Multitasking　マルチタスク

Tasks (or computing processes) are the smallest entities and fundamental elements of asynchronous programming.
タスク(もしくは演算プロセス)は，非同期プログラミングの基本的な要素であり，最小のエンティティである.

They represent the execution of programs in a computing system during the lifetime of the processes.
それらは，プロセスの寿命の中でコンピュータシステムにおいて，プログラムの実行に相当する.

Such a lifetime begins with the registration instant of a process and ends with its cancellation.
そのような寿命は，つかの間のプロセスの登録から始まり，そしてプロセスの取り消しで終わる。

Thus, the computing process exists not only during the actual processing an a functional unit of a computing system, but also before the start of processing and during the planned and forced waiting periods.
それ故に，コンピューティングプロセスは，
実際のコンピューティングシステムの機能単位を処理している間だけでなく，
プロセスの前や，計画され，そして強制された待ち期間の間も存在する.

Tasks are programmed using adequate programming methods, languages and tools.
タスクは，適切なプログラミングの方法や言語，ツールを使うことによってプログラムされた.

The basic requirement for hard real-time tasks is that it must be possible to assess their maximum execution time, called Worst-Case Execution Time (WCET).
ハードリアルタイムなタスクに対する基本的な要求は，WCET と呼ばれるそれらのタスクの最大実行時間を評価することが，可能でなければならない.

A precondition, in turn, for this is that the behavior of the execution platform is deterministic and predictable.
振り返って，これに対する前提条件は，
実行プラットフォームの振る舞いが決定されており，予測可能であることである.

More details about WCET analysis will be given Section 4.2.
WCET に関するより詳細な分析は，4.2 章で示す．

In the following sections, first the concepts of task and multitasking will be dealt with.
続く章では，最初にタスクとマルチタスクの概念を取り扱う．

Then, the ultimate guideline to design architectures for embedded systems – schedulability – will be explored, and a feasible scheduling policy presented.
その時，組み込みシステムに対するアーキテクチャを設計するための基本的な指針　―スケジュール可能性―　が調査され，そして，実行可能なスケジューリングポリシが提示されるだろう．

★スケジュラビリティ　：　すべてのタスクがデッドラインを守ることが出来るスケジュール
(想定した開始時刻から処理を終了すべき時刻（デッドライン）までの時間を「デッドライン時間」)

Finally, the principles of inter-task synchronisation will be explained.
最後に，タスク間の同期の原則が，説明されるだろう．

2.1 Task Management Systems　タスク管理システム

Execution of tasks is administered by system programs.
タスクの実行は，システムのプログラムによって管理される．

Since there are (except in very specific applications) never enough processing units that each task could exclusively be mapped on one of its own, tasks must be arranged or ***scheduled*** in a sequence for execution.
それぞれのタスクは，単独にタスク自身でマッピングされる十分な処理装置では決してないので，タスクは，実行に対してある順番で準備されるか，スケジュールされなければならない．

Sometimes, tasks can be scheduled ***a priori*** during design time.
ときどき，タスクは，設計時間中に経験的にスケジュールされることができる．

This is only possible if the system behavior is completely known in advance like, e.g., for a washing machine, whose tasks filling with water, heating, turning the drum either way, draining water, etc. follow sequences that are always the same.
これは，システムの振る舞いが，洗濯機の例のように完全に前もって知られていることによってのみ可能となる．

洗濯機に関する洗濯機のタスクは，水でいっぱいにし，温め，左右どちらにもドラムを回し，水を抜くなどといった，常に同じ順序に従う.

Such systems are caked static.
そのようなシステムは，静的と呼ばれる.

Here, the execution schedules of parallel tasks can be carefully planned, optimized, and verified off-line during design time.
ここで，並行タスクの実行スケジュールは，設計時間中に入念に計画され，最適化され，そして個別に検証されることができる.

Consequently, this is the safest way to implement embedded applications.
その結果として，これは組み込みアプリケーションを実装するための最も安全な手段である.

Most systems, however, are dynamic:
しかしながら大部分のシステムは，動的である. なぜならば，

at least some of the events requiring service by tasks are known in advance with their parameters, but their occurrence instants are sporadic and asynchronous to other control system processings.
少なくとも，タスクによるサービスを必要とするいくつかのイベントは，その(イベントの)パラメータと一緒に事前に知られている.
しかし，それら(イベント)の発生の瞬間は，散発的で，他の制御システムの処理に非同期である.

In such cases, tasks need to be scheduled at run-time.
そのような事例では，タスクは実行時間にスケジュールされる必要がある.

These systems are more flexible, but their design is also much more challenging.
それらのシステムは，より柔軟性があるが，それらの設計もはるかに困難である.

In this chapter we shall deal with dynamic or mixed systems.
この章では，我々は動的システムまたは混在システムを扱うべきだろう.

There are different approaches to how mixtures of static and dynamic tasks can be executed.
どのように静的タスクと動的タスクの混合物が実行されることができるかへの異なる取り組みが存

在する.

Two representative examples are the ***cyclic executive approach*** and ***asynchronous multitasking***.
2 つの典型となる例は，周期的な執行(方針を立案する)アプローチと，非同期マルチタスクである.

2.1.1 Cyclic Executive　周期的な執行

The simpler approach of two can accommodate both statically scheduled periodic tasks and asynchronous sporadic tasks, although for the latter it can only be used with limitations.
2 つの単純なアプローチは，静的に定期的な周期的タスクと非同期な散発的タスクのどちらも適応することが出来る.
後者の場合，唯一の制限事項を考慮に入れて使用することが出来ます.

The approach is based on periodic execution of a schedule, part of which is determined statically and the other part dynamically.
アプローチは，スケジュールの周期的な実行に基づいている.
スケジュールの一部は，静的に決定され，そしてその他の部分は動的に決定される.

The start of each cycle is triggered by a timer interrupt.
それぞれの周期の開始は，タイマ割り込みによって引き起こされる.

Periodic tasks may have different requirements and need to be executed within their specified periods.
周期的タスクは，異なる要求を持つ可能性があり，特定の期間内で実行される必要がある.

Further, it is assumed that the tasks are completely independent of each other.
その上，タスクが完全に互いに独立であることを仮定した.

The periodic tasks are assigned to be run in cycles according to their requirements.
周期的タスクは，タスクの要求に従って周期的に実行されることを割り当てられる.

The longest cycle is called the ***major cycle***, consisting of a number of ***major cycles*** of different durations.
最長の周期は，異なる継続期間の主要な繰り返しの数で構成されるので，主循環と呼ばれる.

All process periods must be integer multiples of the minor cycle.
すべてのプロセスの期間は，小循環の整数倍でなければならない.

It is possible to include asynchronously arriving sporadic tasks into this scheme:
非同期的にこの構想(スキーマ)に，到着する散発的タスクを含めることが可能である.

they can be executed in the slack time between the termination of the last periodic task in a cycle and the start of the next cycle.
それら(散発的タスク)は，周期において一番最後の周期的タスクの終了と，次の周期の始まりの間の余分な時間で実行されることが出来る.

This also means that the execution time of a sporadic task is limited to this slack; if it is longer, an overrun situation occurs.
これは，散発的タスクの実行時間がこの合間(余分な時間)に限られることも意味する.
もし散発的タスクの実行時間が長ければ，オーバーランの状態が発生する.

A major use of such sporadic tasks is to report exceptions.
そのような離散的タスクの主な活用は，例外を報告することである.

An example of a schedule generated by a cyclic executive for periodic tasks A, B, C and D with periods 1, 2, 2 and 4, respectively, and two sporadic tasks E1 and E2 is shown in Figure 2.1.
スケジュールの例は，それぞれ 1,2,2,4 の期間で周期的タスク A,B,C,D に対して周期的実行によって作り出した.
そして，散発的タスク E1 と E2 は，図 2.1 に示した.

A very simple implementation of the cyclic executive approach is sketched by pseudo-code in Figure 2.2.
周期的実行アプローチのとても単純な実装は，図 2.2 における疑似コードによって概要を示されている.

It is based on an array of task control blocks.
それ(疑似コード)は，たくさんのタスク制御ブロックに基づいている.
★タスク制御ブロック：OS のカーネルにおいて対応するプロセス(タスク)の状態を表すデータ構造

For each task there are parameters: duty cycle, cycle control, and start address.
それぞれのタスクに対し，パラメータが存在する.
すなわち，負荷サイクル，循環制御，開始アドレスである.

Another array mentions signals and addresses of sporadic tasks.
その他の配列は，散発的タスクのアドレスと信号について言及する.

