**PAGE REPLACEMENT ALGORITHMS**

* When a page fault occurs, the operating system has to choose a page to be removed from memory to make room for the incoming page.
* If the page to be removed has been modified while in memory, it must be rewritten to the disk to bring the disk copy up to date.
* If the page has not been changed no rewrite is needed, and the page to be read in just overwrites the page being removed.
* Instead of picking a random page to evict at each page fault, system performance would much better if a page that is not heavily used is chosen. If a heavily used page is removed, it will probably have to be brought back in quickly.
* Two similar examples, When memory cache is full, and When Web Server keeps a certain number of heavily used Web pages in its memory cache.
* Page replacement algorithms addresses a certain issue which is “when a page is to be evicted from memory, does it have to be one of the faulting process' own pages, or can it be a page belonging to another process?”
* The Optimal/best Page Replacement Algorithm
  + Easy to describe, impossible to implement.
  + At the moment that a page fault occurs, some set of pages is in memory. One of these pages will be referenced on the very next instruction (the page containing that instruction). Other pages may not be referenced until 10, 100, or perhaps 1000 instructions later.
  + Each page can be **labeled** with the number of instructions that will be executed before that page is first referenced.
  + Optimal page replacement algorithm says that the page with the highest label should be removed.
  + Example, If one page will not be used for 8 million instructions and another page will not be used for 6 million instructions, removing the first reduces possible future page faults.
  + The only problem with this algorithm is that it is unrealizable, Where at the time of the page fault, the operating system has no way of knowing when each of the pages will be referenced next.
  + Overcoming that issue by running a program on a simulator and keeping track of all page references, and apply optimal page replacement on the second run by using the page reference information collected during the first run.
  + The log of page references refers only to the one program and with only one specific input.
  + It is useful as a benchmark against which other algorithms can be measured
* The Not Recently Used Page Replacement Algorithm
  + Operating system collects useful page usage statistics,each page has two status bits. R is set whenever the page is referenced (read or written). M is set when the page is written/modified. These bits must be updated on every memory reference.
  + These two bits can be set by hardware, or instead simulate them as follows,
    - When a process is started up, all of its page table entries are marked as not in memory.
    - When any page is referenced, a page fault will occur. The operating system then sets the R bit, changes the page table entry to point to the correct page, with mode READ ONLY, and restarts the instruction.
    - If the page is subsequently modified, another page fault will occur, allowing the operating system to set the M bit and change the page's mode to READAVRITE.
  + This algorithm can use these bits as follows,
    - When a process is started up, both page bits for all its pages are set to 0 by the operating system.
    - Periodically (e.g., on each clock interrupt), the R bit is cleared, to distinguish pages that have not been referenced recently from those that have been.
    - When a page fault occurs, the operating system inspects all the pages and divides them into 4 categories based on the current values of their R and M bits:

Class 0: not referenced, not modified.

Class 1: not referenced, modified.

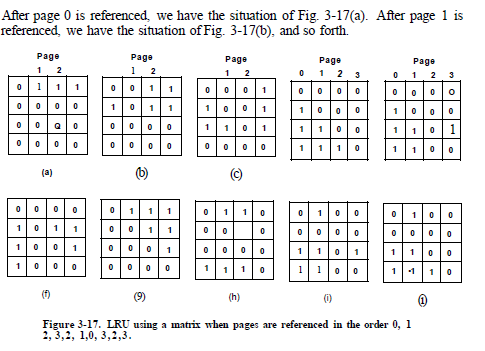
Class 2: referenced, not modified.

Class 3: referenced, modified.

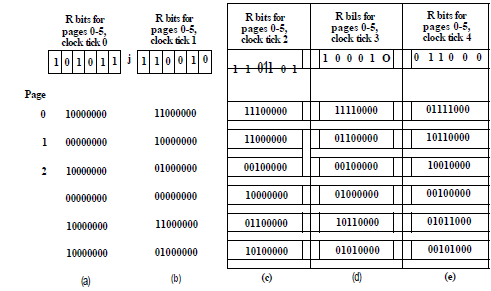
* + - Class 1 pages occur when a class 3 page has its R bit cleared by a clock interrupt. Clock interrupts do not clear the M bit because this information is needed to know whether the page has to be re-written to disk or not.
    - The algorithm removes a page at random from the lowest-numbered nonempty class. Implicitly, it is better to remove a modified page that has not been referenced.
    - Can be implemented but not optimal.
* The First-In, First-Out (FIFO) Page Replacement Algorithm
  + The operating system maintains a list of all pages currently in memory, with the most recent arrival at the tail and the least recent arrival at the head.
  + On a page fault, the page at the head is removed and the new page added to the tail of the list.
  + Is rarely used because it can remove heavily referenced pages.
* The Second-Chance Page Replacement Algorithm
  + A modified version of FIFO that avoids the problem of throwing out a heavily used page.
  + It inspects the R bit of the oldest page.
    - 1 - If it is 0, the page is both old and unused, so it is replaced immediately.
    - 2 - If the R bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory.
    - 3 - Then the search continues.for a page to be removed, check the second one and start again from step one.
  + If all the pages have been referenced, second chance degenerates into pure FIFO.
  + it is unnecessarily inefficient because it is constantly moving pages around on its list.
* The Clock Page Replacement Algorithm
  + A better approach than the second chance algorithm is to keep all the page frames on a circular list in the form of a clock.



