Lecture 20. Operating Systems

- **An *operating system* provides a *virtual machine*: A high-level abstraction of an ugly low-level machine**

- **An OS provides *resources* and *services***

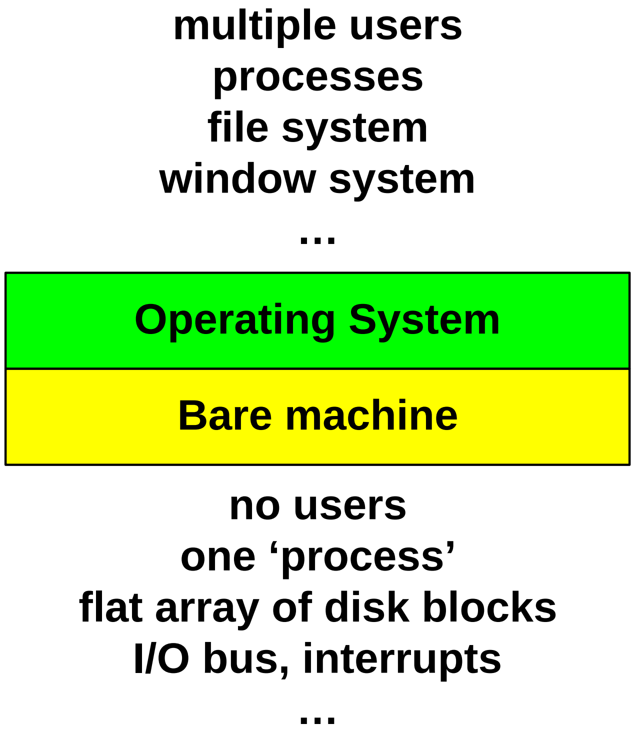
  Memory management: Each user appears to have all the memory

  Concurrency: Many users appear to compute simultaneously

  Protection: User *A* can’t crash *B*’s program or access *B*’s files

  File system: Files appear as streams of bytes, files have names, directories, random access

  Interaction: X window system, window manager, mouse



multiple users
processes
file system
window system
…

Operating System

Bare machine

no users
one ‘process’
flat array of disk blocks
I/O bus, interrupts
…

  Network access: The World Wide Web, remote file systems and printers

- **Programs communicate with the OS via *system calls*, e.g. TOY opcode 4**

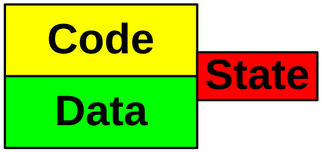
  $\underline{4}402_{16}$ **prints the contents of** R_4

