スレッドとプロセス
本題: スケジューリング

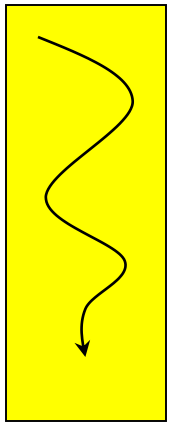
田浦健次朗

スレッド・プロセスの目的

- CPUを仮想化
  - 物理的なCPU数は固定, 少数
    - ラップトップ, スマホ: 1, 2, 4, 8くらい
    - サーバ: 数十
  - ポイント: にもかかわらず数十, 数百のプログラムを立ち上げることができる
  - 個々のプログラムを書く人が明示的な「譲り合い」をする必要はない

スレッドとは?

- 制御の流れ(thread of control)
  - OSに「スレッドを作りたい」と要求
  - OSはスレッドにCPUを割り当て，実行
  - スレッドは「たくさんあってよい」
    - OSが交互に実行
    - CPUが複数あれば各CPU上で



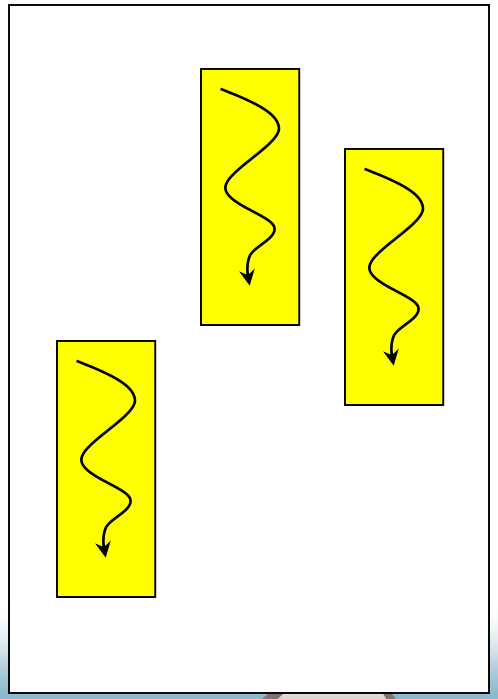
プロセスとは?

- ≈プログラムの起動したときにできるもの

  論理アドレス空間の生成

  +mainスレッドの生成

- プロセス=論理アドレス空間+(1つ以上の)スレッド



実践的知識

- システム内のプロセス・スレッドの観察
  - Linux : ps, top, htop, システムモニタ
  - Windows : タスクマネージャ, perfmon
- 自分のマシンには何個くらいプロセス, スレッドが走っているでしょう?

スレッド・プロセス関係API

- 共通な主要概念
  - 生成，終了
  - 同期，実行の制御
- 代表的スレッドAPI名
  - Unix: POSIX Threads (pthread)
  - Windows: Win32 thread
- 代表的プロセスAPI
  - POSIX
  - Win32

プロセス生成

- Win : CreateProcess(*file, cmd, …, &pid*);
  - *file*をコマンドライン*cmd*で起動
- Unix :

```
pid = fork(); /* プロセス複製 */
if (pid == 0) { /* 子プロセス */
  execv(file, cmdline);  /* fileを実行 */
} else { /* 親プロセス */ }
```

プロセス終了

- Win : ExitProcess(*s*);
- Unix : exit(*s*);
- またはプログラムのmain関数が終了したとき，自動的に終了

プロセス終了待ち

- プロセスpidが終了するのを待つ
- 実例
  - Win : WaitForSingleObject(*pid*, *timeout*)
  - Unix :
    - wait(&*status*),
    - waitpid(*pid*, &*status*, …);

Fork/exec使用例: いわゆる「シェル」の動作

- エラー処理などを除いた, コマンドシェル(bashなど)の疑似コード

```
while (…) {
    show_prompt();
    char ** cmd = read_command();
    int cid = fork();
    if (cid == 0) {
      file = search_PATH(cmd[0]);
      execv(file,  cmd);
    } else {
      waitpid(pid, &status);
    }
}
```

