コンピュータ科学科

情報システム系

卒業論文

単純な並列プログラムに対する GUI を備えた可逆デバッガ

2022 年 2 月

101830125　近藤諒一

概要

本研究では，単純な並列プログラムに対する逆方向実行環境上に GUI を備えたデバッガを作成する．単純な並列プログラムとして，並列の入れ子構造がなく，while ループや if 文そして並列ブロックを持つような単純なプログラミング言語を対象とする．並列プログラムのデバッグでは複数あるプロセスの実行順が実行環境に依存するため，不具合が発生した際に再実行による問題点の同定が困難である．この問題に対して可逆な抽象機械による並列プログラムの実行による可逆実行環境が提案されている．この実行環境ではソースプログラムに対応した抽象機械の命令単位で順方向実行と逆方向実行を行う．順方向実行の際に逆方向実行に必要な情報をスタックに保存する．順方向実行の命令列を逆方向実行の命令列に変換し，可逆実行を実現する．

池田らによって提案された実行環境をもとに並列プログラムに対する可逆デバッガを実現する．可逆デバッガに必要な機能として以下の機能を実現する．

・実行途中で順方向実行モードと逆方向実行モードを切り替える機能．
・順方向実行と逆方向実行におけるブレークポイント機能．

以上の機能により，並列プログラムの不具合を再実行することなく特定し，効率的なデバッグを可能にする．GUI によって実行状況を可視化することで，プログラムの並列実行の状況が視覚的に理解でき，直観的な実行の解析を可能とする．

Abstract

In this study, we present an implementing of a reversible debugger with GUI for simple concurrent programs. Simple concurrent programs have no nesting structures.

In debugging concurrent programs, it is difficult to verify the problem by replaying the faulty execution because the execution order of multiple processes may vowing each time where the debugging method is different from that of sequential programs.

the runtime proposed by Ikeda at all, executes a program in both directions performed by abstract machines concurating. converting a forward stack machine code into a backward stack machine code, the abstract machine executes reversible debugging. Information of the concurrent execution is preserved by storing the history of forward execution.

In order to enable reversible debugging, we provide the following functions for our reversible debugger: an execution mode switching function that switches between forward and reverse execution, and a function to save the previous history and perform the same forward execution as the previous time.

The debugger is equipped with GUI that helps the user to visually comprehead the multiple processes the configurations of.

目次

1 はじめに

並列プログラムの実行において複数のプロセスが同時に計算処理を行うため，実行ごとにプロセスの実行順が異なるのが一般的である．そのため，単一のプロセスのみを使用するプログラムのように，問題が起きた場合に再実行によるデバッグでは状況を再現することが難しい．そこで，先行研究[1]では可逆実行環境による可逆デバッグが研究されている．

可逆デバッグとは，順方向実行によって生じた問題点から逆方向実行を行って問題の原因を解析するデバッグ手法である．本研究では，順方向実行とはプログラムを先頭から順方向に実行することを言い，逆方向実行とは順方向実行によって実行されたプログラムを遡るように逆方向に実行することを言う．通常，順方向のみの実行では捨てられる逆方向実行に必要な情報を順方向実行が行われる際に全て保存する．

本研究では並列の入れ子構造がなく，while ループや if 文，そして並列ブロックを持つプログラムを対象とする．単純な命令を実行する抽象機械によって実行する．並列のスタックマシンコードに変換された命令列構造を取得した後に，各並列プロセスに対応するスタックマシンを生成し逐次命令列を当てはめる．逆方向実行時には対応する順方向実行用のスタックマシンコードを変換させる．

本研究で作成したデバッガーには，ブレークポイントの目標として指定したプログラムカウンタに到達するまで順方向実行または逆方向実行を行う機能と，与えた条件式を満たすまで順方向実行または逆方向実行を行う機能を実装した．さらに，逆方向実行時に履歴の一部を保存することで逆方向実行によって一度巻き戻った部分の順方向実行を再現できる機能を実装した．

2 可逆並列プログラム

2.1 並列プログラム言語

本研究で扱う単純な並列プログラムを以下のように定義する.

ここで，$()^+$は 1 回以上，$()^*$は 0 回以上の繰り返しを意味する.

P ::= DQR | DQ par {Q}({Q})$^+$R
D ::= (var X;)*
R ::= (remove X;)*
Q ::= (S;)*S
S ::= skip | X = E | if C then Q else Q fi | while C do Q od
E ::= X | n | E op E | (E)
C ::= B| C&&C | not C | (C)
B ::= E==E | E>E
(X : 変数 n, op : {+,×,−})

このプログラムでは逐次実行するブロックを並列合成する par 命令によって並列ブロックを持つ. 各逐次命令列のブロックは上から順にプロセス 1, 2, …と定義する.

プログラム末尾の remove 命令はプログラム冒頭の var 命令に対応し，var a; var b;の順に宣言された変数を remove b; remove a;の順に削除する.

2.2 動作的意味

２．１で定義した言語は while ループや if 文，及び並列ブロックを持つような単純なプログラムである. 並列ブロックは par 命令の数によって複数存在し，各並列ブロックが他の並列ブロックと非同期的に実行される. 非同期的に実行されたプログラムは再実行によって同じ手順や同じ結果を得ることが難しいため，デバッグには工夫が必要である.

さらに，並列プログラムではデッドロックが起こる可能性がある. デッドロックが発生するとそこからどれだけ実行を行ってもプログラムを最後まで終了することができないので，プログラムが終了した後の実行結果から問題を検証することはできない.

そこで本研究では可逆デバッグによる並列プログラムの検証について考える.

抽象機械の命令セットを以下に示す.

- nop: 何も行わず，プログラムカウンタの値を 1 増やす.
- ipush: スタックにオペランドの即値をプッシュする.
- load: オペランドの変数番地を読み出し，そこに格納されている値をスタックの先頭に置く.

- store：プロセス番号とオペランドの変数番地の値を保存した後，スタックの先頭から値をポップしオペランドの変数番地にその値を保存する．
- jpc：スタックの先頭をポップしその値が 0 ならオペランドの値を現在のプログラムカウンタの値とする．ポップした値が 0 以外ならプログラムカウンタの値を 1 増やし次の命令に移動する．
- jmp：無条件にオペランドの値を現在のプログラムカウンタの値とする．
- op： スタックの先頭から値を 2 回ポップし，その二つの値に対してオペランドの演算番号(0,1,2,3,4,5,6)に対してそれぞれ(＋,×,―,＞,＝＝,&&,not)の演算を行う．
- label：逆方向実行のための情報をスタックにプッシュする．オペランドの値 n は全命令の数．
- rjmp：スタックから値をポップし，その値を現在のプログラムカウンタの値とする．Jmp 命令の逆方向に，順方向実行を戻るようにジャンプする．
- restore：スタックから値をポップしその値をスタックの変数番地に格納する．
- alloc：変数を追加し，変数を初期化する．
- free：オペランドの値に対応する変数を削除する．
- par： オペランドの値が 0 なら並列プロセスの開始を意味し，1 なら並列プロセスの終了を意味する．

