STEP 05

リアルタイム OS について

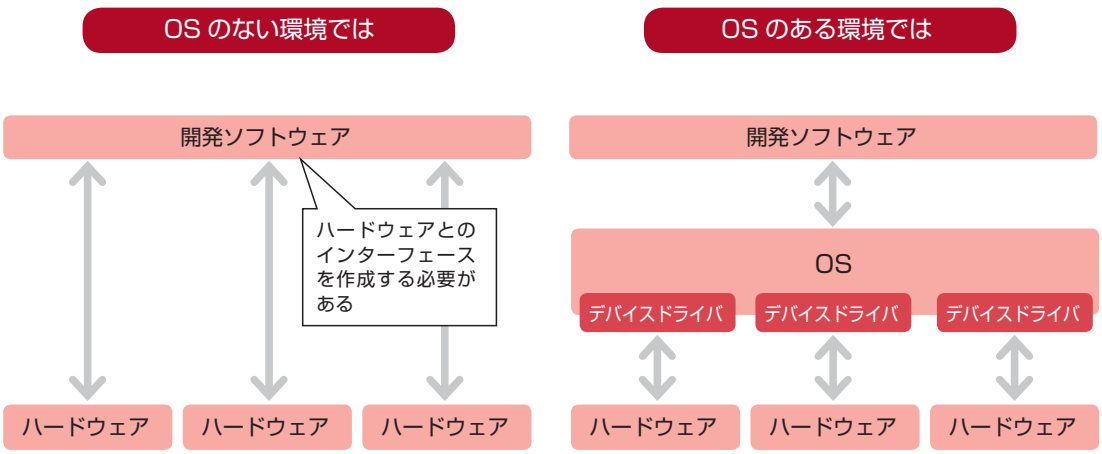
学習内容

この章では、まず OS の概要について学び、
組み込み現場で用いられるリアルタイム OS について学習します。

1. 組み込みシステムにおける OS の必要性

現在でも OS を使わずそのままプログラムを ROM に書き込んで実行する組み込みシステムが多くあります。ローエンド環境ではその方法でも目的を達成できます。
しかし、複数のプログラムを同時に走らせたり、ユーザープログラムがメモリを動的に使ったり、液晶ディスプレイに画像を表示したり、ネットワークを介して通信をしたりと高度なことが要求されると実現するのは困難になります。それらすべてをハードウェアに合わせて自分でプログラムコードを書かなければならないからです。

しかし OS を使えばそれらの要求に比較的容易に応えることができます。OS がメモリの管理や外部ハードウェアとの面倒なやりとりを肩代わりしてくれるのです。



リアルタイム OS について

2. OS を使うメリット

OS を使うメリットを、ミクロからマクロの視点であげると以下になります。

ミクロ
- ■ タスク分割が可能
  仕様書通りの読みやすいソースにすることができる
- ■ モジュール化が可能
  分業を行いやすい

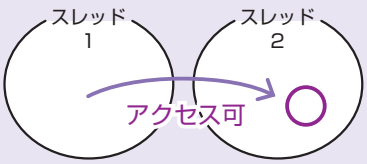
マクロ
- ■ ドライバとアプリケーションの分離が可能
  プログラムの再利用性が高まる

タスク

プログラムの処理単位。OS により スレッド や プロセス と呼ばれる。
OS 上で同時実行可能なタスクは 1 つだけ。
タスクを順番に切り替えて実行することで、同時に複数の処理をしているようにみえる。

スレッド

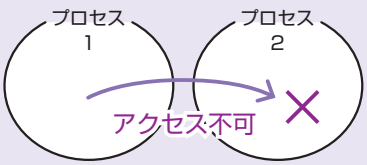
別の実行単位（スレッド）のメモリにアクセス可能。
サイズが小さく、応答性が高い。
例：uC/OS、ITRON、VxWorks、Nucleus など



プロセス

別の実行単位（プロセス）のメモリにアクセスできない。
多くの場合、プロセス内に複数のスレッドを持つことができる。
その場合には同一プロセス内のスレッド間ではメモリの共有が可能。

