Virtual Machine: Policies, Part 1

Hello, and welcome to this first of two lessons on virtual memory management policies.

In the first lesson we start by reviewing virtual memory mechanics and quickly come to the conclusion that page faults are very expensive. They are in fact so expensive that it comes to a surprise that anybody would even think about using on-demand paging.

We will learn about locality of reference, a characteristic of memory access patterns of typical programs that makes it possible to even consider on-demand paging.

Even for programs with much locality of reference the virtual memory system has to very carefully manage the pages that are in memory at any time. This is handled by page replacement policies.

We will become familiar with a number of such policies, such as FIFO, optimal, LRU, and two variations of the so-called 2nd Chance Replacement Algorithm.

In this lesson we focus on page replacement policies and algorithms. In the second lesson on virtual memory policies we will expand the view and explore policies that manage the number of frames to be allocated to processes, how to cache pages, and others, and we will look at how a number of operating systems implement some of these policies in practice.

At the end of this lesson you will have a good understanding of how page replacement policies work, and how the choice of such a policy affects the performance of the system.

Let's begin.

Recap: Steps of a Page Fault

Let's remind ourselves of the steps that make a page fault.

Here the CPU issues a memory reference to a page that is not in memory.

A page-fault exception occurs and the OS takes over.

The OS locates the page on the paging device.

When the main memory is full, a victim is selected to be paged out.

The victim page is evicted.

The page table entry of the victim is updated to indicate that the victim page is paged out.

The new page is loaded from the paging device into main memory.

The page table entry for the newly loaded page is updated.

And the OS returns and the CPU can re-issue the memory access instruction.

Overall, such a page fault takes a lot of time, first there is the two context switches required for the exception handler. Then there are one or two very expensive I/O operations. Reading and writing to the paging device can take multiple milliseconds and may itself give rise to more context switches to allow for concurrent operation of the I/O and computation on the CPU. We remember this from where we discussed multiprogramming.

Cost of Page Faults

So, we said that page faults are very expensive, to a level that they affect the average memory access time of the system.

We call the average memory access time "effective memory access time" EMA, which can be approximated by this simple formula.

We say that the effective memory access time is

- the probability of a page fault times the "page fault time"
- plus the probability of no page fault times the memory access time, which should be the average access time to main memory, taking caching into consideration.

Let's look at an example.

We assume that the access time to main memory is 50 ns and the time for a page fault in average 10 ms (taking I/O into consideration).

Then the effective memory access time for case of 1 page fault per hundred memory accesses is 0.1 msec.

This is 2000 times longer than accessing main memory.

Even with very small page fault rates, where one page fault occurs per 100'000 memory references, the effective memory access time is 150 nsec, which is still 3 times the cost of accessing main memory.

The objective of virtual memory policies therefore is to minimize the number of page faults, that is, to minimize the page fault rate. The probability of a page fault has to be a very small number in order for the system to not be bogged down swapping pages in and out of memory.

Locality of Reference

Fortunately, programs tend to behave nicely and access memory in a way that allows on-demand paging (and any other forms of caching) to work.

Let's look again at the graph of memory references that we saw earlier.

And let's focus on this period of the execution time. Remember that time flow from top to bottom.

During the marked period of time the CPU accesses the only those memory locations that are inside the colored regions. For demand-paging this means that, once the system has brought into main memory those portions of the address space that are marked in color here, there won't be any page faults during the rest of the period.

A little bit later in time, the memory access pattern changes, but no page fault would occur, since no new memory locations are accessed.

In fact, the system could release the frames containing the pages marked in green.

A little bit later the memory access pattern returns becomes similar to the one during the first highlighted period, on top. If we did not page out the pages in the green area we have no page faults to bring those pages back in.

A little bit later the memory access pattern changes significantly, for the first time. Again we have the accessed portions of memory are marked in color.

We notice that, in order to bring in the red interval of memory addresses, a number of page faults would occur.

Overall we notice, however, that the set of memory locations that are accessed by the process remains quite stable over significant portions of time.

