- The concept of Limited Direct Exception is simple; for the most part, let the program run directly on the hardware;
    - however, at certain key points in time (such as when a process issues a system call, or a timer interrupt occurs), arrange so that the OS gets involved and makes sure the “right” thing happens.
- The OS, with a little hardware support, tries its best to get out of the way of the running program, to deliver an efficient virtualization; however, by interposing at those critical points in time, the OS ensures that it maintains control over the hardware.
    - Interposition is a generic and powerful technique that is often used to great effect in computer systems. In virtualizing memory, the hardware will interpose on each memory access, and translate each virtual address issued by the process to a physical address where the desired information is actually stored.
- How can we build an efficient virtualization of memory? How do we provide the flexibility needed by applications? How do we maintain control over which memory locations an application can access, and thus ensure that application memory accesses are properly restricted? How do we do all of this efficiently?
    - With hardware-based address translation, or address translation for short, the hardware transforms each memory access (e.g., an instruction fetch, load, or store), changing the virtual address provided by the instruction to a physical address where the desired information is actually located.
    - on each and every memory reference, an address translation is performed by the hardware to redirect application memory references to their actual locations in memory
- The OS must get involved at key points to set up the hardware so that the correct translations take place; it must thus manage memory, keeping track of which locations are free and which are in use
- Many programs are actually sharing memory at the same time, as the CPU (or CPUs) switches between running one program and the next.
    - It is through virtualization that it appears as if each process has its own memory

Assumptions

- assume for now that the user’s address space must be placed contiguously in physical memory
- Assume that the size of the address space is not too big; specifically, that it is less than the size of physical memory.
- Assume that each address space is exactly the same size.

An Example

- here is a short code sequence that loads a value from memory, increments it by three, and then stores the value back into memory.
    - void func() {
      int x = 3000;
      x = x + 3; // line of code we are interested in ...
- The compiler turns this line of code into assembly, which might look something like this
    - 128: movl 0x0(%ebx), %eax ;load 0+ebx into eax
    - 132: addl $0x03, %eax ;add 3 to eax register

    - 135: movl %eax, 0x0(%ebx) ;store eax back to mem
- The code presumes that the address of x has been placed in the register ebx, and then loads the value at that address into the general-purpose register eax using the movl instruction (for “longword” move). The next instruction adds 3 to eax, and the final instruction stores the value in eax back into memory at that same location
- When these instructions run, from the perspective of the process, the following memory accesses take place.
    - Fetch instruction at address 128
    - Execute this instruction (load from address 15 KB)
    - Fetch instruction at address 132
    - Execute this instruction (no memory reference)
    - Fetch the instruction at address 135
    - Execute this instruction (store to address 15 KB)
- From the program’s perspective, its address space starts at address 0 and grows to a maximum of 16 KB; all memory references it generates should be within these bounds.
    - However, to virtualize memory, the OS wants to place the process somewhere else in physical memory, not necessarily at address 0.
    - Thus, we have the problem: how can we relocate this process in memory in a way that is transparent to the process?

Dynamic (Hardware-based) Relocation

- Introduced in the first time-sharing machines of the late 1950’s is a simple idea referred to as base and bounds; the technique is also referred to as dynamic relocation;
    - we’ll need two hardware registers within each CPU: one is called the base register, and the other the bounds (sometimes called a limitregister).
    - This base-and-bounds pair is going to allow us to place the address space anywhere we’d like in physical memory, and do so while ensuring that the process can only access its own address space.
- However, when a program starts running, the OS decides where in physical memory it should be loaded and sets the base register to that value.
- when any memory reference is generated by the process, it is translated by the processor in the following manner:
    - physical address = virtual address + base
- Each memory reference generated by the process is a virtual address; the hardware in turn adds the contents of the base register to this address and the result is a physical address that can be issued to the memory system.
- For example, when we execute this instruction:
    - 128: movl 0x0(%ebx), %eax
        - The program counter (PC) is set to 128; when the hardware needs to fetch this instruction, it first adds the value to the base register value of 32 KB (32768) to get a physical address of 32896; the hardware then fetches the instruction from that physical address.
        - Next, the processor begins executing the instruction

        - At some point, the process then issues the load from virtual address 15 KB, which the processor takes and again adds to the base register (32 KB), getting the final physical address of 47 KB and thus the desired contents
- Transforming a virtual address into a physical address is exactly the technique we refer to as address translation; that is, the hardware takes a virtual address the process thinks it is referencing and transforms it into a physical address which is where the data actually resides
- Because this relocation of the address happens at runtime, and because we can move address spaces even after the process has started running, the technique is often referred to as dynamic relocation
    - what happened to that bounds (limit) register? After all, isn't this the base and bounds approach?
        - Indeed, it is, as the bounds register is there to help with protection. Specifically, the processor will first check that the memory reference is within bounds to make sure it is legal; in the simple example above, the bounds register would always be set to 16 KB.
        - If a process generates a virtual address that is greater than (or equal to) the bounds, or one that is negative, the CPU will raise an exception, and the process will likely be terminated.
        - The point of the bounds is thus to make sure that all addresses generated by the process are legal and within the "bounds" of the process.
- the base and bounds registers are hardware structures kept on the chip (one pair per CPU). Sometimes people call the part of the processor that helps with address translation the memory management unit (MMU)
- as we develop more sophisticated memorymanagement techniques, we will be adding more circuitry to the MMU.
- Bound registers can be defined in one of two ways.
    - In one way (as above), it holds the size of the address space, and thus the hardware checks the virtual address against it first before adding the base.
    - In the second way, it holds the physical address of the end of the address space, and thus the hardware first adds the base and then makes sure the address is within bounds.

