筑波大学情報システム実験
K-15 バイナリプログラムの解析 最終レポート
～x86_64 の ELF ファイルを対象としたタイムトラベルデバッガ memento の作成～

目次

はじめに

本レポートでは，筑波大学情報科学類において開講されている情報システム実験 B において最終課題として作成した，x86_64 アーキテクチャを対象とした ELF ファイルを対象とした Rust 製のタイムトラベルデバッガ memento を作成したことについて報告する．尚，memento は，ラテン語における meminisse（記憶する）の二人称単数命令形であるため，”記憶せよ”などの意味を有する．

memento が持つ機能の概要

本章では，memento が持つ主要な機能について，各節ごとに述べる．

タイムトラベルデバッグ

世間一般で呼称されるバイナリファイルの”デバッガ”は，バイナリファイルの処理フローを順に辿り，その途中で様々な情報を表示するソフトウェアと解釈される．このため，本レポートにおいては，”通常のデバッグ”という表現で，このようなデバッガが行うデバッグのことを指すこととする．

しかし，このような通常のデバッグは，実行の過程で起こった現象の再現率を 100% 保証することができないという課題が存在する．例えば，確率的に起こるエラーは，確実に再現できる保証は存在しない上に，複雑なプログラムではデータの更新がバイナリファイル中でどのようにして生じたのかを再度追うことは容易ではない．

これに対しタイムトラベルデバッグでは，例えばファイル操作などの副作用を伴わない限り，実行を開始してから現在までに生じた任意のステップの状態まで遡って実行できることが保証される．これにより，例えば「この変数が書き変わったことにより，この部分でエラーを引き起こしている」といったデバッグを行いやすくなるというメリットがある．

ここからは，実際に簡単なプログラムを例として用いて，タイムトラベルデバッグの効能について見ていくこととする．例えば次のような C 言語でのプログラムについて考えてみる．

リスト 1　main 関数の途中で変数 x に格納されている値が変更されるプログラム

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void foo(int *val) { *val = 1; }
5
6  int main() {
7    int x = 0;
8    foo(&x);
```

