# *OS Support for Web Services*

An integrated view

By

Saumitra M Das

## *Why you should spend 1.53 hours listening to this?*

- Huge set of untrustworthy users
- Short TCP connections
- Long and variable network delays
- Transient flaky network connections
- No domain administration
- Large penalties for failure
- No downtime possible
- No load control

## *Motivating Arguments*

- What we have
  - Poor scalability
  - Priority inversion
  - Unfair resource allocation
  - Livelock under excess load
  - Instability under DoS attacks
  - No way to prioritize request handling
- What we need
  - Efficient support for event driven and multi-threaded servers
  - Scheduling control
  - Resource management
  - Scalability
  - Stability under DoS attacks

## *What do we want? The BIG picture*

"Ultimately the driving force for this discussion is how to get a better performing web server by hook or by crook"

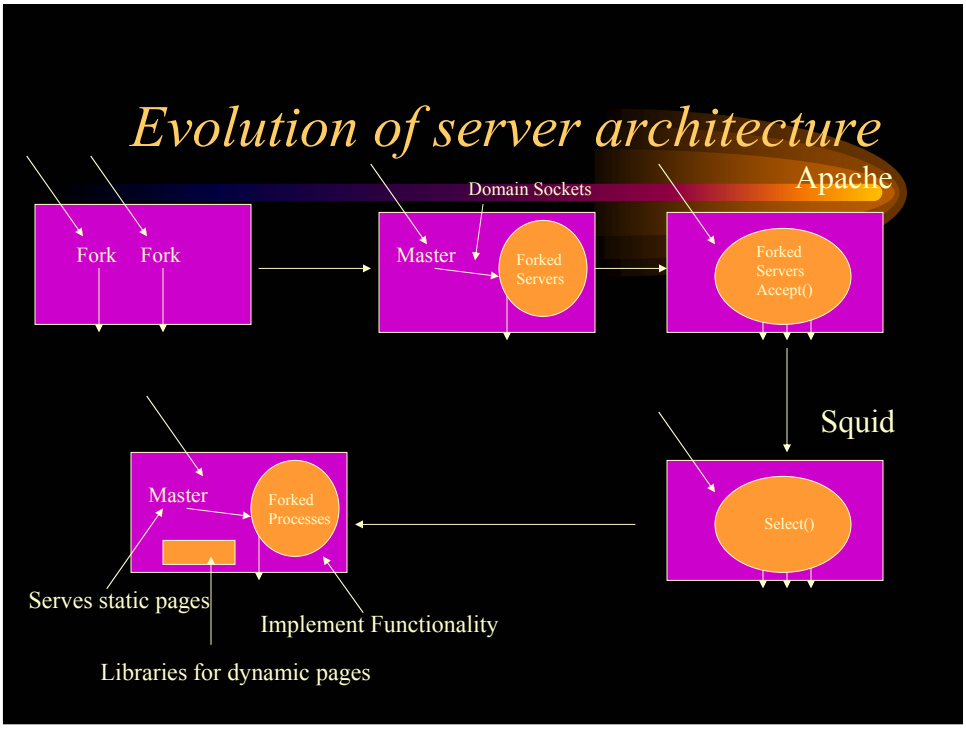
"This problem is serious and warrants modifications to the OS, since they can be sold as a product by itself and doesn't need to be reflected on the clients"

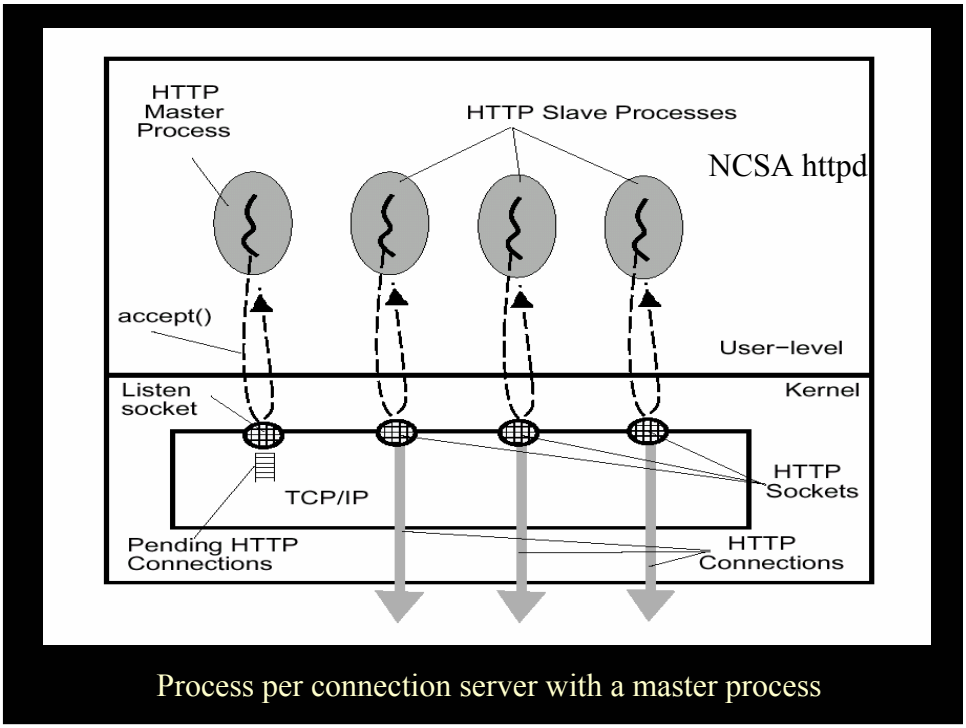
## *Presentation Structure*

- An overview of the problem with a glance at server architectures
- Introduce the radical of a server OS based in exokernel principals. Describe the Cheetah server and explore if exokernels are viable design choices
- Special Topics
  - XN <User Specific Low Level Disk Access>
  - ASH <Application Specific Handlers>
  - ILP <Integrated Layer Processing>
  - RC <Resource Containers and Differentiated QoS>
  - LRP <Lazy Receiver Processing>
  - API Improvements
  - SCF <Connection Scheduling>
  - I/O LITE <Unified Buffering>

