CS 61 Scribe Notes - October 23, 2012

Lewin Xue, Marcus Schorow, and Emmet Jao

CS 61 Scribe Notes October 23, 2012

ANNOUNCEMENTS

- Midterm handed back
- Solutions are posted on Piazza
- Sample problems:
  - 5B:
    - Upon examination, we see that it modifies rows separately
      - eliminates answer (f), as that needs the information from other rows
    - It appears that each hex constant has the same 2 bytes
      - Examine what this function does on a single byte, consisting of bits abcdefgh
    - First line: (abcdefgh >> 1) & 0x55 becomes (0abcdefg & 01010101) becomes 0a0c0e0g. (abcdefgh << 1) % 0xAA becomes (bcdefgh0 & 10101010) becomes b0d0f0h0. 0a0c0e0g | b0d0f0h0 = badcfehg.
      - We note this is essentially swapping adjacent bits (ab, cd, ef, gh → ba,dc, fe, hg)
    - Second line: (badcfehg >> 2) & 0x33 becomes 00badcfe & 00110011 = 00ba00fe. (badcfehg >> 2) & 0x33 becomes dc00hg00. 00ba00fe | dc00hg00 yields dcbahgfe.
      - We note this is essentially swapping chunks of two bits.
    - Third line: (same method as before) hgfedcba.
    - 4th line switches the two bytes. So in essence, this entire function reverses the image horizontally.
    - Image D.
  - f7 (from bonus problems)
    - cute[i]& -cute[i] yields least significant bit turned on. (bitwise arithmetic)
    - Yields image G, as each row only has one bit turned on

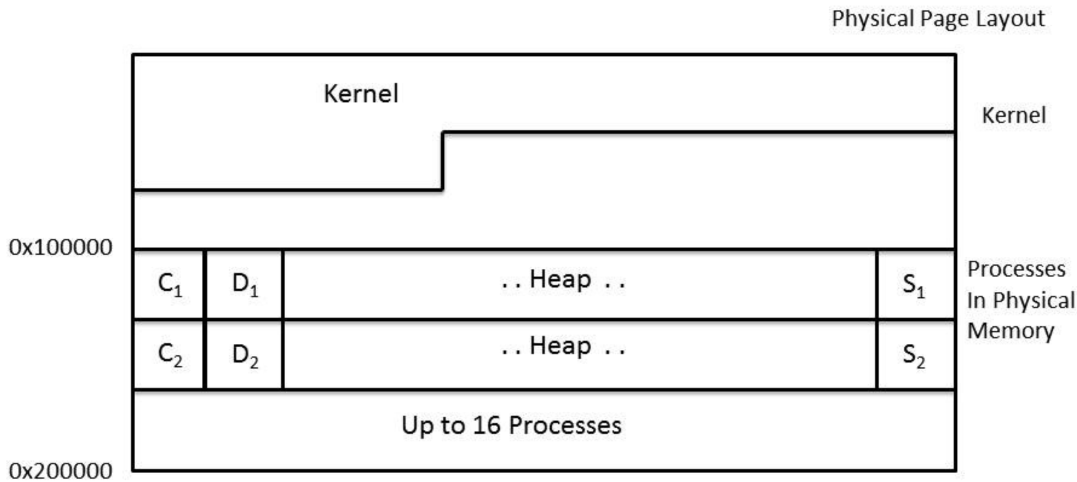
GOALS FOR TODAY:

- x86 Virtual memory
  - page directories
  - 2-level page tables
- VM as a tool for memory management
- OS: Every process has its own memory, corresponds to different places in physical memory.

WHAT'S WRONG WITH PHYSICAL MEMORY ALLOCATION?

- Suppose 2^20 B (1 MB) of physical memory available for processes excluding kernel.

- Each process will use 2^16 B of address space including code & data, stack, and heap.
- Can run at most 16 processes at once. 2^20/ 2^16 = 2^4 = 16 as processes not allowed to overlap.
- Wasteful, because a process such as while(1); requires very few addresses for code, data, stack, and heap.

