COMP 737011 - Memory Safety and Programming Language Design

Lecture 2: Memory Allocation

徐 辉

xuh@fudan.edu.cn

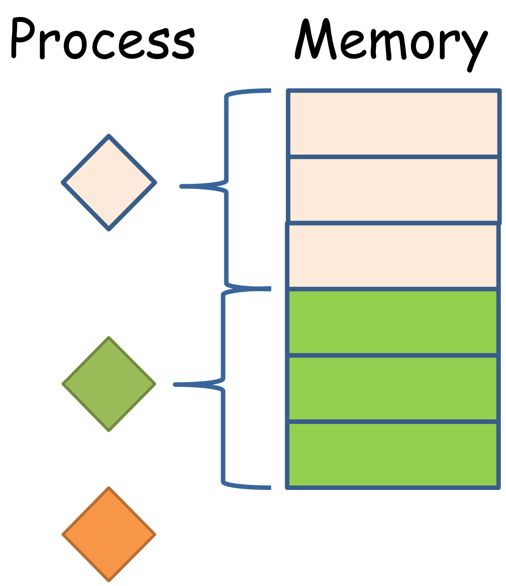
Outline

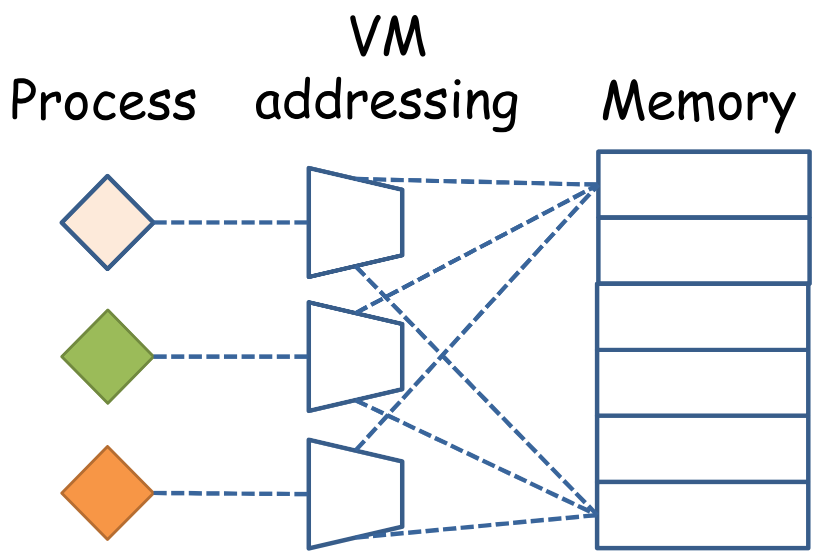
- 1. Virtual Memory System
- 2. Dynamic Memory Allocation
- 3. Auto Memory Reclaim

1. Virtual Memory System

How to Support Multi-tasking OS?

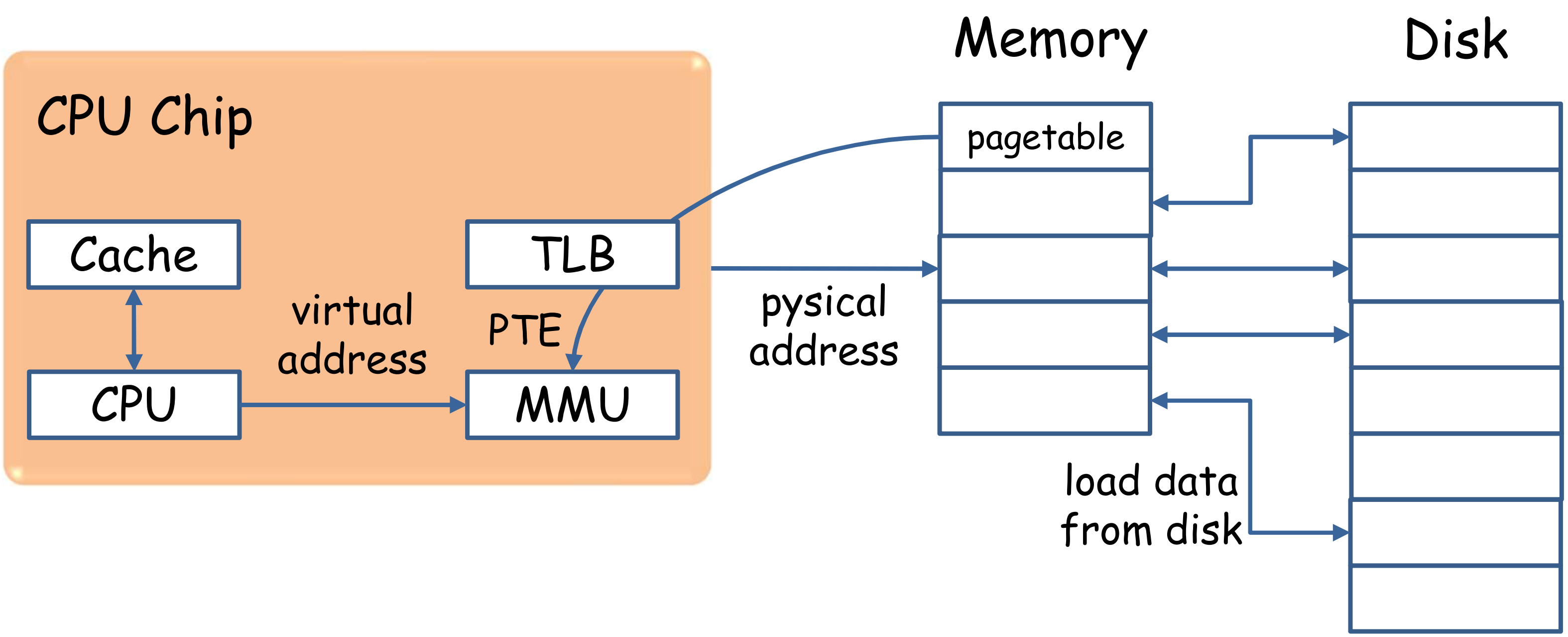
- All processes share the same memory space?
  - Each process uses an exclusive memory region
  - *e.g.*, unikernel or library OS
- Each process uses a distinct memory space
  - Virtual memory addressing
  - Both Linux and Windows use VM





