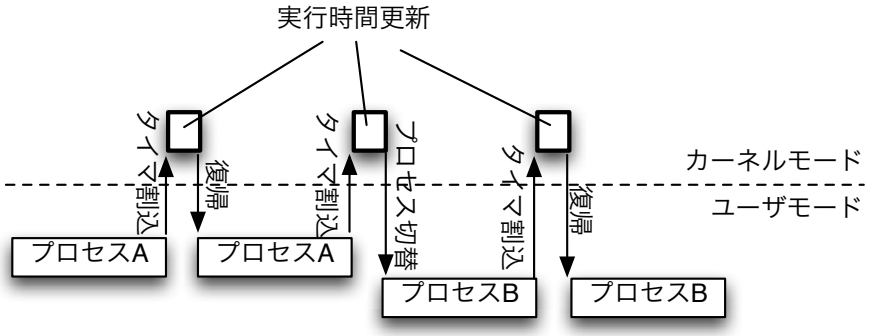

図 2　タイマ割り込みとプロセス切り替え

1. ユーザモードでプロセス A が実行されています。
2. タイマ割り込みが発生し、カーネルモードへ制御が移ります。
3. タイマ割り込みの割り込みハンドラからプロセス利用時間を更新する処理が呼び出されます。
4. 割り当てられた CPU 時間を使い切っていない為プロセス A へ復帰、ユーザモードでプロセス A の実行が再開されます。
5. タイマ割り込みが発生し、カーネルモードへ制御が移ります。
6. タイマ割り込みの割り込みハンドラからプロセス利用時間を更新する処理が呼び出されます。
7. 割り当てられた CPU 時間を超えたので、プロセス B へ切り替えを行います。
8. プロセス B へ復帰、ユーザモードでプロセス B の実行が再開されます。
9. タイマ割り込みが発生し、カーネルモードへ制御が移ります。
10. タイマ割り込みの割り込みハンドラからプロセス利用時間を更新する処理が呼び出されます。

11. 割り当てられた CPU 時間を使い切っていない為プロセス B へ復帰、ユーザモードでプロセス B の実行が再開されます。

コンテキストスイッチ

次に、Linux のコンテキストスイッチ処理がどのように実装されているのかを見ていきます。

Linux カーネルでは、コンテキストの保存領域として thread_info 構造体とカーネルスタックを併用します。この二つはプロセス毎に割り当てられており、プロセス生成時に作成されます。

コンテキストスイッチは以下のような手順で実行されます (図 3)。

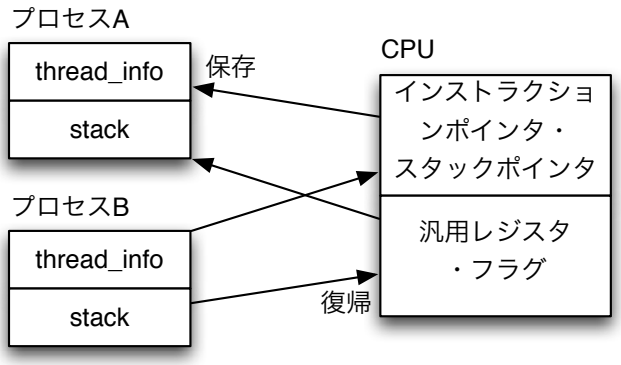

図 3　コンテキストスイッチ

1. スケジューラで次に実行するプロセスを選択します。
2. ページテーブル (詳しくはページングを参照) を切り替えます。
3. スタックへ汎用レジスタ、フラグを退避します。
4. 現在のプロセスの thread_info 構造体へスタックポインタ、インストラクションポインタを退避します。
5. 次に実行するプロセスの thread_info 構造体からスタックポインタを復帰します。
6. セグメントレジスタを切り替えます。
7. 次に実行するプロセスのインストラクションポインタを復帰します。この時点でコンテキストが次のプロセスへ切り替わります。
8. スタックから汎用レジスタ、フラグを復帰し、コンテキストスイッチを完了します。

プロセススケジューリング

ユーザに対し全ての実行中プロセスがスムーズに動作しているように見せ、かつ高い性能が得られるようにするには、

- いつコンテキストスイッチを実行するか
- どのプロセスに切り替えるか

が重要になります。

プロセスへの時間割り当てが適切に行われていないと、そのプロセスはユーザから見て止まっているように見えてしまいます。
また、コンテキストスイッチの頻度が高すぎるとオーバーヘッドが増大し、低すぎるとプロセスが断続的に止まっているように見えてしまいます。
この二点を適切に決定し、実行する事がプロセススケジューラの役割になります。

