Server Architectures

CS 360 Internet Programming

Daniel Zappala

Brigham Young University
Computer Science Department

Building a Scalable Server

- provide good response time to each client
- handle many clients simultaneously

Resource Pools

- one bottleneck to server performance is the operating system
  - system calls to allocate memory, access a file, or create a child process take significant amounts of time
  - as with many scaling problems in computer systems, caching is one solution
- resource pool: application-level data structure to allocate and cache resources
  - allocate and free memory in the application instead of using a system call
  - cache files, recent responses
  - limit critical functions to a small, well-tested part of code

Multi-Processing Architectures

- a critical factor in web server performance is how each new connection is handled
- creating a new thread or process for each connection is not scalable
    - operating system overhead to create and switch among them
    - high CPU utilization from many simultaneous threads or processes
    - limit on the number of threads or processes you can create
- scalable architectures
    - event-driven
    - thread pool
    - hybrid

Event Driven Architecture

- one thread handles all events
- must multiplex handling of many clients and their messages
    - use `select()`, `poll()` or `epoll()` to multiplex socket I/O events
    - provide a list of sockets waiting for I/O events
    - sleeps until an event occurs on one or more sockets
    - can provide a timeout to limit waiting time
- some evidence that it can be more efficient than process or thread architectures

poll()

```c
#include <poll.h>

int poll(struct pollfd *fds, nfds_t nfds, int timeout);

struct pollfd {
  int   fd;          /* file descriptor */
  short events;      /* requested events */
  short revents;     /* returned events */
};
```

- provide a list of file descriptors to monitor for events
- returns the number of file descriptors with that are ready for I/O, or zero if the timeout occurred, or -1 on error
- *see example code on web site*

epoll()

```
#include <sys/epoll.h>

int epoll_create(int size);
int epoll_wait(int epfd, struct epoll_event *events,
               int maxevents, int timeout);
```

- create a poll instance, which returns a poll file descriptor
- then wait for input on a set of events
- returns the number of file descriptors ready for I/O, or zero if the timeout occured, or -1 on error
- more scalable version of `poll()`, used by newer web servers
- *see example code on web site*

Coding Practice

- no shared memory synchronization needed
- must be careful how I/O events are handled
  - with blocking `recv()` – call only once per socket in the event processing loop
  - with non-blocking `recv()` – call as much as needed to handle socket events until it returns EAGAIN or EWOULDBLOCK
- must keep a separate `recv()` cache for each socket, since all sockets are handled by a single thread
- to time out idle sockets
  - keep a variable for each socket that tracks the last time it had an I/O event
  - put a timeout in `epoll()`
  - time how long it takes for `epoll()` to return
  - compare this time to the last time each socket had an I/O event, and if it has been idle too long, close it

Thread Pool Architecture

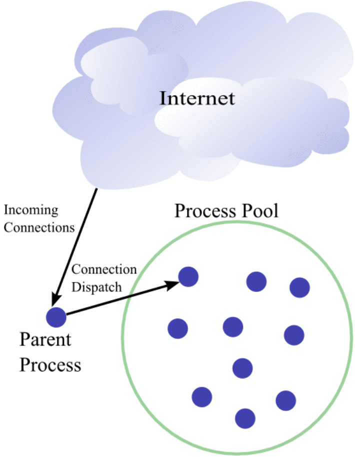
- create a pool of worker threads
- create a listening thread
- listener thread
  - handles accept()
  - put each accepted client into a thread-safe queue
- worker threads
  - dequeue clients from the thread-safe queue
  - handle all I/O events for the socket until it is closed or times out
  - *or* handle one I/O event for the socket and then put it back into the queue

Coding Practice

- must be careful with shared memory
  - use mutexes and condition variables
  - or use semaphores
- decide how to handle `recv()` caching
  - if a worker thread handles *all* I/O events for the socket until it is closed or times out, then it can use a separate cache per thread
  - if a worker thread handles *one* I/O event for the socket and returns it to the queue, then it must use a separate cache per socket
- to time out idle sockets
  - use `setsockopt()` to create a timeout for each client socket
  - if `recv()` returns EAGAIN, then close the socket and dequeue a new client

prefork Module

- default multi-processing module
- process pool
  - pre-fork a set of processes
  - avoids system overhead of creating a new process for a new request
- **places a hard limit on the number of simultaneous clients**



http://httpd.apache.org/docs/2.2/mod/prefork.html

Load Balancing

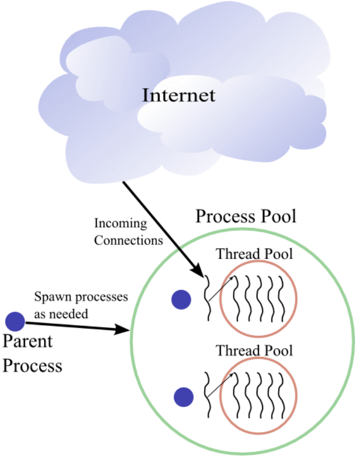
- server must balance between
  - too few processes: slow response to new clients
  - too many processes: idle processes consume resources
- configuration
  - `StartServers`: initial number of processes (default 5)
  - `MaxClients`: maximum number of processes (default 150)
  - `MinSpareServers`, `MaxSpareServers`: limits minimum and maximum number of idle processes (5 - 10)
  - `MaxRequestsPerChild`: limits number of requests for a process, after which it will terminate (default 0)
    - helpful to limit the amount of memory leakage a process can cause
    - reduces the number of processes when server load reduces
- server limits
  - `ServerLimit`: hard limit on number of active child processes (256)

TCP and HTTP Configuration

- TCP
  - ListenBackLog: listen() backlog (default 511), OS max often lower - 128 in Linux
  - for further TCP tuning, http://www-didc.lbl.gov/TCP-tuning/
- HTTP
  - MaxKeepAliveRequests: maximum number of requests that can be processed in a single persistent connection (100)
  - KeepAliveTimeout: maximum idle time for a persistent connection (15 seconds)

worker Module

- hybrid multi-process, multi-threaded server
  - pool of processes, each with a fixed number of server threads
  - listener thread accepts connections and passes them to server thread
  - semaphore protects accept()
  - don't call accept() unless there is some idle thread
- **places a hard limit on the number of simultaneous clients**



http://httpd.apache.org/docs/2.2/mod/worker.html

Load Balancing

- try to maintain a pool of idle threads that are ready to serve incoming requests
- configuration
  - StartServers: initial number of processes (default 2)
  - ThreadsPerChild : number of threads per process (25)
  - MaxClients: maximum total number of threads (150)
  - assesses total number of idle threads and forks or kills processes as needed
  - MinSpareThreads, MaxSpareThreads give bounds (25 - 75)
- server limits
  - ServerLimit: hard limit on number of active child processes (16)
  - ThreadLimit: hard limit on number of active server threads (64), never higher than 20000

Experimental Multi-Processing Modules

- *event* module
  - based on worker module
  - better handling of TCP connections that are idle
- *threadpool* module (Apache 2.0)
  - queues idle worker threads instead of queueing connections
  - benchmark testing shows it does not perform as well as *worker*
- *leader* module (Apache 2.0)
  - uses Leaders/Followers design pattern from academic paper
  - leader thread waits for events
  - when an event occurs, leader promotes a waiting follower thread to be the new leader and then processes the event
  - results in more efficient thread processing on multi-processor systems and for large scale, mult-tier web architectures