COMP 737011 - Memory Safety and Programming Language Design

Lecture 2: Allocator Design

Hui Xu
xuh@fudan.edu.cn

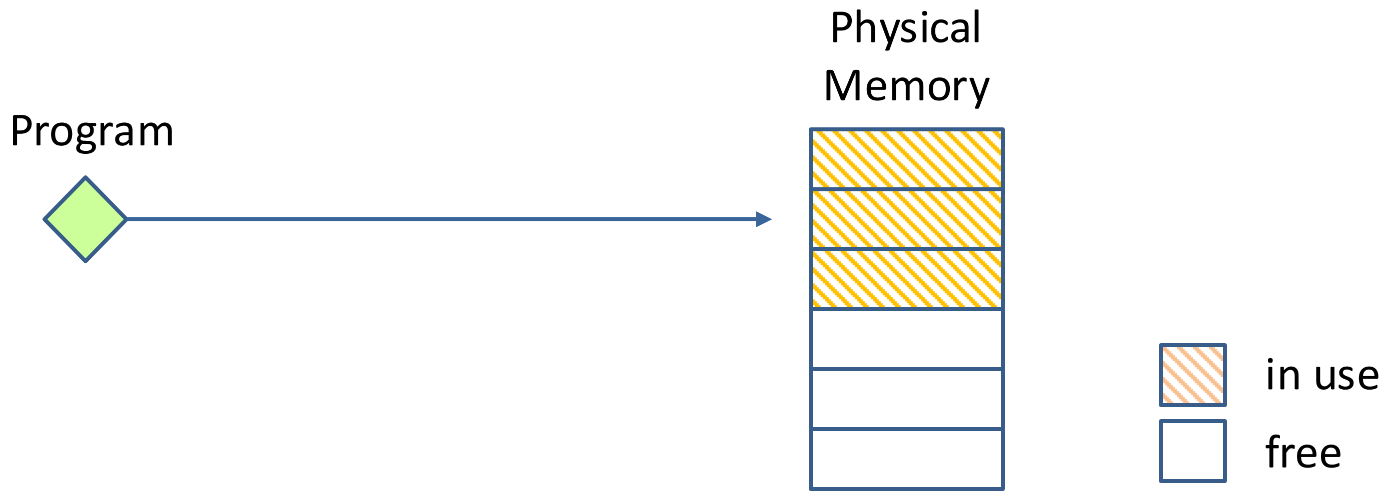
Outline

- 1. Memory Management Overview
- 2. Kernel Space Allocator
- 3. User Space Allocator

1. Memory Management Overview

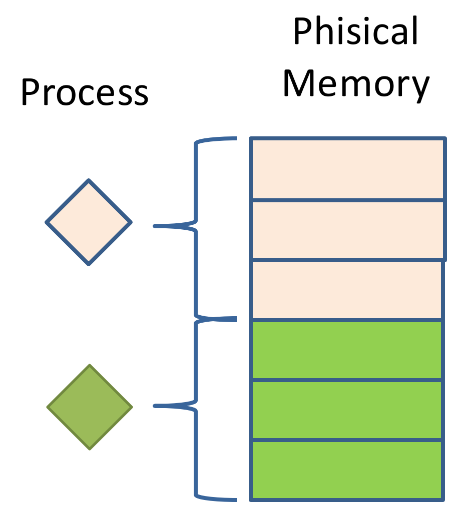
Memory Access

- Consider the scenarios: BIOS, embedded systems, OS kernel
- Access memory via physical addresses
- Direct mapping: physical addr = virtual addr + offset



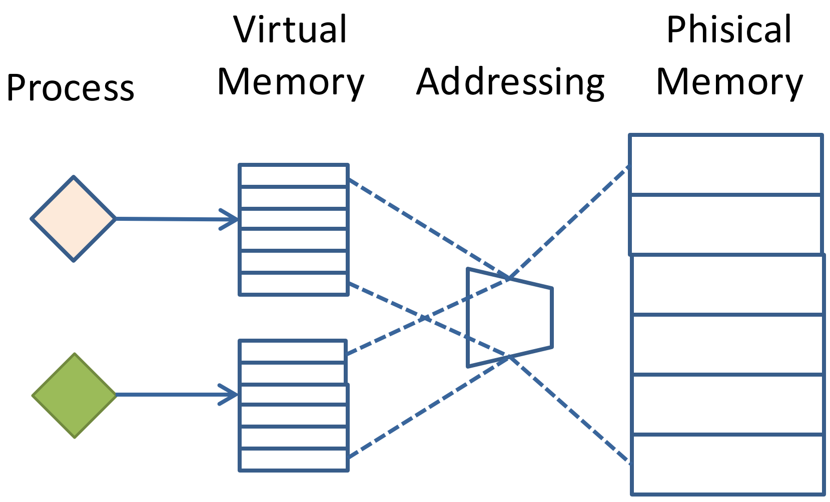
How to Support Multi-tasking? Unikernel

- All processes share the same memory space.
  - Each process uses an exclusive memory region
- Fault isolation for processes is difficult.
  - Require instruction-level boudary checking
- Mainly used in embedded systems



