TSSI基盤技術研修コース
— プログラミング言語 — ＃3

久野　靖*

1999.5.28

はじめに

□ 第2回の内容について…

- 前回のアンケートを拝見しましたが「オブジェクト指向は始めてである」という人が予想したよりだいぶ多かったです。ある程度知っていることを想定して用意したので、ちょっとハードだったかも知れません。
- 「自己反映機能が面白かった」という意見を多く頂きました。一方で、「インタフェースは難しかった」というコメントも頂いています。インタフェースについては重要だと思うので、今回も追加して説明します。併せて前回省略した内部クラスについても説明します。

□ 第3回の構成:

- 前回の復習としてインタフェース機能、また前回省略したけれどJavaでは極めて重要な内部クラス機能について。
- 前回積み残したオブジェクト指向言語の利用技術について。
- 今回予定の内容である、並列プログラミングについて。

1　インタフェース機能と内部クラス

□ 前回の「言語機能としてのインタフェース機能の特性」とは別の視点

- インタフェース機能を何のためにどう使うか
- インタフェース機能があるとどう良いか
- インタフェースを活用する場合の課題
- その課題を解決する機能としての内部クラス

1.1　インタフェース機能の役割

□ オブジェクト指向言語によるシステム開発→複数のオブジェクトが結合して、相互に機能を利用し合う

□ そのオブジェクトが「交換可能」だったらどうか?

- システムの構造は同じままで、さまざまなバージョンのシステムが構成できる
- それに際して、「交換する」オブジェクト以外の部分は変更しないで済む

1.2　「交換可能」の実現方法?

□ 案1:　あるクラスのオブジェクトをそのサブクラスのオブジェクトで交換

- 「交換によって機能を増やす」場合には向いている
- 「まったく別のものに取り換えたい」場合には不向き←継承では変数やメソッドを引き継いでしまうから

□ 案2:　抽象クラスを用意し、交換可能な部品をそれぞれ抽象クラスのサブクラスのオブジェクトとして用意する

- 抽象クラスでメソッドや変数を用意すると、それは継承されてしまう(つまりすべての部品で共通に持つことになる)
- 変数もメソッドも定義しないとすると、何のためのクラス?

□ 案3:　「抽象クラス」でメソッドのシグニチャだけを定義し、各部品は自由に実装を行う→これがインタフェース機能

- つまりインタフェースとは「変数もメソッド実現も持たないような抽象クラス」(インタフェース機能のない言語では実際、インタフェースの代わりに何も定義しない抽象クラスを使う)

*筑波大学大学院経営システム科学専攻

1.3 例題: 「2 つのボタンを持つアプリケーション」

□ ボタンの名前、ボタンが押された時の動作、表示内容などをやりとりするためのインタフェースを定める

□ 具体的な動作やその実装はそのインタフェースを実装するクラスで個別に/自由に決めることができる

□ なお、Button クラスもボタンが押された時そのことを ActionListener インタフェースを実装したオブジェクトに伝達するようにできている→押された時の動作を仲介する「アダプタ」クラスを用意

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Sample11 extends Applet {
  Font fn = new Font("Helvetica", Font.BOLD, 24);
  Sample11Model model = new Sample11App();
  Label lab = new Label(model.getStr());
  Button b1 = new Button(model.leftName());
  Button b2 = new Button(model.rightName());
  public void init() {
    setLayout(null);
    add(lab); lab.setFont(fn);
    lab.setLocation(20, 20); lab.setSize(200, 40);
    lab.setBackground(Color.white);
    add(b1); b1.setFont(fn);
    b1.setLocation(20, 80); b1.setSize(60, 40);
    b1.addActionListener(
      new Sample11Adapter(model, this));
    add(b2); b2.setFont(fn);
    b2.setLocation(120, 80); b2.setSize(60, 40);
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        model.right(); repaint();
      }
    });
  }
  public void paint(Graphics g) {
    lab.setText(model.getStr());
  }
}

interface Sample11Model {
  public String leftName();
  public String rightName();
  public void left();
  public void right();
  public String getStr();
}

class Sample11App implements Sample11Model {
  String str = "STR";
  public String leftName() { return "A"; }
  public String rightName() { return "B"; }
  public void left() { str += "A"; }
  public void right() { str += "B"; }
  public String getStr() { return str; }
}

class Sample11Adapter implements ActionListener {
  Sample11Model model; Applet app;
  public Sample11Adapter(Sample11Model m, Applet a) {
    model = m; app = a;
  }
  public void actionPerformed(ActionEvent e) {
    model.left(); app.repaint();
  }
}
```

1.4 Java の内部クラス

□ 前の例題のようにインタフェースを使ってアダプタやモデルを作るとき:

- 小さいクラスが多数できてしまうので面倒
- それらのクラスから元のクラスのメソッドを呼ぶのが面倒

□ このため「クラス内部にクラス定義が書ける」ようになった

```
public class X {
    ...
    return new Y();

  class Y implements I {
    ... ←この中で X のメソッドが呼べる
  }
}
```

□ さらに、いちいち「Y」のような名前を考えないで済ませることもできるように「無名の内部クラス」構文がある

```
public class X {
    ...
    return new I() {
      ... ←先の Y の定義と同じもの
    }
}
```

- 記述が短くなるという点は便利だが、最初はちょっと分かりづらい
- 元のやり方でやっても別に悪くはない。それぞれの場面で選択

□ インタフェースの代わりに普通のクラスの名前を指定してもよい→その場合は内部クラスは指定したクラスのサブクラスになる

□ 先の例題の内部クラス版

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
```

```
public class Sample11b extends Applet {
  Font fn = new Font("Helvetica", Font.BOLD, 24);
  Sample11Model model = new Sample11Model() {
    String str = "STR";
    public String leftName() { return "A"; }
    public String rightName() { return "B"; }
    public void left() { str += "A"; }
    public void right() { str += "B"; }
    public String getStr() { return str; }
  };
  Label lab = new Label(model.getStr());
  Button b1 = new Button(model.leftName());
  Button b2 = new Button(model.rightName());
  public void init() {
    setLayout(null);
    add(lab); lab.setFont(fn);
    lab.setLocation(20, 20); lab.setSize(200, 40);
    lab.setBackground(Color.white);
    add(b1); b1.setFont(fn);
    b1.setLocation(20, 80); b1.setSize(60, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        model.left(); repaint();
      }
    });
    add(b2); b2.setFont(fn);
    b2.setLocation(120, 80); b2.setSize(60, 40);
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        model.right(); repaint();
      }
    });
  }
  public void paint(Graphics g) {
    lab.setText(model.getStr());
  }
}

interface Sample11Model {
  public String leftName();
  public String rightName();
  public void left();
  public void right();
  public String getStr();
}
```

□ 内部クラスの利点

- 多数のクラスでプログラムが繁雑にならないで済む
- 内部クラスはそれを生成したインスタンスにアクセス可能→受渡しの繁雑さが低減される

1.5 ここまでのまとめ

□ インタフェース→システム内の部品を「取り換える」ことができる「切り口」を提供する

□ 内部クラス→インタフェースに適合するような「小さい」クラスをその場に書くことができる

2 オブジェクト指向の利用技術

□ オブジェクト指向言語→これまでの言語にないさまざまな「道具だて」(例: サブクラス、動的分配、継承、委譲、インタフェース、…)

- それをどのように使うか→なかなか難しい問題

□ どのような方向での利用があるか?

