Virtual Memory: Policies (Part II)

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

In the first lesson we focused on the page replacement policy and on algorithms to perform page replacement.

In this lesson we expand our exploration of operating system policies for virtual memory. First, we will look at the question of how many frames to allocate for each process. This is called "Resident Set Management" because it controls how many pages we keep resident in memory.

We will look at one particular approach to do this, the "Working Set Model".

Aggressively paging out pages that are not in the resident set can lead to high page-fault rates, and one has to think about how to cache and keep in memory pages that are flagged to be paged out.

We will look at a case study, namely how Solaris implements resident-set management and page caching.

In particular this last example will illustrate how operating systems need meta data about pages that goes well beyond what the hardware provides. We will talk about how we can maintain this meta data with only minimal hardware support, by replicating some of the page table entries in software.

Operating system policies for virtual memory are not restricted to page replacement and page caching. We will briefly explore other aspects of virtual memory management and how they affect each other.

At the end of this lesson you will understand how the various components of virtual memory work together to map large numbers of large address spaces onto limited physical memory resources.

Let's begin.

Recap: Page Replacement

We remember from the last lesson how page replacement happens when the physical memory is full and an existing page is selected as victim and evicted and the newly paged-in page takes its space. The old page is replaced by the new page.

We learned that the performance penalty caused by poor victim selection can be significant. This is an old insight, ranging back to the late 60's. System designers realized quickly that the chief problem in memory management is which pages to remove, that is – victim selection. And that clairvoyance would really help: the best choice would be to pick the page with the least likelihood of being reused in the immediate future.

Resident Set Management

Let's talk about a different, but similarly important problem.

How MANY pages should we keep resident in memory for a process? This is equivalent to asking: how much physical memory do we want to allocate for the process?

A good starting point is that at any particular point in time there is a minimum set of of pages that must be in memory for the process to run efficiently, that is, without bogging down with paging.

Let's try to visualize this.

By plotting the page fault rate against the number of allocated frames, that is, the number of pages in memory.

When we can allocate lots of physical memory to the process, the page fault rate is very small, because we have lots of pages fit in memory.

If we reduce the amount of physical memory available to the process, the page fault rate starts increasing

If we reduce the number of frames further, the page fault rate grows very quickly.

To a level, in fact, where the process spends all its time page-faulting, and is not able to make any progress.

We say that the process is thrashing. (10) Thrashing is an insidious effect at system level.

Let's plot the overall system utilization, as we increase the number of processes. More processes means less physical memory allocated for each process.

As long at the number of processes is small, and there is lots of memory for each process, the utilization keeps growing as the number of processes increases. This makes sense, as the system has more work to do at any given time.

When the number of processes grows, the utilization keeps growing as well, but at a slower rate, as the processes start competing for system resources, primarily for memory.

At some point there is no more increase in utilization, and if we increase the number of processes any further, there is a utilization collapse.

This is again due to thrashing, as processes spend all their time paging rather than making progress. The reason the utilization is low is because paging mostly exercises the paging device, such as the hard drive. The CPU is mostly idle.

This may confuse the system administrator, who notices the low utilization and loads more processes, which exacerbates the problem further.

The bad news is that, even if we are able to find determine the set of of pages that are need for the process to efficiently make progress changes over time, this set changes over time, and its size changes as well.

Let's look at our by-now-familiar memory access pattern graph.

If we look at this interval of time, remember that time flow from top to bottom, then the the highlighted pages are accessed and presumably needed.

A little bit later, the size of the set of needed pages has slightly grown.

How many pages should we keep in memory if we want to have sufficiently many pages for the process to make progress?

There is a family of algorithms that attempts to do exactly that. They attempt to optimally allocate the available physical memory to a set of processes such that each process can have have in physical memory a set of pages that allow it to make progress.

We will see in a second that this leads to page replacement where the number of pages allocated to the process varies over time.

The Working Set Model

This family of algorithms takes advantage of the works set of a process.

In order to develop an understanding of what a working set of a process is, let's look at our memory access pattern graph once more.

Let's take a point in time, and call it "t". Remember that time flow from top to bottom.

Now let's go back to a time "delta" in the past, that is, to time "t – delta"

Now let's look at the memory accesses during this time interval.

We see that, during the red time interval we access most of the memory at least once.

This set of accessed memory during the interval from "t – delta" to "t" is called the "working set from t-delta to t". We use the expression W(t,delta) for this.

Working sets have a number of characteristics.

For example, the size of the working set for delta = 1 is 1 for any time t. This is obvious, since over an interval t we access exactly one memory location.

Actually, the working set cannot be smaller than 1 and not bigger than either delta or the total number of pages of the process.