Linux カーネルでは 2.6.23 よりそれまでの O(1) スケジューラに代わり CFS(Completely Fair Scheduler:完全に公平なスケジューラ) がマージされました。
このスケジューラは、”CPU 時間の分配を可能な限り公平にする” というコンセプトの元に実装されています。

いつプロセス切り替えを実行するのか

プロセスが次のプロセスへ切り替えられるまでに与えられる CPU 時間の事をタイムスライスと呼びます。

CFS ではタイムスライスの長さをシステム負荷に応じて動的に調整します。まず、全タスクを一周動作させるまでのスケジューリング周期を求めます。この値は単一プロセッサ時の周期 (20msec) に CPU 数によって重み付けをした値です。

この値に対し、以下のような計算を行った値をプロセスのタイムスライスとしています。

（スケジューリング周期 ÷ プロセス数）× 優先度による重み付け

つまり、CPU 数が増える程スケジューリング周期は長く、プロセスが増える程タイムスライスは短くなります。これは、CPU 数が増えれば一度に実行出来るプロセス数が増えるのでスケジューリング頻度を増やす必要が無くなる為です。

但し、このルールのみを用いていると、プロセスが多くなるに従いタイムスライスが短くなりすぎる為、タイムスライスの最小値を決めそれより短くならないようにしています。

どのプロセスへ切り換えるのか

マルチタスクをサポートする OS では一般的に、実行可能なプロセスを管理するランキューをカーネル内に持っており、ここから一つプロセスを選んでプロセス切り替えを行います。この時に、できるだけ低いコストで、最も適切なプロセスをランキューから取り出し、ランキューをメンテナンスするのがプロセススケジューラの重要な仕事になります。

Linux 2.4 のスケジューラではランキューは単純なリスト構造になっていました。しかし、この構造の場合、プロセススイッチの為に優先度の最も高いプロセスをランキューから一つ選ぶにはランキューを線形探索しな

ければなりません。これでは、探索コストが O(n) になってしまいプロセスが増えると非常に時間がかかってしまいます。

探索コストの問題を解決する為、以前の O(1) スケジューラでは優先度別リストを導入していました。これは、優先度ごとのプロセスリストの配列になっており、配列のエントリ毎に実行可能なプロセスの有無を示すビットを持っているのでスケジューラは常に一定の計算量 (O(1)) で最優先プロセスを探索する事が可能でした。

しかし、単純に優先度の高いプロセスから実行していくとプロセスの実行頻度が大きく偏るという構造的な問題があり、これを避ける為にプロセスの優先度を調整したりしなければならず実装が複雑になっていました。

CFS ではこれらの問題を解決する為、ランキューにプロセスの実行時間でソートされた Red-Black Tree(赤黒木: 二分木の一種) を用いています (図 4)。

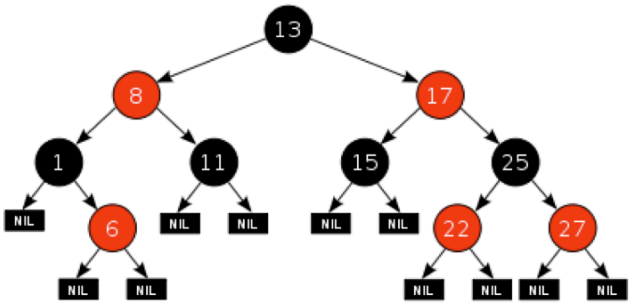

図 4　Red-Black Tree

全てのプロセスはスケジューリングの為に vruntime というパラメータを持ちます。これはプロセスが今までに使用した CPU 時間の合計に対して優先度による重み付けを行った値です。ランキューは vruntime の値でソートされ、スケジューラは vruntime の値が最も小さいプロセスを実行しようとします。１つのプロセスを実行し続けると vruntime が増加していき、他のプロセスの vruntime の方が最も小さい値になった時点でプロセス切り替えが行われます。これを繰り返す事で全てのプロセスが公平に実行され、かつ優先度が反映された時間割り当てが実現されます。

探索コストの面では、Reb-Black Tree の操作にかかるコストは O(log n) なので O(1) に比べて劣るように見えますが、プロセス数が多くなれば差は非常にわずかになります。また、スケジューラのアルゴリズム上、殆どの場合左端のノードしか参照しない事が分かっているので、このノードのポインタをキャッシュする事で更に高速化しています。

