15

Mechanism: Address Translation

In developing the virtualization of the CPU, we focused on a general mechanism known as **limited direct execution** (or **LDE**). The idea behind LDE 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. Thus, 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. Efficiency and control together are two of the main goals of any modern operating system.

In virtualizing memory, we will pursue a similar strategy, attaining both efficiency and control while providing the desired virtualization. Efficiency dictates that we make use of hardware support, which at first will be quite rudimentary (e.g., just a few registers) but will grow to be fairly complex (e.g., TLBs, page-table support, and so forth, as you will see). Control implies that the OS ensures that no application is allowed to access any memory but its own; thus, to protect applications from one another, and the OS from applications, we will need help from the hardware here too. Finally, we will need a little more from the VM system, in terms of *flexibility*; specifically, we'd like for programs to be able to use their address spaces in whatever way they would like, thus making the system easier to program. And thus we arrive at the refined crux:

THE CRUX:
HOW TO EFFICIENTLY AND FLEXIBLY VIRTUALIZE MEMORY

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?

The generic technique we will use, which you can consider an addition to our general approach of limited direct execution, is something that is referred to as **hardware-based address translation**, or just **address translation** for short. With address translation, 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. Thus, 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.

Of course, the hardware alone cannot virtualize memory, as it just provides the low-level mechanism for doing so efficiently. 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, and judiciously intervening to maintain control over how memory is used.

Once again the goal of all of this work is to create a beautiful **illusion**: that the program has its own private memory, where its own code and data reside. Behind that virtual reality lies the ugly physical truth: that many programs are actually sharing memory at the same time, as the CPU (or CPUs) switches between running one program and the next. Through virtualization, the OS (with the hardware's help) turns the ugly machine reality into something that is a useful, powerful, and easy to use abstraction.

15.1 Assumptions

Our first attempts at virtualizing memory will be very simple, almost laughably so. Go ahead, laugh all you want; pretty soon it will be the OS laughing at you, when you try to understand the ins and outs of TLBs, multi-level page tables, and other technical wonders. Don't like the idea of the OS laughing at you? Well, you may be out of luck then; that's just how the OS rolls.

Specifically, we will assume for now that the user's address space must be placed *contiguously* in physical memory. We will also assume, for simplicity, that the size of the address space is not too big; specifically, that it is *less than the size of physical memory*. Finally, we will also assume that each address space is exactly the *same size*. Don't worry if these assumptions sound unrealistic; we will relax them as we go, thus achieving a realistic virtualization of memory.

15.2 An Example

To understand better what we need to do to implement address translation, and why we need such a mechanism, let's look at a simple example. Imagine there is a process whose address space is as indicated in Figure 15.1. What we are going to examine here is a short code sequence

TIP: INTERPOSITION IS POWERFUL

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. However, the general technique of interposition is much more broadly applicable; indeed, almost any well-defined interface can be interposed upon, to add new functionality or improve some other aspect of the system. One of the usual benefits of such an approach is **transparency**; the interposition often is done without changing the client of the interface, thus requiring no changes to said client.

that loads a value from memory, increments it by three, and then stores the value back into memory. You can imagine the C-language representation of this code might look like this:

```
void func() {
    int x = 3000; // thanks, Perry.
    x = x + 3;    // this is the line of code we are interested in
```

The compiler turns this line of code into assembly, which might look something like this (in x86 assembly). Use `objdump` on Linux or `otool` on a Mac to disassemble it:

```
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
```

This code snippet is relatively straightforward; it 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.

In Figure 15.1 (page 4), you can see how both the code and data are laid out in the process's address space; the three-instruction code sequence is located at address 128 (in the code section near the top), and the value of the variable `x` at address 15 KB (in the stack near the bottom). In the figure, the initial value of `x` is 3000, as shown in its location on the stack.

