Operating System 3

岩井将行

2014

連絡先

- 岩井
  - iwai　atmark　im.dendai.ac.jp
- 副手
  - 重田くん
  - 実空間コンピューティング研究室
  - 東京電機大学大学院 未来科学研究科 情報メディア学専攻
  - shige　atmark　cps.im.dendai.ac.jp

第1回

- ハードウエアとOS
- CPUとデバイス、割り込み、記憶装置、ハードディスク装置、RAID、パリティ
- スレッドとプロセス、メモリアドレス空間、ファイルシステム、NIC、ソケット、カーネル

第2回

- CUI
- タイピング，ログイン，コマンド操作，マニュアル，シェル
- ファイル操作，エディタ

第3回

- プロセス，ジョブ
- プロセス管理、時分割処理とプロセス切り替え、スケジューリング、プロセス表

第4回

- 記憶装置
- メモリ階層、キャッシュ、アドレス空間、物理アドレスと論理アドレス、ページング、チェックポインティン
- グ、効率的な自動メモリ管理、GC、フラッシュメモリ、HDD、SSD

第5回

- シェルとアクセス権
- 標準入出力、フィルタコマンド、シェルスクリプト、ファイルのバックアップ、アクセス権、ドライブ、ディレク
- トリ、ファイル、open/read/write
- ファイルのメモリへのマッピング、アクセス制御、権限、空き領域管理

第6回

- ネットワークとOS
- ethernet,ping,socket,tcp/udp,rpc,apach,http,ssh,ftp,remotewindow
- 最新のOS事情

第7回

- 試験

前回の復習

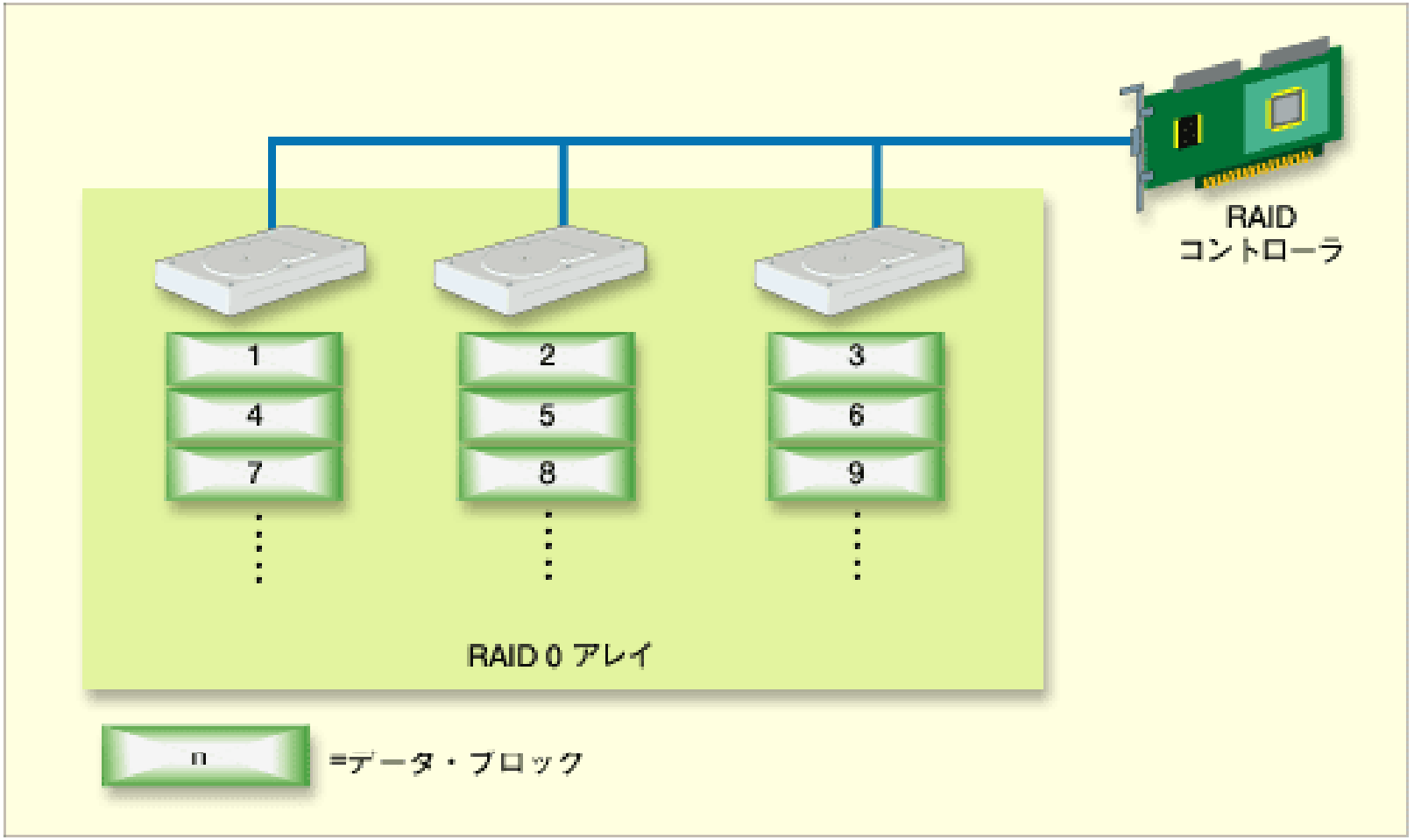
Raid

Redundant Arrays of Independent Disks

– RAID Advisory Board での定義

- レベル0　ストライピング
- レベル1　ミラーリング
- レベル2　分散ハミングコード
- レベル3　バイト分割固定パリティ
- レベル4　ブロック分割固定パリティ
- レベル5　分散パリティ
- レベル6　2重分散パリティ

Raid0



Raid0 ストライピング

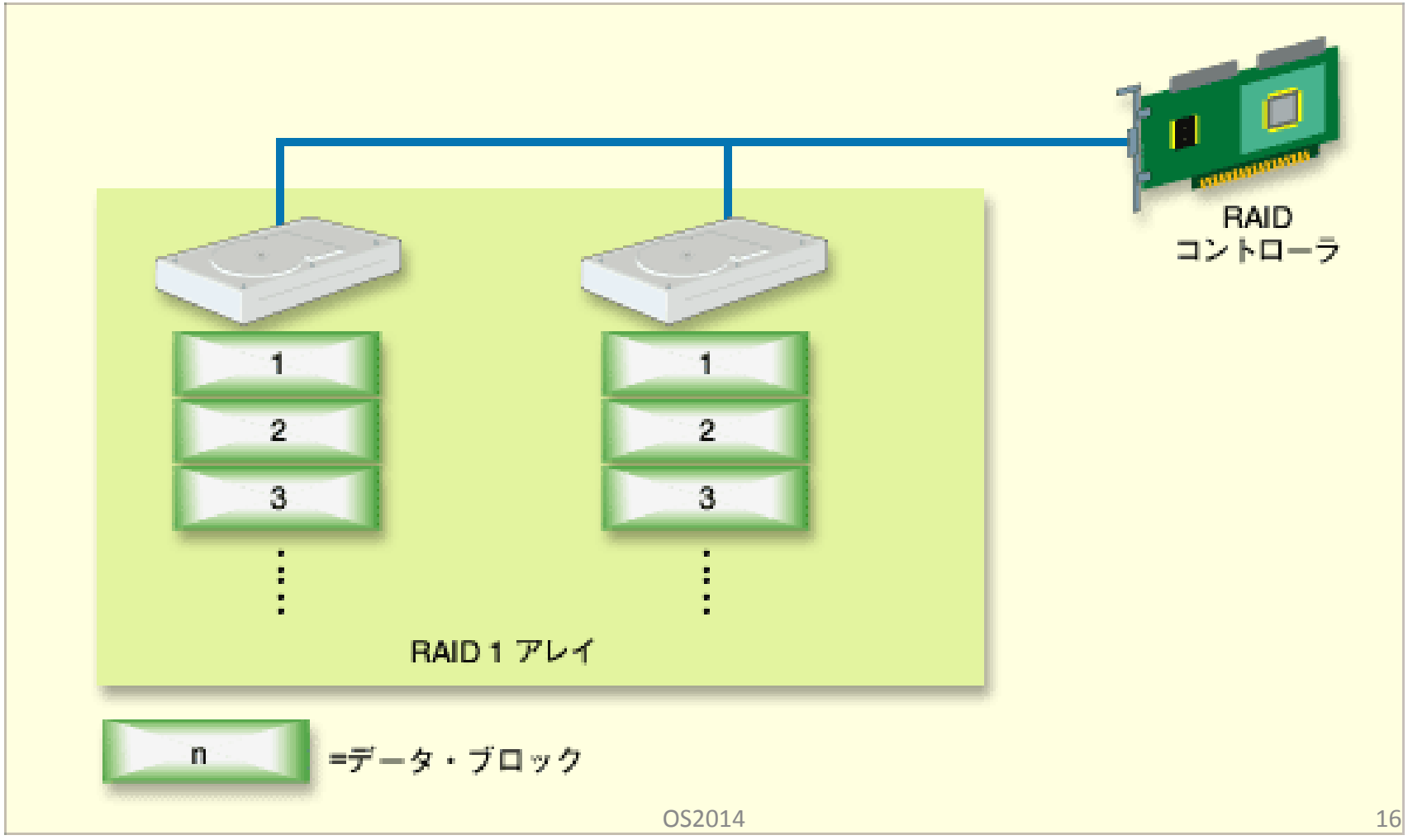
- データ・ブロック1／2／3や4／5／6、7／8／9は、もともと連続しているデータである。
- RAID 0では、ディスクの台数に合わせてデータを分割して、各ディスクに格納する。
- 例えばデータ・ブロック1／2／3の組を読み出す場合、各ディスクに並行してアクセスすることで、ほぼ同時に1／2／3それぞれのデータ・ブロックを読み出すことが可能。

- 高速化技術

Raid1　ミラーリング

- RAID 1は、RAIDレベルの中で、最も単純な手法でディスクの耐障害性を高めている。その手法とは、同一のデータを複数のディスクに書き込み、一方のディスクが故障しても、他方で処理を続行できるようにする。

Raid1ミラーリング

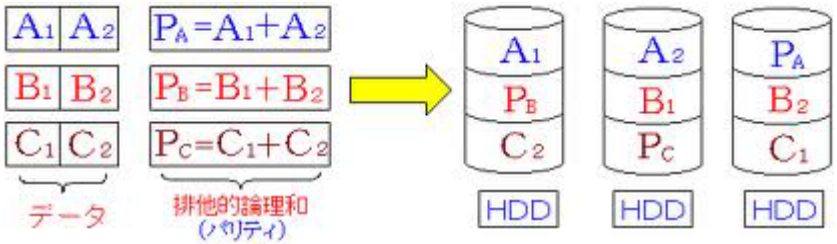


Raid1 稼働率

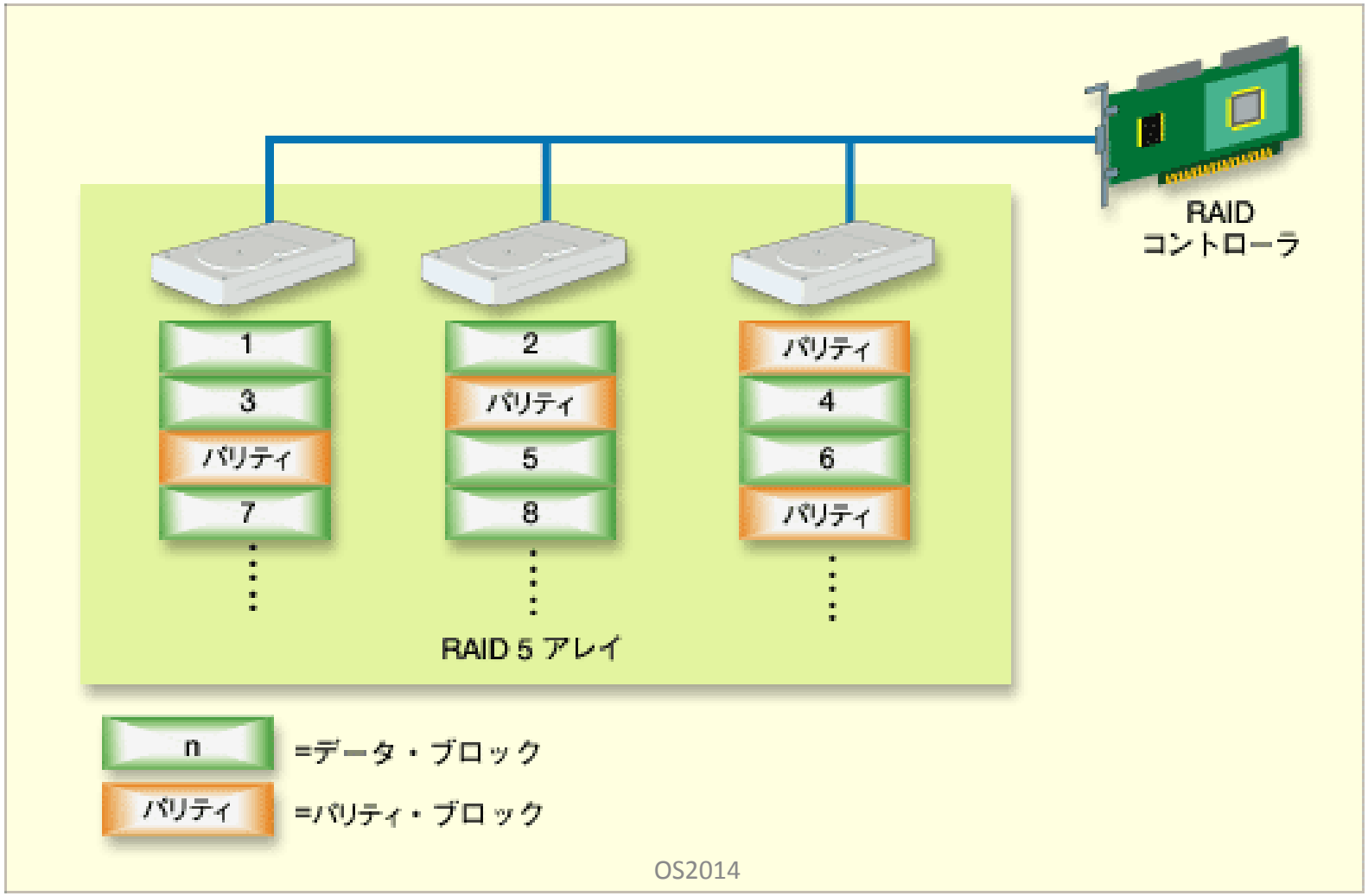
- RAID 1では、同一のデータを2台以上のディスクに書き込むため、ディスク容量の利用効率は50%以下になってしまうというデメリットがある（2台のディスクの 容量が異なると、利用効率は50%よりさらに下がる）。
- 例えば1Tbytesのデータを記録するには、1Tbytes×2＝2Tbytes分の容量のディスクが必要になる。

