Memory Management

Today

- Basic memory management
- Swapping
- Kernel memory allocation

Next Time

- Virtual memory

Memory management

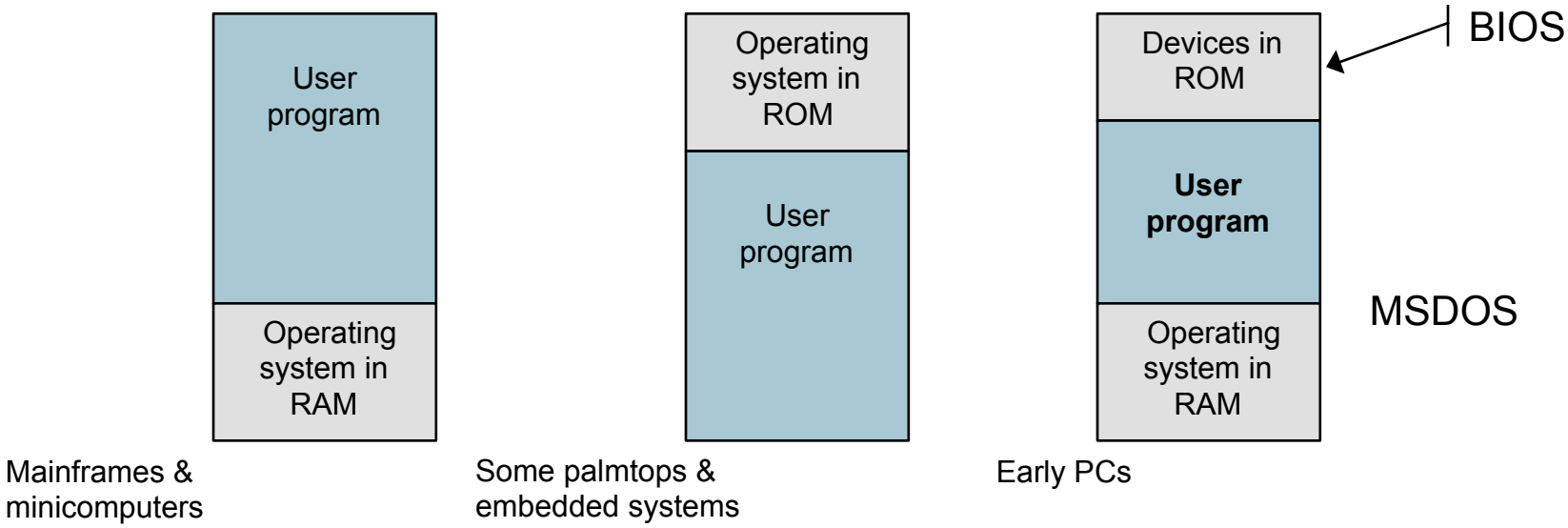
- Ideal memory for a programmer – large, fast, non-volatile *and* cheap – not an option

- Best alternative → memory hierarchy
  - Small amount of fast, expensive memory – cache
  - Some medium-speed, medium price main memory
  - Gigabytes of slow, cheap and non-volatile disk storage

- To handle the memory hierarchy – memory manager
  - Allocates scarce resource among competing requests to maximize (memory) utilization and system throughput
  - Offers a convenient abstraction for programming
  - Provides isolation between processes

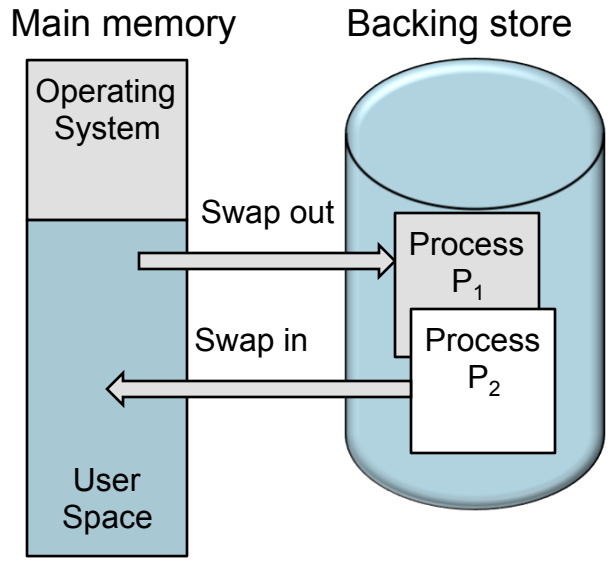
Basic memory management

- Simplest memory abstraction – no abstraction at all
  - Early mainframes (before ‘60), minicomputers (before ‘70) and PCs (before ‘80)
  - `MOV REG1, 1000 #REG1` ← `(physical memory 1000)`
  - Logically, only one program running at a time *Why?*
  - Still here, some alternatives for organizing memory



Multiprogramming and swapping

- Even with no memory abstraction, it is possible to run multiple programs concurrently
- Switching processes with swapping
  - Simple
  - Bring each process entirely
  - Move another one to disk
  - Process swapped back in restarts from where it was