It is assumed that the worst-case execution times are shorter than the smallest slack between the end of the last periodic task and the beginning of the next cycle, which can be calculated at design time.
最悪の場合の実行時間が，一番最後の周期的タスクの終了と，
設計時間で予測することが出来る次の周期の始まりの間の，
とても小さな隙間より短いことを想定している.

Further, it is assumed that sporadic tasks are rare;
さらに，散発的タスクは稀であることを想定されている.

if there are more in a queue they are executed in first-in-first-out (FIFO) manner.
キュー内に複数存在する場合，散発的タスクは，ファーストインファーストアウト(FIFO)方式で実行される.

Finally, there is a check at the beginning whether it came to an overload in the preceding cycle;
最後に，前の周期において，過負荷になったかどうかを始めに検査がある.

that may happen if the worst-case execution times of tasks were exceeded.
検査は，タスクの最悪の場合の実行時間が上回った場合に起こる可能性がある.

(Figure 2.1 and 2.2 exist here.)

It must be noted that the “tasks” here are actually only procedure calls.
ここでいう「タスク」とは，実質的には手続き呼び出しであることだけに留意する必要があります.

No usual tasking operations can be performed; their static schedule assures them exclusive execution slots.
通常のタスク操作は，決して実行されることがない.

静的なスケジュールは，それらの排他的な実行スロットを保証する.

Thus, there is no interdependency, and synchronization with its possible dangers is not necessary.
それ故に，相互依存性がなく，考えられる危険性との同期が必要ない.

Execution of the tasks is temporally predictable in full.
タスクの実行は，完全に一時的に予測可能である.

If it is difficult to construct a feasible schedule off-line due to high system load, different optimization methods can be utilised.
システムの高負荷に起因する個別の実行可能なスケジュールを
構築することが困難な場合，異なる最適化方法を利用することが出来る.

Once a schedule is composed, no further run-time schedulability tests are needs for periodic tasks.
一度スケジュールは構成され，
より一層の実行時間スケジュール可能性のテストは，決して周期的タスクに対して必要ではない.

In the most simple, although not very rare cases, the cyclic executive reduces to periodic invocation of a single task.
とても稀な事例ではなく，最も単純で，周期的な実行は，単一タスクの周期的実施を減らす.

Simple programmable logic controllers (PLC) operate in this manner, first acquiring sensor inputs, then execution control functions, and finally causing process actuations.
単純なプログラマブルロジックコントローラは，この方式で動作する.
最初に取得しているセンサが入力し，その時実行コントローラが動作し，最後に引き起こしているプロセスが作動する.
★プログラマブルロジックコントローラ　：　リレー回路の代替装置として開発された制御装置

A large number of successful implementations of process control application using the cyclic executive can be found.
周期的な実行を利用している，沢山の成功したプロセスコントロールアプリケーションの実装は，見つけることが出来る.

Lawson [76], for instance, has presented a philosophy, paradigm and model called

Cy-Clone (Clone of Cyclic Paradigm), which aims to remove the drawbacks of the cyclic approach while providing deterministic properties for time- and safety-critical systems.
例えば，ローソンは，Cy-Clone(周期パラダイムクローン)と呼ばれるパラダイムとモデルと根本原理を提示した.
Cy-Clone は，タイムクリティカルシステムおよびセーフティクリティカルシステムに対する決定論的なプロパティを規定している，周期的アプローチの欠点を取り除くことを目的としている.

In his cyclic approach, the period should be long enough to allow for all processing, but sufficiently short to ensure the stability of control.
彼の周期的アプローチでは，期間はすべてのプロセスを考慮に入れるために十分長くある必要があるが，制御の安定性を保証するために十分短くする必要がある.

To adapt to dynamic situations, the method can provide mode shifts, where the period varies in a controlled manner during execution.
動的な状況に適応するために，その方法は，モードの変更を提供することが出来る.
その期間は実行の間，制御された方式で変化する.

As Lawson points out, the Cy-Clone approach is not claimed to be the best solution for all real-time systems, but was established on the basis ***"If the shoe fits, wear it"***.
ローソンが指摘するように，Cy-Clone アプローチは，すべての実時間システムに対する最善策であると主張されていないが，「思い当るところがあるならば，改めなさい」ということを基礎として設立された.

In the early 1980s it was employed in a control system of the Swedish National Railway to assist train engineers in following the speed limits along the railway system in Sweden.
1980 年代前半，それは，スウェーデンにおける鉄道システムに従う速度制限以下で，鉄道のエンジニアを支援するために，スウェーデンの国有鉄道の制御システムにおいて採用された.

In 2000, Lawson reported [77] that it was still working (after a non-significant re-design in 1992), but now in high-speed trains!
2000 年に，ローソンは，それ(Cy-Clone アプローチ)が(1992 年に無意味な再構築後)未だに働いている[77]を報告した.
しかし，現在それは高速の列車の中で働いている.

2.1.2 Asynchronous Multitasking　非同期マルチタスク

Although convenient and safe, the above approach cannot be used for dynamic systems, where all r most tasks are asynchronous.
使い易く安全であるが，上記のアプローチは，動的なシステムに対しては利用することが出来ない．

In such cases, real asynchronous multitasking needs to be employed.
そのような場合に，実際の非同期マルチタスクが採用される必要がある．

In contrast to the static cyclic executive, timeliness and simultaneity are to be provided dynamically.
静的な周期的実行と対称的に，瞬時性と同時性は，動的に提供されるべきである．

The dynamic load behavior requires scheduling problems to be solved with an adequate strategy.
動的負荷挙動は，適切な方策で解決されるべきスケジューリングの問題を要求する．

The demand for simultaneity implies ***pre-emptability*** of tasks:
同時性に対する需要は，タスクの先取りを暗示する．

a running task may be pre-empted and put back into the waiting state if another task needs to be executed according to the scheduling policy.
すなわち，実行しているタスクは，占領することができる．そして，他のタスクが，スケジューリング方針に従って実行すべき必要のある場合，(実行しているタスクは)待機状態に戻る．

However, as a consequence this leads to synchronization requirements.
しかしながら，結果としてこれは，同期の必要条件につながる．

Context-switching, i.e., swapping the tasks that are executing on a processor, starts with saving the context (i.e., the state consisting of program counter, local variables, and other internal task information) of the executing task to storage in order to allow its eventual resumption when it is rescheduled to run.
コンテキストの切り替え，すなわち，プロセッサ上で実行するタスクを入れ替えることとは，
実行するためにスケジュールを変更された時，最終的な結果を考慮するために，ストレージに実行中のタスクのコンテキスト(すなわち，プログラムカウンタで構成されている状態，ローカル変数，そしてその他の内部のタスクの情報)を確保することから始まる．

This storage is often called Task Control Block, (TCB) and also includes other information about tasks, e.g., their full and remaining execution time, deadlines, resource contention, etc.
このストレージは，大抵タスクコントロールブロックと呼ばれており，
例えば，タスクの最大限の実行時間，残りの実行時間，デッドライン，リソース競合などといった，
タスクに関するその他の情報も含んでいる．

Then, the state of the newly scheduled task is loaded and the task is run.
その後，新たにスケジュールされたタスクの状態が，読み込まれ，タスクが実行される．

Obviously, as this represents a non-productive overhead consuming certain processing capacity, it should be kept to a minimum by choosing the most adequate task scheduling policy.
明らかに，これが生産効率の悪いオーバーヘッドを使いきる特定の処理能力を意味するように，
それは，最も適切なタスクのスケジュール方針を選択することによって，最小限に抑えるべきである．

★オーバーヘッド　：　ある処理を行うとき、作業そのものに必要な正味のコスト（処理時間・必要メモリー量など）とは別に、作業の準備・管理・後処理などで必要となる付帯的コスト

Multitasking must be supported by operating systems.
マルチタスクは，オペレーティングシステムによって支援されなければならない．

Any one mainly supports
its own scheme of tasks states and transitions or has, at least, different names for them.
主にどれか一つは，タスクの状態と遷移の自身のスキーマを支援する，または少なくともタスクに対する異なる名前を持つ．

A typical, simple but sufficient task state transition diagram is presented in Figure 2.3.
典型的な，シンプルでありながら，十分なタスクの状態遷移図は，図 2.3 に示す．

In this model, tasks may assume one of the following states:
このモデルでは，タスクは次のいずれかの状態を前提とすることが出来る．

--

Dormant (also known or terminated).

休止状態 (また，知られている状態，もしくは終了した状態)
Once a task is introduced in a system by its initialisation, it enters the dormant state.
タスクは，初期化によってシステムに取り入れられると，タスクは休止状態に入る.