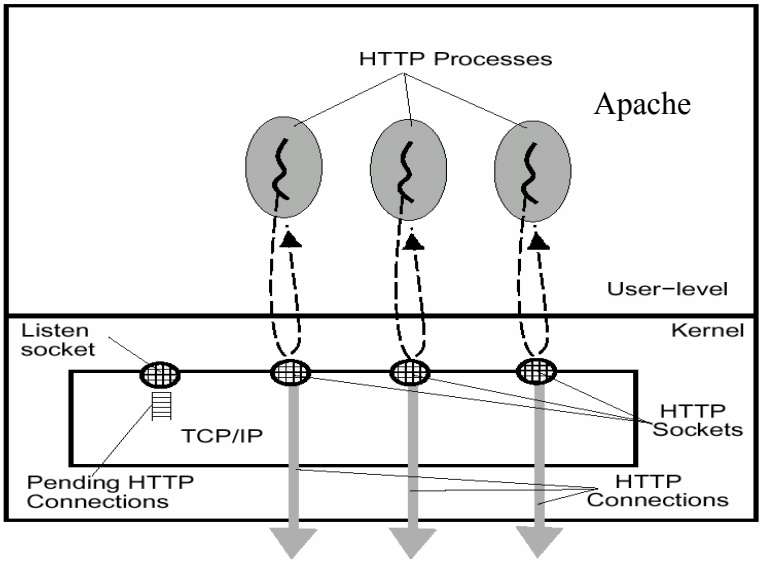
## *Why is this so troublesome?*

- Current OS were designed for timesharing, database or file service where more time was spent in user mode, whereas servers make very frequent kernel calls and access I/O very frequently
- Server Application has no control over system resources. You cant prevent low priority clients from hogging resources
- PCB lookup algorithms and other TCP implementation specific parameters are out of control
- Linear time select() call limits scalability
- TCP is optimized for long connection times but HTTP requests open up frequent short-lived TCP connections
- Under heavy load servers end up in “livelock”
- No prioritization of I/O streams for differentiated QoS