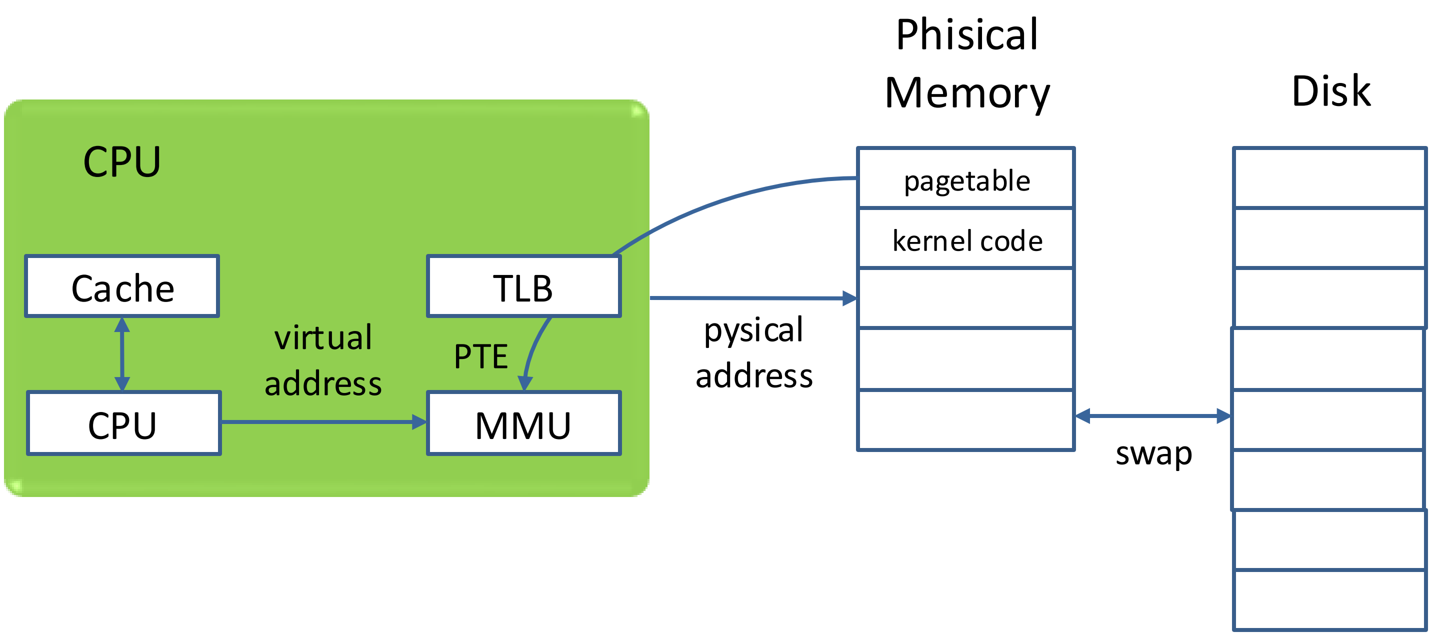
How to Support Multi-tasking? Virtual Memory

- Each process uses a distinct memory space.
- Used in both Linux, Mac, Windows.
- Achieved through virtual memory addressing.



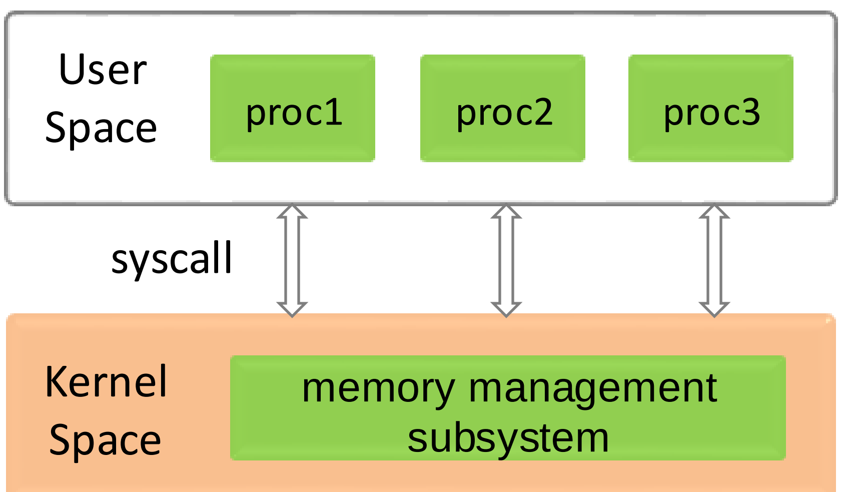
Virtual Memory Addressing

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



OS for VM

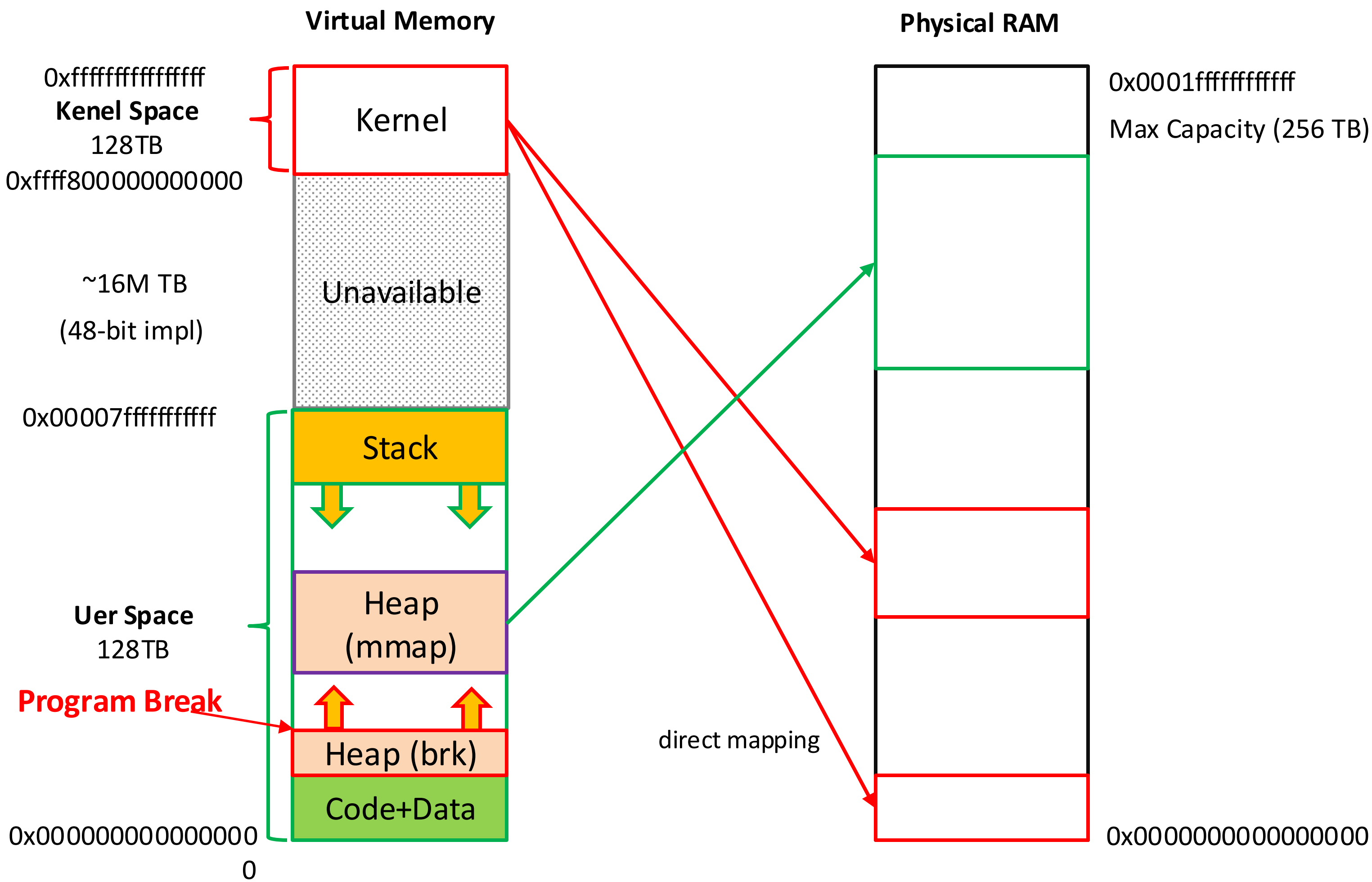
- Each process has a unique memory space.
- Kernel responses for memory management.
  - Map of memory space
  - Addressing
  - Handling page faults
- User space interacts with kernel via syscall.



