高度サイバーセキュリティPBL II
2023 デバッガ

株式会社ティアフォー / 大阪大学
高野 祐輝

もくじ

- コンパイラの基本
- システムコール
- デバッガ
- 割り込みとシグナル
- デバッガの実装
- （DWARF）

コンパイラの基本

代表的なオープンソースコンパイラ

- GCC (GNU Compiler Collection)
  - GNUが作成しているコンパイラ
  - もともとはリチャード・ストールマン1987年に作り始めた
  - gcc: Cコンパイラ、g++: C++コンパイラ
- LLVM (Low Level Virtual Machine)
  - 現在MacOS、FreeBSDなどで標準で利用されるコンパイラ
  - 2000年にイリノイ大学で開発される
  - GCCより効率よく最適化を行うコンパイラとして作成された
  - clang: Cコンパイラ、clang++: C++コンパイラ
- RustはバックエンドにLLVMを利用

コンパイルの流れ

- ソースコードからマクロ展開し、字句・構文。その後意味解析してコード生成

- 最近のコンパイラはもう少し複雑

マクロ

- C言語で#からはじまる命令はマクロと呼ばれ、コンパイルの前に展開される
  #includeや#defineなど

- Rustは手続き型マクロなど、高機能なマクロを提供

```
#define OR(A, B) A || B
int f(a, b) {
  return OR(a, b);
}
```

マクロ展開

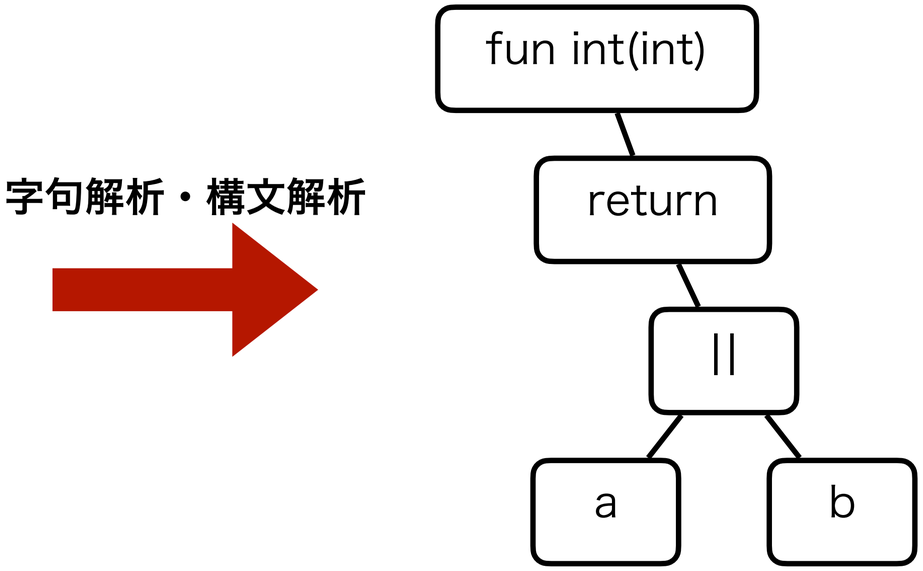
```
int f(a, b) {
  return a || b;
}
```

字句解析・構文解析

- マクロ展開後、字句解析、構文解析を経てソースコードは抽象構文木に変換される

```
int f(a, b) {
  return a || b;
}
```

字句解析・構文解析

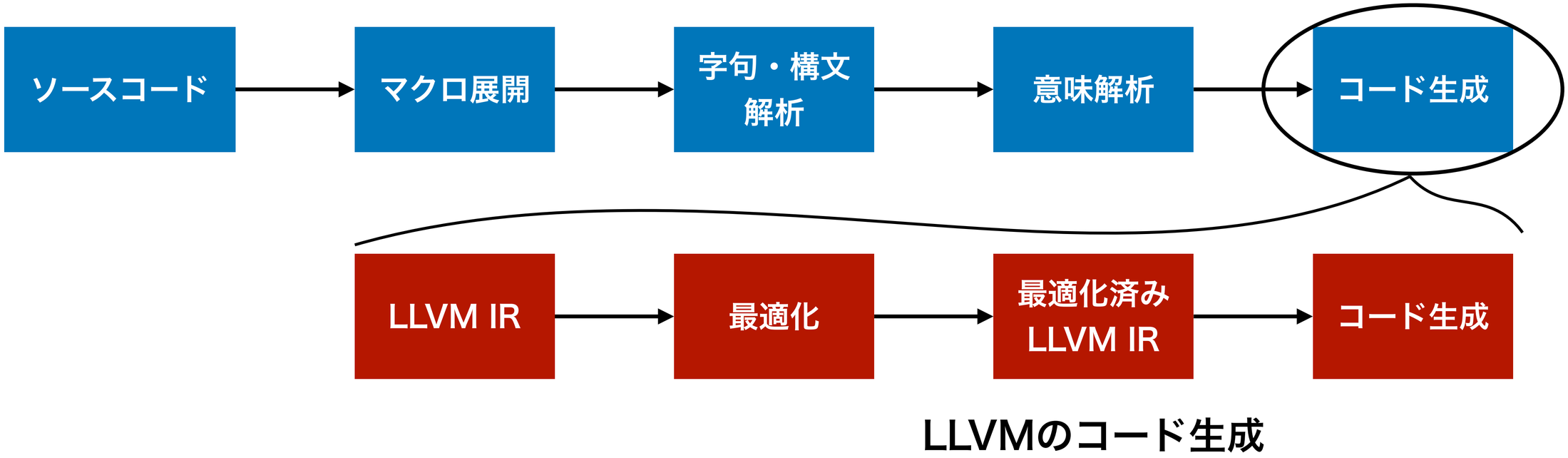


意味解析で行うこと

- 変数や関数が正しい名前かどうかの検査
- 変数や関数のスコープの検査
- 型検査
- 警告を出すべきかの検査
- など

LLVMの最適化方法とコード生成

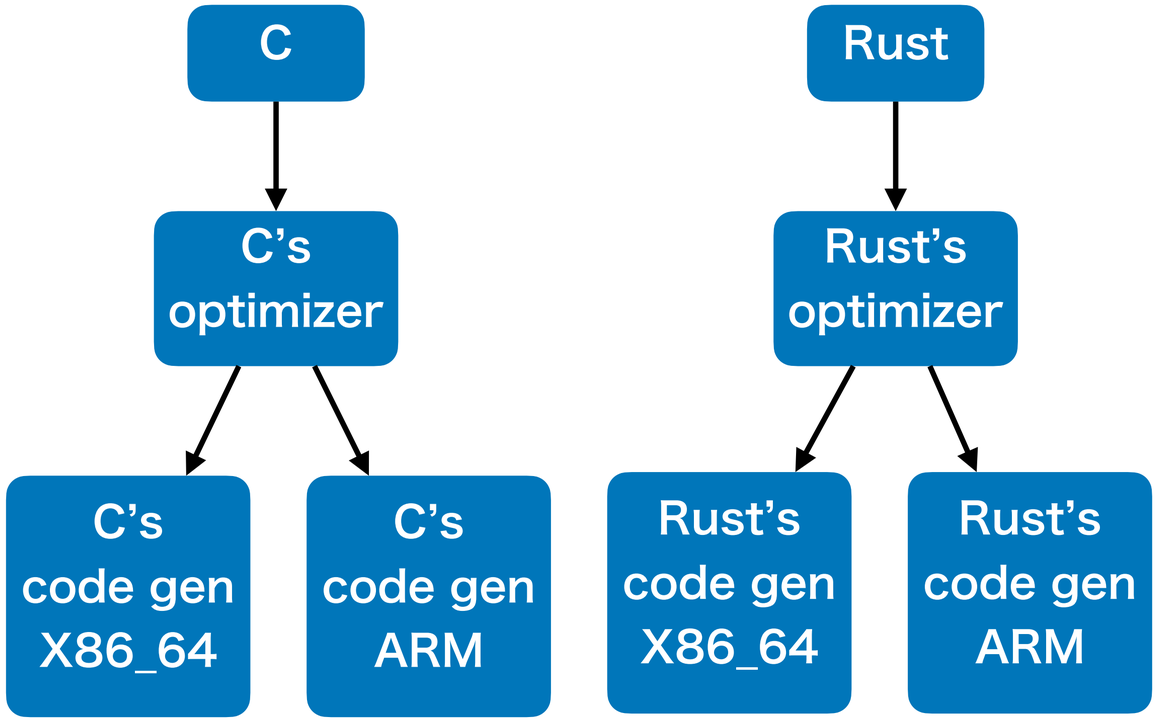
- LLVMでは、ソースコードをLLVM Intermediate Representation (LLVM IR)と呼ぶ中間表現に変換して最適化を行う



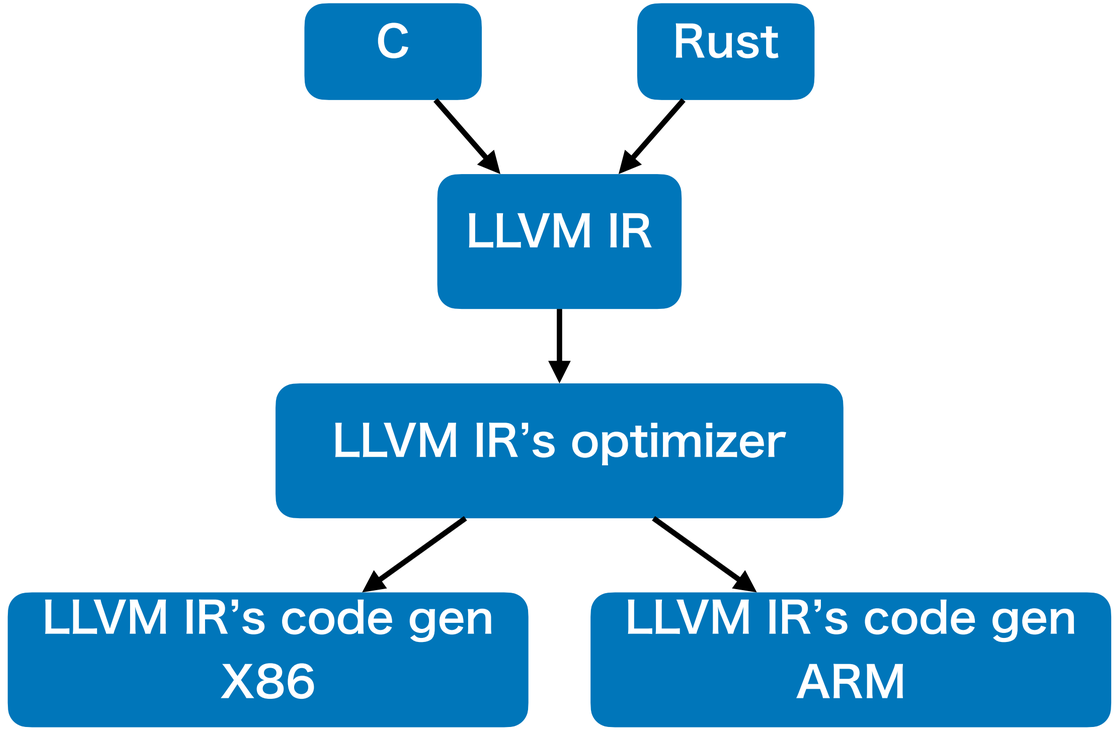
LLVMのコード生成

LLVM IRを使う利点

- 最適化、コード生成を複数のプログラミング言語ごとに用意しなくてよい



各言語独自実装



LLVM IRを利用した実装

CPUアーキテクチャ

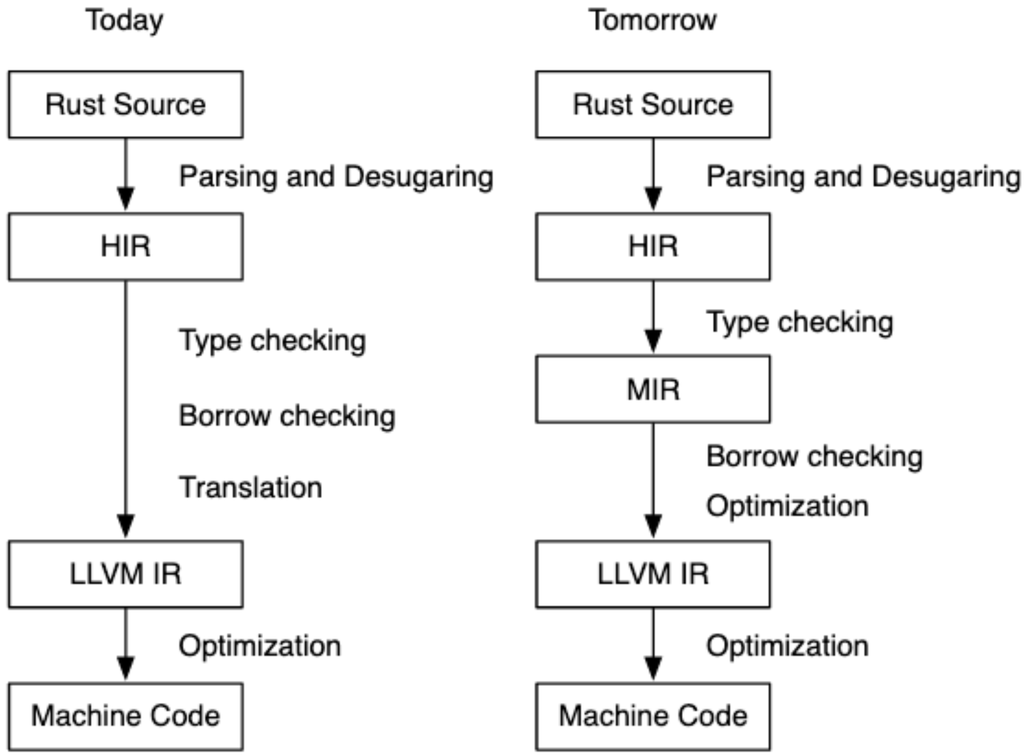
- AMD64（またはx86_64）：Intelのx86アーキテクチャと互換性のある64ビットCPUアーキテクチャ。普段使っているPC、PlayStation 4はこれ。
- ARM：最近ソフトバンクが買収したことで有名なアーキテクチャ。普段使っているスマートフォンやタブレットはこれ。Nintendo Switchはこれ。
- MIPS：有名なCPUの教科書、通称ヘネ・パタ本でも使われているアーキテクチャ。RISC CPUの元祖。2018年にヘネシーとパターソンはチューリング賞を受賞。PlayStation, PlayStation 2はこれ。
- 他にも、Power、SPARC、SHなど色々ある

