COMP 737011 - Memory Safety and Programming Language Design

Lecture 4: Memory Exhaustion

Xu, Hui

xuh@fudan.edu.cn

Outline

- 1. Stack Overflow and Handling
- 2. Heap Exhaustion and Mitigation

1. Stack Overflow and Handling

Warm Up

- Can you find a list to overflow the stack?

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

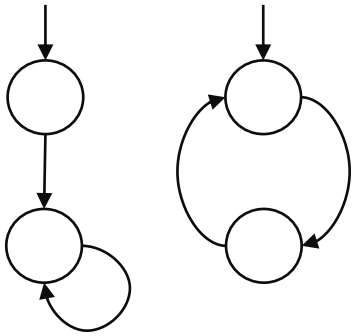
```
void process(struct List* l, int cnt){
    printf("%d\n", cnt);
    if(l->next != NULL)
        process(l->next, ++cnt);
}
```

- 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 larger 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);
```

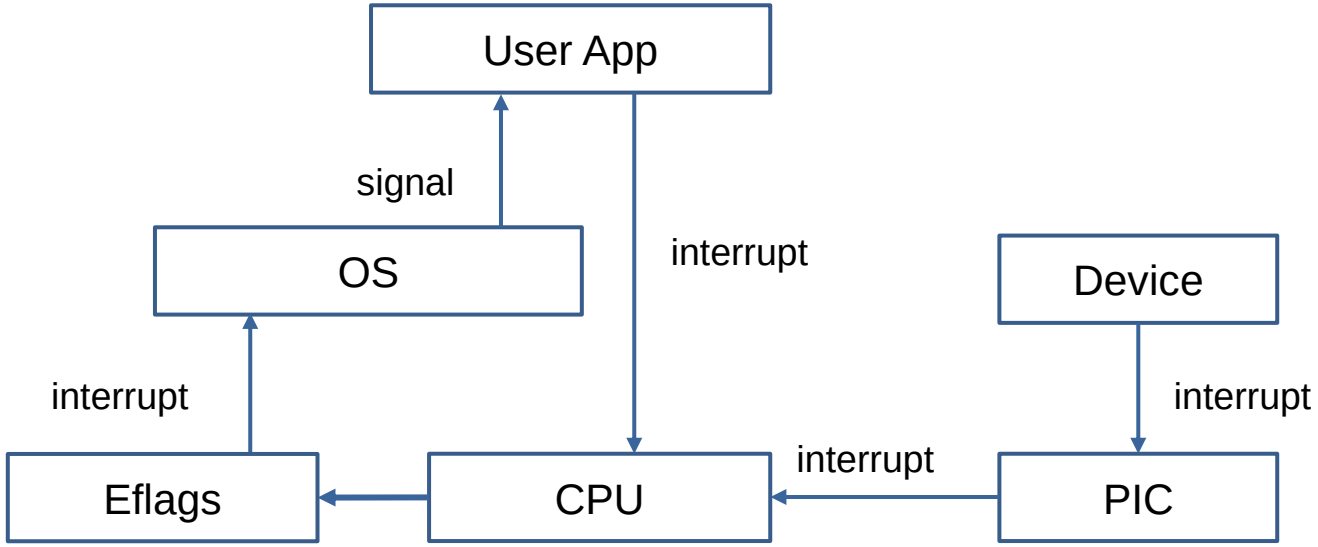
Set the stack size to 64 MB

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.

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 resume?
  - Find a landing pad
- How to set the required execution context for resuming?
  - Restore callee-saved registers: rbp、rsp、rbx、r12-r15
- Release acquired resources
  - e.g, heap, file descriptor

setjmp/longjmp

- setjmp(ctx)：
  - Backup registers and sets a recover point.
  - Return 0 if called directly, otherwise return a value if called by longjmp().
- longjmp(ctx, 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");
}
```

