CS-446/646

Dynamic Memory

C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

Dynamic Memory

Memory Allocation

Static Allocation

- Want to create data structures that are fixed and don't need to grow or shrink
- Global variables ("*Zero-Initialized*"), **`static`** variables ("*Zero-Initialized*" & *Late-Initialized*)
- Allocation done at *Compile-Time*

Dynamic Allocation

- Want to increase or decrease the size of a data structure
- Want to allocate objects that persist outside of a *Scope*
- Done at *Run-Time*

Dynamic Memory

Dynamic Memory Allocation

Almost every useful Program uses it
- Provides important 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 (today's Lecture): General, but difficult to implement

Dynamic Memory Allocation

Today: How to implement *Dynamic Heap Allocation*

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

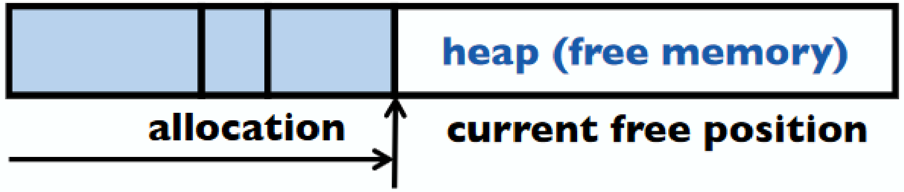
Some interesting facts:

- 2 or 3-line code changes can have huge, non-obvious impacts on how well an *Allocator* works (examples to follow)
- Proven: Impossible to construct an "always good" *Allocator*
- Surprising result: After 25 years, *Memory Management* not fully 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

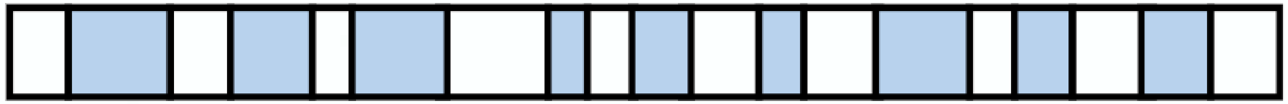
Dynamic Memory

Dynamic Allocation Challenges

- Satisfy arbitrary set of *Allocations* and **`free`***s*
- Easy without **`free`**: Set a pointer to the beginning of some big chunk of *Memory* ("*Heap*") and keep incrementing on each Allocation:

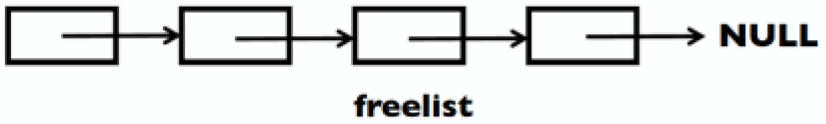


- Problem: **`free`** creates holes ("*Fragmentation*")
  - Result? Lots of free space but possibly cannot satisfy a larger request



Dynamic Allocation Challenges

- What should an *Allocator* do?
  - Track which parts of *Memory* in use, which parts are *Free*
  - Ideally: No wasted space, no time overhead



- What can an *Allocator* not do?
  - Cannot control order of the number and size of requested *Blocks*
  - Cannot know the number, size, & lifetime of future Allocations
  - Cannot move already allocated regions (bad placement decisions permanent)
    - unlike Java *Allocator*

- The core fight: Minimize *Fragmentation*
  - Application `free`s *Blocks* in any order, creating holes in the *Heap*
  - If holes are too small future requests cannot be satisfied

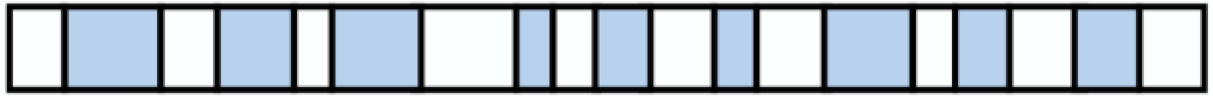
Dynamic Memory

Fragmentation

- Inability to use Memory that is *Free*

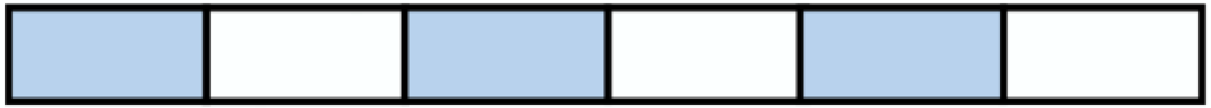
Two factors required for *Fragmentation:*

- 1. Different lifetimes—if adjacent objects die at different times, then *Fragmentation*:



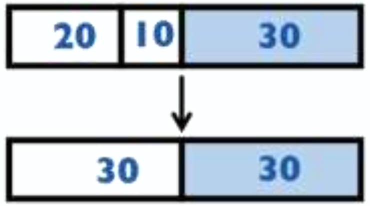
  - If all objects die at the same time, then no *Fragmentation*:

