Operating Systems

プロセスとスレッドの基礎

2020-05

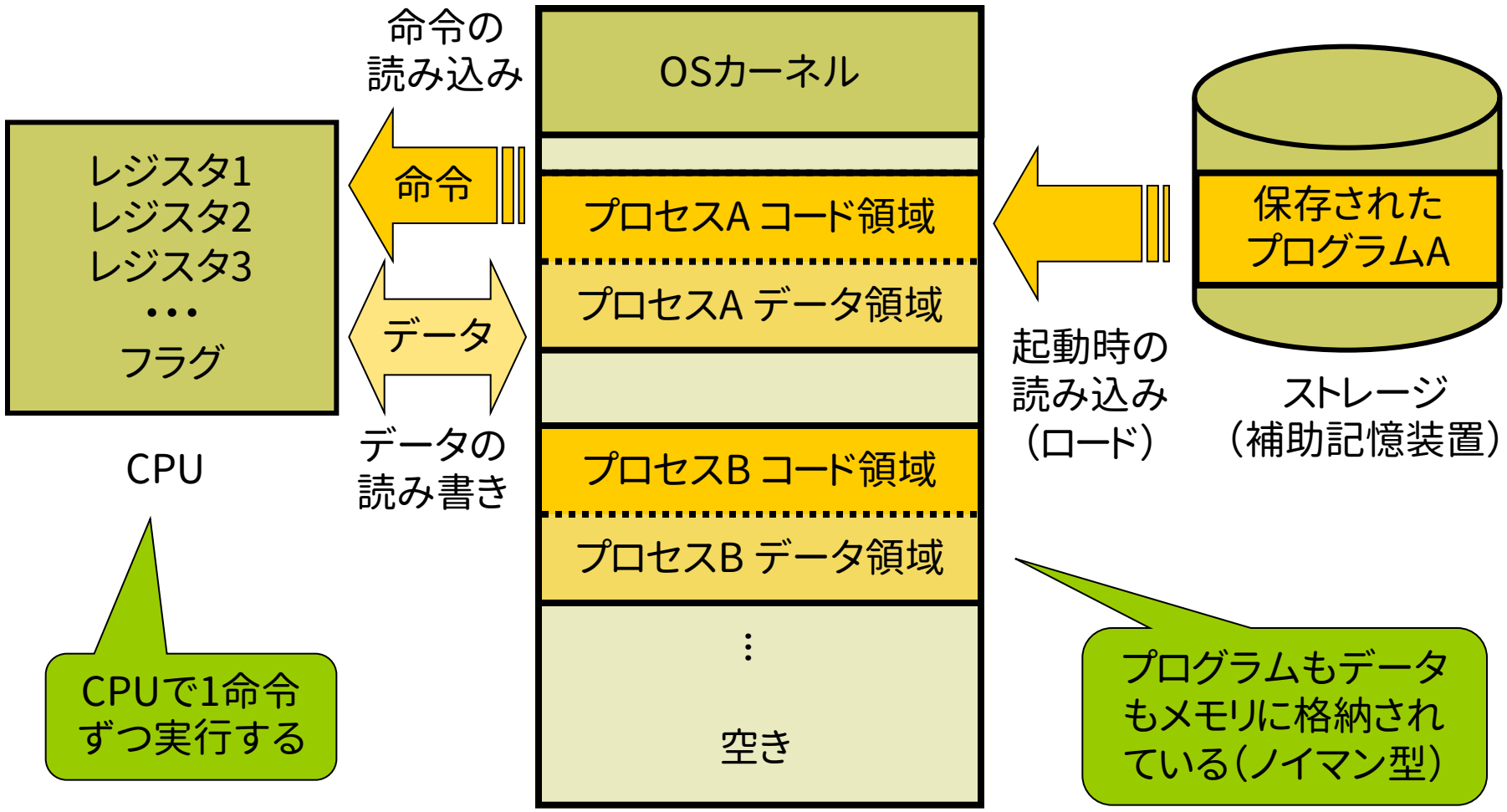
プロセス管理

- プロセス(process)とは
  - 起動して“実行中”のプログラム
  - コンピュータの中で“動いているもの”(CPUを使っているもの)
  - OSによっては,同じ意味で「タスク」(task)ともいう
- OSによるプロセスの管理
  - プロセスの生成(プログラムの開始とメモリ確保)
  - プロセスの消滅(プログラムの停止とメモリ解放)
  - プロセスの切り替え,優先順位の管理,などなど…
- プロセスの“正体”を,よりハードウェア的に考えると…
  - 「動いている」or「実行中」とは = CPUで演算処理をしている
  - 「プログラムとデータがある」とは = メモリの領域を占めている