Here it is waiting to be activated by an event.
ここで，タスクは，イベントによって活性化されるのを待っている.

Further, when the task accomplishes its mission and ***terminates*** in the “running” state, it is brought back to the dormant (or, in this case better, terminated) state.
さらに，「実行中」状態において，タスクがタスクの目的を遂行し，終了させた時，タスクは休止状態(この場合は良く，または終了させられた状態)に戻される.

If for any reason a task is not needed any more in any other state, it can be ***aborted***, i.e., put into the dormant state again.
何らかの理由により，タスクが他の状態においてそれ以上必要とされない場合，タスクは停止されることが出来る. すなわち，再び休止状態におかれる.

--

Ready.
実行可能状態

A task in the dormant state can be ***activated*** by an event and then joins the set of tasks ready to be scheduled for execution.
休止状態におけるタスクは，イベントによって活性化されることが出来，その後，
実行するためにスケジュールされることができる準備の出来たタスクの集合に加えられる.

Such a task enters the ready state.
そのようなタスクは，実行可能状態に入る.

A task can also enter this state when it is ***pre-empted*** from running, or ***resumed*** from suspension.
停止状態から再開された時，または実行状態から割り込まれた時，タスクは，この状態にも入ることができる.

Upon occurrence of any event changing the set of ready tasks, a scheduling algorithm is invoked to order them, according to a certain policy, in a sequence in which they will be

executed.
実行可能状態のタスクの集合を変更する任意のイベントの発生時に,
スケジューリングアルゴリズムは，特定のポリシーに従って，タスクが実行されるであろう順序でタスクを順序づけるために起動される.

If a task from the ready set has lost its meaning for any reason (e.g., it was explicitly aborted by another task, or its waiting time for resources has exceeded, so that its deadlines cannot be met anymore) it can be ***aborted*** to the dormant state.
実行可能な集合からタスクが，何らかの理由(例えば，それが別のタスクによって明確に中断された，またはデッドラインがもはや満たされることが出来ないため，リソースに対する待ち時間が超過した)のために意味を失ってしまった場合,
実行可能な集合のタスクは，休止状態に打ち切られることが出来る.

--

Running.
実行中状態.

As a result of ***scheduling*** the tasks in the ready state, the first task in the sequence is dispatched to the running state, i.e., to be run on the processor.
実行可能状態におけるタスクをスケジューリングした結果として,
順番の最初のタスクは，実行中状態に送りだされる.
例えば，プロセッサ上で実行されることが出来る.

Eventually, it is ***terminated***, when finished, or ***aborted*** to the dormant state.
最終的に，休止状態へ打ち切られるか，終えた時，それは終了する.

A running task can be ***pre-empted***, i.e., its execution is discontinued and the processor allocated to another task, and it is brought back into the ready state to re-join the set of ready tasks for further scheduling.
実行中のタスクは，割り込まれる可能性がある．すなわち，実行が中断され，プロセッサがそのほかのタスクに割り当てられ，そしてさらなるスケジューリングのための一連の実行可能タスクを再結成するために再び元に戻す.

Pre-emption is related to certain overheads, so one always strives to minimize the number of pre-emptions to the lowest amount possible.
割り込みの1つは常に，可能な限り少なく割り込みの数を最小限にするために励んでいるため,

割り込みは，特定のオーバーヘッドに関連している.

--

Suspended.

中断状態

If the running task lacks certain resources for further execution, or if any other precedence relation calls for deferral through a synchronization mechanism, its execution is ***suspended***.

実行中のタスクをさらに実行するための特定のリソースが不足している場合，または他の優先順位関係が，同期メカニズムを介して延期を要求する場合，実行中タスクの実行は，中断される.

Once a certain condition that the suspended task is waiting for is fulfilled, it is ***resumed*** and, thus, brought into the ready state again.

実行されることを中断状態のタスクが待っているという状態は，再開され，こうして，再び実行可能状態になる.

If, on the other hand, due to excessive waiting, the task cannot meet its deadline, it is aborted and a certain prepared action is taken in order to deal with this situation.

一方で，過度な待ちが原因で，タスクがデッドラインを達成できない場合，

タスクは中断され，一定の準備行動がこの状況に対処するためにとられる.

Task transitions are performed when certain events occur.

特定のイベントが発生した時，タスクの遷移が実行される.

These may be external events, e.g., signals from the process environment or arrivals of messages, timing events, when a time monitor recognizes an instant for triggering a task, or internal events, when, for instance, a semaphore blocks execution of a task.

タイムモニタが，タスクまたは内部イベントを引き起こすための瞬間を認識した時，例えば，セマフォがタスクの実行を阻止する時，

それらは外部イベントとなるかもしれない.

例えば，メッセージの到着や，プロセス環境からの信号や，タイミングイベント

★タイミングイベント：指定の時間間隔後にコードを実行することが可能であること(Java)

Internal events can also be induced by other tasks.

内部イベントも，その他のタスクによって誘導される可能性がある.

Instead of explicit operating system calls, tasking operations should be supported by adequate language commands.
明確なオペレーティングシステムコールの代わりに，タスクの操作は，適切な言語コマンドによって支援されるべきだ.

An example of an apt set of tasking commands will be given in Section 4.1.2 in Figure 4.6.
タスクコマンドの適切な集合の例は，図 4.6 において 4.1.2 章で示す.

2.2 Scheduling and Schedulability
スケジューリングとスケジュール可能性

In Section 1.2.1 it has been established that predictability of temporal behavior is the most important property of embedded and real-time applications.
1.2.1 章では，一時的な振る舞いの予測可能性は，リアルタイムアプリケーションと組み込みアプリケーションの最も重要なプロパティであることが確立されている.

The final goal is that tasks complete their functions within predefined time periods.
最終的な目標は，タスクが所定の時間周期以内にタスクの機能を完了することである.

Thus, when running a single task it is enough to know its execution time.
従って，単一のタスクが実行する時，その実行時間を知るには十分である.

Dynamic embedded systems, however, must support multitasking.
しかしながら，動的な組み込みシステムは，マルチタスクを支援すべきである.

There is always a set of tasks which have been invoked by certain event occurrences, and which compete for the same resources, most notably the processor(s).
常に特定のイベントの発生によって引き起こされたタスクの集合，そして
同じリソース，中でも注目すべきはプロセッサを得るために争うタスクの集合が存在する.

Since all of them have their deadlines to meet, an adequate ***schedule*** should be found to assure that.
それらのすべてが，その期限を満たさなければならないので，適切なスケジュールは(期限を)保証するために発見されるべきである.

The pre-condition for being able to elaborate all these tasks is, of course, the availability of resources.
すべてのこのようなタスクを構成できるようにするための事前条件は，もちろん，リソースの利用できる度合いである.

Lawson [76] presented the concept of ***resource adequacy*** meaning that there are sufficient resources to guarantee that all processing requirements are met on time.
ローソンは，すべてのプロセッサの要求が時間どおりに達成されると保証するための，十分なリソースがあることを達成するための，リソースの妥当性の概念を提示した.

A schedule to process a task system is called ***feasible*** if all tasks of the system meet their deadlines (assuming the above mentioned resource adequacy).
上記のリソースの妥当性を仮定するので,
システムのすべてのタスクがそれらのデッドラインを達成する場合,
タスクシステムを処理するためのスケジュールは，実行可能であると呼ばれる.

A task system is called ***feasibly executable*** if there exists a feasible schedule.
実行可能なスケジュールが存在する場合,
タスクシステムは，実行できるように実行可能であると呼ばれる.

At run-time, a ***scheduler*** is responsible for allocating the tasks ready to execute in a such way that all deadlines are observed, following an apt processor allocation strategy, i.e., ***a scheduling algorithm.***
実行時間で，適切なプロセッサ割り当て戦略，すなわちスケジューリングアルゴリズムに従うため,
スケジューラは，すべてのデッドラインが観測されるような状態で
実行する準備が出来ているタスクを割り当てることに関与する.

A strategy is feasible if it generates for any feasibly executable (free) task set a feasible schedule.
アルゴリズムが，任意の実行できるように実行可能な(使われていない)タスクが実行可能なスケジュールを設定するために生成された場合,
戦略(スケジューリングアルゴリズム)は実行可能である.

Another notion concerning feasibility of a task set is ***schedulability:***
一連のタスクの実現可能性に関する別の概念は，スケジューリング可能性です.

the ability to find, ***a priori***, a schedule such that each task will meet its deadline [108].
すなわち，経験的に，それぞれのタスクがそれぞれのタスクのデッドラインを達成することができるようなスケジュールを探すための能力

Sometimes it may happen that a task cannot meet its deadline;
時には，タスクがタスクのデッドラインを達成することが出来ないことが起こる可能性がある.

in hard real-time systems this is considered a severe fault.
ハードリアルタイムシステムにおいて，これは重大な障害と見なされる.