Raid5

- RAID 5は、耐障害性の向上と高速化、大容量化のすべてを実現できるRAID技術。
- 分散データ･ガーディングとも呼ばれる。
- RAID 5では、ディスクの故障時に記録データを修復するために「パリティ」と呼ばれる冗長コードを、全ディスクに分散して保存するの



Raid5


RAID
コントローラ
1
3
パリティ
7
2
パリティ
5
8
パリティ
4
6
パリティ
RAID 5 アレイ
n
=データ・ブロック
パリティ
=パリティ・ブロック

RAID 5の動作原理

- データを分割して各ディスクに格納するという原理はRAID 0（ストライピング）と同じだ。異なるのは、データ・ブロックの組（上図でいえば1／2や3／4、5／6）ごとにパリティが生成される点である。たとえ1 台のディスクが壊れても、残りのディスクに格納されたデータとパリティから、失われたデータを復活させることができる。

Raid5の利用効率

- パリティの保存に必要なのは、全ディスク台数に関係なくディスク1台分の容量である。
  従ってディスク台数が多いほど容量の利用効率も向上する。RAID 1（ミラーリング）と比較した場合、この利用効率の高さがRAID 5のメリットの1つとされる。

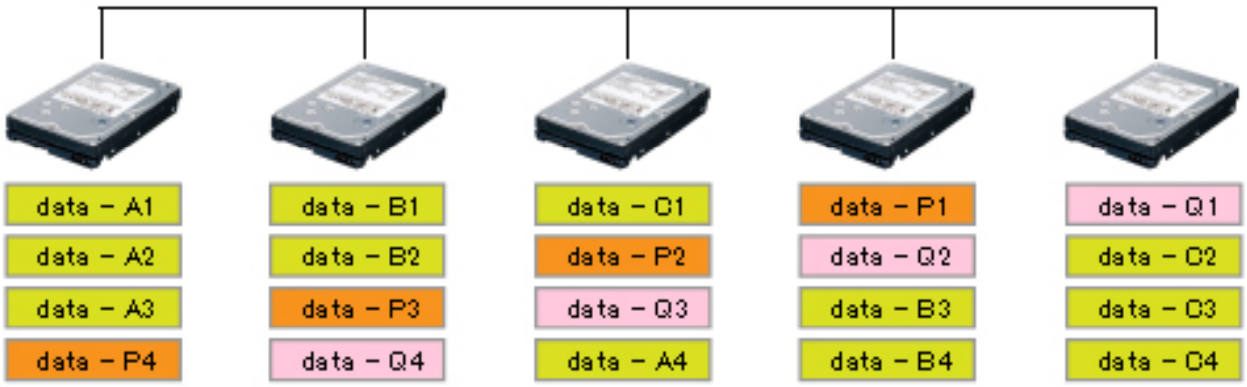
- ディスク容量の利用効率
  - 100 ×（n-1)/n%

Raid5追加

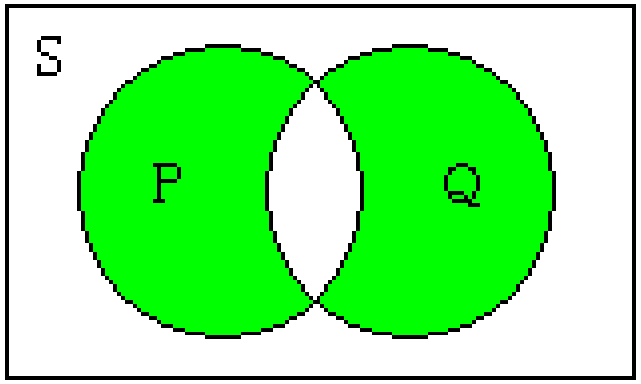
- 【ディスク】最低3台のディスク。
  ・ブロック単位に分割したデータを、生成した冗長コードと共に各ディスクに分散して保存
  ・データだけでなくパリティも複数のディスクに分散させることで、速度面でのボトルネックを回避
  ・冗長コードにパリティを使用するため、2台以上のドライブが同時に故障するとデータの回復ができない。

Raid6

- RAID6はRAID5の冗長性強化版です。冗長コード(パリティ)を2種類生成し、それぞれデータ本体と共に各ディスクに分散(ストライプ)して記録
- 【読み方】「レイドシックス/レイドロク」です。ダブルパリティRAID
  【冗長化機構】2つのパリティ(Parity)を用いて冗長性(Redundancy)を確保
  【ディスク】最低4台のディスクが必要
  ・ブロック単位に分割したデータを、生成した冗長コードと共に各ディスクに分散して保存。
  ・パリティを2種類用意することにより、同時に2台のディスクが故障しても回復。
  ・パリティーの保存に2台分の容量を使うため、ディスクの容量効率は落ちます。
  ・RAID5と同様、パリティを処理するため、書きこみ速度は遅い



パリティ計算に使う排他的論理和



命題 P	命題 Q	$P \veebar Q$
真	真	偽
真	偽	真
偽	真	真
偽	偽	偽

補足説明

- A,B,C,Dのディスクがあった時
- パリティを
- A1 xor B1 xor C1　=P1とおく
- P1をディスクDに保存する。
- ディスクBが壊れた場合
- P1 oxr (A1 xor C1)で復旧できる。

- K xor (R or K) =(K xor K) xor R = 0 xor R = R

- 論理記号

$$\neg(P \vee Q) = \neg P \wedge \neg Q$$

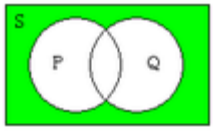
$$\neg(P \wedge Q) = \neg P \vee \neg Q$$

- プログラミングの書き方
  - !(P || Q) == !P && !Q
  - !(P && Q) == !P || !Q
- 集合

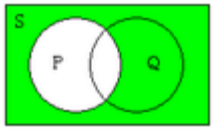
$$\overline{(A \cap B)} = \overline{A} \cup \overline{B}$$

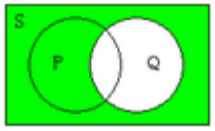
$$(A \cup B) = \overline{A} \cap \overline{B}$$

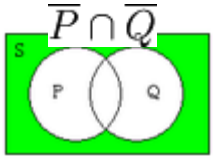
- ベン図

$\neg(P \vee Q)$ 

=

P 

Q 

$\overline{P} \cap \overline{Q}$ 

以下の文は同値か？

- 「私の身長は 160 cm 以上であり、かつ私の体重は 50 kg 以上」である の否定
- 「私の身長は 160 cm 以上であり、かつ私の体重は 50 kg 以上」ではない

- 「私の身長は 160 cm 未満であるか、または私の体重は 50 kg 未満」である

パス名（絶対・相対パス）

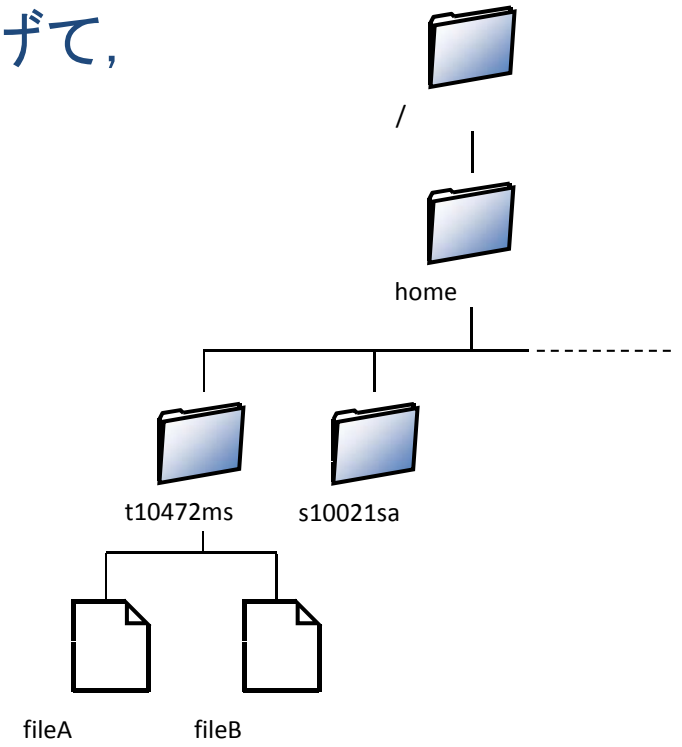
パス名（絶対パスと相対パス）

- ファイルやディレクトリにアクセスするために，ファイルやディレクトリの位置（パス名）を示す必要がある
- パス名の指定方法は以下の2種類

- 絶対パス
  - ルートディレクトリを基点として絶対的な位置を指定する
  - 例：住所は絶対パス「神奈川県藤沢市遠藤5322」
- 相対パス
  - あるディレクトリを基点にした相対的な位置を指定する
  - 場合によっては，絶対パスより短いパス名で指定できる
  - 例：田中君の家は「私の家の右隣」

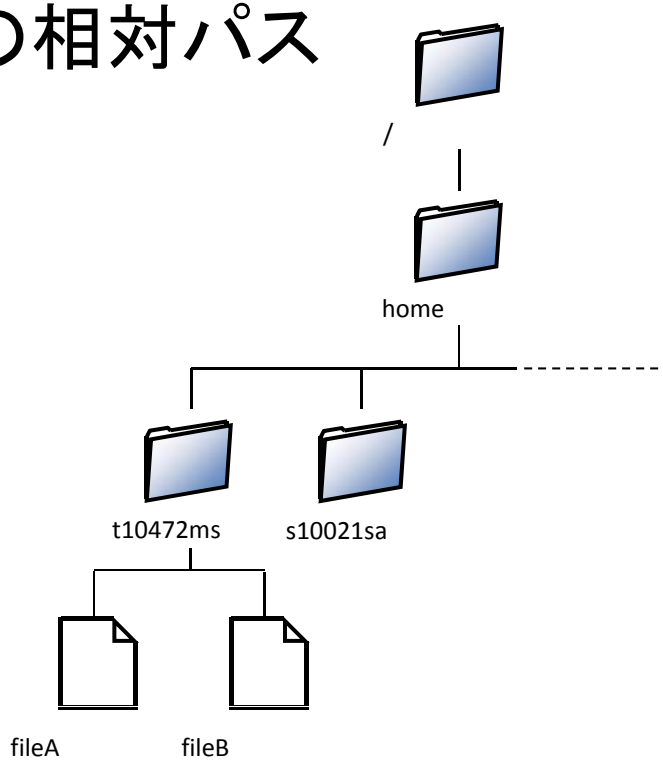
絶対パス

- fileAの絶対パス
  - 日本語だと“**ルートディレクトリの中の，homeディレクトリの中のt10472msディレクトリの中のfileA**”
  - 区切りを「/（スラッシュ）」で繋げて，“**/home/t10472ms/fileA**”



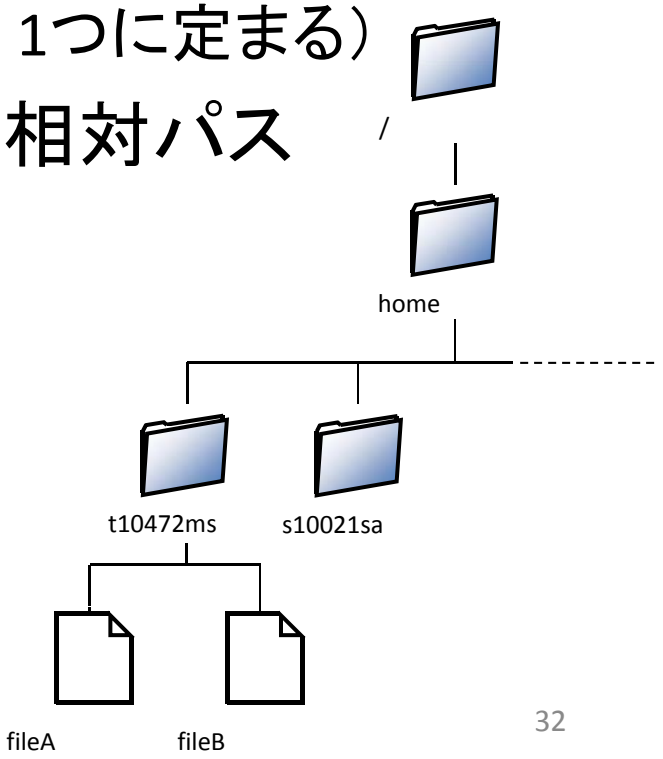
相対パス1

- t10472msを基点にした時のfileBの相対パス
  - “**fileB**”
- homeを基点にした時のfileBの相対パス
  - “**t10472ms/fileB**”