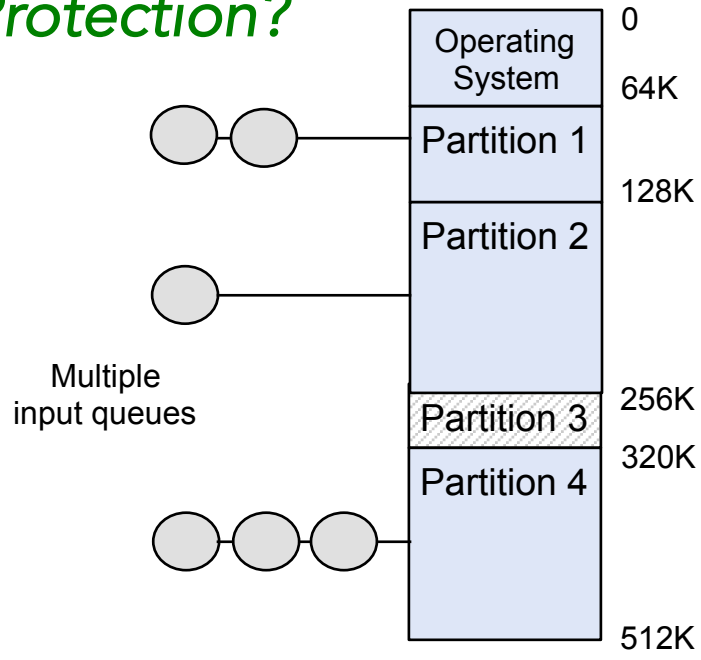


- CTSS – a uniprogrammed swapping system
- Clearly, moving the whole thing out is not *too* good for performance

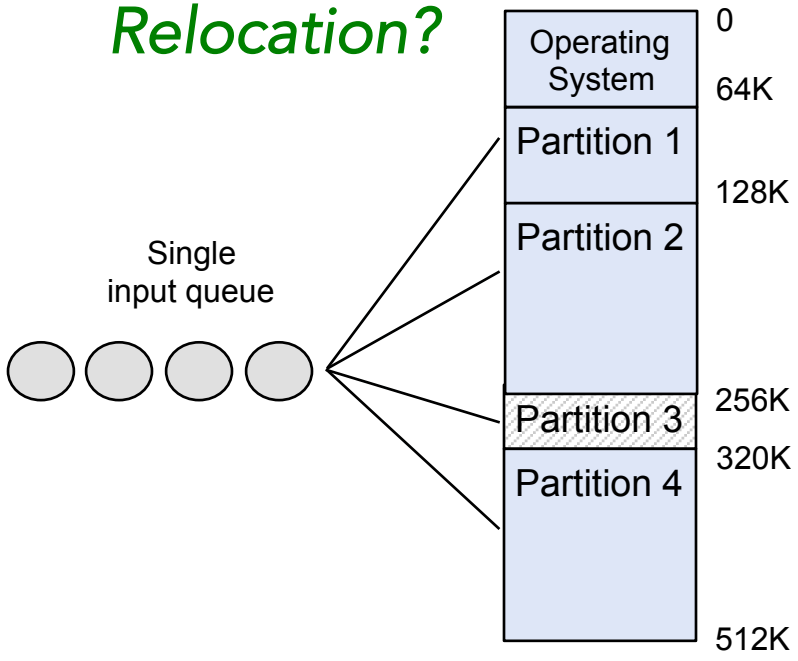
Multiprogramming with fixed partitions

- With some additional hardware, avoid swapping
- IBM OS/360
  - Split memory in *n* parts (possible != sizes)
  - Single or separate input queues for each partition
  - ~MFT: Multiprogramming with Fixed number of Tasks

Protection?

