45分でゼロから作る！
OS自作ライブコーディング

怒田 晟也

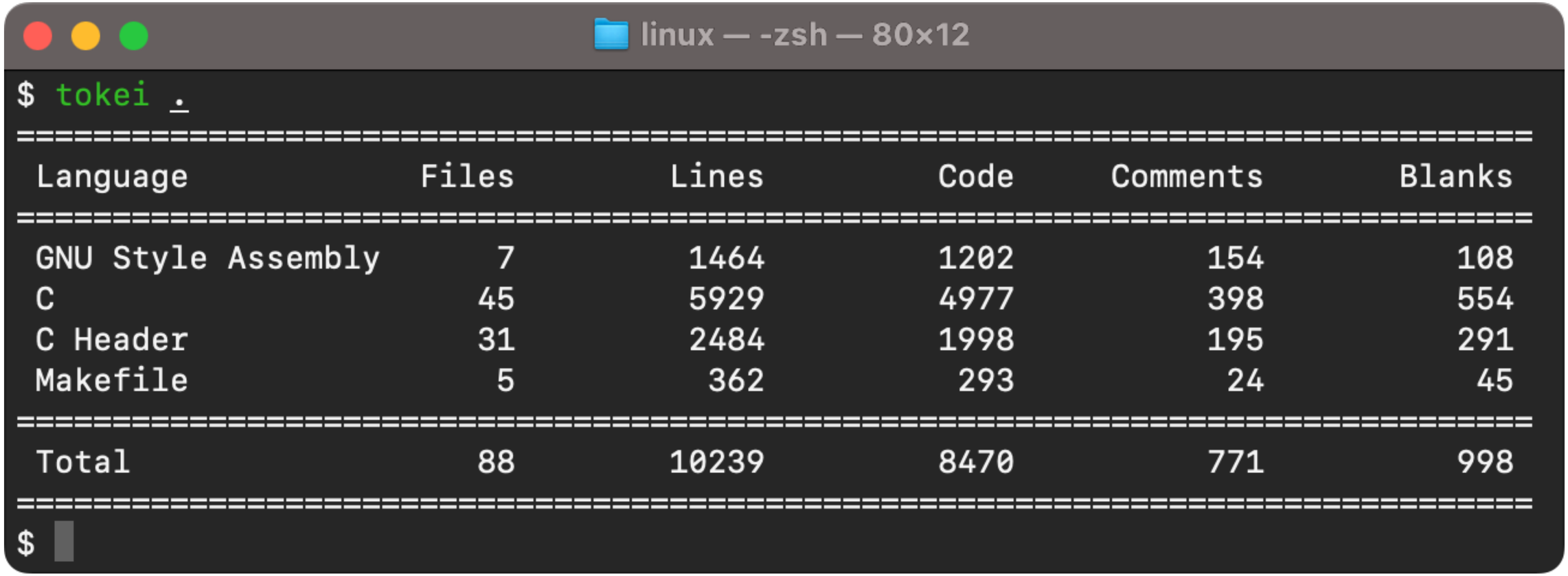
<nuta@seiya.me>

OSC2023 Online/Kyoto

主題

OSのような複雑なソフトウェアも案外簡単に作れる

Linuxも最初は小さかった



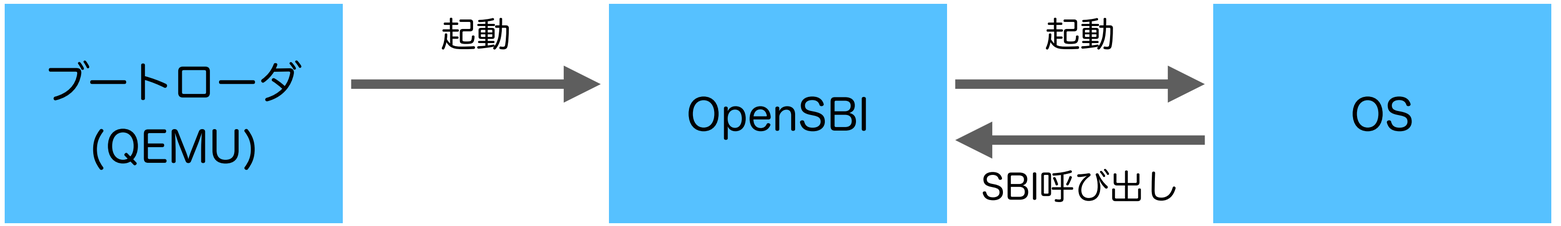
本セミナーの趣旨

- 新しいオープンソース・ソフトウェアを45分で作ろう！
- ブート処理からコンテキストスイッチまで実装
- 理解するより実装の雰囲気を楽しむ内容です
  - 「よく分からんけど自分でもできそう」と感じてもらえれば嬉しい
- しっかり理解して作りたい方用にオンラインブックを公開しています (後述)

ルール

- QEMU (エミュレータ) の仮想的なマシン (virt) に対応
- RISC-V 32ビットCPU
- 標準ライブラリを使わずにゼロからC言語で実装
- 些細な部分はGitHub Copilotで自動生成
- とりあえず動くものを作る (エッジケース・未定義動作は無視)