例）Windowsのプロセス

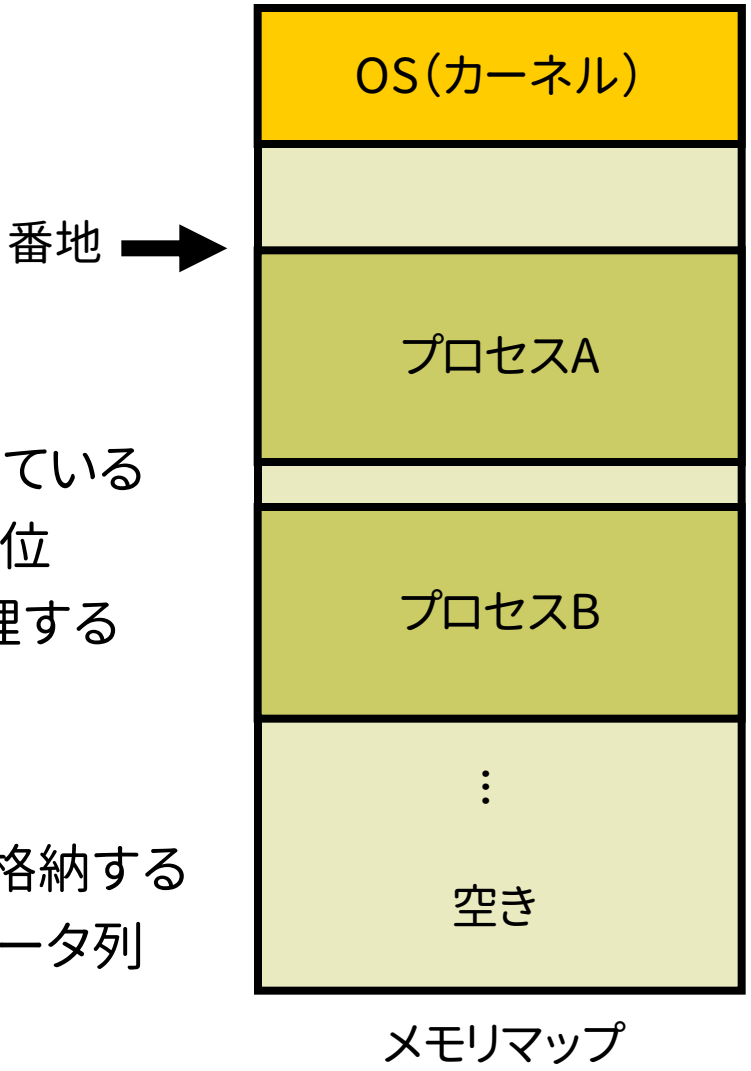
- タスクマネージャー
  - [Ctrl]+[Alt]+[Del] → [タスク マネージャー（の起動）]
  - Windowsのバージョンによって表示の切替方法が違う
- tasklist コマンド（コマンドプロンプト）
  - tasklist | more
  - tasklist /? | more
  - tasklist /v /FI "username eq ユーザ名"
- Sysinternalsのプログラム
  - https://docs.microsoft.com/en-us/sysinternals/downloads/
  - pslist ←tasklistとほぼ同じ
  - pskill ←プロセスの強制終了
  - pssuspend ←プロセスの一時停止

プログラムのロードと実行



メモリ(主記憶)のしくみ

- 半導体メモリ
  - RAM: 読み書き可能なメモリ
  - ROM: 読み出し専用メモリ
- アドレス(番地)
  - メモリの中は通し番号の番地がついている
  - 現在のPCでは,アドレスは1バイト単位
  - CPUは,格納アドレスでデータを管理する
- ノイマン型アーキテクチャ
  - プログラムもデータと同じ主記憶に格納する
  - プログラムの形式は,「機械語」のデータ列
  - 機械語の例: 10110000 01100001

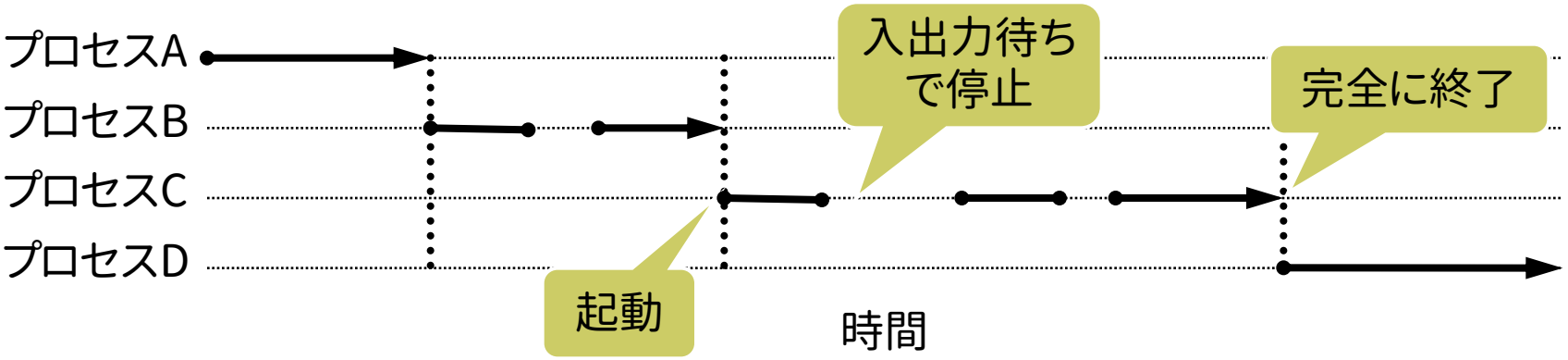


メモリマップ

シングルタスク(シングルプロセス)