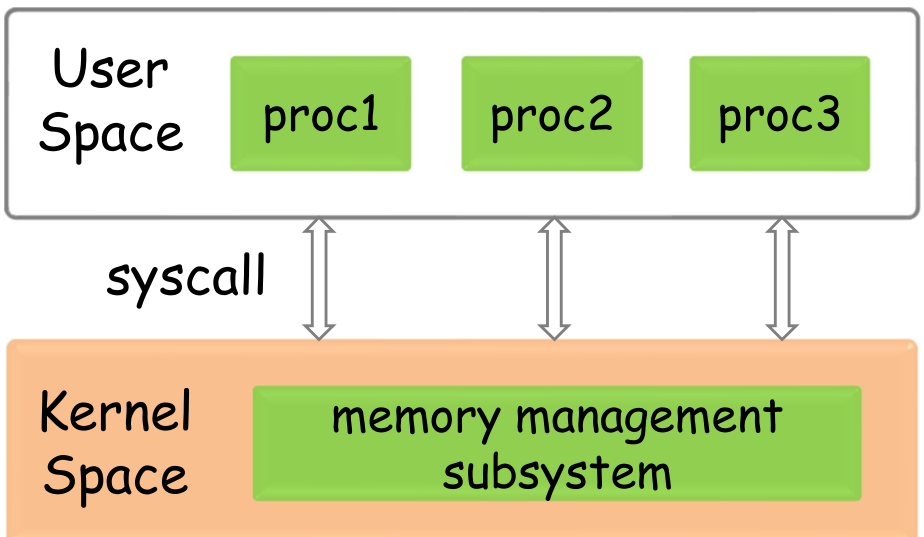
Virtual Memory Addressing

- MMU translates each virtual address to corresponding physical address by looking up the page table
- Cache page table entries with TLB
- Triggers page fault if the page is unavailable in DRAM



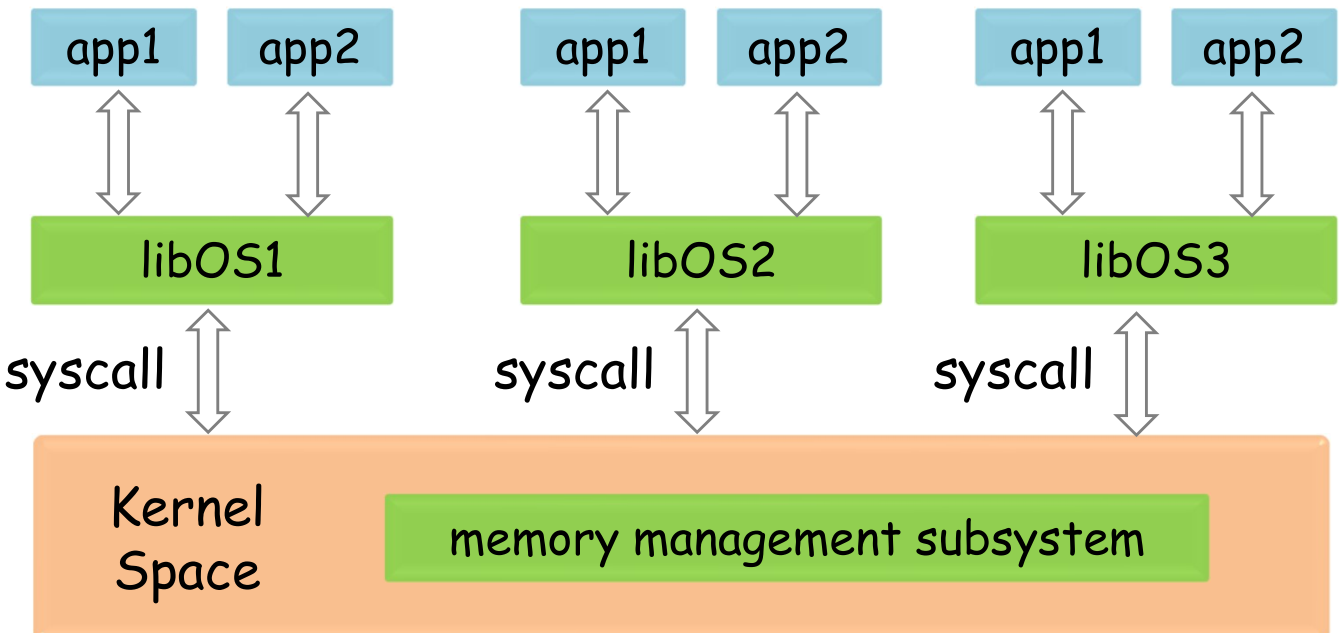
OS for VM

- Each process has a unique memory space
- Kernel responses for memory management
  - Map of memory space
  - Addressing
  - Handling page faults
- User space interacts with kernel via syscall



