【セッションNo．3】

レスポンス向上！
モバイルアプリ実践開発テクニック

株式会社ミガロ.
RAD事業部　営業・営業推進課
尾崎　浩司

■ はじめに

- モバイルアプリの特長
  - タッチによる操作が中心

    キーボードのような打鍵感がない為
    タッチの際には、何かしらの応答が重要
  - 全画面表示

    PCのように複数ウィンドウが開かない為
    一つのアプリの動作が遅いとストレスになる
  - 一般的に端末のスペックがPCより低い

    動作に時間がかかっても、違和感の出ない
    工夫が重要

モバイル開発においては、これまでのPCアプリ開発とは異なるレスポンスの考慮が必要

モバイルアプリでレスポンスを向上する為のテクニックをご紹介！

【アジェンダ】

- レスポンス向上テクニック
  1. 並列処理によるレスポンス向上テクニック
     1-1. タスクを使用したスレッド分割
     1-2. 複数タスクの並列処理

  2. アニメーションによるレスポンス向上テクニック
     2-1. スライドアニメーションを使用した画面遷移
     2-2. アニメーション効果の活用

- まとめ

１．並列処理によるレスポンス向上テクニック

1-1. タスクを使用したスレッド分割

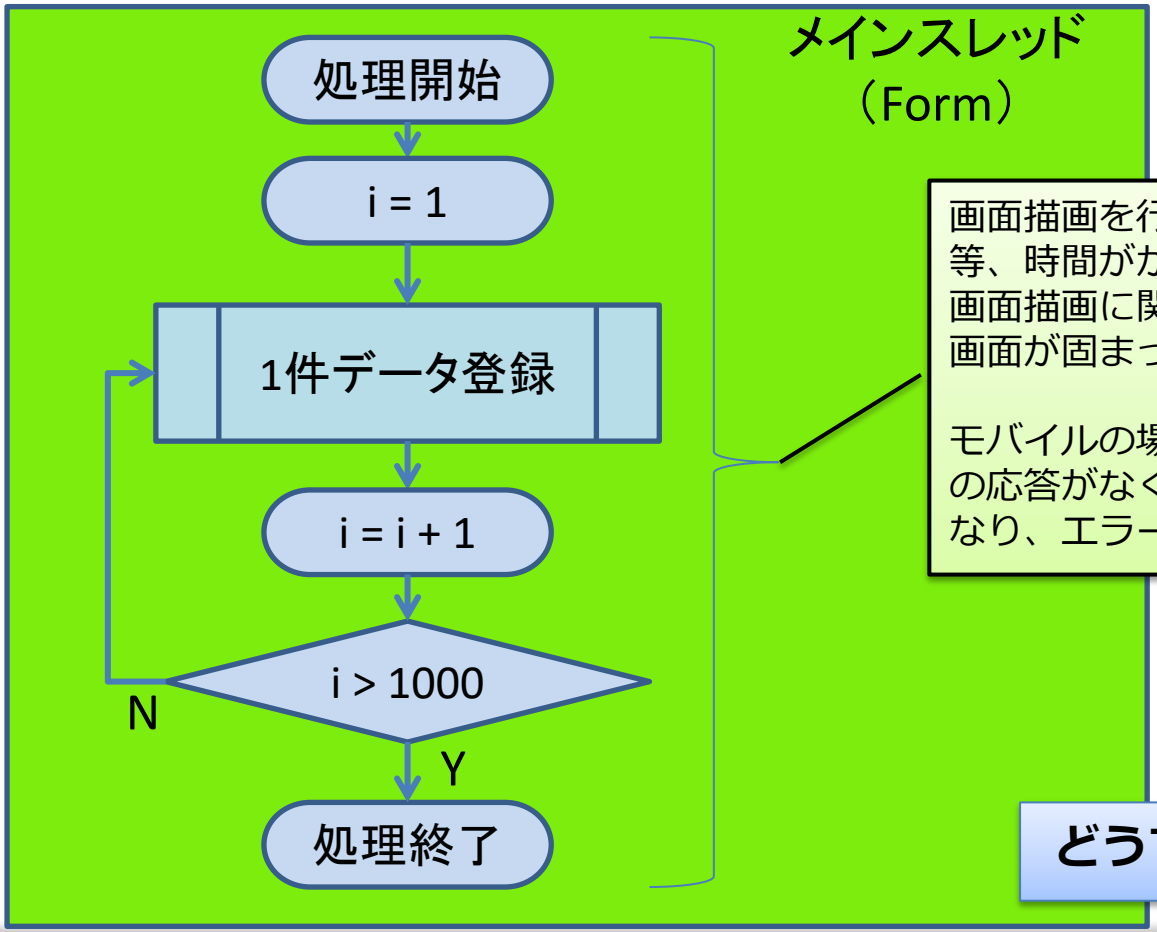
- レスポンスが悪いアプリとは
  - [実行]ボタンを押した後、応答があるまで画面が固まってしまう。
  - モバイルの場合、「応答なし」のメッセージが表示されることもある。

Androidの場合、一定時間応答なしの状態が続くと、「アプリ終了確認」ダイアログが表示されてしまう。

なぜ、画面が固まってしまうのか？

1-1. タスクを使用したスレッド分割

- 一般的なアプリケーション
  - シングルスレッドアプリケーション



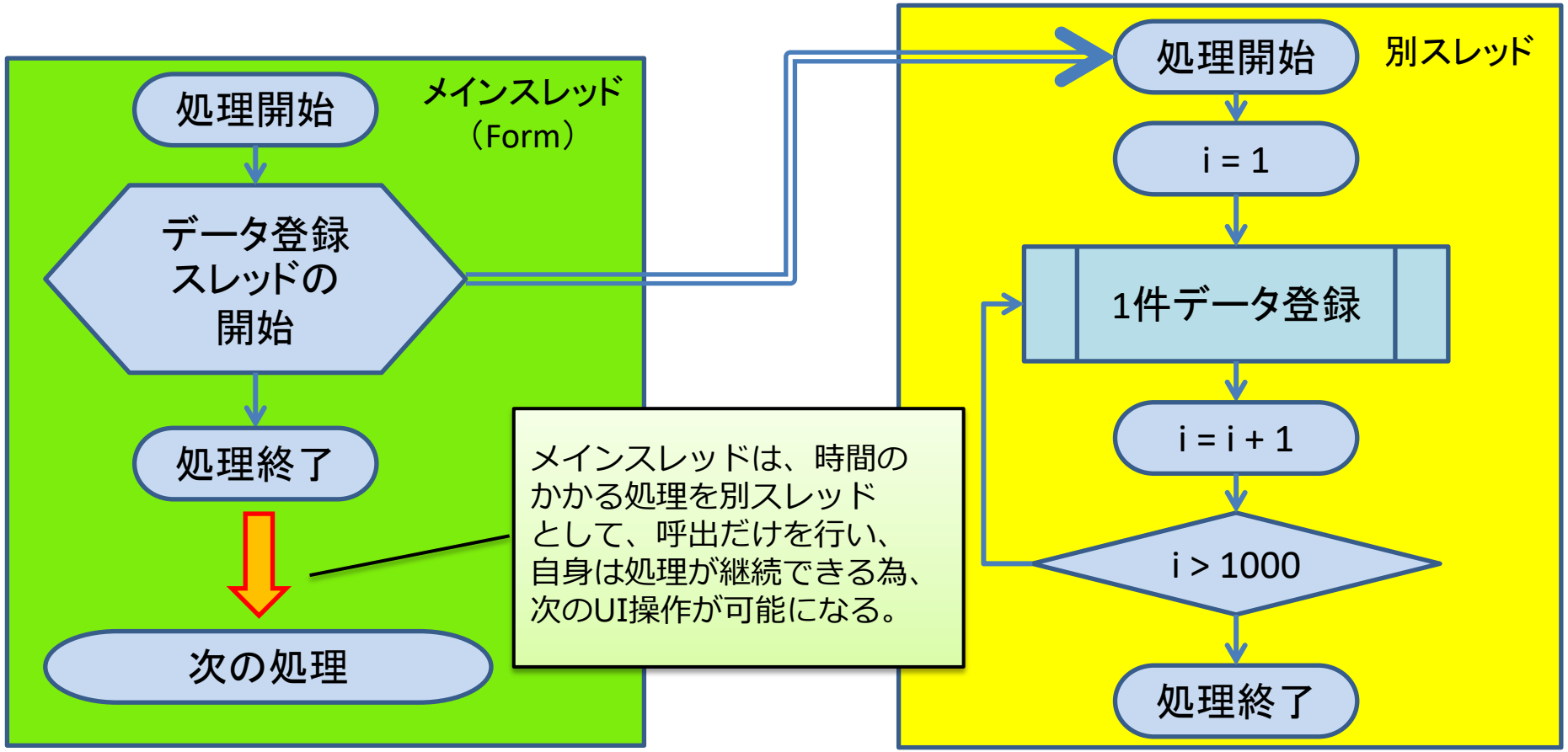
画面描画を行うメインスレッド上に、繰返し処理等、時間がかかる処理が含まれると、処理中に画面描画に関する処理が行えなくなり、結果画面が固まって応答なしの状態となってしまう。

モバイルの場合、画面描画を行うメインスレッドの応答がなくなると、一切のUI操作ができなくなり、エラーになりやすい。

どうすれば、固まらなくなるか？

1-1. タスクを使用したスレッド分割

- スレッドを分割し、メインスレッド（画面描画）が停止しないようにする。

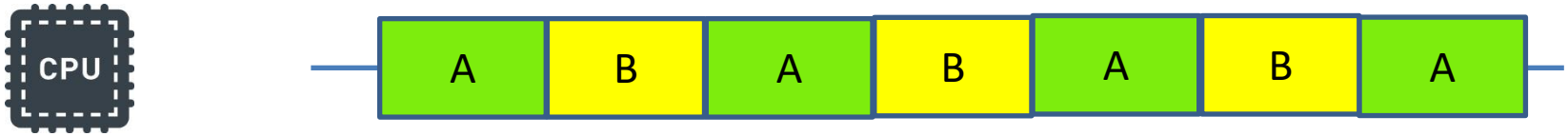


1-1. タスクを使用したスレッド分割

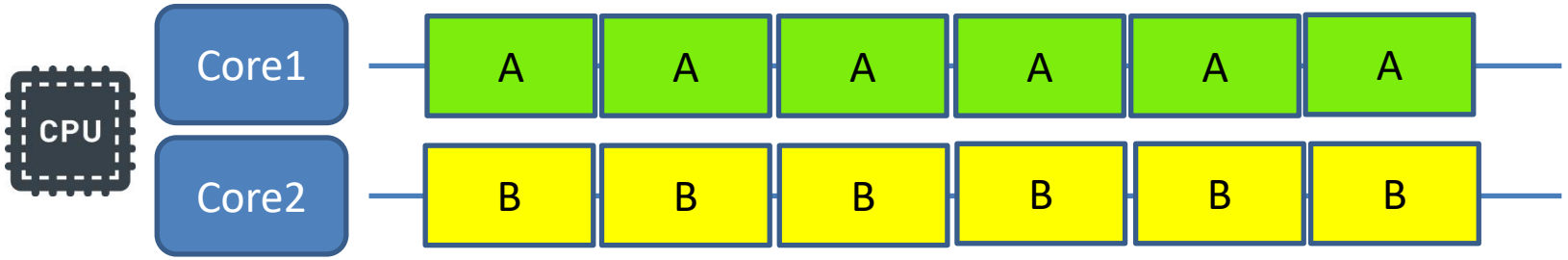
- スレッドを分割する方法
  - **TThreadを使用する方法**
    - 以前からあるスレッド分割処理
    - 使用方法は、第12回テクニカルセミナーにて紹介
      『Delphi/400開発～パフォーマンス向上テクニック～ 』
    - **並行処理**
  - **TTaskを使用する方法**
    - XE7以降で導入された新しい仕組み
    - 並列プログラミングライブラリ（System.Threadingユニット）
    - **並列処理**

1-1. タスクを使用したスレッド分割

- 並行処理と並列処理
  - 並行処理
    - CPUが超高速に処理を切り替えながら実行。（論理的な同時実行）



  - 並列処理
    - 複数のコアが複数処理を同時に実行。（物理的な同時実行）



1-1. タスクを使用したスレッド分割

- TTaskの使用方法
  - メインスレッドの中に直接並列処理を記述可能

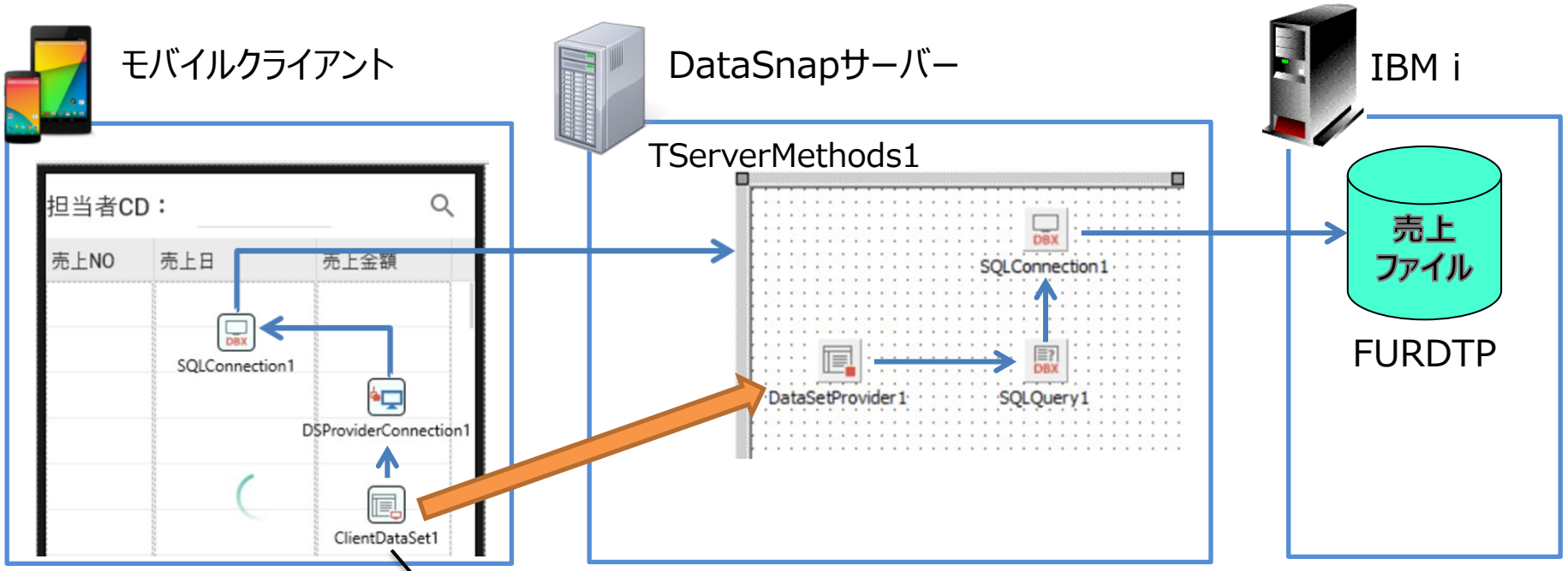
```
uses System.Threading;

procedure TForm1.Button1Click(Sender: TObject);
begin
  //ボタンクリックの処理
      …
  //タスク処理
  TTask.Run(procedure
  begin
    //重たい処理
    Sleep(10000);

    Edit1.Text := '処理終了';
  end);

  //後続UI処理
      …
end;
```

