

SyntacoreTM
Custom cores and tools

ОТЛАДКА С ПОМОЩЬЮ АППАРАТНЫХ ТОЧЕК ОСТАНОВА И ИХ ПОДДЕРЖКА В ОТЛАДЧИКАХ

Март 22, 2025

Юлий Тарасов

yuly.tarasov@syntacore.com

О себе

Типичные сценарии отладки и их ограничения

Аппаратные точки останова

Реализация аппаратных точек останова в отладчиках

Выводы



Юлий Тарасов

Старший разработчик

- Автор прототипа поддержки аппаратных точек останова в GDB для RISC-V
- Тулчейн и средства разработки
- Инструменты верификации аппаратуры

О себе

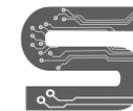
Типичные подходы к отладке и их ограничения

Аппаратные точки останова

Реализация аппаратных точек останова в отладчиках

Выводы

Нашли баг, пытаемся отладить



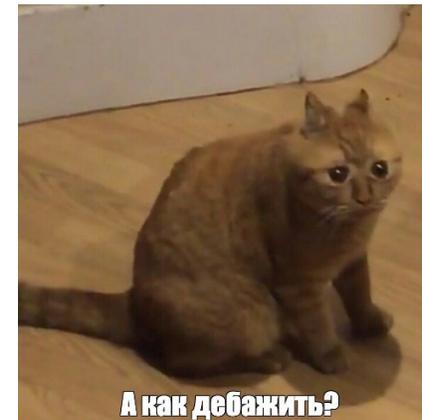
- Данная сортировка не сортирует:

- Ввод:
 - Nick: 5
 - John: 13
 - Betty: 8

- Вывод:

```
^ ~/demo
> ./buggy-sort
Nick: 5
Nick: 5
Betty: 8
^ ~/demo
> ./buggy-sort
Nick: 5
(null): -1064651776
Betty: 8
*** stack smashing detected ***: terminated
zsh: IOT instruction (core dumped) ./buggy-sort
```

- Как будем дебажить этот код?



```
typedef struct {
    char *data;
    int key;
} item;
```

```
void sort(item *a, int n) {
    int s = 1;

    for (int i = 0; i < n && s != 0; i++) {
        s = 0;
        for (int j = 0; j < n; j++)
            if (a[j].key > a[j + 1].key) {
                item t = a[i];
                a[j] = a[j + 1];
                a[j + 1] = t;
                s++;
            }
        n--;
    }
}
```



Дебажимся: запуск GDB

```
^  ~ /demo  at 21:06:26
> gdb ./buggy-sort
GNU gdb (GDB) 16.1
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./buggy-sort...
(gdb) tui enable
```

Дебажимся: запуск GDB



buggy-sort.c

```
29 int main() {
30     item items[] = {{.data = "Nick", .key = 5},
31                     {.data = "John", .key = 13},
32                     {.data = "Betty", .key = 8}};
33     int n_items = sizeof(items) / sizeof (item);
34     sort(items, n_items);
35     print(items, n_items);
36     return 0;
37 }
```

exec No process (src) In:

L?? PC: ??

(gdb) █



Дебажимся: ставим breakpoint

buggy-sort.c

```
29 int main() {
30     item items[] = {{.data = "Nick", .key = 5},
31                     {.data = "John", .key = 13},
32                     {.data = "Betty", .key = 8}};
33     int n_items = sizeof(items) / sizeof (item);
34     sort(items, n_items);
35     print(items, n_items);
36     return 0;
37 }
```

exec No process (src) In:

L?? PC: ??

(gdb) break sort

Breakpoint 1 at 0x1154: file buggy-sort.c, line 10.

(gdb) █



Дебажимся: остановка на breakpoint

```
buggy-sort.c
6   int key;
7 } item;
8
9 void sort(item *a, int n) {
B+> 10   int s = 1;
11   for (int i = 0; i < n && s != 0; i++) {
12       s = 0;
13       for (int j = 0; j < n; j++)
14           if (a[j].key > a[j + 1].key) {
15               item t = a[i];

```

multi-thre Thread 0x7ffff7d887 (src) In: sort L10 PC: 0x55555555154
(gdb) run
Starting program: /home/julius/demo/buggy-sort
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, sort (a=0x7ffffffffffd680, n=3) at buggy-sort.c:10
(gdb) █

Дебажимся: остановка на breakpoint

```
buggy-sort.c
6   int key;
7   } item;
8
9   void sort(item *a, int n) {
B+> 10  int s = 1;
11   for (int i = 0; i < n && s != 0; i++) {
12       s = 0;
13       for (int j = 0; j < n; j++)
14           if (a[j].key > a[j + 1].key) {
15               item t = a[i];
```



```
multi-thre Thread 0x7ffff7d887 (src) In: sort          L10    PC: 0x55555555154
```

```
(gdb) run
```

```
Starting program: /home/julius/demo/buggy-sort
```

```
[Thread debugging using libthread_db enabled]
```

```
Using host libthread_db library "/usr/lib/libthread_db.so.1".
```

```
Breakpoint 1, sort (a=0x7ffffffffffd680, n=3) at buggy-sort.c:10
```

```
(gdb) █
```



Breakpoint под капотом

- Для реализации точек останова используются специальные инструкции-ловушки (traps, **int 3** в x86, **ebreak** в RISC-V)

```
int scalar_mul(int x1, int y1,  
               int x2, int y2) {  
    int z1 = x1 * y1;  
    int z2 = x2 & y2; ← looks like bug  
    return z1 + z2;  
}
```

ORIGINAL

```
scalar_mul:  
    mulw    a0,a0,a1  
    and     a2,a2,a3  
    addw    a0,a0,a2  
    ret
```



Breakpoint под капотом

- Для реализации точек останова используются специальные инструкции-ловушки (traps)
- Отладчик патчит исполняемую программу (образ памяти), заменяя некоторые инструкции ловушками

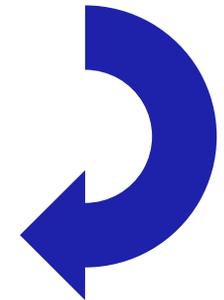
```
int scalar_mul(int x1, int y1,  
               int x2, int y2) {  
    int z1 = x1 * y1;  
    int z2 = x2 & y2; ← looks like bug  
    return z1 + z2;  
}
```

ORIGINAL

```
scalar_mul:  
    mulw    a0,a0,a1  
    and     a2,a2,a3  
    addw    a0,a0,a2  
    ret
```

PATCHED

```
scalar_mul:  
    mulw    a0,a0,a1  
    ebreak  
    addw    a0,a0,a2  
    ret
```





Breakpoint под капотом

- Для реализации точек останова используются специальные инструкции-ловушки (traps)
- Отладчик патчит исполняемую программу (образ памяти), заменяя некоторые инструкции ловушками
- Попадая на такую инструкцию, процессор передает управление ОС

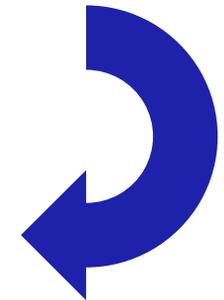
```
int scalar_mul(int x1, int y1,  
              int x2, int y2) {  
    int z1 = x1 * y1;  
    int z2 = x2 & y2; ← looks like bug  
    return z1 + z2;  
}
```

ORIGINAL

```
scalar_mul:  
    mulw    a0,a0,a1  
    and     a2,a2,a3  
    addw    a0,a0,a2  
    ret
```

PATCHED

```
scalar_mul:  
    mulw    a0,a0,a1  
    ebreak  ← pc  
    addw    a0,a0,a2  
    ret
```





Breakpoint под капотом

- Для реализации точек останова используются специальные инструкции-ловушки (traps)
- Отладчик патчит исполняемую программу (образ памяти), заменяя некоторые инструкции ловушками
- Попадая на такую инструкцию, процессор передает управление ОС
- ОС останавливает отлаживаемый процесс

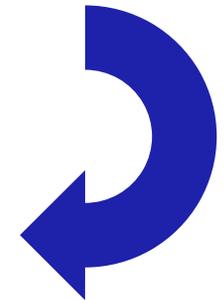
```
int scalar_mul(int x1, int y1,  
              int x2, int y2) {  
    int z1 = x1 * y1;  
    int z2 = x2 & y2; ← looks like bug  
    return z1 + z2;  
}
```

ORIGINAL

```
scalar_mul:  
    mulw    a0,a0,a1  
    and     a2,a2,a3  
    addw    a0,a0,a2  
    ret
```

PATCHED

```
scalar_mul:  
    mulw    a0,a0,a1  
    ebreak  ← pc  
    addw    a0,a0,a2  
    ret
```





Дебажимся: continue

```
buggy-sort.c
11  for (int i = 0; i < n && s != 0; i++) {
12      s = 0;
13      for (int j = 0; j < n; j++)
14          if (a[j].key > a[j + 1].key) {
B+> 15      item t = a[i];
16          a[j] = a[j + 1];
17          a[j + 1] = t;
18          s++;
19      }
20      n--;

multi-thre Thread 0x7ffff7d887 (src) In: sort      L15  PC: 0x555555551b2
(gdb) break 15
Breakpoint 2 at 0x555555551b2: file buggy-sort.c, line 15.
(gdb) continue
Continuing.

Breakpoint 2, sort (a=0x7fffffff680, n=3) at buggy-sort.c:15
(gdb) █
```



Continue после остановки

- Для продолжения исполнения:
 - Ставится временный breakpoint на следующую инструкцию

```
ORIGINAL
scalar_mul:
    mulw    a0, a0, a1
    and     a2, a2, a3
    addw    a0, a0, a2
    ret
```



```
PATCHED
scalar_mul:
    mulw    a0, a0, a1
    ebreak
    addw    a0, a0, a2
    ret
```

