Date . . No.

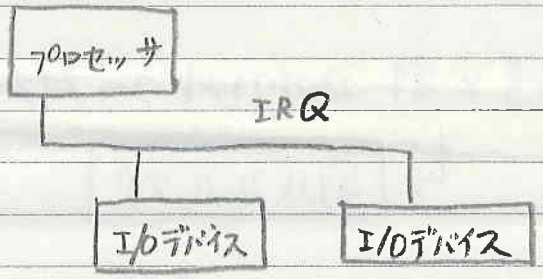
② 割込み（interrupt）

I/Oデバイスが望みの状態になったときに，プロセッサに対して割込み信号を送る方式

ハードウェアを利用して実現

プロセッサにIRQ（Interrupt ReQuest）割込み要求線が用意されている

I/Oデバイスは IRQを使って動作／準備完了をプロセッサに連絡（信号を送る）



信号を受けとったプロセッサ側の動作

1 現在実行中のプログラムを中断する

2 割り込みハンドラにジャンプする

→ 割込みが発生した場合に行う動作を記述した小さなプログラムコード

3 割込みハンドラを実行する

4 実行を中断したプログラムを再開する

No. Date

（補足説明） 割込みはどう実装されているのか

標準的なプロセッサでも15本程度のIRQを備えている

各I/Oデバイスに異なるIRQを割りあてることで区別

足りない場合，同じI/Oデバイスからの信号でも違った動作が要求される場合が考えられる

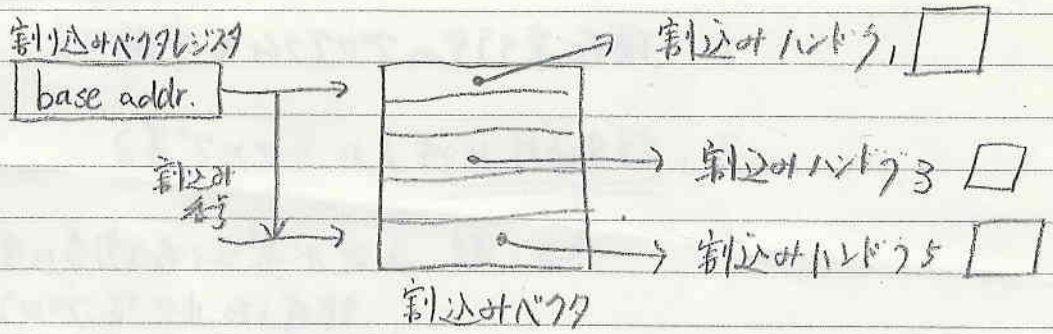
信号とは"割込み番号"を送ることでできる （0〜255程度の値）

各番号に対応して異なる割込みハンドラが起動される

割込み番号と割込みハンドラの対応関係を示すテーブル

↳ 割込みベクタ

割り込みベクタはメモリ上におかれ，その先頭アドレスは"割込みベクタレジスタ"とよばれる特殊なレジスタに保存されている．



Date . . No.

Advanced

同時に割込みが起ったら？　割込みハンドラ実行中に割込みが起こったら？

割込みには優先順位がつけられている

割込みハンドラ実行中，そのハンドラと同じか低い優先度の
割込みはマスクされ，割込み禁止の状態となる

優先度の高い割込みが生じた場合，そちらに処理をうつす

多重レベル割込み

割り込みの重要性について説明

マウスをクリックしたら　← いつおされるかわからない

プロセッサ内部で異常状態となった場合

0で割る等

割込みで処理 → "例外"と呼ばれる

Date . . No.