並列プログラミングライブラリを追加

並列処理を行いたい部分を
TTaskのRunメソッドで記述する。
処理は、無名メソッドとして定義。

1-1. タスクを使用したスレッド分割

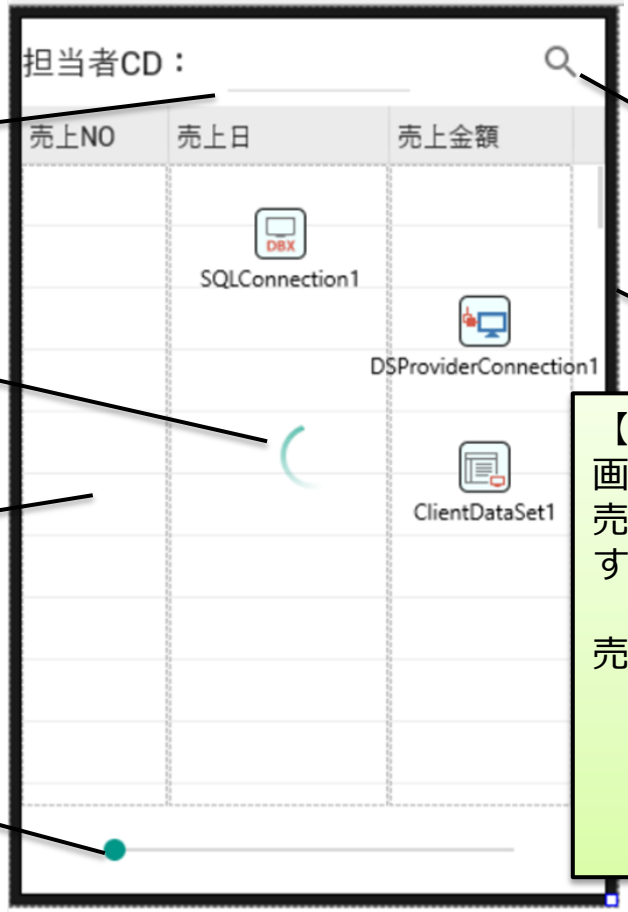
- 3層データベースアプリケーションへの実装
  - DataSnap（dbExpress）を使用したアプリケーション



ClientDataSetは、DataSnapサーバーアプリ上のDataSetProviderを経由して、SQLQueryコンポーネントのSQLが実行できる。

1-1. タスクを使用したスレッド分割

- 3層データベースアプリケーション
  - モバイルクライアント



EditCode: TEdit

Button1: TButton

AniIndicator1: TAniIndicator
回転インジケータコンポーネント

詳細は、「2-2. アニメーション効果の活用」にて紹介

StringGrid1: TStringGrid

TrackBar1: TTrackBar
スライダーコンポーネント

【アプリの仕様】
画面に入力した担当者CDに合致する売上データを検索し、グリッドに表示する。

売上DB（FURDTP）
　売上NO：URDTNO
　売上日　：URSYDT
　売上金額：URKNGK
　担当者CD：URTNCD

1-1. タスクを使用したスレッド分割

検索ボタンクリック時処理 (1)

```
procedure TForm1.Button1Click(Sender: TObject);
var
  iRow: Integer;
begin
  Button1.Enabled := False;

  //インジケーターの表示
  AniIndicator1.Visible := True;
  AniIndicator1.Enabled := True;

  iRow := 1;

  //Grid初期化
  StringGrid1.RowCount := iRow;
  StringGrid1.Visible := False;
  StringGrid1.Cells[0, 0] := '';
  StringGrid1.Cells[1, 0] := '';
  StringGrid1.Cells[2, 0] := '';

  with ClientDataSet1 do
  begin
    Active := False;
    CommandText := 'SELECT * FROM FURDTP'
                 + ' WHERE URTNCD = ' + QuotedStr(EditCode.Text)
                 + ' ORDER BY URDTNO';

    Active := True;

    First;
```

処理中を表す回転インジケーターを表示

StringGridをクリア（初期化）
処理中StringGridは非表示にする

抽出SQLを指定

データセットを開く

1-1. タスクを使用したスレッド分割

検索ボタンクリック時処理 (2)

```
    while not Eof do
    begin
      StringGrid1.RowCount := iRow;
      StringGrid1.Cells[0, iRow - 1] := FieldByName('URDTNO').AsString;
      StringGrid1.Cells[1, iRow - 1] := FormatFloat('0000/00/00',
                                        FieldByName('URSYDT').AsInteger);
      StringGrid1.Cells[2, iRow - 1] := FieldByName('URKNGK').AsString;
      Next;
      inc(iRow);
    end;

    Active := False;

    //インジケーターの終了
    AniIndicator1.Visible := False;
    AniIndicator1.Enabled := False;
    StringGrid1.Visible := True;
    Button1.Enabled := True;
  end;
end;
```

取得データをStringGridにセット（ループ処理）

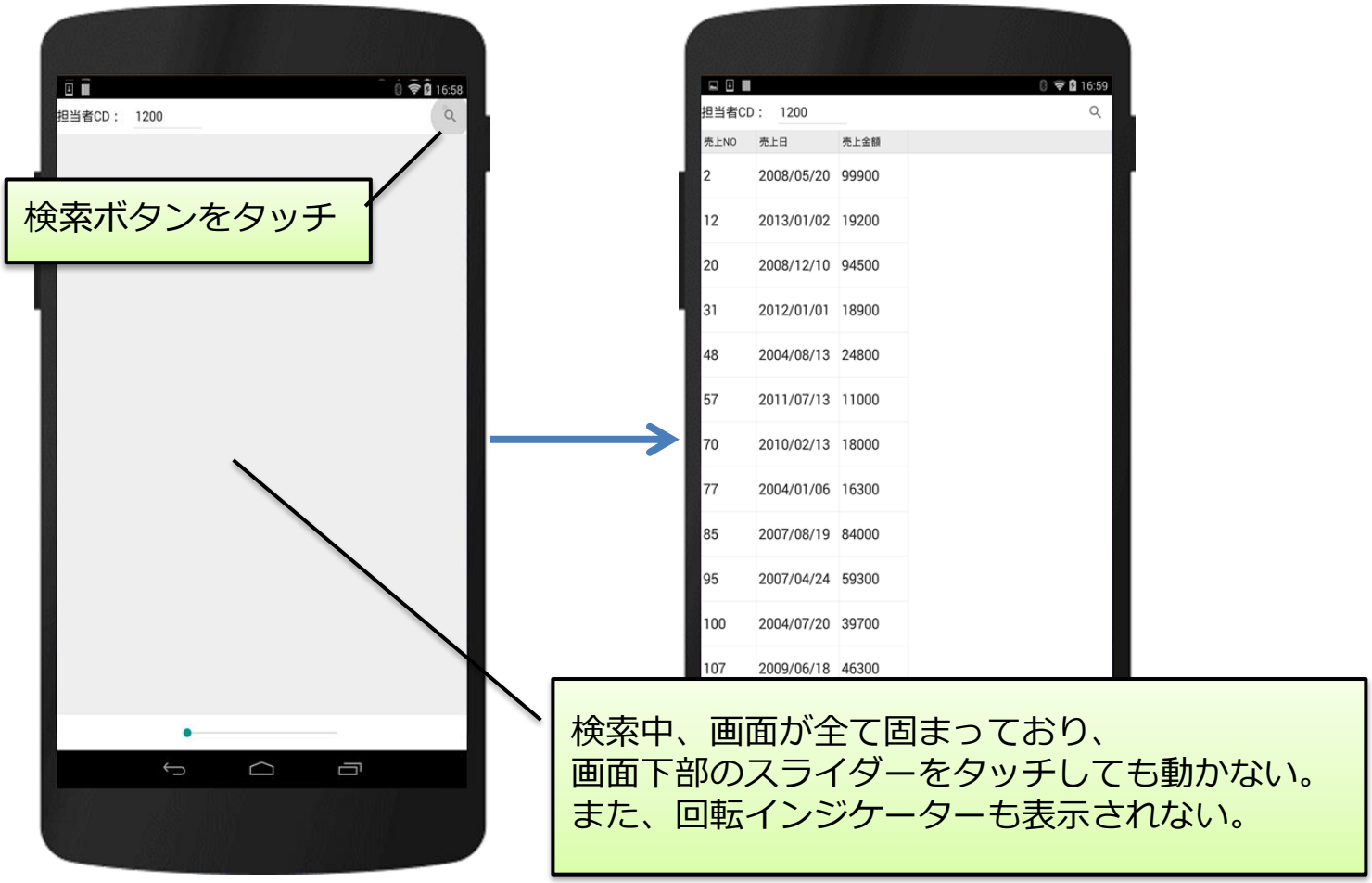
データセットを閉じる

インジケーターを非表示にして、StringGridを表示する

検索ボタンクリック処理にて、データの検索と画面へのセットを全て同じスレッド内で行っている。

1-1. タスクを使用したスレッド分割

- 実行



シングルスレッドアプリでは、処理中UIが固まってしまう。
特にモバイルは全画面表示の為、端末自体が固まった感覚になってしまう。

1-1. タスクを使用したスレッド分割

検索ボタンクリック時処理(1)　TTask追加

```
procedure TForm1.Button1Click(Sender: TObject);
var
  iRow: Integer;
begin
  Button1.Enabled := False;

  //インジケーターの表示
  AniIndicator1.Visible := True;
  AniIndicator1.Enabled := True;

  iRow := 1;

  //Grid初期化
  StringGrid1.RowCount := iRow;
  StringGrid1.Visible := False;
  StringGrid1.Cells[0, 0] := '';
  StringGrid1.Cells[1, 0] := '';
  StringGrid1.Cells[2, 0] := '';

  TTask.Run(procedure
  begin
    with ClientDataSet1 do
    begin
      Active := False;
      CommandText := 'SELECT * FROM FURDTP'
                   + ' WHERE URTNCD = ' + QuotedStr(EditCode.Text)
                   + ' ORDER BY URDTNO';

      Active := True;

      First;
```

データ抽出、StringGridへのセット部分を別のタスクとして記述

1-1. タスクを使用したスレッド分割

検索ボタンクリック時処理(2)　TTask追加

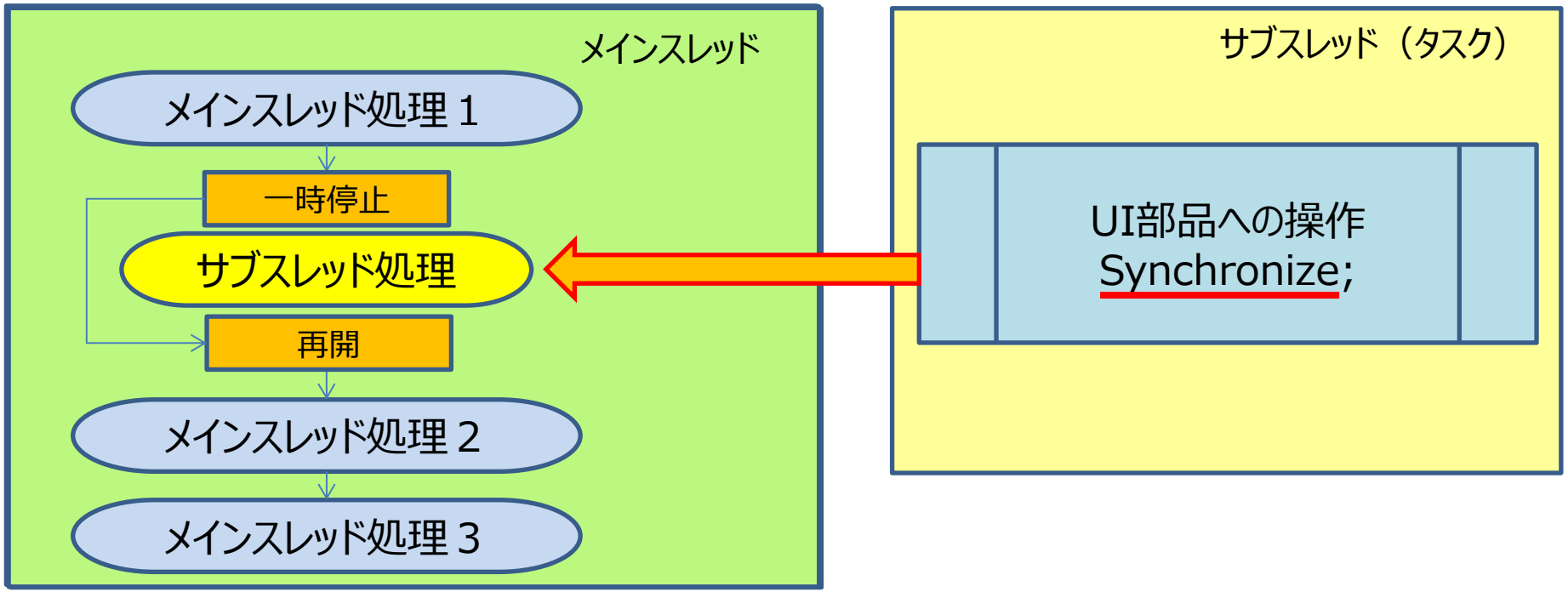
```
        while not Eof do
        begin
          StringGrid1.RowCount := iRow;
          StringGrid1.Cells[0, iRow - 1] := FieldByName('URDTNO').AsString;
          StringGrid1.Cells[1, iRow - 1] := FormatFloat('0000/00/00',
                                            FieldByName('URSYDT').AsInteger);
          StringGrid1.Cells[2, iRow - 1] := FieldByName('URKNGK').AsString;

          Next;
          inc(iRow);
        end;

        Active := False;

        TThread.Synchronize(nil, procedure
        begin
          //インジケーターの終了
          AniIndicator1.Visible := False;
          AniIndicator1.Enabled := False;
          StringGrid1.Visible := True;
          Button1.Enabled := True;
        end);
      end;
    end);
end;
```

