

# 1 MMU/OS Integration

Historically, there are two ways for an OS and MMU to get along. The first way (and the way currently favored by intel x86 architecture) is to have the MMU basically hardwired. The MMU supports a limited range of page sizes and paging schemes, and all the OS can do is specify which page size it would like and make sure that all the OS memory management behaves in an MMU-compatible way. The advantage of this scheme is that the hardware folks can make the MMU ridiculously fast. The disadvantage is that the OS designers have no control over memory organization, which can lead to inefficiencies and complications (remember that the OS has more knowledge about the processes running on the machine than the MMU does).

The second scheme is a programmable MMU. In this setup, the MMU is controlled by specific supervisor code. The MMU communicates back to the OS by use of interrupts/traps. For example, with a programmed MMU, when a virtual address misses in the TLB, an interrupt is raised and the OS steps in to handle the miss. Similarly, permission bits can be specified to raise interrupts. A programmable MMU has the distinct advantage that the OS can directly control the memory layout for the machine. The OS designer can choose the page-table architecture, and the location and meaning of permission flags. Additionally, OS designers can optimize certain operations (such as context-switching). The disadvantage is that programmable MMUs are slower, software is almost always slower than hardware. Also, the OS will tend to receive more interrupts from a programmable MMU, and this can degrade performance somewhat.

# 2 Memory Replacement

By now we've become familiar with the basic problem. Namely, there are  $N$  virtual pages,  $M$  physical pages, and  $N > M$ . The problem only gets worse if the OS is implementing multi-programming. In that case, there are  $k$  processes, each with their own  $N$  virtual pages. Then, the problem becomes mapping  $k*N$  virtual pages onto  $M$  physical pages. Obviously, if there are more virtual pages than physical ones, the OS can't simply keep everything in main memory. Swapping lets the OS save memory pages to disk to free up physical memory, however disks are about one million times slower than main memory. Therefore, if the OS is going to be reasonably responsive or efficient it will have to do a "good" job of picking pages for swapping out and predicting pages for swapping in.

**Mechanisms:** The CPU has an MMU slaved to it. This MMU is responsible for translating virtual addresses into physical addresses. With paging, the MMU performs the page table lookups to transform a virtual address into a physical address. The MMU has two additional responsibilities:

- **Protection:** The MMU inspects the memory protection bits associated with the virtual page. If the memory request violates those bits (e.g. a write to a write-protected page), the MMU raises an interrupt for the CPU to handle.
- **Residence:** The MMU also has access to resident/non-resident flags. If the page is non-resident (swapped out) the MMU raises an interrupt for the CPU to handle.

The MMU raises interrupts when normal address translation is inadequate (these are usually referred to as page faults). The interrupt is basically a cry for help to the OS. In the case of a protection violation, the OS needs to handle an illegal access by a process. In UNIX, this means signalling the process with a SIGSEGV. By default, this will kill the process (and sometimes dump core). However, processes can register their own handlers, which means that processes on UNIX can exercise fine-grained control of their memory behaviors. When the MMU raises an exception for a non-resident page, that is a call to the OS to swap in a page. In this case, the OS has to read the non-resident page back into memory from disk.

**Policies:** For swapping there are three main questions to be addressed by policy:

- **Fetching:** When should the OS swap in a page?
- **Placement:** When swapping in a page, where should it be placed in main memory?
- **Replacement:** Which page(s) should be swapped out?

**Fetch Rule:** There are two basic policies:

- **Demand Fetching:** Swap in pages on-demand. That is, decide to swap in a page only after a running process has requested it. In this case, pages are only swapped in if they are accessed after being swapped out.
- **Prefetching:** The OS tries to swap in pages before they're needed. This can be done with foreknowledge (i.e. the program informs the OS somehow), or with educated guessing (the OS records some kind of history that it uses to predict future behavior).

