Today, we study how memory is managed
   Part architecture: what a modern processor can do
   Part tradition: we've always done it that way.

"what you don't know, that can't kill you (but can kill your performance)"

"CPU speed is a baldfaced lie."

"gdb is a baldfaced lie."

Goal: explain the mechanisms by which we achieve performance, and why we need them.
The OS's view of memory

When the process runs, hardware memory mapping maintains its view of the world.

All it sees are **logical pages**.

It has no knowledge of physical underpinnings. When the OS runs, e.g., during a system call, it runs using the **physical** memory map.

It sees processes as maps.

It must convert between logical and physical
Example: how a mutex works

When you do a `pthread_mutex_init`, the kernel translates your logical address to a physical address. Stores the physical address in a `mutex descriptor` inside the kernel.

Creates a `mutex queue` inside the descriptor.

When you block on a `pthread_mutex_lock`, the kernel locates the queue via the stored physical address of your mutex.
A correspondence is a data structure that associates things in two arrays at the same offset.

Example: frame descriptors

- Small array contains the descriptors (structs)
- Frames are an array of larger blocks

Frame at offset n corresponds to descriptor at offset n.
Understanding correspondences

Offset in descriptor space

Descriptor for x in at x >> 13 in descriptor array
Correspondences as a search problem

Given some kind of key -- typically an integer -- locate the information associated with the key

Examples:

key=Process ID, information=Process Control Block
key=device number, information=driver storage
key=logical address, information=physical address
Arrays, Hashes, and Caches

Monday, November 2, 2015  2:58 PM

Correspondences can be:
  Arrays (if small)
  Hashes (if sparse and large)
  Caches (if dense and large)
If a correspondence is huge, it can be **cached**: 
Put parts of it into memory.
Remember what parts (with part descriptors!)
If you try to look up a non-resident part:
  flush a resident part to disk.
  load the non-resident part in its place.
This is exactly what swapping does.
Understanding cacheing
Sometimes, a correspondence is sparse, i.e., most elements are missing
   Example: process IDs
       65536 possible IDs
       only 100 or so running at a time.
   Like a correspondence, but
   there are holes
Solution: hashing
Set aside an array with 3x the number of keys
Define a hash function $h(k)$ that
  maps from keys to array elements.
  many-to-one
E.g., for process ids,
  array size is 1000
  $h(p) = p \% 1000$
Deal with **collisions** in some way, i.e., cases where
  $h(p) == h(q)$ and $p != q$
Typical way: extra parameter to $h$:
  $h(p,0)$: regular hash function
  $h(p,1)$: second try
  $h(p,2)$: third try...
A **dense** space is one in which "almost every key is valid".
   Example: an array: key=array offset.
   Example: I/O descriptor table: key=file descriptor

A **sparse** space is one in which "few available keys are valid".
   Example: process table. Key=PID
   Example: page map. Key=physical address.
Basic principle of OS design:
   If a correspondence is large and sparse, use a hash.
   If a correspondence is large and dense, use a cache.
   If a correspondence is small, use an array.
Influences upon memory addressing

**Fragmentation:** the fact that allocation patterns create holes in memory space.

**Locality:** the fact that most programs access memory with localized patterns of access.

**Security:** the fact that processes should not be able to see other processes' space.
Correspondence types:
segment --> segment descriptor
page number --> page descriptor
frame number --> frame descriptor
page number --> frame number
byte address --> byte contents
So far, we have a very simple model of memory mapping
logical pages map to physical frames
addresses in the logical page correspond to physical frame addresses.

\[ \text{pbase to pbound-1: page addresses} \]
\[ \text{fbase to fbound-1: frame addresses} \]
\[ \text{paddr: address offset in page} \]
\[ \text{faddr: address offset in frame} \]

The address equation:
\[ \text{paddr-pbase} = \text{faddr-fbase} \]
or
\[ \text{faddr} = (\text{paddr-pbase}) + \text{fbase} \]
Generalization of simple memory mapping

Each logical address is separated into a page number and an offset

A page table keeps track of the correspondence between pages and frames.
Caveats about page tables

Page length is always a (constant) power of 2
For large processes, page table can only be partly loaded into the processor.
Processor contains a page table cache of part of the page table.
The page table cache

page table

frame pool

page table cache

in memory

in process

user memory
How the OS intervenes

Start with first entry needed in page table cache.
Add entry to page table cache until it fills.
Bump entries out of the page table cache via LRU.
Use dirty bit to manage cache flush.

Underneath the hood:
The page table cache is the physical load image of the MMU.