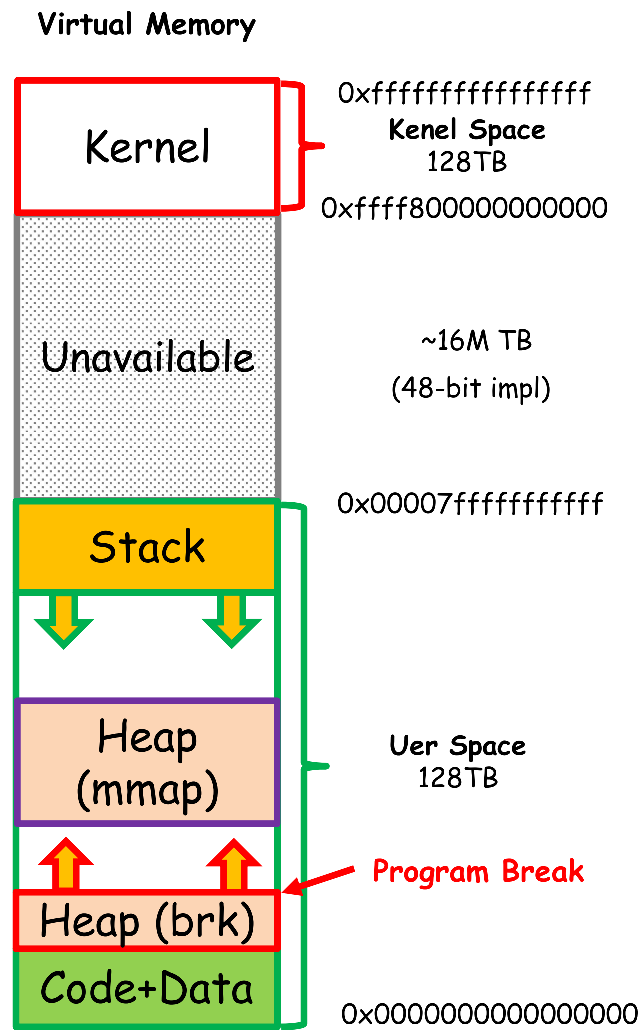
Unikernel

- Fault isolation for processes is difficult
  - Require instruction-level boudary checking
- Mainly used in clould as libOS
  - Each user runs a libOS instead of a Docker image or VM



Memory Allocation in Linux

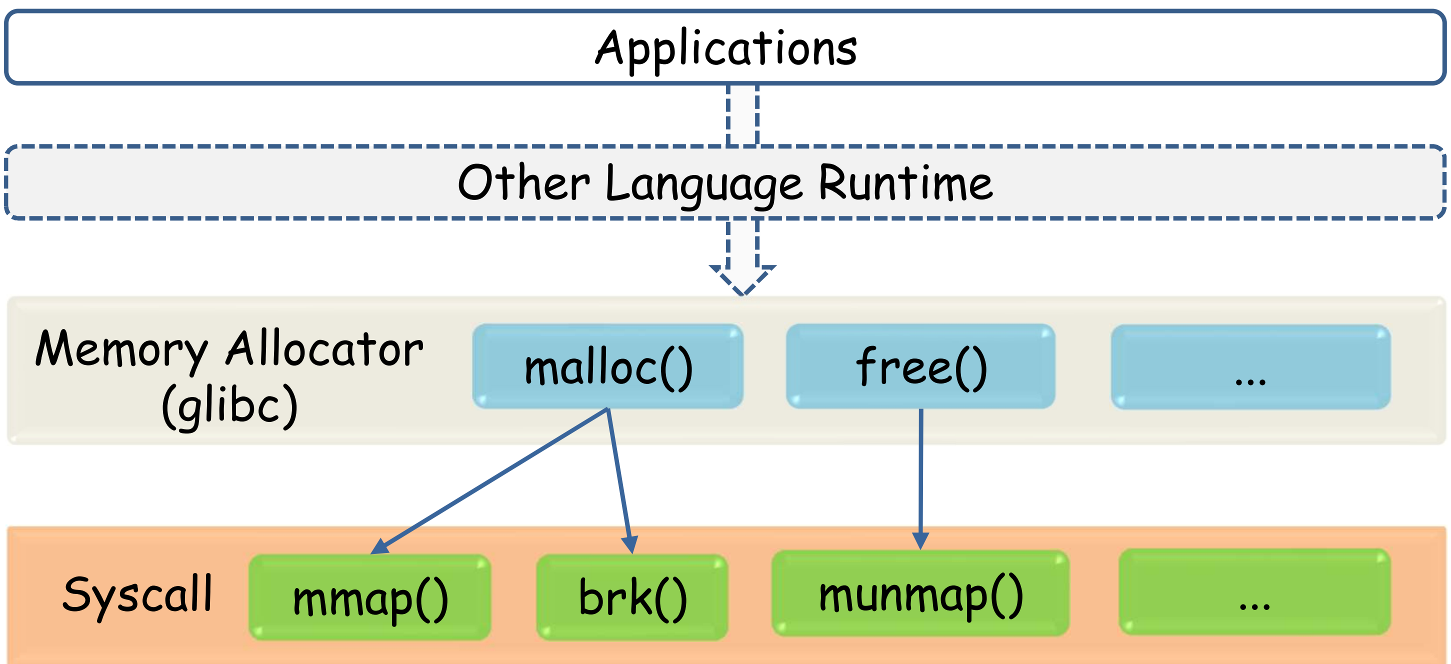
- Static allocation: static data
  - Compile-time constant
- Automatic allocation: stack
  - Each function has a stack frame
  - Multithreading program has multiple independent stacks
  - Compile-time constant
- Dynamic allocation: heap
  - More flexible



