Chapter 10: Case Studies

So what happens in a *real* operating system?

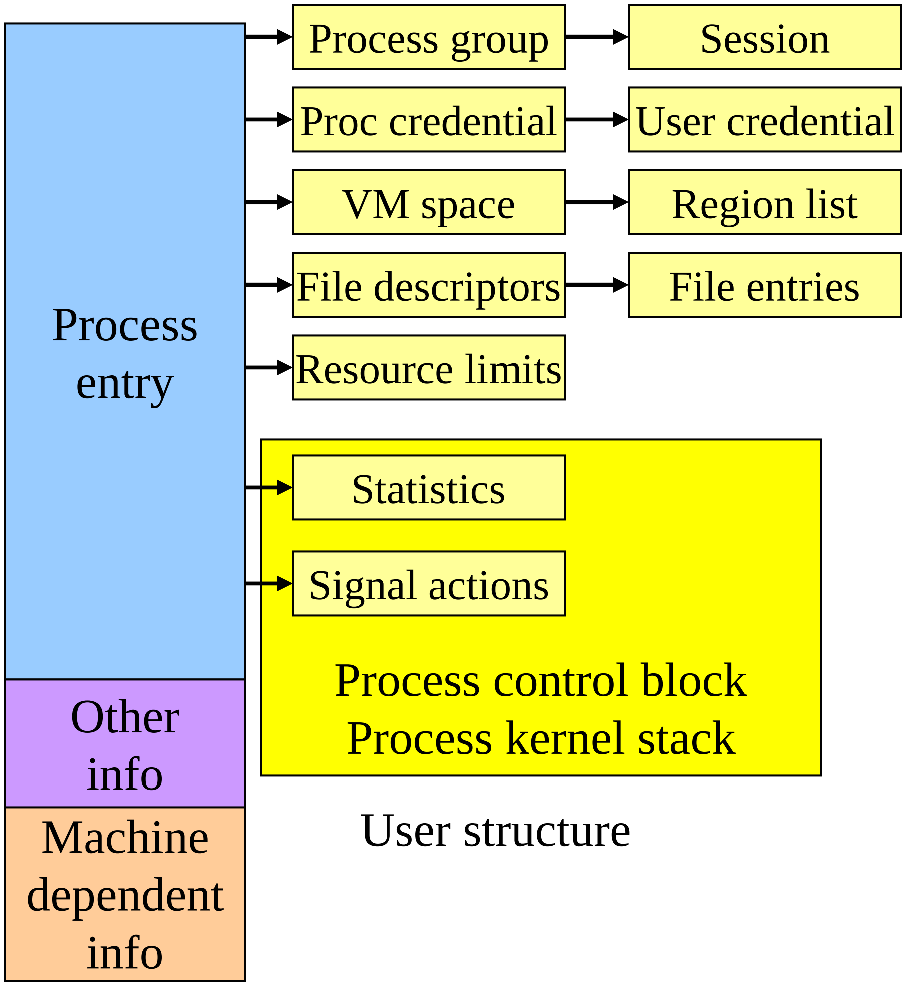
Operating systems in the real world

- Studied mechanisms used by operating systems
  - Processes & scheduling
  - Memory management
  - File systems
  - Security
- How are these done in real operating systems?
- Examples from:
  - Linux
  - BSD
  - Windows NT

But first, a history of Unix and its relatives

- Started in the late 1960's with MULTICS
- Ken Thompson at Bell Labs developed UNICS on a discarded PDP-7
  - Name changed to UNIX
- Important variants:
  - AT&T version 7
  - BSD (Berkeley Software Distribution)
  - Linux (not strictly a Unix derivative!)

Process structure in BSD



- Contents of process control block include
  - Process identifier
  - Scheduling info
  - Process state
    - Wait channel
  - Signal state
  - Tracing info
  - Machine state
  - Timers
- Other stuff is pointed to by process entry
  - Process group implements hierarchy of processes

Process scheduling in BSD

- Uses multilevel feedback queues
  - Processes placed in queues according to priority
  - Priorities adjusted dynamically
- Processes in highest priority queue run round-robin
  - Processes in lower-priority queues may not be run, but…
  - Dynamic priority quickly moves such processes into a higher queue!
- Quantum is always 0.1 second
  - Short enough for good response time
  - Long enough to dramatically reduce context switch overhead

