Memory Management Part 1
The Address-Space Concept

- Protect processes from one another
- Protect the OS from user processes
- Provide efficient management of available storage

The concept of the address space is fundamental in most of today’s operating systems. Threads of control executing in different address spaces are protected from one another, since none of them can reference the memory of any of the others. In most systems (such as Unix), the operating system resides in address space that is shared with all processes, but protection is employed so that user threads cannot access the operating system. What is crucial in the implementation of the address-space concept is the efficient management of the underlying primary and secondary storage.
Early approaches to managing the address space were concerned primarily with protecting the operating system from the user. One technique was the hardware-supported concept of the *memory fence*: an address was established below which no user mode access was allowed. The operating system was placed below this point in memory and was thus protected from the user.
The memory-fence approach protected the operating system, but did not protect user processes from one another. (This wasn’t an issue for many systems—there was only one user process at a time.) Another technique, still employed in some of today’s systems, is the use of base and bounds registers to restrict a process’s memory references to a certain range. Each address generated by a user process was first compared with the value in the bounds register to make certain that it did not reference a location beyond the process’s range of memory, and then was modified by adding to it the value in the base register, insuring that it did not reference a location before the process’s range of memory.

A further advantage of this technique was to ensure that a process would be loaded into what appeared to be location 0—thus no relocation was required at load time.
Swapping is a technique, still in use today, in which the images of entire processes are transferred back and forth between primary and secondary storage. An early use of it was for (slow) time-sharing systems: when a user paused to think, his or her process was swapped out and that of another user was swapped in. This allowed multiple users to share a system that employed only the memory fence for protection.

Base and bounds registers made it feasible to have a number of processes in primary memory at once. However, if one of these processes was inactive, swapping allowed the system to swap this process out and swap another process in. Note that the use of the base register is very important here: without base registers, after a process is swapped out, it would have to be swapped into the same location in which it resided previously.
The concept of overlays is similar to the concept of swapping, except that it applies to pieces of images rather than whole images and the user is in charge. Say we have 100 kilobytes of available memory and a 200-kilobyte program. Clearly, not all the program can be in memory at once. The user might decide that one portion of the program should always be resident, while other portions of the program need be resident only for brief periods. The program might start with routines A and B loaded into memory. A calls B; B returns. Now A wants to call C, so it first reads C into the memory previously occupied by B (it overlays B), and then calls C. C might then want to call D and E, though there is only room for one at a time. So C first calls D, D returns, then C overlays D with E and then calls E.

The advantage of this technique is that the programmer has complete control of the use of memory and can make the necessary optimization decisions. The disadvantage is that the programmer must make the necessary decisions to make full use of memory (the operating system doesn’t help out). Few programmers can make such decisions wisely, and fewer still want to try.
One way to look at virtual memory is as an automatic overlay technique: processes “see” an address space that is larger than the amount of real memory available to them; the operating system is responsible for the overlaying.

Put more abstractly (and accurately), virtual memory is the support of an address space that is independent of the size of primary storage. Some sort of mapping technique must be employed to map virtual addresses to primary and secondary stores. In the typical scenario, the computer hardware maps some virtual addresses to primary storage. If a reference is made to an unmapped address, then a fault occurs (a page fault) and the operating system is called upon to deal with it. The operating system might then find the desired virtual locations on secondary storage and transfer them to primary storage. Or the operating system might decide that the reference is illegal and deliver an addressing exception to the process.

As with base and bounds registers, the virtual memory concept allows us to handle multiple processes simultaneously, with the processes protected from one another.
There are two basic approaches to structuring virtual memory: it is divided either into fixed-size *pages* or into variable-size *segments*.  

With the former approach, the management of available storage is simplified, since memory is always allocated one page at a time. However, there is some waste due to internal fragmentation—the typical program does not require an integral number of pages, but, on the average, requires memory whose size is $1/2$ page less than an integral multiple of the page size.

With the latter approach, memory allocation is more difficult, since allocations are for varying amounts of memory. This may lead to external fragmentation, in which memory is wasted (as we saw in discussing dynamic storage allocation) because there are a number of free areas of memory too small to be of any use. The advantage of segmentation is that it is a useful organizational tool—programs are composed of segments, each of which can be dealt with (e.g. fetched by the operating system) independently of the others.

Segment-based schemes were popular in the ’60s and ’70s but are less so today, primarily because the advantages of segmentation do not outweigh the extra costs due to complexity of the hardware and software used to manage it. We will restrict our discussion to page-based schemes.