Demand fetching makes sense, the OS only swaps in when it receives an interrupt from the MMU telling it that a process is attempting to access a non-resident page. So what's the deal with prefetching? Remember that disks are so much slower than main memory. A process can sit idle through millions of cycles waiting for a page fault. That's potentially several time slices that the process will not be able to use. If the process is unlucky, it could hit one non-resident page after another, executing only a few hundred instructions per second. This means that even if the process is assigned a ridiculously high priority, it won't be able to use the CPU much at all. It's unfair and inefficient. Prefetching is an attempt to avoid this situation. A prefetching OS will attempt to hide the latency of swapping in pages by swapping in the pages ahead of time. The OS could be doing this while executing another process' time slice or while executing the predicted process' time slice. If the prefetching is good, the OS will have swapped in the needed page before it is accessed (and therefore avoid a page fault). Of course, if the prefetching is bad then the OS will litter main memory with unneeded pages and may force needed pages out to disk. Obviously, prefetching is more complicated and dynamic than demand paging.

**Placement Rule:** Placement policies attempt to allocate main memory effectively. There are two components to placement. The first is an allocation algorithm. We've already seen the buddy system, and there are a host of others each specialized for a certain kind of allocation pattern. This course isn't going to cover allocation algorithms in detail (apart from the buddy system). There is a second component to placement, namely where does the OS draw the free pages from:

- **Local frames:** Each process has a set of frames dedicated to its use, and allocation occurs over the local frames for each process.
- **Global frame pool:** All of memory is available for allocation. The allocation algorithm operates over all of main memory for all active processes.

Local allocation may induce a certain amount of fragmentation (if a process underuses its local frames), but it ensures that memory hogs don't shrink the available memory below unusable levels for smaller programs. Global allocation makes the most efficient use of main memory across all processes (no fragmentation), but it means that memory effects (swapping) can have a larger impact on process scheduling and throughput. In reality a certain compromise between these two is reached, by insisting on maintaining a certain minimum resident footprint in memory for active processes.

**Replacement Rule:** Which page gets sacrificed? This policy obviously impacts performance, if the OS swaps out the page that is about to be accessed, then the OS has unfairly retarded the process' progress. If the OS swaps out a page that is never used again, then the OS has freed up memory and still maintained the illusion of an address space without sacrificing speed. Of course, if the OS had a crystal ball (called an oracle in Comp. Sci. circles) it could pick the optimal pages for replacement. In reality, there are no crystal balls, so the OS has to approximate the future somehow. In systems, a useful trick is to use the past to predict the future. The OS will keep track somehow, of the access history of a process. This history (or an approximation of it) is fed into a predictive algorithm that guesses (in an educated fashion, of course) which page is least vital and should be swapped.

## 2.1 Replacement Rule 1: OPT

OPT is the optimal replacement algorithm. To derive OPT, we ask, what is the optimal page to replace? The best page to swap out is that page that will be accessed the furthest in the future. This is unrealizable in practice, because it requires knowledge of the future. However, it can be derived after the fact and used for benchmarking. OPT can also be compared competitively against real-world algorithms.

Page Refs: 1, 2, 1, 3, 1, 2, 4, 2, 3, 5, 1, 4, 3, 2, 1

fault

1	1	1	1	1	1	4	4	4	4	4	4	4	2	2
	2	2	2	2	2	2	2	2	5	1	1	1	1	1
			3	3	3	3	3	3	3	3	3	3	3	3

1 2 1 3 1 2 4 2 3 5 1 4 3 2 1

The example above illustrates OPT in the highly contrived case of a process with 3 pages available and needing 5 pages. OPT generates 7 page faults, 3 are unavoidable (loading the first 3 pages). The other 4 are caused by replacing pages. Because OPT has access to the entire future of the process, it can pick the best possible page to replace (to postpone any future page faults as long as possible).

## 2.2 Replacement Rule 2: Random

With random replacement, the page to replace is selected randomly. It has the advantage of being implementable, and simply cheap. However, the performance is usually dreadful.

Page Refs: 1, 2, 1, 3, 1, 2, 4, 2, 3, 5, 1, 4, 3, 2, 1

fault
-------

1	1	1	1	1	1	1	1	1	5	5	5	5	5	5
	2	2	2	2	2	4	4	3	3	1	1	1	2	1
			3	3	3	3	2	2	2	2	4	3	3	3

1 2 1 3 1 2 4 2 3 5 1 4 3 2 1

In this example, by randomly generating numbers between 0 and 2 to determine which page gets evicted, random replacement generates 12 page faults (3 unavoidable + 9). As you can see, random replacement does worse than OPT. In fact, random replacement would have to generate the exact sequence of replacements to equal opt, and the chances of that are  $\frac{1}{3^4}$ .

