Models for Implementing Servers
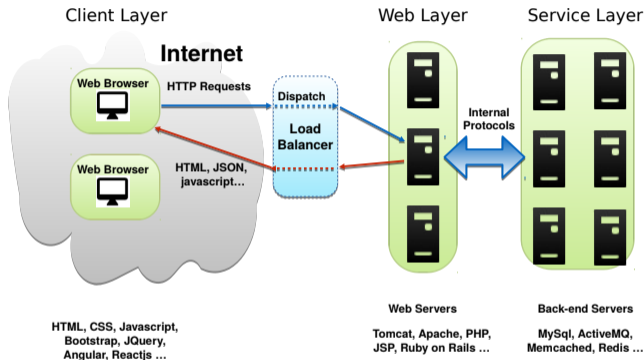
Godmar Back

Virginia Tech

August 9, 2020

Architecture of Internet Applications

Client Layer   Web Layer   Service Layer

- Motivation: modern cloud services need to serve large numbers of clients with finite resources
- The applications are often *multi-tiered*

Iterative Server

- An iterative server accepts one client at a time, serves this client, repeats
- Advantage:
  - Simple
  - Requires little support from OS, suitable for embedded devices
- Drawbacks:
  - While this client is being served, other clients have to wait, even while the server itself is blocked waiting for data from another tier
  - Can use only a single CPU
  - Results in high latency for clients and low resource utilization

High Concurrency Servers

- High-concurrency servers handle multiple clients concurrently
  - Each client may be in a different stage of the sequence necessary to process their request
  - Clients may be short or long lived.
- Goal is to increase performance with offered load, reach peak, and degrade gracefully under overload
- Two basic models:
  - using a separate execution context for each client (threads or processes)
  - using an event-based approach to multiplex multiple connections within a single execution context (and then use one such context per CPU)
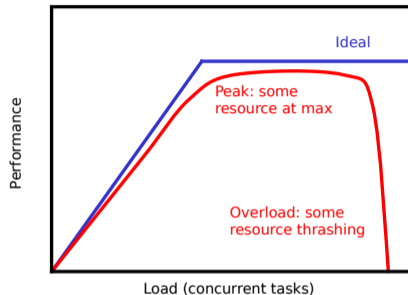

Figure 1: Source: von Behren [2]

Using Multiple Execution Contexts

- May use threads or processes
- Requires policy for managing concurrency
- Different arrangements differ with respect to robustness and scalability
- Underlying system maintains execution contexts and exploits process state transitions
  - e.g., `read(2)` puts only the calling thread/process into the BLOCKED state
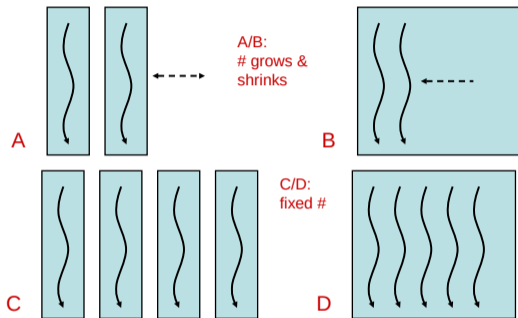

A/B:
grows & shrinks

C/D:
fixed #

A   B

C   D

Figure 2: Prefork vs. On-Demand Models

Thread/Process-Based Model

Pros:

- Programmer's perspective: Linear control flow, using functions that block when I/O is not ready
- Unix file descriptor model well suited for sharing between processes and/or threads
- Multiprocess model exploits process isolation for robustness

Cons:

- Potential for concurrency-related bugs, particularly under fully preemptive scheduling regime
    - dependent on the amount of data shared between threads
- Potential overhead related to state transitions (mode switch, context switch, scheduler)
- Concurrency level can be difficult to tune

Q: how can the state transition overhead be avoided/reduced?

Event-Based Model

Idea

Reorganize the necessary computation so that the application, not the OS scheduler, decides what work is to be next, guided by information from the OS about which clients have I/O data pending. Thus, gain performance by avoiding the overhead of state transitions (READY to BLOCKED, BLOCKED to READY, reschedule from READY to RUNNING). Also gain the opportunity to optimize the application based on knowledge of the stages of request processing.