マルチプロセッサ環境への対応

Linux カーネルは多数の CPU を搭載する NUMA を含むマルチプロセッサ環境にも最適化されており、プロセススケジューリングも例外ではありません。

最初に、ランキューを見てみましょう。旧来の Linux2.4 カーネルではシステム全体で１つのランキューを持っていました。この仕組みはシンプルですが、ランキューに対して操作を行う度にロックをかけなければならず、大規模システムには不向きでした。現在の Linux カーネルでは、CPU 毎のランキューを用意する事で

これを解決しています。しかし、ランキューを分けてしまうと CPU 毎の実行プロセス数が偏ってしまい、あるプロセッサは忙しいが別のプロセッサはアイドル状態というような状況が発生する可能性があります。

そこで、プロセススケジューラでは定期的にランキューの実行プロセス数の偏りをチェックし、一定以上の偏りが有ればプロセスが多いキューから少ないキューへプロセスを移動しています。

また、新しいプロセスを生成する時に最も実行中プロセスが少ないランキューを持つ CPU へプロセスを割り当て、プロセス数の偏りが起きないようにしています。

プロセスに関するデータ構造

プロセスディスクリプタ

プロセスがどのような状態にあるか、どんな資源へアクセスしているかなどを管理する為、全てのプロセスに一つずつプロセスディスクリプタが割り当てられます。Linux でのプロセスディスクリプタは task_struct 構造体という名前になっていて、ここにプロセスに関する全ての情報が詰め込まれています。

項目数が大変多いので、主な項目をピックアップして以下に示します。

プロセスの状態

プロセスの状態にて説明します。

thread_info 構造体

コンテキストスイッチ時にスタックポインタ、インストラクションポインタなどを保存する領域です。

優先度

プロセスをどの程度優先して実行するかを表すパラメータで nice 値を元に決定され、プロセススケジューリングに反映されます。

スケジューリング情報

vruntime などのプロセススケジューリングに必要な情報を記録しています。

PID

プロセスを識別するのに利用されるユニークな ID です。

プロセスの親子関係

Linux のプロセスには親子関係があります。この項目ではプロセス間の親子関係を記録しています。

メモリディスクリプタ

プロセスの仮想メモリ空間に関する情報 (ページテーブルやメモリリージョンの情報など) を記録しています。詳しくはメモリディスクリプタで説明します。

開いているファイルの情報

プロセスがオープンしているファイルディスクリプタを記録しています。

パイプに関する情報

プロセスが利用しているパイプに関する情報を記録しています。

シグナルに関する情報

プロセスが受け取ったシグナルに関する情報を記録しています。

スケジューラが利用するランキューや [ウェイトキュー] で説明するウエイトキューは、このプロセスディスクリプタへのポインタを管理しています。

プロセスの状態

システムで実行中のプロセスは、IO 待ちなどにより一時的に実行を中断しなければならない状態が存在します。このようなプロセスの状態遷移を表す為、プロセスディスクリプタにステータスフラグが用意されています。

TASK_RUNNING

実行中・実行待ちの状態です。停止される要因が無ければプロセスは常にこの状態にあります。この状態のプロセスはランキューによって管理され、実行されるのを待ちます。

TASK_INTERRUPTIBLE

ある要因によって待ち状態に入っている事を表します。ハードウェア割り込み、システム資源の解放、シグナルなどの要因により起床します。この状態のプロセスはウエイトキューによって管理され、待ち要因の事象が発生するのを待ちます。

TASK_UNINTERRUPTIBLE

TASK_INTERRUPTIBLE と同様の状態ですが、シグナルによって起床しません。この状態のプロセスはウエイトキューによって管理され、待ち要因の事象が発生するのを待ちます。

TASK_STOPPED

シグナルにより停止状態に設定されています。ランキュー／ウエイトキューのどちらにも管理されていません。

TASK_TRACED

デバッガにより停止状態に設定されています。ランキュー／ウエイトキューのどちらにも管理されていません。

ウエイトキュー

TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE のプロセスは待ち要因別の双方向リストによって管理されています。プロセスが待ち状態に入る時、対応する待ち要因のウエイトキューに入れられます。

