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

"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

The memory mapping unit (MMU) keeps track of what to do:

Part of the processor chip (on modern architectures).

Told the contents of the page table by the OS.

Asks the OS whenever it can't interpret a page entry.
Entities involved in memory management

- Kernel page
- Descriptor table
- MMU
- Special purpose hardware
- Process memory
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. allocates (or simply reuses) a **mutex queue** in the descriptor.

When you block on a `pthread_mutex_lock`, the kernel locates the queue via the **stored physical address** of your mutex, and queues your thread in it.
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.

"Array" is in quotes!

Something is logically an array if given an index, one can compute the value at the index in $O(1)$. 
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
In the OS, **Correspondences can have missing values.** Different strategies are used depending upon whether the array to be coded is "dense" or "sparse".

In a **dense** correspondence, almost every index corresponds to a value.

  Example: logical pages of a process.

In a **sparse** correspondence, very few indexes correspond to a value.

  Example: pages of memory in active use right now. Many are present; few are active.
Arrays, Hashes, and Caches

Monday, November 2, 2015  2:58 PM

Correspondences can be implemented as:
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.
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 lots of 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 $p \neq q$
Typical way: extra parameter to $h$:
  $h(p, 0)$: regular hash function
  $h(p, 1)$: second try
  $h(p, 2)$: third try...

\[ x, y, z \text{ collide} \]
\[ \text{store } x, y, z \text{ in distinct places} \]

\[ h(z, 0) \rightarrow x \rightarrow x \text{ description} \]
\[ h(z, 1) \rightarrow y \rightarrow y \text{ description} \]
\[ h(z, 2) \rightarrow z \rightarrow z \text{ description} \]

Must store the key
  some where.
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; the likelihood that a page access will be followed by others is high.

**Security:** the fact that processes should not be able to see other processes' space.
Correspondence types:
segment number --> segment descriptor
page number --> page 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.

paddr - pbase = faddr - fbase

The address equation:

paddr - pbase = faddr - fbase

or

faddr = (paddr - pbase) + 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, frames for the page table can only be partly loaded into the processor.
IA32 processors contain a page table cache of part of the page table.
IA64 processors do something quite different and counter-intuitive.
The page table cache: IA32

Page table

Page table cache

Frame pool

In memory

In process

User memory
How the OS intervenes (IA32)

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
Something very strange happened in developing the IA64.

Hardware developers realized that keeping the page table in memory was not that efficient; what they needed was a shortcut mechanism for storing the page mapping that isn't a page table.

This led to a design in which the page table is completely replaced by another mechanism: the **translation lookaside buffer** (TLB).
The following quote comes to mind, erroneously attributed to Einstein:

You see, wire telegraph is a kind of a very, very long cat. You pull his tail in New York and his head is meowing in Los Angeles. Do you understand this? And radio operates exactly the same way: you send signals here, they receive them there. The only difference is that there is no cat.

From <http://www.quotationspage.com/quote/26870.html>
What is a TLB?
Monday, October 30, 2017    1:33 PM

A way of translating between page address and physical address that avoids the page table...
...except that in IA64, **there is no page table.**
Originally, a TLB was a way of short-circuiting a computation that involved an in-CPU page table.

Instead of looking up two parts of the reference, the TLB would store a single record that combines both computations.

Some brilliant hardware engineer decided that the TLB was so much better than the page table, that he would simply make the TLB much larger and eliminate the page table entirely from the hardware.

Same cost, much faster computer!
The TLB is in essence a temporary hash (a cache of a hash :).  
- The TLB lives **inside the CPU**.  
- Entries are temporary.  
- Entries correspond to rows in the page table.  
- When full, entries are flushed according to some **cache eviction strategy** (usually LRU).

Thus, the TLB at any point contains the page records for some subset of all active pages in the page table.  
- When you need a TLB record for a new page, it is **loaded from the OS**.  
- Thus, the structure of the page table **inside the OS** becomes critical.  
- **The OS needs to be able to tell the hardware where a page is in O(1).**
Inside Linux, the IA64 page table for each process is a three-level tree:

From <http://ptgmedia.pearsoncmg.com/images/chap4_0130610143/elementLinks/04fig16.gif>

Translation:
PGD: page table global directory
PMD: page table middle directory
PTE: page table entry
How this works:
The loader utilizes virtual addresses with a particular structure:
  - pgd index: leftmost bits
  - pmd index: middle bits
  - pte index: determines page table entry
  - offset: determines byte
Of these:

**pgd index and pmd index are artificial,**
to speed up indexing and allow caching of the table if needed.

**pte index** determines the actual page table entry

In practice, this is a huge (optimized) array access:

```
    table[pgd index][pmd index][pte index]
```
So, what does a page entry look like:

From http://ptgmedia.pearsoncmg.com/images/chap4_0130610143/elementLinks/04fig23.gif

If page is resident (in real memory), it looks as we described before. If page is not resident, there's no page frame number.
You will find some very common things with uncommon and unusual names:

- **hashing** is a.k.a. **inverted indexing**
- The **translation lookaside** buffer is -- functionally -- **a cache of a hash**.
The TLB was traditionally used in hardware to deal with the following situation:

A segment table in hardware defines segment base and bound, and points to pages.
A page table describes the attributes of each page.

Thus, **two lookups** were necessary to find a logical page:

Find the page in the segment table
Find the page in the page table

To wit:

The translation lookaside buffer replaces these with one hash lookup, to wit:
... decided that even storing the segment and page hashes in the CPU is redundant.

(We have to do that for IA32 anyway)

For IA64, we can just store the TLB, as a combined segment and page hash.

We can make it much larger.

Performance improves, cost goes down!
Virtual memory "complicates" the page table. Only part of the pages are resident. The resident set is typically sparse.

Solution:
Keep the page entry in the page table. Store an offset from which to fetch the page. When the page is needed, load from virtual memory using the offset. Update the page table entry for a resident page.

So, the moral of this story is that:
If a process is totally dormant and swapped out, (e.g., waiting for user input), the page table remains in kernel memory.
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.

**Page table as tiered array**: a more complex idea: allow sparse maps.

**Cached page table (TLB)**: a quite complex idea: allow **big** sparse maps in limited processor real estate.
We need to implement:
   page maps
   segmentation
   virtual memory
What should the arrows be?
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.
Cache large dense spaces.
Now, our story gets 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](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/linear.c)
   [http://www.cs.tufts.edu/comp/111/examples/Hashes/list.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](http://www.cs.tufts.edu/comp/111/examples/Hashes/lookaside.c)
What is an array?

Tuesday, November 02, 2010  11:51 AM
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 cache structure with annotations:]

- Whole store to be cached
- Bound
- Base
- a → c
- b
- Constraint for cache

Hashes Page 45
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
My picture of a hash

Tuesday, November 02, 2010  12:02 PM

This is the hash

Abstract storage

Key

Value

Hash

Hash

Hash
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.
In a modern architecture
  Application address consists of page and offset.
  page tables are fragmented.
  frames are fragmented.
  Inverted indexing (hashing) is used in addressing.
  And the hashes are cached if and when they get large!