CS 318 Principles of Operating Systems

Fall 2021

Lecture 13: Dynamic Memory Allocation

Prof. Ryan Huang

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Administrivia

Lab 3a is out

- Due 11/05 Friday 11:59 pm
- Last lab for 318 section students, hang in there..
- Considered by many students as the most challenging lab
  - Design is important, debugging is hard, need to fix Lab 2 bugs
- Suggest coming up with designs first, making an appointment with the staff to check the design before coding

Lab 3 overview session this week

Image source: https://hashtagmomfail.com/climb-out-of-that-hole-peeps/

Memory Allocation

Static Allocation (fixed in size)

- want to create data structures that are fixed and don't need to grow or shrink
- global variables, e.g., char name[16];
- done at compile time

Dynamic Allocation (change in size)

- want to increase or decrease the size of a data structure according to different demands
- done at run time

Dynamic Memory Allocation

Almost every useful program uses it

- Gives wonderful functionality benefits
- Don't have to statically specify complex data structures
- Can have data grow as a function of input size
- Allows recursive procedures (stack growth)
- But, can have a huge impact on performance

Two types of dynamic memory allocation

- Stack allocation: restricted, but simple and efficient
- Heap allocation (focus today): general, but difficult to implement.

Dynamic Memory Allocation

Today: how to implement dynamic heap allocation

- Lecture based on [Wilson] (good survey from 1995)

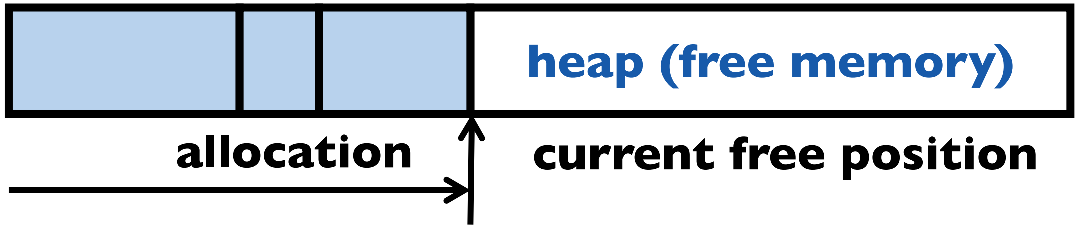
Some interesting facts:

- Two or three line code change can have huge, non-obvious impact on how well allocator works (examples to come)
- Proven: impossible to construct an "always good" allocator
- Surprising result: after 25 years, memory management still poorly understood
  - *Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator* [OSDI '21]
- Big companies may write their own "malloc"
  - Google: TCMalloc
  - Facebook: jemalloc

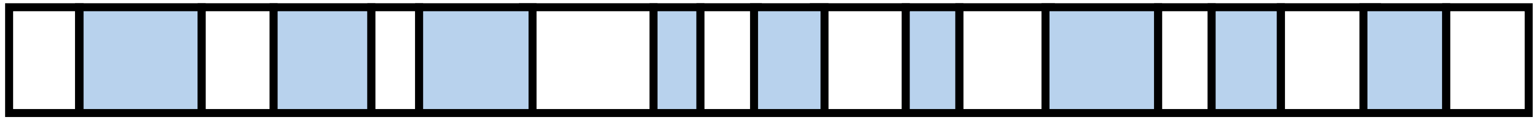
Why Is It Hard?

Satisfy arbitrary set of allocation and frees.

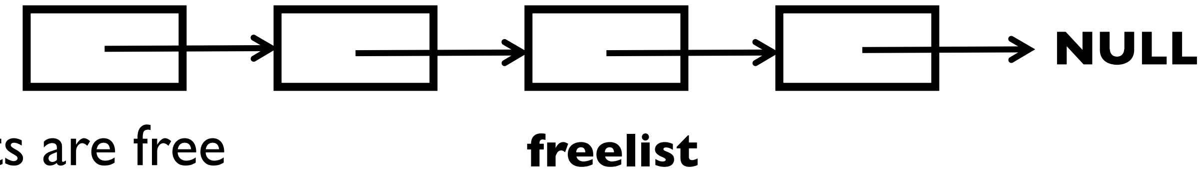
Easy without free: set a pointer to the beginning of some big chunk of memory ("heap") and increment on each allocation:



Problem: free creates holes ("fragmentation") Result? Lots of free space but cannot satisfy request!