For the next characteristic let's have a closer look at the memory access pattern. (10)

Let's look at a very small interval of time.

The working set of this interval is somewhat small.

If we increase the length of the interval, the size of the working set increases significantly.

If we increase the length of the interval even further, the working set increases still, but not by much.

And if we go further the working set barely increases. The limit is obvious, given that the working set covers most of the pages of the process. (15)

Let's draw a graph where we plot the working set size against the length of the interval, say for the given value of "t" in our example.

The resulting curve illustrates that the working set size is a "concave" function of the interval, which for sufficiently large values of "t" goes asymptotically to the number of pages of the p.

A third characteristic is evident from these figures as well as from the principle of locality of reference we introduced earlier: the working set does not wildly change, and therefore the working set W(t, delta) is a good predictor for the working set in the near future, say t + d.

The Working Set Model Algorithm

These characteristics of the working set are leveraged to define an algorithm, or better, a family of algorithms, to determine the optimal resident set. We will see that a page replacement algorithm follows directly from this.

The algorithm is based on the following memory management strategy, which has two rules:

- First, at each reference, determine the current working set, and keep only those pages in memory that are in the working set.
- Second, a program can only run if it entire current working set fits in memory. If it does not, the process is paged out.

We will see that this is a very elegant strategy, which takes care of multiple aspects of virtual memory management:

- First, it determines for all processes the optimal resident set of pages that ensures the the processes in memory make progress efficiently.
- Second, it automatically drives page replacement by pushing those pages that left the working set out to the paging device.
- Third it limits the number of running processes to those whose working set fits in memory.

The underlying assuption of course is that the size of the working set remains constant over small intervals of time and we therefore can predict the working set, or at least its size, for the near future.

Let's see how this memory management strategy works in practice.

We have the familiar setup with time, available frames, and the reference string. Instead of drawing frames we draw the pages as they populate the working set. We only keep those pages in the working set that are part of W(t, delta). The value of Delta is 4.

We make a few references to populate the working set. First we reference Page E.

Page E gets added to the working set.

Then we reference Page D, which gets added to the working set as well.

When we reference A, which also joins the working set. Now we start with the actual references.

C, which is not in the working set, is referenced. A page fault occurs, C is loaded into memory and is added to the working set.

C is referenced again, but nothing happens since c is in the working set and in memory already. We notice that E dropped out of the working set, as it has not been referenced for the last Delta references. It has been paged out.

Now D is referenced, and nothing happens.

When B is referenced, a page fault occurs and B is paged in and added to the working set. Page A has dropped out of the working set and has been paged out.

C is referenced, and nothing special happens.

Now E is referenced again, and it has to be paged back in, since it was paged out a few memory references earlier. E is added to the working set.

C is referenced again, but it still is in the working set. D has been paged out, since it left the working set.

We reference E again, and B is paged out. Only C and E are left in the working set.

A is referenced and has to be paged back in.

Finally, D is referenced, and has to be paged back in. The working set now has size 4.

In generally we observe how the working set grows and shrinks over time. When a page fault occurs and no page drops out, the working set grows. If there is no memory for the new page, the working set does not fit in memory, and the process is swapped out entirely.

The working set is a wonderful algorithm in theory, but has a number of implementation hurdles in practice:

- First in this form of the algorithm the working set has to be updated at every memory reference. This is very expensive. In practice, there are many ways to approximate this, and only periodically update the working set.
- Second, it is not immediately obvious how to choose a good size for Delta.

If we pick delta too small, we keep pushing out pages that we need to page back in shortly afterwards. The example above is a good illustration of this problem. Alternatively, we could pick delta to be too large, in which case we would keep pages in memory that are not really needed, and so lead to waste of memory. There are some wonderful analytical techniques some of which go back to Peter Denning's original paper in the late 60's, that tie together the optimal value for delta with the performance of I/O operations compared to main memory access times.

Improve Paging Performance: Page Buffering

We noticed in the example above how the working set grows and shrinks over time. In a system with many processes this may lead to situations where the union of the working sets is larger than the available memory, or to a situation where the sum of all working sets is smaller than the available memory, and some of the memory, sometimes a significant amount, is unused. This memory needs to be handled carefully.

One way to do this is "page buffering". When a page drops out of the working set, it is not overwritten directly, but its frame is temporarily parked in one of two frame lists. If the page has not been modified we add the frame to the "free frame list", and if the page is dirty, it is added to the "modified frame list".

When a page fault occurs and we need an empty frame, we get the frame from the free-frame list. We notice that frames in the free frame list are clean, and their content therefore does not need to be paged out. We never pick frames from the modified-frame list.

If now a process references a page that earlier dropped out of the working set, but still is in one of the two lists, it simply reclaims the page and saves itself a page fault.