Memory Allocation in Linux

- Static allocation: code and 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

Virtual Address vs Phsical RAM



https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt

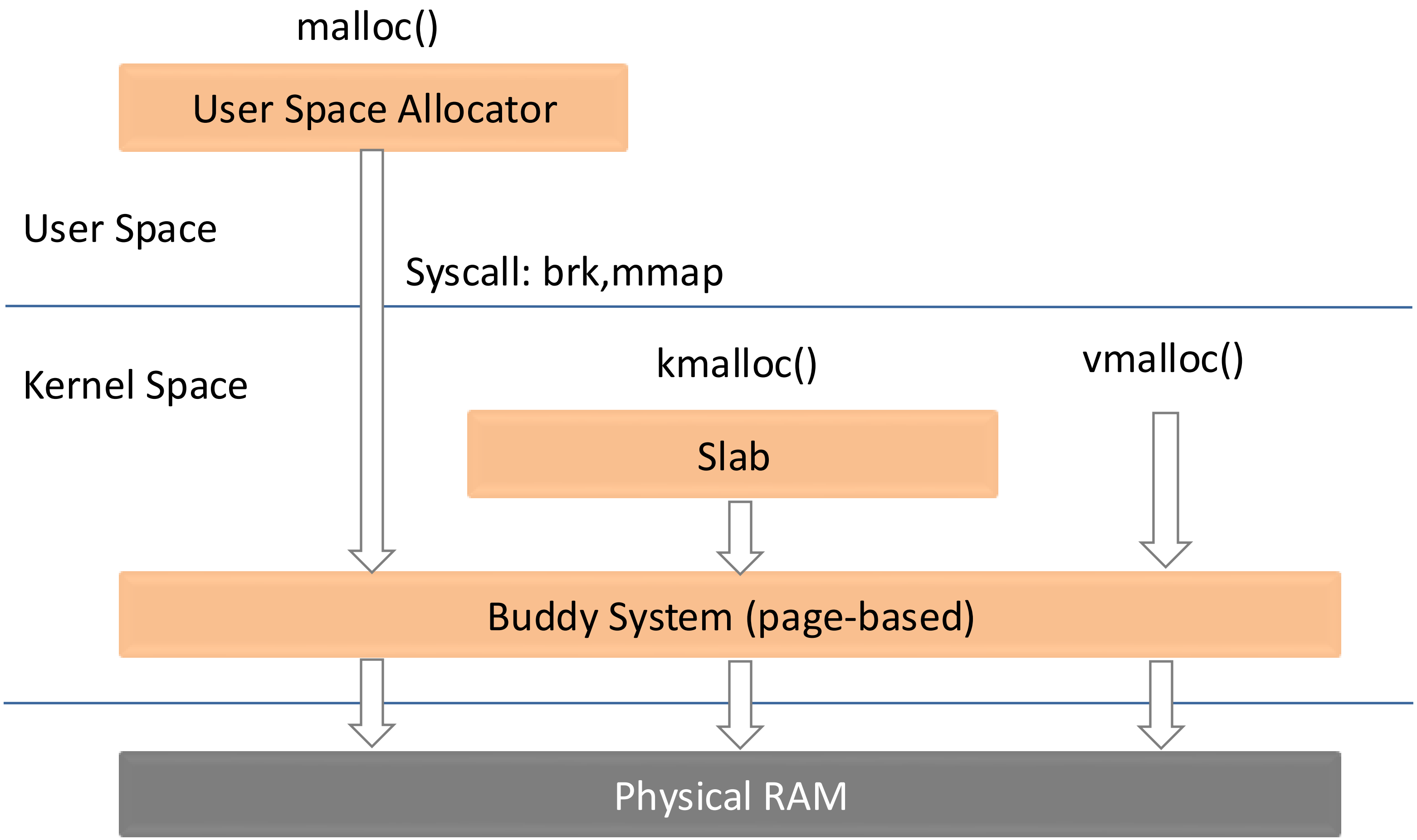
View Virtual Memory via /proc/pid/maps

```
#: cat /proc/213694/maps
00400000-00401000 r--p 00000000 103:02 10223789                ./hello
00401000-00402000 r-xp 00001000 103:02 10223789                ./hello
00402000-00403000 r--p 00002000 103:02 10223789                ./hello
00403000-00404000 r--p 00002000 103:02 10223789                ./hello
00404000-00405000 rw-p 00003000 103:02 10223789                ./hello
01b4f000-01b70000 rw-p 00000000 00:00 0                        [heap]
7f6c23b29000-7f6c23b4b000 r--p 00000000 103:02 9963653         libc-2.31.so
7f6c23b4b000-7f6c23cc3000 r-xp 00022000 103:02 9963653         libc-2.31.so
7f6c23cc3000-7f6c23d11000 r--p 0019a000 103:02 9963653         libc-2.31.so
7f6c23d11000-7f6c23d15000 r--p 001e7000 103:02 9963653         libc-2.31.so
7f6c23d15000-7f6c23d17000 rw-p 001eb000 103:02 9963653         libc-2.31.so
7f6c23d17000-7f6c23d1d000 rw-p 00000000 00:00 0
7f6c23d30000-7f6c23d31000 r--p 00000000 103:02 9963648         ld-2.31.so
7f6c23d31000-7f6c23d54000 r-xp 00001000 103:02 9963648         ld-2.31.so
7f6c23d54000-7f6c23d5c000 r--p 00024000 103:02 9963648         ld-2.31.so
7f6c23d5d000-7f6c23d5e000 r--p 0002c000 103:02 9963648         ld-2.31.so
7f6c23d5e000-7f6c23d5f000 rw-p 0002d000 103:02 9963648         ld-2.31.so
7f6c23d5f000-7f6c23d60000 rw-p 00000000 00:00 0
7ffdf802d000-7ffdf804e000 rw-p 00000000 00:00 0                [stack]
7ffdf80c7000-7ffdf80cb000 r--p 00000000 00:00 0                [vvar]
7ffdf80cb000-7ffdf80cd000 r-xp 00000000 00:00 0                [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0        [vsyscall]
```