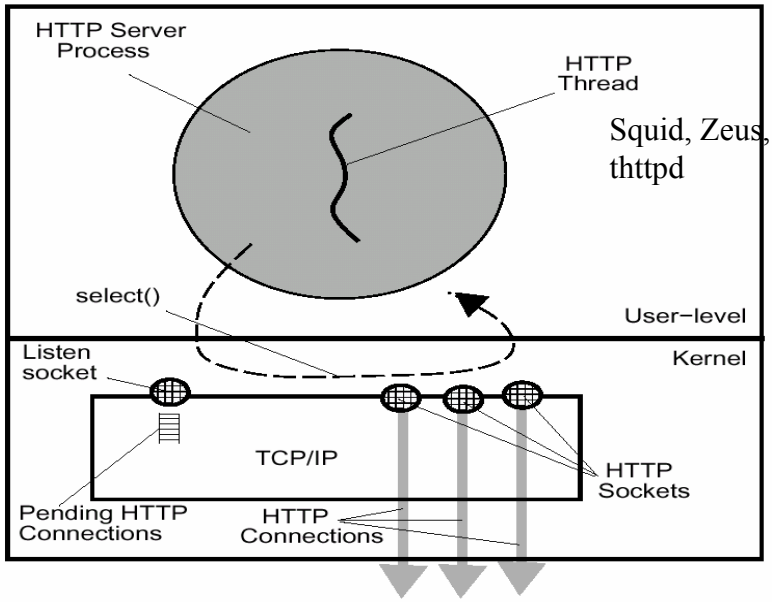


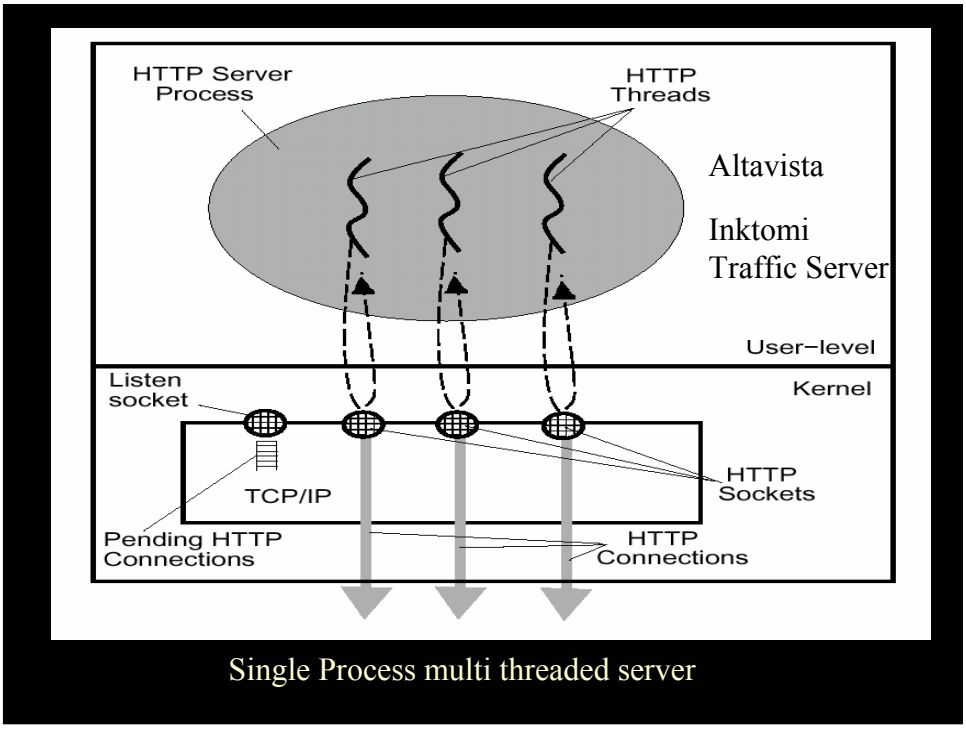

Process per connection server with a master process



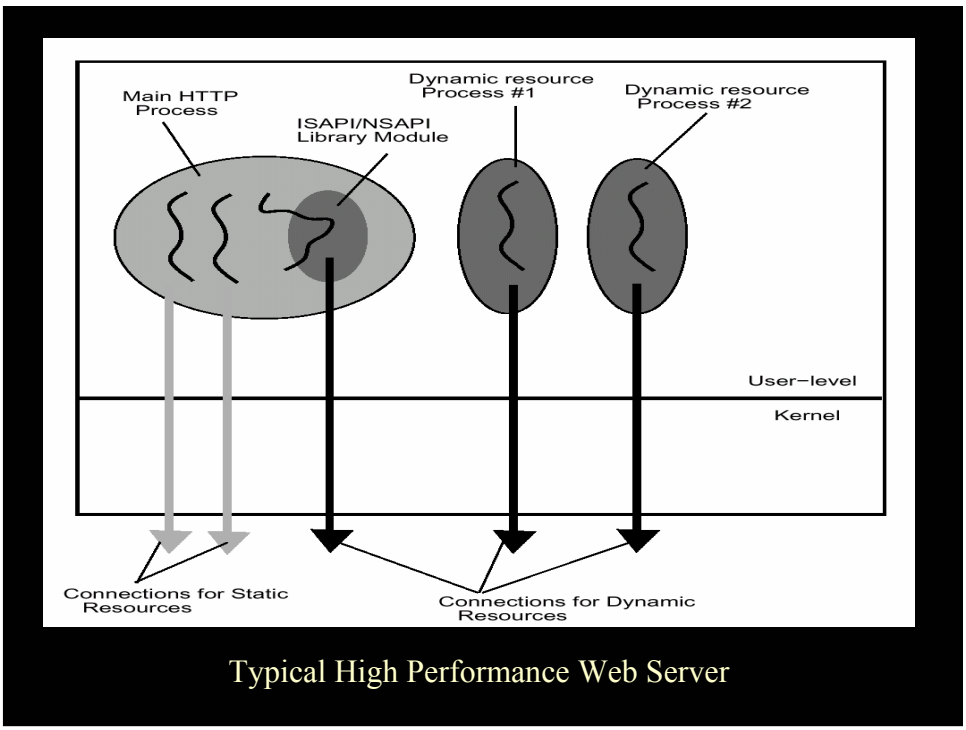

Process per connection server without a master process




Single Process Event Driven Server




Single Process multi threaded server




Typical High Performance Web Server

## *Architecture Specific Problems*

- Event Driven Server <require scalability and efficient event message delivery, single thread of control should not block, asynchronous notification to server of change of state>
  - SIGIO can be used for notification but gives no other information…for ex which descriptor is ready
  - SELECT() scaled poorly with a large number of descriptors
  - Lack of non-blocking I/O
- Multi-Threaded Server
  - Kernel threads require kernel calls for sync and context switch
  - User threads block all other threads in the process if one blocks
  - Kernel must support massively threaded processes
  - Loads of threads cause context switching overhead and high TLB miss rates

## *Approach 1*

- Place a server on top of a general purpose OS
  - Simplifies construction
  - Forces use of overly general OS abstraction
  - High performance in this approach requires very powerful hardware

## *Approach 2*

- Create an OS specifically designed for a single server configuration
  - Different OS constructed for different server types
  - Too much implementation effort
  - No resource multiplexing among servers

## *What to do : SOS !!*

- Set of abstractions and runtime support for high performance servers
  - Tools and standard implementation of abstractions
  - Freedom to specialize server abstractions
  - Protected system access for multiple application timesharing
  - Choice for server applications to access hardware directly
  - Direct access between disk and network removing scheduling delays, file system layers etc
  - Event based support to remove thread concurrency issues
  - Integrated Layer Processing

## *What do you need for this?*

- An “extensible” kernel
- Exokernel OS architecture
- Can be put into UNIX or NT

- Note : keep in mind that is a hotly debated area. Some people feel this is leading OS research in the wrong direction

## *Exokernels*

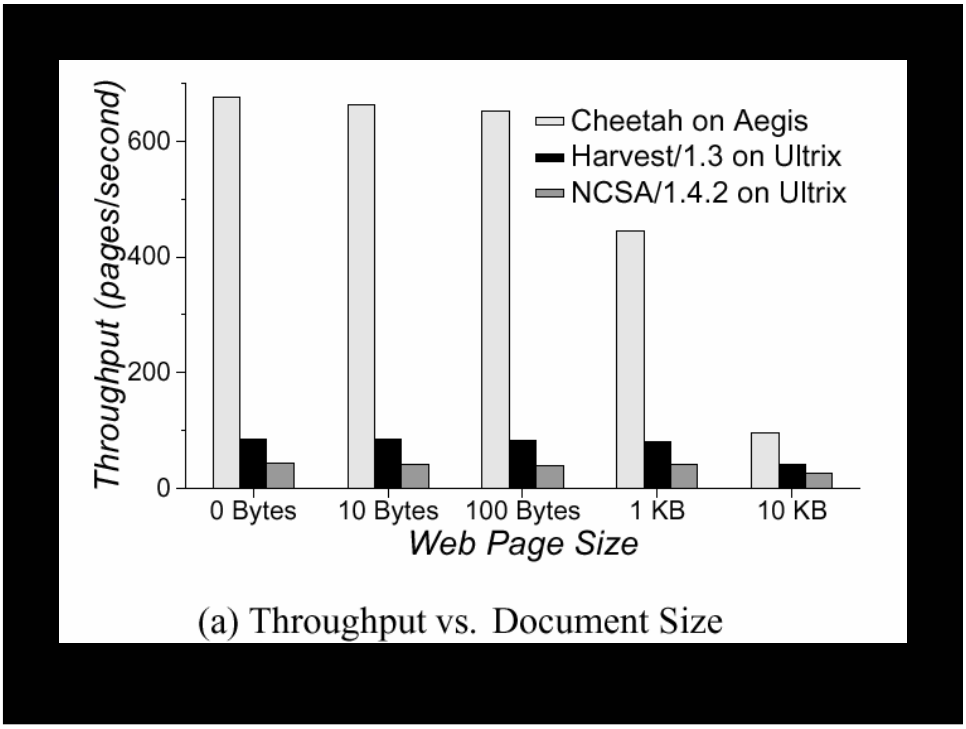
- Expose hardware to applications to the greatest degree possible while maintaining protection between processes
- Exokernel interface seeks to remove abstraction till it reaches the point of protection
- More communication paths between user and kernel space. Presence of kernel initiated communications
- Signal mechanisms are not rich enough or kernel initiated communication. Slow, inflexible, not much information, no queuing.
- Performance improvements seen in Xok show 10-300% for unmodified apps and 800% for modified apps