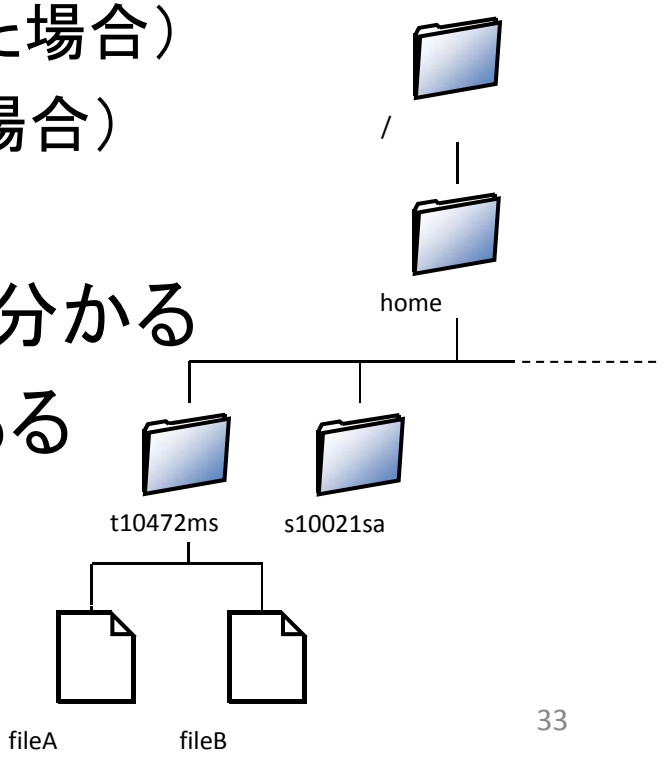
相対パス2

- 基点から見た親ディレクトリを指定するには，“..（ピリオド2つ）”の記号を使う
- t10472msを基点にした時のhomeの相対パス
  - “..”（親ディレクトリは1つだけなので，1つに定まる）
- s10021saを基点にした時のfileBの相対パス
  - **“../t10472ms/fileB”**



相対パス３

- 基点のディレクトリを示すには，“.（ピリオド1つ）”の記号を使う
- t10472msを基点にした時のfileAの相対パス
  - **“./fileA”**（基点ディレクトリを明示した場合）
  - **“fileA”**（基点ディレクトリを省略した場合）
- 基点ディレクトリを明示すると，相対パスによる指定であることが分かる
- パス名が読みやすくなる場合がある



ファイルとディレクトリの操作

GUIとCUI

- Graphical User Interface（GUI）
  - 画面表示にアイコンやメニューを用い，操作の大半をマウスなどのポインティングデバイスによって行なう
  - Finder（Mac）やExplorer（Windows）はGUIを備えたファイルマネージャ（ファイル管理機能をもつソフトウェア）
  - 直感的に操作ができる

- Character User Interface（CUI）
  - すべての操作をキーボードからコマンドと呼ばれる命令を用いて行なう
  - ターミナル（Mac）やコマンドプロンプト（Windows）を使うとCUIを使ってコンピュータを操作できる
  - 効率よく命令を記述でき，慣れれば素早く操作を行える

ターミナル

- コマンド操作を行うためには，ターミナルというプログラムを使う

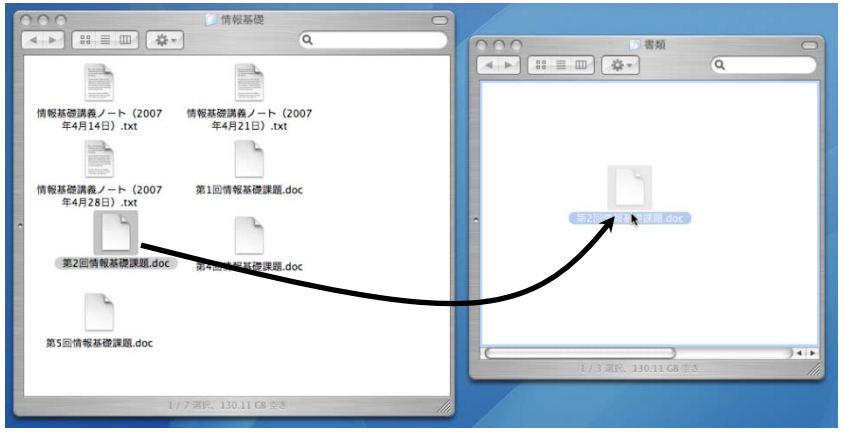
- 起動方法
  - 初期設定ではDockに登録されている
  - Dockにない場合
    - Finderを起動する
    - サイドバーのアプリケーションを選択する
    - ユーティリティフォルダ中の，ターミナルをクリック

コマンド

- コンピュータに与える命令のこと
- CUIのターミナルでは（マウスによるボタン操作ではなく），文字で命令を伝える

- 例：ファイルの移動

Finder（GUI）だとマウスで命令



ターミナル（CUI）だと文字で命令

```
% mv 第2回情報基礎課題.doc /Users/
```

プロンプト

- ターミナルを起動すると，%マークが現れる
- これをプロンプトと呼び，コンピュータがコマンドによる指示を待っている印（しるし）
- コマンドを入力したら，エンターキーを押すと命令が実行される

```
% コマンド↵
```

このスライドではエンターキーを押すタイミングを↵で表現しています

なぜコマンド操作を学習するか

- サーバはコマンドで操作することが多い
  - プロバイダから提供されているWebサーバを設定する
  - 所属する研究室・会社のサーバの管理をする

- 効率よくコンピュータに仕事を指示することができる
  - シェルスクリプト
  - ワイルドカード

ファイル・ディレクトリの操作の
コマンド一覧

- pwd → カレントディレクトリの絶対パスを表示
- ls → ディレクトリの内容を見る
- cd → カレントディレクトリの移動
- less → ファイルの内容を見る
- mkdir → 新しいディレクトリを作る
- cp → ファイルのコピーを作る
- mv → ファイルの移動・ファイル名の変更
- rm → ファイルの削除
- rmdir → ディレクトリの削除

カレントディレクトリ

- コマンドによってファイルやディレクトリを操作する場合，相対パスでファイルやディレクトリを指定する方が便利
- 現在の作業ディレクトリのことをカレントディレクトリ（ワーキングディレクトリ）という
- カレントディレクトリからの相対パスでファイルを指定することができる
- ターミナルを起動した直後のカレントディレクトリは「ホームディレクトリ/CNSiMac」になる

カレントディレクトリの表示

- pwd（print working directoryの略）コマンド
  - カレントディレクトリの絶対パスを表示する

```
% pwd ↵
/a/fs0102a/t10472ms
```

ファイルサーバは何台かのコンピュータで分担してホームディレクトリを保管しているので，本当のホームディレクトリの絶対パス名は /a/fs0102a/t10472msのようにファイルサーバの番号とログイン名を組み合わせたものになっています
誰のホームディレクトリがどのファイルサーバにあるかを覚えるのは大変なので，/a/fs0102a/の部分をまとめて，'/home' と表わします

ディレクトリの内容を見る1

- ls（listの略）コマンド
  - カレントディレクトリにあるファイルとディレクトリの一覧を表示する

```
% ls ↵
Desktop         Maildir         XPAppCNS
Wnn             XPDataCNS
```

ディレクトリの内容を見る2

- -a オプション
  - .emacsのように先頭がドットで始まるファイルは，ソフトの設定に使うファイルなので普通は表示されない
  - ls コマンドに –a オプションを付けると表示できる
  - ls のあとに1つ空白をあけてからオプションを入力する
  - オプションをつけることで，コマンドの機能を拡張できる

```
% ls -a ↵
.                       .gnome2                 .w3m
..                      .gnome2_private         .winman
.ICEauthority           .gstreamer-0.8          .xsession-errors
.cshrc                  .gtkrc-1.2-gnome2       Desktop
.emacs                  .metacity               Maildir
.emacs.d                .mh_profile             Wnn
.folders                .mozilla                XPAppCNS
.fonts.cache-1          .nautilus               XPDataCNS
```

ディレクトリの内容を見る3

- ディレクトリのパス名を引数(ひきすう)として指定
  - カレントディレクトリ以外のディレクトリを見たいときは，そのディレクトリのパス名を ls の後につける
  - コマンドの後に1つ空白をあけてから付け加えるものをこのコマンドの引数(ひきすう)と言う

```
% ls Maildir↵
courierimaphieracl      courierimapuiddb        tmp
courierimapkeywords     cur
courierimapsubscribed   new
```

【演習 Macの人】
lsコマンドを極めよう

- 自分のホームディレクトリにあるファイルのうち最も新しいファイルを見つけてみよう
  - ファイルを新しい順に表示するオプションは -t
- 実験してみよう
  - ls の引数に存在しないディレクトリ名を指定してみる
  - ls -l の引数にディレクトリでは無く，ファイルを指定してみる
- ls コマンドのその他の機能について調べてみよう
  - コマンドのマニュアルを表示するには，man コマンドを使う（manの引数に調べたいコマンド名を指定する）

```
% man ls ↵
```

カレントディレクトリの移動

- cd（change directoryの略）コマンド
  - 移動したいディレクトリのパス名を引数として指定
  - 引数のディレクトリのパス名は相対パスでも絶対パスでもよい
  - 引数を省略すると，カレントディレクトリをホームディレクトリ（特別教室のMacの場合は，「ホームディレクトリ/CNSiMac」）に変更する

```
% cd Maildir↵
% pwd↵
/a/fs0102a/t10472ms/Maildir
```

```
% cd /home/t10472ms/Maildir↵
% pwd↵
/a/fs0102a/t10472ms/Maildir
```

ファイルの内容を見る

- less コマンド
  - テキストファイルの中身を見ることができる
  - 引数に内容を見たいファイル名を指定
  - ファイルをスクロールするには，Spaceキーを使う
  - 閲覧を終了するにはqキーを押す

新しいディレクトリを作る

- mkdir（make directoryの略）コマンド
  - 引数に作りたいディレクトリの名前を指定する
  - 正常に作成できると，何も表示されないので，lsコマンドで確認するとよい

```
% mkdir memo ↵
% ls ↵
Desktop         Maildir         XPAppCNS
Wnn             XPDataCNS       test1
test            memo
```

ファイルをコピーする

- cp（copyの略）コマンド
  - コピー元のファイルと新しく作るファイルの名前を空白で区切って引数で指定する
  - 新しく作るファイルの代わりにディレクトリ名を指定すると，そのディレクトリの中に同じ名前で新しいファイルが作成される

```
% ls ↵
fileA    testdir
% cp fileA fileB ↵
% ls ↵
testdir  fileA    fileB
% cp fileA testdir ↵
% ls testdir ↵
fileA
```

ファイルの移動・ファイル名の変更

- mv（moveの略）コマンド
  - ファイルを移動する場合は，mv の後に移動したいファイルの名前，移動先のパス（相対パスか絶対パス）を空白で区切って指定する
  - ファイル名を変更する場合は，mv の後に変更したいファイルの名前，新しいファイル名を空白で区切って指定する

```
% ls ↵
fileA    testdir
% mv fileA testdir ↵
% cd testdir ↵
% ls ↵
fileA
% mv fileA fileB ↵
% ls ↵
fileB
```

※ファイル名の変更の際，変更先のファイル名が既に存在するものであった場合，そのファイルに上書きされ元の内容は消えてしまうので注意

ファイルの削除

- rm（removeの略）コマンド
  - 削除したいファイル名を引数として指定
  - 空白で区切って複数のファイル名を指定できる

```
% ls ↵
fileA fileB
% rm fileA ↵
% ls ↵
fileB
```

ディレクトリの削除

- rmdir（remove directoryの略）コマンド
  - 削除したいディレクトリ名を引数として指定
  - 空白で区切って複数のディレクトリ名を指定できる
  - ディレクトリの下にファイルがある場合，削除できない
  - ディレクトリの下にあるファイルを全て削除するか移動するかした後，ディレクトリを削除する

```
% ls ↵
testdir fileA fileB
% rmdir testdir ↵
% ls ↵
fileA fileB
```

【前回演習】
宝探しゲームをしてみよう

- 宝探しゲームをしてみましょう
  - 出発点は
  - cd コマンドでサブディレクトリに移動し，ls コマンドで何があるか調べる
  - ファイルが置いてある場合，宝かどうかmore コマンドでファイルの中身を見る（ハズレの場合もあります）
  - 宝が無いと分かったら，親ディレクトリに移動して別のところを探す cd ..

