COMP 737011 - Memory Safety and Programming Language Design

Lecture 4: Memory Exhaustion

Xu, Hui

xuh@fudan.edu.cn

Outline

- 1. Stack Overflow and Heap Exhaustion
- 2. Auto Memory Reclaim
- 3. Exception Handling and Stack Unwinding

1. Stack Overflow and Heap Exhaustion

Warm Up

- Can you find a list to overflow the stack?

```
struct List{
    int val;
    struct List* next;
};
```

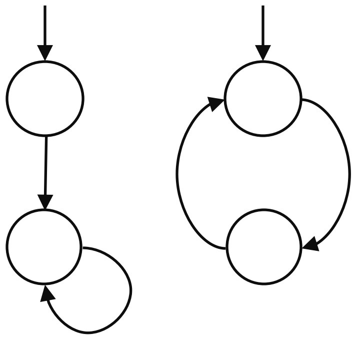
```
void process(struct List* l, int cnt){
    printf("%d\n", cnt);
    if(l->next != NULL)
        process(l->next, ++cnt);
}
```

- Sample solution

```
void main(void){
    struct List* list = malloc(sizeof(struct List));
    list->val = 1;
    list->next = list;
    process(list, 0);
}
```

Stack Size is Limited

- Default stack size: 8MB for each thread in Linux
  - You may check the setting with the ulimit command
- Reaching the limit would cause stack overflow
- Why not use a large stack?
  - Mainly used to save the contexts of function calls
  - Developers should not place large data on stack
- Vulnerable code: recursive function calls



```
#: ulimit -a
max locked memory       (kbytes, -l) 65536
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
stack size              (kbytes, -s) 8192
max user processes              (-u) 30687
```

You May Adjust The Stack Limit

- System users: ulimit command

```
#: ulimit -s unlimited
#: ulimit -a
max locked memory       (kbytes, -l) 65536
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
stack size              (kbytes, -s) unlimited
```

Set the stack size to unlimited

- Developers: use the setrlimit() function

```
struct rlimit r;
int result;
result = getrlimit(RLIMIT_STACK, &r);
fprintf(stderr, "stack result = %d\n", r.rlim_cur);
r.rlim_cur = 64 * 1024L *1024L;
result = setrlimit(RLIMIT_STACK, &r);
result = getrlimit(RLIMIT_STACK, &r);
fprintf(stderr, "stack result = %d\n", r.rlim_cur);
```

Set the stack size to 64 MB

How to Handle Stack Overflow?

- The OS usually kills the process directly. Why?
- We can register a handler for the SIGSEGV signal.
- Executing the event handler needs an extra stack.

Heap Exhaustion and Handling

- Always check: based on the return value of malloc().
  - Returns 0 if fails.
- Overcommit: could be killed by the OS.
  - Register a handler for the SIGKILL signal?
- To Small to Fail & OOM Killer
  - If the required space is small (< 8 pages), malloc() should never fail when overcommit is enabled.
  - If no enough memory, a process would be killed by the OOM killer:
    - based on badness of each process
    - calculated based on the vmsize and uptime of each process

The "too small to fail" memory-allocation rule, https://lwn.net/Articles/627419/

Overcommit

- A lazy mode memory allocation mechanism
  - malloc() successful does not mean the physical memory is allocated
  - The physical memory is allocated when being accessed
- Linux has three options
  - 1: always overcommit, never check
  - 2: always check, never overcommit
  - 0: heuristic overcommit (this is the default)

```
#: sudo sysctl -w vm.overcommit_memory=2
```

Overcommit: Example

- Try the following program with different settings

```
#define LARGE_SIZE 1024L*1024L*1024L*256L
void main(void){
    char* p = malloc (LARGE_SIZE);
    if(p == 0) {
        printf("malloc failed\n");
    } else {
        memset (p, 1, LARGE_SIZE);
    }
}
```

→ 256 GB

→ allocation failure

→ allocation successful access the memory

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

→ killed by the OS

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

→ allocation failure

To Small to Fail: Example

```c
#define SMALL_SIZE 1024L
void exhaustheap() {
    for(long i=0; i < INT64_MAX; i++) {
        char* p = malloc (SMALL_SIZE);
        if(p == 0){
            printf("the %ldth malloc failed\n", i);
            break;
        } else {
            printf("access the %ldth memory chunk,...", i);
            memset (p, 0, sizeof (SMALL_SIZE));
            printf(", done\n", i);
        }
    }
}
```

```
#: sudo sysctl -w vm.overcommit_memory=2
#:~/4-memoxhaustion$ ./a.out
...
access the 2705176th memory chunk,..., done
the 2705177th malloc failed
```

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

2. Auto Memory Reclaim

Memory Leakage

- Forget to free the out-of-use heap memory
- The memory space is unavailable to be reused

```
#define SMALL_SIZE 1024L
char* p = malloc (SMALL_SIZE);
...//free(p)
p = malloc (SMALL_SIZE);
```

Auto Reclaim Challenge

- Memory units for local data allocated on stack are automatically reclaimed when a function returns.
- Heap is hard to be reclaimed automatically.
  - There could be multiple references across functions.
  - Pointer analysis is NP-hard in general.

Cleanup Attribute

- Set a cleanup function to be executed when the function returns.
- The function is ineffective if an exception occurs

```
void free_buffer(char **buffer) {
    printf("Freeing buffer\n");
    free(*buffer);
}

void toy() {
    char *buf __attribute__ ((__cleanup__(free_buffer))) = malloc(10);
    snprintf(buf, 10, "%s", "any chars");
    printf("Buffer: %s\n", buf);
}
```

```
0x00000000004011a0 <+0>:     push   rbp
...
0x00000000004011ed <+77>:    call   0x401040 <printf@plt>
0x00000000004011f2 <+82>:    lea    rdi,[rbp-0x8]
0x00000000004011f6 <+86>:    mov    DWORD PTR [rbp-0x10],eax
0x00000000004011f9 <+89>:    call   0x401160 <free_buffer>
0x00000000004011fe <+94>:    add    rsp,0x10
0x0000000000401202 <+98>:    pop    rbp
0x0000000000401203 <+99>:    ret
```

C++ Auto Destruction

- Execute the destructor of objects on the stack automatically

```
class MyClass{
  private:
    int id;
  public:
    MyClass(int v) { id = v; }
    ~MyClass() { cout << "delete:"<< id << endl; }
};

void toy() {
    MyClass c1 = MyClass(100);
    MyClass* c2 = new MyClass(200);
}
```

