National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Memory Allocation

IOC5226 Operating System Capstone

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao Tung University

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - MIT 6.828 Operating system engineering class, 2018
  - MIT 6.004 Operating system, 2018
  - Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC

Outline

- Dynamic memory allocation
- Buddy memory allocator
- Slab memory allocator

Dynamic memory allocation

- How does the OS manage memory of a single process ?
  - Each process has contiguous logical address space
- **Static (compile-time) allocation** is not always a good choice
  - Recursive procedures
    - Data dependencies are hard to predict
  - Complex data structures
    - Link-list, tree, graph (ptr = malloc(x); free(ptr))
- **Dynamic allocation**
  - Stack allocation
  - Heap allocation

Stack organization

- Stack grows happens via
  - **mremap ()** : remap a virtual memory address
- When is it useful ?
  - Memory allocation and freeing are partially predictable
  - Examples
    - Procedure call frames, tree traversal, recursion
- Advantages
  - Keeps all the free space contiguous
  - Simple and efficient to implement
- Disadvantages
  - Not appropriate for all data structures

alloc(A)
alloc(B)
alloc(C)
free(C)
free(B)
free(A)

Heap organization

- Allocate from random locations
  - Memory contains **allocated** areas and **free** areas
- When is it useful ?
  - Allocation and release are unpredictable
  - Arbitrary list structures, complex data organizations
    - E.g. new in C++, malloc() in C
  - Advantage: works on arbitrary allocation and free patterns
  - Disadvantage: End up with small chunks of free space

Stack vs heap allocation

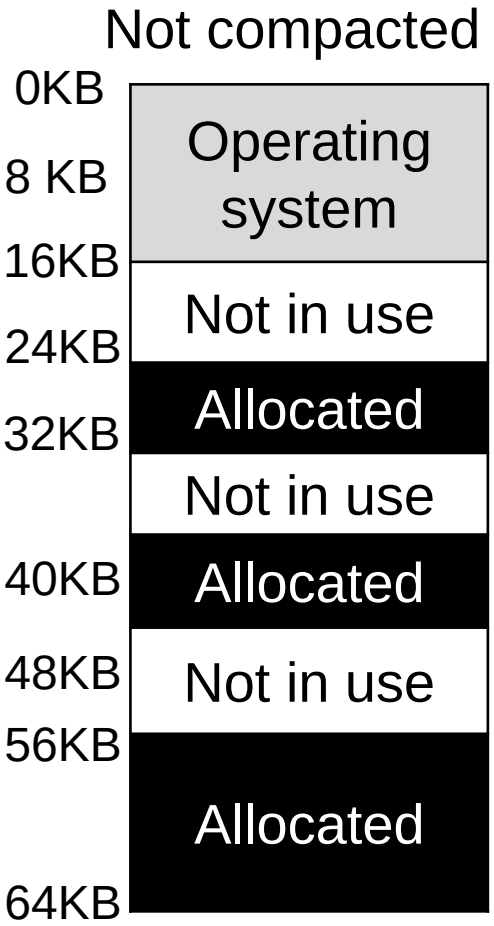
Parameter	Stack	Heap
Basic	Allocated in a contiguous block	Allocated in a random order
Allocation	Automatic by compiler	Manual by programmer
Main issue	Storage of memory	Memory fragmentation
Safety	Thread safe, data only accessed by owner	Not thread-safe, data stored visible to all threads
Flexibility	Fixed-size	Resizing is possible
Access time	Fast	Slow

Fragmentation

- **Internal fragmentation**
  - Waste space when you round an allocation up
- **External fragmentation**
  - When you end up with small chunks of free memory that are too small to be useful

External fragmentation

- **External fragmentation**
  - Full of little holes of free space
  - Have a number of segments per process
  - Each segment might be a different size
  - It is difficult to allocate new segments
- **Compact physical memory**
  - Rearranging the existing segments
  - Compaction is expansive
  - Best-fit, worst-fit, first-fit, buddy algorithm



External fragmentation (cont.)