高水準なAPI

- system(*command_string*)
  - *command_string*を実行するプロセスを作り, 終了を待つ (≈ fork + exec + wait)
- popen(*command_string*)
  - *command_string*を実行するプロセスを作り, そのプロセスと通信するチャネル(パイプ)を返す (パイプ作成 + fork + exec)
- すべて最終的にはfork, exec, waitpid, etc. の組み合わせ(Unixの場合)

スレッド生成

- 基本:
  - 命令列が始まるアドレスを指定．そこから実行を開始するスレッドを生成
  - 命令列のアドレス: Cの関数ポインタ
- 実例
  - Pthread : pthread_create(&*id*, .., *f*, *x*)
  - Win : CreateThread(.., .., *f*, *x*, .., &*id*)

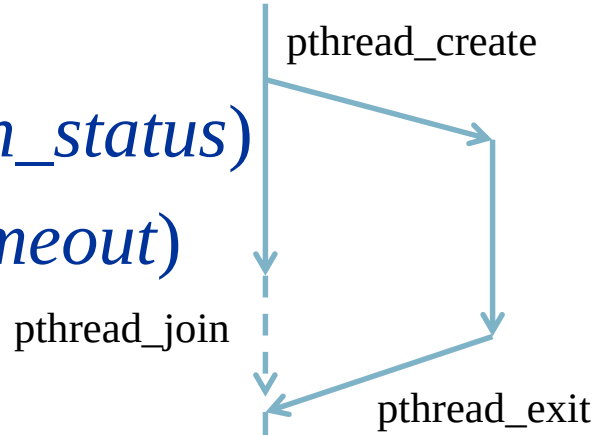
f(*x*) を実行するスレッドを生成．スレッド名を*id*に格納

スレッド終了

- スレッドの終了
- 実例
  - Pthread : pthread_exit(*s*)
  - Win : ExitThread(*s*)
  - または生成時に指定された関数*f*が終了すると自動的にスレッドが終了する
  - *s* : status

スレッドの終了待ち

- スレッドidが終了するのを待つ(idはpthread_createなどによって得られたスレッド名)
- 実例
  - Pthread: pthread_join(*id, &return_status*)
  - Win: WaitForSingleObject(*id, timeout*)
- その他の同期方法は後述



そのほかの言語

- ほとんどの言語(Java, C++, Python, etc.)でスレッド, プロセス関係のAPIが提供されている
  - (名前は違うが)概念は似ている
  - 多くの場合C用のライブラリ, それが最終的には fork, exec, etc.を呼び出している(Unixの場合)

プロセスとスレッド

- スレッドとプロセスは似ているようだが何が違う?
  - 正確な違いは数週間後, メモリについて語るときにわかる
- 現時点では,
  - スレッド: 一筋の実行の流れ
  - プロセス:「プログラムを起動したときにできるもの」
  - プロセス =「箱」＋ 1個以上のスレッド
    - mainを実行するスレッドはプロセスとともに自動的に作られる

スケジューリング

- システム内には多数のスレッドが同時に存在する
  - ほとんどの時点で CPU数 < スレッド数
- OSはそれらに「適切に」CPUを割り当てる(スケジューリング)必要がある
  - 基本: 変わりばんこ

スレッドスケジューリングの目標

- 公平性
  - 独占の禁止
  - 公平なCPUの割り当て
- 効率性
  - ひとつでも実行可能なスレッドがいる限りCPUを「無駄に」しない
  - スレッド切り替えのオーバーヘッドを少なくする
- 対話的プログラムの応答性

スケジューラの挙動観察実験(1)

- いくつかのスレッド(プロセス)を立ち上げる
- スレッドがいつからいつまで「実行中だったか」を調べる
  - 「時刻を知るシステムコール」
- 注目
  - どのくらいの頻度でスレッドは入れ替わっているか?
  - その状態でエディタやブラウザの応答性に影響は?