View Physical Memory via /proc/iomem

```
#: cat /proc/iomem
00000000-00000fff : Reserved
00001000-0009efff : System RAM
0009f000-000fffff : Reserved
  00000000-00000000 : PCI Bus 0000:00
  00000000-00000000 : PCI Bus 0000:00
  00000000-00000000 : PCI Bus 0000:00
  00000000-00000000 : PCI Bus 0000:00
  00000000-00000000 : PCI Bus 0000:00
  00000000-00000000 : PCI Bus 0000:00
  00000000-00000000 : PCI Bus 0000:00
  00000000-00000000 : PCI Bus 0000:00
  000a0000-000dffff : PCI Bus 0000:00
    000c0000-000dffff : 0000:00:02.0
  00000000-00000000 : PCI Bus 0000:00
  00000000-00000000 : PCI Bus 0000:00
  000e4000-000effff : PCI Bus 0000:00
  000f0000-000fffff : System ROM
00100000-3fffffff : System RAM
40000000-403fffff : Reserved
  40000000-403fffff : pnp 00:00
40400000-5a339017 : System RAM
5a339018-5a342a57 : System RAM
5a342a58-62588fff : System RAM
62589000-62589fff : ACPI Non-volatile Storage
...
```

Memory Management Framework



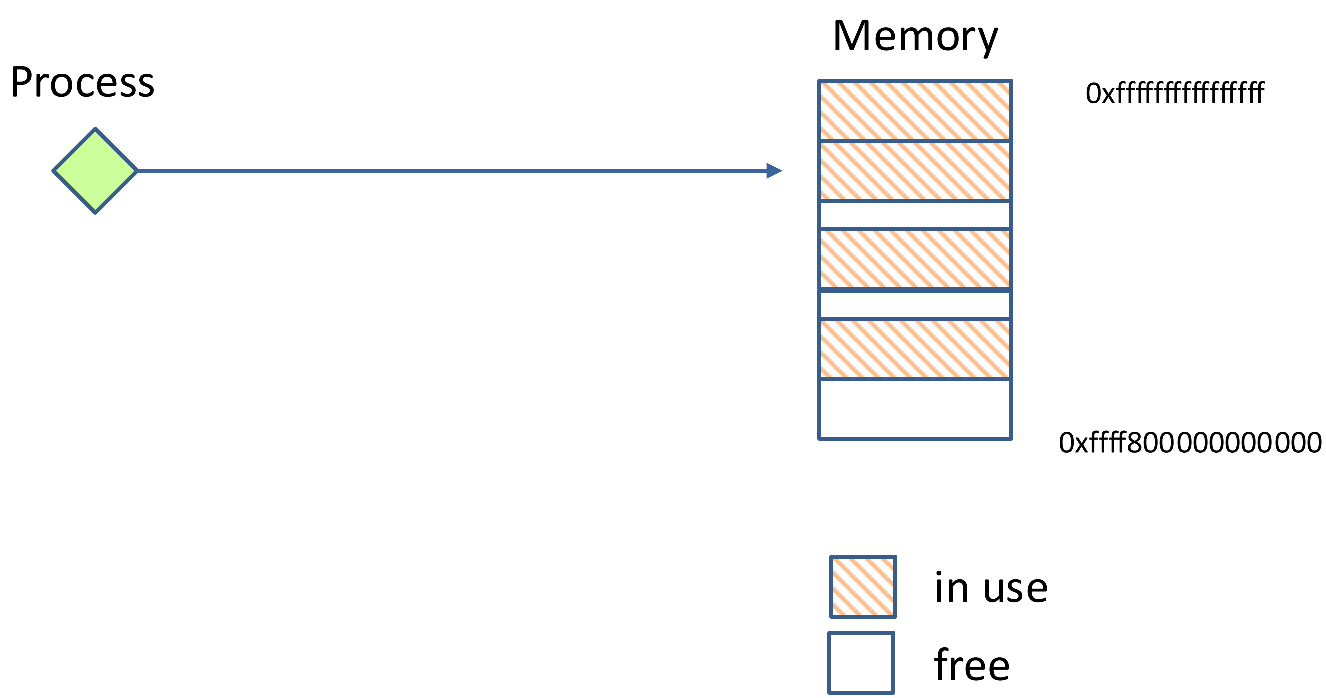
2. Kernel Space Allocator

Buddy Allocator

Slab

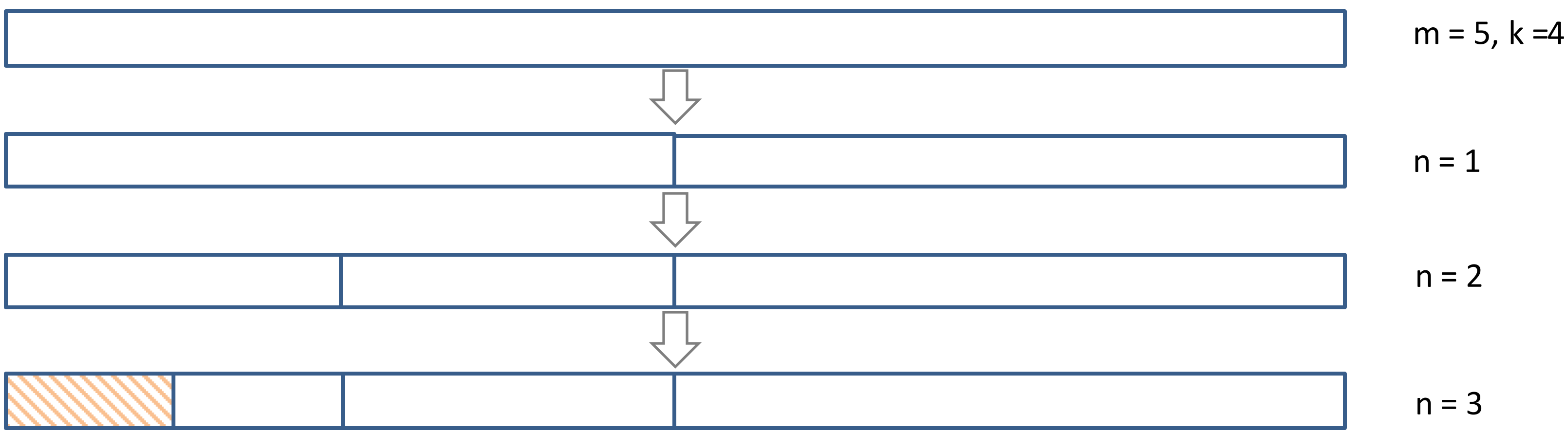
Problem Analysis

- How to manage allocated and freed memory blocks?
- Challenges:
  - May suffer fragmentation issues
  - Slowdown the system when coalescing neighbor chunks



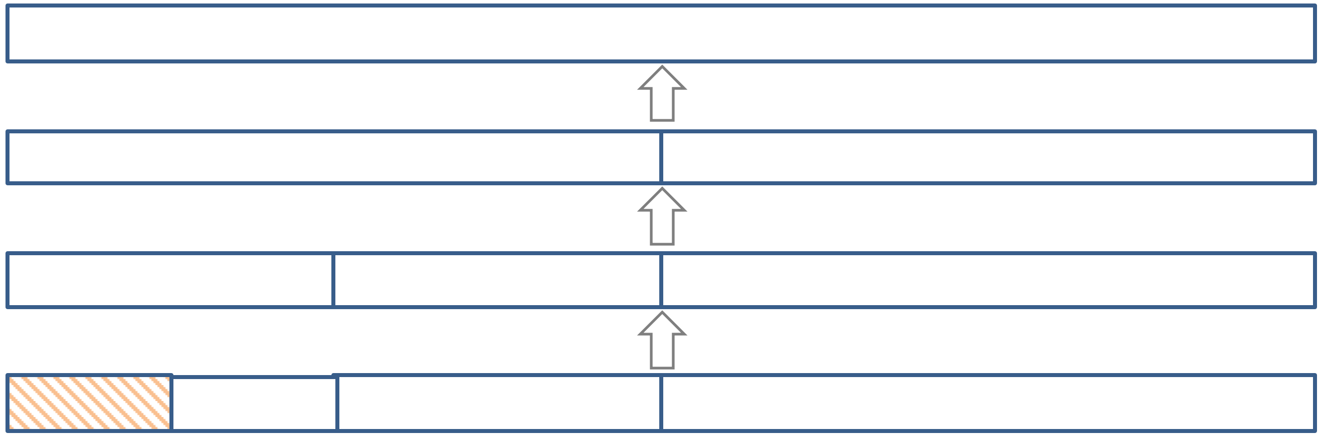
Buddy Algorithm

- Memory spaces are managed as blocks of pages.
  - Block size: 2^m pages, page size: 4K bytes
  - e.g., when m = 10, block size = 1024 * 4K bytes
- Supposing requesting k memory pages, $k < 2^{m-1}$
  - => Segment the blocks n times until $k > 2^{m-n-1}$



Buddy Algorithm: Deallocation

- Repeatedly merge with adjacent blocks if they are free.
- Condition of merge:
  - The adjacent blocks are equal size.
  - The address after merging should always be aligned to the block size.



Buddy Structure

TAG	TYPE	INDEX	data

TAG (1 bit): allocated or free

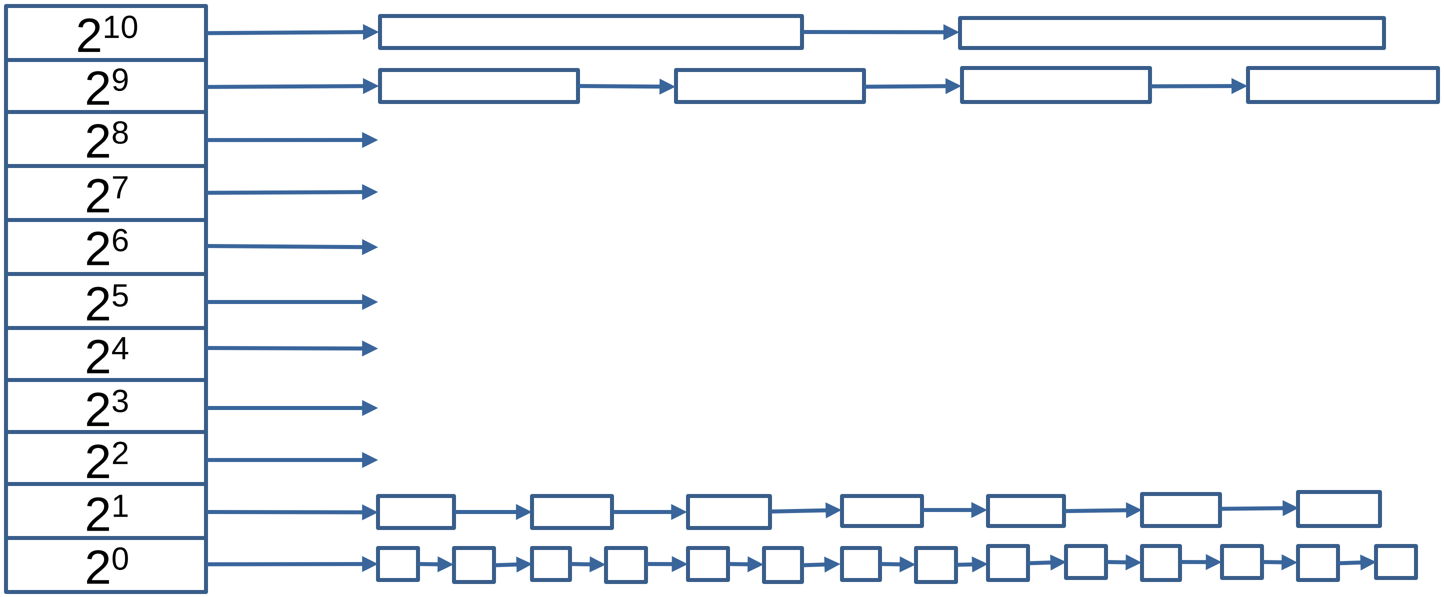
TYPE (2 bits):