- **When does external fragmentation occur ?**
  - The free space consists of variable-sized units
  - This arises in a user-level memory allocation library (malloc())
  - Chops segments into little pieces of different sizes
- **Problems of the external fragmentation**
  - No single contiguous space that can satisfy the request
  - Even the total amount of free space exceeds the size of requests
  - E.g. A request 15 bytes will fail even though there are 20 bytes free

How to allocate memory ?

Virtual address space

0

0 x ffffffff

```
int main () {
        struct foo *x = malloc(sizeof(struct foo));
        ….
}
```

```
void* malloc (ssize_t n) {
        if(heap empty)
                mmap(); // add pages to heap and find a free block of size n
}
```

malloc() issues

- How to implement malloc() or new ?
  - Calls **sbrk()** to request more contiguous memory from OS
  - Add small header to each block of memory
    - Pointer to next free block

sbrk() increments the program's data space by increment bytes. Calling sbrk() with an increment of 0 can be used to find the current location of the program break.

On success, sbrk() returns the previous program break. (If the break was increased, then this value is a pointer to the start of the newly allocated memory). On error, (void *) -1 is returned, and errno is set to ENOMEM.

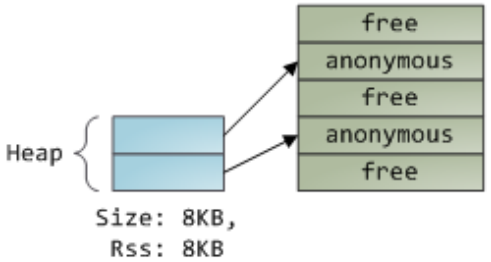
https://pages.cs.wisc.edu/~eli/537/lectures/lecture12.2x2.pdf

Enlarge VMA

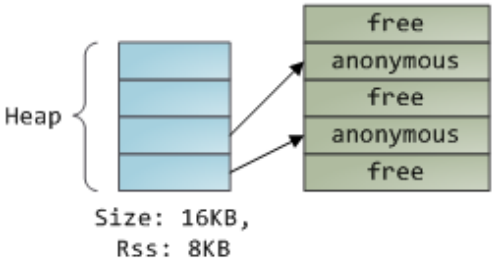
brk() sets the end of the data segment to the value specified by addr, when that value is reasonable, the system has enough memory, and the process does not exceed its maximum data size.

On success, brk() returns zero. On error, -1 is returned, and errno is set to ENOMEM.

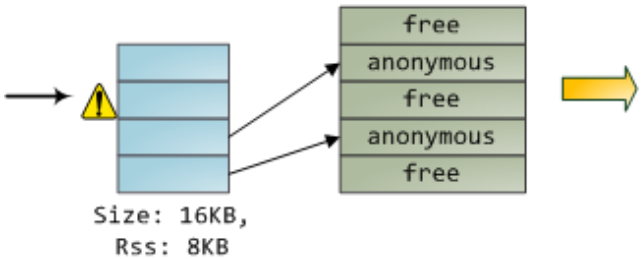
1. Program calls brk() to grow its heap



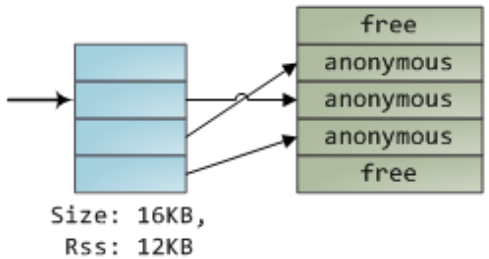
2. brk() enlarges heap VMA.
New pages are **not** mapped onto physical memory.