2.1 QUIZ: オブジェクト指向を活用するには…

□ プログラマでなければ OO を活用するのは無理…Y/N?

□ 継承による差分プログラミングを積極的に活用…Y/N?

□ サブクラスを作るには親クラスを熟知する必要…Y/N?

□ 誰が設計してもクラス間の「配線」は似て来る…Y/N?

2.2 オブジェクト指向にあった設計

□ 美しいプログラム構造 (何が美しい???) ←オブジェクト指向言語の特性に合ったプログラム設計← OOSE(オブジェクト指向分析、オブジェクト指向設計、標準記法 --- UML ---): 本講の範囲外

2.3 再利用

□ 再利用←書くよりは書かないで済ませた方が生産効率はよいに決まっている

- なぜオブジェクト指向で再利用? → クラス、オブジェクトといった単位は従来の「サブルーチン」より固まりが大きく、再利用に向いている
- 再利用するものは何? コード? 設計知識? → さまざまなレベルがある

2.4 クラスライブラリ

□ Smalltalk-80 →充実したクラスライブラリが付属→クラスライブラリを熟知すれば生産性が高まる、というブームに

- 実際にやってみると、よいクラスライブラリの開発/クラスライブラリに熟知した人材の育成ともに簡単ではない

□ Smalltalk-80 クラスライブラリでは差分プログラミング (子クラスで親クラスの機能を少しずつ拡張していく) を多用→これもよい手法だと一時思われていた

- しかしやってみると、よい差分プログラミングは難しい (クラス間の依存関係が大きくなりぐちゃぐちゃになりやすい)

□ 現在では、クラスライブラリはもちろん必要だが、整った機能を一式、分かりやすいインタフェースで提供するという当たり前の結論に

2.5 アプリケーションフレームワーク

□ 抽象メソッド: 親クラスで「自分自身へのメソッド呼び出し」を用いたメソッドを定義→子クラスでそれらのメソッドを具体的なものに差し替え

□ これをさらに発展させて、汎用的なアプリケーション全体の構造を予め定義しておく→その中のいくつかのクラスをサブクラス化してそこに各アプリケーション固有の部分を記述することでアプリケーションを完成させる

- 例: MFC、ET++、Choices、…
- うまく当てはまれば生産性は高まるが、何をどうサブクラス化するか、サブクラスはどのような規約に従う必要があるか、といったことを学ぶのが大変

□ たとえば、アプレットもアプリケーションフレームワークの 1 つ

- アプレットはクラス Applet のサブクラスとして作り、初期設定メソッド init() 描画メソッド paint() 等を必要に応じてオーバライドする

□ MFC →マルチドキュメントアプリケーションのフレームワーク

- アプリケーションクラスで「ファイルを開く」「画面に描く」等の動作を実現
- アプリケーション全体の動作は MFC のライブラリ側で提供

□ 先の例題アプレット→これも簡単なフレームワークの一例

2.6 コンポーネント

□ ここまでの再利用技術→基本的にクラス (群) が対象→プログラマ向け (クラスベースとも言う)

□ プログラミングをしない人に使える再利用技術→インスタンスベースの再利用

- インスタンスを生成し、そのプロパティ(属性、要するにインスタンス変数の値) をカスタマイズする
- カスタマイズしたインスタンス群をディスク等に保存しておき、それを取り出してそのまま動かす
- そのようなインスタンスを「コンポーネント」と呼んでいる。ソフトウェア開発のためのコンポーネント群→「コンポーネントウェア」、コンポーネントウェアに基づくソフトウェア開発→「部品組み立てプログラミング」

□ 代表的な成功例→ VisualBasic (部品: VBX、OCX… COM、DCOM の部品)

- ほかにも国産の IntelligentPad、Java ベースの JavaBeans などいろいろある
- しかし、部品とその配線だけでできるプログラムで十分なの?

2.7 デザインパターン

□ パターン: 「繰り返し現われるようなカタチ」

□ (ソフトウェアにおける) デザインパターン: オブジェクト指向ソフトウェア開発において、有効に使えるようなオブジェクト群の構成のパターン

- 1990 ころから、Peter Cord 他がはじめた。日本では「ガンマ本」が有名になっている。ガンマ本はよく使うパターンを集めた「パターンカタログ」になっている。
- なぜデザインパターン? → オブジェクトの接続関係のノウハウはかなり難しい (ちょっと思いつかないようなものもある) →それを蓄積しておいて流通させると、うまくはまったときに役立つ

□ 例: Command パターン

- メニューの選択、画面上のボタンなどはどれも「何らかの動作」を起動する→「動作をするオブジェクト」を用意して、それをメニューやボタンに結びつけていけばよい。

□ 例: Adapter パターン

- ボタンが押されたときに呼び出されるメソッドはある名前に決まっている。しかし実際に起きて欲しいことを実行するメソッドは別のメソッドである→「仲

介するオブジェクト」を用意して、それが橋渡しをすればよい。

□ 実は上記 2 つは前の例題に含まれていた。

□ 例: Visitor パターン

- オブジェクトの階層構造で構造化グラフィクスとか複合文書のようなものを作ったとする。「印刷する」「表示する」「ファイルに保存する」「スペルチェックする」等それぞれの場合について、各クラスにそれ用のメソッドを作るのは面倒である → どうする???

□ Visitor パターンとは:

- 各オブジェクト側には「accept」というメソッドを 1 つだけ用意しておく。その引数として、「印刷用の Visitor」「表示用の Vsitor」などさまざまな Viritor オブジェクトをそのつど渡せばよい

□ 例: AbstractFactory パターン

- Windows でも Mac でも X11/Unix でも同じに動作する GUI アプリケーションを開発するには???

□ AbstractFactory パターンとは:

- Window、Button、Dialog などの汎用的なクラスを用意する
- そのサブクラスとして MacWindow、X11Window などそれぞれ用意する
- WinFactory という抽象クラスを用意し、メソッドとして makeWindow、makeButton 等を用意する
- MacWinFactory、X11WinFactory などの具象クラスでこれらをそれぞれ実装する
- アプリケーションの実行開始時に MacWinFactory などのインスタンスを作って WinFactory 型の変数に格納し、以後それを利用する

2.8 本節のまとめ

□ オブジェクト指向にはさまざまな「道具」が含まれている→その「道具」をどう使うか、についていろいろな工夫がある

□ しかしこれらもまだほんの一部? →これからより多くの「よりよい利用方法」が現われる (はず)

□ それらの利用方法を個別に「新しい」と思って受け入れるだけでは、流行に追われるだけ→必要なこと:

- その「新しい」技術がこれまで行われてきたさまざまなことの中にどのように位置づけられるのかを考える
- 結局、ソフトウェアの生産において「自分が必要とすることは何か」をまず考え、それに照らして必要なものを取捨選択する