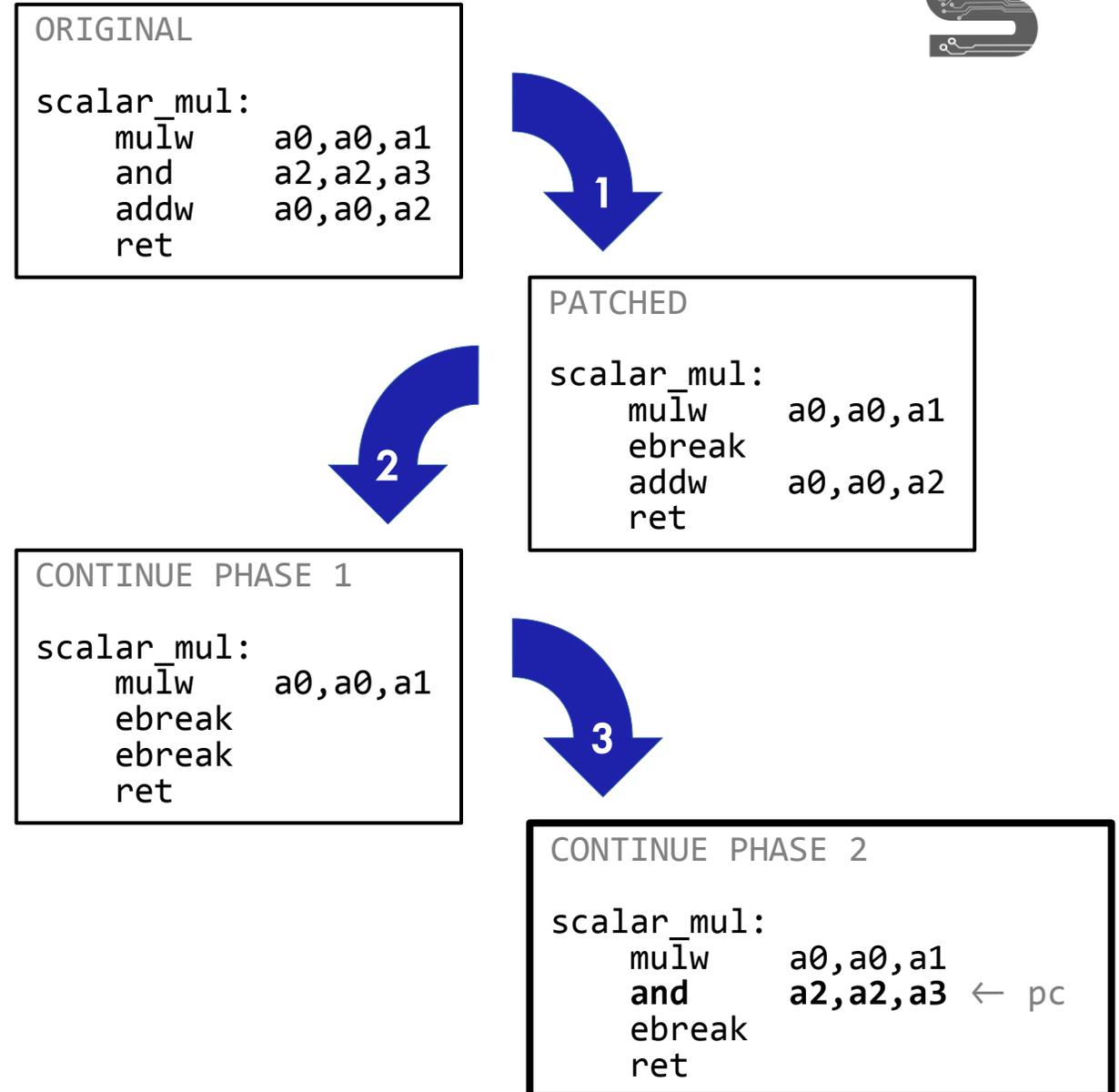


```
CONTINUE PHASE 1
scalar_mul:
    mulw    a0, a0, a1
    ebreak ← pc
    ebreak
    ret
```



Continue после остановки

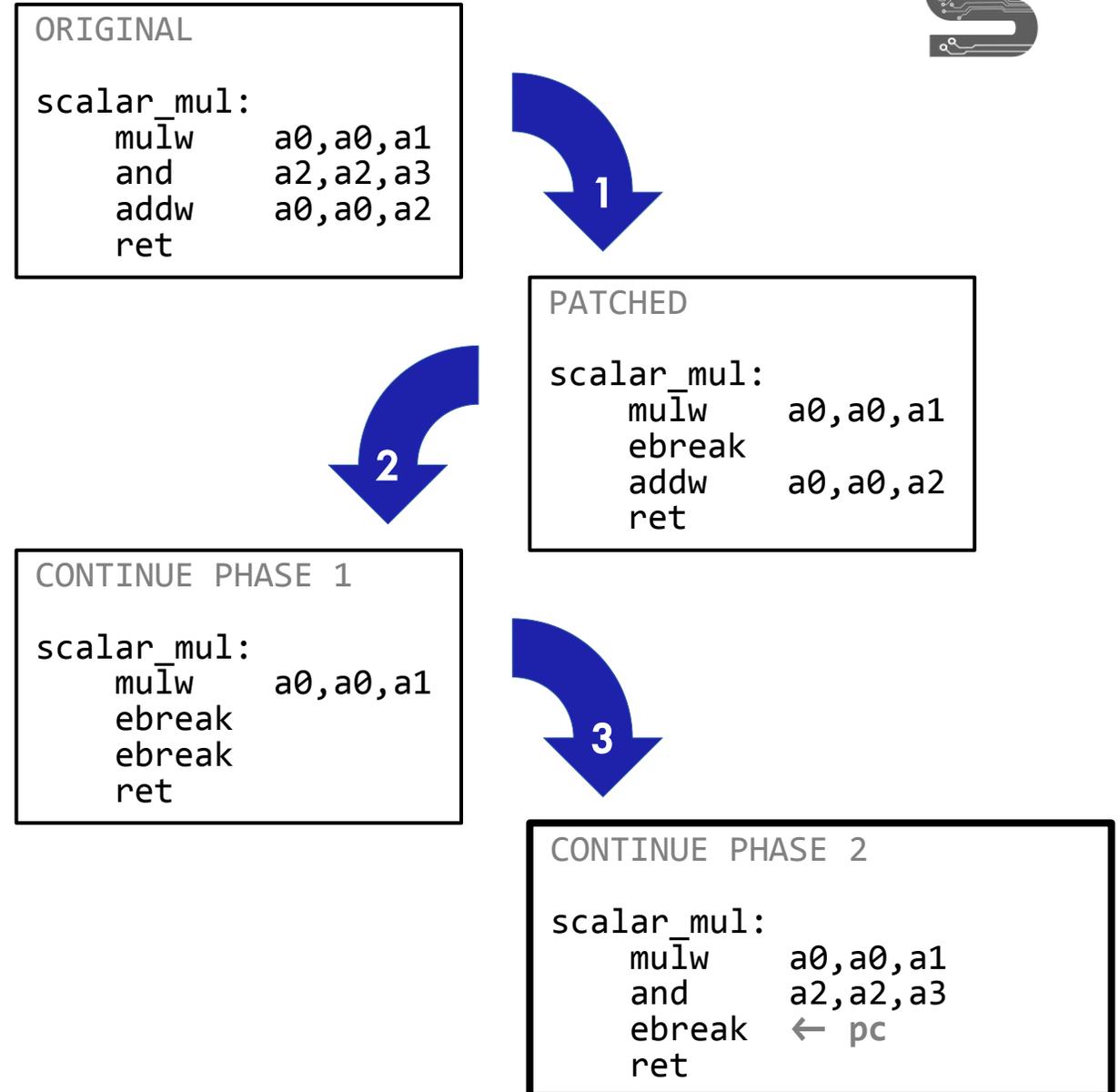
- Для продолжения исполнения:
 - Ставится временный breakpoint на следующую инструкцию
 - Восстанавливается текущая инструкция





Continue после остановки

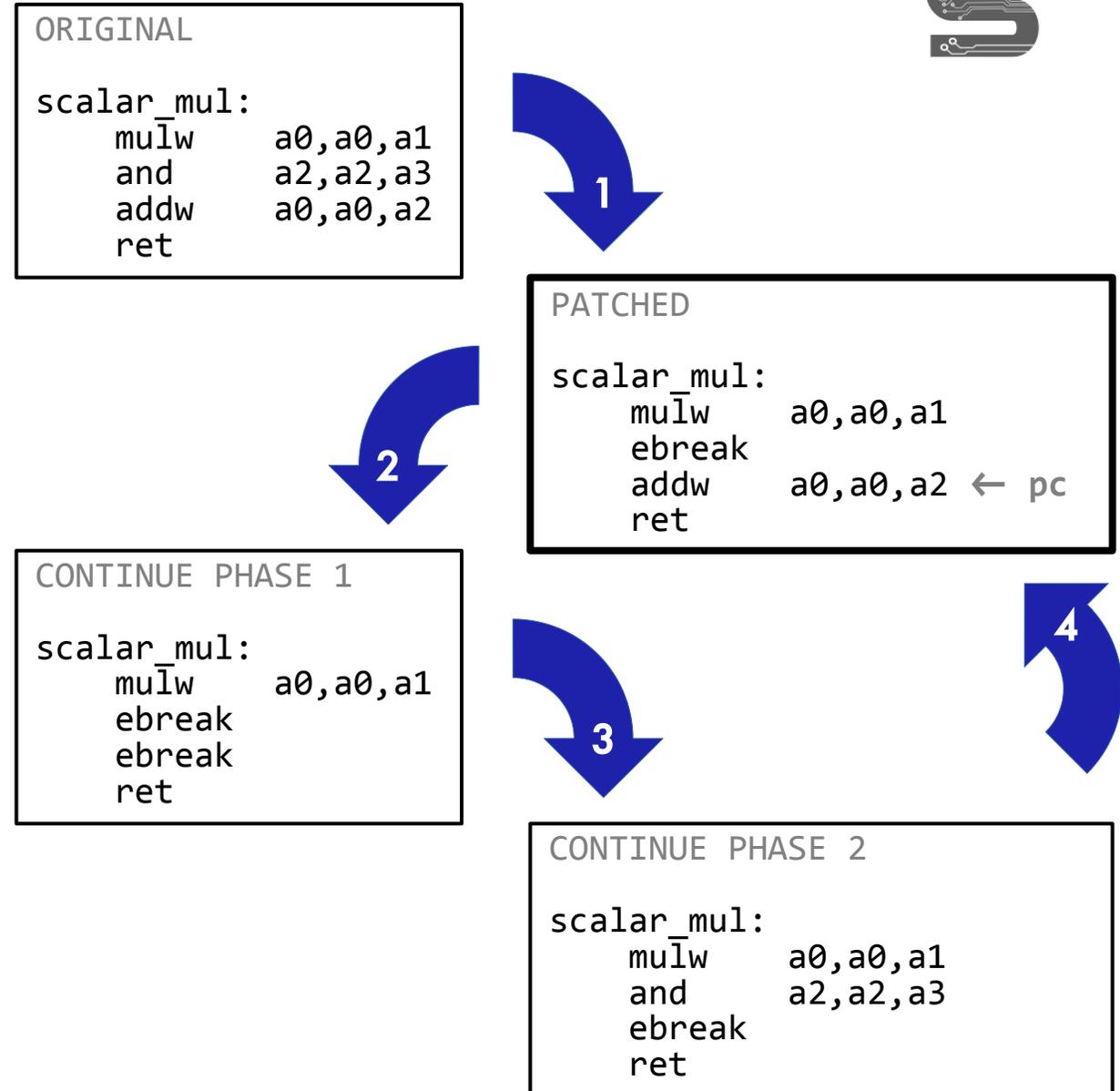
- Для продолжения исполнения:
 - Ставится временный breakpoint на следующую инструкцию
 - Восстанавливается текущая инструкция
 - Возобновляется исполнение отлаживаемой программы, которая тут же останавливается на следующей инструкции