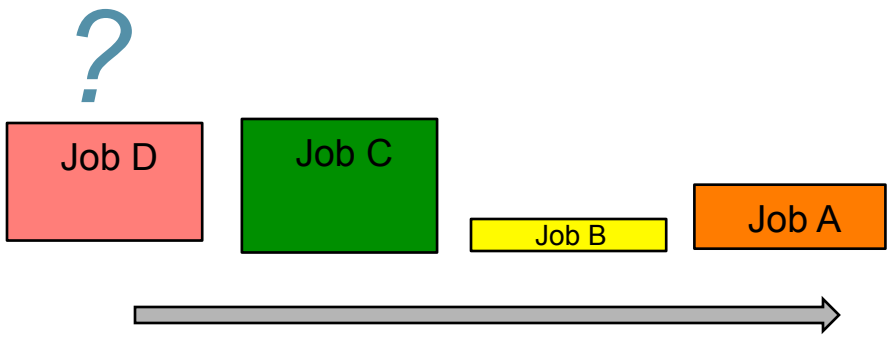


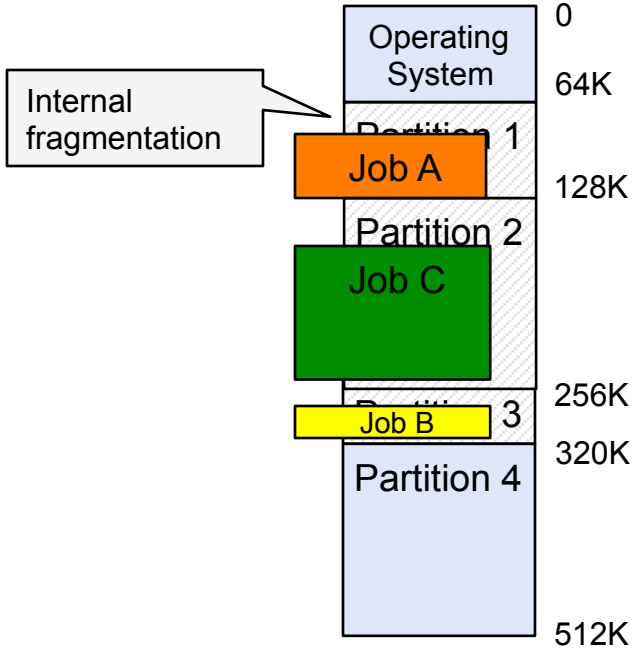
Relocation?



Fragmentation

- A pervasive problem – fragmentation
  - Internal fragmentation – the available portion is larger than what was requested
  - External fragmentation – two small partitions left, but one big partition needed!



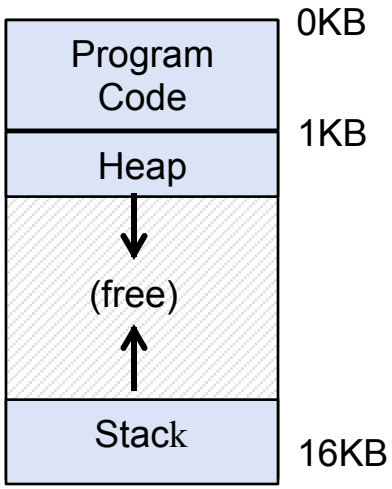


A new abstraction – Address space

- Address space – all the memory a process can reference

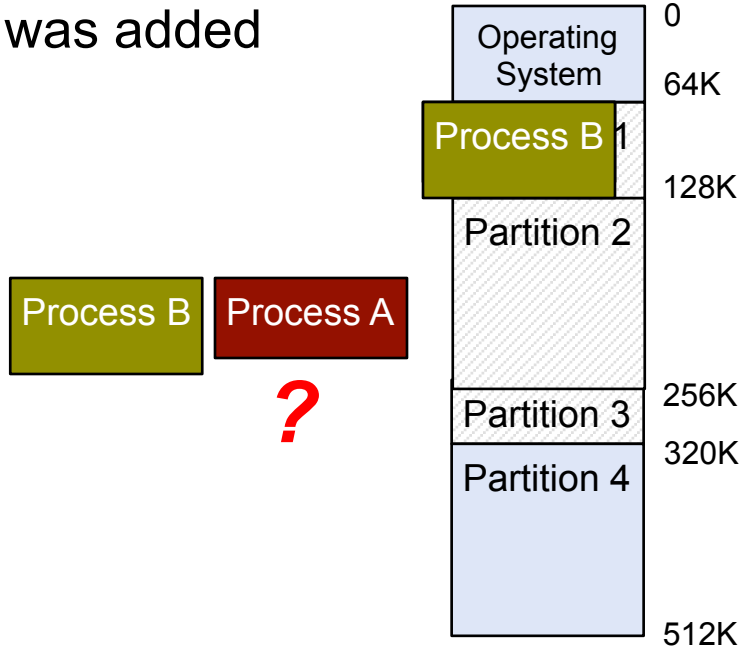
```
MOV REG1, ????
```

- What does it include
  - Code – static, so it won't change while the program runs
  - Stack – to keep track of where it is in the function call chain
  - Heap – for dynamically allocated memory
  - We placed them opposite so they can both grow (but this is just a convention)



Multiprogramming – relocation

- With absolute physical memory – the program must always run in the exact same location
  - What a little JMP *X* can do!
- IBM 360 stop-gap solution – Change it at loading time
  - Static relocation
  - If a program was reloaded starting at 131,072, the constant 131,072 was added to *every* address at load time
  - Not particularly fast!



Multiprogramming – Protection

- What stops a process from writing outside its allocated memory?
- IBM 360 – Check it for each reference
  - Split memory into 2KB blocks
  - Each block was assigned a 4b protection key kept in CPU
  - The PSW (program status word) also kept a 4b key
  - OS trapped any process trying to access memory with protection key != the PSW key

Virtual addresses

- Memory is easier to manage if processes use virtual addresses
  - … independent of location in physical mem where the data is
  - … translated by HW into physical addresses (with OS help)
- *Every address you have ever seen is virtual*

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
   printf("Loc. of code: %p\n", (void *) main);
   printf("Loc. of heap: %p\n", (void *) malloc(1));
   int x = 1;
   printf("Loc. of stack: %p\n", (void *) &x);
   return x;
}
```

