```
* MEMORY MANAGEMENT

OS manages phys mem, precious resource.

Need to ensure you don't run out of memory in kernel (very bad).

Linux has multiple allocators:
- traditional "slab" allocator (variants "slob" and "slub")
- page based allocators, with "order"
- kmalloc() -- random, var. length allocations
- vmalloc() -- virtual mem allocation
- "custom" memory allocators

Linux performs periodic cleaning/reclamation of memory, or on demand
(mem. pressure).

1. kmalloc(len, flags): len can be any length, flags determine whether to
wait for memory or not, etc.  call kfree() to return mem back to "pool".
Allocates physical memory, returns CONTIGUOUS buffer.

Problem: fragmentation.  can't find large contiguous chunks.  Defrag?  can
do in Java, but not in C/C++ b/c of pointers, aliases, and ptr arithmetic.

Rule: use kmalloc sparingly, and not for big chunks of mem.

2. Page allocators.  Useful b/c virtual mem, page protection, caching, f/s,
processors, etc. all work on 4KB page units.

Ask to allocate N pages, or "Order" of pages.
- N pages: you get N pages in a list or array of 'struct page *'
- O(rder): you get 2^O pages allocated (1, 2, 4, 8, etc.).
Get back an array of page pointers, but their mem is NOT contiguous.
- much easier to find non-contig. pages.
Order: get 2^O pages.  Based on a "buddy" allocator.

3. vmalloc().  You CAN get virt. mem in kernel, get a big chunk of virtual
memory, contiguous inside the kernel.

Pro: can get a large contig. mem buffer
Con:
- overhead in translating b/t phys. and virt. addr. spaces.
- cannot use vmalloc mem where not allowed to block (spinlocks,
  dev. drv. bottom halves, interrupt handlers, etc.)

4. CUSTOM ALLOCATORS: how to design efficient allocator, with little or no
   fragmentation, and highly efficient alloc/free methods.

Ex. suppose I have a "struct foo" whose sizeof is 117B.  And I need to
alloc/free many of those.

- get a big chunk of contig. mem, e.g., a 4KB page
- break it up into units of 117 bytes (4KB/117 would fit in)
e.g., 34 x 117 units is 3987B, leaves 118B free, or 944 bits
(actually: 4096*8/(117*8+1)
- reserve 118B in beginning for a bitmap freelist (0=free, 1=used)
- the rest, from byte 119-4095 would fit 34 x 117 units.
- API isn't going to be "alloc/free" but alloc_foo() and free_foo().
alloc_foo():
        - check "header" for the first 0 bit, record its location N
        - addr of allocation is "start_of_4k + 118 + N * 117"
        - mark Nth bit as "1" (used)
        - return addr to caller
free_foo(ptr):
        - translate ptr addr into Nth bit
```

```
        - mark bit as 0 (free)

If you know ahead of time how many "foo" you need, can pre-alloc more pages
for this custom "foo" allocator.
- don't over-alloc, or you waste memory.

Q: What happens if you alloc_foo() tries to find a free unit, but they're
all used?

A: can alloc another page (or order pages) for an extension of this original
pool of 34 "foo"s.  But then you have to follow pointers b/t allocated
sub-pools of "foo" structs, and overhead can grow.

In Linux, all "containers" are essentially custom allocators for specific
units.  e.g., struct ext4_inode_info.

* cleaning/reclaiming memory

Lots of caches in kernel, e.g.:
- custom allocators may have "freed but not reclaimed objects"
- page cache has clean/dirty 4KB pages
- dcache/icache with objects whose rc=0

Periodically, kthread(s) wakes up to see what can be cleaned
- BSD/Solaris: updated; Linux: bdflush, pdflush, kflushed, "BDI-based" threads
Policy:
- BSD/Solaris: wakeup every 30s and flush stuff older than 30s (LRU)
        try to keep ~80% used, ~20% free
- Linux: wakeup every 5 sec for meta-data; 30s for data.
        try to keep closer to 100% used/cached, but keep multiple thresholds

Cleaning process:
- invoked from a scheduled kthread(s), or indirectly due to a waiting
  request, such as kmalloc().
- scans all caches in memory, in some order (may be in parallel)
- first, try to discard any objects that can simply be removed from memory:
        e.g., neg dentries, objects w/ rc=0
        e.g., page cache (4KB): data from disks/filesystem, process code
        segments (TEXT), process HEAP/STACK and data segments.  Look for
        "uptodate" (or clean) pages and free them.  Follow LRU.  TEXT
        segments can be discarded (marked readonly).  May also swap out
        HEAP/STACK (and changed DATA) segments.
- next, look for "dirty" pages: ask f/s (e.g., ->writepage) to flush/sync
  page into disk, then can discard page.

Linux has 2 thresholds traditionally: lower watermark and high-watermark
- measured as % of DIRTY data used vs. max page cache size
- e.g., lower threshold may be 30%, high threshold may be 60%, configurable

When flushing kthread wakes up, it checks the usage vs. low and high
threshold
1. if below low threshold, nothing to do. all good.

2. if b/t low and high, ask system (e.g., all f/s) to flush N pages
asynchronously (N often 32).

3. if above high threshold, now ask f/s to flush N (=32 default) pages,
SYNCHRONOUSLY!  This prevents new dirty pages from getting created --
"throttle the heavy writers".

If, even after several rounds of cleaning, we are still above high
threshold, and it's getting worse.  Then linux kernel invokes a special
emergency procedure called the OOMK (Out Of Memory Killer). Finds biggest
process w/ mem footprint, and KILL it!
```