データ抽出、StringGridへのセット部分を別のタスクとして記述

メインスレッドと異なるタスクから、UI操作（GUI部品の更新）を行う場合、Synchronizeメソッドを使用する

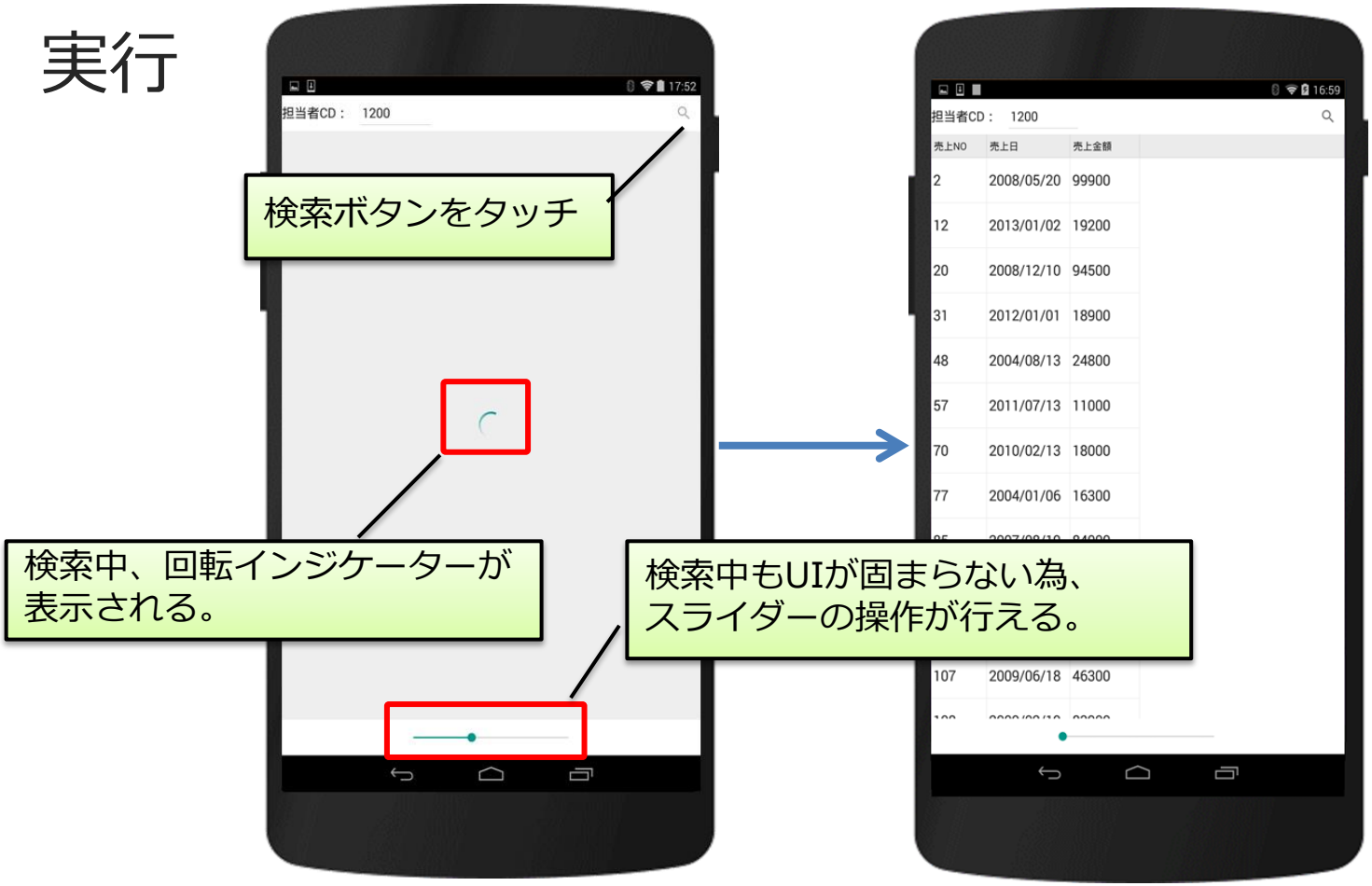
1-1. タスクを使用したスレッド分割

- マルチスレッドアプリの考慮点
  - ビジュアルコンポーネントが使用できるのは、メインスレッドのみ。タスクのスレッド側からビジュアルコンポーネントを操作したい場合Synchronizeメソッドを使用して、メインスレッド側を一時停止し、操作を行えるようにする必要がある。



