

Concurrent Online Servers

CS 256/456

Dept. of Computer Science, University of Rochester

Concurrent Online Servers

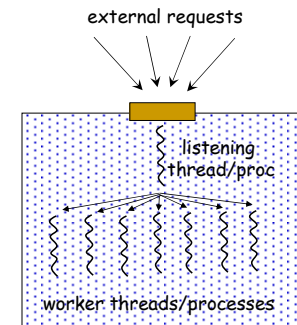
- Servers that
 - accept concurrent requests (potentially high concurrency)
 - serve online users (interactive responses)
- Examples
 - Web server: online server that implements HTTP
 - More complex ones: search engines (Google), online auction sites (eBay), discussion forums (slashdot),
- We examine supporting concurrent online servers
 - application level issues (with a good understanding of the OS)
 - OS issues

Issues of Concerns

- Performance
 - potentially high concurrency
 - fluctuating load
- Reliability
- Isolation
 - performance isolation
 - fault isolation
- Security
 - concerning network-oriented services
- Manageability

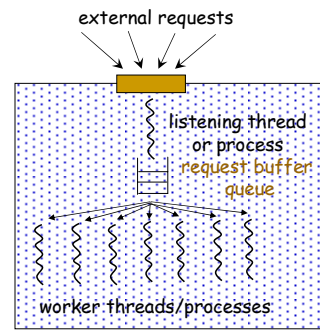
Multi-processing vs. Multi-threading

- Multi-processing server
 - each request is served by a process (Apache).
- Multi-threading server
 - each request is served by a thread.
- Compare multi-processing server with multi-threading server
 - efficiency
 - robustness/isolation
- Pooling can be used to reduce the overhead on process/thread creation and termination



Controlling the Concurrency

- Overhead of high concurrency
 - more frequent context switches?
 - or what?
 - e.g., scalability of the `select()` system call
- How to control the execution concurrency?
 - dropping requests
 - employ a request buffer queue



4/9/2007

CSC 256/456 - Spring 2007

5

User-level Threads

- Kernel threads
 - thread management/scheduling done by the OS kernel
- User threads
 - thread management/scheduling done at user-level
 - Benefits: (lightweight) less context switching overhead
- Problem of user threads
 - oblivious to kernel events, transparent to the kernel
 - e.g., all threads in a process are put to wait when only one of them blocks on I/O (e.g., `read()`)
- How to solve this problem?

4/9/2007

CSC 256/456 - Spring 2007

6

How to solve the blocking problem for user threads?

- For each potential blocking operation (e.g., `read()`)
 - the server forwards it to a *helper* process (either spawned or pooled); the server then does something else
 - when the blocking operation in the helper process completes, the server is informed through IPC
- OS support for asynchronous I/O
 - server doesn't call blocking operations (e.g., `read()`) directly
 - instead, it calls asynchronous I/O operations (e.g., `aio_read()`), which doesn't block
 - the server is notified by a signal (e.g., `SIGIO`) when the asynchronous I/O completes
 - an I/O handler is invoked at the receipt of the signal

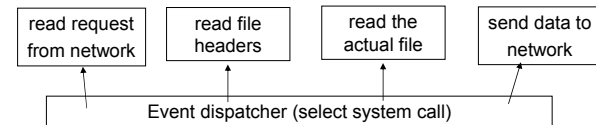
4/9/2007

CSC 256/456 - Spring 2007

7

Event-driven Servers

- Event-driven servers
 - divide request processing into stages, each of which is non-blocking
 - each stage is triggered by an event
 - the whole event controller runs in a single user thread



- Flash Web server [Pai et al, USENIX1999]

4/9/2007

CSC 256/456 - Spring 2007

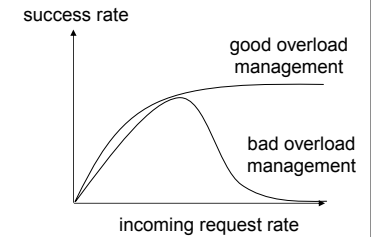
8

Resource Management

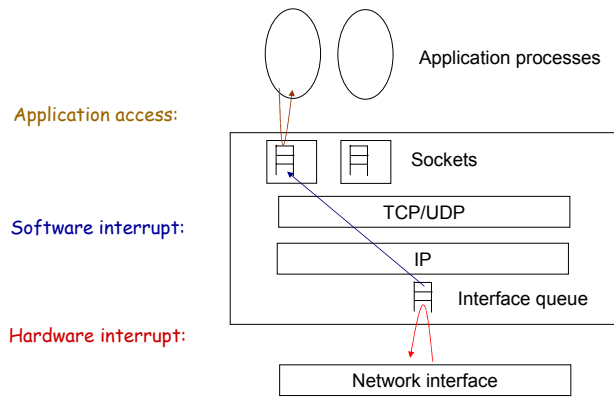
- Scheduling or request execution
 - according to default CPU scheduling policy
- Staged request scheduling
 - Each request execution is partitioned into stages (like in event-driven servers)
 - Request scheduling according to
 - which stage each request execution is at; and
 - whether the primary required resource is scarce or not.
- SEDA [Welsh et al., SOSP 2001]
- Capriccio [von Behren et al., SOSP 2003]

Handle Server Overload

- Overhead of server overload:
 - some requests have to be abandoned
 - when a request has to be abandoned, resources already consumed by this request is wasted
 - **principle**: when abandoning a request, do so as early as possible
- Managing overload?
 - drop requests if the buffer queue is already very long



OS Overhead for Each Request



Lazy Receiver Processing [Druschel&Banga OSDI1996]