```
 9
10      if (x == 0) {
11        puts("zero");
12      } else {
13        exit(-1);
14      }
15    }
```

このプログラムでは，7 行目において変数 x に格納されている値が 0 に設定されているが，8 行目において呼ばれている foo 関数によって，変数 x の値が 1 に変更されている．foo 関数の実行が終了した後は，10 行目から始まる分岐によってプログラムの挙動が変わるようになっていて，x に格納されている値が 0 なら”zero”と標準出力に出力して終了するが，0 でない場合は何も出力せず，異常終了するようになっている．また，当然ながらこのプログラムをコンパイルして動作させると，何も出力せずに異常終了するようになっている．

このプログラムをコンパイルすることにより作成されたバイナリファイルをデバッグすることを考えると，通常のデバッグでは，上のプログラムの 10 行目に相当する部分に到達した際に初めて，変数 x に格納された値により分岐が生じていることが分かる．しかし，実際にはどのタイミングで変数 x の値が書き変わったのか，それとも元々 0 で無い値であったのかは非自明である．

無論，これは簡単なプログラムを用いた例であるので，例えば逆アセンブルを行ったコードを最初に読み込んでおいて変数の値の動きに注意したり，もしくは通常のデバッグを再度やり直して変数 x がどのような変化を辿っているかに注視すればさほど時間を要さずとも関数 foo が変数 x の値を操作していることが容易に特定できるかもしれない．しかし，実際にデバッグ対象となるようなファイルは往々にしてさらに複雑であり，また，実行中に生じる現象も再度デバッグをやり直しても発生が保証されるものではないケースも存在する．

そういった場合に，タイムトラベルデバッグを用いることができると，デバッグを開始してから現在までに生じた現象に対して，その現象が発生した時点でのレジスタ，メモリが復元されるため，現象の再発が保証されることにより，タイムトラベルデバッグの有効性はあるといえる．

バイナリダンプ

ターゲットファイルのバイナリ情報をダンプする．これは xxd コマンドのような出力を行う．

図 1　バイナリダンプを行っている様子

逆アセンブル

解析対象の ELF ファイルに対して，逆アセンブルを適用した結果を表示する．この結果はコマンドの指定によっては外部ファイルに出力することも可能である．

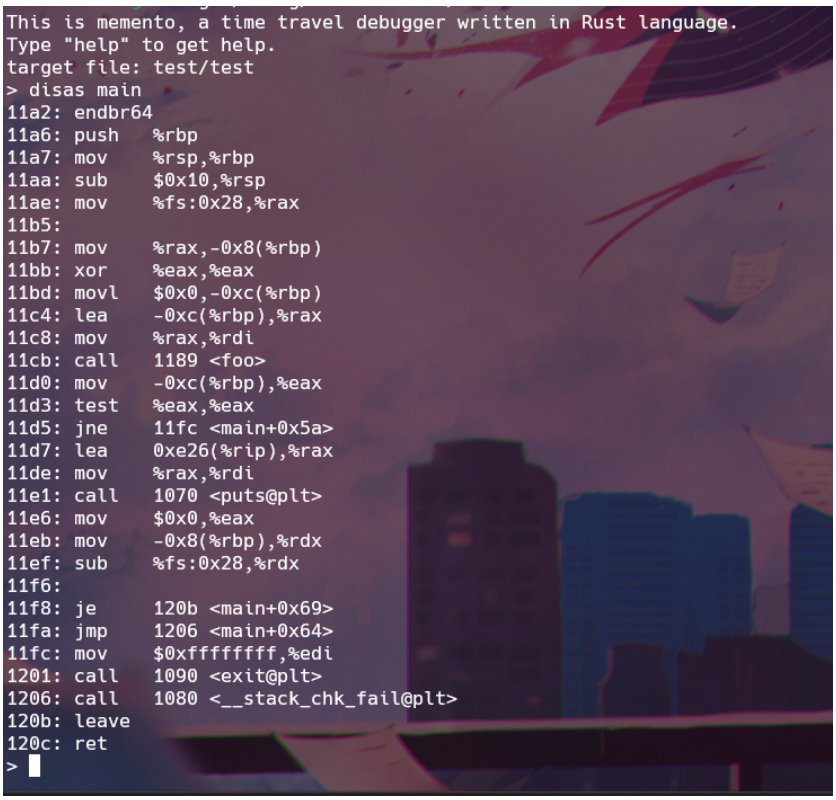

図 2　main 関数を逆アセンブルする様子

ブレークポイントの設置

解析対象の ELF ファイルに対して，ブレークポイントを設置する．memento を用いてブレークポイントが張られている命令まで到達すると，その実行が一時停止され，またシェルを通して memento の操作が行えるようになる．

図 3　ブレークポイントを設置する様子

レジスタに格納されている値のダンプ

現在の各種レジスタに格納されている値をダンプする．

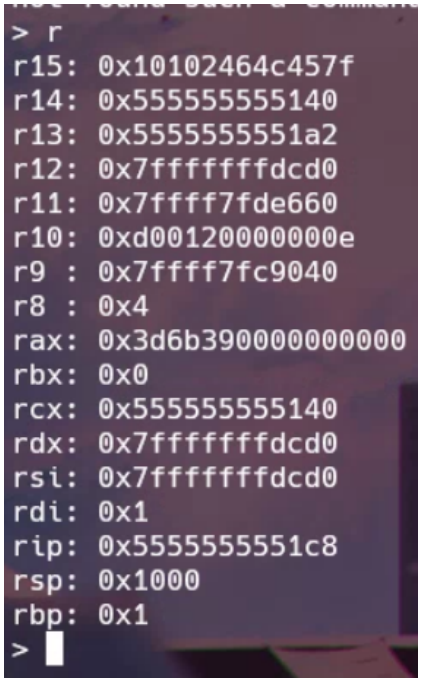

図 4　レジスタダンプを行っている様子

類似した機能を持つアプリケーションとの比較

本章では，memento と類似した機能を持つようなソフトウェアと，その特性の差異について比較する．

LLDB

LLDB は，LLVM プロジェクトのコンパイラによってコンパイルされたバイナリプログラムに対するデバッガである．これはタイムトラベルデバッグの機能を持たない．同様に，他の数多のデバッガもタイムトラベルデバッグの機能を持っていない．

GDB

著名なデバッガの 1 つとして挙げられるのが，GDB である．GDB はそれ単体ではタイムトラベルデバッグの機能を有しないが，rr と呼ばれるプラグイン[*1]を併用すると，可能になる．

しかし GDB は Apple Silicon が搭載された端末上では動作しないという問題点がある．一方，memento は，Apple Silicon 上でも動作を確認したため，この点で優位性があるといえる．

[*1] https://rr-project.org/

FireDBG

FireDBG[*2]は，SeaQL[*3]と呼ばれるオープンソースコミュニティによって開発された，シークバーのような UI でタイムトラベルデバッグを行えたり，変数の値が分かりやすく表示されたりと，ビジュアライズ機能を持つタイムトラベルデバッガである．この点，デバッグ性に優れていると言えるが，対象とするバイナリターゲットが Rust コンパイラによってコンパイルされたバイナリファイルに限定されているという欠点がある．

また，本レポート執筆時点（2024 年 2 月現在）ではブレークポイントの設置やステップ実行にも対応できていないという欠点も存在する．

実装の前提知識

本章では，次章で実装の詳細について説明するにあたって，必要となる前提知識について述べる．

クレート

memento を実装するにあたって用いた Rust 言語では，クレートという単位によって外部プログラムを取り込むことが可能になる．これは例えば Python 言語でいうところのライブラリや，Ruby 言語でいうところの gem に相当する概念である．

Rust 言語では，クレートの追加や削除は cargo と呼ばれるパッケージマネージャー兼ビルドシステムが担当することがほとんどである．

Rust における safe と unsafe

Rust 言語は，メモリ安全性やスレッド安全性などを重視して開発されているマルチパラダイム言語であり，その側面から，特に近年は”Rust は安全な言語である”という主張が妄信的に述べられることが多いように感じる．