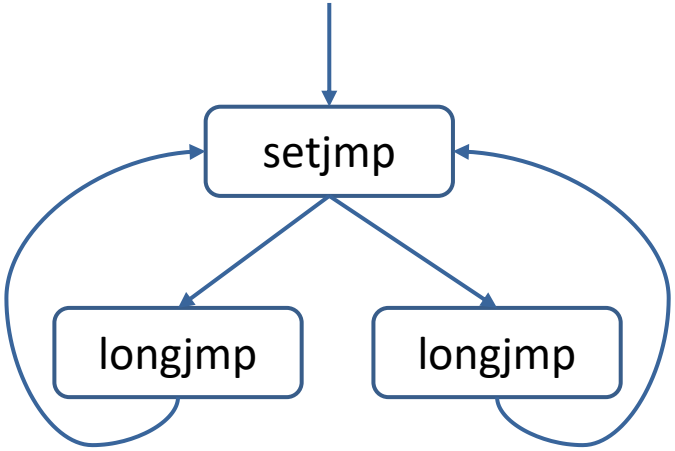
Full Code

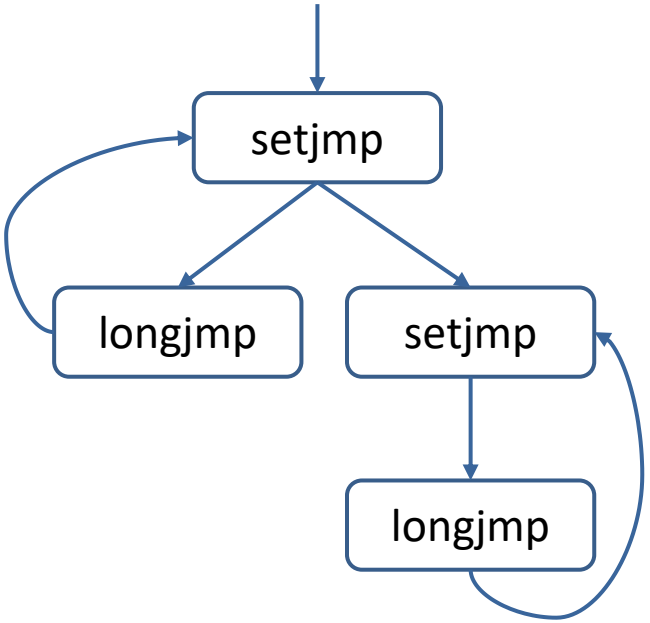
```
#define SIGSTACK_SIZE 1024
void sethandler(void (*handler)(int, siginfo_t *,void *)){
    static char stack[SIGSTKSZ];
    struct sigaction sa;
    stack_t ss = {
              .ss_size = SIGSTKSZ,
              .ss_sp = stack,
    };
    memset(&sa, 0, sizeof(sigaction));
    sigemptyset(&sa.sa_mask);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

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

Question

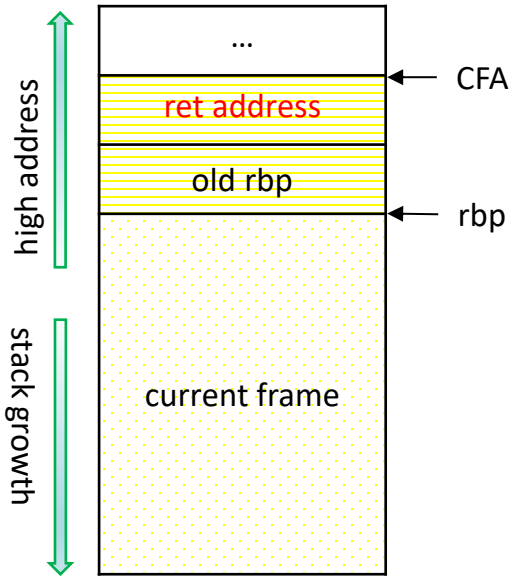
- How to support multiple setjmps/longjmps?





Stack Unwinding Problem

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



Debugging With Attributed Record Formats

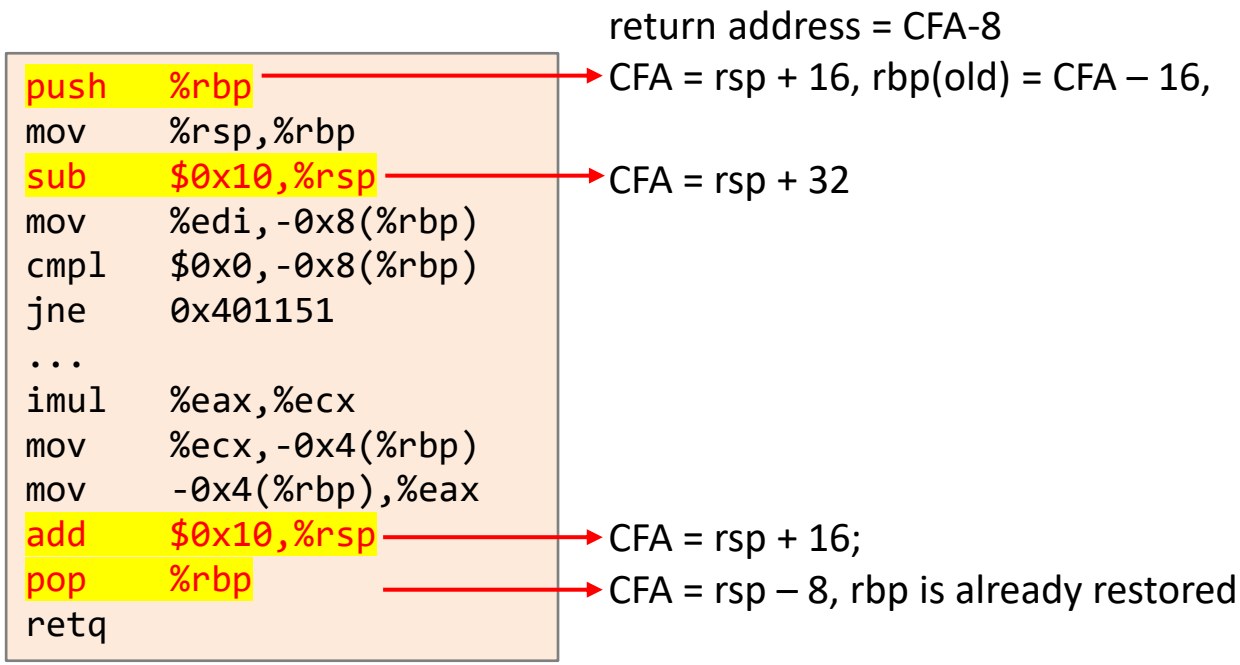
- Calculate the information required for recovering from each instruction during compilation.
- Such data format (DWARF) and mechanism is defined in the standard of ABI.
- The program unwinds the call stack iteratively.
- Different from the dynamic solution with setjmp.
  - It more convenient and more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

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 the CFA for each instruction related to stack expansion/reduction.
- Record the address of callee-saved registers related to CFA.



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

https://github.com/eliben/pyelftools

Usage of DWARF

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

Question

- For a multiple program, if one thread encounters stack overflow, will other threads be killed as well?

2. Heap Exhaustion and Mitigation

Heap Exhaustion and Handling

- Let malloc() return 0 if there is not enough memory.
- Overcommit: malloc() is "always" successful until accessing the memory.
  - Could be killed by the OS if the allocated memory is unavailable.
  - 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 or 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() success does not mean the physical memory is allocated.
  - The physical memory is allocated when being accessed.
- Linux has three options
  - 2: always check, never overcommit
  - 1: always overcommit, never check
  - 0: heuristic overcommit (this is the default)