  Each OS has its own (usually large) system call vocabulary

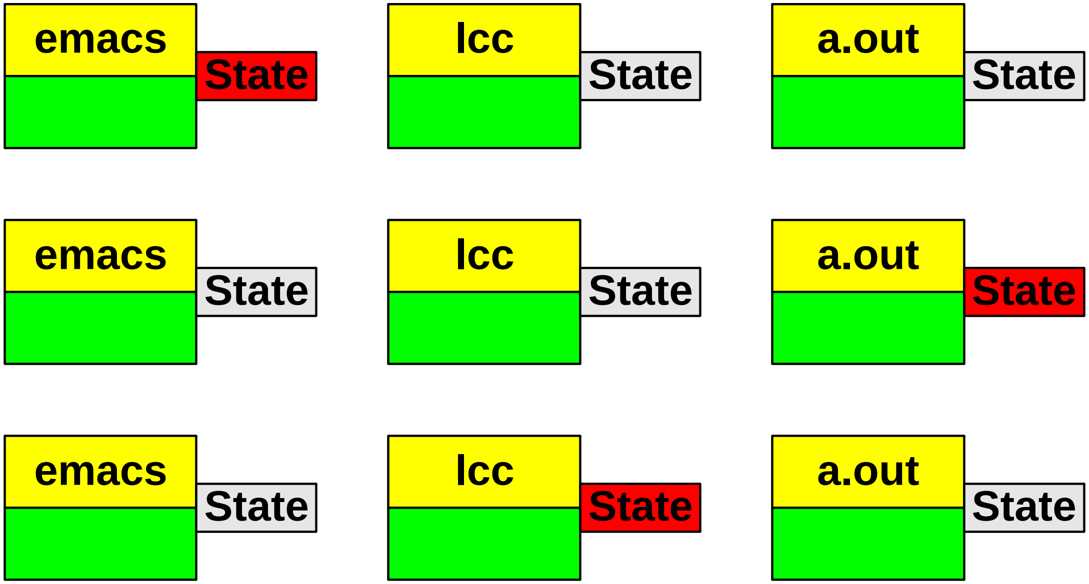
Multiprogramming

- **A *process* is an executing *instance* of a program**

  ***State* includes registers, PC, memory management information**



- **The OS, a.k.a. kernel, *multiplexes* the processor between the processes, switching between processes at each *interrupt***



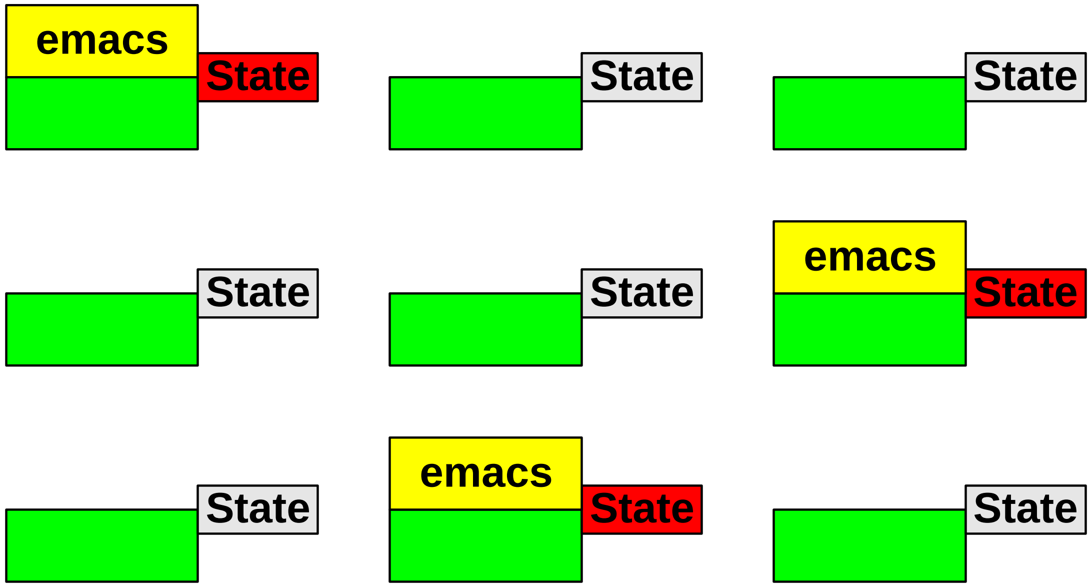
- **When a periodic *clock interrupt* occurs (≈ every 1/60 second), do a *context switch***

  Stop
  Store the registers, PC, etc. in the current process’s state
  Load the registers, PC, etc. from the new process’s state
  Continue (‘dismiss the interrupt’)

Reentrant Programs

- **A *reentrant program* does not modify its own code; it changes only its data**
- **One copy can be shared among many processes; each process has it own data**

Three processes running `emacs`



Reentrant programs use less memory

- **What about the *addresses* in each process?**

Virtual Memory

- **Problem 1**

  Several programs need to use the same memory

  Direct solution: Divide up the memory

- **Problem 2**

  If the OS can load program anywhere in memory, what is its starting address?