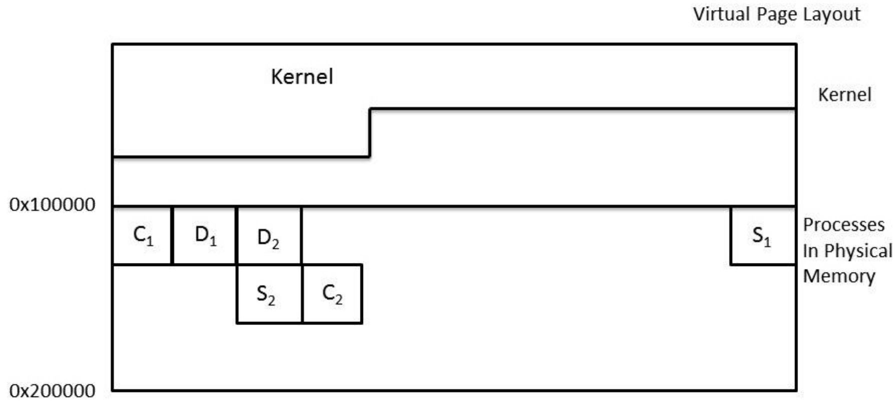


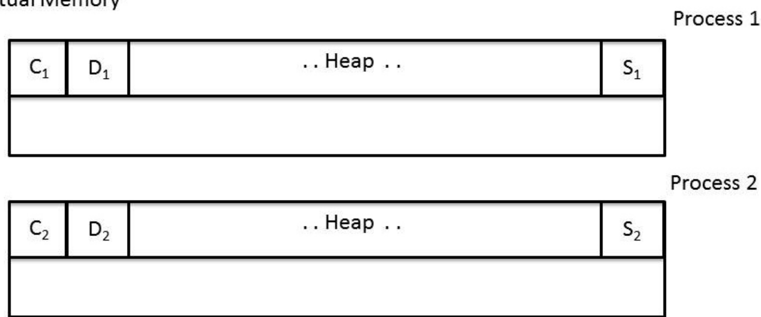
- Key term: **Utilization**: Fraction of a resource that is allocated for useful work. What is considered "useful" may be subjective:
  - From Kernel perspective: useful work defined as when a process is running, and non-useful work when the kernel itself is running in an idle loop.
  - Suppose 16 copies of while(1) running on this machine. CPU utilization is HIGH since max # of processes are running. On a dual core (2 CPU) machine, running one process of while(1) will completely utilize one of the cores. Running 2 of these processes will result in complete CPU utilization. But we might not consider this "useful."
- Most processes give up the CPU usage when they run out of useful work to do. Called **blocking.**
  - But, running this while(1); process will use up less than 12 * 2^10 bytes of memory (<4KB code, < 4 KB data, < 4 KB stack). Overall, these 16 processes will have memory utilization of less than 12 * 2^10 * 16 / 2^20 = 3 / 2^4 = 3/16, which is quite low.
- We are assuming that every process is given a code page, data page and a stack page. Unused memory will be in the heap areas of the processes
- Key Term: **Fragmentation**: Utilization problem where free space cannot be used to satisfy space requests. (In this case, 13/16 of memory is free but no process slots are available)

VIRTUAL MEMORY ALLOCATION

- Virtual memory eliminates this address space fragmentation problem
- Rework this case using virtual memory allocation

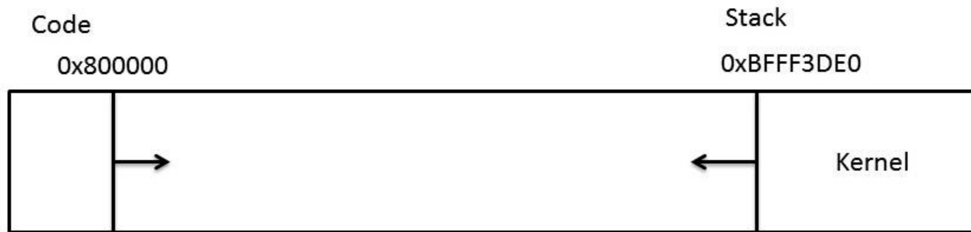
- We create a map between physical pages and virtual pages. Different processes no longer need contiguous regions of physical memory, but they "see" the same virtual address space even though the pages in physical address space can now be jumbled up as necessary to save space





- x86 pages are 2^{12} B (4 kB) each
  - Again suppose that each instance of the process gets 3 pages: code, data, stack
  - How many processes can we run simultaneously now?
    - $2^{20} / (3*2^{12}) \sim 85$.
  - Much better memory utilization, close to 100%!

Linux VM addresses



- Code/Data at 0x00800000
- Stack starts around 0xBFFF3DE0
- Kernel is the upper ¼ of memory above stack
- Mapping of kernel is generally the same
- Only one kernel in physical memory; physical memory for processes generally disjoint (process isolation)
- Most processes don't use that much code and data or stack space

What if virtual memory page directories were represented as arrays (pagedir[])?

- Would require large contiguous allocation of memory to hold the page directory. The 2-level tree system does not require contiguous blocks of memory.
- Also, size of the array must be sizeof(address space) * sizeof(pageentry_t) / PAGESIZE = 2^{32} * 4 / 2^{12} = 2^{22} B = 4 MB. This is expensive!

Each entry in a page directory page covers 2^{22} B = 4 MB of address space.

- The process has 8 kB of code and data. Might need two pages to cover these
- Recall that in a Linux VM address, the lowest 12 bits are the offset, the next 10 bits are the page table index, and the highest 10 bits are the page directory index
- Suppose virtual address of code/data is 0x007FF000
  - offset = 0, page table index = 0x3FF, page directory index = 1
- 8 kB higher is virtual address 0x00801000
  - offset = 0, page table index = 0x1, and page directory index = 2
  - So we potentially need 2 page table pages to handle code/data
- Also need 1 page directory page.
- If the 4 KB of stack memory needed just overlaps a page boundary, need 2 page table pages to handle stack (worst case).
  - Unlike the case where the page directory is an array, we don't need to represent the heap
    - process does not have memory stored in the heap!
- Kernel:
  - same mappings apply for each process
  - so the page tables that represent the kernel mappings can be shared amongst the processes' page directories.
- Obtaining a representation of a virtual address in physical memory is still O(1), so no disadvantage compared to an array page directory.

- Note that all of the addresses in a page directory are physical addresses!