More Abstractly

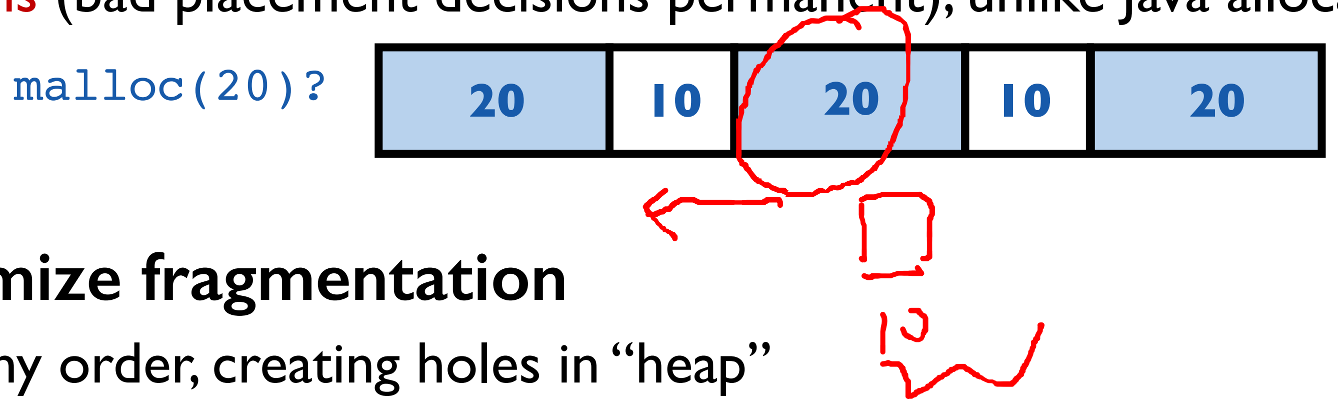


What an allocator must do?

- Track which parts of memory in use, which parts are free
- **Ideal:** no wasted space, no time overhead

What the allocator cannot do?

- Control order of the number and size of requested blocks
- Know the number, size, & lifetime of future allocations
- Move allocated regions (bad placement decisions permanent), unlike Java allocator



The core fight: minimize fragmentation

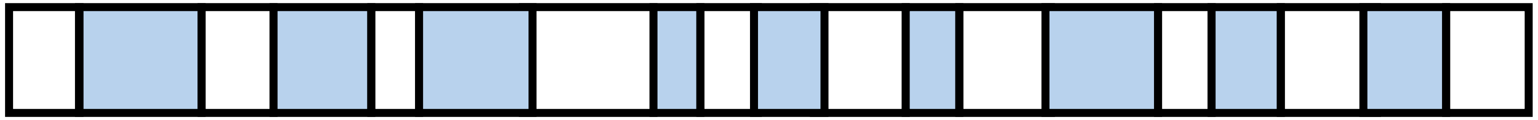
- App frees blocks in any order, creating holes in "heap"
- Holes too small? cannot satisfy future requests

What Is Fragmentation Really?

Inability to use memory that is free

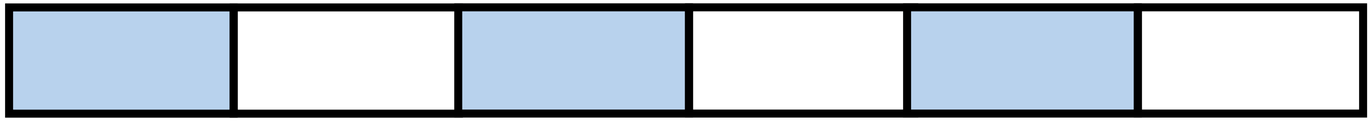
Two factors required for fragmentation

1. Different lifetimes—if adjacent objects die at different times, then fragmentation:



   - If all objects die at the same time, then no fragmentation:

2. Different sizes: If all requests the same size, then no fragmentation (that's why no external fragmentation with paging):



Important Decisions

Placement choice: where in free memory to put a requested block?

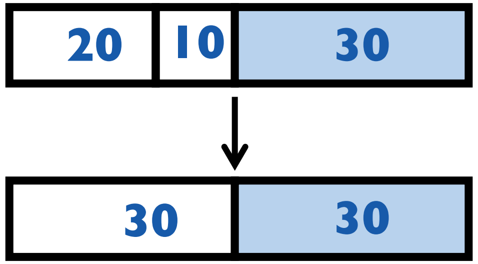
- Freedom: can select any memory in the heap
- Ideal: put block where it won't cause fragmentation later (impossible in general: requires future knowledge)

Split free blocks to satisfy smaller requests?

