Module 2, Lesson 4 “Advanced Paging”

Page Table Implementations

Hello, and welcome to this lesson on page table implementations.

In the previous lesson we learned about how page tables work conceptually. Specifically, we assumed that the page table is one big array of page table entries, and we did not think about how this table is to be stored. Storing page tables becomes a big problem when the system supports large address spaces. This is particularly the case when we run multiple processes in the system, each of which needs a separate mapping from its address space to the the physical memory.

In this lesson, we look at how page tables are to be implemented for very large address spaces.

We look at two solutions in particular, one is to use multi-level page tables in order to page the page-table, so-to-say. The other is called “inverted page table”, and it uses the frame number as index rather than the page number.

The performance of the inverted page table can be significantly improved by using a hash function, resulting in a variation called “hashed page table”.

Let’s begin.

Implementation of Page Table

We recall from the previous lecture on paging how the logical address is mapped to the physical memory by partitioning the address into page number and offset, and then using the page number to index into the page table to look up the frame number, which is the location of the page in physical memory.

In this lesson, we focus on the page table itself.

Specifically, we will learn that page tables have to be implemented in a way that allows to manage very large address spaces.

We start with the observation that the page table needs one entry per page.

This leads to the question of how and where to store the page table.

An obvious but overly simplistic solution is the store the page table as one big array in memory.

A register in the memory management unit of the CPU, which we call the page table base register, or PTBR, stores the address of the start of the array.

Whenever we switch from one process to the next, we load the location of the new page table into the page table base register.

The main problem with storing the page table as an arrays of entries is that, for large address spaces, a page table becomes very large.

If we take as example a the x86 architecture with a 32-bit address space and 4kB pages and a 4-byte page-table entry for each page, then we end up with 2^20, that is, about 1 million entries.

With 4 bytes per entry this sums up to 4MB per page table. Since each process needs its own page table, this make for lots of memory used to store the page table. We leave it as a homework exercise to calculate how big a page table would have to be for a 64-bit address space. We can safely say that the page table will be humongous.

Hierarchical (Multilevel) Paging

We concluded that page tables can be very large if naively implemented.

Once we think about it, we notice that most of the address space of a process is not used. The process needs memory to store the executable, the statically allocated data, the stack, and the heap. The remaining portions of the logical address space, most often a significant amount, are simply not used, and there is no need at all to map them to physical memory. Those portions of the page table would be wasted.

One solution is therefore to page the page table! Let's see how this works.

First, we partition the page table itself into pages. This is no problem, since the page table is stored in memory, and the memory is paged. Instead of having one large page table, we have a collection of small ones, each one page long. Let's call these "page table pages".

Now we need to keep track of the page table pages. For this we have a one more table, which we call a "page directory". We call the entries in the page directory "page directory entries", and they contain the frame number where the respective page table page is stored. We notice that, since the directory needs one entry per page table page instead of one per page, the directory is actually very small.

The page table base register now points to the page directory. We notice that the page table now has two levels. The first level is the directory, and the second are the page table pages.

A logical address now is resolved as follows. Instead of having a singe page number, the address has a "page table number", which determines which page table page the address belongs to, and a "page number" which defines the number of the page in the given page table page. The entry in the page table page contains the frame number, which is used to construct the physical address, just as in the simple page table.

So, we use the "page table number" to index into the page directory to find where the page table page for the given address is stored.

Once we know which page table page to use, we index into it using the page number to find the frame number. We notice that if a large portion of memory is not used, then we don't need to allocate the page table pages that would be necessary to map this memory. The corresponding entry in the directory would simply be invalid, with no page table page allocated. The amount of memory needed to store the page table is now a function of the how much of the address space is used, rather than how big the address space is. The memory savings are significant, in particular for small processes on machines with large address spaces.

Multilevel, also called hierarchical, paging is very popular.

The 32-bit x86, for example, uses a two-level paging scheme. The logical address is partitioned into a 10-bit page table number, followed by a 10-bit page number, and a 12-bit offset. With a 12-bit offset, pages have a 4kB size.

Page table pages are stored themselves in pages. Since page table entries are 4-byte long, each page table page has 1024 entries.

This works out perfectly, since the 10-bit page number can index up to 2^10, that is 1024, entries as well.

The same goes for the directory, which is stored in a page, and therefore can store 1024 4-byte page directory entries. Again, this would out perfectly, given that the page table number is 10-bit long and so can index up to 1024 entries.

We will go into more details of the x86 memory management in a separate lesson.

We are not limited to 2 levels.

When the address space is very large, the directory becomes very large as well, as it must manage a very large number of page table pages.

One solution is to page the directory. We now have three levels of paging.