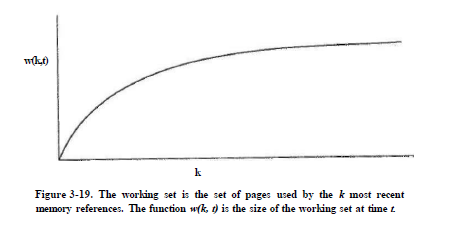
* + When a page fault occurs, the page being pointed to by the hand is inspected. If its R bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position.
  + If R is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with R = 0.
* The Least Recently Used (LRU) Page Replacement Algorithm
  + A realized version of optimal algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for ages will probably remain unused for a long time.
  + LRU (Least Recently Used) Algorithm says that when a page fault occurs, throw out the page that has been unused for the longest time.
  + LRU is theoretically realizable, it is not cheap.
  + To fully implement LRU, a linked list of all pages should be maintained in memory, with the most recently used page at the front and the least recently used page at the rear.
  + The difficulty is that the list must be updated on every memory reference. Finding a page in the list, deleting it, and then moving it to the front is a very time consuming operation.
  + To overcome that difficulty, First **Hardware** Solution is
    - implement LRU with special hardware that equipped with a 64-bit counter, C, that is automatically incremented after each instruction.
    - Each page table entry must also have a field large enough to contain the counter. After each memory reference, the current value of C is stored in the page table entry for the page just referenced.
    - When a page fault occurs, the operating system examines all the counters in the page table to find the lowest one. That page is the least recently used.
  + Second **Hardware** Solution is
    - For a machine with n page frames, the LRU hardware can maintain a matrix of n x n bits, initially all zero.
    - Whenever page frame k is referenced, the hardware first sets all the bits of row k to 1, then sets all the bits of column k to 0.
    - At any instant of time, the row whose binary value is lowest is the least recently used, the row whose value is next lowest is next least recently used, and so forth.



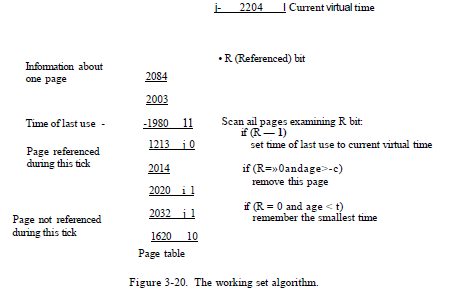
* + LRU is an excellent algorithm, but it cannot be implemented without special hardware. If this hardware is not available, it cannot be used.
* Simulating LRU in **Software**
  + NFU (Not Frequently Used) algorithm
    - It requires a software counter associated with each page, initially zero. The counters roughly keep track of how often each page has been referenced.
    - At each clock interrupt, the operating system scans all the pages in memory. For each page, the R bit, which is 0 or 1, is added to the counter.
    - When a page fault occurs, the page with the lowest counter is chosen for replacement.
    - The main problem with NFU is that it never forgets anything. For example, in a multipass compiler, pages that were heavily used during pass 1 may still have a high count well into later passes. In fact, if pass 1 happens to have the longest execution time of all the passes, the pages containing the code for subsequent passes may always have lower counts than the pass 1 pages. Consequently, the operating system will remove useful pages instead of pages no longer in use.
  + aging algorithm
    - A small modification to NFU makes it able to simulate LRU quite well. The modification has two parts.
      * First, the counters are each shifted right 1 bit before the R bit is added in.
      * Second, the R bit is added to the leftmost rather than the rightmost bit.
    - When a page fault occurs, the page whose counter is the lowest is removed. It is clear that a page that has not been referenced for, say, four clock ticks will have four leading zeros in its counter and thus will have a lower value than a counter that has not been referenced for three clock ticks.
    - This algorithm differs from LRU in two ways
      * Consider pages 3 and 5 in the following figure. Neither has been referenced for two clock ticks; both were referenced in the tick prior to that. According to LRU, if a page must be replaced, we should choose one of these two. The trouble is, we do not know which of them was referenced last in the interval between tick 1 and tick 2. By recording only one bit per time interval, we have lost the ability to distinguish references early in the clock interval from those occurring later. All we can do is remove page 3, because page 5 was also referenced two ticks earlier and page 3 was not.