3. Program tries to access new memory.
Processor page faults.



4. Kernel assigns page frame to process, creates PTE, resumes execution. Program is unaware anything happened.



Source: http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/

https://pages.cs.wisc.edu/~eli/537/lectures/lecture12.2x2.pdf

Reclaiming free memory

- When can dynamically-allocated memory be freed ?
  - Explicitly call free()
  - Hard, can't be recycled until all sharers are finished
    - Sharing is indicated by the presence of pointers to the data
- Two possible problems
  - **Dangling pointers**
    - Recycle storage while it's still being used
  - **Memory leaks**
    - Forget to free storage even when can't be used again
    - Not a problem for short-lived user processes
    - Issue for operating systems and long-running applications

Buddy allocator

- Fast, simple allocation for blocks that are 2^n bytes
- Allocation restrictions
  - Block sizes: 2^n
- Allocation strategy for k bytes
  - Raise allocation request to the nearest 2^n
  - Search free list for appropriate size
    - Recursively divide large blocks until reach block of correct size
  - Free strategy
    - Recursively coalesce block with buddy if buddy free
    - May coalesce lazily to avoid overhead

Case study: buddy allocation

Memory blocks

0	1	2	3	4	5	6	7

- In a memory
  - Block 0, 4, 5, 6, 7 is used
  - Will buddy allocator merges block 1 and 2 if both of them are free ?
    - No !! Block 1 and 2 are not buddy

```
static inline unsigned long _find_buddy_pfn
                                (unsigned long page_pfn, unsigned int order)
{
        return page_pfn ^ (1 << order);
}
```

Buddy Allocator Issues

- Appropriate for medium-size allocations
- A page is usually **4KB** that is dependent to the hardware
- **Buddy allocator strategy**
  - Only allocations of power of two numbers of pages such as 1, 2, 4, 8, 16 pages, etc.
  - Typical maximum size is 8192 KB
  - The allocated area is contiguous in the kernel virtual address space
  - Maps to physically contiguous pages

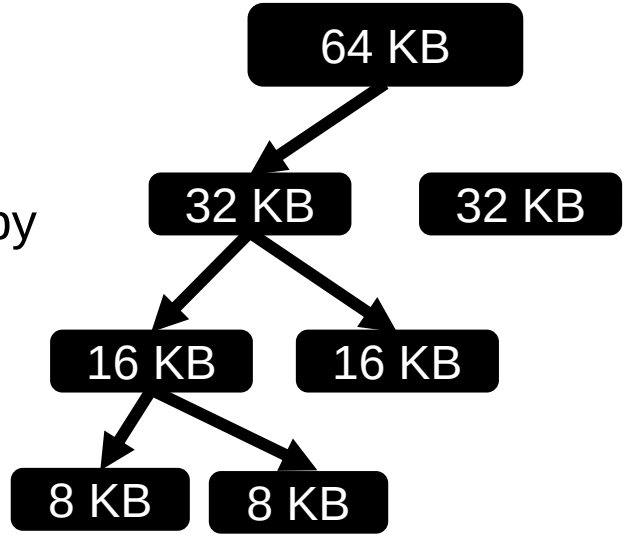
Buddy Allocator Issues

- **Memory fragmentation**
  - Wastage of memory due to fragmentation
  - Buddy allocator still leads to few reserved pages that prevent the allocation of larger contiguous blocks
- **Performance**
  - Very fast, since the simple binary shift or bit change arithmetic

