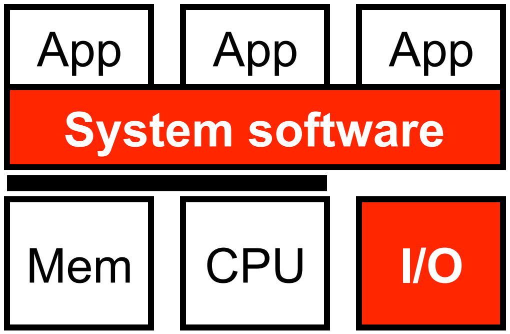
ECE 590.03
Fundamentals of Computer Systems and Engineering

Operating Systems

Operating Systems



- File Systems
  - Reading:

http://www.cs.berkeley.edu/~brewer/cs262/FFS.pdf

- Scheduling
  - Processes: where do they come from?
- Bootstrapping
  - How does the system start?

Previously…

- Have been talking about IO-related topics
  - Interrupts
  - Hard drives
  - Memory-mapped IO
- Now: into the OS
  - First up: how do we store files/directories on the disk?
  - Disk: stores blocks of data
  - **Filesystem**: imposes structure on that data
    - Directories contain files
    - Files have data
    - …and meta-data: access time, ownership, permissions,…

Filesystems (ext2,ext3,ext4)

- Filesystem made of **blocks**
  - Fixed size allocations of space (e.g., 4KB)
  - Can hold file data or filesystem information
- Blocks organized into block groups
- Block Group locations in table after **superblock**
  - Array specifying where block groups start
- Superblock: describes key info about file system
  - One per file system
  - But replicated (avoid single point of failure)
  - At fixed locations

Block Groups

- Block Group Descriptor Table
  - One or more blocks (super block says how many)
  - Follows superblock
  - Array telling where each block group starts
- Block groups
  - Many blocks with good spatial locality (e.g., same cylinder)
  - Use one block to track free data blocks
  - Another block to trace free **inode** blocks
  - Main point: spatial locality—try to allocate blocks within same group

Inodes

- Inodes contain information about a file
  - Owner
  - Permissions
  - Access time
  - **Where data blocks are located**
  - Number of blocks used
  - …

  - All meta-data about a file except its name

- Fixed size: 256 bytes

Inodes: Where to find data

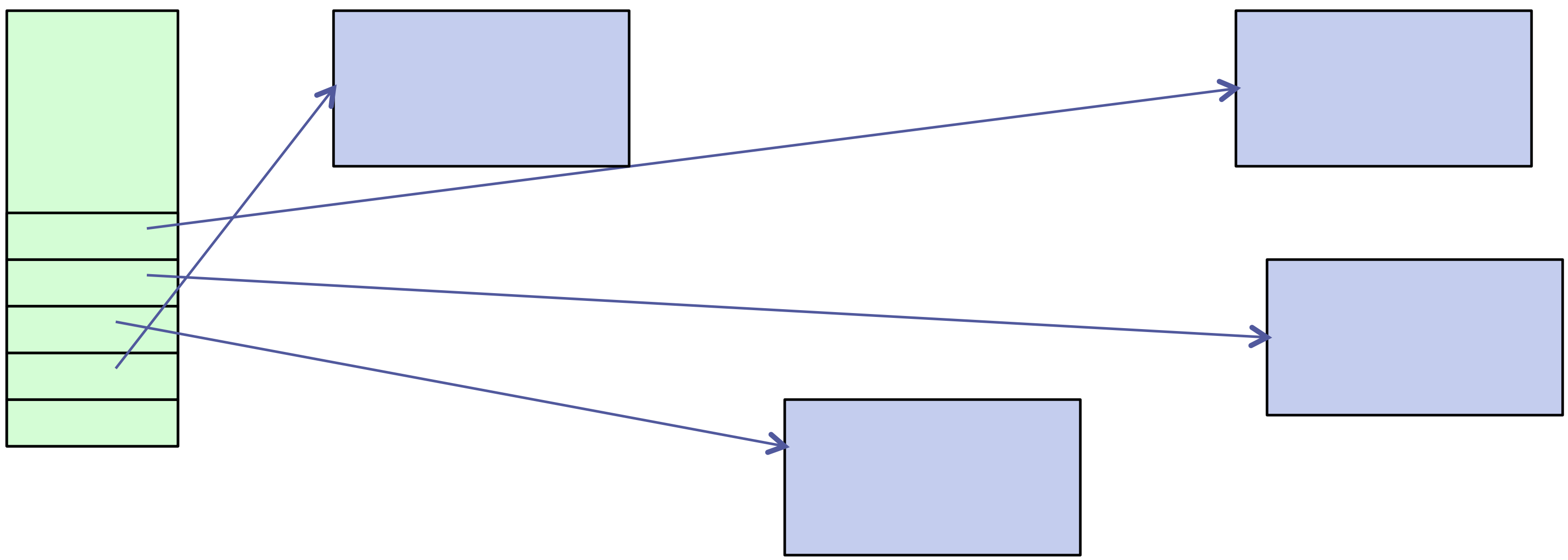
- Inodes specify where the data blocks reside.. But how?
  - Pointers (e.g., block numbers) to the data