- 宝を発見したら，宝島の地図（ディレクトリ構造図）を書いておきましょう

第3回

- プロセス，ジョブ
- プロセス管理、時分割処理とプロセス切り替え、スケジューリング、プロセス表

アクセス権と保護モード

アクセス権と保護モード

- CNSの他のユーザのファイル（メールの内容等）を勝手に閲覧されては困るため，適切なアクセス権を設定する必要がある
- ファイルやディレクトリごとに，他のユーザからのアクセスを許可したり，禁止したりする保護モードを設定する機能がある
- 保護モードは「誰が」と「どうする」という組み合わせ（3×3）に対して，許可か禁止かを決めたもの

誰が

1. ファイルの持ち主のユーザ自身（user）
2. グループのメンバ（group）
   ※ 学生はすべて同じグループに属する
3. その他（other）

×

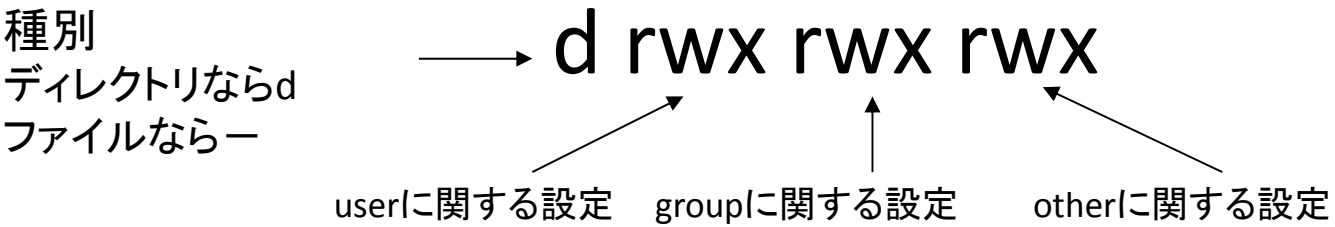
どうする

1. 読み出し（read）
2. 書き込み（write）
3. 実行（execute）

保護モードの確認

- ls -lで保護モードの確認ができる

```
% ls -l
-rw-r--r-- 1 t10472ms  student  153  Apr 20 15:30  fileA
```



「r（readの略）」,「w（writeの略）」,「x（executeの略）」それぞれの許可
「ー」は禁止を示す

- 新しく作ったファイルは rw-r--r--になる
- ディレクトリは rwxr-xr-x になる
- メールを保存するディレクトリ（Mail）はrwx------

保護モードの変更

- chmod（change modeの略）コマンド
  - 1番目の引数で，「誰が（u，g，o）」と「どうする」（r，w，x）を＋か－でつないで保護モードを指定（＋は許可，－は禁止）
  - 2番目の引数で変更したいファイル名またディレクトリ名を指定する

```
% ls -l ↵
total 0
-rw-r--r-- 1 t10472ms  student  153  Apr 20 15:30  fileA
% chmod go-r fileA
% ls -l
total 0
-rw------- 1 t10472ms  student  153  Apr 20 15:30  fileA
```

保護モード数字で指定

- ファイルのパーミッション（権限）は、数字で表現することも出来る。
  r（読み込み）を『4』
- w（書き込み）を『2』
- x（実行）を『1』と表現。

- 例）rw-rw-r-- hoge.txt
  上記ファイルはrw-（4+2）rw-（4+2）r--（4) hoge.txt
- のため、パーミッション『664』とも表現できます。
- ですので、Otherに書き込み権限を与えるには、下記でも同じ結果になります。
  chmod 666 hoge.txt

chown

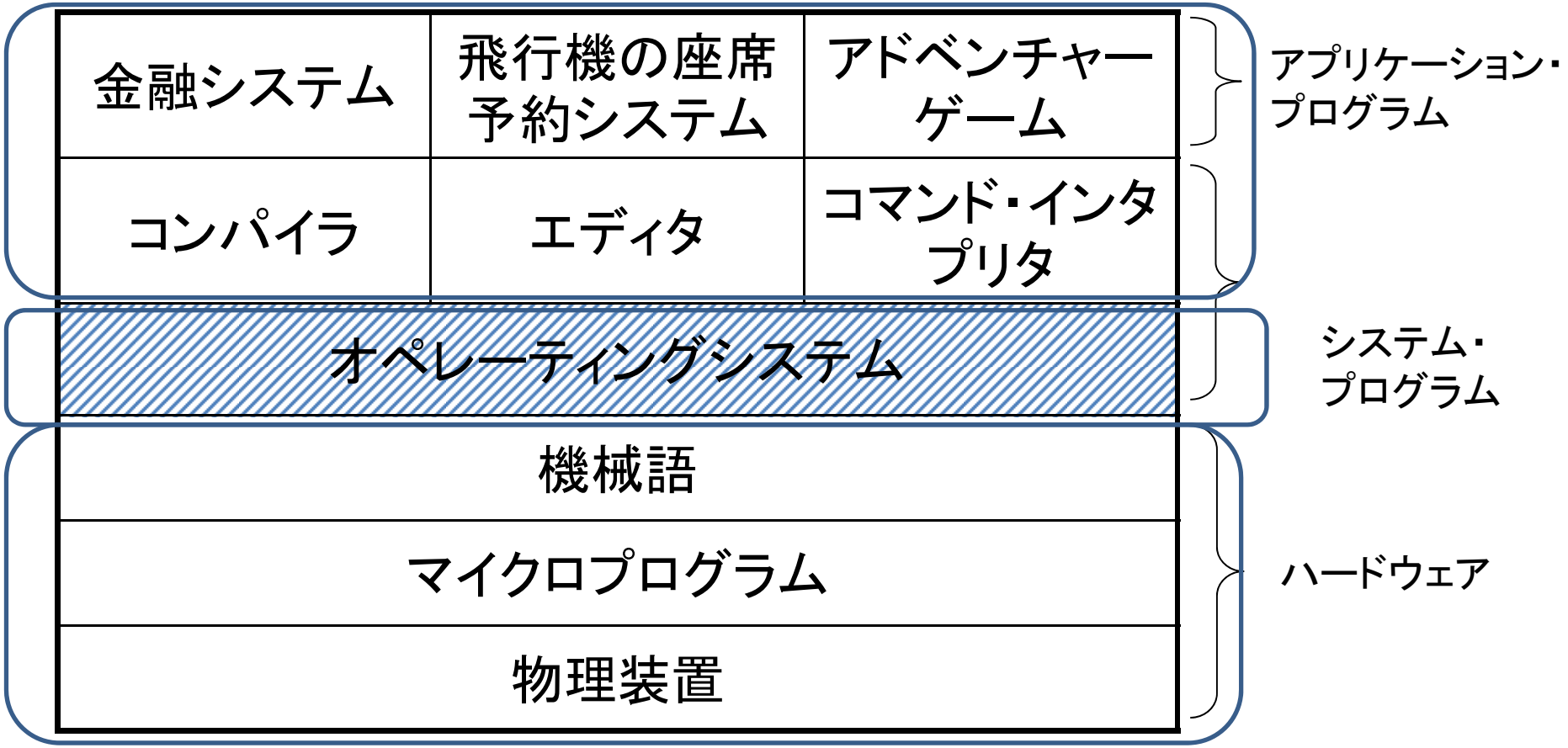
- すべてのファイルには所有者という概念が存在します。
- ファイルのオーナーを変更したいとき。

- 例）hoge.txtのオーナーをuserAからuserBに変更する。

  chown userB hoge.txt

  -rw-rw-r-- 1 userB userA hoge.txt

OSの位置づけ


金融システム
飛行機の座席予約システム
アドベンチャーゲーム
コンパイラ
エディタ
コマンド・インタプリタ
オペレーティングシステム
機械語
マイクロプログラム
物理装置
アプリケーション・プログラム
システム・プログラム
ハードウェア

OSとは何か

- 拡張マシンとしてのOS
  - ハードウェアの詳細を隠蔽し、拡張マシン(extended machine)、仮想マシン(virtual machine)としての機能をユーザに提供
- 資源管理システムとしてのOS
  - CPU、メモリなどを管理する
    - 複数のプロセス、複数のユーザに競合や不整合が起こらないように資源を割り当てる

単位の説明

- ジョブ
  - ユーザがシステムに対して依頼する仕事の単位
- プロセス
  - システムが処理する仕事の単位
  - システムはジョブをプロセスに分割して処理
  - リソースはプロセス単位で割り当てられる

プログラムの処理形態

- バッチ処理
  - 必要な情報を前もって決定
  - 実行してほしいジョブを一括依頼

- 対話（インタラクティブ）処理
  - そのつど、プログラムに対して入力を行う

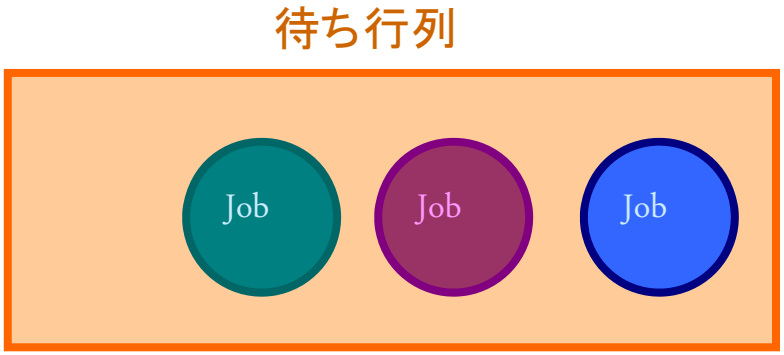
バッチ処理

- ジョブを一括して依頼
  - ユーザは、必要なリソース、実行したいプログラム、処理に使うデータの全てをあらかじめ決定
  - それをJCL (Job Control Language) によって記述し、コンピュータに投入

```
//JOB1                    JOB (12345), CLASS=X

//STEP1     EXEC PGM=TEST
//DDIN      DD DISP=SHR, DSN=INPUT1
//DDOUT     DD DISP=(NEW, CATLG), DNS=OUTPUT(+1),
//                    UNIT=SYSDA, SPACE=(CYL,(15,15),RLSE),DCB+*.DDIN
                      :
```

バッチ処理



リソース
（CPU, メモリなど）

- 一定期間ごとに大量のデータを集めて処理するような場合に便利
  - 例）売り上げデータ・受注データの集計

バッチ処理

- 利点
  - スケジューリングが単純
    - 前もって、プログラムが必要とするリソースがわかる
    - 複数のジョブのスケジューリングが比較的楽
      例）必要リソースの少ないジョブを優先的に実行すると
      全ジョブの平均待ち時間（ターンアラウンドタイム）
      が短くなる
    - ジョブの切り替えも少なくてすむため、無駄が少ない
- 欠点
  - 前もって全てを決めないとジョブが投入できない

対話（インタラクティブ）処理

- 必要に応じて人間が入力し、それを処理する
  - 人間の反応速度は 10^{-1}秒 程度
  - 計算機は 10^{-9}秒 オーダで命令を処理
  - 人間の入力を待っている時間はすごく無駄
    特に昔、計算機が非常に高価だった時代はなおさら
  - 一定時間に処理できる仕事量（スループット）が低下
    ＝もったいない

- **タイムシェアリングシステム（TSS）**
  - **使っていないCPU時間を他に割り当てよう!**

タイムシェアリングシステム

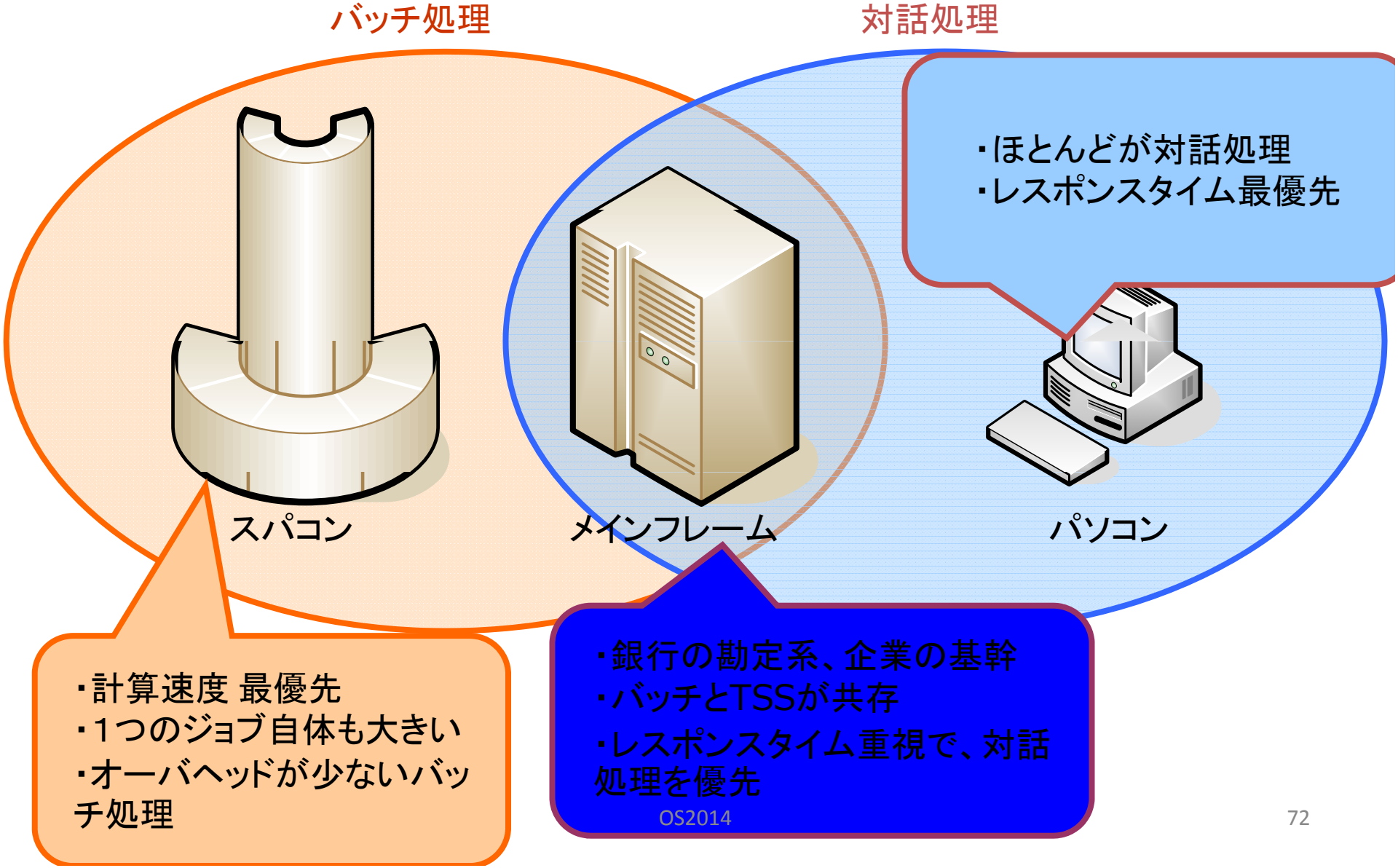
- 非常に短いCPU時間を複数プロセスに順に割当
  - 割り当てられる単位時間（クオンタム）は数十ms
  - 割当が一周したらまた最初から