- 2. Different sizes: If all requests are of the same size, then no *Fragmentation:*
  - *Remember:* That's how *Paging* resolved *External Fragmentation*



Important Decisions

- 1. *Placement* Choice: Where in *Free Memory* to put a requested *Block*?
  - Flexibility: Can select any *Virtual Memory Address* in the *Heap*
  - Ideal: Put *Block* where it won't cause *Fragmentation* later
- 2. *Split Free Blocks* to satisfy smaller requests
  - Fights *Internal Fragmentation*
  - Can choose any larger *Block* to *Split*
  - One way: Choose *Block* that will leave the smallest remainder ("*Best-Fit*")
- 3. *Coalescing* of *Free Blocks* to yield larger *Blocks*
  - Fights *External Fragmentation*
  - Strategy 1: *Immediate Coalescing* (at *Freeing* time)
  - Strategy 2: *Deferred Coalescing* (at *Allocation* time)
    - e.g. while scanning/traversing the *Freelist*; can save work



Impossible to "Solve" *Fragmentation*

- In all *Allocation* papers all discussions revolve around Tradeoffs
  - There cannot be an "always-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, n_{min} = Smallest *Allocation*, n_{max} = Largest *Allocation*
  - Bad *Allocator*: $M \times (n_{max} / n_{min})$
    - e.g. make all *Allocations* of size n_{max} regardless of requested size
  - Good *Allocator*: $\sim M \times \log (n_{max} / n_{min})$

Pathological Examples

- Suppose *Heap* currently has 7 20-Byte chunks

  - Example of a bad stream of `free`s and then *Allocates*:
    - Free every other chunk
    - Then try to *Allocate* 21 Bytes → No fit

- Next: Two Allocators (*Best-Fit*, *First-Fit*) that, in practice, work pretty well
  - "Pretty well" = ~20% *Fragmentation* under various workloads

Best-Fit Allocator

Strategy: Minimize *Fragmentation* by *Allocating* space from block that leaves the smallest fragment

- Supporting Data Structure:
  - Heap **is** a List of *Free Blocks*,
    each has a *Header* holding the *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 (optionally) *Coalesce* adjacent *Blocks*
    - ("optionally" refers to case of *Immediate Coalescing*)
- Potential problem: "Sawdust"
  - Remainder so small that over time we are left with "sawdust" everywhere
  - Fortunately not a problem in practice

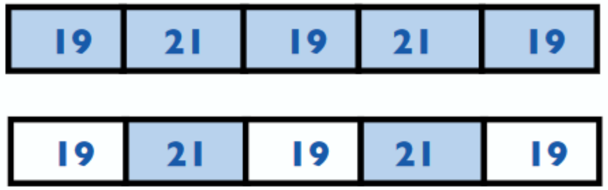
Best-Fit Allocator Gone Wrong

Simple bad case:

- 1. Allocate n, m ($n < m$) in alternating orders
- 2. **`free`** all of the n,
- 3. 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)



Note:
Approximate idea,
Blocks also have *Headers* to contend with

- However, doesn't seem to happen in practice

First-Fit Allocator

Strategy: Pick the first *Block* that fits

- Supporting Data Structure:
  - Heap **is** a List of *Free Blocks*, also **sorted** (LIFO, FIFO, or by-Address)
  - Code: Scan list, immediately take the first *Free Block* that fits

Suppose Memory has *Free Blocks*: 20 15

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

Best Fit (w/out *Splitting*)	20	15	First Fit (w/out *Splitting*)	20	15
	20	10		10	x
					Fail!

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

Best Fit (w/out *Splitting*)	20	15	First Fit (w/out *Splitting*)	20	15
	12,x	8		8,12	12
	Fail!				

Note:
Approximate idea, *Blocks* also have *Headers* to contend with

Dynamic Memory

First-Fit Allocator

LIFO : Put **`free()`**d object's *Block* at front of *Freelist*

- Simple, but causes higher *Fragmentation*
- Potentially good for CPU *Cache Locality*

Address-Sort : Order *Free Blocks* by-Address

- Makes *Coalescing* easy (just check if next *Block* is *Free*)
  - Also preserves contiguity and location of empty/idle space (Good *Locality* of *Virtual Memory* when having to *Page-Out/In*)

FIFO : Put **`free()`**d object's Block at end of *Freelist*

- Gives similar *Fragmentation* as *Address-Sort*, but unclear why

Subtle Pathology: *LIFO First-Fit*

Storage management example of the subtle impact of simple decisions

- *LIFO First-Fit* seems good:
  - Put object at front of *Freelist* (cheap), hope that same size will be requested again (cheap + good CPU *Cache Locality*)
- But, exhibits big problems with simple allocation patterns:
  - E.g., repeatedly intermix short-lived ($2\,n$)-Byte allocations, with long-lived ($n+1$)-Byte allocations