Calculating process priority in BSD

- Two values in process structure
  - Estimated CPU utilization: p_estcpu
  - “Nice” value (user-settable): p_nice
    - Between -20 and 20
    - Lower is better (and below 0 requires root)
- Priority calculated every 40ms as
  - Priority = PUSER+(p_estcpu/4)+2*p_nice
  - Result moved into range PUSER–127
- P_estcpu incremented each time the clock ticks while the process is running
- P_estcpu decays over time: recalculated each minute
  - P_estcpu = ((2**load*)/(2**load*+1))*p_estcpu+p_nice
  - *Load* is a function of the number of runnable processes
- Penalizes CPU-intensive processes, but intensive CPU use is eventually forgotten

Scheduling in Linux

- Fully preemptive
  - Scheduler called whenever any process switches from blocked to runnable
  - Higher priority processes preempt lower priority ones
- Scheduling done by *epochs*
  - Each process gets a fixed fraction of the time in an epoch
  - Time remaining is decremented when the process runs
  - Variable-length scheduling quantum!
- Fields used by the scheduler are:
  - Priority: base priority of the process
  - Counter: number of ticks of CPU time remaining in this epoch for this process

Calculating priority in Linux

- Scheduler picks the next process by
  - Finding the highest value of *counter+priority*
  - 1 point bonus for sharing memory space with current process (better use of cache & TLB)
- Epoch ends when all runnable processes exhaust their quantum (*counter* = 0)
  - For each process, new *counter* = (*counter* >> 1) + *priority*
  - If process was blocked, *counter* > 0, increasing priority
  - Note: *counter* can never become greater than 2**priority* because it's a geometric series
- Linux also supports other scheduling algorithms
  - Real-time
  - True FIFO scheduling (non-preemptive)

So how well does this scheduling work?

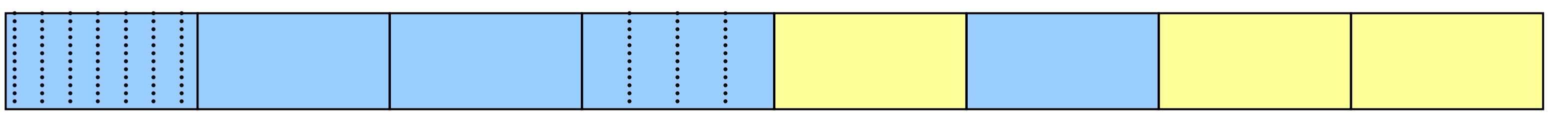
- BSD: fixed-length quantum, vary priorities frequently
  - Bump up priorities of processes that haven't been using the CPU, penalize processes that use the CPU often
  - Run highest priority processes => long-running processes can run if there's nothing better to do
- Linux: variable-length quantum, reschedule after every process has had its turn
  - Epoch length varies by number of processes
  - Priority can only change after each epoch
  - Limits to CPU time in each epoch
- Research at UCSC: real-time scheduler that still handles "regular" processes well

Memory allocation in BSD & Linux

- Problem: kernel memory allocation can cause internal fragmentation
  - Space wasted due to inefficiently handling small objects
  - Memory difficult to reclaim: can’t just kill the process!
- Solution: build efficient memory allocators
  - Use “powers of 2” to allocate variably-sized objects
  - Allow allocation of small as well as large objects
- BSD has a relatively simple system
- Linux has a more complex system (powers of 2 and “slab” allocation”)

Memory allocation in BSD

- Allocation "chunk" constrained to 2^k bytes if less than a page
  - Keep a free list for each chunk size
  - Keep a list of chunk size for each page to quickly free chunks
  - Difficult to reclaim a page that has been subdivided into chunks
- Allocation in whole pages if greater than a page
  - Use first fit to find consecutive free pages



kmemsize[]={ 512, 8192, cont, 1024, free, 4096, free, free}

Buddy system for memory allocation in Linux

- Uses powers of two to allocate regions
- *Buddy system* used to coalesce regions into larger regions
  - Keep a bitmap for regions of 1, 2, 4, …, 512 pages
    - Each bit tracks two *buddies*: 2^k page regions that start on a 2^{k+1}-aligned address
    - 0 => both buddies are free or both are allocated
    - 1 => exactly one buddy is allocated
  - On allocation
    - Check to see if there's a region of the desired size free
    - If not, split the next larger region
    - Continue this way until the desired region is free
    - If no space, return an error
    - Update bitmap aaccordingly
  - When a page is freed, check to see if its buddy is free
    - If so, mark the larger region as free
    - Recursively move up the list in this way
- Also uses *slab* allocation for lots of fixed-size objects