- Solution 1: Direct pointers in inodes
  - Pros?
  - Cons?

Inodes: Where to find data

- Inodes specify where the data blocks reside.. But how?
  - Pointers (e.g., block numbers) to the data

- Solution 1: Direct pointers in inodes
  - Pros: Fast (read inode, read data)
  - Cons: Small limit on file size (~16 pointers * 4KB = 64KB max?)



Inodes: Where to find data

- Inodes specify where the data blocks reside.. But how?
  - Pointers (e.g., block numbers) to the data

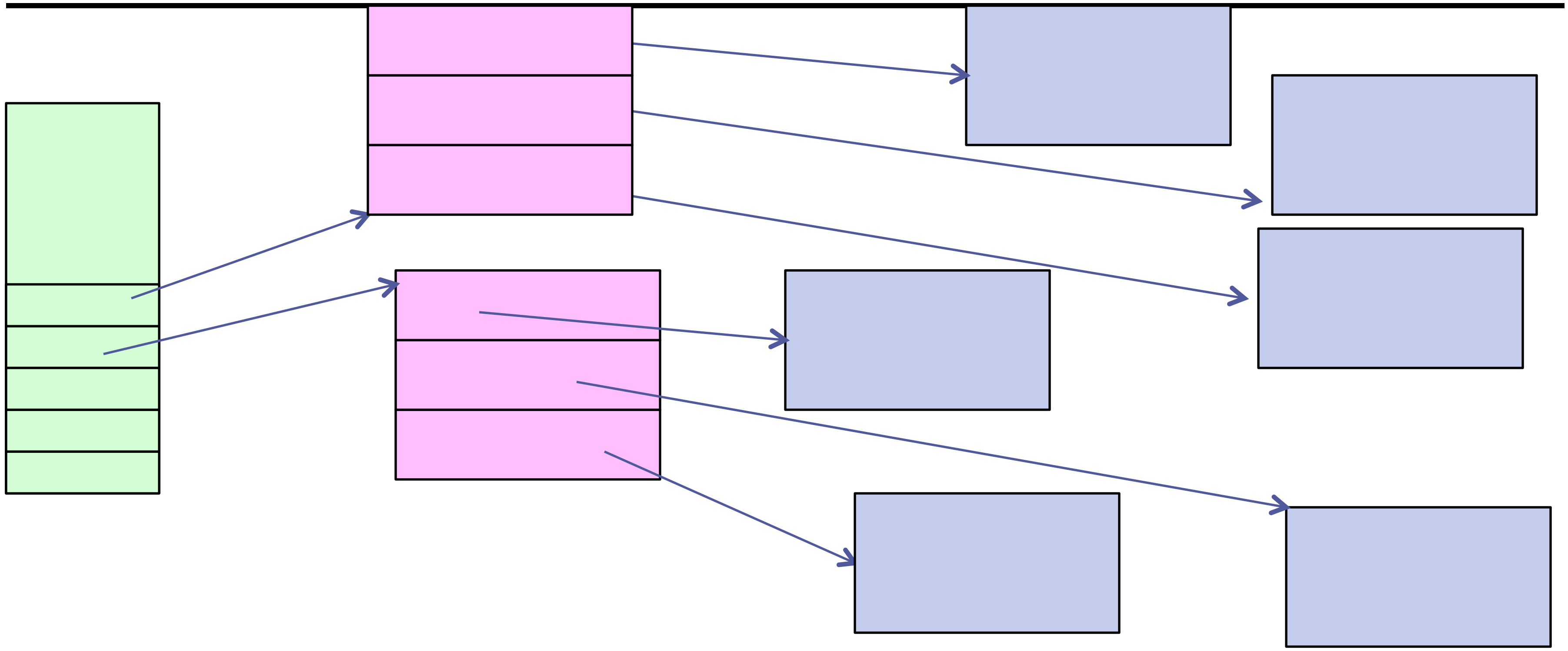
- Solution 1: Direct pointers in inodes
  - Pros: Fast (read inode, read data)
  - Cons: Small limit on file size (~16 pointers * 4KB = 64KB max?)