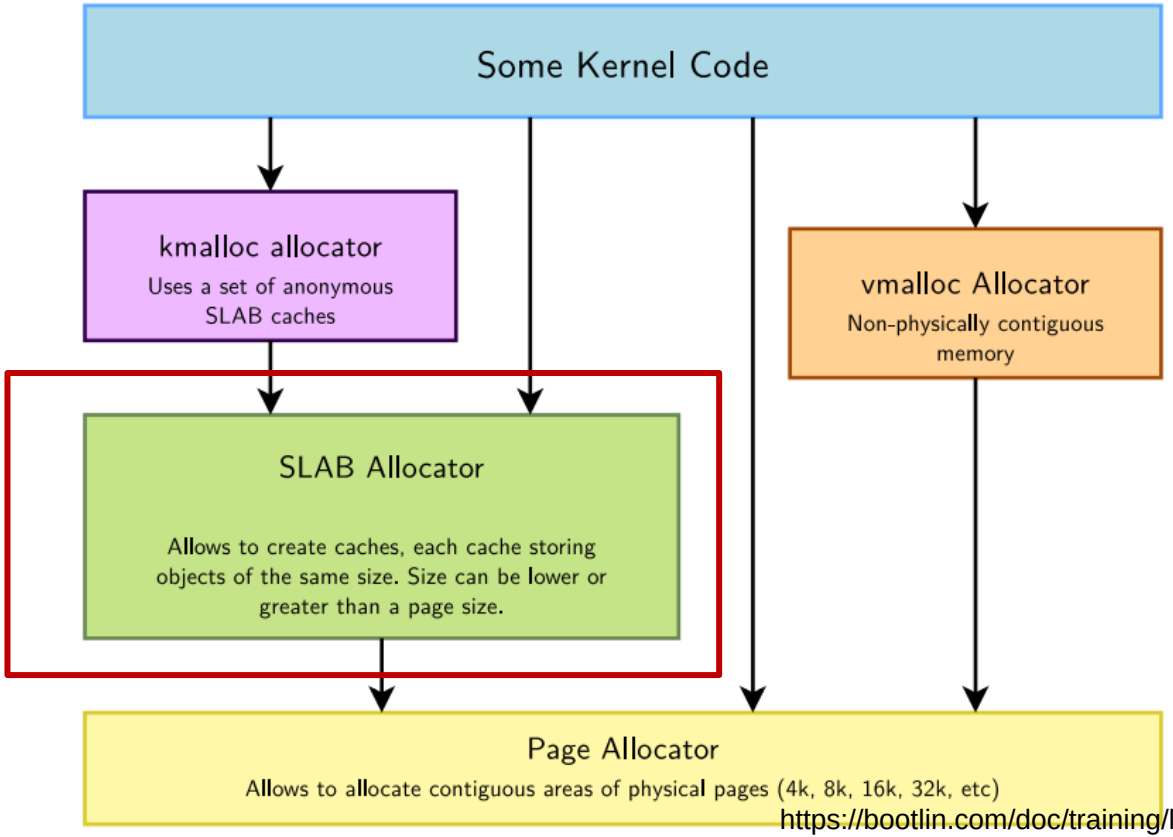
Memory layout of the buddy allocator

Buddy Allocator Issues

- **Binary buddy allocator**
  - Free memory as one big space of size 2^N
  - Recursive search by dividing free space by two until a block that is big enough to accommodate the request is found
  - **Internal fragmentation** as only allowed to power-of-two-sized block
  - Check whether the “buddy” 8KB is free when returning the 8KB block to the free list
  - Keep coalescing when the buddy is still free
  - Making coalescing simple



Slab Allocation



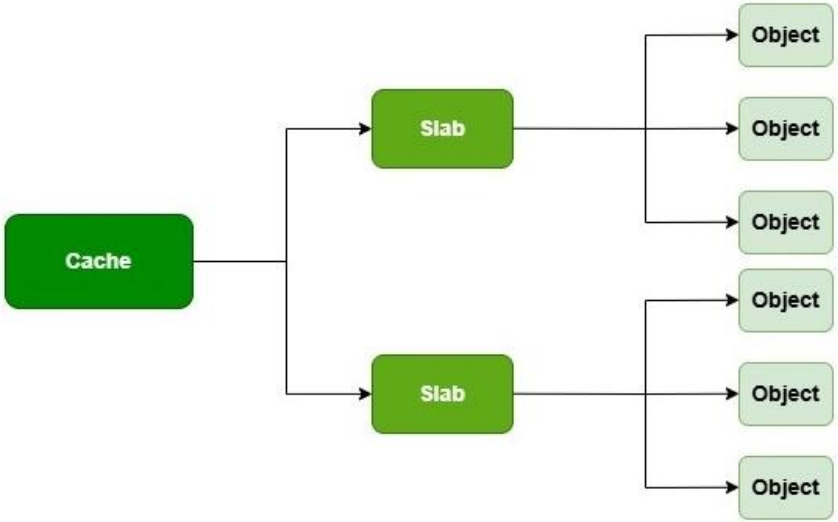
https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