- プロセスから見ると…
  - 細切れのCPU使用権（時間）が、短い間隔を置きながら与えられる
  - プロセスは入力待ちなどの遊びも多いので、この「短い間隔」はユーザからは見えにくく、さもCPUを占有して使っているように見える

タイムシェアリングシステム

- 利点
  - 自分のほかに大きなプロセスがあっても、
    そのプロセスが終わるまで長い間待たされたりしない
    （↑バッチ処理ではありうる）
  - ユーザから見ても、対話的に入力してからその反応が
    返ってくるまでの時間（レスポンスタイム）が
    短くなる

バッチ処理と対話処理


バッチ処理
対話処理
・ほとんどが対話処理
・レスポンスタイム最優先
スパコン
メインフレーム
パソコン
・計算速度 最優先
・1つのジョブ自体も大きい
・オーバヘッドが少ないバッチ処理
・銀行の勘定系、企業の基幹
・バッチとTSSが共存
・レスポンスタイム重視で、対話処理を優先

リアルタイム処理は実はとても難しい

- ある決められた時間に必ず処理を終わらせる
  - 例）自動車の安全装置
  - 「他のプロセスにリソースが使われていたので間に合いませんでした」では済まない！
  - クオンタムの短いTSSでも間に合わない可能性

- どうやったら実現できるか ＝ 非常に複雑
  - 同時実行プロセス数に制限
  - 記憶領域の使用制限
  - etc, etc...
  - しかも処理速度を落としてはいけない

プロセスとは

- プロセス
  - リソースの割当対象となる（仕事の）単位
  - OSに対してリソースを要求
  - OSからリソースの割当を受ける

プロセス(プログラム)とプロセッサ

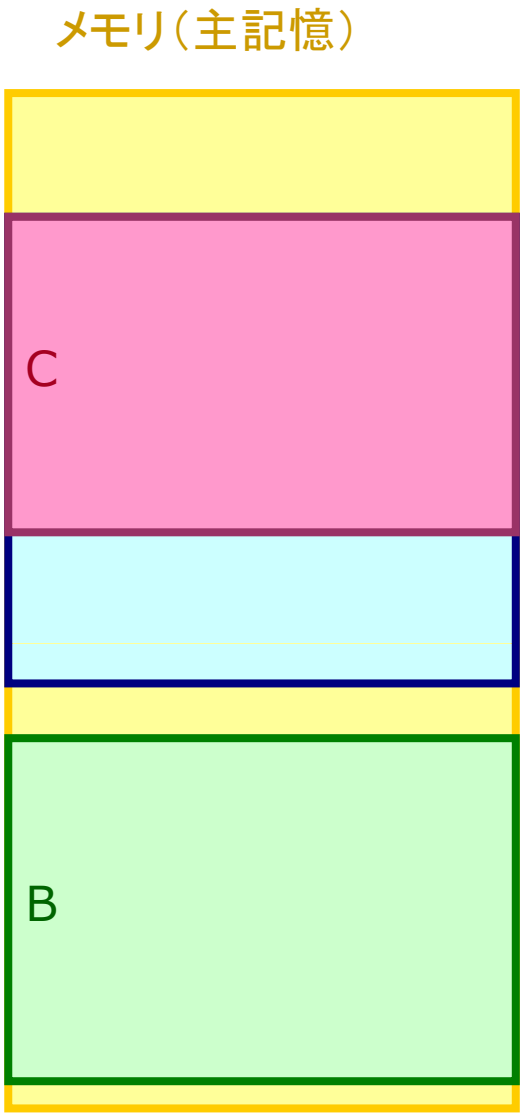
- ユニプロセッサ・ユニプログラミング
  - ひとつのCPUに対してひとつのプロセス
  - バッチ処理

- ユニプロセッサ・マルチプログラミング
  - ひとつのCPUに対して複数のプロセス
  - TSS

- マルチプロセッサ・マルチプログラミング
  - 複数のCPUに対して複数のプロセス
  - 並列・分散処理

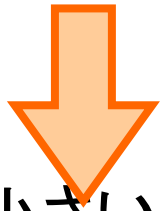
プロセスの切り替え

- 複数プロセスを切替えながら実行
  - プロセスA
  - プロセスB
    - 記憶領域の圧迫
  - プロセスC
    - 記憶領域の不足
    - 置き換えコスト
  - プロセスA
    - また不足
    - また置き換え



プロセスとスレッド

- 複数プロセスの同時実行はコストが高い
  - メモリ使用量が増加
  - 切り替えコストも大きい
- 複数CPUを備えた計算機の一般化
  - デュアルプロセッサ，デュアルコア
  - 同時実行できるプロセス数よりCPUが多いとCPUが遊んでいてもったいない



- スレッド
  - プロセスをさらに小さい単位に分割
  - CPUリソースをスレッドごとに割当

スレッド

- 例）Microsoft Office
  - プロセス
    - Microsoft Word
    - Microsoft Excel
    - 各プログラムはプロセスとして処理
  - スレッド
    - たとえばWordの場合
    - 印刷
    - 編集
    - など、同じ「Word」というプログラムの中で、同時（並行）動作できる単位がある！

スレッド

- リソース割当
  - プロセス単位
    - メモリ，入出力デバイス，etc...
  - スレッド単位
    - CPU
- スレッド
  - TSSによる切り替えオーバヘッドが軽い
    - 同一プロセスから生成されてるからメモリ領域が同じ
    - メモリ使用量は1プロセス分ですむ
  - 別名 : Light Weight Process (軽いプロセス)

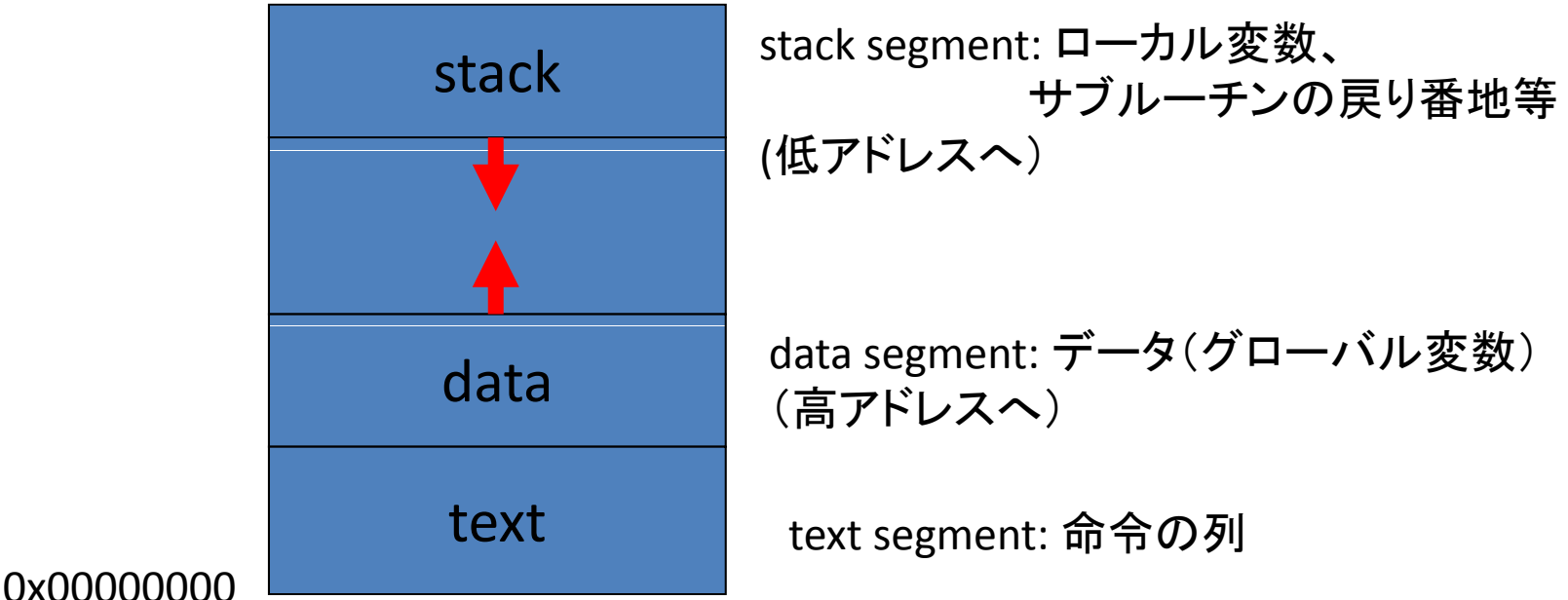
プロセス

- プロセス = 実行中のプログラム
  - アドレス空間（コアイメージ）
    - 実行可能プログラム、データ、スタック
  - プロセス表
    - レジスタや実行に必要なほかの情報
- プロセスにおける木構造
- シグナル
  - タイマ終了などの通知
  - プロセス間通信

プロセスの概要

- 実行中のプログラム
  - プログラム・カウンタ、レジスタ、変数を含む

プロセスのメモリ・イメージ



マルチプログラミング

- コンピュータ上では複数のプロセスが実行されている
- 1つのCPUは一度に1つのプロセスのみ実行される

- 複数のプロセスが並列実行されているようにみせかける（疑似並列:pseudoparallelism）

→ マルチプログラミング

実行するプロセスを高速に切り替える（スケジューリング）

【超重要】プロセスの状態

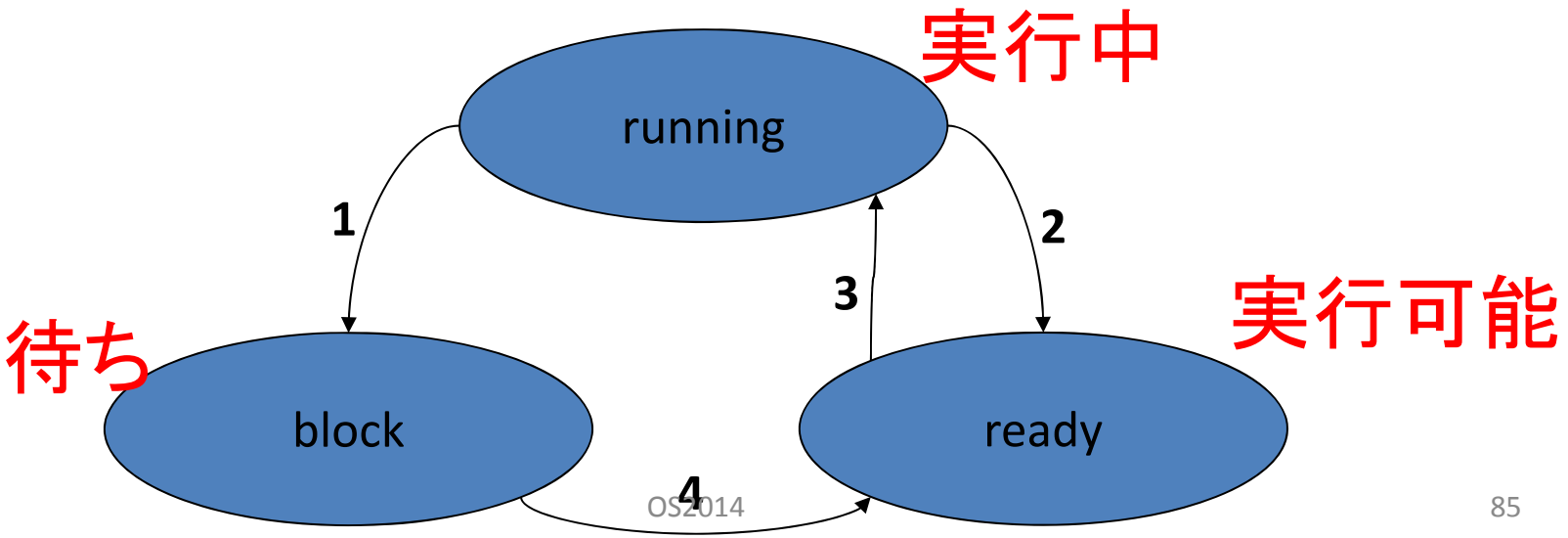
- プロセスの3つの状態
  - 実行中（running）
    - CPUが割り当てられている状態
  - 実行可能状態（ready）
    - 実行可能だが、ほかのプロセスが実行中のため、一時的に待機している状態
  - 待ち状態（block）
    - 外部イベントが発生するまで実行不可能な状態

