

# スケジューリング

田浦健次郎

# CPUスケジューリング

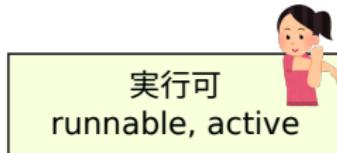
- ▶ OSの重要な仕事：スレッドにCPU<sup>1</sup>を割り当てる
- ▶ 基本：かわりばんこ (round robin)
- ▶ ただし、文字通り存在するすべてのスレッド (> 1,000 個は当たり前) に均等時間ずつ、ではない

---

<sup>1</sup> 「仮想コア」が正確な言い方だが

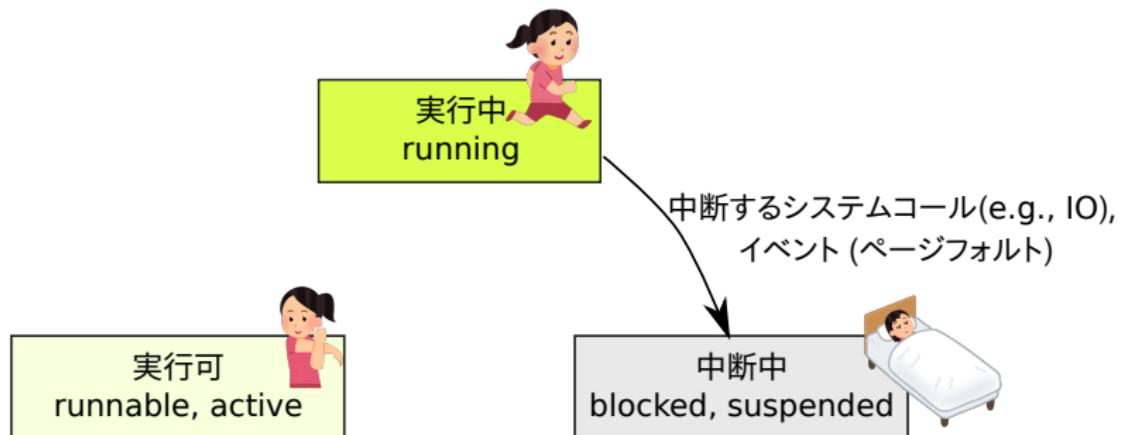
# スレッドの状態

- ▶ OSは以下の3つの状態を区別している
  - ▶ 実行中 (running)
  - ▶ 実行可能 (runnable, active)
  - ▶ 中断中 (blocked, suspended)



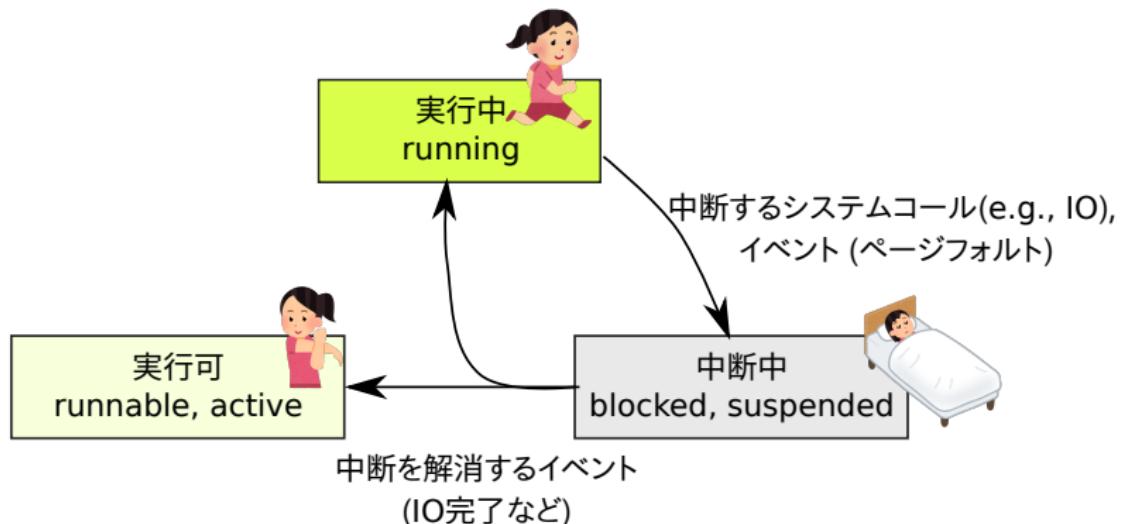
# スレッドの状態

- ▶ OSは以下の3つの状態を区別している
  - ▶ 実行中 (running)
  - ▶ 実行可能 (runnable, active)
  - ▶ 中断中 (blocked, suspended)