This is due to a common characteristic of programms, called "locality of reference". We can think of a program that exerts locality of reference as accessing the memory nicely: whenever the program references a location, say N at some point, the same program is very likely to reference either the same location, or a location in the vicinity of N in the near future.

Formulated differently, a program does not randomly access memory locations in its address space. Rather, it tends to have rather constant access patterns, which transition to other access patterns with low probability, that is rarely in time.

Page Replacement

Unfortunately, locality of reference alone does not alone guarantee low page-fault rates. It needs to be complemented by a sound page-replacement policy.

The page replacement policy controls how existing pages are evicted and new pages brought in, thus how existing pages are replaced by new pages, which gives rise to the term "page replacement" policy.

The most important aspect of the page replacement policy is the selection of the victim. A poor victim selection can negate all the benefits of locality of reference.

As we will see in the following example.

Page Replacement: Why bother?

Why should we bother with a careful page replacement policy at all?

Let's look at the following example, with this little program iterating 10 times, and each time computing some data and assigning it to a variable. To keep the example program simple we assume that the compiler does no optimization at all.

It also performs a very poor memory layout for the three variables in the program, by placing them all in a different page in the process' memory.

Moreover, the OS decides to allocate only one frame in main memory for this small program. It could also be that the OS allocates a lot of frames to this program, but the victim selector always selects the same victim.

Let's look at the code that our compiler generated for us.

First, we fetch variable I from memory and store it in a register.

We then increment the value and store it back in variable i.

We then fetch variable a from memory and store it in a register.

We then do the same for variable x.

We multiply the two together.

And write the result back to variable a in memory.

We repeat these steps 10 times.

Let's keep track of the I/O operations, that is read/write operations from and to disk.

When we fetch I, we have our first page fault, where we read in the page with variable i.

We increment i.

And we fetch a.

This causes a page fault where we have to I's page first because we updated it since we brought in, and bring in A's page.

Now we fetch x.

This causes another page fault to bring in x. Since we did not change a's page we don't have to page out.

We multiply the two values together. There is no memory access here.

Now we store the result back in a.

This causes another page fault, where we bring in the page for A, and write to it.

We also need to page out this page again in preparation for the loading of I's page at the beginning of the next iteration. As a result, the ill-advised execution of this little program generates 6 I/O operations for each iteration, for a total of 60 I/O operations. If we don't have caching in the I/O device, and an I/O operation takes about 10 msec, then the execution of this trivial part of a program takes more than half a second.

FIFO Page Replacement

Let's look at a number of page replacement algorithms. Let's start with the maybe simple one, the FIFO algorithm, which replaces pages in a FIFO order.

Let's eliminate everything that does not strictly have to do with page replacement. And let's make the number of allocated frames smaller.

At the beginning the page table and the frames are empty.

The FIFO replacement policy picks the one page that has been in memory for the longest period of time. We call it FIFO because the first page to be loaded into memory is the first to be picked as victim for replacement.

For illustrative purposes we keep a FIFO Queue. We will see later that an implementation may not really need it. At the beginning the queue is empty, of course.

Now the CPU starts by referring to Page 0.

The page is loaded into the first available frame, the page table is updated, and a reference to Page 0 is added to the FIFO queue.

Now the CPU continues and at some time later accesses Page 3. The page is loaded into the first free frame, Frame 1 in this case, and the page table is updated.

A reference to Page 3 is added to the FIFO queue.

The same happens when again a new page, this time Page 2 is referenced.

And again for Page 6.

When we refer to Page 5 now, the memory is full, and we need to find a victim to replace.

The first page in the FIFO queue is Page 0, the page is paged out, its page table entry updated, and its reference is removed from the FIFO queue.

Now finally we page in the page into the newly available frame, and update the page table entry.

We also add a reference to the new page at the end of the FIFO queue.

The same procedure is repeated every time we have a page fault.

Let's look at an example.

We will walk the algorithm through 10 references to new pages.