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

  - Now, each time a **short-lived** large object (($2\,n$)-Byte) is `free()`d, a small chunk of it will be quickly taken by a **long-lived** object (($n+1$)-Byte) , leaving a useless *Fragment* (cannot fit a second ($n+1$)-Byte object)
- Example of *Pathological Fragmentation*

Dynamic Memory

Other *Allocator* Ideas

Worst-Fit :

- Strategy: Fight against Sawdust by *Splitting Blocks* to maximize leftover size
  - In practice seems to ensure that no large *Blocks* stay around for too long

Next-Fit :

- Strategy: Use *First-Fit*, but remember where we found the last one and start searching from there
  - Seems like a good idea, but tends to break down entire *Freelist*

Buddy Systems:

- Round up *Allocations* to power of 2 to make management faster
  - Coming up next

Buddy Allocator **Motivation**

Allocation requests are frequently in 2^n order

- e.g. allocation of *Physical Pages* in Linux

- Generic *Allocation* strategies can prove to be overly generic

- Fast search (*Allocate*) and merge (*Free*)
  - Avoid iterating through *Freelist*

- Avoid *External Fragmentation* for requests of 2^n

- Keep *Physical Pages* Contiguous

- Used by Linux, FreeBSD

Buddy Allocator Implementation

Supporting Data Structures

- $N+1$ *Freelists* of *Blocks* of size $2^0, 2^1, \ldots, 2^N$

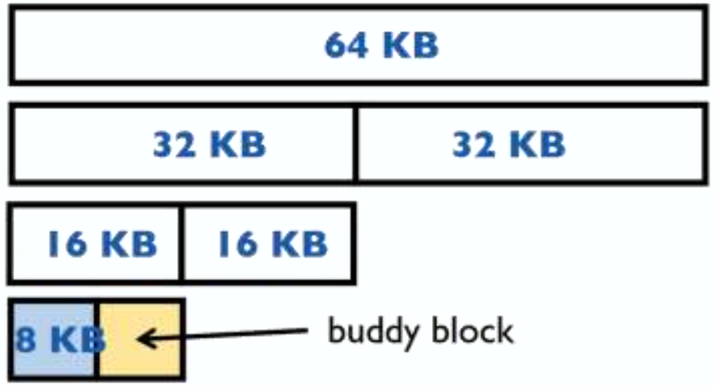
- *Allocation* restrictions: 2^k, $0 \leq k \leq N$

- *Allocation* request of 2^k:
  - Search *Freelists* (k, $k+1$, $k+2$, …) for appropriate size (smallest that can fit request)
  - Recursively divide larger *Blocks* until we reach a suitable *Block* of correct size
  - Insert "*Buddy*" *Blocks* into *Freelists*

- *Free* request:
  - Recursively *Coalesce* the `free()`d *Block* with "*Buddy*", if *Buddy* is *Free*

Dynamic Memory

Buddy Allocation

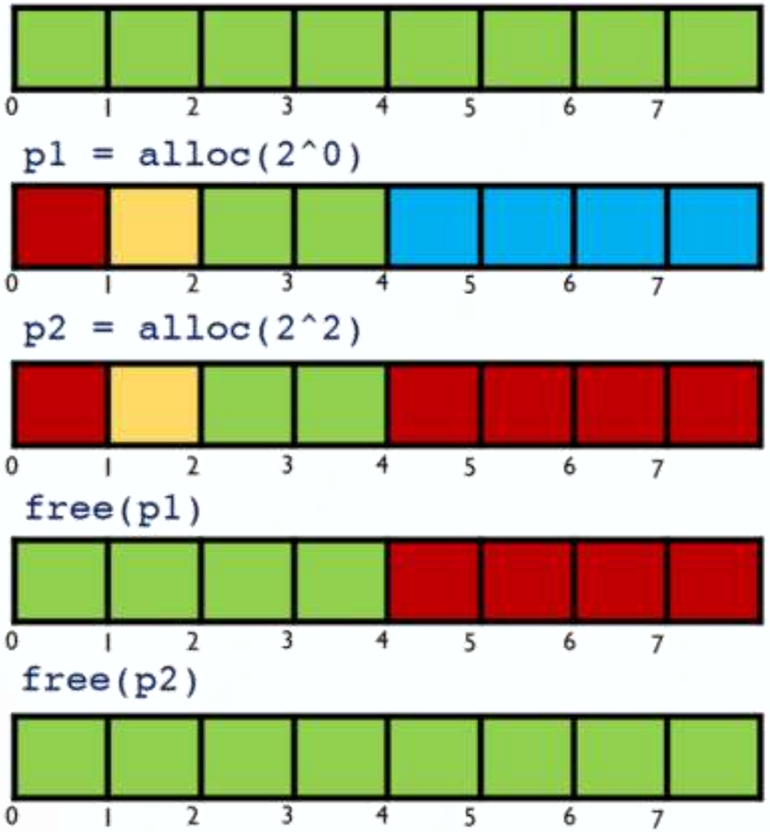


- Recursively divide larger *Blocks* until we reach a suitable *Block*
  - "Until" meaning: Large enough to fit, but further splitting would be too small
- Insert "*Buddy*" *Blocks* into *Freelists*
  - The Addresses of the *Buddy Pair* only differ by one bit
    - Given a currently **`free()`**d *Block*, efficient finding of its *Buddy Block*
- Upon **`free`**, recursively *Coalesce Block* with *Buddy* if *Buddy* is *Free*