マルチプログラミング

- プロセッサは常に1つのプログラムしか実行できない
  - マルチタスクでは，複数のプログラムを切り替えて実行しなければならない。
  - プリエンプティブ方式は，OSがプロセッサの実行権限を管理し，プロセスの実行を切り替える方式である。現在のOSは，プリエンプティブ方式が主流
  - ノンプリエンプティブ方式（「擬似マルチタスク方式」）は，プロセスの切り替えをプログラム自身に任せる方式。あるプログラムがプロセッサを長時間占有することも可能で，この場合はシングルタスクと同じになってしまう。
  - 昔のMac OSやWindows 3.1は，ノンプリエンプティブ方式。

【重要】プロセスの3状態遷移

1. プロセスが(例えば)入力待ちでブロック状態へ
2. スケジューラが別のプロセスを選択する
3. スケジューラがこのプロセスを選択する
4. 入力データが準備できた



スケジューラ

- ready状態にあるプロセス群から、どのプロセスにCPUを割り当てrunning状態にするかを決定する機能

スレッド

- 軽いプロセス
- アドレス空間共有
- スレッド毎に、プログラムカウンタ、レジスタ管理、スタック、PSW(状態)

プロセスの切り替え

- CPUの仮想化
  - OSがプロセス・スレッドに対して CPUの実行権を微小時間与える
- 割込み
  - 通常のCPU演算動作とは異なる事象のこと
    - キーボード入力を受け取った
    - 自動車がどこかに衝突した
    - サーバからデータが送られてきた
  - 割込み発生時にプロセスの切り替えが起こる
  - TSSでは、プロセス切り替えのために インターバルタイマーが定期的に割込みを発生

割込み

- 割込み処理
  - 割込みは、即座に処理すべき場合が多い
  - 高速かつ軽量に割込みを処理する実行方式

割込みの種類

- 内部割込み
  - 実行中のプログラムを発生原因とする
  - 例）プログラム自体が他の処理を要求
    プログラム自体の異常

- 外部割込み
  - その他の要因で発生する
  - 例）他の優先的処理からの要求
    順番待ちしていた他の処理への移行
    ハードウェア異常
    特殊な処理

割込みの種類

- 内部割込み
  - スーパバイザコール割込み
  - プログラムチェック（例外）割込み

- 外部割込み
  - 入出力割込み
  - タイマ割込み
  - マシンチェック割込み
  - リスタート割込み

割込みの種類

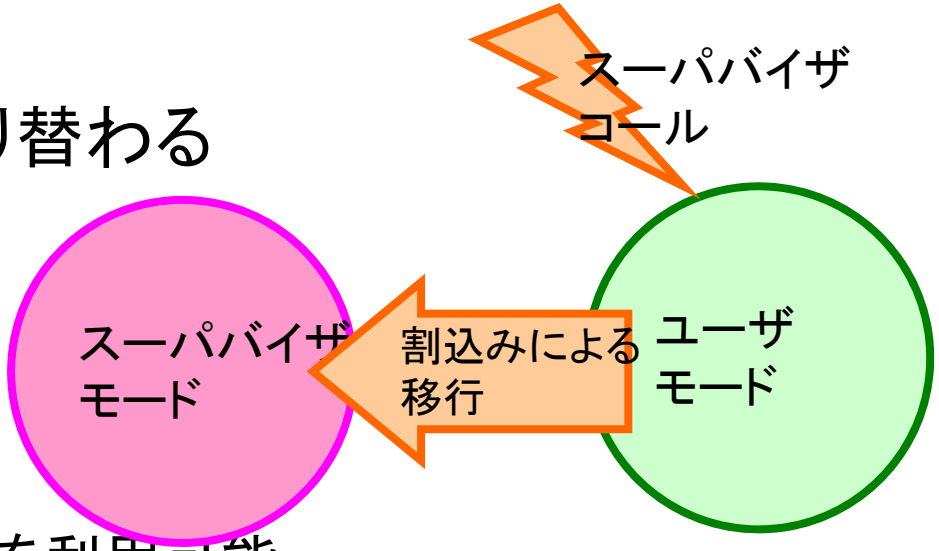
- 内部割込み
  - スーパバイザコール割込み
  - プログラムチェック（例外）割込み
- 外部割込み
  - 入出力割込み
  - タイマ割込み
  - マシンチェック割込み
  - リスタート割込み

内部割込み：スーパバイザコール割込み

- ユーザモード
  - アプリケーションには許されていない処理がある
    - プロセスの切り替え
    - 入出力デバイスへのアクセス
    - etc..
- スーパバイザモード
  - そこでアプリケーションは、OSに対して処理を依頼
  - OSの権限で，処理を実行してもらう
- スーパバイザコール
  - アプリケーションがOSに処理を依頼すること

スーパバイザコール割込み

- スーパバイザコール
  - このとき割込みが発生
  - CPUの実行モードが切り替わる



- CPUの実行モード
  - スーパバイザモード
    - OSを実行するモード
    - CPU内の全てのリソースを利用可能
  - ユーザモード
    - アプリケーションを実行するモード
    - 利用できるリソースに制限あり

割込みの種類

- 内部割込み
  - スーパバイザコール割込み
  - プログラムチェック（例外）割込み

- 外部割込み
  - 入出力割込み
  - タイマ割込み
  - マシンチェック割込み
  - リスタート割込み

内部割込み：プログラムチェック（例外）割込み

- 実行中のプログラムで異常が発生したとき
  - ゼロによる除算

```
division by zero
```

  - 演算時のオーバフロー

```
integer overflow
```

  - 不正なメモリアドレスへのアクセス

```
segmentation violat
```

- この割込みを検知するしくみがないと…
  - 上記の異常が発生したときに
    システム全体が停止してしまうかも…

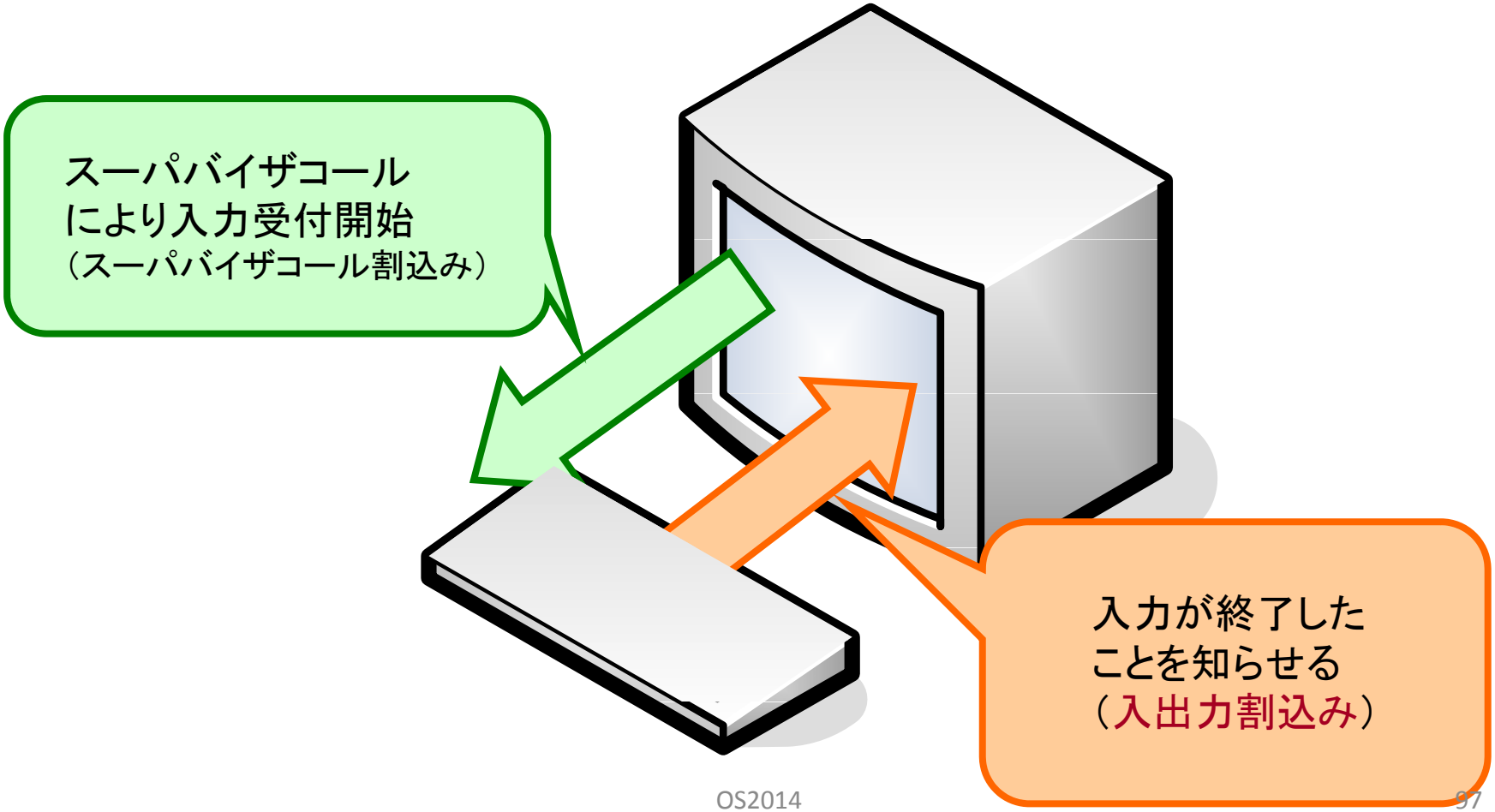
割込みの種類

- 内部割込み
  - スーパバイザコール割込み
  - プログラムチェック（例外）割込み

- 外部割込み
  - 入出力割込み
  - タイマ割込み
  - マシンチェック割込み
  - リスタート割込み

外部割込み：入出力割込み

- 入出力装置から発生する割込み



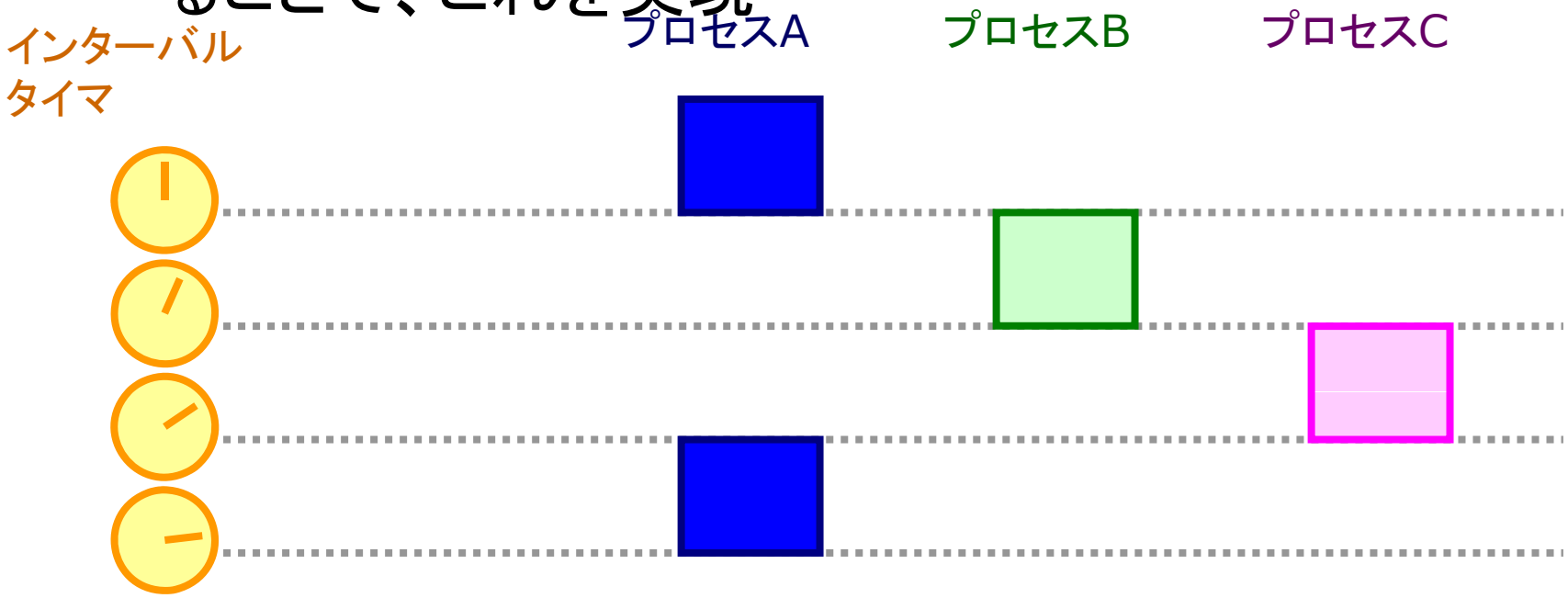
割込みの種類

- 内部割込み
  - スーパバイザコール割込み
  - プログラムチェック（例外）割込み

- 外部割込み
  - 入出力割込み
  - タイマ割込み
  - マシンチェック割込み
  - リスタート割込み

外部割込み：タイマ割込み

- インターバルタイマによる割込み
  - TSSでは，定期的な切り替えが必要
  - インターバルタイマが定期的に割込みを発生させることで、これを実現



割込みの種類

- 内部割込み
  - スーパバイザコール割込み
  - プログラムチェック（例外）割込み

- 外部割込み
  - 入出力割込み
  - タイマ割込み
  - マシンチェック割込み
  - リスタート割込み

外部割込み:
マシンチェック割込み

- ハードウェアによって通知される異常時に発生する割り込み
  - 冷却装置の異常
    - 内部の温度が上がりすぎているのを検出
  - 電源装置の異常
  - etc...