Continue после остановки

- Для продолжения исполнения:
 - Ставится временный breakpoint на следующую инструкцию
 - Восстанавливается текущая инструкция
 - Возобновляется исполнение отлаживаемой программы
 - Восстанавливается начальный breakpoint, убирается временный, происходит «настоящее» возобновление исполнения



Дебажимся: step



```
buggy-sort.c
11  for (int i = 0; i < n && s != 0; i++) {
12      s = 0;
13      for (int j = 0; j < n; j++)
14          if (a[j].key > a[j + 1].key) {
B+  15              item t = a[i];
>  16              a[j] = a[j + 1];
17              a[j + 1] = t;
18              s++;
19          }
20      n--;

```

multi-thre Thread 0x7ffff7d887 (src) In: sort L16 PC: 0x555555551d4
Breakpoint 2 at 0x555555551b2: file buggy-sort.c, line 15.
(gdb) continue
Continuing.

Breakpoint 2, sort (a=0x7fffffff680, n=3) at buggy-sort.c:15
(gdb) step
(gdb) █



Step

- Иногда после остановки нужно перейти на следующую строку
- Можно поставить на нее breakpoint
- После остановки убрать breakpoint
- Отладчики предоставляют такую функциональность с помощью команды **step**
- Когда пользователь делает step, отладчик ставит breakpoint на следующую строку исполнения, а после остановки сам убирает этот breakpoint

ORIGINAL

```
scalar_mul:  
    mulw    a0,a0,a1  
    and     a2,a2,a3 ← pc  
    addw    a0,a0,a2  
    ret
```



Step

- Иногда после остановки нужно перейти на следующую строку
- Можно поставить на нее breakpoint
- После остановки убрать breakpoint
- Отладчики предоставляют такую функциональность с помощью команды **step**
- Когда пользователь делает step, отладчик ставит breakpoint на следующую строку исполнения, а после остановки сам убирает этот breakpoint

ORIGINAL

```
scalar_mul:
    mulw    a0,a0,a1
    and     a2,a2,a3 ← pc
    addw    a0,a0,a2
    ret
```



STEP PHASE 1

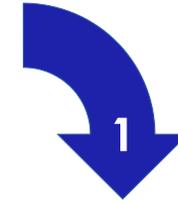
```
scalar_mul:
    mulw    a0,a0,a1
    and     a2,a2,a3 ← pc
    ebreak
    ret
```



Step

- Иногда после остановки нужно перейти на следующую строку
- Можно поставить на нее breakpoint
- После остановки убрать breakpoint
- Отладчики предоставляют такую функциональность с помощью команды **step**
- Когда пользователь делает step, отладчик ставит breakpoint на следующую строку исполнения, а после остановки сам убирает этот breakpoint

```
ORIGINAL
scalar_mul:
    mulw    a0,a0,a1
    and     a2,a2,a3 ← pc
    addw    a0,a0,a2
    ret
```



```
STEP PHASE 1
scalar_mul:
    mulw    a0,a0,a1
    and     a2,a2,a3 ← pc
    ebreak
    ret
```



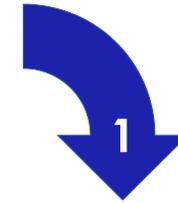
```
STEP PHASE 2
scalar_mul:
    mulw    a0,a0,a1
    and     a2,a2,a3
    ebreak  ← pc
    ret
```



Step

- Иногда после остановки нужно перейти на следующую строку
- Можно поставить на нее breakpoint
- После остановки убрать breakpoint
- Отладчики предоставляют такую функциональность с помощью команды **step**
- Когда пользователь делает step, отладчик ставит breakpoint на следующую строку исполнения, а после остановки сам убирает этот breakpoint

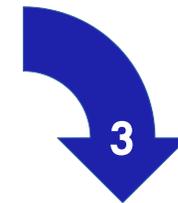
```
ORIGINAL
scalar_mul:
    mulw    a0,a0,a1
    and     a2,a2,a3 ← pc
    addw    a0,a0,a2
    ret
```



```
STEP PHASE 1
scalar_mul:
    mulw    a0,a0,a1
    and     a2,a2,a3 ← pc
    ebreak
    ret
```



```
STEP PHASE 2
scalar_mul:
    mulw    a0,a0,a1
    and     a2,a2,a3
    ebreak ← pc
    ret
```



```
STEP PHASE 3
scalar_mul:
    mulw    a0,a0,a1
    and     a2,a2,a3
    addw    a0,a0,a2 ← pc
    ret
```

Дебажимся: watch



```
buggy-sort.c
11  for (int i = 0; i < n && s != 0; i++) {
12      s = 0;
13      for (int j = 0; j < n; j++)
14          if (a[j].key > a[j + 1].key) {
B+  15              item t = a[i];
>  16              a[j] = a[j + 1];
17              a[j + 1] = t;
18              s++;
19          }
20      n--;

```

multi-thre Thread 0x7ffff7d887 (src) In: sort L16 PC: 0x555555551d4
Continuing.

Breakpoint 2, sort (a=0x7fffffd680, n=3) at buggy-sort.c:15
(gdb) step
(gdb) watch *a@3
Watchpoint 3: *a@3
(gdb) █

Дебажимся: watch



```
buggy-sort.c
11  for (int i = 0; i < n && s != 0; i++) {
12      s = 0;
13      for (int j = 0; j < n; j++)
14          if (a[j].key > a[j + 1].key) {
B+  15              item t = a[i];
>  16              a[j] = a[j + 1];
17              a[j + 1] = t;
18              s++;
19          }
20      n--;
}

void sort(item *a, int n) {
}

multi-thre Thread 0x7ffff7d887 (src) In: sort      L16  PC: 0x555555551d4
Continuing.

Breakpoint 2, sort (a=0x7ffffffffffd680, n=3)
(gdb) step
(gdb) watch *a@3
Watchpoint 3: *a@3
(gdb) █
```

Дебажимся: watch

```
buggy-sort.c
11  for (int i = 0; i < n && s != 0; i++) {
12      s = 0;
13      for (int j = 0; j < n; j++)
14          if (a[j].key > a[j + 1].key) {
15              item t = a[i];
16              a[j] = a[j + 1];
17              a[j + 1] = t;
18              s++;
19          }
20      n--;
```

B+
>

```
void sort(item *a, int n) {
```

3
↓

multi-thre Thread 0x7ffff7d887 (src) In: sort 16 PC: 0x555555551d4
Continuing.

Breakpoint 2, sort (a=0x7fffffd680, n=3)
(gdb) step
(gdb) watch *a@3
Watchpoint 3: *a@3
(gdb) █

Дебажимся: watch



```
buggy-sort.c
11  for (int i = 0; i < n && s != 0; i++) {
12      s = 0;
13      for (int j = 0; j < n; j++)
14          if (a[j].key > a[j + 1].key) {
15              item t = a[i];
16              a[j] = a[j + 1];
17              a[j + 1] = t;
18              s++;
19          }
20      n--;
```

B+
>

```
void sort(item *a, int n) {
```

multi-thre Thread 0x7ffff7d887 (src) In: sort 16 PC: 0x555555551d4
Continuing.

Breakpoint 2, sort (a=0x7fffffd680, n=3)
(gdb) step
(gdb) watch *a@3
Watchpoint 3: *a@3
(gdb) █

3

...

a[0] a[1] a[2]

...

Дебажимся: watch



```
buggy-sort.c
11  for (int i = 0; i < n && s != 0; i++) {
12      s = 0;
13      for (int j = 0; j < n; j++)
14          if (a[j].key > a[j + 1].key) {
B+  15              item t = a[i];
>  16              a[j] = a[j + 1];
17              a[j + 1] = t;
18              s++;
19          }
20      n--;

multi-thre Thread 0x7ffff7d887 (src) In: sort          L16  PC: 0x55555555209

Old value = {{data = 0x55555555600c "Nick", key = 5}, {data = 0x555555556011 "John", key = 13}, {data = 0x555555556016 "Betty", key = 8}}
New value = {{data = 0x55555555600c "Nick", key = 5}, {data = 0x555555556016 "Betty", key = 13}, {data = 0x555555556016 "Betty", key = 8}}
0x000055555555209 in sort (a=0x7fffffff680, n=3) at buggy-sort.c:16
(gdb) █
```

Дебажимся: watch



```
buggy-sort.c
11  for (int i = 0; i < n && s != 0; i++) {
12      s = 0;
13      for (int j = 0; j < n; j++)
14          if (a[j].key > a[j + 1].key) {
B+  15              item t = a[i];
16              a[j] = a[j + 1];
>  17              a[j + 1] = t;
18              s++;
19          }
20      n--;

```

```
multi-thre Thread 0x7ffff7d887 (src) In: sort          L17  PC: 0x5555555520d
Old value = {{data = 0x55555555600c "Nick", key = 5}, {data = 0x555555556016 "Betty", key = 13}, {data = 0x555555556016 "Betty", key = 8}}
New value = {{data = 0x55555555600c "Nick", key = 5}, {data = 0x555555556016 "Betty", key = 8}, {data = 0x555555556016 "Betty", key = 8}}
sort (a=0x7fffffd680, n=3) at buggy-sort.c:17
(gdb) █
```