サイズが大きく、応答性に限界がある。
例：Linux、Windows 系、OS-9、QNX など



3.　OS を使うデメリット

OS を使うのはメリットだけではありません。
まず、プログラム（コンパイル済みの実行可能ファイル）のサイズや処理量は OS が存在している分だけ増加します。また、以下のデメリットも認識しておく必要があります。

■ 速度的なオーバーヘッド

タスク切り替えに時間がかかる（100 μ秒 [16bit CISC] ～ 数μ秒 [32bit RISC]）

■ スタック領域の増加

タスクごとにスタックが必要になる
検出困難な スタックオーバーフロー の危険性がある

■ 排他制御が必要

リエントラント ではない関数や共有データなどを、複数のタスクから同時利用できないように管理しなければならない

スタックオーバーフロー・・・
関数呼び出しが多すぎるとスタックは「オーバーフロー」し、プログラム側で対策をとっていなければ通常はクラッシュしてしまう。

リエントラント・・・
（Reentrant、再入可能）とは、プログラムやサブルーチンが、静的な内部状態をもたないので、再帰的にも、複数のスレッドからも、データを壊すおそれなく呼び出せる場合をいう。

リアルタイム OS について

4.　リアルタイム OS（RTOS）とは

組み込み系で用いられることが多いリアルタイム OS とは、どのようなものでしょうか。

■ リアルタイムシステム向けに設計された OS

あるタスクの実行にかかる時間の上限が（より優先度の高いタスクが無い場合に）容易に算定できる。
最も優先度の高いタスクであれば、OS の処理がより優先度の低いタスクで立て込んでいる場合であっても、最優先のタスクの処理が遅くなる（最優先のタスクの実行開始・完了までの時間が伸びてしまう）ということは無い。

■ タスクのスケジューリングが主な役割

最も優先度の高い実行可能なタスクが実行される。
汎用の OS のように複数のタスクを優先度を指定せずに平行して実行することは想定していない。

5.　RTOS のスケジューリング方法

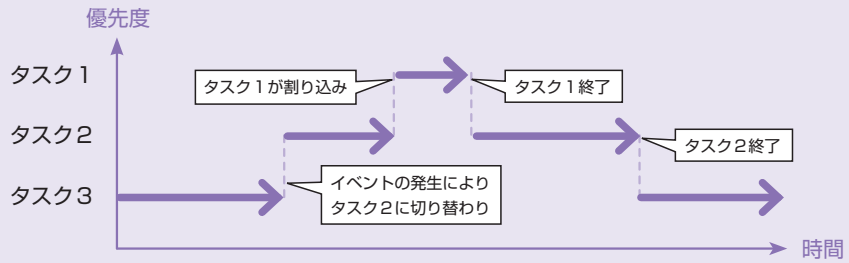
OS のスケジューリング方式の代表的な方法として、以下の 2 種類があります。

■ イベントによって要求された時にタスク切り替えを行う イベント駆動型 設計
■ 一定時間ごとにタスク切り替えを行う タイムシェアリング型 設計

RTOS は一般にイベント駆動型設計が採用されており、リアルタイム性が要求されるアプリケーションに向いています。

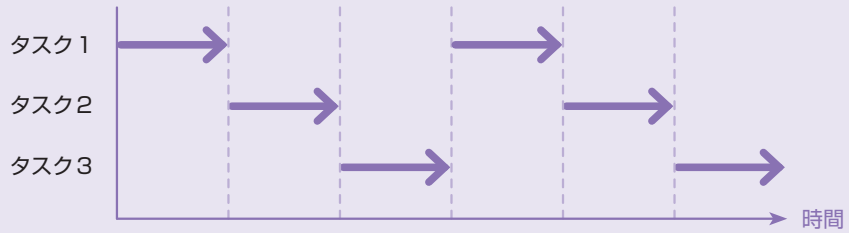
イベント駆動型（Event Driven System）

・イベントの発生によりタスクが切り替わる
・各タスクに優先度がある
　・イベントによって起動された優先度の高いタスクは、実行中のタスクに割り込んで処理を行うことができる。（プリエンプティブ　preemption, 横取り ）
　・優先度の高いタスクが MPU の利用権を放棄するまで、優先度の低いタスクは実行されない。