```
Loc. of code: 0x10e355e60
Loc. of heap: 0x7fe4334000e0
Loc. of stack: 0x7fff518aaac4
```

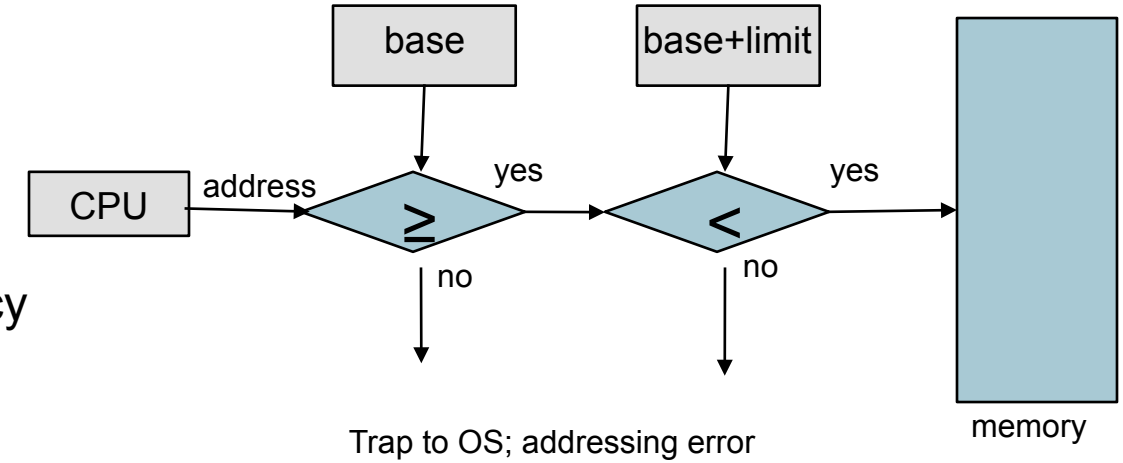
*R. And A. Arpaci-Dusseau, Three Easy Pieces, v0.6

Virtual memory system – Goals

- Transparency
  - Programs should not know that memory is virtualized; the OS +HW multiplex memory among processes behind the scenes
  - … (there's more to transparency, next lecture!) …
- Efficiency
  - Both in time and space; not making them too slow and efficiently using physical memory
- Protection
  - Isolating the address spaces of processes from each other; i.e., a process should not be able to access or affect the memory of any other process or the OS itself

Multiprogramming with variable partitions

- A simple solution – partition physical memory into dynamically allocated blocks
  - CDC 6600 and Intel 8088
- Base and limit values
  - Address locations + base value → physical address
  - Ideally, base and limit registers can only be modified by OS
  - A disadvantage – Comparisons can be done fast but additions can be expensive



- Transparency
- Protection
- Efficiency

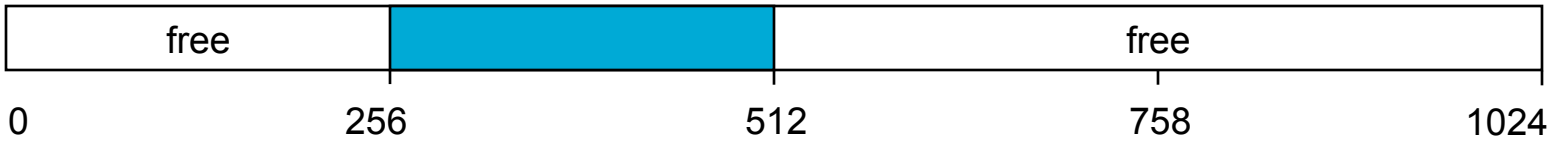
And now a short break ...

Memory management

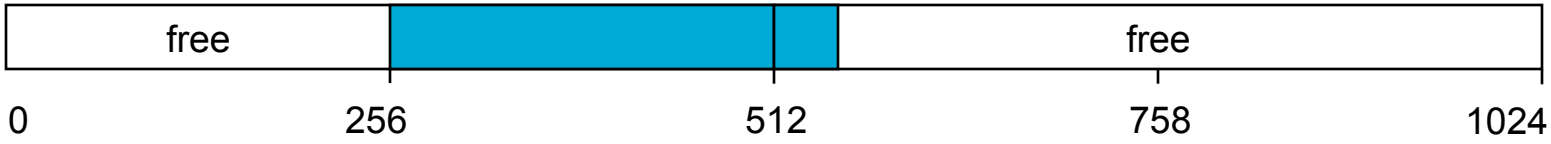
- With dynamically allocated memory
  - OS must keep track of allocated/free memory
  - Two general approaches - bit maps and linked lists
- Bit maps
  - Divide memory into allocation units, track usage with a bitmap
  - Design issues - Size of allocation unit
    - The smaller the size, the larger the bitmap
    - The larger the size, the bigger the waste
  - Simple, but slow – find a big enough chunk?
- Linked list of allocated or free spaces
  - List ordered by address
  - Double link will make your life easier
    - Updating when a process is swapped out or terminates

Picking a place – different algorithms

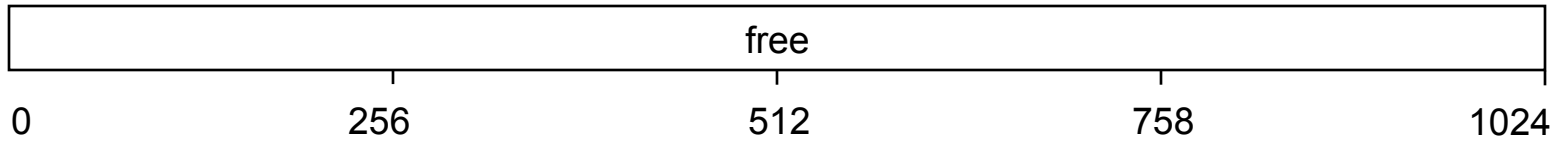
- Assume a chunk of memory, maintained somehow
  - A free list perhaps
    - Head ➔(Address: 0, Len: 256) ➔(Address: 512, Len: 512) ➔ NULL



    - A request for 10B: `malloc(10)`
    - Head ➔(Address: 0, Len: 256) ➔(Address: 522, Len: 1024) ➔ NULL

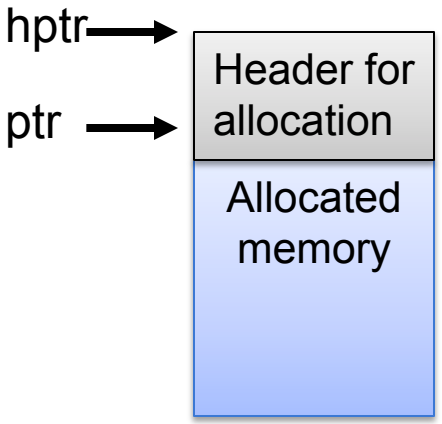


    - Free 266: `free(266)`
    - Head ➔(Address: 256, Len: 266) ➔(Address: 0, Len: 256) ➔(Address: 522, Len: 1024) ➔ NULL ?!?!



Tracking the size of allocated blocks

- Not that free does not take size
  - Keep extra info in a header block
    - Size (this for sure)
    - Additional pointers for faster deallocation
    - Magic number for additional integrity checks