- シングルタスクとは?
  - OSが実行状態を管理できるプロセスは,“1つ”だけ(シングル)
  - 前のプロセスが完全に終了しないと,次のプロセスを起動できない

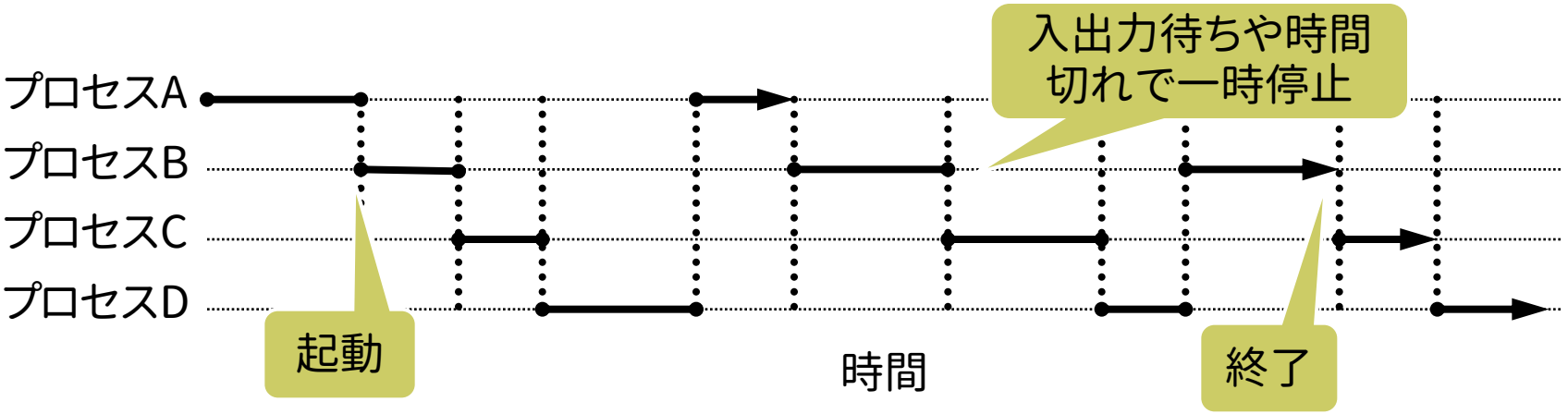
- シングルタスクの問題点
  - プロセスがディスク装置の読み書きなどで瞬間的に止まっている間,CPUは何もせずに待っているしかない　⇒ CPUの利用効率が悪い
  - 緊急性の高いプログラムがあっても,現在実行中のプロセスの完全終了を待つしかない ⇒ 柔軟な資源配分ができない



マルチタスク(マルチプロセス)

- マルチタスクとは?
  - 1つのCPUで複数のプログラムを(見かけ上)“同時に”実行できる
  - 前のプロセスが終了していなくても,新しいプロセスを起動できる

- マルチタスクの基本アイデア
  - あるプロセスがディスク装置の処理などで瞬間的に止まっている間,別のプロセスに空いているCPUを使わせる
  - さらに,実行プロセスを細かい時間で次々に切り替えて,同時並行に動作しているかのように見せかける



マルチタスクの利点

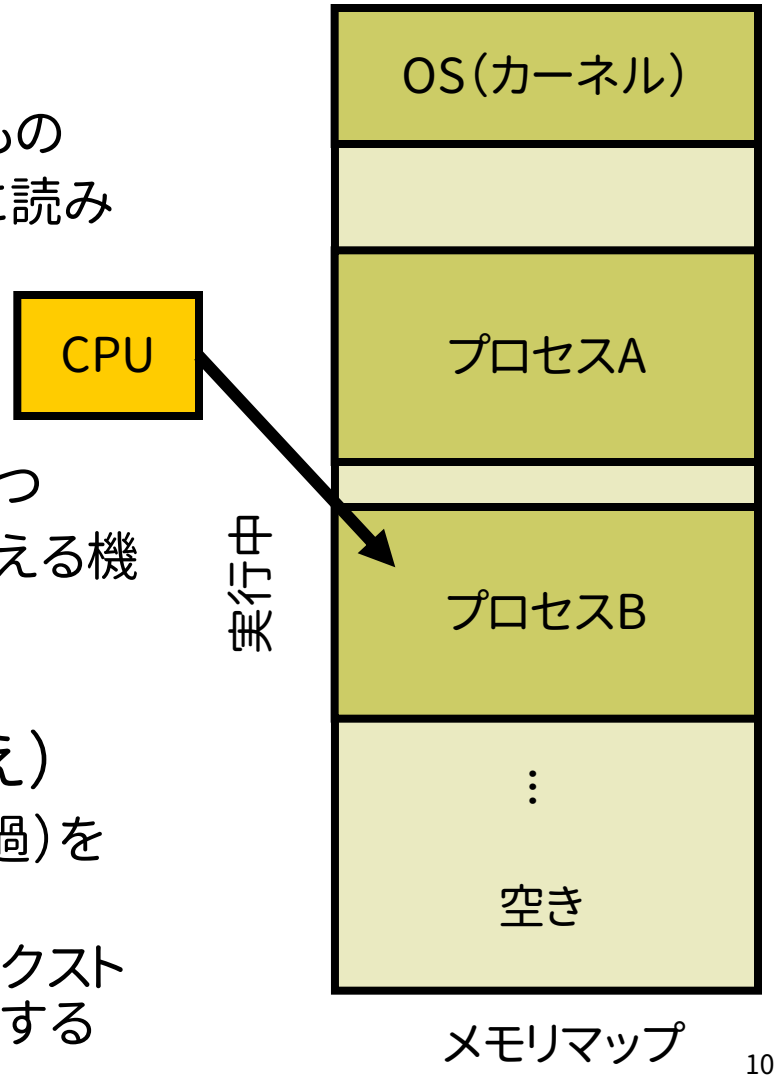
- 実行効率の向上
  - CPUの時間を，無駄なく有効に活用できる
  - OSカーネルやデバイスドライバとの並行的な実行も実現しやすい
- マルチユーザ機能
  - 複数のユーザがコンピュータを利用するようなサーバが実現できる
  - 1人での使用でも，同時に複数のソフトウェアを使えるようになる
- 多様な機能のプロセス
  - バックグラウンド（裏）で動作し続けるような処理が実現できる
  - あるプロセスを実行中に，それより優先度の高いプロセスが発生した場合，中断して対応できる
  - 指定時間に自動的に起動するプログラムなどが作りやすい