Dynamic Memory

Buddy Allocation

Example:



freelist[3] = {0}, freelist[2] = { }, freelist[1] = { }, freelist[0] = { }

Split order 3 *Block*… **Buddy Block** … then insert *Buddy Blocks* into *Freelists*

freelist[3] = { }, freelist[2] = {4}, freelist[1] = {2}, freelist[0] = {1}

Buddy Block **Buddy Block**

Order 2 *Block* available, directly *Allocate*

freelist[3] = { }, freelist[2] = { }, freelist[1] = {2}, freelist[0] = {1}

p1 & *Buddy* are *Free; Coalesce* → newly formed *Block* & *Buddy* are *Free*; *Coalesce*

freelist[3] = { }, freelist[2] = { }, freelist[1] = {2}, freelist[0] = {0,1}

freelist[3] = { }, freelist[2] = { }, freelist[1] = {0,2}, freelist[0] = { }

freelist[3] = { }, freelist[2] = {0}, freelist[1] = { }, freelist[0] = { }

Main Block & *Freed Buddy* are *Free; Coalesce*

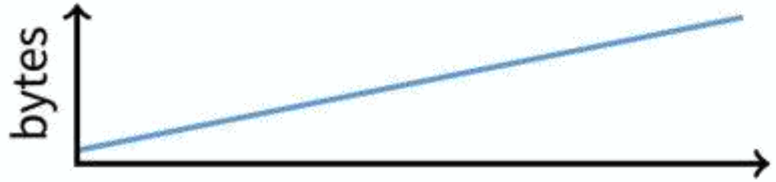
freelist[3] = { }, freelist[2] = {0,4}, …

freelist[3] = {0}, freelist[2] = { }, …

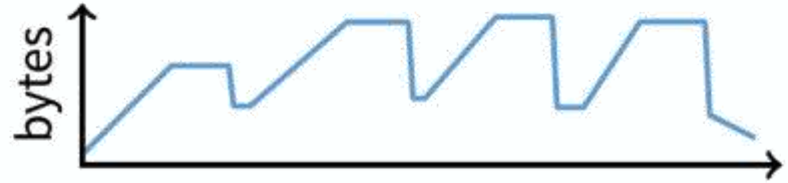
Known Patterns of Real Programs

Most Programs exhibit 1, 2, or all 3 of the following **`alloc`** / **`free`** patterns:

- *Ramps*
  - Accumulate data monotonically over time

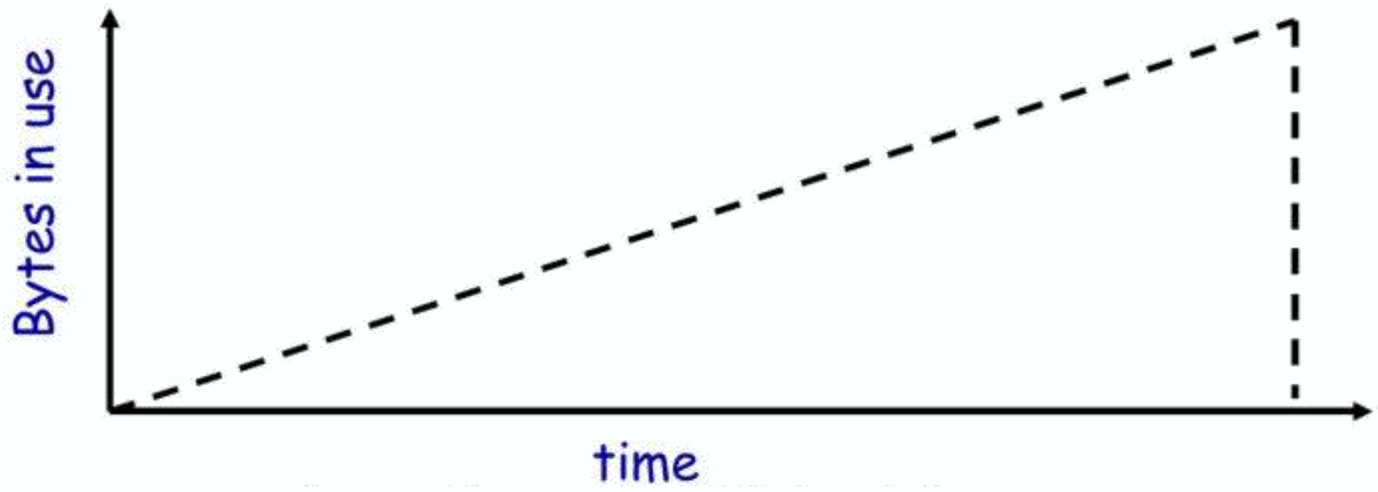


- *Peaks*
  - *Allocate* many objects, use briefly, then *Free* all



- *Plateaus*
  - *Allocate* many objects, use for a long time

Pattern 1: *Ramps*

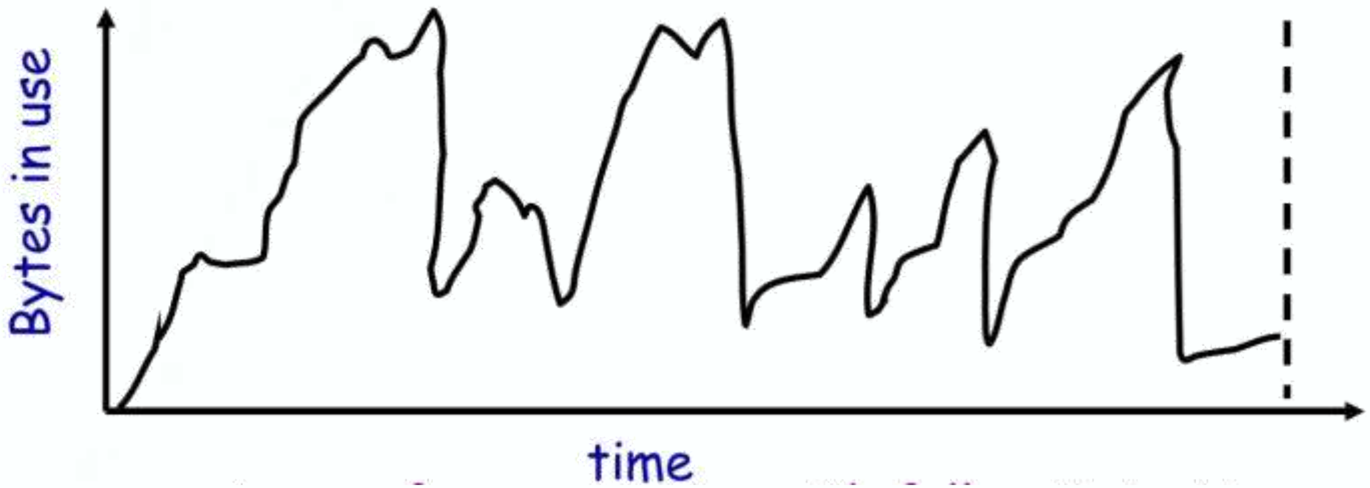


Trace from an LRU Simulator

- In effect: *Ramp* ≡ no **`free`**
  - Implications for *Fragmentation*?
  - What happens if you evaluate *Allocator* with *Ramp* Programs only?

Dynamic Memory

Pattern 2: *Peaks*



Trace of `gcc` compiling with full optimization

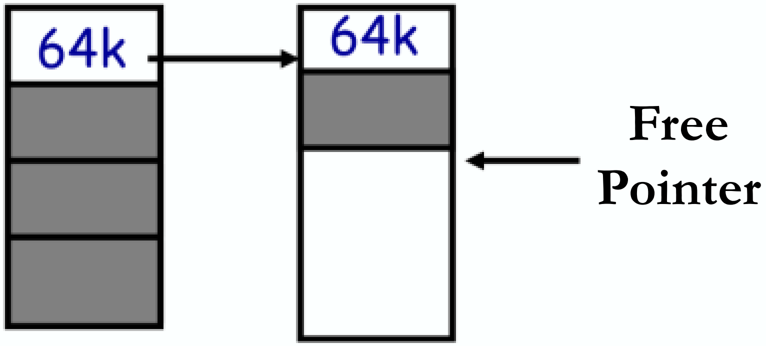
- *Peaks* : *Allocate* many objects, use briefly, then **`free`** all at once
  - *Fragmentation* a real danger
  - What happens if *Peak Allocated* from Heap with a single Contiguous *Memory* area?
    - Think about *Locality* of longer-term Allocated *Memory* that took place before and after *Peak*
    - E.g. if we interleave *Peak* & *Ramp*, interleave two different *Peaks*, etc.