Questions:

- How to avoid blocking?
- How to manage a task's progress to completion?

Historical Aside

Both models perform the same work and are equivalent, which has been known since 1978 [1]. But they can perform differently.

Programming Model Comparison

Event-based Model

```c
struct conn_state {
    request req;
    response res;
};
void handle_event(int event, conn_state *s) {
  switch (event) {
    case REQUEST_ARRIVED:
      read_request_from_client(&s->req);
      process_request(&s->req);
      send_req_to_database(&s->req);
      return;
    case DB_REQUEST_COMPLETED:
      get_db_response(&s->res);
      transform_response(&s->res);
      send_response(&s->res);
      return;
  }
}
```

```c
// event loop:
while (true) {
    // identify pending events
    // handle pending events
}
```

Thread-based Model

```c
void handle_request() {
    request req;
    read_request_from_client(&req);
    process_request(&req);
    response res;
    contact_database(&req, &res);
    transform_response(&res);
    send_response(&res);
}
```

VIRGINIA TECH.

Writing Event-Based Code

- Basic idea: identify blocking points and introduce a state machine to transition at each point. Event handler must not block
- Add an event loop that identifies pending events and calls appropriate handlers.
- Maintain actual client state explicitly – as opposed to using local (on the stack) variables, hence the term "stack ripping"
- This is generally complex. Actual events - in a standard http server - are as fine-grained as receiving a group of bytes from a network.
  - Consequence: program must maintain a state machine for http parsing that can advance with an arbitrary non-zero number of bytes
    ```
    GET /api/login HTTP/1.1\r\n
    Host: localhost:9999\r\n
    User-Agent: curl/7.61.1\r\n
    Accept: */*\r\n
    \r\n
    ```
  - E.g. nginx http parser

Performing Asynchronous I/O

- The event-based model requires that the event loop efficiently learns when new events become available; for network servers, this corresponds to
  - New connection attempts
  - New data received on established connections
  - Connections being closed
- This requires I/O multiplexing system calls that efficiently inform the program which file descriptors have data pending:
  - *readable*: data has been received and buffered, EOF, or new client is pending for accepting sockets
  - *writable*: possible to write to a TCP connection without blocking due to flow control
- May be combined with non-blocking mode [Klitzke'17]:
  - Calls such as `read(2)` return EWOULDBLOCK if calls were to block

Multiplexing I/O Interfaces

- OS provide I/O multiplexing interfaces that allow a process to learn about multiple file descriptors at once
  - Traditional Unix: select(2), poll(2)
  - Linux: epoll(7)
  - BSD kqueue
  - Windows: I/O completion ports
- Efficiency & Complexity
  - Efficient: (algorithmic) complexity proportional to number of file descriptors that provide events
  - Can be tricky to use, particularly when combined with multi-threading and non-blocking file descriptors
- Libraries such as libevents or libuv provide an abstraction layer on top of these low-level facilities

Emerging async I/O facilities in High-Level Languages

- High-level languages (e.g. ES6 JavaScript, Python 3, C++20) provide async/await constructs that allow a programmer to write what looks like thread-based code, but which is executed under an event-based model
- Often integrated into the interpreter/virtual machine/runtime system of these languages
  - Programmer still required to use async/await for any "blocking" function
  - May be implemented under the hood using blocking helper threads that provide higher-level events to the event queue
- Can potentially avoid the low-level race conditions that may occur under a preemptive, multi-threaded regime
  - Potential for concurrency-related bugs such as ordering violations remains

Conclusion

- The existing multi-threading and multi-process facilities are well-suited to writing concurrent servers
- However, for high-performance, high-concurrent servers, an alternative event-based model may lead to higher throughput. Paid for with a more complex programming model
- Direct support for asynchronous I/O in the language may reduce this cost

References

[1] Hugh C. Lauer and Roger M. Needham.
On the duality of operating system structures.
SIGOPS Oper. Syst. Rev., 13(2):3–19, April 1979.

[2] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer.
Capriccio: Scalable threads for internet services.
In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 268–281, New York, NY, USA, 2003. Association for Computing Machinery.