2. Dynamic Memory Allocation

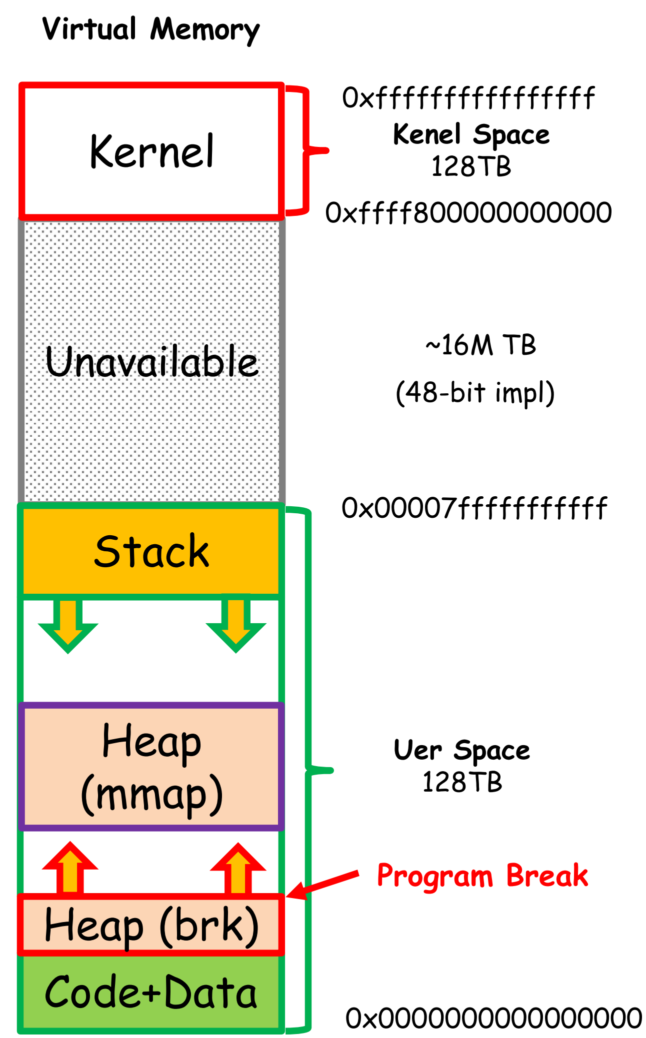
Overview of Memory Allocation APIs

- Applications use memory allocation APIs to acquire and release memory
- Memory allocator provides user-friendly APIs
  - e.g., malloc() and free() in glibc APIs
- Memory allocator invokes syscalls for achieving memory allocation.
  - e.g., brk() and mmap() in Linux



Heap Management in Linux

- Program break
  - Linux syscall brk()
  - For small-size memory trunks
  - Increase the brk pointer for memory allocation
  - Continuous address space
- Memory mapping:
  - Linux syscall mmap()
  - For file mapping and memory of large size (usually 256 KB)
  - Freed via munmap()



brk()/sbrk()/mmap()

```
int brk(void* end_data_segment); //Linux syscall
//change brk pointer to the specified addr value

void *sbrk(intptr_t increment); //a library API in Linux
//increase the brk pointer according to the increment

void *mmap(void *addr, // starting address
           size_t length, // byte
           int prot, // memory protection: read/write/exec
           int flags, // visibility: shared or private
           int fd, // file descriptor
           off_t offset); // offset of the file

int munmap(void *addr, size_t length);
```

https://man7.org/linux/man-pages/man2/brk.2.html
https://man7.org/linux/man-pages/man2/mmap.2.html
https://man7.org/linux/man-pages/man2/syscalls.2.html

glibc APIs for Heap Management

- malloc(size_t n)
  - Allocate a new memory space of size n
  - The memory is not cleared
  - Return the address pointer
- free(void * p)
  - Release the memory space pointed by p
  - Do not return to the system directly (for brk)
  - What would happen if p is null or already freed?
- calloc(size_t nmemb, size_t size)
  - Allocate an array of nmemb * size byte
  - The memory is set to zero
- realloc(void *p, size_t size)
  - Resize the memory block pointed by p to size bytes

Design Challanges for Allocator

- Each syscall costs nearly a hundred CPU cycles
  - =>An allocator should not frequently invoke syscalls
- Heap data are not compact
  - =>Reducing brk pointer for free is less effective
- How to manage and reuse freed memory chunks?