We noticed above that victims are always picked from the free frame list and not from the modified frame list. This is of course not sustainable. Therefore, we periodically (or when we risk running out of free frames) we write the entire modified list to disk. In this way we page out the contents of the modified frames all in one run. Paging out many pages at once avoids a lot of overhead compared to writing the pages individually.

Working Set with Page Caching

Let's illustrate how this form of page caching works. We use the same reference string as earlier, except that we will distinguish read from write accesses.

In addition, we now maintain a free list for clean frames and a modified list for frames with dirty pages.

Our first reference is a write operation to Page E.

E is paged in and marked dirty.

We then reference D and A, which are both added to the working set. (5)

After warming up our working set, we make our first reference, to Page C, which is paged in.

The next access is to page C again, and we see that Page E dropped out of the working set.

Rather than paging out E, we add it to the modified frame list.

We reference D (10) and write to B, which is paged in. A drops out of the working set. Because the page is not dirty, it is added to the free frame list.

We reference Page C again, and Page E, which is not in the working set and therefore has to be paged in.

Fortunately, it is still in the modified frame list, which means that we can just add it to the working set without any paging.

We now reference Page C again, and Page D drops out of the working set. Because Page D is clean, we add it to the free frame list. (15)

We reference Page E, and Page B drops out of the working set. Because Page B is dirty, we add it to the modified frame list.

We reference Page A, which dropped out of the working set earlier. It is still in the free list, and we can therefore reclaim it from there.

Let's assume now the some other process needs memory, and we use the free frames for that. Resulting in an empty free list. (20)

This may be a good time to page out the dirty pages in the modified list.

The pages are not clean and move to the free list.

The process now reference D, which was in the free list earlier but was replaced by some page of another process when we needed memory. So now we need to page it in. We need memory for it, and we use the frame of page B in the free list for that.

Page B is picked as victim, and Page D is paged in.

Case Study: Page Buffering in Solaris

Let's look at how working sets and page buffering work together, in this case in the Solaris operating system. This approach uses a combination of USE bits to determine which pages are in the working set, and AGING REGISTERS to determine which pages are ready to be paged out.

There is a process in the kernel (called "pageout" in Solaris) which manages pages that are not part of the working set. These pages are all easy to determine, as their USE bit is not set.

The process manages these pages by periodically incrementing these pages' age register. The larger the value of the age register of a page, the longer ago the page has left the working set. It then clear all use bits. If for a given page the use bit does not get set back by the time the process rung again, the page leaves the working set.

Whenever the free memory in the system drops below a low-water mark, a "page-stealer" wakes up and pages out "old" pages and so frees memory. Memory is freed until there is sufficient memory again.

The beauty of this approach is that it pages out pages in a single run, thus saving on IO overhead compared to individual page outs.

Let's walk through how this works. (5)

Initially, a page is paged into memory.

After some time it drops out of the working set, when the pageout process runs, it detects this, and increments the aging register, which now has value one.

In this case this happens twice more.

After that, the page happens to get referenced. Its USE bit is set to zero, (10) and the AGE REGISTER is reset.

Some times later it stops being reference, and the age register keeps increasing until it reaches some maximum value N.

When this value is reached, the page is ready to be paged out.

Some times later the page stealer wakes up, and – if the page has not been referenced still – it is paged (15) out of memory.

Sometimes later it may be referenced and paged back in.

And the whole process repeats itself.

We notice that this mechanism requires page table entries, in this case the AGING REGISTERS, which are not typically provided in hardware.

Demand Paging on Less-Sophisticated Hardware

Let's see how this can be done. We make two observations.

First, in order to provide any form of demand paging, we need at least a VALID bit that gets managed in hardware. If there is no VALID bit, we have no exceptions, and on-demand paging cannot possibly work.

Second, it is generally agreed upon that for demand paging to work efficiently, we need a USE bit and a DIRTY bit. In addition, it would help to have some sort of copy-on-write bit, which we won't further discuss here.

Other systems need more fields in the page table entry, such as the aging register in the Solaris pageout scheme.

Most of these fields can be easily simulated in software, however. This is done in general by duplicating the valid bit in software. Whenever we need to OS to intercept a page reference, we turn off the hardware valid

bit. When the page gets referenced we get an exception and check whether the software valid bit is set. If it is, the page is valid, and we can now simulate all other entries in software.

Let's illustrate how this is done at the example of a software implementation of the USE bit on a CPU that does not support use bits in hardware.

For each page the OS maintains a virtual page table entry in addition to the page table entry managed by the MMU.

This virtual page table entry has a software-valid bit and a software-use bit.