```
#: sudo sysctl -w vm.overcommit_memory=1
```

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
#: ./heap
Killed
```

killed by the OS

```
#: sudo sysctl -w vm.overcommit_memory=2
#: ./heap
malloc failed
```

allocation failure

To Small to Fail: Example

```
#define SMALL_SIZE 1024L*20
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
#: ./smallheap
...
the 110060th malloc failed
```

```
#: sudo sysctl -w vm.overcommit_memory=1
#: ./smallheap
...
access the 2159257th memory chunk., done
Killed
```

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);
```

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);
}
```

```
...
mov     DWORD PTR [rbp-0x10],eax
call    0x401160 <free_buffer>
add     rsp,0x10
pop     rbp
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);
}
```

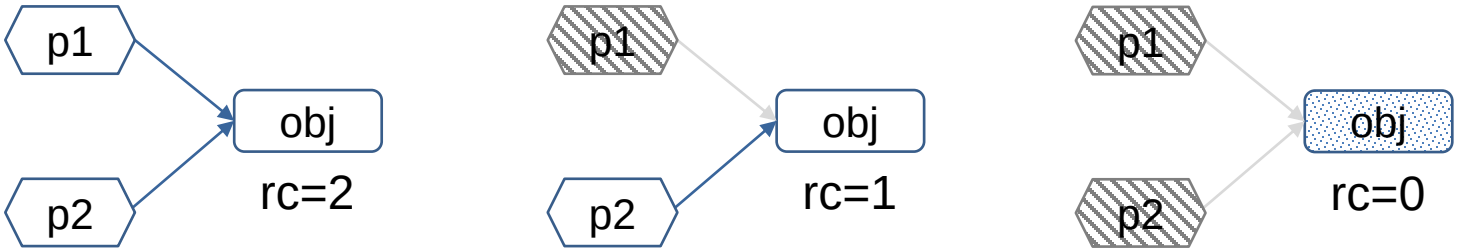
```
#:./a.out
delete:100
```

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.
- Two popular ways of automated memory reclaim.
  - Smart pointer
  - Garbage collection