The first level is the directory pages directory. The second level are the directory pages. And the third level are the page table pages as before.

The address now is partitioned into 4 sections, with a directory page number (also called first-level directory number), followed by the page table page number (also called the second level directory number) and then followed by the page number and the offset.

The SPARC architecture uses a three-level scheme.

We can go further.

The old Motorola 68030 processor had a four-level paging scheme.

Here we simply dispatch with the effort to name the levels and just number them. The logical address is now partitioned into three directory level portions, the page number, and the offset.

If you think this is crazy, keep in mind that some 64-bit address architectures have 5 or even 6 − level paging.

Something that we need to keep in mind is that each of these directory pages and page table pages are stored somewhere in memory.

If we ignore caching, all these entries must be read from memory whenever we resolve an address. In a four-level paging architecture this means that the memory management unit must issue four memory references (one for each directory level and one to know the frame number) before it can construct the physical address. A single memory reference as issued by the CPU causes 5 memory references issued by the memory management unit.

Caching is therefore very important, and we will discuss in a separate lesson how this is done.

Variations: Inverted Page Table

An alternative way to keep page tables small is the use of so-called "inverted page tables".

Rather than having a separate page table per process we have only one page table in the system. In order to not cause confusion when we have multiple address spaces in the system, we prefix each logical address with a process identifier.

The inverted page table is an array of page entries that is indexed by the frame number, rather than by the page number. The number of entries now is equal to the number of frames in the system. Therefore, the page table grows with the amount of physical memory in the system, rather than with the size of the address space.

When the CPU issues a logical address, the memory management unit adds the ID of the running process and searches the inverted page table for an entry that matches the process ID and the page number of the address.

In this example, the entry for frame number 3 matches, and this frame number is concatenated with the offset to generate the physical address.

The advantages of inverted page tables are two-fold: First, as we pointed our a minute ago, the page table grows with the size of the physical memory, as it has one entry per frame. Also, there is only one table for the entire system for the entire system to manage, rather than one table per process.

The main disadvantage of this system is the need to search through the table to find an entry. As a result, the basic inverted page table has not been very popular. It was mainly used in early virtual memory systems, such as the Atlas computer in the early 60's.

The page table lookup can be significantly improved with the use of judicious hashing. Let's assume we want to improve the performance a naïve page table. Let's further assume that we want to maintain only a single page table per system, rather than a separate page table per process.

Similarly to the case of the inverted page table we prefix the logical address with the process ID.

Now we could use the combination of process ID and page number to index into the page table. The size of the resulting page table would be humongous, and we need to find ways to reduce it. One way to do this is to hash the combination of process ID and page number.

For this we augment the page table with the a column for the process ID and another for the page number.

Rather than indexing the into the page table as before, we hash the combination of process ID and page number, and use the hash value to index into the page table. Since the hash function maps the its input to a fixed number of bits we can choose the size of the page table by selecting an appropriate hash function.

Various forms of hashed page tables are used in many 64-bit architectures, such as the IBM Power architecture, or Hewlett Packard's Precision Architecture, or Hewlett Packard's and later Intel's Itanium architecture.

The advantage of hashing the page table is that it allows the page table to grow with the amount of physical memory in the system. If we choose to use the process ID as part of logical address, we have a single page table, which is an advantage as well.

A possible disadvantage is possibility of collisions. Collisions happen when multiple pages, possibly belonging to different processes, hash to the same entry in the page table. Collisions are easy to detect because, as part of the resolution step we compare the process ID and page number of the address with those of the page table entry. If they don't match we have a collision.

Collisions are handled using traditional collision resolution techniques that we know from software hash tables, such as chaining. Attention must be paid when the resolution must be implemented in hardware of course.

Summary

What did we learn in this lesson on page table implementations?

First we concluded that page tables for large address spaces can become very large. If we need to store one of these page table for each address space, we end up using an inordinate amount of memory just to store the page table to manage that memory.

One common solution to reduce the page table size is to page the page table itself. This is called hierarchical, or multi-level, paging. As the address space increases, several levels are needed to keep the page table size manageable, and address resolution performance begins to suffer.

One approach that had been forgotten for some time, is the inverted page table, where we use the frame number as index rather than the page number. The memory management unit now needs to search through the inverted table to find a matching page number. The performance of this is obviously poor.

One way to speed this up is to add hashing. The resulting hashed page table is similar to the inverted page table, except that a hashing step replaces the search through the table. Of course, one now has to deal with collisions, but the benefits are significant, and this type of page table is quite popular in modern 64-bit architectures.

In this lecture we have focused on how to make the page table fit in memory. In the following lectures we will investigate other aspects of paging implementations, for example, how to speed up the memory resolution process.

Thank you for watching!