1-1. タスクを使用したスレッド分割

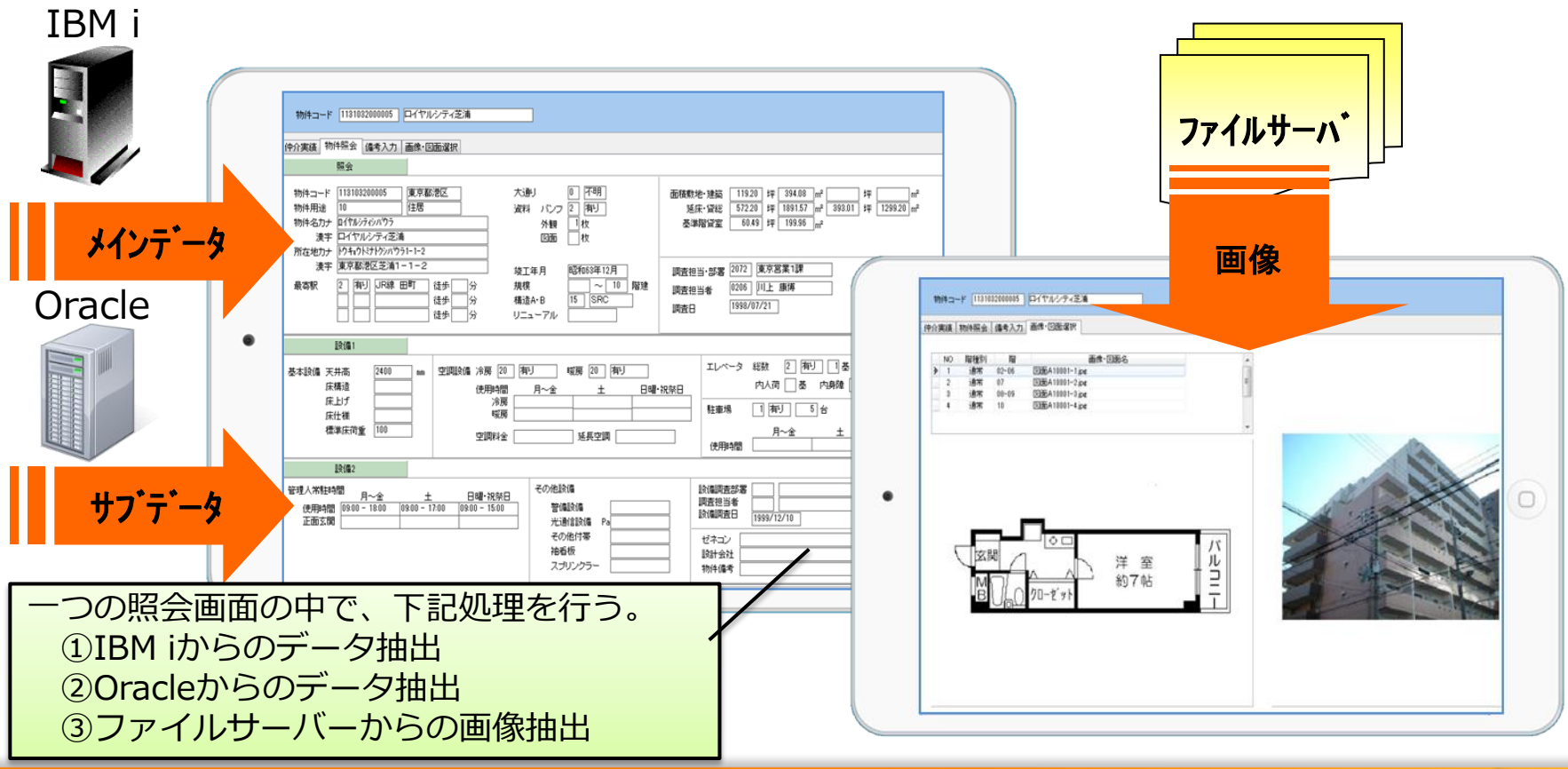
- 実行



**タスクを使用することで並列処理が可能になり、処理中でも
画面操作が可能になり、ユーザーはレスポンス悪化を感じにくくなる。**

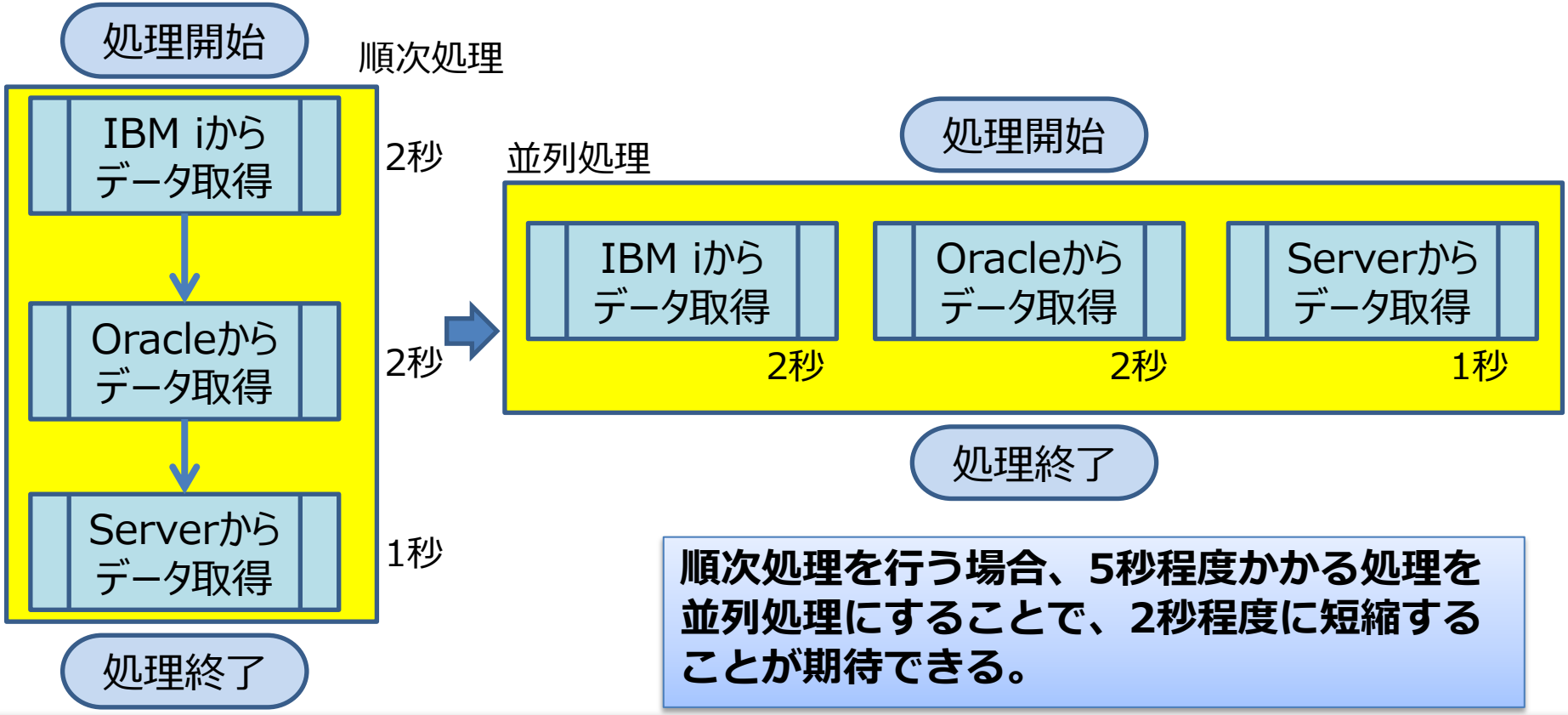
1-2. 複数タスクの並列処理

- 複雑な処理における並列処理の検討
  - ビジネスロジックの中にいくつかの処理がある場合、レスポンス向上を図ることは可能か？



1-2. 複数タスクの並列処理

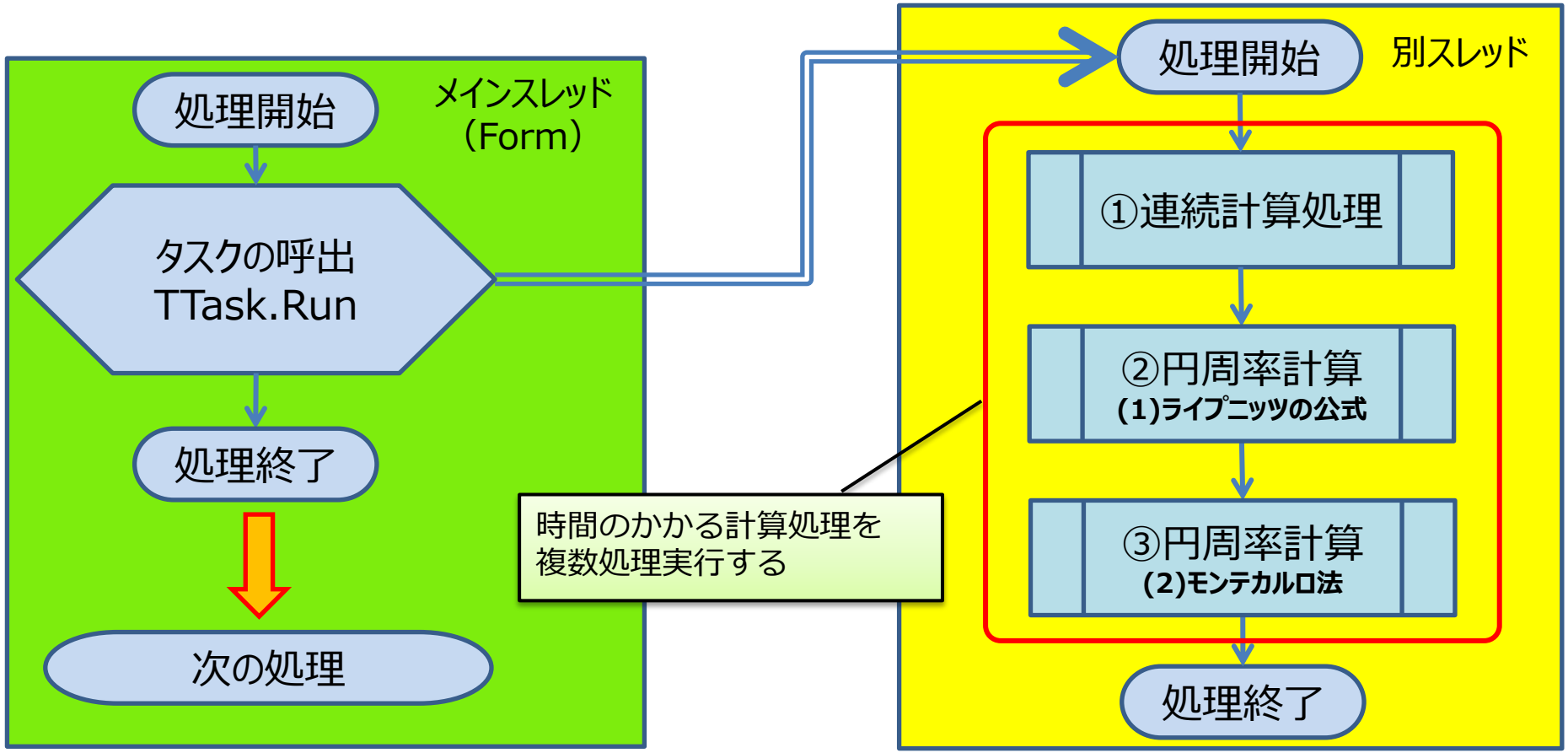
- 複雑な処理における並列処理の検討
  - それぞれの処理をタスクとして、処理を並列化することにより レスポンスの向上が期待できる。



順次処理を行う場合、5秒程度かかる処理を並列処理にすることで、2秒程度に短縮することが期待できる。

1-2. 複数タスクの並列処理

- 複雑なタスクのレスポンス向上
  - 時間のかかる計算処理を題材に複数処理の実行を検討する。



1-2. 複数タスクの並列処理

- 複雑なタスクのレスポンス向上
  - 画面レイアウト

Button1: TButton

Edit1: TEdit

AniIndicator1: TAniIndicator

Edit2: TEdit

Edit3: TEdit

【アプリの仕様】
[開始]ボタンをクリックしたら、3つの計算処理を実行し、結果を各Editに出力する。

全ての計算処理にかかった時間をミリ秒単位で画面下部に表示する。

LabelTime: TLabel

1-2. 複数タスクの並列処理

- 複雑なタスクのレスポンス向上
  - ①連続計算処理

```
//----- 計算 1
function TfrmMain.Calc1: Int64;
var
  i, j: Integer;
  Value: Int64;
const
  Max = 3000000;  //計算回数
begin
  //指定回数だけ、0～99の数値を足し算する
  Value := 0;

  for i := 1 to Max do
    for j := 0 to 99 do
      Value := Value + j;
  Result := Value;
end;
```

0+1+2+ … +98+99 の足し算を
指定回数だけ繰返し行う。

連続計算処理　単体の処理時間：約2秒

1-2. 複数タスクの並列処理

- 複雑なタスクのレスポンス向上
  - ②円周率計算（ライプニッツの公式を使用）

```
//----- 計算２
function TfrmMain.Calc2: Double;
var
  i: Integer;
  Value: Double;
const
  Max = 3000000;    //計算回数
begin
  // ライプニッツの公式で円周率を計算する
  Value := 0;

  for i := 1 to Max do
  begin
    Value := Value + (Power(-1, i - 1) / (2 * i - 1));
  end;
  Result := Value * 4;
end;
```

$$\frac{1}{1}-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\frac{1}{11}+\cdots$$
の総和を計算回数分求める。

ライプニッツの公式 $$\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots=\sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1}.$$

ライプニッツ公式　単体の処理時間：約7秒

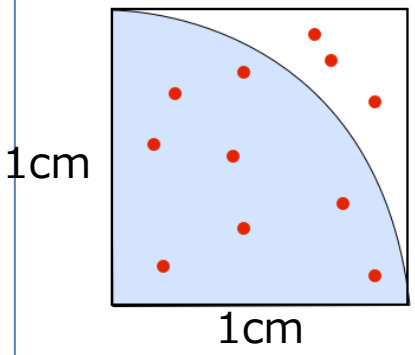
1-2. 複数タスクの並列処理

- 複雑なタスクのレスポンス向上
  - ③円周率計算（モンテカルロ法を使用）

```
//----- 計算３
function TfrmMain.Calc3: Double;
var
  i: Integer;
  x, y: Double;
  Count: Integer;
const
  Max = 3000000;    //計算回数
begin
  //モンテカルロ法で円周率を計算する
  Count := 0;

  for i := 1 to Max do
  begin
    x := Random;
    y := Random;
    if (x * x + y * y) < 1 then
      Count := Count + 1;
  end;

  Result := (Count / Max) * 4;
end;
```

多数のランダムな点の距離を使用して、近似値を求める。

モンテカルロ法



1．ランダムに点をうち、左下の原点からの距離が1未満の数を数える。(Xとする)

2．総数Nとすると、円周率は、X/N×4で近似される。

モンテカルロ法　単体の処理時間：約3秒

1-2. 複数タスクの並列処理

開始ボタンクリック時処理

```
uses System.Threading, System.DateUtils, System.Math;

procedure TForm1.Button1Click(Sender: TObject);
var
  StrTime: TDateTime;          //開始時刻
  ProcessTime: Integer;        //処理時間
  Ret1: Int64;                 //計算結果1
  Ret2: Extended;              //計算結果2
  Ret3: Extended;
begin
  //インジケーター表示
  Button1.Enabled := False;
  AniIndicator1.Enabled := True;
  AniIndicator1.Visible := True;

  StrTime := GetTime;  //処理開始時刻
  Randomize;

  TTask.Run(procedure
  begin
    //3つの計算処理を実行する
    Ret1 := Calc1;
    Ret2 := Calc2;
    Ret3 := Calc3;

    //処理終了
    TThread.Synchronize(nil, procedure
    begin
      //インジケーター終了
      AniIndicator1.Enabled := False;
      AniIndicator1.Visible := False;
      Button1.Enabled := True;

      //結果表示
      Edit1.Text := IntToStr(Ret1);
      Edit2.Text := FloatToStr(Ret2);
      Edit3.Text := FloatToStr(Ret3);
      //処理時間取得
      ProcessTime := MilliSecondsBetween(GetTime, StrTime);
      LabelTime.Text := IntToStr(ProcessTime);
    end);
  end);
end;
```

時間のかかる計算処理を別タスクとして実行

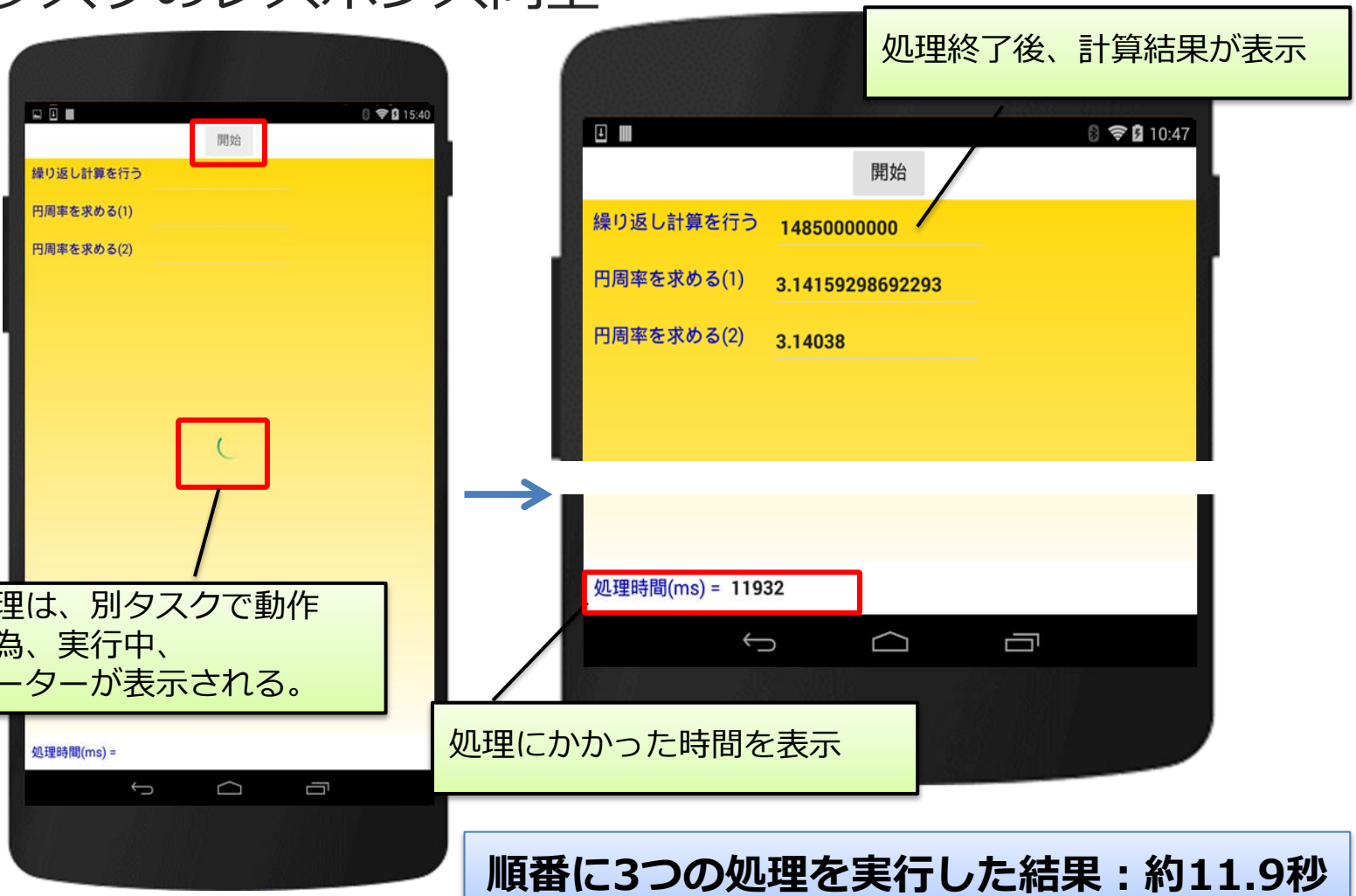
順番に3つの計算処理を実行

処理結果を画面に表示

処理時間を計算して表示

1-2. 複数タスクの並列処理

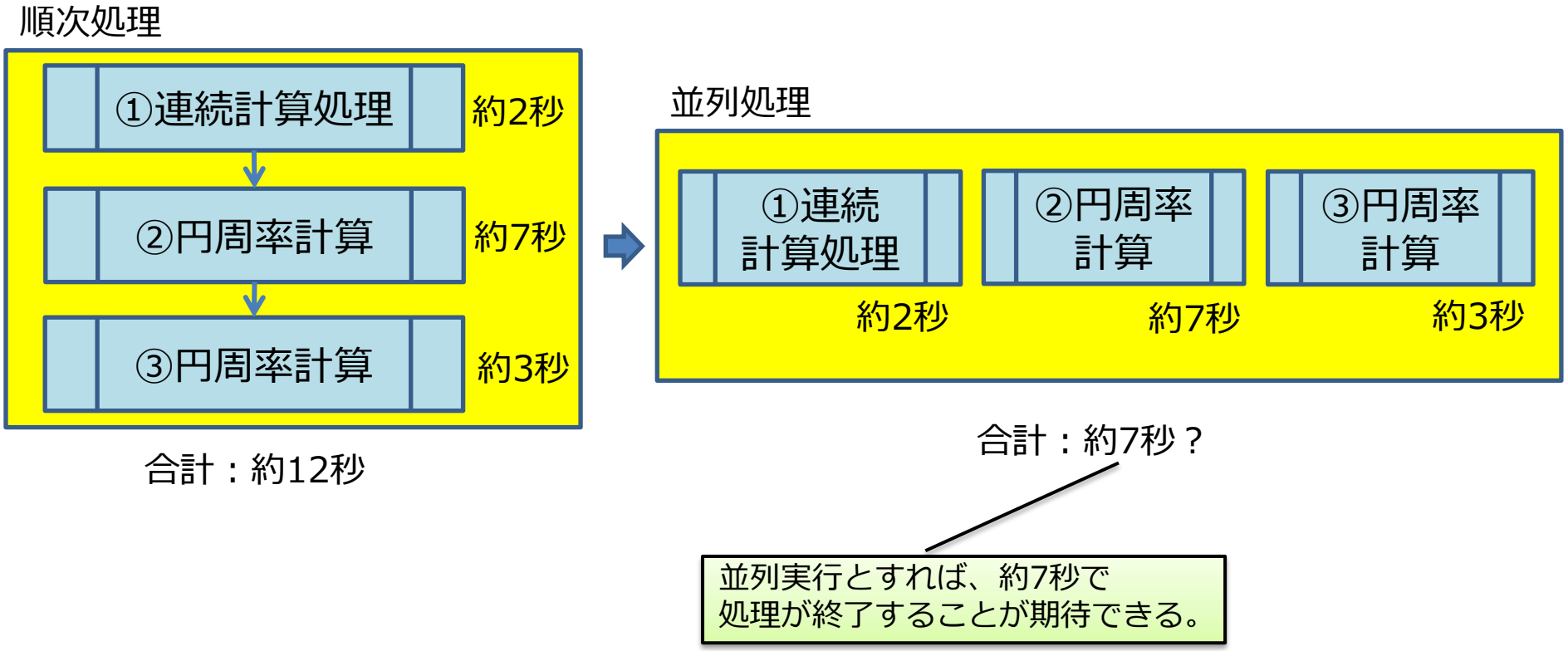
- 複雑なタスクのレスポンス向上
  - 実行



順番に3つの処理を実行した結果：約11.9秒

1-2. 複数タスクの並列処理

- 複数タスクの並列実行
  - 計算処理を同時に実行できるようタスクの並列化を行う。
    → タスクを配列化し、複数タスクを同時に実行する。



1-2. 複数タスクの並列処理

開始ボタンクリック時処理

```
procedure TForm1.Button1Click(Sender: TObject);
var
  StrTime: TDateTime;          //開始時刻
  ProcessTime: Integer;        //処理時間
  Ret1: Int64;                 //計算結果１
  Ret2: Extended;              //計算結果２
  Ret3: Extended;              //計算結果３
  Tasks: array[1..3] of ITask;  //タスクの配列
begin
  //インジケーター表示
  Button1.Enabled := False;
  AniIndicator1.Enabled := True;
  AniIndicator1.Visible := True;

  StrTime := GetTime;  //処理開始時刻
  Randomize;

  TTask.Run(procedure
  begin
    //3つの計算処理を実行する
    Tasks[1] := TTask.Create(procedure
    begin
      Ret1 := Calc1;
    end);
    Tasks[1].Start;

    Tasks[2] := TTask.Create(procedure
    begin
      Ret2 := Calc2;
    end);
    Tasks[2].Start;

    Tasks[3] := TTask.Create(procedure
    begin
      Ret3 := Calc3;
    end);
    Tasks[3].Start;

    //全てのタスク終了を待つ
    TTask.WaitForAll(Tasks);

    //処理終了
    TThread.Synchronize(nil, procedure
    begin
      //インジケーター終了
      AniIndicator1.Enabled := False;
      AniIndicator1.Visible := False;
      Button1.Enabled := True;

      //結果表示
      Edit1.Text := IntToStr(Ret1);
      Edit2.Text := FloatToStr(Ret2);
      Edit3.Text := FloatToStr(Ret3);
      //処理時間取得
      ProcessTime := MilliSecondsBetween(GetTime, StrTime);
      LabelTime.Text := IntToStr(ProcessTime);
    end);
  end);
end;
```

