CS61 lecture 10/23/12

1 Midterm, Problem 5B:

- We can simplify the problem by thinking in 8 bits!

  The original 16-bits shits

  $$\text{cute[i]} = ((\text{cute[i]} >> 1)\&0\text{x}5555)|((\text{cute[i]} << 1)\&0\text{xaaaa});$$

  are simplified into 8-bit shifts

  $$\text{cute[i]} = ((\text{cute[i]} >> 1)\&0\text{x}55)|((\text{cute[i]} << 1)\&0\text{xaa});$$

- We can also see that all the bitwise-and's are complements of each other

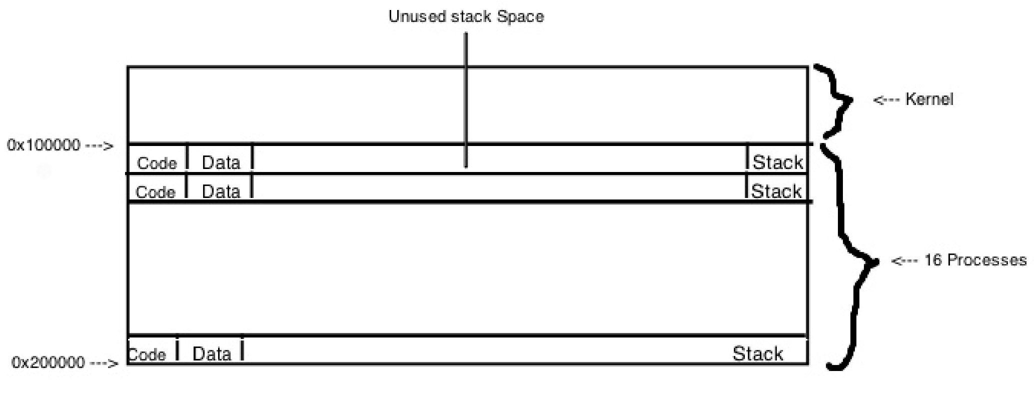
  for example 0x5555+0xAAAA = 0xffff

2 x86 virtual memory

2.1 Page table organization

Assumption We first consider the case that no two processes ever have the same virtual address.

Memory Layout Kernel, process, process space is 2^{20} bytes total. Each process uses 2^{16} bytes of address space. The details are described in the graph below:



Note 1 Notice that the code and data live at the bottom, stack at the top, and in between live the space for heap and for the stack to grow.

Note 2 We can only fit $2^{20}/2^{16} = 2^4 = 16$ processes at a time into our memory because of our assumption that the virtual memmories can't overlap. In other words, we can only run 16 processes at a time.

Note 3 An unfortunate consequence is that that even processes that only use a little bit of memory still count towards our 16 processes. This consideration leads us to the topic of utilization.

2.2 Utilization

Definition Utilization is defined as the fraction of a resource allocated for useful work. Usefulness is perspective and is considered in the following discussion.

2.2.1 From CPU's Perspective

- A while(1) loop counts as "useful" for a CPU.
- When a process finishes its tasks, it enters into an infinite loop, where the processor will still be working.

2.2.2 From Memory's Perspective

Example A very small program uses ≤ 4 KB code, ≤ 4 KB data, ≤ 4 KB stack. A quick survey gives the following results: Total bytes used per process ≤ 12 KB $= 12 * 2^{10} = 3 * 2^{12}$ Bytes, Total processes $= 16 = 2^4$, Total available memory $= 12 * 2^{20}$.

Utilization The Fraction of utilized memory is $\frac{3*2^{12}*2^4}{2^{20}} \rightarrow \frac{3}{16}$. This amounts to only 3/16th of our process's allocated memory.

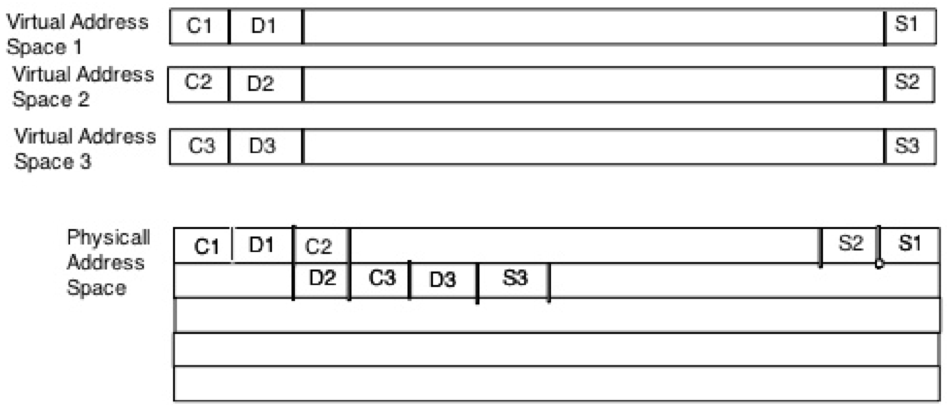
Problem Since processes can't overlap in our example, we see that the memory resources are under-utilized.