```
typedef struct _header_t {
        int size;
        int magic;
} header_t;

void free(void *ptr) {
   header_t *hptr = (void *) ptr - sizeof(header_t);
   ...
   assert(header_t->magic == MAGICN);
   ...
}
```

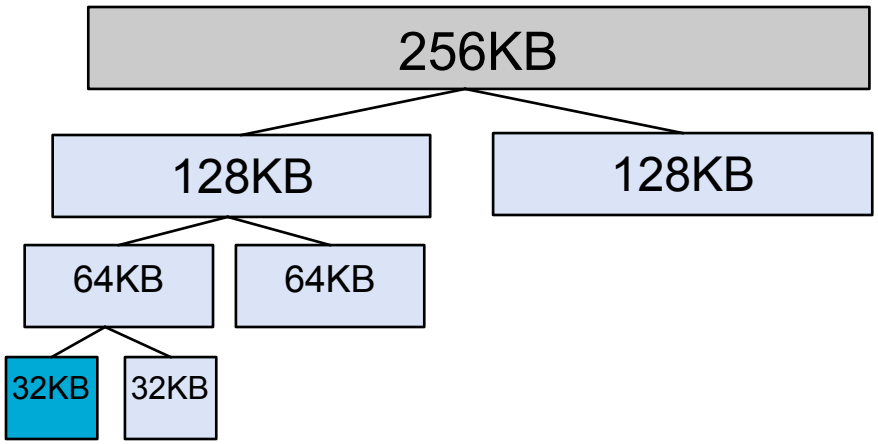
Picking a place – basic algorithms

- If list of processes & holes is ordered by addresses, different ways to get memory for a new processes …
  - First fit – simple and fast
  - Next fit - ~ First fit but start where it left off
    - Slightly worst performance than First fit
  - Best fit – try to waste the least but wastes more in tiny holes and have to search through the list
  - Worst fit – try to “waste” the most (easier to reuse)
    - Not too good either

- Speeding things up
  - Two lists (free and allocated) – slows down de-allocation
  - Order the hole list – first fit ~ best fit
  - Use the same holes to keep the list

Picking a place – other algorithms

- Buddy allocation
  - Coalescing is critical; buddy allocator
  - Free memory is a 2^N size block
    - When requesting a block (e.g., 23KB), split in half until is big enough
    - When the block is freed, check if buddy is free & coalesce them
      - Finding your buddy is easy – the address of each buddy pair differs only by a single bit!
    - Repeat

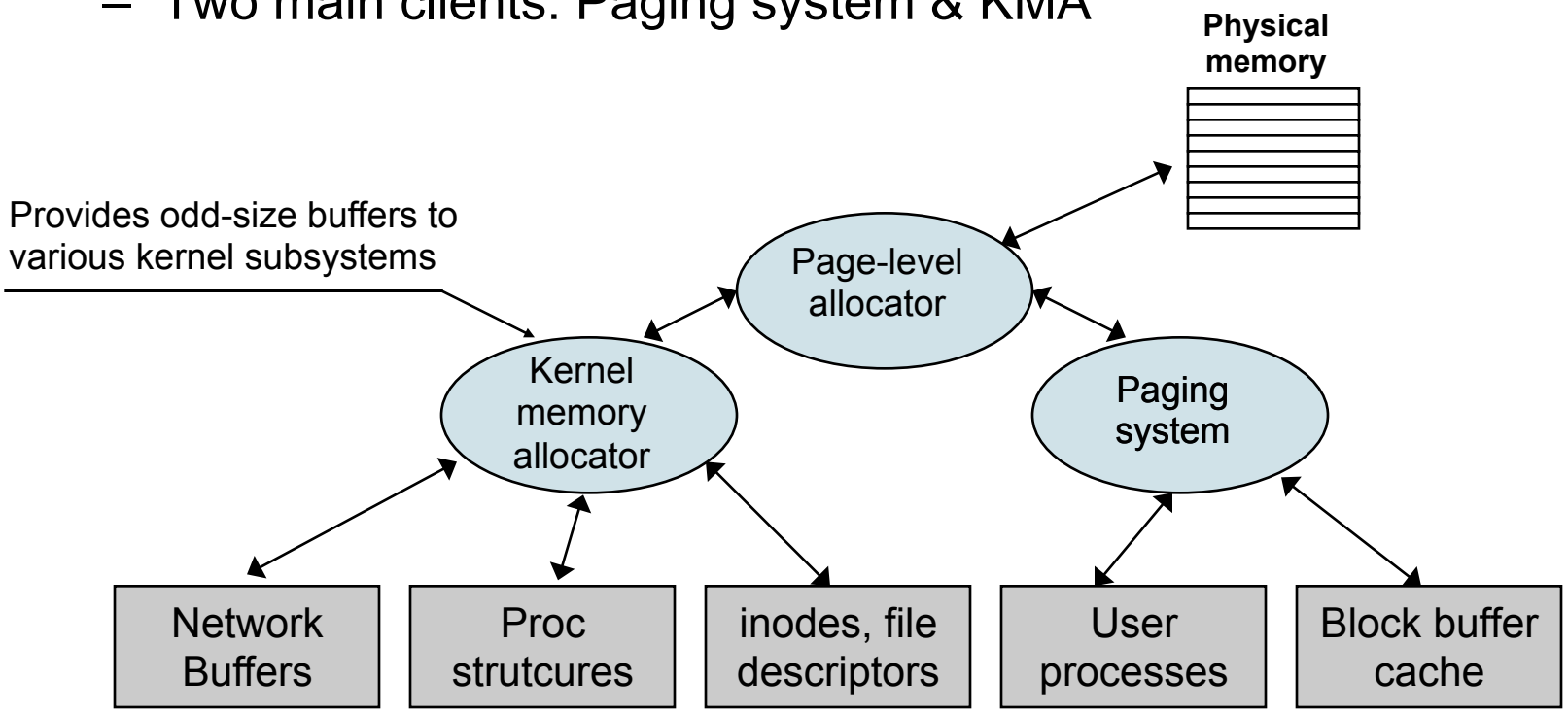


Picking a place – other algorithms

- Segregated list
  - List of commonly used hole sizes - allocation is quick, merging can be expensive
  - Example – slab allocator (design for Solaris)
