CS162, Spring 2004 **Discussion #7** **Amir Kamil**
UC Berkeley **3/4/04**

Topics: Address Translation

1 Announcements

- Group evaluations should be done by 3/5. You will lose points if anyone in your group doesn't fill one out.
- Midterm Thursday, March 18th, 2004, 5:30-7:30pm, 155 Dwinelle
- Project 2 initial design doc due next Wednesday, March 10, 2004 @ 11:59pm
- Project stuff:
  1. You should be meeting regularly and often, at least every other day.
  2. Take notes at your meetings, and make sure you notify group members of changes through email.
  3. For each part, the person(s) doing should be writing glass box tests, but somebody else should do the black box tests.
  4. Program defensively: use asserts, requires/effects/modifies (assert preconditions). This simplifies debugging, since your code will fail closer to the point of error.
  5. For the final design documents, mark the sections you revised.
  6. Do not change your ssh keys; if you have trouble logging into CVS, send mail to inst.

2 Address Translation

2.1 Introduction

In a multiprogramming environment, malicious programs may attempt to corrupt other processes or even the operating system by accessing their allocated physical memory. In order to protect against this, modern systems provide address space abstractions, where each process has a *virtual address space* and can only access the portion of physical memory allocated to its virtual space. This requires a process's memory accesses to be translated from virtual to physical addresses. In this section, we briefly look at the hardware requirements for implementing this, and discuss various types of address translation.

2.2 Requirements

Address translation requires specialize hardware to translate virtual addresses to physical, in the form of a *memory management unit (MMU)*. This hardware must support different mappings for different processes in order to provide protection between their spaces. In addition, the operating system must be able to directly address physical memory, in order to allocate resources and process I/O, for example. So the hardware must support *dual-mode operation*, where user programs run in *user mode* with address translation but the operating system runs in *kernel mode* without it. Some privileged operations, such as access to page tables, must also be allowed in kernel mode.

The hardware must also allow transitions between user and kernel mode through system calls and interrupts. System calls are implemented by a special instruction invoked by user programs, which traps to the operating system. The OS then examines the arguments of the call, jumps to the appropriate kernel function, and performs the desired operation. Interrupts must also transition to the kernel, in order to complete I/O or allow some other privileged operation to occur.

2.3 Translation Schemes

2.3.1 No Translation

It is possible to support multiprogramming without providing a virtual address space abstraction. The operating system can place multiple programs in different locations of physical memory, and at execution time, the loader can translate all memory references according to where the program is in memory. While these scheme is fast, since it requires no runtime translation, there is no protection between processes.

2.3.2 Base and Bounds

An alternative to no translation that provides memory protection is the *base and bounds* method. In this scheme, a process is loaded into contiguous memory, and a base register holds the address where the process starts in memory, and a bounds register holds where it ends. Then at runtime, a virtual memory access is translated by the MMU by adding the base to the given address, and comparing the result to the bounds to make sure it doesn't exceed it. For example, suppose a process starts at address `0x532A032B` and ends at `0x6CD93E21`. Now when it accesses the virtual address `0x103AD098`, the MMU translates this to `0x103AD098 + 0x532A032B = 0x6364D3C3`, makes sure that this is less than `0x6CD93E21`, and performs the access. But if it instead accesses the address `0x203AD098`, the MMU translates this to `0x7364D3C3`, determines that this exceeds the bounds, and throws an error.

Base and bounds provides address space protection and is quite cheap, requiring only two registers and an add and compare for each memory access. However, it is quite difficult to implement sharing and processes must be contiguous in memory. The latter results in problems when there is enough free space to load a process into memory, but the space is split or *fragmented* into different areas of memory. In addition, increasing the memory allocated to a process is also difficult. Either the base or bounds must be moved, but this can't happen if the memory below and above a process is in use. The only solution to these last two problems is to move processes around in memory, a very expensive operation. Other problems include the fact that no distinction is made between logically distinct chunks of memory, such as code, heap, and stack, and that the heap and stack can't both grow dynamically; it is best to put the heap and stack as far apart in virtual memory as possible, so that they can grow towards each other.

2.3.3 Segmentation

We can fix some of the problems of base and bounds by generalizing the scheme to use *segments*, or logically contiguous pieces of memory. The address space of a process is divided into multiple parts, for example code, data, and stack, and each segment has its own base and bounds register. A virtual address is now divided into two pieces: the segment number (usually the most significant few bits), and offset in the segment. The MMU now translates an address by looking up the base for the appropriate segment, adding to it the offset, and comparing with the bounds of the segment.

Segmentation has advantages over base and bounds. Whole segments can be shared between processes, and each segment can have some protection info; for example, the code section could be read-only. The heap and stack can now grow independently. Segmentation is still cheap, since only a small number of segments are needed, so only a few registers are required. However, fragmentation is still a problem, and increasing the size of a segment still may require moving memory around.

2.3.4 Paging

A simplification of segmentation is to use fixed-size segments. Such segments are called *pages*, and the translation scheme *paging*. Physical memory is also divided into pages, so that each virtual page corresponds to exactly one physical page. Each page still belongs to a logically contiguous piece of memory, though each contiguous piece can be composed of multiple pages. Pages are usually small, and there are many of them, so a *page table* is required in memory that maps virtual pages to physical pages. The MMU uses a *page table base register* to access the page table of a given process, looks up a virtual page number of an address in the table, and replaces it with the appropriate physical page number.

Paging greatly simplifies memory allocation, since an allocation of a virtual page can be satisfied by any physical page. Logically contiguous pieces of virtual memory don't actually have to be contiguous in physical memory, and entire pages can be shared. However, paging is more costly than segmentation, as it requires a sizeable page table in memory and a memory lookup to translate each memory access. In addition, paging suffers from *internal fragmentation*, where most of the space inside a page is unused.

We will see more details of paging next week, including quantification of costs and paging schemes to reduce them.