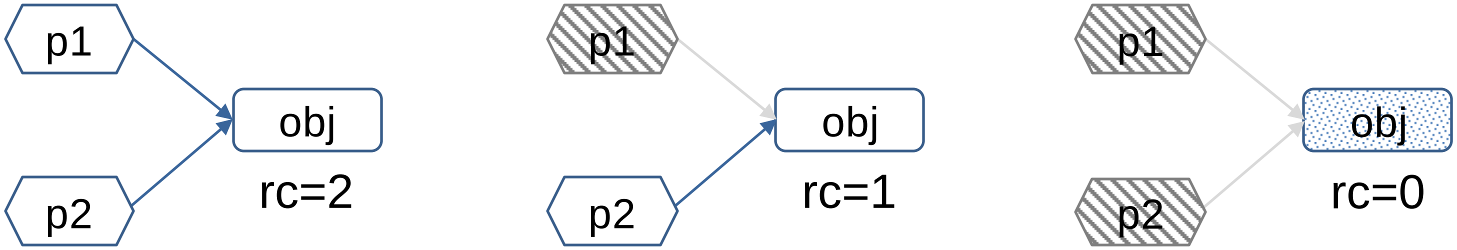
```
#:./a.out
delete:100
```

C++ Auto Destruction: Assembly Code

```
0x0000000000401250 <+0>:     push   rbx
0x0000000000401251 <+1>:     sub    rsp,0x10
0x0000000000401255 <+5>:     lea    rdi,[rsp+0x8]
0x000000000040125a <+10>:    mov    esi,0x64
0x000000000040125f <+15>:    call   0x4012b0 <_ZN7MyClassC2Ei>
0x0000000000401264 <+20>:    mov    edi,0x4
0x0000000000401269 <+25>:    call   0x401090 <_Znwm@plt>
0x000000000040126e <+30>:    mov    rdi,rax
0x0000000000401271 <+33>:    mov    esi,0xc8
0x0000000000401276 <+38>:    call   0x4012b0 <_ZN7MyClassC2Ei>
0x000000000040127b <+43>:    lea    rdi,[rsp+0x8]
0x0000000000401280 <+48>:    call   0x4012c0 <_ZN7MyClassD2Ev>
0x0000000000401285 <+53>:    add    rsp,0x10
0x0000000000401289 <+57>:    pop    rbx
0x000000000040128a <+58>:    ret
~~0x000000000040128b <+59>:    mov    rbx,rax~~
~~0x000000000040128e <+62>:    lea    rdi,[rsp+0x8]~~
~~0x0000000000401293 <+67>:    call   0x4012c0 <_ZN7MyClassD2Ev>~~
~~0x0000000000401298 <+72>:    mov    rdi,rbx~~
~~0x000000000040129b <+75>:    call   0x401100 <_Unwind_Resume@plt>~~
```

Smart Pointers

- Why? Static analysis cannot handle pointers
- Dynamically track the number of object pointers
- Reclaim the memory once no variable owns it



Smart Pointer in C++: shared_ptr

- Share an object among multiple pointers with a reference counter.
- Destroy the object when the last remaining shared_ptr owning the object is destroyed or reassigned.

```
void toy() {
    shared_ptr<MyClass> p1(new MyClass(100));
    //cout << "Ref counter: " << p1.use_count() << endl;
    shared_ptr<MyClass> p2 = p1;
    //cout << "Ref counter: " << p1.use_count() << endl;
}
```

How to Implement shared_ptr<T>

```
0x0000000000401290 <+0>:      push   r14
0x0000000000401292 <+2>:      push   rbx
0x0000000000401293 <+3>:      sub    rsp,0x28
0x0000000000401297 <+7>:      mov    edi,0x4
0x000000000040129c <+12>:     call   0x4010a0 <_Znwm@plt>
0x00000000004012a1 <+17>:     mov    rbx,rax
0x00000000004012a4 <+20>:     mov    rdi,rax
0x00000000004012a7 <+23>:     mov    esi,0x64
0x00000000004012ac <+28>:     call   0x401380 <_ZN7MyClassC2Ei>
0x00000000004012b1 <+33>:     lea    r14,[rsp+0x18]
0x00000000004012b6 <+38>:     mov    rdi,r14
0x00000000004012b9 <+41>:     mov    rsi,rbx
0x00000000004012bc <+44>:     call   0x401390 <_ZNSt10shared_ptrI7MyClassEC2IS0_vEEPT_>
0x00000000004012c1 <+49>:     lea    rbx,[rsp+0x8]
0x00000000004012c6 <+54>:     mov    rdi,rbx
0x00000000004012c9 <+57>:     mov    rsi,r14
0x00000000004012cc <+60>:     call   0x4013a0 <_ZNSt10shared_ptrI7MyClassEC2ERKS1_>
0x00000000004012d1 <+65>:     mov    rdi,rbx
0x00000000004012d4 <+68>:     call   0x4013b0
<_ZNSt12__shared_ptrI7MyClassLN9__gnu_cxx12_Lock_policyE2EED2Ev>
0x00000000004012d9 <+73>:     mov    rdi,r14
0x00000000004012dc <+76>:     call   0x4013b0
<_ZNSt12__shared_ptrI7MyClassLN9__gnu_cxx12_Lock_policyE2EED2Ev>
0x00000000004012e1 <+81>:     add    rsp,0x28
0x00000000004012e5 <+85>:     pop    rbx
0x00000000004012e6 <+86>:     pop    r14
0x00000000004012e8 <+88>:     ret
```

Create a shared_ptr

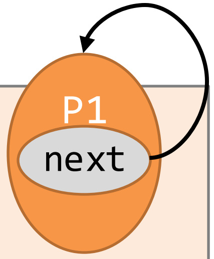
Increase the counter

Decrease the counter

Decrease the counter

Problem of Shared Pointer

- Problem of shared_ptr: reference cycles



```
class MyList{
private:
    int id;
public:
    MyList(int v) { id = v; }
    weak_ptr<MyList> next;
    ~MyList() { cout << "delete obj:"<< id << endl; }
};

int main() {
    shared_ptr<MyList> p(new MyList(100));
    p->next = p;
    cout << "Ref counter: " << p.use_count() << endl;
}
```

```
#:./a.out
Ref counter: 2
```