スケジューラの挙動観察実験(2)

- 各スレッド

```
f (x) {
  t =現在時刻();
  while (1) {
    t’ = 現在時刻();
    if (t’ – t > 1ms) {
      どうやらしばらくCPUを奪われていたらしい;
    }
    t = t’;
  }
  記録を出力;
}
```

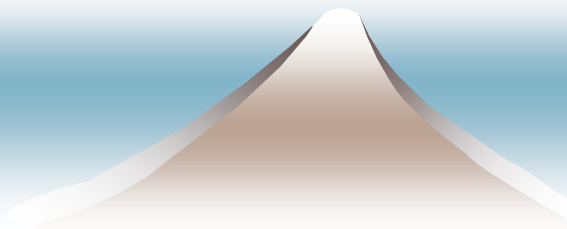
ヒント: 時刻を知るシステムコール

- UNIX : gettimeofday
  - man gettimeofday
- Win : QueryPerformanceCounter
  - MSDNライブラリで検索
- 授業のHPに例題掲載

特定のCPUへスレッドを固定する

- システムコール
  - `sched_setaffinity` (*pid*, *size*, *mask*)
    - mask : size bitのbit列
    - maskで1となっているCPUでのみ, pidを実行
  - 子プロセスへ自動的に継承
- コマンド
  - `taskset` –c 0,1 *command*

スレッドスケジューラの実現



クイズ

- あなたが無限ループするプログラムを書いてもマウスが動き, ブラウザが動き, (大概の場合)Ctrl-Cで消せるのはなぜですか?
  - (a) マウスはプログラムで動いているわけではないから
  - (b) マシンにCPUコアが2個(以上)あるから(ブラウザと無限ループは別のコアで動いている)
  - (c) 実は無限ループであってもC言語処理系が時々他のスレッドに譲るような機械語に変換しているから
  - (d) その他
- https://forms.gle/8vTtuSr43R6FGg4g8

復習: OSのないCPU

- CPUの状態 : レジスタ
  - 汎用
  - プログラムカウンタ(PC)
  - モード(ユーザモード・特権モード)
  - etc.
- CPUの動作:

  以下を永遠に繰り返す {
  外部割込みが発生していれば割り込み処理
  PCが指す場所の命令を取り出す
  命令を実行(状態書き換え)
  }

もし割り込みがなかったら?

- 以下のようなプログラム(無限ループ)がCPUを独占するのを防ぐ方法はない
  - C:

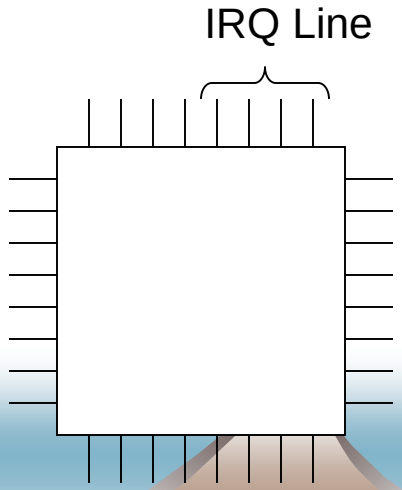
    ```
    for (;;) {
    }
    ```
  - 機械語:

    ```
    L:
      jmp L
    ```

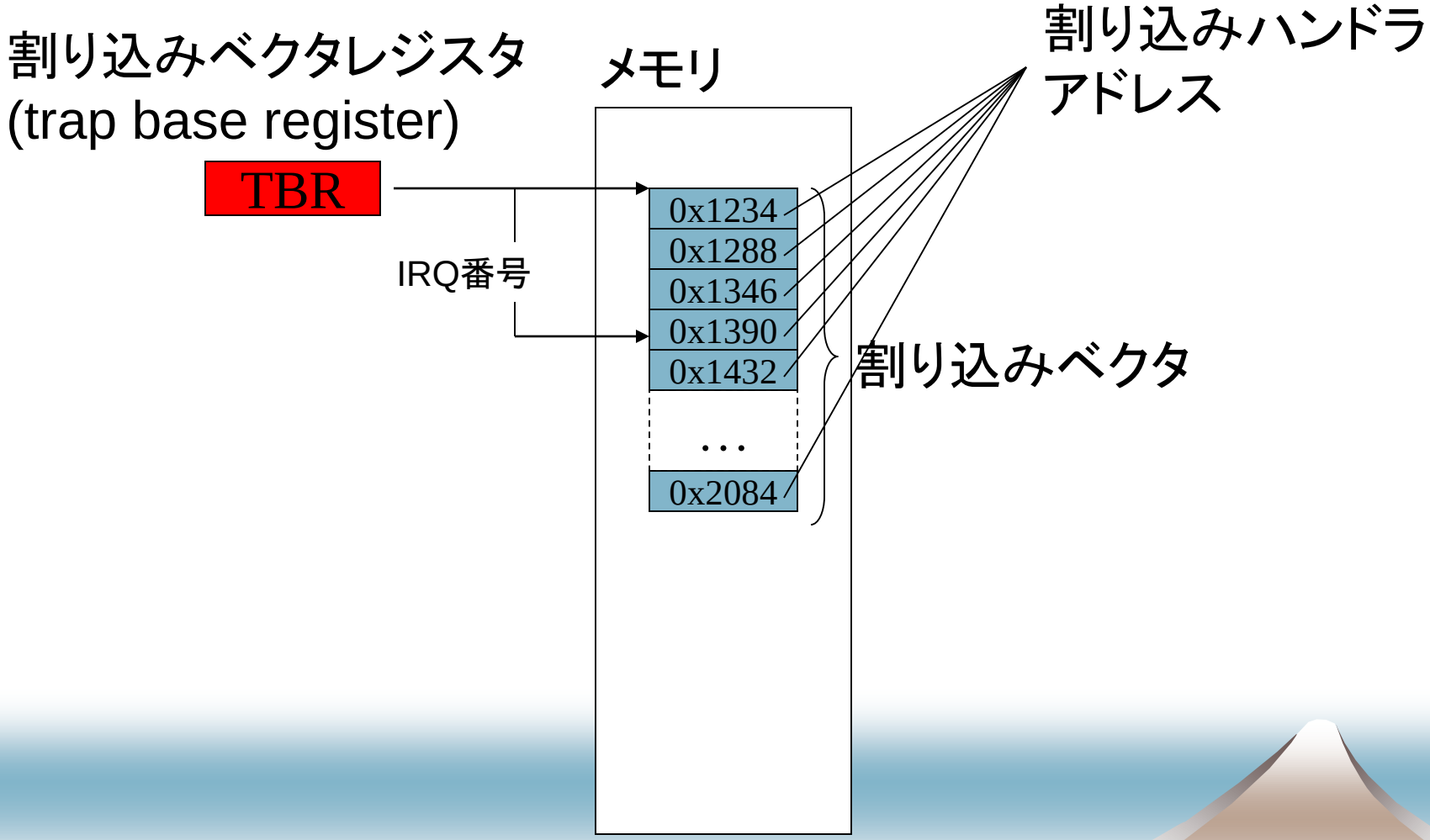
割り込み処理

- 割り込み: CPU外部からの信号
- 割り込み時にCPUが行うこと

```
if (その割り込み許可中) {
    割り込みを禁止にする;
    一部の状態(割り込み発生時PCなど)
      を特定のレジスタに保存;
    割り込みベクタを参照し，指定されてい
      る値をPCに設定(制御の移動);
    特権モードへ移行;
  }
```



割り込みベクタ


割り込みベクタレジスタ
(trap base register)
TBR
IRQ番号
メモリ
0x1234
0x1288
0x1346
0x1390
0x1432
...
0x2084
割り込みハンドラ
アドレス
割り込みベクタ

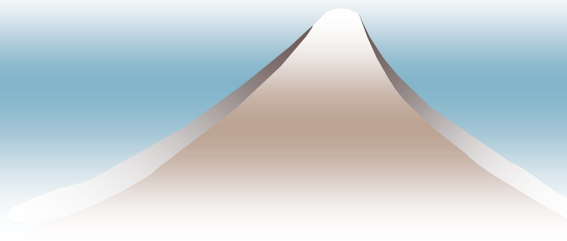
割り込みベクタ

- 通常OSの起動時に設定される
- IRQ通常の用途
  - 各種入出力装置からの通知
    - キーボード，ネットワークコントローラ，etc.
    - タイマ!!
  - システムコール(トラップ命令)

