Virtual Memory: Beyond the Physical Memory

Hello, and Welcome to this first in a series of lessons on Virtual Memory, where we venture beyond physical memory to explore how we can make lots of processes, each with large memory demands, fit in limited physical memory.

The underlying assumption in these lectures is that there is a class of storage devices, which we call secondary storage, which is significantly cheaper than RAM, and we therefore can afford it in much larger quantities.

The disadvantage of this secondary storage is that it is significantly slower than RAM.

We start with a very simple idea. How about we pick a – preferably unimportant – process, and copy it from memory to secondary storage when we need additional memory space? This is called "swapping" processes to secondary storage. Whenever memory becomes available again, we can swap the process back to memory.

We will expand this idea of swapping processes by observing that processes don't always need all their memory to make progress. Rather than swapping entire processes to memory, we only swap their portions of memory that is currently not used. For a lack of better term, we call this "partial swapping" of the process memory.

Now that we are on a roll, we will have a closer look at the dynamics of how processes access memory. We will visualize how the portions of memory that are being used by processes remain somewhat constant over at least short periods of time. This can be leveraged when we "partially swap" the process memory to secondary storage.

Finally, we will briefly look at how this partial swapping could be done at page level.

In the following lessons we will explore how this form of page-level swapping is implemented with the type of paging memory management unit that we discussed in earlier lessons.

Let's begin.

Swapping of Processes

Let's start, once more, with the by now well-known system with a simple operating system and several processes sharing the rest of the memory.

We observe that the memory is quite full.

Now a fourth process, presumably an important one, becomes ready to execute. Unfortunately we don't have enough memory to fit this new process.

Let's assume, however, that we have a large secondary-storage device, in this case a hard drive, that can be used as swapping device. We can now copy the entire Process 2 to the swapping device, including the memory image of the process and the process state, including the CPU state.

In this case we do the same for Process 3.

This gives us enough space to load Process 4 into memory and run it.

If at some time later Process 4 terminates, we can load some of the swapped-out processes back into memory, in this case Process 3.

In this case we have enough memory to swap Process 2 back in as well.

As part of the swap-in operation the memory image of the process is loaded into memory, and the process becomes ready to continue execution.

"Partial Swapping" of Process Memory

In the previous example, we swapped out entire processes at a time. This is easy to implement, but misses out on an excellent opportunity to make efficient use of the physical memory.

An important observation is that processes only very rarely use large portions of their memory at once. We will elaborate on this later, but for now let's assume that our three processes only really make use of highlighted segments of memory. All other portions of memory can be thought of as dormant.

Let's assume that we can identify and isolate these portions of memory and swap them out to secondary storage.

If we can now segment the memory of the new process, we can place the segments into the newly freed portions of memory.

In this way all four processes can make progress in memory. If sufficient memory is available, we can support even more processes when they become ready.

Again, if there is support for this, we can split up the memory for the new process, and distribute it across the available memory.

Memory Access Patterns

Let's have a closer look at the memory access pattern of processes.

For this, let's focus on a single process.

This graph displays the results of an experiment where we mark all memory location as they are referenced by a running process.

The horizontal-axis represents the page number of the memory location.

The vertical axis represents execution time, time is measured in discrete memory references and flow from top to bottom.

If we look closer at this portion of execution, we see that three distinct portions of memory are referenced at all, and two intervals of memory, here left in white, are not touched. Simply put, the process would not know the difference if these portions of memory were not stored in RAM at all. We could store the contents of this memory in secondary storage and bring them in whenever the process gets around to need them.

If we look at longer portions of time, the execution of the process is of course a bit more complicated. Here we see that the memory access pattern goes through distinct mode changes.

During this interval, for example, the process accesses a slightly different set of memory locations.

As a result, the system could swap the memory in the green region to secondary storage, since it is not used.

In this interval later in the process's execution, the memory access pattern changes quite dramatically.

Of particular importance would be this red portion of memory, which would have to be swapped back in from stable storage for the process to continue to make progress.

Let's look more closely at this snapshot at the beginning of the execution of the process.

Page-Level on-Demand Swapping

Let's assume that the memory management uses paging, and that these are the pages for this process.

At this point in time, the process is accessing memory locations in the highlighted pages. These pages better be stored in memory.

These other pages are not being used, and they can as well be swapped to secondary storage, and so freeing the physical memory frames to be used by other pages, either of this process or of another process altogether.

If now the CPU make a reference to a memory location that is present in physical memory, the reference goes ahead.

If, on the other hand, the CPU references a memory location for which the page is swapped out, the CPU should stop, and wait with the memory reference, until the page has located on the secondary storage device and has been successfully swapped in.

The CPU re-issues the memory reference, which now goes ahead.

The opposite occurs when a page is not being used.

This page, for example, is clearly ripe for being swapped out.

Here it gets swapped out to secondary storage, and the newly available frame in physical memory can be reused.

Summary

We have come to the conclusion of this lesson, titled "Virtual Memory, Beyond the Physical Memory", where we have explored at very high level the benefits of using secondary storage to lessen the pressure on physical memory.

We have introduced swapping of processes, where entire processes can be temporarily saved to secondary storage and so make memory available to other processes.

We have learned that it is not necessary to swap entire processes. If one could swap out just those portions of memory that are not used, this could free enough physical memory to increase the number of processes that would fit in memory.

We learned about memory access patterns, and we observed that processes often do have distinct portions of memory that are not accessed over extended periods of time.

For the case of page memory, it would be great if one could distinguish between pages that are used by the process, and pages that are dormant. If so, one could swap the unused pages to secondary storage and only allocate physical frames for pages that are needed by the process. The swapping would the work at page level. The community does not use the term page-level swapping, but rather "on-demand paging", or quite loosely "virtual memory".

I sincerely hope that you enjoyed this introductory lesson on venturing out beyond physical memory. On-demand paging will keep us busy for a while. In the following lessons we will learn how the mechanics of on-demand paging works. We will see that we are familiar with most of the tools for this from previous lessons, we just need to put them to good use.

We will also learn how to design policies to control these mechanisms. How should the system behave under memory pressure? How many frames should it allocate to a process? How does it detect when a page won't be needed in the near future? and so on.

Thank you for watching!