Use weak_ptr Instead

- weak_ptr: do not update the reference counter

```
class MyList{
private:
    int id;
public:
    MyList(int v) { id = v; }
    weak_ptr<MyList> next;
    ~MyList() { cout << "delete obj:"<< id << endl; }
};

int main() {
    shared_ptr<MyList> p(new MyList(100));
    p->next = p;
    cout << "Ref counter: " << p.use_count() << endl;
}
```

```
#:./a.out
Ref counter: 1
delete obj:100
```

Smart Pointer: unique_ptr

- Object is uniquely owned by one pointer
- Checked during compile time (similar to Rust ownership)
- User can transfer ownership through move()

```
int main() {
    unique_ptr<MyClass> p1(new MyClass(100));
    cout << "Before move: p1 = " << p1.get() << endl;
    //unique_ptr<MyClass> p2 = p1;
    unique_ptr<MyClass> p2 = move(p1);
    cout << "After move: p1 = " << p1.get() << endl;
    //cout << p1->val << endl;
    cout << p2->val << endl;
}
```

```
#:./a.out
Before move: p1 = 0x1476eb0
After move: p1 = 0
100
delete:100
```

Question

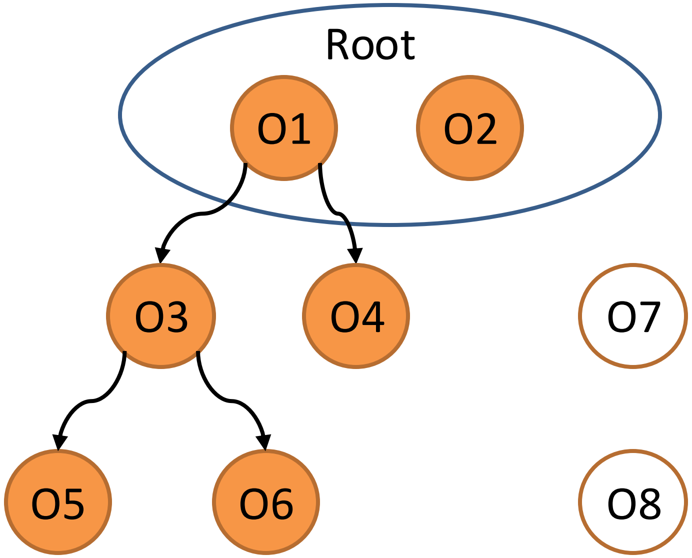
- Can you write a C++ code snippet with use after free bugs?
  - You cannot use delete/free
  - Based on the auto delete or shared_ptr mechanism

Garbage Collection

- When should the GC be triggered?
- Which objects should be recycled?
  - Reachability analysis
- How to recycle?
  - May cause slowdown due to intensive GC operation
  - Memory fragmentation issue

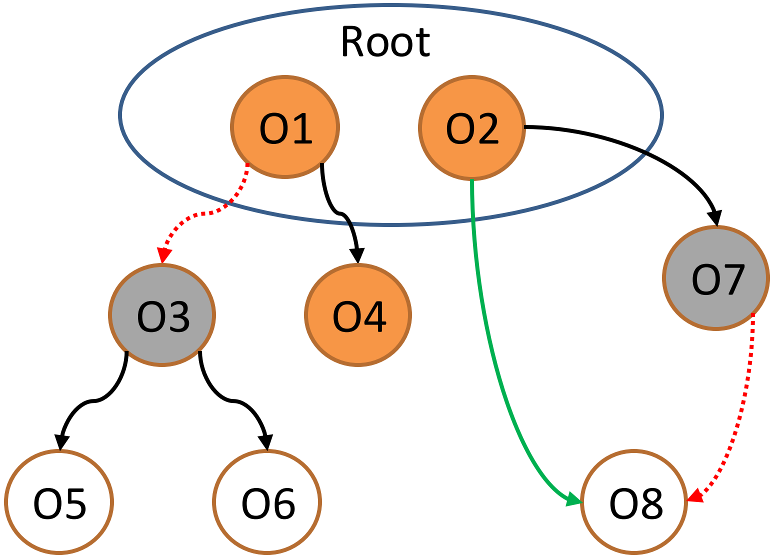
Reachability Analysis

- Stop the world
- Analyze from the root
- Unreachable objects should be recycled immediately



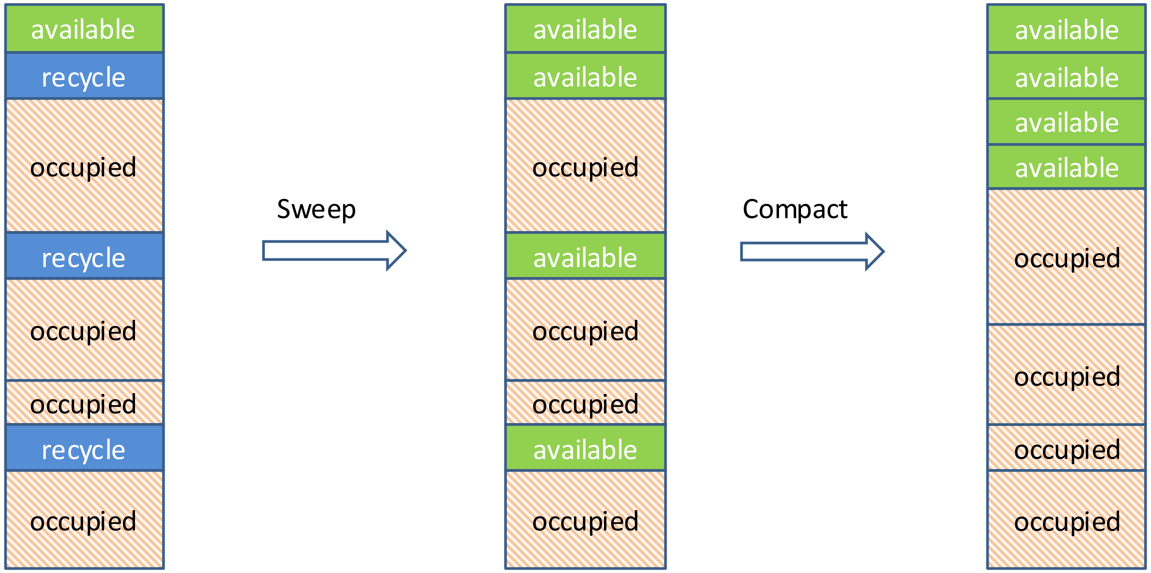
Incremental Analysis

- Do not need to stop the world
- Use three colors to record the temporary result
  - Orange: reached, and analysis (to other objects) is done
  - Gray: reached, but analysis is not finished
  - White: unreached object
- false negative?
- false positive?