## 2.3 Replacement Rule 3: FIFO

Random replacement was a bad idea, so why not use something more structured? FIFO makes a certain amount of sense, as it would evict the older pages before the newer ones. This corresponds roughly to temporal locality, in that after a while pages will get cold and should be evicted. In fact, with competitive analysis, FIFO can be shown to generate no more than 3 times the number of page faults as OPT:

Page Refs: 1, 2, 1, 3, 1, 2, 4, 2, 3, 5, 1, 4, 3, 2, 1

fault

1	1	1	1	1	1	4	4	4	4	4	4	3	3	3
	2	2	2	2	2	2	2	2	5	5	5	5	2	2
			3	3	3	3	3	3	3	1	1	1	1	1
1	2	1	3	1	2	4	2	3	5	1	4	3	2	1

In this example, we see that FIFO generates 8 faults, only 1 more than OPT. The difference is at time 11, when OPT replaces the most recently replaced page. However, as you can see, FIFO tends to match OPT pretty well.

## 2.4 Replacement Rule 4: LFU

LFU stands for least frequently used. It is a refinement over simple FIFO. LFU picks the least active page for eviction. The reasoning is that cold pages will get hit less often than hot pages. Therefore, the least frequently used page would tend to be a cold page.

Page Refs: 1, 2, 1, 3, 1, 2, 4, 2, 3, 5, 1, 4, 3, 2, 1

fault

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2	2	2	2	2	2
			3	3	3	4	4	3	5	5	4	3	3	3
1	2	1	3	1	2	4	2	3	5	1	4	3	2	1

There are some issues. For instance, over what time periods do you calculate this frequency? The system has to have some kind of refresh period in order to avoid favoring pages that were very popular in the past, but not now. The system could incorporate some kind of steady decay, but that's expensive. In the end, LFU adapts slowly to changes in program behavior, and that is undesirable. As you can see from the example above, LFU tends to overwrite newly installed pages, which can have bad effects.

## 2.5 Replacement Rule 5: LRU

LRU stands for least recently used. It is a different take from LFU. Rather than counting accesses to a page, LRU just keeps a list of pages, sorted by time of access (usually greatest to least, because

time “increases”). When a page is hit, it moves to the front of the list and when a page needs to be replaced, the system takes the guy from the end of the list. This exploits locality by assuming that the immediate future closely mirrors the immediate past. LRU is actually better than FIFO, and can be proven to generate no more than twice the number of faults as OPT.

Page Refs: 1, 2, 1, 3, 1, 2, 4, 2, 3, 5, 1, 4, 3, 2, 1

fault
-------

1	1	1	1	1	1	1	1	3	3	3	4	4	4	1
	2	2	2	2	2	2	2	2	2	1	1	1	2	2
			3	3	3	4	4	4	5	5	5	3	3	3

1 2 1 3 1 2 4 2 3 5 1 4 3 2 1

In this example, LRU actually does fairly poorly, scoring 11 page faults. However, with a large memory and a large stream of page references, LRU tends to outperform all the other rules (except OPT). However, LRU is difficult to implement. Actually maintaining a list of thousands of pages and keeping it sorted is a fair amount of overhead. In practice, real memory managers use approximations of LRU to do page replacement.