可逆実行のスタックマシンコードは順方向実行用と逆方向実行用に変換することができる．各スタックマシンコードの順方向実行用から逆方向実行用への変換は以下のように定義される．ただし(v,a,n)はそれぞれ(変数番地，ジャンプ先のプログラムカウンタの値，全命令の数)である．

- store v → restore v
- jpc a → label 0
- jmp a → label n
- label n → rjmp 0
- par 0 → par 1
- par 1 → par 0
- alloc v → free v
- free v → alloc v
- その他の命令は nop 0 に変換される．

また，以上の変換を逆方向に行うことで，各スタックマシンコードは逆方向実行用から順方向実行用に変換することができる．

2.3 可逆実行環境

本研究では，並列プログラムの実行を順方向に実行することに加えて，逆方向に行うことも考える．スタックを用いた状態保存機構によって，バックトラックによる逆方向実行を行う．順方向実行の際にスタックに情報を収集することによって，逆方向実行を行う．

label stack は順方向の label 命令が実行された際に情報を収集するスタックである．どのプログラムカウンタからジャンプしたのかを保存することで，逆方向実行時に正しく戻ることができる．また，プログラムカウンタの値も逆になり，n-k+1 を逆方向のプログラムカウンタとして保存する．同様に value stack で順方向の store 命令が実行された際に，変数の変化を保存しておく．

rjmp 命令は label stack から，restore 命令は value stack からそれぞれ逆方向実行のために状況を復元する．rjmp 命令は順方向実行の元の場所にジャンプして戻り，restore 命令は変数を前の値に戻す．

図 2.1 は label 命令と rjmp 命令の仕組みを示している．label 命令は jpc 命令と jmp 命令の行き先になる．label 命令が実行されると，そのジャンプ元のアドレスが label stack にプッシュされる．逆方向実行では label 命令は rjmp 命令に変換され label stack をポップすることで抽象機械が rjmp 命令を実行し元のアドレスにジャンプして戻る．

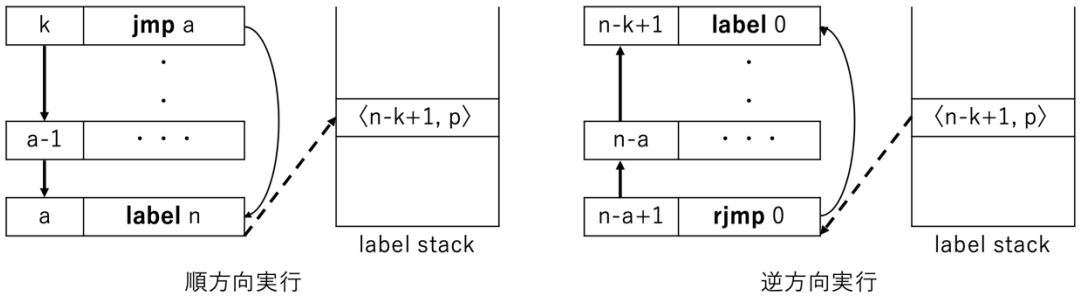


図 2.1 label stack

ソースプログラムは抽象機械が実行する単純な命令列にコンパイルされる．

先行研究[1]では，while ループや if 文，及び並列ブロックを持つような単純なプログラムを対象として，3 番地コードレベルの簡単なスタックマシンコードに変換している．順方向実行のスタックマシンコードを逆方向実行のスタックマシンコードに変換することで，同一の抽象機械で可逆実行を実現している．そして，逆方向実行はスタックから取り出した情報をもとに行われ，順方向実行を遡ることができる．特に while ループの部分でジャンプ命令を加えることによって同じコードでループの処理を何回でも行えるようになっている．

本研究の抽象機械はこの先行研究[1]の環境をもとにして作成した．

２．１で定義した言語を一般のスタックマシンと同様の動きをし，かつ逆方向実行に必要な情報を保存しておける抽象機械が Python によって作成されている．この抽象機械によって対象プログラムから生成したスタックマシンコードを実行する．逆方向実行の際にはスタックマシンコード自体とコードの順番を反転させて作成する逆方向実行用のスタックマシンコードを抽象機械に与えることでプログラムを逆方向に実行する．

表 1 は順方向実行用のスタックマシンコードを逆方向実行用のスタックマシンコードに変換した例である．順方向実行 1 行目の alloc 0 は逆方向実行 8 行目の free 0 に変換され，順方向実行 2 行目の ipush 0 は逆方向実行 7 行目の nop 0 に変換されている．他のスタックマシンコードも同様である．

1: alloc 0	1: alloc 0
2: ipush 0	2: par 1
3: sotre 0	3: resotre 0
4: par 0	4: nop 0
5: ipush 0	5: par 0
6: store 1	6: restore 0
7: par 1	7: nop 0
8: free 0	8: free 0

表 1　順方向実行の命令列(右)と逆方向実行の命令列(左)の例

2.4 可逆実行環境におけるデバッグ

並列プログラムは非同期的に各プロセスが実行されるため，再実行によって前回と同じ状況を再現しデバッグするということが難しい.

例として表 2 のような並列プログラムを考える場合，実行手順によって変数 x の最終的な値が変化する．表 3 のように単純な 3 通りの手順が存在すると考えた場合でも，再実行により同じ結果を得る確率はそれほど大きくないことが直感的にわかる.

実際の並列プログラムは大抵がこの例のプログラムより複雑なので，並列プログラムを再実行によってデバッグすることが難しいということが理解できる.

可逆実行環境でのデバッグでは逆方向実行によって手順を戻すことができるので，再実行をすることなくデバッグを行うことができる.

プロセス 1	プロセス 2
x = 2	x = 1 x = x + 1

表 2　並列プログラムの例

1 → 2 → 2	2 → 1 → 2	2 → 2 → 1
x = 2 x = 1 x = x + 1	x = 1 x = 2 x = x + 1	x = 1 x = x + 1 x = 2
x = 2	x = 3	x = 2

表 3　表 2 の実行結果

GNU デバッガなど多くのデバッガにはブレークポイントによって実行を途中停止させることができる．しかし，このブレークポイントは主に順方向の実行を途中停止させる機能として実装されていて，逆方向実行におけるブレークポイントを設定する機能は一般的に使われていない.

そこで，本研究では逆方向実行におけるブレークポイントを設定できるようにすることで既存のデバッガとの差別化を図る．逆方向実行におけるブレークポイントでは，各プロセスがブレークポイントの状況が満たされるまで逆方向実行を行うようにした.

この逆方向実行におけるブレークポイントは本研究の可逆デバッガの大きな特徴である.

表 3 の手順を 2→1→2 と進めて，逆方向実行におけるブレークポイントとして x の値が 1 になる時を設定した場合，最初に x=1 が実行された所まで戻ることができる．この時点まで戻れば次は手順を 2→2→1 と変えて異なる状況を追うことができる.

