CS 360 Internet Programming

Concurrent Programming

Web Server Architectures

Daniel Zappala
Computer Science
Brigham Young University

Building a Scalable Web Server

- handling an HTTP request
  - map the URL to a resource
  - check whether client has permission to access the resource
  - choose a handler and generate a response
  - transmit the response to the client
  - log the request
- must handle many clients simultaneously
- must do this as fast as possible

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, URL mappings, recent responses
  - limits 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
  - common optimization strategy: *identify the most commonly-executed code and make this run as fast as possible*
  - common case: accept a client and return several static objects
  - make this run fast: pre-allocate a process or thread, cache commonly-used files and the HTTP message for the response
- must multiplex handling many connections simultaneously
  - `select()`, `poll()`: event-driven, singly-threaded
  - `fork()`: create a new process for a connection
  - `pthread_create()`: create a new thread for a connection
- synchronization among processes/threads
  - shared memory: semaphores
  - message passing

Event Driven Architecture

- one process handles all events
- must multiplex handling of many clients and their messages
  - use `select()` or `poll()` 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
- must use non-blocking system calls
- some evidence that it can be more efficient than process or thread architectures

Process Driven Architecture

- devote a separate process/thread to each event
  - master process listens for connections
  - master creates a separate process/thread for each new connection
- performance considerations
  - creating a new process involves significant overhead
  - threads are less expensive, but still involve overhead
- may create too many processes/threads on a busy server

Process/Thread Pool Architecture

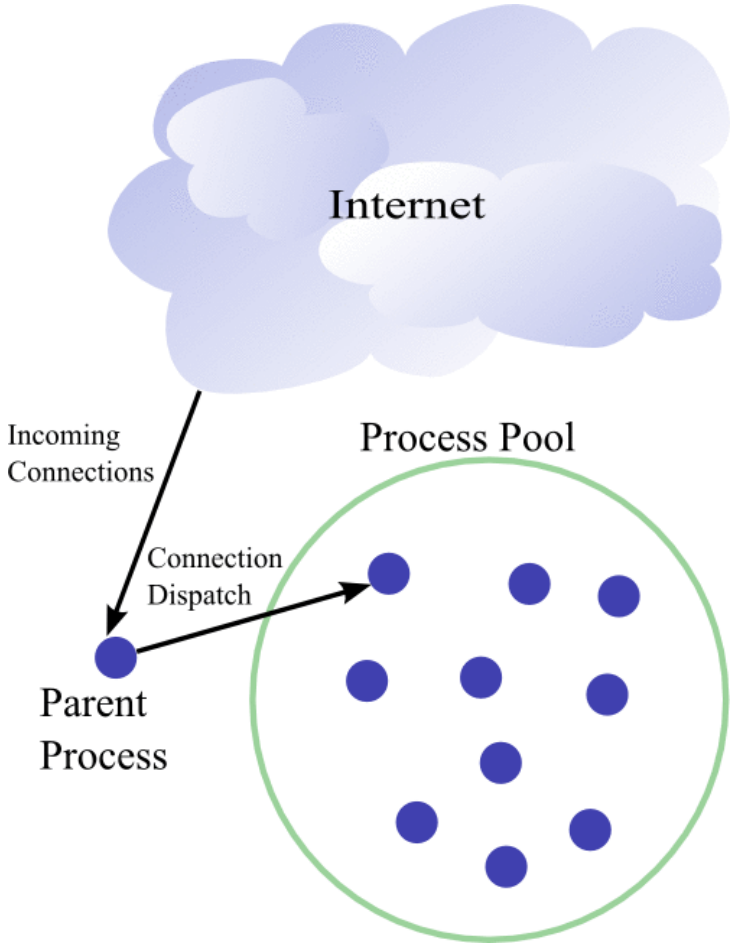
- master thread
  - creates a pool of threads
  - listens for incoming connections
  - places connections on a shared queue
- processes/threads
  - take connections from shared queue
  - handle one I/O event for the connection
  - return connection to the queue
  - live for a certain number of events (prevents long-lived memory leaks)
- need memory synchronization

Hybrid Architectures

- each process can handle multiple requests
  - each process is an event-driven server
  - must coordinate switching among events/requests
- each process controls several threads
  - threads can share resources easily
  - requires some synchronization primitives
- event driven server that handles fast tasks but spawns helper processes for time-consuming requests

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

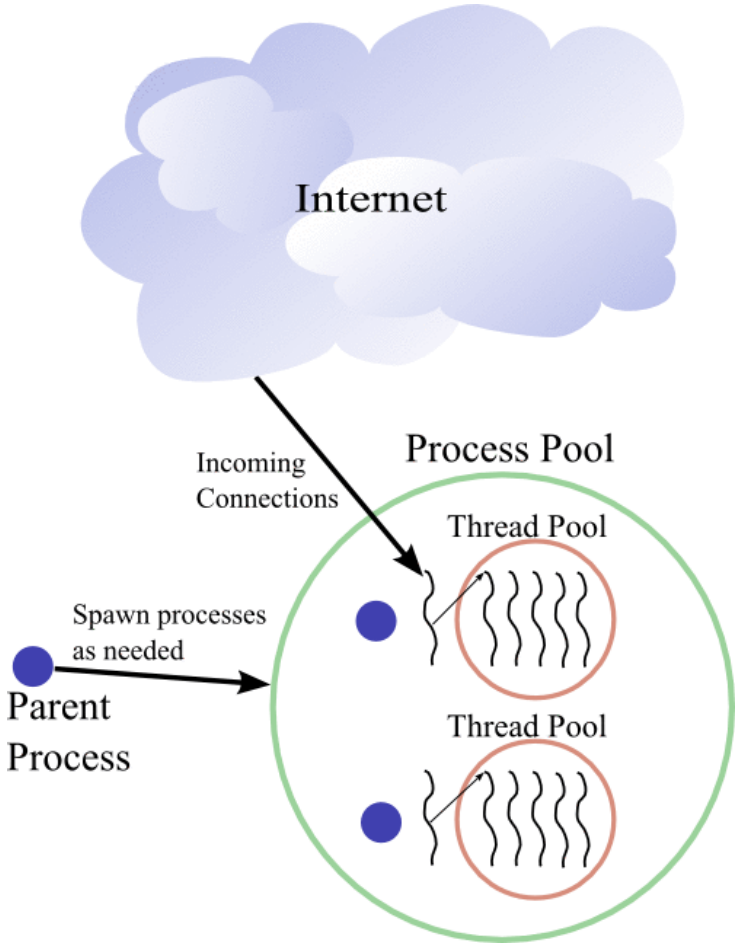
- server must balance between
  - too few processes: slow response to new clients
  - too many processes: idle processes consume resources
- configuration
  - `StartServers`: initial number of processes (default 5)
  - `MaxClients`: maximum number of processes (default 150)
  - `MinSpareServers`, `MaxSpareServers`: 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