## *SOS Architecture*

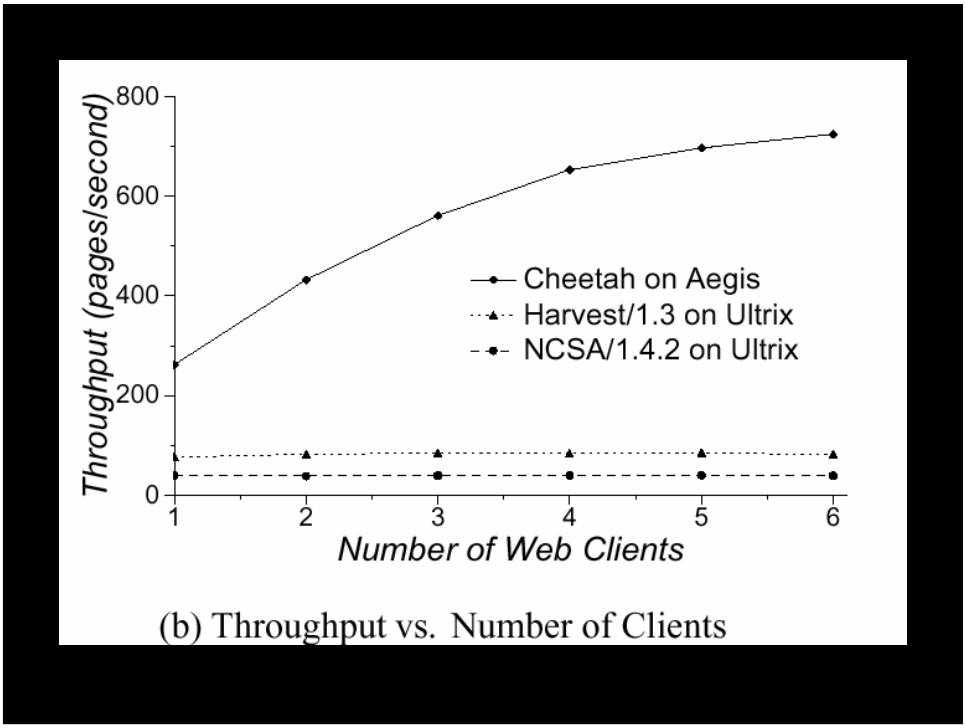
- Specialization
  - Specializing abstractions to applications
- Direct device-to-device access
  - Eliminate scheduling, file system and network layer delays
- Event Driven Organization
  - Non blocking I/O and event driven organization to achieve concurrency without problems
- ILP compiler support
  - Multiple layer processing integration support

## *Implementation Details*

- Server applications manage PCBs
- Specification of last transfers
- Pre-computed checksums
- Highly configurable non-blocking file system
- Combined copy-free disk cache/retransmission pool
- Pre-allocated PCBs to reduce setup latency and tuning for large number of TIME_WAIT states
- Header pre-computation
- Disk allocation by hyperlinks




(a) Throughput vs. Document Size




(b) Throughput vs. Number of Clients

## *So what do we do*

The ambitious approach
This is way cool get me an
Endless supply of pizza and
I'll start on writing BSD again

The lazy approach
I have spent too much
Energy writing BSD
Gimme a break! To hell
With you "exowhatever"

## *Are exokernels viable ?*

Maybe…. They do have several things against them such as complexity, interoperability and protection problems. Maybe the answer lies in applying the aggressive enhancements seen in exokernels to current OSes.

My view is that exokernel performance tells us what is lacking in the general OS designs. They are like warning bells to performance bottlenecks and though a great tool for research may not be particularly viable


Special Topics


XN <Application Specific Disk Access>

## XN

- Allows a server application to implement it's own file system
- Each application may individually manipulate disk blocks without concern that another application may corrupt it
- Users have complete control of allocation patterns and disk structure
- XN provides stable storage at the level of disk blocks by exporting a buffer cache registry

## XN

- XN uses UDF <metadata translation functions>
- UDF allow the kernel to handle any metadata layout without understanding the layout
- UDFs are stored as disk structures called templates <ex UNIX would have templates for data blocks, inodes, indirect blocks etc>
- Each template T has one UDF:owns-UDFt
- For a piece of metadata m of template T owns-UDFt(m) returns the set of blocks which m points to and their template type
- Ex. You want to allocate a disk block b, by placing a pointer to it in metadata m, libFS will call XN with m,b and proposed modifications to m(list of bytes). XN runs own-UDFt(m) makes a copy of m as m' and runs owns-UDFt(m').
- Then verifies if own-UDFt(m)=own-UDFt(m')+b
- Also acl-ufs(m,b,capabilities) is used for protection of data

## *Application Specific Handlers*

## *User Level Communication*

- Problems
  - Communication code is not integrated with OS
  - Very long round trip times if process is not scheduled
  - Even if preemptive scheduling is done it is very costly to be done for every packet that arrives
  - Certain NIC drivers use a limited buffer for copying and the OS needs to make copies anyway if the process is not scheduled