  Direct solutions: Have OS adjust relocatable addresses upon loading
  Use only *position-independent code* (impossible in TOY)

- **Problem 3**

  One program needs more memory than the machine has, or more than is left

  Direct solution: 'Overlay' unused functions with other functions

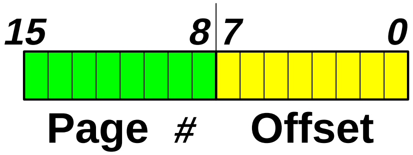
- **'Better' solution to all these problems**

  Each program assumes access to the entire memory — its *virtual address space*

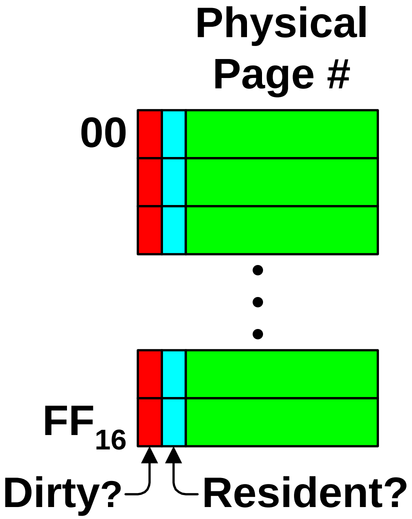
  Hardware helps OS associate a small part of *physical address space* with each process, keep some of the virtual address space on disk

Paging

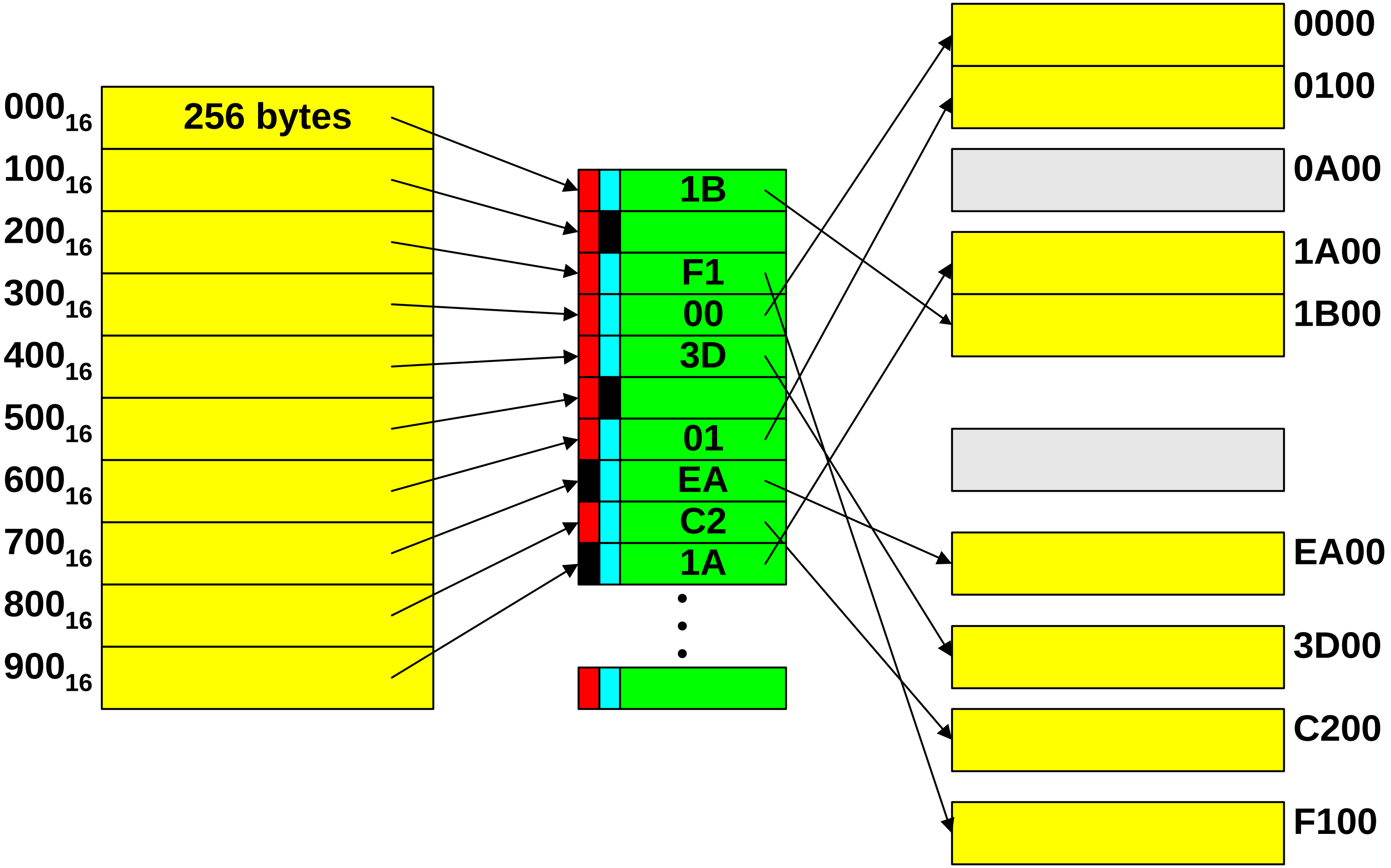
- ***Paging*** **is the predominant method for implementing virtual memory**
- **Maximum *effective address* determines the virtual address space size**
- **Divide physical memory and virtual memory into fixed-size ‘pages’**
    - **Use a power of 2**
        - **Leading address bits give the page number**
        - **Trailing address bits give the offset in that page**
    - **Example: 16-bit addresses, 8-bit page #s, 256-byte pages**



