Advanced Unix System Administration

Lecture 4
February 25, 2008

Steven Luo
<sluo+decal@OCF.Berkeley.EDU>

Memory Management

- Memory use from user space
  - Each process sees its own private virtual address space
  - Code is mapped into memory from disk
  - A few pages are mapped for local storage as the stack – this grows as needed
  - Process can explicitly request memory from the heap using malloc() – though this can be lazy!
  - Files can be mapped into memory using mmap()

Memory Management

- Efficient VM operation
  - Kernel must keep track of many things about pages:
    - Which bits of disk and RAM correspond to an address
    - Whether the disk and RAM are in sync (dirty bit)
    - Purpose of the allocation (data, code, mmap file, cache)
  - Ideally, the stuff that's in use and/or used most often should stay in RAM even when memory pressure strikes

Memory Management

- Efficient VM operation con't
  - Without prescience, figuring out what's going to be used next is a difficult art
  - Getting it wrong is a very large performance penalty
  - Lots of different algorithms for doing this: FIFO, random, NRU, LRU, NFU, aging; performance varies by application
  - All of the generally useful ones need to keep track of when pages are used

Memory Management

- Fragmentation
  - Kernel's keeping track of lots and lots of stuff per page, so the fewer pages the better
  - Keeping large allocations together means less work for the kernel and faster allocations
  - Some applications (i.e. DBs) actually need contiguous blocks of physical memory
  - Various strategies for keeping memory allocations together

Memory Management

- Large pages
  - Bigger pages means fewer pages, of course
  - Advantage: less overhead for large allocations, ensure contiguous physical memory
  - Disadvantages: difficult to allocate in presence of fragmentation and/or memory pressure, reduces flexibility
  - Not fully supported by all OSes

Input and Output

- Files
  - The file is (in principle) the fundamental abstraction behind Unix I/O
    - “Everything is a file” – the famous Unix mantra that's maybe true
  - As far as user-space programs are concerned, a “file” should be a stream of data which can be read from and written to
    - Could be a file on disk, a network socket, a device, etc.
    - Whether the file is opened via a filesystem is another story

Input and Output

- Synchronous I/O
  - At simplest: process makes syscall to I/O facility, kernel does I/O, returns
  - This is what read(), write(), and friends do
  - Because we treat network sockets and various other things as files, they can be handled in a similar way
  - This model has some inefficiencies – context switches, copies, and blocked processes

Input and Output

- Asynchronous I/O
  - Allows the process to do something else while I/O is running
  - Different ways of doing this: don't bother notifying the process, polling, event loop, signals/callbacks
- Memory-mapped I/O
  - Processes and kernel arrange to read/write from memory in orderly fashion
  - Fundamentally async

Input and Output

- I/O scheduling
  - When multiple requests to a particular I/O source come, we should try to arrange them efficiently
    - Simple first in, first out model works fine for networks – not so well for rotational disk media
    - On rotational disks, try to arrange requests so that reads and writes are near each other on the platter
    - When multiple devices are concerned, take into account which device data is on
  - If we're going to schedule, we might as well do priority scheduling too ...

Input and Output

- Filesystems
  - At the core, a FS is just a way of collecting files efficiently
  - Construction: usually laid out as blocks of various types
  - Directories contain pointers to other directories and inodes
  - inodes store filenames, metadata (permissions, ACLs, timestamps), and pointers to the actual data blocks

Input and Output

- POSIX filesystems
  - Unix filesystems traditionally make various guarantees – i.e. creating links will be atomic
  - This means that applications make assumptions about the way they operate on files (example: the standard way of safely replacing a file – especially a binary – while in use)
  - NFS breaks quite a few of these assumptions – hence random tricks and workarounds