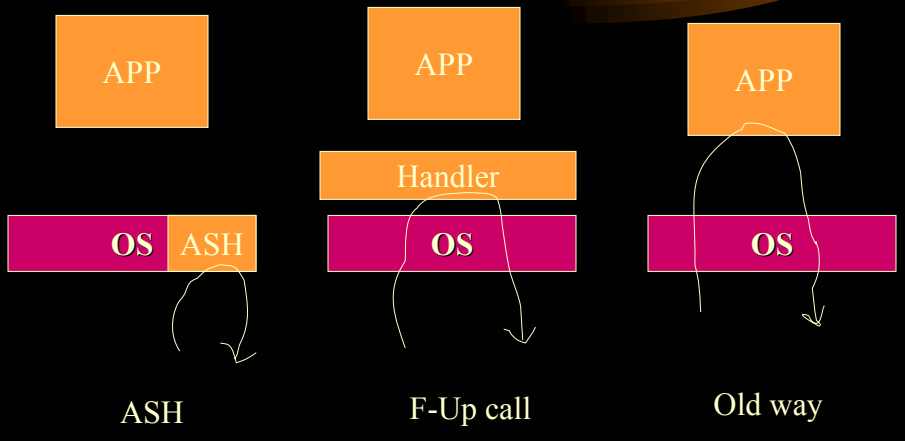
## *User Level Communication*

- Requirements [have the cake and eat it]
  - Performance of in-kernel protocol with flexibility of user-level protocols
- Solution Considerations
  - Control of where to copy a message on arrival
  - Do checksums and byte swaps as well while copying [ILP sort of]
  - General computation
  - Message response by application

## *How do we manage this?*

- Application Specific Handler
  - User code downloaded to the kernel and run in response to a message
- Fast Upcall
  - Handler run in response to the message in user-space . <Asynchronous>

## *The three mechanisms*




## *Application Specific Handlers*

- User written handlers that are safely and efficiently executed in the kernel in response to a message arrival
- Written by application programmers, downloaded in to the kernel, invoked after a message is multiplexed

## *Application Specific Handlers*

- Dynamic control of message copying.
- Message Initiation for low latency message replies
- Perform general computation for control operations at message reception
- Integrated Layer Processing

## *Integrated Layer Processing*

# *Integrated Layer Processing*

- Data manipulation – is one of the costliest aspects of data transfer
- Message data passing from one protocol to another in system data structures requires loading and storing each byte of the message
- Integration combines manipulation from a series of protocols into a pipeline that shares access to the same data structure

# *Integrated Layer Processing*

```
for( i = 0; i < 10000; i++ )
      msgData[i]++;                    /* LOAD, ADD, STORE */

for( i = 0; i < 10000; i++ )
      msgData[i] = ~msgData[i];        /* LOAD, COMPLEMENT, STORE */
```

(a)

```
for( i = 0; i < 10000; i++ ){
      temp = msgData[i];               /* LOAD */
      temp++;                          /* ADD */
      temp = ~temp;                    /* COMPLEMENT */
      msgData[i] = temp;               /* STORE */
}
```

(b)

# *DILP*

- Data manipulations such as checksumming or conversions can be integrated into the data transfer engine itself
- Can be used with ASH or FUP

# *ILP Gains*

TABLE I
SERIAL VERSUS INTEGRATED WHEN DATA IN CACHE

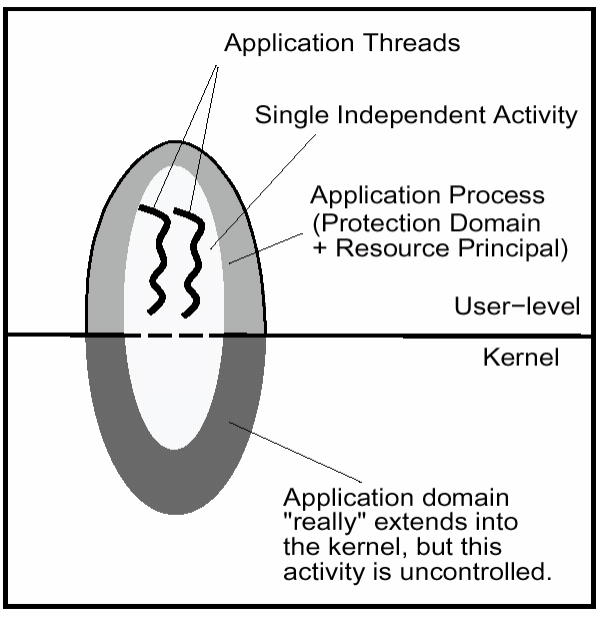
| | DS5000 | | HP720 | | Sparc1 | |
|---|---|---|---|---|---|---|
| | Serial (Mbps) | Integrated (Mbps) | Serial (Mbps) | Integrated (Mbps) | Serial (Mbps) | Integrated (Mbps) |
| CKSUM+BSWAP | 44.7 | 54.4 | 84.2 | 101.0 | 24.0 | 30.0 |
| BSWAP+PES | 33.9 | 43.5 | 72.1 | 101.0 | 17.1 | 25.0 |
| BSWAP+PES+CKSUM | 25.8 | 35.4 | 54.1 | 79.7 | 13.6 | 21.1 |

TABLE II
SERIAL VERSUS INTEGRATED WHEN DATA NOT IN CACHE

| | DS5000 | | HP720 | | Sparc1 | |
|---|---|---|---|---|---|---|
| | Serial (Mbps) | Integrated (Mbps) | Serial (Mbps) | Integrated (Mbps) | Serial (Mbps) | Integrated (Mbps) |
| CKSUM+BSWAP | 29.3 | 39.6 | 42.4 | 55.8 | 19.3 | 26.1 |
| BSWAP+PES | 23.4 | 34.0 | 33.1 | 56.2 | 14.6 | 22.2 |
| BSWAP+PES+CKSUM | 17.5 | 29.0 | 26.2 | 48.9 | 11.3 | 19.0 |