Slab allocation in Linux

- Buddy system is good, but not for small (less than one page) objects
- For frequently-used small objects, use *slab allocation*
  - Keep a free list of objects of a particular type (size)
  - Allocate new pages when needed, dividing them into objects of the appropriate size
  - Keep track of slabs: areas of contiguous memory that have been subdivided
    - This allows them to be freed when no objects in them are in use
  - When dividing up pages, shift objects slightly to avoid CPU caching issues
    - Vary the free space at the start and end of the slab
- Infrequently-used objects handled by "generic" slab with objects ranging from 32 bytes – 128 KB by powers of 2

Real-world file systems

- File systems have two layers
  - Virtual file system layer: does directory management, caching, file locking, bookkeeping, etc.
  - Physical file system layer: does data layout and disk free space management
- Lots of physical file systems in BSD & Linux
  - FFS (Berkeley Fast File System)
  - LFS (log-structured file system)
  - Ext2 (Linux standard file system)
  - Ext3 (ext2 with journaling)

VFS layer

- VFS does the things that *all* file systems need to do
- Directory management
  - Directories == files in Linux & BSD, so VFS translates directory operations into file reads & writes
  - Allows the lower-level file system to take over some or all of this functionality: permits more efficient directories in systems such as XFS
- Metadata management
  - Returns information about a given file
  - Metadata kept in a consistent format (underlying physical file system must convert into this format)
- Caching…

Caching in Linux

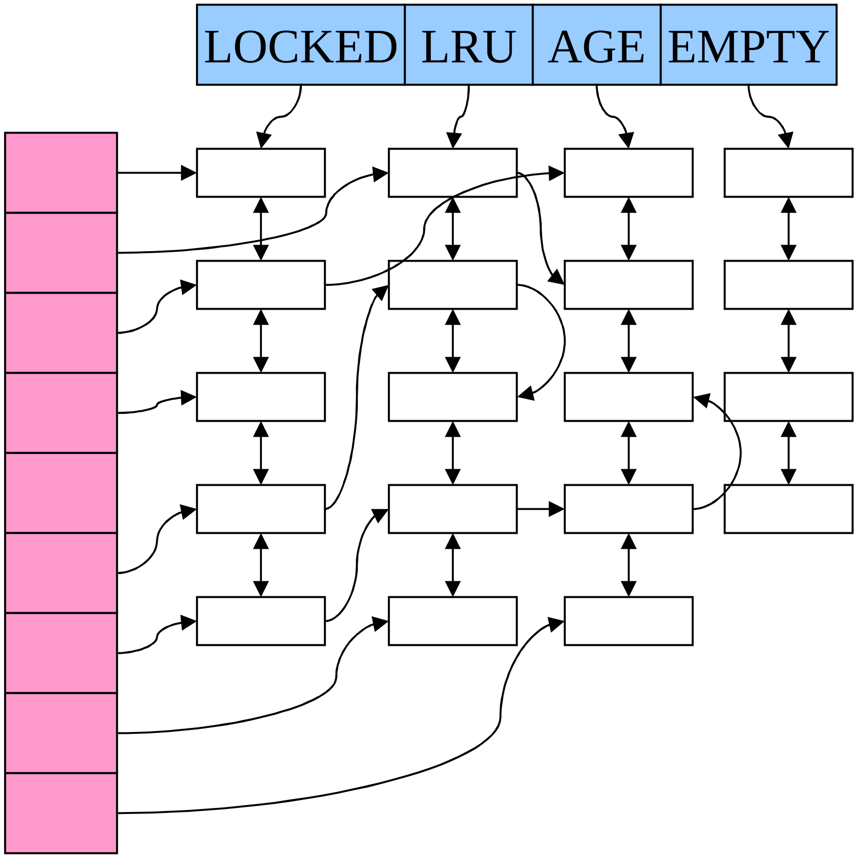
- Linux uses a *buffer cache* to store frequently-used disk data
- Cache consists of
  - Buffer heads: one per buffer, describes the buffer and its contents
  - Hash table: quickly find the buffer head for a given block
  - Buffers themselves: just pages from memory
- Buffer heads contain
  - Block number, size, ID
  - Status information
  - Pointers to buffer, other buffer heads in lists & hash table
- File buffers reclaimed in same way as pages from VM
  - Kernel process goes through memory in a clock-like way
  - If pages haven’t been used recently, they’re freed up