    - Keep blocks for locks, file-systems i-nodes, etc
  - *More on this in a bit*

Kernel memory allocation

- Most OS manage memory as set of fixed-size pages
- Kernel maintains a list of free pages
- Page-level allocator has
  - Two main routines: e.g `get_page()` & `freepage()` in SVR4
  - Two main clients: Paging system & KMA



Kernel memory allocation

- KMA's common users
  - The pathname translation routine
  - Proc structures, vnodes, file descriptor blocks, …
- Since requests << page → page-level allocator is inappropriate
- KMA & the page-level allocator (three models)
  - Pre-allocates part of memory for the KMA
  - Allow KMA to request memory
  - Allow two-way exchange with the paging system

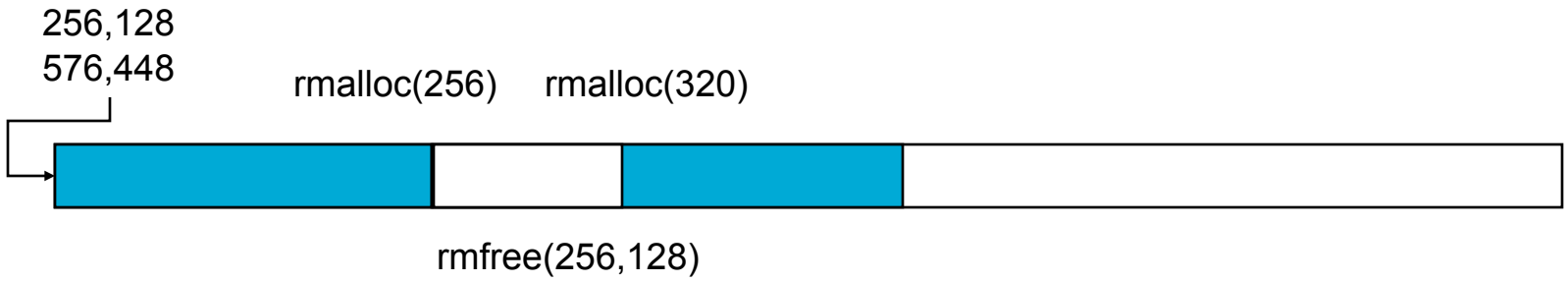
Kernel memory allocation – evaluation

- Memory utilization
  - Physical memory is limited after all
  - Major cause of wasted memory – fragmentation
- Speed
  - It is used by various kernel subsystems (e.g. interrupt handler)
  - Better be fast – both in avg and worst case
- Simple API
  - A simple free & malloc is nice but you can't release partial memory with too simple of a malloc
- Allow a two-way exchange with page-level allocator

KMA – Resource map allocator

- Resource map – a set of <base, size> pairs
- Initially the pool is described by a single pair
- … after a few exchanges … a list of entries per contiguous free regions
- Allocate requests based on
  - First fit, Best fit, Worst fit
- A simple interface