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)

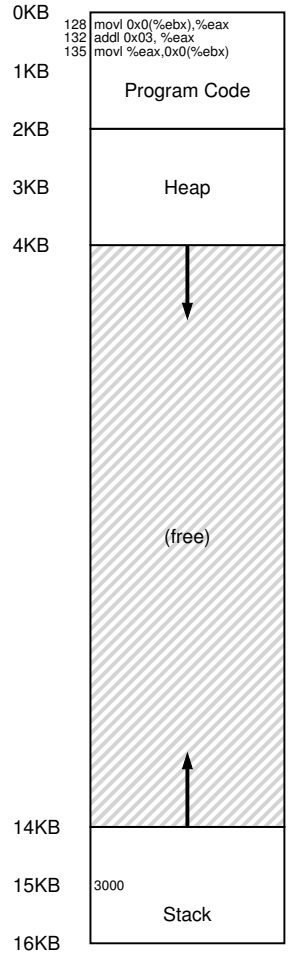

Figure 15.1: **A Process And Its Address Space**

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? How can we provide the illusion of a virtual address space starting at 0, when in reality the address space is located at some other physical address?

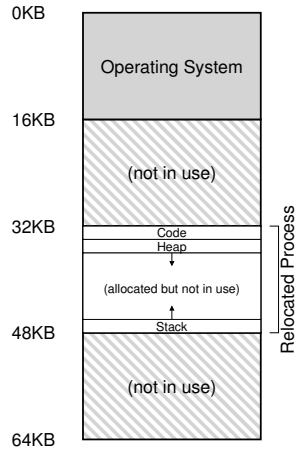

Figure 15.2: **Physical Memory with a Single Relocated Process**

An example of what physical memory might look like once this process's address space has been placed in memory is found in Figure 15.2. In the figure, you can see the OS using the first slot of physical memory for itself, and that it has relocated the process from the example above into the slot starting at physical memory address 32 KB. The other two slots are free (16 KB-32 KB and 48 KB-64 KB).

15.3 Dynamic (Hardware-based) Relocation

To gain some understanding of hardware-based address translation, we'll first discuss its first incarnation. 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 use both terms interchangeably [SS74].

Specifically, we'll need two hardware registers within each CPU: one is called the **base** register, and the other the **bounds** (sometimes called a **limit** register). 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.

In this setup, each program is written and compiled as if it is loaded at address zero. 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. In the example above, the OS decides to load the process at physical address 32 KB and thus sets the base register to this value.

Interesting things start to happen when the process is running. Now, when any memory reference is generated by the process, it is **translated** by the processor in the following manner:

```
physical address = virtual address + base
```

ASIDE: **SOFTWARE-BASED RELOCATION**

In the early days, before hardware support arose, some systems performed a crude form of relocation purely via software methods. The basic technique is referred to as **static relocation**, in which a piece of software known as the **loader** takes an executable that is about to be run and rewrites its addresses to the desired offset in physical memory.

For example, if an instruction was a load from address 1000 into a register (e.g., `movl 1000, %eax`), and the address space of the program was loaded starting at address 3000 (and not 0, as the program thinks), the loader would rewrite the instruction to offset each address by 3000 (e.g., `movl 4000, %eax`). In this way, a simple static relocation of the process's address space is achieved.

However, static relocation has numerous problems. First and most importantly, it does not provide protection, as processes can generate bad addresses and thus illegally access other process's or even OS memory; in general, hardware support is likely needed for true protection [WL+93]. Another negative is that once placed, it is difficult to later relocate an address space to another location [M65].

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.