Writing data back to disk

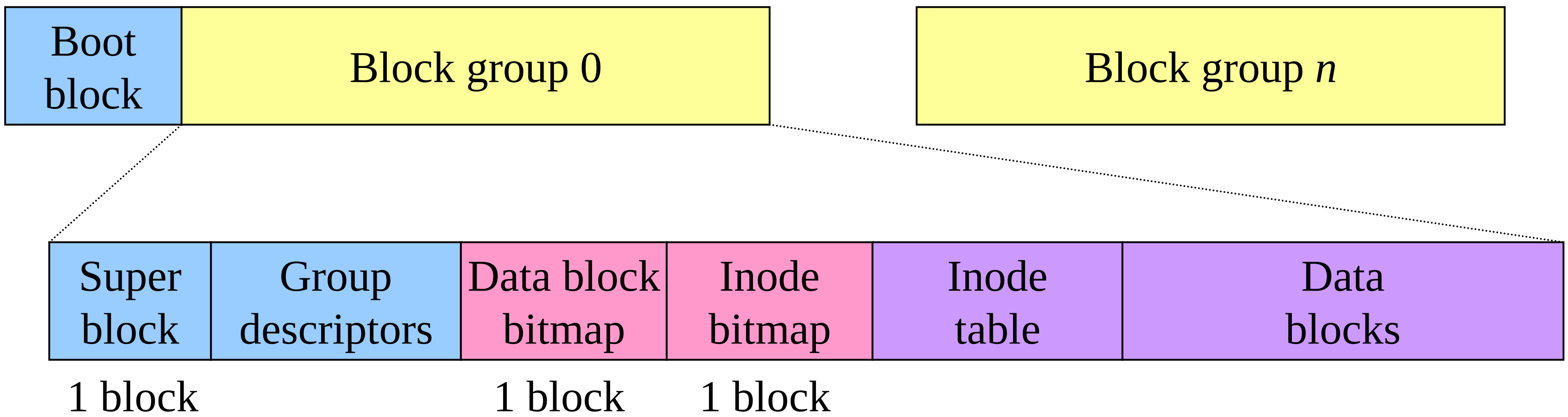
- File writes go to buffers, then to disk
  - Delay in writing depends on the type of block
    - Regular buffers: defaults to 30 seconds
    - Superblocks (contain info about the file system): defaults to 5 sec
  - Buffers flushed every 5 seconds (by default)
  - Buffers may be flushed more frequently if too many are dirty
- Entire cache may be written to disk at once
  - Usually done with a sync() system call
  - All buffers for a file can be written with fsync() call
- Caches for metadata are handled separately

Caching in BSD

- Same kinds of structures as in Linux
  - Buffer heads
  - Hash tables
    - Look up buffer by logical block number and file ID
  - Buffers themselves
- Kernel keeps several lists
  - Locked
  - LRU
  - AGE
    - Prefetched buffers
    - Data not likely to be reused
  - Empty (free buffers)
- Buffers moved off AGE when they're referenced
- Buffers reclaimed first from AGE, then from LRU



Ext2 file system: data layout



- Disk divided into *block groups*
  - Each block group has inodes, data blocks
  - File system tries to keep data from a file in a single block group
- Bitmaps showing which blocks & inodes are free
  - Limited in size to 1 block => max of 8*BLOCKSIZE data blocks (or inodes) in any one block group
- Super block and group descriptors are backups in case of file system corruption

Ext2: directory layout

- Each entry is a variable length
  - File names up to 255 characters long
  - Records padded to a multiple of 4 bytes
- File type indicates whether it’s a directory, file, symbolic link, device, etc.
- Record length & file name are kind of redundant…

21	Inode number (4 bytes)
12	Record length (2 bytes)
4	File name length (1 byte)
1	File type (1 byte)
a	
b	
c	
d	
\0	
\0	
\0	
\0	

Ext3 vs. ext2

- Ext3 is very similar to ext2
  - Ext2 can be converted to ext3 without reformatting!
  - Ext3 can be read by ext2 file system!
- Big difference: journal
  - Ext2 was unreliable if a crash occurred
  - Inconsistency because an operation didn’t complete
  - Ext3 uses a *journal* to prevent this
- Journal: write (to a file / region of the disk) the operation you’re about to perform *before* actually doing it
  - Journal is relatively small, and circular
  - On recovery from a crash, read the journal to see what operations were recently written to the journal
  - Check to see if those operations actually completed
  - Perform the operations that hadn’t completed

BSD: Fast File System (FFS)

- Very similar to ext2 (FFS came first, though!)
  - Disk divided into *cylinder groups* (similar to block groups)
  - Inodes have similar structure
  - Bitmap for tracking free blocks in a cylinder group
  - Multiple copies of superblock, descriptors
- FFS has *fragments*
  - 2^k fragments per block
  - Allow files to efficiently use fractions of a block
  - Fragments can only be used as the last block of a file
  - Tracking fragments adds complexity
  - Using fragments dramatically reduces internal fragmentation
- Tries to keep a file within a cylinder group
  - Large files spread across multiple cylinder groups
  - Goal: big chunks of files kept together