It can be a consequence of improper scheduling, or of system overload.
それ(重大な障害)は，不適切なスケジューリング，またはシステムのオーバーヘッドの結果である可能性がある.

In the former case this is due to selecting an improper or infeasible scheduling strategy.
前者(不適切なスケジューリングによる障害)の場合には，
これは，不適切なスケジューリング戦略，または不可能なスケジューリング戦略を選択することによるものである.

In the latter case the specification of system behavior and, thus, the dimensioning of the computer control system was wrong.
後者(システムのオーバーヘッドによる障害)の場合には，
システムの動作の仕様と，上に述べたように，コンピュータ制御システムのディメンジョニングが誤っていた.

To prevent such specification errors, it is essential to have – prior to system design – a clear understanding of the peak load a system is expected to handle.
そのような仕様の誤りを防ぐために，
システムが処理することが期待されるピーク時の負荷をはっきりと理解することが不可欠である.

As it is not possible to design, for instance, a bridge without knowing the load it is expected to carry, so it is not possible to design a real-time system with predictable performance without estimating the load parameters and task execution times.
例えば，支えることが要求される負荷(要求される負荷への耐性)を知らないで，橋を設計することは不可能なように，

タスクの実行時間と負荷パラメータを見積もることなく予測可能なパフォーマンスでリアルタイムシステムを設計することはできない.

In order to estimate correctly the necessary capacity of a system and, thus, prevent overload situations, a proof of feasible executability or scheduability of occurring tasks set has to be carried out in the planning phase.
正しく必要なシステムの容量と，過負荷状態を推定するために，
発生するタスクセットが実行できるような実行可能性またはスケジュール可能性の証明は，計画段階で実行すべきだ.

This is a difficult problem, especially in dynamic systems with task reacting to sporadically occurring events.
これは，特に離散的に発生するイベントに反応するタスクを含む動的なシステムにおいて，難しい問題である.

There are methods that may provide such proof, cf. e.g., [108].
そのような証明を提供することができる方法が存在する. 例えば，[108]を参照.

Their complexity, however, can easily become NP-complete, and the methods may yield rather pessimistic estimations.
しかしながらその複雑さは，簡単に NP 完全になることができ，また
その方法は，かなり悲観的な推定を生じさせる可能性がある.

A decision problem C is NP-complete if
1. it is in NP and …NP と呼ばれる問題の集まりは、問題の答え（Yes/No）とその「求め方」がわかってしまえば、その答えが正しいかどうかのチェックは「素早く」できてしまう問題、そういった問題の集まり
2. it is NP-hard, i.e. every other problem in NP is reducible to it. …NP-hard と呼ばれる問題の集まりは、（すでに述べた）NP と呼ばれる問題のどの問題よりも、同じかそれ以上に難しいといえる問題の集まり

2.2.1 Scheduling Methods and Techniques
スケジューリングメソッドとスケジューリング技術

Scheduling policies fall into two classes, depending on whether the tasks, once executing, can be pre-empted by other tasks of higher urgency or not.
スケジューリング方針は，　一度実行するタスクが，
より緊急度の高い他のタスクによって割り込まれる可能性があるか，ないかどうかによって，

2 つのクラスに分類される.

For scheduling dynamic systems, pre-emptions are necessary in order to assure feasibility of the scheduling algorithms.
動的なシステムをスケジューリングするために,
割り込みは, スケジューリングアルゴリズムの実現可能性を保証するために必要である.

In this subsection, two scheduling policies, fixed priority and rate monotonic scheduling, will be elaborated.
このセクションでは, 優先順位を固定し, 単調なスケジューリングを評価する
2 つのスケジューリング方針を詳細に述べるつもりだ.

Deadline driven scheduling policies, which are problem-oriented and more suitable, will be covered in more detail in the next subsection.
問題指向で, かつより適切なデッドライン駆動スケジューリング方針は,
次のセクションでより詳細に扱うつもりだ.

--

Fixed Priorities
固定の優先順位

Many Popular operating systems, which are currently most frequently employed in embedded systems, base their scheduling on fixed priorities.
世間一般に組み込みシステムで最も頻繁に採用されている
多くの大衆向けのオペレーティングシステムは, 固定の優先順位に基づいてスケジューリングを行う.

The advantage is that the latter are in most cases built into the processors themselves in a form or priority interrupt handling systems.
その利点は, 後者(2 つのスケジューリング方針の？)がフォーム, または優先順位の割り込み処理システムにおける, プロセッサ自身に組み込まれているほとんどの事例であるということである.

Thus, implementation is fast and simple.
従って, 実装は, 迅速かつ簡単である.

As scheduling criteria, however, priorities are not problem-oriented, but artificially invented categories not emerging from the nature of the applications.
しかしながら，スケジューリングの基準として，優先順位は，問題指向ではなく
人為的にアプリケーションの性質から生じないカテゴリを発明した.

As a rule, it is difficult to assign adequate priorities to tasks, bearing in mind all their properties (frequency of their invocations, required resources, temporal tightness, relative importance, etc.).
原則として，
すべてのそれらのプロパティ(それらの呼び出しの頻度，要求されるリソース，一時的な圧迫感，相対的重要性)を念頭に置くとき，適切な優先順位をタスクに割り当てることは難しい.

Designers tend to over-emphasise relative importance, which leads to starvation of other tasks waiting for blocked resources.
設計者は，妨げられたリソースを待っているその他のタスクの処理停止(窮乏)につながる
相対的重要性を過剰に協調する傾向がある.

Priorities are not flexible and cannot adapt to the current behavior of systems.
優先順位は柔軟ではなく，システムの現在の振る舞いに順応することが出来ない.

For these reasons, it is not possible to prove feasibility of such a policy.
これらの理由から，そのような方針の実現可能性を証明することはできない.

Once a task is assigned a fixed priority, the scheduler allocates the resources regardless of real needs of that and other tasks.
タスクは，固定の優先順位を割り当てられると，
スケジューラは，他のタスクと他のタスクの真のニーズにかかわらず，リソースを割り当てる.

As an example, let us consider a situation of three tasks with arbitrarily assigned priorities; see Figure 2.4.
例として，任意に割り当てられた優先順位で 3 つのタスクの状況を考えてみましょう.
すなわち，図 2.4 を見よ.

As we can see, the selection of priorities was unfortunate:
我々が見てわかるように，優先順位の選択は間が悪い.

the lowest priority task misses its deadlines, although the laxities of the tasks are relatively large.
タスクの緩みが比較的大きいけれども,
最も優先順位の低いタスクは，そのデッドラインを逃す.

Possibly, the importance of a task suggested it be assigned the highest priority, although its deadline is far and execution time short.
おそらく，タスクの重要性は,
タスクのデッドラインが，2 つのうち遠い方であり，実行時間が短いけれども,
タスクが最も高い優先順位を割り当てられること
を意味する.

A feasible schedule, however, is possible by simply overturning the priorities, as shown in Figure 2.5.
しかしながら，実行可能なスケジュールは，図 2.5 で示すように,
単純に優先順位を覆すことが可能である.

Further, a number of problems emerge from the priority-based scheduling policy.
さらに，多くの問題は，優先順位指向のスケジューリング方針から現れる.

For instance, if there is as set of tasks of different priorities requesting the same resources, it could easily result in priority inversion:
例えば，同じリソースを要求している異なる優先順位のタスクの集合がある場合,
それは，簡単に優先順位の逆転をもたらすことが出来る.

a low-priority task pre-empted by a higher-priority one is blocking a resource and, thus, the execution of all those that wait for the same resource.
低い優先順位のタスクは，高い優先順位のタスクがリソースを妨げていることによって割り込まれる．従って，それらの実行は同じリソースを待つことになる.

In this case the waiting time cannot be bounded.
この事例では，待ち時間は境界をつけられない.

The situation is depicted in Figure 2.6.
その状況は，図 2.6 で表現されている.