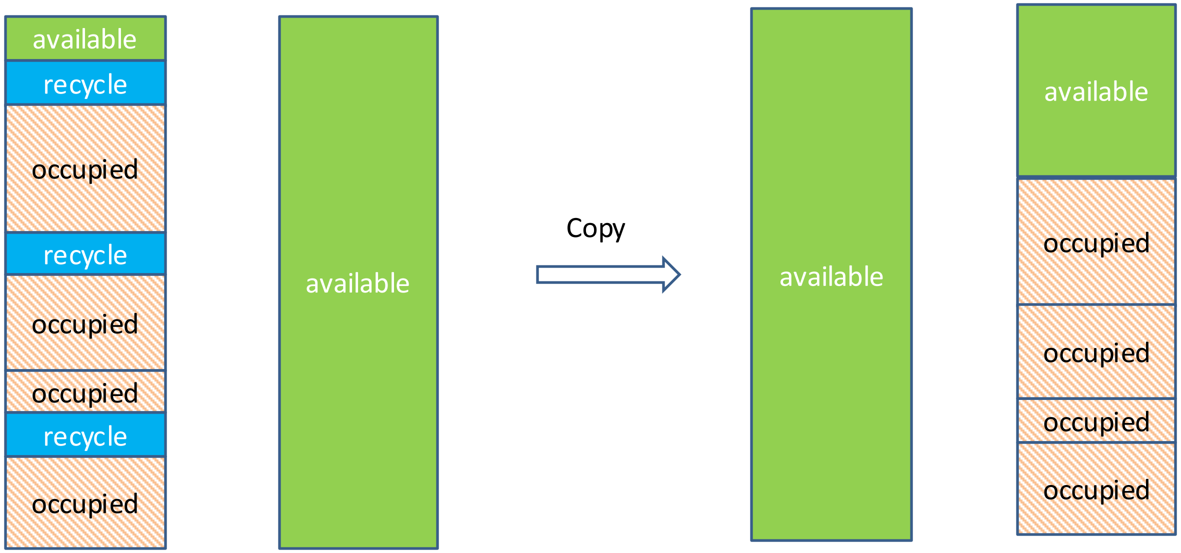
How to Recycle?

- For consecutive memory chunks (e.g., program break)
- Mark-sweep: suffers fragmentation issue
- Mark-compact: move all used units to one side
  - nontrivial overhead for moving data
  - when should the process be triggered?



Mark-Copy

- Two pieces of memory with the same size
  - the memory piece is still usable during copy
  - tradeoff between time and space

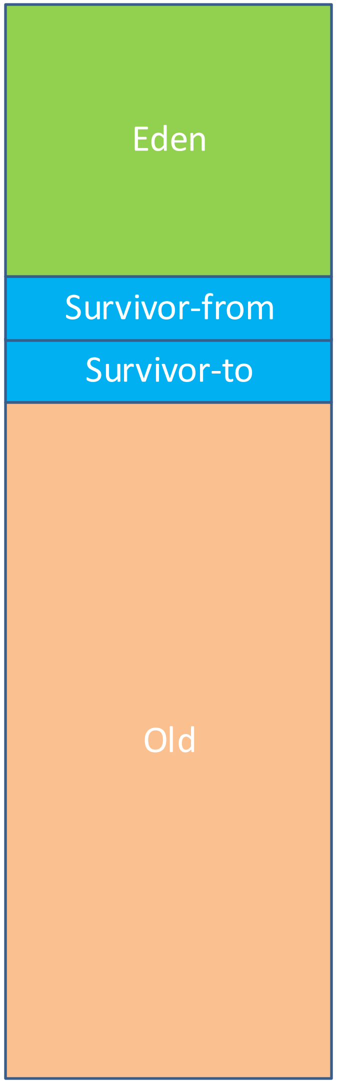


Observation

- Newly created objects tend to be recycled
- The objects survived after several GC rounds has a high chance to survive in the following round
- How can we utilize the observation for optimization?
  - Avoid frequent copy of old objects

Generational Collection

- Eden: for new objects
  - trigger minor GC if no space available
- Survivor: to host survived objects after minor GC
  - with two sub areas: from, to
  - minor GC(eden+from)=>to,
  - minor GC(eden+to)=>from
- Old: for objects survived after several rounds of minor GC
  - trigger major GC if no space available
  - large objects are saved to this area directly to avoid the overhead of copy.



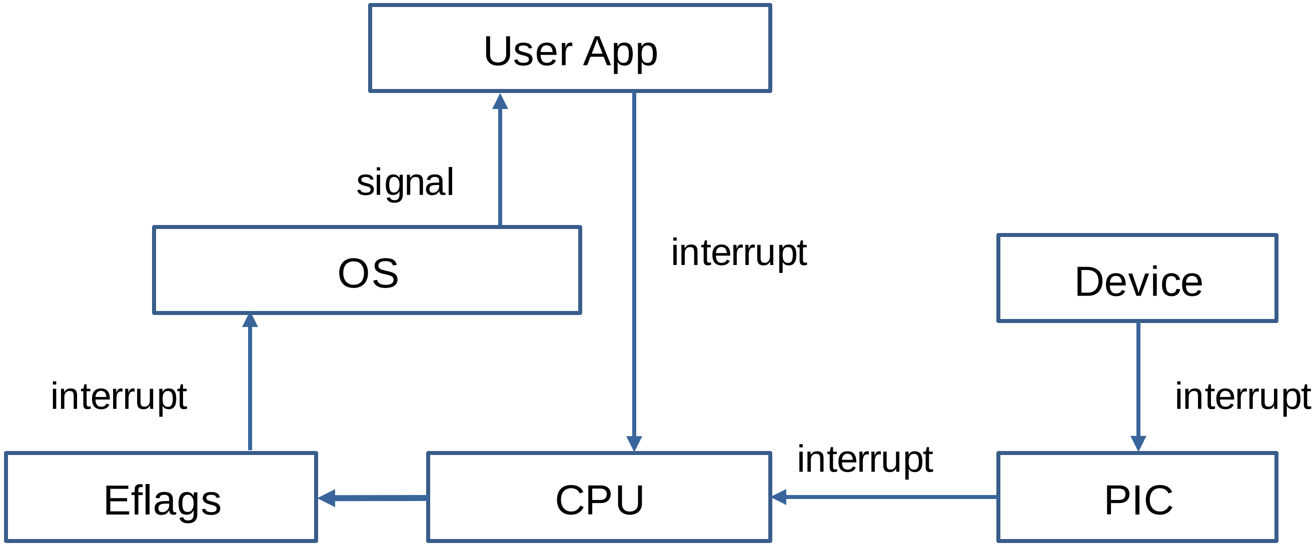
Implementing GC for C?