- "I can't store large files" = functionality problem
  - Solution?

Inodes: Where to find data

- Inodes specify where the data blocks reside.. But how?
  - Pointers (e.g., block numbers) to the data
- Solution 1: Direct pointers in inodes
  - Pros: Fast (read inode, read data)
  - Cons: Small limit on file size (~16 pointers * 4KB = 64KB max?)
- "I can't store large files" = functionality problem
  - Solution? Level of indirection
  - Inode has pointers to blocks containing pointers to data

Solution 2: Indirection



- Max size?
  - 16 pointers, each to a 4KB block
  - 1K pointers per block, each to a 4KB block of data
  - 16 * 1K * 4KB = 64MB
  - Ok… better, but we still need bigger

Advice from the xkcd stick-figure guy

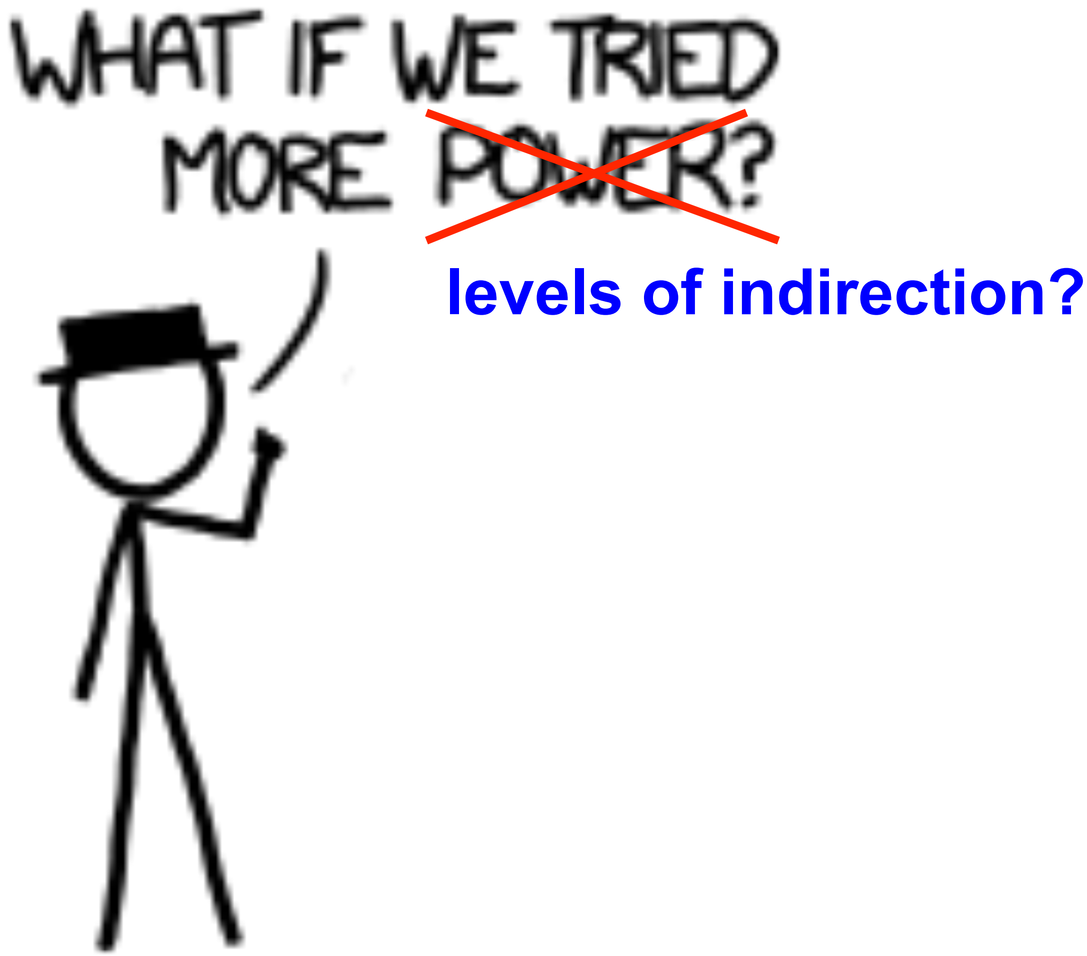


Image from xkcd.com

More indirection

- 2 levels of indirection:
  - ~16 ptrs in inode * 1K 1st level * 1K second lvl * 4KB = ~64 GB
  - Better, but we still might need more?