* + - * In aging the counters have a finite number of bits which limits its past horizon. Suppose that two pages each have a counter value of 0. All we can do is pick one of them at random. In reality, it may well be that one of the pages was last referenced nine ticks ago and the other was last referenced 1000 ticks ago. We have no way of seeing that.
* The Working Set Page Replacement Algorithm
  + Demand paging is a strategy where processes are started up with none of their pages in memory. As soon as the CPU tries to fetch the first instruction, it gets a page fault, causing the operating system to bring in the page containing the first instruction. After a while, the process has most of the pages it needs and runs with relatively few page faults.
    - pages are loaded only on demand, not in advance.
  + During any phase of execution, the process references only a relatively small fraction of its pages.
  + The set of pages that a process is currently using is known as its working set.
  + If the entire working set is in memory, the process will run without causing many faults until it moves into another execution phase.
  + If the available memory is too small to hold the entire working set, that changes over time., the process will cause many page faults and run slowly.
  + A program causing page faults every few instructions is said to be **thrashing**.
  + In a multiprogramming system, processes are frequently moved to disk and all their pages are removed from memory.
    - When the process brought back, it will just cause page faults until its working set has been loaded.
    - The problem is that having many page faults every time a process is loaded is slow.
    - A Solution is that many paging systems try to keep track of each process' working set and make sure that it is in memory before letting the process run to reduce the page fault rate. (working set model).
    - Loading the pages before letting processes run is called prepaging.
  + The fact that most programs randomly access a small number of pages, but that this set changes slowly in time explains the initial rapid rise of the curve and then the slow rise for large k.
    - the contents of the working set is not sensitive to the value of k chosen.



* + It is possible to make a reasonable guess as to which pages will be needed when the program is restarted on the basis of its working set when it was last stopped.
    - Prepaging consists of loading these pages before resuming the process.
  + Based on pages which are in the working set, that operating system is keeping track of, page replacement algorithm can be done when a page fault occurs, find a page not in the working set and evict it.
    - Where we need a precise way of determining which pages are in the working set.
    - The working set is the set of pages used in the k most recent memory references
    - To implement any working set algorithm, some value of k must be chosen in advance.
    - Once some value has been selected, after every memory reference, the set of pages used by the most recent k memory references is uniquely determined.
    - Instead of counting back k memory references, execution time can be used.
      * For example, instead of defining the working set as those pages used during the previous 10 million memory references, we can define it as the set of pages used during the past 100 msec of execution time.
  + Each entry of page table contains (at least) two key items of information: the (approximate) time the page was last used and the R (Referenced) bit.
  + The algorithm works as follows.
    - The hardware is assumed to set the R and M bits, as discussed earlier.
    - Similarly, a periodic clock interrupt is assumed to cause software to run that clears the Referenced bit on every clock tick.
    - On every page fault, the page table is scanned to look for a suitable page to evict.
    - As each entry is processed, the R bit is examined. If it is 1, the current virtual time is written into the Time of last use field in the page table, indicating that the page was in use at the time the fault occurred. Since the page has been referenced during the current clock tick, it is clearly in the working set and is not a candidate for removal.
    - If R is 0, the page has not been referenced during the current clock tick and may be a candidate for removal. To see whether or not it should be removed, its age (the current virtual time minus its Time of last use) is computed and compared to x.
      * If the age is greater than x, the page is no longer in the working set and the new page replaces it. The scan continues updating the remaining entries.
      * if R is 0 but the age is less than or equal to x, the page is still in the working set.
      * If the entire table is scanned without
      * finding a candidate to evict, that means that all pages are in the working set. In that case, the one with the greatest age is evicted.
    - In the worst case, all pages have been referenced during the current clock tick (and thus all have R = 1), so one is chosen at random for removal.
  + working set algorithm is heavy since the entire page table has to be scanned at each page fault until a suitable candidate is located.



* The WSCIock Page Replacement Algorithm
  + It is based on the clock algorithm but also uses the working set information,
  + Due to its simplicity of implementation and good performance, it is widely used in practice.
  + The data structure needed is a circular list of page frames, as in the clock algorithm, Initially, this list is empty.
  + When the first page is loaded, it is added to the list. As more pages are added, they go into the list to form a ring.
  + Each entry contains the Time of last use field from the basic working set algorithm, as well as the R and M bits.
  + As with the clock algorithm, at each page fault the page pointed to by the hand is examined first If the R bit is set to 1, the page has been used during the current tick so it is not an ideal candidate to remove. The R bit is then set to 0, the hand advanced to the next page, and the algorithm repeated for that page.
  + if the page pointed to has R = 0,
    - If the age is greater than x and the page is **clean**, it is not in the working set and a valid copy exists on the disk. The page frame is simply claimed and the new page put there.
    - If the page is **dirty**, it cannot be claimed immediately since no valid copy is present on disk. To avoid a process switch, the write to disk is scheduled, but the hand is advanced and the algorithm continues with the next page.
  + if the hand comes all the way around to its starting point, there are two cases to consider:
    - At least one write has been scheduled
      * the hand just keeps moving, looking for a clean page. The first clean page encountered is evicted.
    - No writes have been scheduled
      * all pages are in the working set, the simplest thing to do is claim any clean page and use it. If no clean pages exist, then the current page is chosen as the victim and written back to disk
  + This algorithm gives good performance but is also efficient to implement
* Summary of Page Replacement Algorithms