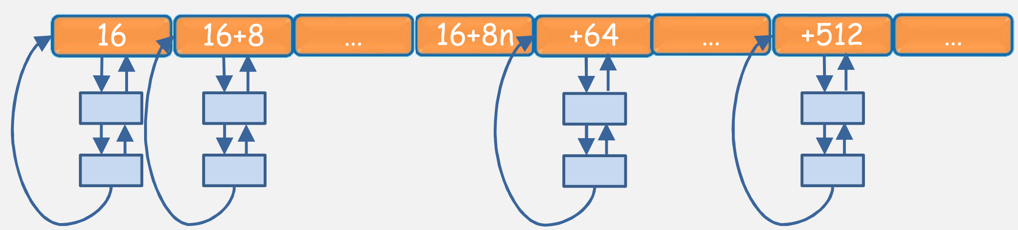
Basic Idea: Doug Lea's Allocator (dlmalloc)

- Freed memory chunks are managed as bins
- Each bin is a double-lined list of freed chunks of "fixed" size
- malloc() finds the corresponding bin for allocation
  - index is computed by logical shift: e.g., size >> 3
  - first-in-first-out: each chunk with the same opptunity to be consolidated

```
Bins for sizes < 512 bytes contain chunks of all the same size,
spaced 8 bytes apart. Larger bins are approximately logarithmically
spaced:

    64 bins of size       8
    32 bins of size      64
    16 bins of size     512
     8 bins of size    4096
     4 bins of size   32768
     2 bins of size  262144
     1 bin  of size what's left
    The bins top out around 1MB because we expect to service large
    requests via mmap.
```

small bins

large bins



https://github.com/ennorehling/dlmalloc/blob/master/malloc.c

Structure of Chunks: Boundary Tag

- Sizes of free chunks are stored both in the front of each chunk and at the end.
- This makes consolidating fragmented chunks into bigger chunks very fast.

prev_size	
size	PREV_INUSE
forward pointer	
backward pointer	
unused space	
size	

free trunk

prev_size	
size	PREV_INUSE
user data	
size	

allocated trunk

Fastbins and Unsorted Bins

- Design consideration for consolidation is expensive
- Fastbins: light-weight bins in single-linked list
  - cannot be coalesced with adjacent chunks automatically
- Unsorted bins: free chunks are first put into unsorted bins

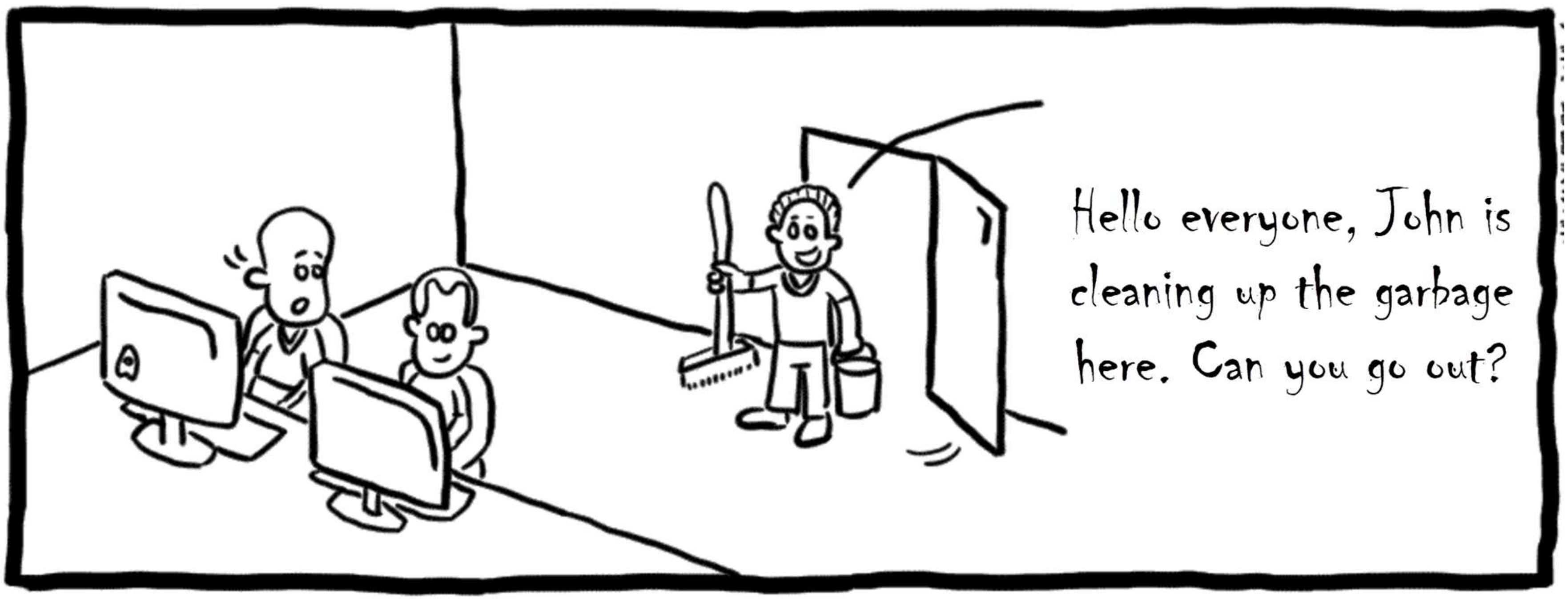
	list	**coalesce**	**data**
Fast bin	single-linked	no	small
Regular bin	double-linked	may	could be large

https://github.com/ennorehling/dlmalloc/blob/master/malloc.c

More Allocators

- ptmalloc (pthreads malloc): used in glibc
  - a fork of dlmalloc with threading-related improvements
  - https://sourceware.org/glibc/wiki/MallocInternals
- tcmalloc (thread-caching malloc) by Google
  - https://google.github.io/tcmalloc/
- jemalloc
  - http://jemalloc.net/