外部割込み:リスタート割込み

- システムをリセットするときに発生する割込み

割込みの種類:まとめ

- 内部割込み
  - スーパバイザコール割込み
  - プログラムチェック(例外)割込み

- 外部割込み
  - 入出力割込み
  - タイマ割込み
  - マシンチェック割込み
  - リスタート割込み

割込みによる
プロセスの中断と再開

割込み発生時の処理

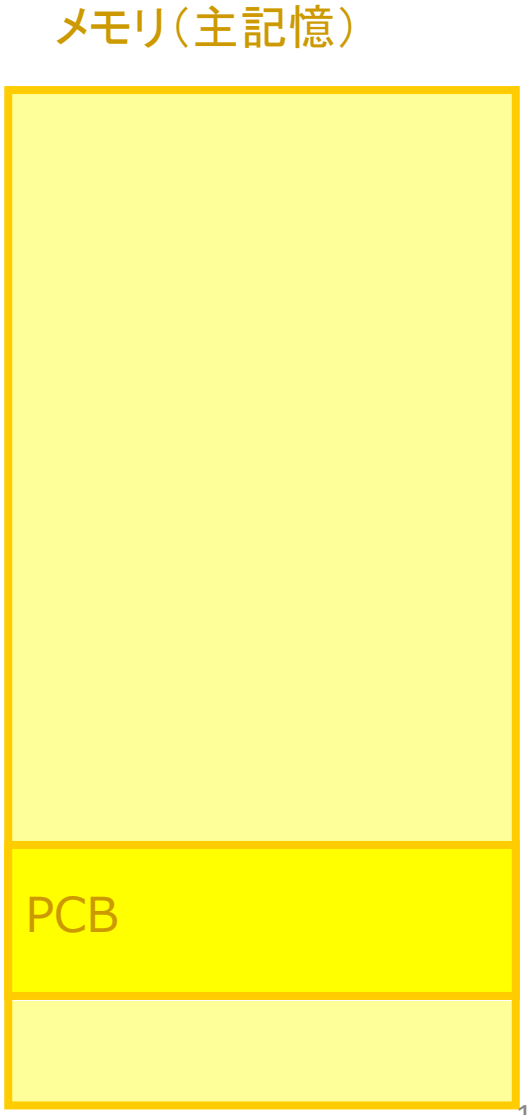
- 実行中のプロセスを中断
- 割込み処理ルーチンに移行
- 割込み処理が終わったら、プロセスを再開

プロセスの中断

- PSW（Program Status Word）
  - プロセスは、後で再開しないといけない
  - 再開するためには、今の途中状態を覚えておかないと
    いけない
  - 状態：
    - プログラムカウンタの値
    - スタックレジスタの値
    - 汎用レジスタの値
    - 割込みマスクの値
    - etc...

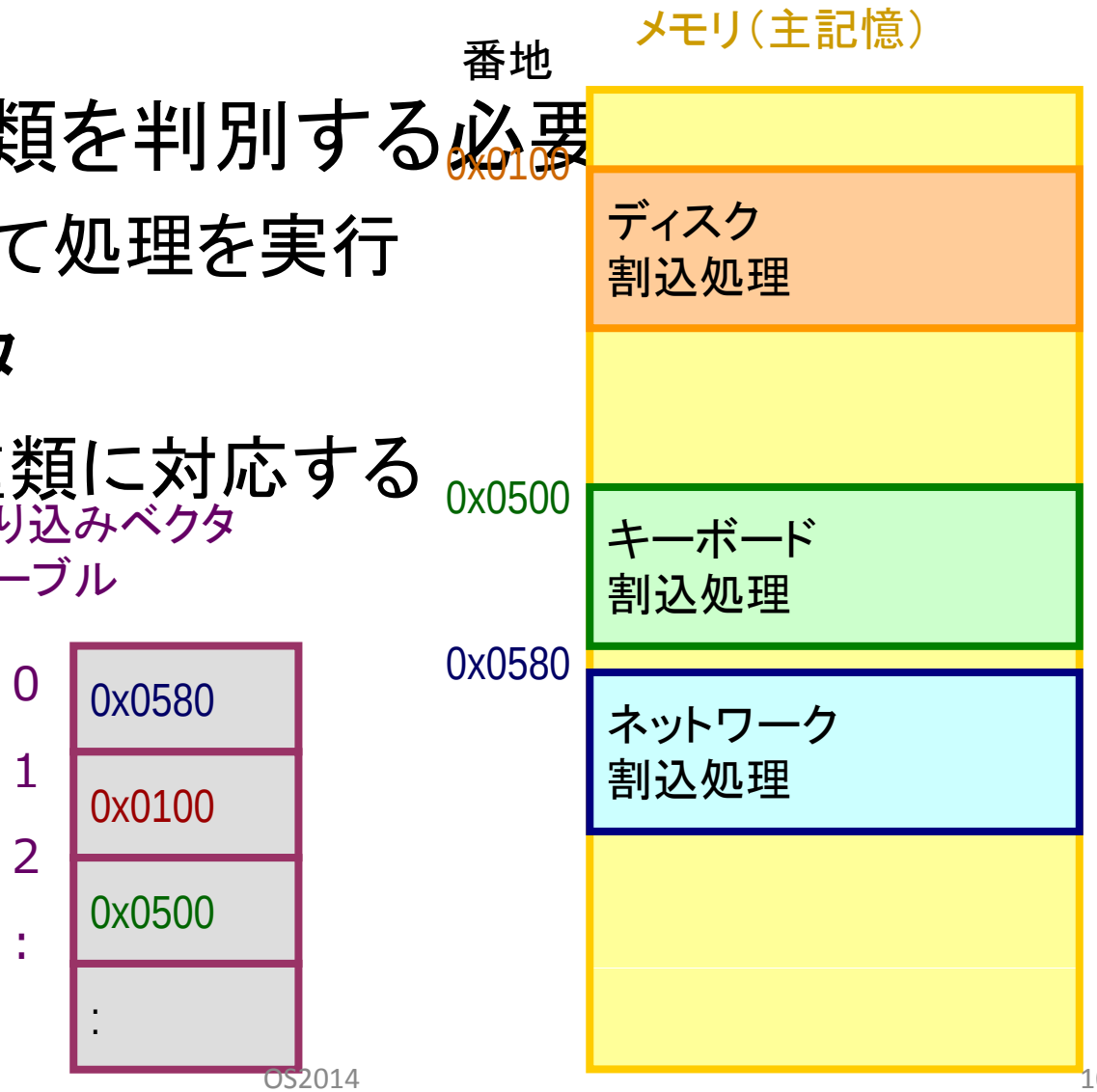
プロセスの中断

- PCB (Process Control Block)
  - メモリ上の、PSWを退避するための領域



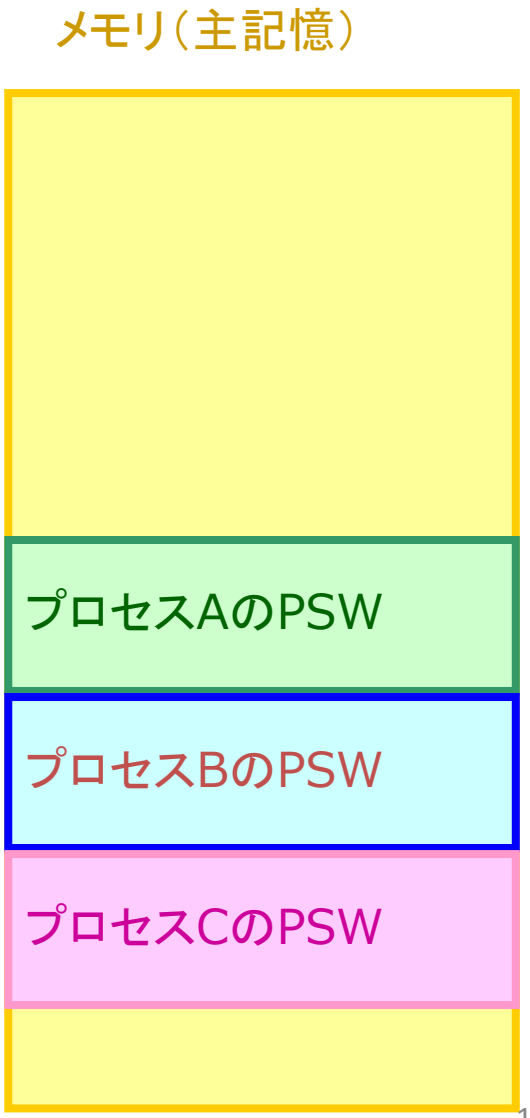
割込み処理ルーチンの仕事

- 割込みの種類を判別する必要
  - 種類に応じて処理を実行
- 割込みベクタ
  - 割込みの種類に対応する数字（ID）



プロセスの再開

- 実行可能なプロセスからプロセスを選択
  - 割込によって中断されたプロセスが常に再開されるわけではない
- 選択されたプロセスのPSWからCPU状態を復元して再開



プロセスの中断と再開：まとめ

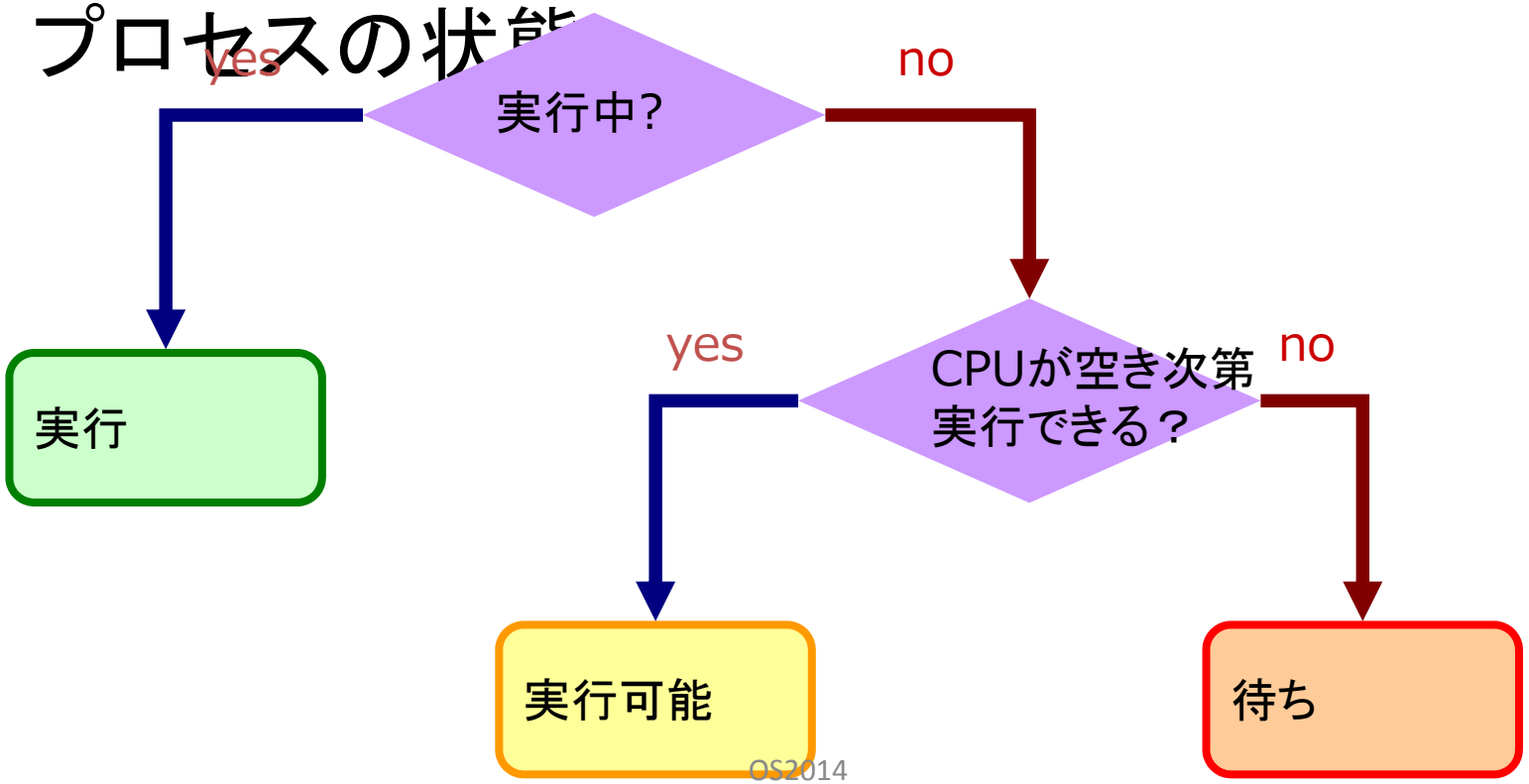
- 中断
  - CPU状態をPSWという形で、
    メモリ内のPCBへ保存
- 割込ルーチン
  - 割込ベクタ（割込の種類を示す値）を放送
  - その値を割込ベクタテーブルでひいて、
    割込に対応するルーチンの主記憶アドレスを取得
  - ルーチン実行
- 再開
  - 実行可能プロセスからスケジューラが1つ選択
  - 対応するPSWからCPU状態を復元

プロセスの三状態

プロセスの状態

- 「実行可能なプロセス」とは?