複数のタスクを管理する配列を定義

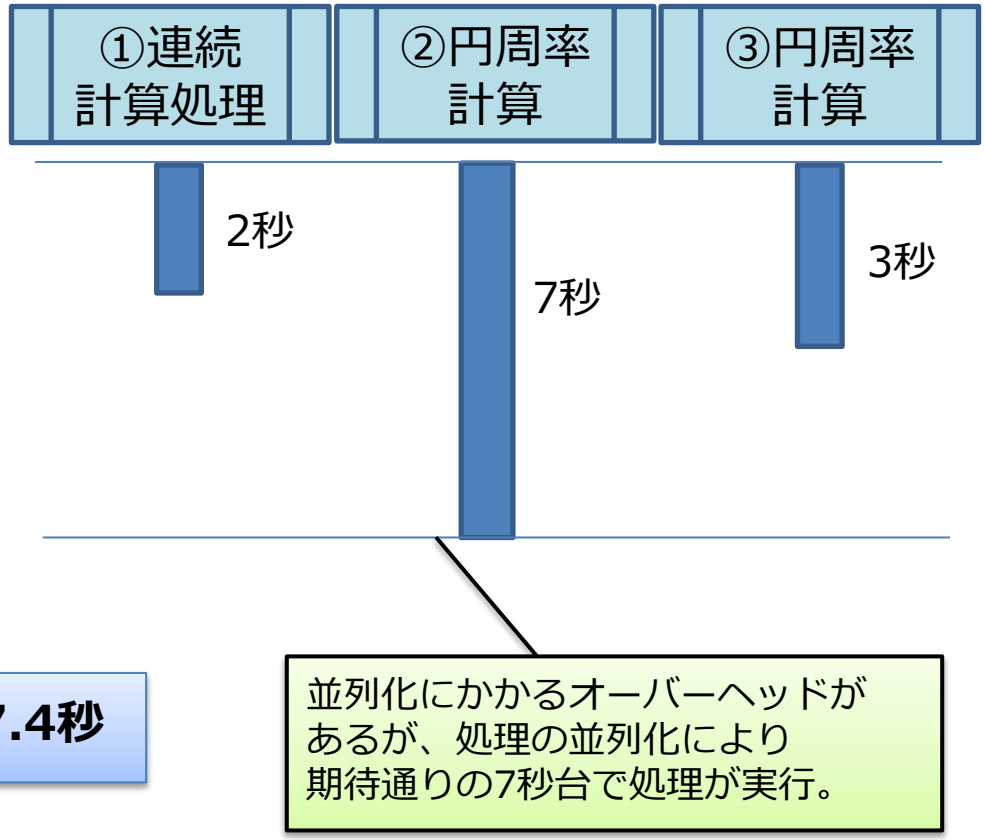
複数の処理を各タスク配列の要素として定義し、Startメソッドで実行する

配列のタスクが全て終了するまで待つ

1-2. 複数タスクの並列処理

- 複数タスクの並列実行
  - 実行

並列にタスクが実行



並列に3つの処理を実行した結果：約7.4秒

並列化にかかるオーバーヘッドが
あるが、処理の並列化により
期待通りの7秒台で処理が実行。

1-2. 複数タスクの並列処理

- 複数タスクを並列実行する時の留意点
  - 余り処理時間のかからないものをむやみに分割しない
    - 並列処理にかかるオーバーヘッドで、余計に遅くなってしまう。
  - 処理順序が重要なものを分割しない
    - 処理Aの結果をもって、処理Bを実行するようなものを並列にすると、処理待ちロック等が必要となる。

独立した複数の処理を同時に実行する場合に、並列処理は特に効果を発揮する。

２．アニメーションによる
レスポンス向上テクニック

2. アニメーションによるレスポンス向上テクニック

- モバイルアプリに関するレスポンスの課題
  - モバイルは、全画面で表示されるアプリとなる為、操作に対する応答がないと、実行しているかフリーズしているかわからない。
  - ボタン等にタッチしても打鍵感が無い為、応答に対するレスポンスがないと、正しく操作したかわからない。

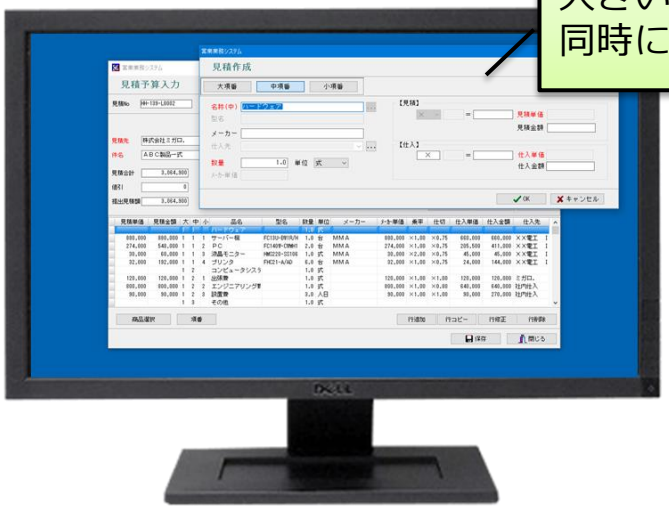
モバイルでは、上記課題の解決にアニメーション効果を使用することが多い。

今回、アニメーション効果によるレスポンス向上テクニックとして、下記をご紹介。

- 画面遷移におけるスライドアニメーション
- 処理待ち中を示すアニメーション効果

2-1. スライドアニメーションを使用した画面遷移

- Windowsアプリとモバイルアプリとの比較
  - Windowsアプリとモバイルアプリのフォームの違い



デスクトップは画面サイズが
大きい為、複数画面（フォーム）
同時に開く事が可能。

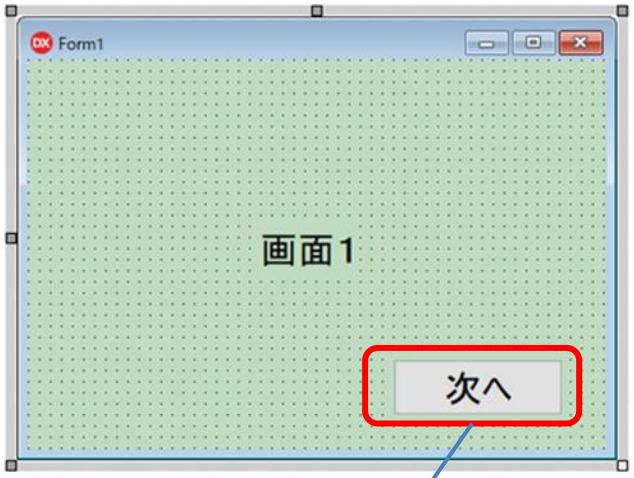
モバイルは画面サイズが
限定される為、常に現在実行
している画面（フォーム）
のみが開かれる。

それぞれ画面遷移は、どのような形になるか？

2-1. スライドアニメーションを使用した画面遷移

- Windowsアプリにおける画面遷移
  - VCLフォームアプリでの画面遷移

Form1



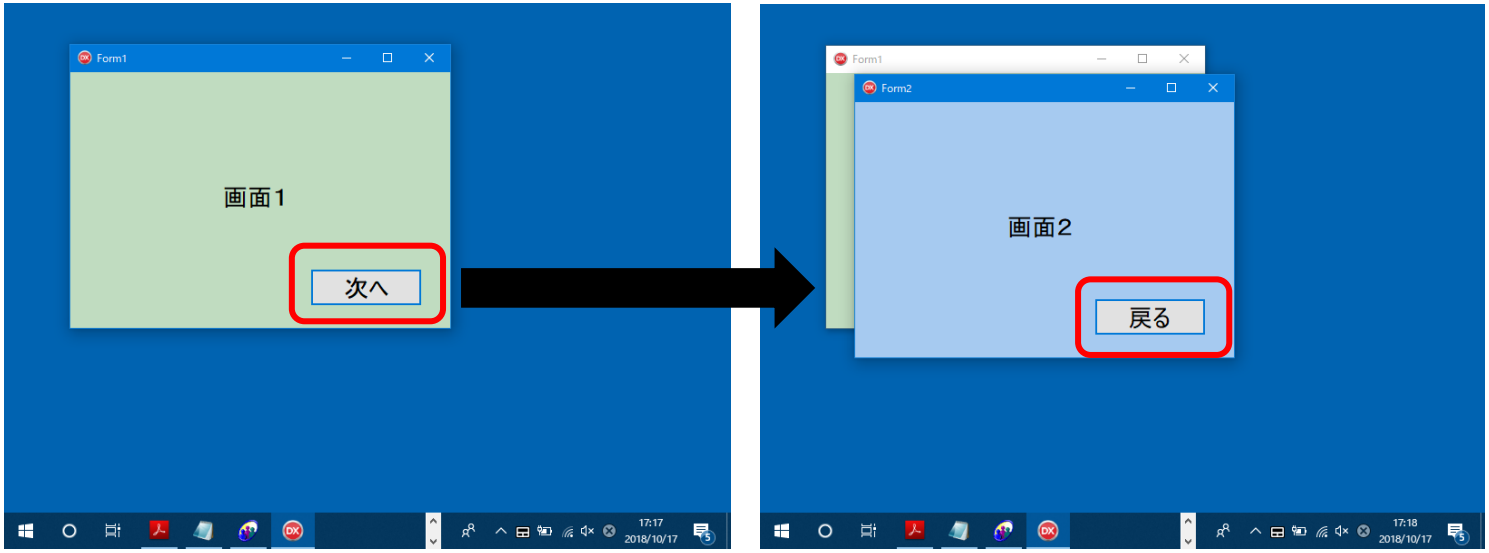
Form2


```
uses Unit2;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Show;
end;
```