中間言語

- 実際のコンパイラでは、ソースコードから、さらに中間言語と呼ばれる別の表現に何度か、変換して最終的に実行バイナリを出力する
- 中間言語は英語ではIntermediate LanguageなのでIRと略される

Rustの中間コード

- High-level IR
- Typed High-level IR
- Mid-level IR



https://blog.rust-lang.org/2016/04/19/MIR.html

Rustでのアセンブリの出力

`cargo rustc -- --emit asm`
と指定すると、

target/debug/deps以下に、.sファイルが生成される

```
$ cargo rustc -- --emit asm
$ cat target/debug/deps/hello-98fdac5d7aca4307.s
```

RustでのLLVM IRの出力

`rustc -- --emit=llvm-ir`
と指定すると、
target/debug/deps以下に、.llファイルが生成される

```
$ cargo new hello
$ cd hello
$ cargo rustc -- --emit=llvm-ir
$ cat target/debug/deps/hello-98fdac5d7aca4307.ll
```

ミニ演習

- 適当なRustのプログラムを実装し、LLVM IRとアセンブリを取得してみよ

システムコール

システムコール

- システムコールとはOSの提供するAPI
- OS管理下のリソースを扱うために利用
  - メモリ、ネットワークソケット、ファイル、プロセスなど
- システムコールを呼び出すと、ユーザランドからOSに処理が移行
- ソフトウェア割り込みとして実現される

システムコールの例

- read, write, open, close: ファイル読み書き、オープン、クローズ
- socket, connect, sendto: ソケット操作
- fork, exec: プロセス生成等

システムコール等の関数呼び出しに必要なヘッダや、関数の引数、返り値の調べ方

- UNIX系OSの場合は基本的にmanでマニュアルを参照すると調べることが可能
- システムコールはmanのセクション2にあるため、システムコールを調べたい場合は以下のようにする
  $ man 2 write
  $ man open
- writeのエントリはシステムコールとユーザーコマンド（セクション1）にあるため、セクションを指定する必要あり
- openのエントリはシステムコールのみにあるため、セクション番号は省略可能
- 大事なことは大体manに書いてある

manの例

```
$ man 2 write
WRITE(2)                    Linux Programmer's Manual                    WRITE(2)

NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write() writes up to count bytes from the buffer starting at buf to the
       file referred to by the file descriptor fd.

       The number of bytes written may be less than  count  if,  for  example,
       there  is  insufficient space on the underlying physical medium, or the
       RLIMIT_FSIZE resource limit is encountered (see setrlimit(2)),  or  the
       call was interrupted by a signal handler after having written less than
       count bytes.  (See also pipe(7).)
```

manのセクション

$ man man
MAN(1)　　Manual pager utils　　MAN(1)

NAME
　　man - an interface to the on-line reference manuals
中略
　　1　Executable programs or shell commands
　　2　System calls (functions provided by the kernel)
　　3　Library calls (functions within program libraries)
　　4　Special files (usually found in /dev)
　　5　File formats and conventions eg /etc/passwd
　　6　Games
　　7　Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
　　8　System administration commands (usually only for root)
　　9　Kernel routines [Non standard]

strace

- straceコマンドを用いると、プログラム実行中のシステムコール呼び出しを監視することが可能

- 実行方法は、コマンドを指定するかプロセスIDを指定する
  $ strace コマンド
  $ strace -p PID

- 例えば、ソースコードを変更せずに、プロセスの入出力を取得することが可能

straceの利用例

```
$ cargo new hello
$ cd hello
$ cargo build
$ strace ./target/debug/hello
sched_getaffinity(28301, 32, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15]) = 8
write(1, "Hello, world!\n", 14Hello, world!
)         = 14
sigaltstack({ss_sp=NULL, ss_flags=SS_DISABLE, ss_size=8192}, NULL) = 0
munmap(0x7fafd3738000, 12288)           = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

straceで隠されたログ出力を覗き見る（1/2）

```rust
use std::{fs::File, io::prelude::*};

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut file = File::create("/dev/null")?;
    file.write_all(b"Hello, world!")?;
    Ok(())
}
```

/dev/nullという、書き込んだデータが捨てられる特殊なデバイスファイルにメッセージを書き込むプログラム

コンパイルして実行しても何も表示されない

straceで隠されたログ出力を覗き見る（2/2）

```
$ strace ./target/debug/write_null
sched_getaffinity(29355, 32, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]) = 8
openat(AT_FDCWD, "/dev/null", O_WRONLY|O_CREAT|O_TRUNC|O_CLOEXEC, 0666) = 3
write(3, "Hello, world!", 13)                = 13
close(3)                                     = 0
sigaltstack({ss_sp=NULL, ss_flags=SS_DISABLE, ss_size=8192}, NULL) = 0
munmap(0x7f1914a68000, 12288)                = 0
exit_group(0)                                = ?
+++ exited with 0 +++
```

UNIXのスペシャルデバイスファイル

- /dev/null
  書き込したデータは全て捨てられ、読み込むとファイル終端を返す

- /dev/zero
  読み込むと常に0を返す

- /dev/random, /dev/urandom
  乱数値を返す。randomはブロッキングで、urandomは非ブロッキング

- /dev/full
  常にディスク容量が最大であるエラーを返すファイル

よくある使い方

出力を捨てる

```
$ echo hello > /dev/null
```

ディスク使用量が100%になったときのファイル書き込み時の挙動をテストする

```
$ echo hello > /dev/full
-bash: echo: write error: No space left on device
```

ミニ演習

- manをつかってシステムコールのマニュアルをみてみよ
- /dev/nullにメッセージを書き込むプログラムを実装し、straceでシステムコールトレースしてみよ
  - straceは下のコマンドでインストールすること
  - $ sudo apt install strace　**(通常のLinux環境の場合はこちら)**
  - # apt install strace　**(Dockerの場合はこちら)**

デバッガ

デバッガ

- デバッガとは、プログラムの動作を実行時に解析するためのソフトウェアの総称
- 有名なデバッガとして、GNUのgdb、LLVMのlldbなどがある
- プログラムのステップ実行、実行中メモリ内容の確認、スタックトレースなどを行うことができる

デバッガの仕組み

- デバッグ情報が埋め込まれた実行バイナリをデバッガで動作させる
- デバッグ情報に含まれる情報
  - アドレスとコンパイルユニット（ソースコードに相当）の対応
  - call stack情報（関数呼び出し時のメモリと値の関係）
  - 関数名や引数の名前

デバッグ情報を埋め込んでコンパイル

- cargoでコンパイルすると、デフォルトではデバッグ情報を埋め込んでコンパイルされる

- 最適化してコンパイルする場合は--releaseオプションをつけてコンパイル

```
$ cargo build --release
$ ls ./target/release
```

デバッガの起動とプログラムの実行

lldb helloというコマンドで、helloというプログラムのデバッグを開始し、runで実行

```
$ lldb hello
(lldb) target create "./hello"
Current executable set to './hello' (x86_64).
(lldb) run
Process 27313 launched: '/home/vagrant/program/nhello/hello' (x86_64)
Process 27313 exited with status = 0 (0x00000000)
```

ブレークポイント

- ブレークポイントとは、プログラムを一時停止するポイントのこと
- プログラムを停止した状態で変数の値などを見ることが可能

ブレークポイントの設定

関数名を指定

```
(lldb) breakpoint set -n main
Breakpoint 1: where = hello`main + 22 at nhello.c:9:11, address =
0x0000000000401156
```

ファイル名と行番号を指定

```
(lldb) breakpoint set -f hello.c -l 9
Breakpoint 1: where = hello`main + 22 at hello.c:9:11, address =
0x0000000000401156
```

ブレークポイントの一覧と削除

ブレークポイント一覧

```
(lldb) breakpoint list
Current breakpoints:
1: file = 'hello.c', line = 9, exact_match = 0, locations = 1
  1.1: where = hello`main + 22 at hello.c:9:11, address =
0x0000000000401156, unresolved, hit count = 0
```

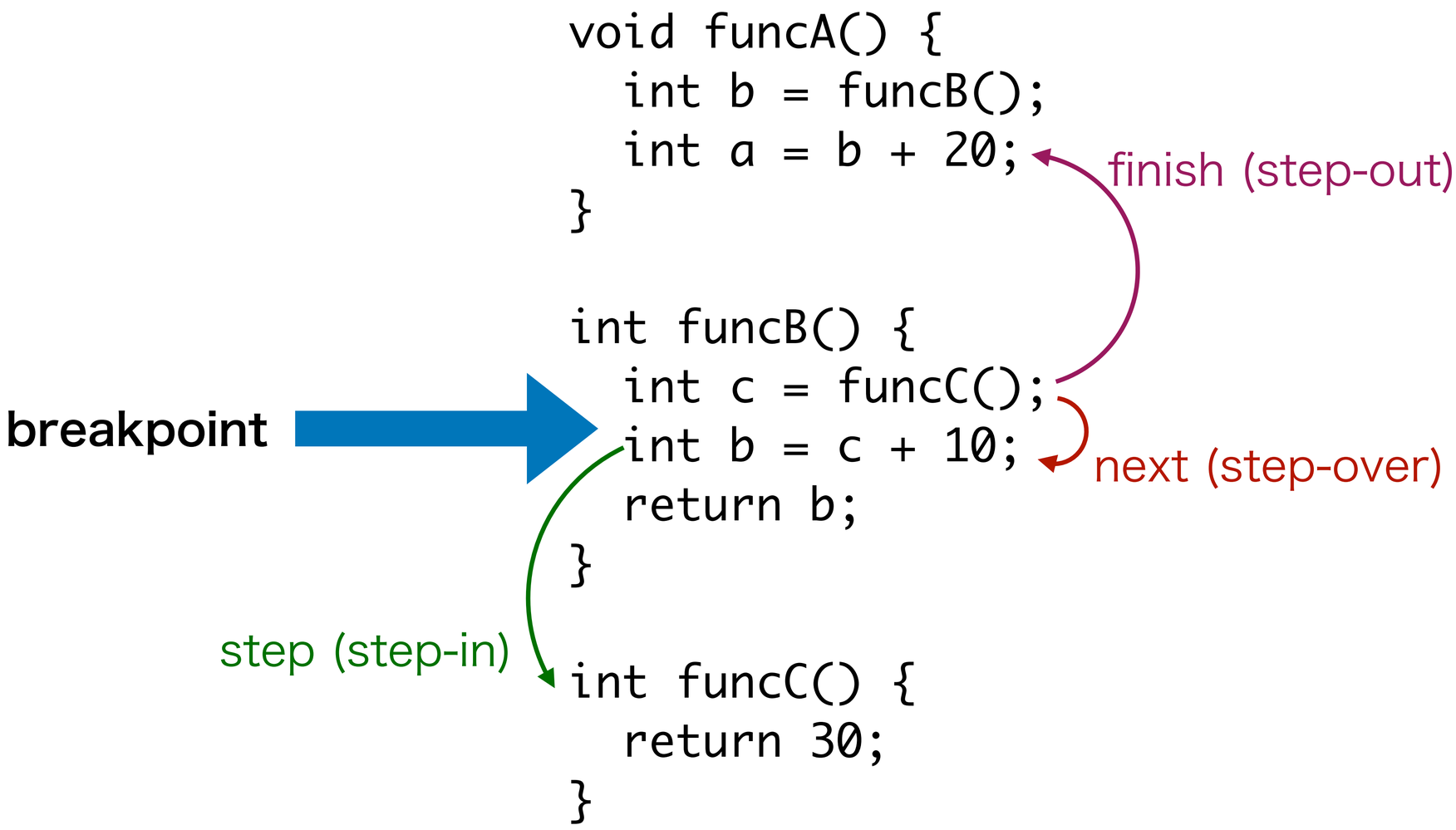
1番目のブレークポイント削除（ちなみに数字を指定しないと全削除）