3 並列システム

□ 並列 (parallel) システムとは

- 複数の動作が「同時に」「並行的に」行われるようなシステム
- 動作が「順番に」「逐次的に」行われるよりも速い

□ cf. 並行 (concurrent) →「論理的に並列」(例: 1CPU でのマルチタスク)

- 並行であっても並列でないと速くはならない

3.1 システムを速くする方法

□ 方法 1: CPU の動作を速くする

- ソフトウェアを変更しなくて済む→大きな利点
- 素子の高速化は限界に近付きつつある (といいつつ…)

□ 方法 2: CPU をたくさん搭載する (マルチプロセサ)

- 過去においてはハードウェア量が N 倍になるので不利だとされていた
- 現在は CPU が安く/小さくなり、実装技術も進歩
- 結果的に「動作を速くする」よりコストパフォーマンスが良くなった
- しかし、ソフトウェアは同じでは済まない「かも」

3.2 細粒度並列性

□ 命令レベル並列性ともいう

- 1 つの機械命令の中でも並列に動作可能な部分が含まれる
- 隣接する命令群の中でも同様
- これを活かすには→命令スケジューリング (ハード、ソフト) →# 1 で解説した通り

□ 命令レベル並列性の限界→並列度が 4 くらいで頭打ち

□ 命令レベル並列性の利点→言語は同じままでコンパイラだけ交換

3.3 粗粒度並列性

□ 命令レベルよりもっと上のレベルでの並列性→マルチプロセサ

- 例: 配列のスカラー倍→各要素独立に計算可能
- 例: 返値を使わないサブルーチン呼び出し→本体と並列に実行可能

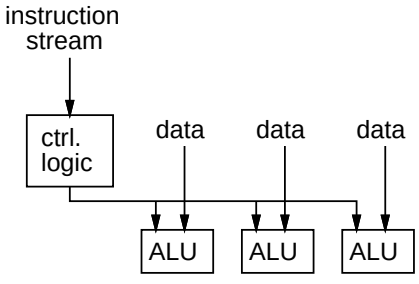
□ 並列度の上限はいくらでも大きくできる→以下ではこちらを対象

□ 粗粒度並列性の利用には 2 つ半 (?) のアプローチ

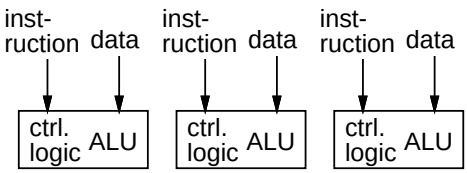
- 言語を変更しない→自動並列化コンパイラ
- 言語を変更しないが、プログラマが多少サポート→自動並列化+プラグマ
- プログラマが明示的に並列を意識してコーディング→ライブラリまたは並列言語

3.4 並列システムのモデル

□ SIMD (single instruction stream、multiple data) →プログラムの実行の流れは 1 つで、データは多数→データ並列とも呼ぶ→行列/ ベクトル計算などに適したモデル→現在は純粋な SIMD マシンはない。ベクトルプロセサがややこれに近い



□ MIMD (multiple instruction stream、multiple data) → CPU が複数あり、それぞれが別の実行の流れを持つ→現在のマルチプロセサの素直なモデル、主流

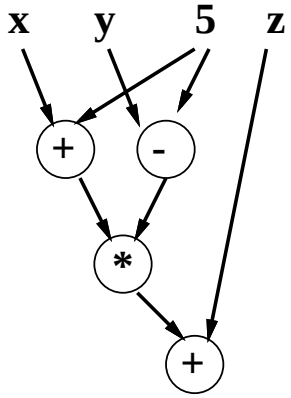


□ SPMD (single program、multiple data) →複数の CPU で同一のプログラムを同期を取りながら実行→MIMD マシンで SIMD のまねをする→データ並列の計算に使われる手法

3.5 データフローマシン

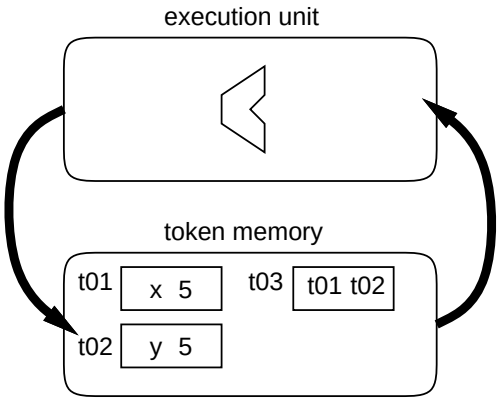
□ データの依存関のみに基づいて実行順序を制御するような専用マシン

□ プログラムはデータフローグラフとして表される



- 読みづらいので普通の言語ふうのものも

□ フローグラフの「ノード」をプールして「準備ができた」ものから「発火」

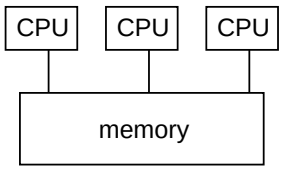


- 並列単位が小さいので同期処理のオーバヘッドが大きく性能が出なかった
- 処理単位をもっと大きくしたものを電総研などで開発 (EM-4 など)
- しかしあまり主流とは言えない

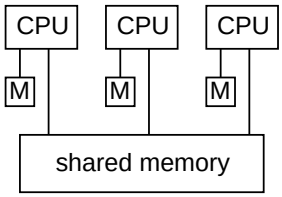
3.6 マルチプロセサの分類

□ マルチプロセサのモデル分類

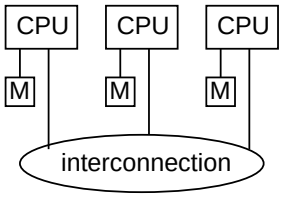
- UMA(uniform memory access) アーキテクチャ→各 CPU は 1 つの共有メモリをアクセスし、CPU ごとにアクセス時間の変動はない



- NUMA(non-uniform memory access) アーキテクチャ→各 CPU は手元のメモリとそうでないメモリにともにアクセスできる→場所によってアクセス時間が異なる



- NORMA(no remote memory access) アーキテクチャ→各 CPU は手元のメモリだけにアクセスできる (リモートアクセスはできない)



□ UMA → NUMA → NORMA の順で実装は面倒、使いやすさは低くなるが、その代わりスケーラビリティが増す

- これまでは普通の OS でそのまま利用できる UMA が多く使われて来た
- これからは 1000 とか 10000 とかの CPU を搭載したシステムも→ NUMA、NORMA が中心に

3.7 相互結合網

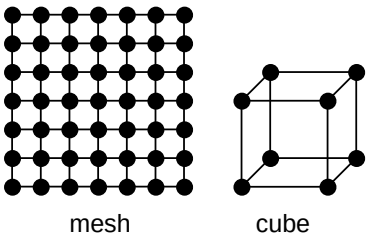
□ マルチプロセサ内部のデータのやりとり→相互結合網

□ UMA →スケーラビリティはあまりない (30 くらいまで?)