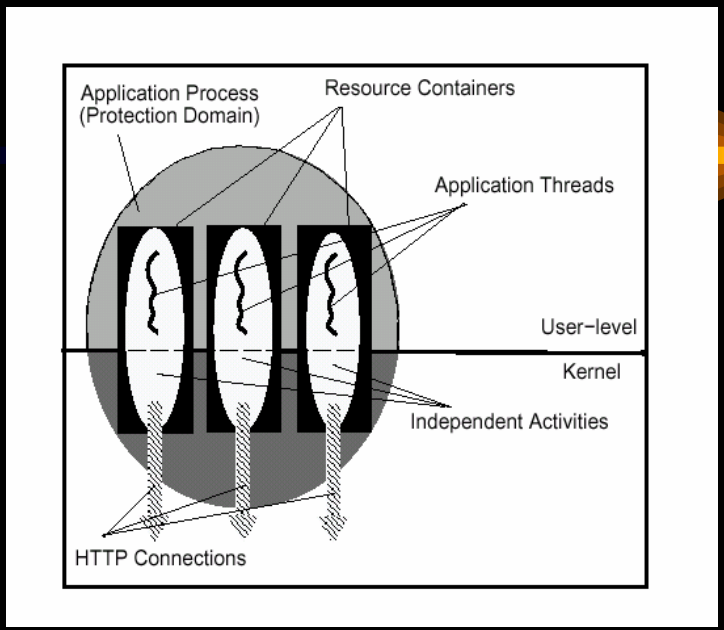
# *Resource Containers & Differentiated QoS*

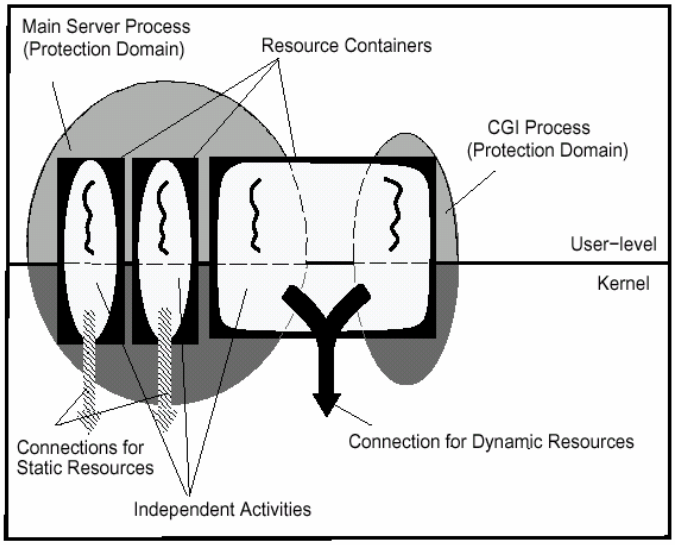
Resource Model In Current OSes



# *Resource Containers*

- A resource container is an abstract OS entity that logically contains all OS resources used for a particular activity
- Containers have attributes used to provide scheduling parameters and QoS
- There is a distinction between protection domains and resource principles
- Each thread has a container binding and multiple threads might have the same binding



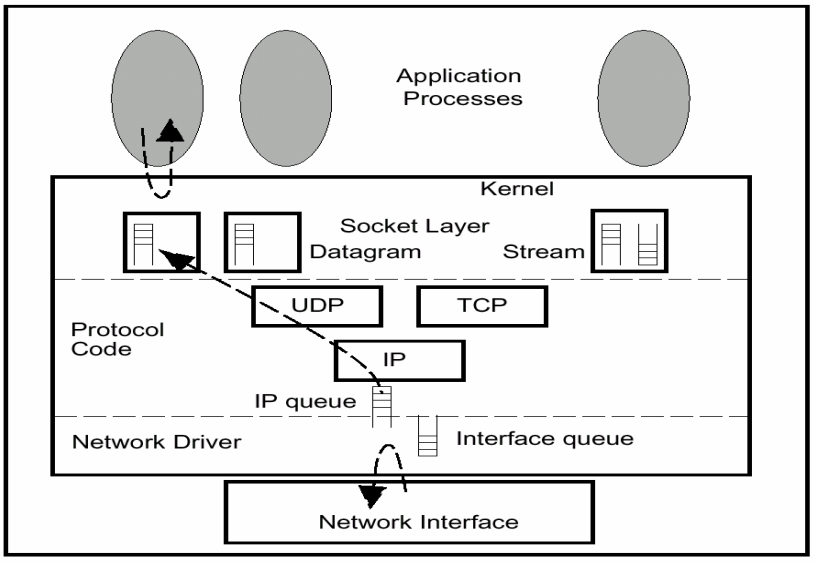



```
struct resource_container {
    /* pure mechanism data */
    int ref_count;
    simple_lock_data_t rc_lock;
    struct resource_container *parent;
    struct rc_list children;
    struct thread_list sched_binding;

    /* resources inside the container */
    struct resource_list socket_list;
    struct resource_list pcb_list;
    struct resource_list ni_channel_list;
    struct resource_list mem_page_list;

    /* scheduler-specific information */
    int rc_cpu_limit;
    int rc_cpu_reserve;
    int rc_cpu_pri;
    int rc_mem_user_pages;
    int rc_max_sockbufspace;
    struct sched_info;

    /* access control */
    int rc_rights_inherit;

    /* statistics */
    struct rc_stats stats;
    int propagate_stats;
};
```