しかし実際のところ，本章で述べる，Rust の unsafe な部分は安全性が完全に保証されているとはいえない．すなわち，unsafe な部分ではプログラマも安全性に責任を負うことになる．もちろん，unsafe でない部分については Rust のコンパイラが安全性に責任を負うため，意図しないクラッシュなどが発生した場合，プログラマは自身の書いた unsafe な部分の実装について注視すればよいため，この点で C 言語や C++ 言語よりも Rust 言語を用いる優位性は依然として存在するといえる．

unsafe な部分においては，プログラマは変数などに対する生ポインタを参照外ししたり，システムコールを呼んだりできる．

[*2] https://firedbg.sea-ql.org/
[*3] https://www.sea-ql.org/

ptrace

ptrace は，Linux において，プロセス中から別のプロセスを制御することのできるシステムコールである．ptrace によって制御できる項目のうち，主要なものを挙げると，次のようになる．

- プロセスのメモリ空間の読み書き
- プロセスのレジスタの読み書き
- プロセスのシステムコールの呼び出し
- プロセスのシグナルの送信

そもそも，ptrace は，その man ページにも”It is primarily used to implement breakpoint debugging and system call tracing.”とあるため，デバッガ用に作成されたシステムコールであるため，これを memento に用いるのは自然な発想であるといえる．

実際には，Rust 言語にはシステムコールを簡単に呼べるようにしているクレートがいくつか存在するため，ptrace を用いる際にもこういったクレートを用いることを検討した．

そのようなものの中の 1 つに，libc クレート[*4]が存在する．これは単純に C 言語の標準ライブラリである libc を Rust 言語から利用できるようにしたものである．そのため，呼び出す際にも unsafe な部分となるものが多く，また，関数の返り値も，c_int 型という，C 言語での signed int 型と同等の型で返ってくるものが多いため，Rust 言語でプログラムする利点を活かせているとは言い難いものになっている．

一方，nix クレート[*5]は，libc クレートのラッパーであり，libc クレートでは unsafe となるようなものを safe に呼ぶことができるようになっていたり，返り値の型も，Rust 言語での Result 型として返すようなものが多い．このため，エラーハンドリングのしやすさなどの点で libc クレートよりも優位性があると言える．このため，memento では libc クレートを用いずに，nix クレートを用いている．

x86_64 アーキテクチャにおけるブレークポイントの実現手法

x86_64 アーキテクチャにおいては，INT3 命令と呼ばれる命令を CPU が解釈することでソフトウェア割り込みを発生させることができる．実際には，この命令は 0xCC というバイト列により表現される．

INT3 命令によってプログラムの実行が途中で停止されると，SIGTRAP というシグナルが発生する．よってデバッガはこのシグナルを検知して，ブレークポイントへの到達を検知することができる．

よって，デバッガが，プログラムのブレークポイントへの到達を検知した場合，次のような処理

[*4] https://crates.io/crates/libc/
[*5] https://crates.io/crates/nix/

を行うことにより，ブレークポイントの機能を実現できる．

- 各レジスタの情報などを開示
- 元の命令に書き換える
- プログラムカウンタを 1 つ戻す
- レジスタを復元
- プログラムカウンタを 1 つ進める
- ブレークポイントを再度復元

ASLR と execv

ASLR（Address Space Layout Randomization）は，バイナリファイル中での関数や変数のアドレスをランダムに配置する仕組みのことを指す．これはセキュリティの観点から重要な機能であるため，例えば gcc を用いて C 言語のプログラムをコンパイルする際には，デフォルトでこの機能が有効になっている．

一方で，デバッガを用いてプログラムをデバッグする際には，この機能が邪魔になることがある．例えば，プログラムの特定の関数にブレークポイントを設置したい場合，その関数のアドレスを事前に知っておく必要があるが，ASLR が有効になっていると，そのアドレスが毎回変わってしまうため，ブレークポイントの設置が困難になる．

このため，多くのデバッガは，ターゲットとなるプログラムからプロセスを作成する際に，personality システムコールを呼び出すことで, ASLR を無効にする処理を行っている．personality システムコールは，プロセスの各種設定を変更するためのシステムコールである．

例に漏れず memento でも ASLR を無効にする処理を行っている．Rust 言語では，nix クレート内に nix::sys::personality::set という関数が存在し，これを用いることで personality システムコールを経由して，プロセスの各種設定を変更することができる．具体的には，ASLR を無効にしたい場合は，nix::sys::personality::Persona 内において ADDR_NO_RANDOMIZE という定数が提供されているので，これを引数として次のように渡せばよい．

リスト 2　ASLR を無効にする処理