We have 4 frames allocated, which are depicted here.

We will simulate the behavior of the replacement algorithm for a sequence of page references. We don't care about the exact memory locations, just about the pages where they are in.

At the beginning the frames are empty and we refer to 4 pages, which we call A, B, C, D.

There is no need for page replacement until now. We just fill the frames as depicted.

The next reference is to Page C, which is already in memory, and therefore can be accessed without page fault. The pages in memory don't change.

The same happens with the next reference to Page A - then to Page D - then to Page B (10)

The reference to Page E is different because E cannot be found in memory.

- we choose to replace Page A because A has been in memory the longest
- and we load Page E into memory.

The next reference is to Page B, which is in memory already.

While Page A just got paged out, and is not in memory anymore.

We pick the next victim, which in FIFO order is Page B, because it has been in memory the longest.

And load Page A into its frame.

Unfortunately, B is just the page that we reference next.

And the next victim in FIFO order is Page C, which we page out, (20) and we page in Page B in its location.

We are still unfortunate, because the next referenced page is Page C.

We we bring in by replacing Page D, which in turn is the next in FIFO order.

The page faults continue with the reference to Page D, which was just paged out. We pick the next in FIFO order, which is Page E, and page in Page D into its location.

As a result we have 5 page faults, which sums up to a page fault rate of 50%.

Let' analyze the pros and cons of the FIFO algorithm.

On the *pro* side we have its simplicity. You may have noticed that we are simply cycling through the frames in round-robin fashion. There is no need to keep truly a FIFO queue.

The disadvantage of this algorithm is its performance. The heuristic of the algorithm is that pages residing the longest in memory are the least likely to be referenced in the future. This of course is silly, as it does not take into consideration the locality of reference.

Optimal Replacement Algorithm

Let's compare the FIFO's performance to that of the Optimal Replacement algorithm.

There is a page replacement algorithm which is provably optimal. That is, it can be provided that, for any given reference string, the optimal algorithm generates less page faults than any other algorithm one can imagine. The algorithm is very simple, in fact, it always picks that page as victim that will not be referenced for the longest time in the future.

Let's look at how this optimal algorithm does for the same reference string where FIFO generated 5 page faults.

Because this algorithm is clairvoyant, we display the entirety of the reference string, including the future references.

At the beginning we load the page A, B, C, D into memory.

The next few references cause no page faults.

Page E is not in memory, however, and a page fault occurs.

Among all the pages referenced in the future, Page D won't be used for the longest period of time.

So we select Page D for replacement.

D is paged out, and Page E is loaded into the newly free frame.

The next few references don't cause page faults,

Until we hit a reference for Page D.

At this point we have reached the end of our reference string, and we don't know what will be referenced later. We can pick a page randomly to replace.

At the end, we have only two page faults, which is a significant improvement over FIFO, which for the same sequence had 5 page faults.

Clearly the advantage of this algorithm is its performance. Since it provably optimal, there is no algorithm that can have fewer page faults.

Unfortunately, this algorithm cannot be implemented because it requires the OS to know all page reference in the future. This cannot be done. This is not a problem. We do what we always do when something cannot be done.

We approximate.

Approximation to Optimal: LRU

An excellent approximation of the optimal algorithm is LRU, which stands for Least Recently Used.

The LRU algorithm always replaces that one page that has not been accessed for the longest time. The assumption is that past is a good predictor for future behavior. If a page has not been used for a long time, there is a good chance, according to this algorithm, that the page won't be used in the near future as well.

Let's look at how LRU performs for the given reference string. Notice that we don't need future references as the optimal algorithm did.

At the beginning we load pages A, B, C, D into memory, and the next few references don't cause page faults.

As before, Page E cannot be found in memory, and a page fault is initiated.

We look at when the pages in memory have been referenced last, and the least recently used page is page C.

We therefore replace page C, and page in Page E. The next few references don't cause page faults.

When we reference Page C, which just got paged out, we of course have a page fault.

The page that has been referenced least recently is Page D, which gets selected for replacement.