- Fights internal fragmentation
- Freedom: can choose any larger block to split
- One way: choose block with smallest remainder (best fit)

Coalescing free blocks to yield larger blocks

- Freedom: when to coalesce (deferring can save work)
- Fights external fragmentation



Impossible to “Solve” Fragmentation

If you read allocation papers to find the best allocator

- All discussions revolve around tradeoffs
- The reason? There cannot be a best allocator

Theoretical result:

- For any allocation algorithm, there exist streams of allocation and deallocation requests that defeat the allocator and force it into severe fragmentation ☹

How much fragmentation should we tolerate?

- Let M = bytes of live data, $\mathrm{n_{min}}$ = smallest allocation, $\mathrm{n_{max}}$ = largest allocation
- Bad allocator: $\mathrm{M \cdot (n_{max}/n_{min})}$
  - E.g., make all allocations of size $\mathrm{n_{max}}$ regardless of requested size
- Good allocator: $\sim \mathrm{M \cdot \log(n_{max}/n_{min})}$

Pathological Examples

Suppose heap currently has 7 20-byte chunks

- What’s a bad stream of frees and then allocates?

Next: two allocators (best fit, first fit) that, in practice, work pretty well

- “pretty well” = ~20% fragmentation under many workloads

Pathological Examples

Suppose heap currently has 7 20-byte chunks

- What's a bad stream of frees and then allocates?
- Free every other chunk, then alloc 21 bytes

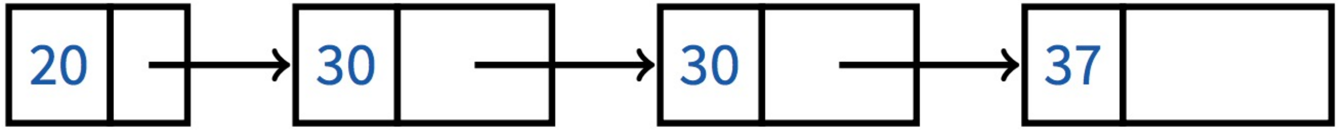
Next: two allocators (best fit, first fit) that, in practice, work pretty well

- "pretty well" = ~20% fragmentation under many workloads

Best Fit

Strategy: minimize fragmentation by allocating space from block that leaves smallest fragment

- Data structure: heap is a list of free blocks, each has a header holding block size and a pointer to the next block



- Code: Search freelist for block closest in size to the request. (Exact match is ideal)
- During free: return free block, and (usually) coalesce adjacent blocks

Potential problem: Sawdust

- Remainder so small that over time left with “sawdust” everywhere
- Fortunately not a problem in practice

Best Fit Gone Wrong

Simple bad case: allocate n, m $(n < m)$ in alternating orders, free all the ns, then try to allocate an $n + 1$

Example: start with 99 bytes of memory

- alloc 19, 21, 19, 21, 19

- free 19, 19, 19:

- alloc 20? Fails! (wasted space = 57 bytes)

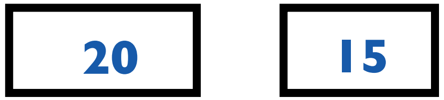
However, doesn’t seem to happen in practice

First Fit

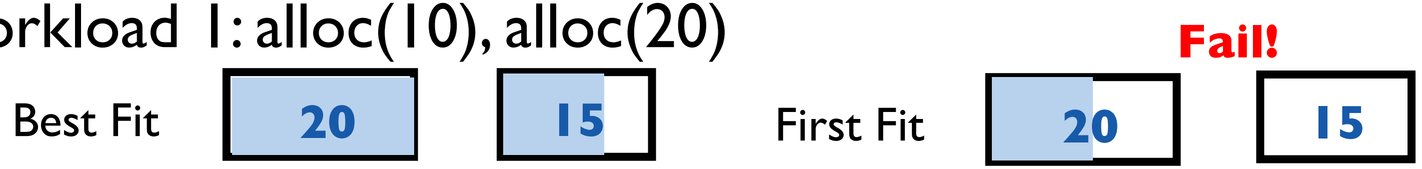
Strategy: pick the first block that fits

- Data structure: free list, sorted LIFO, FIFO, or by address
- Code: scan list, take the first one

Suppose memory has free blocks:



- Workload 1: alloc(10), alloc(20)



- Workload 2: alloc(8), alloc(12), alloc(12)

First Fit

LIFO: put free object on front of list.