タイムシェアリング型（Time Sharing System）

・時分割システム
・一定時間が経過するとタスクが切り替わる
・各タスクに平等に（あるいは差をつけて）MPU の実行権を与える



リアルタイム OS について

リアルタイム OS と汎用 OS

ある程度以上の大きさのシステムでは、一般に複数のサービスを平行して行う必要が出てきます。ただし、ひとつの MPU が同時に処理できるサービスは 1 つだけであり、MPU の時間的なリソース（時間資源）を個々のサービスに何らかの方法で割り振らなければなりません。また、時間以外のリソース（ペリフェラル等）の各サービスへの割り振りも必要になります。OS（オペレーティングシステム）の役割は、複数のサービスへのリソースの割り当て・管理（スケジューリング）を行うことです。

リアルタイム OS は、優先度の高いタスク※1 を実行したい時にそのタスクがただちに実行され、一度実行されたタスクは、より優先度の高いタスクが実行されない限り最優先で実行され続けるように設計されています。つまり、実行したいタスクが立て込んでいる場合であっても、最優先のタスクは他にタスクが無い場合と同じ時間で開始・完了することが予想されます。一方で、優先度の低いタスクはより優先度の高いサービスが実行中である限りいつまでも実行されません。また、各タスクの優先度はあらかじめ決めておかなければならず、複数のタスクを同じ優先度にしたいというような状況にも一般に対応していません※2。

一方、汎用の OS の場合は、複数のサービスを同じ優先度で実行できるように、すべてのサービスに平等に時間を配分することを基本としています。そのため、特定のサービスがいつまでも実行されないような状況は避けることができ、また個々のサービスに優先度をつける必要もありません。一方で、実行したいサービスが立て込んでいる場合、あるサービスの実行がいつ始まっていつ完了するかは予想がつかなくなります。

リアルタイム OS は、あらかじめ決められた優先度に応じてタスクのスケジューリングを行うことに特化したコンパクトな OS であり、組み込みシステムでの使用に適しています。また、優先度の高いタスクは最優先で実行されるので、リアルタイム性が要求されるシステムでも使うことができます※3。

汎用の OS は、元々は大型のコンピュータを複数のユーザーが同時に使ったり、任意のプログラムを任意のタイミングで実行したりするような状況を想定したものであり、それらを適切に管理するために OS の仕組みはより複雑になります。ただし、近年では組み込みシステムにおいても、シビアなリアルタイム性よりも複雑な処理をこなすことが要求される場合などには、組み込みシステム向けにカスタマイズした汎用 OS が使われるようになってきています。

※1　リアルタイム OS では、一般にサービスを「タスク」と呼びます
※2　同じ優先度のタスクが存在している場合、OS にもよりますが、例えば「先に実行されたタスクが優先」といったような基準で一方が優先的に処理されることになり、汎用 OS におけるような「同じ優先度」は実現されません。また、OS によっては複数のタスクを同じ優先度に設定できない場合もあり得ます。
※3　例えば、OS を使ったシステムに自動車のブレーキ制御を組み込むことを考えてみます。汎用の OS では、ブレーキが必要という命令が入ってから実際にブレーキがかかるまでに要する時間は一般に不明であり、普段は問題なくブレーキが効いていたとしても、たまたま OS の処理が立て込んでいた場合などになかなかブレーキがかからず非常に危険な状態になる可能性があります。一方、リアルタイム OS では、ブレーキ制御タスクの優先度を最高にしていれば、常に最小限の時間でブレーキをかけてくれます。

6.　RTOS の実装方法

マイコンの RTOS は、パソコンのようにあらかじめ OS を入れておいて、そこにいろいろなアプリを乗せていく形ではありません。
自分で作成したプログラムと OS を合体させて実行ファイルをつくり、その実行ファイルをマイコンに入れて使うのです。