Page D gets replaced by page C.

The last reference is to page D, which of course we just paged out.

We look at the reference times of the pages in memory, and Page E is the page that was referenced least recently.

We replace Page E by the newly paged-in Page D.

The LRU algorithm causes three page faults in this example.

Let's look at some pros and cons.

The advantage of LRU is its performance, which is quite good, both in this example as well as in practice. When locality of reference holds, the past is indeed a good predictor for the future.

The problem with LRU is that it is very difficult to implement. It is not impossible to implement like the optimal algorithm, but maintaining an LRU order of references is not trivial at memory-reference speed.

LRU: Implementation

The problem is that we need to maintain a history of page references, and this history can change with each memory reference.

More precisely, the OS would have to maintain a stack of memory references, with the most recent reference on the top of the stack. Whatever is at the bottom of the stack can be replaced.

Unfortunately, this stack may have to be updated for every memory reference, which is clearly not possible.

System designer have gone through extremes to build support for LRU page replacement policies. One, rather esoteric, solution was to associate – in hardware - each frame with a capacitor, which would be recharged as part of the memory referencing. As the charge of the capacitor would exponentially decay after that, one could select for replacement that frame with the least charge. This solution is difficult to implement.

An alternative software solution is to keep an array of so-called aging registers in memory. Each register has say N bits, and whenever a page is referenced one sets the most significant bit. Periodically, all registers are shifted to the right, basically divided by two. Those frames with the smallest aging register values are the least-recently-used, because their most significant bit in the aging register has been set the longest time ago.

Managing these aging registers as described here is expensive. We will describe next how systems utilize the "use" bit as a one-bit aging register.

Approximation to LRU: 2nd Chance Algorithm

One of these approaches is the so-called 2nd Chance Algorithm, also called the Clock Algorithm. It works similarly to 1-bit aging registers algorithm, except that it does not clear the bits periodically, but rather as part of the search for the next victim page.

Upon each reference, the use_bit is set to one. This is typically done in hardware.

In software we keep a "victim candidate" pointer, which is the number of the next victim frame.

Whenever we need to select a victim, we check if the current victim's use bit is 0. If it is, we have found our victim. Otherwise, clear the use_bit, increment the pointer, and try again. At some point we have traversed all the frames and we start again. This time all bits must be cleared, and we will find a victim for sure.

Let's look how this works for our reference string. The first few references don't cause page faults. We visualize pages with the use-bit set by drawing their frames with a gray background. After the last reference to Page B, all pages in memory have their use_bit set.

Page E is not in memory, however, and so a page fault is initiated. Page A is the current victim candidate, but its use_bit is set. So we cannot pick it as a victim.

We clear the use_bit and move the victim pointer to the next page, Page B. This page cannot be picked as a victim, so we clear the use_bit and move the victim pointer to the next page, Page C.

We do this again, and again, and land at Page A again. This time the use_bit is cleared, and we pick Page A as the victim.

(10) Page A is paged out, and Page E is paged into the newly free frame. Page B is the new victim candidate.

When B is referenced now, we set its use_bit.

Now Page A cannot be found in memory, and a page fault is initiated. We cannot pick the current victim candidate Page B because the use_bit is set.

We clear the bit and continue our search with the next candidate, Page C.

Page C's use_bit is not set, so we pick Page C as the victim.

(15) We page out C and in its space we page in A. A's use_bit is set because we are just now referencing it. Page D becomes the victim candidate.

When we reference Page B, which is in memory, we set its use_bit.

Page C is not found in memory, and we initiate a page fault. The use_bit of the current victim candidate, Page D, is not set, so we pick it for replacement.

We page out D and in its space we page in C. The current victim candidate now is Page E.

(20) The next reference, for Page D triggers a page fault because we just paged out Page D. The use_bit of victim candidate E is set.

So we clear it and go to the next candidate, B.

We clear its use_bit and continue to A, then to C, and then back to E.

The use_bit for E is now not set, and so we pick E as victim.