Дебажимся: watch



```
buggy-sort.c
11  for (int i = 0; i < n && s != 0; i++) {
12      s = 0;
13      for (int j = 0; j < n; j++)
14          if (a[j].key > a[j + 1].key) {
B+  15              item t = a[i];
16              a[j] = a[j + 1];
>  17              a[j + 1] = t;
18              s++;
19          }
20      n--;

multi-thre Thread 0x7ffff7d887 (src) In: sort          L17  PC: 0x55555555230

Old value = {{data = 0x55555555600c "Nick", key = 5}, {data = 0x555555556016 "Betty", key = 8}, {data = 0x555555556016 "Betty", key = 8}}
New value = {{data = 0x55555555600c "Nick", key = 5}, {data = 0x555555556016 "Betty", key = 8}, {data = 0x55555555600c "Nick", key = 8}}
0x000055555555230 in sort (a=0x7fffffff680, n=3) at buggy-sort.c:17
(gdb) █
```

Дебажимся: watch



```
buggy-sort.c
11  for (int i = 0; i < n && s != 0; i++) {
12      s = 0;
13      for (int j = 0; j < n; j++)
14          if (a[j].key > a[j + 1].key) {
B+  15              item t = a[i];
16              a[j] = a[j + 1];
17              a[j + 1] = t;
>  18              s++;
19          }
20      n--;

```

```
multi-thre Thread 0x7ffff7d887 (src) In: sort          L18  PC: 0x55555555234
Old value = {{data = 0x55555555600c "Nick", key = 5}, {data = 0x555555556016 "B
etty", key = 8}, {data = 0x55555555600c "Nick", key = 8}}
New value = {{data = 0x55555555600c "Nick", key = 5}, {data = 0x555555556016 "B
etty", key = 8}, {data = 0x55555555600c "Nick", key = 5}}
sort (a=0x7fffffd680, n=3) at buggy-sort.c:18
(gdb) █
```



Watchpoint

- Иногда может быть удобно следить за изменением значения выражения
- Это делается с помощью *точек наблюдения (watchpoints)*
- **Точки наблюдения** реализованы через step после каждой инструкции с проверкой обращений в память после каждой инструкции

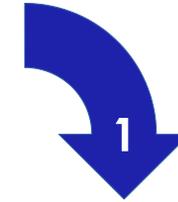
```
add_arr:  
    lw    a4,0(a0) ← pc  
    lw    a5,4(a0)  
    addw  a5,a5,a4  
    sw    a5,8(a0)  
    ret
```



Watchpoint

- Иногда может быть удобно следить за изменением значения выражения
- Это делается с помощью *точек наблюдения (watchpoints)*
- **Точки наблюдения** реализованы через step после каждой инструкции с проверкой обращений в память после каждой инструкции

```
add_arr:  
lw    a4,0(a0) ← pc  
lw    a5,4(a0)  
addw  a5,a5,a4  
sw    a5,8(a0)  
ret
```



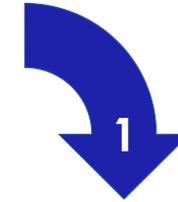
```
add_arr:  
lw    a4,0(a0)  
ebreak ← pc  
addw  a5,a5,a4  
sw    a5,8(a0)  
ret
```



Watchpoint

- Иногда может быть удобно следить за изменением значения выражения
- Это делается с помощью точек наблюдения (watchpoints)
- **Точки наблюдения** реализованы через step после каждой инструкции с проверкой обращений в память после каждой инструкции

```
add_arr:  
lw    a4,0(a0) ← pc  
lw    a5,4(a0)  
addw  a5,a5,a4  
sw    a5,8(a0)  
ret
```



```
add_arr:  
lw    a4,0(a0)  
ebreak ← pc  
addw  a5,a5,a4  
sw    a5,8(a0)  
ret
```

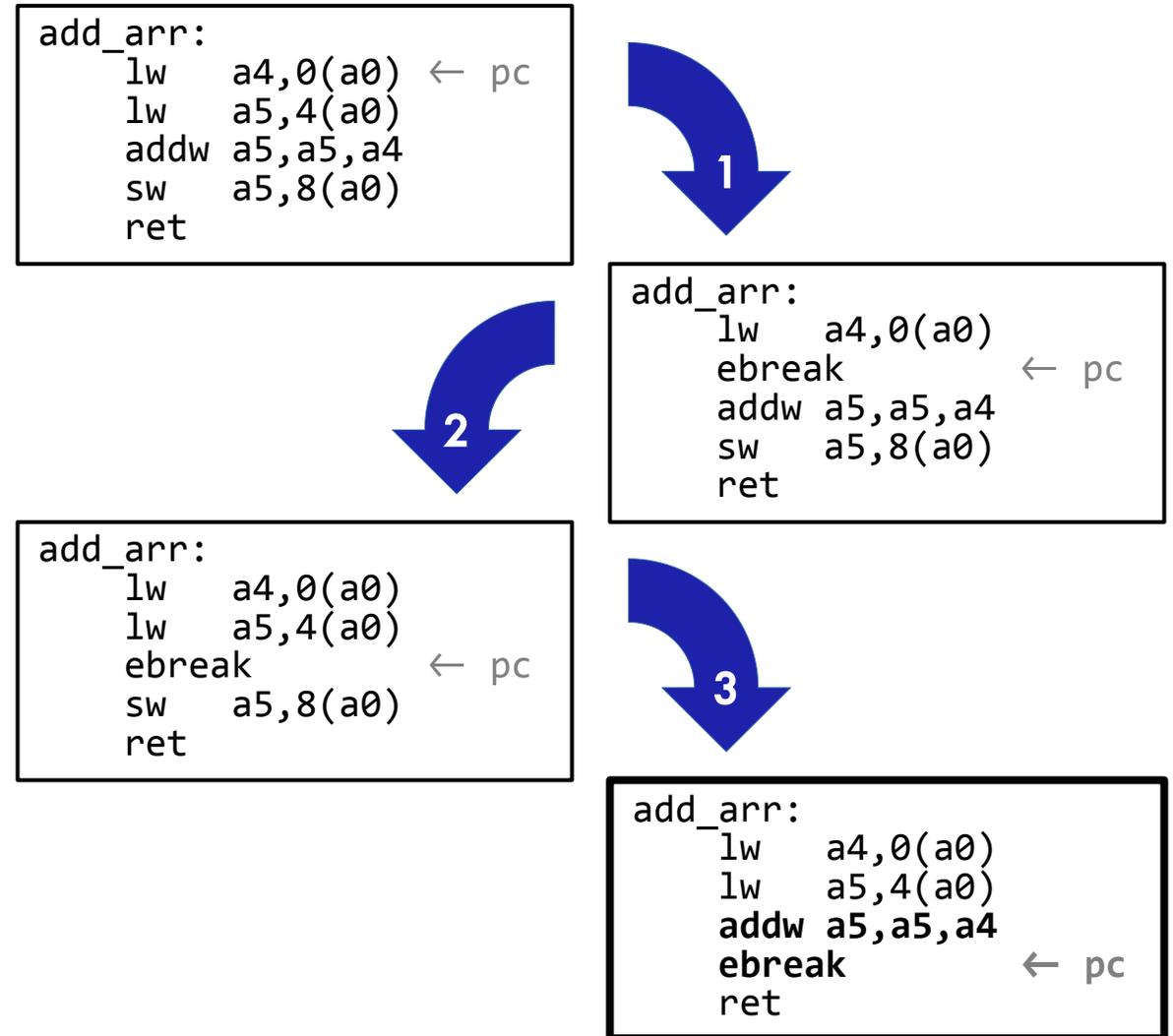


```
add_arr:  
lw    a4,0(a0)  
lw    a5,4(a0)  
ebreak ← pc  
sw    a5,8(a0)  
ret
```



Watchpoint

- Иногда может быть удобно следить за изменением значения выражения
- Это делается с помощью *точек наблюдения (watchpoints)*
- **Точки наблюдения** реализованы через step после каждой инструкции с проверкой обращений в память после каждой инструкции

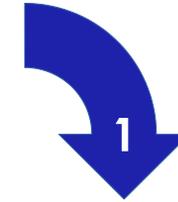




Watchpoint

- Иногда может быть удобно следить за изменением значения выражения
- Это делается с помощью *точек наблюдения (watchpoints)*
- **Точки наблюдения** реализованы через `step` после каждой инструкции с проверкой обращений в память после каждой инструкции
- К сожалению, операция `step` сама по себе довольно медленная, поэтому `watchpoint` может заметно замедлять исполнение отлаживаемой программы

```
add_arr:
    lw    a4,0(a0) ← pc
    lw    a5,4(a0)
    addw  a5,a5,a4
    sw    a5,8(a0)
    ret
```



```
add_arr:
    lw    a4,0(a0)
    ebreak ← pc
    addw  a5,a5,a4
    sw    a5,8(a0)
    ret
```



```
add_arr:
    lw    a4,0(a0)
    lw    a5,4(a0)
    ebreak ← pc
    sw    a5,8(a0)
    ret
```



```
add_arr:
    lw    a4,0(a0)
    lw    a5,4(a0)
    addw  a5,a5,a4
    ebreak ← pc
    ret
```



Ставим watchpoint

- Поставим watchpoint так, чтобы он сработал через 10 тысяч итераций цикла

```
bench.c
1 int main() {
2     int x = 0;
B+> 3     for (int i = 0; i < 10000; ++i)
4         if (i == 9999)
5             x = 1; ← watch for this
6 }
```

```
multi-thre Thread 0x7ffff7d887 (src) In: main      L3      PC: 0x55555555124
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, main () at bench.c:3
(gdb) watch x
Watchpoint 2: x
(gdb) continue
```

 17:40:23



Ставим watchpoint

- Поставим watchpoint так, чтобы он сработал через 10 тысяч итераций цикла

```
bench.c
1 int main() {
2     int x = 0;
B+> 3     for (int i = 0; i < 10000; ++i)
4         if (i == 9999)
5             x = 1; ← watch for this
6 }
```

```
multi-thre Thread 0x7ffff7d887 (src) In: main      L3      PC: 0x5555555513d
Watchpoint 2: x
Old value = 0
New value = 1
main () at bench.c:3
(gdb) █
```