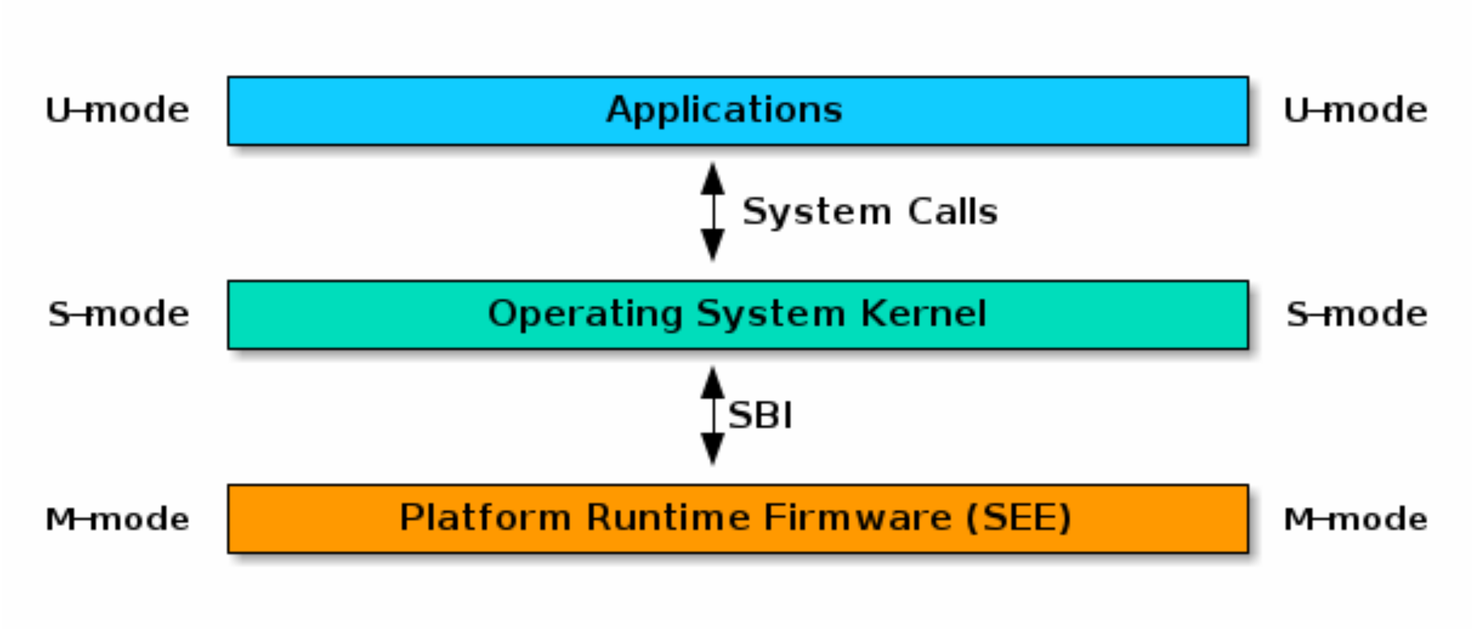
ステップ1: ブート処理



1. QEMUがOSの起動イメージ・OpenSBIをメモリ上に展開
2. OpenSBIがCPUの最低限の初期化処理をしてOSを起動
3. OSはSBI (APIみたいなもの) を活用しながら起動

（マシンによってブート周りは大きく異なる）

ステップ2: SBI呼び出し



RISC-V SBI specification (CC BY 4.0) - https://github.com/riscv-non-isa/riscv-sbi-doc

ステップ2: SBI呼び出し

要約: a7レジスタに拡張番号、a0-a5レジスタに引数をセットしてECALL命令を使う

Chapter 3. Binary Encoding

All SBI functions share a single binary encoding, which facilitates the mixing of SBI extensions. The SBI specification follows the below calling convention.