Page E is paged out and page D takes its space. Page B becomes the current victim candidate. This algorithm generated a total of 4 page faults,

A common improvement to the basic 2nd Chance Algorithm distinguishes clean from dirty pages and so tries give, among the less-recently-used pages, preference for replacement to those pages that are clean, and so don't need to be swapped out.

For this it takes advantage of the dirty bit.

Fundamentally, the algorithm is very similar to the 2nd Chance Algorithm, except that it scans for pages that have both the use_bit and the dirty bit not set.

In the 2nd Chance Algorithm we clear the use_bit whenever we advance the "current victim candidate" pointer. In this algorithm the bit manipulation is a bit more complicated. It is described in this table: whenever the use bit is set, it is cleared. Whenever only the dirty bit is set, it is cleared. In this latter case we need to remember that the page needs to be written to the paging device. We mark such a page with an asterisk. If both bits are clear, we select the page.

As a result, a recently used dirty page is not selecxted until two full scans of the entire set of pages in memory.

We note that there is some confusion with terminology and definition of the algorithms. Some authors describe a slightly different algorithm as the enhanced 2nd Chance Algorithm. Other authors call this algorithm the 2nd Chance Algorithm. In this course we use the term 2nd Chance for the basic 2nd Chance Algorithm, and the term "Enhanced 2nd Chance" for this algorithm.

How does the algorithm work in practice. We again use the familiar reference string.

For reference we have the table up here.

The first few references are routine. Whenever we refer to a page we set its use bit, which we mark by a gray background.

The the reference to A is a write operation, which sets the dirty bit. This is marked using the symbol "d" in the page.

The next reference is routine (5) a Write operation to Page B sets the dirty bit.

Page E cannot be found in memory and so a page fault is initiated. The current victim candidate, Page A, has both the use bit and the dirty bit set.

We clear the use bit as per the table above and Page B becomes the candidate.

We clear the use bit of Page B as well, and Page C becomes candidate.

We clear the use bit of Page C and continue. The same with Page D. Now page A is candidate again, but it has the dirty bit set.

(10) We clear the dirty bit and remember, using the asterisk, that we need to write back the page if it ever gets selected.

We do the same with the next candidate, Page B. The victim now is Page C, which has both bits cleared.

We select C, and page it out. In its space we page in Page E, and Page D becomes current victim candidate.

Now read from B, which sets the use bit in the page.

(15) We write to A, which set its dirty bit.

We read again from B, which is routine.

We try to read from C, which is not in memory. The victim candidate is D, which has both bits cleared and therefore ready to be selected.

We page out D and page in C in its space. A becomes current victim candidate.

(20) Unfortunately, the next reference is to D, which we just paged out.

We leave it to you as homework to fill in the content and marking of the frames after the page fault.

Summary: Virtual Memory: Policies, Part 1

We have come to the conclusion of the first lesson on virtual memory policies.

After a refresher on page fault mechanics and an appreciation of the cost of page faults, we looked at ways to reduce the overall cost of these faults. One important aspect is the program's locality of reference, which leads to rather stable memory access patterns.

We need page replacement algorithms, however, which can take advantage from locality of reference.

We looked at FIFO as a really bad algorithm, and at optimal as a really good one. Unfortunately, the optimal algorithm is aspirational at best, as it relies on Clairvoyance, that is, the capability to foresee the future. It can therefore not be implemented.

A good approximation of Optimal is LRU, which uses the past to make predictions about future behavior. Unfortunately, LRU is very difficult to accurately implement. A number of approximate implementations do work, however.

We explored two such approximations 2nd Chance and Enhanced 2nd Chance.

I truly hope that you enjoyed this lesson on Virtual Memory Policies.

Virtual memory policies will keep us busy for a while, as we need to address questions such as how many pages to keep in memory for each process, whether and how to cache pages, and so on. This is a particularly rich area for study, and poor decisions on virtual memory policies can have major effects on the performance of the overall system.

Thank you for watching.