また，順方向実行時に無限ループに陥ってしまうと逆方向実行が行えないという問題を解決するために，命令列の実行を途中停止するボタンを実装した．この機能によって実行

環境が動かなくなりデバッグ不可能となることを回避できる．

最後に，本研究では状態の可逆性に焦点を当てているので，順方向の実行を正確に逆にしているわけではない．本研究の可逆デバッガでは順方向から逆方向への命令列の変換の際に，値を計算する op 命令は何もしない nop 命令に変換される．これは op 命令が直接状況に影響を与えず，その後の store 命令が実行されて初めて状況が変化するため，逆方向実行を簡略化できると判断したためである．そのため，本研究の逆方向実行が正確に順方向実行を取り消すことはない．変数名，label stack，value stack を共有することで並行して実行される抽象機械間の変数の更新を管理している．

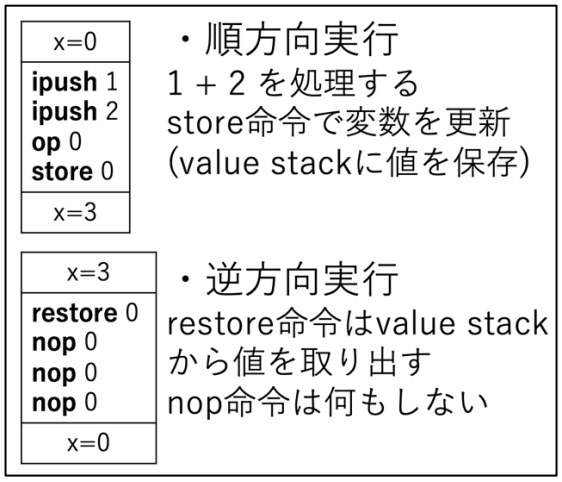


図 2.2　逆方向実行の簡略化の例

3 GUI を持つ可逆デバッガーの実装

本章ではデバッガーとしての機能と GUI を追加した実行環境の実装について説明する．対象プログラムは２．１で定義した言語とする．GUI は Python の標準ライブラリである Tkinter によって実装した．

この対象プログラムをテキストファイルなどで用意し，そのファイルの場所を入力し read ボタンを押すとファイルが読み込まれる．読み込まれた対象プログラムはスタックマシンコードに変換され，それぞれの複数プロセスに対応したスタックマシンに逐次命令を割り当てる．

if 文や while 文から発生した jmp 命令が実行されると，オペランドで指定されたジャンプ先の label 命令のアドレスがプログラムカウンタに書き込まれ，プログラムの実行位置が変更される．これと同時に，プログラムの実行位置の履歴はプロセス番号とペアで履歴として保存される．また，変数の値の変更は store 命令によって変更されると共に，変更前の値も履歴として保存される．これらの更新履歴は逆方向実行を行う際に使われる．

本研究の可逆デバッグに必要な情報を残すことができる抽象機械は Python によって作成されている．抽象機械は par 命令ごとにその並列構造に必要なだけのプロセスを生成し，それぞれに逐次命令列を当てはめる．抽象機械は命令とオペランドの組を一つずつ読み取り，その命令を実行していく．この命令とオペランドの組はそれぞれ固有のプログラムカウンタを持ち，その値でどの命令を実行するのかを制御する．

3.1 Tkinter による GUI の設計

本研究では GUI の作成のために Python の標準ライブラリである Tkinter を使用した．

Tkinter の利用には図 3.1 に示すようにインポートが必要である．ttk は Tkinter の従来のウィジェットに加えて Treeview を使用するために，filedialog は GUI 上でファイルを選択するためにそれぞれ必要である．

```
import tkinter as tk


from tkinter import ttk
from tkinter import filedialog
```

図 3.1　tkinter の準備

本研究で使用した tkinter のウィジェットは Label，Entry，Button，Text，Listbox，Checkbutton，Treeview である．

- Label: ウィンドウ上に文字を表示するためのウィジェット.
- Entry: テキストを入力できるテキストボックス. get メソッドによって文字列を取得することができる.
- Button: ボタンを作成する. クリックすると用意したイベントが実行される.
- Text: テキストを表示するボックス. Entry との違いは複数行のテキストを扱える点である.
- Listbox: 一行のテキストをリスト表示させるウィジェット.
- Checkbutton: クリックするとチェックマークが表示され, もう一度クリックするとチェックマークが消える. テェックされているかされていないかを真偽値で取得することができる.
- Treeview: 表を作成するウィジェット. クリックによって行番号を id として取得することができる. id は 0 から始まる 16 進数である.

続いて,以上のウィジェットを用いて作成した可逆デバッガーの GUI について説明する. 図 3.2 の sample.txt を読み込んだ様子を示して説明を行う.

```
var x;
x=0;
par{
  x=2;
}{
  x=1;
  x=x+1;
}
remove x;
```

図 3.2 sample.txt

まず, 以下のコマンドでプログラム vm_TK.py[付録]を実行する.

```
Python3 vm_TK.py
```

図 3.3 ファイルを選択

プログラムを実行すると図 3.3 が表示される．テキストボックスに可逆デバッグを行う対象プログラムを入力し保存されているファイルを読み込む．

三点リーダーが書かれたボタンをクリックして，ファイルを選択することもできる．読み込みは Read ボタンをクリックすると実行される．sample.txt を読み込むと図 3.4 が表示される．

可逆デバッグを実行するとウィンドウ左側に実行履歴がテキスト形式で表示される．図 3.4 では実行を行っていないので空白になっている．

ウィンドウ中央にはソースプログラムから変換されたスタックマシンコードが表形式で表示されている．pc/com/op は program counter/command/operand の略称．最上部と最下部の表は，並列構造を持たない部分のスタックマシンコードが表示され．中央部には，並列構造を形成するプロセスの数だけスタックマシンコードの表が横並びで表示される．sample.txt は 2 つのプロセスからなる並列構造を持つので，図 3.4 には 2 つの表が横並びで表示されている．

そして，ウィンドウ右上には変数名とその値の履歴が表形式で表示されている．この表は store 命令または restore 命令が実行されるたびに更新される．図 3.4 では store 命令を実行していないので空白になっている．