- An `ECALL` is used as the control transfer instruction between the supervisor and the SEE.
- `a7` encodes the SBI extension ID (**EID**),
- `a6` encodes the SBI function ID (**FID**) for a given extension ID encoded in `a7` for any SBI extension defined in or after SBI v0.2.
- All registers except `a0` & `a1` must be preserved across an SBI call by the callee.
- SBI functions must return a pair of values in `a0` and `a1`, with `a0` returning an error code. This is analogous to returning the C structure

```
struct sbiret {
    long error;
    long value;
};
```

In the name of compatibility, SBI extension IDs (**EIDs**) and SBI function IDs (**FIDs**) are encoded as signed 32-bit integers. When passed in registers these follow the standard above calling convention rules.

ステップ3: トラップ

```
int counter;

void main (void) {

    puts("Hello World!");

    counter++;

    for (;;);

}
```

文字列出力システムコール

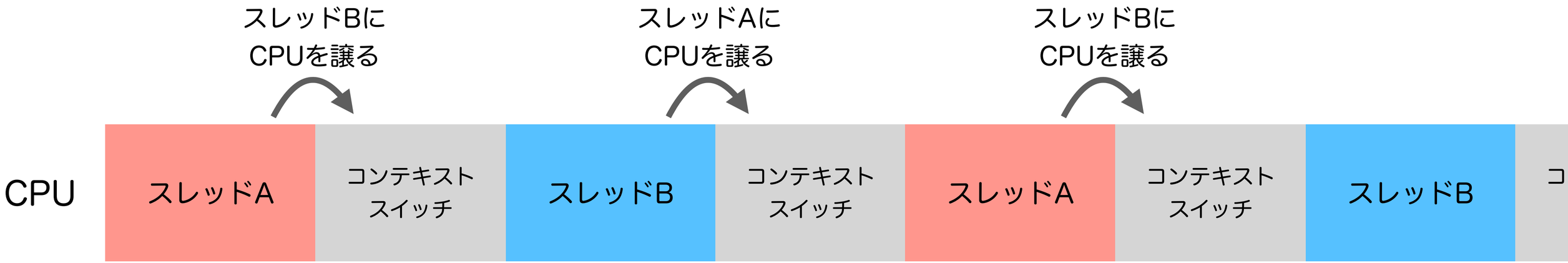
ページフォルト (遅延メモリ割り当てをする)

タイマー割り込み (他のプロセスに切り替える)

(↑は一例)

- CPUはOSの介入が必要なときに、OSの「トラップハンドラ」に処理を移す
  - 例）システムコール、割り込み (ハードウェアからの通知)、無効な機械語、
    ページフォルト (ヌルポインタ参照) …
- 今回は例外が発生したことを表示するだけ (いわゆる「カーネルパニック」)

ステップ4: コンテキストスイッチ



- プログラムの実行状態の保存・復元をする仕組み
- マルチタスク機能の基盤技術

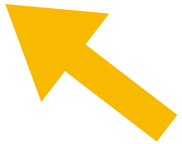
ステップ4: コンテキストスイッチ

CPU

- プログラムカウンタ
- スタックポインタ
- その他のレジスタ

① 各レジスタを
スタックに保存

② 各レジスタを
スタックから復元



実行中スレッド

- スタックポインタ
- 専用スタック領域
(ローカル変数など)

