Overview

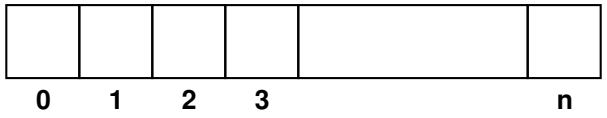
- In addition to providing an extended machine, an operating system manages resources
- We have covered how access to the CPU is managed, now we examine another important resource, memory
- We start off by considering what we want from a memory manager before examining how memory management techniques have evolved to meet those requirements
- Important concepts covered along the way include: address translation, logical address space, physical address space and virtual memory
- We finish by looking at virtual memory in Linux

Memory Management Objectives

- Ideally a memory manager should:
  1. Handle memory allocations efficiently and effectively
  2. Exploit multilevel technologies in the memory hierarchy
  3. Support dynamic relocation
  4. Provide memory protection
  5. Support memory sharing
  6. Extend memory beyond physical memory limits

Memory Allocation

- Physical memory is a linear, one-dimensional structure
- Like with any array, the range of valid addresses goes from zero to some maximum
- However, a program is not linear in design but composed of sections e.g. instructions and data
- Also a running program's memory requirements change over time



Memory Allocation

- We would like our memory manager to support viewing an executable as being composed of sections and to allow appropriate permissions to be applied to those sections
- For example an executable's instructions should not change and we would like to mark that memory readable and executable but not writable and mark the data section readable and writable but not executable
- Memory allocation should support both dynamic memory requirements of a process and its structured make-up
- Thus, the memory manager should give out separate segments of memory and not a single large chunk

Memory Allocation

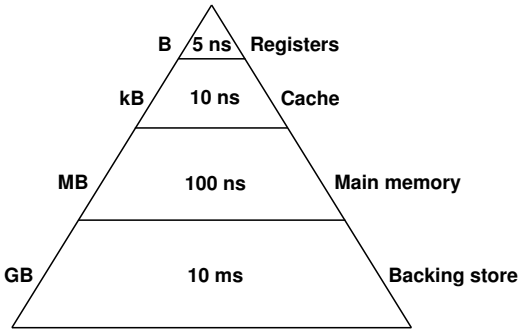
- So what would be the advantages of representing an address space as a collection of segments?
  1. With segmented memory we could give different levels of protection to each segment (any combination of `rwx`)
  2. When looking for space in physical memory it would be easier to find and manage segments rather than find a single large space to hold an entire executable
  3. When sharing is required, rather than sharing full process address spaces, we could share individual segments between processes

Multilevel Storage

- The memory hierarchy reflects a speed/cost compromise and it is typically four levels deep
- At the top of the pyramid is the fastest memory, the CPU registers where some of the program will be stored
- Also available is fast memory close to the CPU, caches where recently used instructions and data will be kept
- Below the caches lies main memory or RAM, more of the program will be kept here
- At the bottom of the pyramid is the hard disk where the on-disk copy of executable itself is kept, this is the cheapest but slowest memory of all

The Memory Hierarchy

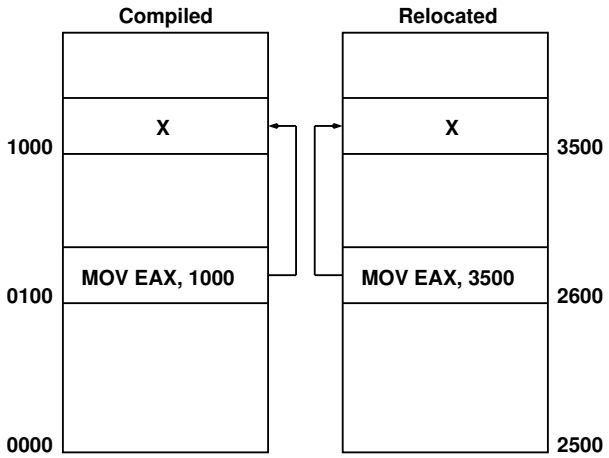
- The memory manager must work with all and ensure each of these levels works efficiently with its neighbours



Dynamic Relocation

- A program should not always have to be loaded into physical memory at the same location so we must disallow absolute physical memory references in our code i.e. variable `x` will not always be at address `1000`
- How are variables then referenced in code? Compiler assumes all programs are loaded at start address 0
- When run, a relocating loader could adjust each memory access to reflect the program's actual load address i.e. perform static relocation at startup
- But we may want to move a program after loading it, to dynamically relocate it, and we want to do so efficiently

Static Relocation



Memory Protection

- One user must not be able to read from or write to memory belonging to another user or to the operating system
- Thus every memory reference must be preceded by a legality check carried out at run time by the memory manager hardware to ensure the requesting process has the privileges required to access the specified memory

Memory Sharing

- Sometimes we want more than one process to be able to access the same memory whether it be data or code (e.g. shared libraries/DLLs)
- Thus our memory manager should support the sharing of memory