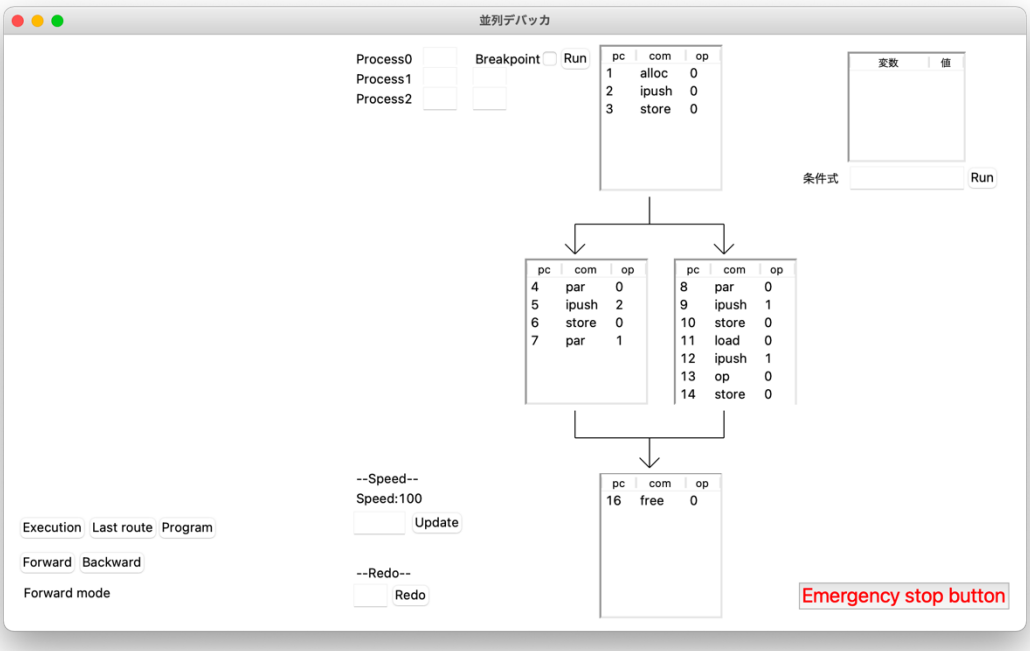


図 3.4　作成した GUI

その他，各ボタンとそれに対応する入力用テキストボックスについては以下の通りである．

- Execution: ウィンドウ左側に実行履歴を表示する Execution モードに切り替えるボタン. 初期モードは Execution モード.
- Last route: ウィンドウ左側に逆方向実行の経路を表示する Last route モードに切り替えるボタン. Redo を実行する際に使用するモード. 初期モードは Execution モード.
- Program: ウィンドウ左側に元のプログラムを表示する Program モードに切り替えるボタン. ソースプログラムとスタックマシンコードの対応を確認するモード. 初期モードは Execution モード.
- Forward: 順方向実行モードに切り替えるボタン. ウィンドウ左下に文字で現在のモードが記載される. 図 3.4 は順方向実行モード.
- Backward: 逆方向実行モードに切り替えるボタン. ウィンドウ左下に文字で現在のモードが記載される.
- Breakpoint/Run: それぞれの並列プロセスにブレークポイントを指定してそこまで実行させることができる. Breakpoint の文字の横にあるチェックボックスにチェックを入れた状態でスタックマシンコードをクリックするとそのスタックマシンコードにブレークポイントを設定できる.
- 条件式/Run: テキストボックスの欄に入力した条件式を満たすまで順方向実行または逆方向実行を行い続ける.
- Update: 順方向実行の速度を更新するボタン. テキスト欄には整数値を入力する. 初期値は 100. ここで言う速度 n とは n 個の命令を実行するごとに 1 秒間の入力待ち時間を設けるという意味である. ただし, n=0 の場合は入力待ち時間を設けない.
- Redo: 逆方向実行のさらに逆方向実行を行うボタン. 逆方向実行の履歴を保存することで順方向実行を行い, 逆方向実行を遡る.
- Emergency stop button: 順方向実行を停止させるボタン. デッドロックが起きたと思われる際に, ループから抜け出して順方向実行を停止させるためにある.

3.2 ソースプログラムとスタックマシンコードの対応

本研究で作成した可逆デバッガーではソースプログラムとそれから変換されたスタックマシンコードの対応関係を調べる機能を実装した.

ウィンドウの左下にある Program ボタンをクリックすることでウィンドウ左側に元のプログラムが表示される. この状態で, 任意のスタックマシンコードをクリックするとそのスタックマシンコードの変換元にあたる部分のプログラムが青文字, 背景色シアン色にハイライトされる.

以下に具体例を示す.

まずプログラムを実行し, 図 3.2 に示す sample.txt を読み込む. 次に, Program ボタン

をクリックしてソースプログラムを表示させる．この状態から pc/com/op = 1/alloc/0 の行をクリックすると図 3.5 のようになる．

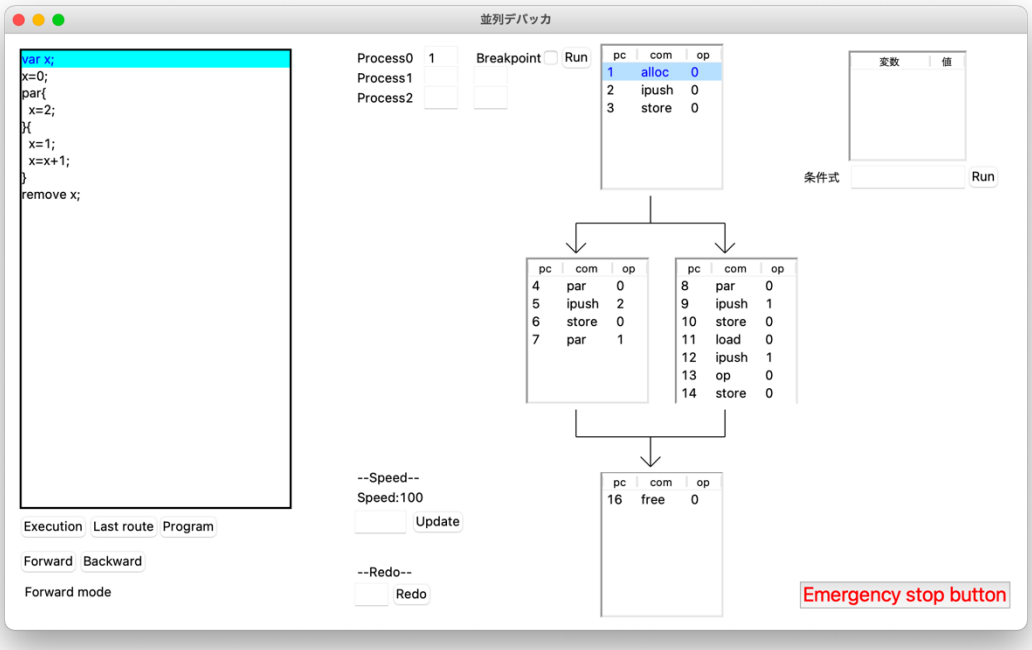


図 3.5　対応するソースプログラムとスタックマシンコードの例

3.3 実行例

ここで，いくつかの実行例を示す．実行例には図 3.2 の sample.txt とデッドロックを起こす可能性のあるプログラム philosophers.txt を用いる．

philosophers.txt では食事する哲学者の問題を取り上げる．図 3.6 に philosophers.txt のソースコードを示す．本研究で考える食事する哲学者は単純化のため 3 人とする．

3 人の哲学者は円卓を囲むように座っており，それぞれの哲学者の間には一本ずつフォークが置かれている．食事をするには左右 2 本のフォークが必要であり，哲学者は同時に 2 本のフォークを取ることはできない．そして，哲学者は自分の食事が終わるまでフォークを手放すことはない．