- Simple, but causes higher fragmentation
- Potentially good for cache locality

Address sort: order free blocks by address

- Makes coalescing easy (just check if next block is free)
- Also preserves empty/idle space (locality good when paging)

FIFO: put free object at end of list

- Gives similar fragmentation as address sort, but unclear why

Subtle Pathology: LIFO FF

Storage management example of subtle impact of simple decisions

LIFO first fit seems good:

- Put object on front of list (cheap), hope same size used again (cheap + good locality)

But, has big problems for simple allocation patterns:

- E.g., repeatedly intermix short-lived $2n$-byte allocations, with long-lived $(n + 1)$-byte allocations

  `alloc`(8), `free`(8), `alloc`(5), `alloc`(8), `free`(8), `alloc`(5), `alloc`(8), `free`(8), …

- Each time large object freed, a small chunk will be quickly taken, leaving useless fragment. Pathological fragmentation

Some Other Ideas

Worst-fit:

- Strategy: fight against sawdust by splitting blocks to maximize leftover size
- In real life seems to ensure that no large blocks around

Next fit:

- Strategy: use first fit, but remember where we found the last thing and start searching from there
- Seems like a good idea, but tends to break down entire list

Buddy systems:

- Round up allocations to power of 2 to make management faster

Buddy Allocator Motivation

Allocation requests: frequently 2^n

- E.g., allocation physical pages in Linux
- Generic allocation strategies: overly generic

Fast search (allocate) and merge (free)

- Avoid iterating through free list

Avoid external fragmentation for req of 2^n

Keep physical pages contiguous

Used by Linux, FreeBSD

Buddy Allocator Implementation

Data structure

- *N* free lists of blocks of size 2^0, 2^1, …, 2^N

Allocation restrictions: 2^k, 0<= k <= N

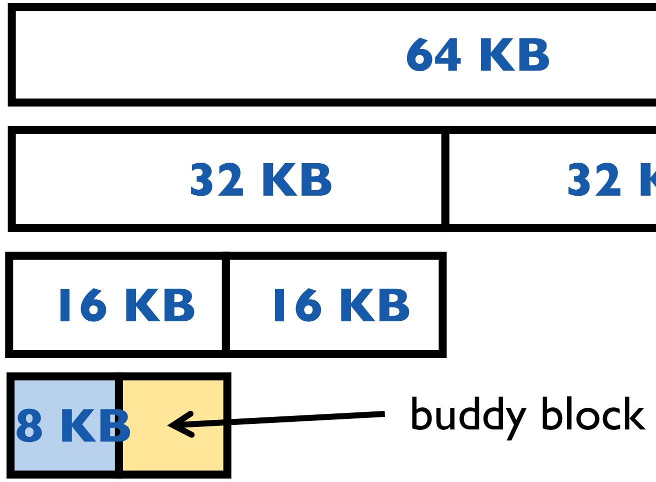
Allocation of 2^k:

- Search free lists (k, k+1, k+2, …) for appropriate size
- Recursively divide larger blocks until reach block of correct size
- Insert "buddy" blocks into free lists

Free

- recursively coalesce block with "buddy" if buddy free

Buddy Allocation



Recursively divide larger blocks until reach suitable block

- Big enough to fit but if further splitting would be too small

Insert "buddy" blocks into free lists

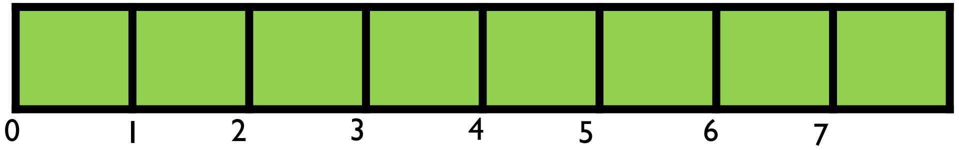
- The addresses of the buddy pair only differ by one bit!