Smart Pointers

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



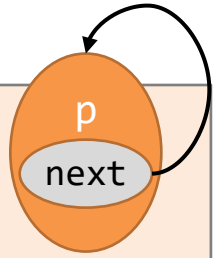
Smart Pointer in C++: shared_ptr

- 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;
}
```

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

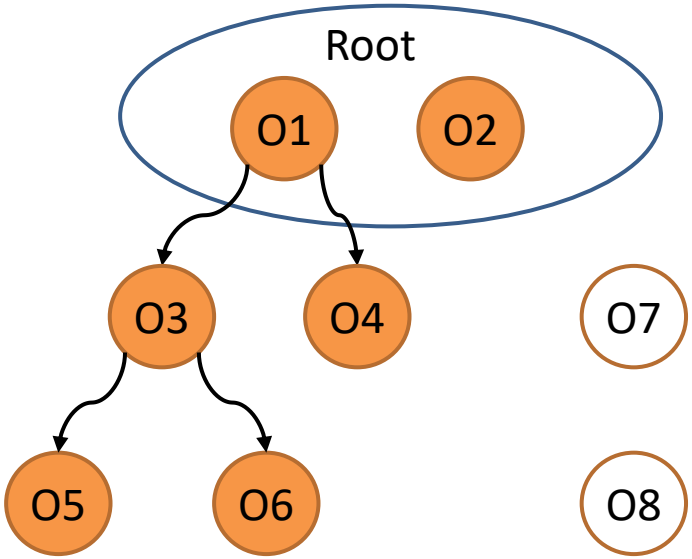
- Is it possible to introduce use-after-free bugs based on the auto delete or shared_ptr mechanism?
  - Without using delete

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

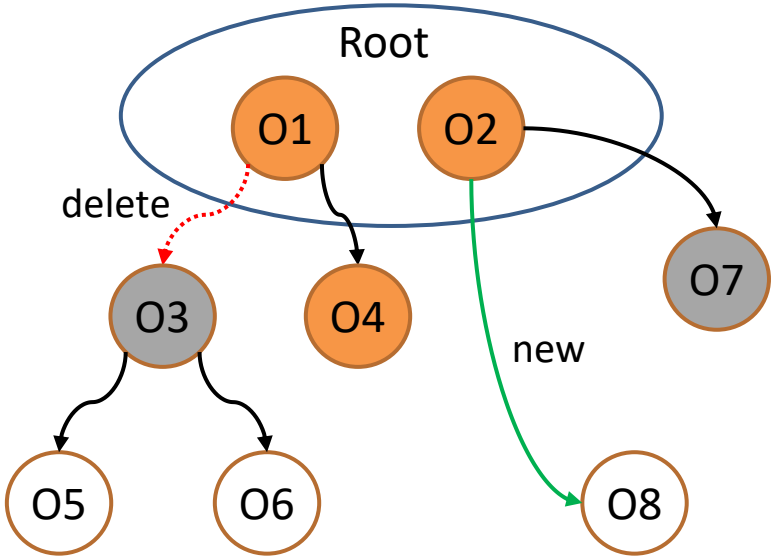
Reachability Analysis

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



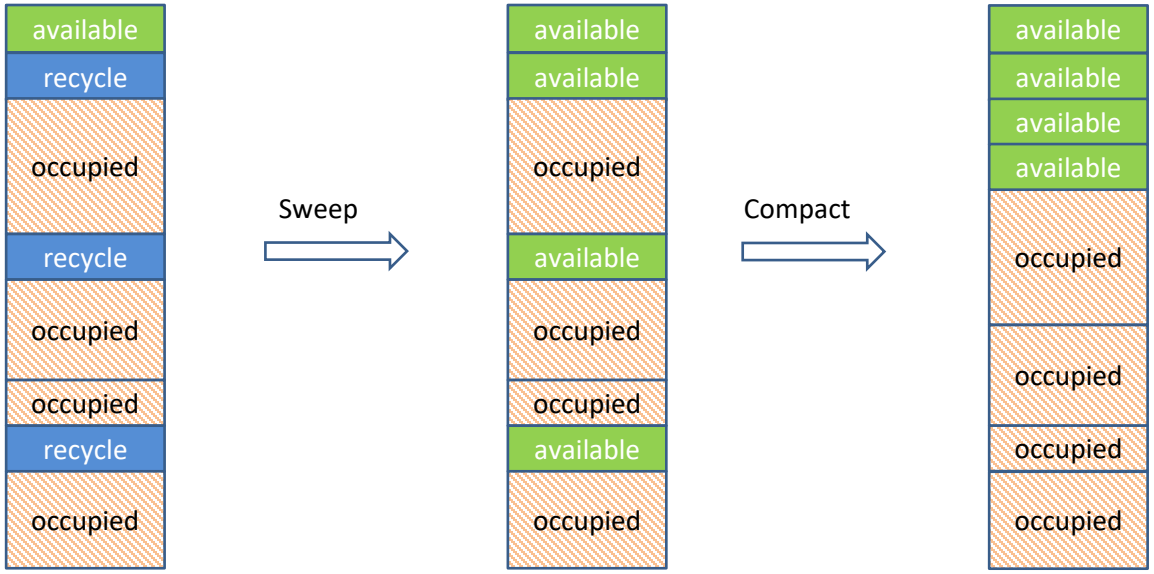
Incremental Analysis

- 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?



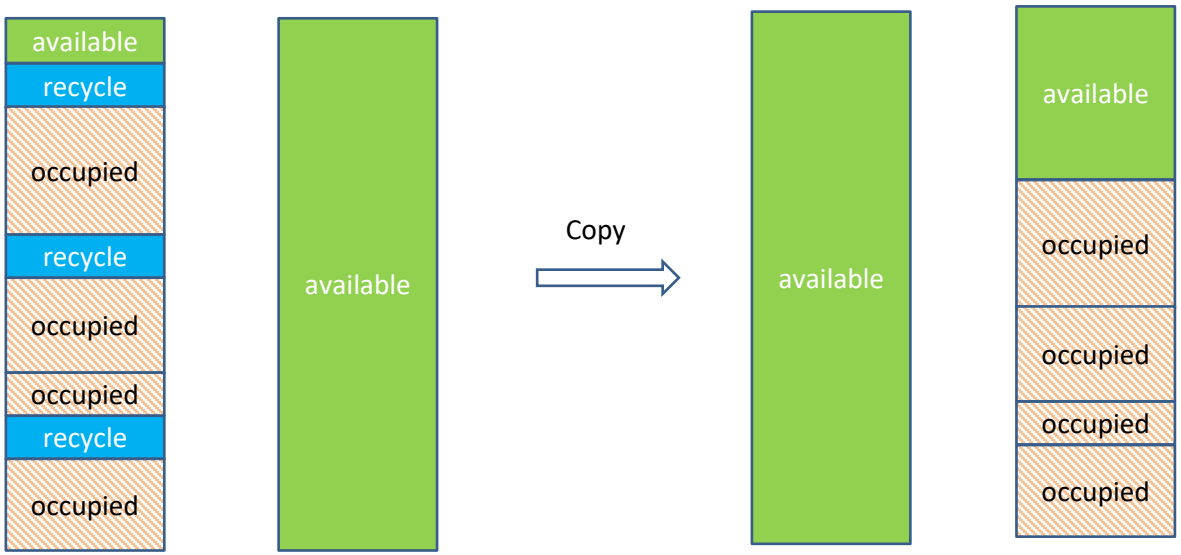
How to Recycle?