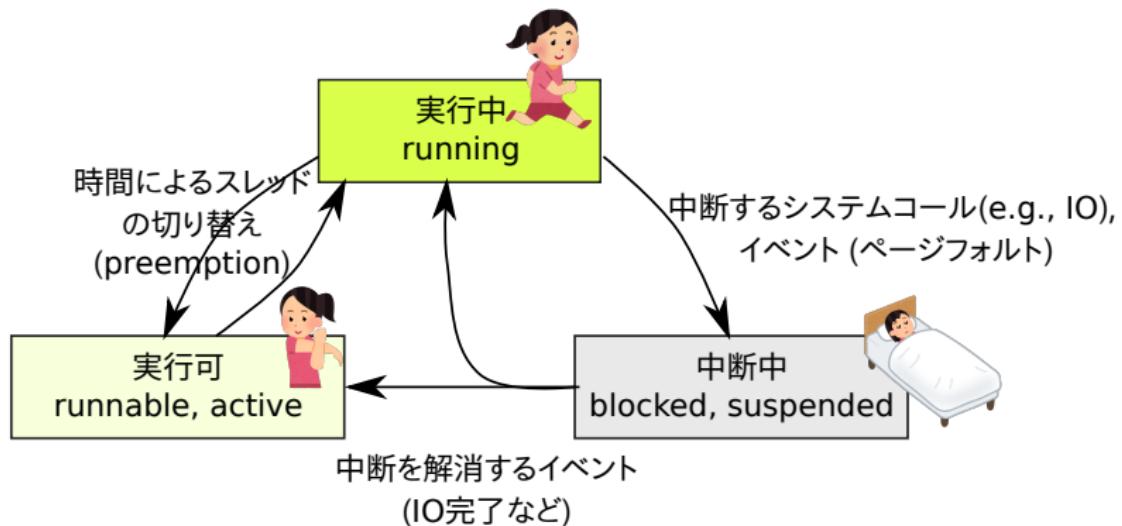
# スレッドの状態

- ▶ OSは以下の3つの状態を区別している
  - ▶ 実行中 (running)
  - ▶ 実行可能 (runnable, active)
  - ▶ 中断中 (blocked, suspended)



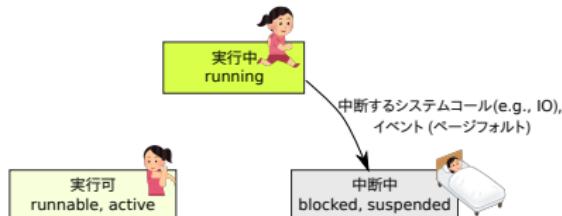
# スレッドの状態

- ▶ OSは以下の3つの状態を区別している
  - ▶ 実行中 (running)
  - ▶ 実行可能 (runnable, active)
  - ▶ 中断中 (blocked, suspended)



# スレッドが中断する時の例

- ▶ OSが、現在実行中のスレッドがこれ以上実行不能と判断する時
  - ▶ `read`, `recv` などブロッキングI/Oで読むデータがない
  - ▶ `waitpid`, `pthread_join` などで、子プロセス・スレッドが終了していない
  - ▶ `pthread_mutex_lock`, `pthread_cond_wait` など同期APIで同期が成立していない（次週以降）
  - ▶ `sleep`, `usleep`, `nanosleep` など休眠API
  - ▶ ページフォルト（次週以降）でI/Oが発生
- ▶ OSは、現在実行中のスレッドから、別のスレッドに切り替える（コンテクストスイッチ）



# ほとんどのスレッドは中断している

- ▶ ps auxww, top コマンドの STAT 欄
  - ▶ R 実行中または実行可
  - ▶ S 中断中 (割り込み可能)
  - ▶ D 中断中 (割り込み不可能)
  - ▶ その他 ...
- ▶ S と D の違いはあまり気にしなくて良い. D はそれほど見かけない
- ▶ まめ知識: Unix の load average (負荷平均) = R か D 状態にあるスレッド数  $\approx$  R 状態にあるスレッド数 (の最近何分かの平均)

# スレッドはどこで中断しているかの観察

- ▶ デバッガ (gdb) で実行中のプロセスをデバッグ可能 (attach)