2.3 Virtual Memory

Problem A species of under-utilization where free space cannot be used to satisfy space requests.

Solution Virtual memory

We no longer need a contiguous region of physical address space to allocate for a process.

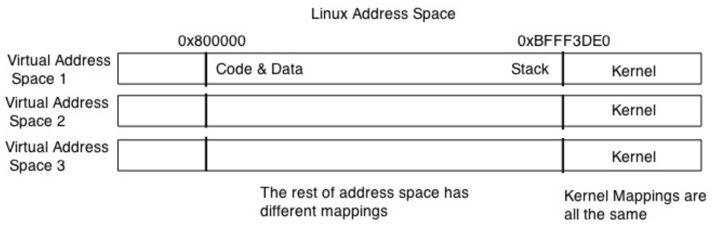


a process's virtual memory still appears to have memory layed out with code, then data, stack at the top, and then the middle being the heap and room for the stack to grow down, but we can store all of this *anywhere* in physical memory

Now we can fit $\frac{2^{20}}{3 \cdot 2^{12}} \approx 85$ processes in our 2^{20} byte process space! So our maximum memory utilization is $\frac{85*3*2^{12}}{2^{20}} \approx 1$

What about the heap? The Heap is virtual address space that is mostly unused

The processor does not allocate physical memory for the heap until the the process asks for memory, at which point it tries to satisy the request



What about Linux?

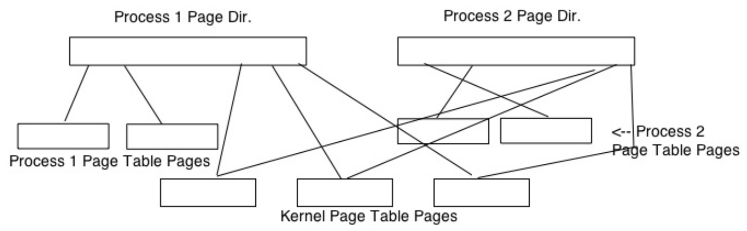
(another diagram)—stack starts at 0xBFFF... , kernel is above that, etc.

The mappings for the kernel is roughly the same for every process

The rest of the address space has different mappings because the physical memory is mostly disjoint

physical memory is mostly disjoint among processes (because of process isolation)

What if VM page dirs were represented as one big array? • We'd have to allocate all of the pagedir memory at once (or else run out of contiguous space to hold the whole thing), so the pagedir for each process would take up $\frac{\text{size of addresses}}{\text{pagesize}} \cdot \text{size of pageentry\_t} = \frac{2^{32}}{2^{12}} = 2^{22}$ bytes. For each process!

- With our 2-level tree, we don't have to store all of the pagedir contiguously, and only allocate as much memory as we need to store that process's addresses
- Could need at most 2 pages to represent the $\leq$ 8 KB of the code and data
  - This is because, if the code starts at, e.g. address 0x7FF000—offset is 0, page table index is 0x3ff, pagedir index is 0x1
  - Then 8KB (0x2000) plus 0x7FF000=0x800FFF—offset is 0xFFF, page table index is 0, pagedir index is 0x2
    - ∗ since these two addresses have two different pagedir indexes, we need two page tables (one at index 1 in the pagedir, and one at index 2 in the pagedir)
- To represent the 4KB of the stack we also could need at most 2 pages (if we run into a similar mapping problem where the stack memory crosses two pagedir indexes)
- 

- ALSO: since each process shares the same memory for the kernel portion of memory, they can all point to the same page tables for the kernel's portion of memory
- Our 2-tree implementation is also O(1) (because the tree is of fixed depth)
- Note that all the addresses *stored* in the pagedir are all physical
- Why does each process have to have a portion for the kernel?
  - So it can do system calls!