- 3 levels of indirection?
  - 64 TB: probably big enough….
  - But kind of slow? Now need 5 disk reads to get the data?
    - (Inode, 1st lvl, 2nd lvl, 3rd lvl, Data)
  - Might be willing to pay this price if using a 100+G file… but what about a tiny little file?

Real inodes: a mix of approaches

- Real inodes mix approaches for best of both worlds
  - 12 direct pointers (first 48KB of data)
  - 1 indirect pointer (next 4MB of data)
  - 1 doubly indirect pointer (next 4GB of data)
  - 1 triply indirect pointer (next 4TB of data)

- Example of "make the common case fast"
  - Small files = fast
  - Only need slow technique for really large files
    - Rare
    - Can cache indirect block tables when accessing

Stepping back a level

- Inodes: meta-info on files
  - Including how to find its data
  - Not including names (we'll see why soon…)

- How do we find files?
  - We organize them into directories
  - **`cd /home/drew/ece590.03/lectures`**
  - How do we store directories?
    - They are just files too!

UNIX: file types

- UNIX has multiple file types
  - All have inodes, type is in the inode

  - **Regular files**: what you think of for files (contain data)
  - **Directories**: contain a list of (name, inode #) pairs
  - **FIFOs**: aka named pipes
    - Allow two processes to communicate via a queue
  - **Symlinks**: a symbolic link to another file
    - Contains the path to the other file
    - But accessing it takes you to the other file
  - **Devices (char/block)**: interface to hardware devices
  - **Sockets**: inter-process communication
    - Similar to FIFOs, but different

Directories

- Directories contain (name, inode #) pairs
  - Iterate through them looking for name you want
  - Find inode #
  - Want a sub-directory? Works same as other files
  - Two special names: . and ..
    - . = current directory (name maps back to own inode #)
    - .. = parent directory (maps back to parent inode #)
  - Only special in that they are created automatically and can't be deleted

- Some types of filesystems support more scalable directory lookup

Filesystem misc

- Hard Links (not to be confused with symlinks)
  - Two names, same inode number
  - Why inodes don't have the name: may be multiple names

  - Delete one: other one still exists
  - Inode tracks how many links to it (hard links, not sym links)

  - Delete last reference: inode and data blocks released

- Other
  - We have talked about ext2, other file systems exist
  - Many modern file systems have **journaling** for crash protection
    - Log what you are about to write, then write it

Filesystem vs swap space

- Filesystem for files
  - But disk also used for virtual memory ("swap space")
- Different **partitions** of the disk used for each
  - May also have multiple file systems on multiple partitions
  - File systems are **mounted** at some path, then look identical to normal directories to user
- **Swap space**: managed differently
  - Temporary (no need to remember layout across reboot)
  - Fixed-size: always operate on a page at a time
  - Kernel can just track what is free/what is in use, where each page is

