システムプログラム概論

スレッド

第3講: 平成20年10月9日 (木) 1限 S1教室

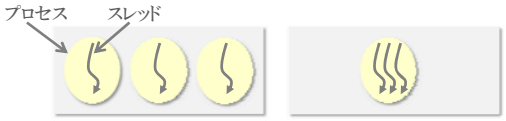
中村 嘉隆(なかむら よしたか)
奈良先端科学技術大学院大学 助教
y-nakamr@is.naist.jp
http://narayama.naist.jp/~y-nakamr/

今日の講義概要

- スレッドのモデルと用途
- ユーザスレッドとカーネルスレッド
- スレッドの管理
- 共有資源に対する競合状態の回避法

プロセスとスレッドの違い

- プロセス
  - 資源(メモリ等)割り当ての単位
- スレッド
  - 実行(プロセッサ割り当て)の単位
  - あるプロセス内のスレッド群は同じ資源を共有
  - 並行処理を表現する一手法



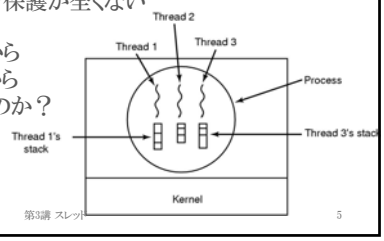
プロセスとスレッドの違い(続き)

- プロセス群が共有する資源
  - 物理的なメモリ, ディスク, プリンタなど
- スレッド群(同じプロセス内)が共有する資源
  - アドレス空間, 開いたファイルの状態など
    - 同じプロセスの別のスレッドが開いたファイルを読み書き可能
  - プロセスが持つ特性も幾つか併せ持つ → 軽量プロセスとも呼ぶ
- プロセス毎, スレッド毎に割り当てられる資源

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

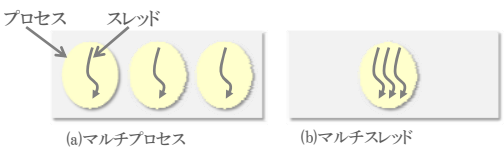
スレッドモデル

- 全スレッドはアドレス空間(グローバル変数も)を共有
- スタック(プログラムカウンタや局所変数を保存する領域)はスレッド毎に用意される
- あるスレッドが別のスレッドのスタックに書き込み, 完全に破壊さえすることができる
- すなわち, スレッド間で保護が全くない
  - 理由は?
    - → (1) 不可能だから
    - → (2) 必要ないから
- なぜスレッドは必要なのか?



プロセスかスレッドか

- 適切な使用状況
  - (a) は, 3 つのプロセスが本質的に関係を持たないときに使用
  - (b) は, 3 つのスレッドが密接に協力しており, それぞれが同じ仕事の一部である時に使用

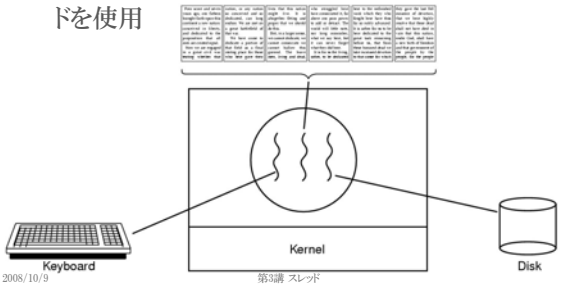


プロセスかスレッドか（続き）

- スレッドはプロセスに比べ生成・消去が容易
  - スレッドは付随するリソースをほとんど持たない
  - スレッド生成はプロセス生成の 100 倍速い
  - 例：CGI はプロセス，PHP はスレッド
    - 多数の Web アプリを動かす際，メモリ消費は CGI > PHP
- スレッドは，次に CPU を受け渡したい待機中スレッドを指定可能
  - カーネルを経由しない → 高速にスケジュール可能

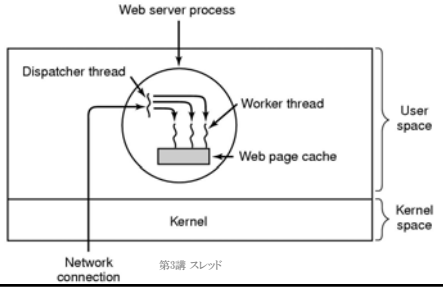
典型的なスレッドの使い方 (1)

- ワープロソフト内の並列処理
  - キー入力，文書整形，自動保存にそれぞれスレッドを使用



典型的なスレッドの使い方 (2)