- **Build hardware to map all addresses through a *page table***
    - **Indexed by virtual page #**
    - **Maps virtual page # → physical page #**
    - **Indicates whether page is in memory or on disk**
    - **Indicates whether in memory page is ‘dirty’ or clean**
- **Keep virtual memory for each program on disk**



Paging, cont’d



- **Each page read in from disk has to replace another page: Use page replacement strategies, such as *L*east *R*ecently *U*sed**

Size of Virtual Memory

- 16 bits is not enough
- 24 bits is not enough
- 32 bits is not enough!
- Is 64 bits enough?

  18,446,744,073,709,551,616 $>$ 10^{19} addresses

- 64-bit address space needs more sophisticated paging strategy and hardware

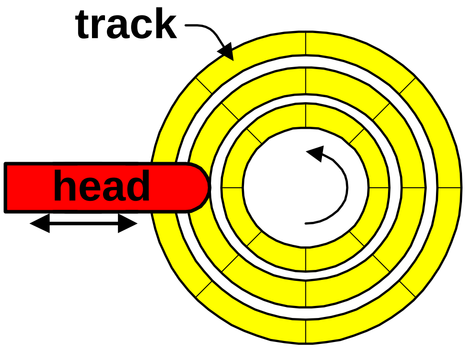
  Page table would be too big: 2^{13} $=$ 8Kbyte pages needs 2^{51} page-table entries

  Associative page tables, multilevel page tables

- Some big numbers

10^{20}	Number of grains of sand on a beach
10^{27}	Number of oxygen atoms in a thimble
$2^{256} > 10^{77}$	Number of electrons in the universe