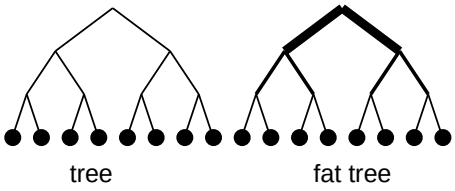
- クロスバスイッチで N 個の CPU と M 個のメモリを結合
- 共有バスとキャッシュ(後述) による結合

□ NUMA、NORMA →大規模マルチプロセッサ

- CPU+ローカルメモリ+キャッシュ→ 1 モジュール
- モジュール間の相互結合方式: ネットワークのようなもの
- メッシュ、トーラス、ハイパーキューブ、…



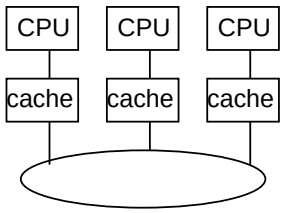
- ツリー、fat tree



3.8 キャッシュと整合性

□ キャッシュ→ CPU に隣接した高速記憶→メモリの内容を一時的に保持

- UMA、NUMA ではキャッシュを利用することで「遅い共有メモリを見なくて済む」ように



□ ただし共有メモリが書き換えられたときそのことを知らないでキャッシュを利用し続けると困る→整合性 (コヒーレンス) の問題

- スヌープキャッシュ→バスにキャッシュ+CPU がつながっている場合→バス上の通信を監視することで、各キャッシュが整合性を取れる→バス方式ではスケーラビリティはあまりない
- ディレクトリ方式→階層構造のシステムで使用→「どことどこにキャッシュされているか」を記憶しておき個別に通知

3.9 この節のまとめ

□ 並列とはどういう意味か、なぜ必要か

□ 並列システムの分類

□ 並列システムの構造

4 並列プログラミングのモデル

□ 並列プログラミング→「並列性を明示的に扱うような」プログラミング

□ 〃のモデル→「どのようなシステム」を抽象化したもの

□ 実際のシステムの構成と一致しない場合も

4.1 自動並列化

□ 通常のプログラムを自動的に (コンパイラが) 変換

□ 自動ベクトル化、MIMD 化、SPMD 化ともある

- データの依存解析が必要→コンパイラにとって難しい課題
- プログラマがプラグマでコンパイラに教えてやることも一般的

□ NUMA、NORMA の場合は配列などのデータ配置も重要→これも自動的に行う場合、プログラマがアドバイスする場合ともにある

□ 以下では明示的に並列性をプログラムするものとする

4.2 通信モデル

□ 複数の実行主体がどのように情報を交換するかのモデル

- 共有メモリモデル→ UMA のモデル化
- メッセージモデル→ NORMA のモデル化
- 実際にハードが UMA や NORMA だとは限らない

4.3 共有メモリモデル

□ 複数の実行主体がメモリ領域を (少なくとも部分的に) 共有しているというモデル

□ 利点:

- 通信が高速 (送ろうと思った瞬間にはもう相手にアクセスできる状態になっている
- 通信の単位が任意 (1 語ずつでも読み書きできる)
- ポインタが渡せる

□ 弱点:

- 競合/副作用の問題→排他制御 (後述)

4.4 メッセージモデル

□ 実行主体どうしでメモリは共有せず、メッセージ送信によって互いにデータをやりとりする

□ 利点:

- やりとりの機構に同期が組み込まれている (後述)
- 競合や副作用が起こりにくい

□ 弱点:

- 遅い、ポインタが渡せない、通信のオーバヘッド

4.5 モデルとシステム構成の独立性

□ 共有メモリシステムでメッセージ←別に難しくない (キューやバッファをメッセージチャネルと思えばよい)

□ NORMA や分散システムで共有メモリモデル←分散共有メモリ (ページフォルトが起きたときにページの内容を転送し、これまでページを持っていたところはページをはがす)

4.6 並列性のモデル

□ データ並列→プログラムの実行箇所は 1 つ (SPMD かも) →理解しやすい。並列性は行列、ベクトルなどのデータによる

- 数値計算ものではその利用は確立している
- 行列、ベクトルではメモリ配置が重要になる

□ 制御並列→複数の実行主体があるようなモデル

- 分配収集 (scatter/gather) モデル→ある程度規則的→実装はしやすい
- 並列オブジェクト指向→もっとばらばら

4.7 並列プログラミングの枠組み

□ 専用言語か旧来の言語か

- 新しい言語を作って普及させるのはとても大変
- そのため、研究レベルでは多くの言語が作られても、世の中に普及することは極めて少ない
- Java は極めて特異な例 (ただし Java は並列言語かどうか?)」

4.8 並列言語

- □ 並列言語→並列実行をモデルとして持つようなプログラミング言語
- □ 実際にはまだ「広く普及した」例はない
- □ データ並列型→それなりに実用。C*とか HPF (High Performance Fortran) など。基本的には多数のデータに対する並列動作をいかに輻輳を起こさず (マシンの特性を活かし切るように) 制御するか
- □ それ以外→複数の「実行の流れ」(MIMD に対応) →さまざまな流儀→以下で取り上げて行く

4.9 直列言語による並列プログラミング

- □ 通常の言語+ライブラリ呼び出しで並列実行を実現
- □ たとえばプロセスを複数生成→複数の CPU で実行可能
- □ 複数プロセス間でネットワーク通信→メッセージ通信による並列プログラミング
  - PVM、MPI、Linda 等→分散システムとしての利用が多い
- □ 複数プロセス間でメモリを共有→共有メモリモデルによる並列プログラミング
  - 複数プロセスを利用した場合、それらの切り替わりには必ず OS が介在→性能上は不利→スレッドの利用に

4.10 スレッド

- □ スレッドとは「実行の流れ」程度の意味
  - 1 つのプロセスの中で複数の実行の流れ→切り替わりのオーバヘッドがない→性能上有利
  - 実現→複数のスタックを置く
  - スタック以外のコード、データは共有されている
- □ カーネルスレッド→カーネルがそのスケジューリングを管理→複数の CPU で並列に動かすためにはこれが必須
- □ ユーザレベルスレッド→システムコールなしで、自前で (ライブラリの内部で) 切り替わりを実現→オーバヘッドが小さい
- □ 実際には両者を併用→論理的なスレッド数>>CPU 数なので、CPU 数+ α 程度のカーネルスレッドを用意し、カーネルスレッドが多数のユーザレベルスレッドを切替えながら動かして行く

4.11 Java のスレッド機能

- □ スレッドはクラス Thread で実現
- □ 方法 1: Thread のサブクラスを作り run() をオーバライド

```
public class Sample12 {
  public static void main(String args[]) {
    Thread t1 = new Sample12Thread("A", 10, 2000);
    Thread t2 = new Sample12Thread("B", 15, 1500);
    t1.start(); t2.start();
    try {
      t1.join(); t2.join();
    } catch(Exception e) { e.printStackTrace(); }
  }
}

class Sample12Thread extends Thread {
  String mesg = "???";
  int count = 10, time = 1000;
  public Sample12Thread(String m, int c, int t) {
    mesg = m; count = c; time = t;
  }
  public void run() {
    try {
      for(int i = 0; i < count; ++i) {
        Thread.sleep(time);
        System.out.println(mesg);
      }
    } catch(Exception e) { e.printStackTrace(); }
  }
}
```

