CS4254

Computer Network Architecture and Programming

Dr. Ayman A. Abdel-Hamid

Computer Science Department

Virginia Tech

Client Server Design Alternatives

Outline

- Client Server Design Alternatives (Chapter 30)
  - ➢Introduction
  - ➢TCP Test Client
  - ➢Different TCP Server Alternatives
  - ➢Experiments Summary

Introduction 1/2

- Options of process control when writing a Unix Server
  - ➢ Iterative server
  - ➢ Fork-based, concurrent server. Spawn a child process for every client
  - ➢ Single process using **select** to handle any number of clients
  - ➢ Thread-based, concurrent server. Create one thread per client
- Two more alternatives
  - ➢ Pre-forking → create a pool of child processes
  - ➢ Pre-threading → create a pool of available threads
- Details for pre-forking or pre-threading
  - – What if there is not enough processes or threads in the pool?
  - – What if there are too many processes or threads in the pool?
  - – How can the parent and its children or threads synchronize with each other?

Introduction 2/2

- Testing strategy
  - ➢ Typical web scenario (small request to server, who responds with data back to the client
  - ➢ Run multiple instances of a client against each server, measuring the CPU time required to service a fixed number of client requests (see Figs 30.1 and 30.2)
  - ➢ Times in figure measure CPU time required for *process control* (measurement for iterative server is the baseline)
  - ➢ Run client of 2 different hosts on same subnet as server. Both clients spawn 5 children to create 5 simultaneous connections to the server (max of 10 connections)
  - ➢ Each client requests 4,000 bytes from the server
  - ➢ When a pre-forked or pre-threaded server is involved, the server creates 15 children or threads when it starts

TCP Test Client

- Source code in **server/client.c**
- Usage
  - ➢ **Client** <hostname or IP address of server> <Server port> <#children> <#loops/child> <#bytes/request>
  - ➢ Typical usage >**client** 192.168.1.20 8888 5 500 4000
  - ➢ 2,500 TCP connections to server
    - ✓500 connections from each of five children
  - ➢ On each connection, 5 bytes sent to server (“4000\n”)
  - ➢ 4000 bytes sent from server back to client
  - ➢ Client run on 2 different hosts → total of 5000 connections (max of 10 simultaneous connections to server)

TCP Concurrent Server, 1 Child per Client

- Traditional Iterative Server in **server/serv00.c**
- Source code in **server/serv01.c** and **server/web_child.c**
- Problem is the amount of CPU time it takes to fork a child for each client
- Handles SIGCHLD
- Handles SIGINT for data collection upon user input (terminal interrupt key)
  - ➢ Print CPU time required for the program
  - ➢ Source code in **server/pr_cpu_time.c**
  - Return resource utilization of calling process and terminated children of calling process
  - Total user time and total system time
- Results are in row 1 of Fig. 30.1 (largest CPU time)

TCP Pre-forked Server

No Locking around Accept 1/2

- Server pre-forks a number of children when it starts
- Children ready to service clients
- How many children to pre-fork?
- What happens if number of children equals number of clients?
  - ➢ Can monitor the number of available children
    - ✓Drops below some threshold → fork additional
    - ✓Number of available children exceeds some threshold → terminate some of the excess children
- Source code in **server/serv02.c** and **server/child02.c**
  - ➢ Usage: >**serv02** [<host>] <port#> <#children>
- Need a new **SIGINT** handler since **getrusage()** reports resource utilization of terminated children → terminate all children before calling **pr_cpu_time**

TCP Pre-forked Server

No Locking around Accept 2/2

- Every child calls **accept**?
  - ➢ 4.4BSD implementation
    - ✓Multiple processes calling accept on the same listening descriptor
    - ✓With *N* children, reference count for listening descriptor would be *N*+1 (Why?)
    - ✓When *N* children call accept → put to sleep by kernel
    - ✓When first client connection arrives, all *N* children are awakened
    - ✓First of the *N* to run obtains the connection and remaining *N*-1 go back to sleep
    - ✓Thundering herd problem!
    - ✓Results are in row 2 of Fig. 30.1
    - ✓*Metered version* to display how many client connections have been served by each child → Source code in **server/serv02m.c**, **server/child02m.c**, and **server/meter.c**