待ち要因の事象が発生したら、ウエイトキューからプロセスをランキューへ移動して起動させます。

プロセスの生成

Linux では新しいプロセスを生成しプログラムを実行するのに fork() と execve() の二つのシステムコールを順に呼ぶ必要があります。fork() では現在のプロセスを複製して子プロセスを作り、execve() では実行ファイルを読み込み、現在のプロセスを上書きします。

fork() の流れ

fork() の中で大まかにどのような処理が行われているかを以下に示します (図 5)。

1. 子プロセスに新しい PID を割り当てます。
2. プロセスディスクリプタを割り当て、親プロセスのプロセスディスクリプタから内容をコピーします。
3. thread_info 構造体を割り当て、親プロセスの thread_info 構造体から内容をコピーします。
4. 利用中ファイルの情報、シグナルの情報、メモリディスクリプタなどをコピーします。
5. 親プロセスのスタックに退避されているレジスタの値を子プロセスのスタックにコピーします。
6. ステータスを TASK_RUNNING に設定します。
7. ランキューに登録します。

プロセスディスクリプタ、thread_info 構造体、利用中の資源の情報などは全てコピーされます。但し、メモリ空間はコピーせずに親プロセスと共有し、後で書き込みが起きた時点で初めてコピーを行います。このような方式をコピーオンライトと呼びます。詳しくはコピーオンライトで解説します。

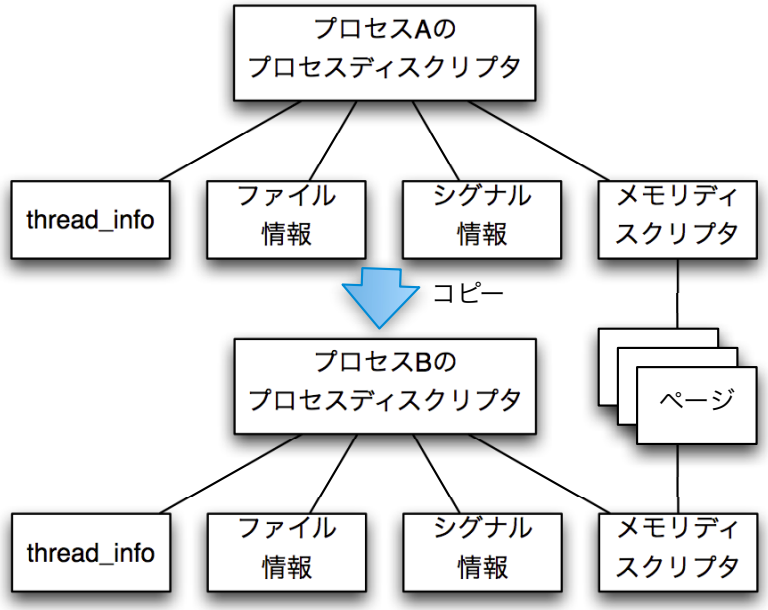

図 5　fork()

execve() の流れ

execve() の中で大まかにどのような処理が行われているかを以下に示します (図 6)。

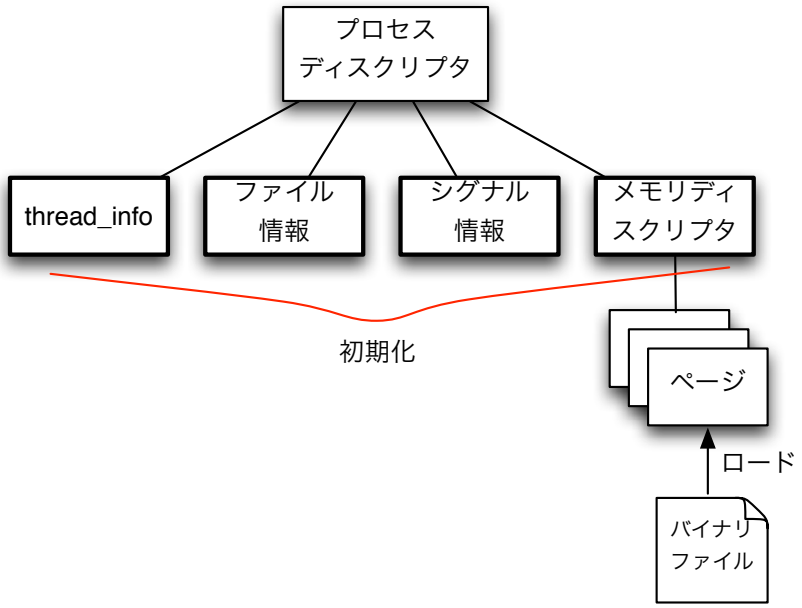

図 6　exec()