マルチタスクの種類

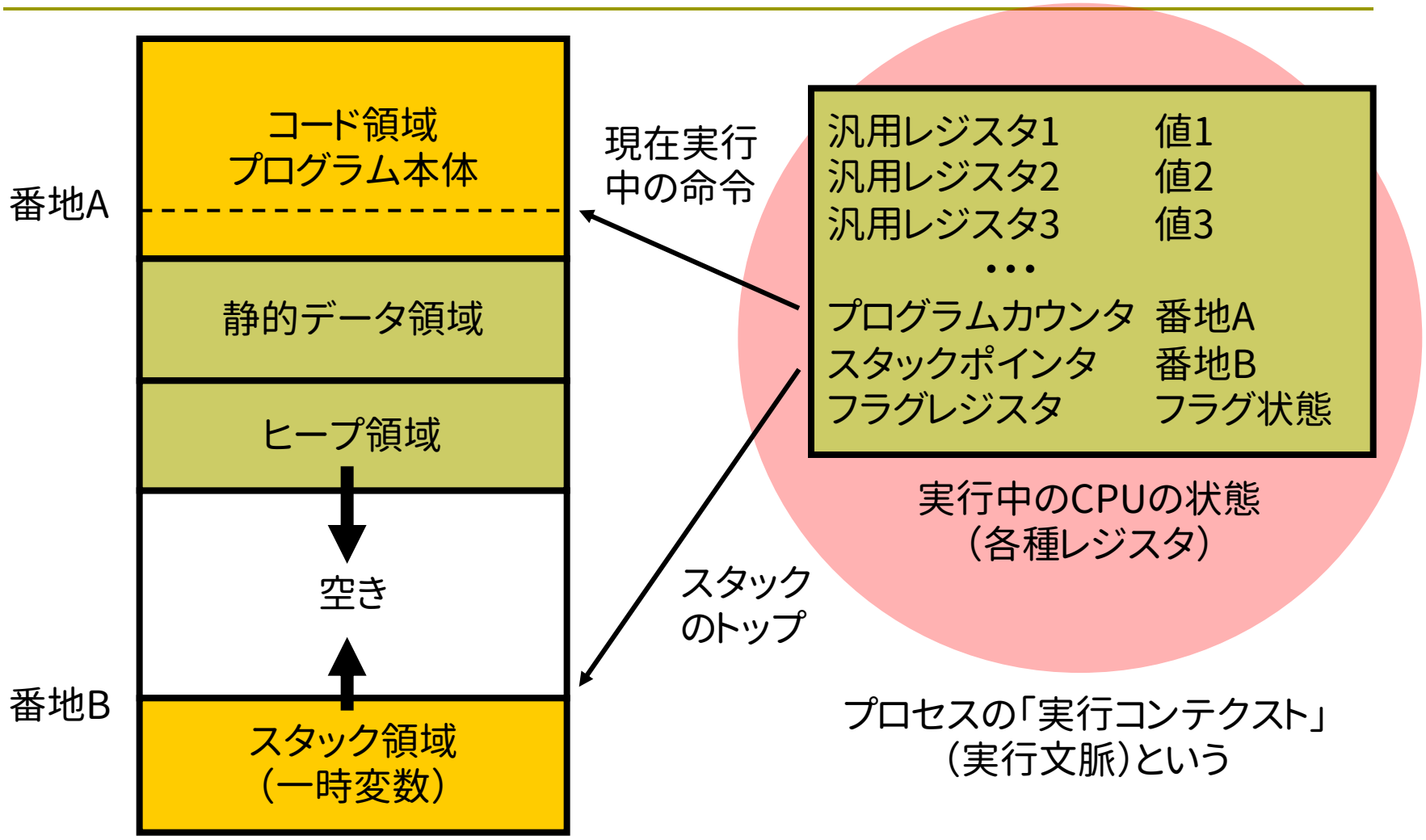
- ノンプリエンプティブなマルチタスク
  - 別名：協調型マルチタスク，疑似マルチタスク
  - OSはほとんどCPUの時間管理をしない
  - OSは実行中のプロセスからCPUを横取り（プリエンプション）できない
  - よって，各プロセスが自主的・明示的にCPUを次のプロセスに譲る
  - または，OSのAPI利用時に制御がOSに移り，プロセスが切り替わる
  - 例：16bit時代のWindowsやMac，組み込みOS等
- プリエンプティブなマルチタスク
  - 本来のマルチタスク
  - OSがタイマー割り込み等を利用してCPUの時間管理をする
  - OSは，適当なタイミングごとに，実行中のプロセスからCPUを横取り（プリエンプション）して，次のプロセスを切り替える
  - 例：低機能な組み込みシステムを除く，ほとんどのコンピュータ