 17:40:25



Ограничения

- Breakpoint можно ставить тогда и только тогда, когда
 - Возможно писать в память инструкций
 - Доступ в память инструкций эксклюзивный



Ограничения: ROM

- Некоторые платформы, не поддерживают запись в память инструкций во время исполнения
 - Например, микроконтроллер, в котором прошивка хранится в ROM памяти
- В ОС некоторые участки памяти могут быть защищены от записи даже через специальный системный вызов
 - Например, если это специальная shared область памяти



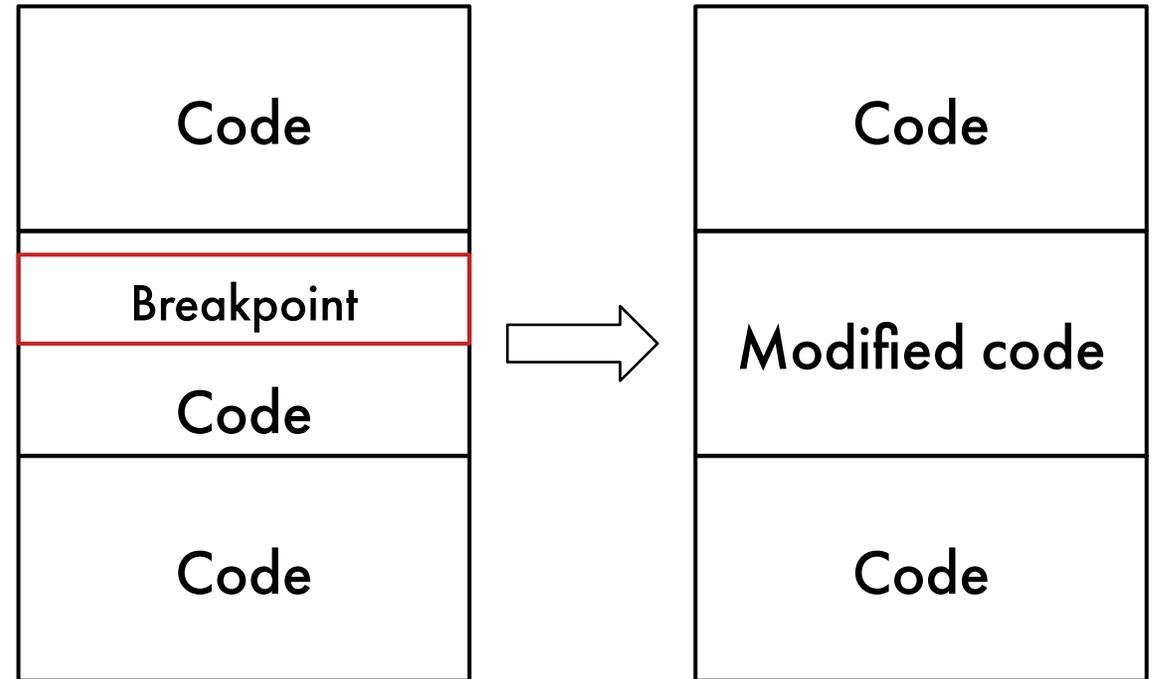
Ограничения: многопоточные приложения

- При отладке многопоточных приложений нам может быть не так важно знать состояние программы в конкретном месте
- Вместо этого может быть удобно останавливаться как только произошло изменение какого-то выражения (например, общего ресурса)
- Стандартная реализация точек наблюдения
 - Влияет на ход исполнения программы (инвазивна)
 - Заметно замедляет исполнение программы



Ограничения: самомодифицирующийся код и JIT

- При исполнении самомодифицирующего кода не только отладчик может изменять исполняемый код
- Такие модификации не будут согласованы
- Любой breakpoint может быть перезаписан в процессе исполнения
- Если breakpoint будет перезаписан, то не произойдет остановка



О себе

Типичные сценарии отладки и их ограничения

Аппаратные точки останова

Реализация аппаратных точек останова в отладчиках

Выводы



Идея аппаратных точек останова

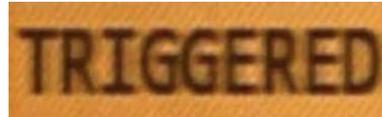
- Все проблемы рассмотренных break/watchpoint'ов связаны с тем, что остановка требует возможности эксклюзивной записи в память
- Можем ли мы как-то обойти это требование?



Идея аппаратных точек останова

- Все проблемы рассмотренных break/watchpoint'ов связаны с тем, что остановка требует возможности эксклюзивной записи в память
- Можем ли мы как-то обойти это требование?
- Да — научить аппаратуру более умному способу остановки

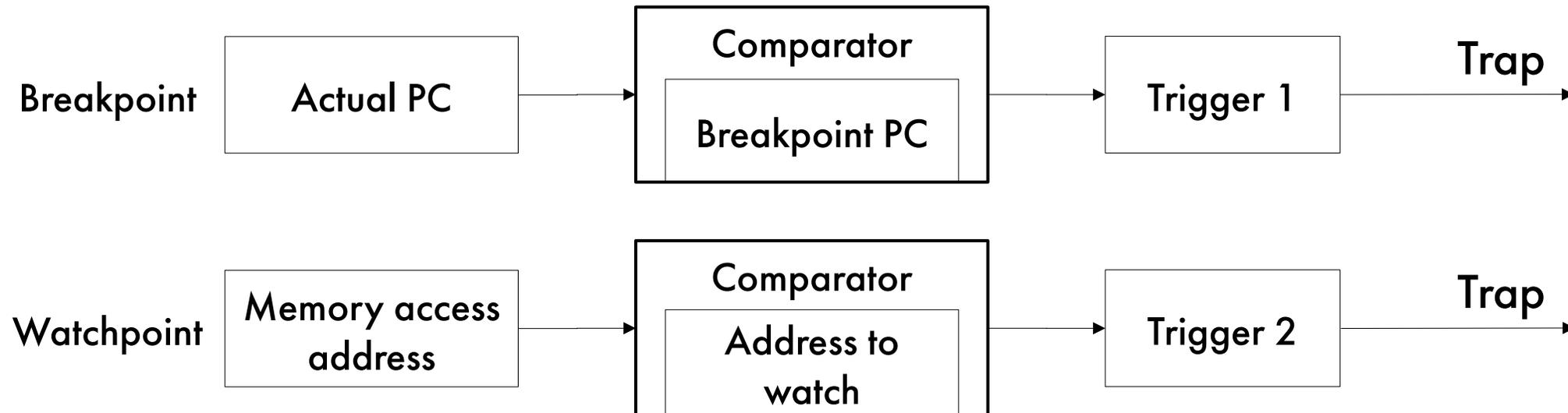
Идея аппаратных точек останова



- **Триггер** — в общем смысле, приводящий нечто в действие элемент
- По сути это это «спусковой крючок», который запускает определённый процесс или действие
- Пример: триггерная зона в игре — если игрок находится в заданной области, то на него накладывается эффект

Идея аппаратных точек останова

- На основе триггеров создадим новый особый модуль, в котором можно регистрировать значения РС, на которых требуется останавливаться
- Триггер «слушает» шину с актуальным РС и обращениями в память
- При срабатывании триггер генерирует такое же прерывание/исключение как и обычный trap





Бенчмаркаем аппаратные точки наблюдения

- Замерим сколько времени понадобится, чтобы watchpoint сработал через 100 тысяч итераций цикла:

bench.c:

```
int main() {
    int x = 0;
    for (int i = 0; i < 100000; ++i)
        if (i == 99999)
            x = 1; ← watch for this
}
```

- Для автоматизации замера времени напишем скрипты



Бенчмаркаем аппаратные точки наблюдения

hw-watch.py:

```
import gdb
import timeit

def main():
    gdb.execute("break 3")
    gdb.execute("run")
    gdb.execute("watch x")
    gdb.execute("continue")
    gdb.execute("continue")
    gdb.execute("continue")

print(f"Elapsed time: \
      {timeit.timeit(main, \
                     number=1):.3f}s")
gdb.execute("quit")
```

sw-watch.py:

```
import gdb
import timeit

def main():
    gdb.execute("break 3")
    gdb.execute("run")
    gdb.execute("set can-use-hw-watchpoints 0")
    gdb.execute("watch x")
    gdb.execute("continue")
    gdb.execute("continue")
    gdb.execute("continue")

print(f"Elapsed time: \
      {timeit.timeit(main, \
                     number=1):.3f}s")
gdb.execute("quit")
```



Бенчмаркаем аппаратные точки наблюдения

```
^ ~/demo at 15:00:08
> gdb --command=hw-watch.py ./bench

Breakpoint 1, main () at bench.c:3
3         for (int i = 0; i < 1000000; ++i)
Hardware watchpoint 2: x

Hardware watchpoint 2: x

Old value = 0
New value = 1
main () at bench.c:3
3         for (int i = 0; i < 1000000; ++i)

Watchpoint 2 deleted because the program has left the block in
which its expression is valid.
0x00007ffff7db0e08 in ?? () from /usr/lib/libc.so.6
[Inferior 1 (process 29196) exited normally]
--Type <RET> for more, q to quit, c to continue without paging--
Elapsed time:          0.028s

0.028s

^ ~/demo at 15:00:22
>
```

Бенчмаркаем программные точки наблюдения

```
A ~/demo at 15:00:22
> gdb --command=sw-watch.py ./bench
GNU gdb (GDB) 16.1
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./bench...
Breakpoint 1 at 0x1124: file bench.c, line 3.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, main () at bench.c:3
3      for (int i = 0; i < 1000000; ++i)
Watchpoint 2: x
```



Бенчмаркаем программные точки наблюдения

```
A ~/demo at 15:00:22
> gdb --command=sw-watch.py ./bench
GNU gdb (GDB) 16.1
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./bench...
Breakpoint 1 at 0x1124: file bench.c, line 3.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, main () at bench.c:3
3      for (int i = 0; i < 1000000; ++i)
Watchpoint 2: x
```