- Enumerate the Root node:
  - Variables of pointer types
  - Variables of data structures with pointers
- Check unreachable objects and delete them:
  - The allocator maintains the info of all allocated chunks
  - When? Before a function returns
- More reference:
  - BoehmGC: https://www.hboehm.info/gc/#details
  - Writing a Simple Garbage Collector in C: https://maplant.com/gc.html

3. Exception Handling and Stack Unwinding

Exceptions based on Origin

- CPU: interrupt
- OS: signal
- Application: user-defined exceptions



CPU Interrupt

- Page fault, divided by zero, etc
- Jump to the target exception handling address based on an interrupt vector, e.g., for X86
  - 0x00 Division by zero
  - 0x01 Single-step interrupt (see trap flag)
  - 0x03 Breakpoint (INT 3)
  - 0x04 Overflow
  - 0x06 Invalid Opcode
  - 0x0B Segment not present
  - 0x0C Stack Segment Fault
  - 0x0D General Protection Fault
  - 0x0E Page Fault
  - 0x10 x87 Floating Point Exception

OS Signal

- Kernel sends to other processes (IPC)
- POSIX signals
  - SIGFPE: floating-point error, overflow, underflow...
  - SIGSEGV: segmentation fault, invalid address...
  - SIGBUS: bus error, memory alignment issue
  - SIGILL: illegal instruction
  - SIGABRT: abort
  - SIGKILL:
  - …

Register the OS Signal

- Register the OS signal with signal or sigaction

```
void sethandler(void (*handler)(int,siginfo_t *,void *)){
    struct sigaction sa;
    sa.sa_sigaction = handler;
    sigaction(SIGFPE, &sa, NULL);
}

void handler(int signo, siginfo_t *info, void *extra){
    printf("SIGFPE received!!!\n");
    exit(-1);
}

int main(void){
    sethandler(handler);
    int a = 0;
    int x = 100/a;
}
```

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

Exception Handling Issue

- Where should the process continue?
  - find a landing pad
- How to set the required execution context?
  - restore callee-saved registers: rbp、rsp、rbx、r12-r15
- Release acquired resources
  - e.g, heap, file discriptor

setjmp/longjmp

- setjmp(env)：
  - backup registers and sets a recover point
  - return 0 if called directly, otherwise return a value if called by longjmp()
- longjmp(env, value)：
  - jump to a target address determined by value
  - restore all callee-saved registers: rbp、rsp、rbx、r12-r15

```
jmp_buf buf;
void handler(int signo, siginfo_t *info, void *extra){
    printf("SIGFPE received!!!\n");
    longjmp(buf,1);
}

int main(void){
    sethandler(handler);
    int a = 0;
    if (!setjmp(buf))
        int x = 100/a;
    else
        printf("Contine execution after a longjmp.\n");
}
```

Example of Stack Overflow Handling

- Need to confirm a handler with an extra stack.

```
sigjmp_buf buf;
struct List{
    int val;
    struct List* next;
};
void process(struct List* list, int cnt){
    if(list->next != NULL)
        process(list->next, ++cnt);
}
void sethandler(void (*handler)(int,siginfo_t *,void *))
void handler(int signo, siginfo_t *info, void *extra);
void main(void){
    sethandler(handler);
    struct List* list = malloc(sizeof(struct List));
    list->val = 1;
    list->next = list;
    if (setjmp(buf) == 0)
        process(list, 0);
    else
        printf("Continue after segmentation fault\n");
}
```

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

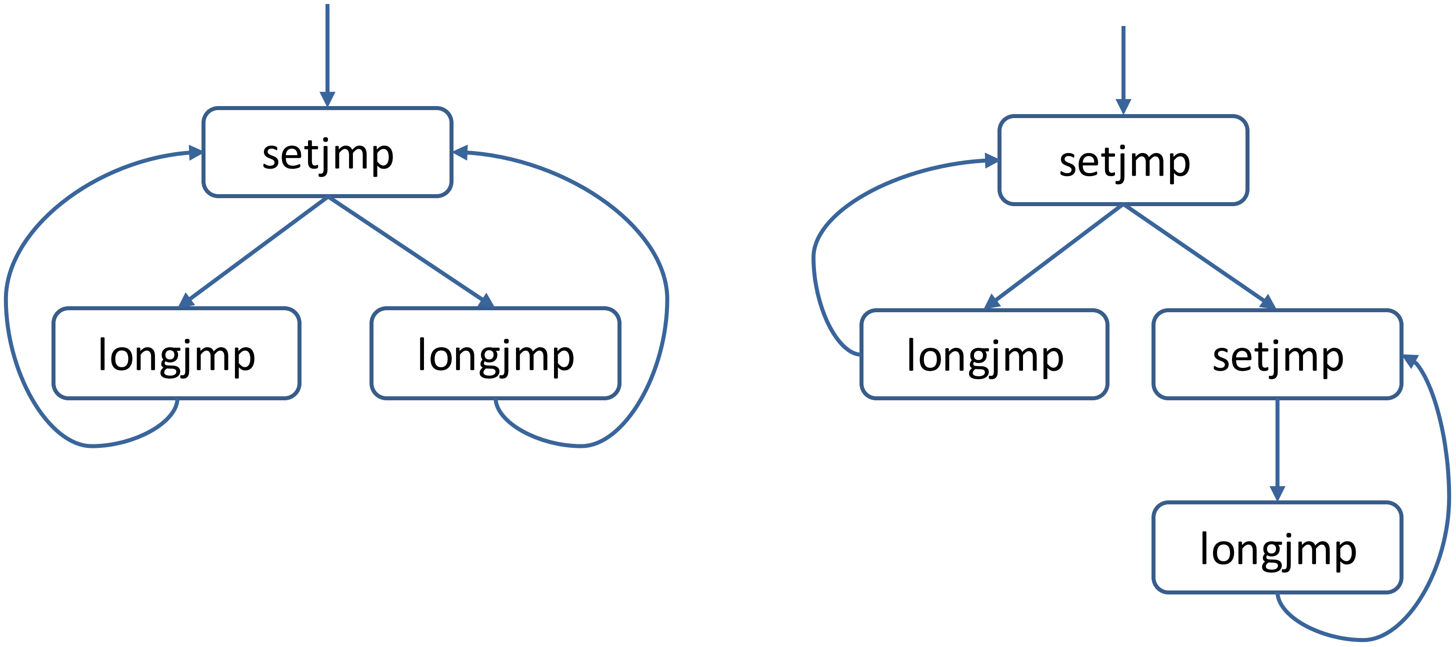
Full Code

```
#define SIGSTACK_SIZE 1024
void sethandler(void (*handler)(int,siginfo_t *,void *)){
    static char stack[SIGSTKSZ];
    struct sigaction sa;
    stack_t ss = { .ss_size = SIGSTKSZ, .ss_sp = stack, };
    memset(&sa, 0, sizeof(sigaction));
    sigemptyset(&sa.sa_mask);
    sa.sa_flags     = SA_NODEFER|SA_ONSTACK;
    sa.sa_sigaction = handler;
    sigaltstack(&ss, 0);
    sigaction(SIGSEGV, &sa, NULL);
}

void handler(int signo, siginfo_t *info, void *extra){
    longjmp(buf, 1);
}
```