- first bit: left or right buddy
- second bit: whether the parent is the left or right buddy

INDEX ($\log_2 n$ bits): size

Buddy Allocator: Free Lists

- Free blocks are managed as lists.
- Each list maintains blocks of the same size.
  - Largest block: 2^{10} pages
  - Smallest block: 2^{0} pages
- Allocation: search from the list of the best fit
  - If the list is empty, try another list with larger blocks



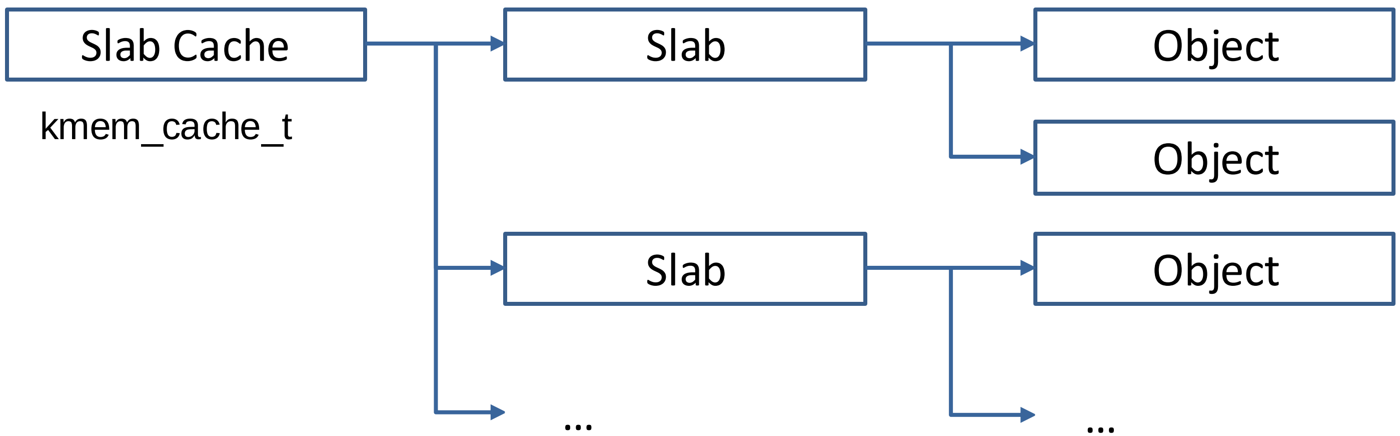
View Free Lists via /proc/buddyinfo

```
#: cat /proc/buddyinfo
Node 0, zone      DMA      0    0    0    0    0    0    0    0    0    1    3
Node 0, zone    DMA32      9    6    7    6    9    5    7    7    6    5  445
Node 0, zone   Normal   1388  445  216   94   56   47   45   12    2    4  700
```

```
#: dmesg
[    0.016942] Faking a node at [mem 0x0000000000000000-0x000000027c7fffff]
[    0.016957] NODE_DATA(0) allocated [mem 0x27c7d6000-0x27c7fffff]
[    0.017361] Zone ranges:
[    0.017362]   DMA      [mem 0x0000000000001000-0x0000000000ffffff]
[    0.017366]   DMA32    [mem 0x0000000001000000-0x00000000ffffffff]
[    0.017369]   Normal   [mem 0x0000000100000000-0x000000027c7fffff]
[    0.017371]   Device   empty
```

Slab Allocation

- Byte-based allocation
- Reduce interactions with the buddy system
- Use cache to save the initialization cost of frequently used data structures (*e.g.*, task_struct, inodes)