TCP Pre-forked Server
File Locking Around Accept

- Multiple processes calling **accept** on the same listening descriptor works only for Berkeley-derived kernels (**accept** implemented within the kernel)
- Some systems may not allow this (e.g., if **accept** implemented as a library function → System V Kernels)
- *Place a lock of some form around the call to accept*
- This version uses **POSIX** file locking with **fcntl** function
- Source code in **server/serv03.c** and **server/child03.c, and server/lock_fcntl.c**
- Results are in row 3 of Fig. 30.1
  - Locking adds to server's process control CPU time
- Metered version in **server/serv03m.c**, **server/child03m.c**
- Apache web server uses the pre-forked server with children blocked in accept if allowed, or file locking around accept

TCP Pre-forked Server
Thread Locking Around Accept

- File locking around **accept** is portable to all **POSIX**-compliant systems, but involves file system operations overhead
- This version uses *thread locking*
  - ➢ Have to inform thread library that **mutex** is shared among different processes
  - ➢ **Mutex** variable stored in memory that is shared between all processes
- Source code in **server/serv04.c**, **server/child04.c**, and **server/pthread_lock.c**
- Results in row 4 of Fig. 30.1
  - ➢ Thread locking faster than file locking

TCP Pre-forked Server, Descriptor Passing

- Only parent calls **accept** and then passes connected socket to one child
- Requires *descriptor passing* from parent to child
  - ➢ Using a stream pipe → Unix domain stream socket
- Parent must keep track of which children are busy and which are free (to pass new connected socket to a free child)
  - ➢ Data structure declared in **server/child.h**
- Source code in **server/serv05.c** and **server/child05.c**
- Results in row 5 of Fig 30.1
  - ➢ Slower than "locking around accept" versions
  - ➢ Overhead of writing to the stream pipe
- Client distribution among children in Fig 30.2

TCP Concurrent Server, 1 Thread/Client

- Source code in **server/serv06.c**
- Main thread calls **accept**
- Results in row 6 of Fig. 30.1
- Fastest so far!

TCP Pre-threaded Server, per-Thread Accept

- Create a pool of threads, where each thread calls accept
- Mutual exclusion on **accept** call using a mutex
- Source code in **server/serv07.c**, **server/pthread07.h**, and **server/pthread07.c**
- Results in row 7 of Fig. 30.1
  - *Faster than create one thread per client upon connection*
  - *Note that the numbers in Fig. 30.1 for this experiment seem incorrect*
- Client distribution among children in Fig 30.2

TCP Pre-threaded Server Main Thread Accept 1/2

- Create a pool of threads upon start
- Only main thread calls **accept** and passes each client connection one of the available threads in the pool
- How to pass connected socket to thread?
  - A shared array to hold connected sockets
  - Main thread deposits connected sockets into array (**iput** index)
  - Other threads retrieve from array (**iget** index)
  - if (**iget** == **iput**) → have to wait
  - Control access to array through a mutex and a condition variable

TCP Pre-threaded Server Main Thread Accept 2/2

- Source code in **server/serv08.c**, **server/pthread08.h**, and **server/pthread08.c**
- Results in row 8 of Fig. 30.1
  - Slower than per-thread accept→ use of **mutex** and condition variable
- Client distribution among children in Fig. 30.2

Experiments Summary 1/2

- Creating a pool of children or a pool of threads reduces process control CPU time compared to one-fork-per-client
- Some implementations allow multiple children or threads to block in a call to **accept**, while others need some type of lock around **accept**
- Having all children or threads accept is simpler and faster than having main thread call **accept** and then pass descriptor to child or thread
- Using threads is normally faster than using processes

Experiments Summary 2/2

Row	Server Description	Process Control CPU time (Difference from baseline)
0	Iterative Server (baseline)	0.0
1	Concurrent Server, one fork per client request	20.90
2	Pre-fork, each child calling accept	1.80
3	Pre-forking, file locking around accept	2.07
4	Pre-forking, thread mutex locking around accept	1.75
5	Pre-fork, parent passing descriptor to child	2.58
6	One thread per client request	0.99
7	Pre-threaded, mutex locking to protect accept	**1.93** ?
8	Pre-threaded, main thread calling accept	2.05