To understand this better, let's trace through what happens when a single instruction is executed. Specifically, let's look at one instruction from our earlier sequence:

```
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** [M65].

TIP: HARDWARE-BASED DYNAMIC RELOCATION

With dynamic relocation, a little hardware goes a long way. Namely, a **base** register is used to transform virtual addresses (generated by the program) into physical addresses. A **bounds** (or **limit**) register ensures that such addresses are within the confines of the address space. Together they provide a simple and efficient virtualization of memory.

Now you might be asking: what happened to that bounds (limit) register? After all, isn't this the base *and* bounds approach? Indeed, it is. As you might have guessed, 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 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.

We should note that 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 memory-management techniques, we will be adding more circuitry to the MMU.

A small aside about bound registers, which 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. Both methods are logically equivalent; for simplicity, we'll usually assume the former method.

Example Translations

To understand address translation via base-and-bounds in more detail, let's take a look at an example. Imagine a process with an address space of size 4 KB (yes, unrealistically small) has been loaded at physical address 16 KB. Here are the results of a number of address translations:

Virtual Address		Physical Address
0	→	16 KB
1 KB	→	17 KB
3000	→	19384
4400	→	*Fault (out of bounds)*

As you can see from the example, it is easy for you to simply add the base address to the virtual address (which can rightly be viewed as an *offset* into the address space) to get the resulting physical address. Only if the virtual address is "too big" or negative will the result be a fault, causing an exception to be raised.

ASIDE: **DATA STRUCTURE — THE FREE LIST**

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.

15.4 Hardware Support: A Summary

Let us now summarize the support we need from the hardware (also see Figure 15.3, page 9). First, as discussed in the chapter on 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. Imagine the havoc a user process could wreak[1] if it could arbitrarily change the base register while running. Imagine it! And then quickly flush such dark thoughts from your mind, as they are the ghastly stuff of which nightmares are made.

Finally, 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.

[1] Is there anything other than "havoc" that can be "wreaked"?

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**

15.5 Operating System 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. Specifically, there are a few critical junctures where the OS must get involved to implement our base-and-bounds version of 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. With variable-sized address spaces, life is more complicated, but we will leave that concern for future chapters.

Let's look at an example. In Figure 15.2 (page 5), you can see the OS using the first slot of physical memory for itself, and that it has relocated the process from the example above into the slot starting at physical memory address 32 KB. The other two slots are free (16 KB-32 KB and 48 KB-64 KB); thus, the **free list** should consist of these two entries.

Second, the OS must do some work when a process is terminated (i.e., when it exits gracefully, or is forcefully killed because it misbehaved), 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. There is only one base and bounds register pair on each CPU, after all, and their values differ for each running program, as each program is loaded at a different physical address in memory. Thus, the OS must *save and restore* the base-and-bounds pair when it switches be-

OS Requirements	Notes
Memory management	*Need to allocate memory for new processes;* *Reclaim memory from terminated processes;* *Generally manage memory via **free list***
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise;* *likely action is to terminate offending process*

Figure 15.4: **Dynamic Relocation: Operating System Responsibilities**

tween 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). Similarly, 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.

We should note that 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. The common reaction of the OS will be one of hostility: it will likely terminate the offending process. The OS should be highly protective of the machine it is running, and thus it does not take kindly to a process trying to access memory or execute instructions that it shouldn't. Bye bye, misbehaving process; it's been nice knowing you.

Figure 15.5 (page 11) illustrates much of the hardware/OS interaction in a timeline. The figure shows what the OS does at boot time to ready the machine for use, and then what happens when a process (Process A) starts running; note how its memory translations are handled by the hardware with no OS intervention. At some point, a timer interrupt occurs, and the OS switches to Process B, which executes a "bad load" (to an illegal memory address); at that point, the OS must get involved, terminating the process and cleaning up by freeing B's memory and removing its entry from the process table. As you can see from the diagram, we are still following the basic approach of **limited direct execution**. In most cases, the OS just sets up the hardware appropriately and lets the process run directly on the CPU; only when the process misbehaves does the OS have to become involved.

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember addresses of...	
	system call handler	
	timer handler	
	illegal mem-access handler	