```
use nix::{
  sys::personality::{self, Persona},
  unistd::{
      fork,
      ForkResult::{Child, Parent},
  },
};

let pers = personality::get().unwrap();
if let Err(e) = personality::set(pers | Persona::ADDR_NO_RANDOMIZE) {
```

```
11      panic!("failed to diable ASLR: ERRNO={e}");
12  }
```

このようにすることで，現在 memento が動作しているプロセスについて，ASLR を無効にすることができる．

ASLR を無効にした後は，このプロセスから fork したプロセスに対して，execv システムコールを呼び出し，デバッグ対象のプログラムを実行することで，memento がデバッグを行えるようになる．

リスト 3　fork したプロセスに対して execv システムコールを用いている様子

```
1   use memento::memento;
2   use nix::{
3     sys::personality::{self, Persona},
4     unistd::{
5         fork,
6         ForkResult::{Child, Parent},
7     },
8   };
9   use std::{fs::read, path::Path};
10  use target::target;
11
12  let pid = unsafe { fork() };
13
14  let pid = match pid {
15      Ok(fork_result) => fork_result,
16      Err(e) => panic!("failed to fork: ERRNO={e}"),
17  };
18
19  match pid {
20      Parent { child } => memento(child, &args.target),
21      Child => target(
22          Path::new(&args.target),
23          &args.args.iter().map(|s| &**s).collect::<Vec<&str>>(),
24      ),
25  }
```

実装の詳細

memento の構成図

memento は，Rust 製のプロジェクトであり，複数のソースファイルからなる．それらの大まかな関係を図示すると，次のようになる．memento 実行開始時からの実行順序が明確なものについては，各ファイルを示すノード間を接続する，ファイル同士の関係性を表記したエッジの部分にナンバリングを行った．

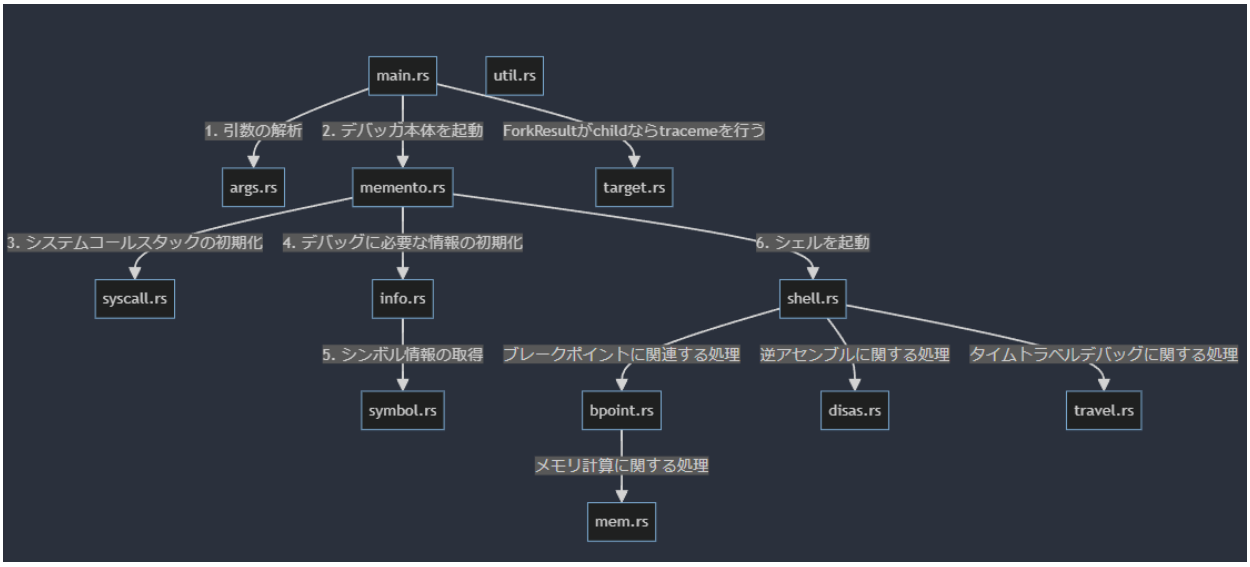

図 5　memento の構成図

memento のプログラムの実行は，main.rs から開始される．ここでは，コマンドライン上でユーザーから与えられた引数を，args.rs にある処理を用いて解析し，その後は先に述べた ASLR の無効化や fork 等を行ってから，memento.rs において定義されている，memento の機能を本格的に実行する関数を呼び出す．

memento.rs では，まずシステムコールスタックの初期化と，デバッグに必要な情報，例えばブレークポイントのリストやセクションヘッダのオフセット，メモリの状態などについて初期化を行う．この際，ターゲットとなるプログラムのシンボル情報を取得するために，symbol.rs にファイルを分割して処理を定義している．

memento の操作は，基本的に簡易的なシェルをインターフェースとして行われる．この機能を提供しているのが shell.rs である．shell.rs 中で定義されているシェル本体のプログラムでは，ユーザーから受け取ったコマンドを解析し，妥当な入力であるなら，その内容によって処理を分岐させている．

util.rs は，memento のプログラム中において様々な場面で用いることのできる汎用的な関数な

どを集積したものであり，様々なファイルから参照されているが，その呼び出し関係を逐一記すことはファイル間の関係性を示すことにおいては非本質的であると考えたため，上図には util.rs の呼び出し関係は表記していない．

タイムトラベルデバッグ

各ステップでの状態の管理

タイムトラベルデバッガを実装するにあたっては，各ステップでの状態を保存しておく必要がある．具体的には，各ステップでのメモリの状態やレジスタの値を保持しておく必要がある．

これらの情報をメモリに載せることを考えると，各ステップごとにこれらの情報を愚直に保存すると，実行に大量のメモリが必要になってしまい，実用性に欠けるものになってしまうことが予想される．このため，保存する情報量は極力削減されるべきである．

memento では，この問題に対して，暫定的に各ブレークポイントにおけるメモリとレジスタの値を保持するようにしている．この手法は，ブレークポイント単位でのタイムトラベルデバッグしか行えなくなってしまうが，メモリの消費量は削減できることが予想される．

ブレークポイント

ブレークポイントの実現には，あらかじめ述べてある通り，ptrace システムコールを用いる．

リスト 4　ブレークポイントを設置する関数 set