- □ 方法 1 では、既に別のクラスのサブクラスになっているものを Thread のサブクラスにできないので不便
- □ 方法 2: Runnable インタフェースを実装したオブジェクトを用意し Thread のコンストラクタで引数として渡す

```
public class Sample13 {
  public static void main(String args[]) {
    Runnable r1 = new Sample13Run("A", 10, 2000);
    Runnable r2 = new Sample13Run("B", 15, 1500);
    Thread t1 = new Thread(r1);
    Thread t2 = new Thread(r2);
    t1.start(); t2.start();
    try {
      t1.join(); t2.join();
    } catch(Exception e) { e.printStackTrace(); }
  }
}

class Sample13Run implements Runnable {
  String mesg = "???";
  int count = 10, time = 1000;
  public Sample13Run(String m, int c, int t) {
    mesg = m; count = c; time = t;
  }
  public void run() {
    try {
```

```
      for(int i = 0; i < count; ++i) {
        Thread.sleep(time);
        System.out.println(mesg);
      }
    } catch(Exception e) { e.printStackTrace(); }
  }
}
```

□ これだけでは何が嬉しいのかよく分からないかも知れないが…たとえば、アニメーションを行うアプレットを作る場合、アプレットクラスに run() を用意してこれをスレッドで実行できると話が簡単

```
import java.applet.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;

class AnimCircle {
  Color cl;
  float vx, vy;
  float px, py;
  int rad;
  float basetime;
  AnimCircle(Color c, int x, int y, int r,
      float vx1, float vy1) {
    cl = c; px = x; py = y; rad = r; vx = vx1; vy = vy1;
  }
  public void setTime(float time) {
    basetime = time;
  }
  public void changeSpeed(float ratio) {
    vx *= ratio; vy *= ratio;
  }
  public void moveTime(float time) {
    px += (time - basetime)*vx;
    py += (time - basetime)*vy;
    if(px-rad < 0) { vx = -vx; px = 2*rad - px; }
    if(px+rad > 300) { vx = -vx; px = 600 - (px+2*rad); }
    if(py-rad < 0) { vy = -vy; py = 2*rad - py; }
    if(py+rad > 200) { vy = -vy; py = 400 - (py+2*rad); }
    basetime = time;
  }
  public void draw(Graphics g) {
    g.setColor(cl);
    g.fillOval((int)px-rad, (int)py-rad, 2*rad, 2*rad);
  }
}

public class Sample14
     extends Applet implements Runnable {
  AnimCircle a1, a2;
  long basetime = System.currentTimeMillis();
  boolean running;
  public void init() {
    this.setBackground(Color.white);
    a1 = new AnimCircle(Color.blue,
              100, 100, 20, 33.0f, 15.0f);
    a2 = new AnimCircle(Color.green,
              130, 110, 25, -23.0f, 45.0f);
    addKeyListener(new KeyAdapter() {
      public void keyPressed(KeyEvent e) {
        setKey(e.getKeyChar());
      }
    });
  }
  public void start() {
    running = true;
    (new Thread(this)).start();
  }
  public void stop() {
    running = false;
  }
  public void paint(Graphics g) {
    a1.draw(g); a2.draw(g);
  }
  public void setKey(int ch) {
    if(ch == '+') a1.changeSpeed(1.1f);
    if(ch == '-') a1.changeSpeed(0.9f);
  }
  public void run() {
    float t = 0.001f *
      (System.currentTimeMillis() - basetime);
    a1.setTime(t); a2.setTime(t);
    while(running) {
      try {Thread.sleep(100);}catch(Exception e){}
      t = 0.001f *
        (System.currentTimeMillis() - basetime);
      a1.moveTime(t); a2.moveTime(t); repaint();
    }
  }
}
```

4.12 この節のまとめ

□ 通信のモデル→共有メモリ vs メッセージ

□ 実行のモデル→データ並列 vs 制御並列

□ 言語の分類→並列言語 vs 直列言語+ライブラリ

5 並行制御と並列言語

□ 並行制御とは→複数の実行主体間で「実行の順序」を制御すること

□ なぜ必要?

- 例: 複数の下請けが仕事を請け負って並列実行→全部完了したらまとめて完成→「全部完了した」後でなければまとめられない
- 例: 複数の作業者が 1 つのデータベースを更新→同時に更新を行うと矛盾が生じる。たとえば次の NG

```
Process_A:          Process_B:
  //金額 x を計算      //金額 x を計算
  bal += x;           bal += x;
  //完了              //完了
```

- 上のようなコードは機械語レベルでは…

```
Process_A:        Process_B:
  load r1,x         load r1,x
* load r2,bal      load r2,bal
* add r2,r1        add r2,r1
* store r2,bal     store r2,bal
```

- だから何らかの並行制御により、「*」の 3 命令を同時には実行しないようにする必要がある。

□ 並行制御のための「機能」としてさまざまなものが提案されている→以下で見ていく→共有メモリを前提としたものが多い (歴史的事情)

5.1 排他領域

□ 排他領域とは→一時点には 1 つの実行主体しか実行が入れないようなプログラムコードの範囲

- 1CPU の場合「割り込み禁止」にすれば実現できる。マルチプロセッサでは別の方法が必要
- ある実行主体が入っているとき、別の実行主体が入ろうとすると待たされる
- 入っている主体が出て行くと別の実行主体が入れるようになる

5.2 排他領域の実現手法

□ 「専用命令」???

□ 共有メモリ→メモリ上の特定番地は 1 つの値しか保持しない (CPU ごとに別の値に見えることはない) →これを利用して実現。

- ダメな例:

```
Process_A:           Process_B:
 while(lock != 0) ;   while(lock != 0) ;
 lock = 1;            lock = 1;
 // critical          // ciritcal
 lock = 0;            lock = 0;
```

- 「同時に」何か起こっても大丈夫にするのはかなり面倒

5.3 Dekker のアルゴリズム

□ 最初に「正しい解」を考えた人が Dekker。

```
Process_A:          Process_B:
 in[A] = 1;          in[B} = 1;
 while(in[B]==1) {   while(in[A]==1) {
   while(pri!=A)       while(pri!=B)
     in[A] = 0;          in[B] = 0;
   in[A] = 1; }        in[B] = 1; }
 // critical         // critical
 pri = B;            pri = A;
 in[A] = 0;          in[B] = 0;
```

5.4 Peterson のアルゴリズム

□ Dekker の解よりも簡単なのを発見したのが Peterson。

```
Process_A:       Process_B:
 in[A] = 1;       in[B] = 1;
 turn = A;        turn = B;
 while(turn==A    while(turn==B
  && in[B]==1) ;   && in[A]==1) ;
 // critical      // critical
 in[A] = 0;       in[B] = 0;
```