- マルチスレッドウェブサーバ
  - ディスパッチャースレッド ＋ 複数のワーカースレッド
  - 多数のリクエストに対する処理をスレッドに振り分け

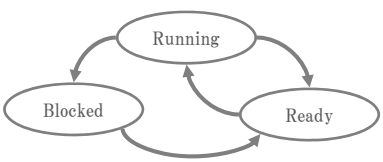


スレッドの作成，終了，切替

- **スレッドの作成**
  - 最初は 1 プロセス 1 スレッド
  - 各スレッドは他のスレッドを作成できる
  - ライブラリ手続：`thread_create`
  - 自動的に，作成元のスレッドのアドレス空間で実行されるため，新スレッドのアドレス空間の指定は不要
- **スレッドの終了**
  - スレッドは仕事を終えたとき終了できる
  - ライブラリ手続：`thread_exit`
- **スレッドの切替**
  - 自発的に他のスレッドに CPU を切り替えることができる
  - ライブラリ手続：`thread_yield`
  - プリエンプションがない時，全スレッドに実行機会与えるため必要

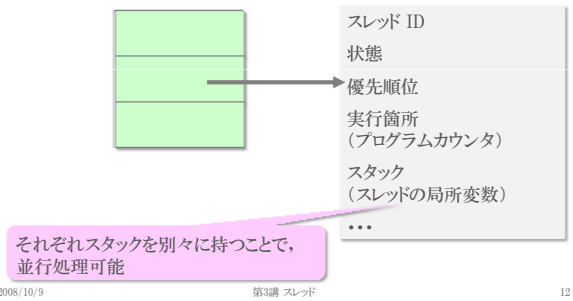
スレッドの状態

- プロセスと同じ
  - 実行状態 (Running)
  - 実行可能状態 (Ready)
  - 待ち状態 (Blocked)



スレッドの管理

- プロセス毎にスレッドを管理



スレッドの切り替え

- 「スレッド数 > プロセッサ数」の場合，プロセス同様に時分割でプロセッサ時間を割り当てる必要がある

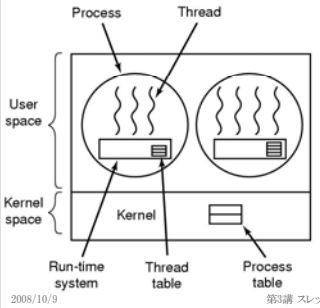
スレッドは必須か

安易なスレッドの利用が招く 3 種のリスク

- 明示的なスレッドの生成，抹消
  - ハードウェア構成に合わないスレッド数
    → 大幅な性能低下の危険性
- メモリ空間の共用
  - バグのあるスレッド同士でメモリ空間を共有
    → 共倒れの危険性
- スレッド・スケジューラ
  - スケジューラの詳細を知らずにプログラム作成
    → 飢餓状態，性能低下の危険性

スレッド機構の実現方法 (1)

- ユーザレベル・スレッド機構

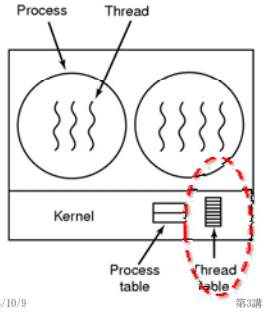


メリット
- スレッド間の切替が高速（スレッドテーブルをプロセス内に所有，かつ，システムコールを伴わないため）
- 次に実行するスレッドを指定できる

デメリット
- あるプロセスがシステムコールを呼び出す → プロセス全体（全スレッド）がブロックされる．
- クロック割込がない→あるスレッドが CPU を独占してしまう

スレッド機構の実現方法 (2)

- カーネル管理型スレッド機構

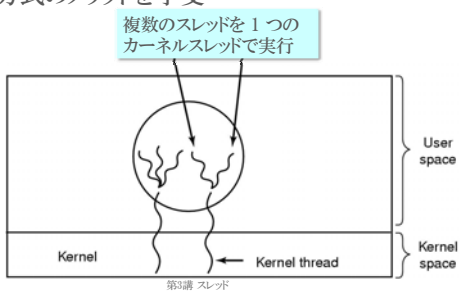


メリット
- スレッドテーブルの管理をカーネルが行うため，プロセス内での管理が楽（生成，抹消などはシステムコールを呼び出すだけ）
- あるスレッドが I/O などでブロックされても，スレッド全体はブロックされない

デメリット
- スレッド管理のためのシステムコールのオーバヘッドが大きい

スレッド機構の実現方法 (3)

- ハイブリッド型スレッド機構
  →両方式のメリットを享受



競合状態（race conditions）

- 複数スレッド（プロセス）動作時の競合問題

```
deposit(N, x)
{
        a = read(N);
        a = a + x;
        write(N, a);
}
```

①a ← 10万
③a ← 10+2万
⑤残高 ← 12万

```
withdraw(N, x)
{
        a = read(N);
        a = a - x;
        write(N, a);
}
```