There is also a compromise approach, *paged segmentation* (as opposed to *segmented paging*, which we discuss later), in which each segment is divided up into pages. This approach serves to make segmentation a more viable alternative, but doesn’t serve well enough.
There are a number of hardware-based facilities for the support of paging. The most common of them is the use of page tables, which are tables implementing complete maps from virtual memory to real memory. In the following pages we examine a number of variations on page-table design. Another approach, either used alone or in conjunction with the other two, is the use of caches called translation lookaside buffers (TLBs).
A page table is an array of page table entries. Suppose we have a 32-bit virtual address and a page size of 4096 bytes. The 32-bit address might be split into two parts: a 20-bit page number and a 12-bit offset within the page. When a thread generates an address, the hardware uses the page-number portion as an index into the page-table array to select a page-table entry, as shown in the picture. If the page is in primary storage (i.e. the translation is valid), then the validity bit in the page-table entry is set, and the page-frame-number portion of the page-table entry is the high-order bits of the location in primary memory where the page resides. (Primary memory is thought of as being subdivided into pieces called page frames, each exactly big enough to hold a page; the address of each of these page frames is at a “page boundary,” so that its low-order bits are zeros.) The hardware then appends the offset from the original virtual address to the page-frame number to form the final, real address.

If the validity bit of the selected page-table entry is zero, then a page fault occurs and the operating system takes over. Other bits in a typical page-table entry include a reference bit, which is set by the hardware whenever the page is referenced, and a modified bit, which is set whenever the page is modified. We will see how these bits are used later in this lecture. The page-protection bits indicate who is allowed access to the page and what sort of access is allowed. For example, the page can be restricted for use only by the operating system, or a page containing executable code can be write-protected, meaning that read accesses are allowed but not write accesses.
Page-Table Size

- Consider a full $2^{32}$-byte address space
  - assume 4096-byte ($2^{12}$-byte) pages
  - 4 bytes per page table entry
  - the page table would consist of $2^{32}/2^{12} (= 2^{20})$ entries
  - its size would be $2^{22}$ bytes (or 4 megabytes)
Forward-Mapped Page Table

<table>
<thead>
<tr>
<th>L1 Page #</th>
<th>L2 Page #</th>
<th>Offset</th>
</tr>
</thead>
</table>

L1 Page table

L2 Page tables

Page frame
The IA32 architecture employs a two-level page table providing a means for reducing the memory requirements of the address map. The high-order 10 bits of the 32-bit virtual address are an index into what’s called the page directory table. Each of its entries refer to a page table, whose entries are indexed by the next 10 bits of the virtual address. Its entries refer to individual pages; the offset within the page is indexed by the low-order 12 bits of the virtual address. The current page directory is pointed to by a special register known as CR3 (control register 3), whose contents may be modified only in privileged mode. The page directory must reside in real memory when the address space is in use, but it is relatively small (1024 4-byte entries: it’s exactly one page in length). Though there are potentially a large number of page tables, only those needed to satisfy current references must be in memory at once.
Quiz 1

Suppose a process on an IA32 has exactly one page residing in real memory. What is the total number of combined pages of page-directory table and page tables required to map this page?

a) 1
b) 2
c) 4
d) 8
Recall that, in current implementations of the x86-64 architecture, only 48 bits of virtual address are used. Furthermore, the high-order 16 bits must be equal to bit 47. Thus the legal addresses are those at the top and at the bottom of the address space. The top addresses are used for the OS kernel, and thus mapped into all processes. The bottom address are used for each user process. The addresses in the middle (most of the address space — the slide is not drawn to scale!) are illegal and generate faults if used.

The reason for doing things this way (i.e., for the restrictions on the high-order bits) is to force the kernel to be at the top of the address space, allowing growth of the user portion as more virtual-address bits are supported.
For the x86-64, four levels of translation are done (the high-order 16 bits of the address are not currently used: the hardware requires that these 16 bits must all be equal to bit 47), thus it really supports “only” a 48-bit address space. Note that only the “page map table” must reside in real memory at all times. The other tables must be resident only when necessary.
Alternatively, there may be only three levels of page tables, ending with the page-directory table an 2MB pages. Both 2MB and 4KB pages may coexist in the same address space; which is being used is indicated in the associated page-directory-table entry.
The hardware also supports 1 GB pages by eliminating the page-directory table. Not many operating systems (if any) yet take advantage of this.
Why Multiple Page Sizes?