1. バイナリファイルをオープンします。
2. バイナリファイルのヘッダをロードします。
3. ファイル名、引数、環境変数をコピーします。
4. バイナリヘッダを読み込みます。
5. 古い資源を解放します。
6. メモリ領域を作成してプログラムをマップします。
7. スタックポインタ、インストラクションポインタの初期値を設定します。

execve() はバイナリファイルをロードし、現在のプロセスのコンテキストを上書きします。バイナリファイルのヘッダからバイナリの形式を判断、適切なローダを用いてメモリレイアウトなどを取得し、新しいメモリ領域を作ってプログラムをマップします。

プロセスのアドレス空間

Linux カーネルではメモリ保護の為プロセス毎に異なる仮想アドレス空間を割り当て、他のプロセスのメモリ領域を破壊する事を防いでいます。

この仮想メモリへ必要な物理メモリを割り当てていく事で、実際にプロセスへ物理メモリを提供しています。

ページング

Linux では他の多くの OS と同様にメモリ管理にページング方式を用います。この方式では、物理メモリ空間をページと呼ばれる固定長のブロックに区切り、仮想アドレスに対するページ割り当てを記録するページテーブルで管理します。

プロセス毎に異なる仮想メモリ空間を割り当てる為、ページテーブルはプロセスに一つ割り当てられます。

x86 アーキテクチャのページテーブルは 2 階層ですが、Linux カーネルでは各種 64bit アーキテクチャに対応する為に最大 4 階層までのページテーブルをサポートしています。

メモリディスクリプタ

メモリディスクリプタはプロセスのアドレス空間に関する情報を全て記録している場所で、プロセスディスクリプタに対して 1 つ割り当てられています。

主な項目をピックアップして以下に示します。

メモリリージョンのリスト

プロセスのアドレス空間に割り当てたメモリ領域の情報です。詳しくはメモリリージョンにて説明します。

ページテーブル

プロセスのアドレス空間を管理しているページテーブルのアドレスです。

コード領域の情報

プロセスのプログラムデータを割り当てている領域です。

ヒープ領域の情報

プロセスが動的に確保するデータ記録用の領域です。C 言語の malloc() 等で使用されます。

データ領域の情報

プロセスが静的に確保するデータ記録用の領域です。C 言語のグローバル等で使用されます。

メモリリージョン

カーネルは execve() 時の初期メモリ領域の割り当て時やプロセスからのシステムコールを通した要求により、仮想アドレス空間にプロセスがアクセス可能なリニアアドレスの区間を作成します。この区間をメモリリージョンと呼んでいます。

メモリリージョンはメモリリージョンディスクリプタ (vm_area_struct 構造体) によりプロセス毎に管理されており、主に以下のような情報を持っています。

- メモリ領域の範囲 (先頭、終端アドレス)
- メモリへのアクセス権
- ファイルオブジェクト（ファイルをメモリマップしている場合のみ）

メモリリージョンにより実行中プロセスの仮想アドレス空間へメモリ領域を割り当てた様子を以下に示します (図 7)。

デマンドページング

Linux カーネルでは、メモリリージョンの作成を行った時点では対応するページ（固定長の物理メモリのブロック）を確保しません。プロセスがこの領域にアクセスを行うと、メモリが割り当てられていないページである為ページフォルトが発生します。この時点で初めてページフレームを割り当てます。この手法をデマントページングと呼びます。