One of the attempts to cure this inherent problem uses priority inheritance; see Figure 2.7.
この固有の問題を解決しようとする試みの一つは，優先順位の継承を利用することである．
すなわち，図 2.7 を見よ．

However, the system is now behaving differently than specified, namely a task is given a different priority than originally assigned, which disturbs the global relationships, especially if the resource is allocated for a longer time.
しかしながら，システムは，すぐに指定されたものと異なった振る舞いをする．
すなわち，タスクは
特にリソースが長時間割り当てられている場合 広範囲の関係を妨げる
最初に割り当てられた優先順位とは異なる優先順位を与えられる．

The higher-priority tasks are still delayed by the lower, although for a shorter time.
高い優先順位のタスクは，
短時間に関してではあるが，低い優先順位のタスクによってさらに遅らされる．

Further, a track must be kept of which resources are allocated by which tasks.
さらに，トラックは，リソースがタスクによって割り当てられることを
保持し続けられなければならない．

Another widely used, but only slightly better policy is round-robin scheduling, where each of the ready tasks is assigned a time slice.
もうひとつ広く使われている，
ほんの少し良い方針は，それぞれの実行可能状態タスクがタイムスライスを割り当てられる，ラウンドロビンスケジューリングである．

Needless to say, the policy is not feasible unless the temporal circumstances of all competing tasks are more or less the same.
言うまでもなく，方針は，
程度の差はあるが(おおよそ)同じであるすべての競合するタスクの一時的な状況を除いて，
実行可能ではない．

--

Rate-Monotonic Priority Assignment
割合が単調な優先順位割り当て

There are other policies to assign priorities to tasks which are more adequate to solve the scheduling problem.
スケジューリングの問題を解決するために，より適切なタスクに優先順位を割り当てる，
その他の方針がある.

Well-known and particularly suitable for periodic tasks is rate-monotonic scheduling.
良く知られており，特に周期的タスクに対して適しているものは，割合が単調なスケジューリングである.

It follows the simple rule that priorities are assigned according to increasing periods of periodic tasks – the shorter the period the higher the priority.
それ(rate-monotonic scheduling)は，
優先順位が，周期的タスク(短い期間では優先順位が高くなる)の増加する期間に応じて割り当てられる，単純な規則に従う.

The policy is feasible, and it is possible to carry out schedulability tests.
その方針は実行可能である．そして，スケジュール可能性のテストを実行することが可能である.

In their widely recognized publication [80], in 1973 Lie and Layland considered periodic task sets whose relative deadlines equal their periods.
広く認知されている出版物[80]において，
1973 年に Lie と Layland は，
それらの期間に等しい，周期的タスクの集合の相対的なデッドラインを検討した.

All $\boldsymbol{n}$ tasks were assumed to start simultaneously at $\boldsymbol{t = 0}$ and to be independent, i.e., there exist no precedence relations or shared resources.
すべての n 個のタスクは，t=0 で同時に開始することと，互いに独立であると仮定した.
すなわち，優先権の関係または共有されたリソースは存在しない.

Under these assumptions they proved that rate-monotonic priority assignment yields a feasible schedule *if*
それらの過程の下で，彼らは，以下の式の場合，
割り当ての単調な優先順位割り当てが，実行可能なスケジュールを生成することを
証明した.

$$U \leq n\left(2^{\frac{1}{n}} - 1\right) \qquad (2.1)$$

where $\boldsymbol{U}$ is the processor utilization
U は，プロセッサの使用率である．

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \qquad (2.2)$$

The utilization of the processor by $\boldsymbol{n}$ rate-monotonically scheduled tasks is shown in Table 2.1.
n の割合で単調にスケジュールされたタスクによる
プロセッサの使用率は，表 2.1 に示される．

As an illustration, a Gantt chart of the feasible schedule of three tasks with their priorities assigned rate-monotonically is shown in Figure 2.8.
実例として，割合単調に割り当てられた優先順位を持つ 3 つのタスクの実行可能なスケジュールのガントチャートは，図 2.8 に示される．

The overall utilization is 0.72, with the maximum for three tasks (from Table 2.1) being 0.78.
0.78 となっている 3 つのタスク(表 2.1 より)に対する最大値とともに，
全体の使用率は 0.72 である．

In the next sample in Figure 2.9, the attempted utilization is 85%.
図 2.9 における次の例において，計画された使用率は 85%である．

As a result, Task 3 only receives four units of service time in its first period, is then pre-empted by Task 1, followed by Task 2, and misses its deadline.
結果として，タスク 3 だけが，最初の期間においてサービス時間の 4 つのユニットを受取った時に，タスク 2 を受けて，タスク 1 によって割り込まれ， そして，タスクのデッドラインに届かない．

For a large number of tasks, Equation 2.1 returns the feasibility bound
タスクの数が多い場合，式 2.1 は，下式で制約された可能性を返す．

$$\lim_{n\to\infty} n\left(2^{\frac{1}{n}} - 1\right) = \ln 2 \cong 0.69$$

Thus, it is proven that the policy provides an adequate and feasible schedule for any set of ready tasks as long as the utilization of the processor is below 0.69

従って，プロセッサの使用率が 0.69 を下回っている間は，
その方針は，任意の実行可能状態のタスクに対する，適切かつ実行可能なスケジュールを提供することを証明される.

Renouncing 30% of processor performance and using rate-monotonicity, it can be guaranteed that the schedule is always feasible.
プロセッサの性能の 30%を放棄し，割合単調性を利用する時，
それは，スケジュールが常に実行可能であることを保証されることができる.

The feasibility bound of Equation 2.1 is sufficient but not necessary, i.e., it is possible that a task set will meet its deadlines under rate-monotonic scheduling although the condition is violated.
式 2.1 の実行可能の限界は，必要ではないが，十分である.
すなわち，タスクの集合が，
条件を違反しているけれど，割合単調なスケジューリングの下で，タスクのデッドラインを達成することが出来ることを可能にする.

Bini et al. [8] proved that a periodic task set is feasible under rate-monotonic scheduling as long as
Bini らは，
周期的タスクの集合は，以下の式を満たす間，割合単調スケジューリングの下で，実行可能であることを証明した.

$$\prod_{i=1}^{n}\left(\frac{C_i}{T_i}+1\right) \leq 2.$$

This schedulability test, which is known as the Hyperbolic Bound, allows higher processor utilization.
このスケジュール可能性テストは，高いプロセッサの利用を考慮する，双曲線の範囲として知られている.

It tasks into account that the schedulability bound improves if the periods have harmonic relations.
それ(テスト)は，スケジュール可能性の限度が，その期間が高調波の関係を持っている場合に，改善することを考慮する.

The rate-monotonic scheduling policy relates to periodic tasks.

割合単調スケジューリングの方針は，周期タスクに関連している.
With the assumption that non-periodic tasks can only be invoked once within a certain period of time, in the worst case sporadic tasks become periodic;
非周期タスクのみ，一定の期間内に一度だけ呼び出すことが出来るという仮定の下で，
最悪の場合，離散的タスクは，周期的になる.

thus, rate-monotonic scheduling can also be used for them.
従って，割合単調スケジューリングも，それら(非周期タスクと離散的タスク？）に対して利用することが出来る.

2.2.2 Deadline-driven Scheduling
デッドライン駆動スケジューリング

Following the ultimate requirement for real-time systems, viz., that all tasks meet their deadlines, leads to problem-oriented deadline-driven scheduling.
リアルタイムシステム，すなわち すべてのタスクがタスクのデッドラインを達成する, リアルタイムシステムに対する究極的な要求に従うことは，問題指向デッドライン駆動スケジューリングを導く.

The artificial priorities of tasks are no more relevant.
人為によるタスクの優先順位は，もはや関連性がない.

In some adequate scheduling algorithms, properties are only used in oder to handle system overloads;
いくつかの適切なスケジューリングアルゴリズムにおいて，
優先順位は，システムの過負荷を操作するためにだけに利用される.

in this case, priorities may indicate which tasks must be retained under any circumstances, and which one may be gracefully degraded or renounced.
この場合，優先順位は，
タスクがいかなる状況下において保持されるべきか，また，
もうひとつ(？）が適切に分解されるか，放棄される可能性があることを意味するかもしれない.

There are several deadline-based algorithms, actually rate-monotonic scheduling as described above being one of them.
いくつかのデッドラインに基づくアルゴリズムは存在する.
実際に，そのうちの一つである上記で説明したような割合単調スケジューリングである.

Some observe deadlines themselves, and others schedule tasks according to slack or margin (the difference between the time to the deadline and the remaining execution time of a task).
いくつかはデッドライン自身を観察し,
その他のスケジュールは，スラックやマージン(タスクの時間とデッドラインの間の差と，タスクの実行時間の残り)に応じて，タスクを実行する.

One of the most suitable scheduling algorithms is the ***earliest deadline first scheduling algorithm*** [80].
最も適したスケジューリングアルゴリズムの一つは，EDF スケジューリングアルゴリズムである.