Question

- How to support multiple setjmps/longjmps?



setjmp

longjmp

longjmp

setjmp

longjmp

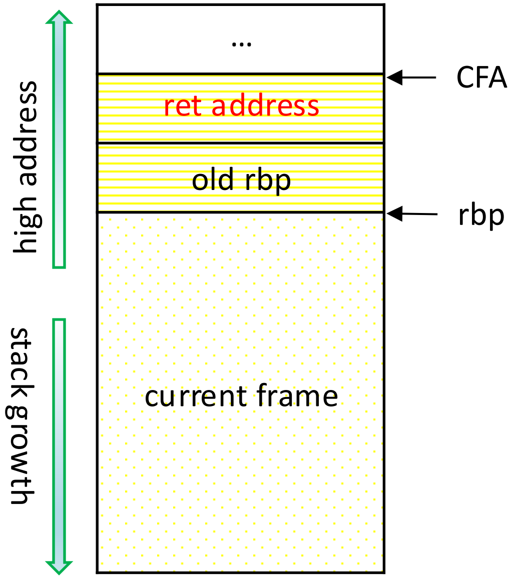
setjmp

longjmp

Stack Unwinding Problem

- Callee-saved registers should be restored
- Setjmp/longjmp is inconvenient or inefficient if widely used
- Can we have a better solution?

```
0x401130: push   %rbp
0x401131: mov    %rsp,%rbp
0x401134: sub    $0x10,%rsp
0x401138: mov    %edi,-0x8(%rbp)
0x40113b: cmpl   $0x0,-0x8(%rbp)
0x40113f: jne    0x401151
0x401145: movl   $0x1,-0x4(%rbp)
0x40114c: jmpq   0x40116d
0x401151: mov    -0x8(%rbp),%eax
0x401154: mov    -0x8(%rbp),%ecx
0x401157: sub    $0x1,%ecx
0x40115a: mov    %ecx,%edi
0x40115c: mov    %eax,-0xc(%rbp)
0x40115f: callq  0x401130
0x401164: mov    -0xc(%rbp),%ecx
0x401167: imul   %eax,%ecx
0x40116a: mov    %ecx,-0x4(%rbp)
0x40116d: mov    -0x4(%rbp),%eax
0x401170: add    $0x10,%rsp
0x401174: pop    %rbp
0x401175: retq
```



DWARF

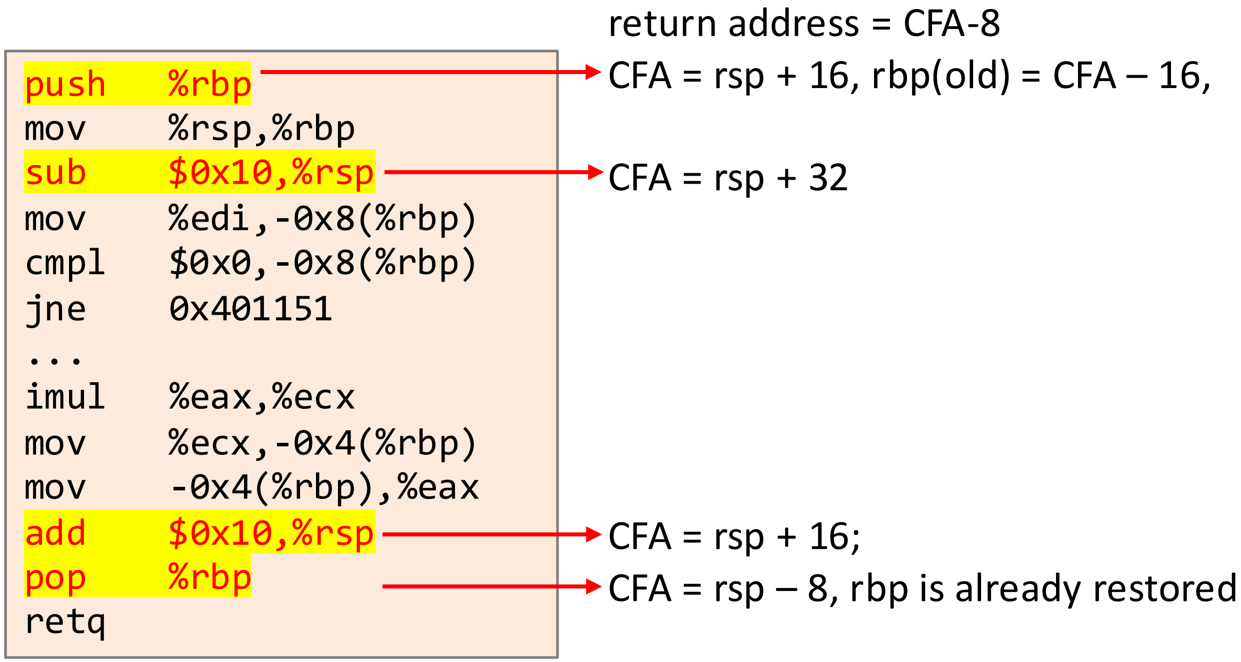
- Calculate the information required for recovering from each instruction during compilation.
- Such data format (DWARF) and mechanism is defined in the standard of ABI.
- The program unwinds the call stack iteratively.
- Different from the dynamic solution with setjmp.
    - more convenient, throw/try/catch is based on DWARF
    - more efficient

How Does DWARF Work?