```
1  use nix::{
2    libc::{c_void, user_regs_struct},
3    sys::{ptrace, wait::wait},
4    unistd::Pid,
5  };
6  use std::error;
7
8  pub fn set(&mut self, addr: u64) -> Result<u8, Box<dyn error::Error>> {
9    wait().unwrap();
10   let read = match ptrace::read(self.pid, addr as *mut c_void) {
11       Ok(res) => res,
12       Err(e) => {
13           panic!("failed to read memory: ERRNO={:?}", e)
14       }
15   };
16
17   let mut read_vec = read.to_le_bytes();
18
19   let head = read_vec[0];
20   read_vec[0] = 0xcc;
```

```
21      let mut write = 0;
22      for (i, item) in read_vec.iter().enumerate() {
23          write += (*item as u64) << (i * 8);
24      }
25
26      unsafe {
27          ptrace::write(self.pid, addr as *mut c_void, write as *mut c_void)?;
28      }
29
30      self.bpoint.push(BPoint::new(addr, head));
31      Ok(head)
32  }
```

コマンド実行結果の外部ファイルへの出力

memento は，逆アセンブルの結果や，バイナリダンプの結果などを，デフォルトの出力先は標準出力であるが，オプションとして外部ファイルに出力することができる．この点については，次のような Write トレイトを拡張した関数を用いることで実現されている．

リスト 5　指定の出力先に出力する関数 write_to_output

```
1  use std::io::{self, Write};
2
3  pub fn write_to_output<W: Write>(output: &mut W, message: &str) -> io::
       Result<()> {
4      write!(output, "{}", message)?;
5      Ok(())
6  }
```

この write_to_output 関数を用いる場合，引数である output の部分には出力ハンドラを指定する．このため，実際に write_to_output 関数を用いる場合は，例えば次のようにする．

リスト 6　ハンドラの指定

```
1  let mut output_handle: Box<dyn Write> = if !filename.is_empty() {
2      Box::new(File::create(filename).unwrap())
3  } else {
4      Box::new(io::stdout())
5  };
6
7  Util::write_to_output(&mut output_handle, &res_str).unwrap();
```

この例では，filename という変数が空でない場合，2 行目のように output_handle は filename の名前でファイルを作成されたハンドラが指定される．一方，空の場合は，4 行目の様に標準出力

のハンドラを指定する．

実行結果

本章では，実際に memento をバイナリファイルに対して作用させた際の実行結果を示す．

まず，動作確認用のプログラムを作成するために，タイムトラベルデバッグの機能説明を行っている節で示した C 言語でのプログラムを用意した．ここに再掲する．

リスト 7　動作確認用の C 言語のプログラム

```c
#include <stdio.h>
#include <stdlib.h>

void foo(int *val) { *val = 1; }

int main() {
  int x = 0;
  foo(&x);

  if (x == 0) {
    puts("zero");
  } else {
    exit(-1);
  }
}
```

このプログラムに対して，gcc を用いて実行可能形式ファイルを作成する．このため，次のようなコマンドを実行した．