3. Auto Memory Reclaim

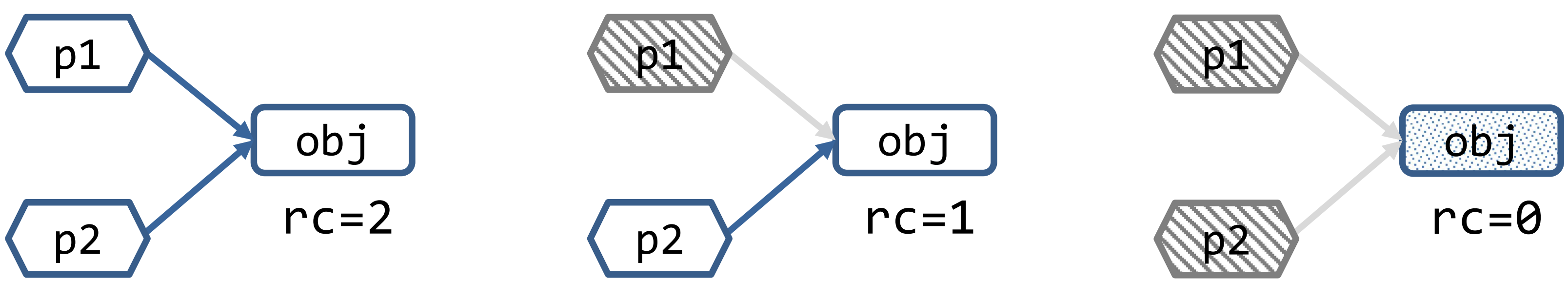


Auto Reclaim Challenge

- Memory units 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

Mechanisms for Auto Reclaim

- Mainly based on dynamic analysis
- Smart pointer
  - Dynamically track the number of references with a reference counter
  - Reclaim the memory once no variable owns it
- Garbage collection
  - Periodically check object references



Smart Pointer: unique_ptr

- Object is uniquely ownd by one pointer
- User can transfer ownership through move()

```
int main() {
    unique_ptr<MyClass> up1(new MyClass(2));
    //unique_ptr<MyClass> up2 = up1; //compilation error
    unique_ptr<MyClass> up2 = move(up1);
    //cout << up1->val << endl; //segmentation fault
    cout << up2->val << endl;
}
```

Smart Pointer: shared_ptr

- Object are shared among pointers with a reference counter.
- Destroyed when the last remaining shared_ptr owning the object is destroyed or reassigned.

```
int main() {
    shared_ptr<MyClass> sp1(new MyClass(2));
    shared_ptr<MyClass> sp2 = p1;
}
```

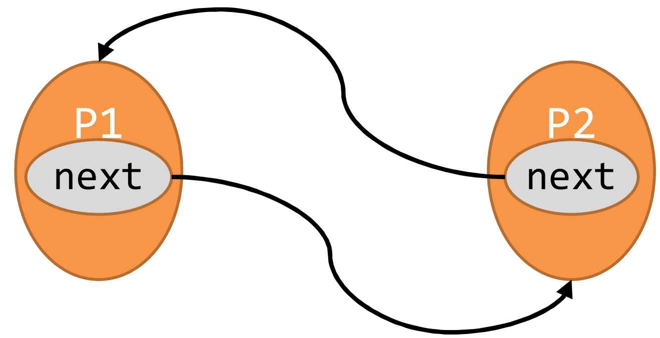
Guess the Output?

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

int main() {
    MyClass* p0 = new MyClass(1);
    {
        shared_ptr<MyClass> p1(new MyClass(2));
        shared_ptr<MyClass> p2 = p1;
        shared_ptr<MyClass> p3(p0);
    }
    cout << p0->val << endl;
}
```