Arena Allocation

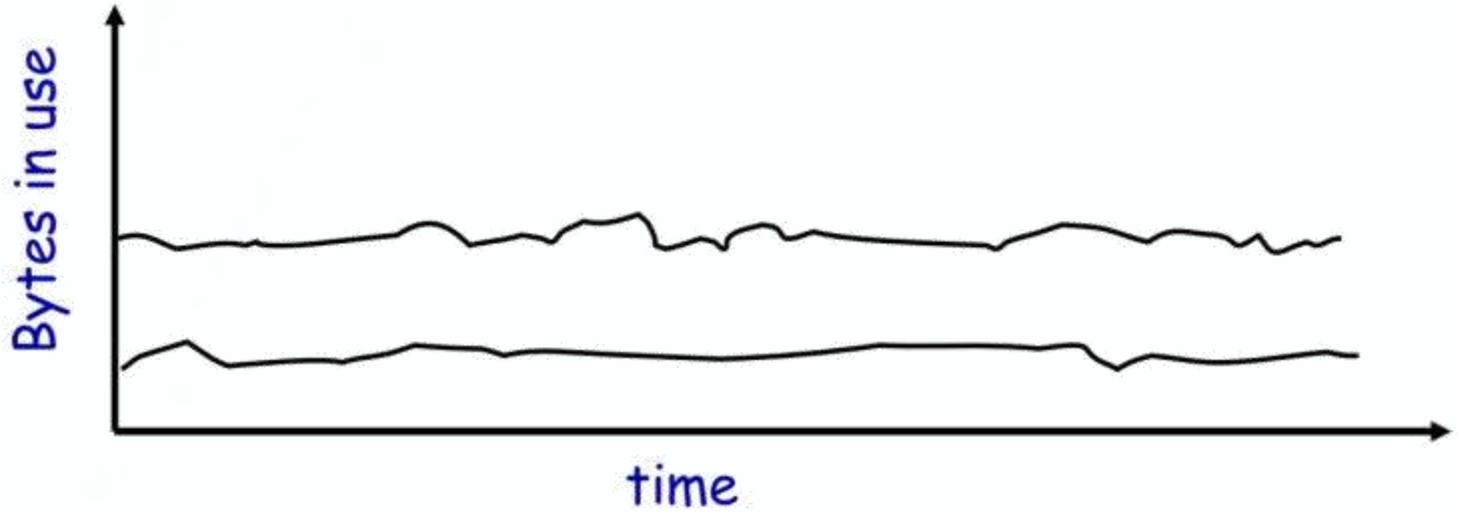
- *Peak* has Phases: *Allocate* a lot, then **`free`** everything
  - Exploit known *Peaks* behavior, change *Allocation* interface:
    - **`alloc`** as before, but only support **`free`** of everything all at once
    - Called "*Arena Allocation*" / "*Obstack*" (Object Stack)

Arena Allocation

- A Linked-List of large chunks of *Memory*
  - Advantages
    - **`alloc`** is just performing Pointer increment
    - **`free`** is "free"
    - No wasted space for *Tags* or *Freelist* Pointers
      - (in Pintos **`threads/malloc.c`**)



Pattern 3: *Plateaus*

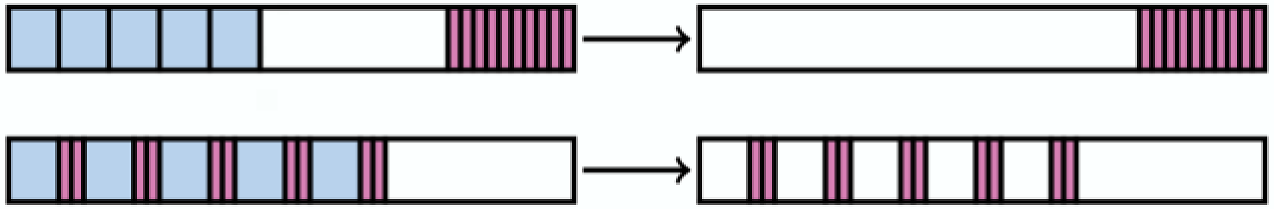


Trace of `perl` running a string processing script

- *Plateaus* : *Allocate* many objects, keep them around for a long time
  - What happens if there is overlap with a *Peak* or different *Process*' *Plateau* ?
    - Again, think about *Locality* of longer-term Allocated *Memory* for 2 different *Processes* that had their *Plateaus* interleaved during subsequent allocations

Fighting *Fragmentation*

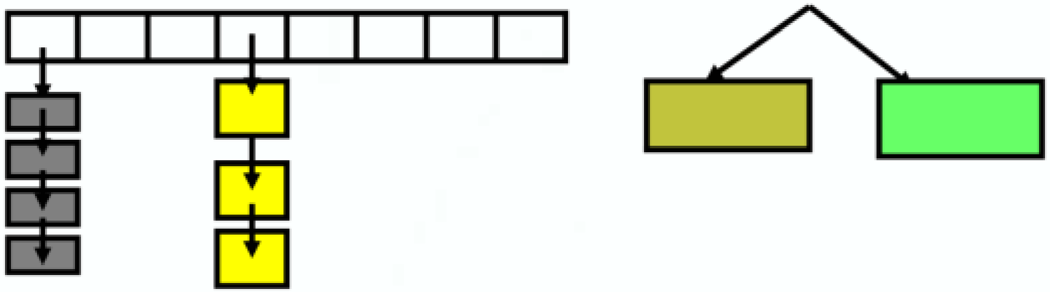
- *Segregation* → Reduced *Fragmentation*
  - *Allocated* at same time ~ *Freed* at same time
  - Different type (/size) ~ *Freed* at different time