```
(lldb) breakpoint delete 1
1 breakpoints deleted; 0 breakpoint locations disabled.
```

ステップ実行

- ブレークポイントでプログラム停止後、ソースコードレベルで1行ごとプログラムを実行可能
- ステップ実行の種類
  - step: 1行実行する。関数の場合は関数の中を実行する。step-in
  - next: 1行実行する。関数であっても中には入らない。step-over
  - finish: 現在のスタックフレームを終了する。step-out

ステップ実行のイメージ



ステップ実行の例: 準備

- まず、右のソースコードをコンパイル
  - -gと-O0フラグを指定すること
- その後lldbをa.outを指定して実行

ソースコードの完全版

```
int funcB();
int funcC();

void funcA() {
  int b = funcB();
  int a = b + 20;
}

int funcB() {
  int c = funcC();
  int b = c + 10;
  return b;
}

int funcC() {
  return 30;
}

int main() {
  funcA();
  return 0;
}
```

ステップ実行の例: コンパイルとブレークポイント

```
$ gcc -g -O0 main.c
$ breakpoint set -f main.c -l 8
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ModuleNotFoundError: No module named 'lldb.embedded_interpreter'
(lldb) target create "a.out"
Current executable set to '/home/ytakano/program/c/debugger_step/a.out' (x86_64).
(lldb) breakpoint set -f main.c -l 8
Breakpoint 1: where = a.out`funcB + 12 at main.c:10:11, address = 0x000000000000115a
(lldb) r
Process 976425 launched: '/home/ytakano/program/c/debugger_step/a.out' (x86_64)
Process 976425 stopped
* thread #1, name = 'a.out', stop reason = breakpoint 1.1
    frame #0: 0x000055555555515a a.out`funcB at main.c:10:11
   7   }
   8
   9   int funcB() {
-> 10    int c = funcC();
   11    int b = c + 10;
   12    return b;
   13  }
```

ステップ実行の例: nextとquit

```
(lldb) next
Process 976425 stopped
* thread #1, name = 'a.out', stop reason = step over
    frame #0: 0x0000555555555167 a.out`funcB at main.c:11:7
   8
   9    int funcB() {
   10     int c = funcC();
-> 11     int b = c + 10;
   12     return b;
   13   }
   14
(lldb) quit
Quitting LLDB will kill one or more processes. Do you really want to proceed: [Y/n] Y
```

変数値の確認

- print、または省略してpのあとに変数名を入力すると、変数値が確認可能

```
(lldb) r
Process 2584 launched: '/tinydbg/src/hello' (x86_64)
Process 2584 stopped
* thread #1, name = 'hello', stop reason = breakpoint 1.1
    frame #0: 0x0000000000401146 hello`main(argc=1, argv=0x00007fffffffec48) at hello.c:4:5
   1    #include <stdio.h>
   2
   3    int main(int argc, char *argv[]) {
-> 4        printf("hello\n");
   5        printf("world\n");
   6        return 0;
   7    }
(lldb) p argc
(int) $0 = 1
```

バックトレース

- 関数呼び出しのスタックフレームをトレース可能
- 例外などでプログラムが停止した際に、スタックフレームを追跡することで問題箇所を特定可能

Segmentation Fault
の起きるプログラム

```
$ cat bt.c
#include <stddef.h>

void b(int *n) {
        *n += 10;
}

void a(int *m) {
        int n = 100;
        b(&n);
        *m += n;
}

int main(int argc, char *argv[]) {
        a(NULL);
        return 0;
}
```

コンパイルして実行するとエラー

```
$ clang -g -O0 bt.c -o bt
$ ./bt
Segmentation fault (core dumped)
```

バックトレースの例

```
$ lldb bt
(lldb) target create "bt"
Current executable set to 'bt' (x86_64).
(lldb) run
Process 32098 launched: '/home/vagrant/program/bt/bt' (x86_64)
Process 32098 stopped
* thread #1, name = 'bt', stop reason = signal SIGSEGV: invalid address (fault address: 0x0)
    frame #0: 0x0000000000401153 bt`a(m=0x0000000000000000) at bt.c:10:5
   7    void a(int *m) {
   8            int n = 100;
   9            b(&n);
-> 10           *m += n;
   11   }
   12
   13   int main(int argc, char *argv[]) {
(lldb) bt
* thread #1, name = 'bt', stop reason = signal SIGSEGV: invalid address (fault address: 0x0)
  * frame #0: 0x0000000000401153 bt`a(m=0x0000000000000000) at bt.c:10:5
    frame #1: 0x0000000000401182 bt`main(argc=1, argv=0x00007fffffffe4e8) at bt.c:14:2
    frame #2: 0x00007ffff7dffb6b libc.so.6`__libc_start_main + 235
    frame #3: 0x000000000040104a bt`_start + 42
```

スタックフレームの選択

```
(lldb) bt
* thread #1, name = 'bt', stop reason = signal SIGSEGV: invalid address (fault address: 0x0)
  * frame #0: 0x0000000000401153 bt`a(m=0x0000000000000000) at bt.c:10:5
    frame #1: 0x0000000000401182 bt`main(argc=1, argv=0x00007fffffffe4e8) at bt.c:14:2
    frame #2: 0x00007ffff7dffb6b libc.so.6`__libc_start_main + 235
    frame #3: 0x000000000040104a bt`_start + 42
(lldb) f 1
frame #1: 0x0000000000401182 bt`main(argc=1, argv=0x00007fffffffe4e8) at bt.c:14:2
   11   }
   12
   13   int main(int argc, char *argv[]) {
-> 14           a(NULL);
   15           return 0;
   16   }
   17
```

f フレーム番号

とすることで解析したいフレームを選択可能

ミニ演習

- デバッガを利用して以下の挙動を実際に確認せよ
  - ブレークポイント設定
  - ステップ実行
  - バックトレース

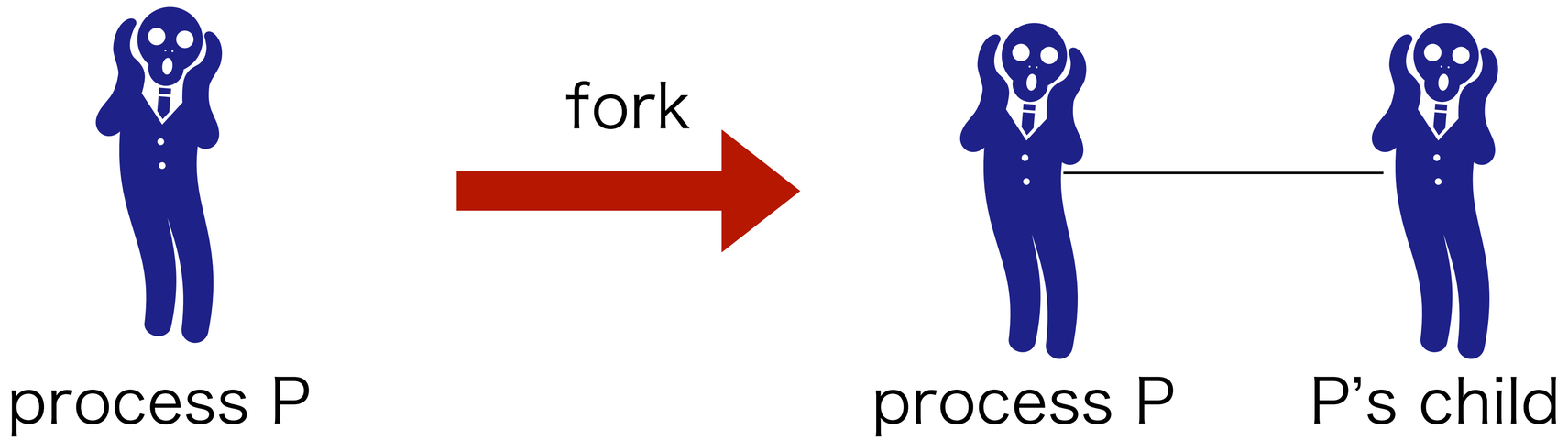
デバッガ実装のための前知識

デバッガ実装のための前知識

- プロセスの起動と停止：fork, exec, wait
- シグナル：signal, raise
- 割り込み、例外
- DWARF

子プロセス生成、fork

- forkを呼び出すと、子プロセスを生成
- 全く同じプログラムの子プロセスが出来上がる



forkのコード

```
use nix::{
    sys::wait::waitpid,
    unistd::{fork, write, ForkResult},
};

match unsafe { fork() } {
    Ok(ForkResult::Parent { child, .. }) => {
        println!("親プロセス : 子プロセスのpid: {}", child);
        waitpid(child, None).unwrap();
    }
    Ok(ForkResult::Child) => {
        // `println!`はforkした直後は使わない
        write(nix::libc::STDOUT_FILENO, "子プロセス\n".as_bytes()).ok();
        unsafe { nix::libc::_exit(0) };
    }
    Err(_) => println!("forkに失敗"),
}
```

parent

child

forkは二度返る

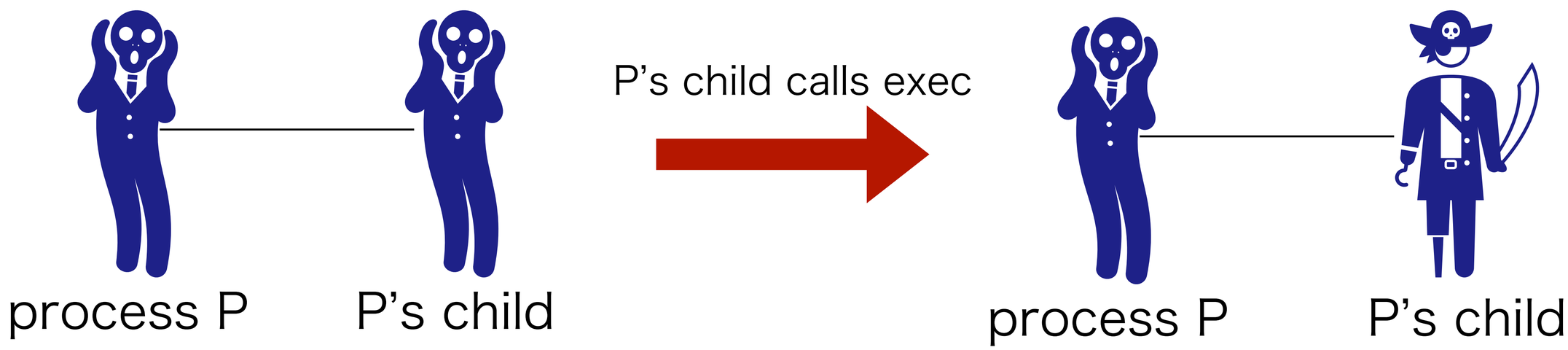
子プロセス、親プロセス

子プロセスの終了処理（後述）

https://docs.rs/nix/latest/nix/unistd/fn.fork.html

exec

- 実行ファイルを読み込み、現在と異なるプログラムを実行
- アプリケーションの起動は、まずforkしたあとにexecが実行されて実現



execvp関数

```
use nix::{
    sys::wait::waitpid,
    unistd::{fork, write, ForkResult},
};

// 実行ファイルをメモリに読み込み（子プロセス内で実行）
match nix::unistd::execvp(&filename, &args) {
    Err(_) => {
        nix::unistd::write(libc::STDERR_FILENO, "不明なコマンドを実行\n".as_bytes()).ok();
        exit(1);
                }
    Ok(_) => unreachable!(),
}
```

第一引数：実行ファイルへのパス

第二引数：コマンドライン引数

https://docs.rs/nix/latest/nix/unistd/fn.execvp.html

fork & execの例

```
use nix::{
    sys::wait::waitpid,
    unistd::{execvp, fork, write, ForkResult},
};
use std::ffi::CString;

match unsafe { fork() } {
    Ok(ForkResult::Parent { child, .. }) => {
        println!("親プロセス : 子プロセスのpid: {}", child);
        waitpid(child, None).unwrap();
    }
    Ok(ForkResult::Child) => {
        // `println!`はforkした直後は使わない
        write(nix::libc::STDOUT_FILENO, "子プロセス\n".as_bytes()).ok();
        let filename = CString::new("/usr/bin/echo").unwrap();
        match execvp(&filename, &[&filename]) {
            Err(_) => {
                nix::unistd::write(
                    nix::libc::STDERR_FILENO,
                    "不明なコマンドを実行\n".as_bytes(),
                )
                .ok();
                unsafe { nix::libc::_exit(0) };
            }
            Ok(_) => unreachable!(),
        }
    }
    Err(_) => println!("forkに失敗"),
}
```

コマンドライン引数の0番目には、実行ファイルを渡す

execファミリー

C言語のAPIはmanで確認可能。nixでは一部のみ実装