3 プロセスとスレッド

プロセスとは　実行状態にあるプログラムのこと

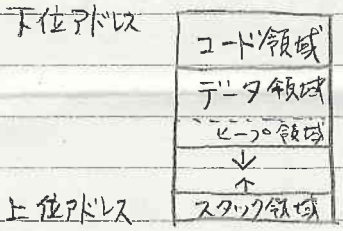
UNIX　a.out
Windows　xxx.exe
} 実行バイナリ実行ファイル　機械語によるプログラム

a.out ↵ → プロセスが生成され，プログラムが実行される

何度もたたくと，その都度 生成される

1つのプロセッサで1つのプログラムを実行するケースで考える

プロセスを生成 → メモリ上にそのプロセスに固有のメモリ空間が割りあてられる



- コード領域　プログラムコードが格納されている領域
  テキスト領域ともいう
- データ領域　プログラムで使用するデータを格納する領域
  大域変数 等がおかれる
  mallocなどにより確保されるデータがおかれる部分
  → ヒープ領域
- スタック領域　プログラムで使用するデータ（一次的に利用して，捨てるもの）がおかれる領域

新しいデータは上におかれ，破棄するときも上から順にしかできない

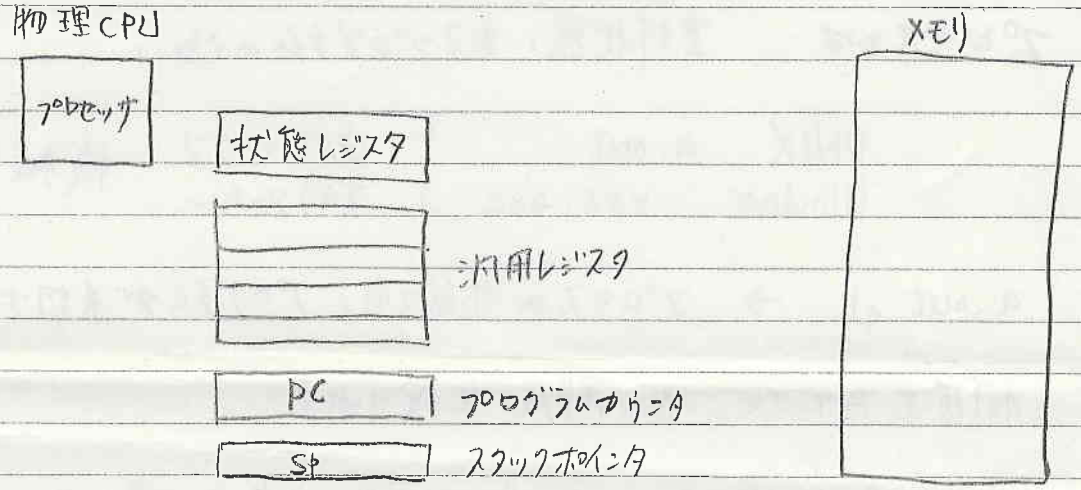
C言語プログラムのauto変数

プログラムを"ロード"する

※ コード領域にプログラムを読み込み，データ領域やスタック領域の初期化を行うと↑

No. Date

1つのプロセスを1プロセッサで実行するケース（続き）



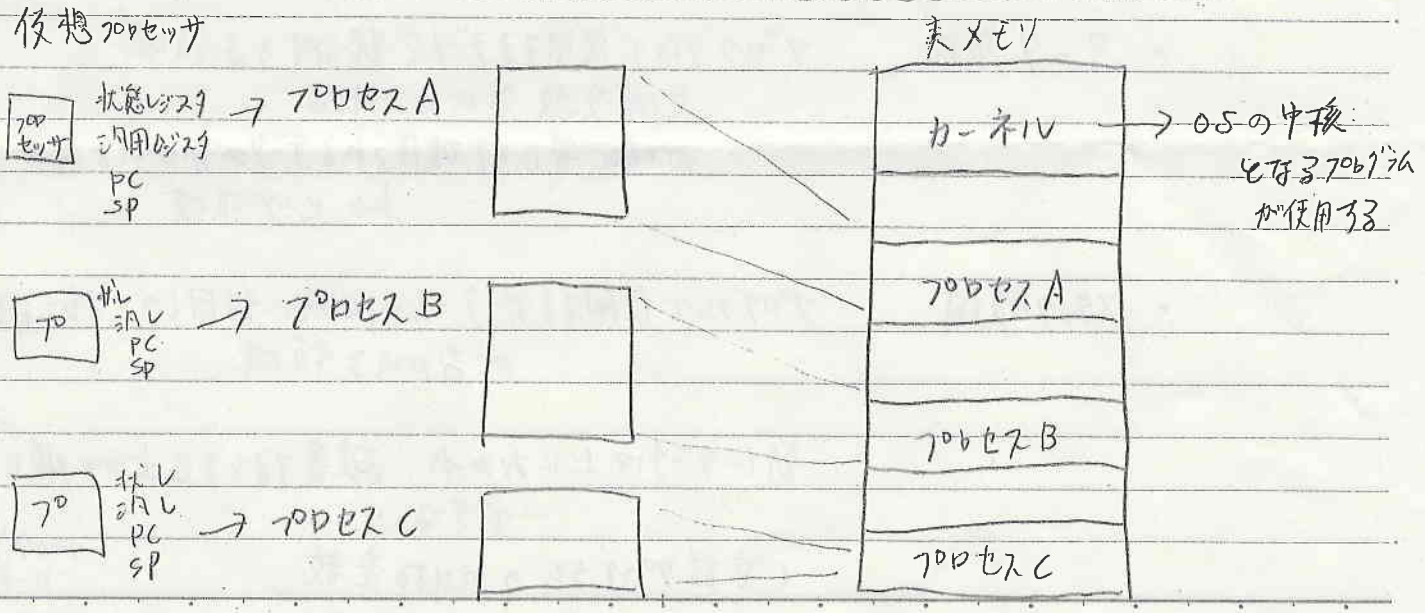
状態レジスタ：プロセッサの状態を保持している
汎用レジスタ：演算に必要なデータや結果を保持
プログラムカウンタ：次に実行する命令のアドレスを保持
SP：スタックの先頭（一番上）のアドレスを保持

複数のプロセスを同時に実行するケース

（動いているようにみせる）

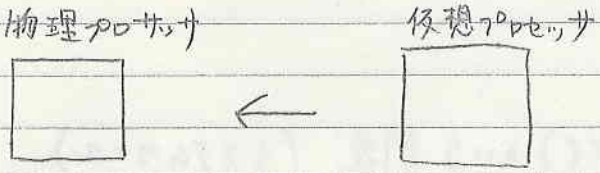
↳ このような機能を多重プログラミング，マルチタスクと呼ぶ

各プロセスに仮想プロセッサ，仮想メモリ空間を与える

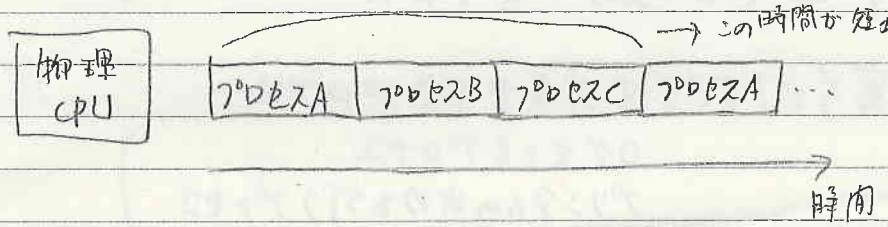


Date . . No.

