Linux Threads

- Dynamic Creation
  - Pthread_create
- Concurrent Execution
  - All threads appear to execute at the same time.
- Preemption
  - Sched_yield to voluntarily release allocated time.
- Private Local Variables
  - Each thread has private stack.
- Shared Globals
  - All threads within the same process share global variables.
- Shared File Descriptions
  - All threads within the same process share files.

Synchronization

- Mutex
  - Generally a separate mutex variable is used for each variables to be protected.
  - pthread_mutex_init
  - pthread_mutex_lock
  - pthread_mutex_unlock.
- Semaphores
  - Generally one semaphore for each resource with count N.
  - Sem_init
  - Sem_wait
  - Sem_post.
- Condition Variables
  - A tool for applications in need of busy wait
Inter Process Signaling

- `signal(int signumber, void *handler)`
  - A process expecting a signal can register a handler subroutine if a specific signal comes. If a catch is not declared a signal can kill a process.
  - Instead of a handler routine, the catch action can be SIG_IGN (ignore it) or SIG_DFL (take default action).

- `pause()`
  - The process (or thread) expecting the signal can sleep until a signal is received that either terminates it or causes it to call a signal-catching function. It only returns when a signal was caught and the signal-catching function returned.
  - See also `sigtimedwait()`, `sigwaitinfo()`, and `sigsuspend()`

- `kill(pid_t pid, int sig)`
  - One process can send signal to another process.

Some Signals

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>signal: abort</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>signal: alarm clock</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>signal: bus error</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>signal: (exit of a) child</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>signal: continue</td>
</tr>
<tr>
<td>SIGEMT</td>
<td>signal: EMT instruction</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>signal: floating point exception</td>
</tr>
<tr>
<td>SIGHUP</td>
<td>signal: hangup</td>
</tr>
<tr>
<td>SIGILL</td>
<td>signal: illegal instruction</td>
</tr>
<tr>
<td>SIGINT</td>
<td>signal: interruption</td>
</tr>
<tr>
<td>SIGIO</td>
<td>signal: input/output (possible or completed)</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>signal: segmentation violation</td>
</tr>
<tr>
<td>SIGSYS</td>
<td>signal: (bad argument to) system call</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>signal: terminate</td>
</tr>
<tr>
<td>SIGTTIN</td>
<td>signal: TTY input</td>
</tr>
<tr>
<td>SIGTTOU</td>
<td>signal: TTY output</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>Signal: terminal stop</td>
</tr>
<tr>
<td>SIGURG</td>
<td>signal: urgent I/O condition</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>signal: user-defined signal 1 (also: SIGUSR2)</td>
</tr>
<tr>
<td>SIGWINCH</td>
<td>signal: window (size) changed</td>
</tr>
</tbody>
</table>

Time & Timer

- `time(time_t *t)`
  - `time_t` is signed integer. Returns seconds elapsed since epoch.

- `Sleep(unsigned int seconds)`
  - Causes the current process to sleep atleast seconds or until a signal that the process does not ignore is received by the process.

- Interval timer
  - `setitimer(int which, *newval, *oldval)`
  - `getitimer(int which, *val)`
  - Interval timer once enabled can be used to deliver signals to a process on a regular basis. The which can be set to get wallclock (ITIMER_REAL), process execution time (ITIMER_VIRTUAL), or system execution slice (ITIMER_PROF).

Concurrency and Advanced Network-based Applications

Issues in Client Design-1

- Choosing A Local Port Number
  - A client can choose any port number.
  - No conflict with other port numbers in use.
  - Not a well known server port number.
  - If a process does not explicitly call bind, Connect or listen, which ever is called kernel automatically picks a valid ephemeral port and source IP.

- Choosing A Local IP address
  - In a multi-homed computer applications may not know which IP adapter is in use.
  - While binding, setting IP address =0 (INADDR_ANY) will allow the kernel to pick the correct IP address.

Issues in Client Design-2

- Reading from a Stream
  - Only one write operation may require multiple read at the other end.
  - The flag can be 0 or AND of few values such as:
    - MSG_PEEK: look at the data that is available to read, but without having system discard it from system buffer.
    - MSG_WAITALL: do not return until the specified amount of data is received.
  - /* Repeatedly read data from socket and write to user's screen. */
  - `n = recv(sd, buf, sizeof(buf), 0);`
  - `while (n > 0) { write(1, buf, n); n = recv(sd, buf, sizeof(buf), 0); }`
Issues in Server Design-1

- Binding to a Well Known Port
  - If the service has to be available to a wide audience the server must bind to a well known port.
- Setting up the IP address
  - Like clients a server may leave it up to the kernel to set the IP address to avoid confusion in multi-homed hosts.