View Slab Info via /proc/slabinfo

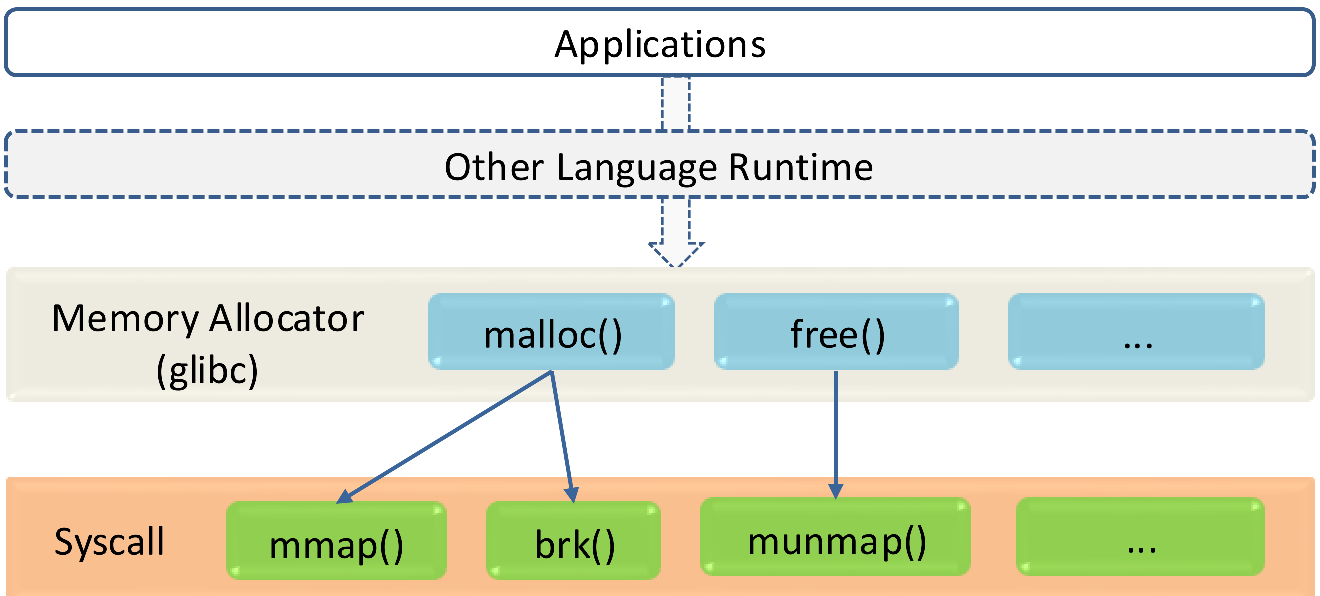
```
#: cat /proc/slabinfo
# name  <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab>
: tunables <limit> <batchcount> <sharedfactor>
: slabdata <active_slabs> <num_slabs> <sharedavail>

nf_conntrack   1000   1075    320   25    2 : tunables   0   0    0 : slabdata    43   43    0
au_vdir           0      0    128   32    1 : tunables   0   0    0 : slabdata     0    0    0
au_finfo          0      0    192   21    1 : tunables   0   0    0 : slabdata     0    0    0
au_icntnr         0      0    832   39    8 : tunables   0   0    0 : slabdata     0    0    0
au_dinfo          0      0    192   21    1 : tunables   0   0    0 : slabdata     0    0    0
ovl_inode        44     44    720   22    4 : tunables   0   0    0 : slabdata     2    2    0
...
```

3. User Space Allocator

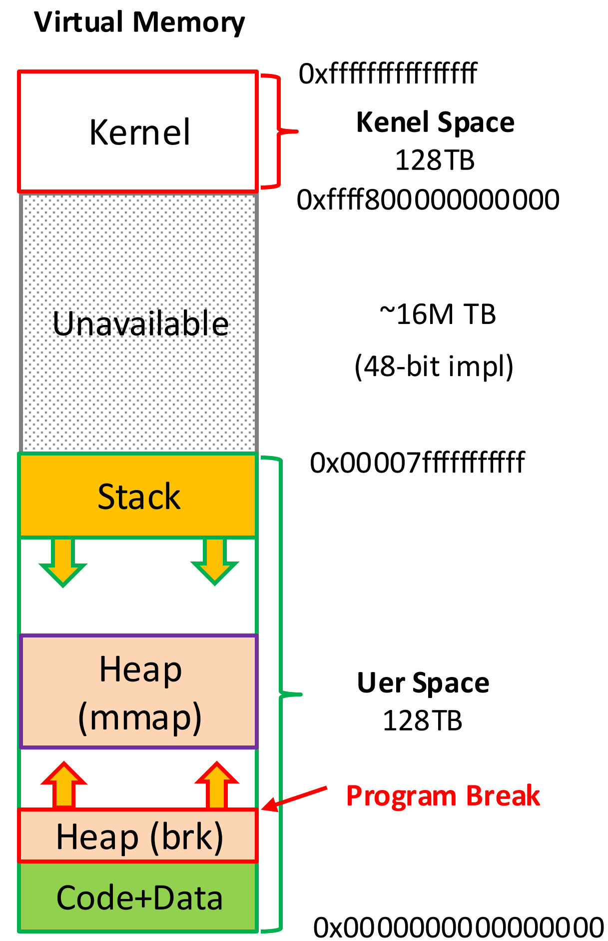
User Space Allocation APIs

- Memory allocator provides user-friendly APIs.
  - e.g., malloc() and free() in glibc APIs
- Memory allocator invokes syscalls for 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 *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

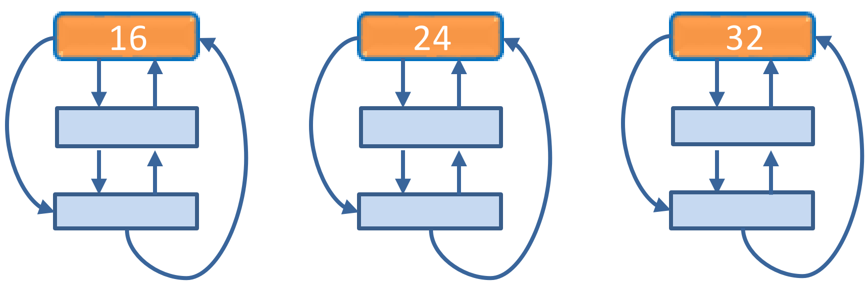
- malloc(size_t n)
  - Allocate a new memory space of size n;
  - The memory is not cleared;
  - Return the address pointer.
- free(void * p)
  - Release the memory space pointed by p;
  - Do not return to the system directly (for brk).
  - What would happen if p is null or already freed?
- calloc(size_t 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 Challenges for Allocator

- Each system call costs nearly a hundred CPU cycles.
  - =>An allocator should not frequently invoke system calls
- Heap data are not compact.
  - =>To free a block, the allocator cannot simply decrease the break pointer.
- How to manage and reuse freed memory chunks?