マルチタスクの実現方法

- メモリはプロセスごとに分割できる
  - メモリは番地があって土地みたいなもの
  - 複数のプログラムを，主記憶に同時に読み込んでおくことは容易
- しかし，CPUは分割できない
  - 1つのCPUコアが実行できる命令は1つ
  - 実行プロセスを，高速に次々と切り替える機能が必要になる
- コンテクストスイッチ（文脈切り替え）
  - 実行プロセスのコンテクスト（途中経過）をカーネル内に一時退避し，
  - 退避してあった別のプロセスのコンテクストを読み出して中断したところから続行する



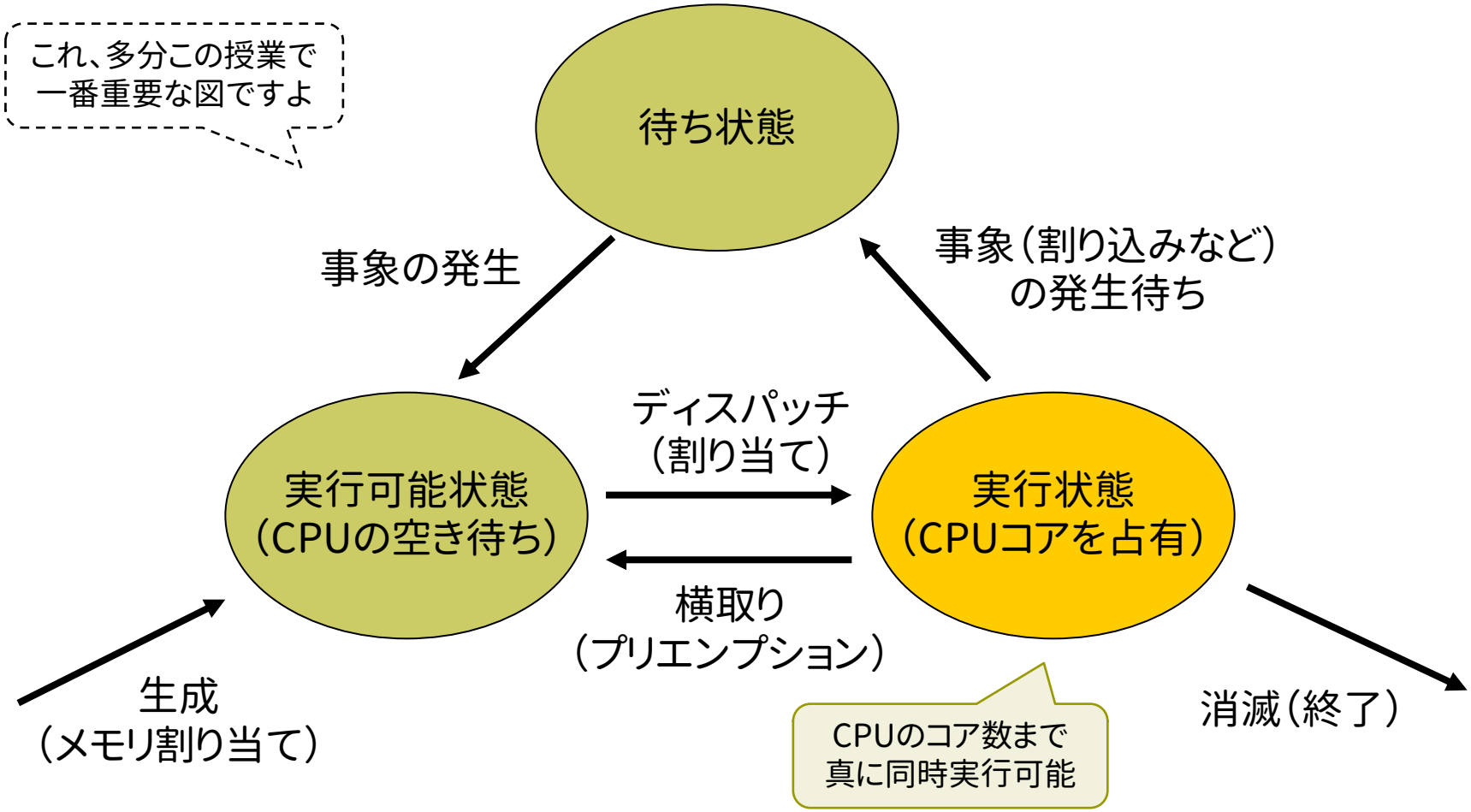
プロセスの実行状態



プロセス空間（プロセスに割り当てられたメモリ空間）

プロセスの状態遷移

- マルチタスクにおけるプロセス（スレッド）の一生