Let's initialize a valid page with zero use bit as follows: we set the hardware valid bit to zero to trigger exceptions. We set the software-valid bit to 1 to indicate that this is a valid page. And we set the software use bit to 0 to indicate that the page has not been referenced.

Whenever the page is referenced, there is a page fault, because the hardware valid bit is 0. This is ok, and the page fault handler checks whether the page is actually valid by checking the software-valid bit.

If the software valid bit is set, we know that the page is indeed valid, and we set the software use bit.

In pictures: Before the reference the hardware valid bit is 0, just to trigger exceptions. The actual valid bit is in software and its value is 1. The page has not been referenced, so the use bit in software is 0.

After a reference to the page, the software use bit is one. In addition, we can set the hardware bit to 1 because we don't need more exceptions to set the use bit. We don't need to set the use bit repeatedly unless we clear it in-between. If we clear the use bit, we have to clear the hardware valid bit as well. The latter in order to allow for exceptions again.

This example illustrates how to implement aging registers in software. The mechanism is very similar to that of the use bit.

Again, we maintain in software a virtual page table entry for each page.

The virtual page table entry has a software-valid bit and an integer field with the aging-register.

Whenever we load a page into memory, the valid bit is set to 0, to trigger exceptions whenever the page is referenced. The aging register is initialized to zero.

Whenever the page is referenced, a page fault occurs because the valid bit is zero. The page fault handler checks the valid bit in software to check whether the page is truly valid.

If it is, the aging-register is reset to 0. This was a reference after all.

We also set the hardware valid bit to 1 because we don't want to have further page fault exceptions for now.

Whenever the OS wants to increment the aging registers, for example periodically, it increments the value of the aging register and sets the hardware valid bit to zero, in order to allow exceptions again. In this way whenever a page is referenced at least once between two increments, an exception occurs, and the aging register is re-set to 0.

We have looked at the implementation of two possible entries in the virtual page table entry, but we could easily think of many others.

OS Policies for Virtual Memory

We have explored a few OS policies to manage virtual memory.

We have started with page replacement policies. Which frame or page to page out if the memory is full? We have seen that this is a critical policy for the overall performance of the system.

We then discussed the resident set management policy. How much memory should we allocate to a process? Of course, this is a zero-sum game: if we allocate more memory to one process, we need to take it away from

another. This is done by replacing an other process' frame. We saw how this is done in the working set algorithm.

Another policy is the Cleaning Policy, which defines when we write a modified page to disk. We explored page caching, which tries to hold on to modified pages until we run out of memory, and even after paging it out it keeps the content of the page in memory to allow possible future references to reclaim it without a page fault.

There are other policies. For example, the Fetch Policy defines how and when we get pages into memory. Clearly, when a page is referenced that this not in memory, we have to page it in immediately. But many systems then take the opportunity to pre-page, where they bring in the requested page, but a few more in anticipation that these pages will be referenced as well. This has the benefit that it amortizes the I/O overhead across multiple pages without adding much CPU overhead.

Another policy defines the placement of pages in physical memory. Which of the available frames does the system use to store a page when it is paged in? In most of the system page placement is irrelevant. There are some systems, for example, AMD Opterons with multiple processors, that are so-called Non-Uniform-Memory-Access machines, where for a given processor it may take longer to access some memory regions compared to others. In such cases, page placement is important.

Finally Load Control decides how many processes can run in the system in order to avoid excessive memory pressure and even thrashing. We explored how the Working Set Algorithm does this by swapping out processes whose working set does not fit in memory.

Summary: Virtual Memory, Policies (Part II)

We have come to the conclusion to this lesson on Virtual Memory Policies.

We learned about the importance of controlling the set of pages that are resident in memory, and therefore the amount of memory allocated to a process. We learned how this is done in the Working Set Model.

We learned how page caching can be used to delay unnecessary paging activity.

We looked at how Solaris manages the working set and how it does page caching. While the mechanisms in the Solaris pagout system are completely different than what we saw in the Working Set Algorithm, the underlying model for working set and page caching is the same.

We noticed that algorithms in practice need more meta-data about pages than what the hardware provides. We explored how to implement support for such meta-data in form of virtual page table entries in software.

We then expanded our investigation of policies to beyond page replacement and caching, and we learned about other policies, such as prefetching and load control among others.

With this we conclude our exploration of virtual memory in operating systems.

I hope that you enjoyed this lesson on Operating System policies for virtual memory, Part II.

You now are in a good position to build your own virtual memory system, once you will have access to a secondary storage device infrastructure. But even if you don't make OS implementation your career you will have benefited from this lesson, as you have gained a good understanding of performance implications of the memory layout of your program and your access patterns. At the very least you will be able to detect when your system is thrashing due to memory pressure!

Thank you for watching!