```
int
execl(const char *path, const char *arg0, ... /*, (char *)0 */);

int
execle(const char *path, const char *arg0, ...
    /*, (char *)0, char *const envp[] */);

int
execlp(const char *file, const char *arg0, ... /*, (char *)0 */);

int
execv(const char *path, char *const argv[]);

int
execvp(const char *file, char *const argv[]);

int
execvP(const char *file, const char *search_path, char *const argv[]);
```

lがつく関数が可変長引数で引数指定

lがつく関数が配列で引数指定

pがつく関数が環境変数PATHから検索

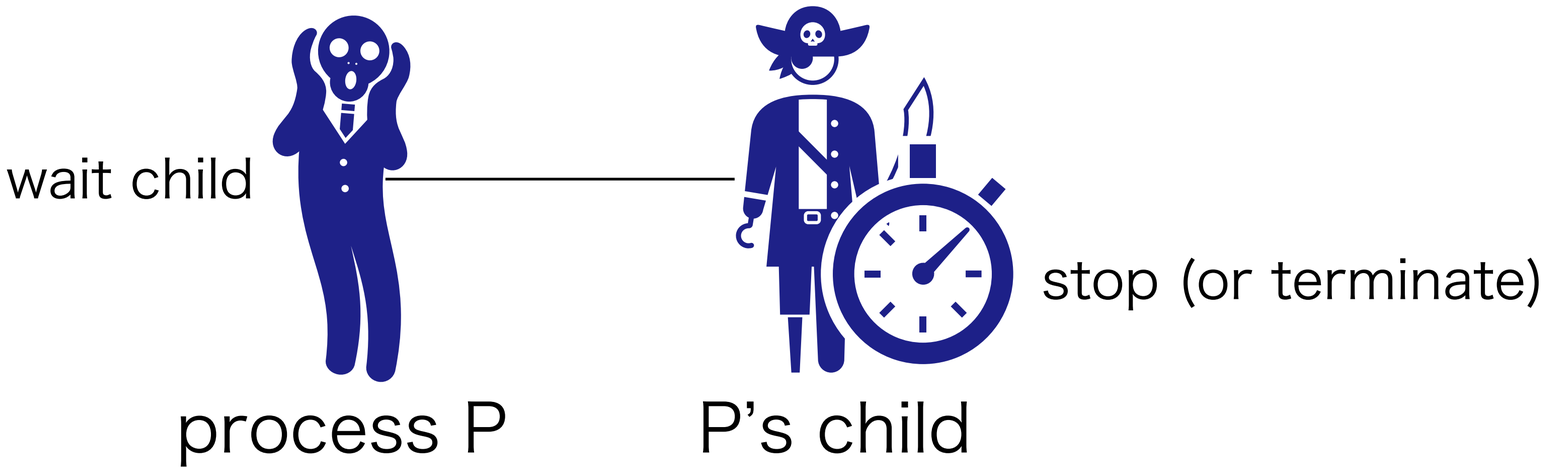
eがつく関数が環境変数を指定可能

execvPは検索パスを指定可能

（たくさんあって正直覚えられないので、使うときはmanで参照すること）

wait, waitpid

- プロセスが停止するのを待つ
- プロセスが終了していた場合は、プロセスの情報を解放する



https://docs.rs/nix/latest/nix/sys/wait/fn.waitpid.html

waitpidの返り値

https://docs.rs/nix/latest/nix/sys/wait/enum.WaitStatus.html

`Exited(Pid, i32)`

The process exited normally (as with `exit()` or returning from `main`) with the given exit code. This case matches the C macro `WIFEXITED(status)`; the second field is `WEXITSTATUS(status)`.

`Signaled(Pid, Signal, bool)`

The process was killed by the given signal. The third field indicates whether the signal generated a core dump. This case matches the C macro `WIFSIGNALED(status)`; the last two fields correspond to `WTERMSIG(status)` and `WCOREDUMP(status)`.

`Stopped(Pid, Signal)`

The process is alive, but was stopped by the given signal. This is only reported if `WaitPidFlag::WUNTRACED` was passed. This case matches the C macro `WIFSTOPPED(status)`; the second field is `WSTOPSIG(status)`.

`PtraceEvent(Pid, Signal, c_int)`

The traced process was stopped by a `PTRACE_EVENT_*` event. See `nix::sys::ptrace` and ptrace(2) for more information. All currently-defined events use `SIGTRAP` as the signal; the third field is the `PTRACE_EVENT_*` value of the event.

`PtraceSyscall(Pid)`

The traced process was stopped by execution of a system call, and `PTRACE_O_TRACESYSGOOD` is in effect. See ptrace(2) for more information.

`Continued(Pid)`

The process was previously stopped but has resumed execution after receiving a `SIGCONT` signal. This is only reported if `WaitPidFlag::WCONTINUED` was passed. This case matches the C macro `WIFCONTINUED(status)`.

`StillAlive`

There are currently no state changes to report in any awaited child process. This is only returned if `WaitPidFlag::WNOHANG` was used (otherwise `wait()` or `waitpid()` would block until there was something to report).

man waitpid

C言語のAPIはmanで確認可能

```
int status;
waitpid(pid, &status, 0);
```

```
WIFEXITED(status)
       True if the process terminated normally by a call to _exit(2) or exit(3).

WIFSIGNALED(status)
       True if the process terminated due to receipt of a signal.