How it's done:
Everything is happy until a page table cache miss.
Then
a) interrupt the OS
b) load a new page table cache entry.
c) continue
Virtual memory "complicates" the page table. Only part of the pages are resident. The resident set is typically sparse.

So

A regular page table performs poorly When possible, we employ an "inverted page table" (a hardware function), which is (actually) a fancy name for a hash table.
Alas, we normally need **both hashing and caching**

The hash is a correspondence with a virtual pointer pool
The cache puts the hash in the processor when needed.
Cached, inverted page indexing

Tuesday, November 02, 2010  1:40 PM

Virtual Page Hash
Memory

Physical Page Hash
Process

Frame Pool

0x342575

Hash
Arithmetic

Memory

Target Frame
When we started this discussion, I pointed out that simple semantics sometimes have complex implementations. Reason is speed.

**Page table**: a simple idea: map logical to physical.

**Cached page table**: a complex idea: allow a small thing to represent big structure.

**Page table hash**: a simple idea: allow sparse maps.

**Cached page table hash**: a complex idea: allow big sparse maps.
What to do?
Tuesday, November 02, 2010  1:52 PM

We need to implement:
   page maps
   segmentation
   virtual memory
What should the arrows be?
Implementing segments

Note that in the previous diagram, arrows can be:
array indexing
cached arrays
hashed arrays ("inverted indexing")
cached hashed arrays
and more...
You might think that these are all arrays. In fact, they're not. The reason for this is memory and table fragmentation. As processes use frames, they fragment frame space. This leads to fragmentation of the page maps. Which leads to fragmentation in segments.
Dealing with fragmentation

Determine whether space is dense or sparse (local pattern of access vs. non-local pattern of access).
Hash sparse spaces.
Compact dense spaces that can be compacted
Segment table means the same thing no matter what the indices of segments are => can move segments => can compact it.
Malloc memory can't be moved, because the pointers that point to it aren't known at runtime.
Now our story gets very complex

Many different ways to address memory.
All think of a page address as a combination of bit strings.

We need

some way to diagram a scheme
some way to critically analyze a scheme
Segment address points to segment descriptor, which contains base of pages.
Page address determined by offset into pages.
Memory address determined by offset into page.
?, ?, ?: different schemes possible for each kind of addressing.
Three common data structures used in Operating Systems

a. An **array** is a structure with $O(1)$ lookup of dense keys.

A **cache** is a structure with **average case $O(1)$ lookup of dense keys** in limited space.

http://www.cs.tufts.edu/comp/111/examples/Hashes/cache.c

b. A **hash** is a structure with **average case $O(1)$ lookup of sparse keys**.

http://www.cs.tufts.edu/comp/111/examples/Hashes/linear.c
http://www.cs.tufts.edu/comp/111/examples/Hashes/list.c

c. A **lookaside** is a structure with **average case $O(1)$ lookup of recent keys** (and no collision resolution strategy).

http://www.cs.tufts.edu/comp/111/examples/Hashes/lookaside.c
What is an array?
A cache is like an array, but with swap-in, swap-out: Several limited windows into a large virtual array. Each window is one "page" of the cache.
How I draw a cache

Tuesday, November 02, 2010  11:56 AM

How I draw a cache

[Diagram of a cache with labels and arrows indicating flow and connections]
if (sbase<=soffset<=sbound), then
    coffset=cbase+(soffset-sbase)
where
    sbase=beginning of cached page in main store
    soffset=memory location in main store
    sbound=end of cached page in main store + 1
    cbase=beginning of cache page
    coffset=offset in cache page
    cbound=end of cache page + 1
A hash stores sparse keys in a hash table. Can be linked list or linear congruential. **We don't care.**
My picture of a hash

Tuesday, November 02, 2010  12:02 PM
A lookaside is a hash without collision resolution.
  Doesn't contain all values.
  Is opportunistic.
  Can fail to contain something useful.
  Typical use: traversing multiple hashes.
If lookaside(key) is found, use it, else find hash2(hash1(key)) and store (key, hash2(hash1(key))) in lookaside. Lookaside hash has to contain the key so we know if we are correct!
In a modern architecture
  Address consists of segment, page, and offset.
  segment tables and page tables are fragmented.
  frames are fragmented.
  Lots of inverted indexing (hashing) is used in addressing.
  And the hashes are cached!
This means that
  It can take a loooong time to look up a page address,
  because we have to
    load the proper cache for each kind of table.
    do the hash lookup in the cache.
    repeat for the next table....
Enter the TLB

The translation lookaside buffer:

  Is not cached.
  Is not a real hash table.
  Is actually a form of opportunistic cache of a hash.
  Doesn't always work.
  When it works, it speeds up the process.

(The reason it's valuable: principle of locality)