Slab Allocation

- Slab allocation goals
  - To fix the memory fragmentation shown in the buddy system
  - Efficiently allocate memory for frequently used objects
    - File descriptors, inodes, and network buffers
    - Work on top of the buddy system
- What is Slab Allocation?
  - A cache for each type of object that the kernel frequently uses
  - Each cache consists pre-allocated memory chucks called slabs
  - Slabs are divided into smaller fixed-size blocks called objects

Slab Allocation

- What is Slab Allocation?
  - The object can be quickly allocated and deallocated from the slab cache without causing fragmentation and memory wastage



https://reurl.cc/0mXVZA

Slab Allocation

- Structure of Slab Allocation
  - **Cache**
    - A group of slabs that store objects of the same type
  - **Slab**
    - A contiguous block of memory
    - Contains multiple objects of the same type
  - **Objects**
    - A fixed-size block of memory that is allocated from a slab
    - Each object maps to a specific data structure used by kernel

https://reurl.cc/0mXVZA

Slab Allocation

- Each of the slab can be in one of three states
  - **Full:** all objects are allocated
  - **Partial:** some objects are allocated
  - **Empty:** No objects are allocated

https://reurl.cc/0mXVZA

Slab Allocation

- The slab allocator
  - Uses partial slab first to satisfy allocation requests
  - This way fragmentation is minimized
  - If no partial slabs are available, a new slab is created from the buddy system

```
struct slab {
        struct list_head  list;       /* full, partial, or empty list */
        unsigned long     colouroff;  /* offset for the slab coloring */
        void              *s_mem;     /* first object in the slab */
        unsigned int      inuse;      /* allocated objects in the slab */
        kmem_bufctl_t     free;       /* first free object, if any */
};
```

A slab descriptor

https://reurl.cc/0mXVZA

Slab Allocation

- Slab Allocator
  - Maintains a data structure, **kmem_cache,** for each type of object
  - Kmem_cache contains information about
    - The size of the object, the number of objects per slab, pointers to the lists of full, partial, and empty slabs

```
struct kmem_cache {
        size_t              object_size;   /* Size of each object */
        size_t              slab_size;     /* Size of each slab */
        unsigned int       objects_per_slab; /* Number of objects per slab */
        struct list_head   full_slabs;    /* List of full slabs */
        struct list_head   partial_slabs; /* List of partial slabs */
        struct list_head   empty_slabs;   /* List of empty slabs */
        // Other fields...
};
```


https://reurl.cc/0mXVZA

Slab Allocation

- Simple List of Blocks (SLOB) Allocator
  - A simpler version of the slab allocator
  - Design for systems with low memory storage
  - Just keep a free list of each available chunk and its size
  - Currently uses a first-fit algorithm
  - Grab the first one big enough to work
  - Split block if leftover bytes
  - No internal fragmentation
  - External fragmentation? Yes. Trade for low overheads

https://reurl.cc/0mXVZA

Slab Allocation

- Unqueued Slab (SLUB) Allocator
  - Default slab allocator in modern Linux kernels
  - More efficient and scalable than the original slab allocator
  - Simple free list per slab – no per-slab metadata
  - Uses a data structure, **slub_cache**, to manage memory allocation

```
struct slub_cache {
        size_t              object_size;   /* Size of each object */
        size_t              slab_size;     /* Size of each slab */
        unsigned int        objects_per_slab; /* Number of objects per slab */
        struct list_head    full_slabs;    /* List of full slabs */
        struct list_head    partial_slabs; /* List of partial slabs */
        struct list_head    empty_slabs;   /* List of empty slabs */
        // Other fields...
};
```

