

Real-World Memory Management

1 Unix/Solaris Memory

Memory management on UNIX systems must keep track of several kinds of information. The page mappings for a particular process (the page table), the location of swapped pages on disk, and the state of the actual physical pages in memory.

Page Table Entry:

frame #	age	Copy on Write	Dirty	Ref	Valid	Protect
---------	-----	---------------	-------	-----	-------	---------

The page table entry contains the virtual to physical mapping. The **frame #** contains the physical frame where the virtual page is stored. **age** is processor dependant, and is meant to maintain how long it has been since the page was accessed. **Copy on Write** store the copy on write bit, which is used in UNIX systems to, among other things, render `fork` efficient. **Dirty** is a single bit that indicates whether a page has been modified since last swapped in (the opposite of dirty is clean, and a clean page need not be written out to disk if swapped). **Ref** contains the usage information necessary for a CLOCK-style algorithm. **Valid** is the standard UNIX jargon for resident. A valid page is in main memory, an invalid one is swapped out. **Protect** contains the permission information for the page and is also hardware dependant.

Disk Block Descriptor:

Device #	Block #	Type
----------	---------	------

The disk block descriptor contains the information mapping a virtual page to a spot on disk. The OS maintains a table of descriptors for each process. **Device #** is basically a pointer to the disk that this page was swapped to. **Block #** is the actual block that the page is stored on. This is why most UNIX systems prefer to have a seperate swap partition, so that the block size can be set to the page size. **Type** specifies whether the page is new or pre-existing. This lets the OS know if it has to clear the frame first.

Page Frame Data Table:

Page State	Ref. Count	Logical Device	Block #	pfdata Pointer
------------	------------	----------------	---------	----------------

The page frame data table holds information about each physical frame of memory (indexed by frame number). This table is of primary importance for the replacement algorithm. **Page state** indicates whether or not the frame is available or has an associated page (i.e. whether its been allocated to a process or not). **Ref. Count** holds the number of processes that refer to this page (remember, processes can share the same physical page). **Logical device** contains the device number of the disk that holds a physical copy of the page. **Block #** holds the block number on that disk where the page data is located. **pfdata pointer** is a pointer field that is used to thread a singly-linked list of free frames through the frame table. If the page is free, this points to the next free page (useful for free list-style allocation).

2 Linux Memory Management

Linux memory management is a touch more complex than the classic UNIX example sketched above. Performance is of primary importance to the linux kernel developers, so more sophisticated schemes are employed if they result in higher performance at the expense of increased complexity. **First**, Linux uses a 3-level paging scheme. Why 3 levels? Because Linux runs on both 32 and 64 bit hardware, 3 level tables were chosen (most 64-bit hardware has MMU hardware support for 3-level paging). On 32-bit systems (particularly x86) the size of the intermediate table is set to 1 (which means that it is statically compiled away, and doesn't slow down paging on 2-level hardware systems).

Second, Linux uses a buddy system allocator for page level allocations (no free-list stuff here), and a slab allocator for sub-page allocations. **Third**, Linux divides physical memory into **zones** to account for hardware differences in memory addressing. The three zones are:

- **ZONE_DMA:** The first 16MB of memory. This zone exists because old ISA hardware could not perform DMAs to addresses larger than 24 bits. Therefore legacy hardware can only DMA to locations within the first 16MB of physical memory.
- **ZONE_NORMAL:** Memory at and above 16MB and below 896MB. Again, this has to do with limitations of the linear mapping scheme employed on some (x86) platforms. Normal addresses can be directly addressed by the kernel using a segment descriptor.
- **ZONE_HIGHMEM:** Anything above 896MB. This is a separate zone because it must be addressed differently at the hardware level.

When allocations eventually boil down to the kernel level, they must specify one or more zones to be in. **Fourth** Linux has a more complex staged lifecycle for pages that allows the kernel to defer swapping operations until a more convenient time.