- External fragmentation
  - for region composed of 4KB pages, average external fragmentation is 2KB
  - for region composed of 1GB pages, average external fragmentation is 512MB
- Page-table overhead
  - larger page sizes have fewer page tables
    - less overhead in representing mappings
    - both in memory and in cache
A linear page table is, ostensibly, a single-level page table mapping virtual memory to real memory. However, the page table itself may reside in virtual memory thus may, in turn, be mapped by another linear page table (presumably one that resides directly in real memory). In the system illustrated in the slide, there are four separate "spaces". The page table mapping space 10 (whose high-order address bits are 10) resides in real memory. But the page tables mapping the other spaces reside in space 10.
In the VAX architecture there are four spaces, 00, 01, 10, and 11. The user portions of the address space are in spaces 00 and 01, the kernel is in 10. The 00 base register indicates where in 10 space the 00 page table starts. The 10 base register indicates where in real memory the 10 page table starts. Thus mapping a 00 virtual address to real memory involves first determining where the entry of the 00 page table resides in 10 space, then determining where the entry of the 10 page table that maps it resides in real memory.
$ 

- VAX architecture introduced in 1978
  - memory cost $40,000/MB
    - 3.8¢/byte
      (.475¢/bit)
Linear Page-Table Management

• 00 and 01 page tables each require contiguous locations in 10 space
  – with 512-byte pages, 8MB each:
    - maximum of 128 such page tables
    - (need room for other things, e.g. OS)

• Reduce size requirements with partial page tables
  – length registers constrain size of each space
The address-space requirements of traditional Unix work well with linear page tables with length constraints.
Quiz 2

Suppose the page size is 512 bytes \( (2^9) \) and each page-table entry requires 4 bytes. How many pages of page-table entries are required to map 1 megabyte \( (2^{20}) \) of address space?

a) 4  
b) 8  
c) 16  
d) 32
To represent 1 MB of 00 space and 1 MB of 10 space, a total of 32 entries of 10 page table are required, occupying 128 bytes. If we have one page each for the 00 and 01 page tables, that’s 1024 bytes, for a total of 1152 bytes.
Modern Unix systems make extensive use of mapped files, requiring a number of regions in the address space. Thus two length registers wouldn’t be all that effective in reducing the space required of page tables.
$ 

- Requires sufficient 10 page-table entries to map almost all of 00 and 01 space
  - $2^{14}$ 10 page-table entries for each space
    - requiring 64KB each, 128KB total
    - $5000 in 1978

- <1¢/process today
  - who cares?
  - increase address space from $2^{32}$ to $2^{64}$
    - 4,294,967,296-fold increase
    - significant ...

---

Operating Systems in Depth

XX–28

Copyright © 2019 Thomas W. Duquenne. All rights reserved.
In a *hashed page table*, the page number is a key used as the entry into a hash table. Collisions are handled by chaining. In the form shown in the slide, each page requires three words to represent it.

Hashed page tables support widely but sparsely allocated address spaces well. However, they may require multiple memory accesses for some translations. Furthermore, a fair amount of extra space is required for chaining the collisions.
A variation of hashed paging that shows promise for supporting 64-bit architectures well is *clustered paging*. In this scheme, a number of pages (perhaps sixteen) are handled by each entry in the lists of hash synonyms. Thus there are three words of overhead per sixteen pages, rather than per page.

The paper “A New Page Table for 64-bit Address Spaces,” by M. Talluri, M. Hill, and Y. Khalidi, *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995, describes and analyzes this scheme. The contents of the proceedings of this conference can be accessed via the hypertext link in the title of the slide. If you subscribe to ACM's digital library, you can obtain a copy of this paper via the contents page.
Inverted Page Tables

Here, inverted page tables, a variant of hashed page tables, are used to avoid wasting a large amount of memory for the translation map. A page frame table is maintained that indicates for each page frame of real memory what virtual address is mapped into it. In a typical implementation (we describe here a simplification of the IBM RS/6000 scheme), the hardware takes the page number from the virtual address, hashes this into a hash table, and then follows a linked list of hash synonyms in the page-frame table until it finds the desired entry. Then the index of this entry (in the page-frame table) is the page-frame number of the page. If the entry is not found, then a page fault is generated.

This procedure would be quite slow if it were always performed exactly as described. However, it can be combined with the use of a TLB to achieve a system that, on the average, performs well.