この時，変数にセマフォなどのロックが用意していないのでデッドロックが発生する可能性がある．

```
var phil1;
var phil2;
var phil3;
```

```
var fork1;
var fork2;
var fork3;
phil1=0;
phil2=0;
phil3=0;
fork1=0;
fork2=0;
fork3=0;
par{
    while (phil1==0) do
        if (fork1==0) then
            fork1=1;
        else
            skip;
        fi;
        if (fork2==0) then
            fork2=1;
        else
            skip;
        fi;
        if (fork1==1 && fork2==1) then
            phil1=1;
            fork1=0;
            fork2=0;
        else
            skip;
        fi;
    od;
}{
    while (phil2==0) do
        if (fork2==0) then
            fork2=2;
        else
            skip;
        fi;
```

```
            if (fork3==0) then
                fork3=2;
            else
                skip;
            fi;
            if (fork2==2 && fork3==2) then
                phil2=1;
                fork2=0;
                fork3=0;
            else
                skip;
            fi;
        od;
}{
        while (phil3==0) do
            if (fork3==0) then
                fork3=3;
            else
                skip;
            fi;
            if (fork1==0) then
                fork1=3;
            else
                skip;
            fi;
            if (fork3==3 && fork1==3) then
                phil3=1;
                fork3=0;
                fork1=0;
            else
                skip;
            fi;
        od;
}
remove fork3;
remove fork2;
```

```
remove fork1;
remove phil3;
remove phil2;
remove phil1;
```

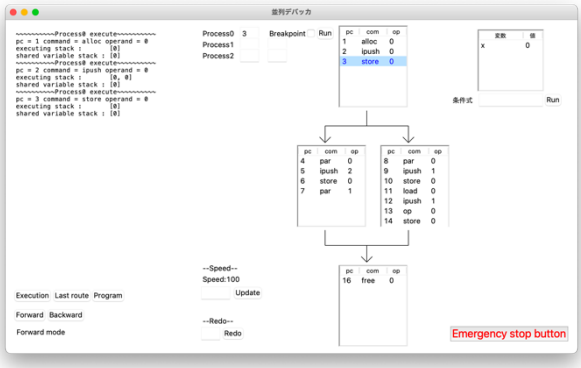
図 3.6　philosophers.txt

3.3.1 可逆デバッグの実行例

sample.txt を読み込ませ pc/com/op = 3/store/0 の行をクリックすると，図 3.7 に示すように par 命令が実行される直前まで順方向に手動実行を行う．

実行後は図 3.7 のように pc/com/op = 3/store/0 の行が青色文字に変わる．この文字色の変化は現在のプログラムカウンタが 3 まで実行が進んだ状態であることを表している．また，ウィンドウ上にある Process0 の横のテキストボックスに 3 が出力される．

