Memory management challenges

*Fragmentation*: memory is broken up into segments with gaps between them; must find a "fragment" big enough for each use.

*Internal fragmentation*: processes must allocate more memory than needed.

*External fragmentation*: unused memory lies in small fragments in the global memory map, between processes.
At any time, only part of the heap is "used". There are "fragments" of unused memory. This is called "memory fragmentation".
We can't compact memory by eliminating fragments, because
We don't know what is a pointer in the code!
So if something points to a thing we move, we don't know how to change the reference!
Kinds of fragmentation

Fragmentation kinds:
How fragmentation happens

Wednesday, October 29, 2014 6:10 PM

[Diagram showing process and fragmentation]

Frames allocate in increasing order

Deallocation when process dies

P1, P2, P3
Fragmentation causes:
  ○ Fixed page size.
    ▪ Program size is not a multiple of page size.
    ▪ So there's room "left over"; internal fragmentation.
  ○ "Buddy system"
    ▪ Many algorithms depend upon dividing up resources.
    ▪ Allocation sizes are a power of two of pages. This is internal fragmentation.
  ○ Reclamation:
    ▪ Processes exit, freeing frames.
    ▪ This leaves holes in the physical frame map.
    ▪ This is external fragmentation.
So far, we've covered what happens in the heap. What happens in other parts of the process? What happens to memory in the OS itself?
As we look inside the operating system,

**Very simple and straightforward semantics**
(subject to $O(1)$ time and $O(1)$ space constraints and $\$ constraints)

require exceedingly complex implementations.
Simple semantics

Preserve the illusion that each process is autonomous.
Map (OS) frames to (process) pages.
Only permit certain operations on each page.
Respond to needs for new pages:
  - malloc: the heap.
  - subroutine calls: the stack.
Complex implementation

Allow process pages to exist in memory or on disk. Allow processes to run before pages are all in memory. Deal with processes' needs dynamically. 

Very, very quickly....
Trading space for time

A common OS design strategy:
trade space for time
Store something
In order to make something else faster.
The concept of memory mapping. Recall:

A **frame** is a unit of memory in the OS.
A **page** is a unit of memory in the process.
**Memory mapping** associates (OS) frames with (process) pages.
Basic memory mapping: base and bound registers
  - Base register: physical address of logical address 0.
  - Bounds register: contains size of map

In a modern OS, the bound is a constant. Why?
  - Recovering from fragmentation is a bin packing problem.
  - Unconstrained bin packing is NP complete
  - Constraining to fixed size is polynomial.
Hardware translation

![Diagram of hardware translation process]

- `base`
- `+`
- `ok`
- `not ok`
- `bounds`
- `relative addr`
- `interrupt OS`
Memory mapping is:

Supported by a memory management unit (MMU)
That the OS configures for each process
That functions autonomously from the OS,
until it generates interrupts when it needs OS attention.
So, the OS has to maintain data on what gets mapped where.
What the OS needs to remember:

What frame goes with what process/page?

Special handling instructions for pages: read-only, shared memory, private, copy-on-write...
Each frame has some concept of a "descriptor" (in the OS):
   For which process?
   At what logical address?
   What privileges does the process have?
      Read/write?
      Read-only?
   Has anything been written to it yet?

How do we encode this information optimally?
The problem with frame descriptors...
... is that there are many, many frames ...
... and not so many page states ...
The concept of a segment

A **segment** is a set of pages in a process (a.k.a. frames in the OS) that share the same attributes.

A **segment descriptor** describes the attributes of a segment.

*Anything we can put into a segment descriptor doesn't have to go into the frame descriptor.*

Some segments and their attributes:

- Text segment: read-only.
- Stack segment: read-write, builds down.
- Heap segment: read-write, builds up.
- Data segment: read-write, static size (defined globals).
- BSS segment: read-write, globals with default (0) values.
- Etc.
**Physical address** of a byte (from the point of view of the OS) has (at least) two bitfields:
- Frame address: where the frame starts.
- Byte address: which byte of the frame.