```
$ gcc -o test test.c
```

これにより，デバッグターゲットのバイナリファイルが作成されたので，memento を起動して，このバイナリファイルから作成されたプロセスにアタッチする．

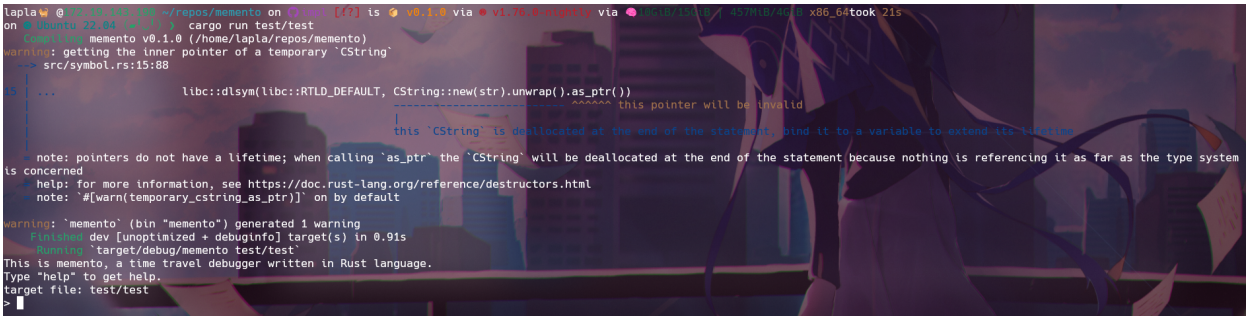

図 6　memento を起動して，バイナリファイルにアタッチする様子

アタッチが完了すると，memento のシェルが起動し，ユーザーからのコマンドを受け付けている様子が確認できる．次に，main 関数を逆アセンブルしたいため，”disas main”とシェルに入力して，逆アセンブル結果を得る．

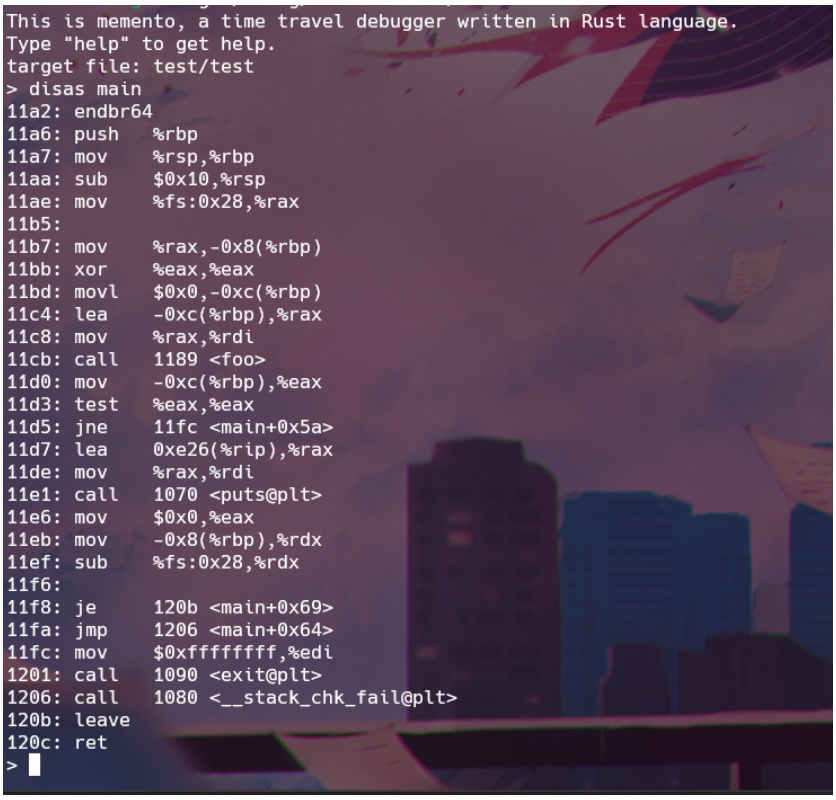

図 7　main 関数を逆アセンブルする様子

memento は，プロセスにアタッチする際に，ELF ファイルからシンボル情報を抽出して，各関数の先頭アドレスとその関数名をハッシュマップに紐づけて保存している．このため，”disas main”のようにしても main 関数のアドレスを逆引きすることで正しいアドレスを認識できるようになっている．

出力を見ると，0x11cb において関数 foo を呼び出していることが分かる．今回は，簡易的に，関数 foo の呼び出し前と呼出し後にブレークポイントを設置して，各地点でのレジスタの値を見ることとする．

ブレークポイントを設置するには，”b { アドレス }”の形式に従って入力する．この際，アドレスは 0x を先頭に付けると 16 進数として解釈され，アラビア数字での指定の場合は 10 進数として，そうでない場合はシンボル名として解釈されるようになっている．

図 8　ブレークポイントを設置する様子

今回は，0x11c8 と 0x11d0 にブレークポイントの設置をするようにコマンドを打っているので，16 進数での指定となっている．これにより，この 2 つのアドレスに到達した場合，ブレークポイント到達時の処理が行われるようになっている．

このため，まずは memento にプログラムをそのまま実行させようとする．これを行うにはシェルに”r”と入力すればよい．

ブレークポイントに到達すると，各レジスタの値が表示されるようになっている．この後に動作を続けるようにコマンドを入力すると，プログラムは次のブレークポイントに到達するかプログラム自体が終了するまで実行を継続する．これを実現するにはシェルに”c”と入力すればよい．

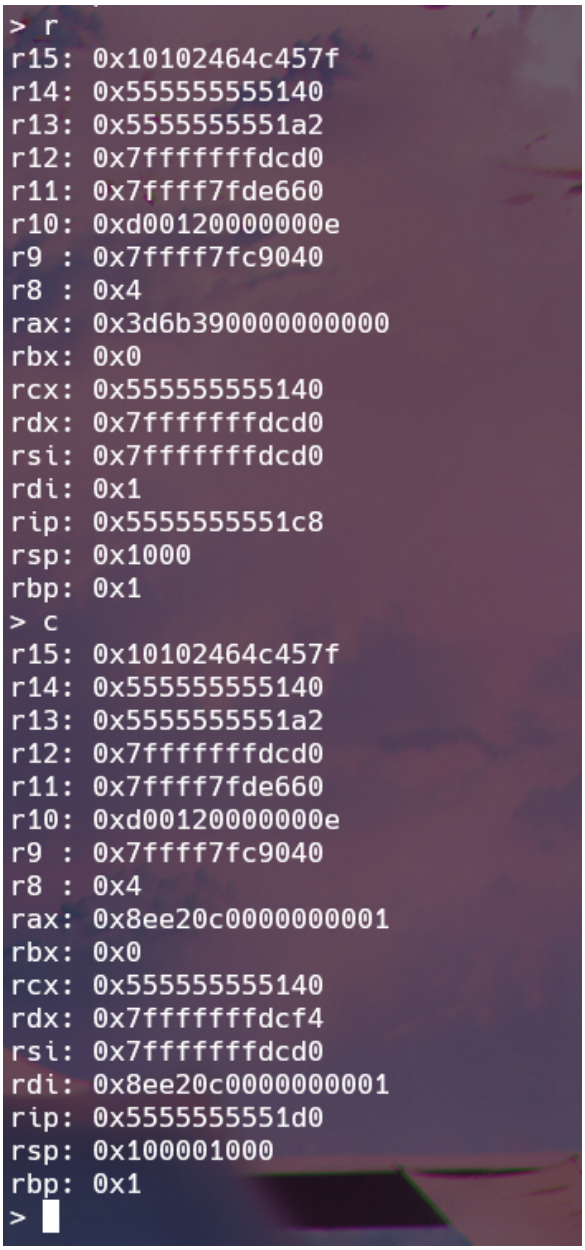

図 9　2 つのブレークポイントに到達した様子

各”r”と”c”によって，それぞれ 1 つ目と 2 つ目のブレークポイントに到達して，各レジスタの値が表示されている様子が確認できる．

さて，ここまででプログラムは 2 つ目のブレークポイント（0x11d0）に到達している．ここで，タイムトラベルデバッグを行えることを確認するために，実際にタイムトラベルデバッグを行えるコマンドをシェルに入力してみる．memento は現在のところブレークポイント単位でのタイムトラベルデバッグにしか対応できていないので”travel”と入力することで現在の地点より 1 つ前のブレークポイントでの状態に立ち返ることができる．

図 10　タイムトラベルデバッグを行う様子

実行結果を見ると，各レジスタの値が 1 つ目のブレークポイントに到達した際と同様になっている．このため，タイムトラベルデバッグが実行できているといえる．

今後の展望

デバッガとしての更なる機能の追加

現在のところ，memento はデバッガとして十分な機能を有しているとは言えないと考えている．具体的には，次に挙げるような機能の追加を展望として持っている．

- ステップ実行
- 各ブレークポイントで停止した際に変数に格納されている値などの表示
- strace，ltrace が有するようなトレース機能
- TUI や GUI ベースでの情報の提示

これらの詳細について，以降で述べる．

ステップ実行

現状の memento は，設置されたブレークポイントの地点でしか停止することができない．このため，細かいステップ単位で状態の変化を確認したい場合，ブレークポイントを多く張らなければ

ならない．これに対しステップ実行を実装できれば，さらにデバッグ性が向上すると考える．

各ブレークポイントでの変数の値表示

現在のところ，memento がブレークポイントに到達した際に表示する情報は，各レジスタの値のみである．しかし，デバッグを行うことを考慮すると，プログラム中で用いている各変数の値等も閲覧できる方が望ましいことは明白である．

strace，ltrace が有するようなトレース機能

本講義では，strace や ltrace を用いて様々なプログラムの動作をトレースするような課題が存在した．このようなトレースツールも，内部的には ptrace システムコールを利用してトレースを行っているため，memento にも同様の機能を組み込むとより面白いのではないだろうかと考えている．

TUI や GUI ベースでの情報の提示

memento は現在，CUI でのシェルをインターフェースとした対話型デバッガとなっている．しかし CUI では提示できる情報に限界があると考えている．このため，例えば TUI や GUI ベースでのアプリケーションとして実装することができれば，より直感的なデバッグ情報の提示を memento が行えると感じている．

例えば，私は本講義の第 2 回課題の一環として，ELF ファイル中で定義されている関数の依存関係を可視化するようなツールを作成したことがある．これは，講義資料の指示としては例えば