他のプロセスを手動実行する場合も同様に行う．




```
~~~~~~~~~~Process0 execute~~~~~~~~~~
pc = 1 command = alloc operand = 0
executing stack :          [0]
shared variable stack : [0]
~~~~~~~~~~Process0 execute~~~~~~~~~~
pc = 2 command = ipush operand = 0
executing stack :          [0,0]
shared variable stack : [0]
~~~~~~~~~~Process0 execute~~~~~~~~~~
pc = 3 command = store operand = 0
executing stack :          [0]
shared variable stack : [0]
```

図 3.7　順方向実行の例(左)と実行履歴の拡大(右)

次に，図 3.7 の状態からブレークポイントを設定して順方向に自動実行を行う．

Breakpoint の横にあるチェックボックスにチェックを入れ Process0 以外のスタックマシンコードをクリックすると，プログラムカウンタにブレークポイントを設定できる．

図 3.8(左)はプログラムカウンタ 7 とプログラムカウンタ 10 にそれぞれブレークポイントを設定した様子である．ブレークポイントを設定すると pc/com/op = 7/par/1 の行と pc/com/op = 10/store/0 の行が赤色文字に変わる．この文字色の変化はブレークポイントが設定された状態であることを表している．また，ウィンドウ上にある Breakpoint の下の

テキストボックスに 7 と 10 が出力される．

この状態から Breakpoint の横の Run ボタンを押すと順方向に自動実行が行われる．自動実行を行ったところ図 3.8(右)のような結果になった．2 章で述べたように，この自動実行の結果は必ずしも図 3.8(右)と一致するとは限らず，これは一例である．

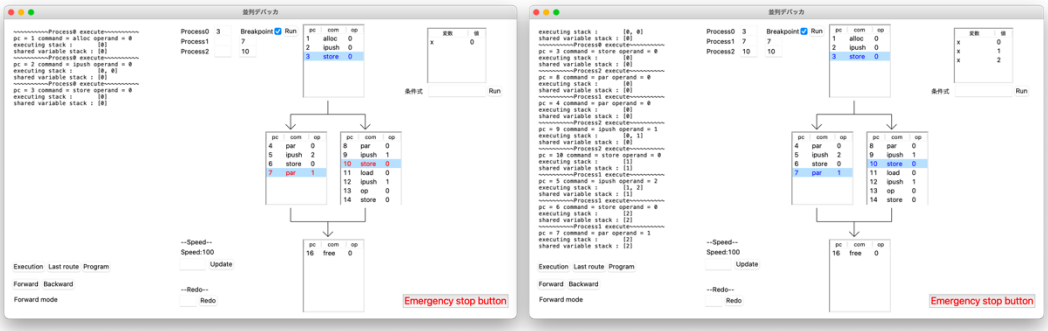


図 3.8　ブレークポイントの設定(左)と実行後の様子(右)

最後に図 3.8(右)の結果に対して逆方向実行を行う．

逆方向実行を行うには，ウィンドウ左下の Backward ボタンをクリックして逆方向実行モードに切り替える．逆方向実行モードに切り替わると図 3.9 のようにスタックマシンコードと矢印が反転する．また，ウィンドウ左下に Backward mode と表示される．逆方向実行モードではプログラムカウンタの値もひっくり返る．

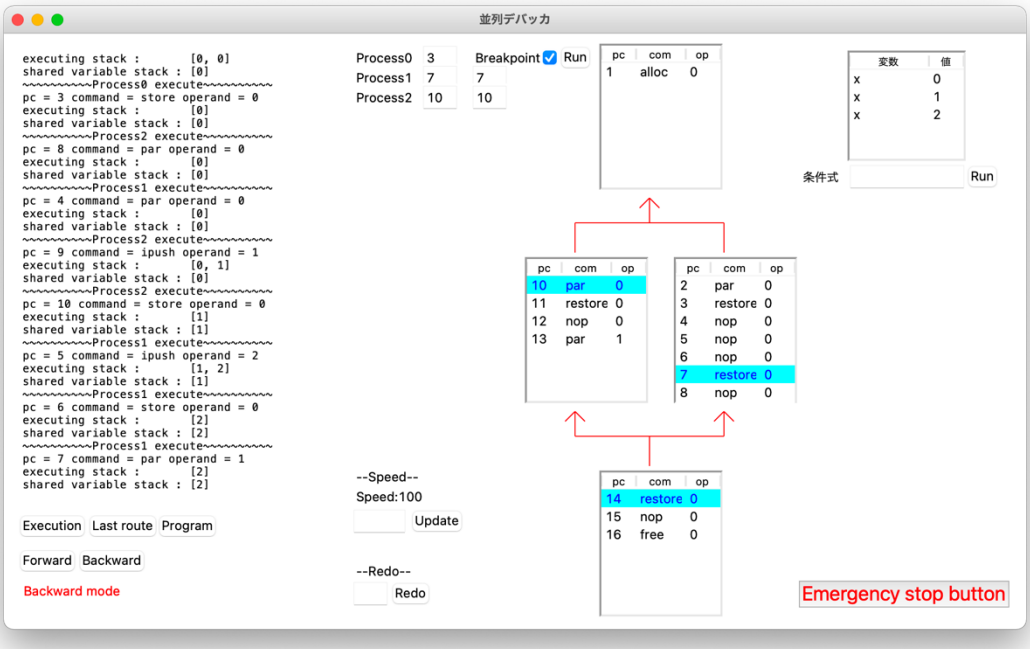


図 3.9　逆方向実行モード

逆方向実行モードで restore 命令の行をクリックすると逆方向実行が実行される．

図 3.9 の状態から pc/com/op = 7/restore/0 の行をクリックすると，図 3.10 に示すように逆方向実行が行われる．

逆方向実行は実行履歴に赤文字で出力される．

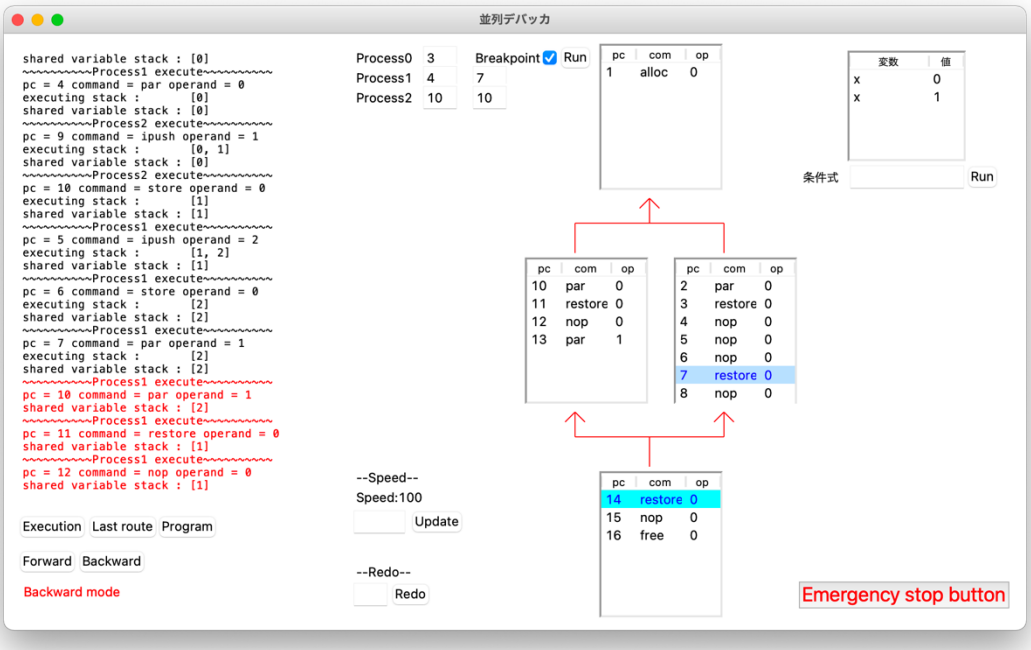


図 3.10　逆方向実行の例

…… ~~~~~~~~~~Process2 execute~~~~~~~~~~ pc = 10 command = store operand = 0 executing stack : [1] shared variable stack : [1] ~~~~~~~~~~Process1 execute~~~~~~~~~~ pc = 5 command = ipush operand = 2 executing stack : [1, 2] shared variable stack : [1] ~~~~~~~~~~Process1 execute~~~~~~~~~~ pc = 6 command = store operand = 0 executing stack : [2] shared variable stack : [2]	~~~~~~~~~~Process1 execute~~~~~~~~~~ pc = 7 command = par operand = 1 executing stack : [2] shared variable stack : [2] ~~~~~~~~~~Process1 execute~~~~~~~~~~ pc = 10 command = par operand = 1 shared variable stack : [2] ~~~~~~~~~~Process1 execute~~~~~~~~~~ pc = 11 command = restore operand = 0 shared variable stack : [1] ~~~~~~~~~~Process1 execute~~~~~~~~~~ pc = 12 command = nop operand = 0 shared variable stack : [1]