```
offset_t rmalloc(size);
void rmfree(base, size);
```

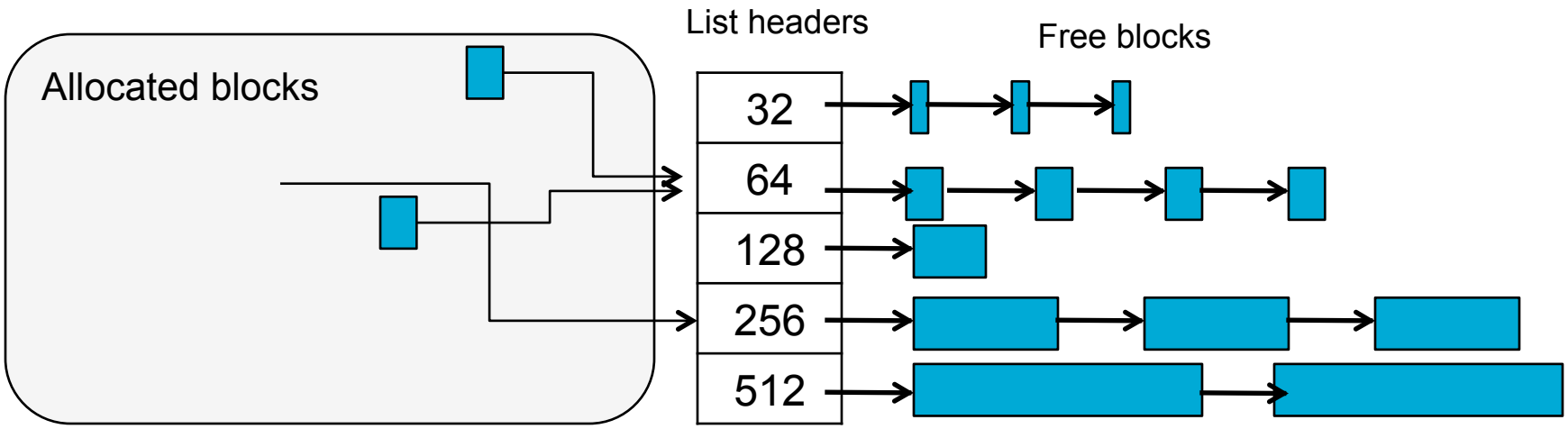


Resource map allocator

- Pros
  - Easy to implement
  - Not restricted to memory allocation
  - It avoid waste (although normally rounds up requests sizes for simplicity)
  - Client can release any part of the region
  - Allocator coalesces adjacent free regions
- Cons
  - After a while maps ended up fragmented – low utilization
  - Higher fragmentation, longer map
  - Map may need an allocator for its own entries
  - To coalesce regions, keep map sorted – expensive
  - Linear search to find a free region large enough

KMA – Simple power-of-two free list

- A set of free lists
- Each list keeps free buffers of a particular size (2^x)
- Each buffer has one word header
  - Pointer to next free buffer, if free or to
  - Pointer to free list (or size), if allocated



KMA – Simple power-of-two free list

- Allocating(size)
  - allocating (size + header) rounded up to next power of two
  - Return pointer to first byte *after* header
- Freeing doesn't require size as argument
  - Move pointer back header-size to access header
  - Put buffer in list
- Initialize allocator by preallocating buffers or get pages on demand; if it needs a buffer from an empty list …
  - Block request until a buffer is released
  - Satisfy request with a bigger buffer if available
  - Get a new page from page allocator

Power-of-two free lists

- Pros
  - Simple and pretty fast (avoids linear search)
  - Familiar programming interface (malloc, free)
  - Free does not require size; easier to program with
- Cons
  - Rounding means internal fragmentation
  - As many requests are power of two and we loose header; a lot of waste
  - No way to coalesce free buffers to get a bigger one
  - Rounding up may be a costly operation

Coming up ...

- Virtual memory in all its beauty

Object caching in KMA

- The cost of initializing and destroying an object can exceed the cost of allocating/freeing memory for it*