### 3 Implementing LRU

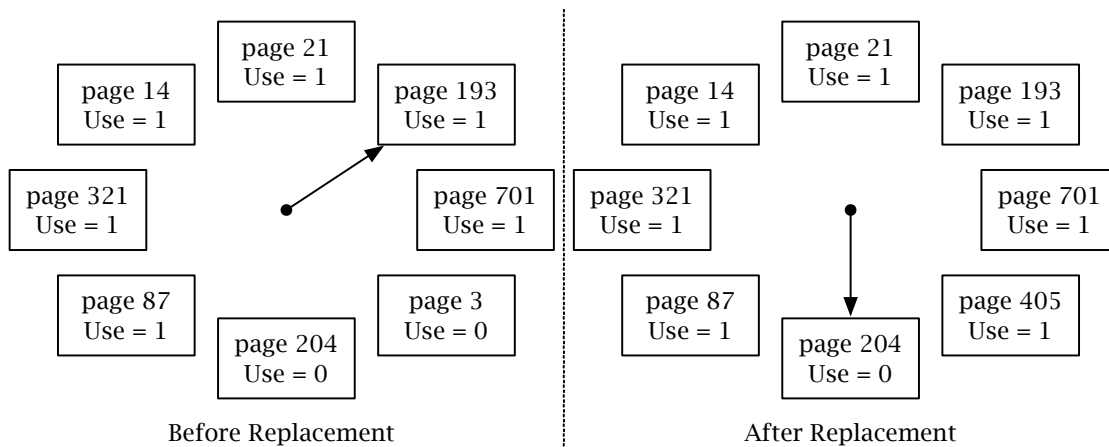
#### 3.1 CLOCK

CLOCK is an approximation of LRU. First, the pages in memory (if using global allocation) are arranged into a circular queue. Each page has a use bit associated with it. When the page is first loaded, its use bit is set to 1. When a page is referenced, its use bit is set to 1. When a page is loaded, a pointer (the clock hand) is set to point at the next page. When memory is full and a page must be evicted, the clock hand sweeps around the circular list looking for a 0 use bit and setting the use bits to 0 along the way. The algorithm is simple:

1. If use == 0, replace page.
2. else use = 1, advance clock hand to next page.

So, if every page has a use bit of 1, then the clock hand will make a sweep of the entire circular list before encountering a 0 use bit. However, in any other case, the clock hand will discover a page that has not been accessed by the program since the last sweep. This is why clock is an approximation to LRU (because there may be several such pages, clock is not guaranteed to get the absolutely least recently used one). The following graphic illustrates clock just before and just

after a page replacement:



In practice, CLOCK offers performance between LRU and FIFO, but with lower overhead than straight LRU.

### 3.2 Two-handed CLOCK

The SVR5 family of UNIX operating systems use a more sophisticated variant of CLOCK, known as two handed clock. As you can imagine, it looks a lot like clock, except there are two hands rather than one. The reference bit is initially set to 0 when a page is brought in, and set to 1 when later referenced by the process. One hand, the fronthand, sweeps through the pages periodically, setting the reference bit to 0. Sometime later, the backhand sweeps through the list and checks the reference bit. If the bit is still set to 0, then the page has not been visited in the interval between the front and back sweeps. These pages are placed in a list to be paged out. There are two parameters that drive this policy:

- **Scanrate:** The rate at which the two hands scan through the list (in pages/sec)
- **Handspread:** The gap (in pages) between the fronthand and the backhand.

Handspread is usually fixed. However scanrate is usually variable. As the amount of free memory shrinks, scanrate increases in order to free up more pages. Conversely as the amount of free memory grows, scanrate decreases to avoid excessive paging.

### 3.3 Aging CLOCK

This is a policy popular on Linux systems. Instead of a single use bit, each page has an 8-bit age associated with it. Each time the page is referenced, the age is incremented (up to 255). Periodically, the hand sweeps through the list and decrements the age. Pages with small ages are “old” and haven’t been used in a while. Ageing clock selects those pages. This clock variant is

actually an approximation of least frequently used, rather than LRU. However, it works fine in practice.

## **4 Enhancements**

### **4.1 Dirty Bits**

Writing out pages is expensive. However, if the contents of a page haven't changed since it was last paged out, it doesn't need to be rewritten out to disk. If the Virtual Memory Manager could detect this, then it could save the OS some expensive work. This is the intuition behind the dirty bit. When a page is swapped in, the dirty bit is set to 0. When the page is written to, the dirty bit is set to 1. The page replacement algorithm should favor pages with a dirty bit set to 0 (to save a write).

### **4.2 Pinning/Locking Pages**

Some pages are so often used that the OS shouldn't every really swap them out. However, the OS can't know this a priori, so applications may have to inform the OS. This is often called pinning or locking a page. An application developer knows that some chunk of data is always hot and should never be swapped out. Therefore after allocating space for the data, the application makes a call to the OS to pin/lock those pages. The OS will then avoid swapping those pages out. Pinning acts like a kind of hint to the OS.

### **4.3 Chance Bits**

Rather than ageing a page, a counter can be associated with it that corresponds to its likelihood of being swapped out. If the use bit is zero, the chance is incremented. When the chance reaches a certain threshold, the OS considers it eligible for swapping. Chance bits remember a bit of history for each page. When the use bit is 1, chance isn't incremented, so pages can go through periods of use and disuse, slowly incrementing the chance counter.