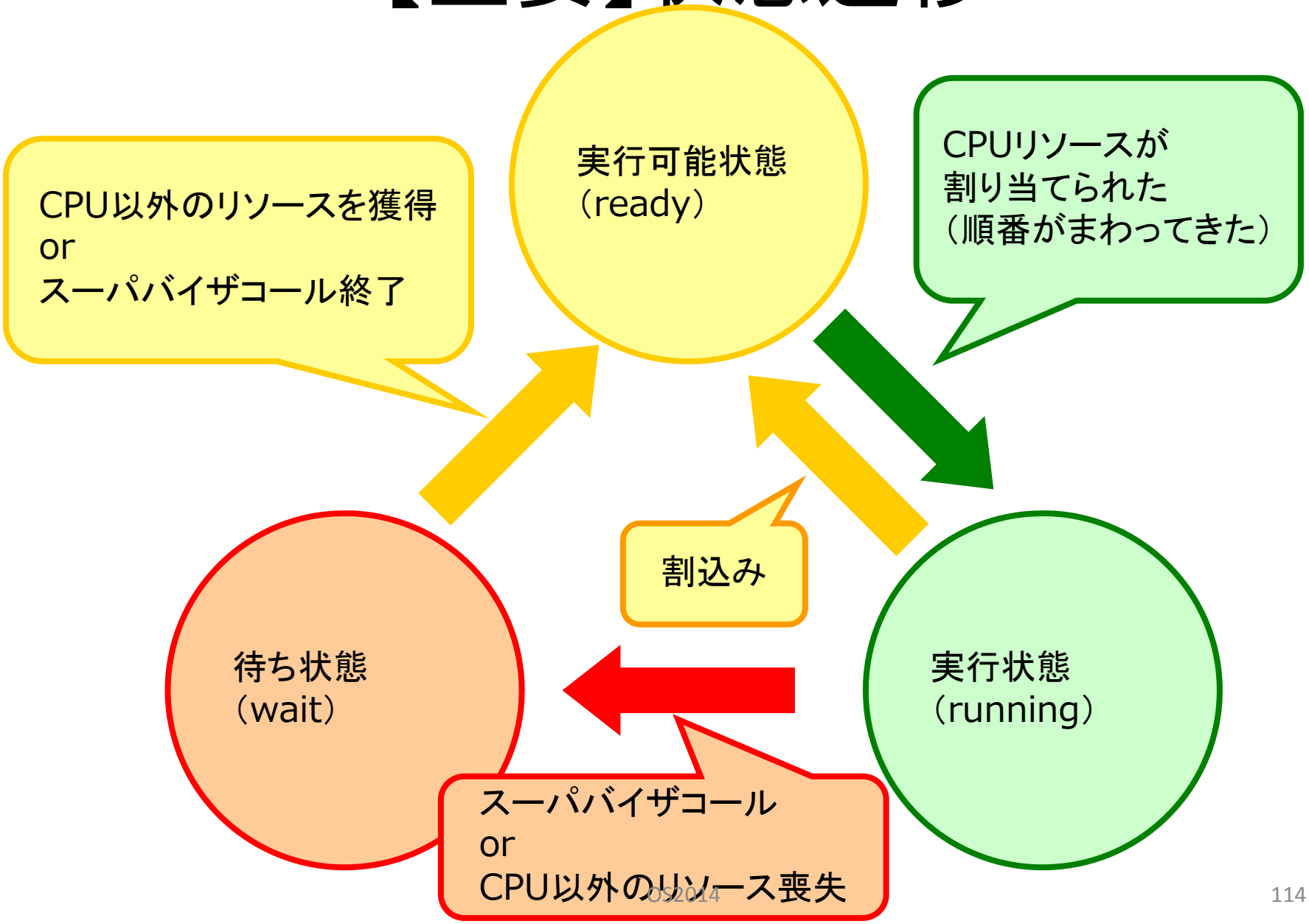
- プロセスの状態



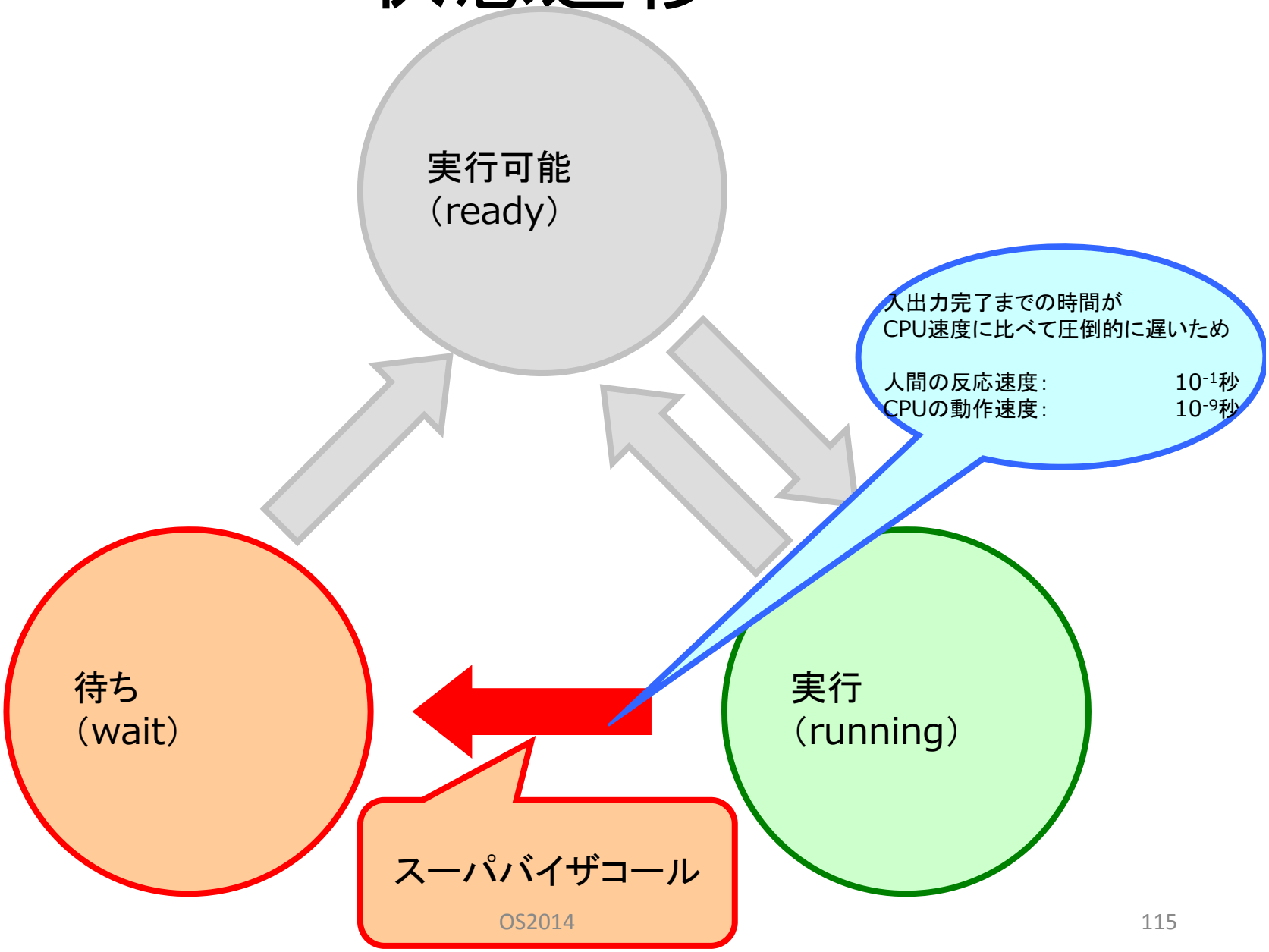
プロセスの状態

- 実行状態（running）
  - プロセスを実行している状態
  - リソースは，そのプロセスのために確保されている
- 実行可能状態（ready）
  - 実行できるが、CPUリソースが確保できていない状態
  - CPUリソースを確保した時点で実行開始される
- 待ち状態（wait）
  - CPU以外のリソースも確保できていない状態
  - 入力待ちなどもこれに含まれる

【重要】状態遷移


実行可能状態
（ready）
CPU以外のリソースを獲得
or
スーパバイザコール終了
CPUリソースが
割り当てられた
（順番がまわってきた）
割込み
待ち状態
（wait）
実行状態
（running）
スーパバイザコール
or
CPU以外のリソース喪失

状態遷移


実行可能
（ready）
入出力完了までの時間が
CPU速度に比べて圧倒的に遅いため
人間の反応速度：　10^{-1}秒
CPUの動作速度：　10^{-9}秒
待ち
（wait）
実行
（running）
スーパバイザコール

演習問題

- 平日(月曜日)の自分の一日をOSに見立て状態遷移図を作成してみよう。

- ◯は状態
- ⇒で遷移条件を表す
- 状態何個あってもよい。
- 色を分けよう。

- ページは1ページ以上、学籍番号、氏名、よみ、学内メールアドレスを記載すること
- 提出は次週授業前に

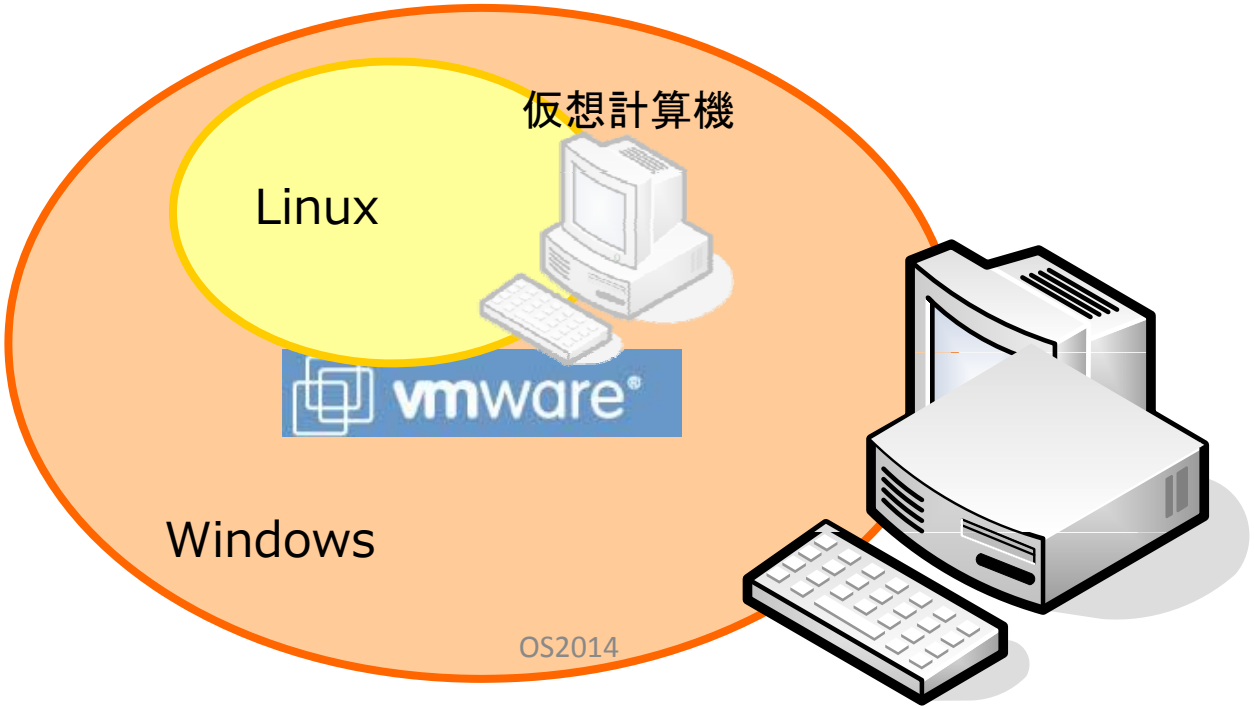
CPUの仮想化

エミュレーション

- 最近は…
  - ハードウェアが非常に高速化
  - 他のハードウェア資源全体(システム)を仮想化することも可能になってきた
- エミュレーション
  - ハードウェア環境をソフトウェアで仮想化
  - 計算機上で他の計算機環境を仮想的に提供

ハードウェアエミュレーション

- 計算機の構造そのものを仮想化
  - VMware
    - IBM/PC環境のOS (Solaris, Linux, Windows) 上に仮想的なIBM/PC環境を構築



お断り

- 本授業資料の作成にあたり
- 慶應大学SFC IPL/ITB　岩井クラス
- 戸辺義人先生
- 田浦健次朗先生
- 降旗 大介先生
- 松尾啓志先生
- Wikipediaなどの資料を参考にさせていただいています。
- ありがとうございます。

MTTF（故障までの平均時間）

- http://www8.plala.or.jp/ap2/shinraisei/shinraisei3.html#mttf

- 有名な計算式
  - MTTFraid1 = (MTTF x MTTF)/(2 x MTTR)
    - MTTF:平均故障時間
    - MTTR:平均修理時間

- ディスクの故障は独立事象か？
  - 同一機種、同一動作
- 故障の要因
  - （偶発＋環境＋使用）×個体差
- 故障確率は一定ではない