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  Self.Close;
end;
```

2-1. スライドアニメーションを使用した画面遷移

- Windowsアプリにおける画面遷移
  - 実行



  - Windowsの場合、一つのフォームが一つのウィンドウとして表示される為、単純にShowメソッドで呼び出す画面遷移であっても、実行時の違和感は感じにくい。

モバイルアプリで同じプログラムはどう見えるか？

2-1. スライドアニメーションを使用した画面遷移

- モバイルアプリにおける画面遷移
  - 実行

  - モバイルでは、常に先頭のフォームのみが全画面に表示される仕様になっており、単純にShowメソッドで表示すると、唐突に画面が切り替わる為、違和感を感じやすい。

一般的なモバイルアプリではどうなっているか？

2-1. スライドアニメーションを使用した画面遷移

- メッセージアプリLINEの場合

- 一般的なモバイルアプリでは画面がスライドしながら切り替わることが多い

なぜ、画面をスライドさせるのか？

2-1. スライドアニメーションを使用した画面遷移

- 画面遷移にスライドが多用される理由
  - 画面遷移をするために操作したことをユーザーに視覚的に理解させる為にスライドを使用している。
  - 画面が唐突に切り替わると、2つの画面の構成が似ていたりすると、そもそも切り替えたことが分からない可能性がある。

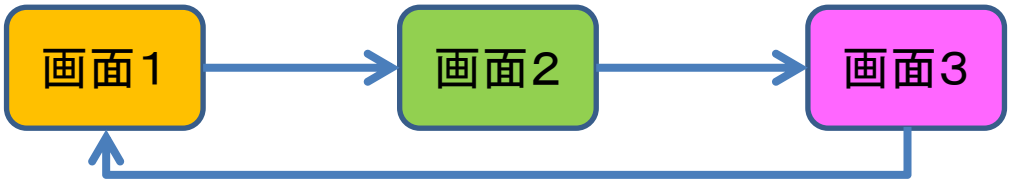
- 画面遷移のレスポンス
  - 通常のフォーム切替の場合、画面切替時に、フォーム生成(OnCreate)や表示(OnShow)を行うために処理時間がかかるが、スライド方式では事前読み込みを行う為、レスポンスが改善する。
  - 画面の切替にアニメーション効果を含むことで、処理時間を意識させなくできる為、感覚的なレスポンス改善を図る事もできる。

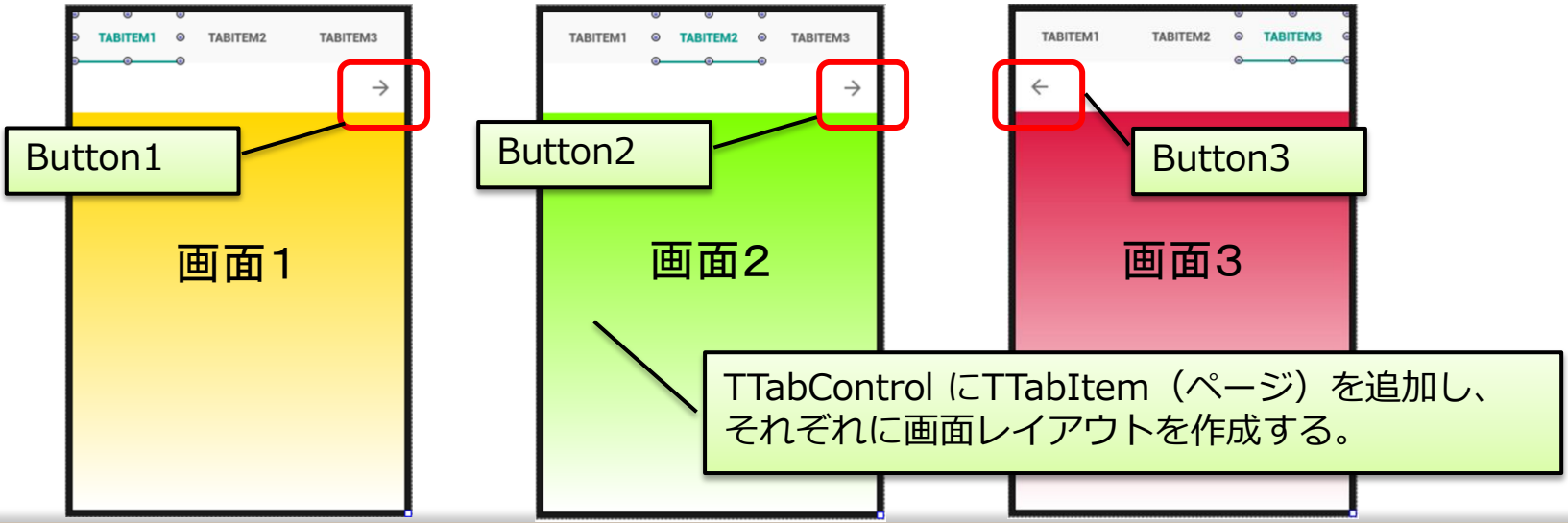
Delphi/400で、スライド効果を実現する方法を紹介！

2-1. スライドアニメーションを使用した画面遷移

- TTabControlを使用した画面遷移
  - タブ切替時にスライドするアニメーションが実現できる為、一つのタブを疑似フォームとして使用することで、スムーズな画面遷移を実現可能

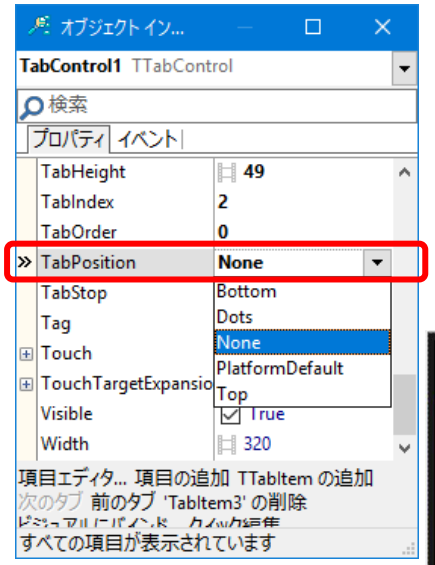
【画面遷移図】





2-1. スライドアニメーションを使用した画面遷移

- TTabControlを使用した画面遷移
  - レイアウト上、タブを非表示にする



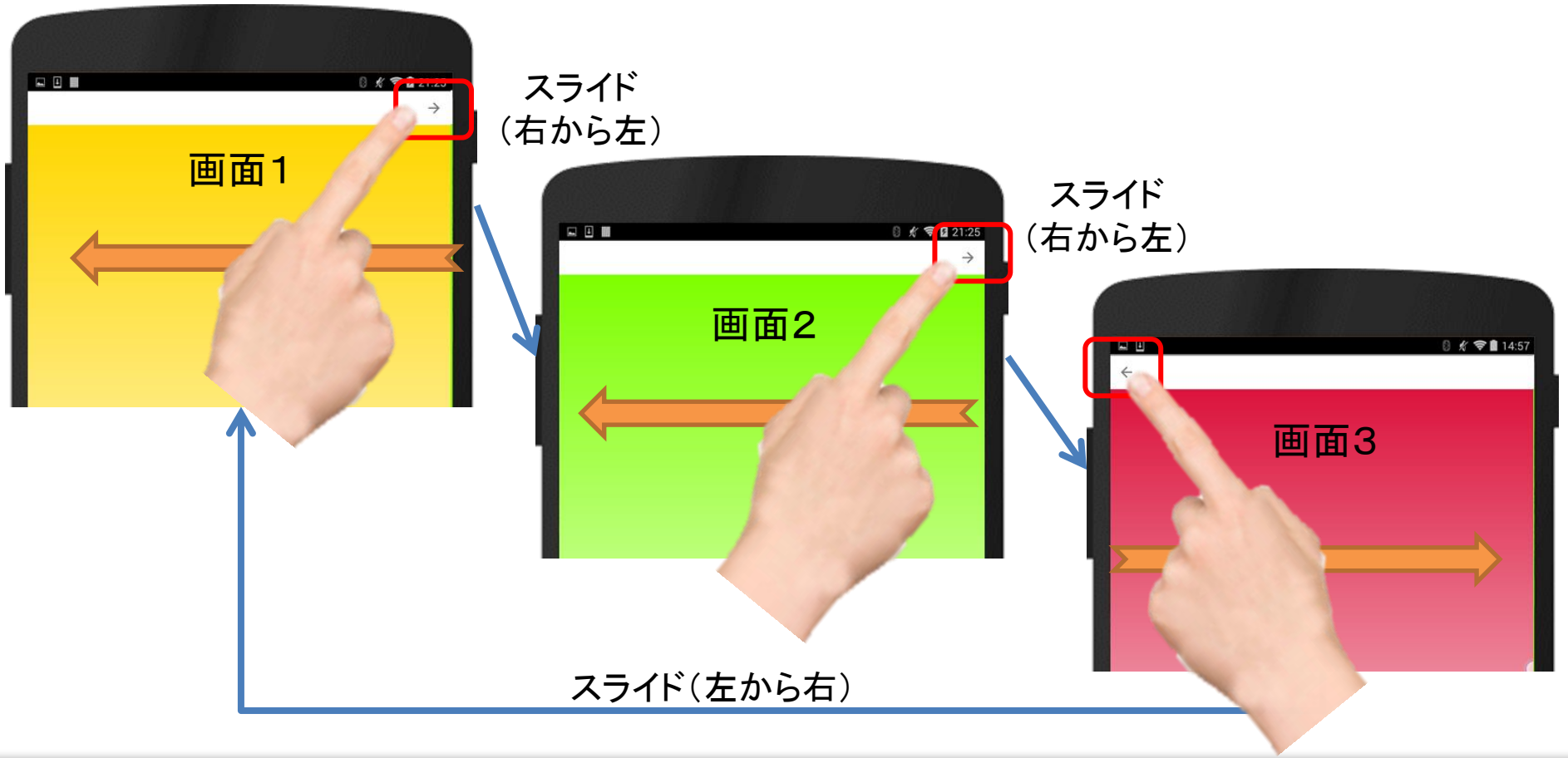

```
procedure TfrmMain.Button1Click(Sender: TObject);
begin
  TabControl1.SetActiveTabWithTransition(TabItem2,
    TTabTransition.Slide, TTabTransitionDirection.Normal);
end;

procedure TfrmMain.Button2Click(Sender: TObject);
begin
  TabControl1.SetActiveTabWithTransition(TabItem3,
    TTabTransition.Slide, TTabTransitionDirection.Normal);
end;

procedure TfrmMain.Button3Click(Sender: TObject);
begin
  TabControl1.SetActiveTabWithTransition(TabItem1,
    TTabTransition.Slide, TTabTransitionDirection.Reversed);
end;
```

設計画面では、点部分クリックで
TabItemの切替が可能。
実行時は、非表示の為コードが必要。

SetActiveTabWithTrainsitionメソッド
　タブの移動方法を定義
　　移動先タブ
　　スライドするかどうか
　　移動する向き
　　　Normal: 右から左にスライド
　　　Reversed：左から右にスライド

2-1. スライドアニメーションを使用した画面遷移

- TTabControlを使用した画面遷移
  - 実行

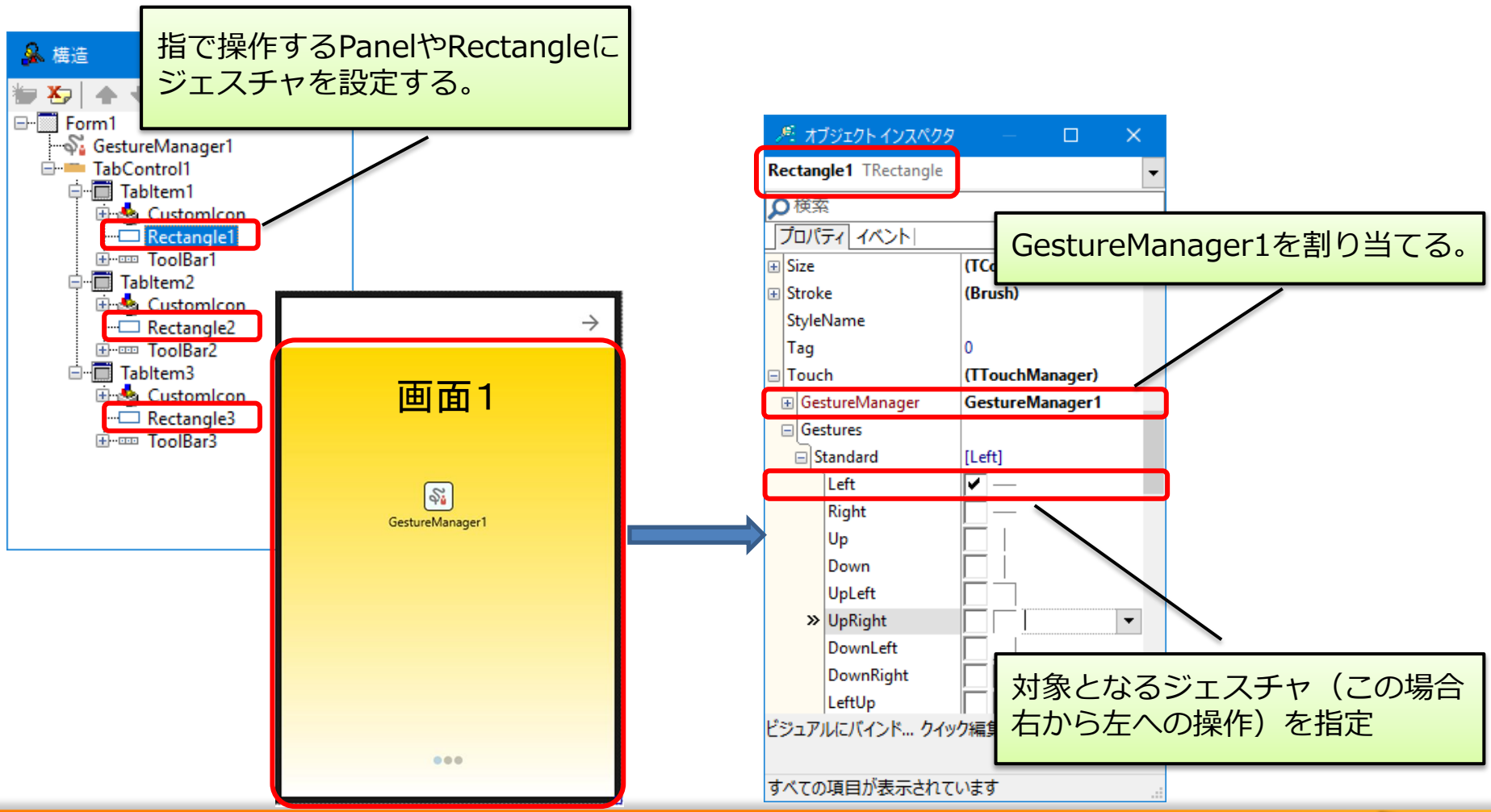


2-1. スライドアニメーションを使用した画面遷移

- TGestureManagerを使用したスワイプ操作の追加
  - モバイルでの画面遷移は、ボタンのタッチより、スワイプ操作の方が自然な操作感になる。
  - TGestureManagerを使用すれば、スワイプ動作に対応できる。

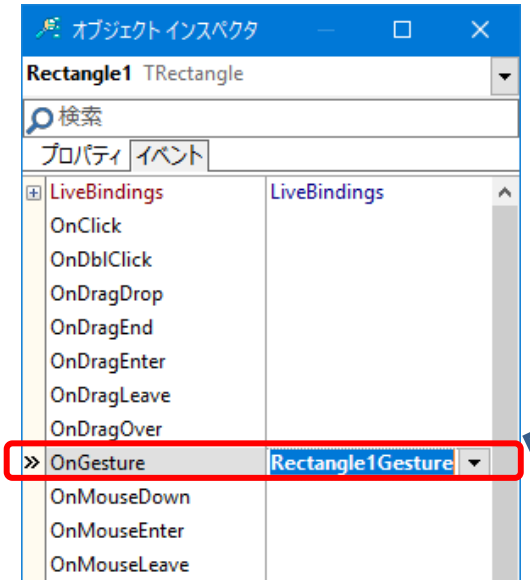
2-1. スライドアニメーションを使用した画面遷移

- TGestureManagerを使用したスワイプ操作の追加



2-1. スライドアニメーションを使用した画面遷移

- TGestureManagerを使用したスワイプ操作の追加



ジェスチャが行われた時に呼び出されるイベント。
2つ以上のジェスチャを判断する必要がある場合は、
引数のEventInfoの中にあるGestureIDを使用する。