- To recover the context of the caller, we should know whether callee-saved registers have been changed.
- Such callee-saved registers should be saved on the stack.
- Record the address of each callee-saved register.

Example

- Calculate the canonical frame address or CFA.
  - Find all instructions related to stack expansion/reduction.
- Record the address of callee-saved registers related to CFA.



```
push    %rbp
mov     %rsp,%rbp
sub     $0x10,%rsp
mov     %edi,-0x8(%rbp)
cmpl    $0x0,-0x8(%rbp)
jne     0x401151
...
imul    %eax,%ecx
mov     %ecx,-0x4(%rbp)
mov     -0x4(%rbp),%eax
add     $0x10,%rsp
pop     %rbp
retq
```

return address = CFA-8

push %rbp → CFA = rsp + 16, rbp(old) = CFA – 16,

sub $0x10,%rsp → CFA = rsp + 32

add $0x10,%rsp → CFA = rsp + 16;

pop %rbp → CFA = rsp – 8, rbp is already restored

Check DWARF Data with pyreadelf

- The data is saved in the eh_frame section of ELF files.

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

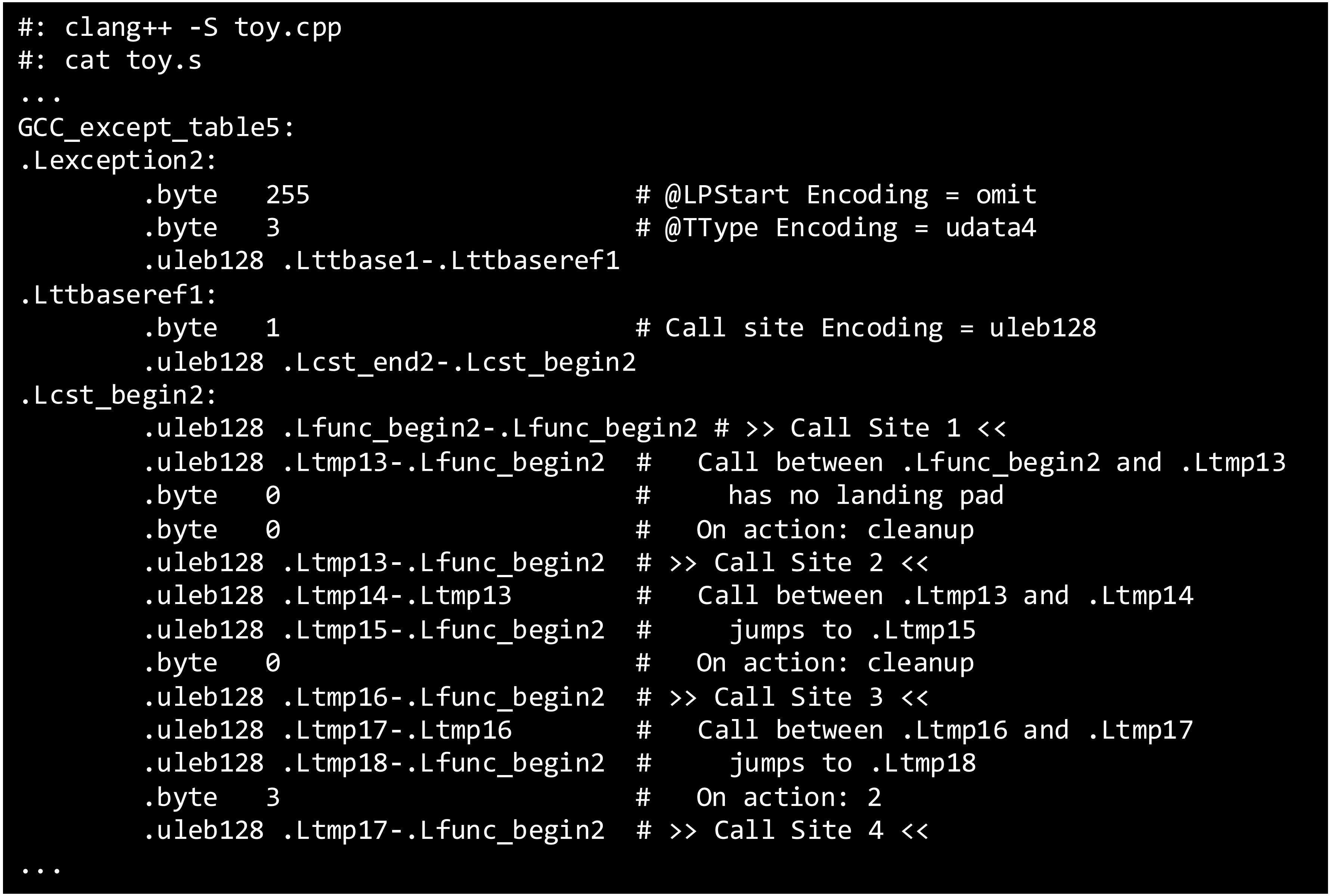
```
2690: endbr64
2694: push   %r15
2696: mov    %rsi,%rax
2699: push   %r14
269b: push   %r13
269d: push   %r12
269f: push   %rbp
26a0: push   %rbx
26a1: lea    0x4f94(%rip),%rbx
26a8: sub    $0x148,%rsp
26af: mov    %edi,0x2c(%rsp)
26b3: mov    (%rax),%rdi
…
27e7: sub    $0x8,%rsp
…
27fb: pushq  $0x0
…
2e96: pop    %rbx
2e97: pop    %rbp
2e98: pop    %r12
2e9a: pop    %r13
2e9c: pop    %r14
2e9e: pop    %r15
2ea0: retq
```

```
   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
00002e97 rsp+48    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002e98 rsp+40    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002e9a rsp+32    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002e9c rsp+24    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002e9e rsp+16    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002ea0 rsp+8     c-56  c-48  c-40  c-32  c-24  c-16  c-8
```

Usage of DWARF

- Debugging: developers can obtain the call stack with backtrace().
- Exception handling: require further information to determine the landing pad or language specific information (personality routine).
  - C++ try-throw-catch
  - Rust stack unwinding