**Logical address** of a byte (in a process) contains (at least) three bitfields:
- Segment address
- Page address (in segment)
- Byte address

A memory map is a **correspondence** (one-one function) between physical and logical addresses.
Virtual memory

At any time, a process's pages are not all in memory
A page is **resident** if it is in memory and mapped.
A page is **virtual** if not.

Virtual pages are kept on disk in a **swap partition**
Organized by logical address.
Left on disk *even if page is resident*.

New language:
Moving a page from virtual to resident is called **swapping in**.
Moving a page from resident to virtual is called **swapping out**.
Handling "faults"

Handling memory "faults"

When a process accesses memory that isn't currently mapped to it, this is called a page fault.

How this is implemented:

The processor/MMU generates an interrupt (of the process).
The OS handles the interrupt (not the process!)
One of several things is done, depending upon the state of the process.
The semantics of virtual memory

The process runs with only part of its pages resident. If a page is needed and isn't resident, block the process, swap in the required page, unblock the process after the page is resident (it returns to the run queue)
To map something that isn't mapped
an interrupt occurs, and:
  block process
  if memory should be mapped
    find some available (physical) memory
    load it with what process expects
    map it to the proper location
  unblock process
else
  segmentation fault: core dumped.
Finding an available frame
   If there is an unused frame,
      read desired contents from disk
      map to requesting process page
   else
      pick a used frame to unmap
      unmap it from whatever process is using it now
      move the used page's contents to disk if needed
      read desired contents from disk.
      map the frame to the requesting process
An important concept in swapping: "dirtiness"
If the memory image of a page matches the disk image, we say that the memory copy of a page is "clean".
If the memory image is different, we say the memory copy of the page is "dirty". Dirty pages must be **flushed to disk** before they can be reused by other processes.
Dirtiness is a **page attribute**.
A classic space-time tradeoff

The dirty bit *consumes space, but saves time.*

1 bit per page saves unneeded flushes, perhaps of very large sets of pages.
The dirty bit doesn't just occur in software
Pages in cache are dirty if changed, and clean if not.
Exact same semantics apply: dirty pages must be flushed before reuse.
Picture of swapping
Another picture of swapping
States of a page
Thursday, October 28, 2010  12:57 PM

[Diagram of state transitions with notes]
- Clean disk
- Can replace without swapping
- Dirty disk
- Swap in
- Swap out
- Write
- Resident & clean
- Resident & dirty

Notes:
- Disk & clean
- Can replace without swapping
We need to choose a page to swap out.

Two prevalent algorithms
- Least recently used (LRU)
- Least frequently accessed (LFA)
How do we choose?

There is no concept of "best" page reuse algorithm.
All choices are based upon "average" process behavior.
All choices are based upon the **principle of locality**: if a process has accessed a page recently, it is **more likely** to access that page than to access another.
The principle of locality in action

Accessing an array: many accesses to one page.
Accessing instructions: always linear.
Access elements of a linked list: buddy system keeps them local to 1-2 pages.
Least recently used

Choose the frame (page) that was least recently accessed.
Swap out its page and reuse it.

*Keep track of time of last access in a frame descriptor(!)*
Least frequently accessed (LFA)

The frame that is least frequently accessed (for a given time) gets unmapped, and is reused.
Keep track of accesses in a frame descriptor(!)
Whenever I say there are two solutions:
  One is optimal for some situations.
  The other is optimal for different situations.
Least-Recently-Used:
  Optimal if memory access is linear, and increasing.
  Cannot account for large loops
Least-Frequently-Accessed:
  Optimal for loops
  Doesn't work well for linear access (all counts are the same).
Several very useful facts:
   If a page is read-only, it **doesn't need to be swapped out**.
   If a page is read-write but hasn't been changed, it **doesn't need to be swapped out**.
Thus, in the **frame descriptor**, there is a **dirty bit**: 1 if something has changed, 0 if not!
   A page is "dirty" if it has been written to, and "clean" if not!
   Thus **only dirty pages need to be swapped out**.
Clean pages/frames can simply be **reused**.

**How this is done:**
   The first write to a clean page generates an interrupt,
   after which it is marked dirty and interrupts are disabled!
Frame descriptor: describes memory
    Frames are memory objects.
Page descriptor: describes process state.
    Pages can be in memory or on disk.
What belongs in each?