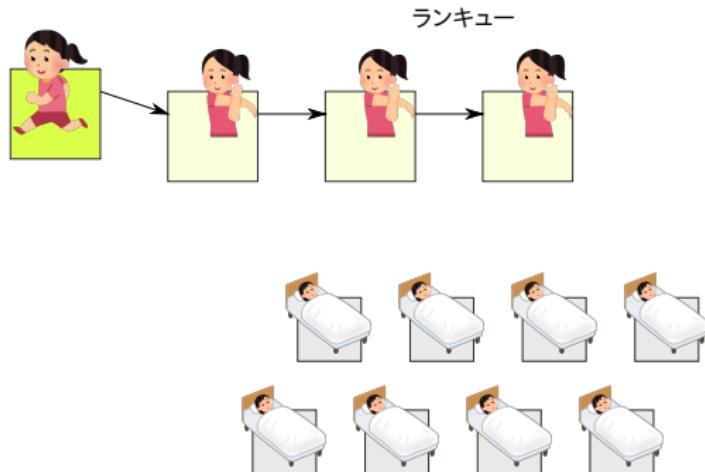
```
1 $ gdb  
2 (gdb) attach pid
```

- ▶ 典型

- ▶ poll (複数のファイルディスクリプタからの入力待ち)
- ▶ read

# タスクキュー, ランキュー

- ▶ 実行中・実行可状態のスレッドを維持する
  - ▶ 「キュー」とは言うが FIFO とは限らない
- ▶ OS はスレッドを切り替える際, ランキュー中から, [ある基準\(後述\)](#) に従って次のスレッドを選んで実行する



# Preemption (横取り)

- ▶ スレッドが自発的に OS に制御を渡さない (なにもシステムコールを呼ばずに走り続けている) 状態でも、強制的に制御を奪うこと ( $\equiv$  **preemption**)
- ▶ **preemption** を行うスケジューラ  $\equiv$  **preemptive** なスケジューラ
- ▶ 今日の OS のスケジューラは事実上すべてが **preemptive**
- ▶ **preemption** を実現する仕組み: タイマ割り込み

# 割り込み, タイマ割り込み

生の CPU (仮想コア): 以下を繰り返す

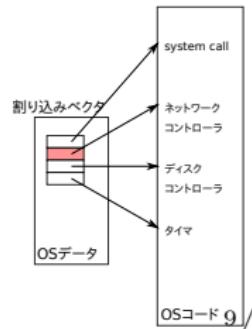
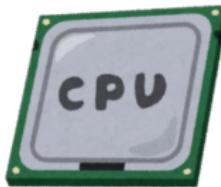
```
while (1) {
```

  PC の指すアドレスから命令を読む

  命令を実行する (一部のレジスタが書き換わる, PC 含め)

```
}
```

注: PC = プログラムカウンタレジスタ



# 割り込み, タイマ割り込み

生の CPU (仮想コア): 以下を繰り返す

```
while (1) {
```

  PC の指すアドレスから命令を読む

  命令を実行する (一部のレジスタが書き換わる, PC 含め)

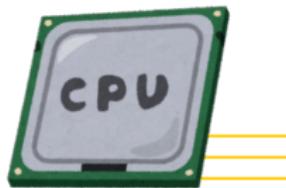
```
  if (割り込みが来た) {
```

    PC := 割り込みベクタ [割り込み番号];

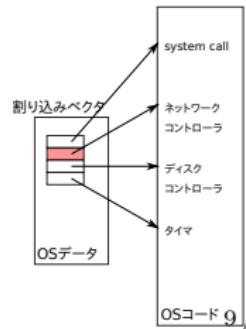
```
  }
```

```
}
```

注: PC = プログラムカウンタレジスタ



} 割り込み線; Interrupt Request (IRQ) Line



# 割り込み, タイマ割り込み

生の CPU (仮想コア): 以下を繰り返す

```
while (1) {
```

  PC の指すアドレスから命令を読む

  命令を実行する (一部のレジスタが書き換わる, PC 含め)