パソコンとマイコンの OS の違い

7.　プログラムの設計手順

C 言語を知っている方なら、一般的なプログラムの関数が RTOS でいうタスクだと思ってもらえればイメージしやすいでしょう。一般的な C 言語のプログラムと、RTOS での C 言語プログラムの設計手順を比べてみると以下のように表せます。RTOS といってもほとんど変わらないでしょう。

一般的なプログラムの場合

実現したい動作がある
↓
処理ごとに分割し、関数作成
↓
その組み合わせで動作の実現

RTOS の場合

実現したい動作がある
↓
処理ごとに分割し、タスク生成
↓
その組み合わせで動作の実現

リアルタイム OS について

8.　タスクの状態

RTOS は複数のタスクが協調動作を行うマルチタスク環境です。

　マルチタスクといっても、複数のタスクがひとつの CPU で同時に実行されるわけではなく、ある瞬間に CPU で実行されているタスクはひとつだけです。ですから、タスクは「実行中」のほかにもいくつかの状態があります。

　本コースで用いる TI-RTOS は、以下の 4 つの状態があります。一般的な RTOS のタスクには、このうち RUNNING、READY、BLOCKED に相当する 3 状態があり※、RTOS の種類によっては追加の状態（例えば TI-RTOS では以下の TERMINATED）があることもあります。

■ RUNNING　実行状態

　実行されている唯一のタスク。ただし、この状態の時でも、もっと優先度の高いタスクが READY に入ったとたん、今実行されているタスクは READY に回されてそちらが RUNNING になる。

　人間の仕事で言えば、最優先の仕事なので今は集中してやっているけど、もしもっと優先度の高い仕事が入ったら今の仕事は中断してすぐそちらに移りますよ、という状態。

■ READY　実行可能状態

　優先度の関係で後回しにされているだけで、いつでもスタンバイ OK のタスク群。

　最優先のタスクの実行が終了したり、一時的に待ち状態になると、READY のタスクの中でもっとも優先度の高いタスクが RUNNING に回される。

　RUNNING 中のタスクよりも優先度の高いタスクが READY に入ると、すぐに RUNNING に遷移し、今まで RUNNING 中だったタスクが READY に回される。

　人間の仕事で言えば、優先度の高い仕事が別に入っているので、それが一段落するまでとりあえず後回しにしている状態。

■ BLOCKED　待ち状態

　ユーザーの事情で実行に「待った」がかかっている状態。

　人間の仕事で言えば、お客様からのお返事待ちや時間待ちなどで「まだとりかかれない」状態。

　待ちの要因が解消されれば READY に遷移。

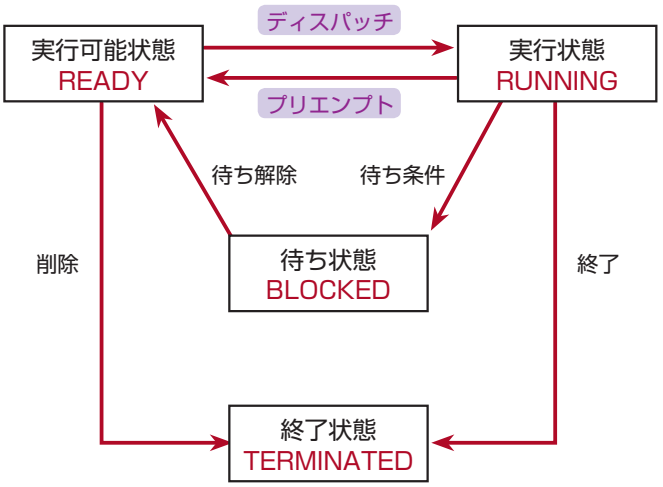
■ TERMINATED　休止状態

　タスクが終了したり、削除されたりして、もう実行されることはない状態。

　人間の仕事で言えば、既に終わった過去の仕事。

※　「BLOCKED」は、RTOS によっては「WAITING」と呼ばれている場合もあります。

リアルタイム OS について



ディスパッチ（プロセッサが実行するタスクを切り替える処理のこと）とプリエンプト（より優先度の高いタスクへ処理権が移ること）は OS によって自動的に行われます。その判断基準はタスクの優先度です。
その他の状態の移行は、ユーザープログラムによって決められます。