Another difficulty with inverted page tables is that there are usually portions of several address spaces in primary storage. Thus the virtual address of a page does not identify it uniquely, since different address spaces have pages with identical virtual addresses. So there must be some sort of address space ID to indicate which page is whose. There is a hardware register that contains the address space ID of the current address space, and each entry in the page-frame table contains an address space ID.
Translation Lookaside Buffers

<table>
<thead>
<tr>
<th>Tag</th>
<th>Key</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>6</td>
<td>12</td>
</tr>
</tbody>
</table>

- Tag  
- Page Frame #
- Tag  
- Page Frame #
- Tag  
- Page Frame #
For details, see text page 297.
TLB Shootdown Algorithm

// shooter code
for all processors i sharing address space
    interrupt(i);
for all processors i sharing address space
    while (noted[i] == 0)
        
    modify_page_table();
update_or_flush_tlb();
done[me] = 1;

// shootee i interrupt handler
receive_interrupt_from_processor j
noted[i] = 1
while (done[j] == 0)
    
flush_tlb()
The Intel documentation on the Intel IA-64 architecture can be found at https://www.intel.com/content/www/us/enprocessors/itanium/itanium-architecture-vol-1-2-3-4-reference-set-manual.html. As the slide shows, a 64-bit virtual address is split into a 3-bit region number and a 61-bit region address. However, regions are assigned 24-bit IDs and thus there can be $2^{24}$ of them. The region-number field of the virtual address selects one of eight region registers, each of which contains a 24-bit ID of a region. Thus, the size of the address space can actually be $2^{85}$ bytes. The page size varies (though is fixed per region) from 4 kilobytes to 256 megabytes.
IA-64 Address Translation

- TLB
  - software-managed
- Virtual Hash Page Table (VHPT)
  - per-region linear page table
  - single large hashed page table

The IA-64 architecture provides a software-managed TLB to handle translation. On TLB misses, an OS can arrange for the hardware to use a virtual hash page table (VHPT) to lookup the translation and reload the cache. There are two options for the VHPT: it can be a per-region linear page table or a single large hashed page table. If the former, then each region has its own linear page table within the region address space. On TLB misses, the hardware looks up the address within the appropriate linear page table. Since the page table itself is in virtual memory, the addresses of its entries are translated by the TLB, which could result in another miss (which, in this case, would be handled directly by the OS).

If a single large hashed page table is used, then an implementation-defined hash function is used to map a virtual address into the (single) hash table, which is itself in virtual memory (and, again, the addresses of its entries are in virtual memory, thus possibly resulting in another TLB miss).
Translation: TLB

region reg → TLB → page number, offset
Translation: TLB Miss

<table>
<thead>
<tr>
<th>region reg</th>
<th>Hash Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>page number</td>
<td>offset</td>
</tr>
</tbody>
</table>

Virtual Hash Page Table
See the text Section 7.2.6, starting on page 299.
With paravirtualization (as in Xen), the guest OS in the virtual machine can cooperate with the VMM to produce a direct translation from virtual virtual memory to real memory.
Recent versions of the x86 architecture from Intel provide extended page tables (EPT) that add an extra layer of translation that’s invisible to the virtual machine. The guest OS in the virtual machine sets up the normal hardware page tables to translate from what it considers to be virtual memory (actually virtual virtual memory) to what it considers to be real memory (actually virtual real memory). The VMM sets up an extra translation step from virtual real to real. The hardware does the composite translation (i.e., it uses both page tables).
On the actual x86 architecture, the translation (by EPT) from virtual real to real must be done at each step in which a real address is needed. The VMM maintains the EPT base pointer register (EPTP), which points to a page directory and its associated page tables that map virtual-real memory to real memory. When the guest OS puts the (virtual real) address of what it wants to be the page directory into CR3, the address is automatically translated by EPT into a real address. When a full translation is done (from virtual virtual to real real), the entry in the page directory containing the virtual real address of the page table is translated by EPT to a real address, and similarly for the entry in the page table.
Quiz 3

We’d like to virtualize EPT. Assume that setting EPTP causes a VMexit if done on a VMM that’s not running in real ring -1. What does the VMM running at level 0 do when it receives such a VMexit from a VMM running at level 1?

a) nothing: the EPT mechanism is virtualized by the hardware

b) it sets EPTP to point to the composition of the page tables mapping the level 1 VM (on which the level 1 VMM runs) and the page tables pointed to by the value being attempted to be put in EPT

c) something else