Hardware Support: A Summary

- In CPU virtualization, we require two different CPU modes. The OS runs in privileged mode (or kernel mode), where it has access to the entire machine; applications run in user mode, where they are limited in what they can do.
    - A single bit, perhaps stored in some kind of processor status word, indicates which mode the CPU is currently running in; upon certain special occasions (e.g., a system call or some other kind of exception or interrupt), the CPU switches modes
- The hardware must also provide the base and bounds registers themselves; each CPU thus has an additional pair of registers, part of the memory management unit (MMU) of the CPU.

    - When a user program is running, the hardware will translate each address, by adding the base value to the virtual address generated by the user program. The hardware must also be able to check whether the address is valid, which is accomplished by using the bounds register and some circuitry within the CPU.
- The hardware should provide special instructions to modify the base and bounds registers, allowing the OS to change them when different processes run. These instructions are privileged; only in kernel (or privileged) mode can the registers be modified.
- The OS must track which parts of free memory are not in use, so as to be able to allocate memory to processes. Many different data structures can of course be used for such a task; the simplest (which we will assume here) is a free list, which simply is a list of the ranges of the physical memory which are not currently in use.
- the CPU must be able to generate exceptions in situations where a user program tries to access memory illegally (with an address that is “out of bounds”);
    - In this case, the CPU should stop executing the user program and arrange for the OS “out-of-bounds” exception handler to run. The OS handler can then figure out how to react, in this case likely terminating the process
    - Similarly, if a user program tries to change the values of the (privileged) base and bounds registers, the CPU should raise an exception and run the “tried to execute a privileged operation while in user mode” handler. The CPU also must provide a method to inform it of the location of these handlers; a few more privileged instructions are thus needed

Hardware Requirements	Notes
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

Figure 15.3: **Dynamic Relocation: Hardware Requirements**

OS Issues

- Just as the hardware provides new features to support dynamic relocation, the OS now has new issues it must handle; the combination of hardware support and OS management leads to the implementation of a simple virtual memory
- First, the OS must take action when a process is created, finding space for its address space in memory.
    - Fortunately, given our assumptions that each address space is (a) smaller than the size of physical memory and (b) the same size, this is quite easy for the OS; it can simply view physical memory as an array of slots, and track whether each one is free or in use.
- When a new process is created, the OS will have to search a data structure (often called a free list) to find room for the new address space and then mark it used.
- Second, the OS must do some work when a process is terminated (i.e., when it exits gracefully, or is forcefully killed), reclaiming all of its memory for use in other processes

or the OS. Upon termination of a process, the OS thus puts its memory back on the free list, and cleans up any associated data structures as need be.

- Third, the OS must also perform a few additional steps when a context switch occurs.
    - The OS must save and restore the base-and-bounds pair when it switches between processes.
    - Specifically, when the OS decides to stop running a process, it must save the values of the base and bounds registers to memory, in some per-process structure such as the process structure or process control block (PCB).
- when the OS resumes a running process (or runs it the first time), it must set the values of the base and bounds on the CPU to the correct values for this process
- when a process is stopped (i.e., not running), it is possible for the OS to move an address space from one location in memory to another rather easily.
    - To move a process's address space, the OS first deschedules the process; then, the OS copies the address space from the current location to the new location; finally, the OS updates the saved base register (in the process structure) to point to the new location.
    - When the process is resumed, its (new) base register is restored, and it begins running again, oblivious that its instructions and data are now in a completely new spot in memory
- Fourth, the OS must provide exception handlers, or functions to be called, as discussed above; the OS installs these handlers at boot time (via privileged instructions).
    - For example, if a process tries to access memory outside its bounds, the CPU will raise an exception; the OS must be prepared to take action when such an exception arises.

Summary

- we have extended the concept of limited direct execution with a specific mechanism used in virtual memory, known as address translation.
- With address translation, the OS can control each and every memory access from a process, ensuring the accesses stay within the bounds of the address space.
    - Key to the efficiency of this technique is hardware support, which performs the translation quickly for each access, turning virtual addresses (the process's view of memory) into physical ones (the actual view).
    - All of this is performed in a way that is transparent to the process that has been relocated; the process has no idea its memory references are being translated, making for a wonderful illusion
- We have also seen one particular form of virtualization, known as base and bounds or dynamic relocation. Base-and-bounds virtualization is quite efficient, as only a little more hardware logic is required to add a base register to the virtual address and check that the address generated by the process is in bounds.
- Base-and-bounds also offers protection; the OS and hardware combine to ensure no process can generate memory references outside its own address space
- this simple technique of dynamic relocation does have its inefficiencies.

- Whenever a memory block gets allocated with a process, and in case the process happens to be smaller than the total amount of requested memory, a free space is ultimately created in this memory block. And due to this, the memory block's free space is unused. This is what causes internal fragmentation.