```
f: g, h, printf
g: fopen
h: connect
```

のような形で，単に呼び出し元の関数名と呼び出している関数名を出力すればよいというものであったが，これでは直感性に欠けると感じた私は，オプションとして Mermaid 記法でグラフ構造を出力するようなツールを作成した．このとき，実際に `https://mermaid.live/edit` 上で出力された Mermaid 記法のグラフを描画させた結果が次である．

図 11　第 2 回課題で作成した，Mermaid 記法で出力された関数間の依存関係を表したグラフを描画した結果

このようにすることでより直感的な情報提示が行えたと感じており，このような過去に作成した課題のプログラムを一部利用し，memento の情報提示性を向上させることができるのではないだろうかと考えている．

外部クレートへの依存の削減

memento は現在のところ，逆アセンブルについては逆アセンブルプラットフォームである Capstone[*6]の Rust でのラッパーである Capstone-rs クレート[*7]を利用している．これを自前の処理に置き換えることで依存性を低下させることが展望として存在する．

また，バイナリファイル中に含まれる関数名などの抽出には goblin クレート[*8]を用いている．

特にこの 2 つのクレートについては，個人的には依存を脱却したいと考えているため，将来の展望の 1 つとしては主にこの 2 クレートを用いないというものが存在する．

多様なアーキテクチャへの対応

現在のところ，memento は x86_64 の ELF ファイルのデバッグにしか対応していない．例えばこれに加えて ARM のバイナリや PE ファイルに対してもデバッグが行えるようになると，memento はより広範な使途を持つことになる．

これには例えば ARM 等の CPU アーキテクチャ向けのバイナリに対応することや x86 の ELF ファイルに対応することが課題として挙げられる．

[*6] https://www.capstone-engine.org/
[*7] https://github.com/capstone-rust/capstone-rs
[*8] https://github.com/m4b/goblin

memento のバイナリサイズの削減

Rust 言語について頻繁に言及されることの 1 つに，コンパイルした際にバイナリサイズが肥大化してしまうというものがある．実際に手元の GDB について，サイズを見てみると約 10MB 程のサイズであるが，memento については後述するような対策を行わないと約 14MB になってしまうという問題がある．

この原因には複数のものが考えられるが，memento においては現状主に以降で述べるような対策を取っている．ただし, 2024 年 1 月下旬に, `https://kobzol.github.io/rust/cargo/2024/01/23/making-rust-binaries-smaller-by-default.html` においても示されているが，Rust のデフォルトのバイナリサイズを小さくするような修正が Nightly Rust という，実験的な機能を含むバージョンである Rust 言語に適用された．これが今後 Rust 言語の標準環境にも組み込まれる可能性が高いことを考えると，以降で述べるような対策は特段不要になる未来が訪れるかもしれないことを付記しておく．

release ビルドでのコンパイル

単純に，”cargo build –release”のようにしてビルドする（以降このビルドのことを release ビルドと呼ぶ）だけでも，バイナリサイズを削減することが出来る．これは成果物のバイナリファイルにシンボル情報などを含めないようにすることによって実現されている．

panic 時の動作変更

Rust 言語では，デフォルトの動作として，panic（プログラムの回復不能な停止を指す）した際にバックトレースを生成するようになっている．これは panic 直前にプログラムがどのような処理を行っていたかを示すものであるが，このバックトレース生成部分のコードを省略するオプションが存在し，これを設定することで成果物のバイナリは，panic 時にバックトレースを生成せず，結果としてバイナリサイズを削減することができる．

具体的には，Cargo.toml に次のような設定を追加することで，release ビルドにおいてバックトレース生成部分のコードを省略することができる．