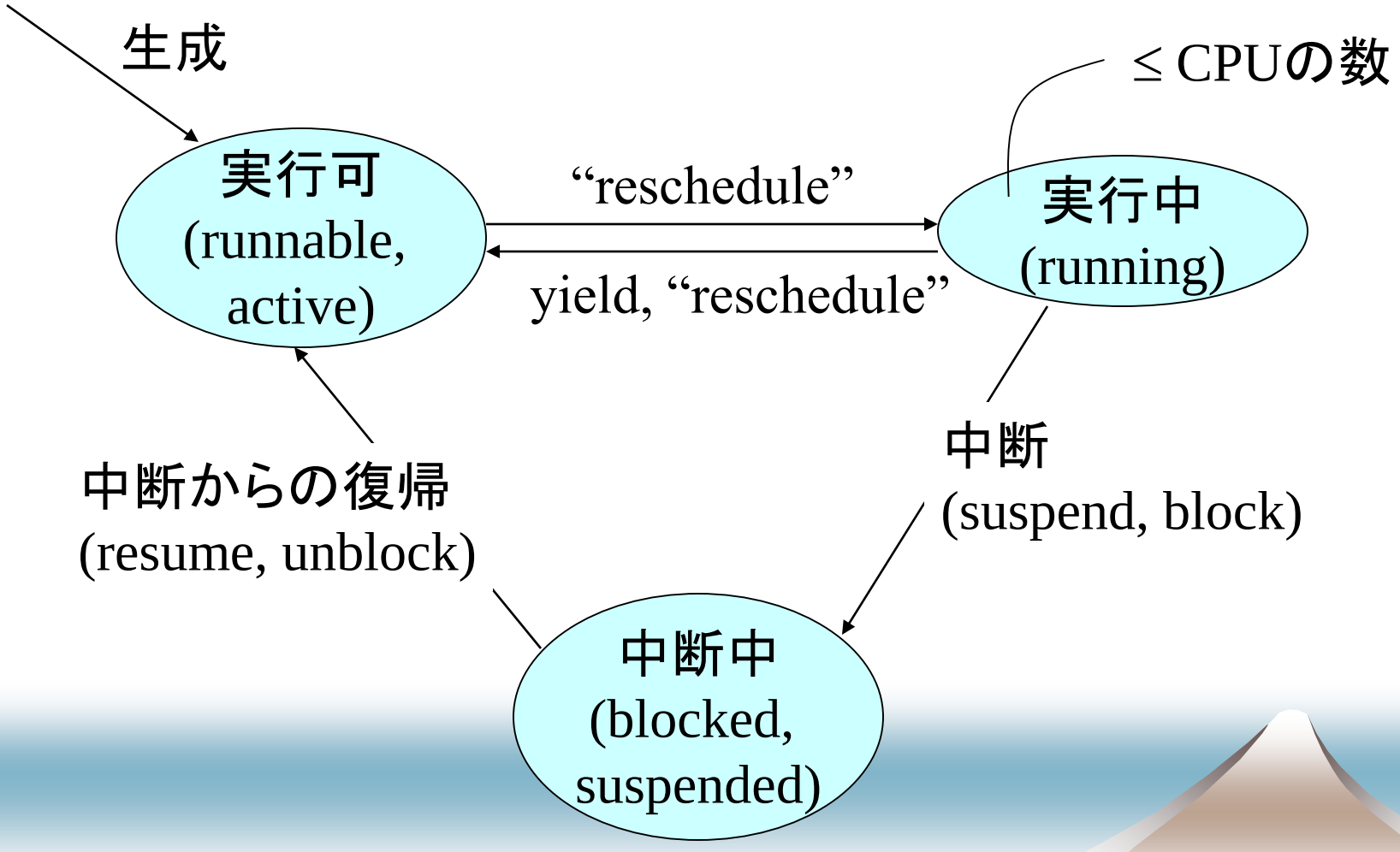
タイマ割り込み

- CPUの独占を防ぐための鍵
- 通常定期的(1- 10ms 程度に一度程度)発生させる
  - Linux (2.4) on x86, Windows on x86, BSDなどで10ms
  - Linux on Alpha 1ms
  - Linux 2.6.22以前 1ms, 4ms, または必要に応じて (tickless)
- タイマ割り込み間隔，クロック間隔，クロックチックなどと呼ぶ
- 必要な時にOSがCPUをアプリケーションから奪う「機会」を保障する