- Implementation observations:
  - Programs *Allocate* a small number of different sizes
  - *Fragmentation* at peak usage more important than at low usage
  - Most *Allocations* small (< 10 Words)
  - Work done with *Allocated Memory* increases with size

Dynamic Memory

Simple, Fast, *Segregated Freelists*



- Array of *Freelists* for small sizes, Tree for larger
  - Place *Blocks* of same size on same *Page*
  - Maintain count of *Allocated Blocks*: If goes to zero, can return *Page* to system
- Pros: a) *Segregated* sizes, b) No size *Tag*, c) Fast & small **`alloc`**
- Cons: a) Worst case waste: 1 *Page* per size even w/o **`free`**,
  b) After pessimal **`free`**: Waste of 1 *Page* per object

Note: TCMalloc (Ghemawat) is a well-documented **`malloc`** like this

Dynamic Memory

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 *Direct Memory Access (DMA)*

Slab Allocation (Bonwick)

- Optimizes for the above case
  - A *Slab* is **multiple** *Pages* of **Contiguous *Physical Memory***
  - A *Slab Cache* contains one or more *Slabs*
  - Each *Slab Cache* (and therefore each of its contained *Slabs*) stores only one kind of object (/class), i.e. has fixed element size
- Each *Slab* can be *Full*, *Empty*, or *Partial*
  - But is Contiguously treated

Slab Allocation

Example: Need a new **`task_struct`**

- Look in the **`task_struct`** *Slab Cache*
- If there is a *Partial Slab*, pick a free **`task_struct`** slot in that
- Else, use an *Empty Slab*
  - or may need to *Allocate* new *Slab* for our *Slab Cache*

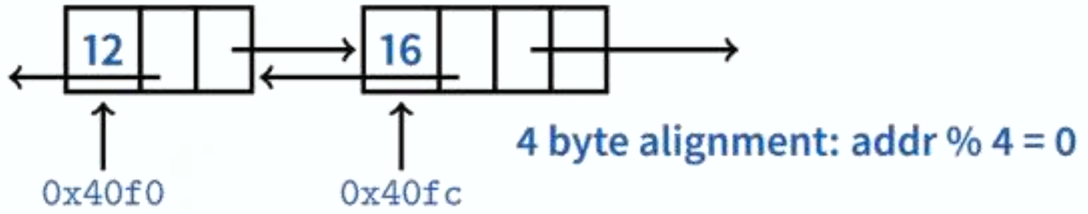
- *Free* Memory management: *Bitmap*
  - **`alloc`**: Set bit and return slot in *Slab*, **`free`**: Clear bit

- Advantages: Speed, and no *Internal Fragmentation*

Note: Used in FreeBSD and Linux, implemented on top of *Buddy Page Allocator*

Dynamic Memory Allocation & Space Overhead

- *Minimum Allocatable Size* determined by *Freelist Bookkeeping* and *Alignment*
  - *Implicit Freelist*: Must store size of *Block* (to infer offset of next Block)
  - *Explicit Freelist*: Must store Pointers to **`next`** (and **`previous`**) *Blocks*



  - *Allocator* doesn't know types
    - Must *Align Memory* to conservative boundary
      - Otherwise *Instruction* can walk off a *Page*!
    - Implementation **has to** return a Pointer *Aligned* to the largest possible ***Machine*** *Alignment* (**`__BIGGEST_ALIGNMENT__`** macro)

https://linux.die.net/man/3/malloc
The **`malloc()`** function returns a pointer to the Allocated Memory that is suitably *Aligned* for any built-in type.

Dynamic Memory

Getting more Space from OS (implementing **`malloc`**)

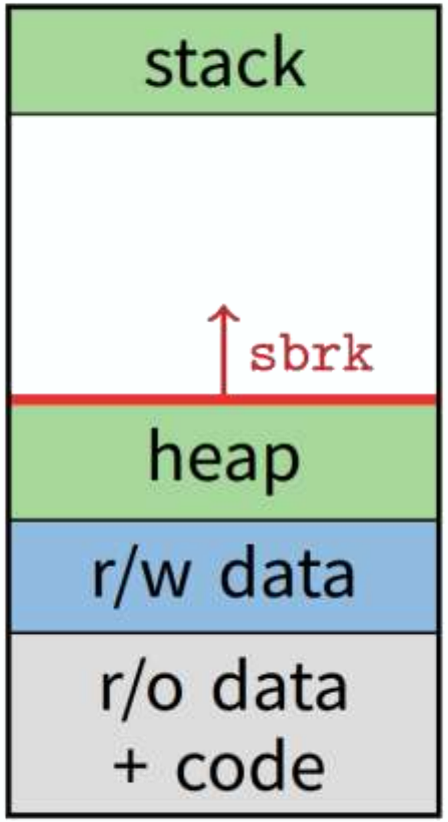
- Example 1: On Unix, can use **`sbrk`** and **`brk`**

`int brk(void *p);`

- Move the Program **`break`** to address **`p`**
- Returns **`0`** if successful, **`-1`** otherwise