Filesystem summary

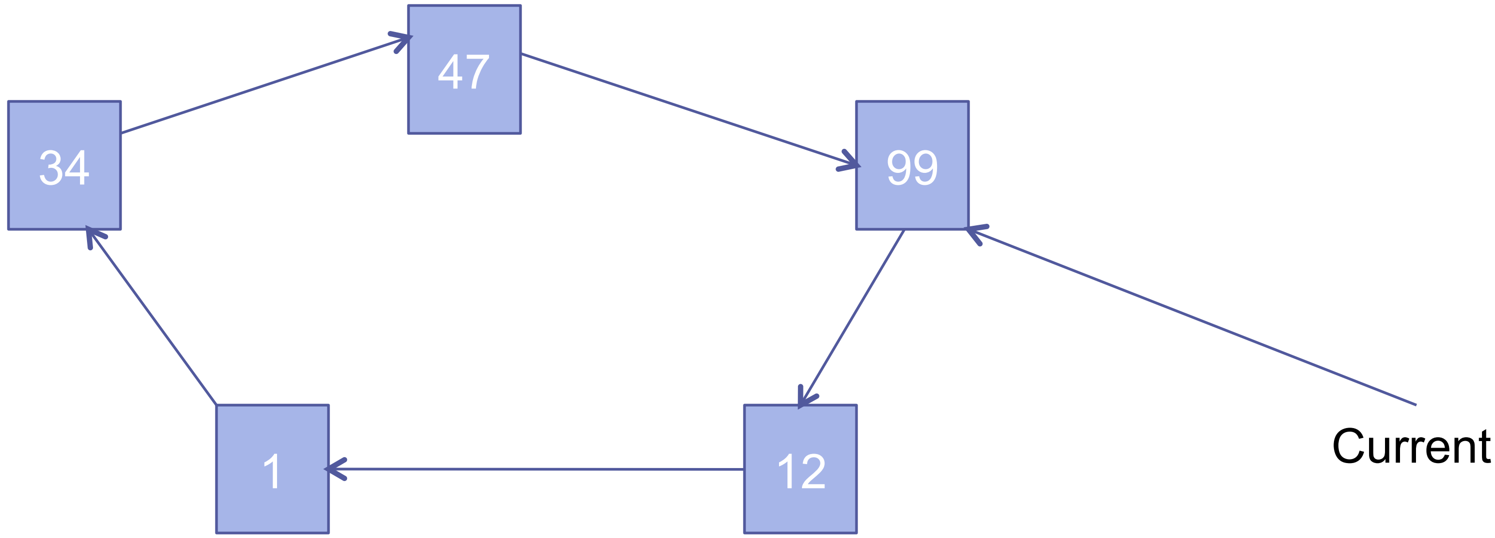
- Organize data on disk
- Inodes track meta-data: including data location
- Directories contain (name, inode #) pairs
  - Iterate to find what you want
- Different types of files, but mostly work the same
- Superblock contains meta-data about whole filesystem
- Blocks grouped for spatial locality

Processes

- A **process** is a running instance of a **program**
  - Program: xterm
  - May run 4 copies of it at once, each a different process

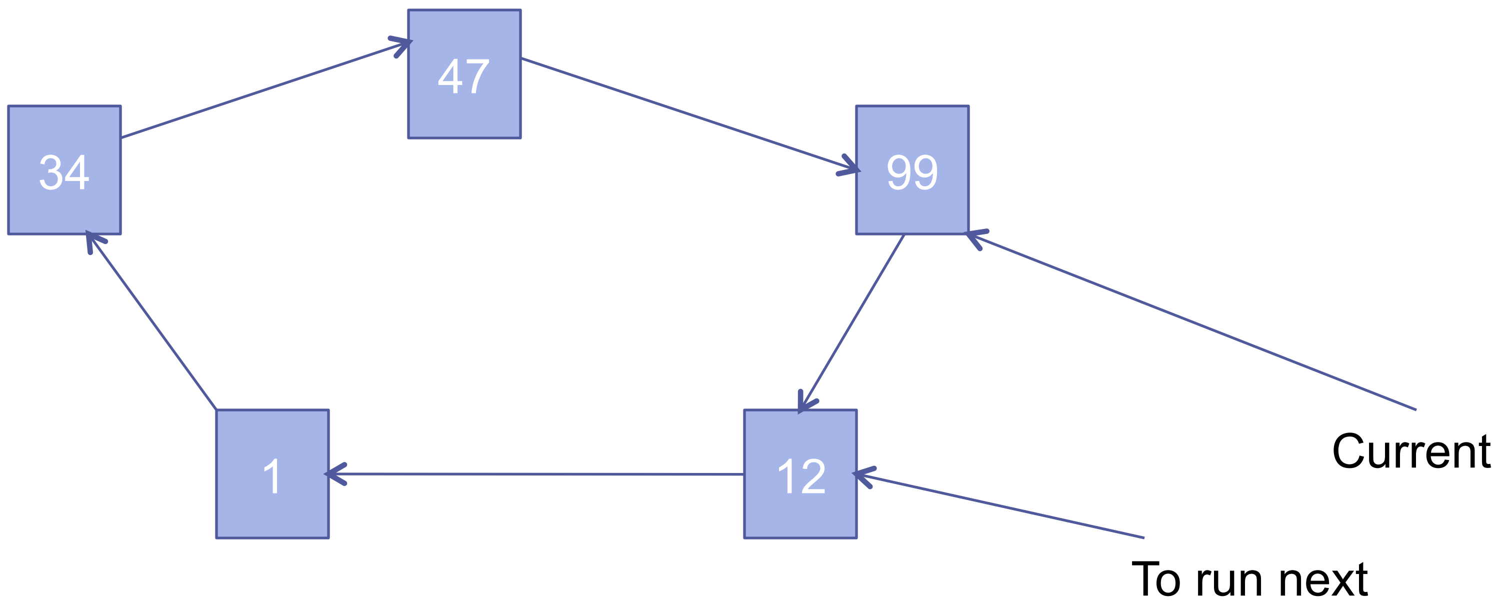
- Processes have a process id (pid):
  - A number which uniquely (at the time) identifies the process
  - System calls which act on other processes identify them by pid
    - Example: kill (send a signal to a process, identified by pid)