WIFSTOPPED(status)
       True if the process has not terminated, but has stopped and can be
       restarted.  This macro can be true only if the wait call specified the
       WUNTRACED option or if the child process is being traced (see ptrace(2)).
```

zombie process

- 親プロセスが適切に子プロセスをwaitしない場合、ゾンビプロセスとして残ってしまう
- リソースが無駄に消費されてしまう
- 親プロセスで適切に子プロセスの終了処理を行うか、子プロセスを親プロセスから切り離す（デーモン化など）必要がある

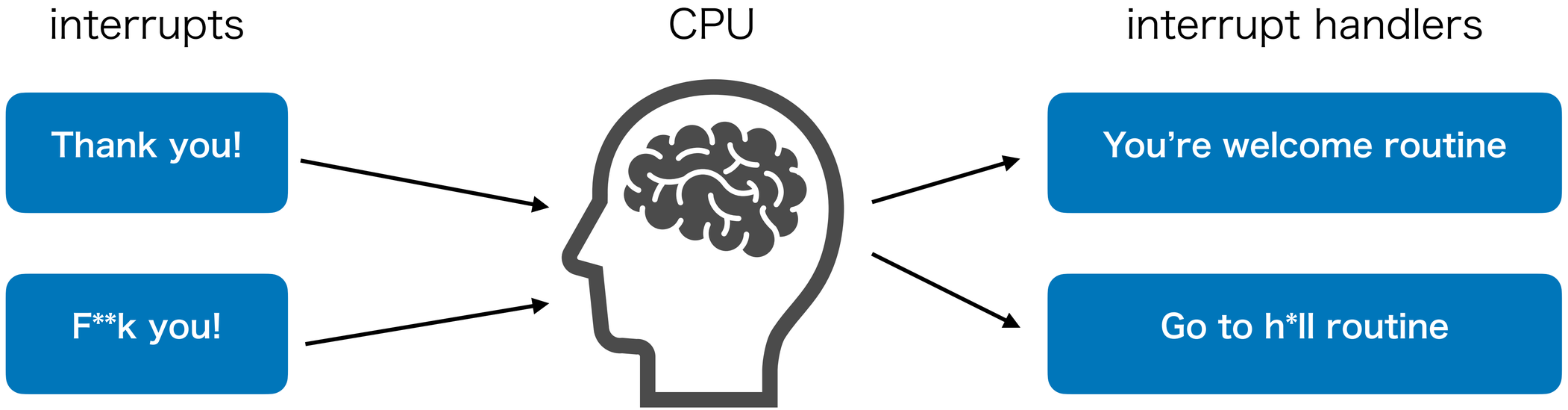
ミニ演習

- fork, execを使ったプログラムを実装し、動作を確認せよ

割り込みとシグナル

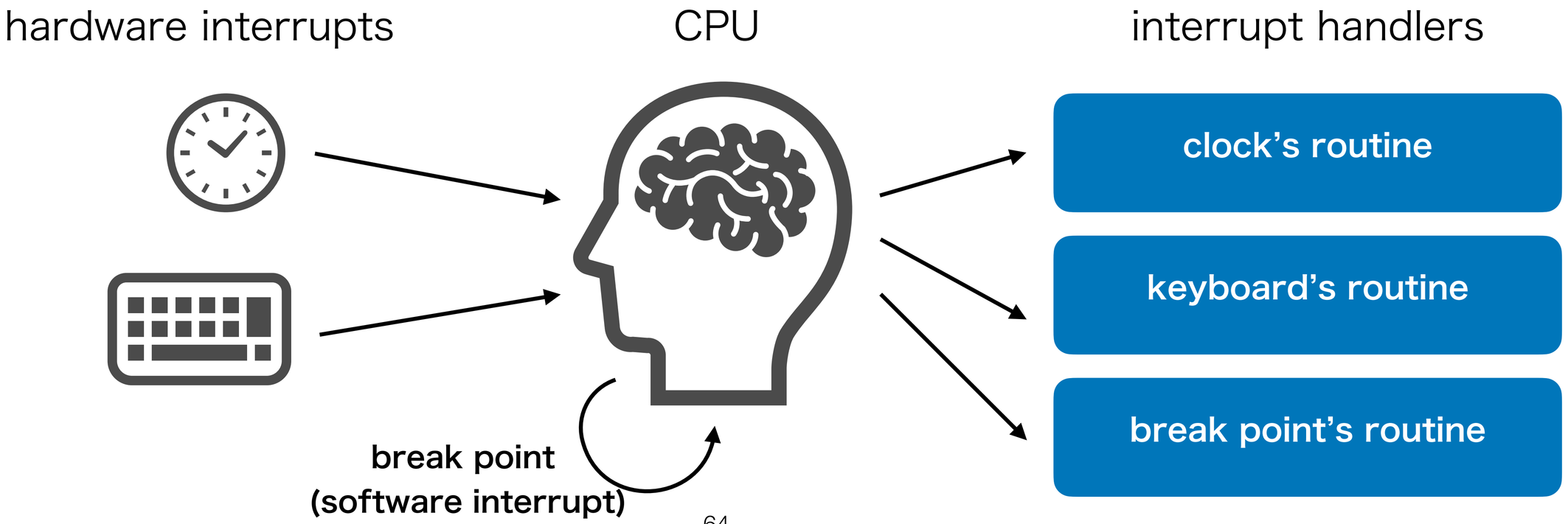
割り込み

- プログラム実行中に発生する、なにか別のイベントを割り込みと呼ぶ
- 割り込みを扱うための処理を、割り込みハンドラと呼ぶ



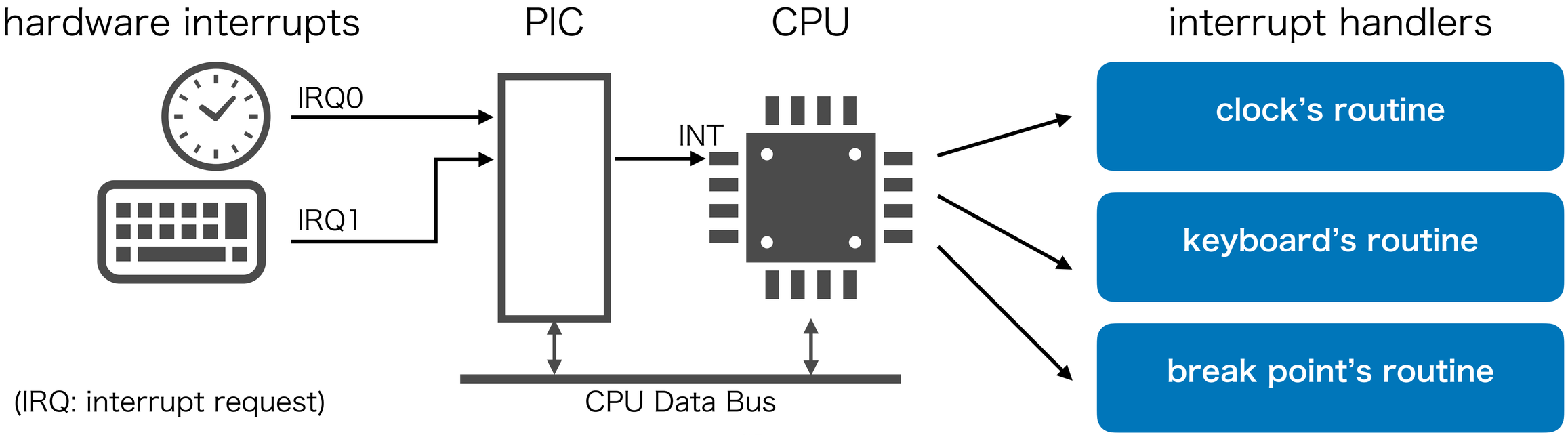
コンピュータの割り込み

- ハードウェア割り込み：CPU以外の周辺機器（キーボード、マウス、ネットワークカード）などから発生する割り込み
- ソフトウェア割り込み：CPU自身が発生するさせる割り込み。CPUの割り込み命令以外で発生するものは「例外」とも呼ばれる



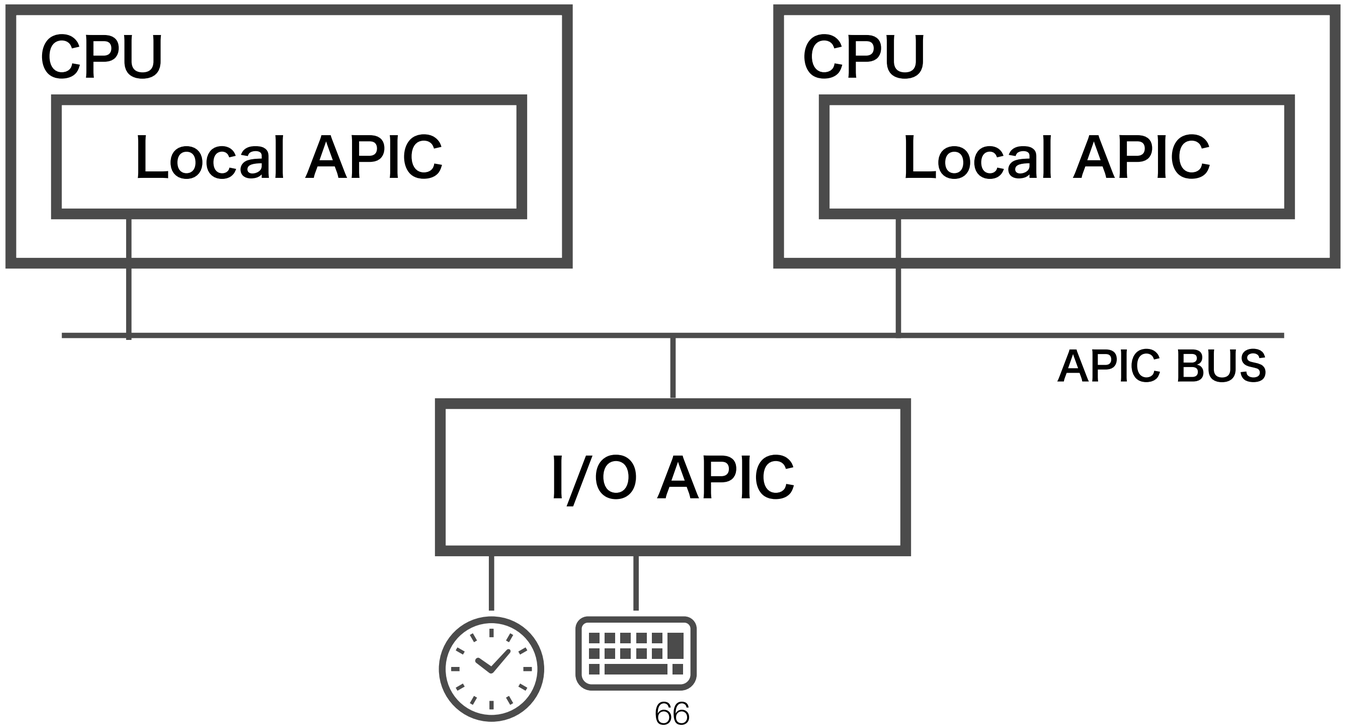
割り込みコントローラ

- ハードウェアからの割り込みを処理するためのコントローラ
- x86アーキテクチャの場合以下のコントローラが利用される
  Programmable Interrupt Controller (PIC)、
  Advanced Programmable Interrupt Controller (APIC)、
  eXtended APIC (xAPIC)、x2APIC



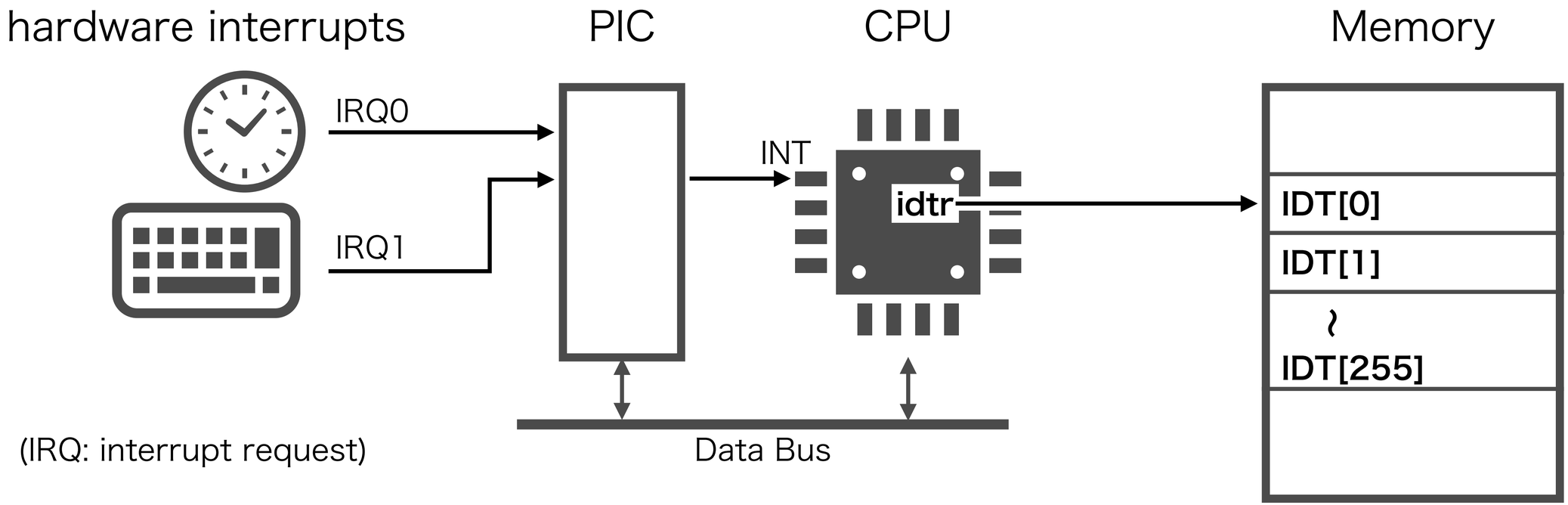
APIC

- PICは単一CPUのみで利用できたが、APICでは複数のCPUを利用可能
- ハードウェアとつながるI/O APICと、CPU内部のLocal APICという2つのコンポーネントから成り立つ



Interrupt Descriptor Table (IDT)

- 割り込み処理を行うための関数テーブル
- x86の場合、idtrレジスタがIDTへのエントリを指す



(IRQ: interrupt request)

例外の種類

- Faults：命令が正しく実行できない時に発生
- Traps：主にデバッグ用途で利用
- Aborts：ハードウェアエラーなど深刻なエラーで発生。プログラムは停止

x86の例外

#	内容	#	内容
0	division by zero（fault）	10	invalid TSS (fault)
1	debug (trap or fault)	11	segment not present (fault)
2	not used	12	stack segment fault (fault)
3	breakpoint (trap)	13	general protection (fault)
4	overflow (trap)	14	page fault (fault)
5	bounds check (fault)	15	reserved
6	invalid opcode (fault)	16	floating-point error (fault)
7	device not available (fault)	17	alignment check (fault)
8	double fault (abort)	18	machine check (fault)
9	coprocessor segment overrun (abort)	19	SIMD floating point exception (fault)

ミニ演習

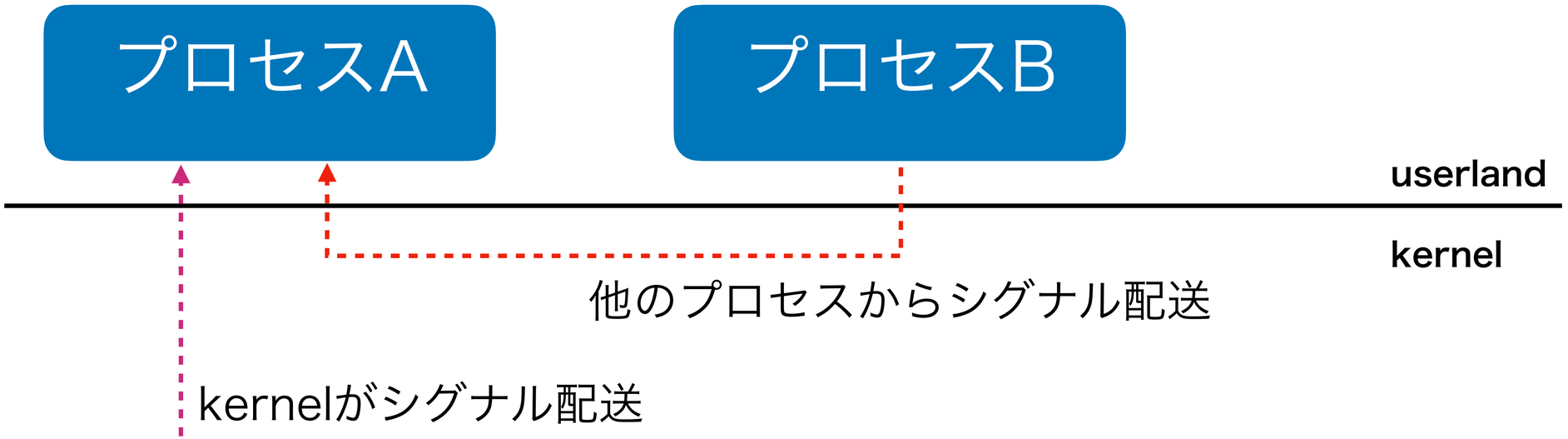
- int 3命令を実行するようなプログラムをインラインアセンブリを用いて実装し、デバッガでbreakとcontinueができることを確認せよ
- 上記プログラムをデバッガを用いずに実行してみて、プロセスが例外trapで終了することを確認せよ

Rustのソース中で、下記インラインアセンブリコードを挿入するとint 3命令を挿入可能

```
use std::arch::asm;
unsafe { asm!("int $3") };
```

シグナル

- シグナルとは割り込みの一種
- OSからユーザプロセスへ発行される割り込み



よく使われるシグナル

- SIGINT：Ctrl+c
- SIGHUP：設定ファイル再読み込みや、ログファイルへの出力などに用いられる事が多い。OSがシャットダウンされる際にプロセスに送信される
- SIGSTP：Ctrl+z。一時停止
- SIGTERM：killコマンドでデフォルトで送信されるシグナル。プロセスを正常終了させる
- SIGKILL：強制終了。絶対にプロセスを止めることができる
- SIGTRAP：デバッグ時のブレークポイントに利用

デフォルトの挙動

```
$ man 7 signal
Signal     Value     Action   Comment
──────────────────────────────────────────────────────────────────────
SIGHUP        1       Term    Hangup detected on controlling terminal
                              or death of controlling process
SIGINT        2       Term    Interrupt from keyboard
SIGQUIT       3       Core    Quit from keyboard
SIGILL        4       Core    Illegal Instruction
SIGABRT       6       Core    Abort signal from abort(3)
SIGFPE        8       Core    Floating-point exception
SIGKILL       9       Term    Kill signal
SIGSEGV      11       Core    Invalid memory reference
SIGPIPE      13       Term    Broken pipe: write to pipe with no
                              readers; see pipe(7)
SIGALRM      14       Term    Timer signal from alarm(2)
SIGTERM      15       Term    Termination signal
SIGUSR1   30,10,16    Term    User-defined signal 1
SIGUSR2   31,12,17    Term    User-defined signal 2
SIGCHLD   20,17,18    Ign     Child stopped or terminated
SIGCONT   19,18,25    Cont    Continue if stopped
SIGSTOP   17,19,23    Stop    Stop process
```

他

Term：終了

Ign：無視

Core：コアファイルを出力

Stop：停止

Cont：プロセスが停止していた場合、実行

killコマンド

- 他のプロセスへシグナルを配送するためのコマンド
- 暴走したプロセスをkillするために使うことが多い
- もともとは、シグナル配送ではなく、単にプロセスを終了させるためだけに使われていた名残

```
$ kill プロセスID
```

```
$ kill -シグナルの種類 プロセスID
```

Rustとsignal_hook

- Rustではsignal_hookというライブラリを利用してシグナルハンドラを記述するのが一般的

- SIGUSR1シグナルを受信し続けるコード例

```rust
fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut signals = signal_hook::iterator::Signals::new(&[signal_hook::consts::SIGUSR1])?;
    for signal in signals.forever() {
        match signal {
            signal_hook::consts::SIGUSR1 => println!("SIGUSR1"),
            _ => println!("received a signal"),
        }
    }
    Ok(())
}
```

SIGKILLシグナル

- SIGKILLシグナルのシグナルハンドラを設定することはできず、デフォルトの挙動を変更できない
- したがって、SIGKILLを受信したプロセスは必ず停止する

kill, raise関数

- nix::sys::signal::kill：他のプロセスへシグナルを配送するための関数。C言語ではシステムコールとなる

```
pub fn kill<T: Into<Option<Signal>>>(pid: Pid, signal: T) -> Result<()>
```

- nix::sys::signal::raise：自分自身へシグナルを配送するための関数。C言語ではシステムコールではない