スケジューリングアルゴリズム



各スレッドの状態


生成
≤CPUの数
実行可
(runnable,
active)
“reschedule”
yield, “reschedule”
実行中
(running)
中断からの復帰
(resume, unblock)
中断
(suspend, block)
中断中
(blocked,
suspended)

中断状態

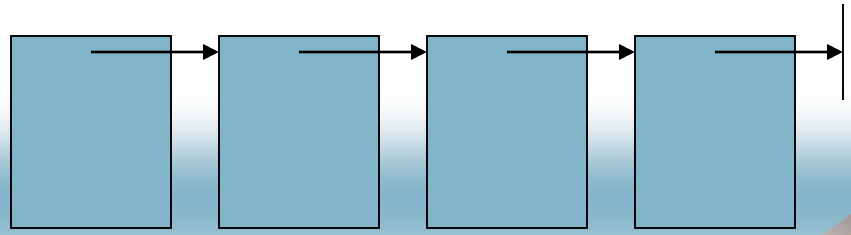
- (たとえCPUがあいていても)直ちに実行をすることができない状態
- OSは中断状態のスレッドを(当然)CPU割り当ての対象から外す

中断と復帰

- スレッドが中断する理由(代表例)
  - 入出力待ち(recv, read, etc.)
  - 自主的休眠(sleep)
  - スレッド間の同期(pthread_join, wait, etc.)
  - ページフォルト(後述)
- 復帰の理由(中断の逆)
  - 入出力完了
  - 休眠時間経過
  - スレッド間の同期成立
  - ページフォルト処理完了

実行可能キュー

- 実行可能スレッドのリスト
  - 「スケジューリングキュー」,「ランキュー」
  - スレッドが実行可能キューにある ⇔ そのスレッドが実行可能
- OSは「機会あるごとに」，実行可能キューから最も適切なスレッドを選んで実行(reschedule)



中断

- 例: ネットワークからのデータ待ちによる中断
- recv() {

```
…
if (読むべきdataがない) {
  現スレッドを実行可能キューからはずす;
  reschedule();
}
…
}
```

中断からの復帰

- 例: ネットワークからのデータ到着による復帰
- /* 割り込み →
  OS内のネットワークからの入力を
  処理する部分 */

```
if (あるスレッドが今到着したデータ待ち) {
  そのスレッドを実行可能キューに入れる;
  reschedule();
}
```

Rescheduleの機会

- OSカーネルが制御を得るあらゆる時点が，潜在的なrescheduleの機会
  - タイマ割り込み時
    - クロック間隔に一度
  - そのほかの割り込み(e.g, 入力)からの復帰時
  - 実行中スレッドが中断したとき
  - システムコールからの復帰時
  - etc.

Reschedule

- Rescheduleの機会をOSが得るたびに，「次に実行すべきスレッド」を選択して実行(CPUの割り当て)
- 次に実行すべきスレッドの選択の仕方が重要
  - 公平
  - 効率
  - 対話的プログラムの応答

対話的なスレッドの応答性

- エディタ，パワーポイント，ブラウザ，メーラに「キー入力」したらすぐに反応してほしい
- 対話的なスレッドの特徴
  - I/Oによる中断が非常に頻繁
  - キーボード，マウス，ネットワークなどの入力待ち
  - 中断が多いため，CPU利用量は少ない

現在のLinuxスケジューラ

- カーネル2.6.23以降. Completely Fair Scheduler (CFS)
- 各スレッドが使用したCPU時間を管理
  - vruntime (virtual runtime)
- Reschedule時には毎回, vruntimeが最小のスレッドを選ぶ
  - ≈ つまり最もCPUを使っていないスレッドを選ぶ
  - ⇒ 公平性の保証

vruntime管理の実際

- スレッドが生まれたとき
  - 子は親のvruntimeを引き継ぐ
  - 子のvruntime = 0というわけにはいかない
- スレッドがA → Bに切り替わるとき
  - Aのvruntime += 今回消費した時間
  - Bのvruntime = 基本はそのまま. ただし, それだけだと問題がある

vruntime管理の実際

- A→Bに切り替え時
  - Bのvruntime 何もしないが, ≥ 全スレッドのvruntimeの最小値 – 20 ms を保証
  - つまり長時間中断していてもその分をまるごと「貯金」はできない