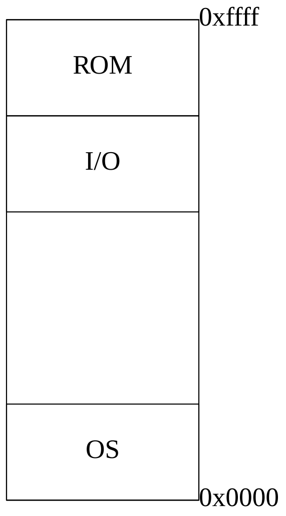
Memory Management

Physical Memory

After considering the CPU for the last few weeks, we're ready to look at a new resource: memory. A hardware memory is a linear array of words. You can think of it as a function mapping an address to a value. Memories vary by the size of *address space,* that is the number of words they store, and their *addressing granularity,* or the size of the words they return. Most modern memories are byte (8-bit word) addressable. Multiple hardware memories are used to provide a larger address space.

Frequently other hardware is multiplexed to the address space:



This is called memory mapping. In a memory-mapped I/O system, preventing a process from accessing devices directly is equivalent to preventing the process from accessing certain ranges of *physical* memory. Physical memory refers to storage provided by real devices. We'll see other kinds.

Memory Protection

When 2 or more processes share the same physical memory, the OS must protect them from writing into each others' memory. This problem is called memory protection, and it's is made more difficult because memory access is such a basic attribute of CPUs. `LOAD` and `STORE` operations are in practically all instruction sets. How much memory protection the OS can provide is generally determined by what mechanisms the hardware (or the external Memory Management Unit (MMU)) provides. For any of these to work, the MMU or CPU registers that control memory protection can only be manipulated in supervisor mode.

Base/Limit Registers

The simplest method of enforcing memory protection is adding 2 registers to the CPU, a *base* and a *limit* that demarcate a range of memory to which valid references can be made. References outside that range cause a trap.

This works great long as all memory is allocated contiguously. Non contiguous memory is harder to protect. Note that sharing between more than 2 processes is also hard.

Memory keys

A second approach (from the IBM/360) is to break up all of memory into 2K pages and assign a 4 bit key to each. Add a register to the CPU that holds the process's current key. References where the process's key and the page's key match are allowed, otherwise trap. Again the key can only be changed in supervisor mode.

This allows pretty general memory protection and allocation.

Relocation

Related to the problem of protection is the problem of relocation - running the same code at different places in memory. CPUs tend to support absolute addressing which means that code runs differently when loaded different places (the subroutines aren't where you want them to be...). There are 2 main ways to make code relocatable:

- Have the complier/linker mark all the addresses in the code and rewrite them all when the program is loaded. This adds startup overhead.
- Use only relative addressing, and require the OS to set a base register to the beginning of the code. This is particularly useful in base/limit protected systems because the same base register can be used for relocation and protection.

  This is nice for a couple reasons. The same code can run under multiple base registers (and may only need to be loaded once if there's a separate base/limit pair for code and data). All relative code can also be relocated while it's running.

Note that relocation without protection is dangerous. Unless fairly difficult precautions are taken to prevent a program from doing so, it is possible to create an absolute addressing instruction, and use it to access some other process's memory. Memory protection prevents the access when it's made.