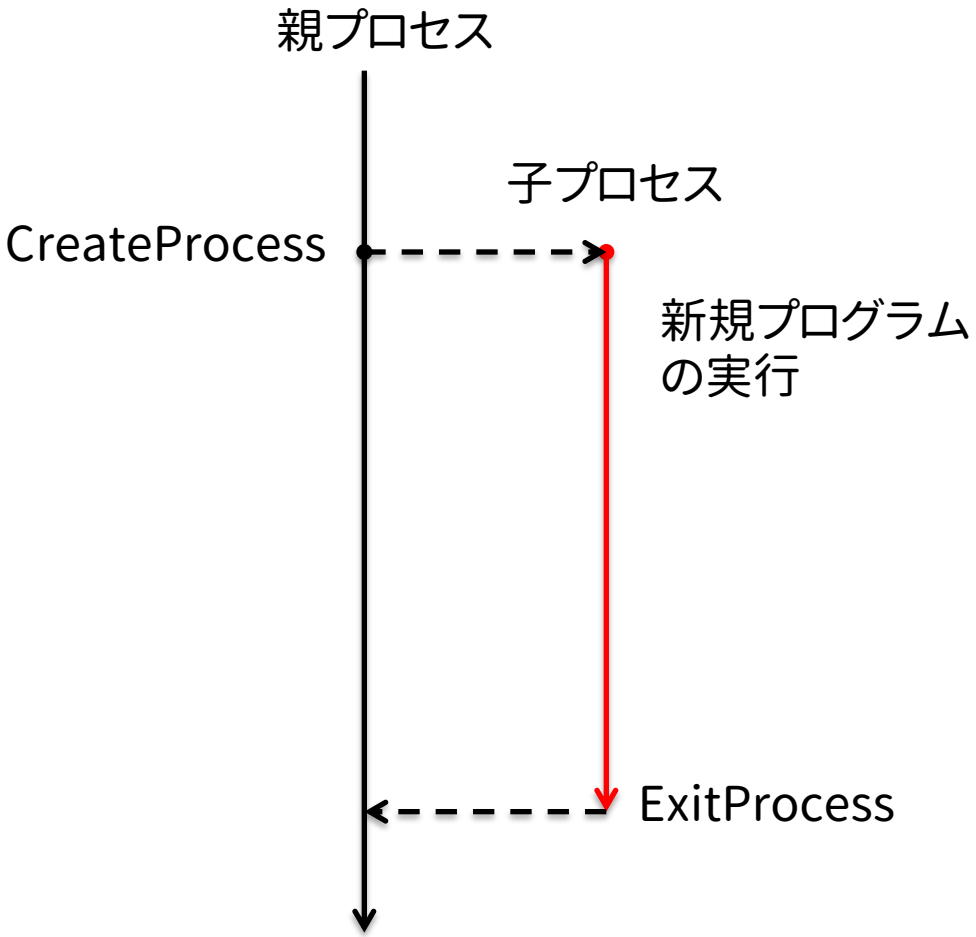
プロセス/スレッドの生成

- プロセスの親子関係
  - プロセスは，親のプロセスが生成する（プロセス生成用APIを使う）
  - 最初のプロセスは？⇒ OSカーネルが生成する
- マルチタスク環境では
  - 親プロセスが子プロセスを生成すると，両者は並行的に存在する
  - 親プロセスは，いくつもの子プロセスを起動して別々に処理を行わせ，必要なら終了報告を待つことができる
- UNIXのfork/exec/joinモデル
  - fork: 自分のプロセスの分身を作る（メモリや設定をまるまるコピー）
  - exec: プログラムをロードし，自分のプロセスの内容を入れ替える
  - join: forkされたプロセス同士が合流する（exit & wait）