```
procedure TForm1.Rectangle1Gesture(Sender: TObject;
  const EventInfo: TGestureEventInfo; var Handled: Boolean);
begin
  TabControl1.SetActiveTabWithTransition(TabItem2,
    TTabTransition.Slide, TTabTransitionDirection.Normal);
end;

procedure TForm1.Rectangle2Gesture(Sender: TObject;
  const EventInfo: TGestureEventInfo; var Handled: Boolean);
begin
  TabControl1.SetActiveTabWithTransition(TabItem3,
    TTabTransition.Slide, TTabTransitionDirection.Normal);
end;

procedure TForm1.Rectangle3Gesture(Sender: TObject;
  const EventInfo: TGestureEventInfo; var Handled: Boolean);
begin
  TabControl1.SetActiveTabWithTransition(TabItem1,
    TTabTransition.Slide, TTabTransitionDirection.Reversed);
end;
```

他のタブ上にあるRectangleも
同様の設定を実施

2-1. スライドアニメーションを使用した画面遷移

- TGestureManagerを使用したスワイプ操作の追加
  - 実行

2-1. スライドアニメーションを使用した画面遷移

- TTabControlを使用するメリットと課題

メリット

- スライドアニメーションが容易に作成でき、一つのユニットに処理が集約される為、画面切り替えの動作も早くなる。

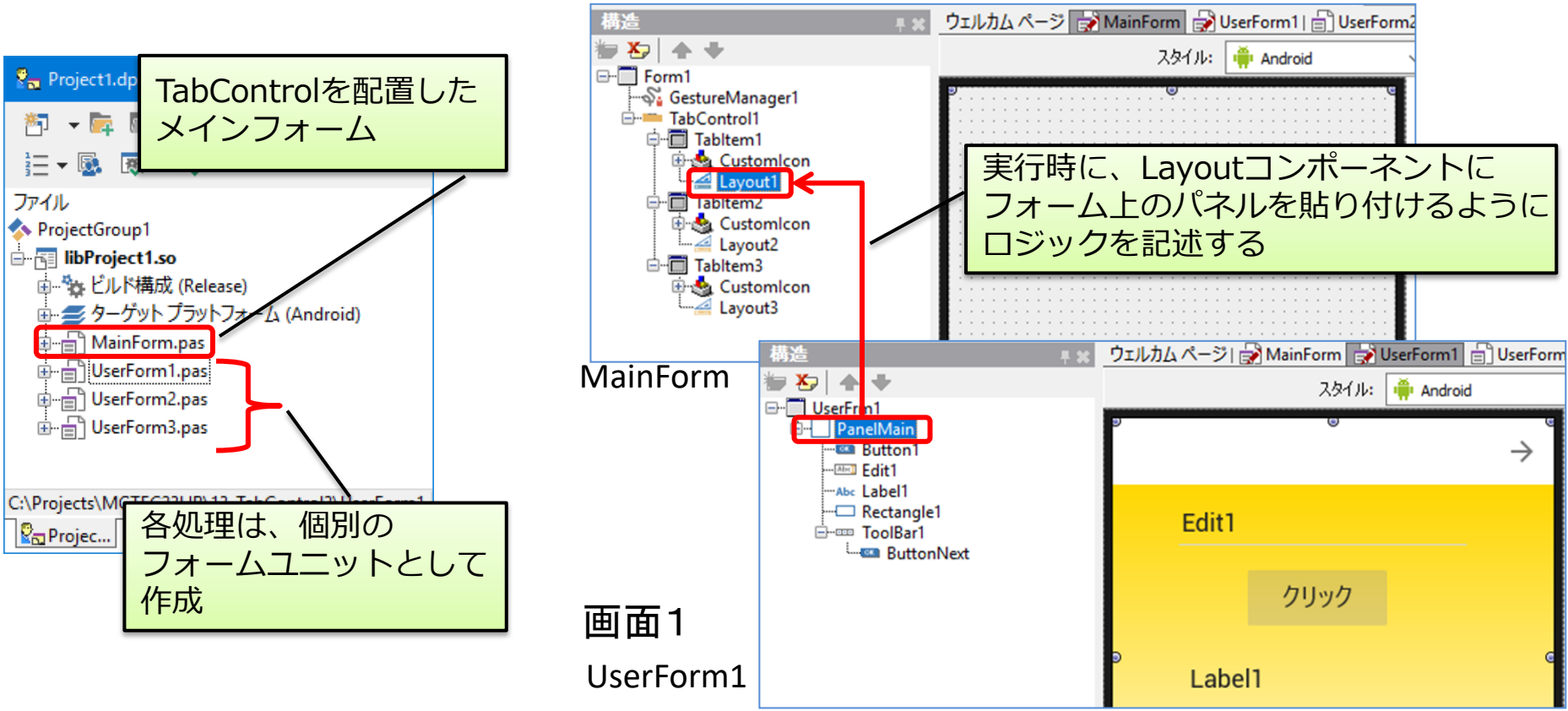
課題

- 一つのユニット上に各タブ（各画面）のロジックを作成しなければならない
  - 画面コンポーネント名の複雑化
    同じフォーム上のコンポーネントは、異なるタブであっても
    コンポーネント名を一意にする必要がある
  - ユニットコードの肥大化
    複数画面分のロジックを1つのユニットに記述
  - 複数人開発の不便さ
    一つのpasファイルだと同時に一人しか編集できない

画面ごとに別のユニットとして開発できないか？

2-1. スライドアニメーションを使用した画面遷移

- TLayoutを使用したユニット分割
  - TabItem上にLayoutコンポーネントを配置し、実行時に別ユニットで作成したフォーム上のPanelをLayoutに貼り付ける



2-1. スライドアニメーションを使用した画面遷移

- TLayoutを使用したユニット分割

MainForm 【宣言部】

```
type
  TForm1 = class(TForm)
    TabControl1: TTabControl;
    TabItem1: TTabItem;
    TabItem2: TTabItem;
    TabItem3: TTabItem;
    GestureManager1: TGestureManager;
    Layout1: TLayout;
    Layout2: TLayout;
    Layout3: TLayout;
    procedure FormCreate(Sender: TObject);
    procedure Layout1Gesture(Sender: TObject;
      const EventInfo: TGestureEventInfo; var Handled: Boolean);
    procedure Layout2Gesture(Sender: TObject;
      const EventInfo: TGestureEventInfo; var Handled: Boolean);
    procedure Layout3Gesture(Sender: TObject;
      const EventInfo: TGestureEventInfo; var Handled: Boolean);
  private
    { private 宣言 }
    procedure UserFrm1ButtonNextClick(Sender: TObject);
    procedure UserFrm2ButtonNextClick(Sender: TObject);
    procedure UserFrm3ButtonPriorClick(Sender: TObject);
  public
    { public 宣言 }
  end;
```

Layoutコンポーネントにジェスチャを割当

各フォーム上の画面遷移ボタンを押下した時のイベント手続きを定義

2-1. スライドアニメーションを使用した画面遷移

- TLayoutを使用したユニット分割

MainForm 生成時処理

```
uses UserForm1, UserForm2, UserForm3;

procedure TForm1.FormCreate(Sender: TObject);
begin
  TabControl1.ActiveTab := TabItem1;

  UserFrm1 := TUserFrm1.Create(Self);
  Layout1.AddObject(UserFrm1.PanelMain);
  UserFrm1.ButtonNext.OnClick := UserFrm1ButtonNextClick;

  UserFrm2 := TUserFrm2.Create(Self);
  Layout2.AddObject(UserFrm2.PanelMain);
  UserFrm2.ButtonNext.OnClick := UserFrm2ButtonNextClick;

  UserFrm3 := TUserFrm3.Create(Self);
  Layout3.AddObject(UserFrm3.PanelMain);
  UserFrm3.ButtonPrior.OnClick := UserFrm3ButtonPriorClick;
end;
```

Layoutコンポーネントに
フォーム上のパネルを追加

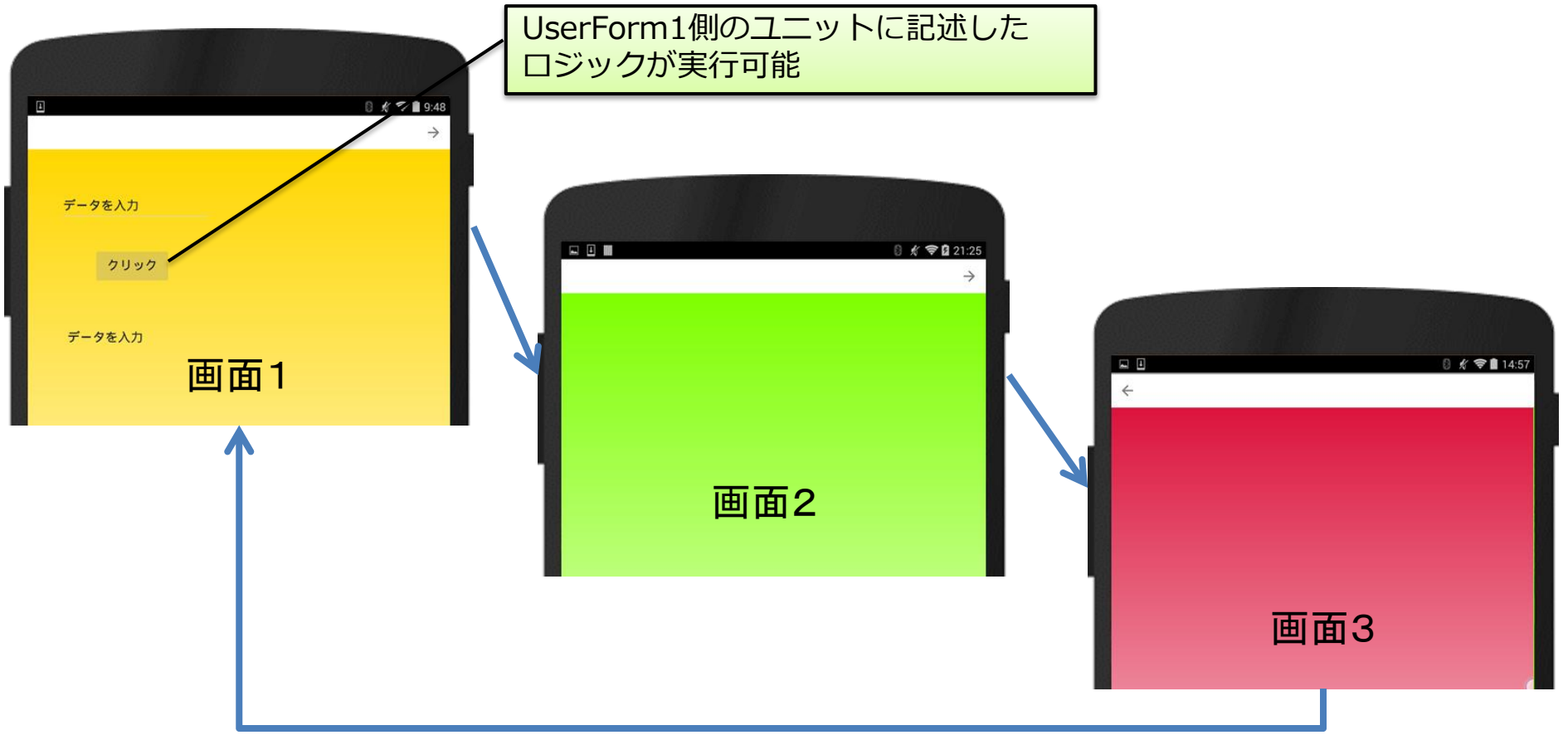
フォーム上の画面遷移ボタン
にイベントを割り当て

```
procedure TForm1.UserFrm1ButtonNextClick(Sender: TObject);
begin
  TabControl1.SetActiveTabWithTransition(TabItem2,
    TTabTransition.Slide, TTabTransitionDirection.Normal);
end;

procedure TForm1.UserFrm2ButtonNextClick(Sender: TObject);
begin
  TabControl1.SetActiveTabWithTransition(TabItem3,
```

TabItemの移動処理は、
MainForm側で記述

2-1. スライドアニメーションを使用した画面遷移

- TLayoutを使用したユニット分割
  - 実行



2-2. アニメーション効果の活用

- モバイルアプリにおけるアニメーション効果
  - 全画面で動作するモバイルアプリは、操作に対するアプリの応答がないと、端末が固まって見えてしまう。
    - → **画面遷移以外にも、モバイルアプリでは操作に対する応答としてアニメーション効果を使用することが多い。**

【例】アルバムアプリ

なぜ、アニメーション効果が有効なのか？

2-2. アニメーション効果の活用

- モバイルアプリにおけるアニメーション効果
  - タッチ操作が基本のモバイルでは、操作を行ったことを確実に伝えるための応答として、アニメーションを使用する。
  - 次の処理までに時間がかかるときに、何も動かないと固まったように見えてしまうため、先にレスポンスとしてアニメーションを実行することで、レスポンスの体感を向上させる効果もある。

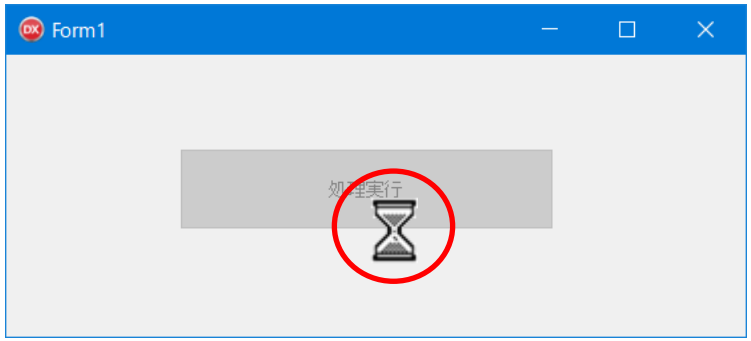
スペックが低く、アプリが全画面表示となるモバイルでは
アニメーション効果の活用で、レスポンスの体感向上が期待できる。

今回ご紹介するアニメーション効果

- 回転インジケーター
- プログレスバー
- アニメーションコンポーネント

2-2. アニメーション効果の活用

- Windowsアプリにおける処理実行の効果
  - マウスカーソルを砂時計にすることで実行中を表現できる。



マウスカーソルを砂時計に変更

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  Button1.Enabled := False;

  Screen.Cursor := crHourGlass;

  //時間のかかる処理
  for i := 1 to 100000000 do
    Application.ProcessMessages;

  Screen.Cursor := crDefault;

  Button1.Enabled := True;
end;
```

