Middleware and Distributed Systems

Design of Scalable Servers

Martin v. Löwis

Problem Statement

- Allow multiple clients to connect to a server simultaneously, and handle connections from them in an overlapping manner
  - On a single processor, parallel processing is not possible, anyway
  - On a MIMD system, parallel processing should allow to process as many requests simultaneously as there are central processors
- Strategies?

Multi-Threading

- Idea: Start a new thread for each client connection (thread-per-client)
  - seemingly-simple programming model (given a good thread library)
  - problem 1: race conditions, synchronization errors, ...
  - problem 2: if connections are short-lived, cost of thread creation is high
- Idea: Thread-pooling
  - solves problem 2
  - how to implement?
  - problem 3: number of concurrent threads in a system often limited (e.g. Windows NT+ ca. 2200)

Multi-Processing

- Child per Request/Client
  - fairly safe against race conditions, synchronization issues
  - e.g. CERN httpd
  - parent process performs accept, then fork; child process closes the connection
  - problem: high overhead for process creation
  - problem: OS load for scheduling increases
  - problem: memory consumption
- Process pool
  - e.g. Apache prefork: StartServers, MinSpareServers, MaxSpareServers, MaxClients, MaxRequestsPerChild, ...
  - how to implement?

Single-Process-Single-Threaded Servers

- Idea: on a single-processor machine, there can be one activity at a time only, anyway.
    - On a multi-processor machine, start roughly as many processes as there are processors (e.g. twice as many)
- historical approach: select
    - problem: fd_set limited to maximum number of file descriptors
        - some system extend size of fd_set, e.g. on a per-call basis
        - problem: copying the fd_set takes time linear with the largest file descriptor (winsock: fd_set is not a bitmap)
    - problem: kernel needs to scan bitmap for relevant file descriptors, the arm them; application needs to scan for ready file descriptors

poll

- System V improvement to select(2)
- allows for arbitrary many file descriptors
- on return, only read file descriptors are returned
- on call, kernel needs processing time only linear with number of polled-for file descriptors (often not linear with maximum file handle)
  - problem: kernel still needs to arm selected all file descriptors
- Win32 version: WaitForMultipleObjects
  - problem: returns only a single ready file descriptor
  - problem: does not work for sockets (can use WSAEventSelect)
  - problem: supports only 64 handles (work-around: use threads)

kqueue

- Implemented in FreeBSD since 2000
- tries to avoid passing the same long lists of inactive file descriptors again and again
- new kernel object: kqueue (created through kqueue(2))
  - operation kevent (add/remove new file handles, return list of ready handles)
  - returns amount of ready data for read-ready events (similar for write-ready events)
  - supports both edge-triggered and level-triggered events
  - more general than file descriptors, e.g. process creation, exec, exit
- Linux version: epoll (Solaris: /dev/poll, ...)

IO Completion Ports

- Available since NT 3.5
- based on thread-pooling: multiple threads all wait for the same completion port
- load management controlled by OS
  - completion port carries "concurrency" value
- handles get associated with completion port
  - it even works for sockets
  - application needs to use ReadFileEx, WriteFileEx
  - pool threads call GetQueuedCompletionStatus