`void *sbrk(intptr_t n);`

- Increment the Program **`break`** by **`n`** bytes
- Returns the location of the previous Program **`break`**
- If **`n`** is **`0`**, then return the current location of the Program **`break`**
- Returns **`0`** if successful, **`(void*)-1`** otherwise



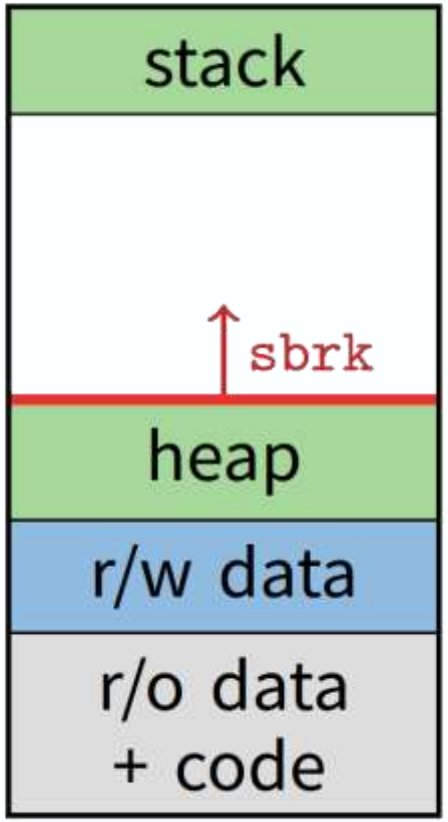
Dynamic Memory

Getting more Space from OS (implementing `malloc`)

- Example 1: Use **`sbrk`** to activate a new Zero-filled *Page*

```
/* add nbytes of valid virtual address space */
void *malloc(size_t nbytes) {
  void *p = sbrk(nbytes);
  if (p == (void *) -1)
    error("Virtual Memory exhausted");
  return p;
}
```

- For **large** *Allocations*, **`sbrk`** a bad idea
  - Can't return *Memory* to the system:
    - **`free()`** could do **`sbrk(-nbytes)`** but this just allows reusing *Blocks* later by the *Process*, does not return them to the OS (can cause *Memory* pressure)
    - Also, assumes that **`break`** will be modified incrementally with consistency

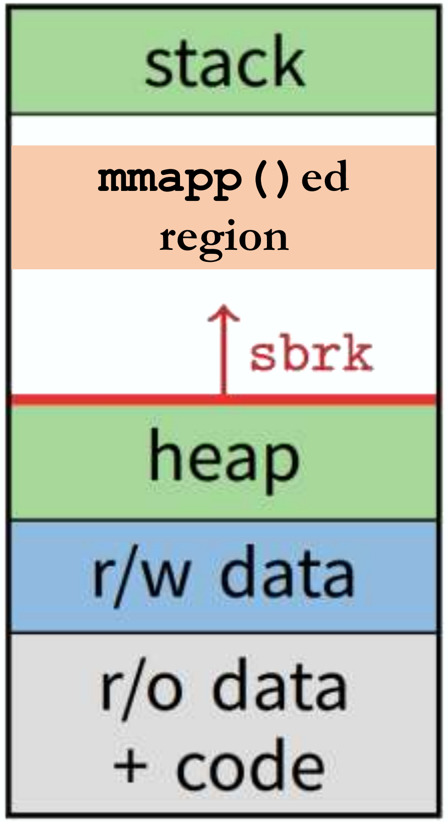


Getting more Space from OS (implementing **`malloc`**)

- Example 2: Use *Virtual Memory Mapping* **`mmap`**

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

- Can create a new *Anonymous Virtual Address* Mapping in the *Virtual Address Space* of the calling *Process*
- **`p`**: Starting *Virtual Address* for the new Mapping (if NULL, the Kernel chooses the *Virtual Address* at which to create the Mapping)
- **`n`**: Length of the Mapping
- On success, returns *Virtual Address* of the Mapped area




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

- Deletes the Mappings for the specified *Virtual Address* range

Getting more Space from OS (implementing **`malloc`**)

- Example: Use *Virtual Memory Mapping* **`mmap`**

```
void* malloc(size_t n) {
  if (n == 0) return NULL;

  size_t* p = mmap(NULL, n + sizeof(size_t),
                   PROT_READ | PROT_WRITE,
                   MAP_PRIVATE | MAP_ANONYMOUS,
                   -1, 0);

  if (p == (void *) -1) return NULL;

  *p = sizeof(size_t) + n; // Store size in header
  ++p; // Advance from header to payload

  return p;
}
```

```
void free(void *_p) {
  if (_p == NULL) return;

  // Advance backwards
  // from payload to header
  size_t* p = (size_t*)_p;
  --p;

  munmap(_p, *p);
}
```

Note: **`malloc()`**'s **`MMAP_THRESHOLD`** is 128 KB by default, but is adjustable using **`mallopt()`**

CS-446/646

Time for Questions !