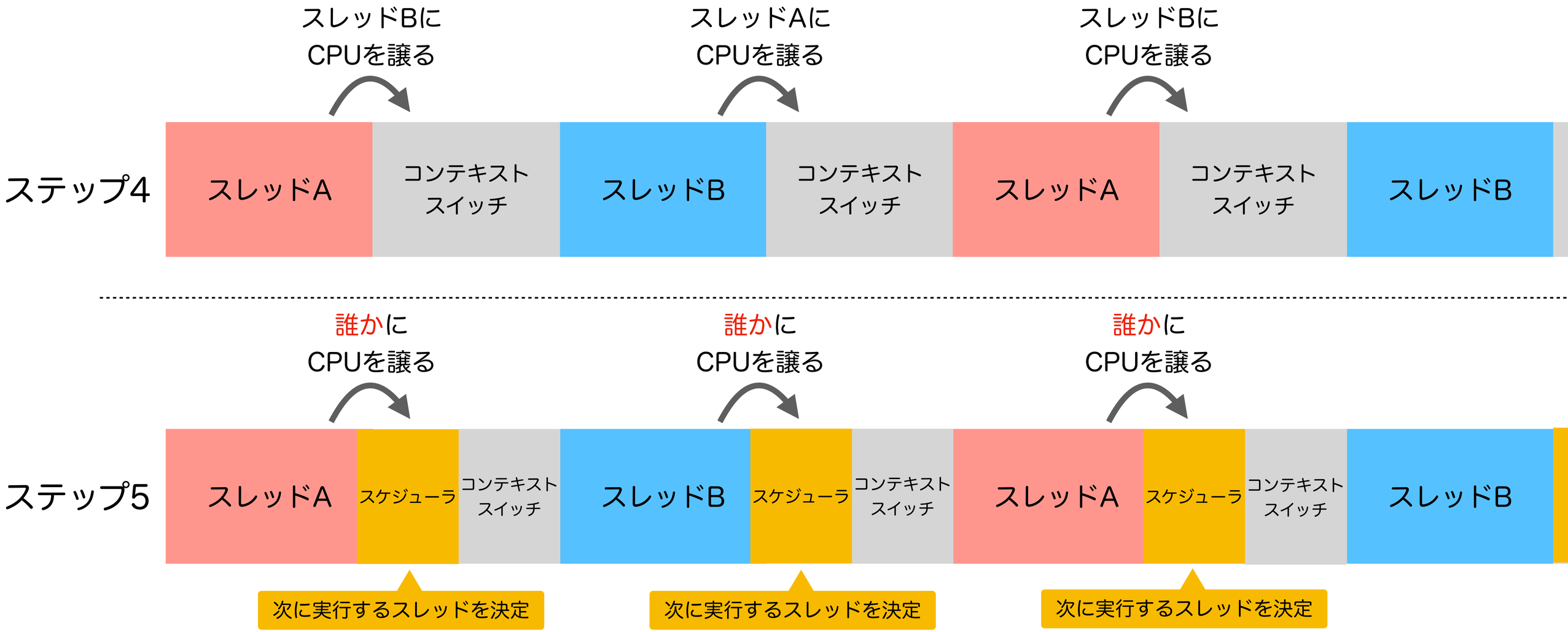
次に実行するスレッド

- スタックポインタ
- 専用スタック領域
(ローカル変数など)

スレッド間共通

- 静的変数 (グローバル変数)
- プログラムの機械語

おまけ: スケジューラ


スレッドBに
CPUを譲る
スレッドAに
CPUを譲る
スレッドBに
CPUを譲る
ステップ4
スレッドA
コンテキスト
スイッチ
スレッドB
コンテキスト
スイッチ
スレッドA
コンテキスト
スイッチ
スレッドB
誰かに
CPUを譲る
誰かに
CPUを譲る
誰かに
CPUを譲る
ステップ5
スレッドA
スケジューラ
コンテキスト
スイッチ
スレッドB
スケジューラ
コンテキスト
スイッチ
スレッドA
スケジューラ
コンテキスト
スイッチ
スレッドB
次に実行するスレッドを決定
次に実行するスレッドを決定
次に実行するスレッドを決定

さいごに

- 続きはオンラインブックで: https://bit.ly/1000-lines
  - 1,000行でアプリケーションやファイルシステムまで実装！
- OSの面白い設計・実装に興味があれば拙作もおすすめ →
  （メディアスポンサー枠宣伝）
- 巨大で複雑なソフトウェアも、シンプルな改善の積み重ね