マウスカーソルを元に戻す

マウスカーソルの無いモバイルの場合、どのように処理中を表現するか？

2-2. アニメーション効果の活用

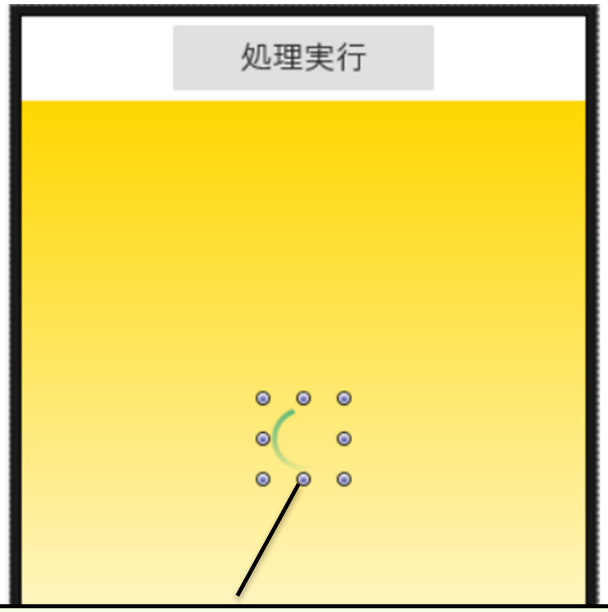
- モバイルアプリにおける処理実行の効果
  - マウスカーソルが無いモバイルでは、回転インジケーターや
    プログレスバー使用したアニメーション効果で表現する。

TAniIndicator
プロパティ
Enabled:　True時回転する

TProgressBar
プロパティ
Min：　最小値
Max：最大値
Value：現在値
機能
MinからMaxまでの範囲について、
Valueプロパティで進捗を表現

2-2. アニメーション効果の活用

- TAniIndicatorの使用
  - 実行中を表す回転インジケーターを表示



AniIndicator1: TAniIndicator
プロパティ
Align：Center (画面の真ん中)
Enabled：False
Visible：False

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  Button1.Enabled := False;

  AniIndicator1.Visible := True;
  AniIndicator1.Enabled := True;

  //時間のかかる処理
  for i := 1 to 1000 do
    Application.ProcessMessages;

  AniIndicator1.Enabled := False;
  AniIndicator1.Visible := False;

  Button1.Enabled := True;
end;
```

回転インジケータを表示して有効にする（回転する）

回転インジケータを非表示にする

2-2. アニメーション効果の活用

- TAniIndicatorの使用
  - 実行

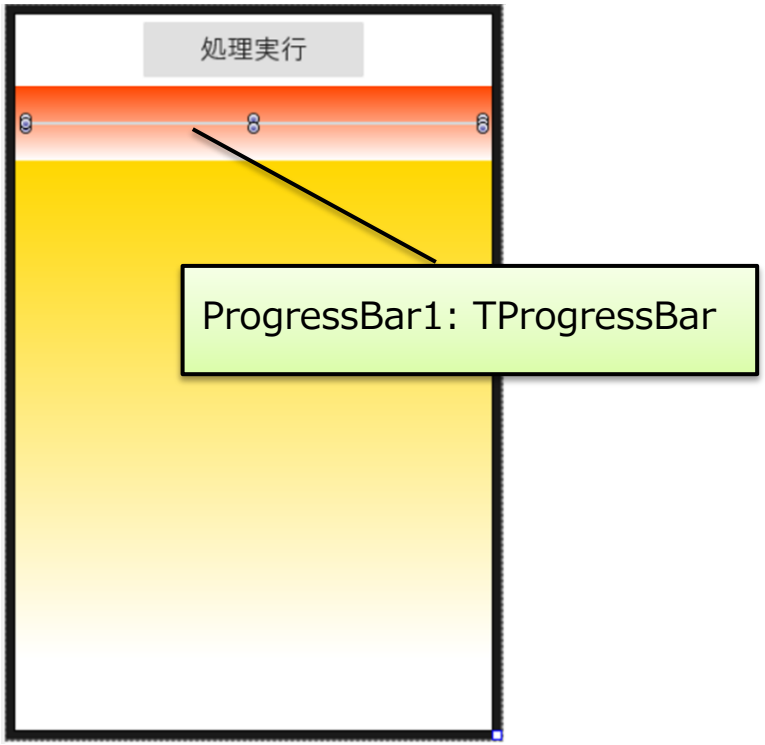
処理実行中、回転インジケーターが表示され、回転アニメーションが実行される。

- 回転インジケーターは、処理ボリュームが不確定な場合に、主に使用される

  （例）
  - IBM i 等DBサーバーにSQLで問合せする。
  - HTTPでWEBサーバーにアクセスし応答を待つ。

2-2. アニメーション効果の活用

- TProgressBarの使用




```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  iVal: Integer;
begin
  Button1.Enabled := False;

  AniIndicator1.Visible := True;
  AniIndicator1.Enabled := True;

  ProgressBar1.Min := 0;      //最小値
  ProgressBar1.Max := 100;    //最大値
  ProgressBar1.Value := 0;    //現在値
  iVal := 0;

  //時間のかかる処理
  for i := 1 to 1000 do
  begin

    if (i mod 10) = 0 then
    begin
      iVal := iVal + 1;
      ProgressBar1.Value := iVal;
    end;

    Application.ProcessMessages;
  end;
  AniIndicator1.Enabled := False;
  AniIndicator1.Visible := False;

  Button1.Enabled := True;
end;
```

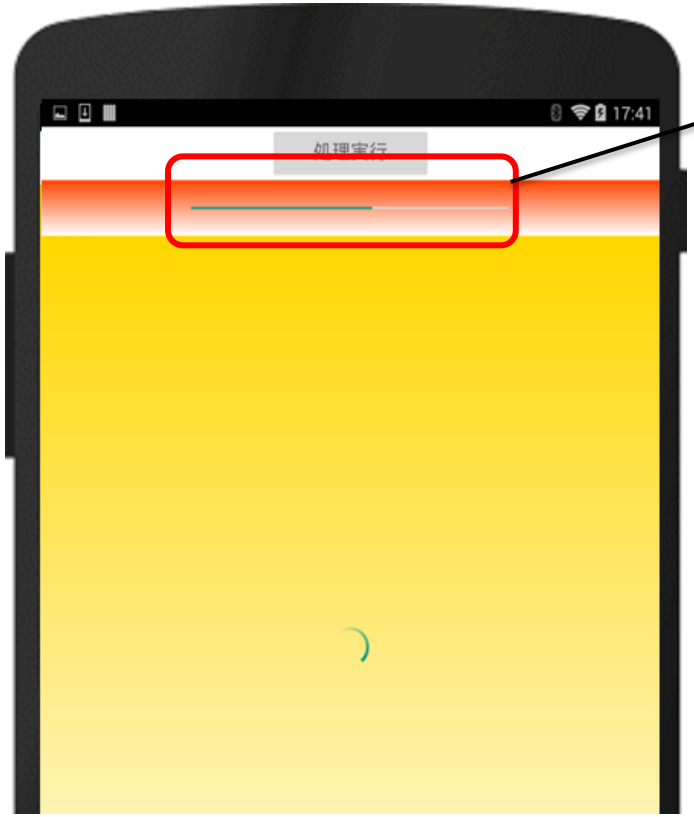
プログレスバーの初期化

処理回数1000回

10回に1回
バーの位置を更新

2-2. アニメーション効果の活用

- TProgressBarの使用
  - 実行



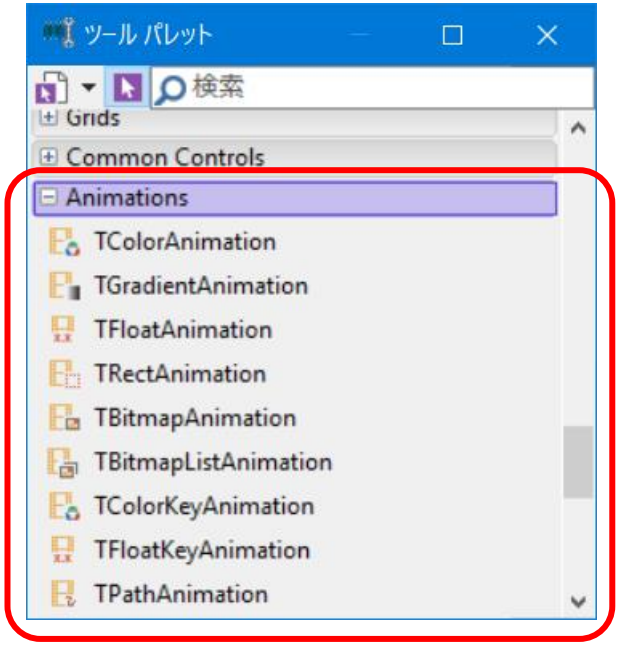
プログレスバーの進捗割合が徐々に更新される

- プログレスバーは、
  処理ボリュームが確定されて
  いる場合に主に使用される
  （例）
  - 繰返し処理の回数が予め
    決まっている場合 (Forループ等)
  - SQL問合せ結果の件数が予め
    分かっている場合

2-2. アニメーション効果の活用

- アニメーションコンポーネント
  - プロパティの値を変更する仕組み
  - 時間の経過にあわせて、プロパティの値を変化させていく。
  - 任意のタイミングで開始/終了したり、トリガーによって実行する。

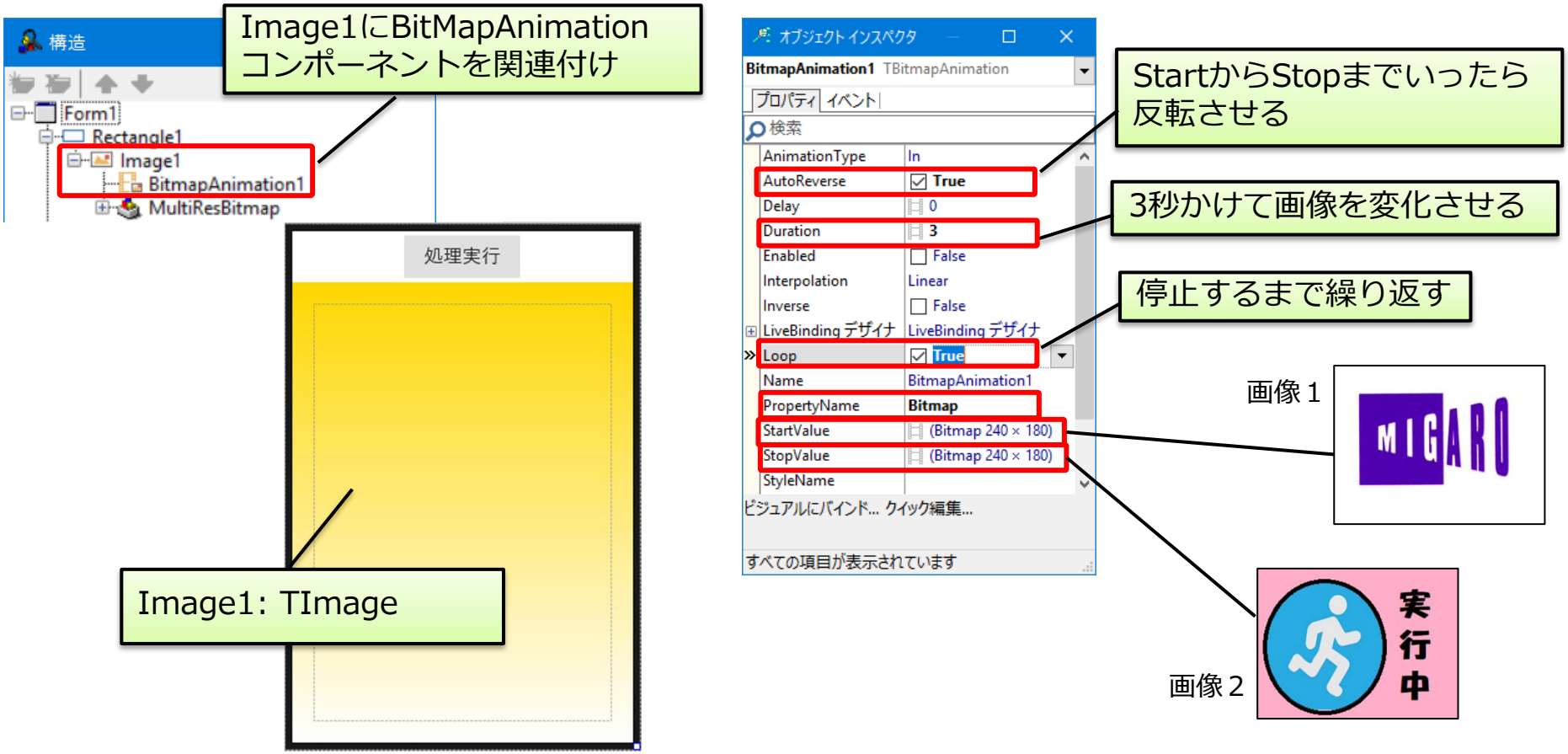
  - アニメーションコンポーネントの主なプロパティ
    - StartValue – 開始時の値
    - StopValue – 終了時の値
    - StartFromCurrenct（Boolean）
      – True：現在の値から開始
    - PropertyName – 対応するプロパティ
    - Duration – アニメーション時間
    - Loop – 停止まで繰り返し実行
    - AutoReverse – 終了後向きを反転する



**アニメーションコンポーネントを使用すれば、独自の処理待ち
アニメーション効果も作成できる。**

2-2. アニメーション効果の活用

- TBitmapAnimationの使用
  - 実行中に画像を変更するアニメーションを実行



2-2. アニメーション効果の活用

- TBitmapAnimationの使用
  - [処理実行]ボタン クリック処理

```
procedure TfrmMain.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  Button1.Enabled := False;

  //アニメーションをスタートする
  BitmapAnimation1.Start;

  //時間のかかる処理
  for i := 1 to 1000 do
    Application.ProcessMessages;

  //アニメーションを停止する
  BitmapAnimation1.StopAtCurrent;

  Button1.Enabled := True;
end;
```

アニメーションを開始する。

アニメーションを終了する。
（画像は現在の絵で停止）

1-2. アニメーション効果の活用

- TBitmapAnimationの使用
  - 実行

画像アニメーションを使用した独自の処理待ち画面の実行

まとめ

■ まとめ

1. 並列処理によるレスポンス向上テクニック
   - TTaskによるメインスレッド（UI処理）とサブスレッド（ロジック）の分割
   - 複雑な処理を並列稼働させることで、処理効率が向上

2. アニメーションによるレスポンス向上テクニック
   - TTabControlを使用することで、スムーズな画面遷移を実現
   - TGestureManagerを組み込むことにより自然な操作感を実現
   - ユニットを分割することで開発効率向上を実現
   - TAniIndicatorやTProgressBarを使用することで、処理待ちに対する体感レスポンスを向上
   - アニメーションコンポーネントを使用すれば、独自に変化する処理待ち画面も作成できる