Process scheduling



- OS maintains scheduler queue
  - Basic: circular queue, round robin
  - Fancier: priority based scheduling, fancy algorithms, etc…
- Remembers which process is currently running

Process scheduling

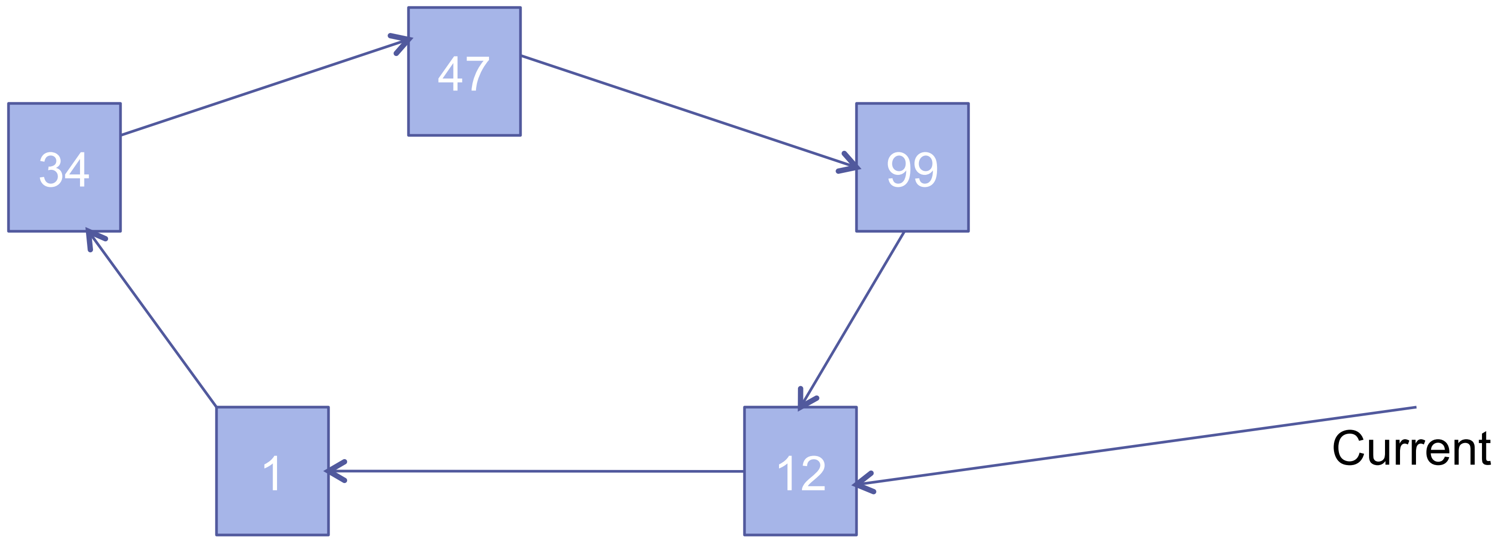


- Timer interrupt drives scheduling
  - Interrupt happens: scheduler figures out what to run next
  - E.g., current->next
  - Some processes may not be runable right now
    - E.g., waiting for disk

Context switching

- To change currently running program, OS does **context switch**
  - Save all registers into OS’s per-process data structure
    - Elements of scheduler list are large structs
  - Change processor’s page table root to point at PT of new process
  - Load registers for new process
  - Return from interrupt
    - Leave privileged mode
    - Jump back to saved PC

Process scheduling



- Now new process runs until interrupt or exception
  - Note: OS only entered by interrupts/exceptions (including syscalls)

- If no process runable, kernel has “idle task”
  - Tells processor to go to sleep until next interrupt

Process creation

- Processes come from duplicating existing processes
  - **fork()**: make an exact copy of this process, and let it run
  - Forms parent/child relationship between new/old process
    - Can tell the difference by return value of fork()
      - Returns 0: child
      - Returns >0: parent (return value = child's pid)
  - No guarantees which scheduler returns to first
    - Or both at same time, if multi-core

Only copies?