```
./a.out
delete obj:1
delete obj:2
0
```

Smart Pointer: weak_ptr

- Problem of shared_ptr
  - reference cycles
  - both p1 and p2 are leaked
- weak_ptr:
  - do up update the reference counter



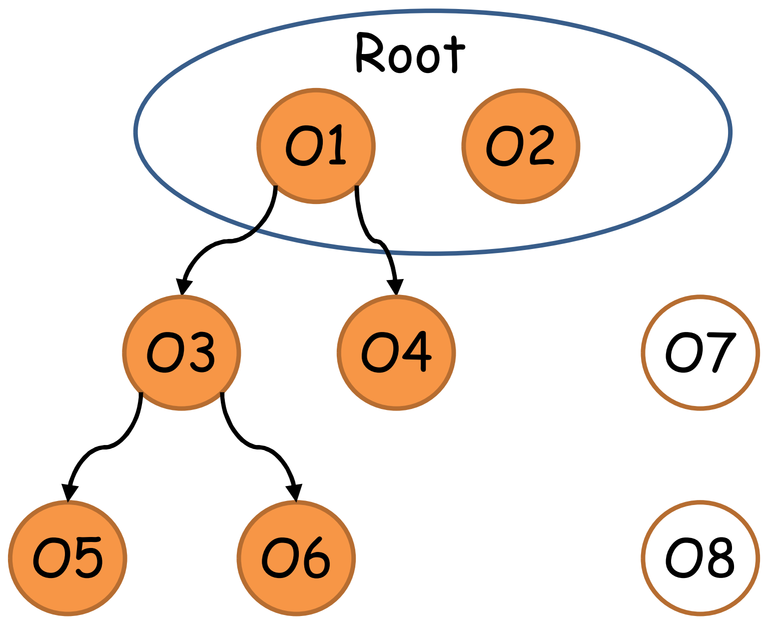

```
class MyList{
public:
    int val;
    //shared_ptr<MyList> next;
    weak_ptr<MyList> next;
    ~MyList() { cout << "delete obj:"<< val << endl; }
};