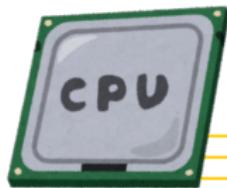
```
  if (割り込みが来た) {
```

    PC := 割り込みベクタ [割り込み番号];

```
  }
```

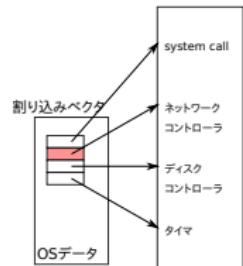
```
}
```

注: PC = プログラムカウンタレジスタ



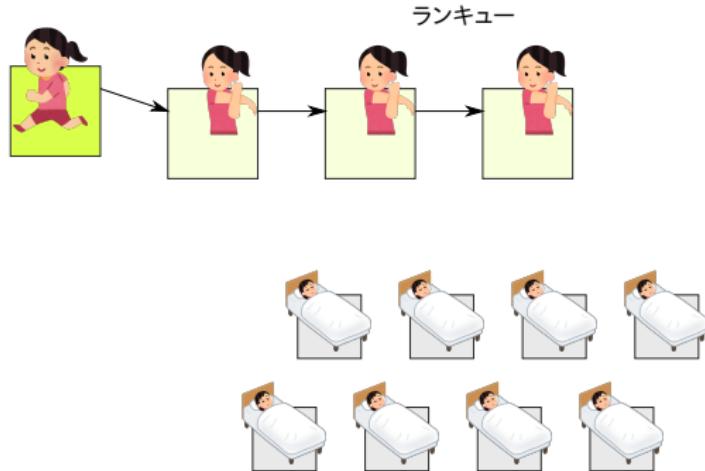
タイマー (Programmable Interval Timer)