	illegal instruction handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table		
initialize free list		

OS @ run (kernel mode)	Hardware	Program (user mode)
To start process A:		
allocate entry in process table		
allocate memory for process		
set base/bounds registers		
return-from-trap (into A)		
	restore registers of A	
	move to **user mode**	
	jump to A's (initial) PC	
		Process A runs
		Fetch instruction
	Translate virtual address	
	and perform fetch	
		Execute instruction
	If explicit load/store:	
	Ensure address is in-bounds;	
	Translate virtual address	
	and perform load/store	
		...
	Timer interrupt	
	move to **kernel mode**	
	Jump to interrupt handler	
Handle the trap		
Call `switch()` routine		
save regs(A) to proc-struct(A)		
(including base/bounds)		
restore regs(B) from proc-struct(B)		
(including base/bounds)		
return-from-trap (into B)		
	restore registers of B	
	move to **user mode**	
	jump to B's PC	
		Process B runs
		Execute bad load
	Load is out-of-bounds;	
	move to **kernel mode**	
	jump to trap handler	
Handle the trap		
Decide to terminate process B		
de-allocate B's memory		
free B's entry in process table		

Figure 15.5: **Limited Direct Execution Protocol (Dynamic Relocation)**

15.6 Summary

In this chapter, 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. Protection is certainly one of the most important goals of the OS; without it, the OS could not control the machine (if processes were free to overwrite memory, they could easily do nasty things like overwrite the trap table and take over the system).

Unfortunately, this simple technique of dynamic relocation does have its inefficiencies. For example, as you can see in Figure 15.2 (page 5), the relocated process is using physical memory from 32 KB to 48 KB; however, because the process stack and heap are not too big, all of the space between the two is simply *wasted*. This type of waste is usually called **internal fragmentation**, as the space *inside* the allocated unit is not all used (i.e., is fragmented) and thus wasted. In our current approach, although there might be enough physical memory for more processes, we are currently restricted to placing an address space in a fixed-sized slot and thus internal fragmentation can arise[2]. Thus, we are going to need more sophisticated machinery, to try to better utilize physical memory and avoid internal fragmentation. Our first attempt will be a slight generalization of base and bounds known as **segmentation**, which we will discuss next.

[2] A different solution might instead place a fixed-sized stack within the address space, just below the code region, and a growing heap below that. However, this limits flexibility by making recursion and deeply-nested function calls challenging, and thus is something we hope to avoid.

References

[M65] "On Dynamic Program Relocation"
W.C. McGee
IBM Systems Journal
Volume 4, Number 3, 1965, pages 184–199
This paper is a nice summary of early work on dynamic relocation, as well as some basics on static relocation.

[P90] "Relocating loader for MS-DOS .EXE executable files"
Kenneth D. A. Pillay
Microprocessors & Microsystems archive
Volume 14, Issue 7 (September 1990)
An example of a relocating loader for MS-DOS. Not the first one, but just a relatively modern example of how such a system works.

[SS74] "The Protection of Information in Computer Systems"
J. Saltzer and M. Schroeder
CACM, July 1974
From this paper: "The concepts of base-and-bound register and hardware-interpreted descriptors appeared, apparently independently, between 1957 and 1959 on three projects with diverse goals. At M.I.T., McCarthy suggested the base-and-bound idea as part of the memory protection system necessary to make time-sharing feasible. IBM independently developed the base-and-bound register as a mechanism to permit reliable multiprogramming of the Stretch (7030) computer system. At Burroughs, R. Barton suggested that hardware-interpreted descriptors would provide direct support for the naming scope rules of higher level languages in the B5000 computer system." We found this quote on Mark Smotherman's cool history pages [S04]; see them for more information.

[S04] "System Call Support"
Mark Smotherman, May 2004
http://people.cs.clemson.edu/~mark/syscall.html
A neat history of system call support. Smotherman has also collected some early history on items like interrupts and other fun aspects of computing history. See his web pages for more details.

[WL+93] "Efficient Software-based Fault Isolation"
Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham
SOSP '93
A terrific paper about how you can use compiler support to bound memory references from a program, without hardware support. The paper sparked renewed interest in software techniques for isolation of memory references.

Homework

The program `relocation.py` allows you to see how address translations are performed in a system with base and bounds registers. See the README for details.

Questions

1. Run with seeds 1, 2, and 3, and compute whether each virtual address generated by the process is in or out of bounds. If in bounds, compute the translation.
2. Run with these flags: `-s 0 -n 10`. What value do you have set `-l` (the bounds register) to in order to ensure that all the generated virtual addresses are within bounds?
3. Run with these flags: `-s 1 -n 10 -l 100`. What is the maximum value that base can be set to, such that the address space still fits into physical memory in its entirety?
4. Run some of the same problems above, but with larger address spaces (`-a`) and physical memories (`-p`).
5. What fraction of randomly-generated virtual addresses are valid, as a function of the value of the bounds register? Make a graph from running with different random seeds, with limit values ranging from 0 up to the maximum size of the address space.