プロセッサ側のふるまいについて

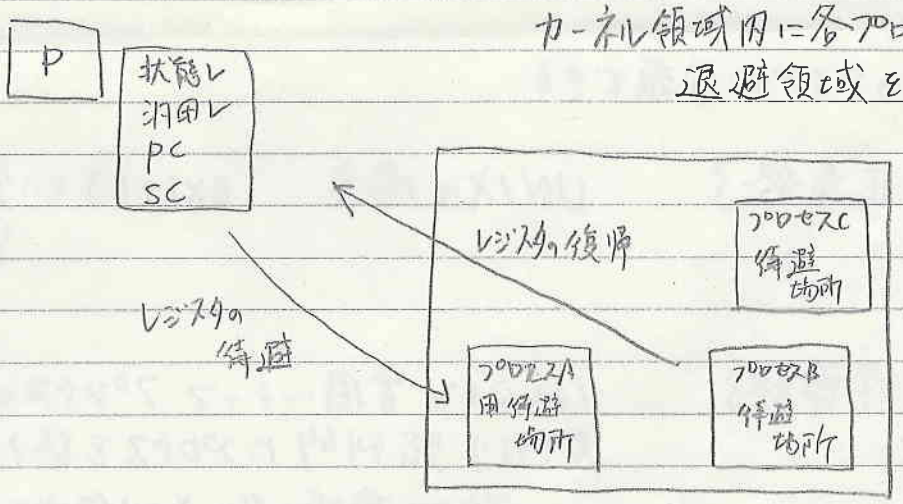


物理プロセッサへの仮想プロセッサへのマッピングを次々に切り替える



プロセスの切り替え → コンテキスト切替え とも呼ぶ

⓪ レジスタの全状態を当該のプロセスのものにする → 元にあったものはどうなる？



No. Date

プロセスの生成，終了，管理

プロセスの生成

UNIXの場合　fork()という関数（システムコール）を呼ぶことによる

PSコマンド — プロセスの一覧を表示

裏でさまざまな作業をするプロセス
- ログをとるプロセス
- プリンタへの出力を行うプロセス

UNIXでは，デーモン（daemon）とよぶ

プロセスの終了

3つのタイプに分類できる

正常終了　UNIXの場合　exit()という関数を呼びだすとプロセスの実行が終了する

異常終了　なんらかの原因によってプロセスの実行が継続できなくなり強制的にプロセスを終了する場合
アクセス権限のないメモリ領域を参照した場合など

他のプロセスによる終了　UNIXではkillというコマンドを使用

いずれの場合でも，OSはプロセスが使用していたリソースを解放し，再び使えるようにする

メモリ領域，オープンされていたファイル，割りあてられたCPU時間

Date . . No.

3.6 カーネルの保護

カーネル - OSの中核部分、プロセスが存在する

プロセスを生成したり、メモリの管理を行っている

カーネルプログラムがハッキングされればどうなる？

↳ ユーザーのプロセスのデータをあさみみることができるかもしれない

カーネル内のデータを勝手に読み書きされては困る

CPUの動作モードと特権命令

(プロセッサ)

CPUは特権モード(privileged mode)と非特権モード(execution mode)の2つの動作モードを持っている

カーネルプログラムを実行するときにはCPUは特権モードで動作

通常のプロセスを実行するときは非特権モードで動作

CPUがどちらのモードで動作しているかはCPUの状態レジスタのビットで示されている

特権モードと非特権モードの違い

- 実行できる命令の違い　特権モード　全てのCPU命令可
  非特権モード　一部の命令不可
- アクセスできるメモリ領域の違い

カーネルが存在しているメモリ領域は非特権モードではアクセスできない

No. Date

OSにしかできない仕事（プロセス，ファイル，メモリ）をユーザのプログラムから行いたい場合にどうするか

システムコール という仕組みを使う

fork，open（ファイルオープン），sbrk（メモリ）

→ 内部で"スーパバイザ呼出し"を行う

スーパーバイザ = OS

→ プログラムで明示的におこす割り込み

→ 特権モードに切りかわると同時にカーネル内の関数にジャンプする

（カーネル内のプログラムしか特権モードで動かないことを保持）

多様なシステムコールがサポートされている

100以上

Linux 3.17

FreeBSD 400

Wikipediaより

逆にいえばシステムコールでサポートされていないような動作をユーザがOSに頼むことはできない

どのようなシステムコールを用意するかはOS設計の肝

前ページ続き

例えばどんな命令が特権命令か

（例）割込みベクタを設定する命令

これができると，割込みベクタをユーザが変更し，他のプロセスが行っているI/Oデバイスとの入出力をすべて横取りできてしまう

CPUのマニュアルに特権命令か非特権命令か記されている