★EDF:リアルタイムオペレーティングシステムで使用される動的スケジューリング規則の一種.
スケジューリングイベントが発生すると(タスク終了、新規タスク生成など)、そのキューを探索して最も実行期限(デッドライン)が近いプロセスを選ぶ

Tasks are scheduled in increasing order of their deadlines, see Figure 2.10.
タスクは，タスクのデッドラインの昇順で，スケジューリングされる. 図 2.10 を見よ.

It was shown that this algorithm is feasible for tasks running on single processor systems;
図は，このアルゴリズムが，シングルプロセッサシステムにおいて実行しているタスクに対して，
実行可能であることを示す.

with the so-called throw-forward extension it is also feasible on homogeneous multiprocessor systems.
いわゆる，スローフォワードの拡張機能とともに，
それは，同種のマルチプロセッサシステムにおいても実行可能である.

However, this extension leads to more pre-emptions, is more complex and, thus, less practical.
しかしながら，この拡張機能がより多くの割り込みをもたらすことは，より複雑である.
従って，実用性は少ない.

To be able to employ the strategy earliest deadline first, it is best to structure real-time computer systems as single processors.
EDF の戦略を用いることが出来るようにするために，シングルプロセッサのようなリアルタイムコ

ンピュータシステムを構築することが最善である.

The idea can be extended to distributed systems, by structuring them as sets of interconnected uni-processors, each of which dedicated to control a part of an external environment.
そのアイデアは，相互接続された，
外部環境の一部を制御することに専念したそれぞれのユニプロセッサのセットとしてそれらを構築することによって，分散システムを拡張することが出来る.

This is actually not a very critical limitation because of the nature of application tasks occurring in control.
これは実際に，
制御に追いて発生するアプリケーションタスクの性質であるため，極めて重要な制限ではない.

To handle the case of overloads, many researchers are considering load sharing schemes which migrate tasks between the nodes of distributed systems.
オーバーロードの場合を処理するために，多くの研究者は，
分散システムのノード間でタスクを移行する負荷分散スキーマを検討している.

In industrial process control environments, however, such schemes are generally not applicable, because only computing tasks can be migrated.
しかしながら，工業的過程制御環境では，
唯一のコンピューティングタスクが移行される可能性があるため，
そのようなスキーマは，一般的には適用できない.

In contrast to this, control tasks are highly input/output bound, and the permanent wiring of the peripherals to certain nodes makes load sharing impossible.
これとは対照的に，制御タスクは，非常に入出力の限界があり，
特定のノードへの周辺機器の常設配線は，負荷を共有することが不可能になる.

Therefore, the earliest deadline first scheduling algorithm is to be applied independently from considerations of the overall system load on each node in a distributed system.
従って，EDF スケジューリングアルゴリズムは，分散システムにおける各ノードでシステム全体の負荷の結果から独立して適用されるべきだ.

The implementation of this scheme is facilitated by the fact that, typically, industrial process

control systems are already designed in the form of co-operating, possibly heterogeneous, single processor systems, even though the processours' operating systems do not (yet) schedule by deadline.
このスキーマの実装は,
一般的に, たとえプロセッサのオペレーティングシステムが, デッドラインによって(まだ)スケジュールされていないとしても,
工業的過程プロセス制御システム, おそらく異種の, シングルプロセッサシステムは, すでに協力する形で設計されている
という事実によって促進される.

Another scheduling policy based on deadlines, ***least laxity first***, schedules tasks according to their slack.
その他のスケジューリング方針は, デッドラインに基づく.
LLF, スケジュールはそれらの余裕時間に応じてタスクを実行する.

★LLF:プロセスの *slack time* (余裕時間)に基づいて優先度を設定する. 一般に組み込みシステム、特にマルチプロセッサシステムで使用

It was shown that it is feasible for both single- and multiprocessor systems.
それは, シングルプロセッサとマルチプロセッサシステムの両方で実行可能であることが示された.

By the algorithm, the task with the least time reserve is executed first; see Figure 2.11.
アルゴリズムによって, 最小時間タスクの制限は, 最初に実行される. 図 2.11 を見よ.

As a consequence, its reserve is maintained (accumulated execution time and the distance to the deadline are both running at the same speed), but the reserves of other, waiting tasks vanish.
結果として, その制限が保持される(累積実行時間とデッドラインまでの距離が両方とも同じ速さで実行する).
しかし, その他の制限, 待ち状態のタスクが消滅する.

Thus, the reserve of one or more tasks being the next smallest reserve eventually reaches the executing task's reserve.
従って, 次の最小の制限となる 1 つ以上のタスクの制限は, 最終的に実行中のタスクの制限を達成する.

From this moment on, the processor must be shared among the two or more tasks in round-robin fashion.
この瞬間から，プロセッサは，ラウンドロビン方式で 2 つ以上のタスク間で共有されるべきだ.

Now, their reserves are not maintained anymore, since their execution is shared, but they expire more slowly than the ready tasks not execution.
それらは，実行しない実行可能状態のタスクより，ゆっくり終了するが，
それらの実行は共有されるので，
今，それらの制限は，もはや維持されない.

Sooner or later, the reserves of other waiting tasks will reach theirs, so that these tasks also join the pool for switched execution.
遅かれ早かれ，それらのタスクは，交換された実行に対してプールに参加するため，
その他の待ち状態のタスクの制限は，それらに到達するだろう.

Obviously, the overhead for context-switching caused in this way is high and usually unacceptable.
明らかに，この方法で生じる実行するプログラムの切り替えのためのオーバーヘッドは，高く，通常は受け入れられない.

For that reason, the least laxity first algorithm is mainly of theoretical interest only.
従って，LLF アルゴリズムは，大部分は理論上興味深いだけである.
(つまり，実際は使えないかも)

In contrast to least laxity first, the earliest deadline first algorithm does not require context-switches unless a new task with an earlier deadline arrives or an executing task terminates.
LLF とは対照的に，EDF アルゴリズムは，
以前のデッドラインで新たなタスクが到着するか，実行中タスクが終了する場合を除いて，
実行するプログラムの切り替える必要がない.

In fact, if the number of pre-emptions enforced by a scheduling procedure is considered as a selection criterion, the earliest deadline first algorithm is the optimum one.
実際に，スケジューリング手順によって実行される割り込みの数が，選択基準として考慮される場合，

EDF アルゴリズムは，最適なアルゴリズムである.
Even when tasks arrive dynamically, this policy maintains its properties and generates optimum pre-emptive schedules [74].
タスクが動的に到着した場合であっても，
この方式は，プロパティを維持し，最適な割り込みスケジュールを生成する.

This scheduling strategy establishes the direction in which an appropriate architecture should be developed.
このスケジューリング戦略は，
適切なアーキテクチャが明らかにされるべき方向性を確立する.

The objective ought to be to maintain, as much as possible, a strictly sequential execution of task sets.
可能な限り，目的は，厳密にタスクセットの順次実行を維持しておくべきだ.

The processor(s) need(s) to be relieved of frequent interruptions caused by external and internal events in the sequential program flow.
プロセッサは，逐次プログラムの流れにおいて内部イベントと外部イベントによって引き起こされる頻繁な中断から解放される必要がある.

These interruptions are counter-productive in the sense that they seldom result in immediate (re-) activation of tasks.
それらの中断は，ほとんど直近のタスクの(再)活性化をもたらさない，という意味では，非生産的である.

2.2.3 Sufficient Condition for Feasible Schedulability Under Earliest Deadline First

EDF の下で実行可能なスケジュール可能性に対する十分条件

When deadlines and processing requirements of tasks are available ***a priori***, the earliest deadline first algorithm schedules the ready tasks by their deadlines.
デッドラインとタスクの処理の要求が，経験的に得られる時，
EDF アルゴリズムは，それらのデッドラインによって実行可能状態のタスクをスケジュールする.

For any time t, $0 \leqq t < \infty$, and any task $T \in \mathbf{F}(t)$ with deadline $t_z > t$, from the set of ready tasks $\mathbf{F}(t)$ with n elements, let

t が 0 以上の任意の時間であり，デッドラインt_zが t より大きい時，n 個のエレメントを持つ実行可能状態のタスク F(t)の集合での，任意のタスク T は F(t)に含まれる．そのことに対し，以下が成り立つ．

$a(t) = t_z - t$ be its response time,

$t_z - t$ は応答時間となる

$l(t)$ = the (residual) execution time required before completion, and

終了する前に要求される(残りの)実行時間

$s(t) = a(t) - l(t)$ its laxity (slack-time, margin, time reserve).