Concurrent Server

TCP Port Numbers and Concurrent Servers

Issues in Server Design-2

- Concurrent vs. Iterative server
  - Simplicity, response time
  - Real vs. apparent concurrency
- Connection-Oriented vs. Connectionless Access
  - Reliability, persistence, data volume, flow-control

Connection Oriented & Connectionless Communication

- UDP Client
- UDP Server
Concurrency Management
Perspectives:
• User perspective: response time
• System perspective: impact on resources.
Issues:
• How can programmer know whether concurrency is warranted?
• How to determine which design is optimal?
• How can a programmer estimate demand or service time?
Concepts:
• Level of Concurrency
• Demand-Driven Concurrency

Cost of Concurrency
- Overhead and Delay

<table>
<thead>
<tr>
<th>Iterative</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process req.1</td>
<td>Process req.1</td>
</tr>
<tr>
<td>Process req.2</td>
<td>Process req.2</td>
</tr>
<tr>
<td>0</td>
<td>p</td>
</tr>
<tr>
<td>2p</td>
<td></td>
</tr>
</tbody>
</table>

Quiz: if the rate at which requests arrives exceeds $1/c$ but is less than $1/p$ which implementation can handle the load?

Process Preallocation
- Demand-driven concurrency can be avoided by limiting the maximum level of concurrency.
- To avoid the run time process creation delay, preallocate service processes.
- Design:
  - Master server creates N slaves at the beginning.
  - Each slave waits/sleeps using OS support.
  - When service request arrives each slave by turn picks it.
  - When done slaves do not exit.
  - Preallocation allows the server process to switch and move to next process faster.

Preallocation in a Connection Oriented Server
Master
Slave forking
Slave-1
Slave-2
Slave-2

Quiz: How do the slaves decide that two of them do not jump for serving one client?

Preallocation in a Connectionless Server
Master
Slave forking
Slave-1
Slave-2
Slave-2

Slaves use the same descriptor to listen for request and send data. Requests arrive in UDP. Descriptors are automatically freed after each communication.

Delayed Process Allocation
- Standby slave approach solve runtime process creation.
- but they too costs in terms of OS resource management.
- Iterative server can yield faster service and higher throughput if the service time is small.
- But, how can programmer know the service time?
- Design:
  - the server estimates the processing time dynamically by looking into service parameters (I/O size bound).
  - server starts processing a new request iteratively.
  - and starts a timer.
  - On time out, it invokes a concurrent slave.

Quiz: Delayed process creation and preallocation are they are just opposite principles?
Combined Technique

- Design:
  - server starts without any slave.
  - creates a slave only when timer expires.
  - but, once a slave has been created, it does not exit immediately.
  - Slaves can also spawn slaves, if needed.

- How to control concurrency?
  - master specifies limit of concurrency MAX to slaves.
  - Slave exits after a specified period of inactivity.

Multi Protocol Servers

- One server can talk via several underlying transport layer network protocols.
  - Example: DAYTIME server maintained in UNIX systems can talk via TCP and UDP both.
- Advantages:
  - easy software maintenance
  - easy debugging
  - service code can be reused.
  - less impact on system resources.
- Disadvantage:
  - less flexibility for network administrator.
- Design:
  - multiple protocol specific listening ports
  - iterative polling.

Multi Service Servers

- A typical host may run multiple services. One server can be designed to provide a set of services instead of one.

- Advantages:
  - easy maintenance.
  - communication code can be reused.
  - less impact on system resources.

- Design:
  - requires service descriptor code.
  - server use descriptor code based switch.
  - single process approach
  - concurrent approach
  - super server (UNIX INETD)

Advantages of Concurrency to Servers

- Improves observed response time.
- Thus it can increase client throughput.
- It can eliminate potential deadlocks and some denial of service attacks.
- It makes server design modular.
- Concurrent execution can easily be ported on multiprocessor hardware.

Concurrency in Client Design

- Concurrency makes client design simpler and modular too.
- Concurrent client can contact several servers without being held by one.
- Concurrency can allow user to change parameters, inquire about client parameters, or control client processing dynamically.
Web Servers

Client Request Handling
- Read and parse the HTTP request message
- Translate the URL to the file name.
- Determine if the request is authorized.
- Generate and transmit the response.

Access Control
- Authentication
  - Identify the user.
  - Must be performed for each request.
  - Browser typically helps user.
  - Users can be lumped in user groups.
- Authorization
  - Which user has access to which resource.
  - Policy is written as a "Access Control List".
  - Can create severe server load, as "subdirectories" exists.
Dynamic Response Handling

- Server Side Include
  - `<!-echo var = "LAST_MODIFIED" ->` etc.
  - Requires every HTML docs to be parsed and scanned.
  - Use of filename extension *.shtml.
  - * php: PHP (Personal Home Page) starts a PHP processor.
  - * asp: Microsoft Active Server pages.

- Server Scripts
  - Not the HTML but the URL points to a script, CGI etc.
  - Separate process can be invoked.
  - Software module in same process.
  - Persistent process contacted by server.

Server Performance: Cookies & State Management

Cookies

- In index to a particular user which can be used by server to identify an user across connection, or even user session.
- Reduces data volume as all user data can be stored in a server side database and cookies can be used as index.
- Also a threat to user privacy.
- First introduced in 1994 in Netscape 4.0. standardized in October 2000 in RFC 2965.
- Typical defaults: each one upto 4KB, maximum of 20 Cookies per server, and total of 300 per browser.