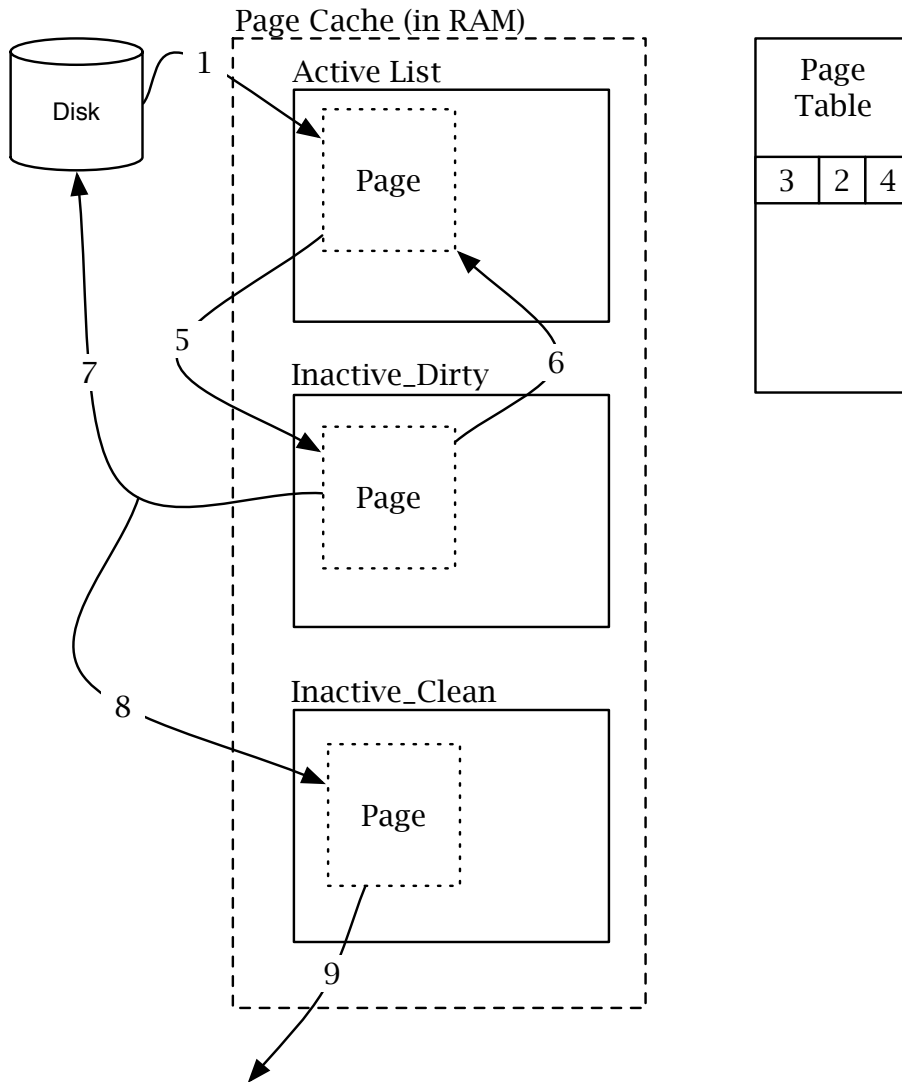
2.1 Linux Aging

Linux, as you may recall, employs an Aging CLOCK algorithm for replacement. This algorithm is an approximation of LFU that has desirable LRU-like properties. Like CLOCK, the active pages

are arranged in a circular list. Each page has an 8 bit (saturating) age field. Each use of the page increments the age by 3 (11 in binary). Each sweep of the arm actually divides the age by 2 (right shift by 1). What this means is that a single use keeps a page alive for two sweeps of the clock hand. Additionally, the age field acts much like the history registers in branch predictors.

2.2 Linux Page Life Cycle

The simplified model we've been studying so far is that resident pages are in main memory, and non-resident pages are on disk. Pages are swapped in when needed, and pages are swapped out when space needs to be made. Linux adopts a staged approach. Pages are stored in one of three separate lists in the main memory "page cache". The first list is the active list, which stores resident, active pages. The second list is the inactive dirty list, which stores inactive pages that have been modified (and marked as non-resident). The third list is the inactive clean list, which stores inactive pages that haven't been modified. The purpose of the inactive lists is three-fold. First, it allows the OS to defer and buffer writing pages out to disk. Second, if memory pressure is low, inactive pages will still be in memory and can be "swapped in" cheaply (by moving them back to the active list). Third, freeing memory in large chunks is easy. The allocator simply frees as many inactive clean pages as needed. This life cycle is illustrated in the following diagram:



The life cycle is explained concisely below. To fully understand this you must first remember that Linux uses an Aging CLOCK for page replacement. Pages are determined to be inactive if they “age out” (i.e. their age field reaches 0). Additionally, each step occurs some time after the previous one precisely so the OS can delay costly operations (perhaps indefinitely).

1. Page created/read-in from disk and added to page cache
2. Page written to, marked as dirty
3. Page ages (not used), decrease age value

4. Age is 0. Mark page as non resident
5. Move page to inactive_dirty
6. Access attempt. Page becomes active again (restored)
7. Otherwise, page eventually written out to disk
8. Written out dirty page marked clean and moved to inactive_clean
9. Frame reclaimed/freed
 - (a) Page request: frame given to another process
 - (b) Or, free memory added to the free list