File Systems



- **Disks are messy: Rotating cylinders with movable heads**

  Rotational latency: Wait for the ‘track’ to appear under the head

  Seek time: Wait for the head to move in/out to the cylinder

  At best, a disk is an array of fixed-size blocks

- **A *file system* provides high-level features on low-level disks**

  Directories

  Named files

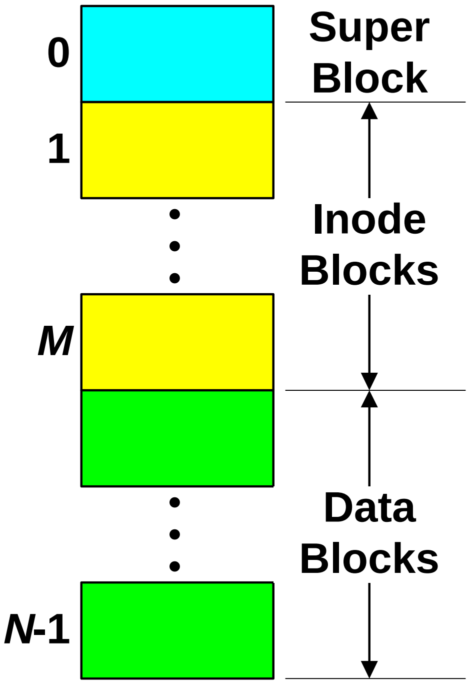
  Read/write arbitrary number of bytes

  Random access

  Automatic growth

UNIX File System

- **Disk, array of fixed size blocks, is divided into 3 regions**
    - **Root block: File system parameters *M*, *N*, list of free data blocks**
    - **'Inode' blocks: Hold 'information' nodes, one per file or directory**
    - **Data blocks: Hold the data, file names in directories**
- **Inode blocks each hold *k* inodes numbered 0 to *k*–1, so a file system can hold *k*×*M* files/directories**
- **An inode holds everything about a file, *<u>except</u>* its name**
    - **Type: directory or file**
    - **Size in bytes**
    - **Block numbers of its data blocks or indirect blocks**
    - **Number of directories pointing to the file**
    - **Times of creation, last modification**
- **A *<u>directory</u>* is just list of (file name, inode number) pairs**



File Layout

- **Small file: Inode points to 10 data blocks**

  For 1Kbyte data blocks, handles files ≤ 10 Kbyte

- **Medium-size file: Inode points to 10 'indirect' blocks that point to data blocks**

  With 4-byte block #s, handles files ≤ $10 \times 256 \times 1024 = 2{,}621{,}440 = 2.5$ Mbyte

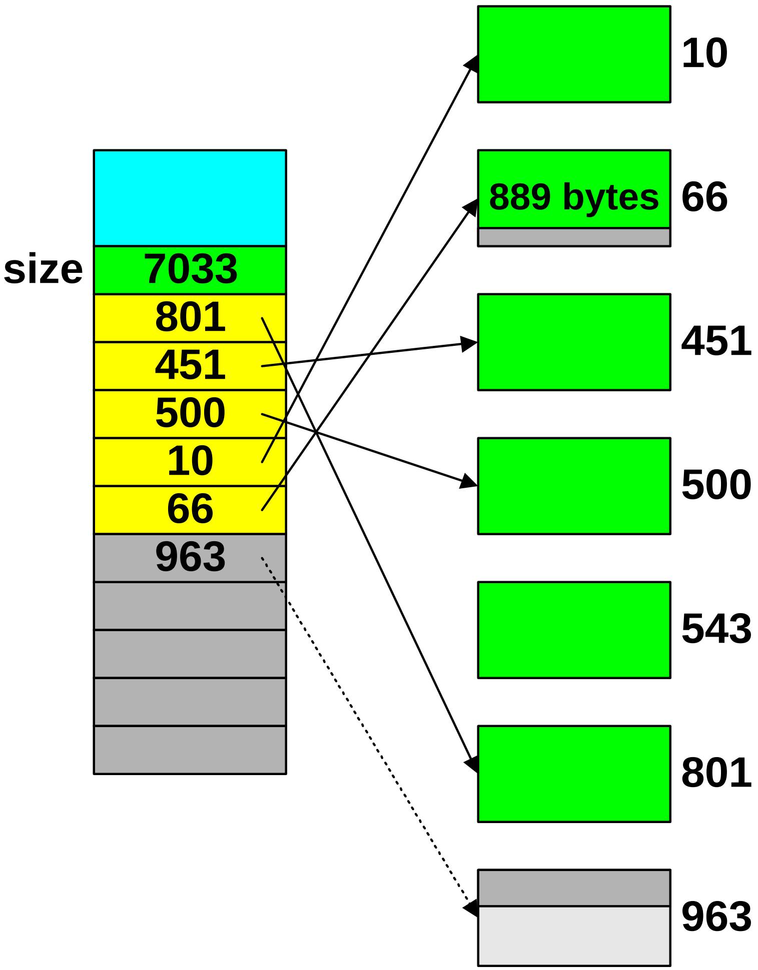
- **Large files: Entries in last indirect block point to other indirect blocks**

  Handles files ≤ $(9 + 256) \times 256 \times 1024 = 69{,}468{,}160 = 66.25$ Mbyte

- **Huge files: Inode points to 10 indirect blocks that each point to 256 indirect blocks**

  Handles files ≤ $10 \times 256 \times 256 \times 1024 = 671{,}088{,}640 = 640$ Mbyte

- **Adjust block size/inode size to span larger disks**



Typical Medium-Size File