計算機プログラミング輪講I&II '97:
Dough Lee: Concurrent Programming in Java ＃1
Tutorial: 並行制御と並行プログラミング

久野　靖*

1997.9.1

1　はじめに

今回は表記の本の輪講に決まったわけですが、本日はキックオフということで、並行プログラミングのチュートリアルを簡単にやります。

2　並行プロセスと競合

2.1　裸の計算機では…

Q 複数の実行主体 (プロセス、スレッド) が並行して実行 → そもそも、なんでそんなことが起こり得るの?

Q 並行プロセスは、なぜ制御しなければならないの?

Q 並行プロセスを制御する方法?

Q たとえば、次の並行プロセスの競合を解消するには?

```
Process A:          Process B:
  load  r1,count      load  r1,count
  add   r1,$1         add   r1,$1
  store r1,count      store r1,count
```

並行プロセスを制御する基本は排他制御 (コードのある部分 — 排他領域、critical region — は一時には 1 つのプロセスしか入れなくする。

- 排他制御を実装する方法? → 旧来の Unix では…
- 以下の puzzle の前提: メモリの読み、書きは atomic
- 基本的なアイデア: メモリに「旗」を立て、旗が立っていれば他人は入らないようにする。
- naive な解だとぼろぼろ → じゃあどうする?

```
Process A:
  while(flag != 0) ;
  flag = 1;
  /* some task */
  flag = 0;
```

- Dekker の解

```
Process A:
  flag_a = 1;
  while(flag_b == 1) {
    while(turn != 'a') flag_a = 0;
    flag_a = 1;
  }
 /* some task */
 turn = 'b'; flag_a = 0;
```

- Peterson の解

```
Process A:
  flag_a = 1; turn = 'a';
  while(turn = 'a' && flag_b == 1) ;
  /* some task */
  flag_a = 0;
```

*筑波大学大学院経営システム科学専攻

2.2 現実の計算機では…

いちいち前節のようなアルゴリズムでは大変

- 順番にスケジュールとは限らない
- busy wait loop は無駄

→ ではどうする?

- より高機能な命令: Test And Set、Swap などの命令
- 短い排他制御の上に長い排他制御を組み立てる

3 様々なプリミティブ/言語機構

3.1 セマフォ

セマフォとは、特別な機能を持ったカウンタのようなもの。

- P 操作: カウンタを減らす。0 のときは呼んだプロセスは寝る
- V 操作: カウンタを増やす。0 だったときは増やす変わりに一人起こす
- 実際には P とか V とか言うより wait, signal という名前
- 単純な排他制御はバイナリセマフォ(初期値が 1) で

```
Process A:
  wait(mutex);
  /* some task */
  signal(mutex);
```

- 例題: 生産者/消費者

```
semaphore empty = 0, full = N;

Producer:          Consumer:
  /* produce */      wait(full);
  wait(empty);       wait(mutex);
  wait(mutex);       /* get item */
  /* put item */     signal(mutex);
  signal(mutex);     signal(empty);
  signal(full);      /* consume */
```

3.2 イベントカウンタ

イベントカウンタ—番号札のようなもの

- Read(E) — E の値を返す
- Advance(E) — E の値を 1 ふやす
- Await(E, v) — E の値が v になるまで待つ
- 例題: 生産者/消費者 (排他が不要)

```
eventcounter in, out;

Producer:            Consumer:
  /* produce */        seq = seq + 1;
  seq = seq + 1;       await(in, seq);
  await(out, seq-N); /* get item */
  /* put item */       advance(out);
  advance(in);         /* consume */
```

3.3 モニタ

セマフォは構造化されてない…簡単にデッドロックさせてしまう。→より構造化/高レベルの言語機構→モニタ (Hore 1974, Brinc Hansen 1975)。

- モニタは言語構造→コンパイラがサポート
- モニタは排他領域→一時には 1 つのプロセスだけが実行
- モジュールの形→付属する手続き=モニタ手続き (入口で排他制御)
- 条件を伴う待ち合わせ→条件変数

```
monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure put;
  begin
    if count = N then wait(full);
    /* put item */
    count = count + 1;
    if count = 1 then signal(empty);
  end;
```

表 1: モニタの分類

特性	実行中優先	実行中非優先
Blocking	Signal and Urgent Wait $E_p < S_p < W_p$ Priority Blocking	Signal and Wait $E_p = S_p < W_p$ No Priority Blocking
Non-Blocking	Signal and Continue $E_p < W_p < S_p$ Priority Non-Blocking	Wait and Notify $E_p = W_p < S_p$ No Priority Non-Blocking
Quasi-Blocking	$E_p < W_p = S_p$ Priority Quasi-Blocking	$E_p = W_p = S_p$ No Priority Quasi-Blocking
Extended Immediate Return	Signal and Return $E_p < W_p$ Priority Immediate Return	$E_p = W_p$ No Priority Immadiate Return
Automatic SIgnal	Automatic Signal $E_p < W_p$ Priority AUtomatic Signal	$E_p = W_p$ No Priority Automatic Signal

```
  procedure get;
  begin
    if count = 0 then wait(empty);
    /* remove item */
    count = count - 1;
    if count = N - 1 then signal(full);
  end;
  count = 0; /* initialization */
end monitor;
```

モニタ機構にはいくつものバリエーションがある。

- Hoare のモニタ: signal したとき、対応する wait プロセスがあれば、まずそれが動く (signal した人は寝る)。終ったら signal した人が続いて動作。間に他のプロセスは介在しない。
- Brinch Hansen のモニタ: signal はモニタ手続きの最後の文でなければいけない。signal した人は直ちにモニタから出ささされる。次は wait していた人が実行。
- 自動シグナルモニタ: wait が「wait 条件式」という形をしている。条件が定期的に調べられ、成立していると自動的に起きる。つまり signal が不要。その代わり実装が大変。条件式がモニタ変数のみしか参照できないという制約を設けるとやや実装が楽。

Buhr, Frotier, Coffin のサーベイではモニタを分類するのに、

- E_p — 呼ぼうとしている人の優先順位
- S_p — signal した人の優先順位
- W_p — wait している人の優先順位

の大小関係に基づいて整理している。

なお、モニタの中で別のモニタを呼ぶ (nested monitor calls) のはデッドロックの可能性があり危うい。

3.4 メッセージパッシング

メッセージパッシングは立派な並行制御機構。一番簡単なものとして次のものを考える

- send(相手, message)
- receive(相手, message)

```
Producrer:                Consumer:
  /* produce */             receive(producer, mes);
  receive(consumer, mes); send(producer, mes);
  send(consumer, mes);    /* consume */
```

メッセージにもいろいろな分類がある。

- 同期的/非同期的
- バッファされるか
- 順序の保存
- 返事の有無
- future

たとえばABCL/1ではメッセージとしてnow、past、futureの3種類が使用できる。

4　オブジェクト指向並行プログラミング

ここまでやってる時間があるのかどうか?

- Actorモデル
- ABCL/1
- 条件同期の表現方法
- ガード
- 明示的受信
- 受理集合
- 継承異常
- ...

参考文献

[1] Tanenbaum, Modern Operating Systems, Prentice-Hall, 1992.

[2] 前川　守, オペレーティングシステム, 岩波, 1988.

[3] Peter A. Buhr, Michel Fortier, Michael H. Coffin, Monitor Classification, ACM Computing Surveys, vol. 27, no. 1, pp. 63-106, 1995.