殆どの場合、プロセスがある時点にアクセスするメモリ領域はごく一部のページに限られています。その為、この手法によってメモリ割り当てを効率化する事が出来ます。

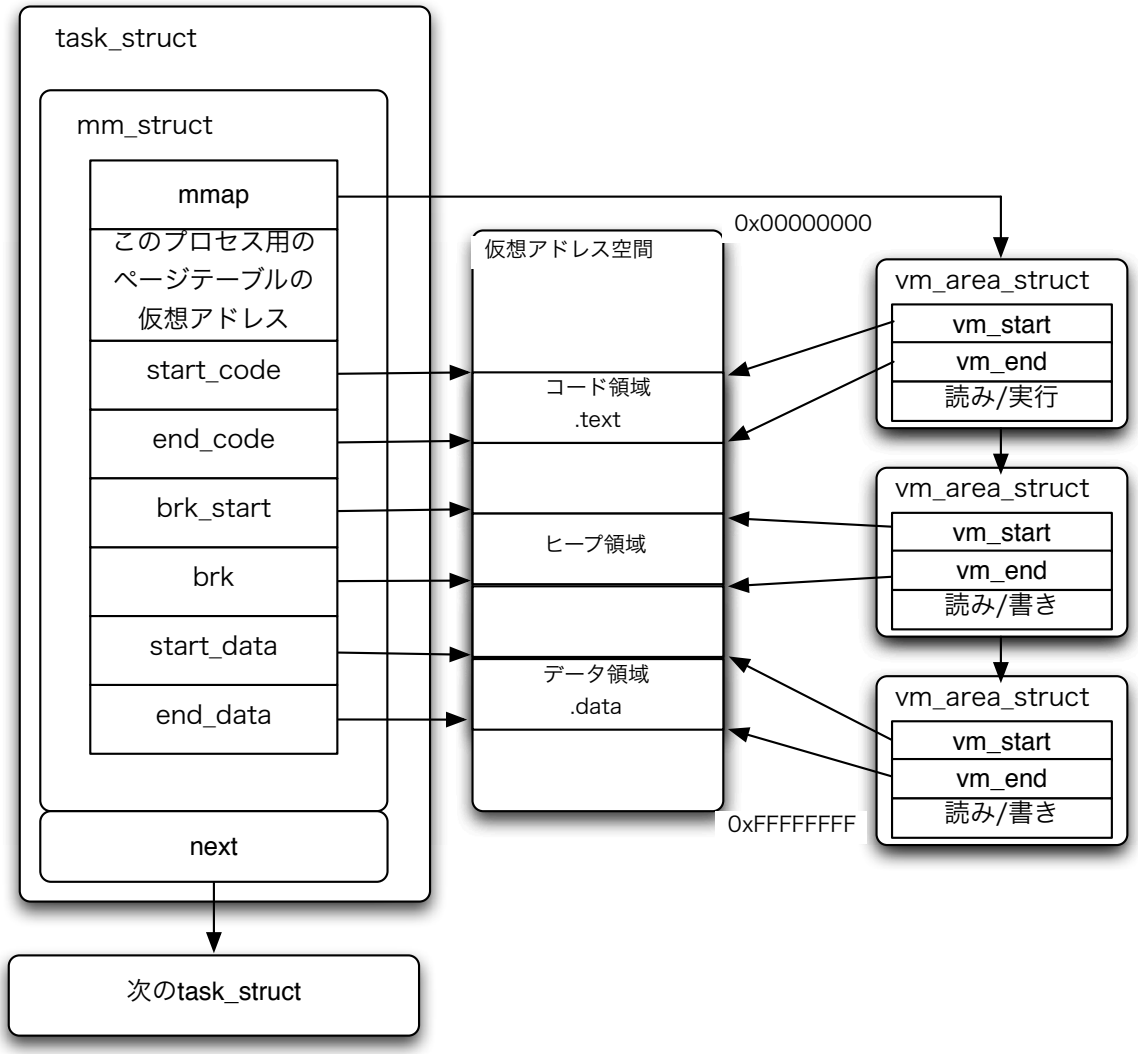

図 7　メモリディスクリプタとメモリマップ

コピーオンライト

fork() でプロセスを複製する際、Linux カーネルでは親プロセスのページフレームをコピーせずにページを読み取り専用に設定して親プロセスと子プロセスの間で共有します。その後、ページに対する書き込みが発生した時点で初めて新しくページを作成しコピーを実行します。このような方式をコピーオンライトと呼び、無駄なコピーを抑制させる事によりカーネルの性能を向上させています。

プログラムをロードして新しいプロセスを作成する時、多くの場合は fork() の直後に execve() を実行する為、親プロセスのメモリ領域は直ぐに必要なくなってしまいます。このような場合、コピーオンライトでページのコピーを抑制しておけば無駄なコピーを行わなくてすみます。

まとめ

以上、プロセス管理の仕組みとプロセススケジューラ、プロセスのアドレス空間について見てきました。複数のプロセスがどのようにして平行動作しているのかお分かりいただけたでしょうか。

ライセンス

参考文献

以下に本記事を執筆するにあたり参照した文献を示します。

- 詳解 LINUX カーネル 第 3 版 Linux カーネル 2.6 解読室 Linux
- Kernel Watch http://www.atmarkit.co.jp/flinux/index/indexfiles/watchindex.html
- developerWorks Japan http://www.ibm.com/developerworks/jp/