```
pub fn raise(signal: Signal) -> Result<()>
```

ミニ演習

- SIGUSR1シグナルを処理するプログラムを実装し、正しく動くことをkillコマンドで動作を確認せよ
- 自分自身にSIGTRAPを配送するプログラムを実装し、デバッガで動作させbreakとcontinueができることを確認せよ

ptrace

- プロセスのトレース、デバッグを行うための関数
- ステップ実行、メモリ読み書き、レジスタ読み書き、システムコールトレースなどを行うことが可能
- 詳細はman ptraceで確認すること

nix::sys::ptrace

- Rustでは、nix::sys::ptraceモジュールにptraceに関する関数が定義される
- 本講義では、これらを利用してデバッガを実装

デバッグ開始までの流れ

- forkする
  - traceme で子プロセスをトレース対象とする
  - 子プロセスのASLR（後述）をオフに
  - 子プロセスをexecする
- 親プロセスは、waitpidで子プロセスの停止を待つ
- 親プロセスは、子プロセスにブレークポイントを設定
- stepやcontで実行再開

自身をトレース対象に nix::sys::ptrace::traceme

- traceme関数を利用すると、自身をデバッガのトレース対象とする
- デバッガではfork後、tracemeしてから、execし対象の実行ファイルをトレース

```
// 子プロセスに渡すコマンドライン引数
let args: Vec<CString> = cmd.iter().map(|s| CString::new(*s).unwrap()).collect();

match unsafe { fork()? } {
    ForkResult::Child => {
        // ASLRを無効に
        let p = personality::get().unwrap();
        personality::set(p | Persona::ADDR_NO_RANDOMIZE).unwrap();
        ptrace::traceme().unwrap();

        // exec
        execvp(&CString::new(self.info.filename.as_str()).unwrap(), &args).unwrap();
        unreachable!();
    }
    ForkResult::Parent { child, .. } => match waitpid(child, None)? {
        WaitStatus::Stopped(..) => {
            println!("<<子プロセスの実行に成功しました：PID = {child}>>");
            self.info.pid = child;
            let mut dbg = ZDbg::<Running> {
                info: self.info,
                _state: Running,
            };
            dbg.set_break()?; // ブレークポイントを設定
            dbg.do_continue()
        }
        WaitStatus::Exited(..) | WaitStatus::Signaled(..) => {
            Err("子プロセスの実行に失敗しました".into())
        }
        _ => Err("子プロセスが不正な状態です".into()),
    },
}
```

Address Space Layout Randomization (ASLR)
アドレス空間配置のランダム化

- バッファオーバーランなどの脆弱性があると、リターンアドレスを書き換え可能になる
- リターンアドレスを書き換えて、特権的なシステムコールを呼び出すと、OSを乗っ取ることができる
- ASLRは、関数を配置するアドレスをランダムに決めて、システムコールなどの位置の特定を難しくする
- ただし、デバッグ時には不便なので、今回はコンパイル時にオフにする

ASLRの無効化

- .cargo/config.tomlに以下を追加

```
[build]
rustflags = ["-Crelocation-model=dynamic-no-pic"]
```

機械語レベルでステップ実行

nix::sys::ptrace::step

- stepをトレース中のプロセスを指定して呼び出すと、対象プロセスを機械語レベルでステップ実行

```
pub fn step<T: Into<Option<Signal>>>(pid: Pid, sig: T) -> Result<()>
```

実行再開
nix::sys::ptrace::cont

- contをトレース中のプロセスを指定して呼び出すと、対象プロセスが停止中の場合実行させる

  pub fn cont<T: Into<Option<Signal>>>(pid: Pid, sig: T) -> Result<()>

メモリ読み書き

nix::sys::ptrace::{read, write}

- 対象プロセスのメモリは、readとwriteで読み書き可能
- 64ビット単位の固定サイズでのみ読み書き可能
- writeはunsafe

```
pub fn read(pid: Pid, addr: AddressType) -> Result<c_long>
pub unsafe fn write(
    pid: Pid,
    addr: AddressType,
    data: *mut c_void
) -> Result<()>
```

writeに書き込むデータ

- writeに書き込むデータの型が、*mut c_voidとなっているため、asで型変換する必要あり

```
unsafe { ptrace::write(self.info.pid, addr, value as *mut c_void) }
```

レジスタ読み書き

nix::sys::ptrace::{getregs, setregs}

- getregsとsetregsで対象プロセスのレジスタを読み書き可能

```
pub fn getregs(pid: Pid) -> Result<user_regs_struct>
pub fn setregs(pid: Pid, regs: user_regs_struct) -> Result<()>
```

user_regs_struct構造体

- user_regs_struct構造体に、レジスタの値が格納される
- ripがプログラムカウンタで、今回はこれを操作

```
pub struct user_regs_struct {
    pub r15: ::c_ulonglong,
    pub r14: ::c_ulonglong,
    pub r13: ::c_ulonglong,
    pub r12: ::c_ulonglong,
    pub rbp: ::c_ulonglong,
    pub rbx: ::c_ulonglong,
    pub r11: ::c_ulonglong,
    pub r10: ::c_ulonglong,
    pub r9: ::c_ulonglong,
    pub r8: ::c_ulonglong,
    pub rax: ::c_ulonglong,
    pub rcx: ::c_ulonglong,
    pub rdx: ::c_ulonglong,
    pub rsi: ::c_ulonglong,
    pub rdi: ::c_ulonglong,
    pub orig_rax: ::c_ulonglong,
    pub rip: ::c_ulonglong,
    pub cs: ::c_ulonglong,
    pub eflags: ::c_ulonglong,
    pub rsp: ::c_ulonglong,
    pub ss: ::c_ulonglong,
    pub fs_base: ::c_ulonglong,
    pub gs_base: ::c_ulonglong,
    pub ds: ::c_ulonglong,
    pub es: ::c_ulonglong,
    pub fs: ::c_ulonglong,
    pub gs: ::c_ulonglong,
}
```

プログラムカウンタ
instruction pointer

デバッガの実装

ファイル

- Cargo.toml : Cargo用のファイル
- src/main.rs : main関数用のファイル
- src/dbg.rs : デバッガ実装用のファイル

今回実装するコマンド

- continue：実行を再開するコマンド
- stepi：機械語レベルで一命令実行するコマンド
- ブレークポイントを実際に設定する関数

型状態プログラミング

- ジェネリクスと特殊化を用いて、ある種の契約プログラミングを実現
- 契約プログラミングとは、ある処理を実行するさい、事前条件と事後条件を記述し遵守させること

デバッガ用の型

- ZDbgがデバッガ用の型
  - ZDbg<Running>が子プロセス実行中のデバッガ用の型
  - ZDbg<NotRunning>が子プロセスを実行していないときのデバッガの型
- RunningとNotRunning型でデバッガの状態を表現

```
/// デバッガ内の情報
pub struct DbgInfo {
    pid: Pid,
    brk_addr: Option<*mut c_void>,
    brk_val: i64,
    filename: String,
}

/// デバッガ
/// ZDbg<Running>は子プロセスを実行中
/// ZDbg<NotRunning>は子プロセスは実行していない
pub struct ZDbg<T> {
    info: Box<DbgInfo>,
    _state: T,
}

/// デバッガの実装
pub struct Running; // 実行中
pub struct NotRunning; // 実行していない
```

デバッガの状態を表す型

- State型でデバッガの状態を表現
- 返り値として利用

```
/// デバッガの実装の列挙型表現。Exitの場合終了
pub enum State {
    Running(ZDbg<Running>),
    NotRunning(ZDbg<NotRunning>),
    Exit,
}
```

共通メソッド

- 子プロセス実行中と非実行中で共通のメソッドは、ZDbg<T>に定義

```
/// RunningとNotRunningで共通の実装
impl<T> ZDbg<T> {
    /// ブレークポイントのアドレスを設定する関数。子プロセスのメモリ上には反映しない。
    /// アドレス設定に成功した場合はtrueを返す
    fn set_break_addr(&mut self, cmd: &[&str]) -> bool { ... }

    /// 共通のコマンドを実行
    fn do_cmd_common(&self, cmd: &[&str]) { ... }
}
```

子プロセス非実行中に呼び出し可能なメソッド

- 以下の関数は子プロセス非実行中にのみ呼び出し可能

- つまり、これら関数を実行する際には、デバッガの状態はNotRunningでなければならないという契約を表記

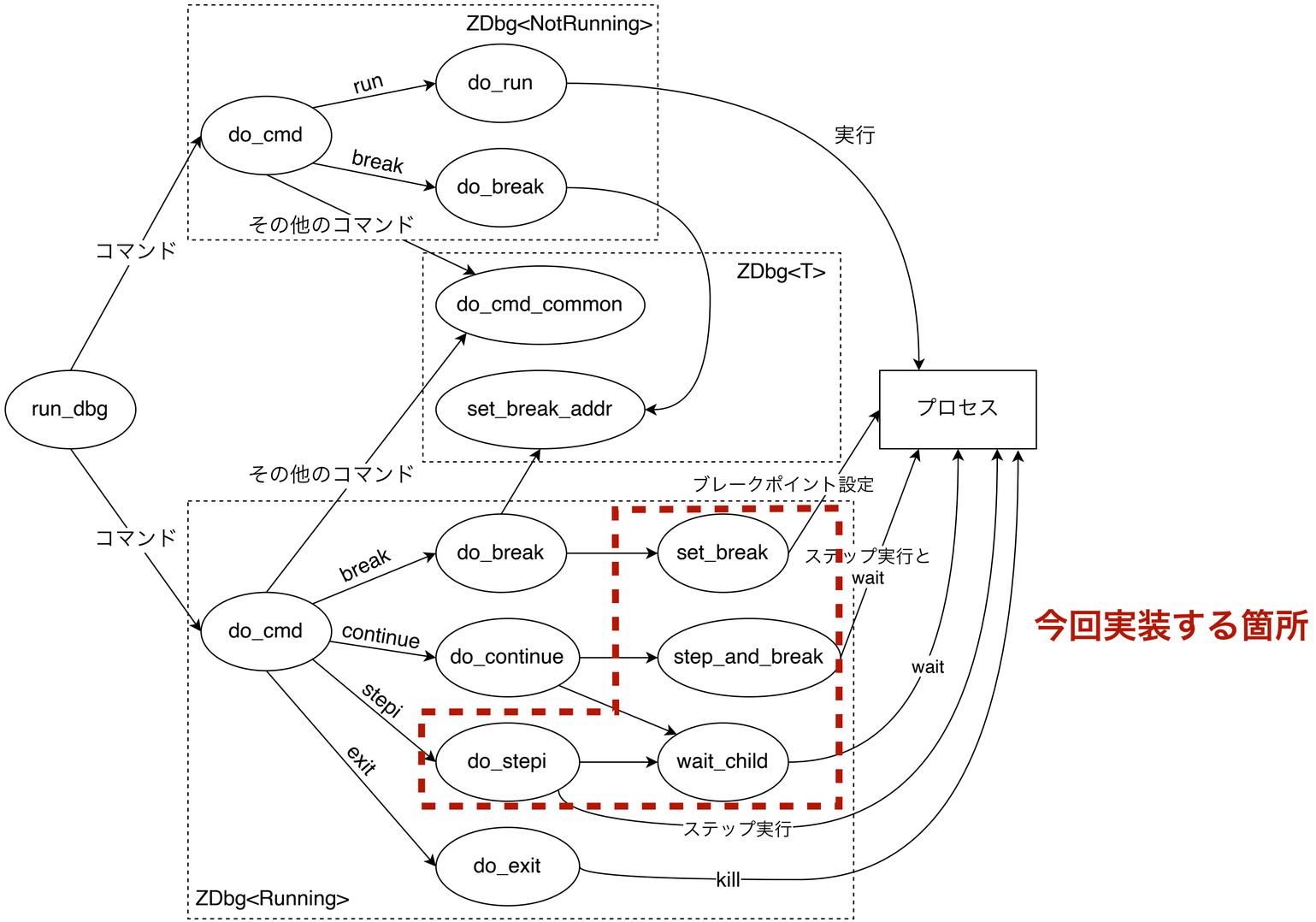
```
/// NotRunning時に呼び出し可能なメソッド
impl ZDbg<NotRunning> {
    pub fn new(filename: String) -> Self { ... }

    /// ブレークポイントを設定
    fn do_break(&mut self, cmd: &[&str]) -> bool { ... }

    /// 子プロセスを生成し、成功した場合はRunning状態に遷移
    fn do_run(mut self, cmd: &[&str]) -> Result<State, Box<dyn Error>> { ... }

    pub fn do_cmd(mut self, cmd: &[&str]) -> Result<State, Box<dyn Error>> { ... }
}
```

子プロセス非実行中に呼び出し可能なメソッド

- 以下の関数は子プロセス実行中にのみ呼び出し可能
- これら関数を実行する際には、デバッガの状態はRunningでなければならない

```rust
/// Running時に呼び出し可能なメソッド
impl ZDbg<Running> {
    pub fn do_cmd(mut self, cmd: &[&str]) -> Result<State, Box<dyn Error>> { ... }