Бенчмаркаем программные точки наблюдения

```
Reading symbols from ./bench...
Breakpoint 1 at 0x1124: file bench.c, line 3.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".
```

```
Breakpoint 1, main () at bench.c:3
3      for (int i = 0; i < 1000000; ++i)
Watchpoint 2: x
```

```
Watchpoint 2: x
```

```
Old value = 0
```

```
New value = 1
```

```
main () at bench.c:3
```

```
3      for (int i = 0; i < 1000000; ++i)
```

```
Watchpoint 2 deleted because the program has left the block in
which its expression is valid.
```

```
0x00007ffff7db0e08 in ?? () from /usr/lib/libc.so.6
```

```
[Inferior 1 (process 35833) exited normally]
```

```
--Type <RET> for more, q to quit, c to continue without paging--
```

```
Elapsed time: 257.575s ←
```

```
^ ~ /demo
```

```
> █
```



257.575s

took 4m 19s ⌛ at 15:12:49 ©



Идея аппаратных точек останова

Благодаря такому подходу:

- Можно ставить breakpoint даже в read-only памяти



Идея аппаратных точек останова

Благодаря такому подходу:

- Можно ставить breakpoint даже в read-only памяти
- Не интрузивно влияет на код
 - Ускоряет отладку точками наблюдения (watchpoints)
 - Убирает лишние точки синхронизации при отладке многопоточки



Идея аппаратных точек останова

Благодаря такому подходу:

- Можно ставить breakpoint даже в read-only памяти
- Не интрузивно влияет на код
 - Ускоряет отладку точками наблюдения (watchpoints)
 - Убирает лишние точки синхронизации при отладке многопоточки
- Можно ставить breakpoint в самомодифицирующемся коде, не боясь, что он будет переписан



Идея аппаратных точек останова

Благодаря такому подходу:

- Можно ставить breakpoint даже в read-only памяти
- Не интрузивно влияет на код
 - Ускоряет отладку точками наблюдения (watchpoints)
 - Убирает лишние точки синхронизации при отладке многопоточки
- Можно ставить breakpoint в самомодифицирующемся коде, не боясь, что он будет переписан

Увы у всего есть цена: **количество триггеров ограничено** дизайном CPU

О себе

Типичные сценарии отладки и их ограничения

Аппаратные точки останова

Реализация аппаратных точек останова в отладчиках

Выводы



Мыслим систематично

- Триггеры — максимально низкоуровневый примитив



Мыслим систематично

- Триггеры — максимально низкоуровневый примитив
- Доступ к триггерам — только привилегированный



Мыслим систематично

- Триггеры — максимально низкоуровневый примитив
- Доступ к триггерам — только привилегированный
- Работать с триггерами может только ОС



Мыслим систематично

- Триггеры — максимально низкоуровневый примитив
- Доступ к триггерам — только привилегированный
- Работать с триггерами может только ОС
- Запрос на установку аппаратной точки останова отладчик делегирует ОС



Мыслим систематично

- Триггеры — максимально низкоуровневый примитив
- Доступ к триггерам — только привилегированный
- Работать с триггерами может только ОС
- Запрос на установку аппаратной точки останова отладчик делегирует ОС
- ОС передает информацию о сработавшем breakpoint через сигнал



Мыслим систематично

- Триггеры — максимально низкоуровневый примитив
- Доступ к триггерам — только привилегированный
- Работать с триггерами может только ОС
- Запрос на установку аппаратной точки останова отладчик делегирует ОС
- ОС передает информацию о сработавшем breakpoint через сигнал
- Отладчик передает управление пользователю



Пример реализации: GDB + Linux + RISC-V

- Схематичный пример возможной реализации для RISC-V и Linux:



Пример реализации: GDB + Linux + RISC-V

- Схематичный пример возможной реализации для RISC-V и Linux:
 - Конфигурация триггеров происходит через специальные системные регистры — CSR (Control&Status Register)



Пример реализации: GDB + Linux + RISC-V

- Схематичный пример возможной реализации для RISC-V и Linux:
 - Конфигурация триггеров происходит через специальные системные регистры — CSR (Control&Status Register)
 - GDB выполняет запрос на конфигурацию CSR через системный вызов ptrace



Пример реализации: GDB + Linux + RISC-V

- Схематичный пример возможной реализации для RISC-V и Linux:
 - Конфигурация триггеров происходит через специальные системные регистры – **CSR** (Control&Status Register)
 - GDB выполняет запрос на конфигурацию CSR через системный вызов **ptrace**
 - Информация об остановке передается через **SIGTRAP**



ptrace - process trace

SYNOPSIS

```
#include <sys/ptrace.h>
```

```
long ptrace(enum __ptrace_request op, pid_t pid,  
            void *addr, void *data);
```

DESCRIPTION

The `ptrace()` system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.



Трогаем лапкой ptrace API

- Пробуем узнать сколько нам доступно триггеров:

```
struct iovec iov;
```

```
struct user_hwdebug_state dbg_state;
```

```
iov.iov_base = &dbg_state;
```

```
iov.iov_len = sizeof (dbg_state);
```

```
ptrace (PTRACE_GETREGSET, pid, NT_RISCV_HW_WATCH, &iov);
```



Трогаем лапкой ptrace API

SYNOPSIS

```
#include <sys/uio.h>

struct iovec {
    void    *iov_base;    /* Starting address */
    size_t  iov_len;     /* Size of the memory pointed to by iov_base. */
};
```

DESCRIPTION

Describes a region of memory, beginning at iov base address and with the size of iov len bytes. System calls use arrays of this structure, where each element of the array represents a memory region, and the whole array represents a vector of memory regions. The maximum number of iovec structures in that array is limited by **IOV_MAX** (defined in <limits.h>, or accessible via the call sysconf(SC IOV MAX)).



Трогаем лапкой ptrace API

- Структура **user_hwdebug_state** является частью платформоспецифичного интерфейса ptrace и может отличаться для разных платформ
- Пример возможной реализации **user_hwdebug_state** для RISC-V:

```
struct user_hwdebug_state {
    uint64_t dbg_info; ← Num regs
    struct {
        uint64_t addr;
        uint64_t type;
        uint64_t len;
    } dbg_regs[16];
};
```



Трогаем лапкой ptrace API

- Если 16 точек останова не хватит, то можно будет расширить тривиальным патчем

```
struct user_hwdebug_state {
    uint64_t dbg_info;
    struct {
        uint64_t addr;
        uint64_t type;
        uint64_t len;
    } dbg_regs[16]; ← HW break/watchpoint configs
};
```



Трогаем лапкой ptrace API

- Высокоуровнево каждый триггер можно настроить с помощью 3 параметров:
- Адрес начала отслеживаемого региона памяти
- Длина отслеживаемого региона памяти
- Тип точки останова (breakpoint, read watchpoint, write watchpoint, access watchpoint)

```
struct user_hwdebug_state {
    uint64_t dbg_info;
    struct {
        uint64_t addr; ← Break/watchpoint address
        uint64_t type; ← Breakpoint or watchpoint
        uint64_t len;  ← Length of watching region
    } dbg_regs[16];
};
```



Трогаем лапкой ptrace API

- Узнаем сколько нам доступно триггеров:

```
ptrace (PTRACE_GETREGSET, pid, NT_RISCV_HW_WATCH, &iov);
```

```
printf ("Num triggers: %d\n", dbg_state.dbg_info);
```



Трогаем лапкой ptrace API

- Узнаем сколько нам доступно триггеров:

```
ptrace (PTRACE_GETREGSET, pid, NT_RISCV_HW_WATCH, &iov);
```

```
printf ("Num triggers: %d\n", dbg_state.dbg_info);
```

- А что за число мы вообще получили?



Трогаем лапкой ptrace API

- Узнаем сколько нам доступно триггеров:

```
ptrace (PTRACE_GETREGSET, pid, NT_RISCV_HW_WATCH, &iov);
```

```
printf ("Num triggers: %d\n", dbg_state.dbg_info);
```

- А что за число мы вообще получили?
 - Это CSR'ы?



Трогаем лапкой ptrace API

- Узнаем сколько нам доступно триггеров:

```
ptrace (PTRACE_GETREGSET, pid, NT_RISCV_HW_WATCH, &iov);
```

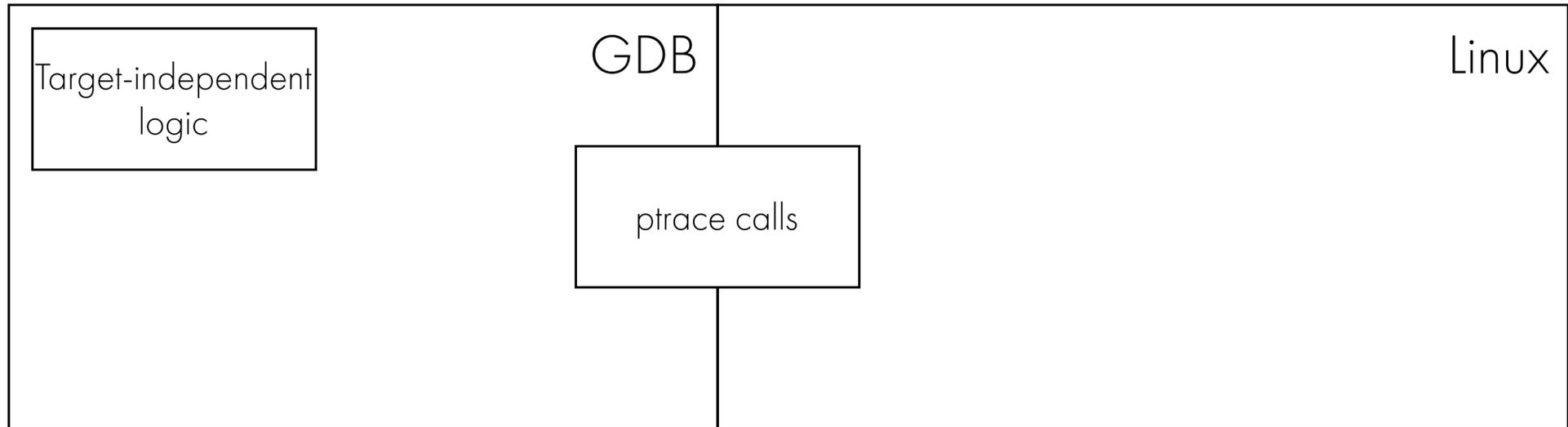
```
printf ("Num triggers: %d\n", dbg_state.dbg_info);
```

- А что за число мы вообще получили?
 - Это CSR'ы?
 - Триггеры?



Что скрывается за ptrace, когда мы ставим watchpoint?