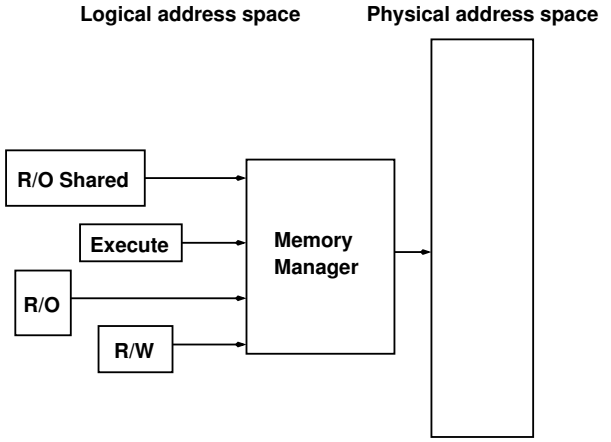
Memory Extension

- The physical address space is limited by the actual amount of memory installed e.g. 512MB, 1GB etc.
- How much memory does a process see? Well a process sees a logical address space whose size is limited only by the number of bits in a register i.e. the logical address space is the range of addresses that a CPU can generate
- We typically have a 32-bit (4GB) logical address space and thus the logical is larger than the physical address space
- We want to provide the illusion to each process that it has 4GB of memory to play with and we want to concurrently run programs whose combined size is larger than the physical address space

Achieving Our Objectives

- As it turns out, the memory manager can achieve its objectives through an address translation mechanism
- This mechanism translates the logical addresses used by a process to actual physical memory locations i.e. it maps from the logical to the physical address space
- This mapping means each process uses the same logical address range but each one maps to a different set of physical addresses where that process's code and data are actually stored in physical memory
- We will even allow mapping from logical addresses to backing store to extend memory giving virtual memory

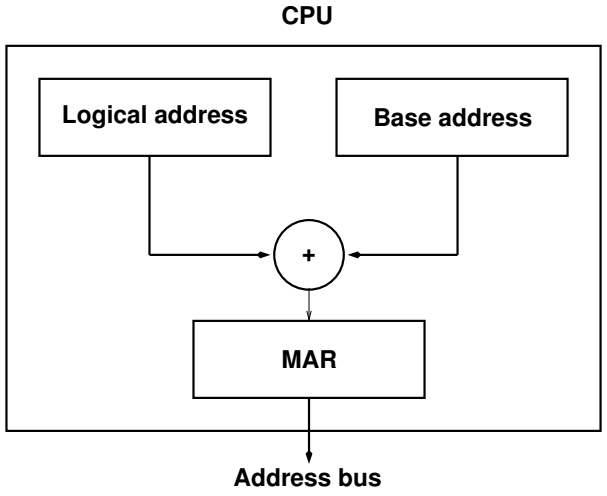
Logical and Physical Address Spaces



Base and Length Registers

- Our first step along the road to implementing this translation is to introduce a memory base register
- It won't meet all of our memory manager requirements but is a good start, so how does it work?
- Compiled program is assumed to begin at address 0 and executes as such i.e. no static relocation at load time
- However, the actual address where the program is loaded in physical memory is saved in a CPU register
- This offset is added to each logical address
- Moving a program in memory is now straightforward. . .

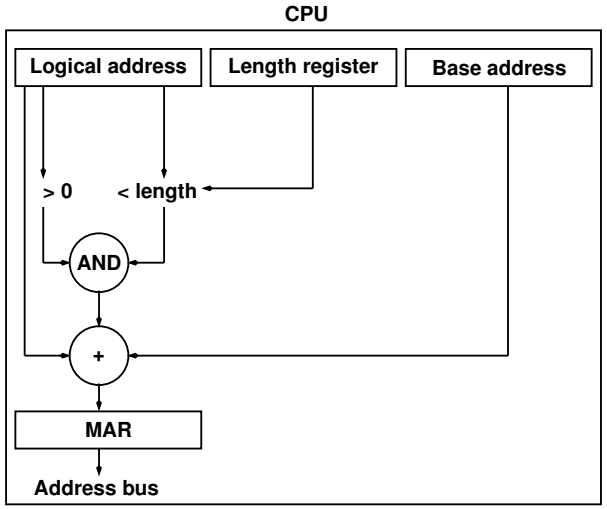
Adding a Base Register



Providing Protection

- So far so good but a programmer can still reference any physical memory location by supplying the appropriate logical address and that's bad news for security
- So we add a length register defining the number of bytes allocated to this process
- When memory is referenced, we check that the logical address is both greater than zero and less than length before adding in the actual physical address where the program is loaded
- Failing either test gives a memory access violation

Adding a Length Register



Placement Policies

- As variable-sized segments of memory are allocated to processes and later freed physical (we are still allocating a single block of memory per process) memory becomes a jumble of allocated and free areas
- A question then arises: Where do we load a new program?
- There are three options:
  1. Best fit: Put the program in smallest free block; but the leftover bit is probably useless
  2. Worst fit: Put the program in largest free block; leftover bit might still be useful
  3. First fit: Put the program in first big enough free block

Fragmentation

- Even with clever placement policies we will eventually end up with many uselessly small chunks of free memory as processes are born and die
- Memory becomes fragmented (just like your hard disk)
- What can we do about it?
- We need to run a compaction routine in order to shuffle memory contents about to produce larger free chunks
- Obviously compaction is only feasible if programs are dynamically relocatable and ours now are since we have incorporated an adjustable base register