そのゆるみ(スラックタイム，マージン，時間の制限)

Then, necessary and sufficient conditions that the task set $F(t)$, indexed according to increasing response times of its n elements, can be carried through meeting all deadlines for single processor systems, are

n 個の要素の応答時間を増加させることに応じての指標とされる，タスクセット F(t)の必要十分条件は，シングルプロセッサシステムに対するすべてのデッドラインを達成することを完遂することが出来る．

$$a_k \geq \sum_{i=1}^{k} l_i \, , k = 1, \cdots, n \qquad (2.3)$$

This necessary and sufficient condition determines whether a task set given at a certain instant can be executed within its specified deadlines.

この必要十分条件は，

特定の瞬間に与えられたタスクセットが，指定されたデッドラインに内に実行されることが可能かどうかを判定する．

In words, it reads

言い換えると，それは以下のように理解できる．

If it holds for each task that its response time (the time until its due date) is greater then, or equal to, the sum of its (residual) execution time and the execution times of all tasks scheduled to be run before it, the schedule is feasible.

それが，タスクの応答時間(予定期日までの時間)が大きい，または同じということを，タスク

毎に保持している場合，
(残りの)実行時間とすべてのタスクの実行時間の合計は，それの前に実行されるためにスケジュールされ，そのスケジュールは実行可能である．

In the ideal case, the earliest deadline first method guarantees one-at-a-time scheduling of tasks, modulo new task arrivals.
理想的なケースでは，EDF 方式は，新たなタスクの到着を法とするタスクのスケジューリングを一つずつ保証する．

Thus, unproductive context-switches are eliminated.
従って，非生産的な実行するプログラムの切り替えは，排除される．

Furthermore, and even more importantly, resource access conflicts and many concurrency problems among the competing tasks, such as deadlocks, do not occur and, hence, do not need to be handled.
また，さらに重要なことは，
デッドロックのような，リソースへのアクセスの競合や，競合するタスク間の多くの同時実行性の問題は，発生しない．従って，制御する必要がない．

Unfortunately, such an ideal case is by its very nature unrealistic.
残念ながら，そのような理想的なケースは，
まさにその性質によって現実的ではない．

In the following, more realistic conditions will be considered.
以下では，より現実的な条件が考慮するつもりである．

--

Earliest Deadline First Scheduling under Resource Constraints and Feasible Schedulability Conditions
リソース制約と実行可能なスケジュール可能性条件のもとでの EDF スケジューリング

The above considerations have revealed that the earliest deadline first algorithm is the best choice for use in a general multitasking environment, provided that the tasks are pre-emptable at any arbitrary point in time.
上記の考慮事項は，EDF アルゴリズムが，タスクが任意のいずれかの時点で割り込まれる，
一般的なマルチタスク環境において利用するために最適な選択であることを明らかにし，

Unfortunately, this pre-condition is not very realistic, since it is only fulfilled by pure computing tasks fully independent of one another.
残念ながら，お互いに完全に独立した純粋なコンピューティングタスクによってのみ実行されるため，
この事前条件は非常に現実的ではない．

In general, however, tasks have resource requirements and, therefore, execute critical regions to lock peripherals and other resources for exclusive and uninterrupted access.
しかしながら，一般的に，タスクはリソース要求を持つ．
従って，タスクは，排他的で中断されないアクセスのための他のリソースと，周辺機器を固定するための重要な領域を実行する．

Hence, the elaboration of a task consists of phases of unrestricted pre-emptability alternating with critical regions.
従って，タスクの精巧さは，重要な領域を交互に入れ替える制限のない割り込みのフェーズから成る．

While a task may be pre-empted in a critical region, the pre-emption may cause a number of problems and additional overheads, such as the possibility of deadlocks or the necessity for a further context-switch, when the pre-empting task tries to gain access to a locked resource.
タスクが，重要な領域において割り込まれる可能性がある間，
割り込みタスクが，ロックされたリソースにアクセスしようとした時に，
割り込みが，さらなる実行するプログラムの切り替えのための，デッドロックの可能性や必要性のような，多くの問題と追加のオーバーヘッドを生じさせるかもしれない．

Therefore, to accommodate resource constraints, the earliest deadline first discipline is modified as follows:
従って，リソース制約に対応するために，EDF の原則は，次のように修正される．
すなわち，

Schedule tasks earliest deadline first, unless this calls for pre-emption of a task executing in a critical region. (In this case, wait until the tasks execution leaves the critical regions.)

スケジュールが，重要な領域において実行しているタスクの割り込みを必要としない限り，EDF でタスクを実行する.

Note that task precedence relations do not need to be addressed here, since the task set in consideration consists of ready tasks only, i.e., the set consists of tasks whose activation conditions are fulfilled (and, thus, their predecessors have already terminated).
ここで注意すべきことは，考慮事項におけるタスクのセットは，実行可能状態のタスクだけで構成するため，タスクの優先順位の関係が，ここでは対処する必要がないことである.
すなわち，タスクのセットは，活性化の条件を満たした(従って，先行していたものはすでに終了している)タスクから成る.

If all tasks competing for the allocation of the processor are pre-emptable, then the entire set is executed sequentially and earliest deadline first.
プロセッサの割り当てに関して競合しているすべてのタスクが割り込み可能であるとき，順次実行され，EDF となる.

Therefore, the (partial) non-pre-emptability of certain tasks can only cause a problem if a newly arrived task's deadline is shorter than that of the task running, which is at that same time executing in a critical region.
新しく到着したタスクのデッドラインが，重要な領域における同じ時間で実行する実行中のタスクのデッドラインよりも短い場合，特定のタスクの(一部の)割り込み可能ではないことが，問題だけを引き起こす可能性がある.

Hence, all tasks with deadlines closer than that of the executing task, including the newly arrived one, have to wait for some time $\boldsymbol{d}$, before the running task can be pre-empted.
従って，新たに到着したタスクを含んでいる，実行中のタスクのデッドラインよりも近いデッドラインのすべてのタスクは，実行中のタスクが割り込まれる前に，いくつかの時間 d を待つ必要がある.

Practically it is not feasible to determine the exact value of $\boldsymbol{d}$.
実質的に，d の正確な値を決定することは，不可能である.

The most appropriate upper bound for $\boldsymbol{d}$ is given by the maximum of the lengths of all non-pre-emptive regions of all tasks in a certain application, since this is the most general expression fully independent on the actual time and the amount of resource contention.
これ(最適な上限)は，実時間において完全に独立した最も一般的な式と，リソース競合の量であるため，

d に対する最適な上限は，特定のアプリケーションにおいて，すべてのタスクのすべての割り込まれない領域の長さの最大値によって与えられる.

With this, a sufficient condition, that allows one to determine a task set's feasible schedulability at any arbitrary time instant under the above algorithm, reads as follows:
これで，上記のアルゴリズムの下で任意の時間におけるタスクセットの実行可能なスケジュール可能性をこの一つ(？) に決定することを許可する十分条件は，次のように導く.

If a newly arrived task has an earlier deadline than the executing one, which is just within a critical region, the schedule according to the above algorithm is feasible if:
新たに到着したタスクが，ちょうど重要な領域の中である実行中のタスクよりも早いデッドラインを持つ場合，
上記のアルゴリズムによるスケジュールは，以下の場合実行可能である.

(a) all tasks, which should precede the running one (T_j) according to the earliest deadline first policy, have their response times greater than, or equal to, the sums of
最も早いデッドラインの最初の方針に従って，実行中のタスク(Tj)より重要な位置にあるべきすべてのタスクは，(1)～(3)の合計の応用時間よりも大きいか，等しいものを持つ.