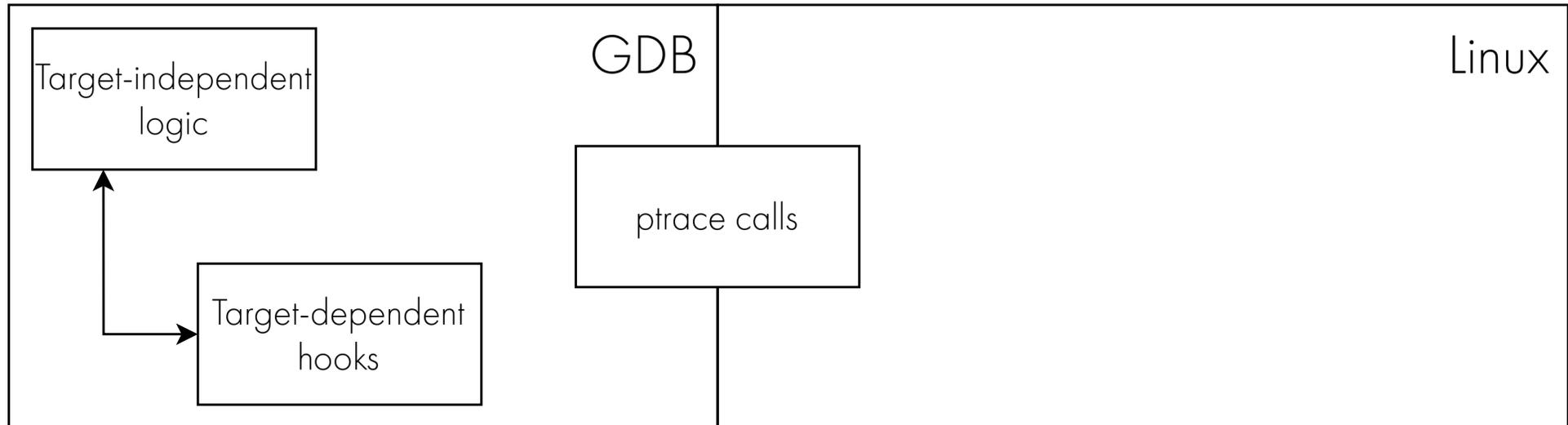
- Все команды GDB сначала обрабатываются платформонезависимой логикой





Что скрывается за ptrace, когда мы ставим watchpoint?

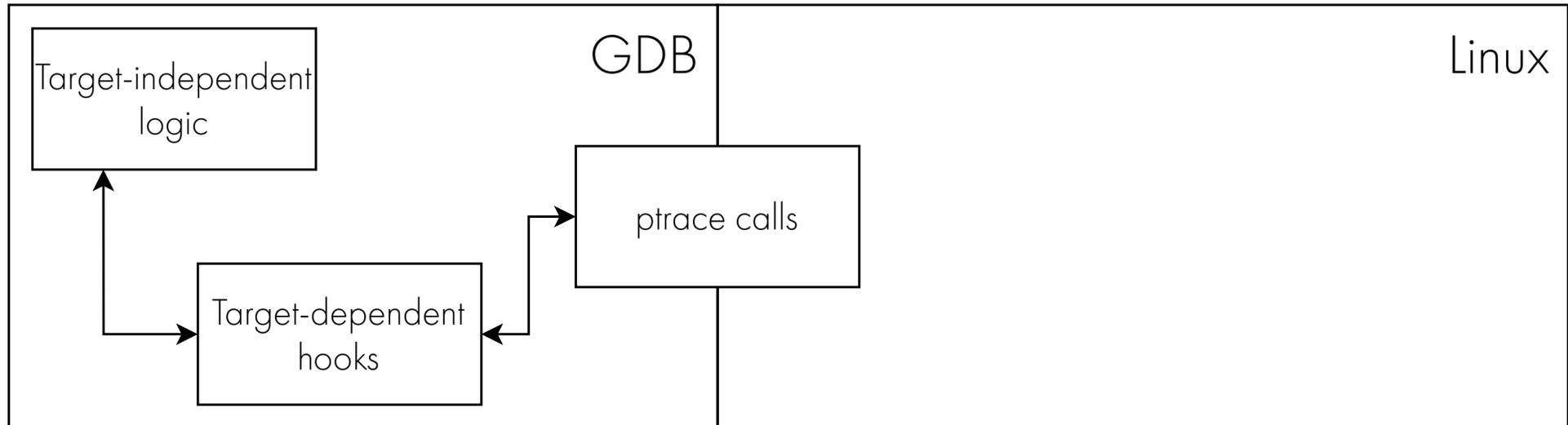
- При необходимости вызываются платформо-зависимые хуки





Что скрывается за ptrace, когда мы ставим watchpoint?

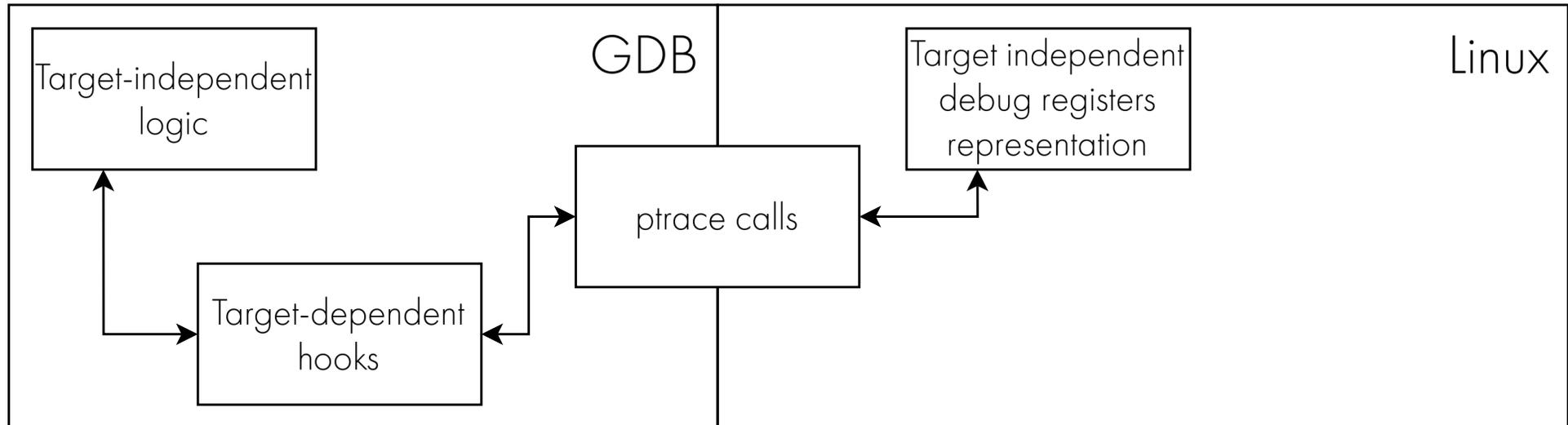
- Содержимое **iovec** структуры, передаваемой в **ptrace**, платформоспецифично, поэтому **ptrace** вызывается внутри хуков





Что скрывается за ptrace, когда мы ставим watchpoint?

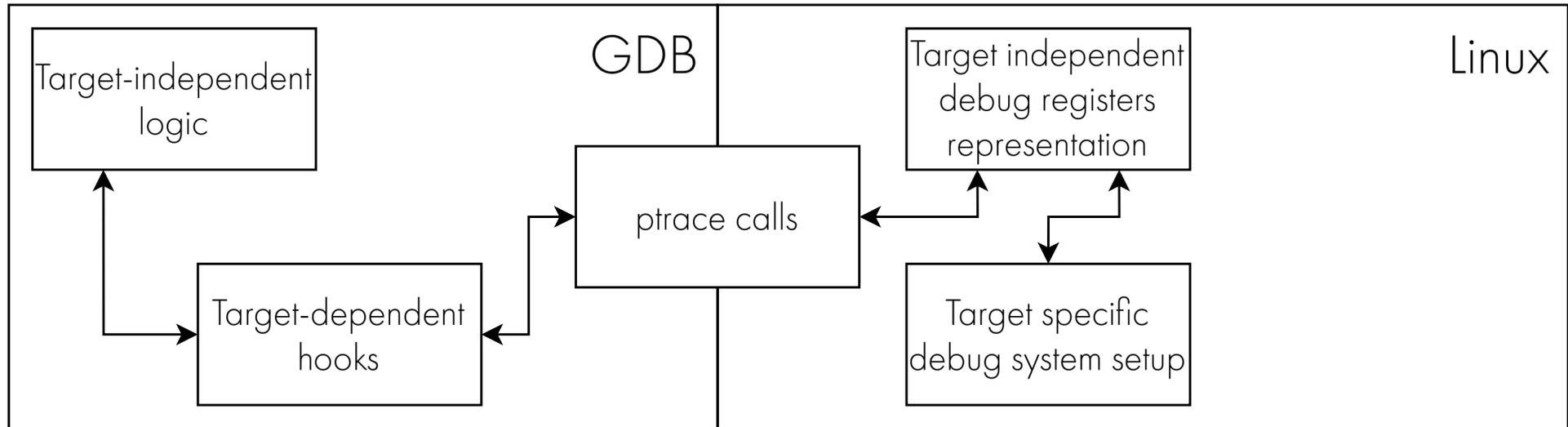
- В Linux, запрос на чтение/запись информации об аппаратных ресурсах представляются в платформо-независимом виде





Что скрывается за ptrace, когда мы ставим watchpoint?