Landing Pad: Check gcc_except_tables



```
#: clang++ -S toy.cpp
#: cat toy.s
...
GCC_except_table5:
.Lexception2:
        .byte   255                     # @LPStart Encoding = omit
        .byte   3                       # @TType Encoding = udata4
        .uleb128 .Lttbase1-.Lttbaseref1
.Lttbaseref1:
        .byte   1                       # Call site Encoding = uleb128
        .uleb128 .Lcst_end2-.Lcst_begin2
.Lcst_begin2:
        .uleb128 .Lfunc_begin2-.Lfunc_begin2 # >> Call Site 1 <<
        .uleb128 .Ltmp13-.Lfunc_begin2  #   Call between .Lfunc_begin2 and .Ltmp13
        .byte   0                       #     has no landing pad
        .byte   0                       #   On action: cleanup
        .uleb128 .Ltmp13-.Lfunc_begin2  # >> Call Site 2 <<
        .uleb128 .Ltmp14-.Ltmp13        #   Call between .Ltmp13 and .Ltmp14
        .uleb128 .Ltmp15-.Lfunc_begin2  #     jumps to .Ltmp15
        .byte   0                       #   On action: cleanup
        .uleb128 .Ltmp16-.Lfunc_begin2  # >> Call Site 3 <<
        .uleb128 .Ltmp17-.Ltmp16        #   Call between .Ltmp16 and .Ltmp17
        .uleb128 .Ltmp18-.Lfunc_begin2  #     jumps to .Ltmp18
        .byte   3                       #   On action: 2
        .uleb128 .Ltmp17-.Lfunc_begin2  # >> Call Site 4 <<
...
```

Combine Them Together

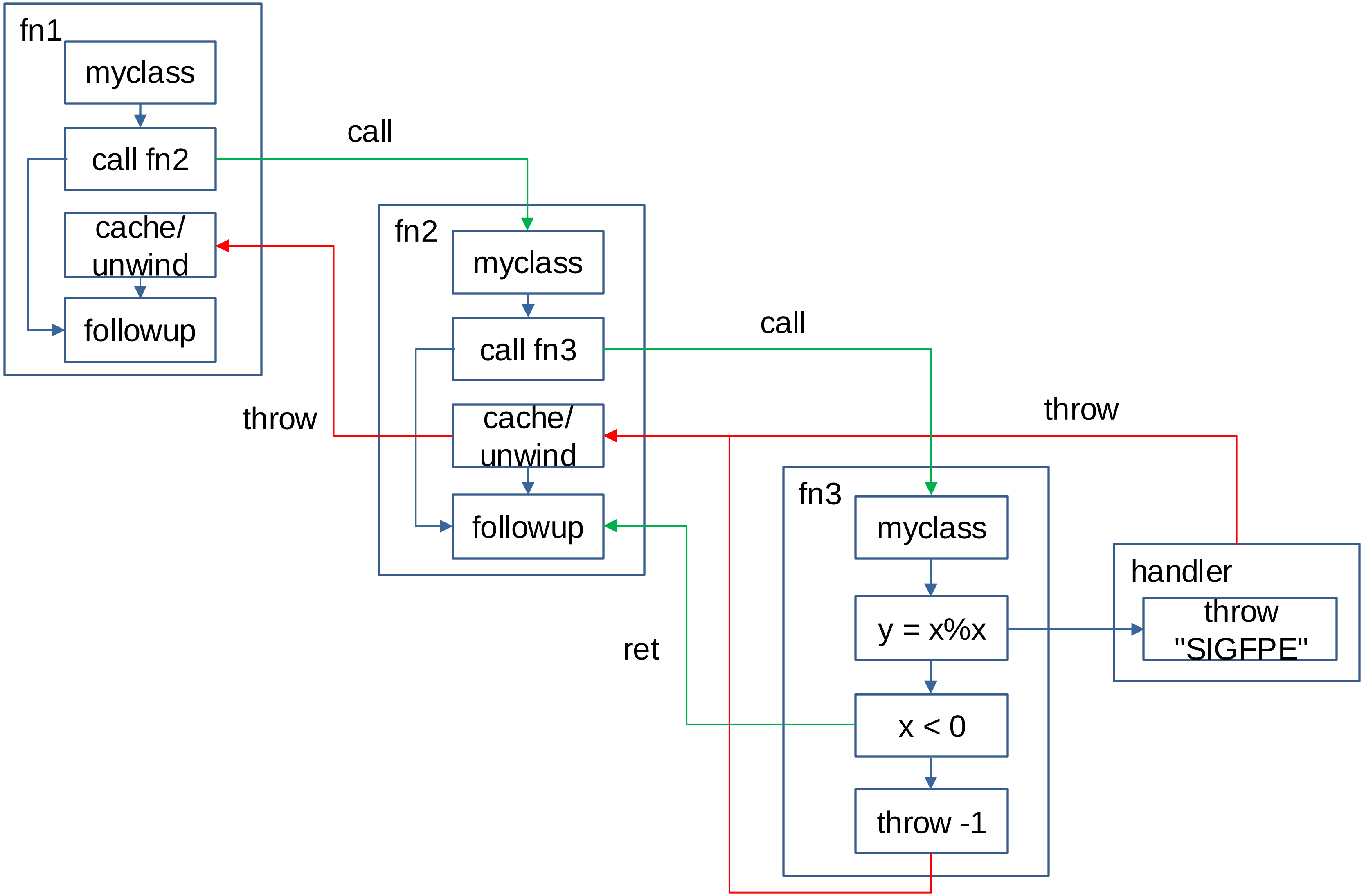
```
void fn3(int x) {
    MyClass c3 = MyClass{300};
    double y = x%x;
    if(x < 0) throw -1;
}
void fn2(int x) {
    MyClass c2 = MyClass{200};
    try{
        fn3(x);
    }catch (const int msg) {
        cout << "Land in fn2:"
             << msg << endl;
    }
}
void fn1(int x) {
    MyClass c1 = MyClass{100};
    try{
        fn2(x);
    }catch (const char* msg) {
         cout << "Land in fn1:"
              << msg << endl;
    }
}
```

```
void handler(int signal) {
    throw "SIGFPE Received!!!";
}

int main(int argc, char** argv) {
    signal(SIGFPE, handler);
    int x;
    scanf("%d", &x);
    fn1(x);
}
```

```
#: ./a.out
0
delete:200
Land in fn1: SIGFPE Received!!!
delete:100
#: ./a.out
-1
delete:300
Land in fn2:-1
delete:200
delete:100
```

Inter-procedural CFG



In-class Practice

- Referencing the shared pointer feature in C++, design and implement a shared pointer feature for C with the following APIs:
  - Create a new shared pointer from a raw pointer.
  - Clone a shared pointer, which increases the reference count.
  - Decrease the reference count when the variable goes out of scope, *e.g.*, based on cleanup attribute.
  - Free the pointer if the reference count becomes 0.