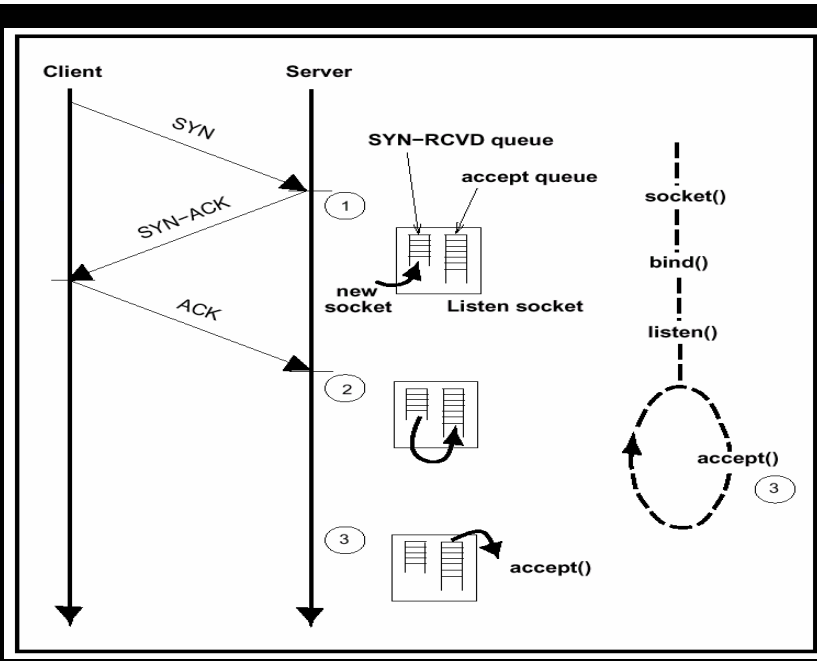
# *Lazy Receiver Processing*

# *Current Model Problems*

- For network intensive applications , the kernel does not control or account for resource consumption accurately
- Execution of software interrupts is done at the cost of the "victim" process
- Network processing occurs not at the priority of the process but at a higher priority than any application
- Crux -> system resources in network processing are generally beyond control of the application
  - Inaccurate accounting
  - Eager beaver processing
  - Ineffective load shedding
  - Lack of traffic separation



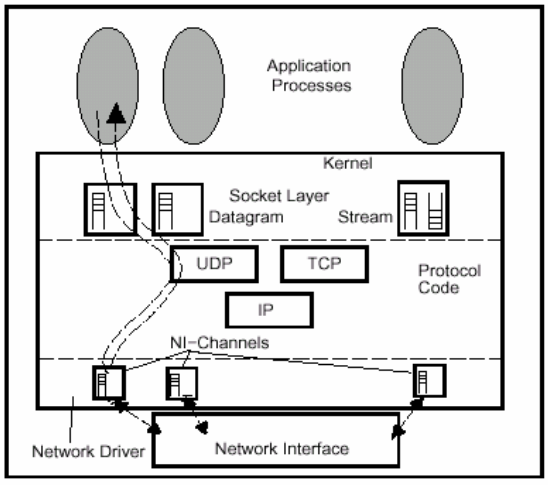

Current Network Processing Model




HTTP Connection Timeline

# *LRP <Lazy Receiver Processing>*

- Per-socket queue is used <NI channel>
- Network interrupt handler de-multiplexes incoming packets according to receive queue. Early packet discard is implemented
- Sender & Receiver protocol processing done at the priority of the resource principal
- Early de-multiplexing done in software or hardware to identify resource principal
- IN FACT THE DPF MECHANISM IS THE MOST EFFICIENT THOUGH NOT USED HERE




```
struct ni_channel {
    struct inpcb *inp;
    struct ifqueue receive_queue;
    simple_lock_data_t ni_channel_lock;
    struct resource_container *owner;
};
```

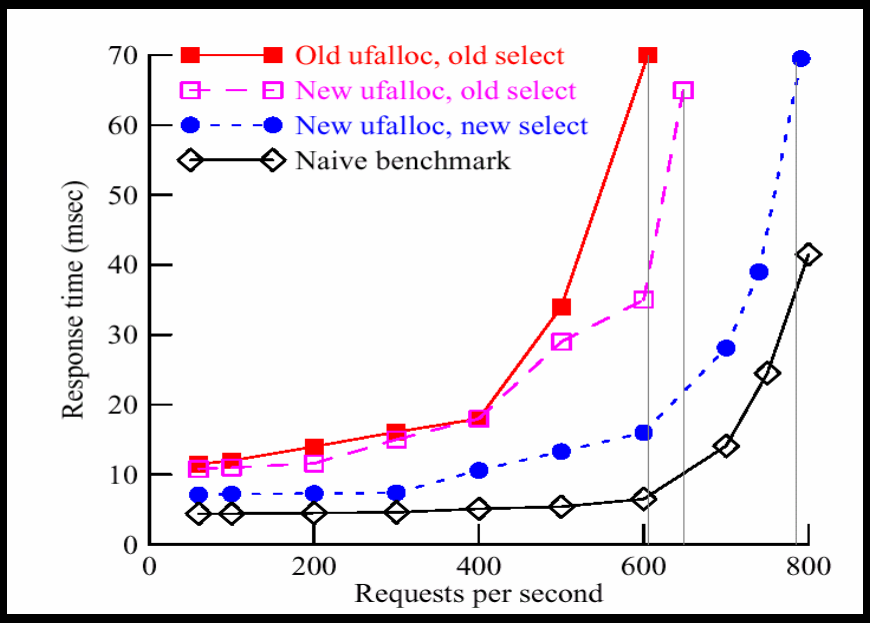
# *API Improvements*

# *The Select() & ufalloc() call*

- Information about descriptors are passed to the kernel using 3 bitmaps
- Problems
  - Each call to select() has costs proportional to number of descriptors
  - No notion of priority for a socket for QoS
  - Busy servers spend upto 60% of their time inside select()
  - Ufalloc() linear search penalty to find lowest numbered descriptor