Upon free, recursively coalesce block with buddy if buddy free

Buddy Allocation Example

$2^3 = 8$ bytes

Note: 2^3



freelist[3] = {0}

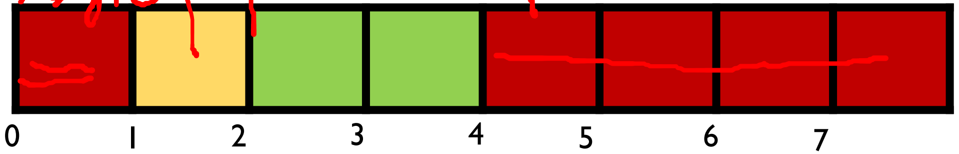
```
p1 = alloc(2^0)
```

2^0 2^1 2^2

freelist[0] = {1}, freelist[1] = {2}, freelist[2] = {4}

```
p2 = alloc(2^2)
```

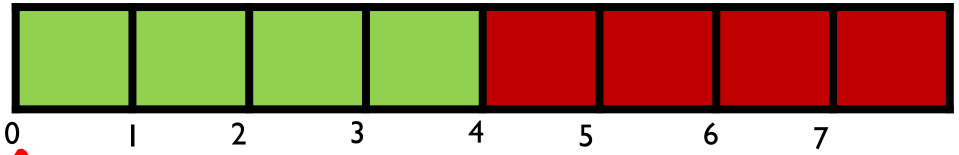
1 byte



freelist[0] = {1}, freelist[1] = {2}

```
free(p1)
```

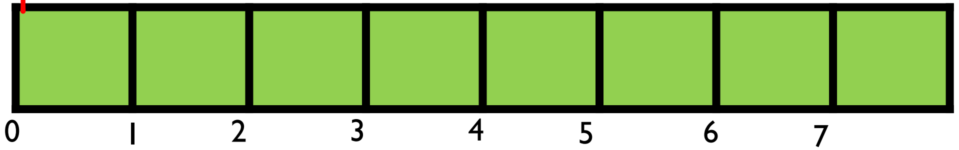
2^2



freelist[2] = {0}

```
free(p2)
```

2^3



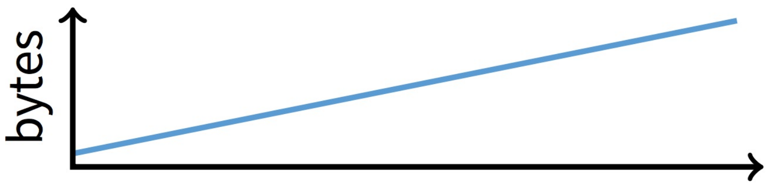
freelist[3] = {0}

Known Patterns of Real Programs

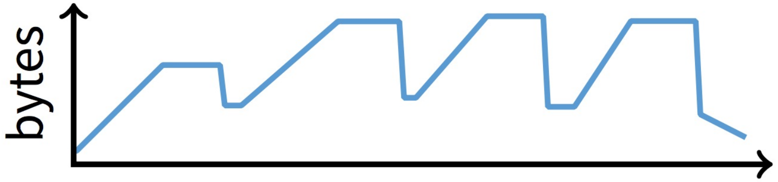
So far we've treated programs as black boxes.

Most real programs exhibit 1 or 2 (or all 3) of the following patterns of alloc/dealloc:

- *Ramps*: accumulate data monotonically over time

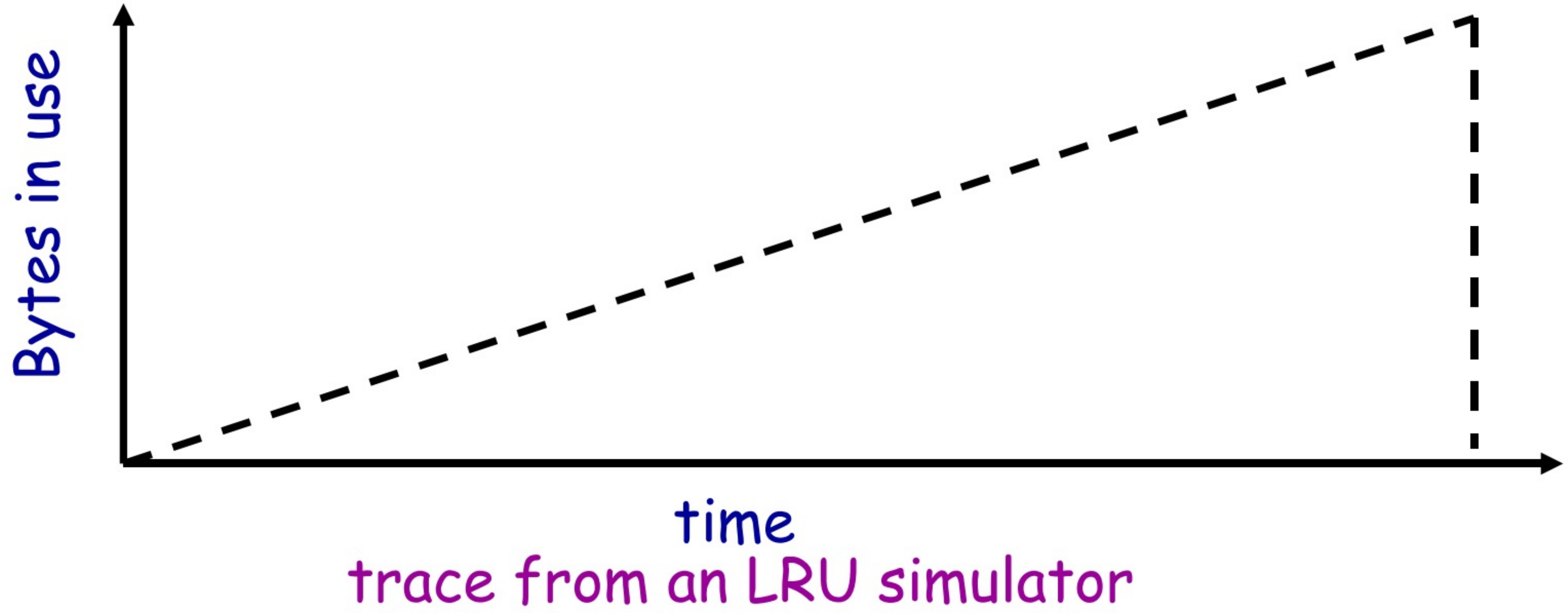


- *Peaks*: allocate many objects, use briefly, then free all



- *Plateaus*: allocate many objects, use for a long time

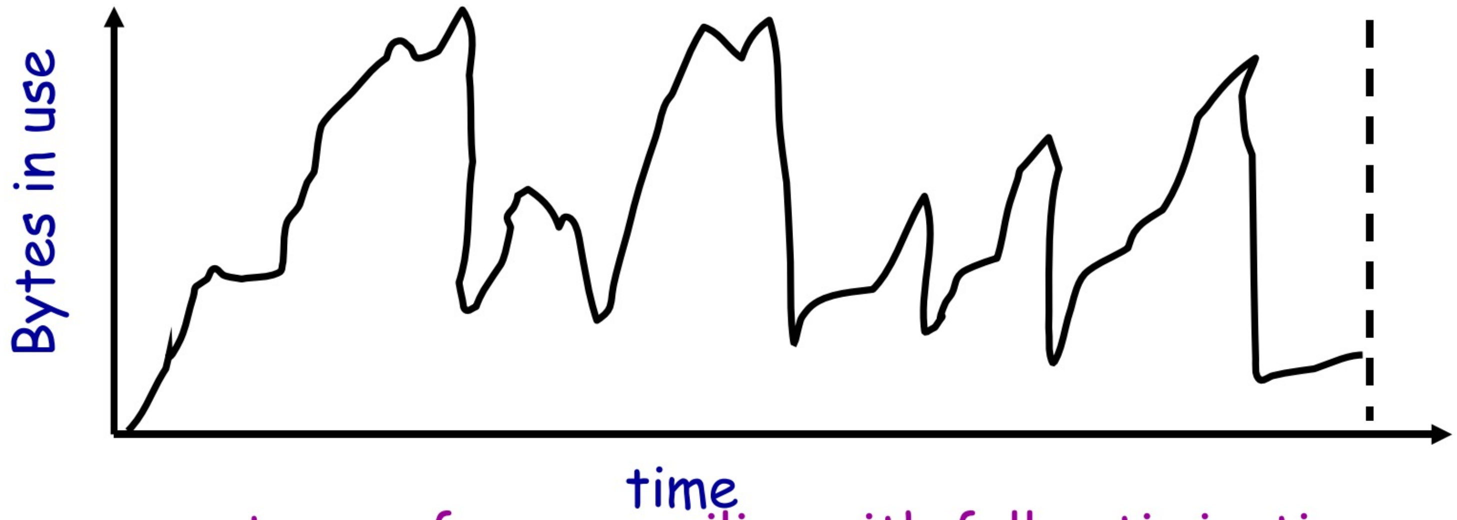
Pattern 1: ramps



In a practical sense: ramp = no free!

- Implication for fragmentation?
- What happens if you evaluate allocator with ramp programs only?

Pattern 2: Peaks



trace of gcc compiling with full optimization

Peaks: allocate many objects, use briefly, then free all

- Fragmentation a real danger
- What happens if peak allocated from contiguous memory?
- Interleave peak & ramp? Interleave two different peaks?