    /// exitを実行。実行中のプロセスはkill
    fn do_exit(self) -> Result<(), Box<dyn Error>> { ... }

    /// ブレークポイントを実際に設定
    /// つまり、該当アドレスのメモリを"int 3" = 0xccに設定
    fn set_break(&mut self) -> Result<(), Box<dyn Error>> { ... }

    /// breakを実行
    fn do_break(&mut self, cmd: &[&str]) -> Result<(), Box<dyn Error>> { ... }

    /// stepiを実行。機械語レベルで1行実行
    fn do_stepi(self) -> Result<State, Box<dyn Error>> { ... }

    /// ブレークポイントで停止していた場合は
    /// 1ステップ実行しブレークポイントを再設定
    fn step_and_break(mut self) -> Result<State, Box<dyn Error>> { ... }

    /// continueを実行
    fn do_continue(self) -> Result<State, Box<dyn Error>> { ... }

    /// 子プロセスをwait。子プロセスが終了した場合はNotRunning状態に遷移
    fn wait_child(self) -> Result<State, Box<dyn Error>> { ... }
}
```

ZDbgの処理の流れ



今回実装する箇所

テスト用コード

- デバッグのテスト用に、
  dbg_targetというコードを利用
- cargo buildでビルドしておく

```
use std::arch::asm;

use nix::{
    sys::signal::{kill, Signal},
    unistd::getpid,
};

fn main() {
    println!("int 3");
    unsafe { asm!("int 3") };

    println!("kill -SIGTRAP");
    let pid = getpid();
    kill(pid, Signal::SIGTRAP).unwrap();

    for i in 0..3 {
        unsafe { asm!("nop") };
        println!("i = {i}");
    }
}
```

逆アセンブリ

```
$ objdump -d target/debug/dbg_target|less
0000000000007da0 <_ZN10dbg_target4main17h7af2db262e1d67b3E>:
    7da0:       48 81 ec f8 00 00 00    sub    $0xf8,%rsp
    7da7:       48 8d 7c 24 28          lea    0x28(%rsp),%rdi
    7dac:       48 8d 35 8d 63 04 00    lea    0x4638d(%rip),%rsi        # 4e140 <__do_global_dtors_aux_fini_array_entry+0x80>
    7db3:       ba 01 00 00 00          mov    $0x1,%edx
    7db8:       48 8d 0d d1 72 03 00    lea    0x372d1(%rip),%rcx        # 3f090 <_fini+0xe98>
    7dbf:       31 c0                   xor    %eax,%eax
    7dc1:       41 89 c0                mov    %eax,%r8d
    7dc4:       e8 57 fa ff ff          call   7820 <_ZN4core3fmt9Arguments6new_v117h86d47485b5f6aefbE>
    7dc9:       48 8d 7c 24 28          lea    0x28(%rsp),%rdi
    7dce:       ff 15 14 92 04 00       call   *0x49214(%rip)        # 50fe8 <_GLOBAL_OFFSET_TABLE_+0x738>
    7dd4:       cc                      int3
    7dd5:       48 8d 7c 24 58          lea    0x58(%rsp),%rdi
    7dda:       48 8d 35 6f 63 04 00    lea    0x4636f(%rip),%rsi        # 4e150 <__do_global_dtors_aux_fini_array_entry+0x90>
    7de1:       ba 01 00 00 00          mov    $0x1,%edx
    7de6:       48 8d 0d a3 72 03 00    lea    0x372a3(%rip),%rcx        # 3f090 <_fini+0xe98>
    7ded:       31 c0                   xor    %eax,%eax
    7def:       41 89 c0                mov    %eax,%r8d
    7df2:       e8 29 fa ff ff          call   7820 <_ZN4core3fmt9Arguments6new_v117h86d47485b5f6aefbE>
    7df7:       48 8d 7c 24 58          lea    0x58(%rsp),%rdi
    7dfc:       ff 15 e6 91 04 00       call   *0x491e6(%rip)        # 50fe8 <_GLOBAL_OFFSET_TABLE_+0x738>
    7e02:       e8 29 fc ff ff          call   7a30 <_ZN3nix6unistd6getpid17h03517788fd7a4bbcE>
    7e07:       89 44 24 24             mov    %eax,0x24(%rsp)
    7e0b:       89 84 24 f4 00 00 00    mov    %eax,0xf4(%rsp)
    7e12:       8b 7c 24 24             mov    0x24(%rsp),%edi
    7e16:       c7 84 24 8c 00 00 00    movl   $0x5,0x8c(%rsp)
    7e1d:       05 00 00 00
    7e21:       8b b4 24 8c 00 00 00    mov    0x8c(%rsp),%esi
    7e28:       e8 f3 fa ff ff          call   7920 <_ZN3nix3sys6signal4kill17h3e7eb33ee0cc7379E>
    7e2d:       89 44 24 20             mov    %eax,0x20(%rsp)
    7e31:       8b 7c 24 20             mov    0x20(%rsp),%edi
```

int 3の箇所

nix::unistd::getpidの箇所

nix::sys::signal::killの箇所

逆アセンブリの続き

```
7ead:       74 06                   je     7eb5 <_ZN10dbg_target4main17h7af2db262e1d67b3E+0x115>
7eaf:       eb 00                   jmp    7eb1 <_ZN10dbg_target4main17h7af2db262e1d67b3E+0x111>
7eb1:       eb 0a                   jmp    7ebd <_ZN10dbg_target4main17h7af2db262e1d67b3E+0x11d>
7eb3:       0f 0b                   ud2
7eb5:       48 81 c4 f8 00 00 00    add    $0xf8,%rsp
7ebc:       c3                      ret
7ebd:       8b 84 24 a4 00 00 00    mov    0xa4(%rsp),%eax
7ec4:       89 84 24 ac 00 00 00    mov    %eax,0xac(%rsp)
7ecb:       90                      nop
7ecc:       48 8d bc 24 ac 00 00    lea    0xac(%rsp),%rdi
7ed3:       00
7ed4:       ff 15 c6 8f 04 00       call   *0x48fc6(%rip)        # 50ea0 <_GLOBAL_OFFSET_TABLE_+0x5f0>
7eda:       48 89 44 24 08          mov    %rax,0x8(%rsp)
7edf:       48 89 54 24 10          mov    %rdx,0x10(%rsp)
7ee4:       48 8b 44 24 10          mov    0x10(%rsp),%rax
```

今回はnop命令の次のアドレスにブレークポイントを設定
0x7eccというアドレスを覚えておく
（環境によって異なる可能性があるので、必ずobjdumpで確認すること）

ブレークポイントの設定方法

1. アドレスの先のデータをptrace::readで取得

2. 該当箇所のみを0xcc (int 3)に変更し、ptrace::writeで書き込み

```
7ebd:       8b 84 24 a4 00 00 00    mov    0xa4(%rsp),%eax
7ec4:       89 84 24 ac 00 00 00    mov    %eax,0xac(%rsp)
7ecb:       90                      nop
7ecc:       48 8d bc 24 ac 00 00    lea    0xac(%rsp),%rdi
7ed3:       00
7ed4:       ff 15 c6 8f 04 00       call   *0x48fc6(%rip)
```

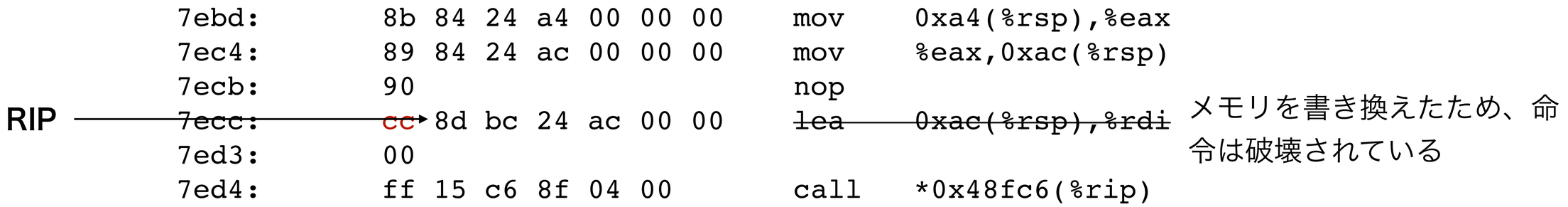
ptrace::{read, write}は8バイト単位で読み書きすることに注意
上の例だと、0x48というデータのみを0xccに変更する

ブレークポイントヒット後の実行

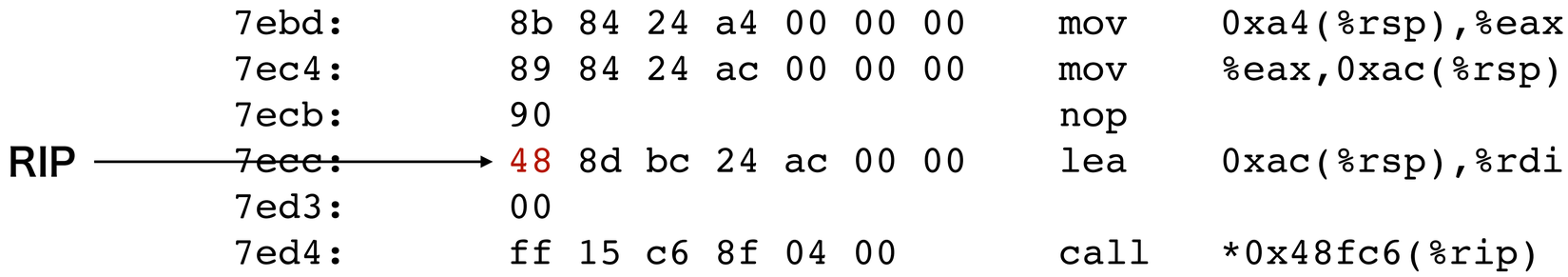
- stepi : アセンブリレベルレベルでのステップ実行
  - 現在のプログラムカウンタ（RIP）が、ブレークポイントのアドレス + 1か？
    - YES :
      1. プログラムカウンタを1減らす
      2. 0xccに書き換えたメモリを元に戻す
      3. ptrace::stepでステップ実行後wait::waitpidで子プロセスの停止を待機
      4. その後に再度、ブレークポイント位置のメモリを0xccに変更
    - NO : ptrace::stepでステップ実行
- continue : 実行再開
  - 現在のプログラムカウンタ（RIP）が、ブレークポイントのアドレス + 1か？
    - YES :
      1. プログラムカウンタを1減らす
      2. 0xccに書き換えたメモリを元に戻す
      3. ptrace::stepでステップ実行後wait::waitpidで子プロセスの停止を待機
      4. その後に再度、ブレークポイント位置のメモリを0xccに変更
      5. 最後にptrace::contで実行再開
    - NO : ptrace::contで実行再開

図解：ブレークポイントヒット後の実行（1/2）

1. ブレークポイントヒット後は、RIPが0x7ecdで、メモリ上のデータが0xccになっている



2. RIPを1へらし、0xccに書き換えたデータを、元の値に戻す



図解：ブレークポイントヒット後の実行（2/2）

3. ステップ実行

```
        7ebd:    8b 84 24 a4 00 00 00    mov    0xa4(%rsp),%eax
        7ec4:    89 84 24 ac 00 00 00    mov    %eax,0xac(%rsp)
        7ecb:    90                      nop
        7ecc:    48 8d bc 24 ac 00 00    lea    0xac(%rsp),%rdi
        7ed3:    00
RIP ——> 7ed4:    ff 15 c6 8f 04 00       call   *0x48fc6(%rip)
```

4. ブレークポイントを再び設定

```
        7ebd:    8b 84 24 a4 00 00 00    mov    0xa4(%rsp),%eax
        7ec4:    89 84 24 ac 00 00 00    mov    %eax,0xac(%rsp)
        7ecb:    90                      nop
        7ecc:    cc 8d bc 24 ac 00 00    lea    0xac(%rsp),%rdi
        7ed3:    00
RIP ——> 7ed4:    ff 15 c6 8f 04 00       call   *0x48fc6(%rip)
```

ブレークするアドレスの注意

- Rustはデフォルトでは、位置独立形式のバイナリを出力する
- 位置独立形式のバイナリは、メモリ上の任意の位置に配置可能
- 位置独立形式の場合の配置方法
  - 0x555555554000が、バイナリ中のアドレスに加算される
  - ALSRが有効な場合、さらにランダム値が加算される
- 位置独立を無効にする場合、.cargo/config.tomlに以下を追加

```
[build]
rustflags = ["-Crelocation-model=dynamic-no-pic"]
```

  - https://github.com/ytakano-lecture/dbg_target.git

zdbgの実行例

```
$ cargo run ../dbg_target/target/debug/dbg_target
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/zdbg ../dbg_target/target/debug/dbg_target`
zdbg > b 0x40393c
zdbg > r
<<子プロセスの実行に成功しました：PID = 16382>>