# Solutions

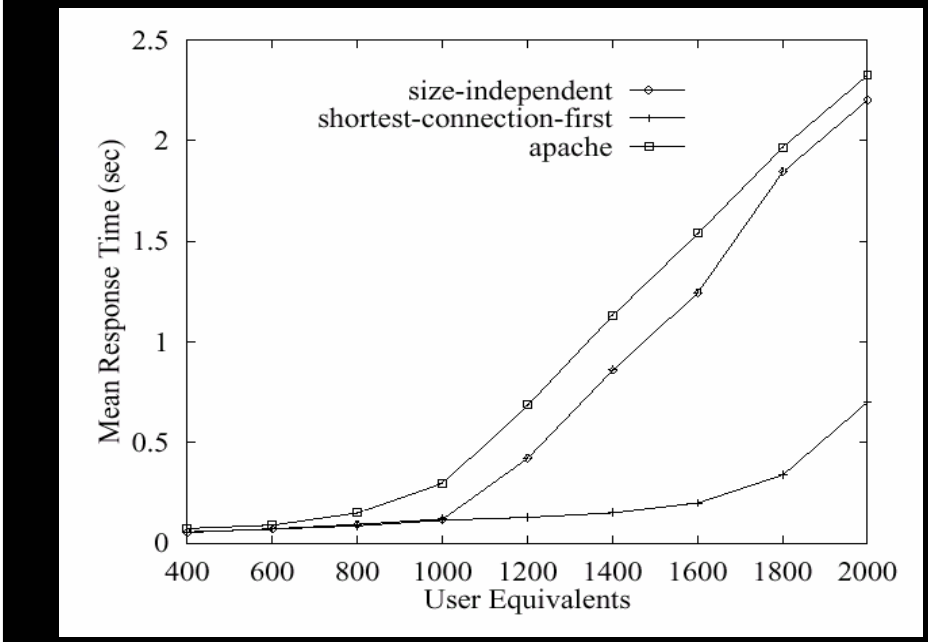
- Select()
  - Preserve information about change in state of a socket between select_wakeup() and do_scan()
  - Inspect only those sockets that need inspection
  - For each thread three sets are tracked – READY, INTERESTED and HINTS <updated by protocol layer>
- ufalloc()
  - Make the search logarithmic time instead by using a two level tree of bitmaps




Squid response 1259 byte files

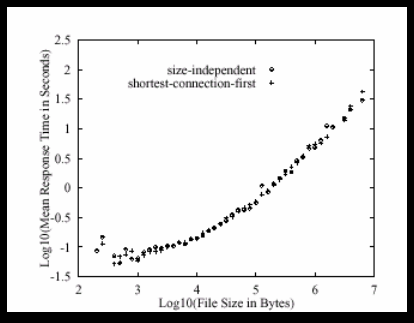

## Connection Scheduling

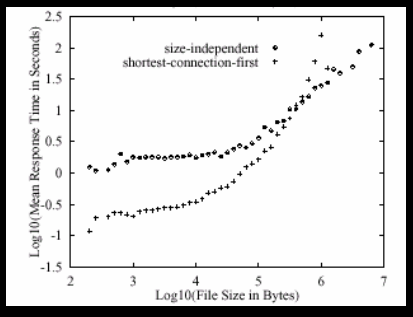
- Ordering of concurrent connection servicing is normally left to the OS
- What effect does SCF have on scheduling? Can servers use knowledge of "size" to improve mean response time?
- No free lunch -> remove fairness
- Conclusions of this work
  - Apache does not favor short connections
  - SCF can improve performance by a factor of 4-5 in moderate loads
  - SCF scheduling does not significantly penalize long-connections !!




Long Jobs don't suffer that much in SCF scheduling

Size of client base



# *Unified Buffering*

# *I/O Lite – the unified buffer cache*

- Allows applications, inter-process communication , file systems and disk cache to use the same buffer
- Requirements : single physical copy, concurrent access, preserved storage, identification of previously seen data
- Benefits Application independent, No redundant copying, Multiple buffering, Cross system optimization

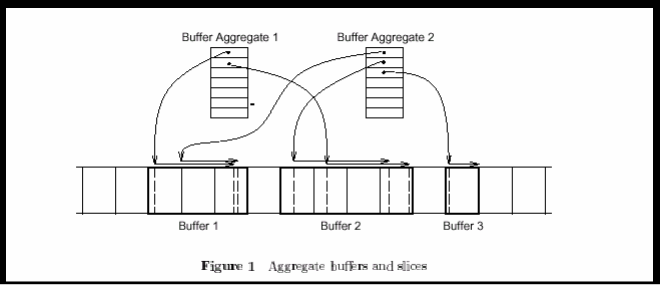



**Figure 1** Aggregate buffers and slices

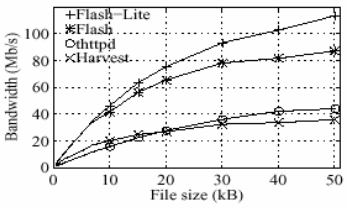



**Figure 4** HTTP transfer speed

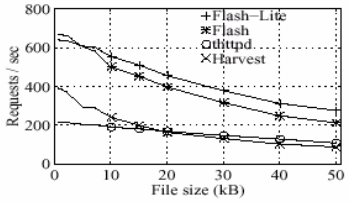



**Figure 5** HTTP request rate

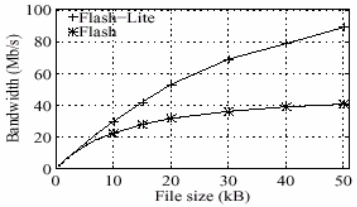



**Figure 8** CGI transfer speed

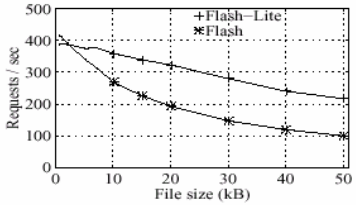



**Figure 9** CGI request rate

# *The Final Solution???*

An overall approach to a great web-server needs to take into account a lot of considerations. I would envision ->

- Performance conscious design like the Flash webserver
- A unified buffering mechanism like I/O Lite
- The resource container model is definitely needed to provide QoS
- To provide QoS we also need LRP which can have early de-multiplexing based in hardware
- A combined SCF / Priority weighted Connection scheduling server
- Integrated Layer Processing capable network code