□ 結局、これを N 個のプロセスでプログラムするのは大変

5.5 スピンロック

□ メモリ番地を「不可分にテストして変更」する命令 (Test and Set) があれば話はずっと簡単→現在の CPU にはほどんと備わっている。

- 具体的な動作→「番地 x が 0 でなければ false を返し、0 なら 1 に変更して true を返す」(CPU ごとに仕様はやや違う)

```
Process_A:       Process_B:
 while(TS(x)) ;   while(TS(x)) ;
 // critical      // critical
 x = 0;           x = 0;
```

- これならプロセスが何個あっても大丈夫
- 待っているプロセスはそこで「ぐるぐる回る」→「スピンロック」
- CPU を消費しながら待つ→「busy wait」→短時間の待ちにのみ使用する (排他領域は極めて小さいものであるべき)
- ハード的には→ TS のループはバストラフィックを作らない (キャッシュに入っているから) →システムの負担にならない (CPU 以外は)

5.6 セマフォ

□ セマフォ→特別なカウンタ。P と V という 2 つの操作が可能。

- P: カウンタを 1 つ減らす。減らした結果が 0 以下ならそのプロセスは「寝る」
- V: カウンタを 1 つ増やす。増やした結果が 1 なら 1 つプロセスを「起こす」

□ N 個の資源を共同利用するには初期値 N のセマフォを用いる

□ 初期値 0 のセマフォ→バイナリセマフォ→排他領域の実現に使える

- スピンロックと何が違うか→待っている間は CPU を消費しない。ただしオーバヘッドは大きい
- 実際には「セマフォの操作」を「スピンロックで排他制御」する

```
Sem_P:              Sem_V:
  lock(sem);          lock(sem);
  if(--sem<0) {       if(++sem==1) {
    unlock(sem);        //wakeup someone }
    //sleep self } unlock(sem);
  else
    unlock(sem);
```

5.7 条件変数

□ セマフォは「カウントが 1 未満の間寝て待つ」

□ より一般的に「これこれの条件を寝て待つ」→条件変数

- wait(c) : 条件変数 c で寝て待つ
- signal(c) : c で寝て待っている人を一人起こす

5.8 モニタ

□ モニタ→ C. A. R. Hoare などが提案した言語機構

□ セマフォやスピンロックは排他領域の「入口」と「出口」でペアになって使わないといけない→失敗すると大変

□ 実際に「保護したい対象」も含めて言語の構文として用意した方がよい

□ モニタ: モジュールのカプセル化機能に排他制御を追加したもの

- モニタの操作 (手続き) を呼び出すとロックが掛かる
- 手続きが終って出るとロックが開放される
- 待ち合わせは条件変数を指定して wait(c) →待ちに入るとロックは開放
- 起こすには別の人がモニタに入って signal(c) → 1 人だけ起きる
- 起きたプロセスはロックを獲得した状態で動作を続行

□ 例: Hoare ふうのモニタ

```
type buf = monitor
  var val:integer;
      isin:boolean := false;
      full,empty: condition;
  procedure put(i:integer);
  begin
    if isin then wait(full);
    isin := true; val := i;
    signal(empty)
  end;
  function get():integer;
  begin
    if not isin then wait(empty);
    isin := false; get := val;
    signal(full)
  end
end;
```

5.9 Java のモニタ機構

□ Hoare のモニタを Java むけに手直し

- モニタ→ 1 つのオブジェクトに 1 つのモニタ (ロック) が対応
- 「synchronized(オブジェクト) {…}」が排他領域
- メソッドの修飾子として synchronized をつけてもよい (そのインスタンスまたは static メソッドならクラスオブジェクトのモニタが使われる)

□ 簡単な例 (金額の更新を排他制御)

```
class Account {
  int bal;
  public calcadjust(...) {
    // amt に調整値を計算
    synchronized(this) { bal += amt; }
  }
  public synchronized adjust(int v) {
    bal += v; // synchronized(this)... と同じ
  }
  int getvalue() {
    return bal; // 読み出すので synchronized 不要
  }
}
```

- もし値が long だとこれは保証されない (Java では long と double は JVM 内部で 2 つ以上の命令に分かれて実行されることを想定)

□ 条件変数→クラス Object でメソッド wait()、notify() を用意→ということは、モニタ 1 つに条件変数が 1 つしかない。結構不便

```
public class Sample15 {
  public static void main(String args[]) {
    BoundedBuf b = new BoundedBuf();
    Thread t1 = new Putter(b, 100, 20);
    Thread t2 = new Putter(b, 200, 30);
    Thread t3 = new Getter(b, 50);
    t1.start(); t2.start(); t3.start();
    try {
      t1.join(); t2.join(); t3.join();
```

```
    } catch(Exception e) { e.printStackTrace(); }
  }
}

class Putter extends Thread {
  BoundedBuf buf;
  int start, count;
  public Putter(BoundedBuf b, int s, int c) {
    buf = b; start = s; count = c;
  }
  public void run() {
    try {
      for(int i = 0; i < count; ++i) {
        System.out.println("put "+(start+i));
        buf.put(start+i);
      }
    } catch(Exception e) { e.printStackTrace(); }
  }
}

class Getter extends Thread {
  BoundedBuf buf;
  int count;
  public Getter(BoundedBuf b, int c) {
    buf = b; count = c;
  }
  public void run() {
    try {
      for(int i = 0; i < count; ++i) {
        System.out.println("got "+buf.get());
      }
    } catch(Exception e) { e.printStackTrace(); }
  }
}

class BoundedBuf {
  int[] a = new int[10];
  int ipt = 0, opt = 0;
  public synchronized void put(int v)
      throws InterruptedException {
    while((ipt+1)%10 == opt) wait();
    a[ipt] = v; ipt = (ipt+1)%10; notifyAll();
  }
  public synchronized int get()
      throws InterruptedException {
    while(opt == ipt) wait();
    int x = a[opt]; opt = (opt+1)%10;
    notifyAll(); return x;
  }
}
```

5.10 デッドロックの検出と回避

□ デッドロック (dead lock) (rock じゃないよ!) とは「死んでしまった (開かない) 錠前」のこと。

□ 簡単に言えば「ロック (ないし一般の資源) の待ち合い」

```
Process_1:  Process_2: Process_3:
  lock a;     lock b;    lock c;
  lock b;     lock c;    lock a;
  //...       //...      //...
  unlock b;   unlock c;  unlock a;
  unlock a;   unlock b;  unlock c;
```

- より定式化すると「待ちに環状の依存関係ができてしまうこと」

□ 検出→待ちの依存関係を調べれば可能

- しかし実際にはこれは結構大変
- 実用的には「ぱたっと止まってしまった」→時間切れ→エラー (→再試行)

□ デッドロックを回避するには???