<<以下のようにメモリを書き換えます>>

<<before: 40393c: 48 8d bc 24 ac 0 0 0>>
<<after : 40393c: cc 8d bc 24 ac 0 0 0>>
int 3
<<子プロセスが停止しました：PC = 0x403845>>

zdbg > c
kill -SIGTRAP
<<子プロセスが停止しました：PC = 0x7ffff7da475b>>

zdbg > c
<<子プロセスが停止しました：PC = 0x40393c>>

zdbg > c
<<以下のようにメモリを書き換えます>>

<<before: 40393c: 48 8d bc 24 ac 0 0 0>>
<<after : 40393c: cc 8d bc 24 ac 0 0 0>>
i = 0
<<子プロセスが停止しました：PC = 0x40393c>>

zdbg > c
<<以下のようにメモリを書き換えます>>

<<before: 40393c: 48 8d bc 24 ac 0 0 0>>
<<after : 40393c: cc 8d bc 24 ac 0 0 0>>
i = 1
<<子プロセスが停止しました：PC = 0x40393c>>

zdbg > c
<<以下のようにメモリを書き換えます>>

<<before: 40393c: 48 8d bc 24 ac 0 0 0>>
<<after : 40393c: cc 8d bc 24 ac 0 0 0>>
i = 2
<<子プロセスが終了しました>>

zdbg >
```

今回実装する箇所

- set_break : ptrace::{read, write}で該当位置のメモリを0xccに変更
- wait_child : プログラムカウンタを1減らすのと、0xccを元に戻すのは共通操作のため、ここで行う
- step_and_break : ブレークポイントで停止していた場合、1ステップ機械語レベルで実行し、再度ブレークポイントを設定
- do_stepi : 次の実行アドレスがブレークポイントの場合、0xccを元に戻してからステップ実行し、再度0xccに設定。そうでない場合は、ptrace::stepとwait_child

演習（1日目）

1-1. ミニ演習を実施しレポートとしてまとめよ

1-2. ZDbgの以下の関数を実装し、レポートとしてまとめよ

1-2 (a). set_break

1-2 (b). wait_child

1-2 (c). step_and_break

学部生必須課題

1-2 (d). do_stepi

大学院生＆社会人必須課題

1-3. 挑戦課題

1-3 (a). 複数のブレークポイントを設定可能にせよ

1-3 (b). ブレークポイントの削除にも対応せよ

1日目 発表スライド

- 進捗、質問、感想などを書いてください

DWARF

DWARF

- デバッグ情報のファイルフォーマット
- 実行ファイルのフォーマットが、Executable and Linkable Format（ELF）であるから、ELFに対して、DWARFと名づけられた。つまりダジャレ。何かの略語というわけではない。

ELF
実行ファイルフォーマット

DWARF
デバッグ情報ファイルフォーマット

Debugging Information Entry（DIE）

- DWARF中で中心をなすデバッグ情報
- 木構造でデータが保存されており、TagとAttributeでノードが分類される
  - Tag：データのタイプを表す。C言語で言うところの構造体のようなもの。DWARF中ではDW_TAG_という接頭辞がつく
  - Attribute：Tag中に含まれる値を表す。C言語で言うところの構造体メンバ変数のようなもの。DWARF中ではDW_ATという接頭辞がつく

objdump

- ELFファイルの中身を表示するコマンド
- 引数に-Wを指定するとDWARF情報がすべて表示される

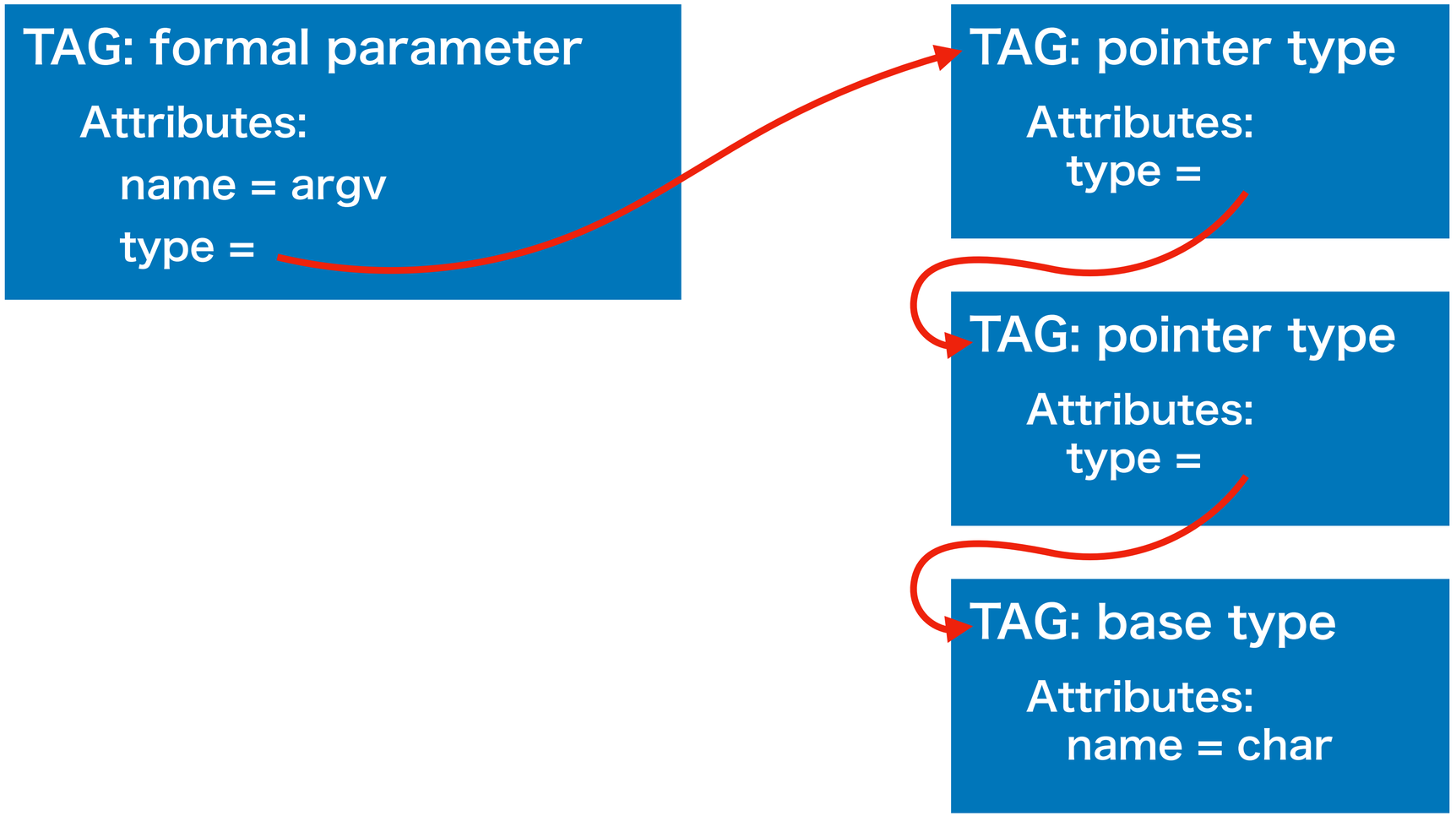
```
$ objdump -W a.out
```

DWARF内のsection

- .debug_abbrev：DIEの構造を定義する箇所
- .debug_info：DIE情報
- .debug_line：行番号に関する情報

など

図解DIE


TAG: formal parameter
Attributes:
name = argv
type =
TAG: pointer type
Attributes:
type =
TAG: pointer type
Attributes:
type =
TAG: base type
Attributes:
name = char

.debug_info中のsubprogramタグの例

```
.debug_info

<1><2a>: Abbrev Number: 2 (DW_TAG_subprogram)
   <2b>   DW_AT_low_pc       : 0x401130
   <33>   DW_AT_high_pc      : 0x48
   <37>   DW_AT_frame_base   : 1 byte block: 56          (DW_OP_reg6 (rbp))
   <39>   DW_AT_name         : (indirect string, offset: 0x44): main
   <3d>   DW_AT_decl_file    : 1
   <3e>   DW_AT_decl_line    : 3
   <3f>   DW_AT_prototyped   : 1
   <3f>   DW_AT_type         : <0x60>
   <43>   DW_AT_external     : 1
```

関数の開始アドレス

関数の終了アドレス

ファイル名を指す値（後述）

関数名

行番号

.debug_abbrevと.debug_info sectionの対応例

```
.debug_abbrev   2       DW_TAG_subprogram    [has children]
                DW_AT_low_pc         DW_FORM_addr
                DW_AT_high_pc        DW_FORM_data4
                DW_AT_frame_base     DW_FORM_exprloc
                DW_AT_name           DW_FORM_strp
                DW_AT_decl_file      DW_FORM_data1
                DW_AT_decl_line      DW_FORM_data1
                DW_AT_prototyped     DW_FORM_flag_present
                DW_AT_type           DW_FORM_ref4
                DW_AT_external       DW_FORM_flag_present
                DW_AT value: 0       DW_FORM value: 0
```

subprogramというabbreviationを定義

```
.debug_info     <1><2a>: Abbrev Number: 2 (DW_TAG_subprogram)
                   <2b>   DW_AT_low_pc        : 0x401130
                   <33>   DW_AT_high_pc       : 0x48
                   <37>   DW_AT_frame_base    : 1 byte block: 56         (DW_OP_reg6 (rbp))
                   <39>   DW_AT_name          : (indirect string, offset: 0x44): main
                   <3d>   DW_AT_decl_file     : 1
                   <3e>   DW_AT_decl_line     : 3
                   <3f>   DW_AT_prototyped    : 1
                   <3f>   DW_AT_type          : <0x60>
                   <43>   DW_AT_external      : 1
```

実際のデータ

subprogram中のdecl_fileアトリビュートと.debug_line中のFile Name Tableの関係

.debug_info

```
<1><2a>: Abbrev Number: 2 (DW_TAG_subprogram)
   <2b>   DW_AT_low_pc       : 0x401130
   <33>   DW_AT_high_pc      : 0x48
   <37>   DW_AT_frame_base   : 1 byte block: 56         (DW_OP_reg6 (rbp))
   <39>   DW_AT_name         : (indirect string, offset: 0x44): main
   <3d>   DW_AT_decl_file    : 1
   <3e>   DW_AT_decl_line    : 3
   <3f>   DW_AT_prototyped   : 1
   <3f>   DW_AT_type         : <0x60>
   <43>   DW_AT_external     : 1
```

.debug_line

```
The File Name Table (offset 0x1d):
 Entry Dir     Time    Size    Name
 1     0       0       0       hello.c
```

.debug_info中の変数と型情報

```
.debug_info
 <2><51>: Abbrev Number: 3 (DW_TAG_formal_parameter)
    <52>   DW_AT_location    : 2 byte block: 91 70       (DW_OP_fbreg: -16)
    <55>   DW_AT_name        : (indirect string, offset: 0x52): argv
    <59>   DW_AT_decl_file   : 1
    <5a>   DW_AT_decl_line   : 3
    <5b>   DW_AT_type        : <0x67>
 <2><5f>: Abbrev Number: 0
中略
 <1><67>: Abbrev Number: 5 (DW_TAG_pointer_type)
    <68>   DW_AT_type        : <0x6c>
 <1><6c>: Abbrev Number: 5 (DW_TAG_pointer_type)
    <6d>   DW_AT_type        : <0x71>
 <1><71>: Abbrev Number: 4 (DW_TAG_base_type)
    <72>   DW_AT_name        : (indirect string, offset: 0x57): char
    <76>   DW_AT_encoding    : 6        (signed char)
    <77>   DW_AT_byte_size   : 1
```

変数名

変数名

DWARF中の型情報

ソースコード中の型名

ソースコード中の行番号とメモリアドレスとの対応

```
$ objdump --dwarf=decodedline hello

hello:     file format elf64-x86-64

Contents of the .debug_line section:

CU: hello.c:
File name                            Line number    Starting address    View    Stmt
hello.c                                      3           0x401130                x
hello.c                                      4           0x401146                x
hello.c                                      5           0x401157                x
hello.c                                      6           0x401170                x
hello.c                                      6           0x401178                x
```

libelfin

- ELFファイル中に含まれるデバッグ情報を取得することのできるライブラリ
- DWARFファイルの構造を理解してパースするのは大変なので、今回はlibelfinを利用

DW_TAG_subprogram中の DW_AT_nameを表示する例

```
int fd = open(argv[1], O_RDONLY);
if (fd < 0) {
    fprintf(stderr, "%s: %s\n", argv[1], strerror(errno));
    return 1;
}

elf::elf ef(elf::create_mmap_loader(fd));
dwarf::dwarf dw(dwarf::elf::create_loader(ef));

for (const auto &cu : dw.compilation_units()) {
    for (const auto &die : cu.root()) {
        if (die.tag == dwarf::DW_TAG::subprogram) {
            if (die.has(dwarf::DW_AT::name)) {
                std::cout << "name = " << dwarf::at_name(die) << std::endl;
            }
        }
    }
}
```

コンパイル単位でデータが保存される

TAGがsubprogramなら

nameというattributeを持っているなら

at_nameというattributeを取得

デバッグレジスタ

- 今回は、割り込みを使ってbreakを実現した
- CPUが備えるデバッグレジスタを利用して、breakを実現することも可能
- 停止したいアドレスをデバッグレジスタに設定しておくとプログラムが停止