Exploiting Peaks

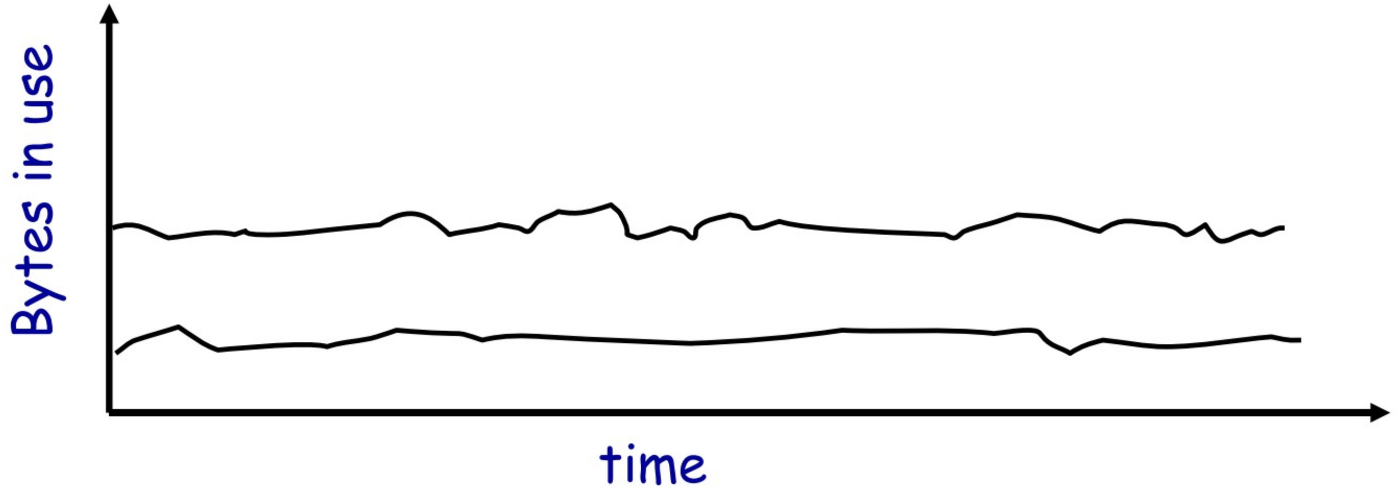
Peak phases: allocate a lot, then free everything

- Change allocation interface: alloc as before, but only support free of everything all at once
- Called "arena allocation", "obstack" (object stack)

Arena = a linked list of large chunks of memory

- Advantages: alloc is a pointer increment, free is "free"
- No wasted space for tags or list pointers
- See Pintos threads/malloc.c

Pattern 3: Plateaus



trace of perl running a string processing script

Plateaus: allocate many objects, use for a long time

Slab Allocation

Kernel allocates many instances of same structures

- E.g., a 1.7 KB `task_struct` for every process on system

Often want contiguous physical memory (for DMA)

Slab allocation optimizes for this case:

- A slab is multiple pages of contiguous physical memory
- A cache contains one or more slabs
- Each cache stores only one kind of object (fixed size)

Each slab is full, empty, or partial

Slab Allocation

E.g., need new `task_struct`?

- Look in the `task_struct` cache
- If there is a partial slab, pick free `task_struct` in that
- Else, use empty, or may need to allocate new slab for cache

Free memory management: bitmap

- Allocate: set bit and return slot, Free: clear bit

Advantages: speed, and no internal fragmentation

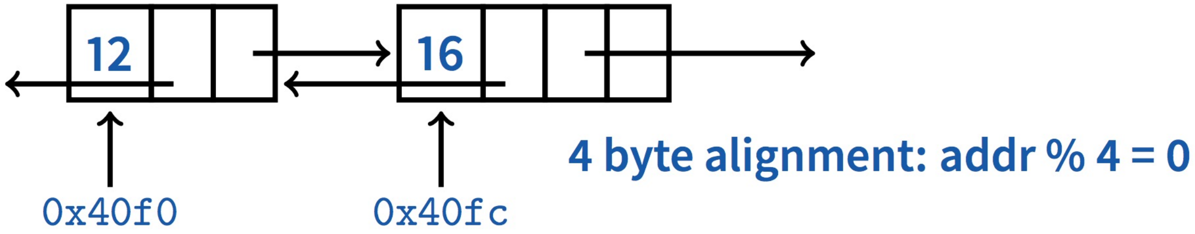
Used in FreeBSD and Linux, implemented on top of buddy page allocator