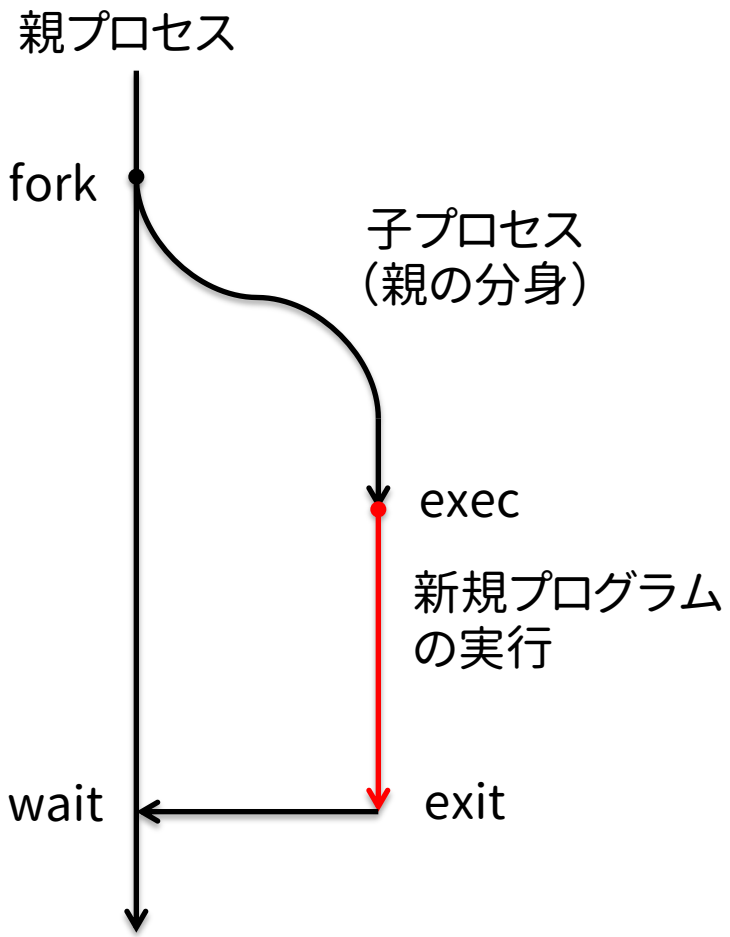
スレッド

- スレッド（thread）とは
  - 専用のメモリ空間を持たない“軽いプロセス”のこと
  - 独自の実行コンテクストを持つが，メモリは親のプロセスと共有する
  - 組み込みOSの場合，プロセスは実質的にスレッドの場合が多い
- スレッドの用途
  - 1つのプロセスの中に，さらに別々に“動くもの”を作ることができる
  - 複数の処理の並列実行によってプログラムを高速化する
  - メインの処理の“裏”でユーザやネットワークからの入力に対応する
- プロセスに比べた利点
  - 新規生成や切り替え（コンテクストスイッチ）の処理が速い
  - メモリ処理がないので，OSによる管理が単純化できる