- For consecutive memory chunks (*e.g.,* program break).
- Mark-sweep: suffers fragmentation issue.
- Mark-compact: move all used units to one side.
  - Incur 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.

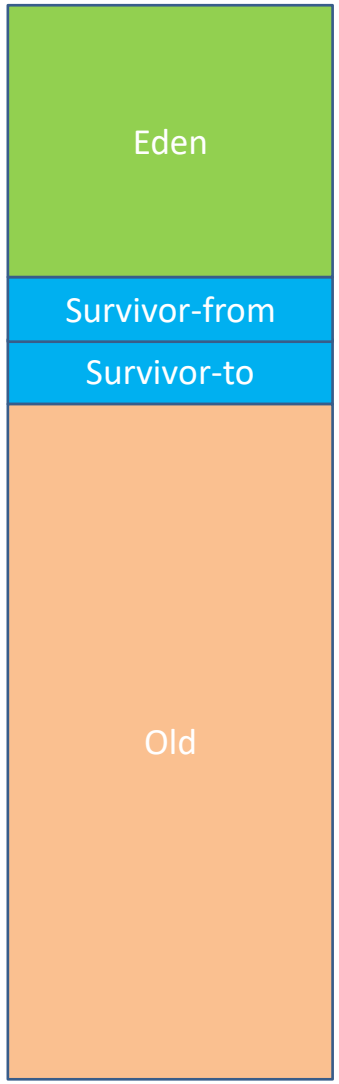


Observation

- Newly created objects tend to be recycled.
- The objects survived after several GC rounds have a higher 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" and "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 mitigate 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? e.g., 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

In-class Practice: Shared Pointer for C

- Recall the shared pointer feature in C++, and design a similar 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.