Space Overheads

Free list bookkeeping and alignment determine minimum allocatable size:

If not implicit in page, must store size of block

Must store pointers to next and previous freelist element



void *p;

p = malloc(6)

free(p);

Allocator doesn't know types

- Must align memory to conservative boundary

Implementing `malloc`

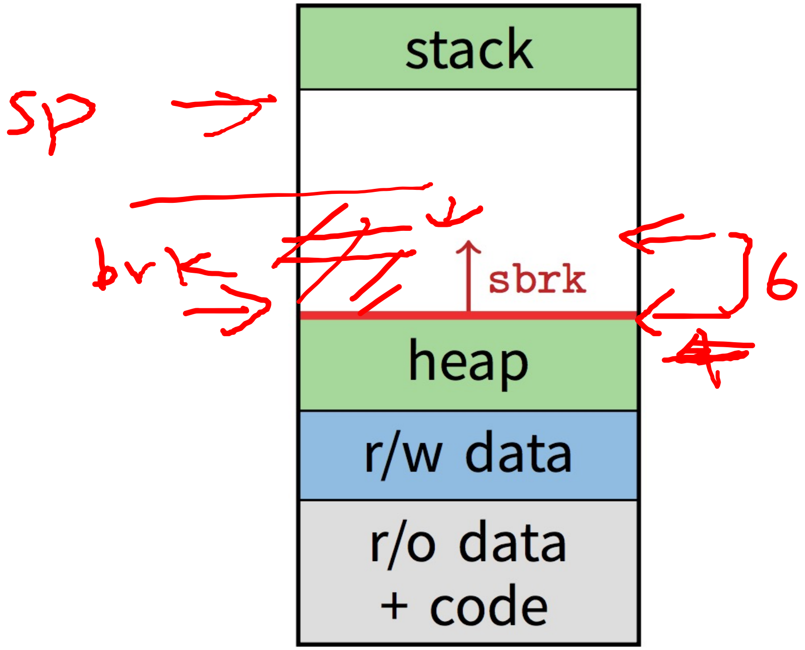
Getting More Space from OS

`malloc` **is a library call, how does** `malloc` **gets free space?**

- Note in Pintos, malloc is provided as a kernel function (see threads/malloc.c)

On Unix, can use `sbrk` **and** `brk`

- `int brk(void *p)`
  - Move the program break to address p
  - Return 0 if successful and -1 otherwise
- `void *sbrk(intptr_t n)`
  - Increment the program break by *n* bytes
  - If n is 0, then return the current location of the program break
  - Return 0 if successful and (void*)-1 otherwise



Implement `malloc()`

```
void *malloc(size_t n)
{
  char *p = sbrk(0);
  if (brk(p + n) == -1)
    return NULL;
  return p;
}

void free(void * p)
{
}
```

get current "program break"

set "program break" to be current plus n

Problem?

- Two system calls for every malloc!
- Freed blocks are not reused

Solutions

- Allocators request memory pool
- Keep track of free list
- If can't find free chunk, request from OS

Returning Heap Memory

Allocator can mark blocks as free when `free()` is called

- These blocks can be reused later by the process
- Problem: they are *not* returned to the system!
  - can cause memory pressure

Allocator can return heap memory with `brk(pBrk–n)`, but…

- `p` in `free(p)` is not always at the end of the heap!
- So can't reduce the heap size with `brk(pBrk–n)`

Therefore, for large allocations, `sbrk()` is a bad idea

- Can't return memory to the system

Solution: VM Mapping

```
void *mmap(void *p, size_t n, int prot, int flags,
  int fd, off_t offset);
```

- Creates a new mapping in the virtual address space of the calling process
- `p`: the starting address for the new mapping
- `n`: the length of the mapping
- If `p` is NULL, the kernel chooses the address at which to create the mapping
- On success, returns address of the mapped area

```
int munmap(void *p, size_t n);
```

- Deletes the mappings for the specified address range

Implement `malloc()` with `mmap()`

```c
void *malloc(size_t n)
{
  size_t *p;
  if (n == 0) return NULL;
  p = mmap(NULL, n + sizeof(size_t),
         PROT_READ|PROT_WRITE,
         MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
  if (p == (void*)-1) return NULL;
  *p = n + sizeof(size_t); // Store size in header
  p++; // Move forward from header to payload
  return p;
}
```


```c
void free(void *p)
{
  if (p == NULL) return;
  p--; // Move backward from
       // payload to header
  munmap(p, *p);
}
```

Next Time…

Chapters 36, 37