プロセスの生成モデル



Windowsのプロセス生成モデル



UNIXのプロセス生成モデル

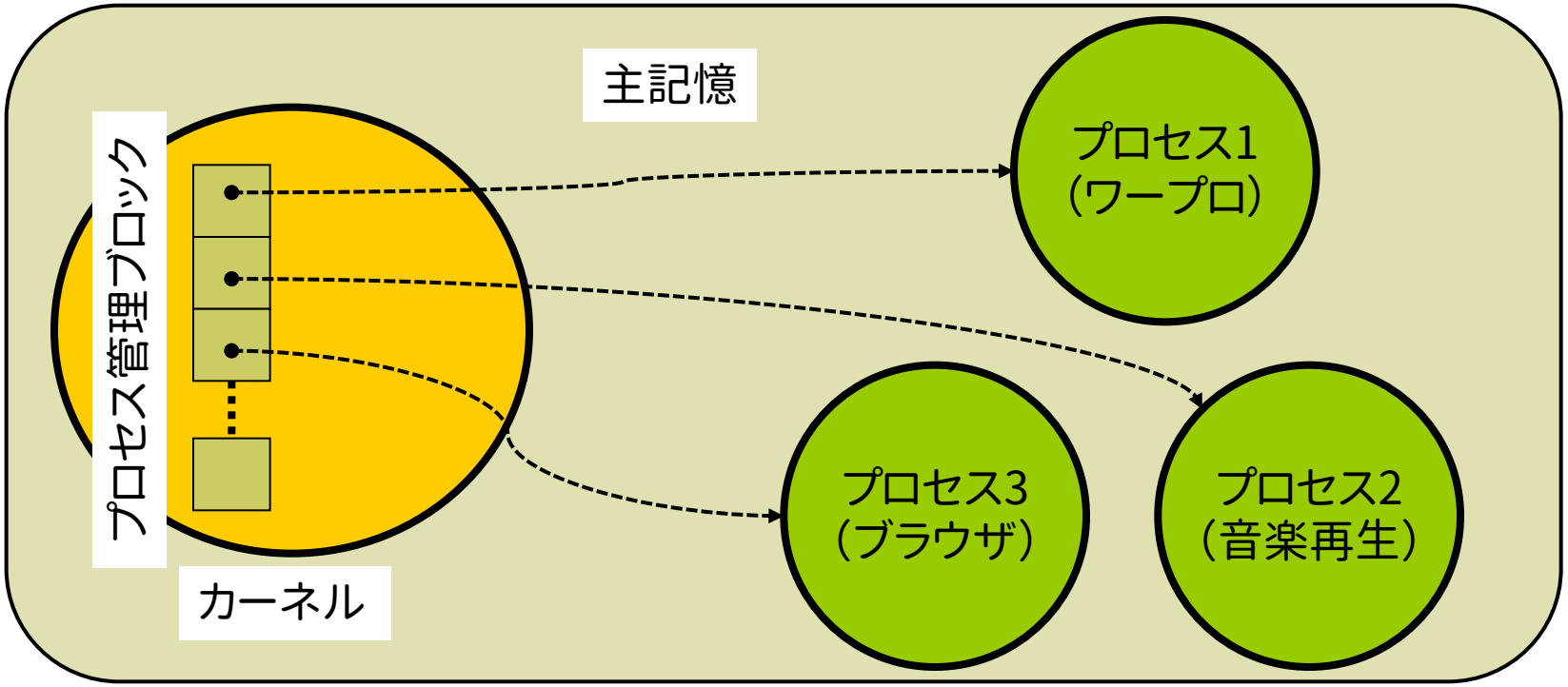
プロセスの生成・消滅処理

- プロセスの生成処理
  1. カーネル内に，新しい「プロセス管理ブロック」を割り当てる
  2. プロセスのためのメモリ（主記憶）の領域を確保する
  3. プログラムコードを読み込み，データ領域やスタック領域を設定する
  4. 実行開始時のレジスタ等の値を初期設定する
     - プログラムカウンタ，スタックポインタ，その他…
  5. 「実行可能状態」にして，OSに実行されるのを待つ

- プロセスの消滅処理
  1. OSが管理する実行可能なプロセスの一覧から削除する
  2. プロセスに割り当てられたメモリを解放する（未使用に戻す）
  3. プロセスの終了コード（正常終了かエラーか）を保持しておく
  4. カーネル内の「プロセス管理ブロック」を削除する

プロセス情報の管理

- カーネルでのプロセス管理
  - カーネルに，プロセス管理ブロック（プロセステーブル）という表がある
  - 各プロセスの実行情報を格納して，カーネルが管理している
  - プロセスの「実行コンテクスト」もここに退避される



演習課題

- 課題 5a HOSにおけるマルチタスクプログラミング
  - この課題の狙いは，マルチタスクの基本を理解することである。
  - マルチタスク処理が可能なOSでは，実行中のタスクを一時中断して，別のタスクに実行を切り替えることができる。
  - HOSでは，実行中のタスクが自ら休止等の処理をすることによって，処理を一時中断して他のタスクに実行を切り替えることができる。
  - そのために，OSはタスクの実行コンテクストを退避・復帰する。

- 手順
  - フォルダ hos-v4¥sample¥win-multitask の内容を確認する。
  - sample.sln を開いてコンパイル・実行する。
  - 出力結果を示し，どうしてそうなるのか考察（理由を説明）せよ。
  - 特に，task1とtask2の処理のタイミングについて考察せよ。
  - さらに，dly_tskの引数（遅延時間）を変更してみるとよい。

演習課題

- 課題 5b HOSにおけるノンプリエンプティブなマルチタスク
  - この課題の狙いは，ノンプリエンプティブな（協調的な）マルチタスクの動作とそのためのプログラミング作法を理解することである。
  - プリエンプティブでないOSでは，プロセス（タスク）が適当なタイミングで自主的に実行権を手放すことで円滑なマルチタスクを実現する。
  - HOSの場合，レディキューを回して次のタスクを実行する rot_rdq（rotate ready queue）というサービスコール（API）を利用する。

- 手順
  - sample¥win-scheduling をフォルダごとコピーして利用する。
  - ソースコードでコメントアウトされている rot_rdq 関数を有効にすると，そのタスクは自主的に実行権を次のタスクに譲るようになる。
  - 一部および全部のタスクの rot_rdq 関数を有効にして実行した結果を示し，それらを比較した上でそのような動作する理由を考察せよ。

HOS（μITRON）のタスク管理

サービスコール	意味	説明
cre_tsk	create task	タスクの生成
acre_tsk		同上（ID自動割り当て）
act_tsk	activate task	タスクの起動
iact_tsk		同上（割り込み用）
can_act	cancel activation	タスク起動予約の取り消し
ext_tsk	exit task	自タスクの終了
ter_tsk	terminate task	タスクの強制終了
chg_pri	change priority	タスク優先度の変更
get_pri	get priority	タスク優先度の参照
rot_rdq	rotate ready queue	実行可能キューの回転
irot_rdq		同上（割り込み用）

HOS（μITRON）のタスク管理

サービスコール	意味	説明
dly_tsk	delay task	自タスクの遅延（指定時間休止）
slp_tsk	sleep task	自タスクの休止（起床要求待ち）
tslp_tsk		同上（タイムアウトあり）
wup_tsk	wake up task	タスク起床要求
iwup_tsk		同上（割り込み用）
can_wup	cancel wakeup	タスク起床要求の取り消し
rel_wai	release wait	待ち状態の強制解除
irel_wai		同上（割り込み用）
sus_tsk	suspend task	強制待ち状態への以降
rsm_tsk	resume task	強制待ち状態からの再開
frsm_tsk	force resume task	強制待ち状態からの強制再開