Chapter 16

Concurrent Server Issues

Design Choice: Iterative vs. Concurrent

Concurrent (Multitasking): easier to program because each process handles one client.

More CPU because creating processes and context switching takes resources.

Iterative: Most efficient throughput.

Inappropriate for long interactions because one connection blocks all others.

Concurrent (Single process): efficient because it avoids creating processes and context switching.

Very difficult to program because a single procedure has to keep track of what lots of clients are doing, and keep them straight.

Unix has found a multiple process approach useful.

Concurrency wins on multi-processor machines

Concurrency Issue:

How many processes should be allowed to run at a time (level of concurrency)

Usually demand driven with an upper limit.

Concurrency Cost

Time required to create a process, c

Time required to process a request, p

If $p < c$, then concurrency looses, even with two requests:

Concurrent: $3c/2 + p$ time

Iterative: $3p/2$ time

Under heavy load, the concurrent server fails before the iterative (on a single processor).

If $p >> c$ and the client does lots of waiting during the interaction. This causes a long delay if you have an iterative server and more than one client.

Process Preallocation

Concept: Preallocate concurrent processes to avoid the cost of creating them on demand.

Master server preallocates N processes at start-up.
Each process waits for requests to arrive.
When a request arrives, one process handles it.

Result: Lower OS overhead

Multiserver coordination: Sometimes shared memory or message passing is available .

UNIX:

- shared sockets (children inherit descriptors)
- mutual exclusion for multiple processes trying to accept a connection from the same socket
- only one process gets the message

Connection Oriented: one server accepts the connection.

Conncetionless: All server processes use the same port, one gets the message and responds.

Example (almost): `nfsd` (network file system servers)

Delayed Process Allocation

Concept: Sometimes efficiency is improved by delaying process allocation

Recall: Concurrency works when the cost of creating a process is smaller than the cost of processing a request

Problem: The processing time may vary with the request.

Solution: Use delayed process allocation

Master server receives a request, starts a timer to measure elapsed time, and begins processing the request (iteratively)

If the timer expires before the server is through processing, the server creates a slave process to handle the rest of the request

Else, the master cancels the timer when the request is through.

Combining Preallocation and Delayed Allocation

(similar to what `httpd` does)

Both approaches use the same principle:

decouple the level of server concurrency from the number of currently active requests

The two ideas can both be combined:

Create new processes based on delayed process allocation technique
BUT leave the processes active after they finish processing, waiting for the next incoming request

Difficulty: How do you know when a slave process should exit?