- その 0: 検出してやり直す→やり直せるとは限らない?
- その 1: 資源を「横取り可能」にする→横取り可能にできるかどうか?
- その 2: 資源を確保する時常に「この先必ず最後まで実行できる経路が存在する」ことを確認→確認するのはかなり大変
- その 3: すべての資源を最初にまとめて確保させる→資源の無駄の可能性
- その 4: 資源に「順番」をつけ、その順番でのみ確保→同上だがややマシ

5.11 楽観的な並行制御

□ 悲観的な並行制御とは→何がどう悪く行っても大丈夫なように考えて行動→ロックを掛けるというのはその代表例

□ 楽観的な並行制御とは→おおむねうまく行くんじゃないかと考えて行動→うまく行かなかったらご破算にしてやり直す (検出機構が必要)

□ トランザクション: 楽観的並行制御の代表

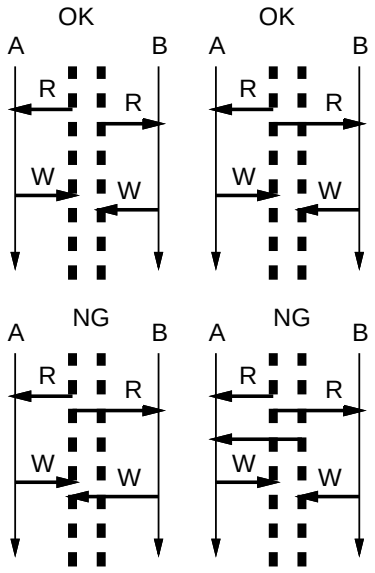
- 重要なデータを参照/更新する作業は必ず「トランザクション開始」と「トランザクション終了」で囲む
- 「トランザクション終了」は commit (成功) と abort (中止) の 2 種類ある
- すべての変更は commit したときはじめて恒久的なものとなる。abort したときは何ら状態に変化は起きない (システムダウン等のときも)

□ トランザクションの実装

- 影のページ方式→変更するデータをメモリ上で用意するが、commit まで書き込まないでおく。abort したらメモリ上のデータを捨てる
- ログ方式→変更する時は前の状態をすべてログに書いておき、abort したらログを見て元に戻す

□ どうやって並行制御?

□ 複数の実行主体による操作が入り混じっていても、「順番にやったのと同等の結果である」(=直列可能) ならばそれは正しい



- 各実行主体が何をどうアクセスしたかを記録しておき検査
- データベース機構としては当然存在するが、プログラミング言語でもトランザクション機能を取り入れた言語はある (例: Argus)。ただし少数派

5.12 メッセージ送信

□ メッセージ送信→「送った人が送った時点よりも、受け取った人が受け取る時点が必ず後になる」(アタリマエ)→同期機構として利用可能。

- 例: 分配収集型の実行→各分担者がマスターに「終った」というメッセージを送る→マスターは全員ぶんのメッセージが集まったら完了とみなす (実際には結果データもメッセージに入れて送るのが普通)

□ メッセージ送信に基づく言語: さまざまなバリエーションがある

□ メッセージの分類: 同期/非同期 (+未来)

5.13 同期メッセージ

□ 同期メッセージ→送り側が「送り終った」時点で、受け取り側も「受け取り終った」ことが保証されている

- 言語によってはさらに、受け取り側が処理を行い、返値を返す→送り側が返値を受け取れる、というものも
- 分散でない並列言語にとっては、1 つの自然な選択 (手続き呼び出しに類似しているため)
- 代表的なもの: CSP、Ada など

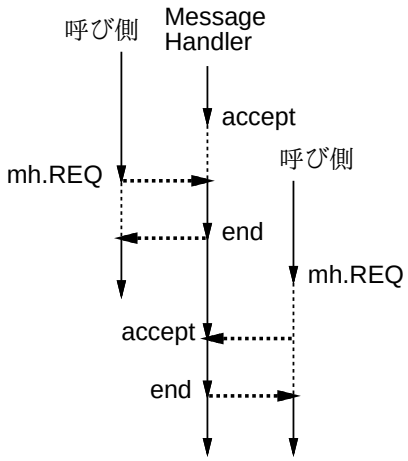
5.14 Ada のタスク機能

□ タスク→並列実行の単位 (スレッドのようなもの)

□ エントリ→タスク中のメッセージを受け取るところ (言語的には手続き呼び出しのように見える)

```
task body MessageHandler is
begin
  loop
    accept REQ(M: in STRING) do
      -- メッセージの処理
    end REQ
    --- その他の処理
   end loop
end MessageHandler
----
mh.REQ(...); ←呼び側
```

- 呼び出し側と受け取り側のタイミングが一致しないと、早い側が待たされる
- accept --- end が終るまで呼び側は先に進まない



□ さらに、複数の accept を並行して行える

```
loop
  select
    accept E1(...) do
      ...
    end REQ
```

```
    ...
  or
    accept E2(...) do
      ...
    end
    ...
  or
    ...
  or
    delay 0.5*SECONDS;
  else   ↑時間待ち文
    ... ←どの accept も
  end select   呼ばれず時間切れ
end loop
```

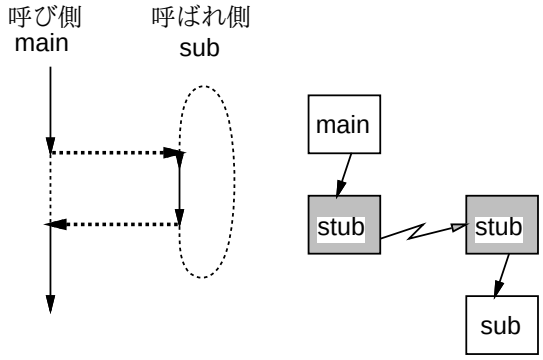
□ Ada では「呼び側」も「呼ばれ側」と同様に選択的に呼び出したり時間待ちできる→機能的にはとても充実

- もともとリアルタイム制御のための言語という側面があったため

5.15 RPC

□ RPC(remote procedure call) とは…手続きの呼びと戻りをメッセージに置き換えるもの

□ 呼び側→クライアント、呼ばれ側→サーバ



- 呼ばれ側は Ada のタスクで言えば

```
loop
  select
    accept ... end
  or
    accept ... end
  ...
end loop
```

に相当する無限ループ。

- パラメタや返値はネットワークメッセージへの詰め込み (marshalling) と取り出し (demarshalling) を行う。
- クライアントスタブ→呼び側で「本ものの代わりに呼ばれる」もの
- サーバスタブ (プロキシ) →呼ばれ側で「クライアントに代わって本ものの呼ぶ」もの

□ C 言語と RPC に基づく分散システム→多く使われている (例: NFS)

□ なぜ RPC か?→使い慣れた「手続き」そのままで済む

- その代わり Ada に見られるような細かい制御はない

5.16 非同期メッセージ

□ 非同期メッセージ→送り側が「送り終った」としても受信側はまだまったくそのことを知らないかも知れない。

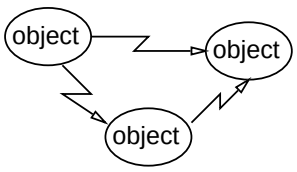
- いわゆる「メッセージ」のイメージによく合う
- 確認や返事などの機能が必要なら自前で実現する必要
- 面倒なので普通の並列言語では採用されない

5.17 並列オブジェクト指向言語

□ Smalltalk-80 のメソッド呼び出し→「メッセージ」

- Smalltalk-80 のメソッド呼び出しは逐次的だったが、これを「並列メッセージ」と解釈することも自然→並列オブジェクト指向言語

□ 並列オブジェクト指向言語とは→各オブジェクトが自律的に並行動作し、メッセージを交換し合いながら計算が進んで行くような言語



- 並列計算のモデルとして Actor、また実際の言語として Plasma、ABCL/1 などがある→実際の言語としてはあくまで研究向け
- ABCL/1 などは同期メッセージもサポート (便利さを重視)