Privacy Concern from Cookies

- Web-sites can identify users and start accumulating session history.
- 3-rd party sites can also collect information about user by embedded URLs (Web Bug).
- User Controls:
  - Decide if one will accept it or not.
  - Set limit on size and number.
  - Set acceptance based on site site/domain.
  - Set limit on the duration.
  - Set limit on current page sites.

Performance Enhancement

- Store in memory the popular files and save disk reading time.
  - How to maintain consistency?
    - If file changes the memory must be invalidated.
    - Check last-update time before serving.
- Store dynamically generated responses for popular queries.
  - How to maintain consistency?
    - QUIZ
- Fast Request & Processing
  - Store URL to file name mapping.
  - Catalogue access and base file attributes (size, last update time) which are to used in response.
  - Response Header information (such as type) stays unchanged and can be pre-catalogued.

Server Farm

- Multiple Web site on a Single Machine
  - Separates content authoring from hosting.
  - Advanced computing and networking infrastructure.
  - Cost amortization by mixing services.
  - Privacy of the content author.

- Concerns:
  - IP address depletion.
  - Synchronization for content upload.
  - Virtual hosting.
Virtual Hosting

- HTTP/1.0 browsers. To get bar.html from www.foo.com may send a request “GET /bar.html HTTP/1.0” to www.foo.com.
- However, if www.virtualhost.com does not see the vanity URL, then it cannot tell which one is the intended server. Thus HTTP/1.0 requires distinct IP for each Web server.
- Solution: HTTP/1.1 introduced the “Host” header, which can specify the vanity URL.
  - GET /bar.html HTTP/1.1
  - Host: www.foo.bar
- The vanity URL can be resolved to www.virtualhost.com’s server IP. But the server by looking into vanity URL header be redirect the message to appropriate server.

Server Replication

- Mirror Site:
  - Replicate the content in multiple servers to enable load distribution.
  - How to update and keep them synchronized?
  - How to redirect requests?
- Server Bank
  - Keep all in one logical front-end server. Locally distribute the load to individual servers working in the backend.
  - How to distribute request sequences evenly?
  - Replicas file system updates.
  - Access control/cookie management.

Server Bank

- Serving high rate of requests requires multiple web-server machines to work concurrently.

![Server Bank Diagram](image)

Case Study: A Business Site

Reference: Arlitt, Krishnamurthy and Rola, Hewlett-Packard Laboratories, Aug 2001

Workload Characterization (Resource Types)

Table 1: Breakdown of Requests by Resource Type

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>March 2000 (%)</th>
<th>July 2000 (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>95.58</td>
<td>95.72</td>
</tr>
<tr>
<td>HTML</td>
<td>2.58</td>
<td>0.46</td>
</tr>
<tr>
<td>Image</td>
<td>0.37</td>
<td>0.16</td>
</tr>
<tr>
<td>Text</td>
<td>0.03</td>
<td>0.01</td>
</tr>
<tr>
<td>Other</td>
<td>0.44</td>
<td>0.63</td>
</tr>
<tr>
<td>Total</td>
<td>100.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>

In the above table requests containing parameters (e.g., examples/example.txt) are placed in the dynamic category. They can only be generated by application servers, not web-servers. And, of course, not all data generated by app servers is dynamic.

Workload Characterization (Usage Analysis)

![Workload Characterization](image)

Fig. 1: Traffic volume (a) March 2006 (b) July 2006.
Classes of Requests

<table>
<thead>
<tr>
<th>Types</th>
<th>Cacheable</th>
<th>Non-cacheable</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples</td>
<td>Product descriptions</td>
<td>Photos</td>
<td>Adding items to shopping carts</td>
</tr>
<tr>
<td>S or D7</td>
<td>Static</td>
<td>Dynamic</td>
<td>Dynamic</td>
</tr>
</tbody>
</table>

Personality

<table>
<thead>
<tr>
<th>Months</th>
<th>Cacheable</th>
<th>Non-cacheable</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>March</td>
<td>73.36</td>
<td>15.89</td>
<td>10.81</td>
</tr>
<tr>
<td>July</td>
<td>70.55</td>
<td>17.94</td>
<td>11.94</td>
</tr>
</tbody>
</table>

Session Analysis

- Every session is assigned a session number.
- A single session times out when over 15 mins.
- Robots consume resources, which we don’t want to see, but it’s difficult to tell the difference between a human user behavior and a robot behavior.
- In order to identify a robot in our survey, we assume sessions consisting of more than 30 requests are from robots. A detail analysis found that 9% of those sessions characterized as robots performed checkout operations.

Session Analysis (Classes of Sessions)

Over 40% of our sessions are characterized as barely cacheable. 90% of requests by these users are cacheable, which means they seldom do searching or checkout. They are “window shoppers.”