int main() {
    shared_ptr<MyList> p1 = make_shared<MyList>();
    shared_ptr<MyList> p2 = make_shared<MyList>();
    p1->val = 1;
    p2->val = 2;
    p1->next = p2;
    p2->next = p1;
}
```

Garbage Collection

- When should the GC be triggered?
- What kind of objects should be recycled?
  - Rechability analysis
- How to recycle?
  - Slowdown due to intensive GC operation
  - Memory fragmentation issue

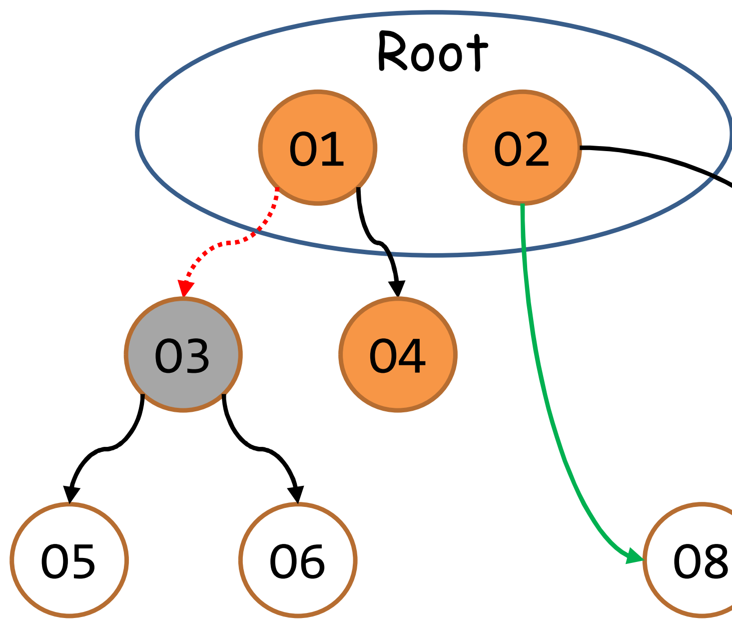
Reachability Analysis

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



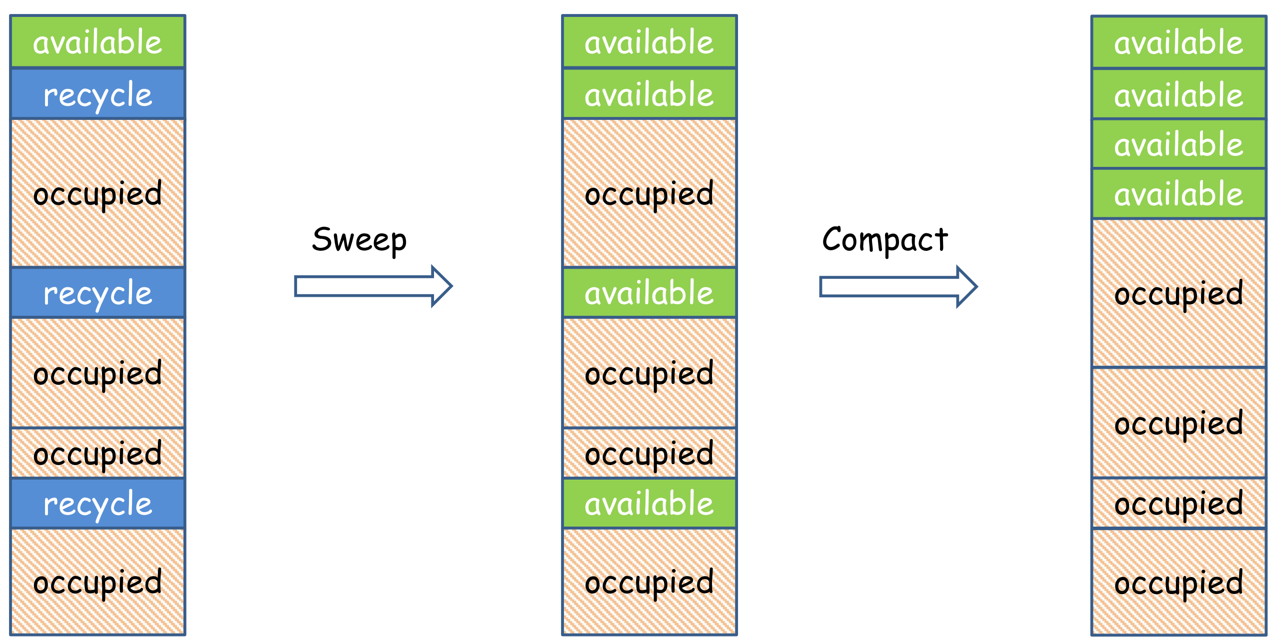
Incremental Analysis

- 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?



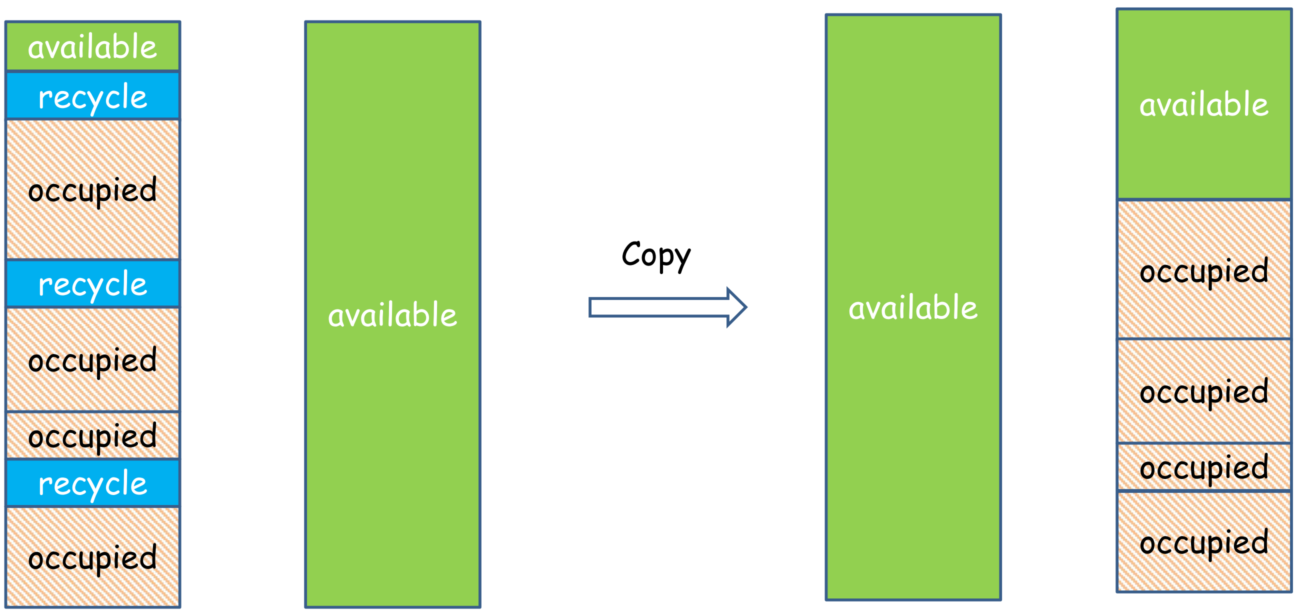
How to Recycle?

- 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

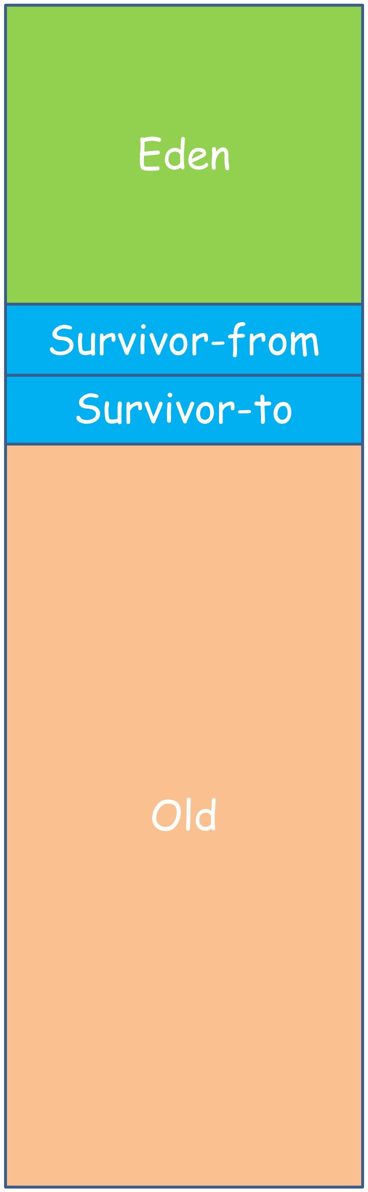


Observation

- 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?

- You may refer the following tutorials
  - https://maplant.com/gc.html
  - BoehmGC: https://www.hboehm.info/gc/#details

More Reference

- https://sourceware.org/glibc/wiki/MallocInternals
- https://cw.fel.cvut.cz/old/_media/courses/a4m33pal/04_dynamic_memory_v6.pdf
- https://heap-exploitation.dhavalkapil.com/diving_into_glibc_heap/bins_chunks