Advanced Unix System Administration

Lecture 3
September 22, 2008

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

Processes

- Threads
  - Recall that the kernel keeps lots of state for each process
  - But if the processes are related, we might be able to get away with less of that
  - Threads = “lightweight processes”
  - Shared resources means programming is more difficult
  - Kernel support for threads reduces overhead and makes implementation easier

Memory Management

- Usually, an n-bit processor can address n bits of memory
- Especially on 64-bit systems, this tends to be much more memory than actually exists on the system
- Besides, the physical address of a particular byte may not be a particularly convenient way to work with it

Memory Management

- Solution: paged memory, virtual memory
  - Divide up physical memory into pages (usually 4K or 8K) and keep track of pages of memory
  - Keep a page table of pages and the memory addresses used to access them
  - Creates more flexibility: per-process virtual address space, non-contiguous allocations, shared memory, etc.

Memory Management

- Virtual memory
  - As long as the information stored at the address can be retrieved somehow, there's nothing wrong
  - We can map pieces of disk storage (whether swap or memory-mapped files) to an address
  - This is slow, so we can also (and where possible, usually do) keep a copy in physical memory
  - When demand spikes, we can quickly drop pages backed by non-volatile storage

Memory Management

- Caching VMs
  - RAM is much faster than disk, so keeping info in RAM will speed up many tasks
  - Some kernels (i.e. Linux) will cache file accesses in “free” memory
  - Again, pages can be dropped quickly if memory pressure arises – though this may not always be profitable
  - Mantra: (truly) free memory is wasted memory

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