22 File Caching

Locality of reference in file accesses allows performance improvements through the use of caching. This is yet another application of the principle of locality. Keep a number of disk blocks in fast memory; when accessing disk, check the cache first.

Where are disk blocks cached?

- special purpose volatile memory in disk controller (PCs)
- fixed portion of main memory (BSD)
- variable portion of main memory (modern Unix); processes and filesystem compete for physical memory

Disk block caches are maintained in software.

Problem: fast cache memory is volatile, but users expect disk files to be persistent. In the event of a system crash (e.g., power failure) dirty disk blocks in the cache are lost.

- Solution 1: use write-through cache
  - modifications are written to disk immediately
  - no performance advantage for disk writes
- Solution 2: limit potential data loss (Unix)
  - write-through caching for metadata (i-nodes, directory, freelist blocks)
  - write back dirty data blocks after no more than 30 seconds
  - write back all dirty blocks during file close
- Solution 3: keep a log in fast non-volatile memory (mainframe OS, databases)
  - recovery program is run after a crash to reconstruct lost dirty blocks from the log

How is the disk cache implemented? Two approaches:
1) Set of kernel buffers maintained by the filesystem (buffer cache)

- with a read/write API, can implement precise LRU replacement
- not used for memory-mapped files
- need to decide how to partition physical memory among buffer cache and VM cache uses.
  - static partitioning during kernel configuration (BSD)
  - dynamic adjustment of partitioning during runtime; for example, keep track of VM page fault and disk cache miss frequencies, and try to balance.

2) Memory-map all open files

- caching comes for free (just like normal VM); no separate file cache
- flexible sharing of physical memory
- VM page replacement policy does not discriminate between cached file data, and other VM pages.
- read/write API can be supported by mapping open files into kernel address space, and copying from/to user buffers.

Other performance-improving techniques:

- read-ahead: for sequential access, start reading next file block from disk when application program reads the previous block.
- write-behind: start disk write, but don't make application wait until the disk operation completes.
- allows overlap of a processes' computation with its own disk I/O.