図 3.11　図 3.10 の実行履歴の拡大

図 3.10 では逆方向実行として par 命令，restore 命令，nop 命令が順に実行されている．この逆方向実行によって直前の順方向実行 par 命令，store 命令，ipush 命令がこの順に取り消されてプログラムカウンタ 10 の store 命令が実行された状態まで戻る．

まとめると，図 3.8(右)の状態では x に 1 が代入されその後に x に 2 が代入されていた．このまま順方向実行を進めると x+1 の値が x に代入され，最終的に x の値は 3 となる．ここで，逆方向実行を行い図 3.10 では x に 1 が代入された直後の状態まで戻った．

このように，実行を遡り別の実行順を試すなどして問題の検証を行うことを可逆デバッグという．

3.3.2 Redo 機能の実行例

続いて，sample.txt を用いて Redo 機能の実行例を示す．

まず，図 3.7 の状態から条件式によるブレークポイントを設定して順方向に自動実行を行う．ウィンドウの条件式の横にあるテキストボックスに x=3 を文字列として与える．さらにその横の Run ボタンをクリックすると，与えた条件 x=3 を満たすまで順方向実行が行われる．

ところが sample.txt は並列構造を持つプログラムであり，その実行順によって x の値が 3 になることなくプログラムを実行し終える可能性がある．図 3.12 は x の値が 3 になることなくプログラムを実行し終えた例である．

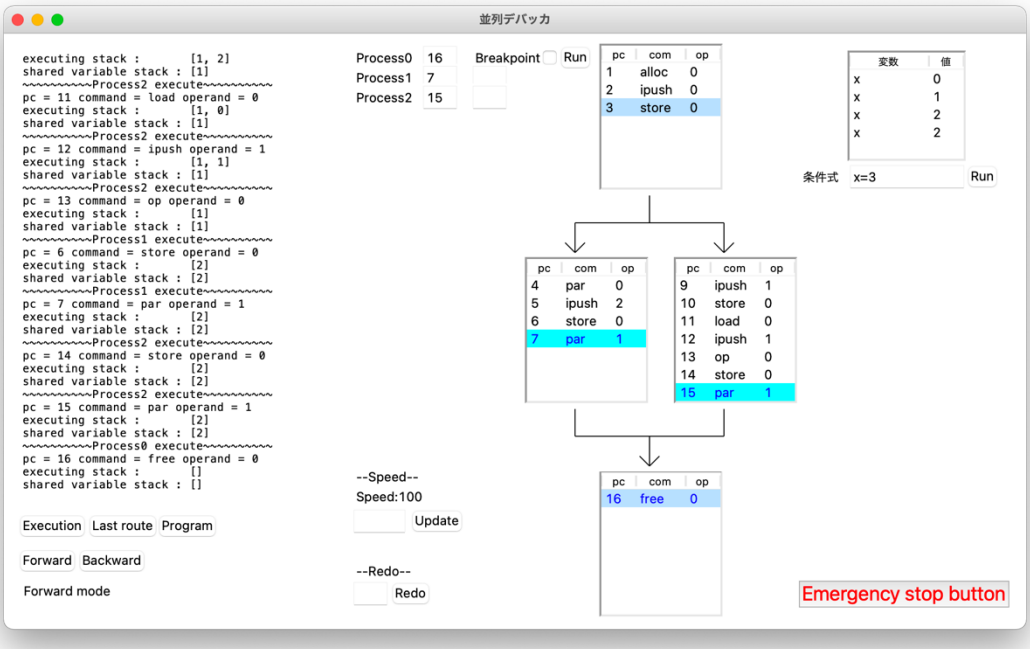


図 3.12　x の値が 3 にならない例

図 3.12 では x に 1 が代入され，その次に x+1 の値として 2 が x に代入された．そして最後に x に 2 が代入されてプログラムの実行が終わった．

この状態から最終的に x=3 になるように可逆デバッグを行う．

まず，x=0 の状態まで戻るように逆方向実行を行う．図 3.12 を逆方向実行モードに切り替え，条件式 x=0 を与えて Run ボタンをクリックする．すると store 命令によって x=0 が代入された状態まで戻る．続いて，再び順方向実行モードに切り替え，Last route ボタンを押して Last route モードに切り替える．すると図 3.13 のようになる．

Last route モードではウィンドウ左側に，前回どの手順で順方向実行が行われたのかが表示される．順方向実行の実行順に，実行された命令のプログラムカウンタが上から並べられ，されぞれのプロセス番号の列に振り分けられる．

ただし，行項目には行番号が降順に振り当てられている．これは複数回に分けて逆方向実行を行う際に行番号とプログラムカウンタの対応関係が崩れることを避けるためにこのようにしている．後から逆方向実行された命令のプログラムカウンタほどその行番号が大きな数になっている．

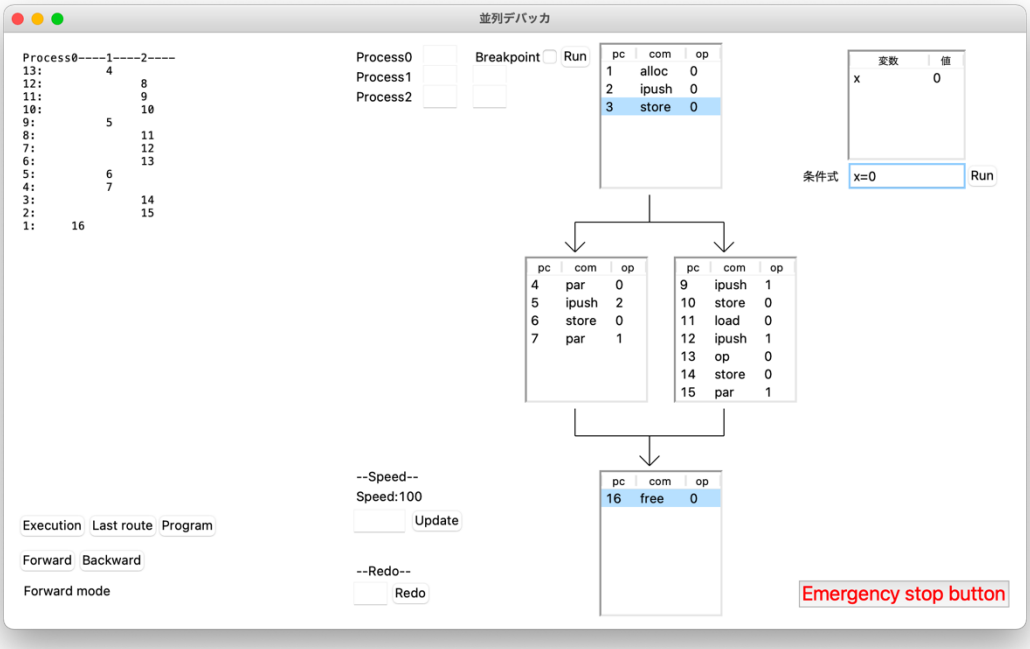


図 3.13 Last route モードの実行例

図 3.13 を見ると最初に実行された store 命令のプログラムカウンタは 10 であることがわかる．今回はここまで Redo による順方向実行を行うことにする．

プログラムカウンタ 10 の行番号は 10 なので Redo のテキストボックスに 10 と入力して Redo ボタンをクリックする．実行結果は図 3.14 のようになる．

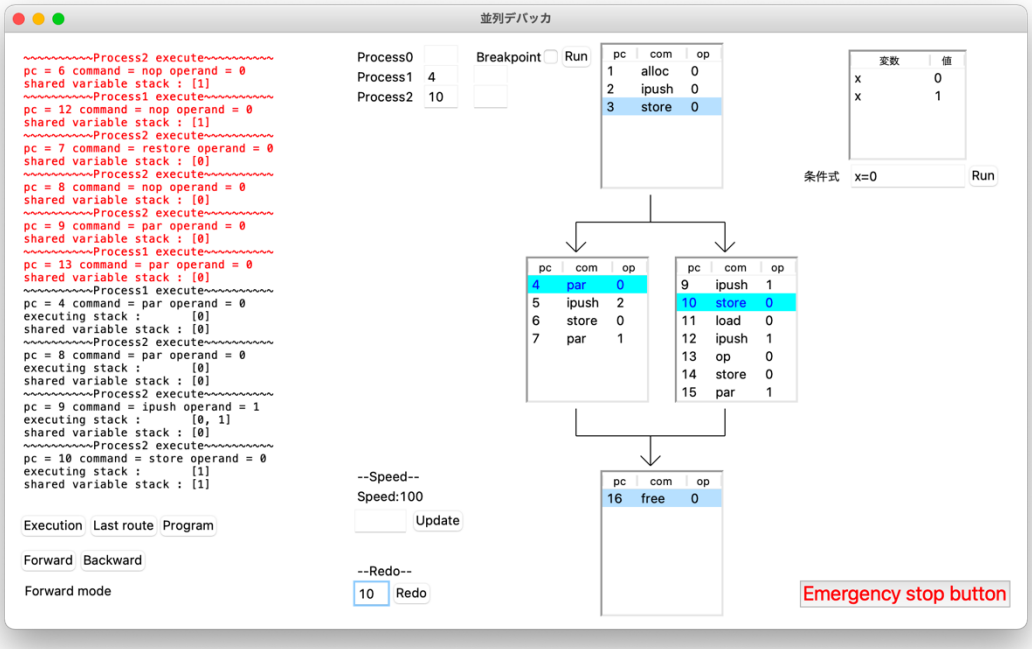


図 3.14　Redo 機能の実行例

Redo のテキストボックスに指定した分だけ図 3.12 と同じ実行手順で順方向実行が行われた.

このように Redo 機能によって，逆方向実行で戻した後に前回と同じ手順で命令を順方向実行することができる．Redo 機能で順方向実行の一部を再現して途中から分岐した新しい手順を実行することができる.

図 3.14 からは Process1 を先に全て実行してから Process2 の続きを実行すると最終的に x=3 になる結果を得られる．このようにしてどこから問題が発生したのかを検証することができる.

3.3.3デッドロックのデバッグの実行例