②a ← 10万
④a ← 10-5万
⑥残高 ← 5万

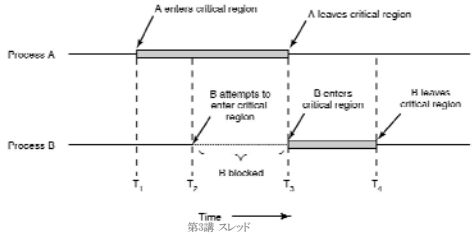
- 銀行口座への入金（deposite）と出金（withdraw）が，同時に実行されたら？
  →**競合状態**（race conditions）：複数プロセスが共有データへ同時にアクセスする可能性があり，共有データへのアクセスのタイミングで結果が異なってしまう状態
- 競合状態にある複数のプロセスの実行を正しく行うには？

競合の回避

- 同時並行に実行されては困る部分を **critical region**（または，**クリティカル・セクション**）という
  - クリティカル・セクションは不可分（atomic）に実行されなければならない
- 解決法
  - 排他ロックする
    - `lock(); critical_section(); unlock();`
  - 排他ロックの実現方法
    - Busy Waiting
    - Sleep & wakeup / Semaphore

競合状態回避のための条件

- 競合状態回避のための 4 つの条件
  1. 2 つ以上のプロセスが同時にクリティカルセクションに入ってはいけない
  2. CPU の数や CPU の処理速度を仮定しない
  3. クリティカルセクション以外で他のプロセスをブロックしない
  4. どのプロセスもいつかはクリティカルセクションに入ることができる



Busy Waiting(1): 交互実行

- 変数の値で実行するプロセスを交互に決める

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```
(a)

CRを出たら (b) の番に

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```
(b)

- 競合状態回避の条件 3 を満たさない
- プロセス(a) が`noncritical_region()` でブロックされると上手く動作しない

Busy Waiting(2): Peterson' Solution

```
#define FALSE 0
#define TRUE  1
#define N     2                     /* number of processes */
int turn;                           /* whose turn is it? */
int interested[N];                  /* all values initially 0 (FALSE) */
void enter_region(int process);     /* process is 0 or 1 */
{
    int other;                      /* number of the other process */
    other = 1 - process;            /* the opposite of process */
    interested[process] = TRUE;     /* show that you are interested */
    turn = process;                 /* set flag */
    while (turn == process && interested[other] == TRUE) /* null */ ;
}
void leave_region(int process)      /* process: who is leaving */
{
    interested[process] = FALSE;    /* indicate departure from critical region */
}
```

交互実行の問題を解決

CR に入りたいという意向を書込む

- 相手が後に `turn` を書換→ループ終了→継続
- 自分が後に `turn` を書換→相手の終了を待つ

問：この方法がうまく機能することを確かめよ

セマフォ（semaphore）

- 登場の背景
  - Peterson の排他ロック法は busy wait に基づく方法のため，priority inversion が発生する可能性がある
    - 低優先度プロセスがクリティカルセクションを実行中，高優先度プロセスが ready になり実行された場合，など
  - `sleep` & `wakeup` の使用
    - `sleep` を実行したプロセス（スレッド）は，`wakeup` で他のプロセス（スレッド）から起こされるまでブロック
- **セマフォ**
  - 実行可能な（既に呼ばれた）`wakeup` の回数を保持する整数変数
  - 値が 0: 呼ばれた `wakeup` は 0 回
  - 値が 1 以上: 1 回以上の `wakeup` がペンディングされている

※Mutex: セマフォの値を 0, 1 に制限したもの

セマフォにおける操作

- **`down()`** または `sleep()`
  - セマフォが正の数になるまで待ち，1 減算
  - クリティカルセクションに入る前に呼ぶ
- **`up()`** または `wakeup()`
  - セマフォを 1 増やす → 待っているプロセス（スレッド）を起こす
  - クリティカルセクションから出る時に呼ぶ

※セマフォの減算，加算処理は不可分に実行できないといけない

セマフォによる競合問題の解決

```
initialize(S, 1);

deposit(N, x)
{
        down(S);
        a = read(N);
        a = a + x;
        write(N, a);
        up(S);
}

withdraw(N, x)
{
        down(S);
        a = read(N);
        a = a - x;
        write(N, a);
        up(S);
}
```

S > 0 になるまで待ち，1 減算

Critical section

S を 1 増やし，待っているスレッドを起こす

Mutual exclusion

問：この方法がうまく機能することを確かめよ

実用上の注意

- critical section の範囲は必要最小限にする
  - `lock()` と `unlock()` の範囲は並行動作できない
  - 並列性が阻害される

まとめ

- スレッドのモデルと用途
  - スレッドは実行の単位．プロセス内に複数スレッドを生成・実行
  - 複数スレッドが同じアドレス空間，他の資源を共有（保護なし）
  - 複数のスレッドが協力・連携して同じ仕事を処理するような用途
- ユーザスレッドとカーネルスレッド
  - それぞれ長所と短所がある
  - 両方の長所を併用するハイブリッド実装方式も存在
- スレッドの管理
  - スレッドの生成，消滅，切替 → プロセスの場合とほぼ同じ
- 競合状態の回避
  - クリティカル・セクション
  - Busy waiting，セマフォ