https://reurl.cc/0mXVZA

Slab Allocation

- Advantages of Slab Allocation
  - **Reduced fragmentation**
    - Reduce the internal fragmentation caused by buddy system
    - Allocates memory in fixed-size objects
  - **Faster allocation and deallocation**
    - Slabs are pre-allocated
    - Allocate and deallocate objects are fast

https://reurl.cc/0mXVZA

Slab Allocation

- Advantages of Slab Allocation
  - **Memory reuse**
    - When an object is deallocated, it is returned to the slab cache for reuse
    - This reduces the need for frequent memory allocation from the buddy system

https://reurl.cc/0mXVZA

Slab Allocation vs Buddy System

Feature	Slab Allocation	Buddy System
Fragmentation	Reduce internal fragmentation by allocating fixed-size objects	Variable-sized allocations
Allocation Speed	Faster, pre-allocated slab	Slow, splitting and merging blocks
Object caching	Caches frequently used objects for quick access	No object caching
Memory Reuse	Deallocated objects are returned to slab cache for reuse	Deallocated memory is returned to the buddy system -> fragmentation

https://reurl.cc/0mXVZA

kmalloc allocator

- **kmalloc()**
  - Allocate memory for the kernel from general purpose caches
  - For small sizes, it relies on generic SLAB caches (see /proc/slabinfo)
  - For large sizes, it relies on the page allocator
  - The allocated area is guaranteed to be physical contiguous
  - The allocated area size is rounded up to the size of the smallest SLAB cache in which it can fit

kmalloc API

- #include <linux/slab.h>
- void ***kmaloc**(size_t size, int flags);
  - Allocate size bytes and return a pointer to the area (virtual address)
  - Size: number of bytes to allocate
  - Flags: same flags as the page allocator (GFP_KERNEL, GFP_ATOMIC, GFP_DMA, etc.)
- void **kfree**(const void *objp);
  - Free an allocated area

```
struct ib_port_attr *tprops;
tprops = kmalloc(sizeof *tprops, GFP_KERNEL);
…
kfree(tprops);
```

drivers/infiniband/core/cache.c

vmalloc allocator

- The **vmalloc()** allocator
  - Used to obtain memory zones that are **contiguous in the virtual addressing space**, but not made out of physically contiguous pages
  - The allocated area is in the kernel space part of the address space
  - Allocations of fairly large areas is possible
  - Physical memory fragmentation is not an issue
  - Areas **cannot be used for DMA**, since DMA usually requires physically contiguous buffers
  - API in **include/linux/vmalloc.h**
  - void *vmalloc(unsigned long size); // return a virtual address

Conclusion

- Dynamic memory allocation
  - Fit for arbitrary complex data structure
- Buddy memory allocation
  - Simple, fast for power of two blocks
  - Fragmentation
- Slab memory allocator
  - Caching the commonly used objects

Takeaway Questions

- When does external fragmentation occur ?
  - (A) The size of each memory block is fixed
  - (B) The free space consists of variable-sized units
  - (C) The size of memory is large
- What are impacts of external fragmentation?
  - (A) No single contiguous space that can satisfy the request
  - (B) Require garbage collection to reclaim free memory space
  - (C) Increase the search time of the first fit allocation strategy

Takeaway Questions

- When are problem of the buddy allocator?
  - (A) Slow memory allocation
  - (B) External memory fragmentation
  - (C) Internal memory fragmentation
- What are impacts of slab allocator?
  - (A) Caching frequent kernel objects
  - (B) Mitigate the external fragmentation
  - (C) The better utilization of hardware cache