philosophers.txt を読み込ませ pc/com/op = 18/store/0 の行をクリックして，par 命令が実行される直前まで順方向に手動実行を行う.

続けて pc/com/op = 34/store/3 の行と pc/com/op = 86/store/4 の行と pc/com/op = 138/store/5 の行を順にクリックして順方向実行を進めると図 3.15 のようになる.

fork1 に 1 が，fork2 に 2 が，fork3 に 3 がそれぞれ代入されている．これは 3 人の哲学者が一本ずつフォークを持って待機している状態である．食事には 2 本のフォークが必要で，かつ食事が終わるまでフォークを放さないのでデッドロックが発生している.

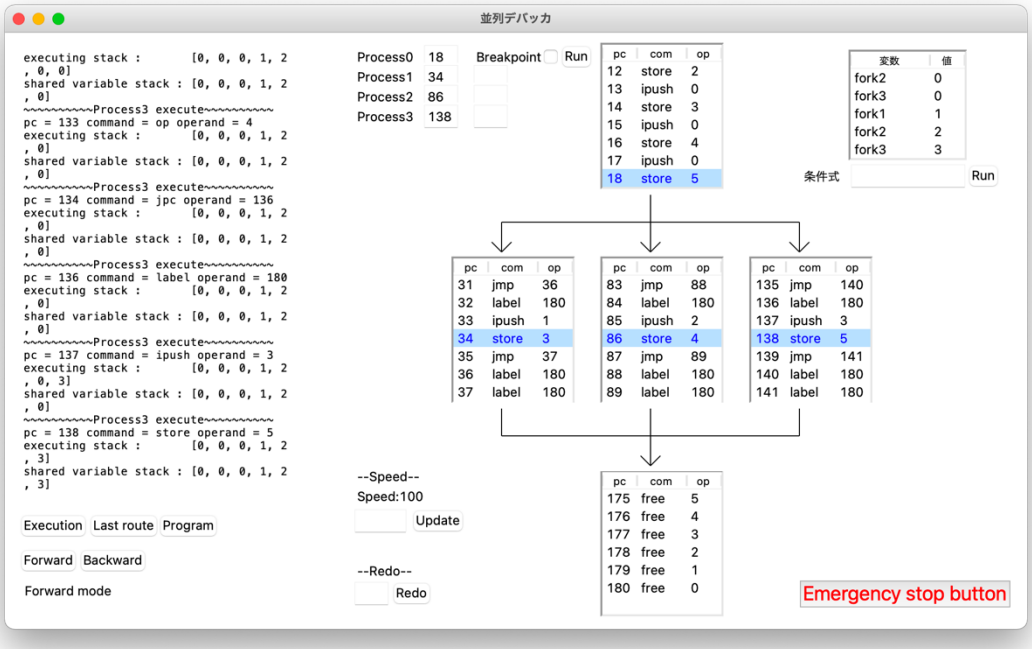


図 3.15　デッドロックのデバッグの実行例

図 3.15 の状態からはどのように順方向実行を進めてもプログラムが終わることはない．そのためブレークポイントの設定などによっては実行が終わることなく動き続けることになる．しばらく様子を見て無限ループに陥ったと判断した時は実行を途中で止めたい．

その際はウィンドウ右下の Emergency stop button をクリックして実行を停止させる．順方向実行中は命令 n 回ごとに 1 秒間入力を受け付けるので Emergency stop button によって実行を停止させることができる．

実行を停止した後は他のプログラムでやるのと同じように，逆方向実行で遡って問題を検証することができる．

4 おわりに

本研究では，先行研究[1]で提案された並列プログラムの実行環境をもとに，単純な並列プログラムに対する GUI を備えた可逆デバッガーを作成した．ネストしない並列ブロックを持つ単純な並列プログラムを対象に，抽象機械で命令を実行できるようにしている．この抽象機械では，順方向実行の際に履歴をスタックに保存し，逆方向実行の際にスタックから情報を取り出すことで可逆実行を可能にしている．

先行研究[1]の実行環境では FILO 方式でスタックから情報を取り出すことでバックトラックによる逆方向実行を行っている．この実行環境では逆方向実行のために順方向実行を完了させる必要があり，デバッグには不十分である．

本研究ではこの実行環境に対し，ブレークポイントによる途中停止機能と順方向実行モードと逆方向実行モードの切り替え機能を実装し，順方向実行を全て終了させなくても逆方向実行が行えるようにした．これによってデッドロックの問題検証など順方向実行が終了できないプログラムに対しても可逆実行を行うことができる．ブレイクポイントでは設定した条件が満たされるまで任意の状況からプログラムを両方向に自動実行する．

本研究で作成した可逆デバッガの GUI によって変数の値の変化を追いかけやすくし，対象プログラムとそれから変換されたスタックマシンコードの対応関係を可視化できるように提示し，可逆デバッグを効率よく行うことが可能になった．

今後の課題として，本研究で実装したデバッガは入れ子構造などを持たないような非常に単純な並列プログラムに限定されているので，その拡充として入れ子構造を持つ並列プログラムに対する可逆デバッガの実装が望まれる．また，本研究では断念したが，部分コンパイルによるプログラムの修正機能などが実装できればより効率的な並列プログラムのデバッグが可能になると期待される．

参考文献

［1］Takashi Ikeda and Shoji Yuen. A Reversible Runtime Environment for Parallel Programs. Nagoya University. 2020 ,p.2-6.

［2］Cercopes In Zipangu. “tkinter.ttk.Treeview【ツリービューウィジェット】”. 2021. https://cercopes-z.com/Python/stdlib-tkinter-widget-treeview-py.html ,(参照 2022-01-24)

［3］ Soshi. “【Python】tkinter でテキストボックスを作成する方法について解説！”. 株式会社 flyhawk. 2021. https://flytech.work/blog/19865/ ,(参照 2022-01-24)

付録　抽象機械

本研究では，抽象機械として動くプログラム vm_TK.py を作成した．このプログラムは github.com で公開している．

https://github.com/ryo2022/Sotsugyo/blob/master/vm_TK.py

実行コマンド

```
Python3 vm_TK.py
```