- If we just duplicate existing programs, how to run anything else?

- fork() can be followed by **exec**()
  - Exec takes filename for program binary from disk
  - Loads that program into the current process's memory
    - Destroying anything currently in it
    - Resetting stack and heap pointers
  - Set PC to be the starting PC of the program (stored in the binary)
  - …and never returns (except on error)—why?

- Note: fork does not have to be followed by exec()
  - May actually want multiple copies of same program

Fork-then-exec…wasteful?

- Fork: make duplicate copy of process
- Exec: overwrite with newly loaded program

- Seems wasteful to make a copy of everything
  - Then throw it away?

- Imagine: Big complicated application (2GB memory)
  - Wants to run external command (often)
    - fork(): copy 2GB memory
    - exec(): discard copy to load new program

Copy-on-write: page table magic

- Virtual memory hackery to the rescues
  - Instead of copying all of memory, just copy page tables
    - Two programs now have PTs pointing at the same physical pages
  - Now, mark each page read-only
    - Writes will cause page-faults
  - Kernel remembers it did this, and copies the page on a write
    - Then marks it writeable, and resumes the process

  - Exec? Only copy page tables!
  - No exec? Copy page tables up front, then copy pages as written

- Special-purpose alternative: **vfork()**

Multiple threads

- A process may also have multiple **threads**
  - Execute concurrently, but share virtual address space
  - Low-level system call: clone()
  - Library call: pthread_create()

  - Different registers
  - Different stack (different $sp)

  - Correct programming with threads requires **synchronization**
    - Locks
    - Barriers

Parent/child relationship

- Children can return an exit status to their parents
  - Generally indicates success or failure
  - Argument of exit() or return value of main()

- How do parents get this return value?
  - Child becomes **zombie** process: still exists in OS's list of processes, but does not run
  - At some point, parent calls waitpid() (or wait()) to wait for a child to terminate.
  - Waitpid() gives the return value to the parent (and "reaps" the process, finally destroying its table entry)

- What if the parent exits before the child?
  - Child gets "adopted" by system process called init, which reaps it

So if processes come from copying…

- If processes come from **fork()**ing, how do we get the first process?
- For that matter, how do page tables get setup?
- And… how does the system start in general?

Booting the system

- Booting is architecture specific: we'll talk about x86_64
  - Processor initializes in 16-bit real mode
    - Virtual memory is off (real mode = use real addresses)
      - Real address is another word for physical address
    - Execute BIOS (low-level firmware) startup code
      - Splash screen/startup/press DEL to enter setup
    - BIOS reads Master Boot Record (sector 0) of hard disk
      - Loads contents into memory and jumps into it
      - This code is tiny (440 bytes)
      - First stage of **bootloader**
    - This (tiny) code loads more data (code) from disk
      - Then loads stage 2 bootloader
      - Asks BIOS to do disk IO for it

Booting continued

- Stage 2 bootloader
  - May present menu, ask for options, etc
  - Then loads kernel—requires reading filesystem
  - Then jumps to kernel entry point

- Now the actual OS kernel is in control
  - Still in 16-bit real mode
  - Sets up page tables
  - Sets up interrupt vector
  - Sets up a few other x86-specific things
  - Enters "protected mode" (switches to 64 bit with virtual memory)
  - Creates idle task and spawns init (pid 1, from /bin/init)

Init

- Init: First "normal" program
  - OS loads /bin/init as pid 1
  - Init reads configuration file (in /etc)
  - Spawns other programs (e.g., /bin/login, sshd etc)
    - Done with normal fork()/exec()
  - Periodically reaps orphaned processes

Much more to it

- Could spend whole semester on Oses
  - Barely scratched surface with an overview
  - If this were an OS class, we would
    - Write kernel modules
    - Modify the linux source
    - Make our own filesystem (?)
    - Fiddle with the scheduler
    - Go into much more detail on all these topics
    - Cover a bunch of other topics