Comparing the new (with caching) and old allocators in a SPARCStation-2 running SunOS 5.4

Stream Head Allocation + Free Costs (µsecs)			
Allocator	Contr/Destr	Mem allocation	Other init.
Old	23.6	9.4	1.9
New	0.0	3.8	1.9

- Basic model for caching
  - Allocate: if there's an object in cache, use it, else allocate
  - Free: return the object to the cache
  - Reclaim from cache: take an object from the cache, destroy and free memory

Jeff Bonwick, The Slab Allocator: An Object-Caching Kernel Memory Allocator, USENIX 1994

Object caching in KMA

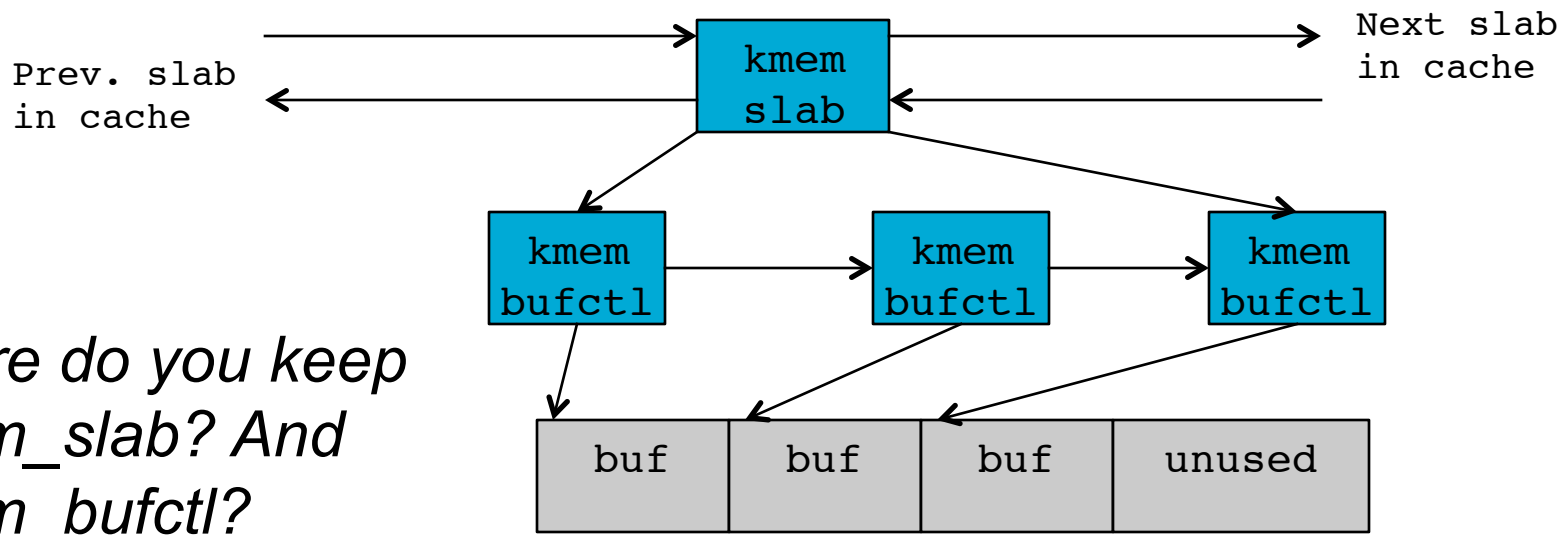
- You can support caching outside KMA; why not?
  - Privately-managed caches cannot handle the tension between an object cache (wants to keep memory) and the rest of the system (wants it back) sensibly
  - … unable to use any accounting or debugging mechanism provided by KMA
  - … result in multiple instances of the same solution, increasing kernel size
- What is the right interface for such an allocator?
  - Descriptions of objects (names, size, constructors …) belong in the client
  - Memory management policies belong in kma

Object caching in KMA – API

```
struct kmem_cache *kmem_cache_create(
   char *name,
   size_t size,
   int align,
   void (*constructor) (void *, size_t),
   void (*destructor) (void *, seize_t));

struct kmem_cache *kmem_cache_alloc(
  struct kmem_cache *cp,
  int flags);

void kmem_cache *kmem_cache_free(
  struct kmem_cache *cp,
  void *buf);

void *kmem_cache_destroy(
  struct kmem_cache *cp);
```

Slab layout

- Solaris' slab allocator
  - A slab – 1+ pages carved into equal-size chunks, with a reference count of allocated chunks
- Content of each slab managed by a kmem_slab
  - Slab's linkage, reference count, list of free buffers
  - Each buffer managed by a kmem_bufctl
    - Freelist linkage, buffer address, back pointer to controlling slab



Where do you keep kmem_slab? And kmem_bufctl?