Basic Idea: Doug Lea's Allocator (dlmalloc)

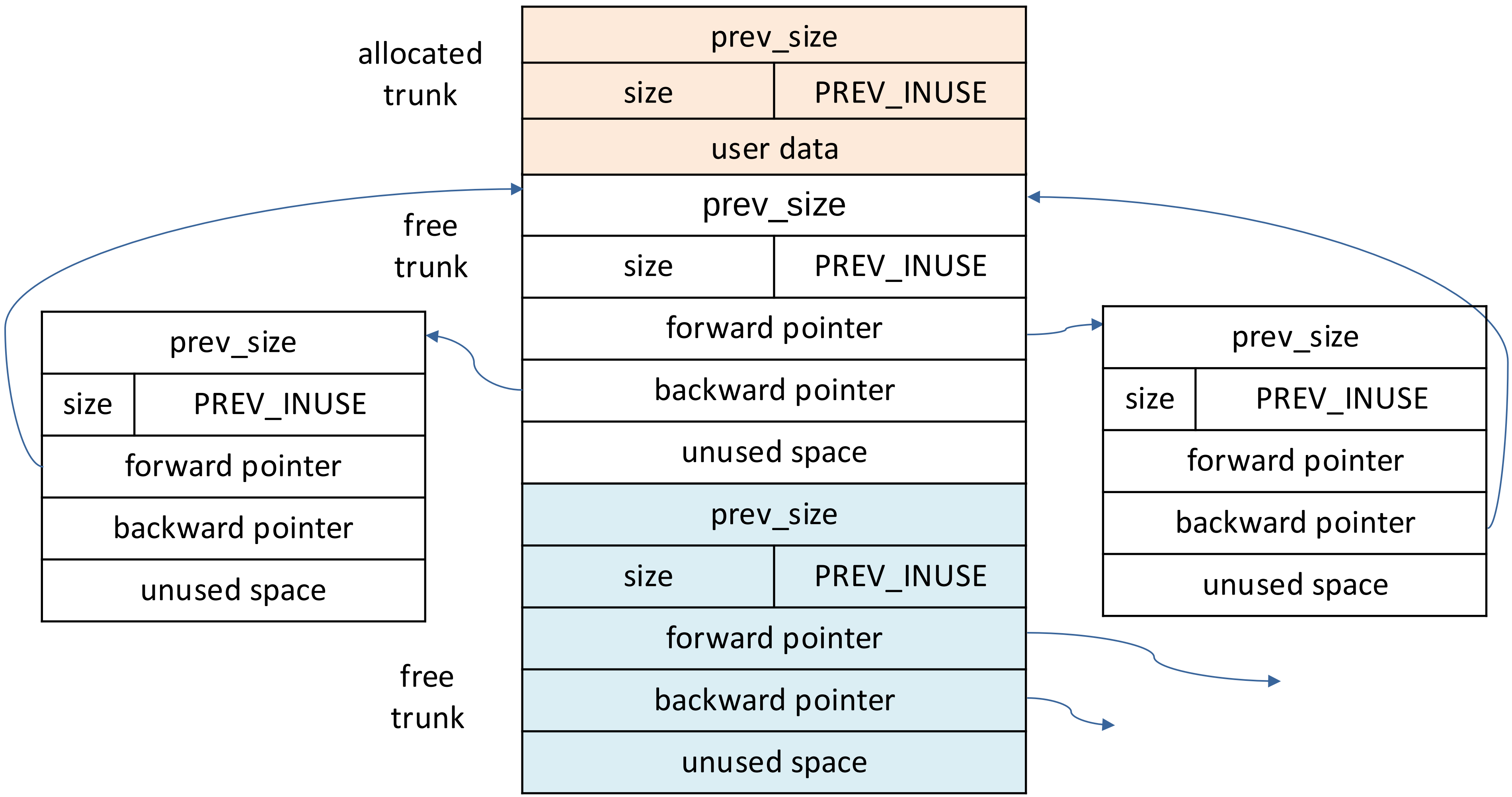
- Freed memory chunks are managed as bins
  - Each bin is a double-lined list of freed chunks of “fixed” size
  - Bins for sizes < 512 bytes are spaced 8 bytes apart
  - Larger bins are approximately logarithmically spaced
- malloc() finds the corresponding bin for allocation
  - index is computed by logical shift
  - first-in-first-out



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

Structure of Chunks: Boundary Tag

- Sizes information is stored in the front of each chunk
- To facilitate consolidating fragmented chunks



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 before adding to lists.

	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/

Coding Practice: A Toy Allocator

Practice

- Write a user space allocator based on a code template.

```
struct chunk {
    uint64_t prev_size; // size of previous chunk
    uint64_t size; // size in bytes including overhead
    struct chunk* fd;
    struct chunk* bk;
};
```

prev_size	
size	PREV_INUSE (1 bit)
forward pointer	
backward pointer	

structure of a free trunk

prev_size	
size	PREV_INUSE (1 bit)
user data	

structure of an allocated trunk

The new_malloc() Function Is Implemented

- new_malloc (): find a trunk in the free list for allocation:
- trunk size > required size => split it into two chunks

```
void *x1 = malloc_new(8);
void *x2 = malloc_new(16);
void *x3 = malloc_new(32);
void *x4 = malloc_new(48);
void *x5 = malloc_new(64);
view_chunk(p);
```

```
===============trunk view begin=====================
chunk addr = 0x555555559000, prev size = 0, size: 24, prev inuse: 1, fd = (nil), bk = 0x18
chunk addr = 0x555555559018, prev size = 24, size: 32, prev inuse: 1, fd = (nil), bk = (nil)
chunk addr = 0x555555559038, prev size = 32, size: 48, prev inuse: 1, fd = (nil), bk = (nil)
chunk addr = 0x555555559068, prev size = 48, size: 64, prev inuse: 1, fd = (nil), bk = (nil)
chunk addr = 0x5555555590a8, prev size = 64, size: 80, prev inuse: 1, fd = (nil), bk = (nil)
chunk addr = 0x5555555590f8, prev size = 80, size: 776, prev inuse: 1, fd = (nil), bk = (nil)
===============trunk view end=====================
```

Tasks: Implement the new_free() function

- new_ free(): add the trunk back to the list
- Merge the trunk with its previous trunk if possible

```
new_free(x1);
new_free(x2);
new_free(x3);
new_free(x4);
new_free(x5);
view_chunk(p);
```

Expected output:

```
===============trunk view begin=====================
chunk addr = 0x555555559000, prev size = 0, size: 248, prev inuse: 1, fd = 0x5555555590f8, bk = 0x18
chunk addr = 0x5555555590f8, prev size = 248, size: 776, prev inuse: 0, fd = (nil), bk = (nil)
===============trunk view end=====================
```