(1) their execution times,
(1)それらの実行時間

(2) those of all tasks in the schedule before them, and
(2)それらの前のスケジュールにおけるすべてのタスク

(3) d, the time required to leave the critical region,
(3)重要な領域に到着するために要求される時間 d

$$a_i(t) \geq d + \sum_{k=1}^{i} l_k(t)\,, i = 1, \cdots, j-1 \qquad (2.4)$$

and

$$a_i(t) \geq \sum_{k=1}^{i} l_k(t), i = j, \cdots, n \qquad (2.5)$$

(b) all other tasks have their response times greater than, or equal to, the sum of their (residual) execution times and the execution times of all tasks scheduled to be run before them:

その他のすべてのタスクは，応答時間より大きいか，等しいものを持つ．
それらの(残りの)実行時間と，すべてのタスクの実行時間の合計は，それらの前に実行されるべきであることをスケジュールされる．

We observe that, if there are no critical regions, this condition reduces to the one holding for feasible schedulability under no resource conditions.
我々は
重要な領域が存在しない場合，この条件は，リソースの条件が存在しない中で実行可能なスケジュール可能性の状態を保つひとつに変換する
ことを観察する．

When running tasks may not be pre-empted at all, a newly arrived task with an earlier deadline has to wait until the running task terminates.
実行中のタスクが全く割り込まれない可能性がある時，
最も早いデッドラインで新たに到着したタスクは，実行中のタスクが終了するまで待つ必要がある．

A sufficient condition for feasibly scheduling non-pre-emptive tasks follows as an easy corollary from the result mentioned above.
割り込みされないタスクを実行できるようにスケジューリングするための十分条件は，
上記で言及された結果から簡単に当然の結果として従う．

This modified earliest deadline first policy offers a practical way to schedule tasks predictably in the presence of resource constraints.
この変更された最も早いデッドラインの最初の方針は，
リソース制約の存在の下で，予想通りにタスクをスケジュールするための実用的な方法を
を提供する．

The policy maintains most of the advantageous properties as listed in Section 2.2.4.
その方針は，2.2.4 章で記載されているような最も有利な特性を保持する.

In fact, the only property no longer exhibited is the attainability of maximum processor utilization because of the pessimistic prediction of the delay in pre-empting the executing task.
実際に，実行中のタスクに割り込むことにおける遅延の悲観的な予測のために，
もはや示されなかった唯一のプロパティは，最大プロセッサ使用率の到達可能性である.

From the point of view of classical computer performance theory, this may be considered a serious drawback.
古典的なコンピュータのパフォーマンス理論の視点から，これは，重大な欠点を考慮する必要がある.

For embedded real-time systems, however, it is not so relevant whether processor utilization is optimum, as costs have to be seen in the framework of the controlled external process, and with regard to the latter's safety requirements.
しかしながら，組み込みリアルタイムシステムに対し，
プロセッサ使用率が最適かどうかは関係がない.
コストが，制御された内部プロセッサの枠組みの中と，後者の安全要求に対して見られるべきである，

Taking the costs of a technical process and the possible damage into account which a processor overload may cause, the cost of a processor is of less importance.
プロセッサの過負荷が起こりうる，技術的プロセスのコストと，考えられる損傷を考慮することは，
プロセッサのコストがあまり重要でないということだ.

Moreover, while industrial production cost in general increases with time, the cost of computer hardware decreases.
さらに，一般的に時間と共に増加する工業生産のコストがかかる間，
コンピュータのハードウェアのコストは減少する.

Hence, processor utilization is not a suitable design criterion for embedded real-time systems.
従って，プロセッサの使用率は，組み込みリアルタイムシステムに対して，適した設計基準ではない.

Lower processor utilization is, thus, a small price to be paid for the simplicity and the other advantageous properties of the scheduling method presented here, which yields high dependability and predictability of system behavior.
従って，低いプロセッサ使用率は，
高い信頼性と，システムの振る舞いの予測可能性を生み出す
単純化のために支払われるべき小さな価値と，
ここで提示されたスケジューリング方法のその他の有利な特性がある．

Avoiding Context-Switches Without Violating Feasible Schedulability
実行可能なスケジュール可能性に違反することなく，実行するプログラムの切り替えを回避する

So far we have neglected the overhead costs due to context-switching.
これまでに我々は，実行するプログラムの切り替えに起因するオーバーヘッドのコストを無視している．

Now we demonstrate how some of the context-switches can be avoided, and how the cost of the remaining switches can be taken into account.
今，我々は，どのようにいくつかの実行するプログラムの切り替えが回避できるのか，
そして，どのように残りのスイッチのコストを考慮することが出来るか
を実証する．

Let us assume that the time required [to prepare a task's execution] and [to load its initial context into the processor registers] as well as [to remove the tasks from the processor] after the task has terminated normally is already included in the maximum execution time specified for the task.
タスクが正常に終了した後に，
タスクを実行する準備をすることを要求される時間と，
プロセッサのレジスタに最初のコンテキストをロードするために要求される時間と同様に，
プロセッサからタスクを削除するために要求される時間は，
すでにタスクに対して指定された最大実行時間に含まれている．

This assumption is realistic, since these context-changes have to be carried out under any circumstances and are not caused by pre-emptions.
これらのコンテキストの変更は，どのような状況の下でも実施されなければならず，割り込みによって引き起こされないため，
この仮定は現実的である．

Thus, we only have to account for the time required for a pre-emptive switch.
従って，我々は唯一割り込みスイッチに対して要求される時間を考慮する必要がある.

Without loss of generality, we further assume that this time is the same for all tasks, and denote it by ***u***, and that to either save or restore a task takes ***u/2***.
一般性を失うことなく,
我々は，さらに
この時間がすべての時間に対して同じであり，u によって表され,
そして，タスクを保存するか，復元するためには u/2 の時間がかかると仮定する.

The following modified form of the earliest deadline first task scheduling algorithm not only tasks resource constraints into account, but also avoids superfluous pre-emptions:
次の最も早いデッドラインの最初のタスクのスケジューリングアルゴリズムの形を変更したものは，
リソース制約を考慮するだけでなく，余分な割り込みを回避する.

Always assign the ready task with the earliest deadline to the processor, unless new tasks with earlier deadlines than the deadline of the currently running task arrive.
常に， 現在実行中のタスクの到着のデッドラインよりも，早いデッドラインを持つ新たなタスクが存在しない限り,
最も早いデッドラインで実行可能状態のタスクをプロセッサに割り振る.

If the laxities of the newly arrived tasks allow for feasible non-pre-emptive scheduling, then continue executing the running task.
新たに到着したタスクのゆるみが，実行可能な割り込みの内スケジューリングを可能にする場合,
実行中のタスクの実行を継続する.

Otherwise, pre-empt the task as soon as its critical region permits, and allocate the processor to the task with the earliest deadline.
それ以外の場合,
出来るだけ早く重要な領域のタスクの割り込みを許可し,
最も早いデッドラインでプロセッサにタスクを割り当てる.

Let us examine what occurs, when a pre-emption is required:

割り込みが要求されるときに発生するものを検討してみる.

If a newly arrived task has an earliest deadline than the executing one, which is just within a critical region, the schedule according to the above algorithm is feasible, if:

新たに到着したタスクが，ちょうど重要な領域の中である実行中のタスクよりも早いデッドラインを持つ場合，
上記のアルゴリズムによるスケジュールは，以下の場合実行可能である.

(a) all tasks, which should precede the running one according to the earliest deadline first policy, have their response times greater than, or equal to, the sums of

(a) 最も早いデッドラインの最初の方針に従って，実行中のタスクより重要な位置にあるべきすべてのタスクは，(1)～(4)の合計の応用時間よりも大きいか，等しいものを持つ.

(1) their execution times,

(1)それらの実行時間

(2) those of all tasks in the schedule before them,

(2)それらの前のスケジュールにおけるすべてのタスク

(3) the time d required to leave the critical region, and

(3)重要な領域に到着するために要求される時間 d

(4) a half of the switching time u/2.

(4)スイッチングの半分の時間 u/2

$$a_i(t) \geq d + \frac{u}{2} + \sum_{k=1}^{i} l_k(t)\,, i = 1, \cdots, j-1 \qquad (2.6)$$

and

(b) all other tasks have their response times greater than, or equal to, the sum of their (residual) execution times, the execution times of all tasks scheduled to be run before them, and the full switching time u:

その他のすべてのタスクは，応答時間より大きいか，等しいものを持つ.

それらの(残りの)実行時間と，それらの前に実行すべき，そして完全なスイッチング時間 u でスケジュールされたすべてのタスクの実行時間の合計

$$a_i(t) \geq u + \sum_{k=1}^{i} l_k(t)\,, i = j, \cdots, n \qquad (2.7)$$

Note that only the status-saving part of the context-switch needs to be considered in the feasibility check of tasks scheduled to be run before the pre-empted one, since they already contain their start-up time.
ここで注意すべきことは，それらはすでに起動時間を含んでいるため，
実行するプログラムを切り替えるステータスを保存する部分だけは，
割り込まれたタスクの前に実行されるべき，
スケジュールされたタスクの実行可能性のチェックにおいて，
考慮する必要があること
である.

For all other tasks, the total context-switching time u has to be taken into account.
他のすべてのタスクに対し，実行するプログラムの切り替えの合計時間 u が考慮されるべきである.

If there is no pre-emption, the feasibility check for non-pre-emptable scheduling applies.
割り込みが存在しない場合，
割り込まれないスケジューリングための実行可能性をチェックすることが適用される.