Consequences of virtual addresses, pages and page tables

Virtual addressing, pages and page-tables are the basis of every modern operating system. It under-pins most of the things we use our systems for.

Individual address spaces

By giving each process its own page table, every process can pretend that it has access to the entire address space available from the processor. It doesn't matter that two processes might use the same address, since different page-tables for each process will map it to a different frame of physical memory. Every modern operating system provides each process with its own address space like this.

Over time, physical memory becomes *fragmented*, meaning that there are "holes" of free space in the physical memory. Having to work around these holes would be at best annoying and would become a serious limit to programmers. For example, if you `malloc` 8 KiB of memory; requiring the backing of two 4 KiB frames, it would be a huge unconvinced if those frames had to be contiguous (i.e., physically next to each other). Using virtual-addresses it does not matter; as far as the process is concerned it has 8 KiB of contiguous memory, even if those pages are backed by frames very far apart. By assigning a virtual address space to each process the programmer can leave working around fragmentation up to the operating system.

Protection

We previously mentioned that the virtual mode of the 386 processor is called protected mode, and this name arises from the protection that virtual memory can offer to processes running on it.

In a system without virtual memory, every process has complete access to all of system memory. This means that there is nothing stopping one process from overwriting another processes memory, causing it to crash (or perhaps worse, return incorrect values, especially if that program is managing your bank account!)

This level of protection is provided because the operating system is now the layer of abstraction between the process and memory access. If a process gives a virtual address that is not covered by its page-table, then the operating system knows that that process is doing something wrong and can inform the process it has stepped out of its bounds.

Since each page has extra attributes, a page can be set read only, write only or have any number of other interesting properties. When the process tries to access the page, the operating system can check if it has sufficient permissions and stop it if it does not (writing to a read only page, for example).

Systems that use virtual memory are inherently more stable because, assuming the perfect operating system, a process can only crash itself and not the entire system (of course, humans write operating systems and we inevitably overlook bugs that can still cause entire systems to crash).

Swap

We can also now see how the swap memory is implemented. If instead of pointing to an area of system memory the page pointer can be changed to point to a location on a disk.

When this page is referenced, the operating system needs to move it from the disk back into system memory (remember, program code can only execute from system memory). If system memory is full, then *another* page needs to be kicked out of system memory and put into the swap disk before the required page can be put in memory. If another process wants that page that was just kicked out back again, the process repeats.

This can be a major issue for swap memory. Loading from the hard disk is very slow (compared to operations done in memory) and most people will be familiar with sitting in front of the computer whilst the hard disk churns and churns whilst the system remains unresponsive.

mmap

A different but related process is the memory map, or `mmap` (from the system call name). If instead of the page table pointing to physical memory or swap the page table points to a file, on disk, we say the file is `mmap`ed.

Normally, you need to `open` a file on disk to obtain a file descriptor, and then `read` and `write` it in a sequential form. When a file is mmaped it can be accessed just like system RAM.

Sharing memory

Usually, each process gets its own page table, so any address it uses is mapped to a unique frame in physical memory. But what if the operating system points two page table-entries to the same frame? This means that this frame will be shared; and any changes that one process makes will be visible to the other.

You can see now how threads are implemented. In the section called “ `clone` ” we said that the Linux `clone()` function could share as much or as little of a new process with the old process as it required. If a process calls `clone()` to create a new process, but requests that the two processes share the same page table, then you effectively have a *thread* as both processes see the same underlying physical memory.

You can also see now how copy on write is done. If you set the permissions of a page to be read-only, when a process tries to write to the page the operating system will be notified. If it knows that this page is a copy-on-write page, then it needs to make a new copy of the page in system memory and point the page in the page table to this new page. This can then have its attributes updated to have write permissions and the process has its own unique copy of the page.

Disk Cache

In a modern system, it is often the case that rather than having too little memory and having to swap memory out, there is more memory available than the system is currently using.

The memory hierarchy tells us that disk access is much slower than memory access, so it makes sense to move as much data from disk into system memory if possible.

Linux, and many other systems, will copy data from files on disk into memory when they are used. Even if a program only initially requests a small part of the file, it is highly likely that as it continues processing it will want to access the rest of file. When the operating system has to read or write to a file, it first checks if the file is in it's memory cache.

These pages should be the first to be removed as memory pressure in the system increases.

Page Cache

A term you might hear when discussing the kernel is the *page cache*.

The *page cache* refers to a list of pages the kernel keeps that refer to files on disk. From above, swap page, mmaped pages and disk cache pages all fall into this category. The kernel keeps this list because it needs to be able to look them up quickly in response to read and write requests XXX: this bit doesn't file?

Prev Up Next
Virtual Addresses Home Hardware Support