5.18 並列オブジェクト指向言語と継続

□ オブジェクトはあくまでも先頭でのみメッセージを受け取る→ある仕事をしてその返事、というのは結構むずかしい

□ 返事を受け取る手段としての継続(continuetion)→返事を元のオブジェクトに返すのではなく、あるメッセージを送って、受け側の処理が終ったら、その結果を継続オブジェクトに送ってもらう

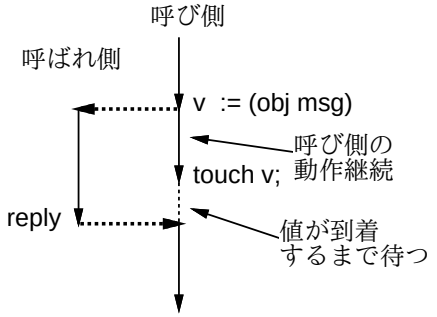
```
(send query: name to OBJ
   reply-to: ((reply: v) => value := value + v))
```

- Smalltalk-80 のブロックのような感じといえばいえる

5.19 未来 (future) メッセージ

□ 便利な形で返事を受け取るためのもう 1 つの方法

□ 同期メッセージと同様に返事を変数に受け取る→実際には返事はすぐにはかえってこない→その変数の値は実際に返事が帰って来るまでは利用できない



- 返事を使うところで touch を実行→返事が到着してなければ待つ
- 返事を使う箇所までは呼び側と呼ばれ側が並行動作可能

5.20 条件同期

□ 同期を取るときには、「このような条件が成立するまで待ち合わせる」ということがしたい

□ モニタでは条件ごとに条件変数を用意して、そこで寝ることで実現できる

- Java のモニタはモニタ 1 つに条件変数が 1 つしかないから面倒なことに…「全員が起きて、条件を調べて、成立してなければ寝る」

□ メッセージによる同期の場合は?→条件に応じて受理するメッセージを変えたい

- 例: 有限バッファなら「満杯のとき」は get()、「空っぽのとき」は put() のみを受理したい

5.21 ガード

□ メッセージの受理部分に書く条件式→ガード。ガードが真であるようなメッセージだけが受理される

□ 例: Ada では select 文の一部としてガード機能が使える

```
select
  when COUNT > 0 =>
    accept GET(V: out ITEM) do
      ...
    end GET;
or
  when COUNT <= MAX =>
    accept PUT(V: in ITEM) do
      ...
    end PUT;
end select
```

- ガードの実装は結構面倒 (なぜだと思いますか?)

5.22 受理集合

□ 受理集合→受理可能なメッセージの集合。受理集合を切替えることで「満杯ならば get() のみを受理」といった制御が行える

- 概念的には単純だが→あらゆるメッセージの on/off を自前で制御する→いわば goto 文のようなもの

5.23 継承異常問題

□ オブジェクト指向→継承機能を使いたい

□ しかし、同期条件を継承するのは難しい

□ 機能を拡張すると、同期条件がすべて変更になり、全部書き直しになりがち

- 例: Ada 流の accept 文→機能が増えると accept 文を全部書き直し
- 例: ガード→機能が増えるとガードの条件を書き直し
- 例: 受理集合→どれかの状態を分割したときに書き直し

□ 書き直しにならないための方法を多くの研究者が提案しているが、まだ決定版はない (久野の研究もある :-)

5.24 Java は並列言語か?

□ 広く普及した言語で並列機能をもとから持っているもの → Ada、Java

- ただし、Java では「オブジェクト」と「スレッド」は直交している→ C 言語にスレッドを入れて書くのとたいして変わらない
- より「並列」をきちんと (言語仕様として) 取り込んだ言語が必要では?
- 個人的にはそろそろ「並列オブジェクト指向言語」が実用になって欲しい (しかし現実にはなかなか…)

5.25 この節のまとめ

□ 並列プログラミングにはさまざまな機能や流儀

□ 進歩→次第に言語機構と統合

□ 今後の並列の利用増大→よりよい言語はまだ模索中 (永遠に?)

6 第 3 回課題

□ 以下の課題から 1 つ以上を選択してプログラムを作成し、実行例をつけて作成したプログラムが課題を達成していることを論ぜよ。課題中に指定した考察をきちんと書くこと。

□ (1) Sample11 の「2 つのボタンをもつアプリケーションフレームワーク」を改良して、N 個のボタン (それぞれ好きなラベルを指定できる) が持てるようにしてみよ。それを使って 2 つの別個のアプレットをデモ用に作成し、このようなアプリケーションフレームワークの利点と欠点について考察せよ。

□ (2) Sample14 のスレッドを用いたアニメーションアプレットを改良して次のどれか 1 つ以上の機能を実装しなさい。作成に当たってどのようにプログラム構造 (オブジェクト間の関係等) を工夫したかを考察すること。

- (a) 複数の物体が相互作用しながら飛ぶようにする。
- (b) 物体に重力 (でなくてもよいが一定方向への加速度) が作用するようにする。なおかつ物体の動作をキーボードで制御可能にしてゲーム (例: 的に当たるように制御) とし、加速度の向きが下向き (重力と同じ) とそうでない場合で人間の制御能力が変化するかどうか実験する。
- (c) 物体の数や飛び方を 2 種類以上に増やし、それらを実行時に制御できる (新しい物体を作ったり飛び方をスイッチする等) ようにする。制御方式を 2 種類以上考え、どちらが優っているかを実験により比較する。

□ (3) Java でセマフォクラスを作成せよ (各セマフォオブジェクトは p() と v() というメソッドを持つ)。これが実際にセマフォとして動作していることをデモプログラムを作って確認すること。次に作成したセマフォと Java 組み込みのモニタの動作を比較するデモプログラムを作成し、両者の使いやすさや性能について考察せよ。

□ (4) Java で汎用の (どのようなオブジェクトを持って来ても使えるような) ガード機能を持つ排他制御クラスを作成せよ。この問題はやや難しいので考察は自由でよい。

- ヒント: 排他制御を行うクラスは内部に排他制御オブジェクトのインスタンスを保持し、排他制御の必要な区間をメソッド enter() と leave() で囲む。ガードということは、enter() の引数として「自身の状態を検査して OK かどうかを調べる」ようなオブジェクト (内部クラスとして記述できると見やすいだろう) を渡すのでしょうね。

7 さいごに

□ この講座では、さまざまなプログラミング言語に現れる概念を、その設計思想、用途、実装、特徴、問題点などの観点から整理して見てきたつもりです。

□ 構成としては、1 回目→通常の言語、2 回目→オブジェクト指向言語、3 回目→並列言語、のように構成しました。

□ 資料を作りはじめて見ると思ったより説明したい内容が多く、ハードになってしまったようです。

□ 今年がはじめてなので分量や必要とするレベルが分からず苦労しましたが、おつき合い頂いた皆様もおつかれ様でした。