} 割り込み線; Interrupt Request (IRQ) Line



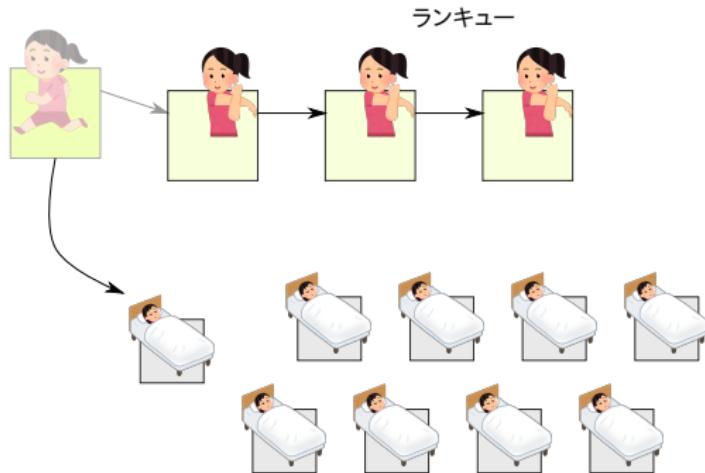
# ランキューの動き

ランキューには実行可・実行中のスレッドが入っている



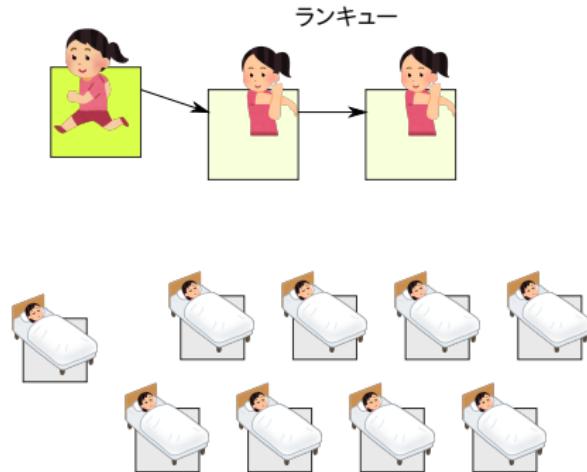
# ランキューの動き

実行中のスレッドが中断 → ランキューから外れる



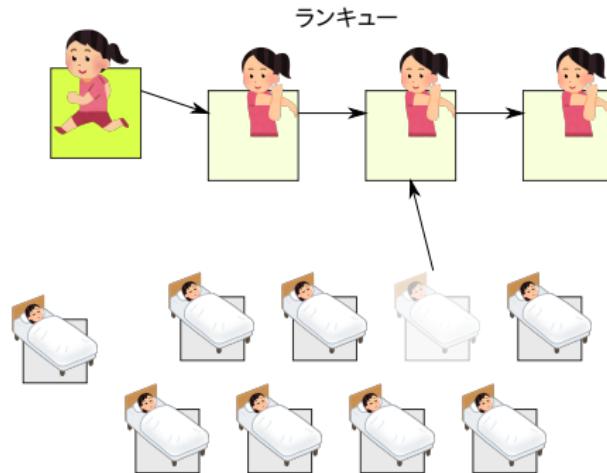
# ランキューの動き

ランキューから次のスレッドが選ばれて実行される



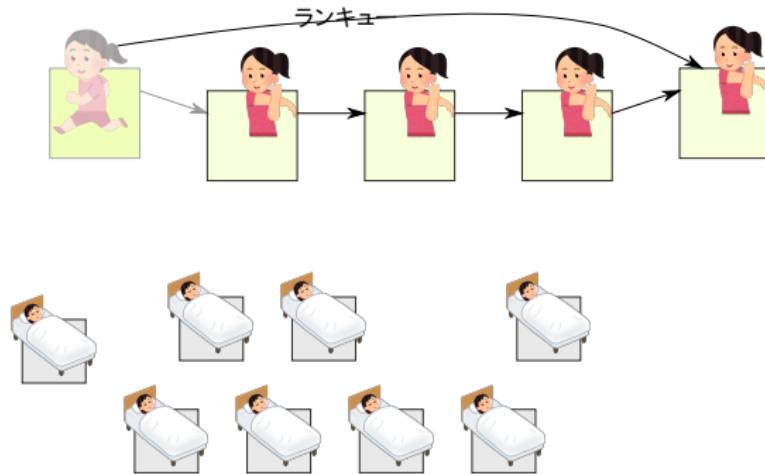
# ランキューの動き

中断中のスレッドが再開 → ランキューに挿入される



# ランキューの動き

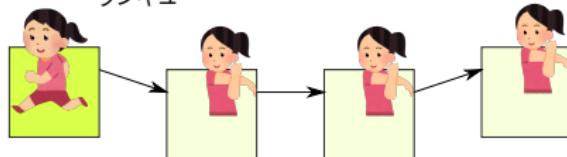
タイマ割り込み → 実行中のスレッドが十分な時間を消費していたら, preemption



# ランキューの動き

ランキューから次のスレッドが選ばれて実行される

ランキュー



# スレッド選択

- ▶ OSはスレッドを切り替える際、ランキュー中から、ある基準に従って次のスレッドを選んで実行する
- ▶ ある基準の例
  - ▶ Round Robin (純粹なかわりばんこ)
  - ▶ Linux Completely Fair Scheduler (CFS)

# Round Robin

- ▶ ランキューが純粋な FIFO
- ▶ 中断から回復したらキューの末尾に入る
- ▶ 実行中のスレッドの time quantum が expire したら末尾に入る
- ▶ 次のスレッドを選ぶ際はキューの先頭が選ばれる

# Round Robin の問題点

- ▶ 公平性: 「中断していた  $\iff$  CPU を使っていなかつた」スレッドも、「preempted された  $\iff$  ずっと CPU を使っていなかった」スレッドも同じ扱い
- ▶ 対話的なプログラムの応答性: 実行可のスレッドが多い (load average が高い) と, それに比例して, 「応答時間 = 中断状態から再開してから実行されるまでの時間」が長くなる

# Linux Completely Fair Scheduler (CFS)

- ▶ Linux 2.6.23 以降のデフォルトスケジューラ
  - ▶ 各スレッドが「消費した合計 CPU 時間 ( $\rightarrow$  vruntime)」を管理
  - ▶ スレッド切り替え (実行中スレッドが中断した, 中断中スレッドが復帰した, タイマ割り込みがおきた, など) 時に, 実行可能スレッドの中で vruntime が最小のスレッドを次に実行する
  - ▶ 注: ランキューを, vruntime の小さい順にスレッドが並ぶ優先度キューとすれば実現可能
- ▶ 効果
  - ▶ 長い目で見て, 各スレッドへの CPU 割当時間を均等にすることができます → 公平
  - ▶ しばらく中断していた (CPU を使っていなかった) スレッドが再開した時, その間実行されていたスレッドよりも vruntime が小さいことが期待される  
→ 対話的スレッドの応答性が高い

# vruntime

- ▶ 文字通り、「vruntime = 割り当てた CPU 時間の合計」とするのは問題がある
  - ▶ 作られたばかりのスレッドの vruntime = 0 ⇒ すでに vruntime=10 秒のスレッドは 10 秒間待たされる?
  - ▶ 中断していたスレッドの vruntime は一切増えない ⇒ 10 秒間中断してから再開した時、その 10 秒間ずっと走っていたスレッドは 10 秒間待たされる?
- ▶ 「公平に CPU を割当る」と言っても、「長時間に渡った合計が均等」が目標ではない(長時間中断していてもその分を無制限に「貯金」されては困る)

# vruntime 管理の実際

- 生成時は親の vruntime を継承. 親スレッド  $A$  が子スレッド  $B$  を生成

$$B.\text{vruntime} = A.\text{vruntime}$$

- タイマ割り込み, スレッド中断時に, 実行中だったスレッド  $T$  の vruntime を加算.

$T.\text{vruntime} += T$  が今回消費した CPU 時間

- スレッド  $T$  が中断から復帰する時は,  $T$  の vruntime が他の実行可・中スレッドよりも極端に小さくならないことを保証

$$T.\text{vruntime} = \max(T.\text{vruntime}, \min_{t: \text{実行可}} t.\text{vruntime} - 20 \text{ ms})$$