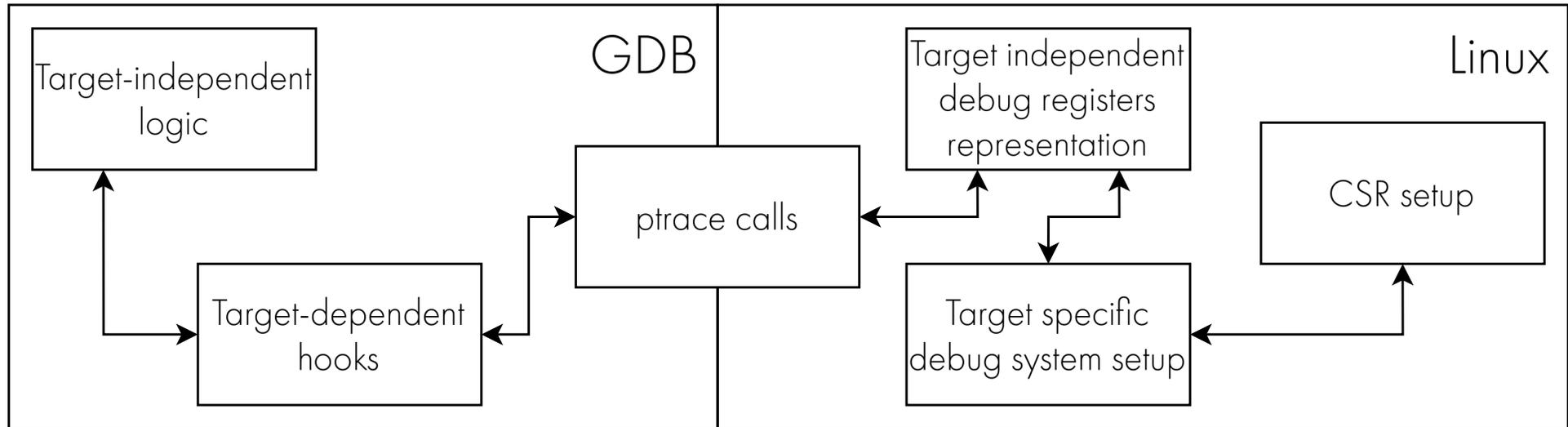
- Поэтому должна быть логика, трансформирующая это представление в настройку для конкретной платформы





Что скрывается за ptrace, когда мы ставим watchpoint?

- Для RISC-V это чтение/запись CSR регистров





Концепция обобщенных debug регистров

```
ptrace (PTRACE_GETREGSET, pid, NT_RISCV_HW_WATCH, &iov);
```

```
printf ("Num debug regs: %d\n", dbg_state.dbg_info);
```

- А что за число мы вообще получили?
- Это число «обобщенных» *debug* регистров
- Это означает, что конфигурируем мы так же «обобщенные» *debug* регистры

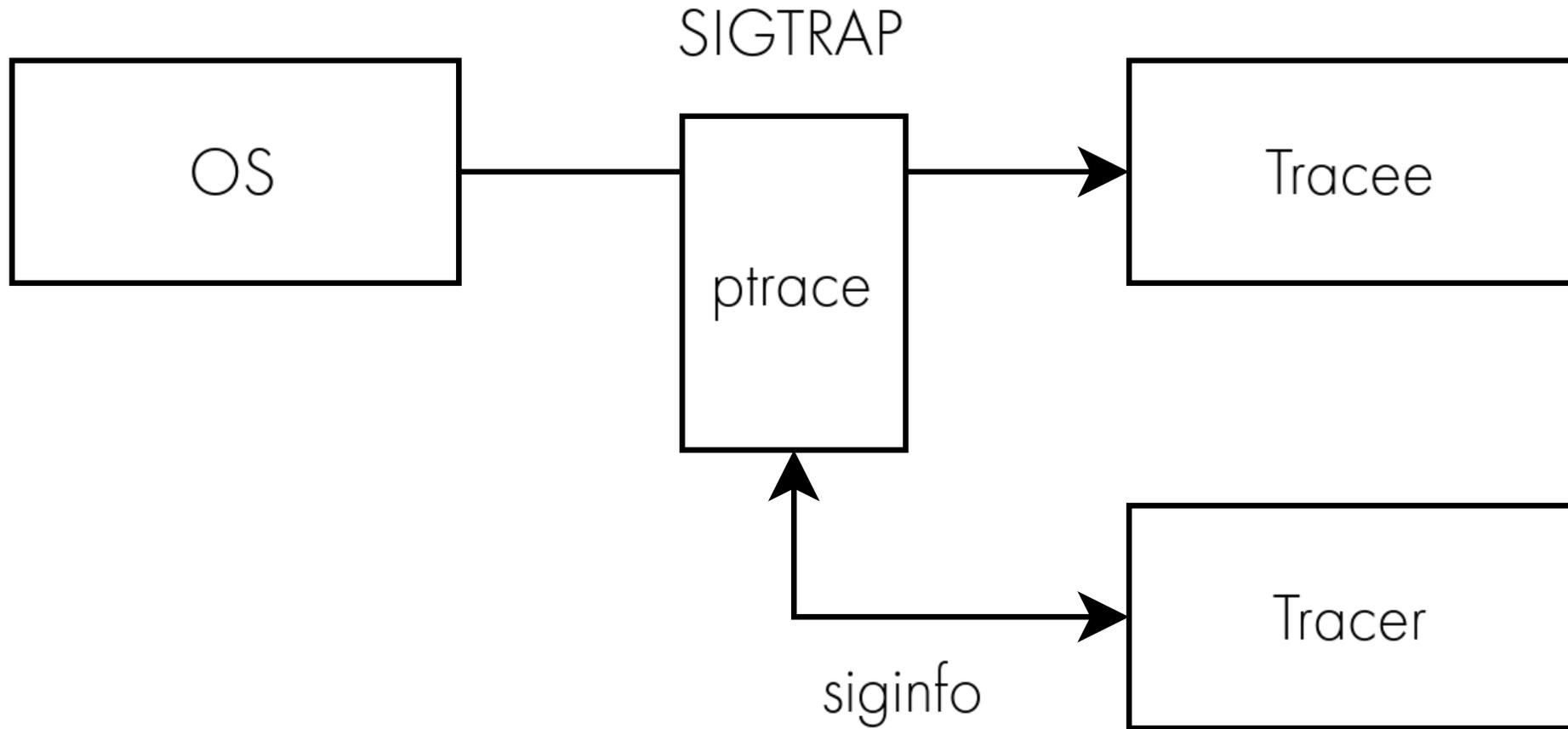


Попробуем сконфигурировать debug регистры

```
struct iovec iov;  
struct user_hwdebug_state regs = setup_debug_regs ();  
  
iov.iov_base = &regs;  
  
int count = num_debug_regs ();  
iov.iov_len = (offsetof (struct user_hwdebug_state, dbg_regs)  
              + count * sizeof (regs.dbg_regs[0]));  
  
ptrace (PTRACE_SETREGSET, pid, NT_RISCV_HW_WATCH, &iov);
```

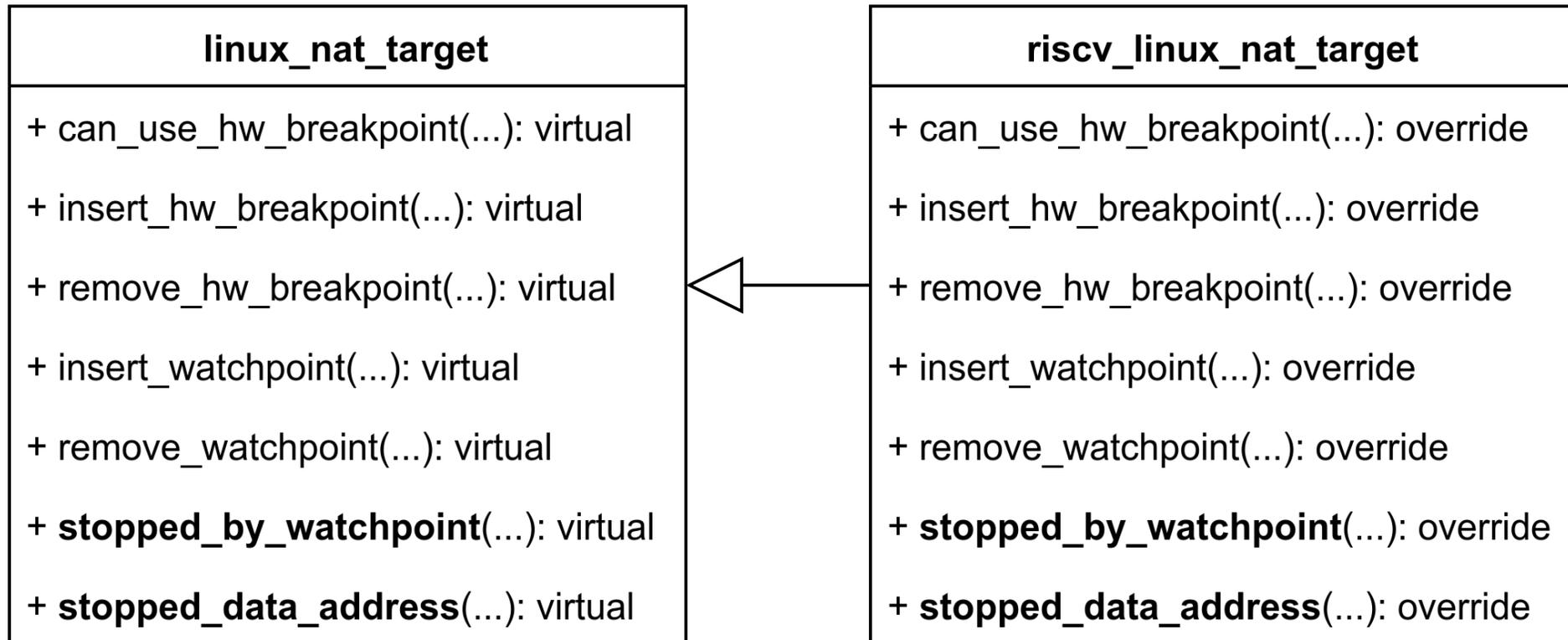


Получаем информацию об остановке





Интегрируем функциональность в отладчик (gdb native)



О себе

Типичные сценарии отладки и их ограничения

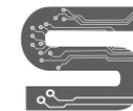
Аппаратные точки останова

Реализация аппаратных точек останова в отладчиках

Выводы

Подводим итоги

- Мы научились:





Подводим итоги

- Мы научились:
 - Узнавать количество доступных «обобщенных» debug регистров



Подводим итоги

- Мы научились:
 - Узнавать количество доступных «обобщенных» debug регистров
 - Конфигурировать «обобщенные» debug регистры



Подводим итоги

- Мы научились:
 - Узнавать количество доступных «обобщенных» debug регистров
 - Конфигурировать «обобщенные» debug регистры
 - Ставить аппаратный breakpoint



Подводим итоги

- Мы научились:
 - Узнавать количество доступных «обобщенных» debug регистров
 - Конфигурировать «обобщенные» debug регистры
 - Ставить аппаратный breakpoint
 - Ставить аппаратный watchpoint

Q&A