```
[profile.release]
panic = 'abort'
```

シンボル情報の削除

コンパイル作業の成果物であるバイナリ中には，デフォルトでシンボル情報が含まれている．これを削除してビルドする設定は，次のようになる．

```
[profile.release]
strip = true
```

コンパイラによるサイズ最適化

Rust 言語のコンパイラは，設定によっては，コンパイル時に最適化を行うことができる．これを行うには，Cargo.toml に次のような設定を追加することで，release ビルドにおいて最適化を行うことができる．

```
[profile.release]
opt-level = "z"
```

LTO の有効化

LTO（Link Time Optimization）は，コンパイラによるリンク時の最適化のことを指す．これを release ビルドで有効化するには，次のように Cargo.toml に記述する．

```
[profile.release]
lto = true
```

並列コードユニット数の減少

Rust コンパイラがコードを生成する際，何個かのコード生成ユニットに分割して処理を行う．この個数を増やすとコンパイル時間は短縮されるが，最適化が阻害される場合がある．このコード生成ユニット数はデフォルトでは 16 であるので，これを減らすには，次のように設定すればよい．

```
[profile.release]
codegen-units = 1
```

結果

本節では，これまでに述べた対策を講じた結果，どの程度バイナリサイズを削減することができたかについて報告する．

手早く結論を示すと，これらの対策を講じた結果，バイナリサイズは約 14MB から約 9.1MB に削減することができた．

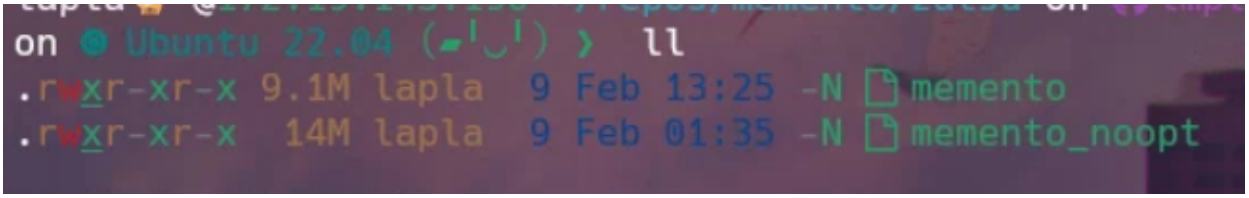

図 12　バイナリサイズの削減結果を示した図．memento_noopt が最適化を講じないで release ビルドしたバイナリとなっている．

このサイズ最適化を施した結果，手元の GDB よりは小さいバイナリとすることができたので，個人的には及第点であると考えている．

まとめ及び本講義に対する所感

本実験では，タイムトラベルデバッグを行うことができる簡易なデバッガを Rust 言語を用いて開発した．タイムトラベルデバッグを補助する様々な機能と共に実装することで，簡単なタイムトラベルデバッグを行えることを確認できた．また，既存の課題を洗い出し，今後の展望も大まかではあるが俯瞰することができた．

また，本講義全体を通した感想を最後に述べることとする．

（以下略）